Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads

Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean
SAP Labs Canada, Waterloo, Canada

Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner
SAP SE, Germany

Wolfgang Lehner†
TU Dresden, Dresden, Germany

⟨*firstname.lastname*⟩@sap.com, †wolfgang.lehner@tu-dresden.de

ABSTRACT

We present an overview of our work on the SAP HANA Scale-out Extension, a novel distributed database architecture designed to support large scale analytics over real-time data. This platform permits high performance OLAP with massive scale-out capabilities, while concurrently allowing OLTP workloads. This dual capability enables analytics over real-time changing data and allows fine grained user-specified service level agreements (SLAs) on data freshness. We advocate the decoupling of core database components such as query processing, concurrency control, and persistence, a design choice made possible by advances in high-throughput low-latency networks and storage devices. We provide full ACID guarantees and build on a logical timestamp mechanism to provide MVCC-based snapshot isolation, while not requiring synchronous updates of replicas. Instead, we use asynchronous update propagation guaranteeing consistency with timestamp validation.

We provide a view into the design and development of a large scale data management platform for real-time analytics, driven by the needs of modern enterprise customers.

1. INTRODUCTION

There are two fundamental paradigm shifts happening in enterprise data management. The first is a dramatic increase in the amount of data being produced and persisted by enterprises. Click data, ad views, sensor readings, stock price tickers, customer orders, company purchases, and financial transactions are just a few streams of data generated by enterprises from which business intelligence needs to be extracted. Driven by the potential business value hidden in these data sources and increasing high-speed storage capacities, enterprises are now storing generated data which in previous generations was considered transient and discarded after its purpose was served. The second shift is the *need* for businesses to have analytical access to up-to-date data in order to make critical business decisions. The legacy extract-transform-load pipelines for offline analytical processing of day-, week-, or even month-old data do not meet the demands of modern enterprises that want real-time insights from their data in order to make critical time-sensitive business decisions.

The combination of these two shifts in enterprise data usage creates a significant challenge for data management systems. On one hand, a system must provide on-line transaction processing (OLTP) support to have real-time changes to data reflected in queries. On the other, systems need to scale to very large data sizes and provide on-line analytical processing (OLAP) over these large and changing data sets. As the role of the modern "data scientist" solidifies its position as a core contributor to enterprise decision making, the demand for rich analytics on large scale data grows. In this context, the challenge is in how a system can provide real-time analytics at big data scale.

Various systems have been developed, both as academic prototypes and commercial products, to address various facets of this problem. There are high performance transaction processing architectures that can scale-out to very large database sizes and achieve impressive OLTP performance. There are also a number of systems that can provide high performance analytics at very large scale. However, the design of an integrated system that can provide large scale analytic processing power over real-time updating data remains a daunting challenge.

In this paper, we describe the architecture for a research prototype system called the SAP HANA Scale-out-extension (SOE), a holistically integrated data platform consisting of different functional components that provide the foundation of a novel data management system.

We advocate the separation of transaction processing from query processing and utilize a high-performance distributed shared log as the persistence backbone. We believe that such an architecture constitutes the basis for a robust and scalable data management platform that can tackle the evolving data management challenges we are currently facing. The resulting requirements include the following.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

- Mixed transactional and analytical workloads: The successful adoption of SAP HANA as an integration platform for traditional enterprise resource planning (ERP) as well as data warehousing has demonstrated the necessity of removing the traditional separation of transactional and analytical processing in database systems. We aim to continue this trend in a large scale-out architecture.

- Ability to take advantage of emerging hardware: SAP HANA is able to effectively leverage current hardware features such as high core count processors, SIMD instructions, large processor caches, and large memory capacities. With SAP HANA SOE, we aim to take advantage of future hardware developments such as storage class memories and high-bandwidth low-latency network interconnects.

- Elastic scale, both on-premise and in the cloud: Today's business challenges require an architecture that can adapt to large load fluctuations by dynamically provisioning resources. This adaptation may include increasing storage capacities, increasing compute capacity by balancing load across dynamically provisioned replicas, or co-locating data and services to reduce the required resources. In a multi-tenant deployment we need to be able to handle large tenants spanning multiple nodes in a scale-out fashion as well as many small tenants, which may share common resources and be placed on a single node.

- Rapid innovation cycles: Separating the functional components and establishing strict contracts between them has proven to significantly improve development performance in a large, globally distributed, research and development team. The approach has also shown to be extremely beneficial in introducing innovative aspects into a large system by decoupling development and release cycles for functional components as well as by providing alternative prototype implementations of certain functional aspects.

Core contributions

In this paper we provide a blueprint of the SAP HANA SOE data management platform. We present an end-to-end system overview, detail the different components and their interplay, and describe how they can provide:

- heterogeneous scale-out of OLTP and OLAP workloads independently within a single cluster;

- an approach to decoupling query processing from transaction management;

- the ability to improve performance by scheduling snapshots for read-only OLAP transactions according to fine-grained SLAs;

- a scalable distributed log providing durability, fault-tolerance, and asynchronous update dissemination to compute engines; and

- support for different compute engines as views over the same transactionally consistent data, e.g., SQL engines, R, Spark, graph, and text.

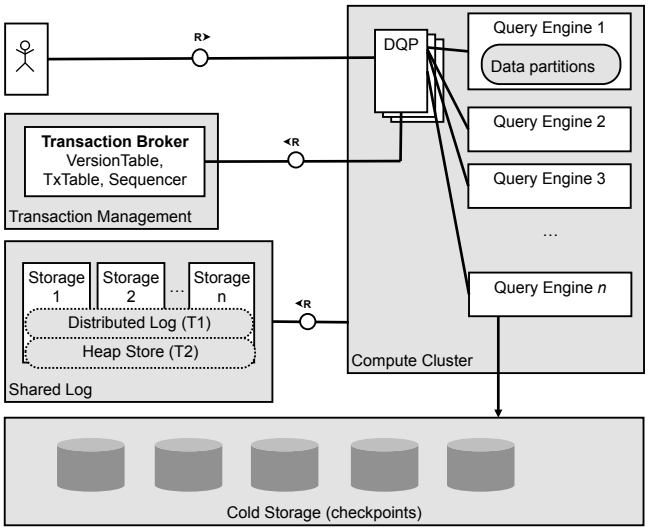

Figure 1: Logical Building Blocks of the SAP HANA SOE

We present sufficient design details on each of the major components in order to describe how they interact to provide scalable database functionality for mixed OLTP and OLAP workloads. However a more fine grained explanation and evaluation of each component is beyond the scope of this paper. We aim instead to present the end-to-end architecture, focussing on the distributed design and integration of the various components.

Structure of the paper

Section 2 outlines the different components of the SAP HANA SOE and sketches their interplay when handling read and write requests. We follow with a more detailed description of the following components: transaction management (Section 3), shared log (Section 4), and the HANA SOE query engine (Section 5). We close with a discussion of related work, summary and conclusion.

2. SAP HANA SOE BUILDING BLOCKS

The goal of the SAP HANA SOE architecture is to build a scalable data management platform for mixed OLTP/OLAP workloads without having to compromise on strict ACID guarantees. In this section, we outline the architecture and describe how the core components interact.

Figure 1 illustrates the architecture. Our design decouples the core database components, and provides them as services in a distributed landscape. The three core components are a distributed query execution cluster, a transaction broker, and a distributed shared log. An additional cold storage layer can also be used for storing checkpoints.

2.1 Components of SAP HANA SOE

The Query Engine: At the core of SAP HANA SOE is a high performance SQL-to-C code generation based query engine (HSQE) [9, 22]. HSQE was initially released as a stand alone query engine embedded in SAP Lumira, a market leading data visualization software solution, and has since been extended for scale-out [29]. SAP HANA SOE further provides an optimizer that decomposes SQL queries

into execution graphs suitable for running on a cluster of HSQE nodes. The optimizer runs on the DQP nodes. To aid in query processing, tables are horizontally partitioned into *slices* of relatively large but bounded sizes. Generally speaking, slices can be thought of as disjoint subsets of rows within logical partitions of the table, like a large physical page. Slices are large enough to fully exploit sequential in-memory processing on dictionary compressed data but small enough to enable efficient, elastic distribution and fast replication amongst HSQE nodes. Slices are directly embedded into HSQE: even a standalone deployment creates slices.

While the data in an HSQE node must be updated over time, HSQE nodes are not aware of, and do not participate directly in, database transactions. Conceptually HSQE nodes process read-only queries on stable database snapshots. An HSQE node may hold data for a multitude of snapshots concurrently. Versioning is typically done at the slice level to maximize OLAP performance but can be done at an arbitrarily fine granularity to reduce space requirements and to improve OLTP performance. In order to deal with this natural tension between OLAP and OLTP requirements, we propose to construct the database cluster as two distinct subsets of HSQE nodes: one each for handling OLAP and OLTP workloads. An OLAP node may then make completely different trade-offs than typically required from an OLTP node. This dichotomy of node types works well for real-world business scenarios where OLTP workloads typically deal with a relatively small amount of *hot operational* data, while the OLAP workloads want to process all of the data. Note that our architecture allows for the same data snapshot for a portion of operational data to be concurrently resident on multiple nodes.

Transaction Broker: Transaction processing is coordinated using a *transaction broker*. The transaction broker holds the shared state needed to process transactions. This information consists of transaction metadata, versioned write-sets for concurrency control, and a transaction sequencer. The system is structured to minimize the shared state required so that, in the common case the broker can be hosted on a single node. However, we allow for both high availability and scale-out for the transaction broker. Furthermore, the state held is such that the consequence of a catastrophic broker failure is at most the abort of all currently running transactions.

Distributed Log: Transaction durability is achieved with a separate distributed shared log built out of a dedicated cluster of servers called *storage units*. The transaction executors write transaction effects into this log: transaction commit occurs when the corresponding commit log entry is made durable. HSQE nodes can always obtain a particular database snapshot by reading from the log. For performance, mechanisms are provided to avoid having to do this continually in the common case.

2.2 Interplay of SAP HANA SOE components

In our system, we differentiate between read-only and read-write transaction requests. When a read request enters the system, the transaction broker issues a read timestamp and forwards the query to the HSQE cluster responsible for the corresponding database. At the executor, if the data slices needed to process the query are not up-to-date with the requested version, the executor consults the log for outstanding updates with respect to the affected data slices. After rolling the slices forward to the required snapshot, the query can be executed, the partial result sets combined, and the final result delivered to the application. Since the execution of a read query may possibly result in reading from the log and compiling the changes into the local database state, engines may instead subscribe to changes at the log and proactively apply changes to the local database without having a specific pull step as part of query execution. More generally, our architecture provides the freedom to balance the local update behavior between eager (push semantics) and lazy updates (pull semantics), depending on the workload. In general, this principle can be used to control the visibility of the most current data in a very fine-grained manner. For example, the same read timestamp for OLAP transactions (even though new transactions are committing) can be refreshed only every k seconds/minutes depending on a given SLA constraint, thereby amortizing the cost of reading from the log, the update effort of the query nodes, and the number of versions which have to be provided by the query engines. It is also worthwhile to note that there is no requirement for replicas of the same slice on different nodes to be synchronized to the same timestamp since any replica can bring itself up to date by reading the log.

Whenever a write request is received by the transaction broker, a read timestamp is generated and the query is also forwarded to the HANA SOE node cluster. At the specific nodes, the updates are executed in "read-only" mode, i.e. no database entities are modified; changes are only cached locally in order to speed-up the change propagation process for the next read request. The engine returns the set of row IDs that would have been affected by the update. At transaction completion time, the transaction broker aborts the transaction if it detects a conflict in the write set. Otherwise a commit timestamp, which corresponds to a sequence number in the log, is requested and the commit record is written to the corresponding log position. After successfully writing to the log, the transaction commits and the acknowledgement is sent to the client.

3. TRANSACTION MANAGEMENT

We propose a distributed transaction processing architecture designed to provide full ACID transactional guarantees without requiring invasive modifications to the core HSQE engine. Our architecture decouples transaction processing from the update mechanism used in the engine, and indeed from query processing, and provides an MVCC layer on top of the existing slice abstraction. This decoupling means that individual compiled query programs can run on a specified version of a specified slice, and need not have internal awareness of transactions or concurrency. Our design provides strong snapshot isolation [8], while internally using asynchronous propagation of changes to replicas, thereby avoiding the debilitating overhead of synchronous replication at large scale.

The HANA SOE architecture provides the following:

- Strong snapshot isolation: Our design enables strong snapshot isolation in a distributed setting. This is achieved by building an MVCC layer on top of a horizontal physical partitioning abstraction called "slices" using a logical timestamp mechanism.
- Independent OLAP scale-out: Our transaction processing architecture allows OLAP nodes to scale in-

dependently of the transactional workload by removing the obligation of transaction coordination from the query compute nodes. However, compute nodes still form transactionally consistent snapshots by reading updates from the log as needed, without having to participate in distributed commit protocols or having to maintain synchronicity of their hosted data slices with the corresponding updates.

- Efficient cross-partition transactions: Transactions atomically commit by writing their updates independently to the shared log. This design means that the overhead of coordinating transaction commit is independent of the number of nodes involved in processing the transaction.

3.1 Transaction Processing in HANA SOE

Our proposal emphasizes strict decoupling of the logical database components. In particular, the database log, the concurrency control component, the transaction manager and the query executors are considered independent components that coordinate using well-defined interfaces. As such, each component can have a number of implementation alternatives without compromising correctness, so long as the alternatives adhere to the design contract.

The system architecture is illustrated in Figure 1. There is a need for a system component to handle a client connection context and to execute transaction stored procedures or to invoke transaction functionality interactively on behalf of a client request. To simplify our discussion, we assume that this transaction coordination is co-located with the logical DQP coordinator component, which may run in-process with the database engine nodes. Our architecture, however, also permits transaction coordination to occur on the broker, in which case version table locality is gained at the expense of limiting concurrency to the capabilities of a single machine. We now describe the individual system components in detail.

3.2 The Transaction Broker

A centralized *Transaction Broker* serializes transaction order using a sequencer (atomic counter), maintains a transaction table of running transactions and their state, and hosts a version table. The version table is used by running transactions to publish their write-sets and to detect conflicts. The transaction broker presents a potential bottleneck for system throughput, as the rate at which transactions can commit is limited by the rate at which the sequencer can hand out sequence numbers. However, based on published literature (e.g., [2]) and our own early experimental results, we expect a TCP-based sequencer to scale to hundreds of thousands of requests per second, far beyond our practical target requirements for transaction processing throughput. As such, we consider the sequencer to be a theoretical, rather than practical, bottleneck for our requirements. The second point of contention in the transaction broker is the version table. For a high rate of concurrent access, we use a scalable concurrent hash-map that exploits multi-core parallelism.

The broker also maintains the timestamp of the last committed transaction, updated on each commit. This timestamp represents the current read timestamp for transactions needing fresh data (e.g., OLTP transactions) and is handed out to new transactions at start time. Additionally, the broker can maintain a current OLAP read timestamp which is incremented at epochs depending on a service-level agreement. This timestamp allows scheduling batches of OLAP queries at the same read snapshot, which amortizes log reads and large updates on OLAP nodes across all transactions reading at the same snapshot. We discuss scheduling snapshots for OLAP transactions in Section 3.6.

The transaction broker maintains no persistent state, instead all state that is needed for recovery is recorded in the log. On failure, any other node can be elected the new transaction broker. An epoch-based versioning scheme is used to ensure a "split-brain" scenario is not possible. All currently running read/write transactions are aborted (the new broker starts with an empty version and transaction table), and the sequencer can be recovered by querying the log to find the last written offset. The highest written sequencer value also represents the last committed transaction, and the current read timestamps will start at this same value. An in-memory high availability scheme for the transaction broker can be implemented to avoid the need for aborting currently running transactions, however this requires synchronous replication of the broker's state which has an associated performance penalty.

The broker is ideally hosted on a single machine. If the load can be handled by a single large node, there is no need to scale-out. It should also be noted that a distributed broker is not prohibited by our design. For example, when the data and workload can be totally partitioned we advocate simply creating a dedicated instance of the broker per logical partition. If the broker is partitioned to distribute load, then consensus among brokers for transactions spanning multiple broker partitions must be gained. A two-phase commit protocol can be used in this scenario.

3.3 The Commit Log

The second major component of the transaction processing architecture is a *distributed shared log*. All transactions commit their changes to the log and it is considered the one true copy of the database, providing the mechanism for durability, disaster recovery, and replication to compute nodes in the system. Details of the log can be found in Section 4.

3.4 Compute Node Contract

The final component of the architecture is the actual *compute nodes* used for processing queries. While our architecture permits a variety of possible compute node engines (e.g., SQL, R, graph, text), we optimize our design to support the HANA SOE SQL query engine as the compute nodes. Compute nodes conceptually monitor the tail of the log and incrementally incorporate updates affecting the table slices that they host. Each log record contains an entire transaction's worth of updates and compute nodes must maintain the ability to produce a new slice snapshot identified by a timestamp or a log sequence number (LSN). The broker references the slice snapshots on individual compute nodes to form transactionally consistent snapshots across the cluster. Our scheme requires that the query engine operates on stable row IDs and that these row IDs can be returned to the transaction broker along with needed values when it queries for write sets.

All queries sent to compute nodes must include a reference logical "as-of" timestamp (LSN). Compute nodes are required to answer the query at the given timestamp by executing the query against an appropriate snapshot, possibly

reading from the log first to bring their snapshots up to date. Note that there is no requirement for different replicas of the same slice to be synchronized to the same LSN since any replica can bring itself up to date by reading the log. The in-memory snapshots maintained by the compute nodes are considered ephemeral and can be rebuilt from the log in the event a node fails or a partition is migrated to a new node. The broker will periodically make available a low-watermark of the oldest transaction executing in the system which the compute nodes can use to discard old snapshots.

For a compute node to participate in the cluster, it has to adhere to a simple interface. To participate as a read-only engine (e.g., OLAP node), the node must:

- perform queries at a given timestamp; and
- atomically apply ordered and timestamped updates to construct new snapshots.

Note that the updates represent changes by already ordered, consistent, and committed transactions. No concept of a transaction or concurrency control is required at the compute node, only atomic application of the updates to the compute node's internal data representation.

To participate as a transaction processing engine (OLTP node), a compute node must:

- perform queries at a given timestamp;
- atomically apply ordered and timestamped updates to construct new snapshots; and
- process an update statement and return write-sets (IDs of affected rows);

An OLTP node may optionally cache the update result in its local memory, labelled with the transaction ID that issued the update, when processing an update statement. OLTP nodes will be given asynchronous notification of transaction commits, allowing them to simply make the cached updates visible without actually having to pull the updates from the log as is needed in the general transaction processing workflow (see Section 3.5). We discuss the internal engine differences for running an HSQE node as an OLAP or OLTP node in Section 5.

3.5 Transaction Execution

To make the abstract description in this section more concrete, we provide a walk through of the lifecycle of a transaction in our system. The request flow for general transaction execution proceeds as follows.

1. Transaction requests a read timestamp from the transaction broker. Broker records the transaction in the transaction table with state RUNNING.
2. Transaction sends queries and update statements to the query cluster.
3. If the compute node does not currently host the specified version of the specified data slice(s), it queries the log for outstanding updates to the affected slices. If the node has the required data updates in its cache, it can be read from local cache instead of querying the log.
4. Query cluster processes queries and update statements.
   (a) Compute nodes execute queries and return results.
   (b) Compute nodes execute updates in read-only mode (no rows are modified).
   (c) Compute nodes may cache the computed updates.
   (d) In the case of two-tier storage, if the update is large: compute node asynchronously writes the updates to the second tier storage, and informs the transaction executor of the *file identifier* on transaction completion (either via the compute node or directly as a remote callback from tier-two storage to the executor).
   (e) Compute cluster returns the set of row IDs that are to be written.
5. Transaction requests publishing write-set (row IDs) to the broker, which is successful if there are no conflicts. The transaction is aborted there are conflicts.
6. Transaction requests a commit timestamp by contacting the transaction broker sequencer and changes transaction state to PRE-COMMIT. Commit timestamps correspond one-to-one to log LSNs.
7. If using two-tier storage for large updates, wait for all writes to complete to obtain file handles. If using single-tier storage or the writes are small, the writes are in-lined in the tier-one commit record.
8. Transaction writes a commit record to the shared log at the LSN previously obtained.
   (a) If the write succeeds, contact the broker to publish commit timestamp and change transaction state to COMMITTED.
   (b) If the write fails contact the broker to change transaction state to ABORTED.
9. Send acknowledgement of commit/abort to the client.

3.5.1 Read-Only Transactions

Read-only transactions first contact the transaction broker to get a read timestamp. Queries are then sent directly to the compute nodes and can be fully handled without imposing any load on the broker, with the exception that a client should inform the broker when the transaction is complete so it can track that the snapshot used by the transaction is no longer needed. The execution flow is described as follows.

1. Request a read timestamp from the transaction broker.
2. Issue queries to compute nodes.
3. If the compute node does not currently host the specified version of the specified data slice(s), it queries the log for outstanding updates to the affected slices.
4. Inform broker on completion (COMMIT).

Note that many read-only transactions do not always require reading the most up-to-date data. We discuss scheduling OLAP transactions according to a configurable freshness SLA in Section 3.6.

3.5.2 Commit Path

Once transactions have successfully published their write-sets they are free to commit. Many transactions are allowed to commit in parallel so long as they don't have write/write conflicts. Our log design supports an efficient mechanism for concurrent writes near the tail of the log by reserving a log position at the same time the commit timestamp is assigned by the broker. It's possible that a client reading the log may encounter an unfilled entry in the log (due either to a failed or slow transaction coordinator). The client can choose to either wait for the log entry to be written, or fill the log entry with a junk value to force the transaction assigned that position to retry writing its commit log record at a new LSN obtained from the sequencer. To avoid starvation the transaction aborts itself after a limited number of retries.

3.6 Snapshot Scheduling for Analytics

Studying the execution flow described in Section 3.5, reveals two potential problems in scaling analytics. First, nodes must construct and host a snapshot for every timestamp version requested by any query running on that node. If a series of k analytics transactions all start around the same (wall-clock) time, but due to a continuous stream of committing OLTP transactions end up with distinct read time-stamps (e.g., 10, 11, 12, 13, ...), then the compute node would have to construct and maintain k different versions of its data to service these k different transactions. This problem is inherent of any mixed workload system providing snapshot isolation. The second problem is that as each request arrives at a compute node, it causes the node to pull the needed entries from the log. In the worst case scenario for the example described above, a compute node would request all records from its last update to time 10, then request log record 11, then request log record 12, and continue requesting individual log records as new queries arrive until the kth query arrives.

We can remedy both of these problems by scheduling analytics transactions to run at the same read timestamp while adhering to a user specified SLA. This is achieved by maintaining a separate read timestamp at the broker which is handed out to analytics transactions that specify they can tolerate stale data within the requirements of the SLA. Along with the timestamp, the wall-clock time representing the commit time of the transaction that wrote that version is stored. Once the timestamp is set to expire, it is then rolled forward to the last committed transaction timestamp already maintained by the broker. Note that any transaction that requires reading fresh data can always run at the last committed timestamp, though in many practical situations we find that analytics transactions can tolerate a certain amount of staleness. Scheduling analytics transactions at SLA defined epochs has significant benefits. First, it amortizes the cost of reading from the log and applying updates to in-memory structures over all transactions that read at the same timestamp. Second, it allows compute nodes to pull all needed updates from the log with a single scan operation, avoiding multiple network round trips. Third, it requires fewer versions to be kept in-memory by the compute nodes, making better use of resources.

4. SHARED LOG ARCHITECTURE

As introduced in Section 3.3, the second major component of HANA SOE is a *distributed shared log* to which all transactions commit their changes. The log conceptually functions as a key-metadata-value store: integer keys (log sequence numbers) are tagged with a variable number of metadata identifiers and mapped to arbitrary variable-length payloads. Each value is immutable once written.

The log constitutes a distributed system in its own right. It is partitioned for scalability, replicated for fault-tolerance and high-availability, and provides persistence of log entries by writing them to non-volatile *storage units*.

Our log's design provides several key features supporting its use as a database transaction log:

1. Each log entry may be written to an arbitrary number of logical streams within the log.

2. The log provides a total order over all writes to all streams, guaranteeing linearizeable operations.

3. The log interface includes a *scan* operation which performs bulk reads of log entries based on a specified predicate on the metadata.

4. A secondary unordered heap store is provided for large offline / asynchronous writes.

Although the distributed shared log was designed with the needs of the HANA SOE landscape in mind, its implementation makes few assumptions on the nature of the data stored within it. Rather, since the shared log is the single true copy of the database, each query processor effectively holds a materialized view of some subset of the log's entries. The log itself is agnostic as to the capabilities and internal organization of its clients.

From this perspective, it is straightforward to incorporate different query executors into a scale-out cluster simply by having them obtain data from the shared log. Moreover, those types of executors may be heterogeneous within the same cluster. For example, a particular cluster installation might include OLAP executors, graph/hierarchy engines, and text processors. The number of compute nodes dedicated to each of these functions may be freely adjusted, and all executors, regardless of their function, share access to the same underlying data by reading transactional updates from the log and applying them to their local data stores. The shared log abstraction thus provides the data backbone through which disparate data processing systems may share information and receive transactional updates.

Additionally, log clients need not be strictly transactional. Any application that requires strong consistency guarantees of its data updates may make effective use of the log abstraction we present. For example, it may be used to implement a pub/sub message bus in which metadata identifiers are used to tag message classes. The shared log simply provides a high-performance, fault-tolerant data store, which it exposes via an append-only interface that guarantees linearizeable updates. It is up to applications to decide how that data store should be employed and interpreted.

The rest of this section describes how each feature enables aspects of our transaction processing architecture.

4.1 Key Design Features

We start by outlining the key design features of the distributed shared log, followed by a description of the implementation decisions made in building this component.

4.1.1 Logical Streams

Recall that the HANA SOE query executors load and operate on small partitions of data called *slices*. Since every executor hosts only a fraction of the slices comprising the entire database, we can improve network utilization and reduce the time required to replay transactions by allowing each executor to only pull updates from the log that pertain to the slices they host. So as to not tie the slice mechanism directly to the log, the log operates instead on logical *streams*. Each log entry is tagged with stream identifiers indicating to which streams it belongs. The relationship between stream identifiers and higher-level constructs (e.g., slices) is left entirely up to the higher levels of the system.

The log allows a single transaction to be split across to an arbitrary number of streams. As such, it provides more flexibility than other stream-oriented designs such as Tango[3], which impose a fixed upper limit on the number of objects that can be involved in a transaction. This flexibility is necessary to permit a single transaction to modify data hosted on many different slices.

4.1.2 Totally-Ordered Writes

The log provides a total ordering over all writes through the use of write-once semantics and chain replication, similarly to the CORFU shared log[2]. In particular, the log provides three primary operations:

- $write(S, n, x)$, which writes a variable-sized payload x into log position n and annotates it with the set S of logical streams. If position n has already been written, *write* returns an error.
- $scan(S, n, m)$, which returns to the client all log entries at positions n through m which are tagged with at least one stream in the set S.
- $trim(n, m)$, which marks all log entries at positions n through m as "trimmed" and reclaims the disk space used to store those entries' payloads. The log positions themselves are not reclaimed.

By enforcing write-once semantics and by replicating log writes using the chain replication protocol [28], these operations are guaranteed to be linearizeable. The writes themselves are totally ordered by log position number.

In HANA SOE, each transaction corresponds to a unique log position. A transaction executor acquires a unique log position from a sequencer node within the cluster (similar to the sequencer used by CORFU to reduce contention between writers), then stores a payload at that log position representing the transaction being committed.

As with any position-preallocation scheme, this approach can result in log entries being written out of order, or never being written at all if clients crash. Such holes in the log may be filled by client-driven processes as in CORFU, or by server-driven means as described in Section 4.1.3.

4.1.3 Scan Operation

Unlike the CORFU design, our log permits variable-sized records and is deployed on servers with significant compute power, affording a more capable interface. This compute power enables our inclusion of the *scan* operation listed in Section 4.1.2, as the filtering of entries by stream identifier is performed locally on each storage unit. In the context of HANA SOE, the *scan* operation reduces network utilization in two ways:

1. By filtering based on stream identifiers, *scan* avoids sending log entries across the network which are irrelevant for the scanning client. This means, for example, that the entire log does not need to be shipped to all OLAP nodes.
2. Compared with a log interface that reads one log entry at a time, *scan* significantly reduces the number of network round trips required to update a HSQE node to a specific slice version.

The *scan* operation may also be used as an efficient hole-filling mechanism by immediately filling any unwritten log positions encountered during the scan. This strategy is quite aggressive, but for a configuration in which OLAP queries are scheduled to run at timestamps somewhat behind the tail of the log, filling holes in this way is unlikely to cause transactions to abort; rather it is likely to fill holes caused by clients which have actually crashed or otherwise become disconnected from the system.

4.1.4 Secondary Heap Store

In the proposed design, each database transaction is stored in the log as a single entry. Large transactions, therefore, will entail large (hence slow) network transfers and disk writes. To move these operations off the critical commit path, we introduce a secondary storage area called the *heap store*. When committing a large transaction, the transaction processor may use the heap store as follows:

1. Write the large log entry to the heap store without synchronization.
2. Acquire the mutual exclusion necessary to commit the transaction.
3. Write into the log a small commit entry containing a reference to the large entry in the heap store.

If the transaction aborts after the heap store has been written, the disk space used to store the large log entry may be reclaimed. In this way, the duration for which mutual exclusion is required for transaction commits is minimized.

The heap store may be implemented as a separate storage unit within the storage cluster, but it is also possible to overlay secondary "heap writes" within the existing storage unit design. One may define a maximum log entry size for which writes to the storage cluster have acceptable latency. If a transaction payload exceeds that size, it is broken into smaller chunks, which are each written to the log individually. Once all the pieces of a large write have been stored successfully, a small "heap commit" record is appended to the log describing the layout and log positions of the entries comprising the single large transaction record. When a scan operation encounters any of the chunks individually, it skips over them; when it encounters the heap commit record, it forms and transmits the entire transaction payload.

4.2 Implementation

4.2.1 Partitioning and Replication

The log is implemented as a cluster of *storage units* over which its entries are partitioned and replicated. Each unit

hosts a portion of the log, for which it provides the full log interface. We are evaluating two alternative approaches to cluster organization:

1. **Shared Map**
   In this approach, all log clients share a mapping of the log positions to the storage units that hold replicas of entries belonging to those positions. Log clients must contact specific storage units in the appropriate replica set to initiate a log operation, and replication may be driven by either server- or client-side logic. This approach requires a minimal amount of coordination amongst storage units, but also requires clients to move consistently from one version of the shared mapping to the next, potentially causing short delays when the mapping changes.

2. **Distributed Hash Table**
   This variant organizes the storage units in a Chord-style ring [24]. Log clients may contact any unit within the ring to initiate a log operation; if that unit is not hosting the log entry in question, the request is forwarded along the ring. To improve operation routing efficiency, each unit maintains a finger table indexing all other ring members. This design avoids sharing state between clients and storage units, but when units join or leave the cluster, the finger tables temporarily degenerate and extra hops may be required to service log requests.

For replication, both implementation variants employ the chain replication protocol. To improve read throughput, this can easily be extended to the CRAQ protocol [25].

4.2.2 Storage Units

The storage units themselves maintain metadata about each log position they host, including the blocks on disk where the log record is stored (represented as a sequence of ⟨offset, block-count⟩ pairs) and the streams applicable to each entry. All changes to this metadata are logged to an internal append-only journal to ensure durability.

The storage unit implementation takes advantage of the parallelism offered by modern SSD devices. I/O operations are performed asynchronously, and many operations are kept simultaneously in flight so that the persistence device is consistently saturated. However, the log design is modular in the sense that any storage unit implementation supporting our API may be used within the storage cluster. We may, therefore, build storage units optimized for forthcoming storage class non-volatile memories (NVM) and deploy them alongside existing SSD-backed storage units. These units may co-exist within the same cluster; for example, NVM-backed units might be deployed on-demand to host entries at the tail of the log, which experiences high read and write load, while SSD-backed units might be used to host older compacted log fragments with lighter read load and no writes.

To improve read throughput the storage units will maintain an in-memory log entry cache. We furthermore intend to expose the entire address space of this cache for RDMA transfer so that all log entries required for a scan operation may be read by a client in a single scatter-gather operation. Such reads avoid context switches and memory copies on the storage unit, which will free its compute resources for more complex operations such as log compaction.

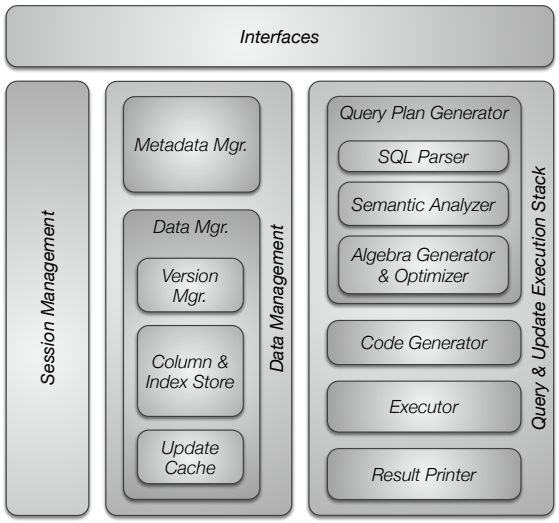

Figure 2: Architectural Components of a HANA SOE Query Engine (Compute Node) instance

4.2.3 Log Compaction

The transactional payloads themselves may be seen as delta updates to a key-value store in which the keys are database row/column identifiers and the values are the data written to the database. Log entries pertaining to the same key in this sense may therefore be compacted simply by keeping only the most recent log entry. This observation may be leveraged to implement log checkpoints by compacting all log entries prior to a particular log position p, then trimming the log to remove all entries prior to p. The compacted entries may be organized by stream identifiers to simplify shipping to clients.

The log may also be compacted on the fly to reduce network utilization. Rather than shipping to clients multiple log entries pertaining to the same database "key", we plan to compact such entries at scan time and only ship the most recent value of each database location touched by transactions within the scan range. If log scan ranges are well-defined and shared between executors, it is reasonable to cache the compacted log entries so that compaction is not repeated for multiple clients.

5. HANA SOE QUERY ENGINE

The third major component of our transaction processing architecture is a cluster of HSQE nodes ("compute nodes") that implements a robust distributed query processor (DQP). The HSQE cluster implements a DQP service managing the mapping from horizontal table partitions ("slices") to compute nodes. For an incoming query, the DQP service generates and globally optimizes a distributed execution plan that is specifically tailored for execution across the compute nodes in combination with efficient communication algorithms [30]. As outlined in Section 3.4, compute nodes are furthermore responsible to monitor the tail of the log and to apply ordered and timestamped updates to construct new versions, based on which queries are executed. In this section, we take a look at the internals of the HANA SOE query engine.

5.1 HANA SOE Query Engine Internals

Figure 2 shows the architecture of a single HANA SOE query engine instance. It resembles a classical in-memory DBMS consisting of an *interface layer*, a *session manager* (keeping track of client interactions), a *data manager* with in-memory data store, and a *query and update execution stack*. As persistence and concurrency control are managed by the shared log and the transaction broker, we find neither functionality here. Furthermore, the update execution stack is greatly simplified since it is only responsible for version propagation for data slices based on ordered committed transactions from the log.

At its heart, the SAP HANA SOE query engine sticks to the HANA storage model paradigm which is based on a main memory column store but uses specialized data structures suitable for SQL-to-C code generation paradigm. When a compute node starts up, it loads its assigned column slices as the *base version* from a checkpoint on the cold store (see Figure 1). As outlined earlier, queries specify an LSN to indicate which version they need to be executed on. The *version manager* is responsible to create and keep track of these versions. It makes use of the *update cache* to receive data from the shared log, if the necessary log entries to construct a new version (e.g., from the base version) are not locally available (see Section 5.3).

In our current design, any compute node can provide the DQP service for an incoming request. Therefore, as shown in Figure 2, the query engine comes with a full-fledged query execution stack comprising of parser, semantic analyzer, algebra generator and optimizer. After optimization and algorithm selection, the engine compiles the physical query plan into C code and translates it into an executable binary format. As Dees and Sanders described in [9] there are significant performance advantages with this approach. The compiler framework LLVM with Clang does the compilation from C into native code. A similar approach is followed in [19], however, we generate C instead of LLVM byte code to support more sophisticated maintenance and debugging.

As outlined in [9], the generated C code aims to minimize data transfer between the CPU caches and main memory by applying as many operations (join, filter, project, aggregate, expression evaluation, post process) on some cached data item as possible. The generated code returns a very compact result representation (in the simplest case, just row IDs are generated) that allows late expression evaluation and result materialization.

In case of pipeline breakers or intermediate result sharing, multiple code units can be emitted by the code generator. The executor collects the (versioned) input column slices, triggers the computation of join indexes and schedules the compiled code units for execution. The executor exploits data parallelism by following a `parallel-reduce` style execution. The *result printer* finally receives the compact result representation and materializes the final result using late materialization.

A first version of the SAP HANA SOE query engine was delivered to customers at the end of 2014 as part of SAP's analytical visualization software solution (SAP Lumira).

5.2 Query and Update Processing

Insert, update, and delete statements are also issued from the client to the DQP service. The DQP engine locates the affected slices and the optimal set of compute nodes hosting these slices, where the log entries may have to be generated before statement execution. The general control flow is described in Section 3.5, here we focus on the compute node internals.

On each involved compute node, update data is generated in a two-phase process using the compute node's code generation framework. In the first phase, code is generated to compute the set of affected row IDs (write set) based on the statement's `WHERE` clause and all update expressions with variable-sized result data. Here, we exploit data parallelism on the input table. Once the write set is validated by the transaction broker, the compute node allocates a memory buffer to hold a log entry. Using information collected in the first phase we know the final size of the update data and can avoid re-allocation. In the second phase, the node computes the fixed-sized update expressions based on the row IDs from the write set and writes the result into the log entry. Here we exploit data parallelism on the range of row IDs from the write set: we can compute the prefix-sum on the size of row-ID-based ranges to fix the write pointers into the log entry for each such range. If the update is large, the compute node writes its log entry to the log's secondary (heap) store and forwards a pointer of the heap location to the transaction broker for inclusion in the payload of the commit log record for the corresponding transaction.

5.3 Version Management

Applying updates in the core engine differs depending on the HSQE node's role as described in Section 3. In case of OLAP query processing, log records stored in the distributed log are consumed by OLAP nodes in either a push or pull fashion. In the push approach, a node is continuously informed about updates corresponding to a specified set of slices by the distributed log using a publish/subscribe-like mechanism. Records are pulled from the distributed log if a specified version of the affected set of slices needs to be materialized due to query requirements. Log record shipping itself is done in a compacted fashion. This means, log records are grouped by their type (update/insert/delete), the affected slice, and the affected columns within the slice.

In our current implementation, a newer specified slice version is created by copying the latest available version of this slice and by applying the log records in order. Since this is a costly operation, we materialize lazily where log records are first stored in an update cache. The update cache consists of two arrays of equal and configurable size to provide double buffering. When the currently active array fills up, it is flushed out by merging its contents into newer versions of affected slices. New slice versions are created in parallel. Since each log position always contains complete transactions, newly materialized slice versions are always transactionally consistent. While one of the arrays is being flushed out, the second array allows for appending additional compacted log records without blocking.

For future work, we plan to extend the version manager implementation. If the number of changes is relatively small, hashing or REDO lists per tuple could be used. Following the first mentioned variant, new row data is stored in a hash map by using the row ID as the key. Following the latter variant, an additional hidden column is added to the base slice storing new row data versions as a linked list. When the number of updates per row increases and version retrieving gets more expensive, a new slice version is materialized.

In order to process OLTP queries, we have implemented a basic strategy using the update cache described above. However, besides persisting compacted log records in the distributed log, they are written to the cache in order to consume them directly, avoiding the detour of obtaining them from the log. In the case a transaction wants to append its updates to the buffer, it first gets validated by the transaction broker for conflict detection and is issued a log sequence number for commit. This number describes the offset within the global log and is mapped to a position in the update buffer. This allows for parallel writing. Since copying slices in order to apply single updates is very expensive, we are also investigating a more sophisticated strategy described in [20] where an additional hidden column is added to the base slice which stores UNDO records as linked list on a per-row/tuple-level. Updates are applied in place. If a reader is not allowed to see the current state of a tuple, an appropriate older version is simply created by traversing the hidden linked list of this tuple and applying the UNDO records.

5.4 Checkpointing

A periodic checkpoint operation can build a new stable version of a table slice by writing a consistent snapshot to cold storage. The checkpoint happens per slice for a specific version and can be handled entirely by the query engine without the need to involve the transaction broker or the shared log. Furthermore, the checkpoint may take place on any node in the system and need not interfere with query execution while the checkpoint is in progress. These checkpoints are used to improve restart times and initialize new replicas. A persistent directory is required to store the location of latest version of every slice.

6. RELATED WORK

In this section we discuss related work in the areas covering the various components of our system.

6.1 Shared Log

CORFU [2] is designed to use simple network-attached flash drives and as such puts as much work onto the clients as possible. Log offsets are coordinated by a centralized sequencer which hands out sequence numbers to clients independent of the actual log writes. Clients map log positions to physical flash units via a replicated *projection* data structure. Replication is done using a client-driven version of chain replication [28]. To deal with holes in the log caused by the sequencer, readers may execute a special *fill* operation to write a known "junk" value to a log record. As in our system, flash units in CORFU employ write-once semantics so either the filler or the writer will win the race and progress can be made. Tango [3] builds on CORFU by adding support for multiple streams within the log and demonstrating how to build transactionally consistent replicated data structures over a shared log. The stream implementation in Tango requires reserving space in every log record for every possible stream supported by the runtime. This makes it unfeasible to support a very large number of streams as is required by our system in order to map slices to streams.

Kafka [15] provides a service which hosts many independent log streams called *Topics*. Topics are partitioned according to application-defined logical criteria. A process called a *broker* holds a number of partitions, possibly from different topics. Parallelism is achieved by having producers write to different partitions and allowing one reader from each consumer-group to read from a partition. Ordering is not guaranteed between partitions of a topic, only within a partition itself. This makes Kafka unsuitable as a database commit log that requires write parallelism.

BookKeeper [11] provides a service which hosts a number of single-writer, append-only logs called *ledgers*. Servers called *Bookies* hosts ledger fragments from a number of different tenants. Writes to a ledger are replicated to a quorum of a client-configurable number of Bookies called an *ensemble*. Under normal operation, a writer closes a ledger (indicating that no further modifications are allowed) before clients are allowed to read from it. In a properly closed ledger, all writes are fully replicated and a client can read for any replica in the quorum that hosts it. This mode is suitable to host e.g. a journal for the HDFS name node which was the original motivation for BookKeeper. It is also possible to read from an open ledger, but doing so requires readers to contact every node in the ensemble to determine the last confirmed write for each bookie and to not issue reads beyond this limit. The single-writer nature and the performance implications of reading from an open ledger make BookKeeper unsuitable as a logging service for our system.

6.2 Decoupled Transaction Processing Systems

Deuteronomy [16, 17] decomposes the database kernel into a transactional component (TC) responsible for concurrency control and recovery and a data component (DC) responsible for supplying access methods and caching. The transaction component operates only on logical entities and relies on the data component to supply physical identifiers to facilitate operations such as locking. A significant source of complexity in this work is ensuring that the TC and DC have a consistent view of the distributed state. The MVCC-based approach in our system where log LSNs double as snapshot versions greatly simplifies the problem by providing a common way for the transaction broker and the query engines to refer to snapshots.

Hyder [5] also separates transaction processing from data storage by having multiple database engines share a common log, as is done in our architecture. Hyder however avoids all (non log) coordination between engines by having each transaction write its intentions into the log followed by each engine deterministically replaying the log to determine commit/abort decisions. This is in contrast to our explicit coordination using a transaction broker, where only committed transactions are written to the log. Our architecture is also designed for facilitating transactions over large partitioned databases, while Hyder fully replicates the log to all database engines.

Calvin [26] implements a single version transaction layer on top of any existing datastore providing a reliable CRUD interface. Executors in Calvin collect transactions into batches and then agree on a deterministic ordering that minimizes contention and maximizes throughput. Once the contents of the batches are agreed upon, they are sent to all nodes where they are executed in a deterministic fashion. This is similar to the approach taken in our system where a transaction's effects are written to the shared log and incorporated without coordination at the query engines. The main difference between the Calvin approach and ours is that Calvin only provides single-version, serializable iso-

lation and cannot support long-running transactions. This makes Calvin unsuitable as a mixed OLTP/OLAP system.

DB2 pureScale [4] runs a number of database servers in an active-active mode over a shared storage architecture. Access to storage and locking is provided through a centralized *cluster caching facility* node accessed over RDMA. We take a similar approach to centralizing resources to avoid contention, but our architecture does not require a general shared-storage pool and is designed to scale beyond the 128-node limit of pureScale.

F1 [23] provides a transaction processing SQL layer on top of Spanner [7], a transactional key/value store. A two-phase commit is used to coordinate cross-partition transactions.

6.3 Scale-out OLTP Systems

Traditional OLTP systems are very difficult to scale out as transactions which cross machine boundaries require commit-time coordination (via algorithms like 2PC) to ensure that all executors agree on the commit/abort decision. 2PC-scalability is known to be poor beyond a handful of nodes [1, 7]. The number of machines involved in a transaction grows with the product of the number of partitions touched by the transaction and the number of replicas.

Like Calvin, H-Store [12] preprocesses transactions to force a deterministic ordering and remove any non-determinism from the transaction itself. In this way, a transaction can be sent to multiple replicas in parallel without the need for coordination at commit time. This strategy performs well when the database may be partitioned such that few cross-partition transactions are executed. Unfortunately, this is not always possible. Additionally, long running transactions are disastrous to H-Store: it cannot execute transactions outside of the predetermined order, so it must wait for the long running ones to finish before others can continue. This makes H-store unable to also support OLAP workloads.

6.4 Mixed OLTP/OLAP

A number of SQL over Hadoop systems such as Hive [27], Impala [14] and Shark [31] have recently shown promise in the area of big-data analytics, but are generally unable to handle OLTP workloads.

SAP HANA [10, 21] already excels at handling mixed OLTP/OLAP workloads in a single scale-up system or in scale-out clusters of a small number of nodes. SAP HANA SOE provides the extensions required to allow HANA to scale out to thousands of nodes.

In HyPer [13], a master in-memory database processes OLTP transactions. Periodically, a snapshot of the processes memory is copied to a new process via the use of fork() and brought back to a consistent state by applying undo records to rows modified by in-progress transactions. OLAP queries are run against these forked snapshots. The ScyPer [18] system extends this idea to a distributed system by using reliable multicast to broadcast all committed updates to all replicas. The primary system continues to act as a master for OLTP transaction, while the replicas are used to run OLAP style read-only queries. ScyPer is less fault-tolerant and less scalable than our system, as it uses PGM-multicast instead of a distributed log and requires that the entire database fit in the memory of a single OLTP master machine.

In ConfluxDB [6] a primary, update-accepting database is partitioned for scale-out across a primary cluster. Within this cluster, cross-partition transactions are coordinated using 2PC in the usual fashion. Logs from the individual executors in the primary are merged together to form a totally ordered log stream which is replicated to a number of secondary, read-only clusters. A relaxed form of snapshot isolation (global-SI) is used across the cluster to identify consistent snapshots. Transaction coordination is only dependent on the number of nodes in the primary cluster, so OLTP and OLAP can be individually scaled by running OLTP workloads on the primary cluster and OLAP queries on the secondary. The distributed shared log in our approach eliminates the need for a log-merging step and also acts as a buffer of the totally ordered log stream to allow read-only replicas to join and leave the cluster, simplifying OLAP scale-out.

7. SUMMARY

This paper has presented the core components of the SAP HANA SOE architecture and described how the pieces fit together to provide a scalable snapshot isolation-based database that decouples OLTP and OLAP processing yet provides support for strict freshness SLAs. The design fundamentally distinguishes and decouples three types of services. The transaction broker handles concurrency control, allowing individual query engines to run without explicit knowledge of transactions or concurrent writers. A distributed shared log allows us to scale write throughput for committing transactions and provides a high throughput scan interface to allow scaling out query engines while keeping them up-to-date. The shared log exposes multiple logical log streams and can be used to model sequences of changes to particular partitions, allowing query engines to query only the changes that affect their hosted data. Overall, this design allows scale-out of mixed OLTP/OLAP workloads with strict freshness guarantees for analytics.

Acknowledgements

We would like to thank the HANA Research and Development team for their incredible ongoing efforts. We would also like to thank Adrian Nicoara for valuable discussions and contributions during his internship.

8. REFERENCES

[1] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, 2014.

[2] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10, 2013.

[3] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[4] V. Barshai, Y. Chan, H. Lu, and S. Sohal. *Delivering Continuity and Extreme Capacity with the IBM DB2 pureScale Feature*, chapter 1. IBM Redbooks, 2012.

[5] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, volume 11, pages 9–20, 2011.

[6] P. Chairunnanda, K. Daudjee, and M. T. Ozsu. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *Proceedings of the VLDB Endowment*, 7(11), 2014.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[8] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 715–726. VLDB Endowment, 2006.

[9] J. Dees and P. Sanders. Efficient many-core query execution in main memory column-stores. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 350–361. IEEE Computer Society, 2013.

[10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[11] F. P. Junqueira, I. Kelly, and B. Reed. Durability with BookKeeper. *ACM SIGOPS Operating Systems Review*, 47(1):9–15, 2013.

[12] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[13] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[14] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, and D. Hecht. Impala: A modern, open-source sql engine for hadoop. In *Proc. CIDR*, volume 15, 2015.

[15] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[16] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, volume 11, pages 123–133, 2011.

[17] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[18] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: Elastic OLAP throughput on transactional data. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, pages 11–15. ACM, 2013.

[19] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[20] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of SIGMOD*, 2015. to appear.

[21] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of SIGMOD*, pages 1–2, 2009.

[22] H. Plattner, F. Färber, J. Dees, M. Weidner, S. Baeuerle, and W. Lehner. Towards a web-scale data management ecosystem demonstrated by SAP HANA. In *IEEE 31st International Conference on Data Engineering, To Appear*, 2015.

[23] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[25] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*. San Diego, CA, 2009.

[26] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[28] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[29] M. Weidner, J. Dees, and P. Sanders. Fast OLAP query execution in main memory on large data in a cluster. In X. Hu, T. Y. Lin, V. Raghavan, B. W. Wah, R. A. Baeza-Yates, G. Fox, C. Shahabi, M. Smith, Q. Yang, R. Ghani, W. Fan, R. Lempel, and R. Nambiar, editors, *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 518–524. IEEE, 2013.

[30] M. Weidner, J. Dees, and P. Sanders. Fast olap query execution in main memory on large data in a cluster. In *IEEE Big Data*, pages 518–524, Oct 2013.

[31] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.