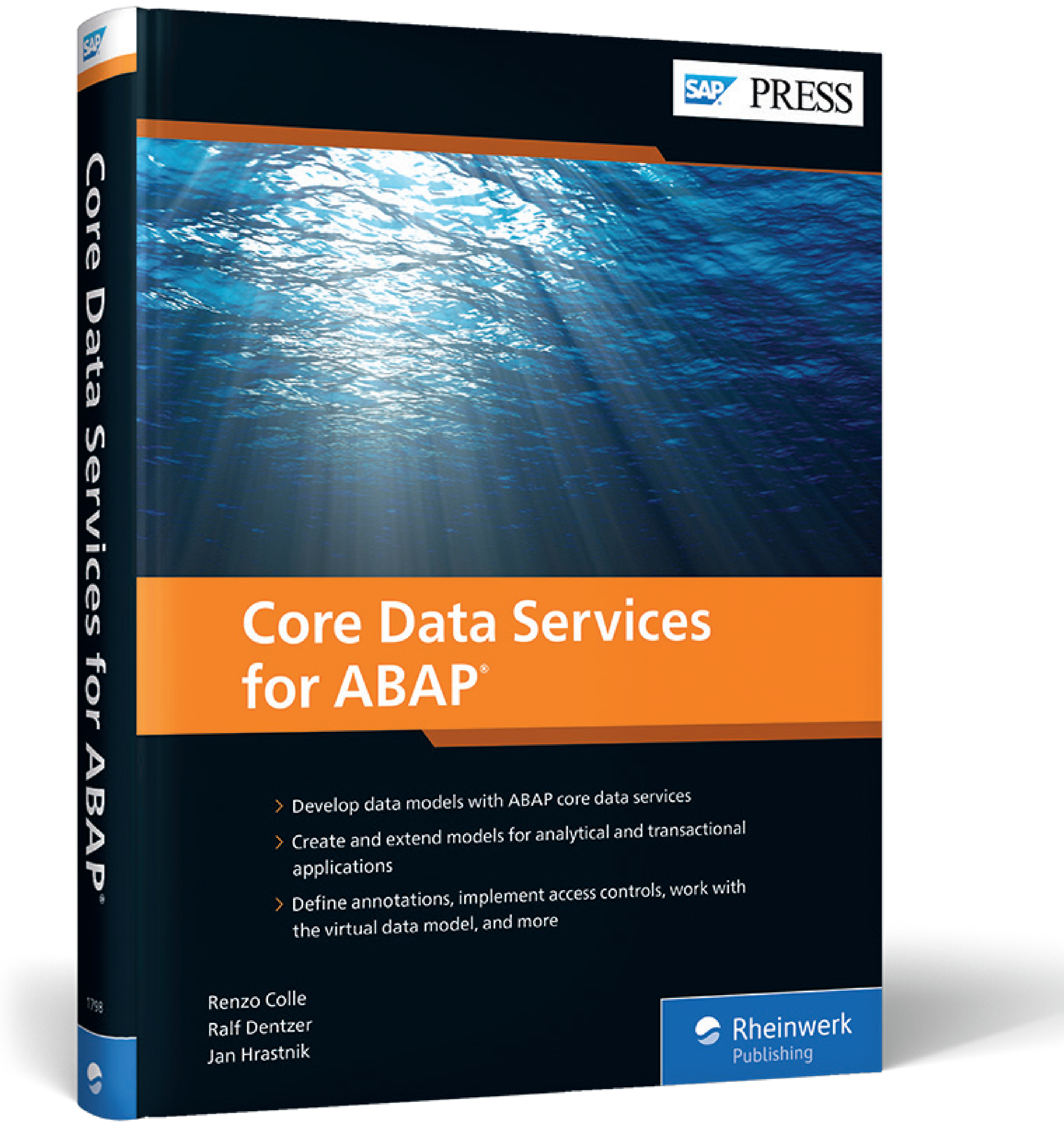
First-hand knowledge.



Browse the Book

In this chapter, you'll learn how essential application data is modeled, using SAP S/4HANA as an example. You'll see how to put together key pieces of metadata, including field labels, foreign key relations, text relations, and more.

"Modeling Application Data"

Table of Contents

Index

The Authors

Renzo Colle, Ralf Dentzer, and Jan Hrastnik

Core Data Services for ABAP

490 Pages, 2019, $79.95
ISBN 978-1-4932-1798-4

www.sap-press.com/4822

6

Chapter 6
Modeling Application Data

Core data services (CDS) support modeling of semantic properties of application data that far exceed the capabilities of traditional database views. Together with a modern technical infrastructure, this simplifies the development of new applications.

In previous chapters, you've learned how to define CDS views and use them in ABAP programs. With that knowledge, you can execute SQL mass operations and complex calculations directly in SAP HANA and benefit from SAP HANA's processing capabilities for large data sets. Moreover, you can conveniently formulate your requests with associations and their path notation. With this approach, you're still within the scope of the classical programming model for applications where you must repeatedly implement many details anew via individual programming.

However, new infrastructure components in the SAP NetWeaver Application Server for ABAP (SAP NetWeaver AS for ABAP) and semantic metadata of CDS views enable a new programming model, in which individual programming is reduced to a minimum. Instead, error-prone recurring programming tasks are handled generically by the infrastructure and controlled by business-motivated semantic annotations of CDS views.

Chapter structure

Section 6.1 gives a short overview of the *application architecture* and the *programming model* in SAP S/4HANA. The following sections present several types of *meta information* for application data that are evaluated by the new *application infrastructure*. They cover the following aspects:

- Field labels (Section 6.2)
- Field properties, such as quantities and amounts, aggregation behavior, system times, and texts in natural language (Section 6.3)
- Foreign key relations (Section 6.4)
- Text relations (Section 6.5)
- Composition relations (Section 6.6)

- Time-dependent data (Section 6.7)
- Hierarchical data (Section 6.8)

Later in this book, we'll discuss analytical and transactional applications (Chapter 8 and Chapter 9, respectively) in more detail and present further powerful metadata for CDS views that control their processing by the infrastructure.

6.1 Application Architecture in SAP S/4HANA

The core task of a business application is to read data, prepare it for display, present it to a user, receive new input or data changes by the user, check its consistency, process its impact, and finally persist the new data.

SAP Fiori UIs

In modern user interfaces (UIs), such as SAP Fiori, that are completely geared to the needs of the user, the first part of the user interaction plays an important role in that many different types of information are relevant for the user's task and should be directly available to support his decisions. The preparation of the required data and its display create substantial development efforts and require knowledge in different application areas.

However, for the second part of the user interaction—checking and processing data—proven program parts can be reused in most cases.

Read access

Experience shows that more than 90% of data accesses are read accesses. Write access happens less frequently. To reduce development effort, program complexity, and maintenance effort, the programming model of SAP S/4HANA was optimized for read accesses. It uses CDS views that prepare the raw data in a reusable way and add semantic metadata. The metadata is evaluated by infrastructure components, thus reducing the volume of individual programming.

Programming models in comparison

You can see the difference in programming models when comparing the classical SAP Fiori architecture in Figure 6.1 with the latest architecture in Figure 6.2, optimized for read accesses.

In the much-simplified representation of the two programming models, you see the common base structure: SAP Fiori apps use OData services that are provided by the ABAP application server, which in turn accesses data via SQL. The technical provisioning is done by the service infrastructure in both models.

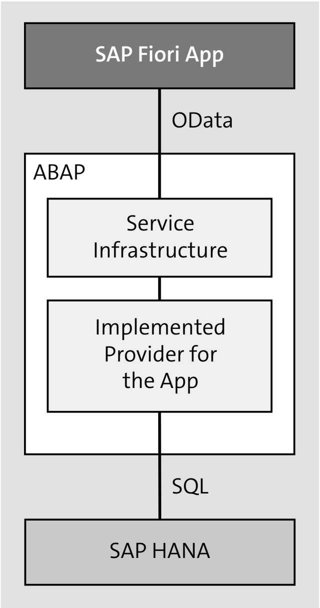


Figure 6.1 Classical SAP Fiori Architecture

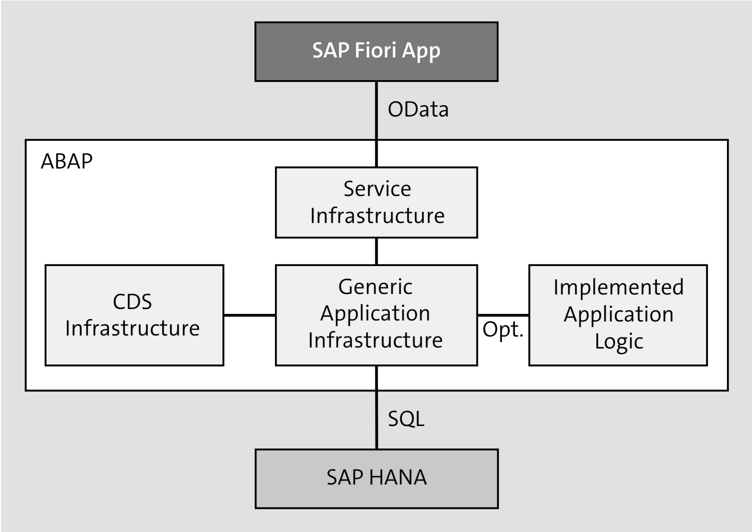


Figure 6.2 New Architecture for Read Access

OData service provider

Differences exist in the implementation of the service provider of the OData service and in the definition of the OData service itself. In the classical model, the OData service and its service provider are both individually implemented. In the new model, CDS views selected for a service provider define the structure of the OData service and its components. Moreover, a generic application infrastructure serves as a generic service provider for retrieving data via CDS views. Therefore, the definition of CDS views is the essential step in the development of a new OData service.

As the structure of the OData service corresponds to the structure of the CDS views, a read request to the OData service can be translated to SQL selections from these CDS views. The result of the SQL `SELECT` request is translated into corresponding entity sets of the OData service and returned as a response to the read request.

Special cases that, for example, need a special logic for the implementation of individual fields, can optionally be implemented in supplied extension methods.

Metadata

Metadata of the CDS views controls how the OData service for the CDS views is formed and how an OData request is translated into a SQL request. Some CDS metadata is translated into corresponding metadata of the OData service and thus exposed to the service consumers.

SAP Fiori elements

This creates further potential to simplify the development of SAP Fiori apps with *SAP Fiori elements*. SAP Fiori elements allow the construction of an SAP Fiori app from reusable *smart templates* and *smart controls*, whose concrete layout is controlled by the used OData service and its annotations. As the *UI annotations* needed for this can already be defined in the CDS views in use, the essential parts of an SAP Fiori app can be defined completely in CDS. We'll show a concrete example for this in Chapter 9, Section 9.4.4.

New architecture

In Figure 6.3, we added write access to data as well as the option to use other communication channels. This completes the conceptual model of the application architecture in SAP S/4HANA.

For changes of data, often existing application logic for checking and processing data and updating the database is connected. Therefore, the development effort in this programming model is mainly spent on defining CDS views with appropriate metadata, which is the central topic of this book.

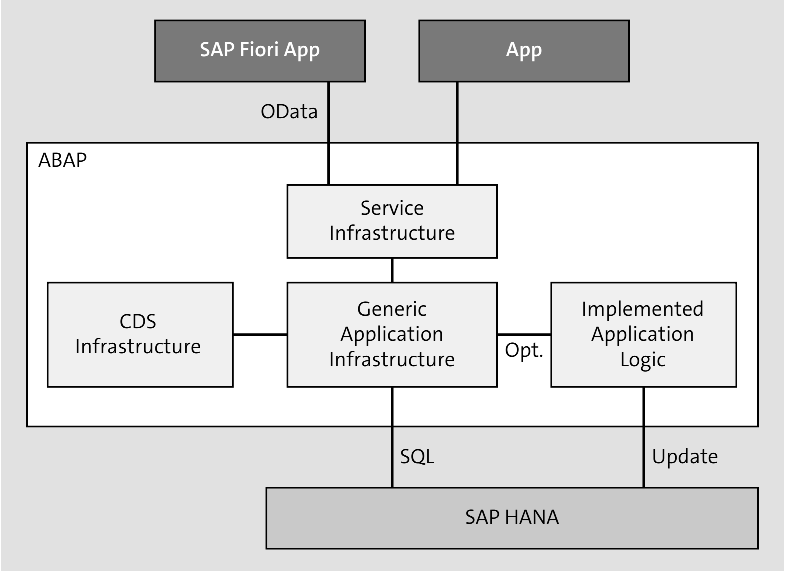


Figure 6.3 Overview of the New Architecture

Infrastructure components

The major piece of the generic application infrastructure is the *ABAP application infrastructure*, a new component of the ABAP application server introduced with SAP NetWeaver 7.5. It's complemented for analytic applications by the *analytic engine*, a component of SAP Business Warehouse (SAP BW), which is also available in SAP S/4HANA. The *service infrastructure* mainly consists of *SAP Gateway*, but it also uses components for other communication protocols, such as for analytic applications.

Benefits of the new architecture

The CDS-based application architecture of SAP S/4HANA offers the following benefits:

- Speed in the selection and preparation of big data sets for the UI as CDS views execute these steps directly in SAP HANA
- Simplified development and high consistency in the offered OData services through model-driven development based on a unified data model (the virtual data model [VDM], introduced in Chapter 7)
- Flexibility by optionally using implemented ABAP logic if needed

General benefits of the ABAP platform, such as the integrated development and transport environment, as well as general application services, such as user management and integrated authorization checks, are also available.

6.2 Field Labels

In this and the following sections, we'll present some metadata used in CDS models. We start with field labels. Every data field in a table or a CDS view needs not only a field name but also a description in the language of a user, that is, a *field label*. Field names in CDS views should be understandable as well but are only expressed in one language—usually English. Field labels, though, are supposed to be translated to many languages.

Origin of field labels

Field labels provide important semantic information about a data field. SAP applications have always offered the option to reuse field labels from the data type of the field or from the data element, or to define them individually for a UI. In the SAP S/4HANA programming model, field labels also can be derived from data elements or can be directly defined in the UI. Additionally, it's possible to define a field label by a field annotation in a CDS view: `@EndUserText.label: '<field label>'`. An annotated field label can be translated like other short texts.

The field label from the data element guarantees identical field labels for all usages of the data field. This avoids multiple translations and ensures consistent terminology in all UIs and apps with that data field.

Sometimes a UI has to use an individual field label. This always takes precedence over a field label from the data element of an annotation. The logic behind when the data element and when an annotation is used is more complex and will be presented in Section 6.2.1. Next, Section 6.2.2 covers variants of labels with different text lengths.

6.2.1 Determination of a Field Label

Data element or annotation?

In a CDS view, a field label can be determined by a data element or by an annotation. In simple cases, an annotation takes precedence over a data element. The situation is more complex if the field's data type is changed to a data element by a `cast` function or if the annotation is propagated from a data source. Table 6.1 shows a stack of views that demonstrates the precise rules.

View	Annotation	Cast to	Data Element	Field Label
V5	–	–	D2	`C`
V4	–	D2	D2	`C`
V3	(propagated)	–	D1	`B`
V2	`@EndUserText.label: 'B'`	–	D1	`B`
V1	–	–	D1	`A`

Table 6.1 Determination of a Field Label

Table 6.1 Determination of a Field Label (Cont.)

The field labels are determined as follows:

1. Starting point is CDS view V1 with field F having data element D1 as the type. The data element defines the field label `A`.
2. View V2 selects field F from V1. In view V2, field F also has data element D1 as the type. However, in view V2, field F also has annotation `@EndUserText.label: 'B'` and therefore gets field label `B` as the annotation takes precedence over the data element.
3. In view V3 that selects field F from view V2, field F still has data element D1 but also field label `B` as the annotation is propagated to view V3.
4. In view V4 that selects field F from view V3, the data type of field F is changed to data element D2 by a `cast` function. Then field F gets field label `C` from data element D2. The `cast` has higher priority than a *propagated* annotation `@EndUserText.label`. In fact, propagation of this annotation is stopped by the `cast`, and it's not propagated to field F in view V4.

   An *explicit* annotation in view V4 would have taken precedence over the `cast` data element, however.
5. In view V5 that selects field F from view V4, the data element and the field label are kept.

The technical realization of this logic treats the field label of the data element like a propagated `@EndUserText.label` annotation. Therefore, the development environment shows for all these views an active annotation `@EndUserText.label` with the respective field label. You can verify this with the **Active Annotations** function from the context menu of the CDS view in the ABAP Development Tools (ADT).

6.2.2 Length of a Field Label

Text variants of a field label

A data element can cover many situations as it offers field labels in three lengths as well as a label for column headings and a short description text.

Conversely, in a CDS view, only two label text variants can be defined by annotations:

- The field label by `@EndUserText.label`
- A short description called *quick info* by `@EndUserText.quickInfo`

The UI technology and the communication channel to the UI determine which of these text variants can be used in a UI. Most SAP Fiori apps use the OData protocol for their communication. The OData standard offers three variants of field labels: `label`, `quickInfo`, and `heading`. The first two labels exactly correspond to the CDS annotations. This isn't by chance: several CDS annotations were introduced to correspond to OData annotations. The `heading` variant isn't available, however.

The selection of field labels for a data element is more complex. The mapping between the data element **Short Description** and `quickInfo` is clear, as is the correlation between the data element **Heading** and `heading`. More difficult, however, is the choice between the **Short**, **Medium**, or **Long** texts for the `label`. Figure 6.4 shows an example of a typical data element.

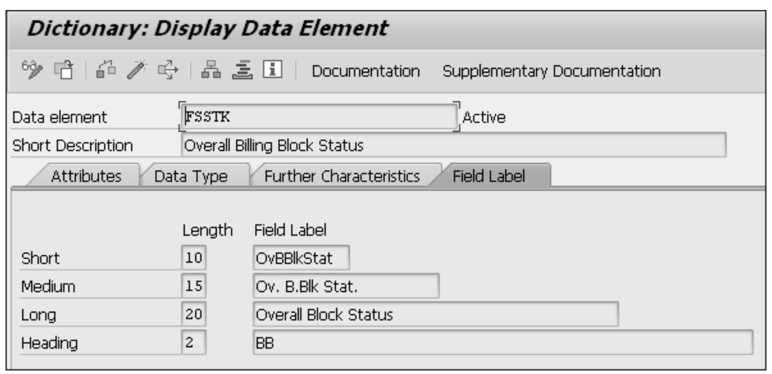


Figure 6.4 Field Label Texts of a Data Element

Field labels from data elements

In earlier releases, the SAP Gateway component had always preferred the *medium text* of the data element as the field label in the OData service, even if its length was quite short. This often led to cryptic abbreviations. On the other hand, field labels should not be too long. Therefore, the logic was

adapted for SAP S/4HANA and now selects the longest field label with a maximum length of 20 characters. With that approach, most cases can be handled. Only in some cases, labels with length 20 are still too short to express the field semantics properly. An alternative is to use CDS annotations that support distinctly longer label texts.

It's also possible to define a data element without short or medium label texts, as shown in Figure 6.5. Then the single remaining text is used as the field label. In this approach, 40 characters are available for the label text.

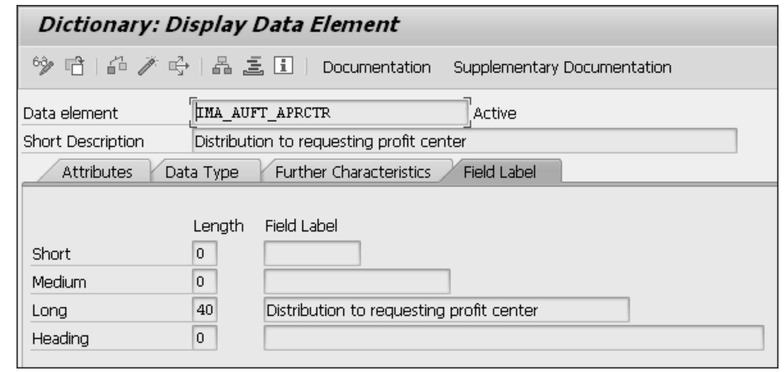


Figure 6.5 Data Element with a Long Field Label

Field labels in OData and analytics

All three variants of OData field labels are provided by the application infrastructure in the metadata of an OData service. Analytic applications support a single field label only. This is determined like the label text in OData and either taken from annotation `@EndUserText.label` or from the data element according to the same logic.

Parameter labels

CDS view parameters also have labels. Like field labels, they can be taken from the data element of the parameter, according to the same logic, or defined by annotation `@EndUserText`. There is no propagation logic for parameter annotations, however.

Parameter Annotations

Like fields, you can also annotate parameters of CDS views. You've already seen an example of this in our initial discussion in Chapter 3, and more examples will follow when we look at analytical application modeling in Chapter 8.

Besides the labels, you can define further language-dependent texts in CDS views, such as a label for the CDS view itself, or some UI texts by UI annotations. However, these texts can't be derived from data elements—only from CDS view annotations.

6.3 Field Semantics

Usually, developers specify the technical type of a data field. This information is far from sufficient, however, to express all business semantics of the field and execute functions for it that depend on these semantics. The field documentation, which often doesn't exist, addresses human readers and is too informal to be a reliable source for automated processing.

Formalized semantics

To address this problem, the ABAP Data Dictionary (ABAP DDIC) introduced formalized descriptions of some semantic aspects of fields in addition to the data type, for example, to identify amount fields and their related currency. These enable automated processing by the ABAP infrastructure.

CDS takes several steps forward. Annotations at CDS view fields can formalize any semantic aspect of a data field. This method is applied in various ways, and you'll see many examples throughout this book. In this section, we'll present several frequently used field annotations.

6.3.1 Quantities and Amounts

Unit and currency

Every quantity field has a unit field, and every amount field has a currency field. The ABAP DDIC can store these semantic properties and relations for tables and structures. For CDS views, they can be expressed as the following CDS annotations:

- **Unit field**
  A unit field is characterized by annotation `@Semantics.unitOfMeasure: true`.
- **Quantity field**
  When indicating a quantity field, the related unit field is provided as `@Semantics.quantity.unitOfMeasure: '<unit field>'`.
- **Currency field**
  A currency field is characterized by `@Semantics.currencyCode: true`.
- **Amount field**
  For amount fields, again the related currency field is indicated as `@Semantics.amount.currencyCode: '<currency field>'`.

By specifying the reference field at the quantity or amount field, multiple fields can reference the same unit or currency field.

Examples

Listing 6.1 shows some examples for these annotations from SAP standard view `I_SalesOrderItem`.

```
@Semantics.unitOfMeasure: true
OrderQuantityUnit,
@Semantics.quantity.unitOfMeasure: 'OrderQuantityUnit'
OrderQuantity,
@Semantics.amount.currencyCode: 'TransactionCurrency'
NetAmount,
@Semantics.currencyCode: true
TransactionCurrency,
```

Listing 6.1 Quantity and Unit, Amount and Currency

Service metadata

The CDS annotations for quantities, units, amounts, and currencies are translated by the application infrastructure into analogous annotations of an OData service. They are available to consumers of the OData service. Analytic applications leverage these CDS annotations as well, as you'll see in Chapter 8.

6.3.2 Aggregation Behavior

SQL `SELECTS` from CDS views can explicitly specify the desired *aggregation* of a data field. This is done for the following purposes:

- For numeric fields, a *summation* or an *average calculation*
- For sortable fields, the determination of a *maximum* or a *minimum*
- For any fields, the *counting* of different values

Standard aggregation

Some data fields in a CDS view are often aggregated and have a preferred method of aggregation. The net amounts of sales order items, for example, are usually summed up but not the net prices. For net prices, sometimes a minimum of maximum determination is useful, but mostly they serve as additional information. You can assign to such fields a *standard aggregation* that fits its semantics. This enables the infrastructure to

request aggregated data without explicitly specifying the type of aggregation. By default, fields are aggregated according to their annotated standard aggregation.

Aggregating selection

Assume, for example, that field `NetAmount` in the sales order items view is annotated for standard aggregation `summation`. In this case, an aggregating selection of the business field division and the net amount results in a list of all divisions together with the sum of the net amounts of all sales order items with that division. Such a result is already a simple analysis of business data.

Two annotation versions

Due to a change in the taxonomy for CDS annotations, there are two variants for specifying a standard aggregation: the older variant `@DefaultAggregation` and the current variant `@Aggregation.default`. Possible types for the standard aggregation are shown in Table 6.2.

Aggregation Type	Description
#AVG	Average calculation: Sum of all values, divided by the number of values
#COUNT_DISTINCT	Number of distinct values
#FORMULA	Special form for analytic queries (see Chapter 8 for more details)
#MAX	Maximum of all values
#MIN	Minimum of all values
#NONE	No standard aggregation
#SUM	Sum of all values

Table 6.2 Supported Aggregation Types

Listing 6.2 shows some examples of this annotation from SAP standard view `I_SalesOrderItem`.

```
@DefaultAggregation: #SUM
OrderQuantity,
@DefaultAggregation: #SUM
NetAmount,
@DefaultAggregation: #NONE
NetPriceAmount,
```

Listing 6.2 Standard Aggregation

No aggregation

Fields without a standard aggregation annotation won't be aggregated. This is equivalent to standard aggregation `#NONE`. This annotation at field `NetPriceAmount` in view `I_SalesOrderItem` (see Listing 6.2) therefore isn't necessary. The developers nevertheless annotated this field to emphasize that it should not be aggregated.

Measures and aggregation

The annotation for a standard aggregation (except `#NONE`) has a big impact on the consumers of a view. In analytic views, this leads to the interpretation of the annotated field as an *analytic measure*. You'll find more details on this in Chapter 8.

In the metadata of an OData version 2.0 service based on a CDS view, a field with annotated standard aggregation is marked as a measure. A read request for the entity set corresponding to the CDS view is performed as an aggregating selection. The type of aggregation is determined by the annotated standard aggregation. For executing the request, the application infrastructure uses a SQL `SELECT` with the standard aggregation of the annotated fields, grouped by the other requested fields. The aggregation is finally executed by SAP HANA. With this method, OData services can be leveraged for simple analytics.

6.3.3 System Times

Creation time and last change time

Applications usually store the creation date/time of a data record, as well as the date it was last changed, in database tables together with the application data. This is important semantic information. The point in time of the last change, for example, can be used in data replication or for optimistic locking mechanisms. A prerequisite is the reliable determination and storage of this information at creation and at every change. CDS annotations can be used to identify fields with this information as system times (see Table 6.3).

Annotation	Field Semantics
`@Semantics.systemDateTime.createdAt`	Point in time of creation (ABAP type `TIMESTAMP` or `TIMESTAMPL`)
`@Semantics.systemDateTime.lastChangedAt`	Point in time of the last change (ABAP type `TIMESTAMP` or `TIMESTAMPL`)

Table 6.3 CDS Annotations for System Times

Annotation	Field Semantics
@Semantics.systemDate.createdAt	Date of creation (ABAP type DATS)
@Semantics.systemDate.lastChangedAt	Date of last change (ABAP type DATS)
@Semantics.systemTime.createdAt	Clock time of creation (ABAP type TIMS); only reasonable in combination with a creation date
@Semantics.systemTime.lastChangedAt	Clock time of the last change (ABAP type TIMS); only reasonable in combination with a last change date

Table 6.3 CDS Annotations for System Times (Cont.)

System times in CDS views

Fields with system times not only exist in tables but also in CDS views. For views that combine data from multiple tables, you must carefully consider which fields should be annotated as system times using the following rules:

- Only a single point in time of creation is reasonable; it should relate to the main data source of the view.
- The point in time of the last change must consider possible changes of *all* data fields of the CDS view, independent of their origin.

Listing 6.3 shows examples of system times from SAP standard view I_SalesOrder.

```
@Semantics.systemDate.createdAt: true
CreationDate,
@Semantics.systemTime.createdAt: true
CreationTime,
@Semantics.systemDate.lastChangedAt: true
LastChangeDate,
@Semantics.systemDateTime.lastChangedAt: true
LastChangeDateTime,
```

Listing 6.3 System Times

6.3.4 Text and Languages

Texts in natural language

Fields that contain a text in natural language should be distinguishable from codes or other technical information. They should preferably be displayed to a human user, but they are usually irrelevant for technical processing. For this purpose, CDS annotation @Semantics.text is available.

Natural language texts are usually written in a certain language; there are a few exceptions, such as names of humans or organizations. If this language is provided in another field of a view, this field is annotated with @Semantics.language. An explicit connection between the fields, such as quantities or amounts, isn't possible, however. Usually, all text fields of a view share the same language.

Listing 6.4 shows examples for this from SAP standard view I_CountryText. This is a language-dependent text view that we'll introduce in Section 6.5.

```
@Semantics.language
key spras as Language,
@Semantics.text: true
cast(landx50 as fis_landx50 preserving type ) as CountryName,
```

Listing 6.4 Annotations for Text and Language

6.3.5 Information for the Fiscal Year

Fiscal year

CDS annotations can also express application-specific semantics. An example is the information for a *fiscal year* in financial accounting and its periods. A fiscal year can deviate from a calendar year and consists of flexibly defined periods. The indication of fiscal year information enables an appropriate formatting of the data. The annotations are shown in Table 6.4.

Annotation	Field Semantics
@Semantics.fiscal.yearVariant	Fiscal year variant; defines the properties of the fiscal year
@Semantics.fiscal.period	Fiscal period given by three digits
@Semantics.fiscal.year	Fiscal year given by four digits

Table 6.4 Information for the Fiscal Year

Annotation	Field Semantics
`@Semantics.fiscal.yearPeriod`	Fiscal year period; the combination of fiscal year and period
`@Semantics.fiscal.quarter`	Fiscal quarter given by one digit
`@Semantics.fiscal.yearQuarter`	Combination of fiscal year and quarter
`@Semantics.fiscal.week`	Fiscal week given by two digits
`@Semantics.fiscal.yearWeek`	Combination of fiscal year and week
`@Semantics.fiscal.dayOfYear`	Number of a day in a fiscal year

Table 6.4 Information for the Fiscal Year (Cont.)

Listing 6.5 shows examples from SAP standard view `I_JournalEntryItem`.

```
@Semantics.fiscal.year: true
ryear as LedgerFiscalYear,
@Semantics.fiscal.period: true
poper as FiscalPeriod,
@Semantics.fiscal.yearVariant: true
periv as FiscalYearVariant,
@Semantics.fiscal.yearPeriod: true
fiscyearper as FiscalYearPeriod,
```

Listing 6.5 Information for the Fiscal Year

6.4 Foreign Key Relations

You may be familiar with the concept of *foreign keys* from the ABAP DDIC. The basic target is to restrict the possible values for a data field of a *foreign key table* to the available values of a key field in a *check table*. This may sound complex but quickly becomes clear with an example (see Figure 6.6).

In the example, the data record for an address has field `Country` for a country. Only countries from a country table should be allowed as admissible values for this field. Therefore, the address table plays the role of foreign key table and the `Country` field is the foreign key field. The table of countries plays the role of check table with the `Country` as key field. This relation

between the tables and the relevant fields can be stored as a foreign key in the ABAP DDIC.

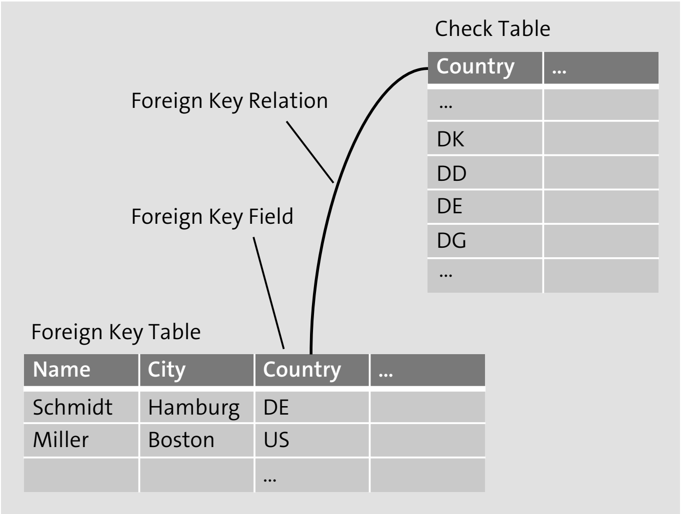


Figure 6.6 Foreign Keys in the ABAP DDIC

Foreign keys in CDS

Foreign key relations can exist between CDS views as well. The relationship between the *foreign key view* and the *value view* or *entity view* can be defined neatly by an association. In CDS, we avoid the term "check view" because the foreign key relationship alone doesn't define a consistency check but only expresses the semantic relationship between the persisted data. The entity view provides possible field values or, more precisely, represents the list of instances to which the *foreign key field* can reference (see Figure 6.7).

Listing 6.6 shows an example of such an association from view `I_ProfitCenter` to view `I_Country`.

```
define view I_Country as select from t005
{ key cast(land1 as land1_gp preserving type ) as Country,
  …

define view I_ProfitCenter  as select distinct from cepc
association[0..1] to I_Country as _Country
  on $projection.Country = _Country.Country
```

```
{ …
  land1 as          Country,
  …
```

Listing 6.6 Association Used by a Foreign Key Relation

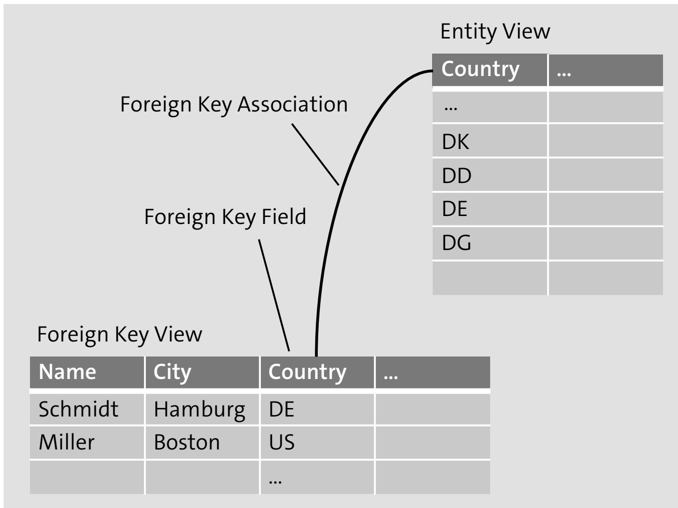


Figure 6.7 Foreign Keys in CDS

Foreign key annotation

The definition of an association doesn't yet establish a foreign key relation. Its foreign key character, as well as the identification of the foreign key field, is defined by CDS annotation `@ObjectModel.foreignKey.association` at the foreign key field (see Listing 6.7). The association given in the annotation is called the *foreign key association* for this field.

```
define view I_ProfitCenter  as select distinct from cepc
association[0..1] to I_Country as _Country
  on $projection.Country = _Country.Country
{ …
  @ObjectModel.foreignKey.association: '_Country'
  land1 as          Country,
  …
```

Listing 6.7 Annotation of a Foreign Key Association

Cardinality

The cardinality of a foreign key association must be either `[0..1]` or `[1..1]`; that is, either at most one country or exactly one country exists for a profit center. The cardinality for the reverse direction of the association can be `[0..*]`, meaning that for a country, there can be no profit centers or any number of profit centers.

Multiple key fields

This way of defining a foreign key association also works for entity views with more than one key field. Listing 6.8 shows such an example for regions (federal states, provinces, etc.).


```
define view I_Region  as select from t005s
  association [1..1] to I_Country as _Country
    on  $projection.Country = _Country.Country
{
  @ObjectModel.foreignKey.association: '_Country'
  key t005s.land1 as Country,
  key t005s.bland as Region,
  …
}

define view I_ProfitCenter  as select distinct from cepc
association[0..1] to I_Country as _Country
  on $projection.Country = _Country.Country
association[0..1] to I_Region as _Region
  on     $projection.Country = _Region.Country
    and  $projection.Region  = _Region.Region
{ …
  @ObjectModel.foreignKey.association: '_Country'
  land1 as          Country,
  @ObjectModel.foreignKey.association: '_Region'
  regio as          Region,
  …
}
```

Listing 6.8 Foreign Key Association with Two Key Fields

Figure 6.8 shows how the three views from the listings are connected by foreign key associations.

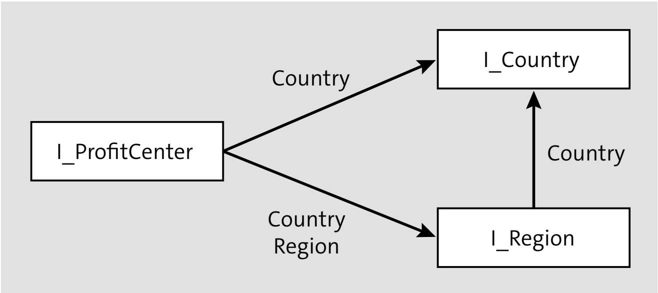


Figure 6.8 Foreign Key Associations

Representative key field

For CDS, the foreign key concept of ABAP DDIC was enhanced by a useful aspect: the indication of a *representative key field* by annotation `@ObjectModel.representativeKey`. The motivation for this can be seen in the preceding example. In Listing 6.8, it's clear from the semantics and the naming that field `Country` has association `_Country` as the foreign key association and not association `_Region`.

From a technical perspective, you could also specify association `_Region` at field `Country` as the foreign key association. This has the following disadvantages, however:

- Not all countries are available as values but only those for which regions are recorded.
- For a consumer of the foreign key view, it's not possible to retrieve further information about the country via the foreign key association.

This second point, in particular, is an import benefit of the data model: following an association reveals further details of the starting point, that is, the foreign key field. Foreign key associations in CDS are meant to serve this purpose and retrieve further data of the entity represented by the foreign key field.

Consistency condition

By indicating the representative key field of an entity view, usage of a wrong association as the foreign key association can be detected. The foreign key association always must bind the foreign key field to the representative key field of the entity view; that is, the two fields must be equal in the `ON` condition of the association definition. In a consistent modeling, both annotations must be in place.

Listing 6.9 shows annotation `@ObjectModel.representativeKey` in views `I_Country` and `I_Region`.

```
@ObjectModel.representativeKey: 'Country'
define view I_Country as select from t005
{ key cast(land1 as land1_gp preserving type ) as Country,
  …

@ObjectModel.representativeKey: 'Region'
define view I_Region  as select from t005s
  association [1..1] to I_Country as _Country
    on  $projection.Country = _Country.Country
{
  @ObjectModel.foreignKey.association: '_Country'
  key t005s.land1 as Country,
  key t005s.bland as Region,
  …
}
```

Listing 6.9 Representative Key Fields

The representative key field of an entity view is the part of the key that semantically represents an entity (row) of the view. In the view of regions, this is key field `Region`, not key field `Country`. Because the view name also reflects the semantics of the entity, the name of the representative key and the view name are usually very similar.

Modeling pattern

Not every CDS view needs a representative key field. Views with a representative key field represent discrete entities that can be connected by foreign keys to enrich a data model semantically. Foreign key relations are an elementary *modeling pattern* in CDS.

Foreign key associations are used to do the following:

- Serve as standard source of value help if no other explicit value help is defined (see Chapter 10, Section 10.1).
- Define dimensions in the analytic model (see Chapter 8, Section 8.2.4).
- Mark the view that provides detailed information for a data field.
- Provide automated input checks in transactional applications.

6.5 Text Relations

Many data fields contain codes or IDs to represent an entity of the business world, for example, a country code or customer ID. While computers are very good at processing such codes, a human consumer wants to see a name or description in natural language. Such texts are stored in different fields or even different tables, so a connection between the coded field and the text field must be created in the data model, also known as a *text relation*. This enables the infrastructure to automatically detect and use the texts. If texts exist in multiple languages, the logon language of the user, or an appropriate substitute, would be chosen for filtering the text to be displayed.

CDS supports two variants of text relations: within a view or between two views. Both variants are based on annotations: `@ObjectModel.text.element` for within a view or `@ObjectModel.text.association` for between views.

Text relation within a view

You can see an example for the first variant in SAP standard view `I_Bank` for banks. A relevant extract is shown in Listing 6.10.

```
define view I_Bank as select from
   bnka
{ …
  @ObjectModel.foreignKey.association: '_Country'
  key banks as BankCountry,
  @ObjectModel.text.element: [ 'BankName' ]
  key bankl as BankInternalID,
      @Semantics.text: true
      banka as BankName,
  …
}
```

Listing 6.10 Text Relation within a View

In this example, `BankInternalID` is a coded representation of a bank, and `BankName` is a text field for the name of the bank. In addition, take note of field annotation `@Semantics.text` from Section 6.3.4. Annotation `@ObjectModel.text.element` at the field with the coded representation references a list of fields that contain a descriptive text. If the list contains text fields, the first one is used as standard text. Note that this variant doesn't support language-dependent texts.

Text relations between views

An example for the second variant is the view for countries, `I_Country`, and its text view, `I_CountryText`, with the related country names in different languages. Listing 6.11 shows the relevant parts.

```
define view I_Country as select from t005
  association [0..*] to I_CountryText as _Text
    on $projection.Country = _Text.Country
{ @ObjectModel.text.association: '_Text'
  key cast(land1 as land1_gp preserving type ) as Country,
  …
}

@ObjectModel.dataCategory: #TEXT
@ObjectModel.representativeKey: 'Country'
define view I_CountryText as select from t005t
{ key land1 as Country,
  @Semantics.language: true
  key spras as Language,
  @Semantics.text: true
  cast(landx50 as fis_landx50 preserving type )
    as CountryName,
  …
}
```

Listing 6.11 Text Relation to a Text View

Text association

View `I_Country` has coded field `Country` and its text—actually, its name—`CountryName` is language dependent and stored in view `I_CountryText`. The relation between the views is established by association `_Text`. Annotation `@ObjectModel.text.association: '_Text'` characterizes the association as *text association* for field `Country`, and the description of this field can be found in the associated view.

In associated view `I_CountryText`, field `CountryName` is annotated as a text field and therefore used as the description. If there are multiple text fields in the view, the first text field will be the standard text.

Text views

Views that essentially contain only textual descriptions should be marked as *text views* by view annotation `@ObjectModel.dataCategory: #TEXT`.

To clarify which field the text view provides the text for, one of the view's key fields is marked as the *representative key field* by annotation

@ObjectModel.representativeKey. This clearly defines the semantics of the view in case there are multiple key fields.

Text associations always bind the annotated field in the source view with the representative key field of the text view. Usually a text view is language dependent. The field with the language of the texts must be a key field and must be annotated with @Semantics.language: true. The infrastructure usually filters on the logon language of the user when retrieving a language-dependent text.

Use the foreign key association

View I_ProfitCenter from the preceding section (see Listing 6.7) doesn't have a text association for its field Country. Still, the infrastructure can automatically determine a text field for it by first following the foreign key association to view I_Country. Its representative key field Country has a text association that leads to a text (see Figure 6.9).

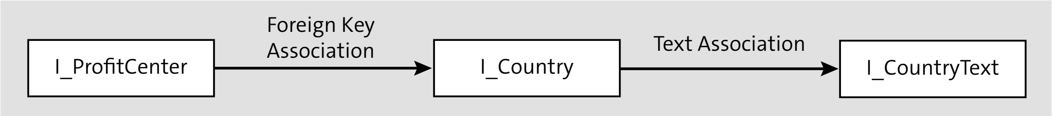


Figure 6.9 Foreign Key and Text Association

6.6 Composition Relations

Many types of related business data are distributed in a normalized form to multiple tables, for example, the header data and items of a sales or purchase order. CDS views for analytics often combine such data into a single view, but CDS views for transactional applications reflect a normalized distribution of data to multiple views. In the data model, we want to express which views belong together and form a well-defined (business) object.

Parent-child relations

For this purpose, *composition relations* are introduced that bring the related views into hierarchical parent-child relations. A view can have at most one superordinate *parent view* but any number of subordinate *child views*. A composition relation implies an existential dependency, meaning that a data row in a child view can only be created if a related row in its parent view exists, and the deletion of a data row forces the deletion of all related rows in child views.

Root view

Besides the parent-child relations, for every group of related views, one view is marked as a *root view*. This is the only view in the group without a

parent view. In the sales order object, for example, the view for the order header data is the root view.

Object

The group of related views is often called the *object* or *business object*. It can be identified by its root view.

Composition associations

Composition relations between views are defined as usual by associations. Annotation @ObjectModel.association.type assigns a type to the associations that marks them as *composition associations*. Three types of composition associations are distinguished:

- **#TO_COMPOSITION_PARENT**
  The association points to the parent view.
- **#TO_COMPOSITION_CHILD**
  The association points to a child view.
- **#TO_COMPOSITION_ROOT**
  The association points to the root view of the object.

An association can simultaneously have types #TO_COMPOSITION_PARENT and #TO_COMPOSITION_ROOT. The root view is identified by annotation @ObjectModel.compositionRoot: true.

Sales order example

The sales order model of SAP consists of CDS views I_SalesOrder for the order header, I_SalesOrderItem for the items, and I_SalesOrderScheduleLine for the schedule lines. It has the composition associations shown in Figure 6.10.

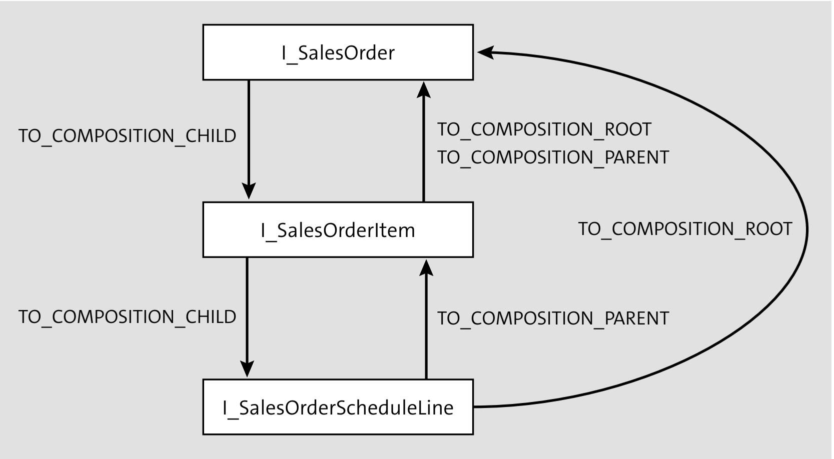


Figure 6.10 Composition Associations

The relevant parts of the view definitions are shown in Listing 6.12.

```
@ObjectModel.compositionRoot: true
define view I_SalesOrder …
  association [0..*] to I_SalesOrderItem as _Item
    on $projection.SalesOrder = _Item.SalesOrder
{ …
  @ObjectModel.association.type: [#TO_COMPOSITION_CHILD]
  _Item,
  …
}

define view I_SalesOrderItem …
  association [1..1] to I_SalesOrder as _SalesOrder
    on $projection.SalesOrder = _SalesOrder.SalesOrder
  association [0..*] to I_SalesOrderScheduleLine
    as _ScheduleLine
    on    $projection.SalesOrder = _ScheduleLine.SalesOrder
      and $projection.SalesOrderItem =
                               _ScheduleLine.SalesOrderItem
{ …
  @ObjectModel.association.type:
    [#TO_COMPOSITION_PARENT, #TO_COMPOSITION_ROOT]
  _SalesOrder,
  @ObjectModel.association.type: [#TO_COMPOSITION_CHILD]
  _ScheduleLine,
  …
}

define view I_SalesOrderScheduleLine …
  association [1..1] to I_SalesOrder as _SalesOrder
    on $projection.SalesOrder = _SalesOrder.SalesOrder
  association [1..1] to I_SalesOrderItem as _SalesOrderItem
    on    $projection.SalesOrderItem =
                               _SalesOrderItem.SalesOrderItem
      and $projection.SalesOrder = _SalesOrderItem.SalesOrder
{ …
  @ObjectModel.association.type: [#TO_COMPOSITION_ROOT]
  _SalesOrder,
  @ObjectModel.association.type: [#TO_COMPOSITION_PARENT]
```



```
  _SalesOrderItem,
  …
}
```

Listing 6.12 Views with Composition Relations

In Chapter 9, Section 9.3, you'll see in more detail how compositions are used.

6.7 Time-Dependent Data

Time-dependent data or *temporal data* in business applications are mainly attributes of master data objects that change their value over time, usually at a certain calendar day. Typical examples are the salary of an employee or the value-added tax percentage of a country.

Business time-dependency

This is a *business*-driven planned time-dependency, not a versioning of data specifying the point in time of a change. That second versioning is called *system time-dependency* and is usually applied for change-tracking or revision purposes. Business time-dependency is usually described by dates and therefore has the granularity of days. Time-dependent data is usually stored in separate tables that have an additional key field for the time dimension.

Modeling pattern

With CDS views, business time-dependent data can be described by a common modeling pattern as follows:

- A *business time-dependent view* has two date fields for the validity period of the time-dependent attributes. These date fields are annotated with `@Semantics.businessDate.from: true` and `@Semantics.businessDate.to: true`, respectively.
- The validity period comprises the `from` date and the `to` date.
- At least one of the date fields is part of the view's key. The remaining fields of the key are called *entity keys*.
- The validity periods of two view rows with the same entity key must not overlap.
- The combined timeline of validity periods of all rows with the same entity key can have multiple gaps between periods.

Both date fields should be stored on the database for efficient access.

[!]

Annotate Only Time-Dependent Views

When using annotations `@Semantics.businessDate.from` and `@Semantics.businessDate.to`, take care that the view really has all qualities of a business time-dependent view.

The infrastructure recognizes a business time-dependent view at its annotations and key. When processing the view data, it can use an appropriate key date as the filter in this case.

Example

Listing 6.13 shows how view `I_CostCenter` is annotated as a business time-dependent view.

```
define view I_CostCenter as select …
{ key kokrs  as ControllingArea,
  key kostl  as CostCenter,
      @Semantics.businessDate.to: true
  key datbi  as ValidityEndDate,
      @Semantics.businessDate.from: true
      datab  as ValidityStartDate,
  …
}
```

Listing 6.13 Business Time-Dependent View

6.8 Hierarchies

Business data is often structured hierarchically; for example, there are hierarchies of organization units, reporting hierarchies of employees, hierarchies of financial accounts in the balance sheet, hierarchies of products and product types, and many more. Hierarchical structures help users survey and process large amounts of data. In analytical applications, measures can be aggregated at hierarchy nodes and selectively expanded to greater detail.

Hierarchies in CDS

You can formalize hierarchical relationships between data objects in CDS and model them with views and annotations. This model can be processed by a generic hierarchy engine. Two basic types of hierarchies are distinguished: *leveled hierarchies* and *parent-child hierarchies*. In the following sections, we'll go through both types, walk through an example, and discuss how to determine and test a hierarchy.

6.8.1 Leveled Hierarchies and Parent-Child Hierarchies

Leveled hierarchies

In *leveled hierarchies* every hierarchy level has a separate data field. A simple example is the two-level hierarchy of countries and regions, which is represented by view `I_Region` with key fields `Country` und `Region` for the levels. As explicit fields are available for every hierarchy level, standard SQL requests are sufficient for executing typical hierarchy requests such as aggregating all value in a subtree. You can, for example, determine all regions of a country via a simple SQL request and aggregate their values. Special hierarchy functions aren't necessary. Therefore, we'll focus on a different type of hierarchy, the parent-child hierarchy.

Parent-child hierarchies

Parent-child hierarchies have an important advantage over leveled hierarchies: the hierarchy levels are independent from the view's fields and their number is theoretically unlimited. Figure 6.11 shows an example.

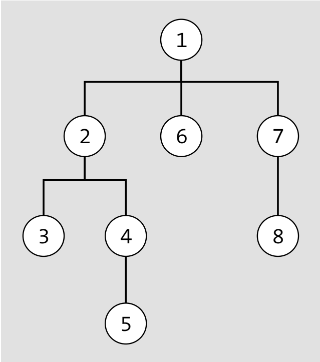


Figure 6.11 Parent-Child Hierarchy

This unlimited number of levels requires programming loops or recursive processing of the hierarchy. Unfortunately, this isn't possible with standard requests in SQL. For using parent-child hierarchies, additional support by the infrastructure is necessary. Until ABAP release 7.52, only the analytic engine was available for that purpose (refer to Section 6.1). In ABAP release 7.53, the first steps were made to leverage the SAP HANA hierarchy engine in CDS. However, as this approach isn't yet completely integrated into the

SAP S/4HANA programming model and not much practical experience is available yet, we focus on the annotation-based parent-child hierarchies and their processing by the analytic engine.

Base entity and hierarchy nodes

The starting points for defining a hierarchy are the *hierarchy base entity* instances, for example, employees or cost centers, which will be structured hierarchically. For that purpose, *hierarchy nodes* are used. These nodes can belong to the base entity (e.g., in an employee-manager hierarchy) or to an entity with different business semantics, or the hierarchy can be structured as a pure node.

The hierarchy nodes get their structure by a relation to a parent node. Every node is either related to a single parent node or has no parent. Nodes without parent are called *root nodes*. The parent relation defines a parent-child hierarchy on the hierarchy nodes. The hierarchical structure on the base entity originates from the assignment of hierarchy nodes to (at most) one instance of the base entity. The base entity instance obtains its position in the hierarchy from this assignment.

Distinguishing between the base entity and the hierarchy nodes allows for the definition of multiple different hierarchies that consist of different sets of nodes for the same base entity. You can use these in parallel for different purposes. The different hierarchies are administered by a *hierarchy directory*.

CDS views for hierarchies

In CDS, the following views for hierarchies are used:

- A view for the hierarchy base entity
- A view for hierarchy nodes
- Optionally views for other entities that are represented as nodes in the hierarchy
- An optional view for the hierarchy directory
- Optional text views for the base entity, for the hierarchy nodes, for other entities, and for the hierarchy directory

These views are connected by associations, as shown in Figure 6.12. The types of the associations are shown as well.

In simple cases, the same view represents the base entity and the hierarchy nodes, and there are no other views. The employee-manager is an example. The relationship of the hierarchy base entity to its hierarchy node view is

established by an association. It's modeled by annotation `@ObjectModel.hierarchy.association` at the representative key field of the base entity.

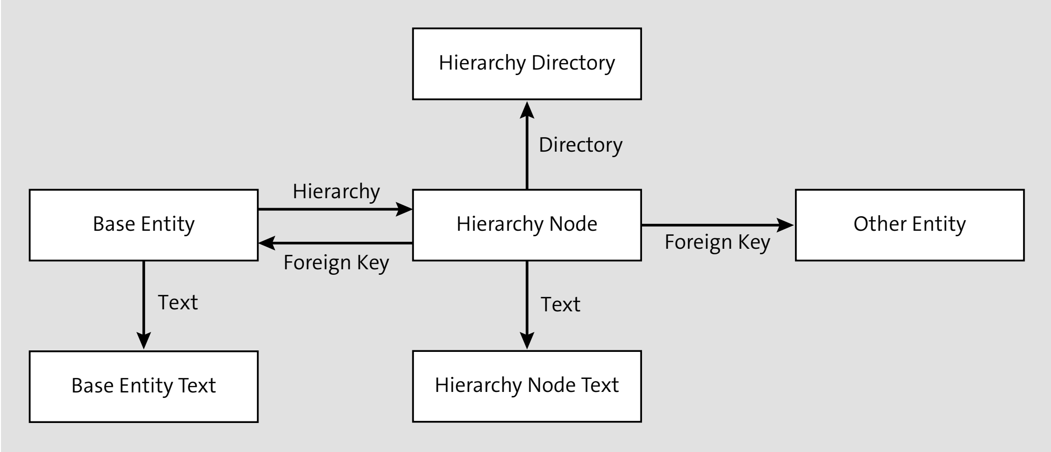


Figure 6.12 CDS Views for Hierarchies

Hierarchy structure

The essential information for the definition of the hierarchy structure is the view of the hierarchy nodes. If it's not identical to the view of the base entity, it's marked as a *hierarchy node view* by annotation `@ObjectModel.dataCategory: #HIERARCHY`. Structured view annotation `@Hierarchy.parentChild` defines the hierarchy structure. The details are shown in Table 6.5.

Annotation	Semantics
`@Hierarchy.parentChild.name`	Specifies a technical name for the hierarchy. It's mandatory if no hierarchy directory is used.
`@Hierarchy.parentChild.label`	Description of the hierarchy (optional).
`@Hierarchy.parentChild.recurseBy`	Used as an alternative to annotation `recurse.parent` and `recurse.child`. The relation to the parent node is defined by an association of the view to the view itself, a self-association.

Table 6.5 Annotations for Defining the Hierarchy Structure

Annotation	Semantics
`@Hierarchy.parentChild.recurse.parent` `@Hierarchy.parentChild.recurse.child`	Used as an alternative to annotation `recurseBy`. The relation to the parent node is defined by specifying corresponding fields.
`@Hierarchy.parentChild.siblingsOrder.by`	Specifies a field that is used to define the order of siblings.
`@Hierarchy.parentChild.siblingsOrder.direction`	Direction for ordering siblings; can be `'ASC'` (ascending) or `'DESC'` (descending), with `'ASC'` as default.
`@Hierarchy.parentChild.directory`	Association to the hierarchy directory (optional).

Table 6.5 Annotations for Defining the Hierarchy Structure (Cont.)

Relation to the parent node

The most important information is the relationship to a parent node. For its definition, using a self-association in annotation `@Hierarchy.parentChild.recurseBy` is most elegant. It must have cardinality `[0..1]` and bind the key fields of the target. Alternatively, the key fields of the view are specified after annotation `@Hierarchy.parentChild.recurse.child`, and the view fields that identify the parent node are specified after annotation `@Hierarchy.parentChild.recurse.parent` in a sequence corresponding to the key fields.

6.8.2 Example of a Parent-Child Hierarchy

Cost center hierarchy

Now, we'll use the cost center hierarchy as an example. Listing 6.14 shows the relevant part of base entity `I_CostCenter`. Note association `_CostCenterHierarchyNode` to the hierarchy node view.

```
@ObjectModel.representativeKey: 'CostCenter'
define view I_CostCenter as select …
association[0..*] to I_CostCenterText  as _Text
  on  $projection.ControllingArea    = _Text.ControllingArea
  and $projection.CostCenter         = _Text.CostCenter
  and $projection.ValidityEndDate    = _Text.ValidityEndDate
association[0..*] to I_CostCenterHierarchyNode
  as _CostCenterHierarchyNode
```



```
  on  $projection.ControllingArea    = … .ControllingArea
  and $projection.CostCenter         = … .CostCenter
{ key kokrs  as ControllingArea,
  @ObjectModel.text.association: '_Text'
  @ObjectModel.hierarchy.association:
    '_CostCenterHierarchyNode'
  key CostCenter,
  @Semantics.businessDate.to: true
  key ValidityEndDate,
  @Semantics.businessDate.from: true
  ValidityStartDate,
  …
}
```

Listing 6.14 Hierarchy Base Entity for the Cost Center

Example: Node view

Related node view `I_CostCenterHierarchyNode` is shown in Listing 6.15. Note that identical key fields of the child and parent node can be left out in the definition of the parent-child relation via `recurse`.

```
@ObjectModel.dataCategory: #HIERARCHY
@Hierarchy.parentChild:
{ recurse:          {   parent: 'ParentNode',
                        child:  'HierarchyNode'   },
  siblingsOrder:    {   by: 'SequenceNumber',
                        direction: 'ASC'   },
  directory:        '_Hierarchy'
}
define view I_CostCenterHierarchyNode …
  association [0..*] to I_CostCenterHierarchyNodeT as _Text
    on  $projection.CostCenterHierarchy =
                                  _Text.CostCenterHierarchy
    and $projection.HierarchyNode       = ….HierarchyNode
    and $projection.ControllingArea     = ….ControllingArea
    and $projection.CostCenter          = ''
  association [0..*] to I_CostCenter as _CostCenter
    on  $projection.CostCenter          =
                                    _CostCenter.CostCenter
    and $projection.ControllingArea     = ….ControllingArea
  association [1..1] to I_CostCenterHierarchy as _Hierarchy
    on  $projection.CostCenterHierarchy =
```

```
                              _Hierarchy.CostCenterHierarchy
    and $projection.ControllingArea     = ….ControllingArea
    and $projection.ValidityEndDate     = ….ValidityEndDate
  association [0..1] to I_ControllingArea as
                                          _ControllingArea
    on  $projection.ControllingArea     = ….ControllingArea
{
  @ObjectModel.foreignKey.association: '_ControllingArea'
  key ControllingArea,
  @ObjectModel.foreignKey.association: '_Hierarchy'
  key CostCenterHierarchy,
  @ObjectModel.text.association: '_Text'
  key HierarchyNode,
  key ValidityEndDate,
      ParentNode,
      ValidityStartDate,
      @ObjectModel.foreignKey.association: '_CostCenter'
      CostCenter,
      SequenceNumber,
      _Text,
      _CostCenter,
      _Hierarchy,
      _ControllingArea
      …
}
```

Listing 6.15 Hierarchy Node View of the Cost Center Hierarchy

Example: Hierarchy directory

The related hierarchy directory, view `I_CostCenterHierarchy`, is shown in Listing 6.16.

```
define view I_CostCenterHierarchy …
{ key ControllingArea,
  key CostCenterHierarchy,
      @Semantics.businessDate.to: true
  key ValidityEndDate,
      @Semantics.businessDate.from: true
      ValidityStartDate,
      …
}
```

Listing 6.16 Hierarchy Directory for Cost Center Hierarchies

Figure 6.13 shows the views of the example.

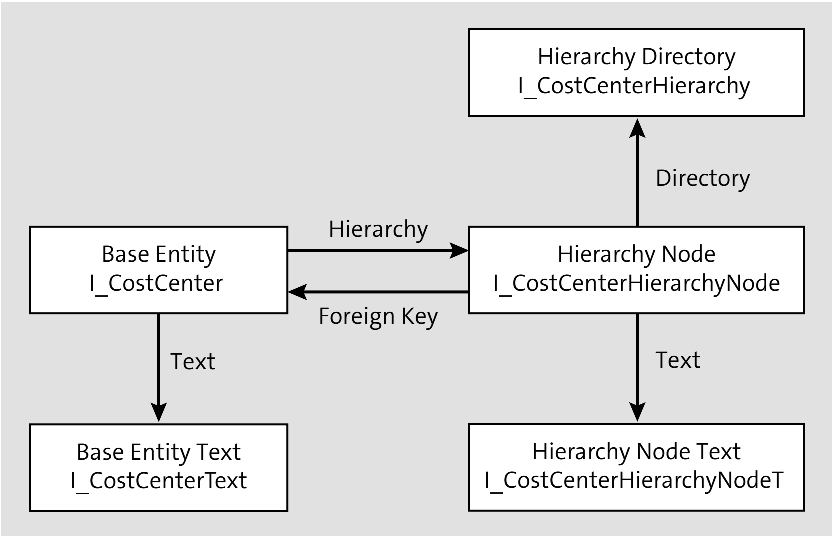


Figure 6.13 Hierarchy Example with Association Types

6.8.3 Determination of a Hierarchy

Evaluate the hierarchy definition

Using the view definitions, the infrastructure (or a developer) can derive how the hierarchy is formed. For the cost center hierarchy, this happens as follows:

1. In the hierarchy annotation of the node view, an association to a hierarchy directory is provided that must be considered. The user must therefore select a specific cost center hierarchy row from the directory.

   The hierarchy directory is time dependent (see Section 6.7), so the selection of the entry is done for a key date.
2. Now, all hierarchy nodes for the selected cost center hierarchy are read. When doing so, filters are used on fields `ControllingArea`, `CostCenterHierarchy`, and `ValidityEndDate` with the values of the selected cost center hierarchy, as on these fields the association to the hierarchy directory is defined.
3. All selected nodes have identical values on key fields `ControllingArea`, `CostCenterHierarchy`, and `ValidityEndDate`. Therefore, key field `HierarchyNode` is the effective key of these nodes.

As specified in the hierarchy annotation, data field `ParentNode` points to the parent node that is uniquely identified by `HierarchyNode` now. By this, the nodes are ordered in a hierarchical structure. The infrastructure creates an optimized hierarchy representation that enables the efficient processing of hierarchical requests.

4. In the next step, the individual cost centers (the instances of the base entity) must be assigned to the hierarchy nodes. This vital assignment is done according to a complex logic. For every single hierarchy node, a *node type* is determined. With the help of the node type, mixed hierarchies can be processed. These can have multiple types of entities as nodes, for example, cost centers, profit centers, and company codes, in the same hierarchy.

   Every possible node type is represented by a corresponding association from the hierarchy node view. For every node of the hierarchy and every association, the `ON` condition of the association definition is checked in sequential order according to their definition. The first association with a valid `ON` condition defines the type of the node, and no further associations are checked.

   In the node view of the cost center, the definition of the text association has additional condition `$projection.CostCenter = ''`. Therefore, all nodes with an initial cost center get node type `_Text`. All nodes with a not-initial cost center get the type of the association checked next, `_CostCenter`.

[!]

Hierarchy Definition Error

The association to the hierarchy directory is excluded from this logic, and the association to the controlling area doesn't play a role because it's defined last. If it were defined as the first association, all nodes would have gotten type `_ControllingArea`. All nodes would get the same text in the following step, the description of the controlling area, and the hierarchy functionality would not work as expected. This is a typical error source for hierarchy definitions.

5. Now, all nodes get a text because the node key alone isn't very meaningful. This text depends on the node type and is determined via the association belonging to the node type. For type `_Text`, this is standard behavior. For other node types, the association is expected to be a foreign key association. Then the text can be determined via the related entity. For the cost center, this is the related cost center text.

Now the hierarchy is sufficiently described for the infrastructure. A user interface can request the hierarchy in a form suitable for displaying from the infrastructure, drilling into child nodes, or retrieving sums calculated over subtrees for analytic applications.

Hierarchy infrastructure

Only the analytic engine as part of the ABAP infrastructure can process a hierarchy defined by CDS annotations. Hierarchies are commonly used in analytical applications. The use of hierarchies in transactional applications isn't yet fully supported.

6.8.4 Test a Hierarchy

Analytic test environment

The easiest way to test a hierarchy definition is the test environment for analytic views, Transaction RSRTS_ODP_DIS, in SAP GUI. Its use is explained in detail in Chapter 8, Section 8.2.2; therefore, only a short instruction follows here:

1. Ensure that the hierarchy base entity is marked as an analytic dimension view by annotation `@Analytics.dataCategory: #DIMENSION`.
2. Start Transaction RSRTS_ODP_DIS, and enter the SQL view name of the base entity, for example, "IFICOSTCENTER", in **ODP-Name** for the cost center view. Execute the transaction.
3. You'll see the analytic model of view `I_CostCenter` in Figure 6.14. In the lower-right corner, beside the line for the cost center field, a tree-like green icon signifies that a hierarchy definition exists for this field.

Analytic model

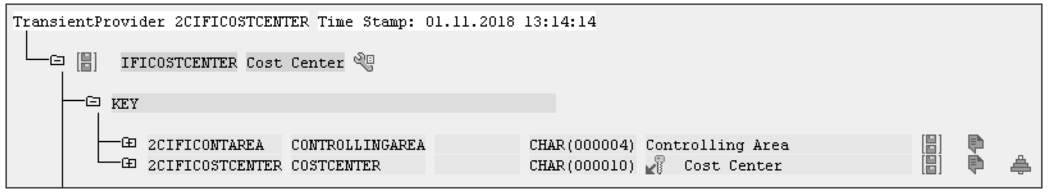


Figure 6.14 Analytic Model of View I_CostCenter

   The cost center field is the representative key field of the view; therefore, this information refers to the view itself—it possesses a hierarchy.

4. Start an analytic test of the view with the **Standard Query** function.
5. In the multidimensional analysis shown in the next screen that appears, group by cost center in rows.

6. Open the **Properties** tab in the context menu of the **Cost Center** field. You'll see the properties screen shown in Figure 6.15.

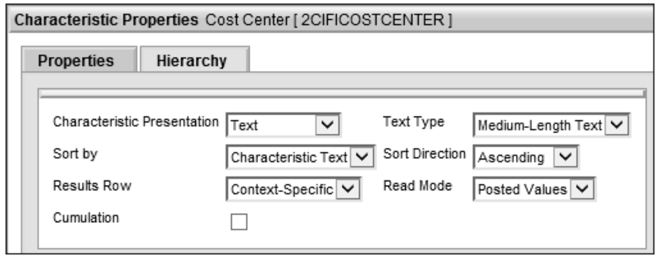


Figure 6.15 Analytic Properties of the Cost Center Field

Select a hierarchy

7. When you change to the **Hierarchy** tab, you can choose an entry from the hierarchy directory (see Figure 6.16). The example shows two versions of the standard hierarchy for cost centers in controlling area **0001** with different validity periods.

   Note that this tab is only shown if hierarchy directory entries exist.

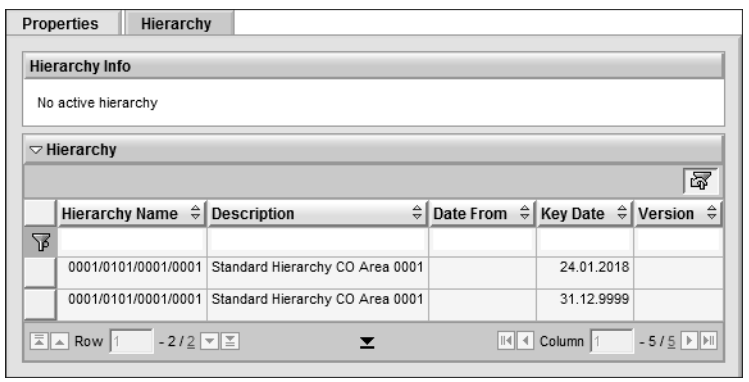


Figure 6.16 Hierarchy Directory

8. After confirming the chosen hierarchy, the list of cost centers is displayed hierarchically (see Figure 6.17).

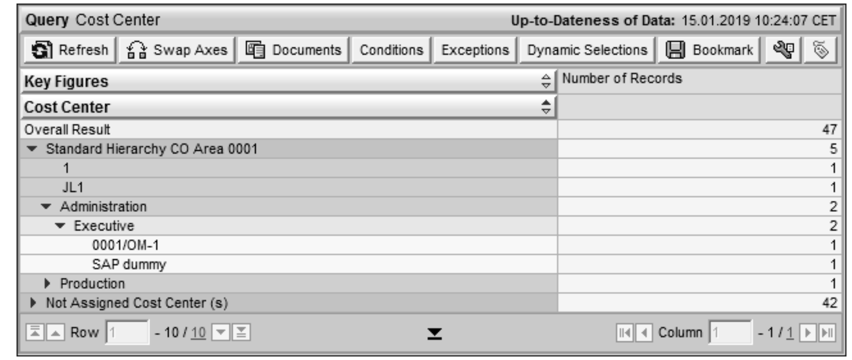


Figure 6.17 Cost Center Hierarchy

Note the artificial root node with the **Not Assigned Cost Center(s)** description, which is added by the infrastructure. It has all instances of the hierarchy base entity as children, which aren't assigned to any node of the selected hierarchy. That way, all selected cost centers are shown in the hierarchical presentation.

6.9 Summary

We started with a high-level overview of the programming model of SAP S/4HANA and showed the central role played by CDS. We then provided many examples of how semantic properties of the data can be captured in CDS metadata, which then controls the generic processing of data by the infrastructure. The example of hierarchy annotations showed how combinations of annotations, which are quite simple by themselves, can model a complex matter.

In the following chapters, in particular Chapter 8 and Chapter 9 on modeling analytical and transactional applications, you'll learn about even more options for defining applications by CDS views, their associations, and their annotations. But first, let's take a closer look at SAP S/4HANA's virtual data model in the next chapter.

Contents

Index

A

H

I

J

K

L

M

N

O

P

Q

R

S

T

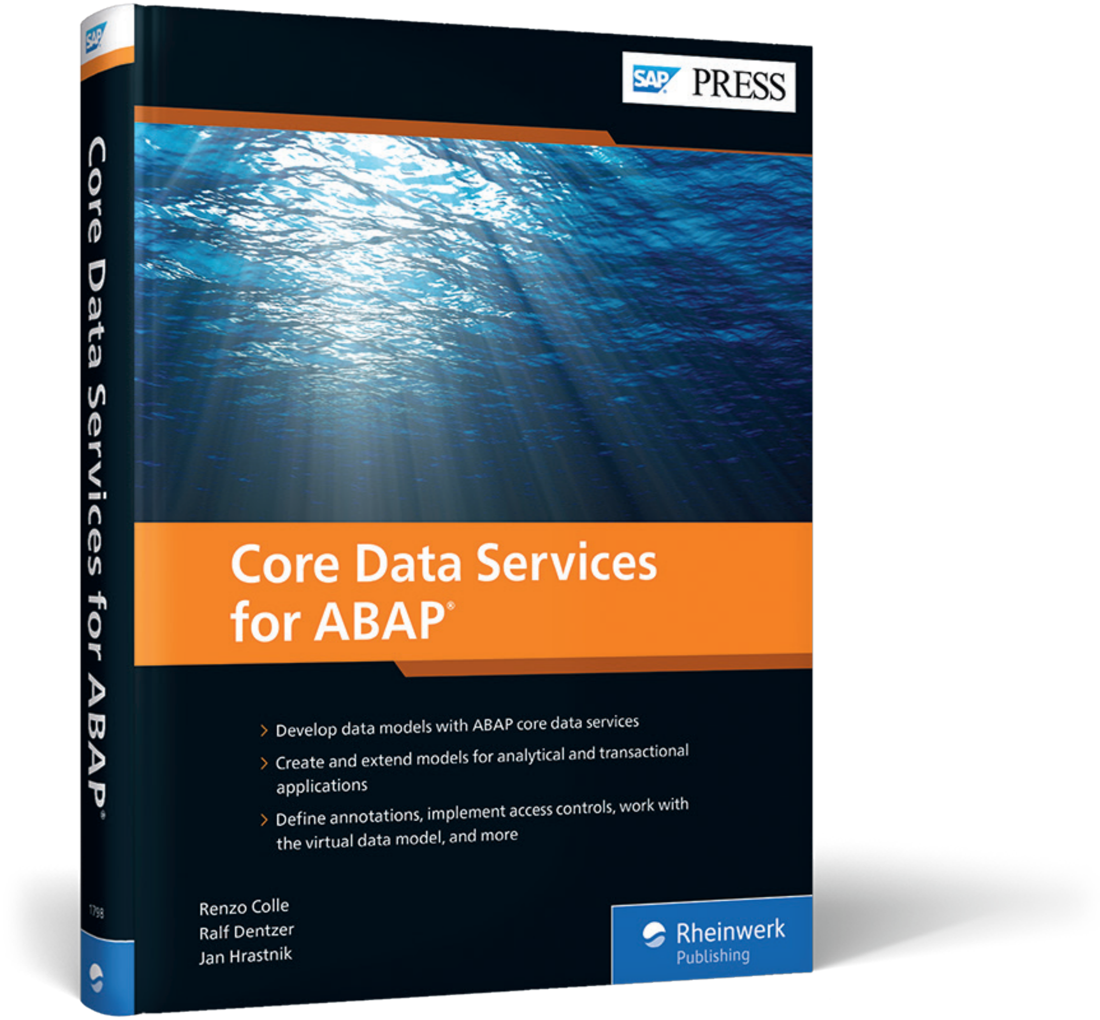
U

V

W

SAP PRESS First-hand knowledge.

Rheinwerk Publishing



Renzo Colle, Ralf Dentzer, and Jan Hrastnik

Core Data Services for ABAP

490 Pages, 2019, $79.95

ISBN 978-1-4932-1798-4

www.sap-press.com/4822

Renzo Colle is currently responsible for the SAP S/4HANA programming model in the central architecture group. He studied business mathematics at the University of Karlsruhe and has worked at SAP for more than 20 years in a wide variety of areas and roles. As the inventor of the Business Object Processing Framework (BOPF), he has worked on model-driven software development and transactional applications for more than 15 years.

Ralf Dentzer has been working for several years in the central architecture group of the SAP S/4HANA suite, with a focus on the use of Core Data Services in SAP S/4HANA. Ralf joined SAP more than 20 years ago, where he began by developing HR applications for SAP R/3, SAP ERP, and SAP Business ByDesign. After that, his tasks shifted to questions of overall architecture for new solutions. Ralf studied mathematics and received his doctorate from the University of Heidelberg.

Jan Hrastnik is a member of the SAP S/4HANA suite's architecture team, where he focuses on the virtual data model and the use of Core Data Services in ABAP applications. He has worked in various SAP development areas for over 15 years. Jan's work initially focused on developing the master data required for the production processes, before he took on overarching expert tasks in central architecture topics. He then worked on SAP SuccessFactors Employee Central and native SAP HANA application development.