

**Hiring?** Toptal handpicks [top Big Data architects](#) to suit your needs.

- [Start hiring](#)
- [Log in](#)
- 
- [Top 3%](#)
- [Why](#)
- [Clients](#)
- [Enterprise](#)
- [Partners](#)
- [Community](#)
- [Blog](#)
- [About Us](#)
- [Start hiring](#)
- [Apply as a Developer](#)
- [Login](#)
- - Questions?
  - [Contact Us](#)
  - 
  - 
  -

Search Topics

[Hire a developer](#)

## A Guide to Consistent Hashing

[View all articles](#)



by [Juan Pablo Carzolio](#) - Freelance Software Engineer @ [Toptal](#)

[#Algorithms](#) [#BigData](#) [#Hashing](#)

- 54shares



•



•



•



- 



- 



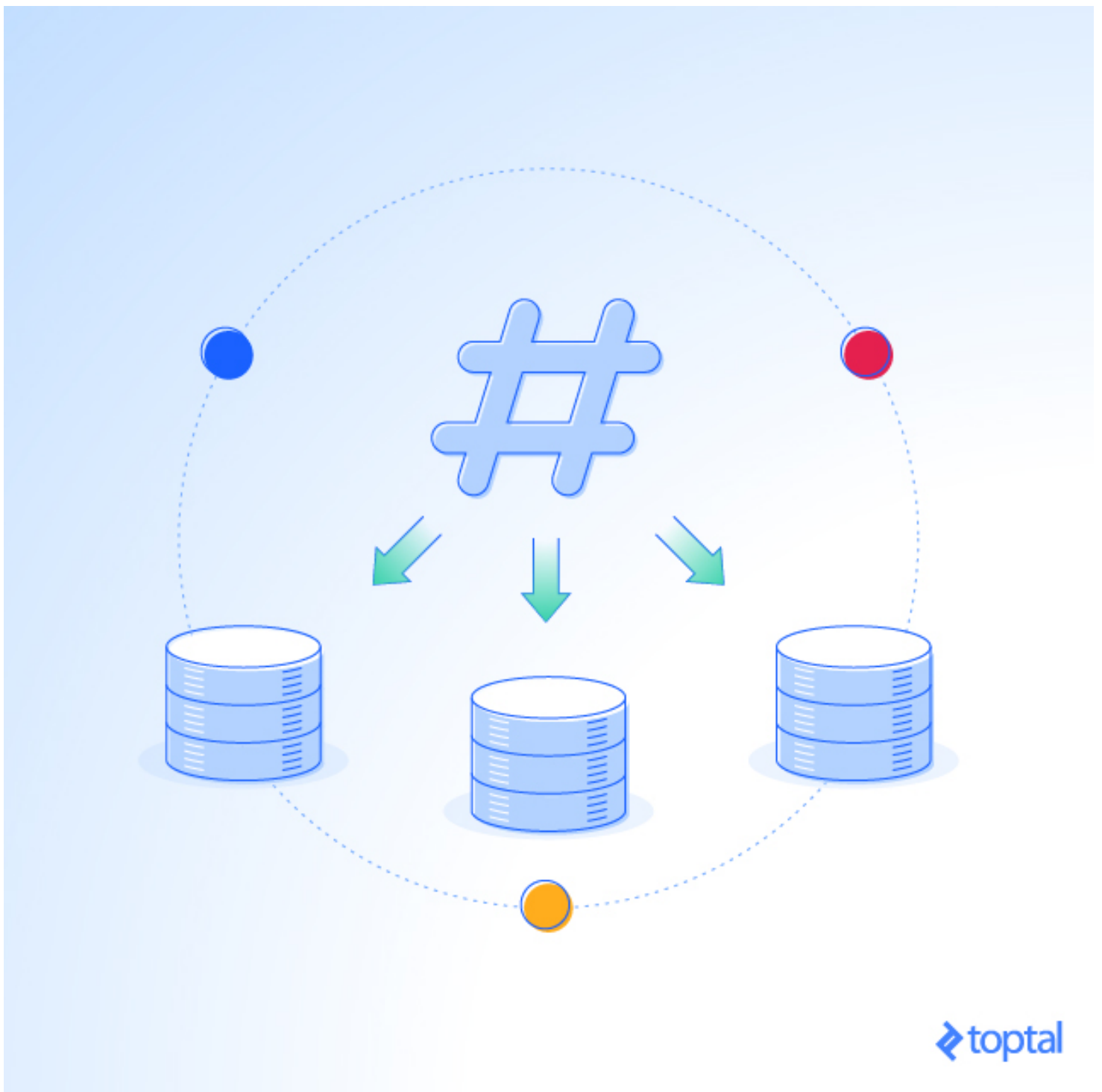
- 

In recent years, with the advent of concepts such as cloud computing and big data, distributed systems have gained popularity and relevance.

One such type of system, [distributed caches](#) that power many high-traffic dynamic websites and web applications, typically consist of a particular case of distributed hashing. These take advantage of an algorithm known as [consistent hashing](#).

What is consistent hashing? What's the motivation behind it, and why should you care?

In this article, I'll first review the general concept of hashing and its purpose, followed by a description of distributed hashing and the problems it entails. In turn, that will lead us to our title subject.



## What Is Hashing?

What is “hashing” all about? [Merriam-Webster](#) defines the noun *hash* as “chopped meat mixed with potatoes and browned,” and the verb as “to chop (as meat and potatoes) into small pieces.” So, culinary details aside, hash roughly means “chop and mix”—and that’s precisely where the technical term comes from.

A hash function is a function that maps one piece of data—typically describing some kind of object, often of arbitrary size—to another piece of data, typically an integer, known as *hash code*, or simply *hash*.

For instance, some hash function designed to hash strings, with an output range of  $0 \dots 100$ , may map the string `Hello` to, say, the number 57, `Hasta la vista, baby` to the number 33, and any other possible string to some number within that range. Since there are way more possible inputs than outputs, any given number will have many different strings mapped to it, a phenomenon known as *collision*. Good hash functions should somehow “chop and mix” (hence the term) the input data in such a way that the outputs for different input values are spread as evenly as possible over the output range.

Hash functions have many uses and for each one, different properties may be desired. There is a type of hash function known as *cryptographic hash functions*, which must meet a restrictive set of properties and are used for security purposes—including applications such as password protection, integrity checking and fingerprinting of messages, and data corruption detection, among others, but those are outside the scope of this article.

Non-cryptographic hash functions have several uses as well, the most common being their use in *hash tables*, which is the one that concerns us and which we’ll explore in more detail.

## Introducing Hash Tables (Hash Maps)

Imagine we needed to keep a list of all the members of some club while being able to search for any specific member. We could handle it by keeping the list in an array (or linked list) and, to perform a search, iterate the elements until we find the desired one (we might be searching based on their name, for instance). In the worst case, that would mean checking all members (if the one we’re searching for is last, or not present at all), or half of them on average. In complexity theory terms, the search would then have complexity  $O(n)$ , and it would be reasonably fast for a small list, but it would get slower and slower in direct proportion to the number of members.

How could that be improved? Let’s suppose all these club members had a member ID, which happened to be a sequential number reflecting the order in which they joined the club.

Assuming that searching by ID were acceptable, we could place all members in an array, with their indexes matching their IDs (for example, a member with ID=10 would be at the index 10 in the array). This would allow us to access each member directly, with no search at all. That would be very efficient, in fact, as efficient as it can possibly be, corresponding to the lowest complexity possible,  $O(1)$ , also known as *constant time*.

But, admittedly, our club member ID scenario is somewhat contrived. What if IDs were big, non-sequential or random numbers? Or, if searching by ID were not acceptable, and we needed to search by name (or some other field) instead? It would certainly be useful to keep our fast direct access (or something close) while at the same time being able to handle arbitrary datasets and less restrictive search criteria.

Here’s where hash functions come to the rescue. A suitable hash function can be used to map an arbitrary piece of data to an integer, which will play a similar role to that of our club member ID, albeit with a few important differences.

First, a good hash function generally has a wide output range (typically, the whole range of a 32 or 64-bit integer), so building an array for all possible indices would be either impractical or plain impossible, and a colossal waste of memory. To overcome that, we can have a reasonably sized array (say, just twice the number of elements we expect to store) and perform a *modulo* operation on the hash to get the array index. So, the index would be  $\text{index} = \text{hash}(\text{object}) \bmod N$ , where  $N$  is the size of the array.

Second, object hashes will not be unique (unless we're working with a fixed dataset and a custom-built [perfect hash function](#), but we won't discuss that here). There will be *collisions* (further increased by the modulo operation), and therefore a simple direct index access won't work. There are several ways to handle this, but a typical one is to attach a list, commonly known as a *bucket*, to each array index to hold all the objects sharing a given index.

So, we have an array of size  $N$ , with each entry pointing to an object bucket. To add a new object, we need to calculate its  $\text{hash} \bmod N$ , and check the bucket at the resulting index, adding the object if it's not already there. To search for an object, we do the same, just looking into the bucket to check if the object is there. Such a structure is called a *hash table*, and although the searches within buckets are linear, a properly sized hash table should have a reasonably small number of objects per bucket, resulting in *almost* constant time access (an average complexity of  $O(N/k)$ , where  $k$  is the number of buckets).

With complex objects, the hash function is typically not applied to the whole object, but to a *key* instead. In our club member example, each object might contain several fields (like name, age, address, email, phone), but we could pick, say, the email to act as the key so that the hash function would be applied to the email only. In fact, the key need not be part of the object; it is common to store key/value pairs, where the key is usually a relatively short string, and the value can be an arbitrary piece of data. In such cases, the hash table or hash map is used as a *dictionary*, and that's the way some high-level languages implement objects or associative arrays.

## Scaling Out: Distributed Hashing

Now that we have discussed hashing, we're ready to look into *distributed hashing*.

In some situations, it may be necessary or desirable to split a hash table into several parts, hosted by different servers. One of the main motivations for this is to bypass the memory limitations of using a single computer, allowing for the construction of arbitrarily large hash tables (given enough servers).

In such a scenario, the objects (and their keys) are *distributed* among several servers, hence the name.

A typical use case for this is the implementation of in-memory caches, such as [Memcached](#).

Such setups consist of a pool of caching servers that host many key/value pairs and are used to provide fast access to data originally stored (or computed) elsewhere. For example, to reduce the load on a database server and at the same time improve performance, an application can be designed to first fetch data from the cache servers, and only if it's not present there—a situation

known as *cache miss*—resort to the database, running the relevant query and caching the results with an appropriate key, so that it can be found next time it's needed.

Now, how does distribution take place? What criteria are used to determine which keys to host in which servers?

The simplest way is to take the hash *modulo* of the number of servers. That is,  $\text{server} = \text{hash}(\text{key}) \bmod N$ , where  $N$  is the size of the pool. To store or retrieve a key, the client first computes the hash, applies a  $\bmod N$  operation, and uses the resulting index to contact the appropriate server (probably by using a lookup table of IP addresses). Note that the hash function used for key distribution must be the same one across all clients, but it need not be the same one used internally by the caching servers.

Let's see an example. Say we have three servers, A, B and C, and we have some string keys with their hashes:

KEY	HASH	HASH mod 3
"john"	1633428562	2
"bill"	7594634739	0
"jane"	5000799124	1
"steve"	9787173343	0
"kate"	3421657995	2

A client wants to retrieve the value for key `john`. Its hash  $\bmod 3$  is 2, so it must contact server C. The key is not found there, so the client fetches the data from the source and adds it. The pool looks like this:

A	B	C
		"john"

Next another client (or the same one) wants to retrieve the value for key `bill`. Its hash  $\bmod 3$  is 0, so it must contact server A. The key is not found there, so the client fetches the data from the source and adds it. The pool looks like this now:

A	B	C
"bill"		"john"

After the remaining keys are added, the pool looks like this:

A

B

C

"bill"

"jane"

"john"

"steve"

"kate"

Like what you're reading?

Get the latest updates first.

Get Exclusive Updates

No spam. Just great engineering posts.

Like what you're reading?

Get the latest updates first.

Thank you for subscribing!

Check your inbox to confirm subscription. You'll start receiving posts after you confirm.

- 111shares



•



•



•

## The Rehashing Problem

This distribution scheme is simple, intuitive, and works fine. That is, until the number of servers changes. What happens if one of the servers crashes or becomes unavailable? Keys need to be redistributed to account for the missing server, of course. The same applies if one or more new servers are added to the pool; keys need to be redistributed to include the new servers. This is true for any distribution scheme, but the problem with our simple modulo distribution is that when the number of servers changes, most  $\text{hashes} \bmod N$  will change, so most keys will need to be moved to a different server. So, even if a single server is removed or added, all keys will likely need to be rehashed into a different server.

From our previous example, if we removed server C, we'd have to rehash all the keys using  $\text{hash} \bmod 2$  instead of  $\text{hash} \bmod 3$ , and the new locations for the keys would become:

KEY

HASH

HASH mod 2

"john"

1633428562

0

"bill"	7594634739	1
"jane"	5000799124	0
"steve"	9787173343	1
"kate"	3421657995	1

A	B
"john"	"bill"
"jane"	"steve"
	"kate"

Note that all key locations changed, not only the ones from server C.

In the typical use case we mentioned before (caching), this would mean that, all of a sudden, the keys won't be found because they won't yet be present at their new location.

So, most queries will result in misses, and the original data will likely need retrieving again from the source to be rehashed, thus placing a heavy load on the origin server(s) (typically a database). This may very well degrade performance severely and possibly crash the origin servers.

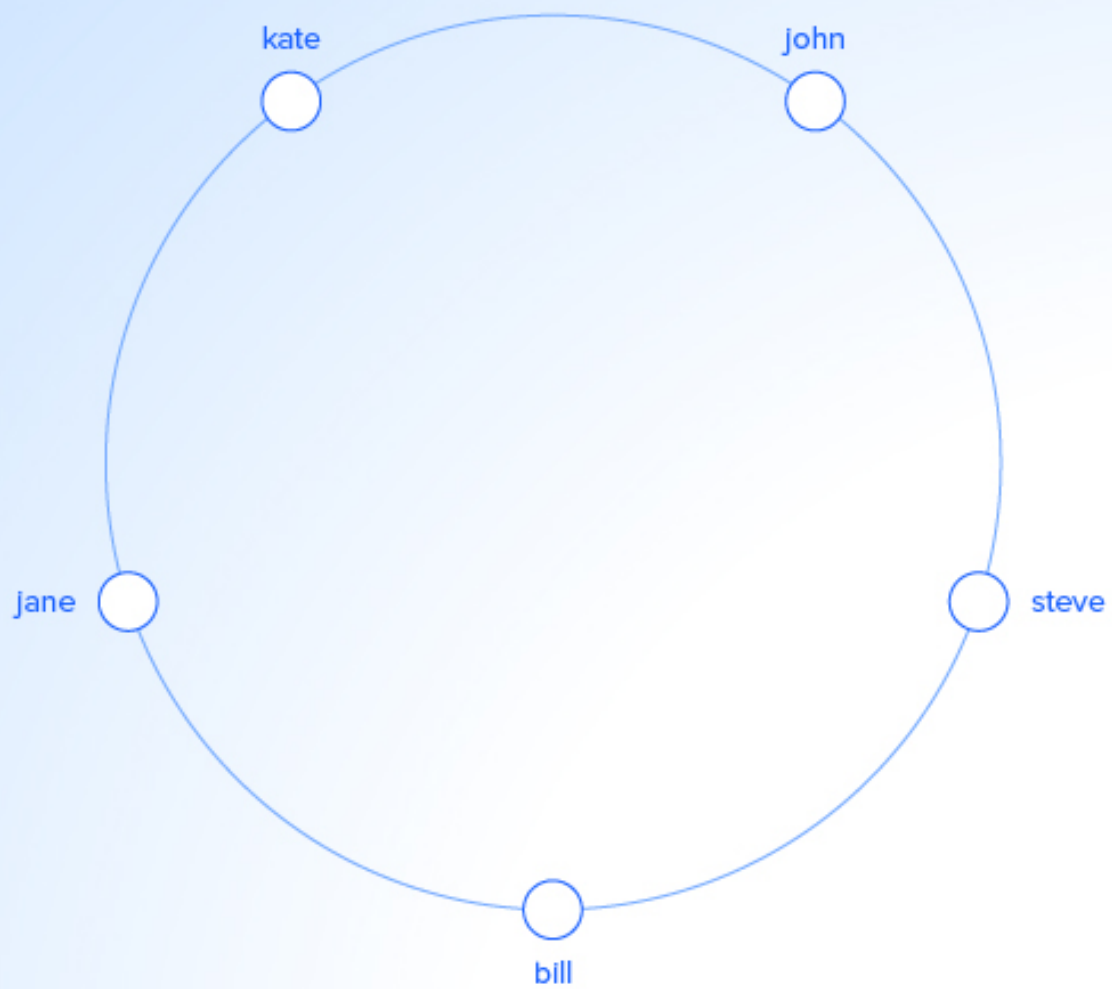
## The Solution: Consistent Hashing

So, how can this problem be solved? We need a distribution scheme that does *not* depend directly on the number of servers, so that, when adding or removing servers, the number of keys that need to be relocated is minimized. One such scheme—a clever, yet surprisingly simple one—is called *consistent hashing*, and was first described by [Karger et al. at MIT](#) in an academic paper from 1997 (according to Wikipedia).

Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed *hash table* by assigning them a position on an abstract circle, or *hash ring*. This allows servers and objects to scale without affecting the overall system.

Imagine we mapped the hash output range onto the edge of a circle. That means that the minimum possible hash value, zero, would correspond to an angle of zero, the maximum possible value (some big integer we'll call `INT_MAX`) would correspond to an angle of  $2\pi$  radians (or 360 degrees), and all other hash values would linearly fit somewhere in between. So, we could take a key, compute its hash, and find out where it lies on the circle's edge. Assuming an `INT_MAX` of  $10^{10}$  (for example's sake), the keys from our previous example would look like this:



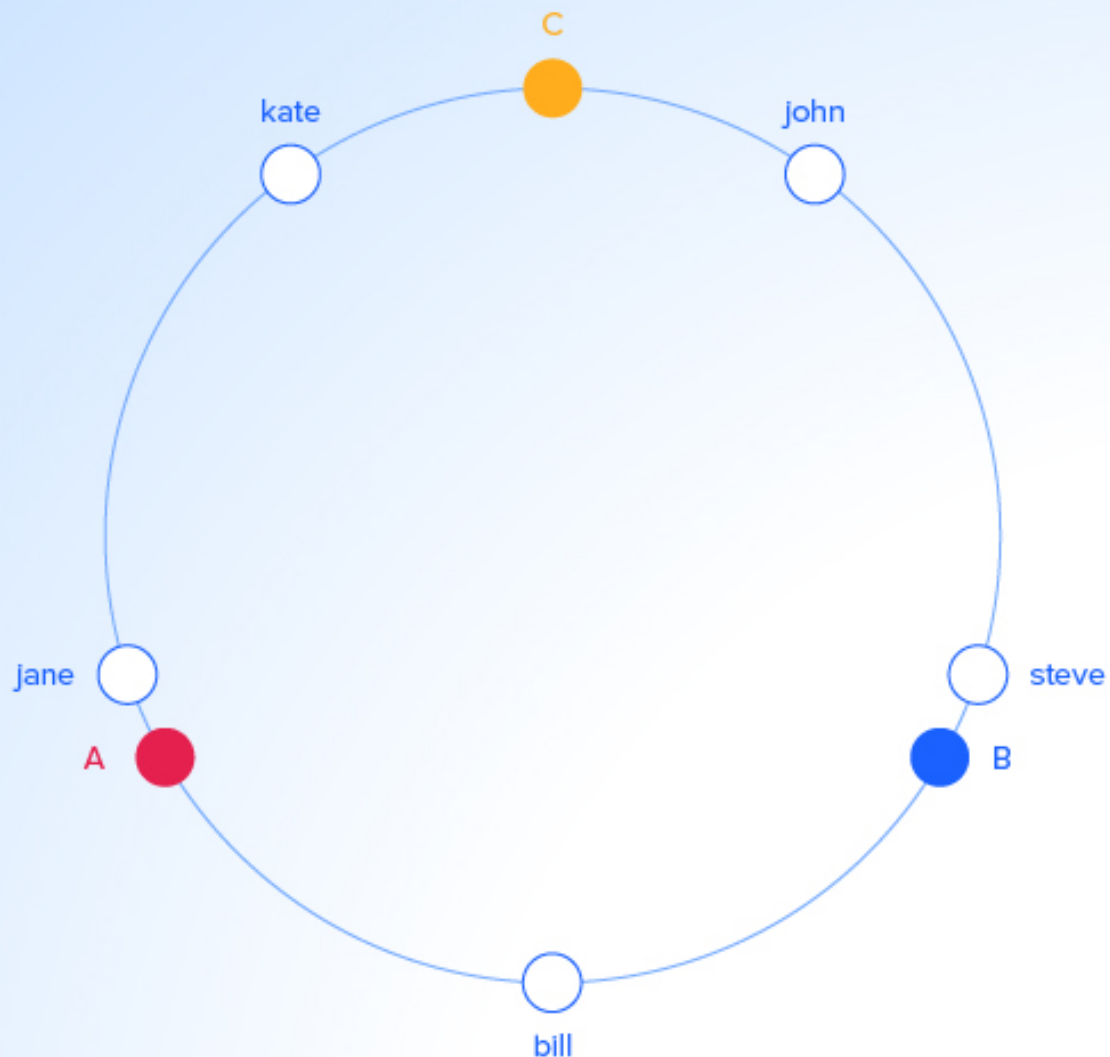


KEY	HASH	ANGLE (DEG)
"john"	1633428562	58.8
"bill"	7594634739	273.4
"jane"	5000799124	180
"steve"	9787173343	352.3

"kate"    3421657995    123.2

Now imagine we also placed the servers on the edge of the circle, by pseudo-randomly assigning them angles too. This should be done in a repeatable way (or at least in such a way that all clients agree on the servers' angles). A convenient way of doing this is by hashing the server name (or IP address, or some ID)—as we'd do with any other key—to come up with its angle.

In our example, things might look like this:

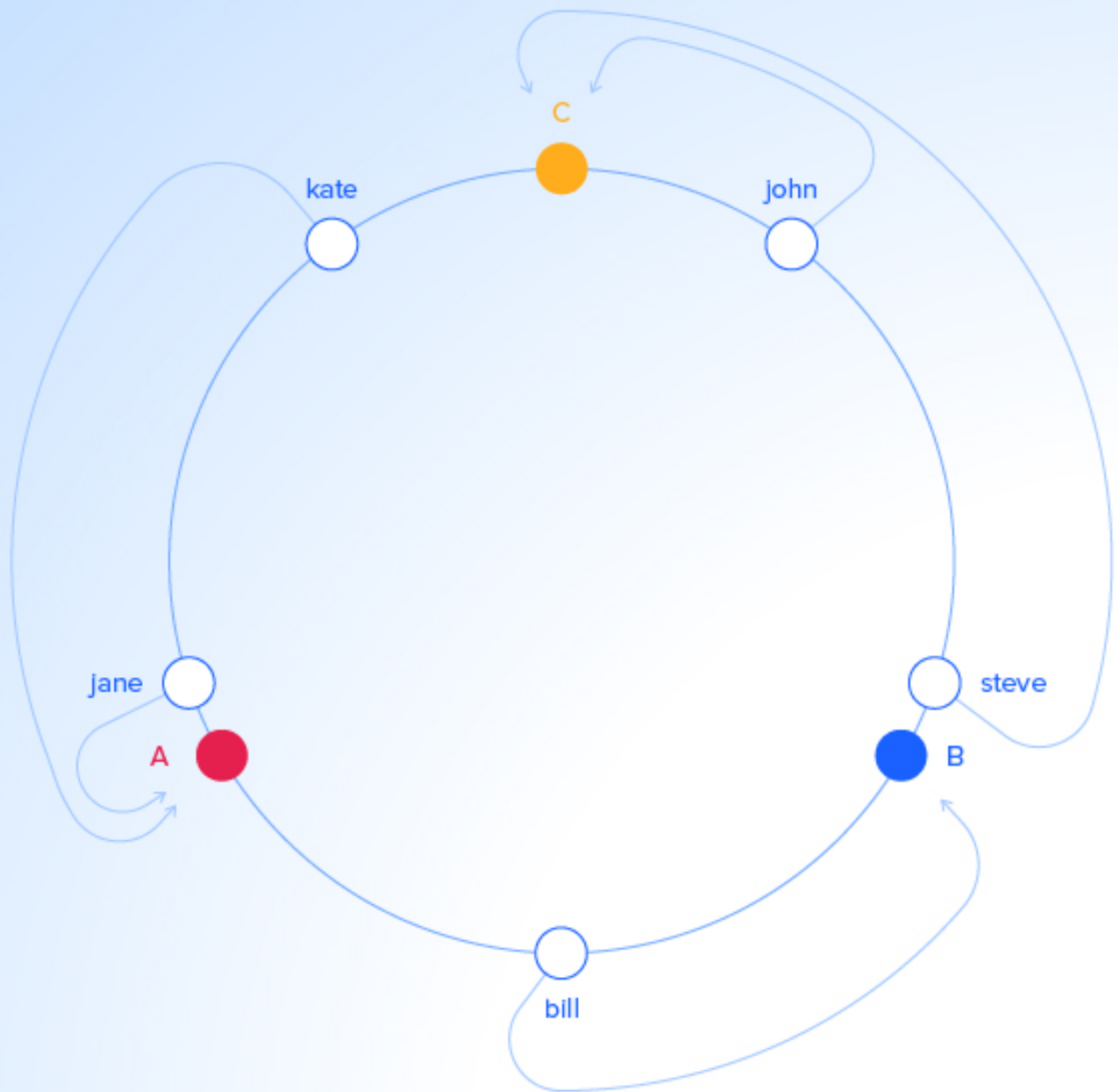


KEY	HASH	ANGLE (DEG)
-----	------	-------------

"john"	1633428562	58.8
"bill"	7594634739	273.4
"jane"	5000799124	180
"steve"	9787173343	352.3
"kate"	3421657995	123.2
"A"	5572014558	200.6
"B"	8077113362	290.8
"C"	2269549488	81.7

Since we have the keys for both the objects and the servers on the same circle, we may define a simple rule to associate the former with the latter: Each object key will belong in the server whose key is closest, in a counterclockwise direction (or clockwise, depending on the conventions used). In other words, to find out which server to ask for a given key, we need to locate the key on the circle and move in the ascending angle direction until we find a server.

In our example:



KEY	HASH	ANGLE (DEG)
"john"	1633428562	58.7
"C"	2269549488	81.7
"kate"	3421657995	123.1
"jane"	5000799124	180

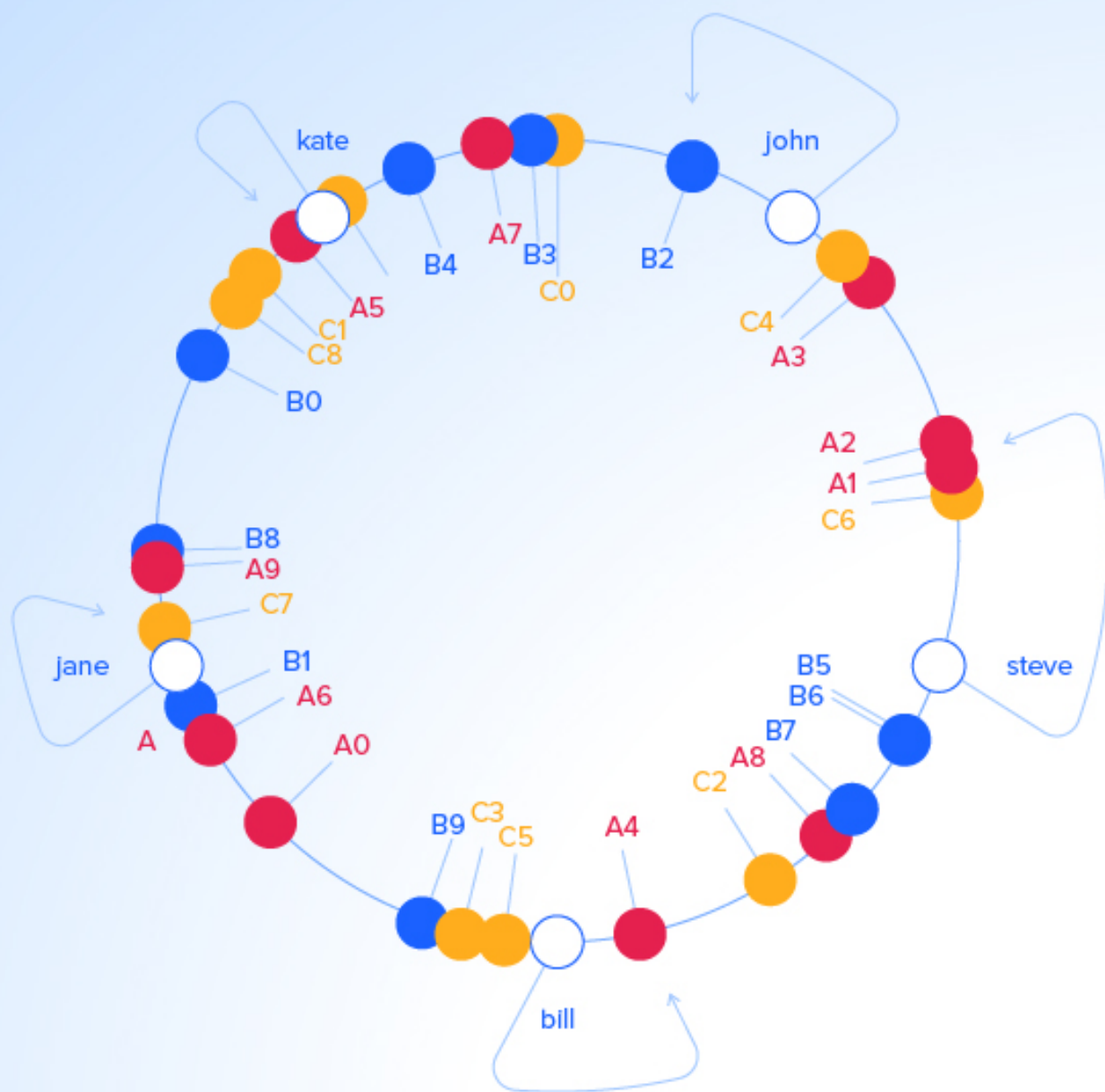
"A"	5572014557	200.5
"bill"	7594634739	273.4
"B"	8077113361	290.7
"steve"	787173343	352.3

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"C"	C
"kate"	3421831276	123.1	"A"	A
"jane"	5000648311	180	"A"	A
"bill"	7594873884	273.4	"B"	B
"steve"	9786437450	352.3	"C"	C

From a programming perspective, what we would do is keep a sorted list of server values (which could be angles or numbers in any real interval), and walk this list (or use a binary search) to find the first server with a value greater than, or equal to, that of the desired key. If no such value is found, we need to wrap around, taking the first one from the list.

To ensure object keys are evenly distributed among servers, we need to apply a simple trick: To assign not one, but many labels (angles) to each server. So instead of having labels A, B and C, we could have, say, A0 .. A9, B0 .. B9 and C0 .. C9, all interspersed along the circle. The factor by which to increase the number of labels (server keys), known as *weight*, depends on the situation (and may even be different for each server) to adjust the probability of keys ending up on each. For example, if server B were twice as powerful as the rest, it could be assigned twice as many labels, and as a result, it would end up holding twice as many objects (on average).

For our example we'll assume all three servers have an equal weight of 10 (this works well for three servers, for 10 to 50 servers, a weight in the range 100 to 500 would work better, and bigger pools may need even higher weights):



KEY	HASH	ANGLE (DEG)
"C6"	408965526	14.7
"A1"	473914830	17
"A2"	548798874	19.7
"A3"	1466730567	52.8

"C4"	1493080938	53.7
"john"	1633428562	58.7
"B2"	1808009038	65
"C0"	1982701318	71.3
"B3"	2058758486	74.1
"A7"	2162578920	77.8
"B4"	2660265921	95.7
"C9"	3359725419	120.9
"kate"	3421657995	123.1
"A5"	3434972143	123.6
"C1"	3672205973	132.1
"C8"	3750588567	135
"B0"	4049028775	145.7
"B8"	4755525684	171.1
"A9"	4769549830	171.7
"jane"	5000799124	180
"C7"	5014097839	180.5
"B1"	5444659173	196
"A6"	6210502707	223.5
"A0"	6511384141	234.4
"B9"	7292819872	262.5
"C3"	7330467663	263.8
"C5"	7502566333	270
"bill"	7594634739	273.4

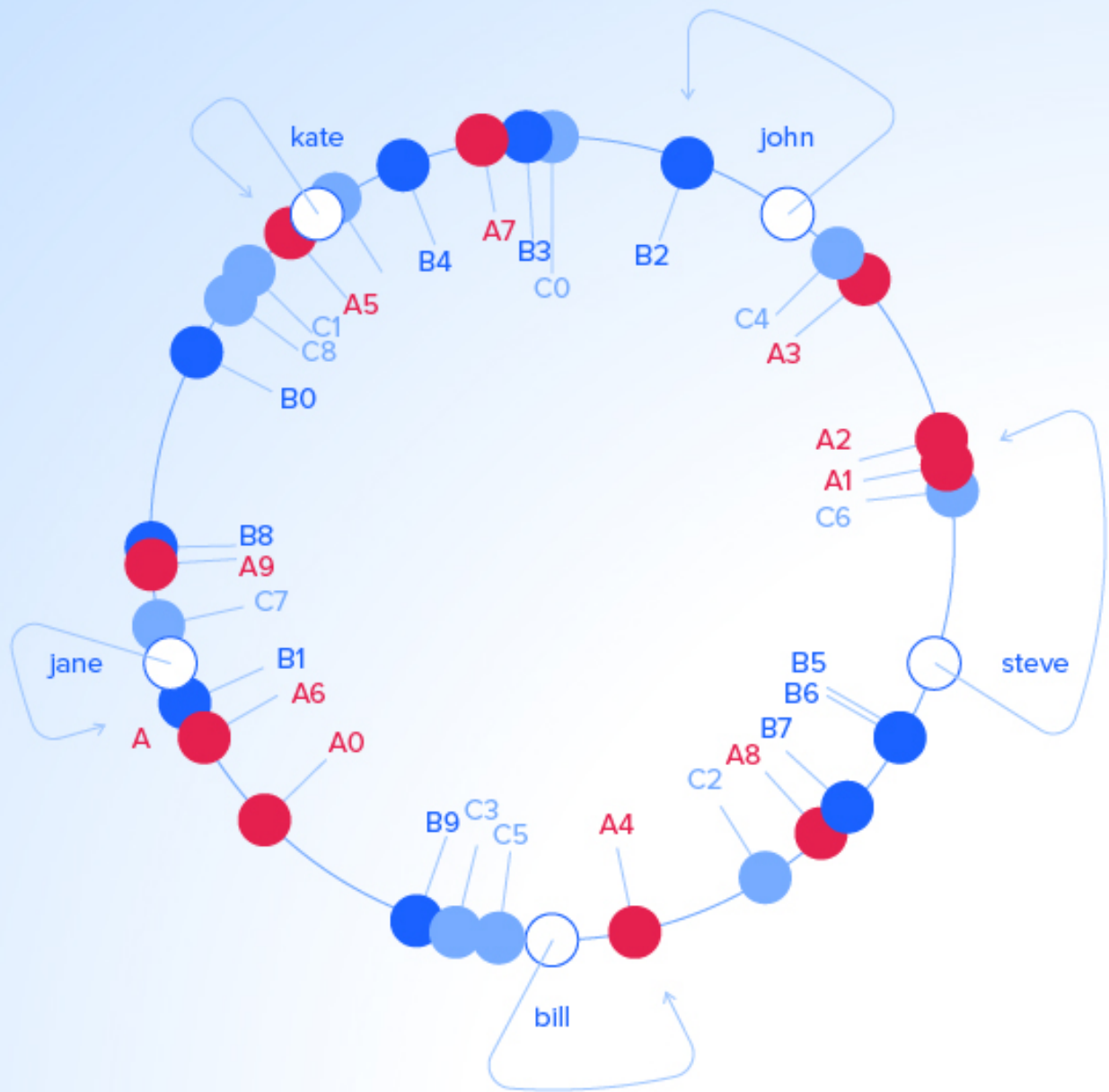
"A4"	8047401090	289.7
"C2"	8605012288	309.7
"A8"	8997397092	323.9
"B7"	9038880553	325.3
"B5"	9368225254	337.2
"B6"	9379713761	337.6
"steve"	9787173343	352.3

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"B2"	B
"kate"	3421831276	123.1	"A5"	A
"jane"	5000648311	180	"C7"	C
"bill"	7594873884	273.4	"A4"	A
"steve"	9786437450	352.3	"C6"	C

So, what's the benefit of all this circle approach? Imagine server c is removed. To account for this, we must remove labels c0 . . c9 from the circle. This results in the object keys formerly adjacent to the deleted labels now being randomly labeled Ax and Bx, reassigning them to servers A and B.

But what happens with the other object keys, the ones that originally belonged in A and B? Nothing! That's the beauty of it: The absence of cx labels does not affect those keys in any way. So, removing a server results in its object keys being randomly reassigned to the rest of the servers, **leaving all other keys untouched**:





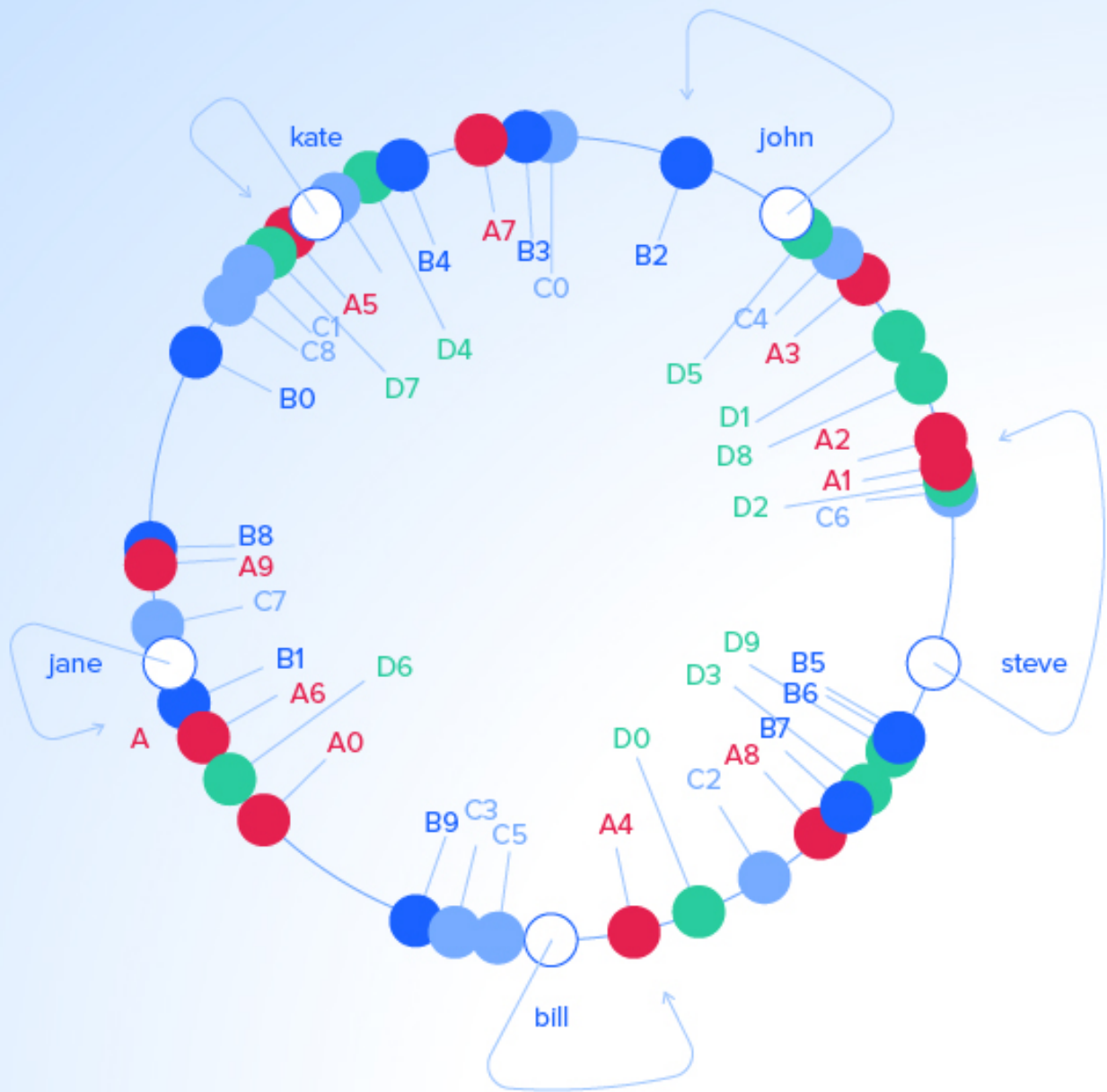
KEY	HASH	ANGLE (DEG)
"A1"	473914830	17
"A2"	548798874	19.7
"A3"	1466730567	52.8
"john"	1633428562	58.7

"B2"	1808009038	65
"B3"	2058758486	74.1
"A7"	2162578920	77.8
"B4"	2660265921	95.7
"kate"	3421657995	123.1
"A5"	3434972143	123.6
"B0"	4049028775	145.7
"B8"	4755525684	171.1
"A9"	4769549830	171.7
"jane"	5000799124	180
"B1"	5444659173	196
"A6"	6210502707	223.5
"A0"	6511384141	234.4
"B9"	7292819872	262.5
"bill"	7594634739	273.4
"A4"	8047401090	289.7
"A8"	8997397092	323.9
"B7"	9038880553	325.3
"B5"	9368225254	337.2
"B6"	9379713761	337.6
"steve"	9787173343	352.3

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"B2"	B

"kate"	3421831276	123.1	"A5"	A
"jane"	5000648311	180	"B1"	B
"bill"	7594873884	273.4	"A4"	A
"steve"	9786437450	352.3	"A1"	A

Something similar happens if, instead of removing a server, we add one. If we wanted to add server D to our example (say, as a replacement for C), we would need to add labels D0 .. D9. The result would be that roughly one-third of the existing keys (all belonging to A or B) would be reassigned to D, and, again, the rest would stay the same:



KEY	HASH	ANGLE (DEG)
"D2"	439890723	15.8
"A1"	473914830	17
"A2"	548798874	19.7
"D8"	796709216	28.6

"D1"	1008580939	36.3
"A3"	1466730567	52.8
"D5"	1587548309	57.1
"john"	1633428562	58.7
"B2"	1808009038	65
"B3"	2058758486	74.1
"A7"	2162578920	77.8
"B4"	2660265921	95.7
"D4"	2909395217	104.7
"kate"	3421657995	123.1
"A5"	3434972143	123.6
"D7"	3567129743	128.4
"B0"	4049028775	145.7
"B8"	4755525684	171.1
"A9"	4769549830	171.7
"jane"	5000799124	180
"B1"	5444659173	196
"D6"	5703092354	205.3
"A6"	6210502707	223.5
"A0"	6511384141	234.4
"B9"	7292819872	262.5
"bill"	7594634739	273.4
"A4"	8047401090	289.7
"D0"	8272587142	297.8

"A8"	8997397092	323.9
"B7"	9038880553	325.3
"D3"	9048608874	325.7
"D9"	9314459653	335.3
"B5"	9368225254	337.2
"B6"	9379713761	337.6
"steve"	9787173343	352.3

KEY	HASH	ANGLE (DEG)	LABEL	SERVER
"john"	1632929716	58.7	"B2"	B
"kate"	3421831276	123.1	"A5"	A
"jane"	5000648311	180	"B1"	B
"bill"	7594873884	273.4	"A4"	A
"steve"	9786437450	352.3	"D2"	D

This is how consistent hashing solves the rehashing problem.

In general, only  $k/N$  keys need to be remapped when  $k$  is the number of keys and  $N$  is the number of servers (more specifically, the maximum of the initial and final number of servers).

## What Next?

We observed that when using distributed caching to optimize performance, it may happen that the number of caching servers changes (reasons for this may be a server crashing, or the need to add or remove a server to increase or decrease overall capacity). By using consistent hashing to distribute keys between the servers, we can rest assured that should that happen, the number of keys being rehashed—and therefore, the impact on origin servers—will be minimized, preventing potential downtime or performance issues.

There are clients for several systems, such as Memcached and Redis, that include support for consistent hashing out of the box.

Alternatively, you can implement the algorithm yourself, in your language of choice, and that should be relatively easy once the concept is understood.

If data science interests you, Toptal has some of the best articles on the subject at the [blog](#)

## About the author



[View full profile »](#)

[Hire the Author](#)

[Juan Pablo Carzolio, Argentina](#)

member since August 31, 2015

[CSSJavaScriptPHP](#)

Juan is a versatile and dependable full-stack engineer with ten years of professional experience and a computer science degree. He taught himself programming twenty years ago at the age of eleven. With a strong math and computer science background, he loves to apply reasoning and creativity to understand and solve challenging problems, regardless of the technologies used. He is proficient in several languages, including JavaScript and PHP. [\[click to continue...\]](#)


[Hiring? Meet the Top 10 Freelance Big Data Architects for Hire in February 2018](#)

16 Comments

Toptal

 1 Zijing Guo ▾

 Recommend 10

 Share

Sort by Best ▾

Toptal requires you to verify your email address before posting. Send verification email to [altergzj@yahoo.cn](mailto:altergzj@yahoo.cn) 



. . . . .



Join the discussion...



**lol** • 4 months ago

If a server crashes how do you redistribute keys from that server as the server is now offline?

1 ^ | v • Reply • Share ›



**keyser soze** ➔ lol • 2 months ago

There must be some sort of backup server for that one so when it goes down we can use that one to recover keys.

^ | v • Reply • Share ›



**Nitin Puranik** ➔ keyser soze • 2 months ago

No, there's no need for a backup server. Lets say key 'A' was in the caching server 'S1' and the server S1 now crashed. With consistent hashing in place, when an external agent runs the hashing function on key 'A', it now maps to, say, caching server 'S2' since S1 is gone. We then go and query for key 'A' in S2. Since S2 does not have that key, we run into a cache miss. This results in reaching out to the database to retrieve the value for key 'A'. This key/value pair is now stored in server S2 and also returned to the external requesting client.

This is how the key A 'moved' from server S1 to S2, through a cache miss.

1 ^ | v • Reply • Share ›



**Daniel Crabtree** • 9 months ago

Jump Consistent Hashing is a better algorithm for consistent hashing than Karger et al. See this research paper by Lamping & Veach: <https://arxiv.org/abs/1406....>

1 ^ | v • Reply • Share ›



**Arthur Okeke** • 9 months ago

There is a difference between Hashing and PreHashing. Suppose there is an object that you want to hash and there is a function that does an object-to-integer conversion, that is the prehashing step. The hashing is actually done when you want to put the integer into a hash bucket.

So for instance, Java strings are normally hashed mod 31 using horners algorithm. From your tutorial, you are calling this process hashing. Though it is a common conception, it is not entirely correct. The right term for it is prehashing.



Hashing is what happens before you assign the key to a bucket.

There are many other suggestions in your article but overall nice stuff.

1 ^ | v • Reply • Share ›



**Juan Pablo Carzolio** ➔ Arthur Okeke • 9 months ago

Arthur, thanks for your comment! I understand your point, but I haven't seen such naming distinction used before, and googling it I couldn't find any references. Can you provide a reference? Thanks!

^ | v • Reply • Share ›



**Arthur Okeke** ➔ Juan Pablo Carzolio • 9 months ago

Prof. Eric Demain

Professor of Computer Science MIT in his Hashing Course.

^ | v • Reply • Share ›



**Tomer Ben David** • 4 months ago

aha so calculation from hash to degrees  $(1633428562 / 10^{10}) * 360$

^ | v • Reply • Share ›



**Neeraj Murarka** • 8 months ago

Excellent article, btw, Juan!

^ | v • Reply • Share ›



**Juan Pablo Carzolio** ➔ Neeraj Murarka • 6 months ago

Thanks for your comments, Neeraj!

^ | v • Reply • Share ›



**Neeraj Murarka** • 8 months ago

...move in the ascending angle direction until we find a server.

I think it is descending, right?

^ | v • Reply • Share ›



**Juan Pablo Carzolio** ➔ Neeraj Murarka • 6 months ago

Here it's ascending (i.e. counter-clockwise under common math conventions), but it's just a convention. Either direction works, as long as it's used consistently.

1 ^ | v • Reply • Share ›



**Neeraj Murarka** • 8 months ago

...an average complexity of  $O(N/k)$ , where  $k$  is the number of buckets).

Is this right? I think that here,  $N$  is being used as the population of objects. Then, the number of items per bucket is  $N/k$  and the search complexity per bucket is  $O(N/k)$ , but if the number of buckets increases as  $N$  goes up (population increases) linearly, then  $k = pN$ , and so the complexity is  $O(N/pN) = O(1)$  (constant)

^ | v • Reply • Share ›



**Juan Pablo Carzolio** ➔ Neeraj Murarka • 6 months ago

Exactly. As long as  $k$  is kept roughly proportional to  $N$ , access takes \*almost\* constant time (I say almost because  $k$  does not usually change immediately as  $N$  changes). So, for any given value of  $k$ , access is linear (albeit with a very low factor,  $1/k$ ), but in practice  $k$  is kept roughly proportional to  $N$ , so  $N/k$  stays almost constant.

^ | v • Reply • Share ›



**Julian** • 9 months ago

although slightly off topic, if we are using posts/products and we assign a number to a product, it is a good idea to generate auto increment numbers or generate a random number to assign it to that product ? like :[www.amazon.com/product/1/tiles](http://www.amazon.com/product/1/tiles)

instead of 1 is it clever to use a perfect random number for this just that the user cannot type in 2 and go to the next product?

^ | v • Reply • Share ›



**Juan Pablo Carzolio** ➔ Julian • 9 months ago

Thanks for your comment, Julian. That's not directly related to the article, but it's an interesting question nonetheless. I think both approaches have pros and cons. The serial approach is simpler, widely supported, and usually results in relatively small/readable numbers. The random approach, as you noted, prevents "exploration", and doesn't require a centralized issuer, enabling distributed and concurrent generation. Which one is better depends on the situation.

1 ^ | v • Reply • Share ›

Get the latest content first.

Enter your email address...

Get Exclusive Updates

No spam. Just great engineering posts.

The #1 Blog for Engineers

Get the latest content first.

Thank you for subscribing!

Check your inbox to confirm subscription. You'll start receiving posts after you confirm.

111shares



•



•



•

Trending articles



[Blockchain, IoT, and the Future of Transportation: Understanding the Motoro](#)



[Coin](#)about 1 hour ago

[Mixed-integer Programming: A Guide to Computational](#)



[Decision-making](#)8 days ago

[Exploring Supervised Machine Learning Algorithms](#)10



[days ago](#)

[Command Line Tools for Developers](#)15 days ago

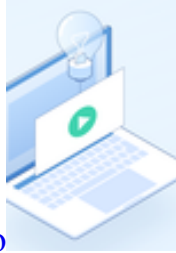


[REST Assured](#)



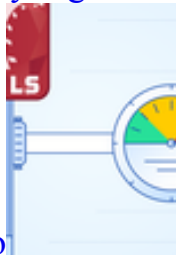
[vs. JMeter: A Comparison of REST Test Tools](#)20 days ago

[Salesforce Einstein AI: An](#)



[API Tutorial](#)21 days ago

[Asynchronous JavaScript: From Callback Hell to Async and](#)



[Await](#)28 days ago

[Field-level Rails Cache Invalidation: A DSL Solution](#)28 days ago

Relevant Technologies

- [Big Data](#)
- [Algorithm](#)
- [Big Data](#)

About the author



## [Juan Pablo Carzolio](#)

### PHP Developer

Juan is a versatile and dependable full-stack engineer with ten years of professional experience and a computer science degree. He taught himself programming twenty years ago at the age of eleven. With a strong math and computer science background, he loves to apply reasoning and creativity to understand and solve challenging problems, regardless of the technologies used. He is proficient in several languages, including JavaScript and PHP.

## [Hire the Author](#)

Toptal connects the [top 3%](#) of freelance talent all over the world.

## Toptal Developers

- [Android Developers](#)
- [AngularJS Developers](#)
- [Back-End Developers](#)
- [C++ Developers](#)
- [Data Scientists](#)
- [DevOps Engineers](#)
- [Ember.js Developers](#)
- [Freelance Developers](#)
- [Front-End Developers](#)
- [Full Stack Developers](#)
- [HTML5 Developers](#)
- [iOS Developers](#)
- [Java Developers](#)
- [JavaScript Developers](#)
- [Machine Learning Engineers](#)
- [Magento Developers](#)
- [Mobile App Developers](#)
- [.NET Developers](#)
- [Node.js Developers](#)
- [PHP Developers](#)
- [Python Developers](#)
- [React.js Developers](#)

- [Ruby Developers](#)
- [Ruby on Rails Developers](#)
- [Salesforce Developers](#)
- [Scala Developers](#)
- [Software Developers](#)
- [Unity or Unity3D Developers](#)
- [Web Developers](#)
- [WordPress Developers](#)

[See more freelance developers](#)

[Learn how enterprises benefit from Toptal experts.](#)

## Join the Toptal community.

[Hire a developer](#)

or

[Apply as a Developer](#)

## Highest In-Demand Talent

- [iOS Developer](#)
- [Front-End Developer](#)
- [UX Designer](#)
- [UI Designer](#)
- [Financial Modeling Consultants](#)
- [Interim CFOs](#)

## About

- [Top 3%](#)
- [Clients](#)
- [Freelance Developers](#)
- [Freelance Designers](#)
- [Freelance Finance Experts](#)
- [About Us](#)

## Contact

- [Contact Us](#)
- [Press Center](#)
- [Careers](#)
- [FAQ](#)

## Social

- [Facebook](#)
- [Twitter](#)
- [Google+](#)
- [LinkedIn](#)

## [Toptal](#)

Hire the top 3% of freelance talent

- © Copyright 2010 - 2018 Toptal, LLC
- [Privacy Policy](#)
- [Website Terms](#)

[Home](#) › [Blog](#) › [A Guide to Consistent Hashing](#)

**Hiring?** Toptal handpicks [top Big Data architects](#) to suit your needs.

- [Start hiring](#)
- [Log in](#)