为什么 Iceberg 社区选择遗弃 MoR 中的 positional delete？

若您关注过 Apache Iceberg 的最新进展，可能已听闻其引入了类似 Delta Lake 的 Deletion Vector（DV），并在读时合并（MoR）表中弃用定位删除（positional deletes）。

Apache Iceberg 格式规范版本 3（format spec 3）对 MoR 表的行级删除处理机制进行了重大调整。在 Iceberg 格式规范版本 2（format spec 2）中，MoR 支持两类删除操作：

- Positional Deletes
- Equality Deletes

在 Iceberg V2 中，多数查询引擎通过 Positional Delete files 标记待删除的特定行位置。而 V3 版本弃用了这类定位删除文件，改用删除向量（deletion vectors）替代。

此变更源于定位删除在实际扩展性中暴露的缺陷，新版删除向量方案旨在解决这些问题。

使用 Iceberg V2 时，我发现定位删除是管理 Iceberg 等开放表格式中删除操作的有效方式。Apache Spark、Trino 等主流查询引擎在写入时均强力支持定位删除。相比之下，Apache Flink 则支持等值删除。

得知 MoR 表将引入删除向量（DV）并逐步弃用定位删除的消息后，身为数据工程师的我产生了深入探究的兴趣。毕竟定位删除与等值删除以往表现相当出色。

让我们共同深度解析这项公告，透彻理解其内涵。

我们将从基础概念切入。我已规划系列博客文章深入探讨此主题。但通过本篇，您将掌握 Apache Iceberg 的以下核心要点：

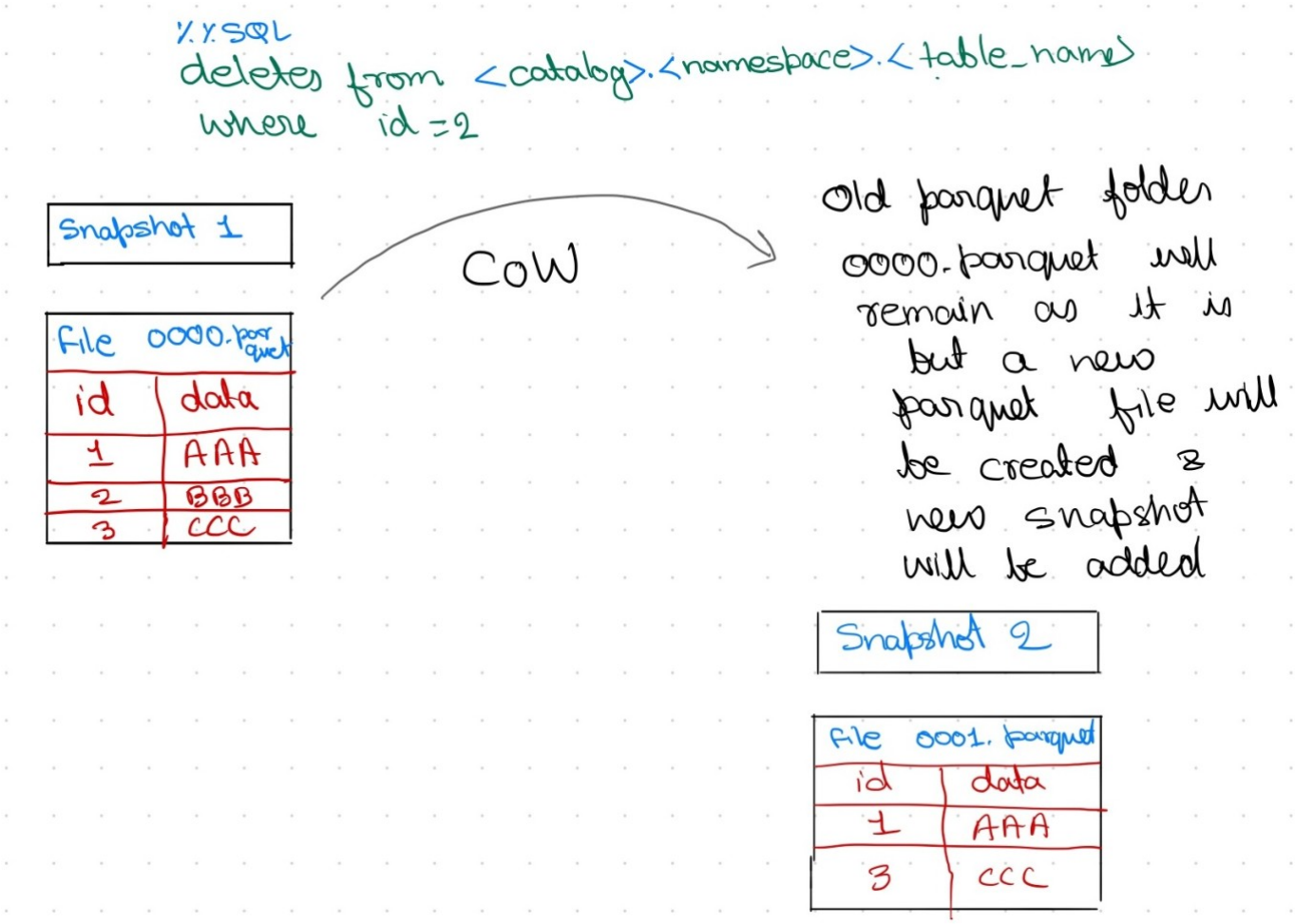
- 什么是写时复制（COW）和读时合并（MOR）？如何配置？
- MoR 中的删除类型：定位删除与等值删除
- Iceberg V2 中定位删除如何运作？
- 定位删除中分区级与文件级删除的权衡？
- 定位删除的缺陷及 V3 弃用原因
- 删除向量的引入如何优化删除机制？

提升 Apache Iceberg 读写性能的关键在于行级更新的管理策略。Iceberg 表可通过配置写时复制（CoW）或读时合并（MoR）两种策略处理行级更新。

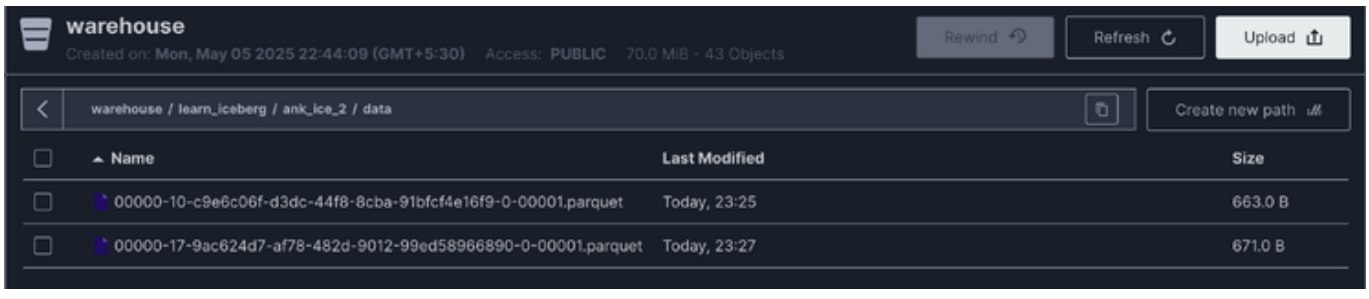
理解这些策略的内部机制，有助于您在 Iceberg 表设计初期准确定义策略，确保表长期保持高性能。

写时复制（Copy-on-Write）

- 此模式下，当 Apache Iceberg 表发生更新或删除时，相关数据文件会被复制并修改。
- 系统将创建指向新数据文件的 Iceberg 表快照，此过程使整体写入速度变慢。
- 若存在冲突的并发写入，需重试操作，进一步增加写入耗时。
- 但读取数据时无需额外处理，查询直接获取最新版本数据。
- CoW 通常用于追求更快读取速度的场景，此时写入速度可稍慢。



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：过往记忆大数据



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：过往记忆大数据

读时合并（Merge-on-Read）

- 此模式下，当对 Iceberg 表执行更新或删除操作时，系统不会重写现有数据文件。删除操作会生成删除文件（delete files），而非重写整个数据文件。
- 在读取过程中，这些删除文件会即时应用，从而过滤掉过期记录。此方法虽能减少写入放大（write amplification），但因需额外处理可能降低读取性能。
- 作为 Apache Iceberg 的消费端，查询引擎需负责将删除文件（delete parquet files）与实际数据文件（actual parquet files）合并，以提供更新后的最新数据。
- 默认情况下，Iceberg 格式规范版本 2 的表以写时复制（CoW）模式组织。若需实现读时合并（MoR），必须为表设定特定属性。

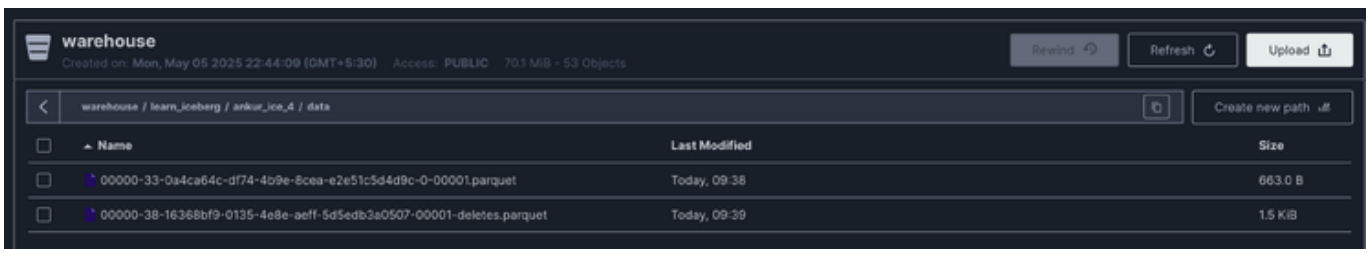
```
spark.sql("""
    CREATE TABLE demo.learn_iceberg.ankur_ice_3 (
       id   INT,
       data STRING
    ) USING iceberg
    TBLPROPERTIES (
       'write.format.default' = 'parquet',
       'write.delete.mode' = 'merge-on-read',
       'write.update.mode' = 'merge-on-read',
       'write.merge.mode' = 'merge-on-read',
       'format-version' = '2'
    )
""")
```

配置属性说明

- write.delete.mode：删除事务的处理方式
- write.update.mode：更新事务的处理方式
- write.merge.mode：合并事务的处理方式

需注意：这些属性仅为规范定义，其实际效果取决于所用查询引擎是否支持。若引擎未实现相关逻辑，可能导致非预期结果。在 MoR 模式下，所有操作均由查询引擎或消费端应用负责处理。

在我的 Apache Iceberg 表数据文件夹截图中，您会观察到存在两个 Parquet 文件：原始 Parquet 文件创建于上午 9:38。上午 9:39 执行 DELETE 命令后，生成了扩展名为 ...deletes.parquet 的删除 Parquet 文件。



需注意：删除文件（Delete Files）仅受 Iceberg v2 表支持，Iceberg v1 不支持。

相信您现已充分理解写时复制（CoW）与读时合并（MoR）。接下来我们将探讨 Apache Iceberg 的核心主题：定位删除（positional delete）与等值删除（equality delete），并解析为何 Iceberg v3 规范要用删除向量（Deletion Vectors）替代定位删除。

MoR 中的删除类型：定位删除与等值删除

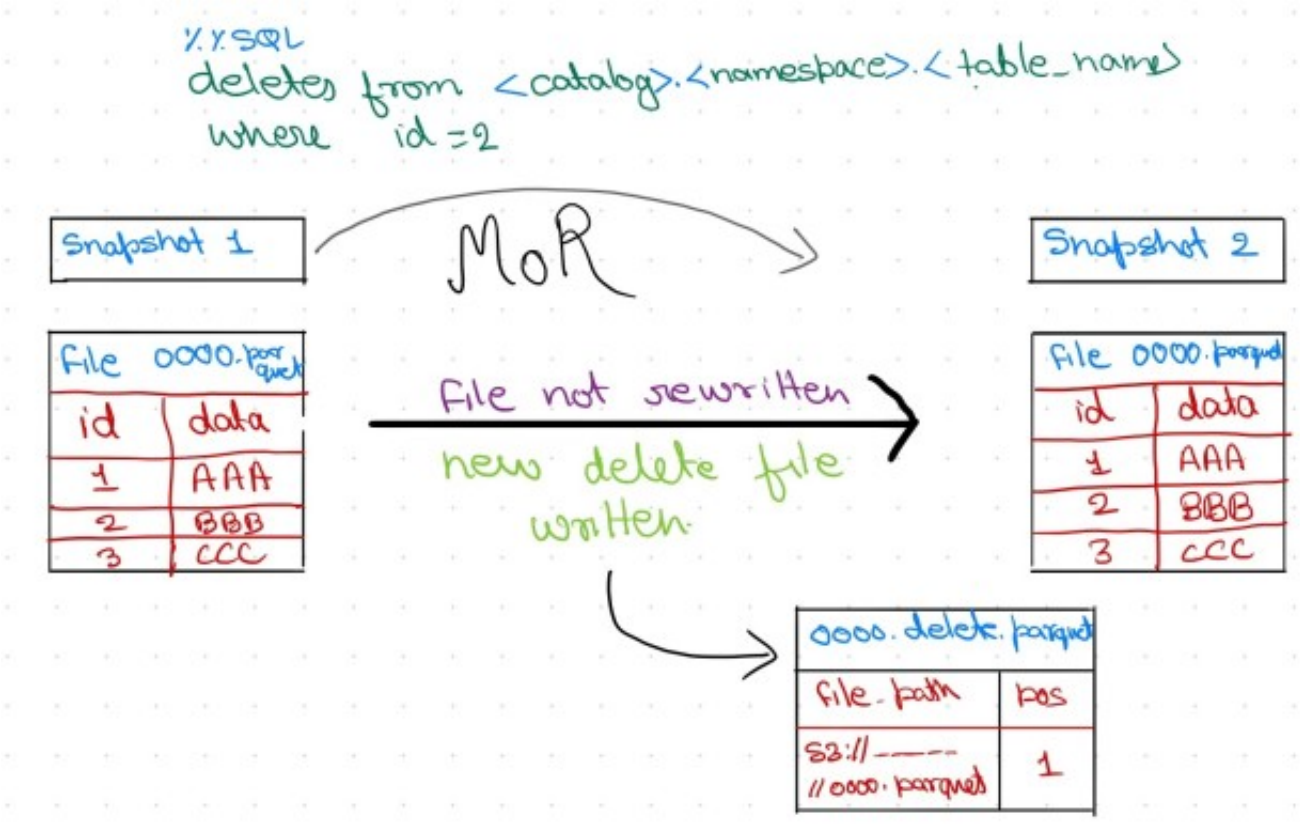
删除文件用于追踪数据集中被逻辑删除的记录，查询引擎访问 Iceberg 表时需忽略这些记录。

此类文件在分区内创建，其依据是具体发生逻辑删除或更新的数据文件。

根据记录删除信息的方式，删除文件分为两类：

定位删除文件（Positional Delete Files）

此类文件记录被删除记录在数据集中的精确位置。它们同时追踪：数据文件的文件路径（file path）以及 被删除记录在文件内的行位置（positions）



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：过往记忆大数据

定位删除文件（Positional Delete Files）列出特定数据文件中应视为已删除的行位置。

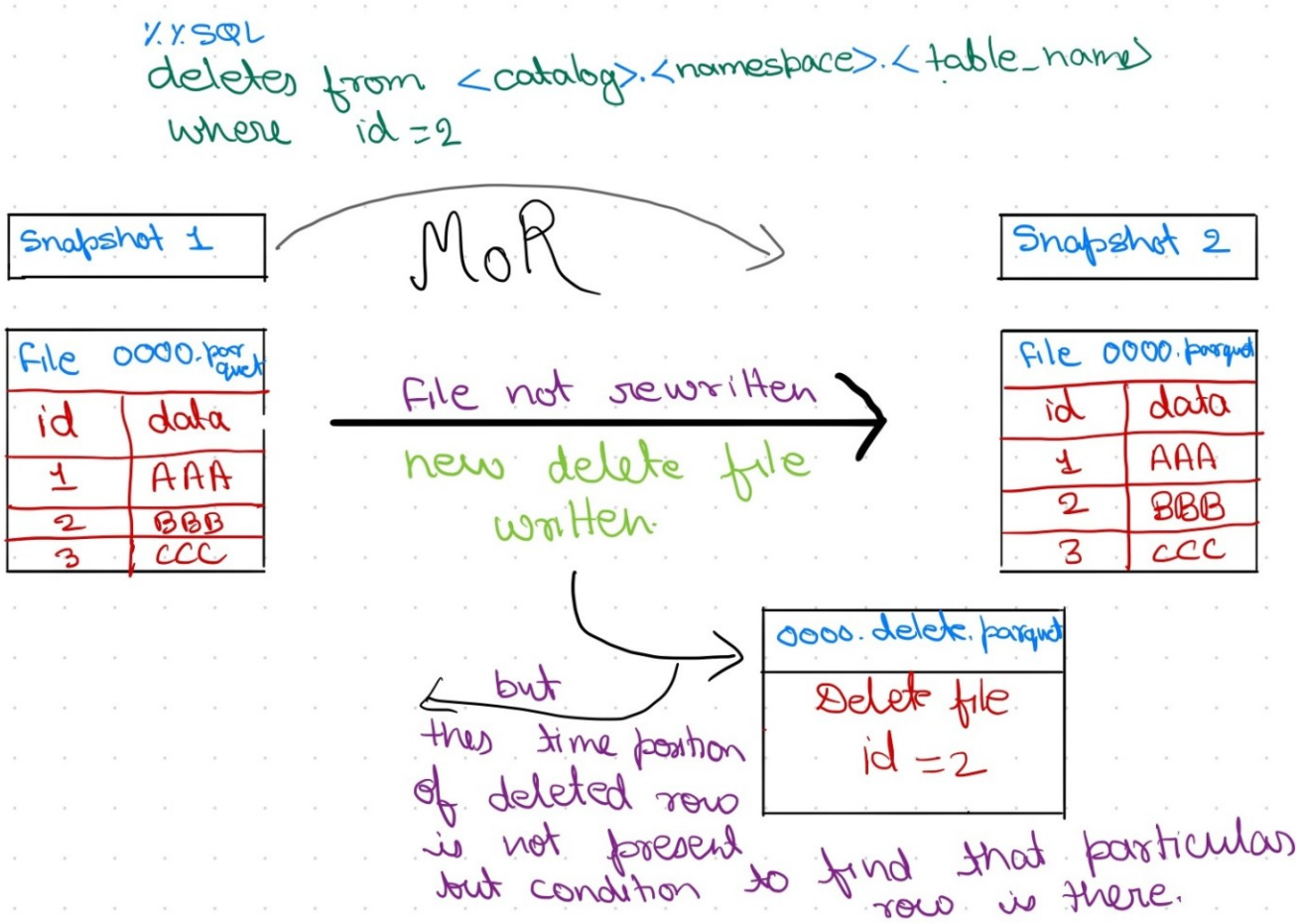
读取时，查询引擎通过元数据和列信息（如file_path和行位置pos），将删除文件与数据文件动态合并，实时屏蔽已删除行。

虽然此方法避免了每次删除操作的高成本重写，但在大规模场景下存在设计与性能缺陷，后文将详细探讨。

在深入分析定位删除的限制前，我们先简要了解等值删除（Equality Deletes）。

等值删除文件（Equality Delete Files）

等值删除文件存储被删除记录的一个或多个列的值。这些列值基于执行删除操作时使用的条件存储。



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：过往记忆大数据

当我初探删除文件时，最困惑的问题是："在PySpark中处理行级更新时，如何选择或配置表使用定位删除或等值删除文件？"

实际上，Apache Spark当前写入Iceberg时不支持等值删除。据我调研，Apache

Flink支持以等值删除形式写入删除文件。

现在您已了解两类删除文件，我们将继续解析定位删除被弃用的原因。

定位删除的运作机制（及其复杂性根源）

Iceberg中的定位删除文件通过数据文件内的行位置标记待删除行。此类文件的每条记录包含数据文件路径（data file path）、文件内的行序号（row ordinal）以及（可选）被删除行数据本身。

示例： 假设存在包含100行的数据文件data-file-1.parquet。若从MoR表中删除其中两行，Iceberg将生成小型删除文件，内容示例如下：

file_path	pos
.../data-file-1.parquet	0
.../data-file-1.parquet	102

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：过往记忆大数据

定位删除文件每行含义为“数据文件 data-file-1.parquet 中位置 X 的行已被删除”。

查询时，引擎将删除信息与实际 Parquet 文件动态合并：读取 data-file-1.parquet 时会跳过位置 0 和 102 的行，确保这些行不出现在查询结果中。

此合并过程需构建内存位图（in-memory bitmap），在读写环节将 Parquet 文件序列化为位图的成本极高。查询引擎虽通过动态合并避免每次删除的重写开销，但大规模场景中暴露显著的设计与性能缺陷。

首先需明确：Apache Iceberg 格式规范版本 2 初始采用分区范围删除文件（Partition-Scoped Delete Files），后转向文件范围删除文件（File-Scoped Delete Files）。

下文将解析两种策略的优劣并通过示例说明：

1. 分区级与文件级删除的权衡

分区范围删除文件：

同一分区的所有删除可归并至单一文件，减少磁盘文件数量，但单个文件会覆盖多个数据文件，导致读取目标数据文件时需扫描整个分区的删除条目。示例：分区 P1 含数据文件 A 和 B 时，读取文件 A 需加载包含 A/B 删除记录的合并文件，再丢弃 B 的无关条目，造成 I/O 和处理资源浪费。在本示例中，若需读取分区 P1 内的文件 A，查询仍须加载合并删除文件（该文件包含 A 和 B 的删除条目），随后再丢弃与 B 相关的无关条目。这将导致 I/O资源与处理能力的浪费——查询被迫获取实际不需要的数据文件删除信息，而这些操作仅是出于存储效率考量对删除条目进行的整合。

文件范围删除文件：

数据文件被删除行时，生成专属定位删除文件。读取目标明确：访问文件 A 时仅加载其删除文件（忽略 B）。代价是产生海量小文件，示例：删除 A 和 B 中各一行会生成两个独立小文件。大规模场景下，文件级删除将导致文件和元数据条目激增，需依赖主动的“删除文件压缩（delete file compaction）”控制文件数量。

当前 Iceberg 格式规范版本 2 支持两种策略，但均不理想：分区级删除减少文件数量，但增加读取开销；文件级删除降低读取开销，但引发文件数量爆炸。 用户常陷入两难：要么承受粗粒度删除文件的读取低效，要么应对海量细粒度删除文件导致的元数据膨胀和运维负担。设计上甚至允许同一数据文件关联多个删除文件（如多次独立删除操作后），加剧文件数量问题。

2. 查询执行期间的读 I/O 开销

使用定位删除的 MoR 表读取数据时，额外 I/O 和计算量随删除文件数量线性增长：

A. 数据文件的附加文件读取

查询任务需为每个数据文件打开并读取所有关联的定位删除文件。加载删除文件内容（通常以 Parquet/Avro/ORC 列表存储位置信息），构建内存中被删除行的位图。此后才能扫描过滤后的数据文件。Iceberg 文档明确指出，读取器必须 "开启所有匹配的删除文件并将其合并" 成删除位集（deletion bitset），然后才能扫描数据。这意味着，若某个数据分片（data split）涉及 5 个适用的独立删除文件（在多次删除且未压缩的场景中较为常见），读取器需执行 5 次额外文件读取操作，并在内存中合并五个列表/位图。
此类开销随每个任务处理的删除文件数量呈线性增长。

B. 无关数据读取

如前所述，若采用分区范围删除（partition-scoped deletes），单个删除文件可能覆盖多个数据文件。此时即使仅需扫描特定数据文件，读取器仍须加载该删除文件（其体积可能较大），即使仅极小部分条目与目标文件相关。无关条目最终将被丢弃，但获取与解析这些条目的工作已完成。这属于纯粹的开销——本质上读取了立即被抛弃的数据。例如：若分区删除文件为 50 MB，而扫描任务的目标数据文件仅关联其中 5 MB 内容，则 45 MB 的 I/O 与解析操作即被浪费。此低效性是分区级优化（减少磁盘文件数量）的代价——以加重读取负担换取存储效率。

C. 海量小文件开启

相反地，采用文件范围删除（file-scoped
deletes）虽然避免了读取无关删除条目，但需承担开启大量小文件的成本。在分布式存储（如HDFS、S3 等）上，每次文件开启操作都存在不可忽略的延迟和开销。实际上，内部基准测试表明：对于小文件而言，压缩删除数据与冗余删除数据之间的体积差异通常"低于单独开启文件的成本"。更直白的解释是：读取 N 个小删除文件可能比读取一个等体积的合并文件更慢，因为设置/寻址时间占主导地位。因此，大量小删除文件（常见于细粒度更新场景）会因每个文件的连接和读取设置成本，显著增加查询开销。

D. 运行时过滤成本

待删除信息加载后，将通过内存位图（Roaring
Bitmap）过滤行记录。此步骤本身效率较高——Iceberg
的向量化读取器能以极低行均开销校验位图。
主要性能损耗发生于前置环节：开启、读取并实例化所有位图的过程。当删除量庞大时，准备删除位图的时间消耗可能非常显著。例如，若表中大量行已被删除（但未压缩），删除文件的总体积可能接近数据文件本身大小。极端场景下，读取50%行被删除的数据会使扫描负载翻倍（需读取100%数据文件 +
近50%体积的删除条目）。尽管每个单元处理高效，总I/O量仍远高于无删除的文件扫描。

影响总结： 每次定位删除（positional delete）都会引入额外的读取开销：需开启一个附加文件并处理位置信息列表。当删除量较小时此开销可忽略，但在大规模场景下（数十万或数百万条删除分散于多个文件中），其将成为关键制约因素。该方法实质上是将删除合并工作转嫁至查询时处理，导致高频更新表的查询I/O吞吐量下降与延迟增加。

3. 删除文件累积与维护负担

由于定位删除（position
deletes）始终独立于数据存储，每次更新/删除操作均会使其累积。Iceberg
规范未要求写入时自动合并新旧删除文件。
这意味着若频繁执行删除或更新命令，表中将随时间推移堆积大量删除文件。

示例：若某分区每日删除少量行且不清理，30天后该分区将积累30个独立删除文件（全表范围内数量更甚）。Iceberg 依赖用户或外部进程定期压缩删除文件，即执行维护操作：
 轻度压缩（Minor compaction）：将多个小删除文件合并为单个大文件

 深度压缩（Major
compaction）：通过重写数据将删除操作合并至数据文件（将相关部分从MoR转为写时复制）

若缺乏维护，性能将快速劣化—— 每个新增删除文件都会增加未来读写开销。实际案例表明：未经压缩的频繁更新表会因引擎处理不断增长的删除文件列表，导致查询显著变慢。

用户必须手动调用操作重写定位删除文件，若维护不足，读写性能劣化速度将超预期。换言之，稳定性能取决于持续监控（运行定期清理任务），这增加了操作复杂性。此设计存在缺陷：高效存储格式本应内部管理元数据增长。 依赖外部维护存在人为错误或延迟风险：一旦压缩任务中断，表将残留海量小删除文件，显著拖慢读取。

生产环境案例：

某按日期分区的PB级Iceberg表每日跨分区删除。由于每个分区的删除文件独立存储，数周后磁盘累积数千万个小删除文件。 即便采用分区范围删除优化单个分区的文件数量，巨量分区（每个至少一个删除文件）仍导致文件总数爆炸式增长。 此状况不仅加重元数据与存储系统负担（海量小对象导致文件系统开销），还使查询计划与执行日益迟缓。此类场景亟需主动压缩来合并/移除删除文件——本质上需与存储格式的固有特性对抗以维持性能。

4. 清单与元数据膨胀

每个定位删除文件（positional delete file）均被记录在 Iceberg 的清单文件（manifest files）元数据中。随着删除文件数量增加，清单会膨胀（即 "manifest bloat"），且在每次快照的元数据中追踪这些文件将加重负担。

比如 1000 个小删除文件可能向清单中添加 1000 个条目。这不仅会增大磁盘上的元数据文件体积，还意味着刷新表或规划查询时，Iceberg 规划器（catalog）需处理更多条目。

Trino 和 Spark 等工具需加载快照元数据，因此膨胀的清单会增加查询计划时间，甚至导致驱动节点/协调器（driver/coordinator）的内存占用上升。需注意：过多元数据文件（含删除文件）将加剧内存压力——因文件列表通常为性能优化而缓存及索引在内存中。

5. 悬空删除与重写问题

定位删除（Positional deletes）使数据文件重写复杂化。若数据文件被重写（例如因压缩或优化文件大小），理想情况下原文件关联的删除条目应同步失效。Iceberg 虽通过快照隔离（snapshot isolation）机制确保数据一致性（在重写提交前不会丢弃删除文件，直至旧快照过期），但实践中仍存在"悬空删除"（dangling deletes）问题——元数据中残留引用已不存在文件的删除条目。这需额外清理步骤（Iceberg 提供 rewrite_position_delete_files 过程处理：移除指向无效数据文件的删除条目并合并其余条目），显著增加操作复杂性。

总结而言，定位删除（positional deletes）虽能实现免整文件重写的行级删除，但引入了显著开销：额外文件、冗余元数据及运行时合并工作。这些成本随每次删除操作累积，使大规模增量删除的管理日益昂贵。通过上述案例与讨论，相信您已理解 Apache Iceberg MoR 表中使用定位删除的复杂性及固有缺陷。

接下来，我们将聚焦 Iceberg 格式规范版本 3（format-spec 3）引入的删除向量（Deletion Vector, DV）——该机制将逐步取代现有定位删除方案。

什么是删除向量（Iceberg v3的解决方案）？

删除向量（Deletion Vectors, DV）是 Iceberg 格式规范版本 3 中取代定位删除文件（positional delete files）的新机制。删除向量本质上是给定数据文件内被删除行位置的位图（bitmap）。不同于在独立文件中存储位置列表，DV 将相同信息记录为二进制位集（binary bitset）：若某位为

"1"（或位图中该位存在），则对应行位置视为已删除。

删除向量的工作机制

每个存在删除记录的数据文件最多关联一个删除向量（deletion vector）。该向量映射该文件内应视为删除的行位置。例如：若 data-file-1.parquet 需删除行 0 和 102（如前述示例），其删除向量会将位（bits）0 和 102 置位（set），其余位（位置）保持未设置（unset）状态。

删除向量以二进制块（binary blobs）形式存储（采用基于 Roaring Bitmap 的压缩位图表示）。这些二进制块不作为独立文件存储于表数据路径，而是整合至特殊结构化文件（使用 Puffin 格式——Iceberg 的辅助二进制数据容器）。多个删除向量位图可打包至单个 Puffin 文件以提升效率。

表元数据（manifests）通过以下方式引用 Puffin 文件：
记录 Puffin 文件路径及每个位图的偏移量（offset）和长度（length）。换言之，清单（manifest）不再罗列多个小型删除文件，而是记录如：
“data-file-1.parquet 的删除向量位于 delete_vectors.puffin 偏移 X 处（长度 Y 字节）”。

由于每个数据文件仅允许关联一个删除向量，因此当该文件新增删除操作时，必须更新或替换现有向量（而非追加独立向量）。

实际操作中，新提交的删除操作会执行位图合并（UNION操作：将新删除位置与现有点阵合并），并写入新的组合删除向量。这确保始终仅存在单个连续位图表示该文件的所有删除记录。旧向量在新快照中将被废弃或取代。此设计彻底避免单文件多层删除信息堆叠——始终保持整合状态。

Iceberg
规范明确定义删除向量二进制块及其元数据格式。删除文件的清单条目现包含：file_path（指向 Puffin 文件）content_offset（偏移量）与 content_size（长度）——定位位图在文件内的位置。同时存储该向量归属的数据文件标识。本质上，清单直接将数据文件映射到其被删除行的特定位图。

删除向量的引入是经过深思熟虑的优化举措，旨在精简行级删除（row-level deletes）流程。

Iceberg 联合创始人 Ryan Blue 与 Anton Okolnychyi 阐释道：Iceberg 社区在设计 v3 版本时，与 Delta Lake 团队（已实现类似方案）协作完善此机制。我尤其欣赏 Anton 在今年 Apache Iceberg Summit 上对这些概念的透彻解析。

其核心目标是以更低开销实现免重写的行级删除。实际上，删除向量达成与定位删除（positional deletes）相同的逻辑结果——在读取时隐藏已删除行——但以更紧凑高效的方式存储信息。

本文翻译自：https://www.linkedin.com/pulse/why-iceberg-choosing-deperecate-positional-delete-mor-ankur-ranjan-fpjjc