An In-Memory Database System for Multi-Tenant Applications

Franz Färber, Christian Mathis, Daniel Duane Culp, and Wolfram Kleis
{franz.faerber|christian.mathis|daniel.duane.culp|w.kleis}@sap.com
SAP AG
Walldorf, Germany

Jan Schaffner
jan.schaffner@hpi.uni-potsdam.de
Hasso-Plattner-Institute
Potsdam, Germany

Abstract: Software as a service (SaaS) has become a viable alternative for small and mid-size companies to outsource IT for a reduction of their total cost of IT ownership. SaaS providers need to be competitive w. r. t. total cost of ownership. Therefore, they typically consolidate multiple customer systems to share resources at all possible levels: hardware, software, data, and personnel. This kind of resource sharing is known as *multitenancy*. In this paper, we will focus on the aspect of multitenancy regarding our new in-memory database system – the SAP in-memory computing engine. In particular, we will motivate the requirements of multitenant applications towards the database engine and we will highlight major design decisions made to support multitenancy natively in the in-memory computing engine.

1 Introduction

Dropping DRAM prices and exponentially growing DRAM volumes changed the way we think about data management. Nowadays, a single blade can hold up to 2 TB of main memory [HPP10]. Database systems running on a cluster with 25 of these blades can store the worlds largest companies' business data, e. g., from enterprise resource planning (ERP) systems, purely in main memory. Why is this observation so important? Because of performance.

The architecture of all of the relevant database systems for enterprise applications is designed to cope with the well-known gap between main memory and external memory (see Table 1). The exchange unit for data transport between external memory and main memory is a *page* containing a number of bytes. To optimize for performance, database-internal data structures (e.g., the B-tree) and query processing algorithms are tailored to paged data access. Furthermore, advanced page-based buffer management and pre-fetching techniques have been developed to shadow the access gap.

Table 1: Access time from main memory and disk []

Action	Time [ns]
Main memory reference	100
Read 1 MB sequentially from main memory	250,000
Disk seek	10,000,000
Read 1 MB sequentially from disk	30,000,000

In-memory database management systems (DBMSs) keep the operational copy of the data entirely in main memory. They only need to access external memory in three particular cases: 1. During system startup to load the main-memory copy, 2. for logging, writing a checkpoint, and recovery, and 3. to persist meta-data and configuration changes. An in-memory DBMS can choose to rely on paged data handling (e. g., paged buffering) for these tasks. All other operations run purely in main memory. For them, the in-memory DBMSs can avoid paged data handling and encode data in contiguous memory DRAM areas (i. e., arrays). Storage structures and query processing algorithms can be tailored to work well with arrays instead of pages. This distinguishes them from a classical disk-based system equipped with a large buffer space. Such a system is entirely page-oriented and, compared to in-memory DBMSs, suffers from the organizational overhead imposed by the page-centric data handling [GMS92].

Besides RAM developments, the multi-core trend also has a substantial impact on data processing systems. Because the "frequency rally" stopped in 2006, software vendors like SAP now cannot rely on frequency-based performance speed-up anymore. Instead, we have to find ways to parallelize our software to – ideally – scale linearly with the number of available cores. Current hardware architectures have up to 64 cores (on eight sockets) per blade. All 64 cores have shared access to the main memory. We again mention that these blades can be switched together in a cluster, resulting in a highly parallel system with local shared-memory access [HPP10].

The *SAP in-memory computing engine* (IMCE) is an in-memory DBMS designed for a multi-blade, multi-core hardware architecture. It is a relational database system with SQL and full ACID support. Apart from providing standard database functionalities, the IMCE is also aimed at a sound integration with SAP business applications. Certain requirements of enterprise applications have influenced the design and functionality of our in-memory database system. A particular requirement, which we like to showcase in this paper, is *multitenancy* – the ability to handle multiple clients or *tenants* within the IMCE.

The remainder of this paper is organized as follows. In the next section, we discuss the concept of multitenancy. Then, in Section 3, we give an overview over the key features of the SAP in-memory computing engine. Section 4 lists the requirements posed by multitenant applications towards the data-management layer. Section 5 highlights the implications of multitenancy support in our new database system, before Section 6 concludes the paper.

2 Multitenancy

The classic *on-premise* software distribution model for enterprise software (e. g., ERP systems) involves software licensing by a customer, software installation on his hardware, and software maintenance by his personnel. Especially smaller and mid-size companies can easily get overburdened or even distracted from their core competencies by the task to run their own IT. Although running enterprise software is mission critical to these companies, the need for cost and complexity reduction makes outsourcing IT by obtaining *Software as a Service* (SaaS) viable.

SaaS solutions are completely hosted by a service provider, i. e., the software and the data resides on the provider's data centers. Thus, service providers operate the companies' IT system off-premise. Compared to on-premise solutions, they can thereby alleviate companies from the risk of over-provisioning/under-provisioning and under-utilization/saturation by "elastically" adding and removing resources. Companies, on the other side, only pay for the service, either by subscription or on a pay-per-use basis.

The reason why all this works for the SaaS provider is *resource sharing* among customers at all levels to reduce total cost of ownership (TCO): hardware, software, data, and personnel. In a SaaS system, a customer is called *tenant*. Several tenants share the hardware of a machine, where the mapping between a tenant and its hardware is kept flexible. If a tenant grows, e. g., because of growing data volumes or because of a growing user base, the service provider can decide to migrate it on a larger machine (*scale up*) or to dedicate more machines to the tenant (*scale out*). Also, if hardware capabilities grow, the service provider can decide to put more tenants on a single machine.

Tenants can also share the same software. This is especially true for standard enterprise software as developed by SAP, but also for generic components, like database systems, application servers, and repositories. Furthermore, because the provider controls both, hardware and software, he is not obliged to port the software to all commercially relevant hardware platforms. Rather, he is able to specially optimize for the hardware of his choice.

Applications running on behalf of multiple tenants can also share (general or public) data, for example country-specific information like population, currency, exchange rate, and gross national product (GNP), e. g., for analytics. Furthermore, multitenant applications can share meta data. Finally, the personnel operating the data centers can share a single administration framework to unify software and hardware administration for all tenants.

As already stated, SaaS is not only about software. Along with the software, the data goes to the SaaS provider as well. The question is, how do SaaS providers manage data from multi-tenant applications. Of course, the solution lies in the realm of database systems. We see three alternatives how to manage data from multiple tenants with the help of database systems [JA07]:

- *Shared machine*: The DBMSs running on behalf several tenants share the resources of a single machine. The advantage of this approach is a good degree of isolation between tenants and simple tenant migration. Especially when the DBMS is installed in a virtualized environment (e. g., XEN, VMware, Amazon EC2), DBMS migration is as simple as virtual machine migration. Furthermore, standard database systems can be used to realize the shared machine approach; no tenant-specific functionality is required. Due to the maximum degree of isolation, every tenant can extend or modify its database schema without restrictions (which is important for tenant-specific customizations). When a DBMS instance crashes, no other tenants are affected. On the flip side, the various DBMSs installed on a single machine compete for resources, like memory pools, communication sockets, or execution threads. Managing these resources generates overhead in each running system. Additionally, there is no centralized control to govern shared resources (except for the operating system, which is, however, not aware of the specific resource consumption patterns of multitenant database systems). Finally, administering one DBMSs instance per tenant without additional (external) tools can become a cumbersome task when a large number of tenants has to be supported.

- *Shared table*: All tenants share the same DBMS instance, the same schema, and store their data in the same tables. To distinguish tenants from each other, a special column containing a tenant identifier is inserted in each table. Again, standard database systems can be used to implement this approach. Compared to shared machine, system resources are shared/controlled, and maintenance applies to only one instance. However, tenant migration becomes more complex, because tables and disk volumes are not isolated. In the case of a crash, all tenants are affected. Furthermore, database queries have to be rewritten to restrict results to the queried tenant. Finally, tenant-specific extensions always influence other tenants.
- *Shared DBMS instance*: All tenants share the same DBMS instance. To isolate the tenants, each tenant runs in a separate process, has its own tables, and its own external-memory volumes. However, resources between tenants, e. g., memory pools, communication sockets, and execution threads can be shared (by a pooling infrastructure that is common for all tenants). Sharing of meta data, i. e., the database schema, is also possible. Clearly, this approach requires that the DBMS is made aware of multiple tenants. However, it combines the best of both previous approaches: Tenants are isolated, maintenance applies to one instance only, resources are shared/controlled, migration can be implemented based on tenant-specific disk volumes, (meta-)data can be shared, a crashing tenant process does not tear down other tenants, and tenant-specific extensions do not influence other tenants.

Because the third approach seems most reasonable, the SAP in-memory computing engine supports multitenancy by sharing the DBMS instance. We will return to the details in Section 5. First, we would like to give an overview over the features of our in-memory database system.

3 The SAP In-Memory Computing Engine

The SAP in-memory computing engine is a relational database system that keeps the primary data copy in main memory. The target hardware consists of a cluster of blades, where each blade is typically equipped with up to 64 cores per blade and up to 2 TB main memory (at the time of writing). Of course, to avoid data loss during power failures, the system needs a persistence on external storage as well as logging and recovery mechanisms (see Section 3.8). However, the principal idea is to optimize the database system towards main-memory access and keep the access frequency to external memory as low as possible. To an end user, the system provides the usual functionality one might expect from a database system: it supports the relational model, has a SQL interface, backup and recovery, and full ACID support. In the following, we will discuss some key concepts of the in-memory computing engine.

3.1 Row Storage and Column Storage

Tables in the in-memory computing engine can be stored column-wise or row-wise. The storage mode can be defined by the user. Both storage types have their advantages regarding access behavior. The column store supports set-based read-intensive data processing along columns, for example, aggregation queries in online analytical processing (OLAP). The row store naturally supports row-based and update-intensive operations, like single key lookup or single row insertions. It serves best performance in applications with online transaction processing (OTLP) characteristics. The storage mode is transparent to the query engine, i. e., queries can freely combine data stored in both table formats. Tables can also be converted on the fly. We describe the internal storage layouts for row and column store below.

3.2 Compressed Column-Store Layout

To improve memory bandwidth utilization and to keep the memory footprint of enterprise data storage small, the column-store data in the IMCE is compressed. Especially, lightweight compression schemes [BHF09, LSF09] are applies, where the CPU overhead for compression and decompression does not overshadow the gain of reduced memory bandwidth utilization. Furthermore, data stored in columns is particularly well-suited to compression, because all values of a column stem from the same value domain.

All columns in the column store are compressed using dictionary coding: The values of the column are replaced by integers (value IDs) that point to the original value, which is stored in a separate sorted array. For example, assume we have a column where each entry is one of four colors: [red, green, red, blue, white, red, ...]. The dictionary-encoded variant consists of a sorted dictionary [blue, green, red, white] and an array named *column vector* [2, 1, 2, 0, 3, 2, ...] containing the dictionary positions. We assume that the size of the dictionary (i. e., the distinct-value cardinality of the column) is known in advance. Therefore, we can encode each integer with the minimum number of bits (two in our example) and pack them in a contiguous memory location. In our example, this contiguous location would contain the following binary string: 100110001110... If string values are stored in the dictionary, the dictionary can be compressed using prefix compression. On the compressed column vector, we apply more lightweight compression schemes, such as run-length encoding or cluster coding [LSF09].

3.3 The Delta Store

The above described dictionary encoding does not allow cheap modifications or insertions. For example, if a new column value appears due to an insert operation, it has to be placed at the correct position in the sorted dictionary. The positions of the following entries have to be shifted, requiring an update to all the affected integers in the column vector.

To solve this problem, the column store provides a write-optimized storage location called *delta store* (the read-optimized store is called *main store*). The delta store is also column-oriented and utilizes dictionary coding. In contrast to the main store, the dictionary is not sorted. For fast access, a CSB+ tree index [RR00] is generated on the dictionary, which maps the value ID (as index key) to the position in the unsorted dictionary (as index value).

The delta store requires more space than the main store. Due to the unsorted dictionary and the additional CSB+-tree lookups, the performance is also slightly worse than the main store. Therefore, from time to time, the delta store is merged together with the main store [KGT$^+$10]. To keep the system accepting updates during a merge, a second delta store is created for each column. The secondary delta is declared to be the primary delta after the merge has finished. Furthermore, while merging a table, the merged table is written to a new main store. The old main store can then still be used to process read queries during the merge phase.

3.4 Row-Store Layout

The row store keeps rows intact and organizes them 16 Kb pages which are kept in a linked list, one per table. Variable-length fields are stored in “referenced mode”, i. e., the data pages contain a pointer to the memory location, where the variable length value is stored. Because of fixed-length rows, each row can be identified by a its physical address. This *row ID* consists of a page identifier and an offset. In contrast to the column store, the row store is not compressed. This allows simple and fast insertion: When a new row has to be inserted, the *page manager* searches for a free slot in an existing page. If no such page exists, a new batch of pages (64 MB) is allocated and the record is placed in a slot of a new page. To speed up key-based lookup operations, columns in the row store can be indexed using a cache-aware index structure. Such an index maps a value to the row IDs of its occurrence. Indexes exist only in main memory and are built on the fly when the table is loaded from external memory.

3.5 Insert-Only and Multi-Version Concurrency Control

Many business applications have the need to keep historical data, e. g., for legal purposes. Therefore, they do not want to update and overwrite data in place. To support these applications, the column store of the in-memory computing engine follows the *insert-only* approach, i. e., physically, data is never updated in place. Rather, when a row is updated, the new values are inserted into the delta store and the old values from the main store are marked as overwritten. Likewise, deletions mark a row to be deleted but do not physically remove the entry. In the IMCE, the so-called *consistent-view manager* (CV manager) keeps track of these changes. As an analysis of several customer systems has revealed that many enterprise applications are not update intensive [Pla09]. However, from time to time, the in-memory store should be freed from "old" entries. These old entries can be removed during the delta merge. To fulfill the requirement that the data can still be queried, the merge process places old rows into yet another store, namely the *history store*, which can be placed on external memory to reduce main-memory consumption.

Insert-only is not a prerequisite for optimistic transaction synchronization [HR01]. However, the two concepts go well together, because concurrently running transactions have to keep their write sets in optimistic concurrency control (OCC) anyway. The IMCE combines the insert-only approach with multi-version concurrency control (MVCC). Because insert-only does not physically remove or overwrite rows, the CV manager can keep track of multiple versions. The CV manager can also make sure that every read transaction can operate on a stable snapshot of the database at the time of transaction start (snapshot isolation). Read/write and write/read conflicts between concurrent transactions can therefore be avoided, thus facilitating transaction parallelism. To resolve write/write conflicts, we do however not rely on an optimistic concurrency control scheme. Instead, we use classic row-level locking. This avoids unnecessary transaction rollbacks due to overlapping write sets. The IMCE provides transaction levels read committed and repeatable read. We omit the discussion of insert-only and MVCC concepts in the row store for brevity.

3.6 Data Distribution

The SAP in-memory computing engine runs on a cluster of blades, where the blades are interconnected by ethernet. The persistent store resides in a network-attached storage (NAS) or a storage area network (SAN), connected to all blades [MSLR09]. Because each blade has access to the same database, we would classically call this architecture *shared disk* [Sto86]. However, because the *primary* copy of the data does not reside on disk, but in the main memory private to each blade, we characterize it as a *shared-nothing* architecture. With each blade having access to the same disk, the system has the chance to survive blade failures: The data stored in the main memory of a corrupted blade can be recovered from disk and re-loaded to the other blades automatically. Alternatively, a backup blade can take over. We will come back to implementation details of distribution in Section 5.1.

3.7 Parallel Data Processing

The in-memory computing engine allows to horizontally partition tables and distribute the partitions across the blades. Several partitioning criteria are possible: round robin, range-based, hash-based, etc. Running queries against partitioned data naturally can be executed in parallel (data parallelism). Distributed queries are implemented on the basis of a distributed query plan, distributed query processing algorithms, and a distributed plan execution engine. To make full use of all compute resources, within one blade, the system supports most kinds of query parallelism: *inter-query*, *inter-operator*, and *intra-operator* parallelism [Rah94]. For fast scan and aggregation, special *single instruction multiple data* (SIMD) instructions are applied, that can, for example, sum up four integers in parallel [WPB$^+$09]. The IMCE does not support the concept of *pipeline parallelism.* Rather, every operation in the query plan runs to completion, before the result is sent to the next operation. This allows us to optimize the data structures keeping intermediate results and to save communication costs, when network access is required to ship intermediate results.

3.8 Persistence, Logging, and Recovery

Just like a disk-based DBMS, an in-memory DBMS has to ensure that after a power failure the database can be recovered. The SAP in-memory computing engine is built on a page-based persistence, i. e., a log entry can be written for a physical page (physical logging) [GR93]. The log protocol adheres to write-ahead logging (WAL), i. e., before a transaction commits, the log buffer is written to disk. To avoid undo logging, the buffer manager implements the shadow-page concept [Lor77]. Due to the WAL protocol, pages can be written to external memory in deferred mode (after transactions have committed).

The column store implements logical logging to keep track of the modifications on the delta store. The logical log is written to a *virtual file*, which consists of pages provided by the page-based persistence layer. Thereby, the logical log is recoverable. Note, the delta store itself is not written to external memory. Rather, it is built from the logical log during system startup. For the row store, as special differential logging technique is applied, which enables parallel log streams to multiple external disk drives [LKC01]. All data and log information is written into *disk volumes* – files provided by the operating system.

4 Data Management Requirements of Multi-Tenant Applications

The in-memory computing engine allows multiple tenants to share one instance, i. e., it follows the *shared DBMS instance* approach described in Section 5.2. Instance sharing is implemented by adding tenant-management logic in the database system. We will see how this works in the next section. First, let us look at the data management requirements posed by multi-tenant applications, such as Business ByDesign [ByD] – SAP's enterprise solution for small and mid-size companies.

In the following, we assume a classical three-tier software architecture, where the *presentation layer* is separated from the application logic running on various application servers, and the *application layer* is separated from the *database layer*. Orthogonal to this software stack, a multi-tenant application needs software to maintain the stack. We omit the discussion on the aspects of multitenancy to the presentation layer (e. g., tenant-specific customizations) and the application layer (e. g., tenant-specific application logic) and focus directly on the requirements posed on the database layer and the database-specifics of the maintenance software.

For an application, the overall requirement to the database system is *transparency of resource sharing*. The application should operate on a tenant database, as if each tenant would possess its own database instance. For example, the application should be able to open a database connection for a specific tenant. All queries via this connection are then local to the connection's tenant. This way, the application does not have to deal with "special logic" to query tenant-specific data (such as having to inject a tenant ID to all SQL queries). The application should also be disallowed to directly access data from other tenants (if an application requires interaction with other tenants, it has to use the standard communication channels, e. g., via middleware software, such as SAP Process Integration [PI]). Transparency of resource sharing also means that a tenant should not be influenced by other tenants whom resources are shared with. For example, when a tenant crashes, it should not tear down other tenants. Furthermore, when a tenant requires peak database performance, other tenants sharing the same resources should still meet their service level agreements (SLAs).

In the requirements discussion so far, we have neglected the fact that the application running on a multi-tenant database system also has to support multiple tenants and needs to share resources. The database-specific fraction of these shared resources are schema artifacts and shared table data. To distinguish them, we classify tables as: 1. *tenant-independent tables*, 2. *tenant-dependent tables*, and 3. *tenant-private tables*. Tenant-independent tables contain information useful for any tenant, for example, currency exchange rates. Tenants are not allowed to modify these tables. Because to the application, every tenant has the same initial schema, the database should support to specify an initial schema (called *tenant template* in the following) from which new tenants can be created by cloning. The tenant template contains tenant-independent tables and tenant-dependent tables. Initially, both can contain data, for example, default values (but do not have to). A running tenant stores its data in the tenant-dependent tables. Applications should be allowed to extend the initial schema with tenant-private tables. It should also be possible, to add new columns to existing tenant-dependent tables. This is important for application customizing.

Of course, the maintenance software of a multi-tenant application cannot be transparent w. r. t. resource sharing in the database layer. It has to organize resource sharing. The database system should provide services for tenant life-cycle management, such as creation (see above) and dynamic tenant startup and shutdown. It should make tenant relocation, backup, and cloning possible without downtime or significant performance degradations. Relocation is required when tenants have to be moved to faster/larger machines or if they want to go on-premise. Closing is often required to create test tenants.

The database system should also allow to define tenant-specific quotas, e. g., to model SLAs. Furthermore, to support flexible software roll-out scenarios, the database system should provide for tenant-specific release management. Of course, this only applies to the database schema and tables, not to the code base of the database system itself (which provide tenant-specific release management without tenant relocation). To react on customer-specific failures and performance glitches, tenant debugging, tracing, and monitoring facilities are required. Finally, the database system should allow to replicate tenants for high availability and load balancing. Please consult [AJPK09] for further readings.

5 Tenant Integration in the In-Memory Computing Engine

The in-memory computing engine implements multitenancy through the shared DBMS instance approach. The basic concept to achieve transparency of resource sharing is *tenant separation*. The IMCE carefully distinguishes between artifacts that have to be separated between tenants and those that have to be shared among them. We will come back to this point in Section 5.2. Because support for multitenancy is intermingled with the distributed system aspects of the IMCE, we turn our focus first to distribution.

5.1 Distributed Instances

The topic map in Figure 1 shows the elements of a distributed IMCE instance and how they are related. A distributed instance consists of multiple database servers, each of which runs in a separate operating system process and has its own disk volumes. The database servers of a distributed system may be distributed across multiple hosts – nevertheless it is also possible to run multiple database servers on one host. The key principle of executing database operations is to run the operations at the location, where the data resides. Therefore, database servers may need to forward the execution of some operations to other servers that own some data involved in the operation. In a data distribution scenario, the database clients do not need to know about the distribution. They may send their requests to any database server. If the server does not own all data involved, it will delegate the execution of some operations to other servers, collect the result, and return it to the database client (performance considerations nevertheless make it desirable to send the request immediately to the processing node).

In a distributed IMCE system, a central component is required that knows the topology of the system and how data is distributed. In our system, this component is called *name server*. The name server holds for example the information, which tables of table partitions are located on which IMCE database server. In a multi-tenant system, the name server is hierarchical in a way, that the global nameserver knows the assignment of tenants to IMCE database servers, whereas the tenant-specific nameservers hold the information about table locations.

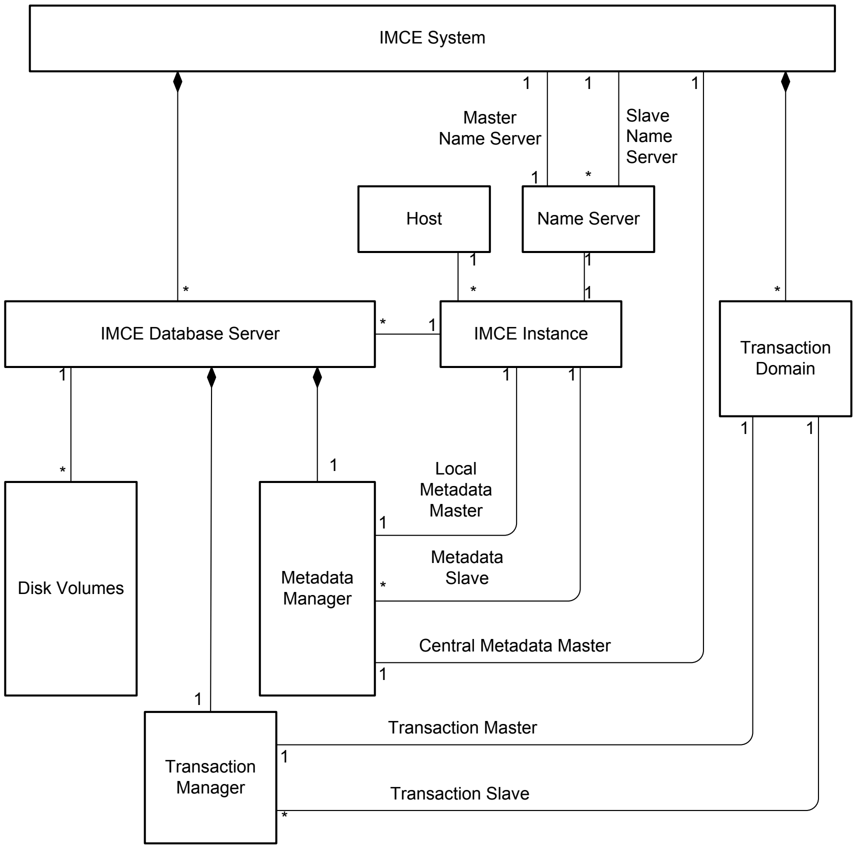

Figure 1: Structure of Distributed IMCE systems

When processing a query, the database servers ask the name server about the locations of the involved tables. To prevent this from having a negative impact on performance, the topology and distribution information is replicated and cached on every host. In each IMCE system, there is one master name server that owns the topology and distribution information. This data is replicated to slave name servers that run on each host. The slave name servers store the replicated data to a cache in shared memory from where the IMCE database servers on the same host can read it (if belonging to the same system).

In a data distribution scenario, the partitioning can be done table-wise or by splitting tables horizontally. With table-wise partitioning, the name server assigns new tables to a IMCE database server based on the current distribution of tables and load (number of tables assigned and resource consumption of the database servers). Data for this table will reside only on that database server, if no additional replicas are specified. It is also possible to specify that a table is split over multiple IMCE database servers. Initial table partitioning is done by the name server based on a size estimation specified by the application. When records are inserted into a partitioned table, they may be distributed to other IMCE database servers based on the table's partitioning specification. In a multi-tenant IMCE system, partitioning is done tenant-wise.

The master name server of an IMCE system is a critical central component in a distributed setup. To ensure high availability, it is possible to have additional name servers as backup master name servers (standard configuration is two). During normal operation, the backup master name servers receive all replicated data like the slave names servers. If the master name server fails, the backup server negotiate and one of them takes over the role of the master server.

Analogous to the topology information and nameserver, each IMCE database server contains a meta-data manager that provides meta-data-related interfaces to all other IMCE components. Meta data are for example table definitions or types of the columns. In a distributed IMCE system, meta data is defined and stored centrally and replicated to all database servers. One of the database servers takes the role of central *meta-data master*, of which there is one per IMCE system. Only on the host running the central meta-data master, global meta data can be created.

On each host, there is a *local meta data master* that receives replicated meta data from the central master. The local meta-data master makes the replicated meta data available to the IMCE database servers on the same host using shared memory. These database servers are called *meta-data slaves*. Meta-data slaves have only read access to the replicated central meta data. Central meta-data master and local meta-data master are not separate server processes, but are hosted by specific IMCE database servers. Meta-data replication is handled transparently by the meta-data managers. All other IMCE components use the meta-data manager interface. Therefore, the replication of meta data is completely hidden from them. For read access, meta-data replication is also transparent for database clients. No matter to which server a database client is connected, it can read all central meta data.

Because central meta data is created in the meta-data master only, database clients that need to create or change central meta data need to connect to the meta-data master. A second type of meta data, *local meta data*, can be created on the local database server. This type of meta data is local to the database server where it is created and is not shared with others, even on the same host. This feature is used in multi-tenant systems for defining meta data that is private to the tenant (see Section 5.2).

To ensure transactional consistency in distributed setups, the IMCE supports distributed transactions. Each IMCE system can have multiple *transaction domains* to which the IMCE database servers are uniquely assigned (as shown in Figure 1). Distributed transactions may span only IMCE database servers within the same transaction domain. In a transaction domain, there is one IMCE database server that has the role of the transaction master, while the others act as transaction slaves. The transaction master also coordinates the two-phase commit protocol.

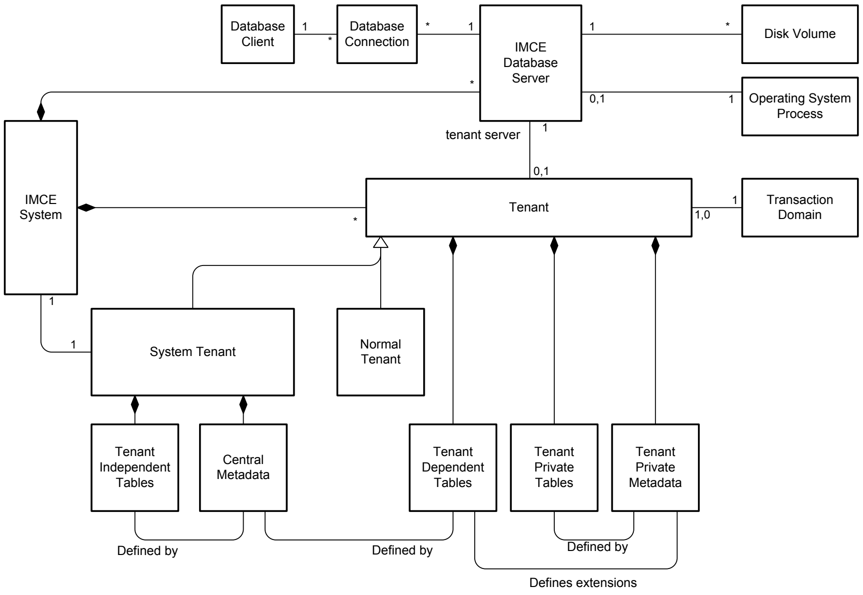

Figure 2: Multitenancy in the In-Memory Computing Engine

5.2 Multitenancy

Since SAP R/2, multitenancy is to some degree achieved with the *client concept*: All client-dependent tables contain a client column that is used by the application server to distinguish multiple clients (shared table approach). The client column can be used for partitioning and clustering at the database level, but for conventional DBMSs, the client identifier is just another column, i. e., conventional DBMSs are not aware of the client concept. In IMCE, this is different, because it offers support for multiple tenants already at the DBMS level (shared DBMS-instance approach).

As we will see, the tenants of one IMCE system share common meta data, but for the actual tables that hold application data, there are separate instances per tenant. This means that each tenant has its own set of application-specific tables storing all the tenant-specific data. As a result, a tenant-identifier column is not needed in the tenant-specific instances of the table. The tenant identifier would always contain a constant value.

The topic map in Figure 2 explains, how important concepts relevant for multitenancy are related in the in-memory computing engine. An IMCE system with multiple tenants is a distributed IMCE system, where each database server uniquely belongs to one tenant. Tenants run in different operating system processes with their own virtual memory. Additionally, they also have their own disk volumes. This separation allows us to isolate tenant crashes or failures and to recover from them tenant-wise. In the following, the term *tenant server* is used to refer to the database server that belongs to a specific tenant.

Table 2: Tenant Separation and Types of Tables

Type	**Table Content (Data)**	**Meta Data**
Tenant-Independent	Stored in system tenant; Read access from other tenants	Stored in system tenant; Read access from other tenants
Tenant-Dependent	Independent instances of the table exist in each tenant with content that is private to the tenant	Table definition is stored centrally in system tenant with read access from other tenants; tenant-specific extensions (additional columns, etc.) are stored as private meta data
Tenant-Private	Stored as private data of the tenant with no access from other tenants	Stored as private meta data of the tenant

To support data and meta-data sharing in multitenant applications (such as Business ByDesign), the in-memory computing engine provides the three table categories introduced in Section 4: tenant-independent tables, tenant-dependent tables and tenant-private tables. To implement these table types, each IMCE system has one special tenant called the *system tenant* (other tenants are called *normal tenants*). The system tenant contains tenant-independent meta data and application data, as well as the tenant-dependent meta data. Both can be accessed in read mode by normal tenants. Tenant-independent data are, for example, country names or time zones for currency rates. Tenant-independent data can be relevant to all tenants and is delivered by SAP as built-in content. As the owner of the global meta data in a distributed IMCE system (see Section 5.1), the system tenant also manages the central meta data of tenant-independent and tenant-dependent tables which is, like the built-in content, available to all other tenants for read access.

Normal tenants can define their own private meta data (i. e., new tables, columns, views, functions), which is only visible to them. Furthermore, The data from normal tenants itself is also isolated: In the context of one tenant, data from other normal tenants cannot be accessed directly. If a client needs access to more than one normal tenant (for example a tenant management tool), it needs to open separate database connections to each tenant. Table 2 summarizes how meta data and content is stored and accessed for the three types of tables.

As a further measure to guarantee tenant isolation, each tenant is assigned to its own transaction domain. This ensures that a transaction is restricted to one tenant and that that a single transaction cannot span multiple tenants. Therefore, we only allow distributed transactions within one tenant but not across tenants.

Having different disk volumes for different tenants makes it easier to support tenant lifecycle management, such as tenant copy, tenant relocation, or tenant deletion. When relocating a tenant to a different IMCE database server in the same IMCE system, the tenant-specific volume can be detached from the original server and attached to the new server.

For moving or copying tenants between different IMCE systems, the problem arises how shared meta data should be handled. For such a cross-system tenant move, it is ensured that the tenant data to be moved is consistent with central meta data in the new system. Tenant creation is implemented by cloning a template tenant.

Database clients need multiple database connections if they need to access more than one tenant. However, sometimes applications need to combine tenant-dependent tables with tenant-independent tables (stored in the system tenant) in one query – for example in a join operation, a sub-query, or a union. To process this type of queries, the tenant servers for normal tenants have indirect read access to tenant independent tables. A database client that is connected to the database server of a normal tenant may combine tenant-independent tables and tenant tables in the same query. If a tenant server receives such a query, it delegates the corresponding operations to the database server of the system tenant, combines the results with local results, and returns them to the database client. Therefore, it provides federation capabilities between the local and the system tenant. This way, the database clients need not be aware of the fact that the system tenant is involved.

Because the system tenant belongs to a different transaction domain, a query that involves access to tenant-independent tables is executed by running two different transactions. Therefore, transparent access to tenant-independent tables by a normal tenant is limited to read-only operations. If a database server assigned to a normal tenant receives a request to modify the content of tenant independent-tables, it reports an error. A database client that needs to write tenant-independent tables needs to open a connection to the system tenant server.

As introduced above, if meta data is created in a normal tenant, it is stored in the tenant as private meta data which is not accessible by other tenants. When reading meta data in the context of one tenant, the result is calculated as the union of central meta data and tenant-private meta data. This is done by the meta-data managers and is hidden from all other IMCE components. It is possible to use private meta data for tenant-specific extensions of centrally defined meta data. Tenant-specific column extensions are implemented by creating new columns (using ALTER TABLE) in the tenant-private meta data. These additional fields only exist in this specific tenant.

6 Conclusion

In this paper, we highlighted the multitenancy features of the in-memory computing engine. The in-memory computing engine is SAP's main-memory database management system. Its target hardware is a cluster of blades, where each blade can hold up to 2 Tb of main memory and up to 64 physical cores. The IMCE has a row store and a compressed column store, supports insert-only data management, data partitioning and distribution across blades, as well as parallel query processing.

Regarding multitenancy, the IMCE provides support at two conceptual levels: at the database level, by tenant separation and life-cycle management, and at the application level by providing capabilities for meta-data and data sharing. In IMCE, distribution and multitenancy are closely related. The multitenancy implementation builds on meta-data replication features of distributed IMCE instances.

Acknowledgements

We would like to thank Meinolf Block for hist valuable input to this document. Furthermore, we would like to thank the experts of TREX, PTime, LiveCache, and MaxDB, who built the SAP in-memory computing engine.

References

[AJPK09] Stefan Aulbach, Dean Jacobs, Jürgen Primsch, and Alfons Kemper. Anforderungen an Datenbanksysteme für Multi-Tenancy- und Software-as-a-Service-Applikationen. In *BTW*, pages 544–555, 2009.

[BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-Based Order-Preserving String Compression for Main Memory Column Stores. In *Proc. SIGMOD*, pages 283–296, 2009.

[ByD] SAP Business ByDesign. `https://www.sme.sap.com`.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions Computer Systems*, 26(2), 2008.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. SOSP*, pages 205–220, 2007.

[FK09] Daniela Florescu and Donald Kossmann. Rethinking Cost and Performance of Database Systems. *SIGMOD Record*, 38(1):43–48, 2009.

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and. Data Engineering*, 4(6):509–516, 1992.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[HPP10] HP ProLiant DL980 G7 Server, 2010. `http://h18004.www1.hp.com/products/quickspecs/DS_00190/DS_00190.pdf`.

[HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2001.

[JA07] Dean Jacobs and Stefan Aulbach. Ruminations on Multi-Tenant Databases. In *Proc. BTW*, pages 514–521, 2007.

[KGT$^+$10] Jens Krüger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. Optimizing Write Performance for Read Optimized Databases. In *Proc. DASFAA*, pages 291–305, 2010.

[LKC01] Juchang Lee, Kihong Kim, and Sang Kyun Cha. Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Databases. In *Proc. ICDE*, pages 173–182, 2001.

[Lor77] Raymond A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, 1977.

[LSF09] Christian Lemke, Kai-Uwe Sattler, and Franz Färber. Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. In *Proc. BTW*, pages 486–497, 2009.

[MSLR09] Olga Mordvinova, Oleksandr Shepil, Thomas Ludwig, and Andrew Ross. A Strategy for Cost efficient Distributed Data Storage for In-Memory OLAP. In *Proc IADIS AC*, pages 109–117, 2009.

[OAE$^+$09] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.

[PI] SAP Process Integration. `http://www.sap.com/platform/netweaver/components/pi/index.epx`.

[Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *Proc. SIGMOD*, pages 1–2, 2009.

[Rah94] Erhard Rahm. *Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley, 1994.

[RR00] Jun Rao and Kenneth A. Ross. Making B$^+$-Trees Cache Conscious in Main Memory. In *Proc. SIGMOD*, pages 475–486, 2000.

[Sto86] Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[WPB$^+$09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using On-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.