- **DupLESS: Serveraided encryption for deduplicated storage**

  USENIX Security Symposium 2013

**密钥怎么管理的，使得其可抵御暴力攻击?**

使用密钥服务器(KS)派生密钥来处理，而不是将密钥设置为消息的哈希值。具体来说，客户端将向KS发送文件的哈希值H，并接收返回的消息派生密钥K '←F(K,H)，其中K是只有KS保存的密钥，F为RSA OPRF。客户端的密钥用于加密派生密钥作为密钥封装密钥$C_{key}$，将$C_{key}$和密文存储在SS上。

如果两个客户端加密同一个文件，那么消息派生的密钥和Cdata将是相同的(密钥封装密钥不同，但是这个密文很小)。

- **Secure Distributed Deduplication Systems with Improved Reliability**

  IEEE TRANSACTIONS ON COMPUTERS 2015

新结构没有对数据进行加密以保持数据的机密性，而是利用秘密分割技术将数据分割成分片。然后，这些碎片将分布在多个存储服务器上。

TagGen 是映射原始数据副本F并输出标签T（F）的标签生成算法，应用于与服务器执行重复检查

TagGen'以文件F和索引j作为输入，输出一个标签。这个由用户生成的标签用于证明F的所有权。

**文件级去重过程**

1. 设置阶段

   假设有n个$s-csp$服务器，其身份为$id_1, id_2, id_3, \ldots, id_n$

2. 文件上传

   文件上传时先进行重复性检查，用户计算标签$\phi_F = TagGen(F)$发送给$s-csps$进行文件重复性检查。

   如果是重复，用户计算$\phi_{F,id_j} = TagGen'(F, id_j)$给第j个服务器，进行所有权检查，如果匹配，用户获得一个指向存储在服务器$id_j$上的分片的指针。

   否则，用户进行秘密共享算法得到$c_j = Share(F), c_j$是文件$F$第$j$块分片。将$\{\phi_F, c_j, \phi_{F,id_j}\}$发送给$id_j$的服务器。

3. 文件下载

   用户将F的指针发送到到n个s - csp中的k，获得其分片后恢复出文件。

**块级去重过程**

对于文件级不重复的文件，进行块级去重，分块后过程与文件级文件上传类似。

- **Deduplication on Encrypted Big Data in Cloud**

  IEEE TRANSACTIONS ON BIG DATA 2016

本文提出了一种基于所有权挑战和PRE的云中重复数据删除加密管理的实用方案。可以灵活地支持数据更新和重复数据删除共享，即使在数据持有者离线的情况下。加密的数据可以安全地访问，因为只有授权的数据持有者才能获得用于数据解密的对称密钥。

**主要目标:**

目标是解决数据持有者不可用或难以参与的情况下的重复数据删除问题，本方案中重复数据删除的性能不受大数据的影响。

在处理大数据重复数据删除方面，与"A scheme to manage encrypted data storage with deduplication in cloud" ICA3PP 2015 比较计算开销更小。

TABLE 2
Computation Complexity of System Entities by Comparing with [33]

| Entity | Algorithm | Computations [our scheme] | Computations [33] | Complexity |
|---|---|---|---|---|
| Data Owner | Setup | 1*PointMulti + 1* Exp | 1*ModInv + 1*Exp | $\mathcal{O}(1)$ |
| | Data upload | 2 *Exp + 1*PointMulti | 3*Exp | |
| CSP | Re-encryption | 1 *Pair | 1*Pair | $\mathcal{O}(n)$ |
| Data Holder | System Setup | 1*PointMulti + 1*Exp | 1*ModInv + 1* Exp | $\mathcal{O}(1)$ |
| | Challenge Response | 2*Exp + 1*PointMulti | – | |
| | Data upload | – | 3*Exp | |
| | Decryption for access | 1*Exp | 1*Exp | |
| AP | System Setup | 1*Exp | 1*Exp | $\mathcal{O}(n)$ |
| | Ownership challenge and re-encryption key generation | 2*Exp + 2*Point Multi | 1*Exp | |

*Notes: Pair: Bilinear Pairing; Exp: Exponentiation; ModInv: Modular Inversion;* n*: Number of data holders who share the same data; PointMulti: Point multiplication in ECC.*

因为点乘法比幂运算更有效。

AES在数据加解密方面的效率很高，因此在大数据方面的应用是切实可行的。

TABLE 6
Total Operation Time of Cloud Users in Our Proposed Scheme and [33] (Unit: millisecond)

| Entity | Total time [our scheme] | Total time [33] |
|---|---|---|
| **Data Owner** | 150.4 | 151.2 |
| **Data Holder** | 4.91 | 151.2 |

数据持有者可以节约大量时间。

- **Decentralized Server-Aided Encryption for Secure Deduplication in Cloud Storage**

  IEEE TRANSACTIONS ON SERVICES COMPUTING 2020

**动机：**

问题1：KS内重复数据删除指在单个KS或一组KS共享密钥，用KS保存的密钥下生成的密文。以前的方案限制了CSP执行这种跨租户重复数据删除的能力。意味着密钥不能在不同的租户之间共享。使用不同的密钥，即使对于相同的数据，也会生成完全不同的收敛密钥，从而最终产生不相同的密文。

问题2：用户在上传数据前需要与KS进行交互，随着用户数量的增加，这种交互成为文件上传过程的性能瓶颈。为了保证高可用性，租户可能希望部署足够数量的KSs，以便随着用户的增长进行扩展。然而多个KS都维护各自的密钥K，就会导致问题1，若多个KS共享一个密钥，会导致严重的安全问题即一个KS泄露密钥将导致系统崩溃。

**核心思想：**

提出了一种新的分散式服务器辅助加密重复数据删除方案，解决了上述问题。具体来说，提议的方案旨在为csp提供在加密数据上执行跨租户重复数据删除的能力，同时提供为每个租户部署KS的灵活性。为了实现这一点，我们将以前方案的集中式KS设置扩展为由多个KS组成的分散设置，每个KS都能够独立设置其密钥，而不需要任何集中的协调实体。

**跨域KS重复数据删除算法：**

**Algorithm 1.** Inter-KS Deduplication Algorithm

**Require:** $T$, $S_{all}$ ▷ $T = (i, t_F, C)$
**Ensure:** $S_{all}$
1: $DuplicateFound \leftarrow$ False
2: **for each** $T' \in S_{all}$ **do** ▷ $T' = (j, t_{F'}, C')$
3: **if** $V_{DDH}(t_F, t_{F'}, g^{x_i}, g^{x_j}) =$ True **then**
4: Replace $C$ in $T$ with $l_{T'}$
5: $S_{all} \leftarrow S_{all} \cup (i, t_F, l_{T'})$
6: $DuplicateFound \leftarrow$ True
7: **break**
8: **end if**
9: **end for**
10: **if** $DuplicateFound =$ False **then**
11: $S_{all} \leftarrow S_{all} \cup (i, t_F, C)$
12: **end if**

$t_F$表示文件$F$的标签为$h（F）^{x_i}, x_i$为序号$i$的$KS$的私钥，设$S_{all}$为存储在存储器中的所有元组的集合($S_{all}$实际上代表$CSP$的存储器)。

如果输出为$True$，则$CSP$可以确定$F = F^{'}$，并将$T$中的$C$替换为指向另一个元组$T^{'}$的逻辑链接$l_T$。

第三行通过第i，j个KS的公钥核实文件$F, F'$的标签是否一致。

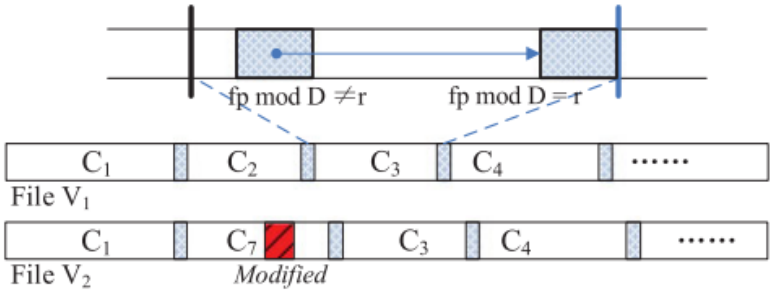
---

- **The Design of Fast Content-Defined Chunking for Data Deduplication Based Storage Systems**

  IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS 2020

**背景:**

固定大小分块FSC：最简单的分块方法，但存在边界移动问题




基于内容分块CDC：根据数据流的字节内容声明块边界，可以检测更多的重复数据删除冗余。

CDC的两个阶段：(1)哈希，在此阶段生成数据内容的指纹;(2)哈希判断，在此阶段将指纹与给定值进行比较，以识别和声明块切点。

基于Rabin指纹的CDC：目前最流行的CDC方法，在重复检测方面非常有些，由于它是逐字节计算和判断(根据条件值)数据流的Rabin指纹，所有非常耗时。

为了加快CDC通过两种途径，1）提出更加快速的哈希算法替代Rabin算法；2）采用多核处理器或GPU用来加速。

根据本篇论文所述，Gear（Ddelta: A deduplication-inspired fast delta compression approach 提出）似乎是CDC目前最快的滚动哈希算法之一，因为它使用的计算操作比其他算法少得多。

基于Gear的CDC的三个观察现象：

1. 低数据去重率，因为在哈希判断阶段，滑动窗口大小低。
2. 由于因为Gear加速了哈希阶段，将瓶颈转移到了哈希判断阶段
3. 扩大预定义的最小块大小(在CDC中使用，以避免非常小的块[6])可以进一步加快分块过程（切点跳过），因为许多具有跳过截断点的块并没有真正根据数据内容(即内容定义)进行划分。

**核心方案**：提出了FastCDC，一种快速有效的内容定义分块方法，用于基于数据重复数据删除的存储系统。结合基于Gear的滚动哈希、优化哈希判断、次最小块切点跳过、规格化分块、每次滚动两个字节五项技术来加速CDC。

- **基于Gear的滚动哈希**

  部署一个256个随机64位整数数组来映射滑动窗口中字节内容。

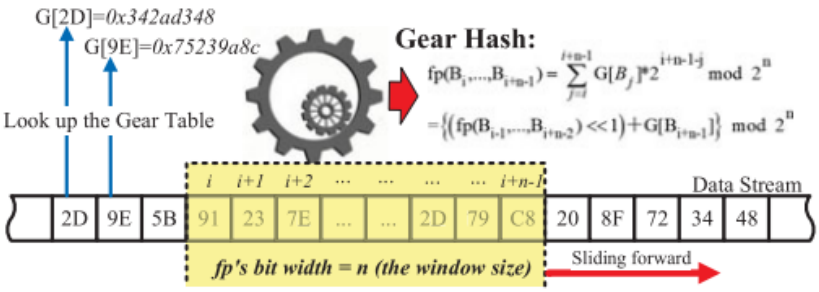



Fig. 4. A schematic diagram of the Gear hash.

TABLE 1
The Hashing Stage of the Rabin- and Gear-Based CDC

| Name | Pseudocode | Speed |
|---|---|---|
| Rabin | $fp = ((fp^{\wedge}U(a)) << 8)\|b^{\wedge}T[fp >> N]$ | Slow |
| Gear | $fp = (fp << 1) + G(b)$ | Fast |

*Here 'a' and 'b' denote contents of the first and last byte of the sliding window respectively, 'N' is the length of the content-defined sliding window, and 'U', 'T', 'G' denote the predefined arrays [6], [11], [13]. 'fp' represents the fingerprint of the sliding window.*

问题：1）由于滑动窗口大小的限制造成低去重率；2）昂贵的哈希判断，哈希判断阶段占CDC期间CPU开销的60%以上。

- **优化哈希判断**

**优化1：通过0填充扩大滑动窗口大小**

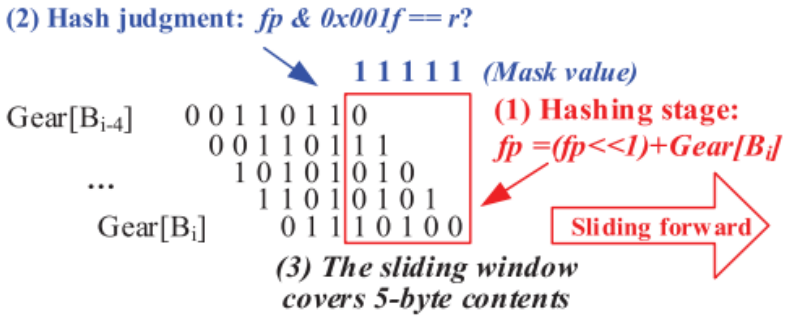



Fig. 6. An example of the sliding window technique used in the Gear-based CDC. Here CDC consists of two stages: hashing and hash judgment. The size of the sliding window used for hash judgment is only 5 bytes because of the computation principles of the Gear hash.

滑动窗口用于哈希判断只有5个字节

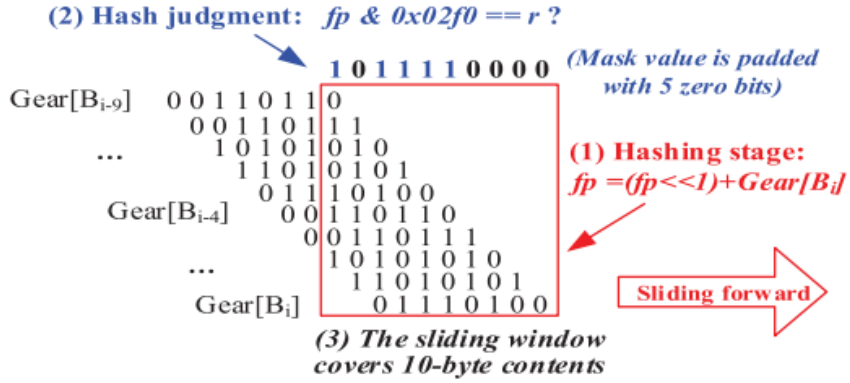



Fig. 7. An example of the sliding window technique proposed for FastCDC. By padding $y$ zero bits into the mask value for hash judgment, the size of the sliding window used in FastCDC is enlarged to about $5+y$ bytes, where $y=5$ in this example.

如图中填充了5个0是的滑动窗口大小扩大为10个字节。

**优化2：简化哈希判断来加速CDC**

在基于rabin的CDC中使用的常规哈希判断过程表示为$fpmodD == r$,定义$D$为$0x02000$,$r$为$0x78$以获得预期的平均块大小为8 KB。

将哈希判断语句优化为$fp\&Mask == 0$减少寄存器存储r的空间，避免了与r比较操作。

- **切点跳过**

  块大小X的累积分布通常遵循如下指数分布：

  $$P(X \leq x) = F(x) = (1 - e^{-\frac{x}{8192}}), x \geq 0$$

  由式可知，小于2 KB和大于64 KB的块分别约占总块数的22.12%和0.03%,在分块之前跳过切点以避免生成小于规定的最小块大小的块，或称为次最小块切点跳过，将显著影响重复数据删除比率。

- **规格化分块**

  规范化分块生成的块，其大小被归一化为以预期块大小为中心的指定区域。在规范化分块的实现中，我们选择性地改变哈希判断语句的有效掩码位的数量。

  对于预期块大小为8 KB(即213)的传统CDC方法，使用13个有效掩码位进行哈希判断。当前分块位置小于8 KB时，超过13个有效掩码位用于哈希判断这使得生成小于8 KB的块变得更加困难。

  归一化分块（NC）的最高归一化水平相当于固定大小分块(FSC)，即所有的块大小都归一化为等于预期的块大小。由于FSC的重复数据删除率非常低，但分块速度非常高，这意味着在归一化水平、重复数据删除率和分块速度之间会存在一个“甜点”。

- **每次滚动两个字节**

  因为与传统方法(每次滚动一个字节，如算法3.2所示)相比，它减少了一次移位操作，同时确保了完全相同的分块结果。注意，它需要再查找一个表，并增加额外的计算操作但是这些开销很小

- **SS-CDC: A Two-stage Parallel Content-Defined Chunking for Deduplicating Backup Storage**

  SYSTOR 2019

  **动机：**

  现有的在并行硬件上运行算法来加速CDC，通过将将输入文件划分为段的方法，并使用线程独立地对段进行块处理。通过这种方法，我们可以利用多核处理器来实现并行分块。但是，它不能保证分块不变性，并且会影响重复数据删除比率。而且它不能充分利用SIMD硬件上的并行性。

  分块不变性:无论多线程程度或段大小，分块的结果都不会改变。

  **核心方案：**

  提出了一种两阶段并行的SS-CDC，它将分块过程分为两个阶段，一个是滚动窗口计算，以生成所有可能的块边界，这是昂贵的，但可以在不同的段中并行执行；另一个是从候选块边界中选择块边界，使它们满足最小和最大块大小的要求，其执行必须跨段序列化，但轻量级。

  **优势：**

  **Table 1: Comparison of existing parallel chunking algorithms with SS-CDC**

  | | Chunking invariability | Multi-core | GPU | AVX |
  |---|---|---|---|---|
  | P-Dedup | No | Yes | No | No |
  | MUCH | Yes | Yes | No | No |
  | Shredder | No | No | Yes | Maybe |
  | SS-CDC | Yes | Yes | Yes | Yes |

  SS-CDC是保证分块不变性的唯一方法，同时支持在多核处理器、AVX和gpu上并行分块。

  **效率比较：**

  单核：

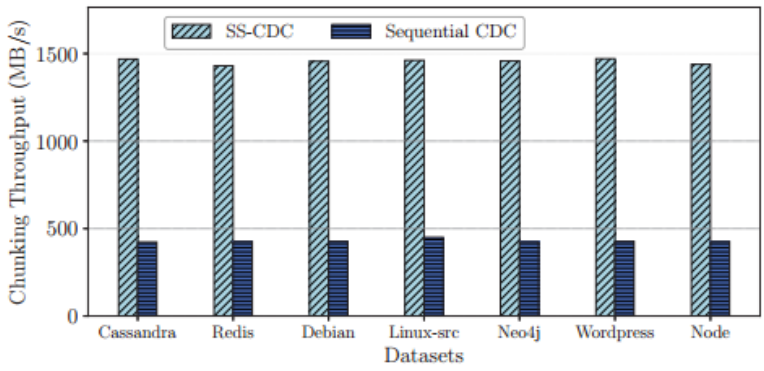
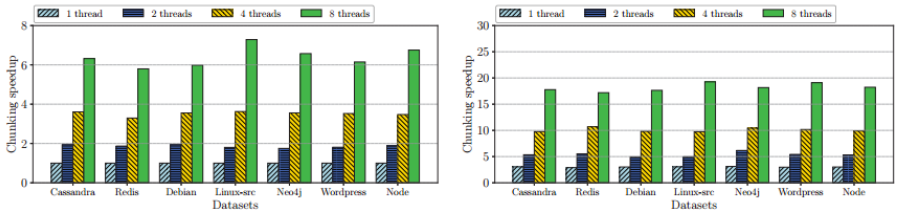



**Figure 4: Chunking speed of single-threaded SS-CDC and sequential CDC with one core. The minimum, expected average and maximum chunk sizes are 2KB, 16KB, and 64KB, respectively.**

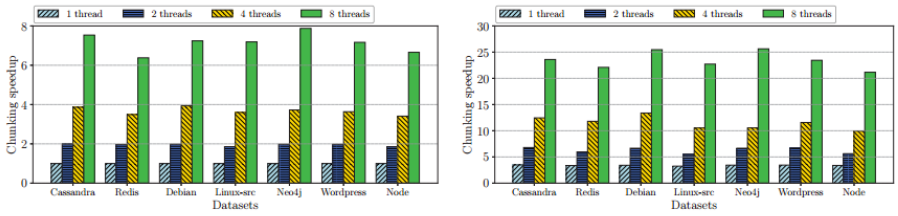
在一个内核上运行一个线程的SS-CDC和顺序CDC的分块吞吐量。尽管不同的数据集具有不同的重复数据删除比率，但速度提升非常一致，大约为3.3倍。

多核：




(a) SFSS Regular CDC (b) SFMS SS-CDC

**Figure 6: Chunking speedups of multithreading regular CDC and multithreading SS-CDC over sequential CDC at one core with different datasets and thread/core counts when a file is processed by all threads.**




(a) MFSS Regular CDC (b) MFMS SS-CDC

**Figure 7: Chunking speedups of multithreading regular CDC and multithreading SS-CDC over sequential CDC at one core with different datasets and thread/core counts when each file is processed by one thread.**

研究了多线程SFMS(针对单个文件将SSCDC扩展到多核)和MFMS(针对多个文件将SS-CDC扩展到多核)，将其与与不使用AVX的多线程常规CDC方法(SFSS和MFSS)进行了比较。可以看出多线程常规CDC获得的加速大约与内核(或线程)的数量成正比，使用AVX指令，多线程SS-CDC实现了超线性分块速度，但SFSS不能实现MFSS一样的加速。

- **RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-based Deduplication Systems**

  SoCC 2019

**核心方案：**

利用了重复数据中存在的一个属性，称为duplicate locality，一个重复的块之后可能紧跟着一系列连续的重复块。序列越长，局部性越强。

RapidCDC，它有两个显著特性。其一，其效率与重复数据删除比率呈正相关。当应用于具有高数据冗余的数据集时，RapidCDC可以与固定大小的分块方法一样快。另一个特点是它的高效率不依赖于高的重复局部强度。

在RapidCDC中，块的指纹与其下一个块的大小一起记录在文件中，一旦检测到重复的块，就可以使用它作为提示来定位下一个块的最有可能的边界，而不需要逐字节窗口滚动。如果该位置被接受，RapidCDC将避免数千次滚动窗口，每次滚动一个字节以到达下一个块边界；如果不被接受，它将在重复块指纹的大小列表中尝试另一个下一个块大小。只有当列表中的所有大小都不被接受时，RapidCDC才会将窗口移回最后一个块边界加上最小块大小的位置。

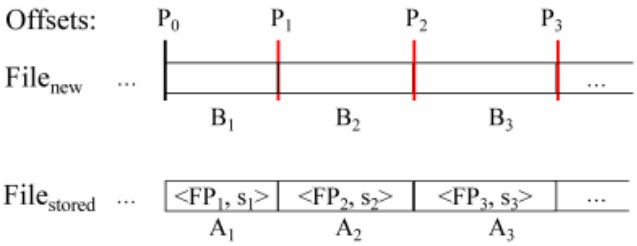



**Figure 5: An illustration of the idea of RapidCDC for rapidly determining chunk boundaries. $A_k$ and $B_k$, where $k$ = 1, 2, or 3, are chunks. $FP_k$ are fingerprints of chunk $A_k$. And $s_k$ are size (in bytes) of chunk $A_k$.**

重复块B1可以使用更简单的关系链(B1→FP1→s2)来获得其下一个块B2的建议大小(s2)。

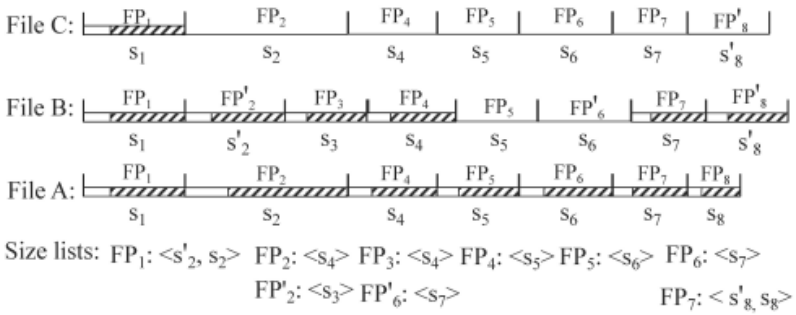



**Figure 7: Use of size lists to accelerate CDC (the shown size lists reflect their contents after Files $A$ and $B$ are stored and before File $C$ is stored.)**

- **Accelerating content-defined-chunking based data deduplication by exploiting parallelism**

  FutureGenerationComputerSystem 2019

  **主要思想：**

  首先将数据流分成几个段(即“Map”)，其中每个段将以独立的线程并行运行CDC，然后重新chunk并连接这些段的边界(即“Reduce”)，以确保并行化CDC的分块有效性。

  **管道去重：**

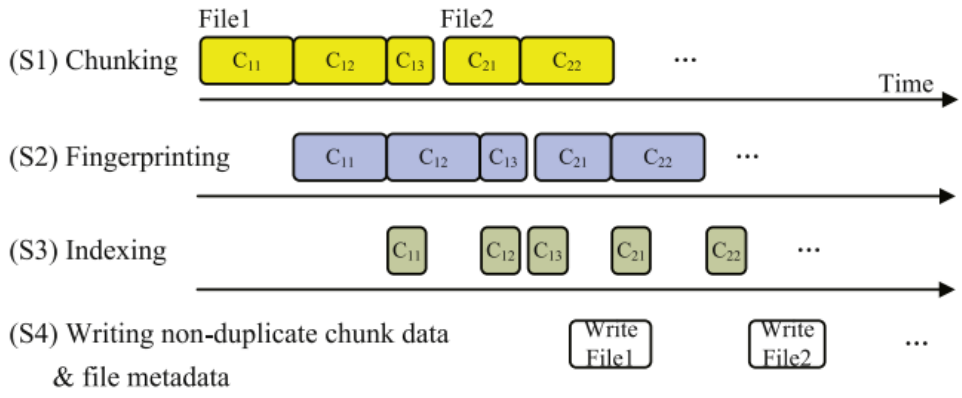



**Fig. 5.** Pipelining Deduplication in P-Dedupe system.

P-Dedupe的流水线技术将重复数据删除过程分为四个阶段:

(S1)内容定义的分块(CDC)、(S2)基于安全哈希的指纹识别、(S3)查找指纹、(S4)存储块数据和元数据。

**Mapreduce内容定义分块：**

P-Dedupe首先将数据流分割成许多固定大小的数据段(即“Map”)，每个段的大小应该远远大于CDC预定义的最大块大小，然后CDC并发地在数据段上运行。在这些段的CDC单独完成后，这些段之间的边界将相应地进行轻微的修正(即“Reduce”)。

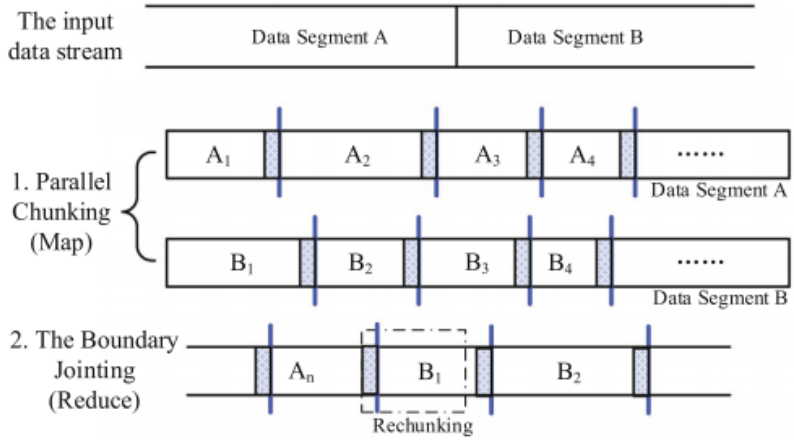
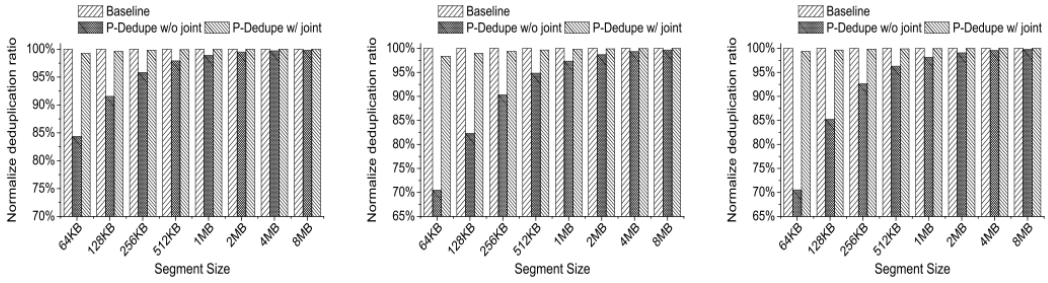


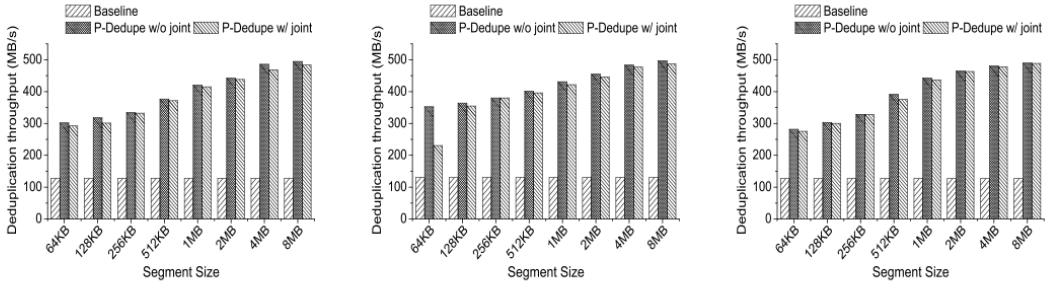

**Fig. 6.** The parallel CDC runs concurrently with two threads. The CDC task is “mapreduced”: the data stream is mapped to two segments *A* and *B* and then the boundaries of segments *A* and *B* are re-chunked for correction.

在效率方面，P-Dedupe系统与传统串行重复数据删除相比，在不降低重复删除率的情况下，重复数据删除吞吐量提高了3到4倍。




(a) Linux Dataset (b) Freebsd Dataset (c) Redis Dataset

**Fig. 9.** Deduplication ratio as a function of different parallel data segment size among the approaches of Baseline, P-Dedupe with and without segment boundary jointing.




(a) Linux Dataset (b) Freebsd Dataset (c) Redis Dataset

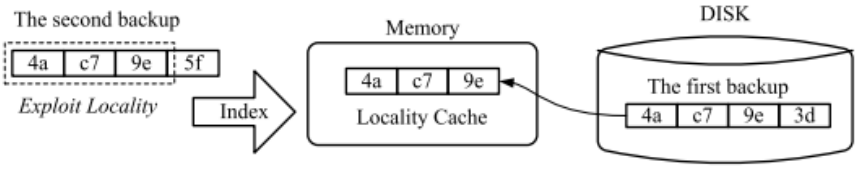
- **Similarity and Locality Based Indexing for High Performance Data Deduplication**

IEEE TRANSACTIONS ON COMPUTERS 2015

**相关背景：**

目前，加快重复数据删除索引查找和缓解磁盘瓶颈的方法一般有两种，即基于局部性的方法和基于相似性的方法。

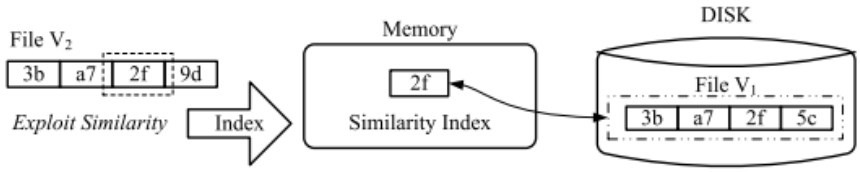
局部性：在重复数据删除的背景下，局部性是指一个备份流中相似或相同的文件，如A、B和C(即它们的数据块)，在多个完整备份中以非常高的概率以大致相同的顺序出现。




(a)

存储在磁盘中第一个备份流的指纹是{4a,c7,9e,3d},当查找到第二次备份的指纹“4a”时，DDFS会预取指纹{4a, c7, 9e}，并将该位置保存在RAM中。

相似性：相似度是指文件或数据流的相似特征，例如，可以提取的块指纹集的最大值或最小值，以表示该文件或数据流。




(b)

组块指纹{3b, a7, 2f, 9d}和{3b, a7, 2f, 5c}分别属于文件V1和V2，该文件的前缀位在所有指纹的相同前缀位中表示最小值。因此，当检测到文件V2的最小指纹“2f”与文件V1的最小指纹“2f”相同时，我们可以认为这两个文件相似，然后检测文件V1和V2之间的重复块，从而避免对文件V2的块指纹进行全局索引。

**核心思想：**是通过将强相关的小文件分组到一个段中并对大文件进行分段来暴露和利用更多的相似性，并通过将连续的段分组到块中来利用数据流中的局部性，以捕获被概率相似性检测遗漏的相似和重复数据。

段的概念用于利用备份流的相似性，而块则保留磁盘上的流通知段的局部性布局。

**工作流：**

首先对强相关的小文件进行分组和对大文件进行分段，然后进行块化、指纹化和打包成段。对于输入段$S_{new}$, SiLo将经历以下关键步骤：

1）检查$S_{new}$是否在相似性哈希表SHTable中，SiLo检查包含$S_{new}$的类似段的block $B_{bk}$是否在缓存中，不在将其加载到缓存中。

2）通过$B_{bk}$缓存中的局部性哈希表（块索引）检查$S_{new}$的指纹集来检测和消除$S_{new}$中的重复的chunk。

3）若$S_{new}$不在相似性哈希表SHTable中，则根据最近访问的block来检查潜在的相似段。

然后将输入段构造成block，以保留输入备份流的访问局部性。对于输入块$B_{new}$, SiLo执行以下操作:

1）检查$B_{new}$的代表性指纹来确定其存储的备份节点

2）将$B_{new}$写入缓冲区

**效率比较：**

论文将本文提出的SiLo系统与基于相似性的Extreme Binning系统和基于位置的ChunkStash系统进行比较。SiLo- 2mb和SiLo-4mb分别表示段大小为2和4mb的SiLo

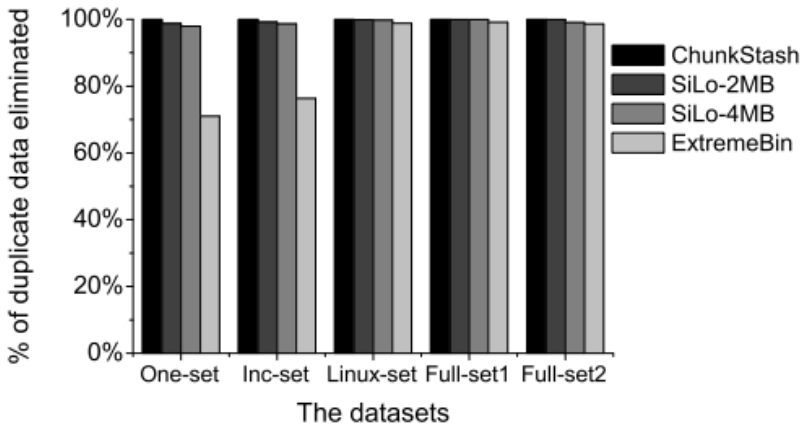



Fig. 15. Comparison among ChunkStash, SiLo, and Extreme Binning in terms of percentage of duplicate data eliminated on the five data sets.

Extreme bininning未能检测到One-set中具有弱局部性和相似性的近30%的重复数据，以及Inc-set中具有弱局部性但强相似性的大约25%的重复数据。由于ChunkStash执行了精确的重复数据删除，因此它消除了100%的重复数据。SiLo消除了98.5 ~99.9%的重复数据。

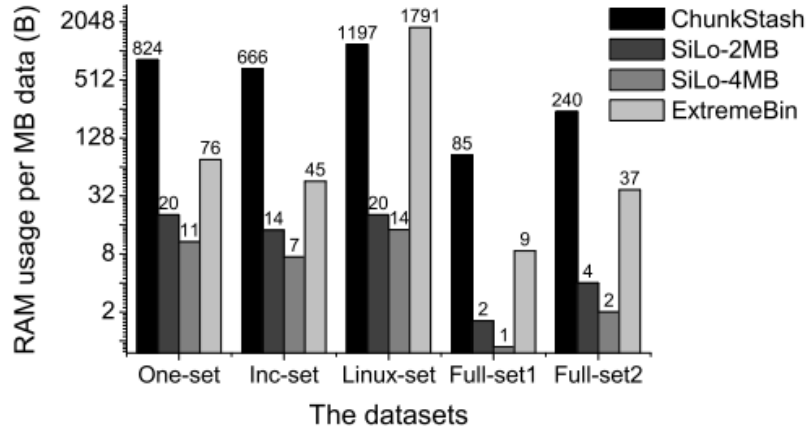



Fig. 16. Comparisons among ChunkStash, SiLo, and Extreme Binning in terms of RAM usage (B: RAM required per MB backup data).

对于具有非常多的小文件和小块的Linuxset, Chunkstash和Extreme bininning都产生了最高的RAM使用。ChunkStash的平均RAM使用量在三种方法中是最高的，因为它执行的重复数据删除需要在内存中使用一个大哈希表来放置块指纹的所有索引。

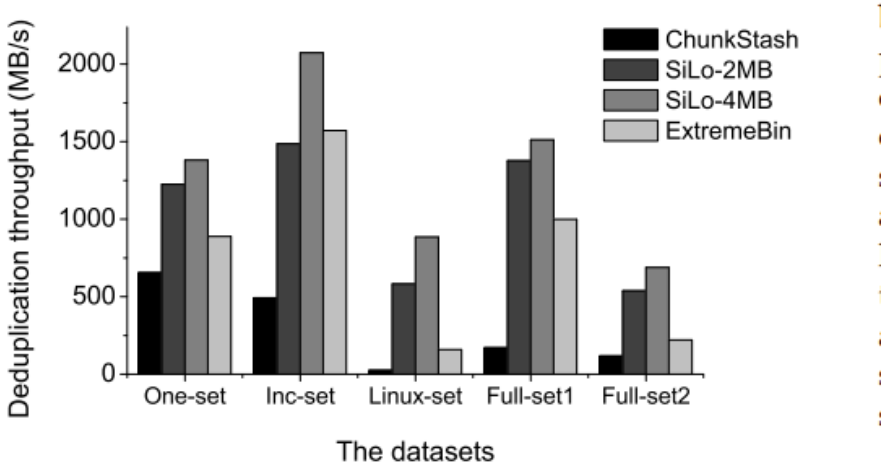



Fig. 17. Comparison among ChunkStash, SiLo, and Extreme Binning in terms of deduplication throughput (MB/sec).

ChunkStash在五个数据集上的平均吞吐量约为292 MB/秒,Extreme Binning在五个数据集上实现了768 MB/秒的平均吞吐量，与Extreme Binning和ChunkStash相比，SiLo提供了稳健且始终如一的良好重复数据删除性能，实现了更高的吞吐量，并以更低的RAM开销实现了近乎精确的重复消除。