Hardware Trends

CPU speed and memory capacity double every 18 months.

Memory performance merely grows 10%/yr:

- Capacity vs speed (esp. latency)

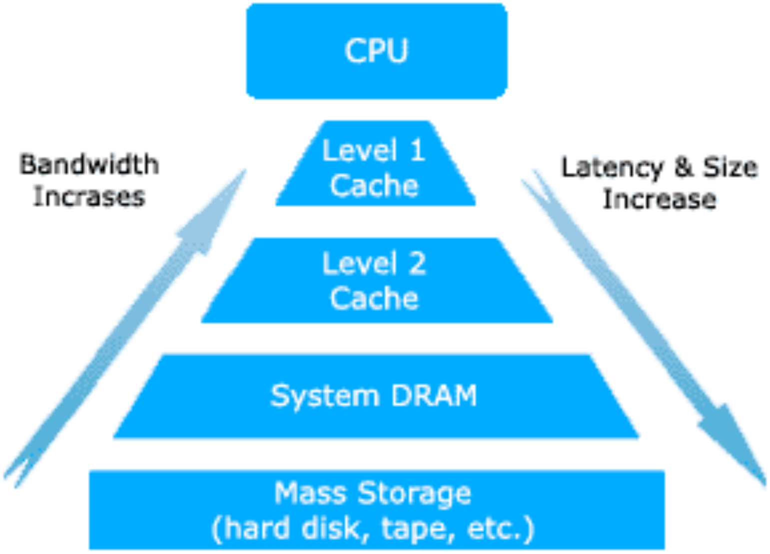
The gap grows ten fold every 6 yr! And 100 times since 1986.

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

Memory Basics

- 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.



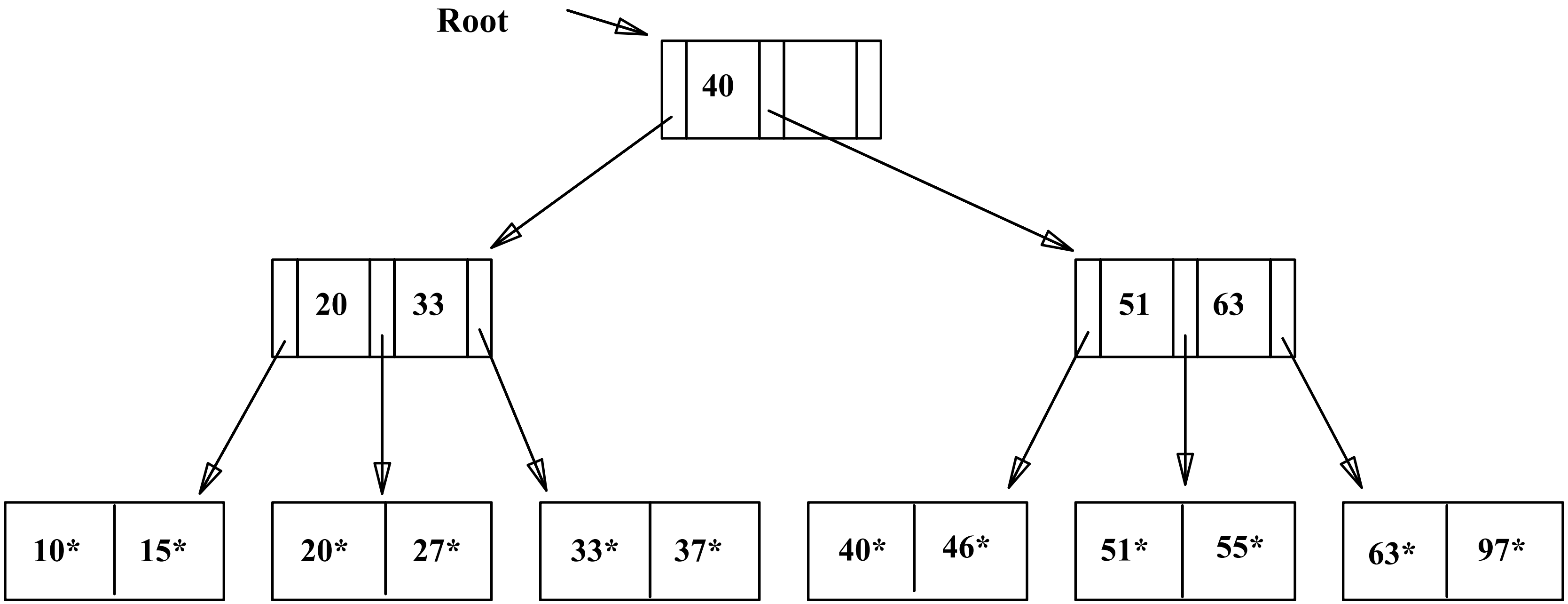
Improving Cache Behavior

- 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

Cache Conscious Indexing

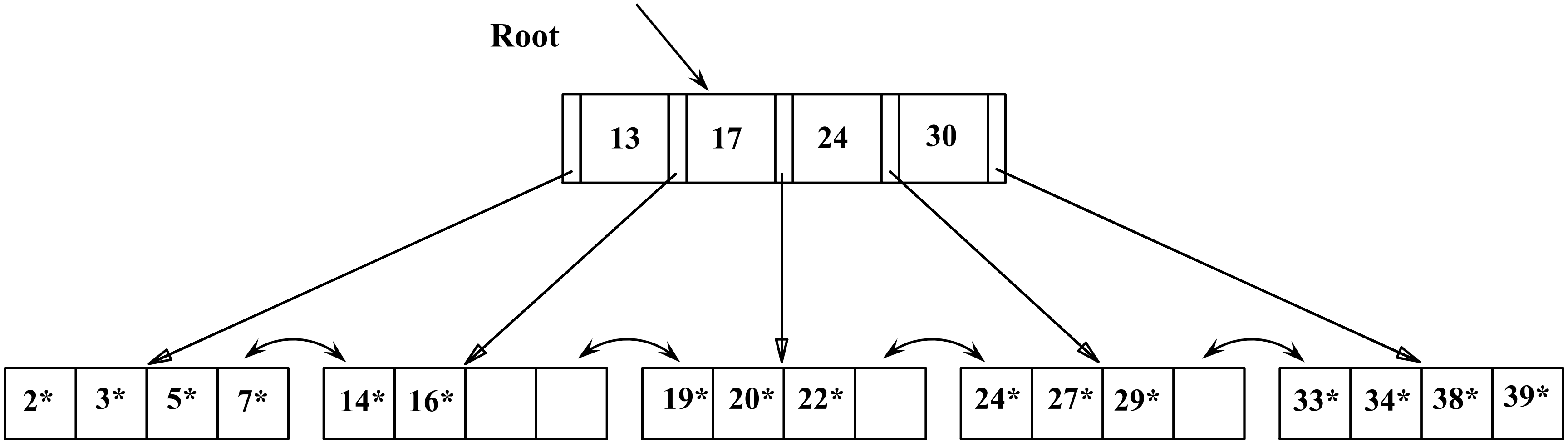
Example Tree Index

- *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* ...



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

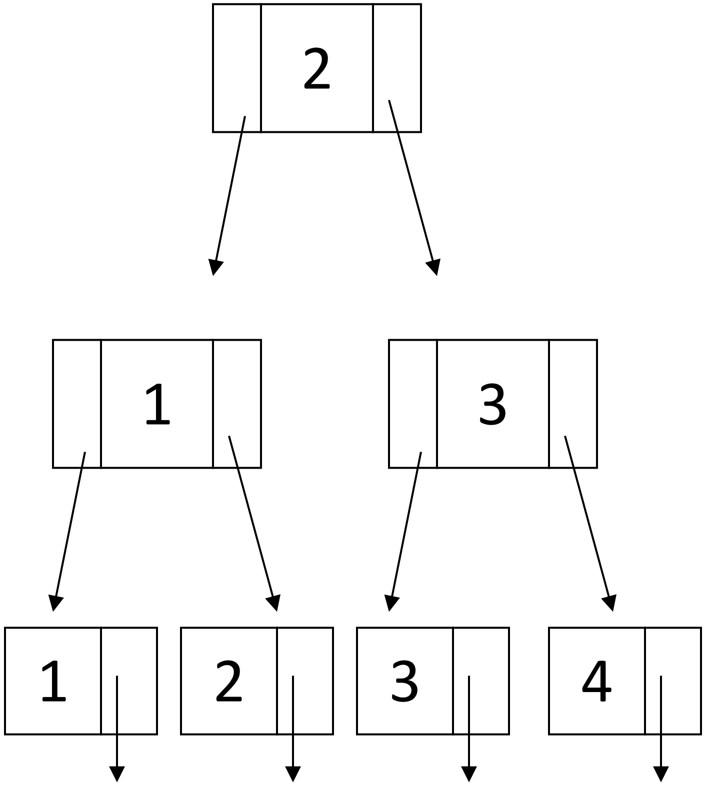
Cache Sensitive Search Tree

- 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

B+ tree, 2-way, 3 misses



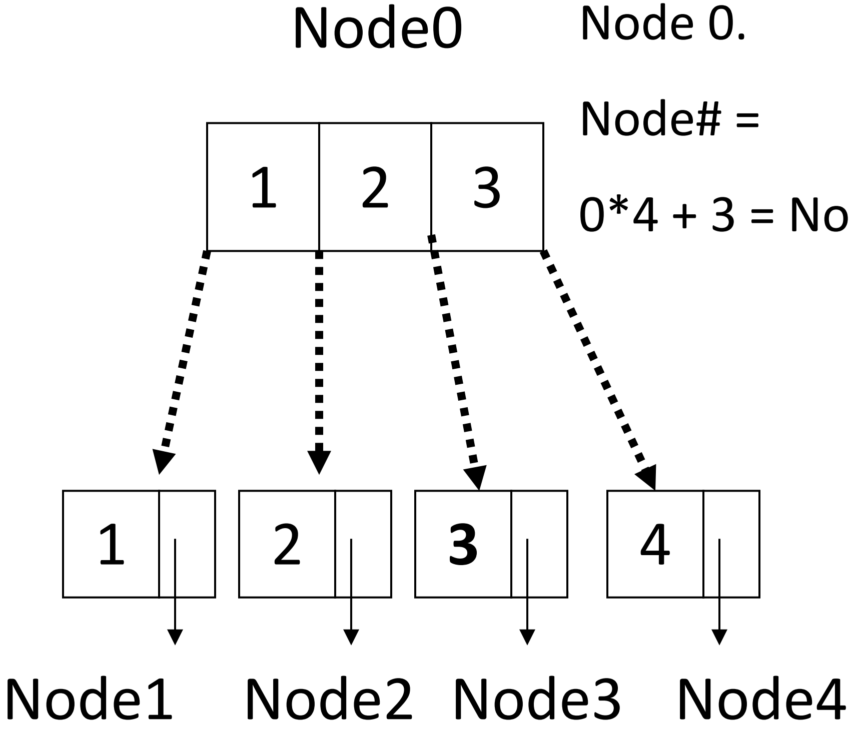
CSS tree, 4-way, 2 misses

Search K = 3,

Match 3rd key in Node 0.

Node# =

0*4 + 3 = Node 3



Performance Analysis (1)

Node size = cache line size is optimal:

- Node size as S cache lines.
- Misses within a node = $1 + \log_2 S$
  - Miss occurs when the binary search distance >= 1 cache line
- Total misses = $\log_m n * (1 + \log_2 S)$
- $m = S * c$ (c as #of keys per cache line, constant)
- Total misses = A / B where:
  - $A = \log_2 n * (1+\log_2 S)$
  - $B = \log_2 S + \log_2 c$
- As $\log_2 S$ increases by one, A increases by $\log_2 n$, B by 1.
- So minimal as $\log_2 S = 0$, $S = 1$

Performance Analysis (2)

- 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)

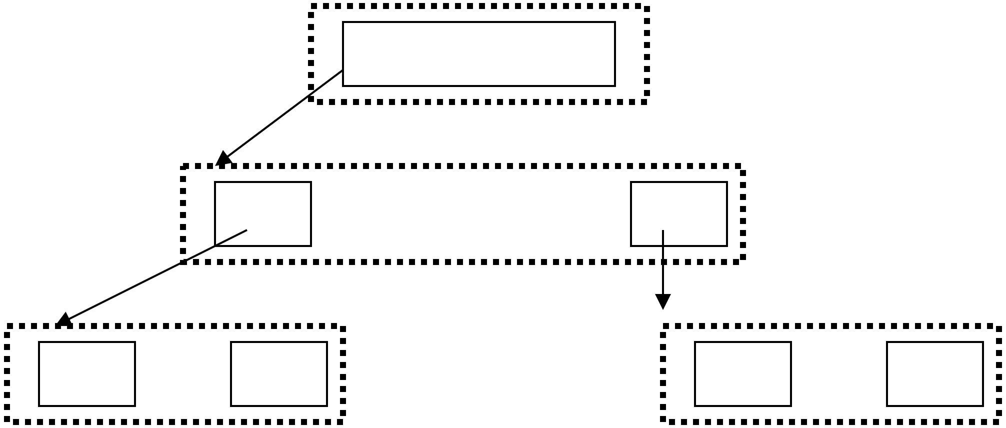
Problem?

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

With Update - Restore Some Pointers

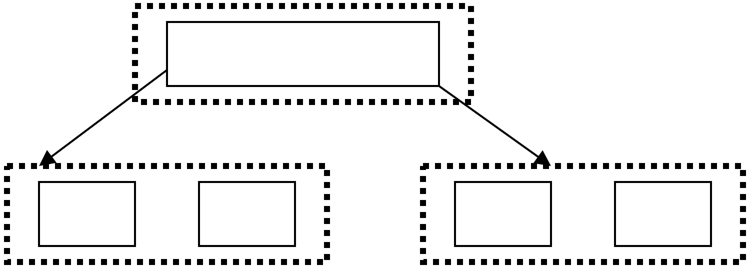
CSB+ tree

- 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. (m decreases by 2)
- 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.

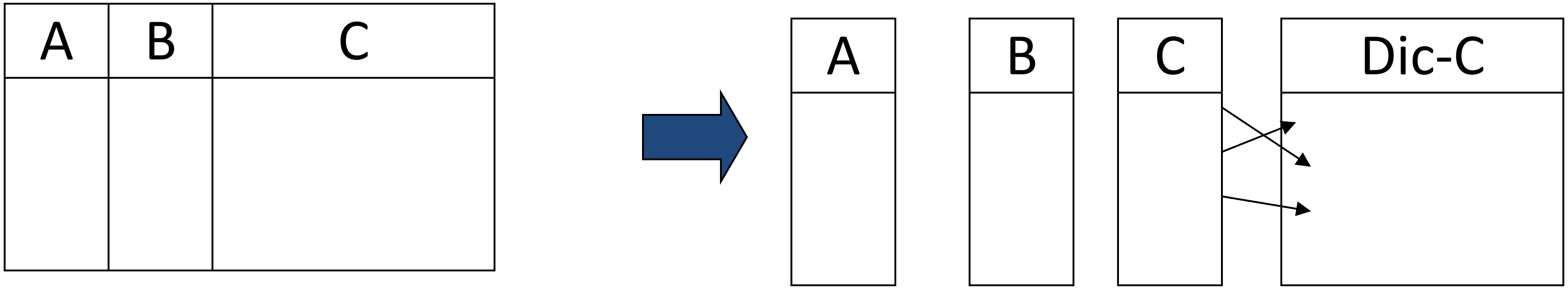
Performance

- 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

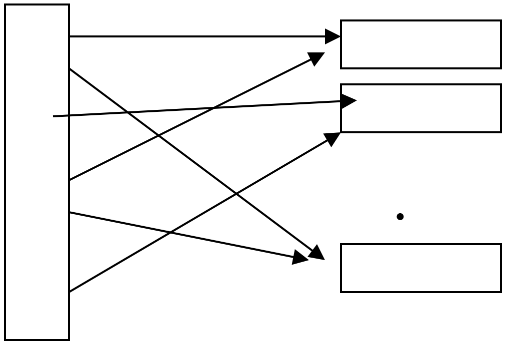
Vertical Decomposed Storage

- 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.
- Hash Join:
  - Bad if inner relation can not fit in cache.
- Clustered hash join:
  - One pass to generate cache sized partitions.
  - Bad if #of partitions exceeds #cache lines or TLB entries.

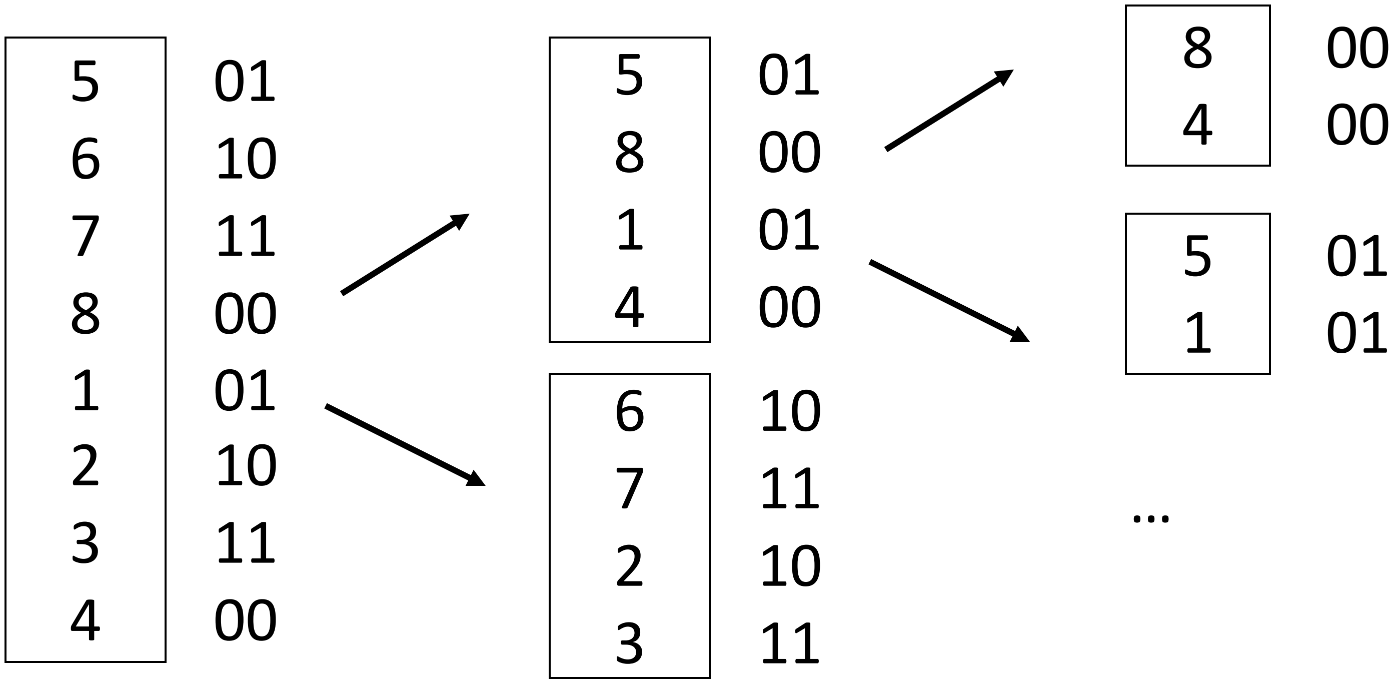


Cache (TLB) thrashing occurs – one miss per tuple

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 (low computational cost)



5	01
6	10
7	11
8	00
1	01
2	10
3	11
4	00

5	01
8	00
1	01
4	00

6	10
7	11
2	10
3	11

8	00
4	00

5	01
1	01

…

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

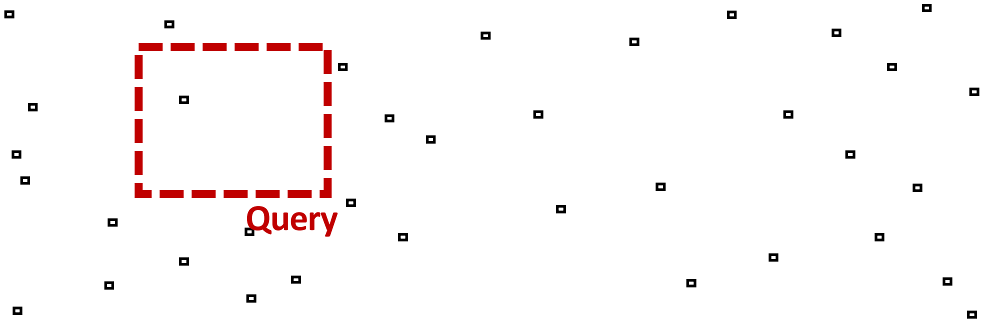
Lessons

- Cache performance important, and becoming more important
- Improve locality
  - Clustering 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.

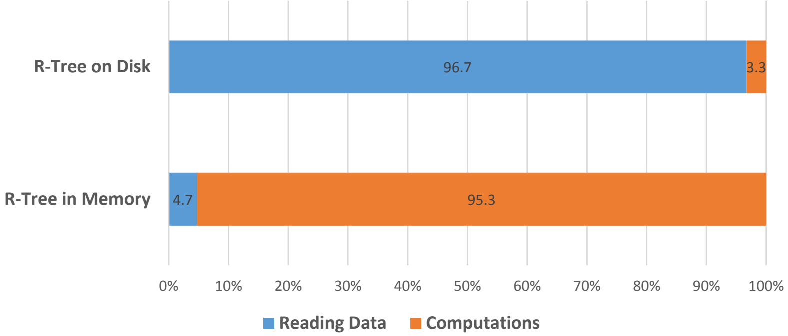
Example – Spatial Data

Spatial Data & Queries

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

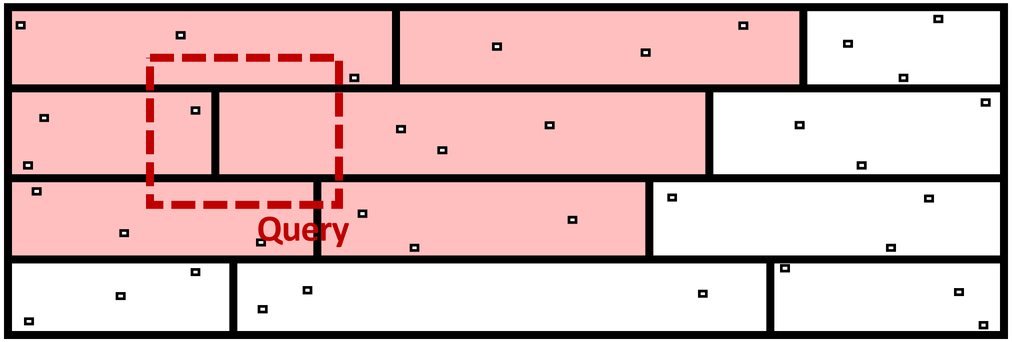


- Store objects near each other on the same disk page (or cache line)
- Time spent on computation becomes a consideration:

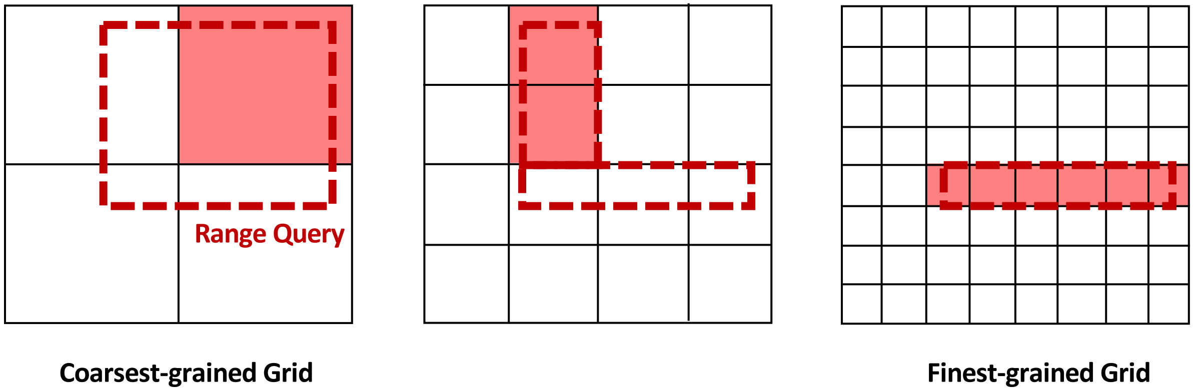


Reduce Computation

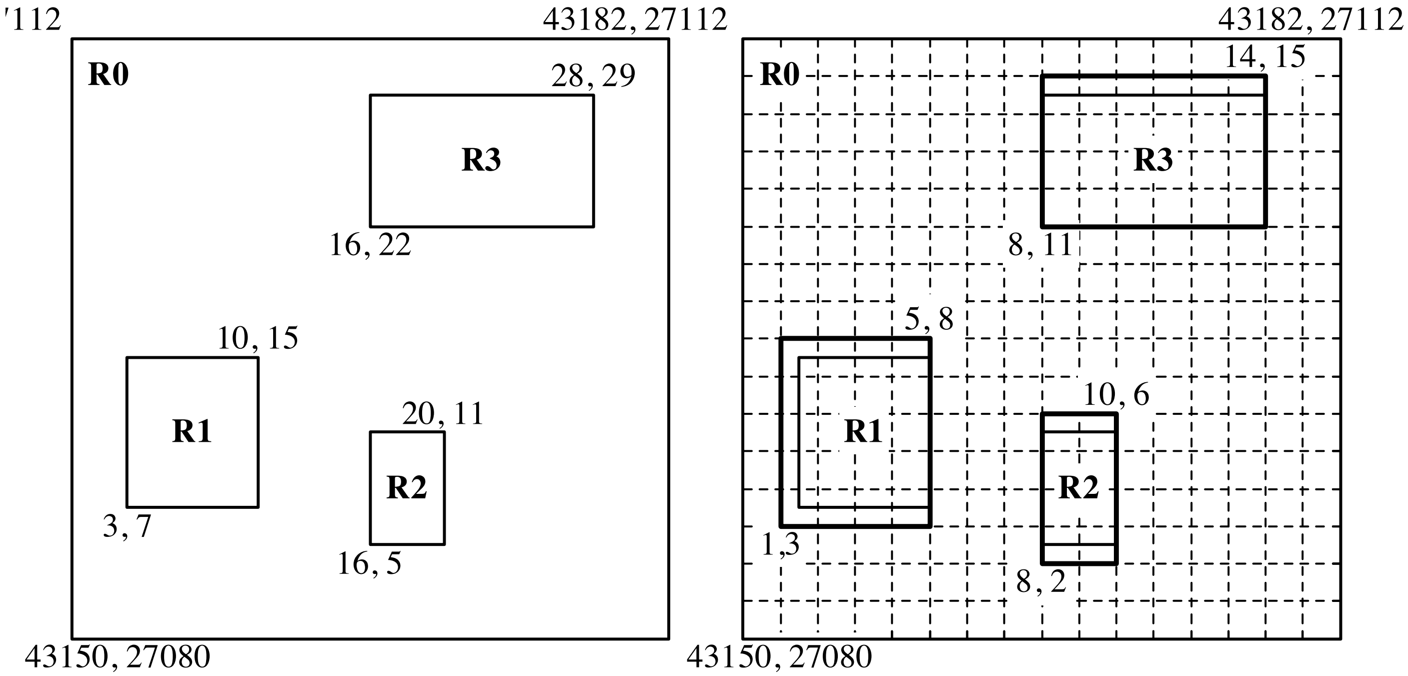
- Traditionally non-uniform partitioning



- Reduce computations by using several grids:



Compressing Spatial Data



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

(c) Quantized relative coordinates