Main Memory Indexing

Hardware Trends

CPU speed and memory capacity double every 18 months.

Memory performance merely grows 10%/year:

- Capacity vs speed (esp. latency)

The gap grows 10X every 6 years! And 100 times since 1986.

Implications

- Many databases can fit in main memory
- But memory access will become the new bottleneck
- No longer a uniform random access model (NUMA)!
- Cache performance becomes crucial

NUMA: The time to access each piece of data is assumed to be the same

Memory Basics

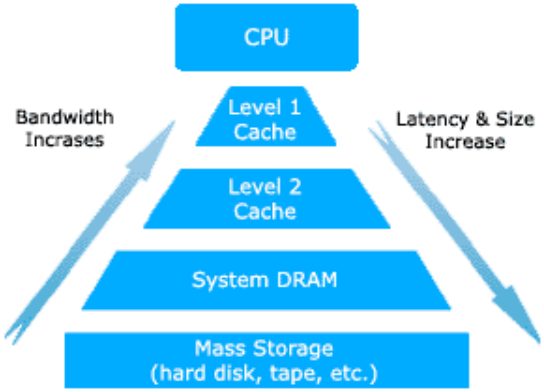
Cache Line: Smallest unit that can be retrieved

Objective: Try to align the data structure to align with the cache size or a multiple of it

- Memory hierarchy:
  - CPU
  - L1 cache (on-chip): 1 cycle, 8-64 KB, 32 byte/line
  - L2 cache: 2-10 cycle, 64 KB-x MB, 64-128 byte/line
  - TLB: 10-100 cycle. 64 entries (64 pages).
  - Capacity restricted by price/performance.
- Cache performance is crucial
  - Similar to disk cache (buffer pool)
  - Catch: DBMS has **no** direct control.

TLB: memory cache that is used to reduce the time taken to access a user memory location.

Separation of concern really. The hardware should be a black box with a well defined interface as should the OS be, also with a defined interface. The reality, however, is that co-design between different layers happens all the time, e.g., because the DB has a better idea on the future data access patterns.



Improving Cache Behavior

- Factors:
  - Cache (TLB) capacity.
  - Locality (temporal and spatial).
- To improve locality:
  - Non random access (scan, index traversal):
    - Clustering to a cache line.
    - Squeeze more operations (useful data) into a cache line.
  - Random access (hash join):
    - Partition to fit in cache (TLB).
  - Often trade CPU for memory access

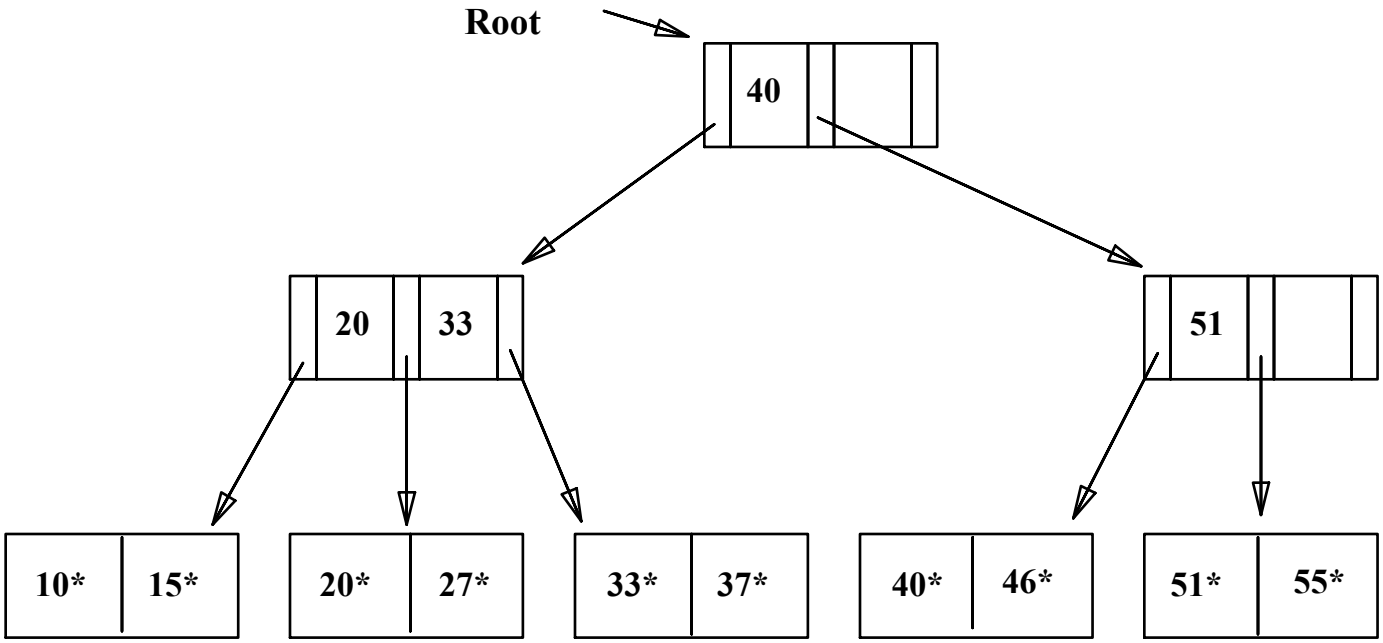
Compress data => Requires less space in RAM => Needs time to decompress

A cache line is the unit of data transfer between the cache and main memory. Typically the cache line is 64 bytes. The processor will read or write an entire cache line when any location in the 64 byte region is read or written. The processors also attempt to prefetch cache lines by analyzing the memory access pattern of a thread.

Cache Conscious Indexing

Example Tree Index

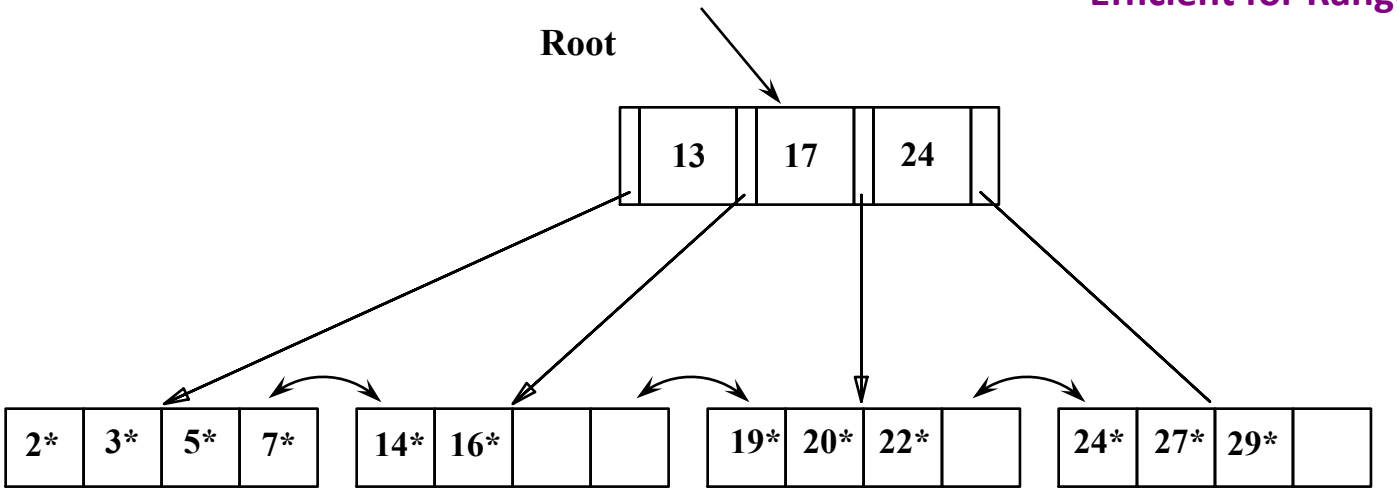
- *Index entries*: <search key value, page id> they direct search for data entries *in leaves.*
- Example where each node can hold 2 entries



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...

Efficient for Range Queries



B+ Tree - Properties

- *Balanced*
- Every node *except root* must be at least ½ full.
- *Order*: the minimum number of keys/pointers in a non-leaf node
- *Fanout* of a node: the number of pointers out of the node

B+ Trees: Summary

- Searching:
  - $\log_d(n)$ – Where *d* is the order, and *n* is the number of entries
- Insertion:
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching
- Deletion
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly

Cache Sensitive Search Tree

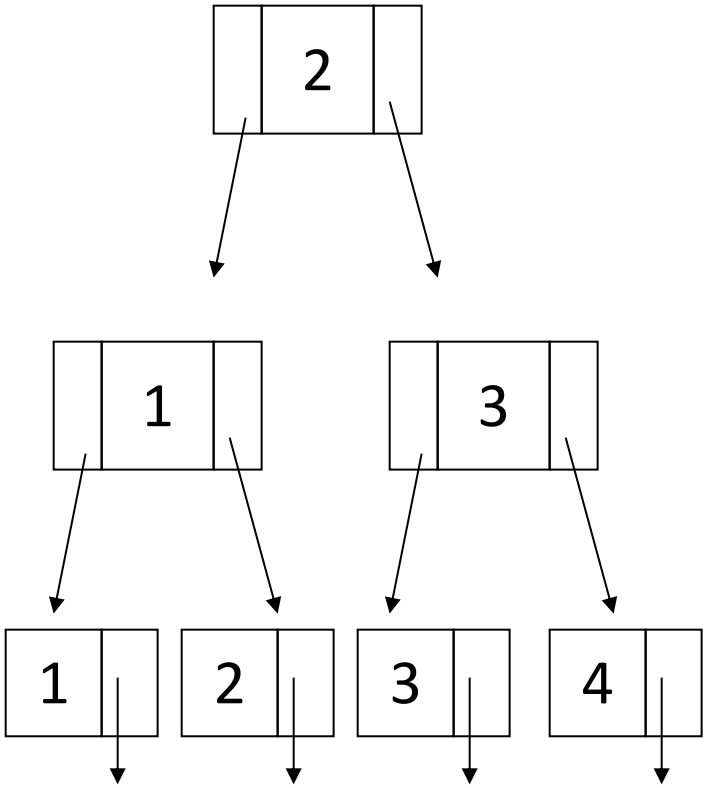
- Key: Improve locality
- Similar as B+ tree (the best existing).
- Fit each node into a L2 cache line
  - Higher penalty of L2 misses.
  - Can fit in more nodes than L1. (32/4 vs. 64/4)
- Increase fan-out by:
  - Variable length keys to fixed length via dictionary compression.
  - Eliminating child pointers
    - Storing child nodes in a fixed sized array.
    - Nodes are numbered & stored level by level, left to right.
    - Position of child node can be calculated via arithmetic.

Suppose cache line size = 24 bytes,

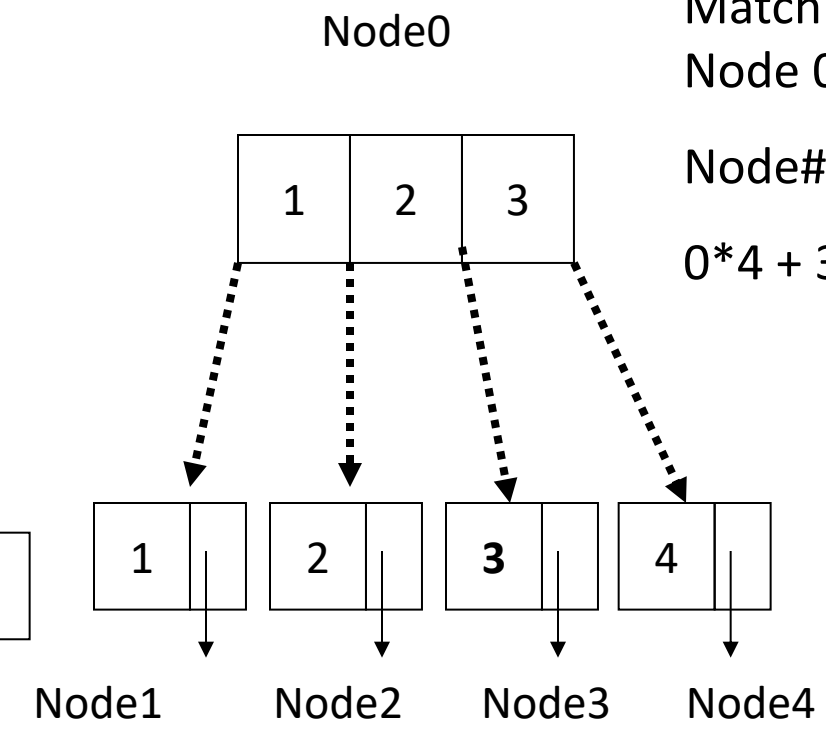
Key Size = Pointer Size = 4 bytes

Misses = Heights of the Tree
E.g. Find value 4 from the Tree

B+ tree, 2-way, 3 misses



CSS tree, 4-way, 2 misses



Search K = 3,

Match 3rd key in Node 0.

Node# =

0*4 + 3 = Node 3

Performance Considerations

- Search improvement over B+ tree:
  - $\log_{m/2} n / \log_{m} n - 1 = 1/(\log_2 m - 1)$
  - As cache line size = 64 B, key size = 4, m = 16.33%.
- Space
  - About half of B+ tree (pointer saved)
  - More space efficient than hashing and T trees
- CSS has the best search/space balance.
  - Second the best search time (except Hash – very poor space)
  - Second the best space (except binary search – very poor search)

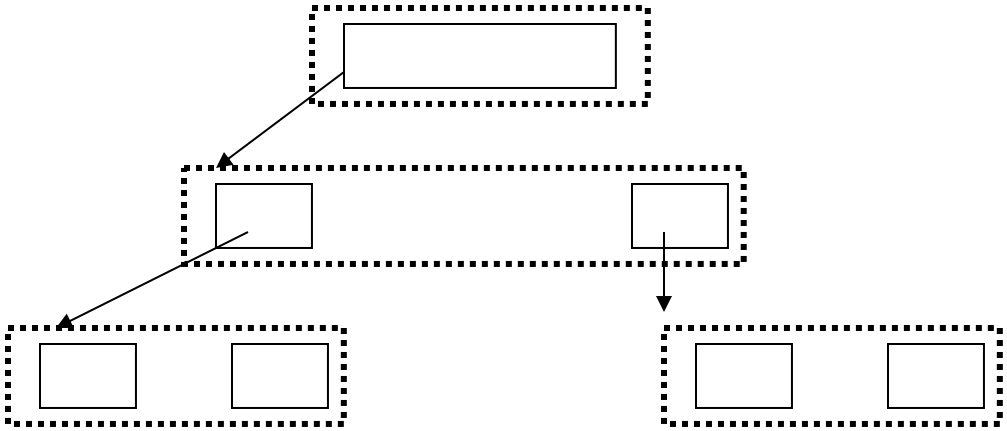
Problem?

No dynamic update because fan-out and array size must be fixed.

With Update - Restore Some Pointers

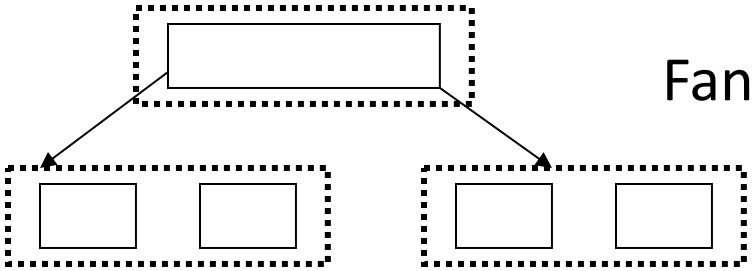
CSB+ tree

- Children of the same node stored in an array (node group)
- Parent node with only a pointer to the child array.
- Similar search performance as CSS tree.
- Good update performance if no split.



Other Variants

- CSB+ tree with segments
  - Divide child array into segments (usually 2)
  - With one child pointer per segment
  - Better split performance, but worse search.
- Full CSB+ tree
  - CSB+ tree with pre-allocated children array.
  - Good for both search and insertion. But more space.



2-segment CSB+ tree.

Fan-out drops by 2*2.

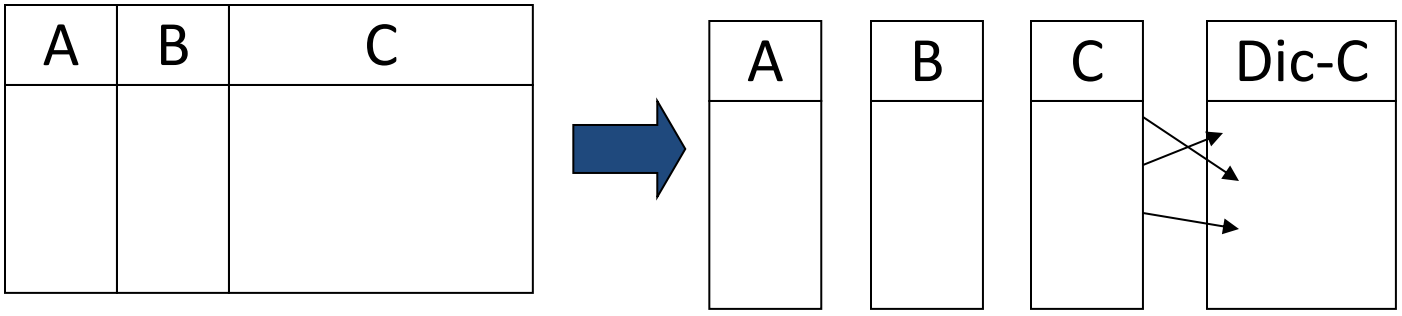
Performance

- Performance:
  - Search: CSS < full CSB+ ~ CSB+ < CSB+ seg < B+
  - Insertion: B+ ~= full CSB+ < CSB+ seg < CSB+ < CSS
- Conclusion:
  - Full CSB+ wins if space not a concern.
  - CSB+ and CSB+ seg win if more reads than insertions.
  - CSS best for read-only environment.

Cache Conscious Join Method

Vertical Decomposed Storage

- Divide a base table into m arrays (m as #of attributes)
- Each array stores the <oid, value> pairs for the i'th attribute.
- Variable length fields to fixed length via dictionary compression.
- Omit oid if oid is dense and ascending.
- Reconstruction is cheap – just an array access.



Existing Equal-Join Methods

- Sort-merge:
  - Bad since usually one of the relation will not fit in cache.

If the two data sets in the join are not already sorted, then the database sorts them. These are the SORT JOIN operations. For each row in the first data set, the database probes the second data set for matching rows and joins them, basing its start position on the match made in the previous iteration. This is the MERGE JOIN operation.

- Hash Join:
  - Bad if inner relation can not fit in cache.

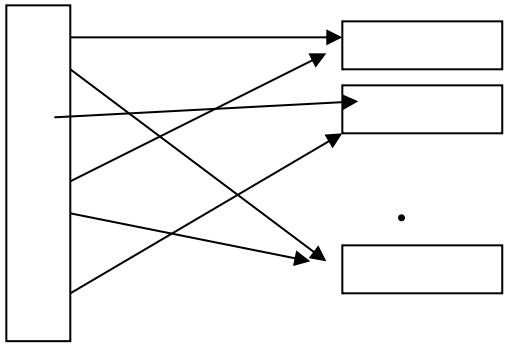
The optimizer uses the smaller of two data sets to build a hash table on the join key in memory, using a deterministic hash function to specify the location in the hash table in which to store each row. The database then scans the larger data set, probing the hash table to find the rows that meet the join condition.

!!! Only useful if one of the relation/dataset can fully stored in the memory

- Clustered hash join:
  - One pass to generate cache sized partitions.
  - Bad if #of partitions exceeds #cache lines or TLB entries.

Two-level Hash Join

Generate 100 cache line partitions. However, if # partitions exceeds the available cache lines,

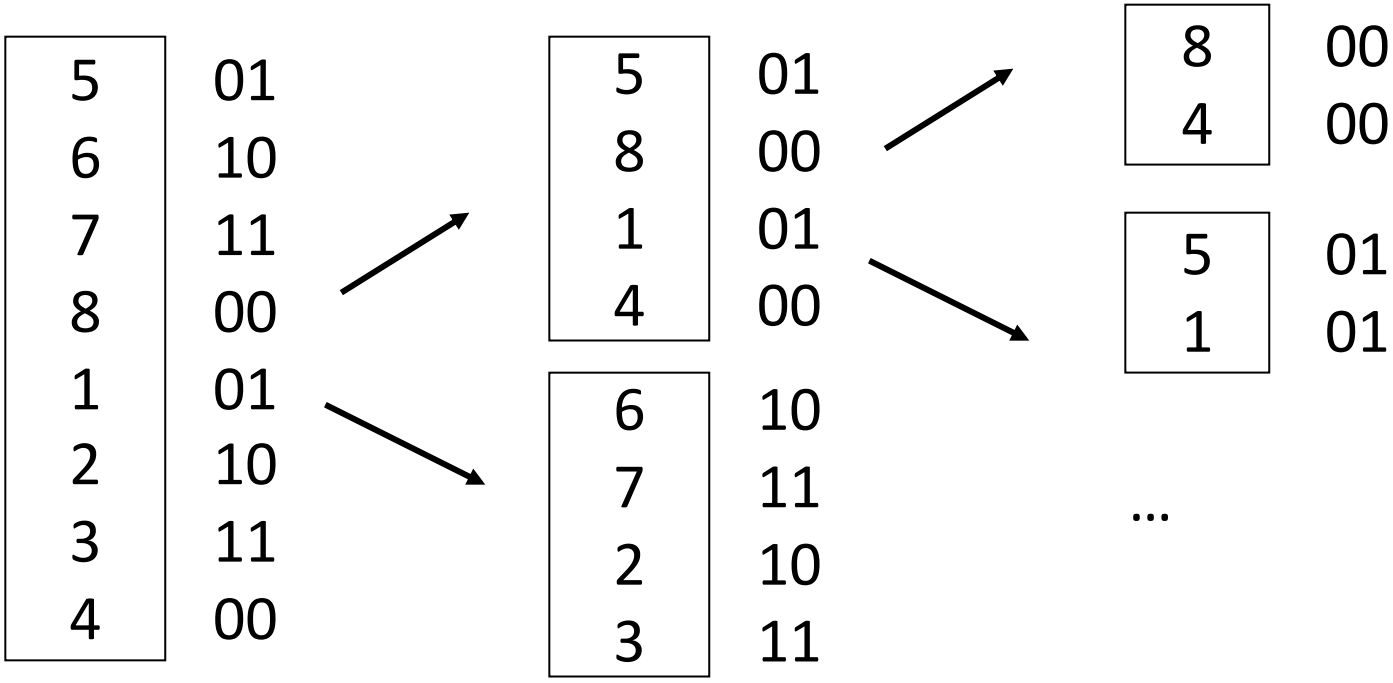


Cache (TLB) thrashing occurs – one miss per tuple

Radix Join (1)

Multi passes of partition:

- The fan-out of each pass does not exceed #of cache lines AND TLB entries.
- Partition based on B bits of the join attribute



Radix Join (2)

- Join matching partitions
  - Nested-loop for small partitions (<= 8 tuples)
  - Hash join for larger partitions
    - <= L1 cache, L2 cache or TLB size.
    - Best performance for L1 cache size (smallest).
- Cache and TLB thrashing avoided.
- Beat conventional join methods
  - Saving on cache misses > extra partition cost

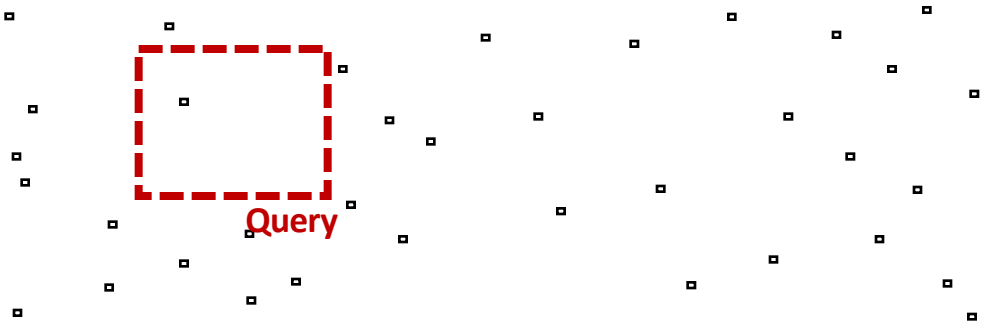
Lessons

- Cache performance important, and becoming more important
- Improve locality
  - Cluster data into a cache line
  - Leave out irrelevant data (pointer elimination, vertical decomposition)
  - Partition to avoid cache and TLB thrashing
- Must make code efficient to observe improvement
  - Only the cost of what we consider will improve
  - Be careful to: functional calls, arithmetic, memory allocation, memory copy

Example – Spatial Data

Spatial Data & Queries

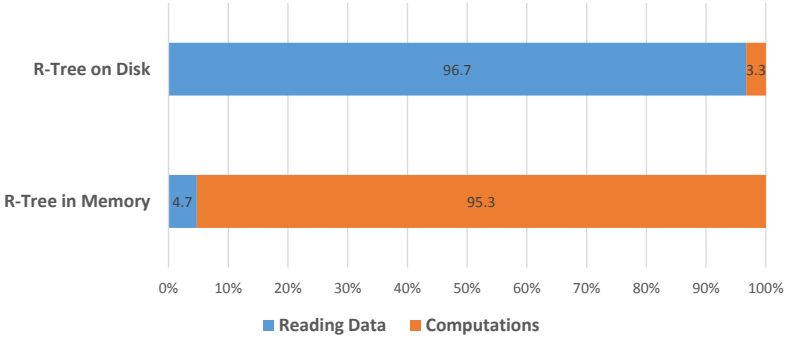
- Any data with three dimensions, e.g., points
- Different queries: range query, nearest neighbor etc.



- Store objects near each other on the same disk page (or cache line)

Adjacency can only be clearly defined in 1D space

- Time spent on computation becomes a consideration

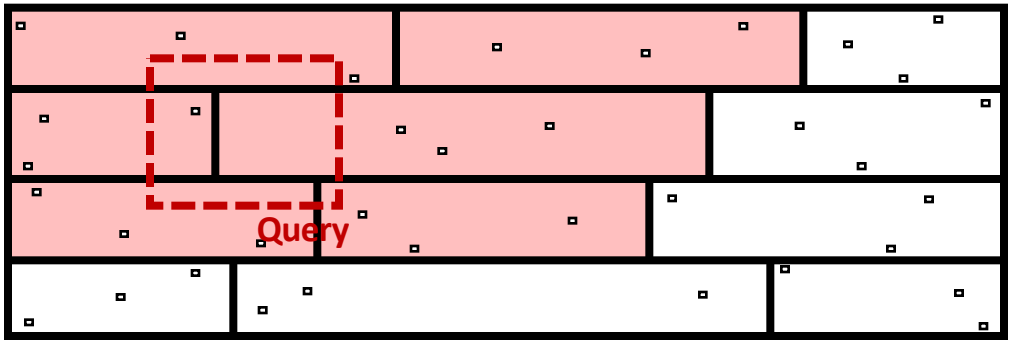


R-Tree: Extended B-Tree for multi-dimensions
Memory R-Tree: Focus on improving Computation Efficiency

Reduce Computation

- Traditionally non-uniform partitioning

Need to search for 5 partitions with far away (spatially) data

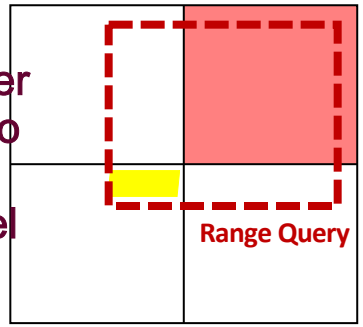


Entries on Disk Page = 3.

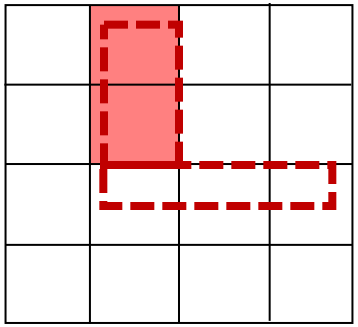
SO spatially speaking, it is non-uniform by the content is identical

- Reduce computations by using several grids:

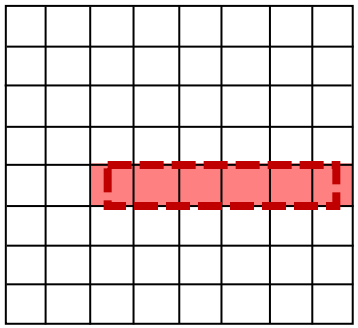
When smaller than 10%, go for the more detailed level



Coarsest-grained Grid



Level of Detail

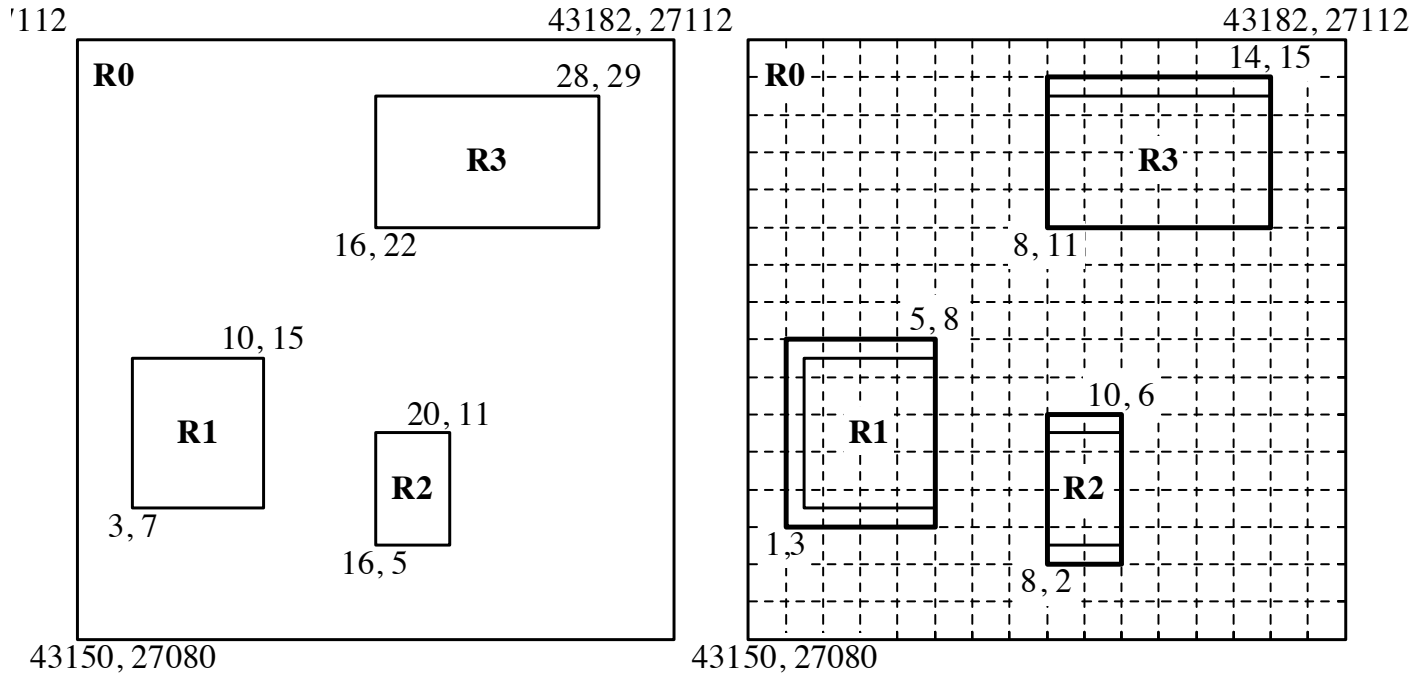


Finest-grained Grid

Store spatial points at all levels (i.e., Coarsest -> Finest) Divide the range into different segments, but search in different levels

Compressing Spatial Data

Lossy



(b) Relative coordinates of R1~R3 to the lower left corner of R0

(c) Quantized relative coordinates

Drop precision => Make the range slightly bigger => Larger spatial object bigger => Smaller representation of data in memory => Fit more in memory