
**SAP HANA distributed in-memory database system: Transaction, session, and metadata management**

Erstveröffentlichung in / First published in:


DOI: [http://dx.doi.org/10.1109/ICDE.2013.6544906](http://dx.doi.org/10.1109/ICDE.2013.6544906)

Diese Version ist verfügbar / This version is available on:

[https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-818911](https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-818911)
SAP HANA Distributed In-Memory Database System: Transaction, Session, and Metadata Management

Juchang Lee#1, Yong Sik Kwon#2, Franz Färber*3, Michael Muehle*4, Chulwon Lee#5, Christian Bensberg*6, Joo Yeon Lee#7, Arthur H. Lee#8, Wolfgang Lehner*9

#SAP Labs, Korea
1juc.lee@sap.com, 2yong.sik.kwon@sap.com, 3ch.lee@sap.com, 7joo.lee@sap.com
*SAP AG, Germany
4franz.faerber@sap.com, 5michael.muehle@sap.com, 6christian.bensberg@sap.com
*Claremont McKenna College, USA
alee@cmc.edu
*Dresden University of Technology, Germany
9wolfgang.lehner@tu-dresden.de

Abstract— One of the core principles of the SAP HANA database system is the comprehensive support of distributed query facility. Supporting scale-out scenarios was one of the major design principles of the system from the very beginning. Within this paper, we first give an overview of the overall functionality with respect to data allocation, metadata caching and query routing. We then dive into some level of detail for specific topics and explain features and methods not common in traditional disk-based database systems. In summary, the paper provides a comprehensive overview of distributed query processing in SAP HANA database to achieve scalability to handle large databases and heterogeneous types of workloads.

I. INTRODUCTION

An efficient and holistic data management infrastructure is one of the key requirements for making the right decisions at an operational, tactical, and strategic level. The SAP HANA database is the core component of SAP’s HANA roadmap playing the foundation to efficiently support all SAP and non-SAP business processes from a data management perspective [1]. In opposite to the traditional architecture of a database system, the SAP HANA database takes a different approach to provide support for a wide range of data management tasks. For example, the system is organized in a main-memory centric fashion to reflect the shift within the memory hierarchy [9] and to consistently provide high performance without any slow disk interactions.

Completely transparent for the application, data is organized along its life cycle either in column or row format, providing the best performance for different workload characteristics [10]. Transactional workloads with a high update rate and point queries are routed against a row store; analytical workloads with range scans over large datasets are supported by column oriented data structures. In addition to a high scan performance over columns, the column-oriented representation offers an extremely high potential for compression making it possible to store even large datasets within the main memory of the database server. Recent developments in the hardware sector economically allow having off-the-shelf servers with 2 TByte of DRAM. The main-memory centric approach therefore turns the classical architectural paradigm upside-down: While traditional disk-centric systems try to guess the hot data to be cached in main memory, the SAP HANA approach defaults to have everything in main memory; only “cold” data—usually determined by complex business rules and not by buffer pool replacement strategies working without any knowledge of the application domain and corresponding business objects—can be staged out onto disk infrastructures. This allows SAP HANA to support even very large databases in terms of a large number of tables and data volumes sufficient to serve all SAP customers with existing and future applications.

In addition to performance, the SAP HANA database also targets to support business processes from a holistic perspective. For example, the system may hold text documents of products within an order together with structured information of the customer and spatial information of the current delivery route. As outlined in [2], the SAP HANA database provides multiple engines exposing special services. Data entered for example in text format can be extracted, semantically enriched, and transformed into structural data for combination with information coming from an engine optimized for graph-structured analytics. Combining heterogeneous datasets seamlessly within a single query processing environment and providing support for the complete life cycle of data on a business object level are some of the unique features of SAP HANA.

Finally, SAP HANA is positioned to act as a consolidation platform for many different use cases, from an application perspective and from a data management perspective. Multiple SAP and non-SAP applications may run on top of one single SAP HANA instance providing the right degree of...
“isolation”—strict isolation, if required, e.g., from security perspective, deep integration if different datasets are supposed to be merged like in typical data-warehouse scenarios. Having a single SAP HANA landscape reduces operational expenses and therefore TCO in general. However, scalability is required to provide such a degree of service. Therefore, from the very beginning on, the SAP HANA database was designed for scalability in different directions:

- **Scale Up:** Due to main memory requirements SAP HANA was designed to run on “big machines” offering multiple CPUs and a fairly large number of threads.
- **Scale Out:** The SAP HANA database runs in a multi-node environment to balance the need of CPU power and main memory capacity providing the same level of transactional guarantees like in a single node scenario.
- **Scale In:** Scale in typically denotes multi-tenancy support and therefore ability to host multiple logical databases within a single physical instance offering a certain level of schema and data sharing.

**Contributions:** Within this paper, we focus on some core concepts of distributed query processing in order to provide a robust and efficient scale-out solution. We outline the need to balance the gain of larger main memory capacities and larger number of computing units against the complexity coming with a distributed environment. Therefore, we start with an overview of distributed query processing in SAP HANA following the life cycle of an individual query pinpointing specific problems and solutions along the way. Thereafter, we dive into detail for some selected problems and give insights into the conceptual solution design. In summary, the paper provides a comprehensive overview of distributed query processing in SAP HANA and describes some procedures and their optimizations in detail.

II. DISTRIBUTED QUERY PROCESSING IN HANA

As already mentioned, scaling database services over multiple nodes connected via a high-speed network infrastructure implies a variety of challenges. Every single component of a database system has to be “distribution-enabled”, i.e., not only working correctly but also efficiently in a distributed environment. From that perspective, the fact of distribution affects functional as well as non-functional service primitives ranging from distributed (multi-node) query processing to caching strategies of metadata repositories.

The overall goal of the SAP HANA database approach consists in scaling over a reasonably large number of nodes without sacrificing overall system performance and all well-known transactional guarantees, i.e., ACID properties.

The core database challenges can be classified into four major categories: distribution of data, distributed transaction management, distributed metadata management, and distributed query optimization and execution.

A. Deployment Schemes and Data Distribution

As in all high-end database systems, a single table can be split into multiple partitions using hash, round-robin, range partitioning strategies. Individual partitions are then allocated at different nodes pursuing two different strategies. One the one hand, specialized reorganization tools exist to provide advice for the DBA reaching optimal partitioning schemes. For example, the toolset checks incoming workloads on a table usage level to come up with a proposal to either spread out partitions of a table or co-locate different tables in order to avoid multi-node joins or expensive commit protocols. As of now, the toolset is optimized to support specific SAP applications, especially SAP Business Warehouse also considering CPU and memory usage of all active nodes. Based on the reference behavior and current system usage, the reorganization tool makes a proposal of a revised allocation scheme. Future versions of the toolset will act in an application-agnostic way supporting any arbitrary SQL-based workload.

On the other hand, the DBA may directly assign partitions of a table to individual HANA nodes. This manual task is especially beneficial to achieve certain performance characteristics of certain tables. For example, a DBA might want to avoid distributed transactions with network traffic and protocol delay to improve query performance. For example, a single landscape may consist of one very large machine and multiple smaller nodes as shown in Figure 1. The large machine node will then host all “transactionally hot” tables or partition of tables avoiding distributed transactions with network traffic and protocol delay. More analytically oriented applications targeting multiple partitions of historical data or databases coming from external data sources will then hit the parallel nodes to improve query performance. Since all datasets are part of one single SAP HANA landscape, the database system is able to run cross-joins within multi-node transactions, if the query demands it—the allocation and deployment scheme just tries to reduce the communication within the cluster.

![Fig. 1. Asymmetric deployment of an SAP HANA landscape](image)

B. Distributed Transaction Management

In opposite to scale-out solutions like Hadoop, SAP HANA follows the traditional semantics of providing full ACID support. In order to make good the promise of supporting both OLTP and OLAP-style query processing within a single platform, the SAP HANA database relaxes neither any consistency constraints nor any degree of atomicity or durability. On the one side, the SAP HANA database applies traditional locking and logging schemes for distributed
scenarios with some very specific optimizations like optimizing the two-phase commit protocol (subsection III-D) or providing sophisticated mechanisms for session management (subsections III-C.1, III-C.2, and III-C.3). As mentioned the deployment of the system usually reflects the intended use in order to have a benefit of a large node for heavy transaction processing and a number of usually smaller nodes for analytical workloads where the additional overhead of distributed synchronization reflects a relatively small portion of the overall query runtime. Since SAP HANA relies on MVCC as the underlying concurrency control mechanism, the system provides distributed snapshot isolation and distributed locking to synchronize multiple writers. Therefore, the system relies on a distributed locking scheme with a global deadlock detection mechanism avoiding a centralized lock server as a potential single point of failure.

C. Distributed Metadata Management

Within an SAP HANA database landscape, a coordinator node stores and manages all the persistent metadata such as table/view schema, user information, privileges on DB objects, etc. To satisfy requirements for consistent metadata access, the metadata object container provides both MVCC based access and transactional update (ACID) on its contents. It also provides index-based fast object lookup.

In order to improve access to metadata at worker nodes, the concept of metadata caches enables local access to “remote” metadata in a distributed environment. Figure 2 shows the metadata object container and cache in the coordinator and worker nodes. When a component in a worker node requires access to a metadata object located at the (remote) coordinator, the metadata manager first tries to locate it in the cache. If there is no result object in the cache, a corresponding retrieval request is sent to the coordinator. The result is placed within the cache and access is granted to the requesting component. In order to reduce potential round-trips to fetch different entries of metadata, the system applies group caching of tightly related metadata, e.g., a cache request for metadata related to a table also returns metadata about columns, existing indexes, etc. within a single request. For consistent query processing, the access to the metadata cache is tightly coupled with the transaction management.

D. Distributed Query Compilation and Execution

In order to illustrate the key concepts of distributed query processing within SAP HANA, we will follow a query in different scenarios. Within a single node setup, the client connects to a particular server and starts the query compilation process. Figure 3 shows the different steps.

Fig. 3. Query compilation and execution in a single node scenario

The session layer forwards an incoming query request to the optimizer (1). After consulting metadata (2) and checking the plan cache (3), the query will eventually be optimized and compiled. In opposite to other systems, the optimizer embeds substantial metadata into the query plan and returns it to the client. For example, metadata flowing back to the client contains information about the most optimal node to actually start and orchestrate the query execution. Within a single node case, the client sends the query plan to the execution component (4), puts the plan into the (current) query plan cache and starts executing the query in a combination of column- and row-store. In the case of an update, additional log information will eventually be written to the persistency layer to ensure atomicity and durability.

The query flow is a bit more complicated in a multi-node deployment as illustrate in Figure 4. As before, a client may send a query to a dedicated coordinator node (1). After the initial query compilation, the returned query plan contains information about the node where the query should be executed (4). This recommendation is based on data locality for the particular query and the current system status. The client then sends the query to the recommended (worker) node (5) and re-compiles the query with respect to the node-specific properties and statistics (6). As before, the optimization step requires access to metadata, which is either already available at the current node (7) or requested on-the-fly from the coordinator (8). After re-compilation, the query is executed (10) by extracting the plan from the plan cache (11), passing it to the execution component which again routes the individual requests to the local storage structures (12) and potentially local persistency layer (13). Figure 5 illustrates the benefits of using statement re-routing with a simple single-table select. The figure shows three cases: (i) single-node case; (ii) multi-mode case with statement routing turned on; and (iii) multi-node case with statement routing turned off. As we can see, cases (i) and (ii) are virtually identical. Case (iii) however is significantly and consistently slower than the other two cases. In addition to statement routing, this scenario also
Fig. 4. Query re-compilation at remote node

exploits the optimization of using a local transaction token to reduce the communication cost between nodes, effectively behaving like a single-node scenario.

Fig. 5. Sample scenario showing benefits of client-side statement routing

The most general case of a distributed query execution spans some of the operators over multiple nodes. While the initial compilation and client-side routing including re-compilation, metadata access, etc. are equivalent to the former case, a multi-node query requires two additional steps (Figure 6). Before starting the execution of the query, the transaction manager consults the coordinator (1), registering the transaction and retrieving a global transaction token holding the visibility information for that particular query (in general for all queries within a transaction). For execution, the executor component distributes the query plan over multiple nodes by shipping sub-query plans with the transaction token and triggering the individual processing steps (5). The decision where to run what part of the global query plan and when to move intermediate data between the nodes is determined on a cost-based basis during query compilation. Obviously, the distributed query optimizer is a non-trivial piece of the SAP HANA database in order to reduce network traffic and decide which part of a query can be parallelized over multiple nodes.

E. Some Experimental Results

Without diving into too much detail at this point, we give some overall performance numbers showing the potential (e.g., read scalability, write behavior, etc.) of the SAP HANA database system with respect to different workload scenarios. The reported numbers are based on an SAP HANA landscape with 31 nodes, each consisting of 0.5 TByte main memory, 20 physical CPU cores, and local disks with 700 MB/s throughput for local logging. A 10 Gbps network was used for the interconnect. The numbers are taken from a real customer SAP Business Warehouse installation on top of SAP HANA database; the SAP HANA database (after compression) showed an overall size of 100 GByte.

<table>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSOLUTE RUNTIMES (IN SECONDS) AND SPEEDUP OF DIFFERENT WORKLOADS</td>
</tr>
</tbody>
</table>

| # nodes | 1 | 3 | 7 | 15 |
|---------------------------|
| query                   | 1.346 (1.0a) | 0.708 (1.5a) | 0.311 (4.3a) | 0.231 (5.8a) | 0.340 (3.5a) |
| #2                     | 0.125 (1.0a) | 0.122 (1.0a) | 0.131 (0.9a) | 0.182 (0.6a) | 0.273 (0.4a) |
| loading chain           | 2.004 (1.0a) | 1.399 (1.6a) | 0.378 (5.6a) | 0.197 (10.1a) | 0.157 (12.7a) |
| #1                     | 0.706 (1.0a) | 0.235 (3.0a) | 0.100 (7.0a) | 0.050 (13.5a) | 0.029 (25.2a) |
| #2                     | 0.706 (1.0a) | 0.235 (3.0a) | 0.100 (7.0a) | 0.050 (13.5a) | 0.029 (25.2a) |

The first workload consists of typical analytical queries issued against the warehouse from Business Intelligence tools. While Table I shows the average execution times over 200 single queries comprising the two different types of workloads, Figure 7 depicts the speedup with a growing number of nodes:

1) Query 1 is a representative set of queries requiring a full table scan over the fact table, which can be parallelized over multiple worker nodes. As we can see, performance increases with the number of nodes but drops as the landscape uses a partitioning scheme stretching the fact table over all 31 nodes implying a larger number of local joins. The performance drop is primarily caused by the size of the resulting partitions. If the partitions are too small, the resulting overhead and the large number of local join operations with distribution of the join partner is counterproductive with respect to the overall query performance.

2) In the opposite, query set 2 touches a small piece of data located only at a single node. As can be seen, the scale-out does not significantly affect query performance, neither in the positive nor in the negative way; the speedup therefore remains constant independent of the number of nodes.
As outlined in the introduction, the SAP HANA database system is designed to deliver extremely good performance for OLAP- and OLTP-style query workloads. We therefore also consider a typical update-heavy workload in the context of a real SAP BW scenario. After loading extracted raw data into the BW system and applying transformations with respect to data cleansing and harmonization requirements, new data is eventually “merged” with the existing state of the fact table implying a mix of insert, delete, and update operations on the potentially very large fact table. Following this “activation step”, the system is applying reorganization steps to better reflect the multidimensional model. Again this “compression step” within the overall SAP BW loading process is a write-intensive step which can nicely be parallelized over multiple nodes.

1) The loading chain type 1 shown in Table I reflects an activation step. As can be seen in Figure 8, the system scales very nicely with the number of nodes, despite the heavy write process.

2) The compression step (loading chain type 2) finally shows excellent scalability with a speedup of 13.6 for 15 nodes or 25.2 for the 31 node case and therefore demonstrates the scale-out capability in an optimal setting with respect to the physical database design.

III. SELECTED OPTIMIZATIONS

In this section we describe in detail some selected optimization problems in the SAP HANA distributed landscape and provide insights into the conceptual solution design for them.

A. Client-Side Statement Routing

In a distributed database environment a query statement issued by a client may be executed in any one of the available nodes. The problem is determining the optimal node to run it on and route it to the node, which in HANA would be any one of the worker nodes in the system. HANA uses what we call client-side statement routing to achieve this. Affinity information between a given compiled query and its accessed data location is cached transparently at the client library. So, without changing the existing client programs and also without requiring any more information on clients, any arbitrary query given by a client program can be routed to a desirable server node.

Figure 9 shows a conceptual view of the solution on how a query is routed by client library and how the affinity information is maintained at the client side. First, at compile time the desired server location of a given query can be decided and it is returned to the client library. On a subsequent repeated execution of such a compiled query, it is routed to the associated server node directly by the client library. Sometimes, by DDL operations such as table movement or repartitioning, however, the desired location of a given query can be changed as for Table T2 in Figure 9. In such a case, on the next query execution following the DDL operation, such inefficiency is detected automatically by comparing the metadata timestamp of the client-side cached information with the server-side latest metadata timestamp.
value and then the client-cached information is updated with the latest information.

Fig. 9. Client-side statement routing

How is the optimal target location decided given a client-side statement then? A server extracts tables and/or views needed to run the statement (or a stored procedure if that is the case) and returns the locations (nodes) of the target tables and/or views. The client then does the following: If the number of nodes returned is 1, route the query to that node. If the number is greater than 1, then do the following: If the table returned is a hash partitioned table or a round-robin partitioned table, and furthermore the query is an INSERT query, then evaluate the input value of the partitioning key and use the result as the target location. Otherwise, route it to the returned nodes in a round-robin fashion.

The routing decision can be resolved at compile time (which we call static resolution) in some cases or at run time (which we call dynamic resolution) in other cases. We now discuss static resolution optimizations in the rest of this subsection and dynamic resolution optimizations will be discussed in the subsection that follows.

Let us consider stored procedures first. How is it decided at which node a stored procedure is executed? Within a stored procedure there may be a variety of multiple queries. Usage pattern of each query in a stored procedure is used in making the routing decision. For example, if one query executes multiple times in a procedure, we can be a little smarter. Suppose there are 10 queries and one of them is in a loop, in which case a higher priority is given to the query in a loop in making the decision.

In some cases analyzing a stored procedure at a server alone may not be sufficient, in which case the server could get some hints from the client code. Application programs could pass some domain knowledge into the application code.

With the client-side statement routing mechanism in place there is no more 1-to-1 mapping between a session and a physical connection or between a transaction and a physical connection. This poses a technical challenge though, which has to do with session context management. A session context is a property of a session (for example, user name, locale, transaction isolation mode, etc.). Without client-side statement routing, there is only 1-to-1 mapping between a logical session and a client-server physical connection, which means that the session context information can be stored in the server-side connection. With client-side statement routing, however, it should be shared across multiple client-server physical connections. In HANA, some of the session context information is also cached at the client library side. The cached information can then be used when it has to make a new physical connection to a different server within the same logical session without having to contact the initial physical connection of the logical session.

In this subsection, we primarily focused on the cases in which the desired execution location of a query is decided at query compilation time. However, there are other types of queries whose desired location can be decided dynamically at run time. Such cases are described in the subsection.

B. Dynamic Resolution in Statement Routing

1) Table partitioning: Using the partitioning feature of the SAP HANA database, a table can be partitioned horizontally into disjunctive sub-tables or “partitions”, each of which may be used by each node of a distributed HANA database system. Problem is how to partition a table optimally so that each partition is shipped to the optimal worker node. Partitioning may be done by using one of three strategies: hash, round-robin, and range. Both hash partitioning and round-robin are used to equally distribute the partitions to worker nodes. Range partitioning can be used to create dedicated partitions for certain values or certain value ranges to be distributed to worker nodes.

Let us consider hash partitioning in more detail. To decide a desirable partition of a given query to a partitioned table, we need to consider the table’s partitioning specification (or the partitioning function) and the execution-time input values of its partitioning keys. The partitioning specification can be given to the client library on the query compilation, but the execution-time input values can be known only during the query execution time. This input value interpretation logic is normally done by the server-side query processing layer. For this optimization though, such logic is also shipped together to the client library as well.

This partitioning optimization technique is used to support client-side routing of various statements such as inserts, selects, updates, and deletes as long as their WHERE clause includes partitioning key information.

2) Load balancing: Each node in a distributed database system may be utilizing the key resources, i.e., CPU and memory, differently with different workloads at any given time. For example, one node may be busy with CPU-bound tasks while there may be at least one other node that is not busy at all at that point in time. With HANA as a main-memory database system, memory is being used not only for processing but also for storage, i.e., holding table data. Again for example, one node may almost be out of memory while there may be at least one other node with plenty of available memory. It is a correctness concern as well as performance concern, i.e., we cannot ever get into a situation where a node
fails because it ran out of memory. Therefore, it is important to balance not only the processing load but also storage load among different nodes in the system.

In addition to improving affinity between data location and processing location, we can extend client-side statement routing to achieve better load balancing across HANA server nodes. When returning a query result to a client library, a HANA server can return its memory and CPU resource consumption status together with the result, i.e., without making any additional round trip. Alternatively, the client library can periodically collect the resource status of HANA server nodes. If the client detects that one of computing nodes does not have enough CPU or memory resource at that point in time, the client library tries to temporarily re-route the current query to other nodes. Furthermore, at query compilation time the client library can attach its expected memory consumption to the query plan. If this information is then cached and attached to the compiled query at the client side, the client library can perform more efficient re-routing.

C. Distributed Snapshot Isolation (Distributed MVCC)

For distributed snapshot isolation figuring out how to reduce the overhead of synchronizing transaction ID or commit timestamp across multiple servers belonging to a same transaction domain has been a challenging problem [3], [4], [5]. SAP HANA database focuses on optimization for a single-node query which is executed without accessing any other node. If the partitioning and table placement is done optimally, then most of queries can be processed within a single node. For long-running analytical queries that have to be spanned across multiple nodes, the overhead incurred by communicating such transactional information will be relatively ignorable compared to the overall execution time of the query. So, our optimization choice that favors single-node queries can make sense for many applications.

The question is how is it figured out whether a transactional snapshot boundary will only touch one node or not in advance? It is feasible especially under the Read-Committed isolation mode [6], which is the default isolation mode in many SAP applications.

In Read-Committed isolation mode the MVCC snapshot boundary is the lifetime of the query, which means that the transaction does not need to consider any other query. So, the isolation boundary for the query can be determined at compile time, i.e., at that time it can figure out exactly what parts of which tables to access to execute the query. When the query finishes, the snapshot finishes its life as well, i.e., the snapshot is meaningful only on that local node while the query is executing. The entire transaction context that is needed for a query to execute is captured in a data structure called transaction token (described below) and is cached on a local node. For the “Repeatable Read” or “Serializable” isolation level, the transaction token can be reused for the queries belonging to the same transaction, which means that its communication cost with the coordinator node is less important.

Whenever a transaction (in a transaction-level snapshot isolation mode) or a statement (in a statement-level snapshot isolation mode) starts, it copies the current transaction token into its context (called snapshot token). And, the transaction (or statement) decides which versions should be visible to itself based on the copied ‘snapshot token’.

Now we describe distributed snapshot isolation (or distributed MVCC). In our transaction protocol every transaction started at a worker node should access the transaction coordinator to get its snapshot transaction token. This could cause (1) a throughput bottleneck at the transaction coordinator and (2) additional network delay to the worker-side local transactions. To remedy these situations we use three techniques: (i) one that enables local read-only transactions to run without accessing the global coordinator; (ii) another that enables local read or write transactions to run without accessing the global coordinator; and (iii) third that uses “write TID buffering” to enable multi-node write transactions to run with only a single access to the global coordinator. We now describe all three techniques in order.

1) Optimization for worker-node local read transactions or statements: Every update transaction accesses the transaction coordinator to access and update the global transaction token. Read-only statements on the other hand just start with its cached local transaction token. This local transaction token is refreshed

- by the transaction token of an update transaction when it commits on the node, or
- by the transaction token of a ‘global statement’ when it comes in to (or started at) the node.

If the statement did not need to access any other node, it can just finish with the cached transaction token, i.e., without any access to the transaction coordinator. If it detects that the statement should also be executed in another node, however, it is switched to the ‘global statement’ type, after which the current statement is retried with the global transaction token obtained from the coordinator.

Single-node read-only statements/transactions do not need to access the transaction coordinator at all. This is significant to avoid the performance bottleneck at the coordinator and to reduce performance overhead of single-node statements (or transactions).

2) Optimization for worker-side local write transactions: Each node manages its own local transaction token independently of the global transaction token. Even the update transaction can just update its own local transaction token if it is a single-node transaction. The difference is that each database record has two TID (or Transaction ID) columns for MVCC version filtering: one for global TID and another for local TID. (In the existing other schemes, there is only one TID/Commit ID column.) If it is a local-only transaction, it reads/updates the local TID. If it is a global transaction, however, it reads either global or local TID (reads a global TID if there is a value in its global TID column; otherwise, reads a local TID) and updates both global and local TIDs. So, the global transactions carry two snapshot transaction tokens: one for global transaction token and another for the current worker node's local transaction token.
In the log record both global and local TIDs are also recorded if it is a global transaction. On recovery a local transaction's commit can be decided by its own local commit log record. Only for global transactions it is required to check the global coordinator. Here again statement type switch protocol is necessary as it is the case in case 1) above.

3) Optimization for multi-node write transactions: A multi-node write transaction needs a global write TID as all multi-node transactions do. The optimization described in case 2) above would not help. It can however be handled by the write TID buffering technique which we describe now.

In a HANA scale-out system one of the server nodes becomes the transaction coordinator which manages the distributed transaction and controls two phase commit. Executing a distributed transaction involves multiple network communications between the coordinator node and worker nodes. Each write transaction is assigned a globally unique write transaction ID. A worker-node-first transaction, one that starts at a worker node first, should be assigned a TID from the coordinator node as well, which causes an additional network communication. Such a communication might significantly affect the performance of distributed transaction execution. This extra network communication is eliminated to improve the worker-node-first write transaction performance.

We solve this problem by buffering such global write TIDs in a worker node. When a request for a TID assignment is made the very first time, the coordinator node returns a range of TIDs which gets buffered in the worker node. The next transaction which needs a TID gets it from the local buffer thus eliminating the extra network communication with the coordinator node. A few key challenges are determining optimal buffer size as a parameter and deciding when to flush the buffer if they are not being used.

Therefore, by combination of the optimizations described in this subsection SAP HANA provides transparency of transaction performance regardless of where it is started and committed without losing or mitigating any transactional consistency.

D. Optimizing Two-Phase Commit Protocol

Two-phase commit (2PC) protocol is widely used to ensure atomicity of distributed multi-node update transactions. A series of optimization techniques we describe in this subsection are our attempts to reduce the network and log I/O delays during a two-phase commit thus increasing throughput.

1) Early commit acknowledgement after the first commit phase: Our first optimization is to return commit acknowledgement early after the first commit phase [7], [8], as shown in Figure 10.

Right after the first commit phase and the commit log is written to the disk, the commit acknowledgement can be returned to the client. And then, the second commit phase can be done asynchronously.

For this optimization three things are considered.

1) Writing the commit log entries on the worker nodes can be done asynchronously. During crash recovery then some committed transactions can be classified as in-doubt transactions, which will be resolved as committed finally by checking the transaction's status in the coordinator.

2) If transaction tokens are cached asynchronously on the worker nodes, the data is visible by a transaction but not by the next (local-only) transaction in the same session. This situation can be detected by storing the last transaction token information for each session at the client side. And then, until the second commit phase of the previous transaction is done, the next query can be stalled.

3) If transactional locks are released after sending a commit acknowledgement to the client, a 'false lock conflict' may arise by the next transaction in the same session. This situation can however be detected by the same problem with (2) above. If this is detected, the transaction can wait for a short time period until the commit notification arrives to the worker node.

2) Skipping writes of prepare logs: The second optimization is to remove additional log I/Os for writing prepare-commit log entries.

In a typical 2-phase-commit the prepare-commit log entry is used to ensure that the transaction's previous update logs are written to disk and to identify in-doubt transactions at the recovery time.

- Writing the transaction's previous update logs to disk can be ensured without writing any additional log entry, by just comparing the transaction-last-LSN (log sequence number) with the log-disk-last-LSN. If the log-disk-last-LSN is larger than the transaction-last-LSN, it means that the transaction's update logs are already flushed to disk.
- If we do not write the prepare-commit log entry, we can handle all the uncommitted transactions at recovery time as in-doubt transactions.
Their commitance can be decided by checking with the transaction coordinator. So, the size of in-doubt transaction list can increase, but with less run-time overhead.

3) Group two-phase commit protocol: This is a similar idea to the one described in Subsection III-C.3, but instead of sending commit requests to the coordinator node individually (i.e., one for each write transaction), we can group multiple concurrent commit requests into one and send it to the coordinator node in one shot.

Also, when the coordinator node multicasts a "prepare-commit" request to multiple-related worker nodes of a transaction, we can group multiple "prepare-commit" requests of multiple concurrent transactions which will go to the same worker node.

By this optimization we can get better throughput of concurrent transactions.

Two-phase commit itself cannot be avoided fundamentally for ensuring global atomicity to multi-node write transactions. However, by combination of the optimizations described in this subsection, we can reduce its overhead significantly.

E. Metadata Cache Management

The system-wide metadata catalogue is kept in the coordinator node. As a worker node processes a query, it would have to obtain the necessary metadata information from the coordinator node. To reduce the overhead caused by the network latency in the communication between the coordinator node and a worker node, HANA supports a worker node requesting multiple metadata objects with a single request when beneficial.

When a worker node needs to obtain metadata information from the coordinator node because it does not have that information already cached on it, it will incur IPC. An example can be seen in Step 8 of Figure 4. Since each single metadata cache miss will cause IPC via network communication, it can be a considerable performance penalty for the cases where there are many metadata cache misses for a single query execution. Group caching for metadata is introduced to minimize cache miss penalties of this kind. Instead of checking cache in an on-demand manner, a worker node collects the entire required metadata object IDs first and sends a single request for all the missing objects. This optimization is particularly good for a query with a complex query plan which requires accessing multiple metadata objects of various kinds such as tables, indexes, views, and privileges.

IV. SUMMARY

The SAP HANA database is primarily designed to cover the three difference scaling principles: scale-in, scale-up, and scale-out. In this paper we outlined some of the hard problems in multi-node scenarios, showed the core architectural designs, and give optimization details for some of the problems. Specifically, we discuss optimization for query routing in single and multiple node scenarios, we show optimizations techniques for client-based statement routing, two-phase-commit protocol, and finally give some insights into caching strategies and techniques for metadata catalogue of the SAP HANA database.

ACKNOWLEDGMENT

We explicitly want to express our Thanks to the SAP HANA team, especially to all the readers of a preliminary version of this paper. We also thank Sang K. Cha for suggesting the idea of writing this paper.

REFERENCES

[9] J. Gray, “Tape is dead, disk is tape, flash is disk. ram locality is king,” 2006.