NEW TECH FORUM
By Kyle J. Davis, InfoWorld
SEP 20, 2017 3:00 AM PDT

About
Emerging tech dissected by technologists

Manage access control using Redis Bitfields

How to create a high-performance, highly available, and flexible access control system using binary data and bitwise operators in Redis

Kyle Davis is the technical marketing manager at Redis Labs.

One of the hardest parts about writing a user-facing app or service is controlling access to resources. Decisions about access control are some of the earliest to be made and can make or break an entire platform. It's usually a trade-off between granularity and speed. Let's explore how to leverage Redis to get granular control and speed at the same time.

One approach is to set up "user levels," typically numbers or roles such as "admin," "regular user," "privileged user," etc. This approach alone is usually not a very viable path as you run into a never-ending additive process ("super-super-admin" or "disabled-regular-user," etc.) or create a mess of widely spaced user levels and hope for the best.

The other approach is to enable specific actions (e.g. edit, view, update, etc.) to be performed on a user-by-user level. For our purposes, we'll call it a "capability," but you can think of it as something similar to a GRANT in SQL or simply a domain-specific, action-based permission. Access control based on action is a flexible, granular approach to securing your resources. Each user is given a list of things they can do and when the user attempts to perform any action, you check the user's capabilities against what is required of that action. Sounds simple enough, right?

[Also on InfoWorld: How to choose the right NoSQL database]

This can be a tricky thing to code and it has to be as fast as possible because whatever latency, transit, or computation time this step requires is overhead that cuts into the processing you need to do with the rest of your app (likely stuff you care more about than capabilities and privileges). First, let's look at a highly efficient way of storing capabilities and later we'll explore some more advanced functionality.

The heart of this approach is to use binary data, which might seem strange. Redis, unlike many databases, can manipulate and store binary data directly. We can leverage this feature to flip individual bits in a bitmap to represent capabilities. Each capability represents a bit at an index. To have access to a page or route in your application, the user must have the bits set by the page or route.

We'll store each user's capabilities in a specific key using `SETBIT`— something like this:

```
> SETBIT user:kyle 0 1 > SETBIT user:kyle 3 1 > SETBIT user:kyle 4 1
```

In the above example I'm setting bits 0, 3, and 4 to 1 in the key `user:kyle`. If a bit isn't yet set, Redis assumes it is a 0.

Each route would have a similar bitmap for its requirements:

```
> SETBIT route:/test/:thing 0 1 > SETBIT route:/test/:thing 4 1
```

The technique relies on bitwise operators with the Redis `BITOP` command. Bitwise operators may seem intimidating at first, but just remember that they are similar to boolean operators. However, whereas boolean operators apply to a single bit (usually represented as true or false), bitwise operators compare each value on a bit-by-bit basis. The 0th bit of value 1 is compared to 0th bit of value 2 and that returns the 0th bit of the result. We do this for every bit in the two values. We'll need to use two operators for this technique: bitwise XOR and bitwise AND.

Let's explore how this works using a few examples.

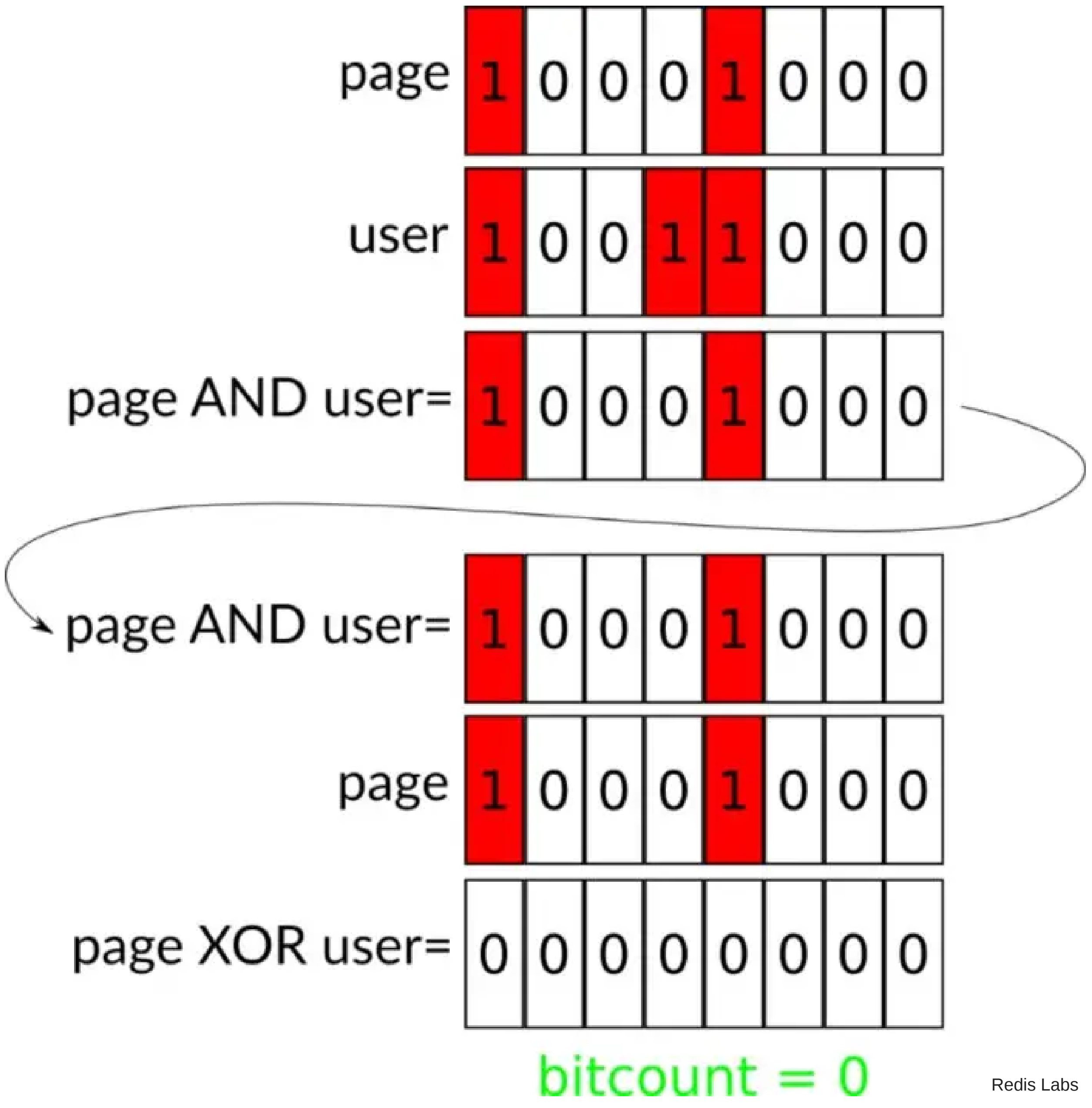
User has more capabilities than required

In this case, the user has all of the capabilities required to access the page plus some. In the first stage (bitwise AND) you can see that we're effectively finding the intersection of the page and user bits. We could stop here and just compare the page with the result of the AND:

```
if (page === (page AND user)) { ... }
```

However, this would involve transporting the bits back to the app level rather than keeping them in Redis. It's not a terrible thing to do, but there are complications. Redis is fine with binary data, but it can be tricky with some languages (for example JavaScript, which wants to do type coercion; this is why we have `return_buffers` in node_redis). In addition, transporting the bits means extra overhead and complexity for your app. Luckily, we can do all this in Redis with the next step.

In this step we'll bitwise XOR the results of the first step with the page bitmask. In this operation, we'll return all zeros if they match. Finally, we can leverage `BITCOUNT` to see how many ones are set in the second step.



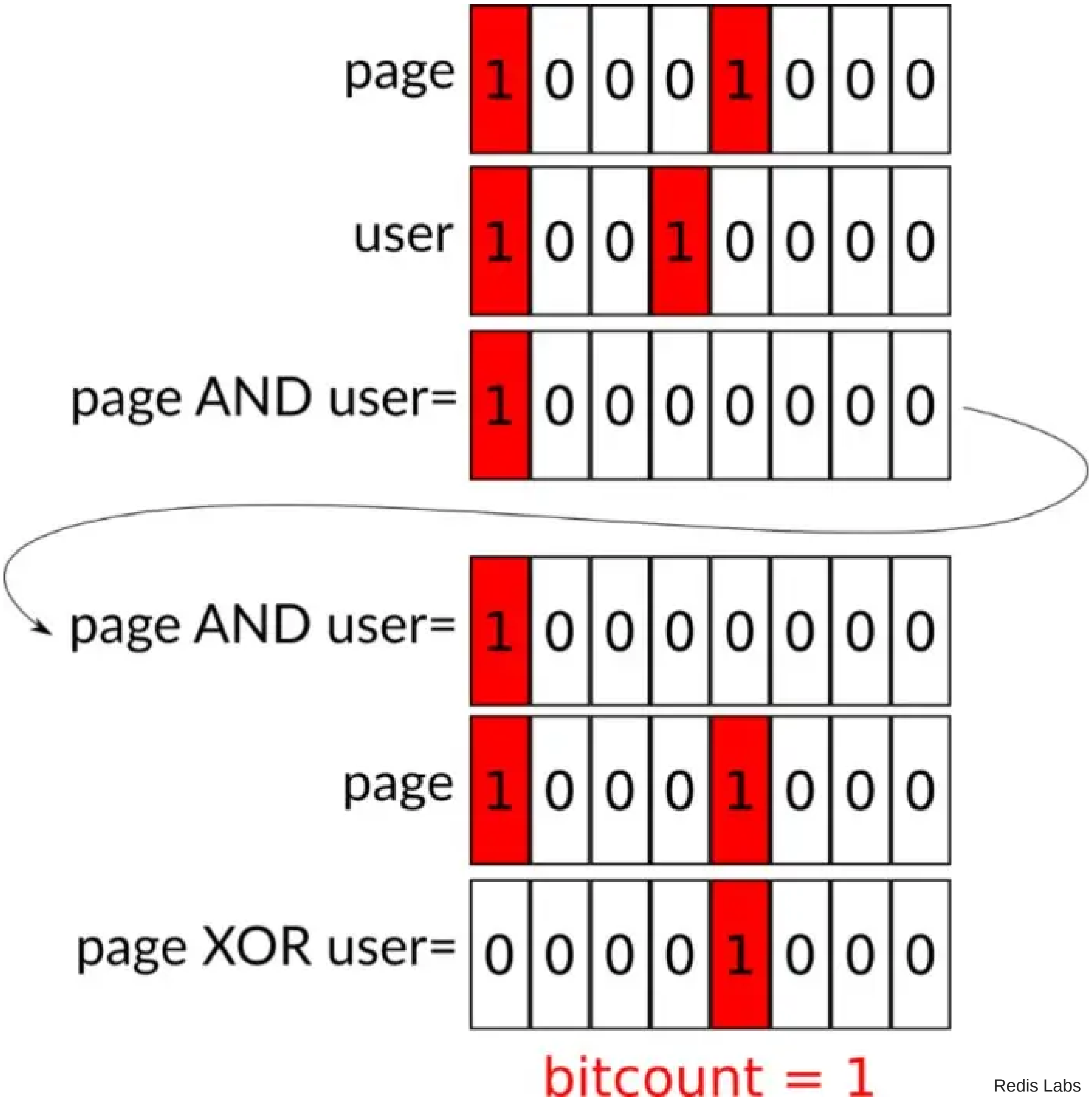
Redis Labs

`BITCOUNT = 0` means that the user has the required capabilities.

If the third step returns 0 to the app level then access would be granted to perform an action. Now, let's take a look at another situation.

User has some of the capabilities but not all

In this case the user has only some of the capabilities required to view the page. In the first stage, ANDing the bits together yields only the common bits between page and user. In the second stage we see that XOR will find the bits that exist only in one. When we `BITCOUNT` it, we'll see that it's greater than zero which indicates that the user is not let through.

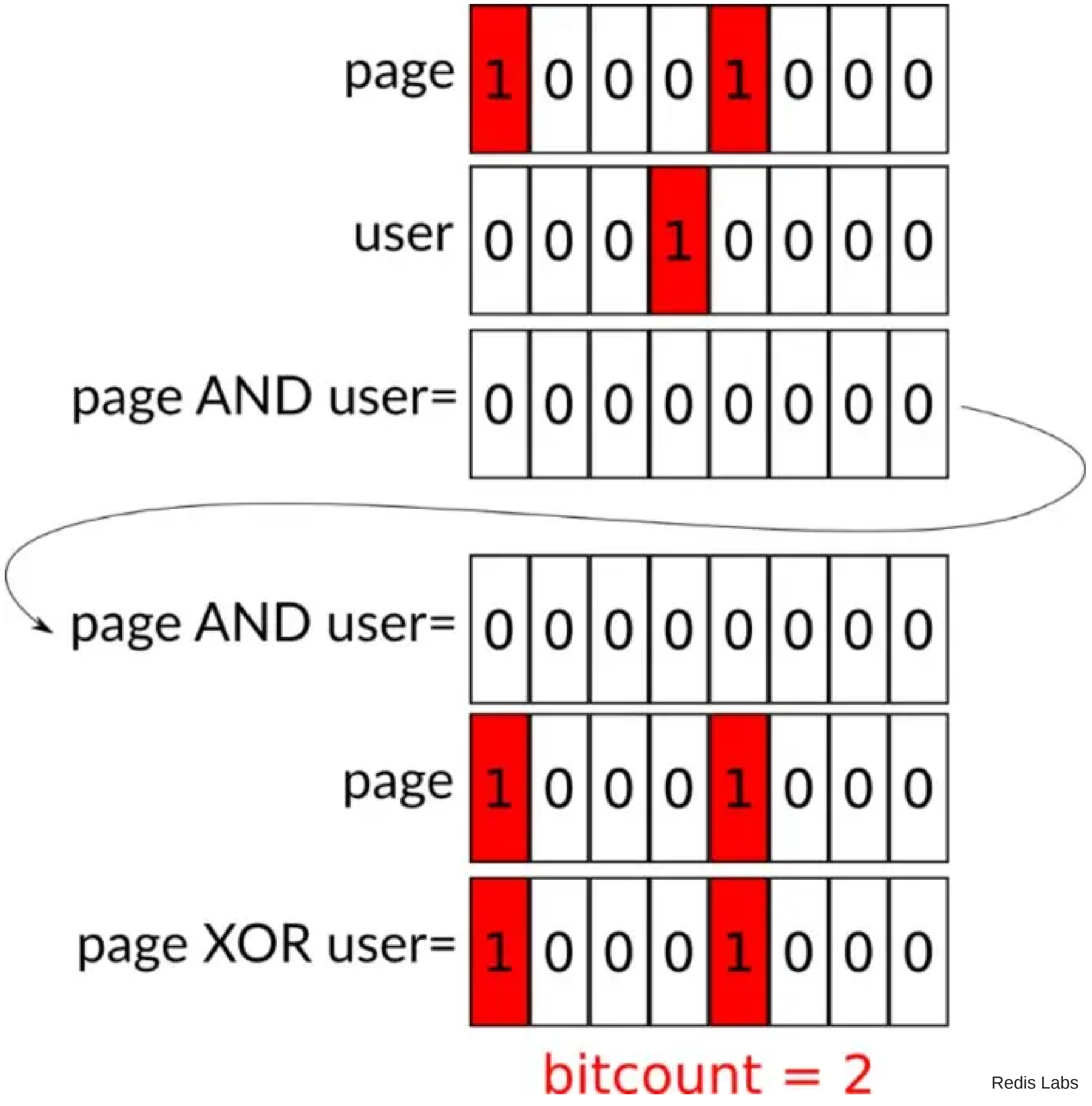


Redis Labs

`BITCOUNT > 1` means the user doesn't have the right capabilities.

User has none of the capabilities needed

When the user has some capabilities but not the ones needed, the ANDing will result in all 0 bits. During the XOR stage, the final result will be the same as the page. Finally, `BITCOUNT`ing the end result will yield the same number of bits as the page (e.g. greater than zero) so the user would not be let in.

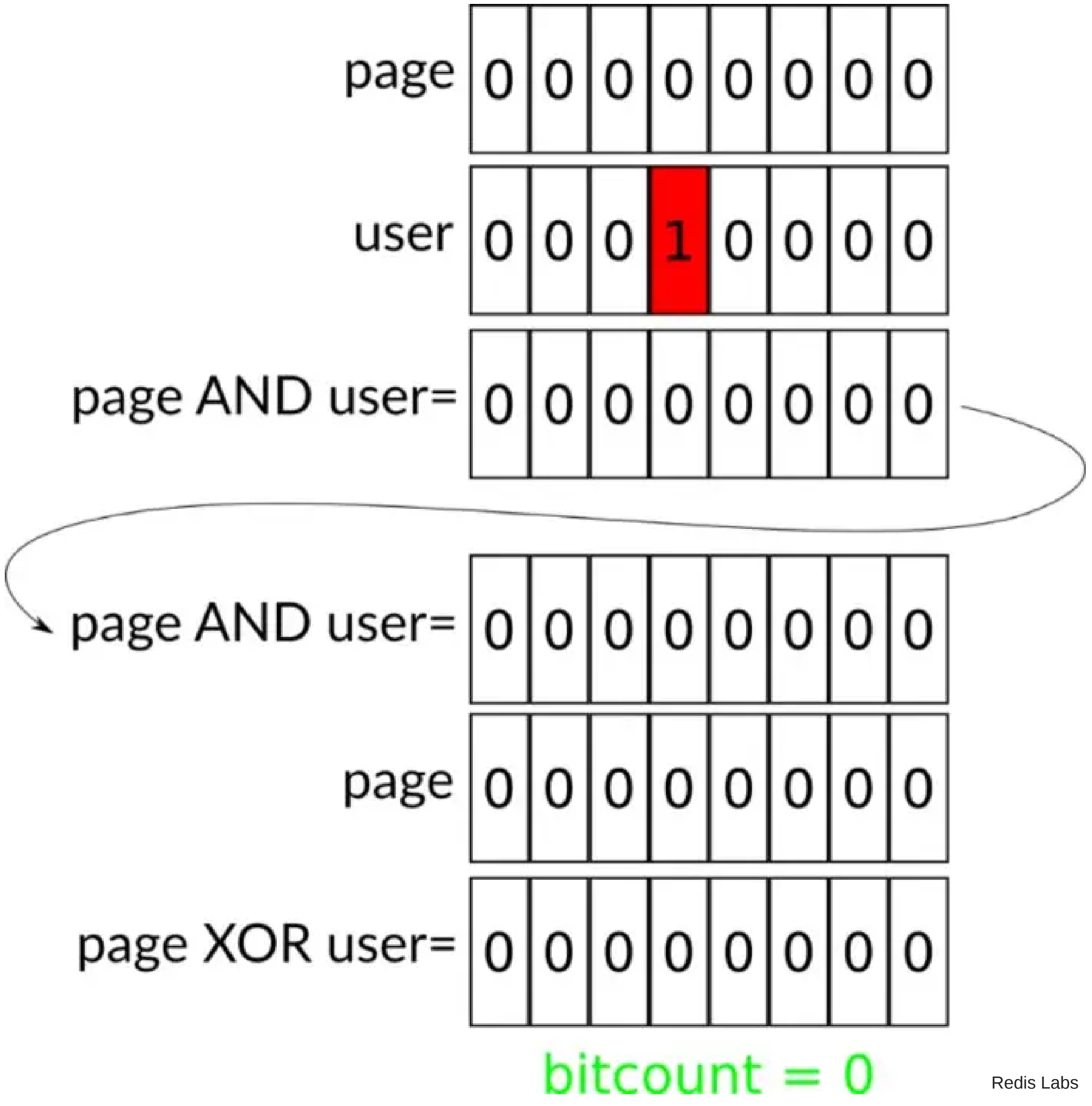


Redis Labs

`BITCOUNT > 1` means the user doesn't have the right capabilities.

Page has no requirements

In this case, the page is publicly viewable and requires nothing, but the user has some capabilities (that aren't actually needed). The first stage will result in all 0 bits. Pulling that down into the second stage and XORing the same thing will, of course, result in all zero bits. `BITCOUNT`ing this will result in 0, which means the user can be let in.



Redis Labs

`BITCOUNT = 0` means that the user has the required capabilities.

Of course, if the user has no capabilities and the page requires none, the truth tables will be nearly identical to this case and will grant the user access.

Implementing this type of logic in Redis requires just a few commands:

```
> MULTI
OK
> BITOP AND cap-temp user:kyle route:/test/:thing
QUEUED
> BITOP XOR cap-temp route:/test/:thing cap-temp
QUEUED
```

```
> BITCOUNT cap-temp
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
3) (integer) 0
```

The only truly relevant result is the third, from `BITCOUNT`. In the above case, we have a result of 0. That's great—that means that there are no conflicting bits between the page and the masked combination of the user and the page. If the result of the `BITCOUNT` was greater than zero, we would know that something doesn't match and we wouldn't let the user access the resource.

Beyond individual bits

Once you've established this binary key for your user capabilities, you aren't limited to just individually storing bits—remember, this is binary: the ultimate free-form.

Redis has a command called `BITFIELD`. This command is similar to `GETBIT` and `SETBIT` as it can manipulate or retrieve data on a bit-by-bit basis, but also has the ability to do a little more. First, let's explore the command:

```
> BITFIELD a-key GET u8 0 SET u4 8 12
          ——- —-—- —-—- —
          |     |   |  | |   |  | |_ value
          |     |   |  | |   |  |___ bit index
          |     |   |  | |   |______ datatype (unsigned, 4-bit)
          |     |   |  | |__________ sub (all above are arguments)
          |     |   |  |____________ bit index
          |     |   |_______________ datatype (unsigned, 8-bit)
          |     |___________________ sub (next 3 are arguments)
          |_________________________ key
```

The above command shows two subcommands (`GET`, `SET`) but you can certainly do as many as you'd like. The datatypes can be either signed (`ix`) or unsigned (`ux`). Since this is binary, nothing is enforced (remember this point). That means, for example, you can set a `u8` to bit index 0 then get a `u4` at index 0 (this would get the first four bits of the `u8`).

Let's take a look at another example of using `BITFIELD`. This time we'll return the binary representation of a byte.

```
> DEL a-key
(integer) 1
```

First, we're going to delete the key we're working with. This is critical (for this example) because we're not working on a key level but rather working on a bit level inside a key and we might not be starting empty.

```
> bitfield a-key SET u8 0 127
1) (integer) 0
```

Now, we're setting the first byte (u8) to 127. Those who know binary numbers will see that it's the highest value of a 7-bit number.

```
> BITFIELD a-key GET u1 0 GET u1 1 GET u1 2 GET u1 3 GET u1 4 GET u1
5 GET u1 6 GET u1 7
1) (integer) 0
2) (integer) 1
3) (integer) 1
4) (integer) 1
5) (integer) 1
6) (integer) 1
7) (integer) 1
8) (integer) 1
```

This looks complicated, but it's actually accomplishing something quite simple. We're getting back the 1 or 0 (`u1`) of each bit in the byte we set previously. As expected, we're seeing one zero and seven ones.

```
> BITFIELD a-key GET u4 0 GET u4 4
1) (integer) 7
2) (integer) 15
```

With the above command we're getting two unsigned 4-bit values at index 0 and index 4. Harkening back to your COMPSCI 101 class and binary counting, we know why we're getting 7 (0111) and 15 (1111).

In the context of our capabilities system, we can leverage this technique in a couple of ways. Knowing now that we can store numeric values inside our bitfield, we can supply this back to our application as sometimes it's useful to think in terms of numbers rather than bits. Taking this from abstract to a little more concrete, look at this situation:

- User A is the overall site administrator.
- User B is a region administrator.
- User C is a section administrator.
- User D is a page administrator.

Let's say only section administrators and site administrators should be able to modify section configuration parameters.

In this example it would be fairly simple to flip bits if you had a small number of users, but what if rather than one page administrator there are 1,000 or 10,000 (or even 1 million)? It would, in this situation, be better to have user levels for this specific area. Anyone above a certain level would be allowed to modify the parameters. To accomplish this, we can store full numbers in the user's bitmap. Back to the example:

- User A: 127 (u7 Binary: 1111111)
- User B: 60 (u7 Binary: 0111100)
- User C: 40 (u7 Binary: 0101000)
- User D: 20 (u7 Binary: 0010100)

There is no real pattern in these bits, but this is a place where you can bring in a u7 unsigned word. What's cool about this is that it can co-exist with bit-based flag capabilities. This is how you would lay it out:

```
      |—byte—|—byte—|—byte—|
User  |........|XYYYYYYY|........|
```

The first and third bytes have something else in them (not important). The second byte is relevant to our access control. The 0th bit is the bit flag that indicates, say, "administrator," while bits 1 through 7 form the u7 word. Our run-of-the-mill bit-based capabilities run the same. So, the route is protected by a bitmap that has no capabilities in the area where the u7 lives:

```
      |—byte—|—byte—|—byte—|
page  |........|.0000000|........| (dots can be 1s or 0s)
```

By using our bitmask technique from above (AND then XOR) we're effectively ignoring the u7. Then we use BITFIELD separately to incorporate the u7 as our additional user level.

All together

First, setup the page/route, requiring the 0th and 8th bit to be set and putting in a u7 at bits 9 to 16 with a value of 0. Then we'll use BITFIELD to set page requirements as well, but at a different key and to the desired level (60):

```
> BITFIELD a-page SET u1 0 1 SET u1 8 1 SET u7 9 0
> BITFIELD a-page:level SET u7 9 60
```

Then we'll setup a user that can access it (User B), a user that can't due to its user level (User C), and finally one that meets the user level requirements but doesn't meet the capability bits (User D):

```
> BITFIELD user-b SET u1 0 1 SET u1 8 1 SET u7 9 60
> BITFIELD user-c SET u1 0 1 SET u1 8 1 SET u7 9 40
> BITFIELD user-d SET u1 8 1 SET u7 9 60
```

Now, let's evaluate these vs our resource.

User B

```
> MULTI
OK
> BITOP AND cap-temp a-page user-b
QUEUED
> BITOP XOR cap-temp a-page cap-temp
QUEUED
> BITCOUNT cap-temp
QUEUED
> BITFIELD a-page:level GET u7 9
QUEUED
> BITFIELD user-b GET u7 9
QUEUED
> EXEC
1) (integer) 2
2) (integer) 2
3) (integer) 0
4) 1) (integer) 60
5) 1) (integer) 60
```

The first few commands look similar to those we used in the first part. On the app level we'll check the following:

- Is the third result 0? If so continue; otherwise reject access
- Is the fifth result greater than that of the fourth? If so continue; otherwise reject access

As you can see, the third result (`BITCOUNT`) is zero (which is our "you're good" capability check). The fourth result is the required user-level of the page. And the fifth result is the user-level.

User C

```
> MULTI
OK
> BITOP AND cap-temp a-page user-c
```

```
QUEUED
> BITOP XOR cap-temp a-page cap-temp
QUEUED
> BITCOUNT cap-temp
QUEUED
> BITFIELD a-page:level GET u7 9
QUEUED
> BITFIELD user-c GET u7 9
QUEUED
> EXEC
1) (integer) 2
2) (integer) 2
3) (integer) 0
4) 1) (integer) 60
5) 1) (integer) 40
```

Evaluating the third result, we're fine since the result is 0. However the fourth (resource level required) is higher than the user level (fifth result), so the app would reject the request.

User D

```
> MULTI
OK
> BITOP AND cap-temp a-page user-d
QUEUED
> BITOP XOR cap-temp a-page cap-temp
QUEUED
> BITCOUNT cap-temp
QUEUED
> BITFIELD a-page:level GET u7 9
QUEUED
> BITFIELD user-d GET u7 9
QUEUED
```

```
> EXEC
1) (integer) 2
2) (integer) 2
3) (integer) 1
4) 1) (integer) 60
5) 1) (integer) 60
```

This is a slightly different case—the third result is returning a number greater than zero so this request would be rejected. However it’s important to note that it would pass on the second check—we see that the user level is acceptable.

Page 1 of 2