Spatial Data Management

Contents

DEFINITION

- A spatial database is a database that is optimized to store and query data related to objects in space, including points, lines and polygons.
- Covers multidimensional points, lines, rectangles, polygons, cubes and other geometric objects.
- While typical databases can understand various numeric and character types of data, additional functionality needs to be added for databases to process spatial data types. These are typically called geometry or feature.
- Spatial Extent: region occupied by spatial object (characterized by location and boundary).

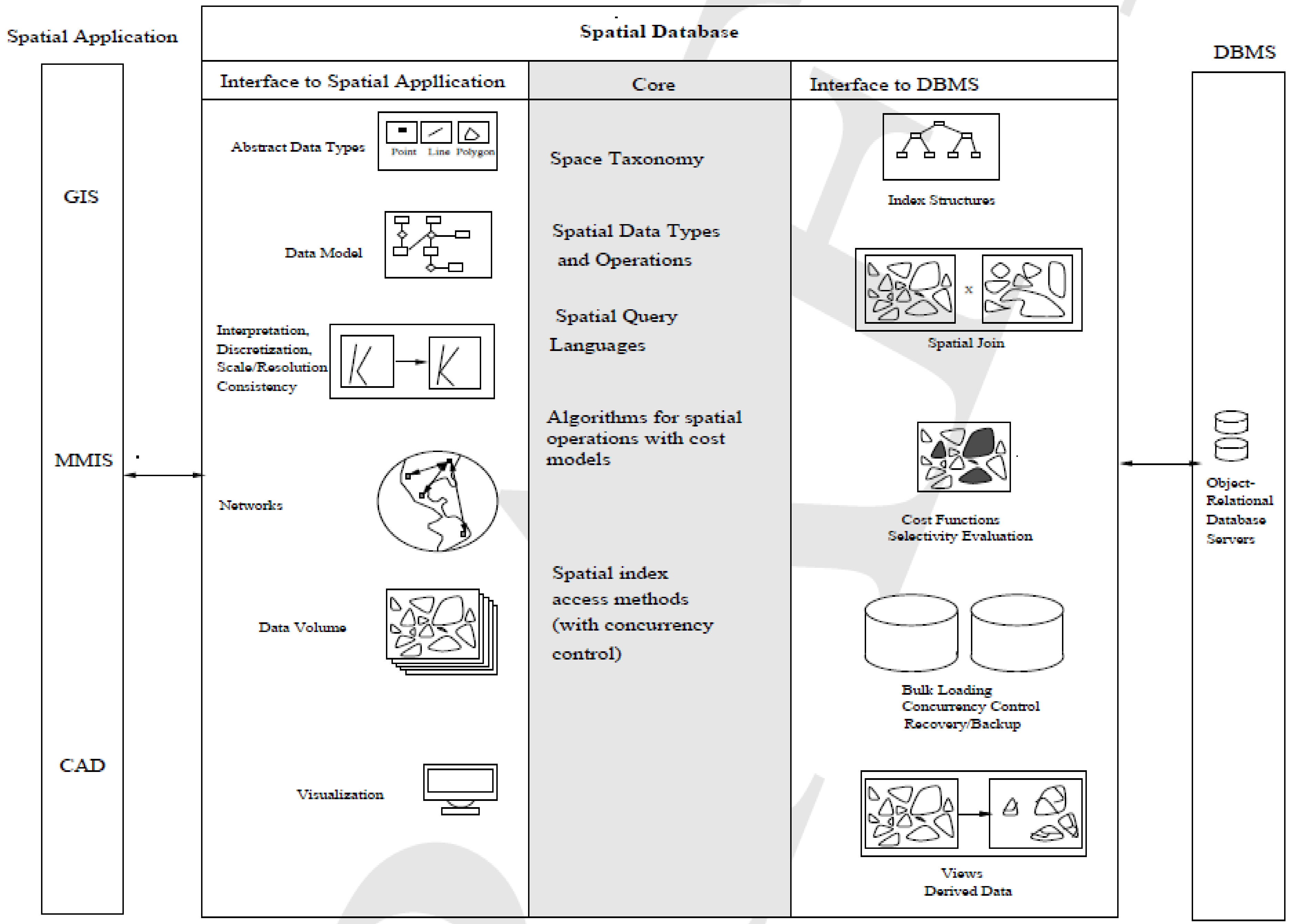
- Spatial databases provide structures for storage and analysis of spatial data
- Storing spatial data in a standard database would require excessive amounts of space
- Queries to retrieve and analyze spatial data from a standard database would be long and cumbersome leaving a lot of room for error.
- Spatial databases provide much more efficient storage, retrieval, and analysis of spatial data.

What is a SDBMS ?

- A SDBMS is a software module that
  - can work with an underlying DBMS
  - supports spatial data models, spatial abstract data types (ADTs) and a query language from which these ADTs are callable.
  - supports spatial indexing, efficient algorithms for processing spatial operations, and domain specific rules for query optimization.
- Example: Oracle Spatial data cartridge, ESRI SDE
  - can work with Oracle 8i DBMS
  - Has spatial data types (e.g. polygon), operations (e.g. overlap) callable from SQL3 query language
  - Has spatial indices, e.g. R-trees

SDBMS Three-layer Structure

- SDBMS works with a spatial application at the front end and a DBMS at the back end
- SDBMS has three layers:
  - Interface to spatial application
  - Core spatial functionality
  - Interface to DBMS


Spatial Application
GIS
MMIS
CAD
Spatial Database
Interface to Spatial Appllication
Abstract Data Types
Point
Line
Polygon
Data Model
Interpretation, Discretization, Scale/Resolution Consistency
Networks
Data Volume
Visualization
Core
Space Taxonomy
Spatial Data Types and Operations
Spatial Query Languages
Algorithms for spatial operations with cost models
Spatial index access methods (with concurrency control)
Interface to DBMS
Index Structures
Spatial Join
Cost Functions Selectivity Evaluation
Bulk Loading Concurrency Control Recovery/Backup
Views Derived Data
DBMS
Object-Relational Database Servers

Applications of Spatial Data

- Geographic Information Systems (GIS)
  - E.g., ESRI's ArcInfo; OpenGIS Consortium
  - Geospatial information
  - All classes of spatial queries and data are common
- Computer-Aided Design/Manufacturing
  - Store spatial objects such as surface of airplane fuselage
  - Range queries and spatial join queries are common
- Multimedia Databases
  - Images, video, text, etc. stored and retrieved by content
  - First converted to *feature vector* form; high dimensionality
  - Nearest-neighbor queries are the most common

Types of Spatial Data

- Point Data
  - Points in a multidimensional space
  - E.g., *Raster data* such as satellite imagery, where each pixel stores a measured value
  - E.g., Feature vectors extracted from text.

- Region Data
  - Consists of a collection of regions
  - Objects have spatial extent with location and boundary.
  - DB typically uses geometric approximations constructed using line segments, polygons, etc., called *vector data*.

Types of Spatial Queries

- Spatial Range Queries
  - *Find all cities within 50 miles of Madison*
  - Query has associated region (location, boundary)
  - Answer includes ovelapping or contained data regions
- Nearest-Neighbor Queries
  - *Find the 10 cities nearest to Madison*
  - Results must be ordered by proximity
- Spatial Join Queries
  - *"Find pairs of cities within 200 miles of each other" and "Find all cities near a lake"*
  - Expensive, join condition involves regions and proximity

Types of Data Stored in Spatial Databases

- Two-dimensional data examples
  - Geographical
  - Cartesian coordinates (2-D)
  - Networks
  - Direction

- Three-dimensional data examples
  - Weather
  - Cartesian coordinates (3-D)
  - Topological
  - Satellite images

Introduction to indexes

- ❖ Single level index
    - Primary index
    - Clustering index
    - Secondary index
- ❖ Multilevel index

Primary index on the ordering key field

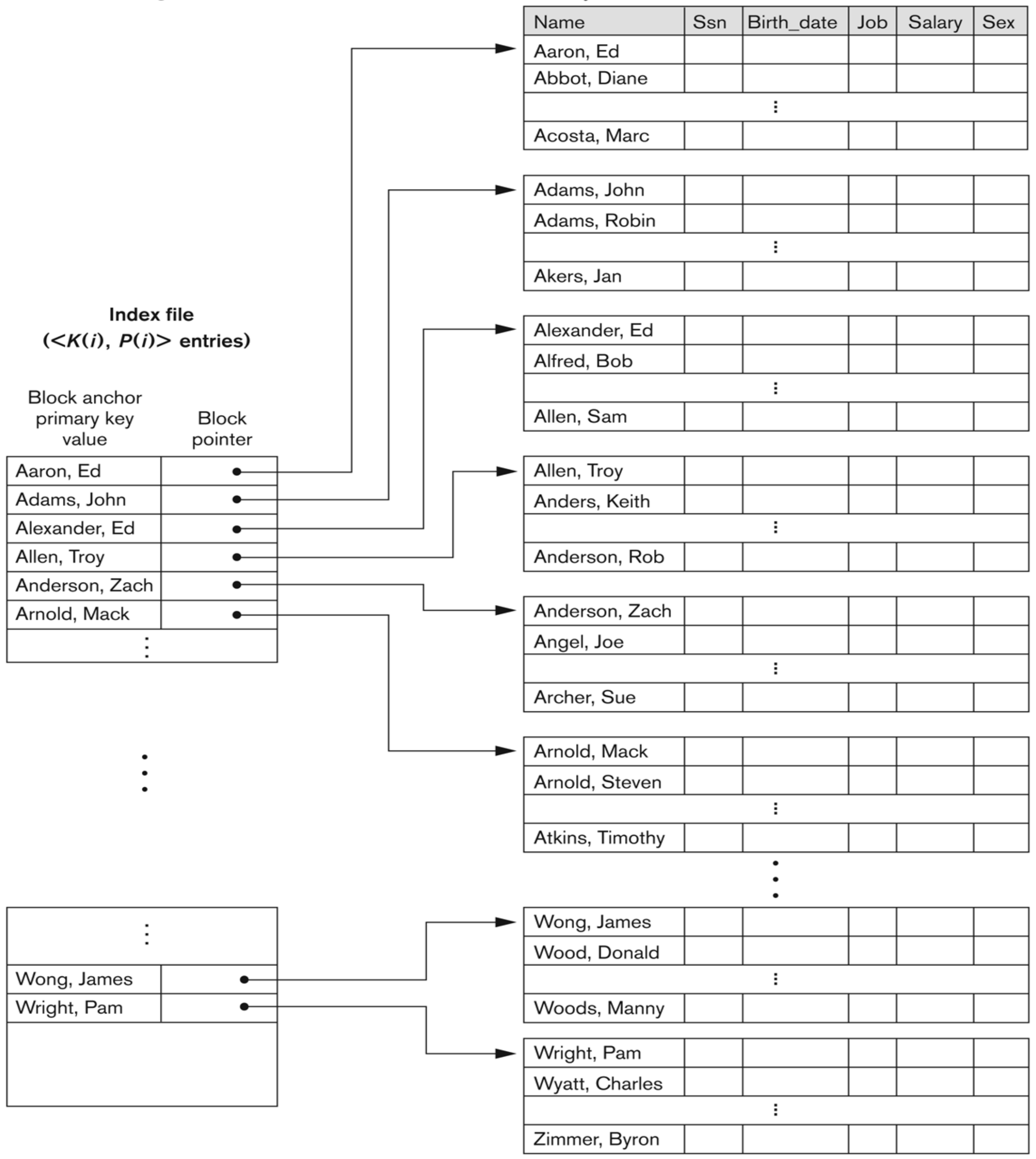
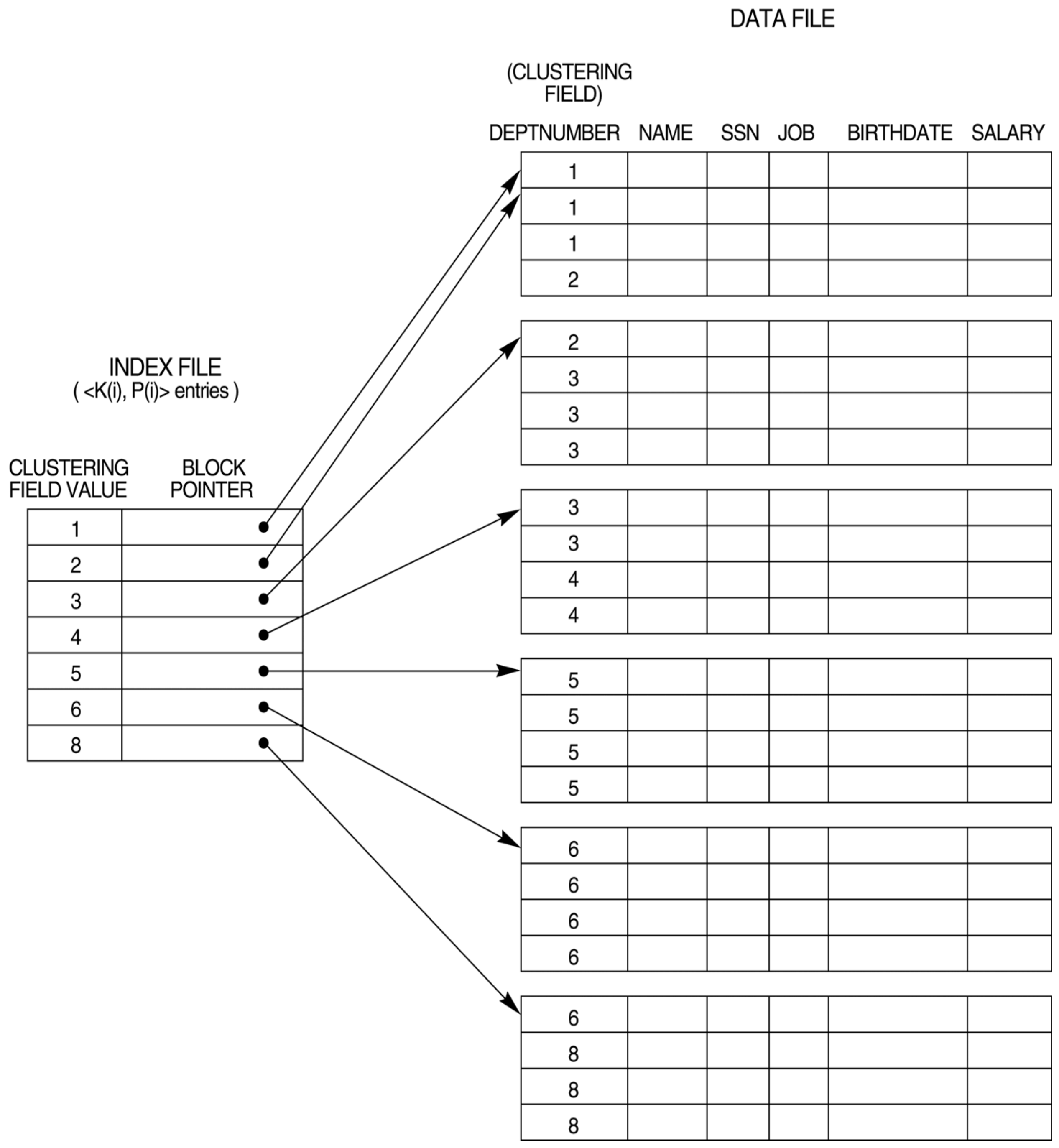


Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.

A Clustering Index

- A clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file.



A Two-level Primary Index

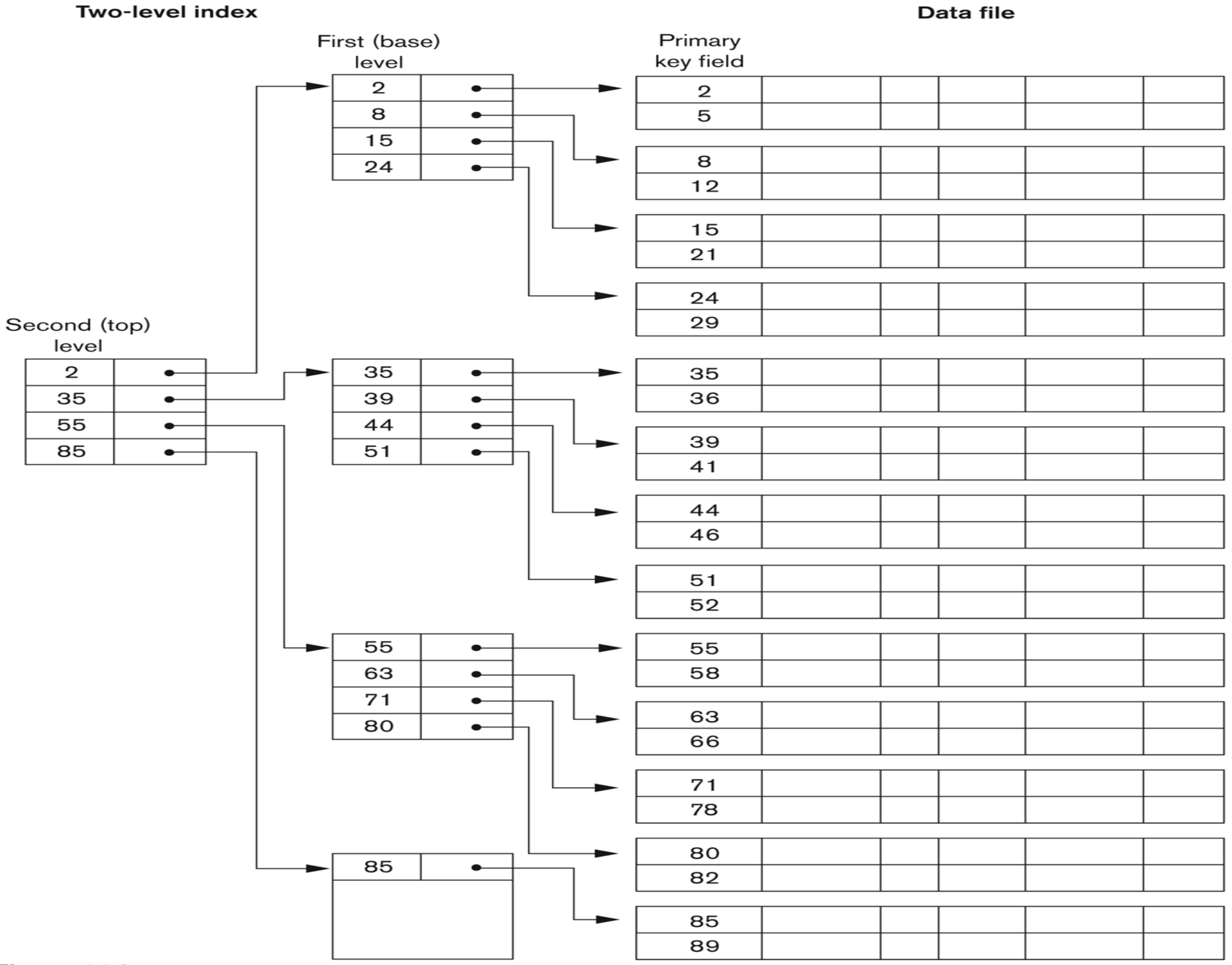


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

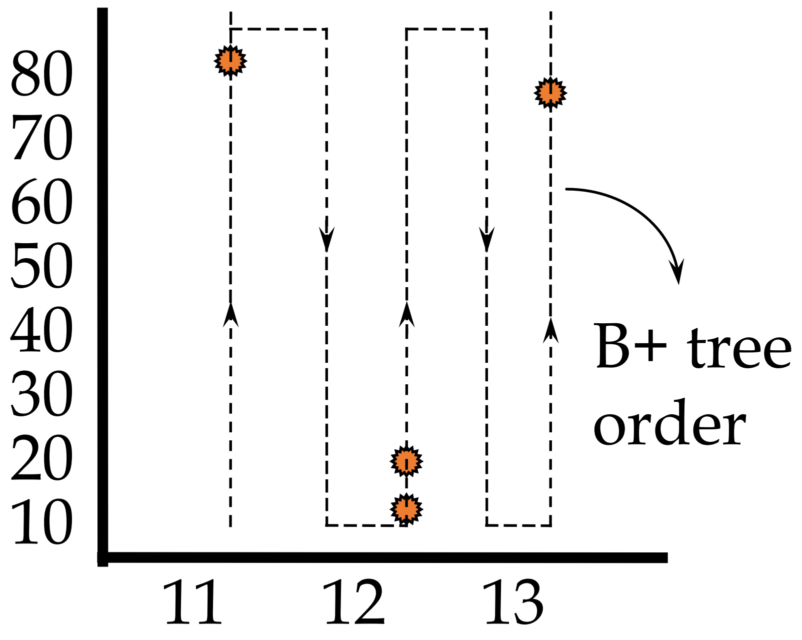
Spatial Indexes

- Multidimensional index
- Main purpose is to support spatial selection.

Single-Dimensional Indexes

- B+ trees are fundamentally single-dimensional indexes.
- When we create a composite search key B+ tree, e.g., an index on <age, sal>, we effectively linearize the 2-dimensional space since we sort entries first by age and then by sal.

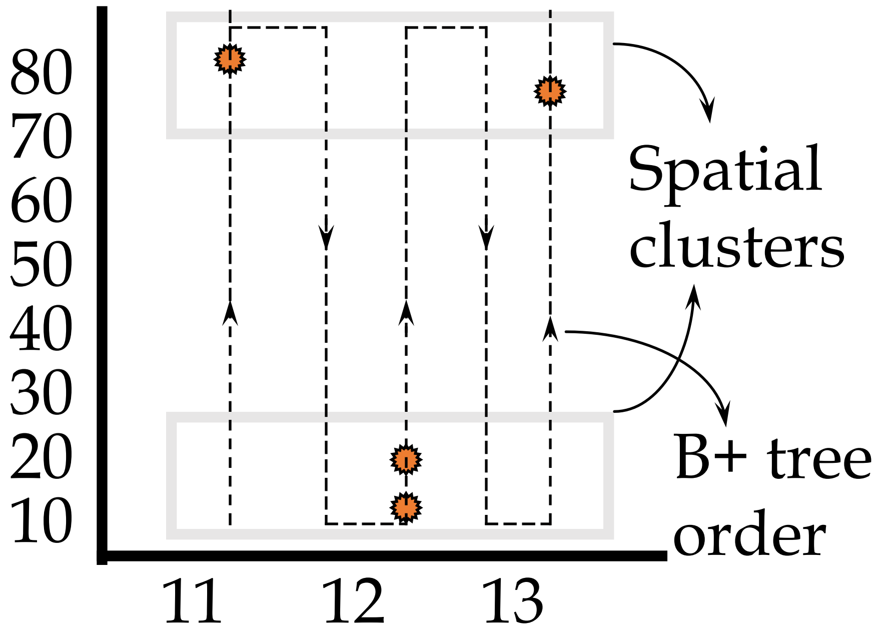
Consider entries:
<11, 80>, <12, 10>
<12, 20>, <13, 75>



Multidimensional Indexes

- A multidimensional index clusters entries so as to exploit “nearness” in multidimensional space.
- Keeping track of entries and maintaining a balanced index structure presents a challenge!

Consider entries:
<11, 80>, <12, 10>
<12, 20>, <13, 75>



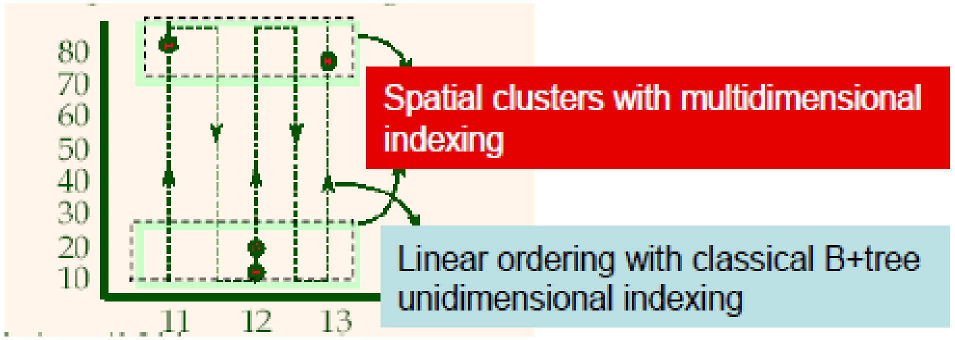
Multi-dimensional indexes

- A multidimensional index clusters entries so as to exploit “nearness” in multidimensional space.
- The basic motivation for multi-dimensional indexing is that for efficient content-based retrieval it is necessary to cluster entities so as to exploit *nearness* in multidimensional space.

.

Consider the entries:

<11, 80>, <12, 10>
<12, 20>, <13, 75>


80
70
60
50
40
30
20
10
11
12
13
Spatial clusters with multidimensional indexing
Linear ordering with classical B+tree unidimensional indexing

- Most used multidimensional index structures:
  - k-d Trees
  - Point Quadtrees
  - R, R*, R+ Trees

  ⇐ For low/high-dimensional indexes

  - SS, SR trees
  - M Trees

  ⇐ For very high-dimensional indexes

Motivation for Multidimensional Indexes

- Spatial queries (GIS, CAD).
  - Find all hotels within a radius of 5 miles from the conference venue.
  - Find the city with population 500,000 or more that is nearest to Kalamazoo, MI.
  - Find all cities that lie on the Nile in Egypt.
  - Find all parts that touch the fuselage (in a plane design).
- Similarity queries (content-based retrieval).
  - Given a face, find the five most similar faces.
- Multidimensional range queries.
  - 50 < age < 55 AND 80K < sal < 90K

What's the difficulty?

- An index based on spatial location needed.
  - One-dimensional indexes don't support multidimensional searching efficiently. (Why?)
  - Hash indexes only support point queries; want to support range queries as well.
  - Must support inserts and deletes gracefully.
- Ideally, want to support non-point data as well (e.g., lines, shapes).
- The R-tree meets these requirements, and variants are widely used today.

Index structures

- Many spatial index structures have been proposed.
- Index structures for point data.
    - Grid files
    - hB trees
    - kD trees
    - Point quad trees
    - SR trees
- Index structures that handle regions as well as point data.
    - Region quad trees
    - R trees
    - SKD trees

- Three main approaches:
  - –Index structures that rely on space filling curves to organize points.
  - –Grid files
  - –R trees

Note:

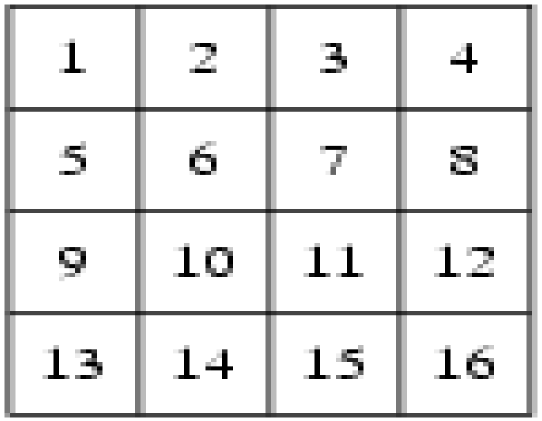
We have no proof to say that one indexing is the best. But R trees have been widely implemented and found their way into commercial DBMS.

Index structures that rely on space filling curves to organize points

- Z ordering for point data
- Z ordering for region data
- Region quad trees illustrate an indexing approach based on recursive subdivision of the multidimensional space.
- Several variants of region quad trees exists.

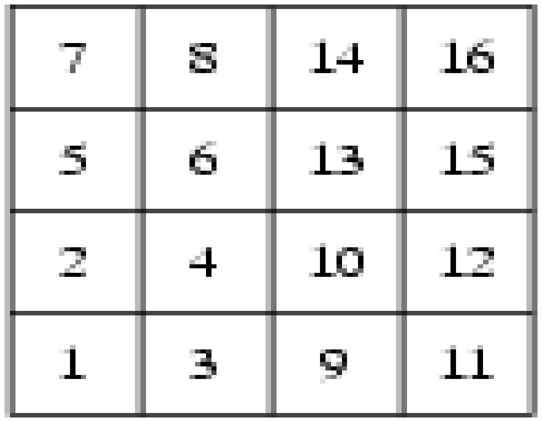
Space filling curves

- Impose an ordering on the locations in a multi-dimensional space
- Examples: row-order (Fig.(a), z-order (Fig (b))
- Allow use of traditional efficient search methods on spatial data



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a)



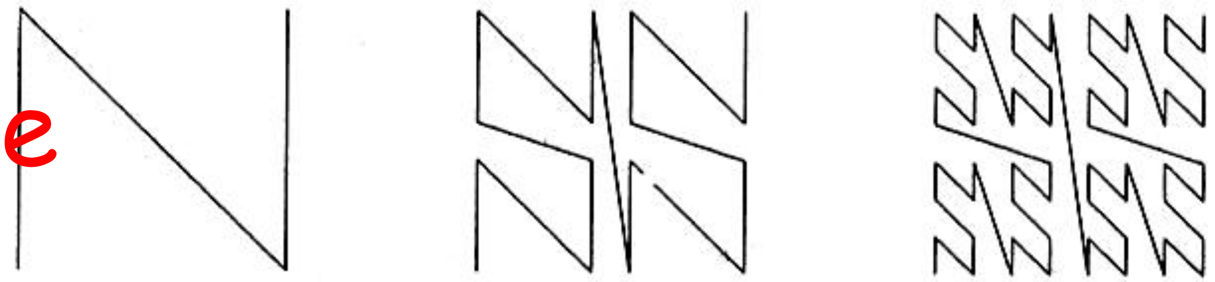
7	8	14	16
5	6	13	15
2	4	10	12
1	3	9	11

(b)

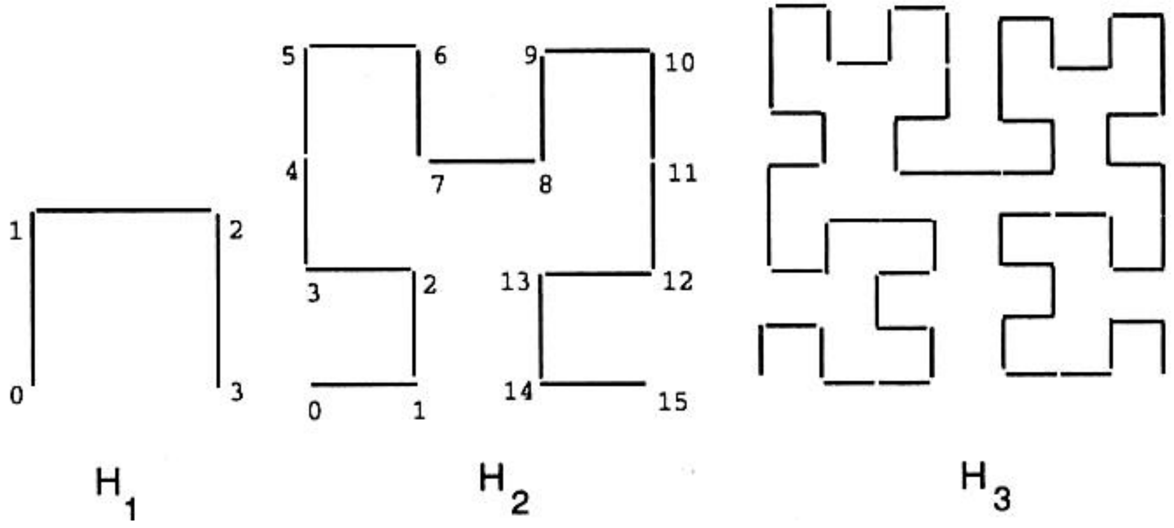
- Mapping of multi-dimensional space to one dimension
  - visit all data points in space in some order
  - this order defines an *1D* sequence of points
  - points which are close together in the multi-dimensional space must be assigned similar values in the 1D sequence
  - A B^+–tree for indexing

Two Common Curves

Z-curve



Hilbert

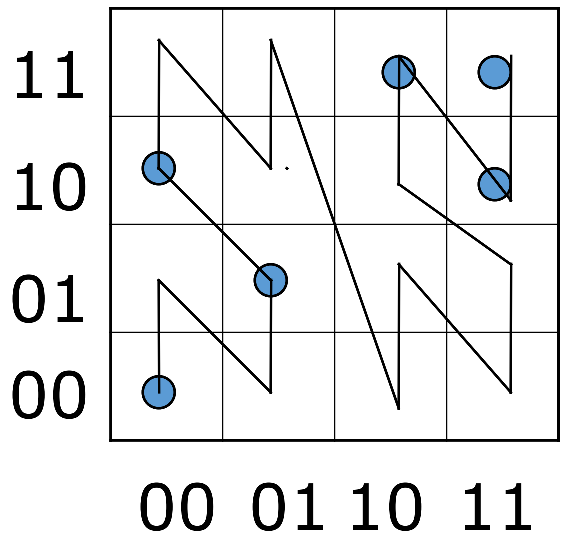


Z- ordering

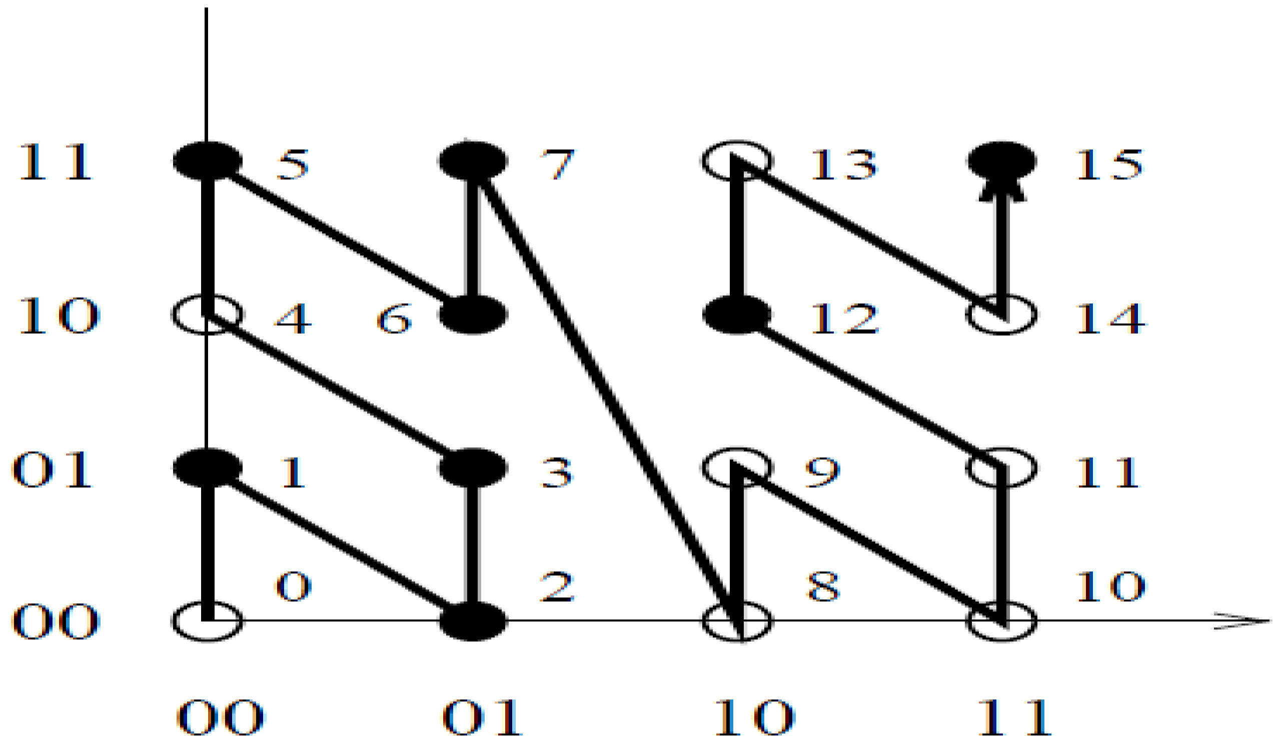
- a function which maps multidimensional data to one dimension while preserving the locality of data points.

Space-filling curve: Z-ordering

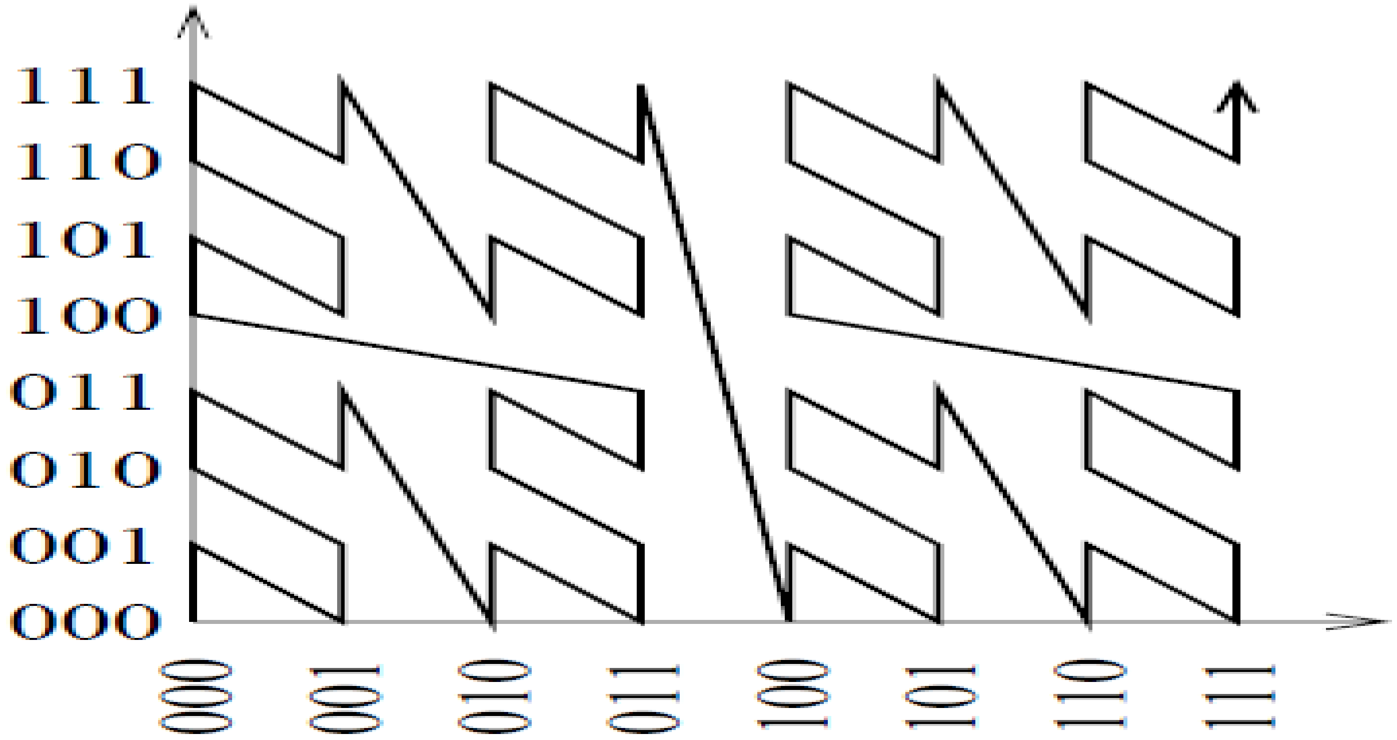
- Interleaves x-axis bits and y axis bits:
  - X=00, Y=10, Z=0100 (4)
  - X=10, Y=11, Z=1101 (13)
- Long diagonal jumps



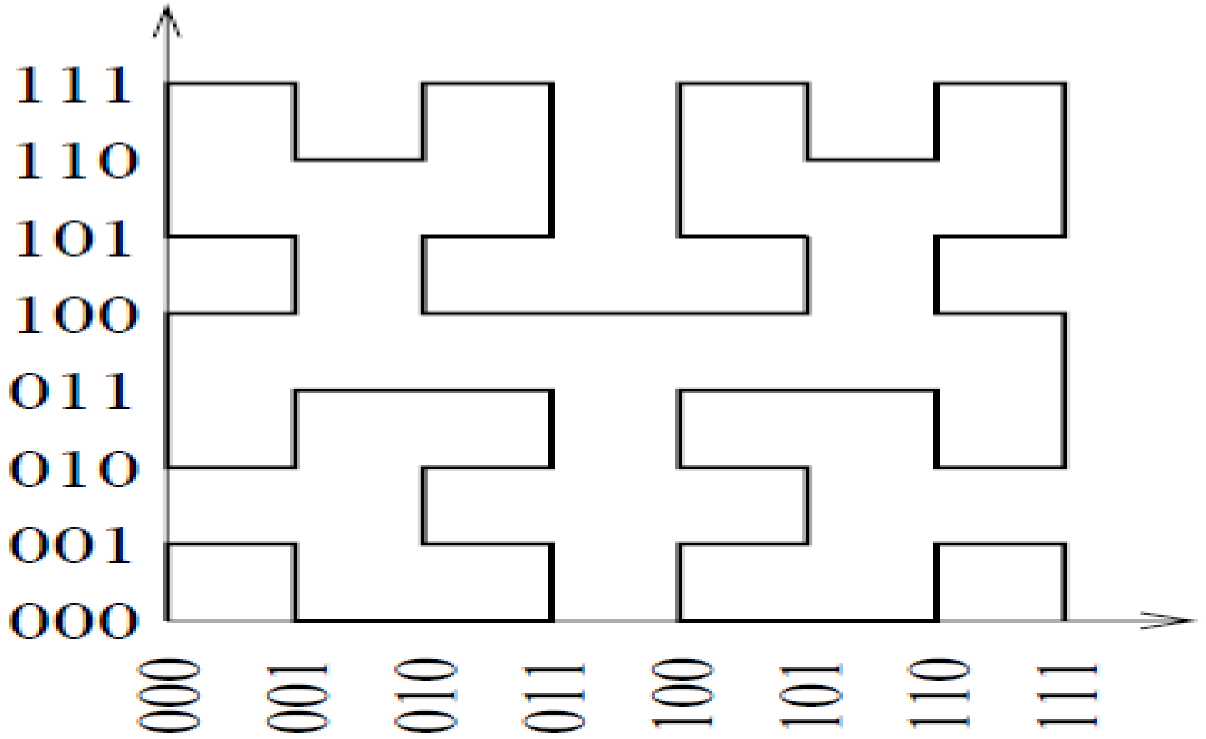
Z-ordering with two bits



Z-ordering with three bits



Hilbert curve with three bits

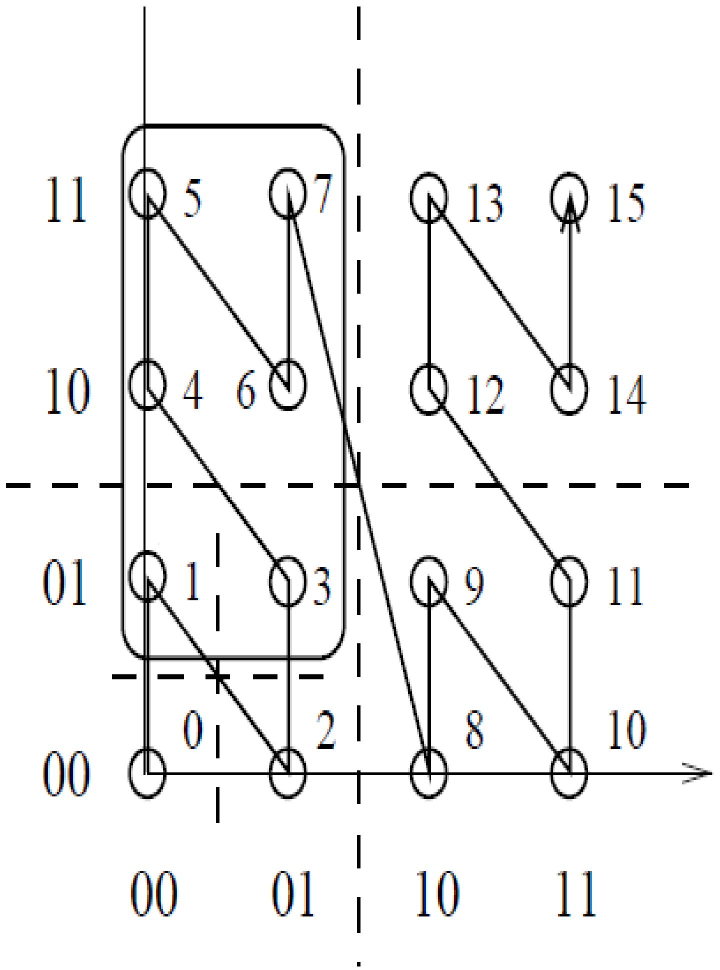


- Space-filling curves are based on the assumption that any attribute value can be represented with some fixed number of bits, say k bits.
- The maximum number of values along each dimension is therefore 2^k

- The points in a dataset are stored in Z-value order and indexed by a traditional indexing structure such as a B+ tree. That is, the Z-value of a point is stored together with the point and is the search key for the B+ tree.
- We do not have to store the X and Y values for a point if we store the Z-value, since we can compute them from the Z-value by extracting the interleaved bits.
- To insert a point, we compute its Z-value and insert it into the B+ tree. Deletion and search are similarly based on computing the Z-value and then using the standard B+ tree algorithms.

Region Quad Trees and Z-Ordering: Region Data

- The Region Quad tree structure corresponds directly to the recursive decomposition of the data space.
- Each node in the tree corresponds to a square-shaped region of
- the data space.
- The root corresponds to the entire data space, and some leaf nodes correspond to exactly one point.
- Each internal node has four children, corresponding to the four quadrants into which the space corresponding to the node is
- partitioned: 00 identifies the bottom left quadrant, 01 identifies the top left quadrant, 10 identifies the bottom right quadrant, and 11 identifies the top right quadrant.



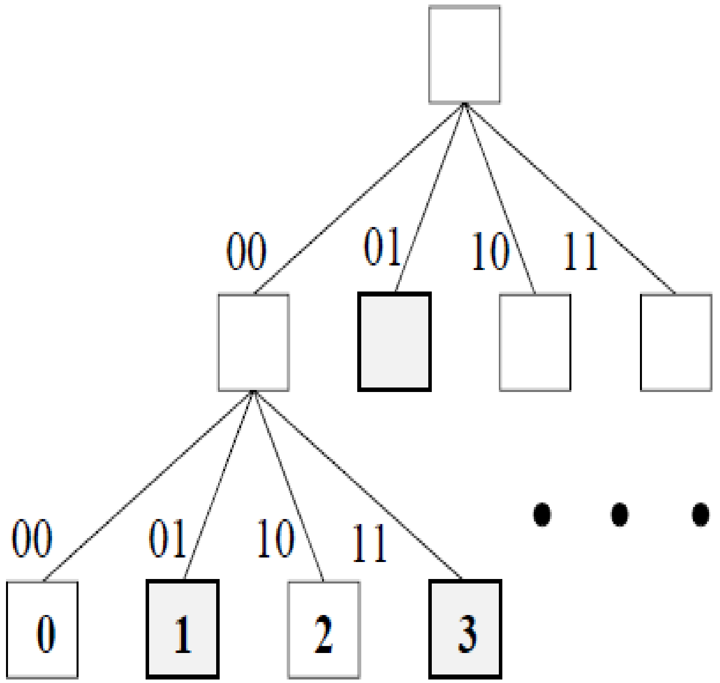


Figure 26.3 Z-Ordering and Region Quad Trees

- consider the children of the root. All points in the quadrant corresponding to the 00 child have Z-values that begin with 00, all points in the quadrant corresponding to the 01 child have Z-values that begin with 01, and so on.
- The Z-value of a point can be obtained by traversing the path from the root to the leaf node for the point and concatenating all the edge labels.

GRID FILES

- The method is designed to guarantee that any point query (a query that retrieves the information associated with the query point) can be answered in at most two disk accesses.
- Grid files rely upon a grid directory to identify the data page containing a desired point.
- The grid directory is similar to the directory used in Extendible Hashing.
- When searching for a point, we first find the corresponding entry in the grid directory.
- The grid directory entry, like the directory entry in Extendible Hashing, identifies the page on which the desired point is stored, if the point is in the database.

- describe the Grid file structure for two-dimensional data.
- The method can be generalized to any number of dimensions.
- The Grid file partitions space into rectangular regions using lines that are parallel to the axes. Thus, we can describe a Grid file partitioning by specifying the points at which each axis is ‘cut.’
- If the X axis is cut into i segments and the Y axis is cut into j segments, we have a total of i × j partitions.
- The grid directory is an i by j array with one entry per partition.
- This description is maintained in an array called a linear scale; there is one linear scale per axis.

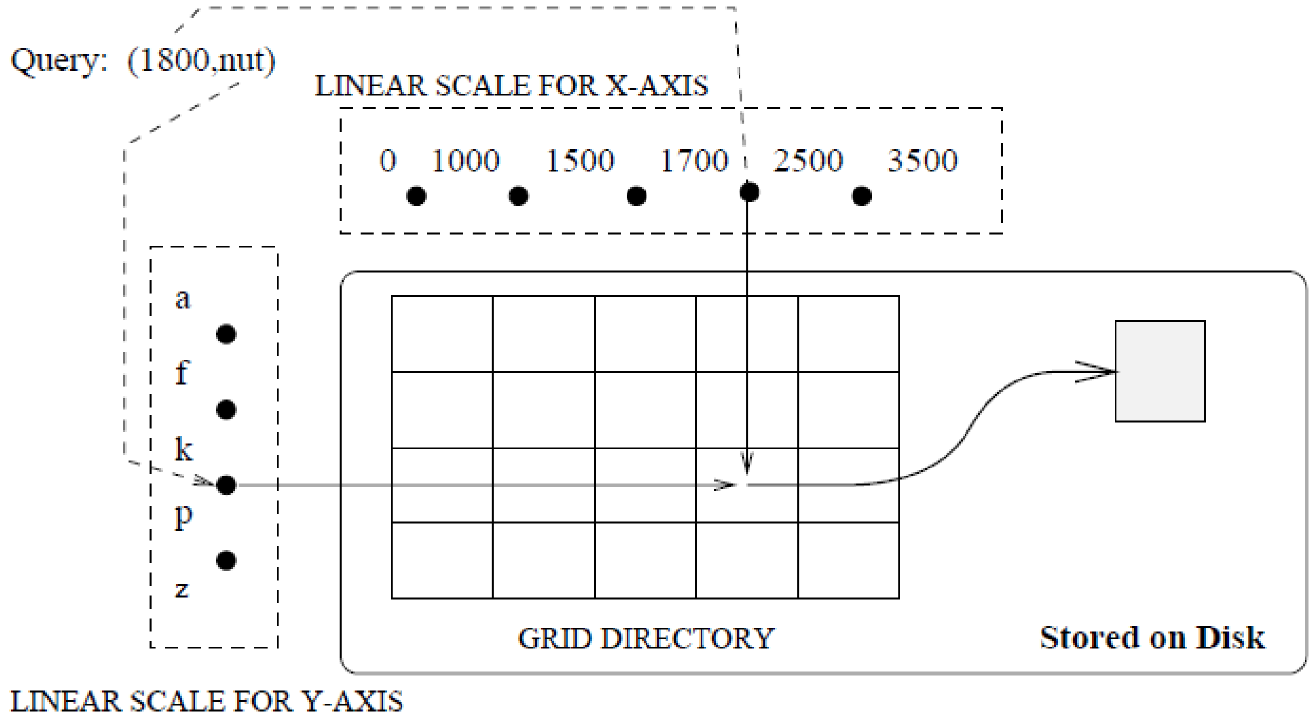


Figure 26.4 Searching for a Point in a Grid File

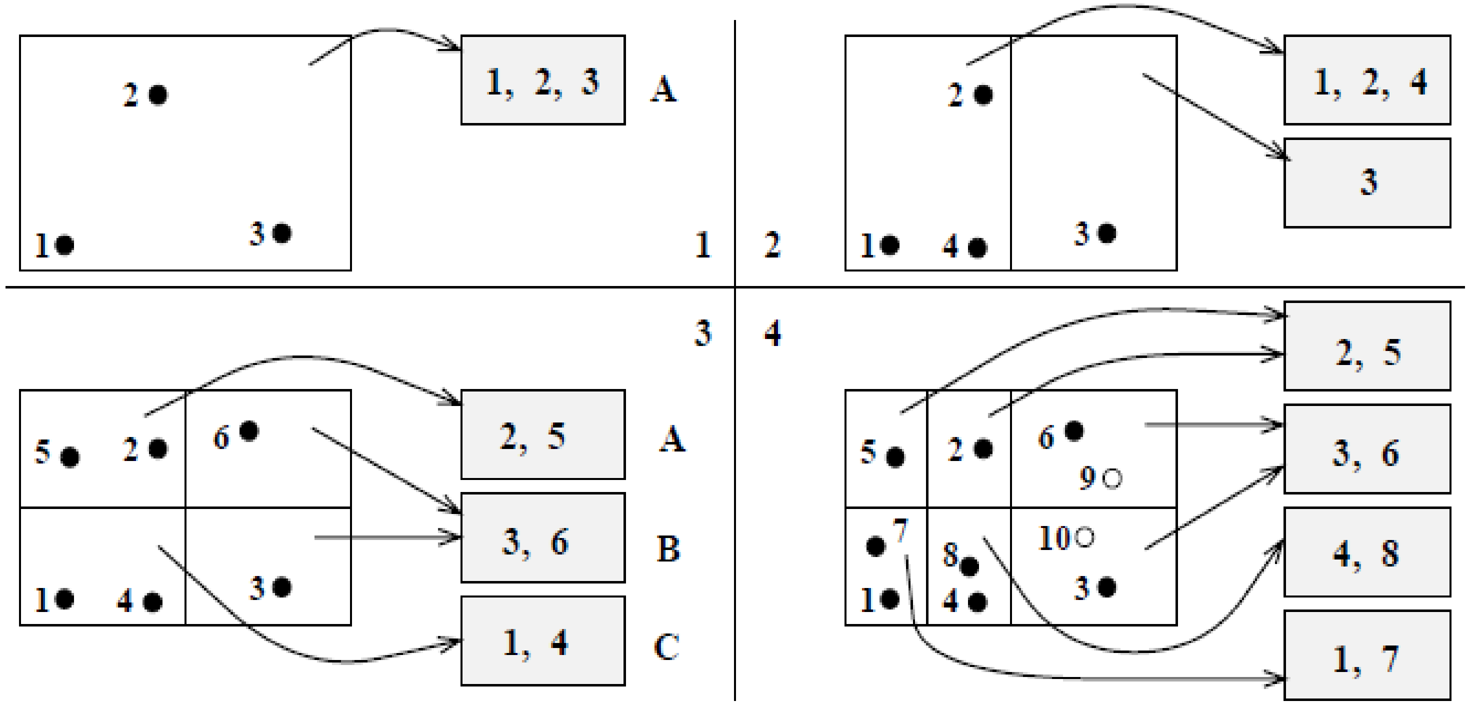
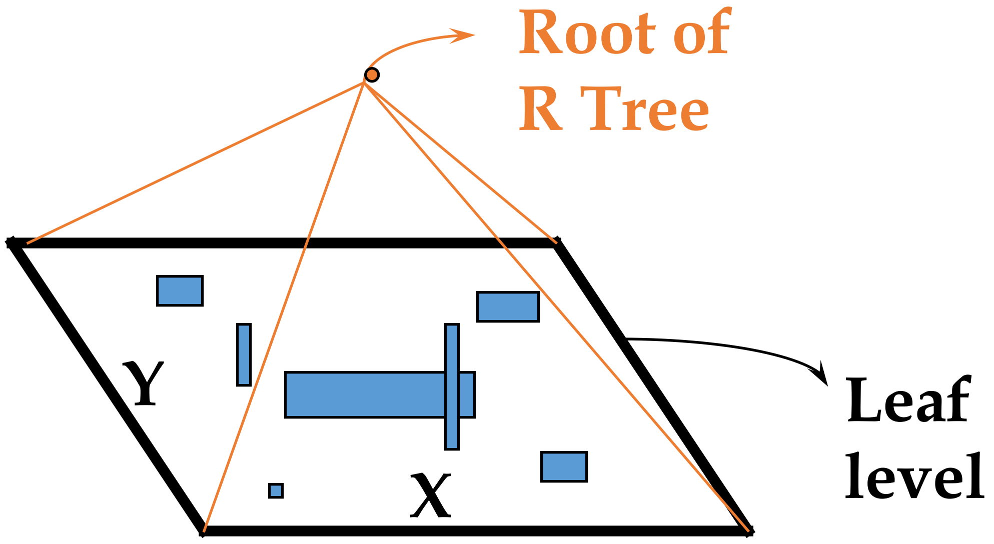


Figure 26.5 Inserting Points into a Grid File

The R-Tree

- The R-tree is a tree-structured index that remains balanced on inserts and deletes.
- Each key stored in a leaf entry is intuitively a box, or collection of intervals, with one interval per dimension.
- Example in 2-D:



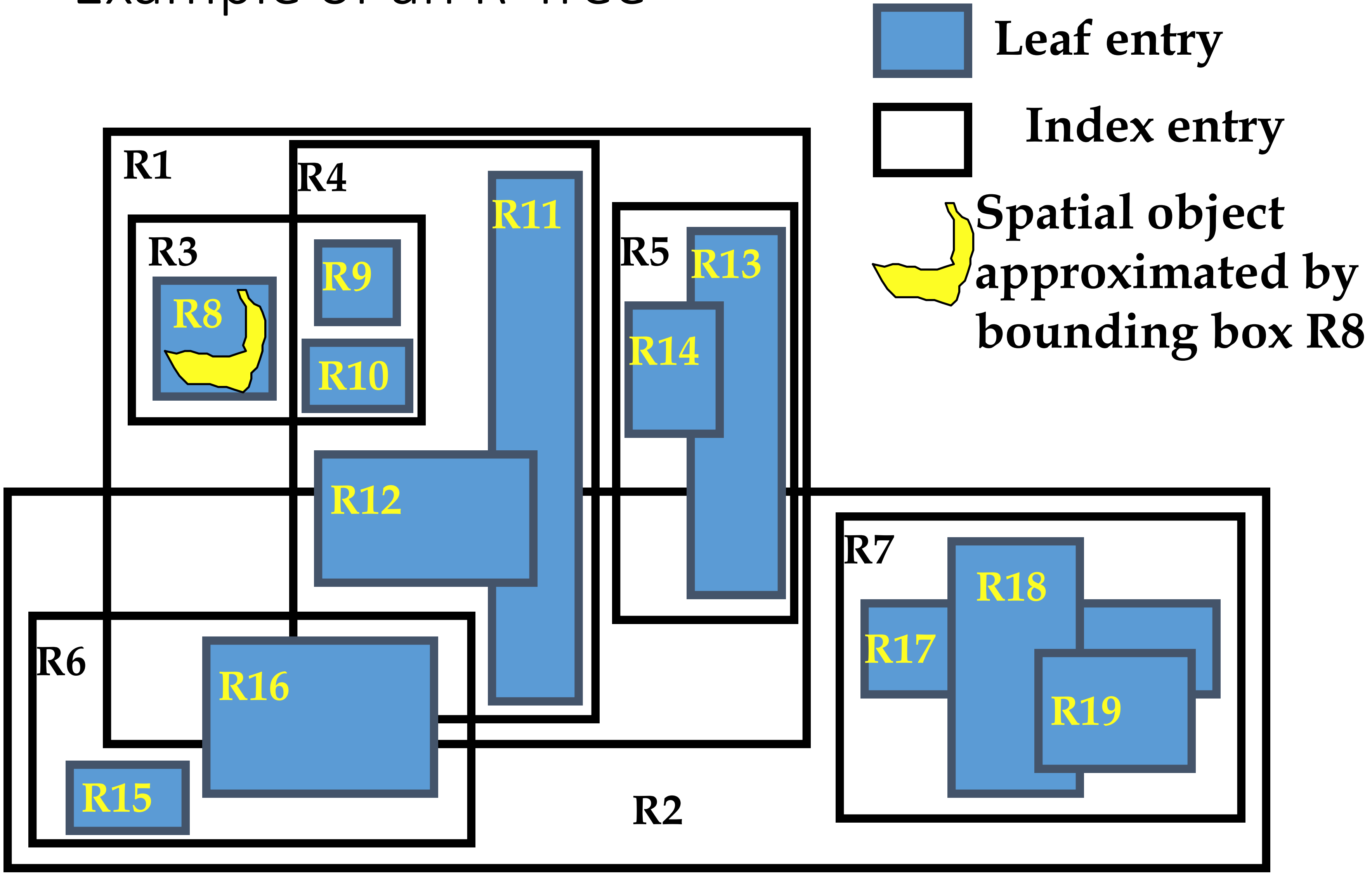
- An adaptation of the B+ tree to handle spatial data
- Height balanced tree
- The search key for an R Tree is a collection of intervals, with one interval per dimension.
- Search key value: A box that is bounded by the intervals, each side of the box is parallel to an axis. (Bounding boxes)

R-Tree Properties

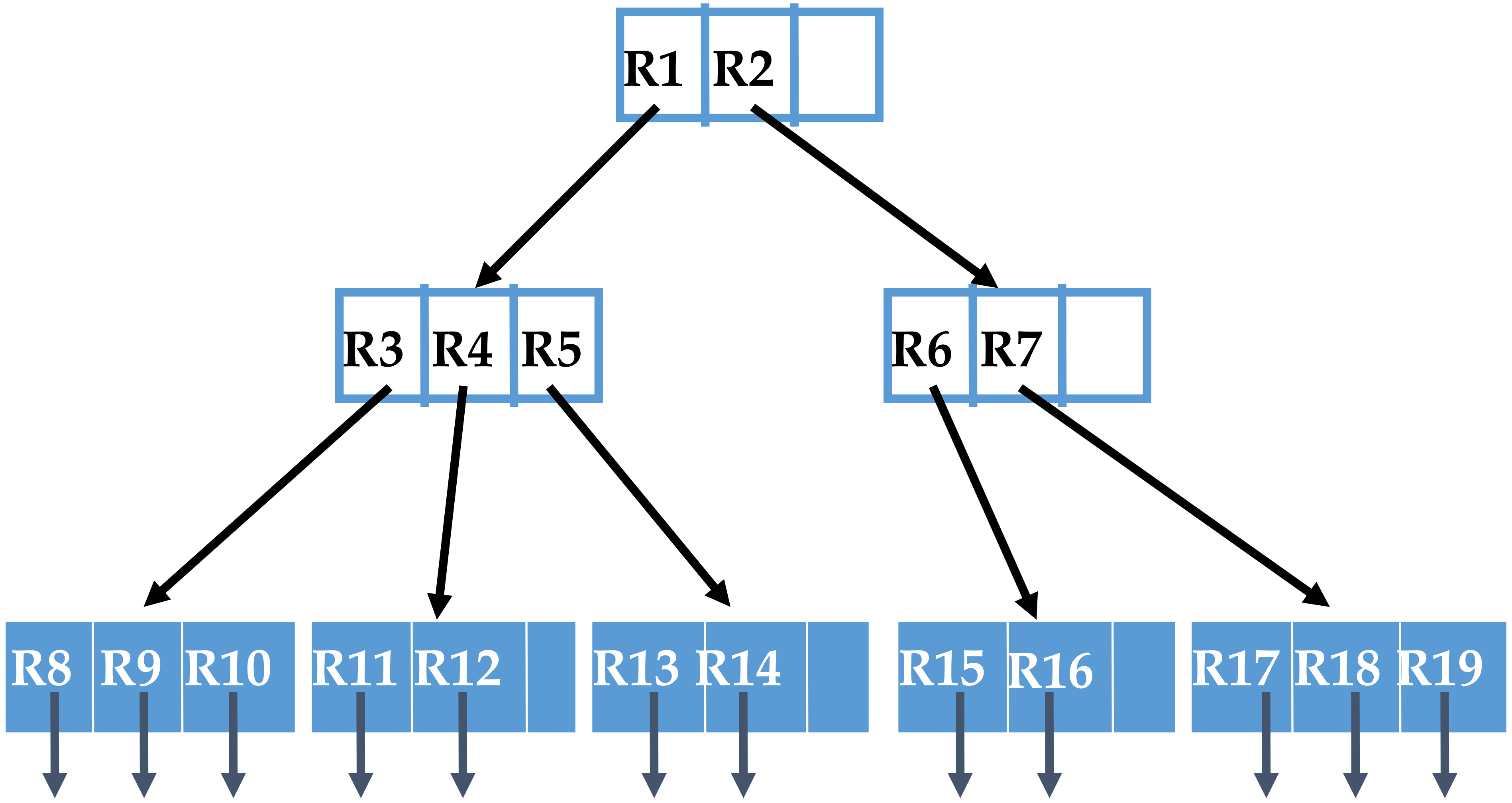
- Leaf entry = < n-dimensional box, rid >
  - This is *key value* being a box.
  - Box is the tightest bounding box for a data object.
- Non-leaf entry = < n-dim box, ptr to child node >
  - Box covers all boxes in child node (in fact, subtree).
- All leaves at same distance from root.
- Nodes can be kept 50% full (except root).
  - Can choose a parameter *m* that is <= 50%, and ensure that every node is at least *m*% full.

  - Rid: identifies an object

Example of an R-Tree



Example R-Tree (Contd.)



Search for Objects Overlapping Box Q

Start at root.

1. If current node is non-leaf, for each entry <E, ptr>, if ***box*** E overlaps Q, search subtree identified by ptr.
2. If current node is leaf, for each entry <E, rid>, if E overlaps Q, rid identifies an object that might overlap Q.

te: May have to search ***several*** *subtrees at each node!*
contrast, a B-tree equality search goes to just one leaf.)

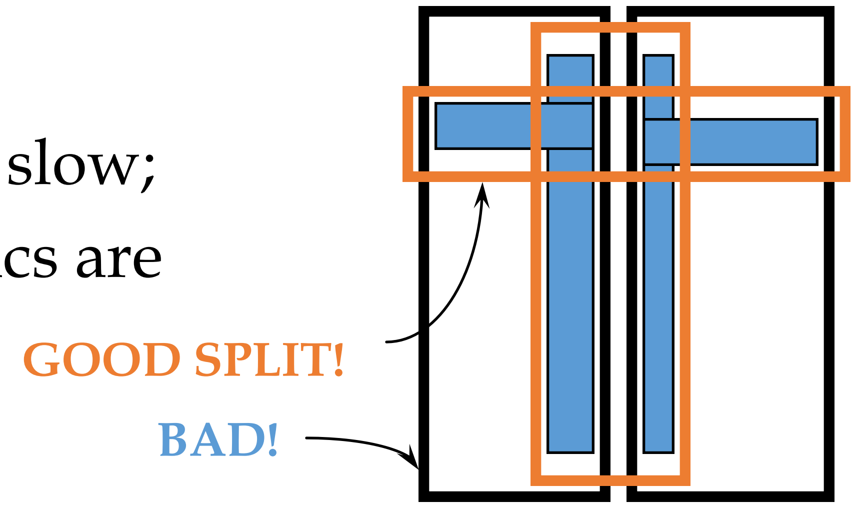
Insert Entry <B, ptr>

- Start at root and go down to "best-fit" leaf L.
  - Go to child whose box needs least enlargement to cover B; resolve ties by going to smallest area child.
- If best-fit leaf L has space, insert entry and stop. Otherwise, split L into L1 and L2.
  - Adjust entry for L in its parent so that the box now covers (only) L1.
  - Add an entry (in the parent node of L) for L2. (This could cause the parent node to recursively split.)

Splitting a Node During Insertion

- The entries in node L plus the newly inserted entry must be distributed between L1 and L2.
- Goal is to reduce likelihood of both L1 and L2 being searched on subsequent queries.
- Idea: Redistribute so as to minimize area of L1 plus area of L2.

Exhaustive algorithm is too slow; quadratic and linear heuristics are described in the paper.



R-Tree Variants

- The R* tree uses the concept of forced reinserts to reduce overlap in tree nodes. When a node overflows, instead of splitting:
  - Remove some (say, 30% of the) entries and reinsert them into the tree.
  - Could result in all reinserted entries fitting on some existing pages, avoiding a split.
- R* trees also use a different heuristic, minimizing box perimeters rather than box areas during insertion.
- Another variant, the R+ tree, avoids overlap by inserting an object into multiple leaves if necessary.
  - Searches now take a single path to a leaf, at cost of redundancy.