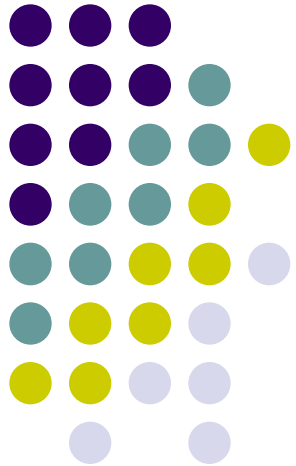
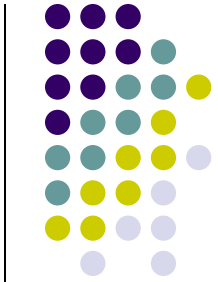
CMSC424: Database Design

Instructor: Amol Deshpande

amol@cs.umd.edu

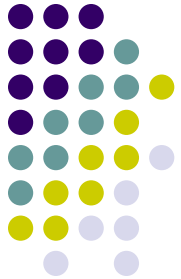


Spring 2020 – Online Instruction Plan



- Week 1 (March 30 – April 2):
  - File Organization and Overview of Indexes
  - B+-Trees
  - Hashing
  - Miscellaneous topics in Indexes
- Week 2: Query Processing
- Week 3: Transactions 1
- Week 4: Transactions 2
- Week 5: Parallel Database and MapReduce

Hash-based File Organization



Store record with search key *k*
in block number *h(k)*

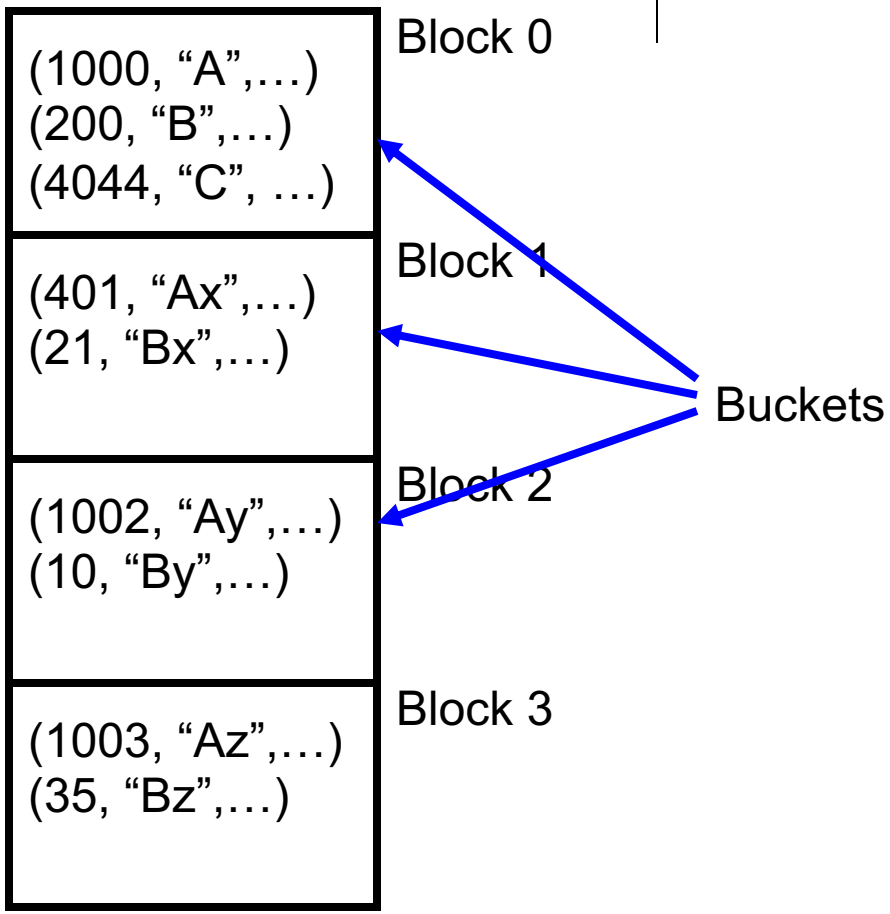
e.g. for a person file,
h(SSN) = SSN % 4

Blocks called "*buckets*"

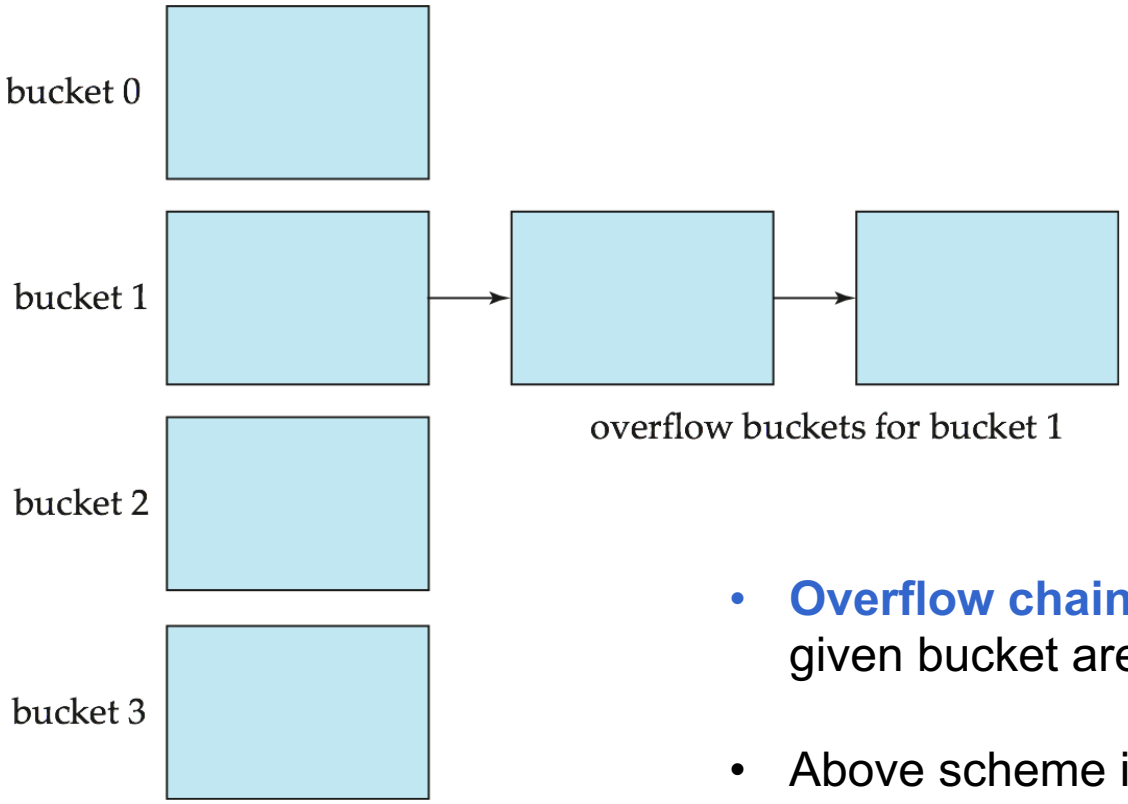
What if the block becomes full ?
Overflow pages

Uniformity property:
Don't want all tuples to map to
the same bucket
h(SSN) = SSN % 2 would be bad

Hash functions should also be **random**
Should handle different real datasets

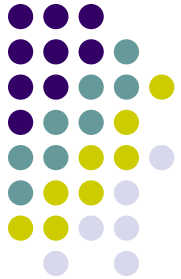


Overflow Pages



- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.

Hash-based File Organization



Hashed on “branch-name”

Hash function:

a = 1, b = 2, .., z = 26

h(abz)

= (1 + 2 + 26) % 10

= 9

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash Indexes

Extends the basic idea

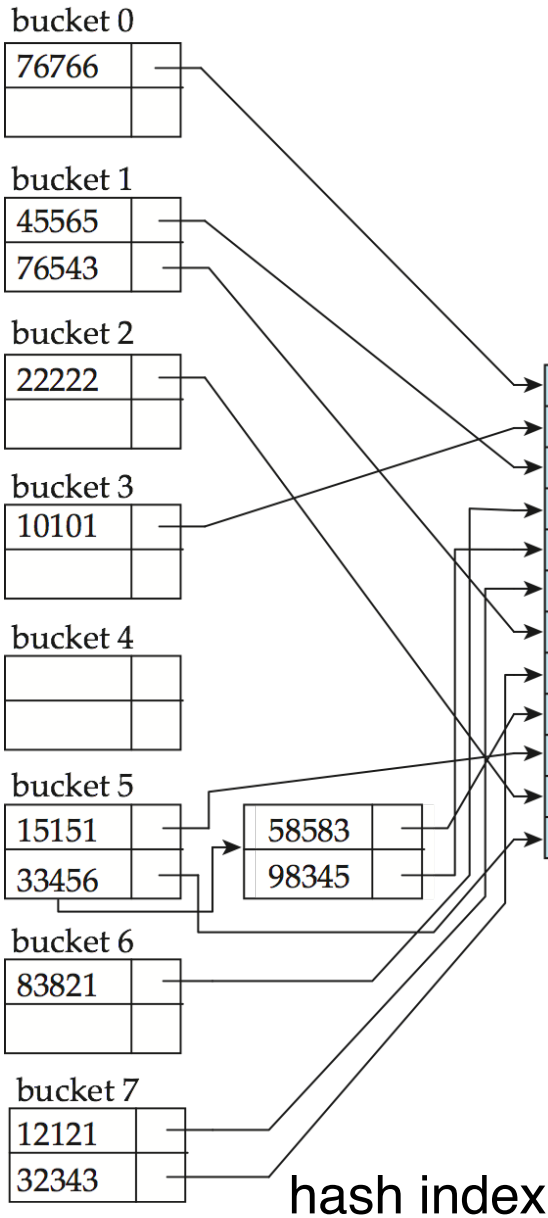
Search:

Find the block with search key

Follow the pointer

Range search ?

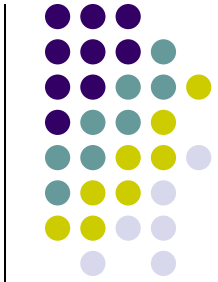
a < X < b ?



76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

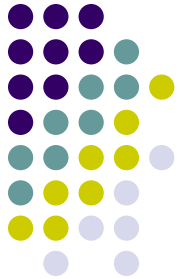
hash index on *instructor,* on attribute *ID*

Hash Indexes



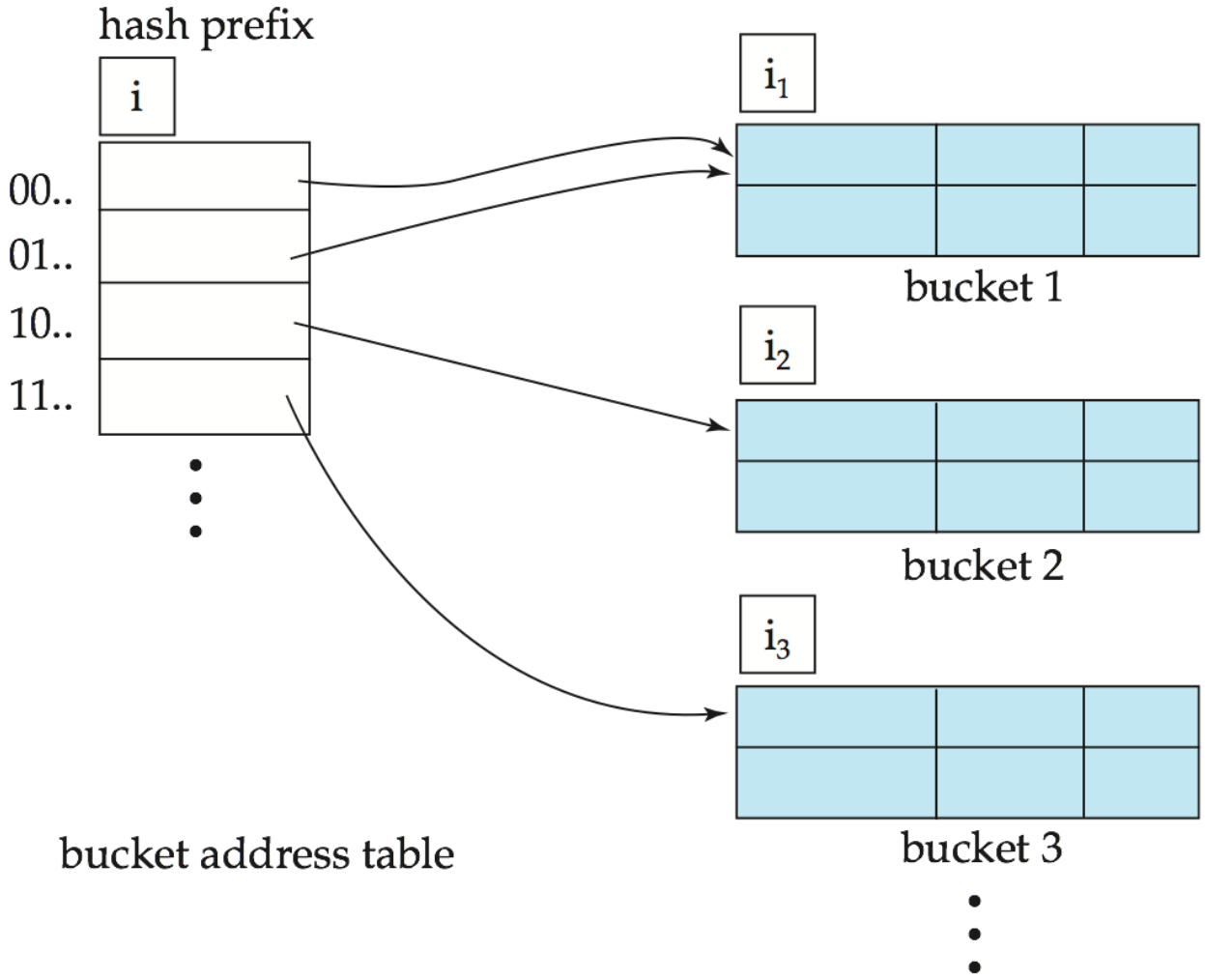
- Very fast search on equality
- Can't search for "ranges" at all
  - Must scan the file
- Inserts/Deletes
  - Overflow pages can degrade the performance
  - Can do periodic reorganization (by modifying hash functions)
- A better approach is to use "dynamic hashing"
  - Allow use of a hash function that can be modified
  - We discuss one such technique: Extendable Hashing

Extendable Hashing



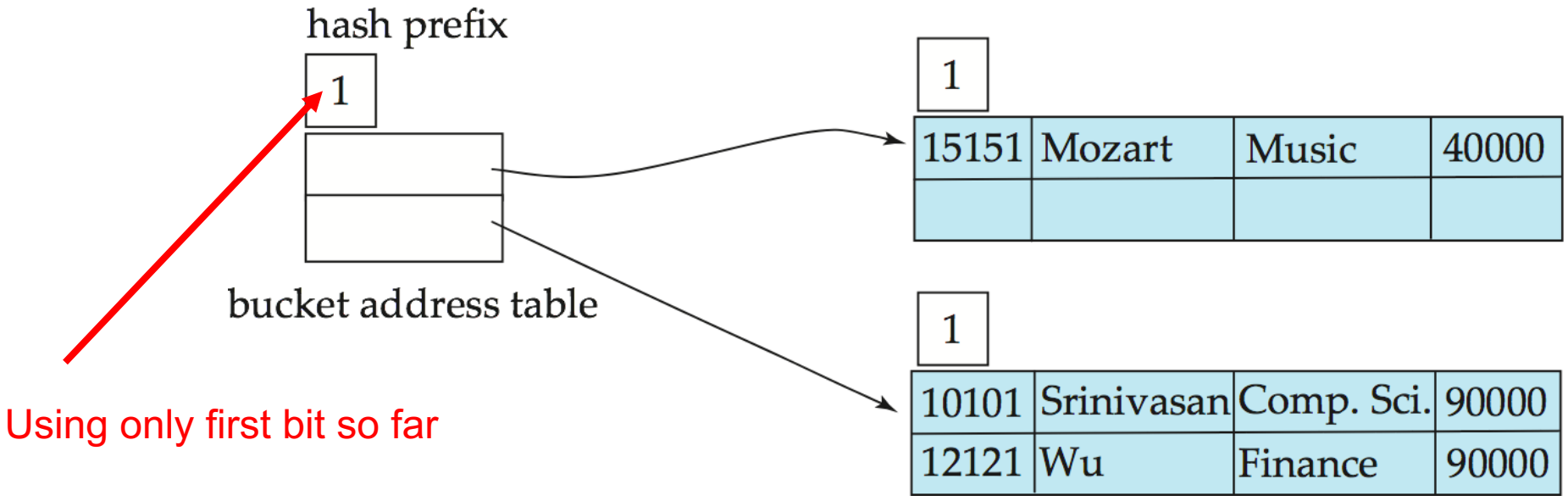
- Use a hash function that outputs a large number of bits, e.g., 32 bits or 64 bits
- However, only use a “prefix” of that hash function based on the size of the database
- Different parts of the database may use different length prefix
- When “inserting”, if the bucket becomes too big, split it and use an extra bit

General Extendable Hash Structure



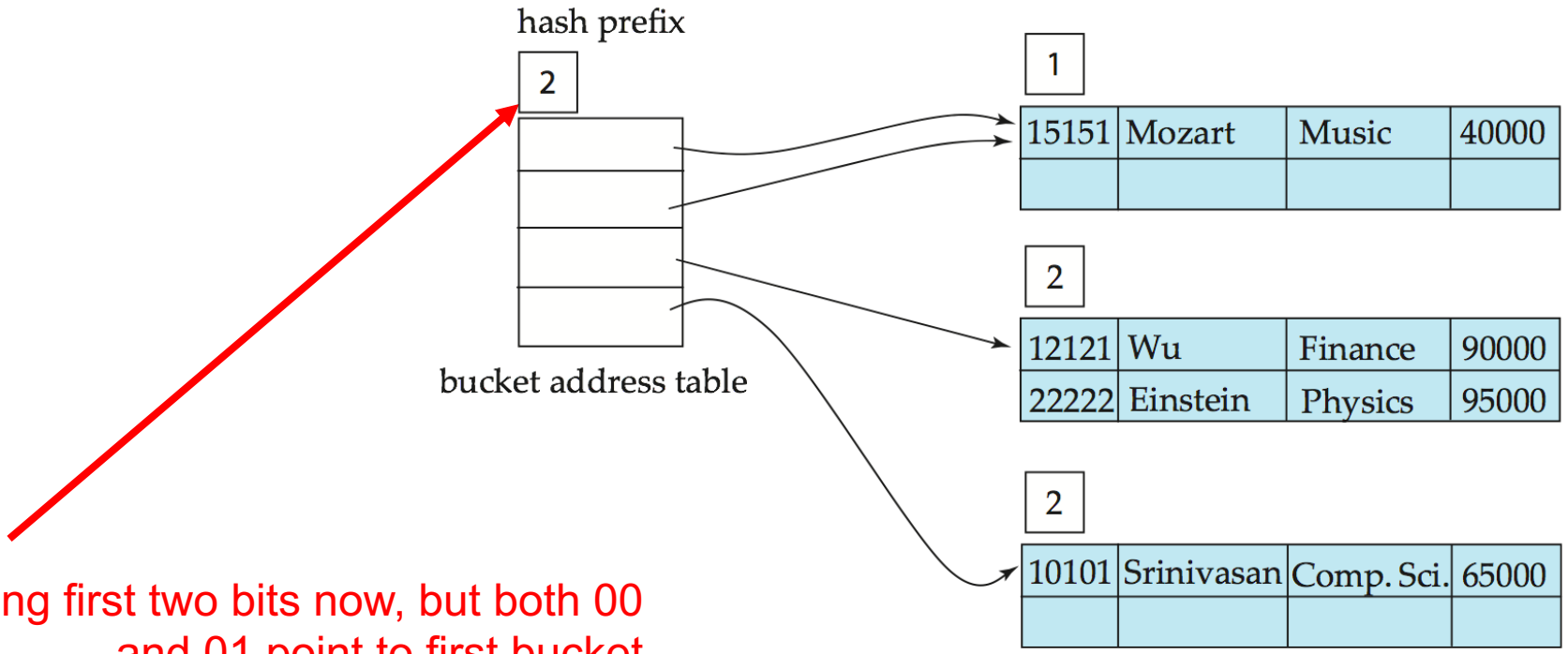
Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001



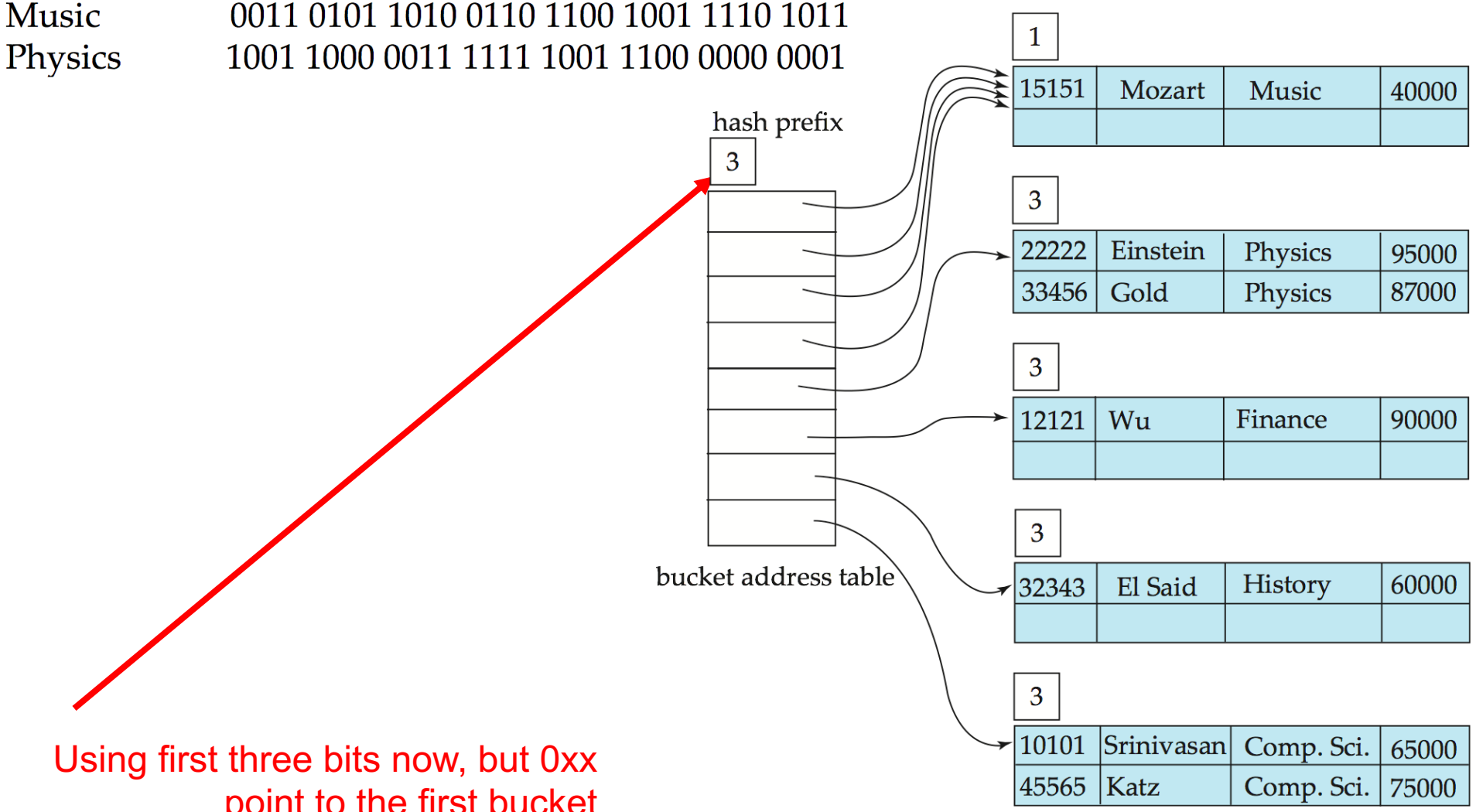
Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001



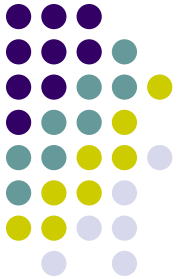
Using first two bits now, but both 00 and 01 point to first bucket

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001



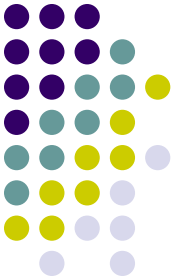
Using first three bits now, but 0xx point to the first bucket

Extendable Hashing vs. Other Schemes



- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

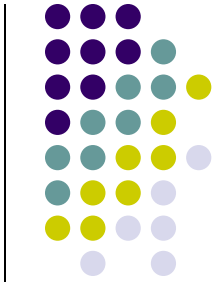
Comparison of Ordered Indexing and Hashing



- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

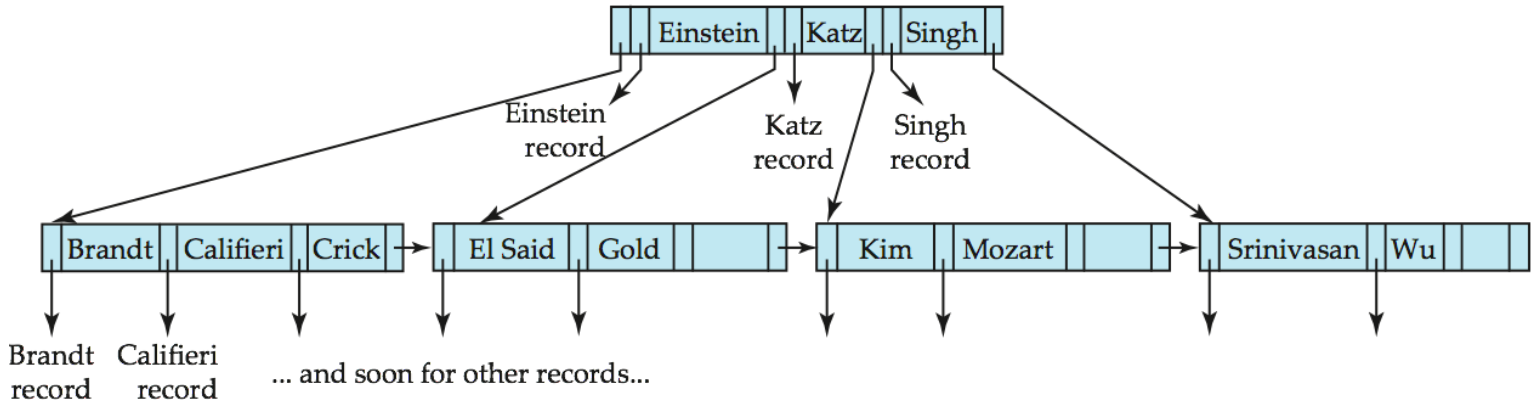
- Hashing very common in distributed settings (e.g., in key-value stores)

Spring 2020 – Online Instruction Plan

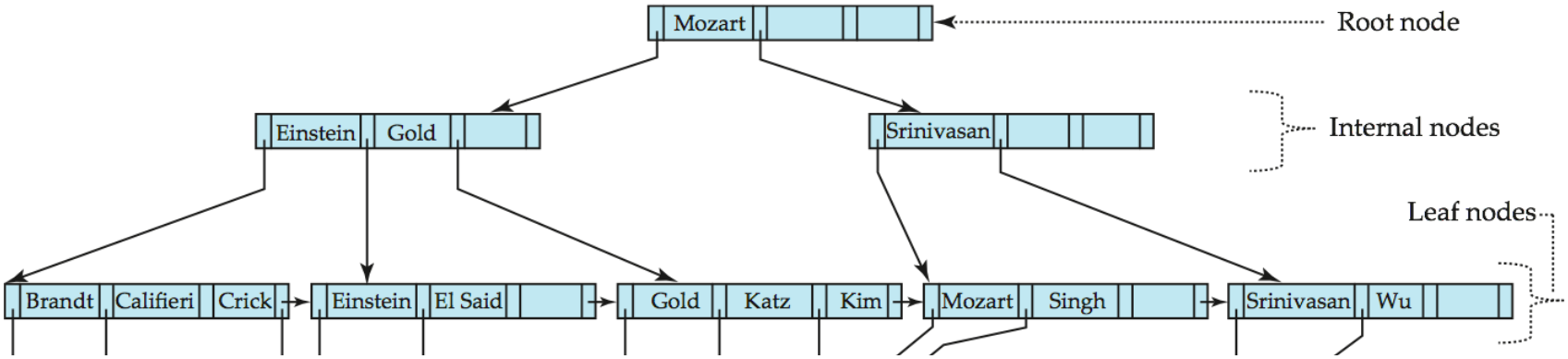


- Week 1 (March 30 – April 2):
  - File Organization and Overview of Indexes
  - B+-Trees
  - Hashing
  - Miscellaneous topics in Indexes
- Week 2: Query Processing
- Week 3: Transactions 1
- Week 4: Transactions 2
- Week 5: Parallel Database and MapReduce

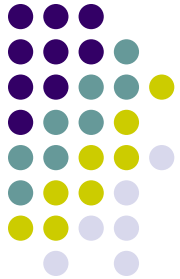
B-Tree Index Example



B-tree (above) and B+-tree (below) on same data – B-Trees have ”record pointers” at interior nodes



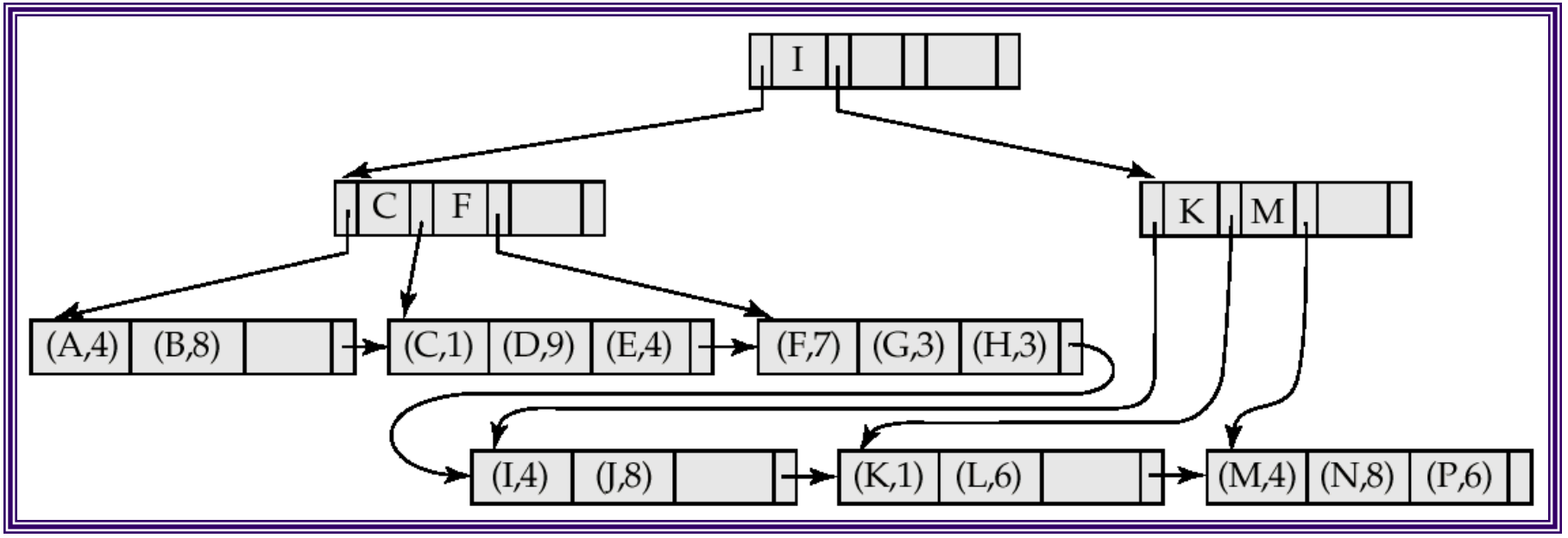
B-Tree Index Files (Cont.)



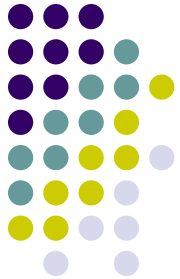
- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

B+-Tree File Organization

- Store the records at the leaves
- Sorted order etc..



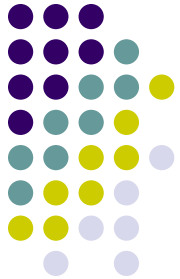
Multiple-Key Access



select *ID*
from *instructor*
where *dept_name* = “Finance” **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
  - Use index on *dept_name* to find instructors with department name Finance; test *salary* = *80000*
  - Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = “Finance”.
  - Use *dept_name* index to find pointers to all records pertaining to the “Finance” department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.
    - Called “INDEX-ANDING”

Indices on Multiple Keys



- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$, or
  - $a_1 = b_1$ and $a_2 < b_2$

- Ideal for something like:

  where *dept_name* = “Finance” **and** *salary* = 80000
- Can also efficiently handle

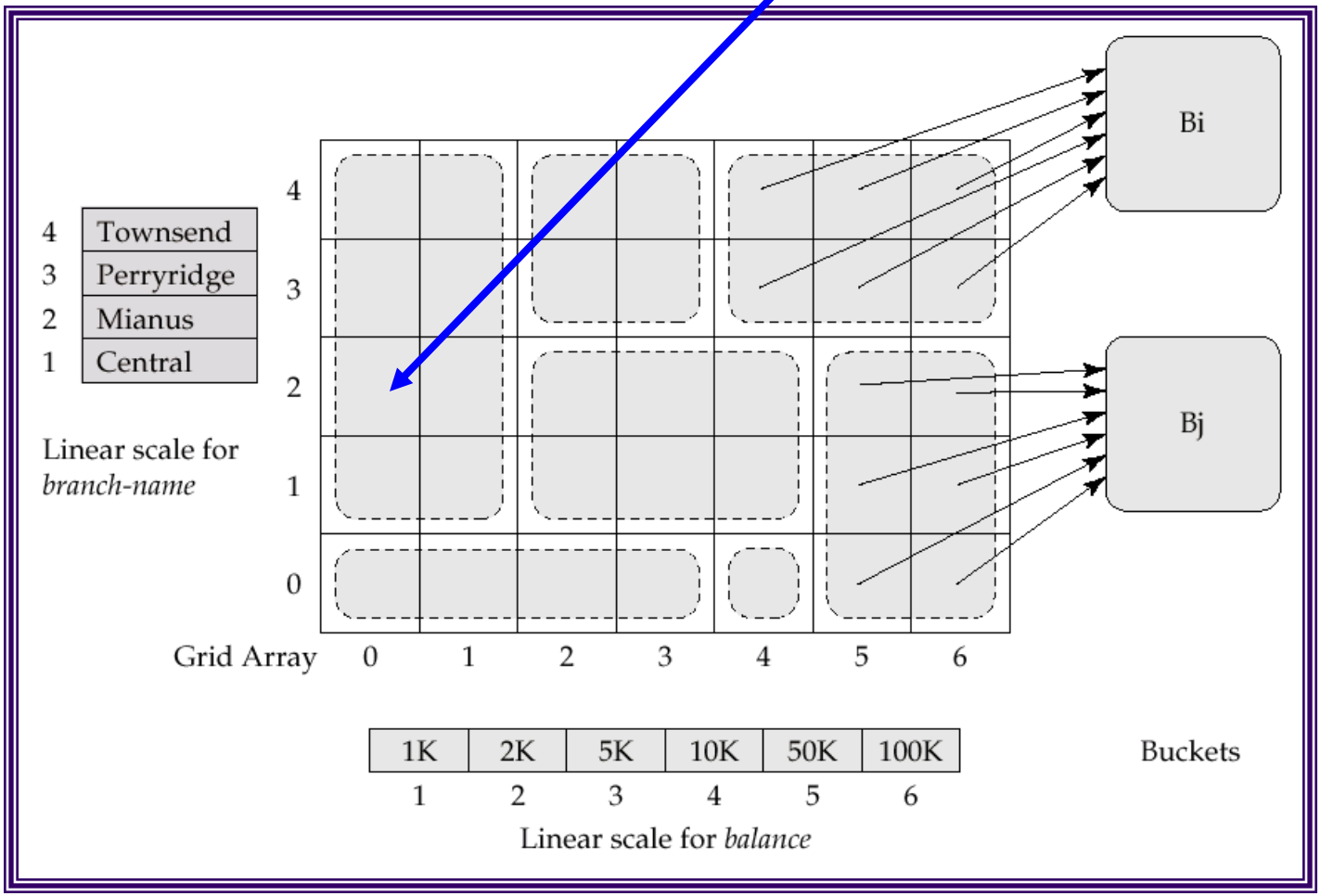
  where *dept_name* = “Finance” **and** *salary* < 80000
- But cannot efficiently handle

  where *dept_name* < “Finance” **and** *balance* = 80000

Grid Files

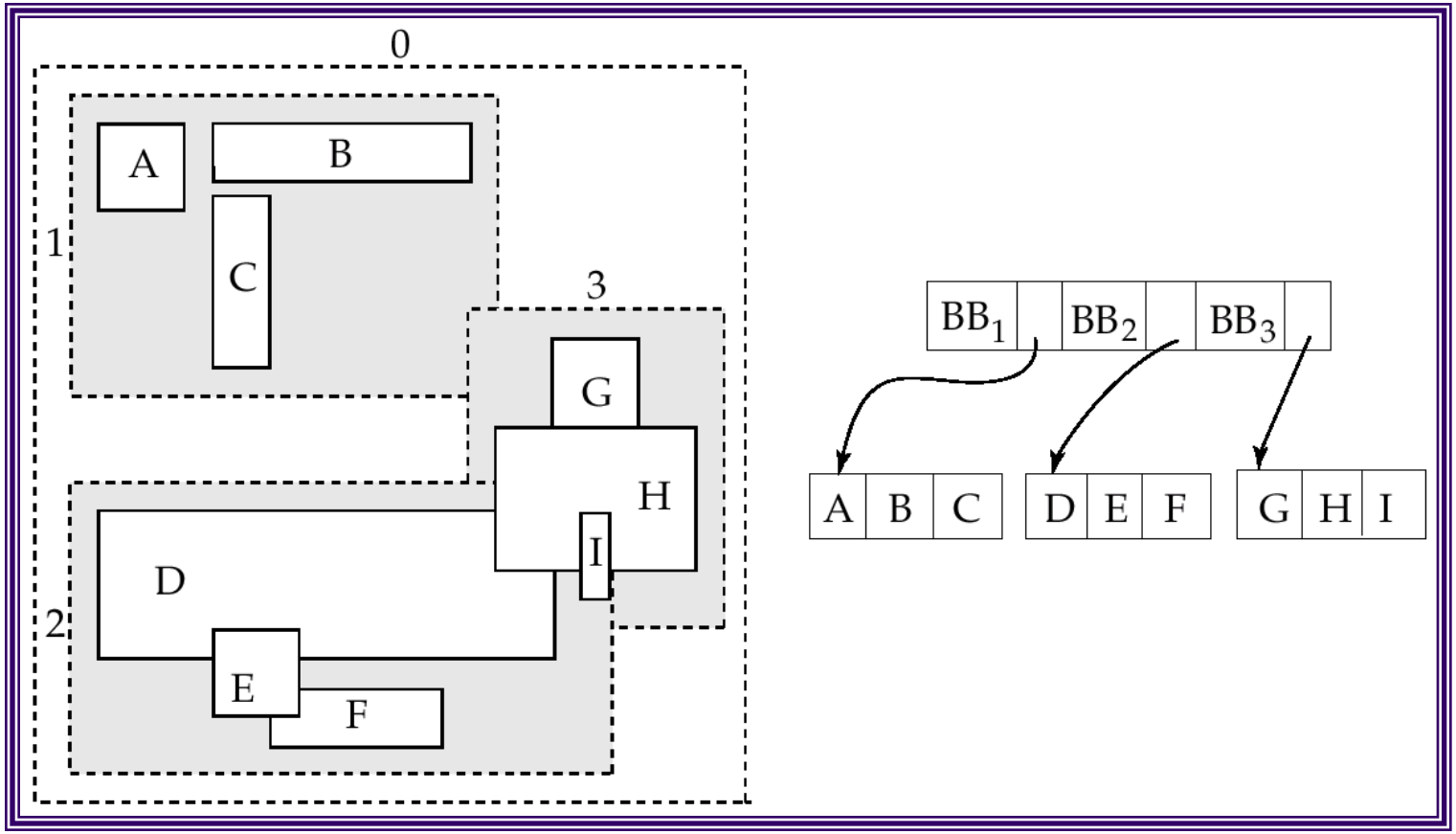
Multidimensional index structure
Can handle: *X = x1 and Y = y1*
a < X < b and c < Y < d

Stores pointers to tuples with :
branch-name between Mianus and Perryridge
and balance < 1k

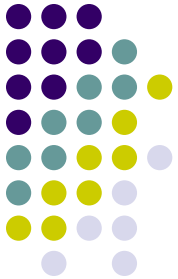


R-Trees

For spatial data (e.g. maps, rectangles, GPS data etc)



Conclusions



- Indexing Goal: “Quickly find the tuples that match certain conditions”
- Equality and range queries most common
  - Hence B+-Trees the predominant structure for on-disk representation
  - Hashing is used more commonly for in-memory operations
- Many many more types of indexing structures exist
  - For different types of data
  - For different types of queries
    - E.g. “nearest-neighbor” queries