



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, 12. 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 -

Für die effiziente Entwicklung von neuen Objekten und Features in ABAP CDS ist es essentiell, Entscheidungen so zu treffen, dass sie bestehende Anforderungen erfüllen und gleichzeitig zukünftige Bedürfnisse antizipieren. Durch die Analyse historischer Metadaten über diese Objekte lassen sich wertvolle Einblicke gewinnen, die dabei helfen, diese Entscheidungen fundierter zu treffen. Das Ziel dieses Projektes ist es, ein System zu entwickeln, das die benötigten Daten in regelmäßigen Abständen ausliest und in entsprechenden Abbildungen der Ursprungstabellen speichert. Außerdem soll dieses System die Daten mit Versionen versehen, um die historische Entwicklung nachvollziehbar zu machen. Letztendlich soll ein Framework entwickelt werden, das diese historischen Daten durch vordefinierte Queries analysierbar macht. Dadurch soll es möglich werden, Informationen über die einzelnen Objekte und Features, nicht nur im aktuellen Zustand, sondern auch historisch in früheren Versionen zu erhalten.

ABSTRACT

- English -

In order to efficiently develop new objects and features in ABAP CDS, it is essential to make decisions in such a way that they meet existing requirements while also anticipating future needs. By analyzing historical metadata about these objects, valuable insights can be gained that help make these decisions more well-founded. The goal of this project is to develop a system that regularly reads out the required data and stores it in corresponding representations of the tables. Additionally, this system is supposed to assign a version for every new data-pull in order to make the historical development traceable. Ultimately, a framework should be developed that makes it possible for users able analyse this historical data through predefined queries. This will make it possible to obtain information about the individual objects and features not only in their current state, but also historically in earlier versions.

Inhaltsverzeichnis

| | |
|---|-------------|
| Abbildungsverzeichnis | VII |
| Quellcodeverzeichnis | VIII |
| 1 Einleitung | 1 |
| 1.1 Motivation | 1 |
| 1.2 Zielsetzung | 2 |
| 2 Grundlagen | 3 |
| 2.1 ABAP | 3 |
| 2.2 SQL | 4 |
| 2.3 Core-Data-Service | 4 |
| 2.3.1 CDS Views | 5 |
| 2.3.2 CDS Annotationen | 5 |
| 2.4 Metadatentabelle | 7 |
| 2.5 Star-Schema | 7 |
| 2.6 Queries | 8 |
| 2.7 OLAP Würfel | 9 |
| 2.8 Assoziation | 9 |
| 2.9 Kardinalitäten | 10 |
| 3 Bisheriger Stand | 12 |
| 4 Implementierung | 13 |
| 4.1 Feature Reporting | 13 |
| 4.1.1 Sammeln der historischen Daten | 13 |
| 4.1.2 Speichern der Daten in eigenen Tabellen | 15 |
| 4.2 Datenanalyse | 18 |
| 4.2.1 Aufbau des Datenmodell | 18 |
| 4.2.2 OLAP-Würfel erstellen | 19 |
| 4.2.3 Queries definieren | 20 |
| 4.3 Auswertungen | 21 |
| 4.3.1 Query für Assoziationen | 22 |
| 4.3.2 Query für Funktionen und durchschnittliche Anzahl an Funktionen | 23 |
| 4.3.3 Query für skalare Funktionen | 24 |
| 4.3.4 Query für die Anzahl der Elemente | 25 |
| 5 Fazit | 26 |
| Literaturverzeichnis | a |

Abkürzungsverzeichnis

ABAP – Advanced Business (and) Application Program 3, 13

CDS – Core-Data-Service 4, 5

GB – Giga-Byte 12

MB – Mega-Byte 12

SQL – Standard Query Language 4

UUID – Universally Unique Identifier 16

Abbildungsverzeichnis

| | |
|---|----|
| Abbildung 1 Ergebnisse der Assoziations-Query | 23 |
| Abbildung 2 Ergebnisse des Average-Function-Query | 24 |
| Abbildung 3 Ergebnisse des Scalar-Function-Query | 24 |
| Abbildung 4 Ergebnisse des Element-Count-Query | 25 |

Quellcodeverzeichnis

| | | |
|---------|--|----|
| Code 1 | Annotationsstruktur | 6 |
| Code 2 | Beispiel einer Annotation | 6 |
| Code 3 | Beispiel Query | 8 |
| Code 4 | Beispiel Assoziation | 10 |
| Code 5 | Mitgliederliste füllen | 13 |
| Code 6 | Objekttypen speichern | 14 |
| Code 7 | Versionstabelle | 15 |
| Code 8 | Bereits existierende Tabelle | 15 |
| Code 9 | Meine zugehörige Tabelle | 16 |
| Code 10 | UUID erstellen | 16 |
| Code 11 | Neue Version erstellen | 16 |
| Code 12 | Daten selektieren | 17 |
| Code 13 | Anbindung der Headertabelle für 'DRAS' | 19 |

1 Einleitung

1.1 Motivation

In einer zunehmend datengetriebenen Unternehmenslandschaft gewinnen analytische Auswertungen und Berichte kontinuierlich an Bedeutung. Entscheidungen darüber, welche Features oder welche Software entwickelt werden sollen, 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 im Laufe der 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 gesamte 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, das durch eine versionsbasierte Analyse der Metadaten die historische Entwicklung der Feature-Nutzung transparent macht und somit datenbasierte Entscheidungen ermöglicht.

Um dieses Hauptziel zu erreichen, implementiere ich eine automatisierte Datenspeicherung mit zugeordneten Versionen. Das System liest in regelmäßigen Intervallen festgelegte Daten aus Quelltabellen aus und überführt sie in dedizierte Abbilder dieser Tabellen zur historischen Speicherung. Jede Auslesung erstellt dabei eine neue, eindeutig identifizierbare Version. 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 Objekt, 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 ermöglichen es, datenbasiert zu entscheiden, welche Features und Objekte weiter entwickelt werden sollen und welche 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 Betriebssys-

temen und Datenbanken gewährleistet. Darüber hinaus bietet die Entwicklungsumgebung „ABAPDevelopmentTools“ 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 Core-Data-Service

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's 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's 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 werden beim Programmieren unter anderem als Strukturierungsmittel verwendet. In den CDS's 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 verschieden Typen von Annotationen; man differenziert zwischen View-Annotationen und Element-Annotationen. View-Annotationen sind Vermerke auf ein

gesamtes View, wohingegen sich Element-Annotationen nur auf einzelne 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 einer 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 verschiedenen 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 mit Hilfe 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 Fremdschlüssele, 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 „Her-einzoomen“ 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.

ben. 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 Momentanaufnahmen 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 bau 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 ursprünglicherweise 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 'DDLS' 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 Schlüsselattribut 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 timestamp data(lv_ts) time zone
17    sy-zonlo.
18  append value #( versionid = curr_version
19                saved_on = lv_ts
20                author = sy-uname ) to version_table.
21
22  insert versiontable from table @version_table.
23  return curr_version.
24 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 abruft 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_dddras_header_d.
11

```

```
12 insert zrvn_dddras_head from table @lt_dddras_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 „It_“ 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 „Abbildung“ 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 Tabellschema basiert auf einer Art OLAP-Würfel, einem mehrdimensionalen relationalen Datenmodell. Als sogenannte Hauptdimension liegt meine Versionstabelle, die die Version des Datenabrufes speichert. Da jeder Datenauftrag 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 benutztten 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 verschieden 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 das System ODE 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 aus den erstellten Versionen einige Trends erkennen:

4.3.1 Query für Assoziationen

Es lässt sich zum Beispiel erkennen, dass die Anzahl der Assoziationen mit jeder neuen Version leicht ansteigt. Dies lässt sich dadurch erklären, dass mit der Zeit immer mehr Objekte erstellt werden, die wiederum Assoziationen benötigen um in Cube-Views - also OLAP-Würfeln - eingebunden zu werden.

Zwischen den Versionen 1 und 2 lagen ca. 2 Stunden Zeit, weshalb in dieser Zeit nur sehr wenige Objekte und entsprechend Assoziationen hinzugekommen sind. Die Veränderungen hängen auch sehr stark von den Arbeitszeiten der Entwickler ab, weshalb es in manchen Versionen kaum Veränderungen gibt, während es in anderen Versionen wieder mehr sind. So gab es beispielsweise zwischen Version 1 und 2 in fast allen Source Typen einen kleinen Anstieg an genutzten Assoziationen, während zwischen den Versionen 2 und 3 in manchen Source Typen sogar ein leichter Rückgang zu verzeichnen war.

Interessant ist jedoch zu sehen, dass es in Version 4 nur extrem geringe Veränderungen in den Daten gab, im Vergleich zu Version 3. Am „Time-Stamp“ der Speicherung lässt sich erkennen, dass die Version 4 erst am nächsten Tag erstellt wurde. Demnach ist es verwunderlich, dass es in der Zeit keine Veränderungen gab. Es lässt sich jedoch dadurch erklären, dass das System „ODE“, in welchem die Daten gespeichert und abgerufen werden, lediglich ein Test-System ist, in dem in der Nacht nur wenige Entwickler an den Objekten arbeiten und somit auch kaum neue Objekte oder Assoziationen erstellt wurden.

Siehe folgende Grafik:

| Measures | Association Counter | | | |
|----------------|---------------------|---------------------|---------------------|---------------------|
| Version number | 1 | 2 | 3 | 4 |
| Time Stamp | 11.12.2025 11:37:32 | 11.12.2025 13:25:28 | 11.12.2025 15:55:30 | 12.12.2025 09:50:02 |
| Source Type | | | | |
| A | 140 | 158 | 159 | 159 |
| E | 0 | 0 | 0 | 0 |
| F | 0 | 0 | 0 | 0 |
| H | 6 | 31 | 7 | 7 |
| O | 0 | 0 | 0 | 0 |
| P | 576 | 681 | 664 | 664 |
| Q | 38 | 51 | 57 | 57 |
| T | 27 | 35 | 31 | 31 |
| V | 771 | 1.147 | 1.643 | 1.647 |
| W | 1.049 | 1.944 | 1.673 | 1.675 |
| X | 0 | 0 | 0 | 0 |
| # | 0 | 0 | 0 | 0 |

Abbildung 1: Ergebnisse der Assoziations-Query

Die Veränderungen an der Anzahl der Assoziations hängt außerdem sehr stark von der Software-Komponente ab. In manchen Komponenten gibt es einen deutlichen Anstieg an genutzten Assoziationen, während in anderen Komponenten kaum Veränderungen zu verzeichnen sind.

4.3.2 Query für Funktionen und durchschnittliche Anzahl an Funktionen

Auch bei der Anzahl der Funktionen gibt es einige Veränderungen. Die unterscheiden sich jedoch stark von den Veränderungen bei den Assoziationen. Während es bei den Assoziationen meist eher kleinere Veränderungen in den Daten gab, gibt es bei den Funktionen teils deutliche Sprünge. So steigt die Anzahl der Funktionen aus dem Source Type „W“ zum Beispiel von 1.612 Elementen in der ersten Version auf 2.172 Elementen in der zweiten Version und 4.140 Elementen in der dritten Version an. Spannend zu sehen ist jedoch das sich die für diesen Source Type die durchschnittliche Anzahl an Funktionen pro Einheit nur marginal von 3,558 auf 4,052 und 4,059 ändert. Dies lässt darauf schließen, dass in der Zeit zwischen den Versionen vor allem neue Einheiten erstellt wurden, die jedoch im Durchschnitt eine ähnliche Anzahl an Funktionen besitzen wie die bereits existierenden Einheiten.

Bei dem Source Type „P“ ist die Veränderung der Anzahl an Funktionen sowohl in der absoluten Anzahl als auch im Durchschnitt pro Einheit extremer.

Siehe Abbildung:

| | Version number | 1 | 2 | 3 | 4 |
|-------------|------------------------|---------------------|---------------------|---------------------|---------------------|
| | Time Stamp | 11.12.2025 11:37:32 | 11.12.2025 13:25:28 | 11.12.2025 15:55:30 | 12.12.2025 09:50:02 |
| Source Type | Measures | | | | |
| P | FUNCTIONCOUNTER | 238 | 268 | 60 | 60 |
| | Average Function Count | 9,154 | 8,375 | 6,667 | 6,667 |
| W | FUNCTIONCOUNTER | 1.612 | 2.172 | 4.140 | 4.140 |
| | Average Function Count | 3,558 | 4,052 | 4,059 | 4,059 |

Abbildung 2: Ergebnisse des Average-Function-Query

Hier kann man sogar erkennen, dass es keine Veränderungen in den Anzahlen der Funktionen zwischen der dritten und vierten Version gab.

4.3.3 Query für skalare Funktionen

In der Analyse der Anzahl der Funktionen und der skalaren Funktionen lassen sich in den ersten drei Versionen nur sehr wenige Veränderungen erkennen. Dies liegt daran, dass in dem kurzen Zeitraum nur wenig neue skalare Funktionen erstellt wurden. Lediglich im Source Type „W“ unter der Software-Komponente „SAP_Basis“ gibt es einen deutlichen Anstieg an Funktionen zwischen der zweiten und der dritten Version. Dies lässt sich vor allem auf die Art des Datenabrufes zurückführen, da ich die Anzahl der Datensätze aus Speicher- und Verarbeitungsgründen auf 10.000 Objekte limitieren musste. Das kann dazu führen, dass bei den unterschiedlichen Versionen, verschiedene Teilmengen aus den ursprünglichen Daten abgezogen werden. Hätte ich die Daten über einen längeren Zeitraum und mit einer größeren Datenmenge abrufen können, wären die Ergebnisse vermutlich aussagekräftiger gewesen.

| | Version number | 1 | 2 | 3 | 4 |
|-------------|-----------------------|---------------------|---------------------|---------------------|---------------------|
| | Time Stamp | 11.12.2025 11:37:32 | 11.12.2025 13:25:28 | 11.12.2025 15:55:30 | 12.12.2025 09:50:02 |
| Source Type | Measures | | | | |
| P | FUNCTIONCOUNTER | 238 | 268 | 60 | 60 |
| | SCALARFUNCTIONCOUNTER | 0 | 0 | 8 | 8 |
| W | FUNCTIONCOUNTER | 1.612 | 2.172 | 4.140 | 4.140 |
| | SCALARFUNCTIONCOUNTER | 91 | 91 | 61 | 61 |

Abbildung 3: Ergebnisse des Scalar-Function-Query

Auch hier ist zu erkennen, dass es in der vierten Version keine Veränderungen, im Vergleich zur dritten Version, gab.

4.3.4 Query für die Anzahl der Elemente

Im Gegensatz zu den vorherigen Queries, gibt es bei der Anzahl der Elemente in den verschiedenen Source Typen und Software-Komponenten keine Veränderungen. Das heißt es sind keine neuen Elemente hinzugekommen oder entfernt worden. Dies lässt sich vor allem durch die kurzen Abrufintervalle erklären, da es in so einem kurzen Zeitraum sehr unwahrscheinlich ist, dass derart wichtige Veränderungen im System vorgenommen werden.

| | Measures | Number of Elements | | | | ▼ |
|-------------|--------------------|---------------------|---------------------|---------------------|---------------------|---|
| | Version number | 1 | 2 | 3 | 4 | |
| | Time Stamp | 11.12.2025 11:37:32 | 11.12.2025 13:25:28 | 11.12.2025 15:55:30 | 12.12.2025 09:50:02 | |
| Source Type | Software Component | | | | | |
| H | HOME | 14 | 14 | 14 | 14 | |
| | SAP_BASIS | 11 | 11 | 11 | 11 | |
| P | HOME | 83 | 83 | 83 | 83 | |
| | SAP_ABA | 48 | 48 | 48 | 48 | |
| | SAP_BASIS | 40 | 40 | 40 | 40 | |
| | SAP_BW | 24 | 24 | 24 | 24 | |
| | SAP_GWFND | 24 | 24 | 24 | 24 | |
| | SAP_UI | 27 | 27 | 27 | 27 | |
| W | HOME | 138 | 138 | 138 | 138 | |
| | SAP_ABA | 51 | 51 | 51 | 51 | |
| | SAP_BASIS | 102 | 102 | 102 | 102 | |
| | SAP_BW | 23 | 23 | 23 | 23 | |
| | SAP_GWFND | 23 | 23 | 23 | 23 | |
| | SAP_UI | 33 | 33 | 33 | 33 | |

Abbildung 4: Ergebnisse des Element-Count-Query

5 Fazit

Das Projekt, welches ich in dieser Arbeit beschrieben habe, dient dazu die historische Entwicklung der Feature-Nutzung innerhalb von der internen ABAP CDS-Views zu speichern und zu analysieren. Durch die automatischen Speicherung der Daten in regelmäßigen Abständen in meine eigenen Tabellen und der Zuweisung einer Versionsnummer, ist es nun möglich die Daten historisch zu analysieren und abzurufen. In Kombination dazu habe ich einen OLAP-Würfel erstellt, der die verschiedenen Tabellen miteinander verknüpft und es mir ermöglicht analytische Queries zu erstellen, die diese Daten nach bestimmten Kriterien auswerten und visualisieren. Dieses Framework ermöglicht es dem Team nun Einblicke in die genaue Entwicklung und Nutzung der verschiedenen CDS -Objekte zu bekommen und datenbasiert die Entscheidungen zu treffen, welche Funktionen, Objekte und Elemente weiterentwickelt werden sollten. Die Integration des Datenabrufes erfolgte mit Absprache mit meinem Betreuer Kolja Groß, der mir bei der Implementierung und den technischen Herausforderungen zur Seite stand. Durch regelmäßige Meetings und Updates konnte ich sicherstellen, dass die Anforderungen des Teams erfüllt wurden und das Projekt in die richtige Richtung ging.

Es gab einige Herausforderungen und Probleme, die während der Implementierung des Datenabrufes auftraten, wie zum Beispiel lange Verarbeitungszeiten beim Zugriff auf die Daten, wordurch es nötig geworden ist die Datenmenge zu limitieren. Auch die Übertragung der erstellten Objekte in das System ODE führte zu Datenverlusten, da es nicht möglich war die bisher gespeicherten Daten zu übertragen. Dies und anfängliche Ungenauigkeiten im Code führten zu vermehrten Datenverlusten, die die schlussendliche Qualität der erhobenen Daten und folglich auch der Auswertung beeinflusst haben. Trotz dieser Herausforderungen konnte ich ein funktionierendes System erstellen, welches die Anforderungen des Projektes erfüllt. Für zukünftige Ausarbeitungen des System kann es sinnvoll sein, den Prozess der Datenspeicherung zu optimieren, um

die Limitierung der Datenmenge entfernen zu können. Außerdem können weitere Tabellen und Objekttypen in das System integriert werden, um nicht nur den Teams rund um die ABAP CDSs Einblicke in die Objektnutzung zu geben, sondern auch anderen Teams und Abteilungen innerhalb des Unternehmens.

Literaturverzeichnis

- [1] „4ap/ABAP“. Zugegriffen: 3. November 2025. [Online]. Verfügbar unter: <http://www.4ap.de/abap/historie>
- [2] „ABAP - Keyword Documentation“. Zugegriffen: 3. November 2025. [Online]. Verfügbar unter: https://help.sap.com/doc/abapdocu_latest_index_htm/LATEST/en-US/ABENABAP.html
- [3] „Evolution of ABAP“. Zugegriffen: 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“. Zugegriffen: 3. November 2025. [Online]. Verfügbar unter: <https://de.ryte.com/wiki/SQL/>
- [5] „CDS Annotationen“. Zugegriffen: 14. November 2025. [Online]. Verfügbar unter: <https://de.ryte.com/wiki/SQL/>
- [6] „ABAP Queries“. Zugegriffen: 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“. Zugegriffen: 9. Dezember 2025. [Online]. Verfügbar unter: https://help.sap.com/doc/abapdocu_latest_index_htm/latest/en-US/ABENCDS_CARDINALITY_V2.html