Demonstration-Free: Towards More Practical Log Parsing with Large Language Models

CONTENTS

目录

研究背景

背景的三个部分

02

传统方法的局限性

01

日志解析

03

目前使用LLM方法的进展

日志解析

```
/* A logging statement from Spark:
   spark/storage/BlockManager.scala */

logError(s"Failed to report $blockId to master;
giving up.")
```

Raw Log Message

17/08/22 15:50:46 ERROR BlockManager Failed to report rdd_5_1 to master; giving up.

Structured Log

Headers	Date	17/08/22
	Time	15:50:46
	Level	ERROR
	Component	BlockManager
Content	Template	Failed to report <*> to master; giving up.
	Parameters	[rdd_5_1]

日志解析是将原始日志解析为结构化格式，提取出模板以便后续的自动化分析（如异常检测）

传统方法的局限性

现有的日志解析方法利用日志的语法和语义模式来识别和分离静态文本和动态变量。

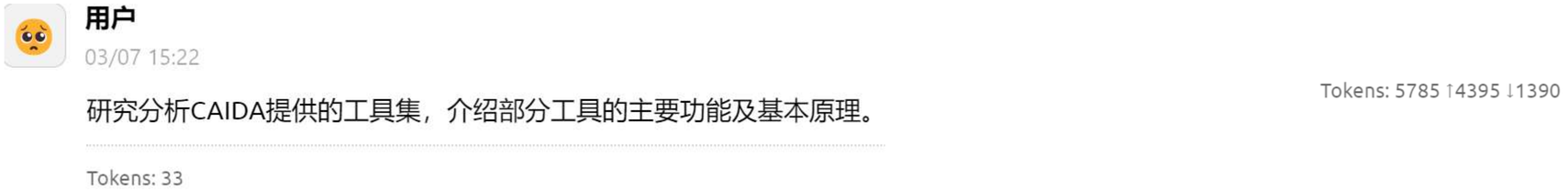
基于语法的日志解析器
严重依赖构建的规则和领域知识，因此在遇到以前未见过的日志模式时效率低下

基于语义的日志解析器
需要一定的训练开销，例如从头开始训练模型或使用标记数据微调预训练语言模型，这些都是稀缺且昂贵的

目前使用LLM方法的进展

潜力
LLM 凭借其强大的**上下文学习能力**，能够理解日志的结构和语义信息，从而生成高质量的结构化日志模板
LLM 可以处理多样化的日志格式和语言，减少对特定领域知识的依赖，具有一定的**通用性**

局限
尽管有效，但这些基于 LLM 的日志解析器仍然无法满足日志解析的实际应用。



概念解释：tokens　　示例

提出问题

过度依赖示例

问题一

过度依赖示例

大语言模型并非专门用来处理日志，因此需要示例来提高正确性。

[Instruction] I want you to act like an expert in log parsing. I will give you a log message wrapped by backticks. Your task is to identify all the dynamic variables in logs, replace them with {variables}, and output a static log template. Please print the input log's template wrapped by backticks.

[Query] Log message: `Created local directory at /opt/hdfs/nodemanager/usercache/curi/appcache/application_1485248649253_0147/blockmgr-70293f72-844a-4b39-9ad6-fb0ad7e364e4`

[Demo 1]

Log message: `Starting executor ID 5 on host mesos-slave-07`

Log template: `Starting executor ID {variables} on host {variables}`

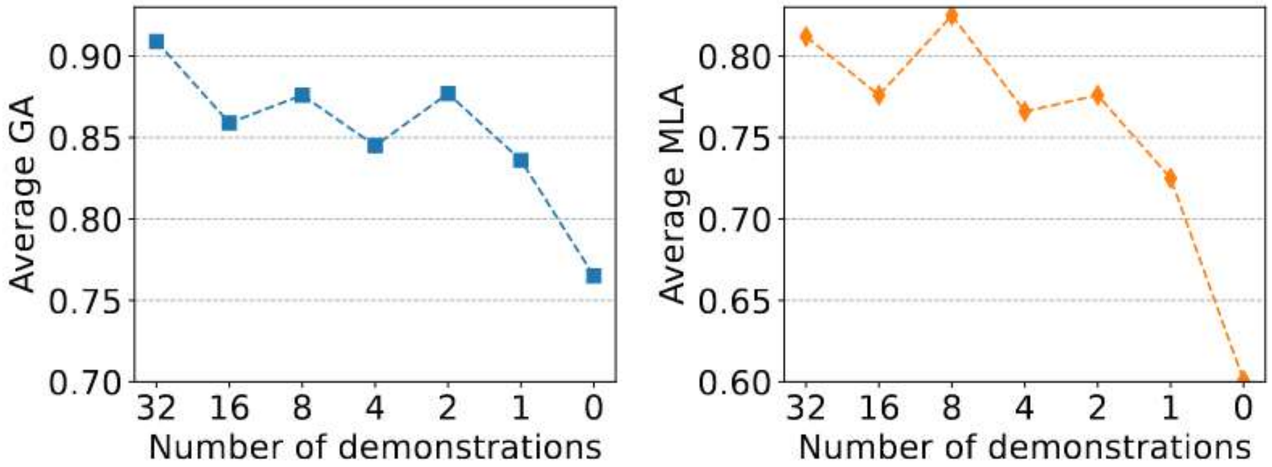
[Demo 2]

Log message: `Connecting to driver: spark://CoarseGrainedScheduler@10.10.34.11:48636`

Log template: `Connecting to driver: spark://{variables}`

Input	Output
[Instruction] [Query]	Created local directory at {directory_path} ✓
[Instruction] [Demo 1] [Query]	Created local directory at {variables}/blockmgr-{variables} ✗
[Instruction] [Demo 2] [Query]	Created local directory at {variables} ✓
[Instruction] [Demo 1] [Demo 2] [Query]	Created local directory at {variables}/blockmgr-{variables} ✗

过度依赖示例



LILAC性能与示例数量的关系图

组准确率（Group Accuracy, GA）
一条日志消息被视为“正确解析”，当且仅当它与其他日志消息分组的结果与 **真实标签（ground truth）** 一致
消息级准确率（Message-Level Accuracy, MLA）
与 GA 不同，MLA 更关注每条日志消息的解析结果是否与真实标签完全匹配，而不涉及分组问题

GA 侧重于分组的一致性，而 MLA 侧重于逐条解析的精确性

调用成本

问题二

模型调用成本

日志数据过于庞大，调用模型的成本过大

成本包含:
输入包括提示词和示例以及日志内容
输出为日志解析后的模板

若提高示例的数量来提高准确性，那么token的消耗将更多

标准时段价格 (北京时间 08:30-00:30)	**百万tokens输入（缓存命中）(4)**	**0.5元**	**1元**
	百万tokens输入（缓存未命中）	**2元**	**4元**
	百万tokens输出 (5)	**8元**	**16元**

DeepSeek API调用价格

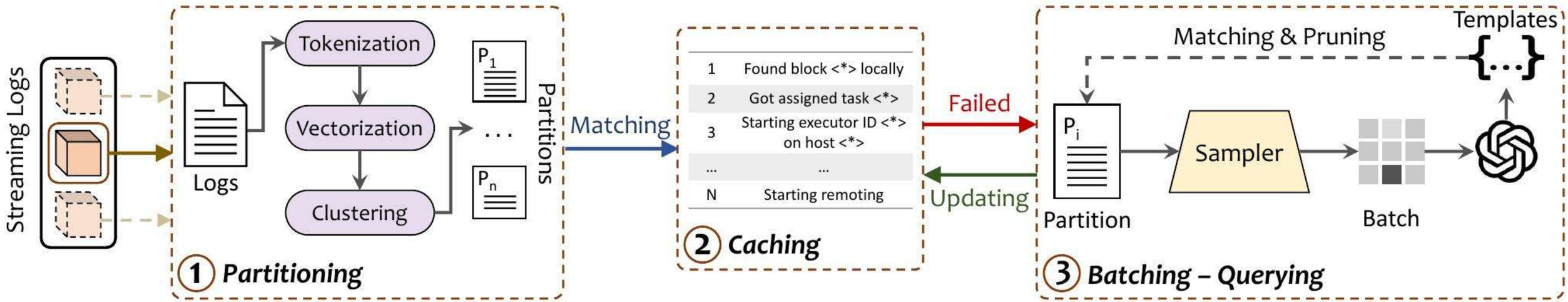
解决问题

存在的问题 | 问题解决措施

LogBatcher的贡献

贡献一	贡献二	贡献三
提出我们所知的第一个基于 LLM 的**无需示例**的日志解析框架	引入了一种特定于日志的提示策略，为 LLM 提供一批日志来处理	实验评估在准确性和 LLM 调用成本方面都优于最先进的baseline
无演示	批处理	高性能 低成本

LogBatcher的具体实现



日志分区（Partitioning）

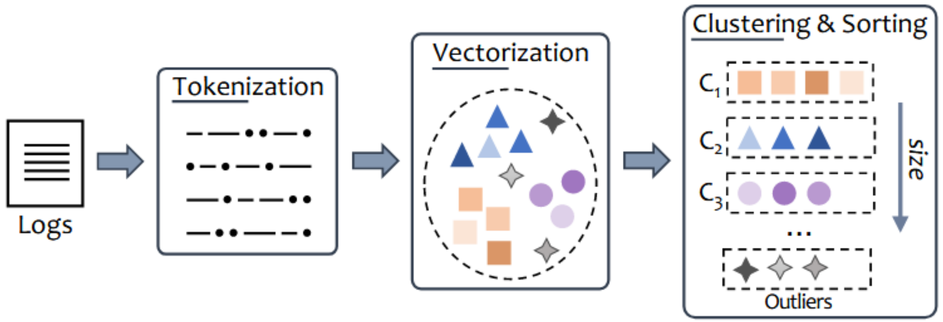
目的：确保分配给同一分区的日志具有共同点

1.令牌化（Tokenization）
指将字符串根据特定分隔符（如空格、符号）拆分为令牌（tokens） 的过程

例如，“=”可以作为分隔符
日志“START: tftp pid=16563”可以被令牌化为“START”、“:”、“tftp”、“pid=16563”

但是，如果 “=”出现在 URL 中，如“after trim url = https://www.google.com/search?q=test”，
就会导致错误分隔
我们通过屏蔽类似于参数的标记（如数字、IP 地址和 URL）来提高聚类性能

日志分区（Partitioning）



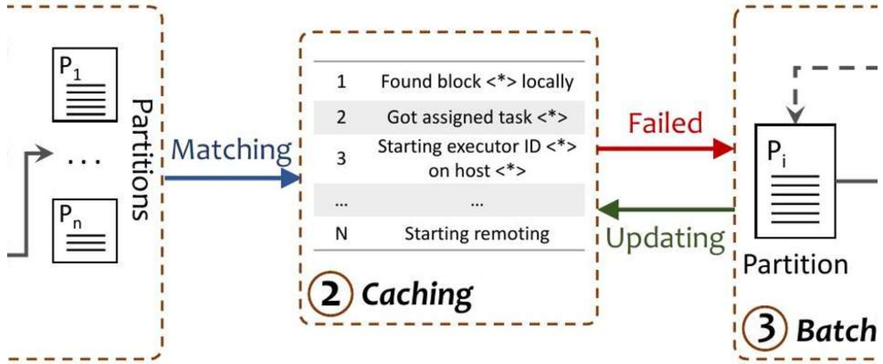
2.向量化（Vectorization）

目的：将日志数据转换为数值格式，以便用于聚类算法
方法：使用 TF-IDF对日志进行向量化

3. 聚类与排序（Clustering & Sorting）
目的：得到日志分组
方法：使用DBSACAN，按大小降序对聚类进行排序，并将所有异常值视为一个单独的聚类，最后进行处理

缓存机制（Caching）

缓存机制的实现



缓存机制存储先前解析的日志模板，并将它们与当前分区中的日志进行匹配
缓存实际上就是模板集合

匹配过程：

在日志分区后，首先尝试将当前日志与缓存中的**已解析模板**进行匹配
如果匹配成功，则直接返回模板，无需调用 LLM

动态排序：

缓存中的模板会根据出现频率进行动态排序，优先匹配高频率的模板，进一步提高匹配效率

批处理 - 查询（Batching – Querying）

属于同一模板的日志不仅包含高频出现的关键词（Tokens），其动态部分也呈现丰富的变化性
这种日志数据的特性已在实践中被广泛观测到，并被多数数据驱动的日志解析方法所采用
然而，最近基于LLM的日志解析器忽略了这些特征，导致LLM对示例过度依赖

为了解决这个问题，我们提出了一种**批处理查询方法**，
为LLM提供输入日志中的共性和可变性，以实现无演示的日志解析

批处理（Batching）

我们从每个分区中采样一组日志，形成 LLMs 的输入批次

多样性采样：

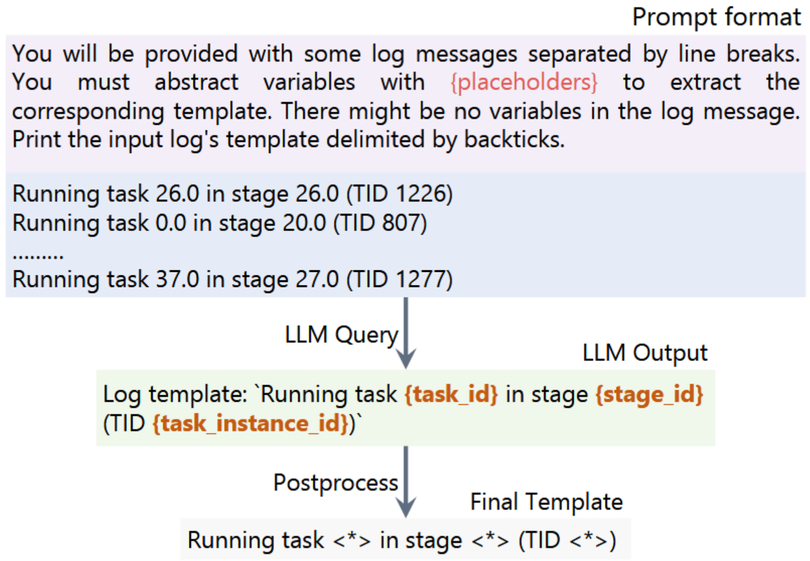
为了确保输入批次中包含共性和多样性，采用基于多样性的采样方法
计算所有日志对的 TF-IDF 向量之间的余弦相似度，形成相似度矩阵
使用 Determinantal Point Process (DPP) 算法选择日志，确保采样日志的多样性最大化

输入批次：
最终形成的输入批次包含共性（由聚类引入）和变异性（由多样性采样引入），帮助 LLM 更好地关联任务描述与输入日志，提高解析准确性

分区 1：

"Connection established from [IP1]"
"Connection established from [IP2]"
"Connection established from [IP3]“
…….

提示词设计（Prompting Design）



本文的提示词只包含指令和询问，不含示例
所以简要描述了日志解析的目标，以及输入和输出的格式

输入批次中的日志具有可变性和共性，便于LLM理解日志模板

后处理（Post-Processing）
匹配与剪枝（Matching & Pruning）

后处理是对 LLM 输出结果的优化步骤，旨在过滤冗余信息并统一模板格式

过滤冗余信息，通过定位符和占位符 {placeholder}，可以轻松提取所需的模板
为了确保每个系统的标签样式一致，使用**启发式规则**

对于聚类的结果，通常会出现两个常见问题：
1.属于**同一模板**的日志被分组到**不同的集群**中
2.属于**不同集群**的日志被错误地分组到**同一个集群**中

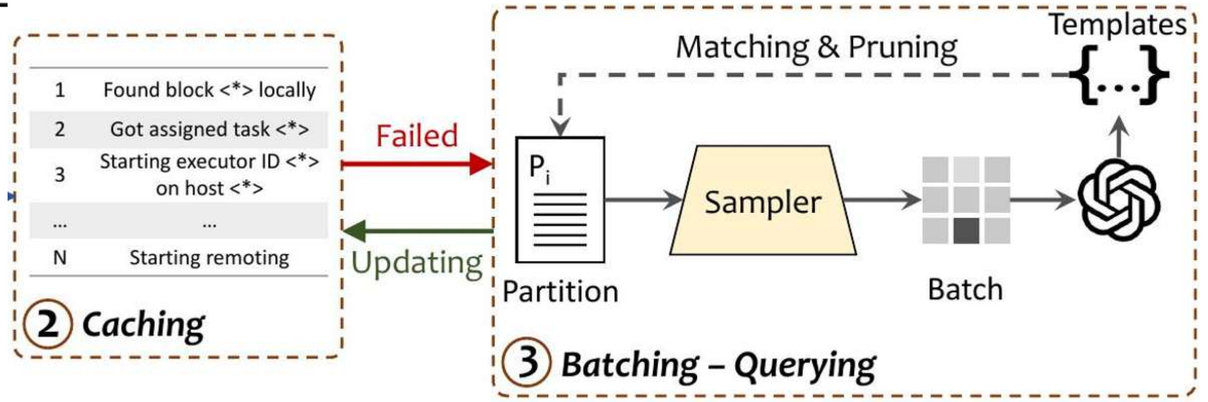
前面提到的方法有效地解决了第一个问题（采用缓存机制）
对于第二个问题，我们使用匹配与剪枝方法

匹配剪枝算法

Algorithm 1: Matching & Pruning

Input: C: Parsed Cluster
T: Identified Template
Output: C_{new}: New Cluster

```
1 regex ← convertToRegex(T)
2 C_new ← ∅
3 foreach log ∈ C do
4     if match(log, regex) then
5         C ← C \ {log}
6         C_new ← C_new ∪ {log}
7     end
8 end
9 return C_new
```



实验评估

评价指标

组准确率（Group Accuracy, GA）
一条日志消息被视为“正确解析”，当且仅当它与其他日志消息分组的结果与**真实标签**一致

消息级准确率（Message-Level Accuracy, MLA）
与 GA 不同，MLA 更关注每条日志消息的解析结果是否与真实标签完全匹配，而不涉及分组问题

GA 侧重于分组的一致性，而 MLA 侧重于逐条解析的精确性

编辑距离（Edit Distance,ED）
是一种衡量两个字符串相似度的指标，定义为将一个字符串转换为另一个字符串所需的最小 编辑操作数（插入、删除、替换字符）
在日志解析场景中，ED 主要用于评估生成的日志模板与真实模板之间的差异

与baseline的对比

Table 1: Comparison with the state-of-the-art log parsers

	Drain			AEL			Brain			Logram			*DivLog*			LILAC w/o ICL			*LILAC*			LogBatcher		
	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED	GA	MLA	ED
HDFS	0.998	0.999	0.999	0.998	0.999	0.999	0.998	0.959	0.997	0.930	0.961	0.993	0.930	0.996	0.999	**1.000**	0.943	0.999	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Hadoop	0.948	0.613	0.882	0.869	0.606	0.901	0.950	0.158	0.751	0.694	0.195	0.708	0.683	0.744	0.915	0.958	0.843	0.928	0.991	**0.958**	**0.986**	**0.989**	0.888	0.953
Spark	0.920	0.398	0.963	0.905	0.398	0.952	0.998	0.376	0.950	0.470	0.296	0.915	0.738	0.960	0.983	0.998	0.806	0.990	**0.999**	**0.983**	**0.998**	0.998	0.973	0.989
Zookeeper	0.967	0.799	0.981	0.965	0.800	0.981	0.989	0.779	0.987	0.956	0.805	0.970	0.955	0.979	0.998	0.989	0.374	0.886	**1.000**	0.987	**0.999**	0.995	**0.988**	0.995
BGL	0.963	0.479	0.885	0.957	0.474	0.883	**0.996**	0.426	0.891	0.702	0.282	0.785	0.736	0.950	**0.990**	0.941	0.870	0.955	0.983	**0.972**	0.989	0.994	0.950	**0.992**
HPC	0.887	0.662	0.872	0.904	0.680	0.880	0.945	0.660	0.973	**0.978**	0.751	0.870	0.935	0.980	0.997	0.911	0.641	0.913	0.970	**0.994**	**0.999**	0.953	0.946	0.995
Thunderbird	0.957	0.180	0.941	0.945	0.180	0.943	0.971	0.060	0.932	0.554	0.097	0.826	0.234	0.879	0.978	0.957	0.852	0.960	**0.984**	**0.913**	**0.983**	0.897	0.838	0.950
Windows	0.997	0.466	0.948	0.691	0.158	0.840	0.997	0.463	**0.976**	0.694	0.141	0.915	0.710	0.715	0.903	0.694	0.020	0.692	0.696	**0.685**	0.897	**0.997**	0.644	0.871
Linux	0.422	0.217	0.750	0.405	0.205	0.745	0.358	0.176	0.770	0.186	0.125	0.684	0.484	0.620	0.935	0.298	0.344	0.903	0.298	0.422	0.926	**0.995**	**0.974**	**0.990**
Android	0.885	0.750	**0.972**	0.773	0.540	0.876	0.960	0.253	0.924	0.795	0.436	0.822	0.737	0.677	0.952	0.931	0.481	0.890	0.953	0.627	0.923	**0.967**	**0.791**	0.955
HealthApp	0.901	0.375	0.749	0.893	0.368	0.744	**1.000**	0.261	0.871	0.833	0.677	0.850	0.876	0.984	0.997	0.901	0.866	0.945	0.998	**0.988**	**0.998**	0.984	0.980	0.970
Apache	**1.000**	0.978	0.996	**1.000**	0.978	0.996	**1.000**	0.984	0.996	1.000	0.972	0.995	0.984	0.985	0.997	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.999
Proxifier	0.765	0.704	0.980	0.826	0.690	0.972	0.527	0.704	0.945	0.531	0.477	0.816	0.531	0.993	**0.999**	0.730	0.512	0.941	**1.000**	0.995	0.999	**1.000**	**1.000**	**1.000**
OpenSSH	0.789	0.594	0.919	0.547	0.729	0.965	**1.000**	0.287	0.948	0.802	0.928	0.960	0.749	0.987	**0.993**	0.492	0.439	0.910	0.753	0.805	0.983	**0.996**	**0.975**	0.987
OpenStack	0.224	0.105	0.693	0.249	0.034	0.718	0.492	0.112	0.937	0.315	0.071	0.724	0.220	0.437	0.873	0.717	0.400	0.815	**1.000**	0.977	0.991	0.968	**0.984**	**0.993**
Mac	0.814	0.392	0.896	0.765	0.284	0.835	**0.949**	0.383	0.902	0.759	0.359	0.843	0.712	0.549	0.898	0.834	**0.626**	**0.915**	0.805	0.562	0.892	0.857	0.543	0.881
Average	0.840	0.544	0.902	0.793	0.508	0.889	0.883	0.440	0.922	0.700	0.473	0.855	0.701	0.839	0.963	0.834	0.626	0.915	0.902	0.867	**0.973**	**0.974**	**0.904**	0.970

数据集来自**Loghub-2k**

成本对比

Ttotal：解析数据集时所有调用消耗的token总数
Tinvoc：每次调用消耗的平均token数

Table 2: Efficiency of LLM-based Log parsers (#tokens)

	T_{total}			T_{invoc}		
	DivLog	LILAC	LogBatcher	DivLog	LILAC	LogBatcher
HDFS	706689	**4793**	6101	353	**342**	436
Hadoop	491299	33519	**15695**	246	308	**141**
Spark	409054	10587	**4993**	205	294	**156**
Zookeeper	360558	14858	**6930**	180	310	**122**
BGL	420346	35061	**15692**	210	297	**139**
HPC	288954	9179	**5076**	144	255	**110**
Thunderbird	522583	43546	**18781**	261	283	**107**
Windows	461847	14459	**5721**	231	295	**114**
Linux	444417	28972	**9161**	222	287	**82**
Android	430660	32574	**17967**	215	256	**115**
HealthApp	351681	16393	**6980**	176	264	**94**
Apache	364608	1549	**950**	182	258	**158**
Proxifier	494994	6724	**4160**	247	480	**347**
OpenSSH	490727	7921	**4206**	245	317	**162**
OpenStack	686037	14924	**13798**	343	364	**337**
Mac	581290	109260	**51002**	291	339	**155**
Average	469109	24020	**11701**	235	309	**173**

鲁棒性对比

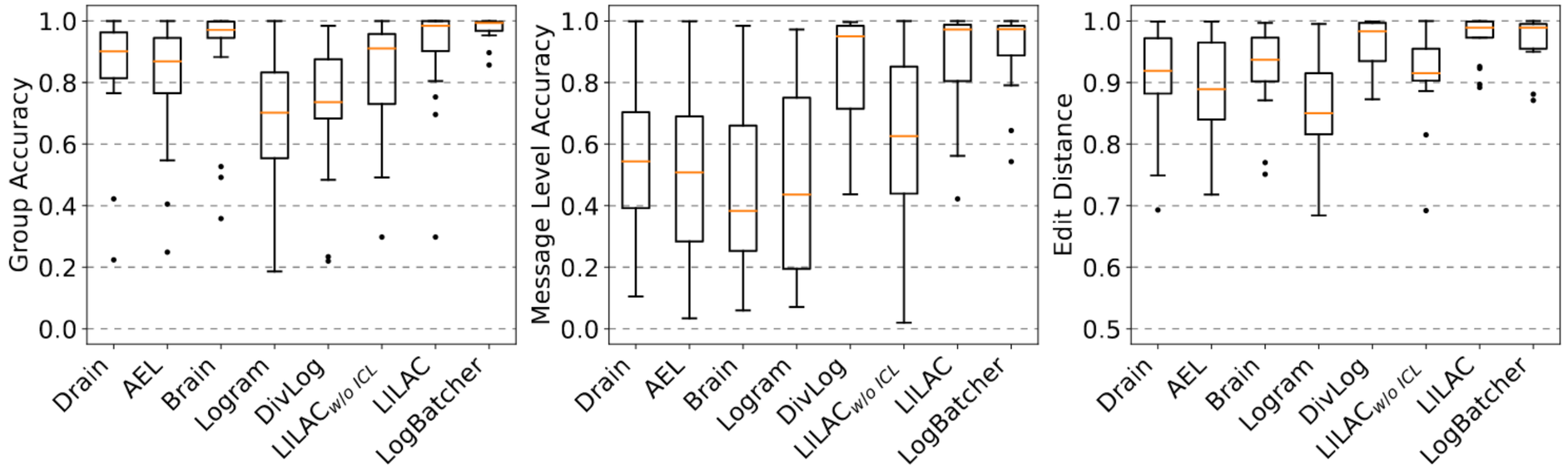


Figure 7: Robustness comparison between baselines and LogBatcher

消融实验

Table 3: Ablation study results

	GA	MLA	ED
Full LogBatcher	0.974	0.904	0.970
w/o$_{\text{partitioning}}$	0.790$_{(\downarrow 23.3\%)}$	0.770$_{(\downarrow 17.4\%)}$	0.896$_{(\downarrow 8.3\%)}$
w/o$_{\text{caching}}$	0.830$_{(\downarrow 17.3\%)}$	0.803$_{(\downarrow 12.6\%)}$	0.935$_{(\downarrow 3.7\%)}$
w/o$_{\text{batching}}$	0.928$_{(\downarrow 5.0\%)}$	0.724$_{(\downarrow 24.9\%)}$	0.910$_{(\downarrow 6.6\%)}$

分区是对三个分量之间的分组准确性影响最大的分量，如果不进行分区，LLM 输入包含的共性较少

缓存机制通过共享大分区的解析知识避免重复处理，移除后小分区无法复用信息，整体性能显著下降

批处理 当提供的数据没有多样性时，LLM 的效果会降低。这表明批量处理模块对于 LogBatcher 在精确匹配方面实现高解析精度至关重要

不同设置对LogBatcher的影响

Table 4: Results with different clustering methods

	GA	MLA	ED
LogBatcher	0.974	0.904	0.970
w/ Hierarchical clustering	$0.934_{(\downarrow 4.3\%)}$	$0.810_{(\downarrow 11.6\%)}$	$0.944_{(\downarrow 2.8\%)}$
w/ Meanshift clustering	$0.943_{(\downarrow 3.3\%)}$	$0.864_{(\downarrow 4.6\%)}$	$0.960_{(\downarrow 1.0\%)}$

不同的聚类方法

Table 5: Result with different sampling methods

	GA	MLA	ED
LogBatcher	0.974	0.904	0.970
w/ random sampling	$0.963_{(\downarrow 1.1\%)}$	$0.890_{(\downarrow 1.6\%)}$	$0.964_{(\downarrow 0.6\%)}$
w/ similarity sampling	$0.937_{(\downarrow 3.9\%)}$	$0.831_{(\downarrow 8.8\%)}$	$0.953_{(\downarrow 1.8\%)}$

不同的采样方法

不同设置对LogBatcher的影响

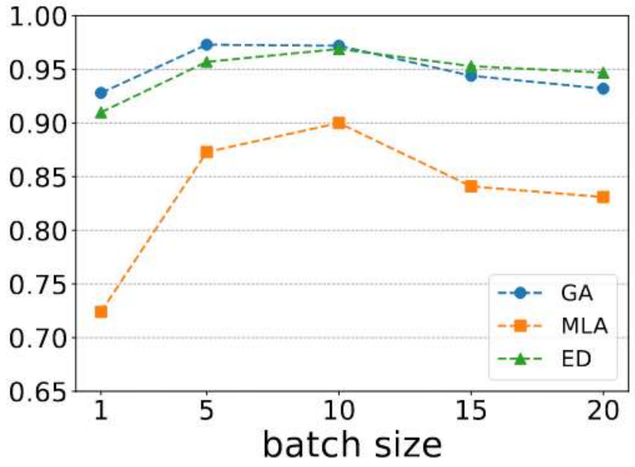


Figure 9: Average accuracy with different batch sizes

发现最佳批量大小介于 5 到 10 之间
当 batch size 超过 10 时，LogBatcher 的性能会略有下降
考虑到较大的批处理大小会导致更高的 LLM 调用开销，在我们的实验中，我们将默认批处理大小设置为 10

LogBatcher 的实用性

Table 6: Comparison with LILAC on large-scale datasets from Loghub-2.0

		LILAC w/o ICL					LILAC					LogBatcher				
Dataset	#Log entries	GA	MLA	ED	T_{total}	T_{invoc}	GA	MLA	ED	T_{total}	T_{invoc}	GA	MLA	ED	T_{total}	T_{invoc}
HDFS	11,167,740	**1.000**	0.948	0.999	**5036**	**187**	**1.000**	**0.999**	**1.000**	16128	343	**1.000**	0.948	0.994	10276	214
Hadoop	179,993	0.915	0.840	0.960	42120	183	0.872	**0.832**	**0.947**	77464	335	**0.947**	0.802	0.937	**28685**	**117**
Spark	16,075,117	0.998	0.862	0.996	37027	177	**1.000**	0.973	0.995	76335	321	0.974	**0.997**	**0.999**	**34889**	**129**
Zookeeper	74,273	0.997	0.351	0.891	14270	172	**1.000**	0.687	0.936	22787	253	0.988	**0.934**	**0.995**	**8591**	**101**
BGL	4,631,261	0.884	0.892	0.967	52274	177	**0.894**	**0.958**	**0.989**	88829	261	0.952	0.864	0.957	**47556**	**114**
HPC	429,987	0.984	0.648	0.891	**10447**	**171**	0.869	0.705	0.906	15384	237	**0.990**	**0.983**	**0.993**	15986	188
Thunderbird	16,601,745	0.725	0.556	0.878	901863	197	**0.806**	0.558	**0.928**	491934	354	0.802	**0.564**	0.879	**159588**	**111**
Linux	23,921	0.802	0.693	0.963	60368	171	**0.971**	0.767	0.959	81110	234	0.953	**0.876**	**0.974**	**34056**	**91**
HealthApp	212,394	0.993	0.551	0.811	20709	171	**1.000**	0.729	0.879	30575	251	0.983	**0.970**	**0.989**	**13015**	**81**
Apache	51,977	0.997	0.965	0.981	5349	173	**1.000**	**0.999**	**1.000**	7417	256	0.997	0.972	0.991	**2914**	**91**
Proxifier	21,320	0.000	0.114	0.854	**2681**	**192**	**1.000**	**1.000**	**1.000**	3787	344	**1.000**	**1.000**	**1.000**	3540	322
OpenSSH	638,946	0.745	0.349	0.947	5164	178	0.690	**0.941**	**0.997**	11080	308	**0.925**	0.866	0.946	**3999**	**118**
OpenStack	207,632	**1.000**	0.485	0.944	**9456**	**197**	**1.000**	**1.000**	**1.000**	17915	373	**1.000**	0.980	0.994	14484	315
Mac	100,314	0.769	0.529	0.877	**116508**	197	**0.877**	**0.638**	**0.930**	219922	309	0.795	0.595	0.896	118264	**172**
Average	3,601,187	0.844	0.627	0.926	91662	182	0.927	0.842	0.962	82905	299	**0.950**	**0.882**	**0.967**	**35417**	**155**

在Loghub-2.0 中的大规模数据集上与LILAC进行比较

对有效性有威胁的因素

数据泄露

1.LLM 预训练阶段的数据泄露LLM 在训练过程中可能 **意外记忆了日志模板的真实信息**，导致评估时模型性能因利用预训练知识而产生偏差 。

2.上下文学习中示例的泄露在**上下文学习**阶段，若提示中包含了标注的真实模板，可能导致 LLM 直接泄露这些敏感模板信息 。

最近的研究表明LLM直接记忆模板的可能性很低，且本文不需要示例，所以数据泄露可能性很低

真实数据的质量

Loghub-2k来自LogPai,由人工标注并经过校正，Loghub-2.0是大数据集。结果证实 LogBatcher 对这两组数据集都有效。

随机性

（1）LLM 的随机性（2）实验过程中引入的随机性，包括批次中日志的排列和日志的随机采样。

为了减轻 LLM 输出的不稳定性，我们将 LLM 的温度设置为 0。为了减轻后一种威胁，我们重复实验五次并报告平均结果。

总结

为了克服现有日志解析器的局限性，我们提出了 LogBatcher，这是一种基于 LLM 的经济高效的日志解析器，它不需要训练过程或标记演示。LogBatcher 利用日志数据的潜在特征，并通过对一组日志进行批处理来减少 LLM 推理开销。我们对公共日志数据集进行了广泛的实验，结果表明 LogBatcher 对于日志解析是有效且高效的。我们相信这种无需演示、无需训练且经济高效的日志解析器有可能使基于 LLM 的日志解析更加实用。

谢谢观看

THANKS

★ ★ ★