Dynamic Indexability
and
Lower Bounds for Dynamic One-Dimensional Range Query Indexes

Ke Yi

HKUST

First Annual SIGMOD Programming Contest (to be held at SIGMOD 2009)

- "Student teams from degree granting institutions are invited to compete in a programming contest to develop an indexing system for main memory data."

  "The index must be capable of supporting range queries and exact match queries as well as updates, inserts, and deletes."

  "The choice of data structures (e.g., B-tree, AVL-tree, etc.) ... is up to you."

First Annual SIGMOD Programming Contest (to be held at SIGMOD 2009)

- "Student teams from degree granting institutions are invited to compete in a programming contest to develop an indexing system for main memory data."

  "The index must be capable of supporting range queries and exact match queries as well as updates, inserts, and deletes."

  "The choice of data structures (e.g., B-tree, AVL-tree, etc.) ... is up to you."

- We think these problems are so basic that every DB grad student should know, but do we really have the answer?

Answer: Hash Table and B-tree!

- Indeed, (external) hash tables and B-trees are both fundamental index structures that are used in all database systems

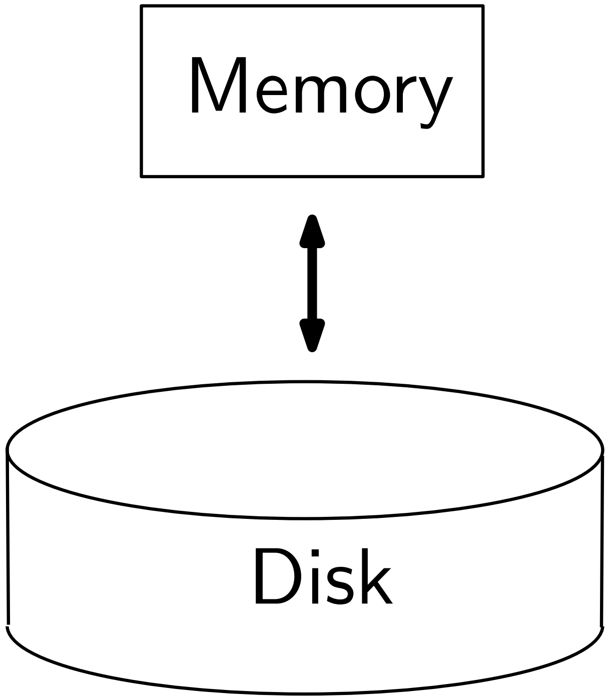
Answer: Hash Table and B-tree!

- Indeed, (external) hash tables and B-trees are both fundamental index structures that are used in all database systems
- Even for main memory data, we should still use external versions that optimize cache misses

Answer: Hash Table and B-tree!

- Indeed, (external) hash tables and B-trees are both fundamental index structures that are used in all database systems
- Even for main memory data, we should still use external versions that optimize cache misses

External memory model (I/O model):

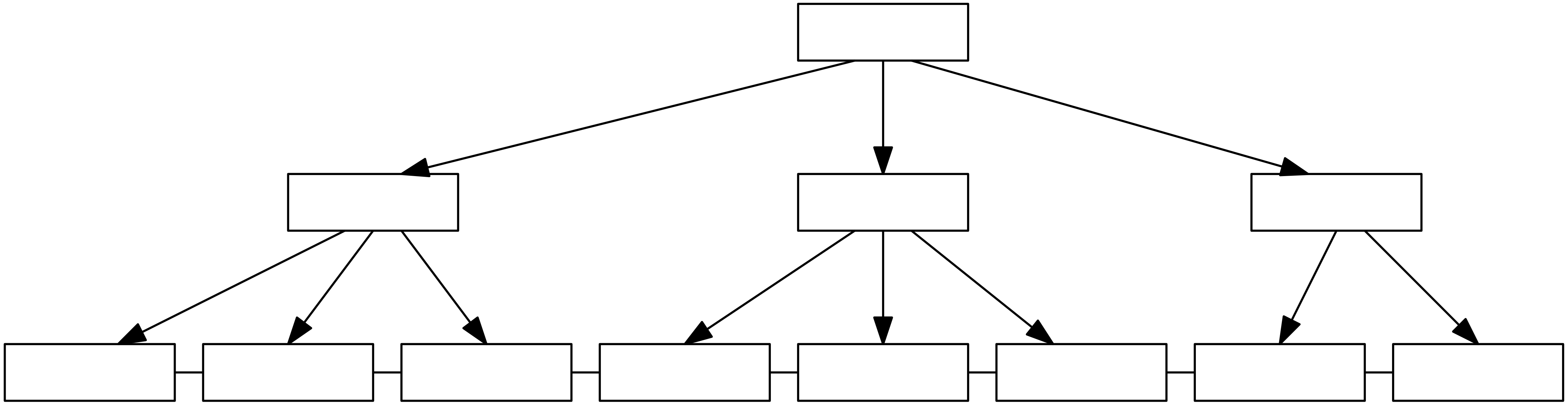


Memory of size m

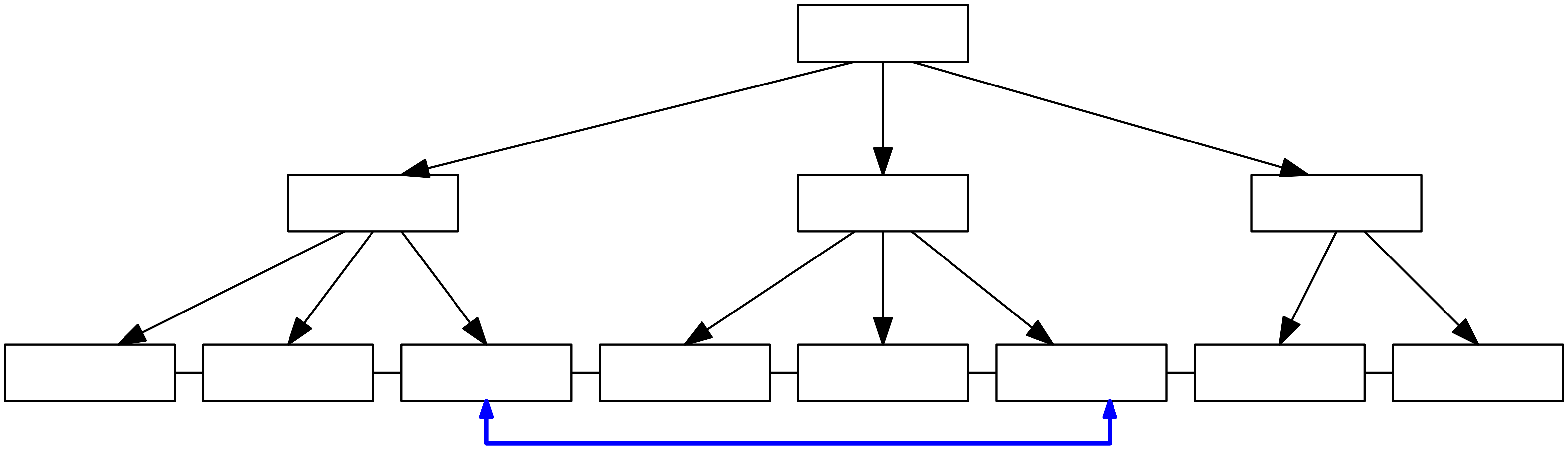
Each I/O reads/writes a block

Disk partitioned into blocks of size b

The B-tree



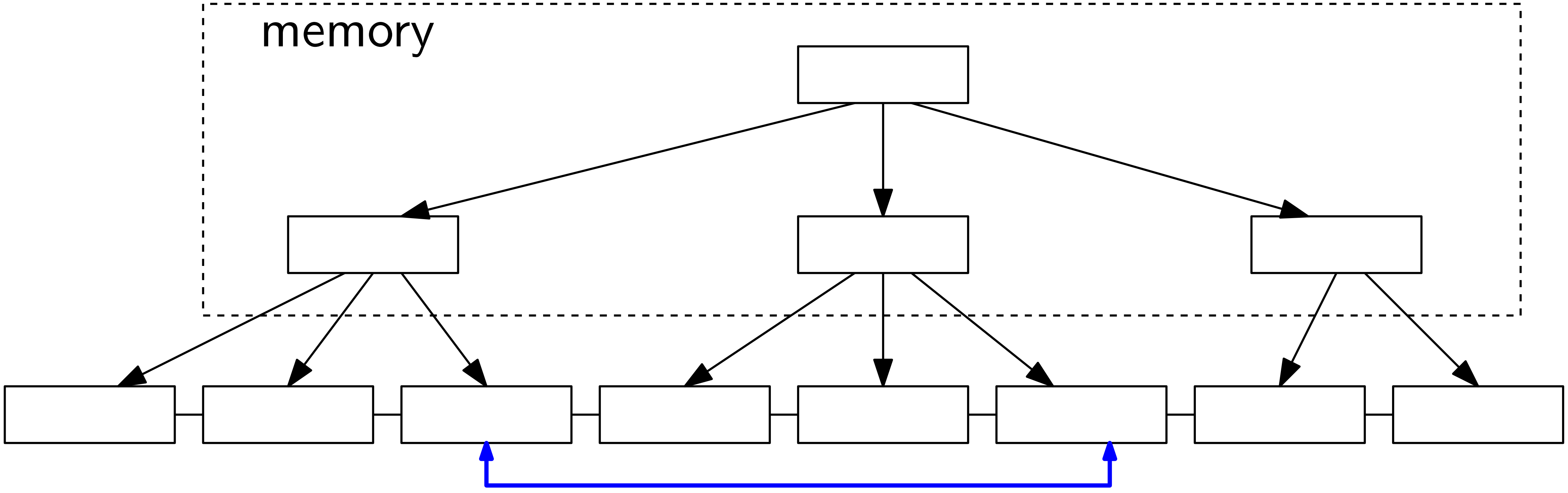
The B-tree



A range query in $O(\log_b n + k/b)$ I/Os

k: output size

The B-tree

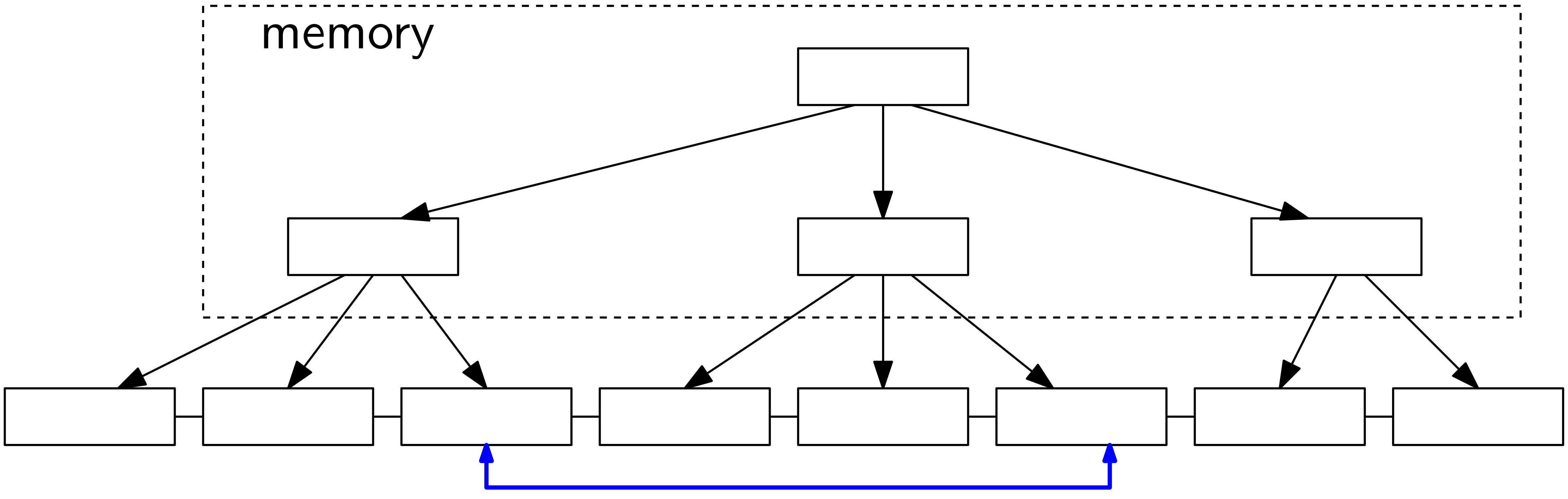


A range query in $O(\log_b n + k/b)$ I/Os

k: output size

$$\log_b n - \log_b m = \log_b \frac{n}{m}$$

The B-tree



A range query in $O(\sout{\log_b n} + k/b)$ I/Os

k: output size

$$\log_b n - \log_b m = \log_b \frac{n}{m}$$

The height of B-tree never goes beyond 5 (e.g., if $b = 100$, then a B-tree with 5 levels stores n = 10 billion records). We will assume $\log_b \frac{n}{m} = O(1)$.

Now Let's Go Dynamic

- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block
  - B-tree: Split blocks when necessary
  - Hashing: Rebuild the hash table when too full; *extensible hashing* [Fagin, Nievergelt, Pippenger, Strong, 79]; *linear hashing* [Litwin, 80]

Now Let's Go Dynamic

- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block
  - B-tree: Split blocks when necessary
  - Hashing: Rebuild the hash table when too full; *extensible hashing* [Fagin, Nievergelt, Pippenger, Strong, 79]; *linear hashing* [Litwin, 80]
  - These resizing operations only add $O(1/b)$ I/Os amortized per insertion; bottleneck is the first search + insert

Now Let's Go Dynamic

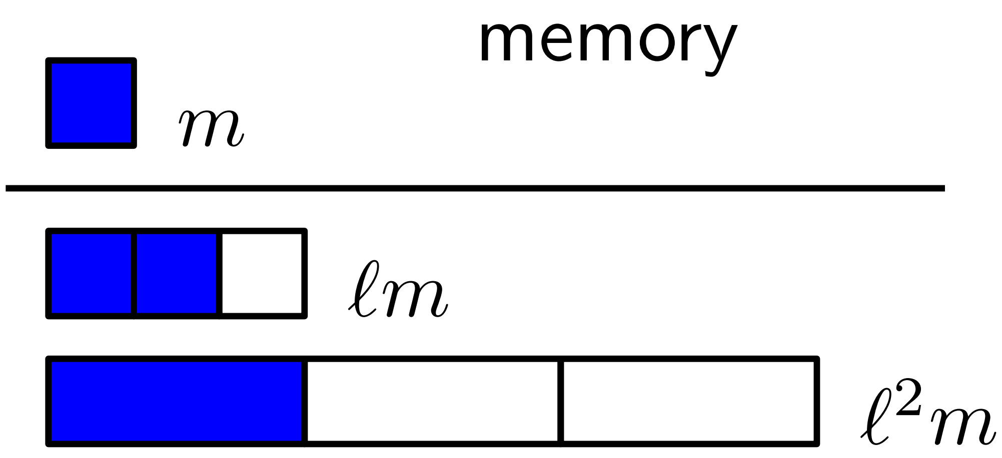
- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block
  - B-tree: Split blocks when necessary
  - Hashing: Rebuild the hash table when too full; *extensible hashing* [Fagin, Nievergelt, Pippenger, Strong, 79]; *linear hashing* [Litwin, 80]
  - These resizing operations only add $O(1/b)$ I/Os amortized per insertion; bottleneck is the first search + insert
- Cannot hope for lower than 1 I/O per insertion only if the changes must be committed to disk right away (necessary?)

Now Let's Go Dynamic

- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block
  - B-tree: Split blocks when necessary
  - Hashing: Rebuild the hash table when too full; *extensible hashing* [Fagin, Nievergelt, Pippenger, Strong, 79]; *linear hashing* [Litwin, 80]
  - These resizing operations only add $O(1/b)$ I/Os amortized per insertion; bottleneck is the first search + insert
- Cannot hope for lower than 1 I/O per insertion only if the changes must be committed to disk right away (necessary?)
  - Otherwise we probably can lower the amortized insertion cost by buffering, like numerous problems in external memory, e.g. stack, priority queue,... All of them support an insertion in $O(1/b)$ I/Os — the best possible

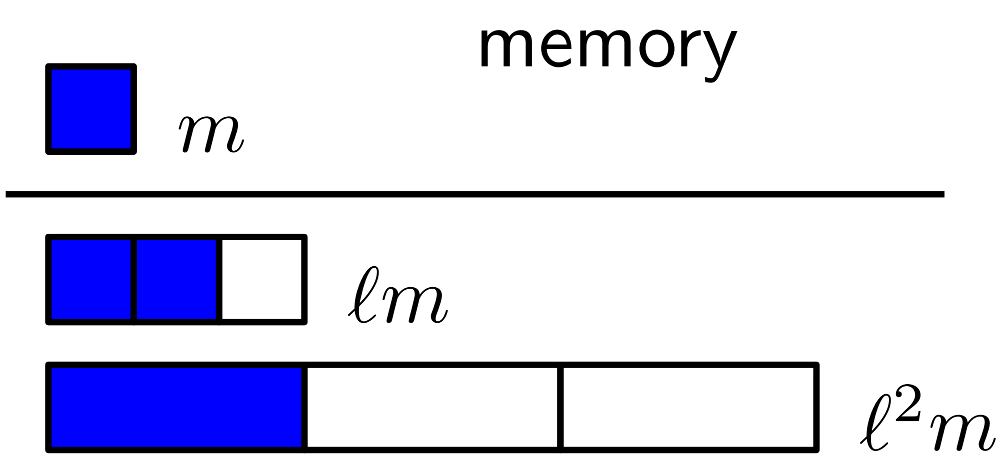
Dynamic B-trees for Fast Insertions

- *LSM-tree* [O'Neil, Cheng, Gawlick, O'Neil, 96]: Logarithmic method + B-tree



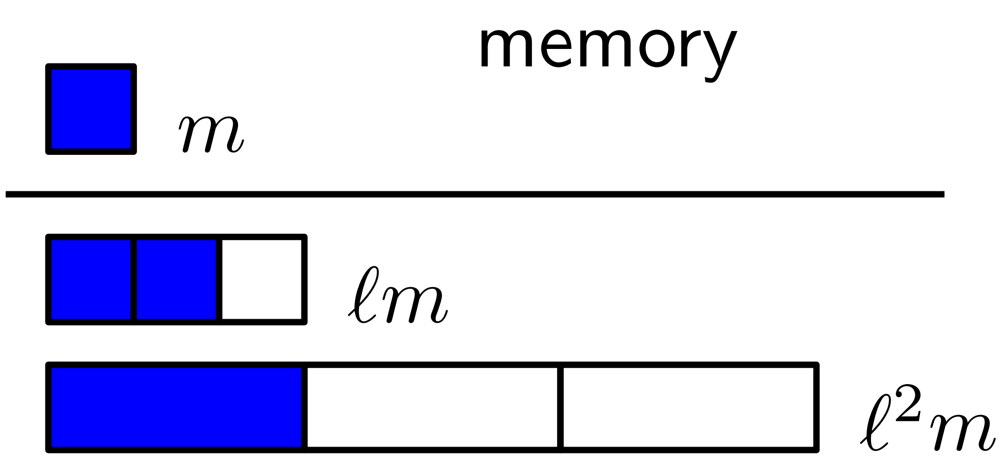
Dynamic B-trees for Fast Insertions

- *LSM-tree* [O'Neil, Cheng, Gawlick, O'Neil, 96]: Logarithmic method + B-tree
  - Insertion: $O(\frac{\ell}{b} \log_\ell \frac{n}{m})$
  - Query: $O(\log_\ell \frac{n}{m} + \frac{k}{b})$



Dynamic B-trees for Fast Insertions

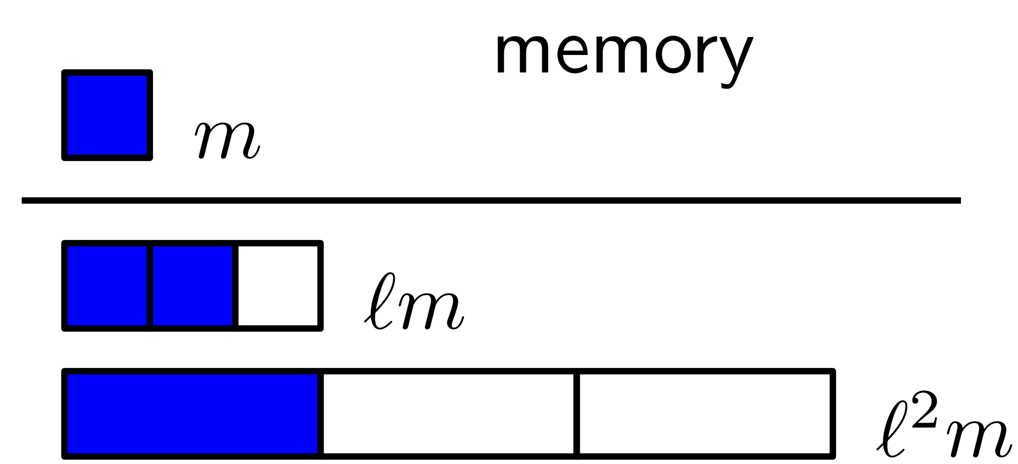
- *LSM-tree* [O'Neil, Cheng, Gawlick, O'Neil, 96]: Logarithmic method + B-tree
  - Insertion: $O(\frac{\ell}{b} \log_\ell \frac{n}{m})$
  - Query: $O(\log_\ell \frac{n}{m} + \frac{k}{b})$



- *Stepped merge tree* [Jagadish, Narayan, Seshadri, Sudarshan, Kannegantil, 97]: variant of LSM-tree
  - Insertion: $O(\frac{1}{b} \log_\ell \frac{n}{m})$
  - Query: $O(\ell \log_\ell \frac{n}{m} + \frac{k}{b})$

Dynamic B-trees for Fast Insertions

- *LSM-tree* [O'Neil, Cheng, Gawlick, O'Neil, 96]: Logarithmic method + B-tree
  - Insertion: $O(\frac{\ell}{b}\log_\ell \frac{n}{m})$
  - Query: $O(\log_\ell \frac{n}{m} + \frac{k}{b})$



- *Stepped merge tree* [Jagadish, Narayan, Seshadri, Sudarshan, Kannegantil, 97]: variant of LSM-tree
  - Insertion: $O(\frac{1}{b}\log_\ell \frac{n}{m})$
  - Query: $O(\ell\log_\ell \frac{n}{m} + \frac{k}{b})$
- Usually ℓ is set to be a constant, then they both have $O(\frac{1}{b}\log \frac{n}{m})$ insertion and $O(\log \frac{n}{m} + \frac{k}{b})$ query

More Dynamic B-trees

- *Buffer tree* [Arge, 95]
- *Yet another B-tree* (Y-tree) [Jermaine, Datta, Omiecinski, 99]

More Dynamic B-trees

- *Buffer tree* [Arge, 95]
- *Yet another B-tree* (Y-tree) [Jermaine, Datta, Omiecinski, 99]
  - Insertion: $O(\frac{1}{b}\log\frac{n}{m})$, pretty fast since $b \gg \log\frac{n}{m}$ typically, but not that fast; if $O(\frac{1}{b})$ insertion required, query becomes $O(b^\epsilon + \frac{k}{b})$
  - Query: $O(\log\frac{n}{m} + \frac{k}{b})$, much worse than the static B-tree's $O(1+\frac{k}{b})$; if $O(1+\frac{k}{b})$ query required, insertion cost becomes $O(\frac{b^\epsilon}{b})$

More Dynamic B-trees

- *Buffer tree* [Arge, 95]
- *Yet another B-tree* (Y-tree) [Jermaine, Datta, Omiecinski, 99]
  - Insertion: $O(\frac{1}{b}\log\frac{n}{m})$, pretty fast since $b \gg \log\frac{n}{m}$ typically, but not that fast; if $O(\frac{1}{b})$ insertion required, query becomes $O(b^\epsilon + \frac{k}{b})$
  - Query: $O(\log\frac{n}{m} + \frac{k}{b})$, much worse than the static B-tree's $O(1 + \frac{k}{b})$; if $O(1 + \frac{k}{b})$ query required, insertion cost becomes $O(\frac{b^\epsilon}{b})$
  - Deletions? Standard trick: inserting "delete signals"

More Dynamic B-trees

- *Buffer tree* [Arge, 95]
- *Yet another B-tree* (Y-tree) [Jermaine, Datta, Omiecinski, 99]
  - Insertion: $O(\frac{1}{b}\log\frac{n}{m})$, pretty fast since $b \gg \log\frac{n}{m}$ typically, but not that fast; if $O(\frac{1}{b})$ insertion required, query becomes $O(b^{\epsilon} + \frac{k}{b})$
  - Query: $O(\log\frac{n}{m} + \frac{k}{b})$, much worse than the static B-tree's $O(1 + \frac{k}{b})$; if $O(1 + \frac{k}{b})$ query required, insertion cost becomes $O(\frac{b^{\epsilon}}{b})$
  - Deletions? Standard trick: inserting "delete signals"
- No further development in the last 10 years. So, seems we can't do better, can we?

Main Result

For any dynamic range query index with a query cost of $q+O(k/b)$ and an amortized insertion cost of u/b, the following tradeoff holds

$$\begin{cases} q \cdot \log(u/q) = \Omega(\log b), & \text{for } q < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log q = \Omega(\log b), & \text{for all } q. \end{cases}$$

Main Result

For any dynamic range query index with a query cost of $q+O(k/b)$ and an amortized insertion cost of u/b, the following tradeoff holds

$$\begin{cases} q \cdot \log(u/q) = \Omega(\log b), & \text{for } q < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log q = \Omega(\log b), & \text{for all } q. \end{cases}$$

Current upper bounds:

q	u
$\log \frac{n}{m}$	$\log \frac{n}{m}$
1	$(\frac{n}{m})^{\epsilon}$
$(\frac{n}{m})^{\epsilon}$	1

Main Result

For any dynamic range query index with a query cost of $q+O(k/b)$ and an amortized insertion cost of u/b, the following tradeoff holds

$$\begin{cases} q \cdot \log(u/q) = \Omega(\log b), & \text{for } q < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log q = \Omega(\log b), & \text{for all } q. \end{cases}$$

Current upper bounds:

q	u
$\log \frac{n}{m}$	$\log \frac{n}{m}$
1	$(\frac{n}{m})^\epsilon$
$(\frac{n}{m})^\epsilon$	1

Assuming $\log_b \frac{n}{m} = O(1)$, all the bounds are tight!

Main Result

For any dynamic range query index with a query cost of $q+O(k/b)$ and an amortized insertion cost of u/b, the following tradeoff holds

$$\begin{cases} q \cdot \log(u/q) = \Omega(\log b), & \text{for } q < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log q = \Omega(\log b), & \text{for all } q. \end{cases}$$

Current upper bounds:

q	u
$\log \frac{n}{m}$	$\log \frac{n}{m}$
1	$(\frac{n}{m})^\epsilon$
$(\frac{n}{m})^\epsilon$	1

Assuming $\log_b \frac{n}{m} = O(1)$, all the bounds are tight!

The technique of [Brodal, Fagerberg, 03] for the predecessor problem can be used to derive a tradeoff of

$$q \cdot \log(u \log^2 \frac{n}{m}) = \Omega(\log \frac{n}{m}).$$

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, 97]

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, 97]

4 7 9	1 2 4	3 5 8	2 6 7	1 8 9	4 5

- Objects are stored in disk blocks of size up to b, possibly with redundancy.
  Redundancy $r = (\text{total \# blocks})/\lceil n/b \rceil$

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, 97]

  a query reports $\{2,3,4,5\}$

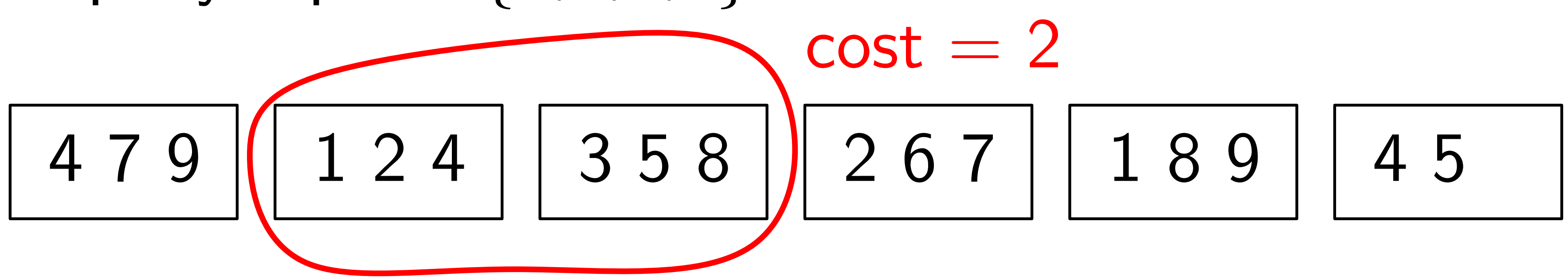
4 7 9	1 2 4	3 5 8	2 6 7	1 8 9	4 5

- Objects are stored in disk blocks of size up to b, possibly with redundancy.
  Redundancy $r = $ (total # blocks)$/\lceil n/b \rceil$

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, 97]

a query reports $\{2,3,4,5\}$

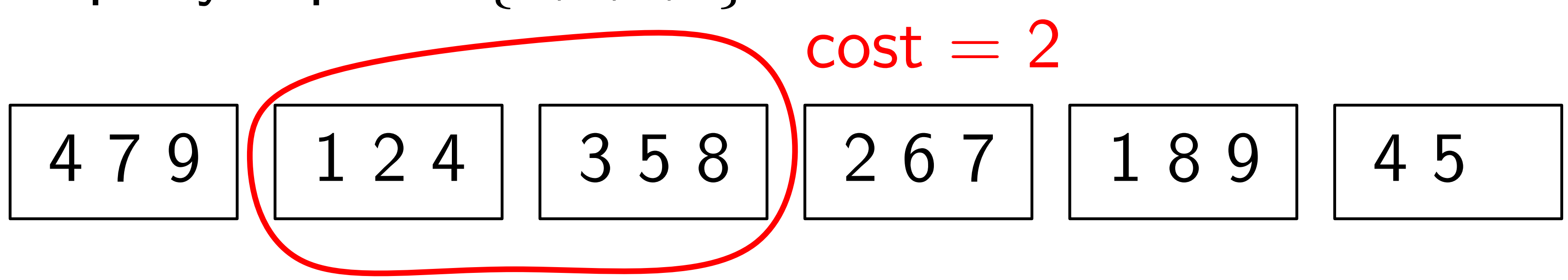


- Objects are stored in disk blocks of size up to b, possibly with redundancy.
  Redundancy $r = (\text{total } \# \text{ blocks})/\lceil n/b \rceil$
- The query cost is the minimum number of blocks that can cover all the required results (search time ignored!).
  Access overhead $A = (\text{worst-case})$ query cost $/\lceil k/b \rceil$

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, 97]

a query reports $\{2,3,4,5\}$



  - Objects are stored in disk blocks of size up to b, possibly with redundancy.
    Redundancy $r = (\text{total } \# \text{ blocks})/\lceil n/b \rceil$
  - The query cost is the minimum number of blocks that can cover all the required results (search time ignored!).
    Access overhead $A = (\text{worst-case})$ query cost $/\lceil k/b \rceil$
- Similar in spirit to popular lower bound models: cell probe model, semigroup model

Previous Results on Indexability

- Nearly all external indexing lower bounds are under this model

Previous Results on Indexability

- Nearly all external indexing lower bounds are under this model
- 2D range queries: $r = \Omega(\frac{\log(n/b)}{\log A})$ [Hellerstein, Koutsoupias, Papadimitriou, 97], [Koutsoupias, Taylor, 98], [Arge, Samoladas, Vitter, 99]

Previous Results on Indexability

- Nearly all external indexing lower bounds are under this model
- 2D range queries: $r = \Omega(\frac{\log(n/b)}{\log A})$ [Hellerstein, Koutsoupias, Papadimitriou, 97], [Koutsoupias, Taylor, 98], [Arge, Samoladas, Vitter, 99]
- 2D stabbing queries: $A_0 A_1^2 = \Omega(\frac{\log(n/b)}{\log r})$ [Arge, Samoladas, Yi, 04]
  - Refined access overhead: a query is covered by $A_0 + A_1 \cdot \lceil k/b \rceil$ blocks

Previous Results on Indexability

- Nearly all external indexing lower bounds are under this model
- 2D range queries: $r = \Omega(\frac{\log(n/b)}{\log A})$ [Hellerstein, Koutsoupias, Papadimitriou, 97], [Koutsoupias, Taylor, 98], [Arge, Samoladas, Vitter, 99]
- 2D stabbing queries: $A_0 A_1^2 = \Omega(\frac{\log(n/b)}{\log r})$ [Arge, Samoladas, Yi, 04]
  - Refined access overhead: a query is covered by $A_0 + A_1 \cdot \lceil k/b \rceil$ blocks
- 1D range queries: $A = O(1), r = O(1)$ trivially

Previous Results on Indexability

- Nearly all external indexing lower bounds are under this model
- 2D range queries: $r = \Omega(\frac{\log(n/b)}{\log A})$ [Hellerstein, Koutsoupias, Papadimitriou, 97], [Koutsoupias, Taylor, 98], [Arge, Samoladas, Vitter, 99]
- 2D stabbing queries: $A_0 A_1^2 = \Omega(\frac{\log(n/b)}{\log r})$ [Arge, Samoladas, Yi, 04]
  - Refined access overhead: a query is covered by $A_0 + A_1 \cdot \lceil k/b \rceil$ blocks
- 1D range queries: $A = O(1), r = O(1)$ trivially
  - Adding dynamization makes it much more interesting!

Dynamic Indexability

- Still consider only insertions

Dynamic Indexability

- Still consider only insertions

memory of size m	blocks of size $b = 3$	
time t: 1 2 7	4 7 9	4 5	$\leftarrow$ snapshot

Dynamic Indexability

- Still consider only insertions

	memory of size m	blocks of size $b = 3$		
time t:	1 2 7	4 7 9	4 5	← snapshot
time $t+1$:	1 2 6 7	4 7 9	4 5	6 inserted

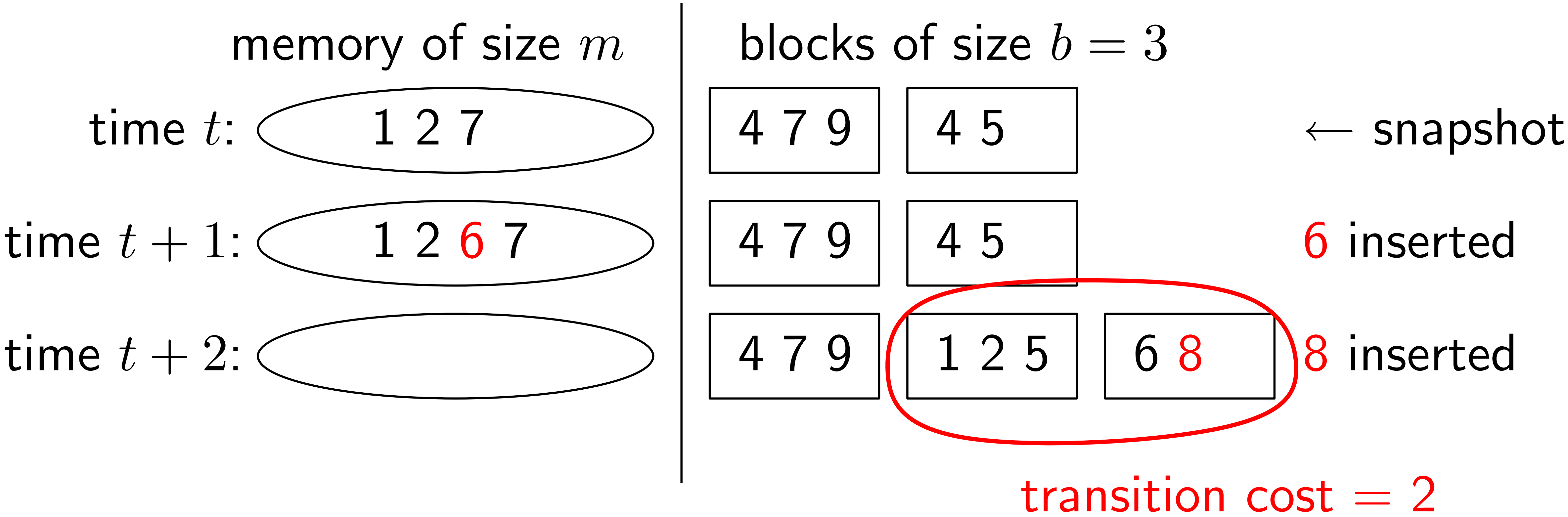
Dynamic Indexability

- Still consider only insertions

	memory of size m	blocks of size $b = 3$			
time t:	1 2 7	4 7 9	4 5		$\leftarrow$ snapshot
time $t+1$:	1 2 6 7	4 7 9	4 5		6 inserted
time $t+2$:		4 7 9	1 2 5	6 8	8 inserted

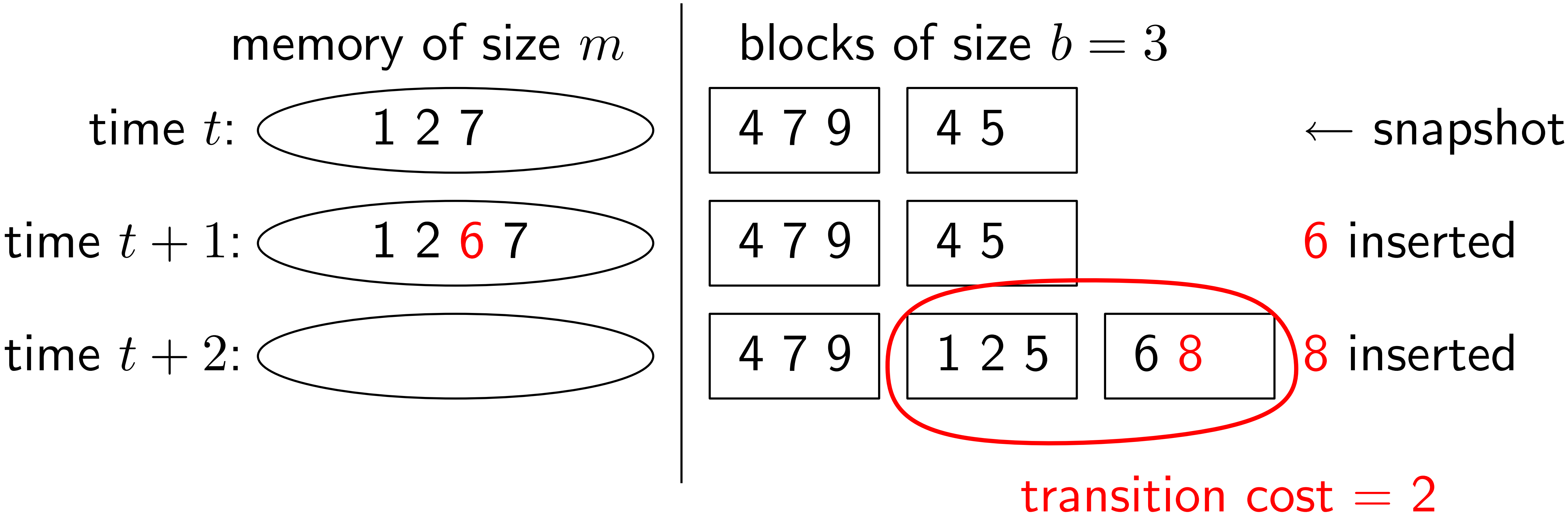
Dynamic Indexability

- Still consider only insertions



Dynamic Indexability

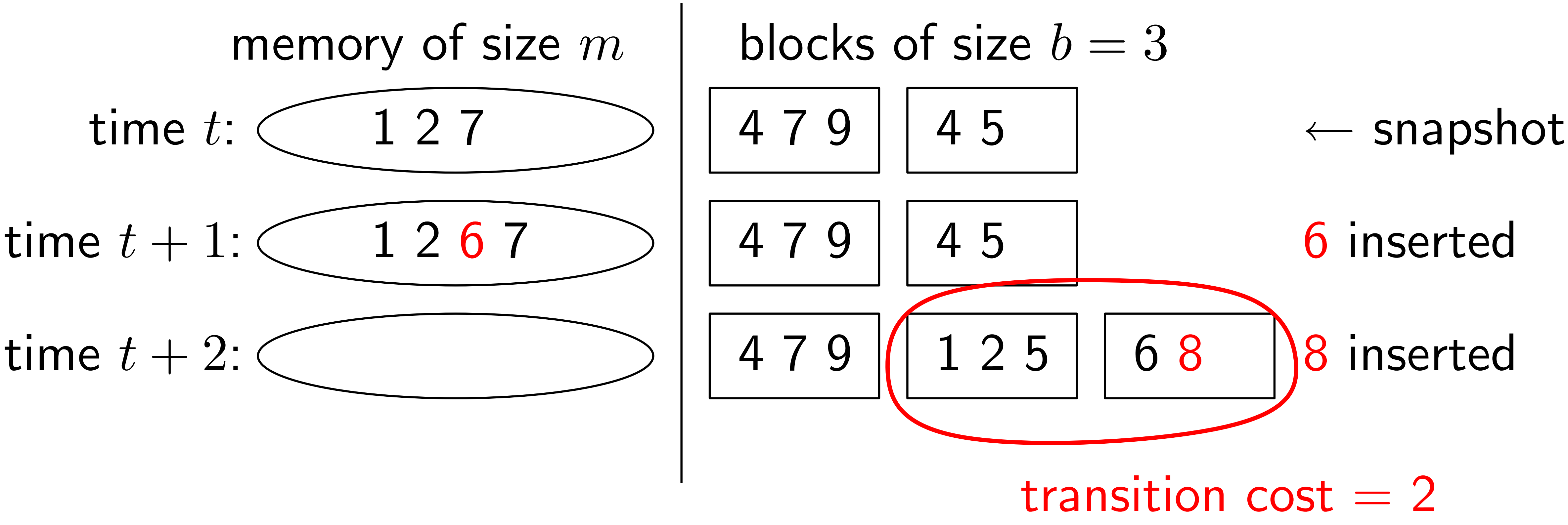
- Still consider only insertions



- Redundancy (access overhead) is the worst redundancy (access overhead) of all snapshots

Dynamic Indexability

- Still consider only insertions



- Redundancy (access overhead) is the worst redundancy (access overhead) of all snapshots

- Update cost: $u =$ the average transition cost per b insertions

Main Result Obtained in Dynamic Indexability

THEOREM: For any dynamic 1D range query index with access overhead A and update cost u, the following tradeoff holds, provided $n \geq 2mb^2$:

$$\begin{cases} A \cdot \log(u/A) = \Omega(\log b), & \text{for } A < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log A = \Omega(\log b), & \text{for all } A. \end{cases}$$

Main Result Obtained in Dynamic Indexability

THEOREM: For any dynamic 1D range query index with access overhead A and update cost u, the following tradeoff holds, provided $n \geq 2mb^2$:

$$\begin{cases} A \cdot \log(u/A) = \Omega(\log b), & \text{for } A < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log A = \Omega(\log b), & \text{for all } A. \end{cases}$$

Because a query cost $O(q + \lceil k/b \rceil)$ implies $O(q \cdot \lceil k/b \rceil)$

Main Result Obtained in Dynamic Indexability

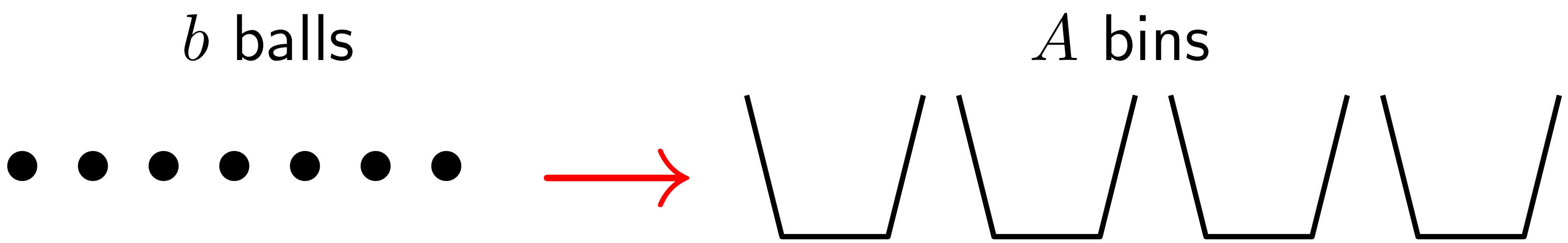
THEOREM: For any dynamic 1D range query index with access overhead A and update cost u, the following tradeoff holds, provided $n \geq 2mb^2$:

$$\begin{cases} A \cdot \log(u/A) = \Omega(\log b), & \text{for } A < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log A = \Omega(\log b), & \text{for all } A. \end{cases}$$

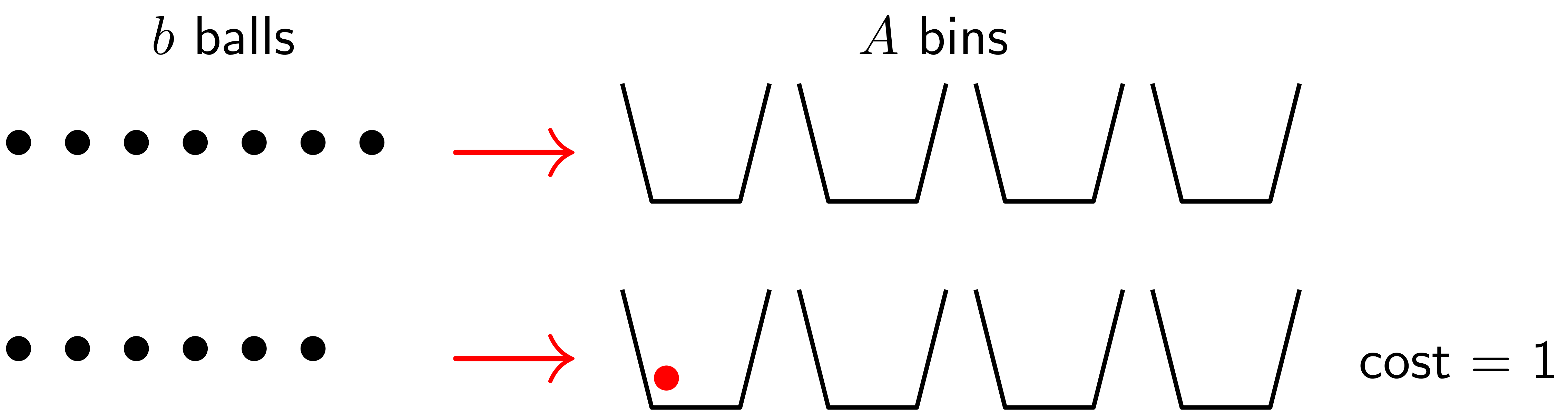
Because a query cost $O(q + \lceil k/b \rceil)$ implies $O(q \cdot \lceil k/b \rceil)$

The lower bound doesn't depend on the redundancy r!

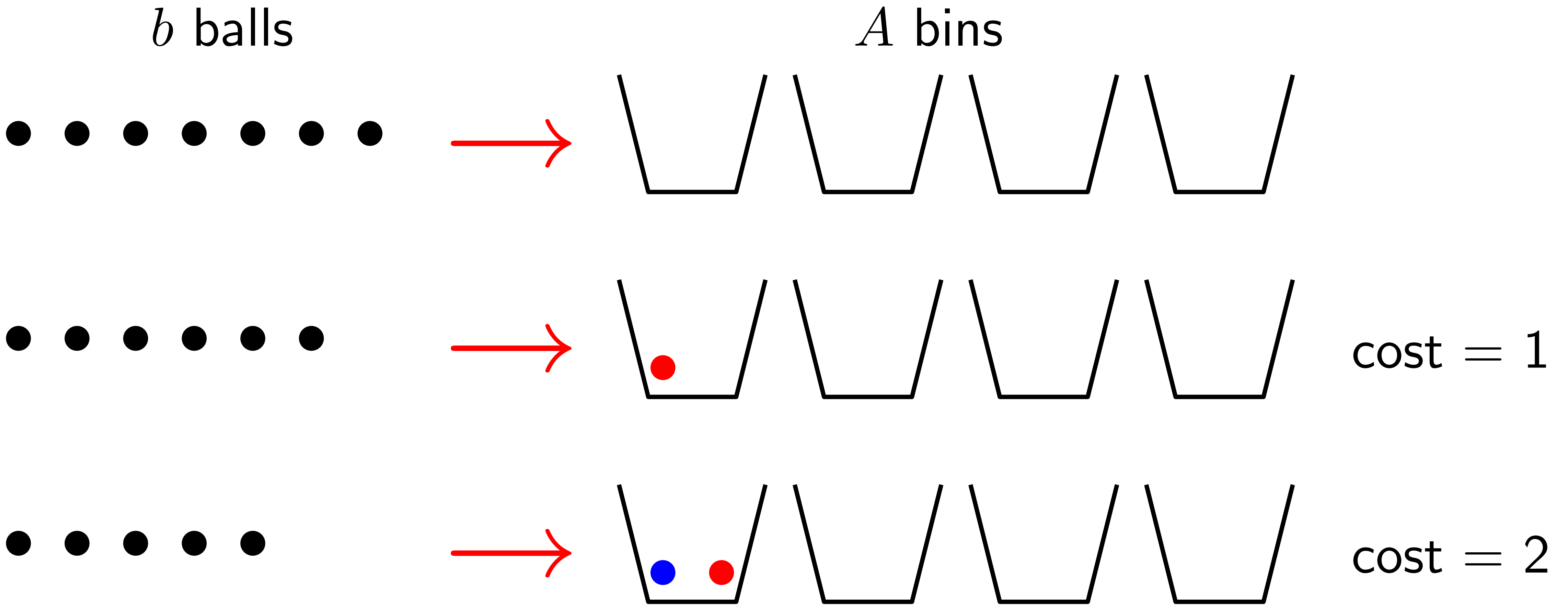
The Ball-Shuffling Problem



The Ball-Shuffling Problem



The Ball-Shuffling Problem

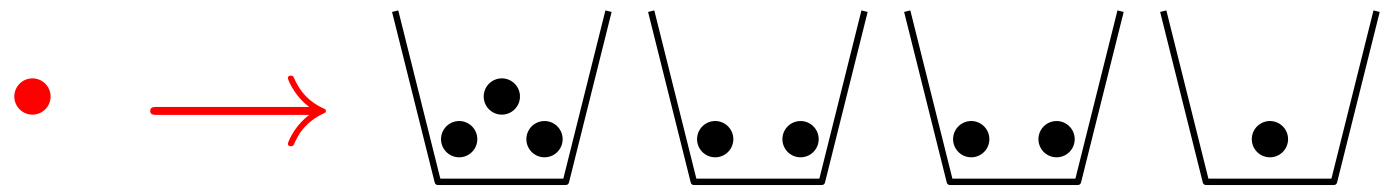


cost of putting the ball directly into a bin $=$ # balls in the bin $+ 1$

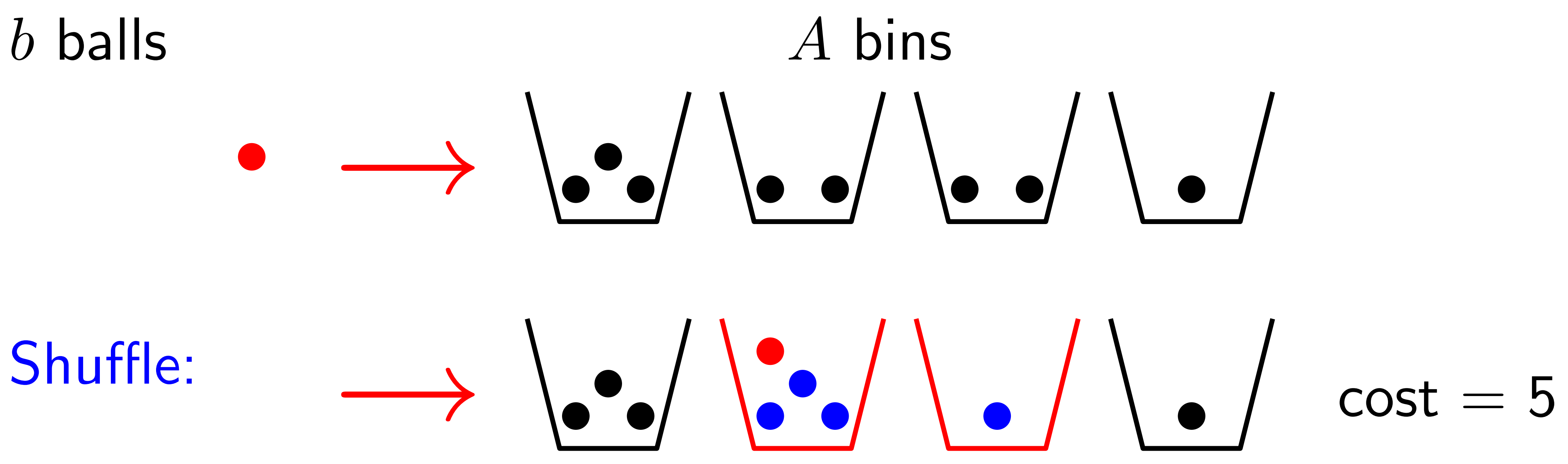
The Ball-Shuffling Problem

b balls

A bins

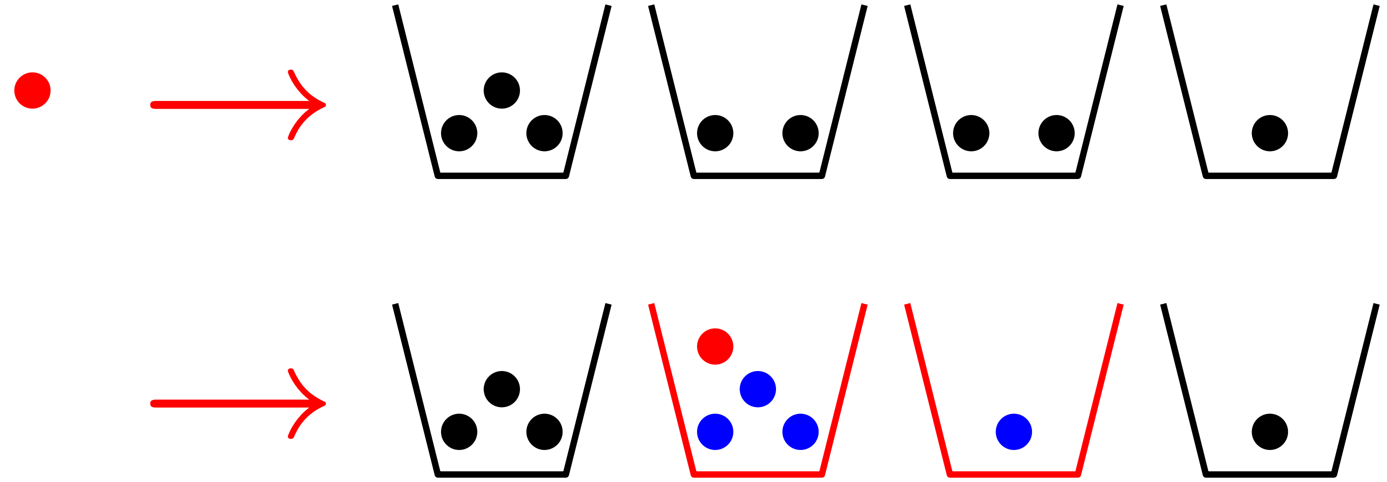


The Ball-Shuffling Problem



The Ball-Shuffling Problem

b balls A bins



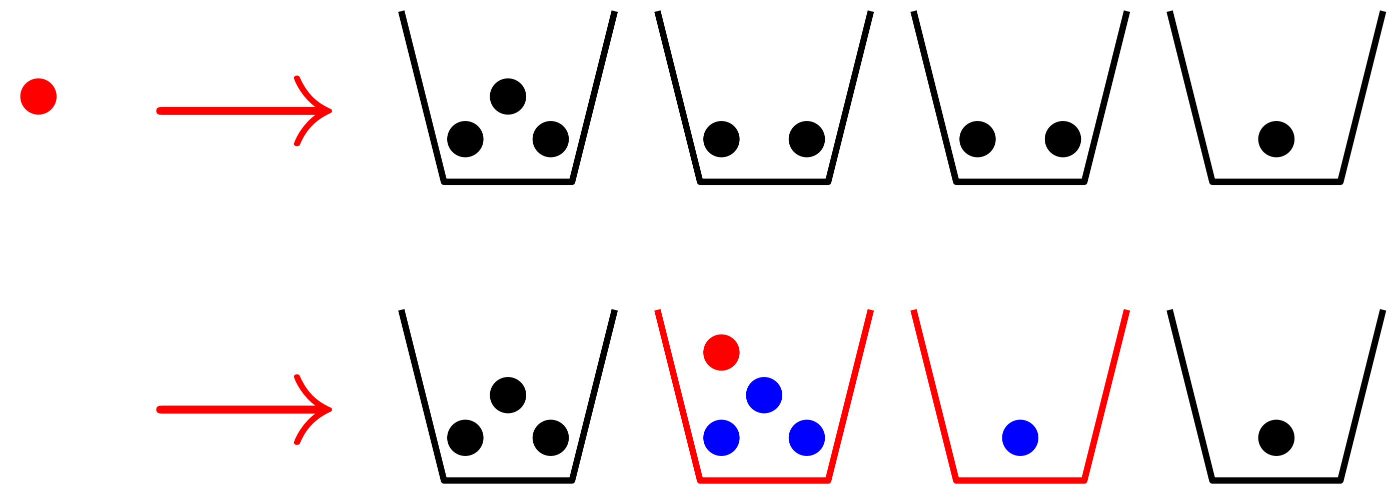
Shuffle: cost $=5$

Cost of shuffling $=$ # balls in the involved bins

The Ball-Shuffling Problem

b balls A bins

Shuffle:



cost $= 5$

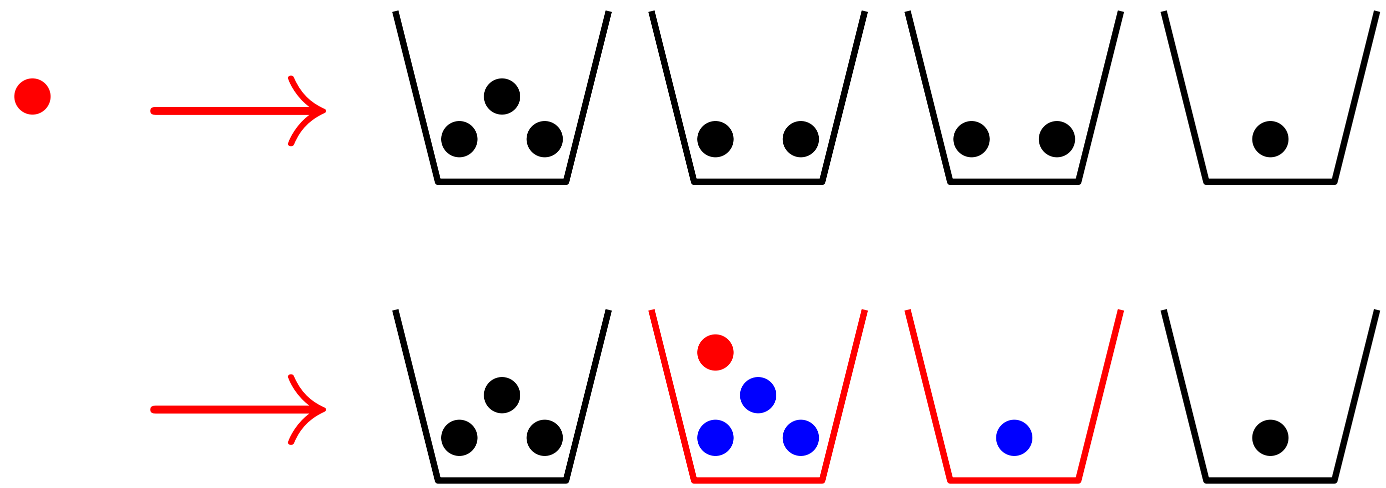
Cost of shuffling $= \#$ balls in the involved bins

Putting a ball directly into a bin is a special shuffle

The Ball-Shuffling Problem

b balls $\quad\quad$ A bins

Shuffle:



cost $= 5$

Cost of shuffling $=$ $\#$ balls in the involved bins

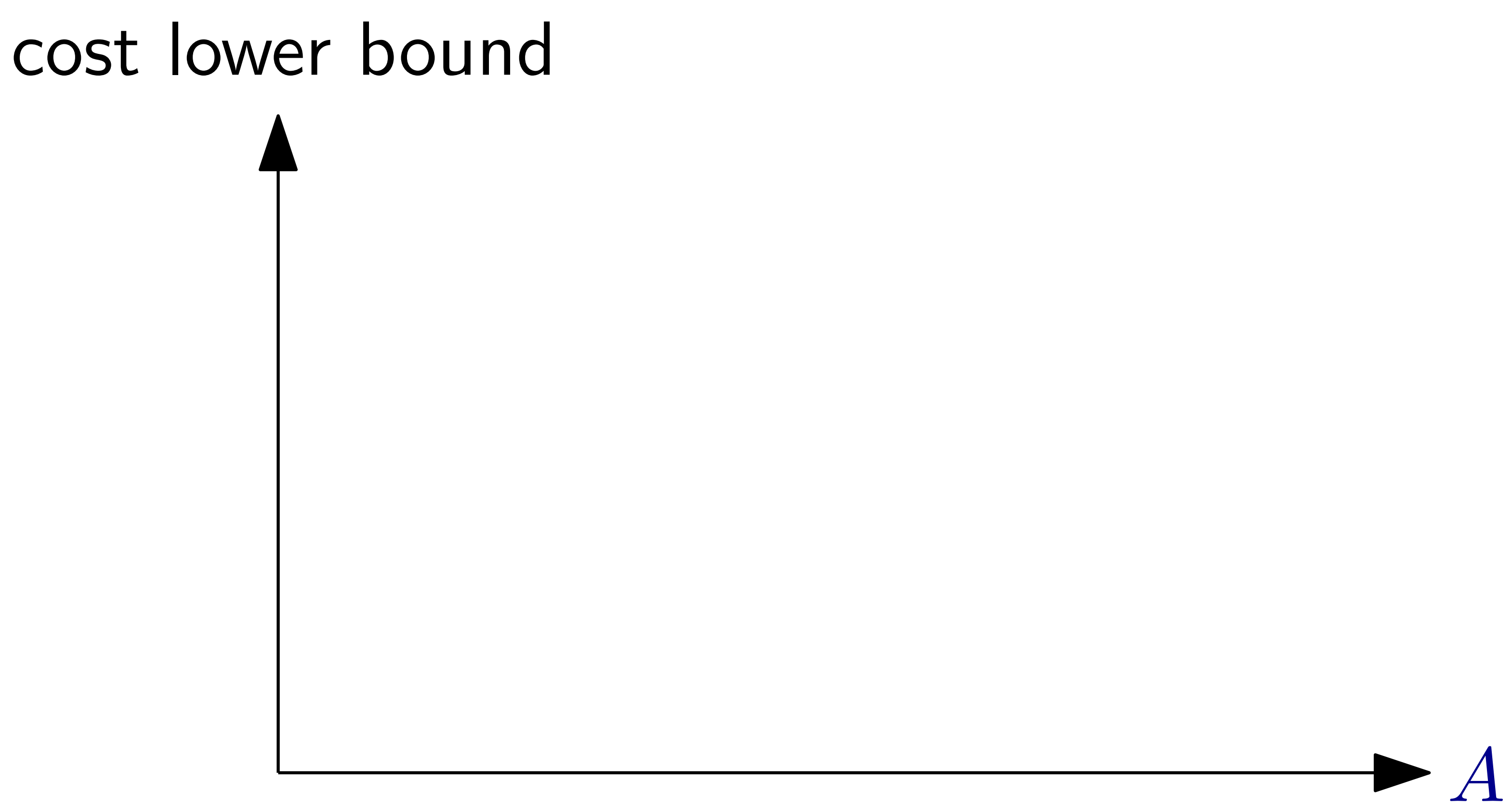
Putting a ball directly into a bin is a special shuffle

Goal: Accommodating all b balls using A bins with minimum cost

Ball-Shuffling Lower Bounds

THEOREM: The cost of any solution for the ball-shuffling problem is at least

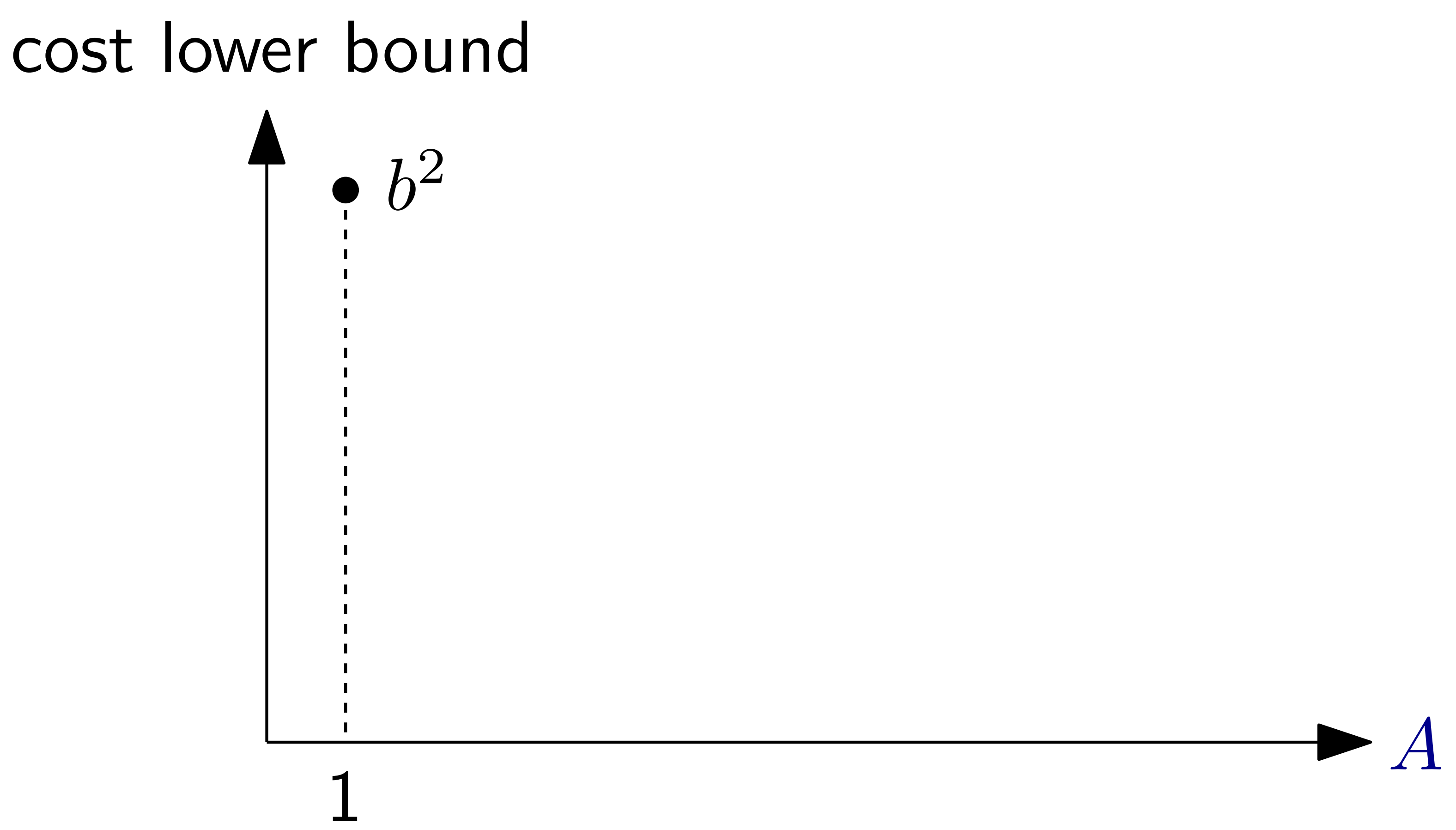
$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$



Ball-Shuffling Lower Bounds

THEOREM: The cost of any solution for the ball-shuffling problem is at least

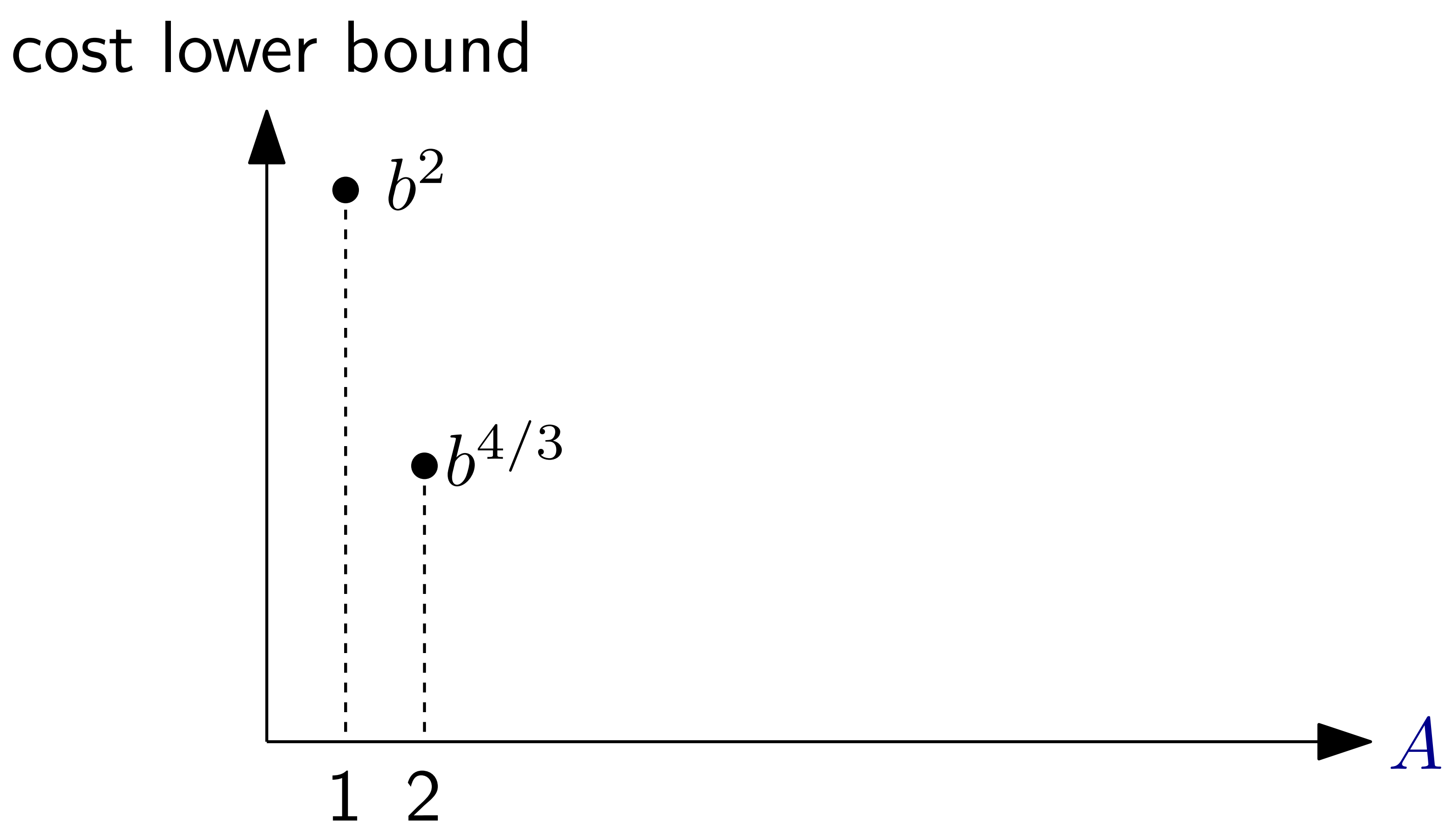
$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$



Ball-Shuffling Lower Bounds

THEOREM: The cost of any solution for the ball-shuffling problem is at least

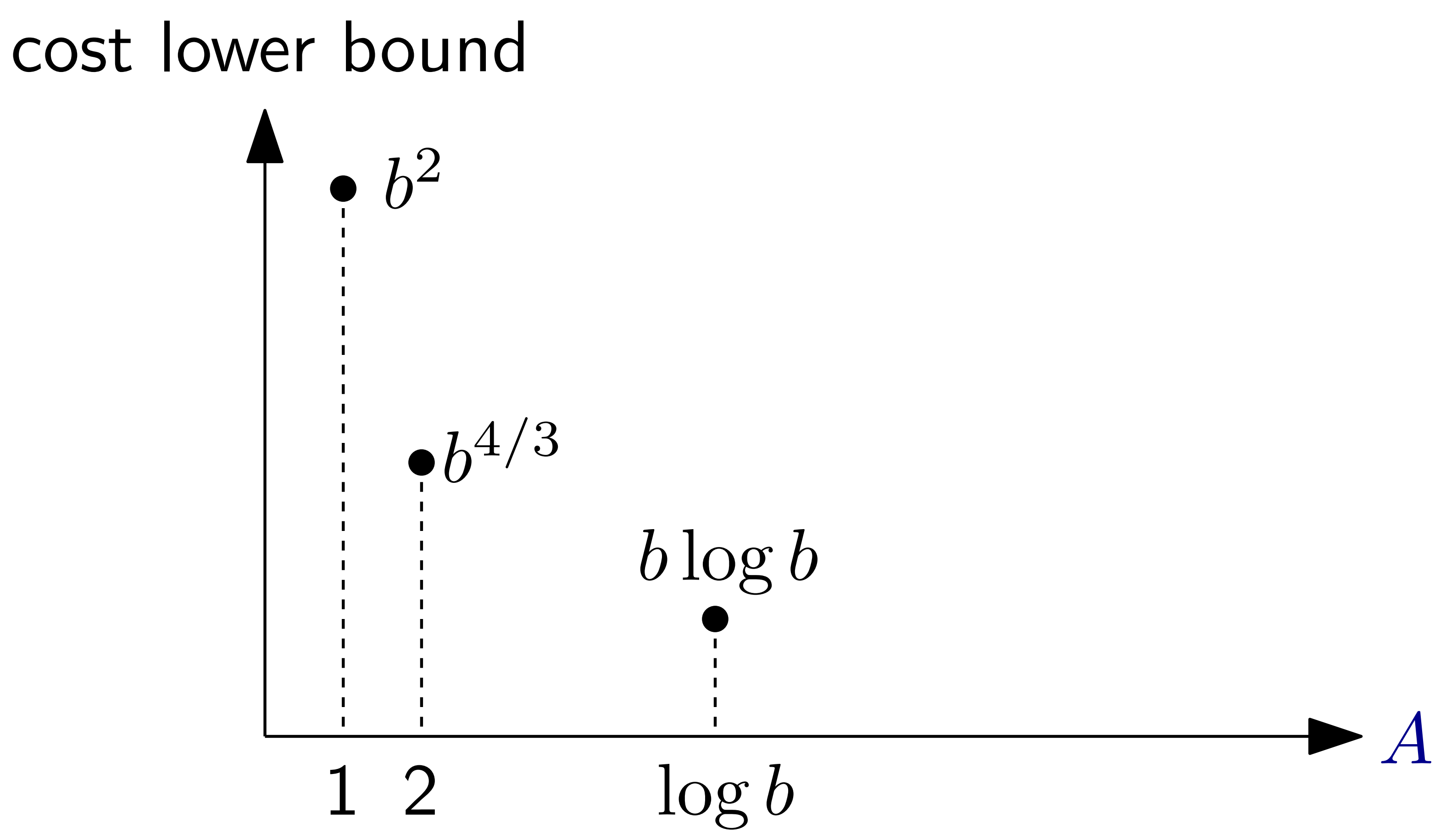
$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$



Ball-Shuffling Lower Bounds

THEOREM: The cost of any solution for the ball-shuffling problem is at least

$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

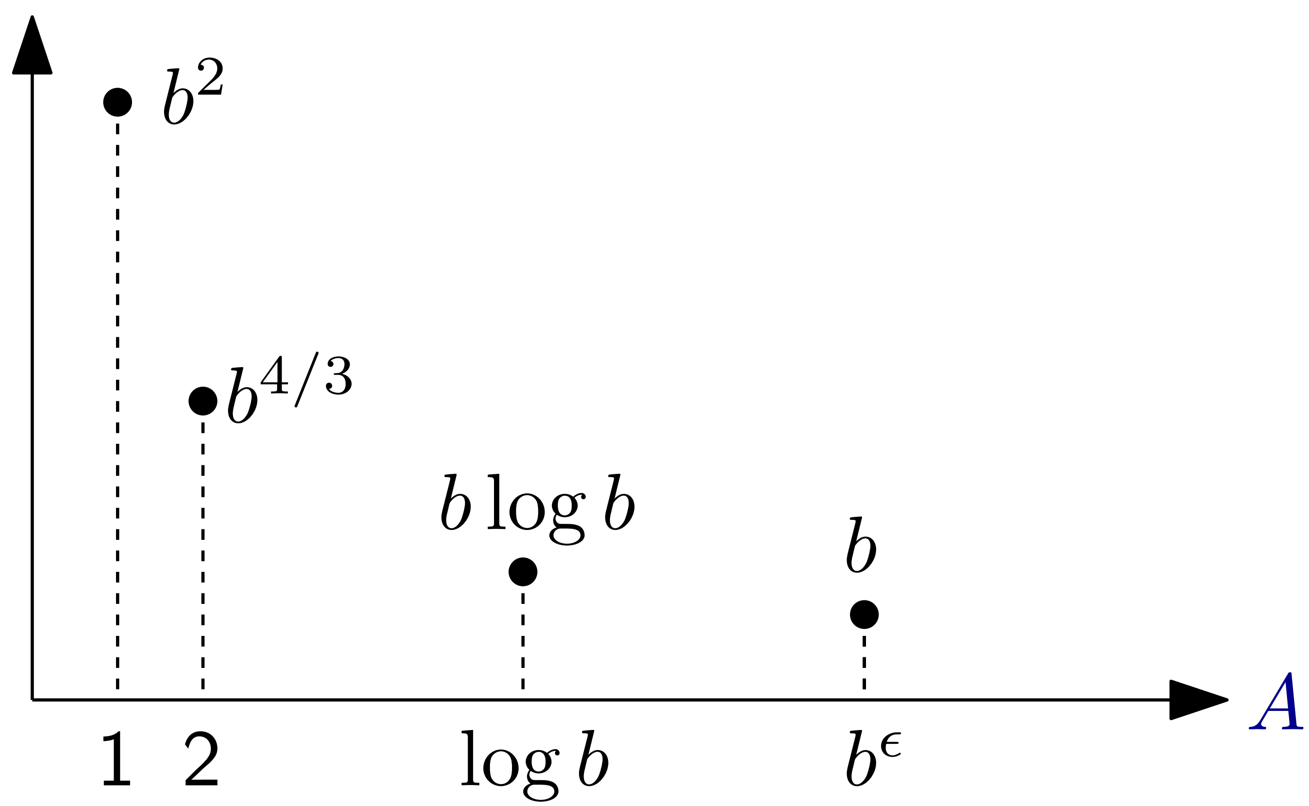


Ball-Shuffling Lower Bounds

THEOREM: The cost of any solution for the ball-shuffling problem is at least

$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

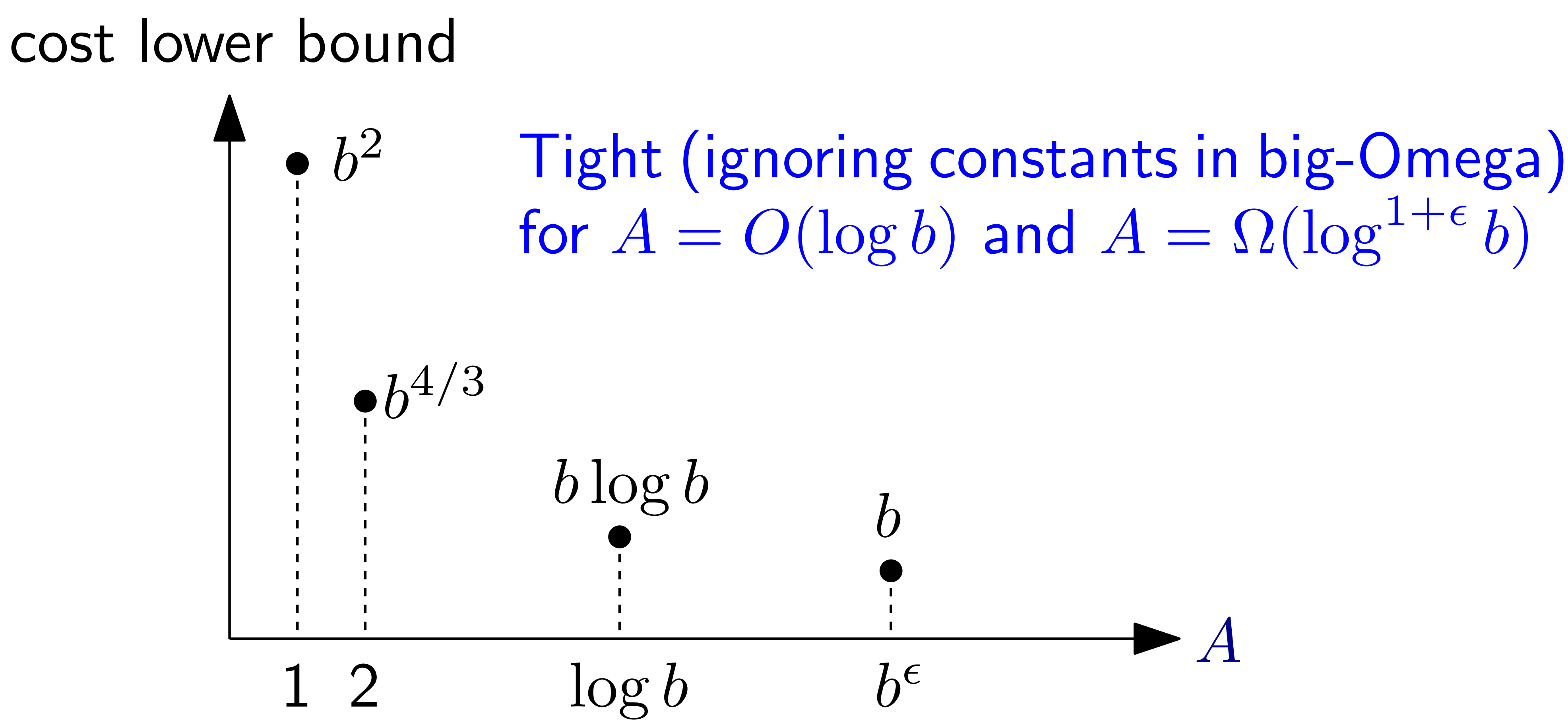
cost lower bound



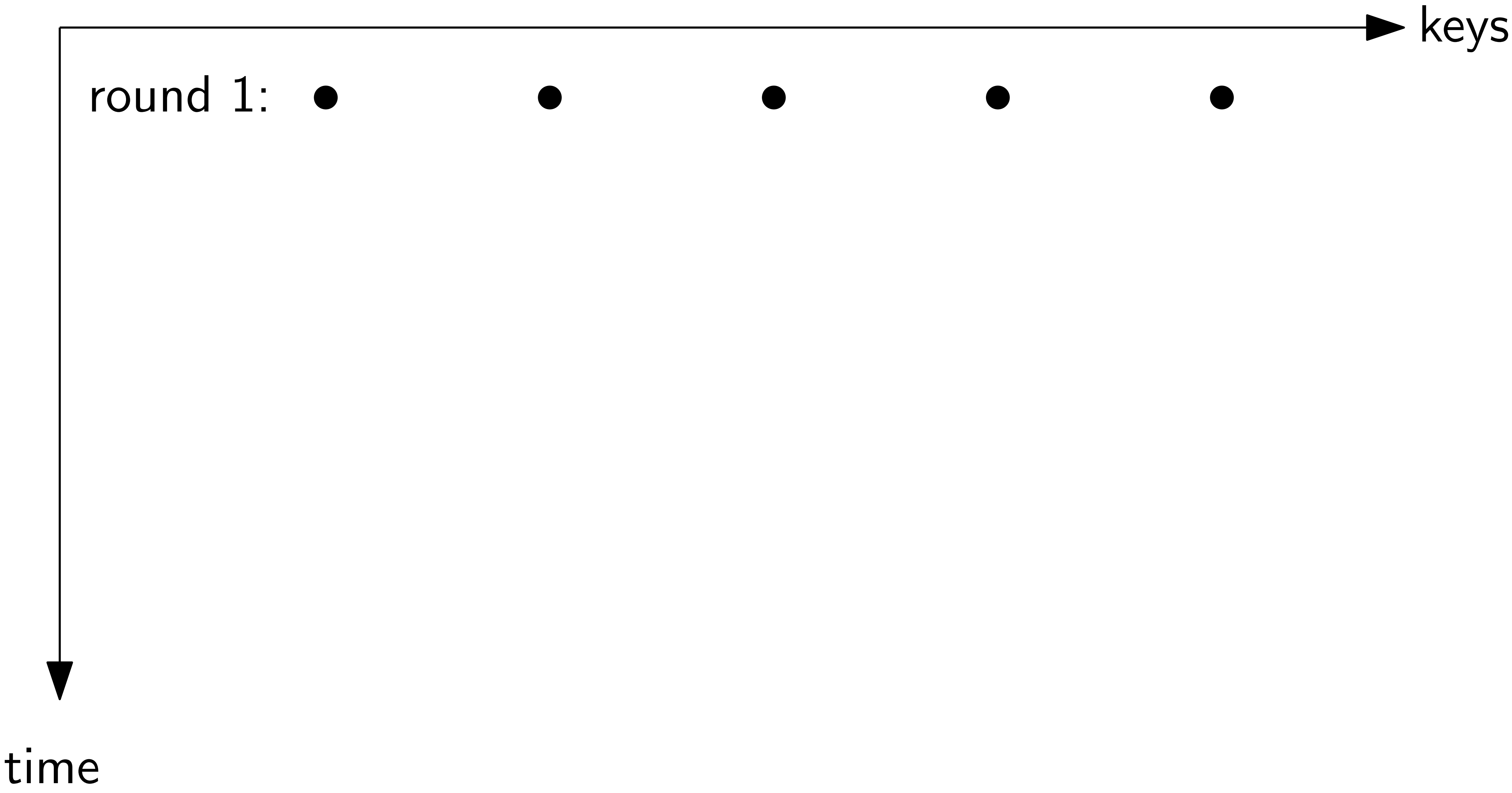
Ball-Shuffling Lower Bounds

THEOREM: The cost of any solution for the ball-shuffling problem is at least

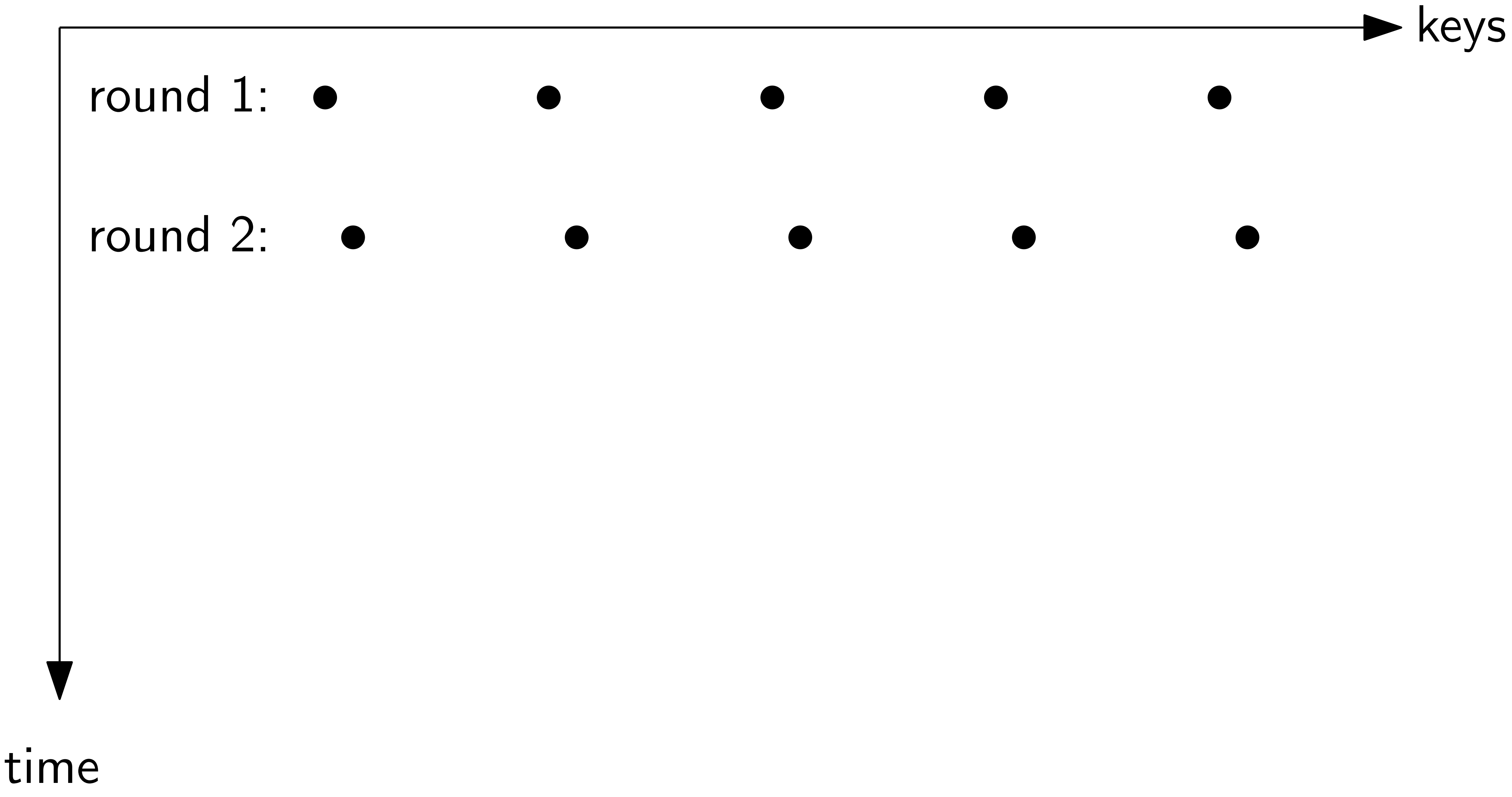
$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$



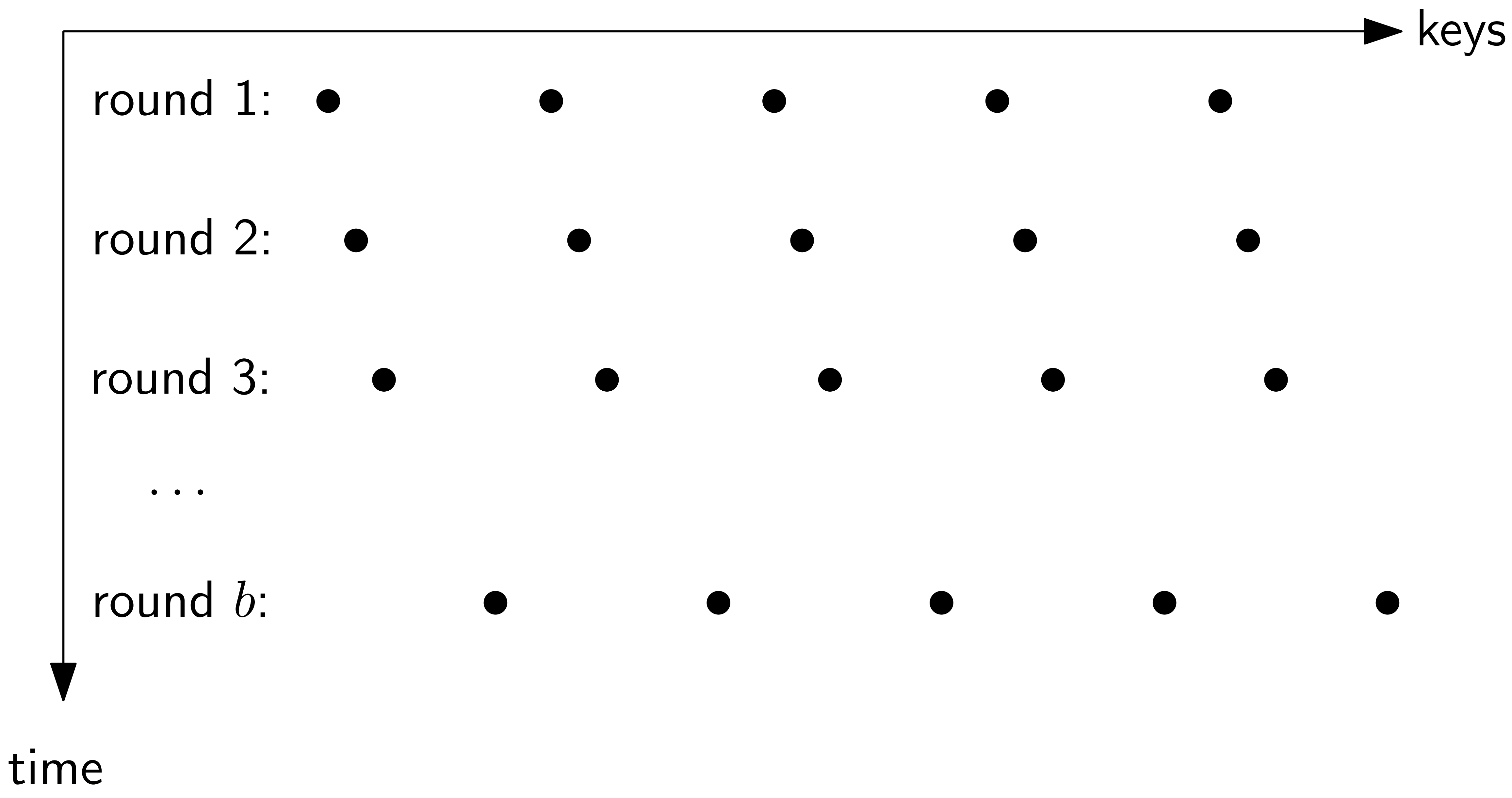
The Workload Construction



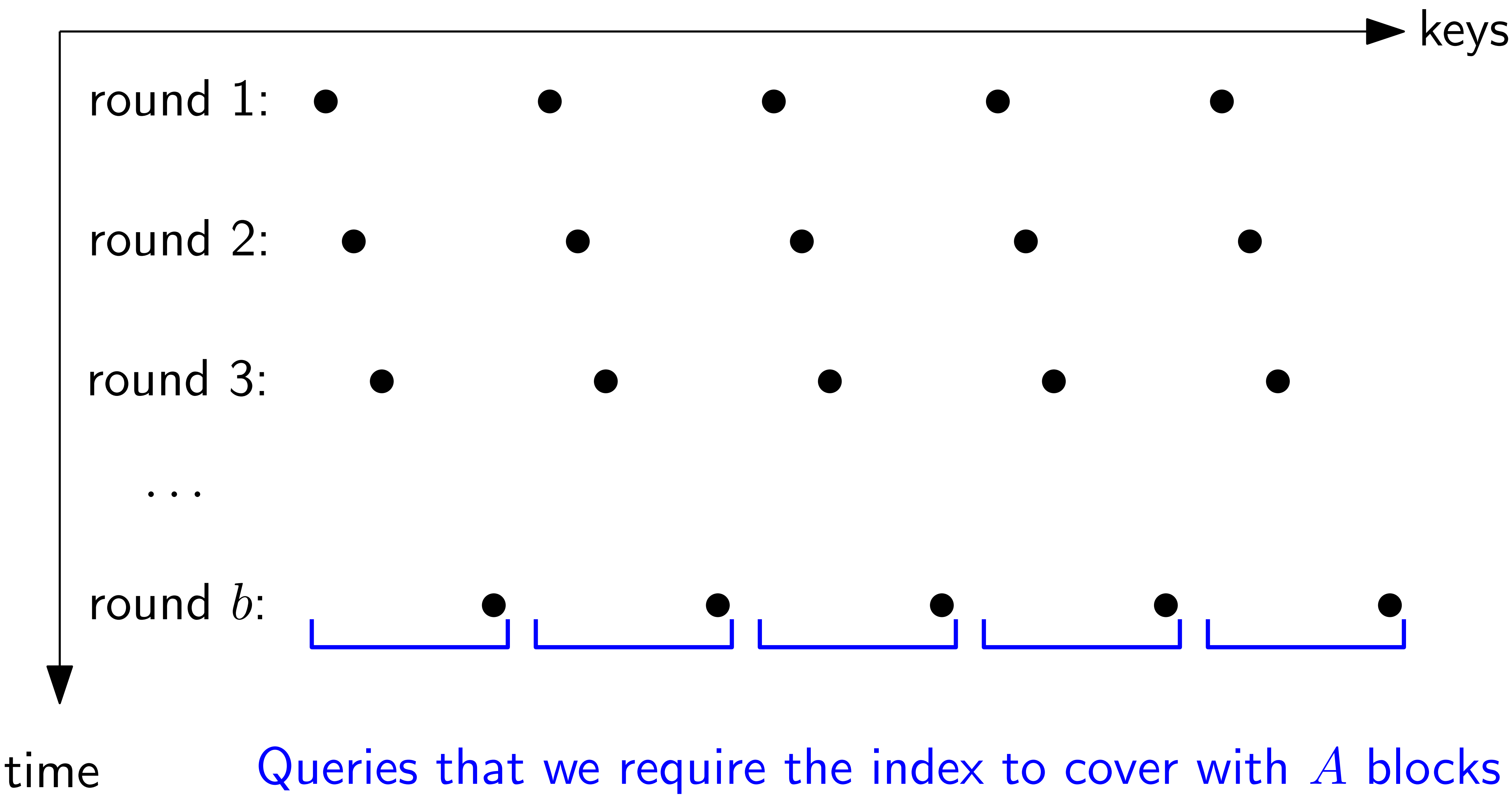
The Workload Construction



The Workload Construction



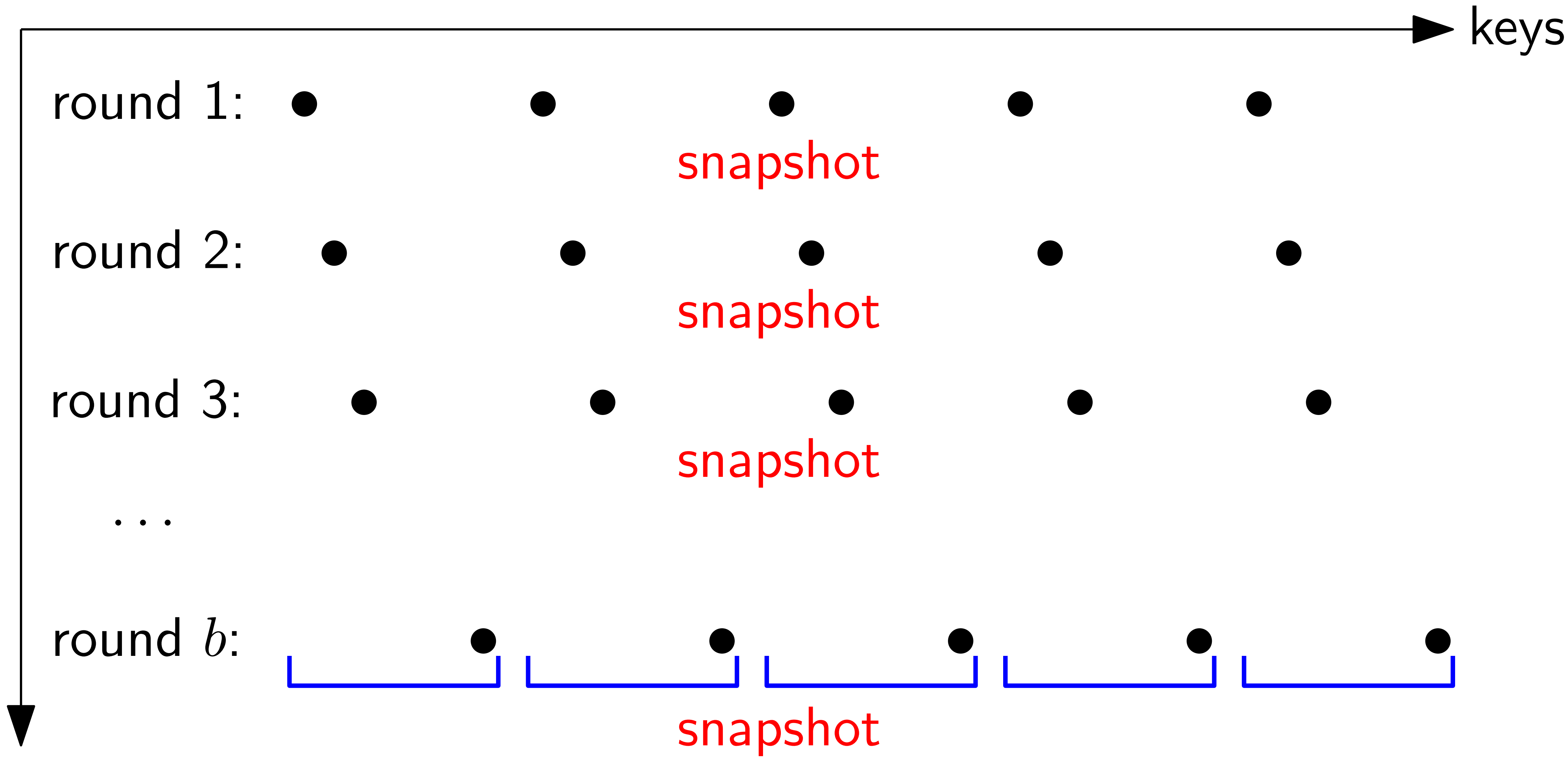
The Workload Construction



Queries that we require the index to cover with A blocks

$\#$ queries $\geq 2mb$

The Workload Construction

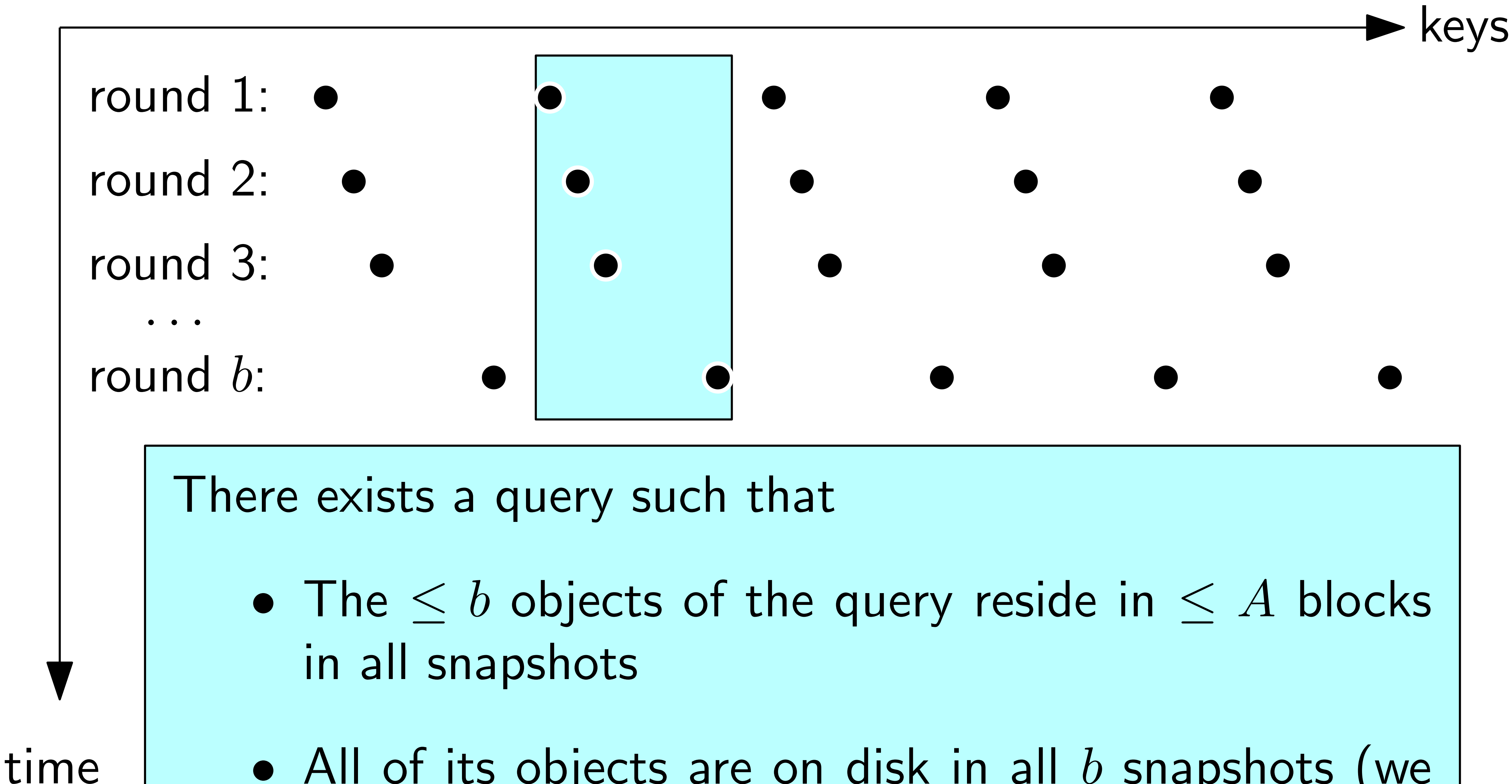


Queries that we require the index to cover with A blocks

$\#$ queries $\geq 2mb$

Snapshots of the dynamic index considered

The Workload Construction



There exists a query such that

- The $\leq b$ objects of the query reside in $\leq A$ blocks in all snapshots
- All of its objects are on disk in all b snapshots (we have $\geq mb$ queries)
- The index moves its objects ub times in total

The Reduction

An index with update cost u and access overhead A gives us a solution to the ball-shuffling game with cost ub for b balls and A bins

The Reduction

An index with update cost u and access overhead A gives us a solution to the ball-shuffling game with cost ub for b balls and A bins

Lower bound on the ball-shuffling problem:

THEOREM: The cost of any solution for the ball-shuffling problem is at least

$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

The Reduction

An index with update cost u and access overhead A gives us a solution to the ball-shuffling game with cost ub for b balls and A bins

Lower bound on the ball-shuffling problem:

THEOREM: The cost of any solution for the ball-shuffling problem is at least

$$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

$$\Downarrow$$

$$\begin{cases} A \cdot \log(u/A) = \Omega(\log b), & \text{for } A < \alpha \ln b, \alpha \text{ is any constant;} \\ u \cdot \log A = \Omega(\log b), & \text{for all } A. \end{cases}$$

Ball-Shuffling Lower Bound Proof

- $$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

Ball-Shuffling Lower Bound Proof

- $$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

- Will show: Any algorithm that handles the balls with an average cost of u using A bins cannot accommodate $(2A)^{2u}$ balls or more.

$$\Downarrow$$

$b < (2A)^{2u}$, or $u > \frac{\log b}{2\log(2A)}$, so the total cost of the algorithm is $ub = \Omega(b \log_A b)$.

Ball-Shuffling Lower Bound Proof

- $$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$
- Will show: Any algorithm that handles the balls with an average cost of u using A bins cannot accommodate $(2A)^{2u}$ balls or more.

  $$\Downarrow$$

  $b < (2A)^{2u}$, or $u > \frac{\log b}{2\log(2A)}$, so the total cost of the algorithm is $ub = \Omega(b \log_A b)$.
- Prove by induction on u
  - $u = 1$: Can handle at most A balls.

Ball-Shuffling Lower Bound Proof

- $$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$
- Will show: Any algorithm that handles the balls with an average cost of u using A bins cannot accommodate $(2A)^{2u}$ balls or more.

  $$\Downarrow$$

  $b < (2A)^{2u}$, or $u > \frac{\log b}{2\log(2A)}$, so the total cost of the algorithm is $ub = \Omega(b \log_A b)$.
- Prove by induction on u
  - $u = 1$: Can handle at most A balls.
  - $u \to u + \frac{1}{2}$?

Ball-Shuffling Lower Bound Proof (2)

- Need tol show: Any algorithm that handles the balls with an average cost of $u + \frac{1}{2}$ using A bins cannot accommodate $(2A)^{2u+1}$ balls or more.

  $$\Updownarrow$$

  To handle $(2A)^{2u+1}$ balls, any algorithm has to pay an average cost of more than $u + \frac{1}{2}$ per ball, or

  $$\left(u + \frac{1}{2}\right)(2A)^{2u+1} = (2Au + A)(2A)^{2u}$$

  in total.

Ball-Shuffling Lower Bound Proof (2)

- Need tol show: Any algorithm that handles the balls with an average cost of $u+\frac{1}{2}$ using A bins cannot accommodate $(2A)^{2u+1}$ balls or more.

$$\Updownarrow$$

To handle $(2A)^{2u+1}$ balls, any algorithm has to pay an average cost of more than $u+\frac{1}{2}$ per ball, or

$$\left(u+\frac{1}{2}\right)(2A)^{2u+1} = (2Au+A)(2A)^{2u}$$

in total.

- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.

Ball-Shuffling Lower Bound Proof (2)

- Need tol show: Any algorithm that handles the balls with an average cost of $u + \frac{1}{2}$ using A bins cannot accommodate $(2A)^{2u+1}$ balls or more.

  $$\Updownarrow$$

  To handle $(2A)^{2u+1}$ balls, any algorithm has to pay an average cost of more than $u + \frac{1}{2}$ per ball, or

  $$\left(u + \frac{1}{2}\right)(2A)^{2u+1} = (2Au + A)(2A)^{2u}$$

  in total.
- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.
  - Accommodating each batch by itself costs $u(2A)^{2u}$

Ball-Shuffling Lower Bound Proof (3)

- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.
  - Accommodating each batch by itself costs $u(2A)^{2u}$

Ball-Shuffling Lower Bound Proof (3)

- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.
  - Accommodating each batch by itself costs $u(2A)^{2u}$
- The "interference" among the $2A$ batches costs $> A(2A)^{2u}$

Ball-Shuffling Lower Bound Proof (3)

- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.
  - Accommodating each batch by itself costs $u(2A)^{2u}$
- The "interference" among the $2A$ batches costs $> A(2A)^{2u}$
  - If a batch has at least one ball that is never shuffled in later batches, it is a bad batch, otherwise it is a good batch.

Ball-Shuffling Lower Bound Proof (3)

- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.
  - Accommodating each batch by itself costs $u(2A)^{2u}$
- The "interference" among the $2A$ batches costs $> A(2A)^{2u}$
  - If a batch has at least one ball that is never shuffled in later batches, it is a bad batch, otherwise it is a good batch.
  - There are at most A bad batches

Ball-Shuffling Lower Bound Proof (3)

- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.
  - Accommodating each batch by itself costs $u(2A)^{2u}$
- The "interference" among the $2A$ batches costs $> A(2A)^{2u}$
  - If a batch has at least one ball that is never shuffled in later batches, it is a bad batch, otherwise it is a good batch.
  - There are at most A bad batches
  - There are at least A good batches

Ball-Shuffling Lower Bound Proof (3)

- Divide all balls into $2A$ batches of $(2A)^{2u}$ each.
  - Accommodating each batch by itself costs $u(2A)^{2u}$
- The "interference" among the $2A$ batches costs $> A(2A)^{2u}$
  - If a batch has at least one ball that is never shuffled in later batches, it is a bad batch, otherwise it is a good batch.
  - There are at most A bad batches
  - There are at least A good batches
  - Each good batch contributes at least $(2A)^{2u}$ to the "interference" cost

Lower Bound Proof: The Real Work

- $$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

Lower Bound Proof: The Real Work

- $$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$
- The merging lemma: There is an optimal ball-shuffling algorithm that only uses merging shuffles

Lower Bound Proof: The Real Work

- $$\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$$

- The merging lemma: There is an optimal ball-shuffling algorithm that only uses merging shuffles

- Let $f_A(b)$ be the minimum cost to accommodate b balls with A bins

Lower Bound Proof: The Real Work

- $\begin{cases} \Omega(A \cdot b^{1+\Omega(1/A)}), & \text{for } A < \alpha \ln b \text{ where } \alpha \text{ is any constant;} \\ \Omega(b \log_A b), & \text{for any } A. \end{cases}$
- The merging lemma: There is an optimal ball-shuffling algorithm that only uses merging shuffles
- Let $f_A(b)$ be the minimum cost to accommodate b balls with A bins
- The recurrence

$$f_{A+1}(b) \geq \min_{k, x_1+\cdots+x_k=b} \{f_A(x_1-1)+\cdots+f_A(x_k-1) + kx_1+(k-1)x_2+\cdots+x_k-b\}$$

Open Problems and Conjectures

- 1D range reporting
  - Current lower bound: query $\Omega(\log b)$, update $\Omega(\frac{1}{b}\log b)$. Improve to $(\log\frac{n}{m},\ \frac{1}{b}\log\frac{n}{m})$?

Open Problems and Conjectures

- 1D range reporting
  - Current lower bound: query $\Omega(\log b)$, update $\Omega(\frac{1}{b}\log b)$. Improve to $(\log \frac{n}{m}, \frac{1}{b}\log \frac{n}{m})$?
- Closely related problems: range sum (partial sum), predecessor search

The Grant Conjecture

	Internal memory (RAM) w: word size	External memory b: block size (in words)
range sum	$O : (\log n, \log n)$ binary tree $\Omega : (\log n, \log n)$ [Pătrașcu, Demaine, 06]	
predecessor		
range reporting		

The Grant Conjecture

	Internal memory (RAM) w: word size	External memory b: block size (in words)
range sum	$O : (\log n, \log n)$ binary tree $\Omega : (\log n, \log n)$ [Pătrașcu, Demaine, 06]	
predecessor	O : query = update = $\min\left\{\frac{\log\log n \log w}{\log\log w}, \sqrt{\frac{\log n}{\log\log n}}\right\}$ $\Omega : \ldots$ [Beame, Fich, 02]	
range reporting		

The Grant Conjecture

	Internal memory (RAM) w: word size	External memory b: block size (in words)
range sum	$O:(\log n, \log n)$ binary tree $\Omega:(\log n, \log n)$ [Pătrașcu, Demaine, 06]	
predecessor	O : query = update = $\min\left\{\frac{\log\log n \log w}{\log\log w}, \sqrt{\frac{\log n}{\log\log n}}\right\}$ $\Omega: \ldots$ [Beame, Fich, 02]	
range reporting	$O:(\log\log w, \log w)$ $O:(\log\log n, \log n/\log\log n)$ [Mortensen, Pagh, Pătrașcu, 05] Ω : open	

The Grant Conjecture

	Internal memory (RAM) w: word size	External memory b: block size (in words)
range sum	$O:(\log n, \log n)$ binary tree $\Omega:(\log n, \log n)$ [Pătraşcu, Demaine, 06]	$O:(\log_\ell \frac{n}{m}, \frac{\ell}{b}\log_\ell \frac{n}{m})$ B-tree + logarithmic method
predecessor	O : query = update = $\min\left\{\frac{\log\log n \log w}{\log\log w}, \sqrt{\frac{\log n}{\log\log n}}\right\}$ $\Omega:\ldots$ [Beame, Fich, 02]	
range reporting	$O:(\log\log w, \log w)$ $O:(\log\log n, \log n/\log\log n)$ [Mortensen, Pagh, Pătraşcu, 05] Ω : open	

The Grant Conjecture

	Internal memory (RAM) w: word size	External memory b: block size (in words)
range sum	$O : (\log n, \log n)$ binary tree $\Omega : (\log n, \log n)$ [Pătrașcu, Demaine, 06]	$O : (\log_\ell \frac{n}{m}, \frac{\ell}{b} \log_\ell \frac{n}{m})$ B-tree + logarithmic method Optimal for all three?
predecessor	O : query = update = $\min\left\{\frac{\log\log n \log w}{\log\log w}, \sqrt{\frac{\log n}{\log\log n}}\right\}$ $\Omega : \ldots$ [Beame, Fich, 02]	
range reporting	$O : (\log\log w, \log w)$ $O : (\log\log n, \log n / \log\log n)$ [Mortensen, Pagh, Pătrașcu, 05] Ω : open	

The Grant Conjecture

	Internal memory (RAM) w: word size	External memory b: block size (in words)
range sum	$O: (\log n, \log n)$ binary tree $\Omega: (\log n, \log n)$ [Pătrașcu, Demaine, 06]	$O: (\log_\ell \frac{n}{m}, \frac{\ell}{b} \log_\ell \frac{n}{m})$ B-tree + logarithmic method Optimal for all three? How large does b need to be for B-tree to be optimal?
predecessor	O : query = update = $\min\left\{\frac{\log\log n \log w}{\log\log w}, \sqrt{\frac{\log n}{\log\log n}}\right\}$ $\Omega: \ldots$ [Beame, Fich, 02]	
range reporting	$O: (\log\log w, \log w)$ $O: (\log\log n, \log n / \log\log n)$ [Mortensen, Pagh, Pătrașcu, 05] Ω : open	

The Grant Conjecture

	Internal memory (RAM) w: word size	External memory b: block size (in words)
range sum	$O : (\log n, \log n)$ binary tree $\Omega : (\log n, \log n)$ [Pătrașcu, Demaine, 06]	$O : (\log_\ell \frac{n}{m}, \frac{\ell}{b} \log_\ell \frac{n}{m})$ B-tree + logarithmic method Optimal for all three? How large does b need to be for B-tree to be optimal? We now know this is true for range reporting for $b = (\frac{n}{m})^{\Omega(1)}$; false for $b = o(\log \log n)$
predecessor	O : query = update = $\min \left\{ \frac{\log \log n \log w}{\log \log w}, \sqrt{\frac{\log n}{\log \log n}} \right\}$ $\Omega : \ldots$ [Beame, Fich, 02]	
range reporting	$O : (\log \log w, \log w)$ $O : (\log \log n, \log n / \log \log n)$ [Mortensen, Pagh, Pătrașcu, 05] Ω : open	

The End

THANK YOU

Q and A