MELBOURNE SHUFFLE ON OBLIVIOUS STORAGE

Presented by: Yi Qian

ORAM

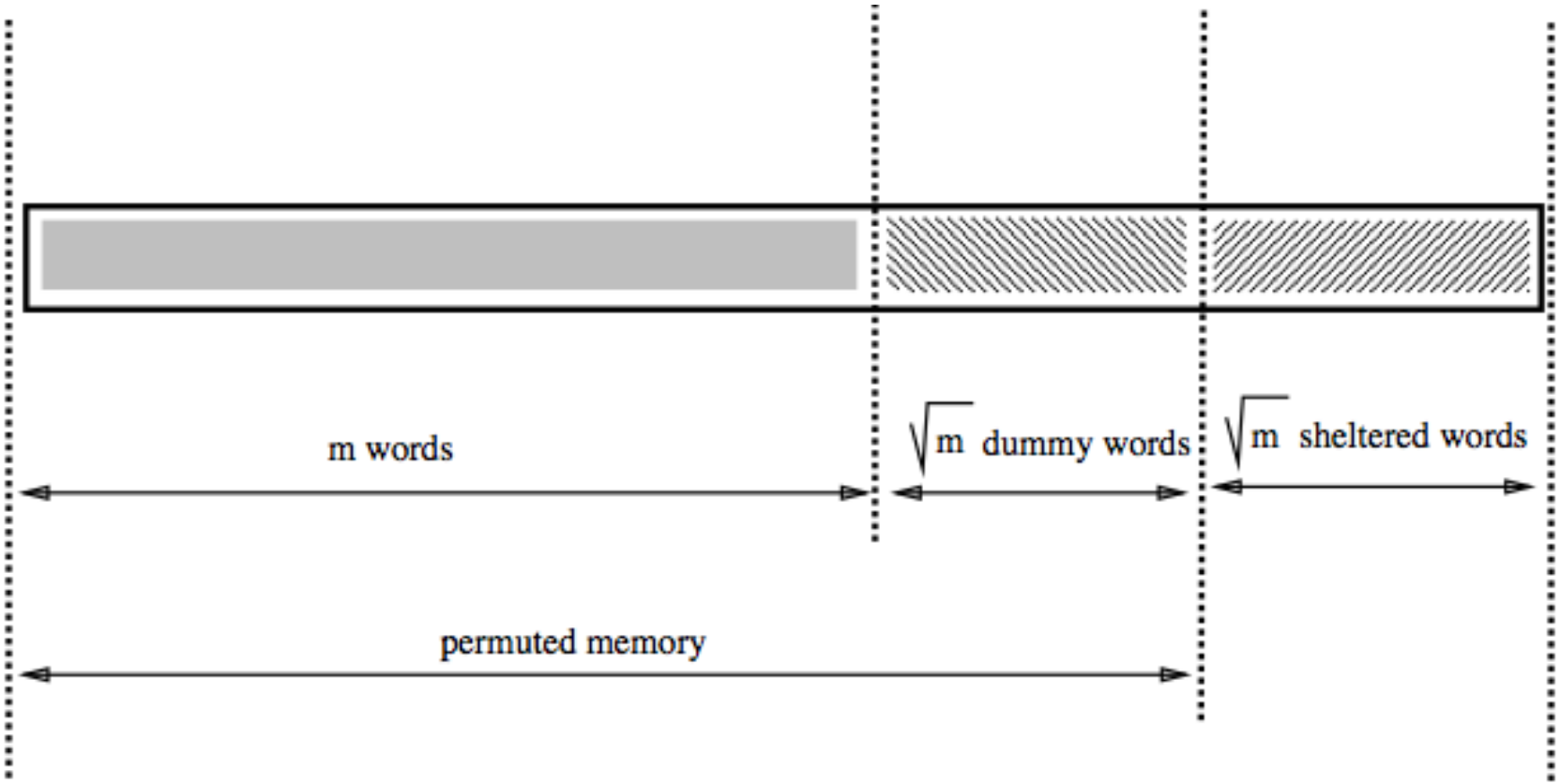
- A client wishes to store data at a remote untrusted
- Data access pattern can reveal very sensitive information to the untrusted server

- The goal of ORAM is to completely hide the data access patter :
- 1) which data is being accessed; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); 4) access pattern (sequential, random, etc); or 5) whether the access is a read or a write.

Naïve & Square Root Solutions

- Naive: For each access (write/read), download all data
- **Square Root: Covered by Prof already**
- Access overhead: O(√n (log n)^2)
- Client Memory (Message size): O(1)
- Server Storage : O(n + √n)

Square Root Solutions



Square Root Solutions

- 1. A permutation P over the integers [1, n + $\sqrt{n}$] is uniformly selected and the content of the permuted memory is obliviously relocated.
- 2. For each access with (location-index i):
- 2a. Scan through the entire shelter in a sequential order to find an item whose location-index is i. If the item is not found in the shelter, access the actual location P(i) in the permuted area. If found, access the next dummy item.
- 2b. Scan again through the entire shelter in a sequential order, and write back the updated (re-encrypted) item of location-index i, in the next available location.

Square Root Solutions

- Step 2 takes O(√n). The **bottleneck** is **step 1**.

- Step 1: oblivious shuffle.
  - Need to shuffle the entire data every √n accesses.

- Assign a random integer to each data. Then do Oblivious sorting.
- Oblivious sorting: sort an array non-adaptively
- Batcher's Sorting Network: O(n (log n)^2)

Batcher's Sorting Network

- Normal sorting alg: O(n log n). Non-oblivious.
  - Do compare-and-switches between pairs of keys. Which pairs depends on the input array.
  - Batcher's Sorting: fixed indexes of compare-and-switches.
- Sort(x1,…,xn):
  - Sort(x1, . . . , xn/2), then Sort(xn/2+1, . . . , xn), and then Merge(x1, . . . , xn).
- Merge(x1, . . . , xn):
  - Merge(xi, for i odd), then Merge(xi for i even), and then Comp(x2, x3), Comp(x4, x5), · · ·Comp(xn−2, xn−1).

Comp(xi, xj) : compare and switch with key i and key

Batcher's Sorting Network Example

- Given 2 7 6 3 9 4 1 8
- If we sort the first and second halves separately we obtain: 2 3 6 7 1 4 8 9
- Sorting the odd-indexed keys (2, 6, 1, 8) and then the even-indexed keys (3, 7, 4, 9) while leaving them in odd and even places respectively yields: 1 3 2 4 6 7 8 9
- This list is now almost sorted: doing a comparison switch between the keys in positions (2 and 3),
- (4 and 5) and (6 and 7) will in fact finish the sort.

Batcher's Sorting Network Complexity

- **S**(**n**): comparisons of sort(x1,…,xn)
- **M**(**n**) comparisons of merge(x1,…,xn)
- **S(n) = 2S(n/2) +M(n),**
- **M(n) = 2M(n/2) + n/2 − 1.**
- **S(2)=M(2)=1**
- Inductively, **M(n) <=** n/2 **log n and S(n) <=** n/2 (**log n)^2**

- **Squre root solution shuffle every** √n accesses. Hence, amortized overhead access is (√n(**log n)^2**)

Melbourne Shuffle

Table 1: Comparison of data-oblivious sorting and shuffle algorithms over n items.

	Randomized	Private Memory	Message Size	External Memory	I/Os
Batcher's network [3]		$O(1)$	$O(1)$	$O(n)$	$O(n(\log n)^2)$
Batcher's network I		$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n}(\log n)^2)$
Batcher's network II [9]		$O(\sqrt{n})$	$O(\sqrt[8]{n})$	$O(n)$	$O(n^{7/8})$
AKS [1], Zig-zag sort [8]		$O(1)$	$O(1)$	$O(n)$	$O(n \log n)$
Randomized shellsort [7]	✓	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n} \log n)$
Melbourne shuffle	✓	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n})$
Melbourne shuffle ($c \geq 3$)	✓	$O(\sqrt[c]{n})$	$O(\sqrt[c]{n})$	$O(n)$	$O(c\sqrt[c]{n^{c-1}})$

Melbourne Shuffle

- Takes advantage of allowing reasonable client memory and message size.
- The access overhead is measured w.r.t. the amortized number of I/O executions.

- Supported operations:
- Read/write an element at location loc
- Read/write a list of elements at locations loc, …, loc+L-1
- Read/write a dist of elements at locations specified by:
- Loc_i + L_i-1 for all i.

Melbourne Shuffle Problem

- $(Enc(\pi(A)), \alpha) \leftarrow Shuffle(s, S, A, \pi)$
- Input: s is the secret key, S is the server storage, A is the input array of length n and π is a pseudorandom permutation.

- Return (1) the encryption of the permutation of A according to π; (2) a transcript α of the operations that transform Enc(A) to Enc(π(A)) using space S.

- Every operation in α must be oblivious

Melbourne Shuffle Basic Algorithm

- Client memory / message size: $O(\sqrt{n} \log n)$
- Server storage: $O(n \log n)$
- Overhead access: $O(\sqrt{n})$

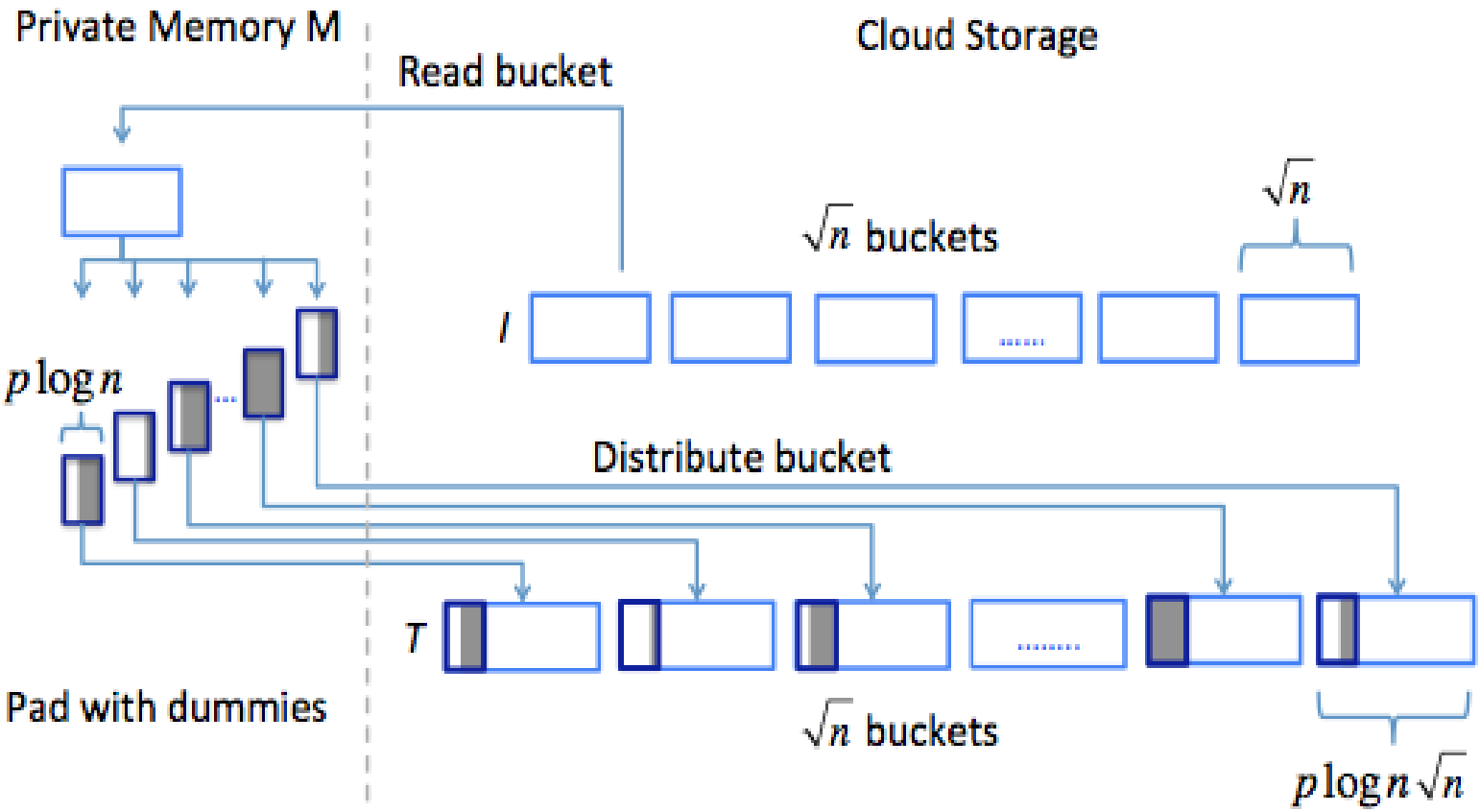
- **Intuition:**
- 1. Split input array A into $\sqrt{n}$ buckets (each of $\sqrt{n}$ elements)
- 2. For every bucket do some kind of oblivious shuffle(according to π).
- 3. Clean up and return the $Enc(\pi(A))$.

Melbourne Shuffle Basic Algorithm

- Storage S split into 3 parts: I, T and O.
- I: for input A
- T: An encrypted temporary array for shuffle
- O: output array

- Algorithm includes two phases:
- 1. Distribution phase: Put elements from every bucket of I to every bucket of T according to the permutation.
- 2. Clean up and return the final output array

Distribution Phase



1: max_elems $\leftarrow p \log n$
2: num_buckets $\leftarrow \sqrt{n}$
3: {*Distribution phase: distribute elements of I into T*}
4: **for** $\mathsf{id}_I \in \{0, \ldots, \mathsf{num\_buckets} - 1\}$ **do** {read buckets of I}
5: $\quad \mathsf{bucket}_M \leftarrow \mathsf{getRange}(I, \mathsf{id}_I \times \sqrt{n}, \sqrt{n})$
6: $\quad \mathsf{rev\_bucket}_M \leftarrow \mathsf{empty\_map}()$ {Reverse map of bucket ids in T to elements}
7: $\quad$ **for** $e \in \mathsf{bucket}_M$ **do** {Assign elements their bucket ids in T}
8: $\quad\quad (x, v) \leftarrow \mathsf{Dec}(e)$
9: $\quad\quad \mathsf{id}_T \leftarrow \lfloor \rho(x)/\sqrt{n} \rfloor$ {Bucket id of element (x, v) in T according to its location in O}
10: $\quad\quad \mathsf{rev\_bucket}_M[\mathsf{id}_T].\mathsf{add}(\mathsf{Enc}(x, v))$ {Collect elements of same bucket}
11: $\quad$ **end for**
12: $\quad$ {Can be done via a single putRangeDist for $\sqrt{n}$ batches of size max_elems}
13: $\quad$ **for** $\mathsf{id}_T \in \{0, \ldots, \mathsf{num\_buckets} - 1\}$ **do** {Distribute bucket_M in buckets of T}
14: $\quad\quad$ **if** $\mathsf{size}(\mathsf{rev\_bucket}_M[\mathsf{id}_T]) > \mathsf{max\_elems}$ **then**
15: $\quad\quad\quad$ **fail** {ρ moves more than $p \log n$ elements from a bucket of I to a bucket of T}
16: $\quad\quad$ **end if**
17: $\quad\quad$ {Hide how many real elements go to T's buckets by padding with encrypted dummies}
18: $\quad\quad \mathsf{rev\_bucket}_M[\mathsf{id}_T] \leftarrow \mathsf{dummy\_pad}(\mathsf{rev\_bucket}_M[\mathsf{id}_T], \mathsf{max\_elems})$
19: $\quad\quad$ {Write a batch of max_elems from every bucket of I to every bucket of T}
20: $\quad\quad \mathsf{putRange}(T, \mathsf{id}_T \times \sqrt{n} \times \mathsf{max\_elems} + \mathsf{max\_elems} \times \mathsf{id}_I, \mathsf{rev\_bucket}_M[\mathsf{id}_T])$
21: $\quad$ **end for**
22: **end for**

Clean up Phase

23: {*Clean-up phase: clean T and write the result to O*}
24: **for** $\mathsf{id}_T \in \{0, \ldots, \mathsf{num\_buckets} - 1\}$ **do** {read buckets of T}
25: $\quad \mathsf{bucket}_M \leftarrow \mathsf{getRange}(T, \mathsf{id}_T \times \sqrt{n} \times \mathsf{max\_elems}, \sqrt{n} \times \mathsf{max\_elems})$
26: $\quad$ {Decrypt the bucket, remove dummy, sort real elements using ρ and re-encrypt}
27: $\quad \mathsf{bucket}_M \leftarrow \mathsf{clean}(\mathsf{bucket}_M)$
28: $\quad$ {The distribution phase guarantees that bucket_M contains exactly $\sqrt{n}$ elements}
29: $\quad \mathsf{putRange}(O, \mathsf{id}_T \times \sqrt{n}, \mathsf{bucket}_M)$
30: **end for**

Security & Oblivious

- The algorithm fails when there is a bucket in T contains
- > p log n elements from a bucket in I.

- Throw $\sqrt{n}$ balls into $\sqrt{n}$ bins. Pick p >=e is sufficient to guarantee failure occurs with neg-probability

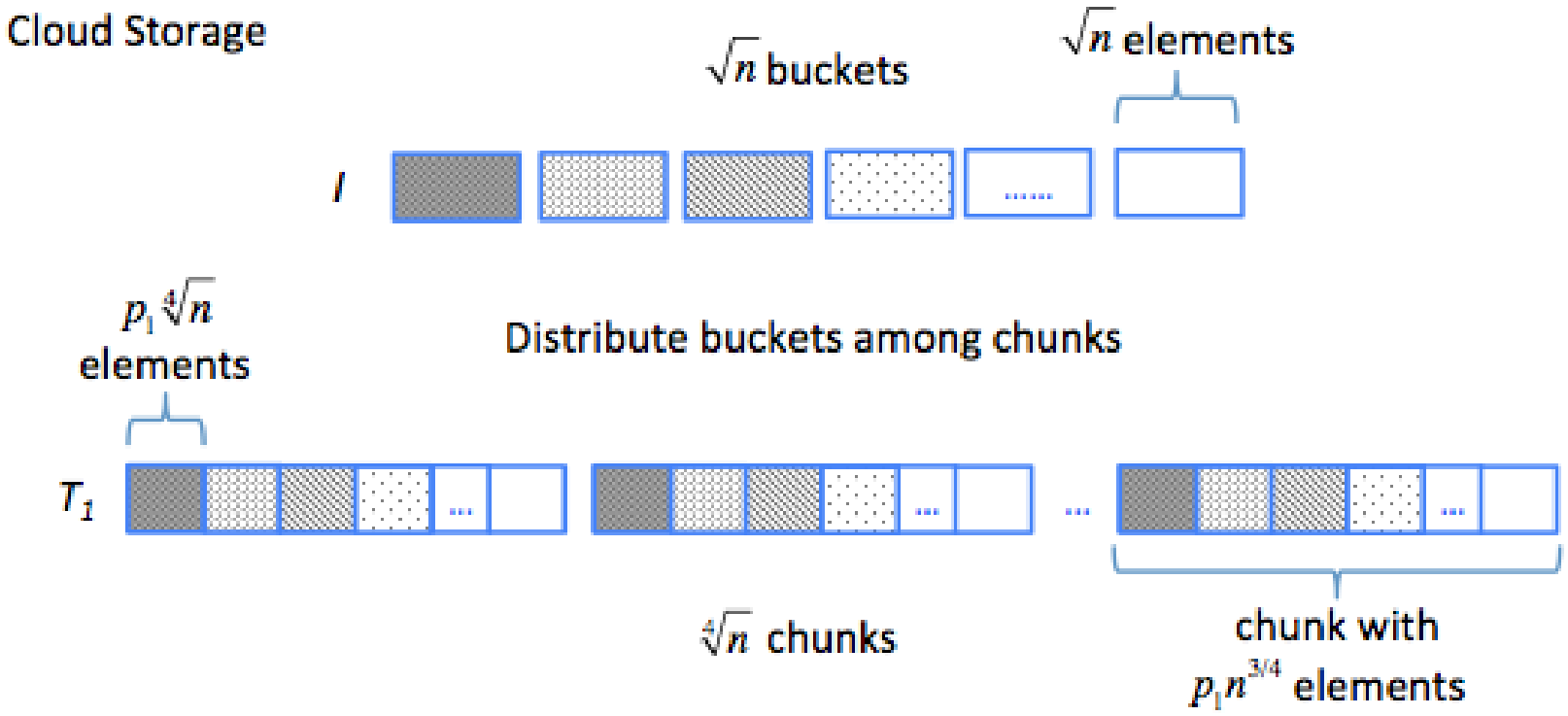
- Roughly: Every operation in the algorithm is oblivious.
- 1. Read data from every bucket of I
- 2. Write to T with encrypted buckets of length $\sqrt{n}$ p log n
- 3. Clean up phase is oblivious

Optimized Melbourne Shuffle

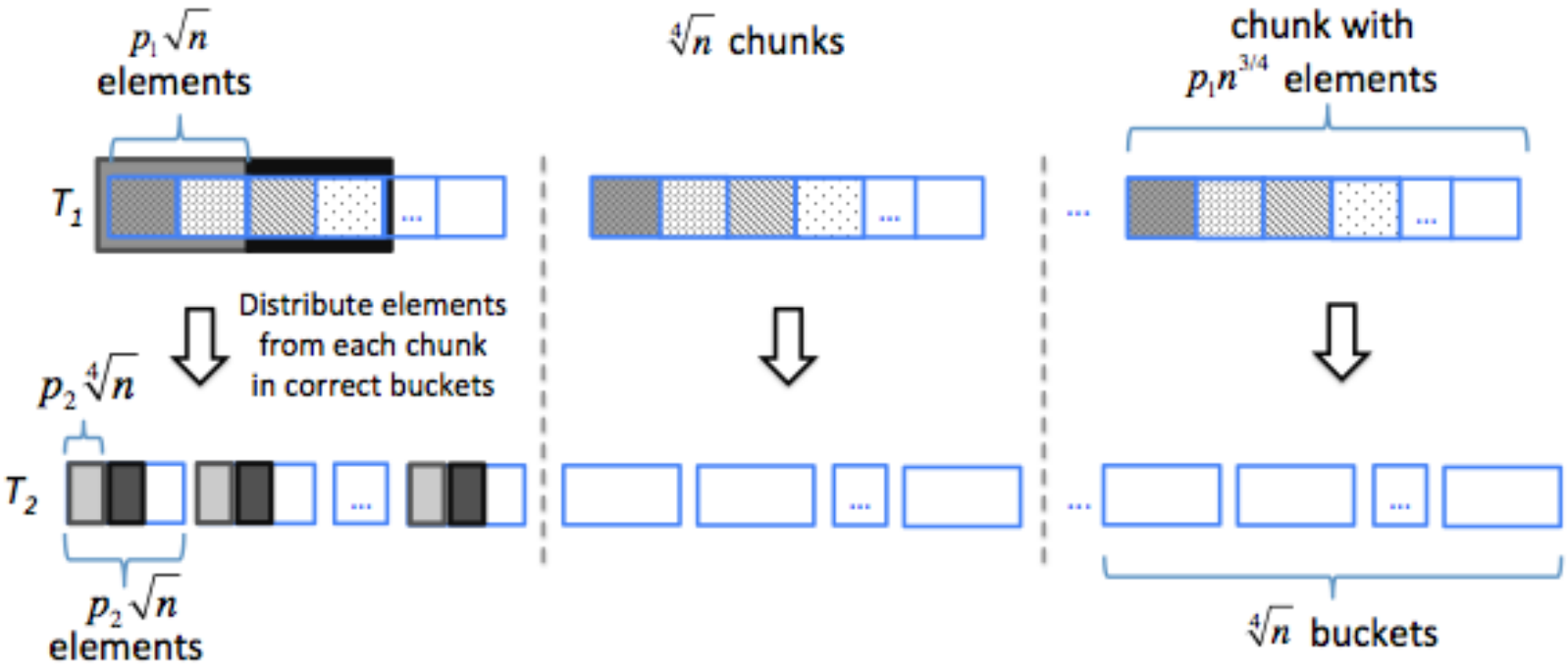
- Idea: split the distribution phase into two phases.
- 1. From I to T1. Again, split I into $\sqrt{n}$ buckets. Distribute (according to the permutation) the elements in every buckets into chucks of length p1 n^{3/4} in T1 (there are n^{1/4} such chucks and T1 is of length p1n).

- 2. From T1 to T2. Distribute (according to the permutation) the elements in every chucks to the buckets of length p2$\sqrt{n}$ in T2 (there are $\sqrt{n}$ such buckets and T2 is of length p2n)

I to T1


Cloud Storage
$\sqrt{n}$ buckets
$\sqrt{n}$ elements
I
$p_1\sqrt[4]{n}$ elements
Distribute buckets among chunks
T_1
$\sqrt[4]{n}$ chunks
chunk with $p_1 n^{3/4}$ elements

T1 to T2


$p_1\sqrt{n}$ elements
$\sqrt[4]{n}$ chunks
chunk with $p_1 n^{3/4}$ elements
T_1
Distribute elements from each chunk in correct buckets
$p_2\sqrt[4]{n}$
T_2
$p_2\sqrt{n}$ elements
$\sqrt[4]{n}$ buckets

Analysis

- O(√n) I/O access
- client memory / message size: <= p2√n
- Server Storage: (p1+p2)n

- Security & Oblivious: by similar arguments.