CDS View Performance Optimization

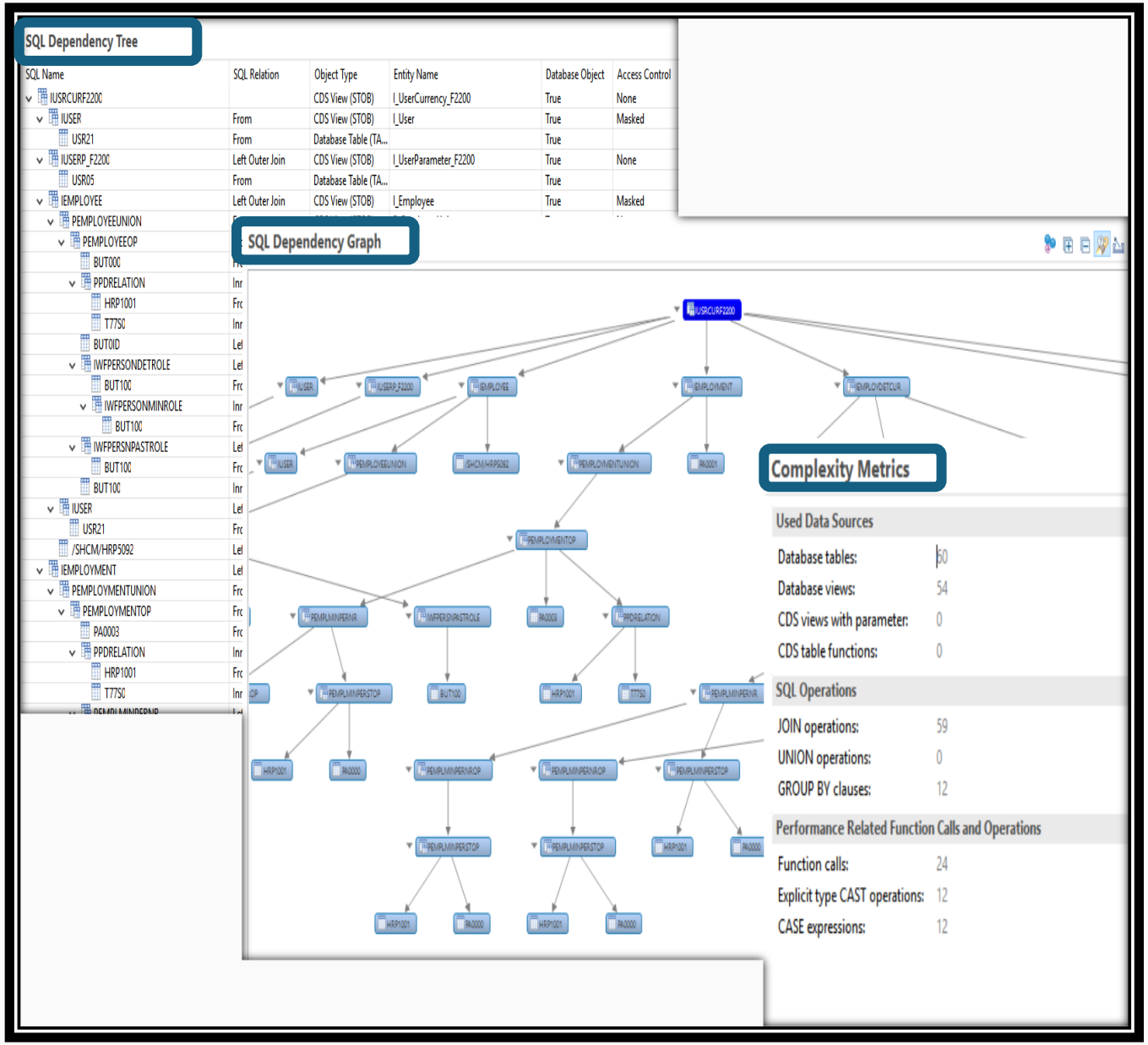
- Assessing Static Complexity
- Utilizing Performance Annotations
- Adopting Best Practices
- Optimization Example

CDS View – Static Complexity Assessment

- **SQL Dependency Tree:** This visualizes the hierarchical relationships between CDS views and their dependencies. It helps in understanding how views are interlinked and how changes in one might impact others.

- **SQL Dependency Graph:** This tool shows the dependencies in a graphical format, which can make it easier to grasp complex relationships between various views and tables.

- **Complexity Metrics:** This provides quantitative data on the complexity of a CDS view. The key metric you mentioned—the number of database tables used—helps in understanding the data sources involved. This number reflects how many tables are directly referenced or associated in the design of the view.



CDS View – Performance Annotations

@ObjectModel.usageType.serviceQuality
Purpose: Defines the expected performance level of the CDS view in terms of service quality. This annotation helps in identifying how critical the performance of the view is.

@ObjectModel.usageType.dataClass
Purpose: Specifies the type of data the CDS view handles. This helps in understanding the nature of the data and its role in the business process.

@ObjectModel.usageType.sizeCategory
Purpose: Indicates the volume of data that needs to be processed or searched to compute the result set of the CDS view.

serviceQuality	Each CDS view shall be assigned to one of the following quality categories: **A**: the view may be consumed within business logic for **high volume transactions** or background processing **B**: the view may be consumed within business logic for **transactions** or background processing **C**: the view may be consumed **from the UI** in transactions for single object retrieval **D**: the view may be consumed for **analytical reporting** **X**: the view is built to **push down application code** to HANA
sizeCategory	Each CDS view shall have assigned a size category. The size category enables the consumer to judge the possible result set. It reflects the number of rows that has to be searched through to get a result. The labels correspond to the following size categories (expected number of rows in customer production systems): **S** < 1000, **M** < 100.000, **L** < 10.000.000, **XL** < 100.000.000, **XXL** > 100.000.000
dataClass	To support the decision on cache strategies for higher layers and to enable client side statement routing using these caches, each CDS view shall have assigned a data class. The different data classes correspond to different life time cycles. **TRANSACTIONAL** data is written or changed in high volume transactions **MASTER** data is read, but not written or changed in high volume transactions **ORGANIZATIONAL** data describes the organizational structure of a company and its business processes **CUSTOMIZING** data describes how a concrete business process is executed at the customer **META** data specifies how the system is configured or describes the technical structure of entities **MIXED** data shall be chosen if the CDS-View contains tables with several different of the above types

CDS View – Best Practices

- **Start Simple**: Begin by creating basic CDS views and gradually advance to more intricate ones as you build confidence and understanding.

- **Limit Exposed Fields:** Only include the fields that are essential to minimize the number of tables accessed and the complexity of joins and operations.

- **Prefer Associations:** Use associations rather than LEFT OUTER JOIN to simplify the view structure and enhance performance, as associations can be more efficient.

- **Optimize Joins**: When possible, use LEFT OUTER JOIN MANY TO ONE to enable join pruning, which improves query efficiency.

- **Apply Performance Annotations:** Set and maintain performance annotations to clearly define the expected service quality, type of data, and size category of your CDS views.

- **Manage Table Limits:** Avoid exceeding 100 underlying tables in a CDS view and follow stricter guidelines for views with higher service quality levels.

- **Focus on Data Quality:** Prioritize the quality of your data model and the data within your sources, as these factors have a greater impact on performance than the complexity of the CDS view itself.

CDS View – Optimization Scenario

Let's explore a scenario where we join Purchase Order header and item data using CDS views. Take a look at the calculated field in the Purchase Order header CDS view with its logic detailed below. We also have some sample data and the expected result of this join.

Input Data

Order Header

Order Number	Status	Release Strategy(frgrl)
ABC	9	X
XYZ	3	X
MNO	9	

Order Item

Order Number	Order Item	Amount	Material
ABC	1	10	TG11
XYZ	1	50	TG11
XYZ	2	100	TG11
NNN	1	200	TG11
MNO	1	30	TG13

Logic for new field enjoyReleaseNotCompleted

```
case
  when statu = '9' and frgrl = 'X' then 'X'
  else ''
end as enjoyReleaseNotCompleted,
```

Expected Result

Order Number	Order Item	Amount	Material	Status	Release Strategy(frgrl)	enjoyReleaseNotCompleted
ABC	1	10	TG11	9	X	X
XYZ	1	50	TG11	3	X	
XYZ	2	100	TG11	3		
NNN	1	200	TG11	NULL	NULL	NULL

CDS View – Design

```
@AbapCatalog.preserveKey: true
@AbapCatalog.compiler.compareFilter: true
@ClientHandling.algorithm: #SESSION_VARIABLE
@AbapCatalog.sqlViewName: 'ZRWPURCHASINGDOC'
@EndUserText.label: 'Purchasing Document'

define view ZHG_PURCHASINGDOCUMENT
  as select from ekko as ekko //PurgDoc

    @ObjectModel.foreignKey.association: '_PurchasingDocumentOrigin'
    ekko.statu                                  as PurchasingDocumentOrigin,

    ekko.frgrl                                  as ReleaseIsNotCompleted,

    ekko.statu,
    ekko.frgrl,

    case
      when statu = '9' and frgrl = 'X' then 'X'
      else ''
    end as enjoyReleaseNotCompleted,

    case
      when statu = '9' then 'X'
      else ''
    end as OrderCompleted,
```

Purchasing Document Header CDS View

Purchasing Document Item CDS View

```
define view ZHG_PURCHASEORDERITEM
  as select from I_PurchasingDocumentItem

  association [1..1] to ZHG_PURCHASINGDOCUMENT    as _PurchaseOrder    on  $projection.PurchaseOrder = _PurchaseOrder.PurchasingDocument
```

CDS View – Performance Observation

Query 1 : *Below query selects fields from just one table (order item). We'll use this as a baseline to compare performance and observe the impact of more complex queries.*

```
SELECT  purchaseorder, purchaseorderitem, material
FROM    ZRW_PURCHORDRITEM
WHERE   material = ?
ORDER BY purchaseorder, purchaseorderitem
LIMIT 200;
```

Result Runtime *– 5 ms*

```
Statement 'select purchaseorder, purchaseorderitem, material from ZHGPURCHORDRITEM where material = ? order by ...'
successfully executed in 29 ms 473 µs  (server processing time: 5 ms 329 µs)
Fetched 200 row(s) in 23 ms 458 µs (server processing time: 0 ms 724 µs)
```

Query 2: *Let's add the additional field "enjoyReleaseNotCompleted" to the projection list while keeping the rest of the query unchanged.*

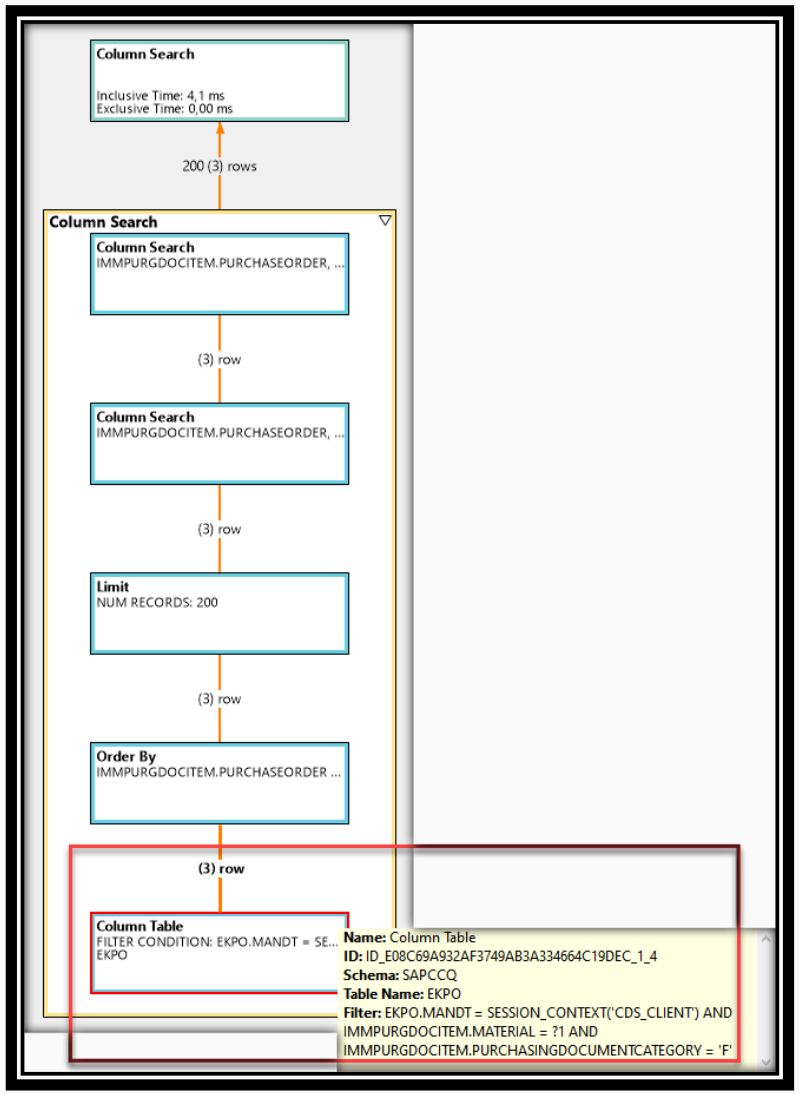
Result Runtime *– 2 s*

```
Statement 'select purchaseorder, purchaseorderitem, material,enjoyReleaseNotCompleted from ZHGPURCHORDRITEM ...'
successfully executed in 2.264 seconds  (server processing time: 2.244 seconds)
Fetched 200 row(s) in 1.306 seconds (server processing time: 1.284 seconds)
```

The execution time increased from 5 ms to over 2 seconds.

Plan Viz Analysis

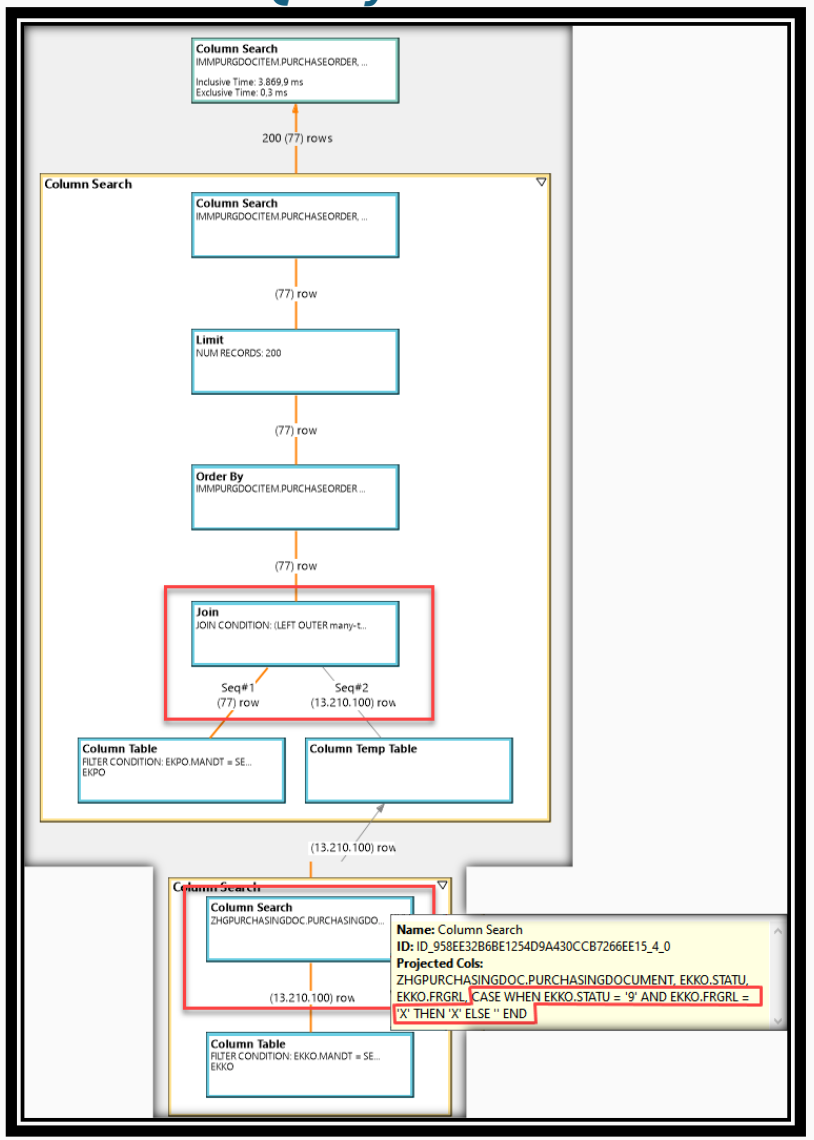
Plan viz for Query 1



1.Only Purchasing Document Item table hit as no fields requested from Header

2. Filters Applied on the lower most node.

Plan Viz for Query 2



1. First Calculation is performed on all entries of Header Table.

2. Second a filter is applied on items data.

3. A join is done between header and item

Performance Analysis

In the second SELECT, why wasn't the filter applied before performing the calculations?
Why didn't the HANA SQL Optimizer choose the optimized algorithm **(Filter > Join > Calculate)** rather than the less efficient one (**Calculate > Filter > Join)** ?

Calculate > Filter > Join

Calculate

Order Number	Status	Release Strategy(frgrl)	enjoyReleaseNotCompleted
ABC	9	X	X
XYZ	3	X	
MNO	9		

Filter

Order Number	Order Item	Amount	Material
ABC	1	10	TG11
XYZ	1	50	TG11
XYZ	2	100	TG11
NNN	1	200	TG11

Join

Order Number	Order Item	Amount	Material	Status	Release Strategy(frgrl)	enjoyReleaseNotCompleted
ABC	1	10	TG11	9	X	X
XYZ	1	50	TG11	3	X	
XYZ	2	100	TG11	3		
NNN	1	200	TG11	NULL	NULL	NULL

Filter > Join > Calculate

Filter

Order Number	Order Item	Amount	Material
ABC	1	10	TG11
XYZ	1	50	TG11
XYZ	2	100	TG11
NNN	1	200	TG11

Join

Order Number	Order Item	Amount	Material	Status	Release Strategy(frgrl)
ABC	1	10	TG11	9	X
XYZ	1	50	TG11	3	X
XYZ	2	100	TG11	3	
NNN	1	200	TG11	NULL	NULL

Calculate

Order Number	Order Item	Amount	Material	Status	Release Strategy(frgrl)	enjoyReleaseNotCompleted
ABC	1	10	TG11	9	X	X
XYZ	1	50	TG11	3	X	
XYZ	2	100	TG11	3		
NNN	1	200	TG11	NULL	NULL	

As technically the result of optimized algorithm is not same as expected result hence the HANA SQL optimizer opts for the bad algorithm, as the bad algorithm matches the expected results.

Optimization Options

Option 1

One way to improve this performance issue is to remove the "not null preserving" function. You can achieve this by eliminating the ELSE branch from the calculation, so the function returns NULL when the condition is not met.

```
case
  when statu = '9' and frgrl = 'X' then 'X'
end as enjoyReleaseNotCompletedOpt,
```

Option 2

Another way to address the problem is to rewrite the `ELSE` branch expression as a NULL-preserving expression by adding a second expression with the inverted condition and omitting the `ELSE` argument . By doing this, the `NULL` value is not explicitly handled but will be implicitly evaluated by the Optimizer, as the `ELSE` argument becomes optional.

```
case
  when statu = '9' and frgrl = 'X' then 'X'
  when not (  statu = '9' and frgrl = 'X' ) then ''
end as enjoyReleaseNotCompletedOpt3,
```