Data Organization and Processing

Spatial Join

(NDBI007)

David Hokszа, Škoda Petr
http://siret.ms.mff.cuni.cz/hoksza

Outline

- Spatial join basics
- Relational join
- Spatial join

Spatial join definition (1)

- Given **two sets of multi-dimensional** objects in Euclidean space, a spatial join finds **all pairs** of objects **satisfying** a given **spatial relation** between the objects, such as intersection.

- **Simplified spatial join**
    - Given two sets of rectangles, R and S, find all of the pairs of intersecting rectangles between the two sets, i.e. $\{(\mathrm{r}, \mathrm{s}): r \cap s \neq \emptyset, r \in R, s \in S\}$

Spatial join definition (2)

- **Spatial overlay join (general spatial join)**
  - the data set can consist of **general spatial objects** (points, lines, polygons)
  - the data sets can have **more** than **two dimensions**
  - the relation between pairs of spatial objects can **be any spatial relation** (nearness, enclosure, direction, …)
  - there can be more sets in the relation (**multiway spatial join**) or one set joined with itself (**self spatial join**)

Spatial join examples

- Find all pair of rivers and cities that intersect.
- Find all of the rural areas that are below sea level, given an elevation and land use map.
- Find the houses inside the areas with poor slope stability.

Non-spatial join

- There exist algorithms for **standard relational join**
  - only equi-joins (the join predicate is equality) considered here
  - **nested loop** join *(hnízděné cykly)*
  - **sort-merge** join *(setřídění-slévání)*
  - **hash** join *(hashované spojení)*
  - most of the standard relational join algorithms are not suitable for spatial data because the join condition involves multidimensional spatial attribute

Non-spatial nested loop join

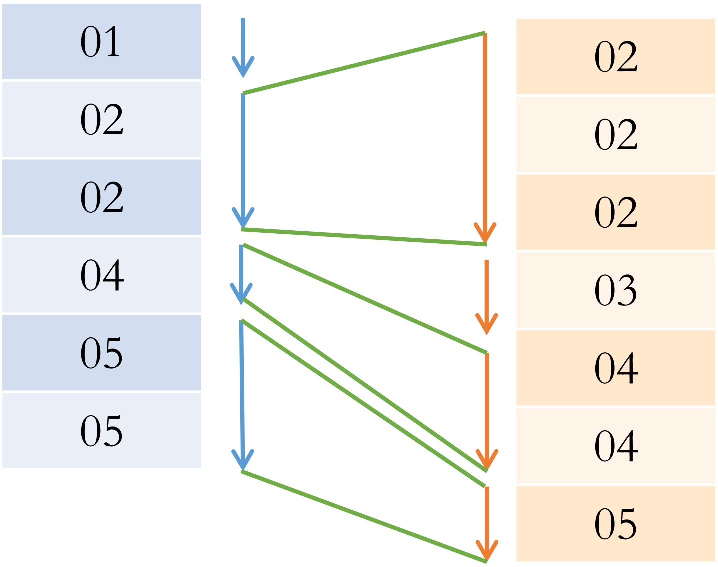
- Nested loop join checks one by one for each element of a relation *R* all elements in relation in *S*

```
FOREACH r ∈ R DO
  FOREACH s ∈ S DO
    IF cond(r,s) THEN REPORT(r,s)
```

- In its basic version, the nested loop join is the least efficient algorithm from the relational joins algorithms
- The only of the standard relational join algorithms applicable also to spatial data
  - Therefore special spatial join algorithms had to be developed

Non-spatial sort-merge join

- Two-phase algorithm
  - sort both relations R, S independently
  - scan both relations at once in the same order and join



- Multi-dimensional data do not preserve proximity so this method (used as is) is not applicable to spatial data

Non-spatial hash join

- Hash join
  - One of the relations (R) is hashed with a hash function h (the assumption that R fits into main memory)
  - The other relation (S) is processed one by one and the elements' ids are hashed with h
  - If for two elements $r \in R, s \in S: h(r) = h(s)$ then r and s are checked for $r.cmpr\_attributes = s.cmpr\_attributes$

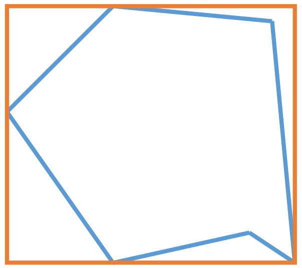
- Equijoins rely on grouping objects with the same value which is not possible for spatial objects since these have an extent

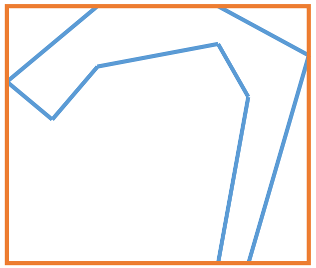
Filter-refine policy

- The **objects** to be searched for **can be very complex**
  - → testing of the join condition (spatial predicate) itself can be highly time demanding
  - → not many objects fit into main memory
    - → filter-refine strategy
- Spatial objects are approximated using simple spatial objects
- **Filter-refine**
  - **filter** - the spatial join is conducted using the objects' **approximations** → **candidate set**
  - **refine** – pairs which pass the filter are **tested for the spatial predicate** using their **full spatial representation**

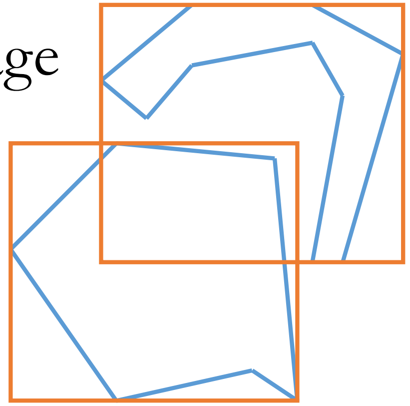
Approximations (1)

- The most common approximation is **minimum bounding rectangle (MBR)** – the smallest rectangle fully enclosing given object whose sides are parallel to the axes
- **Dead space**
  - the amount of **space covered by the approximation but not the approximated object**
  - the approximation should aim to **minimize dead space**
  - large dead space areas can lead to higher false hit rate in the filtering stage





Large dead area

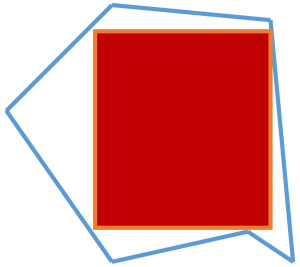


False hit

Approximations (2)

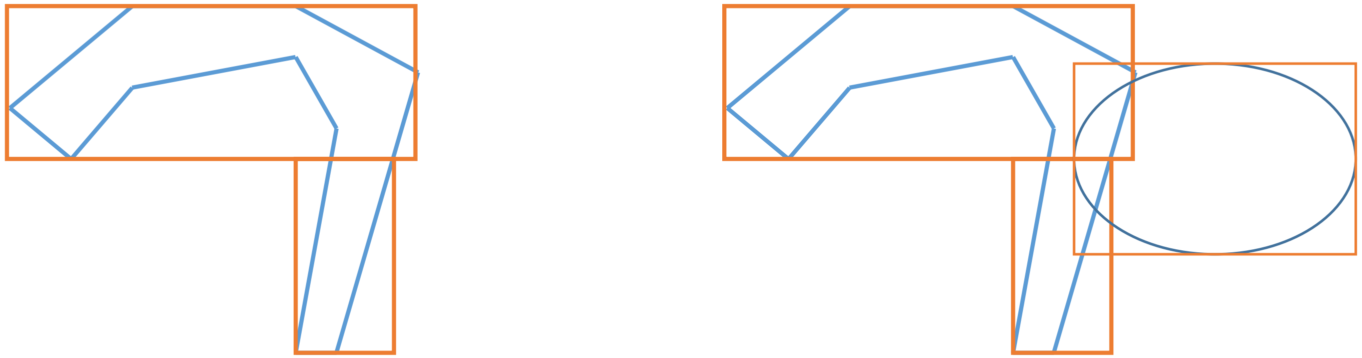
- **Conservative**
  - every point of the object is also point of the approximation
    - concave
    - convex
      - MBR – most common
      - minimum bounding polygon
      - minimum bounding circle/ellipse

- **Progressive**
  - every point of the approximation is also point of the object
    - maximum nested rectangles, circles, …



Approximations (3)

- **Object decomposition**
  - minimizes dead space by decomposing an object into disjoint fragments with their own MBRs
  - after refine step and extra filtering stage is needed to remove duplicities



Spatial join methods

Internal Memory Methods

- Nested loop join
- Index nested loop join
- Plane Sweep
- Z-order

External Memory Methods

- Hierarchical traversal
- Partitioning-based methods

Internal memory methods

- Nested loop join
- Index nested loop join
- Plane Sweep
- Z-order

Nested loop join

- The algorithm is **identical to the standard relational one**
  - works with **arbitrary object type** and **join condition**
- Given datasets A and B the join takes $O(|A| \times |B|)$ time
  - suitable for **small datasets**

```
NESTED_LOOP_JOIN(setA, setB, joinCondition)
INPUT: Sets to join and condition based on which the join
happens
OUTPUT: Pairs of objects satisfying the join condition

FOREACH a ∈ setA
   FOREACH b ∈ setB
      IF Satisfied(a, b, joinCondition) THEN
         REPORT(a, b);
```

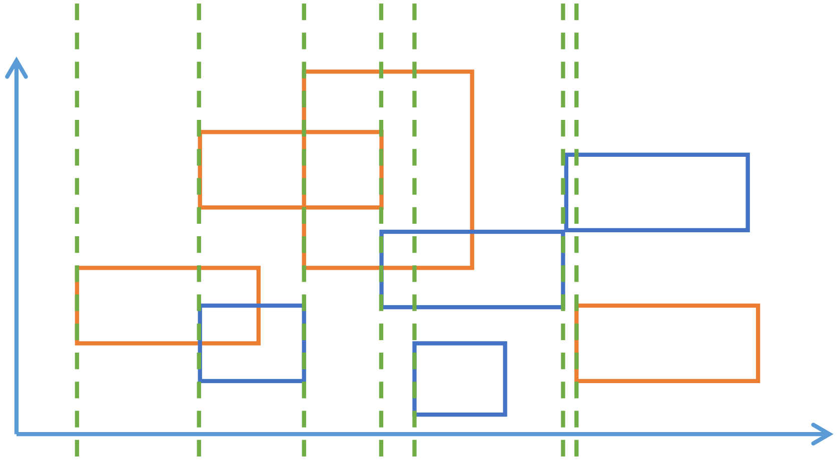
Index nested loop join

- Variant of nested loop join where **first a spatial index** is created over one of the sets
- The **indexed set is queried** by each of the objects from the second set for intersection (window query)
- Given datasets A and B the join takes $O(\log(|A|) \times |B|)$ time (not including time needed for building the index)

```
INDEX_NESTED_LOOP_JOIN(setA, setB)
INPUT: Sets to join
OUTPUT: Pairs of intersecting objects

    ix := CreateSpatialIndex(setA)
    FOREACH b ∈ setB
        REPORT(ix.Search(b));
```

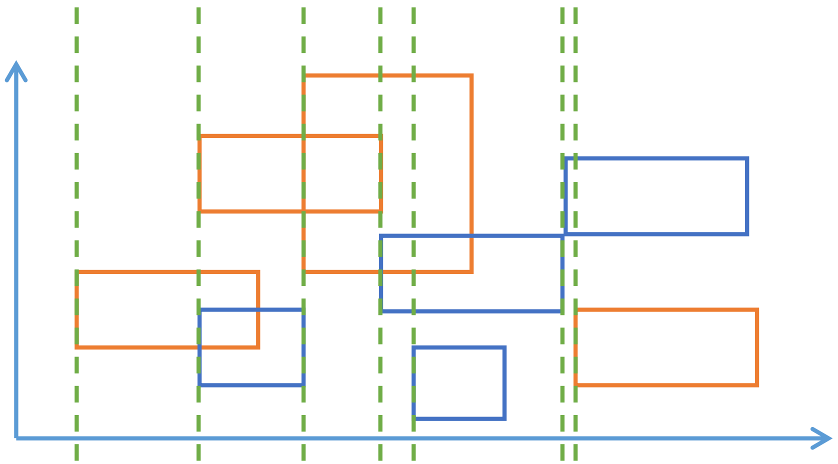
Plane sweep

- **Two-phase** algorithm for identification of intersecting rectangles
  - **sorting the rectangles** in ascending order on the basis of their left sides (x-axis)
  - **sweeping a vertical scan line** through the sorted list from left to right, halting at each of the rectangles lower x coordinates and identification of rectangles intersecting (vertical line) with the current rectangle and checking for intersection based on the y-axis



Similar to non-spatial sort-merge join

Plane sweep algorithm (1)



```
PLANE_SWEEP(setA, setB)
INPUT: Sets of rectangles to join
OUTPUT: Pairs of intersecting rectangles

    listA ← SortByLeftSide(setA);
    listB ← SortByLeftSide(setB);
    sweepStructA ← CreateSweepStructure();
    sweepStructB ← CreateSweepStructure();
    WHILE NOT(listA.End()) || NOT (listB.End()) DO
        IF listA.First() < listB.First() THEN
            sweepStructA.Insert(listA.First());
            sweepStructB.RemoveInactive(listA.First());
            sweepStructB.Search(listA.First());
            listA.Next();
        ELSE
            sweepStructB.Insert(listB.First ());
            sweepStructA.RemoveInactive(listB.First());
            sweepStructA.Search(listB.First());
            listB.Next();
```

Plane sweep algorithm (2)

- Sweep structure tracks active rectangles and has to support three operations
  - **`Insert`**
    - insert a rectangle into the active set
  - **`RemoveInactive`**
    - removes from the active set all rectangles that do not overlap a given rectangle (line)
  - **`Search`**
    - searches for all active rectangles that intersect a given rectangle and outputs them
  - if the data are sorted, only the data in the sweep structures need to be kept in internal memory
- Given datasets A and B the algorithm takes $O(\log(n) \times n)$ where $n = |A + B|$, including the list sort time.

Z-ordering

- Z-order methods are based on **representing an object using a set of continuous segments corresponding to Z-ordering**
  - object $o \rightarrow \{(z_1, o), \ldots (z_n, o)\}$
- The z-order-based representation can be processed in different ways
  - **sort-merge**
    - standard relational sort-merge but the sets are z-ordered
  - **plane-sweep**
    - the active set, instead of being the objects that intersect the sweep line, are the enclosing cells of the objects that intersect the current Z-order grid cell
    - the sweep structure can be represented by a stack where each object in the stack intersects the query object

External memory methods

- Hierarchical traversal
- Partitioning-based methods

Hierarchical traversal

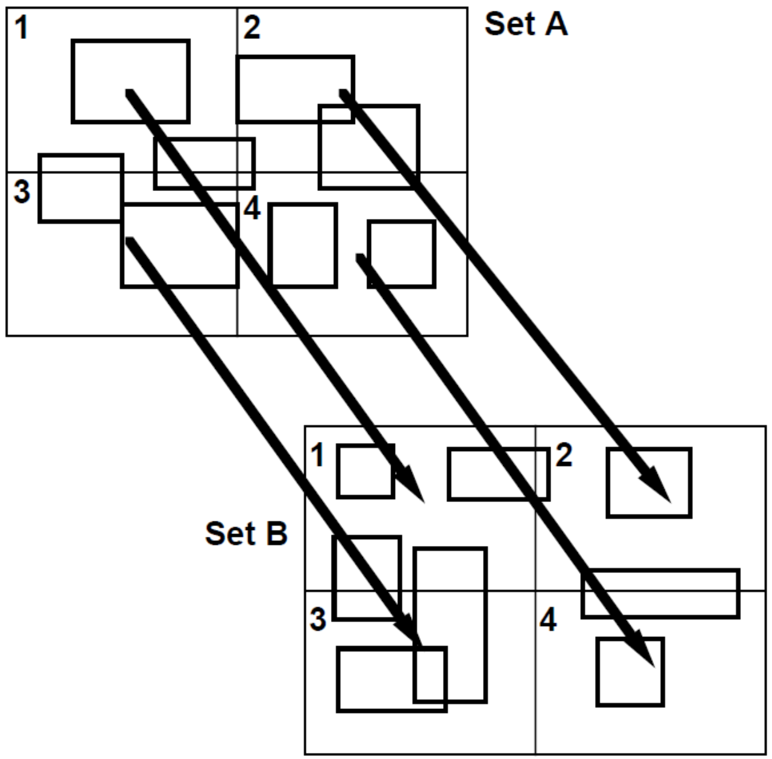
- Assumption - **both datasets** are **indexed** using a hierarchical index such as R-tree
- **Synchronized traversal** can be used to test the join condition
  - the algorithm traverses the two trees in a synchronized fashion and **compares bounding objects at given levels**
    - for indexes of different heights, the join of a leaf of one index with a sub-tree of the other can be accomplished using a window query, or handling leaf-to-node comparison as special cases
  - if a node corresponding to a part of the space does not match the condition it can be excluded from the traversal
    - iterative filter and refine approach
  - reading a node usually corresponds to reading one memory page

General synchronized traversal

```
INDEXED_TRAVERSAL_JOIN(rootA, rootB)
INPUT: Roots of the structures representing the sets to be joined
OUTPUT: Pairs of intersecting rectangles

   queue ← CreateQueue();
   queue.Add(pair(rootA, rootB));
   WHILE NOT(queue.Empty()) DO
      nodePair ← queue.Pop();
      pairs ← IdentifyIntersectingPairs(nodePair);
      FOREACH p ∈ pairs DO
         IF p is leaf THEN ReportIntersection(p);
         ELSE queue.Add(p);
```

Partitioning

- Often applied when neither of the sets to be joined is indexed
- Resulting partitions should be small enough to fit in internal memory
- Once the data are partitioned, each pair of overlapping partitions is read into internal memory and internal memory techniques are used

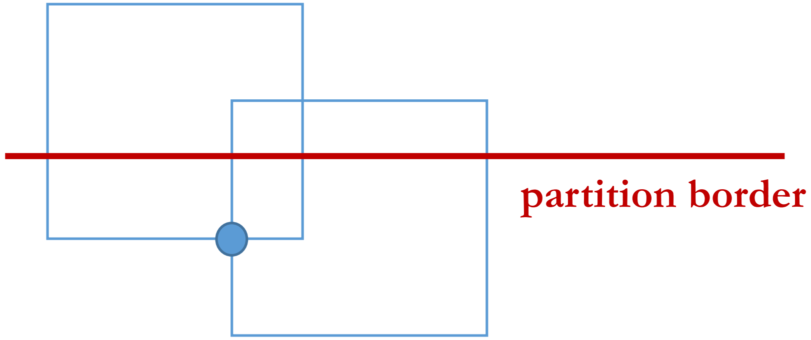


Partition join of uniform data

```
GRID_JOIN(setA, setB)
INPUT: Sets of objects to be joined
OUTPUT: Pairs of intersecting objects

   m ← AvailableInternalMemory();
   mbrSize ← BytesToStoreMBR();
   minNrOfPartitions ← (setA.Size() + setB.size())*mbrSize() / m;
   partList ← DeterminePartitions(minNrOfPartitions);

   {object appears in every partition it intersects}
   partitionPointersA ← PartitionData(partList, SetA);
   partitionPointersB ← PartitionData(partList, SetB);

   FOREACH part ∈ partList DO
      partitionA ← ReadPartition(partitionPointersA, part);
      partitionB ← ReadPartition(partitionPointersB, part);
      PLANE_SWEEP(partitionA, partitionB);
```

Avoiding duplicate results

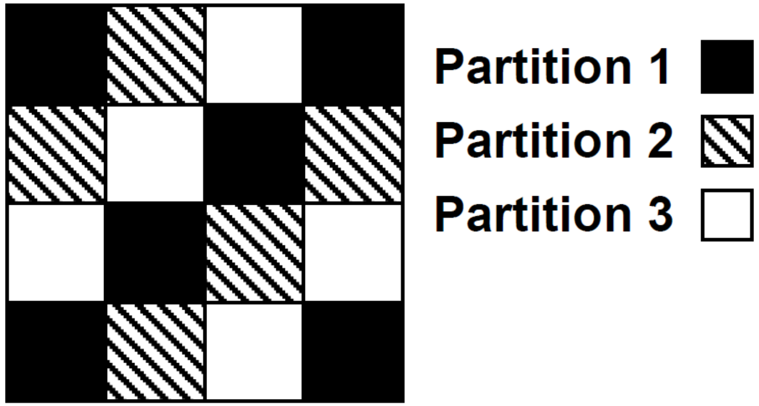
- Space partitioning can lead to object duplication which can, in turn, lead to duplication of results

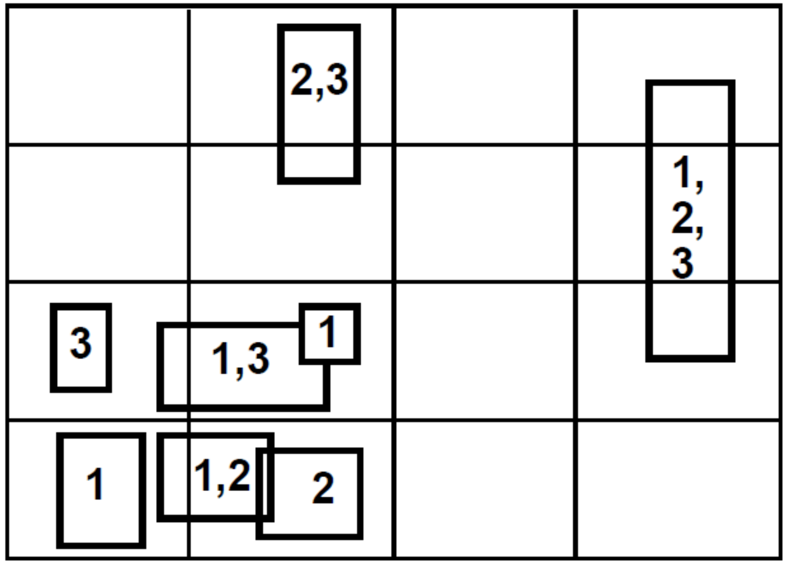


- **Deduplication**
  - **sort and remove duplicates**
    - requires sorting which implies increased computational demands
  - **reference point method**
    - a consistently chosen reference point is selected from the intersecting region
    - intersection is reported only if the reference point lies within given partition

Grid partitioning – skewed distributions

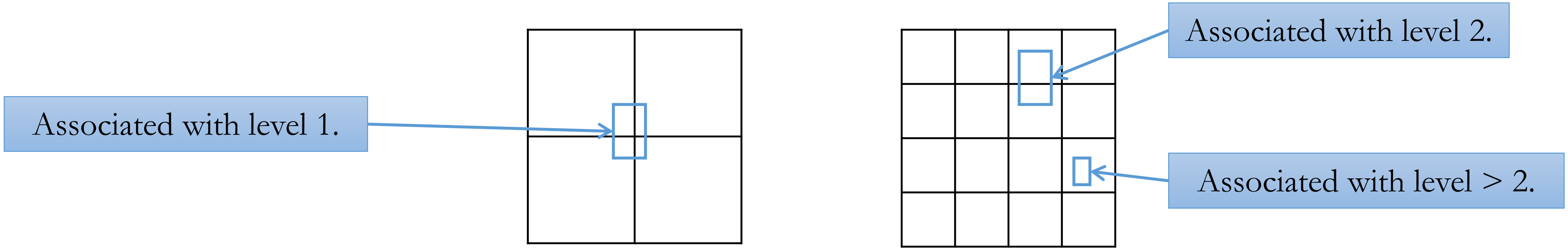
- Basic grid algorithm is rarely used since the objects distribution is often not uniform
- [Patel & DeWitt; 1996] proposed to group partitions using a mapping function to minimize skew by creating partitions having similar number of items





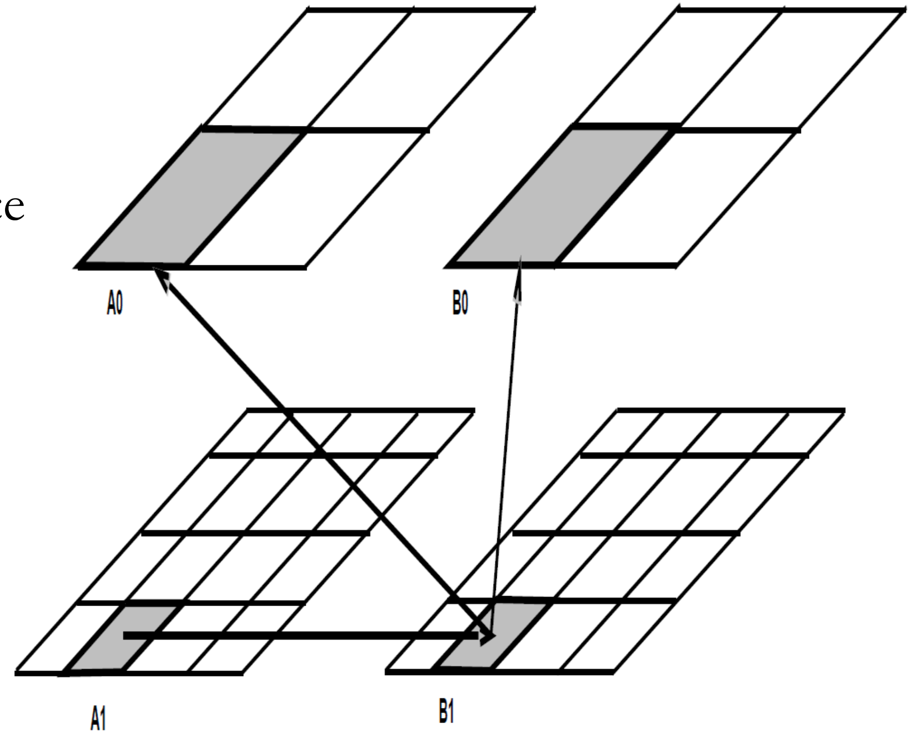
Size-based partitioning

- [Koudas & Sevcik; 1997]
- Partitioning of data using a **series of finer and finer grids**
  - **level *i* grid contains 4^{i-1} partitions** (*i* horizontal and *i* vertical splitting axis)
  - each **object is placed into the partition** associated with the grid in which the object **does not intersect the grid lines**
    - in second-level grid are objects which cross level 3 partition borders (16 cells) but not level 2 partition borders (4 cells)
    - each partition is a filter and objects fall through to the lowest level partition where a partition boundary is crossed



Searching in size-based partitioning

- Given two spatial data sets A and B
  - Scan data sets A and B and for each entity:
    1. Compute the Hilbert (Z) value of the entity
    2. Determine the level at which the entity belongs and place its entity descriptor in the corresponding level file
  - For each level file
    1. Sort by Hilbert (Z) value
  - Perform a synchronized scan over the pages of level files



- Each page of each level file is read just once
  - Entries in page A^l (l-th level) need to be checked only against actual pages $B^m (m \leq l)$