ABAP Joins & CTE

---

1. What Are Joins?

A **join** is the process of combining data from two or more database tables based on a related column.
In HANA-optimized ABAP, joins happen **at the database layer**, and the execution engine determines the best access path.

Understanding joins is not optional anymore — it's the backbone of **CDS views, AMDP methods, Open SQL**, and even analytical data modeling.

---

2. Types of Joins in ABAP

2.1 INNER JOIN

Concept:
Returns only records that have matching entries in both tables.

Syntax Example:

```
SELECT a~matnr, a~mtart, b~maktx
  FROM mara AS a
  INNER JOIN makt AS b
    ON a~matnr = b~matnr
  INTO TABLE @DATA(it_materials)
  WHERE a~mtart = 'FERT'.
```

Explanation:

- `MARA` (material master) and `MAKT` (material descriptions) are linked using `matnr`.
- Only materials that exist in **both** tables appear in the result.

Use Case:
When you need only consistent, relationally correct data — for example, materials that have both technical and language descriptions.

When to Avoid:

- When you need a full list of records even if data is missing in one table.
- Overuse of INNER JOIN in multi-table scenarios can degrade performance.

Alternative:
Use **LEFT OUTER JOIN** if you want to retain unmatched records from the left table.

2.2 LEFT OUTER JOIN

Concept:
Returns all records from the left table and the matched ones from the right. Unmatched right-side fields will have **NULL** values.

Syntax Example:

```
SELECT a~matnr, a~mtart, b~maktx
  FROM mara AS a
  LEFT OUTER JOIN makt AS b
    ON a~matnr = b~matnr
  INTO TABLE @DATA(it_left_join)
  WHERE a~mtart = 'FERT'.
```

Explanation:

- All materials are fetched from `MARA` even if no description exists in `MAKT`.

Use Case:
Reporting or interfaces where you must show all master records, even if supplementary data is missing.

When to Avoid:

- If you only need matched data (INNER JOIN suffices).
- On massive datasets — NULL handling in HANA adds cost.

Alternative:
In CDS, you can use **association with optional cardinality** to handle left joins gracefully.

2.3 RIGHT OUTER JOIN

Concept:
Returns all records from the right table and matching ones from the left. Rarely used in practice.

Syntax Example:

```
SELECT a~matnr, b~maktx
  FROM mara AS a
  RIGHT OUTER JOIN makt AS b
    ON a~matnr = b~matnr
  INTO TABLE @DATA(it_right_join).
```

Explanation:
All entries from `MAKT` are fetched even if the material doesn't exist in `MARA`.

Use Case:
Hardly needed in business logic, but occasionally useful for data audits or quality reports to find orphan text entries.

When to Avoid:
In most cases. Use LEFT OUTER JOIN with swapped table order — it's simpler and more intuitive.

2.4 FULL OUTER JOIN

Concept:
Returns all records when there is a match in either table. Non-matching rows from both sides appear with NULLs in missing columns.

Example (CDS View):

```
@AbapCatalog.sqlViewName: 'ZFULLJOIN'
define view zfull_join_example as
  select from mara
    full outer join makt
      on mara.matnr = makt.matnr
{
  mara.matnr,
  mara.mtart,
  makt.maktx
}
```

Use Case:
For data reconciliation between systems or validating synchronization between tables.

When to Avoid:

- Not supported in classic Open SQL (only via CDS or native SQL).
- Usually unnecessary for transactional logic.

Alternative:
Perform a UNION of LEFT OUTER and RIGHT OUTER joins if FULL OUTER JOIN is unsupported in your ABAP stack.

2.5 CROSS JOIN

Concept:
Cartesian product — combines every record of one table with every record of another.

Example (CDS):

```
define view zcross_join as
  select from t001, t005
```

```
{
  t001.bukrs,
  t005.land1
}
```

Explanation:
Each company code (`T001`) is combined with each country (`T005`).

Use Case:
Almost never used directly. It's helpful only in data generation scenarios or when preparing matrix-like datasets for analytics.

When to Avoid:
In real business logic. The record explosion can cripple performance.

Alternative:
Use meaningful joins or associations with filter conditions.

2.6 SELF JOIN

Concept:
Join a table with itself to compare hierarchical or parent-child relationships.

Example:

```
SELECT a~pernr, a~ename, b~ename AS manager_name
  FROM pa0001 AS a
  INNER JOIN pa0001 AS b
    ON a~manager = b~pernr
  INTO TABLE @DATA(it_hierarchy).
```

Use Case:
Employee-manager hierarchy, BOM parent-child link, or organizational reporting lines.

When to Avoid:

- On large datasets without indexed join keys.
- If you can achieve the same with hierarchy functions or CDS associations.

Alternative:
Use CDS **hierarchy views** or recursive CTEs (Common Table Expressions) in HANA for better performance.

---

3. Joins in CDS Views

In **ABAP CDS**, joins are declarative and optimized automatically by the HANA engine.

Example:

```
@AbapCatalog.sqlViewName: 'ZMATJOIN'
define view z_material_join as
  select from mara
    inner join makt
      on mara.matnr = makt.matnr
{
  mara.matnr,
  mara.mtart,
  makt.maktx
}
```

Best Practices:

- Define only required fields.
- Prefer associations over direct joins where relationships are naturally defined.
- Filter early (push-down where conditions).

---

4. Performance & Pitfalls

Pitfall	Description	Recommendation
Unnecessary joins	Joining lookup tables when not needed	Fetch lookup data in a separate call if reused multiple times
NULL propagation	LEFT JOIN may create unwanted NULL handling	Use COALESCE() or CASE in CDS to manage defaults
Join cardinality	Incorrect join key design leads to duplicates	Ensure key fields are uniquely identifying
Database load	Heavy joins executed in application layer	Always push logic to HANA via Open SQL or CDS
Complex joins in AMDP	Difficult to maintain or optimize	Split logic into multiple temporary result sets

---

5. Alternative Approaches

Scenario	Alternative Approach	Example
Need multiple joins with repeated data	Use **CTEs or temporary tables**	`WITH temp AS (SELECT ...)` in AMDP
Need lookup data	Use **FOR ALL ENTRIES** if key list is small	Fetch selective data only
Need model reuse	Create **reusable CDS views**	Base view → Consumption view
Need optional joins	Use **CDS associations with cardinality [0..1]**	Better readability and reusability

---

6. Expert Notes for ABAP on HANA

- Push joins to the **database layer** — avoid joining internal tables in ABAP loops.
- Use **Open SQL joins** or **CDS views** instead of nested SELECTs.
- When possible, filter data **before joining**, not after.
- Analyze join execution with **SQL Monitor (SQLM)** or **Performance Trace (ST05)**.
- Understand that **HANA optimizer rewrites** joins intelligently — design clean, not complex SQL.

---

7. Quick Reference Summary

Join Type	Description	When to Use	Avoid When
INNER JOIN	Matching records only	Clean relational data	You need all records
LEFT OUTER JOIN	Keep all from left	Reporting completeness	Right table is huge
RIGHT OUTER JOIN	Keep all from right	Data audits	Complex business logic
FULL OUTER JOIN	Keep all from both	Data reconciliation	Transactional logic
CROSS JOIN	Cartesian product	Test data generation	Any production query
SELF JOIN	Same table relationships	Hierarchies	Without indexes

---

8. Closing Thoughts

In the HANA era, join design determines your **application speed, scalability, and maintainability**.
Think joins not as syntax but as **data flow design** — how data interacts and how efficiently your system can interpret it.

The goal is not to use every join type but to **understand when to use which** — and, equally, **when not to**.

Mastering joins means mastering the language of relationships in SAP's data world.

HANA Common Table Expressions (CTE) – Practical Guide for ABAP on HANA

1. Introduction

In SAP HANA and ABAP on HANA, **CTEs (Common Table Expressions)** represent a powerful way to structure complex SQL logic cleanly.
They simplify multi-step queries, enhance readability, and let developers express recursive or temporary data transformations in a single SQL statement.

Unlike traditional joins where logic is stacked in one large SELECT, CTEs allow modular and sequential query building — making your HANA SQL or AMDP more maintainable and optimized.

---

2. What Is a CTE?

A **CTE** is a **temporary named result set** defined within the execution scope of a SQL statement.
It starts with the `WITH` clause and can be used just like a table or view inside your main query.

Key Points:

- Exists only during query execution (not stored permanently).
- Can be referenced multiple times within the main query.
- Supports **recursive** and **non-recursive** definitions.
- Greatly improves SQL clarity compared to nested subqueries.

---

3. Basic CTE Structure

Syntax (HANA SQL / AMDP style):

```
WITH cte_name AS (
  SELECT column1, column2
  FROM some_table
  WHERE condition
)
SELECT *
FROM cte_name
WHERE another_condition;
```

In ABAP, you can implement this via **AMDP methods** or **native SQL in CDS table functions**.

---

4. Example 1 – Simplifying Multi-Join Logic

Scenario:
You want to fetch active sales orders with their item counts and related customer names.

Without CTE (Nested Joins):

```
SELECT vbak~vbeln, vbak~kunnr, kna1~name1, COUNT(vbap~posnr) AS item_count
  FROM vbak
  INNER JOIN vbap ON vbak~vbeln = vbap~vbeln
  INNER JOIN kna1 ON vbak~kunnr = kna1~kunnr
  WHERE vbak~auart = 'OR'
  GROUP BY vbak~vbeln, vbak~kunnr, kna1~name1
  INTO TABLE @DATA(it_sales).
```

With CTE (AMDP or HANA SQL):

```
WITH cte_orders AS (
    SELECT vbeln, kunnr
    FROM vbak
    WHERE auart = 'OR'
),
cte_items AS (
    SELECT vbeln, COUNT(posnr) AS item_count
    FROM vbap
    GROUP BY vbeln
)
SELECT a.vbeln, a.kunnr, b.item_count, c.name1
  FROM cte_orders AS a
  LEFT JOIN cte_items AS b ON a.vbeln = b.vbeln
  LEFT JOIN kna1 AS c ON a.kunnr = c.kunnr;
```

Why better:

- Each logical piece (orders, items) is isolated.
- Reusable and readable — especially when debugging AMDPs.
- HANA optimizer merges CTEs efficiently during execution.

---

5. Example 2 – Recursive CTE for Hierarchies

Scenario:
You need to fetch a Bill of Materials (BOM) structure — parent and all nested components.

Recursive CTE (HANA SQL):

```
WITH RECURSIVE cte_bom (parent, component, level) AS (
    SELECT parent, component, 1 AS level
      FROM z_bom
      WHERE parent = 'FG1001'
  UNION ALL
    SELECT a.parent, b.component, a.level + 1
      FROM cte_bom AS a
      INNER JOIN z_bom AS b
        ON a.component = b.parent
)
SELECT * FROM cte_bom;
```

Explanation:

- The first SELECT defines the base level.

- The UNION ALL recursively adds deeper hierarchy levels.
- The recursion stops when no more child components exist.

Use Case:
Useful for BOM explosion, org hierarchies, or managerial reporting structures.

---

6. Example 3 – Combining Joins and CTEs in AMDP

AMDP Method Example:

```
METHOD get_sales_data BY DATABASE PROCEDURE
  FOR HDB LANGUAGE SQLSCRIPT
  OPTIONS READ-ONLY
  USING vbak vbap kna1.

  WITH cte_orders AS (
      SELECT vbeln, kunnr
      FROM vbak
      WHERE erdat >= ADD_DAYS(CURRENT_DATE, -30)
  ),
  cte_items AS (
      SELECT vbeln, SUM(netwr) AS total_value
      FROM vbap
      GROUP BY vbeln
  )
  SELECT a.vbeln, b.total_value, c.name1
    FROM cte_orders AS a
    INNER JOIN cte_items AS b ON a.vbeln = b.vbeln
    LEFT JOIN kna1 AS c ON a.kunnr = c.kunnr
    ORDER BY b.total_value DESC;

ENDMETHOD.
```

Advantages in HANA:

- CTEs are treated like optimized temporary views.
- Greatly reduce complexity in reporting or analytical AMDPs.
- Ideal for step-by-step transformations or aggregations.

---

7. CTE vs Traditional Joins and Subqueries

Aspect	Traditional Join / Subquery	HANA CTE
Readability	Difficult in deep nesting	Modular, easy to follow
Reusability	Hard to reuse mid-results	Each CTE can be reused
Performance	Often re-executes subqueries	CTE processed once and reused
Maintainability	Harder to debug	Logical step isolation
Recursive Capability	Limited	Supported natively

---

8. When to Use and When to Avoid CTEs

Use CTEs When:

- You need to **modularize complex SQL logic**.
- You want to **reuse the same subset** multiple times in one query.
- Performing **hierarchical or recursive processing**.
- You're working in **AMDPs** or **native SQL** scenarios on HANA.

Avoid CTEs When:

- Query is simple and single-step.
- Performance profiling shows no improvement (CTE may not always be cached).
- Your ABAP release or HANA revision does not fully support recursive syntax.

---

9. Expert Notes for ABAP on HANA

- HANA internally **flattens non-recursive CTEs** into a single optimized plan; there's usually no runtime overhead.
- Combine CTEs with **analytic functions (RANK, ROW_NUMBER)** for complex reports.
- Recursive CTEs can replace custom loops or internal table processing.
- Test CTE logic directly in **HANA Studio** or **Database Explorer** before embedding into AMDP.
- For massive datasets, ensure **JOINs inside CTEs** have proper indexes and filters.

---

10. Conclusion

CTEs transform how we structure SQL in ABAP on HANA — making it **modular, elegant, and performant**.
Where joins define relationships, **CTEs define logical flow**.
Together, they form the foundation of clean, maintainable data logic in modern HANA-based development.

HANA CTE vs Joins – A Deep Comparative Guide for ABAP on HANA

1. Preface

In ABAP on HANA, **Joins** and **CTEs (Common Table Expressions)** both serve the purpose of combining and transforming data at the database level.
While they might seem interchangeable at first glance, their internal behavior, performance characteristics, and ideal use cases differ greatly.

This document breaks down the conceptual, technical, and practical differences between **traditional joins** and **HANA CTEs**, allowing you to decide which fits better in different real-world scenarios.

---

2. Conceptual Difference

Aspect	Joins	CTEs
Purpose	Combine data from multiple tables in a single SQL statement	Create intermediate result sets that can be reused or built upon
Nature	Relational and direct	Step-wise and modular
Readability	Becomes complex with multiple joins	Highly readable and structured
Scope	Single statement-level operation	Multiple sequential transformations within a statement
Reusability	No – each join is fixed per query	Yes – each CTE can be referenced multiple times

---

3. Execution Flow Difference

3.1 How Joins Work

When you perform a join, HANA's optimizer decides:

- The join order,
- Which table acts as the driving table,
- And how filters are applied (join pruning, index usage).

Everything is executed **in a single flat SQL plan**.

3.2 How CTEs Work

When you define a CTE, each `WITH` clause acts as a **temporary view**.
Each CTE can:

- Simplify multi-step transformations,
- Be referenced multiple times,
- And, in recursive cases, call itself to derive hierarchies.

The HANA optimizer may **inline** non-recursive CTEs (treating them like subqueries) or **materialize** them temporarily for performance stability.

---

4. Example Comparison

4.1 Complex Join-Based Query

```
SELECT a~vbeln, a~kunnr, b~posnr, b~netwr, c~name1
  FROM vbak AS a
  INNER JOIN vbap AS b ON a~vbeln = b~vbeln
  INNER JOIN kna1 AS c ON a~kunnr = c~kunnr
  WHERE a~auart = 'OR'
  INTO TABLE @DATA(it_result).
```

Challenges:

- Hard to debug or modify.
- If additional logic is needed (e.g., aggregate item totals), nesting becomes messy.

4.2 The Same Logic Using CTE

```
WITH cte_orders AS (
    SELECT vbeln, kunnr
    FROM vbak
    WHERE auart = 'OR'
),
cte_items AS (
    SELECT vbeln, SUM(netwr) AS total_value
    FROM vbap
    GROUP BY vbeln
)
SELECT a.vbeln, a.kunnr, b.total_value, c.name1
  FROM cte_orders AS a
  INNER JOIN cte_items AS b ON a.vbeln = b.vbeln
  INNER JOIN kna1 AS c ON a.kunnr = c.kunnr;
```

Advantages:

- Each logical segment (orders, items, customers) is modular.
- CTEs make transformations explicit.
- Future enhancements are easier (e.g., additional filters, extra joins).

---

5. Performance Comparison

Factor	Joins	CTEs
Optimization	HANA optimizer merges joins efficiently	Non-recursive CTEs are often inlined, performance similar to joins
Reusability	Repeated logic causes multiple joins	Reused result sets processed once

Materialization	No temporary materialization	May materialize CTE results for reuse
Execution Plan	Flat and compact	Stepwise or recursive, depending on design
Scalability	Excellent for straightforward joins	Better for multi-layered aggregations or recursive logic

Insight:
For simple relational data fetching, **joins are faster**.
For complex transformations, **CTEs are more maintainable** and often equally efficient after optimization.

---

6. Maintainability and Readability

Aspect	Joins	CTEs
Code Readability	Decreases with more tables	Structured and readable
Debugging	Difficult due to deep nesting	Each step is testable
Business Logic Segmentation	Hard to isolate transformations	Natural step-by-step organization
Future Enhancement	Fragile for change	Scalable and easy to extend

Example: Adding one more condition or calculation in a 5-table join can cause massive rewrite, whereas adding another CTE step feels natural.

---

7. Recursive and Hierarchical Scenarios

With Joins

You would need multiple self-joins or looping logic in ABAP to fetch parent-child hierarchies like BOM or organization trees.

With CTEs

You can achieve it in a single recursive query:

```
WITH RECURSIVE cte_bom (parent, component, level) AS (
  SELECT parent, component, 1 AS level
  FROM z_bom
  WHERE parent = 'FG1001'
  UNION ALL
  SELECT a.parent, b.component, a.level + 1
  FROM cte_bom AS a
  INNER JOIN z_bom AS b ON a.component = b.parent
)
SELECT * FROM cte_bom;
```

Advantage:
CTEs handle recursion natively, eliminating the need for iterative logic in ABAP loops or self-joins.

---

8. When to Use Each

Situation	Prefer Joins	Prefer CTEs
Simple relational data fetch	✅	
Aggregation within a single dataset	✅	
Multi-step transformation		✅
Recursive or hierarchical data		✅
Reusing filtered subsets		✅
Performance-critical transactional queries	✅	
Analytical or reporting queries		✅

---

9. Practical Guidance for ABAP on HANA

1. **Use Joins** when:
   - The dataset is straightforward.
   - You're fetching data for transactions or standard reports.
   - Query structure is simple and will not evolve.
2. **Use CTEs** when:
   - Query involves multiple derived datasets or transformations.
   - You are developing AMDP procedures or complex analytics.
   - The logic benefits from recursion or step separation.
   - You want clarity in debugging and maintenance.
3. **Hybrid Approach:**
   - Start with a base join in one CTE, and build subsequent steps with CTE chaining.
   - This combines performance with readability.

---

10. Pitfalls to Avoid

Pitfall	Explanation	Recommendation
Over-nesting joins	Hard to optimize and maintain	Break into smaller CTEs
Unnecessary CTE materialization	May cause memory overhead	Use only when reuse or recursion needed
Ignoring optimizer rewrites	HANA may flatten your CTE	Don't rely on forced step-by-step execution

Complex recursive limits	Recursive depth can explode	Limit recursion with `LEVEL <= n` or termination conditions
Misusing CTE in simple queries	Adds no real benefit	Keep simple joins where possible

---

11. Real-World Example: Sales Analytics

Without CTE (Joins):

```
SELECT a~vbeln, a~kunnr, SUM(b~netwr) AS total_value, COUNT(b~posnr) AS
item_count, c~name1
  FROM vbak AS a
  INNER JOIN vbap AS b ON a~vbeln = b~vbeln
  INNER JOIN kna1 AS c ON a~kunnr = c~kunnr
  WHERE a~auart = 'OR'
  GROUP BY a~vbeln, a~kunnr, c~name1
  INTO TABLE @DATA(it_sales).
```

With CTE:

```
WITH cte_orders AS (
    SELECT vbeln, kunnr
    FROM vbak
    WHERE auart = 'OR'
),
cte_items AS (
    SELECT vbeln, COUNT(posnr) AS item_count, SUM(netwr) AS total_value
    FROM vbap
    GROUP BY vbeln
)
SELECT a.vbeln, a.kunnr, b.item_count, b.total_value, c.name1
  FROM cte_orders AS a
  INNER JOIN cte_items AS b ON a.vbeln = b.vbeln
  INNER JOIN kna1 AS c ON a.kunnr = c.kunnr;
```

Observation:

- The CTE version is cleaner and easier to extend for additional analytics like product mix or delivery dates.
- HANA execution time is almost identical since non-recursive CTEs are optimized equivalently.

---

12. Summary Table

Comparison Area	Joins	CTE
Simplicity	Simple for 2–3 tables	Better for 3+ logical layers
Readability	Decreases with size	Highly modular
Reusability	None	Reusable intermediate results

Recursive Support	No	Yes
Performance	Best for flat queries	Equal or better for multi-step
Maintenance	Complex for changes	Easy to modify
Analytical Use	Limited	Highly suitable

---

13. Conclusion

In ABAP on HANA, **Joins are the foundation**, but **CTEs are the architecture** that makes large-scale data logic sustainable.

- Use **Joins** for simplicity and speed when relationships are straightforward.
- Use **CTEs** when logic is layered, recursive, or analytical.

A skilled ABAPer on HANA doesn't choose one over the other — they blend both wisely. The art lies in knowing when a join becomes a bottleneck and when a CTE becomes a necessity.