Performance Modeling

Here are some considerations and advice for CDS modeling.

Avoid UNION

Using the UNION statement to merge data from different sources should be avoided. Especially, if other activities like SORTING or FILTERING are performed after the UNION statement.

In general, there are two use cases for UNIONs:

- You want to implement Polymorphism
- You port legacy applications to CAP, which already use UNION statements.

> **WARNING**
>
> UNIONs in views come with a performance penalty and complex modelling.

> **Rules of Thumb:**
>
> - If you can't change your data model, you might have to use UNION to collect semantically close data.
> - The effort of transforming data structures to avoid UNION has the benefit of better performance as well as easier modelling and less complex application code.
> - Starting a new model, you should never need to use UNION. See Polymorphism.

Polymorphism

Polymorphism might be the root cause for severe performance issues due to the usage of UNIONs, CASEs and complex JOINs. Here are some good and bad examples.

Bad

Modeling many semantically related entities:

```cds
entity Apples : cuid, managed {
  description  : String;
  vendor       : Association to one Vendor;
  appleDetails : appleDetailsType;
}

entity Bananas : cuid, managed {
  description   : String;
  vendor        : Association to one Vendor;
  bananaDetails : bananaDetailsType;
}

entity Cherries : cuid, managed {
  description   : String;
  vendor        : Association to one Vendor;
  cherryDetails : cherryDetailsType;
}

entity Mangos : cuid, managed {
  description  : String;
  vendor       : Association to one Vendor;
  mangoDetails : mangoDetailsType;
}
```

Good - normalized

Try to summarize semantically:

```cds
entity Fruit : cuid, managed {
    type          : String enum { apple; banana; cherry; mango };
    description   : String;
```

```
    vendor        : Association to one Vendor;
    appleDetails  : Composition of AppleDetails;
    bananaDetails : Composition of BananaDetails;
    cherryDetails : Composition of CherryDetails;
    mangoDetails  : Composition of MangoDetails;
}
```

You can reach common parts of any `Fruit` from anywhere using the association to `Fruit`. So `Fruit` is like an *interface* to all children. If you need the details of a certain child, you could either follow the corresponding composition or even build a specific view:

```cds
view Banana as select from Fruit
{
    type,
    description,
    vendor,
    bananaDetails,
}
    where type = 'banana';
```

Good - de-normalized

As an alternative you can also use a completely de-normalized version:

```cds
aspect apple  { appleDetails  : appleDetailsType; };
aspect banana { bananaDetails : bananaDetailsType;};
aspect cherry { cherryDetails : cherryDetailsType;};
aspect mango  { mangoDetails  : mangoDetailsType; };
entity Fruit : apple, banana, cherry, mango, cuid, managed {
    type        : String enum { apple; banana; cherry; mango };
    description : String;
    vendor      : Association to one Vendor;
}
```

This results in a single, sparsely populated DB table, which is not an issue using modern databases with variable page sizes. The optimizer will take care of it.

> **Rules of Thumb:**
>
> - Come up with a good **general** approach. You get less specific associations and a less complicated model.

> - The normalized or de-normalized `Fruit` entities have the advantage that there is only one associations to `Vendor` to be provided.

View Building

Polymorphism done right, also results in simplified view building. Assume you want to provide a list of all products of a certain vendor.

Good

Using the (de-) normalized version:

```cds
view FruitsByVendor as
select from Fruit
{ID, description, vendor}
where vendor.description = 'TopFruitCompany';
```

You have less associations to be built and no UNIONs in your queries.

Bad

Using many semantically related entities:

```cds
view FruitsByVendor as
select from Apples   UNION
select from Bananas  UNION
select from Cherries UNION
select from Mangos
{ID, description, vendor}
where vendor.description = 'TopFruitCompany';
```

Avoid JOIN

We use `OrdersHeaders` and their `OrdersItems` as an example.

```cds
entity OrdersHeaders : managed {
  key ID    : UUID;
  OrderNo   : String;
  buyer     : User;
  currency  : Currency;
  Items     : Composition of many OrdersItems on Items.Header = $self;
}
entity OrdersItems   {
    key ID     : UUID;
    product    : Association to Products;
    quantity   : Integer;
    title      : String;
    price      : Double;
    Header     : Association to OrdersHeaders;
};
```

View Building

Bad

Add a static view, using a JOIN.

```cds
view OrdersItemsViewJoin as select
    OrdersHeaders.ID        as Header_ID,
    OrdersHeaders.OrderNo   as OrderNo,
    OrdersHeaders.buyer     as buyer,
    OrdersHeaders.currency  as currency,
    OrdersItems.ID          as Item_ID,
    OrdersItems.product     as product,
    OrdersItems.quantity    as quantity,
    OrdersItems.title       as title,
    OrdersItems.price       as price
from OrdersHeaders JOIN OrdersItems on OrdersHeaders.ID = OrdersItems.Header.
```

Good

Use a dynamic entity, where you can query each fields individually, including following the association to OrderItems on demand.

```cds
entity OrderItemsViewAssoc as projection on OrdersHeaders;
```

When retrieving the *`OrderItemsViewAssoc`* via OData, you get only the *`OrdersHeaders`* without the corresponding *`Items`* by default. A JOIN will not be executed until you explicitly use the OData feature *`$expand`* to get the *`Items`* as well. Additionally, the inclusion of property lists and filters gives better control of the projection of data you like to fetch.

```http
GET http://localhost/odata/OrderItemsViewAssoc?$expand=Items&$select=OrderNo,
```

Sorting

Good

First sort on the *`OrdersItems`* and then join back to the *`OrdersHeaders`* with the help of an association:

```cds
view SortedOrdersAssoc as select {*, Header.OrderNo, Header.buyer, Header.cur
from (
  select from OrdersItems {*} order by OrdersItems.title
);
```

Bad

Sort on the right table after a JOIN. For example:

```cds
view SortedOrdersJoin as select
    OrdersHeaders.ID        as Header_ID,
    OrdersHeaders.OrderNo   as OrderNo,
    OrdersHeaders.buyer     as buyer,
    OrdersHeaders.currency  as currency,
    OrdersItems.ID          as Item_ID,
    OrdersItems.product     as product,
    OrdersItems.quantity    as quantity,
    OrdersItems.title       as title,
    OrdersItems.price       as price
```

```
from OrdersHeaders JOIN OrdersItems on OrdersHeaders.ID = OrdersItems.Header.
order by title;
```

This can lead to performance issues.

Filtering

Basically, what is true for Sorting is also valid for filtering.

Good

```cds
view FilteredOrdersAssoc as select {*, Header.OrderNo, Header.buyer, Header.c
from (
  select from OrdersItems {*}
  where OrdersItems.price > 100
);
```

Bad

```cds
view FilteredOrdersJoin as select
    OrdersHeaders.ID        as Header_ID,
    OrdersHeaders.OrderNo   as OrderNo,
    OrdersHeaders.buyer     as buyer,
    OrdersHeaders.currency  as currency,
    OrdersItems.ID          as Item_ID,
    OrdersItems.product     as product,
    OrdersItems.quantity    as quantity,
    OrdersItems.title       as title,
    OrdersItems.price       as price
from OrdersHeaders JOIN OrdersItems on OrdersHeaders.ID = OrdersItems.Header.
where price > 100;
```

This query needs to identify that prices can be filtered before the join. Filtering beforehand prevents the full join from being materialized and then reduced to a smaller subset.

Calculated Fields

Database operations on calculated fields cannot leverage any DB indexes. This impacts performance significantly, as calculated fields cause full table scans.

Typical examples of calculated fields are:

- **String concatenation** - `PrintedName = concat (FirstName, LastName)`
- **Formatting** - IBAN with spaces in between `DE99 6611 7788 5544 1122`
- **Algebra** - `FinalPrice = Rebate * ListPrice`
- **Dynamic calculations** - `Age = dynamicFunction(DateOfBirth)`
- Case statements

The following steps show you which option takes precedence over another. Use options one/two as the preferred way and three/four as fallback.

1. Do the calculation on the UI with help of field controls or dedicated custom controls. This applies to all kinds of **String concatenation** and **Formatting**.
2. Pre-calculate using CDS on write calculated fields.
3. Some calculations are dynamic in nature. If possible, use CDS on read calculated fields.
4. As a **very last resort**, use event handlers on *read*.

Hints:

- Disable sorting, filtering on calculated fields on **live** calculated fields: `@Capabilities.SortRestrictions.NonSortableProperties : [propertyA, propertyB]`
- Beware of hidden sorting (UI) & hidden filtering (authorization checks)
- Don't use **live** calculated fields in `where` clauses, as in JOIN conditions or association filtering.

Example: Case Statement → Calculation on Write

Case statements are often results of porting legacy data models. They are expensive, since they can't leverage indices and require explicit materialization. In addition, sorting or filtering forces a full table scan and expression materialization. If re-modelling to avoid case statements isn't possible, the best optimization is to pre-calculate on write (once) instead on read (many times).

Bad → Explicit case statement:

service.cds

```cds
entity OrdersItemsView as projection on OrdersItems {
    *,
    case
       when quantity > 500 then 'Large'
       when quantity > 100 then 'Medium'
       else 'Small'
    end as category : String
};
```

Good → Redundant attribute filled at write:

schema.cds

```cds
extend my.OrdersItems with {
    category: String = case
       when quantity > 500 then 'Large'
       when quantity > 100 then 'Medium'
       else 'Small'
     end stored;
}
```

Compositions vs Associations

From the performance perspective there are some cases, where you have to check out carefully if the general semantic rules of compositions vs associations should be applied. In general, go for **compositions** in the following cases:

- Parent and child share a life-cycle.
- You can semantically establish a clear parent-child-hierarchy.
- You never expose a *child* entity on its own.
- The parent of an item never changes.
- You want to keep entities together transactionally.

In general, go for **associations** in the following cases:

- You can't semantically establish a clear parent-child-hierarchy:
- You expose a document entities fully on their own.
- Relationships are likely to change over time.
- Individual entities should have individual life-cycles.

> **Rule of Thumb:**
>
> Your arm is composed to your body, your smart phone is associated to you, because it could belong to somebody else tomorrow ...

> **WARNING**
>
> Large documents, containing compositions with thousands of children, are copied entirely into draft state, even when only one little part is changed. In such cases, deviate from the general rules above, and decouple the document using associations instead of compositions.

Legacy systems

In legacy systems you find emulations for data types like string-encoded booleans (*“X”* = *true* , *“ ”* = *false*) or emulated decimal numbers. A lot of application complexity stems from the emulation of such data types and is unnecessary when using modern infrastructure. In modern systems you have dedicated *native data types*.

In addition, legacy systems often used UNIONs to save hard drive space, avoid database JOINs, or to accommodate new features without basic refactoring of the existing models. Don't take over such patterns into newly implemented applications.

> **Rules of Thumb:**
>
> - Each conversion, case statement, or unnecessary data parsing is causing a performance impact and should therefore be avoided.
> - With each conversion to a native data type, you have the opportunity to simplify the model, simplify the application logic, and improve performance from the start.

Common patterns:

- String encoded Booleans (like `“X” = true`, `“ ” = false`) might infer future case statements. Better convert them to real booleans ("true" and "false") directly at the beginning in the database.
- Convert emulated decimal numbers (like integers with additional positional formatting info) to real decimal numbers on the database.
- Multiple attribute columns (like Address01, Address02, Address03, ...) to avoid extensive JOINs in old DB systems are causing a lot of complex queries and business logic. Better use compositions of type instead and only de-normalize your data model where it really makes sense.
- Positional strings, like `A_G___U_I`, with complex internal logic might infer future case statements or complex internal calculations. Better use compositions of type here as well.
- If you encounter UNION statements in your legacy model, we strongly suggest to re-model as described in the section on Polymorphism.
- If you encounter CASE statements in your legacy model, we strongly suggest to re-model as described in the section on Calculated Fields.
- Omit unnecessary abstraction views. When you are porting an "ABAP" CDS Model starting with the corresponding Virtual Data Model (VDM), C_Views and I_Views don't serve a purpose anymore. Please design your entities for optimized persistence, and your service layer for optimized processing. CAP already has a separation of concerns between the "DB" layer (persistency model) and the "SRV" layer (service / consumption model), there is no need to insert additional and unnecessary further abstraction layers. For example, the public interface layer with views like `I_COSTCENTER` will be replaced by the CDS services from which the OData consumption services are generated.
- Legacy systems often have convoluted or overly complex data structures just to satisfy multiple processing requirements or use-cases with the same data structure. In CAP there is no need to create overly complex service entities, since you can use bound and unbound ACTIONs and FUNCTIONs for more complex data manipulation. Keep service entities as simple as possible and make them serve one purpose only, and rather create multiple simple entities instead of a complex one.

Summary:

Legacy	Re-Model → Modern → Better Performance
String encoded Booleans	Convert them to real booleans on the database.
Emulated decimal numbers	Convert to real decimal numbers on the database.

Legacy	Re-Model → Modern → Better Performance
Multiple attribute columns	Better use compositions of type instead.
Positional strings, like `A_G___U_I`	Better use compositions of type.
UNION statements	Re-model as described in Polymorphism.
CASE statements	Re-model as described in Calculated Fields.
Complex data structures to satisfy multiple processing requirements	Use bound and unbound ACTIONs and FUNCTIONs and keep service entities as simple as possible.
Unnecessary abstraction views (C_Views, I_Views)	Design your entities for optimized persistence, and your service layer for optimized processing.

Edit this page

Last updated: 25/02/2025, 10:33

Previous page
Reuse & Compose

Next page
CDS

Was this page helpful?

👍 👎