Table of Contents

[Problem 1](#_Toc523148967)

[Analysis of current Ivy dedupe pattern generation 1](#_Toc523148968)

[Proposed Solution 4](#_Toc523148969)

[ADR pattern generation IO-Sequencer Input Template: 4](#_Toc523148970)

## Problem

To maintain the targeted, dedupe ratio (data reduction savings ratio) on a prefill and subsequent workloads doing random writes and/or sequential writes. To keep it simple, all unique blocks will have the same compression ratio.

Q = number of unique blocks

R = Reference Count/Frequency of occurrence = number of references of 1 or more to a unique block

## Analysis of current Ivy dedupe pattern generation

Balls and buckets paradigm:

* Balls represents unique blocks.
* Buckets for frequency (reference count) of each unique block.

(Case: For dedupe ratio = 2)

Step 1:

Ivy generates unique patterns of reference count = 2.

Step 2:

Ivy does random writes of unique patterns of reference count 2.

* Q is the number of unique blocks.
* Assume two buckets for unique blocks with reference counts 1 and 2.
* Tn(r): Count of unique chunks in bucket with reference count = r for iteration n.

Dedupe Ratio weighted average formula (D):

D =

Reference count state transitions:

Next set of unique blocks (2 in this case) combinations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Case 1 | Case 2 | Case 3 | Case 4 |
| Bucket 1 | 0 | 2 | 1 | 1 |
| Bucket 2 | 2 | 0 | 1 | 1 |

Initial condition:

Tn=0(r=1) = 0

Tn=0(r=2) = Q

Transitions:

Case 1: P (1, t) = 1 🡪 P (1, ∞) = 1/4

Tn + 1 (1) = Tn (1) + 2

Tn + 1 (2) = Tn (2) - 1

Case 2:  P (2, t) = 0 🡪 P (2, ∞) = 1/4

Tn + 1 (1) = Tn (1) – 2

Tn + 1 (2) = Tn (2) + 1

Case 3: P (3, t) = 0 🡪 P (3, ∞) = 1/4

Tn + 1 (1) = Tn (1) (unchanged)

Tn + 1 (2) = Tn (2) (unchanged)

Case 4: P (3, t) = 0 🡪 P (3, ∞) = 1/4

Tn + 1 (1) = Tn (1) (unchanged)

Tn + 1 (2) = Tn (2) (unchanged)

Over time/iterations: p (t, case number) = {1, …1/4)

(Until equilibrium)

**(n < Q)**

**T n + 1 (1) = T n (1) + 2 => Expected value: T n + 1(1) = T (1) + 2 × n = 2n**

**T n + 1 (2) = T n (2) - 1 => Expected value: T n + 1(2) = T (2) – n = Q - n**

**2n = Q – n (at equilibrium)**

**n = Q / 3**

T (1) = 2/3 Q

T (2) = 2/3 Q

T (1) + T (2) = 4/3 Q (total unique blocks at equilibrium => new unique blocks created = 4/3 Q – Q = Q/3)

**D =** (1 × T (1) + 2 × T (2))/ (T (1) + T (2)) **= 1.5**

Asymptotically and after number of iterations -> ∞

P (1) -> ½

P (2) -> ½

(Two buckets form 🡪 for starting ref count R, R buckets from ref count 1 – R are created and at asymptotic equilibrium all the buckets will have the same number of unique blocks)

In this pattern generation, for a targeted dedupe ratio R, after iterations of overwrites, R buckets get formed and there is a migration of old chunks from Right -> Left.

2

R

1

Extending to R > 2 by recursion – [1,2, …, R-1] [R]

**1, 2, …, R reference counts:**

D =

Asymptotic D = (1 + 2 + … + R)/R = R (R + 1)/2R = **(R + 1)/2**

## Proposed Solution

1. In a realistic customer workload – it is unlikely all unique blocks have the same reference count. There would be likely unique blocks with varying number of reference counts with many with a reference count 1. So, it makes sense to have a distribution of reference counts.

Ivy already does this for fractional dedupe ratios, for example a dedupe ratio of 2.8 a mix of unique blocks with two copies and unique blocks with three copies.

1. Each unique block has life cycle. How long does a unique block stay in the system?
2. For overwrites use a mix of unique patterns and repeating patterns, so that new buckets are formed on with higher ref counts as well, say R + 1, R + 2, …., R + n. Set limits for maximum number ref count. So that a distribution around the desired R is maintained, Unique patterns need to be also introduced to keep the number of unique blocks (Q ± ∆Q).
3. Rate of increased unique chunk increase needs to be offset by decrease in unique chunks.

Solution:

1. For prefill create a distribution of ref counts with target Dedupe ratio
2. For overwrites, use a mix of repeating and unique patterns
3. For sequential writes and prefill generate unique/random data blocks following a ref count distribution.
4. For random writes cycle through fixed set of patterns from a pattern generator.
5. Constraints are to maintain the expected ref count distribution, such as limiting the maximum ref counts to targeted, max, by switching to/ reinitializing with a new set of repeating patterns.

Example:

* LUN size - 96 GB
* Block size/dedupe unit size – 8KB
* Number of blocks in LUN (B) = 96 GB/ 8KB = 12 M
* Number of unique chunks (Q) = (B) / (dedupe ratio) = 6 M
* Size of repeating pattern set = 0.1 \* Q = 600 K

## ADR pattern generation IO-Sequencer Input Template:

Workload parameters are used in setting up the IO-Sequencer input used by the IO generation.

Example: [CreateWorkload] "owl" [select] "port" is "1A" [**iogenerator**] "random\_steady" [**parameters**] %% "IOPS=max,fraction\_read="50%", blocksize = "4KiB" **dedupe** = 1.5 **%%;**

**Extending control knobs via additional parameters**:

* + Dedupe ratio (R)
  + Compressibility factor
  + Pattern generation - Ivy generates unique patterns with a given dedupe ratio using a serpentine random sequence with random pattern seed across Ivy test steps.
    - Seed - Use “fixed starting pattern seed” across steps to generate repeating patterns
  + Sequential vs. Random
    - Sequential write (use the pattern generation sequence P1, …, Pn (Pattern Space 1 (PS1))
    - Random write (repeating patterns from the pattern sequence + unique patterns from pattern space Pn+1, …, P2n (Pattern Space 2 (PS2))
  + Spread - Ref count distribution spread (Example: spread = 2; R-2, R-1, R, R+1, R+2) (Default: Spread = 0)
  + Pattern generation control – probability (p) of choosing from unique patterns sequence (Default: PGC = 1)
  + Pattern space size (Example: 12mB) (Default: PSS = 0) -> PS1, PS2

IO-Sequencer Input for ADR

Predefined patterns (TBD)

Unique patterns

Repeating patterns

Decide: repeat or new unique

Dedupe ratio,  
Ref count spread, pattern seed

IO