Quorum (distributed computing)

From Wikipedia, the free encyclopedia
Jump to navigationJump to search

A **quorum** is the minimum number of votes that a distributed transaction has to obtain in order to be allowed to perform an operation in a distributed system. A **quorum**-based technique is implemented to enforce consistent operation in a distributed system.

Contents

- 1Quorum-based techniques in distributed database systems
  - 1.1Quorum-based voting in commit protocols
  - 1.2Quorum-based voting for replica control
- 2See also
- 3References

Quorum-based techniques in distributed database systems[edit]

Quorum-based voting can be used as a replica control method,[1] as well as a commit method to ensure transaction atomicity in the presence of network partitioning.[1]

Quorum-based voting in commit protocols[edit]

In a distributed database system, a transaction could be executing its operations at multiple sites. Since atomicity requires every distributed transaction to be atomic, the transaction must have the same fate (commit or abort) at every site. In case of network partitioning, sites are partitioned and the partitions may not be able to communicate with each other. This is where a quorum-based technique comes in. The fundamental idea is that a transaction is executed if the majority of sites vote to execute it.

Every site in the system is assigned a vote V_i. Let us assume that the total number of votes in the system is V and the abort and commit quorums are V_a and V_c, respectively. Then the following rules must be obeyed in the implementation of the commit protocol:

1. $V_a + V_c > V$, where $0 < V_c, V_a$ V.
2. Before a transaction commits, it must obtain a commit quorum V_c.
   The total of at least one site that is prepared to commit and zero or more sites

   waiting V_c.[2]
3. Before a transaction aborts, it must obtain an abort quorum V_a

   The total of zero or more sites that are prepared to abort or any sites waiting V_a.

The first rule ensures that a transaction cannot be committed and aborted at the same time. The next two rules indicate the votes that a transaction has to obtain before it can terminate one way or the other.

Quorum-based voting for replica control[edit]

In replicated databases, a data object has copies present at several sites. To ensure serializability, no two transactions should be allowed to read or write a data item concurrently. In case of replicated

databases, a quorum-based replica control protocol can be used to ensure that no two copies of a data item are read or written by two transactions concurrently.

The quorum-based voting for replica control is due to [Gifford, 1979].[3] Each copy of a replicated data item is assigned a vote. Each operation then has to obtain a *read quorum* (V_r) or a *write quorum* (V_w) to read or write a data item, respectively. If a given data item has a total of V votes, the quorums have to obey the following rules:

1. $V_r + V_w > V$
2. $V_w > V/2$

The first rule ensures that a data item is not read and written by two transactions concurrently. Additionally, it ensures that a read quorum contains at least one site with the newest version of the data item. The second rule ensures that two write operations from two transactions cannot occur concurrently on the same data item. The two rules ensure that one-copy serializability is maintained.

Chapter 4. Distribution Models

The primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on. A more appealing option is to scale out—run the database on a cluster of servers. Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution.

Depending on your distribution model, you can get a data store that will give you the ability to handle larger quantities of data, the ability to process a greater read or write traffic, or more availability in the face of network slowdowns or breakages. These are often important benefits, but they come at a cost. Running over a cluster introduces complexity—so it’s not something to do unless the benefits are compelling.

Broadly, there are two paths to data distribution: replication and sharding. Replication takes the same data and copies it over multiple nodes. Sharding puts different data on different nodes. Replication and sharding are orthogonal techniques: You can use either or both of them. Replication comes into two forms: master-slave and peer-to-peer. We will now discuss

these techniques starting at the simplest and working up to the more complex: first single-server, then master-slave replication, then sharding, and finally peer-to-peer replication.

4.1. SINGLE SERVER

The first and the simplest distribution option is the one we would most often recommend—no distribution at all. Run the database on a single machine that handles all the reads and writes to the data store. We prefer this option because it eliminates all the complexities that the other options introduce; it's easy for operations people to manage and easy for application developers to reason about.

Although a lot of NoSQL databases are designed around the idea of running on a cluster, it can make sense to use NoSQL with a single-server distribution model if the data model of the NoSQL store is more suited to the application. Graph databases are the obvious category here—these work best in a single-server configuration. If your data usage is mostly about processing aggregates, then a single-server document or key-value store may well be worthwhile because it's easier on application developers.

For the rest of this chapter we'll be wading through the advantages and complications of more sophisticated distribution schemes. Don't let the volume of words fool you into thinking that we would prefer these options. If we can get away without distributing our data, we will always choose a single-server approach.

4.2. SHARDING

Often, a busy data store is busy because different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers—a technique that's called **sharding** (see Figure 4.1).

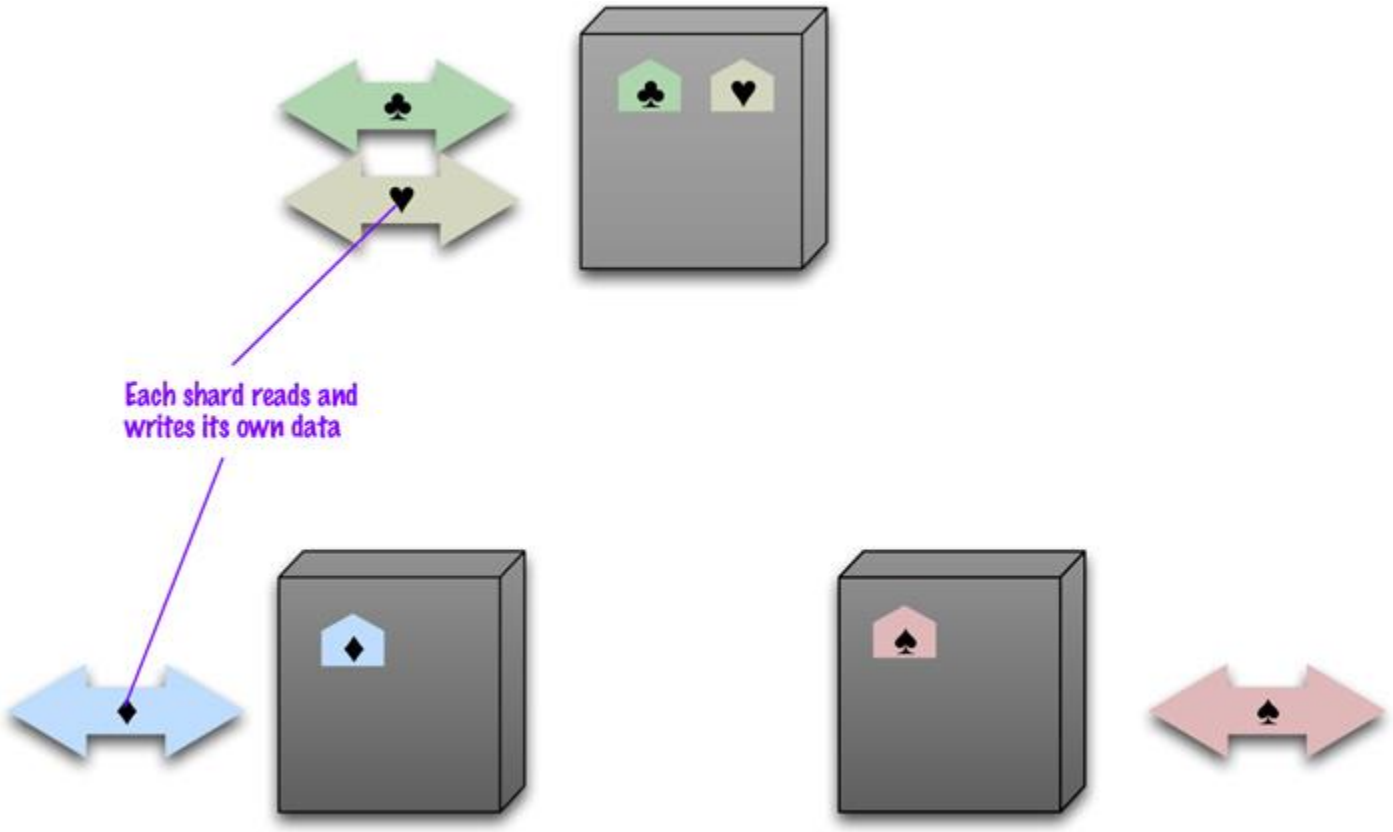

Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

In the ideal case, we have different users all talking to different server nodes. Each user only has to talk to one server, so gets rapid responses from that server. The load is balanced out nicely between servers—for example, if we have ten servers, each one only has to handle 10% of the load.

Of course the ideal case is a pretty rare beast. In order to get close to it we have to ensure that data that's accessed together is clumped together on the same node and that these clumps are arranged on the nodes to provide the best data access.

The first part of this question is how to clump the data up so that one user mostly gets her data from a single server. This is where aggregate orientation comes in really handy. The whole point of aggregates is that we design them to combine data that's commonly accessed together—so aggregates leap out as an obvious unit of distribution.

When it comes to arranging the data on the nodes, there are several factors that can help improve performance. If you know that most accesses of

certain aggregates are based on a physical location, you can place the data close to where it's being accessed. If you have orders for someone who lives in Boston, you can place that data in your eastern US data center.

Another factor is trying to keep the load even. This means that you should try to arrange aggregates so they are evenly distributed across the nodes which all get equal amounts of the load. This may vary over time, for example if some data tends to be accessed on certain days of the week—so there may be domain-specific rules you'd like to use.

In some cases, it's useful to put aggregates together if you think they may be read in sequence. The Bigtable paper [Chang etc.] described keeping its rows in lexicographic order and sorting web addresses based on reversed domain names (e.g., `com.martinfowler`). This way data for multiple pages could be accessed together to improve processing efficiency.

Historically most people have done sharding as part of application logic. You might put all customers with surnames starting from A to D on one shard and E to G on another. This complicates the programming model, as application code needs to ensure that queries are distributed across the various shards. Furthermore, rebalancing the sharding means changing the application code and migrating the data. Many NoSQL databases offer **auto-sharding**, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard. This can make it much easier to use sharding in an application.

Sharding is particularly valuable for performance because it can improve both read and write performance. Using replication, particularly with caching, can greatly improve read performance but does little for applications that have a lot of writes. Sharding provides a way to horizontally scale writes.

Sharding does little to improve resilience when used alone. Although the data is on different nodes, a node failure makes that shard's data unavailable just as surely as it does for a single-server solution. The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it's not good to have a database with part of its data missing. With a single server it's easier to pay the effort and cost to keep that server up and running; clusters usually try to use less reliable machines, and you're more likely to get a node failure. So in practice, sharding alone is likely to decrease resilience.

Despite the fact that sharding is made much easier with aggregates, it's still not a step to be taken lightly. Some databases are intended from the beginning to use sharding, in which case it's wise to run them on a cluster

from the very beginning of development, and certainly in production. Other databases use sharding as a deliberate step up from a single-server configuration, in which case it's best to start single-server and only use sharding once your load projections clearly indicate that you are running out of headroom.

In any case the step from a single node to sharding is going to be tricky. We have heard tales of teams getting into trouble because they left sharding to very late, so when they turned it on in production their database became essentially unavailable because the sharding support consumed all the database resources for moving the data onto new shards. The lesson here is to use sharding well before you need to—when you have enough headroom to carry out the sharding.

4.3. MASTER-SLAVE REPLICATION

With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary. This master is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master (see Figure 4.2).

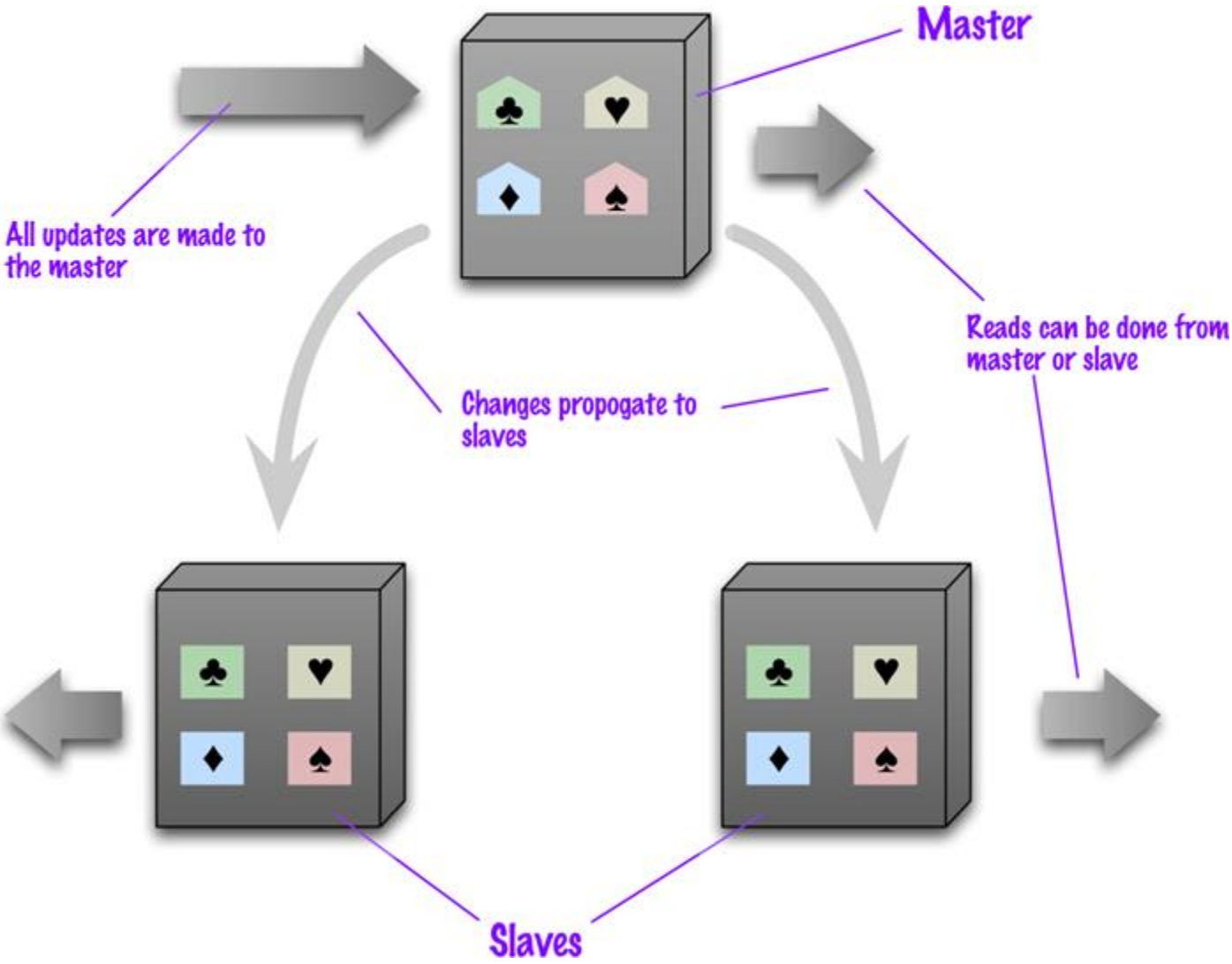

Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

Master-slave replication is most helpful for scaling when you have a read-intensive dataset. You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves. You are still, however, limited by the ability of the master to process updates and its ability to pass those updates on. Consequently it isn't such a good scheme for datasets with heavy write traffic, although offloading the read traffic will help a bit with handling the write load.

A second advantage of master-slave replication is **read resilience**: Should the master fail, the slaves can still handle read requests. Again, this is useful if most of your data access is reads. The failure of the master does eliminate the ability to handle writes until either the master is restored or a new

master is appointed. However, having slaves as replicates of the master does speed up recovery after a failure of the master since a slave can be appointed a new master very quickly.

The ability to appoint a slave to replace a failed master means that master-slave replication is useful even if you don't need to scale out. All read and write traffic can go to the master while the slave acts as a hot backup. In this case it's easiest to think of the system as a single-server store with a hot backup. You get the convenience of the single-server configuration but with greater resilience—which is particularly handy if you want to be able to handle server failures gracefully.

Masters can be appointed manually or automatically. Manual appointing typically means that when you configure your cluster, you configure one node as the master. With automatic appointment, you create a cluster of nodes and they elect one of themselves to be the master. Apart from simpler configuration, automatic appointment means that the cluster can automatically appoint a new master when a master fails, reducing downtime.

In order to get read resilience, you need to ensure that the read and write paths into your application are different, so that you can handle a failure in the write path and still read. This includes such things as putting the reads and writes through separate database connections—a facility that is not often supported by database interaction libraries. As with any feature, you cannot be sure you have read resilience without good tests that disable the writes and check that reads still occur.

Replication comes with some alluring benefits, but it also comes with an inevitable dark side—inconsistency. You have the danger that different clients, reading different slaves, will see different values because the changes haven't all propagated to the slaves. In the worst case, that can mean that a client cannot read a write it just made. Even if you use master-slave replication just for hot backup this can be a concern, because if the master fails, any updates not passed on to the backup are lost. We'll talk about how to deal with these issues later ("Consistency," p. 47).

4.4. PEER-TO-PEER REPLICATION

Master-slave replication helps with read scalability but doesn't help with scalability of writes. It provides resilience against failure of a slave, but not of a master. Essentially, the master is still a bottleneck and a single point of failure. Peer-to-peer replication (see Figure 4.3) attacks these problems by

not having a master. All the replicas have equal weight, they can all accept writes, and the loss of any of them doesn't prevent access to the data store.

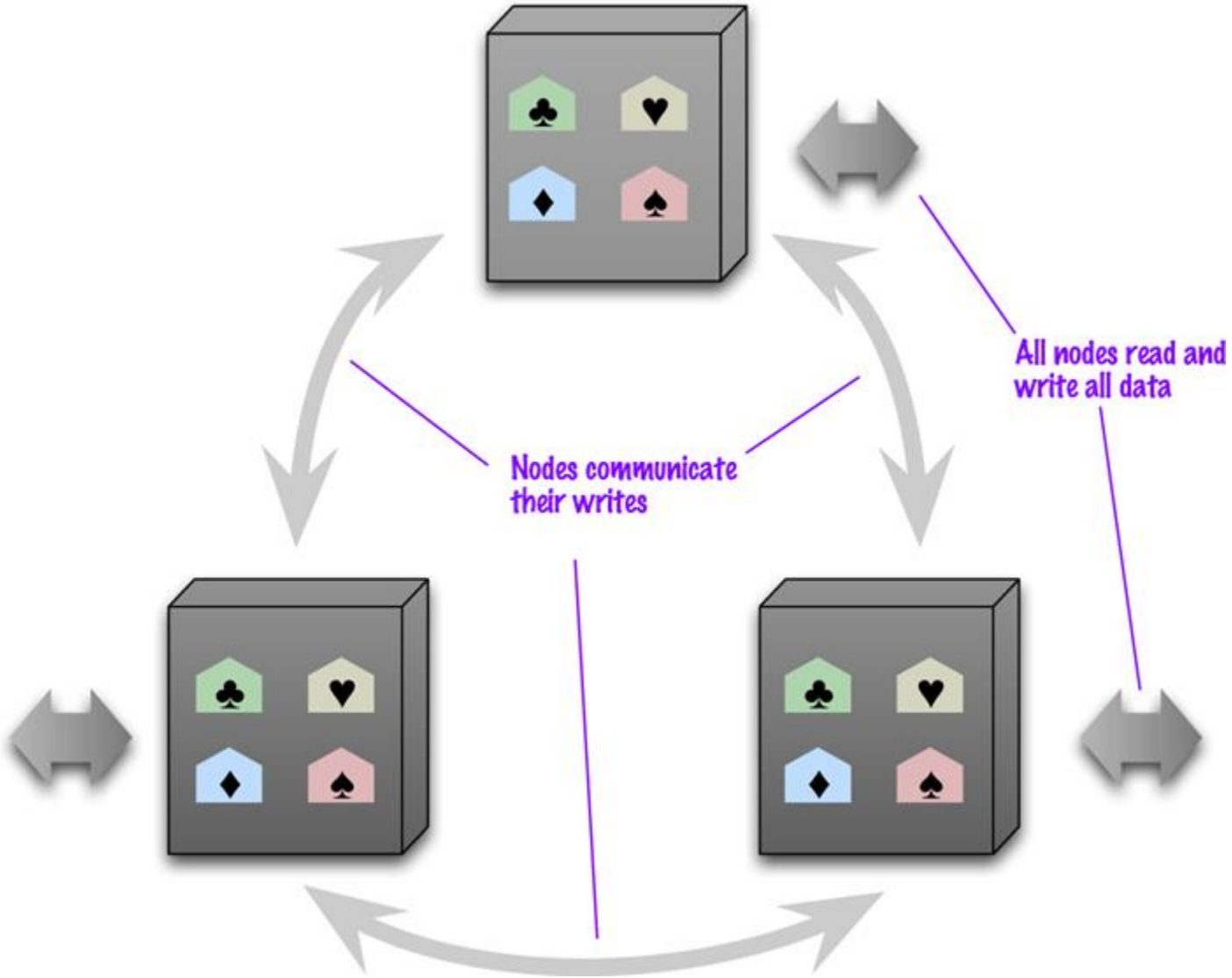

Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

The prospect here looks mighty fine. With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance. There's much to like here—but there are complications.

The biggest complication is, again, consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time—a write-write conflict. Inconsistencies on

read lead to problems but at least they are relatively transient. Inconsistent writes are forever.

We'll talk more about how to deal with write inconsistencies later on, but for the moment we'll note a couple of broad options. At one end, we can ensure that whenever we write data, the replicas coordinate to ensure we avoid a conflict. This can give us just as strong a guarantee as a master, albeit at the cost of network traffic to coordinate the writes. We don't need all the replicas to agree on the write, just a majority, so we can still survive losing a minority of the replica nodes.

At the other extreme, we can decide to cope with an inconsistent write. There are contexts when we can come up with policy to merge inconsistent writes. In this case we can get the full performance benefit of writing to any replica.

These points are at the ends of a spectrum where we trade off consistency for availability.

Introduction to JSON

JSON is a text-based data exchange format derived from JavaScript that is used in web services and other connected applications. The following sections provide an introduction to JSON syntax, an overview of JSON uses, and a description of the most common approaches to generate and parse JSON.

The following topics are addressed here:

- JSON Syntax
- Uses of JSON
- Generating and Parsing JSON Data

JSON Syntax

JSON defines only two data structures: objects and arrays. An object is a set of name-value pairs, and an array is a list of values. JSON defines seven value types: string, number, object, array, true, false, and null.

The following example shows JSON data for a sample object that contains name-value pairs. The value for the name `"phoneNumbers"` is an array whose elements are two objects.

```
{

   "firstName": "Duke",

   "lastName": "Java",

   "age": 18,

   "streetAddress": "100 Internet Dr",

   "city": "JavaTown",

   "state": "JA",

   "postalCode": "12345",

   "phoneNumbers": [

      { "Mobile": "111-111-1111" },

      { "Home": "222-222-2222" }

   ]

}
```

JSON has the following syntax.

- Objects are enclosed in braces (`{}`), their name-value pairs are separated by a comma (`,`), and the name and value in a pair are separated by a colon (`:`). Names in an object are strings, whereas values may be of any of the seven value types, including another object or an array.
- Arrays are enclosed in brackets (`[]`), and their values are separated by a comma (`,`). Each value in an array may be of a different type, including another array or an object.
- When objects and arrays contain other objects or arrays, the data has a tree-like structure.

Uses of JSON

JSON is often used as a common format to serialize and deserialize data in applications that communicate with each other over the Internet. These applications are created using different programming languages and run in very different environments. JSON is suited to this scenario because it is an open standard, it is easy to read and write, and it is more compact than other representations.

RESTful web services use JSON extensively as the format for the data inside requests and responses. The HTTP header used to indicate that the content of a request or a response is JSON data is

```
Content-Type: application/json
```

JSON representations are usually more compact than XML representations because JSON does not have closing tags. Unlike XML, JSON does not have a widely accepted schema for defining and validating the structure of JSON data.

Generating and Parsing JSON Data

For generating and parsing JSON data, there are two programming models, which are similar to those used for XML documents.

- The object model creates a tree that represents the JSON data in memory. The tree can then be navigated, analyzed, or modified. This approach is the most flexible and allows for processing that requires access to the complete contents of the tree. However, it is often slower than the streaming model and requires more memory. The object model generates JSON output by navigating the entire tree at once.
- The streaming model uses an event-based parser that reads JSON data one element at a time. The parser generates events and stops for processing when an object or an array begins or ends, when it finds a key, or when it finds a value. Each element can be processed or discarded by the application code, and then the parser proceeds to the next event. This approach is adequate for local processing, in which the processing of an element does not require information from the rest of the data. The streaming model generates JSON output to a given stream by making a function call with one element at a time.

There are many JSON generators and parsers available for different programming languages and environments. JSON Processing in the Java EE Platform describes the functionality provided by the Java API for JSON Processing (JSR 374)

Introduction to NoSQL query

Last updated on: Jun 19, 2016

- Discuss on HN
- Tweet

If NoSQL is all about leaving SQL behind, then what takes its place?

First we need to clear something up: NoSQL is a bad name. Really these are non-relational databases. There's a good reason for that confusion, though.

Up until recently, data model and query method were closely tied to each other. That's understandable: getting the most out of limited hardware has been a leading driver of database development. So query method has been, for much of the history of database development, a product of the data model.
Today we have the luxury of more compute power. Thanks to cloud computing and Moore's Law, we can create database systems that prioritise developer productivity and begin to abstract the query method from the data model. Think of it as similar to what has happend in

application development: we have taken a path from machine code, through assembly, to low level and then high level languages.

The early databases

The world's first commercial database was SABRE, a collaboration between IBM and American Airlines for improving the efficiency of airline ticketing. Before computerisation, ticket booking was a lengthy manual process taking up to 90 minutes. SABRE launched in 1960 and reduced that time to seconds.

SABRE was a hierarchical database. Hierarchical databases have an entry point at the top with links that descend through the data, much like a family tree or the Windows Registry. To query such a database the developer works down the paths of the hierarchy until they find the element they want.

More recent hierarchical databases — such as IMS, which grew out of SABRE — have since introduced declarative query but in those early days query was very much procedural:

```
Is this the thing?

No?

Okay, go to the next one down.

Is this the thing?
```

A few years after SABRE, the same group who created COBOL produced a specification for the network database. This built on the hierarchical database by adding peer-to-peer links resulting in something that worked much like a linked list. Still, query was quite manual.
It wasn't until 1970 when IBM researcher EF Codd published his paper A Relational Model of Data for Large Shared Data Banks, in which he described the relational databases we know today. However, it would be another four years before two of Codd's IBM colleagues would propose the language that became SQL and supplanted Codd's own query proposals.

The hierarchical, network and relational models each dictated the nature of the query method. While the implementations might vary, the scope of the query language was bound by the data model.

Query in NoSQL today

When the current crop of NoSQL databases came on the scene from 2005 onwards, query was not the priority. With the exception of graph databases, non-relational databases have optimised for scalability, up-time, redundancy, flexibility and usually at the expense of queryability.

So, where does that leave NoSQL query? Broadly, there have been five approaches:

- Not my problem
- Map-reduce
- DBMS-specific approaches.
- Attempts at standardisation.
- SQL-derivatives.

That last one is perhaps the strongest reason against using the term NoSQL: those same databases are increasingly implementing some form of SQL.

Not my problem/map-reduce

If you're building a database system that's designed to scale effortlessly and handle infinitely varied data, then perhaps query isn't your priority.

When talking about those databases, their proponents often talk about moving query to the application layer. In other words, you as the application developer get to do more work. The database vendor or open source project provides a REST API and some libraries that let you put data in and pull that same data out.

Actually, that's fine if you have a genuinely key-value shaped use case. If all you need is something that behaves like a file system, then the lack of query frees the DMBS creators to fix other hard problems.

Map-reduce

For some use cases, though, that's not enough.

CouchDB, arguably the first of the current NoSQL databases, drew from the big data world to offer map-reduce query. That suited its document data model and its architecture where data and compute-power could be distributed across a cluster of nodes. Riak and MongoDB also added map-reduce.

However, for all its usefulness in dealing with distributed data sets, map-reduce isn't the friendly ad-hoc declarative query that SQL led us to expect.

DBMS-specific query

MongoDB is the one non-relational database that has prioritised developer friendliness above all else.

While MongoDB offers map-reduce as an option, most MongoDB query takes the form of method chaining:

```
db.collection.command(data)
```

If we're working in the MongoDB shell and we want to find all the fish and chip restaurants in our database, we might write something like this:

```
db.restaurants.find( { "cuisine": "Fish and chips" } )
```

Each language specific driver then implements the same query in an idiomatically correct way.

This is easy to learn but it's still something new; and it doesn't integrate easily with SQL tooling, for example.

Other databases, such as ArrangoDB, also have their own query languages. While that might allow them to innovate quickly, learning something entirely new is a barrier to adoption.

Attempts at standardisation

Standards are enablers of innovation: if all document databases, for example, shared a common query language then developers could move easily from one DBMS to another and everyone could focus on innovating in areas other than the query language.

XQuery is an attempt at creating such a standard for working with XML and Jsoniq is a superset tuned for JSON. The vision for both languages is to create a standard method of querying the type of hierarchical documents we see represented in XML and JSON.

MarkLogic, a document database that works with XML, implements a form of XQuery. Here's how we'd insert a new document:

```
xquery version "1.0-ml";

xdmp:document-insert("restaurants.xml",

    <restaurants xmlns="http://example.com/namespace/restaurants">

        <restaurant restaurantid="1">

            <name>Philipp's Fish and Chips</name>

            <cuisine>Fish and chips</cuisine>

            <owner>

                <first>Philipp</first>

                <last>Strube</last>
```

```
            </owner>

        </restaurant>

    </restaurants>

)
```

To return all the fish and chip restaurants in our `restaurants.xml` file, we'd do something like:

```
xquery version "1.0-ml";

declare namespace rest = "http://example.com/namespace/restaurants";

{

    for $restaurant in doc("restaurants.xml")/rest:restaurants/rest:restaurant

    return

    {

        $restaurant/rest:name/text()

    }

}
```

Jsoniq is somewhat different to allow for the difference in the data model and that goes to show that both XQuery and Jsoniq are deeply tied to the format of the data stored.

XQuery has seen commercial implementations but none of the major document databases have implemented Jsoniq.

SQL for NoSQL

The name NoSQL shows just how closely tied SQL and the relational model are. However, two things are making that less true: increasing compute power that makes even more complex indexing and query parsers possible and SQL's place as the one query language every developer knows.

Even if you don't know SQL, it's easy to pick up. Then there's all that tooling out there: including ODBC, JDBC, query analysers and more.

Perhaps that's why more and more non-relational databases are adopting or adapting SQL. The first to do so was Cassandra, with CQL.

Cassandra's columnar data model shares some superficial similarities with the relational model but it's sufficiently far removed that Cassandra provides an intermediary representation of the data model that differs somewhat from what is actually stored on disk.

Thanks to this, CQL looks remarkably familiar. Translating our XQuery query above into CQL gives us:

```
SELECT name FROM restaurants WHERE cuisine='Fish and chips';
```

It's readable and, importantly, it's declarative: we're not telling the query processor what to do but instead we're telling it what we want to get back.
For document databases, there are two related query languages. SQL++ is a research project from the University of California San Diego that modifies SQL to work better with denormalised data. In particular, it adds keywords to handle the missing and nested data that we expect to find in a denormalised document database.
Similar to SQL++ is N1QL, Couchbase's implementation of a SQL-like language for JSON documents. N1QL (or non-first form query language) is supremely SQL-like. If we took our restaurant example, we might have one document per restaurant. To find our fish and chip restaurants we'd write:

```
SELECT name FROM restaurants WHERE cuisine='Fish and chips';
```

For this query, it's impossible to tell SQL, CQL and N1QL apart.

Both Couchbase and Cassandra have put the development effort into their indexing and query parsing to enable SQL, which was designed for relational data, to fit entirely different data models.

Despite the nesting and variable schema, document databases are pretty well suited to a SQL-like query language. Even though relations are not enforced, we create and store documents that rely on each other