Relational Database System Implementation

CS122 – Lecture 12

Winter Term, 2017-2018

Hash Indexes

- B^+-tree indexes are very effective in most situations
  - Direct access only requires small number of block-reads
  - Also provides sequential traversal of records in search-key order
  - *Most* databases provide B^+-tree indexes
    - Many only provide B^+-tree indexes
- Can also build indexes based on hashing the search key
  - For lookups based on equality, should require a *very* small number of block-accesses
  - Hash indexes cannot improve range-query performance, and do not allow traversal in search-key order

Hash Index Challenges

- Hash-indexes group entries into buckets based on hash function applied to key
- Want our hash function to generate a uniform, random distribution of records when applied to search-key
  - Want all hash buckets to be equally full
- Can fail to achieve this for two reasons:
  - Hash function doesn't map search-key values uniformly
  - May have many records with the same search-key value
- Produces *bucket skew*
  - Some buckets overflow while others still have free space

Hash Index Challenges (2)

- Biggest challenge is how to handle bucket overflows
  - More records map to a bucket than the bucket can hold

- Open addressing (a.k.a. *closed hashing*):
  - If a key hashes to a bucket that is already full, search through hash-table until an empty bucket is found
- Overflow chaining (*open hashing*, *closed addressing*):
  - If a key hashes to a bucket that is already full, link an overflow bucket to the full bucket, and put it in there

Hash Index Challenges (3)

- Generally, open addressing works best in memory
  - Can often exploit memory caches very effectively
  - On disk, requires *many* block IOs
- Hash file organizations and hash index files usually rely on overflow chaining
  - Requires far fewer disk accesses than open addressing
  - Also tends to simplify hash-table restructuring over time
- If bucket overflow reaches a certain level, simply want to increase number of buckets in hash structure
  - Ideally, will relieve overflow issues, but it doesn't always do so (e.g. if many records have same key-value)

Static Hashing

- Previously discussed *static hashing*:
  - Number of buckets n_b is fixed when hash file is created
  - Apply a hash function $h(K)$ to produce a bucket b, $0 \leq b < n_b$
    - Entry is stored in this bucket
  - If a bucket overflows, use chaining to store overflow records
- Static hashing isn't effective if number of entries increases over time, or if distribution of key-values changes over time
  - Performance will slowly degrade as number of overflow records increases
  - Rehashing entire file to increase n_b will be *very* slow, requiring many disk IOs to complete

Dynamic Hashing

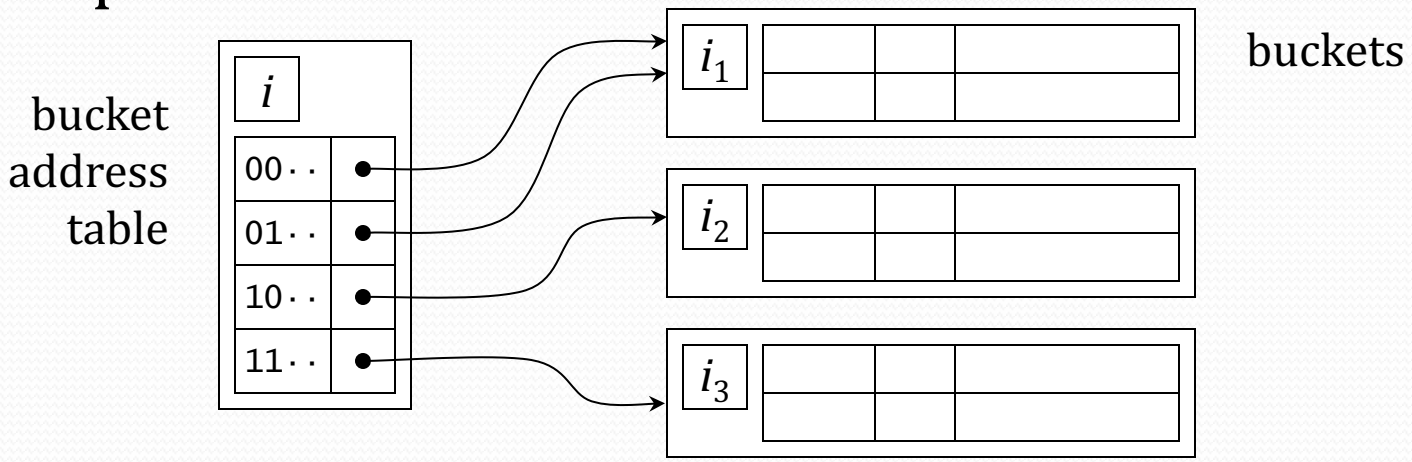
- To handle hash indexes that change over time, need some form of *dynamic hashing* mechanism
- Idea: make *incremental* changes to hash structure, instead of large-scale changes
  - Increase number of buckets by small amount as needed; change hash function slightly to accommodate change
  - Amount of data that must be rehashed is kept very small, so maintenance overhead is acceptably low
- Today, two dynamic hashing mechanisms:
  - *Extendable hashing* and *linear hashing*

Extendable Hashing

- In static hashing, $h(K)$ maps keys to a (smallish) fixed number n_b
  - e.g. $h(K) = h'(K)$ **mod** n_b
  - Not easy to change this hash function incrementally
- For extendable hashing, choose a hash function that produces a wide range of values
  - $h(K)$ produces b-bit integers, e.g. where $b = 32$
- Provide a mapping from hash values to buckets, using a *bucket address table*
  - Can incrementally change this table as needed

Bucket Address Table

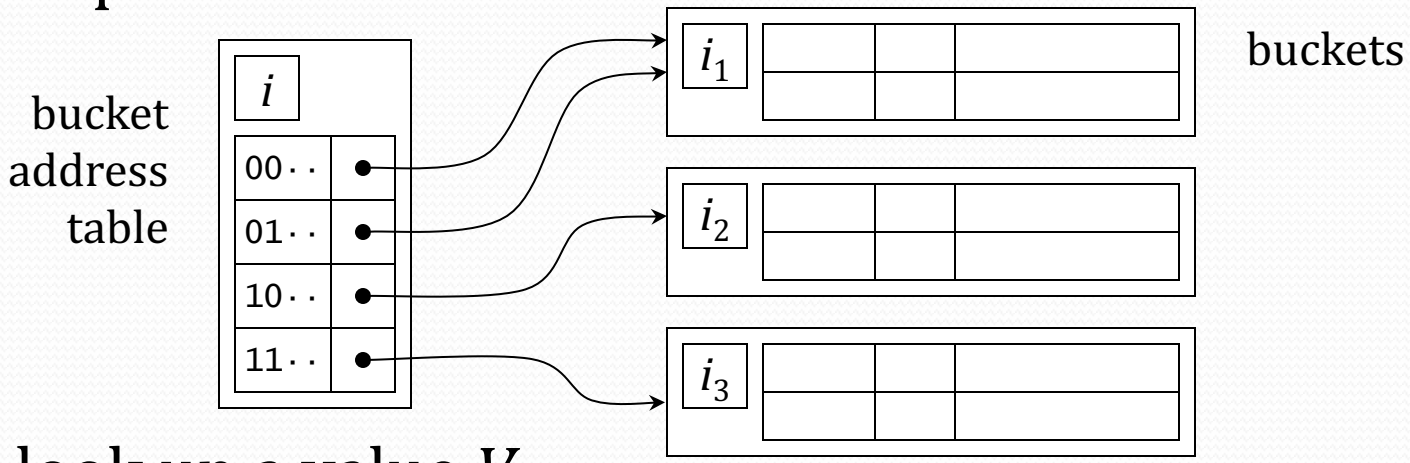
- The bucket address table maps the top *i* bits of hash values into buckets that hold those entries
- Example: $i = 2$



- Bucket address table has 2^i entries in it
  - Note: may be fewer than 2^i buckets

Bucket Address Table (2)

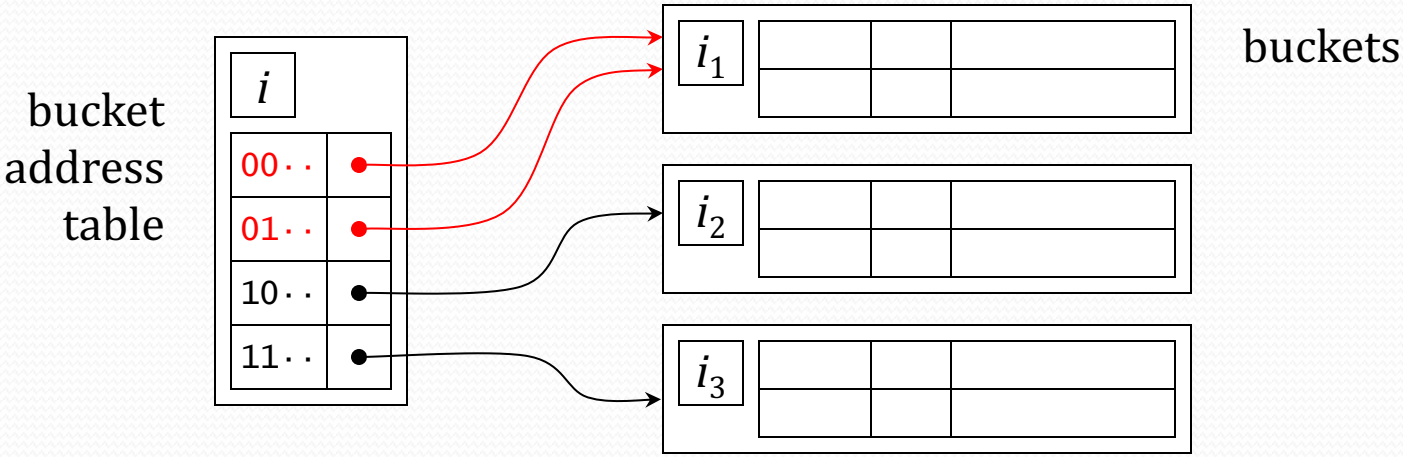
- Example: $i = 2$



- To look up a value V:
  - Compute $h(V)$
  - Use the top i most significant bits to look up address of bucket that will contain records with search-key value V

Adjacent Bucket-Table Entries

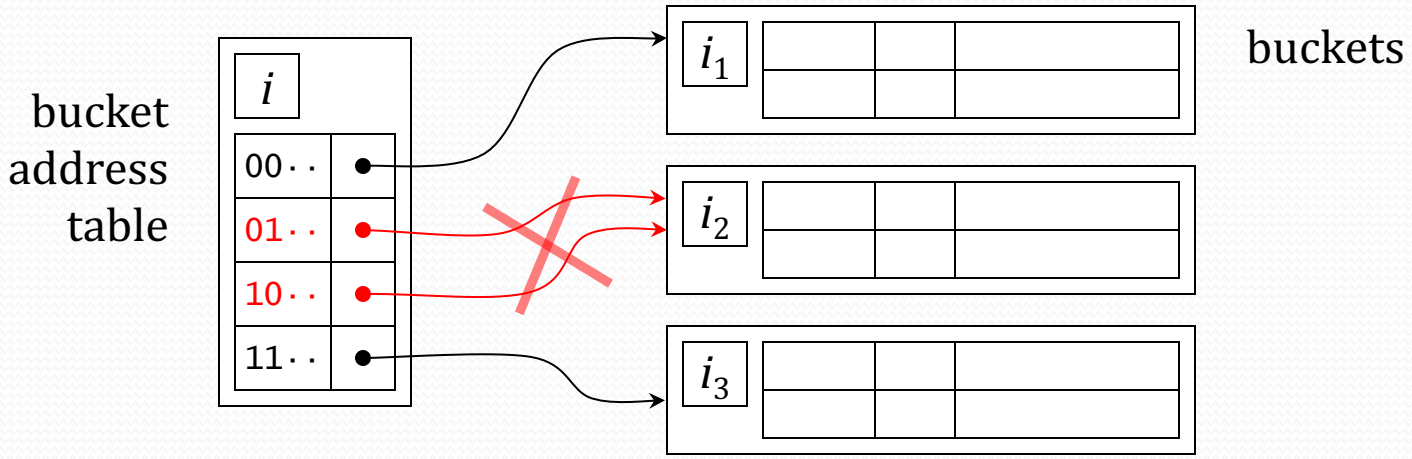
- Example: $i = 2$



- Adjacent table entries can reference the same bucket, if they share a common bit-prefix
  - Here, first two entries share hash-code prefix "0⋯", so they can refer to the same bucket

Adjacent Bucket-Table Entries (2)

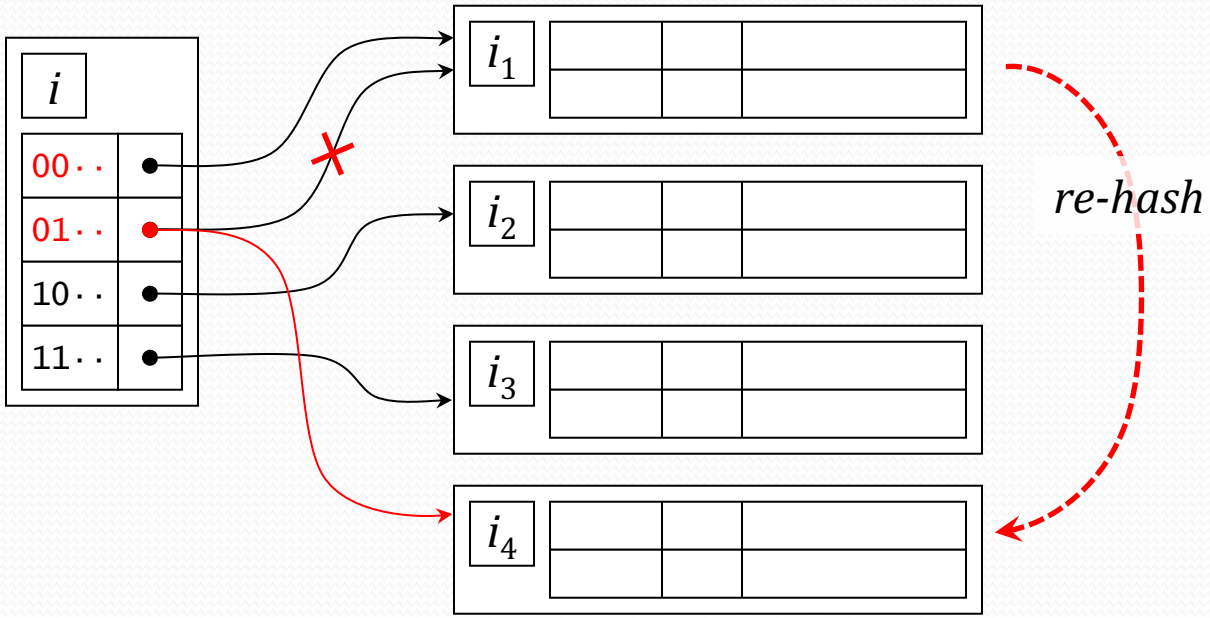
- This configuration is <u>invalid</u>:



- Entries don't have a common prefix

Adjacent Bucket-Table Entries (3)

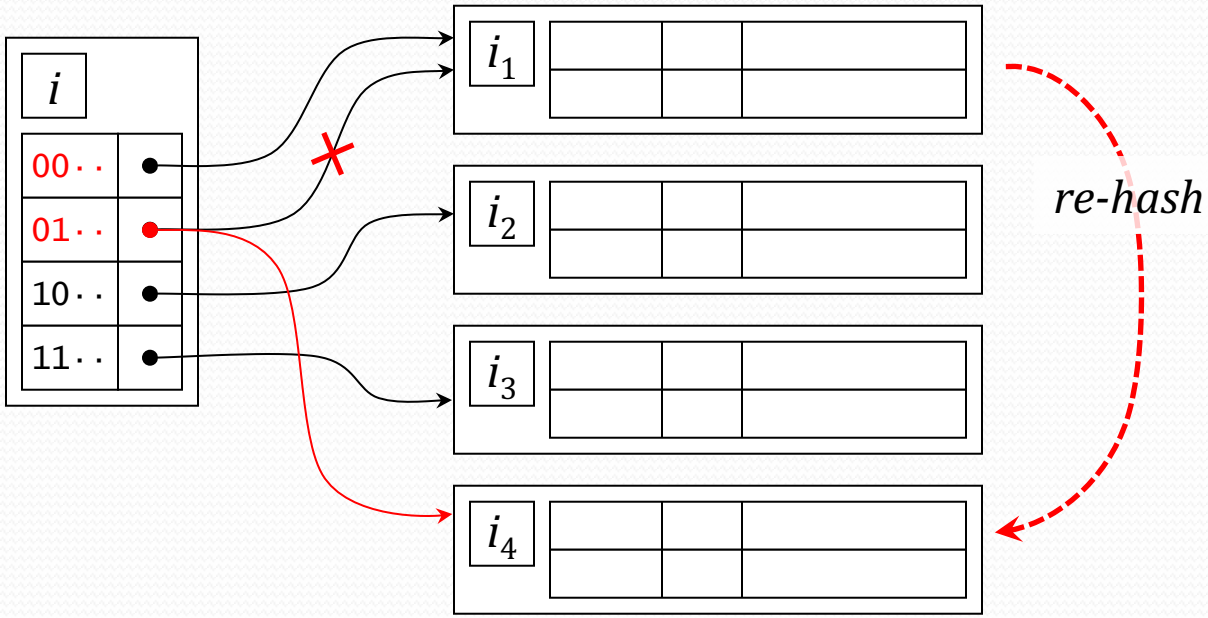
- Example: $i = 2$



- Adjacent entries don't affect lookup procedure at all…
- Does allow us to split a bucket into two, when it overflows!
  - Simply create a new bucket, update bucket address table, and rehash contents of overflowed bucket

Adjacent Bucket-Table Entries (4)

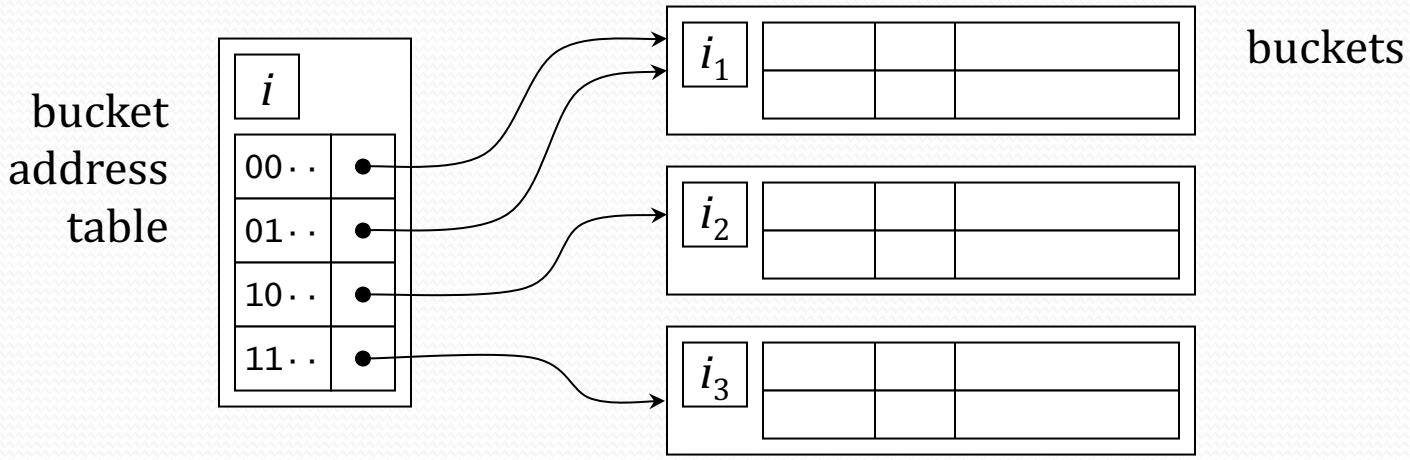
- Example: $i = 2$



- When a bucket is split, we are examining one more bit in the hash-code (e.g. "0⋯" values are split into "00⋯", "01⋯")
  - If hash function is uniform/random, should see approx. half of records stay in old bucket, and half move to new bucket

Extendable Hash Buckets

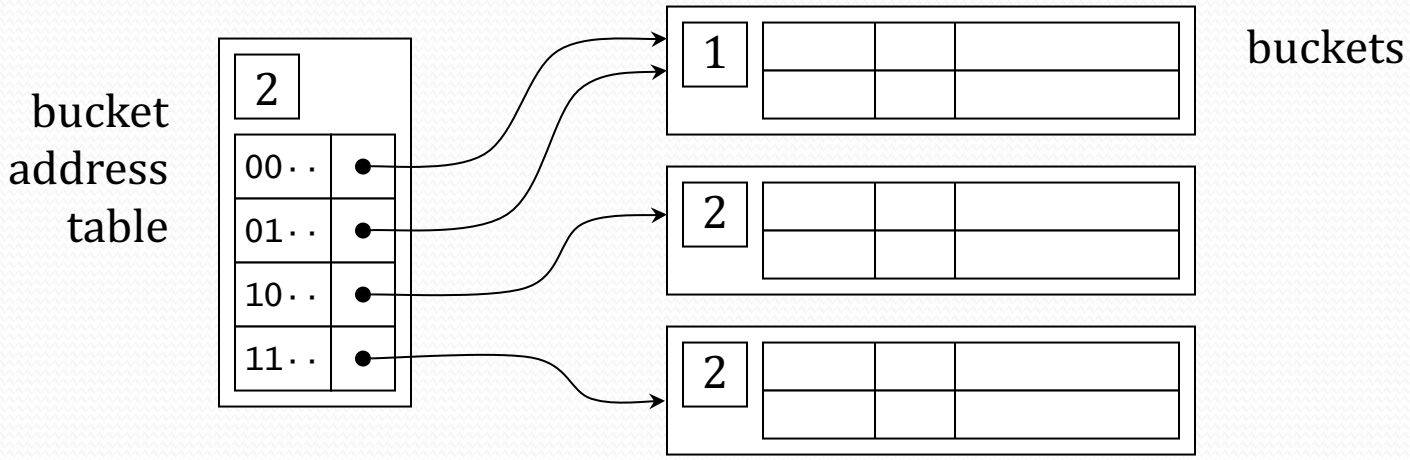
- Each hash-bucket j has its own i_j:



- Specifies the *actual* size of the hash-prefix used for that bucket (i.e. number of leading bits used from hash value)
  - In our example: $i = 2$
  - $i_1 = 1$, $i_2 = 2$, and $i_3 = 2$

Extendable Hash Buckets (2)

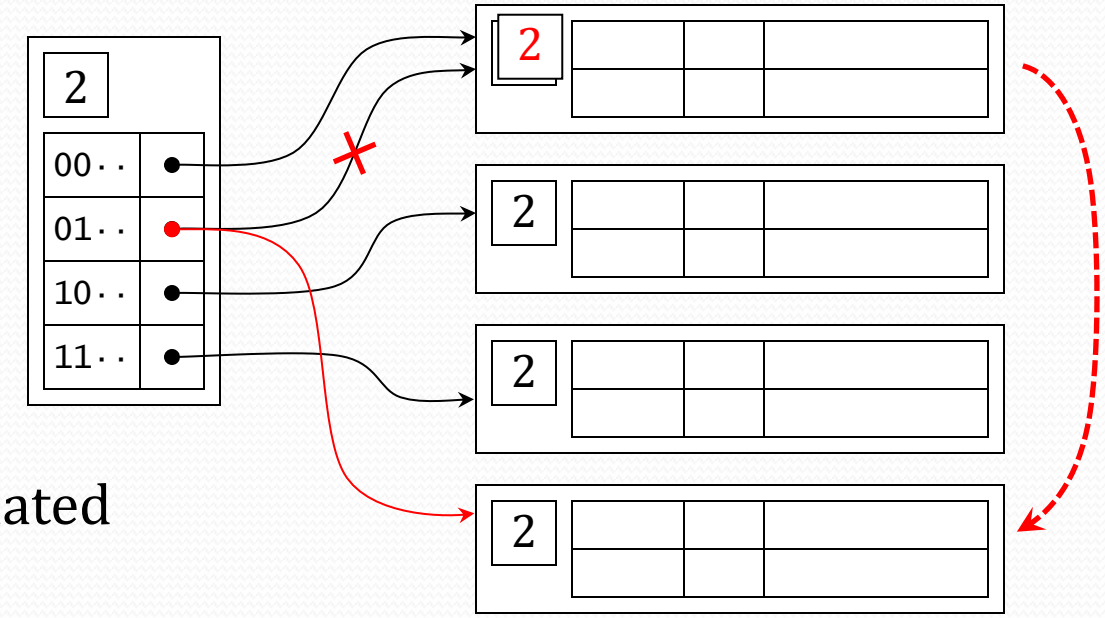
- Plugging actual values into our example:



- When bucket *j* overflows: if $i_j < i$
  - Multiple adjacent table entries refer to the bucket
  - We can split the bucket without increasing the table size

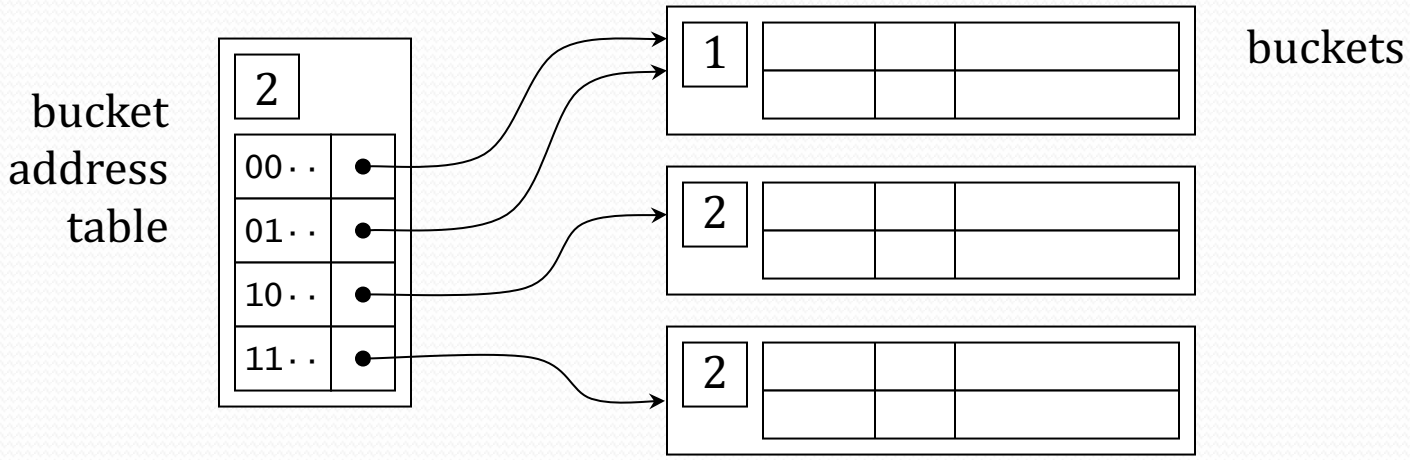
Extendable Hash Buckets (3)

- When bucket j overflows: if $i_j < i$
  - We can split the bucket without increasing size of table
  - New bucket will have a prefix-value $i_j + 1$
  - Also, split bucket's i_j is incremented
- Records in bucket j are rehashed
- Note:
  - Bucket address table may have > 2 entries pointing to bucket j
  - All affected entries must be properly updated



Extendable Bucket Table

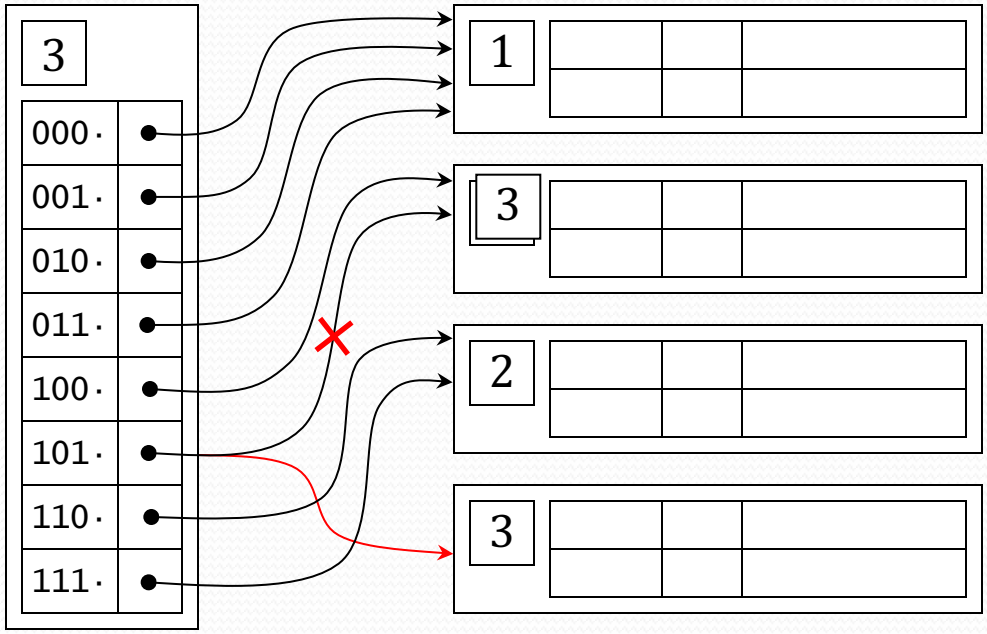
- Our example:



- When bucket *j* overflows: if $i_j = i$
  - Only one table entry refers to the hash bucket ☹
  - Must increase size of address table to split bucket

Extendable Bucket Table

- Increase *i* by 1
  - Each bucket address table entry is expanded into two
  - Now, can split overflowing bucket and rehash contents, as before



Extendable Bucket Table

- Sometimes, splitting a bucket doesn't help
  - All rows in old bucket still end up in one bucket
- Need to be careful to properly diagnose these situations!

- If all entries in a bucket have same search-key value, there's no point in splitting it
  - In these cases, use overflow chaining to handle new additions to the bucket
- If entries in a bucket have different key-values, can attempt to re-split bucket
  - Multiple re-splits may be necessary before bucket size shrinks

Hash Mapping Tables

- Both static hashing and extendable hashing have a common feature:
  - They both must dedicate storage space to an index mapping hash-values to buckets
- Extendable hashing can run into issues as this grows:
  - Bucket address table doubles in size each time
  - As extendable hash index grows, table will occupy more and more blocks, incurring more and more disk IO cost

Linear Hashing

- Linear hashing doesn't require a mapping table
- Instead, it maintains *two* active hash functions at once
- Given a b-bit hash function $h(K)$, as before
- Linear hash table initially starts with N buckets
  - As usual, need to map our hash function $h(K)$ to the bucket address-space [0..N)
  - $h_0(K) = h(K)$ **mod** N
- As hash-table grows, must increase number of buckets
  - Aim to <u>double</u> number of buckets: $h_1(K) = h(K)$ **mod** $2N$
  - *Only expand the linear hash-table one bucket at a time!!!*

Linear Hashing (2)

- The current bucket address-space, and the next bucket address-space, are called *levels*
  - Linear hash table's contents are hashed based on both the current level of splitting, and the next level
- For a given level: $h_{level}(K) = h(K)$ **mod** $(N \times 2^{level})$
  - $h_0(K) = h(K)$ **mod** N Address space is [0..N)
  - $h_1(K) = h(K)$ **mod** $2N$ Address space is [0..2N)
  - $h_2(K) = h(K)$ **mod** $4N$ Address space is [0..4N)
  - $h_3(K) = h(K)$ **mod** $8N$ Address space is [0..8N)
  - …

Linear Hashing (3)

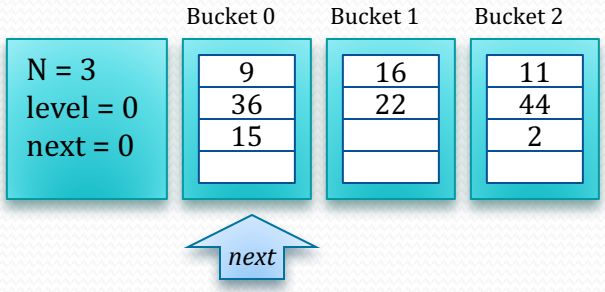
- Linear hash table initially starts with N buckets
- For a given level: $h_{level}(K) = h(K)$ **mod** $(N \times 2^{level})$
- Examine $h_1(K)$ and $h_2(K)$:
  - $h_1(K) = h(K)$ **mod** $2N$
  - $h_2(K) = h(K)$ **mod** $4N$
- If we are given that for a specific value V, $h(V) < 2N$:
  - $h_2(V) = h_1(V)$
- If given that $h(V) \geq 2N$:
  - $h_2(V) = h_1(V) + 2N$

Linear Hashing (4)

- For a given level: $h_{level}(K) = h(K)$ **mod** $(N \times 2^{level})$
  - $h_1(K) = h(K)$ **mod** $2N$
  - $h_2(K) = h(K)$ **mod** $4N$
- For $h_1(K)$ and $h_2(K)$, we expect <u>one</u> of these to be true:
  - $h_2(K) = h_1(K)$
  - $h_2(K) = h_1(K) + 2N$
- If $h(K)$ gives a uniform, random distribution of values, can expect either case to occur with equal likelihood
- Generalize: $h_{i+1}(K) = h_i(K)$, or $h_{i+1}(K) = h_i(K) + N \times 2^i$

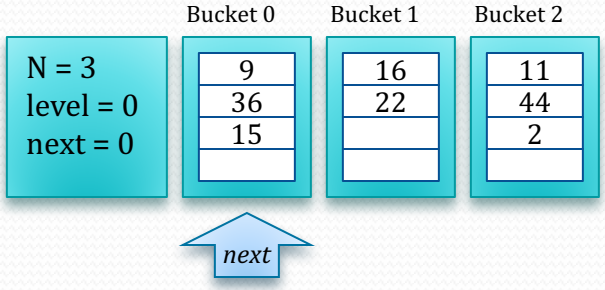
Linear Hash Table Structure

- Our hash table has N buckets initially
  - Initial level is 0. $h_0(K) = h(K)$ **mod** N
  - Example: $N = 3$. K are integers, with $h(K) = K$.
- If a particular bucket overflows, don't just increase N
  - Instead, use overflow chains when a bucket overflows
- Expand buckets in a round-robin fashion
  - A "next" value records which bucket will be split next



Linear Hash Table: Splits

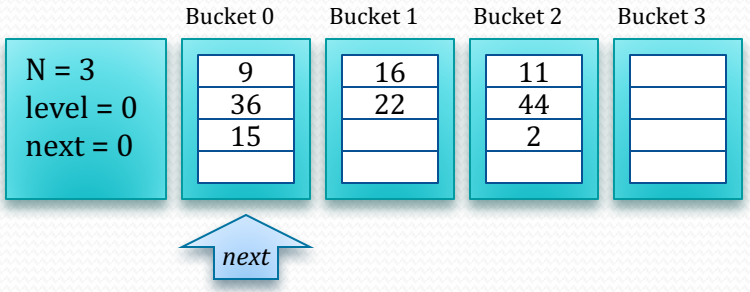
- Can use various criteria to govern when to split the next bucket
- Example: packing factor (a.k.a. load factor)
  - Ratio of records stored to available storage locations
  - Doesn't include overflow pages, so ratio can exceed 1.0
- If packing factor grows beyond a specific limit (e.g. 80%), split the next node in the hash table
  - Given a uniform, random hash function and a reasonable limit, will maintain an upper bound on number of overflow pages.



Linear Hash Table: Splits (2)

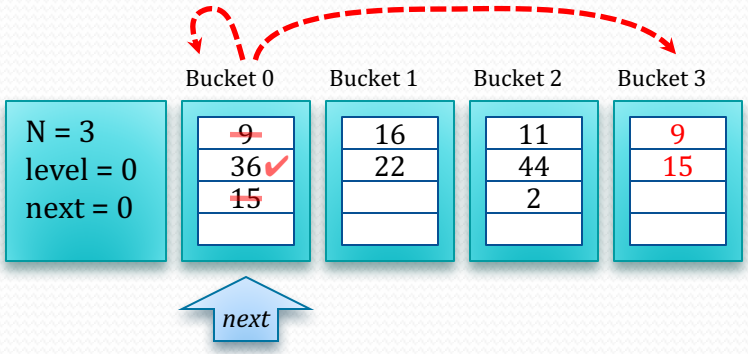
- Example: split bucket 0
  - Add another bucket to the linear hash table (bucket 3), and rehash contents of bucket 0
  - Current level is 0, next level is 1
  - Split bucket 0 using next-level hash function: $h_1(K) = h(K)$ **mod** $2N$

$h_1(K) = h(K)$ **mod** $2N$



Linear Hash Table: Splits (3)

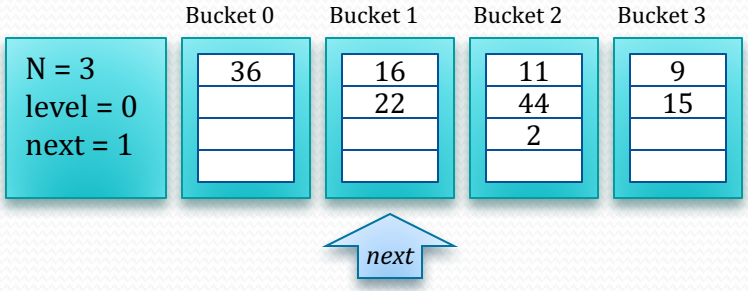
- Example: split bucket 0
- As stated earlier, $h_i(K)$ and $h_{i+1}(K)$ are related
  - Specifically, $h_1(K)$ will either be $h_0(K)$, or $h_0(K) + N$
- Values in bucket 0 will either remain in bucket 0, or they will hash into bucket 3
  - This is what allows us to split buckets one at a time



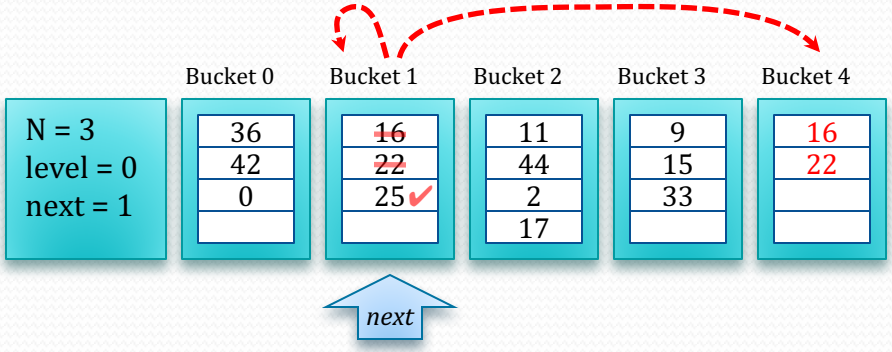
$h_1(K) = h(K)$ **mod** $2N$

Linear Hash Table: Splits (4)

- After splitting, move *next* value forward



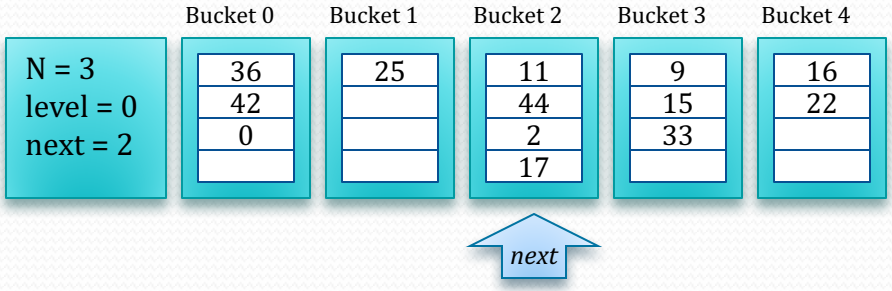
- Next time a bucket needs to be split, split the bucket specified by the *next* value
  - Again, bucket 1 values hash to bucket 1, or to bucket 4



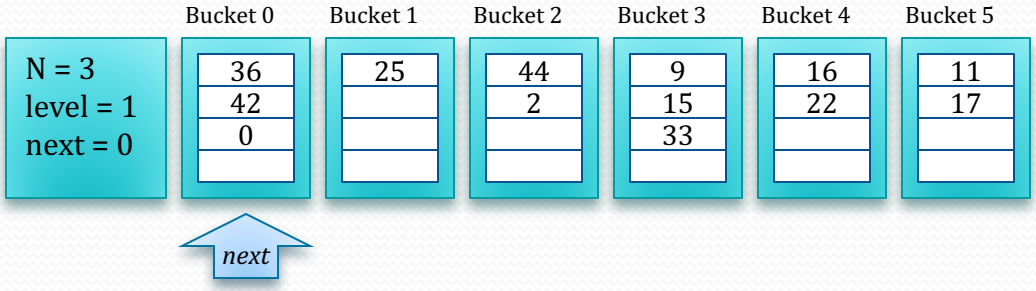
$h_1(K) = h(K) \bmod 2N$

Linear Hash Table: Splits (5)

- Again, move *next* value forward

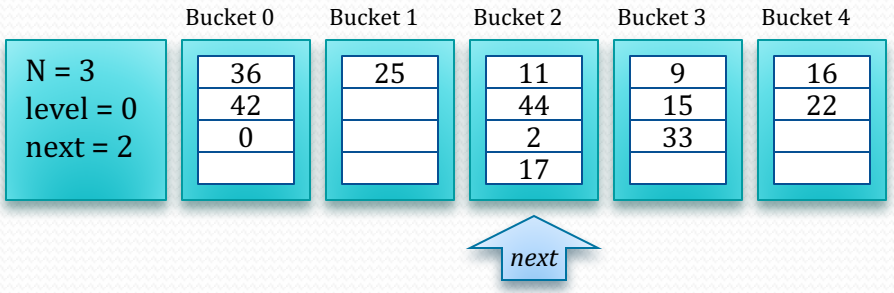


- This process continues until *next* = $N \times 2^{level}$
  - At that point, all buckets at current level have been split!
  - Increment *level*, and reset *next* = 0
- After splitting bucket 2, increment *level*, reset *next*:



Linear Hash Table: Lookups

- At any given time, two hash functions are in effect!
- Example:



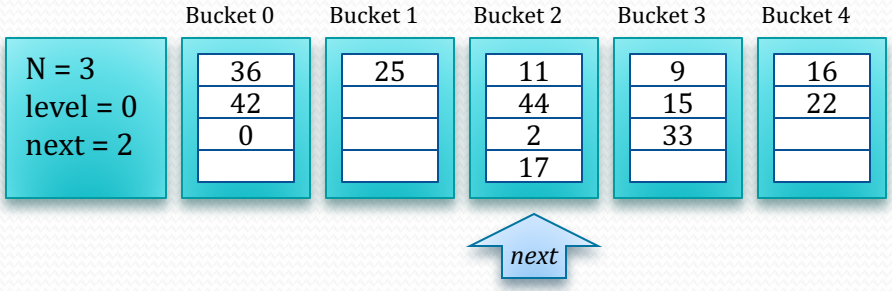
$h_0(K) = h(K) \textbf{ mod } N$

$h_1(K) = h(K) \textbf{ mod } 2N$

- $h_0(K)$ is in effect, and only hashes to buckets 0..2
- $h_1(K)$ is also in effect, and hashes to buckets 0..5
  - *(although bucket 5 doesn't exist yet…)*
- How do we look up the entry for a key-value V ?

Linear Hash Table: Lookups (2)

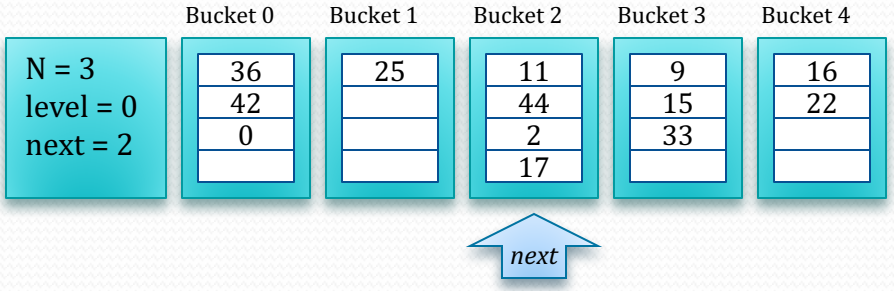
- Example:



- When looking up the entry for a key-value V:
  - Compute $m = h_{level}(V)$
    - Use hash-function for current level
  - If $m < next$:
    - The bucket has already been split!
    - Recompute $m = h_{level+1}(V)$ to get the actual bucket for V
  - Otherwise, if $m \geq next$, bucket hasn't been split yet
    - $h_{level}(V)$ will be the correct hash-function to find bucket for V

Linear Hash Table: Lookups (3)

- Example:



$h_0(K) = h(K)$ **mod** N

$h_1(K) = h(K)$ **mod** $2N$

- Look up entry for $V = 22$
  - Start out with hash-function for the current level
  - $h_0(22) = h(22)$ **mod** $N \times 2^0 = 22$ **mod** $3 = 1$
  - Clearly, 22 isn't in bucket 1!
  - <u>But</u>: $1 < next$, so the bucket has already been split. Need to use $h_1(22)$ instead.
  - Recompute $h_1(22) = h(22)$ **mod** $N \times 2^1 = 22$ **mod** $6 = 4$

Hash vs. Sequential Indexes

- Hash indexes can be extremely effective for speeding up performance of equality-based retrievals
  - Can often find records in 1-2 disk reads
  - Equivalent B^+-tree indexes could take 3-5 or more disk reads to find the record (ignoring caching)
- Unfortunately, generally limited in their usefulness
  - Can't help with range queries, which are very common
  - I/O cost-savings isn't *that* impressive…
- Few commercial databases provide hash indexes
  - Interestingly, both PostgreSQL and MySQL have them…