Hunter：使用变点检测追踪性能回退

Matt Fleming*
matt@codeblueprint.co.uk

Piotr Kołaczkowski*
pkolaczk@datastax.com

Ishita Kumar*
ishitakumar@umass.edu

Shaunak Das*
shaunak.das@datastax.com

Sean McCarthy*
sean.mccarthy@datastax.com

Pushkala Pattabhiraman*
Pushkala.Pattabhiraman@datastax.com

Henrik Ingo*
henrik.ingo@avoinelama.fi

arXiv:2301.03034v1 [cs.DB] 8 Jan 2023

摘要

变点检测近来日益受到关注，因为它能够处理噪声数据，因而可用于检测软件中的性能变化。本文介绍开源工具 Hunter，它能够自动检测时间序列数据中的性能回退和性能提升。Hunter 使用经过修改的 E-divisive 均值算法，识别正态分布性能指标中的统计显著变化。我们介绍了对 E-divisive 均值算法所做的修改及其动机。我们采用的主要修改是将基于随机置换的显著性检验替换为学生 t 检验，因为我们发现，随机方法无法产生确定性结果，至少在迭代次数合理的情况下无法做到。此外，我们还进行了调整，使算法能够找到原始算法无法发现的变点，例如相邻较近的两个变化。为了进行评估，我们开发了一种生成真实时间序列的方法，并在其中人为注入延迟变化。我们使用这些数据集，将 Hunter 与另外两种著名算法 PELT 和 DYNP 进行了比较。最后，我们总结了在多个团队中推广 Hunter 的经验；这些团队各自负责其项目的性能。

关键词

变点检测、性能、基准测试、持续集成

ACM 引用格式：
Matt Fleming、Piotr Kołaczkowski、Ishita Kumar、Shaunak Das、Sean Mc- Carthy、Pushkala Pattabhiraman 和 Henrik Ingo。2023年。Hunter：使用变点检测来寻找性能回退。收录于论文集 *ICPE 2023*：国际性能工程会议（*ICPE2023*）。ACM，美国纽约州纽约市，8页。https://doi.org/10.1145/1122445.1122456

*DataStax, Inc

允许为个人或课堂用途免费制作本作品全部或部分内容的数字或纸质副本，前提是这些副本不得为盈利或商业利益而制作或分发，并且须在首页保留本声明和完整引文。对于本作品中由 ACM 以外的其他方拥有版权的组件，必须尊重其版权。注明出处后允许摘录。若要以其他方式复制、转载、发布到服务器或分发给列表，则须事先获得明确许可和/或支付费用。请向 permissions@acm.org 申请许可。*ICPE2023，2023年4月15日至19日*，葡萄牙科英布拉 © 2023年 计算机协会。ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00 https://doi.org/10.1145/1122445.1122456

1 引言

对分布式数据库（如 Apache Casandra）进行性能测试，是开发流程中不可或缺的一部分，通常也会纳入持续集成流水线，以便定期运行性能测试和基准测试，或在将更改推送到源代码仓库时运行。然而，鉴于分布式系统的复杂性，其性能往往不稳定，即使在相同硬件上，性能测试结果也可能因运行次数不同而波动。这种结果不稳定由多种因素造成，包括底层硬件的差异[7], 、操作系统层的后台进程和 CPU频率调节，以及应用程序层面的请求调度和优先级排序 [1]。所有这些因素都使得自动判断性能变化究竟是由软件更改导致，还是仅仅源于测试噪声，变得极其困难。文献中介绍了基于阈值的技术，但这些方法无法妥善处理基准测试数据中的噪声，而且要求为每项测试分别设置阈值 [5]。此外，随着性能提升不断加入并建立新的基线，阈值也需要定期调整。

过去，我们一直高度依赖经验丰富的工程师，通过目视检查图表和基准测试数据来识别性能变化。然而，这种做法存在诸多缺点，包括：

- 识别变化所需的专业知识难以传授给其他工程师
- 规模较小的专家团队能够检查的测试数量有限
- 即使是经验丰富的工程师也可能会漏掉变化

鉴于这些缺点，我们最近创建了 Hunter[15],这一开源工具，它利用变点检测来找出时间序列数据中具有统计显著性的变化。近年来，变点检测逐渐成为应对性能测试和基准测试数据固有不稳定性（即噪声）的一种方法 [5] ，并且能够识别性能回退和性能提升。

Hunter 的设计目标是无需再由专门的工程师团队筛查性能测试

结果。各团队可以将基准测试数据输入中央数据存储库，再由 Hunter 从中提取并分析数据。我们使用 Hunter 验证多个分布式数据库和流式处理产品的多个发布版本，因此需要让 Hunter 直观易用，方便那些精通各自领域、但并非性能专家的工程师使用。

本文的贡献如下：

- 我们提出了一种开源工具，可对包含多个指标的任何时间序列数据运行变点检测。这些数据可以存储在 CSV 文件中，也可以存储在 Graphite 服务器上。
- 我们讨论了对 E-divisive 均值算法所做的修改，以提升其性能并增强结果的可预测性。
- 我们开发了一种方法，用于生成真实基准测试结果的时间序列，并在离散时间点人为注入延迟变化。这使我们能够根据一组已知的正确变点，客观地评估算法的准确性。
- 我们将 Hunter（修改后的 E-divisive 均值）与另外两种变点检测算法 DYNP 和 PELT 进行了比较。
- 我们分享了在多团队环境中运行 Hunter 的经验。在这种环境中，每个团队负责不同的产品，并倾向于使用不同的基准测试。

2 高层概述

Hunter 是一款使用 Python 编写的命令行工具，可检测存储在 CSV 文件或 Graphite 服务器上的时间序列数据中具有统计显著性的变化。它旨在轻松集成到构建流水线中 [10]，并提供自动化性能分析，以判断代码是否应部署到生产环境。除了在命令行中打印变点数据外，Hunter 还支持 Slack，并可配置为将结果发送到 Slack 频道。

2.1 数据源

Hunter 可以分析从 Graphite 服务器获取的数据，也可以分析 CSV 文件中包含的数据。要将 Hunter 集成到我们的测试与部署工作流中，Graphite 支持必不可少。如果开发人员没有使用 Graphite 作为基准测试数据的中央存储库，CSV 支持则提供了一种通用的数据输入方式，可将数据提供给 Hunter。

2.2 配置

Hunter 使用的数据源在 YAML 配置文件中指定。该配置文件包含 Graphite 服务器、Slack 令牌和数据定义等部分。Hunter 还支持模板功能，可重复使用通用定义，避免重复编写测试定义。由于我们经常使用 Hunter 运行数百项测试并监控数百个指标，模板功能有助于缩小配置文件的体积。例如，我们使用 Graphite 指标前缀将相关指标分组，从而让特定 Apache Cassandra 版本的所有指标都通过一个共同的字符串关联起来。

一个例子是 testdb.20k-rw-ts.fixed，这是在 Datastax Enterprise 上运行的基准测试，以固定的吞吐量执行读写操作。我们会在两种配置下运行此测试，一种配置的复制因子为 1，另一种为 3。尽管存在这一差异，我们仍然可以复用约 95% 的 Hunter 配置，因为指标类型相同。

下面是一个配置文件示例。

graphite: url: http://graphite.local 后缀：- ebdse_read.result

模板：common_指标：指标：吞吐量：缩放：1 方向：1 p99：缩放：1.0e-6 方向：-1 测试：db.20k-rw-ts.fixed：继承：- common_指标 标签：- db.20k-rw-ts.fixed.1-bm2small-rf-1 前缀：性能_回归. db.20k-rw-ts.fixed

对于 CSV 文件中的测试数据，Hunter 允许用户指定文件的属性，例如文件路径、哪些列包含时间戳、哪些列包含指标，以及用于分隔每行字段的分隔符。

2.3 持续集成

由于 Hunter 是一个简单的 Python 应用程序，因此连接到不同团队的持续集成流水线已被证明非常简单。我们使用 Docker 镜像运行 Hunter，对存储在中央 Graphite 服务器上的每日性能测试结果进行分析。该 Docker 镜像由每天运行一次的 Jenkins 作业启动。

2.4 将结果发送到 Slack

对给定的时间序列运行变点检测后，Hunter 可以将分析结果提交到 Slack 频道。我们发现，这里是通知开发人员性能变化的理想位置，主要是因为每个频道都已按团队或项目分类。开发人员通常会对 Hunter 的结果进行初步分析，调查性能方面的任何意外变化，以确定产品性能是否确实发生了变化，还是由工作负载或平台中的噪声造成的。

将 Hunter 的结果显示在 Slack 频道等显眼的位置，促使我们改进了运行性能测试所用的底层基础设施。当测试结果因平台噪声而频繁波动时，我们的一个团队便提升了这些平台的稳定性，以便从 Hunter 获得更多可采取行动的结果。

3 实现

Hunter 基于 MongoDB 的 signal_processing_algorithms 库中提供的 E-divisive 均值算法构建而成，但我们对其进行了两方面的扩展，以提高效率（从而更快地生成结果），并确保对同一数据集进行多次迭代时能够获得可重复的结果。

3.1 E-divisive 均值算法

E-divisive 均值 [12] 是一种离线变点检测算法，它使用层次估计来估算分布中变点的数量和位置。由于这是一种非参数方法，因此它不对底层数据分布或分布中的变化作任何假设，非常适合用于基准测试数据，因为这类数据通常并非正态分布。层次结构体现在决定要在哪一组数据点中搜索变点。E-divisive 均值会将时间序列分成两部分，并递归地搜索变点。

使用文献中称为 $\hat{q}$ 的检验统计量来检验各个数据点，该统计量基于先前的变点；$\hat{q}$ 的 p 值则通过随机置换检验确定，这需要进行多次计算。使用随机排列会带来性能开销，我们发现，对于我们的数据集，检测变点所需的时间长得令人难以接受。此外，由于排列是随机的，我们发现 Hunter 的结果具有非确定性，每次运行都会有所不同。增加排列次数可以减少结果的非确定性，但会导致 Hunter 的运行时间增加。就我们的情况而言，使用标准 E-divisive 均值算法运行 Hunter，对单个测试和单个指标的数百个数据点进行分析，需要 2 到 3 秒。但为了验证夜间构建或发布版本，开发人员需要能够对数十项测试进行变点检测，而每项测试都记录了数十个指标。这会使运行时间延长到几分钟，不再理想。

3.2 显著性检验

在最初开发 Hunter 时，我们对代码进行了性能分析，以了解哪些部分耗时最长，导致检测数据中的变点所需时间较长。我们发现，绝大部分时间都花在了显著性检验上。考虑到使用了 $\hat{q}$ 统计量，并且该统计量依赖随机排列，这一点并不完全令人意外。我们改用学生 t 检验后，Hunter 的运行时间缩短了一个数量级，而且对同一数据集多次运行时，结果也始终一致。虽然学生 t 检验并不是适用于任意数据集的稳健统计显著性度量，但事实证明，它非常适合我们的场景。

我们还测试了使用曼-惠特尼U检验。这个方法颇具吸引力，因为与学生 t 检验不同，它是一种非参数检验，不假设输入数据呈正态分布。但结果发现，它在数据量较少时表现不佳，因为至少需要 30 个数据点才能得出确定结论。相比之下，原始的 E-divisive 和我们基于学生 t 检验的版本，都能在极短的时间序列中检测出变化，只需 4 至 7 个数据点即可。由于 E-divisive 是一种层次化算法，会将

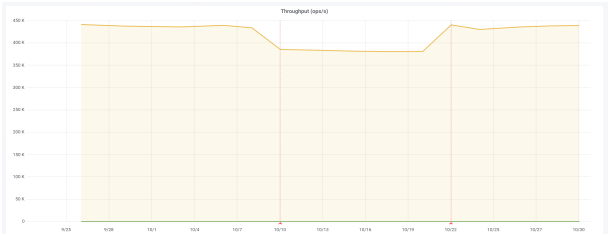

图 1：暂时性异常示例

原始时间序列不断划分为更小的窗口，因此这是一个显著差异。

3.3 固定大小窗口

当我们开始在越来越大的数据序列上使用 Hunter 时，发现先前运行中识别出的变点会突然从 Hunter 的结果中消失。后来发现，这个问题是由性能回退导致的，而这些回退在引入后不久便得到修复。E-divisive 均值存在一个已知问题，[5] 中对此有讨论。由于 E-divisive 均值会将时间序列分成两部分，分割点两侧的大多数数据点都呈现相似的值。因此，该算法按设计会将这两个相近的变化视为暂时性异常，而非持续性变化，并将其过滤掉。

图 1 展示了这个问题。

我们解决这个问题的方法，是将整个时间序列拆分为固定大小的窗口，并分别在每个窗口上运行 E-divisive 均值算法。位于窗口边界的变点需要特别处理，因为一般而言，变点检测算法无法判断数据序列中最新的点是否为变点。为了解决这个问题，Hunter 允许窗口相互重叠，并确保不会因为某个变点存在于多个窗口中而重复报告该变点。

3.4 弱变点

将数据序列拆分为多个窗口，部分解决了大型数据集中遗漏变点的问题，但我们还需要一种方法，强制 E-divisive 均值算法继续递归分析数据序列。E-divisive 均值算法会在显著性检验（Hunter 中使用的是学生 t 检验）未通过时终止。如果算法首先选出的变点的 p 值高于用户设定的阈值（通常为 0.05），它就会立即终止，即使继续运行本可以检测到 p 值低于该阈值的变点。我们将未通过显著性检验、但会引出其他 p 值低于该阈值的点的变点称为弱变点。

处理弱变点分为两个步骤。首先，在拆分时使用较大的 p 值阈值，以便检测到弱变点。其次，我们重新评估 p 值，并采用自底向上的方式合并拆分后的数据序列，移除 p 值高于较小的用户指定阈值的变点。我们发现，如果不强制递归继续进行，Hunter 就会漏掉一些变点。我们的修改使 p 值准确得多。

此外，我们会过滤掉相对变化较小的变点，例如指标值差异低于 5% 的变点。这个相对阈值可用作筛选条件，剔除不可采取行动的变点，也就是说，剔除小到开发人员无法复现或验证修复效果的变点。

4 评估

我们使用取自 Datastax Enterprise 每日 Gatling [9]性能测试的基准测试数据评估了我们的算法。基准测试数据保存为 CSV 文件，并通过以下命令行传递给 Hunter：

```
poetry run hunter analyse db-gatling.csv .
```

db-gatling.csv 中的数据包含 175 个条目，涵盖了 15 个月的数据。其中包含多项性能变化，包括性能提升（吞吐量更高或延迟更低）和性能回退（吞吐量更低或延迟更高）。

我们选择从 CSV 文件中读取数据，以避免与 Graphite 服务器之间的网络通信延迟影响每次运行的时长。每种算法都在同一个 CSV 文件上运行了 30 次，表 1 报告了均值及其 95% 置信区间。

由于我们发现置换算法产生的结果不稳定，因此表 1 还列出了每种算法检测到的变点平均数量及标准差。

4.1 快速回退的性能回退

在图 1 中，我们可以看到吞吐量在 2020年10月10日前后有所下降。这次性能回退是由 Datastax Enterprise 中网络数据包解码和处理方式的变更导致的。这项有问题的变更于 2020年10月21日回退，这解释了为什么吞吐量指标不久后就恢复到了先前的数值。两条红线标示出了存在性能回退的数据范围。这是变点检测中一个已知的问题， [5] 中对此有明确说明。

学生 t 检验和弱变点算法都在数据中检测到了这次回归，并在 30 次运行中的每一次都检测到了回退。基于置换的算法仅在 30 次运行中的 15 次，即 50% 的情况下检测到了这些变化。

4.2 MongoDB 性能测试结果数据集

我们还使用了公开的 MongoDB 性能测试结果数据集 [13]，将 E-divisive 均值与随机排列进行比较，并将学生 t 检验与弱变点过滤结合起来作为统计显著性检验。这是 [5]，中使用的同一数据集，因此，对于新兴的变点检测领域而言，此分析应当具有可比性，也更为熟悉。

我们从微基准测试套件中任意选取了 5 个测试，并且只关注 最大值_每_秒_操作数 结果。所有结果均来自任务 杂项读取命令 和变体 *linux-wt-standalone*。这 5_ _个时间序列如图 2 所示。为避免图表过于杂乱，我们只在其中一个时间序列上标出了检测到的变点，但其他 4 个时间序列的结果也类似。

如上所述，原始 E-divisive 算法并非确定性的。表2显示了在

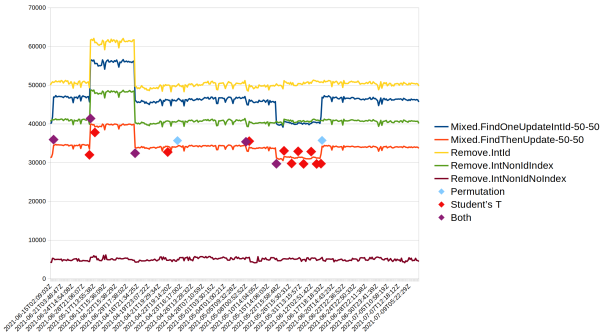

图 2：来自公开 MongoDB 微基准测试数据集的 5 项测试。

每个时间序列进行 100 次迭代后检测到的变点数量。结果令人担忧。例如，对于*Remove.IntNonIdNoIndex*（第 5 行），43% 的情况下会检测到 4 个变点，50% 的情况下会检测到 3 个变点，但还有 7% 的情况下竟然检测不到任何变点！

图 2 还展示了我们解决的另一个主要问题。如果性能回退后很快就得到修复或反弹，那么原始算法往往会将其中一个或两个变化视为噪声而忽略。图中的第 9 个变点就是这样的例子；只有借助本文采用的固定大小窗口方法，才能检测到它。

最后，我们可以清楚地看到，我们的实现检测到了更多变点。（红色菱形仅由学生 t 检验配置检测到。）这是预期结果，因为大多数修改都是为了让算法尽可能灵敏。全部 16 个变点是否都有意义，最终取决于主观判断；但仔细观察图表，至少可以理解算法为何会选择每个点。该实现提供了过滤过小、不可采取行动的变化的功能，但这些测试并未使用此功能，因为我们希望展示修改后的 E-divisive 算法的完整输出，而不进行后置过滤。

我们可以预料到一个显而易见的问题：如果 MongoDB 最初的实现表现如此糟糕，MongoDB 自己又怎么能如此成功地使用它呢？答案是，MongoDB 性能持续集成系统的更高层经过了专门设计：一旦某个点被标记为变点，系统就会永远记住它。这样一来，当开发者正在着手修复问题时，变点就不会随机消失；同样，被标记为假阳性的点也会继续保持静默，不会在第二天再次出现。不过，针对本文所指出的问题，其影响是：即使某一天 E-divisive 算法可能提前停止、只返回部分结果，整个 MongoDB 持续集成系统最终仍会找出并记住所有变点。这个更高层的系统记录在 [11] 中，记录下来的变点也属于 [13] 的一部分。

4.3 使用已知变点的数据集进行评估

上述分析，以及据我们所知迄今为止发表的所有相关文献，都存在一个弱点：对算法或其实现的准确率进行评判总是主观的。归根结底，决定某个报告的

表 1：Hunter 的 e.divisive 实现中，不同显著性检验的性能和结果准确率。

算法	平均持续时间	平均95%置信区间	变点	变点标准差
置换检验	2.221	2.209, 2.233	16	1.174
学生 t 检验	1.863	1.853, 1.873	20	0
学生 +弱变点	1.594	1.584, 1.603	16	0

表2：使用不同统计检验发现的变点数量分布。（MongoDB 数据集，100 次迭代）

算法	检验名称	0	1	2	3	4	5	6	7	8	9	12	14	16	18
置换检验	Mixed.FindOneUpdateIntId-50-50					1			1		98				
置换检验	Mixed.FindThenUpdate-50-50					1			99						
置换检验	Remove.IntId				64		32	4							
置换检验	Remove.IntNonIdIndex				61		34	5							
置换检验	Remove.IntNonIdNoIndex	7			50	43									
学生 +弱	Mixed.FindOneUpdateIntId-50-50														100
学生 +弱	Mixed.FindThenUpdate-50-50													100	
学生 +弱	Remove.IntId												100		
学生 +弱	Remove.IntNonIdIndex												100		
学生 +弱	Remove.IntNonIdNoIndex											100			

变点是否为真正例、是否“有用”或“可采取行动”，取决于人工评估者。请注意，这些判断可能并不相同！这是因为我们无法获得正确变点集合的客观事实。如果我们掌握了这些信息，一开始就不需要这个系统了。

当然，可以生成在已知步骤处注入变化的合成时间序列，例如正弦波，甚至白噪声，其中均值或振幅在离散点发生变化。不过，这些测试往往显得很直观，而 E-divisive 在这些测试中表现相当出色。

为了获取真实数据集，我们使用 Chaos Mesh[3] 在被测系统中人为生成网络延迟。换句话说，我们在已知时间点人为注入了真实变化，从而在真实基准测试中获得了其他方面都符合实际情况的结果。所用的基准测试是使用与持续集成[6]相同的工具链对 Cassandra 进行测试。

为了创建不同的时间序列，我们选取了一组变量，并通过改变这些变量来生成不同的场景。我们通过调整以下变量的取值创建了 9 种不同的场景：变点数量、组间方差变化幅度、组间变化幅度以及组的长度。组定义为两个变点之间出现的点的集合。每种场景包含 5 个测试序列，每个序列都有细微差异。这些场景本身可分为三类。第一种场景是均值变化，通过改变各组的均值来产生变点，而方差基本保持不变。同样，方差变化场景中均值保持不变，方差则不断变化。这种情况非常适合复现噪声环境。均值和方差同时变化的场景，则通过随机延迟真实地复现噪声环境。

请注意，本次评估使用的时间序列与前文不同。前文中的时间序列由一系列（夜间）构建组成，数据点表示测试期间的平均吞吐量等数值；而本次评估中的时间序列来自单个基准测试，数值则是快照每秒生成一次。这是因为等待一年来创建由真实夜间构建组成的时间序列并不现实。

4.3.1 评估指标。 获得包含已知变点的数据集后，我们现在可以采用客观的统计检验来衡量 Hunter 的准确率。从本质上讲，我们已将评估任务重新定义为一个机器学习问题：给定输入训练数据集后，算法应当生成已知的输出。

我们将使用两个指标来评估 Hunter：F1 分数和兰德指数。具体而言，我们将评估 **p99** 指标。p99 表示每 100 名用户中有 1 名会遇到延迟问题。这是常见的行业标准，在开发 Cassandra 时也被用于设定性能目标。

4.3.2 真正例。 我们将使用以下两个变量来表示真实值集合和预测点集合：

X^*：真实值集合X：
预测点集合

真正例定义为检测类别中实际存在的变点集合，即它们存在于该集合中。M 表示误差范围。如果预测点与真实变点之间的差值小于或等于M，我们将其视为正确预测的变点。

$$TP(X, X^*) := \{x \in X | \exists x^* \in X^* s.t. |x - x^*| \leq M\}$$

重要的是确保不存在重复，即预测集合中的两个点不会同时落入 X^* 中同一个点的误差范围内。当 X 中的某个点落入 M 范围内并被加入 $TP(X, X^*)$ 集合后， X^* 中的变点就会被标记为 已访问 ，且之后不能再被考虑。

4.3.3 F1 分数。 使用 F1 分数计算 Hunter 准确率的原因是，它不受数据规模和密度的影响，会对假阳性进行惩罚，并认可正确检测。

F1 分数定义为

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

精确率是预测的变点中真正变点所占的比例：[17]

$$Precision = \frac{|TP(X, X^*)|}{|X|^*}$$

召回率是被准确预测的真实变点占所有真实变点的比例：[17]

$$Recall = \frac{|TP(X, X^*)|}{|X|}$$

4.3.4 兰德指数。 我们评估的另一个指标是兰德指数。

$$RandIndex(X, X^*) = \frac{TP + TN}{TP + TN + FP + FN}$$

- **真正例**： 正确预测为正类的点：计算出的真实变点
- **真负例**： 正确预测为负类：本例中为无
- **假正例**： 模型错误地预测为正类：$|X^*| - |TP(X, X^*)|$
- **假负例**： 模型错误地预测为负类：$|X| - |TP(X, X^*)|$

4.3.5 与 *PELT* 和 *DYNP* 进行基准测试。有了可供基准测试的客观事实，也就可以将 Hunter 与其他变点检测算法进行比较。因此，我们还展示了另外两种知名离线算法 PELT 和 DYNP 的结果。这些算法通过 Python 中的 ruptures 软件包运行。[17]

4.3.6 结果。我们对 Hunter 进行了 45 次测试运行。测试运行在使用 Kubernetes 的 GKE 集群上进行。该集群在 *us-central1-a* 区域的 4 个 *n2-standard-4* 节点上运行。

所有算法都根据两项指标进行了评估。可以看出，Hunter 在这两项指标上的表现始终优于*PELT* 和*DYNP* 。所有实验的误差范围均为**10 秒**。

4.3.7 与点数的相关性。点数与准确率之间存在正相关关系。随着点数增加，Hunter 的性能也会提升。从图3中可以看出，

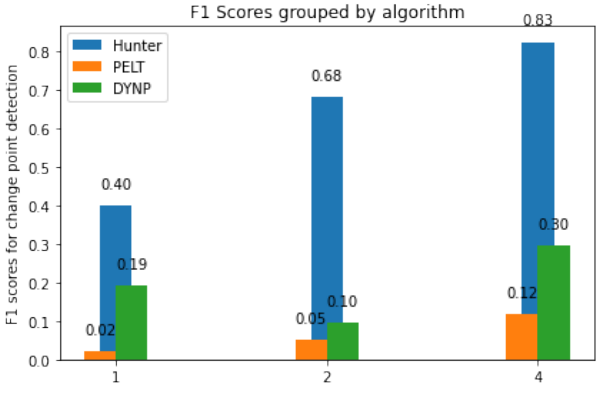

图3：F1 分数与点数之间的相关性

Hunter 以显著优势超过了 PELT 和 DYNP。

4.3.8 误差幅度与算法之间的相关性。 如果允许更大的误差范围，Hunter 的性能就会提升。当误差幅度为一秒时，Hunter 的*F1* 分数 可达到**0.1481**；而 PELT 和 DYNP 则需要至少 3 秒的误差范围才能获得非零分数。这是 E-divisive 一直以来都能有效帮助我们在持续集成中检测回归的关键特性。我们更希望知道导致回归的确切提交，而不仅仅是大致判断回归可能发生的区域。E-divisive 在满足这一要求方面尤其出色！随着误差范围稍微放宽，所有算法的性能都会大幅提升。误差范围为 4 秒时，Hunter 的性能提升至*0.612*。误差范围达到 15 秒时，DYNP 的准确率开始赶上 Hunter。

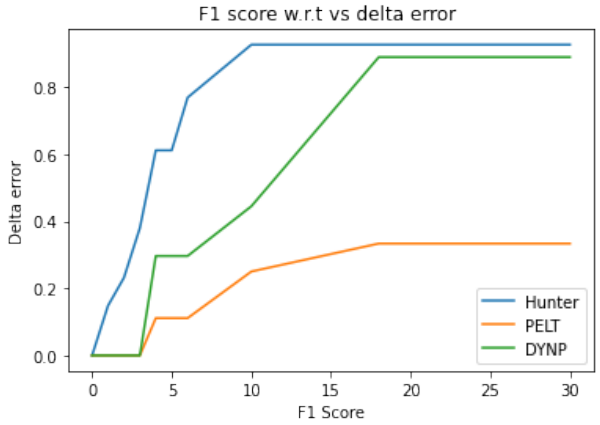

图 4：F1 与误差幅度之间的相关性

5 经验教训

目前，我们运行 Hunter 为多个团队提供服务已经接近 2 年。在此期间，除了第 3 节介绍的算法改进外，我们还进行了多项性能提升。我们吸取的经验教训以及据此采取的改动，帮助 Hunter 成为 DataStax 内部进行统计显著性检测的事实标准选择。

5.1 更多数据点更好

我们最初使用了为期 2 周的数据点。由于每天运行一次性能测试，因此共有 14 个数据点。这样决定主要是因为我们想避免从 Graphite 服务器收集大量数据所造成的延迟。事实证明，这些数据点对于 Hunter 来说太少，无法得出有意义的结果，因此我们将默认时长增加到了一个月（约 30 个数据点）。更长的时间范围让 Hunter 能更好地应对结果中的噪声，现在假阳性变点也少了。我们计划今后尝试使用涵盖更长时间段的数据集，看看能否进一步降低假阳性率。

表 3：评估结果

算法	*Hunter*		*Pelt*		*Dynp*	
指标	*F1*	*Rand*	*F1*	*Rand*	*F1*	*Rand*
单变点						
场景1	0.261904	0.166667	0.0	0.0	0.0	0.0
场景2	0.466667	0.305556	0.027027	0.013698	0.285714	0.166667
场景3	0.466667	0.305556	0.040556	0.020698	0.285714	0.166667
两个变点						
场景4	0.666667	0.666667	0.060606	0.031746	0.190476	0.111112
场景5	0.888889	0.833334	0.090909	0.047619	0.095238	0.055556
场景6	0.490476	0.327777	0.0	0.0	0.0	0.0
四个变点						
场景7	0.925926	0.866667	0.249999	0.142857	0.444445	0.285714
场景8	0.731313	0.579365	0.085713	0.045073	0.095238	0.055556
场景9	0.818182	0.714286	0.016460	0.00843	0.0	0.0

5.2 新变点至关重要

向开发者报告某个变点后，就没有必要继续报告它了。当 Hunter 发现许多变点时，通过 Slack 报告可能会让结果变得过于繁杂，也让开发人员难以分析。如果某个变点表示的性能回退后来已经修复，情况就会更糟，因为 Hunter 会将先前的回退和之后的性能改善作为两个独立的变更分别报告。

为了减少 Hunter 的 Slack 功能产生的输出，我们将结果限制为仅显示过去 7 天内的变点。虽然这确实会忽略有价值的数据，因为随着新数据的处理，变点的幅度可能会更新，但这些变化还没有重要到需要在 Slack 频道中向所有人发送大量消息。对于需要查看完整结果列表的开发人员，他们可以直接对数据序列手动运行 Hunter。

5.3 变点检测无法修复噪声数据

有一支使用 Hunter 的团队经常通过 Slack 机器人收到变点消息。在调查这些变点后，他们发现应用程序的性能并未发生变化，基准测试结果的变化实际上是由私有数据中心中不稳定的硬件性能造成的。中位延迟出现 +- 10% 的变化属于常见情况。

虽然 Hunter 可以检测时间序列数据中的统计显著变化，但它仍然无法避免受到剧烈波动数据点的影响，例如在未经调优的硬件上运行基准测试所产生的数据。

然而，团队无法充分发挥 Hunter 的优势，这促使他们调查问题的根源，随后将基准测试和测试迁移到云上。事实证明，云上的结果比内部

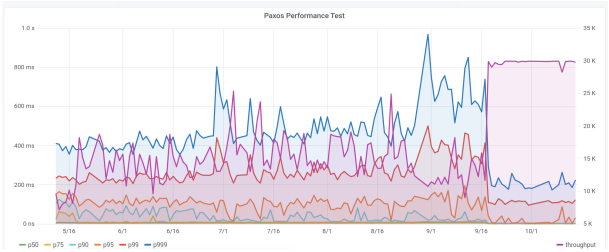

图5：性能不稳定示例

基准测试实验室硬件上的结果更具可重复性。迁移后，基准测试结果稳定得多，Hunter 产生的假阳性也大幅减少。图5展示了一项基于 Paxos 的性能测试的基准测试数据。在 2021-09-18 于公有云上运行该测试之前，Hunter 平均每月检测到 3 个变点。这些全都是假阳性，也就是说，结果的变化并非由软件或配置修改造成。迁移后，Hunter 一次假阳性都没有检测到。

6 相关工作

我们在创建 Hunter 时直接采用了 [5] 中的研究成果，而本文的新贡献则针对其中提出的一些尚未解决的问题。具体而言，[5] 的作者指出，E-divisive 均值算法存在固有偏差，倾向于检测簇中心的变点。他们通过将变点检测与异常检测相结合来弥补这一偏差；异常检测能够在新数据序列中出现第一个数据点时，立即识别出性能的大幅变化。我们使用窗口分析数据序列，以解决同样的偏差，而无需诉诸异常检测；与变点检测相比，异常检测对变化的敏感度较低。此外，我们能够更早地检测到变化，通常在 1 至 2 天内即可发现。

持续基准测试 [10] 是一种常用技术，可确保在新代码合并到源代码仓库时，产品性能得以维持或提升。相关文献中也有示例，展示了如何使用变点检测 [4]和基于阈值的方法，在持续集成流水线中识别软件性能的变化 [16] 。还可以将多种变点检测算法组合成一个集成算法，在识别性能变化时，其表现可能优于单个算法 [19] 。

变点检测领域的文献浩如烟海， [2] 和 [17]对在线与离线、有监督与无监督的变点检测算法进行了出色的综述和分类。在 [2] 中，尤其详细介绍了在线滑动窗口算法。

在线变点检测也已用于识别性能变化。 [20] 将变点检测与区间马尔可夫链的概率模型检测相结合，以便及时检测软件系统参数的变化，并验证系统的正确性、可靠性和性能。

众所周知，即使在不同时间使用相同硬件运行相同软件，在云端运行性能测试也容易受到性能变异性的影响 [18] 。历史性能数据可用于预测云环境中的未来性能； [21]研究了两种变点检测算法——robseg 和 breakout——以预测云中的变异性，从而帮助用户规划可重复的实验。 [8] 使用 E-divisive 均值算法回答了这样一个问题：无服务器应用程序的性能稳定性是否会随时间变化?

7 结论

要在一系列产品版本中检测性能回退，需要借助自动化手段快速识别这些回退，而不必依赖专业开发人员手动检测。变点检测已成为解决这一问题的一种方案，因为它能够应对性能测试中固有的数据噪声。

Hunter 是一个开源[15]工具，使用变点检测自动识别时间序列数据中的变化。这些数据可以来自 Graphite 服务器或 CSV 文件，Hunter 会报告是否存在变点。Hunter 扩展了 E-divisive 均值算法，引入学生 t 检验，以消除原始版本中的非确定性，并确保每次对同一数据序列运行时都能得到可复现的结果。我们还引入了滑动窗口技术，用于检测在时间上彼此接近的变点。除了优于原始的 E-divisive 均值实现外，Hunter 似乎也优于另外两种知名算法 PELT 和 DYNP。

8 致谢

我们感谢 Guy Bolton King 为 Hunter 作出的贡献。

参考文献

[1] 2014. 尾部的故事：硬件、操作系统和应用程序层面的尾延迟来源 （美国华盛顿州西雅图）。计算机协会，美国纽约州纽约市。https://doi.org/ 10 1145 2670979 2670988 . / . Samaneh Aminikhanghahi 和 Diane [2] Cook。2017。《时间序列变点检测方法综述》。*Knowledge and Information Systems* 51 （2017年5月）。https://doi.org/10.1007/s10115-016-0987-z

[3] 云原生计算基金会。 [n.d.]. Chaos Mesh：面向 Kubernetes 的强大混沌工程平台。https://chaos-mesh.org/ 访问日期：2022- 10-20。[4] David Daly。2021。在 MongoDB 开展性能测试，创建良性循环。载于*ACM/SPEC*国际性能工程会议论文集 （线上活动，法国）（*ICPE '21*）。计算机协会，美国纽约州纽约市， 33 41 10 1145 3427921 3450234 – 。https://doi.org/ 。 / 。 [5] David Daly、William Brown、Henrik Ingo、Jim O'Leary 和 David Bradford。2020。利用变点检测在持续集成系统中识别软件性能回退。载于*2020*年*ACM/SPEC*国际性能工程会议论文集（*ICPE '20*）（2020）。https://doi.org/ 10 1145 3358960 3375791 . / 。Datastax。n.d.。Fallout：分布式系统测试即服务。 [6] [ ] https://github.com/datastax/fallout 访问日期：2022-10-20。[7] Dmitry Duplyakin、Alexandru Uta、Aleksander Maricq 和 Robert Ricci。2020。在数据中心性能领域，唯一不变的就是变化。载于*2020*年第*20*届*IEEE/ACM*集群、云和互联网计算国际研讨会（*CCGRID*）。 370 379 10 1109 49817 2020 00 56 – 。https://doi.org/ 。 / CCGrid 。 。[8] - Simon Eismann、Diego Elias Costa、Lizhi Liao、Cor-Paul Bezemer、Weiyi Shang、André van Hoorn 和 Samuel Kounev。2021。无服务器应用程序性能测试稳定性的案例研究。arXiv:cs.DC/ 2107 13320 。Gatling。n.d.。 [9] [ ] Gatling 开源负载测试——适用于 DevOps 和持续集成/持续交付。https://gatling.io/ 访问日期：2021-10-07。[10] Martin Grambow、Fabian Lehmann 和 David Bermbach。2019。持续基准测试：在构建流水线中使用系统基准测试。载于*2019*年*IEEE*云工程国际会议（*IC2E*）。241–246。https://doi.org/ 10.1109/IC2E.2019.00039[11] Henrik Ingo 和 David Daly。2020。MongoDB 的自动化系统性能测试。载于数据库系统测试研讨会（*DBTest'20*）（2020）。https: //doi.org/ 10 1145 3395032 3395323 . / 。David S. Matteson 和 Nicholas A. James。2014。多变量数据多变点分析的非参数方法。*J. Amer. Statist. Assoc.* 109, 505 2014 334 345 ( ), – 。http://www.jstor.org/stable/ 24247138 [12] [13] [ ] MongoDB。n.d.。MongoDB 性能测试结果数据集。https://zenodo.org/record/5138516#.YW6T-erMIUH 访问日期：2021-10-13。[14] MongoDB。 [n.d.]。信号处理算法。https://github.com/mongodb/signal-processing-algorithms 访问日期：2021-10-13。[15] Piotr Kołaczkowski。 [n.d.]。Hunter——追踪性能回退。https://github.com/datastax-labs/hunter 访问日期：2022-12-12。[16] Kim-Thomas Rehmann、Changyun Seo、Dongwon Hwang、Binh Than Truong、Alexander Böhm 和 Dong Hun Lee。2016。SAP HANA 持续集成流程中的性能监控。*ACM SIGMETRICS Performance Evaluation Review*。 *43* 2016 43 52 10 1145 2897356 2897362 ( ), – 。 2019 [17] https://doi.org/ 。 / 。Charles Truong、Laurent Oudre 和 Nicolas Vayatis。。离线变点检测方法精选综述。信号处理 167（2019年9月），107299。https://doi.org/ 10 1016 2019 107299 . /j.sigpro. 。Alexandru Uta、Alexandru [18] Custura、Dmitry Duplyakin、Ivo Jimenez、Jan Rellermeyer、Carlos Maltzahn、Robert Ricci 和 Alexandru Iosup。2020。现代云网络中的大数据性能是否可复现？ 载于第*17*届*USENIX*网络系统设计与实现研讨会（*NSDI 20*）。USENIX协会，圣克拉拉，加利福尼亚州，513–527。https://www.usenix.org/conference/nsdi20/presentation/uta [19] Ti m van der Horst。2021。持续集成性能测试中的变点检测。（2021）。https://repository.tudelft.nl/islandora/object/uuid: b9ef4b8e-a18e-40cb-b222-a4221cb22431 理学硕士论文。[20] Xingyu Zhao、Radu Calinescu、Simos Gerasimou、Valentin Robu 和 David Flynn。2020。用于运行时概率模型检测的区间变点检测。载于第*35*届*IEEE/ACM*自动化软件工程国际会议。[21] Yuxuan Zhao、Dmitry Duplyakin、Robert Ricci 和 Alexandru Uta。2021。云性能变异性预测。载于*ACM/SPEC*国际性能工程会议伴随活动 （线上活动，法国）（*ICPE '21*）。计算机协会，美国纽约州纽约市，35–40。https: //doi.org/10.1145/3447545.3451182