arXiv:2305.05671v1 [cs.DB] 8 May 2023

使用学习模型对 ASCII 记录进行并行外部排序

Ani Kristo 布
朗大学
ani@brown.edu

Tim Kraska 麻
省理工学院
kraska@mit.edu

摘要

外部排序是大规模数据库系统中许多操作的核心，例如对大型结果集进行排序和聚合查询、构建索引、排序合并连接、删除重复项、分片以及记录聚类。与内存排序不同，这些算法需要与操作系统和文件系统协同工作，以高效利用系统资源并最大限度地减少磁盘 I/O。

本文介绍 ELSAR：一种使用基于学习型数据分布模型的创新范式的并行外部排序算法。该算法利用这一模型，将输入记录排列到相互排斥、单调且等深的分区中；这些分区排序后，只需将其连接起来即可形成输出。这种方法完全消除了多路文件合并的需求，而多路文件合并通常用于外部排序。

我们针对多种存储介质上的均匀分布和倾斜数据集进行了全面的基准测试，测量了 ELSAR 及其他排序算法的排序速率、规模可扩展性和能效。我们发现，在固态硬盘上，ELSAR 的排序速率比次优的外部排序工具（Nsort）最高快 1.65×；在英特尔® Optane™ 非易失性内存上，其排序速率比 GNU coreutils 的 sort工具最高快 5.31×。此外，在 SortBenchmark 最具能效的外部字符串排序算法评选中，ELSAR 以高达 41% 的优势超越了当前的胜出者。这些结果进一步证实，新颖的学习增强型算法能够显著优于传统算法。

1 引言

外部排序是大规模数据库系统中许多操作的核心。在关系型数据库中，它最显然的用途是处理大型结果集的 ORDER_BY 和 GROUP_BY 查询[23, 32]。然而，外部排序也用于内部操作，例如构建索引[22, 31], 、排序归并连接[2], 、删除重复项，以及分片和记录聚类[33]。外部排序也是搜索引擎及其他文档检索系统的基础组成部分；这类系统使用倒排文件将记录映射到其在磁盘上的位置[39]。此外，它还用于基于约束的系统；这类系统内存有限，必须依赖持久化存储来执行任务[26]。因此，提高外部排序的速度能够对大规模数据处理系统产生深远影响。

外部排序的主要瓶颈是磁盘访问所带来的延迟，其速度比动态随机存取存储器慢一个数量级。与内存排序不同，这些算法需要与操作系统和文件系统协同工作，以高效利用系统资源并尽量减少磁盘 I/O。出现在顶级排序基准测试中的大多数外部排序算法[7, 13, 37, 44] 都遵循外部归并排序[3]范式。从概念上说，这些方法首先将输入划分为较小的文件，使其大小足以完全放入内存。然后，使用快速排序或其他内存排序算法，分别并行地对这些中间文件进行排序。最后，已排序的文件通过一个包含每个文件中最小键的堆进行合并。该算法通过弹出堆的根节点，持续将记录追加到输出文件中。

与此不同，本文介绍了一种并行外部排序算法（ELSAR），它采用了一种基于学习型数据分布模型的创新范式。与外部归并排序不同，它采用分区并连接方法。ELSAR 首先基于键样本构建一个估算输入分布的模型。随后，它利用该模型将输入记录排列为互不重叠、单调且等深的分区。这样，模型便无需执行归并，ELSAR 只需依次连接各个分区，其速度远快于多路文件归并。虽然在概念上与基数排序相似，但基于模型的分区对偏斜不那么敏感，并能为工作线程实现均衡负载。

对于内存排序，ELSAR 使用 LearnedSort 算法[16]，该算法已在许多具有不同分布的数据集中展现出高性能。请注意，仅将 LearnedSort 用作外部归并排序框架中的内部例程，并不会显著提升性能，因为外部排序受磁盘 I/O 限制。该算法的大部分执行时间都花在与磁盘之间执行读写操作上；因此，只有与 I/O 相关的改进才能显著提高算法的吞吐量。ELSAR 的设计目标是实现无锁并发、负载均衡和优化文件 I/O。

目前，ELSAR 支持对 ASCII 字符串进行排序，因为它们仍是最常见的数据类型，而且多个系统原生只支持 ASCII 字符[19, 21, 42]。我们尚未支持 Unicode，因为这需要专门的处理函数[1]，不过我们考虑在未来的工作中将 ELSAR 扩展至 Unicode。

为了评估 ELSAR，我们测量了它在各种机器、数据分布和输入规模下的排序速率。我们使用配备多种存储技术（即硬盘驱动器、固态硬盘和持久内存）的桌面级和服务器级机器，这些机器具有不同的 I/O 带宽和接口（SATA、PCIe 和 DDR-T）。我们还测试了 ELSAR 在 TB 级均匀和倾斜数据集上的性能，这些数据集的规模最高可达内存容量的 40× 倍。我们的实现表现出显著的性能提升：在固态硬盘上，其排序速率最高比次优的外部排序工具 Nsort 高 1.65×，在英特尔® Optane™ 非易失性内存上，最高比 GNU coreutils 的 sort 工具高 5.31×。在我们评估的各种情况下，这些结果始终如一，进一步印证了一个观点：融入学习机制的新型算法能够为传统算法带来卓越的性能提升。

最后，我们还使用 SortBenchmark 的 JouleSort 指标评估了 ELSAR。该指标用于评估外部字符串排序算法的能效[36]。截至本文发表时，ELSAR 的表现比当前获胜者高出 41%；这一结果无疑令人印象深刻，

[1]例如，由于字典序取决于语言，因此 Unicode 字符串没有统一的排序定义[14]（例如，在德语中是 ö<z，而在瑞典语中是 ö>z）。

我们已正式将这一结果提交给 SortBenchmark 委员会，供其考虑纳入下一次官方排名。

本文余下部分将：

(1) 介绍一种使用学习模型对 ASCII 记录进行排序的新型并行外部排序算法；(2) 描述我们如何将 ASCII 字符串映射到数值空间，以建模其分布和经验累积分布函数（CDF）；(3) 分析这一新排序算法的复杂度；(4) 展示基准测试，说明 ELSAR 在各种硬件组合下的性能；(5) 讨论 ELSAR 相较于其他外部排序算法的优势。

下一节将介绍外部排序现有方法的背景、各种变体及其差异。

2 背景与现有方法

外部排序算法通常分为两个阶段：1 2 [7, 13, 20, 30, 37, ）运行 41, 44] 创建阶段和）归并阶段。在运行创建阶段，排序算法将输入文件划分为多个块，这些块也称为归并段。通常会并行执行此操作，以提升文件读取性能。该阶段的目标是创建中间文件，其中包含足够小、可完全装入内存的归并段。第一阶段结束时，每个中间文件都会使用内存排序算法进行排序，其中最常用的是快速排序[7, 20, 30, 37]。根据可用内存，可以并行对多个归并段进行排序，因为它们之间不存在依赖关系。

在第二阶段，归并段会按照类似于内存归并排序算法的方式归并到输出文件中。这种算法模板称为外部归并排序，也是 GNU Coreutils 中 sort 工具所采用的算法 [7]。

2.1 外部归并排序实现的变体

外部归并排序有许多旨在提高算法性能的变体，例如内存排序例程的选择、I/O 操作调度以及归并策略。表1总结了下文讨论的方法。

内存排序变体。 最简单的变体是选择用于对所创建归并段进行排序的内存排序例程。例如，一些算法使用 Tim 排序，这是一种结合了内部归并排序算法和插入排序的混合算法。其原理在于，Tim 排序利用输入数据中自然存在的模式，并且在现实世界数据集上的性能优于快速排序[27]。Tim 排序首先在输入中查找已经预排序的元素序列，并将它们归并以形成全序。当找不到足够大的序列时（通常是超过 32 个元素），它会使用插入排序，将连续的无序元素组成排序后的序列。NADSort[44] 和 SparkSort[45]等算法将 Tim 排序用作内部排序例程。

I/O 调度变化。外部排序算法通常受磁盘 I/O 限制，因此，诸如选择内部排序例程之类的计算方面改进，对解决这个更为重要的问题帮助不大。最成功的外部排序策略着重于：1）尽量减少 I/O 操作的数量；2）尽可能通过与计算重叠来隐藏 I/O 延迟。例如，当前 SortBenchmark[37], 的优胜者之一 KioxiaSort 通过采用流水线方法来解决这一问题。借助互斥机制，它让三分之一线程的读取操作与第二个三分之一线程的排序操作重叠，并让最后三分之一线程的写入操作与之重叠 [37]。同一基准测试的前几届优胜者 FuxiSort[43] 和 DEMSort[34], 也采用了类似策略。除了让操作重叠之外，这些算法还通过生成不含记录负载的较小中间文件来减少 I/O 量。相反，算法只将原始输入文件中的记录键和索引写出[37]。

归并策略变化。 另一方面，在外部归并排序算法的第二阶段，如何归并中间文件有几种主要方案。第一种策略是执行多路外部归并：算法维护一个堆，其中存放每个文件中最小的键，并将堆顶元素写入输出。不过，这种方法受内存可容纳元素数量的限制，从而限制了算法可使用的中间文件数量以及第一阶段的并行程度。此外，由于堆大小较大，向堆中插入元素的开销也会很高。而且，这种归并例程只能顺序执行，因此会成为算法的瓶颈。

另一种方法是在归并阶段不使用堆，而是进行分层二路外部归并（例如使用锦标赛树）。这种方法不仅能减少该阶段的内存占用，还能让外部归并的较低层并行执行。不过，这会增加磁盘 I/O 开销、产生额外的中间文件，并且需要使用代价高昂的同步原语（即互斥锁）。

因此，第三种也是最佳的方法是采用混合归并例程：分层多路归并兼具较小堆和并行性的优点。这种方法的代价较低，只需为归并例程额外使用少量中间文件，并且层级之间所需的同步变量也更少。例如，KioxiaSort 采用两阶段归并：第一阶段并行执行 6 次 200 路归并，第二阶段再执行 6 路归并，将结果写入输出文件[37]。

2.2 Nsort

Nsort 是一种颇为有趣的外部排序算法，应用广泛，排序吞吐量也非常高[25]。Nsort 被用作多个 SortBenchmark 获奖方案的软件层，例如 RezSort[35],TaichiSort[18],NTOSort[6], EcoSort[1],FlashSort[4], 和 FAWNsort[28]。因此，探究自 2007 年以来 SortBenchmark 获奖方案成功的秘诀非常引人入胜。作者提到，该算法具有“复杂的缓冲区管理机制，可使计算与 I/O 重叠”，并且“特别注重处理器缓存局部性，以充分利用高速微处理器”[24]。然而，我们撰写本文时，其源代码仍为专有代码，而且我们对该算法没有更多了解。

表 1：部分外部排序算法的特性概述。

算法名称	框架类型	内存排序	合并策略	以……闻名
Unix sort	外部归并排序	归并排序	二叉堆	Linux 中的默认文件排序
Nsort	未知	未知	未知	许多获胜的 SortBenchmark 参赛作品
铠侠排序	外部归并排序	快速排序	分层 K 路	JouleSort 当前优胜者
MySQL 文件排序	外部归并排序	内省排序	分层 K 路	MySQL 中使用的外部归并排序变体
Postgres 元组排序	外部归并排序	快速排序	K 路	Postgres 中 Ext MS 的变体
SQLite vdbesort	外部归并排序	归并排序	分层 K 路	SQLite 中 Ext MS 的变体
ELSAR	分区与拼接	LearnedSort	N/A	本文描述的算法

尽管存在众多外部排序算法，但显然它们无一脱离运行-归并范式。事实上，算法层面的创新寥寥无几，大多数工作都集中在构建硬件配置，以利用现代高速块存储、处理器和网络适配器。下一节将介绍一种不同的排序方法：借助学习型 CDF 模型的优势，该方法完全无需归并阶段。

3 排序大于内存容量的文件

ELSAR 是一种由学习增强、基于数据分布的外部排序算法，它利用小型、高精度且快速的机器学习模型对 ASCII 数据集进行排序。该算法结合了多种技术以实现较高的排序速率，例如基于采样的分布学习、针对 ASCII 键的数值嵌入，以及并行、缓冲和无锁文件 I/O。ELSAR 的设计融入了 I/O 优化，可避免瓶颈，同时最大限度地利用内存和可用的硬件并行性。ELSAR 的核心思想是将输入记录组织到彼此互斥、单调且等深的分区中；这些分区排序后，只需将它们串接起来即可形成输出文件。为了快速且准确地完成这一任务，ELSAR 使用 CDF模型来近似输入数据集的经验分布函数。借助该模型，算法只需查看输入文件中每条记录的键，无需进行

与其他记录进行任何比较。这种方法比基数分区方案更能确保分区大小均匀，而后者容易受到数据倾斜的影响。

图 1 展示了 ELSAR 的示意图，下面的章节将详细介绍排序过程。

3.1 模型训练

CDF模型可以说是ELSAR的核心组件。模型使用递归模型索引（RMI）架构，根据输入数据中的一个小样本（X）进行训练，具体说明见 [16], 和 [15]。该样本从线程 T_0读取的第一个批次中均匀随机抽取。样本大小必须足够大，以便模型能够充分学习分区的边界；同时又不能过大，以免模型训练变得代价高昂。在 [16] 中，我们观察到，CDF模型能够很快达到满足分区任务要求的准确度。因此，即使样本较小，这种方法也能很好地发挥作用。在我们的ELSAR实现中，我们通过经验观察到，取数据量的1%作为样本即可满足这些约束。

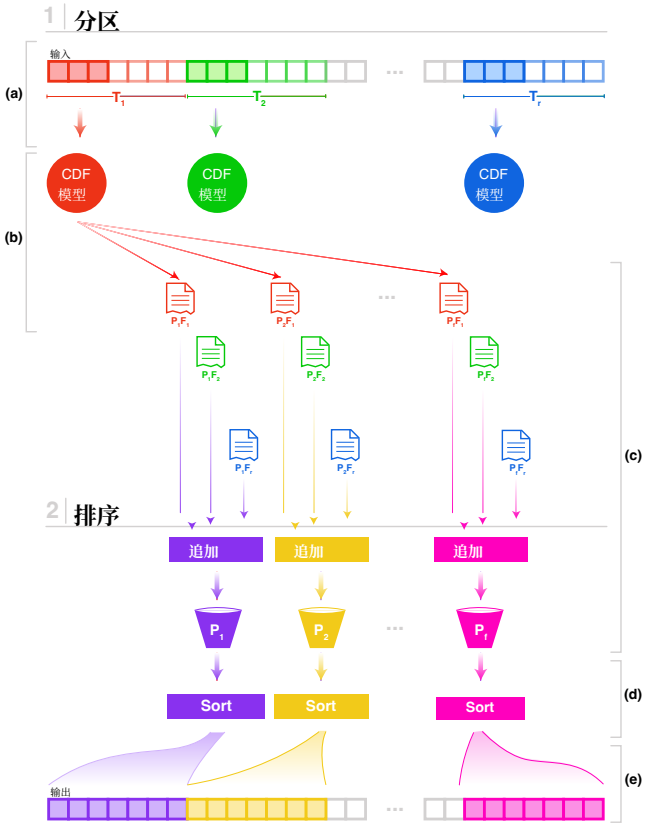

图1：ELSAR的架构。分区期间，r 多个并行线程以批次为单位读取输入文件中不同的数据条带。每条记录都会被放入 f 线程本地分区片段中，然后刷新到中间文件。分区由相应片段文件的内容组成。分区使用LearnedSort在内存中排序 [16]，并直接写入输出文件中的对应偏移位置。

另一方面，RMI 结构是一种由线性回归模型构成的无环图式排列，充当专家系统。从根节点开始，输入中的记录键（$x \in X$）会遍历内部节点，这些节点会递归地选择另一个线性模型，该模型针对当前键所属的某个值子集更加“专业化”。当键到达叶节点时，

输出是一个预测的累积分布函数值 $y \in [0,1]$，对应于$y = P(X \leq x)$，也就是该记录在估算总体中的百分位排名。

这个CDF模型兼具高精度、训练速度快和推理时间短等优点。从某种意义上说，该模型的作用类似于保序哈希函数或基数分区器。然而，与它们不同的是，CDF模型能更好地吸收输入偏斜，并生成大小均匀的分区，这对于排序线程的负载均衡至关重要[16]。这是因为训练过程会将高密度的定义域区域分配给RMI中的更多节点，从而将偏斜分散到分布更加均匀的桶中，理想情况下，这些桶的作用类似于等深直方图桶。

需要注意的是，我们关注的是对经验分布进行建模，也就是说，对观测到的数据分布进行建模，而非理论分布。两者的区别在于，观测到的样本并不遵循一种平滑分布，无法仅用少数几个线性模型来建模。数据在细粒度尺度上会呈现阶跃函数的特征，包含更多结构、噪声和不规则性。因此，有必要采用更复杂的架构，以捕捉经验分布的细微特征。

“学习型索引结构的理由”[15] 详细阐述了架构、CDF模型的训练算法、推理以及进一步分析。

3.2 输入处理

训练完成后，算法会启动 r个线程，每个线程负责从输入文件中读取一个特定且不重叠的记录范围（图 1a）。在我们的实现中，我们使用了 OpenMP 线程，原因有四：(1) 相较于 pthreads，OpenMP 的可移植性更好；(2) 线程池可以最大限度地降低线程创建和销毁的开销；(3) 可自动调度；(4) 支持 CPU 亲和性。每个线程以批次为单位读取输入记录（键和有效载荷）。这种技术旨在尽可能利用顺序 I/O，从而受益于其相较于随机访问 I/O 更高的读取速度。尽管如此，在任何排序过程中，记录洗牌的模式都不可预测，因此几乎无法进行顺序读取和写入。为进一步加快读取速度，我们也可以使用异步线程（或协程），在读取线程处理当前批次的同时，预先读取下一批次。然而，我们发现，由于同步开销和并行性降低，这种扩展并未提升性能。此外，我们也没有观察到使用直接 I/O 能显著提升性能，因此该算法依赖 libc 的缓冲文件实用程序。

3.3 分区

接下来，算法通过训练好的CDF模型处理批次中的键，该模型预测记录在输入的所有记录中的秩（图1b）。每个读取线程 T_i 都有自己的模型只读副本，并维护一组 f线程本地分区片段，这些片段的作用类似于等深直方图桶（即，P_1F_i，P_2F_i，..... P_fF_i）。线程利用模型的预测结果，将记录放入相应的片段中。对于第 j个读取线程中的第 P_jF_i 个片段 i，算法维护以下单调性不变量：

$$\{x \leq y \mid \forall x \in P_jF_i, \forall y \in P_{j+1}F_i, \\ i \in [1,f], j \in [1,f-1]\} \tag{1}$$

完成第一次批处理后，线程会将其分区片段刷新到临时文件中。对于后续批次，只需将片段的内容追加到现有的对应临时文件中即可。请注意，每个读取线程维护的是其线程本地分区片段，而不是直接将它们追加到一个大型分区文件中。这样可以避免使用开销较大的互斥锁和锁。另一方面，在此过程中，算法只使用指向记录的指针，以减少内存复制和移动操作，尤其是对于较长的字符串。最后，由于读取线程拥有彼此互不相交的工作集，我们可以使用不加锁版本的读取操作和写入操作（即 fread_unlocked() 和 fwrite_unlocked()）执行文件 I/O。这些函数省去了文件指针的锁检查，因此速度更快。

我们在大型偏斜数据集上将这种分区方法与基数分区进行了比较，并观察到CDF模型提供了更好的均匀性。基数分区查看键中最重要的字节，并将其转换为分区文件的索引。这种方法类似于构建等宽直方图，其中每个区间对应于键域中的一个固定区间，该区间与其他区间具有相同的宽度。相比之下，ELSAR中使用的基于模型的分区会生成等深分区，其键区间大小可变，但区间大小均匀。根据我们的实验，与基数方法相比，我们的方法将分区大小方差降低了23%。这将为排序线程提供更好的负载均衡，使其能够在下一阶段处理分区文件。

3.4 排序

处理完整个输入文件后，ELSAR 会启动 s 个线程，每个线程负责排序并刷新一个分区。该数量 s 的计算方式为可同时驻留在内存中的分区最大数量。请注意，分区数量经过选择，以确保没有任何单个分区超出内存容量。此外，这也使多个分区能够同时驻留在内存中，从而实现并行排序。

排序线程会读取所有属于其所分配逻辑分区的片段文件，并将其中的记录追加到一个大型缓冲区中（图 1c）。

然后，算法会调用 LearnedSort 作为内存排序例程（图 1d）。这是一个绝佳的选择，因为与许多现代排序算法相比，它的内存排序速率最高[17]。它实现了多种优化，包括提高 CPU 缓存利用率、处理倾斜输入，以及通过采用提前终止策略避免在重复值较多的数据集上出现性能下降[17]。

3.5 写入输出

最后，在分区内容完成排序后，它们会与相邻分区的记录依次拼接，形成一个连续的输出文件（图 1e）。每个排序线程都会为输出文件维护一个打开的文件描述符，并且对于要刷新的每个分区 P_i，都会定位到预先计算好的偏移位置

该位置是各分区大小之和 $P_1..P_{i-1}$:

$$write\_offset = \sum_{k=1}^{i-1} len(P_k)$$

由于排序缓冲区仅包含键和指向记录的指针，因此无法通过一次顺序调用刷新整个缓冲区。因此，线程首先按排序顺序取消引用指针，将记录分批合并。然后，对合并后的缓冲区（通常为 100KB）执行缓冲式顺序写入，从而优化写入性能。

4 ASCII 记录编码

为了学习输入数据的分布，必须将 ASCII 记录投影到一个数值空间中，以便 CDF模型使用线性回归训练其单个节点。因此，在内存中，该算法对键的数值编码和指向记录的指针进行操作。

假设键具有固定大小，则使用键中每个字符的二进制值计算编码，并将其表示为 95进制数，因为可打印ASCII字符的编码介于 32 和 127 之间。不可打印ASCII字符（0-31）是用于向外围设备（例如打印机）提供元信息的控制码，因此与排序无关。因此，长度为 l（$1 \leq i \leq l$）的键中位置 x_i 处字符的编码为 $(\text{ASCII}(x_i) - 32) \times 95^{l-i}$。那么，整个键的数值编码为:

$$\sum_{i=1}^{l} \left((\text{ASCII}(x_i) - 32) \times 95^{l-i}\right)$$

如果输入包含可变长度键，则 l设置为观测到的最大长度（$l = \max(len(x_1, x_n), \ldots, len(x_i))$，$i \geq len(x_i)$，以及 () 的 ASCII 值。

使用 64位基本类型时，我们最多可以编码键的第九个字节，因为对于 $l = 9$，$\log_2 \sum_{i=1}^{l} 95^{l-i+1}$ 为 59。14；对于 $l = 10$，则为 65。71。如果记录键的长度超过九个字节，此编码方案将无法以数值形式表示其余字节。只有在分区数超过 6。37·10$_{17}$且键至少为 10 个字节时，这才会违反公式 1 中的不变量，也就是说，我们排序的输入至少为 6369.5 PB。在其他所有情况下，这都不会影响排序算法的正确性。这是因为内存中的LearnedSort例程包含一个修整步骤，它使用 strncmp() 函数进行成对键比较，对键的其余部分执行最后阶段排序。此外，这个修整步骤也会处理累积分布函数模型的预测误差。该步骤使用插入排序，对近乎有序数组的处理速度接近线性[16]。

5 实现

ELSAR算法的完整伪代码如算法 1 所示。该算法的输入包括 I-输入文件名、 O - 输出文件名、 B - 批次大小、 f - 分区 / 片段的数量、 r - 处理器数量，以及 M - 系统中的可用内存。

第 1 行，算法会创建一个稀疏输出文件，其占用的字节数恰好与输入文件 I 相同。

第 2 行，算法会调用模型训练函数，对输入文件进行采样，并返回一个训练好的累积分布函数模型（即 $F_X(x)$）。

在第 3 行，算法初始化一个由动态大小数组（向量）组成的二维矩阵（$r \times f$）。该矩阵将存储读取到内存中的记录，这些记录按读取线程组织在第一维，按分区片段组织在第二维。

在第 4 行，算法初始化一个临时文件二维矩阵（$r \times f$），这些临时文件对应于在 D 中收集的记录。

在第 5 行，算法创建一个大小向量，用于跟踪分配给每个分区的记录数量。

在第 6–20 行，算法首先生成 r 个读取线程，每个线程都从不同的偏移量（o）打开输入文件，以进行并行读取。请注意，每个线程负责从输入文件中读取 $|I|/r$ 条记录。每个读取线程 i 以大小为 B 的批次（$\mathbf{C}$）读取其负责的记录，同时跟踪目前已读取的字节数（b）。随后，对于每个批次中的每条记录，算法使用 F_X（x）预测器估计键（p）的秩，并将记录放入预测的片段（$\mathbf{D}_{i,\ p}$）中。为执行此调用，算法使用上一节所述的方法对 ASCII 键进行编码。记录放入片段后，相应分区的大小（即 S_p）便会递增。

在第 21 行，算法进入第二阶段，首先计算 s -要使用的排序线程数量。请注意， s 的值是可同时完全驻留在内存中的连续分区的最大数量。

在第 22-31 行中，算法会生成 s 个工作线程，用于处理所有 f 个分区。因此，每个排序线程将负责 f/s 个分区文件。此处，算法还会初始化一个内存缓冲区，用于存放属于这些分区的所有记录；截至目前，这些记录仍分散在先前生成的 r 个临时片段文件中。每个排序线程 i 都会读取其正在处理的分区所对应片段的内容（$\mathbf{F}_{j,i}$），并将其汇集到一个内存缓冲区中以供排序（$\mathbf{P}$）。将每个片段完整读取到内存后，其临时文件便会关闭，随即通知操作系统删除该文件并释放内存。然后，排序线程会调用 LearnedSort 内部排序例程，对分区内容进行排序。此调用会在内部将每条记录的键重新编码为数值表示，并使用该表示对键进行排序，如 [17] 所述。最后，排序线程会将整个分区刷新到输出文件中各自的偏移量 o 处，其中 o 是所有分区大小的总和 $[0, i - 1]$。此并行写入例程只是按顺序串接各个分区。每个排序线程处理完其对应的分区后，算法便会退出，此时输出文件中已完整写入已排序记录。

6 计算复杂度

本节将针对输入大小（n）对 ELSAR 进行工作量—跨度分析。并行随机访问机（PRAM）[12] 模型中的工作量—跨度框架与顺序算法的复杂度分析类似。不过，两者的区别在于，该框架将计算视为有向任务依赖图。工作量 指算法执行的操作总数，它代表最坏情况下顺序计算复杂度的下界。另一方面，跨度表示在假设并行性不受限时，因数据依赖而无法进一步缩短的最长顺序操作序列（即

算法 1： ELSAR 的伪代码

输入： I - 输入文件名
输入： O - 输出文件名
输入： M - 可用内存
输入： r - 处理器数量
输入： f - 分区数量
输入： B - 读取批次大小

$out \leftarrow fcreate(O,\ |I|)$
$F_X \leftarrow train\_CDF(I)$
$\mathbf{D}^{r\times f} \leftarrow vector()$
$\mathbf{F}_{r\times f} \leftarrow tmpfile()$
<escape>$\mathbf{S}^{1\times f} \leftarrow \{0\}$</escape>
对于 $i \in [0, r)$ **并行执行**
  $o \leftarrow i \cdot\ |I|/r$
  $in \leftarrow fopenat(I, \text{“}rb\text{”}\ o)$
  $b \leftarrow 0$
  当 $b < |I|/r$ **时**
    $\mathbf{C} \leftarrow fread\ (in,\ B)$
    $b \leftarrow b + B$
    对 $x \in \boldsymbol{C}$ **执行**
      $p \leftarrow F_X(encode(x))$
      $\mathbf{D}_{i,\ p} \leftarrow \mathbf{D}_{i,\ p} \cup \{x\}$
      $\mathbf{S}^p \leftarrow \mathbf{S}_p + 1$
    对 $j \in [0, f)$ **执行**
      $fwrite(\mathbf{F}_{i,j}, \mathbf{D}_{i,j})$
      $clear(\mathbf{D}_{i,j})$
  $fclose(in)$
$s \leftarrow \{x \mid \sum_{i=0}^{x} \mathbf{S}_i \le M < \sum_{i=0}^{x+1} \mathbf{S}_i\}$
对 $i \in [0, f)$ **执行** s **并行线程**
  $\mathbf{P} \leftarrow \{\}$
  对 $j \in [0, r)$ **执行**
    $\mathbf{P} \leftarrow \mathbf{P} \cup \{fread(\mathbf{F}_{j,i})\}$
    $fclose(\mathbf{F}_{j,i})$
  $LearnedSort(\mathbf{P})$
  $o \leftarrow \sum_{l=0}^{i} S_l$
  $out \leftarrow fopenat(O, \text{“}wb\text{”}\ o)$
  $fwrite(out,\ \mathbf{P})$
  $fclose(out)$

关键路径）由于数据依赖关系而无法进一步缩短，同时假设并行性不受限制[3, 12]。

算法首先会在磁盘上为输出文件分配足够的空间，其大小等于输入大小。如果底层文件系统支持稀疏文件，那么这是一个 $O(1)$ 操作（即 XFS、EXT4、NTFS、APFS）。否则，该操作相对于输入大小呈线性复杂度。接下来，ELSAR 从输入中抽取一个小样本，并训练 CDF模型。由于输入大小超过 1B 时，样本大小上限为 10M，因此从渐近角度看，这是一个常数时间操作。

然后，算法在输入读取阶段从顺序模式切换到并行模式。算法使用 r 个线程从输入中读取大小相等的范围（即，每个线程负责 n/r 条记录），并将其打乱后写入相应的分区片段。由于使用 RMI模型可以在常数时间内进行CDF预测，因此分区阶段的跨度为 $O(n/r)$，工作量为 $O(n)$。在此情况下，我们假设存储设备支持随机读写访问，且寻道时间为常数。请注意，旋转磁盘并非如此；例如，由于寻道时间不是常数，并行读取操作的工作量将增加到 $O(n(r-1)/2)$。

将每个读取线程批次中的记录放入预测的分区片段后，属于同一分区的各线程片段会追加到相应的分区文件中。分区文件总数为 f，该操作由 s 个并行线程完成，其中 s 是同一时刻能够完全保留在内存中的最大分区数。每个分区文件平均包含 n/f 条记录，而每个 s 个排序线程平均负责 f/s 个分区文件。因此，该操作的跨度为 $O(n/s)$，工作量为 $O(n)$。

每个分区文件完成后，排序线程会调用 LearnedSort。平均情况下，其复杂度与输入大小呈线性关系[16, 17]。因此，该操作的跨度为 $O(n/s)$，工作量为 $O(n)$。

最后，每个排序线程都会将已排序的分区内容刷新到输出文件中的特定偏移位置。同样，假设存储设备支持随机写入访问，则寻道时间为常数，该操作的跨度为 $O(n/s)$，工作量为 $O(n)$。否则，每个排序器的寻道时间为 $O(n(f-1)/(2s))$。

总计，跨度为 $O(n/r + 3n/s)$，工作量为 $O(n)$。

7 基准测试

在本节中，我们将评估 ELSAR 在 ASCII 数据集和各种机器配置下相较于其他流行外部排序算法的性能。首先，我们将介绍实验所使用的基准测试数据和硬件。

7.1 实验设置

基准方案。在接下来的实验中，我们将比较 ELSAR 与 GNU coreutils 的 sort 工具（又称 Unix sort）[7] 和 Nsort[25]的排序速率。Unix sort 是一种外部归并排序，代表了最常见的外部排序范式（即，创建运行段并进行归并），许多数据库管理系统（如 MySQL[20], 、Postgres[30], 和 SQLite[41]）都采用这种范式。另一方面，据我们所知，Nsort 是性能最高且可公开获取的外部 ASCII 排序工具。请注意，根据 SortBechmark 网站上展示的结果，KioxiaSort 也是一种很有竞争力的算法。然而，尽管我们多次尝试，仍未能从其作者处获得代码。

数据。 我们在单文件数据集上进行评估，这些数据集包含键和有效载荷大小固定的可打印 ASCII 字符串。键的大小为 10 字节，有效载荷的大小为 90 字节，因此可能的键值有 95_{10} 个。我们展示两个数据集上的结果：倾斜数据集和非倾斜数据集，它们由 gensort 实用程序[40]生成。对于均匀分布输入，gensort 会以相同的概率独立生成每个键字符。而对于倾斜数据集，gensort 会先生成非倾斜记录，然后修改数据集以注入一定的倾斜。为此，它维护一个包含 128 个 6 字节条目的表，并针对索引为 rec_idx的每条记录，

表2：用于评估 ELSAR 性能的机器的关键统计信息。

	中央处理器型号	中央处理器频			磁盘类型 技		内存
McGraw	英特尔® 至强® 金牌 6230	2.1GHz	40	英特尔® 固态硬盘SC2KB03	SSD	SATA III	256 GB
				英特尔® 固态硬盘PE2KX010T8	SSD	NVMe	
				希捷 ST4000NM0115-1YZ	HDD	SATA III	
				英特尔® 傲腾™ 100系列	PMem	DDR-T	
Aurora	英特尔® 酷睿™i5-12600K	3.7GHz	16(12P + 4E)	4 × WD_BLACK SN850（2TB）固态硬盘		M.2 NVMe	32GB

将键中最高有效字节替换为表中索引为 $table\_idx = \log_2(rec\_idx) \bmod 128$ 的条目。

机器。 我们使用两台不同的 Linux 机器，一台服务器级机器和一台桌面级机器。服务器级机器 McGraw 连接了多种存储设备，即硬盘驱动器、SATA 固态硬盘、NVMe 固态硬盘和英特尔® Optane™ 持久内存。这使得比较排序算法在带宽和特性各异的磁盘上的性能颇有意义。这台机器配备了一颗英特尔® 至强® Gold 6230 处理器，具有强大的并行性（40 个线程），但时钟频率并非最高。另一方面，桌面机器 Aurora 配备了一颗英特尔® Core™ i5-12600K 处理器，可实现高得多的时钟频率。该处理器采用混合架构，包含六个高性能核心和四个能效核心。Aurora 配备了 4 块 2TB NVMe 固态硬盘，每块均采用 M.2 规格，可直接连接至主板插槽。该机器拥有 32 GB 内存和 8TB 磁盘空间，因此我们可以开展可扩展性实验，测量排序算法在规模最大可达内存容量 40 倍的数据集上的性能。表 2 总结了影响排序速率的关键规格。

方法。 我们根据 ASCII 二进制顺序对记录进行排序，该顺序由 GNU C 库[8]中的 strncmp(3) 或 memcmp(3) 函数确定。需要注意的是，所有实验都仅通过文件系统模式访问磁盘数据，因此利用了操作系统的缓冲文件 I/O 机制。所有测量均使用系统中可用的最大线程数：McGraw 机器上的实验为 40 个，Aurora 机器上的实验为 16 个。对于每次测量，(1) 清空系统缓冲区，(2) 使用 Unix time 实用程序记录经过的时间，其中包括进程启动和关闭所用的时间，(3) 使用 valsort [40] 验证输出文件是否已排序及其校验和。我们报告每种算法连续运行五次的平均值。

7.2 不同存储类型

如前所述，磁盘 I/O 速度会显著影响外部排序算法，因为它们占据了大部分执行时间。因此，我们首先考察排序程序在 McGraw 机器上的性能。该机器连接了四种不同类型的存储设备：SATA 硬盘驱动器、SATA 固态硬盘、NVMe™ 固态硬盘和英特尔® Optane™ DC 持久性内存。结果如图 2 所示。对于每种存储设备，我们用红色虚线标示磁盘带宽。带宽的计算方式是：读取输入文件后，立即将其写回同一磁盘，并使用与排序算法相同数量的线程。在本节中，我们只讨论非倾斜数据集。

对于机械硬盘而言，带宽非常低，ELSAR 的排序速率为 70 MB/s。该速率低于 Nsort（110 MB/s），与 Unix sort（60 MB/s）大致相当。这是因为 ELSAR 高度依赖磁盘寻道，以最大限度地提高线程的并行性并尽量减少线程间的相互依赖。与固态硬盘不同，硬盘上的文件寻道操作会随寻道偏移量而变化。

另一方面，在 SATA 固态硬盘上，ELSAR 的排序速率为 380 MB/s，而 Nsort 为 230 MB/s，低 39%。Unix sort 的速率为 130 MB/s，比 ELSAR 低 66%。ELSAR 的吞吐量仅比磁盘带宽低 10%。

正如预期，在 NVMe™ 磁盘上，这三种算法的吞吐量都更高。ELSAR 提升了 a2.35×，Nsort 提升了 3.35×，而 Unix sort 仅提升了 1.41×。ELSAR 的排序速率为 900 MB/s，比表现次佳的算法 Nsort（760 MB/s）高 18%；而且，与之前一样，仅比磁盘带宽低 7%。

这项实验令人振奋的一点是这些算法在英特尔® Optane™ DC 持久内存（PMem）上的表现。PMem 是一种非易失性内存技术，可在存储级容量和成本下实现类似内存的性能[11]。Optane™ 设备通过 DIMM 插槽连接到主板，因此读取和写入时可达到接近内存总线速度的水平。不过，由于它们属于非易失性介质，这项技术的速度仍不及动态随机存取存储器，因此在性能层级中位于动态随机存取存储器和固态硬盘之间[38]。尽管如此，ELSAR 仍保持领先，排序速度比在 NVMe™ 上快 13%，并比 Nsort 快 9%。在这种情况下，ELSAR 与参考线之间的差距更大，因为我们尚未针对使用 libpmem.h 库调整代码；该库包含用于访问 Optane™ 驱动器的内建函数[29]。

最后一种存储类型是 RAM 磁盘，这是一种内存挂载文件系统（例如 ramfs、tmpfs），能够模拟延迟近乎为零的磁盘。这是一项有趣的实验，因为我们展示了这些算法在不再受磁盘 I/O 限制时的表现。存储介质会随着时间推移不断加快，而这项评估近似反映了高速磁盘上的排序速率。在这种情况下，ELSAR 的排序速率跃升至 1.43 GB/s，比 Nsort 高 20%，比 Unix sort 高约 7.53×，但比内存带宽低 22%。

7.3 偏斜数据集

除了不同存储类型下的性能外，图 2 还比较了每种磁盘类型在倾斜数据集上的排序速率

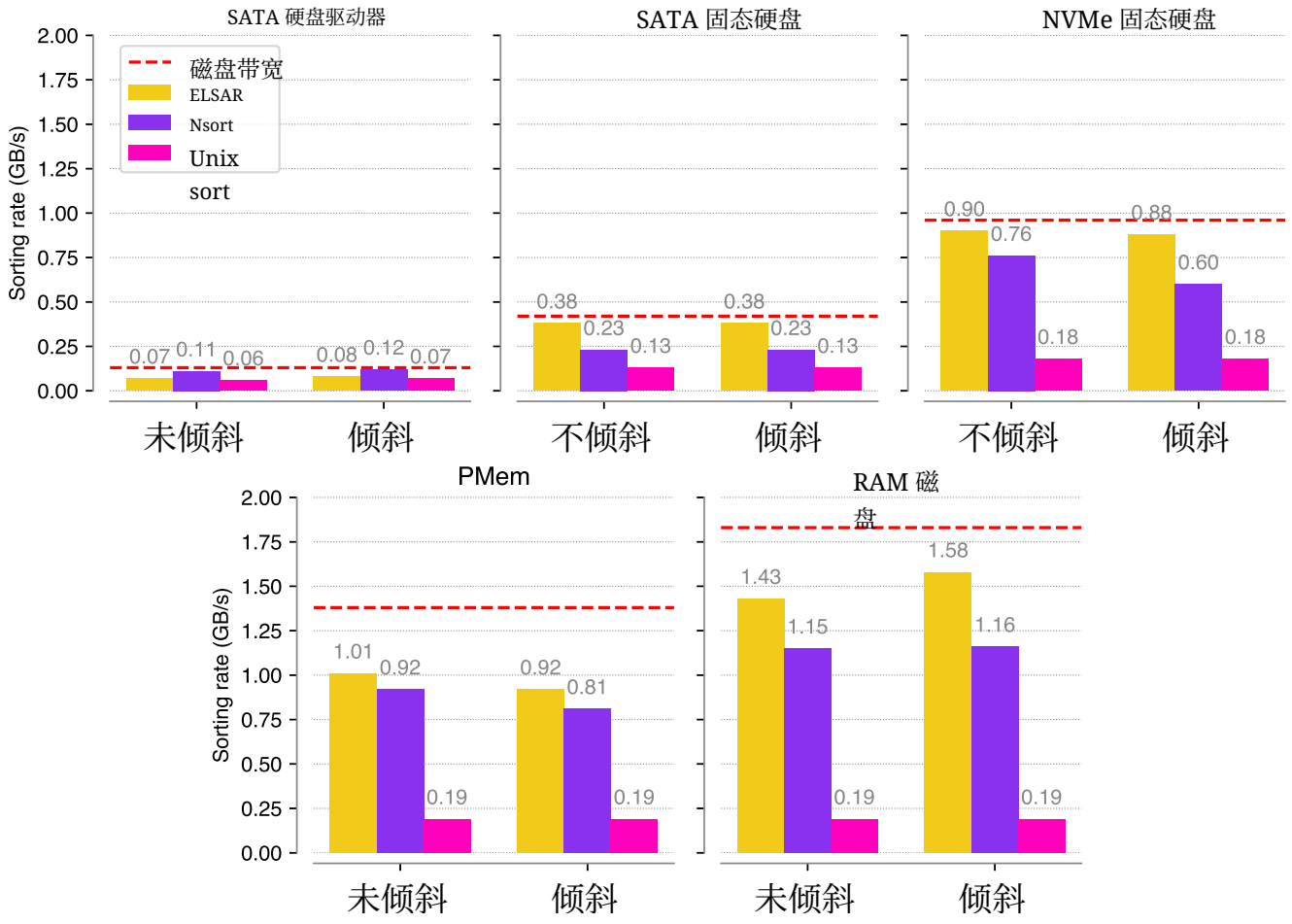

图 2：McGraw 机器上不同存储类型下 ELSAR 与其他基线方法的排序速率（越高越好）。该图还展示了均匀数据集和偏斜数据集上的性能。参考线（虚线）表示磁盘的带宽，其测量方式为完全读取输入文件后，立即将其写入同一目录所需的时间。所有这些测量均使用硬件上可用的最大并行性执行。

。倾斜数据集是使用 gensort[40] 工具和 -s 选项生成的。这会在数据直方图中产生各种“尖峰”，增大直方图箱大小的方差（见图 3）。对于倾斜数据集，区间大小的标准差从平均箱大小的 0.14% 增至平均箱大小的 65.65%（高出 ~ 468×）。

图 2 表明，尽管排序速率会受到偏斜的轻微影响，但不会出现严重的性能下降，ELSAR 能够吸收这种偏斜。原因有两点：1）CDF模型预测会创建等深分区，如[16], 中所述，因为这些分区没有固定的值范围；2）即使某些分区比其他分区更大、排序时间会更长，OpenMP 的动态线程调度也会通过为工作不足的线程分配更多分区来处理，从而平衡负载。

结果表明，对于存储设备，ELSAR 的速率平均下降了 3 %，而 Nsort 下降了 11%。由于减少了随机寻道，在硬盘上的性能对两种算法都提升了 5%。对于内存磁盘，由于算法不再受磁盘限制，ELSAR 的排序速率比非倾斜情况高 10%。这是因为输入、输出和临时文件都位于内存中，并且反复访问同几个文件能够利用页缓存局部性。

7.4 数据可扩展性

接下来，我们考察这些算法如何随输入规模的增大而扩展。因此，我们比较 ELSAR 对输入大小从 5× 到 40× 物理内存容量的排序速率。图4展示了收集到的结果。

我们在 Aurora 机器上使用 10 个硬件线程进行了这些测量。磁盘是 M.2 NVMeTM 固态硬盘，顺序读取速度最高可达 7.00 GB/s[5]。根据上一节所述的测量结果，并行读取完整输入并立即写入这些磁盘，吞吐量为 2.93 GB/s。这比.....高出近 3×

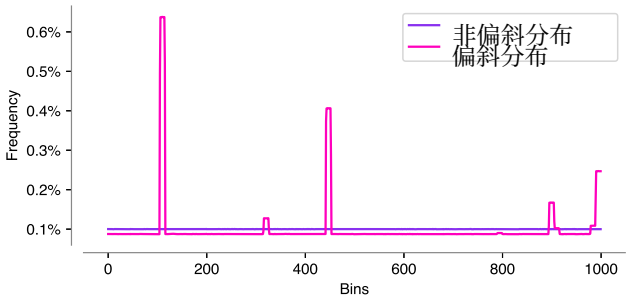

图 3：使用 gensort 生成的倾斜和非倾斜数据集的直方图。该直方图包含 1000 个分箱（x 轴），y 轴显示条目的频率，计算方式为分箱大小与输入大小的比值。对于倾斜数据集，某些直方图箱的大小可能比平均值大 6× 。请注意，分箱大小还存在较小的“尖峰”，但由于 y 轴的比例，这些尖峰并不可见。非倾斜数据集的分箱大小标准差为平均箱大小的 0.14%。对于倾斜数据集，分箱大小的标准差为平均箱大小的 65.65%。

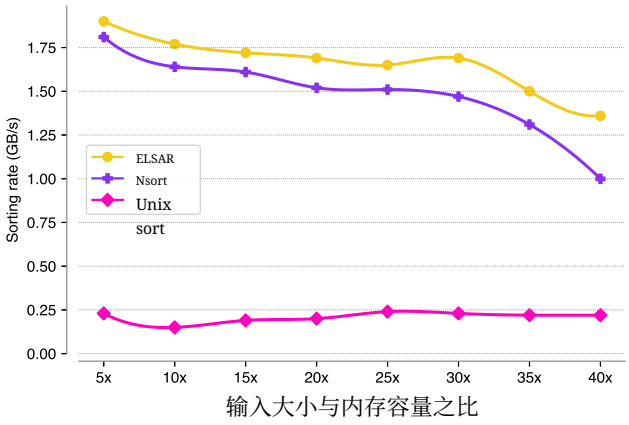

图4：可扩展性实验展示了随着输入大小逐步增大，ELSAR 与其他基线方法的性能（越高越好）。评估在 Aurora 机器上进行，使用大小从 150 GB 到 1.2 TB 不等的非倾斜数据集。Aurora 机器的平均磁盘带宽为 2.93 GB/s。

McGraw 机器所用 NVMeTM 磁盘的吞吐量，因此图 2 和图 4 中算法的实测速度存在差异。

我们观察到，ELSAR 的平均排序速率为 1.66 GB/s，Nsort 的为 1.48 GB/s，而 Unix sort 的为 0.21 GB/s。每增加 5×（即 150 GB），ELSAR 的速度平均下降 5%，而 Nsort 下降 8%。因此，当输入大小达到内存容量的 40× 时，Nsort 的吞吐量相比起始点下降了 45%，而 ELSAR 仅下降了 28%。最后，Unix sort 的性能相对稳定，每一步平均仅下降 1%，总计下降 4%。

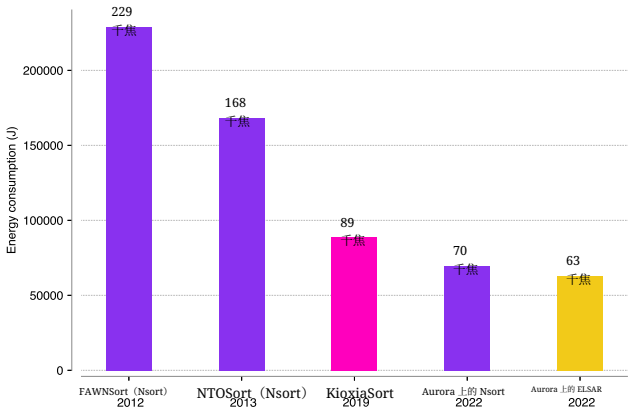

图5：ELSAR 和 Nsort 在 Aurora 机器上的能效，以及截至 2012 年的 Sort-Benchmark JouleSort 类别最近三届冠军的能效（越低越好）。ELSAR 在 Aurora 机器上的总能耗为 63 kJ，比当前基准测试冠军 KioxiaSort[37] 的 89 kJ 低 41%。KioxiaSort 是一种外部归并排序算法。FAWNSort 和 NTOSort 的提交均使用 Nsort，但运行在不同硬件上。为便于比较，我们也展示了 Nsort 在我们的硬件（Aurora）上的能耗。

7.5 能效

评估外部排序算法的另一个角度是能效。能耗按总执行时间与系统平均功耗的乘积计算。例如，在中央处理器拥有大量核心以提高并行性，或连接了许多高速磁盘以增加 I/O 带宽的系统上运行的排序算法，可能会非常快。然而，其功耗也会非常高。例如，McGraw 机器中使用的持久内存驱动器性能比 NVMeTM 固态硬盘快 1.4×，但其平均功耗额定值高出 3× [9, 10]。相比之下，在 CPU 核心和磁盘数量有限的桌面系统上，算法的速度可能较慢，但总体功耗较低。例如，典型的桌面级处理器，如拥有十个核心的英特尔® CoreTM i5，基础热设计功耗为 65 W，低于 McGraw 机器所用的英特尔® 至强® Gold 6230 处理器；后者拥有 20 个核心。因此，能耗指标非常适合用于比较不同硬件上的外部排序算法，因为它通过功耗测量同时纳入了时间和系统资源。此外，我们没有图 5 中排序算法的源代码或二进制文件，因此只能比较其报告的能耗读数。

SortBenchmark 有一个专门面向提交能效最高算法的类别——JouleSort[36], 。能效的重要性体现在：为运行这些进程的计算机系统供电所需的运营成本更低，建设足够冷却基础设施的成本更低，以及环境与可持续性问题。每年，该基准测试都会接受能够完成排序的提交。

处理 1TB 的 100 字节 ASCII 记录（键为 10 字节），同时消耗的能量少于现有记录。基准测试当前的获胜者是名为 KioxiaSort[37], 的 2019 年提交方案，该方案以 89 kJ 的能量完成了这项任务。由于衡量指标是能耗，而不仅仅是执行时间，因此允许参赛方案使用任意硬件配置，其中大多数使用桌面级机器[6, 28, 34, 37]。其他 SortBenchmark 类别（如 GraySort 和 CloudSort）包含来自腾讯[13] 或阿里巴巴[44], 等大型云服务提供商的提交方案；这些方案使用由数百台虚拟机组成的高速网络，对 100 TB 的记录进行排序。虽然我们也很希望参加这些基准测试，但看来，在这些类别中，云定价以及能否获得海量服务器，比实际使用的排序技术更为关键。图 5 总结了 SortBenchmark JouleSort 类别最近三位获胜者的能耗，以及 Nsort 和 ELSAR 在 Aurora 机器上的能耗；Aurora 是我们专门构建用于实现高能效排序的机器。

在图 5 中，第一个柱表示 FAWNSort（2012 年），它在配备英特尔® Core™ i7、8 GB 内存和 16 个固态硬盘的系统上使用 Nsort[28]。第二个柱表示 NTOSort（2013 年），它同样使用 Nsort，其系统也配备了新一代 i7 处理器和 16 个速度更快的固态硬盘[6]。第三个柱表示 KioxiaSort（2019 年），它在配备英特尔® Core™ i9 处理器、64 GB 内存和 8 个 RAID-0 NVMe™ 固态硬盘的系统上实现了其版本的外部归并排序。最后两个柱展示了 Nsort 和 ELSAR 在我们的 Aurora 机器上的能耗。

ELSAR 在 Aurora 机器上的能耗为 62,912 J，比当前基准测试的领先者低 41%。这以显著优势为基准测试任务的最低能耗纪录树立了新的候选者。

作为参考，我们还展示了 Nsort 在 Aurora 上的能耗。Nsort 与 FAWNSort 和 NTOSort 使用相同的软件，但运行于我们的硬件上。我们在自己的机器上评估 Nsort，是为了强调 ELSAR 的性能提升源自算法设计，而不仅仅是所用的硬件。Nsort 的能耗为 69,772 J，比 ELSAR 高 11%。

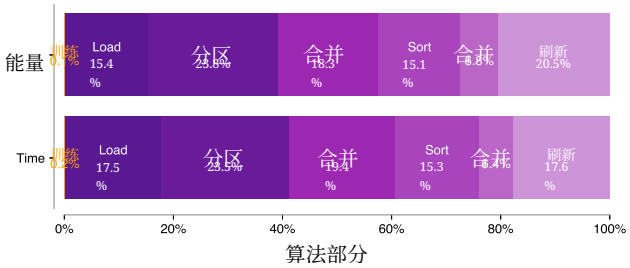

图 6：EL-SAR 各算法部分的执行时间和能耗构成。这些测量是在 Aurora 机器上进行的。各部分的能耗通过将其消耗的秒数乘以截至该部分的算法平均功率计算得出。训练时间和能耗在图中不可见，占比不到 1%（以橙色标示以便区分）。

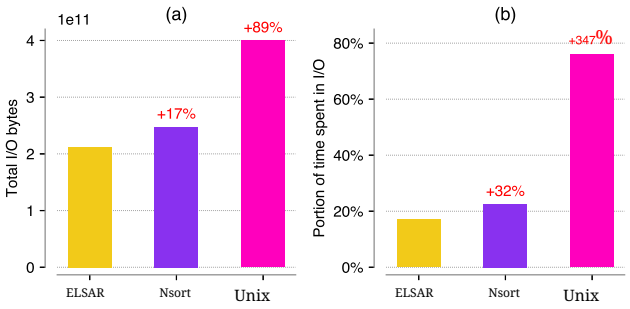

图 7：ELSAR 和其他基线算法的 I/O 性能。(a) 展示各算法的 I/O 大小，定义为写入磁盘和从磁盘读取的字节总数（越低越好）。(b) 展示这些算法执行读取和写入系统调用所花费的时间占比。

7.6 算法部分的细分

我们进一步分析 ELSAR，细分其耗时
算法在各个算法阶段所花费的时间。此外，我们还分析了各部分的能耗构成。结果如图 6 所示。

用于分区记录的训练过程是算法中占比最小的部分，占总执行时间和能耗的 <1%。另一方面，占比最大的部分是记录分区（占总时间的 23.5%，占总能耗的 23.8%）。在这一过程中，算法根据 CDF预测，将输入记录拆分到不同的片段文件中。由于记录没有特定顺序，并会导致随机写入模式，这是算法中开销最大的操作。

此外，记录合并大约占用 7% 的时间和能量。我们添加了这项优化，通过按顺序批次进行写入操作，加快算法将记录刷新到输出文件所需的时间。虽然 7% 不容忽视，但总合并和刷新所花费的时间（~ 24%）仍少于不进行合并时刷新所需的时间。

7.7 I/O 性能

最后，我们还通过考察 ELSAR、Nsort 和 Unix sort 的 I/O 工作负载，以及它们执行 I/O 所花费的执行时间占比，来剖析其 I/O 性能。这项分析使我们能够量化这些算法受磁盘 I/O 限制的程度，而磁盘 I/O 正是外部排序的主要瓶颈。为提供这一指标，我们使用 Linux 工具 strace，记录各算法及其子进程（即线程）的读取和写入系统调用大小。我们在 Aurora 机器上使用大小为 100GB 的输入进行评估。

在图 7(a) 中，我们展示了各算法的 I/O 负载，并将其定义为 strace 工具报告的写入磁盘和从磁盘读取的字节总数。与 ELSAR 相比，Nsort 的 I/O 负载高出 17%，Unix sort 则高出 89%。

另一方面，在图 7(b) 中，我们展示了这些算法在磁盘上执行 I/O 操作所花费的执行时间占比。ELSAR 将总执行时间的 17% 用于执行 I/O 操作，而 Nsort 将 23% 的时间用于 I/O，比 ELSAR 长 32%。最后，对于 Unix sort，I/O 时间占其总执行时间的 76%，比 ELSAR 高出近 3.5×。

这些结果表明，由于加载量更小，ELSAR 执行 I/O 所花费的时间更少，从而带来了更好的整体性能。

8 结论

本文介绍了一种新颖的外部排序算法 ELSAR，该算法利用学习型数据分布模型。不同于外部归并排序范式，ELSAR 以互斥、等深度且单调递增的方式对输入进行分区，从而无需执行文件合并例程。相反，它只需进行简单的文件拼接，速度显著更快。我们还在不同的机器、存储介质、分布和输入规模上，对 ELSAR 及现有外部排序算法进行了多项评估。我们始终观察到，ELSAR 的排序速率高于其他排序算法。此外，ELSAR 以 41% 的优势超越了 SortBenchmark JouleSort 纪录，成为能效最高的排序算法。

不过，这项工作仅代表了该排序算法的初步成果。未来，我们计划将 ELSAR 打造成一种高性能的分布式排序算法，使其能够处理规模达数百 TB 的数据集，并支持 Unicode 字符。

参考文献

[1] A. Beckmann、U. Meyer、P. Sanders 和 J. Singler。2010。使用固态硬盘进行节能排序。http://sortbenchmark.org/ecosort_2010_May_15.pdf [2] Mike W. Blasgen 和 Kapali P. Eswaran。1977。关系数据库中的存储与访问。*IBM Systems Journal* 16 卷，第 4 期（1977），363–377 页。[3] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein。2009。《算法导论》，第三版（第 3 版）。麻省理工学院出版社，马萨诸塞州剑桥。[4] J. D. Davis 和 S. Rivoire。2010。FlashSort：JouleSort 基准测试参赛项目及 2010 年 Daytona 10 GB 组别。http://sortbenchmark.org/flashsort_2010_Jan_01.pdf[5] 西部数据。2021。WD_BLACK SN850 NVMe™ 固态硬盘 2TB（带散热片）规格。https://www.westerndigital.com/products/internal-drives/wd-black-sn850-nvme-ssd#WDS200T1XHE[6] A. Ebert。2013。NTO Sort。http://sortbenchmark.org/NTOSort2013.pdf [7] GNU。2022。自由软件基金会，GNU coreutils/sort。https://github. co m/coreutils/coreutils/blob/master/src/sort.c[8] GNU。2022。GNU C 库。https://www.gnu.org/software/libc/manual/ html mono/libc.html[9] ® 英特尔。2018。SSD DC P4510 系列。https://ark.intel.com/content/www/us/en/ark/products/122573/intel-ssd-dc-p4510-series-1-0tb-2-5in-pcie-3-1-x4-3d2-tlc.html[10] 英特尔® 。2019。Optane™ 持久内存 200 系列简介。https: // www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html [11] 英特尔® 。2019。Optane™ 技术。https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html [12] J. JáJá。1992。《并行算法导论》。Addison-Wesley 出版公司，美国马萨诸塞州雷丁。https://books.google.com/books?id=Uom6wAEACAAJ [13] J. Jiang、L. Zheng、J. Pu、X. Cheng、C. Zhao、M. Nutter 和 J. D. Schaub。2016。腾讯排序。http://sortbenchmark.org/TencentSort2016.pdf [14] Ken Whistler 和 Markus Scherer。2021。*Unicode* 排序算法。标准。Unicode。https://unicode.org/reports/tr10/ [15] Tim Kraska、Alex Beutel、Ed H Chi、Jeffrey Dean 和 Neoklis Polyzotis。2018。学习型索引结构的理由。载于《*2018* 年国际数据管理会议论文集》。计算机协会，纽约，纽约州，美国，489–504 页。[16] Ani Kristo、Kapil Vaidya、Ugur Çetintemel、Sanchit Misra 和 Tim Kraska。2020。学习型排序算法的理由。载于《*2020* 年 *ACM SIGMOD* 会议论文集》国际数据管理大会。计算机协会，美国纽约，1001–1016。[17] Ani Kristo、Kapil Vaidya 和 Tim Kraska。2021。击败重复项：重新设计 LearnedSort 算法。https://doi.org/10.48550/arXiv.2107.03290) arXiv:arXiv:2107.03290[18] M. Liu、K. Zhang、S. Peter 和 A. Krishnamurthy。2019。TaichiSort：使用 NVMe 和 Coffee Lake 对 1TB 数据进行能效排序。http://sortbenchmark.org/TaichiSort2019.pdf [19] P. Mockapetris。1987。域名——实现与规范。技术报告 1035 。10 17487 1035 。域名系统。https://doi.org/ 。/RFC [20] 。2021 8 。甲骨文，MySQL 文件排序 filesort.cc。版本 。https://github.com/mysql/mysql- server/blob/8.0/sql/filesort.cc [21] MySQL。2022。5.7 参考手册 :: Unicode 支持。https://dev.mysql.com/doc/refman/5.7/en/charset-unicode.html[22] MySQL。2022。索引创建。https://github.com/mysql/mysql-server/blob/7ed30a748964c009d4909cb8b4b22036ebdef239/mysql-test/r/ single_delete_update.result [23] MySQL。2022。ORDER BY 子句。https://github.com/mysql/mysql-server/blob/7ed30a748964c009d4909cb8b4b22036ebdef239/mysql-test/t/single_delete_update.test [24] Chris Nyberg、Jim Gray 和 Charles Koester。2004。在一台 32P 日本电气 Windows Itanium2 服务器上运行 Nsort 的一分钟。[25] C. Nyberg 、C. Koester 和 J. Gray。2000。Nsort：面向非统一内存访问和对称多处理机器的并行排序程序。http://www.ordinal.com/NsortPara.pdf [26] Hwee Hwa Pang、Michael J Carey 和 Miron Livny。1993。内存自适应外部排序。技术报告。威斯康星大学麦迪逊分校计算机科学系。[27] T. Peters。2022。Tim 排序。https://svn.python.org/projects/python/trunk/Objects/ listsort.txt[28] P. Pillai、M. Kaminsky、M. Kozuch 和 D. Andersen。2012。FAWNSort：对 10GB、100GB 和 1TB 数据进行能效排序。http://sortbenchmark.org/fawnsort-joulesort-2012.pdf [29] PMDK。2022。libpmem 库。https://pmem.io/pmdk/libpmem/ [30] Postgres。2021。PostgreSQL，元组排序 tuplesort.c。https://github.com/postgres/postgres/blob/166f94377c886516ca986ef8a623cd2e854fe911/src/backend/utils/sort/tuplesort.c [31] Postgres。2022。哈希和 B 树索引创建。https://github.com/postgres/postgres/blob/40af10b571bdabbab06839d090083c9f9c2091d2/src/include/utils/tuplesort.h [32] Postgres。2022。ORDER BY 子句。https://github.com/postgres/postgres/blob/c30f54ad732ca5c8762bb68bbe0f51de9137dd72/src/test/regress/sql/tuplesort.sql [33] Postgres。2022。关系聚簇源代码。https://github.com/postgres/postgres/blob/4036bcbbb92839f20ed6d0db418996954dd66470/src/backend/commands/cluster.c[34] M. Rahn、P. Sanders、J. Singler 和 T. Kieritz。2009。DEMSort——分布式外部存储排序。http://sortbenchmark.org/demsort.pdf [35] Waleed Reda 和 Dejan Kostić。2021。RezSort：使用能效型 NVMe 固态硬盘对 1TB 数据进行排序。http://sortbenchmark.org/RezSort2021.pdf [36] Suzanne Rivoire、Mehul A. Shah、Parthasarathy Ranganathan 和 Christos Kozyrakis。2007。JouleSort：一项均衡的能效基准测试。载于《2007 年 ACM SIGMOD 国际数据管理大会论文集》（中国北京）（SIGMOD '07）。计算机协会，美国纽约，365 376 10 1145 1247480 1247522 – 。https://doi.org/ 。/ 。[37] 2019 1 。S. Sano、Z. Mahmoud 和 T. Suzuki。。KioxiaSort：用 89K 焦耳对 TB 数据进行排序。http://sortbenchmark.org/KioxiaSort2019.pdf [38] Anil Shanbhag、Nesime Tatbul、David Cohen 和 Samuel Madden。2020。基于英特尔® Optane^{TM} DC 持久内存的大规模内存中分析。载于《第 16 届新型硬件数据管理国际研讨会论文集》（俄勒冈州波特兰）（DaMoN '20）。计算机协会，美国纽约，文章 4，共 8 页。https://doi.org/ 10.1145/3399666.3399933[39] Muthian Sivathanu、Saurabh Goyal 和 Rajiv Mathews。2018。更新倒排索引。美国专利 10,073,874。[40] SortBenchmark。2022。gensort 数据生成器。http://www.ordinal.com/gensort. html[41] SQLite。2011。vdbesort.c。https://sqlite.org/src/file?name=src/vdbesort.c&ci=trunk[42] PHP 组。2022。PHP 字符串：手册。https://www.php.net/manual/en/ language.types.string.php [43] J. Wang、Y. Wu、H. Cai、Z. Tang、Z. Lv、B. Lu、Y. Tao、C. Li、J. Zhou 和 H. Tang。2015。FuxiSort。http://sortbenchmark.org/FuxiSort2015.pdf [44] Q. Wang、R. Gu、Y. Huang、R. Xin、W. Wu、J. Song 和 J. Xia。2016。NADSort。http://sortbenchmark.org/NADSort2016.pdf [45] R. Xin、P. Deyhim、A. Ghodsi、X. Meng 和 M. Zaharia。2014。Databricks 使用 Apache Spark 运行 GraySort。http://sortbenchmark.org/ApacheSpark2014.pdf 优化声明：性能测试中使用的软件和工作负载可能仅针对英特尔微处理器进行了性能优化。

SYSmark 和 MobileMark 等性能测试是在特定的计算机系统、组件、软件、操作和功能上进行测量的。上述任何因素发生变化，都可能导致结果有所不同。您应参考其他信息和性能测试，以协助您请全面评估您考虑购买的产品，包括该产品与其他产品合并使用时的性能。更多信息请访问 http://www.intel.com/performance。Intel、Xeon 和 IntelXeon Phi 是英特尔公司在美国和/或其他国家/地区的商标。