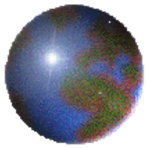
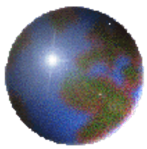
CSEG601 & CSE5601:



Spatial Data Management & Applications

Sungwon Jung

Big Data Processing & DB Lab
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr



Spatial Access methods 4

1. The z-Ordering Tree
2. A Raster Variant of z-ordering (with redundancy)

The z-Ordering Tree

- The structure of z-ordering tree does not use as an approximation of the object its *mbb*
  - The geometry of each object is decomposed into a quadtree of depth bounded by *d*
  - One indexes the set of quadtree leaves that approximate the geometry
  - The leaves' labels (whose size is ≤ *d*) are inserted into a B+tree

The z-Ordering Tree

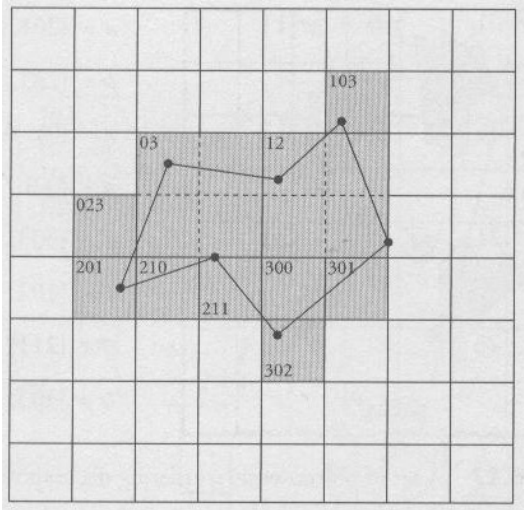
- Basic Step

```
DECOMPOSE (o: geometry, q: quadrant): set(quadrant)

begin
  decomp_NW, decomp_NE, decomp_SW, decomp_SE: set(quadrant)
  result = ∅
  // Check that q overlaps o, else do nothing
  if (q overlaps o) then
    if (q is minimal) then
      result = {q}
    else
      // Decompose o into pieces, one per subquadrant
      for each sq in {NW(q), NE(q), SW(q), SE(q)} do
        decomp_sq = DECOMPOSE (o, sq)
      end for
      // If each decomposition results in the full subquadrant, return q
      if (decomp_NW ∪ decomp_NE ∪ decomp_SW ∪ decomp_SE = q) then
        result = {q}
      else
        // Take the set union of the four decompositions
        result = decomp_NW + decomp_NE + decomp_SW + decomp_SE
      end if
    end if
  end if
  return result
end
```

The z-Ordering Tree

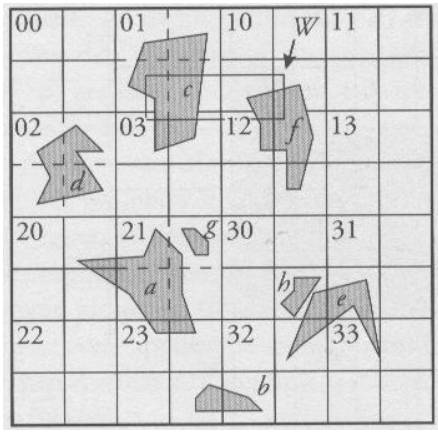
- z-ordering and object decomposition:
  - d = 3



{023, 03,103,12,201,210, 211, 300, 301, 302}

The z-Ordering Tree

- A Set of objects with z-ordering decomposition



a = {201, 203, 21, 230, 231}
b = {233, 322}
c = {01, 030, 031}
d = {02}
e = {303, 312, 321, 330}
f = {102, 103, 120, 121, 123}
g = {211}
h = {303}

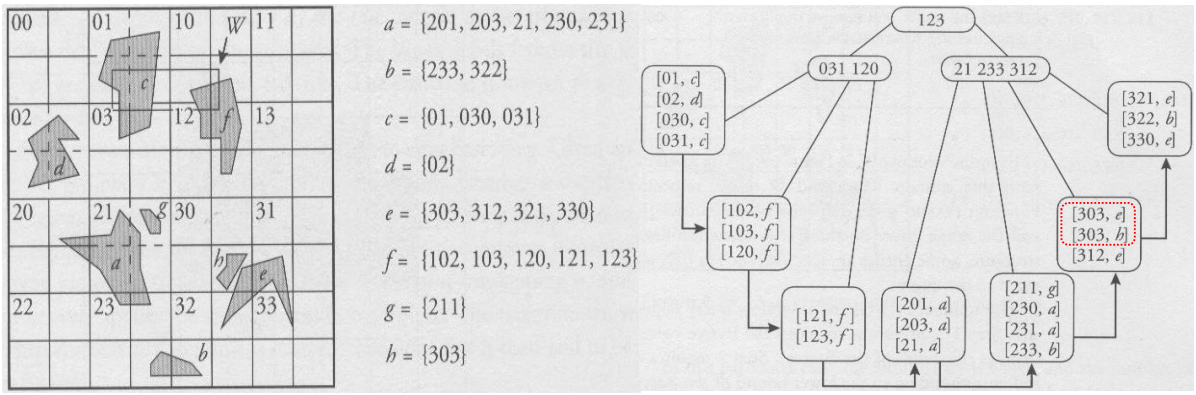
- object a is represented by
  - {[201, a], [203, a], [21, a], [230, a], [231, a]}

The z-Ordering Tree

- Construction algorithm:
  - Decompose each object by *z*-order
  - Represent each object in the form of [*l, oid*] entry (where *l* represents label)
    - Example: object *a*
    - {[201, *a*], [203, *a*], [21, *a*], [230, *a*], [231, *a*]}
  - Construct B^+ tree from a set of entries[*l, oid*]

The z-Ordering Tree

- Example

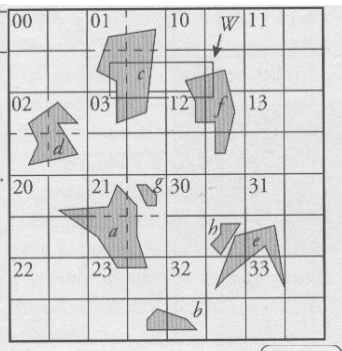


- the same object may be represented in two quite distant leaves of the B^+ tree due to the inherent proximity property of z-order
  - e.g. object *b*

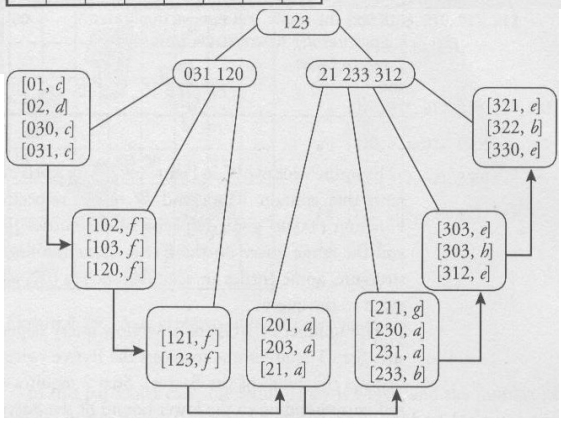
The z-Ordering Tree

```
ZO-WindowQuery (W: rectangle): set(oid)

begin
  result = ∅
  // Step 1: From the vertices W.nw and W.se of the window, compute
  // the interval [L, L']. This necessitates two searches through the B+tree.
  l = PointLabel (W.nw); [L, p] = MaxInf (l)
  l' = PointLabel (W.se); [L', p'] = MaxInf (l')
  // Step 2: Compute the set E of entries [l, oid] such that l ∈ [L, L'].
  E = RangeQuery ([L, L'])
  // Step 3: For each entry in E that overlaps W, report the oid
  for each e in E do
    if (Quadrant (e.l) overlaps W) then result += {e.oid}
  end for
  // Sort the result, and remove duplicates
  Sort (result); RemoveDupl (result);
  return result
end
```



a = {201, 203, 21, 230, 231}
b = {233, 322}
c = {01, 030, 031}
d = {02}
e = {303, 312, 321, 330}
f = {102, 103, 120, 121, 123}
g = {211}
h = {303}



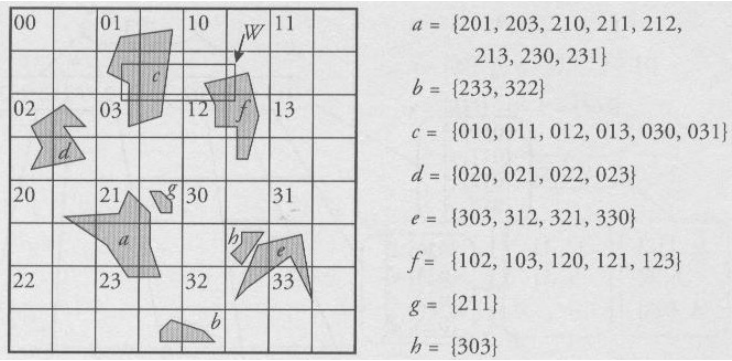
Result = {c, f}

The z-Ordering Tree

- The # of I/Os for a window query depends on the window size
  - Step 1: $2d$ I/Os
  - Step 2: d I/Os + k where k is the # of chained B^+ tree leaves to be scanned
- Comparison the linear quad tree to the *z*-ordering tree
  - *z*-ordering tree
    - the B^+ tree depth is likely to be larger because there are as many entries as there are cells in the decomposition of all objects
    - The finer grid resolution, the better the objects' approximation but the larger # of entries to be indexed in the B^+ tree
  - the linear quad tree
    - the # of B^+ tree entries is much smaller because it is equal to the # of quadtree quadrants
    - a supplementary I/O is required per quadrant overlapping the window

A Raster Variant of z-ordering (with redundancy)

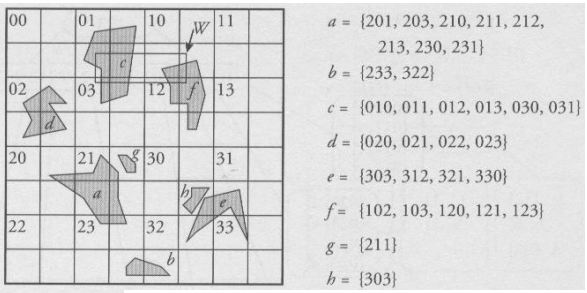
- A set of objects with raster decomposition
  - It consists of consists of decomposing the object into elementary grid cells

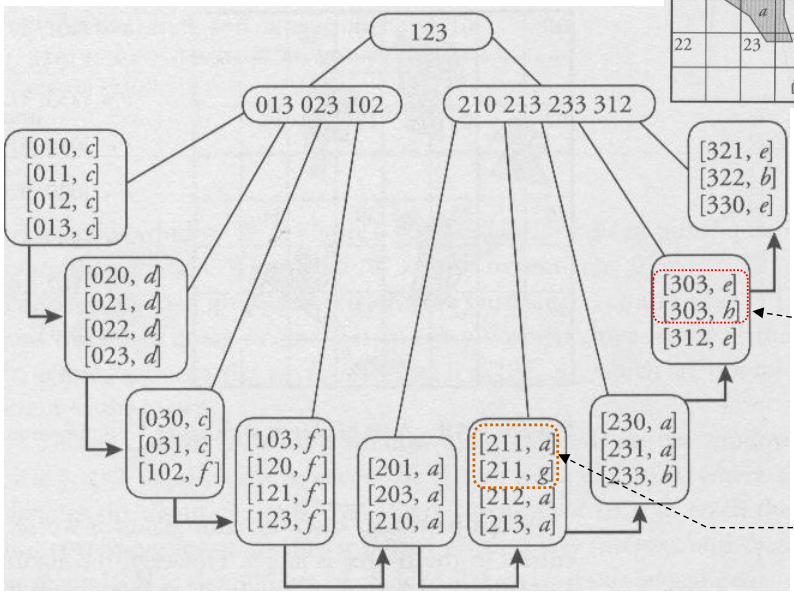


- the redundancy is worse and the # of entries in the B^+ tree is larger
- the algorithms for point and window queries are much simpler
  - Because all cells are minimal,
    - don't need function MAXINF and can use the B+ tree existing code implemented in each DBMS without any modification

A Raster Variant of z-ordering (with redundancy)

- z-raster tree



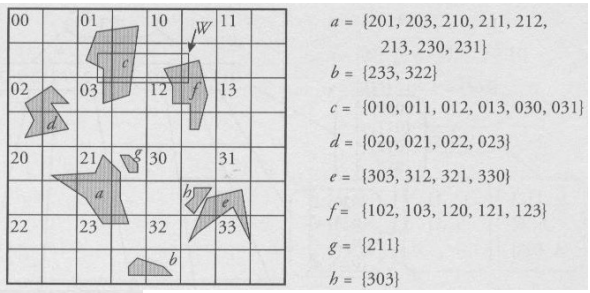


object e and object h overlap with 303

object a and object g overlap with 211

A Raster Variant of z-ordering (with redundancy)

- Window Query Algorithm




```
ZR-WINDOWQUERY (W: rectangle): set(oid)

begin
   result = ∅
   // Step 1: From the vertices W.nw and W.se of the window, compute
   // the interval [L, L']. This necessitates two searches through the B+tree.
   L = POINTLABEL (W.nw); L' = POINTLABEL (W.se)
   // Step 2: Compute the set E of entries [l, oid] such that l ∈ [L, L'].
   E = RANGEQUERY ([L, L'])
   // Step 3: For each entry in E that overlaps W, report the oid
   for each e in E do
      if (QUADRANT (e.l) overlaps W) then result += {e.oid }
   end for
   // Sort the result, and remove duplicates
   SORT (result); REMOVEDUPL (result);
   return result
end
```