Dynamic Indexability and the Optimality of B-trees and Hash Tables

Ke Yi

Hong Kong University of Science and Technology

Dynamic Indexability and Lower Bounds for Dynamic One-Dimensional Range Query Indexes, *PODS* '09

Dynamic External Hashing: The Limit of Buffering, with Zhewei Wei and Qin Zhang, *SPAA* '09

+ some latest development

An index is . . .

- An index is a single number calculated from a set of prices
  - Dow Jones, S & P, Hang Seng

An index is . . .

- An index is a single number calculated from a set of prices
    - Dow Jones, S & P, Hang Seng
- An index is a list of keywords and their page numbers in a book
- An index is an exponent
- An index is a finger
- An index is a list of academic publications and their citations

An index is . . .

- An index is a single number calculated from a set of prices
  - Dow Jones, S & P, Hang Seng
- An index is a list of keywords and their page numbers in a book
- An index is an exponent
- An index is a finger
- An index is a list of academic publications and their citations
- An index (search engine) is an inverted list from keywords to web pages
- An index (database) is a (disk-based) data structure that improves the speed of data retrieval operations (queries) on a database table.

Hash Table and B-tree

- Hash tables and B-trees are taught to undergrads and actually used in all database systems

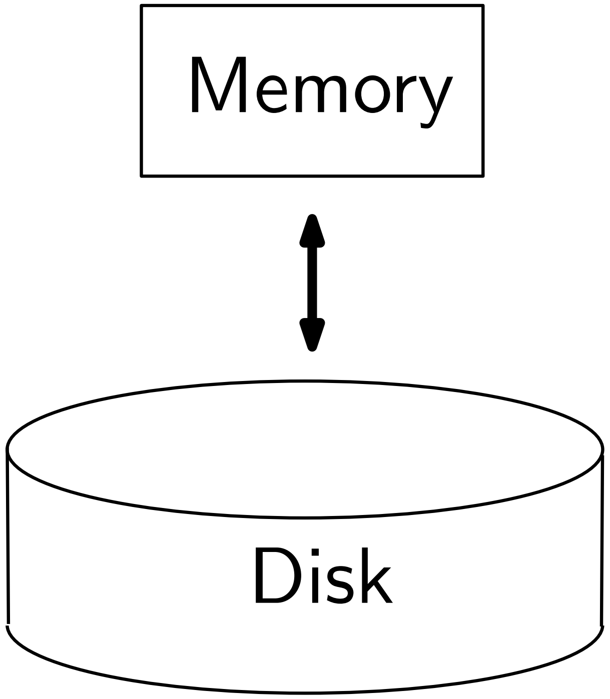
Hash Table and B-tree

- Hash tables and B-trees are taught to undergrads and actually used in all database systems
- B-tree: lookups and range queries; Hash table: lookups

Hash Table and B-tree

- Hash tables and B-trees are taught to undergrads and actually used in all database systems
- B-tree: lookups and range queries; Hash table: lookups

External memory model (I/O model):

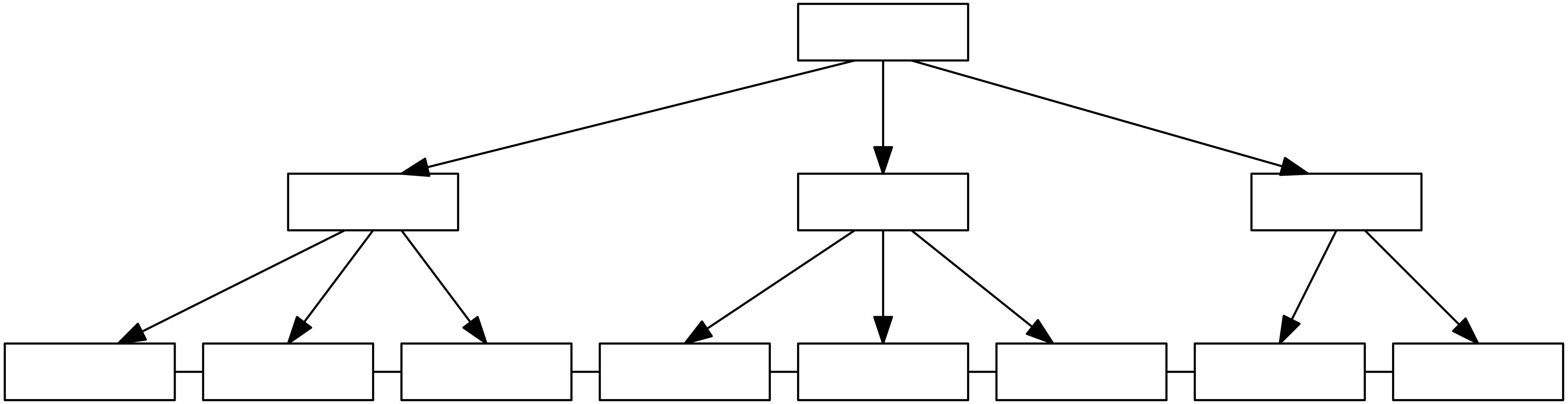


Memory of size M

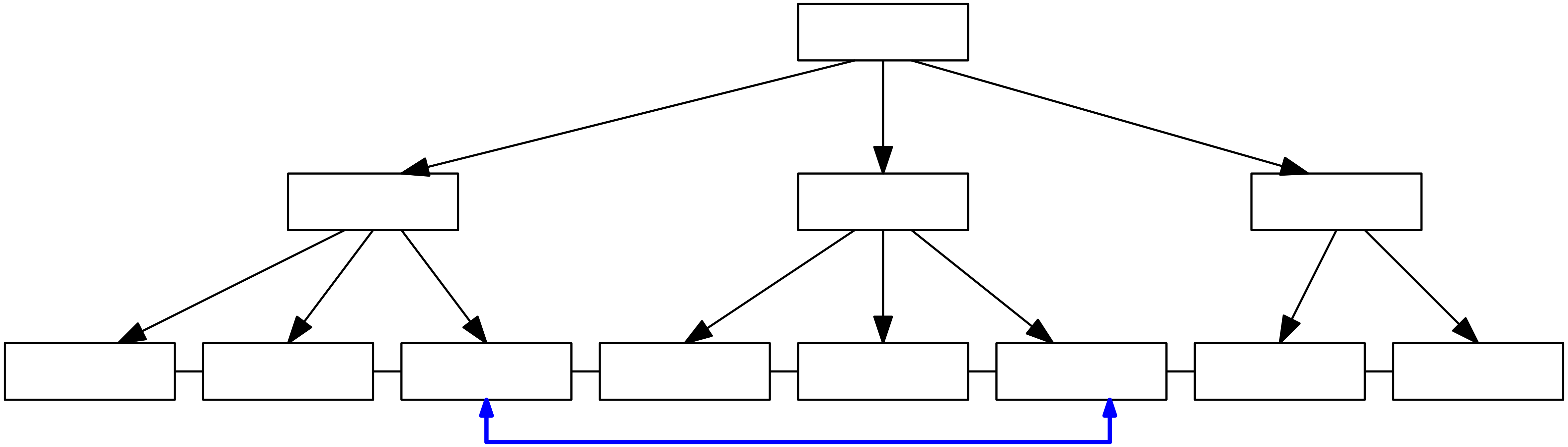
Each I/O reads/writes a block

Disk partitioned into blocks of size B

The B-tree



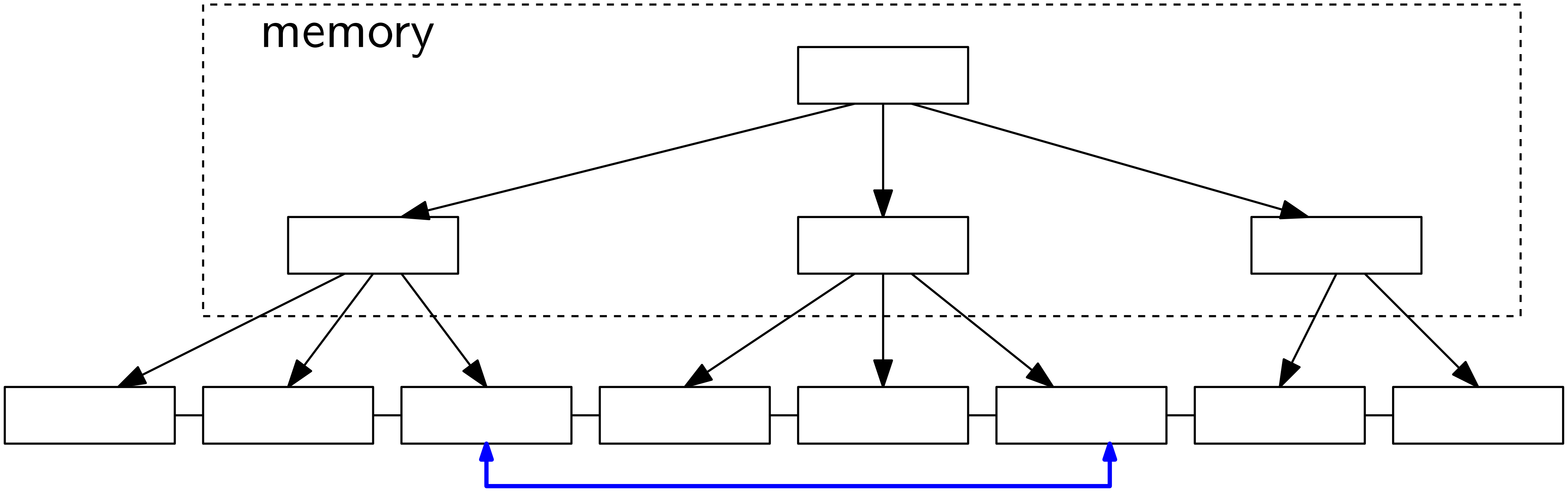
The B-tree



A range query in $O(\log_B N + K/B)$ I/Os

K: output size

The B-tree

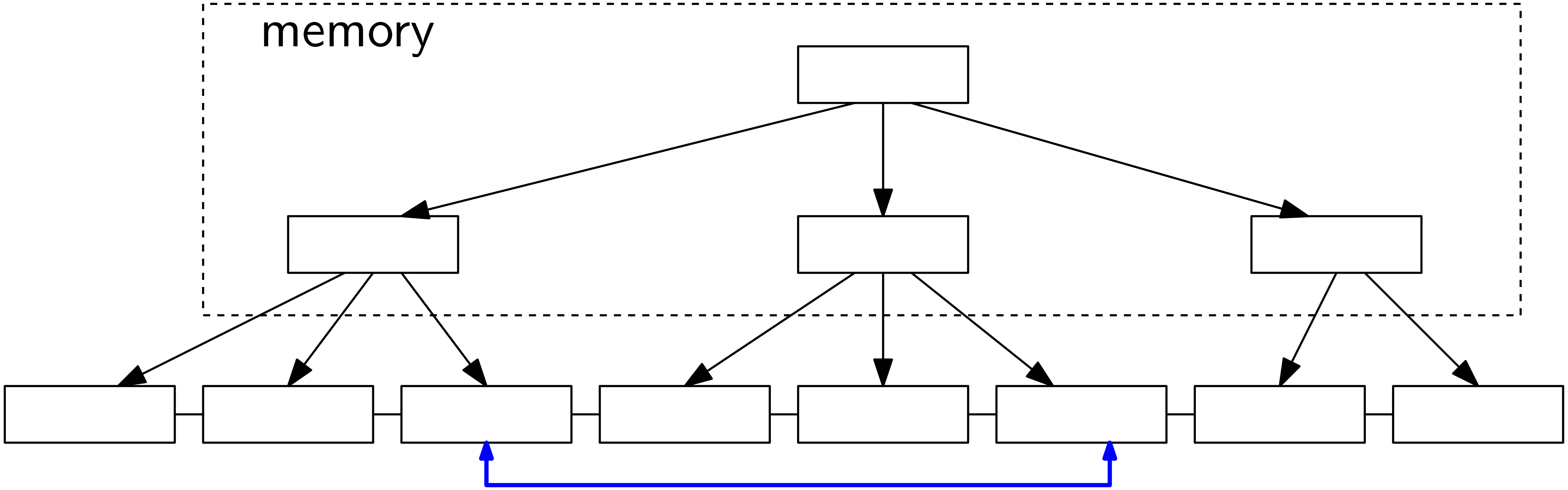


A range query in $O(\cancel{\log_B} N + K/B)$ I/Os

K: output size

$$\log_B N - \log_B M = \log_B \frac{N}{M}$$

The B-tree



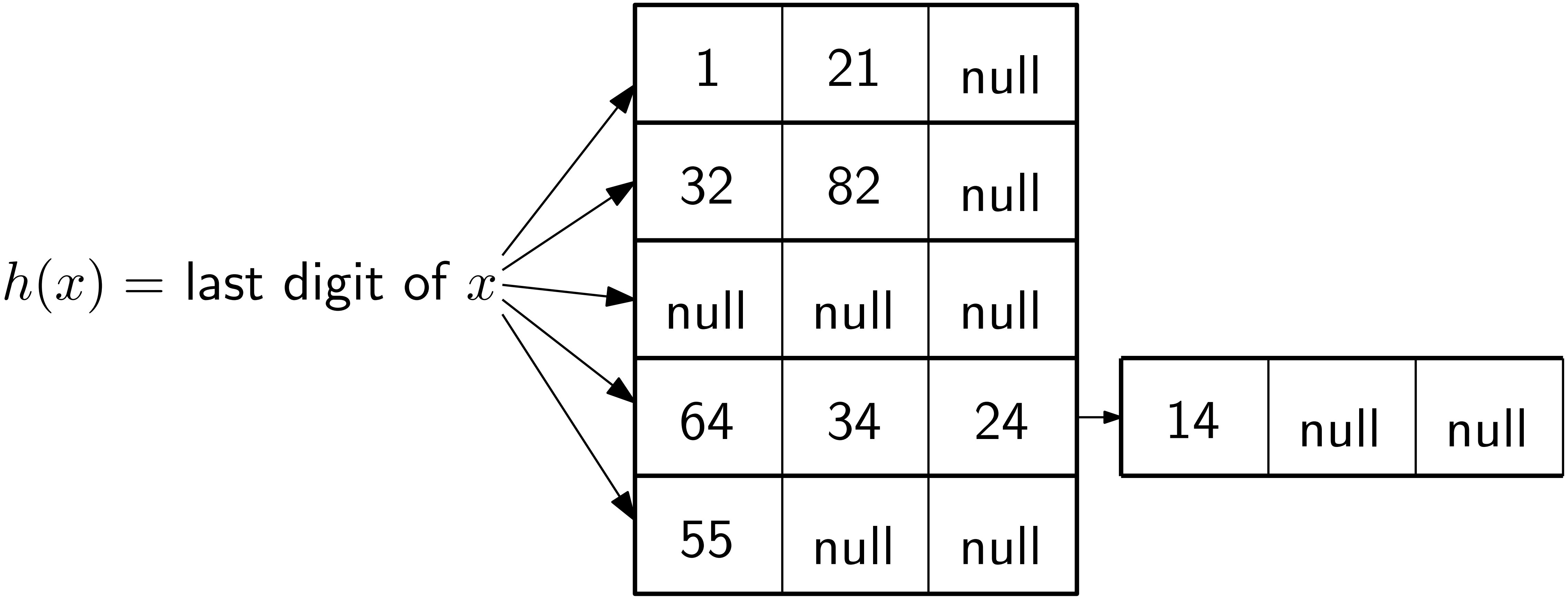
A range query in $O(\cancel{\log_B} N + K/B)$ I/Os

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 $\log_B \frac{N}{M} = O(1)$.

External Hashing

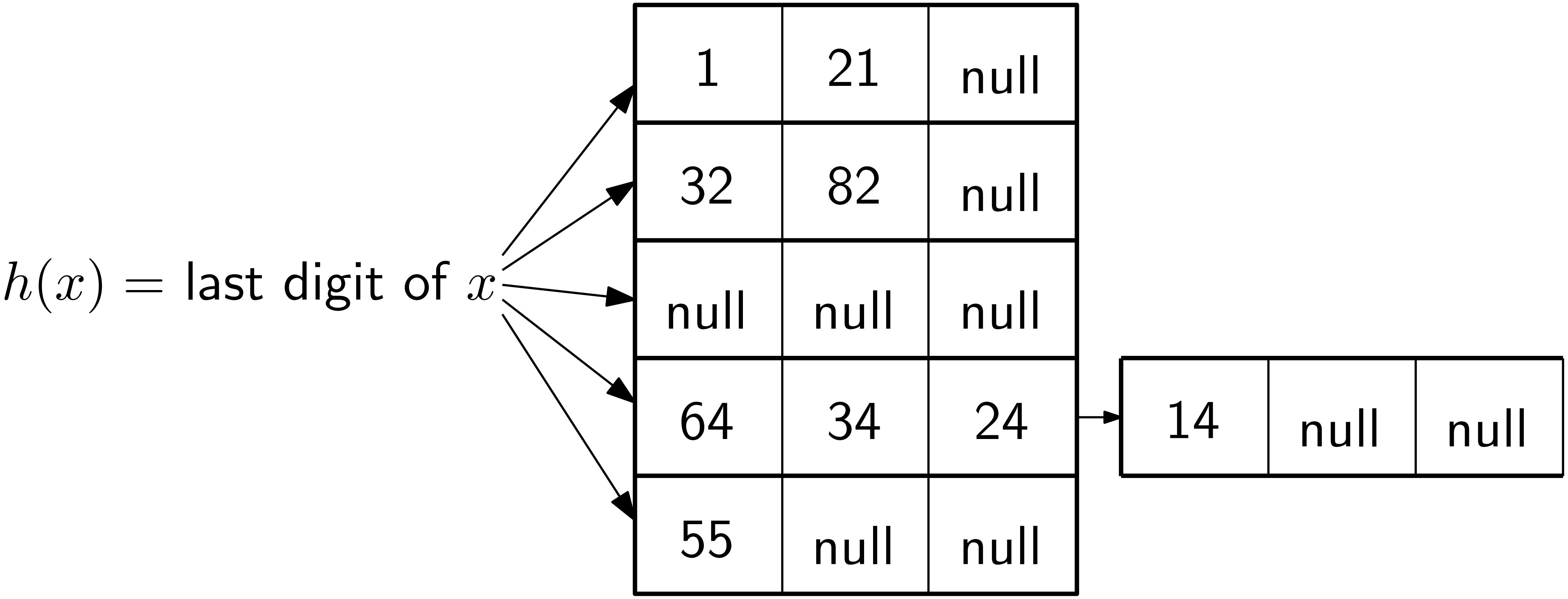


$h(x) =$ last digit of x

1	21	null
32	82	null
null	null	null
64	34	24
55	null	null

14	null	null

External Hashing



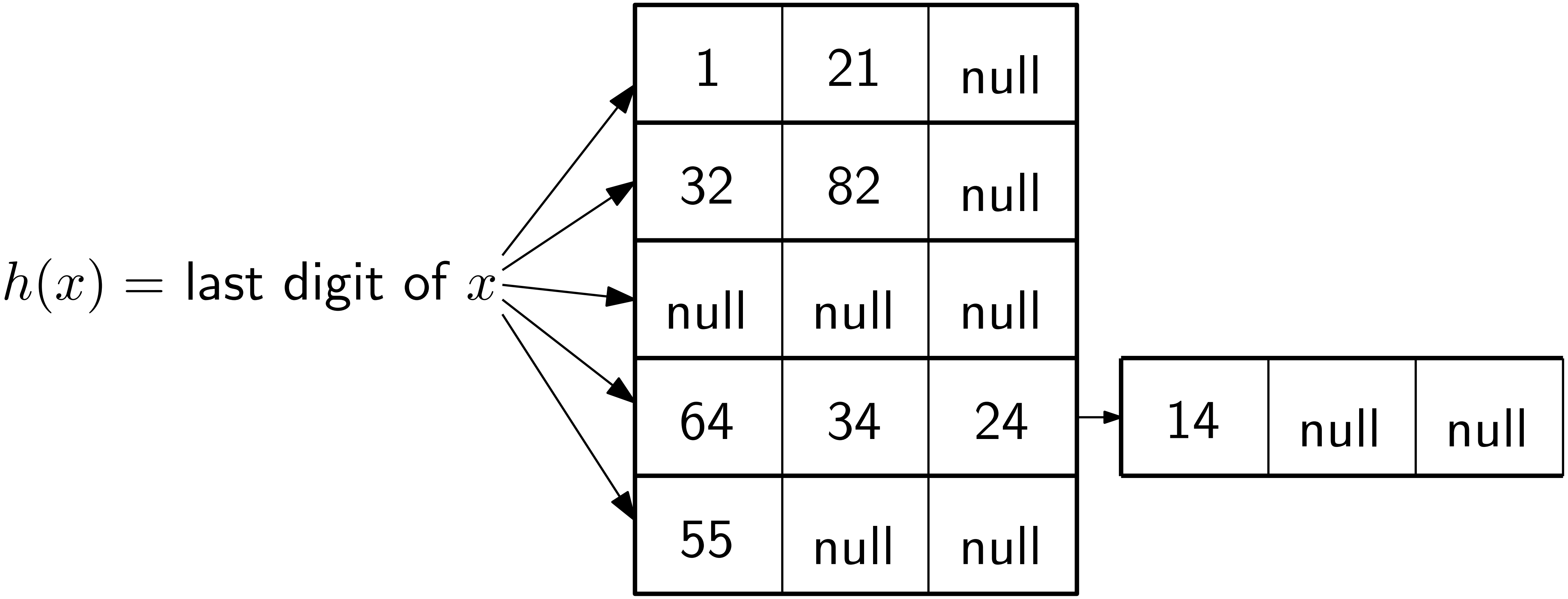
$h(x) =$ last digit of x

1	21	null
32	82	null
null	null	null
64	34	24
55	null	null

14	null	null

Ideal hash function assumption: h maps each object to a hash value uniformly independently at random

External Hashing



$h(x) =$ last digit of x

1	21	null
32	82	null
null	null	null
64	34	24
55	null	null

14	null	null

Ideal hash function assumption: h maps each object to a hash value uniformly independently at random

Expected average cost of a successful (or unsuccessful) lookup is $1 + 1/2^{\Omega(B)}$ disk accesses, provided the load factor is less than a constant smaller than 1 [Knuth, 1973]

Exact Numbers Calculated by Knuth

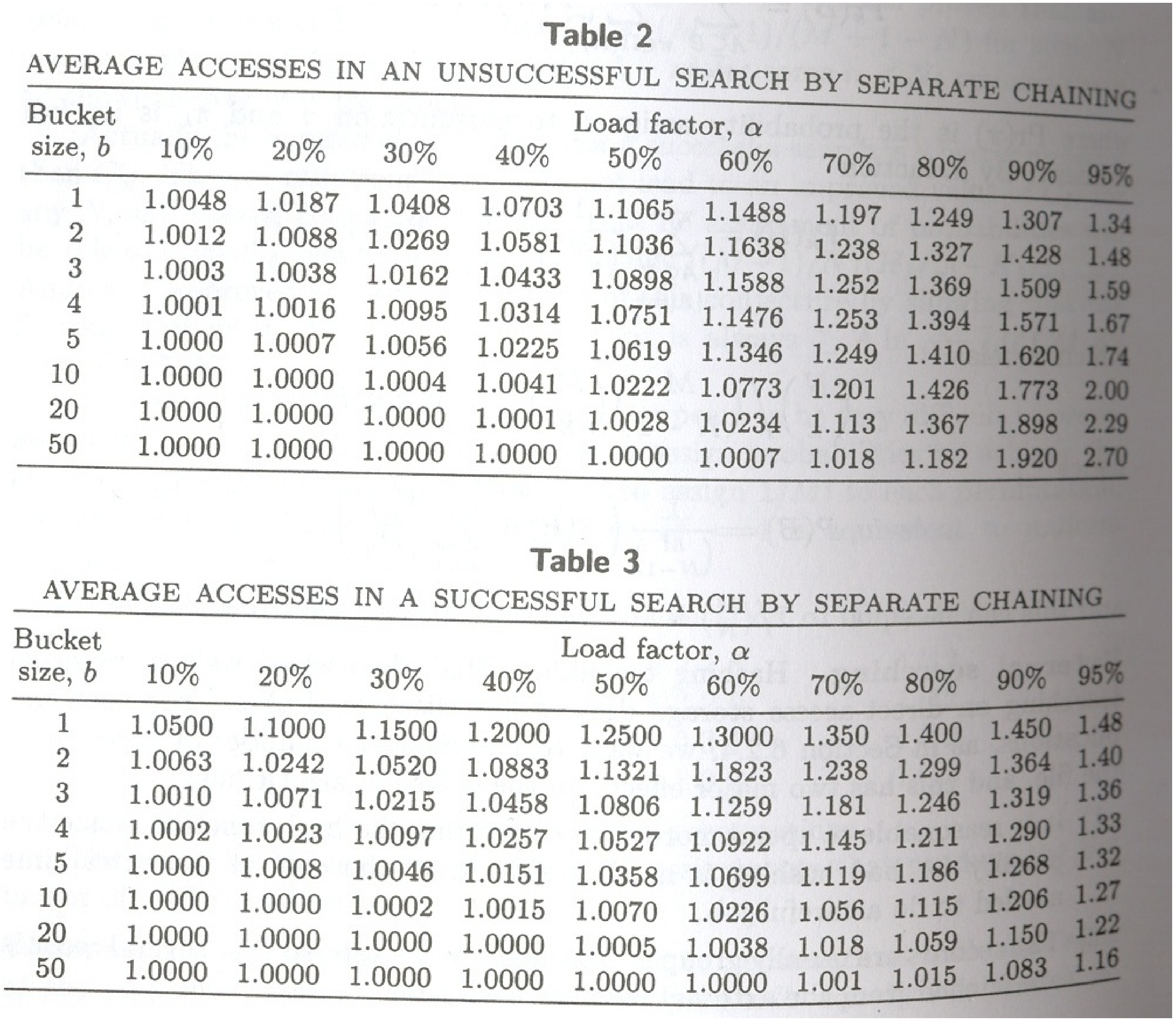


Table 2

AVERAGE ACCESSES IN AN UNSUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket size, b	Load factor, α 10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0048	1.0187	1.0408	1.0703	1.1065	1.1488	1.197	1.249	1.307	1.34
2	1.0012	1.0088	1.0269	1.0581	1.1036	1.1638	1.238	1.327	1.428	1.48
3	1.0003	1.0038	1.0162	1.0433	1.0898	1.1588	1.252	1.369	1.509	1.59
4	1.0001	1.0016	1.0095	1.0314	1.0751	1.1476	1.253	1.394	1.571	1.67
5	1.0000	1.0007	1.0056	1.0225	1.0619	1.1346	1.249	1.410	1.620	1.74
10	1.0000	1.0000	1.0004	1.0041	1.0222	1.0773	1.201	1.426	1.773	2.00
20	1.0000	1.0000	1.0000	1.0001	1.0028	1.0234	1.113	1.367	1.898	2.29
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0007	1.018	1.182	1.920	2.70

Table 3

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket size, b	Load factor, α 10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0500	1.1000	1.1500	1.2000	1.2500	1.3000	1.350	1.400	1.450	1.48
2	1.0063	1.0242	1.0520	1.0883	1.1321	1.1823	1.238	1.299	1.364	1.40
3	1.0010	1.0071	1.0215	1.0458	1.0806	1.1259	1.181	1.246	1.319	1.36
4	1.0002	1.0023	1.0097	1.0257	1.0527	1.0922	1.145	1.211	1.290	1.33
5	1.0000	1.0008	1.0046	1.0151	1.0358	1.0699	1.119	1.186	1.268	1.32
10	1.0000	1.0000	1.0002	1.0015	1.0070	1.0226	1.056	1.115	1.206	1.27
20	1.0000	1.0000	1.0000	1.0000	1.0005	1.0038	1.018	1.059	1.150	1.22
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.015	1.083	1.16

The Art of Computer Programming, volume 3, 1998, page 542

Exact Numbers Calculated by Knuth

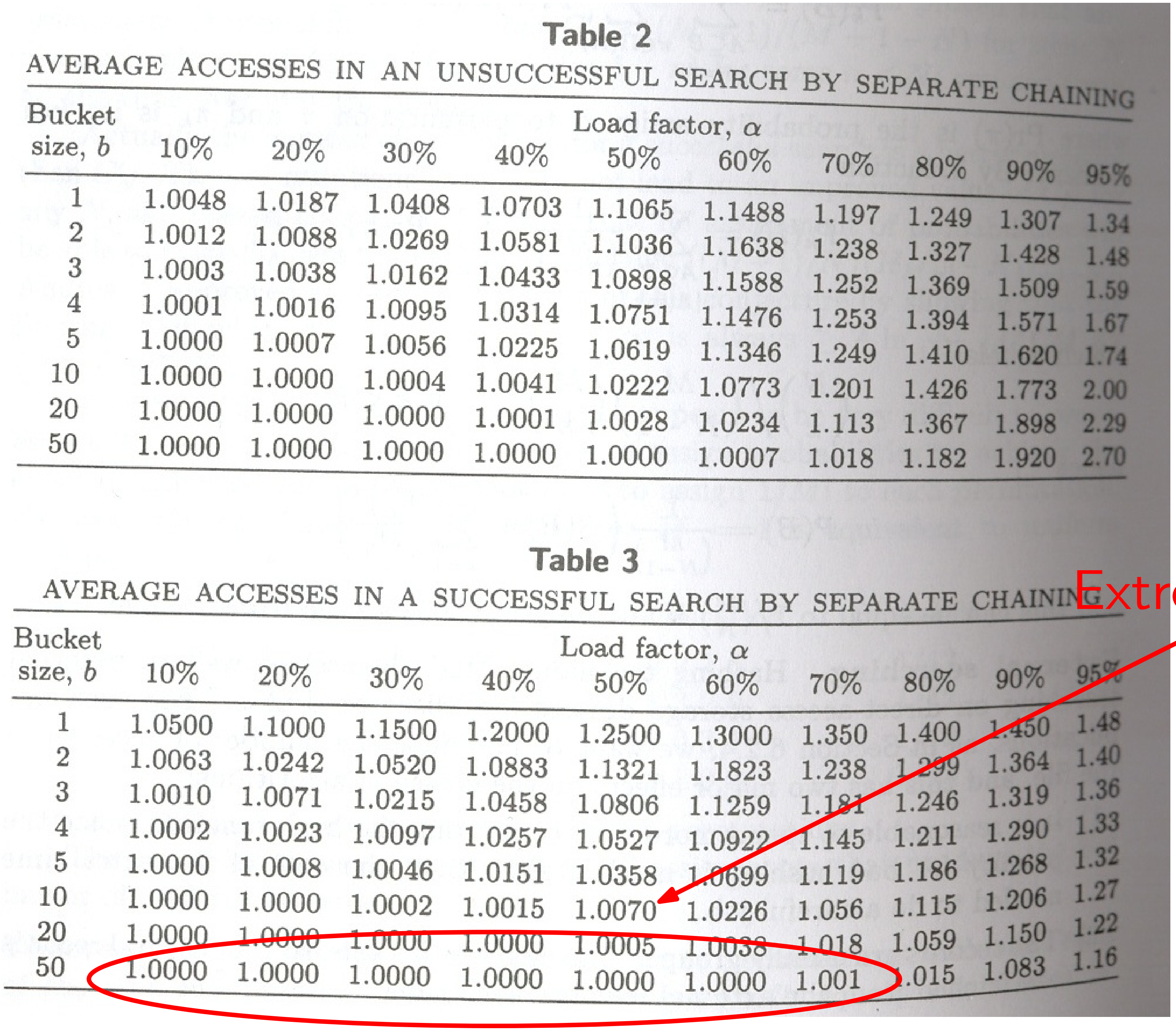


Table 2

AVERAGE ACCESSES IN AN UNSUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket size, b	Load factor, α 10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0048	1.0187	1.0408	1.0703	1.1065	1.1488	1.197	1.249	1.307	1.34
2	1.0012	1.0088	1.0269	1.0581	1.1036	1.1638	1.238	1.327	1.428	1.48
3	1.0003	1.0038	1.0162	1.0433	1.0898	1.1588	1.252	1.369	1.509	1.59
4	1.0001	1.0016	1.0095	1.0314	1.0751	1.1476	1.253	1.394	1.571	1.67
5	1.0000	1.0007	1.0056	1.0225	1.0619	1.1346	1.249	1.410	1.620	1.74
10	1.0000	1.0000	1.0004	1.0041	1.0222	1.0773	1.201	1.426	1.773	2.00
20	1.0000	1.0000	1.0000	1.0001	1.0028	1.0234	1.113	1.367	1.898	2.29
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0007	1.018	1.182	1.920	2.70

Table 3

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket size, b	Load factor, α 10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0500	1.1000	1.1500	1.2000	1.2500	1.3000	1.350	1.400	1.450	1.48
2	1.0063	1.0242	1.0520	1.0883	1.1321	1.1823	1.238	1.299	1.364	1.40
3	1.0010	1.0071	1.0215	1.0458	1.0806	1.1259	1.181	1.246	1.319	1.36
4	1.0002	1.0023	1.0097	1.0257	1.0527	1.0922	1.145	1.211	1.290	1.33
5	1.0000	1.0008	1.0046	1.0151	1.0358	1.0699	1.119	1.186	1.268	1.32
10	1.0000	1.0000	1.0002	1.0015	1.0070	1.0226	1.056	1.115	1.206	1.27
20	1.0000	1.0000	1.0000	1.0000	1.0005	1.0038	1.018	1.059	1.150	1.22
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.015	1.083	1.16

Extremely close to ideal

The Art of Computer Programming, volume 3, 1998, page 542

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

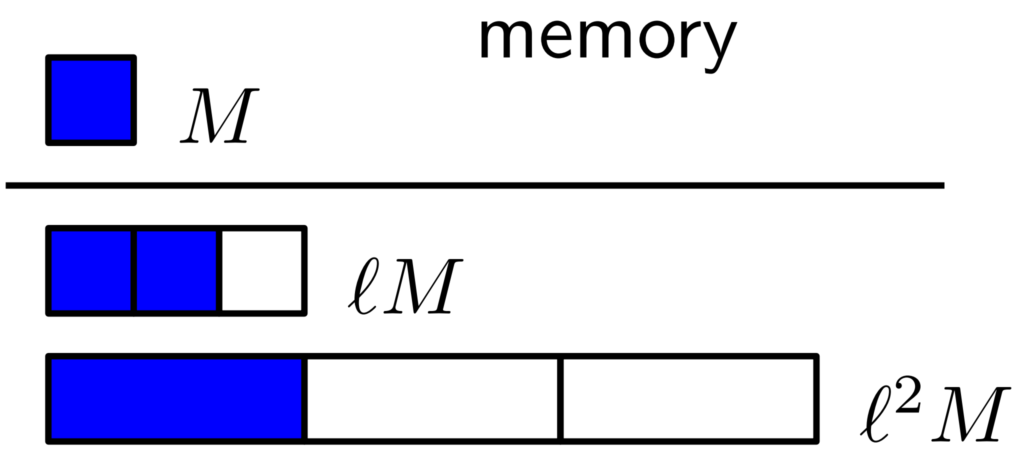
- Focus on insertions first: Both the B-tree and hash table do a search first, then insert into the appropriate block
  - B-tree: Split blocks when necessary
  - Hashing: Rebuild the hash table when too full; *extensible hashing* [Fagin, Nievergelt, Pippenger, Strong, 79]; *linear hashing* [Litwin, 80]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

Dynamic Hash Tables

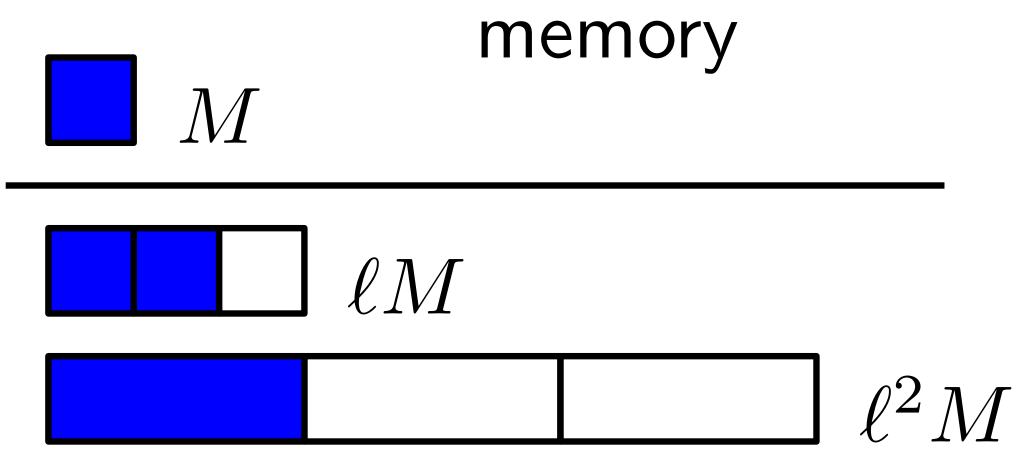
Dynamic B-trees for Fast Insertions

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

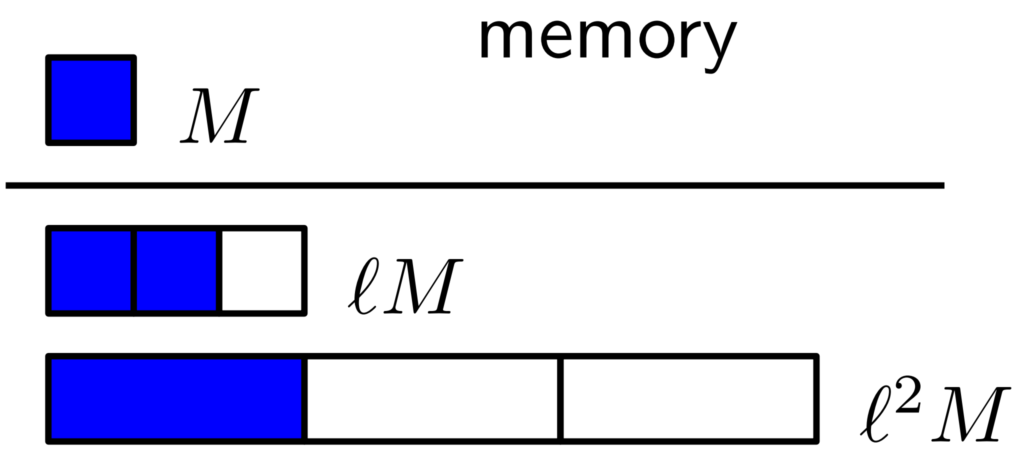


Dynamic B-trees for Fast Insertions

- *LSM-tree* [O'Neil, Cheng, Gawlick, O'Neil, Acta Informatica'96]: Logarithmic method + B-tree
  - Insertion: $O(\frac{\ell}{B}\log_\ell\frac{N}{M})$
  - Query: $O(\log_\ell\frac{N}{M})$ (omit the $\frac{K}{B}$ output term)

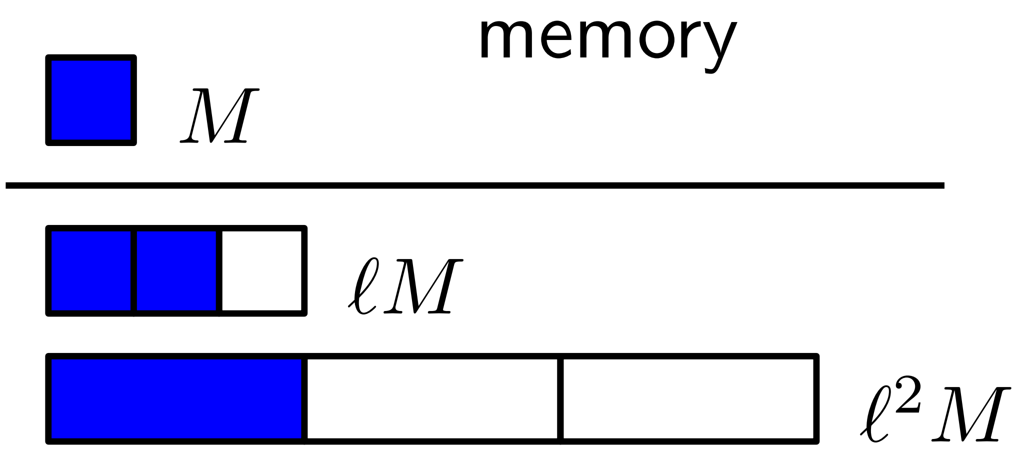


Dynamic B-trees for Fast Insertions



- *LSM-tree* [O'Neil, Cheng, Gawlick, O'Neil, Acta Informatica'96]: Logarithmic method + B-tree
  - Insertion: $O(\frac{\ell}{B} \log_\ell \frac{N}{M})$
  - Query: $O(\log_\ell \frac{N}{M})$ (omit the $\frac{K}{B}$ output term)
- *Stepped merge tree* [Jagadish, Narayan, Seshadri, Sudarshan, Kannegantil, VLDB'97]: variant of LSM-tree
  - Insertion: $O(\frac{1}{B} \log_\ell \frac{N}{M})$
  - Query: $O(\ell \log_\ell \frac{N}{M})$

Dynamic B-trees for Fast Insertions



- *LSM-tree* [O'Neil, Cheng, Gawlick, O'Neil, Acta Informatica'96]: Logarithmic method + B-tree
  - Insertion: $O(\frac{\ell}{B} \log_\ell \frac{N}{M})$
  - Query: $O(\log_\ell \frac{N}{M})$ (omit the $\frac{K}{B}$ output term)
- *Stepped merge tree* [Jagadish, Narayan, Seshadri, Sudarshan, Kannegantil, VLDB'97]: variant of LSM-tree
  - Insertion: $O(\frac{1}{B} \log_\ell \frac{N}{M})$
  - Query: $O(\ell \log_\ell \frac{N}{M})$
- Usually ℓ is set to be a constant, then they both have $O(\frac{1}{B} \log \frac{N}{M})$ insertion and $O(\log \frac{N}{M})$ query

More Dynamic B-trees

- Buffer-tree (buffered-repository tree) [Arge, WADS'95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA'00]
- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA'07]
- *Y-tree* [Jermaine, Datta, Omiecinski, VLDB'99]

More Dynamic B-trees

- Buffer-tree (buffered-repository tree) [Arge, WADS'95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA'00]
- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA'07]
- *Y-tree* [Jermaine, Datta, Omiecinski, VLDB'99]

q	u
$\log B$	$\frac{1}{B}\log B$
1	$\frac{1}{B}B^{\epsilon}$
B^{ϵ}	$\frac{1}{B}$

More Dynamic B-trees

- Buffer-tree (buffered-repository tree) [Arge, WADS'95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA'00]
- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA'07]
- *Y-tree* [Jermaine, Datta, Omiecinski, VLDB'99]

q	u
$\log B$	$\frac{1}{B}\log B$
1	$\frac{1}{B}B^{\epsilon}$
B^{ϵ}	$\frac{1}{B}$

- Deletions? Standard trick: inserting "delete signals"

More Dynamic B-trees

- Buffer-tree (buffered-repository tree) [Arge, WADS'95; Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook, SODA'00]
- Streaming B-tree [Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson, SPAA'07]
- *Y-tree* [Jermaine, Datta, Omiecinski, VLDB'99]
- Cache-oblivious model [Demaine, Fineman, Iacono, Langerman, Munro, SODA'10]

q	u
$\log B$	$\frac{1}{B}\log B$
1	$\frac{1}{B}B^{\epsilon}$
B^{ϵ}	$\frac{1}{B}$

  - Deletions? Standard trick: inserting "delete signals"
- No better solutions known ...

Compare with the rich results in RAM!

- Range reporting
  - $O(\sqrt{\log N / \log\log N})$ insertion and query [Andersson, Thorup, JACM'07]
  - $O(\log N / \log\log N)$ insertion and $O(\log\log N)$ query [Mortensen, Pagh, Pătrașcu, STOC'05]
  - Other results that depend on the word size w
- Predecessor
  - $\Theta(\sqrt{\log N / \log\log N})$ insertion and query [Andersson, Thorup, JACM'07]
- Partial-sum
  - $\Theta(\log N)$ insertion query [Pătrașcu, Demaine, SODA'04]

Are the EM and DB people just dumb?

Our Main Result

For any dynamic range query index with a query cost of q and an amortized insertion cost of u, the following tradeoff holds

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

Our Main Result

For any dynamic range query index with a query cost of q and an amortized insertion cost of u, the following tradeoff holds

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

Assuming $\log_B \frac{N}{M} = O(1)$, all the bounds are tight!

Current upper bounds:

q	u
$\log B$	$\frac{1}{B} \log B$
1	$\frac{1}{B} B^{\epsilon}$
B^{ϵ}	$\frac{1}{B}$

Our Main Result

For any dynamic range query index with a query cost of q and an amortized insertion cost of u, the following tradeoff holds

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

Assuming $\log_B \frac{N}{M} = O(1)$, all the bounds are tight!

Current upper bounds:

$\log \frac{N}{M}$

q	u
$\log B$	$\frac{1}{B} \log B$
1	$\frac{1}{B} B^{\epsilon}$
B^{ϵ}	$\frac{1}{B}$

$\frac{1}{B} \log \frac{N}{M}$

Can't be true for $B = o(\sqrt{\log n \log\log n})$, since the *exponential tree* achieves $u = q = O(\sqrt{\log n / \log\log n})$ [Andersson, Thorup, JACM'07]. ($n = N/M$)

The real question

How large does B need to be for buffer-tree to be optimal for range reporting?

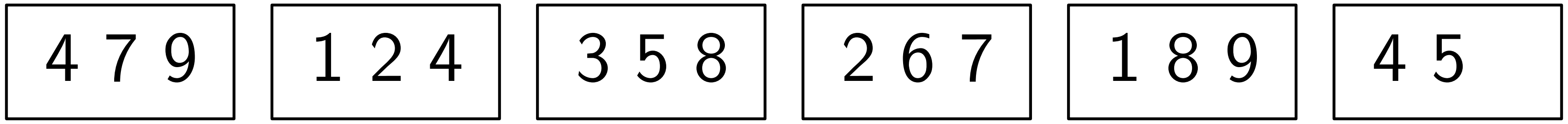
Known: somewhere between $\Omega(\sqrt{\log n \log\log n})$ and $O(n^{\epsilon})$

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, PODS’97, JACM’02]

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, PODS'97, JACM'02]



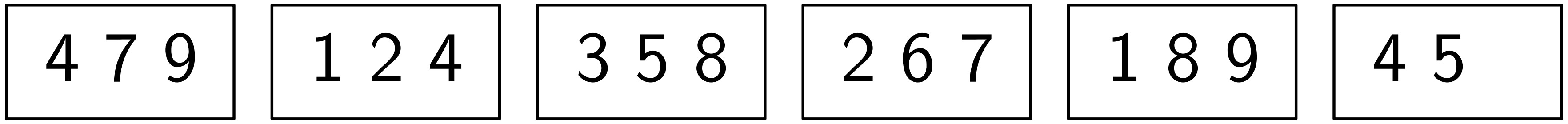
4 7 9 | 1 2 4 | 3 5 8 | 2 6 7 | 1 8 9 | 4 5

- Objects are stored in disk blocks of size up to B, possibly with redundancy.

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, PODS'97, JACM'02]

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

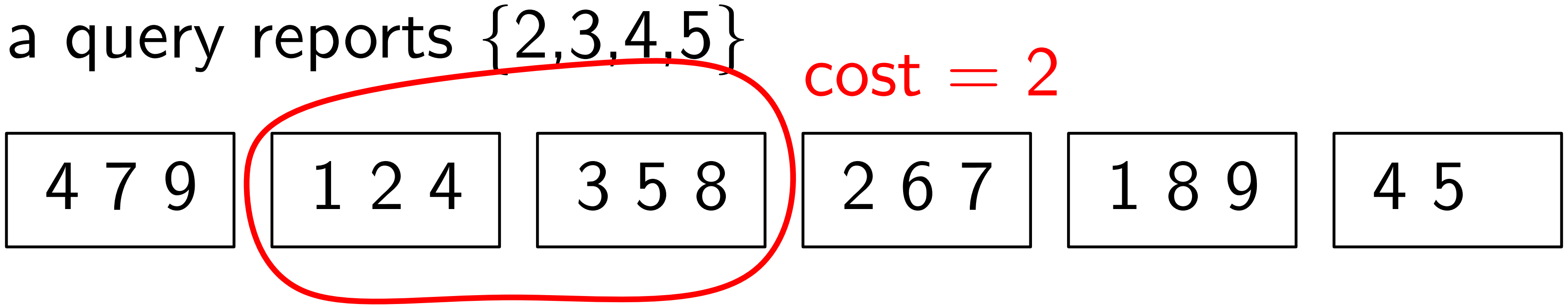


4 7 9 | 1 2 4 | 3 5 8 | 2 6 7 | 1 8 9 | 4 5

- Objects are stored in disk blocks of size up to B, possibly with redundancy.

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, PODS'97, JACM'02]



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

cost = 2

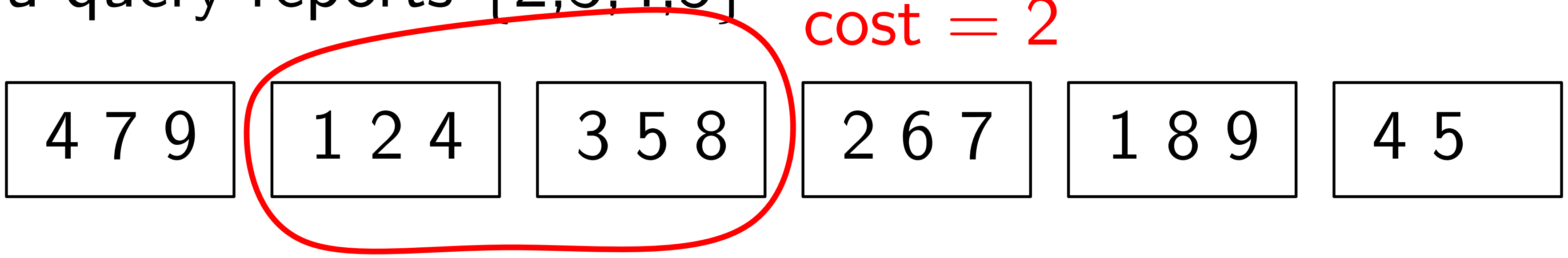
4 7 9	1 2 4	3 5 8	2 6 7	1 8 9	4 5

- Objects are stored in disk blocks of size up to B, possibly with redundancy.
- The query cost is the minimum number of blocks that can cover all the required results (search time ignored!).

Lower Bound Model: Dynamic Indexability

- *Indexability*: [Hellerstein, Koutsoupias, Papadimitriou, PODS'97, JACM'02]

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

  cost $= 2$

  - Objects are stored in disk blocks of size up to B, possibly with redundancy.
  - The query cost is the minimum number of blocks that can cover all the required results (search time ignored!).

- Similar in spirit to popular lower bound models: cell probe model, semigroup model

Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
  - Tradeoff between space (s) and query time (q)

Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
  - Tradeoff between space (s) and query time (q)
- 2D range queries: $s/N \cdot \log q = \Omega(\log(N/B))$ [Hellerstein, Koutsoupias, Papadimitriou, PODS'97], [Koutsoupias, Taylor, PODS'98], [Arge, Samoladas, Vitter, PODS'99]

Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
    - Tradeoff between space (s) and query time (q)
- 2D range queries: $s/N \cdot \log q = \Omega(\log(N/B))$ [Hellerstein, Koutsoupias, Papadimitriou, PODS'97], [Koutsoupias, Taylor, PODS'98], [Arge, Samoladas, Vitter, PODS'99]
- 2D stabbing queries: $q \cdot \log(s/N) = \Omega(\log(N/B))$ [Arge, Samoladas, Yi, ESA'04, Algorithmica'99]

Previous Results on Static Indexability

- Nearly all external indexing lower bounds are under this model
    - Tradeoff between space (s) and query time (q)
- 2D range queries: $s/N \cdot \log q = \Omega(\log(N/B))$ [Hellerstein, Koutsoupias, Papadimitriou, PODS'97], [Koutsoupias, Taylor, PODS'98], [Arge, Samoladas, Vitter, PODS'99]
- 2D stabbing queries: $q \cdot \log(s/N) = \Omega(\log(N/B))$ [Arge, Samoladas, Yi, ESA'04, Algorithmica'99]
- 1D range queries: $s = N, q = 1$ trivially

Previous Results on Static Indexability

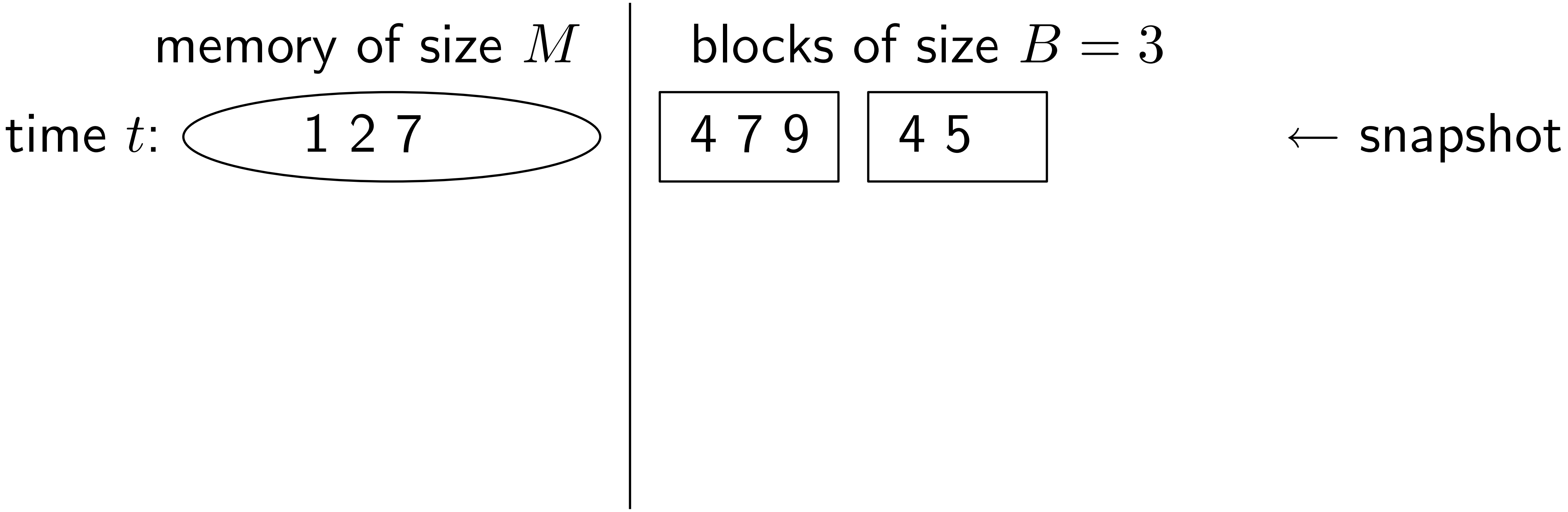
- Nearly all external indexing lower bounds are under this model
  - Tradeoff between space (s) and query time (q)
- 2D range queries: $s/N \cdot \log q = \Omega(\log(N/B))$ [Hellerstein, Koutsoupias, Papadimitriou, PODS'97], [Koutsoupias, Taylor, PODS'98], [Arge, Samoladas, Vitter, PODS'99]
- 2D stabbing queries: $q \cdot \log(s/N) = \Omega(\log(N/B))$ [Arge, Samoladas, Yi, ESA'04, Algorithmica'99]
- 1D range queries: $s = N, q = 1$ 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

← snapshot

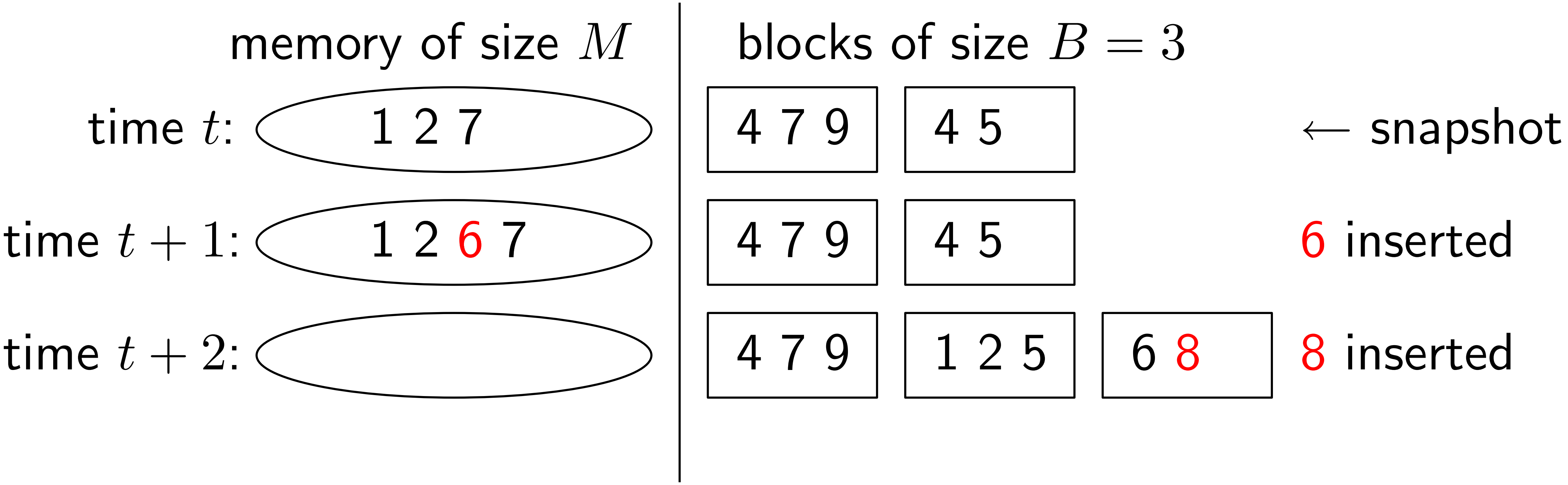
Dynamic Indexability

- 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

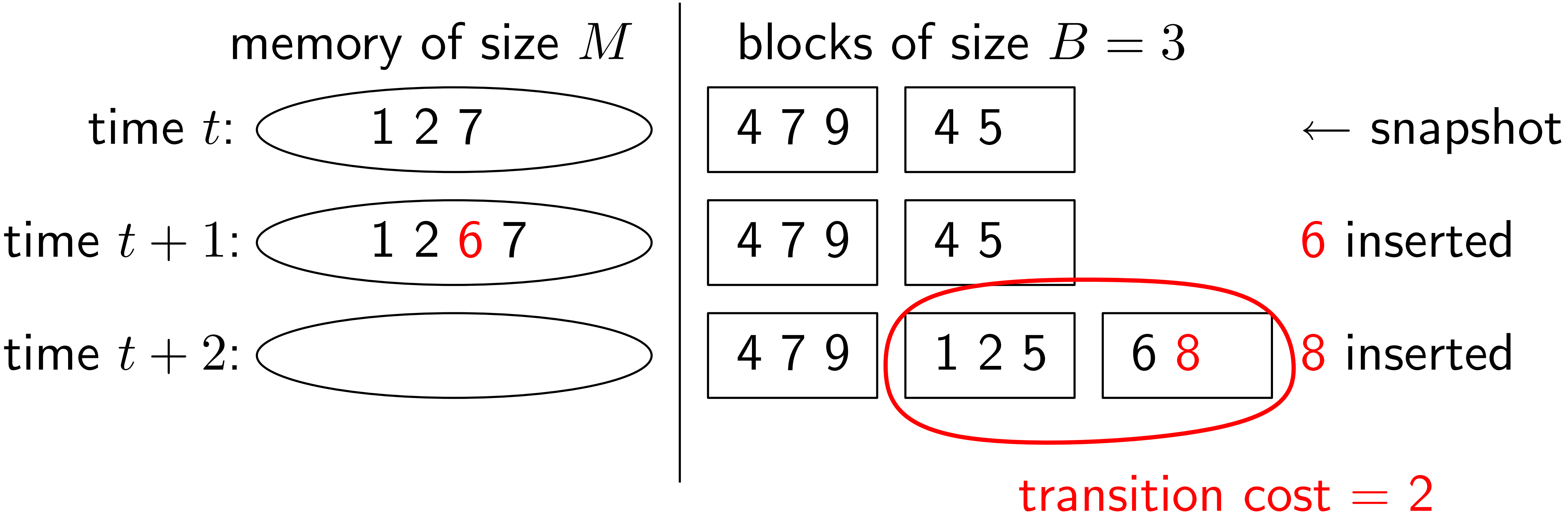
- Still consider only insertions



	memory of size M	blocks of size $B = 3$	
time t:	1 2 7	4 7 9 4 5	← 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



memory of size M

blocks of size $B = 3$

time t: 1 2 7 | 4 7 9 | 4 5 | $\leftarrow$ snapshot

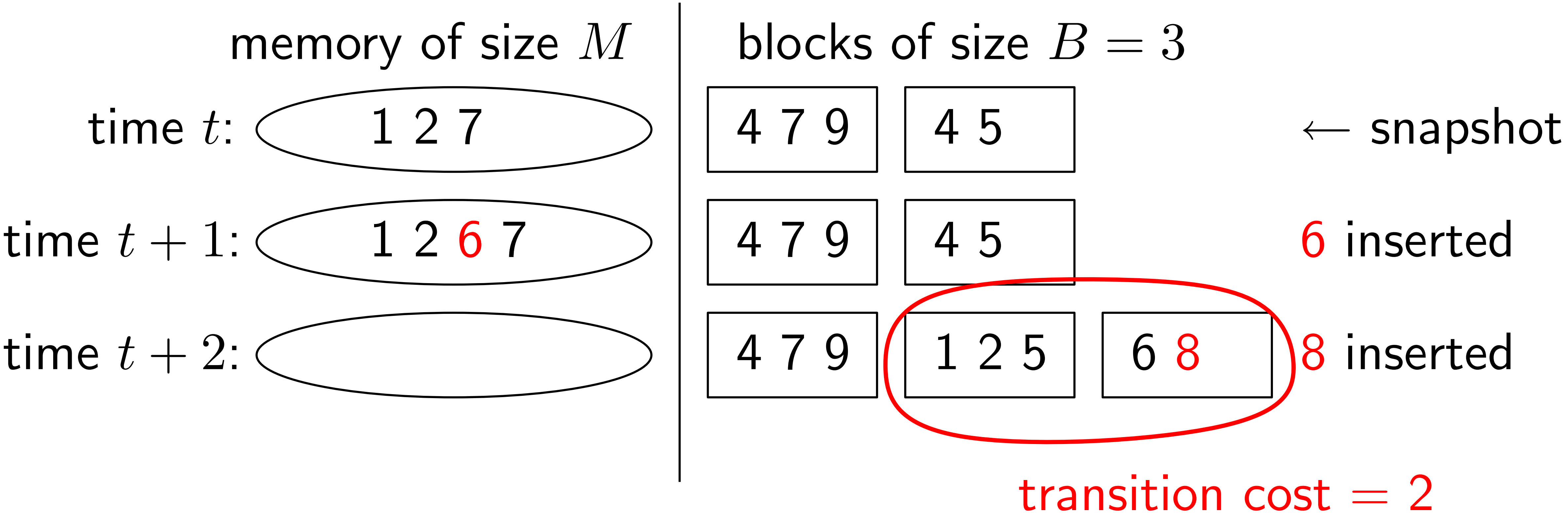
time $t+1$: 1 2 6 7 | 4 7 9 | 4 5 | 6 inserted

time $t+2$: | 4 7 9 | 1 2 5 | 6 8 | 8 inserted

transition cost $= 2$

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

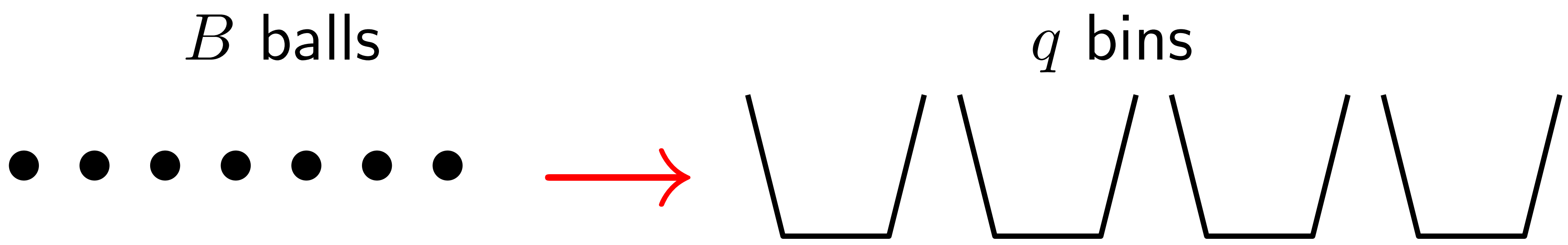
time $t + 1$: 1 2 6 7 | 4 7 9 | 4 5 6 inserted

time $t + 2$: | 4 7 9 | 1 2 5 | 6 8 8 inserted

transition cost = 2

- Update cost: u = amortized transition cost per insertion

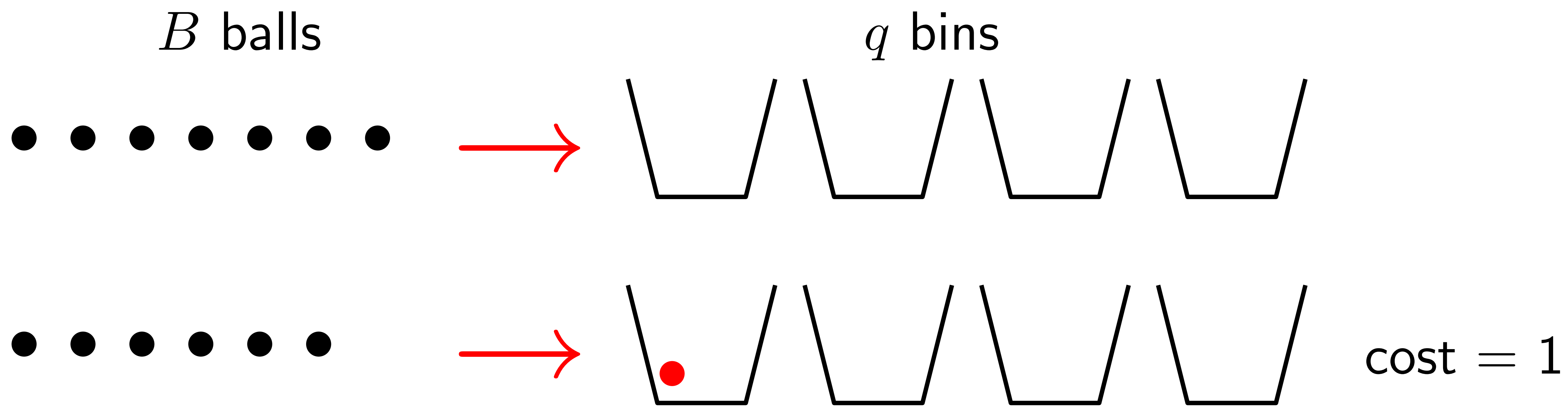
The Ball-Shuffling Problem



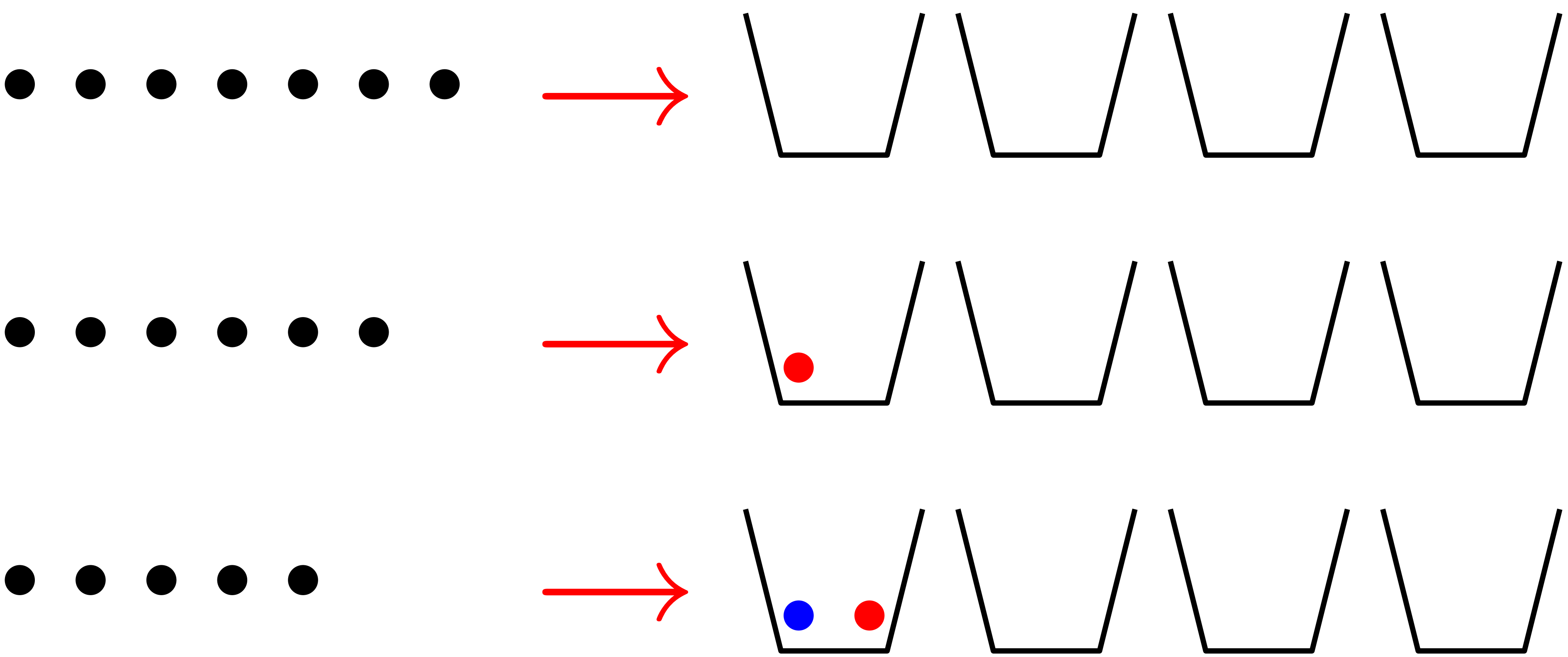
B balls

q bins

The Ball-Shuffling Problem



The Ball-Shuffling Problem



B balls

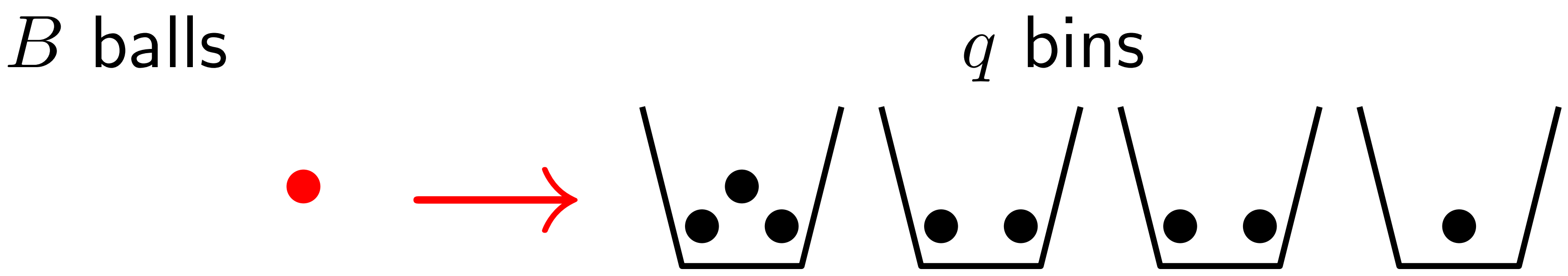
q bins

cost $= 1$

cost $= 2$

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

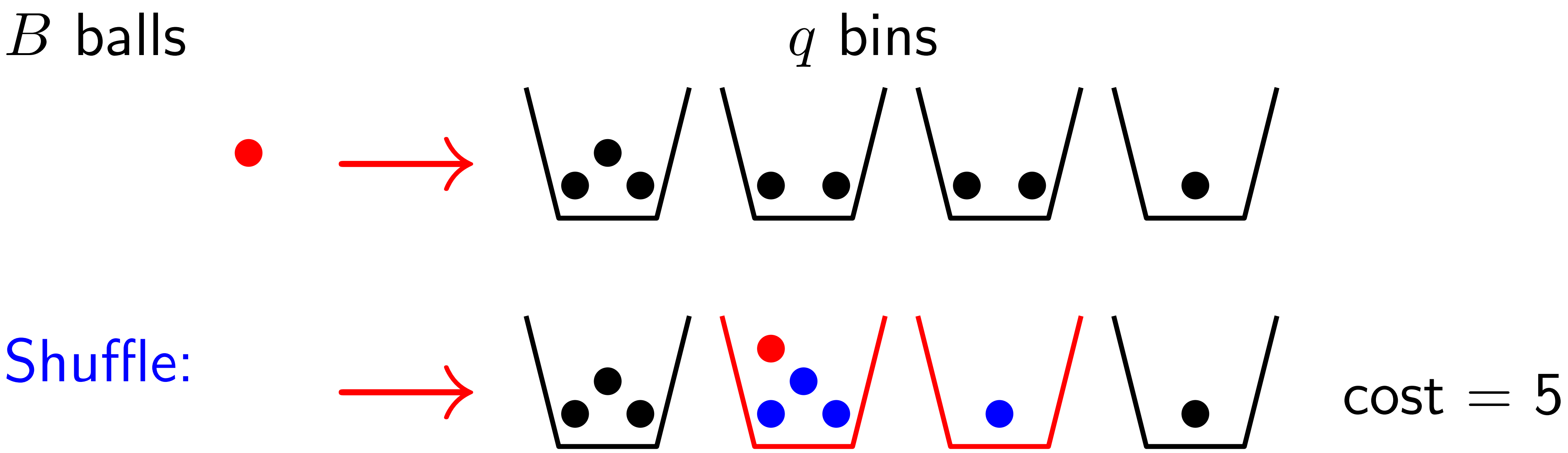
The Ball-Shuffling Problem



B balls

q bins

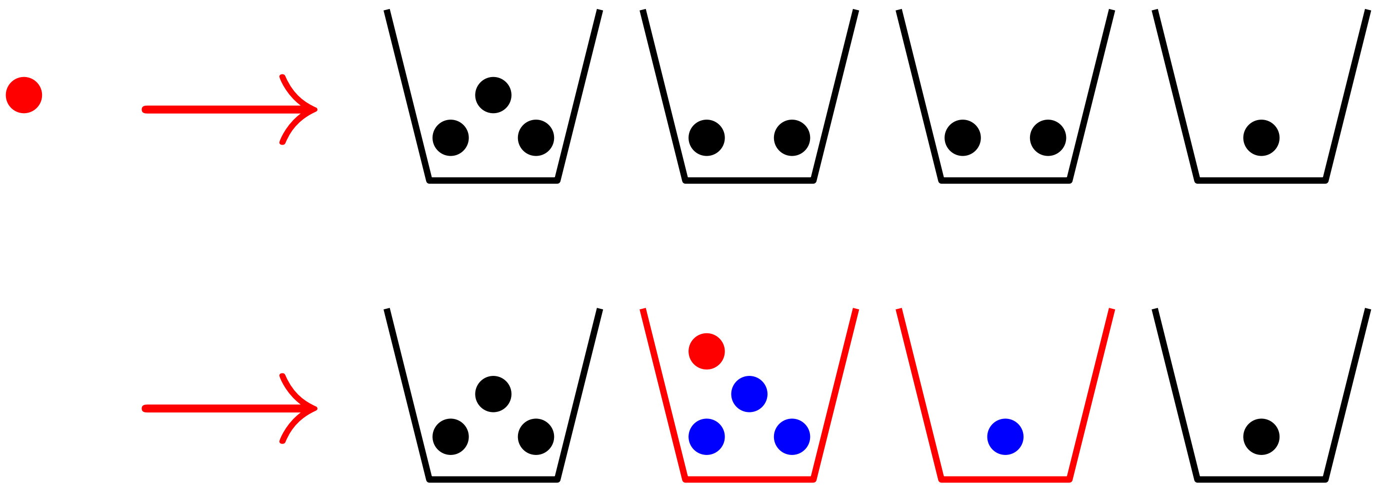
The Ball-Shuffling Problem



The Ball-Shuffling Problem

B balls

q bins

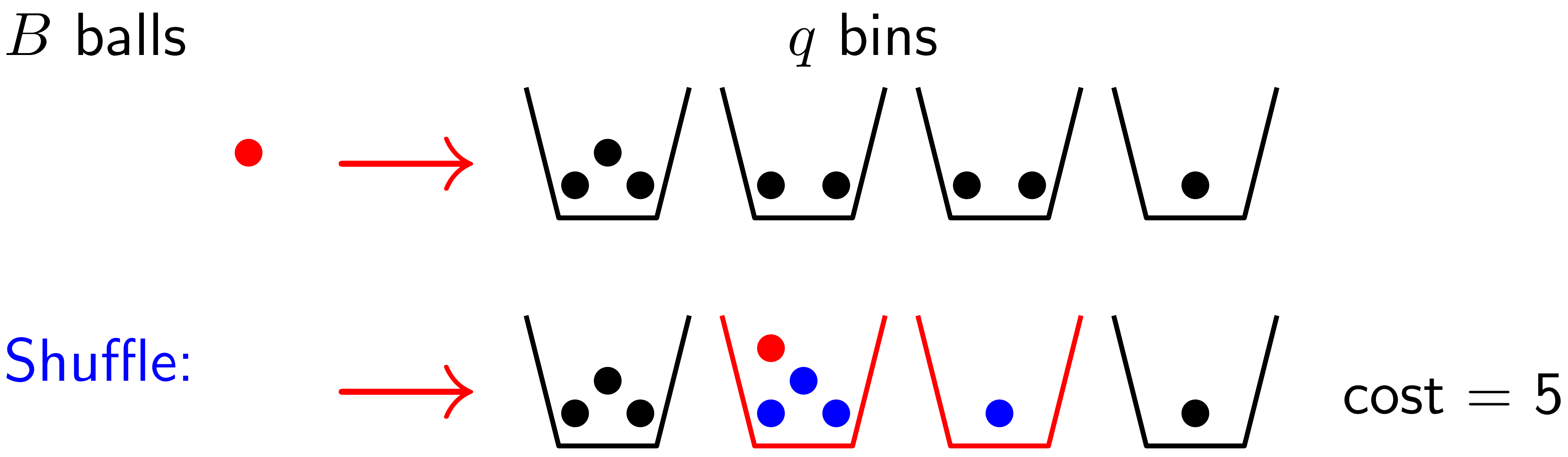


Shuffle:

cost $= 5$

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

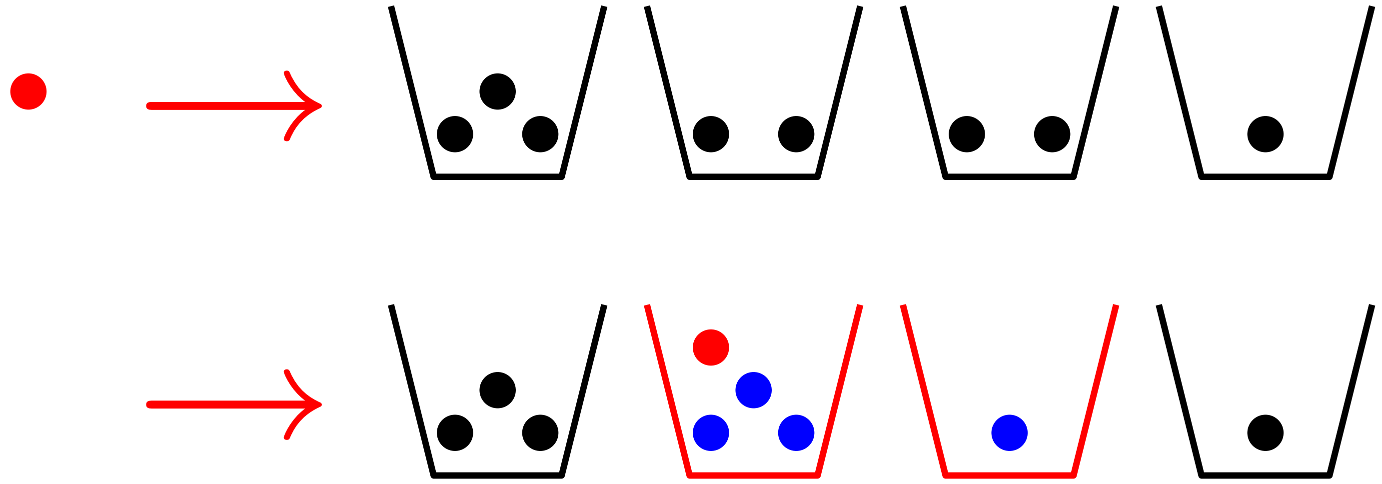
The Ball-Shuffling Problem



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

q bins

Shuffle:

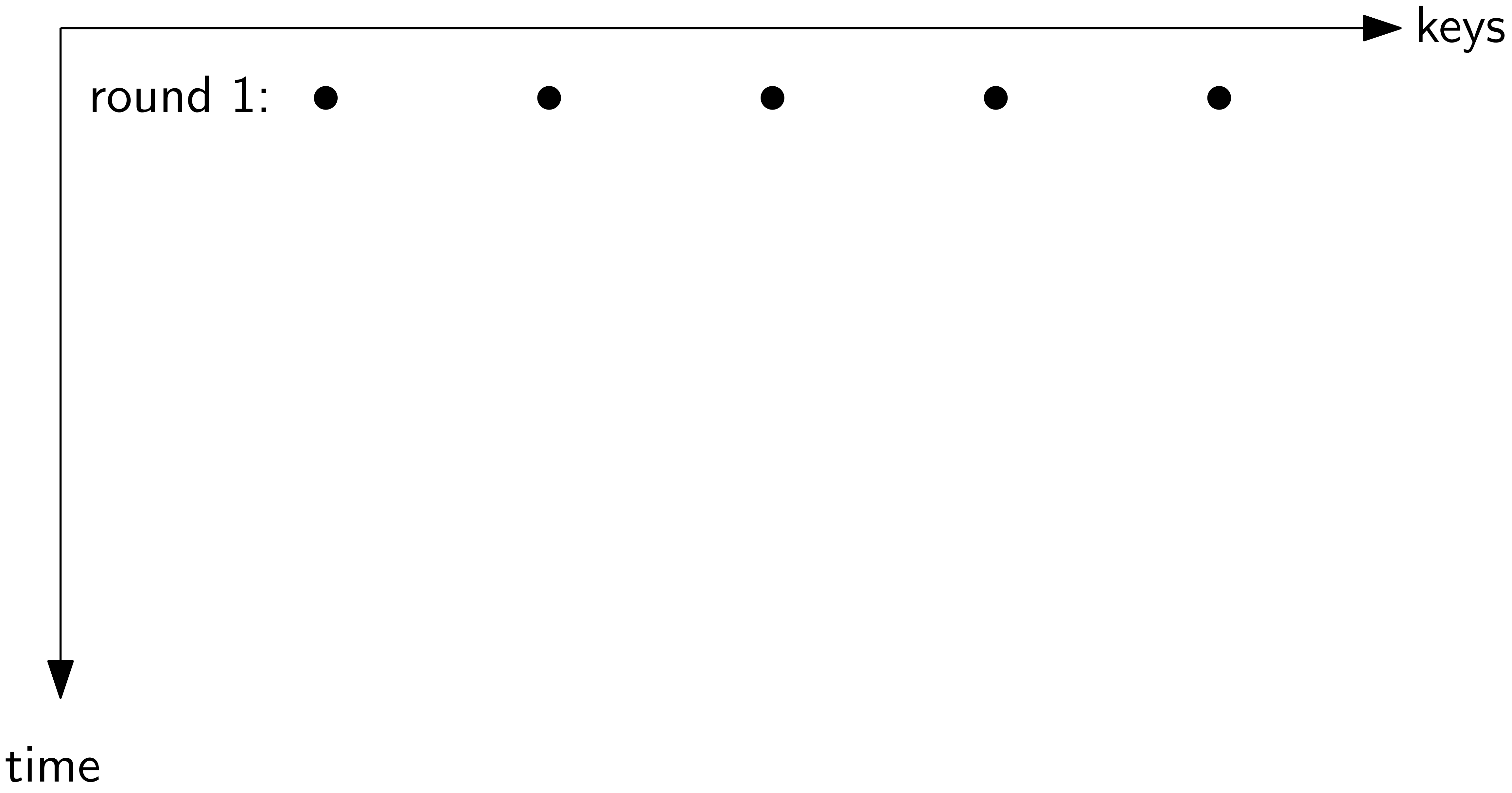
cost $= 5$

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

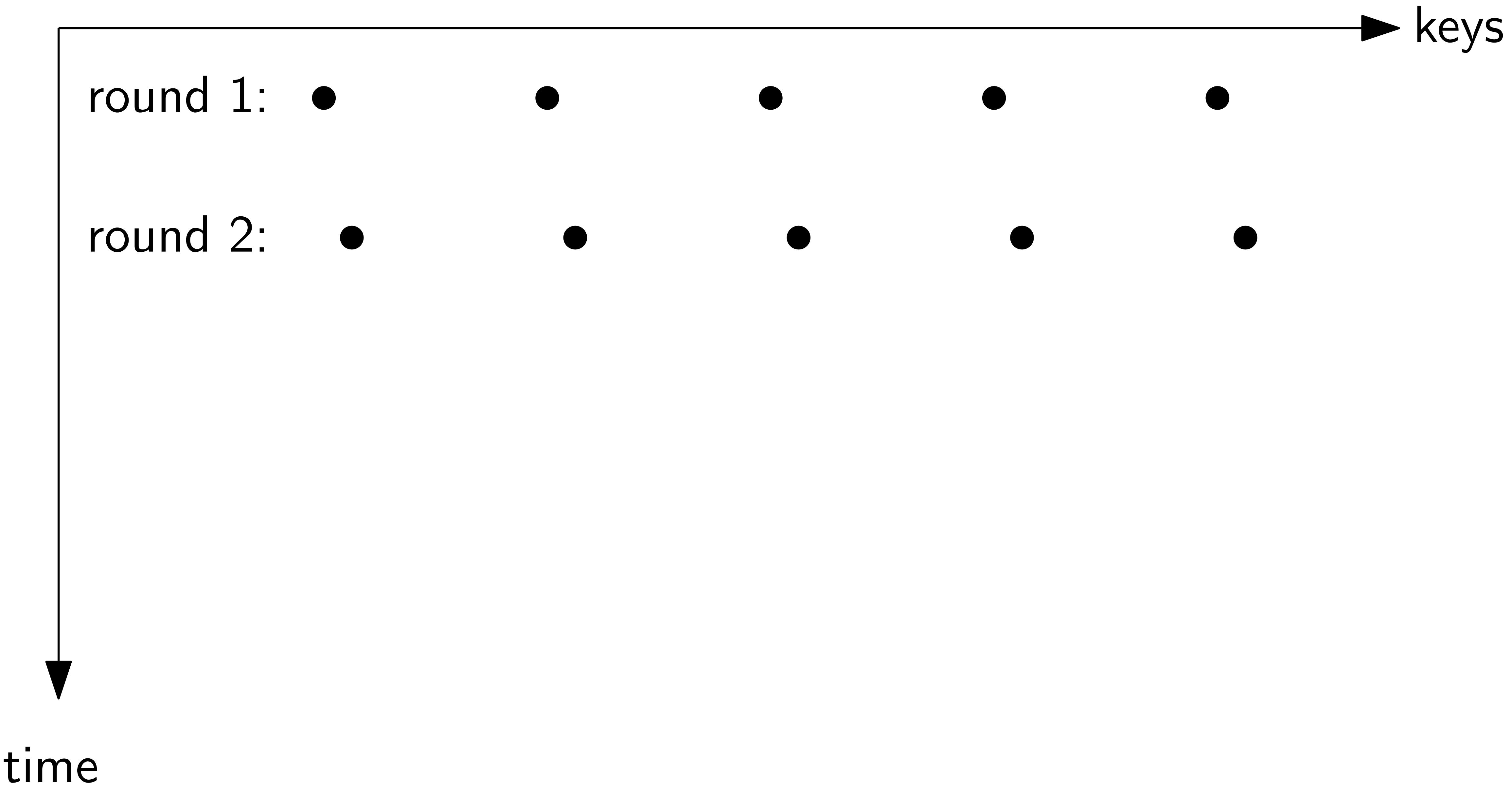
Putting a ball directly into a bin is a special shuffle

Goal: Accommodating all B balls using q bins with minimum cost

The Workload Construction



The Workload Construction



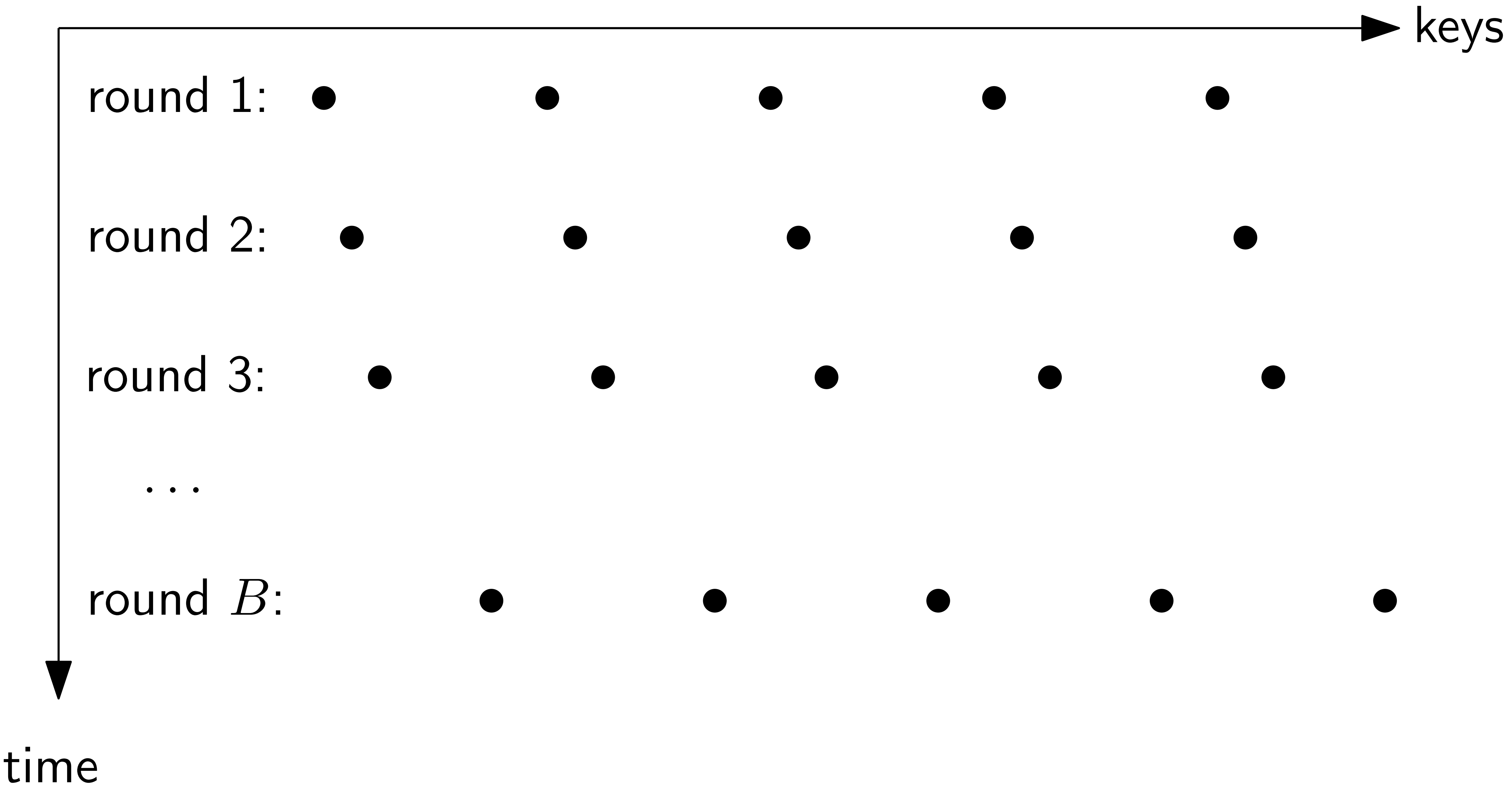
keys

round 1:

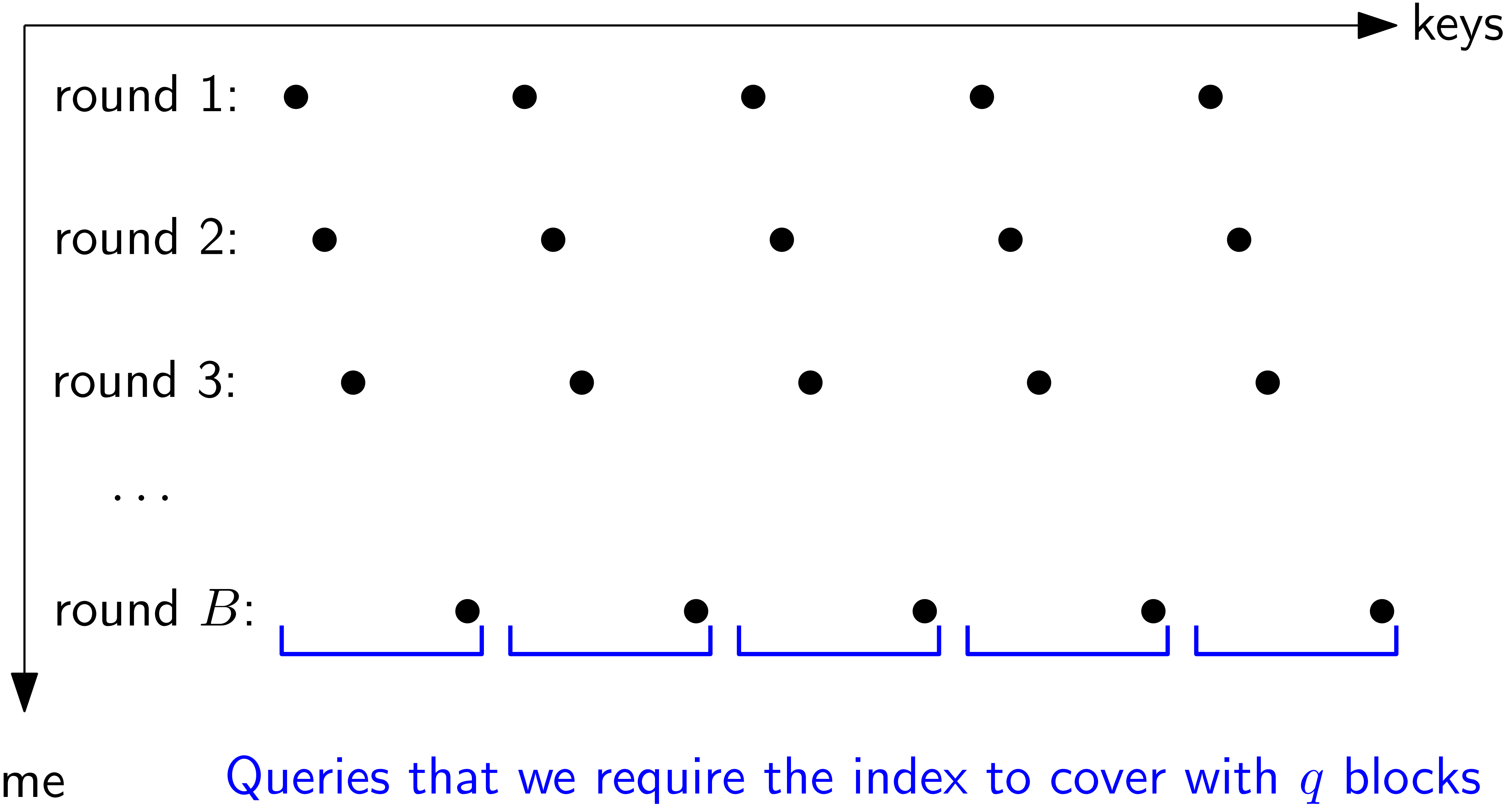
round 2:

time

The Workload Construction



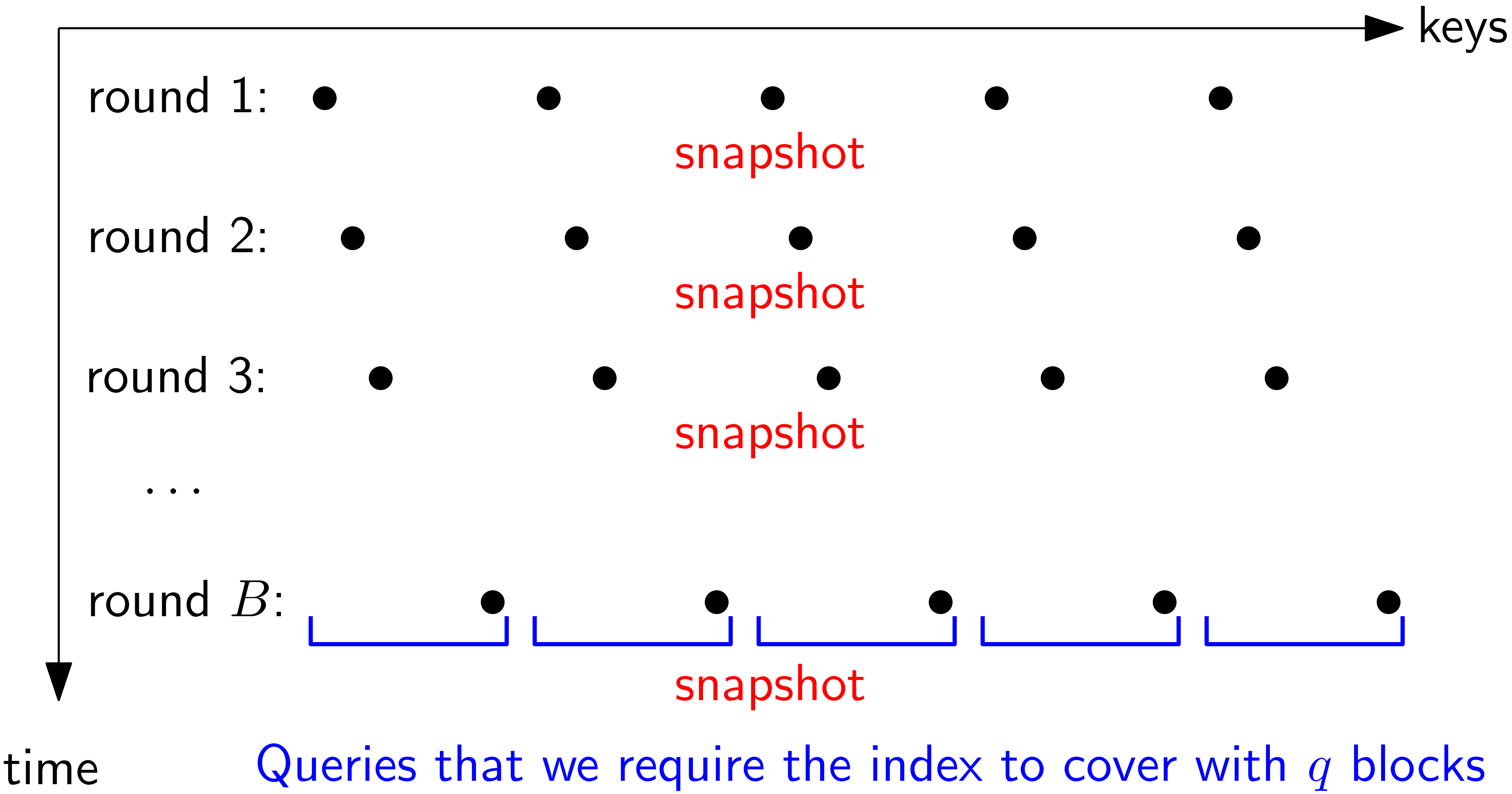
The Workload Construction



Queries that we require the index to cover with q blocks

\# queries $\geq 2MB$

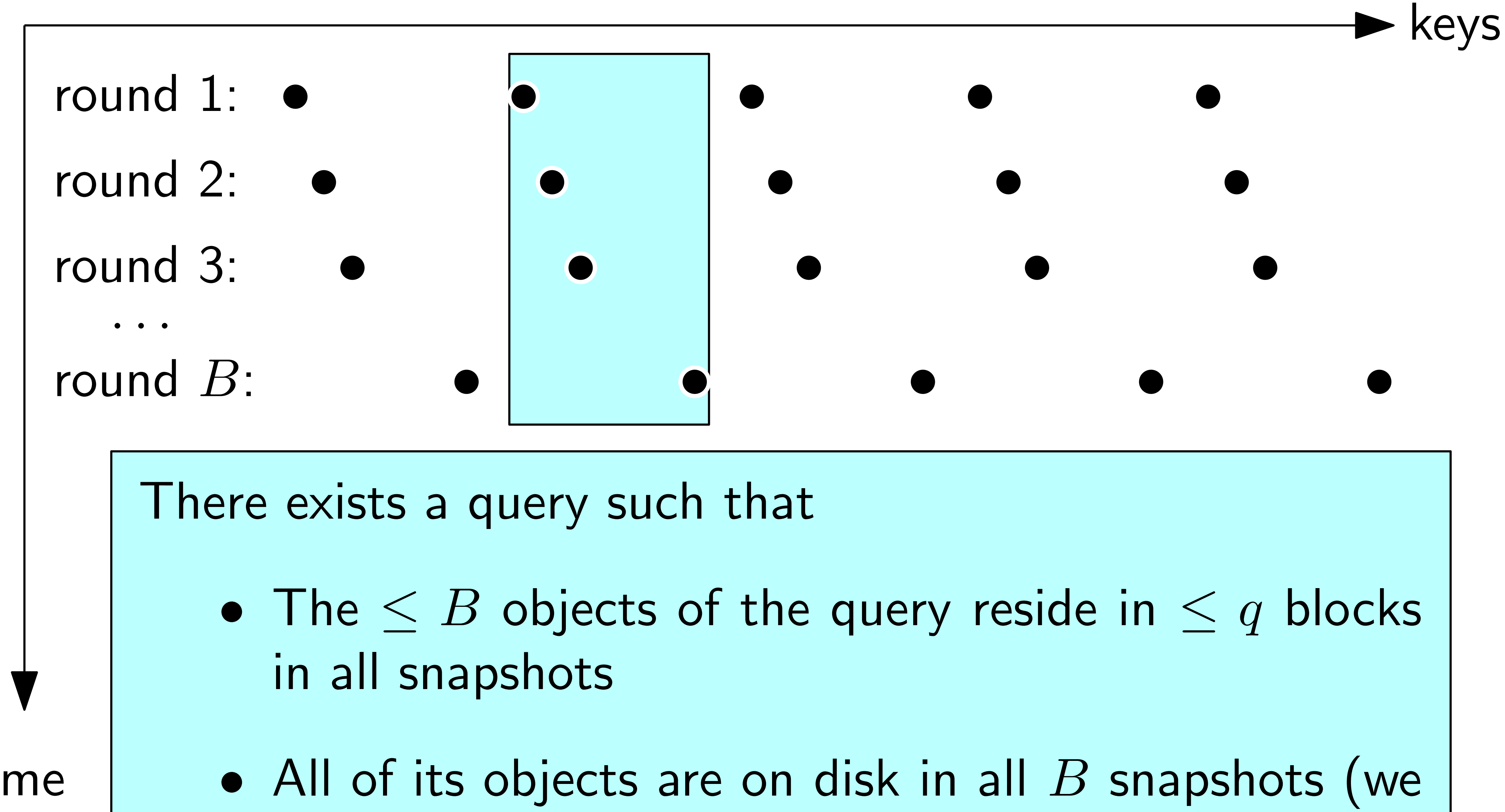
The Workload Construction



Queries that we require the index to cover with q 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 $\leq q$ 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^2 times in total

The Reduction

An index with update cost u and query A gives us a solution to the ball-shuffling game with cost uB^2 for B balls and q bins

The Reduction

An index with update cost u and query A gives us a solution to the ball-shuffling game with cost uB^2 for B balls and q bins

Lower bound on the ball-shuffling problem:

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

$$\begin{cases} \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\ \Omega(B \log_q B), & \text{for any } q. \end{cases}$$

The Reduction

An index with update cost u and query A gives us a solution to the ball-shuffling game with cost uB^2 for B balls and q bins

Lower bound on the ball-shuffling problem:

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

$$\begin{cases} \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\ \Omega(B \log_q B), & \text{for any } q. \end{cases}$$

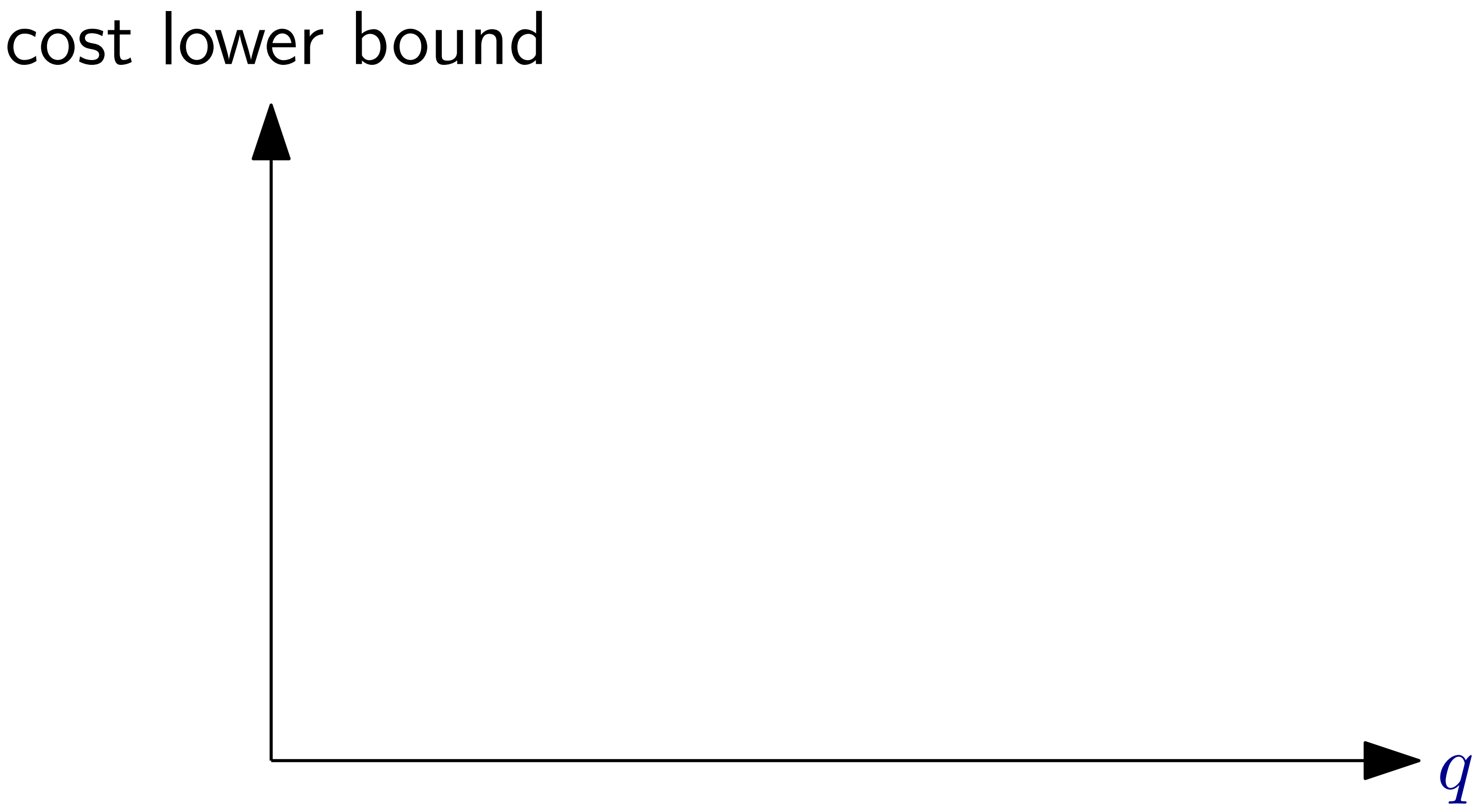
$$\Downarrow$$

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

Ball-Shuffling Lower Bounds

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

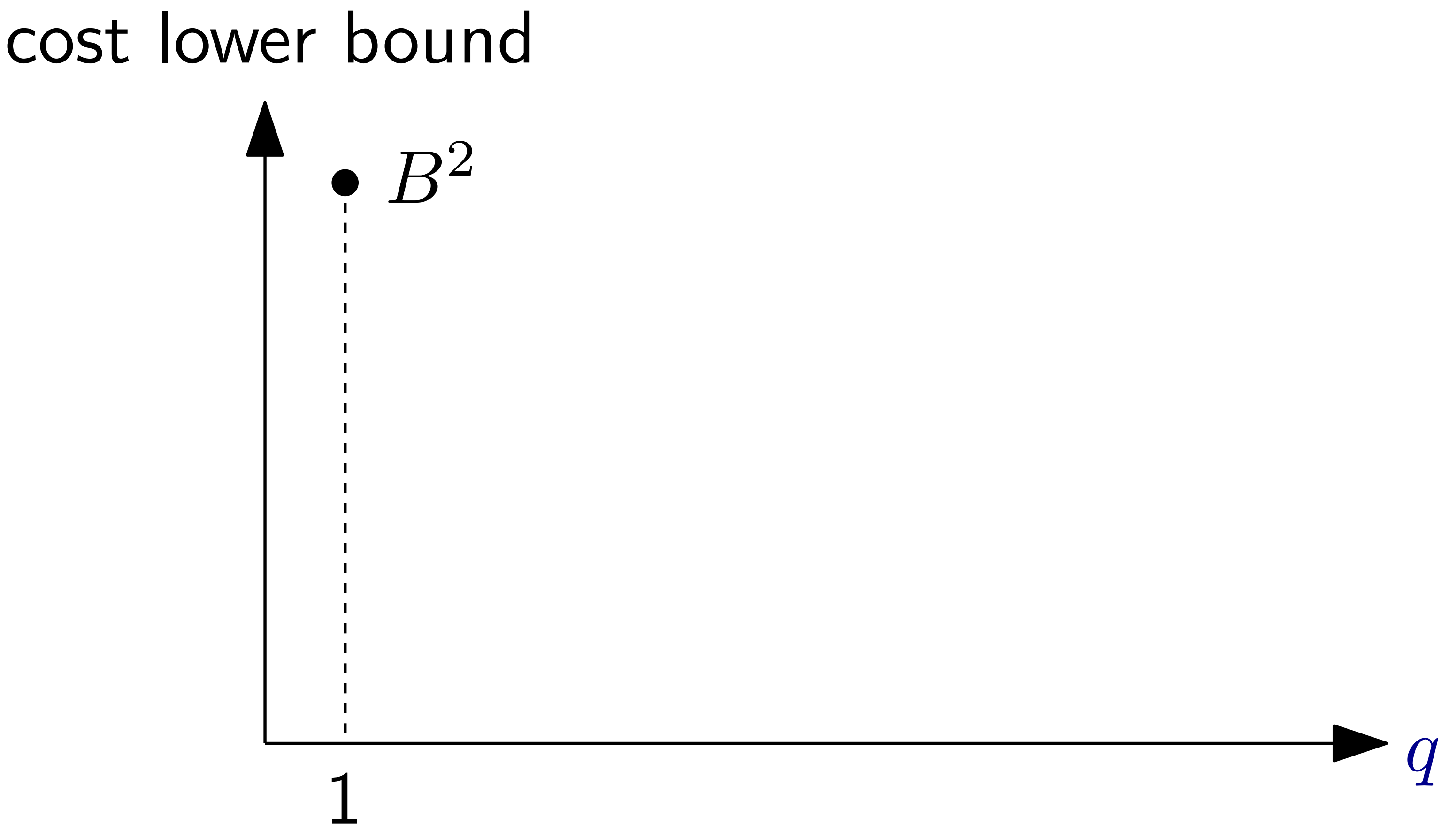
$$\begin{cases} \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\ \Omega(B \log_q B), & \text{for any } q. \end{cases}$$



Ball-Shuffling Lower Bounds

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

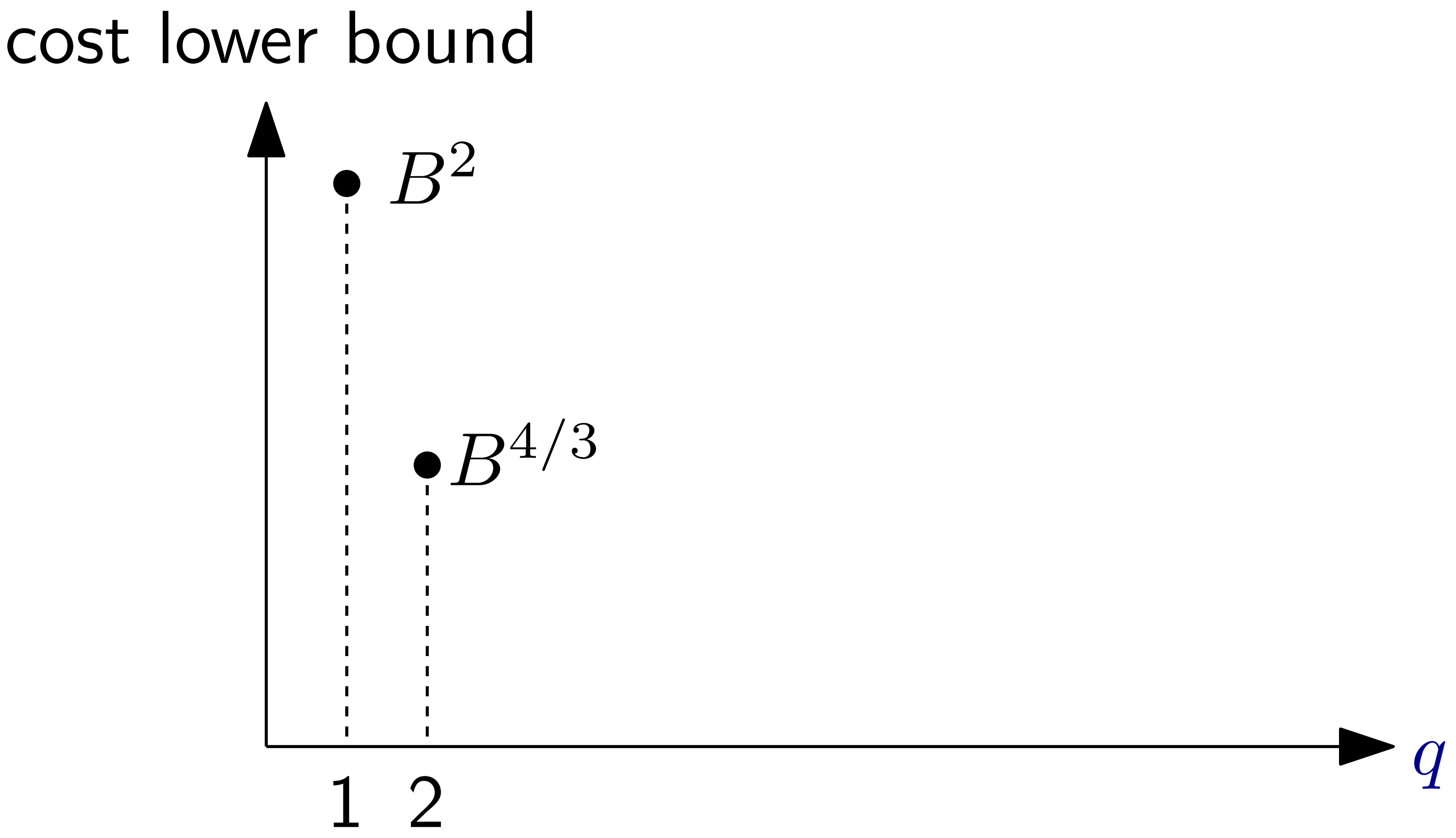
$$\begin{cases} \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\ \Omega(B \log_q B), & \text{for any } q. \end{cases}$$



Ball-Shuffling Lower Bounds

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

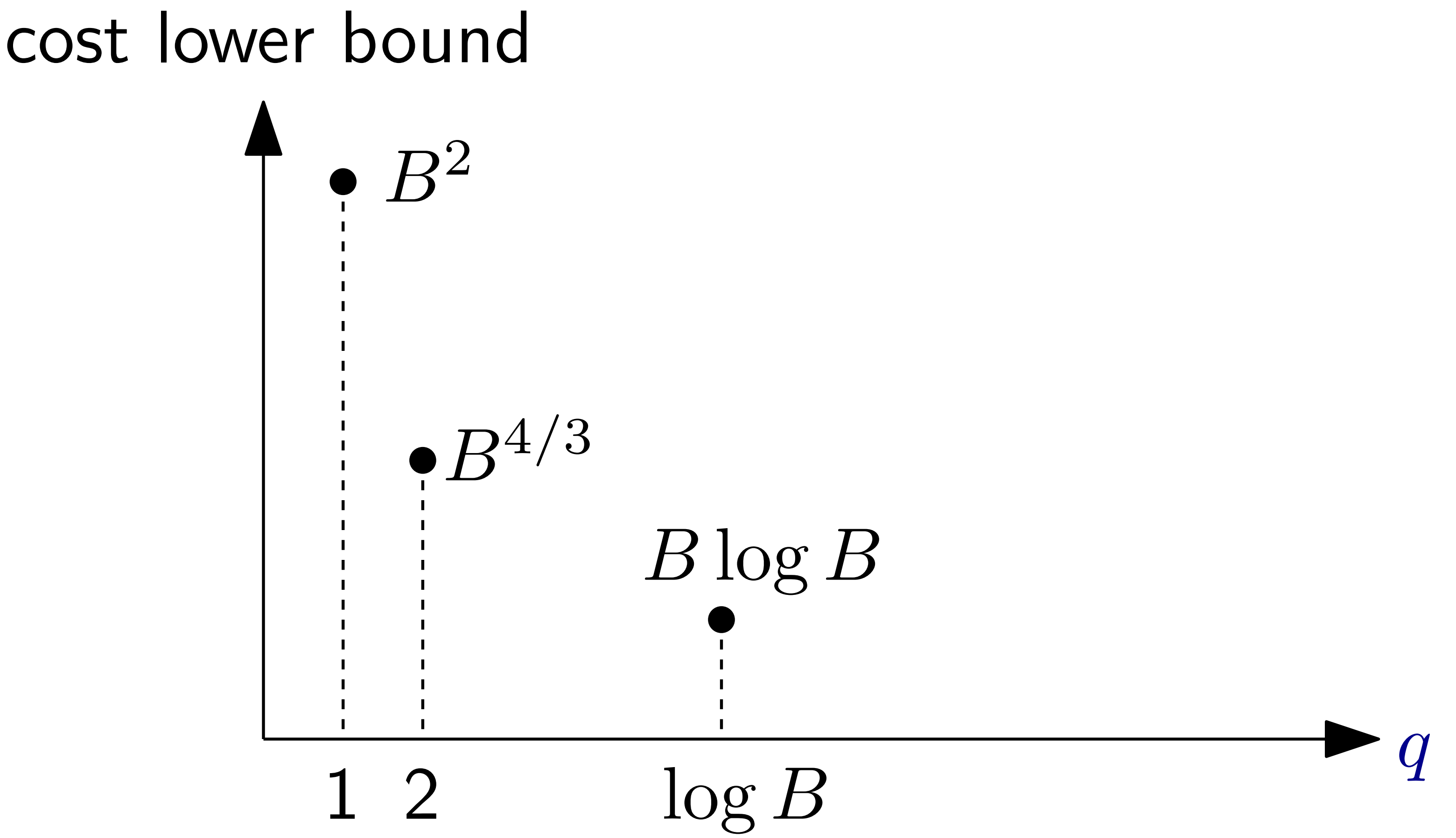
$$\begin{cases} \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\ \Omega(B \log_q B), & \text{for any } q. \end{cases}$$



Ball-Shuffling Lower Bounds

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

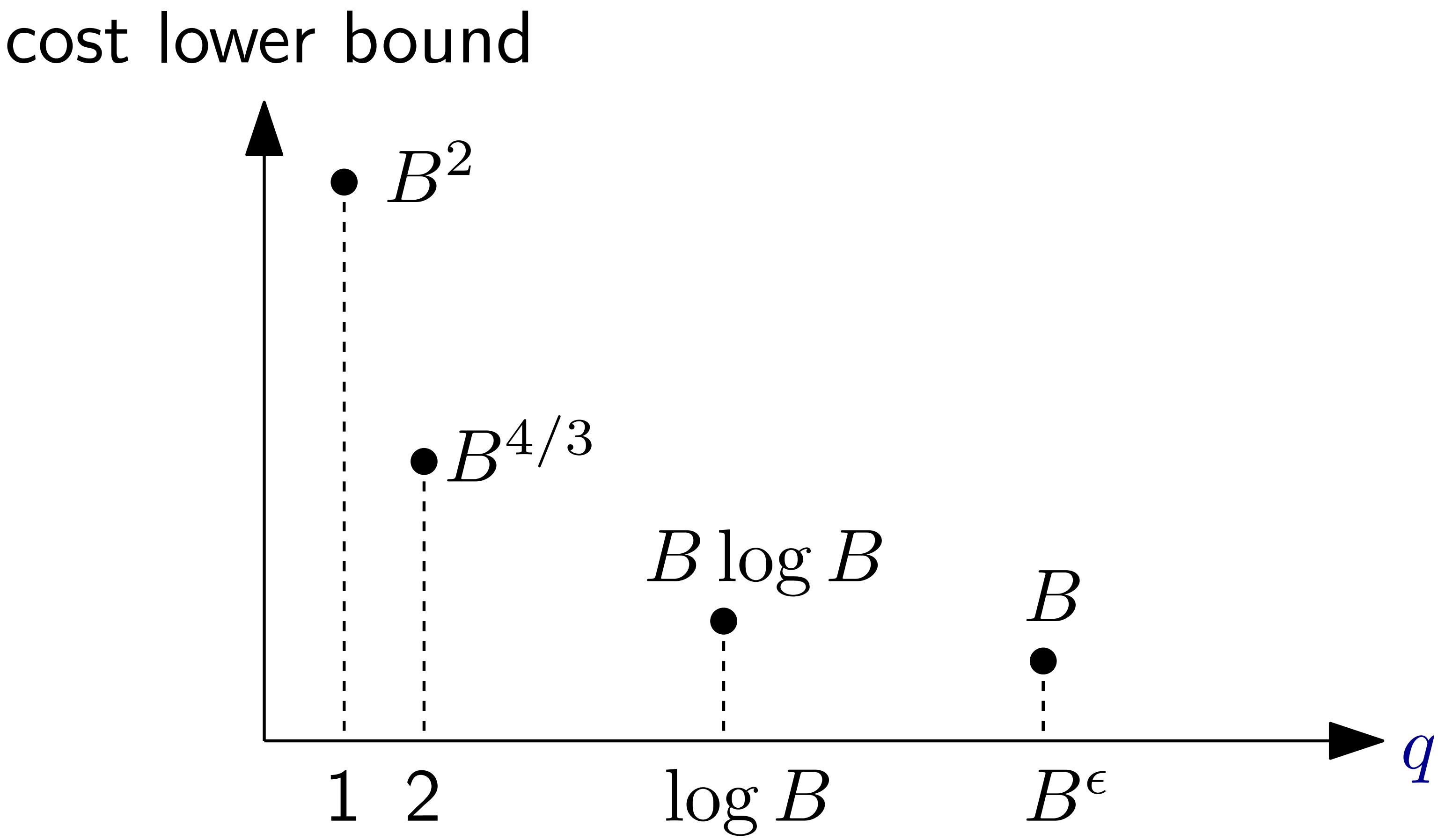
$$\begin{cases} \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\ \Omega(B \log_q B), & \text{for any } q. \end{cases}$$



Ball-Shuffling Lower Bounds

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

$$\begin{cases} \Omega(q \cdot B^{1+\Omega(1/q)}), & \text{for } q < \alpha \log B \text{ where } \alpha \text{ is any constant;} \\ \Omega(B \log_q B), & \text{for any } q. \end{cases}$$



Dynamic B-trees

Dynamic Hash Tables

Dynamic Hash Tables

- B-tree query I/O: $O(\log_B \frac{N}{M})$
- Hash table query I/O: $1 + 1/2^{\Omega(B)}$; insertion the same

Dynamic Hash Tables

- B-tree query I/O: $O(\log_B \frac{N}{M})$
- Hash table query I/O: $1 + 1/2^{\Omega(B)}$; insertion the same

A long-time conjecture in the external memory community:

The insertion cost must be $\Omega(1)$ I/Os if the query cost is required to be $O(1)$ I/Os.

Dynamic Hash Tables

- B-tree query I/O: $O(\log_B \frac{N}{M})$
- Hash table query I/O: $1 + 1/2^{\Omega(B)}$; insertion the same

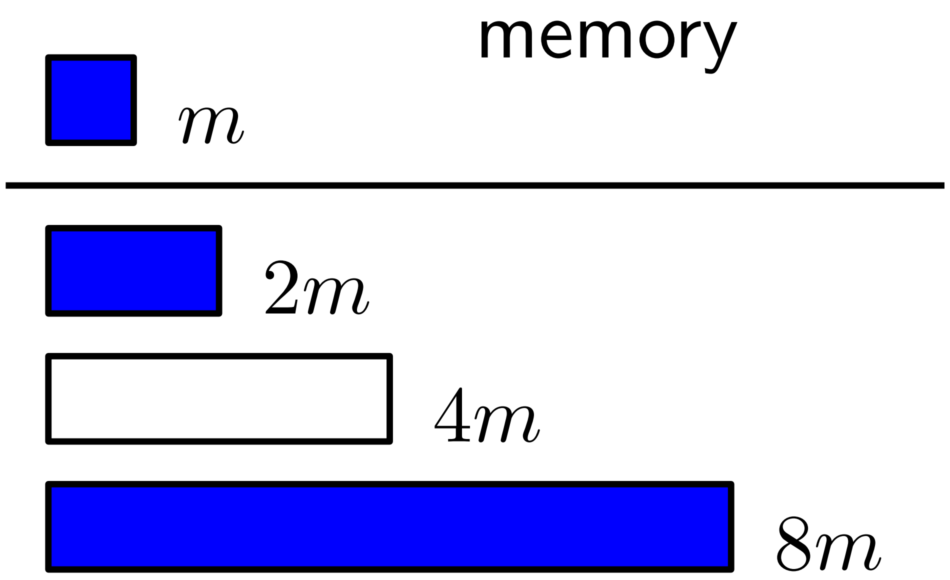
A long-time conjecture in the external memory community:

The insertion cost must be $\Omega(1)$ I/Os if the query cost is required to be $O(1)$ I/Os.

Buffering is useless?

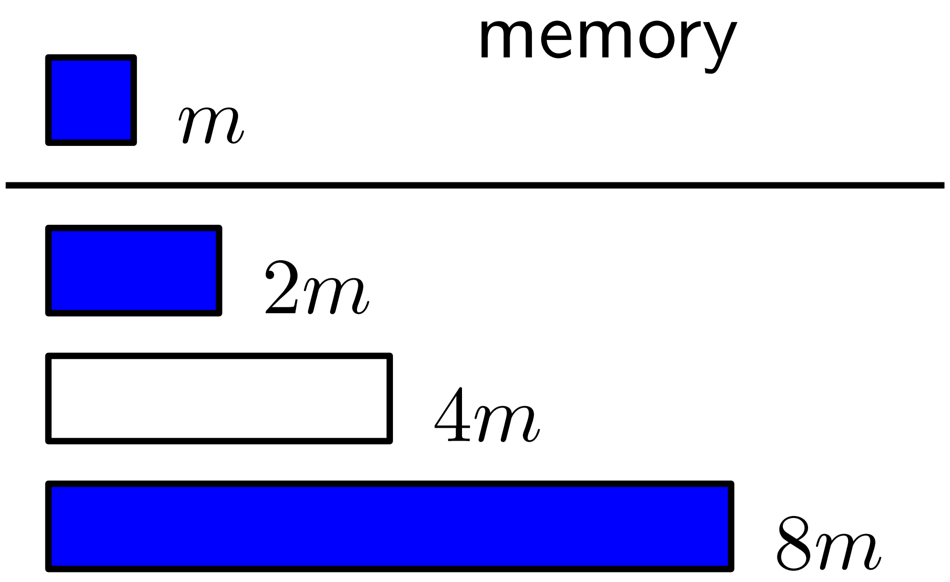
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)



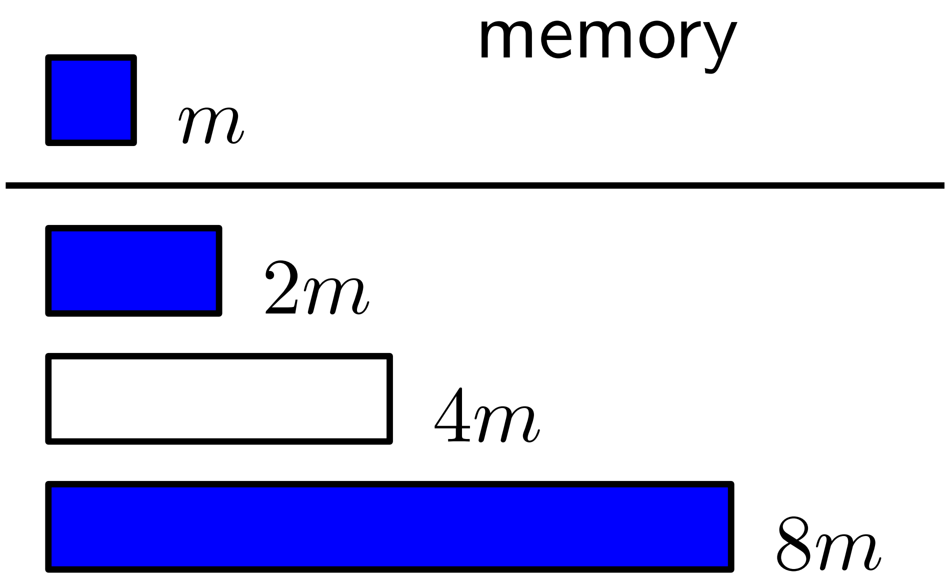
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B} \log \frac{N}{M})$
  - Expected average query: $O(1)$



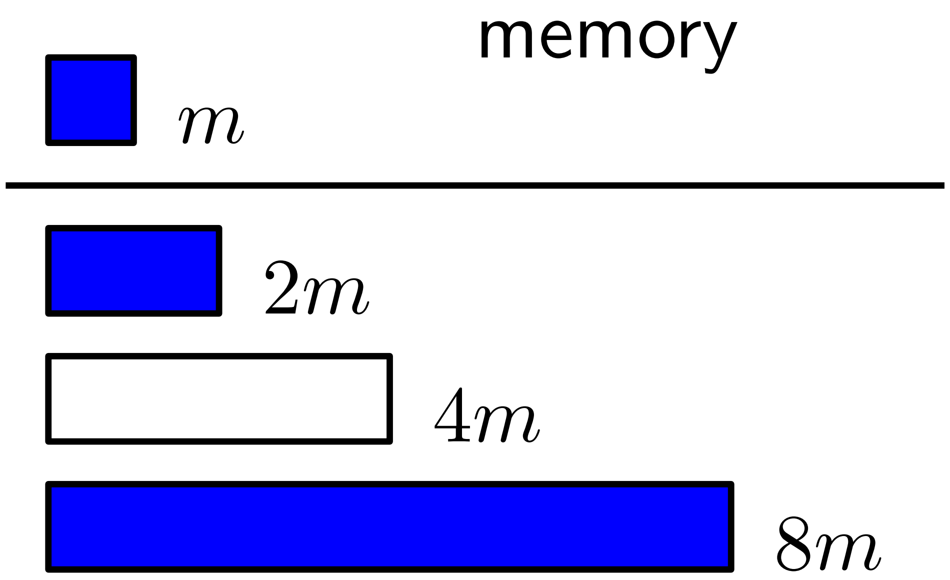
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B}\log\frac{N}{M})$
  - Expected average query: $O(1)$
- Improving query time
  - Idea: Keep one table large enough



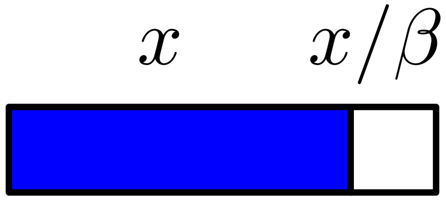
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B}\log\frac{N}{M})$
  - Expected average query: $O(1)$



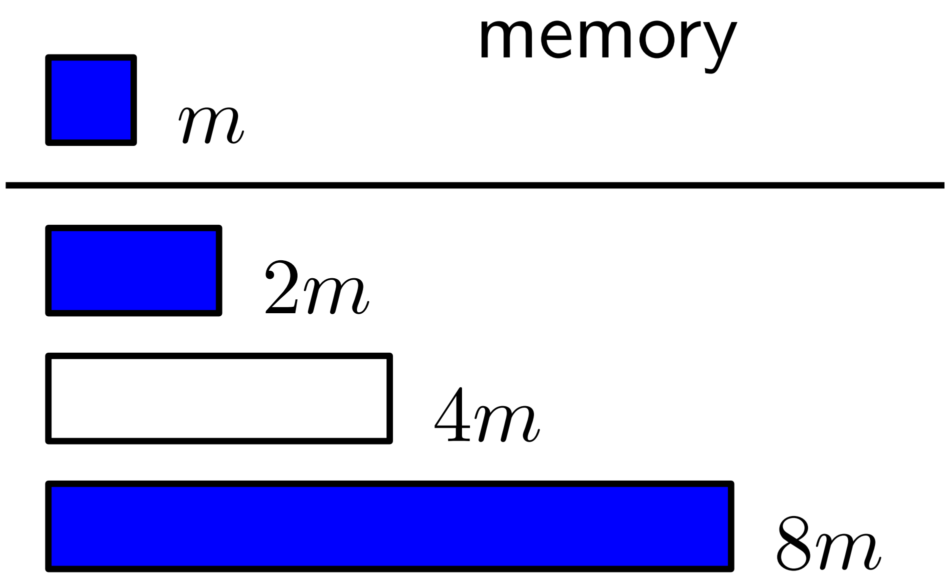
- Improving query time
  - Idea: Keep one table large enough

For some parameter $\beta = B^c, c \leq 1$



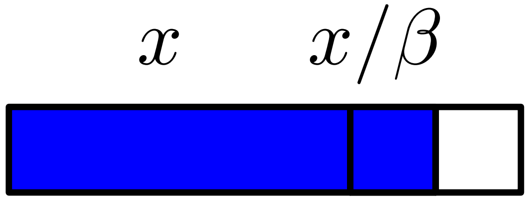
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B} \log \frac{N}{M})$
  - Expected average query: $O(1)$



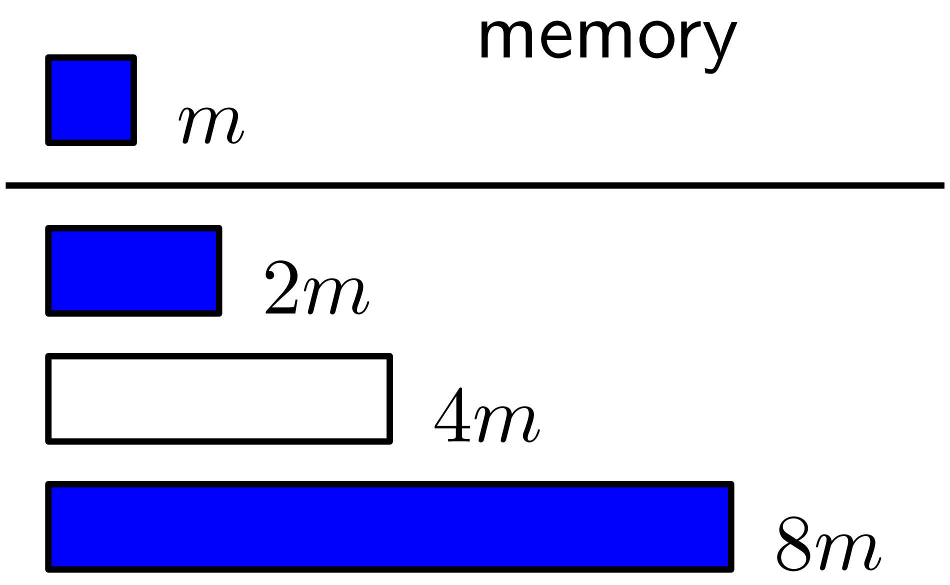
- Improving query time
  - Idea: Keep one table large enough

For some parameter $\beta = B^c, c \leq 1$



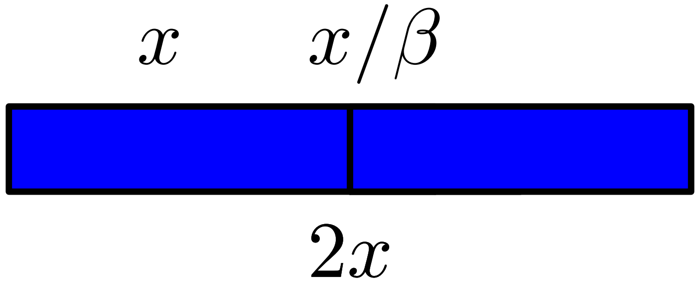
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B}\log\frac{N}{M})$
  - Expected average query: $O(1)$



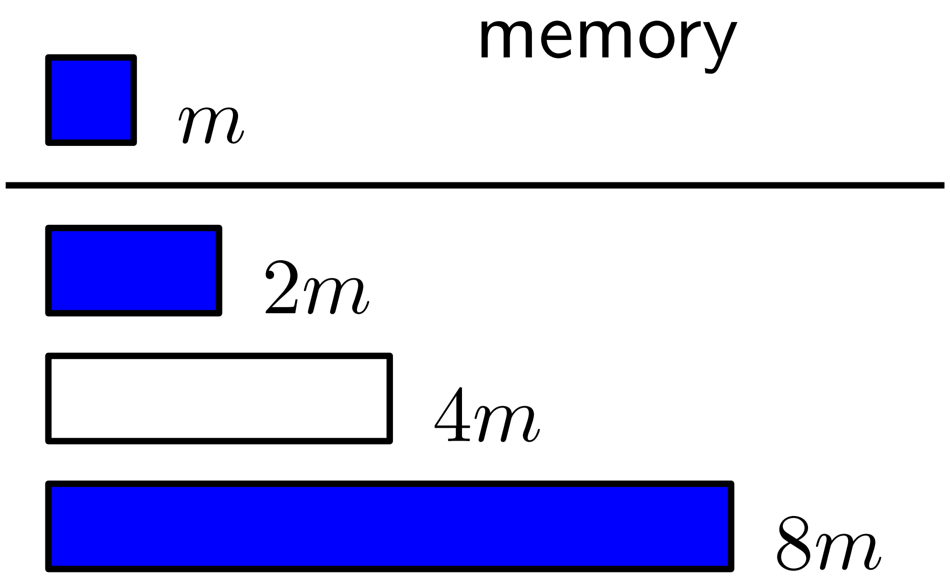
- Improving query time
  - Idea: Keep one table large enough

For some parameter $\beta = B^c, c \leq 1$



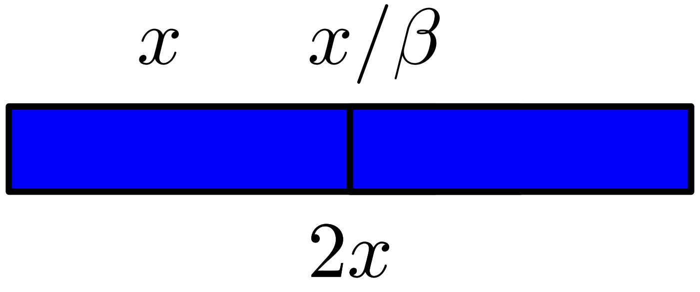
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B}\log\frac{N}{M})$
  - Expected average query: $O(1)$



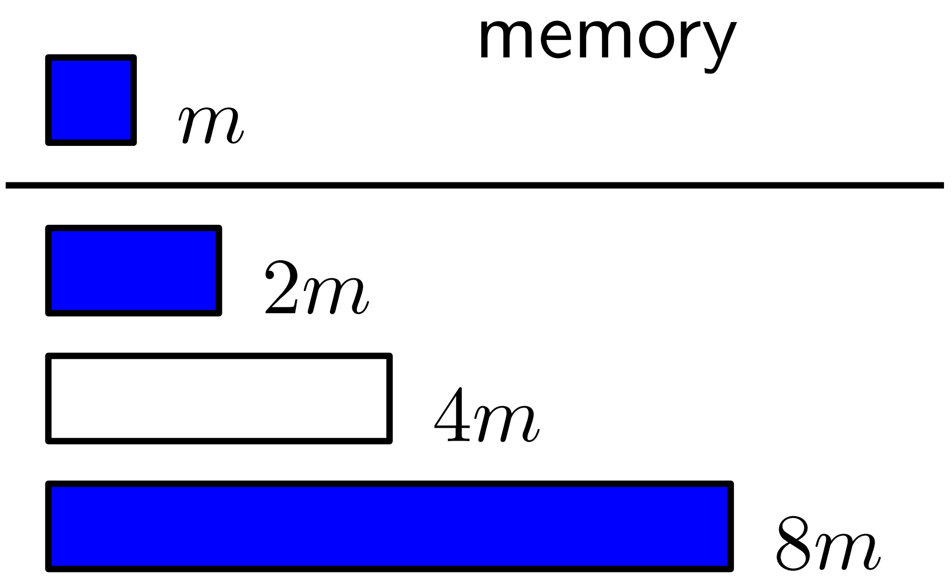
- Improving query time
  - Idea: Keep one table large enough
  - Insertion: $O(B^{c-1})$

For some parameter $\beta = B^c, c \leq 1$



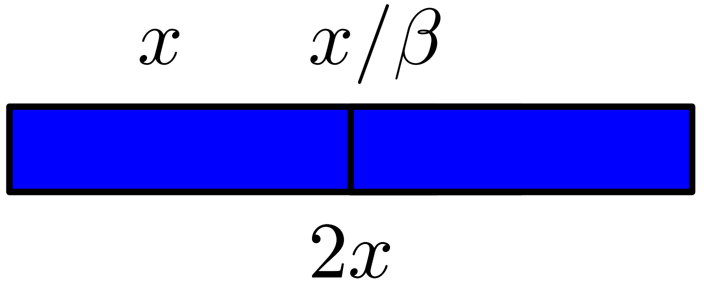
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B}\log\frac{N}{M})$
  - Expected average query: $O(1)$



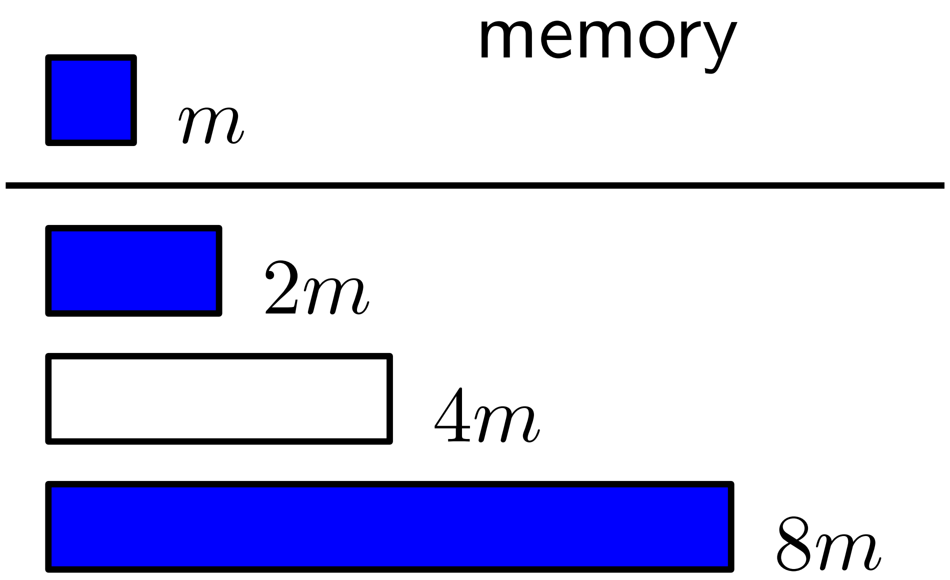
- Improving query time
  - Idea: Keep one table large enough
  - Insertion: $O(B^{c-1})$
  - Query: $1+O(1/B^c)$

For some parameter $\beta = B^c, c \le 1$



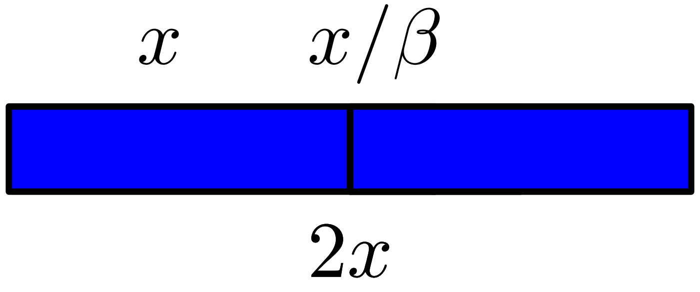
Dynamic Hash Tables (for successful queries)

- Logarithmic method (folklore?)
  - Insertion: $O(\frac{1}{B}\log\frac{N}{M})$
  - Expected average query: $O(1)$



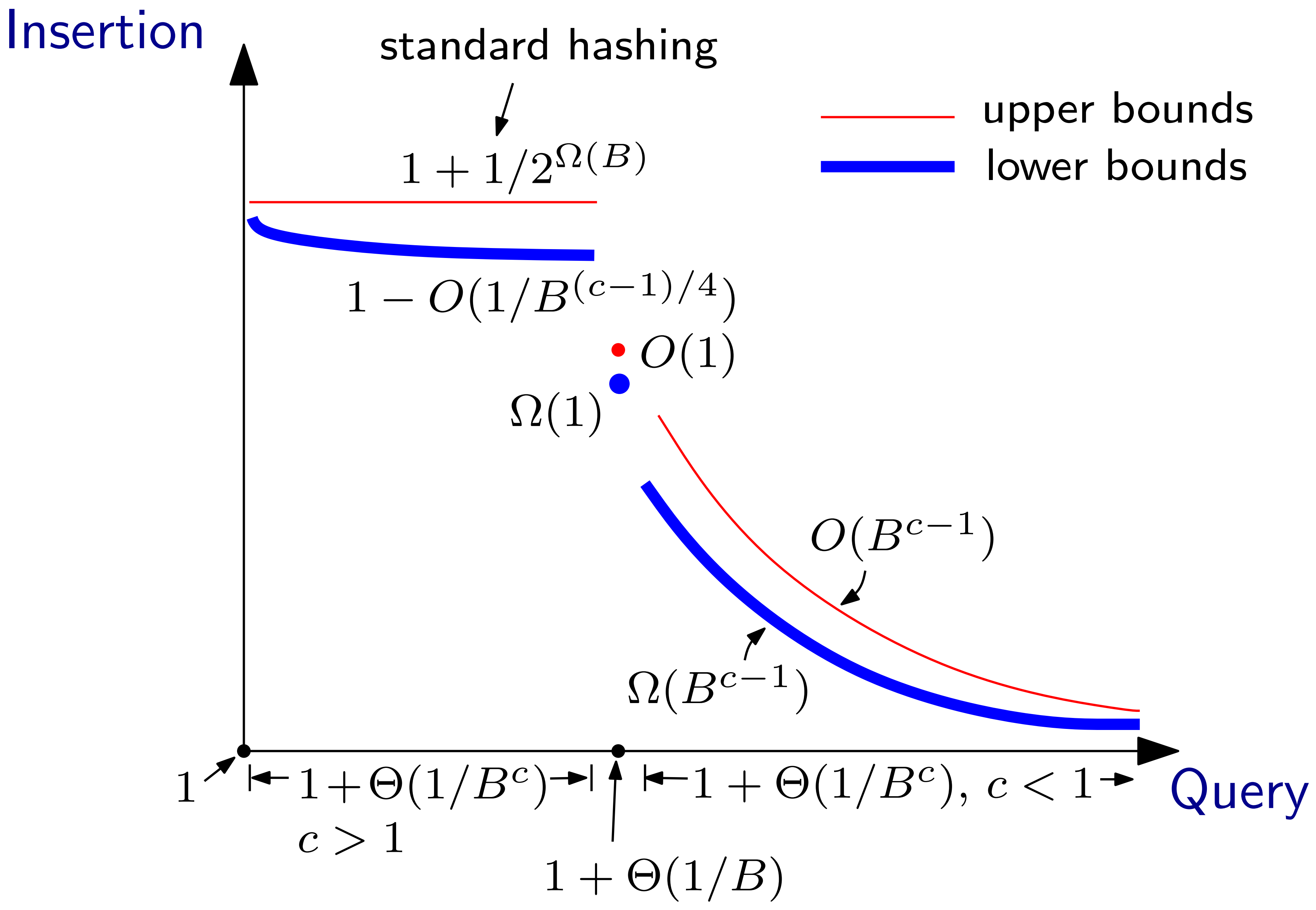
- Improving query time
  - Idea: Keep one table large enough
  - Insertion: $O(B^{c-1})$
  - Query: $1+O(1/B^c)$
  - Still far from the target $1+1/\Omega(2^B)$

For some parameter $\beta = B^c, c \leq 1$



Query-Insertion Tradeoff for Successful queries

[Wei, Yi, Zhang, SPAA'09]

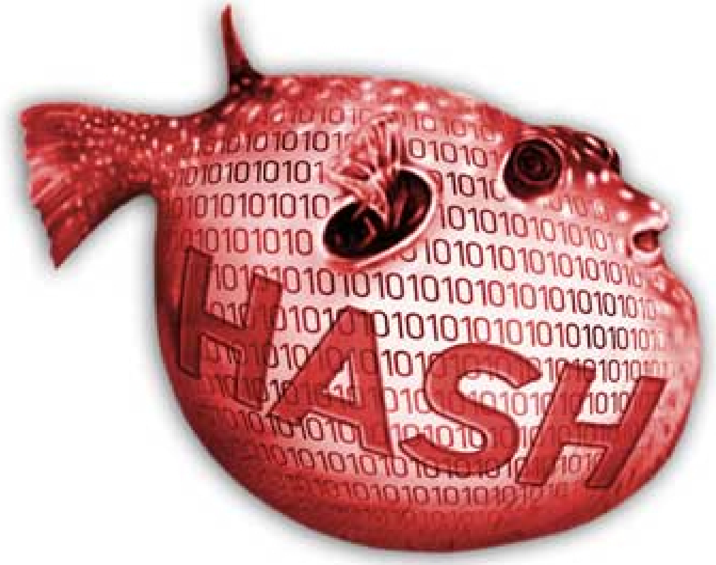


Indexability Too Strong!

- Naïve solution: For every B items, write to a block.
  - Query cost is 1, insertion is $1/B$

Indexability Too Strong!

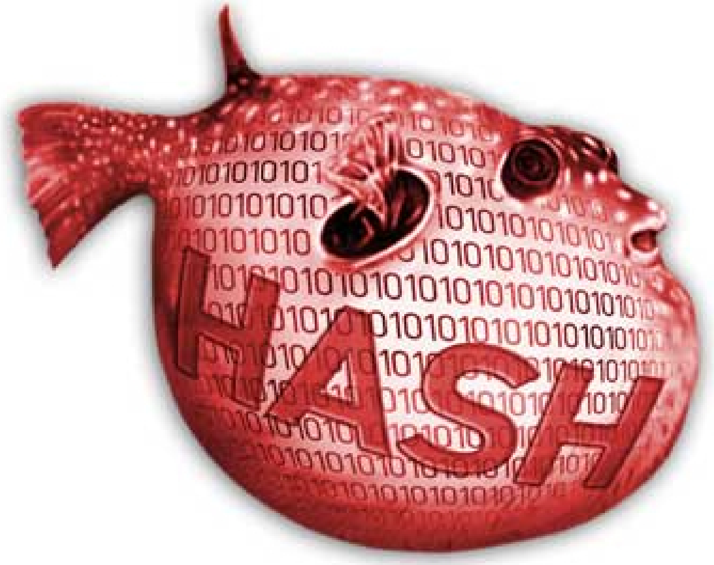
- Naïve solution: For every B items, write to a block.
  - Query cost is 1, insertion is $1/B$



Too many possible mappings!

Indexability Too Strong!

- Naïve solution: For every B items, write to a block.
  - Query cost is 1, insertion is $1/B$



Too many possible mappings!

- Indexabilty $+$ information-theoretical argument

  If with only the information in memory, the hash table cannot locate the item, then querying it takes at least 2 I/Os.

The Abstraction

- Consider the layout of a hash table at any snapshot. Denote all the blocks on disk by $B_1, B_2, \ldots, B_d$. Let $f : U \rightarrow \{1, \ldots, d\}$ be any function computable within memory.

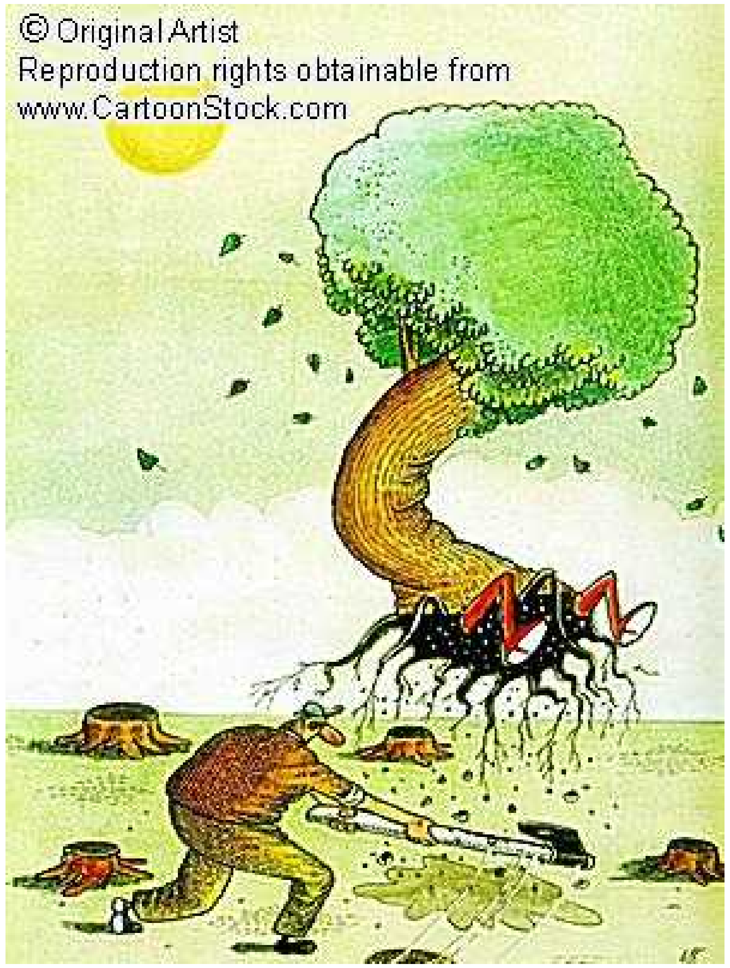
  We divide items inserted into 3 zones with respect to f.

  - Memory zone M: set of items stored in memory. $t_q = 0$.
  - Fast zone F: set of items x such that $x \in B_{f(x)}$. $t_q = 1$.
  - Slow zone S: The rest of items. $t_q = 2$.

The Key

The hash table can employ a family $\mathcal{F}$ of at most 2^M distinct f's.

Note that the current f adopted by the hash table is dependent upon the already inserted items, but the family $\mathcal{F}$ has to be fixed beforehand.

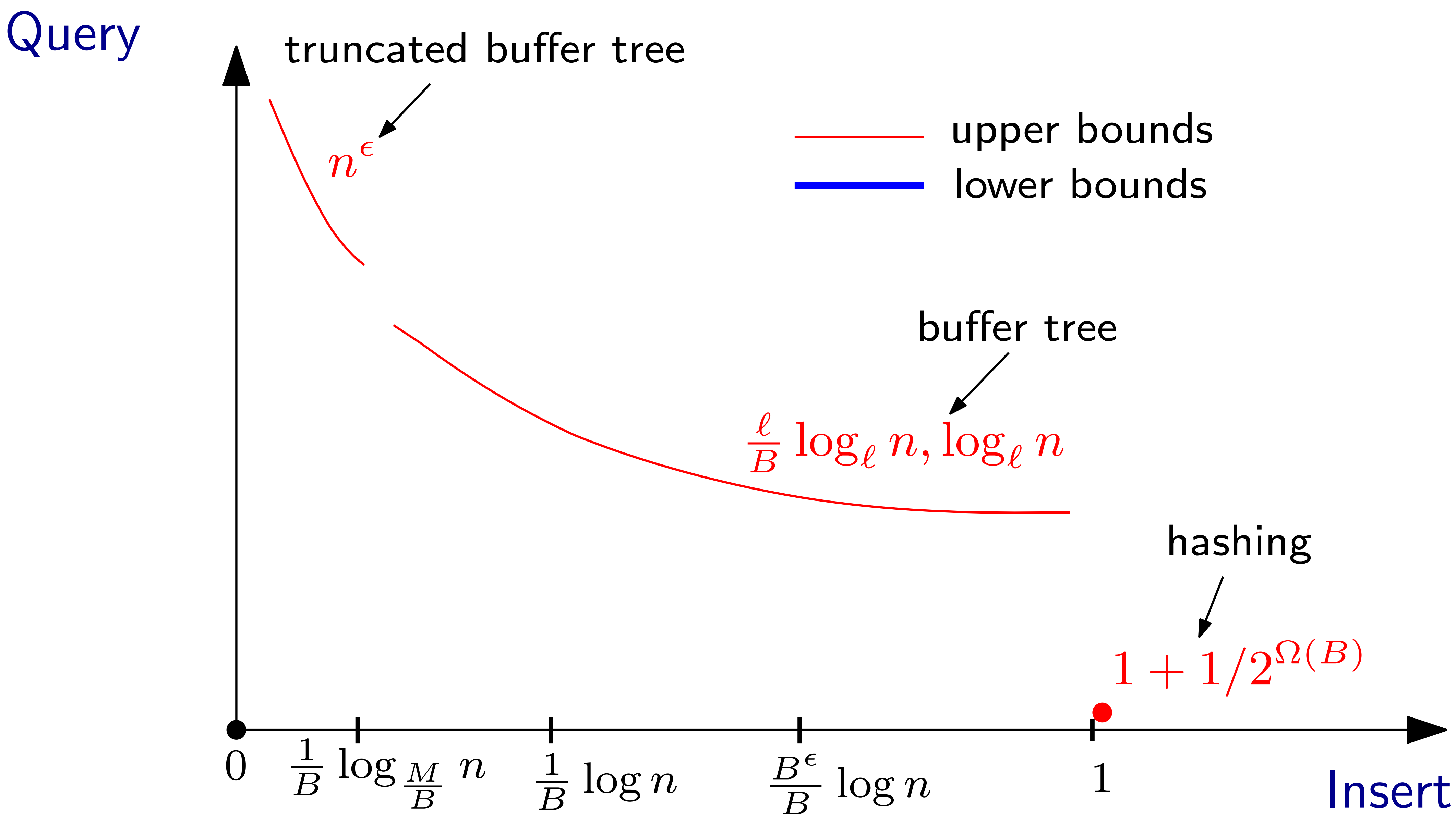


How about All Queries? (Latest results)

- We are essentially talking about the membership problem
    - Can't use indexability model
    - Have to use cell probe model

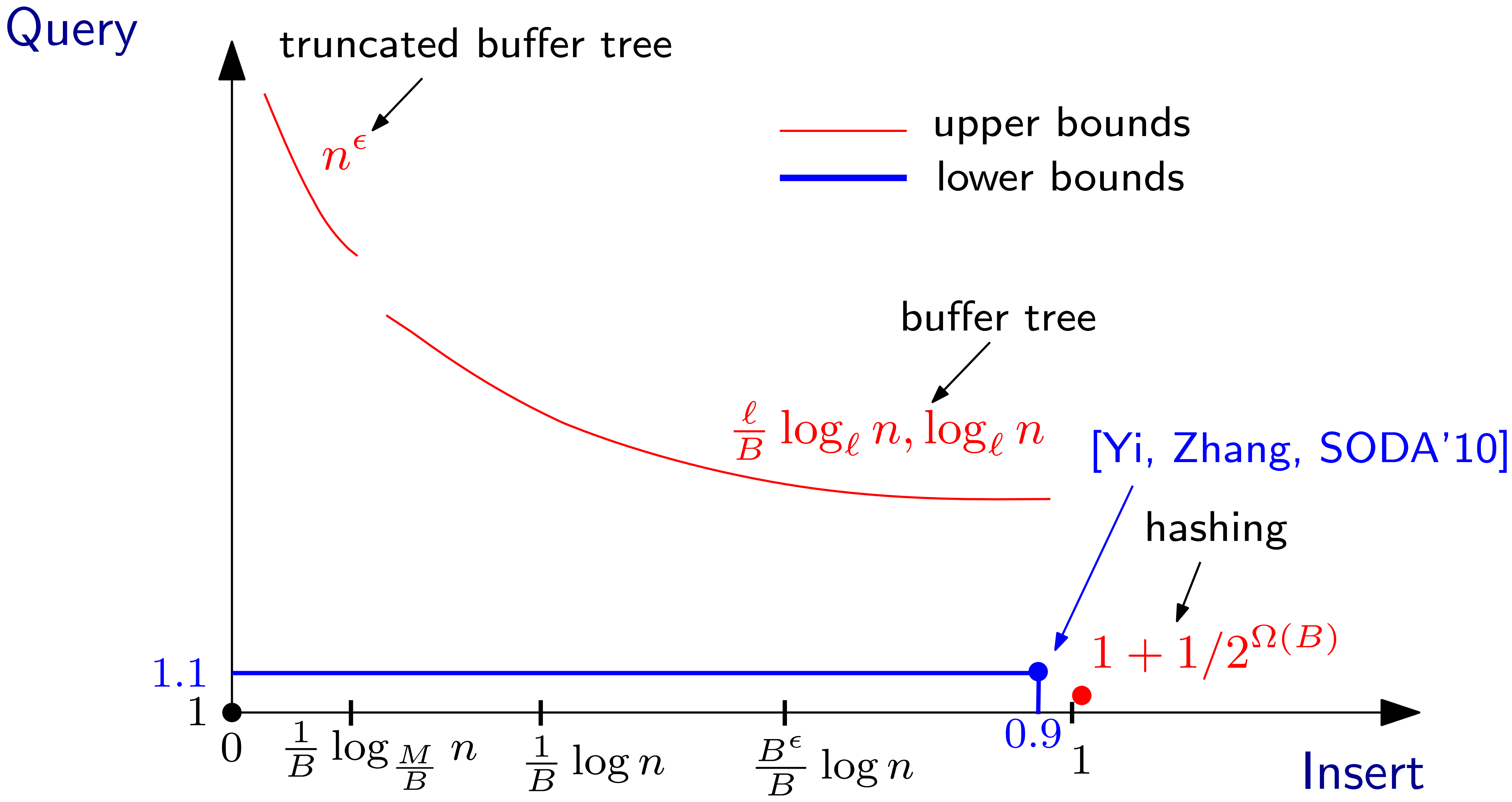
All queries (the membership problem)

(The cell probe model)



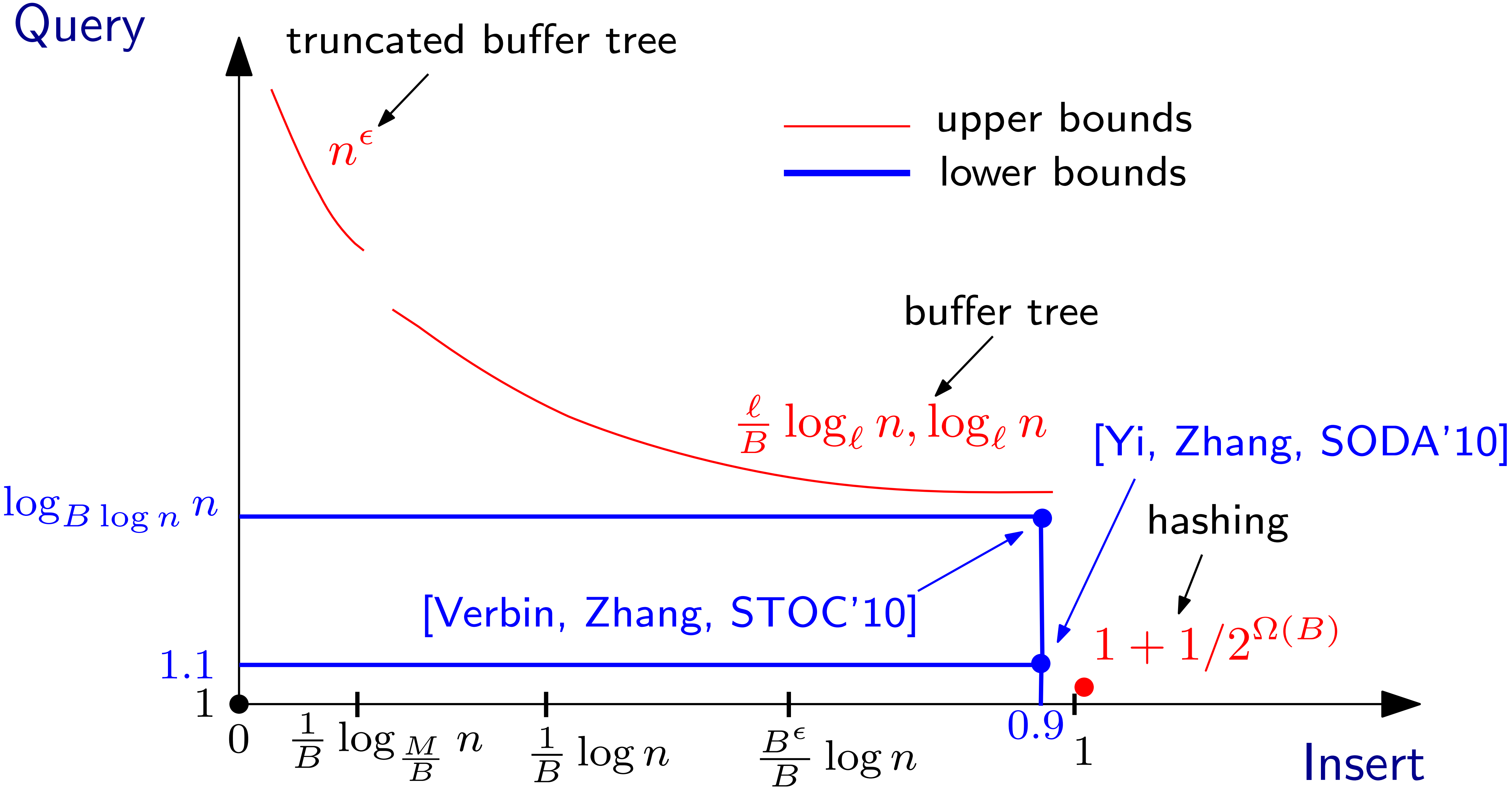
All queries (the membership problem)

(The cell probe model)



All queries (the membership problem)

(The cell probe model)



THE BIG BOLD CONJECTURE

All these fundamental data structure problems have the same query-update tradeoff in external memory when $u = o(1)$, for sufficiently large B.

Partial-sum: all B; Range reporting: $B > n^{\epsilon}$; Predecessor: unknown.

THE BIG BOLD CONJECTURE

All these fundamental data structure problems have the same query-update tradeoff in external memory when $u = o(1)$, for sufficiently large B.

Partial-sum: all B; Range reporting: $B > n^{\epsilon}$; Predecessor: unknown.

Strong implication: The buffer tree (and many of the log method based structures) is simple, practical, versatile, and optimal!

The End

THANK YOU

Q and A