Improving Cache Performance

Namrata Jain (06329014)
under Guidance of
Prof. S. Sudarshan

Kanwal Rekhi Institute of Technology and Science
IIT, Bombay

Outline

Motivation

- Growing need for efficient cache memory utilization in Modern Database System
- Significant amount of execution time is spent on second level (L2) data cache misses and first level (L1) instruction cache misses
- Little research has been done to improve instruction cache performance

Outline

1. Improving Cache Performance
   - Improving Instruction Cache Performance [2]
   - Improving Data Cache Performance [1]

A typical scenario

- In demand-driven query execution engine (open-next-close iterator interface), child operator returns control to its parent operator immediately after generating one tuple
- So, operator execution sequence is like ‘PCPCPCPCPCP..’
- Instruction cache thrashing occur when combined size of the two operators exceeds the size of the smallest, fastest cache unit

Solution: Buffering

- Add a special “buffer” operator in certain places between a parent operator and a child operator.
- Each buffer operator stores a large array of pointers to intermediate tuples, generated by the child operator
- Now operator execution sequence becomes ‘PCCCCCPPPPPCCCCCPPPPP...’
- Advantages :
    - reduce the number of instruction cache misses by up to 80 percent .
    - less overhead
    - increases temporal and spatial instruction locality below the buffer operator.
    - decreases the number of branch mispredictions.

Buffer Operator

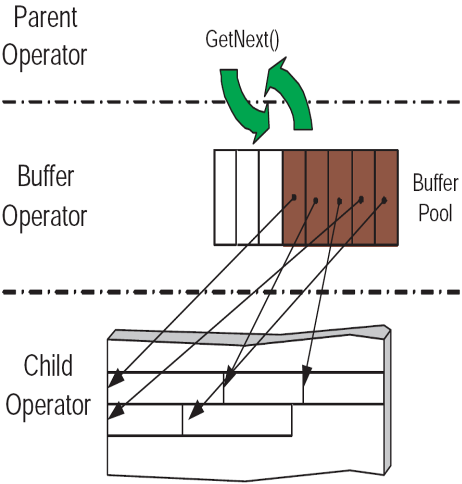


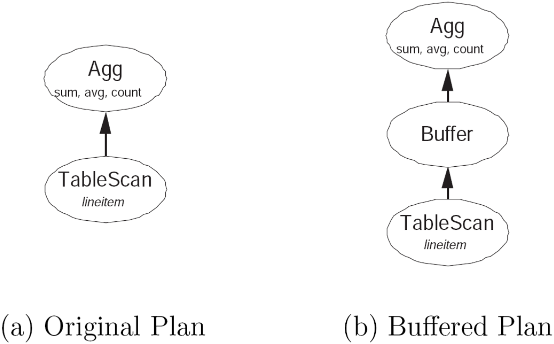
Figure: Buffer Operator

Buffer Operator Example

```
SELECT SUM(l_extendedprice *
           (1 - l_discount) *
           (1 + l_tax)) as sum_charge,
       AVG(l_quantity) as avg_qty,
       COUNT(*) as count_order
FROM lineitem
WHERE l_shipdate <= date '1998-11-01';
```

Figure: Query

Buffer Operator Example



(a) Original Plan (b) Buffered Plan

Figure: Query Execution Plan

Buffer Operator

```
GetNext()
if empty and !end_of_tuples
    then while !full
            do CHILD.GetNext()
               if end_of_tuples
                  then break
                  else store the pointer to the tuple
return the next pointed tuple
```

Figure: Pseudocode for Buffer Operator

When and Where to buffer ?

- Depends on interaction between consecutive operators
- No need to buffer blocking operators like hash-table building and sorting
- Execution group
  - Candidate units of buffering
  - Larger execution group means **less buffering**
  - How to choose execution groups?
- Cardinality
  - Operators with small cardinality estimates are unlikely to benefit from buffering.
  - How to determine cardinality threshold ?

Selecting Execution groups

- The instruction footprint of each execution group combined with the footprint of a new buffer operator should be less than the L1 instruction cache size.
- How to estimate the footprint size ?

How to estimate the footprint size ?

1. The naive way could be to use static call graphs. The instruction footprint of a module is the sum of sizes of all the functions that are called within the module.
    - It gives an overestimate of the size.
2. The ideal footprint estimate can only be measured by actually running the query. But it would be too expensive.
    - In postgres, it was observed that execution paths are usually data independent.
    - Study the dynamic call graphs for different modules, by running a small query set that covers all kinds of operators.
    - While combining footprints of instruction, count common functions only once.

How to determine cardinality threshold ?

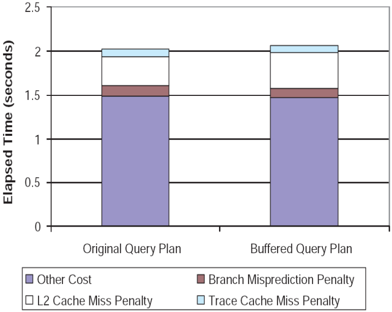
- Using a calibration experiment
  - Running a single query with and without buffering at various cardinalities.
  - Cardinality at which buffered plan begins to beat unbuffered plan would be the cardinality threshold.

Plan Refinement Algorithm

```
Input : Query plan tree
Output : Enhanced plan tree with buffer operator added.
Assumptions : All operators are non blocking with output
cardinality exceeding the calibration threshold.
// Perform a bottom up pass
for each leaf operator do
    Add an execution group including the leaf operator
end for
while Not Root do
    Enlarge each execution group by including parent operators or
    merging adjacent execution groups.
    if Footprint(Execution Group) > L1 instruction cache then
        Finish current execution group.
        Label parent operator as a new execution group.
    end if
end while
for each execution group do
    Add a buffer operator above it.
end for
```

Validating Buffer Strategies

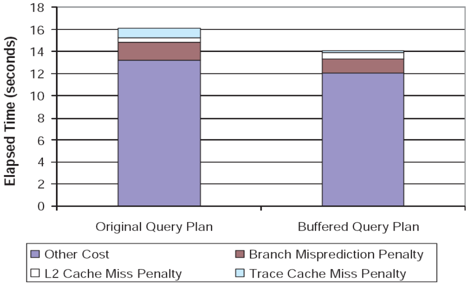
```
SELECT COUNT(*) as count_order
FROM lineitem
WHERE l_shipdate <= date '1998-11-01';
```



The combined footprint is slightly less than the size of the L1 instruction cache.

Validating Buffer Strategies

```
SELECT SUM(l_extendedprice *
           (1 - l_discount) *
           (1 + l_tax)) as sum_charge,
       AVG(l_quantity) as avg_qty,
       COUNT(*) as count_order
FROM lineitem
WHERE l_shipdate <= date '1998-11-01';
```



The combined footprint is more than the size of the L1 instruction cache.

Cardinality Effects

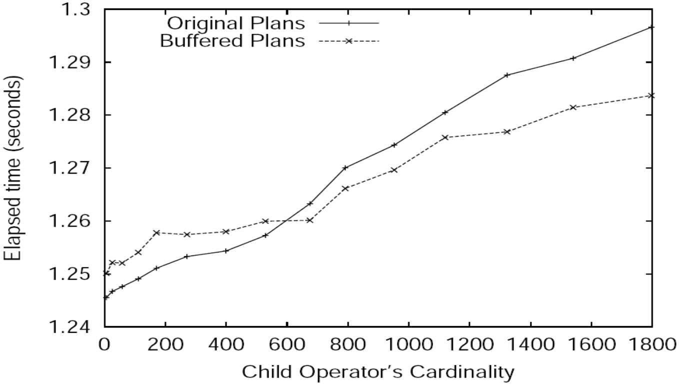


Figure: Cardinality Effects

The benefits of buffering become more obvious as the predicate become less selective. (Cardinality threshold = 600)

Buffer Size

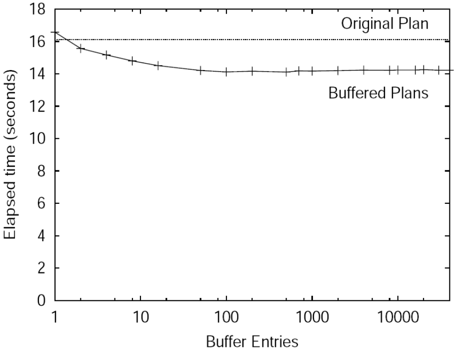
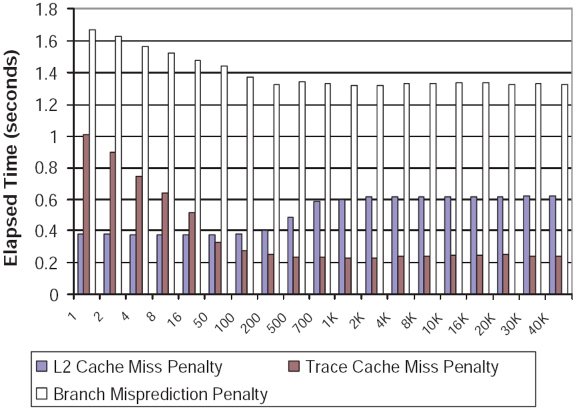


Figure: Varied Buffer Sizes

Buffer Size



The instruction cache miss penalty drops as the buffer size increases.

Buffer operators incur more L2 data cache misses with large buffer sizes.

Conclusion

- To reduce instruction cache thrashing, buffering of intermediate results is done
- Buffering exploits instruction cache spatial and temporal locality
- Buffer operators are especially useful for complex queries, that have large instruction footprints and large output cardinality.

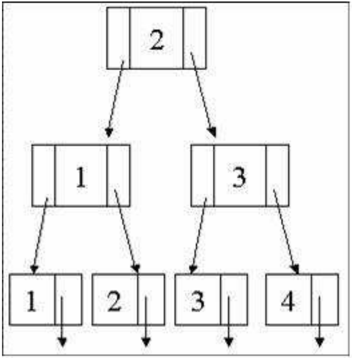
Outline

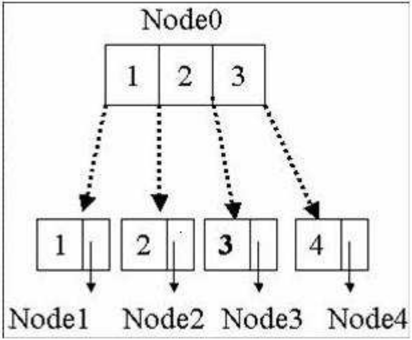
1. Improving Cache Performance
    - Improving Instruction Cache Performance [2]
    - Improving Data Cache Performance [1]

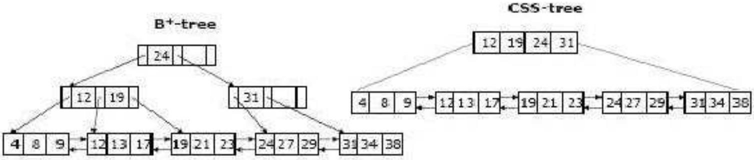
Reducing Data Cache Misses

- Introducing new cache conscious index structure
- Making B^+-Trees cache conscious in main memory

Comparison between B^+-Tree and CSS Tree







Comparison between B^+-Tree and CSS Tree

- 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
- B^+-Trees
  - Each node has keys as well as pointers
  - Good incremental update performance
  - Search performance and Cache line utilization inferior as compared to CSS trees

Pointer elimination is an important technique in improving cache line utilization

Cache Sensitive B^+-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^+-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

Example CSB$^+$-Tree

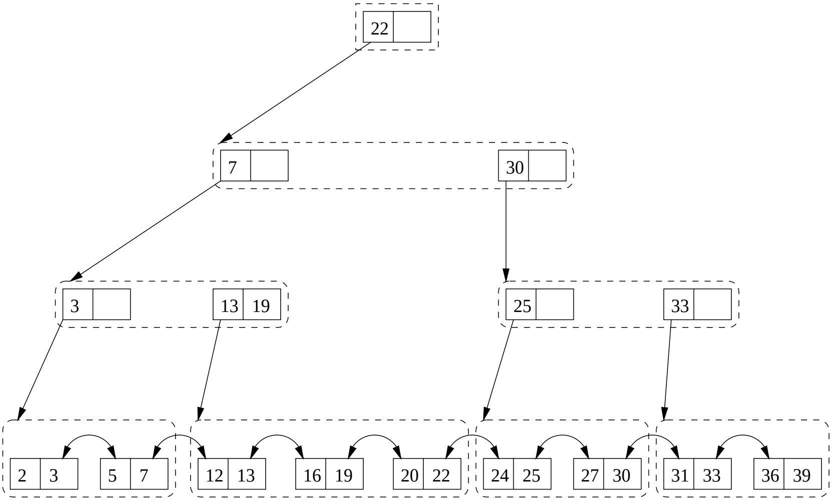


Figure 2: A CSB$^+$-Tree of Order 1

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^+-Tree, number of cache lines to be searched are fewer

Operations on CSB^{+}-Tree

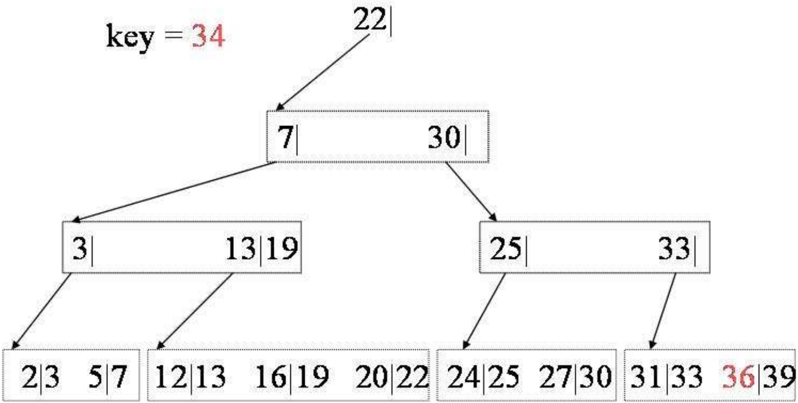
- Bulkload
  - Allocate space for leaf entries
  - Calculate how many nodes are needed at higher level and allocate them contiguously
  - Fill in the entries at higher level appropriately and set first child pointers
  - Continue with the same process until only one node remains i.e, root
- Search
 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

- Insertion - Pseudo-code

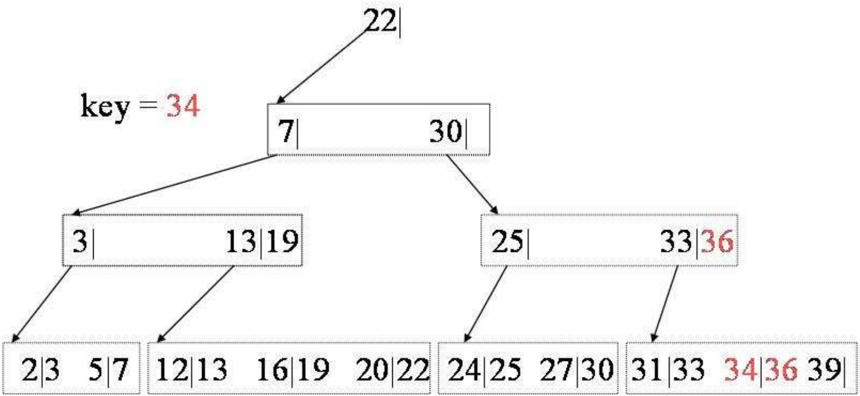
```
Locate the leaf entry by searching the key of new entry
if the leaf entry has enough space then
    Insert the new key into the leaf node
else
    if the parent node has enough space then
        Create a new node group $g'$ having one more node than original node group $g$
        Copy all the nodes from $g$ to $g'$. Split node results in two nodes in $g'$
        Update the first child pointer of parent and deallocate old node group $g$
    else
        Create a new node group $g'$ and evenly distribute nodes between $g$ and $g'$
        Transfer half keys of earlier parent $p$ to a new node $p'$
        Set the first child pointer of $p'$ to $g'$
        The process of recursive split will continue if parent's node group is full
```

Insert example CSB$^+$-Tree



a CSB+-Tree of Order 1

Insert example CSB$^+$-Tree



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
- Optimized Searching within a node
    - Binary Search over keys using conventional while loop
    - Uniform approach
    - Hardwiring all possible optimal search trees and use array of function pointers to view

Segmented CSB$^+$-Tree

- Problem: Increase in maximum size of the node group due to increase in cache line size ⇒ 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

Example SCSB^{+}-Tree

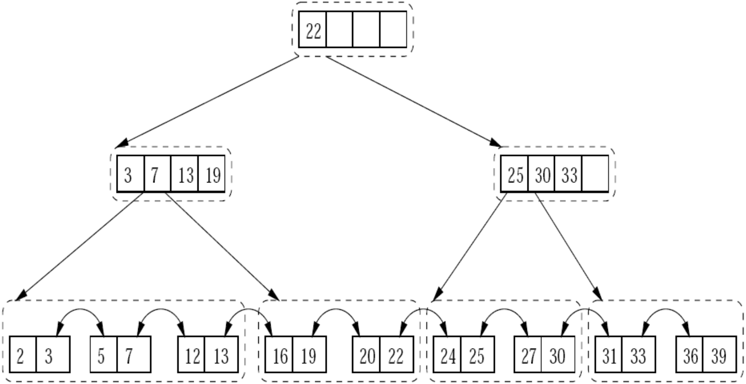


Figure: SCSB^{+}-Tree of order 2 with 2 segments

Segmented CSB$^+$-Tree

- Two variants of CSB$^+$-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

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
- Feature Comparison table:

	B^+	CSB$^+$	SCSB$^+$	Full CSB$^+$
Search	slower	faster	medium	faster
Update	faster	slower	medium	faster
Space	medium	lower	lower	higher
Memory Mgmt.	medium	higher	higher	lower

Bibliography I

Jun Rao and Kenneth A. Ross.
Making B$^+$-trees cache conscious in main memory.
In *SIGMOD Conference*, pages 475–486, 2000.

Jingren Zhou and Kenneth A. Ross.
Buffering database operations for enhanced instruction cache performance.
In *SIGMOD Conference*, pages 191–202, 2004.

Thank You