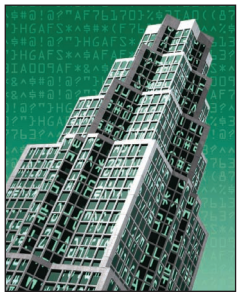
Editor: Tim Kraska • kraskat@cs.brown.edu



The New Database Architectures

Tim Kraska • *Brown University*

Beth Trushkowsky • *University of California, Berkeley*

The rise of big data and cloud computing significantly changed the database market. Here, the authors describe how developers decomposed the traditional database system and questioned the classical three-tier system architecture.

Every big data system faces one fundamental challenge: scalability. The common assumption that even the most obscure data might contain some value, and the resulting obsession to store as much as possible, make predicting data volumes essentially impossible. Even worse, for interactive (Web) applications, sudden spikes in usage can create short-term resource demands, which go way beyond a system's envisioned capacity.

Cloud computing lets system administrators adjust resources in an on-demand fashion. However, as application developers began to leverage the cloud, many felt that standard databases were too inflexible to adapt to the cloud-scale setting. This sentiment ultimately resulted in a spike in innovation: a variety of novel system designs, features, and data models have arisen to replace traditional database systems. Although these new systems share many of the same fundamental properties with the old ones (such as logical or physical independence), their developers have changed how they are deployed and used on clusters of computers to achieve better scalability and performance.

Here, we aim not to describe in detail specific existing systems, but rather to further understanding of how developers decomposed the database and, by doing so, changed the classical three-tier system architecture.

Why Reinvent the Database System?

The pay-as-you-go model that cloud computing provides for leasing resources presents a new opportunity for developers to optimize their system designs for cost.[1] However, to realize cloud computing's cost benefits, data management in the cloud has several new requirements:

- *Elasticity*. Cloud database systems should easily scale up and down with the workload. This avoids over-provisioning the system and thus incurs lower operational costs. Horizontal scaling is a best practice because it is easier to add or remove machines than to upgrade a single machine, and because (theoretically) it does not limit scalability.
- *Flexibility and management ease*. On the application side, deploying new applications should be easy using a cloud database system. This requirement implies adaptable data models and schemata.
- *Fault tolerance*. Transparent failure handling is important for cloud database systems because failures are the norm rather than the exception in large-scale deployments.

Traditional database systems do not fulfill all these requirements, which led some developers to believe that a problem existed with the relational model and SQL. This belief helped spawn the NoSQL movement, leading to the development of many new systems that aimed to meet the aforementioned cloud-based data management requirements while forgoing the data model and query language of traditional database systems. However, neither SQL nor the relational model are the real problem; the monolithic nature of

 Published by the IEEE Computer Society 1089-7801/13/$31.00 © 2013 IEEE

traditional database architectures is to blame. As we discuss next, these systems demonstrated that new, more flexible architectures were required to meet the needs of scalable, interactive applications in the cloud.

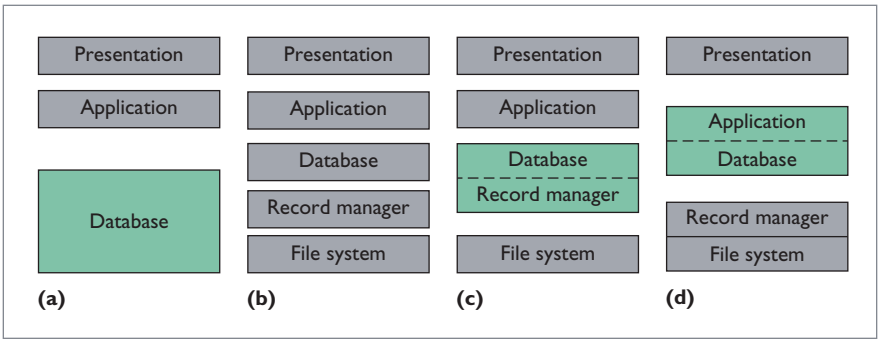


Figure 1. Tiered architectures. We can see (a) the classic three-tier, (b) five-tier, (c) four-tier, and (d) three-tier key-value architectures. Solid lines represent separate components, whereas dashed lines show that components are bundled together.

Three-Tier Architecture

Databases, in the cloud or not, are rarely used alone. They are the backbone for applications, either transactional or analytical, and are embedded in complex system architectures. The classic architecture, which is taught in every computer science curriculum, consists of three tiers: presentation, application, and data (Figure 1a). Each tier, or layer, is provided as a service and is typically deployed on one or more dedicated servers. Building a layered system has many advantages. Most importantly, it reduces complexity and simplifies testing. For cloud-scale deployments, layering makes it easier to scale, replicate, partition, or cache each layer independently. Furthermore, moving computation from the database to the application and presentation layers reduces the load on the data tier. The application and presentation tiers are easier to scale because they can be designed as stateless services: system administrators can improve scalability and fault-tolerance by adding more machines (also referred to as *scaling out*) without worrying about complicated synchronization protocols. In contrast, the database tier is stateful; scaling it out requires more complicated control architectures (such as master-slave) as well as data consistency protocols (for example, two-phase commit). Consequently, high scalability and fault-tolerance are harder to achieve in the database tier.

To make the situation worse, because databases are typically implemented as monolithic systems, scaling the database implies upgrading its server with more powerful hardware (also referred to as *scaling up*), which limits availability and scalability. Despite these restrictions, well-known cloud service providers still employ the classic monolithic design because it allows for reusing existing mature database systems. For example, Microsoft SQL Azure and Yahoo's small application platform use this architecture together with special fabrics on top of the data tier to improve availability and scalability.

New Data Tiers

Many developers were dissatisfied with the monolithic approach and consequently broke it down to gain the flexibility required to make new systems amenable to the pay-as-you-go elasticity the cloud provides. As part of this process, they divided the traditional three-tier architecture into five tiers: presentation, application, database, record manager, and file system (Figure 1b). New systems recombine these tiers into layers in various ways; each layer is a service that can be scaled independently and managed for fault-tolerance. The three new or redefined tiers offer various functionalities.

File Tier

The file tier provides raw file access, and is the bottom-most tier in the stack. The file system interface provides read/write capability on file granularity and is typically optimized for larger files. The most prominent cloud examples are the Google File System[2] and Amazon Simple Storage Service. Typically, files are semantically organized into a hierarchical structure; file names are unique within their local context. Many existing distributed file systems have a relaxed consistency model but support monotonicity (for instance, you see your own writes).

Record Manager Tier

The record manager extends the file system's functionality by providing more fine-grained data access, and is optimized for accessing smaller items. In the simplest case, the record manager supports only reads and writes for a single value based on a key. Systems providing this simple interface became popular during the NoSQL movement and are often referred to as key-value stores. However, many systems in this category started to provide more advanced functionality, including relational data models and sorted key-range requests,[3–5] making the term "key-value store" misleading. Here, we refer to these systems as *record managers*, because they provide similar functionality to the record manager component in a data management system. The record manager

can arrange data items within the file system to provide better data locality for correlated queries. We place access optimizations such as B-tree indexes in this tier, given that indexes provide the same or a similar interface as the data they index. The record manager can implement version control in this tier by storing multiple versions of the same record.[4] The manager might additionally support concurrency control primitives such as test and set.

Database Tier

The database tier sits on top of the record manager and is responsible only for query processing, transaction processing, and view/index maintenance.[3,6,7] Depending on the system's query model, the database has more or less work to do — from compiling SQL-like queries into physical plans for execution against the record manager to simply deciding which servers to access for data retrieval. In systems that support analytical queries (such as Hadoop's MapReduce), the database schedules units of work (*tasks*) from concurrent queries for available servers. Whereas most existing record managers support only atomic access to individual records, the database might provide additional transactional guarantees across records.[3,5,7] Finally, the database tier automatically creates and updates secondary indexes.

The new layers let developers use different distribution techniques for each layer. For instance, it is now possible to apply caching between the layers and combine different control architectures, such as master-slave and multi-master, in a single system. For the data layers, the most crucial design decision is the trade-off between availability and consistency. For the upper layers, maintaining good performance as storage becomes more distributed necessitates a scalable data model and query language; a query shouldn't try to access data from all the storage nodes.

The New Architectures

Existing systems combine the aforementioned tiers differently. Combining more tiers trades off modularity and inter-tier performance, because colocated tiers have low-latency communication.

Five-Tier Architecture

The five-tier architecture maintains each tier as its own service (Figure 1b). The most prominent example is the Google stack, consisting of the GFS (file tier), BigTable (record manager tier), and Megastore (database tier), with the application and presentation tiers on top. This separation provides the highest modularity, allowing developers to easily use subsets of the full stack. For example, analytical data processing via MapReduce can just use the file tier, GFS. BigTable provides simple record manager functionality on top of GFS: key-value access ("rows," in Google's terminology) with atomic single-row operations, used for Google's customized Web index. Megastore adds transactional support and auto-maintained indexes. Although this fine-grained layering has many advantages, it also causes additional network traffic between the layers. For example, a join requires that data be shipped from the file tier through the record manager tier to the database tier. The architecture might also cause inefficiencies by duplicating similar functionality in different layers. In Google's case, different layers implement logging for consistency and fault-tolerance reasons, creating logs of log records.

Four-Tier-Separated File-Tier Architecture

One way to reduce traffic between layers is to merge the database and record manager tiers into one layer (Figure 1c). Beneath it, the file service acts as reliable, *network-attached storage* (NAS). Similar to the classic three-tier architecture, this combination lets developers use off-the-shelf database systems such as MySQL. However, the file tier, not the database system, is responsible for achieving high fault-tolerance. Examples of this architecture include Amazon Relational Database Service (RDS), and Amazon Elastic Compute Cloud (EC2) coupled with Amazon Elastic Block Service (EBS) as the NAS. With the latter service, developers are able to install their favorite database system; they achieve durability simply by configuring the database to store all logs on the mounted EBS drive, which can be mounted on a different EC2 instance if the original instance fails. Unfortunately, this architecture still inherits the scalability limits of the classic three-tier architecture because it typically uses a traditional monolithic database system. It is also hard to align the different consistency protocols used inside the file tier and the combined record-manger/database tier. For example, if there is a failure with Amazon RDS, the application might lose the last 5 minutes of changes.

Three-Tier Database Library

Other systems[3,7] present another combination by including the database tier as a library within the application (Figure 1d). The bottom layer consists of the file system and record manager tiers, which can be both replicated and partitioned across many machines. This architecture cleanly separates the stateless part of the overall stack (presentation, application, and database) from the stateful, persistent part (record manager and file system), while still providing optimized access paths to the data through the combined record manager/file layer. In other words, query and transaction

processing simply scale with the application server, reducing load for the stateful layer. Furthermore, a record manager can scale to hundreds of nodes because it bypasses the classical scalability bottleneck inside the data tier. Although this architecture provides great scalability features, it is best suited for transaction-heavy workloads with small datasets per transaction. For larger analytical queries, the network between the application/database and record manager/file layers could become a bottleneck. However, with the current trend toward larger network bandwidth and more powerful record manager services that can perform predicate push-downs, these workload restrictions could vanish.

No clear winner has emerged yet regarding the best combination for these tiered architectures. Some argue that the old three-tier architecture still provides the best benefits because the database system has full control over all data management and, if designed right, can scale. Others argue that every layer provides value on its own, and why should we pay the price of query processing just to store a key and a value? Finally, the database library has many advantages because it can provide the best of both worlds: query processing and direct key-value access.

References

1. D. Florescu and D. Kossmann, "Rethinking Cost and Performance of Database Systems," *SIGMOD Record*, vol. 38, no. 1, 2009, pp. 43–48; http://doi.acm.org/10.1145/1558334.1558339.
2. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proc. 19th ACM Symp. Operating Systems Principles*, ACM, 2003, pp. 29–43; http://doi.acm.org/10.1145/945445.945450.
3. M. Armbrust et al. "SCADS: Scale-Independent Storage for Social Computing Applications," *Proc. Conf. Innovative Data Systems Research* (CIDR 09), 2009; http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_86.pdf.
4. F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Computer Systems*, vol. 26, no. 2, 2008; http://doi.acm.org/10.1145/1365815.1365816.
5. "Scalaris: Distributed Transactional Key-Value Store," Zuse Inst. and onScale Solutions, 2009; http://code.google.com/p/scalaris/.
6. J. Baker et al., "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," *Proc. Conf. Innovative Data Systems Research* (CIDR 11), 2011, pp. 223–234; www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.
7. M. Brantner et al., "Building a Database on S3," *Proc. 2008 ACM SIGMOD Int'l Conf. Management of Data*, ACM, 2008, pp. 251–264; http://doi.acm.org/10.1145/1376616.1376645.

Tim Kraska is an assistant professor in the Computer Science Department at Brown University. His research focuses on big data management and hybrid human-machine database systems. Kraska has a PhD from ETH Zurich in Switzerland. Contact him at kraskat@cs.brown.edu.

Beth Trushkowsky is a PhD candidate in computer science at the University of California, Berkeley. Her research interests include scalable databases and crowdsourcing. Trushkowsky has an MS in computer science from the University of California, Berkeley. Contact her at trush@eecs.berkeley.edu.