KOREA UNIVERSITY
DATAX LAB

Chapter 14: Indexing (2/2)

(+ Ch. 24)

Outline

- Basic Concepts
- Ordered Indices
- B^+-Tree Index

- Hashing
- Write-optimized indices
- Spatio-Temporal Indexing

Chapter 24.

Hashing

Static Hashing

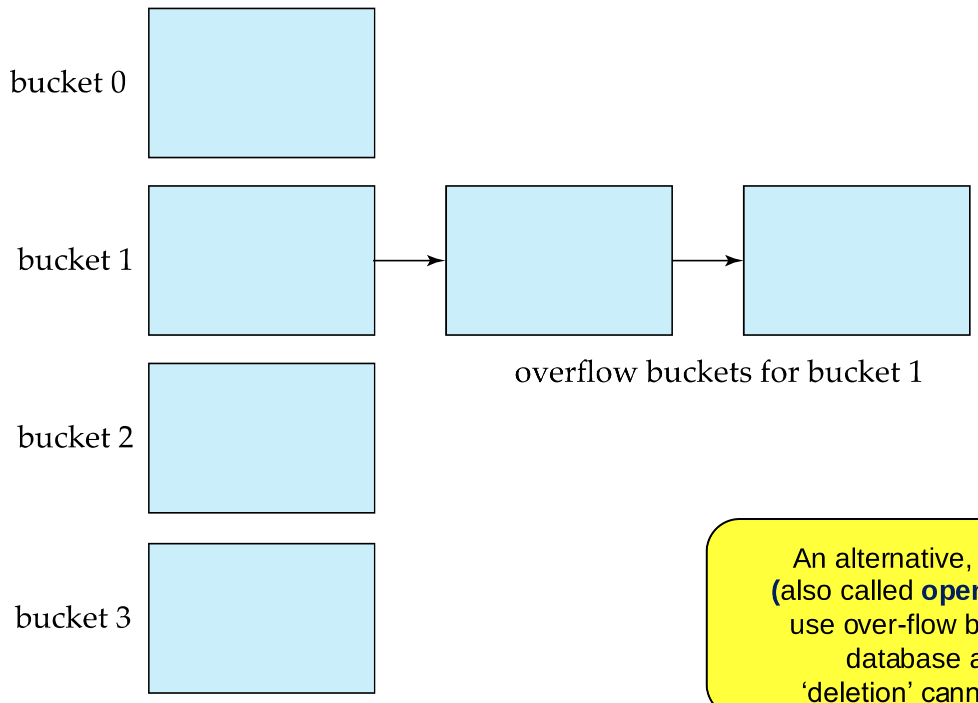
- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - We obtain the bucket of an entry from its search-key value using a **hash function**.
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records because of the following two reasons:
    1. chosen hash function produces non-uniform distribution of key values
    2. multiple records have same search-key value
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets****.*

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing (**also called **closed hashing)**



An alternative, called **open addressing (**also called **open hashing**) which does not use over-flow buckets, is not suitable for database applications because 'deletion' cannot be handled efficiently.

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key

- There are 8 buckets,
- The i th character is computed as integer *i.*
  - *a → 1, …, z →26*
- The hash function returns the sum of the converted values of the characters modulo 8
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key.

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

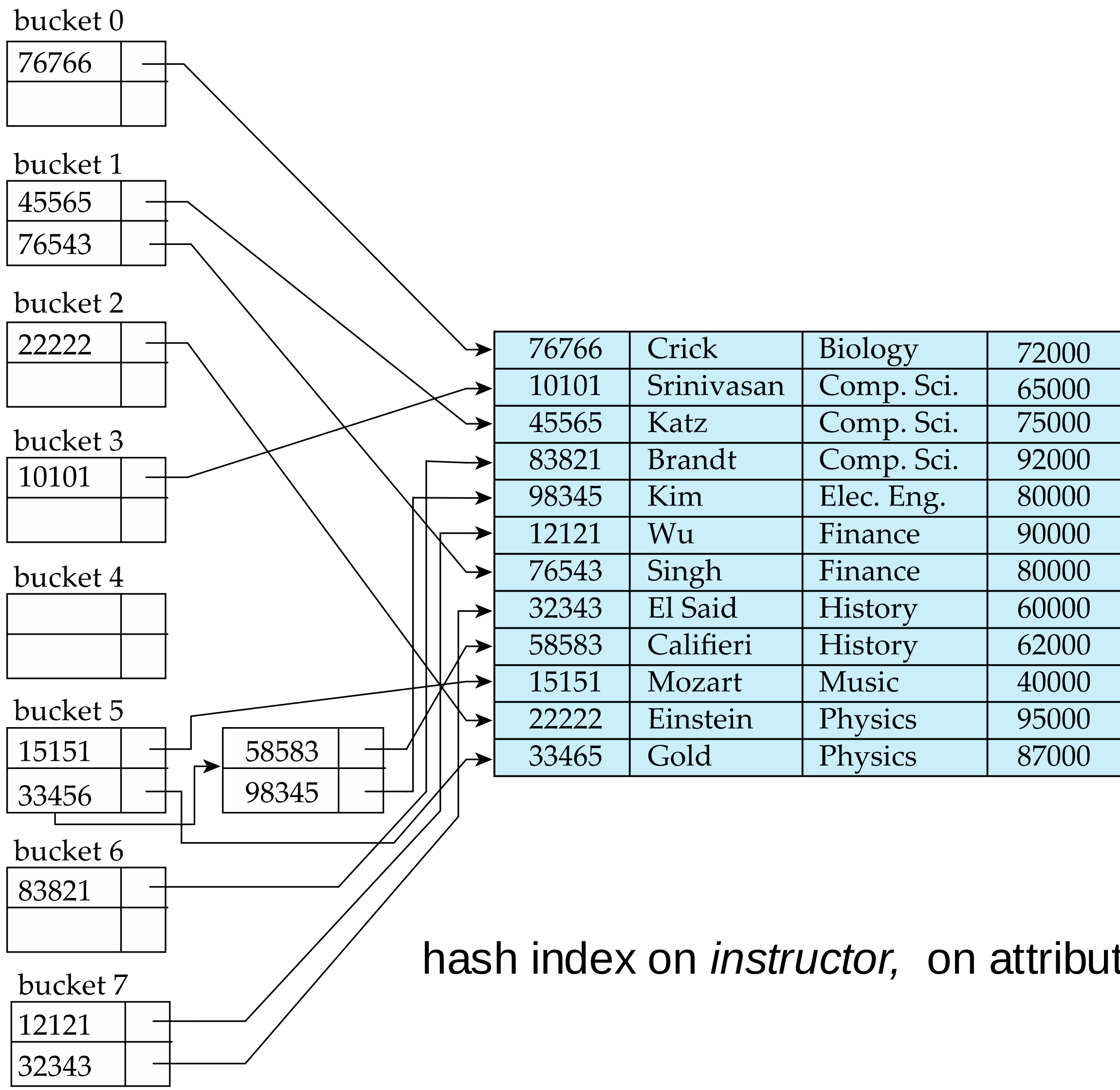
76766	Crick	Biology	72000

bucket 6

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

bucket 7

Example of Hash Index



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*

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to new table
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- **(See Ch. 24)**

- Linear Hashing
  - Do rehashing in an incremental manner

- Extendable Hashing
  - Tailored to disk-based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets

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

- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Multiple (Single-Attribute) Indices

- Use multiple indices for certain types of queries.
- Example:

  select *ID*

  from *instructor*

  where *dept_name* = "Finance" **and** *salary* = 80000
- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = "Finance".
  3. 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.

Indices on Multiple Search 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 $a_2 < b_2$

Indices on Multiple Search Keys (cont'd)

Suppose we have an index on combined search-key
(*dept_name, salary*).

- With the **where** clause
  where *dept_name* = "Finance" **and** *salary* = 80000
  the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.
  - Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
  where *dept_name* < "Finance" **and** *salary* = 80000
  - May fetch many records that satisfy the first but not the second condition

Other Features

- **Covering indices**
  - Add extra attributes to index so (some) queries can be answered without fetching the actual records

    (e.g.) index on ‘name’ (+ ‘salary’ values on the leaf index entries)

  - Store extra attributes only at leaf
    - Why?
  - Particularly useful for secondary indices
    - Why?

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

  E.g.,: **create index** *b-index* **on** *branch(branch_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported

- Drop an index

  drop index <index-name>

Creation of Indices

- Example

  create index *instructor_dept_index* **on** *instructor* (*dept_name*) ;
  drop index *instructor_dept_index* ;

- Most database systems allow specification of type of index (and clustering).
- Indices on primary key created automatically by most databases
  - Why?
- Some database also create indices on foreign key attributes automatically
  - Why might such an index be useful for this query ?
    - $takes \bowtie \sigma_{name='Shankar'} (student)$
- Indices can greatly speed up lookups, but impose cost on updates
  - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload

Write Optimized Indices

- Performance of B^+-trees can be poor for write-intensive workloads
  - One I/O per leaf, although assuming all internal nodes are in memory
    - With magnetic disks, < 100 inserts per second per disk
    - With flash memory, one page overwrite per insert (+ page erase ?)

- Two approaches to reducing cost of writes
  - **Log-structured merge tree**
  - **Buffer tree**

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n*, it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form, a bitmap index on an attribute has a bitmap for each value (possibly including the *null*) of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

- Example

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g., 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster
- Bitmap indices generally very small compared with relation size

Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a bit-wise ***and*** (a basic CPU instruction) computes ***and*** of 32 or 64 bits at once
  - E.g., 1-million-bit maps can be ***and***-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 (=2^8) elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost

{0,1,1,2,1, …, 6, 7}

  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B^+-trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
    - *64* bits **N* records * *selectivity* vs. *N* bits
  - Above technique merges benefits of bitmap and B^+-tree indices

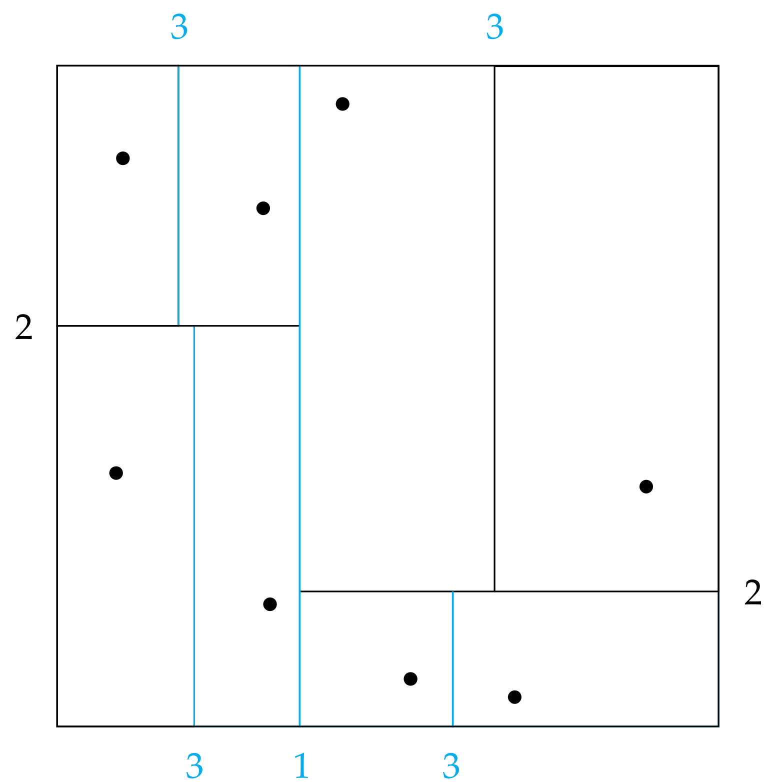
Spatial and Temporal Indices

Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
  - Allows relational databases to store and retrieve spatial information.
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - Queries can mix spatial and nonspatial conditions.
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.

K-d trees

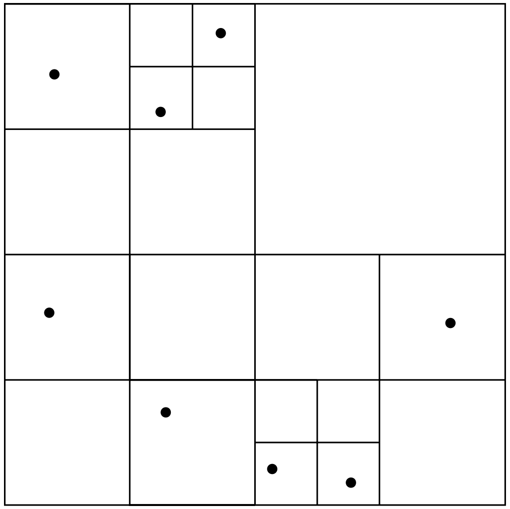
- **k-d tree** - early structure used for indexing spatial points in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - Choose one dimension for partitioning at the root level of the tree.
  - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.



- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in the below example).

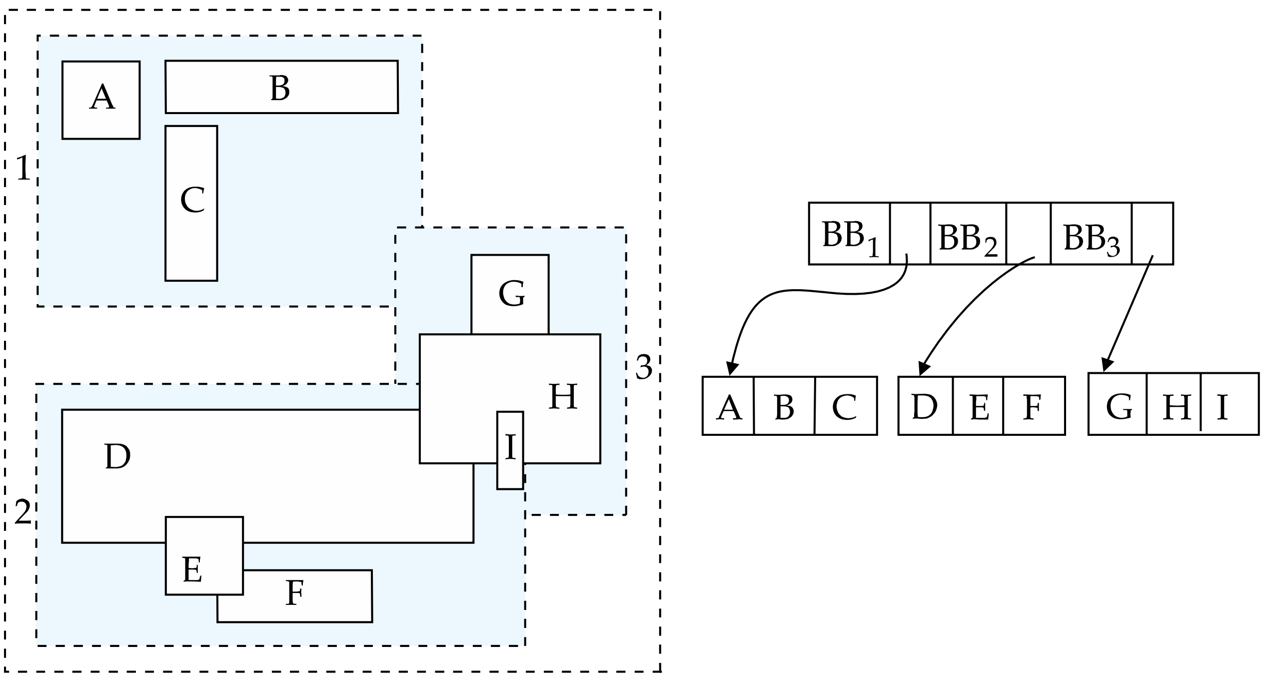


R-Trees

- **R-trees** are a N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R^+ -trees and R*-trees.
- Basic idea:
  - Generalize the notion of a one-dimensional interval associated with each B^+-tree node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

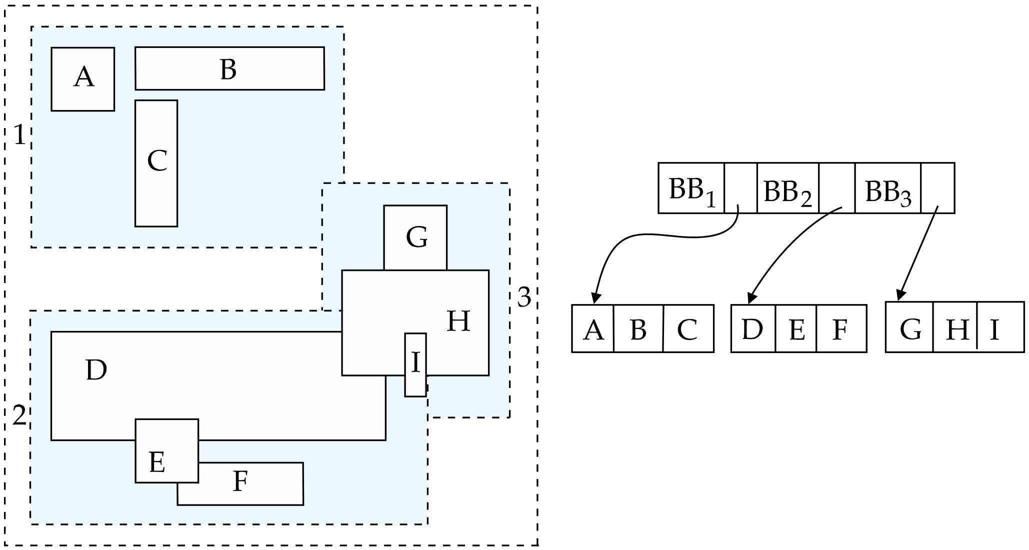
Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.



Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.



Indexing Temporal Data

- Temporal data refers to data that has an associated time period (interval)
  - Example: a temporal version of the *course* relation

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

- Time interval has a start and end time
  - End time set to infinity (or large date such as 9999-12-31) if a tuple is currently valid and its validity end time is not currently known
- Query may ask for all tuples that are valid at a point in time or during a time interval
  - Index on valid time period speeds up this task

Indexing Temporal Data (Cont.)

- How to retrieve records with attribute *a* is *v* and valid at time *t*
  1. Use index created on *a*, then filter records using time values
  2. Use spatial index, such as R-tree, with attribute *a* as one dimension, and time as another dimension
     - Valid time forms an *interval* in the time dimension
     - Tuples that are currently valid cause problems, since value is infinite or very large (→ not good for spatial indices)
       - Solution → Store all current tuples (with end time as infinity) in a separate index, indexed on (*a, start-time*) using B^+-tree
       - To find tuples valid at a point in time *t*, search for tuples in the range (a = *v, start-time* <= *t*) in the *current tuple index* (e.g. B^+-tree) + search for tuples in the range (a = *v, 0 start-time* <= *t* <= *end-time*) in the spatial index (e.g. R-tree)
- Temporal index on primary key can help enforce temporal primary key constraint (i.e., non-overlapping time intervals with the same PK).

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

End of Chapter 14 (+Ch. 24)