CS 4440 A

Emerging Database Technologies

Lecture 8
02/03/25

Announcements

Project proposal due tonight

In-class exam next Monday (Feb 10)

- Contents covered: Lec 2 (relational algebra) – Lec 7 (excluding LSM Tree)
- Open book, open note, but no laptops
- Review lecture on Wednesday
- Last year's exam and answer are on canvas
  - Note: Only the first two problems are relevant to this exam

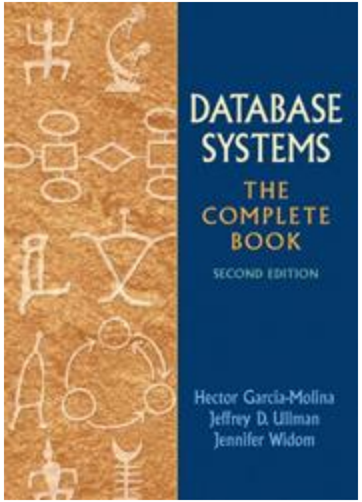
Agenda

1. Static Hash Table
2. Dynamic Hash Table

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 14.3: Hash Tables



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang and CS145 (Intro to Big Data Systems) taught by Peter Bailis.

Indexing vs hashing

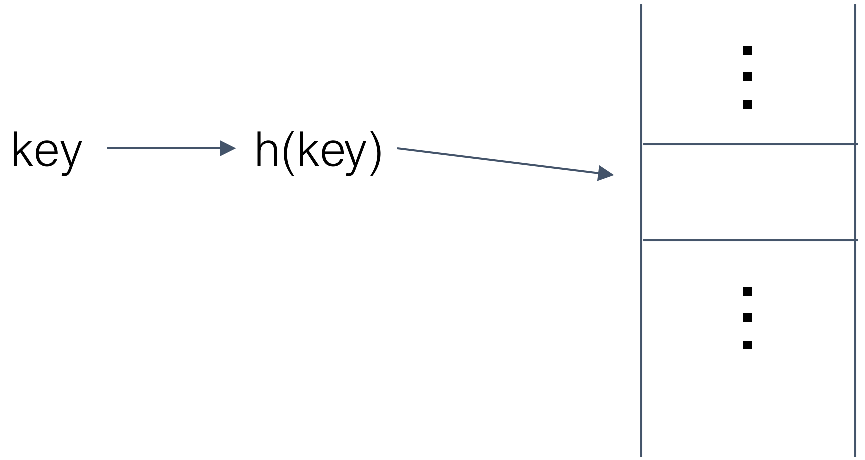
- Indexing (including B+ trees) is good for range lookups
- Hashing is good for equality-based point lookups

```sql
SELECT *
FROM Movies
WHERE year >= 2000;
```

```sql
SELECT *
FROM Movies
WHERE title = ‘Ponyo’;
```

Hash table

- A hash function h takes a key and returns a block number from 0 to B - 1
- Blocks contain records and are stored in secondary storage
- Complexity:
  - O(1) operation complexity
  - O(n) storage complexity



Hash table: Design Decisions

Hash Function

- How to map a large key space into a smaller domain of array offsets
- Trade-off between fast execution vs. collision rate

Hashing Scheme

- How to handle key collisions after hashing
- Trade-off between allocating a large hash table vs. extra steps to location/insert keys

Hash function

For any input key, return an integer representation of that key.

- Output is deterministic

Example:

- Given a key that is a string, return the sum of the characters x_i modulo B (i.e., Σx_i % B)
- This function is not idea since there might be many collisions

We do NOT want to use a cryptographic hash function (e.g., SHA-256) for DBMS hash tables

In general, we only care about the hash function's speed and collision rate.

1. Static Hash Table

Static hash table

- The number of buckets is fixed
- Often used during query execution because they are faster than dynamic hashing schemes.
- If the DBMS runs out of storage space in the hash table, it has to rebuild a larger hash table (usually 2x) from scratch, which is very expensive!

Examples

- Linear Probing Hashing
- Robinhood Hashing (not covered)
- Cuckoo Hashing

Linear Probing Hashing

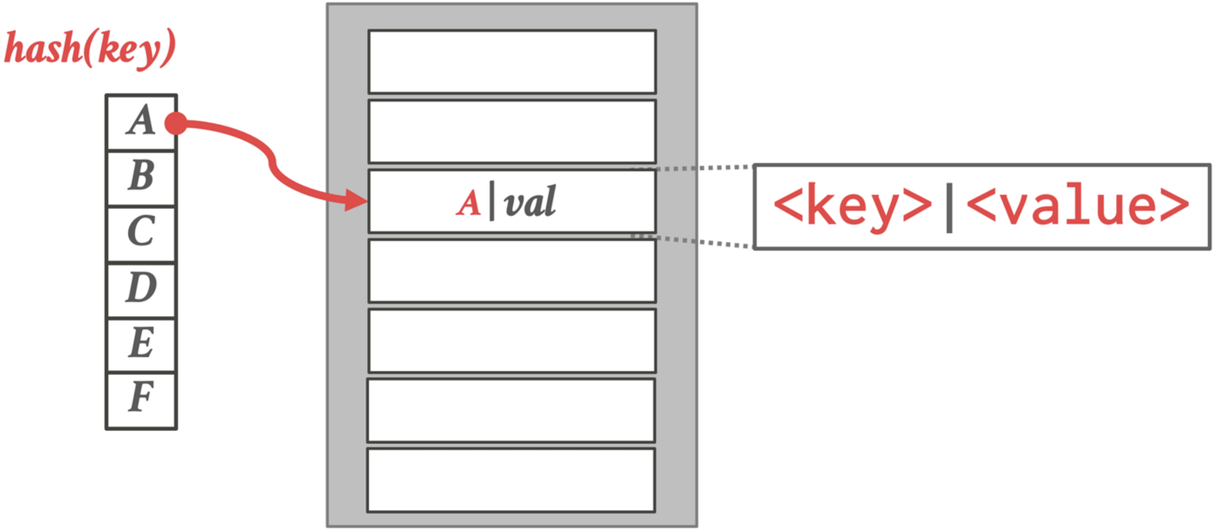
Single giant table of slots

Resolve collisions by linearly searching for the next free slot in the table.

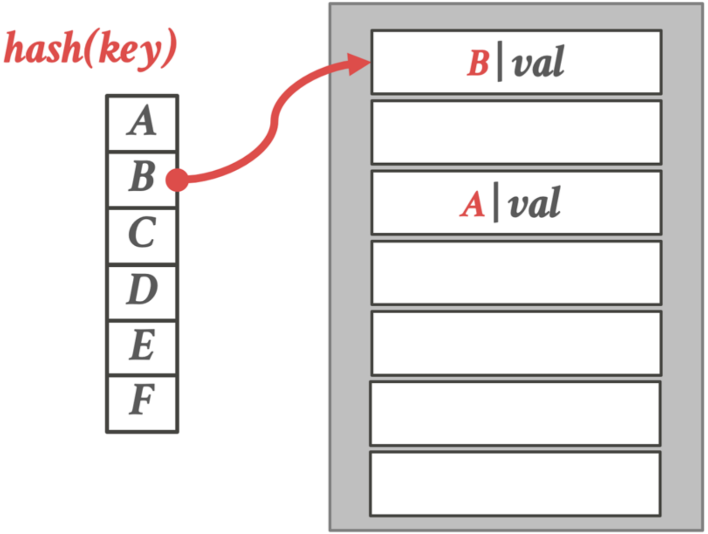
- To determine whether an element is present, hash to a location in the index and scan for it.
- Has to store the key in the index to know when to stop scanning
- Insertions and deletions are generalizations of lookups

Example: Google's absl::flat_hash_map

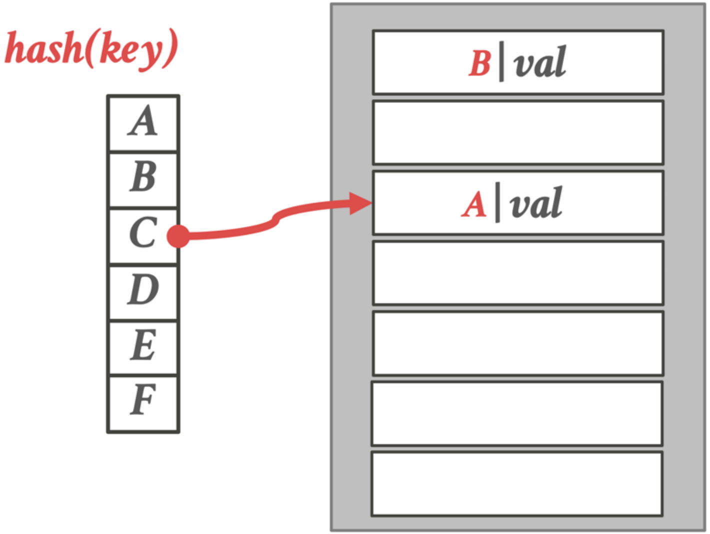
Linear Probing Hashing



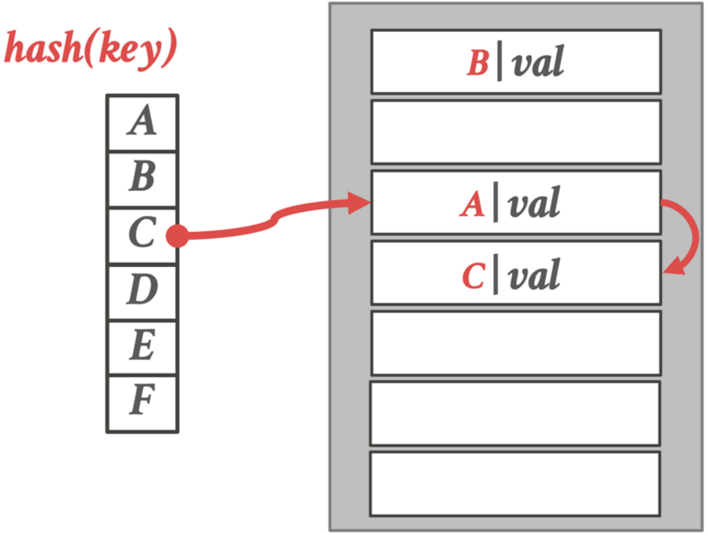
Linear Probing Hashing



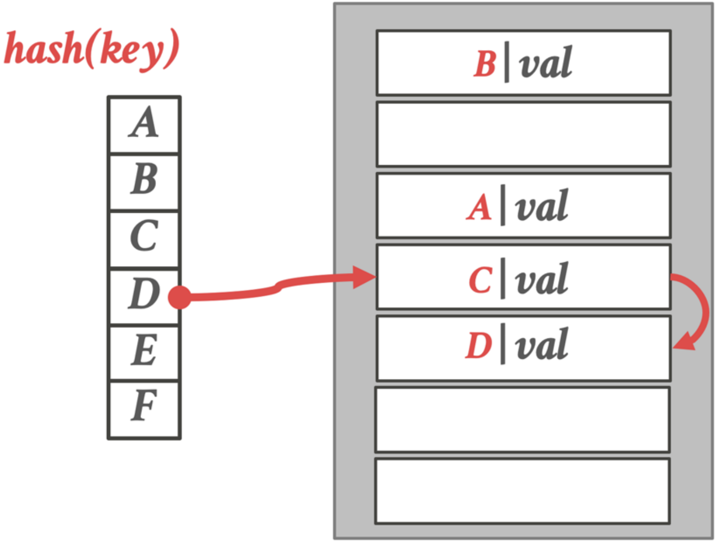
Linear Probing Hashing



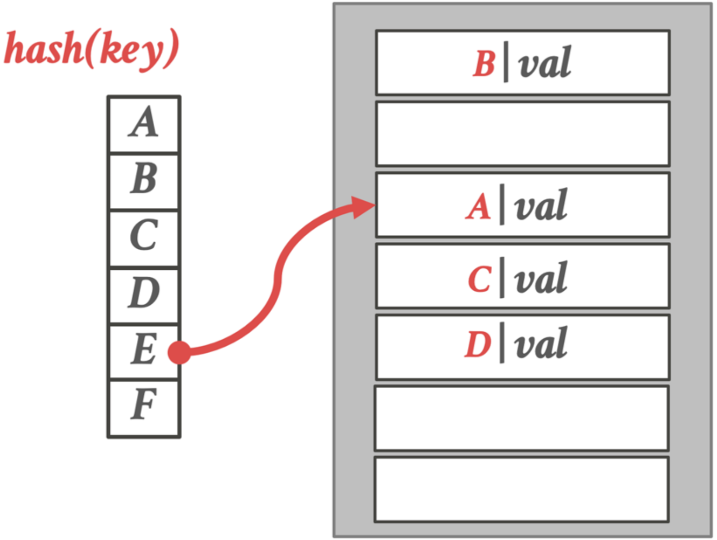
Linear Probing Hashing



Linear Probing Hashing

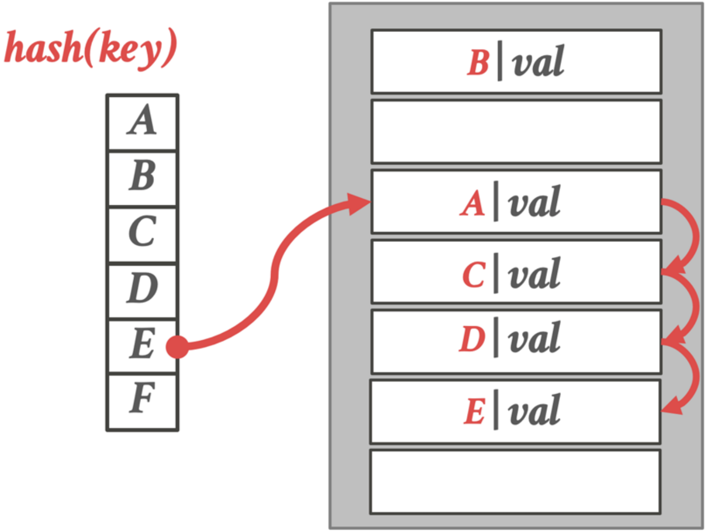


Linear Probing Hashing

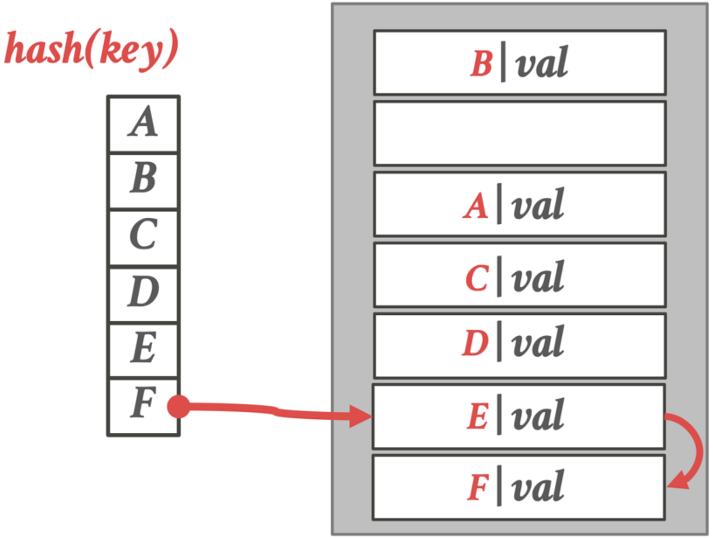


Q: What would happen in this case?

Linear Probing Hashing



Linear Probing Hashing



Linear Probing Hashing - Delete

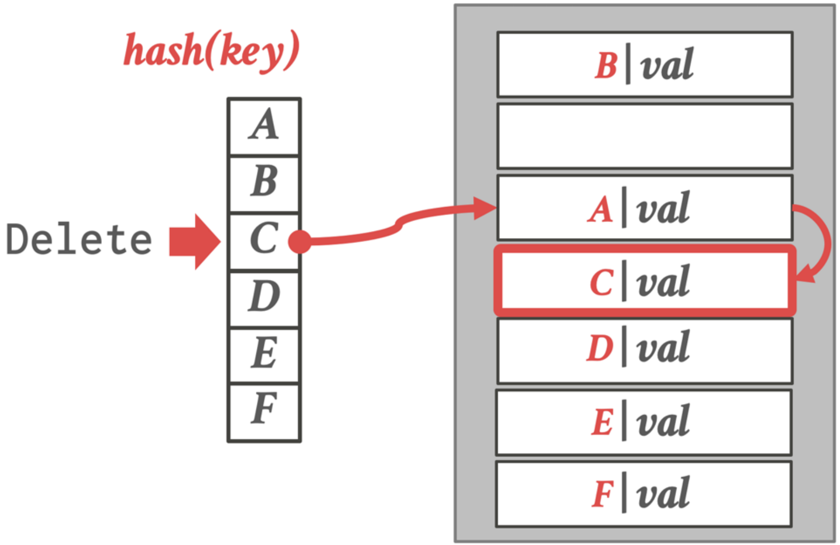
It is not sufficient to simply delete the key

This would affect searches for keys that have a hash value earlier than the emptied cell, but are stored in a position later than the emptied cell.

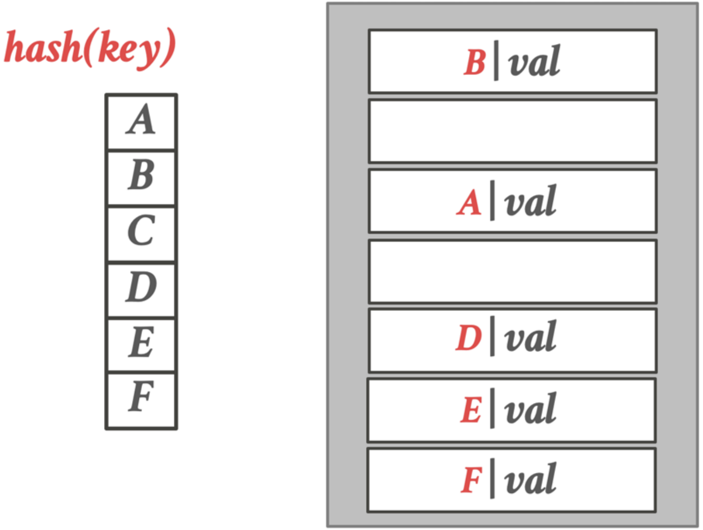
Two solutions:

- Tombstone
- Movement (less common)

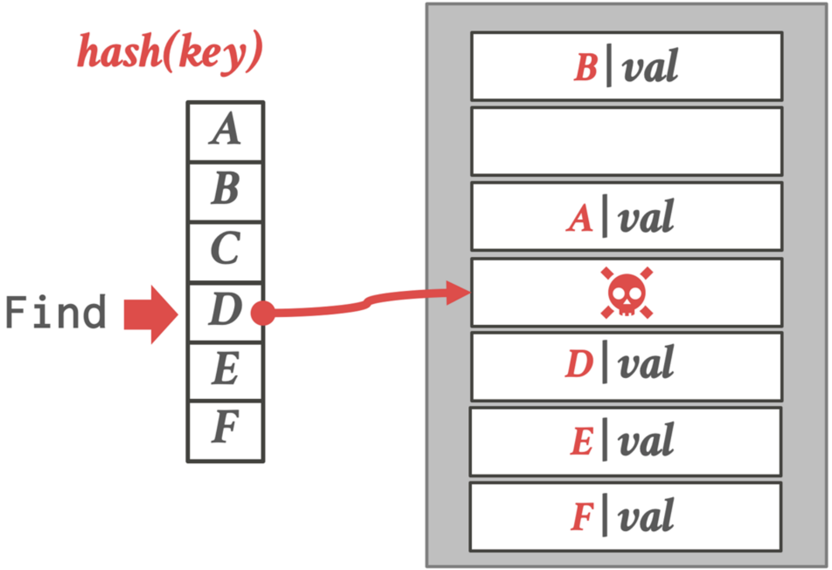
Linear Probing Hashing



Linear Probing Hashing

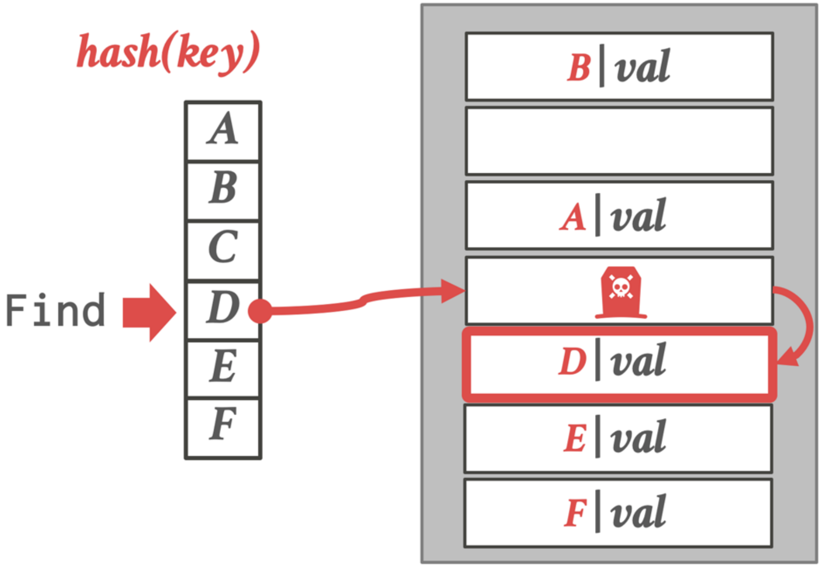


Linear Probing Hashing



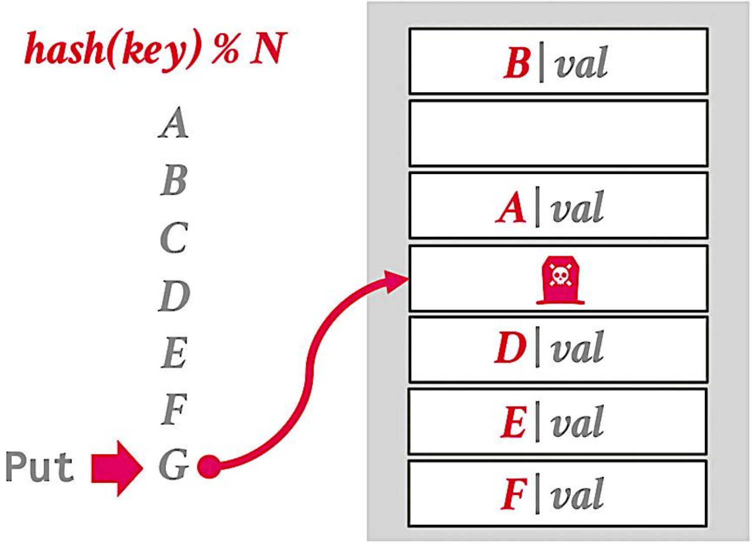
- Set a marker to indicate that the entry in the slot is logically deleted.

Linear Probing Hashing



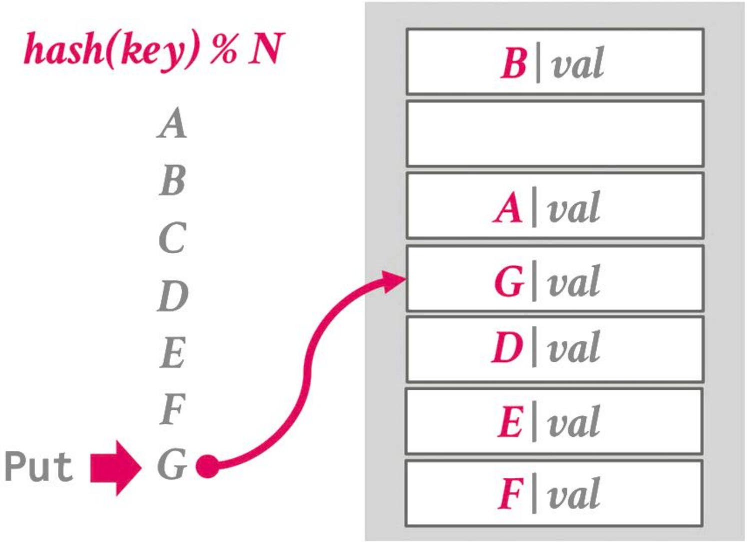
- Set a marker to indicate that the entry in the slot is logically deleted.

Linear Probing Hashing



- Set a marker to indicate that the entry in the slot is logically deleted.
- Reuse the slot for new keys

Linear Probing Hashing



- Set a marker to indicate that the entry in the slot is logically deleted.
- Reuse the slot for new keys

Cuckoo Hashing

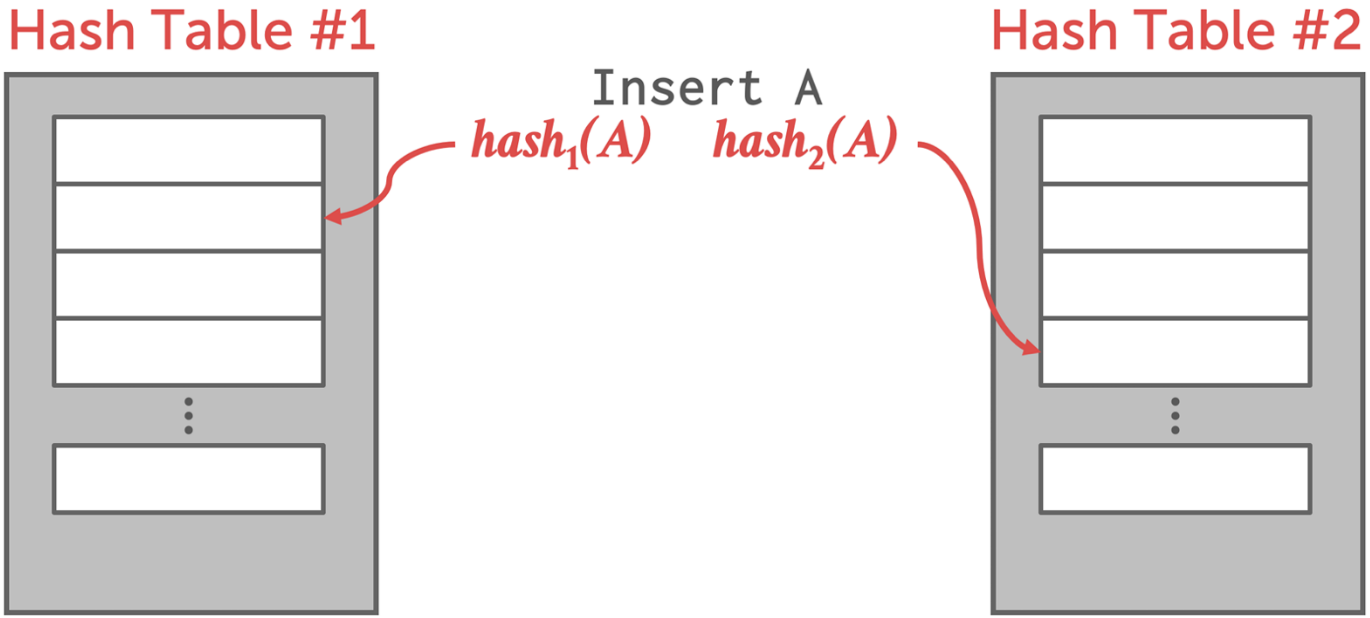
Power of 2 choices: Use multiple hash tables with different seeds

- On insert, check every table and pick one with a free slot
- If no table has a free slot, evict the element from one of then and then re-hash it to find a new location
- In rare cases, we may end up in a cycle. If this happens, we can rebuild using larger hash tables

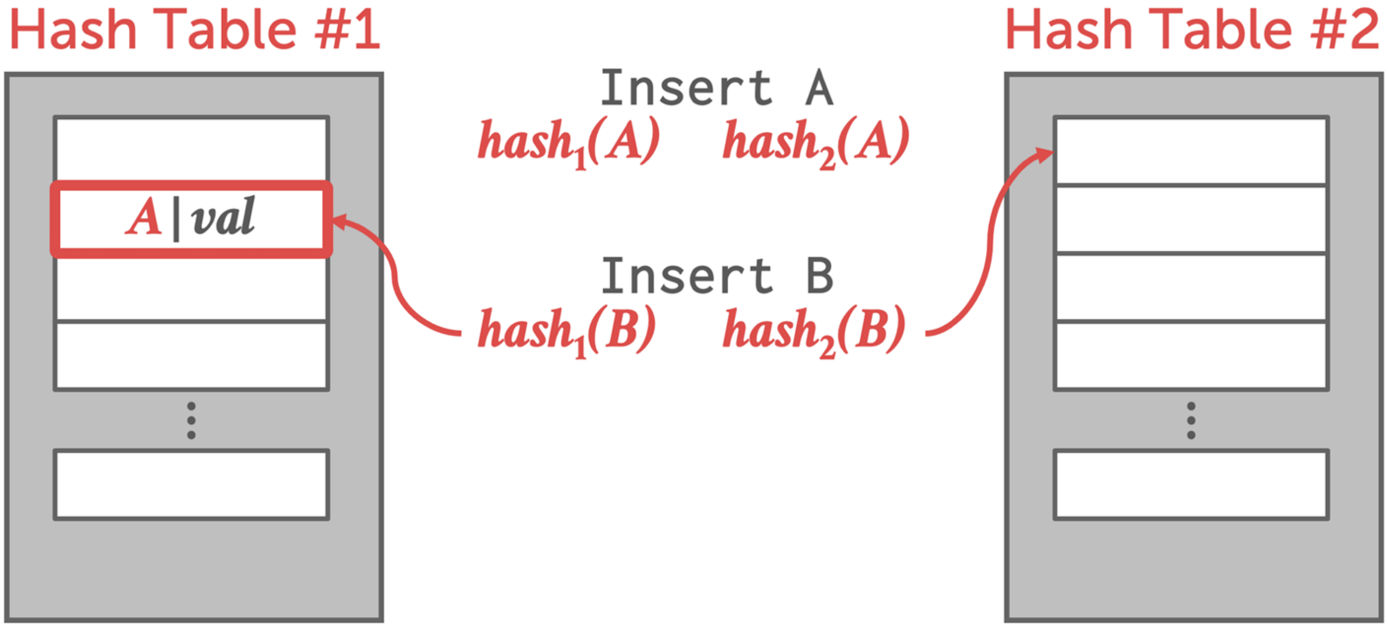
Look-ups and deletions are ~O(1) because only one location per hash table is checked.

Image source: https://theconversation.com/egg-colours-make-cuckoos-masters-of-disguise-34217

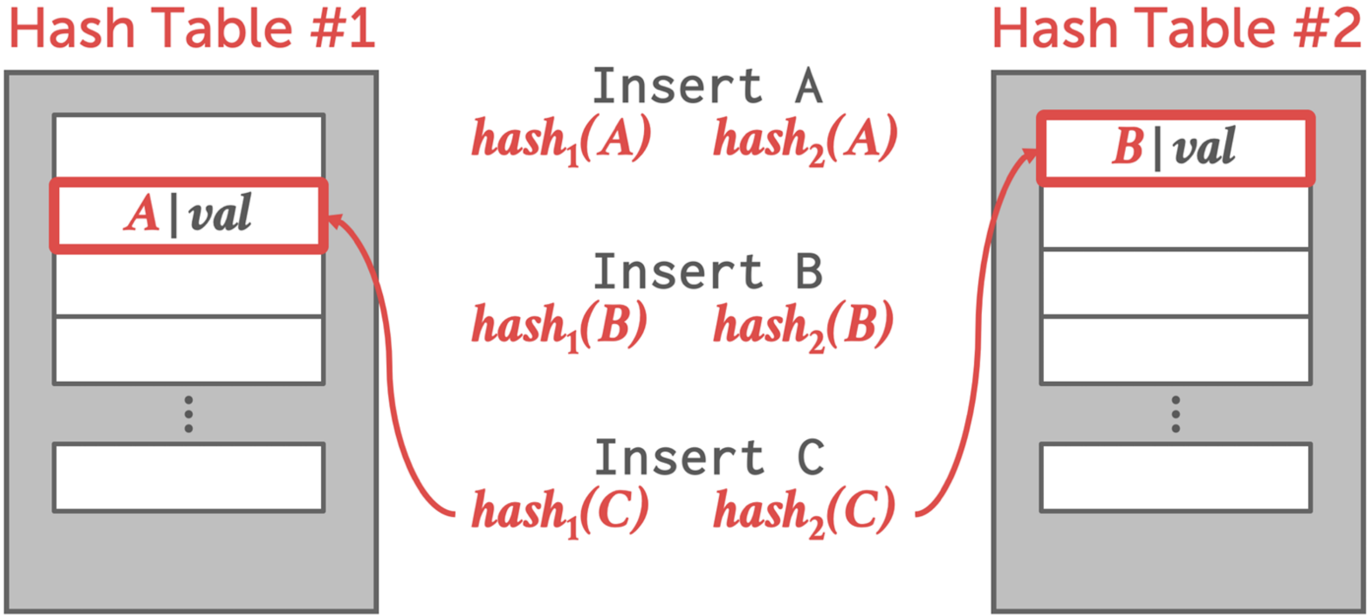
Cuckoo Hashing



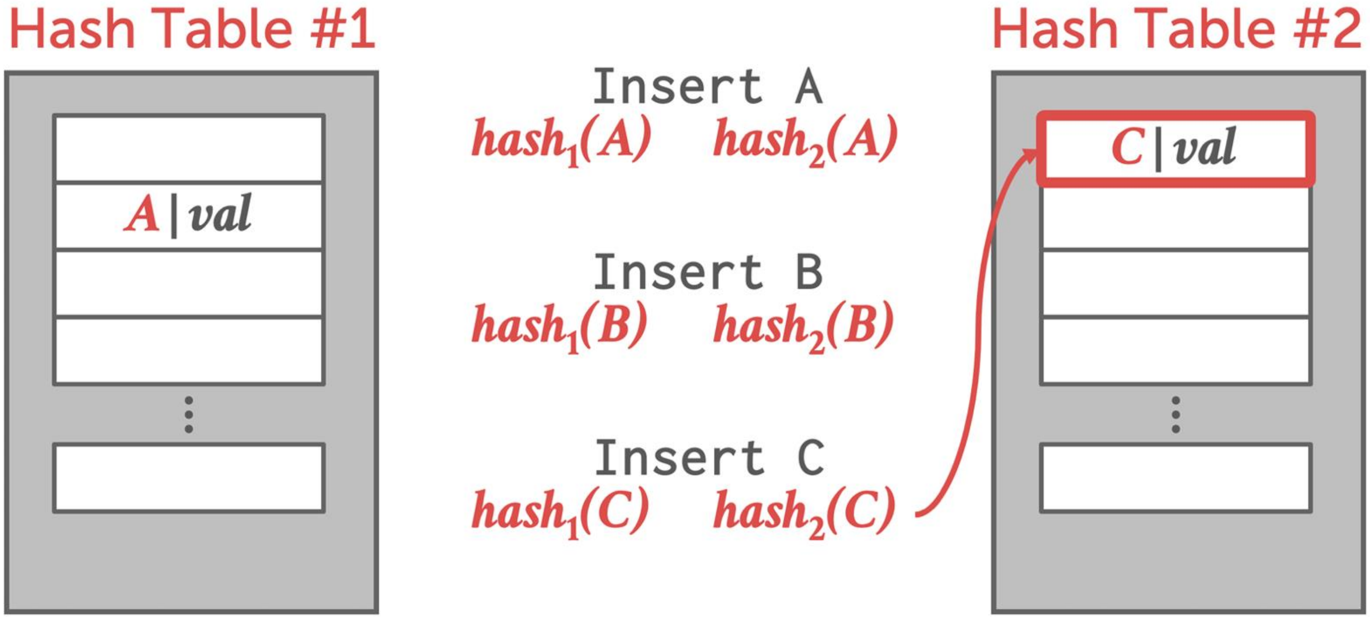
Cuckoo Hashing



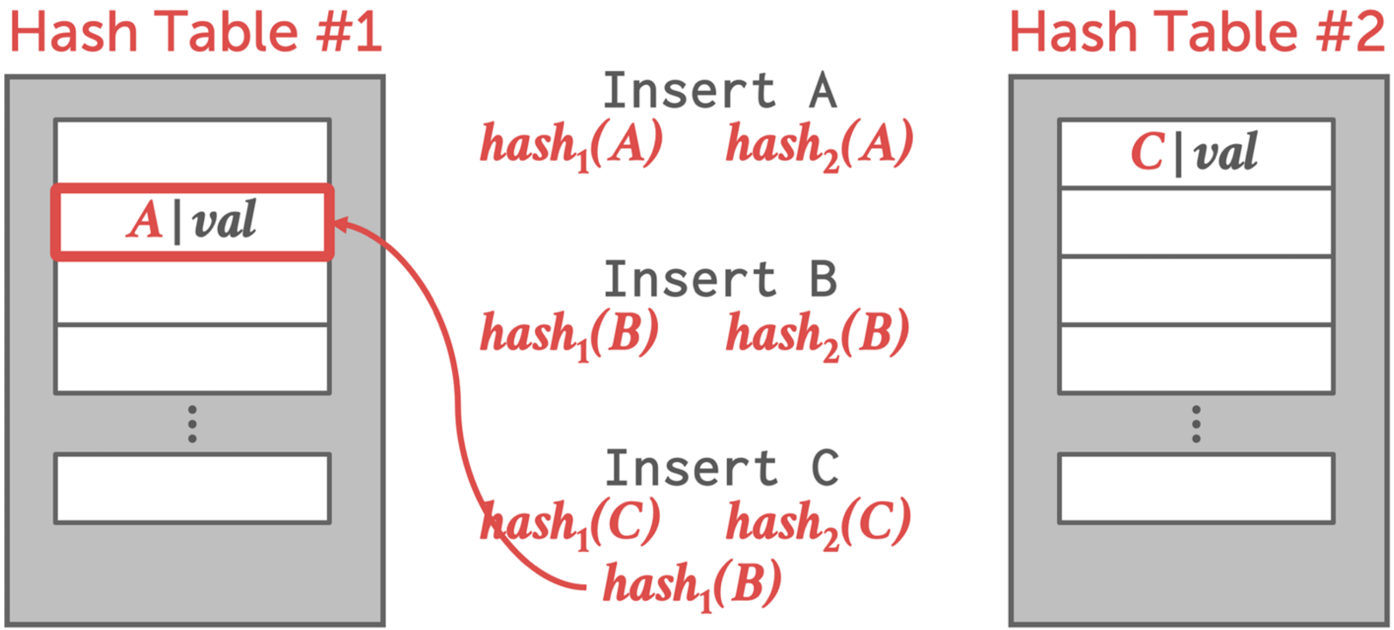
Cuckoo Hashing



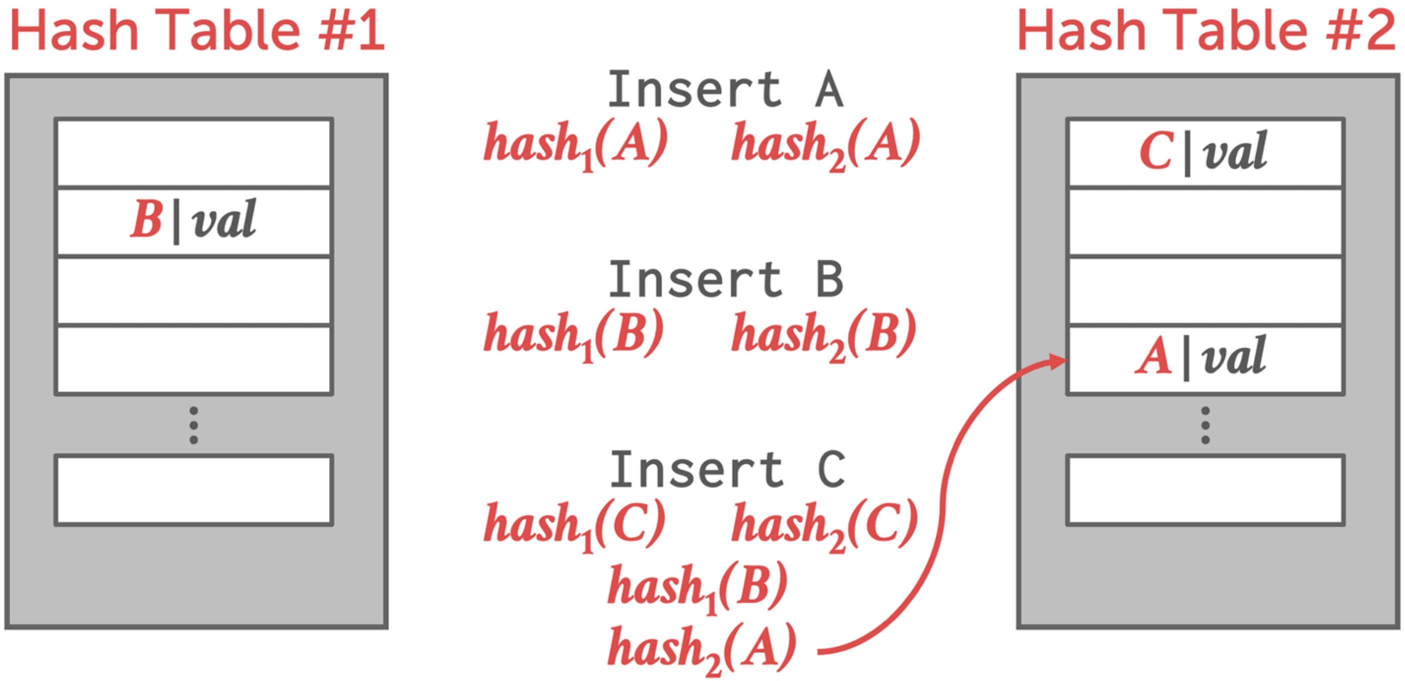
Cuckoo Hashing



Cuckoo Hashing



Cuckoo Hashing



Slides adapted from CMU CS 15-445/645 by Andy Pavlo

2. Dynamic Hash Table

Dynamic hash table

The previous hash tables require the DBMS to know the number of elements it wants to store.

- Otherwise it needs to rebuild the table to resize

Dynamic hash tables incrementally resize the hash table on demand without needing to rebuild the entire table.

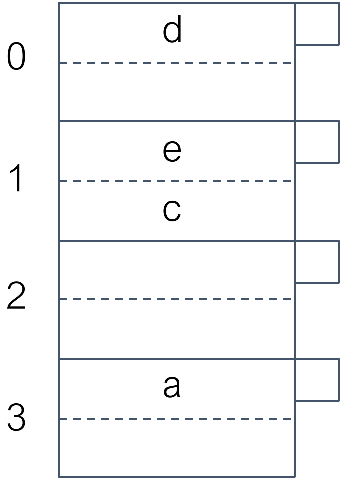
Examples

- Chained Hashing
- Extensible Hashing
- Linear Hashing

Chained Hashing

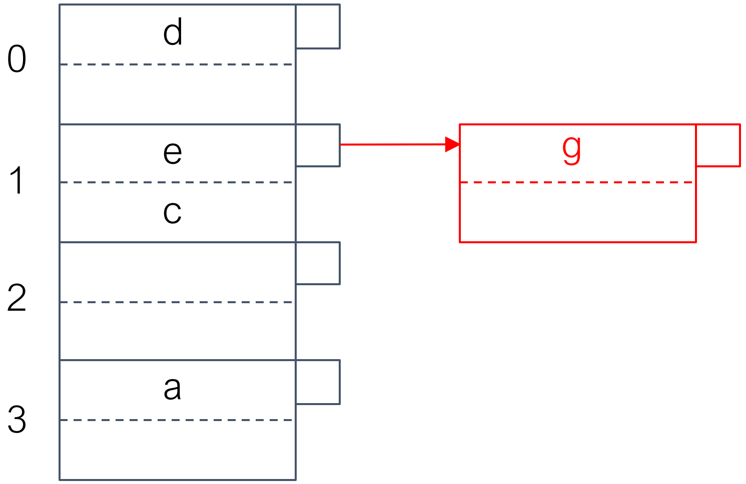
- Maintain a linked list of buckets for each slot in the hash table.

- Resolve collisions by placing all elements with the same hash key into the same bucket.
  - To determine whether an element is present, hash to its bucket and scan for it.
  - Insertions and deletions are generalizations of lookups.



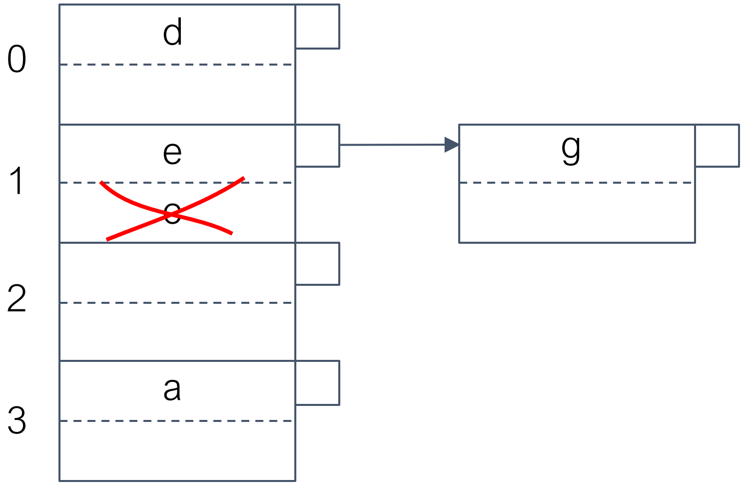
Chained Hashing

- Add g where h(g) = 1



Chained Hashing

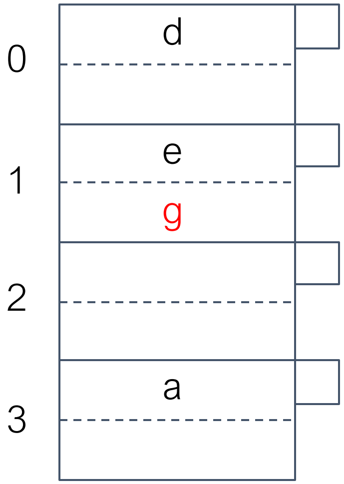
- Remove c where h(c) = 1



Chained Hashing

- Remove c where h(c) = 1

Q: What can go wrong with chained hashing?



Extendible Hashing

Chained-hashing approach that splits buckets incrementally instead of letting the linked list grow forever.

- Long chains of blocks -> many disk I/Os

Multiple slot locations can point to the same bucket chain.

Reshuffle bucket entries on split and increase the number of bits to examine.

- Data movement is localized to just the split chain.

Extensible hash table

Use first i bits of hash value to locate block

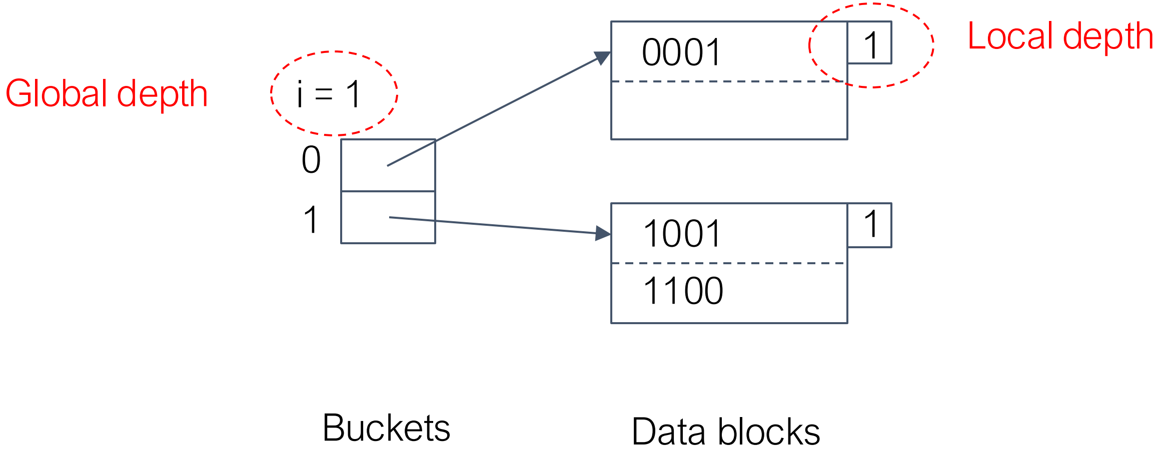
- i grows over time

i = 3

h(key): 00101100

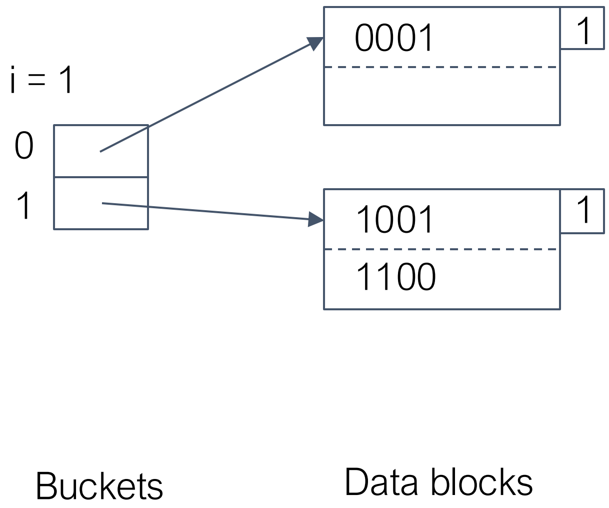
Extensible hash table

Use level of indirection where buckets are pointers to blocks



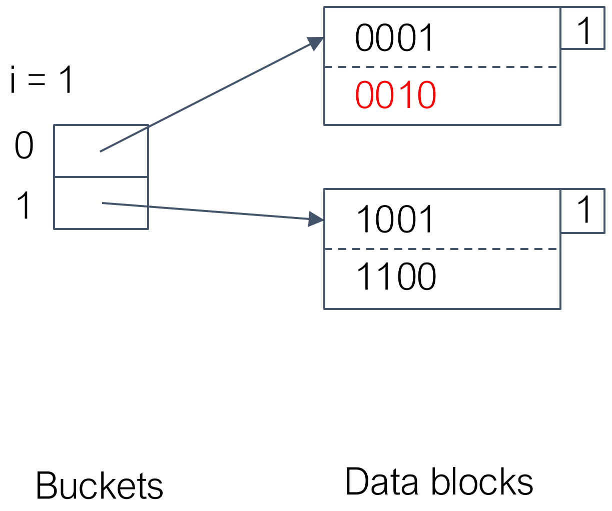
Extensible hash table

- Add 0010



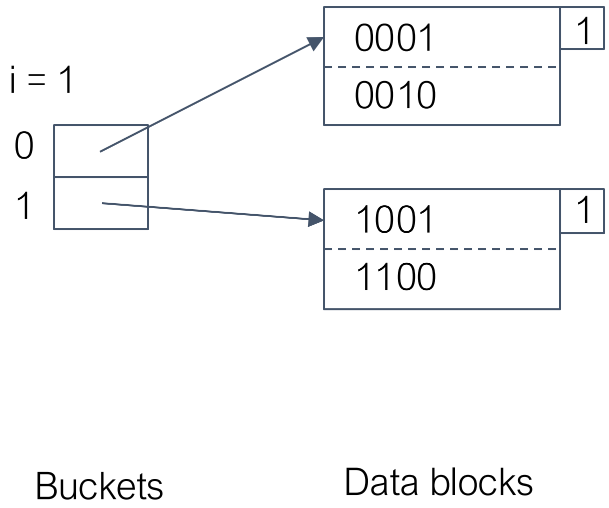
Extensible hash table

- Add 0010



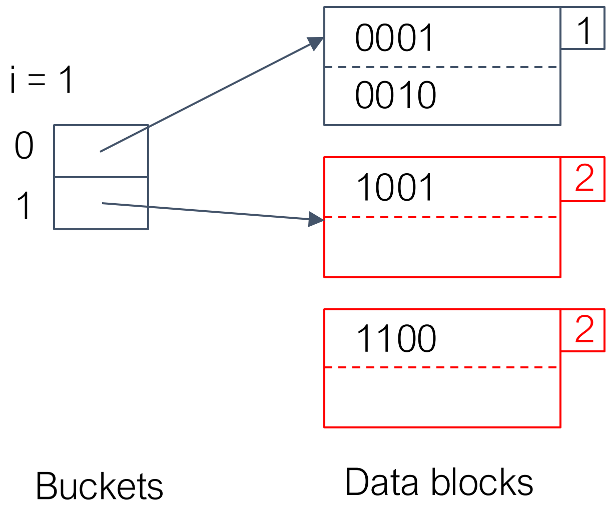
Extensible hash table

- Add 1010



Extensible hash table

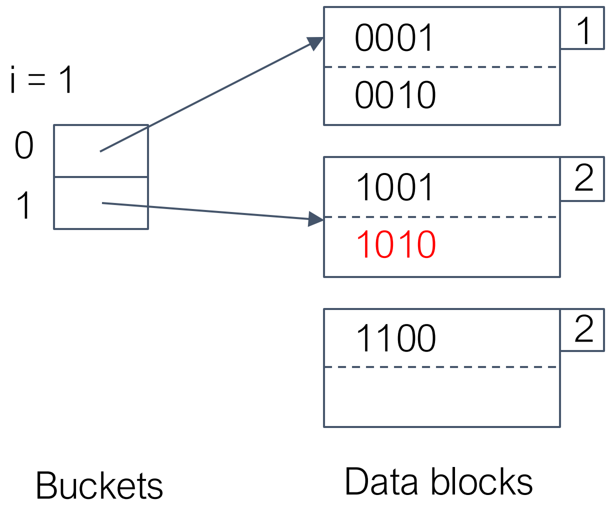
- Add 1010



May need to repeat splitting until there is space

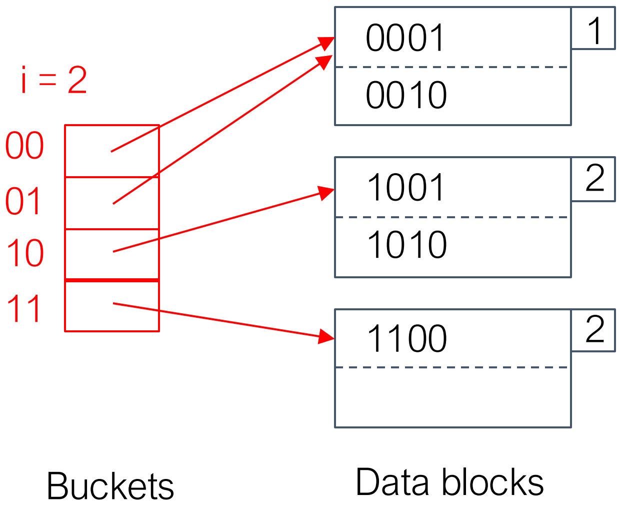
Extensible hash table

- Add 1010



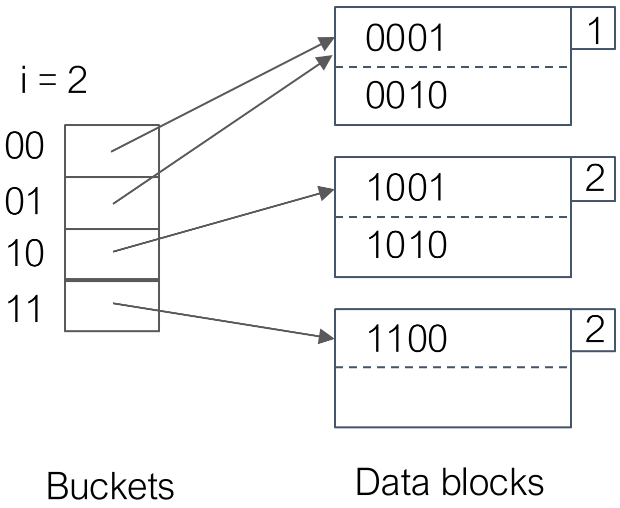
Extensible hash table

- Add 1010



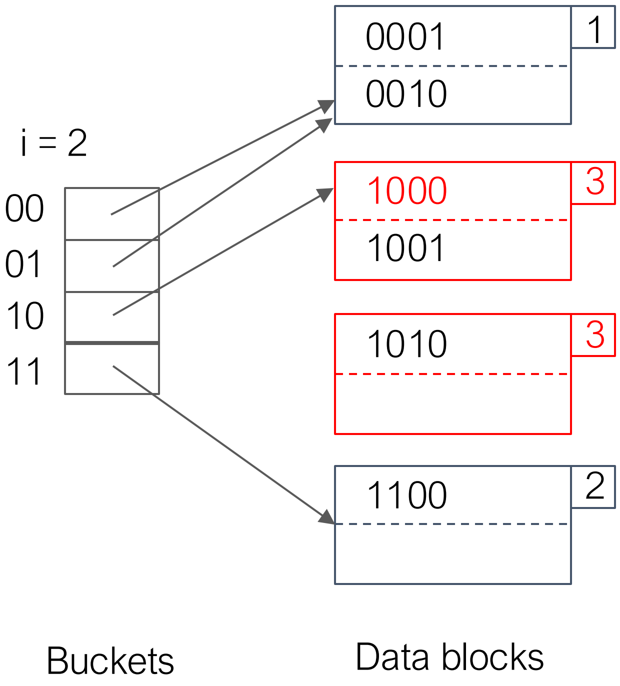
In-class Exercise

- Add 1000
- What happens in this case?



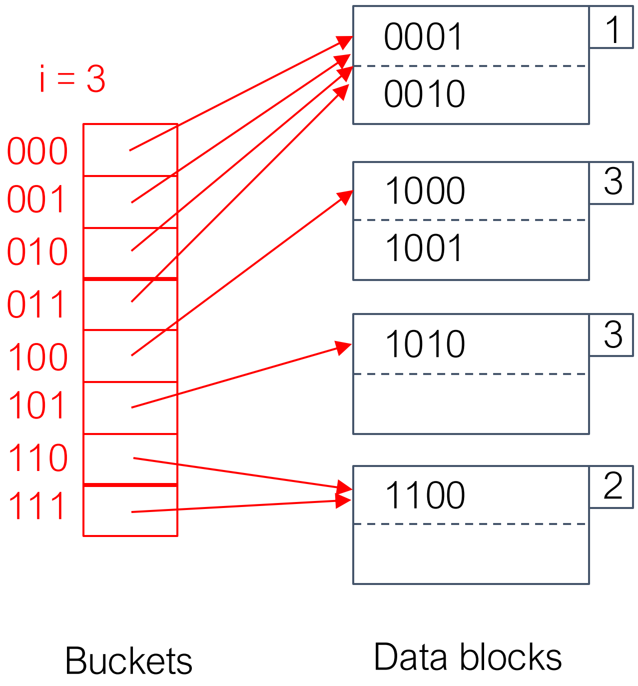
In-class Exercise

- Add 1000



In-class Exercise

- Add 1000



Extensible hashing summary

If bucket array fits in memory, lookup is always 1 disk I/O

Can grow table with little wasted space and avoiding full reorganizations

However, doubling the bucket array is expensive

- Splitting can occur frequently if the number of records per block is small
- At some point, the bucket array may not fit in memory

Linear hashing (covered next) grows the number of buckets more slowly

Linear hashing

The hash table maintains a pointer that tracks the next bucket to split.

- When any bucket overflows, split the bucket at the pointer location.

Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

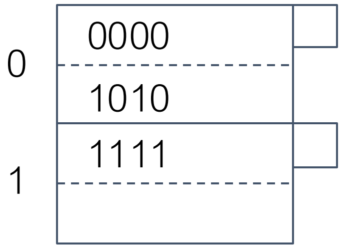
- Space Utilization
- Average Length of Overflow Chains

Linear hash tables

- Use last i bits of hash value to locate block
- Hash table grows linearly

# bits used	i = 1
# buckets	n = 2
# records	r = 3

Policy: limit $r \leq 1.7n$



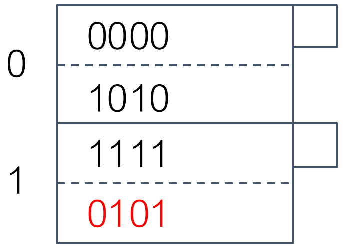
Linear hash tables

- Add 0101

# bits used	i = 1
# buckets	n = 2
# records	r = 4

Policy: limit r ≤ 1.7n

Violation

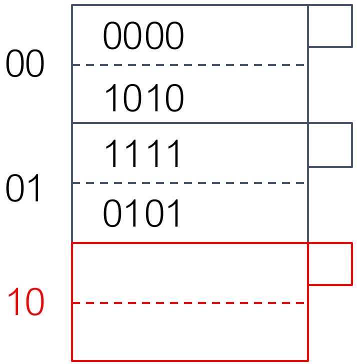


Linear hash tables

- Add 0101

# bits used	i = 2
# buckets	n = 3
# records	r = 4

Policy: limit r ≤ 1.7n

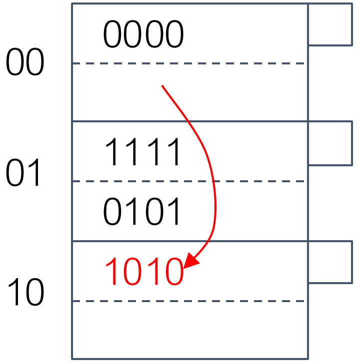


Linear hash tables

- Add 0101

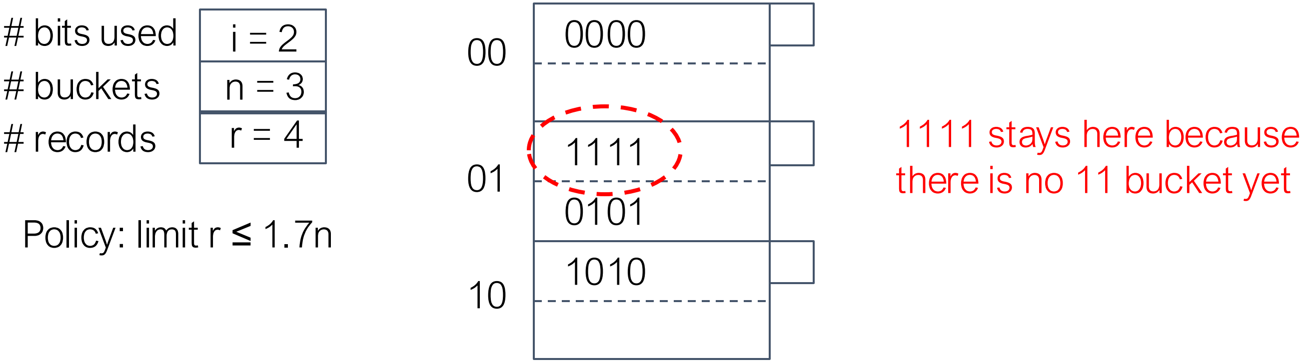
# bits used	i = 2
# buckets	n = 3
# records	r = 4

Policy: limit r ≤ 1.7n



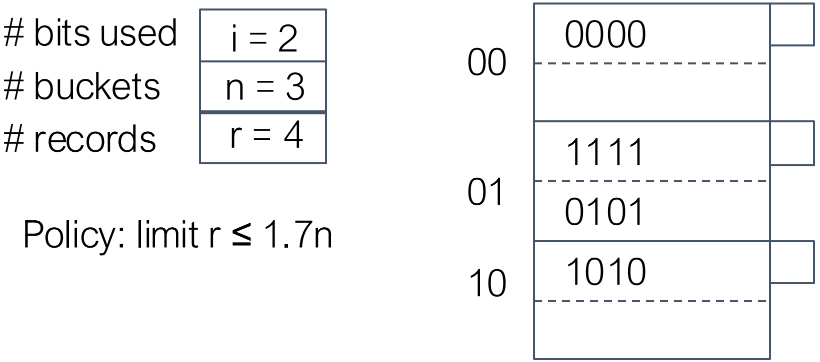
Linear hash tables

- Add 0101



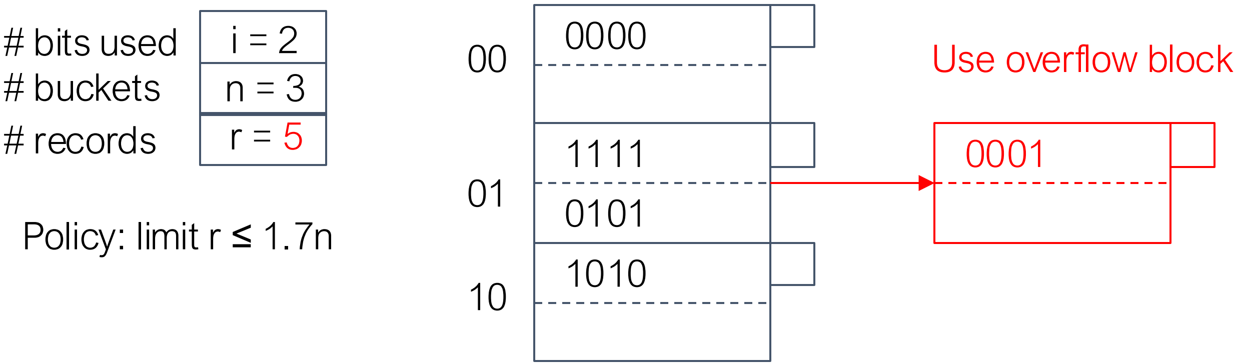
Linear hash tables

- Add 0001



Linear hash tables

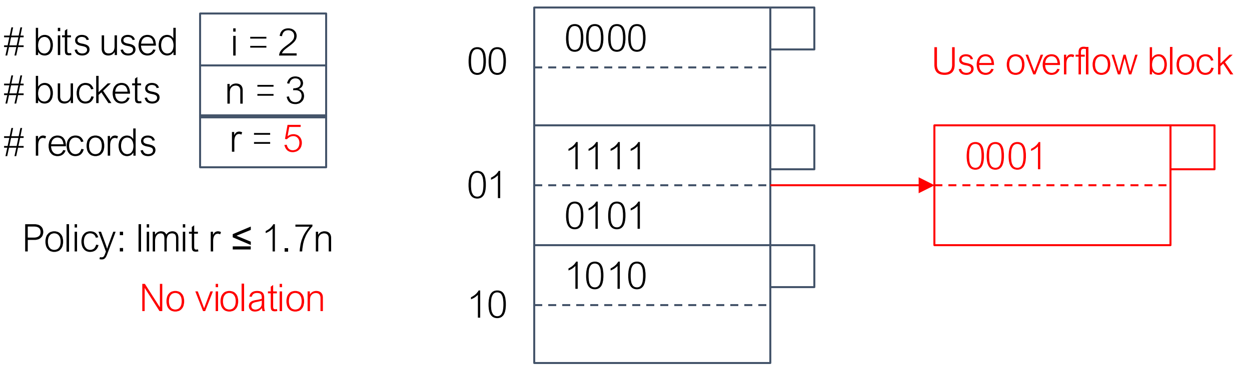
- Add 0001



Policy: limit r ≤ 1.7n

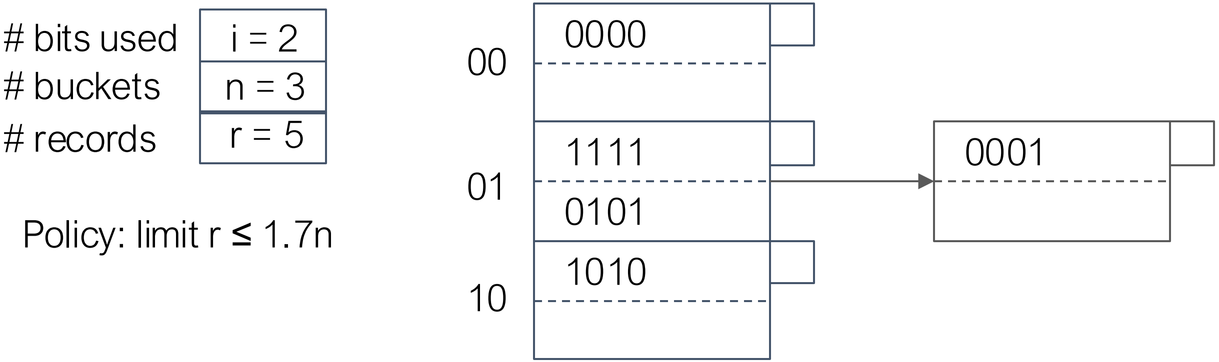
Linear hash tables

- Add 0001



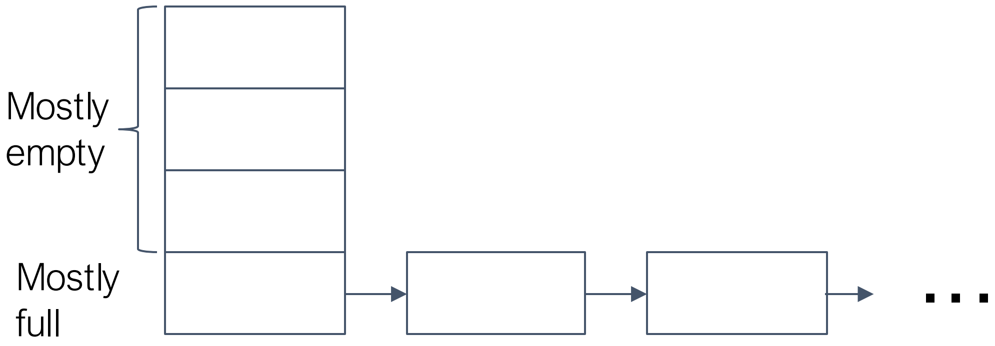
In-class Exercise

- Continuing with example, add 0111.
  What happens here?



Linear hashing summary

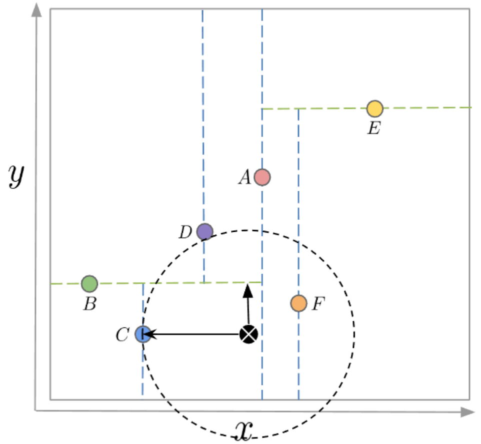
- Can grow table with little wasted space and avoiding full reorganizations
- Compared to extensible hashing, there is no array of buckets
- However, there can be a long chain of overflow blocks



Multidimensional Indexes (14.4)

All the index structures discussed so far are one dimensional

- Assume a single search key, and they retrieve records that match a given search key value.
- The key can contain multiple attributes



Examples:

- KD-tree, R-tree