NoSQL concepts

This chapter covers

- NoSQL concepts
- ACID and BASE for reliable database transactions
- How to minimize downtime with database sharding
- Brewer's CAP theorem

Less is more.
—Ludwig Mies van der Rohe

In this chapter, we'll cover the core concepts associated with NoSQL systems. After reading this chapter, you'll be able to recognize and define NoSQL concepts and terms, you'll understand NoSQL vendor products and features, and you'll be able to decide if these features are appropriate for your NoSQL system. We'll start with a discussion about how using simple components in the application development process removes complexity and promotes reuse, saving you time and money in

2.1 Keeping components simple to promote reuse

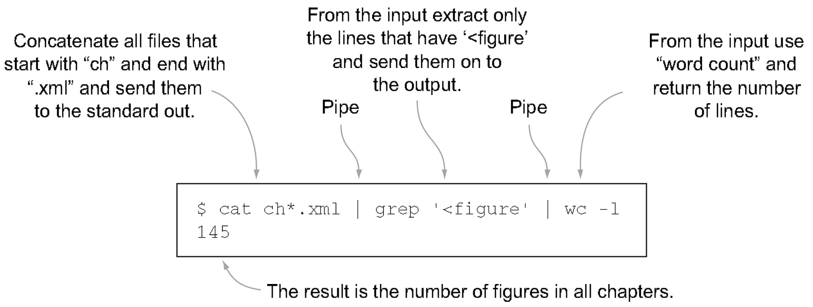
If you've worked with relational databases, you know how complex they can be. Generally, they begin as simple systems that, when requested, return a selected row from a single flat file. Over time, they need to do more and evolve into systems that manage multiple tables, perform join operations, do query optimization, replicate transactions, run stored procedures, set triggers, enforce security, and perform indexing. NoSQL systems use a different approach to solving these complex problems by creating simple applications that distribute the required features across the network. Keeping your architectural components simple allows you to reuse them between applications, aids developers in understanding and testing, and makes it easier to port your application to new architectures.

From the NoSQL perspective, simple is good. When you create an application, it's not necessary to include all functions in a single software application. Application functions can be distributed to many NoSQL (and SQL) databases that consist of simple tools that have simple interfaces and well-defined roles. NoSQL products that follow this rule do a few things and they do them well. To illustrate, we'll look at how systems can be built using well-defined functions and focus on how easy it is to build these new functions.

If you're familiar with UNIX operating systems, you might be familiar with the concept of UNIX pipes. UNIX pipes are a set of processes that are chained together so that the output of one process becomes the input to the next process. Like UNIX pipes, NoSQL systems are often created by integrating a large number of modular functions that work together. An example of creating small functions with UNIX pipes to count the total number of figures in a book is illustrated in figure 2.1.

What's striking about this example is that by typing about 40 characters you've created a useful function. This task would be much more difficult to do on systems that don't support UNIX-style functions. In reality, only a query on a native XML database might be shorter than this command, but it wouldn't also be general-purpose.

Many NoSQL systems are created using a similar philosophy of modular components that can work together. Instead of having a single large database layer, they often


Concatenate all files that start with "ch" and end with ".xml" and send them to the standard out.
From the input extract only the lines that have '<figure' and send them on to the output.
Pipe
Pipe
From the input use "word count" and return the number of lines.
The result is the number of figures in all chapters.

```
$ cat ch*.xml | grep '<figure' | wc -l
145
```

Figure 2.1 UNIX pipes as an example of reusing simple tools to create new functions. This figure concatenates (puts together) all chapter files in a book into a single file and counts the number of figures in all chapters. With UNIX pipes, we do this by stringing together three simple commands: concatenate (`cat`), a search function called `grep`, and word count (`wc`). No additional code is needed; each function takes the output from the previous function and processes it.

have a number of simpler components that can be reassembled to meet the needs of different applications. For example, one function allows sharing of objects in RAM (memcache), another function runs batch jobs (MapReduce,) and yet another function stores binary documents (key-value stores). Note that most UNIX pipes were designed to transform linear pipelines of line-oriented data streams on a single processor. NoSQL components, though modular, are more than a series of linear pipeline components. Their focus is on efficient data services that are frequently used to power distributed web services. NoSQL systems can be documents, messages, message stores, file stores, REST, JSON, or XML web services with generic application program interfaces (APIs). There are tools to load, validate, transform, and output large amounts of data, whereas UNIX pipes were really designed to work on a single processor.

STANDARDS WATCH: UNIVERSAL PIPES FOR STRUCTURED DATA At this point you might be asking if you can still use the concepts behind UNIX pipes to process your structured data. The answer is yes, if you use JSON or XML standards! The World Wide Web Consortium has recognized the universal need for standard pipe concepts that work with any unstructured data. They have provided a standard for processing pipelined data called *XProc*. Several NoSQL databases have XProc built in as part of their architecture. XProc standards allow data transformation pipelines built with one NoSQL database to be ported to

http://www.w3.org/TR/xproc/. There's also a version of the UNIX shell that's specifically used for XML called XMLSH. You can read more about XMLSH at http://www.xmlsh.org.

The concept of simple functions in NoSQL systems will be a recurring theme. Don't be afraid to suggest or use a NoSQL system even if it doesn't meet all your needs in a single system. As you come to learn more about NoSQL systems, you'll think of them as collections of tools that become more useful the more you know about how they fit together. Next, we'll see how the concept of simplicity is important in the develop-

2.2 Using application tiers to simplify design

Understanding the role of tiered applications is important to objectively evaluate one or more application architectures. Since functions move between tiers, the comparison at times might not be as clear as when you look at a single tier. The best way to fairly and objectively compare systems is to take a holistic approach, looking at the overall application and how well it meets system requirements.

Using application tiers in your architecture allows you to create flexible and reusable applications. By segregating an application into tiers, you have the option of modifying or adding a specific layer instead of reworking an entire application when modifications are required. As you'll see in the following example, which compares RDBMSs and NoSQL systems, functions in NoSQL applications are distributed differently.

When application designers begin to think about software systems that store persistent data, they have many options. One choice is to determine whether they need to use application tiers to divide the overall functionality of their application. Identifying each layer breaks the application into separate architectural components, which allows the software designer to determine the responsibility of each component. This *separation of concerns* allows designers to make the complicated seem simple when explaining the system to others.

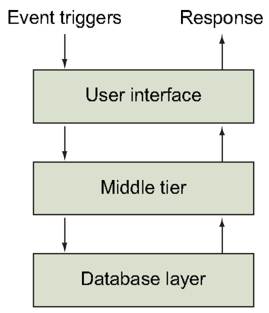


Figure 2.2 Application tiers are used to simplify system design. The NoSQL movement is concerned with minimizing bottlenecks in overall system performance, and this sometimes means moving key components out of one tier and putting them into another tier.

Application tiers are typically viewed in a layer-cake-like drawing, as shown in figure 2.2. In this figure, user events (like a user clicking a button on a web page) trigger code in the user interface. The output or response from the user interface is sent to the middle tier. This middle tier may respond by sending something back to the user interface, or it could access the database layer. The database layer may in turn run a query and send a response back to the middle tier. The middle tier then uses the data to create a report and sends it to the user. This process is the same whether you're using Microsoft Windows, Apple's OS X, or a web browser with HTML links.

When designing applications it's important to consider the trade-offs when putting functionality in each tier. Because relational databases have been around for a long time and are mature, it's common for database vendors to add functionality at the database tier and release it with their software rather than reusing components already delivered or developed. NoSQL system designers know that their software must work in complex environments with other applications where reuse and seamless interfaces are required, so they build small independent functions. Figure 2.3 shows the differences between RDBMS and NoSQL applications.

In figure 2.3 we compare the relational versus NoSQL methods of distributing application functions between the middle and database tiers. As you can see, both models have a user interface tier at the top. In the relational database, most of the application functionality is found in the database layer. In the NoSQL application, most of the application functionality is found in the middle tier. In addition, NoSQL systems leverage more services for managing BLOBs of data (the key-value store), for storing full-text indexes (the Lucene indexes), and for executing batch jobs (MapReduce).

A good NoSQL application design comes from carefully considering the pros and cons of putting functions in the middle versus the database tier. NoSQL solutions allow you to carefully consider all the options, and if the requirements include a high-scalability component, you can choose to keep the database tier simple. In traditional relational database systems, the complexity found in the database tier impacts the overall scalability of the application.

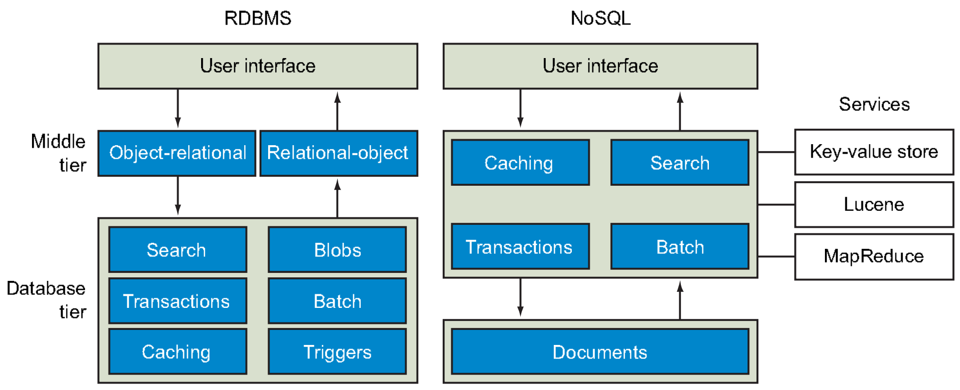


Figure 2.3 This figure compares application layers in RDBMSs and NoSQL systems. RDBMSs, on the left, have focused on putting many functions into the database tier where you can guarantee security and transactional integrity. The middle tier is used to convert objects to and from tables. NoSQL systems, on the right, don't use object-relational mapping concepts; they move database functions into the middle tier and leverage external services.

Remember, if you focus on a single tier you'll never get a fair comparison. When performing a trade-off analysis, you should compare RDBMSs with NoSQL systems as well as think about how repartitioning will impact functionality at each tier. This process is complex and requires an understanding of *both* RDBMS and NoSQL architectures. The following is a list of sample pros and cons you'll want to consider when performing a trade-off analysis.

RDBMS

- ACID
- Fine-grained security on columns and rows using views prevents views and
- Most SQL code is portable to other SQL databases, including open source
- Typed columns and constraints will validate data before it's added to the data-
- Existing staff members are already familiar with entity-relational design and SQL.

RDBMS

-
- Entity-relationship modeling must be completed before testing begins, which
- RDBMS
- Sharding over many servers can be done but requires application code and will
-
-

- Loading test data can be done with drag-and-drop tools before ER modeling is
-
-
-
-
-
- It's easy to store high-variability data.
- ACID transactions can be done only within a document at the database level.
-
- NoSQL systems are new to many staff members and additional training may be
- The document store has its own proprietary nonstandard query language,
- The document store won't work with existing reporting and OLAP tools.

Understanding the role of placing functions within an application tier is important to understanding how an application will perform. Another important factor to consider is how memory such as RAM, SSD, and disk will impact your system.

Terminology of database clusters

The NoSQL industry frequently refers to the concept of *processing nodes in a database cluster*. In general, each cluster consists of racks filled with commodity computer hardware, as shown in figure 2.4.

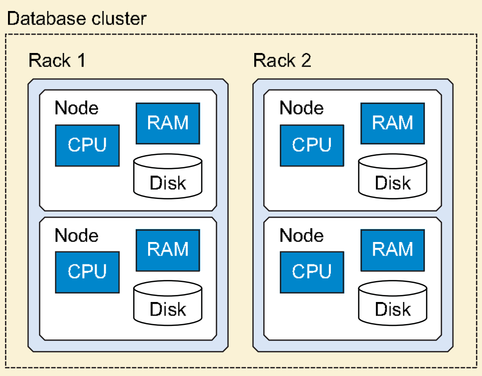


Figure 2.4 Some of the terminology used in distributed database clusters. A cluster is composed of a set of processors, called nodes, grouped together in racks. Nodes are commodity processors, each of which has its own local CPU, RAM, and disk.

(continued)
Each independent computer is called a *node*.

For the purposes of this book, unless we're discussing custom hardware, we'll define nodes as containing a single logical processor called a *CPU*. Each node has its own local RAM and disk. The CPU may in fact be implemented using multiple chips, each with multiple core processors. The disk system may also be composed of several independent drives.

Nodes are grouped together in racks that have high-bandwidth connections between all the nodes within a rack. Racks are grouped together to form a database cluster within a data center. A single data center location may contain many database clusters. Note that some NoSQL transactions must store their data on two nodes in different geographic locations to be considered successful transactions.

2.3 Speeding performance by strategic use of RAM, SSD, and disk

How do NoSQL systems use different types of memory to increase system performance? Generally, traditional database management systems weren't concerned with memory management optimization. In contrast, NoSQL systems are designed to be cost effective when creating fast user response times by minimizing the amount of expensive resources you need.

If you're new to database architectures, it's good to start with a clear understanding about the difference in performance between queries that retrieve their data from RAM (volatile random access memory) and queries that retrieve their data from hard drives. Most people know that when they turn off their computer after a long work day, the data in RAM is erased and must be reloaded. Data on *solid state drives (SSDs)* and *hard disk drives (HDDs)* persists. We also know that RAM access is fast and, in comparison, disk access is much slower. Let's assume 1 nanosecond is equal to approximately a foot, which is in fact roughly the time it takes for light to travel one foot. That means your RAM is 10 feet away from you, but your hard drive is over 10 million feet away, or about 2,000 miles. If you use a solid state disk, the result is slower than RAM, but not nearly as slow as a spinning disk drive (see figure 2.5).

Let's start by putting you in the city of Chicago, Illinois. If you want to get something from your RAM, you can usually find it in your back yard. If you're lucky enough to have data stored in a solid state disk, you can find it by making a quick trip somewhere in your neighborhood. But if you want to get something from your hard drive, you'll need to go to the city of Los Angeles, California, which is about 2,000 miles away. Not a round trip you want to make often if you can avoid it.

Rather than drive all the way to Los Angeles and back, what if you could check around your neighborhood to see if you already have the data? The time it takes to do a calculation in a chip today is roughly the time it takes light to travel across the chip.

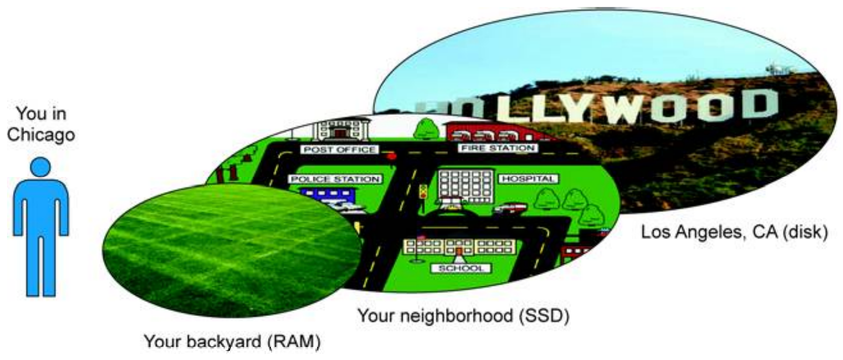


Figure 2.5 To get a feel for how expensive it is to access your hard drive compared to finding an item in RAM cache, think of how long it might take you to pick up an item in your back yard (RAM). Then think of how long it would take to drive to a location in your neighborhood (SSD), and finally think of how long it would take to pick up an item in Los Angeles if you lived in Chicago (HDD). This shows that finding a query result in a local cache is more efficient than an expensive query that needs HDD access.

You can do a few trillion calculations while you're waiting for your data to get back from LA. That's why calculating a hash is much faster than going to disk, and the more RAM you have, the lower the probability you need to make that long round trip.

The solution to faster systems is to keep as much of the right information in RAM as you can, and check your local servers to see if they might also have a copy. This local fast data store is often called a *RAM cache* or *memory cache*. Yet, accomplishing this and determining when the data is no longer current turn out to be difficult questions.

Many memory caches use a simple timestamp for each block of memory in the cache as a way of keeping the most recently used objects in memory. When memory fills up, the timestamp is used to determine which items in memory are the oldest and should be overwritten. A more refined view can take into account how much time or resources it'll take to re-create the dataset and store it in memory. This "cost model" allows more expensive queries to be kept in RAM longer than similar items that could be regenerated much faster.

The effective use of RAM cache is predicated on the efficient answer to the question, "Have we run this query before?" or equivalently, "Have we seen this document before?" These questions can be answered by using *consistent hashing*, which lets you know if an item is already in the cache or if you need to retrieve it from SSD or HDD

2.4 Using consistent hashing to keep your cache current

You've learned how important it is to keep frequently used data in your RAM cache, and how by avoiding unnecessary disk access you can improve your database performance. NoSQL systems expand on this concept and use a technique called *consistent hashing* to keep the most frequently used data in your cache.

Consistent hashing is a general-purpose process that's useful when evaluating how NoSQL systems work. Consistent hashing quickly tells you if a new query or

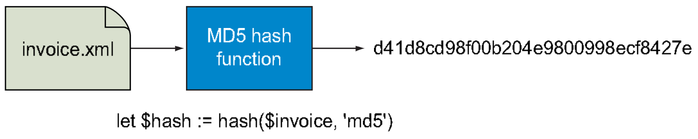

invoice.xml
MD5 hash function
d41d8cd98f00b204e9800998ecf8427e
let $hash := hash($invoice, 'md5')

Figure 2.6 Sample hashing process. An input document such as a business invoice is sent through a hashing function. The result of the hashing function is a string that's unique to the original document. A change of a single byte in the input will return a different hash string. A hash can be used to see if a document has changed or if it's already located in a RAM cache.

document is the same as one already in your cache. Knowing this information prevents you from making unnecessary calls to disk for information and keeps your databases running fast.

A hash string (also known as a *checksum* or *hash*) is a process that calculates a sequence of letters by looking at each byte of a document. The hash string uniquely identifies each document and can be used to determine whether the document you're presented with is the same document you already have on hand. If there's any difference between two documents (even a single byte), the resulting hash will be different. Since the 1990s, hash strings have been created using standardized algorithms such as MD5, SHA-1, SHA-256, and RIPEMD-160. Figure 2.6 illustrates a typical hashing process.

Hash values can be created for simple queries or complex JSON or XML documents. Once you have your hash value, you can use it to make sure that the information you're sending is the same information others are receiving. Consistent hashing occurs when two different processes running on different nodes in your network create the same hash for the same object. Consistent hashing confirms that the information in the document hasn't been altered and allows you to

Hash collisions

There's an infinitesimally small chance that two different documents could generate the same hash value, resulting in a *hash collision*. The likelihood of this occurring is related to the length of the hash value and how many documents you're storing. The longer the hash, the lower the odds of a collision. As you add more documents, the chance of a collision increases. Many systems use the MD5 hash algorithm that generates a 128-bit hash string. A 128-bit hash can generate approximately 10^{38} possible outputs. That means that if you want to keep the odds of a collision low, for example odds of under one in 10^{18}, you want to limit the number of documents you keep to under 10^{13}, or about 10 trillion documents.

For most applications that use hashing, accidental hash collisions aren't a concern. But there are situations where avoiding hash collisions is important. Systems that use hashes for security verification, like government or high-security systems, require hash values to be greater than 128 bits. In these situations, algorithms that generate a hash value greater than 128 bits like SHA-1, SHA-256, SHA-384, or SHA-512 are preferred.

determine whether the object exists in your cache or message store, saving precious resources by only rerunning processes when necessary.

Consistent hashing is also critical for synchronizing distributed databases. For example, version control systems such as Git or Subversion can run a hash not only on a single document but also on hashes of hashes for all files within a directory. By doing this consistently, you can see if your directory is in sync with a remote directory and, if not, you can run update operations on only the items that have changed.

Consistent hashing is an important tool to keep your cache current and your system running fast, even when caches are spread over many distributed systems. Consistent hashing can also be used to assign documents to specific database nodes on distributed systems and to quickly compare remote databases when they need to be synchronized. Distributed NoSQL systems rely on hashing for rapidly enhancing data-

2.5 Comparing ACID and BASE—two methods of reliable database transactions

Transaction control is important in distributed computing environments with respect to performance and consistency. Typically one of two types of transaction control models are used: ACID, used in RDBMS, and BASE, found in many NoSQL systems. Even if only a small percentage of your database transactions requires transactional integrity, it's important to know that both RDBMSs and NoSQL systems are able to create these controls. The difference between these models is in the amount of effort required by application developers and the location (tier) of the transactional controls.

Let's start with a simple banking example to represent a reliable transaction. These days, many people have two bank accounts: savings and checking. If you want to move funds from one account to the other, it's likely your bank has a transfer form on their website. This is illustrated in figure 2.7.

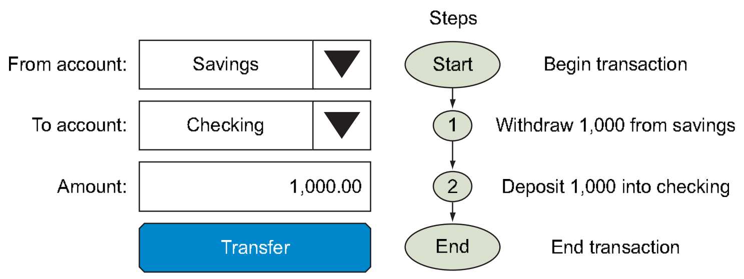


Figure 2.7 The atomic set of steps needed to transfer funds from one bank account to another. The first step subtracts the transfer amount from the source savings account. The second step adds the same transfer amount to the destination checking account. For the transaction to be considered reliable, both steps must work or both need to be undone. Between steps, no reports should be allowed to run that show the total amount as dropping by the transaction amount.

When you click the Transfer button on the web page, two discrete operations must happen in unison. The funds are subtracted from your savings account and then added to your checking account. Transaction management is the process of making sure that these two operations happen together as a single unit of work or not at all. If the computer crashes after the first part of the transaction is complete and before the second part of the transaction occurs, you'd be out $1,000 and very unhappy with your bank.

Traditional commercial RDBMSs are noted for their reliability in performing financial transactions. This reputation has been earned not only because they've been around for a long time and diligently debugged their software, but also because they've made it easy for programmers to make transactions reliable by wrapping critical transactions in statements that indicate where transactions begin and end. These are often called BEGIN TRANSACTION and END TRANSACTION statements. By adding them, developers can get high-reliability transaction support. If either one of the two atomic units doesn't complete, both of the operations will be rolled back to their initial settings.

The software also ensures that no reports can be run on the accounts halfway through the operations. If you run a "combined balance" report during the transaction, it'd never show a total that drops by 1,000 and then increases again. If a report starts while the first part of the transaction is in process, it'll be blocked until all parts of the transaction are complete.

In traditional RDBMSs the transaction management complexity is the responsibility of the database layer. Application developers only need to be able to deal with what to do if an entire transaction fails and how to notify the right party or how to keep retrying until the transaction is complete. Application developers don't need to know how to undo various parts of a transaction, as that's built into the database.

Given that reliable transactions are important in most application systems, the next two sections will take an in-depth look at RDBMS transaction control using ACID, and NoSQL transaction control using BASE.

2.5.1 RDBMS transaction control using ACID

RDBMSs maintain transaction control by using atomic, consistent, independent, and durable (ACID) properties to insure transactions are reliable. The following defines

- *Atomicity*—In the banking transaction example, we said that the exchange of funds from savings to checking must happen as an all-or-nothing transaction. The technical term for this is *atomicity*, which comes from the Greek term for "dividable." Systems that claim they have atomic transactions must consider all failure modes: disk crashes, network failures, hardware failures, or simple software errors. Testing atomic transactions even on a single CPU
- *Consistency*—In the banking transaction example, we talked about the fact that when moving funds between two related accounts, the total account balance

must never change. This is the principle of consistency. It means that your database must never have a report that shows the withdrawal from savings has occurred but the addition to checking hasn't. It's the responsibility of the database to block all reports during atomic operations. This has an impact on the speed of a system when many atomic transactions and reports are all being run

- *Isolation*—Isolation refers to the concept that each part of a transaction occurs without knowledge of any other transaction. For example, the transaction that adds funds doesn't know about the transaction that subtracts funds from an

- *Durability*—Durability refers to the fact that once all aspects of a transaction are complete, it's permanent. Once the transfer button is selected, you have the right to spend the money in your checking account. If the banking system crashes that night and they have to restore the database from a backup tape, there must be some way to make sure the record of this transfer is also restored. This usually means that the bank must create a transaction log on a separate computer system and then play back the transactions from the log after the backup is complete.

If you think that the software to handle these rules must be complex, you're right; it's very complex and one of the reasons that relational databases can be expensive. If you're writing a database on your own, it could easily double or triple the amount of software that has to be written. This is why new databases frequently don't support database-level transaction management in their first release. That's added only after the product matures.

Many RDBMSs restrict transaction location to a single CPU. If you think about the situation where your savings account information is stored in a computer in New York and your checking account information is stored in a computer in San Francisco, the complexity increases, since you have a greater number of failure points and the number of reporting systems that must be blocked on both systems increases.

Although supporting ACID transactions is complex, there are well-known and well-publicized strategies to do this. All of them depend on *locking* resources, putting extra copies of the resources aside, performing the transaction and then, if all is well, unlocking the resources. If any part of a transaction fails, the original resource in question must be returned to its original state. The design challenge is to create systems that support these transactions, make it easy for the application to use transactions, and maintain database speed and responsiveness.

ACID systems focus on the consistency and integrity of data above all other considerations. Temporarily blocking reporting mechanisms is a reasonable compromise to ensure your systems return reliable and accurate information. ACID systems are said to be pessimistic in that they must consider all possible failure modes in a computing environment. At times ACID systems seem to be guided by Murphy's Law—if anything *can* go wrong it *will* go wrong—and must be carefully tested in order to guarantee the integrity of transactions.

While ACID systems focus on high data integrity, NoSQL systems that use BASE take into consideration a slightly different set of constraints. What if blocking one transaction while you wait for another to finish is an unacceptable compromise? If you have a website that's taking orders from customers, sometimes ACID systems are *not* what you want.

2.5.2 Non-RDBMS transaction control using BASE

What if you have a website that relies on computers all over the world? A computer in Chicago manages your inventory, product photos are on an image database in Virgina, tax calculations are performed in Seattle, and your accounting system is in Atlanta. What if one site goes down? Should you tell your customers to check back in 20 minutes while you solve the problem? Only if your goal is to drive them to your competitors. Is it realistic to use ACID software for every order that comes in? Let's look at another option.

Websites that use the "shopping cart" and "checkout" constructs have a different primary consideration when it comes to transaction processing. The issue of reports that are inconsistent for a few minutes is less important than something that prevents you from taking an order, because if you block an order, you've lost a customer. The alternative to ACID is BASE

- *Basic availability* allows systems to be temporarily inconsistent so that transactions are manageable. In BASE systems, the information and service capability

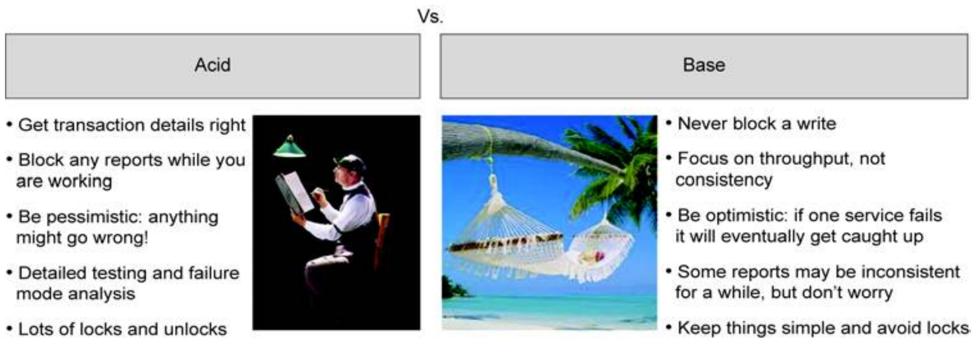
- *Soft-state* recognizes that some inaccuracy is temporarily allowed and data may

- *Eventual consistency* means eventually, when all service logic is executed, the system is left in a consistent state.

Unlike RDBMSs that focus on consistency, BASE systems focus on availability. BASE systems are noteworthy because their number-one objective is to allow new data to be stored, even at the risk of being out of sync for a short period of time. They relax the rules and allow reports to run even if not all portions of the database are synchronized. BASE systems aren't considered *pessimistic* in that they don't fret about the details if one process is behind. They're *optimistic* in that they assume that eventually all systems will catch up and become consistent.

BASE systems tend to be simpler and faster because they don't have to write code that deals with locking and unlocking resources. Their mission is to keep the process moving and deal with broken parts at a later time. BASE systems are ideal for web storefronts, where filling a shopping cart and placing an order is the main priority.

Prior to the NoSQL movement, most database experts considered ACID systems to be the only type of transactions that could be used in business. NoSQL systems are



- Get transaction details right
- Block any reports while you are working
- Be pessimistic: anything might go wrong!
- Detailed testing and failure mode analysis
- Lots of locks and unlocks

- Never block a write
- Focus on throughput, not consistency
- Be optimistic: if one service fails it will eventually get caught up
- Some reports may be inconsistent for a while, but don't worry
- Keep things simple and avoid locks

Figure 2.8 ACID versus BASE—understanding the trade-offs. This figure compares the rigid financial accounting rules of traditional RDBMS ACID transactions with the more laid-back BASE approach used in NoSQL systems. RDBMS ACID systems are ideal when all reports must always be consistent and reliable. NoSQL BASE systems are preferred when priority is given to never blocking a write transaction. Your business requirements will determine whether traditional RDBMS or NoSQL systems are right for your application.

highly decentralized and ACID guarantees may not be necessary, so they use BASE and take a more relaxed approach. Figure 2.8 shows an accurate and somewhat humorous representation of ACID versus BASE philosophies.

A final note: ACID and BASE aren't rigid points on a line; they lie on a continuum where organizations and systems can decide where and how to architect systems. They may allow ACID transactions on some key areas but relax them in others. Some database systems offer both options by changing a configuration file or using a different API. The systems administrator and application developer work together to implement the right choice after considering the needs of the business.

Transactions are important when you move from centralized to distributed systems that need to scale in order to handle large volumes of data. But there are times when the amount of data you manage exceeds the size of your current system and you need

2.6 Achieving horizontal scalability with database sharding

As the amount of data an organization stores increases, there may come a point when the amount of data needed to run the business exceeds the current environment and some mechanism for breaking the information into reasonable chunks is required. Organizations and systems that reach this capacity can use automatic database *sharding* (breaking a database into chunks called *shards* and spreading the chunks across a number of distributed servers) as a means to continuing to store data while minimizing system downtime. On older systems this might mean taking the system down for a few hours while you manually reconfigure the database and copy data from the old system to a new system, yet NoSQL systems do this automatically. How a database grows and its tolerance for automatic partitioning of data is important to NoSQL systems. Sharding has become a highly automated process in both big data and fault-tolerant systems. Let's look at how sharding works and explore its challenges.

Let's say you've created a website that allows users to log in and create their own personal space to share with friends. They have profiles, text, and product information on things they like (or don't like). You set up your website, store the information in a MySQL database, and you run it on a single CPU. People love it, they log in, create pages, invite their friends, and before you realize it your disk space is 95% full. What do you do? If you're using a typical RDBMS system, the answer is buy a new system and transfer half the users to the new system. Oh, and your old system might have to be down for a while so you can rewrite your application to know which database to get information from. Figure 2.9 shows a typical example of database sharding.

The process of moving from a single to multiple databases can be done in a num-

1. You can keep the users with account names that start with the letters A-N on the first drive and put users from O-Z
2. You can keep the people in the United States on the original system and put the
3. You can randomly move half the users to the new system.

Each of these alternatives has pros and cons. For example, in option 1, if a user changes their name, should they be automatically moved to the new drive? In option 2, if a user moves to a new country, should all their data be moved? If people tend to share links with people near them, would there be performance advantages to keeping these users together? What if people in the United States tend to be active at the same time in the evening? Would one database get overwhelmed and the other be

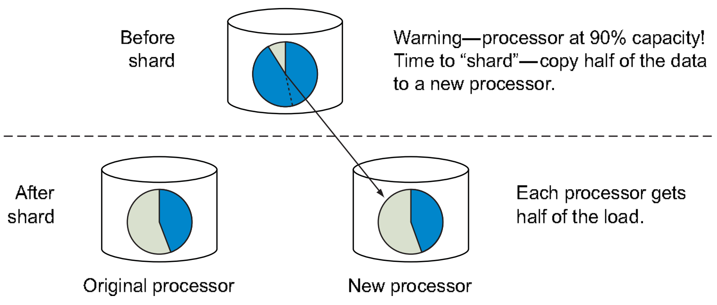

Before shard
Warning—processor at 90% capacity! Time to "shard"—copy half of the data to a new processor.
After shard
Each processor gets half of the load.
Original processor
New processor

Figure 2.9 Sharding is performed when a single processor can't handle the throughput requirements of a system. When this happens you'll want to move the data onto two systems that each take half the work. Many NoSQL systems have automatic sharding built in so that you only need to add a new server to a pool of working nodes and the database management system automatically moves data to the new node. Most RDBMSs don't support automatic sharding.

idle? What happens if your site doubles in size again? Do you have to continue to rewrite your code each time this happens? Do you have to shut the system down for a weekend while you upgrade your software?

As the number of servers grows, you find that the chance of any one server being down remains the same, so for every server you add the chance of one part not working increases. So you think that perhaps the same process you used to split the database between two systems can also be used to duplicate data to a backup or mirrored system if the first one fails. But then you have another problem. When there are changes to a master copy, you must also keep the backup copies in sync. You must have a method of data replication. The time it takes to keep these databases in sync can decrease system performance. You now need more servers to keep up!

Welcome to the world of database sharding, replication, and distributed computing. You can see that there are many questions and trade-offs to consider as your database grows. NoSQL systems have been noted for having many ways to allow you to grow your database without ever having to shut down your servers. Keeping your database running when there are node or network failures is called *partition tolerance*—a new concept in the NoSQL community and one that traditional database managers struggle with.

Understanding transaction integrity and autosharding is important with respect to how you think about the trade-offs you're faced with when building distributed systems. Though database performance, transaction integrity, and how you use memory and autosharding are important, there are times when you must identify those system aspects that are most important and focus on them while leaving others flexible. Using a formal process to understand the trade-offs in your selection process will help drive your focus toward things most important to your organization, which we turn to

2.7 Understanding trade-offs with Brewer's CAP theorem

In order to make the best decision about what to do when systems fail, you need to consider the properties of consistency and availability when working with distributed systems over unreliable networks.

Eric Brewer first introduced the *CAP* theorem in 2000. The CAP theorem states that any distributed database system can have at most two of the following three desirable

- *Consistency*—Having a single, up-to-date, readable version of your data available to all clients. This isn't the same as the consistency we talked about in ACID. Consistency here is concerned with multiple clients reading the same items

- *High availability*—Knowing that the distributed database will always allow database clients to update items without delay. Internal communication failures between replicated data shouldn't prevent updates.

- *Partition tolerance*—The ability of the system to keep responding to client requests even if there's a communication failure between database partitions. This is analogous to a person still having an intelligent conversation even after a link between parts of their brain isn't working.

Remember that the CAP theorem only applies in cases when there's a broken connection between partitions in your cluster. The more reliable your network, the lower the probability you'll need to think about CAP.

The CAP theorem helps you understand that once you partition your data, you must consider the availability-consistency spectrum in a network failure situation. Then the CAP theorem allows you to determine which options best match your business requirements. Figure 2.10 provides an example of the CAP application.

The client writes to a primary master node, which replicates the data to another backup slave node. CAP forces you to think about whether you accept a write if the communication link between the nodes is down. If you accept it, you must take responsibility for making sure the remote node gets the update at a later time, and you risk a client reading inconsistent values until the link is restored. If you refuse the write, you sacrifice availability and the client must retry later.

Although the CAP theorem has been around since 2000, it's still a source of confusion. The CAP theorem limits your design options in a few rare end cases and usually only applies when there are network failures between data centers. In many cases, reliable message queues can quickly restore consistency after network failures.

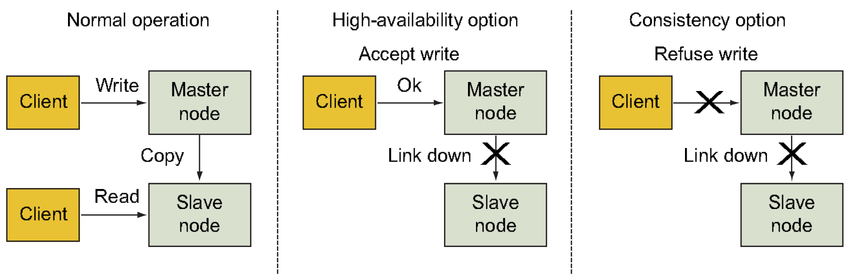


Figure 2.10 The partition decision. The CAP theorem helps you decide the relative merits of availability versus consistency when a network fails. In the left panel, under normal operation a client write will go to a master and then be replicated over the network to a slave. If the link is down, the client API can decide the relative merits of high availability or consistency. In the middle panel, you accept a write and risk inconsistent reads from the slave. In the right panel, you choose consistency and block the client write until the link between the data centers is restored.

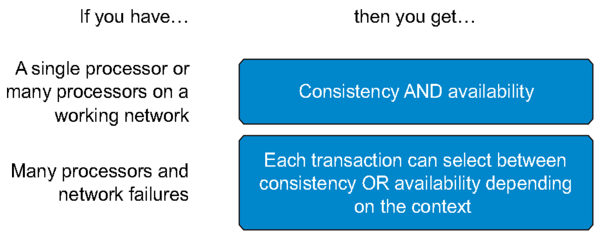

If you have...
then you get...
A single processor or many processors on a working network
Consistency AND availability
Many processors and network failures
Each transaction can select between consistency OR availability depending on the context

Figure 2.11 The CAP theorem shows that you can have both consistency and availability if you're only using a single processor. If you're using many processors, you can chose between consistency and availability depending on the transaction type, user, estimated downtime, or other factors.

The rules about when the CAP theorem applies are summarized in figure 2.11.

Tools like the CAP theorem can help guide database selection discussions within an organization and prioritize what properties (consistency, availability, and scalability) are most important. If high consistency and update availability are simultaneously required, then a faster single processor might be your best choice. If you need the scale-out benefits that distributed systems offer, then you can make decisions about your need for update availability versus read consistency for each transaction type.

Whichever option you choose, the CAP theorem provides you with a formal process that can help you weigh the pros and cons of each SQL or NoSQL system, and in

2.8 Apply your knowledge

Sally has been assigned to help a team design a system to manage loyalty gift cards, which are similar to bank accounts. Card holders can add value to a card (deposit), make a purchase (withdrawal), and verify the card's balance. Gift card data will be partitioned and replicated to two data centers, one in the U.S. and one in Europe. People who live in the U.S. will have their primary partition in the U.S. data center and people in Europe will have their primary partition in Europe.

The data line between the two data centers has been known to fail for short periods of time, typically around 10-20 minutes each year. Sally knows this is an example of a *split partition* and that it'll test the system's partition tolerance. The team needs to decide whether all three operations (deposit, withdraw, and balance) must continue when the data line is down.

The team decides that deposits should continue to work even if the data line is down, since a record of the deposit can update both sites later when the connection is restored. Sally mentions that split partitions may generate inconsistent read results if one site can't update the other site with new balance information. But the team decides that bank balance requests that occur when the link is down should still return the last balance known to the local partition.

For purchase transactions, the team decides that the transaction should go through during a link failure as long as the user is connecting to the primary partition. To limit risk, withdrawals to the replicated partition will only work if the transaction is under a specific amount, such as $100. Reports will be used to see how often multiple withdrawals on partitions generate a negative balance during network outages.

2.9 Summary

In this chapter, we covered some of the key concepts and insights of the NoSQL movement. Here's a list of the important concepts and architectural guidelines we've dis-

-
-
-
- Use distributed caching, RAM, and SSD
- Relaxing ACID
-
- The CAP theorem allows you to make intelligent choices when there's a network failure.

Throughout this book we emphasize the importance of using a formal process in evaluating systems to help identify what aspects are most important to the organization and what compromises need to be made.

At this point you should understand the benefits of using NoSQL systems and how they'll assist you in meeting your business objectives. In the next chapter, we'll build on our pattern vocabulary and review the strengths and weaknesses of RDBMS architectures, and then move on to patterns that are associated with NoSQL data architec-

2.10 Further reading

- http://mng.bz/54gQ
- http://mng.bz/Wfm5
- http://mng.bz/Z09P
- http://mng.bz/157p
- http://mng.bz/U5tm
- Preshing, Jeff. "Hash Collision Probabilities." Preshing on Programming. May 4, http://mng.bz/PxDU
- http://mng.bz/w2P8
- http://mng.bz/sg4R
- http://www.w3.org/TR/xproc/
- http://www.xmlsh.org.