Lazy Exact Deduplication

Jingwei Ma, Rebecca J. Stones, Yuxiang Ma, Jingui Wang, Junjie Ren, Gang Wang, Xiaoguang Liu

College of Computer and Control Engineering, Nankai University, China.

5 May 2016
Lazy exact deduplication
Lazy exact deduplication

Lead author: Jingwei Ma, PhD student at Nankai University (supervisor: Prof. Gang Wang).
Lazy exact deduplication

Lead author: Jingwei Ma, PhD student at Nankai University (supervisor: Prof. Gang Wang). Couldn’t get USA visa in time ⇒ I will present this work.
Lazy exact deduplication

Lead author: Jingwei Ma, PhD student at Nankai University (supervisor: Prof. Gang Wang). Couldn’t get USA visa in time \(\implies\) I will present this work.

Credit where credit is due: Jingwei Ma did the lion’s share of this work (development, implementation, experimentation, etc.).
Lazy exact deduplication

Lead author: Jingwei Ma, PhD student at Nankai University (supervisor: Prof. Gang Wang). Couldn’t get USA visa in time ⇒ I will present this work.

Credit where credit is due: Jingwei Ma did the lion’s share of this work (development, implementation, experimentation, etc.).

Lazy deduplication: ‘Lazy’ in the sense that we postpone disk lookups, until we can do them as a batch.
Lazy exact deduplication

Lead author: Jingwei Ma, PhD student at Nankai University (supervisor: Prof. Gang Wang). Couldn’t get USA visa in time $\implies$ I will present this work.

Credit where credit is due: Jingwei Ma did the lion’s share of this work (development, implementation, experimentation, etc.).

Lazy deduplication: ‘Lazy’ in the sense that we postpone disk lookups, until we can do them as a batch. (Lazy is exact.)
Deduplication: What usually happens...

We have a large amount of data, with lots of duplicate data (e.g. weekly backups).
Deduplication: What usually happens...

- We have a large amount of data, with lots of duplicate data (e.g. weekly backups).
- We read through the data, and if we see something we’ve seen before, we replace it with an index entry (saving disk space).
Deduplication: What usually happens...

- We have a large amount of data, with lots of duplicate data (e.g. weekly backups).
- We read through the data, and if we see something we’ve seen before, we replace it with an index entry (saving disk space).
Deduplication: What usually happens...

- We have a large amount of data, with lots of duplicate data (e.g. weekly backups).
- We read through the data, and if we see something we’ve seen before, we replace it with an index entry (saving disk space).

- The data is broken up into *chunks* (Rabin Hash).
Deduplication: What usually happens...

- We have a large amount of data, with lots of duplicate data (e.g. weekly backups).
- We read through the data, and if we see something we’ve seen before, we replace it with an index entry (saving disk space).

- The data is broken up into *chunks* (Rabin Hash).
- The chunks are *fingerprinted* (SHA1): same fingerprint $\Rightarrow$ duplicate chunk.
Deduplication: What usually happens...

Disk bottleneck: Most fingerprints are stored on disk $\implies$ lots of disk reads ("have I seen this before?") $\implies$ slow.
Deduplication: What usually happens...

- **Disk bottleneck**: Most fingerprints are stored on disk $\implies$ lots of disk reads ("have I seen this before?") $\implies$ slow.
- **Caching** and **prefetching** reduce the disk bottleneck problem:
Deduplication: What usually happens...

- **Disk bottleneck**: Most fingerprints are stored on disk → lots of disk reads (“have I seen this before?”) → slow.
- **Caching and prefetching** reduce the disk bottleneck problem:

![Diagram showing caching and prefetching]

The first time we see fingerprints $f_A$, $f_B$, ...
Deduplication: What usually happens...

- **Disk bottleneck**: Most fingerprints are stored on disk \(\implies\) lots of disk reads (“have I seen this before?”) \(\implies\) slow.

- **Caching** and **prefetching** reduce the disk bottleneck problem:

The **first time** we see fingerprints \(f_A, f_B, \ldots\)

The **second time** we see fingerprints \(f_A, f_B, \ldots\)
Lazy deduplication...
Lazy deduplication...

Bloom filter: identifies many uniques (not all). [Commonly used.]
Lazy deduplication...

- **Bloom filter**: identifies many uniques (not all). [Commonly used.]

- **Buffer**: stores fingerprints in hash buckets; searched later on disk (“lazy”)—when full, whole buckets are searched in one go (stored on-disk in hash buckets)
Lazy deduplication...

- **Bloom filter**: identifies many uniques (not all). [Commonly used.]

- **buffer**: stores fingerprints in hash buckets; searched later on disk ("lazy")—when full, whole buckets are searched in one go (stored on-disk in hash buckets)

- **post-lookup**: searching the cache after buffering (maybe multiple times)
Lazy deduplication...

- **Bloom filter**: identifies many uniques (not all). [Commonly used.]

- **buffer**: stores fingerprints in hash buckets; searched later on disk ("lazy")—when full, whole buckets are searched in one go (stored on-disk in hash buckets)

- **post-lookup**: searching the cache after buffering (maybe multiple times)

- **pre-lookup**: searching the cache before buffering [not shown]
Lazy deduplication...

- **Bloom filter**: identifies many uniques (not all). [Commonly used.]
- **buffer**: stores fingerprints in hash buckets; searched later on disk (“lazy”)—when full, whole buckets are searched in one go (stored on-disk in hash buckets)
- **post-lookup**: searching the cache after buffering (maybe multiple times)
- **pre-lookup**: searching the cache before buffering [not shown]
- **prefetching**: bidirectional; triggers post-lookup
Ordinarily, we prefetch the subsequent on-disk fingerprints after a duplicate is found on disk.
Ordinarily, we prefetch the subsequent on-disk fingerprints after a duplicate is found on disk—these will probably be the next incoming fingerprints.
Prefetching...

Ordinarily, we prefetch the subsequent on-disk fingerprints after a duplicate is found on disk—these will probably be the next incoming fingerprints. But this doesn’t work with the lazy method (where fingerprints are buffered).
Ordinarily, we prefetch the subsequent on-disk fingerprints after a duplicate is found on disk—these will probably be the next incoming fingerprints. But this doesn’t work with the lazy method (where fingerprints are buffered).

To overcome this obstacle, each buffered fingerprint is given a...
Ordinarily, we prefetch the subsequent on-disk fingerprints after a duplicate is found on disk—these will probably be the next incoming fingerprints. But this doesn’t work with the lazy method (where fingerprints are buffered).

To overcome this obstacle, each buffered fingerprint is given a... 

*rank*, used to determine the on-disk search range;
Ordinarily, we prefetch the subsequent on-disk fingerprints after a duplicate is found on disk—these will probably be the next incoming fingerprints. But this doesn’t work with the lazy method (where fingerprints are buffered).

To overcome this obstacle, each buffered fingerprint is given a...

- rank, used to determine the on-disk search range; and a
- buffer cycle, indicating where duplicates might be on-disk.
Ordinarily, we prefetch the subsequent on-disk fingerprints after a duplicate is found on disk—these will probably be the next incoming fingerprints. But this doesn’t work with the lazy method (where fingerprints are buffered).

To overcome this obstacle, each buffered fingerprint is given a...

*rank*, used to determine the on-disk search range; and a *buffer cycle*, indicating where duplicates might be on-disk.

It looks like this:
Experimental results...

(See our paper for the details and further experiments.)
(See our paper for the details and further experiments.)

The time it takes to deduplicate a dataset (on SSD):

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Vm (220GB)</th>
<th>Src (343GB)</th>
<th>FSLHomes (3.58TB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eager way</td>
<td>282 sec.</td>
<td>476 sec.</td>
<td>5824 sec.</td>
</tr>
<tr>
<td>lazy way</td>
<td>151 sec.</td>
<td>226 sec.</td>
<td>3939 sec.</td>
</tr>
</tbody>
</table>

(eager = non-lazy [exact] way—i.e., no buffering before accessing the disk)

**Conclusion:** Lazy is faster.
## On-disk lookups...

Disk access time (sec.) on SSD:

<table>
<thead>
<tr>
<th></th>
<th>Vm</th>
<th></th>
<th>Src</th>
<th></th>
<th>FSLHomes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eager</td>
<td>lazy</td>
<td>eager</td>
<td>lazy</td>
<td>eager</td>
<td>lazy</td>
</tr>
<tr>
<td><strong>on-disk lookup</strong></td>
<td>176</td>
<td>20</td>
<td>325</td>
<td>45</td>
<td>4598</td>
<td>1639</td>
</tr>
<tr>
<td>prefetching</td>
<td>46</td>
<td>60</td>
<td>52</td>
<td>68</td>
<td>298</td>
<td>655</td>
</tr>
<tr>
<td>other</td>
<td>59</td>
<td>71</td>
<td>99</td>
<td>113</td>
<td>928</td>
<td>1645</td>
</tr>
<tr>
<td><strong>total disk access</strong></td>
<td>222</td>
<td>80</td>
<td>377</td>
<td>113</td>
<td>4896</td>
<td>2294</td>
</tr>
<tr>
<td><strong>total dedup.</strong></td>
<td>282</td>
<td>151</td>
<td>476</td>
<td>226</td>
<td>5824</td>
<td>3939</td>
</tr>
</tbody>
</table>

*Conclusion:* Lazy reduces the disk bottleneck.
**Conclusion:** Lazy has better throughput on both SSD and HDD, but moreso on slower HDD.
thank you