# Call me maybe: Elasticsearch

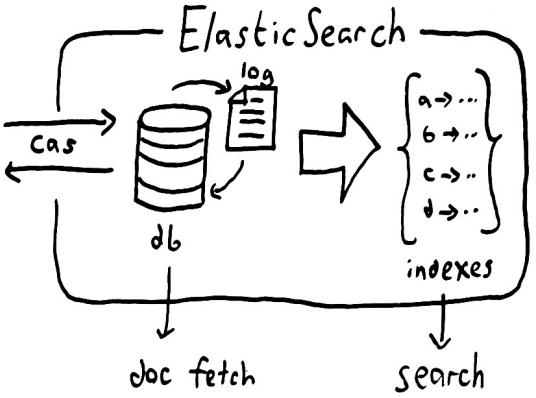
2014/06/15
Distributed Systems Jepsen Elasticsearch

Previously, on Jepsen, we saw RabbitMQ throw away a staggering volume of data. In this post, we'll explore Elasticsearch's behavior under various types of network failure.


Elasticsearch is a distributed search engine, built around Apache Lucene–a well-respected Java indexing library. Lucene handles the on-disk storage, indexing, and searching of documents, while ElasticSearch handles document updates, the API, and distribution. Documents are written to collections as free-form JSON; schemas can be overlaid onto collections to specify particular indexing strategies.

As with many distributed systems, Elasticsearch scales in two axes: *sharding* and *replication*. The document space is sharded–sliced up–into many disjoint chunks, and each chunk allocated to different nodes. Adding more nodes allows Elasticsearch to store a document space larger than any single node could handle, and offers quasilinear increases in throughput and capacity with additional nodes. For fault-tolerance, each shard is *replicated* to multiple nodes. If one node fails or becomes unavailable, another can take over. There are additional distinctions between nodes which can process writes, and those which are read-only copies–termed "data nodes"–but this is primarily a performance optimization.

Because index construction is a somewhat expensive process, Elasticsearch provides a faster, more strongly consistent database backed by a write-ahead log. Document creation, reads, updates, and deletes talk directly to this strongly-consistent database, which is asynchronously indexed into Lucene. Search queries lag behind the "true" state of Elasticsearch records, but should eventually catch up. One can force a flush of the transaction log to the index, ensuring changes written before the flush are made visible.




But this is Jepsen, where nothing works the way it's supposed to. Let's give this system's core assumptions a good shake and see what falls out!

## What does the docs say?

What is the Elasticsearch consistency model? When I evaluate a new database, I check the documentation first.

The docs tell us that Elasticsearch provides optimistic concurrency control: each document has an atomic version number, and updates can specify a particular version required for the write to go through; this allows atomic CAS and provides the basis for independently linearizable updates to each document; i.e. every update to a particular document appears to take place atomically between the start of a request and its completion.

Moreover, Elasticsearch requires synchronous acknowledgement of each write from a majority of nodes, which suggests that a.) nodes in the minority side of a network partition will reject writes, and b.) acknowledged writes will be durable in the event that a new primary node is elected. From the documentation:

> To prevent writes from taking place on the "wrong" side of a network partition, by default, index operations only succeed if a quorum (>replicas/2+1) of active shards are available.
>
> By default, the index operation only returns after all shards within the replication group have indexed the document (sync replication).

Synchronous replication to a majority of replicas, plus the use of version-based CAS, suggests that Elasticsearch has all the primitives required to treat documents as linearizable registers. Moreover, if we issue a flush command, Elasticsearch will apply any outstanding writes from the transaction log to Lucene; a composite `flush`+`search` operation should also be linearizable, much like Zookeeper's `sync`+`read`.

So: the consistency docs paint an optimistic picture–but what does Elasticsearch do when a node or network fails?


A search for partition on `elasticsearch.org` site reveals only a single reference–the "on the 'wrong' side of a network partition" comment regarding quorum writes. A search for fault tolerance returns only a single result–an overview blog post which simply describes Elasticsearch as "resilient to failing nodes".

The results for "failure" are a bit more comprehensive, though. Coping with failure, part of the introductory tutorial, explains that after killing a node,

> The first thing that the new master node did was to promote the replicas of these shards on Node 2 and Node 3 to be primaries, putting us back into cluster health yellow. This promotion process was instantaneous, like the flick of a switch.

I like "instantaneous" promotion. Fast convergence times are key for testing a system in Jepsen; systems that take minutes to converge on a new state drag out the testing process, especially when testing eventual consistency.

> So why is our cluster health yellow and not green? We have all 3 primary shards, but we specified that we wanted two replicas of each primary and currently only one replica is assigned. This prevents us from reaching green, but we're not too worried here: were we to kill Node 2 as well, our application could still keep running without data loss because Node 3 contains a copy of every shard.

So Elasticsearch claims to cope with the loss of a majority of nodes. That could be a problem from a consistency standpoint: it means that reads might, depending on the implementation, be able to read stale data. As we saw in the RabbitMQ post, stale reads rule out a number of consistency models. We'll be careful to wait for the cluster to report green before doing a read, just in case.

## The broader landscape

So the official documentation is inspecific about fault tolerance. The next thing I look at in evaluating a database is reports from users: Blog posts, mailing list questions, and so on. Let's start with a search for Elasticsearch partition tolerance.

Result #1 is this mailing list discussion from 2010, where Tal Salmona asks what ElasticSearch guarantees with respect to the CAP theorem. Shay Banon, the primary author of Elasticsearch, responds:

> When it comes to CAP, in a very high level, elasticsearch gives up on partition tolerance. This is for several reasons:

2010 was a simpler time.

> I personally believe that *within the same data center", network partitions very rarely happen, and when they do, its a small set (many times single) machine that gets "partitioned out of the network". When a single machine gets disconnected from the network, then thats not going to affect elasticsearch.

Quantitative evidence is hard to find, but we have some data. Rare events are still worth defending against. Still, this is somewhat reassuring: single-node partitions were a design consideration, and we can expect Elasticsearch to be robust against them.

> When it comes to search engines, and inverted index, its very hard to the point of impossible to build a solution that tries to resolve consistency problems on the fly as most products do (the famous "read repair"). When you search, you search over a large amount of docs and you can't read repair each one all the time. Key value / column based solutions have life easy

This is true! Search is a hard problem to distribute. It doesn't tell us much about Elasticsearch's *behavior*, though. Sergio Bossa asks what happens in a simple partitioned scenario, and Banon responds with some very interesting discussion of the general problem, but not many specifics. One key point:

> One simple option to solve this is to have the user define a "minimum" size of cluster that should be up, and if its not, the nodes will not allow writes. This will be in elasticsearch and its actually not that difficult to implement.

This is now built into Elasticsearch as an option called discovery.zen.minimum_master_nodes. As numerous users have noted, Elasticsearch will happily allow concurrent primary nodes when this number is less than n/2+1. Why is it even

legal to set it lower? I'm not sure, but we'll make sure to set the minimum_master_nodes to 3 for our five-node cluster.

All this is old news, though. The next interesting result is a mailing list discussion from early 2014, in which Nic asks for a clarification of Elasticsearch's CAP tradeoffs, given the 2010 thread.

> From my understanding it seems like Kimchy was confused here. As a distributed system ES can't give up on the P - you can't will network/communication failures out of existent!
>
> Instead, it seems like ES mostly compromises on the A (availability) part of CAP. For example, unless you are willing to suffer potential split-brain scenarios, setting min master nodes to n/2 + 1 will mean the smaller group under a network partition will become unavailable (it will not respond to read/writes). If you do allow split-brain then clearly consistency is compromised and the client service will need to have some kind of conflict resolution mechanism.

My thoughts exactly. What's going on here?

> It would be great if there were a page on the ES site/guide which went into these issues in more detail as it is (IMO) essential information in understanding how ES works and in deciding whether it is appropriate for your use case.

Yes, yes, a thousand times yes. One of the principle reasons I put so much time into Jepsen is to prompt database vendors to clearly explain their safety and liveness invariants.

Jörg Prante, a regular on the mailing list, responds:

> ES gives up on partition tolerance, it means, if enough nodes fail, cluster state turns red and ES does not proceed to operate on that index.

That sounds like giving up availability to me. Remember, CAP A means that every request to a non-failing node completes successfully.

> ES is not giving up on availability. Every request will be responded, either true (with result) or false (error). In a system being not available, you would have to expect the property of having some requests that can no longer be answered at all (they hang forever or the responder is gone).

**Errors are not successful responses; they don't count towards availability.**

> The principle design of distributed operations in ES is like this: write all ops on an index into a WAL (the translog). Send the ops to the nodes while even some nodes may work reliable, some not. Stopping a node does not harm the cluster as long as the replica level is high enough. When a stopped node rejoins, initiate a recovery, using the WAL. Let the "best" WAL result of all consistent results of the replica win for recovering the index state.

This is useful, concrete information, and appears to describe something like a CP system, but given the poster's confusion with respect to CAP I'd like a more authoritative explanation.

Itamar Syn-Hershko responds to Prante, arguing Elasticsearch gives up consistency when configured with fewer than quorum primary-eligible nodes. This seems valid to me, but I'm curious what happens when we *do* use the quorum constraints correctly.

> The very fact that you can have enough replicas of your index which will make the cluster never get to a red state (e.g. the number of nodes you have) proves this. An index (an eventually, all indexes on your cluster) can survive a network split. It can also be always available, hence ES is AP.
>
> Elasticsearch's compromise is on C - consistency - like most NoSQL databases. It uses Eventual Consistency to answer queries, not just because of NRT search, but also because you may be querying a replica (a slave node) which hasn't been brought up to speed yet.

Writes may be linearizable even if reads aren't. We might be able to get Elasticsearch to do CAS safely, and test it by waiting for the cluster to recover and performing an index flush before read.

Prante responds, asserting that Elasticsearch compare-and-set operations are linearizable.

> Consistency is not given up by ES. First, on doc level, you have "write your own read" consistency implemented as versioning - a doc read followed by a doc write is guaranteed to be consistent if both read and write versions match (MVCC).
>
> Replica are not interfering with consistency, they are for availability.

As we've discussed, replicas are the reason why consistency is hard in the first place.

> split brains can happen and ES can happily proceed reading and writing to the index in such a case, but the result is not predictable - the usual case is that two masters are going to control two divergent indices, and that is catastrophic. This is not a fault but a (nasty) feature, and must be controlled by extra safeguarding, by setting the minimum master value in the configuration ⋯

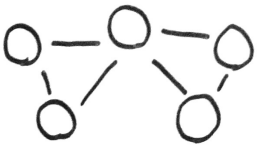
Well, I'm a little confused by all this, but the impression that I get is that if we set `minimum_master_nodes` to a majority of the nodes in the cluster, Elasticsearch should be safe.

That's really *it* for partition discussion, as far as I can tell. No official docs explaining the resolution algorithm, and some possibly outdated, contradictory discussion on a mailing list. There are a number of blog posts that repeat these sources, and some meetup discussions, but nothing concrete.

## Nontransitive partitions

Then somebody pointed me at this GitHub issue, reported by Saj Goonatilleke in December 2012. It's titled "minimum_master_nodes does not prevent split-brain if splits are intersecting." That sounds interesting.

> With this setup, I can easily split a 3-node cluster into two 'hemispheres' (continuing with the brain metaphor) with one node acting as a participant in both hemispheres. I believe this to be a significant problem, because now minimum_master_nodes is incapable of preventing certain split-brain scenarios.



In Saj's test case, a split in a three-node cluster isolates two nodes from each other, but not from a common node visible to both. Because both isolated nodes can see two thirds of the cluster (themselves and the common node), they believe they are eligible for leader election even when `minimum_master_nodes` is at least a majority.

This points to a deeper problem in Elasticsearch's leader election process. Most leader election algorithms divide time into monotonic *epochs* or *terms*, and allow only one node to become the leader for a given term. They often enforce this constraint by requiring that a given node can only support *one* candidate node per term. This property, combined with the constraint that a candidate must receive votes from a majority to become leader for a term, ensures that there will be at most one leader per term.

ZenDisco, Elasticsearch's cluster membership system, has no such invariant. A node will happily support two leaders simultaneously.

Some users reported success in switching to the Zookeeper plugin for leader election, but that component has been, as far as I know, broken since 0.90.6.

## Building a nemesis


This sort of scenario isn't hard to verify with Jepsen, and it's a good opportunity to introduce a Jepsen concept that we haven't explored yet: the *nemesis*. Nemeses are a special type of Jepsen client, but instead of performing operations against a node in the database, they play havoc with the entire cluster; severing network links, adjusting clocks, killing processes, causing single-bit errors on disk, and so on.

First things first: we'll use iptables to drop traffic from a node, or a collection of nodes.

```
(defn snub-node!
  "Drops all packets from node."
  [node]
  (c/su (c/exec :iptables :-A :INPUT :-s (net/ip node) :-j :DROP)))

(defn snub-nodes!
  "Drops all packets from the given nodes."
  [nodes]
  (dorun (map snub-node! nodes)))
```

Next, we need a way to fully describe a given network topology. One way would be as an adjacency list, but for our purposes it's a little clearer to describe the links that we're going to *cut*.

```
(defn partition!
  "Takes a *grudge*: a map of nodes to the collection of nodes they should
  reject messages from, and makes the appropriate changes. Does not heal the
  network first, so repeated calls to partition! are cumulative right now."
  [grudge]
  (->> grudge
       (map (fn [ [node frenemies]]
              (future
                (c/on node (snub-nodes! frenemies)))))
       doall
       (map deref)
       dorun))
```

What kind of grudge might we want to hold? Let's call a *complete* grudge one in which all links are symmetric: if A can talk to B, B can talk to A, and vice-versa. We'll be nice to our networks, and avoid unidirectional partitions for now–though they can happen in production, and Magnus Haug reported just last week that [asymmetric partitions will wedge an Elasticsearch cluster in a split-brain state](#).

```
(defn complete-grudge
  "Takes a collection of components (collections of nodes), and computes a
  grudge such that no node can talk to any nodes outside its partition."
  [components]
  (let [components (map set components)
        universe   (apply set/union components)]
    (reduce (fn [grudge component]
              (reduce (fn [grudge node]
                        (assoc grudge node (set/difference universe component)))
                      grudge
                      component))
            {}
            components)))
```

We want to cut the network roughly in half, so we'll take a list of nodes and chop it into two pieces. Then we can feed those pieces to `complete-grudge` to compute the links we need to cut to isolate both nodes into their own network components.

```
(defn bisect
  "Given a sequence, cuts it in half; smaller half first."
  [coll]
  (split-at (Math/floor (/ (count coll) 2)) coll))
```

Finally, a slight modification to the complete grudge: we're going to allow one node to talk to *everyone*.

```
(defn bridge
  "A grudge which cuts the network in half, but preserves a node in the middle
  which has uninterrupted bidirectional connectivity to both components."
  [nodes]
  (let [components (bisect nodes)
        bridge     (first (second components))]
    (-> components
        complete-grudge
        ; Bridge won't snub anyone
        (dissoc bridge)
        ; Nobody hates the bridge
        (->> (util/map-vals #(disj % bridge))))))
```

Last piece of the puzzle: we've got to respond to operations by initiating and healing the network partition we've computed. This `partitioner` nemesis is generic–it takes a pluggable function, like `bridge`, to figure out what links to cut, then calls `partition!` with those links.

```
(defn partitioner
  "Responds to a :start operation by cutting network links as defined by
  (grudge nodes), and responds to :stop by healing the network."
  [grudge]
  (reify client/Client
```

```
    (setup! [this test _]
      (c/on-many (:nodes test) (net/heal))
      this)

    (invoke! [this test op]
      (case (:f op)
        :start (let [grudge (grudge (:nodes test))]
                 (partition! grudge)
                 (assoc op :value (str "Cut off " (pr-str grudge))))
        :stop  (do (c/on-many (:nodes test) (net/heal))
                   (assoc op :value "fully connected"))))

    (teardown! [this test]
      (c/on-many (:nodes test) (net/heal)))))
```

Let's give it a shot. In this test we'll implement a linearizable set which supports adding elements and reading the current set, just like we did with Etcd, by using Elasticsearch's compare-and-set primitives that enqueues integers (slightly staggered in time to avoid too many CAS failures) , while cutting the network into the nontransitive partition shape reported in the ticket.

```
(deftest register-test
  (let [test (run!
              (assoc
                noop-test
                :name      "elasticsearch"
                :os        debian/os
                :db        db
                :client    (cas-set-client)
                :model     (model/set)
                :checker   (checker/compose {:html timeline/html
                                             :set  checker/set})
                :nemesis   (nemesis/partitioner nemesis/bridge)
                :generator (gen/phases
                            (->> (range)
                                 (map (fn [x] {:type  :invoke
                                               :f     :add
                                               :value x}))
                                 gen/seq
                                 (gen/stagger 1/10)
                                 (gen/delay 1)
                                 (gen/nemesis
                                   (gen/seq
                                     (cycle [(gen/sleep 60)
                                             {:type :info :f :start}
                                             (gen/sleep 300)
                                             {:type :info :f :stop}])))
                                 (gen/time-limit 600))
                            (gen/nemesis
                              (gen/once {:type :info :f :stop}))
                            (gen/clients
                              (gen/once {:type :invoke :f :read}))))]
    (is (:valid? (:results test)))
    (pprint (:results test))))
```

## Speed bumps

The first thing I found was, well, nothing. The test crashed with a cryptic error message about exceptions–and it took a while to figure out why.

Elasticsearch supports two protocols: an HTTP protocol, which is slower, and a native binary protocol. In most cases you want to use the native protocol where available. Unfortunately, Elasticsearch uses Java serialization for exceptions–which is problematic because different JVMs serialize classes differently. In particular, InetAddress's representation is not stable across different JVM patchlevels, which means that servers or even clients running different JVM versions will explode when trying to parse an error message from another node.

Another roadblock: "instantaneous" cluster convergence isn't. It takes Elasticsearch *ninety seconds* (three rounds of 30-second timeouts) to detect a node has failed and elect a new primary. We'll have to slow down the test

schedule, allowing the network to stabilize for 300 seconds, in order to get reliable cluster transitions. There's a configuration option to adjust those timeouts…

```
# Set the time to wait for ping responses from other nodes when discovering.
# Set this option to a higher value on a slow or congested network
# to minimize discovery failures:
#
discovery.zen.ping.timeout: 3s
```

But as far as I can tell, that knob doesn't make failure detection any faster.

```
[2014-06-16 17:25:40,901][INFO ][discovery.zen                ] [n5] master_left [[n1]
[9fvd-dZBTo2re4dukDuuRw][n1][inet[/192.168.122.11:9300]]], reason [failed to ping,
tried [3] times, each with  maximum [30s] timeout]
```

Elasticsearch has some really terrific tools for cluster introspection: you can request a JSON dump of the cluster status easily via CURL, which makes writing tests a breeze. You can also–and this is really awesome–block until the cluster reaches some desired status. In the Jepsen client, I block until the cluster health reaches green before starting the test, and once the network is repaired, before doing a final read of the set.

The problem is that the health endpoint will lie. It's happy to report a green cluster during split-brain scenarios. I introduced additional delays to the test schedule to give Elasticsearch even more time to recover, with mixed success. Sometimes the cluster will wedge hard, and refuse to make progress until I start bouncing nodes.

Anyway, with those caveats: results!

## Nontransitive partition results

Elasticsearch compare-and-set isn't even *close* to linearizable. During this type of network partition, both primaries will happily accept writes, and the node in the middle… well, it's caught in a three-way call with Regina George, so to speak. When the cluster comes back together, one primary blithely overwrites the other's state.


Knossos reported a linearization failure–but the final state was misssing *tons* of writes, and Knossos panics as soon as a single one is lost. "Welp, my job is done here: peace".

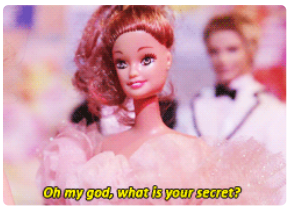
But we're not just interested in *whether* the system is linearizable. We want to quantify just how much data might be corrupted. I wrote a custom checker for Elasticsearch to try and quantify that write loss, similar to the RabbitMQ queue checker.

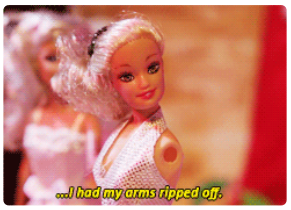
Here's a result from a typical run with that checker

```
:total 1961,
:recovered-count 4,
:unexpected-count 0,
:lost-count 645,
:ok-count 1103
```

**645 out of 1961 writes acknowledged then lost**. Only 1103 writes made it to the final read. Elasticsearch issued more false successes than failure results.

Clearly Elasticsearch's MVCC support is neither consistent nor available; it happily allows two concurrent primaries to accept writes, and destroys the writes on one of those primaries when the partition resolves, but the majority-write constraint prevents Elasticsearch from offering total availability.





What's really surprising about this problem is that it's gone unaddressed *for so long*. The original issue was

reported in July 2012; almost two full years ago. There's no discussion on the website, nothing in the documentation, and users going through Elasticsearch training have told me these problems weren't mentioned in their classes. Fixing a distributed system is hard, but documenting its gotchas is *easy*. Look to RabbitMQ or Riak, both of which have extensive documentation around failure recovery, and the scenarios in which their respective systems could lose or corrupt data.

This is not a theoretical constraint. There are dozens of users reporting data loss in this ticket's discussion, and more on the mailing list. Paul Smith reports

> I'm pretty sure we're suffering from this in certain situations, and I don't think that it's limited to unicast discovery.
>
> We've had some bad networking, some Virtual Machine stalls (result of SAN issues, or VMWare doing weird stuff), or even heavy GC activity can cause enough pauses for aspects of the split brain to occur.

Ivan Brusic confirms:

> I have seen the issue on two different 0.20RC1 clusters. One having eight nodes, the other with four.

Trevor Reeves observes:

> We have been frequently experiencing this 'mix brain' issue in several of our clusters - up to 3 or 4 times a week. We have always had dedicated master eligible nodes (i.e. master=true, data=false), correctly configured minimum_master_nodes and have recently moved to 0.90.3, and seen no improvement in the situation.

From Elad Amit:

> We just ran into this problem on a 41 data node and 5 master node cluster running 0.90.9

And Mark Tinsley:

> I have been having some strange occurrences using elasticsearch on aws…. For some reason, node A and B cannot talk to each other… but both can still talk to C and C can talk to A and B i.e. a 'on the fence' network partition as C can still see all…. As you can see B is now a new master but A has not been removed as a master, because A can still see C so has the minimum master node criteria satisfied.

Some of these problems can be ameliorated by improving Elasticsearch's GC performance and overall responsiveness–but this will only reduce the frequency of split-brain events, not fix the underlying problem. Elasticsearch needs a real consensus algorithm to linearize updates to documents.

## Needless data loss


So let's back off. Maybe version control is a lost cause–but there is something Elasticsearch can *definitely* handle safely, and that's inserts. Inserts will never conflict with one another, and can always be merged with set union. Updates and deletes to those documents may not linearize correctly, but we can bias towards preserving data. Moreover, insertions can theoretically be 100% available: every node can take writes and queue them for later, modulo durability constraints.

Here's a client which implements a set of integers by inserting a fresh document with an auto-generated ID for each element.

The results? Massive write loss.

```
:total 2023,
:recovered-count 77,
:unexpected-count 0,
:lost-count 688,
:ok-count 1265
```

**Of two thousand attempts, six hundred eighty eight documents were acknowledged by Elasticsearch then thrown away.**

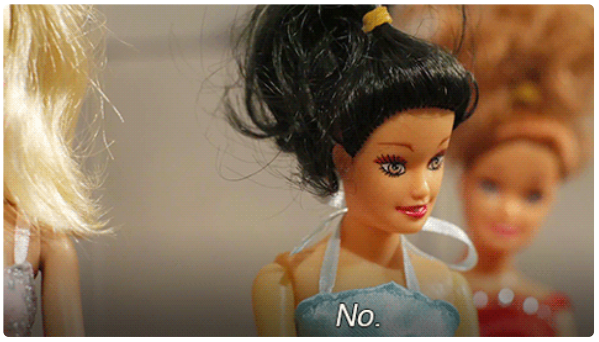
This is something the ticket didn't make clear: Elasticsearch *could* merge divergent replicas by preserving inserts on both nodes–but it doesn't. Instead it throws all that data away. In fact, sometimes you'll get mind-boggling errors like

```
{:body
  {"error": "RemoteTransportException[ [Cadaver][inet[/192.168.122.12:9300]][index]];
nested: RemoteTransportException[ [Death Adder][inet[/192.168.122.11:9300]][index]];
nested: DocumentAlreadyExistsException[ [jepsen-index][1] [number][EpVU56YERBOfRqyVc-
_hAg]: document already exists]; ",
   "status": 409}}
```

Elasticsearch claims–and Wireshark traces confirm–that documents inserted without an ID will receive an auto-generated UUID for their document ID. How is it possible that an insert of a fresh document with a fresh UUID can fail because that document already exists? Something is seriously wrong here.

## Random transitive partitions

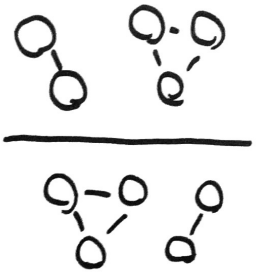
Maybe you don't believe that nontransitive–e.g. bridged–network partitions are all that common. Can Elasticsearch handle a complete, disjoint partition which isolates the cluster cleanly into two halves, without a node connecting the two sides?




Here's a nemesis to generate this kind of partition. To determine the grudge–the links to cut–we'll shuffle the list of nodes randomly, cut it in half, and isolate both halves from each other completely. We'll use `comp` again to compose several functions together.

```
(defn partition-random-halves
  "Cuts the network into randomly chosen halves."
  []
  (partitioner (comp complete-grudge bisect shuffle)))
```

Because this partitioner shuffles the nodes, each time we invoke it we'll get a *different* minority and majority component to the partition. This is a pattern that a CP system can handle well: because a majority is always visible, after a brief reconvergence latency spike a linearizable database could offer availability *throughout* this test.



What does Elasticsearch do in this scenario?

Elasticsearch loses data even when the partitions are total. You don't need a bridge node to make replicas diverge. Again, we're not even talking CAS updates–these are all freshly inserted documents; no changes, no deletions. Every single one of these writes could have been preserved safely.

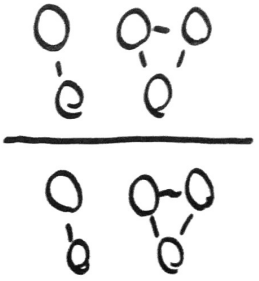
```
FAIL in (create-test) (elasticsearch_test.clj:83)
expected: (:valid? (:results test))
  actual: false
{:valid? false,
 :html {:valid? true},
 :set
 {:valid? false,
  :lost
  "#{348 350 372 392..393 396 403 427 446 453 458 467 476 504 526 547 568..569 573 578
599 603 607 648 652}",
  :recovered
  "#{273 281 285..286 290..292 296 301 305 311 317 323 325 328..329 334 340 345 353 356
360 365 368..369 377..378 384 395 398..399 404 406 412..413 417..419 422 425..426
430..432 435 437..438 442 445 449..450 452 454 457 462..463 470 473 475 477 582 593 611
615 630 632 653 657 671 675 690 694 708 712 727 729 744 748 1034 1038 1040 1042..1043
1045..1046 1050 1052 1055 1057..1058 1060 1062 1067..1068 1070 1072 1075 1077..1078
1080 1082..1083 1085 1087 1090 1092..1093 1095 1098 1100 1107 1112..1113 1115
1117..1118 1120 1122..1123 1125 1127 1130 1132..1133 1135 1138 1140 1142..1143 1145
1147..1148 1150 1153 1155 1157..1158 1160 1162..1163 1165 1167..1168 1170 1172..1173
1175 1177..1178}",
  :ok
```

```
  "#{0..269 273 278 281 285..286 290..292 296..297 301..302 305 311..313 317 323..325
328..329 334 338 340 345..347 351 353 356 358..360 365 368..369 375..378 380 383..384
389 395 398..399 401..402 404 406 409 411..413 417..419 422..426 430..432 435..438
441..443 445 447 449..450 452 454 456..457 459 461..463 465..466 468 470..473 475 477
479 481..484 486..488 490..503 505 507..508 510..525 528..530 532..546 549..551
553..567 570..572 575..577 579..598 600..602 604..606 608..647 649..651 653..1035 1038
1040 1042..1043 1045..1046 1050 1052 1055 1057..1058 1060 1062 1067..1068 1070 1072
1075 1077..1078 1080 1082..1083 1085 1087 1090 1092..1093 1095 1098 1100 1107
1112..1113 1115 1117..1118 1120 1122..1123 1125 1127 1130 1132..1133 1135 1138 1140
1142..1143 1145 1147..1148 1150 1153 1155 1157..1158 1160 1162..1163 1165 1167..1168
1170 1172..1173 1175 1177..1178}",
  :total 1180,
  :recovered-count 149,
  :unexpected-count 0,
  :unexpected "#{}",
  :lost-count 25,
  :ok-count 970}}
```

Only 25 writes thrown away this round, but that number *should* have been zero. You can lower the probability of loss by adding more replicas, but even with a replica on every single node, Elasticsearch *still* finds a way to lose data.

## Fixed transitive partitions

Maybe the problem is that the partitions are *shifting* over time–perhaps isolating one set of nodes, then another set, is what confuses Elasticsearch. We might be looking at a time horizon too short for Elasticsearch to stabilize between failures, causing it to interpret the disjoint components as some partially-transitive graph.



But as Shikhar Bhushan noted last week, even a nemesis which always generates the *same* partitioning pattern, like [n1 n2] [n3 n4 n5], causes split brain and write loss. This nemesis is just like the random partitoner, but we drop the `shuffle` step.

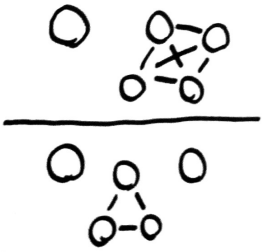
```
(defn partition-halves
  "Responds to a :start operation by cutting the network into two halves--first
  nodes together and in the smaller half--and a :stop operation by repairing
  the network."
  []
  (partitioner (comp complete-grudge bisect)))
```

Running this test reveals that it's not just shifting partitions: a constant partition pattern will cause Elasticsearch to enter split brain and throw away data.

```
 {:valid? false,
  :lost
  "#{90 94 104 107 122..123 135 160 173 181 188 200 229 279 337 398 422}",
  :recovered
  "#{6 8 12 15 17 19 23 27 29 31 34 36 38 40 43..44 47 50 53 55 61 64 66 71 74 78 81
83..84 86 91..92 97 99..100 103 109..110 114 116 119..121 126 132..133 137..139
142..144 147..149 152..154 157..159 163..165 168..169 171 176..177 179 182 184..185}",
  :ok
  "#{0..4 6 8 12 15 17 19 23 27 29 31 34 36 38 40 43..44 47 50 53 55 61 64 66 71 74 78
81 83..84 86 91..92 97 99..100 103 109..110 114 116 119..121 126 132..133 137..139
142..144 147..149 152..154 157..159 163..165 168..171 175..179 182 184..185 187 189
191..192 194..199 201..203 205..210 212..223 225..228 230..231 233..241 243..251
253..261 263..270 272..278 280..282 284..289 291..301 303..307 309..319 321..326
328..336 338..342 344 346..360 362..364 366..378 380..382 384..397 399..401 404..420
423..425 427..439 441..443 445..458 460}",
  :total 461,
  :recovered-count 73,
  :unexpected-count 0,
  :unexpected "#{}",
  :lost-count 17,
  :ok-count 319}}
```

Seventeen writes lost here. It doesn't take a shifting pattern of failures to break the cluster.

## Single-node partitions



Maybe this is to be expected; Banon asserted on the mailing list that partitions involving multiple nodes are rare–but Elasticsearch should be able to handle a single node being isolated.

By symmetry, the most interesting node to isolate is the primary. To *find* the primary, we can query Elasticsearch's cluster status endpoint on each host, to see what *it* thinks the current primary is.

We'll take the nodes, make an HTTP request to each, and extract the `master_node` field of the result. That's a unique instance identifier, so we dig into the `nodes` structure to map that identifier back to a node name. Then we turn those `[node, supposed-primary]` pairs into a hashmap.

```
(defn primaries
  "Returns a map of nodes to the node that node thinks is the current primary,
  as a map of keywords to keywords. Assumes elasticsearch node names are the
  same as the provided node names."
  [nodes]
  (->> nodes
       (pmap (fn [node]
               (let [res (-> (str "http://" (name node)
                                  ":9200/_cluster/state")
                             (http/get {:as :json-string-keys})
                             :body)
                     primary (get res "master_node")]
                 [node
                  (keyword (get-in res ["nodes" primary "name"]))])))
       (into {})))
```

Let's limit ourselves to nodes that think *they're* the primary.

```
(defn self-primaries
  "A sequence of nodes which think they are primaries."
  [nodes]
  (->> nodes
       primaries
       (filter (partial apply =))
       (map key)))
```

We're just filtering that map to pairs where the key and value are equal, then returning the keys–nodes that think they themselves are current primaries. Next, we'll write a special partitioner, just for Elasticsearch, that isolates those nodes.

```
(def isolate-self-primaries-nemesis
  "A nemesis which completely isolates any node that thinks it is the primary."
  (nemesis/partitioner
    (fn [nodes]
      (let [ps (self-primaries nodes)]
        (nemesis/complete-grudge
          ; All nodes that aren't self-primaries in one partition
          (cons (remove (set ps) nodes)
                ; Each self-primary in a different partition
                (map list ps)))))))
```

The first time we run this, a single node will be isolated. We'll let the cluster converge for… say 200 seconds–more than long enough for pings to time out and for a new primary to be elected, before repairing the partition. Then we'll initiate a second partition–again cutting off every node that thinks it's a primary. There might be more than one if Elasticsearch is still in split-brain after those 200 seconds.

Elasticsearch still loses data, even when partitions isolate only single nodes.

```
FAIL in (create-test) (elasticsearch_test.clj:86)
```

```
expected: (:valid? (:results test))
  actual: false
{:valid? false,
 :html {:valid? true},
 :set
 {:valid? false,
  :lost "#{619..620 624 629..631 633..634}",
  :recovered
  "#{7..8 10..11 15 24 26..27 30 32..35 41 43..44 46..49 51 53 55 57 59..61 67..70 72
75..77 79..81 86..87 91..94 96..99 527 544 548 550 559 563 577 579 583..584 589 591 597
600 604 612 615 618}",
  :ok
  "#{0..5 7..11 13 15..20 22..24 26..28 30 32..35 37 39..44 46..49 51..55 57..61 63..70
72..77 79..82 84..89 91..94 96..99 101..108 110..131 133..154 156..176 178..200
202..221 223..243 245..267 269..289 291..313 315..337 339..359 361..382 384..405
407..427 429..451 453..475 477..497 499..521 523..527 529..531 533 537..539 544..545
548 550 552..553 556 559 563 566 572 574..575 577..579 583..584 587..591 596..597 600
602 604 607 610 612..613 615 617..618 621 623 625 627..628 632 635..637}",
  :total 638,
  :recovered-count 66,
  :unexpected-count 0,
  :unexpected "#{}",
  :lost-count 8,
  :ok-count 541}}
```

This test lost eight acknowledged writes, all clustered around write number `630`, just prior to the end of the second partition. It also **wedges the cluster hard**–even though cluster status reports green, some nodes refuse to converge. All kinds of weird messages in the logs, including an enigmatic `reason [do not exists on master, act as master failure]`. In order to get any data back from the final read in these tests, I had to manually restart the stuck nodes during the end-of-test waiting period. Anecdotal reports suggest that this is not merely a theoretical problem; Elasticsearch convergence can deadlock in production environments as well.


## One single-node partition

Perhaps isolating nodes more than once is too difficult to handle. Let's make things easier by only having one partition occur. We'll cut off a single primary node once, wait for the cluster to converge, let it heal, wait for convergence again, and do a read. This is the simplest kind of network failure I know how to simulate.

```
                                (gen/nemesis
                                  (gen/seq
                                    [(gen/sleep 30)
                                     {:type :info :f :start}
                                     (gen/sleep 200)
                                     {:type :info :f :stop}]))
```

This time the cluster manages to recover on its own, but it still lost inserted documents.

```
 {:valid? false,
  :lost
  "#{687..689 692..693 700 709 711 717..718 728..730 739 742 755 757 764 766 771 777
780 783 785 797 800 816..817 820 841 852}",
  :recovered
  "#{140..142 147 151..153 161 163 166 168..169 172 175 178..179 182..185 187..192 197
200 203..207 209..210 212..213 215..216 218 220..221 223..225 229 231..232 235 237
239..240 242 323 368 393 484 506 530 552 577 598 623 644 714 720 727 746 758 760 762
772 774 790..791 802 806 836 851}",
  :ok
  "#{0..142 144..145 147..149 151..154 156..163 165..166 168..173 175 177..179 181..185
187..213 215..226 228..242 244..253 255..276 278..301 303..346 348..414 416..437
```

```
439..460 462..666 668..685 694..699 701..707 710 712..716 719..727 731..738 740..741
744..748 750..754 756 758..763 765 767..770 772..775 778..779 781..782 784 786..787
789..796 798..799 801..815 818 821..822 824..826 828..840 842..851 853..854}",
  :total 855,
  :recovered-count 79,
  :unexpected-count 0,
  :unexpected "#{}",
  :lost-count 31,
  :ok-count 792}}
```

31 acknowledged writes lost, out of 855 attempts. Or, if you're unlucky, it might throw away almost all your data.

```
 {:valid? false,
  :lost
  "#{0..1 4..6 8..11 14 17..18 20..23 25..26 28 30..32 35 38..42 44 46..49 51..52
54..56 58 60 62 64..66 68 70 72..73 77..78 80 84..85 87 89 91..92 94 96 98..100
105..110 112..113 118..123 125 128..129 131 134..137 139 144 147 154..158 160..162
166..167 172..174 180 182..183 186 190..192 196 200 202 207..208 221 226..228 230..233
235..237 239 244..256 258..277 279..301 303..323 325..346 348..368 370..390 392..413
415..436 438..460 462..482 484..506 508..528 530..552 554..574 576..598 600..619}",
  :recovered "#{}",
  :ok
  "#{2..3 7 12..13 15..16 19 24 27 29 33..34 36..37 43 45 50 53 57 59 61 63 67 69 71
74..76 79 81..83 86 88 90 93 95 97 101..104 111 114..117 124 126..127 130 132..133}",
  :total 620,
  :recovered-count 0,
  :unexpected-count 0,
  :unexpected "#{}",
  :lost-count 484,
  :ok-count 54}}
```

In this run, **a single network partition isolating a primary node caused the loss of over 90% of acknowledged writes.** Of 619 documents inserted, 538 returned successful, but only 54–10%–of those documents appeared in the final read. The other 484 were silently discarded.


Remember, this is the one kind of network failure Elasticsearch was designed to withstand. Note that the write loss pattern varies with time; we lost a mixture of writes through the first quarter of the test, and recovered nothing written after 133.

This test is a little harder to reproduce; sometimes it'll recover all writes, and occasionally it'll drop a ton of data. Other times it just loses a single write. I thought this might be an issue with flush not actually flushing the transaction log, so I upped the delay before read to 200 seconds–and still found lost data. There may be multiple bugs at play here.

## Recommendations for Elasticsearch

To summarize: Elasticsearch appears to lose writes–both updates and even non-conflicting inserts–during asymmetric partitions, symmetric partitions, overlapping partitions, disjoint partitions, and even partitions which only isolate a single node once. Its convergence times are slow and the cluster can repeatably deadlock, forcing an administrator to intervene before recovery.

I wish I had a better explanation for these problems, but I've already burned a hundred-odd hours on Elasticsearch and need to move on to other projects. I've got a few conjectures, though.

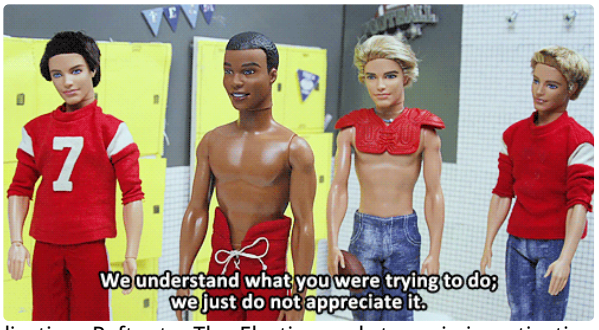
Elasticsearch's cluster membership protocol is a mess, and Elasticsearch opts against using an external coordination service like Zookeeper for membership on the grounds that it has access to less information about the cluster's ongoing operations. Elasticsearch has started to put work into their fault-tolerance mechanisms, but that work is still incomplete.

The thing is that leader election is a well-studied problem with literally dozens of formally described algorithms and none of that literature appears to have influenced Zendisco's design. Leader election is not sufficient to ensure correctness, but using a peer-reviewed algorithm would be a heck of a lot better than the current morass of deadlocks and persistent split-brain.

Despite what the docs claim, I'm not convinced that Elasticsearch actually does synchronous replication correctly. It

looks to me–and I haven't verified this with a packet trace yet, so this is a bit shaky–that primary nodes can acknowledge writes made even when they're isolated from the rest of the cluster. The patterns of write loss in the Jepsen histories are pretty darn suggestive. I'd expect to see a clear-cut pattern of timeouts and failures when a partition starts, and no successful writes until a new primary is elected. Instead, Elasticsearch continues to acknowledge writes just after a partition occurs. It might be worth investigating that write path to make sure it won't ack a write until a majority confirm.

Speaking of majority, Shikhar Bhushan notes that Elasticsearch considers a quorum for two nodes to be a single node. This definition invalidates, well, basically every proof about non-dominating coterie consensus, but the Elasticsearch team doesn't seem keen on changing it. I don't think this affects the tests in Jepsen, which maintain three or five replicas, but it's worth keeping in mind.



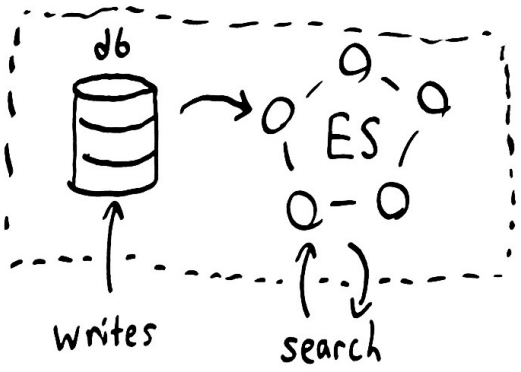

None of these changes would be sufficient for linearizability, incidentally. Distributed commit requires a real consensus protocol like Paxos, ZAB, Viewstamped Replication, Raft, etc. The Elasticsearch team is investigating these options; from unofficial murmurings, they may release something soon.

In the short term, Elasticsearch could provide better safety constraints by treating a collection of documents as a CRDT–specifically, an LWW-element set where the timestamps are document versions. This isn't linearizable, but it would allow Elasticsearch to trivially recover inserts made on both sides of a cluster, and to preserve writes made on only one side of a partition. Both of these behaviors would significantly improve Elasticsearch's fault tolerance, and is significantly simpler to implement than, say, Raft or Paxos.

## Recommendations for users

If you are an Elasticsearch user (as I am): good luck.

Some people actually advocate using Elasticsearch as a primary data store; I think this is somewhat less than advisable at present. If you can, store your data in a safer database, and feed it into Elasticsearch gradually. Have processes in place that continually traverse the system of record, so you can recover from ES data loss automatically.




Folks will tell you to set `minimum_master_nodes` to a majority of the cluster, and this is a good idea–but it won't, by any means, buy you safety. I'm honestly not sure what will. Raising the replica count reduces the odds of losing inserts, but means more replicas to conflict during CAS and won't guarantee safety even when a replica is present on *every single node in the cluster*.

The good news is that Elasticsearch is a *search engine*, and you can often afford the loss of search results for a while. Consider tolerating data loss; Elasticsearch may still be the best fit for your problem.

Moreover, none of this invalidates the excellent tooling available around Elasticsearch. People love Logstash and Kibana. The ES API is surprisingly straightforward, and being able to just spew JSON at a search system is *great*. Elasticsearch does a great job of making distributed search user-friendly. With a little work on correctness, they can make it safe, too.

This concludes the third round of Jepsen tests. I'd like to thank Comcast for funding this research, and to Shay Banon, Bob Poekert, Shikhar Bhushan, and Aaron France for their help in writing these tests and understanding Elasticsearch's behavior. I am indebted to Camille Fournier, Cliff Moon, Coda Hale, JD Maturen, Ryan Zezeski, Jared Morrow, Kelly Sommers, Blake Mizerany, and Jeff Hodges for their feedback on both the conference talks and these posts, and to everyone who reached out with reports of production failures, paper recommendations, and so much more.

And finally, thank you, reader, for sticking with what was probably far more than you ever wanted to know about databases and consistency. I hope you can use the techniques from these posts to help verify the safety of your own distributed systems. Happy hunting.


shikhar, on 2014/06/19

I've been jepsen-testing ES using my discovery plugin (http://github.com/shikhar/eskka) and I tend to see much fewer writes lost (0-4 typically, using different partitioning strategies), which I attribute to it avoiding split-brains with multiple masters that Zen is currently prone to. But of course there should really be no acked writes lost.

> It might be worth investigating that write path to make sure it won't ack a write until a majority confirm.

Agreed! TransportShardReplicationOperationAction where this stuff is happening, goes like this, in case you are running with write consistency level quorum + sync writes:

check quorum perform the write on primary perform the sync write on replicas

Seems like this should work if the same cluster state is used throughout and it actually fails hard on each step. However from what I see there is a bunch of logic in performReplicas() (https://github.com/elasticsearch/elasticsearch/blob/a06fd46a72193a387024b00e226241511a3851d0/src/main/java/c L676) where it decides to take into account updated cluster state, and there seem to be exceptions for certain kinds of failures being tolerated.

I think this would be so much more straightforward if a write were to be fanned-out and then block uptil max of timeout for checking that the requried number of replicas succeeded (with success on primary being required).

shikhar, on 2014/06/19

seems like the ES team is moving in the right direction with testing this stuff

Anders Hovmöller, on 2014/06/19

"reduces the odd of losing" <- should be "odds".

Great article! Scary. But great :P

Jon, on 2014/06/19

Animated images on articles = NO

ss, on 2014/06/19

I like Jon :)

Nat, on 2014/06/19

Awesome post!(but hate those annoying gifs, a big distraction while reading this blog, please avoid them in the future)

Alex, on 2014/06/19

I love the animated gifs. That other guy just needs some epilepsy meds.

Levi Figueira, on 2014/06/19

I came here for the Barbie GIFs.

I like. :)

Chris, on 2014/06/19

Just wanted to say that this series has been an exceptional contribution to discourse around distributed systems; thanks for all you've done so far, it's been amazing reading.

Jerome, on 2014/06/19

Thanks a lot for this post (and all the others). Now how do we convince you to write a similar one for Solr? It would be really insightful to compare the two leading distributed search systems.

Jonathan Channon, on 2014/06/20

Is RavenDB next on the list?

Mo, on 2014/06/20

Very nice article. Thank you for the effort spent.

Jason, on 2014/06/20

Excellent article - thanks for writing these.

Radu Gheorghe, on 2014/06/20

If you want to reduce the time for failure detection, you'd need to decrease discovery.zen.fd.ping_timeout and ping_retries instead of discovery.zen.ping_timeout: http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/modules-discovery-zen.html#fault-detection

The flip side to this (Captain Obvious to the resq!) is that false positives may occur during small glitches like stop-of-the-world GC.

Jason Kolb, on 2014/06/20

Thanks for writing this up. We're using Elasticsearch and I've raised these issues to several people there. Hopefully they'll publish a response soon.

On another note, holy f*&k are those gif's annoying.

Xiao Yu, on 2014/06/20

Thanks for the post, just wanted to note one thing.

It looks like you may have been using the wrong setting for ping timeouts, the correct setting appears to be `discovery.zen.ping_timeout` (note the underscore not dot between ping and timeout) as it appears in the docs and code. It appears the default config file distributed with elasticsearch is wrong.

Eric, on 2014/06/20

Great article, expertly augmented by Most Popular Girls in School.

State! State! State! State!

mbdas, on 2014/06/20

Is it possible to evaluate against the zookeeper discovery plugin. I think this fork has more upto date compatibility. https://github.com/imotov/elasticsearch-zookeeper I am seen lots of remarks mentioning zk plugin solves a lot of issues including split brain. And it is not

officially endorsed by ES team I think.

Swen Thümmler, on 2014/06/24

It would be interesting to see the testresult with the elasticsearch-zookeeper plugin. I have made a version compatible with ES 1.2 and zookeeper 3.4.6 (https://github.com/grmblfrz/elasticsearch-zookeeper/releases/download/v1.2.0/elasticsearch-zookeeper-1.2.0.zip). I'm using this because of repeated split brains with multiple masters despite setting discovery.zen.minimum_master_nodes to n/2 + 1. Now the split brain conditions are gone…

Philip O'Toole, on 2014/06/24

Interesting article – thanks. Interestingly, I have very recently heard about a split-brained correctly-configured, ES cluster. I don't know much else, but I know for a fact the cluster was following the advice in my blog post, and was running a recent release.

I have some feedback and questions.

You mention early on "…those which are read-only copies–termed "data nodes"–but this is primarily a performance optimization". I've never heard of read-only nodes. Replicas, yes, which can serve search traffic, not but not index. Is this what you mean? Also I understand data-nodes to mean any node that has shards – in contrast to node-clients, which just serve to route indexing and search requests.

Also in parts of the article you refer to "primary", when I think you mean "master". For example, you mention the election of a "primary". In the context of ES, "primary" usually refers to a primary copy of a replicated shard, as opposed to the replica of the shard.

## Post a Comment

Name

Email

Http

Supports github-flavored markdown for [links](http://foo.com/), *emphasis*, _underline_, `code`, and > blockquotes. Use ```clj on its own line to start a Clojure code block, and ``` to end the block.

Post Comment