Seminar on
Approaches for improving Cache Line Utilization in Database Systems

Kamlesh Ladhhad (05329014)
Unmesh Deshmukh (05329012)

March 14, 2006
Motivation

- Growing need for efficient cache memory utilization in Modern Database Systems
- Different Approaches
  - Cache conscious index structures
  - New layout for data records
  - Explicit buffering of operators at specific points
Growing need for efficient cache utilization

- CPU speeds have been increasing at a much faster rate than memory speeds
- Conclusion: improving cache behavior is going to be an imperative task in main memory data processing
Cache Memories

- Small, fast SRAM memories that improve performance by holding recently referenced data
- Memory reference: Cache Hit, Cache Miss
- Parameters:
  - Capacity
  - Block size (cache line)
  - Associativity

<table>
<thead>
<tr>
<th>Type of Memory</th>
<th>Typical Size</th>
<th>Typical Speed (latency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>32 * 4 Bytes</td>
<td>CPU Speed (&lt; 2 ns)</td>
</tr>
<tr>
<td>Level 1 Cache</td>
<td>&lt; 64 KBytes</td>
<td>CPU speed (&lt; 2 ns)</td>
</tr>
<tr>
<td>Level 2 Cache</td>
<td>&lt; 1 MegaByte</td>
<td>5 - 20 cycles</td>
</tr>
<tr>
<td>Main Memory Disk</td>
<td>&lt; 1 Gigabyte</td>
<td>10 - 100 cycles</td>
</tr>
<tr>
<td></td>
<td>10 Gigabytes</td>
<td>10,000,000 cycles</td>
</tr>
</tbody>
</table>
Cache conscious index structures

- **Cache Sensitive Search (CSS) trees**
  - Each node contains only keys and no pointers
  - Nodes are stored level by level from left to right
  - Arithmetic operations on offsets to find child nodes
  - Better Search Performance and Cache line utilization than $B^+$-Trees
  - Incremental updates difficult so suitable for DSS workloads only
Comparison between $B^+$-Tree and CSS Tree

- Cache line size = 12 bytes, Key size = Pointer size = 4 bytes
- Search key = 3
Cache Sensitive $B^+\text{-Tree}$

- **Goal**
  - Retain good cache behaviour of CSS-Trees while at the same time being able to support incremental updates
  - This way it will be useful even for non-DSS workloads

- **Idea**
  - Use Partial Pointer Elimination Technique
  - Have fewer pointers per node than a $B^+\text{-Tree}$ so more space for keys
  - Use limited amount of arithmetic on offsets to compensate for less number of pointers

- **Structure**
  - Put all child nodes of a given node in a *Node Group*
  - Store nodes within a node group contiguously and use offset arithmetic for access
B⁺-Tree Vs CSB⁺-Tree

- Cache line size = Node size = 64 bytes
- Key and child pointer each occupy 4 bytes
- Keys per node for B⁺-Tree = 7
- Keys per node for CSB⁺-Tree = 14
- In CSB⁺-Trees, number of cache lines to be searched are fewer
workloads. We describe another variant of CSB Trees that further reduces split cost in Section 3.3.

A CSB tuned to obtain good performance under particular child pointer explicitly, it can store more keys per node than a B-tree in a CSB. Each leaf node stores a list of keys, where the number of these pairs, and two sibling pointers. A version of CSB handles updates. However, a CSB CSB structures. Our tree structure, which we call a...
Operations on CSB\(^+\)-Tree

- **Bulkload**
  - Allocate space for leaf entries
  - Calculate how many nodes are needed at higher level and allocate them contiguously
  - Fill in the entries at higher level appropriately and set first child pointers
  - Continue with the same process until only one node remains i.e, root

- **Search**
  - Similar to B\(^+\)-Tree search algorithm
  - Locate rightmost key K in the node that is smaller than the search key and add the offset of K to the first child pointer to get the address of the child node
Operations on $\text{CSB}^+$-Tree ..contd.

- **Insertion**
  - Again similar to $\text{B}^+$-Tree insertion algorithm
  - Pseudo-code
  - Search the leaf node $n$ to insert the entry
  - If $n$ is not full then insert the new entry into the appropriate place
  - Otherwise split $n$. Let $p$ be the parent node of $n$, $f$ be the first child pointer in $p$ and $g$ be the node group pointed to by $f$.
    - If $p$ is not full then copy $g$ to $g'$ in which $n$ is split in two nodes. Let $f$ point to $g'$
    - If $p$ is full copy half of $g$ to $g'$. Let $f$ point to $g'$. Split the node group of $p$ according to as above
Insertion example

key = 34

22

7 30

3 13 19

2 | 3 5 | 7 12 | 13 16 | 19 20 | 22

25 33

24 | 25 27 | 30 31 | 33 | 36 | 39

a CSB+-Tree of Order 1
Insertion example

key = 34
Operations on $CSB^+$-Tree ..contd.

- **Deletion**
  - Handled in a way similar to insertion
  - Lazy deletion - Locate the data entry, remove it but don’t restructure the tree
Segmented CSB$^+$-Tree

- **Problem:** Increase in maximum size of the node group due to increase in cache line size means more copying of data in case of split
- **Solution:** Divide the child nodes into segments, store in each node pointers to segments and only child nodes in the same segment are stored contiguously
Segmented CSB\(^+-\) Tree

- Tree of order 2 with 2 segments
Variants of SCSB$^+$-Tree

- Two variants of SCSB$^+$-Tree:
  - Fixed Size Segments
    - Start by filling the nodes in the first segment till it is full
    - Then fill the nodes in second segment, this requires copying nodes in this segment only
  - Varying Size Segments
    - For bulkload, distribute nodes evenly among the segments
    - On every new node insertion, create a new segment for the segment to which the new node belongs
    - Touches only one segment in each insert as opposed to the fixed size variant
Full CSB$^+$-Tree

- Higher frequency of memory allocation and deallocation calls in CSB$^+$-Trees is a problem
- Another approach is to pre-allocate memory for entire node group
- Space-time tradeoff:
  - Node split in Full CSB$^+$-Tree is efficient than normal CSB$^+$-Tree
  - This efficiency comes at the expense of pre-allocated space
Implementation details

- Node size = Cache line size = 64 bytes
- Key size = Pointer size = 4 bytes
- For CSS trees: 16 keys per node
- For B⁺-Trees: Internal node 7 keys, 8 child pointers and number of keys used
- For CSB⁺-Trees: Internal node 14 keys, first child pointer and number of keys used
Pure Search Performance Graph

- Time for 200K searches
- $B^+$-Trees are more than 25% slower than CSB$^+$-Tree
Experiments on stabilized index structures-Search

- Segmented CSB$^+$-Tree search slower than CSB$^+$-Tree because: branching factor of former is less (More cache misses), extra comparisons needed to choose right segment.
Experiments on stabilized index structures-Delete

- Because of lazy deletion most of the time is spent in locating the record, so delete performance similar to search.
Experiments on stabilized index structures-Insert

- CSB$^+$-Trees are worse than B$^+$-Trees for insertion because of the split cost
- SCSB$^+$-Trees reduce split cost so give intermediate performance
- B$^+$-Trees have to allocate a new node on every split while Full CSB$^+$-Trees make allocation when node group is full.
Conclusion

- Full CSB$^+$-Trees are better than B$^+$-Trees in all aspects except for space.
- In limited space environment CSB$^+$-Trees and Segmented CSB$^+$-Trees provide faster searches while still being able to support incremental updates.
- Suitable for applications like Digital libraries, Online shopping- Searching much more frequent than updates.
Weaving Relations for Cache Performance
Motivation for devising new data layout model

- Main Problem Being Addressed: **Only a fraction of data transferred to cache is useful for the query**
- Ill-effects caused by the problem:
  - Wastage of bandwidth
  - Polluting the cache
  - May result in replacing useful information
An illustrative example

- Most widely used N-ary Storage Model (NSM) stores relation’s records sequentially in slotted disk pages
- Sample Query:
  - select name from R where age < 40
- Relation R contains three attributes SSN, Name and Age
- For the above query the NSM model has inferior cache performance that is shown in the next slide
NSM cache behaviour

**Figure 1:** The cache behavior of NSM.
DSM Example

sub-relation R1

sub-relation R2

sub-relation R3
Decomposition Storage Model (DSM)

- Fully decomposed form of Vertical Partitioning
- Partitions an n-attribute relation into n sub-relations
- Each sub-relation contains two attributes: a logical record id and the attribute value
- Sub-relations are stored as regular relations in slotted pages
- Advantages:
  - High degree of spatial locality for sequential access of an attribute
  - Better I/O and Cache performance
- Disadvantage:
  - Performance significantly deteriorates for queries involving multiple attributes for each participating relation
Partition Attributes Across (PAX)

- Idea is to keep the attribute values of each record on the same page as in NSM while using a cache-friendly algorithm for placing them inside the page.
- Vertically partition records within page, storing together values of each attribute in a minipage.
- Advantages:
  - maximizes inter-record spatial locality thus improving cache performance
  - minimal record reconstruction cost
  - orthogonal to other design decisions as it affects only the data within a page.
- The following slide shows the cache behaviour of PAX.
Cache Behaviour of PAX

**Figure 3:** The cache behavior of PAX.
An example PAX page

Figure 4: An example PAX page.
Design of Page in PAX

- For storing a relation of degree $n$, PAX partitions the page into $n$ minipages.
- Page Header contains pointers to the beginning of each minipage, number of attributes, the attribute sizes, current number of records on the page and free space available.
- Fixed length attributes are stored in F-minipages. The end of F-minipage has presence bit vector.
- Variable length attributes are stored in V-minipages. These are slotted with pointers to the end of each value.
Data Manipulation Algorithms

- **Bulk-loading and Insertions**
  - Allocate each minipage on the page based on attribute value size
  - Inserts records by copying actual value to each minipage
  - When variable length values are present, minipage boundaries need to be adjusted to accommodate records as they are inserted in the page
  - PAX calculates the position of each attribute value of the page, stores the value and updates the bitmaps and offset arrays appropriately

- **Updations**
  - Find the position of the attribute value of the record and then update the value
  - Updates to variable length values may require minipage level reorganizations
  - If the space is not sufficient to accommodate and re-organization is not possible then record is moved to other page
Data Manipulation Algorithms...contd.

- Deletion
  - NSM uses slot array to mark an entry as deleted
  - PAX keeps track of deleted records using a bitmap at the start of the page and uses bitwise calculations to find whether a record is deleted
  - Reorganization can be done within minipage after deletion so as to minimize fragmentation
  - For deletion intensive workloads, reorganization can be deferred.
Experimental Results-1

**NSM/PAX/DSM Elapsed Time**

- **NSM**
- **PAX**
- **DSM**

**Elapsed time (seconds)**

**Number of attributes in query**

- 1
- 2
- 3
- 4
- 5
- 6
- 7

The graph shows the elapsed time in seconds for different numbers of attributes in a query for NSM, PAX, and DSM. The elapsed time increases with the number of attributes for all three methods, but DSM shows a more significant increase compared to NSM and PAX.
Experimental Results-2

- **NSM Vs PAX Impact on cache behaviour**
  - PAX reduces data penalty at both cache levels L1 and L2 and reduces stall time
  - This reduction in number of misses results in further reduction of instruction cache misses as cache space is judiciously used

**Figure 7:** PAX impact on memory stalls
Experimental Results-3

NSM/PAX Sensitivity Analysis

- Query execution time of NSM and PAX converge as the number of projected attributes increase.
- As the degree of relation increases other factors such as buffer manager start to play a dominant role.
Buffering Database Operations for Enhanced Instruction Cache Performance
A typical scenario

- In a demand-driven query execution plan child operator returns control to parent operator immediately after generating one tuple
- So the operator execution sequence is like 'PCPCPCPCPCPCP..'
- Instruction cache thrashing can occur when the combined size of two operators exceeds the size of the smallest, fastest cache unit
Buffer operator
Solution that uses buffering

- Given a query, add a special buffer operator at certain places between a parent operator/operator group and child operator/operator group.
- Buffer operator above child has an array of pointers that point to intermediate result tuples.
- This effectively changes the execution sequence to 'PCCCCCPPP PPP CCCP P'.
- The execution sequence shows that number of instruction cache misses decrease substantially.
- The reduced cache misses are due to improved instruction spatial and temporal locality.
New Buffer Operator

- Given a query plan identify the execution groups that are candidate units for buffering
- Add a new explicit buffering operator above the execution group, if necessary
- Implementation of buffer operator:
  - Supports open-next-close interface
  - Maintains two states: Whether end-of-tuples is received from the child operator and Whether its buffered tuples have been consumed
  - Maintains an array of pointers to tuples that are stored in child operator’s space
- Benefits of buffer operator:
  - Increase in query throughput due to decrease in instruction cache misses
  - Better hardware branch prediction
Other Details

- All operators don’t benefit from buffering e.g. small cardinality operators, blocking operators like sort
- The placement of buffer operators in a query plan can be done by using a bottom-up pass of the plan tree
- This however needs some mechanism of estimating the memory needed by various query operators
Conclusion

- We looked at three approaches for improving cache performance
- CSB⁺-Tree approach was able to give better search performance while at the same time allowing incremental updates
- PAX approach changed the data layout model to ensure that cache space is occupied by useful data and it also remained orthogonal to other design decisions
- Buffering approach tried to solve the problem of improving instruction cache performance for demand-driven pipelined query execution environment
References

Thank You!