Rule 14—Use Databases Appropriately

Rule 14: What, When, How, and Why
What: Use relational databases when you need ACID properties to maintain relationships between your data. For other data storage needs consider more appropriate tools.
When to use: When you are introducing new data or data structures into the architecture of a system.
How to use: Consider the data volume, amount of storage, response time requirements, relationships, and other factors to choose the most appropriate storage tool.
Why: An RDBMS provides great transactional integrity but is more difficult to scale, costs more, and has lower availability than many other storage options.
Key takeaways: Use the right storage tool for your data. Don't get lured into sticking everything in a relational database just because you are comfortable accessing data in a database.

Relational database management systems (RDBMSs), such as Oracle and MySQL, are based on the relational model introduced by Edgar F. Codd in his 1970 paper "A Relational Model of Data for Large Shared Data Banks." Most RDBMSs provide two huge benefits for storing data. The first is the guarantee of transactional integrity through ACID properties, see Table 2.1 in Chapter 2, "Distribute Your Work," for definitions. The second is the relational structure within and between tables. To minimize data redundancy and improve transaction processing, the tables of most Online Transaction Processing databases (OLTP) are normalized to Third Normal Form, where all records of a table have the same fields, nonkey fields cannot be described by only one of the keys in a composite key, and all nonkey fields must be described by the key. Within the table each piece of data is highly related to other pieces of data. Between tables there are often relationships known as foreign keys. While these are two of the major benefits of using an RDBMS, these are also the reason for their limitations in terms of scalability.

Because of this guarantee of ACID properties, an RDBMS can be more challenging to scale than other data stores. When you guarantee consistency of data and you have multiple nodes in your RDBMS cluster, such as with MySQL NDB, synchronous replication is used to guarantee that data is written to multiple nodes upon committing the data. With Oracle RAC there is a central database, but ownership of areas of the DB are shared among the nodes so write requests have to transfer ownership to that node and reads have to hop from requestor to master to owner and back. Eventually you are limited by the number of nodes that data can be synchronously replicated to or by their physical geographical location.

The relational structure within and between tables in the RDBMS makes it difficult to split the database through such actions as sharding or partitioning. See Chapter 2 for rules related to distributing work across multiple machines. A simple query that joined two tables in a single database must be converted into two separate queries with the joining of the data taking place in the application to split tables into different databases.

The bottom line is that data that requires transactional integrity or relationships with other data are likely ideal for an RDBMS. Data that requires neither relationships with other data nor transactional integrity might be better suited for other storage systems. Let's talk briefly about a few of the alternative storage solutions and how they might be used in place of a database for some purposes to achieve better, more cost-effective, and more scalable results.

One often overlooked storage system is a file system. Perhaps this is thought of as unsophisticated because most of us started programming by accessing data in files rather than databases. Once we graduated to storing and retrieving data from a database we never looked back. File systems have come a long way, and many are specifically designed to handle very large

amounts of files and data. Some of these include Google File System (GFS), MogileFS, and Ceph. File systems are great alternatives when you have a "write once-read many" system. Put another way, if you don't expect to have conflicting reads and writes over time on a structure or object and you don't need to maintain a great deal of relationships, you don't really need the transactional overhead of a database; file systems are a great choice for this kind of work.

The next set of alternative storage strategies is termed NoSQL. Technologies that fall into this category are often subdivided into key-value stores, extensible record stores, and document stores. There is no universally agreed classification of technologies, and many of them could accurately be placed in multiple categories. We've included some example technologies in the following descriptions, but this is not to be considered gospel. Given the speed of development on many of these projects, the classifications are likely to become even more blurred in the future.

Key-value stores include technologies such as Memcached, Tokyo Tyrant, and Voldemort. These products have a single key-value index for data and that is stored in memory. Some have the capability to write to disk for persistent storage. Some products in this subcategory use synchronous replication across nodes while others are asynchronous. These offer significant scaling and performance by utilizing a simplistic data store model, the key-value pair, but this is also a significant limitation in terms of what data can be stored. Additionally, the key-value stores that rely on synchronous replication still face the limitations that RDBMS clusters do, which are a limit on the number of nodes and their geographical locations.

Extensible record stores include technologies such as Google's proprietary BigTable and Facebook's, now open source, Cassandra. These products use a row and column data model that can be split across nodes. Rows are split or sharded on primary keys, and columns are broken into groups and placed on different nodes. This method of scaling is similar to the X and Y axes in the AKF Scale Cube, shown in Figure 2.1 in Chapter 2, where the X axis split is read replicas, and the Y axis is separating the tables by services supported. In these products row sharding is done automatically, but column splitting requires user definitions, similar to how it is performed in an RDBMS. These products utilize an asynchronous replication providing eventual consistency. This means that eventually, which may take milliseconds or hours, the data will become consistent across all nodes.

Document stores include technologies such as CouchDB, Amazon's SimpleDB, and Yahoo's PNUTS. The data model used in this category is called a "document" but is more accurately described as a multi-indexed object model. The multi-indexed object (or "document") can be aggregated into collections of multi-indexed objects (typically called "domains"). These collections or "domains" in turn can be queried on many different attributes. Document store technologies do not support ACID properties; instead, they utilize asynchronous replication, providing an eventually consistent model.

NoSQL solutions limit the number of relationships between objects or entities to a small number. It is this reduction of relationships that allows for the systems to be distributed across many nodes and achieve greater scalability while maintaining transactional integrity and read-write conflict resolution.

As is so often the case, and as you've probably determined reading the preceding text, there is a tradeoff between scalability and flexibility within these systems. The degree of relationship between data entities ultimately drives this tradeoff; as relationships increase, flexibility also

increases. This flexibility comes at an increase in cost and a decrease in the ability to easily scale the system. Figure 4.1 plots RDBMS, NoSQL, and file systems solutions against both the costs (and limits) to scale the system and the degree to which relationships are used between data entities. Figure 4.2 plots flexibility against the degree of relationships allowed within the system. The result is clear: Relationships engender flexibility but also create limits to our scale. As such, we do not want to overuse relational databases but rather choose a tool appropriate to the task at hand to engender greater scalability of our system.

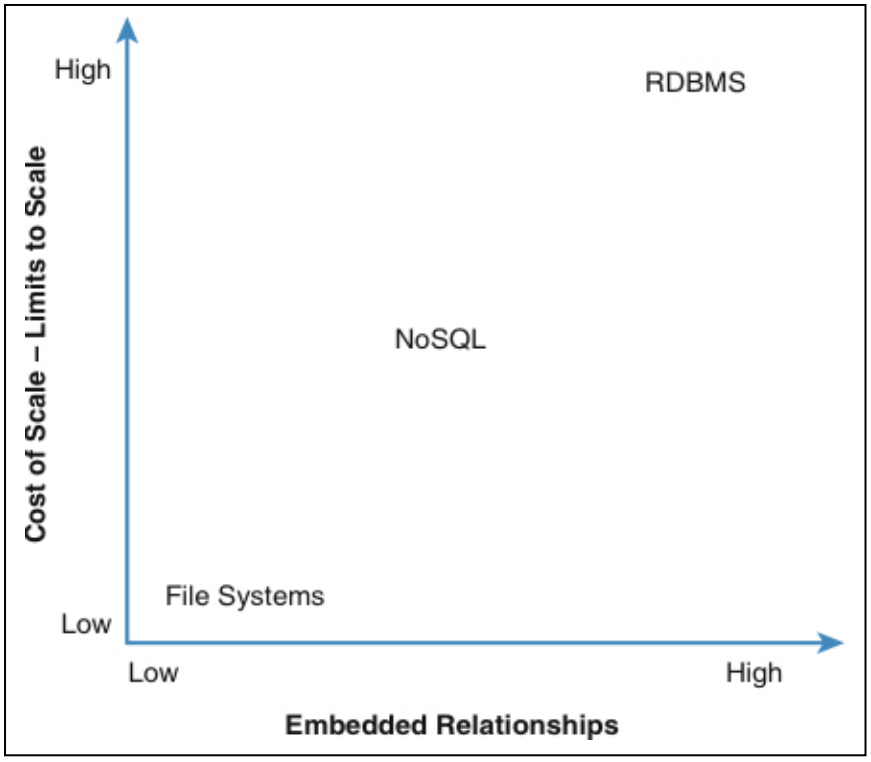

Figure 4.1 Cost and limits to scale versus relationships

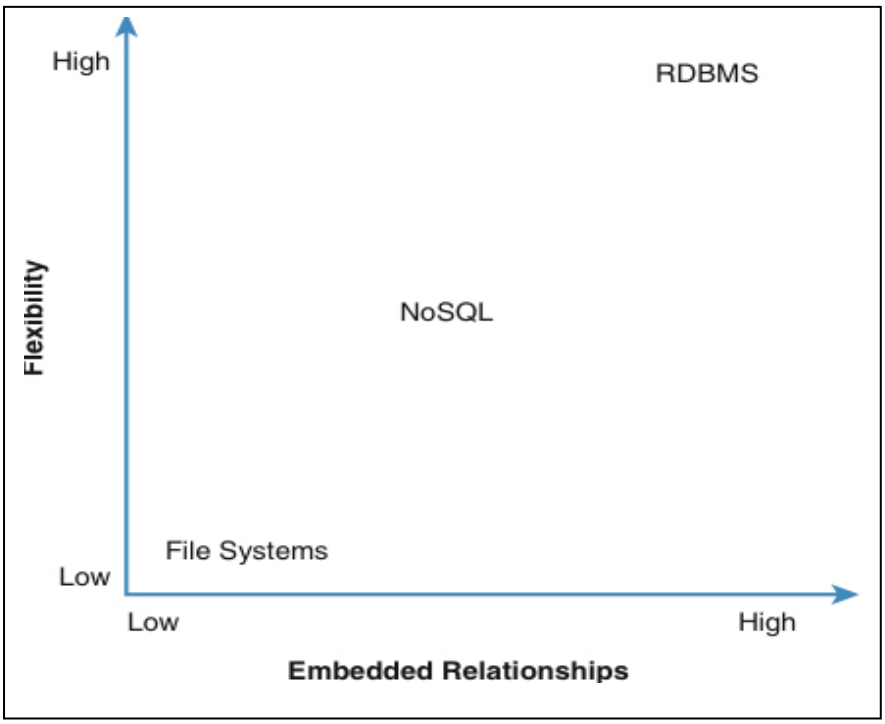

Figure 4.2 Flexibility versus relationships

Another data storage alternative that we are going to cover in this rule is Google's MapReduce.[1] At a high level, MapReduce has both a Map and a Reduce function. The Map function takes a key-value pair as input and produces an intermediate key-value pair. The input key might be the name of a document or pointer to a piece of a document. The value could be content consisting of all the words within the document itself. This output is fed into a reducer function that uses a program that groups the words or parts and appends the values for each into a list. This is a rather trivial program that sorts and groups the functions by key. The huge benefit of this technology is the support of distributed computing of very large data sets across many servers.

An example technology that combines two of our data storage alternatives is Apache's Hadoop. This was inspired by Google's MapReduce and Google File System, both of which are described previously. Hadoop provides benefits of both a highly scalable file system with distributed processing for storage and retrieval of the data.

So now that we've covered a few of the many options that might be preferable to a database when storing data, what data characteristics should you consider when making this decision? As with the myriad of options available for storage, there are numerous characteristics that should be considered. A few of the most important ones are the number of degree of relationships needed between elements, the rate of growth of the solution, and the ratio of reads to writes of the data (and potentially whether data is updated). Finally we are interested in how well the data monetizes (that is, is it profitable?) as we don't want our cost of the system to exceed the value we expect to achieve from it.

The degree of relationships between data is important as it drives flexibility, cost, and time of development of a solution.

As an example, imagine the difficulty of storing a transaction involving a user's profile, payment, purchase, and so on, in a key-value store and then retrieving the information piecemeal such as through a report of purchased items. While you can certainly do this with a file system or NoSQL alternative, it may be costly to develop and time consuming in delivering results back to a user.

The expected rate of growth is important for a number of reasons. Ultimately this rate impacts the cost of the system and the response times we would expect for some users. If a high degree of relationships are required between data entities, at some point we will run out of hardware and processing capacity to support a single integrated database, driving us to split the databases into multiple instances.

Read and write ratios are important as they help drive an understanding of what kind of system we need. Data that is written once and read many times can easily be put on a file system coupled with some sort of application, file, or object cache. Images are great examples of systems that typically can be put on file systems. Data that is written and then updated, or with high write to read ratios, are better off within NoSQL or RDBMS solutions.

These considerations bring us to another cube, Figure 4.3, where we've plotted the three considerations against each other. Note that as the X, Y, and Z axes increase in value, so does the cost of the ultimate solution increase. Where we require a high degree of relationships between systems (upper right and back portion of Figure 4.3), rapid growth, and resolution of read and write conflicts we are likely tied to several smaller RDBMS systems at relatively high cost in both our development and the systems, maintenance, and possibly licenses for the databases. If

growth and size are small but relationships remain high and we need to resolve read and write conflict, we can use a single monolithic database (with high availability clustering).

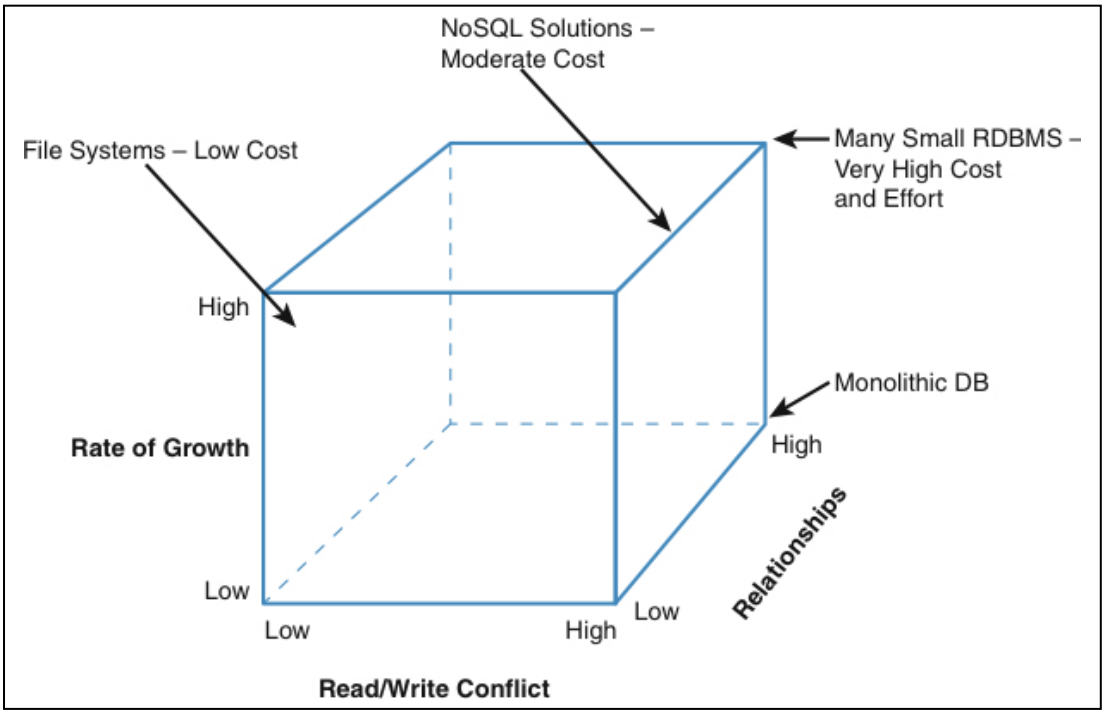

Figure 4.3 Solution decision cube

Relaxing relationships slightly allows us to use NoSQL alternatives at any level of reads and writes and with nearly any level of growth. Here again we see the degree to which relationships drive our cost and complexity, a topic we explore later in Chapter 8, "Database Rules." Cost is lower for these NoSQL alternatives. Finally, where relationship needs are low and read-write conflict is not a concern we can get into low-cost file systems to provide our solutions.

Monetization value of the data is critical to understand because as many struggling startups have experienced, storing terabytes of user data for free on class A storage is a quick way to run out of capital. A much better approach might be using tiers of data storage; as the data ages in terms of access date, continue to push it off to cheaper and slower access storage media. We call this the Cost-Value Data Dilemma, which is where the value of data decreases over time and the cost of keeping it increases over time. We discuss this dilemma more in Rule 47 and describe how to solve the dilemma cost effectively.