The HANA Native Query Engine for Lakehouse Systems

Daniel Ritter, Mihnea Andrei, Sukhyeun Cho, Maik Görgens, Taehyung Lee, Norman May, Amit Pathak, Paul R. Willems
SAP
{firstname.lastname}@sap.com

ABSTRACT

Modern enterprise applications and data warehouse systems move data into data lakes for economical and scalability reasons. Data is then stored in popular columnar file formats like Parquet which are optimized for writing using open table formats like Iceberg or Delta. This presents new challenges for existing database systems and their execution engines because excellent performance and scalability when accessing this data in complex analytical queries is expected while data is located in a remote data lake.

In this work, we present how we adapted the HANA Cloud Database Engine for efficient processing of files in data lakes, which we call *SQL-on-Files* (SoF). We motivate this evolution by its relevance for Business Data Cloud, SAP's Lakehouse, we discuss the viability of general architecture choices like *pushdown* and *direct access* architectures, and give insights into our SoF design decisions towards scalable, analytical query processing around execution engine, optimizer and caching. Our evaluation of SoF shows benefits of direct access over pushdown architectures for a new warehouse benchmark with complex, analytical workloads.

KEYWORDS

Cloud Data Platform, Database System, Data Lake, Lakehouse

PVLDB Reference Format:
Daniel Ritter, Mihnea Andrei, Sukhyeun Cho, Maik Görgens, Taehyung Lee, Norman May, Amit Pathak, Paul R. Willems. The HANA Native Query Engine for Lakehouse Systems. PVLDB, 18(12): 4831 - 4845, 2025.
doi:10.14778/3750601.3750608

1 INTRODUCTION

Lakehouses for Analytical Workloads The past few years have seen the rise of a new type of data management system, called lakehouse (LH), which combines the benefits of low-cost, open-format data lakes / object storage (i. e., cloud-based storage system for large amounts of any data) and transactional data warehouses [14, 33, 38, 71]. With the data gravity leading towards data lakes for economic reasons, LHs for analytical query processing became crucial for modern enterprise applications and warehousing use cases (e. g., [4, 5, 25, 27, 28, 35, 38, 54, 55, 71]). While the separation of compute and storage of data in data lakes opens up a new design space, there are several challenges for cloud data systems, as also recently acknowledged in the database community [1].

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750608

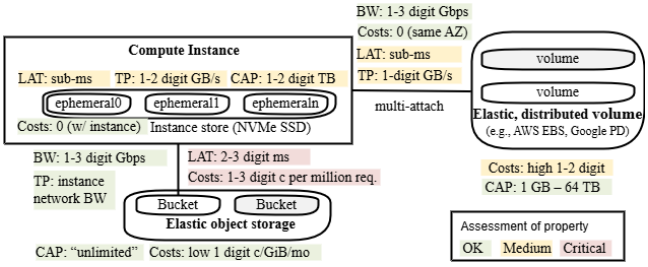


Figure 1: "Physics" of current cloud infrastructure in terms of cost, latency (LAT), throughput (TP) and capacity (CAP).

Challenges for Analytical Workloads The data is stored in open columnar storage formats, e. g., Parquet [45] or ORC [18], often using clustering schemes like HIVE partitioning or Z-ordering. These clustering schemes and associated statistics can be leveraged for pruning in queries. However, in the context of complex analytical workloads, the execution engine often needs to process large portions or even the entirety of the table data. As pointed out in [7], reading data from data lakes can become the primary bottleneck.

On the one hand, storage in cloud data lakes is cheaper than alternatives like multi-attach persistent disks or elastic block storage (see Fig. 1), offering high throughput (TP) and high availability (not shown). On the other hand, the read and write latencies (LAT) are within 2–3 digits milliseconds and incur variable costs per access. That makes the overall costs of queries in LHs less predictable than in classical data systems.

For the required database table semantics and transactional properties, LHs use open table formats (OTFs) such as Delta Lake [48], Apache Iceberg [17], or Apache Hudi [16] to define separate metadata layers on the data lake for each table [3, 71]. Those formats implement transactions for single tables and operations, indexes, and time travel [3, 71], but also lack support for critical features like recovery, multi-statement transactions and isolation for multi-table queries with separate transaction domains (e. g., [3, 22, 30]).

Finally, complex, analytical queries of modern enterprise applications with concurrent workloads of 90% reads and 10% writes at SAP [34] (up to 60% / 40% from independent LH applications and sources [54]) require support of operations beyond standard SQL, making in-situ processing (e. g., no indexes) on data in data lakes challenging for current systems.

State-of-the-Art To address these challenges, current cloud data systems feature elastic compute (e. g., Apache Spark [15, 70] with Photon for pipeline processing [6] and adaptivity [67], Presto, Trino [19, 59]) for processing data stored in write-optimized, columnar formats. In addition, there are many dedicated big data processing systems on data lakes with Azure Datalake [46], Microsoft Fabric

[8], AWS Redshift (with Spectrum) [4], Umbra / CedarDB [43] with file access improvements [49], and data ingestion AWS Glue [53].

The economic cost vs. performance trade-off is usually addressed by data and metadata caches, which are even more relevant for non-co-located compute and data lake. For example, Databricks [6, 31], Google BigQuery [41], Snowflake [9, 13] and Redshift [4] use data and Parquet file or OTF metadata caches on compute node local SSDs or based on shared services like FoundationDB [73]. Umbra's Crystal [10] has a ring buffer-based metadata cache and a columnar format agnostic data cache. Shared result caches are used by Snowflake (across virtual warehouses), and shared read-through caches in Google Napa [2]. However, it is unclear which caches are meaningful for complex analytics.

We found that there are two fundamentally different architecture styles for analytical processing on data in data lakes: "push-down" (e. g., PushdownDB [68, 69]) vs. "direct access" (e. g., Databricks [6, 31]). However, there are no insights into these styles from a real-world perspective.

Multi-table, multi-query transactions remain a challenge (e. g., [22]) and are practically mitigated by limiting the number of concurrent writers (impacting data freshness), and data normalization (incl. nesting data) to avoid inconsistencies and guarantee isolation.

SAP HANA SQL-on-Files (SoF) SAP has recently introduced Business Data Cloud, its Lakehouse, which we briefly present below, and whose data processing backbone is HANA Cloud. However, it was *a priori* unclear how to adapt an in-memory database system like HANA Cloud [32, 39] for complex analytics that fully leverages read parallelism, mitigating the data access challenge of data lakes (e. g., [7, 11]). How we addressed that is the subject of this paper.

HANA Cloud is an in-memory, HTAP database system [20, 32, 39] in the cloud. In this work, we introduce the Lakehouse capabilities in HANA Cloud, referred to as SoF, for complex analytics that address these challenges with a focus on the direct access system architecture, elastic scaling via elastic compute nodes (ECNs) [24] and caching (cf. [12]).

In particular we productively implemented both architecture styles, deciding for direct access (i. e., SoF), and share our insights that we gained and resulting design decisions. We show that the pushdown architecture style inherently breaks regarding expressiveness, performance, costs and ironically data transfer due to the limited operator expressiveness.

However, optimizations like pushdown are still relevant and specified for new file processing capabilities as part of the existing smart data access layer in HANA close to the expressive HANA Execution Engine (HEX). The file processing with HEX is scaled across multiple ECNs and integrated with HANA workload classes for massively parallel processing (MPP) of data in data lakes.

The user-facing interface is based on existing (fabric) virtual tables [40], HANA's remote data access technology, which allows for application switching between replication and federation, depending on their use case. We consciously decided to enable few caches, i. e., enabling existing static view caches and existing, more sophisticated, transactional analytical caches, and added OTF metadata caching, and discuss our decisions.

Contributions To summarize, we make the following contributions with HANA for LHs that we think are interesting for the database research community as well as practitioners:

- We motivate the business impact and introduce a motivating LH scenario in Sect. 2 that we use for benchmarking and discuss predominent lakehouse architecture styles and motivate our choice in Sect. 3,
- we specify an enterprise-scale data architecture "one-size-fits-all" with HANA cloud for complex analytics on data in delta lakes in Sect. 4 and for our execution engine in Sect. 5,
- we show improvements how to scale the parallel processing in Sect. 6, we describe options for caching in Sect. 7,
- for a better understanding, we explain transaction processing across HANA and Delta tables by example of DDLs in Sect. 8 and leave a more general LH description (e. g., transformations, ingestion or DML) for future work, and
- we evaluate the different architecture styles and evaluate standard TPC-H workloads in Sect. 9,

before we conclude in Sect. 10.

2 MOTIVATION: BUSINESS IMPACT AND LAKEHOUSE SCENARIOS

In this section, we discuss the business impact of LHs on modern enterprise applications, and motivate our SoF solution. We introduce a corresponding scenario that we also use in the evaluation.

2.1 Business Impact

At SAP, HANA Cloud [39] plays both the role of main database for business applications, and the role of technical backbone for SAP Business Data Cloud (BDC). BDC is SAP's Lakehouse and data integration technology.

In its role of SAP's Lakehouse, all of SAP's business applications publish to BDC all of their business data as Data Products. Data Products are curated and governed datasets, which are published as a service under the zero-copy model: any byte of business data produced by SAP's business applications is extracted only once and published for general consumption as a producer-oriented *primary data product*. There is no further copy, unless a specific optimization is needed for a specific consumption pattern. In that case, data is transformed and a consumption-oriented *derived data product* is created and published. A typical example would be denormalizing and pre-aggregating to reduce the cost and enhance the performance of pre-defined analytical processing on top of the Lakehouse. Data products are currently implemented as delta tables and published using the delta sharing [47] protocol, required for governed, secure, open, and cross-platform data sharing.

At SAP, we distinguish process integration and data integration. The former is based on application APIs, gives access to data with low latency and high freshness, and is record or object oriented. It does not scale well for set-oriented access. Conversely, data integration focuses on set-oriented access at scale. BDC's role as the SAP data integration technology is likewise based on data products and a zero-copy consumption model. Data published by an application as delta tables is shared at scale across all consumer applications and services, without wasting resources for each producer-consumer pair, like point-to-point integration would do.

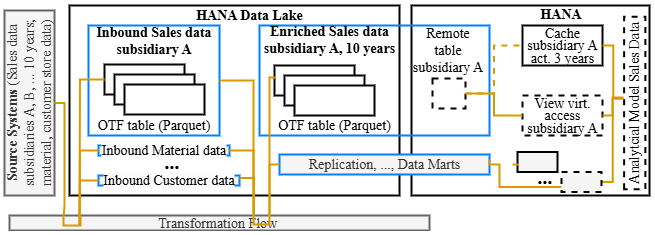


Figure 2: Motivating Retail Lakehouse Scenario

Although it is out of the scope of this paper to cover SAP BDC in depth, we introduced it as the main motivation for HANA Cloud evolving to native data lake, SQL-on-Files (SoF), and Spark integration. The Spark component of HANA Cloud is BDC's runtime for data ingestion and transformation. To become the data processing backbone of BDC, HANA Cloud had thus to evolve and become more than a HTAP multi-model database engine [39].

At the same time, this evolution serves the role of HANA Cloud as the application's main database. An SAP application accesses in a seamless way both its private data and data lake data, under the same programming model. The application uses SQL `SELECT` for all data reads, including when accessing the delta table of a data product published by another application. Delta table data is either accessed directly in the data lake, through SQL-on-Files federation, or locally in a database table where the data is replicated. When replicating data, HANA Cloud leverages the delta table's native change data capture (CDC) mechanism. Likewise, an application uses SQL `INSERT`, `UPDATE`, and `DELETE` to write to local tables the datasets which it publishes as data products, using database ACID transactions. HANA Cloud replicates out on behalf of the application those datasets to the delta tables of the data products and thus decouples its own transaction domain of the data lake.

The "*one-size-fits-all*" role of HANA Cloud at SAP, already described in [39] for business applications, is yet again notable here for BDC: its data processing needs are covered by the database engine, Spark, and data lake. We do not debate the general relevance of Stonebraker et al. [63]; our architecture choice might not hold for other software editors. It nevertheless holds for SAP, predominantly a business application editor. The investment in HANA Cloud, as a single data processing service covering all SAP needs, is a better solution compared to spreading the investment to support and evolve multiple different specialized services. At the same time, reinventing established state of the art open source technology would be a poor decision, hence the approach described in this paper: the reuse of Apache Spark and delta table, integrated with the database engine and operated as part of the HANA Cloud data processing service "*menage a trois*". In this paper, Spark-based data transformations, replicating data in for local access and replicating data out for Data Products creation are out of scope. The focus is on SoF access to delta tables in the data lake, with the business motivation of accessing BDC Data Products, both for application data integration and analytical processing.

The business impact could be materialized through a relational compute engine that allows for efficient, complex analytics on data lakes. Subsequently, we discuss the two most common LH architecture styles for that.

Table 1: LH Scenario Stories / Queries.

Story / Query	filter	union-all	aggr.	join	sort	sql-h	select	fems
Q1	○	●	●	○	●	○	○	○
Q2	●	●	●	○	●	○	○	○
Q3	○	●	●	⊕	●	○	○	○
Q4	●	●	●	●	●	○	○	○
Q5	●	●	●	●	●	○	●	●
Q6	○	●	●	○	●	○	●	●
Q7	○	●	●	●	●	●	○	○
Q8	●	●	●	●	●	●	●	●
Q9	○	●	●	⊕	●	○	○	○
Q10	●	●	●	○	●	○	○	○
Q11	○	●	●	●	●	●	○	○
Q12	●	●	●	●	●	●	○	●
Q13	●	●	●	⊕	●	●	○	○

contained: ●, ⊕(left-outer join) not contained: ○

2.2 Motivating Lakehouse Scenario

The motivating LH scenario shown in Fig. 2 is inspired by our previous BW/4HANA experience, containing three use cases of (1) complex analytics on data lakes, (2) data marts, and (3) data lake to LH replication for real-time analytics. These use cases are subsumed today by SAP's Lakehouse, Business Data Cloud.

Storage In the LH use cases, data arrives in a data lake from various source systems from different business domains like warehousing, smart meters, or retail. The data comes in a multitude of formats that are transformed into OTF tables like Delta or Iceberg stored in columnar Parquet format, which grow into multiple TB of compressed data, making this a good fit for data lakes. The columnar format allows for storage closer to the relational engines and OTFs provide basic transaction guarantees.

At SAP, the business applications are the main producers of business data. Domain-driven design and service decomposition, architecture patterns of modern cloud applications, require the Business Data Cloud LH for data to be seamlessly accessed, independently of the service producing it.

Compute The transformations require versatile user-defined functions (UDFs) that are provided by scalable data processing systems like Apache Spark [15]. While real-time analytics cannot be guaranteed on current, non-proprietary storage formats in data lakes (e. g., [72]) and thus requires replication into the relational data system, the cases (1) requiring execution of SQL queries and (2) requiring data transformations using SQL (e. g., to avoid operational costs of multiple data systems) can be done directly on data in data lakes.

In this work we focus on relational queries directly on data lakes (case (1)) that are adapted from a use case out of the retail domain. The queries require complex analytical capabilities like hierarchies, currency conversions that are available in relational engines like HANA Cloud and are not supported by other systems like SparkSQL. For example, query 12 (Q12) of our motivating scenario is shown in Fig. 3. While Q12 is a relatively simple query with standard SQL (e. g., aggregation, join, sort), it contains two types of hierarchies at leaf nodes served with filtering joins (i. e., a time and a sales org. dimension) and restricted and calculated key figures. Table 1 gives an overview of the capabilities of all 13 queries with `sql-h` denoting hierarchies and `fems` being "Form EleMent Selection" (short FEMS) local filters in LH InfoCubes. FEMS filters target selection groups in the SAP Business Warehouse. When these filters are pushed down to the HANA database, they are

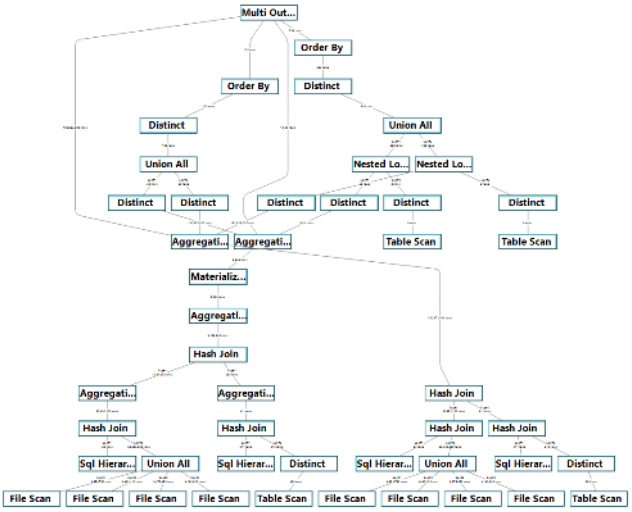


Figure 3: Motivating Lakehouse scenario, plan of query Q12 with hierarchies at leaf nodes.

converted into standard SQL queries that include sum aggregation, CASE expressions, and bitwise operations.

Similar to related work on LHs (e. g., [54, 64, 66]), we observe highly concurrent read-write workloads with writes coming from ETL, transformations or data ingestion that are not part of query engines, and reads like complex, multi-dimensional analytical queries, on which we focus in this work.

3 LAKEHOUSE ARCHITECTURE STYLES

In this section, we discuss the predominant Lakehouse (LH) architecture styles that we found in current research and industrial implementations that attempt to support the scenario presented in Sect. 2. While both styles show huge potential, we discuss our rationale on the selection of the direct access style.

3.1 Pushdown Architecture

In disaggregated architectures, the network connects the computation and storage layers. While cloud networks are no bottleneck anymore (e. g., see [11]), the inherent challenges based on physics of current cloud data systems (see Fig. 1) need to be addressed. Hence, two intuitive solutions are widely applied in such systems: caching [23] and computation pushdown closer to the storage [26].

Sub-Query Plan Pushdown For the latter, systems like Oracle Exadata server [29] and IBM Netezza machine [61] showed significant performance improvements. More recently, PushdownDB studied the question on how to divide query processing by pushing down limited computation to Amazon S3 Select [56], close to data lakes as shown in Fig. 4, and gained good results through pushdown [69]. Driven by the assumption that cloud networks are the bottleneck, the architecture evolved toward a dedicated, custom elastic compute in [68], shown in Fig. 4, which caters for cloud vendors without in-place query capabilities. In all cases, sub-query plans are passed from the relational DBMS with elastic compute to the dedicated compute. The specific pushdown architecture parts are shown in purple. With the reduced capabilities, the dedicated

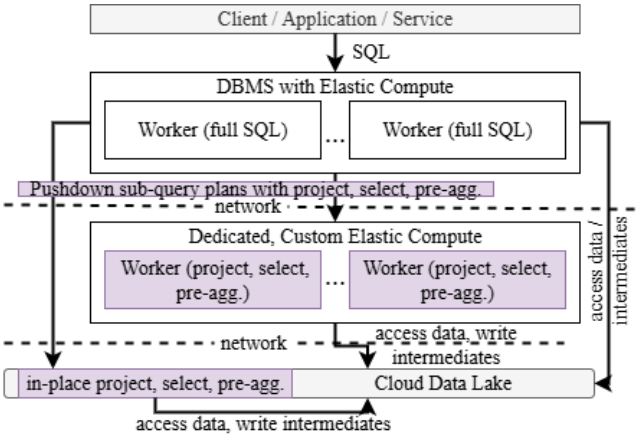


Figure 4: Pushdown architectures (similar to [56, 68, 69])

compute with externalized file access can be placed into smaller, more cost efficient nodes with elastic scaling.

HANA Data Lake Files Select In 2022, we explored a pushdown architecture, short Data Lake Select (DL-Select), which externalized the Parquet file-format handling and utilized specified elastic compute on top of data lakes. Similar to S3 Select, DL-Select started out with simple relational operators like projection, selection and pre-aggregation, which are easy to scale. Later support for Bloom filters, more complex expressions and other optimizations were added. Sub-query plans were passed from the HANA optimizer into DL-Select and executed in a dedicated compute engine different from HEX. To gain robustness through persistent failover states, intermediate results were shared via data lake which served as an infinite size, persistent buffer between DL-Select and HEX. The results were picked up by HEX via returned URLs. Alternatives like direct streaming from DL-Select into HEX were considered. However, failover and back-pressure handling would complicate the architecture significantly. We did not pursue those approaches further and focused on a direct access architecture instead.

3.2 Direct Access Architectures

While the pushdown architecture requires two "data hops" from data lake to the dedicated compute and then to the DBMS, systems like Databricks access the data directly through compute like Apache Spark as shown in Fig. 5 or specialized compute (e. g., [6]), marked in red.

Data Lake Adapted DBMS Consequently, such a direct access to data lakes requires engines of existing DBMS to add capabilities of reading from storage formats like Parquet and directly integrating with OTF semantics (i. e., metadata, handling table versions). That requires a file access layer which applies read optimizations like file skipping, data skipping and passes available metadata like zone maps or min-max statistics to the DBMS optimizer. This naturally includes pushdowns into the file access, which are similar to the ones discussed for the pushdown architecture. There are also multiple implicit "data hops" necessary (i. e., data lake → file access → DBMS engine). The distributed query processing of horizontally scaling engines likewise introduces such extra hops. However, in contrast, the file access can be integrated tightly with the engine

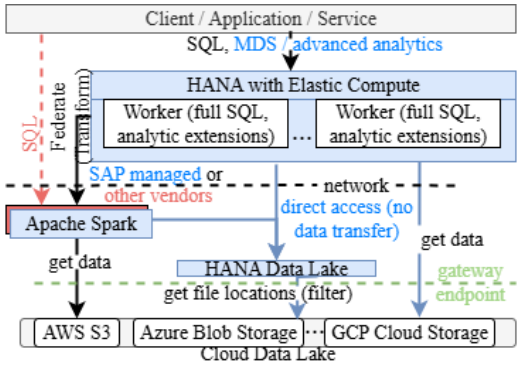


Figure 5: Direct Access Architecture (red parts like [71])

of the DBMS and incurs fewer format conversions (e. g., Parquet column chunks to DBMS engine format). In such an architecture elasticity is inherited from the DBMS.

3.3 Discussion of Performance and Costs

Having efficient SQL compute close to data lakes seems desirable to implement a pushdown architecture. However, pushdown architecture components like S3 Select and AWS Aqua [58] were either discontinued or are not exposed to customers of cloud vendors.

DL-Select evolution to HANA SoF We share our rationale for evolving DL-Select with pushdown architecture to SoF based on the direct access option. Our decision was mainly based on the expressiveness of required complex SQL and multidimensional (MDS) queries. A dedicated pushdown compute would need the same capabilities as the database engine to be effective. However, pushing down more complex operations into the dedicated compute would have ultimately meant having the same engine in two different layers, effectively mimicking the scalable technology of ECNs. The additional operational costs of a dedicated, elastic compute, the increased variable costs of a pushdown architecture with its higher number of data lake accesses, and the multiple data hops involving data format transformations were the main reasons to adopt the direct-access architecture in HANA Cloud. Avoiding such complexity and also the associated development and operational costs is a concrete illustration of *"one size fits all"* at SAP.

A Simple Analytic Cost Model To support this decision we devised a simple analytic model to estimate the cost, using prices for AWS services, for some query considering the main cost components of the two architecture alternatives. Figure 6 shows a cost calculation for our direct and pushdown solutions for TPC-H query Q1 (scan heavy), assuming a single ECN instance for the direct option and a downsized ECN instance for pushdown to DL-Select. The CPU utilization is assumed up to be 100% (equivalent to 44.2 query executions per time) using all available CPU resources. The direct option (i. e., SoF) has fixed costs per time per instance and variable read-only costs to fetch data from the Amazon S3 data lake. The DL-Select option has lower fixed costs per time because it uses less powerful hardware and lower variable costs for read access because less data needs to be fetched in the pushdown architecture. But additional write costs for intermediate results and elastic compute compute costs apply. The results show a break even for low CPU utilization making the direct access architecture the preferred option in most cases.

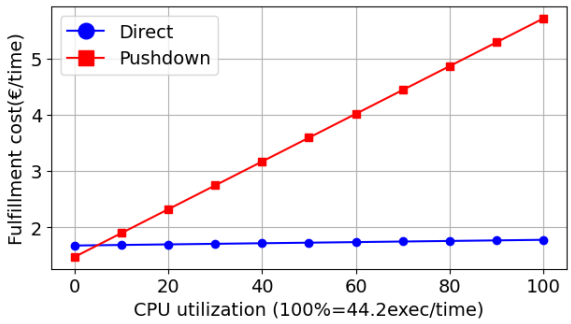


Figure 6: Cost trade-off: direct access (SoF) vs. pushdown (DL-Select)

4 HANA CLOUD ARCHITECTURE FOR LHS

In this section, we introduce the high-level architecture style for realizing query processing on data lakes in HANA Cloud.

4.1 Direct Access Architecture Style for HANA

The high-level direct access architecture for HANA Cloud is shown in Fig. 5 with the specific extensions marked in blue components. The file access component is integrated with HEX (i. e., file scan next to table scan), which accesses data lakes directly during processing of complex SQL or multidimensional (MDS) queries. The elasticity is achieved on two levels. The actual data from the data lake is read massively parallel through multiple file access threads and the compute runs on multiple elastic compute nodes (short ECNs).

To guarantee a multi-cloud, enterprise-scale service, HANA uses an abstraction called *HANA Data Lake Files* (HDLF), which deals with cloud vendor specifics, locates the data needed for a query, and performance effective file and data skipping. As a result, HDLF passes direct access URLs to the file access layer in HANA that loads the data. For secure network isolation, HANA Cloud uses virtual private clouds (VPCs), which specify so called gateway endpoints [57] for a more secure (i. e., no public NAT) and cost efficient data lake access (i. e., no network bandwidth costs) with lower latencies.

4.2 HANA Architecture and Design Decisions

Based on the direct access architecture style and the high-level architecture, we carefully adapted HANA's core architecture as shown in Fig. 7 and made the following design decisions.

Integrate File Access Layer The highly optimized file processing of DL-Select (e. g., with latency hiding through thread scheduling) is a natural fit for the file access layer in HANA, moving the "second hop" to the data into HANA. Figure 7 shows its main components including plan execution and statistics estimation as well as format handling for storage formats like Parquet and CSV, and OTF accessors for formats like Delta and services like Delta Sharing (e. g., for leveraging OTF metadata when handling filtering predicates to limit the set of accessed files). The latter will be adapted for upcoming OTF formats like Apache Iceberg. The main aspect of the integration concerns an efficient interplay with HEX.

Extend HANA Federation for Files using Virtual Tables A natural choice for the SQL interface are HANA's (fabric) virtual tables [51], which expose tabular data of remote source systems as a table in HANA and are used for federating queries to remote source systems. Instead of making OTF tables a first-class citizen in

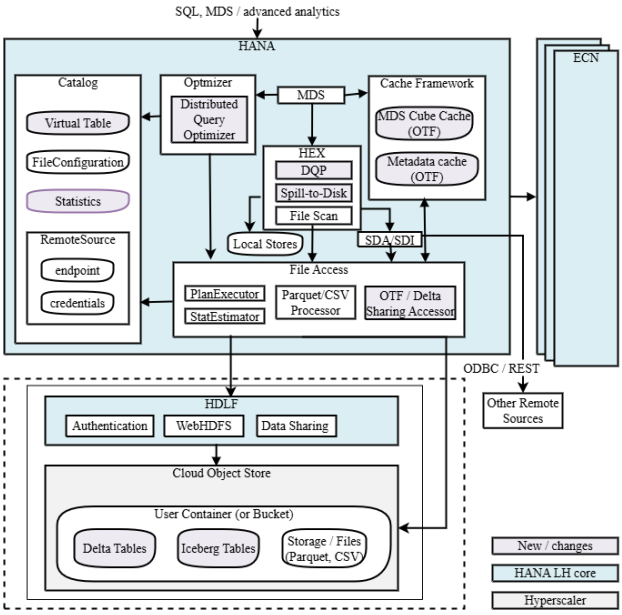


Figure 7: Adapting HANA Core for LHs.

HANA, the federation approach allows for a loose coupling to OTFs with a focus on read-cases. Fabric virtual tables provide flexible toggling between federation and replication on a per-table basis, allowing applications to navigate the cost vs. performance trade-off of queries on data lakes.

Adapt DQP for Files For scalability, we employ HANA's distributed query processing (DQP) across multiple ECNs. Consequently, optimizer and execution engine require extensions to work well for data in data lakes.

Elasticity with ECNs Elasticity is an important optimization that allows for efficient query processing over large data, while allowing to process data in smaller in-memory HANA instances. HANA ECNs naturally provide massively parallel processing (MPP) for efficient processing based on DQP. The HANA optimizer distributes the query processing over multiple ECNs, each with a local file access, HEX query processing, and caching, as shown in Fig. 7.

Enable Specific Caches for Delta Changes Another optimization is data and metadata caching, which helps dealing with the cost vs. performance trade-off when working with data lake, especially when compute and storage are not co-located in the same data center or region. While HANA's static view caches [52] immediately work by our design, we identified OTF metadata for reducing the number of non-data calls and the transactional MDS Cube Cache for efficient multidimensional query processing as shown in Fig. 7 as most beneficial caches in the LH architecture.

OTF Table Lifecycle—Combine Transaction Domains The creation and update of virtual tables via DDL crosses the transaction domains of HANA and OTFs, which have different ACID guarantees. We deal with this issue by applying operations inspired by the SAGA pattern [21, 62], which was originally designed for long-running transactions. Instead of implementing these capabilities natively in HANA, we opted to utilize existing open-source engines. This allows us to capitalize on the maturity and extensive functionality of these engines, while minimizing development effort and ensuring compatibility with industry standards.

Subsequently, we give more details of the above design decisions for file access, HEX and optimizer and the two optimizations for elasticity and caching, as well as OTF lifecycle.

5 HANA ENGINE MEETS DATA LAKE

In this section we give a background on the execution engine and query optimizer in SAP HANA Cloud and share some experiences on integrating data lake access. The resulting architecture is depicted in Fig. 7.

5.1 Base Engine

Execution Engine About 2016 we started elaborations to completely re-engineer query execution in SAP HANA [20]. At that point we had multiple engines specializing in different areas, e. g., one optimized for star-schema queries (but only those), another for generic joins (but only of column store tables), and a separately tuned access path for small OLTP selects. Those excelled at their respective tasks, but the tight design space made them hard to extend to anything beyond pure in-memory processing over the HANA column store. Thus major design goals for the new *HANA Execution Engine (HEX)* were future extensibility and flexibility to support any specific optimization without mandating its applicability.

HANA being an HTAP system, many of the decisions went along a similar trajectory as the HyPer system [37, 42, 50]. HEX does push-based pipelining and JIT compilation using LLVM. A pipeline consists of multiple executable operators, each of which may or may not use JIT code generation (cg) or pre-compiled C++. Subsequent relational operators that all generate code, like a join probe followed by an aggregation, can be fused into one executable operator. At runtime, cg-operators can be interpreted as long as the compilation is not finished. To amortize for compilation costs (which are significant), we use heuristics to determine if and when actual JIT code compilation for an operator is triggered. Data is pushed from one operator to the next in small *chunks* of tuples that fit into cache. We use morsel-based parallelization with dynamic sampling to determine task sizes and degree of parallelism. Because a pipeline can have multiple operators, we are free to *re-parallelize* within a pipeline, e. g., after a potentially expanding join. Distribution is based on distributed exchange. In anticipation of Sect. 6 let us point out that from the perspective of the engine, an ECN is just another compute node used for distributed query processing.

Just as was observed in [43], we realized a while back that pure in-memory systems are uneconomical, especially in the cloud. To that end, HANA had grown beyond a pure in-memory system by incorporating *Native Storage Extension (NSE)* tables [60]. As natural counterpart for that and to lower TCO in the cloud further, HEX provides disk-spilling variants of all pipeline breaking operators.

Query Optimizer The HANA query optimizer was not rewritten, but rather evolved and grew alongside the development of HEX further into a full-fledged general purpose relational optimizer. In a nutshell it is Volcano style, transform-based and cost-based. Specialized rule-based optimizations represent accumulated learnings

to deal with the sophisticated HTAP workloads and deeply nested views in the SAP ecosystem [20, 32].

In federated query processing, the optimizer has been enhanced to consider the capabilities offered by remote sources. It makes informed operator pushdown decisions based on these capabilities and employs a cost model that aims to minimize network traffic between federated systems. By pushing down operations to the remote sources whenever feasible and beneficial, the optimizer can significantly reduce data transfer volumes, leading to more efficient query execution. We refer to Sect. 6 for details on plan enumeration considering DQP and partition-wise optimization alternatives.

5.2 Integrating Object-Store Access

The new engine was solidly established (although surely not "finished") when the need for object-store access arose. It was never a reasonable option to start from scratch and develop yet another engine that is designed specifically for the lakehouse use case. Besides being too costly (from an engineering perspective), such a narrow focus does not fit to the heterogeneous nature of applications using HANA. In our opinion, just the one example in Fig. 3 clearly shows why. We need full composability of object-store access with all other sources of data. A dedicated engine for data lake data would either need to have complete semantic coverage, or mandate cutting the plan into pieces, effectively federating between engines. Having the full plan in integrated form provides the most optimization potential for plan generation and execution, and the easiest way to ensure consistent semantics.

Therefore, our baseline is somewhat different from other engines that are tailored for the lakehouse, such as Presto [59], Photon [6] or Apache DataFusion [36]. We stay close in spirit to HyPer and Umbra, which got also extended to data lake access [11, 43].

There are two main requirements coming with the lakehouse use case. One is how to get the data into the engine, which will be the focus of the next section. The other is that data volumes expected to be handled by a query become a lot bigger, in particular relative to the available main memory of an instance. To that end, some of the design decisions underlying the engine came to be very beneficial. Push-based pipelining with adaptive parallelization is a good basis to handle unexpectedly large data, and we had disk-spilling operators as a fallback. Clearly, efficient and scalable distributed processing is mandatory; see Sect. 6 for more details. Fortunately, distributed exchange combined with semi-join reduction goes a long way. One can build the necessary broadcast- and shuffle-joins by merely adding repartitioning steps before the exchanges.

Our point here is that a solid general purpose engine can be extended in a pragmatic way to efficiently handle essentially a new data source (or table type). Techniques like repartitioning or semi-join reduction do become particularly necessary, but they will benefit also queries that do not involve tables on data lakes.

A strong case is made in [67] that the need for *adaptive* execution is exacerbated for lakehouses, due to, basically, more uncurated data with less (reliable) statistics. We fully concur with that assessment. Currently our execution plans still incorporate some hard estimation-based decisions made by the query optimizer. Those cannot be changed at runtime, except by falling back and recompiling the plan. For instance, while we can adjust the order of filters on column store tables at runtime, we cannot switch join types yet. Improving the interplay of optimizer and execution engine to allow for more adaptivity is an ongoing effort.

5.3 File Access and the FileScan Operator

As teased in Sect. 4 and shown in Fig. 5, the decision for a direct access architecture naturally led to a new component *File Access* to handle accessing files (via HDLF or direct) and scanning them efficiently. File Access remains a separate sub-component, not subsumed by HEX, because it is also used for data import and replication. Furthermore as we will elaborate below, needs for parallelization are a bit different.

Initially, we employed the existing federation capabilities of optimizer and execution engine to connect to the File Access layer. The interface takes generic SQL and returns a generic internal table structure. This turned out, unsurprisingly, to be too inflexible and inefficient: With SQL as input, the query optimizer is limited in optimizations that can be pushed down, and the results needed first to be converted to HEX chunks. Thus we implemented a dedicated *FileScan* operator that acts as the bridge between HEX and File Access. In Sect. 6.3 we outline the distributed semi-join reduction for FileScans—an optimization that would not have been easily possible using the generic federation capabilities.

Similar to the pushdown architecture, the optimizer assigns specific operations, e. g., projection, selection, or pre-aggregation, to the FileScan operator. Based on file clustering schemes and statistics, entire files or portions of them are pruned. Simple aggregations like `MIN`, `MAX`, and `COUNT` can often be computed directly from the available file statistics without reading the actual data.

The File Access layer processes data in a column-wise manner, which aligns naturally with columnar file formats. After applying projection, selection, or pre-aggregation, the resulting data is transformed into the row-wise format required by HEX for further processing. This conversion from a columnar to a row-oriented format has proven to be a major contributor to overall query runtime. However, we partially mitigate this overhead by leveraging dictionaries stored in the files—particularly for string columns. Instead of storing the actual strings in the row format, we store ValueIds—markers that reference the corresponding string in the dictionary.

The File Access layer enables parallel processing across multiple files and row-groups within files. Retrieving the data from data lake is inherently network I/O-intensive. Therefore, while one chunk of data is being processed, subsequent data is prefetched from the data lake to improve throughput and reduce latency. As demonstrated in [11], hundreds of outstanding requests are required to fully utilize high-bandwidth networks. HANA does not yet employ asynchronous techniques such as io_uring or coroutines, as used in systems like Umbra [65]. Instead, it relies on a separate thread pool within the File Access layer. Threads in this pool primarily wait for network responses and consume minimal CPU resources. They are not counted against the thread usage limits imposed by HANA workload classes.

5.4 All in one Engine

We ended up with the situation as in Figure 7, where a FileScan operator (or rather family of operators) is integrated into the engine as full counterpart to the TableScan for database tables and RemoteScan for federation scenarios. From the perspective of the

engine, FileScan represents "just another data source". Similar to how, say, the implementation of a TableScan on a column table can exploit internal knowledge of the HANA column store, the FileScan is a direct and native connector to the FileAccess component dealing with OTF tables in the data lake.

6 ELASTIC SCALE-OUT VIA ECNS

In modern data processing, elastic scale-out architectures are instrumental for efficiently handling large-scale, heterogeneous data workloads. SAP HANA's Elastic Computation Node (ECN) [24] enables such scale-out by distributing query execution across multiple nodes, leveraging partition-aware optimization (PAO). This section describes new HANA optimizer strategies for generating distributed query plans that effectively utilize ECNs, ensuring optimal performance for queries of data lakes.

6.1 Partition-Aware Optimization

PAO was designed to optimize execution plans for partitioned column tables, enabling more granular data processing within individual HEX operators. In contrast to conventional single-stream approaches, PAO allows each operator to manage multiple partitioned data streams distributed across multiple compute nodes. During the plan enumeration phase, the optimizer determines the method and location for accessing partitioned data, aiming to minimize the execution costs of operators distributed across multiple compute nodes. Unlike traditional approaches where table partition locations are fixed, PAO with SoF tables has been extended to leverage the flexibility to assign FileScan operator locations across multiple ECNs. As a consequence, subsequent operators can be distributed as well, involving repartitioning operations to balance workload distribution.

6.2 Distributed Plan Generation

For SoF queries, the optimizer transforms enumerated plans into distributed plans through a post-enumeration process. Initially, predefined heuristic rules populate execution information, guiding partition strategy decisions. Key rules include creating pre-aggregation for group-by operations, repartitioning for joins and window operations, and generating broadcast joins. The optimizer then calculates partition execution details, specifying where and how partitioned data is executed. Joins employ broadcast or repartition strategies, selected based on estimated network transfer costs. In broadcast joins, smaller data sets are broadcasted to all ECNs, while repartition joins hash and distribute data using join keys, ensuring parallel execution at corresponding ECNs.

Aggregations benefit from pre-aggregation and repartitioning. Pre-aggregation is applied to group-by operations for aggregation functions with associativity, while repartitioning is considered for high cardinality grouping operations. Each ECN performs an initial and a final aggregation after merging results. Repartitioning involves hashing grouping columns and distributing data for parallel aggregation across ECNs.

Integrating SoF with traditional in-memory table operations requires effective handling of partition-unaware operators, such as non-partitioned tables or shared views. These operators can, for example, broadcast results for joins and repartition for group-bys. Partition-aware operators preserve output partitions until they have to be converged or repartitioned for a subsequent operation.

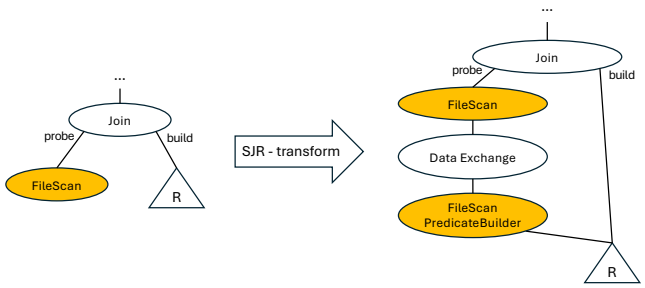


Figure 8: Distributed SJR for FileScan operations

6.3 Distributed Semi-Join Reduction

Semi-join reduction (SJR) is a well-established technique for reducing the workload on the probe side of a join by leveraging values already observed on the build side. In this context, we refer to the build side as the reducer.

In our initial federation-based approach, where every request to a remote service had to be expressed in generic SQL, all distinct values from the reducer side had to be transferred to the probe side in order to formulate an In-List clause or similar. In a distributed setup, this typically requires broadcasting a potentially large number of values across the network.

With the introduction of the custom FileScan operator, we adopted a more efficient approach to reduce the amount of data broadcasted. On the sender side, we construct a compact structure representing the semi-join filter. This structure may consist of min/max values from the reducer, a compressed list of distinct values, a bitvector, a Bloom filter, or a combination of these (e.g., min/max values combined with a Bloom filter). The creation of this structure is explicitly modeled in the algebra through the *FileScan Predicate Builder*, allowing it to seamlessly integrate into the distributed query-processing architecture: broadcasting the filter is handled via standard data exchange operators.

On the receiver side, as part of the FileScan operation, semi-join filters from multiple nodes are merged and applied to prune files and row-groups that are guaranteed not to contain matching rows for the join. The transformation rule is illustrated in Fig. 8, where a table stored in files is joined with an arbitrary derived table R. For simplicity, the figure omits additional data exchanges that may occur on either side of the join before or after the transformation.

It is worth noting that this filter-building approach before broadcasting is not limited to SQL-on-Files processing and can also be applied to regular TableScans. However, given the large volumes of data typically encountered in lakehouses, minimizing the amount of exchanged data becomes particularly crucial.

6.4 Decision on ECN Utilization

In order to leverage available ECN resources, workload classes [44] are employed to determine statements to be executed with distributed query processing without modifying application code. Since any node can fetch files from the data lake at the same cost for SoF processing, by leveraging ECNs and employing partition-aware optimization techniques, the query optimizer enables efficient and

scalable execution of SoF queries, enhancing the performance and scalability of data processing infrastructures.

Determining the optimal number of ECNs for query execution is under exploration. The goal is to develop a heuristic approach that dynamically allocates ECNs based on query workload and available resources, ensuring efficient use of computing resources in distributed systems.

7 CACHING

In this section we give insights into our analysis on data and metadata caching, eviction strategies and locality, before we describe a more complex, transactional cache for multidimensional queries.

7.1 The Relevance of Caches

There are multiple options and layers at which caching can be useful. Ideally, we want to have 1) a high cache hit ratio to effectively use the caches and 2) small caches because they consume precious memory resources. Therefore, we implemented an OTF format metadata cache and a Parquet column chunk data cache for which we study the performance impact.

We compare "no caching" as performance baseline and run the LH benchmark (see Sects. 2 and 9). Only caching the metadata of the OTF tables already reduces the query runtimes to ca. 32%. The root cause of this performance improvement is a drastic reduction in the number of network requests to the data lake. Consequently, the caching of metadata does not only reduce the query response times but also saves costs for requests to the data lake. We found that additionally caching the Parquet column chunks further reduces query response times to less than 10% of the baseline. However, to achieve this speedup much more memory resources are needed. Consequently, we focus on caching metadata only.

7.2 The Relevance of Eviction Strategies and Storage Locality

For data caches like the in-memory Parquet column chunk cache, we elaborated on the importance of different caching strategies for the LH scenario in Sect. 3. For that we executed the queries with an increasing cache size and evaluate the cache hit ratio for different cache replacement strategies. Not surprisingly, the cache hit ratio increases up to 60% starting with a cold cache as cache size increases to the size of the working set of the data. Also, we see no clearly dominating cache replacement strategy when comparing LRU, LFU, CLOCK and WATT-2; LRU has slightly worse cache hit ratios for smaller cache sizes. Therefore, it might be worth investing into larger and cheaper caches, e. g., instance local SSD compared to memory, rather than sophisticated cache eviction strategies, for the parquet column chunks.

In another experiment, we explore the effect of caches regarding the locality of compute and storage in the same data center compared to remote data center, which might be the case for some scenarios from Sect. 3. The results in Fig. 9 show a significantly higher fluctuation rate between all data in the compute (i. e., `unlimited` cache) compared to different caching strategies.

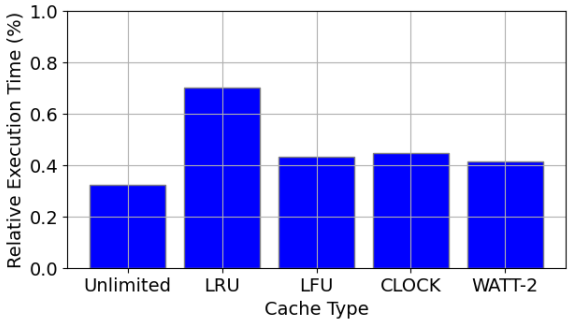


Figure 9: Relative performance with enabled column chunk cache with data in local vs. remote data center

7.3 MDS Cube Cache with Delta Changes

In addition to caches that work on the file I/O layer discussed above SAP HANA also supports caching query results for complex analytical queries. Specifically, multi-dimensional services (MDS) is a SAP HANA component that is responsible for processing analytical and multidimensional queries. MDS queries are commonly used in analytic applications of SAP, e.g. Business Data Cloud. MDS uses the MDS cube cache to cache the results of aggregation and calculation operations.

Internally MDS stores a data structure called a cube, which is used to produce the query results. By caching these query results, the repeated computation of the cube can be avoided which reduces the query response times and saves CPU resources. As MDS queries may access data stored in Delta tables or Iceberg tables, the caching of cubes also avoids reading data repeatedly from the data lake.

To make sure that consistent and fresh results are returned from the MDS cube cache, it maintains version information of the accessed OTF tables with the cached results. When the OTF tables in the data lake are updated and subsequently accessed, a new version of a cube cache is computed, cached and returned as result. When older versions of the OTF tables are accessed, also older cached versions of an MDS cube can be returned from the cache.

8 OTF TABLE LIFECYCLE

While the focus of this paper is on query processing (`SELECT`) over OTF tables, this section only briefly describes some of the challenges when dealing with DDL statements across different transaction domains for HANA and OTF tables.

8.1 Spark Integration

Apache Spark's rich support for OTFs makes it an ideal choice for executing DDL statements on OTF tables in data lakes. We extended the HANA Smart Data Access (SDA) to use Spark as shown in Fig. 10. The integration provides a SQL interface for HANA users to create OTF tables in data lake using HANA DDL. The statements are passed to remote Spark compute which applies the actual DDL changes to OTF tables. A HANA virtual table is made available as HANA object referring to the remote OTF table. The virtual table can be queried and is used for other lifecycle operations. Two examples of HANA DDL SQLs to create OTF tables are shown in Listing 1. They support the creation of OTF tables in two different modes (a) where the table is created purely for consumption and its lifecycle is managed by some other application / DB engine and

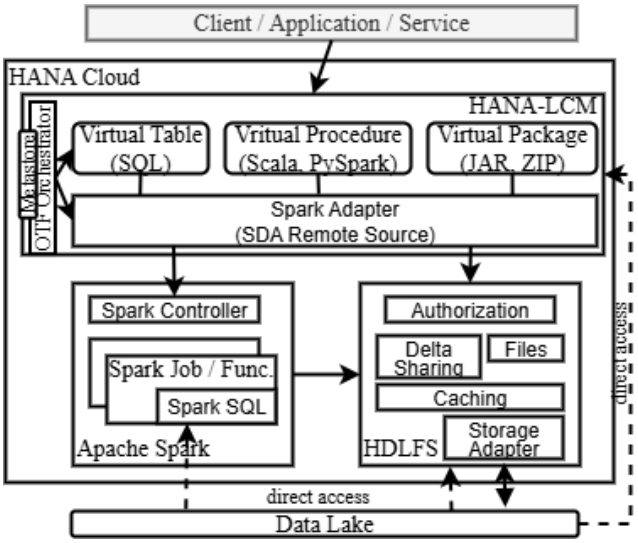


Figure 10: OTF Lifecycle Management and Spark Integration

(b) where the table's lifecycle is managed by this HANA instance. In both cases, SQL processing is performed in HANA, whereas Spark is only used to perform transactional operations (e. g., DDL, transformations). It is worth noting that the architecture leverages LH-capability for optimal choice of compute engines (HANA or Spark) based on the use-case.

```
(a) CREATE VIRTUAL TABLE ITEMS
    AT <HDLF Remote Source> AS DELTA
(b) CREATE VIRTUAL TABLE ORDERS
    AT <HDLF Remote Source> AS DELTA
    USING REMOTE_CONTROLLER <Spark Remote Source>
```

Listing 1: Modes to create OTF tables in SAP HANA

The HANA SQL interface allows for describing Spark applications as virtual procedures in HANA, containing programs, e. g., written in SparkSQL or using DataFrame APIs. That allows users to specify and perform required transformations on the OTF tables. The execution of virtual procedures is done by an orchestrator that creates and executes Spark jobs through a Spark controller and returning the results. The HANA SQL interfaces also support the necessary configuration, monitoring, and troubleshooting of remote jobs, providing complete integration.

When the virtual procedures are executed by a HANA DB user, the Spark integration takes care of creating and executing jobs on the remote Spark engine and transporting outcomes back to the HANA users in an efficient, column-oriented format. Such data transfer also requires handling differences in datatype support between HANA and Spark engine.

8.2 Atomic Operations

Maintaining data consistency across different transaction domains and heterogeneous artifacts of HANA and OTF tables is crucial. Applications demand atomic operations to ensure transactional integrity. However, SQL's single-statement transaction model and HANA's transactional capabilities, which include multi-statement transactions, creates a significant conflict. This mismatch can lead to partial commits, jeopardizing data integrity and undermining

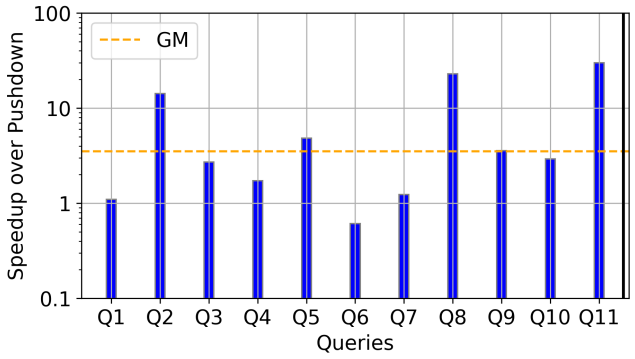


Figure 11: Speedup direct vs. pushdown in LH benchmark.

application consistency. Furthermore, this discrepancy renders traditional mechanisms such as two-phase commit (2PC) protocols unsuitable for ensuring atomicity in this hybrid environment.

To address this challenge, we used and extended the SAGA pattern [21, 62]. In this approach, operations on native HANA artifacts and OTF tables are executed within the scope of a single HANA transaction. HANA acts as the orchestrator, ensuring that all remote OTF table participants eventually converge to the same commit state as the HANA transaction, even in the presence of failures. To achieve this, HANA maintains a metastore to track the state of remote OTF operations associated with a given HANA transaction. This metastore is populated before executing remote OTF operations and truncated upon successful commit or rollback of the HANA transaction and associated remote OTF tables.

Remote OTF operations are committed immediately upon execution by the remote Spark engine, adhering to Spark's single-statement transaction behavior. In the event of a rollback, triggered by errors such as a failed remote OTF operation or a user-issued rollback, the metastore facilitates undo (compensation) of operations for all registered remote OTF operations under the HANA transaction. We define a compensation action for an operation as a sequence of idempotent steps, reversing the effects of the operation.

Defining a compensation action for a table operation can be a complex task as it is based on the operation logic. However, availability of OTF time travel simplifies compensation operations. Before performing an operation on a OTF table, HANA registers the latest version of the OTF table in the metastore. If the operation needs to be compensated then the OTF table is restored to the version at start of the operation. This stands as base mechanism to perform compensation for any unit operation on the OTF table.

It is also possible to define compensation actions which are based on the operation logic. This is useful when certain compensation actions might require additional processing beyond changing the state of an OTF table (time travel is not applicable). For example: `DROP TABLE` compensates `CREATE TABLE`. Time travel is not sufficient in this case, as compensation needs to drop table entries from Spark catalog and delete object store files. Similarly, `CREATE TABLE` compensates `DROP TABLE`. If a transaction has performed multiple operations on the same OTF table then rollback needs to ensure compensation actions are performed in the reverse order.

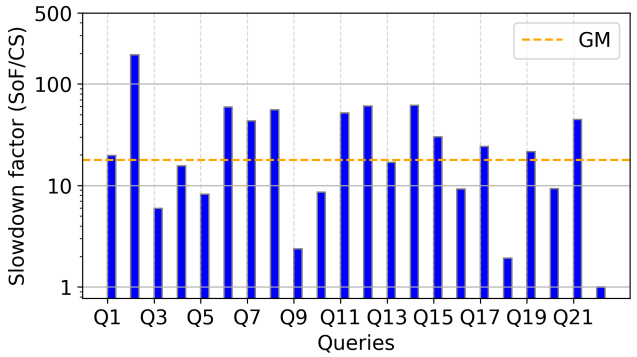


Figure 12: Latency impact factor of SoF relative to HANA in-memory for TPC-H queries, SF 1k.

9 EXPERIMENTAL EVALUATION

In this section, we compare our pushdown / DL-Select and direct access / SoF architecture solutions using our LH benchmark from Sect. 3, study the "data lake tax", and assess the effects of data size scaling, before we evaluate the scaling optimization using ECNs based on TPC-H, guided by the questions in subsection titles.

9.1 System Resources and Configurations

Regarding the resource configuration, if not stated otherwise, subsequent experiments use a data in memory to object storage ratio of roughly 1:10 with an upper limit of one single ECN with 60vCPU and 960GB RAM. In experiments with multiple ECNs the resources are divided by their number, leading to multiple, smaller ECNs.

Regarding a potential impact of SoF on concurrent queries, HANA workload classes enable administrators to manage and optimize resource consumption within the database system [44]. They provide dynamic resource allocation and control at the session or statement level, ensuring efficient utilization of system resources and preventing individual queries from monopolizing the entire system. As discussed in Sect. 5 and Sect. 6, we have configured a workload class for this experimental evaluation in order to maximize thread and ECN utilization. However, I/O throughput is not bound to this configuration, since the threads waiting for network responses are not restricted to the thread usage limits. In our evaluation, we did not observe I/O or CPU resource contention under these configurations.

9.2 What is the Overhead of Pushdown vs. Direct Access Architectures?

In the first experiment we investigate the fundamental difference between the pushdown and direct access approaches. To that end we run the LH benchmark with a data set size of 20GB for 11 out of 13 queries; queries Q12 and Q13 are left out as they have more complex logic (cf. Sect. 3) that would bias the results toward the direct access approach. The data is provided via Delta Sharing. For direct access we use the FileAdpater on one HANA ECN, while for pushdown we employed up to 50 DL-Select instances with sizes like 8 vCPU and 32GB RAM each. Fig. 11 shows the results as relative speedup of direct access over pushdown (baseline).

Note that the underlying access capabilities to the data are the same: both FileAdapter and DL-Select can push simple predicates, conjunctions, disjunctions as well as simple aggregations down to the point where the files are read, leading in particular to the same opportunities for file pruning and min-/max-skipping.

The disadvantage of a pushdown to DL-Select is the extra network hop getting the results to HANA. This is a decisive effect in most queries (especially Q2, Q8, Q11) that have complex joins on large intermediate results. The speedup of direct access ranges up to factor 30× for some queries with a geomean (GM) of 3.7×. Where DL-Select can compete or even shine compared to the *single* HANA ECN are scan-intensive queries like Q1, Q6 and Q7 where the capability to scale as an independent service can come to bear.

9.3 How much is the "Data Lake Tax"?

In the second experiment we investigate the performance impact of SoF on data lake compared to the in-memory HANA Column Store (CS). We run TPC-H queries with scale factor (SF) 1k on the same instance and track the required I/O for SoF in GBs.

The results are shown in Fig. 12 with a geomean (GM) "tax" of the current SoF version is ≈ 17×. The queries with larger tables, such as `LINEITEM` and `ORDERS`, generally show larger amount of SoF I/O and thus longer latency. However, there are some outliers where in-memory execution shows relatively small improvement over SoF (e. g., Q18). Those queries are affected by a suboptimal utilization of the in-memory index operator, which is not enumerated for SoF.

We emphasize these findings provide just a preliminary snapshot. We leave it as a future work to offer a systematic exploration of the data lake tax, across a broader spectrum of workloads, system sizes, and optimization configurations.

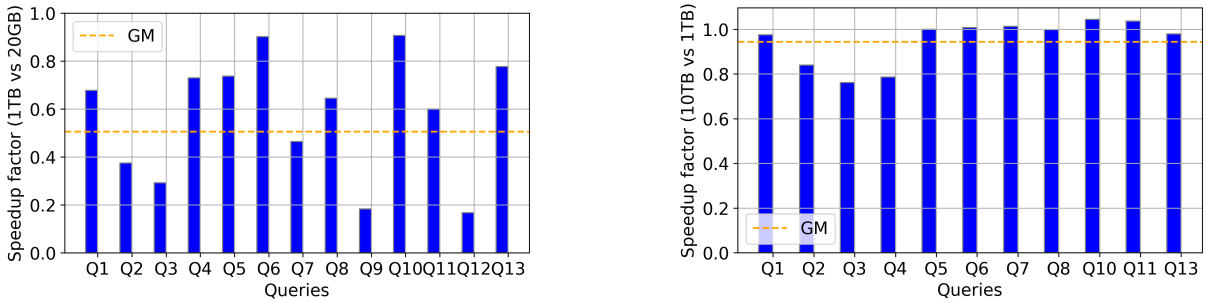
9.4 What is the Impact of Data Size Scaling with a Single ECN?

In this experiment, we explore the impact of data size scaling for a single ECN using the LH benchmark. Data lakes are meant to allow for accessing large amounts of data. We add the two queries Q12 and Q13, measure only the direct access case using one ECN, and increase the data size stepwise, i. e., 20GB to 1TB (factor 50×), 1TB to 10TB (factor 10×), with constant Parquet file size 4GB.

The results were normalized to 1.0 as shown in Fig. 13 for linear scaling with the data. Figure 13a shows effects for smaller data that leads to higher fluctuations and sub-linear scaling, due to limited I/O optimizations on small data sets and tables with few Parquet files. Going to larger, more realistic data sizes in Fig. 13b, the data processing with HANA mostly scales linear to the data size increase. However, for queries Q9 and Q12, which scale very well for 1TB, we encountered memory management challenges at scale of 10TB. Those showed us some opportunities for improvements, which are in progress, but unfortunately for now we had to exclude them from the experiment.

9.5 How well is DQP Compute Scaling with Multiple ECNs?

To assess scalability along two dimensions of increased data size and more available compute, we performed measurements using the TPC-H benchmark at scale factors SF-1k (1TB), SF-10k (10TB), and SF-100k (100TB). Fig. 14 presents the execution times for the TPC-H queries, with outliers removed to ensure clarity. The left axis features execution times on a logarithmic scale, with vertical bars signifying the execution duration for each query: blue bars



(a) 20GB vs. 1TB (factor 50), normalized to 1

(b) 1TB vs. 10TB (factor 10)

Figure 13: Size scaling using LH benchmark for smaller (20GB to 1TB) and bigger (1TB to 10TB) data.

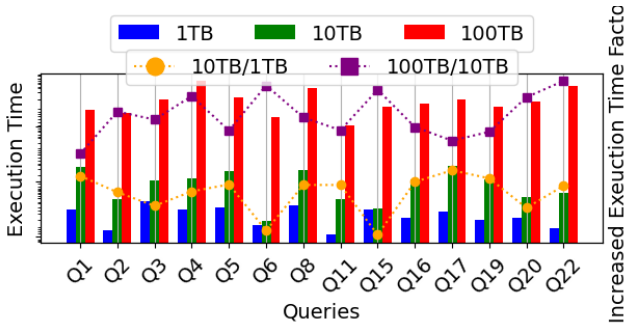


Figure 14: ECN scalability (data sizes, TPC-H benchmark).

for SF-1k, green for SF-10k, and red for SF-100k. The right axis illustrates the factor of increased execution time. The dotted lines represent comparative factors, with a round marker line showing the comparison between SF-1k and SF-10k, and a square marker line showing the comparison between SF-10k and SF-100k.

The results indicate a 7.6× increase in execution time for the 10TB/1TB and a 228× increase for the 100TB/10TB. Given the scale factor increase of 10× for both 10TB/1TB and 100TB/10TB, the anticipated execution time increase should be 10×. However, for the 10TB/1TB comparison, performance shows a favorable increase of 7.6× in execution time. In contrast, the 100TB/10TB scenario indicates scalability issues which are caused by group-by operations and additional communication overhead between ECNs. A more detailed analysis will be part of future work.

10 SUMMARY AND CONCLUSIONS

In-situ processing of data in data lakes by relational (in-memory) database engines remains a challenge. In this work we show how SAP HANA Cloud natively supports complex analytical queries over data in data lakes for modern lakehouses. We follow a direct access architecture style by integrating the necessary file access into HANA's federation layer and specify necessary changes to the HANA execution engine and optimizer. The required elasticity is reached through HANA's elastic compute nodes, and we showed metadata and transactional caching for multidimensional analytical queries. With respect to our design decisions we report the following key takeaways:

- The direct access architectures style morphs the multi-network hop pushdown style into an extra hop within HANA, allowing for using HEX, but disallowing highly elastic compute on tiny compute instances.
- The integration of OTF tables in a loosely coupled, federation approach is elegant for read-only cases based on existing virtual tables and required minimal changes to HANA, but makes write cases challenging.
- The usage of ECNs allows for efficient data loading and compute scaling for DQP on data lakes but could be made even more elastic in terms of HEX on serverless compute.
- The selection of caches and their impact on locality are crucial to achieve good performance and for cost-effectiveness. With its comparably smaller storage footprint, metadata caches become relevant. With non-co-located compute and storage, caching and selection of eviction strategy are crucial. Cache sizes are more important than eviction strategy.
- Lakehouse systems are essentially composable data systems with different transaction domains, type systems, and semantics that either require sophisticated data system solutions or standardization.

We identified the following (research) topics—potentially interesting to the community—that need to be addressed by …

- … studying on more specialized, elastic database operator compute for complex queries.
- … specifying OTF support for multi-table, multi-query transactions with concurrent readers and writers.
- … elaborating serverless vs. ECN-style compute cases regarding elasticity and performance vs. costs and robustness (e. g., adding ad-hoc compute for out-of-memory situations).
- … studying the selection of caches and their interdependencies for complex HTAP stacks.
- … standardizing composable data systems like lakehouses using open formats, engines, transactions and type systems.

Like any other rapidly evolving domain, SAP HANA's support for query processing on data lakes is and will be an ongoing endeavor, and we share the current state with our community.

ACKNOWLEDGMENTS

We thank all involved SAP colleagues, especially Anisoara Nica, Dae Ho Kim, Hak-Wo Kim, Ilju Lee, Jinyong Lee, Ji-won Park, Johannes Alberti, Joo Yeon Lee, Joo Young Yoon, Lars Hömke, Martin Kolb, Sebastian Droll, Till Merker, Yeonghee Choi, YounKyoung Lee.

REFERENCES

[1] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh M. Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (2022), 72–79. https://doi.org/10.1145/3524284

[2] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Jim Chen, Min Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh Sr, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Jun'ichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divy Agrawal, Jeffrey F. Naughton, Sujata Kosalge, and Hakan Hacigümüs. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *Proc. VLDB Endow.* 14, 12 (2021), 2986–2998. https://doi.org/10.14778/3476311.3476377

[3] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[4] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045

[5] Edmon Begoli, Ian Goethert, and Kathryn Knight. 2021. A Lakehouse Architecture for the Management and Analysis of Heterogeneous Data for Biomedical Research and Mega-biobanks. In *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*, Yixin Chen, Heiko Ludwig, Yicheng Tu, Usama M. Fayyad, Xingquan Zhu, Xiaohua Hu, Suren Byna, Xiong Liu, Jianping Zhang, Shirui Pan, Vagelis Papalexakis, Jianwu Wang, Alfredo Cuzzocrea, and Carlos Ordonez (Eds.). IEEE, 4643–4651. https://doi.org/10.1109/BIGDATA52589.2021.9671534

[6] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2326–2339. https://doi.org/10.1145/3514221.3526054

[7] Thomas Bodner, Theo Radig, David Justen, Daniel Ritter, and Tilmann Rabl. 2025. An Empirical Evaluation of Serverless Cloud Infrastructure for Large-Scale Data Processing. *CoRR* abs/2501.07771 (2025). https://doi.org/10.48550/ARXIV.2501.07771 arXiv:2501.07771

[8] Nicolas Bruno, César A. Galindo-Legaria, Milind Joshi, Esteban Calvo Vargas, Kabita Mahapatra, Sharon Ravindran, Guoheng Chen, Ernesto Cervantes Juárez, and Beysim Sezgin. 2024. Unified Query Optimization in the Fabric Data Warehouse. 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, 18–30. https://doi.org/10.1145/3626246.3653369

[9] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[10] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *Proc. VLDB Endow.* 14, 11 (2021), 2432–2444. https://doi.org/10.14778/3476249.3476292

[11] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782. https://doi.org/10.14778/3611479.3611486

[12] Kira Duwe, Angelos-Christos Anadiotis, Andrew Lamb, Lucas Lersch, Boaz Leskes, Daniel Ritter, and Pinar Tozun. 2025. The Five-Minute Rule for the Cloud: Caching in Analytics Systems. In *CIDR*. CIDR, www.cidrdb.org.

[13] Pavan Edara and Mosha Pasumansky. 2021. Big metadata: when metadata is big data. *Proc. VLDB Endow.* 14, 12 (July 2021), 3083–3095. https://doi.org/10.14778/3476311.3476385

[14] Soukaina Ait Errami, Hicham Hajji, Kenza Ait El Kadi, and Hassan Badir. 2023. Spatial big data architecture: From Data Warehouses and Data Lakes to the LakeHouse. *J. Parallel Distributed Comput.* 176 (2023), 70–79. https://doi.org/10.1016/J.JPDC.2023.02.007

[15] The Apache Software Foundation. 2018. Apache Spark - Unified Engine for large-scale data analytics. https://spark.apache.org/. Accessed: 2025-03-17.

[16] The Apache Software Foundation. 2021. Hello from Apache Hudi | Apache Hudi. https://hudi.apache.org/. Accessed: 2025-03-17.

[17] The Apache Software Foundation. 2024. Apache Iceberg - Apache Iceberg. https://iceberg.apache.org/. Accessed: 2024-04-23.

[18] The Apache Software Foundation. 2024. Apache ORC - High-Performance Columnar Storage for Hadoop. https://orc.apache.org/. Accessed: 2025-03-17.

[19] Trino Software Foundation. 2024. Trino | Distributed SQL query engine for big data. https://trino.io/. Accessed: 2025-03-17.

[20] 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.

[21] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM Press, 249–259. https://doi.org/10.1145/38713.38742

[22] Tobias Götz, Daniel Ritter, and Jana Giceva. 2025. LakeVilla: Multi-Table Transactions for Lakehouses. *arXiv preprint arXiv:2504.20768* (2025).

[23] Jim Gray and Gianfranco R. Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM Press, 395–398. https://doi.org/10.1145/38713.38755

[24] 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* (2025), 1–12.

[25] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (2023), 2090–2102. https://doi.org/10.14778/3598581.3598584

[26] Robert B. Hagmann and Domenico Ferrari. 1986. Performance Analysis of Several Back-End Database Architectures. *ACM Trans. Database Syst.* 11, 1 (1986), 1–26. https://doi.org/10.1145/5236.5242

[27] Sasun Hambardzumyan. 2023. Deep Lake: A Lakehouse for Deep Learning. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p69-buniatyan.pdf

[28] Gartner Inc. 2023. Gartner Says Cloud Will Become a Business Necessity by 2028. https://www.gartner.com/en/newsroom/press-releases/2023-11-29-gartner-says-cloud-will-become-a-business-necessity-by-2028. Accessed: 2024-07-17.

[29] Oracle inc. 2025. Oracle Exadata. https://www.oracle.com/de/engineered-systems/exadata/. Accessed: 2025-03-17.

[30] Prakhar Jain. 2024. [Protocol Change Request] Delta Coordinated Commits #2598. https://github.com/delta-io/delta/issues/2598. Accessed: 2024-11-25.

[31] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p92-jain.pdf

[32] Kihong Kim, Hyunwook Kim, Jin Su Lee, Taehyung Lee, Mihnea Andrei, Alexander Böhm, Ralf Dentzer, Heiko Gerwens, Irena Kofman, Norman May, Daniel Ritter, and Guido Moerkotte. 2025. Enterprise Application-Database Co-Innovation for Hybrid Transactional/Analytical Processing: A Virtual Data Model and ts Query Optimization Needs. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, to appear.

[33] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Xiangyao Yu, and Matei Zaharia. 2023. Epoxy: ACID Transactions Across Diverse Data Stores. *Proc. VLDB Endow.* 16, 11 (2023), 2742–2754. https://doi.org/10.14778/3611479.3611484

[34] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *CoRR* abs/1109.6885

(2011). arXiv:1109.6885 http://arxiv.org/abs/1109.6885

[35] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26. https://doi.org/10.1145/3589263

[36] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD-/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 5–17. https://doi.org/10.1145/3626246.3653368

[37] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[38] Justin J. Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. 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, 334–346. https://doi.org/10.1145/3626246.3653388

[39] 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.

[40] Norman May, Wolfgang Lehner, P. ShahulHameed, Nitesh Maheshwari, Carsten Müller, Sudipto Chowdhuri, and Anil K. Goel. 2015. SAP HANA - From Relational OLAP Database to Big Data Infrastructure. In *International Conference on Extending Database Technology*.

[41] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[42] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[43] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[44] Stefan Noll, Norman May, Alexander Böhm, Jan Mühlig, and Jens Teubner. 2019. From the Application to the CPU: Holistic Resource Management for Modern Database Management Systems. *IEEE Data Eng. Bull.* 42, 1 (2019), 10–21. http://sites.computer.org/debull/A19mar/p10.pdf

[45] Apache Parquet. 2024. Parquet. https://parquet.apache.org/. Accessed: 2025-03-17.

[46] Conor Power, Hiren Patel, Alekh Jindal, Jyoti Leeka, Bob Jenkins, Michael Rys, Ed Triou, Dexin Zhu, Lucky Katahanas, Chakrapani Bhat Talapady, Joshua Rowe, Fan Zhang, Rich Draves, Marc Friedman, Ivan Santa Maria Filho, and Amrish Kumar. 2021. The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward. In *VLDB 2021*.

[47] Delta Lake Project. 2023. Delta Sharing. https://delta.io/sharing/ Accessed: 2025-6-12.

[48] The Linux Foundation Projects. 2024. Home | Delta Lake. https://delta.io/. Accessed: 2024-04-23.

[49] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *BTW 2023*. Gesellschaft für Informatik e.V., Bonn, 235–258. https://doi.org/10.18420/BTW2023-12

[50] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-speed query processing over high-speed networks. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 228–239. https://doi.org/10.14778/2856318.2856319

[51] SAP. 2025. SAP HANA Cloud – Virtual Tables. https://help.sap.com/docs/hana-cloud-database/sap-hana-cloud-sap-hana-database-data-access-guide/managing-virtual-tables. Accessed: 2025-03-17.

[52] SAP. 2025. SAP HANA Performance Guide for Developers. https://help.sap.com/docs/SAP_HANA_PLATFORM/9de0171a6027400bb3b9bee385222eff/4951c0a07a324da58d5bca9685415b0a.html. Accessed: 2024-12-07.

[53] Mohit Saxena, Benjamin Sowell, Daiyan Alamgir, Nitin Bahadur, Bijay Bisht, Santosh Chandrachood, Chitti Keswani, G. Krishnamoorthy, Austin Lee, Bohou Li, Zach Mitchell, Vaibhav Porwal, Maheedhar Reddy Chappidi, Brian Ross, Noritaka Sekiyama, Omer Zaki, Linchi Zhang, and Mehul A. Shah. 2023. The Story of AWS Glue. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3557–3569. https://doi.org/10.14778/3611540.3611547

[54] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. 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, 347–359. https://doi.org/10.1145/3626246.3653395

[55] Amazon Web Services. 2020. Amazon Redshift Spectrum adds support for querying open source Apache Hudi and Delta Lake. https://aws.amazon.com/about-aws/whats-new/2020/09/amazon-redshift-spectrum-adds-support-for-querying-open-source-apache-hudi-and-delta-lake/. Accessed: 2025-03-17.

[56] Amazon Web Services. 2025. Amazon S3 Select. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html. Accessed: 2025-03-17.

[57] Amazon Web Services. 2025. Amazon VPC Gateway Endpoints. https://docs.aws.amazon.com/vpc/latest/privatelink/gateway-endpoints.html. Accessed: 2025-03-17.

[58] Amazon Web Services. 2025. AWS Aqua. https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/. Accessed: 2025-03-17.

[59] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[60] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, Sarika Iyer, Sasikanth Gottapu, Robert Schulze, Chaitanya Gottipati, Nirvik Basak, Yanhong Wang, Vivek Kandiyanallur, Santosh Pendap, Dheren Gala, Rajesh Almeida, and Prasanta Ghosh. 2019. Native store extension for SAP HANA. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2047–2058. https://doi.org/10.14778/3352063.3352123

[61] Malcolm Singh and Ben Leonhardi. 2011. Introduction to the IBM Netezza warehouse appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*. 385–386.

[62] Martin Stefanko, Ondrej Chaloupka, and Bruno Rossi. 2019. The Saga Pattern in a Reactive Microservices Environment. In *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019*, Marten van Sinderen and Leszek A. Maciaszek (Eds.). SciTePress, 483–490. https://doi.org/10.5220/0007918704830490

[63] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 2–11. https://doi.org/10.1109/ICDE.2005.1

[64] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. In *VLDB 2024*. https://www.amazon.science/publications/why-tpc-is-not-enough-an-analysis-of-the-amazon-redshift-fleet

[65] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. 2022. What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 36–46. http://www.adms-conf.org/2022-camera-ready/ADMS22_merzljak.pdf

[66] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[67] Maryann Xue, Yingyi Bu, Abhishek Somani, Wenchen Fan, Ziqi Liu, Steven Chen, Herman Van Hovell, Bart Samwel, Mostafa Mokhtar, Rk Korlapati, Andy Lam, Yunxiao Ma, Vuk Ercegovac, Jiexing Li, Alexander Behm, Yuanjian Li, Xiao Li, Sriram Krishnamurthy, Amit Shukla, Michalis Petropoulos, Sameer Paranjpye, Reynold Xin, and Matei Zaharia. 2024. Adaptive and Robust Query Execution for Lakehouses At Scale. *Proc. VLDB Endow.* 17, 12 (2024), 3947–3959. https://www.vldb.org/pvldb/vol17/p3947-bu.pdf

[68] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *VLDB J.* 33, 5 (2024), 1643–1670. https://doi.org/10.1007/S00778-024-00867-8

[69] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1802–1805. https://doi.org/10.1109/ICDE48307.2020.00174

[70] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, Erich M. Nahum and Dongyan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[71] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[72] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161. https://doi.org/10.14778/3626292.3626298

[73] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2653–2666. https://doi.org/10.1145/3448016.3457559