

Chapter 2

Virtualization for Data Management Services

Virtualization is the key concept to provide a scalable and flexible computing environment in general. In this chapter, we focus on virtualization concepts in the context of data management tasks. We review existing concepts and technologies spanning multiple software layers. We first start outlining the general principles of virtualization in the context of Database-as-a-Service by looking at the different layers in the software and hardware stack. Thereafter, we will dive into detail by walking through the software stack and discuss different techniques to provide virtualization at different layers. Overall, the chapter provides a comprehensive introduction and overview of virtualization concepts for data management services in large scale scenarios.

2.1 Core Concepts of Virtualization

The concept of virtualization is one of the conceptual and system architectural pillars of computer science and computer infrastructures. Virtualization in general provides a layer of indirection to build an abstract view in order to hide the specific implementation of computing resources. Creating such an indirection step between the resources and the access to the resources provides a huge variety of advantages. The decoupling hides the implementation details to the using component. It also adds flexibility and agility to the computing infrastructure to reduce capital expenditures as well as operational expenditures as already discussed in the introductory chapter. In general, the concept of virtualization can be used to solve many problems related to provisioning, manageability, security etc. by pooling and sharing computing resources, simplifying administrative and management tasks and improving fault tolerance with regard to a complex software stack. Already in

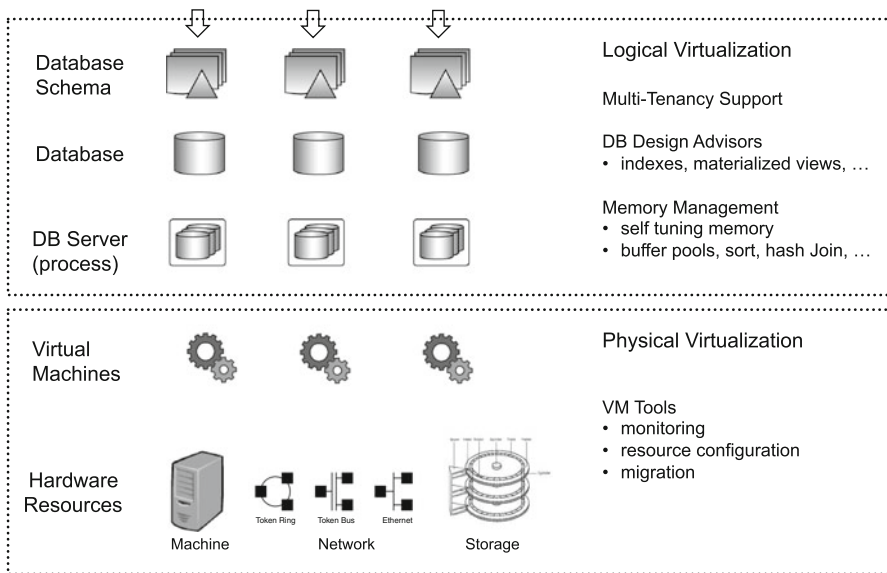


Fig. 2.1 Physical and logical virtualization in a data management service software stack

2010, IDC forecasted that “Enterprise Server Virtualization Market to Reach \$19.3 Billion by 2014”.¹

While virtualization is a general concept for computing infrastructure, we will focus on data management specific views within this section. To enable data management services, we distinguish between physical and logical virtualization within the complete system stack. As shown in Fig. 2.1, we can clearly identify two different layers of physical and logical virtualization.

Physical virtualization comprises the abstract view with regard to physical devices such as computing nodes on the one side and storage systems on the other side. At the same time, all infrastructural devices like switches, routers etc., are subsumed by the virtual view of computing resources. More specifically, physical virtualization abstracts from

- **CPU:** The variety of different types of CPU, multi cores or heterogenous processors like GPU or FPGA for special purpose computing tasks are hidden by a virtual CPU.
- **Memory:** The running application does no longer see the real memory of the hardware but – similar to classical virtual memory concept of operating systems – will see only the fraction of (virtual) memory which is explicitly assigned to the

¹http://www.information-management.com/news/IDC_predicts_virtualization_growth-10019216-1.html.

virtual resource. Additionally, the software does no longer have access to features like DMA (direct memory access) or memory of a graphic card.

- **Network:** The physical virtualization concept abstracts from specific network infrastructures and topologies. Load balancing at a lower level (e.g. HTTP request for web server farms) happens “behind the scenes” and therefore transparently for the application running in a virtualized computing environment.
- **Storage:** Most important for the application area of data management services, physical virtualization can also provide an abstract view of specific storage environments, e.g., to transparently perform data replication between different sites/systems or integrate heterogenous types of storage (SAN versus NAS; SSDs versus HD drives; fast versus slow devices; ...). We will detail this kind of virtualization when considering different use cases in the remainder of this chapter.

Logical virtualization in our context addresses the abstraction on a data management level. The scope of logical virtualization comprises the following concepts:

- **Database Server:** The level of a database server provides a runtime for database services and acts as host for multiple databases with individual schemas, users, and potentially physical setups. A database may be placed at a server without any knowledge about the specific characteristics of the underlying software and (when running in a physically virtualized environment) hardware setup.
- **Database Schema and Databases:** Although not in the focus of a traditional database design process, there are many applications with multiple users (or user groups) having a slightly different view on a database schema. The concept of multi-tenancy recently addressed this issue as a modeling and system design problem. This level abstracts from having multiple tenants in the same database schema (with common and private data sets) or maintaining individual databases for different user groups. Section 2.4 will discuss the different options and alternative implementations in depth.

Since the specifics of the physical devices or the database systems are hidden from the application layer, the properties of the devices and software components have to be published in the form of Service Level Agreements (SLAs) to enable an efficient and cost-effective software stack. SLAs can be considered contracts between different layers which can be (dynamically) negotiated and monitored during runtime. The core concept of SLAs will be discussed in great detail in Sect. 5.1.

Moreover, it is important to point out that the different layers shown in Fig. 2.1 reflect the possible points of virtualization. Every layer or component within a layer can be virtualized in a specific setup; there is no must to apply virtualization for every piece in the stack. For example, hosting multiple applications within the same database running on one single database system, directly on bare hardware is a valid setup for data management services. The same holds for the other extreme of creating a separate database for every application running in an isolated database

server on a virtualized node. The great challenge in the context is therefore to determine an “optimal” or “well-balanced” setup based on an end-to-end design.

2.1.1 Limits of Virtualization

Having listed most of the benefits of virtualization in general, we also have to pinpoint some of the limits. We already mentioned the loss of direct access of the computing resources by introducing the indirection step of virtualization. Specific features of the underlying mechanisms have to be explicitly modeled in form of some service description and SLAs have to be potentially determined. Hiding the details of physical resources however turns out to be extremely unfortunate for applications making some of their behavioral decisions based on assumptions about the physical layout. This of course is true for database systems in terms of node configurations. For example, the size of a buffer pool may have an impact on the choice of a specific physical plan operator (hash versus nested loop join). Another example with respect to storage systems is the alternative of an index scan compared to a table scan; the optimizer usually bases the decision on a cost model which is supposed to directly reflect the physical representation.

In addition to the abstraction of features, virtualization always causes some degree of performance penalty by the overhead introduced through the virtual machine, virtual storage, virtual schemas in the context of a shared database for multiple applications. The performance issue can only be reduced by better virtualization support, e.g. native support of multi-tenant database or hardware support when applying virtualization at the node level.

2.1.2 Use Cases for Virtualization

In order to achieve a better understanding, we outline three typical use cases for virtualization in the context of data management services: server consolidation, migration and load balancing, and high availability.

- **Server consolidation:** One of the driving forces to exploit some form of virtualization is based on the observation that typical server systems are grossly underutilized. For multiple reasons, ranging from organizational to security or fault tolerance reasons, we can observe a single server system per application. Not only applications but also database servers (sometimes in combination with web servers) are placed onto a single system. Without virtualization, provisioning is usually performed to cope with potential peak load requirements leading to usually underutilized computing resources. Figure 2.2 illustrates the situation. Running multiple applications in virtual machine environments opens up the potential to share the same physical machine and saving hardware energy costs

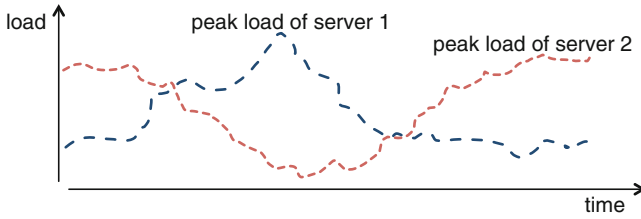


Fig. 2.2 Time dependent peak load provisioning as motivation for server consolidation

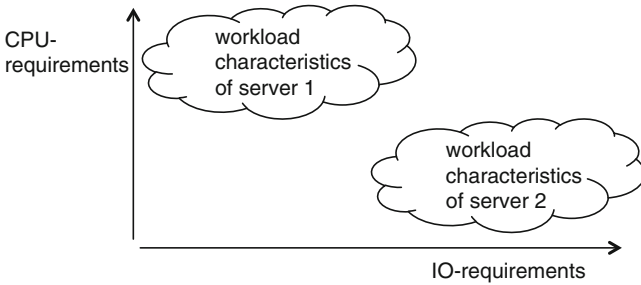


Fig. 2.3 Time invariant workload characteristics as hint for server consolidation

and reducing administration overheads. The major challenge however is to cluster applications with “compatible” applications and deploy them on the same component. In the context of physical virtualization, such a component is either the computing node or some storage system. For logical virtualization, database systems may hold multiple databases or applications may share a common database schema (multi-tenancy) to achieve the goal of peak leveling. In all scenarios, applications with contrary SLA requirements or contrary resource requirements should share the same underlying component. For example, as illustrated in Fig. 2.3, database workloads with IO-intensive queries should be “paired” with workloads exhibiting a more CPU intensive query load.

- **Migration and Load Balancing:** Usually ranked second behind the server consolidation scheme, virtualization can substantially help in load balancing or support of maintenance task. Exploiting the indirection layer introduced by the virtualization scheme enables to migrate the current state of an application to a different “location”, i.e. either physical node or database server at the data management application level. The application within a virtual machine or the virtual machine itself can be suspended; the image can be shipped either directly or via a shared storage to another resource; afterwards, the application (or the computing node) can be resumed. Figure 2.4 shows the basic principle of this migration scheme which is very popular in enterprise-scale setups.

The triggering event of a migration procedure can be either caused by administrative tasks, i.e. reconfiguration of the “source” systems or driven by load balancing actions. As soon as the infrastructure detects SLA violations at

Fig. 2.4 Virtualization to support server/service migration

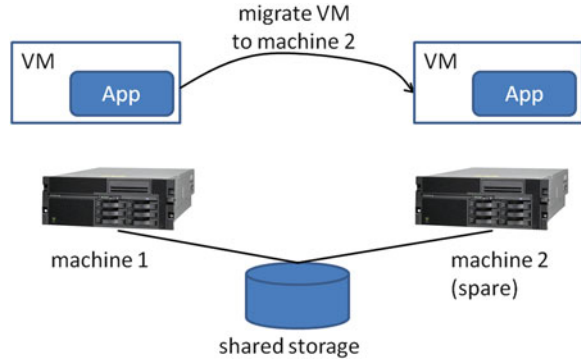
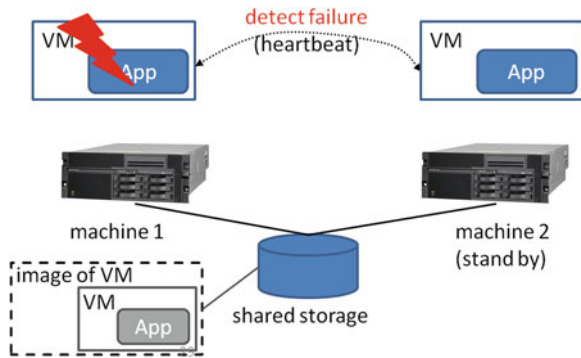


Fig. 2.5 Virtualization to support high availability requirements



certain levels, a migration of specific applications or computing nodes happens to less utilized nodes.

- **High Availability:** In analogy to the server migration use case, virtualization helps to achieve some basic level of high availability. Although the same principles apply, the triggering event is implicitly caused by the failure of the “home system”. As illustrated in Fig. 2.5, a failure will be detected at a stand-by system by a missing heartbeat of the original system. The stand-by system may then start up an image of the virtual machine and continue to deliver the required service.

Achieving high availability using only the image of a virtual machine is obviously restricted to stateless services like for example web servers. If a physical node fails, another system may load the image, start the system and provide access to the web site. For stateful applications especially for databases with transactional guarantees, this simple setup has to be extended with additional recovery procedures at the database server level to protect against data loss or inconsistent database states.

In order to achieve high or permanent availability at a database level, usually a stand-by database is maintained receiving logical or physical log information from the primary database system. For example IBM HADR² provides a mechanism to propagate data changes (DML operations) and structural changes, e.g. DDL statements, operations on table spaces and buffer pools, and reorganization operations to the stand-by system. In order to mirror the behavior of the primary system at the stand-by system, even meta-data of user-defined functions and database procedures are reflected at the stand-by system allowing a different implementation to reflect the procedural semantics if applied at the secondary system. Some systems, e.g. Oracle Data Guard,³ provide the choice of propagating physical log structure to the stand-by system providing a binary compatible system, i.e. usually exactly the same system, as target for logical log. This option is based on SQL statements which are sent to the secondary system and applied like coming from any other application. This functionality provides the benefit to mix high availability with techniques to implement information integration scenarios by propagating state changes of a database to systems requiring the data from a business perspective.

Keeping a stand-by system running causes obviously extra costs; in some enterprise-scale data management infrastructures, such systems are either running in smaller virtual machines or are used for read-only applications. Especially decision-support application may benefit from having a separate database instance next to the operational system.⁴ For example Oracle Data Guard or IBM DB2 explicitly have an option to provide read-consistent views on the secondary system for read-only applications.

Obviously, deploying virtualization schemes in large enterprise IT infrastructures offers even more scenarios with significant benefit for the use of virtualization techniques. Also, all illustrated use cases are valid on the physical section as well as on the logical section of the complete software stack. For example, migration may be performed on a virtual machine layer or on the database layer. In this case, replication techniques are usually used to propagate data from one database system to the other; as soon as the complete database has been moved to a different system, the corresponding connections to the dependent applications are either re-routed to the new system or explicitly disconnected – a re-connect, triggered by the application, then connects the application to the new target system.

²<http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/topic/com.ibm.db2.luw.admin.ha.doc/doc/c0011267.html>.

³<http://www.oracle.com/technetwork/database/availability/index.html>.

⁴This typical separation of transactional mode and analytical mode is also reflected within the structure of the book (see next two chapters); Although there is a clear trend in bringing both world closer together in order to provide real-time analytics, the methods and techniques used to implement and optimize transactional systems or platforms for large-scale data analytics are worth to be considered separately.

2.1.3 Summary

To summarize, there exists a huge variety of use cases exploiting the concept of virtualization; virtualization may be implemented at different layers – ranging from a physical device layer to abstract from the real implementation and physical characteristics to the concept of schema sharing where multiple (obviously similar) applications share the same database schema. In the following section, we will detail the concepts and techniques of virtualization at the physical and at the database system level.

2.2 Virtualization Stack

Having the data management service software stack in Fig. 2.1, there are several layers that can possibly be virtualized. We assume that only one layer is virtualized at a time – merely for simplicity reasons. It is of course possible to virtualize at several layers in a single setup, although such configurations shall be beneficial only in rare cases. As shown in Fig. 2.6, this assumption results in four distinct classes of data management service configurations [25]:

- Class 1: PRIVATE OPERATING SYSTEM (PRIVATE OS)
- Class 2: PRIVATE PROCESS/PRIVATE DATABASE
- Class 3: PRIVATE SCHEMA
- Class 4: SHARED TABLES

A brief overview of these classes will be given shortly, followed by a comparison of the important characteristics. Details about Classes 1 and 4 will be provided in the following sections (while these classes have experienced great attention in the research community, Classes 2 and 3 are only about to be explored in detail and hence have not been researched to that extent.)

Class 1: PRIVATE OPERATING SYSTEM

In Class 1 (PRIVATE OS), physical virtualization occurs on the level of hardware resources (CPU, memory, network). Virtual machine monitors, e.g., VMware, XEN, or KVM, can be used to realize the virtualization. Access to the physical storage is usually virtualized independently of the computing resources (in order to allow for light-weight migration without moving large amounts of data).

Each application in a Class 1 data management architecture owns a separate operating system as well as a database server and databases. Consequently, isolation between different applications with respect to security, performance, and availability is the strongest among all classes. At the same time, resource usage per application is very high which leads to poor utilization of the underlying hardware. Given

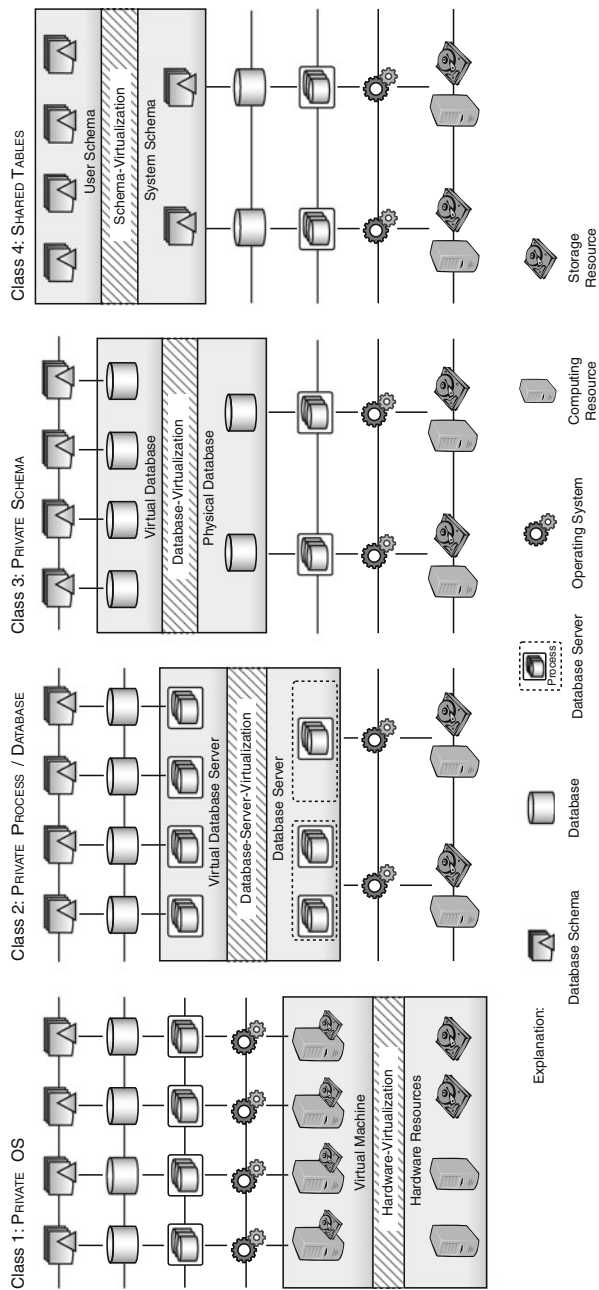


Fig. 2.6 Different classes for data management virtualization

today's hardware, only a few tens of different applications can be hosted on a single machine and hence this architecture does not easily scale beyond a certain number of applications.

Also, it is important to note that this class can be implemented transparently and without any modification to application or database management system.

Class 2: PRIVATE PROCESS/PRIVATE DATABASE

With Class 2 virtualization – PRIVATE PROCESS/PRIVATE DATABASE – logical virtualization occurs at the level of the database server. This can be achieved in two slightly different ways with similar characteristics: (1) Each virtual database server is executed in one or several private processes on the physical machine or (2) all virtual database servers are executed in a single server instance and each application creates private databases inside this server.

While applications in this architecture require less dedicated resources than applications in Class 1, the applications' isolation from one another is weaker compared to Class 1. On the positive side however, this leads to better scalability up to a few hundreds applications per machine. Implementing this class requires the database management system to either provide private server processes (e.g. as instances) or allow for private databases that are maintained and used by different applications without interfering with one another.

Class 3: PRIVATE SCHEMA

Data management software stacks that implement Class 3 (PRIVATE SCHEMA) virtualize the database. Each application accesses private tables and indexes that are in turn mapped to a single physical database. On that account, different applications can use the same physical database in a shared fashion. This leads to weaker isolation between applications than was possible with the previous classes – database facilities like buffer management, logging, etc. are shared now. However, isolation with respect to security can either be enforced in the application or on the database level using access rights to different database objects (like tables).

Each application in a Class 3 architecture has a smaller footprint compared to the previous classes. Accordingly, Class 3 leads to good resource utilization and scalability of up to a few thousand applications per machine. Implementing this class can lead to modifications of today's database management systems in order to hide the existence of other users' database objects and activities from each application.

Class 4: SHARED TABLES

In Class 4 virtualizations (SHARED TABLES), applications share all components in the software stack. The database schema is virtualized such that each application

sees a private schema. However, all private schemas are mapped to a single system schema. This class is best suited for Software-as-a-Service infrastructures where tenants (applications) use the same or a very similar database schema. The databases are then often referred to as “Multi-Tenant-Databases” [3, 4, 23]. Applications that do not share common ground cannot easily be implemented with this class. Although there are means for generic schemas that allow for storing arbitrary relational data – e.g., pivot tables that mimic key-value-stores or universal tables with a number of string columns that all data are mapped to – these implementations have disadvantages with respect to, for example, poor performance or the loss of strong typing.

This class offers the least isolation between applications. Security can be enforced on application level or with row-level-authorization in the database system, but with respect to performance or availability there is almost no isolation. Losing the natural separation of different applications leads to high complexity in the database management system – e.g., the query optimizer needs to be aware of the intermingled data – and complicates maintenance tasks like, e.g., backup, restore or migration of single tenants. The latter operations require costly queries to the operational system.

Particularly advantageous in using this class is the extreme scalability. Because each application requires a minimal amount of dedicated resources, such setups can scale to up to several thousands of applications per single machine.

Although it is possible to implement a Class 4 setup with today’s database management systems without major modifications, the system should be aware of the multi-tenancy (and particularly support it) in order to provide fundamental isolation and good performance.

Comparison of all Classes

Table 2.1 summarizes the high-level characteristics of all four classes. Details about all classes will be provided in the following sections and the comparison will be further detailed in the summary of this chapter. The classes differ greatly with respect to different characteristics and the application of any of the classes is highly dependent on the particular application, its requirements, and desired system properties.

An evaluation of the basic properties is not straightforward, as the following example shall demonstrate. Take for example the “isolation” of applications which is obviously strong in Class 1 and decreases stepwise all the way to Class 4 where it is the weakest. While a strong isolation may be beneficial with respect to security or performance, this strong isolation may prevent, e.g. access to shared data. On the one hand, applications being isolated on a low level implies that single tenants can be backed up, restored, and migrated individually and with little overhead on the remaining tenants. On the other hand, when tenants are hosted in individual operating systems or database servers, strong isolation leads to increased complexity

Table 2.1 Comparison of Classes 1 (PRIVATE OS), 2 (PRIVATE PROCESS/PRIVATE DATABASE), 3 (PRIVATE TABLE) and 4 (SHARED TABLES) with evaluations ranging from “particularly advantageous” (++) to “particularly disadvantageous” (--)

	Class 1	Class 2	Class 3	Class 4
Resources per application (costs)	--	o	++	++
Resource utilization/scalability	--	-	+	++
Provisioning time and costs	--	-	++	++
Maintainability (updates/patches)	--	-	+	+
Isolation (performance)	+	+	o	-
Application independence	++	+	+	-
Isolation (security)	++	++	+	o
Maintainability (Backup/Restore)	+	+	o	--

and costs of other maintenance tasks like applying updated or patches to the system (which is simple in a single database like it is given in Classes 3 and 4).

The comparison in Table 2.1 furthermore shows that the scalability of the architectures increases steadily from Classes 1 to 4. This relates to the circumstance that a single application requires a decreasing amount of dedicated resources and hence has a smaller footprint in Class 4 as compared to, e.g., Class 1. A smaller footprint also leads to lower provisioning costs for new applications which in turn leads to faster provisioning times.

Finally, Table 2.1 shows that all classes are not equally independent from the applications they host. While Class 1 setup can host any application (as long as it provides the necessary performance), Class 4 setups require applications with a common core and similar database schemas to be beneficial.

2.3 Hardware Virtualization

As mentioned in Sect. 2.2, the software stack provides different options to introduce virtualization concepts or specific techniques. In the following, we will focus on the opportunities and existing techniques to implement physical virtualization shielding the specific hardware setup. Following the basic scheme of discussing the relationship of the different layers in the overall software stack, we call this hardware virtualization scenario the **Class 1 Virtualization Scheme (PRIVATE OS)**.

As shown in Fig. 2.7, this class preserves the classic “application software stack” by simulating individual machines, storage devices, and network infrastructures. While the software stack remains basically intact by providing a private operating system environment, hardware resources are shared. Referring to the different classes of “as-a-Service”-models, this setup pictures a “Infrastructure-as-a-Service”-model. The virtual machines and virtual disks are playing the role of the infrastructures which can be used by the application software stack without any

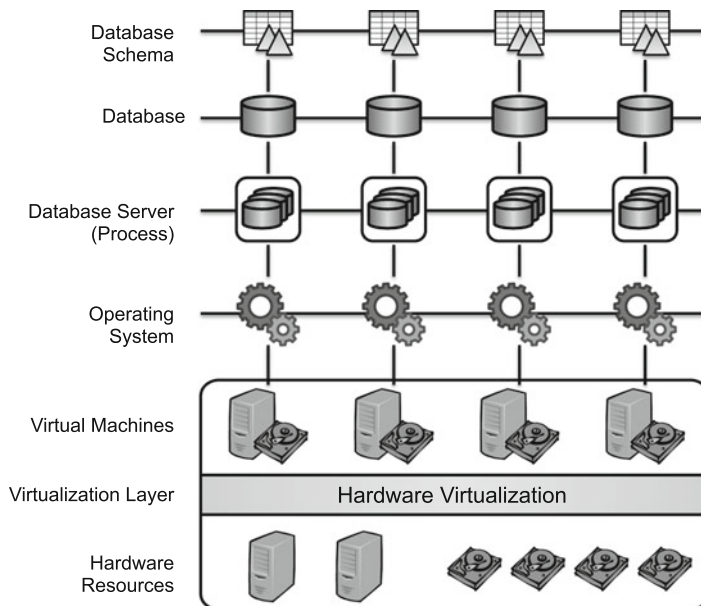


Fig. 2.7 Class 1 Virtualization scheme (PRIVATE OS) – hardware virtualization

detailed knowledge about the technical setup. The overall properties of this scheme can be best illustrated by looking at the notion of “isolation” from different angles:

- **Security:** Since the system resources are accessible only within a private operating system environment, low-level techniques of the virtual machines with the help of native hardware support provide the highest degree of isolation. Software failures or explicit security attacks in a VM container do not affect other concurrently running operating systems with their individual applications. This holds for activating code as well as for accessing data, be it in (virtual) main memory or explicitly stored on virtual disk storage. The downside of this property can be seen in the fact that access to shared data has to be implemented on the application side, i.e. the strict isolation of the virtual environment conceptually does not enable shortcuts for virtual machines running on the same system. Classical remote access techniques have to be used.
- **Performance:** Since virtual machine or disk environments allow to limit the usage of physically available resources, overload on one application system does not affect other applications running in another virtual machine. For example, a VM can be allowed to use only 50 % of the available CPU power. Due to this strict isolation, there is no starvation of other virtual environments.
- **Failure:** Strong isolation also shows the benefit in case of system failures. The failure of one application stack including the private operating systems does not influence other virtual environments. This holds on the machine level as well as on the disk level. If a partition of a virtual disk “fails”, e.g. by running out of

available space, other parts of the storage system are not affected. Obviously, if the underlying hardware fails, multiple virtual environments are typically affected, which requires compensation techniques, potentially deployed on other layers. For example, real disk failures can be compensated by redundantly holding a replica on a different physical node.

- **Manageability:** The final aspect to discuss the main characteristics of hardware virtualization considers manageability of the complete infrastructures. Due to the strict separation of application stacks on a private operating system level, virtual machines and virtual disk can be seen as the logical unit for administrative jobs. Moving an application scenario implies the movement of the virtual disks without any side effects to other concurrently running scenarios. Therefore, tasks of individual migration or other maintenance tasks like backup/recovery can be easily performed on a machine and disk level. It also provides a high degree of flexibility for determining the optimal configuration. However, strict isolation has a price for bulk administrative tasks, e.g. upgrading or fixing software components on the operating system or database system level. Each logical administrative entity has to be configured independently and individually making it hard to operate large service infrastructures with homogeneous application software stacks.

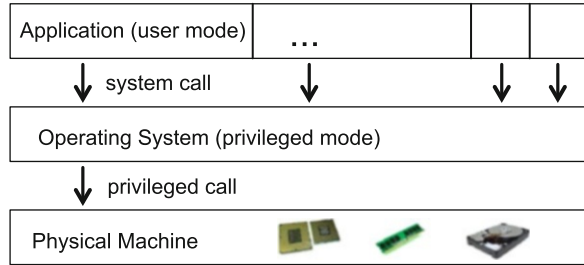
In addition, we have to consider that hardware virtualization on the one hand exhibits a significant footprint with respect to resources, because every instance of a working unit requires a separate OS installation, separate software stack etc. On the other hand, hardware virtualization allows to run heterogeneous application stacks within the same computing environments, for example to support different applications or even different versions of applications requiring different version-s/configurations of the underlying operating system. The strong isolation also allows to provide a very scalable infrastructure, because different virtual machine environments are totally independent of each other. From an overall resource perspective however, a price has to be paid for the strong isolation guarantees. Since no software can be shared, the footprint is typically larger compared to other schemes.

As already shown in Fig. 2.1, hardware virtualization comprises the three types of components machines, storage, and network infrastructures. While virtual network infrastructures are highly specific to the concrete physical environment and not of primary interest for data management services, we will focus on machine and storage virtualization in the following two subsections.

2.3.1 *Machine Virtualization*

Machine virtualization has a long history going back to 1972, when IBM offered the concept of hypervisors for the S/370 under the label of “Virtual Machine Facility/370”. As of now, there are a huge variety of different machine virtualization

Fig. 2.8 Classical stack of hardware, operating system, and applications



techniques as open source as well as commercially available. Although the different techniques cover a wide spectrum of technical properties in terms of their usage scenario, they all share the same architectural concepts of hypervisors. In order to get the basic idea of machine virtualization, Fig. 2.8 shows the classical setup of hardware, operating system, and applications. An application sitting on top of an operating system issues a system call to get access to the resources of the underlying machine. The operating system schedules the physical accesses and coordinates the privileged access to the hardware. Specific hardware characteristics are shielded by the operating system.

In an environment with virtual machines, a hypervisor basically decouples the pure hardware from the operating system level. Applications interact with the operating system in the same way as in the native stack. However, an operating system is now running in user mode. System calls issued by the application are propagated to the hardware by the operating system. Since the operating system is running in user mode, the hardware intercepts the call and propagates the request to the hypervisor running in privileged mode. The hypervisor is handling the request coming from the operating system and maps it to the existing hardware. In order to be able to run on different hardware configurations, virtual machine hypervisors are providing only a restricted set of virtual hardware components. Since this set of hardware components is fixed and implemented by hypervisors running on a variety of different hardware platforms, virtual machines can be easily moved from one hardware to another. Figure 2.9 illustrates the revised system stack with the additional layer of a hypervisor. The figure also intuitively shows that based on hypervisors, it is now possible to run multiple virtual machines on a single physical box.

As a final comment, we would like to point out two variants of the classical virtualization principle. First of all, para-virtualization comprises the fact that a hypervisor is semi-transparent, i.e., some real hardware components are directly visible and accessible by the guest operating systems. Para-virtualization is used in situations when either specific hardware has to be exploited by the application (e.g. license dongles) or the performance penalty coming with the virtualization layer is not acceptable (e.g. graphic cards). However, para-virtualization requires the guest operating system to be virtualization-aware. To run in a para-virtualized setup, the guest operating system has to be modified in order to implement the modified interface that is provided by the hypervisor. Secondly, virtualization on

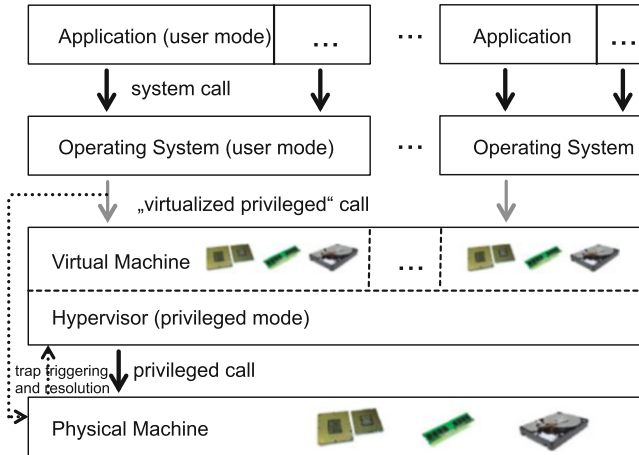


Fig. 2.9 System stack using virtual machines

the machine layer should not be mixed with the concept of emulation. In this setup, an intermediate layer has to simulate the complete hardware allowing applications with an underlying operating system to be run for different machines architectures especially for different types of CPUs. The Bochs IA-32 Emulator Project⁵ is a famous example to provide an x86 hardware architecture for multiple platforms. Recently, Fabrice Bellard demonstrated an emulator of an x-86 architecture running linux completely written in JavaScript and running inside a regular Web browser.⁶ Finally, some operating systems provide virtualization techniques on its own. In such a setup, guest and host operating system are using the same kernel. Examples are the Solaris Container concept⁷ or the Parallels Virtuozzo Container concept.⁸ While typically having a significantly lower overhead, such a container concept provides only a homogeneous operating system environment by exhibiting different views of the same system.

2.3.2 Virtual Storage

Virtual storage follows the same ideas and principles as other virtualized hardware components like CPUs or memory. However, storage takes a prominent role in data management architectures because it hosts the actual persistent data and often is

⁵<http://bochs.sourceforge.net/>.

⁶<http://bellard.org/jslinux/>.

⁷<http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html>.

⁸<http://www.parallels.com/ptn/documentation/virtuozzo/>.

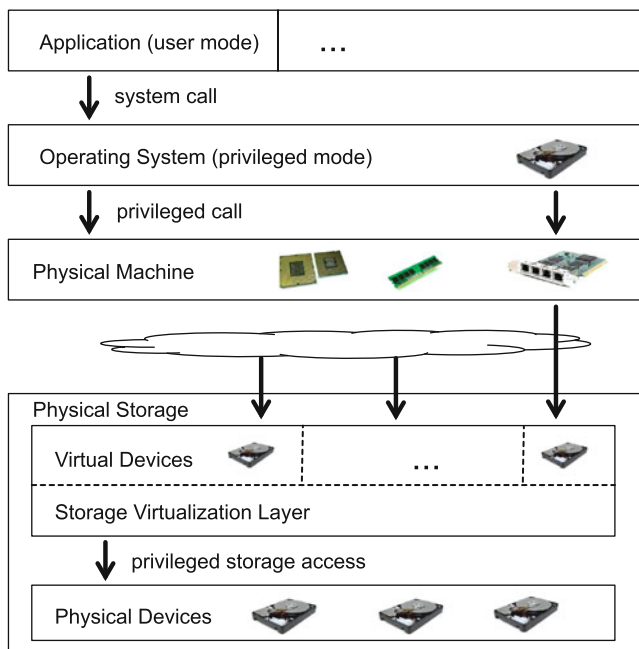


Fig. 2.10 System stack using virtual storage

the bottleneck in data-intensive applications. When moving to flexible, service-oriented architectures, storage requires even more attention because the ability to provision resources or migrate workloads on demand depends heavily on the storage system's flexibility. Storage virtualization introduces a layer of indirection that allows the definition of virtual storage devices (see Fig. 2.10). Whenever an application accesses a storage device, the appropriate driver or the operating system catches the call and redirects it either to the correct partition on a local disk or to the network where it is sent to an external storage device, e.g., a disk array. Here we concentrate on storage that is outside and hence not part of the machine that is running the software (disks that are part of the system are covered by virtual machines, described in the previous section).

The benefits of storage virtualization are manifold and stem from the layer of indirection that is introduced. Motivations to introduce storage virtualization can be as diverse as:

- **Minimize/avoid downtime:** Managing virtual storage devices can be done on demand and while the system is running. Otherwise disruptive operations like changing the RAID level of a disk array or higher level operations like resizing a file system can be performed without the need to take down the system. More precisely, storage virtualization allows for non-disruptive creation, expansion and deletion of virtual storage targets (logical units), non-disruptive

data consolidation and data migration, and non-disruptive re-configuration (as fundamental as adding disks or changing a RAID level).

- **Improve performance:** Introducing a level of indirection comes – by nature – at the cost of a certain performance penalty. Nevertheless, storage virtualization can help to greatly improve storage performance. Virtualized storage can easily be distributed among several physical disks in order to spread and balance the storage load. Furthermore, virtual storage can be provisioned dynamically, on demand and data placement can be controlled to prevent data contention even during peak load phases.
- **Improve reliability and availability:** Virtual storage can be used to greatly improve the reliability and availability of storage targets by, e.g., transparently maintaining (synchronous) replicas of the data on different disks or disk arrays.
- **Simplify/consolidate administration:** Virtualized storage presents a consistent interface to the operating system and therefore simplifies software development and administration. Heterogeneous storage systems can be tiered to benefit from different characteristics of, e.g., magnetic or solid state disk based systems. Administration can further be simplified by consolidating tasks like backup and restore or archiving of data in the storage layer that is hidden behind the virtualization.

Storage virtualization can span multiple levels in the system stack. Actual disks, RAID groups or disk arrays are split into logical units (LUNs – logical unit numbers) and presented as regular storage targets to the operating system. There, a logical volume manager (LVM) further divides storage targets or spans logical volumes over several targets. Hence, there can be arbitrary and flexible mappings from raw disks to actual file systems. Basic operations that are supported by virtualized storage devices are:

- **create/destroy:** Logical volumes can be created or destroyed even without taking down the system.
- **grow/shrink:** Logical volumes can be resized on demand to overcome changes in storage requirements.
- **add/remove bandwidth:** Basic properties of logical storage devices can be changed as needed, for example to increase the bandwidth to a certain target or to add additional disks as needed.
- **increase/decrease reliability:** Adding and removing disks can also be used to change, e.g., RAID levels transparently to the user and hence increase the reliability of a certain storage target.

Technical Realization of Storage Virtualization

While significant work has been conducted to provide an abstract model for CPUs, virtualizing disks and describing the performance behavior for virtual storage did not receive much attention. Pioneering work was done by Kaldewey et al. [24] providing an abstract model and an infrastructure to implement admission control

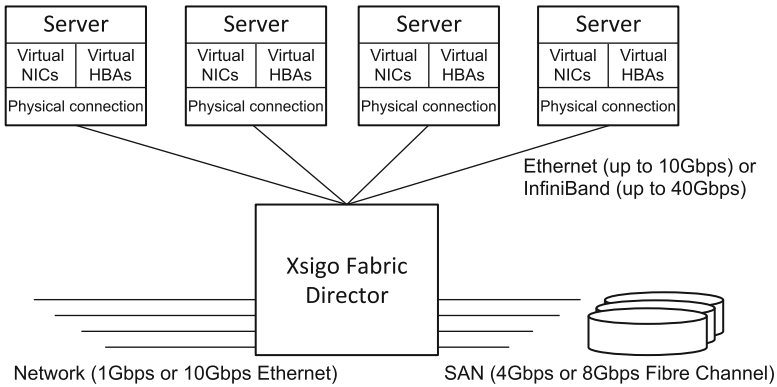


Fig. 2.11 Xsigo systems storage virtualization solution

for different types of access, i.e. sequential versus random access. Within [24], the authors show that disk time utilization (similar to utilization of CPU in the context of CPU scheduling) provides a significantly more efficient use of the underlying disk resources.

To provide an even deeper insight on how commercially available storage virtualization can be realized, an example setup is presented. A sophisticated solution to the problem is provided by a company named Xsigo systems⁹ recently acquired by Oracle.¹⁰ The solution’s architecture is shown in Fig. 2.11. Each machine in a server rack is connected to a central I/O director that handles the virtualization of storage devices and network connections. To connect the server to the machine, either existing Ethernet ports with up to 10 Gbps or faster InfiniBand ports with up to 40 Gbps can be used. All requests can be served by a single connection, though a redundant second connection can be added for availability.

The I/O director switches all machines’ requests and forwards them to the appropriate networks or storage devices. It is therefore connected to the network or SANs via Ethernet or to disk arrays directly via fibre channel.

The virtualization of storage and network is transparent to the operating system, where special drivers help to access virtual host base adapters (vHBAs) as well as virtual network interface controllers (vNICs). Virtual interfaces can be moved, created or dropped, and modified in the running system. The interfaces are even consistent when the (virtual) machine itself is moved, MAC addresses stay the same. It is hence possible to migrate virtual machines without affecting the storage system at all and without any need to propagate the changes to connected software applications. In order to control the service quality of virtualized storage and network, QoS parameters can be configured in the I/O director, e.g., to allow for a certain bandwidth.

⁹<http://www.xsigo.com/>.

¹⁰<http://www.oracle.com/us/corporate/acquisitions/xsigo/index.html>.

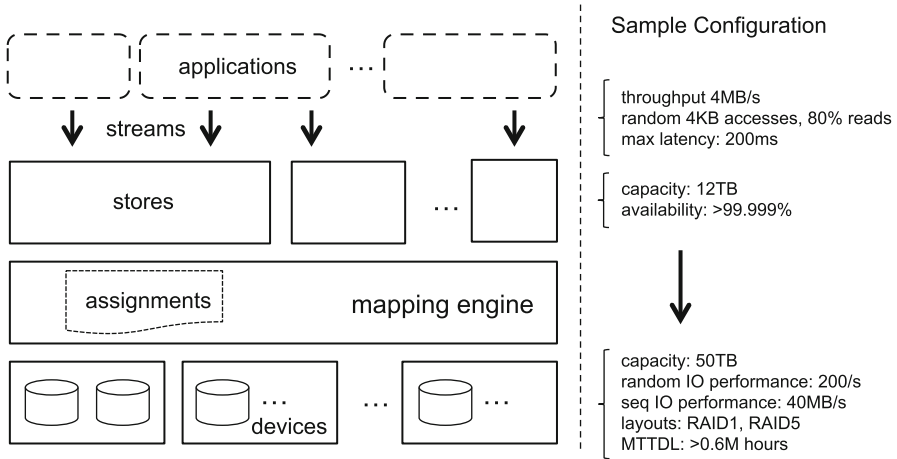


Fig. 2.12 The mapping problem for an attribute-managed storage system [42]

Storage System Organization and Configuration

When storage is separated from the actual machines and consolidated in large disk array systems, the question of how to manage such beasts quickly arises. This includes questions like how to build LUNs (on which disk, what size, ...) or how to choose RAID subsystems and the appropriate RAID levels. Hewlett Packard's Storage System Program (SSP)¹¹ tackled many of these questions in the last two decades (partly related to data management workloads, partly agnostic to the specific workload). A recent work by John Wilkes [42] provides a good overview of the different concepts and components that have been developed in the SSP.

The mapping problem, that is fundamental to storage management, is shown in Fig. 2.12. On a very abstract level, applications that run in the system generate I/O load – dubbed streams – that is directed to virtual storage containers, called stores. These containers on the other hand are mapped to actual devices like disks or disk arrays. The challenge is to find a mapping that satisfies all applications' I/O needs and at the same time fulfills certain optimality criteria like minimizing the number of disks needed and hence reducing costs.

One key concept is to annotate streams, stores, and devices with their characteristic attributes and requirements. Attributes associated with a stream capture, e.g., the dynamic aspects of the workload like the rate at which data is accessed or whether data is read or written. Furthermore, the I/O workload that builds a stream may be associated with attributes like the average request size and whether requests are random or sequential in nature. Attributes associated with stores capture

¹¹<http://www.hpl.hp.com/research/ssp/>.

```

{ store store1 {                                     # a 100GB store
  { capacity 100e9 }                                # mapped to a logical unit
  { boundTo array4.lu_3 }                            # on an array (not shown)
}}

{ stream stream1 {                                   # a stream
  { boundTo store1 }                                 # bound to that store
  { source host_A1 }                                 # originating at this host
  { interArrivalTimeOpen {                           # inverse of request rate
    { datamodelNormal best {                         # normal fit
      { mean 0.83e-3 } { stddev 0.6e-3 }             # mean = 1200/sec
      { chiSquare 0.7 }                             # goodness of fit metric
    }
    { datamodelExponential poor {                   # exponential fit
      { mean 0.83e-3 } { chiSquare 0.2 }             # 1200/sec, less-good fit
    }
  }
}}
{ requestSize {                                     # a simple behavior
  { datamodelUniform {                               # uniform size in 4-12KiB,
    { mean 8192 }                                     # on 1024-byte boundaries
    { lbound 4096 } { ubound 12288 } { granularity 1024 }
  }
}}
{ responseTime {datamodelExponential {mean 50e-3}} } # a goal
{ stream read {                                     # just the read requests
  { filteredBy { opType read } }
  { interArrivalTimeOpen 1e-3 }                     # 1000/sec
  { requestSize 9216 }                               # larger requests on avg.
}}
{ stream write {
  { filteredBy { opType write } }
  { interArrivalTimeOpen 5e-3 }                     # 200/sec
  { requestSize 4096 }
}}
{ stream degraded {                                 # something not right
  { filteredBy { { outageDuration 3600 }             # 1 hour at a time
    { outageFraction 0.002 } } } # 17 hours/year
  { interArrivalTimeOpen 1.67e-3 } # 600/sec
  { stream write {
    { filteredBy { opType write } }
    { interArrivalTimeOpen 0.1 } # 10/sec
  }
}}
{ stream broken {                                   # completely stopped
  { filteredBy { { outageDuration 300 }               # 5 min at a time
    { outageFraction 0.00001 } } } # 5 min/year
  { interArrivalTimeOpen inf }                     # nothing: 0/sec
}}
}}

```

Fig. 2.13 A (much simplified) sample workload specification example [41]

the requirements of these virtual containers, such as how much capacity they must provide, and their desired availability. Finally, devices, i.e., the actual disks, have attributes that capture their capabilities – capacity, performance, reliability, or cache behavior. Figure 2.13 shows a sample workload specification, i.e., annotations for stores and streams. One store is accessed by one stream. In normal mode, it receives 1,200 requests/s; in “degraded” mode, it can limp along for an hour at a time at half that rate; and it can be “broken” (non-accessible) no more than 5 min a year

(“five nines availability”). For simplicity, most of the data models shown are simple numeric values; in practice, distributions would normally be used.

In order to solve the mapping problem and to guarantee performance and availability of the data access, a mapping engine needs to take care of the following aspects: map stores to (possibly redundant) devices (or groups of devices) and configure devices with the appropriate RAID level. Different orthogonal questions (or demands) may be asked during the optimization process like: “How many devices are needed to support this load?” or “How much load can this set of devices support?”. Hence, once implemented, the mapping engine can be used to achieve different optimization goals.

The possible solution space for a mapping from stores to devices is huge and as such prohibits searching it exhaustively for the optimal solution (the problem is actually a variant of the multi-dimensional, multi-knapsack problem and as such NP-complete [42]). Solvers that intend to solve the constraint-based optimization problem apply greedy search, advanced heuristics, or speculative exploration in order to find near-optimal configurations after a reasonable amount of time.

2.4 Schema Virtualization

While hardware virtualization offers techniques for the lower part of the virtualization stack, this section focuses on the virtualization techniques for the upper part of the stack. Usually, “Software-as-a-Service”-providers host the same type of application for many users or tenants. For example, a webmail service such as Google Mail or Yahoo Mail provides the same email application to millions of users. Although managing their private emails isolated from each other, all webmail users manage the same type data, i.e. data of similar structure. As a consequence on the database layer, many tenants in “Software-as-a-Service”-infrastructures use the same or a very similar database structures or database schemas. Besides similar metadata, many tenant-specific applications partially work on shared data, stored once and used in multiple applications. In webmail services, for instance, users, especially from the same social group, may have overlapping address books. Other typical cases of common data are currencies, currency conversion rates or topographical entities such as nations or cities. Therefore, it is important for the data management layer that many of the tenants’ entities (or tables in a classical relational context) may exhibit not only a similar structure but also same sort of overlapping content.

The promise of “as-a-Service” is to lower the total cost of ownership for each tenant compared to an on-premise installation operating the computing infrastructure individually and locally at a customer’s site. Within the database, the noticeable overlap of data and metadata of tenants offers the opportunity to lower the cost of data management by keeping only one copy with different views for the specific applications. Depending on the scenario, a multi-tenant database may therefore be significantly smaller than the sum of single tenant databases [23]. However,

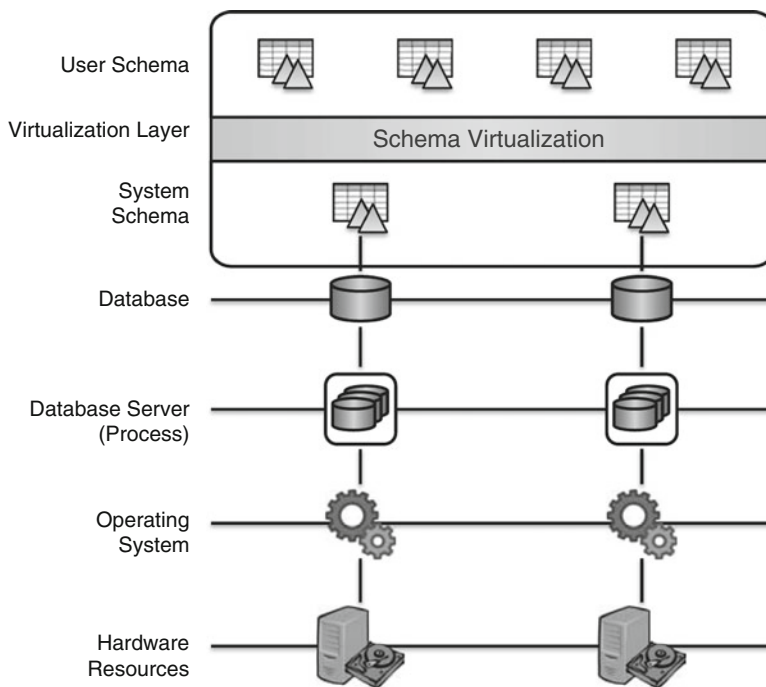


Fig. 2.14 Class 4 Virtualization scheme (SHARED TABLES) – schema virtualization

consolidating the shared data and metadata of multiple tenants into one single database requires a virtualization layer that offers each tenant a private and isolated view of their respected data. Through this virtualization layer, a tenant sees the shared database like a private database. As the actual database schema becomes transparent to the tenant, such a virtualization is called **SCHEMA VIRTUALIZATION**.

Following the basic scheme of the virtualization stack, schema virtualization is the Class 4 Virtualization Scheme (SHARED TABLES). Within this section, we further detail on the overall properties of schema virtualization and present existing techniques to implement virtualized schemas.

As illustrated in Fig. 2.14, this class simulates individual database schemas. While the external view of the database remains unchanged, the complete database stack is shared, including the database, the database server, the operating system, and the hardware resources. This scheme corresponds to a “Database-as-a-Service”-model. Each simulated individual database schema plays the role of a database which can be used by applications without any detailed knowledge about the specific setup. Again, looking at the different aspects of isolation provides the best summary of the overall properties of this virtualization scheme.

- **Security:** Generally, schema virtualization provides a very low level of isolation. Disasters or security attacks to the database affect all tenants equally. Once an

attacker has gained access privileges for the database, the attacker may also have access to data of all tenants. More specifically, sharing metadata, i.e., sharing a table structure requires the management of isolation on tuple level. A corrupted private tuple affects only the tenant owning the tuple. Sharing also data reduces the isolation to transactions. A corrupted shared tuple affects all tenants that share the tuple. In such a case, the transactional isolation determines how quickly transactions see a tuple after its corruption. The same holds for shared access data, such as index entries. On the positive side however, the general cost for security and safety are shared among all tenants: an average tenant puts its data into a system with more security features managed on a highly professional level than the tenant could afford on premise. In the same way, all security updates are to the benefit of all tenants in a single operation. However, with respect to explicit security attacks, the effective security may not increase, because a system with many thousand tenants is much more worth to attack than a system serving only a single user. Hosted serves draw the attention of more attackers and – very often – these attackers are willing to invest more in order to gain access to the database.

- **Performance:** Since all tenants operate on the same database, they influence each others performance. Besides the simple competition for computing resources among the tenants, the performance of a tenant can suffer because of transactional isolation of tenants. The number of transactions of different tenants influencing each other depends on the transactional isolation level and the degree of sharing on the level of metadata, payload data, or index data. Because the tenants share the cost of the system, they run either on a more powerful system which they would afford on their own or the system is running in an environment where performance bottlenecks can be reduced by moving tenants to different systems. Depending on the specific virtualization scheme, query processing for a single user may also touch irrelevant data, whenever queries are implemented by database scans. If the data is not clustered by tenants, each tenant loses data locality and has to pay the price for the “as-a-Service”-paradigm. Obviously, a table scan over the data of thousands of tenants is unacceptable costly for a single tenant. Therefore, data structures, especially access paths such as B-trees, scaling logarithmically with the number of entries are absolutely essential to achieve reasonable performance. Further, data statistics are less accurate for a single tenant because they represent the mean of all tenants. A “Database-as-a-Service”-provider may take special measures to limit and control the mutual performance influence among tenants. That way, the provider is able to offer different Service Level Agreements to the various tenants. For example, if a system holds some large tenants and a huge number of smaller tenants, the query optimizer may decide to use an index access for a specific query pattern, because the average tenant is small. Unfortunately, if the same query plan is executed for a large tenant, the query might end up in an enormous number of index accesses and an suboptimal plan.
- **Failure:** If a tenant’s application accidentally corrupts any metadata, payload data or access data that is shared, the failure affects all tenants. Failures that result in corrupted private data are not visible to other tenants. To prevent the impact of

a single tenant's failure on other tenants, a provider usually grants restricted rights on any shared data. For instance, tenants cannot simply drop a table, delete or update a shared tuple or remove an index. Obviously, if the underlying database stack fails, all tenants are affected. Since tenants share the costs, more failures prevention and compensation mechanism become affordable. The reliability of the hardware and database stack increases.

- **Manageability:** Since almost the complete database stack is shared, most administrative tasks, e.g., backup, recovery, tenant migration, user management or database software updates can be easily done for all tenants at once. Performing these tasks for single tenants, however, is difficult or impossible. For instance, most database systems support only backup and recovery of the complete database or tables as a whole. To back up a shared table for a single tenant, the backup process has to query the operational system to extract the tenant's specific data from the shared tables. If the used database system offers no explicit multi-tenancy support, management tasks of the individual tenants have to be implemented on top of the database system on application level. The same observation holds for constraints. Typically, a database system enforces a set of user-defined constraints to maintain data quality, integrity, and consistency. Without explicit multi-tenancy support, the database system can enforce only global constraints that hold for all tenants. Tenant-specific constraints have to be enforced on application level.

2.4.1 *Sharing*

Using schema virtualization, the data of many tenants is stored in a single database schema. On top of the database schema the virtualization layer simulates private schemas for every tenant. These private schemas exist only virtually. Regarding to scope of shared content of the tenants, we may distinguish four schema virtualization patterns with an increasing order of amount of shared data:

- **Common Metadata:** In the case of shared metadata, tenants share the definition of a table, but not the content of a table. Each tenant has access to multiple tuples in the shared table. Each tuple, however, is accessed only by a single tenant.
- **Locally Shared Data:** The next level of locally shared data allows that tenants share the definition of a table and partially the content of a table. Each tenant has access to multiple tuples and some of these tuples can be also accessed by other tenants. Nevertheless, tuples are not shared among all tenants.
- **Globally Shared Data:** For globally shared data, tenants share the definition of a table and partially the content of a table. Each tenant has access to multiple tuples. Some tuples can be accessed by all tenants equally.
- **Common Data:** Tenants share the definition of a table and the complete content of a table. All tenants have access to all tuples.

Table 2.2 Schema virtualization patterns

Tenant	Tuple	Pattern
1	n	Common Metadata
m	n	Locally Shared Data
*	n	Globally Shared Data
*	*	Common Data

Table 2.2 lists the four patterns with the cardinality of the relationship between tenants and tuples for each pattern. In the following, we discuss the four patterns in more detail.

In the **Common Metadata** pattern, tenants share the definition of a table. The content of the same tenant tables are stored in a single system table, while the tenants interact only with their virtual tables. If a tenant inserts data into a virtual table, the tuples are associated with the tenant and inserted into the corresponding system table. In the system table, each tuple is associated with a single tenant. The tenant references the entries via the primary key of the virtual table as part of a composite primary key of the system table. If a tenant queries its private data, the virtualization layer rewrites the query. Each reference to a virtual table in a query is replaced with the corresponding system table. Additionally, the virtualization layer adds a selection of the tuples associated with the specific tenant, such that the each tenant only sees its own data. Figure 2.15a illustrates the process for the `Product` table of an online shop service and three tenants. Similarly, the virtualization layer masks every update and delete operation triggered by a tenant.

In the **Locally Shared Data** pattern, tenants additionally share tuples. Shared tuples are not only accessed by a single tenant, but by groups of tenants. For each locally shared table, the system manages the system table that contains the data and a access table consisting of tenant id and primary key of the shared table. The content of the access table determines which tenant is allowed to access which tuple in the system table. Joining the access table and the corresponding system table allows selecting the tuple associated with a tenant. For each tenant query, the virtualization layer rewrites virtual table references accordingly. Figure 2.15b illustrates the process for the online shop service scenario. Rewriting update and delete operation is more complicated. Before the actually update or delete operation on the system table happens, the virtualization layer has to poll the access table for the list of tuples the tenants is allowed to access. In a second step, the virtualization layer rewrites the selection predicate of the update or delete operation. The primary key of the requested data entries has to be in the list of accessible tuples. For a list L of accessible tuples, a given predicate p and a primary key k , the virtualization layer rewrites p to $p \wedge k \in L$. The pattern can be easily extended to more access control. With another column in the access table, the virtualization layer can manage different access privileges of the tenants. The Locally Shared Data pattern is the most general pattern; it can represent all four patterns at once. On the downside, locally shared data in combination with the access table induces the most management overhead.

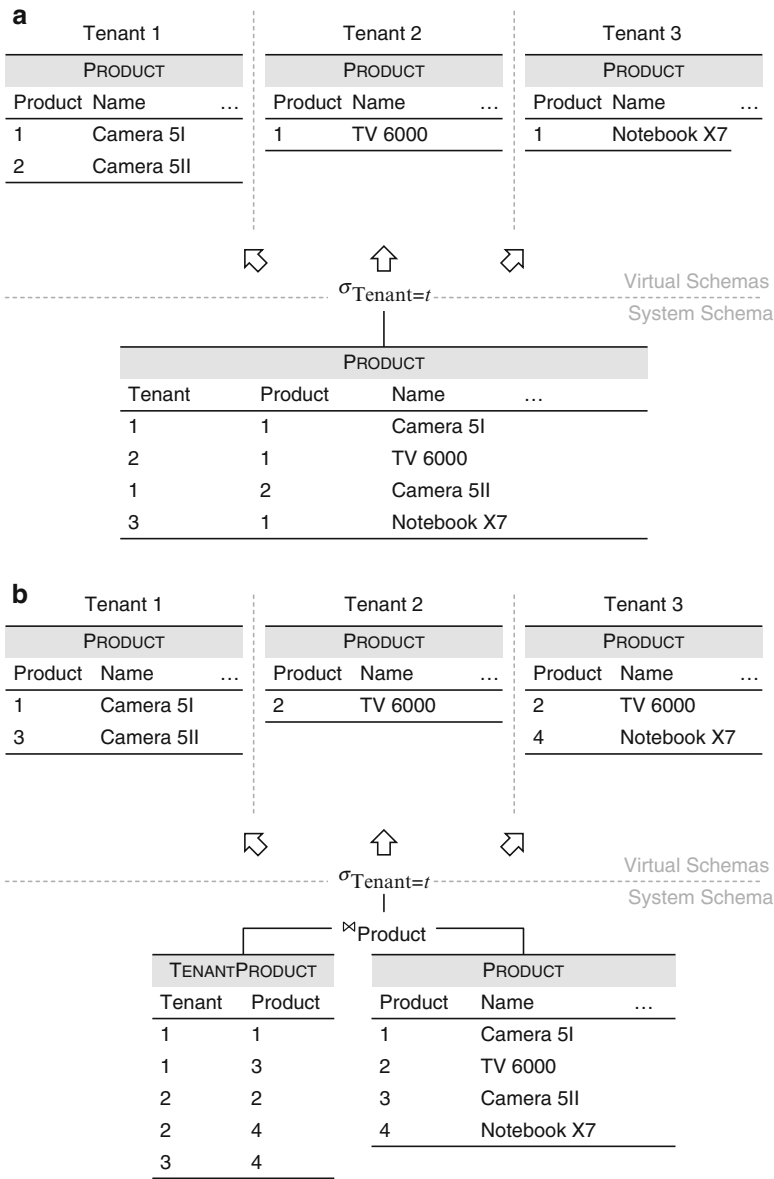


Fig. 2.15 Schema virtualization patterns detailed. (a) Common metadata. (b) Locally shared data

In the **Globally Shared Data** pattern, shared tuples are always shared among all tenants. Similar to the Common Metadata pattern, each tuple in the system table is associated with a single tenant, who has access to the tuple. All globally shared tuples reference a special system tenant. If a tenant queries its virtual table, the

virtualization layer filters the system table for all tuples that reference the tenant and the system tenant. Figure 2.16a shows the pattern in the online shop service scenario. Here, the system tenant has the tenant id G , implying that the product TV 6000 is visible in the virtual product table of all three tenants. Update and delete operations are rewritten by the virtualization layer in the same way. With multiple system tenants, the pattern can be easily extended to manage different access privileges for the shared tuples.

In the **Common Data** pattern, all tenants share a complete table. This is useful for topographical entities such as cities, nations or currencies. Again, the system manages the tuple in system table. Since every tenant has access to all tuples, the virtualization layer replaces references to the tenant's virtual tables with the system tables as show in Fig. 2.16b. Similarly, the virtualization layer reroutes update and delete operations if tenants are allowed to change the common data. The Common Data pattern requires the least management overhead.

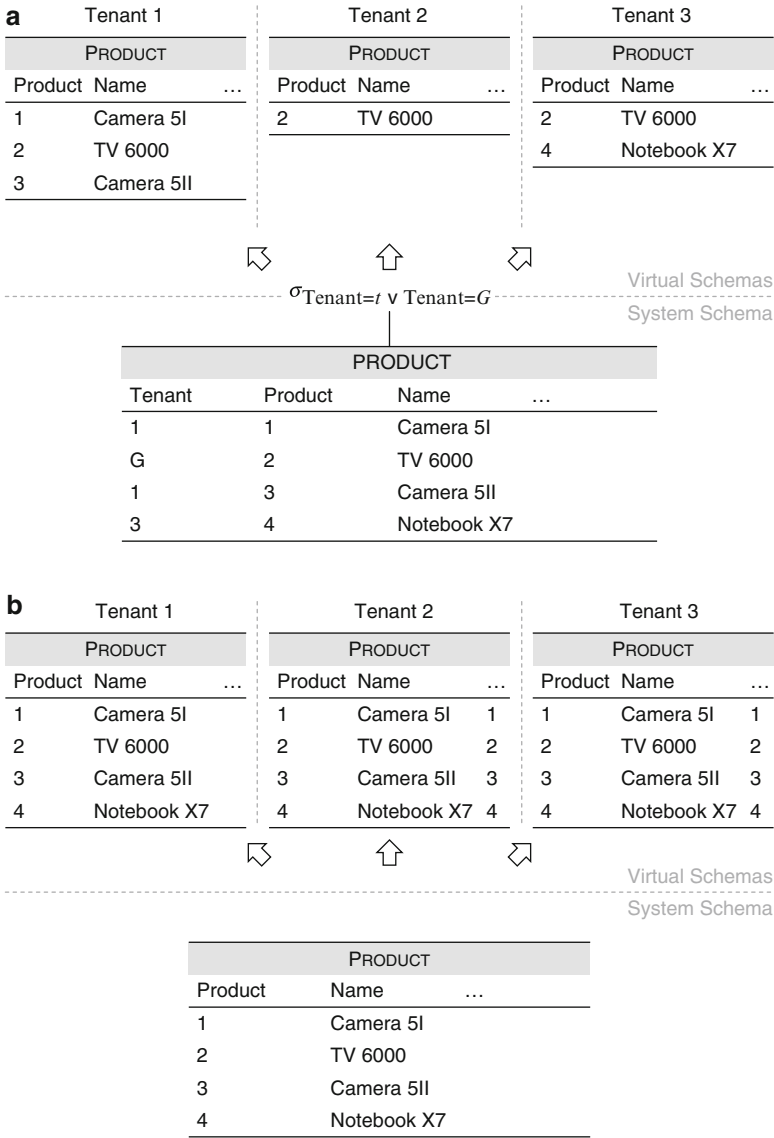
The four basic virtualization patterns describe how metadata and payload data can be shared among multiple tenant. Consolidating different tenants' schemas in a single system schema requires that the tenants use the same schema. However, in business applications, which are more complex than the schema of an email service, this situation is very unlikely the case. The next section discusses techniques to represent varying tenant schemas in a single database schema.

2.4.2 Customization

Real business applications such as Customer Relationship Management are extremely complex and require customization during installation for the individual tenants. Customization implies that different tenants may have a different view on the content of the underlying database infrastructure. The customization of an application often includes schema adjustments on database level but may also reach as far out as changing common data. For example in a web shop scenario, a retailer wants to extend the product table with attributes specific to the product the retailer is offering: Wine bottles have different attributes then TVs or books. A Software-as-a-Service provider must be able to customize its application for every single tenant without affecting other tenants. This section discusses seven techniques that allow customization of virtual schemas.

Extension Table

A big part of customization can be carried out in modules, extensions, or add-ons. Such extensions add well-defined functionality to the service and are offered by the service provider. Tenants can book extensions to adapt the service to their needs. Since the service provider is in full control of the offered extension, the provider can model the system tables of the database accordingly. Decomposing the system tables



Tenant 1		
PRODUCT		
Product	Name	...
1	Camera 5I	
2	TV 6000	
3	Camera 5II	
4	Notebook X7	

Tenant 2		
PRODUCT		
Product	Name	...
1	Camera 5I	1
2	TV 6000	2
3	Camera 5II	3
4	Notebook X7	4

Tenant 3		
PRODUCT		
Product	Name	...
1	Camera 5I	1
2	TV 6000	2
3	Camera 5II	3
4	Notebook X7	4

↙

↑

↘

Virtual Schemas

System Schema

PRODUCT		
Product	Name	...
1	Camera 5I	
2	TV 6000	
3	Camera 5II	
4	Notebook X7	

Fig. 2.16 Schema virtualization patterns detailed (cont.). (a) Globally shared data. (b) Common data

avoids NULL values in the extension columns [12]. The base table encompasses the columns every tenant needs. The columns of an extension are combined in an extension table. Each tuple in the extension table references its corresponding tuple in the base table. If a tenant has not booked an extension, no tuple in the corresponding extension table belong to the tenant.

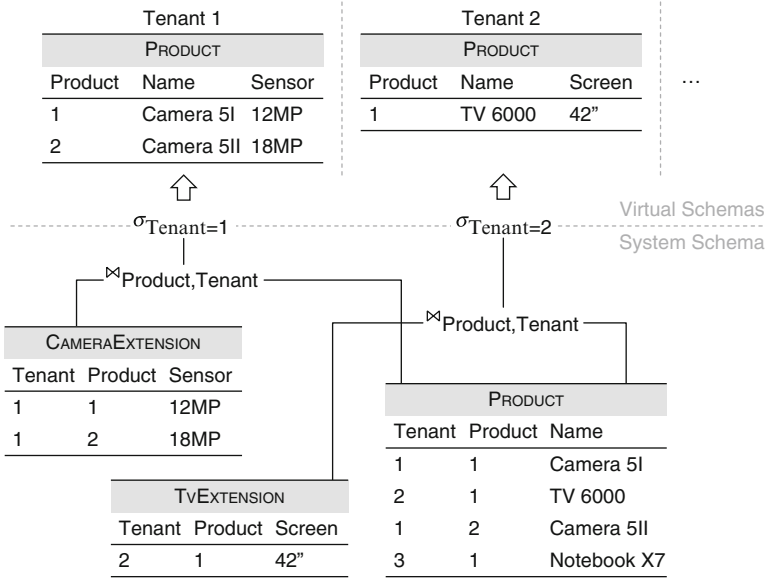


Fig. 2.17 Extension tables

The virtual table of a tenant exhibits the column of the base table and the column of all extensions the tenant has booked. If the tenant queries its virtual table, the virtualization layer selects the tenant's tuples from the base table and joins them with all booked extension tables. This way, the tenant sees its data including all booked extensions without seeing empty columns of not booked extensions. Figure 2.17 illustrates the querying procedure. Note that the example shown in the figure assumes the Common Metadata pattern. Insert, update, and delete operations have to occur on the base table and all booked extensions. Insert operations can be simply routed to all involved tables by decomposing the inserted tuples into base columns and extensions. Update and delete operations with a predicate require the virtualization layer to determine which tuples the operation affects. Therefore, the virtualization layer probes all involved tables with the applying parts of the given predicate. Depending on the predicate, the virtualization layer combines or intersects resulting lists of tuples into a single list. Finally, the operation is routed to all involved tables.

With extension tables, the data resides in a well modeled system schema. Hence, most features of the database system such as constraints, indexes, or aggregation are usable on the level of the system table without any modifications. The additional joins required to answer queries impose overhead for the virtualization layer. If the number of extensions per table and tenant are small, this overhead remains moderate. On the downside, extension tables allow customization only in relatively coarse-grained, predefined modules. Fine-grained extensions strongly decompose the system tables to tables with a single payload column in the extreme

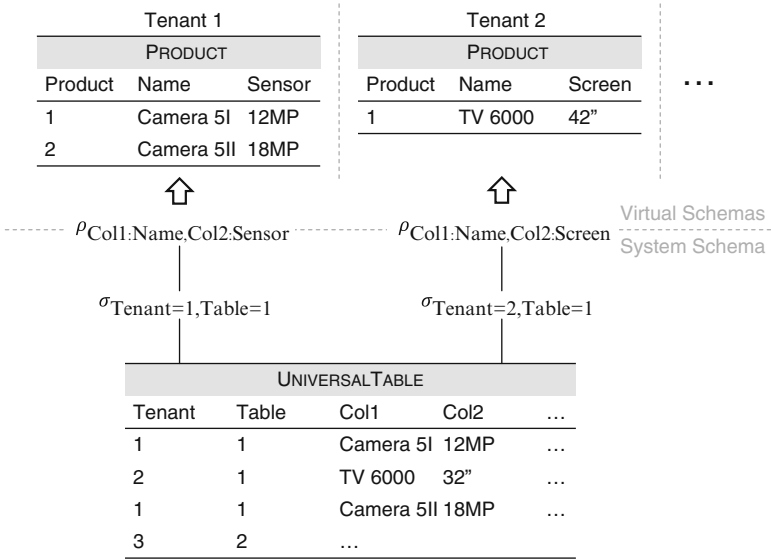


Fig. 2.18 Universal table

case. Although tenants can easily add new extensions without any change of the system schema, offering new extension requires changing the system schema. New extensions add new system tables. If many extensions are only booked by a single tenant, the consolidations effects decline. The extension table approach also has been used to map object-oriented inheritance to the relational model [8, 36].

Universal Table

A generic system schema allows consolidating arbitrary virtual schemas. The universal table is one possible generic system schema. Here, the system schema contains a single table consisting of a fixed number of generic byte string or char string columns [26]. Each tuple of the universal table corresponds to one tuple in a virtual table of a tenant. The system associates each tuple with the tenant and the virtual table it belongs to. Furthermore, the system relates the columns of each virtual table to the generic columns in the universal table while the virtual table is created. The mapping between virtual tables and universal table is stored in the system catalog. Whenever a tenant inserts a tuple into a virtual table, the virtualization layer maps each value to the corresponding generic column. Note that the values have to be cast to the technical type of the columns of the universal table.

If a tenant queries a virtual table, the virtualization layer rewrites the query to the universal table. Each table reference is replaced with the universal table and all column references are replaced with their corresponding generic column. Additionally, the virtualization layer has to wrap each column reference with a cast operation to map the values back to their original technical type. Figure 2.18

exemplarily shows the query rewrite procedure for the universal table (except the value casting). Analogously, the virtualization layer rewrites update and delete operations.

Since a generic schema does not reflect any application-specific data schema, it provides the most flexibility for customization. A tenant can adapt its virtual schema without affecting the system schema. The query rewriting procedure is simple and does not impose additional joins on a query. However, the necessary cast operations introduce a considerable overhead to the query processing. Further, the generically typed columns prevent direct constraints and index support; both have to be implemented on top. To mitigate the overhead, the universal table approach can be changed to technically typed columns. Instead of generically typed columns the universal table provides a fixed number of columns for each technical type. In any case, the number of columns has to be sufficiently high to fit every tuple in the whole system. As result, the universal table contains very wide rows with many NULL values, which imposes additional overhead. As reported, Salesforce successfully uses some variation of the universal table approach within its Force.com platform [40].

Pivot Table

The pivot table represents another generic system schema. The pivot table, also known as vertical schema, is the relational version of a triple store. It consists of five columns and contains a tuple for each value in all virtual tables. A single generically typed column contains the values; the other four columns reference the tenant, the virtual table, the virtual column, and the virtual tuple the value belongs to [2, 14]. Table names and column names can be stored directly in the pivot table, which, however, produces a heavy redundancy of these names. Hence, table names and column names are usually normalized into additional catalog tables.

If a tenant inserts a new tuple with n values into a virtual table, the virtualization layer inserts n new tuples into the pivot table. When tuples are queried, the virtualization layer has to reconstruct the originally inserted tuples. For every virtual column of a virtual table reference in a query, the virtualization layer queries the pivot table and joins the results by tenant, table, and reference to the virtual tuple. Figure 2.19 illustrates the process. Predicates on a single virtual column can be evaluated before the join; more complex predicate have to be evaluated after the join. An alternative way of reconstructing tuples is a conditional projection with grouping. Here, the virtualization layer projects every value to its virtual column and groups the fanned out tuples by tenant, table, and reference to the virtual tuple. Each group represents one virtual tuple and contains a single value in each of the value columns; the remains are NULL values. During grouping, the virtualization layer aggregates each column to the single value it contains. Update and delete operations follow the two stage model, again. First, the virtualization layer determines which virtual tuples need to be updated or deleted. Second, it performs to actual operation.

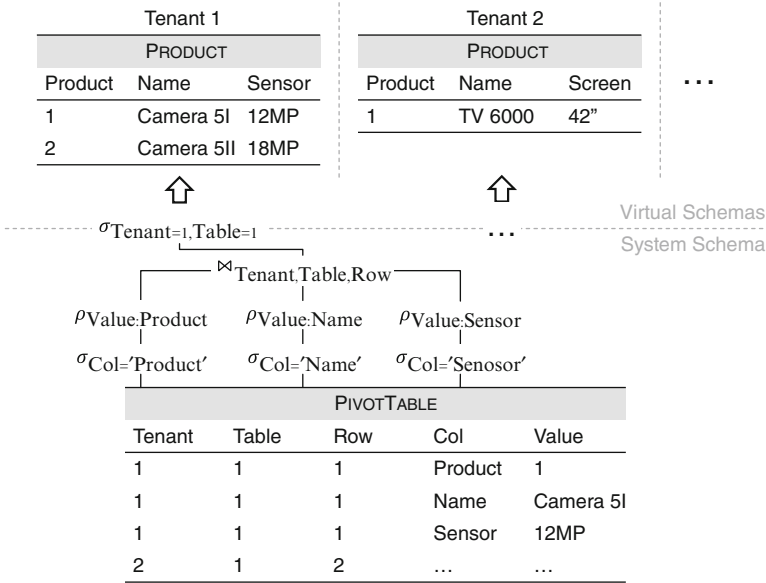


Fig. 2.19 Pivot table

In both querying rewriting schemes, however, the values have to be cast to the respective technical type (not shown in Fig. 2.19). Updates on a virtual table require one update of the pivot table per updated virtual tuple and updated virtual column. Delete operations require one delete in the pivot table per deleted virtual tuple and column in the virtual table.

Like the universal table, the pivot table allows tenants to customize their virtual schema without affecting the system schema. The pivot table avoids NULL values, to the price of considerable processing overhead. Querying a virtual table with n column requires $n - 1$ self-joins on the system table. The generically typed columns prevent direct constraints and index support. Similar to the universal table, this can be mitigated by partitioning the pivot table by technical type. Here, the system schema encompasses a pivot table for every technical type. This avoids casts and allows indexing. Pivot tables are also used successfully in clinical systems, e.g., to store the multifarious and varying examination data collected for a patient during consultation and surgery [20]. Another application of the concept maps XML to a pivot table for fast evaluation of XPath queries [21].

Chunk Table

The chunk table approach combines the universal table with the pivot table. While the number of different virtual columns is very high, the number of technical types is low and not higher than in a single tenant system. With only a small number of

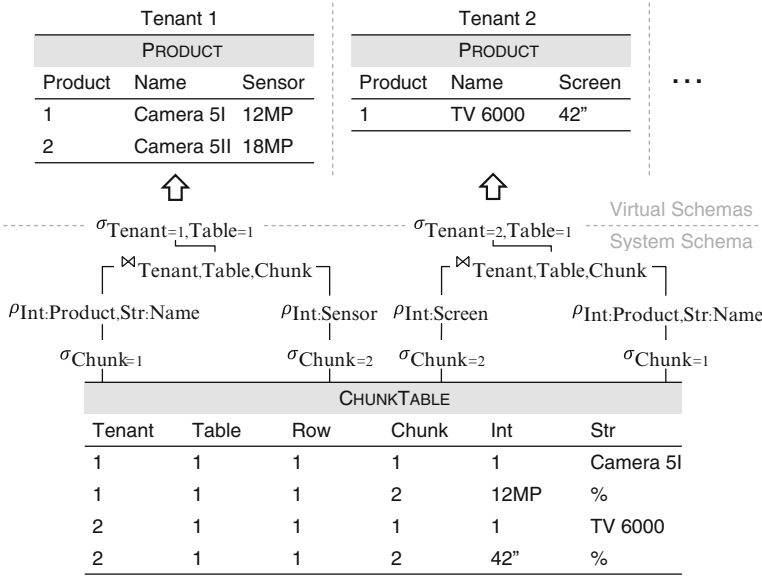


Fig. 2.20 Chunk table

technical types, certain combinations of technical types can be found across many virtual tables. Many tuples may consist of, e.g., integer – string pairs or string triples. Chunk tables decompose the tenant data into chunks of these frequent technical type combinations. The system schema contains for each frequent combination a chunk table. It contains the respective chunks from tenant tuples accompanied by a chunk number and references to the tenant, the virtual tables, and the virtual tuple. A tenant tuple can be spread across different chunk tables and multiple chunk in the same chunk table. The system maintains a mapping for each virtual table. Each column is mapped to a defined chunk table and chunk number. This allows reconstructing the tenant tuples from the chunk tables [3].

If a tenant queries its data, the virtualization layer rewrites the query to the chunk tables. The rewriting process is a mixture of the rewriting for the universal table and the rewriting for the pivot table. As for the universal table, the virtualization layer has to replace a reference to virtual columns with the corresponding column in a chunk table. Similar to the pivot table, the system has to self-join a chunk table if it contains multiple chunks of the same virtual table. Each branch of the self-join selects chunks with a specific chunk number. If the queried virtual table spreads across multiple chunk tables, these have to be joined to. All chunk table access branches are joined by the tenant, the virtual table, and the virtual tuple reference. Similar to other approaches, the virtualization layer also adds filters for tenant and virtual table. Figure 2.20 illustrates the rewriting process. For clarity, the virtual product tables in the figure are mapped to a single chunk table only. The rewrite process for update and delete operations is analogous to the pivot table.

The virtualization identifies the affected chunks and then routes the operation to every chunk table. Since all value columns are technically typed, neither queries nor manipulation operations require casting.

Chunk tables form a semi-generic system schema. Tenants cannot change their virtual schema arbitrarily, though. Any virtual table has to fit on the available chunk tables. Nevertheless, this approach provides considerably more flexibility than extension tables. For full flexibility, the system requires a strategy how to handle the cutoff chunks that do not fit any of the existing chunk tables. Two strategies are reasonable. First, the system can store cutoff chunks in a full-generic table structure such as a universal table or a pivot tables to the price of a twice as complex virtualization layer. Second, the system maps cutoff chunks to the most similar of the available chunk tables to the price of NULL values (as shown in Fig. 2.20). The crucial point is the design of the chunk tables. With well selected chunk tables, the approach achieves full flexibility and technically typed values with a bearable number of required joins and a negligible small need for NULL values. The technical typed columns avoid expensive casts and allow index support. Because different virtual columns are consolidated into one system column, constraints have to be implemented on top, however.

Interpreted Column

If customization concentrates on augmenting a well-modeled database schema with a moderate number of additional columns, the interpreted column approach offers an easy solution. The system schema matches the common core schema, except that each extendable table contains an additional text column. Without any customization the column remains empty. If a tenant adds columns to a virtual table, the virtualization layer serializes the data in these columns to text and stores it in the text column. For each tuple, the serialization contains the values of the custom columns plus a reference to the column definition [1, 19]. The specific process of serialization of the data is independent of the principles of the approach as long as all accesses use the same technique. Usually, other data models with a defined text serialization, such as XML [38], JSON [13] or CSV [30], serve the purpose.

If a tenant queries data, the virtualization layer rewrites the query to the system tables according to the used sharing pattern. Before handing the result to the tenant, the system de-serializes the text column and projects the contained values to their respective virtual columns. Figure 2.21 illustrates the basic process in case of XML. In the example, the XPath expression [37] applied to the XML column marks the de-serialization. On a pure relational database system, predicates on the custom columns have to be evaluated by the virtualization layer. However, some database systems offer direct query support for complex values in columns typed by a secondary data model. For instance, IBM DB2 supports XML columns [9, 28], which can be directly queried with XQuery [39]. In such a case, the supported data model is strongly preferable, because the virtualization layer can rewrite predicates on the custom columns to the specific query capabilities pushing the evaluation

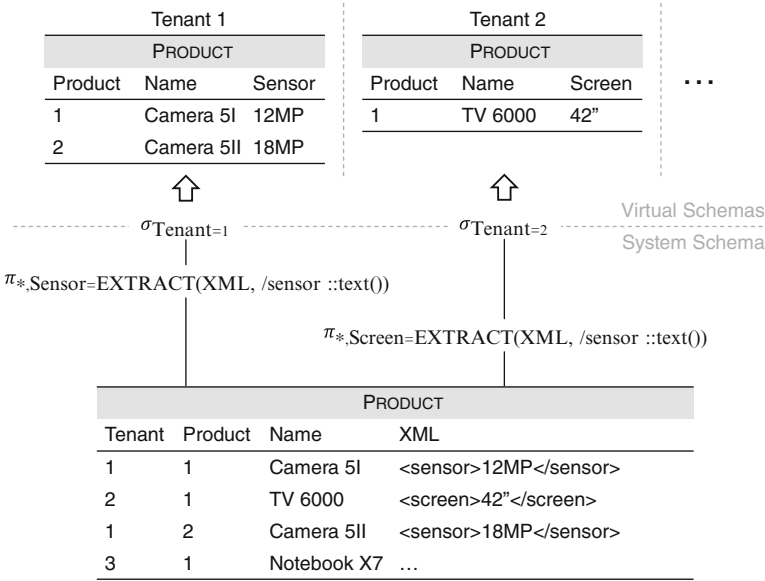


Fig. 2.21 Interpreted column

of these predicates down to the data. Similarly, the database system performs update and delete operations on custom columns directly if it offers the necessary capabilities. On a pure relational database system, the virtualization layer helps out. Following the two stage model, the virtualization layer queries the affected tuples and then executes the operation on the base table. For update operations, the first stage is necessary if the tenant updates values in custom columns, because the virtualization layer has to de-serialize the text column value to perform these updates. Before the virtualization layer writes the tuples back to the system table, it serializes the custom columns again. For update and delete operation the first stage is also required if the tenant specifies the affected tuples with predicates on custom columns.

The interpreted column approach is appealing, because it provides flexibility to add custom columns without sacrificing capabilities of the database system for the common part of the database schema. If the database system supports complex values such as XML including query and update capabilities, the virtualization layer remains very lean and the only overhead arises from the serialization and de-serialization. Without database system support, the necessary serialization and de-serialization within the virtualization layer imposes additional overhead in shape of extra data movement between database system and virtualization layer. The same holds for constraints and indexing. If constraints or indexing is not supported by the database system but required on custom columns, it must be implemented on top in the virtualization layer.

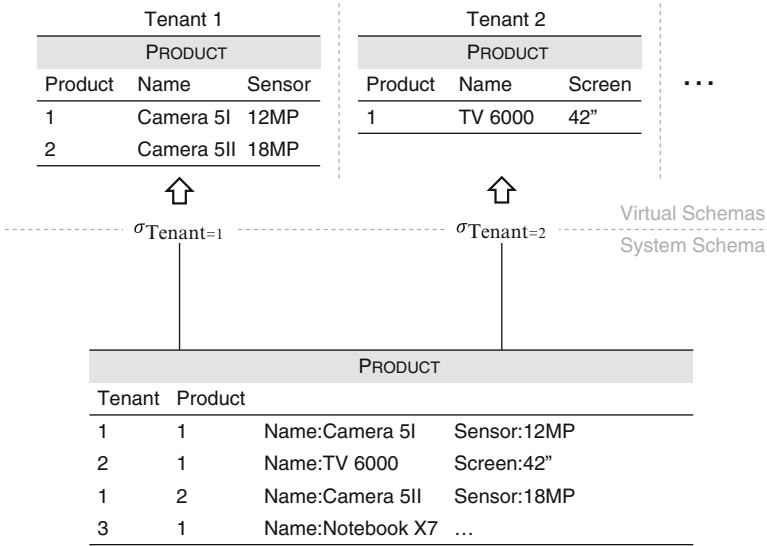


Fig. 2.22 Interpreted record

Interpreted Record

The interpreted record concept implements support for flexible structured tuples in a relational database system without the detour of a secondary database model. Instead of storing only the values of a tuple in its record, an interpreted record explicitly marks every value with a column reference. The interpreted record omits NULL values as representation of not instantiated columns and stores sparse tuples efficiently. Further, the user can add new columns to an interpreted record table easily. None of the tuples existing in the table needs to be touched. The database system adds the new column only to the system catalog, that new tuples or updated tuples can reference the column [6, 10].

With interpreted record support in the database system, the virtualization layer only realizes the sharing pattern. The flexibility required for customization is directly provided by the system tables. If a tenant adds a custom column to a virtual table, the virtualization merely routes this operation to the system table. If a tenant drops a column, the virtualization has to check if other tenants use the column before it drops the column on the system table. Queries, inserts, updates, and deletes are handled by the virtualization layer according to the used sharing pattern; no additional processing is required, as illustrated in Fig. 2.22.

The interpreted record concept convinces with no extra overhead in the virtualization layer and full support for constraints and indexing. In disk-bound database systems the additional expense of interpreting every record individually is negligible. The gained flexibility is limited to customization of the table structure, i.e., adding and dropping tables. The concept does not allow the consolidation of complete individual tenant schemas, as universal table and pivot table do.

Polymorphic Table

All approaches discussed so far consider only schema customization, which is perfectly suitable for metadata sharing. However, if the tenants share data, tenants may want to customize the shared data as well. Polymorphic tables [5] combine sharing and customization of metadata and data in a single concept. Generally, customization is the process of deriving a specialized table from a base table offered by the service provider. The tenant may add columns and may add or update tuples. Customization is similar to object inheritance (or specialization) [17]. The custom table of a tenant inherits structure and data of the table it customizes. Tenants derive their virtual tables, by customizing base tables or specialized extension tables offered by the service provider. The multiple customizations of the same base table form an inheritance hierarchy. A polymorphic table represents such an inheritance hierarchy. The system database is the collection of all polymorphic tables.

Each tenant table is a leaf node in the inheritance tree of its corresponding polymorphic table and has an inheritance path to the base table. The inheritance path of a table segments the table into the stages of its customization. Metadata customization slices the table vertically into base schema, extension schemas, and tenant schema. Data customization slices the tables horizontally into shared base data, shared extension data, and private tenant data. The table header decomposes into fragments; the table body decomposes into segments. Each segment schematically complies with a specific fragment. Beside the columns defined by its fragments, the segment contains the primary key columns of the polymorphic table. The polymorphic table manages all fragments and segments in the inheritance hierarchy of the customized tables.

If a tenant accesses a virtual table, the virtualization layer traverses the inheritance hierarchy of the polymorphic table from the node that represents the queried table up to the root node. While traversing the hierarchy, the virtualization layer collects the fragment information and the segment references. All collected fragments are concatenated to the schema of the virtual table. The segments form the content of the table. The virtualization layer reads every segment and unites the tuples of all segments from the same fragment. Finally, the unions of all fragments are joined by the primary key to form the virtual table. Figure 2.23 illustrates the basic process for one polymorphic tables with one extension and two tenants. Remember that segments of similar structure, for instance `ProductBase`, `ProductBaseTenant1`, and `ProductBaseTenant2`, share their metadata by referencing the same fragment. Polymorphic tables are meant to be implemented within the database system. Nevertheless, the concept can also be realized on top of a database system. Then, segments of the same fragment reside in a dedicated system table or share a system table. Figure 2.23 shown the first case. Here, fragments with multiple segments are stored redundantly in the metadata of multiple tables. In the second case, the system table storing multiple segments requires an additional column to distinguish the segments.

A polymorphic table allows comprehensive customization of shared data beyond the addition of attributes. In such a case, the updated version of a shared tuple is

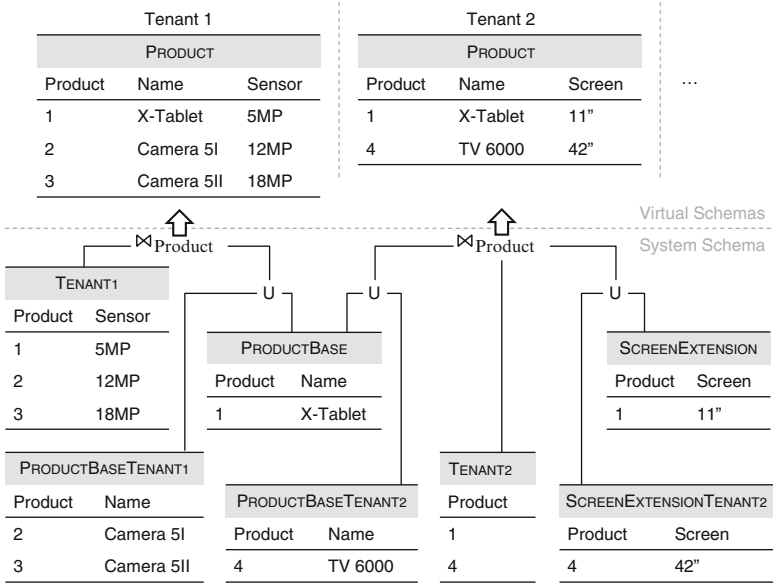


Fig. 2.23 Polymorphic table

placed in the lower segment that represents the customization step. During query processing, the virtualization layer additionally probes every tuple of a segment against the lower segments that belongs to the same fragment. The probing step is a left anti-semi-join of the higher segment with the lower segment on the primary key.

The polymorphic table concept covers multiple sharing patterns in one strike. Segments are the granularity of sharing. Base segments are shared globally among all tenants, extension segments are shared locally among a subset of tenants, and tenant segments are private to each tenant. The approach consolidates metadata and data as much as possible, while allowing very flexible customization of a table and its data. The processing overhead depends on the degree of customization. The deeper the inheritance tree of customizations gets, the more joins the processing requires. Similar to extension tables, the data resides in a well modeled system schema. Constraints, indexes, or aggregation operators are usable in a polymorphic table without any modification.

2.5 Configuring the Virtualization

Running an infrastructure based on virtual machines raises the issues of virtual machine *allocation* and *configuration*. Allocation addresses the issue of mapping a set of virtual machines to a set of given (potentially heterogeneous) hardware machines. Configuration focuses on the parameterization of the individual virtual

machines. Since the hypervisor is in control of the hardware and the scheduling of the computing resources with regard to the currently deployed virtual machines instances, the administrator is able to assign limits usually in terms of main memory and overall CPU ratio. Allocation and configuration goes hand in hand and is usually a rather static and (at least initially) time-consuming administrative activity. The goal is to find a “reasonable” good balance between system utilization and the SLA-based goals of throughput and delay. As of now, most virtualized system environments are performing allocation and configuration tasks manually resulting in a sub-optimal resource utilization and static assignment. Very often, administrators are applying heuristics by grouping applications with complete behaviorally opposites in terms of CPU or storage usage patterns.

Tool support, for example the VMware Vmotion product¹² is provided on the layer of virtual machines without considering application knowledge. For example, hypervisor monitors are proposing changes to the infrastructure by considering memory and CPU requirements plus low-level system-oriented metrics like reference counters or process waiting times. Such tools are therefore not able to exploit higher level knowledge potentially provided by higher-level software components.

The overall challenge in the context of creating an optimal deployment scheme for virtualized services is – on the one hand – to develop a design advisor to capture the dynamic behavior. A monitoring component should signal situations of a re-configuration or re-allocation based on high-level utilization characteristics specified within the SLA contracts. On the other hand a design advisor component is supposed to compile knowledge of the running applications into design recommendations. For example, if a design advisor has knowledge about the temporal access behavior, it should allocate services for peak loads at different times within the same computing environment (depending on the virtualization class). In the literature, the topic of optimizing the deployment scenario gained some attention as well. For example [29] outlined a techniques of feedback-controlled virtual machines in the context of HPC infrastructures to precisely predict the resource consumption and runtime-behavior evaluated on five widely-used HPC applications. Other approaches tackling this challenge from a more database-centric perspective. For example the NIMO project at Duke [31] learns application performance models based in active learning methods and subsequently uses this models to adjust the deployment of virtual resources. The database-specific properties stem from the fact that NIMO exhibits a special data profiler which is used to feed what-if analysis procedures to provide performance estimates for potential scenarios like performance improvements when doubling the memory bound for the database system.

¹²<http://www.vmware.com/products/vmotion/overview.html>.

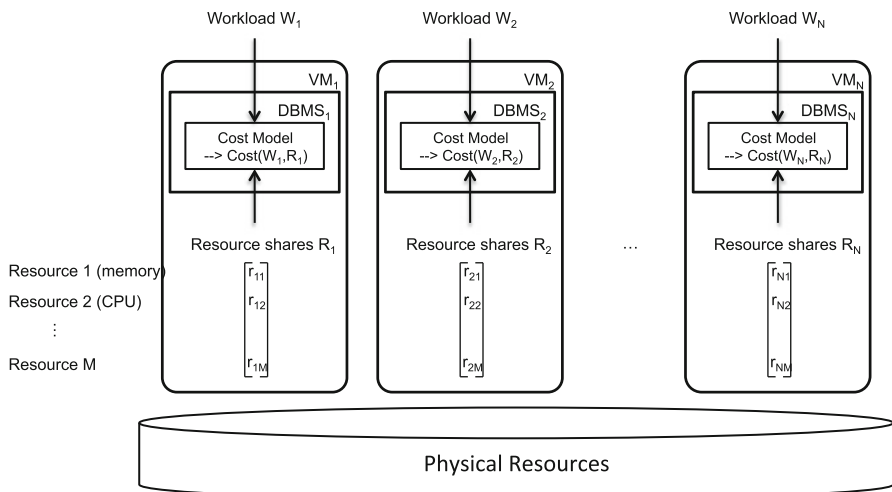


Fig. 2.24 Overview of the virtualization design advisor components

Virtualization Design Advisor for Database Services

One way to tackle the problem of incorporating application knowledge in a virtualization design advisor was proposed by Soror et al. in [33] and later more detailed in [34]. Since this is the first and most detailed work in this field, some of the main concepts are presented here to convey the basic idea and show how application knowledge is used to find and improve VM configurations.

The idea is to optimize Class 1 Virtualization Schemes by analyzing the anticipated workloads of all DBMSs, estimating the costs that occur from executing these workloads, and allocating resources to the virtual machines accordingly. The knowledge about the workload that is to be expected in the different virtual machines helps to, e.g., distinguish CPU intensive from I/O intensive workloads, and allows for optimal configurations. Treating the DBMS as a privileged application instead of a black box (with a simple performance model) enables the proposed virtualization design advisor to utilize the database management system's internal optimizer, especially its cost model, in a what-if mode to predict a configuration's performance with little overhead. Comparable to classical database advisors, specific configurations do not have to be implemented and tested in a time consuming experimental phase, but instead can be evaluated quickly. In contrast to classical database design advisors, the virtualization design advisor recommends the VM configuration instead of the DBMS configuration.

The general *virtualization design problem* that is introduced and solved by Soror et al. is formalized as follows (all components are shown in Fig. 2.24). Assume there are N virtual machines, running independent – possibly heterogeneous – database management systems, competing for a pool of shared physical resources on a single machine. Each workload W_i that is presented to the different virtual machines is

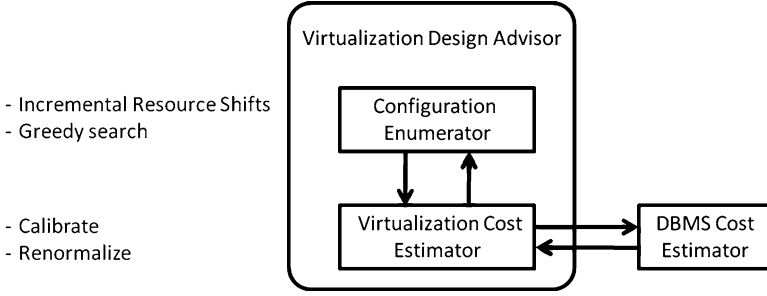


Fig. 2.25 Virtualization design advisor components

characterized by a set of SQL statements. Furthermore, there are M resources like CPU, memory, I/O or network bandwidth and each virtual machine gets a share $R_i = [r_{i1}, \dots, r_{iM}]$ (with $0 \leq r_{ij} \leq 1$) of these resources. Soror et al. concentrate on CPU and main memory as primary resources to provision as these two can be configured in all available virtual machine monitors. Having formalized workloads and configurations, one can denote $Cost(W_i, R_i)$ as the cost of workload W_i under resource allocation R_i . The virtualization design problem is then given as choosing r_{ij} ($\leq i \leq N, 1 \leq j \leq M$) such that

$$\sum_{i=1}^N Cost(W_i, R_i)$$

is minimized, subject to $r_{ij} \geq 0$ for all i, j and $\sum_{i=1}^N r_{ij} = 1$.

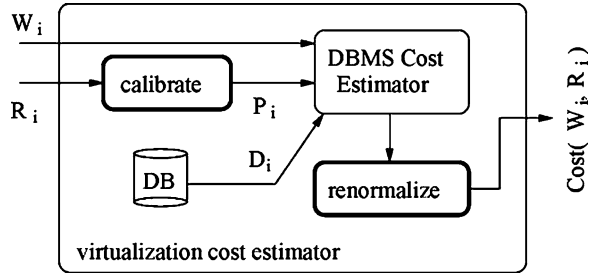
The rather general formulation of the problem can further be refined, e.g., by introducing the notion of degradation and limiting this degradation in order to prevent certain workloads from being discriminated too much or by adding weights to the costs of each virtual machine to prioritize certain workloads over others.

The proposed virtualization design problem is solved by the virtualization design advisor in two major components as is shown in Fig. 2.25: (1) a configuration enumerator that searches the solution space and (2) a cost estimator that predicts the cost for each configuration.

The configuration enumerator assumes a smooth and concave objective function (which is reasonable as is shown in several experiments) and applies a greedy search in the search space. The search is accomplished by starting with a fair share of all resources among all virtual machines and increasing or decreasing shares in increments of, e.g., 5 %. Each new configuration is evaluated and the best is chosen as the starting point for a new search. The search ends when no better configuration can be found.

The cost estimation component of the virtualization design advisor is illustrated in Fig. 2.26. It wraps the two methods *calibrate* and *renormalize* around the DBMS cost estimator as the key component that subsumes deep knowledge about the

Fig. 2.26 Cost estimation for virtualization design [34]



database management system's specific characteristics, methods, and behavior. Wrapping the cost estimator is required because the cost model cannot directly be applied to the given resource shares – which requires the calibrate method – and the cost model's results cannot easily be compared among different database management systems, which calls for the renormalization. Given that both methods require deep knowledge about the internals of the database management system, they need to be implemented (once) by an expert.

The calibration method takes the given resource shares R_i and maps them to a number of DBMS configuration parameters P_i . These parameters can either be *descriptive*, i.e., they describe the underlying system such that the DBMS understands its performance (e.g., `cpu_tuple_cost` or `random_page_cost` in PostgreSQL). Or parameters can be *prescriptive*, i.e., they define the configuration of the DBMS itself and thereby mimic rules or policies of the configuration (e.g., `shared_buffers` or `work_mem` in PostgreSQL). Both sets of parameters can be obtained by either applying policies and rules-of-thumb or by carefully running calibration queries and building simple models. Once obtained, the parameters can be applied and the DBMS's optimizer can be invoked with the expected workload on the database D_i in question. The database contains necessary statistics about the data and their distribution.

Because the costs that are estimated by DBMS optimizers are intended to compare different plans for the same query they do not need to be meaningful outside the DBMS. Hence, they are usually formulated in abstract units and cannot easily be compared among different database management systems. In order to compare them, they need to be normalized to a common unit like, e.g., consumed resources of a certain type or elapsed time.

In summary, the outlined technique presents a first step into the direction of automatically optimizing the deployment scenarios in Database-as-a-Service environments. Unfortunately, a comprehensive virtualization advisor requires to be aware not only of the characteristics and behavior of the individual layers but also has to incorporate the relationships and dependencies between the different layers. The core concept of layering software environments by shielding the working behavior from each other comes into a critical situation if a global optimization is required. Cross-cutting optimizations are still a hot topic for research projects with a certainly significant commercial relevance.

2.6 Summary

The principle of economy of scale demands to have multiple database users of tenants deployed on a shared system. The concept of virtualization plays a key role in this discussion. By definition, virtualization comprises the technique to hide physical or logical properties of computing devices by providing a virtual version of such a device (or computing entity). Unfortunately, virtualization also has two faces. On the one hand, clearly layering software stacks by means of abstracting from physical entities provides clear interfaces and makes the software stack easier to control and manage. On the other hand, finding cross-cutting optimizations for defining allocation and configuration schemes is extremely hard, if only the functional behavior of the underlying software layer is known. Within this section, we position the notion of virtualization as the key concept in order to provide “Database-as-a-Service”-functionality. We therefore introduced and discussed different classes with respect to the application of virtualization in a software stack, clearly distinguishing physical and logical virtualization. We also intensively described different techniques to achieve different requirements for “Data-Management-as-a-Service”-environments. For example, isolation of different users (or tenants) is highest, if virtualization is performed on a physical level. Alternatively, the overall system is flexible for a huge number of tenants, if database schemas or even some parts of the database content are shared. Having the notion of virtualization in mind, we may now dive into details on providing transactional or large-scale analytical data management services.

Additional related material can be found in various recent publications. Clark et al. [11] and Elmore et al. [18] investigate the problem of virtual machine migration and how migration can be used to implement shared nothing databases. Other publications concentrate on storage virtualization [32] as well as workload characterization [22] and resource allocation [35] for virtualized storage. FlurryDB, a dynamically scalable relational database that builds on virtual machine cloning, is described by Mior et al. [27]. The Relational Cloud project is a research prototype at MIT proposing scale-in, scale-out, and security for a private database virtualization – the project is driven by Curino et al. [15, 16]. Finally, Bernstein et al. report on Microsoft’s approach to adapt the SQL Server for cloud computing [7].

References

1. Acharya, S., Carlin, P., Galindo-Legaria, C.A., Kozielczyk, K., Terlecki, P., Zabback, P.: Relational support for flexible schema scenarios. *VLDBJ* **1**(2), 1289–1300 (2008)
2. Agrawal, R., Somani, A., Xu, Y.: Storage and querying of e-commerce data. In: *VLDB*, pp. 149–158 (2001)
3. Aulbach, S., Grust, T., Jacobs, D., Kemper, A., Rittinger, J.: Multi-tenant databases for software as a service: schema-mapping techniques. In: *SIGMOD*, pp. 1195–1206 (2008)

4. Aulbach, S., Jacobs, D., Kemper, A., Seibold, M.: A comparison of flexible schemas for software as a service. In: SIGMOD, pp. 881–888 (2009)
5. Aulbach, S., Seibold, M., Jacobs, D., Kemper, A.: Extensibility and data sharing in evolving multi-tenant databases. In: ICDE, pp. 99–110 (2011)
6. Beckmann, J.L., Halverson, A., Krishnamurthy, R., Naughton, J.F.: Extending rdbms to support sparse datasets using an interpreted attribute storage format. In: ICDE, p. 58 (2006)
7. Bernstein, P.A., Cseri, I., Dani, N., Ellis, N., Kalhan, A., Kakivaya, G., Lomet, D.B., Manne, R., Novik, L., Talus, T.: Adapting Microsoft SQL Server for Cloud Computing. In: ICDE conference, pp. 1255–1263 (2011)
8. Cabibbo, L., Carosi, A.: Managing inheritance hierarchies in object/relational mapping tools. In: CAiSE, *Lecture Notes in Computer Science*, vol. 3520, pp. 135–150. Springer (2005)
9. Cheng, J.M., Xu, J.: Xml and db2. In: ICDE, pp. 569–573 (2000)
10. Chu, E., Beckmann, J.L., Naughton, J.F.: The case for a wide-table approach to manage sparse relational data sets. In: SIGMOD, pp. 821–832 (2007)
11. Clark, C., Fraser, K., Hand, S.M., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: NSDI, pp. 273–286 (2005)
12. Copeland, G.P., Khoshafian, S.: A decomposition storage model. In: SIGMOD, pp. 268–279 (1985)
13. Crockford, D.: The application/json media type for javascript object notation (json), rfc 4627. <http://tools.ietf.org/html/rfc4627> (2006)
14. Cunningham, C., Graefe, G., Galindo-Legaria, C.A.: Pivot and unpivot: Optimization and execution strategies in an rdbms. In: VLDB, pp. 998–1009 (2004)
15. Curino, C., Jones, E.P., Madden, S., Balakrishnan, H.: Workload-Aware Database Monitoring and Consolidation. In: SIGMOD conference, pp. 313–324 (2011)
16. Curino, C., Jones, E.P.C., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational Cloud: A Database-as-a-Service for the Cloud. In: CIDR (2011). URL <http://dspace.mit.edu/handle/1721.1/62241>
17. Currim, F., Ram, S.: When entities are types: Effectively modeling type-instantiation relationships. In: ERW, *Lecture Notes in Computer Science*, vol. 6413. Springer (2010)
18. Elmore, A.J., Das, S., Agrawal, D., El Abbadi, A.: Zephyr: live migration in shared nothing databases for elastic cloud platforms. In: SIGMOD conference, pp. 301–312 (2011). URL <http://cs.ucsb.edu/~sudipto/papers/zephyr.pdf>
19. Foping, F.S., Dokas, I.M., Feehan, J., Imran, S.: A new hybrid schema-sharing technique for multitenant applications. In: ICDIM, pp. 211–216 (2009)
20. Friedman, C., Hripsak, G., Johnson, S.B., Cimino, J.J., Clayton, P.D.: A generalized relational schema for an integrated clinical patient database. In: SCAMC, pp. 335–339 (1990)
21. Grust, T., van Keulen, M., Teubner, J.: Accelerating xpath evaluation in any rdbms. *ACM Transactions on Database Systems* **29**(1), 91–131 (2004)
22. Gulati, A., Kumar, C., Ahmad, I.: Storage Workload Characterization and Consolidation in Virtualized Environments. In: VPACT (2009)
23. Jacobs, D., Aulbach, S.: Ruminations on multi-tenant databases. In: BTW, *LNI*, vol. 103, pp. 514–521 (2007)
24. Kaldewey, T., Wong, T.M., Golding, R.A., Povzner, A., Brandt, S.A., Maltzahn, C.: Virtualizing disk performance. In: RTAS, pp. 319–330 (2008)
25. Kiefer, T., Lehner, W.: Private table database virtualization for dbaas. In: UCC, pp. 328–329 (2011)
26. Maier, D., Ullman, J.D.: Maximal objects and the semantics of universal relation databases. *ACM Transactions on Database Systems* **8**(1), 1–14 (1983)
27. Mior, M.J., Lara, E.D.: FlurryDB: A Dynamically Scalable Relational Database with Virtual Machine Cloning. In: SYSTOR conference. Haifa, Israel (2011)
28. Nicola, M., der Linden, B.V.: Native xml support in db2 universal database. In: VLDB, pp. 1164–1174 (2005)
29. Park, S.M., Humphrey, M.: Self-tuning virtual machines for predictable escience. In: CCGrid, pp. 356–363 (2009)

30. Shafranovich, Y.: Common format and mime type for comma-separated values (csv) files, rfc 4180 (2005)
31. Shivam, P., Babu, S., Chase, J.S.: Learning application models for utility resource planning. In: Proc. of the 3rd Int. Conf. on Autonomic Computing (ICAC) 2006, Dublin, Ireland, pp. 255–264 (2006)
32. Singh, A., Korupolu, M., Mohapatra, D.: Server-storage virtualization: integration and load balancing in data centers. In: Super Computing Conference, p. 53 (2008)
33. Soror, A.A., Minhas, U.F., Aboulnaga, A., Salem, K., Kokosiellis, P., Kamath, S.: Automatic Virtual Machine Configuration for Database - SIGMOD. In: SIGMOD (2008)
34. Soror, A.A., Minhas, U.F., Aboulnaga, A., Salem, K., Kokosiellis, P., Kamath, S.: Automatic Virtual Machine Configuration for Database Workloads. *ACM Transactions on Database Systems* **35**(1), 1–47 (2010)
35. Soundararajan, G., Lupei, D., Ghanbari, S., Popescu, A.D., Chen, J., Amza, C.: Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In: FAST, pp. 71–84 (2009)
36. Sun: JSR 220: Enterprise JavaBeans™ 3.0 (persistence) (2006)
37. W3C: XML Path Language (XPath) 2.0. <http://www.w3.org/TR/2007/REC-xpath20-20070123/> (2007)
38. W3C: Extensible Markup Language (XML) 1.0 (Fifth Edition). <http://www.w3.org/TR/2008/REC-xml-20081126/> (2008)
39. W3C: XQuery 1.0: An XML Query Language (Second Edition). <http://www.w3.org/TR/2010/REC-xquery-20101214/> (2010)
40. Weissman, C.D., Bobrowski, S.: The design of the force.com multitenant internet application development platform. In: SIGMOD, pp. 889–896 (2009)
41. Wilkes, J.: Traveling to Rome: QoS specifications for automated storage system management. In: IWQoS (2001)
42. Wilkes, J.: Traveling to Rome: a retrospective on the journey. In: SIGOPS (2009)



<http://www.springer.com/978-1-4614-6855-4>

Web-Scale Data Management for the Cloud

Lehner, W.; Sattler, K.-U.

2013, XV, 193 p., Hardcover

ISBN: 978-1-4614-6855-4