



Analytisches Feature Reporting mit integrierter Versionskontrolle auf Basis von Metadaten

Projektarbeit 1 (T3_1000)

Im Rahmen der Prüfung:

Bachelor of Science (B. Sc.)

des Studienganges Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Robin van Nuis

Abgabedatum 31. Dezember 2025

Bearbeitungszeitraum 20.10.2025 - 31.12.2025

Matrikelnummer, Kurs 8771293, TINF25B2

Ausbildungsfirma SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Deutschland

Betreuer der Ausbildungsfirma Kolja Groß

Gutachter der Dualen Hochschule Prof. Dr. Sebastian Ritterbusch

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit 1 (T3_1000) mit dem Thema:

Analytisches Feature Reporting mit integrierter Versionskontrolle auf Basis von Metadaten

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, 11. Dezember 2025

gez.: van Nuis, Robin

Sperrvermerk

Die nachfolgende Arbeit enthält vertrauliche Daten der:

SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf
Deutschland

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung vom Dualen Partner vorliegt.

ABSTRACT

- Deutsch -

Dies ist eine kurze Zusammenfassung, um die Formatierung und den allgemeinen Stil der Vorlage zu zeigen

Es ist möglich, mehrere Abstracts in Unterschiedlichen Sprachen zu haben

ABSTRACT

- English -

This is a short abstract to show the formatting and general style of the template It is possible to have multiple abstracts in different languages

Inhaltsverzeichnis

Abbildungsverzeichnis	VII
Quellcodeverzeichnis	VIII
1 Einleitung	9
1.1 Motivation	9
1.2 Zielsetzung	9
2 Grundlagen	11
2.1 ABAP	11
2.2 SQL	12
2.3 CDS	12
2.3.1 CDS Views	12
2.3.2 CDS Annotationen	13
2.4 Metadatentabelle	14
2.5 Star-Schema	14
2.6 Queries	15
2.7 OLAP Würfel	16
2.8 Assoziation	16
2.9 Kardinalitäten	17
3 Bisheriger Stand	18
4 Implementierung	19
4.1 Feature Reporting	19
4.1.1 Sammeln der historischen Daten	19
4.1.2 Speichern der Daten in eigenen Tabellen	20
4.2 Datenanalyse	23
4.2.1 Aufbau des Datenmodell	23
4.2.2 OLAP-Würfel erstellen	24
4.2.3 Queries definieren	25
4.3 Auswertungen	26
5 Fazit	27
6 Advanced Elements	28
6.1 Figures	28
6.1.1 Image Figures	28
6.1.2 Code Snippets:	28
6.2 Math	29
6.3 Block Quotes	30
6.4 Notes	30
7 References and Citations	31
7.1 Local Elements	31
7.4 Code Blocks	31
Literaturverzeichnis	a

Abkürzungsverzeichnis

ABAP – Advanced Business (and) Application Program 11, 19

CDS – Core-Data-Service 12, 13

GB – Giga-Byte 18

MB – Mega-Byte 18

SQL – Standard Query Language 12

UUID – Universally Unique Identifier 22

Abbildungsverzeichnis

Abbildung 1 SAP Logo	28
----------------------------	----

Quellcodeverzeichnis

Code 1	Annotationsstruktur	13
Code 2	Beispiel einer Annotation	14
Code 3	Beispiel Query	15
Code 4	Beispiel Assoziation	17
Code 5	Mitgliederliste füllen	19
Code 6	Objekttypen speichern	19
Code 7	Versionstabelle	21
Code 8	Bereits existierende Tabelle	21
Code 9	Meine zugehörige Tabelle	21
Code 10	UUID erstellen	22
Code 11	Neue Version erstellen	22
Code 12	Daten selektieren	23
Code 13	Anbindung der Headertabelle für 'DRAS'	24
Code 14	My Code	29
Code 15	My Code from a file	29
Code 16	Code w/ Ref	31

1 Einleitung

1.1 Motivation

In einer zunehmend datengetriebenen Unternehmenslandschaft gewinnen analytische Auswertungen und Berichte durchgehend an Bedeutung. Entscheidungen darüber welche Features oder Software entwickelt werden soll, basieren zumeist noch auf groben Schätzungen. Diese Entscheidungen sollen jedoch fundiert auf Daten und Erfahrungen entstehen, wodurch man sowohl Zeit als auch Ressourcen sparen möchte.

Eine zentrale Möglichkeit, diese Datenlücke zu schließen, besteht darin, die Nutzung verschiedener Features systematisch zu erfassen und deren Entwicklung über Zeit zu analysieren. Durch historische Auswertungen lässt sich ableiten, welche Features tatsächlich Mehrwert schaffen, wo Optimierungsbedarf besteht und welche neuen Funktionalitäten entwickelt werden sollten.

Daraus ergibt sich die zentrale Fragestellung:

Wie können wir Metadaten so aufbereiten, dass historische Analysen zur Feature-Nutzung, aussagekräftige Erkenntnisse für datenbasierte Entscheidungen im Hinblick auf Entwicklungen liefern?

Dieser Prozess nennt sich analytisches Feature Reporting und orientiert sich methodisch an den vier Hauptpunkten der SAP-Discovery-Strategie:

1. Wer ist der Enduser und welche Aufgaben/Verantwortungen hat er?
2. Welches Problem ist zu lösen und wie bringt dies den User weiter?
3. Welche Funktionen gilt es zu bündeln und was ist die Priorität für SAP und den User?
4. Welche Strategien sind am effektivsten für die Umsetzung?

1.2 Zielsetzung

Das zentrale Ziel meines Projektes ist die Entwicklung eines Systems für ein analytisches Feature Reporting, welches durch Metadaten-basierte Versionskontrolle die historische Entwicklung der Feature-Nutzung transparent macht und somit datenbasierte Entscheidungen ermöglicht.

Um dieses Hauptziel zu erreichen, implementiere ich eine automatisierte Datenspeicherung mit Versionskontrolle, die in regelmäßigen Intervallen festgelegte Daten aus Quelltabellen ausliest und in dedizierte Archivtabellen überführt, wobei jede Auslesung eine neue, eindeutig identifizierbare Version erstellt. Diese Tabellen verbinde ich zu einem sogenannten Star-Schema, mit welchem die Daten nun deutlich übersichtlicher und besser zu analysieren sind.

Die Abteilung, in welcher ich dieses Projekt durchführe, ist für die Erstellung und Entwicklung von Objekten und Features in den ABAP CDS zuständig. Daher konzentriere ich mich in meinem Projekt auf die Metadaten dieser Objekte und Features, wie zum Beispiel, die Anzahl der Views, die ein Objekt besitzt, oder die Anzahl der implementierten Features in einem bestimmten Zeitraum.

Auf diesen Metadaten aufbauend entwickle ich ein Query-Framework, mit dem man die gespeicherten historischen Daten über vordefinierte Abfragen auslesen kann. Dadurch sollen die Daten, sowohl historisch, als auch über verschiedene Versionen hinweg analysiert und verarbeitet werden können.

Diese Analysen sollen dabei helfen datenbasiert zu entscheiden, welche Features und Objekte weiter entwickelt werden sollen und welche eher weniger Aufmerksamkeit benötigen und nur noch bedingt gewartet werden müssen.

2 Grundlagen

2.1 ABAP

Das *Advanced Business (and) Application Program (ABAP)* ist eine von SAP SE entwickelte Objektorientierte Programmiersprache, die speziell für betriebswirtschaftliche Anwendungen konzipiert wurde. Ursprünglich stand das Akronym für „Allgemeiner Berichts und Aufbereitungs-Prozessor“, wurde jedoch erst 1996 zu „Advanced Business (and) Application Programming“ umbenannt, um den erweiterten Funktionsumfang der Sprache zu reflektieren. [1]

Die Sprache wurde in den 1980er Jahren entwickelt und ist direkt mit dem SAP-System verbunden. *ABAP* ermöglicht es Entwicklern, komplexe betriebswirtschaftliche Anwendungen zu erstellen, die nahtlos in die SAP-Umgebung integriert sind. Als objektorientierte Programmiersprache bietet *ABAP* moderne Programmierkonzepte wie Klassen, Vererbung und Polymorphismus. [2]

Ein besonderes Merkmal von *ABAP* ist die enge Integration mit Datenbanksystemen. Die Sprache verfügt über eingebaute zu SQL ähnliche Befehle (*ABAP SQL* genannt), die eine effiziente Datenmanipulation ermöglichen. Dies macht *ABAP* besonders geeignet für die Entwicklung von Anwendungen, die große Mengen an Datensätzen verarbeiten müssen. *ABAP* wird hauptsächlich in SAP R/3 und SAP S/4HANA Systemen eingesetzt und ermöglicht die Entwicklung verschiedener Anwendungstypen, darunter Reports, Dialoge, Formulare und Web-Services. Die Sprache unterstützt sowohl prozedurale als auch objektorientierte Programmierung und bietet umfangreiche Bibliotheken für typische Geschäftsanwendungen wie Finanzwesen, Personalwesen und Logistik. [1]

Ein wichtiger Vorteil von *ABAP* ist die plattformunabhängige Ausführung innerhalb der SAP-Umgebung. Programme werden in einem eigenen virtuellen System, dem *ABAP Application Server*, ausgeführt, was Portabilität zwischen verschiedenen Betriebssystemen und Datenbanken gewährleistet. Darüber hinaus bietet die Entwicklungsumgebung „*ABAPDevelopment-Tools*“ integrierte Werkzeuge für Debugging, Performance-Analyse und automatisierte Tests, die die Entwicklung und Wartung von Anwendungen erheblich erleichtern. [3]

2.2 SQL

Die *Standard Query Language (SQL)* ist die weltweit am besten etablierte und populärste Sprache für Datenabfragen und ist aus der modernen Datenverarbeitung nicht mehr wegzudenken. Die Sprache bietet umfassende Funktionen zum Erstellen, Manipulieren und Verknüpfen von Datenbanken und ermöglicht sowohl einfache Abfragen, als auch komplexe Datenoperationen. Anders als vollständige Programmiersprachen wird *SQL* typischerweise als eingebettete Abfragesprache in andere Sprachen wie Python, Java oder C# integriert. Dies ermöglicht Entwicklern, die Stärken beider Welten zu kombinieren: Die logische Programmstruktur der Ausgangssprache und die spezialisierte Datenbankmanipulation durch *SQL*. Die meisten Datenbanksysteme, von MySQL bis hin zu PostgreSQL, orientieren sich an einheitlichen *SQL*-Standards. Diese Standardisierung erhöht nicht nur die Kompatibilität zwischen verschiedenen Systemen, sondern erleichtert auch den Wissenstransfer für Entwickler. [4]

2.3 CDS

Core-Data-Services (CDSs) stellen die zentrale Infrastruktur zur semantischen Datenmodellierung im SAP-System dar. Sie ermöglichen es, Datenmodelle direkt auf der Datenbankebene zu definieren und dadurch Daten effizient, sicher und performant bereitzustellen. Ein zentrales Element der *CDS* sind die sogenannten *CDS Views*, mit denen sich Daten aus unterschiedlichen Tabellen logisch zusammenfassen, anreichern und für verschiedene Anwendungsszenarien nutzbar machen lassen.

Im Folgenden werde ich näher auf verschiedene *CDSs* eingehen und wie diese für mein Projekt relevant sind.

2.3.1 CDS Views

CDS Views (Core Data Services Views) sind ein zentraler Bestandteil der *CDS* und dienen zur semantischen und logischen Modellierung von Daten im SAP-System. Es handelt sich um virtuelle Datenmodelle, das heißt, die Daten werden nicht physisch gespeichert, sondern nur aus den zugrundeliegenden Datenbanktabellen gelesen.

CDS Views können sowohl im SAP HANA Produkt, als auch im ABAP-System definiert werden. Ein großer Vorteil ist, dass sie Tabellenbeziehungen und Verknüpfungen direkt in der Definition abbilden, sodass keine manuelle Verknüpfung der Tabellen in Programmen mehr nötig ist.

Ein weiterer wichtiger Aspekt ist die Wiederverwendbarkeit: Ein einmal definierter CDS View kann in mehreren Anwendungen, wie zum Beispiel Fiori-Apps, OData Services, analytischen Berichten oder anderen CDS Views, verwendet werden.

Darüber hinaus bieten CDS Views die Möglichkeit, Annotationen zu nutzen, um zum Beispiel Sicherheitsregeln, UI-Eigenschaften, analytische Kennzahlen oder ODataServices zu definieren. Dadurch stellen sie eine Brücke zwischen der Datenbankebene und der Anwendungsebene dar und bilden die Grundlage für viele moderne S/4HANA-Anwendungen.

2.3.2 CDS Annotationen

Annotationen an sich sind Anmerkungen oder Vermerke und beim Programmieren werden sie unter anderem als Strukturierungsmittel verwendet. In den CDS werden Annotationen benutzt um CDS Views mit Metadaten anzureichern. Außerdem dienen sie zur semantischen Beschreibung von Entitäten, als Ersatz für Eigenschaften aus der alten SAP GUI und geben Informationen über aufsetzende Entwicklungsframeworks.

Es gibt zwei verschiedenen Typen von Annotationen, man differenziert zwischen ViewAnnotationen und Element-Annotationen. View-Annotationen sind Vermerke auf ein gesamtes View, wohingegen sich Element-Annotationen nur auf ein einzelnes Element beziehen.

Der Aufbau wie Annotationen im Code angegeben sind folgt einer bestimmten Syntax und wird immer durch ein '@' am Anfang als Annotation gekennzeichnet. Danach kommt der jeweilige Namespace (z.B. UI) und der Name der Annotation (z.B. selectionField). Die einzelnen Elemente werden immer durch einen Punkt getrennt und die Werte sind nach einem Doppelpunkt anzugeben. Zeichenketten werden in einfache Anführungszeichen gesetzt, während sogenannte Enumerationen mit einem Hash-Zeichen gekennzeichnet sind, zum Beispiel #STANDARD. [5]

Die Struktur eine Annotation lässt sich konkret wie folgt darstellen:

```
1 @<Namespace>.<Name>: <Wert>
```

Code 1: Annotationsstruktur

Ein Beispiel für eine Annotation könnte so aussehen:

```
1 @UI.selectionField: [{ position: 10 }]
```

Code 2: Beispiel einer Annotation

2.3.2.1 Enumeration

Enumerationen in CDS definieren einen eigenen, stark typisierten Datentyp mit einem festgelegten Satz zulässiger Werte. Eine Enumeration besteht aus einer Liste benannter Konstanten, die jeweils einem technischen Wert zugeordnet sind. Felder, die diesen Enumerations-Typ verwenden, dürfen ausschließlich diese definierten Werte annehmen.

Dies erhöht die Qualität der Daten und erleichtert Validierungen, da nur gültige, klar benannte Werte gespeichert werden können. Sie werden typischerweise in Statusfeldern, Kategorien oder Klassifikationen benutzt und werden mit der folgenden Syntax referenziert: #BEISPIELENUMERATION

2.4 Metadatentabelle

Eine Metadatentabelle ist eine Datenstruktur, die Metadaten - also sozusagen „Daten über Daten“ - speichern. In ihr wird unter anderem die Herkunft, Bedeutung und Art von verschiedensten Daten gespeichert. Sie ist dafür da eine nachvollziehbare Beschreibung und Steuerung von Datenmodellen bereitzustellen, sodass man diese interpretieren, wiederverwenden, oder automatisieren kann. Außerdem kann man mithilfe von Metadatentabellen genau nachvollziehen was die Daten machen und wozu sie benutzt werden.

2.5 Star-Schema

Ein Star-Schema ist ein mehrdimensionales Datenmodell, in dem Daten in einer Datenbank so verknüpft und organisiert werden können, dass sie besser zu verstehen und analysieren sind. Man kann Star-Schemas auf viele verschiedene Datenmodelle, wie „Data Warehouses“, Datenbanken oder „Data Marts“ anwenden. Sie eignen sich außerdem sehr gut zur Verarbeitung von großen Datenmengen, da sie speziell auf diese optimiert sind.

In der Mitte eines Star-Schemas findet man eine einzelne „Faktentabelle“, die verschiedene Daten aus den anderen Tabellen enthält. Sie ist mit den anderen Tabellen, durch verschiedenste Fremdschlüssel, wie zum Beispiel Zeitpunkte oder andere „Dimensionen“, verknüpft. Dadurch kann es dazu kommen, dass Daten denormalisiert werden, das heißt einige Tabellen

werden mit redundanten Spalten erweitert. Auch wenn dies ein wenig kontraproduktiv wirkt, hilft es dabei die Abfragen und Bearbeitungszeit zu verkürzen. Außerdem sind Star-Schema nicht so sehr auf Joins zwischen den einzelnen Tabellen angewiesen und eignen sich somit besser für einfache Datenabfragen. Die Struktur von Star-Schemata ist des Weiteren sehr leicht zu verstehen und ermöglichen dem Enduser ein einfaches Auffinden benötigter Daten.

2.6 Queries

In der Datenanalyse werden sogenannte Queries (Singular: Query) genutzt, um Daten anhand bestimmter Kriterien auszulesen. Sie erstellen Listenausgaben von Datenbanken gruppiert auf Daten wie zum Beispiel Zeit- oder Preisspannen. Darüber hinaus kann man Queries gut benutzen, um Trends, Muster oder Änderungen in den Daten zu erkennen. Innerhalb von SAP können Queries auch ohne besondere Programmierkenntnisse erstellt werden und lassen sich für viele verschiedene Arten von Berichten wie Grundlisten, Statistiken oder Ranglisten nutzen. Man kann sie außerdem in andere Systeme exportieren oder aus diesen importieren. Queries sind ein Teil der CDS und werden als „transient-view-entity“ definiert. [6]

Die Definition eines Queries kann zum Beispiel so aussehen:

```
1 EndUserText.label: 'Example Query'
2 @AccessControl.authorizationCheck: #NOT_ALLOWED
3 define transient view entity example_query
4   provider contract analytical_query
5   as projection on example_view
6 {
7   @Enduser.label: 'Field1 caption'
8   Field1,
9   Field2,
10  @AnalyticsDetails.query.axis: #COLUMNS
11  Field3
12 }
```

Code 3: Beispiel Query

Die Felder die in einer Query angegeben sind, müssen vorher auch innerhalb der View implementiert worden sein. Wenn ein Feld innerhalb des Views als „Measure“ (Kennzahl) definiert wurde, kann dies in der graphischen Oberfläche unter einer gesonderten Auswahl „Measures“ ausgewählt werden. Dadurch lassen sich zum Beispiel mehrere Zähler auf einer Achse der

Tabelle anzeigen und miteinander vergleichen. Diese Kennzahlen lassen sich ganz einfach in der Query Definition erstellen, ohne die darunterliegende Queries verändern zu müssen. Als Gegenstück zu den Kennzahlen gibt es Dimensionen, in denen sich die Attribute anzeigen und gruppieren lassen. Die Dimensionen lassen sich unterschiedlich anordnen und zeigen so die einzelnen Kennzahlen in Abhängigkeit von verschiedenen Faktoren.

2.7 OLAP Würfel

Ein OLAP Würfel auch Feature-Cube genannt, ist ein logisches Datenmodell zur Darstellung von Daten. Im SAP Kontext ähnelt es vom Aufbau und der Definition her einer CDS View. In ihm werden Daten wie Elemente in einem mehrdimensionalen Würfel angeordnet. Die einzelnen Dimensionen erlauben einen geordneten Zugriff auf die Daten und erlauben für Operationen wie zum Beispiel „Slicing“ (dem Ausschneiden von Scheiben aus dem Würfel), „Dicing“ (dem Erstellen eines kleineren Würfels durch Teileinschränkungen auf einer oder mehreren Dimensionen), „Drill-Down“ (dem „Hereinzoomen“ an detailliertere Werte, oder „Roll-Up“ (dem Gegenteil des „Drill-Downs“, es wird „herausgezoomt“ um auf eine höhere Hierarchiestufe zu verdichten).

Es gibt noch viele weitere Operationen die häufig in Cubes benutzt werden, die für mein Projekt jedoch nicht relevant sind.

2.8 Assoziation

Eine Assoziation (zu Englisch: association) ist eine Möglichkeit in ABAP CDS Tabellen anhand von gemeinsamen Feldern zu verknüpfen. Sie ähnelt im Kern einer „Join“-Verknüpfung und kann verschiedene Kardinalitäten haben. Man definiert eine Assoziation, indem man eine Zieltabelle, eine Kardinalität und Felder angibt, anhand von denen die Tabellen verknüpft werden sollen. Dadurch wird den einzelnen Zeilen in Tabelle A eine bestimmte Anzahl aus Wert von Tabelle B zugeordnet. Im Normalfall entspricht eine Assoziation einem „left-outer-join“, das heißt es werden alle Zeilen aus Tabelle A und die Zeilen aus Tabelle B, die den Fremdschlüssel aus A implementieren, zurückgegeben. Falls eine Zeile aus Tabelle B keinen übereinstimmenden Wert für einen Datensatz aus Tabelle A hat, werden die Spalten aus Tabelle B auf „null“ gesetzt. Angegeben wird eine Assoziation nach folgendem Schema:

```
1 association [Kardinalität] to Zieltabelle as Alias
2   on Quelle.Feld1 = Ziel.Feld2
3   and Quelle.Feld3 = Ziel.Feld4
```

Code 4: Beispiel Assoziation

2.9 Kardinalitäten

Eine Kardinalität gibt an wie Fremdschlüssel die Zeilen zwischen Tabellen verknüpfen und kann die Formen eins-zu-eins (1 : 1), eins-zu-viele (1 : n) oder viele-zu-viele (n : m) haben. Durch Kardinalitäten können effizienter Abfragen auf Datenbanken erstellt werden.

Eins-zu-eins (1 : 1): Eine Zeile in Tabelle A kann mit maximal eine Zeile aus Tabelle B verknüpft werden und anders herum.

Eins-zu-viele (1 : n): Eine Zeile in Tabelle A kann mit beliebig vielen Zeilen aus Tabelle B verknüpft werden. Eine Zeile in Tabelle B kann jedoch nur mit genau einer Zeile aus Tabelle A verknüpft werden.

Viele-zu-viele (n : m): Eine Zeile in Tabelle A kann mit beliebig vielen Zeilen aus Tabelle B verknüpft werden und anders herum.

In CDS-Views gibt es zwei verschiedene Syntax um Kardinalitäten anzugeben.

Bei der ersten wird nur die „Zielkardinalität“ angegeben und gibt einen minimalen und einen maximalen Wert an, die diese annehmen kann. Eine eins-zu-eins Kardinalität kann dort mehrere Formen annehmen und sieht entweder so [1..1] oder so [1] aus. [1..1] steht in dem Falle dafür, dass jeder Zeile von Tabelle A mindestens eine, aber auch maximal eine Zeile von Tabelle B zugewiesen werden kann. Eine eins-zu-viele Beziehung würde in CDS entweder so [1..*] oder so [*] aussehen. Eine Kardinalität anzugeben ist optional und kann ausgelassen werden. Der Standardwert in so einem Fall ist (1 : n).

Bei der zweiten (neueren) Syntax werden sowohl die Quell- als auch die Zielkardinalität angegeben. Diese Syntax sieht dann zum Beispiel so aus: „OF EXACT ONE TO EXACT ONE“ für eine eins-zu-eins Beziehung, „OF EXACT ONE TO MANY“ für eine eins-zu-viele Beziehung und „OF MANY TO MANY“ für eine viele-zu-viele Beziehung. [7]

3 Bisheriger Stand

Grundlage des Projektes ist sind die verschiedenen - bereits existierenden - Tabellen im internen System. Aus diesen Metadatentabellen stammen die Daten, die es gilt zu verarbeiten und zu analysieren. Des weiteren existieren bereits zwei CDS-Views, die die Daten analysieren. Diese beziehen sich jedoch nur auf Laufzeitanalysen und nicht auf die - für mein Projekt benötigten - Veränderungen jener Daten und sind somit nicht relevant für meine Arbeit.

Die Tabellen auf die ich im Folgenden zugreifen werde speichern Informationen über verschiedene Objekte / Teilobjekte und die Beziehungen zwischen den Objekten. Es gibt verschiedene Arten von Tabellen, wovon die meisten unterschiedlich aufgebaut sind und verschiedenste Aufgaben haben. Daher muss ich für jede Tabelle verschiedene eigene Tabellen zum Speichern der Daten erstellen. Auch die Selektion der Daten muss für jede Tabelle leicht angepasst werden.

Das Problem bei den Daten liegt darin, dass alle bereits existierenden Tabellen nur Momentaufnahmen speichern. Das heißt, ich kann immer nur auf die aktuellsten Daten zugreifen und nicht auf jene aus vorherigen „Versionen“. Da es ein Ziel des Projektes ist, Trends in den Daten zu erkennen und zu analysieren, reichen die bisherigen Tabellen nicht aus.

Aus diesem Grund baue ich mir eigene Tabellen um die Daten historisch zu speichern. Jedoch können meine Tabellen auch kein genaues Abbild der bisherigen Tabellen sein, da die Datenmenge sonst deutlich zu groß werden. Als Beispiel: Eine Tabelle, die ursprünglichweise 100 *Mega-Byte (MB)* groß war, erstellt nach einem monatlichen Abruf und Speichern der Daten über 10 Jahre, eine Tabelle die nun 240.000 *MB* groß ist, oder auch 240 *Giga-Byte (GB)*.

4 Implementierung

4.1 Feature Reporting

4.1.1 Sammeln der historischen Daten

Der erste Teil des Feature Reportings ist es, die Daten historisch zu sammeln und zu speichern. Dafür benötigt sind die oben angesprochenen - bereits existierenden - Tabellen. Es existieren nahezu unendlich viele verschiedene Tabellen. Daher musste ich als erstes festlegen, welche Tabellen gebraucht werden. Da ich das Feature Reporting für SAP interne Daten erstelle, basiert die Auswahl der Tabellen auf Objekten, die in den Teams rund um *ABAP* genutzt werden. Das sind die Teams: „ABAP CDS 1“, „ABAP CDS 2“ und „ABAP DDIC (Data-Dictionary)“. Aus dieser Information lässt sich jedoch noch nicht schließen, welche Objekttypen den Teams gehören. Um daran zu kommen, gibt es Listen mit den Usernamen der Mitglieder der Teams, welche ich mit den Autoren der Objekttypen, verglichen habe. Der erste Schritt von diesem Prozess ist es die verschiedenen Mitglieder der Teams zu einer Liste zusammenzufassen. Es gibt bereits einzelne Mitgliederlisten der drei Teams, welche ich nun zu einer einzelnen „member_list“ zusammenfasse:

```
1 method fill_members.  
2   member_list = get_members_of( co_team-dictionary ).  
3   append lines of get_members_of( co_team-cds1 ) to member_list.  
4   append lines of get_members_of( co_team-cds2 ) to member_list.  
5 endmethod.
```

Code 5: Mitgliederliste füllen

Da ich nun die Mitgliederlisten zusammengefasst habe, kann ich diese mit den Autoren der Objekte vergleichen um an die Liste von Objekttypen zu kommen, die zu diesen Teams gehören. Dazu kommen noch ein paar weitere Datentypen, die nicht speziell zu den Teams gehören, jedoch trotzdem wichtig für das Projekt sind. Diese füge ich im Nachhinein zur Liste von Objekten hinzu.

```
1 method get_objecttypes.  
2   select object_types  
3   from tadir  
4   inner join @team_members as members on tadir~author = members~table_line  
5   where tadir~object = 'SVAL'
```

```
6  into table @data(object_type_names).
7
8  append 'ABL' to object_type_names.
9  append 'DDL' to object_type_names.
10 append 'DRAS' to object_type_names.
11 append 'DSFD' to object_type_names.
12 append 'DSFI' to object_type_names.
13 append 'DRTY' to object_type_names.
14 append 'TABL' to object_type_names.
15
16 return object_type_names.
17 endmethod.
```

Code 6: Objekttypen speichern

Diese dadurch ermittelten Objekttypen speichere ich in einer von mir erstellten Tabellen. Anhand von dieser Tabelle erstelle ich später ein CDS-View, um die gesamten Daten zu verknüpfen.

Außerdem gibt es für jeden der dadurch ermittelten Objekttypen weitere Metadatentabellen, welche die einzelnen Objekttypen näher beschreiben. Meist ist dies eine sogenannte „Header“ Tabelle und eine „Fields“ Tabelle. In einer „Header“ Tabelle werden grundlegende Informationen über die Objekttypen gespeichert, zum Beispiel das originale System, der Ersteller, oder der Objektname im Objekteverzeichnis. In einer „Fields“ Tabelle findet man alle möglichen Informationen über die einzelnen Felder eines Objektes, zum Beispiel der Name des zugehörigen Objektes, ob das Feld ein Schlüsselattribut ist, oder den Datentyp.

Da diese Tabellen auch viele - für mein Projekt irrelevante - Informationen enthalten, musste zunächst auch noch festgelegt werden, welche Spalten und Datensätze überhaupt nützlich sind und gespeichert werden sollten. Diese Auswahl basiert vor allem auf der Überprüfung der einzelnen Felder. Es gibt viele historische Felder, die mittlerweile redundant sind oder nicht mehr genutzt werden. Diese beiden Arten von Felder sind demnach nicht mehr relevant, sodass dafür keine Ressourcen zum Speichern aufgewendet werden sollten.

4.1.2 Speichern der Daten in eigenen Tabellen

Da die gespeicherten Daten im Nachhinein historisch nachvollziehbar sein sollen, muss es einen Informationssatz geben, der speichert wann und von wem die Daten gespeichert wurden. Um dies zu erfüllen habe ich eine eigene Tabelle zum Speichern von Versionen erstellt. Diese Tabelle enthält die Felder: „versionID“, „saved_on“, „author“. „VersionID“ ist das Schlüsselat-

tribut vom Typ „char“ und ist maximal 20 Zeichen lang. „Saved_on“ ist vom Typ „timestamp“ und speichert den - bis auf die Sekunde genauen - Zeitpunkt des Abruf. „Author“ ist vom gleichnamigen Typ „author“ und speichert den SAP internen Username der Person, die den Datenabruf durchgeführt hat. Diese Tabelle rufe ich in den weiteren Tabellen, durch eine Referenz auf das Schlüsselattribut „versionid“ auf.

```
1  key versionid : versionid not null;
2  saved_on      : timestamp;
3  author        : author;
```

Code 7: Versionstabelle

Die Daten werden in meinen eigenen Tabellen gespeichert, welche ich auf Basis der bisher vorhandenen Tabellen gebaut habe, jedoch habe ich einige der - für mein Projekt irrelevanten - Spalten zum Sparen von Ressourcen ausgelassen. Als Beispiel ist hier der Datenabruf aus einer der bereits existierenden Tabelle mit den Feldern:

```
1  key aspect_name : dd_dras_name not null;
2  key as4local    : as4local not null;
3  as4user         : as4user;
4  as4date         : as4date;
5  as4time         : as4time;
6  aspect_name_raw : dd_dras_name_raw;
```

Code 8: Bereits existierende Tabelle

Diese Daten werden in einer zugehörigen - von mir erstellten - Tabelle, mit den folgenden Feldern, gespeichert:

```
1  key uuid        : sysuuid_c32 not null;
2  key aspect_name : dd_dras_name not null;
3  as4user         : as4user;
4  as4date         : as4date;
5  as4time         : as4time;
6  aspect_name_raw : dd_dras_name_raw;
7  version         : versionid;
```

Code 9: Meine zugehörige Tabelle

Das Feld „as4local“ aus der Originaltabelle ist für mich nur zum Teil relevant. Es speichert den Aktivierungsstatus von einzelnen Objekten. Für mich relevant sind nur aktive Objekte, weshalb ich die Daten so filtere, dass nur Daten für as4local = 'A' gespeichert werden. Da jeder Eintrag nun as4local = 'A' enthält, brauche ich den Wert des Feldes nicht mehr in meiner Tabelle speichern.

Die neu dazugekommenen Felder „uuid“ und „version“ weise ich bei jeder Abspeicherung zu. Der *Universally Unique Identifier (UUID)* wird durch einen Abruf einer externen Methode erstellt. Dies erstellt einen 16 Byte *UUID* im Hex Format.

```
1 uuid = cl_system_uuid=>create_uuid_c32_static( ).
```

Code 10: UUID erstellen

Das Feld „version“ soll dazu dienen das historische Speichern der Daten nachvollziehen zu können. Dafür erstelle ich mit folgender Methode jedes Mal, wenn die Daten neu abgerufen und gespeichert werden, einen neuen Eintrag in der „versiontable“.

```
1 method new_version.  
2   select versionid from versiontable  
3   into @data(current_version)  
4   order by saved_on ascending.  
5   endselect.  
6  
7   if current_version is initial.  
8     curr_version = 1.  
9   else.  
10    curr_version = current_version + 1.  
11  endif.  
12  
13  data(lv_date) = sy-datum.  
14  data(lv_time) = sy-uzeit.  
15  
16  convert date lv_date time lv_time into time stamp data(lv_ts) time zone sy-  
    zonlo.  
17  append value #( versionid = curr_version  
18                  saved_on = lv_ts  
19                  author = sy-uname ) to version_table.  
20  
21  insert versiontable from table @version_table.  
22  return curr_version.  
23 endmethod.
```

Code 11: Neue Version erstellen

Der Datenabruf geschieht über eine Methode „pull_data“, welche unter anderem die oben genannte Methode „new_version“ (Code 11) aufruft. Außerdem gibt es für jede Tabelle eine separate Methode, die die Daten abrufen und in meiner dazugehörigen Tabelle speichert. Diese werden auch durch die „pull_data“ Methode aufgerufen.

Hier ist zur Veranschaulichung eine Methode, die die Daten aus der oben genannten Tabelle zieht und in meiner zugehörigen Tabelle speichert:

```
1 select aspect_name,  
2        as4user,  
3        as4date,  
4        as4time,  
5        aspect_name_raw,  
6        @current_version as version,  
7        @( cl_system_uuid=>create_uuid_c32_static( ) ) as uuid  
8 from dddras_header  
9 where as4local = 'A'  
10 into corresponding fields of table @lt_dddcras_header_d.  
11  
12 insert zrvn_dddcras_head from table @lt_dddcras_header_d.
```

Code 12: Daten selektieren

In dem select-Statement werden die einzelnen Felder - die ich speichern möchte - definiert. Außerdem wird der Methode ein Wert „current_version“ als Übergabeparameter übergeben. Dieser wird durch das select-Statement in dem Feld „version“ gespeichert. Das Feld „uuid“ wird über den Methodenaufruf erstellt und gespeichert, den ich bereits hier (Code 10) erklärt habe. Danach werden die Daten erst in einer lokalen Tabelle gespeichert - erkennbar durch „lt_“ vor dem Bezeichner - und danach erst in meiner zugehörige Tabelle eingefügt.

Für die Analyse brauche ich in verschiedenen Tabellen noch Zähler, die zum Beispiel die Anzahl an genutzten Funktionen zählen. Da man diese nicht direkt in der Definition einer Tabelle erstellen kann, benötigt jede Tabelle noch eine „Dimension-View“ welche „über“ der Tabelle liegt. Diese dient sozusagen als „Abbild“ der Tabelle und ermöglicht die Implementierung von Aggregationen, wie zum Beispiel einem Zähler.

4.2 Datenanalyse

4.2.1 Aufbau des Datenmodell

Um die genauere Speicherung und Verarbeitung der Daten nachzuvollziehen, ist es nötig zu verstehen wie genau die Tabellen aufgebaut sind. Das gesamte Tabellenschema basiert auf einer Art OLAP-Würfel, einem mehrdimensionalen relationalen Datenmodell. Als sogenannte Hauptdimension liegt meine Versionstabelle, die die Version des Datenabrufes speichert. Da jeder Datenaufwurf anhand derselben Kriterien geschieht, kann man grundsätzlich beschreiben, wie die weiteren Dimensionen aussehen werden. Als nächstes gibt es die Daten aus der „tadir“-

Tabelle, welche alle Objekttypen enthält, die innerhalb des SAP-Systems existieren und genutzt werden. Diese Objekttypen an sich bilden eine weitere Dimension. Jeder dieser Objekttypen enthält verschiedene Objekte, die durch weitere Metadatentabellen näher beschrieben werden. Diese Tabellen werden „Header“- und „Field“-Tabellen genannt und bilden eine zusätzliche Dimension. Die Felder der Metadatentabellen speichern verschiedene Informationen über die Objekte, anhand von welchen ich die Tabellen weiter verknüpfen kann. Verknüpfungen und Vergleiche innerhalb einer Dimension funktionieren ohne Probleme und lassen sich ohne weitere Komplikationen durchführen. Für einige Abfragen und Aufrufe, muss ich jedoch Informationen aus verschiedenen Dimensionen miteinander verknüpfen. Da dies nicht mehr über einfache Abfragen geht, muss ich einen Teil der Informationen von der einen Dimension in eine andere verschieben. Dies funktioniert, indem ich beide Informationspaare in einer „Dimension-View“ zusammenfasse. Diese erstelle ich innerhalb der gewünschten Dimension und kann sie innerhalb des OLAP-Würfels aufrufen. Dadurch lassen sich beispielsweise Funktionsnamen mit denen der skalaren Funktionen vergleichen, um zu überprüfen wo die Schnittmenge liegt und wie viele Elemente sie enthält.

4.2.2 OLAP-Würfel erstellen

Um die Daten in dem OLAP-Würfel organisiert zu speichern und verarbeiten zu können, muss ich diesen erst einmal erstellen. In diesem Würfel binde ich alle von meinen Tabellen ein und verknüpfe diese über die gemeinsamen Attribute Objekttyp, Objektname, der Version. Durch diese Verknüpfungen können die Daten aus den verschiedenen Tabellen den zugehörigen Daten aus den anderen Tabellen zugeordnet werden. Als Basis für den OLAP-Würfel nutzte ich die Daten, die ich in meiner obersten „Header“-Tabelle speichere. Das sind die Objekttypen und die zugehörigen Objektnamen. Da es für die einzelnen Objekttypen mehrere Tabellen gibt, die diese weiter beschreiben, binde ich sie durch sogenannte Assoziationen an und lasse mir die Felder anzeigen.

Eine solche Assoziation kann zum Beispiel so aussehen:

```
1 association [*] to dras_header as _DrasHead
2   on $projection.Version      = _DrasHead.version
3   and $projection.ObjectType = 'DRAS'
4   and $projection.ObjectName = _DrasHead.aspect_name
```

Code 13: Anbindung der Headertabelle für 'DRAS'

Als erstes Verknüpfungskriterium benutze ich die Version die ich mir in Code 11 erstelle. Als nächstes filtere ich die Daten aus der „Headertabelle“ anhand des Objekttypen, sodass nur noch Datensätze vom Type ‚DRAS‘ gelesen werden. Das letzte Filterkriterium ist der Name der untergeordneten Objekte, die mit den Aspektnamen aus der ‚DRAS‘-Tabelle übereinstimmen müssen. Durch diese Abfragen verknüpfe ich die Tabellen in den verschiedenen Dimensionen und kann so die Informationen leichter abrufen. Alle Assoziationen in meinem Feature-Cube basieren auf diesem Schema.

Im unteren Teil des OLAP-Würfel werden die Felder angegeben, die ich aufrufen und benutzen möchte. Diese Auswahl wird erst später nach der Definition der Queries vollständig, da ich erst dann genau weiß, welche Felder benötigt werden. Felder die ich in jedem Fall brauche sind die Key-Felder der assoziierten Tabellen. Außerdem muss jede Assoziation selbst in der Liste der Felder angegeben werden.

4.2.3 Queries definieren

Um die von mir gespeicherten Daten nun auch besser analysieren zu können benötige ich sogenannte Queries. Bei der Definition der Queries habe ich mich zum Start an jenen orientiert, die von einem vorherigen Studenten erstellt wurden. Sie haben alle verschieden „Aufgaben“ und implementieren daher verschiedene Felder. Die meisten Queries bauen auf der Zählung von verschiedene Informationen auf, weshalb es von Nöten war in einigen der Tabellen Zählervariablen für verschiedene Elemente einzubauen. Es gibt zum Beispiel einen Zähler für Funktionen, den man später gruppieren kann und somit die Anzahl an Funktionen für ein bestimmten Quelltyp sich anzeigen lassen kann. Außerdem kann man sich die Ergebnisse nach Datum/ Zeitpunkt der Speicherung gruppieren lassen und so Trends in den Daten visualisieren.

Das erste von mir definierte Query dient dazu die Anzahl der genutzten Assoziationen, in Abhängigkeit von verschiedenen Faktoren wie der Version, dem Paket oder der Software-Komponente anzeigen und kalkulieren zu lassen.

Der nächste von mir definierte Query funktioniert ähnlich wie der ersten, außer dass er die Anzahl der Elemente in Abhängigkeit von der Version, dem Paket oder zum Beispiel der Software-Komponente anzeigt.

Alle weiteren Queries funktionieren sehr ähnlich zu den beiden bisher beschriebenen, analysieren jedoch unterschiedliche Werte. Ein letzter Query auf den ich eingehen möchte dient dazu die Anzahl der Funktionen zu bestimmen. Außerdem bestimmt der Query - in Abhängigkeit zu bestimmten Kriterien - die Anzahl der Funktionen, die Anzahl der skalaren Funktionen und den entsprechenden Anteil an skalaren Funktionen.

Die Ergebnisse der Queries kann man sich in zwei möglichen Formen anzeigen lassen, in Tabellenform und in einem Balkendiagramm, wodurch man verschiedenen Blickrichtungen bekommt und die Daten unterschiedlich analysieren kann.

4.3 Auswertungen

Für die Auswertung der Daten habe ich mir verschiedene Versionen der Daten erstellt und gespeichert. Aufgrund von Problemen mit der Verarbeitung der Queries, musste ich die Anzahl der Datensätze limitieren, weshalb ich nur Daten von ca. 10000 Objekten speichern konnte. Dies hat natürlich Auswirkungen auf die Qualität der Auswertung, da sich die Datenmenge bei jedem Datenabruf verändert und unterschiedliche Ausschnitte aus dem gesamten Datensatz gespeichert werden. Außerdem gab es einige Probleme mit dem internen System XT7, weshalb ich all meine erstellten Objekte (Tabellen, Views, Queries) in ein anderes System übertragen musste. Dadurch sind alle bis zu dem Zeitpunkt der Übertragung erstellten Daten verloren gegangen und die nun von mir erstellten Versionen wurden nicht im - wie ursprünglicherweise geplant - wöchentlichen Abstand erstellt, sondern in einem Abstand von wenigen Stunden. Dies hat natürlich auch Auswirkungen auf die Qualität der Auswertung, da sich in so einem kurzen Zeitraum kaum Veränderungen in den Daten ergeben.

Dennoch lassen sich einige Trends erkennen.

Trotz der geringen Zeiträume zwischen den Versionen, gibt es einige Veränderungen in den Daten. Wenn man sich die Ergebnisse der „ScalarFunctionQuery“, die dazu da ist die Anzahl der Funktionen und die Anzahl der skalaren Funktionen anzuzeigen, anguckt sieht man zum Beispiel, dass

5 Fazit

6 Advanced Elements

6.1 Figures

Inserting figures and code blocks into your Typst document enhances its informational depth. When specifying a `caption` for a figure, the template will automatically generate a list of figures, making it easy to navigate your document.

Note When using „ieee“ Sorting for bibliography, the sources for figures will be evaluated before the text. To prevent „false sorting“, you can use `#caption_with_source("Text", [@source])` instead. This will display the caption in outlines without source and will the source evaluate at the time the figure is displayed

6.1.1 Image Figures

Typst Code	Output
<pre>1 #figure(2 image("assets/SAP-Logo.svg"), 3 caption: "SAP Logo" 4)</pre>	 Abbildung 1: SAP Logo

6.1.2 Code Snippets:

This template uses [Codly](#) for code snippets. Look at their documentation on how to further customize and control your code blocks.

Besides that the template provides two functions to create code snippet figures that get listed in a source code listing: `codefigure` and `codefigurefile`.

Use `codefigure` to display a code figure from the provided code.

Typst Code	Output
<pre>1 #codefigure(caption: "My Code") [``rust 2 fn main() { 3 println!("Hello World!"); 4 } 5 ``]</pre>	<div><pre>1 fn main() { 2 println!("Hello World!"); 3 }</pre></div> <p>Code 14: My Code</p>

Note You can also provide custom syntax (`.sublime-syntax`) files for code highlighting. The template already includes a syntax file for CDS, so you can use the `cds` language in your code blocks.

Use `codefigurefile` to create a code snippet figure from the content of a file. Note that the provided file is searched relative to `./`.

Typst Code	Output
<pre>1 #codefigurefile(2 "assets/example-code.typ", 3 caption: "My Code from a file" 4)</pre>	<div><pre>1 #codefigure(caption: "My Code") [``rust 2 fn main() { 3 println!("Hello World!"); 4 } 5 ``]</pre></div> <p>Code 15: My Code from a file</p>

6.2 Math

The math syntax is a loose interpretation of LaTeX, allowing you to create complex mathematical equations with ease. See the [Typst documentation](#) for a detailed overview of the math syntax.

Typst Code	Output
<pre>1 \$ \sum_{k=0}^n k 2 &= 1 + \dots + n \ 3 &= (n(n+1)) / 2 \$</pre>	$\sum_{k=0}^n k = 1 + \dots + n$ $= \frac{n(n+1)}{2}$

6.3 Block Quotes

Typst Code	Output
<pre>1 #quote(attribution: [Frankling D. Roosevelt]) [2 The only thing we have to fear is fear itself. 3]</pre>	<p>The only thing we have to fear is fear itself.</p> <p>— Frankling D. Roosevelt</p>

6.4 Notes

This template uses [Drafting](#) for notes.

Check out their documentation for more advanced use cases.

You might have noticed the notes listing on the first page of this document. This listing reminds you of the notes still present in your document. Once you remove all notes, the listing will disappear.

7 References and Citations

7.1 Local Elements

You can reference local elements like figures, code blocks, and sections using the `ref()` function. You can also use the syntax sugar `<ref>` to define and `@<ref>` to reference references.

Typst Code	Output
<pre>1 = Section 1 <section-1> 2 Some important text 3 4 = Section 2 5 More important text, just like @section-1</pre>	<p>vii.ii. Important Section</p> <p>Some important text</p> <p>vii.iii. Other Section</p> <p>More important text, just like Abschnitt vii.ii</p>

7.4 Code Blocks

If you use the provided `codefigure` function, you can specify a reference name via the `reference` parameter. This allows you to reference the code block later in the document.

Typst Code	Output
<pre>1 #codefigure(caption: "Code w/ Ref", 2 reference: "my-rust-code")[``rust 3 fn main() { 4 panic!("Hilfe!"); 5 } 6 ``] 7 Look at my code in @my-rust-code!</pre>	<pre>1 fn main() { 2 panic!("Hilfe!"); 3 }</pre> <p>Code 16: Code w/ Ref</p> <p>Look at my code in Code 16!</p>

Literaturverzeichnis

- [1] „4ap/ABAP“. Zugriffen: 3. November 2025. [Online]. Verfügbar unter: <http://www.4ap.de/abap/historie>
- [2] „ABAP - Keyword Documentation“. Zugriffen: 3. November 2025. [Online]. Verfügbar unter: https://help.sap.com/doc/abapdocu_latest_index_htm/LATEST/en-US/ABENABAP.html
- [3] „Evolution of ABAP“. Zugriffen: 3. November 2025. [Online]. Verfügbar unter: <https://community.sap.com/t5/enterprise-resource-planning-blog-posts-by-sap/evolution-of-abap/ba-p/13522761>
- [4] „SQL“. Zugriffen: 3. November 2025. [Online]. Verfügbar unter: <https://de.ryte.com/wiki/SQL/>
- [5] „CDS Annotationen“. Zugriffen: 14. November 2025. [Online]. Verfügbar unter: <https://de.ryte.com/wiki/SQL/>
- [6] „ABAP Queries“. Zugriffen: 9. Dezember 2025. [Online]. Verfügbar unter: https://help.sap.com/doc/abapdocu_latest_index_htm/latest/en-US/ABENCDS_ANALYTICAL_QUERY_APV.html
- [7] „CDS Kardinalität“. Zugriffen: 9. Dezember 2025. [Online]. Verfügbar unter: https://help.sap.com/doc/abapdocu_latest_index_htm/latest/en-US/ABENCDS_CARDINALITY_V2.html