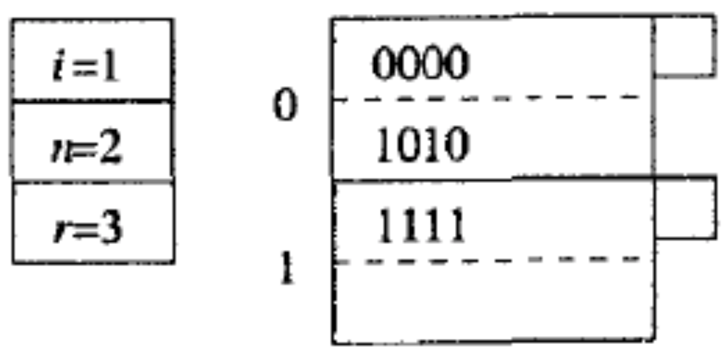
Linear hash table

- This is another dynamic hash table approach
  - Bucket number grows more slowly than with extensible hashing
  - Fill factor: Number of buckets **n** always chosen so average number of <key, value> pairs (value **r**) is some constant fraction of **n** (e.g. $r \leq 1.7n$)
  - Blocks cannot be split, so overflow blocks are used instead (but average # of overflow blocks will be much less than 1)
  - Number of bits used number bucket entries is ceiling($\log_2 n$)

Small linear hash table

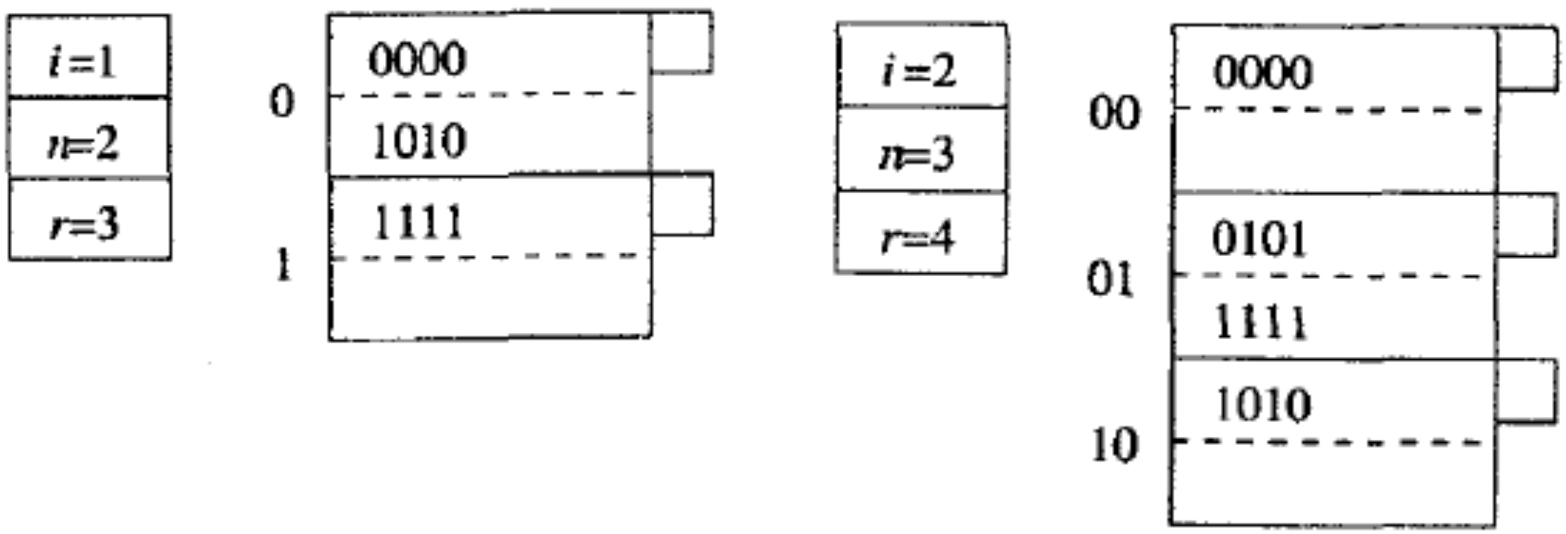
- Note the three global values
  - i: number of bits used to determine bucket for key
  - n: number of buckets
  - r: number of entries in the hash table
- Unlike extensible hashing, we use least-significant bits of key to compute bucket location.



Linear hash table insertion

- Goal:
  - Compute the bucket in which the <key, value> pair should be placed.
  - If there is room in the bucket, great!
  - If no room, then create an overflow bucket.
  - If occupancy exceeds the fill factor, then create a new bucket.
- Computing bucket looks more complicated than it is.
  - Given some key K...
  - ... denote its least-significant i bits as $a_1a_2..a_i$
  - ... and call this bit sequence **m**.
  - If $m < n$: bucket numbered m exists, and place key-value pair in that bucket.
  - If $n \leq m \leq 2^i$ then bucket m does not yet exist, so put key-value pair into bucket $m - 2^{i-1}$ (i.e., same as setting a1 to 0)

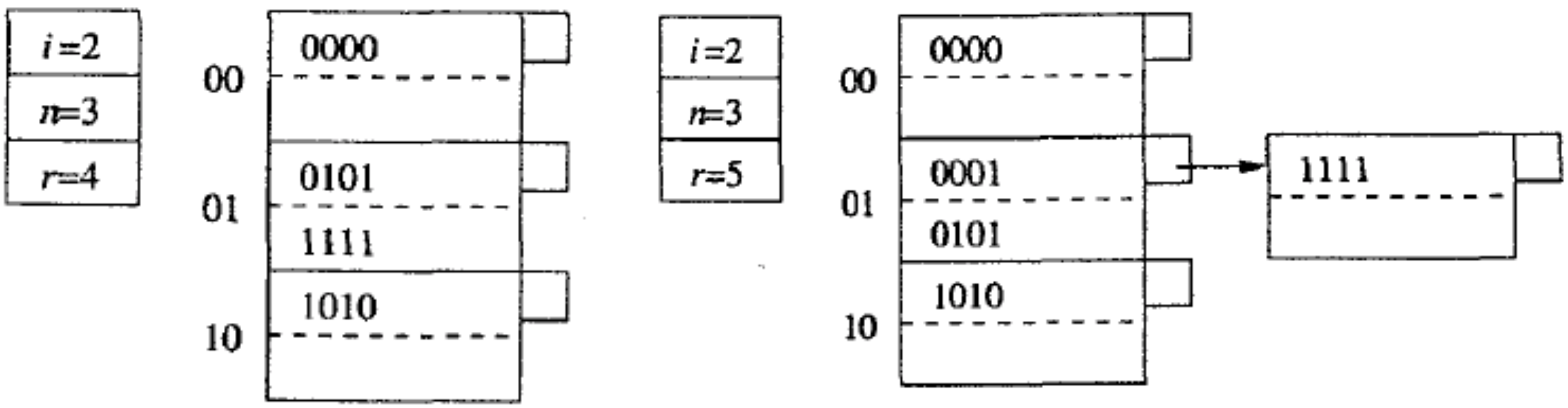
Insert pair with key 0101



before

after

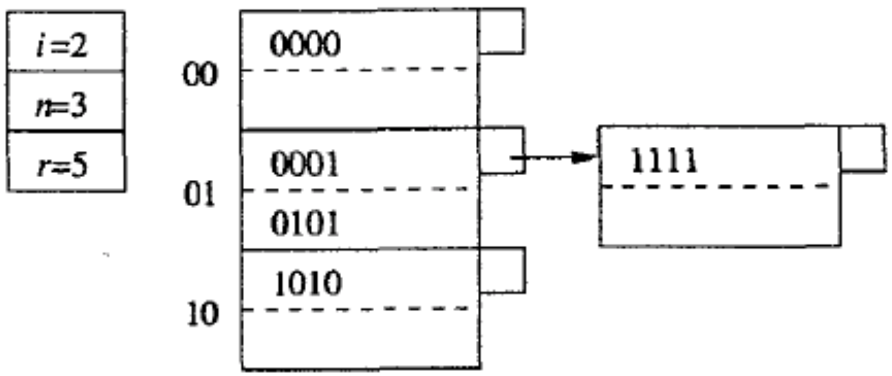
Insert pair with key 0001

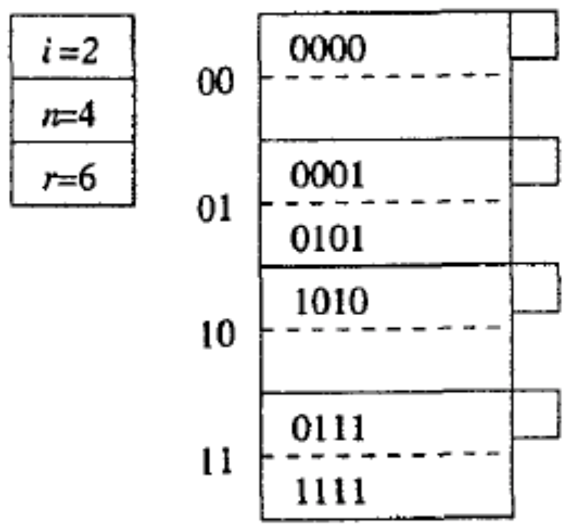


before

after

Insert pair with key 0111





before

after

Story so far

- All of the indexes so far have been one-dimensional
  - Each has a single search key (which may comprise one or more table attributes)
  - Values for all attributes of the search key must be provided.
  - Index search is through this sequence of values for a matching index key (and corresponding data-file block).
- B+ Trees:
  - Single linear ordering for keys
- Hash tables:
  - Search key is completely known for lookup

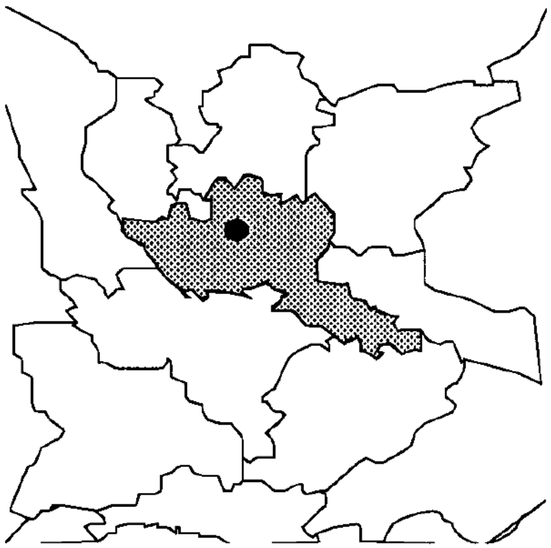
Multidimensionality

- Many current applications rely on the use of **multidimensional data**
  - Geosciences; cartography
  - Mechanical CAD; VLSI CAD; 3D CG modelling
  - Robotics
  - Visual perception
  - Autonomous navigation
  - Environmental protection
  - Medical imaging
- May also refer to this as **spatial data**

What is so special about multidimensional data?

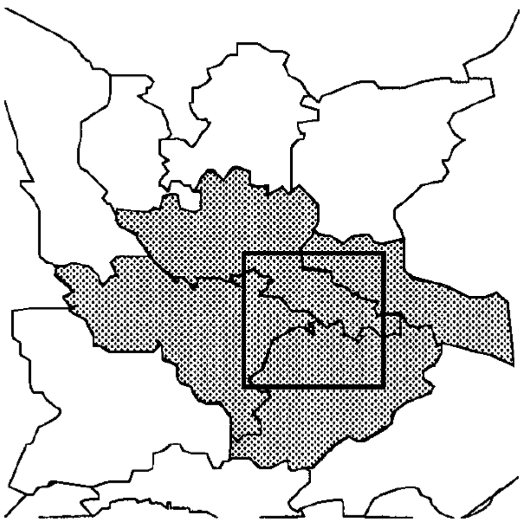
- Complex structure
- Often dynamic
- Tend to be large
- No standard algebra on multidimensional data
- Operators are not closed
- Computational costs vary among multidimensional database operators

Kinds of multidimensional queries (1)



Point query

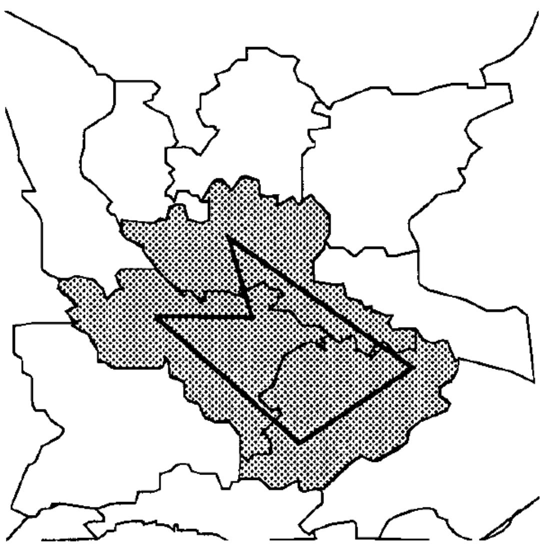
$PQ(p) = \{o \mid p \cap o.G = p\}$



Window query (also Range query)

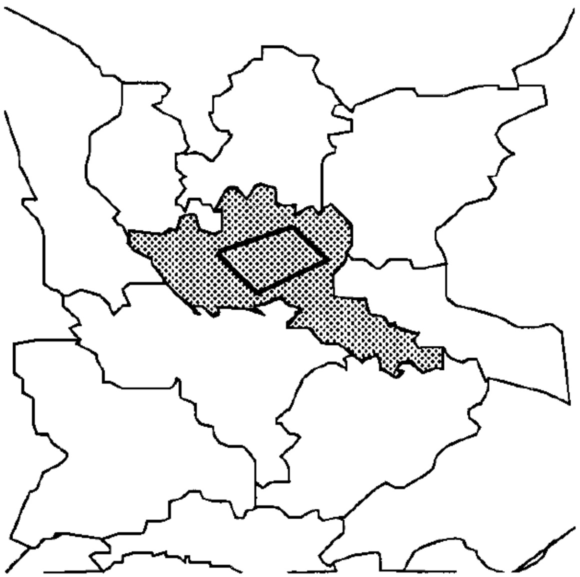
$WQ(I^d) = \{o \mid I^d \cap o.G \neq \emptyset\}$

Kinds of multidimensional queries (2)



Intersection Query

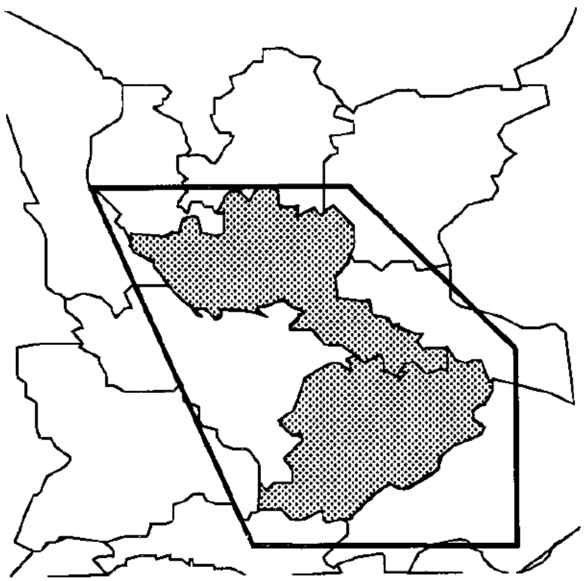
$IQ(o') = \{o \mid o'.G \cap o.G \neq \emptyset\}$



Enclosure query

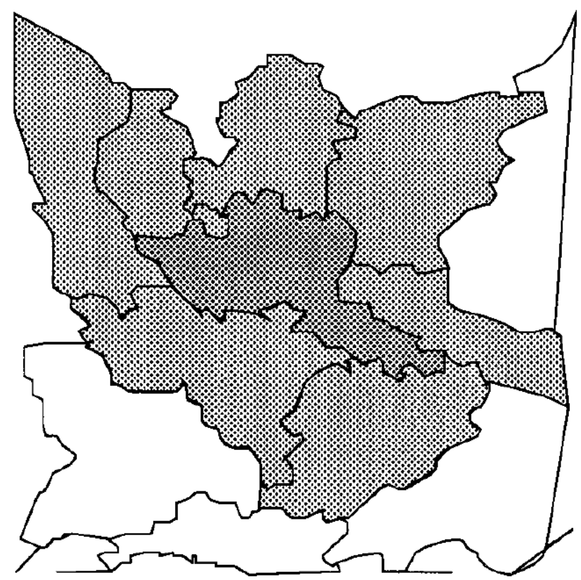
$EQ(o') = \{o \mid (o'.G \cap o.G) = o'.G\}$

Kinds of multidimensional queries (3)



Containment Query

$CQ(o') = \{o \mid (o'.G \cap o.G) = o.G\}$



Adjacency Query

$AQ(o') = \{o \mid o.G \cap o'.G \neq \emptyset \wedge o'.G^{\circ} \cap o.G^{\circ} = \emptyset\}$

Other queries

- Exact match query
  - $EMQ(o') = \{o \mid o'.G = o.G\}$
- Nearest-neighbour query
  - $NNQ(o') = \{o \mid \forall o'' : dist(o'.G, o.G) \leq dist(o'.G, o''.G)\}$
- Spatial join
  - R and S are collections of multidimensional objects
  - $R \bowtie_\theta S = \{(o, o') \mid o \in R \wedge o' \in S \wedge \theta(o.G, o'.G)\}$
  - θ is a spatial predicate (e.g., intersects, contains, is_enclosed_by, distance, etc.)