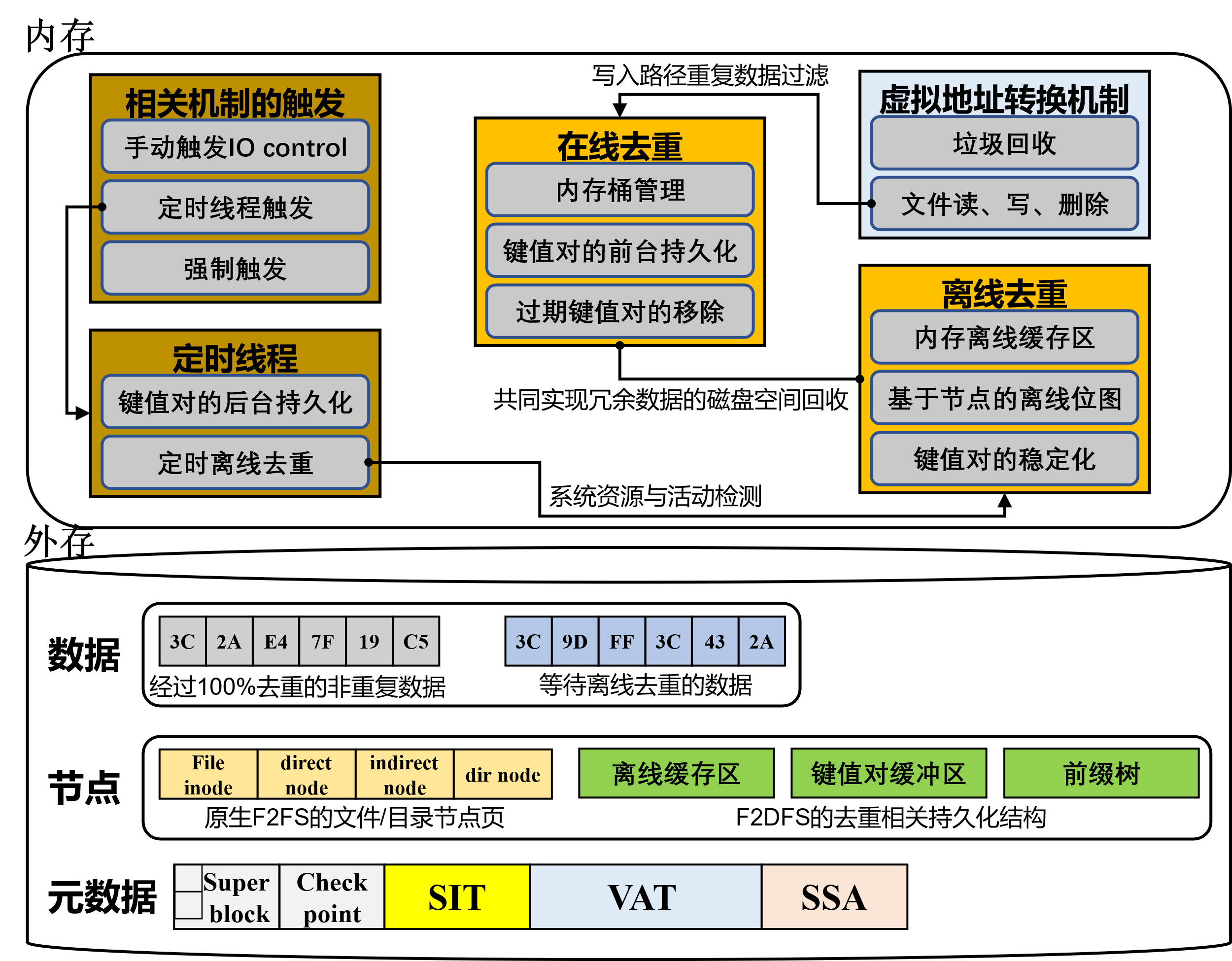
1. **F2DFS总览**

**1.1内存与外存模块**



**图1 - F2DFS系统总览图**

F2DFS分为内存管理和外存管理两部分，具体结构如图1所示。

在原有F2FS外存结构基础上，F2DFS对元数据，数据与节点块结构与管理有以下修改：

1. 数据区域设立虚拟地址表（Virtual Address Table, VAT），通过部分缓存在内存的VAT，在支持高效块级重复数据删除（去重）技术的文件系统上实现垃圾回收（garbage collection, GC），其详细设计结构见“元数据管理”。
2. 持久化的数据块分为非重复数据与等待离线去重的数据，F2DFS会在写入路径记录下后者的索引信息，并在系统空闲时刻对它们进行100%离线去重，最终在外存上保留唯一的数据副本。
3. 为了节省内存管理开销，F2DFS会将内存的一些数据结构进行持久化，比如与键值对相关的键值对缓冲区（key-value buffer, KV buffer）与前缀树（prefix tree），以及离线缓存区（offline buffer），同时，我们修改了原生F2FS的节点页结构，使其支持离线去重（见“离线去重”）。

F2DFS在内存相关操作包含以下三个部分：

1. 虚拟地址的转换，这在系统的正常读写和垃圾回收流程承担重要作用，有效解决了重复数据删除带来的逻辑地址与物理地址“多对一”映射的管理困难，并与数据块的有效状态、键值对高效管理高度相关。
2. 在线与离线相结合的100%重复数据删除，包含内存桶与离线缓存区、基于节点页的离线位图（offline bitmap），以及键值对的持久化与稳定化，二者共同实现外存冗余数据的回收，此外在线去重可以实时缩减数据写入量，减少对闪存设备的磨损，对写入路径的吞吐量有影响。
3. F2DFS设置了定时线程，选择系统负载较低或用户空闲时刻自动进行键值对持久化与离线去重，我们也为终端用户提供可以直接调用相关操作的API（如ioctl）。

文章剩下内容会对以上内存与外存模块原理进行介绍，并提供详尽的系统性能测评。

**1.2元数据管理**

**1.2.1超级块（Superblock）与检查点（Checkpoint）**

F2DFS在原有F2FS的超级块基础上添加了gc\_inode，用于垃圾回收阶段有效数据的迁移，并添加与去重相关的数据结构的内存指针；同时，检查点分离了文件系统逻辑数据量和物理数据量之间的强相关联系，可以为上层应用提供当前系统去重率、剩余可用物理空间、离线去重阶段需要处理的数据量等交互信息。

**1.2.2虚拟地址表**

虚拟地址表是本文重要元数据结构，它提供了由数据块虚拟地址（virtual address, VA）到物理地址（physical address, PA）的翻译功能，其作用和结构类似于原生F2FS的节点地址表（Node Address Table, NAT）。针对节点块，VAT继承了F2FS原有功能，包括记录节点页所在文件的inode id，该节点的物理地址和更新版本号等信息；针对数据块，VAT记录了数据块的物理地址，数据块哈希指纹（fingerprint, FP）的前缀（见“高效键值对管理”），以及该数据块被逻辑引用计数（reference count, RC）。

图示

描述已自动生成

**图2 - 虚拟地址转换机制**

如图2，F2DFS将文件映射关系由逻辑地址（logical address, LA）到物理地址修改为到虚拟地址，并由VAT执行到物理地址的翻译，进而访问外存物理数据块。在F2DFS中，已分配的虚拟地址总数与Main Area的实际有效块数维持严格的一对一关系，由于VAT以原地写方式更新到文件系统元数据区域，系统可以通过文件映射获得的VA直接计算出对应元数据的第几个块而快速进行地址转译。虚拟地址相关操作只在F2DFS实现，对于应用与进程、Page Cache与块层IO等完全透明，实现内存管理与外存介质的隔离，减少了代码移植难度。

对于写负载较大场景（如direct I/O）或者涉及到数据安全（如fsync相关的崩溃一致性），F2DFS不会进行在线去重，而是转交离线去重处理。并且为了确保经历突发的系统崩溃或掉电后能够正常恢复数据，F2DFS不会对它们分配虚拟地址，而是和原生F2FS一样，保留逻辑地址到物理地址的直接映射，相关读写、垃圾回收与数据恢复函数不做修改。

F2DFS会加载部分VAT表项到内存以减少内存使用，使用LRU队列进行管理并及时回收被释放的表项。我们的实验结果表明，即便是细粒度的随机读写，VAT依旧可以表现优秀的转译效率（见实验）。

**1.2.3反向索引表（Segment Summary Area, SSA）**

反向索引表又叫段摘要区域，是F2FS垃圾回收过程最重要的数据结构，它通过记录每个数据块所在父节点与节点内偏移，计算出该块位于哪个文件的第几个逻辑块（相当于SSA记录了物理地址到逻辑地址的反向索引），然后以文件为单位将有效块合并迁移到其他segment，实现旧segment空间的回收。在原生F2FS中，垃圾回收流程涵盖了物理数据块（PA）、SSA和文件映射（即数据块所在父节点，LA）三部分的更新，三者维持“1 : 1 : 1”的关系，这也使得去重引入的“LA : PA = N : 1”与GC存在设计冲突。如果单纯修改SSA结构，使其记录相同数据块的多个反向逻辑索引，会导致三种潜在问题，一是不同数据块引用次数不同但元数据空间大小只能静态分配，RC较小的空间浪费，RC较大的又无法存储全部反向索引；二是每处理一个重复块，需要修改所有引用的父节点，这会增加了更多随机读写，极大影响了GC效率；三是F2FS会在剩余外存空间较少情况下，会触发前台GC以回收更多可用空间，但是每迁移一个重复引用的数据块，就需要异地更新所有相关的节点页，这加剧了外存紧张。

F2DFS使用虚拟地址技术，通过“LA : VA : PA = N : 1: 1”的映射关系解决了以上问题，对于每个物理数据块，SSA只记录其绑定的虚拟地址，并且在垃圾回收阶段只更新VAT中记录的物理地址信息，而所有引用该数据块的文件映射依旧保留着该数据块的虚拟地址不做修改，避免了重复数据的GC阶段对其多个父节点的读取与更新。

**1.2.4数据一致性**

在F2DFS中，部分内存结构（内存的VAT，桶的有效数据，桶的待删除数据，离线缓存区，离线位图）需要持久化到外存以保证在突发断电或系统崩溃后可以快速实现数据恢复，大致分为两类：前滚和后滚。

前滚数据恢复依赖原生F2FS的检查点机制，它保证了元数据区域的有效，尤其是标识Main Area所有数据节点有效状况的段位图信息表（Segment Info Table, SIT），以及新增的系统重复率等交互信息。我们会在每次100%离线去重前后创建检查点，并终止该过程中其他应用或线程创建检查点，以确保去重过程崩溃恢复后能够还原到最初状态。

原生F2FS的后滚数据恢复用于恢复上次检查点建立后到系统意外崩溃之间的fsync相关数据，包括日志（journal）中的NAT表项，以及通过fsync触发的数据所在节点页。在F2DFS的内存结构中，离线位图是保留在节点页里的，位图也可以快速还原离线缓存区未同步的表项，同时借助下次离线去重将因崩溃未持久化的桶有效数据（键值对）更新到前缀树，所以这三者不需要额外的机制保障。所以，F2DFS只需要额外记录fsync触发的内存VAT（这包含了桶的待删除数据），对于这部分信息可以复用F2FS的日志与相关函数实现。

**二、基于虚拟地址的文件操作流程**

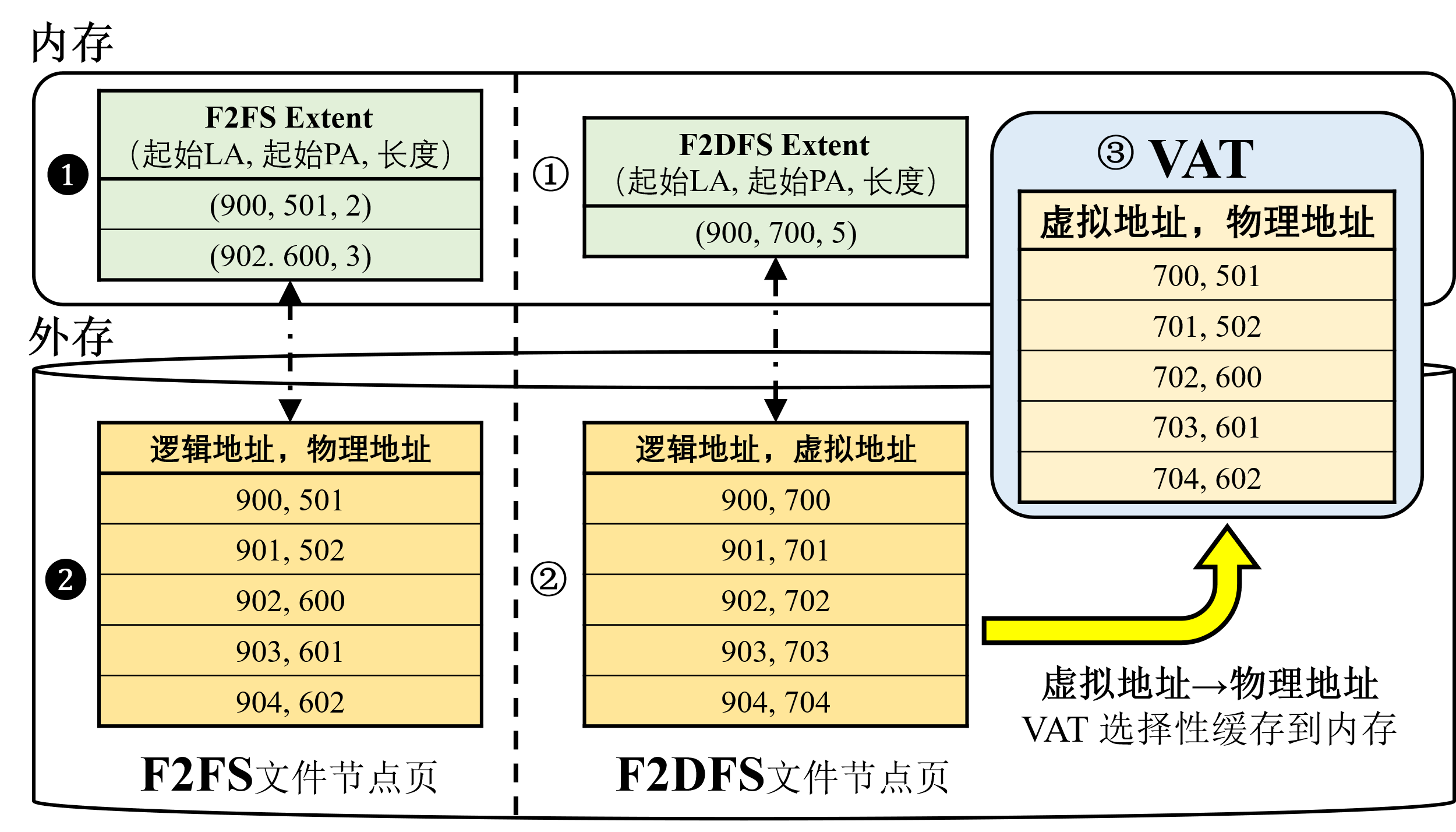
在原生F2FS中，普通文件使用“文件inode->(in)direct node->direct node->data blocks”的多级管理实现基于指针的文件索引，其中文件inode与direct node（合称为dnode）保留了指向外存上固定4KB数据块的物理指针。通过要访问数据在文件内的逻辑偏移量，F2FS会计算出目标位于文件第几个节点页，从而将地址转换控制在从inode到该物理指针路径内的少数节点内，减少了无效节点页的读取。

F2DFS将dnode中存储的物理指针修改为虚拟指针，并且由F2FS原有文件索引方式维护逻辑地址到虚拟指针的读取，并通过VAT再实现虚拟地址到物理地址翻译，最终得到要处理的数据块。

**2.1常见文件操作**

**2.1.1文件读**

在F2FS中，文件系统会通过内存为每个近期访问的文件创建一个extent，它包括（起始逻辑地址，起始物理地址，extent长度）三部分，通过将将近期访问的节点页中连续LA到相同长度的连续PA抽象为这三个参数，大大压缩了内存使用，显著加速了文件顺序读写性能。



**图3 - 读流程**

具体读流程如图3所示，其中F2FS实现逻辑地址到物理地址转移需要以下步骤：

❶ 在文件内存保留的extent中搜索，如果找到，计算出物理地址并直接结束；

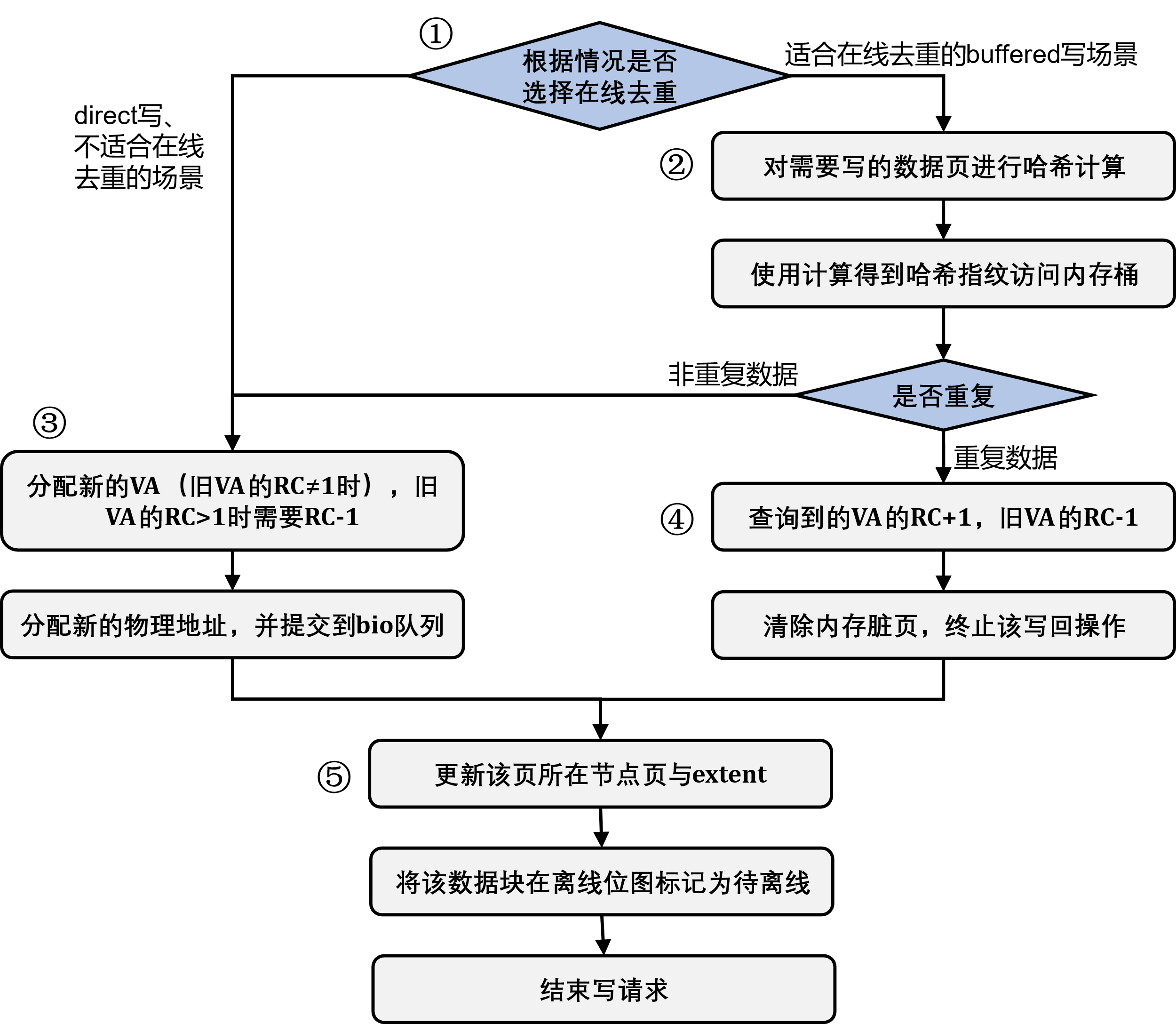
❷ 通过数据逻辑偏移计算出其所在dnode的访问路径，读取外存对应节点页得到物理地址，并更新到extent中。

在F2DFS中，除了此前dnode中的修改，内存extent也改为（起始逻辑地址，起始虚拟地址，extent长度）三部分，并将逻辑地址到物理地址转移修改为以下步骤：

* 1. 在文件内存保留的extent中搜索，如果找到，计算出虚拟地址并转到③；
  2. 通过数据逻辑偏移计算出其所在dnode的访问路径，读取外存对应节点页得到虚拟地址，并更新到extent中；
  3. 通过VAT进行进一步翻译得到物理地址，如果对应表项不在内存，则去外存读取，将VAT选择性缓存到内存。

**2.1.2文件写**

F2FS的写操作只涉及地址分配，包括旧地址的无效与新地址的申请。Linux文件系统的数据写大致分为四个阶段：“write\_begin→write\_end→set\_page\_dirty→write\_datapage(s)”（buffered IO），为了避免阻塞多线程调用write\_end函数，以及脏页redirty导致无效在线去重，F2DFS只对write\_datapage(s)阶段进行修改，包含了地址分配和在线去重两部分。



**图4 - 写流程**

为了在异步写入路径上实现在线去重，如图4所示，我们将写流程分为以下几个步骤：

* 1. 根据不同写模式与系统负载，选择性执行在线去重，如果是direct IO或其他不适合在线去重场景，则跳转③；
  2. 对Page Cache中要处理的页进行抗碰撞哈希计算，得到指纹作为键访问内存桶，如果是重复数据则跳转到④，否则继续执行；
  3. 为逻辑地址分配一个虚拟地址（如果此时旧VA的RC=1，则无须，对于旧VA的RC>1的情况，需要将旧VA的RC-1），然后为这个虚拟地址分配一个新的物理地址，并将其与内存页提交到bio请求，块层仍旧使用物理地址操作，直接跳转到⑤；
  4. 对于在线去重查询到的重复数据，我们从内存桶中获得相同内容数据所绑定的虚拟地址，将该VA的RC+1（同时将节点拥有的旧VA的RC-1），清除该内存页的脏标记并终止该写回操作；
  5. 更新虚拟地址到该文件的逻辑偏移，包括需要持久化的节点页与该文件的内存extent，在F2DFS中，非特殊情况所有新写入数据都需要将节点页中离线位图对应位置标记为“待离线”（见“离线去重”），最后结束写请求。

在上述的①中，我们目前只针对写模式进行判定，即对于buffered写请求（数据页在内存Page Cache可以直接计算哈希）我们启用在线去重，而对于direct写请求，由于数据直接写到块层，文件系统只在该过程预分配地址却无法捕获数据内容，我们对这部分数据只进行记录而在未来执行离线去重。值得说明的是，buffered IO见于绝大多数场景，使用主要文件类型（用户文件、多媒体数据和应用信息等）、常见应用操作（同步与异步写）和软件缓存等，而direct IO使用场景极少，大多见于在外存自带缓存区的数据库或用于硬件压测，所以F2DFS依旧可以对绝大多数文件系统写请求执行离线去重。

F2DFS并不旨在写入路径上完全实现100%冗余数据消除，因为内存桶能缓存的键值对数量是有限的，我们只是尽可能减少相同应用或文件的重复写入，避免写放大、写悬崖，减少对基于闪存的存储设备的写入总量与擦除次数，延长硬件耐用性，这可以极大缓解现代应用工作负载密集性和设备存储的有限性（IO性能、容量、耐久性有限）之间的冲突。

在后续的研究点中，F2DFS也可以充分利用文件系统层的优势，对数据的冷热、文件的语义、数据集的重复率、系统的负载压力等进行动态调整，选择性执行在线去重，这对于CPU、内存资源有限或执行尾延迟要求严格的智能终端系统有更广阔的前景。

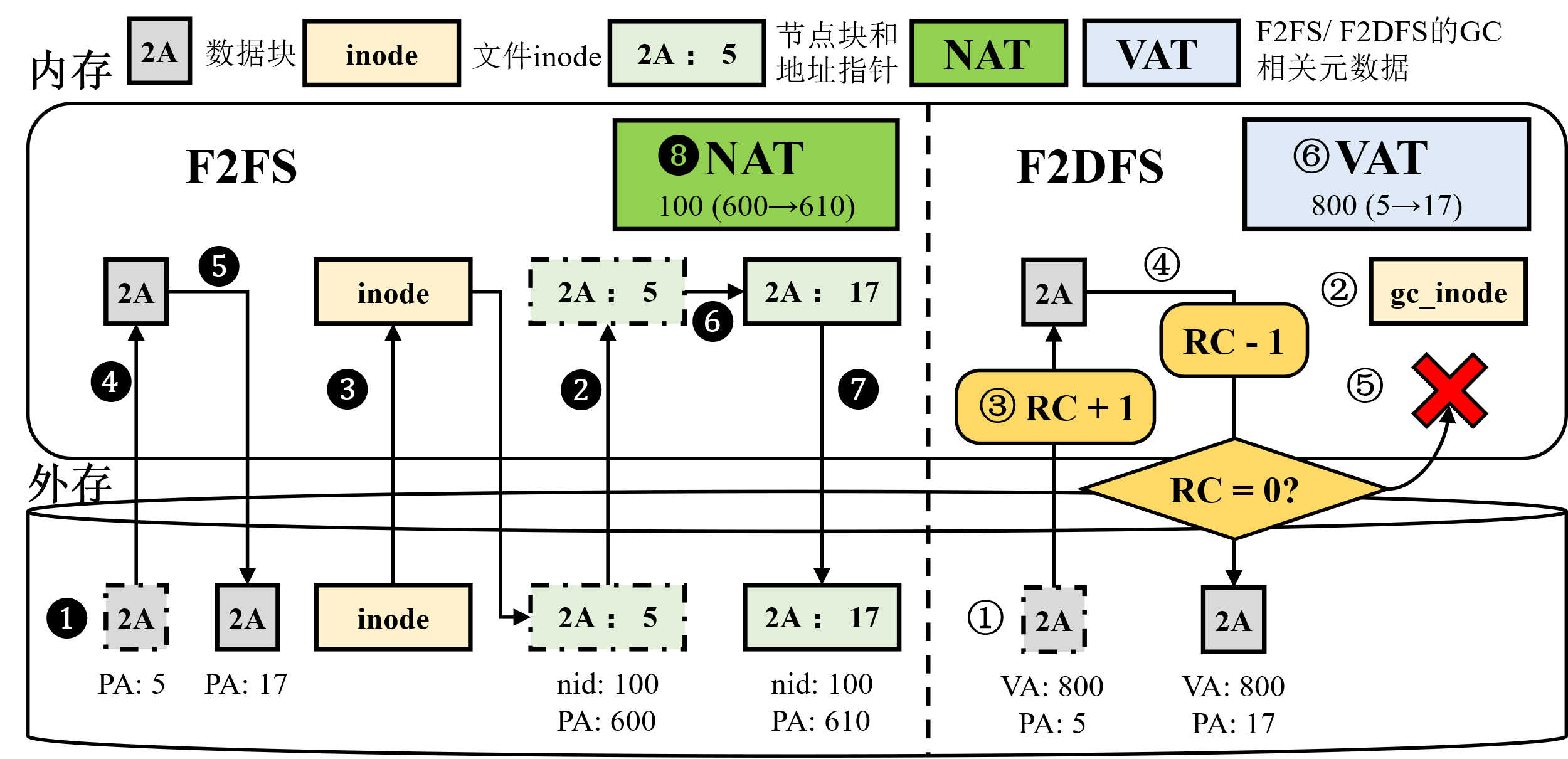
**2.1.3文件删除与数据截断**

在现有文件系统中，数据截断（truncate, Tru）是使用比较广泛的数据更新策略，文件系统会接收来自VFS层的截断与数据修改命令，将文件某逻辑偏移的内容更新，并从该位置以后将文件全部重写。在F2FS中，Tru会将修改位置以后所有属于该文件的数据块无效掉，然后对它们进行移位操作（因为文件更新操作单位是字节，而不是数据块的存储单位4KB）并重写提交写请求，通过异地写方式更新到外存其他位置。相同的还有文件删除，F2FS会将删除文件从起始地址开始进行数据截断，并在最后释放所有相关文件节点页。

由于F2DFS将逻辑地址与物理地址解耦，逻辑数据的删除并不会直接释放物理数据，而是需要对对应的虚拟地址的RC-1，当RC归零，才会释放占用的物理数据块与无效物理地址，此外还需要移除过期的键值对（见“键值对的移除”），最后回收释放的虚拟地址。为了提升非重复数据的异地写操作（即先截断某偏移，再异地更新其后全部内容），我们对RC=1的虚拟地址不进行回收再分配，而是只修改其指向的物理块，并对新旧键值进行管理。

**2.2垃圾回收**

垃圾回收是日志结构文件系统（Log-structured File System, LFS）的特性，由于其选择异地更新方式提升随机写和SQLite更新性能，LFS需要定期执行GC来回收文件碎片，通过将不同segments的文件碎片合并并迁移到新的segment，文件系统可以回收更多旧segments。F2FS就是这样的文件系统，它会在系统空闲时定期执行后台GC回收热度较低但回收性价比较高的segments，同时也会在外存资源占用紧张情况下触发前台GC以回收有效率较低的segments。



**图5 垃圾回收流程**

如图5所示，GC流程如下：

❶ 根据当前GC策略（前台或后台）选择需要迁移的数据块；

❷ 通过系统的反向索引SSA，读出该数据块的父节点dnode；

❸ 利用父节点保留的元数据信息，计算出该数据块位于哪个文件的第几个偏移（即逻辑地址），并将文件inode读出；

❹ 通过LA读出该数据块，并将其加入该文件所在的内存Page Cache中，并标记为脏；

❺ 等待写回，对于后台GC会直到系统发出同步指令或积累一定数量脏页再操作，对应前台GC则会直接写回以尽快回收旧物理地址；

❻ 将迁移后的新PA更新到文件节点页对应偏移；

❼ 等待节点页的写回；

❽ 在NAT中更新节点页的新地址。

通过上述操作，我们可以得出以下结论：F2FS的垃圾回收本质是文件逻辑数据块的异地写更新，只是该过程只将数据页标记为脏而不修改数据内容。需要注意的是，该过程并非有效数据块的简单迁移与父节点的更新，而是由所属文件进行操作，以避免前台访问的数据同时被选中GC导致的一致性问题，同时也有利于后台GC合并多个属于同一文件的数据。所以，原生F2FS的垃圾回收至少需要进行三次随机读（父节点，文件inode与数据块）以及两次写（数据块，更新后的父节点）。

但是正如前文所说，引入了重复数据删除的F2FS，继续执行上述GC流程只会复杂化，并且对前台GC不友好，所以F2DFS使用虚拟地址技术优化了其流程：

1. 根据当前GC策略（前台或后台）选择需要迁移的数据块；
2. 使用F2DFS超级块自带的gc\_inode统一索引被选中的数据块；
3. 将该数据块绑定的VA的RC+1，然后读出该数据块；
4. 等待写回，F2DFS同样支持后台与前台两种模式，对写回的数据块进行RC-1，然后判定，如果为0，说明所有逻辑引用该数据块的节点页均释放，否则跳转到⑥；
5. 将该页无效，放弃写回并直接回收相关VA与PA，直接结束GC流程；
6. 将迁移后的新物理地址更新到VAT，并回收旧物理地址。

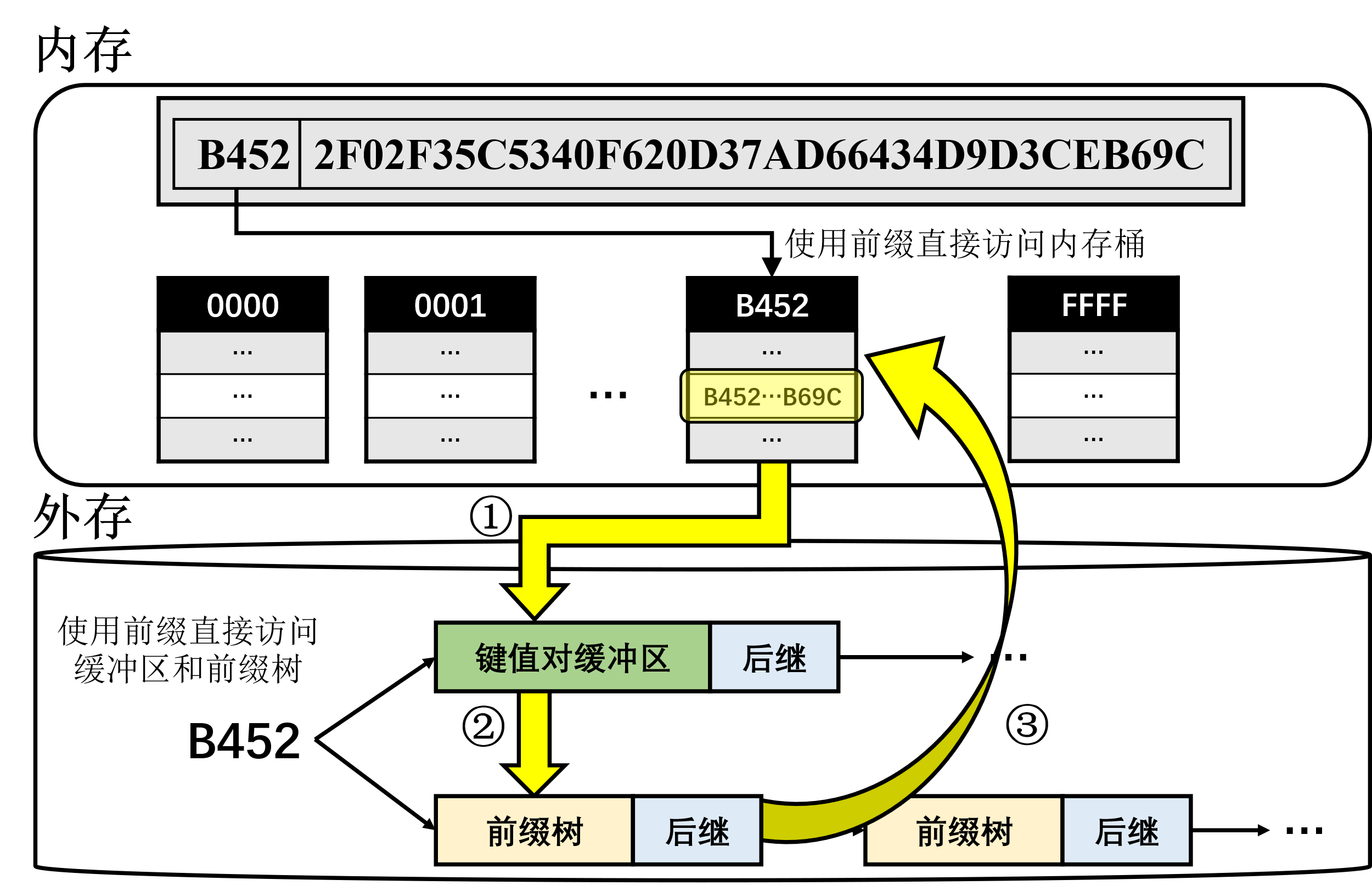
由此可以看出，F2DFS的物理块的回收只涉及数据块自身的读写，而不包含所有指向该块的文件inode与父节点，显著减少了垃圾回收随机读写次数。Gc\_inode只有在执行垃圾回收时才会创建，它通过稀疏文件形式对每个GC的数据块进行内存管理，记录它们各自绑定的虚拟地址，并提供文件级的读写操作，由于重复数据删除导致相邻数据块可能属于多个不同逻辑文件，GC阶段根据所属文件合并写请求收效不大，所以gc\_inode只负责批量处理写请求而不根据各自所属逻辑文件进行合并。此外，为了确保前台文件与GC不会产生一致性冲突，F2DFS在数据块读写前对其RC分别执行加1与减1操作，这样即便前台文件将其删除，数据页也能在GC完成迁移前释放而不会产生“孤儿数据”。

**三、高效键值对管理机制**

**3.1基于指纹的重复数据索引**

现有的重复数据删除系统中，广泛使用对数据进行密码级的抗碰撞哈希计算方式，得到唯一识别数据内容的哈希指纹。我们使用安全散列算法1（Secure Hash Algorithm, SHA-1）对固定4KB大小的数据块（也是最合适的去重单位，以及F2FS的读写GC单位）计算，它可以使用较低的CPU资源快速得到长度20B的键，压缩重复数据对比所需的内存开销与键值对管理的外存空间。由于我们对固定大小数据计算，SHA-1的理论碰撞概率为，远超普通文件系统或硬件存储的上限（32位地址能最多指向的数据）。

**3.2索引键值对的管理**



**图6 重复数据索引管理**

如图6，F2DFS的键值对（指纹FP为键长20B，虚拟地址VA为值长4B）分为内存与外存两部分，共有三个数据结构管理，分别为内存桶，外存键值对缓冲区与外存前缀树。由于键值对特有的“长键短值”特性以及指纹自带的抗碰撞特性，F2DFS在内存超级块维护了一个桶指针表，指向不同的内存桶，并且使用计算得到的指纹前缀在同一时刻访问不同的桶，避免了单一键值存储结构的存在的锁争用问题，实现在线去重阶段键值对的并行查询与修改。每个桶共享自身的读写锁，用于维护键值对热度与桶元数据管理，使用LRU算法驱逐不常访问的键值。

对于内存桶（见图8的桶结构），为每个4KB物理数据块构建25B的内存键值，包括（20B指纹，4B虚拟地址，1B标记位），其内存代价为6.25MB每GB数据。近年来针对内存有限设备专门设计的去重系统SmartDedup使用了与我们相同量级的内存代价（4.6MB每GB数据，主要因为后者使用了更短的哈希）。

在外存中，每个桶拥有一个键值对缓冲区，它与内存桶共享相同指纹所以可以很快进行持久化（①），当桶中键值对溢出，就会将逐出的表项同步到键值对缓冲区（这个过程称为“前台持久化”），F2DFS一次性逐出的表项，以避免频繁触发前台持久化。当系统长时间不进行桶访问，F2DFS会启动定时线程将桶中数据全部同步到缓冲区（这个过程称为“后台持久化”），减少桶中缓存键值数量以应对未来可能的突发负载，或者释放桶内存节省终端电量。

在系统空闲时刻，F2DFS会将键值对缓冲区中的相关信息（包括新增数据或待删除数据）同步到相同前缀的前缀树（即②，该过程称为“键值对的稳定化”），前缀树中管理了唯一键值，在系统执行100%离线去重时将相关键值提取到内存桶中（③）。

对于普通的键值对持久化系统（如B树，LSM树），并不是很适合应对“长键短值”的键值操作，很难平衡多个更新与查询请求的锁操作，严格有序的存储也无法感知键值热度等。前缀树与内存桶结合，充分发挥指纹前缀较低碰撞的优势，可以一次性执行多个相同前缀的键值对的批量查询与修改，也支持不同前缀的键值对的并行操作，同时也便于持久化管理。

假定使用2B前缀长度，那么每个前缀树只需1个树节点就可以索引到多达40GB数据，对于一个256GB的智能终端设备只约6层就可以访问所有键值，对于不同内存和外存大小的设备可以选择合适的前缀长度进行调整。对于一个256GB的智能终端，尽管每个树节点固定为4KB（能访问170个键值），使用B树需要访问4层才能得到一个键值信息，尽管前缀树可能需要访问1~6层不等，但后者可以批量查询更新同一前缀的多个键值，并且不存在不同前缀键值之间的树锁争用，其读放大代价与修改效率显著优于传统树结构。在本文代码开发与实验测试中，前缀统一定为14bits，每个桶的大小约为4KB，可以提高前台持久化执行效率，并在内存中缓存大约8GB热数据的键值对。

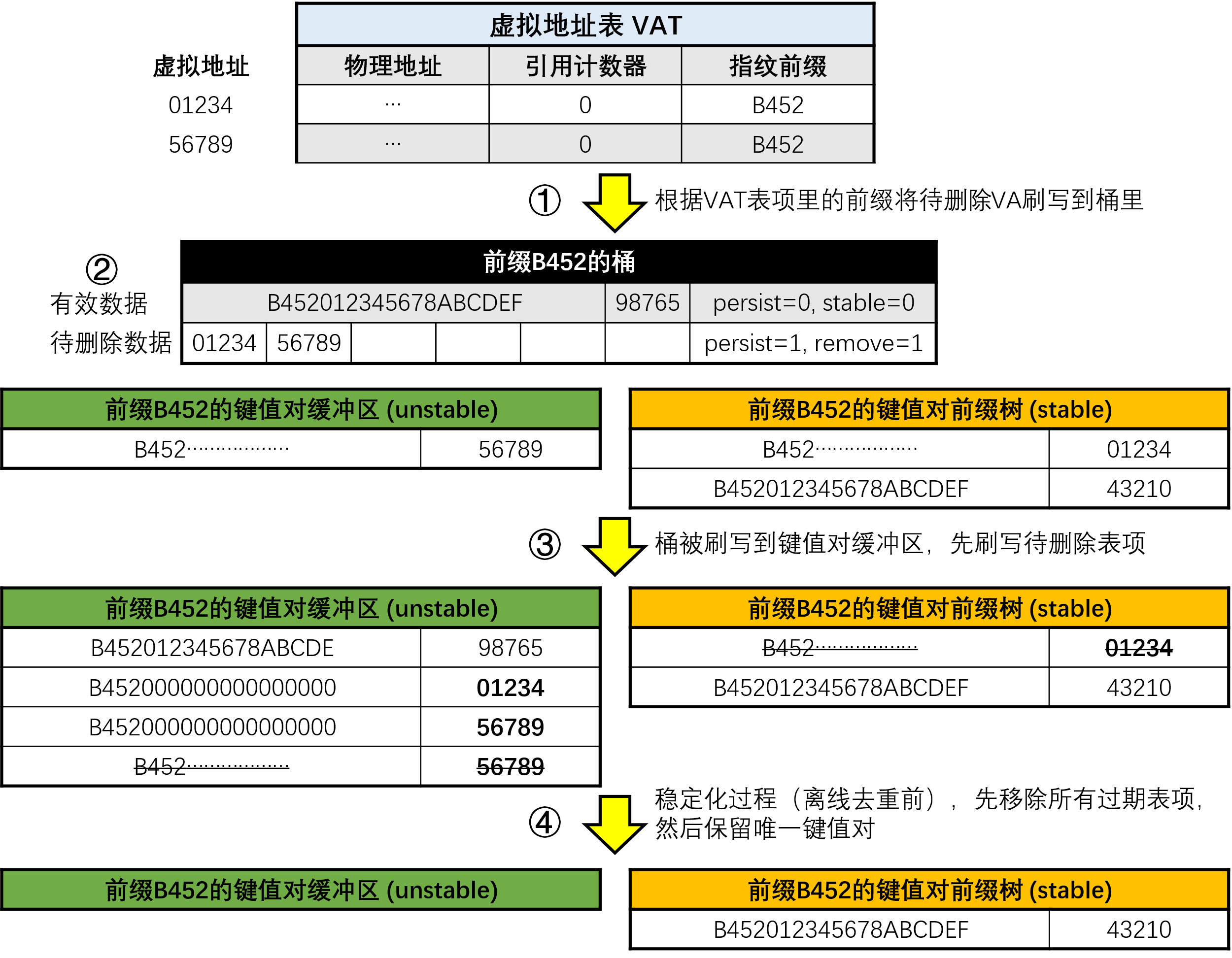
此外，键值对缓冲区和前缀树在Main Area以节点页的方式存储与索引，通过模拟稀疏文件实现单键的快速查询与多键的并行管理，并且采用异地更新提升前台持久化和键值对更新效率，其垃圾回收由原生F2FS的节点segment流程支持。

对于后续的研究，内存桶也设计了一系列标记位，可以选用对重复数据感知的缓存替换算法D-ARC或D-LRU，优化逐出策略（CacheDedup等工作表明，传统LRU算法并没有将重复数据热度管理发挥到最佳）；在前缀树中，键值对不需要严格有序，我们可以在后续研究中利用这一特点，将访问热度或重复率较高的键值转移到前缀树前端，减少稳定化与离线去重操作树的访问深度。

**3.3过期键值对的移除**

当数据块引用次数降为0后，除了需要释放该块所占物理空间，还需要移除持久化键值管理的过期键值对，传统支持去重的系统对于这一操作有以下两种方式：

1. 在数据块无效之前读出该块，重新计算哈希得到指纹（即键），然后在相关持久化信息进行移除（比如通过见移除B树或LSM树的过期键值）。但是由于文件系统对读写等场景操作性能要求较高，例如F2FS，其只通过段位图信息表（Segment Info Table, SIT）和旧物理地址无效掉旧数据，而不会将该数据读出。这种频繁的随机读操作对文件删除、数据截断与数据的异地更新都有很大影响，只适合对在线读写要求不高的归档系统使用，不适合终端存储系统。
2. 在逻辑地址到物理地址相关翻译过程记录完整指纹信息，例如块层去重技术Dmdedup记录了“逻辑地址索引”（逻辑地址→物理地址+完整指纹）与“哈希索引”（完整指纹→物理地址+引用计数）两部分，当逻辑地址无效后可以快速得到完整指纹，然后到哈希索引中进行计数器的修改。然而这种设计模式存在两个严重问题，首先是完整指纹的长度远超地址和计数器之和，而且还需要记录两次，相关元数据管理开销极大；其次是使用这种索引结构的系统都需要进行哈希计算，也即100%参与在线去重，这对负载较大的工作场景很不友好，不适合对系统资源和性能都要求较高的智能终端设备。



**图7 键值对的稳定化**

我们研究了重复数据键（FP）值（VA）对存储的两个特点，即键值之间关系是严格的一对一，不存在一个值对应多个键的场景（只要数据本身不修改，每个VA对应数据块FP一定不变），而相同键不同值的场景我们通过“键值对的稳定化”操作来实现。此外在某一时刻之前，系统在所有持久化信息中拥有的有效值一定唯一（不存在一个VA在某一时刻指向多个不同数据块）。利用这两点，我们可以得出以下结论：在每次键值对缓冲区与前缀树实现稳定化后，即所有相关键值都存储到保留唯一键值对的前缀树里，删除一个过期键值只需要使用“FP前缀+VA”，而不需要完整FP。

如图7，F2DFS过期键值移除包含以下步骤：

* 1. 利用VAT表项里保留的前缀访问对应的桶（对于未经在线去重的数据，VAT里记录的前缀无效），在桶内查询该键值是否已经持久化（由于LRU算法已经逐出到外存缓存区，或找到了相关键值但标记为已持久化过），对于未经持久化的键值F2DFS只需在桶中删除该表项即可，对于已持久化的会在内存桶记录待删除的过期值（VA）；
  2. 为了节省内存空间，F2DFS将桶分为“有效数据”与“待删除数据”区域，后者复用了前者的指纹存储内存，即一个桶表项可以存储1条有效键值，也可以存储6条待删除键值，两个区域的空间动态调整；
  3. 等待触发桶的持久化，将待删除数据持久化到缓冲区里，为了避免误删桶内已有键值对，该步骤先刷写待删除表项；
  4. 在稳定化过程，用待删除VA移除过期表项，先寻找缓冲区里先于该信息的键值（如图7 56789的删除），如果没有则在前缀树里寻找（如图7 01234的删除），最后再同步两个区域的唯一键值。

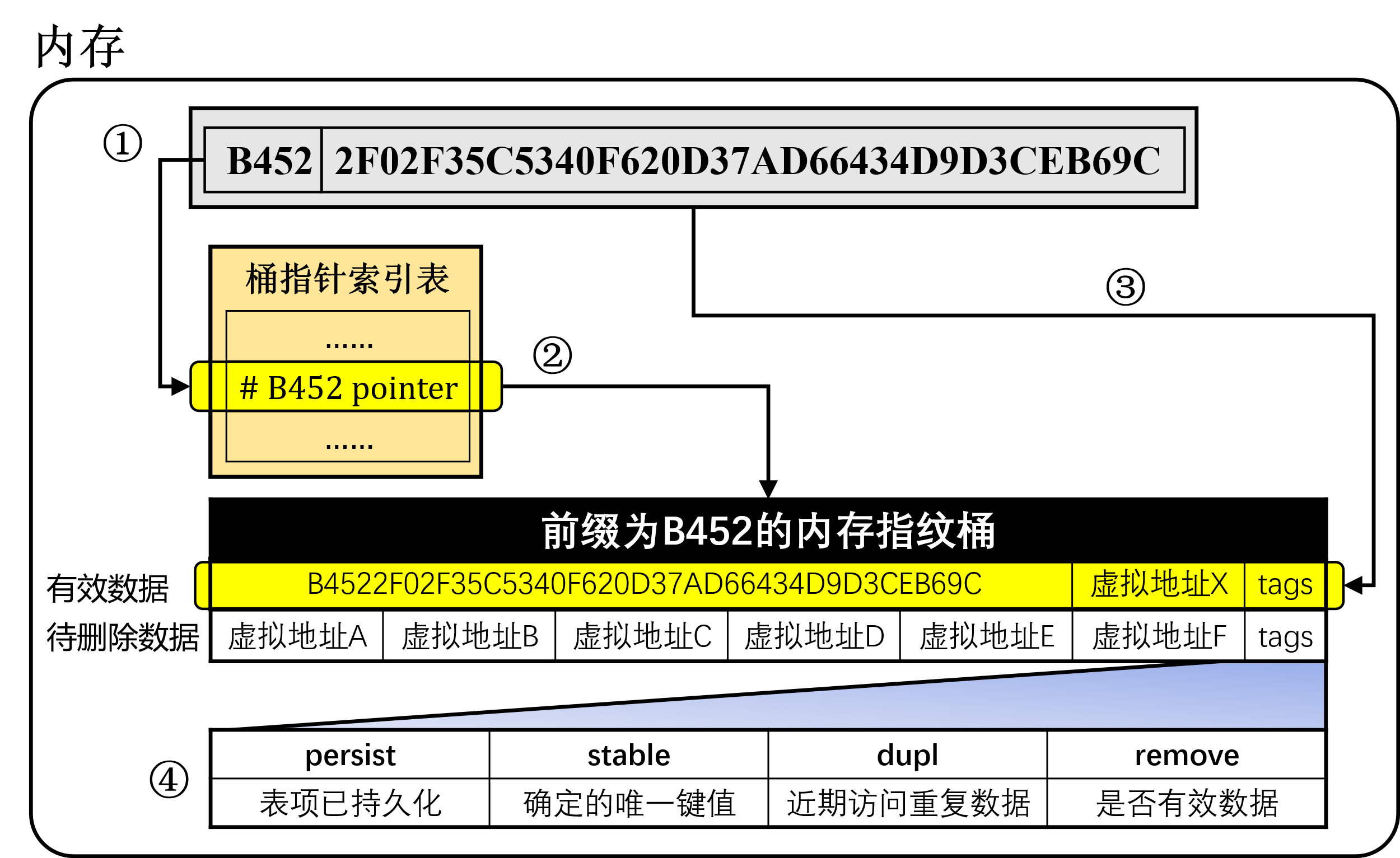
对于同步过程发现的相同键（如图7 98765与43210有着相同指纹），F2DFS只保留最先来的键值而不再将后来键值添加或更新到前缀树，具体操作留到离线去重阶段完成。

在我们的设计中，只需要“FP前缀+VA”即可移除过期键值对，而不需要额外读取旧数据块计算哈希，对于原型F2FS这种异地写更新频繁系统友好，也减少了完整FP存储的空间管理开销，有效解决了上述两个传统去重系统方案存在的问题。

**四、在线与离线结合的100%重复数据删除**

F2DFS采用在线与离线去重结合的方式，在写入路径上过滤重复数据，减少无用写入，延长闪存设备寿命，在系统空闲时刻由定时线程或用户手动触发离线去重，在控制写入量的同时节省存储空间。通过虚拟地址和键值对存储技术，在每次离线去重结束，可以确定系统只保留每个数据的唯一副本，实现100%冗余数据外存空间回收，同时，在线去重也可以减少离线去重执行时访问外存的次数。

**4.1在线重复数据删除**



**图8 在线去重流程图**

我们在写函数write\_datapage(s)实现在线去重（见“文件写流程”），如图8所示，具体过程包括以下步骤：

* 1. 对Page Cache中的数据进行4KB为单位的指纹计算；
  2. 通过指纹前缀访问对应的内存桶；
  3. 在桶内进行查找，如果未找到则需要申请新的VA然后插入新键值，反正则将查到的重复VA的RC+1，并更新桶内键值对热度；
  4. 更新桶内标记为，并根据标记位信息选择是否标记该逻辑地址为“待离线”。

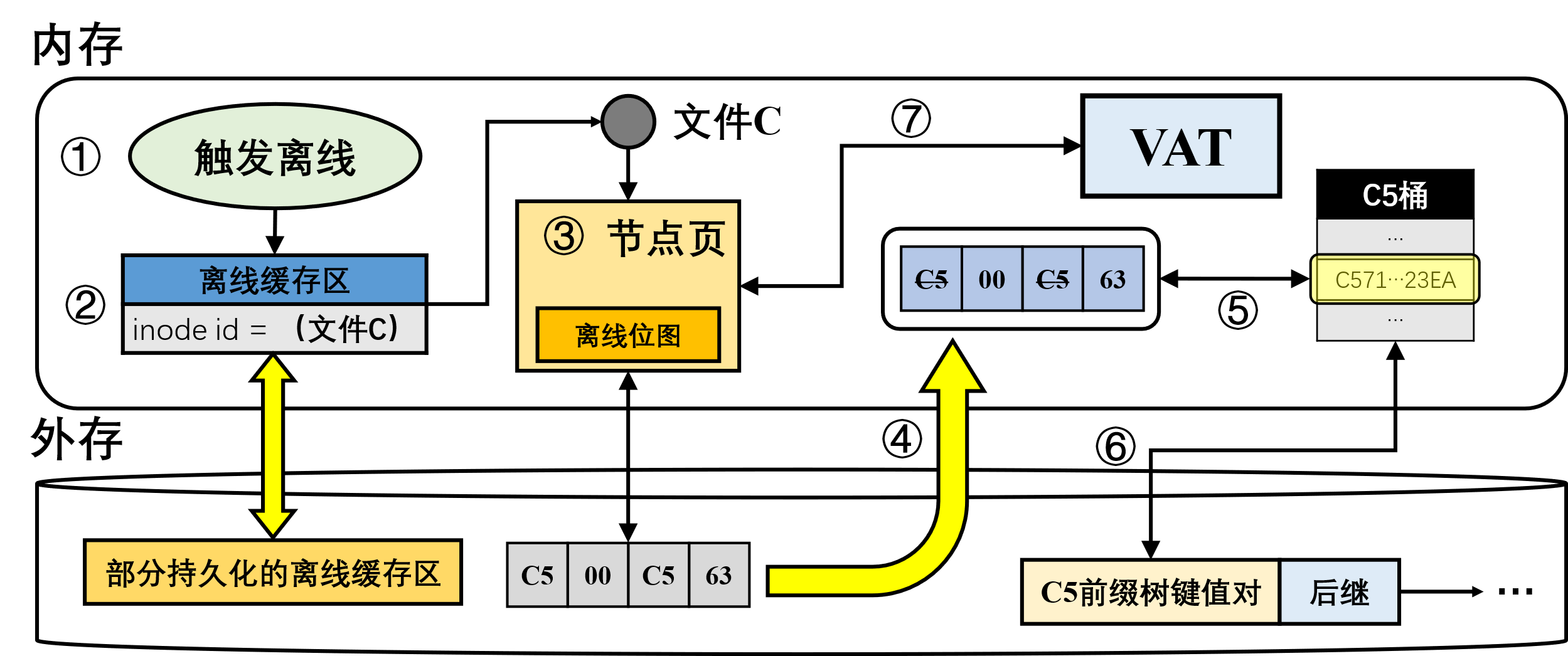
在桶中，每个键值对有4个标记位（占1B），分别是persist、stable、dupl和remove，persist代表该键值是否已经持久化，如果该键值被删除则需要将对应VA插入待删除数据；stable表示该键值是否为前缀树中唯一的键值对，如果是则该逻辑地址跳过离线去重阶段并将其在离线位图标记为“已离线”（见“离线去重”）；dupl表示该键值是否是近期访问的重复数据，可以为D-ARC或D-LRU等重复数据感知的缓存替换算法服务，它们能够有效减少冷数据批量查询导致的重复数据热键值被频繁逐出的概率，适合重复率较高的场景，并且没有破坏重复数据的时间局部性特征；remove用于判断该表项是有效数据还是待删除数据，对于每个待删除的键值，F2DFS将它们组织成一系列删除日志，每个删除日志大小仅有有效数据的键值的六分之一（当采用SHA1算法），在桶执行完持久化后，删除日志所占桶的内存空间可以立即被回收，重新用于键值存储。

**4.2离线重复数据删除**

传统离线去重方式一共分为以下两种：

1. 将某目录、设备或分区下所有数据全部读出，计算哈希再在这些相关文件之间进行重复数据删除，典型的有BTRFS和XFS。这种方式的问题在于已经去重过的数据与新来的数据之间可能存在重复数据，而无法将新来的数据记录下来以减少下次去重的数据处理量，每次100%系统去重都会读出已去重的数据（尽管BTRFS和XFS的extent粒度去重可以这一问题，但是相较细粒度的块级别去重空间回收占比很低）。
2. 通过记录新来数据的索引，在每次离线去重只处理这些数据可以解决（1）的读放大问题，但是以数据块为单位记录的内存代价过高，例如SmartDedup消耗0.5MB内存只能记录250MB数据，对于现代智能终端设备来说性价比很低。如果将这部分索引持久化来减少内存消耗，由存在无法移除过期索引等问题。
3. 设计一个全局位图，其中每个比特代表磁盘上每个数据块的状态，如果被标记则代表这个数据未经过在线去重或者可能是一个潜在的重复块，这种方式可以避免存储大量待离线去重数据块的记录，但是会增加全局位图更新频率导致写放大，并且不支持基于某个目录下或者某个文件集合的指定范围的去重。

为了解决以上问题，F2DFS的离线去重设计了两个重要数据结构：离线缓存区，以及基于节点页的离线位图。离线位图设置在每个节点页里，它标识了该节点页每个偏移指向的数据块是否需要执行离线去重，随着节点页持久化，由于节点页的可靠性由原生F2FS的后滚恢复保障，我们并不需要额外设计数据恢复模块。离线缓存区是一个与文件系统高度耦合的记录区域，其每个表项包含三个部分（文件inode id 4B，节点node id 4B，逻辑离线数offline count 2B），inode id使得离线去重可以以单一文件为单位执行，保障不同文件之间的隔离性并利用预读提升单一文件离线去重效率；node id指向需要去重的节点页；逻辑离线数代表了该节点页里离线位图的有效位数，当该节点新增一个待离线标记，offline count+1，当待离线数据被删除或执行完离线去重则-1，归零后则删除该表项。F2DFS会在离线缓存区填满后将其部分持久化，在离线去重时现对已有的表项进行合并，当里面逻辑离线树归0即可删除表项，保证过期索引记录的移除。



**图9 离线去重流程图**

离线去整体流程如图9所示，包括以下步骤：

1. 由线程定时、用户手动或系统强制触发离线去重，在此之前需要完成所有前缀树键值的稳定化；
2. 根据离线缓存区中的信息执行以文件为单位的离线去重，该过程可能会读取此前部分持久化的信息；
3. 读取相关节点页，通过离线位图还原需要读取的虚拟指针，并通过VAT得到这些指针对应的物理地址；
4. 将相关物理地址指向的物理数据块预读出来，为了保障预读效率，可能会读取一些离线位图不需要的数据；
5. 对这些数据进行指纹计算，将这些键值对更新到内存桶中；
6. 在该文件相关数据铅笔执行完，单一桶满或者节点页达到批处理上限后，利用前缀树批次查询特点与外存唯一键值对进行同步，如果是非重复数据则更新离线位图为“已离线”并结束，如果该键值并找到，还需要将其插入前缀树；
7. 将查询到的值（VA）对应的VAT表项RC+1，节点页中旧VA的RC-1，更新该节点页的虚拟指针，更新离线位图为“已离线”，全部结束后清空离线缓存区。

对于在线去重未经确认是否保留唯一副本的数据（即被标记为“待离线”），以及跳过在线去重阶段的数据（例如direct I/O，前台负载较大的工作集等），F2DFS会对这些数据执行离线去重。这保证了F2DFS每次只处理上次完整离线去重之后的数据，解决了传统方式（1）的弊端；同时，通过“离线缓存区+离线位图”结合，显著减少索引记录管理的内存开销，在相同内存消耗下，我们的设计可以索引的数据量是SmartDedup的800倍，并且可以灵活地移除过期索引，解决了传统方式（2）的弊端。同时，F2DFS支持用户指定某个文件或目录去重（通过文件inode id处理指定的离线缓存区条目），解决了传统方式（3）的弊端。

相较于BTRFS/ XFS等文件系统，F2DFS每次离线去重只需要处理上一次完整离线去重之后更新的数据（即增量的离线缓存区记录的表项），这使得F2DFS执行100%去重不需要读出外存全部数据，极大缩减了整体执行时间，避免了已去重过的数据被反复读出，对于多媒体文件等冷数据类型友好，同时对于有内容修改的文件，F2DFS只需要针对它们修改的部分进行离线，而不需要读取和处理整个文件。此外，由于BTRFS/ XFS自身特性，会将大量时间和系统资源集中在文件重映射，它们在细粒度执行效率、CPU使用率和内存驻留上显著劣于F2DFS。并且这两个文件系统离线执行时间随着重复率增长有剧烈变化，F2DFS的执行时间则很稳定。

值得说明的是，离线缓存区每个表项共计10B，却能最多记录整个节点页指向的逻辑数据，即每100GB逻辑数据的离线信息，只需要254KB内存即可全部记录，对于1GB碎片数据也只需要2.5MB内存。同样支持增量的离线去重记录的SmartDedup以及BEES（一个BTRFS的离线去重模块）由于每个数据块都记录的一次索引，每100GB逻辑数据和1GB碎片数据分别需要200MB和2MB内存，并且很难删除过期表项。

**五、崩溃一致性**

F2DFS数据恢复分为前滚（roll-forward）和后滚（roll-backward），都是基于F2FS原有数据恢复模式进行改进。简单来说，后滚恢复即基于最近创立的检查点保护整个系统的有效信息，前滚恢复是恢复最近创立的检查点以后写到磁盘并且可以恢复的数据。

**5.1简介**

图示

描述已自动生成

**图10 原生F2FS的数据恢复流程**

对于后滚恢复，容易理解，这里不做解释，我们主要针对前滚恢复进行修改。

首先介绍一下F2FS的前滚恢复流程：

1. 如图所示，初始时刻文件系统包含一个文件的旧节点，指向数据块A、B、C。
2. 然后F2FS会周期性建立检查点（checkpoint，ckpt），刷写内存中全部脏页（依次刷写所有的脏数据页，所有的脏节点页，所有的脏元数据页）。检查点成功建立后，在该时刻之前的磁盘所有结构均是可靠安全的（持久性设计）。
3. 之后，用户对于该文件节点进行了修改，比如图中将节点指向数据块A、B’（A不变，B修改为B’，删除C），并进行了该文件的强制持久化（fsync或者fdatasync等）。在F2FS中，为了避免不同文件的fsync导致系统频繁建立检查点，进而强制性刷回不属于该文件的数据与元数据（隔离性设计），fsync只会刷回属于该文件的修改数据页与修改节点页，并且给该节点页一个fsync标记位（fsync flag）。
4. 系统经历了突然的崩溃（crash），例如正常关机前的突然断电，或者系统崩溃。
5. 在重新挂载文件系统后，会触发恢复模块（recovery），首先将系统“后滚”到最近一次检查点（即③中所创建），但是如果只做到这一点，F2FS也只能读取旧节点指向的数据A、B、C，所有还需要经历后续的前滚恢复。
6. 在最近建立的检查点记录的节点块位置开始，向后查找具有fsync标记的节点
7. 将这些fsync的节点与检查点保护起来的旧节点（由于新旧节点具有相同的node id，而检查点保护的NAT记录了旧节点的物理地址，所以可以访问旧节点）进行对比，具体来说就是两个版本的节点各自记录的一系列数据块的物理指针的差异。
8. 将这些差异增量更新到检查点保护的旧的块分配位图SIT（本例中即无效掉数据B、C的物理地址，有效数据块B’的物理地址），将NAT中修改为新节点的物理地址，最后在恢复了所有数据后，建立一个检查点。

**5.2后滚恢复**

依旧是检查点，相比于F2FS，主要额外写回两个额外数据结构：

❶ VAT脏表项，包含自上个检查点后新分配的虚拟地址到物理地址，修改的RC，释放的虚拟地址（指向一个空物理地址）。由于F2DFS中VAT的消耗速度相比于原生F2FS的NAT消耗速度更快，意味着每次检查点可能写回更多VAT表项，我们后续会采用预读或者批量更新等策略缓解性能开销。

❷离线去重队列，很少，每20GB数据只需64KB，而且离线去重后可直接以释放占用的磁盘空间。

对于其他F2DFS新增的数据结构，例如“指纹→地址”键值对，键值存储结构的修改日志，离线去重位图等，都可以在前滚恢复或者离线去重阶段重建，因而没有必要在每次检查点时强制写回或维护。

**5.3前滚恢复**

F2DFS在写入路径上，执行两种策略，对于fsync等同步写入，文件节点页里存储数据块的物理地址（与原生F2FS一样）；对于异步写入，文件节点页里存储数据块的虚拟地址，而VAT记录虚拟地址到物理地址的映射（如图3）。这样，即便遇到意外断电，fsync的文件可以依照原生F2FS前滚恢复策略和函数进行数据恢复，在这个过程中，一些元数据类似离线位图、离线缓存区、自上次检查点后的新增加的删除日志等都可以得到重建，对于内存桶未及时持久化的键值，可以在离线去重中重新更新到前缀树。而异步写入则和F2FS一样，依靠检查点保证安全，而检查点会同时强制写回包含VAT在内的元数据，不需要额外处理。对于文件正常读写操作，我们在节点页里设计一个标记位（physical flag）表示指向的地址是物理地址还是虚拟地址，对于物理地址，就直接访问数据块，对于虚拟地址，则经过VAT翻译得到物理地址后再访问数据块，相关标记位占据的存储空间可以在离线去重分配虚拟地址后进行回收。