Cassandra Availability with Virtual Nodes

Joseph Lynch
joe.e.lynch@gmail.com
josephl@netflix.com

Josh Snyder
josh@code406.com
joshs@netflix.com

April 16, 2018

1 Introduction

When Cassandra introduced vnodes in version 1.2, the database made new trade-offs with respect to cluster maintainability and availability [1] [2]. In particular, large Cassandra clusters became easier to scale and operate, but they lost availability during temporary or permanent node failures.

The two main benefits of vnodes are that the variance in data size held by any one host decreases, and that operators can more easily change the number of hosts participating in a cluster. In return, vnodes trade-off availability during node failures. As we document in this report, the availability cost of vnodes is both quantifiable and unavoidable. We present a model of the availability of a single-datacenter Cassandra cluster under failures. We attempt furthermore to predict, for a given cluster configuration, what availability guarantees can feasibly be promised to users of the database.

Based on our model, we conclude that the Cassandra default of 256 virtual nodes per physical host is unwise, as it results in steadily rising chance of unavailability as cluster size grows. This increasing failure rate is antithetical to Cassandra's design goal of fault-tolerance. We further use the model to argue for a number of possible solutions which both achieve operability goals of vnodes while at the same time providing much higher fault-tolerance, especially for large clusters (hundreds of nodes).

Section 2 introduces basics of the Cassandra data placement algorithm, and Section 3 discusses situations that can degrade Cassandra service. Readers already familiar with Cassandra operations may wish to skip to Analysis (Section 4) or Solutions (Section 4.3).

2 Cassandra Background

Data in Cassandra is distributed across the cluster hosts according to a consistent hashing algorithm. All data has a partition key, which Cassandra hashes to yield an integer bounded within the hash space. Each host in the cluster takes responsibility for storing data along one or more contiguous ranges of the hash space as decided by the replication strategies assigned to the keyspaces on the cluster.

Cassandra clusters achieve fault-tolerance in the `SimpleStrategy` and `NetworkTopologyStrategy` replication strategies by storing multiple copies of each piece of data. The quantity of copies (replication factor) is configurable, but is typically an odd number, with three being most common. Higher replication factors achieve greater fault-tolerance, at the cost of increased storage requirements.

Hosts in a Cassandra cluster are assigned one or more tokens which represent positions in a hash space created by the hashing function. Having more than one token per physical host leads to "virtual" nodes. The start of each range is variable, depending on the location of tokens held by other nodes in the cluster. The intervals in the hash space between tokens are known as *token ranges* and are the basis of all data replication. Depending on the keyspace replication strategy, and the placement of the "virtual" nodes, different physical hosts will be chosen to store data in a token range. For example, to achieve a replication factor of two using `SimpleStrategy`, Cassandra assigns two different physical hosts to each token range, while with `NetworkTopologyStrategy` Cassandra adds additional constraints to assure that data is replicated between racks and across datacenters.

3 Problem

In Cassandra's data placement strategy with a replication factor of three, availability is compromised when at least two hosts holding intersecting token ranges become unavailable to serve traffic. Practically speaking, unavailability results when a second node fails so quickly that the first hasn't had time to recover. Prior attempts have been made to model Cassandra's failure modes [4], but they often do not take into account the recovery aspect of Cassandra where it automatically restarts after process failure or in the case of permanent failure streams lost data to new hosts [1]. We approach the problem from a different direction, opting instead to model outages as a combination of losing a single node plus losing a neighboring node that owns an intersecting token range before recovery finishes, causing `UnavailableExceptions` to clients operating

[1]Not exactly automatically, but with the right automation it can be automatic

at `QUORUM` consistency levels. We consider any unavailability of the ring to be unacceptable to customers, whether it is of a large portion of the ring or small.

4 Analysis

We conduct our analysis on a cluster with replication factor 3 ($R = 3$). Under $R = 3$, the failure of two nodes with a shared token range leaves their shared token range(s) with only a single active node. The remaining node can continue to serve reads and writes, but *cannot* produce a quorum.

Without virtual nodes, a host in an $RF = 3$ cluster would inter-depend on 4 other hosts: its two left-neighbors and two right-neighbors. If the cluster is small enough, the same node may serve as both a left-neighbor and a right-neighbor. For example on a cluster with three hosts, each node inter-depends with two others; with four hosts each node has three inter-dependents; and with five or more hosts each node has four inter-dependents. When a brand new host places 256 virtual nodes into the ring, each vnode it places inter-depends with 2-4 other nodes.

By adding multiple token ranges to the responsibility of a single server, virtual nodes increase inter-dependency and decrease availability by coupling physical hosts together. Conversely, they theoretically increase the speed at which nodes can recover data by streaming[2].

Whenever it was necessary to build assumptions into the model, we chose them to make the model underestimate outages.

For the purposes of this analysis:

let:	n	= number of hosts in the cluster
	λ	= average failure rate of hosts (1/interval)
	S	= size of the dataset per host (MB)
	v	= number of tokens per physical host
	R	= replication factor (number of replicas per token)
	B_{inc}	= maximum incoming bandwidth of a joining physical host (MB/s)
	B_{out}	= maximum outgoing bandwidth of a stream (MB/s)

n and S are details of a particular Cassandra deployment, v and B_{out} are configuration options in `cassandra.yaml`, R is determined by the keyspace replication strategy and λ and B_{inc} are determined by the hardware Cassandra is deployed to (B_{inc} is also impacted by how efficiently Cassandra streams data, which we take into account with an arbitrary adjustment in the model).

[2]In our experience, at least with Cassandra 2.x and 3.0.x, inbound streaming tops out at significantly under network hardware capacity, but this simply makes this model underestimate outages

4.1 Availability Model

Crucial to the availability calculation is how quickly nodes recover: $E[T_{recovery}]$. The longer a host takes to recover, the longer a second failure can occur and cause an outage.

We model the recovery time of a host failure in Eq. 1 by taking the dataset S and dividing it by the rate at which a recovering node can stream data from peers. To determine the speed, we must determine how many neighbors the node can stream from, which we model as a variant of the Birthday Problem [6]. For each token, we choose a random $2 * (R - 1)$ replica hosts (1b), those that precede and those that follow the token on the Cassandra ring. This is an underestimate of how many neighbors each additional token adds because actual Cassandra replication prohibits duplicates [5]. We then make the model underestimate outages further by assuming rack awareness, and that the number of racks equals R, which eliminates $\frac{n}{R}$ nodes from being possible replicas (1c). Finally we can determine the number of expected neighbors, $E[n_{neighbors}]$ by applying the formula for the number of distinct items expected from k draws of n_p items in 1d. To find the final recovery time we assume that the node is being replaced with `replace_address` rather than with `removenode` [3], so we only consider the incoming bandwidth of a single host (1e). This leads to 1f and concludes the recovery model.

$$E[T_{recovery}] = \frac{S}{speed} \tag{1a}$$

$$k = v * (2 * (R - 1)) \tag{1b}$$

$$n_p = n - \frac{n}{R} \tag{1c}$$

$$E[n_{neighbors}] = n_p * [1 - (1 - \frac{1}{n_p})^k] \tag{1d}$$

$$speed = \min(B_{inc}, E[n_{neighbors}] * B_{out}) \tag{1e}$$

$$E[T_{recovery}] = \frac{S}{\min(B_{inc}, E[n_{neighbors}] * B_{out})} \tag{1f}$$

This is a very conservative model, meaning that in reality vnodes would likely have a higher impact (e.g. because more nodes can be neighbors and may not have racks=R), but for cloud environments like AWS where Cassandra racks generally map 1:1 with availability zones this kind of model is reasonably accurate. This model gives us what we dub the "critical window" where any neighbor failing will cause a token range to lose quorum.

From Eq. 1 we can see that as v increases, we get faster streaming until we

[3]See Section 4.3.3 for a discussion of why we think this replacement model is more appropriate for production

reach diminishing returns because we have saturated the inbound network of the joining node. At that point additional streams do not help speed recovery.

Now that we can model the expected recovery time, we begin the failure model by assuming that hosts fail as independent Poisson processes with parameter λ. Note that modeling host failures as a Poisson process represents a "best case scenario" for cluster availability: real-life computers are not so polite that they fail in a manner completely uncorrelated with their peers.

Under such a model the inter-arrival time of host failures follows an exponential distribution with parameter λ, and therefore once a single host failure happens, we lose a second replica of an overlapping range before the recovery finishes with probability:

$$
\begin{aligned}
P(outage|failure) &= F(\tau; \lambda_{eff}) && \text{(2a)}\\
P(outage|failure) &= 1 - e^{-\lambda_{eff} * \tau} && \text{(2b)}\\
\tau &= E[T_{recovery}] && \text{(2c)}\\
\lambda_{eff} &= E[n_{neighbors}] * \lambda && \text{(2d)}\\
P(outage|failure) &= 1 - e^{-E[T_{recovery}] * E[n_{neighbors}] * \lambda} && \text{(2e)}
\end{aligned}
$$

Equation 2b follows from the CDF of an exponential distribution with parameter λ_{eff}, and equation 2c and 2d follow because we only care about failures during the recovery period of the neighboring hosts (which form a combined Poisson process with effective parameter λ_{eff}).

Now, if we model each host as a splitting Poisson process with parameter λ and splitting probability $P(outage|failure)$, then each node forms an outage Poisson process with parameter $\lambda_{split} = \lambda * P(outage|failure)$. If we then join the n such processes, we have a resulting global Poisson process with parameter $\lambda_{global} = \lambda_{split} * n$. This yields Eq. 3, which models the average number of outages over an interval τ.

$$
\boxed{
\begin{aligned}
E[outages] &= \lambda_{global}\\
&= (n * \lambda_{split})\\
&= (n * (\lambda * P(outage|failure)))\\
&= (n * (\lambda * (1 - e^{-E[T_{recovery}] * E[n_{neighbors}] * \lambda_s})))
\end{aligned}
} \qquad (3)
$$

From this model we can see that availability is compromised by increasing the number of nodes (n), increasing the failure rate of those nodes (λ), increasing the number of neighbors (via v), or decreasing the recovery speed. To see how badly availability reacts with plausible default settings on a 96 node cluster in a cloud environment with machine failure rate $\lambda = \frac{25}{century} = \frac{0.25}{year}$.

let: $n = 96$
$\lambda = 25 \frac{1}{century}$ $(= 0.25 \frac{1}{year} = 7.9\text{e-}9 \frac{1}{s})$
$v = 256$
$R = 3$
$S = 307{,}200$ (MB, 300 gigabyte)
$B_{inc} = 125$ (MB/s, 1 gigabit)
$B_{out} = 25 / 2 = 12$ (MB/s, 100 megabit)

We model this over a century because the probabilities of outage are so low in a single cluster that in any reasonable time span, the most likely number of outages is zero. To fairly compare the different configuration options, we model this system for a century and find that, on average, we expect 2.98 outages, or equivalently 0.03 per year:

$$\begin{aligned} A &= 96 * 25 * (1 - e^{-E[T_{recovery}]*E[n_{neighbors}]*\lambda_s}) \\ &= 96 * 25 * (1 - e^{-2457*64*7.927e-9}) \\ &= 2.98 \end{aligned} \tag{4}$$

In the case with only 4 vnodes, the availability is much better with only 0.35 failures per century = 0.0035 failures per year, a full 10x more availability!

$$\begin{aligned} A &= 96 * 25 * (1 - e^{-E[T_{recovery}]*E[n_{neighbors}]*\lambda_s} \\ &= 96 * 25 * (1 - e^{-2457*7.5*7.927e-9}) \\ &= 0.35 \end{aligned} \tag{5}$$

We can clearly see in Fig.1 that small numbers of vnodes do not appreciably change availability (due to the streaming benefits), but as the number of vnodes gets large we rapidly lose large amounts of availability until we level out after the number of vnodes passes the number of nodes. In fact, for this particular λ, the median number of outages is still zero until $v = 5$.

In Fig.2 we see the effects of varying the failure rate of machines, and observe that this has a significant impact on the availability of a Cassandra cluster running with vnodes. In particular, doubling the rate of machine failures can more than double the expected number of outages. This makes intuitive sense because as we add more tokens to each physical node, we are increasing the number of vulnerable replicas during a single node failure. Rack placement helps a lot to limit the number of potential replicas to only those residing in other racks, but it is not enough to prevent likely outage with a high number of tokens per node.

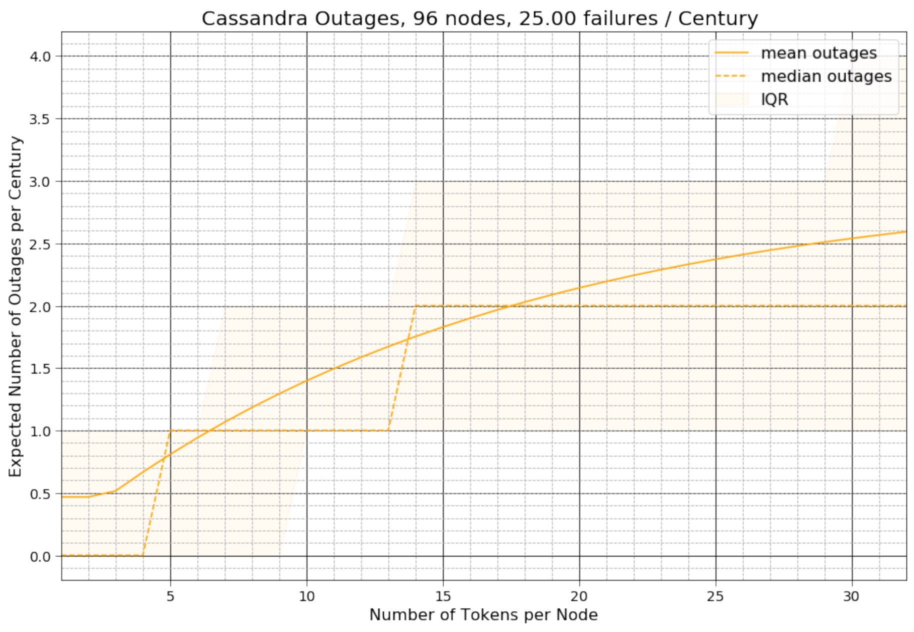

Figure 1: Outages with Small Number of Tokens per Node

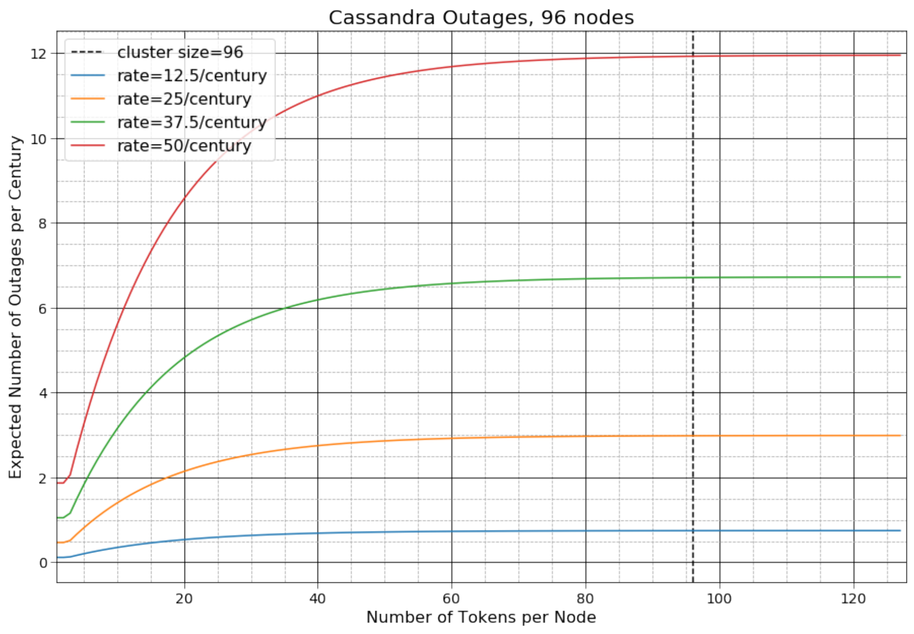

Figure 2: Outages with Varying Tokens per Node

It is also interesting to note that as clusters get larger, holding everything else constant, they become *less available*. This intuitively makes sense because as you increase the number of hosts significantly past RF, you are creating more replicas that can fail and be vulnerable to a second failure. Virtual nodes *significantly exacerbate* the problems for large clusters ($n > 200$) causing substantially more outages at large cluster sizes. This is shown in figure Fig. 3 and Fig. 4 . Fig 3 is particularly interesting because we can see with 256 vnodes, we rapidly approach an expected failure every year, compared to 16 vnodes where we expect an outage closer to every decade.

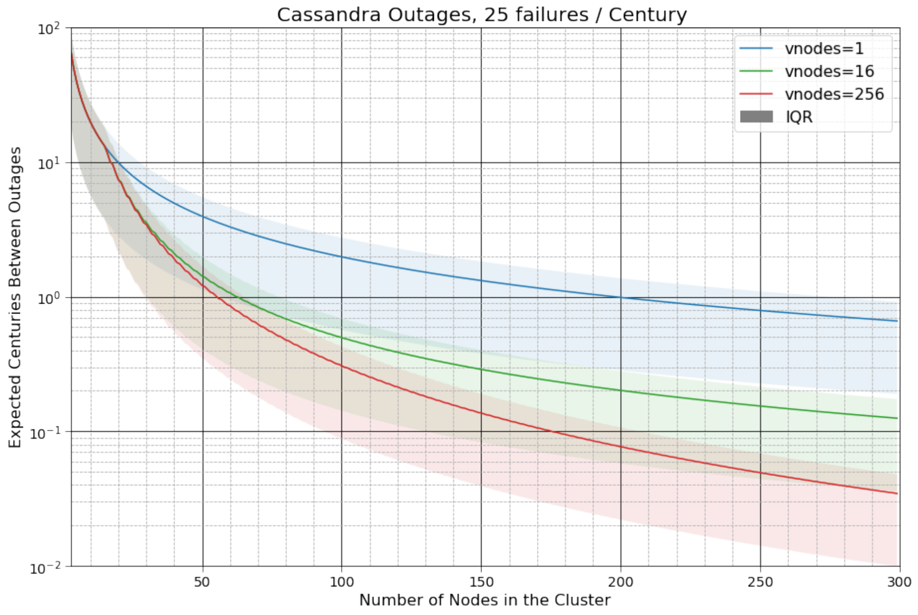

Figure 3: MTBF with Varying Nodes in the Cluster

However, as can be seen in Fig. 5, small clusters ($\leq$ 16 for `NetworkTopologyStrategy` or $\leq$ 12 for `SimpleStrategy`) are not appreciably impacted by vnodes because all their hosts are already neighbors with every possible host. As the cluster gets bigger, small clusters have a constant number of neighbors, but large clusters gain more and more neighbors, leading to 10-20x more risk of failure.

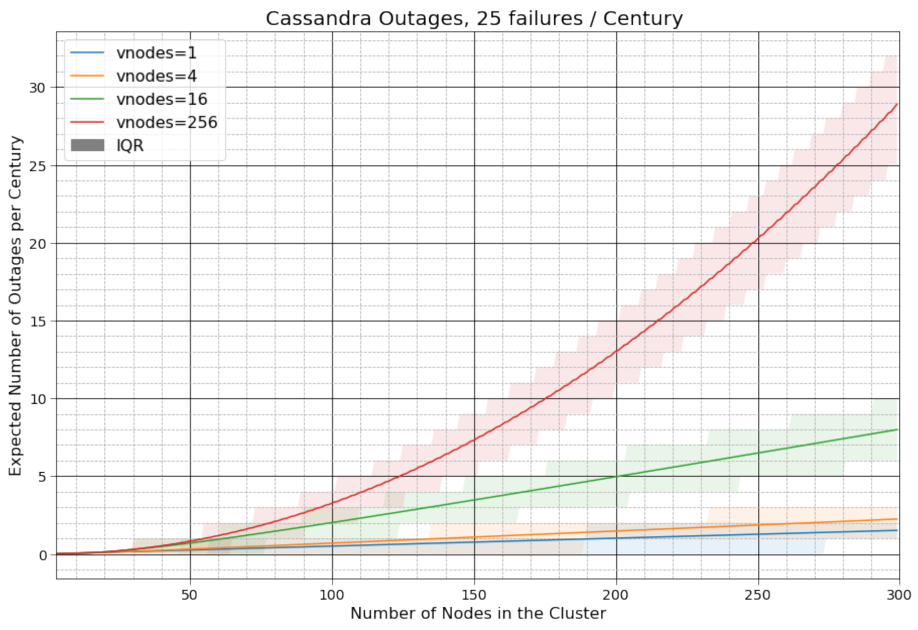

Figure 4: Outages with Varying Nodes in the Cluster

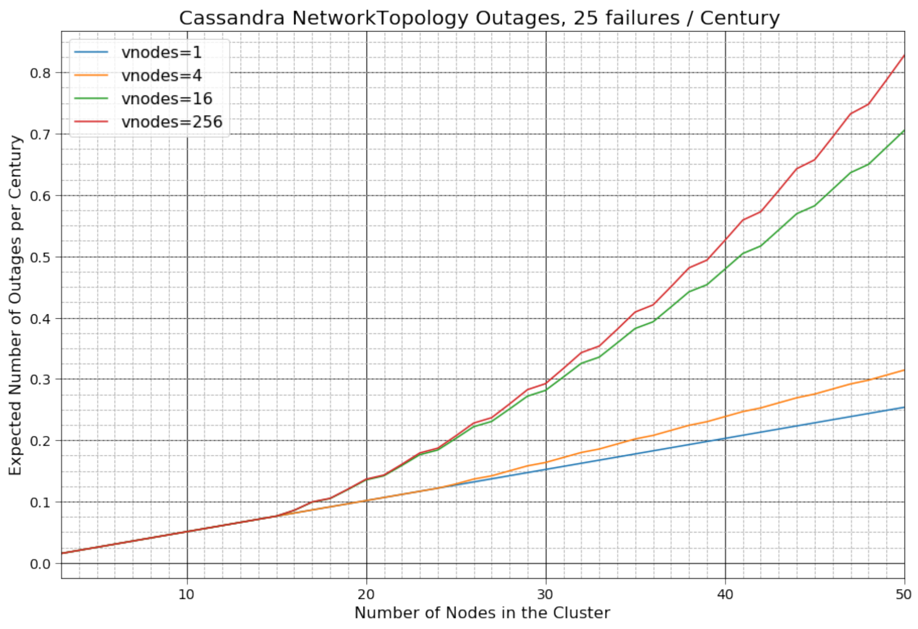

Figure 5: Outages with Varying Nodes in the Cluster

The odd wiggle pattern you can see happens because rack awareness removes n/R hosts from consideration; the `SimpleStrategy` graph is more smooth but also much less available in Fig 6.

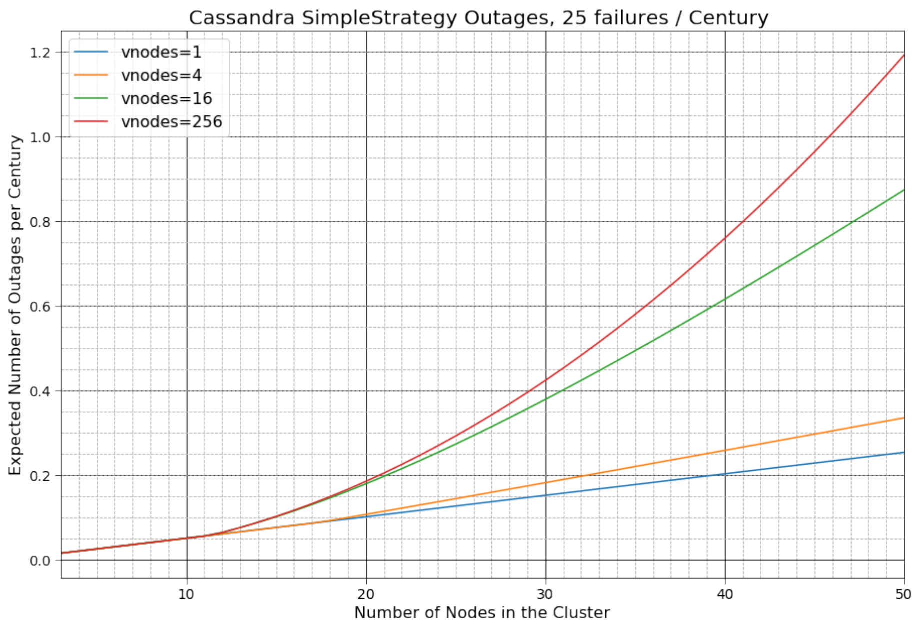

Figure 6: Outages with Varying Nodes in the Cluster

Clearly, while 4 or 16 virtual nodes do not appreciably impact availability, the default of 256 is quite unacceptable for large clusters.

4.2 Virtual Nodes, the Benefits

Having a number of tokens greater than one per physical host trades off availability, but it gains data distribution and operations benefits. In particular, a major problem pre-vnode was “hot” nodes, where one replica may have significantly more data than other nodes. This was particularly problematic when scaling up the cluster because in order to keep data evenly distributed you had to effectively double the cluster. In particular to add capacity to a cluster of size n with v vnodes you need proportionally fewer new nodes N to do so. Now that Cassandra token placement is reasonably good at inserting new tokens where they are needed to balance the ring [9], we can model the number of nodes needed to scale up with Eq. 6 as seen in Fig. 7.

$$N = \left\lceil \frac{n}{v} \right\rceil \tag{6}$$

As we can clearly see, as we add more vnodes we must add fewer physical hosts to the cluster at a time in order to keep it balanced. While 1 token per host requires doubling, 16 tokens per host reduces this, naturally, by a factor of 16, which is a *significant* improvement. For a 96 host cluster instead of needing 96 more machines, we can add in increments of 6. The operational savings only get better as we add more tokens per host, until we reach the size of the cluster at which point we reach diminishing returns.

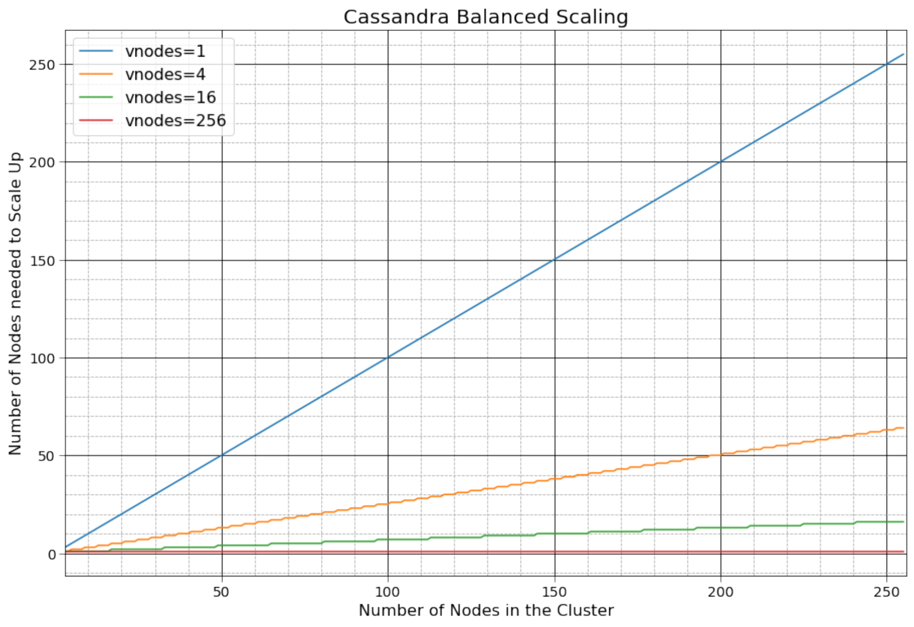

Figure 7: Nodes Required for Scaling Up

4.3 Solutions to the Availability Problem

There are a number of solutions to the availability problem caused by vnodes with various trade-offs. We briefly discuss some here:

4.3.1 Solution 1: Change the Number of vnodes

A simple solution to the availability problem is to make the trade-off less drastic. Moving to 4-16 tokens per node achieves almost all of the benefits of vnodes, while trading off significantly less availability. For example a setting of 16 gives 10x better availability than 256 while still having 10x easier scaling than a single token [10].

4.3.2 Solution 2: Decouple Data Transfer from Recovery

An elegant solution to this problem is to use networked attached storage such as EBS [11]. With a high reliability, low latency, network block device like AWS EBS `gp2`, Eq. 1 becomes a fixed constant of roughly five minutes. This increases availability by multiple orders of magnitude as replacement no longer varies with data size. Multiple tokens per host still reduce availability (since neighbors is still higher), but it is much less significant. Figure 8 shows just how much availability clusters can get by using EBS to store their data. The gains are even higher for large datasets (this model only has 300GB, if it were 2TB the gains would be significantly higher).

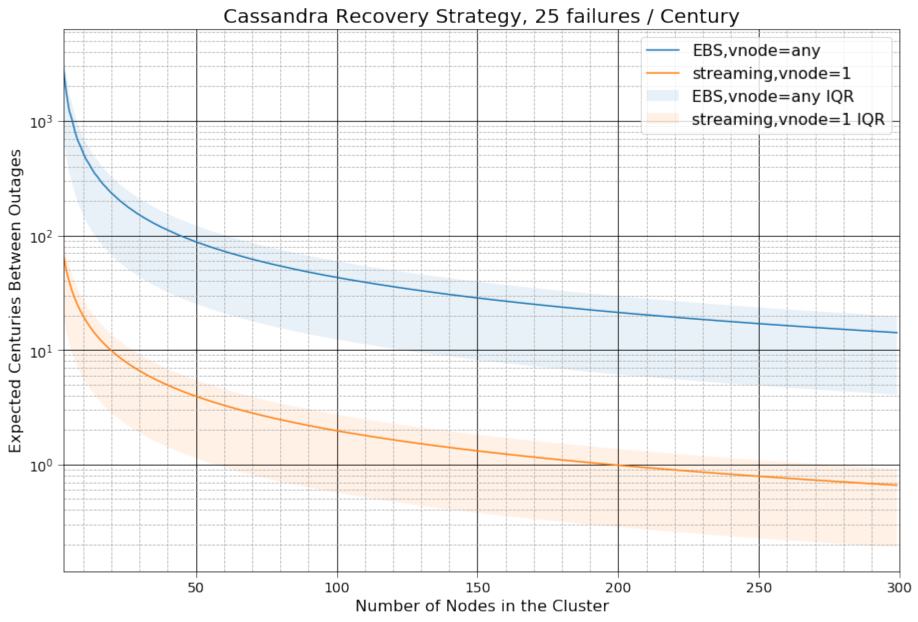

Figure 8: Availability Under Differing Recovery Strategies

4.3.3 Solution 3: Replacing Hardware via `removenode`

There is an alternative streaming model where instead of replacing a failed node with `replace_address`, the operator runs `removenode` on the dead node. This recovers the cluster to full replication rapidly as all nodes are participating in the re-replication of data, but simply with one fewer node present [8]. The subsequent bootstrap may take a long time, but that time is not spent at risk of outage. This technique is strictly dominated by `replace_node` for single token clusters because the data must be streamed twice, but for multiple token per node clusters it can make the availability much higher because `removenode` theoretically involves almost every node in the cluster transferring data between themselves, which decreases $E[T_{recovery}]$ in Eq. 3 and limits outage.

This approach is likely unwise in production, however, as unlike `replace_address`, `removenode` causes operational load on nodes that are simultaneously doing latency sensitive query workloads. In particular `removenode` causes the following unique impacts (versus `replace_node`) to hosts taking latency sensitive queries:

1. Inbound network traffic
2. Disk IO and CPU cycles to write the data to disk (and later clean it up)
3. Loss of OS disk cache for data that will be invalidated shortly

Inbound network traffic is particularly problematic because unlike outbound network traffic which can be effectively be controlled via QoS [12], inbound network traffic is very difficult to insulate query traffic from, and would likely cause operational impact. In contrast, with `replace_address` the cluster operator can use `tc-fq` insulate query traffic from the streaming flows [13]

This approach would require significant production vetting before we believe it is a viable alternative to the production hardened `replace_node` approach.

4.3.4 Solution 4: Stop Using vnodes Entirely

An alternative way to solve the operational pain of large Cassandra clusters, while maintaining availability, is to use `moves` rather than tokens to re-balance data in a cluster [14][15]. This could either happen through a central planner which determines the optimal state of the cluster, or some kind of distributed greedy algorithm. Now that Cassandra has good streaming via the Netty refactor, this appears to be a very plausible approach and has additional benefits in repair and scaling (simultaneous bootstrap).

5 Conclusion

We have shown using a reasonable model that using a number of tokens per node greater than the number of physical nodes in the cluster significantly increases the expected outages, with default settings causing orders of magnitude more frequent outages. This availability is traded away for more even distribution of data and the ability to add or remove fewer nodes and still maintain even distribution. We have proposed a number of possible solutions, but we believe the current `cassandra.yaml` setting of 256 tokens per node has significantly decreased Cassandra cluster availability, especially at large cluster sizes, and are interested in improving the situation. Regardless of how the Cassandra community chooses to solve this problem, we believe it is worth understanding how vnodes have changed the math.

References

[1] Brandon Williams: Virtual nodes in Cassandra 1.2 https://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2

[2] Sam Overton: RFC: Cassandra Virtual Nodes https://www.mail-archive.com/dev@cassandra.apache.org/msg03837.html

[3] Karger, et. al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web https://dl.acm.org/citation.cfm?id=258660

[4] Martin Kleppmann: The probability of data loss in large clusters https://martin.kleppmann.com/2017/01/26/data-loss-in-large-clusters.html

[5] Cassandra Data replication, NetworkAwareTopologyStrategy https://docs.datastax.com/en/cassandra/latest/cassandra/architecture/archDataDistributeReplication.html

[6] Kyle Siegrist: The Birthday Problem http://www.randomservices.org/random/urn/Birthday.html

[7] Apache Software Foundation: Replacing a Dead Node http://cassandra.apache.org/doc/latest/tools/nodetool/removenode.html

[8] Apache Software Foundation: nodetool removenode http://cassandra.apache.org/doc/latest/tools/nodetool/removenode.html

[9] Branimir Lambov: New token allocation algorithm in Cassandra 3.0 https://www.datastax.com/dev/blog/token-allocation-algorithm

[10] Chris Lohfink: Lower default num_tokens https://issues.apache.org/jira/browse/CASSANDRA-13701

[11] Jim Plush and Dennis Opacki: Amazon EBS & Cassandra https://www.youtube.com/watch?v=1R-mgOcOSd4

[12] Bert Hubert et al.: Linux Advanced Routing & Traffic Control http://lartc.org/lartc.html#LARTC.QDISC

[13] Eric Dumazet: Fair Queue packet scheduler https://lwn.net/Articles/565421/

[14] Apache Software Foundation: nodetool move http://cassandra.apache.org/doc/latest/operating/topo_changes.html#moving-nodes

[15] Stu Hood: Automatic, online load balancing https://issues.apache.org/jira/browse/CASSANDRA-1418

6 Appendix

Source code for all graphs is available on github as well as reproduced below.

```

# coding: utf-8

# In[1]:

from __future__ import print_function
import math
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from scipy.stats import poisson
from scipy.stats import expon

get_ipython().magic(u'matplotlib inline')

# Boils down to "If I pick hosts 2 * (rf - 1) * vnode times, how
    many
# distinct hosts will I have in expectation". Note that this is a
    slightly
# optimistic estimate because Cassandra won't place two replicas
    of the
# same token on the same machine or rack, but this is close enough
    for
# the model
# This is a variant of the Birthday Problem where we are interested
# in the number of distinct values produced
# http://www.randomservices.org/random/urn/Birthday.html
def num_neighbors(n, v, rf, strategy="rack"):
    k = 2 * v * (rf - 1)
    if strategy == "rack":
        # As cassandra is rack aware, we assume #racks == #replicas
        # This is maybe a bad assumption for some datacenter
    deployments
        n = n - (n // rf)
    else:
        # SimpleStrategy
        n = n - 1
    estimate = (n * (1.0 - (1.0 - 1.0/n) ** k))
    return max(rf - 1, min(estimate, n))

def p_outage_given_failure(recovery_seconds, num_neighbors,
    rate_in_seconds):
    x = math.exp(-1 * recovery_seconds * num_neighbors *
    rate_in_seconds)
    return 1 - x

def global_rate(node_rate, nodes, split_probability):
    return node_rate * nodes * split_probability

def recovery_seconds(size, bw_in, bw_out, neighbors,
    recovery='streaming'):
    if recovery == 'ebs':
        return 60 * 5
    return int(size / (min(bw_in, neighbors * bw_out)))


# Default model
nodes = 96
vnodes = 256
rf = 3
# 1000 gigabytes
node_dataset_mb = 300 * 1024
# MB/s
bw_in = 125
# MB/s, cassandra.yaml has 25MBPS as the default
# but most operators observe maybe half of that
bw_out = 25 / 2
strategy = 'rack'

year_seconds = 60.0*60*24*365
century_seconds = 100 * year_seconds

# Model machines that fail on average
# 25 times per century a.k.a 1 in 4 machines
# fails per year, or a machine fails every
# 4 years
arate = 25
arate_in_seconds = 25 / century_seconds


print("\nFailure Rate Variability")
print("Neighbors for {0} vnodes: {1:.3f}".format(1,
    num_neighbors(nodes, 1, rf)))
print("Neighbors for {0} vnodes: {1:.3f}".format(4,
    num_neighbors(nodes, 4, rf)))
print("Neighbors for {0} vnodes: {1:.3f}".format(16,
    num_neighbors(nodes, 16, rf)))

aneighbors = num_neighbors(nodes, vnodes, rf)
arecovery = recovery_seconds(node_dataset_mb, bw_in, bw_out,
    aneighbors)
print("Neighbors for {0} vnodes: {1:.3f}".format(vnodes,
    aneighbors))


def outage_stats(
        vnodes, failure_rate_per_century, num_nodes,
        rf, bw_in, bw_out,
        strategy='rack', recovery='streaming'):
    neighbors = num_neighbors(num_nodes, vnodes, rf, strategy)
    recovery_s = recovery_seconds(node_dataset_mb, bw_in, bw_out,
    neighbors, recovery)
    p_failure = p_outage_given_failure(
        recovery_s, neighbors, failure_rate_per_century /
    century_seconds)

    lmb = global_rate(failure_rate_per_century, num_nodes,
    p_failure)
    return (
```

```python
        poisson.mean(lmb), poisson.interval(0.50, lmb),
    poisson.median(lmb),
        expon.mean(scale=1/lmb), expon.interval(0.50, scale=1/lmb)
    )

# Returns outages _per century_
def compute_outage(
        vnodes, failure_rate_per_century, num_nodes,
        rf, bw_in, bw_out,
        strategy='rack', recovery='streaming'):
    return outage_stats(
        vnodes, failure_rate_per_century, num_nodes, rf, bw_in,
    bw_out, strategy
    )[0]

print("{0:<6} {1:<8} {2:<8} {3:<8} -> {4:<6}".format(
    "rate", "rec_s", "p_fail", "g_lmb", "outages"
))
for rate in (12.5, 25, 50, 100, 200):
    recovery_s = recovery_seconds(node_dataset_mb, bw_in, bw_out,
    aneighbors)
    p_failure = p_outage_given_failure(
        recovery_s, aneighbors, rate / century_seconds)
    gl = global_rate(rate, nodes, p_failure)
    p = "{0:6.2f} {1:6.2f} {2:8.6f} {3:8.4f} -> {4:6.6f}".format(
        rate, recovery_s, p_failure, gl, poisson.mean(gl)
    )
    print(p)


# In[2]:

plt.figure(figsize=(15,10))
plt.title("Cassandra Outages, {0} nodes".format(nodes),
    fontsize=20)
plt.ylabel("Expected Number of Outages per Century", fontsize=16)
plt.xlabel("Number of Tokens per Node", fontsize=16)
plt.xlim(1, 128)
plt.axvline(x=96, color='k', linestyle='--', label='cluster
    size={0}'.format(nodes))
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(4))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(4))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)

num_vnodes = range(1, 128)
rates = [12.5, 25, 37.5, 50]
for rate in rates:
    outages = []
    for vnode in num_vnodes:
        outages.append(compute_outage(vnode, rate, nodes, rf,
    bw_in, bw_out))
    print(outages[:3])
    plt.plot(num_vnodes, outages,
    label="rate={0}/century".format(rate))
```

```
plt.legend(fontsize=16)

outages = [outage_stats(v, arate, nodes, rf, bw_in, bw_out) for v
    in num_vnodes[:32]]
outage_mean = [o[0] for o in outages]
outage_lower = [o[1][0] for o in outages]
outage_upper = [o[1][1] for o in outages]
outage_median = [o[2] for o in outages]

plt.figure(figsize=(15,10))
plt.title(
    "Cassandra Outages, {0} nodes, {1:.2f} failures / Century
    ".format(
        nodes, arate), fontsize=20)
plt.ylabel("Expected Number of Outages per Century", fontsize=16)
plt.xlabel("Number of Tokens per Node", fontsize=16)
plt.xlim(1, 32)
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)
plt.plot(num_vnodes[:32], outage_mean, color='orange', label='mean
    outages')
plt.plot(num_vnodes[:32], outage_median, color='orange',
    label='median outages', linestyle='--')
plt.fill_between(num_vnodes[:32], outage_lower, outage_upper,
    color='orange', alpha=0.05, label='IQR')
plt.legend(fontsize=16)


# In[3]:

# Hold failures constant, vary size of cluster
# Look at MTBF
plt.figure(figsize=(15,10))
plt.title(
    "Cassandra Outages, {1} failures / Century ".format(
        vnodes, arate), fontsize=20)
plt.ylabel("Expected Centuries Between Outages", fontsize=16)
plt.xlabel("Number of Nodes in the Cluster", fontsize=16)
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(4))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.gca().set_prop_cycle('color', colors[0:1] + colors[2:])

num_vnodes = (1, 16, 256)
num_nodes = range(3, 300)
lines = []

for v in num_vnodes:
    outages = [outage_stats(v, arate, n, rf, bw_in, bw_out) for n
    in num_nodes]
    outage_mean = [o[3] for o in outages]
    outage_lower = [o[4][0] for o in outages]
    outage_upper = [o[4][1] for o in outages]
    line, = plt.semilogy(num_nodes, outage_mean,
    label="vnodes={0}".format(v))
    lines.append(line)
    plt.fill_between(
        num_nodes, outage_lower, outage_upper, alpha=0.1,
        label='vnodes={0} IQR'.format(v)
    )

plt.xlim(3, 300)
plt.ylim(1/100.0, 100)

iqr_patch = mpatches.Patch(color='gray', label='IQR')
plt.legend(
    handles=lines + [iqr_patch], loc='upper right', fontsize=16
)


# In[4]:


# Hold failures constant, vary size of cluster
plt.figure(figsize=(15,10))
plt.title(
    "Cassandra Outages, {1} failures / Century ".format(
        vnodes, arate), fontsize=20)
plt.ylabel("Expected Number of Outages per Century", fontsize=16)
plt.xlabel("Number of Nodes in the Cluster", fontsize=16)
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)

num_vnodes = (1, 4, 16, 256)
num_nodes = range(3, 300)
lines = []
for v in num_vnodes:
    outages = [outage_stats(v, arate, n, rf, bw_in, bw_out) for n
    in num_nodes]
    outages = [outage_stats(v, arate, n, rf, bw_in, bw_out) for n
    in num_nodes]
    outage_mean = [o[0] for o in outages]
    outage_lower = [o[1][0] for o in outages]
    outage_upper = [o[1][1] for o in outages]
    line, = plt.plot(num_nodes, outage_mean,
    label="vnodes={0}".format(v))
    lines.append(line)
    plt.fill_between(
        num_nodes, outage_lower, outage_upper, alpha=0.1,
        label='vnodes={0} IQR'.format(v)
    )
```

```python

plt.xlim(3, 300)
iqr_patch = mpatches.Patch(color='gray', label='IQR')
plt.legend(
    handles=lines + [iqr_patch], loc='upper left', fontsize=16
)


# Hold failures constant, vary size of cluster, NetworkTopologyStrategy
plt.figure(figsize=(15,10))
plt.title(
    "Cassandra NetworkTopology Outages, {0} failures / Century".format(
        arate), fontsize=20)
plt.ylabel("Expected Number of Outages per Century", fontsize=16)
plt.xlabel("Number of Nodes in the Cluster", fontsize=16)
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(4))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(10))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)

num_vnodes = (1, 4, 16, 256)
num_nodes = range(3, 51)
for v in num_vnodes:
    outages = [compute_outage(v, arate, n, rf, bw_in, bw_out, 'rack') for n in num_nodes]
    print(v, outages[:3])
    plt.plot(num_nodes, outages, label="vnodes={0}".format(v))

plt.xlim(3, 50)
plt.legend(fontsize=16)


# In[5]:

# Hold failures constant, vary size of cluster
plt.figure(figsize=(15,10))
plt.title(
    "Cassandra SimpleStrategy Outages, {0} failures / Century".format(
        arate), fontsize=20)
plt.ylabel("Expected Number of Outages per Century", fontsize=16)
plt.xlabel("Number of Nodes in the Cluster", fontsize=16)
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(4))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(10))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)

num_vnodes = (1, 4, 16, 256)
num_nodes = range(3, 51)
for v in num_vnodes:
    outages = [compute_outage(v, arate, n, rf, bw_in, bw_out, 'simple') for n in num_nodes]
    print(v, outages[:3])
    plt.plot(num_nodes, outages, label="vnodes={0}".format(v))

plt.xlim(3, 50)
plt.legend(fontsize=16)


# In[6]:

# Observe impact of vnodes on Scale Up Balancing
plt.figure(figsize=(15,10))
plt.title("Cassandra Balanced Scaling",fontsize=20)
plt.ylabel("Number of Nodes needed to Scale Up", fontsize=16)
plt.xlabel("Number of Nodes in the Cluster", fontsize=16)
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)

num_vnodes = (1, 4, 16, 256)
num_nodes = range(3, 256)
for v in num_vnodes:
    scale_up = [math.ceil(float(n) / v) for n in num_nodes]
    plt.plot(num_nodes, scale_up, label="vnodes={0}".format(v))

plt.xlim(3, 256)

plt.legend(fontsize=16)


# In[9]:

# Observe impact of EBS on availability
plt.figure(figsize=(15,10))
plt.title(
    "Cassandra Recovery Strategy, {0} failures / Century".format(arate), fontsize=20)
plt.ylabel("Expected Centuries Between Outages", fontsize=16)
plt.xlabel("Number of Nodes in the Cluster", fontsize=16)
plt.gca().grid(True, which='major', linestyle='-', color='k')
plt.gca().grid(True, which='minor', linestyle='--')
plt.gca().yaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(4))
plt.gca().xaxis.set_minor_locator(matplotlib.ticker.AutoMinorLocator(5))
plt.tick_params(axis='both', which='major', labelsize=14, length=6)
plt.tick_params(axis='both', which='minor', length=0)

num_nodes = range(3, 300)
lines = []

# Plot fixed recovery speed
outages = [outage_stats(1, arate, n, rf, bw_in, bw_out, recovery='ebs') for n in num_nodes]
outage_mean = [o[3] for o in outages]
```

```
outage_lower = [o[4][0] for o in outages]
outage_upper = [o[4][1] for o in outages]
line, = plt.semilogy(num_nodes, outage_mean, label="EBS,vnode=any")
lines.append(line)
plt.fill_between(
    num_nodes, outage_lower, outage_upper, alpha=0.1,
    label='EBS,vnode=any IQR'.format(v)
)

outages = [outage_stats(1, arate, n, rf, bw_in, bw_out,
    recovery='recovery') for n in num_nodes]
outage_mean = [o[3] for o in outages]
outage_lower = [o[4][0] for o in outages]
outage_upper = [o[4][1] for o in outages]
line, = plt.semilogy(num_nodes, outage_mean,
    label="streaming,vnode=1")
lines.append(line)
plt.fill_between(
    num_nodes, outage_lower, outage_upper, alpha=0.1,
    label='streaming,vnode=1 IQR'
)

plt.xlim(3, 300)

plt.legend(fontsize=16)


# In[8]:

import itertools
import random
import sys

def simulate(l):
    return random.expovariate(l)

def offset(values, max_value=float("inf")):
    nvalues = values[:1] + [0] * (len(values) - 1)
    for i in range(1,len(values)):
        nvalues[i] = values[i] + nvalues[i-1]
    return [n for n in nvalues if n <= max_value]

def outage(o, f_i, t, neighbors):
    failures = 0
    neighbor_indices = range(0, len(o))
    neighbor_indices.remove(f_i)
    random.shuffle(neighbor_indices)
    for n in range(int(round(neighbors))):
        failures += near(o[neighbor_indices[n]], o[f_i], t)
    return failures

def near(a, b, t):
    failures = 0
    for i in a:
        for j in b:
            if j > i + t:
                break
```

```python
            if j - i > 0 and j - i < t:
                failures += 1
    return failures

def run_simulate(l, neighbors, nodes):
    rs = []
    for r in range(5):
        o = [offset([simulate(l) for j in range(300)]) for i in range(nodes)]
        maxes = [x[-1] for x in o]
        m = max(maxes)
        outages_per_century = (
            sum([outage(o, i, arecovery / century_seconds, neighbors) for i in range(nodes)]) /
            m
        )
        print("Run {0} gave {1:.3f} outages/century".format(r, outages_per_century))
        rs.append(outages_per_century)
    print("Simulation outages/century: ", sum(rs) / len(rs))

def run_simulate_naive(l, neighbors, nodes):
    p_split = p_outage_given_failure(arecovery, aneighbors, l / century_seconds)
    l_split = l * p_split
    l_global = nodes * l_split
    print(p_split, l_global)
    rs = []
    for r in range(10):
        events = 1000
        o = offset([simulate(l_global) for j in range(events)])
        ##failure_years = [int(x/year) for x in o]
        num_centuries = max(o)
        rs.append(events / num_centuries)
    print("Simple simulation outages/century: ", sum(rs) / len(rs))


print(p_outage_given_failure(arecovery, num_neighbors(nodes, vnodes, rf), arate))
print(vnodes, arate, nodes, rf, bw_in, bw_out, arecovery, num_neighbors(nodes, vnodes, rf))
print("Predicted outages/century:", compute_outage(vnodes, arate, nodes, rf, bw_in, bw_out))
#run_simulate(arate, num_neighbors(nodes, vnodes, rf), nodes)
run_simulate_naive(arate, num_neighbors(nodes, vnodes, rf), nodes)
#run_simulate(arate, num_neighbors(nodes, vnodes, rf), nodes)
```