

Indexing

CS348

Instructor: Sujaya Maiyya

Outline

- Types of indexes
- Index structure
- How to use index

What are indexes helpful for?

- Given a value, locate the record(s) with this value

SELECT * FROM R WHERE *A = value*;

SELECT * FROM R, S WHERE *R.A = S.B*;

- Find data by other search criteria, e.g.

- Range search

SELECT * FROM R WHERE *A > value*;

- We call A in the above example a *search key*
 - The attribute whose values will be indexed

Functionality of Indices

- Indices are the primary mechanism to:
 1. *retrieve records quickly*
 2. *search records in sort order*

SELECT * FROM User WHERE ID = 912;

SELECT * FROM User WHERE ID > 100;

- Without index: sequential scans
 - Read each page and each record
 - Can be very slow for large tables
 - At a high-level: if a file is sorted on some columns:
 - Can now do binary search
 - Key Question: How to sort a file efficiently?

page 1	ID	name	age
	123	Milhouse	10
	142	Bart	10
	279	Jessica	10
page 2	345	Martin	8
	456	Ralph	8
	512	Nelson	10
	679	Sherri	10
page 3	697	Terri	10
	857	Lisa	8
	912	Windel	8
	997	Jessica	8
...			
page k	...	...	...
	...	...	...
	...	...	...

users.db

Naïve Approach for Keeping a Table Sorted

- Sorting: primary technique to find things & data quickly!
- How to keep a relation file in sorted order (e.g., users.db)?
- Assume a sequence of insertions. 2 records/page. Sort on ID
- Simple/naïve approach: Shift-based (index-less) sorting

Naïve Approach for Keeping a Table Sorted

Next Insertion

857	Lisa	8
-----	------	---

PAGES

pg 1

Naïve Approach for Keeping a Table Sorted

Next Insertion

512	Nelson	10
-----	--------	----

PAGES

857	Lisa	8

pg 1

Naïve Approach for Keeping a Table Sorted

Next Insertion

279	Jessica	10
-----	---------	----

PAGES

512	Nelson	10
857	Lisa	8

pg 1

Naïve Approach for Keeping a Table Sorted

Next Insertion

912	Windel	8
-----	--------	---

PAGES

279	Jessica	10
512	Nelson	10

pg 1

857	Lisa	8

pg 2

Naïve Approach for Keeping a Table Sorted

Next Insertion

345	Martin	8
-----	--------	---

PAGES

279	Jessica	10
512	Nelson	10

pg 1

857	Lisa	8
912	Windel	8

pg 2

Naïve Approach for Keeping a Table Sorted

Next Insertion

697	Terri	10
-----	-------	----

PAGES

279	Jessica	10
345	Martin	8

pg 1

512	Nelson	10
857	Lisa	8

pg 2

912	Windel	8

pg 3

Naïve Approach for Keeping a Table Sorted

Next Insertion

123	Milhouse	10
-----	----------	----

PAGES

279	Jessica	10
345	Martin	8

pg 1

512	Nelson	10
697	Terri	10

pg 2

857	Lisa	8
912	Windel	8

pg 3

Naïve Approach for Keeping a Table Sorted

Next Insertion

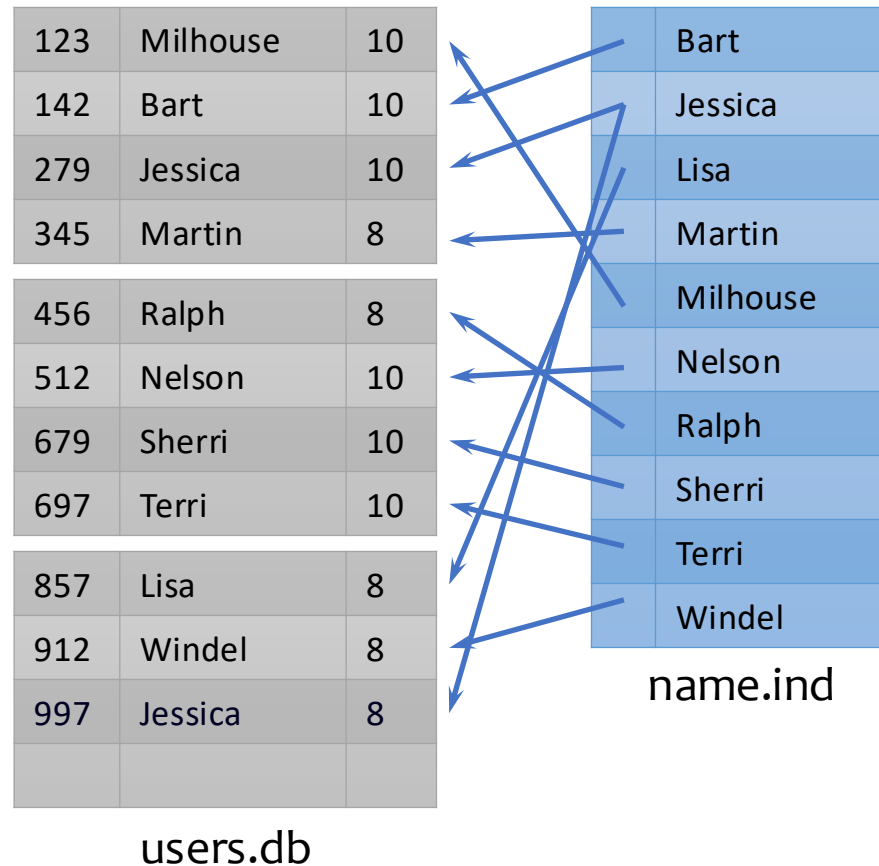
PAGES

123	Milhouse	10	pg 1
279	Jessica	10	
345	Martin	8	pg 2
512	Nelson	10	
697	Terri	10	pg 3
857	Lisa	8	
912	Windel	8	pg 4

- **Pro:** Very simple to implement
- **Con:** Each insertion could require up to $2 \times b$ many I/Os (to read and correct pages) if the table has b pages.
- **Con:** Can only sort of one attribute.

Indices to rescue!!

- Indices: are persistent data structures that are stored along with table files that allow fast search.
- Indices are often much smaller than the original table records.



Dense v.s. sparse indexes

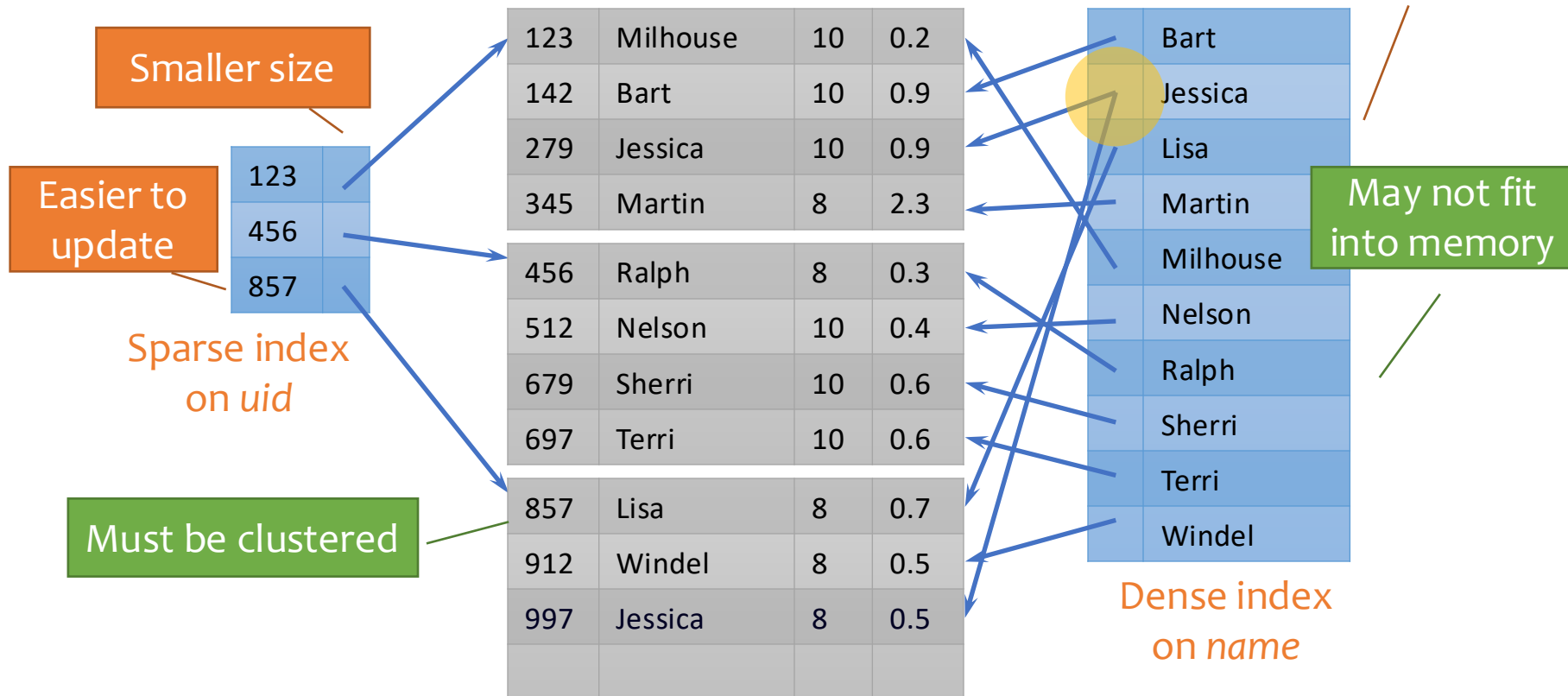
- **Dense**: one index entry for each search key value
 - One entry may “point” to multiple records (e.g., two users named Jessica)
- **Sparse**: one index entry for each page
 - Records must be **clustered** according to the search key on disk



Dense v.s. sparse indexes

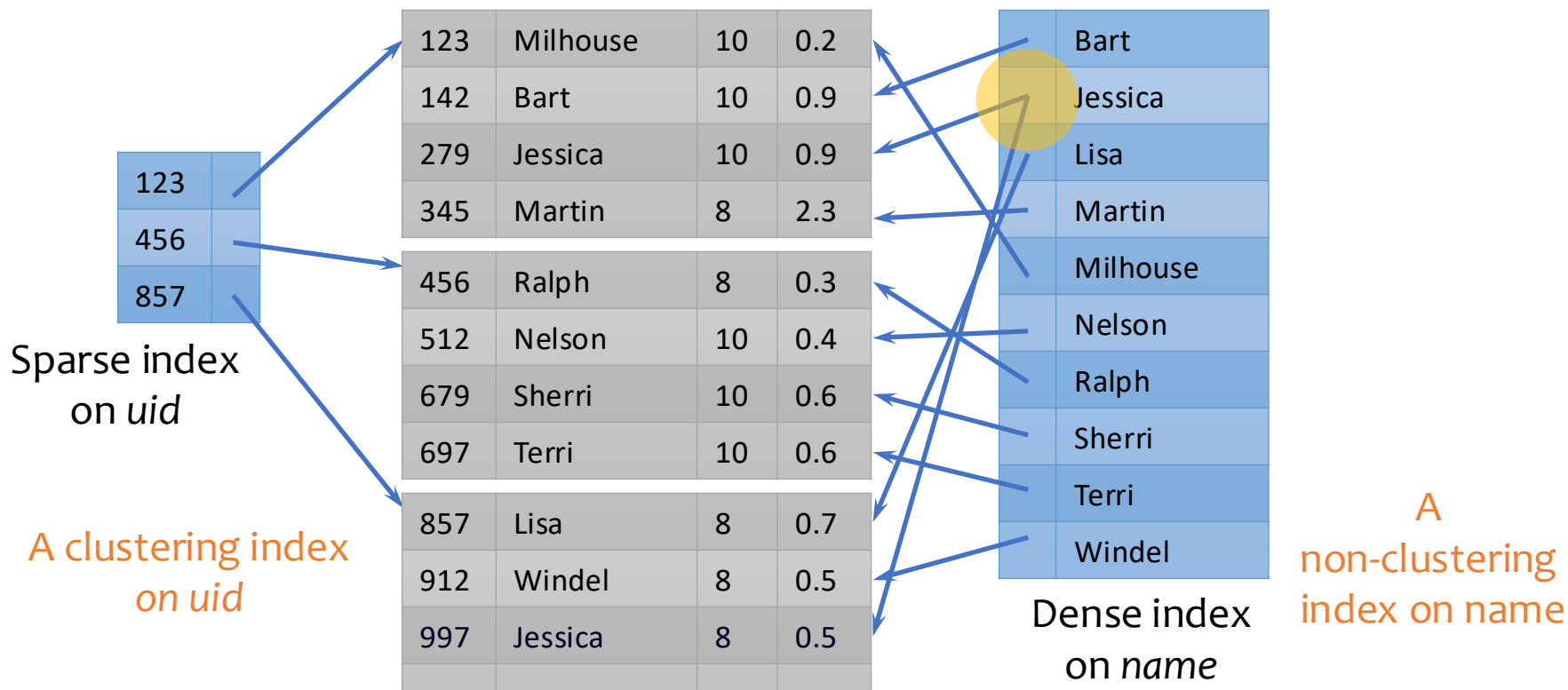
- **Dense**: one index entry for each search key value
 - One entry may “point” to multiple records (e.g., two users named Jessica)
- **Sparse**: one index entry for each page
 - Records must be **clustered** according to the search key on disk

Can tell directly if a record exists



Clustering v.s. non-clustering indexes

- An index on attribute A is a **clustering** index if tuples in the relation with similar values for A are stored together in the same block.
- Other indices are **non-clustering (or secondary)** indices.
- Note: A relation may have **at most one clustering index**, and any number of non-clustering indices.



Primary and secondary indexes

- **Primary index**

- Typically created for the **primary key** of a table
- Records are usually clustered by the primary key
- Clustering index → sparse

- **Secondary index**

- Non-clustering index, usually dense (to find each search key value, since records are not clustered by this search key)

- **SQL**

- PRIMARY KEY declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index
- Additional secondary index can be created on non-key attribute(s):

CREATE INDEX UserPopIndex **ON** User(pop);

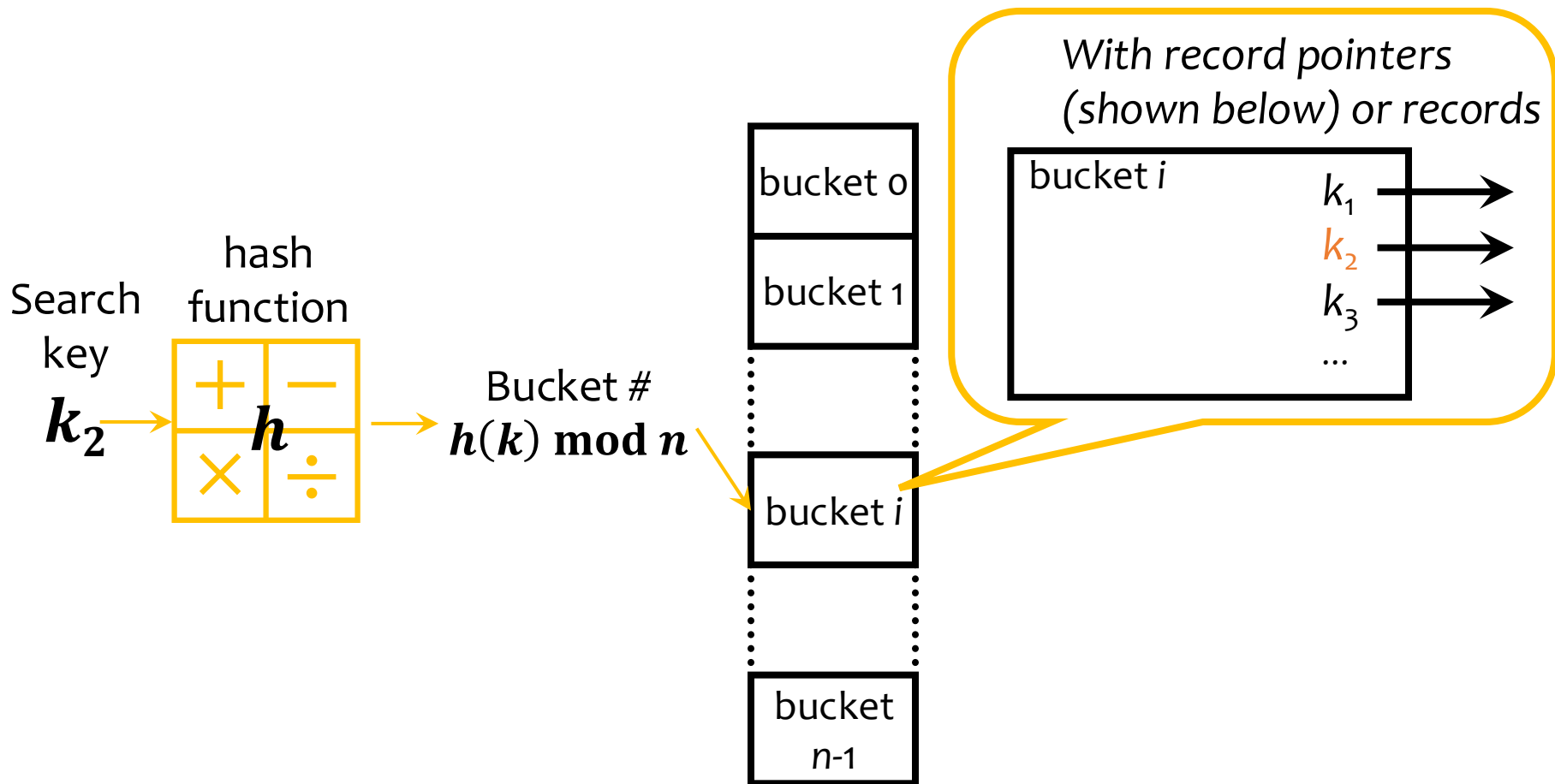
Outline

- Types of indexes
 - Sparse v.s. dense
 - Clustering v.s. non-clustering
 - Primary v.s. secondary
- Index structure
 - Hash based – unsorted, for point queries
 - Layered based – sorted, for point and range queries
- How to use index

Static hashing

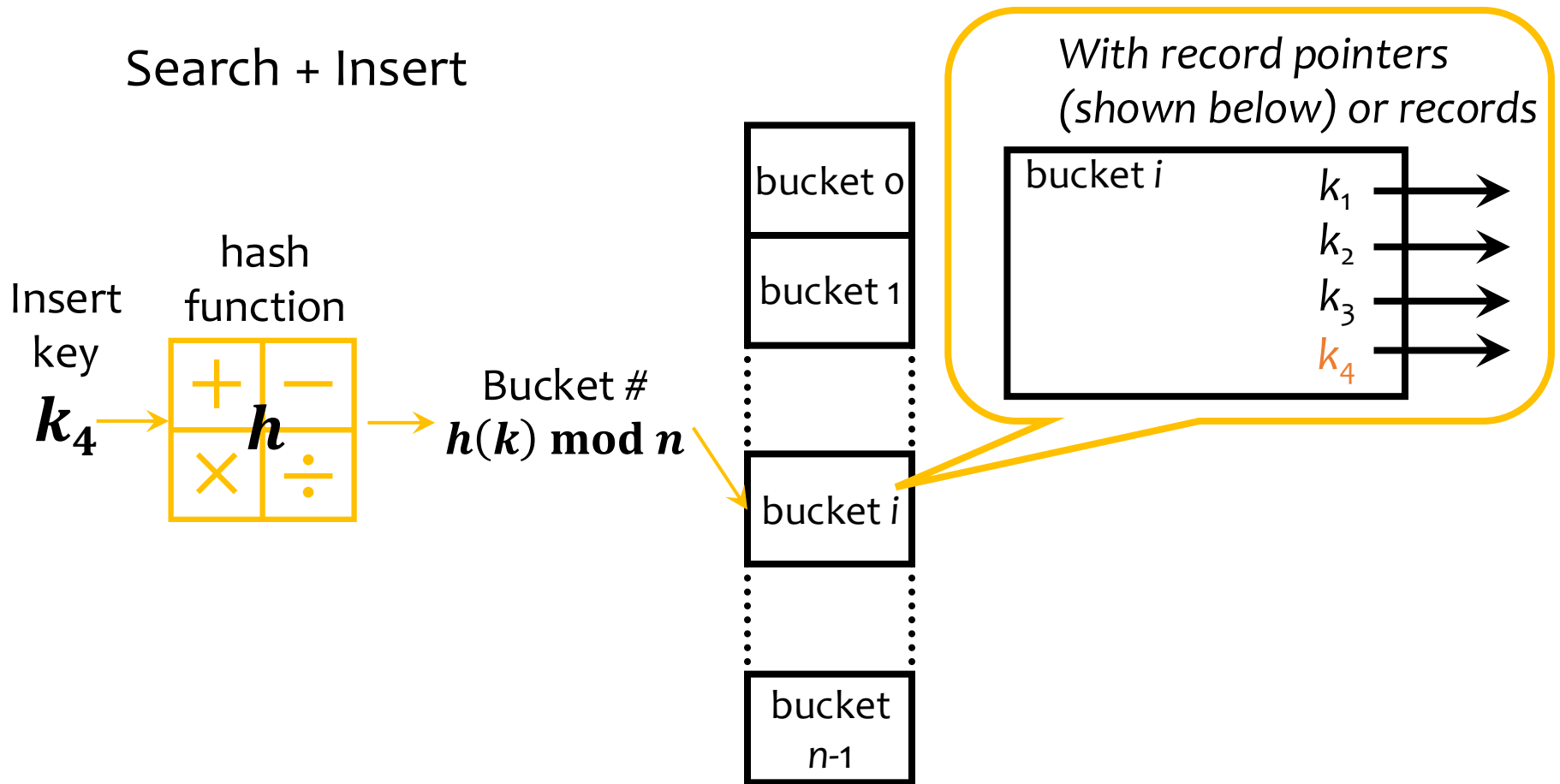
- Create n buckets to store the index. Each bucket occupies one page
- Use a hash function h to hash every index key; store in a bucket based on the hash
- Static hashing $\rightarrow$ the hashing technique remains unchanged

Static hashing: Searching



Static hashing: Inserting

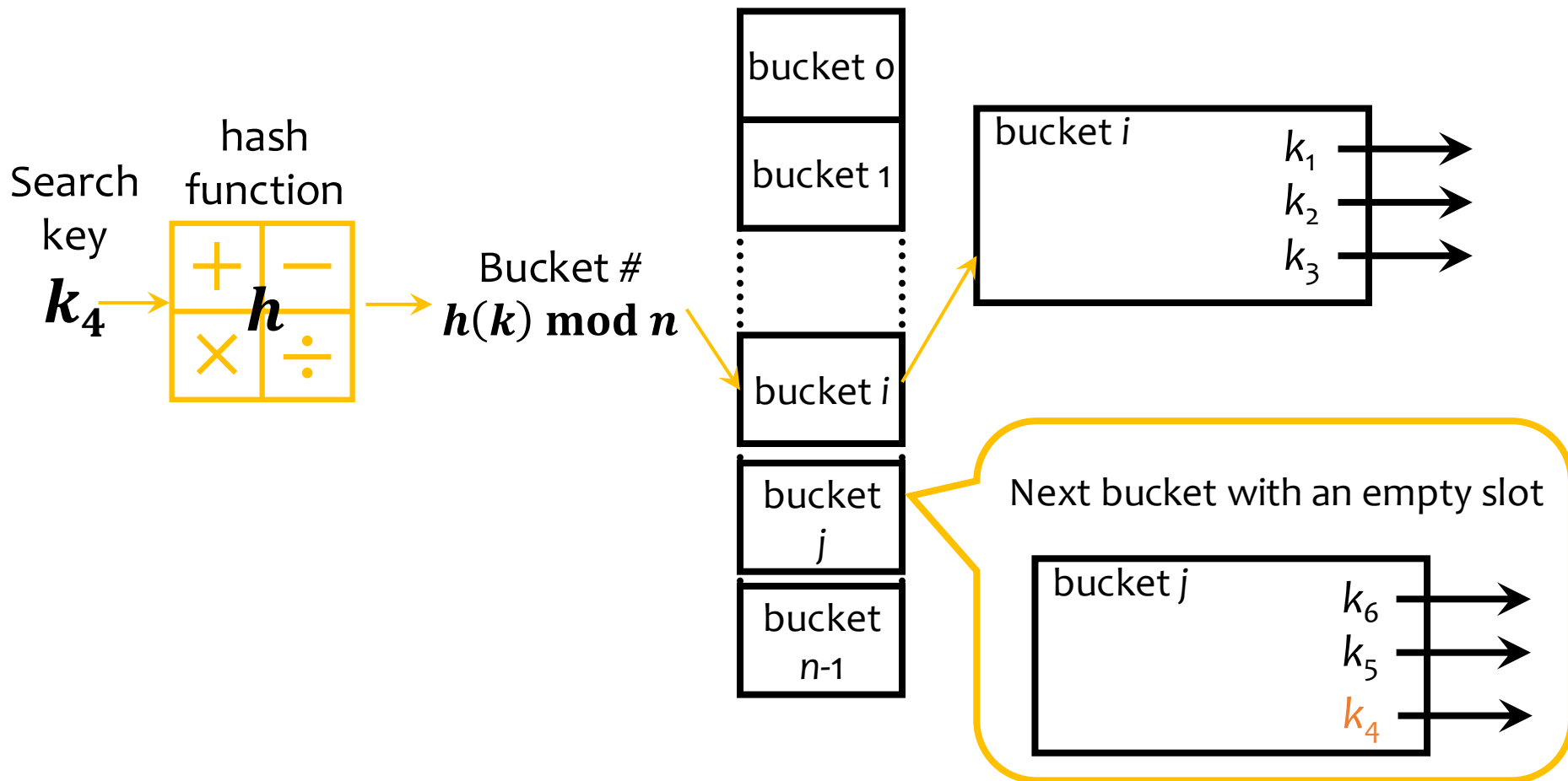
Search + Insert



What if a bucket is full??

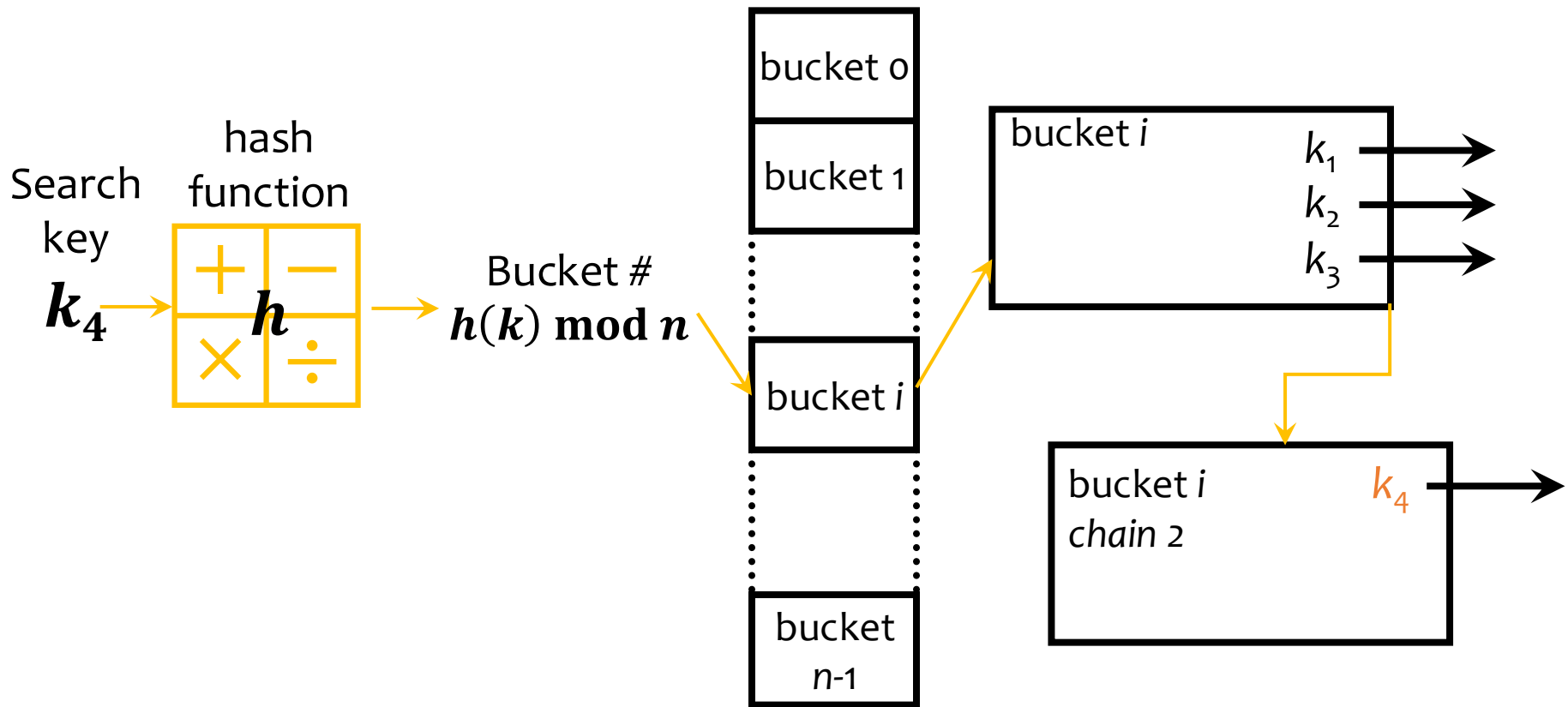
Option 1: Linear probing

Probe until we find a bucket with space!



Option 2: Bucket Chaining

Create a new bucket and chain it to the last one



In class exercises

Performance of static hashing

- Depends on the quality of the hash function
 - Best (hopefully average) case: **one I/O**
 - Worst case: **n I/O**, all keys hashed into one bucket
- What creates worst case behavior?
- Rule of thumb: keep utilization at 50%-80%
- 👉 How do we cope with growth?

Strawman solution

Rehash the whole table — using a new hash function, or changing n in $\text{mod } n$ to the new number of buckets

- Entries in an old bucket may now map to many different new buckets, causing lots of I/Os
- Cost of (re)building a giant hash table on external storage $\approx$ sorting (later in this course)

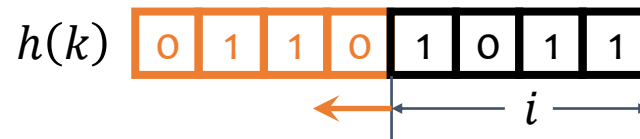
👉 Is it possible to **reduce data movement**?

Extensible hashing

- Instead of using a hash function's entire output to determine the bucket a key belongs to, **use only a few bits to determine the bucket**
- As the database changes and more keys are inserted (or deleted), dynamically adjust the number of bits

Extensible hashing

- Have hash function h output a large number of bits, but **only use the lowest i bits**, and **dynamically change i** as needed to create new buckets and pointers



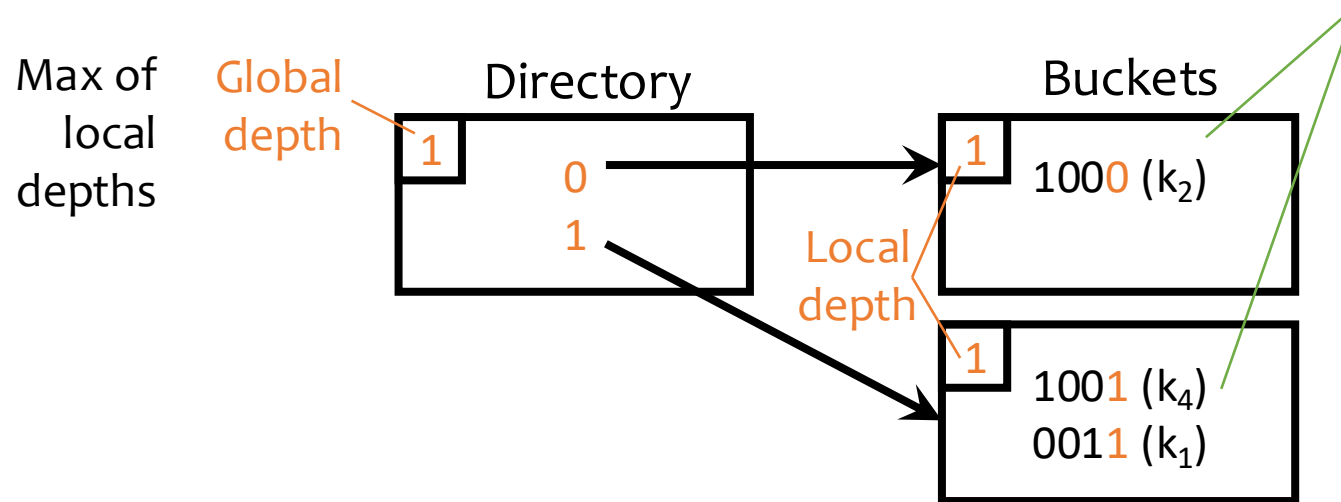
- E.g., $h_4(k) = 1011$, then $h_5(k)$ is either **0**1011 or **1**1011 $\Rightarrow$ contents in one old bucket can now go to either of the two new buckets!

Extensible hashing

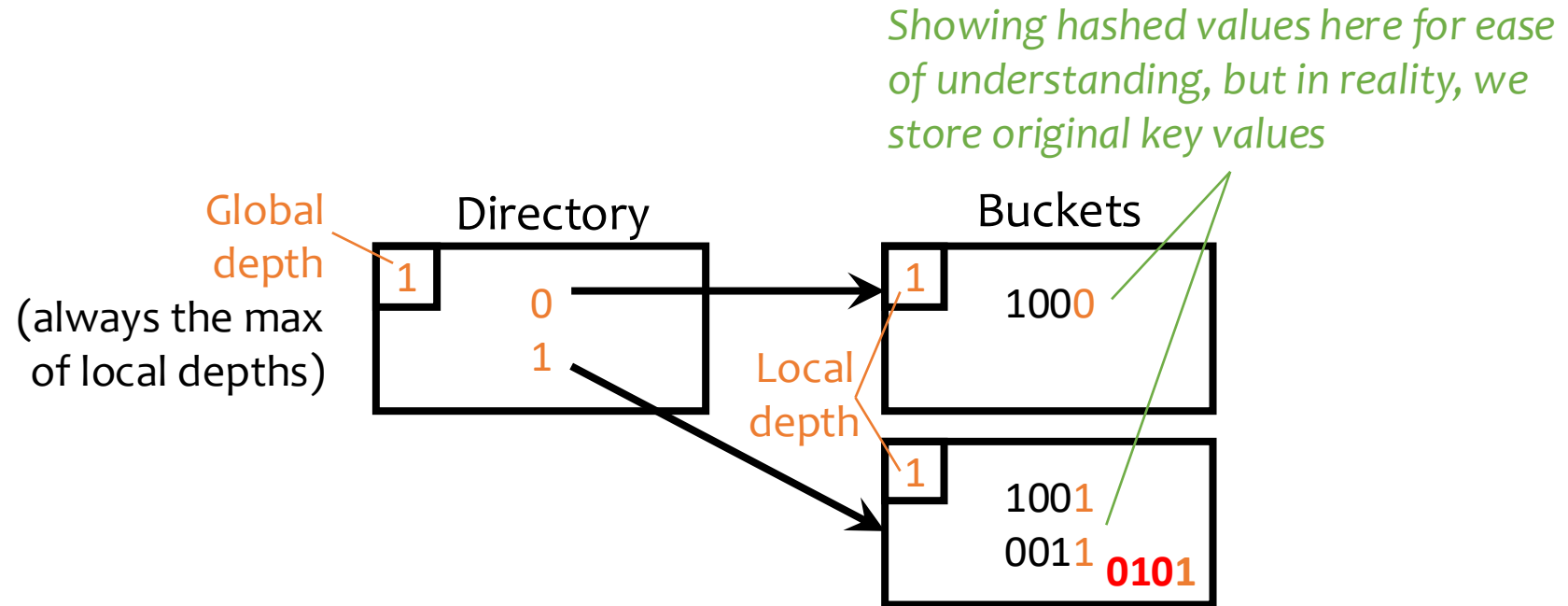
- How many bits to look at? And the pointers for each bucket?

- Global directory!

Showing hashed values here for ease of understanding, but in reality, we store original key values

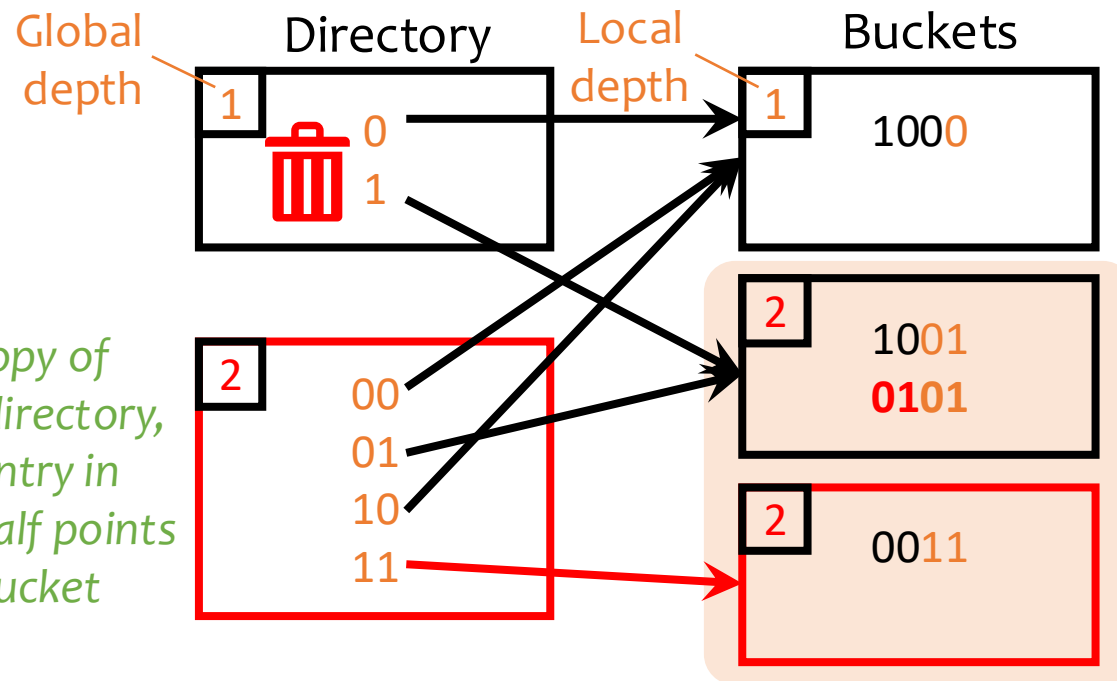


Extensible hashing example – 1



- Insert k with $h(k) = \mathbf{0101}$
- Bucket too full? **Split!**

Extensible hashing example – 2



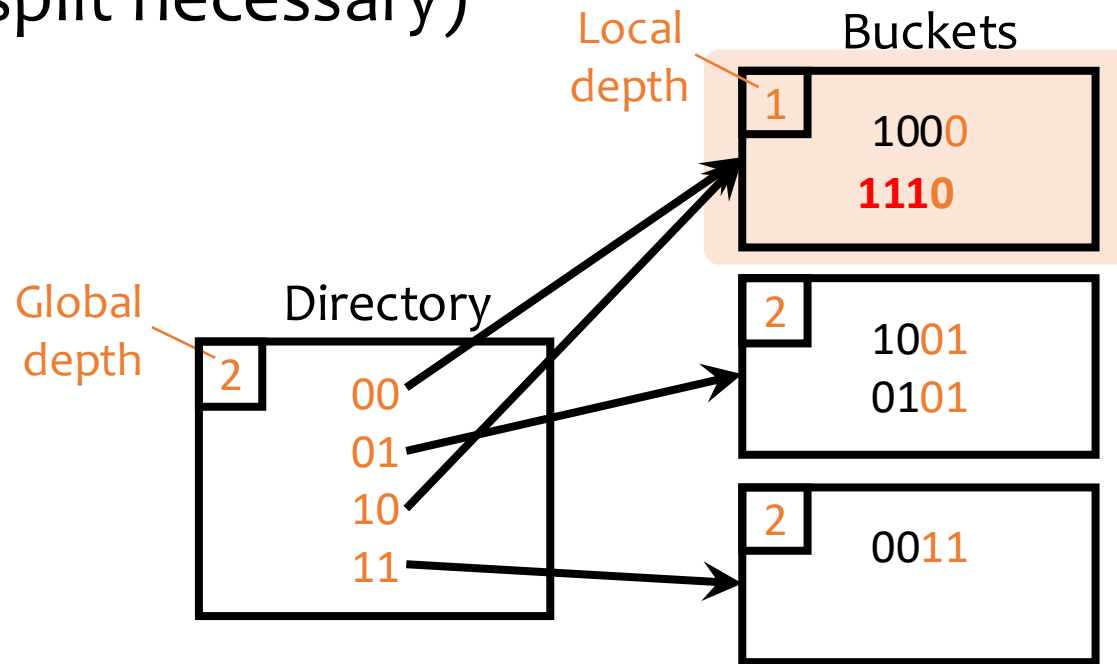
Each half = copy of the original directory, except one entry in the second half points to the new bucket

Upon split:

- ++local depth, redistribute contents, and ++global depth (double the directory size) if max local depth changes

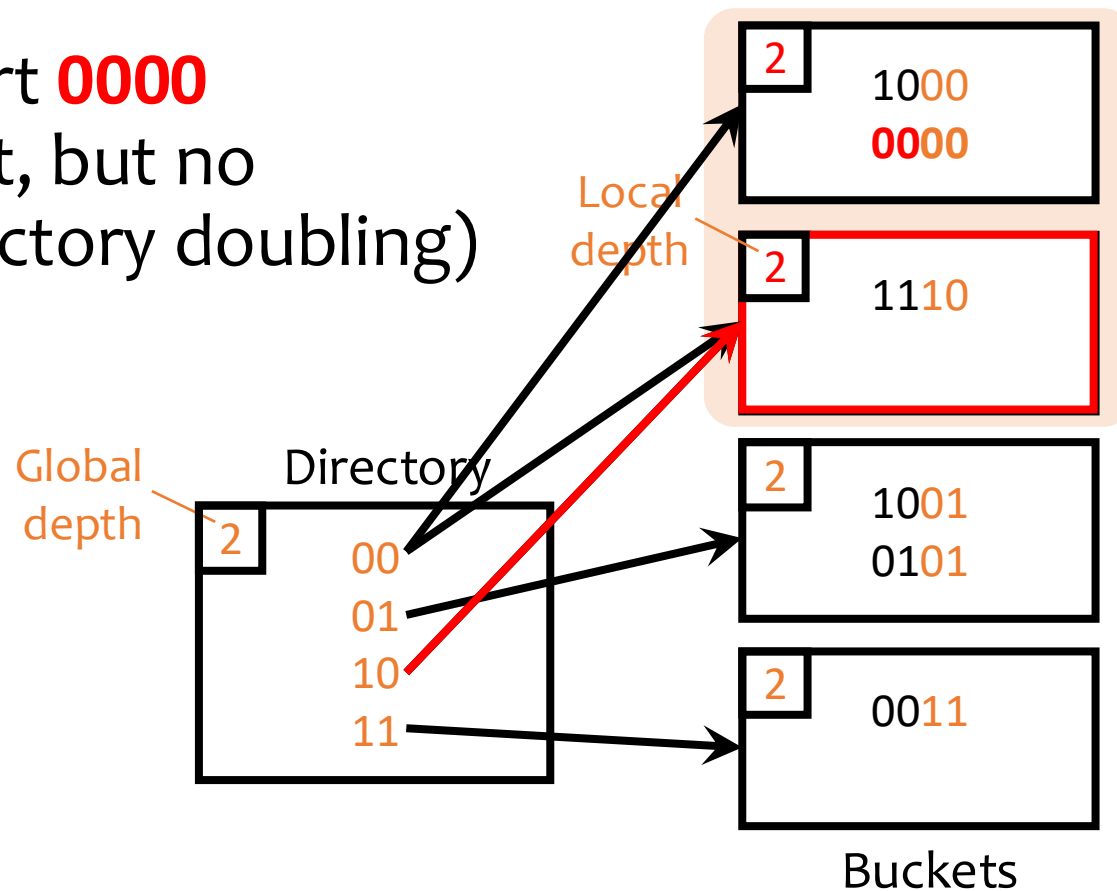
Extensible hashing example – 3

- Insert **1110**
(no split necessary)



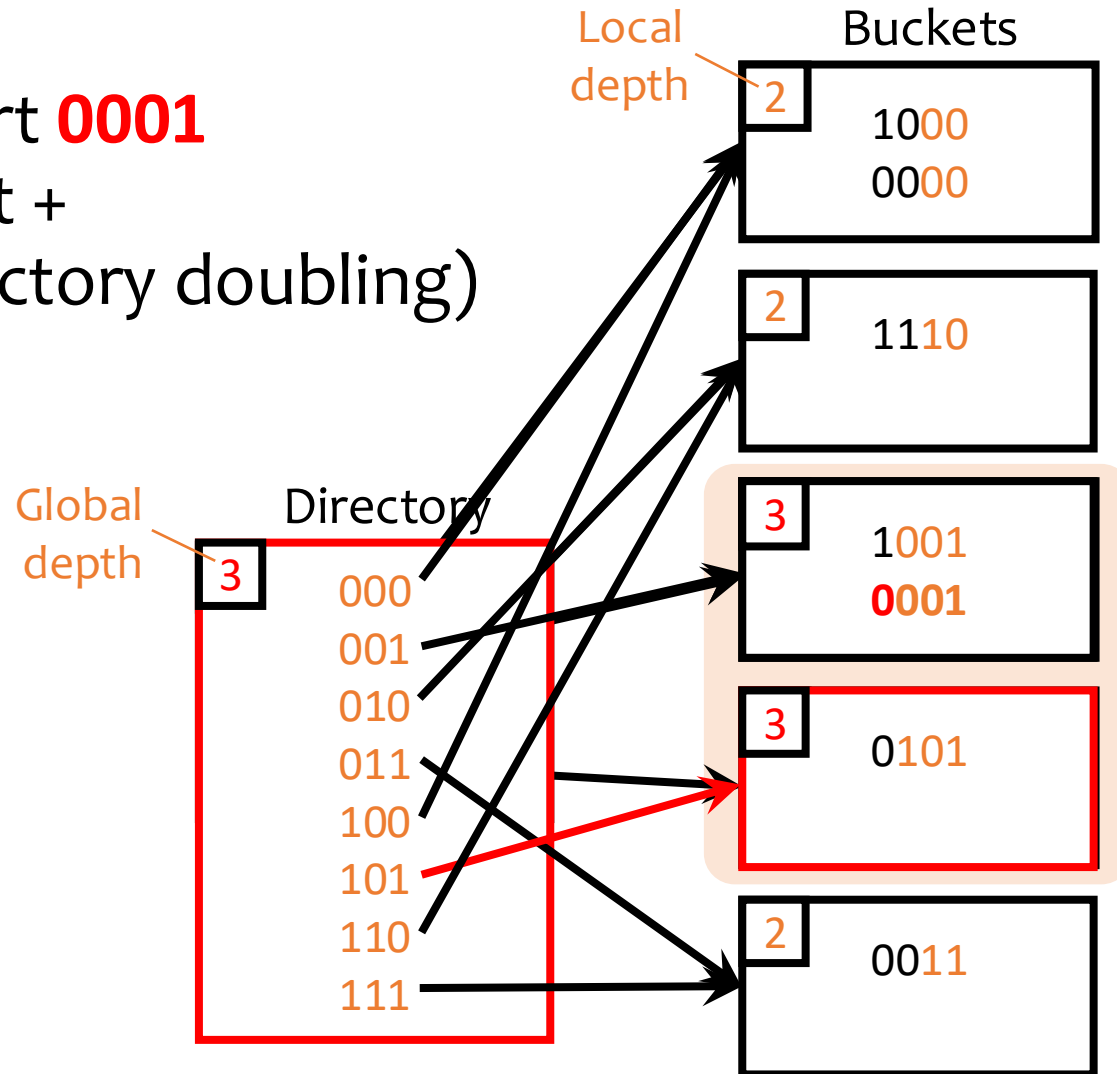
Extensible hashing example – 4

- Insert **0000**
(split, but no
directory doubling)



Extensible hashing example - 5

- Insert **0001**
(split +
directory doubling)



Extensible hashing – deletions

Just the reverse of insertions

- If the bucket becomes too empty:
 - Merge with “sibling” (differing only on the leading bit)
 - Adjust any pointer from the directory as needed
 - – – local depth
 - If possible, – – global depth and half the directory
 - Invariant: global depth = max of all local depths

In class ex: deletions

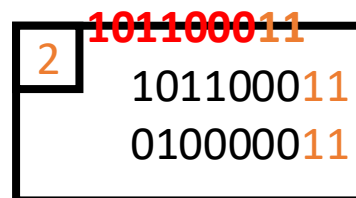
Summary of extensible hashing

Pros:

- Handles growing/shrinking indexes
- No full reorganization

Cons:

- One more level of indirection through the directory
- Directory size still doubles/halves
- There are cases when doubling may not be enough and needs chaining



Linear hashing (optional)

- The hash table maintains a pointer that tracks the next bucket to split.
 - When any bucket overflows, split the bucket at the pointer location.
 - What happens to the overflowing bucket??
 - Bucket chaining!
- Sometimes split happens when utilization (avg. # entries per bucket / max # entries per block) exceeds a given threshold

Hash based indexes with range queries

- Select * Where age > 10 and age <= 20;
- Index lookup for 11, 12,... 20 and union/merge the results
- Can we do better? Pay the index lookup cost only once?

Summary

- Types of indexes
 - Sparse v.s. dense
 - Clustering v.s. non-clustering
 - Primary v.s. secondary
- Index structure
 - Hash based
 - Layered based
- How to use index