Summary

We've long observed that our balancer doesn't really work very well at my day job. After digging into the balancer code I've realized that the cost functions that we care about are basically all being overshadowed by some default multipliers that are huge for a few costs that are always zero when not using secondary replicas.

Further, we'd like our HBase balancing to be smarter in many ways, but expanding the suite of cost functions requires needlessly complicated code that's never evaluated in a vacuum, and elaborate configuration incantations.

How the balancer works today

The Stochastic load balancer relies on a set of cost functions. The cost functions attempt to convey imbalances among things like region count, storefile size, read request volume, etc..

Each cost function also has a configurable multiplier, which is any number >=0. This multiplier tells the balancer how significantly to weigh a given cost function in the overall balancer decision. You would want a higher multiplier for the cost functions that you consider to be most important.

These code snippets, mostly taken verbatim from the Stochastic load balancer, explain pretty clearly what it's doing:

```java
sumMultiplier = 0; // the sum of all cost function multipliers
for (CostFunction c : costFunctions) {
  if (c.isNeeded()) {
    sumMultiplier += c.getMultiplier();
  }
}

double totalCost = 0.0; // the total cost from all cost functions and their
multipliers
for (CostFunction c : costFunctions) {
  if (!c.isNeeded()) {
    LOG.trace("{} not needed", c.getClass().getSimpleName());
    continue;
  }
  totalCost += c.cost() * c.getMultiplier();
}
```

```
// the minimum cost which would indicate an imbalance. This is configurable
minCostNeedBalance = getMinCostNeedBalance();

// if true, we are balanced
boolean balanced = (totalCost / sumMultiplier < minCostNeedBalance);
```

It's important to notice that we divide the `totalCost` by the `sumMultiplier` — in other words, the existence of a large multiplier on one cost function can make other, smaller, cost outputs basically irrelevantly small.

A few dominating cost functions

There are a few problematic cost functions:

1. PrimaryRegionCountSkewCostFunction
2. RegionReplicaRackCostFunction
3. RegionReplicaHostCostFunction

`PrimaryRegionCountSkewCostFunction` has a high default multiplier of 100000 and a cost of 0 without secondary replicas enabled. `RegionReplicaRackCostFunction` has a default multiplier of 10000. `RegionReplicaHostCostFunction` has a default multiplier of 500 which is also quite high compared to other defaults, but obviously not to the same magnitude.

For example, some other cost function multiplier defaults are:

- `ReadRequestCostFunction`: 5
- `StoreFileCostFunction`: 5
- `TableSkewCostFunction`: 35

The result is that tables, even those without secondary replicas, must become egregiously imbalanced in order for the default balancer to take reasonable action, and even when it does it's virtually exclusively acting on the `RegionCountSkewCostFunction` and `HeterogeneousRegionCountCostFunction` functions (which have the highest default, excluding the 3 aforementioned replica costs, of 500). **Having any of these multipliers at 100,000 while the others are single digit values makes the latter costs basically irrelevant.**

Proposal

HBASE-28513 Pull Request

We should expand the stochastic load balancer to no longer only evaluate on continuous scales. There should be a set of discrete **"balancer conditionals"** that it will evaluate in addition to the existing cost functions.

This will both fix deficits in the balancer today, and enable a more powerful, flexible, and straightforward balancer in the future.

For example, we should write a "DistributeReplicasBalancerConditional" which will evaluate each RegionPlan, and reject it if it would colocate two replicas on a single host/rack. This would be much easier to reason about compared to the existing replica cost functions, and would not require squashing read/write/storefile size cost functions as well.

Another example, but this time a new feature: we should write a "IsolateSystemTablesBalancerConditional" which will evaluate each RegionPlan, and reject those that prevent system tables from running on a dedicated RegionServer.

A third example, we should expand on the system table isolation above and support meta table isolation.

At my day job, these are all features that we would love to have OOTB tomorrow, and balancer conditionals would make these features easy to support.

How Balancer Conditionals Could Fix This

Decoupling Constraints from Cost Functions

Balancer conditionals would allow us to decouple discrete, high priority, constraints from soft balancing goals. Instead of relying solely on multipliers and cost functions to enforce critical rules (like distributing read replicas), conditionals could outright reject any `RegionPlan` that violates predefined rules..

Examples of Balancer Conditionals in Action

1. **DistributeReplicasBalancerConditional**:
    - Rejects `RegionPlan` proposals that colocate replicas of the same region on the same host or rack.
    - This is much clearer and more direct than relying on the `RegionReplicaRackCostFunction` or `RegionReplicaHostCostFunction`, which must weigh the "cost" of violations against other costs.

2. **IsolateSystemTablesBalancerConditional**:
    - Rejects `RegionPlan` proposals that colocate system tables with user tables.

    - Ensures that system tables like `hbase:quota` are assigned to their own dedicated RegionServers.
    - This could be done relatively trivially with balancer conditionals, and I'd challenge you to write a usable cost function that could achieve the same.
    - Theoretically this could be achieved through RS groups — but that's a management nightmare. Suddenly you own the operational burden of defining groups for your user tables and system tables; groups that must be large enough to have redundancy, but small enough to avoid being tremendously wasteful. It is much simpler to have a balancer that will clearly prefer system table isolation, while also having nothing extra to manage and nothing strictly blocking the assignment of critical tables to any given server.

3. **MetaTableIsolationConditional**:
    - Rejects `RegionPlan` proposals that colocate meta table replicas with user tables.
    - Further isolates `hbase:meta` to a dedicated RegionServer, independent of other system or user tables.
    - Much like the above, this could be achieved through RS groups, but I think there's a strong argument for this approach being simpler and better.

Benefits of Balancer Conditionals

- **Simplicity**: Developers and operators can define clear, easy-to-understand rules without needing to configure or debug complex cost function multipliers.
- **Precision**: Removes ambiguity around balancing goals by separating "strong" constraints (enforced by conditionals) from "soft" optimization goals (addressed by cost functions).
- **Extensibility**: Adding new conditionals is straightforward and doesn't require adjusting multipliers or rebalancing existing cost functions.
- **Improved Debugging**: Conditionals can provide detailed logging for why specific `RegionPlan` decisions were rejected, improving debuggability and operability.

Implementation Overview

Adding Balancer Conditionals

1. **Define Conditional Interface**:
    - Introduce an interface (e.g., `BalancerConditional`) with methods to evaluate and reject invalid `RegionPlan` proposals.

```java
public interface RegionPlanConditional {
  boolean isViolating(RegionPlan plan);
}
```

2. **Integrate with StochasticLoadBalancer**:
    - Modify the balancer to evaluate conditionals alongside cost functions

```java
public class StochasticLoadBalancer {

  public void balanceTable(BalancerClusterState) {
    // This is short hand for how the balancer works today
    RegionPlan regionPlan = getRandomRegionPlan();

    // We can evaluate the conditional violation count change caused by each
plan
    int conditionalViolationChange =
balancerConditionals.getViolationChange(regionPlan);

    // Then we can get the cost change of the region plan like we do today
    boolean costsImproved = doCostsImprove(regionPlan);

    boolean conditionalsImproved = conditionalViolationChange < 0;
    boolean conditionalsSimilarCostsImproved =
        conditionalViolationChange == 0 && costsImproved;
    if (conditionalsImproved || conditionalsSimilarCostsImproved) {
      accept(regionPlan); // plan looks good
    }
  }

}
```

3. **Initial Conditionals**:
    - Add some initial conditionals for replica distribution, system table isolation, and meta table isolation.
    - Provide configuration keys to enable or disable each conditional.
    - All of these conditionals will be disabled by default to maintain familiar behavior by default.

Operational Changes

New Configuration Options

The following new configuration options would be introduced:

- `hbase.master.balancer.stochastic.conditionals.isolateSystemTables`: set this to true to enable system table isolation
- `hbase.master.balancer.stochastic.conditionals.isolateMetaTable`: set this to true to enable meta table isolation
- `hbase.master.balancer.stochastic.conditionals.distributeReplicas`: set this to true to enable conditional based replica distribution
- `hbase.master.balancer.stochastic.additionalConditionals`: much like cost functions, you can define your own RegionPlanConditional implementation and install it here.

Backward Compatibility

- The proposed changes are backwards-compatible with existing configurations, and will be turned off by default.
- Operators can gradually adopt conditionals by enabling them one at a time.
- Existing cost functions remain functional and are unaffected by these changes.
- All initial conditionals can be enabled in tandem without deadlock.

Conclusion

Introducing balancer conditionals would make the HBase balancing process:

- **More predictable** by separating strong constraints from optimization goals.
- **Easier to configure** by eliminating the need for complex multiplier tuning.
- **More powerful** by providing a clear framework for adding new balancing rules.

These changes would address long-standing issues with the balancer and lay the groundwork for a more robust and maintainable balancing strategy. By adopting this proposal, HBase can significantly improve its balancing logic and meet the needs of both traditional and modern workloads.

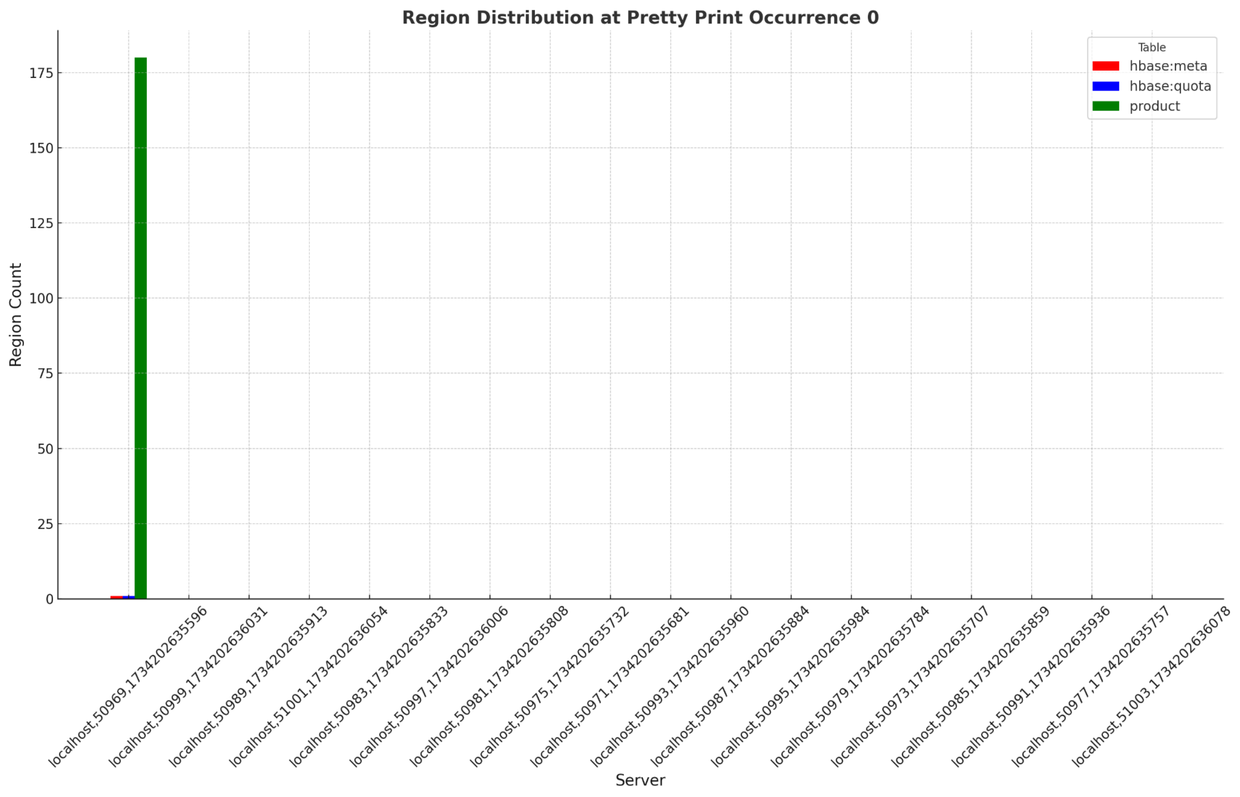
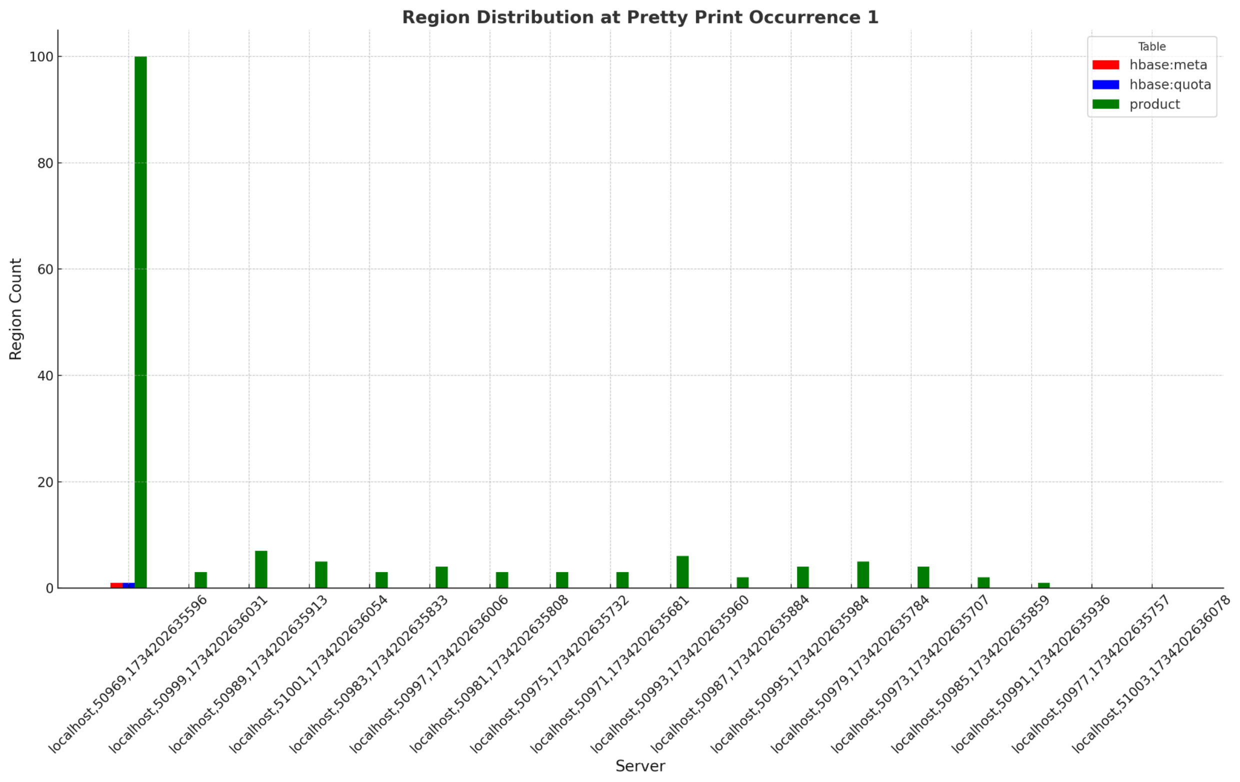
Testing

Table Isolation

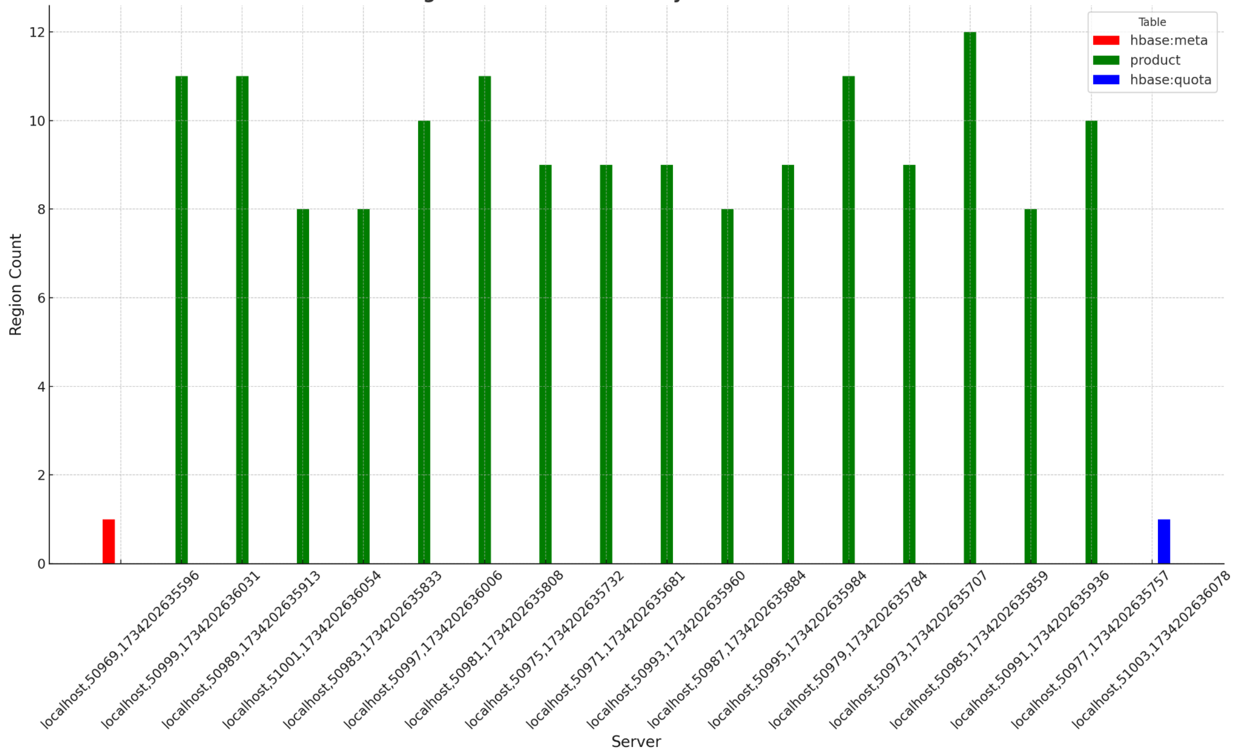
See below where we ran a new unit test, TestLargerClusterBalancerConditionals, and tracked the locations of regions for 3 tables across 18 RegionServers:

1. 180 “product” table regions
2. 1 meta table region
3. 1 quotas table region

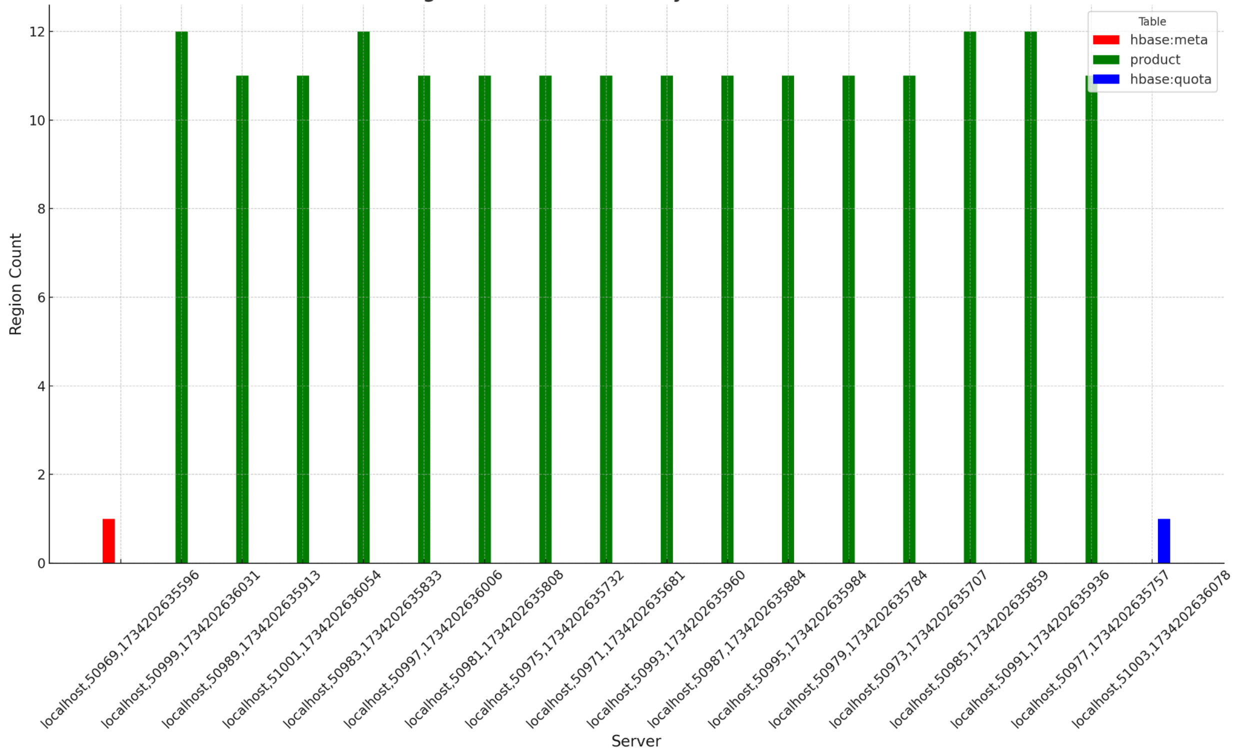
All regions began on a single RegionServer, and within 4 balancer iterations we had a well balanced cluster, and isolation of key system tables.


Region Distribution at Pretty Print Occurrence 0
Region Count
Server
Table
hbase:meta
hbase:quota
product
0
25
50
75
100
125
150
175
localhost,50969,1734202635596
localhost,50999,1734202636031
localhost,50989,1734202635913
localhost,51001,1734202636054
localhost,50983,1734202635833
localhost,50997,1734202636006
localhost,50981,1734202635808
localhost,50975,1734202635732
localhost,50971,1734202635681
localhost,50993,1734202635960
localhost,50987,1734202635884
localhost,50995,1734202635984
localhost,50979,1734202635784
localhost,50973,1734202635707
localhost,50985,1734202635859
localhost,50991,1734202635936
localhost,50977,1734202635757
localhost,51003,1734202636078

Region Distribution at Pretty Print Occurrence 1
Region Count
Server
Table
hbase:meta
hbase:quota
product
0
20
40
60
80
100
localhost,50969,1734202635596
localhost,50999,1734202636031
localhost,50989,1734202635913
localhost,51001,1734202636054
localhost,50983,1734202635833
localhost,50997,1734202636006
localhost,50981,1734202635808
localhost,50975,1734202635732
localhost,50971,1734202635681
localhost,50993,1734202635960
localhost,50987,1734202635884
localhost,50995,1734202635984
localhost,50979,1734202635784
localhost,50973,1734202635707
localhost,50985,1734202635859
localhost,50991,1734202635936
localhost,50977,1734202635757
localhost,51003,1734202636078

Region Distribution at Pretty Print Occurrence 2



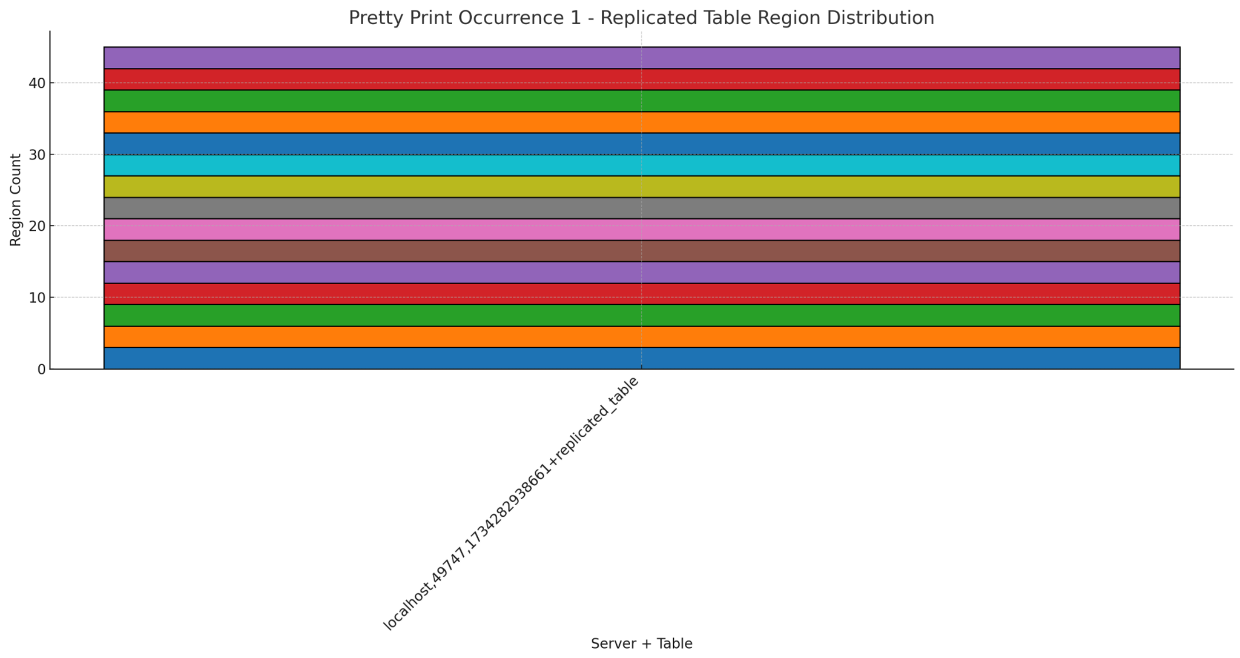
Region Distribution at Pretty Print Occurrence 3

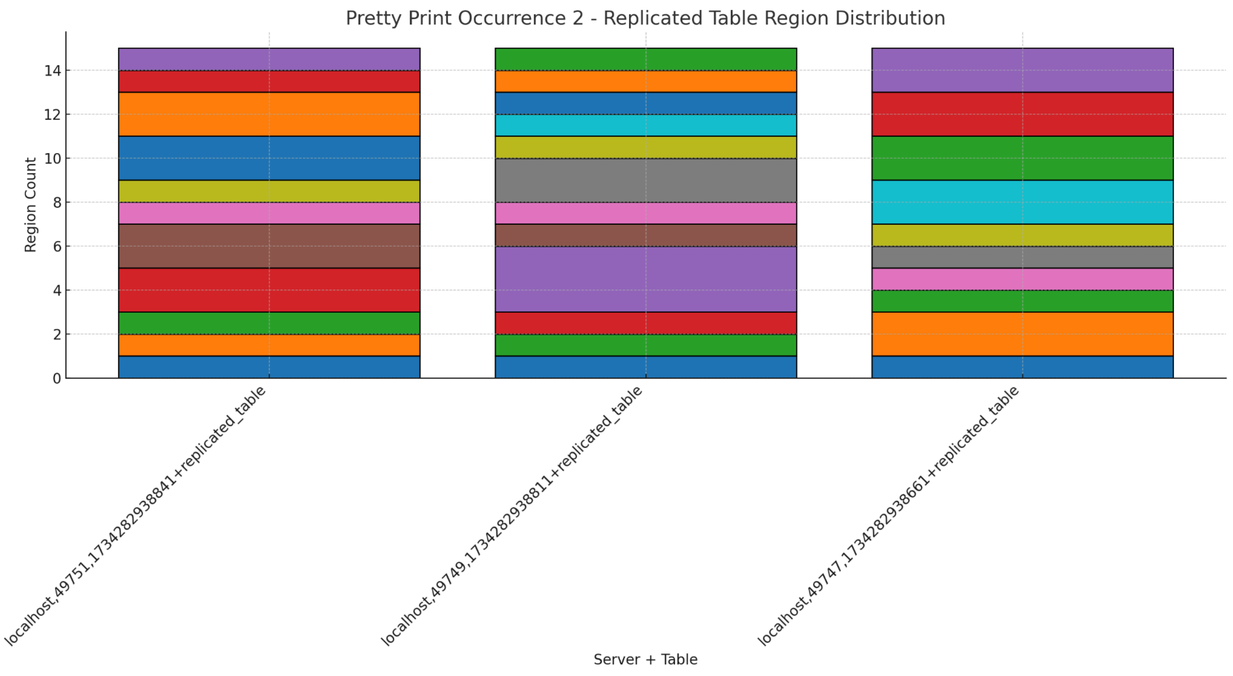


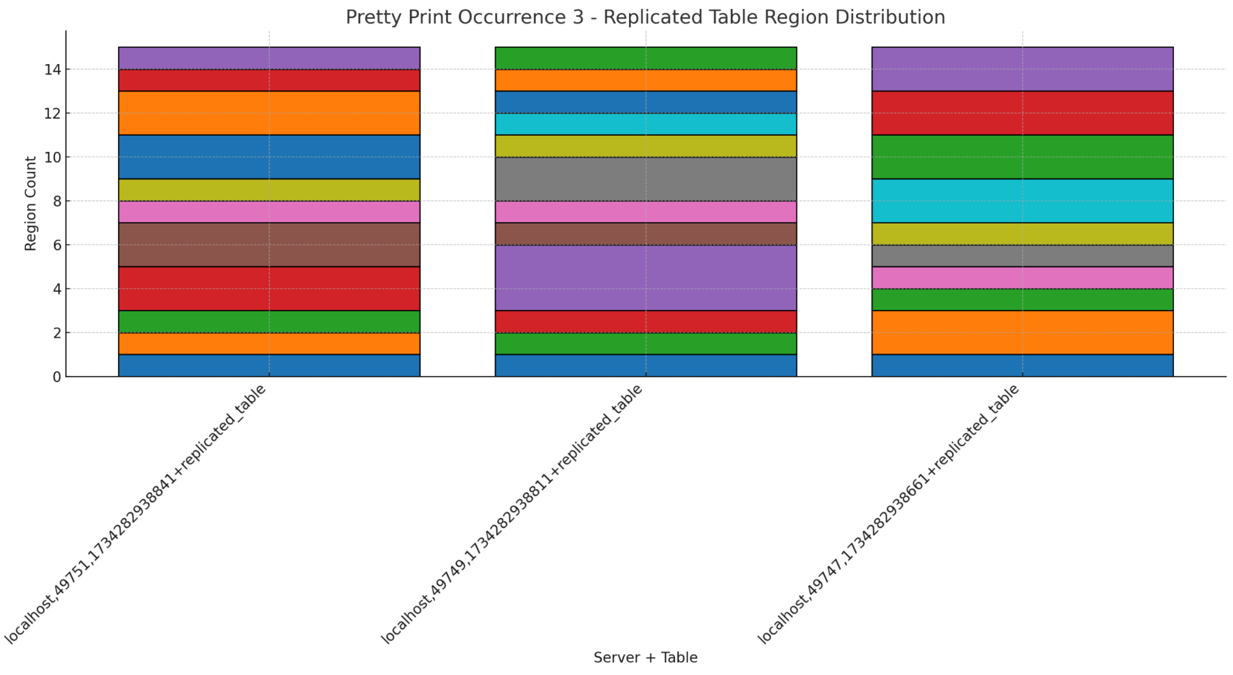
Replica Distribution

Traditional Replica Cost Functions *Don't* Work

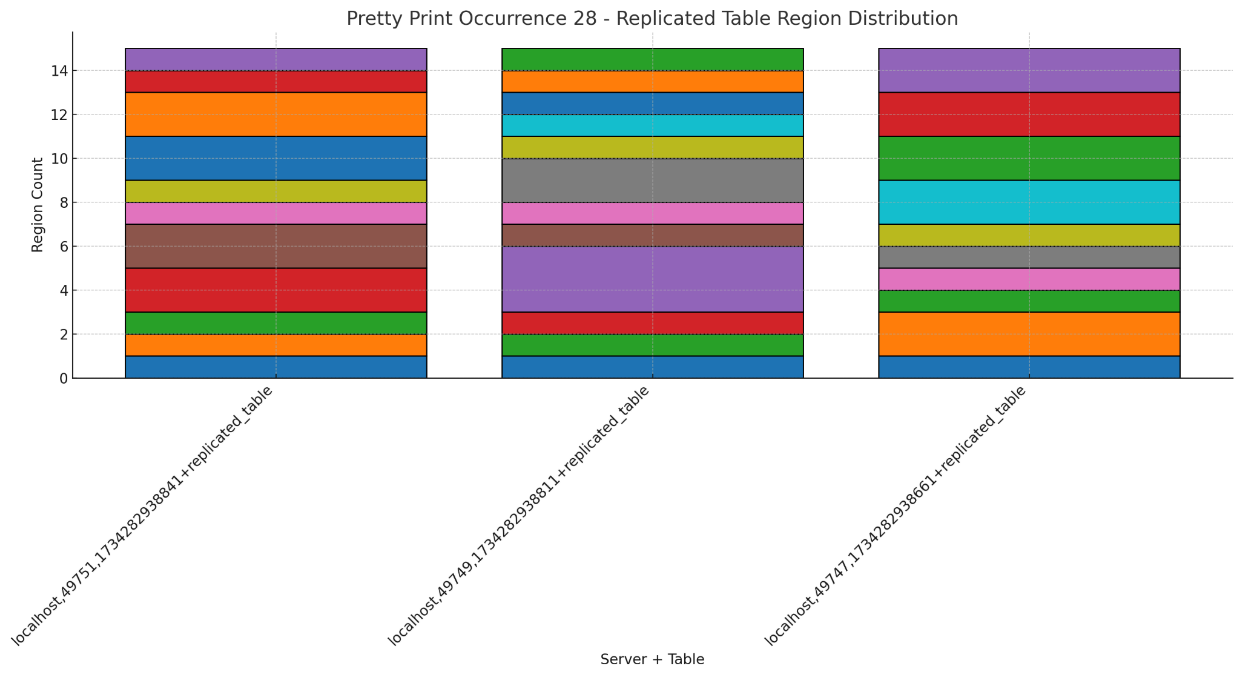
Below, we have `replicated_table`, a table with 3 region replicas. The 3 regions of a given replica share a color, and there are also 3 RegionServers in the cluster. We expect the balancer to evenly distribute one replica per region per server across the 3 RegionServers, and can **watch our traditional replica cost functions fail** to do so**.**







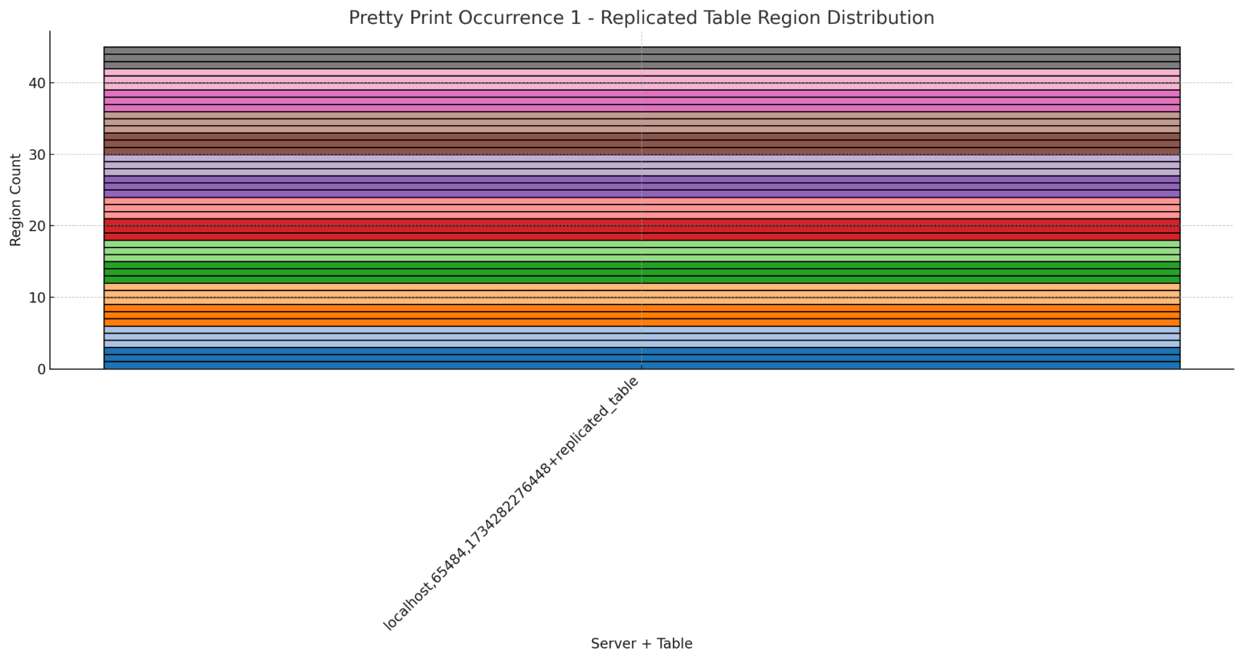
….omitting the meaningless snapshots between 4 and 27…

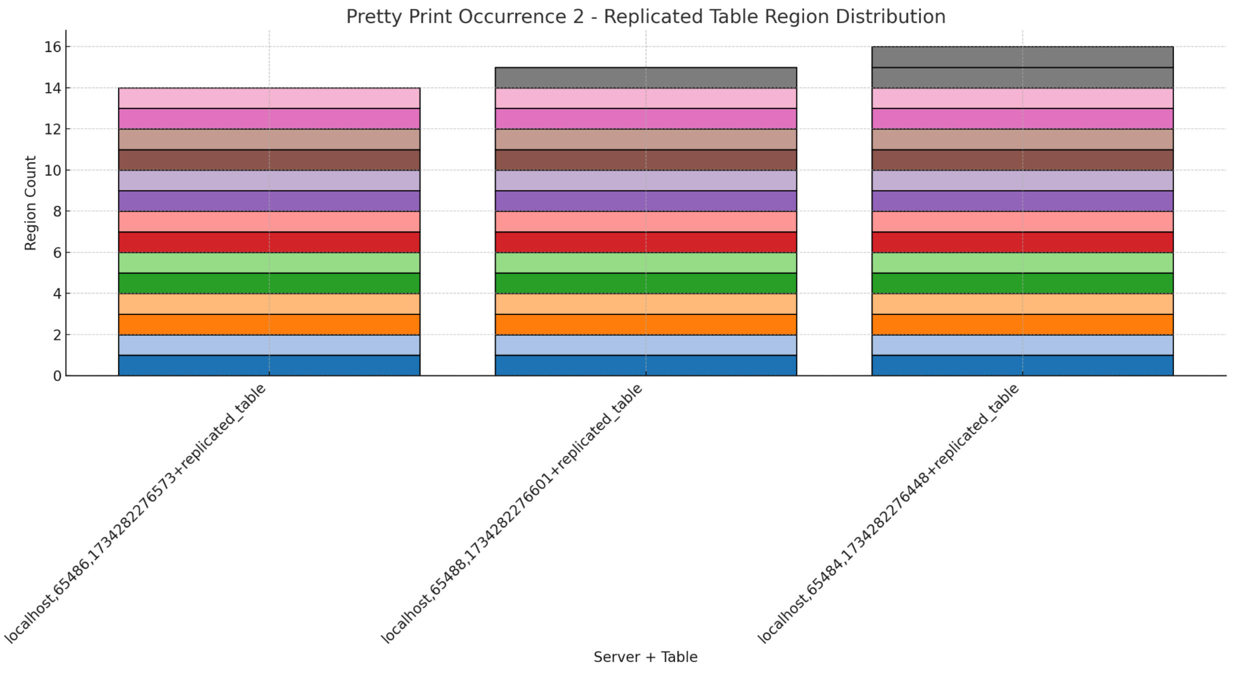


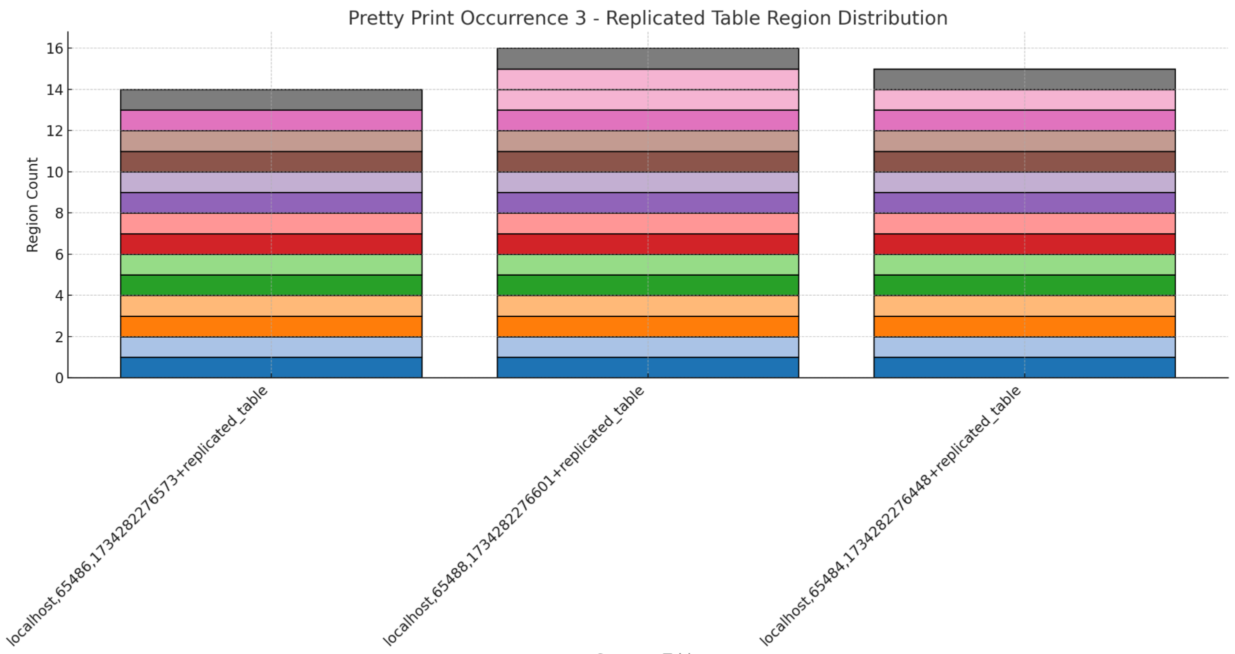
At this point, I just exited the test because it was clear that our existing balancer would never achieve true replica distribution.

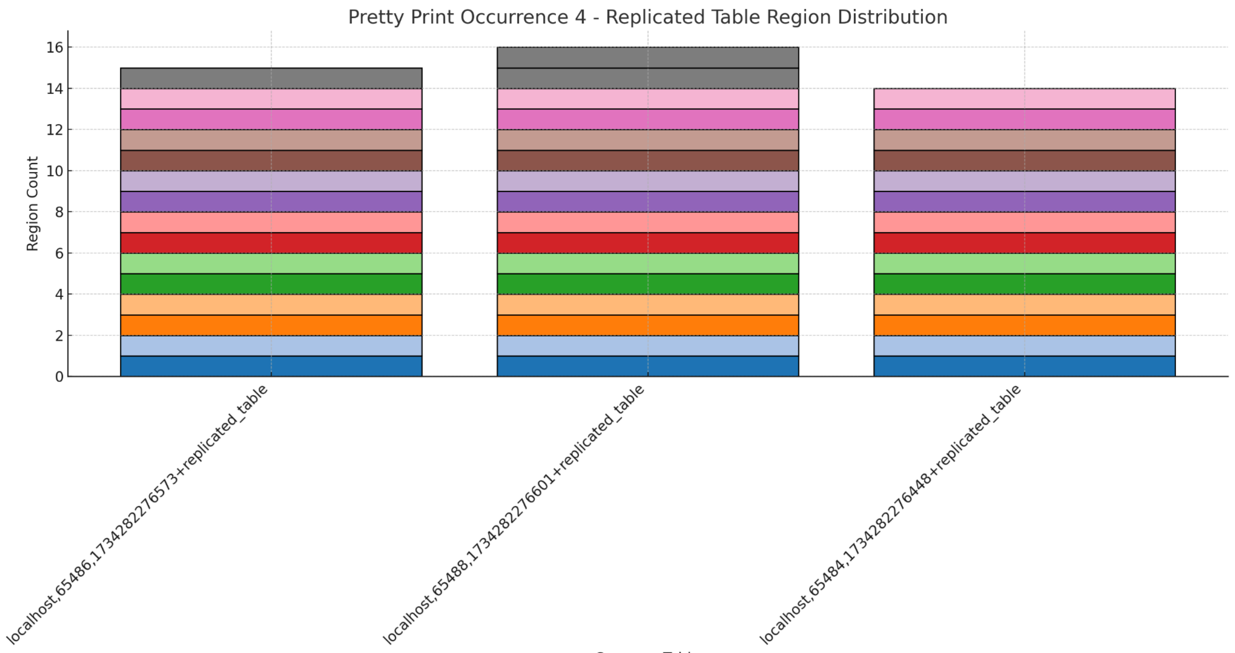
Balancer Conditionals *Do* Work

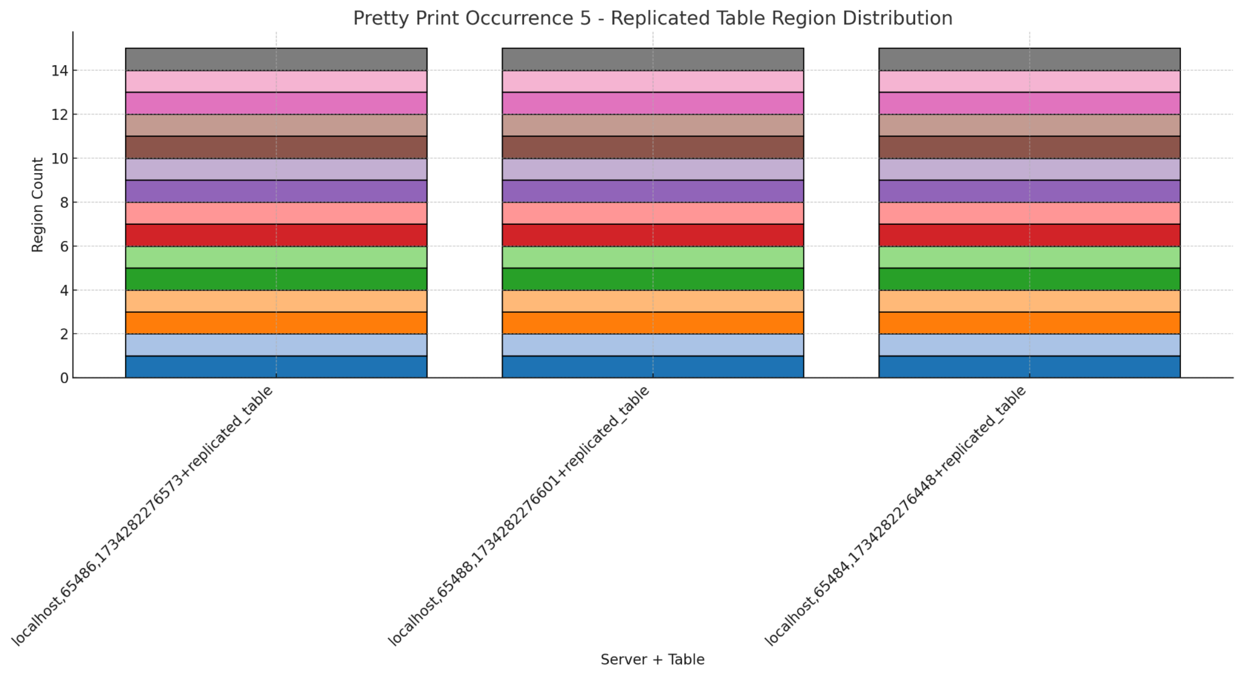
Below, we have `replicated_table`, a table with 3 region replicas. The 3 regions of a given replica share a color, and there are also 3 RegionServers in the cluster. We expect the balancer to evenly distribute one replica per server across the 3 RegionServers, and can **watch balancer conditionals do so successfully** in approximately one minute on my local machine:


Pretty Print Occurrence 1 - Replicated Table Region Distribution
Region Count
0
10
20
30
40
localhost,65484,1734282276448+replicated_table
Server + Table


Pretty Print Occurrence 2 - Replicated Table Region Distribution
Region Count
0
2
4
6
8
10
12
14
16
localhost,65486,1734282276573+replicated_table
localhost,65488,1734282276601+replicated_table
localhost,65484,1734282276448+replicated_table
Server + Table


Pretty Print Occurrence 3 - Replicated Table Region Distribution
Region Count
0
2
4
6
8
10
12
14
16
localhost,65486,1734282276573+replicated_table
localhost,65488,1734282276601+replicated_table
localhost,65484,1734282276448+replicated_table
Server + Table


Pretty Print Occurrence 4 - Replicated Table Region Distribution
Region Count
0
2
4
6
8
10
12
14
16
localhost,65486,1734282276573+replicated_table
localhost,65488,1734282276601+replicated_table
localhost,65484,1734282276448+replicated_table
Server + Table


Pretty Print Occurrence 5 - Replicated Table Region Distribution
Region Count
0
2
4
6
8
10
12
14
localhost,65486,1734282276573+replicated_table
localhost,65488,1734282276601+replicated_table
localhost,65484,1734282276448+replicated_table
Server + Table