Enterprise Application-Database Co-Innovation for Hybrid Transactional/Analytical Processing: A Virtual Data Model and Its Query Optimization Needs

Kihong Kim
ki.kim@sap.com
SAP Labs Korea
Seoul, Republic of Korea

Hyunwook Kim
hyunwook.kim01@sap.com
SAP Labs Korea
Seoul, Republic of Korea

Jinsu Lee
jin.su.lee@sap.com
SAP Labs Korea
Seoul, Republic of Korea

Taehyung Lee
taehyung.lee@sap.com
SAP Labs Korea
Seoul, Republic of Korea

Alexander Böhm
alexander.boehm@sap.com
SAP SE
Walldorf, Germany

Norman May
norman.may@sap.com
SAP SE
Walldorf, Germany

Guido Moerkotte
moerkotte@uni-mannheim.de
Universität Mannheim
Mannheim, Germany

Daniel Ritter
daniel.ritter@sap.com
SAP SE
Walldorf, Germany

Ralf Dentzer
ralf.dentzer@sap.com
SAP SE
Walldorf, Germany

Heiko Gerwens
heiko.gerwens@sap.com
SAP SE
Walldorf, Germany

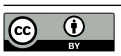
Irena Kofman
irena.kofman@sap.com
SAP SE
Walldorf, Germany

Mihnea Andrei
mihnea.andrei@sap.com
SAP Labs France
Paris, France

Abstract

The advent of hybrid transactional/analytical processing (HTAP) systems has reshaped enterprise data management, breaking the wall between transaction processing and analytics. This paper explores the co-innovation between enterprise applications and databases to advance HTAP architecture and drive its scalable adoption, as exemplified by SAP S/4HANA powered by SAP HANA. At the core of this innovation lies the Virtual Data Model (VDM), which simplifies the formulation of analytical queries directly over transactional data, making HTAP databases more accessible to application developers.

This paper delves into the complexities of the VDM, highlighting key query optimization challenges such as expansive join views, augmentation joins, and augmentation self-joins. Our findings underscore the pivotal role of robust query optimizers in achieving efficient HTAP query execution. By providing actionable insights into the design and optimization of database systems, this paper may serve as motivation for further research into advancing HTAP database technology.



SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724436

CCS Concepts

• **Information systems** → *Enterprise resource planning*; *Enterprise applications*; *Online analytical processing*; **Query optimization**; **Structured Query Language**.

Keywords

HTAP, query optimization, application-database co-innovation, SAP HANA, SAP S/4HANA

ACM Reference Format:
Kihong Kim, Hyunwook Kim, Jinsu Lee, Taehyung Lee, Alexander Böhm, Norman May, Guido Moerkotte, Daniel Ritter, Ralf Dentzer, Heiko Gerwens, Irena Kofman, and Mihnea Andrei. 2025. Enterprise Application-Database Co-Innovation for Hybrid Transactional/Analytical Processing: A Virtual Data Model and Its Query Optimization Needs. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724436

1 Introduction

SAP undertook the challenge of developing a common database approach for OLTP and OLAP by leveraging an in-memory columnar database technology [31]. Gartner coined the term HTAP (Hybrid Transactional/Analytical Processing) [30], while Forrester referred to it as Translytical [50]. SAP HANA, an in-memory columnar database, serves as the foundation for enabling HTAP [32]. SAP S/4HANA, an ERP (Enterprise Resource Planning) software powered by SAP HANA, exemplifies an HTAP system by unifying transactional and analytical processing on a single database system. With in-memory computing and real-time data access, businesses

can derive both operational and analytical insights instantly, enabling better decision making and greater business agility [3, 37].

Technically, SAP S/4HANA executes both transactional and analytical queries on a single set of database tables. Traditional OLTP systems utilize normalized tables, such as the order and order-line table pair found in TPC-C [44]. In contrast, OLAP systems often use star-schema tables, denormalizing the order and order-line tables into a fact table (like the store_sales table in TPC-DS [46]) with corresponding dimension tables created through dimensional data modeling [18]. Query processing benefits from this data organization optimized for analytical queries. The process of replicating transactional data from an OLTP system into separate star-schema tables in a dedicated OLAP system is called ETL (Extract, Transform, Load) and typically performed periodically (e.g., daily or weekly) [19]. Consequently, analytical queries would access stale data.

S/4HANA executes analytical queries directly on its OLTP database without any data replication, providing real-time access to transactional data for analytics [5]. Such an HTAP architecture was not feasible decades ago, necessitating data warehouses dedicated for OLAP workloads. Analytical HTAP queries are inherently more complex and slower than equivalent queries on star-schema tables, as they often require on-the-fly data transformations that ETL processes would handle in advance. Nevertheless, S/4HANA achieves competitive performance levels by capitalizing on modern hardware advancements and SAP HANA's in-memory columnar technology, which unleashes the high computing power of contemporary servers [13, 23].

While the technology to run HTAP workloads within a single database system is available, application developers often struggle to craft analytical HTAP queries directly over transactional tables, which slows down the development of HTAP applications. To overcome this, the Virtual Data Model (VDM) was introduced as the foundation for data access in S/4HANA [29, 38]. The VDM provides a collection of SQL views that expose application data in a standardized, business-oriented format. For example, the *SalesOrder* view presents sales order records in a format meaningful to human users joining multiple tables, and *SalesOrderFulfillmentIssue* combines data from multiple business processes (e.g., sales, delivery, billing, and related purchasing or manufacturing) presenting the combined data in a format easily consumable for identifying fulfillment anomalies.

While the VDM significantly simplifies query formulation and makes HTAP databases more accessible to application developers, it also adds complexity to analytical HTAP queries, necessitating sophisticated query optimizers. Unlike typical analytical queries that involve only necessary tables, VDM views are designed to address a wide range of business needs, often including more tables and fields than required for individual queries. For instance, *SalesOrder* exposes 148 fields related to sales orders, covering customer information, sales personnel, purchase orders, delivery, invoices, and more. While this comprehensive structure supports a variety of queries, most queries only utilize a small subset of these fields. This mismatch presents a significant query optimization challenge: identifying and removing unnecessary operations from VDM queries to ensure efficient execution without compromising flexibility or usability.

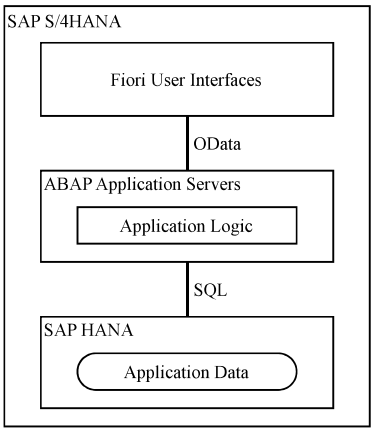


Figure 1: S/4HANA's Three-Tier Architecture

This paper examines the query design patterns employed in VDM-based HTAP queries and identifies key query optimization opportunities, unique to these patterns. Instead of presenting query optimization techniques, this paper focuses on highlighting the optimizations essential for HTAP queries in SAP S/4HANA, explaining why they are necessary, and evaluating their implementation status in SAP HANA and other database systems. By addressing these topics, this paper provides practical insights into the challenges posed by HTAP workloads in enterprise environments and underscores the pivotal role of robust query optimizers in similar systems.

Section 2 provides an overview of SAP S/4HANA, SAP HANA, and the VDM. Section 3 presents a real-world VDM query to establish the overall problem context in a tangible manner. Section 4, 5, and 6 examine three key query patterns in HTAP queries using the VDM, detailing corresponding query optimization requirements and evaluating the implementation status of these optimizations in today's query optimizers. Section 7 discusses SQL language extension requirements, identified while advancing the method of formulating analytical queries on VDM views.

2 SAP S/4HANA and its Virtual Data Model

2.1 SAP S/4HANA

SAP S/4HANA is the core of SAP's Enterprise Resource Planning (ERP) application suite, designed to cover essential business processes, such as accounting, sourcing and procurement, manufacturing, supply chain, asset management, sales, project management, and engineering [3, 37, 51]. Additionally, it provides country-specific and industry-specific solutions across 64 countries and 25 industries, such as retail, automotive, healthcare, and utilities, helping organizations meet unique regulatory and market needs. SAP S/4HANA offers deployment flexibility, allowing it to be hosted on-premises, in a private cloud, or in a public cloud. It is highly configurable, extendable, and designed to integrate smoothly with other business applications.

SAP S/4HANA employs a three-tier architecture, as illustrated in Figure 1. The presentation layer is powered by SAP Fiori, SAP's user experience design framework, which delivers both transactional

and analytical user interfaces. The application layer processes business logic, primarily developed in the ABAP (Advanced Business Application Programming) language. The database layer relies on SAP HANA, SAP's in-memory columnar database technology, to store extensive application data, including around 111,500 tables and 108,500 views, sometimes more than 100 columns per table, and over 150 billion records in a table (such as sales line items for large retail enterprises). Fiori communicates with ABAP servers through OData [17], while the ABAP layer communicates with SAP HANA using SQL.

Beyond transactional interfaces and APIs, SAP S/4HANA offers embedded analytics that enable real-time, multidimensional analysis on transactional data [5]. Capitalizing on SAP HANA's in-memory capabilities, SAP S/4HANA delivers real-time data processing and analytics, leading to enhanced performance, faster transaction processing, and advanced analytical capabilities. This architecture makes SAP S/4HANA ideal for large-scale enterprises aiming to streamline operations, improve decision-making, and respond quickly to changing business demands. Its design emphasizes simplicity and agility, allowing organizations to efficiently manage complex processes.

2.2 SAP HANA

The SAP HANA database manages all the transactional data of S/4HANA while offering efficient access to this transactional data in analytical queries without any redundant copy of the data or the need of ETL processes [10, 15, 23, 25].

SAP HANA offers clients access via SQL interface. SQL requests are parsed and analyzed for semantic correctness using catalog metadata. The parsed query is translated into a relational algebra tree which is first optimized using heuristic rewrite rules, e.g. inlining views or unnesting nested queries. Subsequently, a Volcano-style cost-based optimizer generates a query execution plan relying on data statistics to compute the cost of alternative query execution plans. The query execution plan is then transformed into a push-based plan representation for the HANA Execution Engine (HEX) [20].

HEX leverages just-in-time compilation to efficiently blend precompiled C++ code, e.g. for the data access in the columnar storage, and LLVM code for expressions or even complete operators, e.g. sorting. The pipelined and push-based query processing strategy of HEX helped to significantly reduce the memory footprint of most queries compared to the materializing operators of SAP HANA's legacy query engines.

While also row-oriented tables are supported by SAP HANA, the majority of tables follows a columnar layout where updates are processed by a write-optimized delta fragment which is periodically merged into the read-optimized main fragment of a column. This columnar layout can be compressed using various compression methods reducing the memory footprint of tables and at the same time is the foundation for efficient analytical query processing using vector instruction sets of modern processors, e.g. Intel AVX 512 or ARM SVE [22, 49]. Tables with billions of rows are typically partitioned. S/4HANA carefully tuned the physical layout of tables so that partition pruning can be applied effectively and only hot partitions of frequently updated tables need to be merged when using

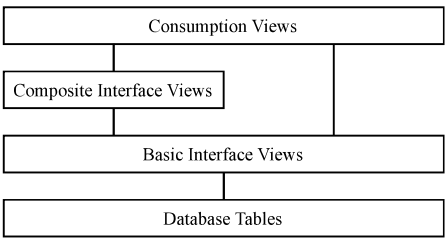


Figure 2: Virtual Data Model Layers

range-partitioning based on time. In general, in SAP HANA secondary indexes are used much less frequently than in row-oriented databases. On top of the compression of columnar data, tables can be stored in a page-wise fashion avoiding to load complete columns when accessing data. Instead, only accessed pages are loaded into an in-memory page buffer and evicted as needed. This page-oriented layout in SAP HANA is called *native storage extension* (NSE) [39], and S/4HANA uses NSE, e.g., for storing write-mostly data like journals of change documents. Switching between page-based vs. column-based organization of a column table is easy by changing the metadata of the table and reloading the column, partition or table.

Because SAP HANA supports transactional workloads with millions of transactions per hour concurrently with analytical workload it implements a MVCC-based concurrency scheme. ARIES-style write-ahead logging makes sure that all committed changes are stored in durable storage when a transaction is committed.

SAP HANA offers enterprise features like HA/DR as needed by business-critical applications like S/4HANA. Further features of SAP HANA that are not in the focus of this paper include support for graph-data processing, an integrated JSON-document store, native support for hierarchies and an integrated vector store.

2.3 Virtual Data Model

Application developers require simple and intuitive database interfaces. While S/4HANA application developers are highly skilled in specific business domains and proficient in the imperative programming languages, such as ABAP, used in those domains, they are not necessarily SQL experts. Many have limited knowledge of relational algebra and are unfamiliar with DBMS-level optimizations. Furthermore, despite its immense success, SQL is an old language and can be challenging to learn and use effectively[40].

The basic idea of SAP S/4HANA's Virtual Data Model (VDM) is to offer all application data via standardized business-oriented views that are easy to understand and easy to consume [29]. For the definition of views, SAP's Core Data Services (CDS) technology is used [8, 24]. VDM views are modeled in CDS and deployed as SQL views into the database. Cryptic technical details and names in historically grown database table designs are hidden by VDM views. In addition, VDM views are enriched with semantical information and connected to other VDM views by CDS associations. These associations can be used in a CDS path notation to add fields from the associated view - an easy and convenient way to join a view and project columns from it.

VDM consists of multiple layers of views, each serving a distinct purpose, as shown in Figure 2:

- Basic view layer: Close to the database tables, these views introduce VDM's added value of business terminology, semantics, and associations.
- Composite view layer: Built upon the basic views, this intermediate layer is designed to serve functional purposes, such as transactional processing, data extraction, or analytics. It may transform and de-normalize the data according to application-specific needs.
- Consumption view layer: This top layer offers views that are tailored for specific user interfaces and APIs or that define specific analytical queries. These views are either deployed as database views or retained as pure VDM artifacts, from which the consumption infrastructure generates a query and executes it on demand. They are constructed upon basic or composite views.

While basic and composite views expose all data (rows and columns) to support a wide range of use cases, consumption views are specifically designed for their intended tasks. These views include only the necessary data and should not be used outside their designed purposes. For consumption scenarios not predefined by SAP S/4HANA, VDM allows customers to construct simple queries from a broad composite view or precise but complex queries from lean basic views. As of 2024, the number of basic/composite/consumption views amounts to over 7,500 [36]. Some consumption views are deeply nested, with the highest nesting depth reaching 24.

3 Motivating Example

Figure 3 shows the logical query plan of a seemingly simple VDM query, "`select * from JournalEntryItemBrowser`". To illustrate the raw complexity of VDM views, all nested views are unfolded and no optimizations have been applied. The query plan comprises 47 table instances, 49 joins, one five-way UNION ALL, one GROUP BY, one DISTINCT, and many selections and projections in the DAG form. SAP HANA is able to share a subquery in a query plan, forming a DAG instead of a tree. When unshared, the number of table instances increases from 47 to 62.

`JournalEntryItemBrowser` is one of the most frequently accessed VDM views in SAP S/4HANA Cloud Public Edition. It is centered around the ACDOCA table, one of the core tables in SAP S/4HANA. ACDOCA, known as the universal journal table, serves as a unified source of all financial transactions at the line-item level. This query plan can be decomposed into three parts.

- The three-way join in the lower left corner is the core of this view, forming a composite interface view for ACDOCA. A global enterprise usually has multiple legal entities (named Company in SAP S/4HANA) and multiple ledgers. By combining ACDOCA with the company table and the ledger table, users can access a specific ledger for a specific company.
- The interface view is then augmented through 30 many-to-one left outer joins with various other views, forming a consumption view. Its maximum nesting depth is 6.
- The consumption view is protected with record-wise data access control (DAC), filtering out the records that a user is not authorized to access [2]. The DAC filter is automatically injected per user when quering, further increasing the complexity of VDM queries.

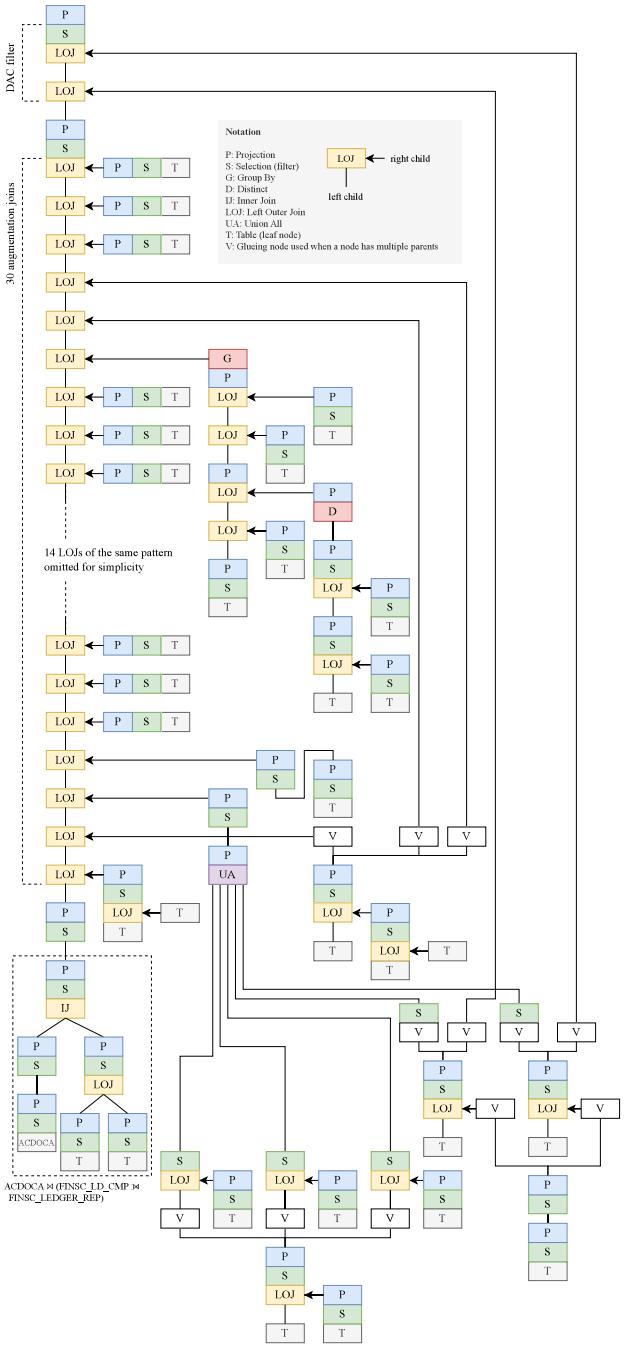


Figure 3: A Typical VDM Query (`select * from` JournalEntryItemBrowser)

However, processing such a complex query plan in its entirety is not the intent behind VDM. VDM designers expect modern query

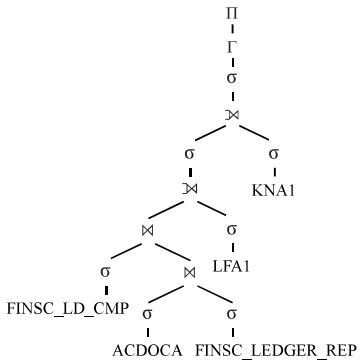


Figure 4: An Optimized Query Plan for "`select count(*) from` JournalEntryItemBrowser"

optimizers to simplify these complex views for each query, retaining only the necessary operations and eliminating unnecessary ones. Figure 4 shows an optimized query plan for "`select count(*) from JournalEntryItemBrowser`", showcasing the extent to which such a complex view can be optimized. The two many-to-one left outer joins in Figure 4—one with LFA1 (supplier data) and the other with KNA1 (customer data)—are not removed because they are used in the data access control filters placed above each of them. The rest of the joins are pruned out as they don't change the cardinality due to the many-to-one left outer join semantics.

This view can be seen as a virtual star schema, with all dimensions pre-joined into the fact table, ACDOCA. The key advantage is the ability to operate directly on transactional data without ETL-based replication, meaning real-time analytics. Its downside is that calculations and transformations are performed on the fly during each query execution, rather than being pre-computed once during ETL.

Key characteristics of VDM views include:

- **Rich set of fields**: VDM views provides a comprehensive selection of fields through joins, reducing the need for additional joins in queries. However, this adds complexity to the views, requiring query optimizers to remove unnecessary joins.
- **Minimal use of filters and aggregations**: VDM views intentionally minimize the use of filters and aggregations, ensuring their versatility across diverse use cases. Aggregations are typically placed higher in the stack, either in consumption views or in queries built atop. Advanced optimizer logic is essential to push these operations as far down as possible.
- **Incorporation of calculations**: Business-specific calculations are incorporated as view fields, simplifying application development while maintaining the view's versatility. As these views work directly with transactional data, the calculations are performed on the fly during each query execution, not once during ETL. Advanced optimizer logic is again essencial to minimize the impact of on-the-fly calculations.

This paper explores the challenges and solutions associated with the VDM. Formulating analytical queries over HTAP tables is inherently complex. The VDM simplifies the formulation of analytical queries for application developers, but it introduces additional layers of complexity, significantly complicating query optimization. This paper highlights some challenges for query optimizers in such a widely used and business critical application. These insights may serve as motivation for further research into effective methods for formulating analytical HTAP queries and developing advanced query optimization techniques to address these challenges

Note that views can be materialized for query performance. SAP HANA provides static cached views (SCV) and dynamic cached views (DCV). They are primarily materialized in memory and thus called cached views. SCV is refreshed periodically, providing a delayed snapshot of view. DCV is incrementally maintained, providing the up-to-date snapshot.

4 VDM Query Optimization

The VDM provides a set of predefined SQL views, enabling application developers to choose the most appropriate view for a given business context and write queries without needing to manually handle joins. This simplifies the process of writing correct and performant queries.

4.1 Requirement: Expansive Join Views

Many VDM views are expansive, sometimes joining over 100 tables and exposing hundreds of fields. This expansive structure allows each view to support a broad range of business queries. In contrast, creating highly specialized views for each specific business context is generally undesirable, as it not only shifts development effort from application developers to VDM view designers but also drastically increases the number of views, which increases maintenance efforts and adds complexity to schema evolution.

For example, consider a view that joins a sales transaction table with various master data tables—such as customers, suppliers, stores, cost centers, and general ledger accounts—as well as administrative data tables like regions, countries, and calendar data, often used to create hierarchies. This type of broad view supports multiple analytical scenarios, for instance, customer-focused, country-focused, and supplier-focused revenue analyses on a single view.

However, expansive VDM views do not necessarily require that all joins be executed. Typically, a query accesses only 10-20 fields out of hundreds, making many joins and other relational operations in the view unnecessary. Therefore, optimizing out these redundant operations is essential; otherwise, queries on VDM views may run significantly slower than handcrafted queries executed directly on the tables, reducing the benefit of easier query formulation.

The expansiveness of VDM views partly stems from nesting, where views are often built on top of others, with the highest nesting depth reaching 24. Since a view rarely uses all the information from its nested views, certain joins become superfluous at each subsequent level. These unnecessary joins from nested views contribute to making the outermost view even more expansive than originally intended.

4.2 Modeling Pattern: Augmentation Join

As described earlier, application requirements led us to encounter queries containing numerous unnecessary joins that ideally should be optimized out. We introduce the term **augmentation join** (**AJ**) to denote this query pattern frequently observed in VDM views. In a join $R \bowtie S$, a record $r \in R$ is augmented with $s \in S$, filtered out if no matching $s \in S$ is found, or duplicated if multiple $s \in S$ records match. An augmentation join, however, is purely augmentative, meaning it neither filters nor duplicates any $r \in R$. Therefore, an augmentation join can be removed if no fields introduced by it are used in the query. We refer to such removable augmentation joins as **unused augmentation join**s, or **UAJ**s. The left child of an augmentation join is referred to as the **anchor** while the right child is the **augmenter**.

To remove a UAJ, it is essential to determine whether a join is purely augmentative and does not filter or duplicate records [4]. We identified two types of many-to-one joins that meet this criterion:

- (**AJ 1**) many-to-exact-one inner join, specifically, $1..m : 1..1$
- (**AJ 2**) many-to-one left outer join, specifically, $1..m : 0..1$

where m is a positive integer, and $1..m$ denotes a join cardinality with a lower bound of 1 and an upper bound of m. One-to-one joins are not treated separately because they are a subset of many-to-one. The right-side join cardinality $1..1$ implies exactly one match, ensuring that no records from the left relation are filtered or duplicated. The left outer join, on the other hand, doesn't filter records even when no match is found, instead augmenting with NULL values. This allows the lower bound to be relaxed to zero for left outer join, denoted as $0..1$.

To identify these augmentation joins, it is necessary to determine the join cardinality of the right child. We categorized our findings into four cases:

- (**AJ 1a**) Inner equi-join based on a foreign-key constraint: This is a typical example of a many-to-one join. Although not uncommon, foreign key constraints are infrequent within the SAP ecosystem, making this case rare in S/4HANA.
- (**AJ 1b**) Inner equi-self-join on key: This will be separately discussed in section 5.
- (**AJ 2a**) Left-outer equi-join on a unique field (or fields): Joining on a unique field ensures at most one match, making this a subset of AJ2 and the most frequently encountered case.
- (**AJ 2b**) Left-outer theta-join with an empty relation or $R \rhd\!\bowtie\! \phi$: This means a many-to-zero join, a subset of AJ 2. The same logic applies to equi-join. This occurs when an always-false filter is applied to the right relation, making it a rare case.

Case AJ 2a is the most common case in SAP S/4HANA. It requires verifying the uniqueness of the join field on the right relation. We identified three possible scenarios for a join like R `LEFT JOIN` S `ON` $R.a = S.x$, where a and x are either a single field or a composite one:

- (**AJ 2a-1**) $S.x$ is from a table and is guaranteed to be unique, for instance, by a uniqueness or primary key constraint.
- (**AJ 2a-2**) $S.x$ is the grouping key from a `GROUP BY` operation.

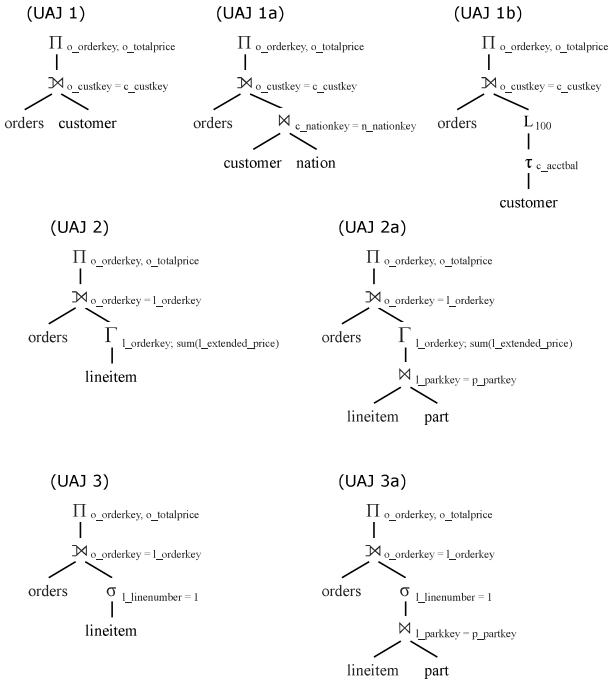


Figure 5: Basic UAJ Queries

- (**AJ 2a-3**) The pair $(S.x, S.y)$ is unique (by one of the scenarios above) and $S.y$ is restricted to a constant value through a filter such as $S.y = 1$.

4.3 Unused Augmentation Joins

UAJ optimization doesn't demand novel algorithms but does require strong engineering to accurately derive join cardinality and determine whether augmenter relations are necessary for subsequent operations, especially in complex queries [7]. These two functions are well-known and are implemented to varying degrees in modern query optimizers. However, we observed that many query optimizers fail to optimize not only complex VDM queries but also very simple queries, likely because UAJ optimization has not been a priority for them.

Figure 5 shows 7 simple UAJ queries based on the TPC-H schema [45]. It is assumed that primary keys are defined according to the benchmark but optional foreign-key constraints are omitted. All seven queries can be optimized into a single projection operation with all other operations removed.

- UAJ 1 is the simplest form of AJ 2a-1, where the join field *c_custkey* is unique on the right relation because it is the primary key.
- UAJ 2 is the simplest form of AJ 2a-2, where the join field *l_orderkey* is unique because it is the grouping key of group-by operation.
- UAJ 3 is the simplest form of AJ 2a-3, where the join field *l_orderkey* is unique because the pair $(l\_orderkey, 1)$ forms a composite key.

- UAJ 1a, 2a, and 3a increase complexity marginally by adding a non-duplicating join to the augmenter table. It checks if UAJ optimization applies not only to tables (or leaf nodes) but also to sub-queries (or intermediate nodes), correctly deriving the uniqueness property along a query plan graph.
- UAJ 1b increases complexity by adding an order-by operation followed by a limit operation on top of the augmenter table. Note that both operations don't change the uniqueness of fields.

We evaluated five DBMSs (SAP HANA Cloud, PostgreSQL 17, and three of the most popular commercial RDBMSs) to assess their ability to optimize the seven simple UAJ queries shown in Figure 5. This evaluation involved creating a TPC-H schema with primary keys, loading data, and analyzing query execution plans for the seven queries through EXPLAIN PLAN or equivalent tools. Table 1 summarizes the findings. The SAP HANA optimizer successfully removes UAJs for all queries. Postgres handles UAJ 1, 2, 3 and 2a effectively. The three commercial systems show varying levels of support for optimizing these queries.

Table 1: UAJ Optimization Status

	HANA	Postgres	System X	System Y	System Z
UAJ 1	Y	Y	-	Y	Y
UAJ 2	Y	Y	-	-	Y
UAJ 3	Y	Y	-	Y	Y
UAJ 1a	Y	-	-	-	Y
UAJ 2a	Y	Y	-	-	Y
UAJ 3a	Y	-	-	-	Y
UAJ 1b	Y	-	-	-	-

4.4 Paging Queries with Augmentation Joins

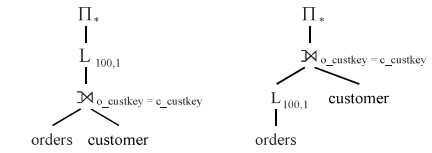
Paging queries are frequently used in SAP S/4HANA to retrieve a specific subset or "page" of records from a large dataset, particularly in scenarios involving UI features such as paginated tables or infinite scrolling. A typical paging query looks like `"select * from Foo limit 100 offset 1"`, which retrieves 100 records starting from the 1st record (or the first page of results).

Figure 6 shows a simple paging query with an augmentation join and its corresponding optimized query plan. The limit operation can be pushed down across the augmentation join, which is a critical optimization for query performance. For example, this directly impacts which side of the join builds the hash table in a hash join operation, significantly influencing efficiency.

Although the optimization of limit pushdown across augmentation join is straightforward, it is not implemented in most optimizers. Table 2 shows the implementation status of the five selected optimizers for the very basic query in Figure 6. SAP HANA alone implements this optimization.

Table 2: Limit-on-AJ Optimization Status

	HANA	Postgres	System X	System Y	System Z
Fig. 6	Y	-	-	-	-



(a) Simple Paging Query (b) Optimized Query Plan

Figure 6: Limit on Augmentation Join

4.5 Lessons Learned

From our collaboration with the application development team, we gained the following insights:

- Applications tend to avoid foreign key constraints, relying on application-side means to ensure referential integrity. For instance, an ABAP application transaction may consist of multiple database transactions, and it is a common practice to verify input data in the end of an application transaction.
- Left outer joins are far more common than inner joins, reflecting the need to handle missing data gracefully.
- Augmentation joins are widely used, especially in modeling-based applications like SAP S/4HANA.
- Unused augmentation joins are also common in order to make views reusable for various business queries. Optimizing them out can provide significant performance gains.
- Limit operations are widely used for paging queries. It is crucial to properly push them down across augmentation joins.

5 Extending VDM with Custom Fields

One of SAP S/4HANA's core strengths is its ability to support custom extensions, offering enterprise customers the flexibility to tailor the system to their specific needs while ensuring stability and compatibility with future S/4HANA upgrades. Business experts or implementation consultants can customize applications, user interfaces, reports, forms, and more. An important custom extension scenario from a database perspective is adding custom fields to SAP-managed database tables and making these custom fields available across SAP-provided UIs, reports, forms, free-text search, and other business scenarios [34].

Figure 7 illustrates such an extension scenario with an SAP Fiori app. Fiori, SAP's user experience design framework, allows customers to extend app screens (for instance, analysis report or a transaction input form) by adding custom fields, which are ultimately stored in SAP-managed database tables by adding custom fields. The app accesses these tables via the OData protocol [17], using SAP-managed VDM views. To expose custom fields correctly, both VDM and the corresponding database views must be extended appropriately. SAP leverages the Core Data Services (CDS), SAP's data modeling framework, to manage VDM views at the app server level and the related SQL views at the database level [24].

This section examines how to extend database views to expose custom fields in an upgrade-safe manner. The extension process at the application layer (app server, gateway, and Fiori UI) is beyond the scope of this paper.

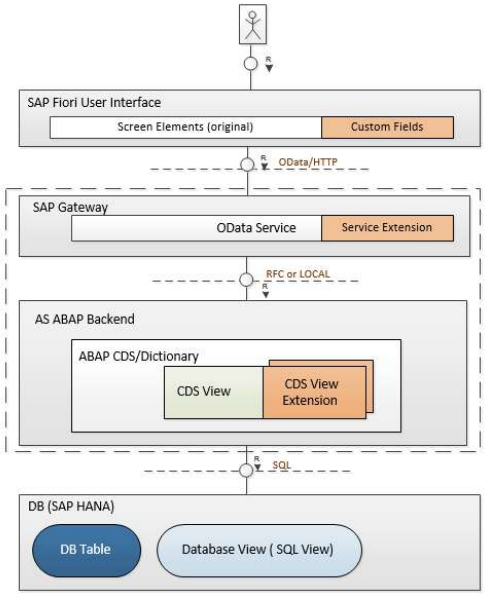


Figure 7: Extending Apps with Custom Fields

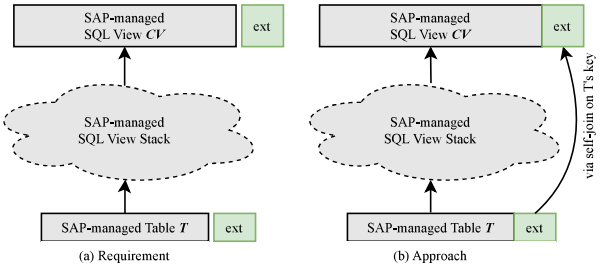


Figure 8: Custom Fields Extension in the DB Perspective

5.1 Requirement: Custom Fields Extension

Figure 8(a) illustrates the need to extend an SAP-managed consumption view *CV* in order to expose the extension field *ext* added to table *T* by a customer. A straightforward approach might be redefining *CV* to include *ext*. However, this method requires cascading redefinitions of interim views in the view stack between *CV* and *T*. This cascading redefinition is problematic because it is not upgrade-safe. While SAP ensures the stability of certain views to guarantee that customer applications remain unaffected by SAP S/4HANA upgrades, this stability contract does not extend to many interim views, which are SAP-internal. As a result, the cascading redefinition method is not a viable solution.

Figure 8(b) illustrates SAP's approach, which leverages a self-join on the key field. *CV* is redefined to include *T.ext* through an augmentation join with table *T* on its key. This method modifies table *T* at the bottom to add custom fields and updates the consumption view *CV* on the top to expose custom fields, while keeping interim views unchanged. Although this approach effectively handles the functional requirement of exposing *ext* without cascading view

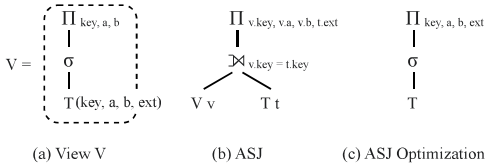


Figure 9: Augmentation Self-Join

redefinitions, it introduces a query optimization challenge. The additional self-join is both costly and redundant, necessitating its optimization to improve query performance.

5.2 Modeling Pattern: Augmentation Self-Join

Figure 9 illustrates SAP's query design pattern, called the Augmentation Self-Join (ASJ), used to expose fields that are not projected in an existing view. In Figure 9(a), the view *V* does not expose the custom field *ext*, a situation that arises when a customer extends an SAP-managed table with custom fields. Figure 9(b) demonstrates how the *ext* field can be exposed by performing a self join with table *T* on its key. While both inner joins and left outer joins are viable, the latter is used here for consistency with section 4. This technique works when *V* already projects the key field of *T*.

An important point is that this additional self-join doesn't degrade query performance if properly optimized. Figure 9(c) shows an equivalent query plan, where the ASJ is optimized out. ASJ is a query design method that allows unprojected fields to be exposed without sacrificing query performance.

ASJ is a special form of augmentation join (AJ). While AJ can be removed when unused, ASJ can be removed even when in use. This is because augmenter fields are internally accessible in the left child, the original view, even if they are not initially projected.

5.3 Optimizing Augmentation Self-Join

ASJ is technically a self-join on key, which can be easily optimized out with field accesses wired to the left relation. However, we found that other query optimizers don't optimize out even obvious ones. This section examines basic ASJ queries that can be optimized out.

Figure 10(a) shows a very simple query with a self-join on key and its optimized query plan. Interestingly, none of query optimizers we checked optimizes this simple query. Perhaps, this has been regarded as an inferior query formulation, and it hasn't been a priority to optimize such queries. Our argument is that this is a useful query pattern, worth optimizing, as described earlier.

Figure 10(b) shows an ASJ optimization when the anchor relation is a sub-query. An ASJ can be removed when references to augmenter fields can be re-wired to the anchor sub-query. In other words, the same table exists in the anchor sub-query, and the required fields are accessible. While it is not the scope of this paper to investigate when such re-wiring is possible, there are two points worth mentioning. First, projection operations don't block ASJ optimization because an optimizer can modify them to expose un-projected fields. Second, outer joins need to be carefully checked to see the effect of NULL values generated for unmatched records, which can prevent the ASJ from being removed.

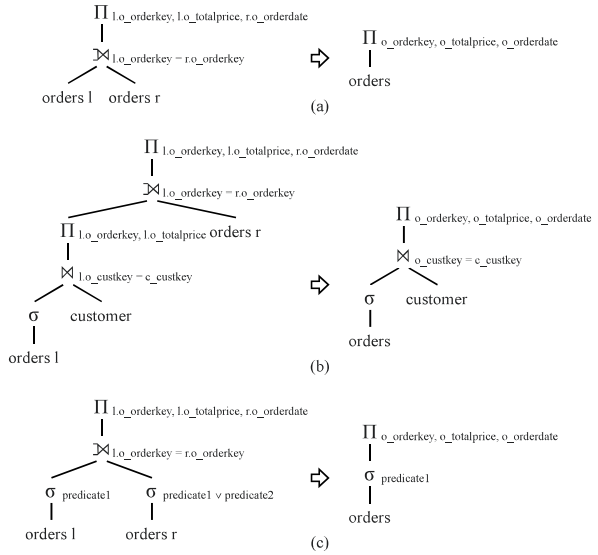


Figure 10: ASJ Optimization Examples

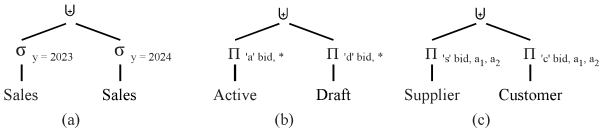


Figure 11: Union All Patterns in S/4HANA

Figure 10(c) shows the case when selection is placed on the augmenter relation. The ASJ optimization is applicable when the augmenter predicate subsumes the anchor predicate. Otherwise, some anchor records will be augmented with NULL values, which don't exist in the source table, preventing the ASJ from being removed.

Table 3 summarizes how the five selected optimizers handle the basic ASJ queries in Figure 10. Except for the SAP HANA optimizer, the rest ignores the self-join optimization opportunity.

Table 3: ASJ Optimization Status

	HANA	Postgres	System X	System Y	System Z
Fig. 10(a)	Y	-	-	-	-
Fig. 10(b)	Y	-	-	-	-
Fig. 10(c)	Y	-	-	-	-

6 Optimization for Union All in VDM

6.1 Union All Patterns in VDM

Figure 11 shows three frequent patterns of Union All, encountered in the VDM. Figure 11(a) illustrates a query design pattern, where each child represents a distinct subset of the same relation, formulated with a selection operation. This pattern usually occurs when each child is a complex view, where multiple joins exist in-between.

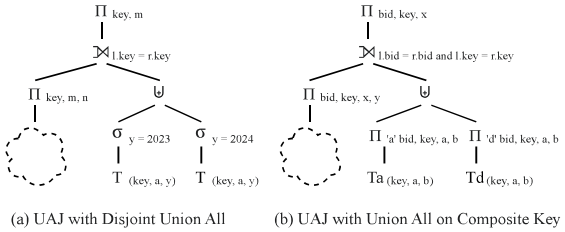


Figure 12: Unused Augmentation Join with Union All

Figure 11(b) illustrates a table design pattern to support RESTful communication between apps and users [33]. Modern cloud apps typically employ a stateless communication model, allowing incoming requests to be distributed across multiple backend servers for load balancing, elastic scaling, and availability. Although the backend app servers are stateless, applications are stateful from the user's perspective, meaning user-entered data must be stored, validated, and enriched throughout the business process flow. This in-progress data is temporarily stored in a separate database table, called the Draft table. While analytical queries operate solely on the Active table, operational queries combine data from the Active table with session-specific data from the Draft table.

Figure 11(c) occurs although it seems pointless due to the information gap between the application and the database. Since apps are usually designed in an object-oriented way, they may represent *Supplier* and *Customer* as two subclasses of a common *Company* class in a B2B context, which are then mapped to two separate database tables [41, 43]. In this setup, Union All of *Supplier* and *Customer* becomes meaningful to consolidate information from both entities.

6.2 Union All and UAJ Optimization

Union All can appear in augmentation joins, requiring advanced optimizer logic for effective UAJ optimization. Figure 12 illustrates two forms of augmentation joins with Union All, which can be optimized out when unused.

Figure 12(a) depicts a left outer join with a Union All operation over two distinct subsets of table T. If $T.key$ is a unique field of T, its uniqueness is preserved after this form of Union All operation. This ensures the join is purely augmentative, enabling the UAJ optimization.

Figure 12(b) depicts a left outer join with a Union All operation following the pattern shown in Figure 11(b) or (c). If $T_a.key$ and $T_d.key$ are unique fields of T_a and T_d, respectively, the composite join fields <bid, key> are unique because bid (branch ID) is uniquely assigned to each Union All child. This property enables the UAJ optimization. Note that the five-way Union All in Figure 3 follows the pattern of Figure 11(c) and is removed by UAJ optimization in Figure 4.

Table 4 summarizes the extent to which the five optimizers implement UAJ optimization involving Union All. Apart from SAP HANA, none of the optimizers were found to support these optimizations.

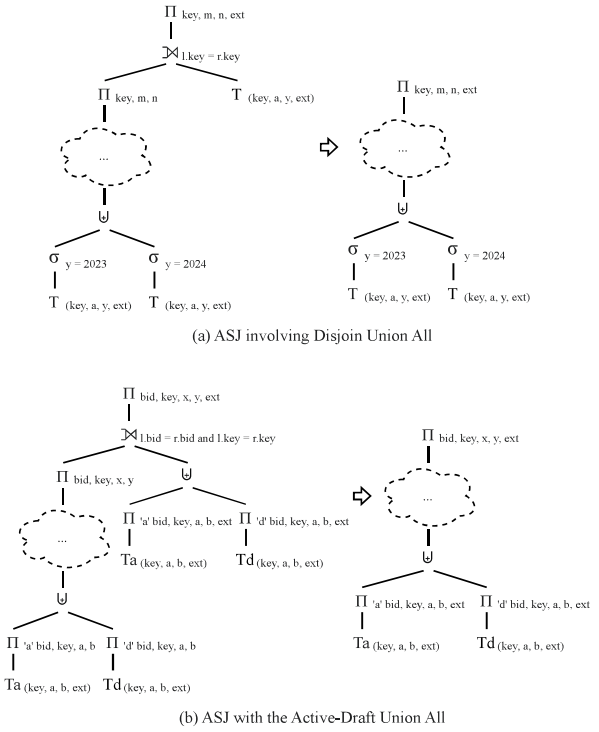


Figure 13: ASJ involving Union All

Table 4: UAJ Optimization Status for Union All

	HANA	Postgres	System X	System Y	System Z
Fig. 11(a)	Y	-	-	-	-
Fig. 11(b)	Y	-	-	-	-

6.3 Union All and ASJ Optimization

Union All also appears in the context of ASJ optimization. Figure 13 illustrates two such patterns encountered in SAP S/4HANA VDM queries.

In Figure 13(a), a Union All operation appears in the anchor relation, and the self-join table is present in both children of the Union All. This Union All follows the pattern described in Figure 11(a). The ASJ optimization logic can be easily extended for this query. It begins by determining that the left outer join is purely augmentative because $T.key$ is unique. It then traverses the left subgraph to identify ASJ optimization opportunities. Upon encountering the Union All, it checks if each child of the Union All forms a self join. In this query, both children involve T and form a self join. Consequently, ASJ optimization is applied, producing the optimized query plan shown on the right side of Figure 13(a).

In Figure 13(b), Union All operations appear on both sides of a join operation, theoretically forming an ASJ. This ASJ pattern occurs when the draft table pattern, shown in Figure 11(b), is combined with the custom fields extension. From the application's perspective, the Union All of Active and Draft tables (T_a and T_d) is regarded as a single logical table. The custom fields extension adds an extension field (ext) to this logical table, resulting in the ext field being added to both T_a and T_d. Then, an ASJ with this logical table is introduced to expose ext, as depicted in Figure 8.

Recognizing this ASJ pattern involving Union All operations on both sides of a join is complex and challenging, though not impossible. Furthermore, it is not a practical trade-off to increase the query optimization time by routinely checking this optimization opportunity in every Union All scenario because this pattern is relatively uncommon.

To address this challenge, we introduced a mechanism for application developers to explicitly indicate the intention to perform an ASJ with a Union All operation. This is achieved through an extension to the HANA SQL syntax with a new join type called a case join, specifically designed to express an ASJ involving multiple relations [35]. With the ASJ intention explicitly indicated, executing more complex optimizer logic becomes a highly advantageous trade-off. While it increases query optimizing time, it results in a significantly reduced query execution time, yielding substantial performance benefits.

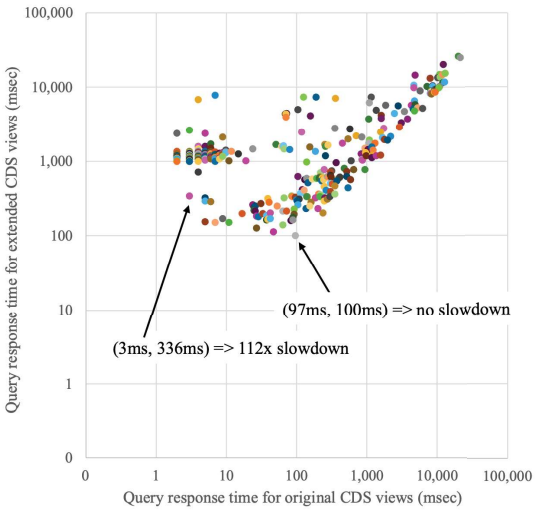
Figure 14 demonstrates the impact of this ASJ optimization extension on query execution time. A simple paging query, "`select * from V limit 10`", was executed, replacing `V` with two forms of 100 VDM views: the original view and an extension view that include a custom field. Ideally, both queries, one on the original view and the other on the extension view, should exhibit similar execution times, while the latter involves an additional query optimization overhead and a minor execution time increase to project the custom field. In such an ideal scenario, data points representing the execution times would align along the diagonal line.

The desired result was achieved, as shown in Figure 14(b), after properly implementing the new case join on the database side and adopting it on the application side. The average execution time were measured on the application server over five runs, excluding the query optimization time. Any deviations from the diagonal line are considered measurement errors.

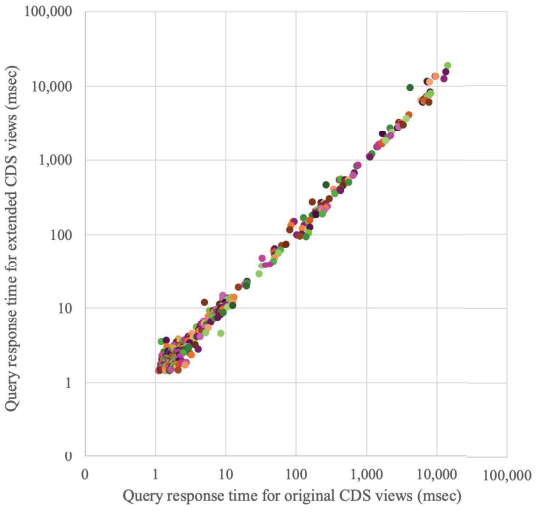
Figure 14(a) exhibits the challenges encountered when attempting to recognize ASJ patterns involving Union All without knowing the ASJ intention. While some ASJ patterns using Union All were successfully recognized and optimized, many were not, as evidenced by many data points significantly above the diagonal line. When patterns were not recognized, queries on extended views were up to 2~3 orders of magnitude slower than those on the original views.

Explicitly indicating the ASJ intention in a query is crucial to query optimization when Union All is involved. A Union All subgraph can take various forms during query optimization due to various query transformations such as filter pushdown, projection pullup, join through union all, and so on [48]. Checking for ASJ optimization opportunities in every variant of Union All is computationally expensive. By explicitly indicating the ASJ intention, the query optimizer can preserve the augmenter-side Union All subgraph, restricting query transformations with the Union All suggraph unless clearly advantageous. This approach significantly reduces the number of alternative query plans to consider, making it easier to identify ASJ optimization opportunities.

Since other query optimizers do not implement ASJ optimization, as shown in table 3, they also lack its extension for Union All, making this optimization unique to the SAP HANA query optimizer.



(a) without an SQL extension



(b) with an SQL extension

Figure 14: Performance Impact of Optimization for Custom Fields Extension

7 SQL Extension Requirements in VDM

This section presents SQL extension requirements encountered during the development of analytical HTAP queries. Over time, ANSI SQL has been extended multiple times to support analytical queries, introducing features such as rollup and grouping sets. Here, we address additional requirement that emerged from our work.

7.1 Aggregation Pushdown across Decimal Rounding

Aggregation pushdown is a powerful optimization technique that can dramatically reduce the number of records to process, thereby speeding up subsequent operations. However, we encountered unexpectedly slow VDM queries, where decimal rounding blocked this optimization to avoid discrepancies in insignificant trailing decimal digits. Addressing this issue requires an SQL extension to allow controlled precision loss in aggregated decimal values, such as monthly revenue or yearly revenue.

Decimal rounding is common in business applications and carefully handled to ensure consistent and accurate results. For example, an 11% tax on a $119.95 item calculates to $13.1945, which is then rounded to $13.19. Another important scenario for rounding is foreign currency conversion, such as converting an amount in USD to EUR using the exchange rate on a specific date. These conversions are often performed dynamically during query processing, looking up exchange rate tables. Note that different business scenarios may require different conversion dates, such as transaction dates or quarter-end closing dates, affecting exchange rates.

Decimal rounding is not interchangeable with addition, thereby preventing the pushdown of the *sum* aggregation. For instance, $round(1.3)+round(2.4)$ evaluates to 3 when rounding is performed first, whereas $round(1.3+2.4)$ evaluates to 4 when addition is performed first. Similarly, $\texttt{sum}(\texttt{round}(price*1.11, 2))$ cannot be rewritten into the seemingly more efficient form $\texttt{round}(\texttt{sum}(price)*1.11, 2)$.

However, many analytical users are not concerned with minor discrepancies in aggregated decimal values and prioritize enabling aggregation pushdown across decimal rounding to improve query performance. To meet this need, SAP HANA introduced an SQL extension, named *allow_precision_loss*, allowing users to explicitly indicate their preference for query performance over minor inaccuracies in decimal aggregates on a per-query basis. By leveraging this, SAP HANA can interchange addition and decimal rounding and enumerate more alternative query plans. For example, SAP HANA treats the following two queries as equivalent.

```
select allow_precision_loss(
         sum(round(price*1.11, 2))
       )
  from SalesOrder

-- equivalent query by allow_precision_loss
select round(sum(price)*1.11, 2))
  from SalesOrder
```

7.2 Reusing Calculation Formulas over Aggregates

VDM views often expose calculation formulas as view fields, abstracting their complexity from end-users and enabling formula reuse without repetition. For instance, TPC-H calculates revenue as $sum(extendedprice*(1-discount))$. By exposing this calculation as a distinct view field (e.g., *revenue*), queries become more concise and readable. Once *revenue* is defined in an aggregated, order-level view, higher-level aggregations such as monthly revenues can be easily computed using $sum(revenue)$ in conjunction with an appropriate GROUP BY clause.

However, this approach doesn't extend to non-additive calculations over aggregate values. For example, margin is typically calculated as $sum(profit)/sum(revenue)$, a ratio of two aggregate values that is inherently non-additive. In such cases, higher-level

aggregates (e.g., monthly margins) cannot be derived from lower-level aggregates (e.g., daily margins). Consider a simple example: a 10% margin on $100 revenue on day 1 and a 20% margin on $900 revenue on day 2. The average of 10% and 20% is 15%, but the correct overall margin is 19%, reflecting the revenue weighting.

To support calculation formulas over aggregate values, SAP HANA introduced the concept of expression macros, inspired by macros in programming languages. The following query example illustrates how expression macros define complex aggregate expressions once and reuse them across queries.

```
-- Define expression macros once
create view vLineitem as
select *
  from lineitem join partsupp on l_partkey =
      ps_partkey and l_suppkey = ps_suppkey
  with EXPRESSION MACROS(
         1 - sum(ps_supplycost)/sum(
             l_extendedprice*(1-l_discount))
         as margin)

-- Reuse expression macros across queries
select o_custkey, EXPRESSION_MACRO(margin)
  from vLineitem
 group by o_custkey
```

7.3 Specifying Join Cardinality

As demonstrated in this paper, identifying whether a join is purely augmentative is crucial for optimizing analytical VDM queries and achieving reasonable query performance. While augmentation joins can often be inferred from uniqueness constraints, these constraints are not always practical or suitable for every application.

Uniqueness constraints introduce storage and computational overheads, as they typically require the creation of an index that must be accessed for each record update [14]. Furthermore, they can impose unnecessary restrictions on application design. Applications often validate data toward the end of a transaction, but uniqueness constraints demand that every SQL statement adhere to the constraints prior to being committed. This rigid enforcement can significantly limit design flexibility.

To address the need for formulating augmentation joins without solely relying on uniqueness constraints, SAP HANA extends the join syntax to support join cardinality specifications. For instance, a regular left join, `R left outer join S`, can be enhanced with a cardinality specification, such as `R left outer many to one join S`. This indicates that a record in `R` can join with zero or one record in `S`, achieving the same effect as uniqueness constraints without the associated overhead.

Unlike uniqueness constraints, join cardinality specifications are not enforced by the database system, leaving their use to the discretion—and risk —of application developers. To mitigate the risk, SAP HANA offers a tool that verifies whether the specified join cardinality in a query aligns with the actual data. This tool enables application developers to ensure data consistency while enjoying greater design flexibility and performance optimization.

8 Related Work

HTAP is still a relatively new area of study. While recent surveys have examined HTAP database systems [27, 42, 52], there is limited research on the workloads that HTAP databases are expected to support in the context of comprehensive enterprise applications such as SAP S/4HANA. Distributed transactional workloads in SAP R/3, a decades-old predecessor to SAP S/4HANA, has been studied [9, 51], but comparable studies for HTAP scenarios are limited.

This paper presents practical insights into analytical HTAP queries, emphasizing the critical role of advanced query optimizers in eliminating unnecessary operations. The elimination of redundant relational operations has been extensively studied in database research. Redundant joins, for instance, frequently arise in scenarios involving queries over views [4, 6, 21, 26] or queries generated by front-end tools [11, 12]. However, as this paper demonstrates, such optimizations are still not fully realized in today's query optimizers. A similar optimization principle underlies late materialization techniques in columnar databases [1].

Additionally, this paper presents the need for SQL extensions to support analytical HTAP queries. A recent SQL extension proposal, AS MEASURE [16], marks a meaningful step in this direction. It enables the reuse of calculation formulas over aggregates, described in section 7.2. It also enables formulating multi-row calculations (such as year-over-year growth when each row corresponds to a different year), which is referred to as calculated measures in multidimensional query languages [28, 47] and partially supported in SQL through window functions.

9 Conclusion

The co-innovation of SAP S/4HANA and SAP HANA demonstrates how modern enterprise systems can unify transactional and analytical processing to deliver real-time analytical insights. The Virtual Data Model simplifies query formulation, empowering HTAP application developers while introducing unique challenges for query optimizers. By addressing expansive join views, augmentation joins, augmentation self-joins, and their combination with Union All, this paper highlights the critical role of advanced query optimizers in achieving efficient query execution. Moreover, the proposed SQL extensions, such as specifying join cardinalities and enabling aggregation pushdown across decimal rounding, further enhance the adaptability and performance of HTAP systems. These innovations underscore the necessity of continued collaboration between applications and database systems to meet the evolving needs of enterprise users.

Acknowledgments

The authors extend their heartfelt gratitude to their (former and current) colleagues, JinUk Bae, Stefan Bäuerle, Timm Falter, JeongAe Han, Arne Harren, Sangyong Hwang, Andreas Kemmler, Boyung Lee, Danbi Park, George Qiao, Florian Scheid, Uwe Schlarb, Sangil Song, Sungheun Wi, JooYoung Yoon, Yongsik Yoon, and Di Wu, whose invaluable expertise and contributions have been instrumental in this work.

We also acknowledge and deeply appreciate the efforts of many others whose names are not mentioned here. This work is the result of collective contributions from diverse teams over a decade-long period, making it impossible to recognize every individual contributor. Nonetheless, their collective input and dedication have been instrumental in advancing this project.

References

[1] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 466–475. https://doi.org/10.1109/ICDE.2007.367892

[2] Alessandro Banzer and Alexander Sambill. 2022. *Authorizations in SAP S/4HANA and SAP Fiori.* SAP Press.

[3] Devraj Bardhan, Axel Baumgartl, Nga-Sze Choi, Mark Dudgeon, Piotr Górecki, Asidhara Lahiri, Bert Meijerink, and Andrew Worsley-Tonks. 2016. *SAP S/4HANA: An Introduction.* SAP Press.

[4] Gautam Bhargava, Piyush Goel, and Balakrishna R. Iyer. 1995. Simplification of outer joins. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research, November 7-9, 1995, Toronto, Ontario, Canada*, Dennis Bockus, Karen Bennet, W. Morven Gentleman, J. Howard Johnson, and Evelyn Kidd (Eds.). IBM, 7. https://dl.acm.org/citation.cfm?id=781922

[5] Jürgen Butsmann, Thomas Fleckenstein, and Anirban Kundu. 2021. *SAP S/4HANA Embedded Analytics: The Comprehensive Guide* (2 ed.). SAP Press.

[6] Arbee L. P. Chen. 1990. Outer Join Optimization in Multidatabase Systems. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems, Trinity College, Dublin, Ireland, July 2-4, 1990*, Rakesh Agrawal and David A. Bell (Eds.). IEEE Computer Society / ACM, 211–218. https://doi.org/10.1109/DPDS.1990.113712

[7] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. 1999. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 687–698. http://www.vldb.org/conf/1999/P64.pdf

[8] Renzo Colle, Ralf Dentzer, and Jan Hrastnik. 2024. *Core Data Services for ABAP* (3 ed.). SAP Press.

[9] Jochen Doppelhammer, Thomas Höppler, Alfons Kemper, and Donald Kossmann. 1997. Database Performance in the Real World - TPC-D and SAP R/3 (Experience Paper). In *SIGMOD 1997*, Joan Peckham (Ed.). ACM Press, 123–134. https://doi.org/10.1145/253260.253280

[10] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[11] Ahmad Ghazal, Ramesh Bhashyam, and Alain Crolotte. 2003. Block Optimization in the Teradata RDBMS. In *Database and Expert Systems Applications, 14th International Conference, DEXA 2003, Prague, Czech Republic, September 1-5, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2736)*, Vladimír Marík, Werner Retschitzegger, and Olga Stepánková (Eds.). Springer, 782–791. https://doi.org/10.1007/978-3-540-45227-0_76

[12] Ahmad Ghazal, Alain Crolotte, and Ramesh Bhashyam. 2004. Outer Join Elimination in the Teradata RDBMS. In *Database and Expert Systems Applications, 15th International Conference, DEXA 2004 Zaragoza, Spain, August 30-September 3, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3180)*, Fernando Galindo, Makoto Takizawa, and Roland Traunmüller (Eds.). Springer, 730–740. https://doi.org/10.1007/978-3-540-30075-5_70

[13] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengießer, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads. *Proc. VLDB Endow.* 8, 12 (2015), 1716–1727. https://doi.org/10.14778/2824032.2824069

[14] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

[15] Boris Gruschko, Kihong Kim, Hyunjun Kim, Taehyung Lee, Michael Mueller, and Daniel Ritter. 2025. Elastic Compute in SAP HANA Cloud by Example of SAP Integrated Business Planning. *Datenbank-Spektrum* to appear (2025).

[16] Julian Hyde and John Fremlin. 2024. Measures in SQL. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 31–40. https://doi.org/10.1145/3626246.3653374

[17] International Organization for Standardization. 2016. Information technology — Open Data Protocol (OData) v4.0 — Part 1: Core. https://www.iso.org/standard/69208.html

[18] Ralph Kimball. 1996. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses.* John Wiley.

[19] Ralph Kimball and Joe Caserta. 2004. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data.* Wiley.

[20] Lukas Landgraf, Florian Wolf, and Wolfgang Lehner. 2025. Experimental Evaluation of Optimizing Memory Consumption in SAP HANA using PEOopt. In *Companion of the 2025 International Conference on Management of Data, SIGMOD-/PODS 2025, Berlin, Germany, June 22-27, 2025.* ACM, to appear.

[21] Byung Suk Lee and Gio Wiederhold. 1994. Outer Joins and Filters for Instantiating Objects from Relational Databases Through Views. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 108–119. https://doi.org/10.1109/69.273031

[22] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. 2010. Speeding Up Queries in Column Stores - A Case for Compression. In *DaWak.* 117–129.

[23] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *BTW2017.* 545–563. https://dl.gi.de/handle/20.500.12116/656

[24] Norman May, Alexander Böhm, Meinolf Block, and Wolfgang Lehner. 2015. Managed Query Processing within the SAP HANA Database Platform. *Datenbank-Spektrum* 15, 2 (2015), 141–152.

[25] Norman May, Alexander Böhm, Daniel Ritter, Frank Renkes, Mihnea Andrei, and Wolfgang Lehner. 2025. SAP HANA Cloud: Data Management for Modern Enterprise Applications. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025.* ACM, to appear.

[26] Nikolaus Ott and Klaus K. Horländer. 1985. Removing redundant join operations in queries involving views. *Inf. Syst.* 10, 3 (1985), 279–288. https://doi.org/10.1016/0306-4379(85)90021-3

[27] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/-Analytical Processing: A Survey. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1771–1775. https://doi.org/10.1145/3035918.3054784

[28] Amol Palekar, Bharat Patel, and Shreekant Shiralkar. 2015. *BW 7.4—Practical Guide* (2 ed.). SAP Press.

[29] Abani Pattanayak. 2017. SAP S/4HANA Embedded Analytics: An Overview. *Journal of Computer and Communications* 05 (01 2017), 1–7. https://doi.org/10.4236/jcc.2017.59001

[30] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Gartner Resarch: Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation.

[31] Hasso Plattner. 2009. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 1–2. https://doi.org/10.1145/1559845.1559846

[32] Hasso Plattner. 2014. The Impact of Columnar In-Memory Databases on Enterprise Systems. *Proc. VLDB Endow.* 7, 13 (2014), 1722–1729. https://doi.org/10.14778/2733004.2733074

[33] SAP SE. 2024. ABAP RESTful Application Programming Model - Draft. Available at: https://help.sap.com/docs/abap-cloud/abap-rap/draft.

[34] SAP SE. 2024. ABAP RESTful Application Programming Model - RAP Extensibility-Enablement. Available at: https://help.sap.com/docs/abap-cloud/abap-rap/rap-extensibility-enablement.

[35] SAP SE. 2024. SAP HANA Cloud SQL Reference. Available at: https://help.sap.com/docs/hana-cloud-database/sap-hana-cloud-sap-hana-database-sql-reference-guide/sql-reference.

[36] SAP SE. 2024. SAP S/4HANA Cloud Public Edition - On Stack Extensibility. Available at: https://api.sap.com/products/SAPS4HANACloud/onstackextensibility.

[37] Thomas Saueressig, Jan Gilg, Uwe Grigoleit, Arpan Shah, Almer Podbicanin, and Marcus Homann. 2022. *SAP S/4HANA Cloud: An Introduction* (2 ed.). SAP Press.

[38] Thomas Saueressig, Tobias Stein, Jochen Boeder, and Wolfram Kleis. 2023. *SAP S/4HANA Architecture* (2 ed.). SAP Press.

[39] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, et al. 2019. Native store extension for SAP HANA. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2047–2058.

[40] Jeff Shute, Shannon Bales, Matthew Brown, Jean-Daniel Browne, Brandon Dolphin, Romit Kudtarkar, Andrey Litvinov, Jingchi Ma, John D. Morcos, Michael Shen, David Wilhite, Xi Wu, and Lulan Yu. 2024. SQL has problems. We can fix them: Pipe syntax in SQL. *Proc. VLDB Endow.* 17, 12 (2024), 4051–4063. https://www.vldb.org/pvldb/vol17/p4051-shute.pdf

[41] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition.* McGraw-Hill Book Company. https://www.db-book.com/

[42] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. *VLDB J.* 33, 5 (2024), 1485–1515. https://doi.org/10.1007/S00778-024-00858-9

[43] Toby J. Teorey. 1998. *Database Modeling & Design, Third Edition.* Morgan Kaufmann.

[44] Transaction Processing Performance Council. 2010. TPC-C Benchmark. Version 5.11, Available at: http://www.tpc.org/tpcc/.

[45] Transaction Processing Performance Council. 2022. TPC-H Benchmark. Version 3.0.1, Available at: http://www.tpc.org/tpch/.

[46] Transaction Processing Performance Council. 2024. TPC-DS Benchmark. Version 4.0.0, Available at: http://www.tpc.org/tpcds/.

[47] Mark Whitehorn, Robert Zare, and Mosha Pasumansky. 2006. *Fast track to MDX (2. ed.).* Springer. https://doi.org/10.1007/1-84628-182-2

[48] Sungheun Wi, Wook-Shin Han, Chu-Ho Chang, and Kihong Kim. 2020. Towards Multi-way Join Aware Optimizer in SAP HANA. *Proc. VLDB Endow.* 13, 12 (2020), 3019–3031. https://doi.org/10.14778/3415478.3415531

[49] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. 2013. Vectorizing Database Column Scans with Complex Predicates. In *ADMS 2013*. 1–12. http://www.adms-conf.org/2013/muller_adms13.pdf

[50] Noel Yuhanna, Mike Gualtieri, Gene Leganza, and Robert Perdoni. 2019. The Forrester Wave™: Translytical Data Platforms, Q4 2019.

[51] Bernhard Zeller and Alfons Kemper. 2002. Experience Report: Exploiting Advanced Database Optimization Features for Large-Scale SAP R/3 Installations. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 894–905. https://doi.org/10.1016/B978-155860869-6/50088-3

[52] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. HTAP Databases: A Survey. *IEEE Trans. Knowl. Data Eng.* 36, 11 (2024), 6410–6429. https://doi.org/10.1109/TKDE.2024.3389693