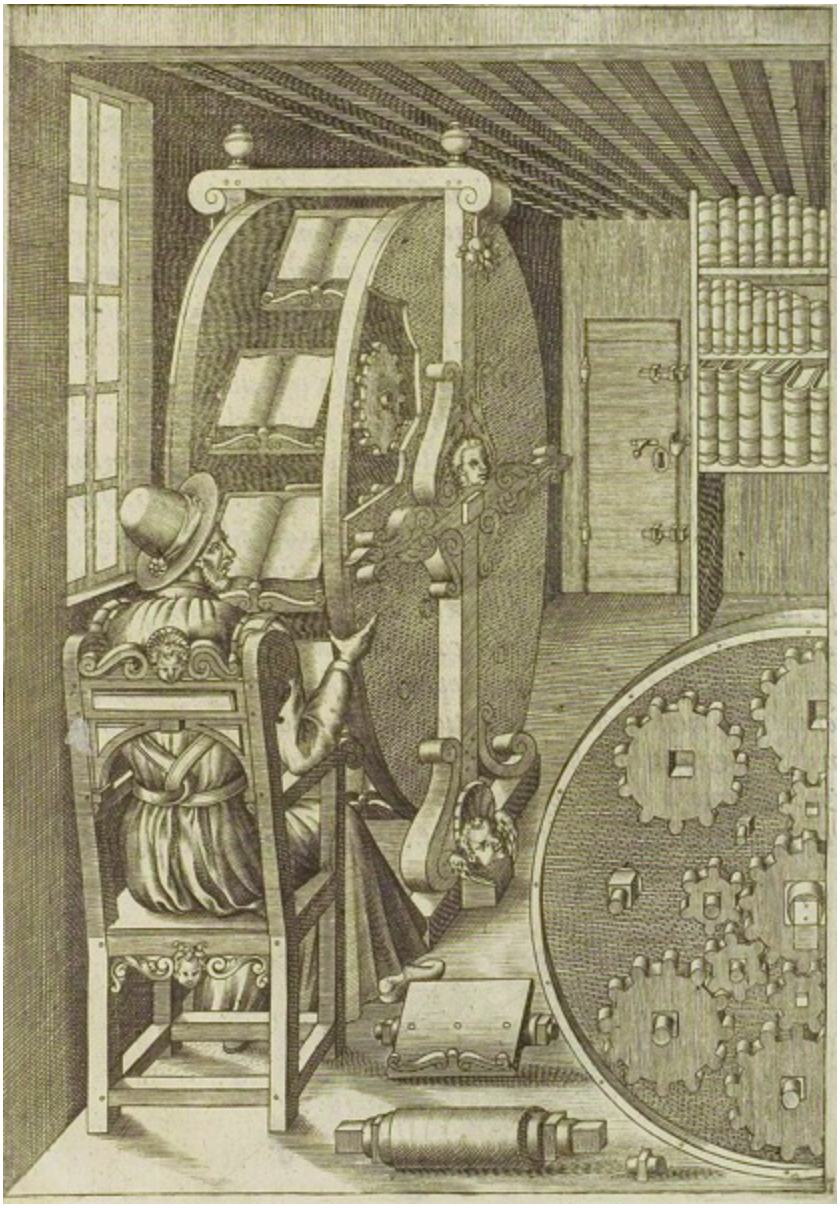
A. Jesse Jiryu Davis

Dec 6, 2012

Reading from MongoDB Replica Sets with PyMongo



Read preferences are a new feature in MongoDB 2.2 that lets you finely control how queries

are routed to replica set members. With fine control comes complexity, but fear not: I'll explain how to use read preferences to route your queries with PyMongo.

(I helped write 10gen's spec for read preferences, and I did the implementation for PyMongo 2.3.)

Contents:

The Problem

Which member of a replica set should PyMongo use for a `find`, or for a read-only command like `count`? Should it query the primary or a secondary? If it queries a secondary, which one should it use? How can you control this choice?

When your application queries a replica set, you have the opportunity to trade off consistency, availability, latency, and throughput for each kind of query. This is the problem that read preferences solve: how to specify your preferences among these four variables, so you read from the best member of the replica set for each query.

First I'll describe what a read preference is. Then I'll show PyMongo's algorithm for choosing a member. Finally, I'll discuss a list of use cases and recommend a read preference to use for each.

Read Preferences

A read preference has three parts:

Mode. This determines whether to read from the primary or secondaries. There are five modes:

- `PRIMARY`: The default mode. Always read from the primary. If there's no primary raise an exception, "AutoReconnect: No replica set primary available for query".
- `SECONDARY`: read from a secondary if there is one, otherwise raise an exception: `AutoReconnect: No replica set secondary available for query`. PyMongo prefers secondaries with short ping times.
- `PRIMARY_PREFERRED`: read from the primary if there is one, otherwise a secondary.
- `SECONDARY_PREFERRED`: read from a secondary if there is one, otherwise the primary. Again, low-latency secondaries are preferred.
- `NEAREST`: read from any low-latency member.

Tag Sets. If you've tagged your replica set members, you can use tags to specify which members to read from. Let's say you've tagged your members according to which data centers they're in. Your replica-set config is like:

```
{
    _id : "someSet",
    members : [
        {_id : 0, host : "A", tags : {"dc": "ny"}},
        {_id : 1, host : "B", tags : {"dc": "ny"}},
        {_id : 2, host : "C", tags : {"dc": "sf"}},
        {_id : 3, host : "D", tags : {"dc": "sf"}},
        {_id : 4, host : "E", tags : {"dc": "uk"}}
    ]
}
```

You could configure PyMongo to use this array of tag sets:

```
[{'dc': 'ny'}, {'dc':'sf'}, {}]
```

The driver searches through the array, from first tag set to last, looking for a tag set that matches one or more members. So if any members are online that match `{'dc': 'ny'}`, the driver picks among them, preferring those with the shortest ping times. If no members match `{'dc': 'ny'}`, PyMongo looks for members matching `{'dc':'sf'}`, and so on down the list.

The final, empty tag set `{}` means, "read from any member regardless of tags." It's a fail-safe. If you would rather raise an exception than read from a member that doesn't match a tag set, omit the empty set from the end of the array:

```
[{'dc': 'ny'}, {'dc':'sf'}]
```

In this case, if all members in New York and San Francisco are down, PyMongo will raise an exception instead of trying the member in the UK.

You can have multiple tags in a set. A member has to match all the tags. E.g., if your array of tag sets is like:

```
[{'dc': 'ny', 'disk': 'ssd'}]
```

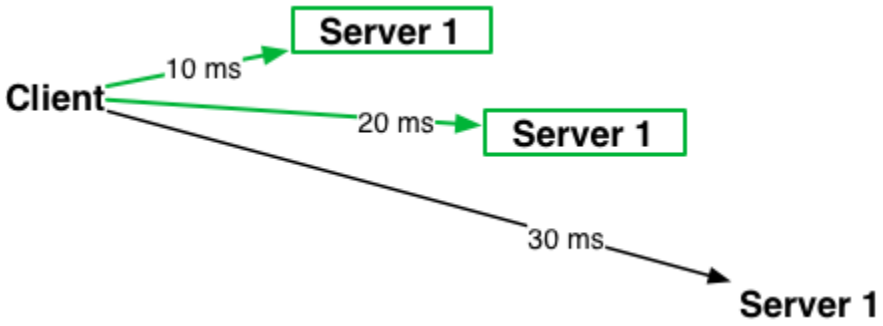
... then only a member tagged **both** with `'dc': 'ny'` and with `'disk': 'ssd'` is a match. A member's extra tags, like `'rack': 2`, have no effect.

Each mode interacts a little differently with tag sets:

- `PRIMARY`: You can't combine tag sets with PRIMARY. After all, there's only one primary, so it's senseless to ask for a primary with particular tags.
- `PRIMARY_PREFERRED`: If the primary is up, read from it no matter how it's tagged. If the primary is down, read from a secondary matching the tags provided. If there is no such secondary, raise an error.
- `SECONDARY`: Read from a secondary that matches the first tag set for which there are any matches.
- `SECONDARY_PREFERRED`: Like `SECONDARY`, or if there are no matching secondaries, like `PRIMARY`.
- `NEAREST`: Like `SECONDARY`, but treat the primary the same as the secondaries.

secondary_acceptable_latency_ms. PyMongo tracks each member's ping time (see monitoring below) and queries only the "nearest" member, or any random member no more than 15ms "farther" than it.

Say you have members who are 10, 20, and 30 milliseconds away:



PyMongo distributes queries evenly between the 10- and 20-millisecond member. It excuses the 30-millisecond member, because it's more than 15ms farther than the closest member. You can override the 15ms default by setting the snappily-named `secondary_acceptable_latency_ms` option.

The Algorithm

PyMongo chooses a member using the three parts of a read preference as a three-stage filter, removing ineligible members at each stage. For `PRIMARY`, the driver just picks the primary, or if there's no primary, raises an exception. For `SECONDARY` and `NEAREST`:

1. Apply the mode. For `SECONDARY`, filter out the primary and continue. For `NEAREST`, keep all the members and continue.

2. Apply the tag sets. If there are no tag sets configured, then pass all the members to the next stage. Otherwise, search through the array of tag sets looking for a tag set that matches some members, and pass those members to the next stage.
3. Apply ping times. First, find the nearest member who's survived filtration so far. Then filter out any members more than 15ms farther.

If several members are left at the end of the final stage, the driver picks one at random and sends it your query.

`PRIMARY_PREFERRED` uses the primary if there is one, otherwise it runs the `SECONDARY` algorithm.

`SECONDARY_PREFERRED` first runs the `SECONDARY` algorithm, and if there's no member left at the end, it uses the primary.

I can hear your objections: "It's complicated," you say. It *is* a bit complicated, but we chose this algorithm because we think it can be configured to work for any use-case you throw at it. (See use cases below.) "It's expensive," you object. The algorithm is cheaper than it sounds because it does no I/O at all. It just uses what it already knows about your replica set from periodic monitoring.

Finally, Some Code

Let's actually use read preferences with PyMongo. The simplest method is to configure a MongoReplicaSetClient. By default, the mode is `PRIMARY`, the tag sets are empty, and `secondary_acceptable_latency_ms` is 15ms:

```
from pymongo.mongo_replica_set_client import MongoReplicaSetClient

rsc = MongoReplicaSetClient('host1,host2,host3', replicaSet='foo')
```

You can override any of these options with keyword arguments.

```
from pymongo.mongo_replica_set_client import MongoReplicaSetClient
from pymongo.read_preferences import ReadPreference

rsc = MongoReplicaSetClient('host1,host2,host3', replicaSet='foo',
    read_preference=ReadPreference.SECONDARY_PREFERRED,
    tag_sets=[{'dc': 'ny'}, {}],
    secondary_acceptable_latency_ms=50)
```

(Note that what I'm calling the "mode" is configured with the `read_preference` option.)

If you initialize a MongoReplicaSetClient like this then all reads use the mode, tag sets, and latency you've configured. You can also override any of these three options post-hoc:

```
rsc = MongoReplicaSetClient('host1,host2,host3', replicaSet='foo')
```

```
rsc.read_preference = ReadPreference.NEAREST
rsc.tag_sets = [{'disk': 'ssd'}]
rsc.secondary_acceptable_latency_ms = 1000
```

You can do the same when accessing a database from a MongoReplicaSetClient:

```
db = rsc.my_database
db.read_preference = ReadPreference.SECONDARY
```

Or a collection:

```
collection = db.my_collection
collection.tag_sets = [{'dc': 'cloud'}]
```

You can even set your preference on individual method calls:

```
results = list(
    collection.find({}, secondary_acceptable_latency_ms=0))

document = collection.find_one(
    {'field': 'value'}, read_preference=ReadPreference.NEAREST)
```

Each of these four levels—connection, database, collection, method—inherits the options of the previous level and allows you to override them.

(Further reading: PyMongo read preferences, PyMongo's MongoReplicaSetClient.)

Remember slave_okay?

The old ReplicaSetConnection had a `slave_okay` option. That's deprecated now, but it still works. It's treated like `SECONDARY_PREFERRED`.

Commands

Some commands like `findAndModify` write data, others like `count` only read it. The read-only commands obey your read preference, the rest are always sent to the primary. Here are the commands that obey read preferences:

- count
- distinct
- group
- aggregate
- inline mapreduce
- collStats, dbStats
- geoNear, geoSearch, geoWalk

If you want, you can override the read preference while executing an individual command:

```
stats = rsc.my_database.command(
    'dbStats', read_preference=ReadPreference.SECONDARY)
```

Sharding

When you run `find` on a sharded cluster of replica sets, PyMongo sends your read preference to mongos. E.g., if you do a query like:

```
collection.find(
    {'field': 'value'},
    read_preference=ReadPreference.SECONDARY_PREFERRED,
    tag_sets=[{'dc': 'ny'}, {}])
```

Then PyMongo sends a query to mongos like:

```
{
    $query: {field: 'value'},
    $readPreference: {
        mode: 'secondaryPreferred',
        tags: [{'dc': 'ny'}, {}],
    }
}
```

Mongos interprets this `$readPreference` field and applies the read-preference logic to each replica set in the sharded cluster.

There are two limitations:

1. You can't override mongos's `secondary_acceptable_latency_ms`, only its mode and tag sets.
2. You can't send the `text` command to secondaries. This will probably remain unfixed, since we plan for the `text` command to be superseded by the `$text` query operator, which **will** run on secondaries in a sharded cluster as expected.

Use Cases

I want maximum consistency. By "consistency" you mean you don't want stale reads under any circumstances. As soon as you've modified some data, you want all your reads to reflect the change. In this case use `PRIMARY`, and be aware that when you have no primary (e.g. during an election, or if a majority of the replica set is offline) that every query will raise an exception.

I want maximum availability. You want to be able to query if possible. Use `PRIMARY_PREFERRED`: when there's a primary you'll get consistent reads, but if there's no

primary you can query secondaries. I like this option, because it lets your app stay online, read-only, during a failover. Be careful to test that your app behaves well under these circumstances, obviously.

I want minimum latency. Use `NEAREST`. The driver or mongos will read from the fastest member and those within 15ms of it. Be aware that you risk inconsistency: if the nearest member to your app server is a secondary with some replication lag, you could read stale data. Also note that `NEAREST` merely minimizes network lag, rather than reading from the member with the lowest IO or CPU load.

I use replica sets to distribute my data. If you have a replica set with members spread around the globe, you can tag them like in the tag sets example above. Then, configure your application servers to query the members nearby. For example, your New York app servers do:

```
rsc.read_preference = ReadPreference.NEAREST
rsc.tag_sets = [{'dc': 'ny'}, {}]
```

Although `NEAREST` favors nearby secondaries anyway, including the tag makes the choice more predictable.

I want maximum throughput. Use `NEAREST` and set `secondary_acceptable_latency_ms` very high, like 500ms. This will distribute the query load equally among all members, thus (under most circumstances) giving you maximum read throughput.

If you want to move read load off your primary, use mode `SECONDARY`. It's tempting to use `SECONDARY_PREFERRED`, but if your primary can't take your full read load, you probably prefer for your queries to fail than to move all the load to the primary whenever your secondaries are unavailable.

Monitoring

PyMongo needs to know a lot about the state of your replica set to know which members to use for your read preference. If you create a MongoReplicaSetClient like:

```
rsc = MongoReplicaSetClient('host1,host2,host3', replicaSet='foo')
```

...then the MongoReplicaSetClient tries to connect to each server, in random order, until it finds one that's up. It runs the `isMaster` command on that server. The server's response tells the MongoReplicaSetClient which members are in the replica set now, how they're tagged, and who's primary. MongoReplicaSetClient then calls `isMaster` on each member currently in the set and records the latency. This is what I called "ping time" above.

Once all that's complete, the MongoReplicaSetClient launches a background thread called the Monitor. The Monitor wakes every 30 seconds and refreshes its view of the replica set: it runs `isMaster` on all the members again, marks as "down" any members it can't reach, and notes new members who've joined. It also updates its latency measurement for each member. It uses a 5-sample moving average to track each member's latency.

If a member goes down, MongoReplicaSetClient won't take 30 seconds to notice. As soon as it gets a network error attempting to query a member it thought was up, it wakes the Monitor to refresh ASAP.

In Conclusion

There's a lot of options and details here. If you just want to query the primary, then accept the default, and if you just want to move load to your secondaries, use `SECONDARY`. But if you're the kind of hotrodder who needs to optimize for consistency, availability, latency, or throughput with every query, read preferences give you total control.

Categories: Mongo, Programming, Python
Tags: pymongo

← The Chuang Tzu

YieldPoints: simple extensions to tornado.gen →

8 Comments emptysquare Login

Sort by Best Share Favorite ★

Join the discussion…

Nguyễn Cảnh Hiếu • 2 years ago

thanks, an interesting article! It is helpful for me

• Reply • Share ›

Vasiliy Faronov • 2 years ago

I (like many) have a read-heavy application. I want maximum consistency *and* minimum read latency, but I'm willing to accept extra write latency. I cannot just set write concern to 3 (the size of the set) since that would break writes when one of the nodes is down. And I cannot set it to 2 because then I risk reading inconsistent data.

I've seen a suggestion to use tag-based write concern. I haven't tried this myself yet. Can you comment on this?

• Reply • Share ›

A. Jesse Jiryu Davis Mod → Vasiliy Faronov • 2 years ago

Read from the primary! If it's too slow, optimize your schema or your indexes. When you read from secondaries there are unavoidable tradeoffs.

• Reply • Share ›

Vasiliy Faronov → A. Jesse Jiryu Davis • 2 years ago

Reading from the primary is what I'm doing now, of course.

It's not "too slow", it's "a little faster (reading from localhost instead of another DC) would make the users a little happier".

How is this unavoidable? Is there any fundamental problem in making writes replicate to "all active RS members" rather than "3 RS members"?

• Reply • Share ›

A. Jesse Jiryu Davis Mod → Vasiliy Faronov • 2 years ago

All writes always replicate to all active members. This is a common and severe misunderstanding of what "w=3" means. "w=3" just means, "wait for 3 members to receive the write." It doesn't control how many members receive the write — all