Hash Indexing (ctd.)

and

Using Indexes

R&G Chapter 11, 14

(slides adapted from content by J.Gehrke, J.Shanmugasundaram, and/or C.Koch)

Announcements

Homework 3 due tonight

No homework 4 assigned this week
(Project 1 due 1 week from Monday)

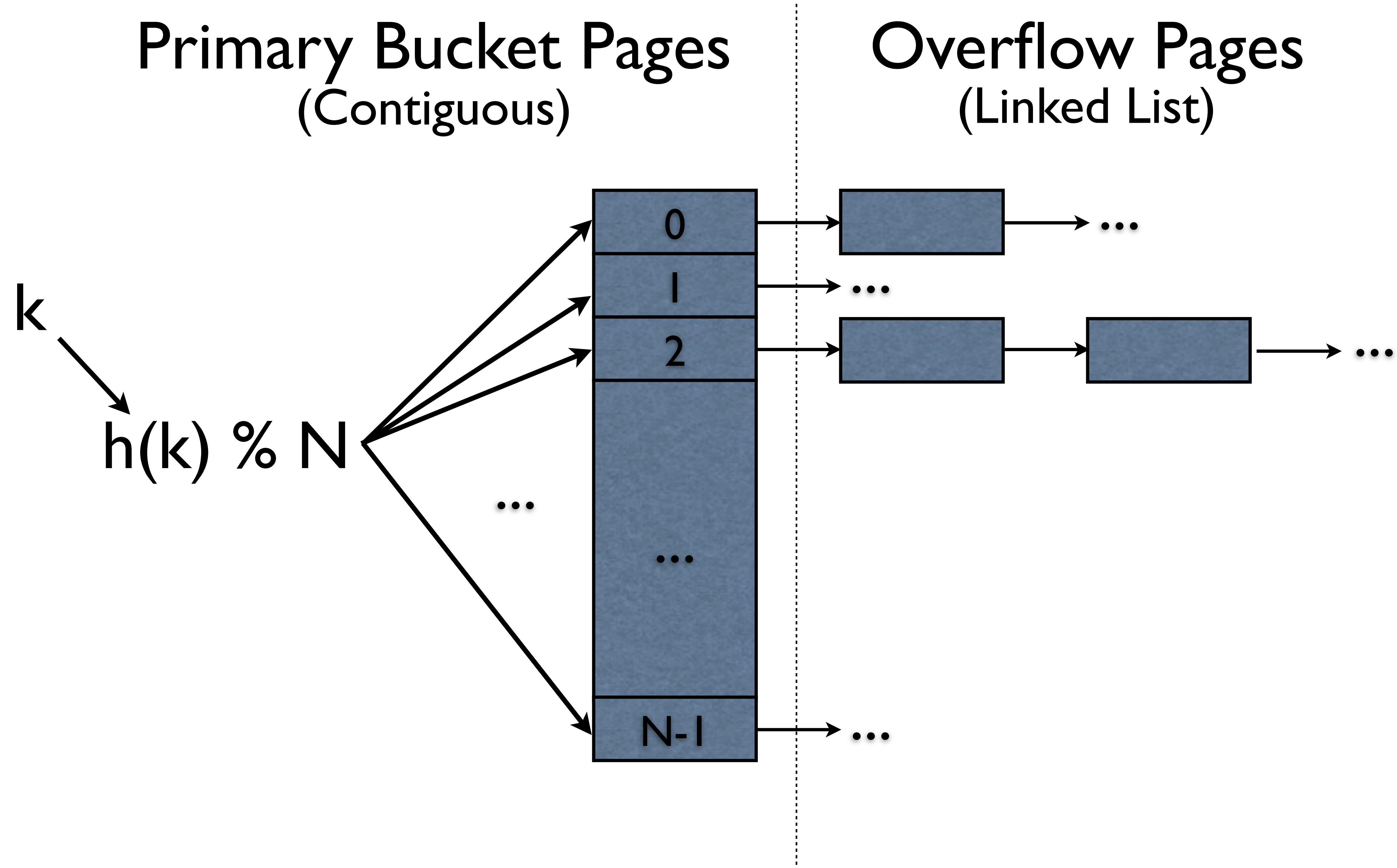
Dr. Chomicki will be substituting Monday
(Monday Office Hours → Wednesday)

Recap: Hash Indexes

- As with trees: request a key k and get record(s) or record id(s) with k.
- Hash-based indexes support equality lookups
  - … in constant time (vs log(n) for tree)
  - … but don't support range lookups
- Static vs Dynamic Hashing
  - Tradeoffs similar to ISAM vs B+Tree

Higher fanouts mean shallower trees, and fewer pages loaded to find an entry. These trees are often quite shallow (depth 4-5), so even a small reduction is huge.
Page sizes are fixed, so the only way to get higher fanouts is to pack more keys/pointers into a page. This is difficult for fixed size keys, but consider variable-length strings.

Recap: Static Hashing

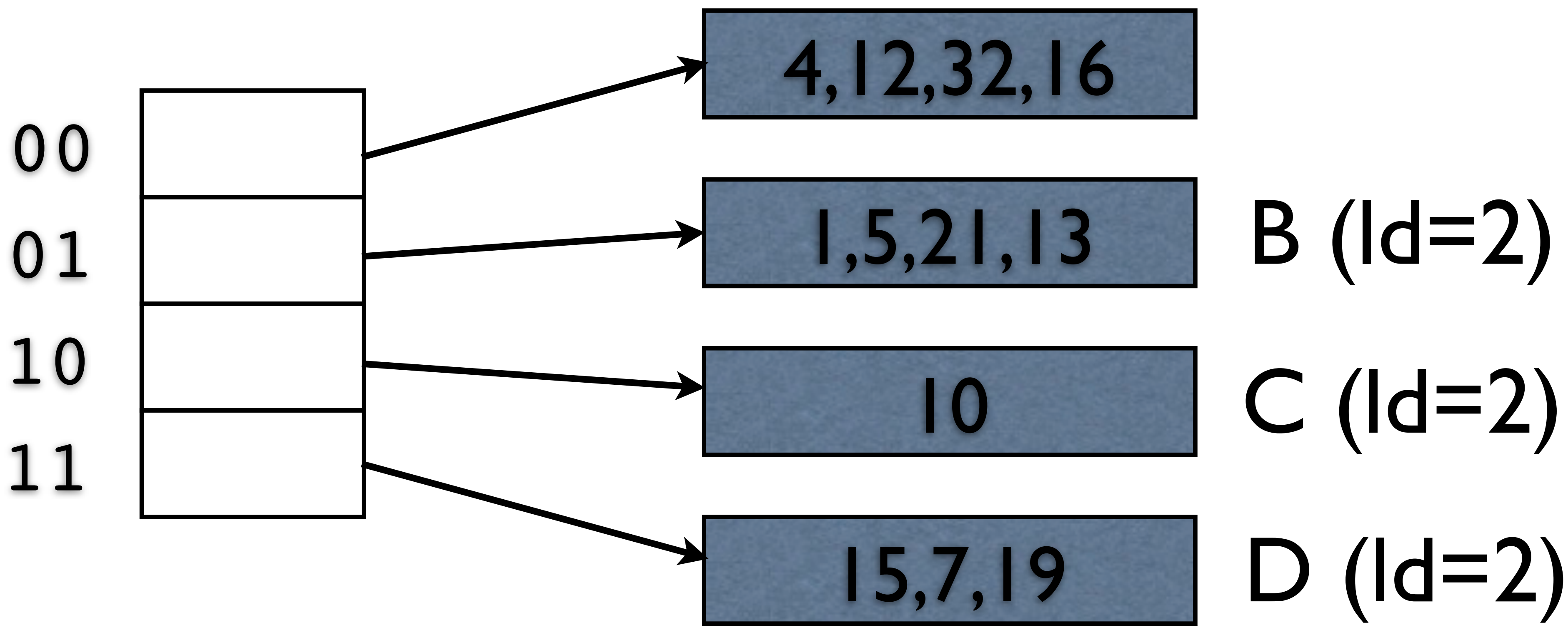


Recap: Extendible Hashing

- **Situation:** A bucket becomes full
  - Solution: Double the number of buckets!
  - Expensive! (N reads, 2N writes)
- **Idea:** Add one level of indirection
  - A directory of pointers to (noncontiguous) bucket pages.
  - Doubling just the directory is much cheaper.
  - Can we double only the directory?

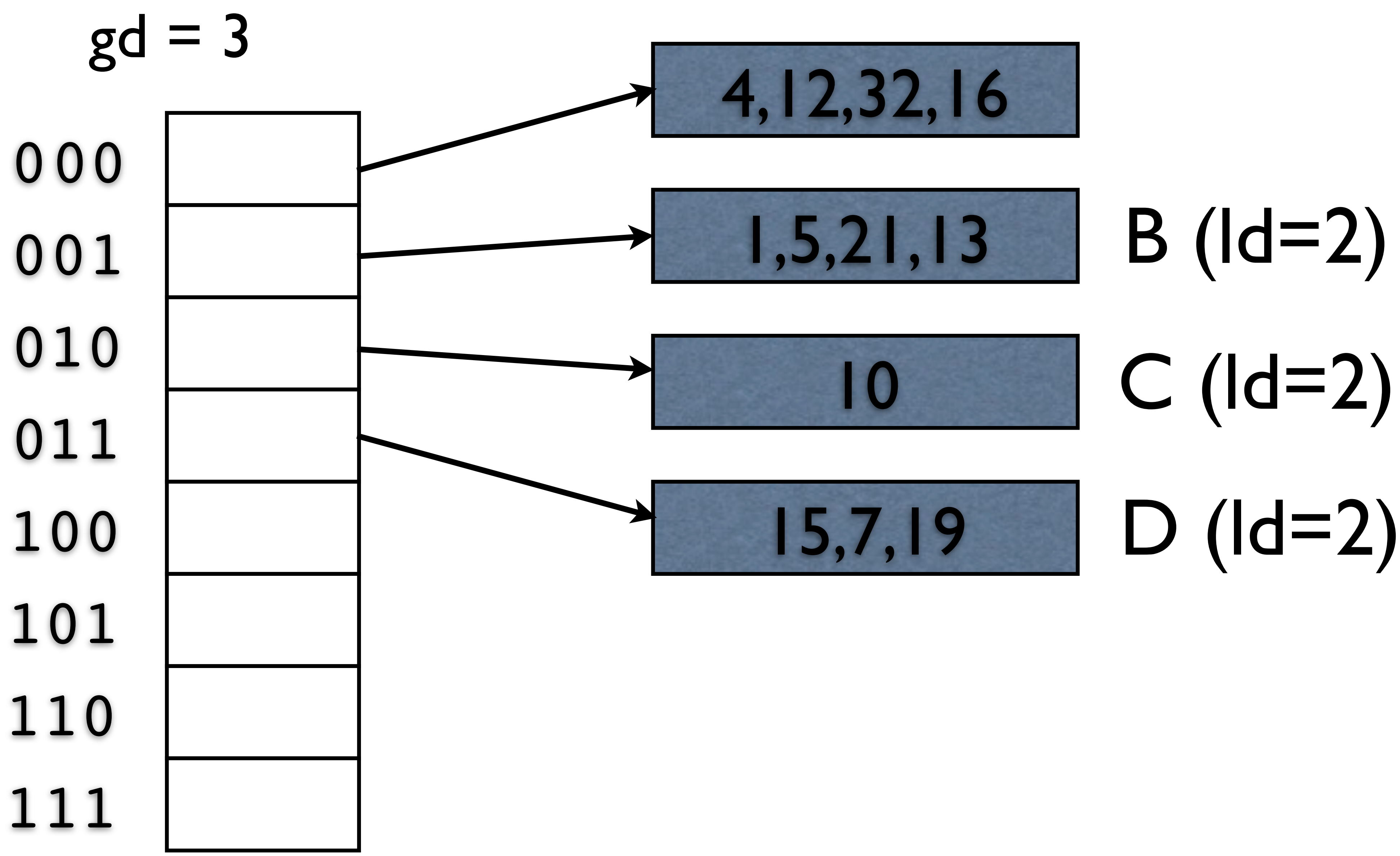
The hash function (and directory) must be changed (from a range of [0,N) to a range of [0,2N)) without affecting data values in buckets not affected by the split.

Extendible Hashing



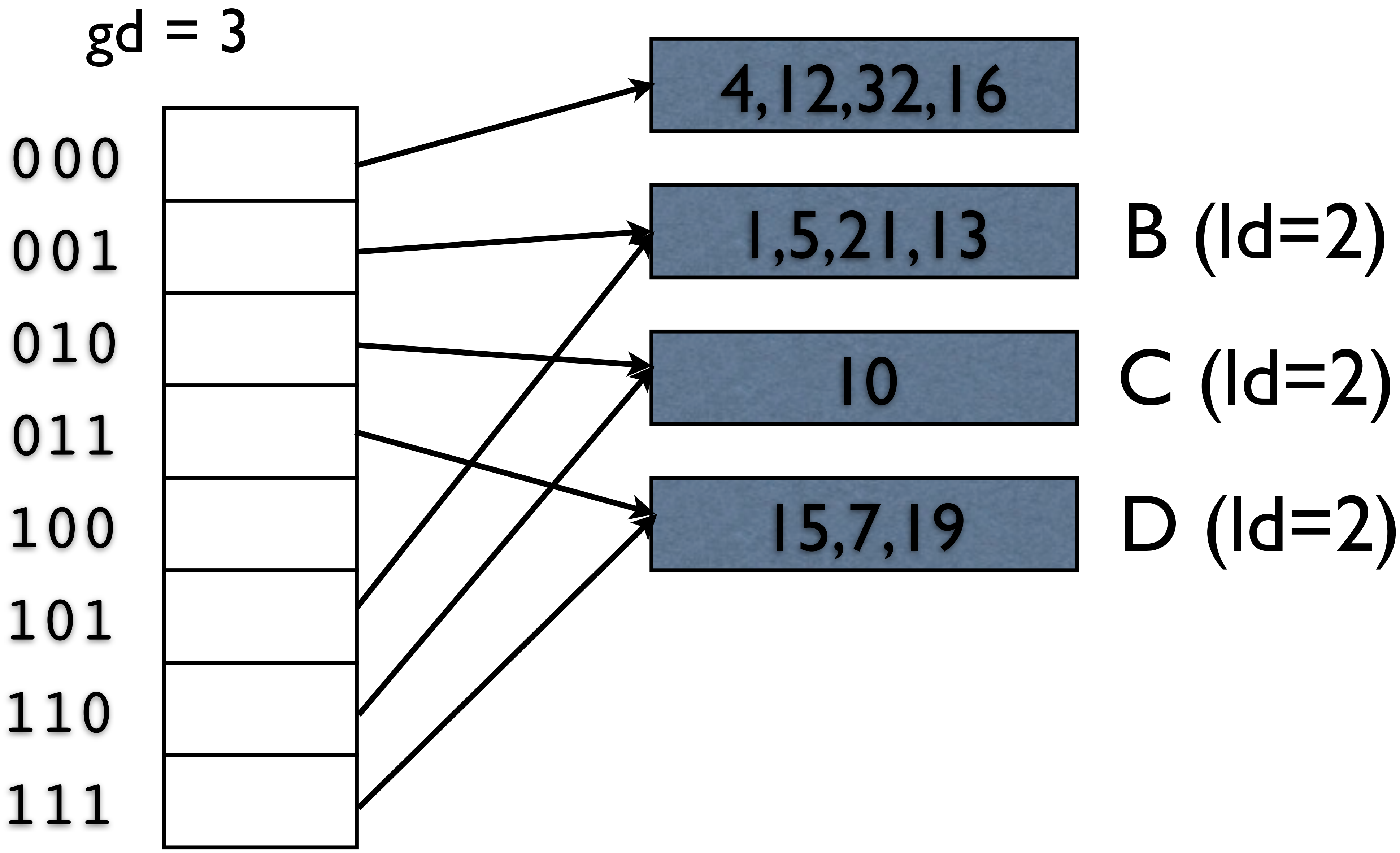
Bucket value **hashes** have the same last **ld** bits

Extendible Hashing



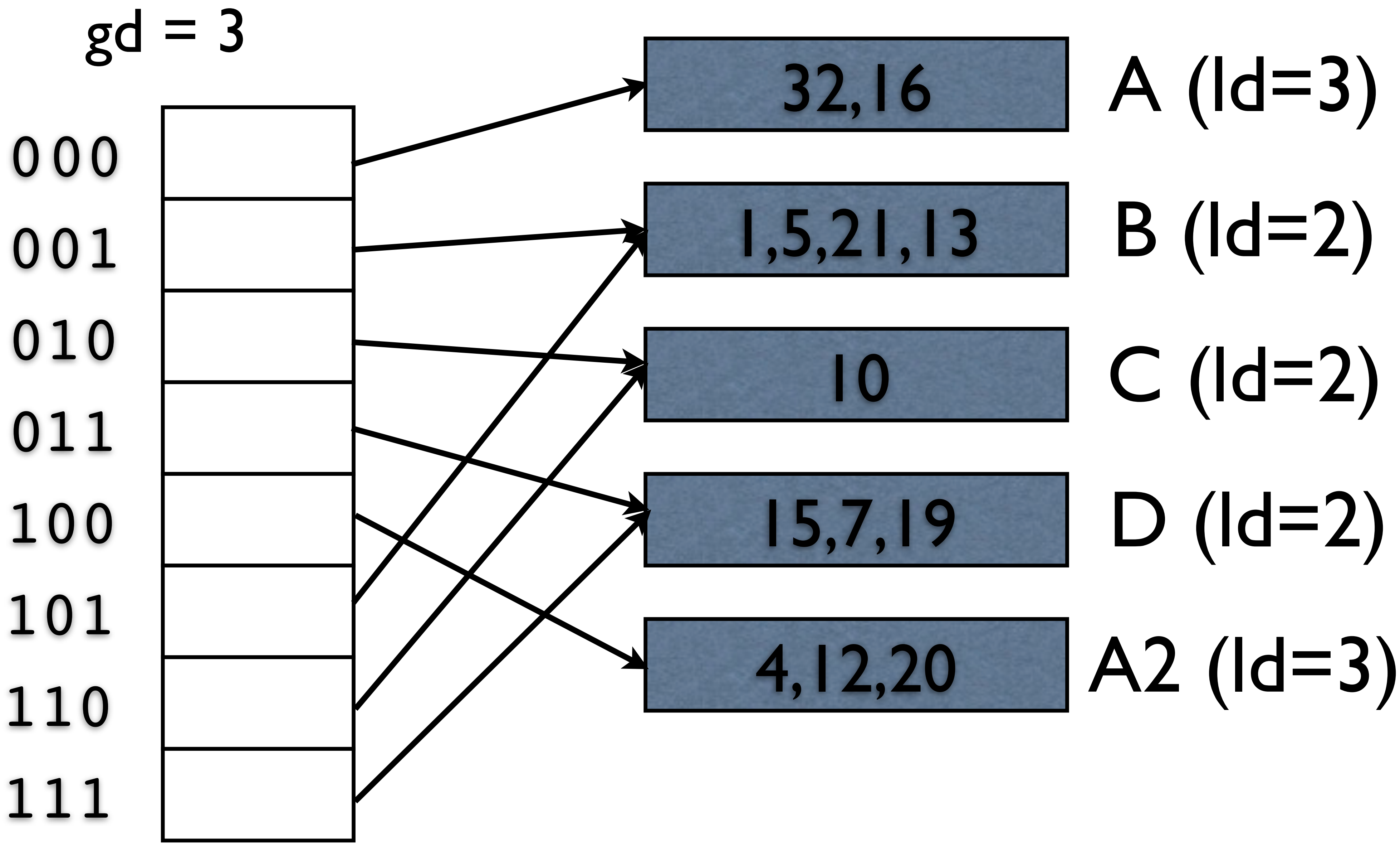
Bucket value **hashes** have the same last **ld** bits

Extendible Hashing



Bucket value **hashes** have the same last **ld** bits

Extendible Hashing



Bucket value **hashes** have the same last **ld** bits

Recap: Extendible Hashing

- Global depth of directory
  - **Upper bound** on # of bits required to determine the bucket of an entry.
- Local depth of a bucket
  - **Exact** # of bits required to determine if an entry belongs in this bucket.
- Using the last ld/gd bits makes it possible to double the directory size by copying entries.

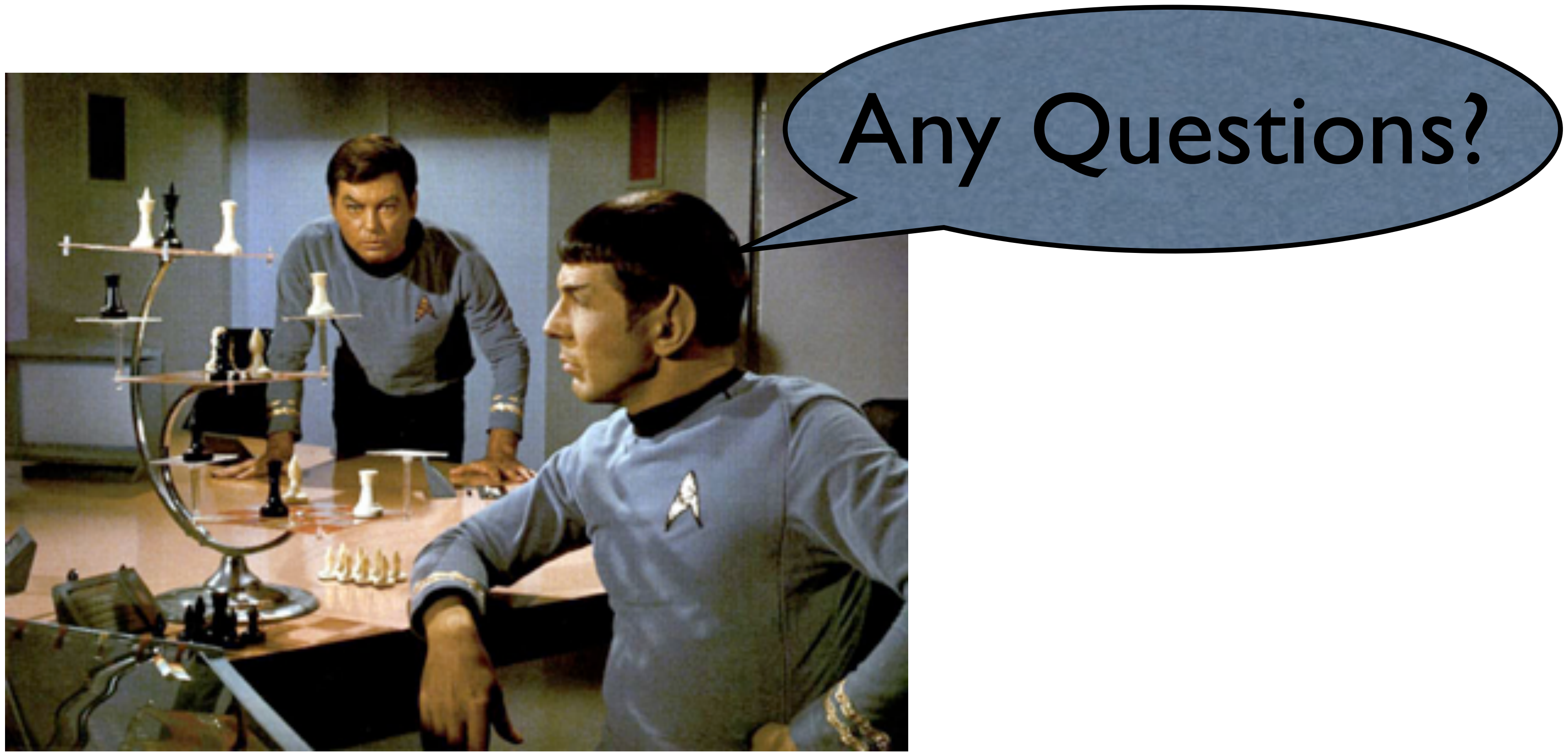


Image copyright: Paramount Pictures

Linear Hashing

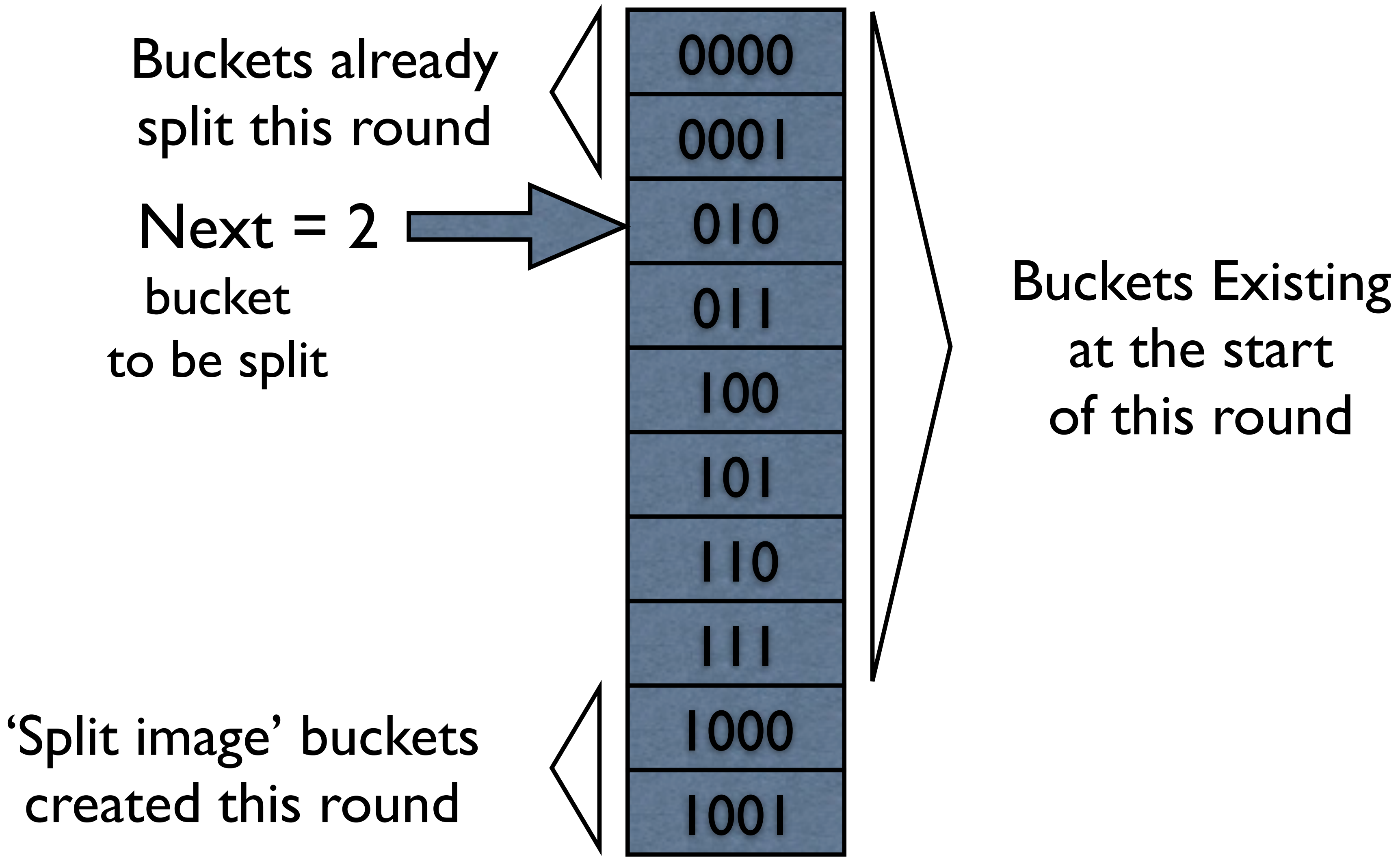
- A directory page adds 1 page lookup overhead.
- Can we do similar splits without indirection?
- Linear Hashing based on similar principle.
  - Start with the last *n* bits of each hash fn.
  - When you decide to split, start using *n+1* bits.
- **Key difference**: Split incrementally
  - Part of the hash table uses *n* bits, rest uses *n+1*
  - Each *round* increase *n* by one (1 round = 1 full split)

Friday, February 8, 13

We can generalize the splitting idea a little bit: We're taking one hash function h(k), and defining a new function: h'(k,n) = h(k) % 2^n (2 to the nth). Another way to look at this is that we're defining a family of hash functions h'1(k) = h(k) % 2, h'2(k) = h(k) % 3, h'4(k) = h(k) % 8, …
Any family of hash functions that satisfies the copy on split property can be switched in for this one
- That is, we can swap in any family as long as h'n(k) = h'(n+1)(k) % 2^n

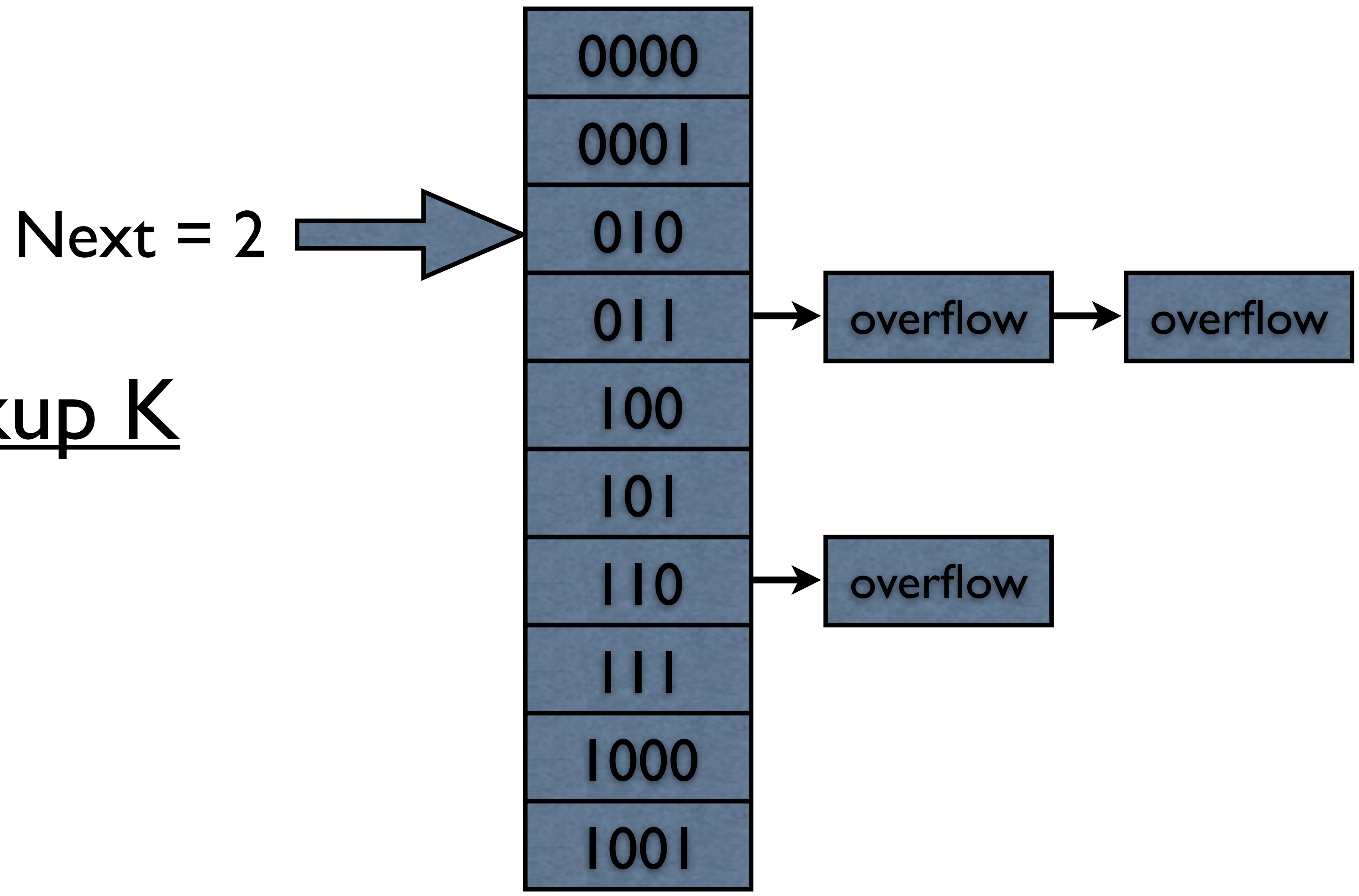
Linear Hashing

Level = 3 (2^3 = 8 Entries)



Linear Hashing:Lookups

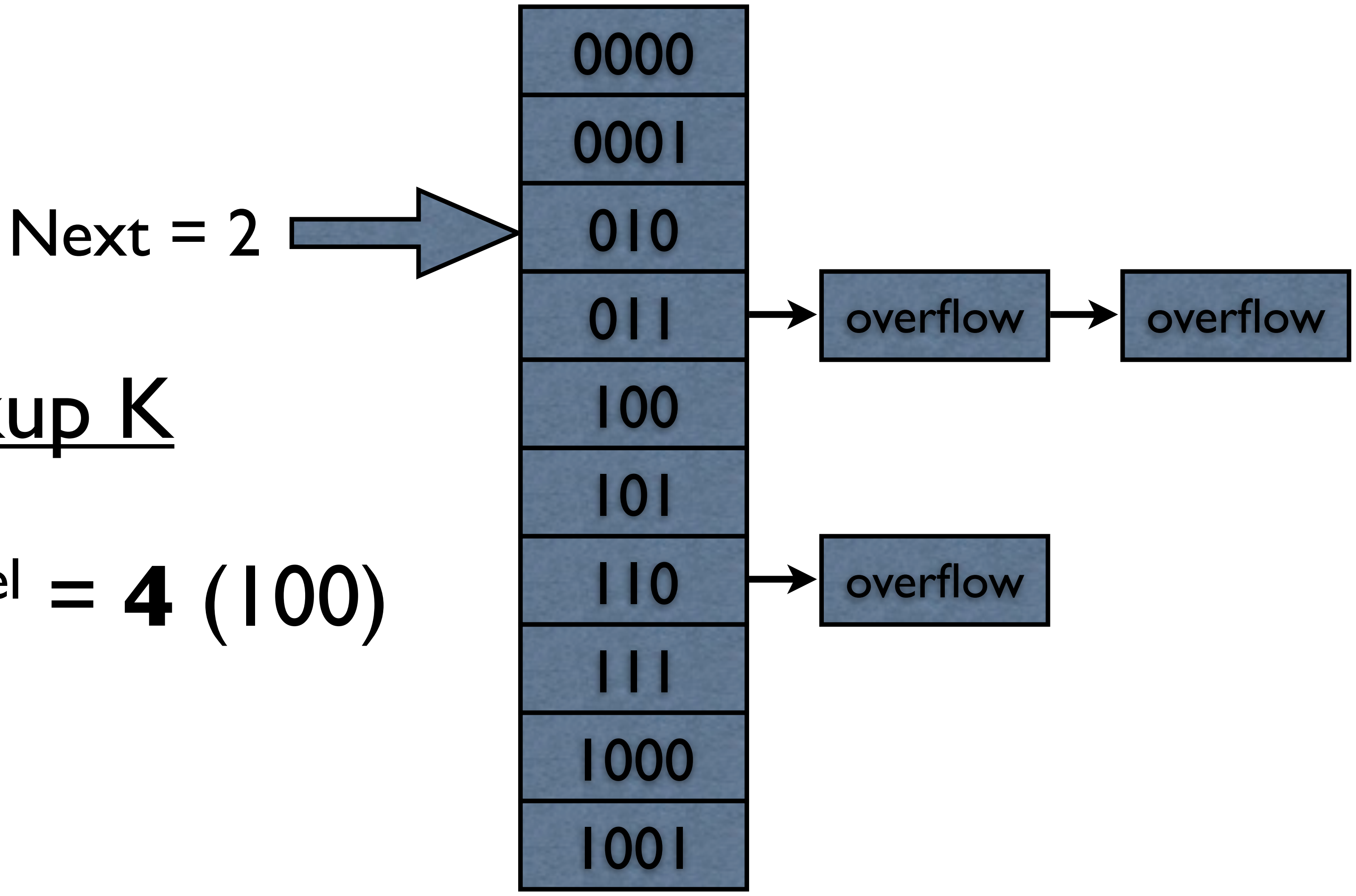
Level = 3 (2^3 = 8 Entries)



Lookup K

Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)

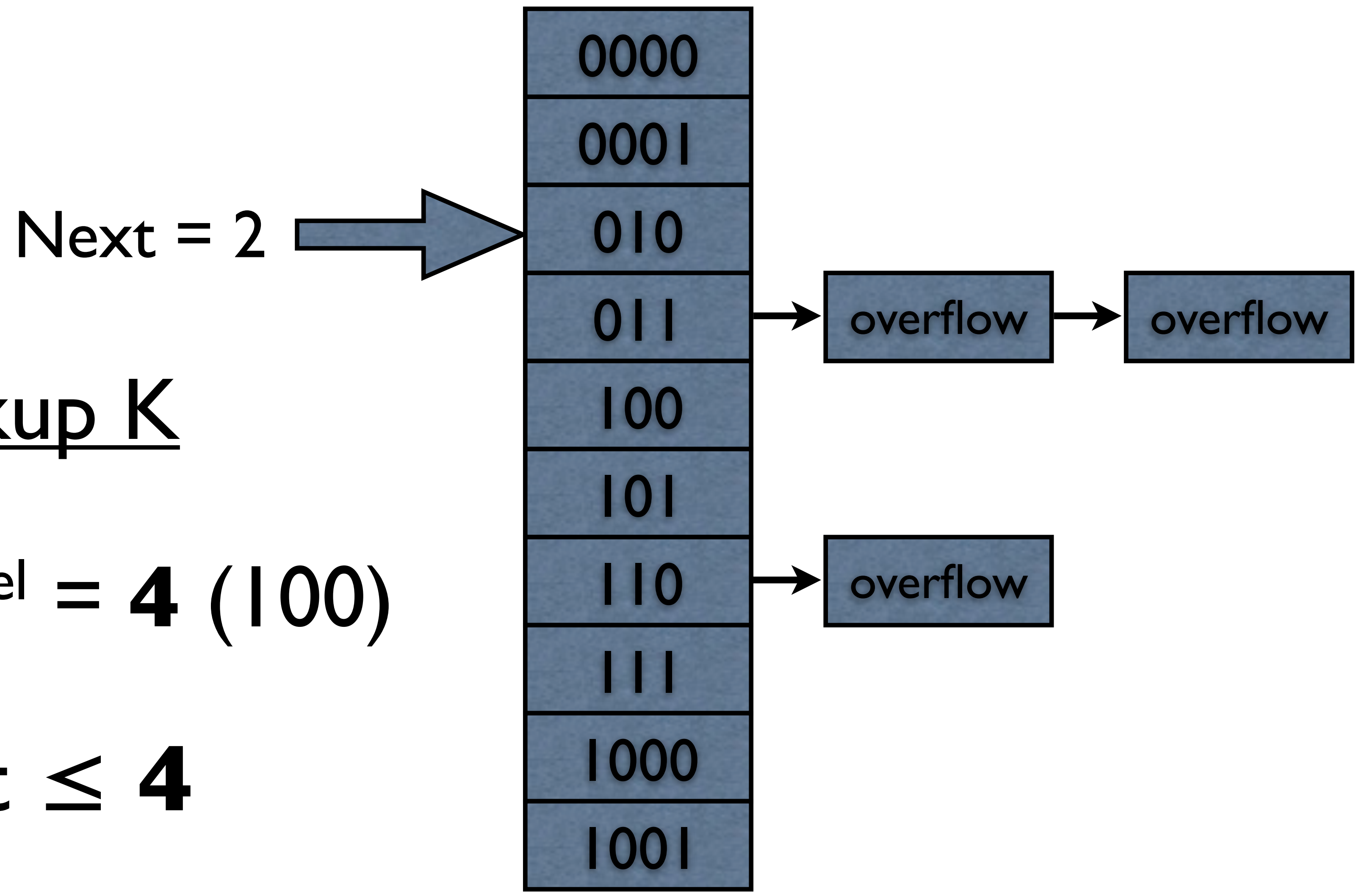


Lookup K

h(k) % 2^{Level} = **4** (100)

Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)



<u>Lookup K</u>

h(k) % 2^{Level} = **4** (100)

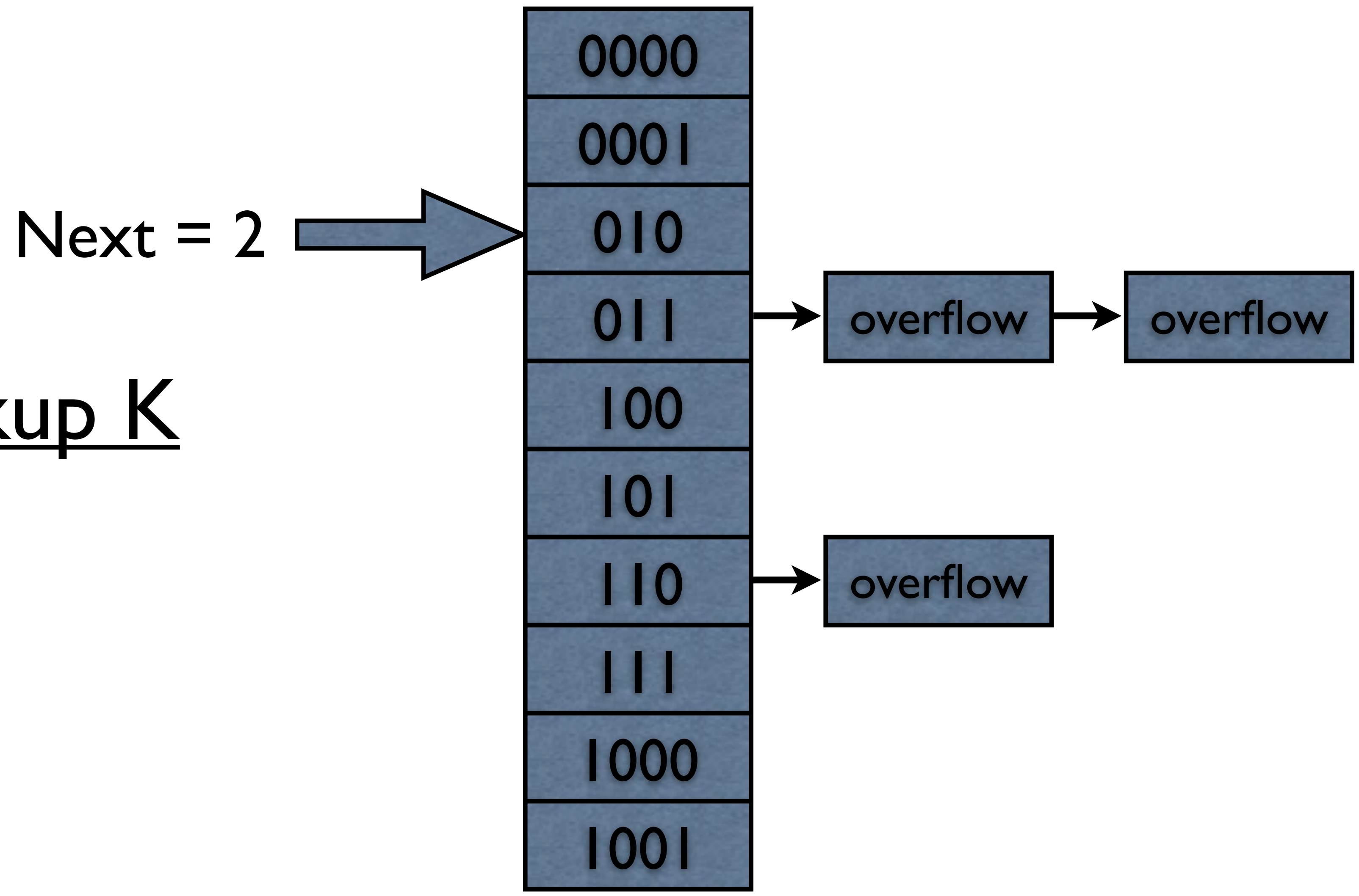
Next ≤ **4**

Use entry 4

Linear Hashing:Lookups

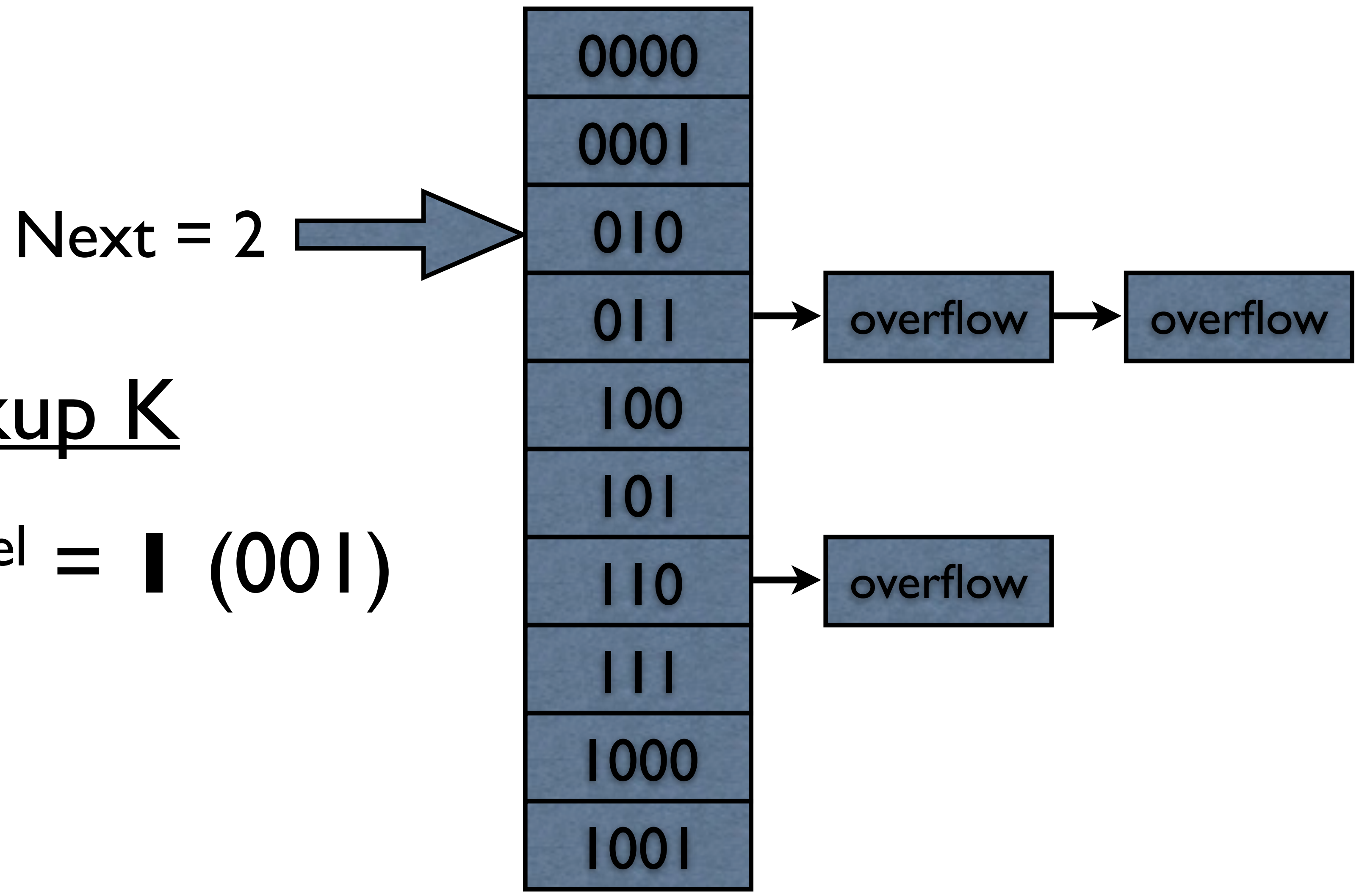
Level = 3 (2^3 = 8 Entries)

Lookup K



Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)

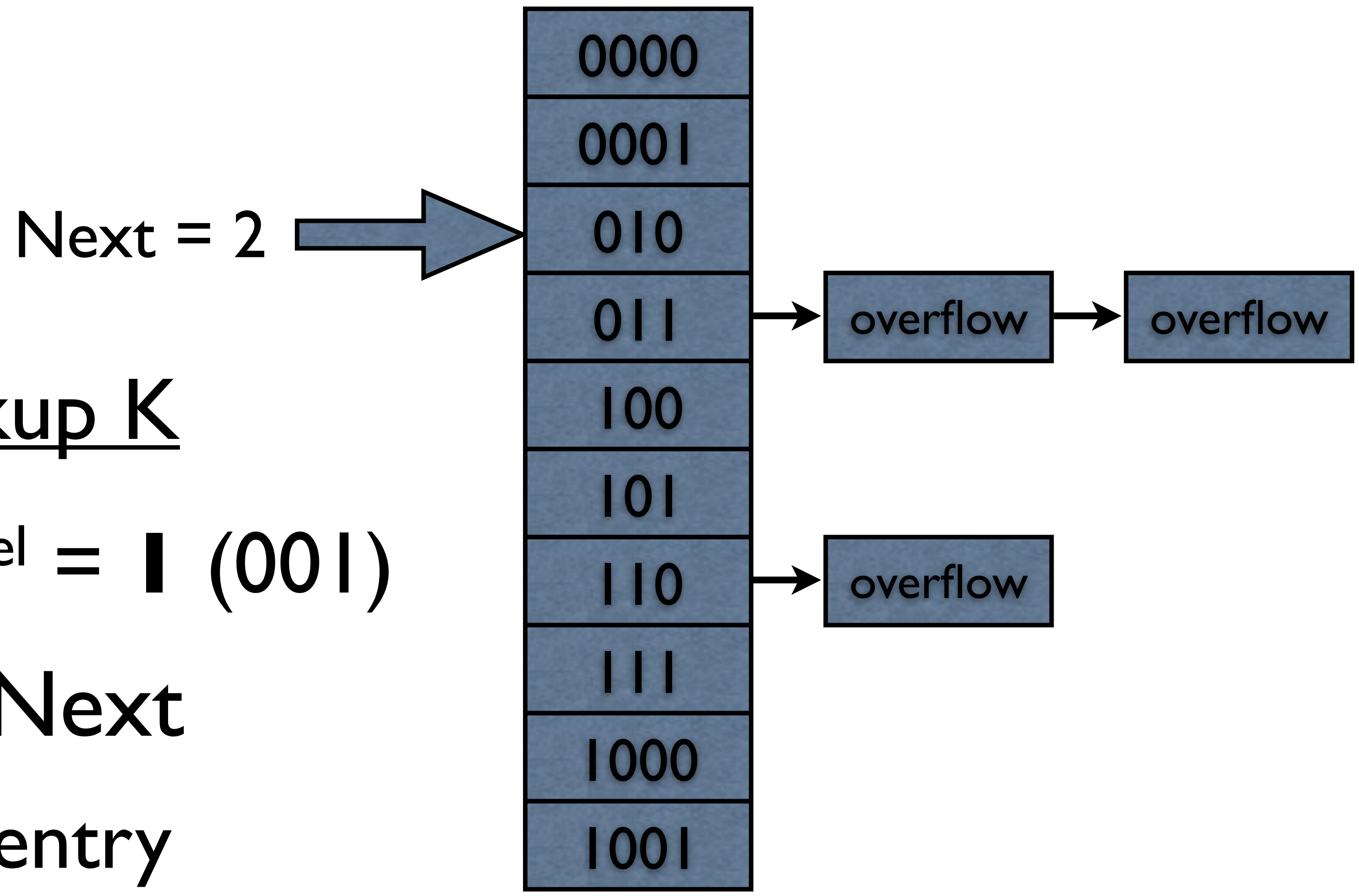


Lookup K

h(k) % 2^{Level} = **1** (001)

Linear Hashing:Lookups

Level = 3 (2^3 = 8 Entries)



<u>Lookup K</u>

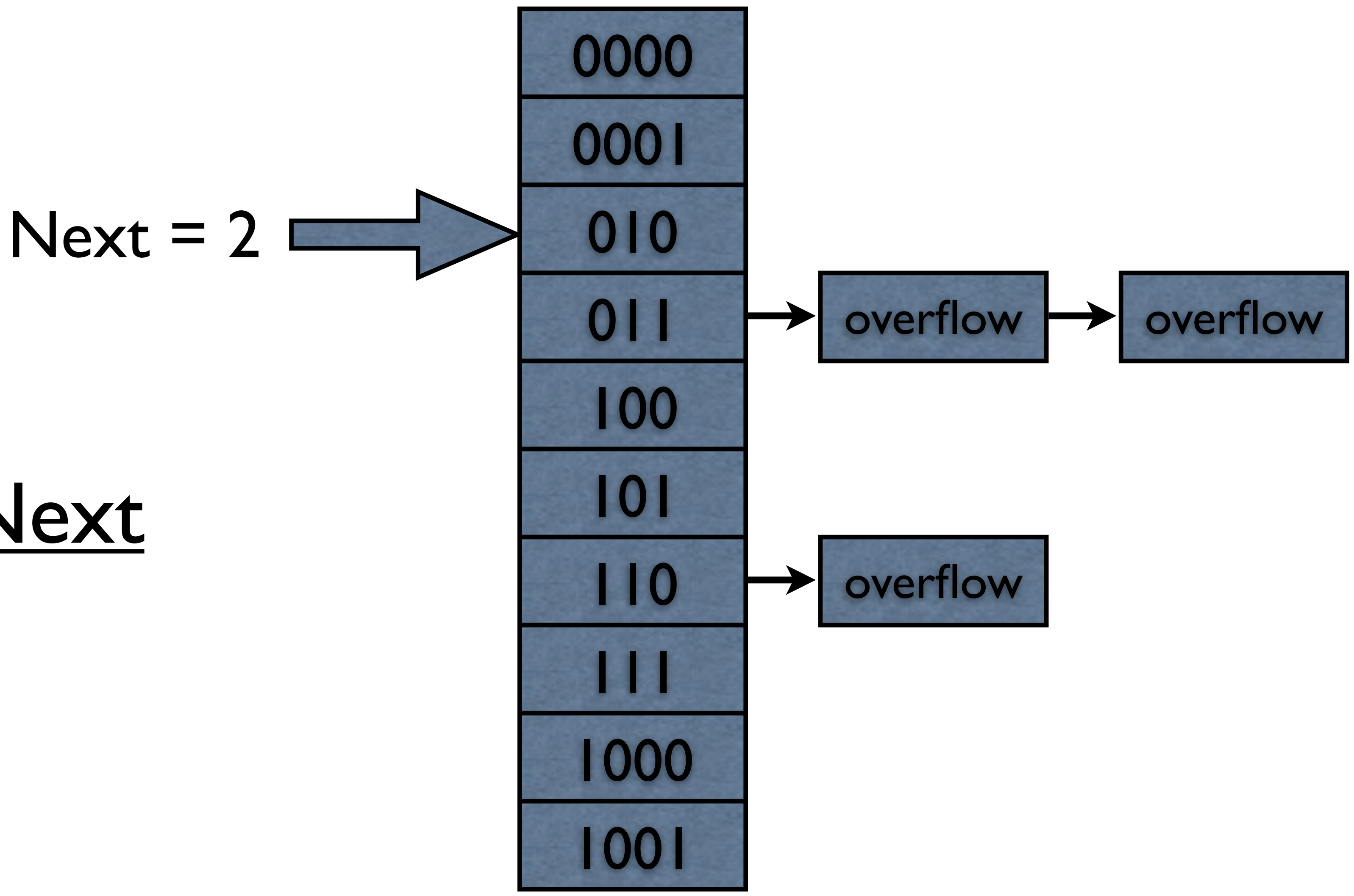
h(k) % 2^{Level} = **1** (001)

1 < Next

Use entry
h(k) % $2^{(\text{Level}+1)}$
1 (1001) or 9 (1001)

Linear Hashing:Splitting

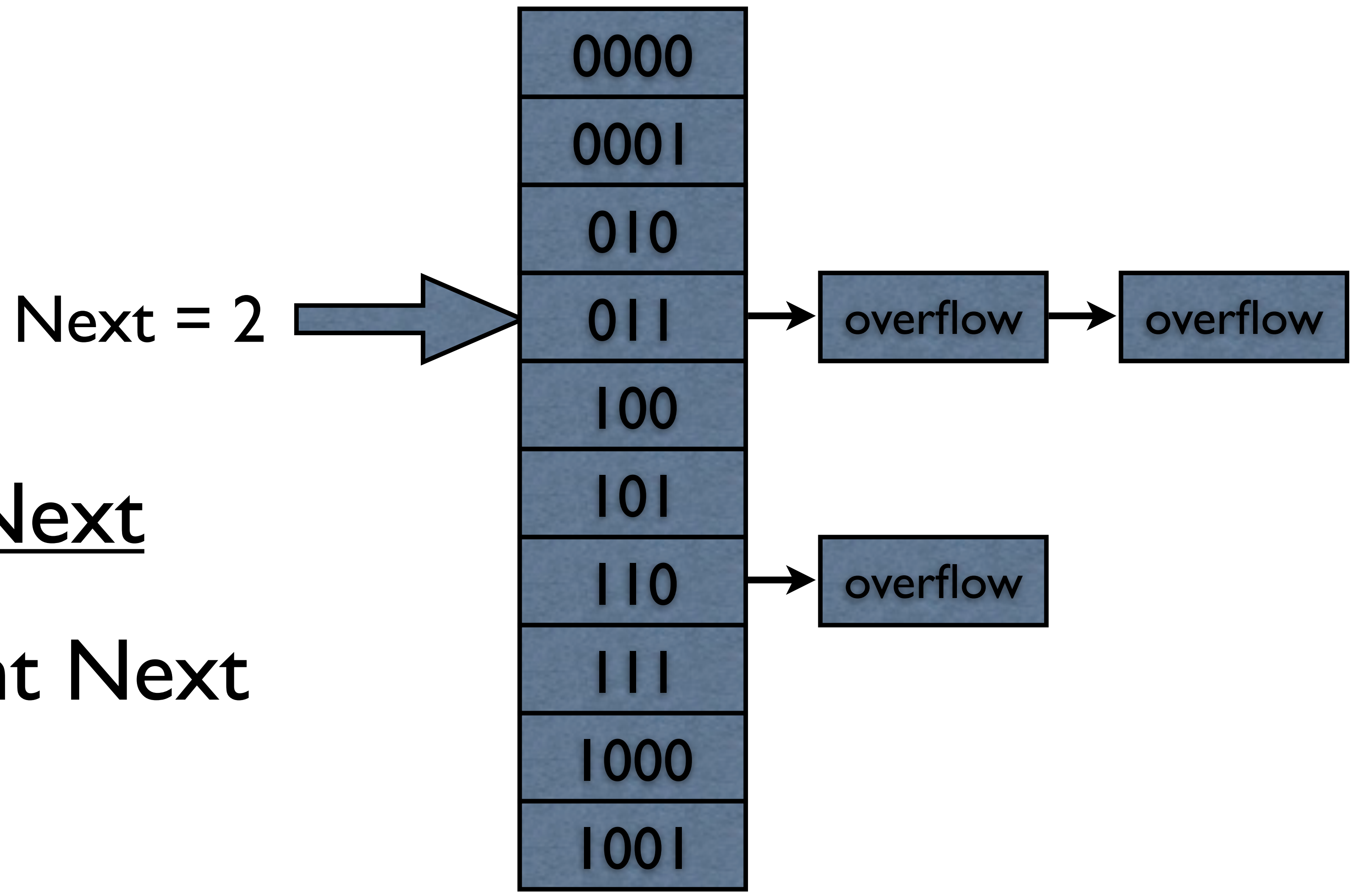
Level = 3 (2^3 = 8 Entries)



Split Next

Linear Hashing:Splitting

Level = 3 (2^3 = 8 Entries)

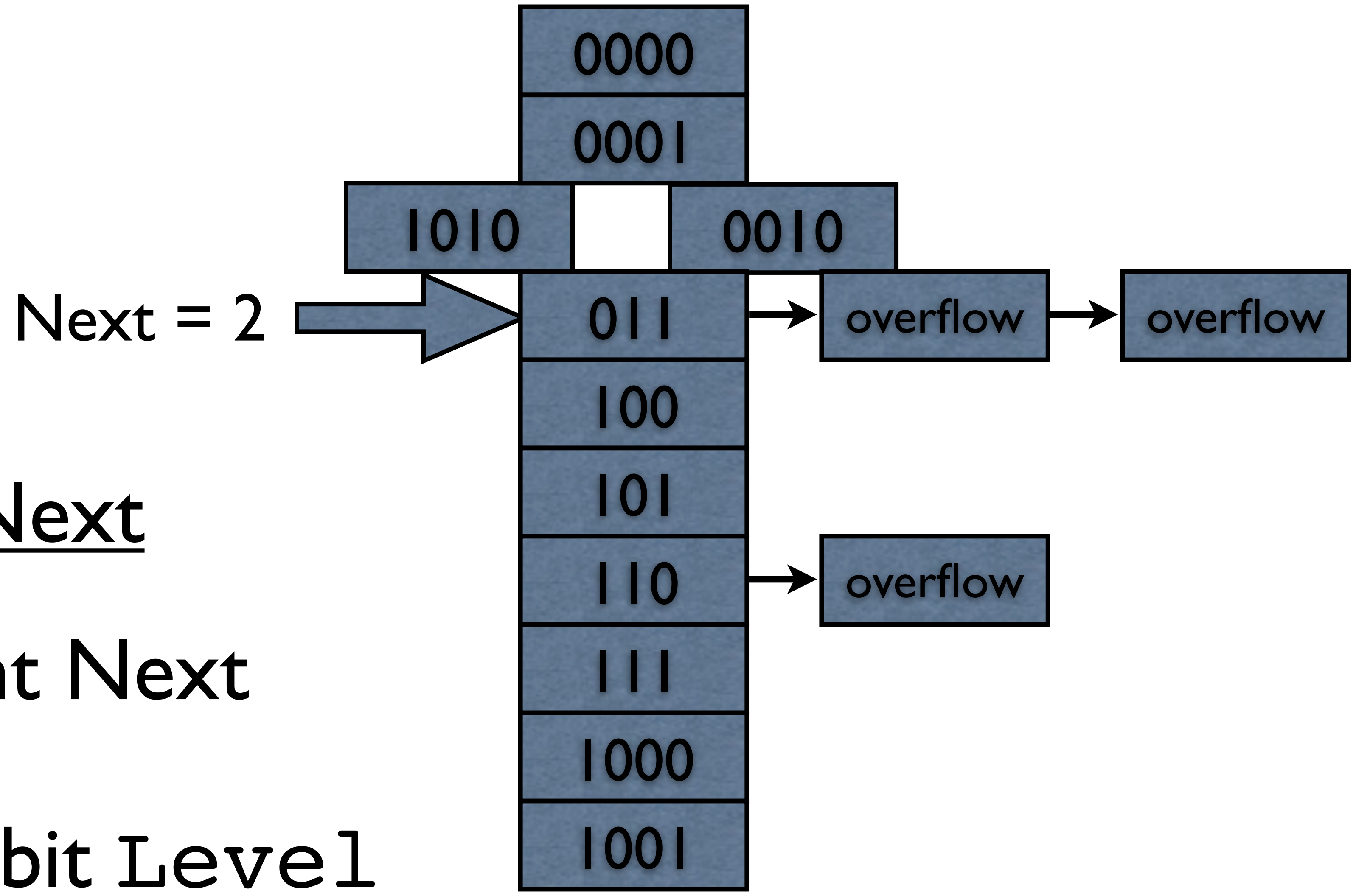


Split Next

Increment Next

Linear Hashing:Splitting

Level = 3 (2^3 = 8 Entries)



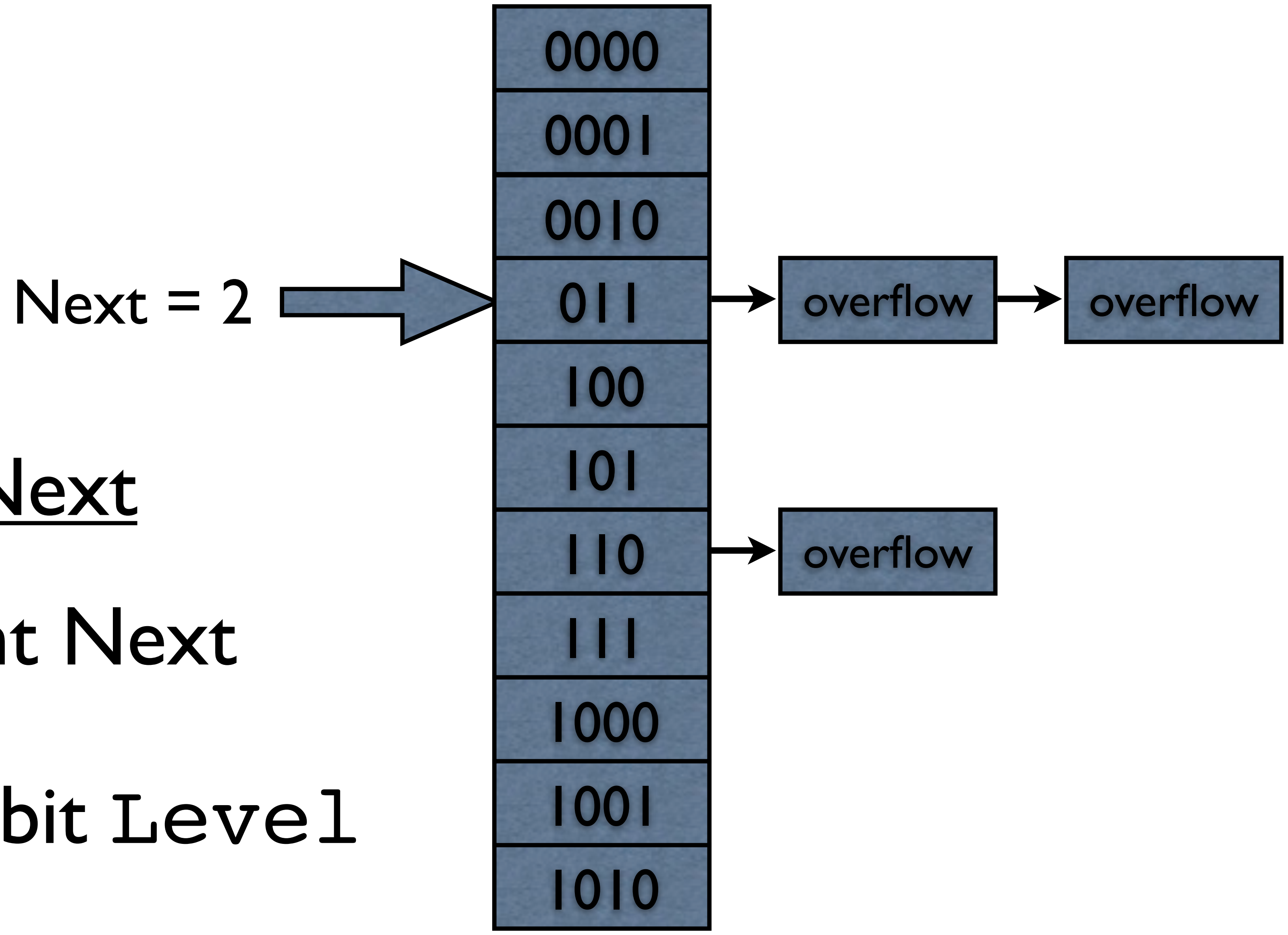
Split Next

Increment Next

Partition on bit `Level`

Linear Hashing:Splitting

Level = 3 (2^3 = 8 Entries)



<u>Split Next</u>

Increment Next

Partition on bit `Level`

- Entries for which the last *Level* bits < *Next*
  - Split, use *Level*+1 bits to determine bucket.
- Entries for which the last *Level* bits ≥ *Next*
  - Unsplit, use *Level* bits to determine bucket.
- Only ever split the *Next* bucket

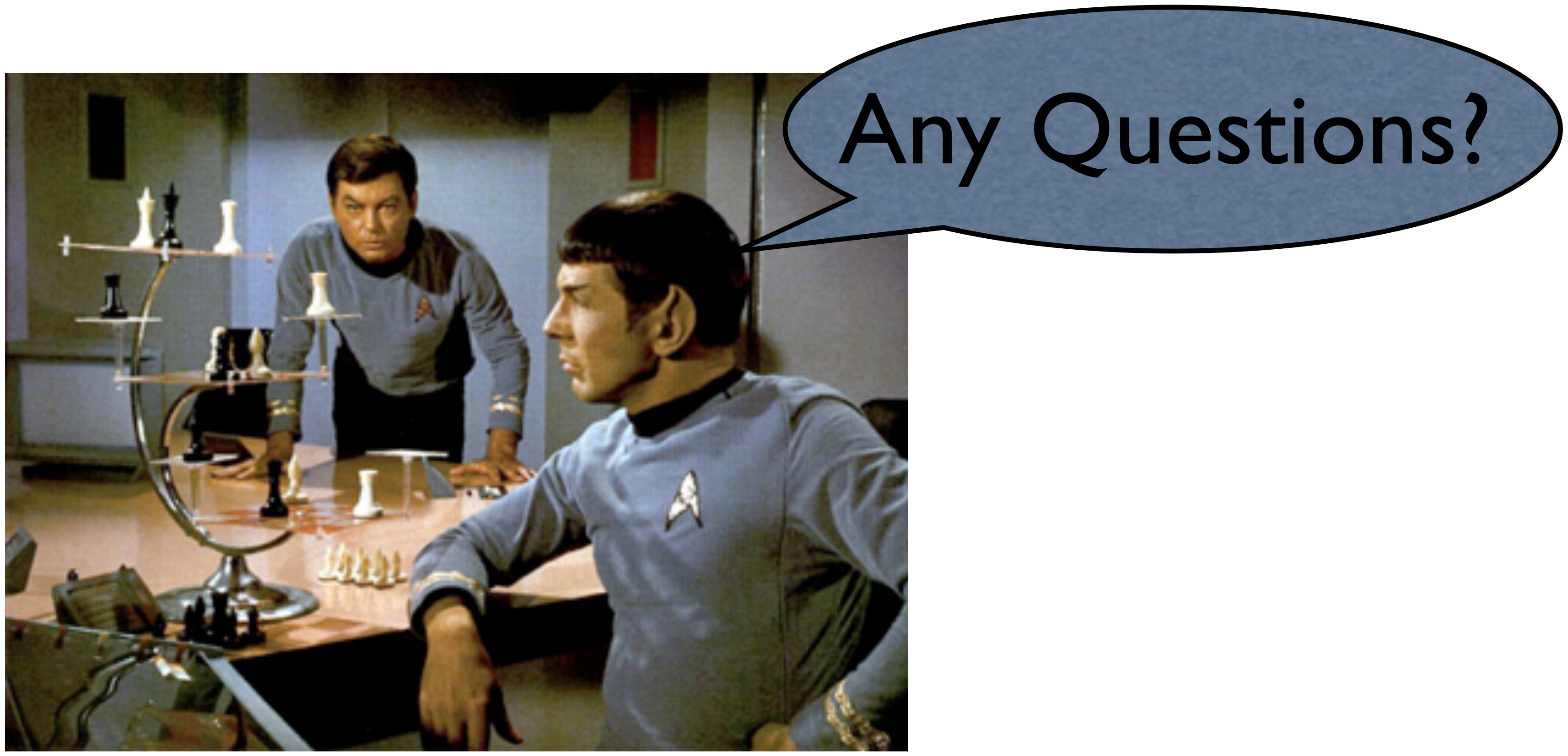


Image copyright: Paramount Pictures

Linear Hashing

- When to we split?
    - It depends on the application.
- Whenever `Next` bucket is full
- After random insertions
- After a fixed number of insertions (size)
- Background process splits as needed.

Extendible vs Linear

- The two algorithms are actually quite similar.
  - Keep some data pages un-split
    - Minimize repartitioning required to split.
  - Use least-significant bits to ensure that new buckets will be appended to the end.
- Linear allocates buckets in sequential order.
  - Is this helpful? When/how?

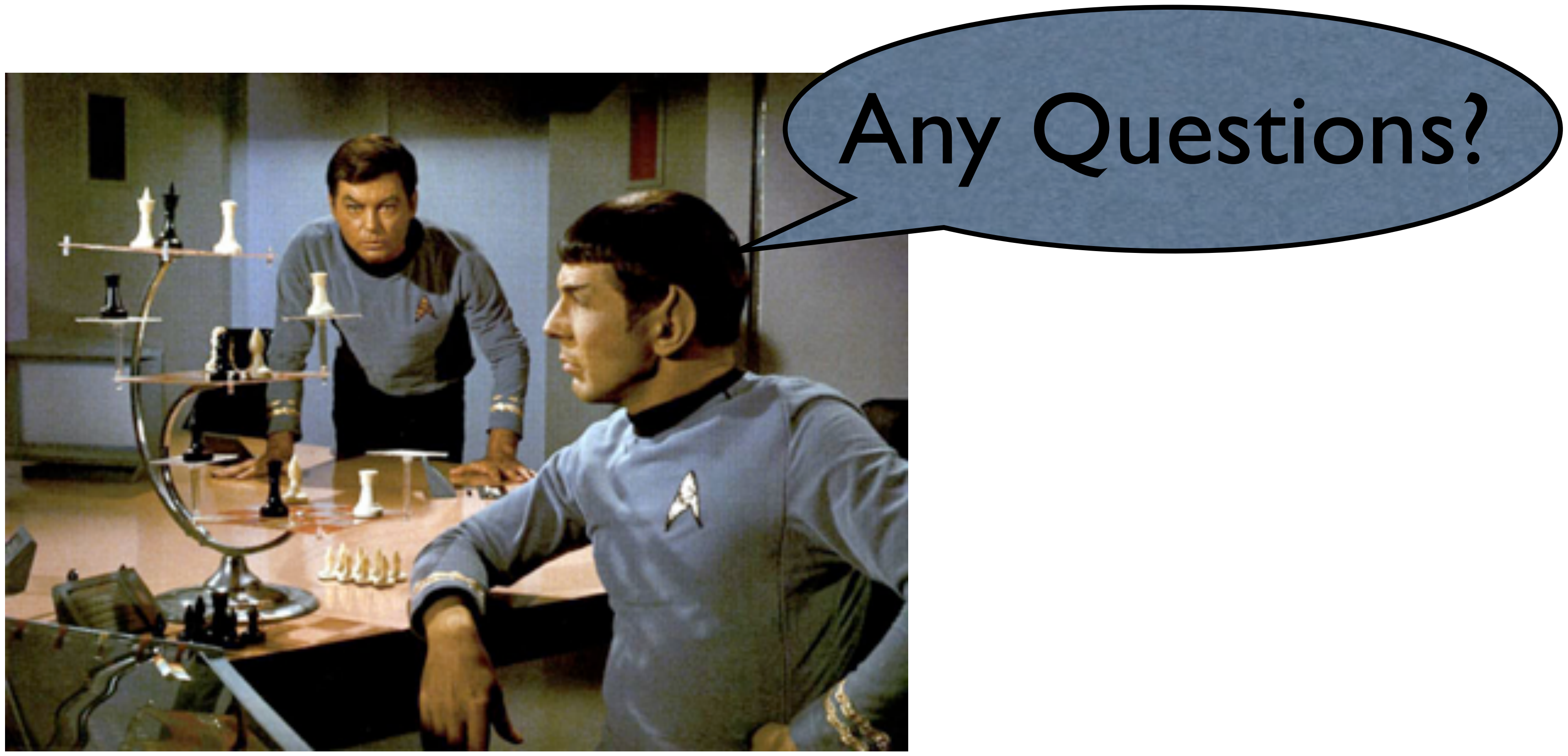


Image copyright: Paramount Pictures

Consistent Hashing

(‘Chord: A Scalable Peer-to-peer…’, Stoica et al.)

- **Insight:** Make split/merge faster by making bin boundaries nondeterministic.
- Used mostly in distributed data-stores
  - (Amazon, Facebook, …)
  - Minimal applications to file-based storage.

Consistent Hashing

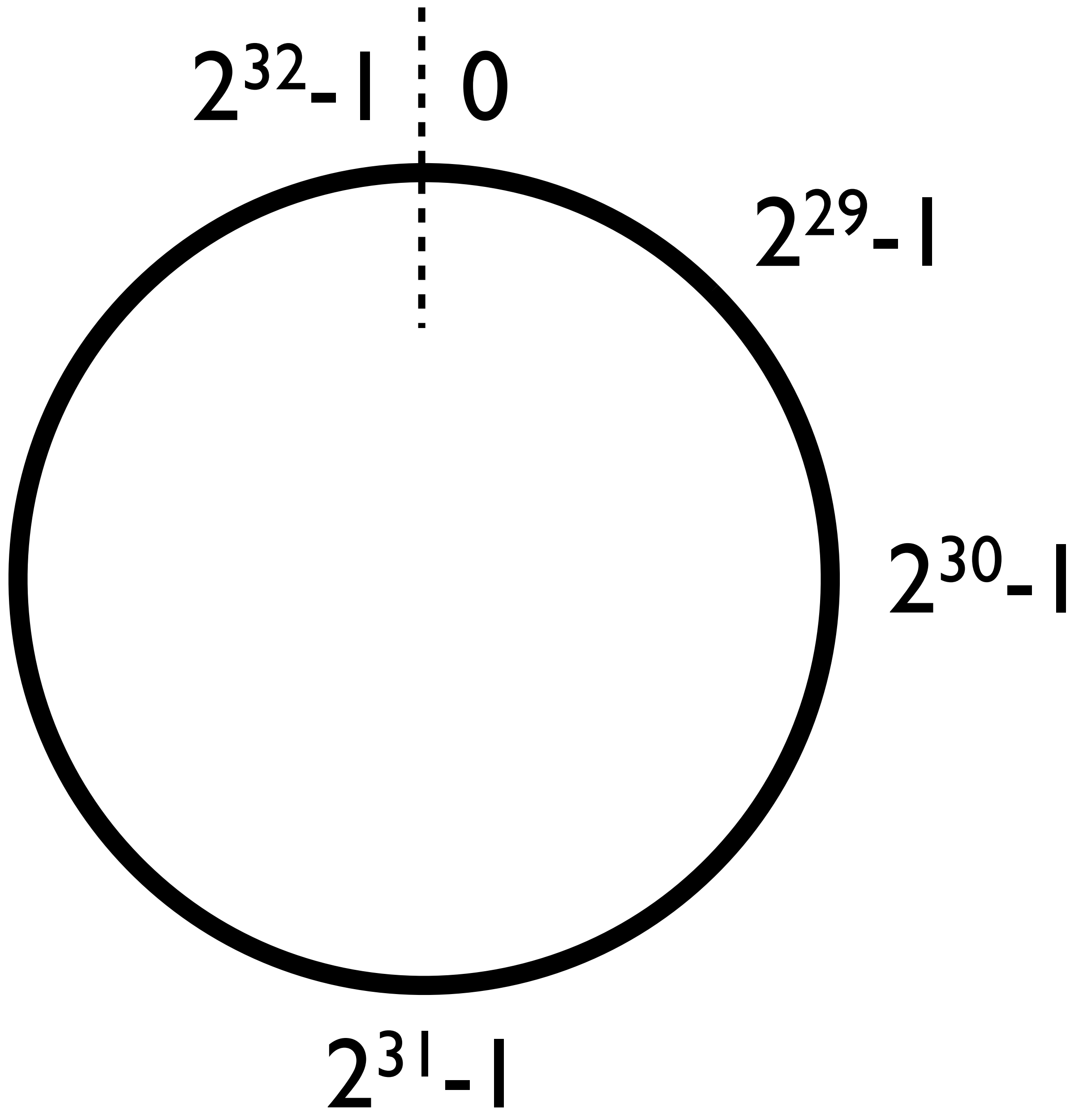
0 $2^{32}-1$

Consistent Hashing

Modular Arithmetic (mod 2^{32})

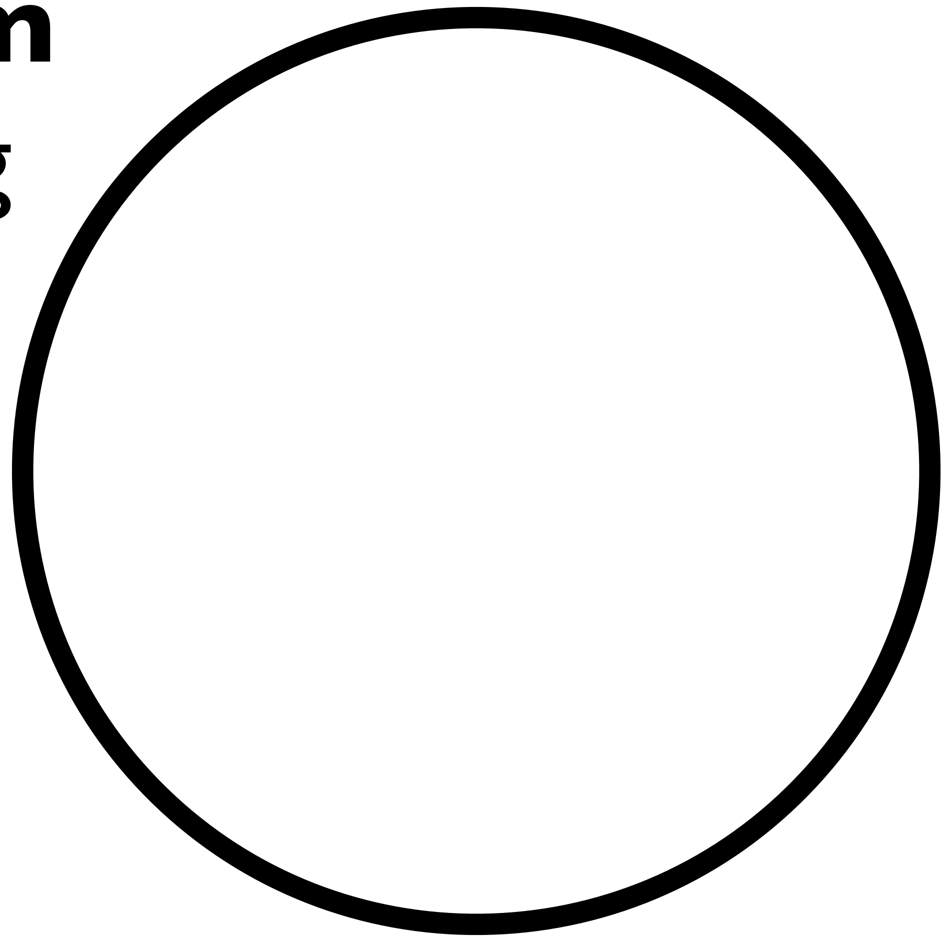
$(2^{32}-1)+1 = 0$

Numbers form a 'Ring'



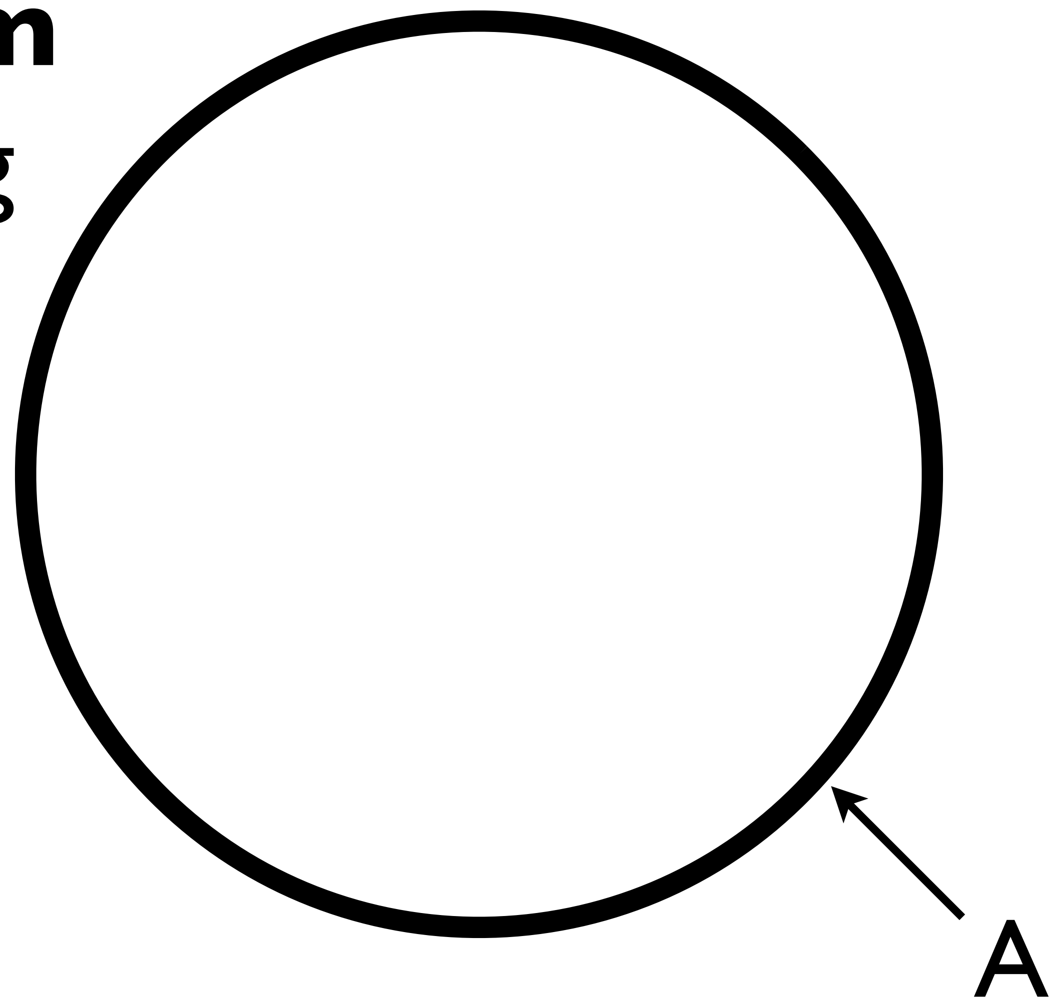
Consistent Hashing

Assign each
bucket a **random**
point on the ring



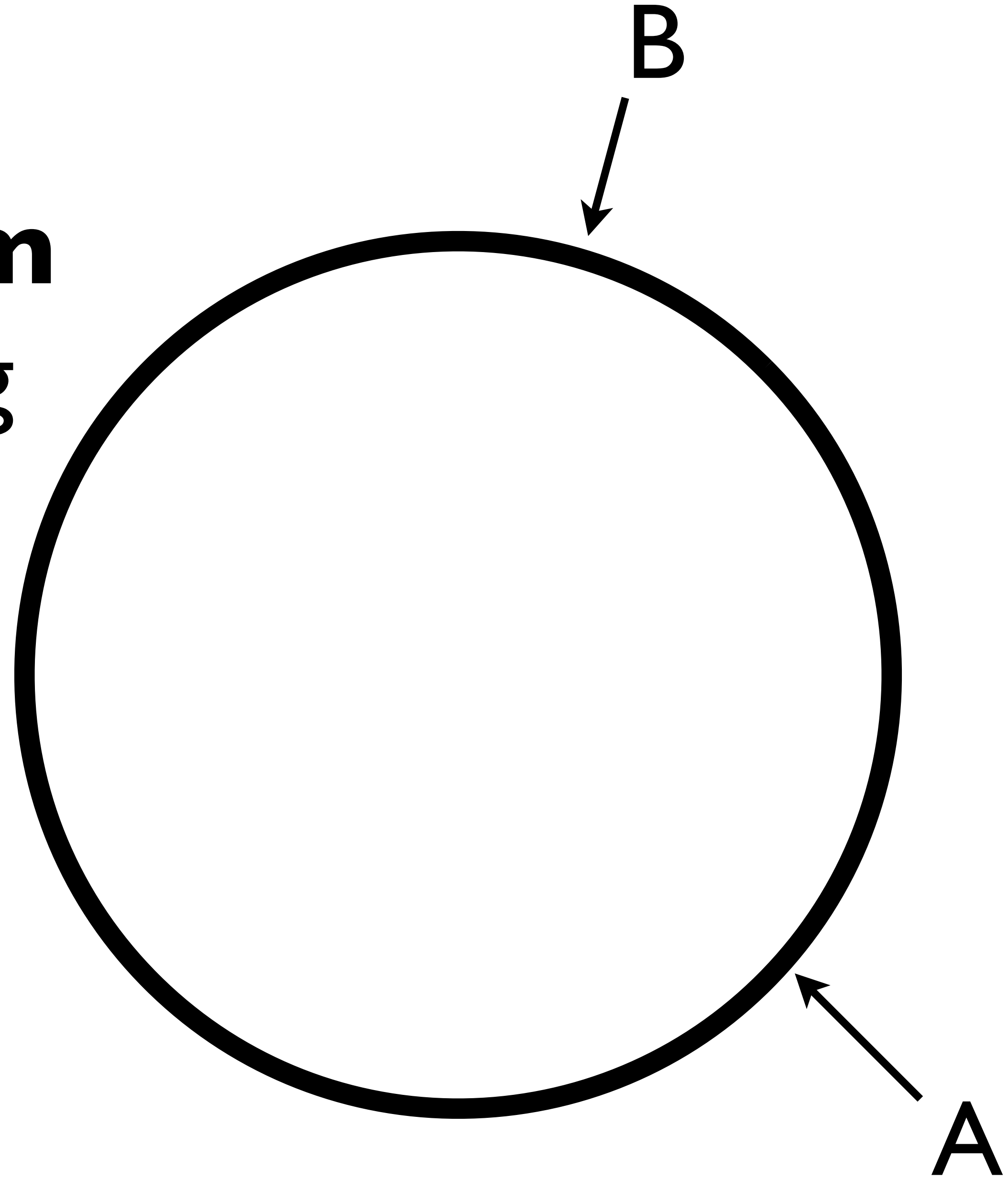
Consistent Hashing

Assign each bucket a **random** point on the ring



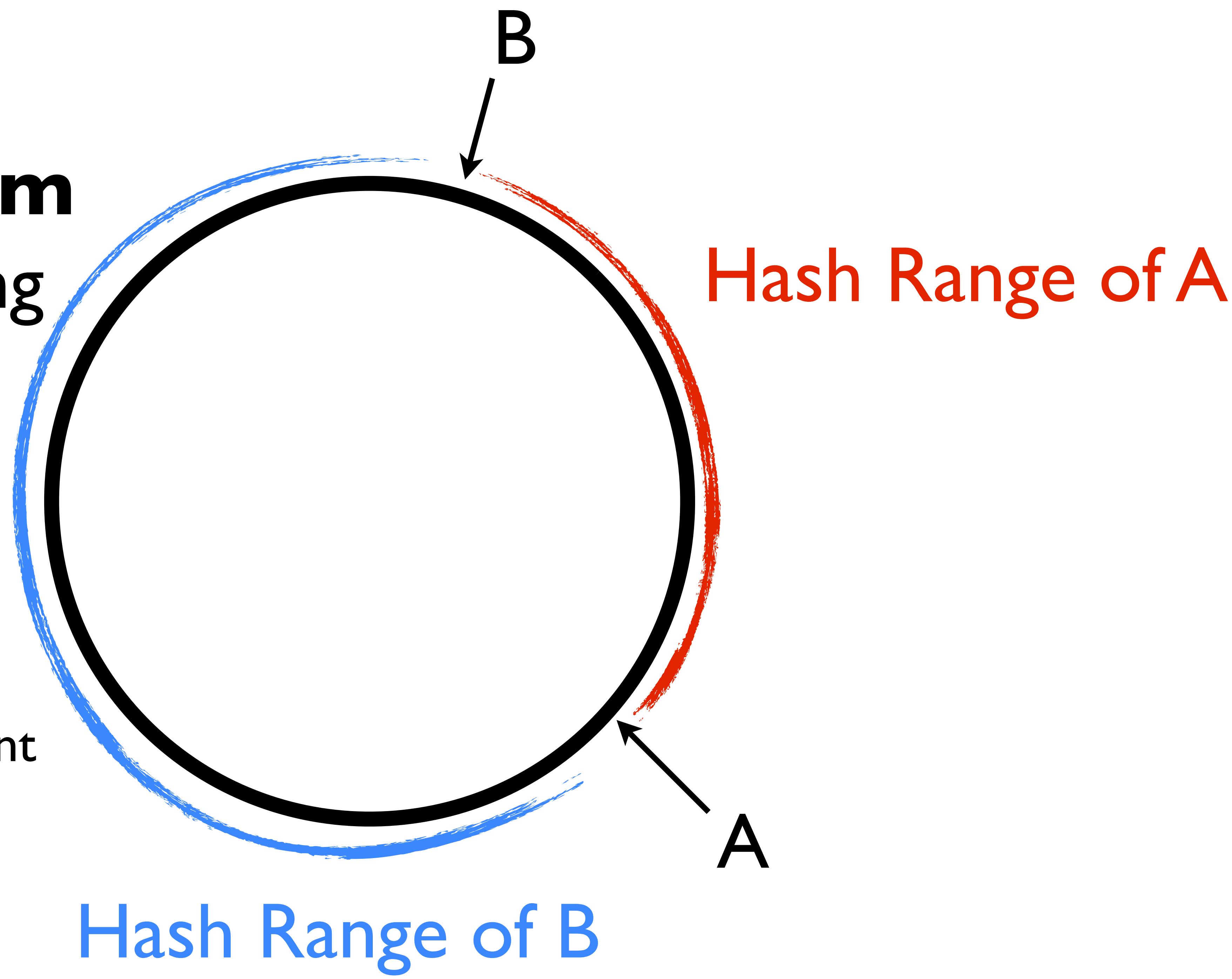
Consistent Hashing

Assign each bucket a **random** point on the ring



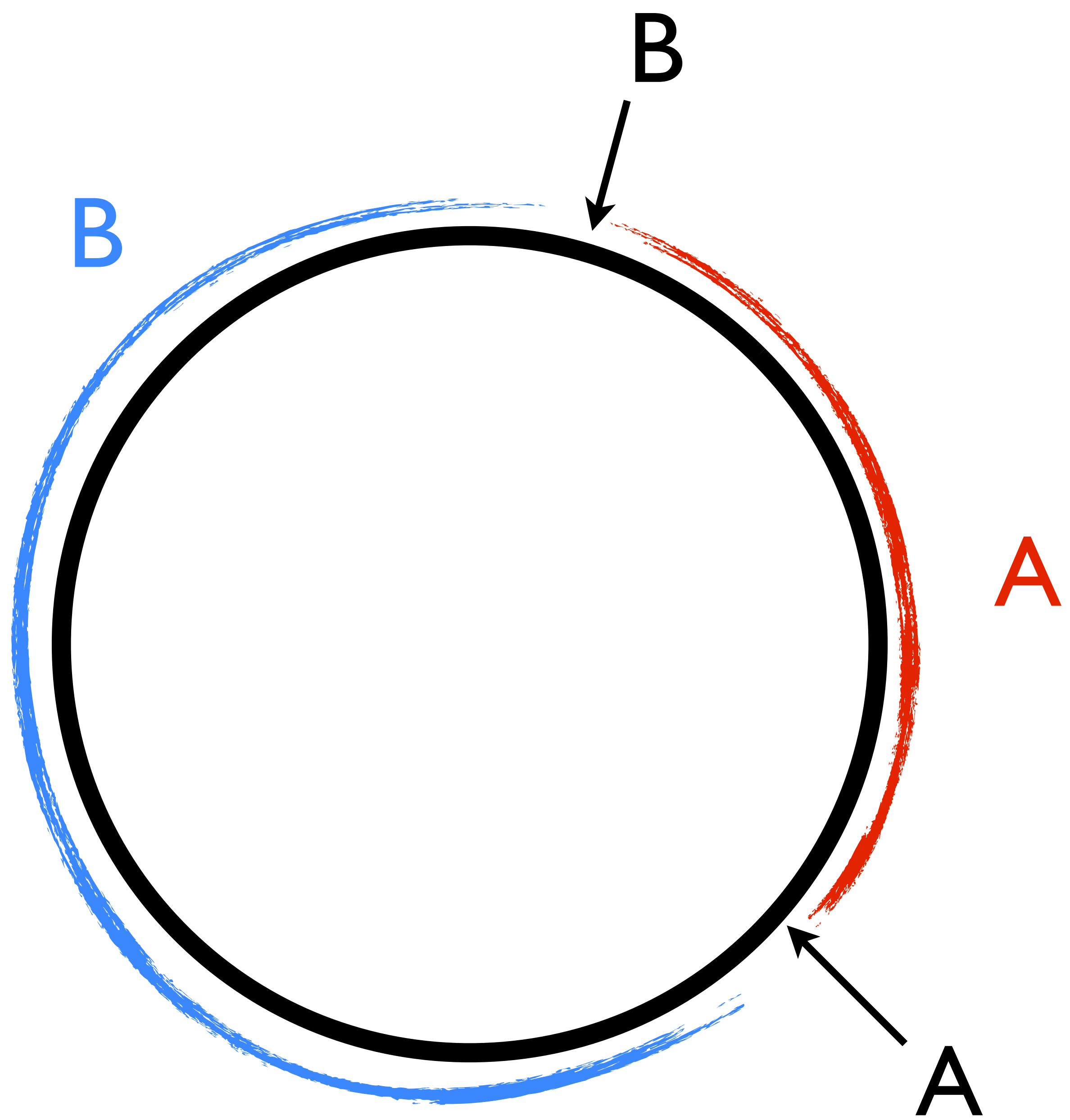
Consistent Hashing

Assign each bucket a **random** point on the ring

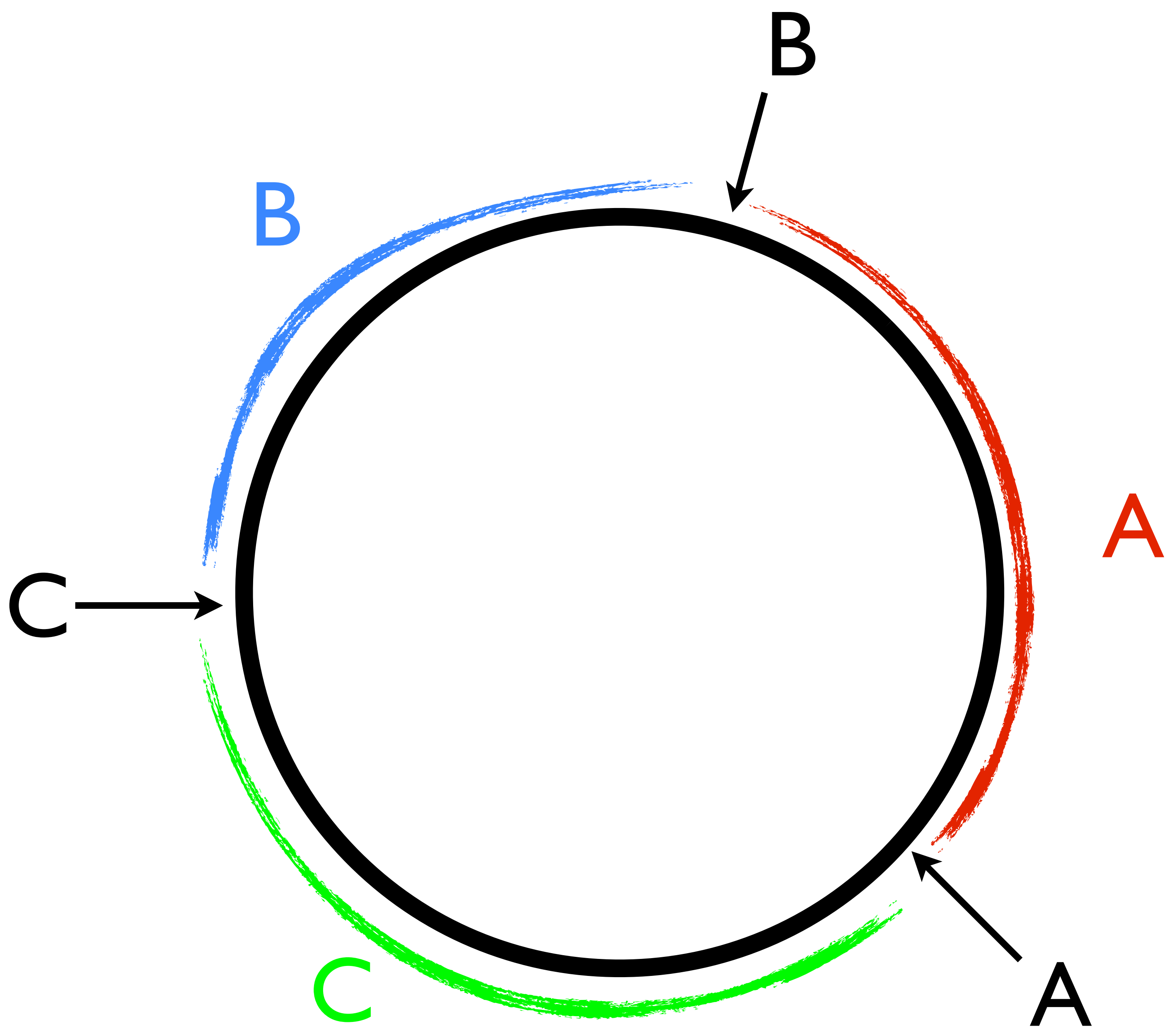


Each bucket contains values that hash to ring positions between its point and its **predecessor**

Consistent Hashing



Consistent Hashing



Consistent Hashing

- Splits/Merges are cheap.
  - At most 2 buckets are affected.
  - No need for page duplication.
- Mapping hash value to bucket is expensive.
  - Need to have a lookup mechanism/directory.
- **Chord**: Decentralized lookup mechanism.

If you're interested, the paper's a fun read
http://dl.acm.org/citation.cfm?id=383071&CFID=178000773&CFTOKEN=34551248

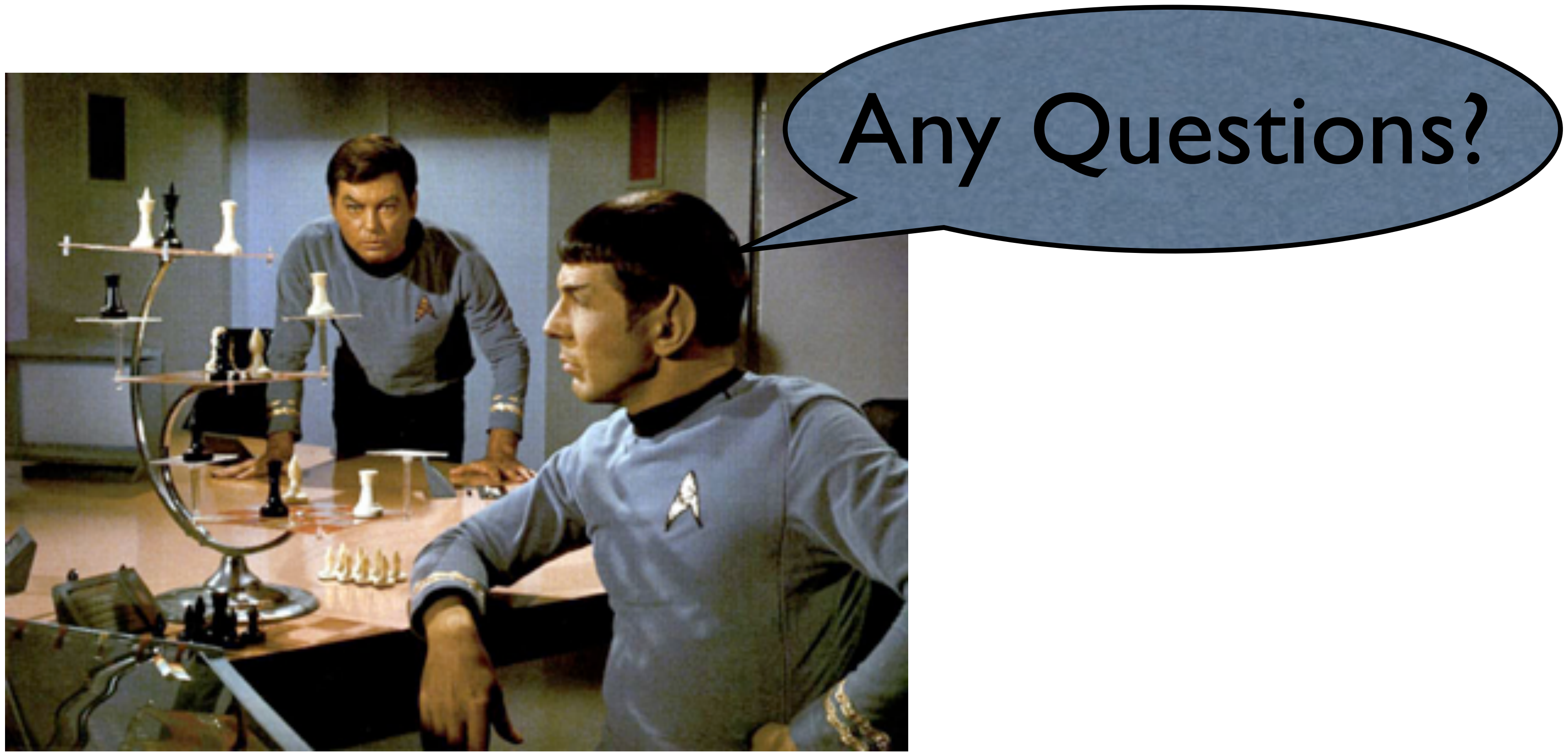


Image copyright: Paramount Pictures

Summary

- Size of a hash table is important
  - Too big: Wasted Space/IOs
  - Too small: Collisions/Overflow Pages
- Dynamic hashing requires carefully managing how data is repartitioned.

Index Keys

- Thus far, we've discussed single-value keys.
- We can also use multi-valued keys <A, B, C, …>
  - Equality Searches: A, B, C, … must all match
  - Range Searches
    - First Compare 'A's.
    - If 'A's equal, compare 'B's
    - If 'A's and 'B's equal, compare 'C's, …

Access Paths

- An access path is a method of retrieving tuples.
  - File Scan, **Scan of an Index** on a *Matching* σ
- A Tree-Index matches (a conjunction of) terms that involve a **prefix** of the search key
  - Does a Tree-Index on <A, B, C> match:
    - A = 5?
    - A = 5 AND B > 6?
    - A > 5 AND B > 6?
    - A < 5 AND A > 3?
    - B > 6?

A = 5 is a prefix defining the range (<5, −∞, −∞>, <5, ∞, ∞>)
A = 5 and B > 6 is a prefix defining the range (<5, 6, −∞>, <5, ∞, ∞>)
A > 5 and B > 6 is not a prefix, because there is no strict lower bound on the range of tuples. That is, if we used <5, 6, −∞> as the lower bound for the index scan, we would still have to apply the selection predicate B > 6 to eliminate tuples such as <6, 4, 3> (which is greater than <5, 6, −∞>, but does not fully satisfy the predicate). Note however, that A > 5 is a prefix, and CAN be used as part of the access path.

Access Paths

- An access path is a method of retrieving tuples.
  - File Scan, **Scan of an Index** on a *Matching* σ
- A Hash Index Matches (a conjunction of) terms that have an equality for **every** attribute in the index.
  - Does a Hash-Index on <A, B, C> match:
    - A = 5?
    - A = 5 AND B = 6?
    - A < 5 AND B = 6 AND C = 4?
    - A = 5 AND B = 6 AND C = 4?

A = 5 is not a match, because we have no unique key value for B or C
A = 5 AND B = 6 is not a match, because we have no unique key value for C
A < 5 AND B = 6 AND C = 4 is not a match, because we have no unique key value for A
A = 5 AND B = 6 AND C = 4 is a match

Access Path Cost

- General Strategy: Find the most **selective** access path to the data
  - The index, file, or combination of both that requires the fewest IOs to access the data.
  - Selection terms that match the index reduce the number of tuples *retrieved*.
  - The remaining terms discard tuples, but do not affect the number of pages fetched.

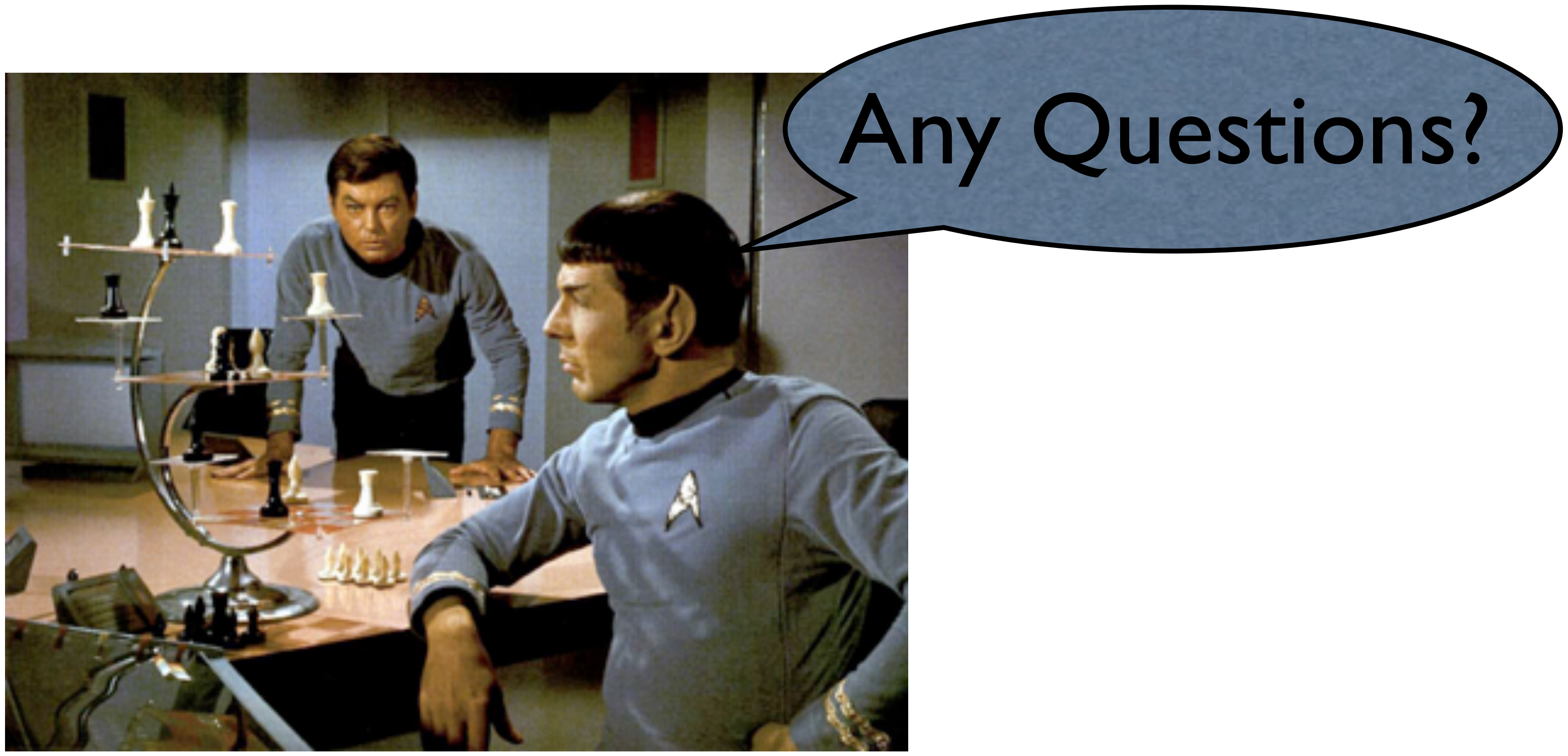


Image copyright: Paramount Pictures