Indexing Techniques

Storing Tables

- B+ Trees store records with keys
- Keys should not have too many duplicates i.e. as unique as possible, though uniqueness is not required
- Normally, in leaf node, the key is just part of record and no duplicate copy is stored

Table to B+Tree

```
create table courses(
 dept varchar(10),
 cid smallint,
 name varchar(20),
 credits smallint,
 description varchar(1000),
 last_taught varchar(20),
 primary key(dept, cid));
```

- This would be stored as a B+ tree with the *key* (dept, cid)
- Index (interior) nodes will have key values and pointers
- Leaf nodes will just have the records ordered by key.
- DBMS can easily extract keys (or any attributes) from records

Secondary Index as a Table

- We can think of secondary index as a table of type
  - Table_SI(*index attribute(s), rid-list*) or
  - Table_USI(*index attribute(s), rid*)
- The structure of this table would be exactly as expected
  - B+ tree with index_attributes in index nodes and the index_attributes + rid(s) in the leaf nodes

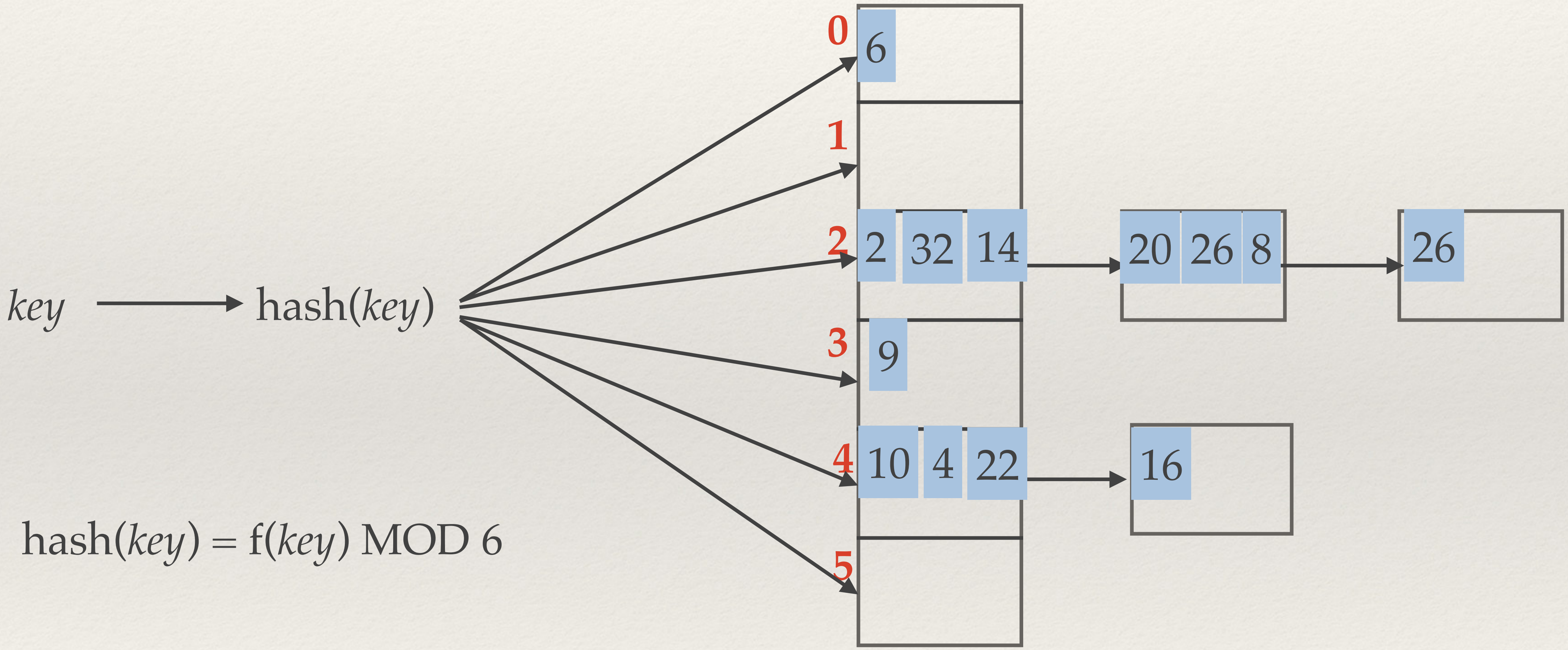
Hash-Based Indexing

- Tree and Hash based indexing are two main approaches
- B+ Tree is the main kind for tree-based indexing
- On to Hash-based indexing …

Static Hash Based Storage

- Let M be the expected number of pages in the file (table)
- hash(*key_val*) mod M = bucket to which record with key = *key_val* would be allocated
- If the initially allocated page for the bucket is full, then add more pages to that bucket
  - like a heap file, but per bucket

Static Hashing



hash(*key*) = f(*key*) MOD 6

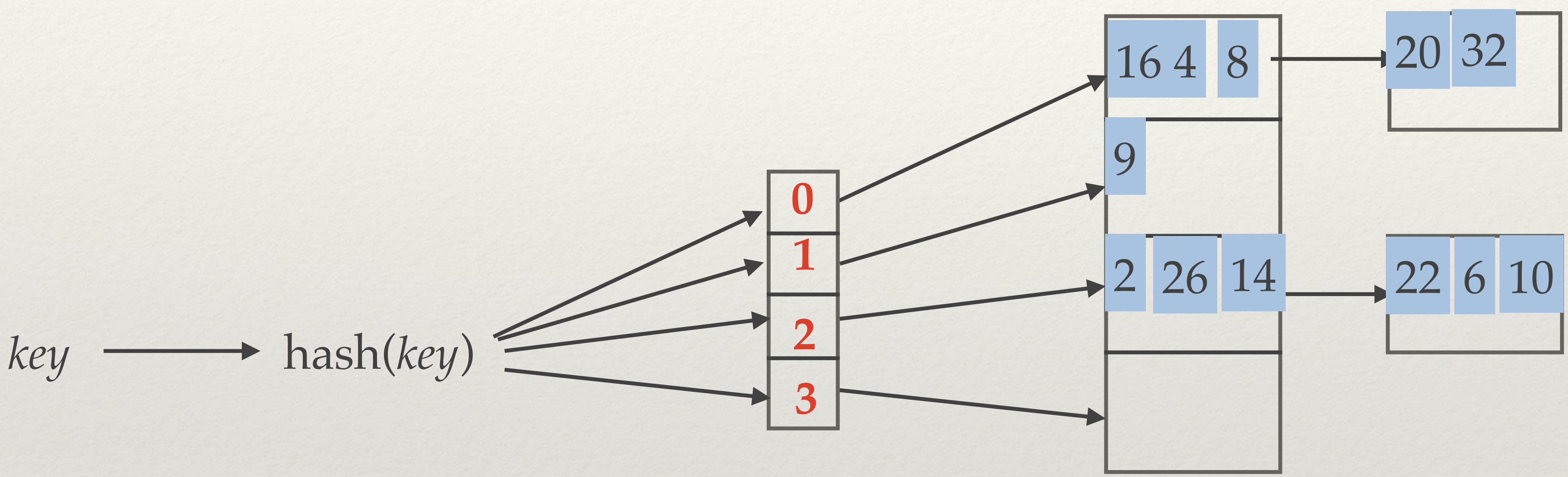
Static Hashing

- Hash function chosen for even distribution
- Long chains can form quickly when storing regular data
- Not adjustable to data volumes
- BUT
  - for non-unique secondary indices i.e. (key, *rid-list*) kind, the number of buckets is sort-of fixed, so it would work

Extendible Hashing

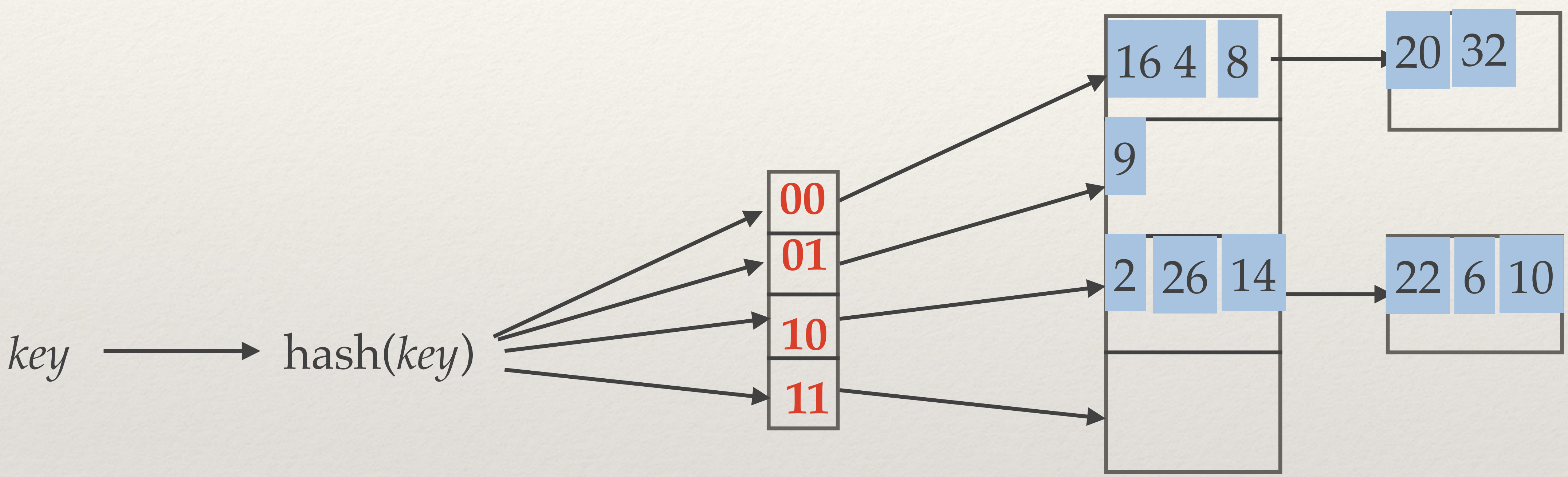
- When bucket (the primary page) becomes full, why not split the page (like B+ tree)?
  - We can double the number of buckets…
    - by introducing another level of indirection
  - Instead of buckets, the hash function points to a directory (of pointers entry).

Static Hashing with Directory



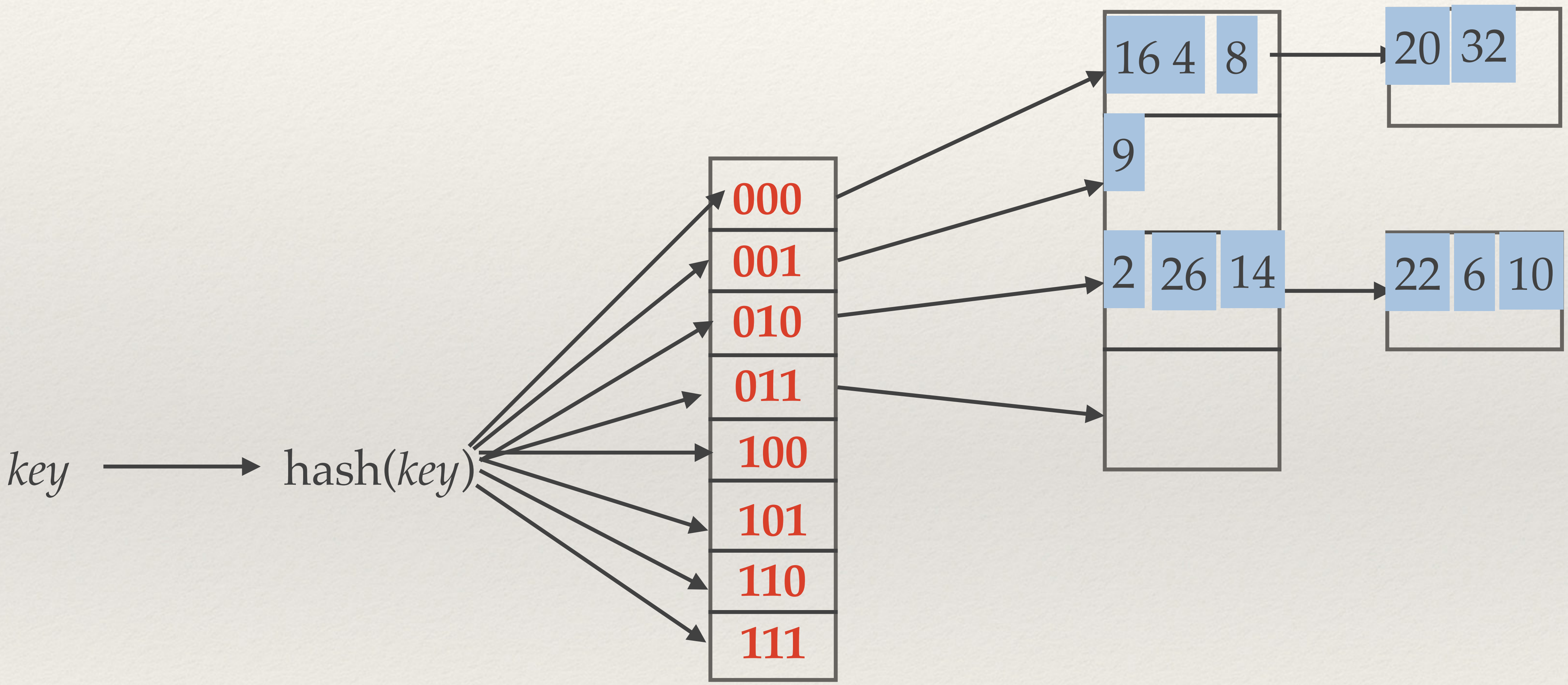
hash(*key*) = f(*key*) MOD 4

Static Hashing with Directory



hash(*key*) = f(*key*) MOD 2^{32} but using last 2 bits

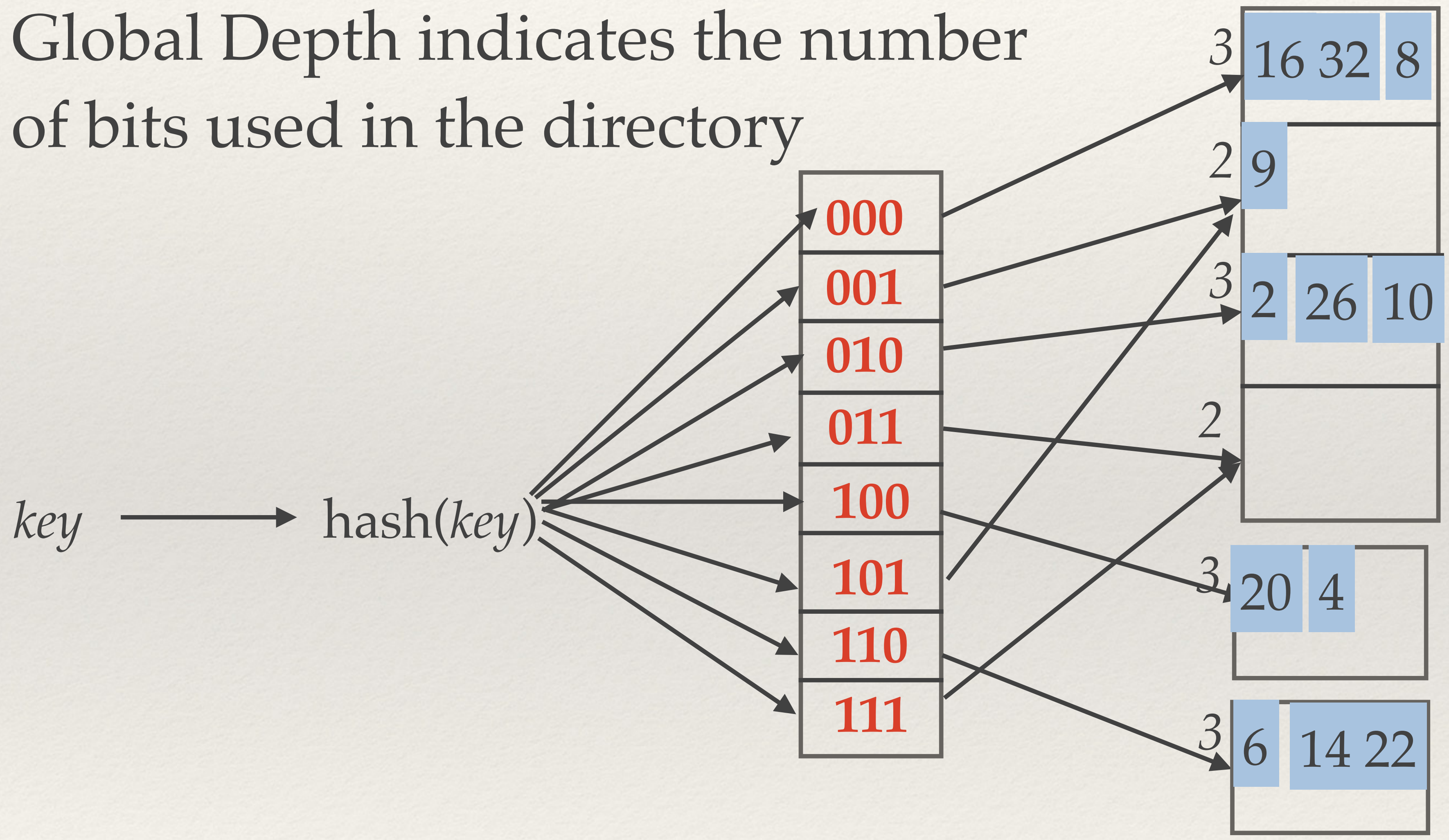
Expanding Directory



hash(*key*) = f(*key*) MOD 2^{32} but now using last 3 bits

Extendible Hashing

- *LOCAL DEPTH* indicates the number of bits used by the bucket
- Global Depth indicates the number of bits used in the directory



hash(*key*) = f(*key*) MOD 2^{32} but now using last 3 bits

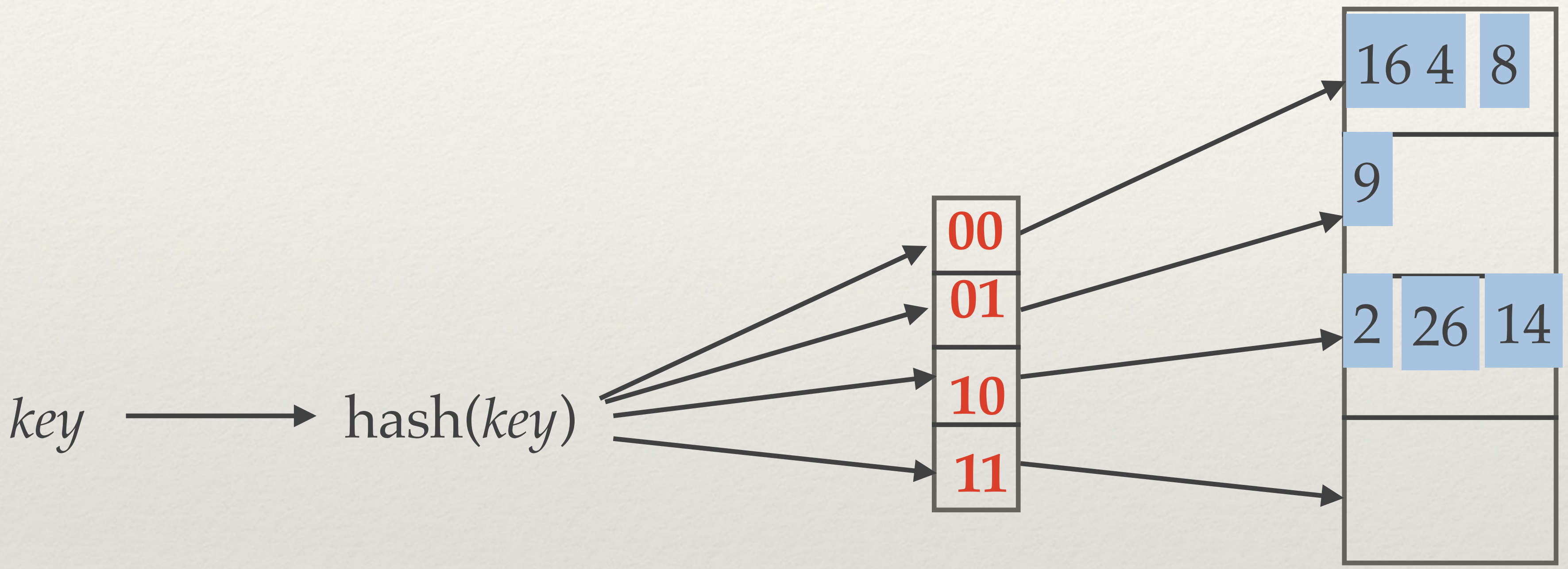
Local and Global Depth

- Last n bits of hash tell us that the *key* belongs to which directory
  - For directory: use #bits = global depth
  - For bucket: use #bits = local depth (entry $\in$ bucket)

Bucket Splits

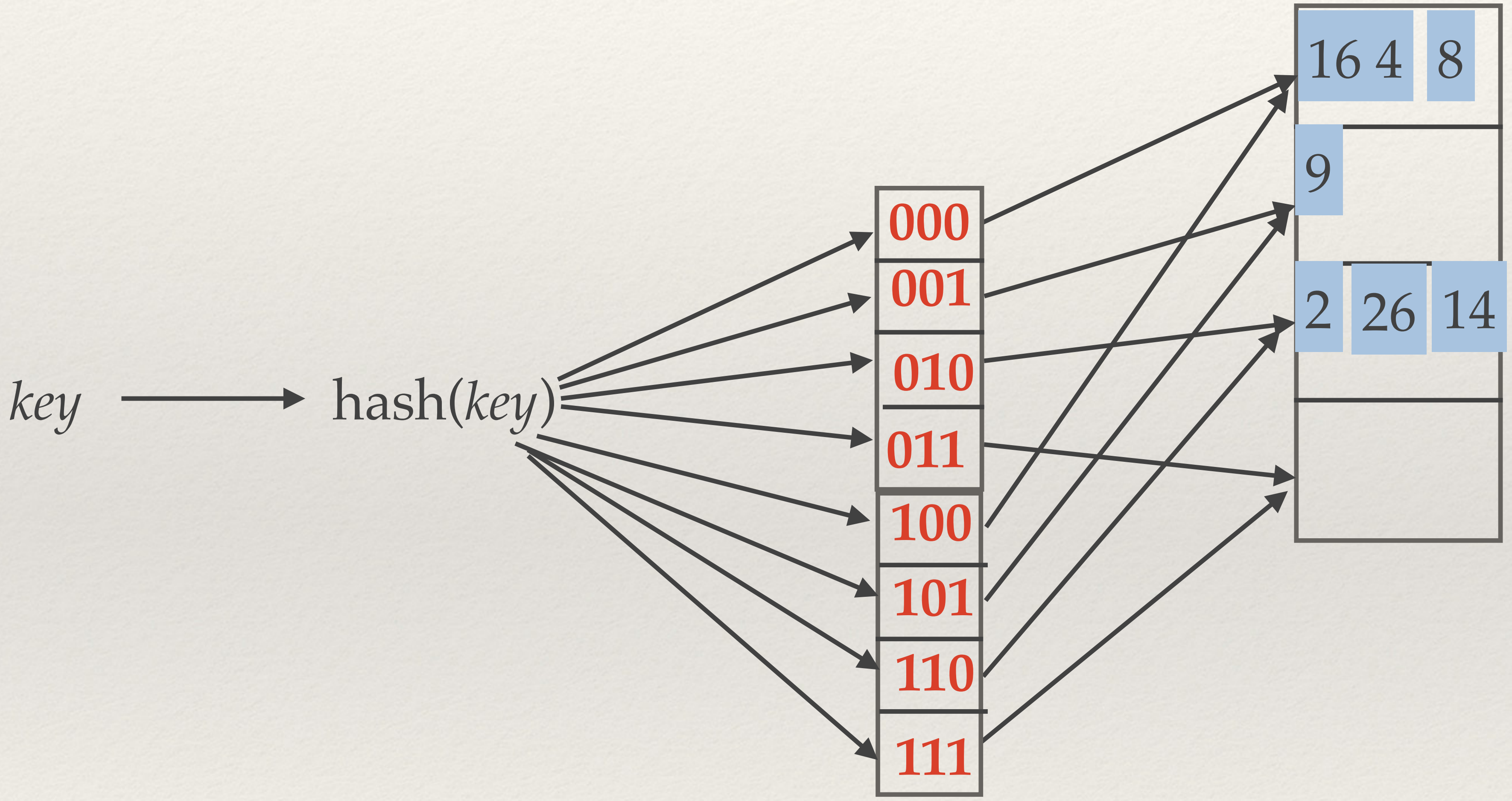
- Bucket may need to split upon insert
- If local depth < global depth, add a new bucket and move records to new bucket per the extra bit of hash (local depth now increases by 1)
- If they are same then the directory needs to double (simply copy it and then “fix” the pointers to new page)

Example



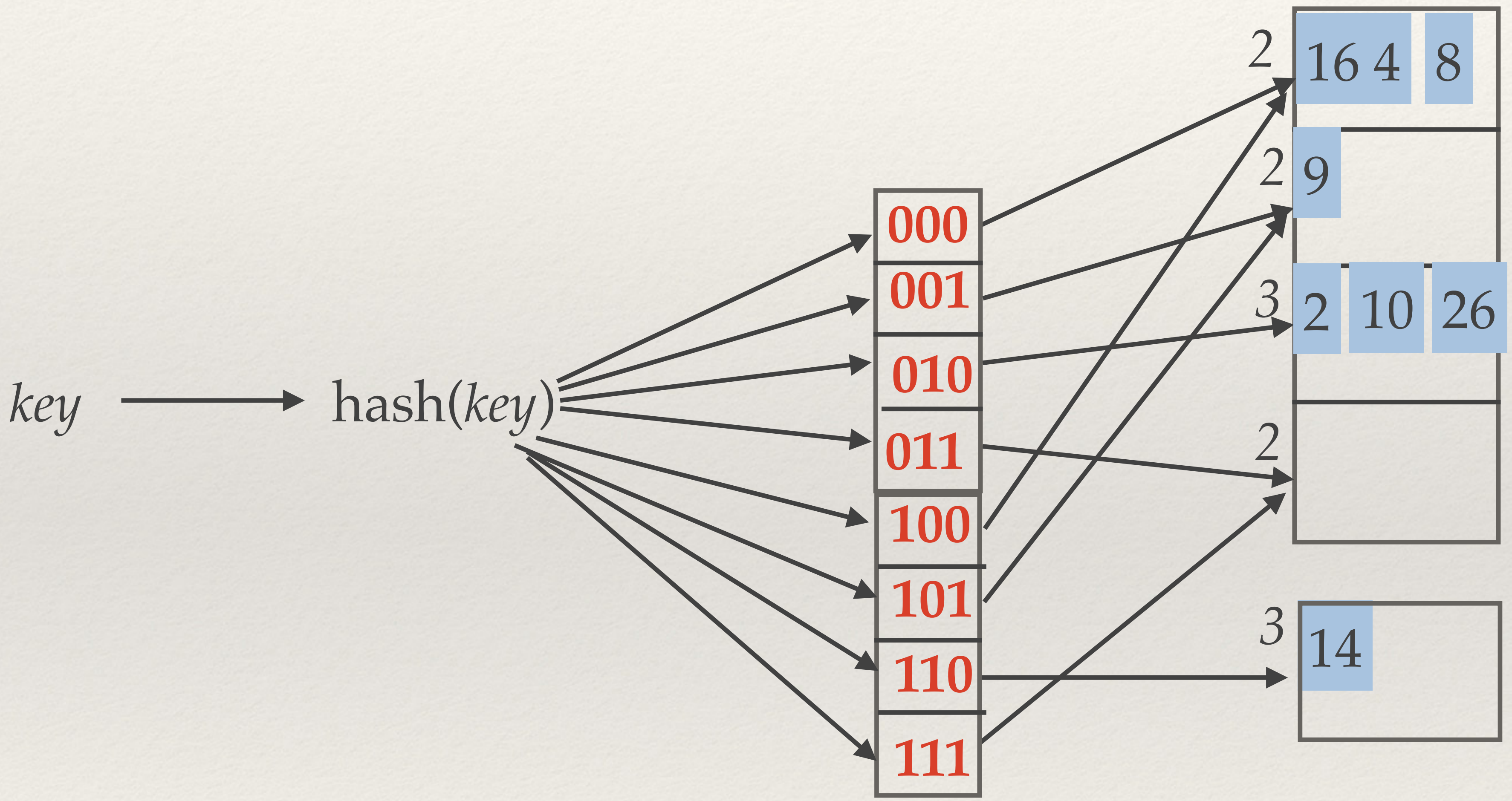
- Record 10 needs to be inserted in bucket "**10**"
- Local depth = global depth = 2

Example: Doubling of directory



- Directory split by copying
- global depth is now 3

Example: Moving Data and Adjustment



- Bucket 10 is split into 010 and 110
- Records are redistributed
- Local depth is increased by 1 for split buckets
- Pointer to new bucket is fixed

Extendible Hashing

- If directory is in memory, equality search still takes only one I/O. (Otherwise, 2)
- Upon delete if the bucket is empty, it can be merged with its “partner” (ignoring the first bit of hash)
  - If all directory elements point to the same bucket as its split partner, can halve the directory too

Bitmap Index

rid	dept	cid	sid	semid	instructor	grade
1	CS	564	2016001	10	Codd	A
2	CS	564	2012144	10	Codd	C
3	CS	564	2014101	10	Codd	A
4	CS	367	2014101	8	Aho	A
5	CS	367	2012144	8	Aho	C
6	CS	367	2015001	8	Aho	B
7	PolSci	104	2015001	10	Madison	A
8	PolSci	104	2014101	10	Madison	A
9	PolSci	104	2012144	10	Madison	B
10	PolSci	104	2012144	8	Franklin	F
11	Math	234	2012144	1	Newton	D
12	Math	201	2012144	2	Laplace	D

- Imagine a making an index on dept
- Represent the rid's as a bitmap
- 8 byte *rid*, 12 rows => 64 vs. 6 bytes

dept	rid-list
PolSci	7,8,9,10
CS	1,2,3,4,5,6
Math	11,12

dept	rid-list-bitmap
PolSci	000000111100
CS	111111000000
Math	000000000011

Why Bitmap Index

- WHERE semid = 8 AND
  dept = 'CS'
- AND the two bitmaps to get 000111000000
- Then get the rows
- Quick filtering on multiple not-so-unique columns
- Can use B+Tree, Hash Index

```
 semid  | rid-list-bitmap
--------+---------
 10     | 111000111000
  8     | 000111000100
  1     | 000000000010
  2     | 000000000001
```

```
  dept  | rid-list-bitmap
--------+---------
 PolSci | 000000111100
 CS     | 111111000000
 Math   | 000000000011
```

Other Indexing Techniques

- So far: Indexing = ToC or “Back of Book” kind
- Indexing can be generalized to mean any performance enhancing storage technique
  - implies automatic maintenance in face of updates
- E.g.
  - Teradata® Join Index - an index to speed up joins
  - Postgres Partial Index - on a constrained table

Relational Concepts and Indexing

- Candidate Key = unique identifier of row
  - Primary Key = chosen candidate key
  - uniqueness constraint = way to identify other candidate keys

Primary Key vs Index

- Primary Index is the way the table records are stored
  - need not be unique, but needs enough distinct values
- Primary Key makes for good PI for most entity tables
  - If a column different from PK is being used often, it could make for a good PI candidate
- For table representing “relations” i.e. with multi-column primary keys, it’s usually best to have a non-unique primary index on the most common joining FK

Secondary Index

- create [unique] index on <table>(<columns>)
- Unique constraints = unique secondary index
- Non-unique secondary index: for fast equality constraints on columns
  - Can be used for some kinds of aggregation operations
  - Can be ANDed with other SI's if there are equality constraints on different columns (not uncommon)
    - bitmap representation (bitmap index)