Implementation of PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

Project Team:
Natasha Saldanha
Karthik Mahadevan

Introduction:

The performance of a database was initially limited by disk I/O bandwidth. However, with the increase in memory capacity entire database tables and their index structures are stored in memory. Additionally, we can achieve high computing power with multiple cores on one chip and also by widening SIMD units on each core.

The use of in-memory databases calls for quick traversals and also requires support for database modifications. The B+ tree is one of the most widely used data structures in databases as it meets the above requirements well. B+ trees have the following advantages:
   a) They efficiently scan sorted data as all the data are stored in the leaves.
   b) The tree structure need not be modified at every insertion or deletion. It is only modified when the order of the leaf exceeds or falls below the threshold.
   c) Time to traverse to the leaves is the same across the entire B+ tree as the tree is balanced and the keys are at the same depth from the root as they are stored in the leaves.

Concurrency control on B+ trees is primarily achieved with latches. However, the use of latches with increasing number of cores poses the following problems:
   i) Poor scaling with increasing number of cores
   ii) Demanding scalability with the use of latches results in complicated code, whose correctness is difficult to verify.
   iii) Acquisition and inspection of latches can introduce overhead as communication costs – memory associated with a latch needs to be fetched into the processor cache on which a thread works resulting in invalidation of the latch in other processor caches and an eventual ping-pong of the latch between processor caches when the latch is used frequently. This incurs latency and bandwidth costs.

The PALM algorithm supports latch-free modifications and traversals on in-memory B+ trees by using the Bulk Synchronous Parallel (BSP) model. The BSP model proceeds in a sequence of steps, each of which is characterized by a local computation, communication and barrier synchronization phase.

PALM operates on batches of queries concurrently. The batch is split into series of queries, and each series is operated on by a thread. The PALM algorithm has various stages. The series of queries assigned to a thread at the beginning of the algorithm is subject to modification in the course of the algorithm. Threads communicate with each other at the end of each stage in order to determine the work they need to do as well as to allow all the threads to start the next stage simultaneously. PALM optimizes this communication by using point-to-point synchronization between pairs of threads instead of expensive all-to-all barriers.
**PALM Algorithm:**

The stages used in the PALM algorithm are given below. At any point all the threads occupy the same stage.

1. **Divide** tree queries in $O$ among threads.
2. Independently **search** for leaves for each query.  \[ \text{Stage 1} \]
3. **Redistribute work** to ensure no modification contention, and **ensure ordering** of queries.
4. **Modify leaves** independently.
5. Proceed in ‘lock-step’ up the tree, modifying internal nodes and redistributing work, up to the root.  \[ \text{Stage 3} \]
6. A single thread **modifies the root**, (if necessary), potentially changing the depth of the tree.  \[ \text{Stage 4} \]

The detailed algorithm is as follows:

```
PALM \( (O, T_D, i, t) \)
    // O are queries, \( T_D \) is the tree
    // i is the thread-id, and t is the # of threads
1  \( O_{\tilde{i}} = \text{PARTITION-INPUT} (O, i, t) \)
2  \( L_{\tilde{i}} = \text{SEARCH} (O_{\tilde{i}}, T_D) \)
3  \( \text{SYNC} (i, t) \)
4  \( L_{\tilde{i}}' = \text{REDISTRIBUTE-WORK} (L_0, \ldots, L_{t-1}, i) \)
5  \( R_{\tilde{i}}, O'_{L_{\tilde{i}}'} = \text{RESOLVE-HAZARDS} (L_{\tilde{i}}', O, D) \)
6  \text{for} \ (O_\lambda, \lambda) \ \text{in} \ (O'_{L_{\tilde{i}}'}, L_{\tilde{i}}') 
7       \( M_{\tilde{i}}^1 = M_{\tilde{i}}^1 \cup \text{MODIFY-NODE} (O_\lambda, \lambda) \)
8       \( \text{SYNC} (i, t) \)
9  \text{for} \ d = 1 \ \text{to} \ \text{depth} (T_D) - 1 
10      \( M_{\tilde{i}}^{d'} = \text{REDISTRIBUTE-WORK} (M_{0}^{d}, \ldots, M_{t-1}^{d}, i) \)
11  \text{for} \ (\Lambda, \eta) \ \text{in} \ M_{\tilde{i}}^{d'} 
12      \( M_{\tilde{i}}^{d+1} = M_{\tilde{i}}^{d+1} \cup \text{MODIFY-NODE} (\Lambda, \eta) \)
13  \( \text{SYNC} (i, t) \)
14  \text{if} \ i == 0 
15      \text{HANDLE-ROOT} \left( \cup M_{\tilde{i}}^{d+1}, T_D \right) 
16  \text{return} \ R_0, \ldots, R_{t-1} 
```
We discuss the sub-procedures used in the algorithm below:

**PARTITION-INPUT:**

**Input:**

i) The batch of queries, which is the input to the PALM algorithm itself.  
ii) Current thread number, i  
iii) Total number of threads, t

**Explanation:** This sub-procedure divides the list of queries evenly among the threads.  

**Output:** List of queries for thread i

**SEARCH:**

**Input:**

i) Current thread number, i  
ii) List of queries (L) assigned to thread i and obtained from PARTITION-INPUT

**Explanation:** This sub-procedure traverses through the tree and collects and returns leaf nodes that the queries in L modify/read.  

**Output:** List of leaves associated with L

**SYNC:**

**Input:**

i) Current thread number, i  
ii) The depth of the tree at which SYNC is performed, d

**Explanation:** At any point all the threads occupy the same stage; at the end of every stage there is a SYNC which ensures that all threads move on to the next stage together. Within a stage, REDISTRIBUTE-WORK function guarantees there are no contention issues by ensuring that the threads do not make modifications to the same tree node. When the order of the queries is changed such that they are sorted by their keys (the order of the queries having the same key should not be jumbled among themselves, their order relative to one another should be maintained from the original batch input), contention issues can occur only between adjacent threads. Therefore, when the queries are first sorted, a thread ‘i’ only needs to be synchronized with threads i-1 and i+1. In other words thread ‘i’ can be allowed to move on to the next stage (at the beginning of which a REDISTRIBUTE-WORK is performed) as soon as threads i+1 and i-1 arrive. This point-to-point synchronization is a significant improvement over all-to-all barriers.  

The SYNC algorithm is given below:
At the beginning of every stage REDISTRIBUTE-WORK re-partitions the work at the current level based on the tree nodes to be modified. REDISTRIBUTE-WORK can be called in Stage 2 or Stage 3. The input and output vary depending upon the stage it is called in. Stage 2 of REDISTRIBUTE-WORK is described below and stage 3 of REDISTRIBUTE-WORK is described after the MODIFY-NODE function.
REDISTRIBUTE-WORK (Stage 2):

Input:
   i) The list of ‘list of leaves’ output by the SEARCH function of each thread
   ii) Current thread number, i

Explanation: Each thread goes through the list of leaves associated with each of the threads before the current thread (threads whose thread number is less than that of the current thread) and removes those leaves from the current thread’s list of leaves (if present). Therefore lower numbered threads are given priority when multiple threads have updates that must occur on a single node. If the queries are previously sorted by the keys, each thread only needs to look up the thread immediately preceding the current thread thereby reducing communication costs significantly.

Output: A subset of the list of leaves returned by the SEARCH function of the current thread, which is the set of leaves the current thread will be modifying in this stage

RESOLVE-HAZARDS:

Input:
   i) List of leaves output by REDISTRIBUTE-WORK for the current thread
   ii) Current thread, i

Explanation: From the batch of queries given as input to the PALM algorithm, this function retrieves the queries that pertain to the list of leaves given as input while maintaining the order in which the queries appear in the batch so as to ensure serializability.

Output: The queries pertaining to the list of leaves passed as input.

MODIFY-NODE:

Input:
   i) Current leaf node, l
   ii) Queries (O) that modify the leaf node l

Explanation: Bulk node modifications are performed by MODIFY-NODE. The following aspects may be noted about the modifications applied to a node by this function:
   a) Any number of insertions/deletions can be handled at once, rather than one.
   b) Insertion and deletion queries to a single node may be performed together.

Once insertions are performed, if the order of the modified leaf/leaf node exceeds or goes below the threshold tree order, a helper function, BIG-SPLIT, creates one or more new nodes and assigns keys to the list of leaves (including the current leaf node, l) in strictly sorted order while satisfying the order property of B+ trees. Any number of splits is permitted to accommodate inserted items, but the changes are not immediately reflected on the relevant parent node. Information on how to update each node's parent is returned in a modification list.
Output: A modification list containing the new leaf/internal nodes and the relevant parent to be modified.

The detailed MODIFY-NODE algorithm is described below:

```
MODIFY-NODE (Λ, η)
  // Λ is sequence of modifications to node η.
  // If η is internal, Λ is a modification list.
  // If η is a leaf, Λ is a series of INSERT and DELETE queries.
  1 E = items (η)
  2 K = ∅
  3 for m in Λ
  4   K = K ∪ orphaned-keys (m)
  5   if class (m) = +
  6     E = E ∪ items (m)
  7   elseif class (m) = -
  8     E = E \ items (m)
  9   if |E| > MAX-DEGREE
 10      η, η', η'', ..., = BIG-SPLIT (E)
 11      return {+, parent (η), η', η'', ..., K}
 12   elseif |E| < MIN-DEGREE
 13      return {−, parent (η), η, K ∪ descendant-keys (E)}
 14   else
 15      child-ranges (η) = E
 16      return {∅, K}
```

REDISTRIBUTE-WORK (Stage 3)

Input:

i) The list of ‘modification list’s output by ‘MODIFY-NODE’ of each thread
ii) Current thread number, i

Explanation: Similar to the REDISTRIBUTE-WORK of Stage 2, the current thread discards from its modification list the entries corresponding to the nodes that are going to be modified by a lower numbered thread. In addition to this, the current thread goes through the modification lists of all higher numbered nodes to get a list of new nodes created. The current thread uses this information to update the nodes that it will modify in this stage. Again, if the queries are sorted by the keys the current thread only needs to look up the previous and the next nodes.

Output: The subset of the modification list output by MODIFY-NODE of the current thread including the information that it salvaged from higher numbered threads.
HANDLE-ROOT:

Input:
(i) Union of all the modification lists obtained from the output of the function MODIFY-NODE
(ii) The B+ tree

Explanation:
Outstanding modifications that may remain after processing the highest depth in the tree are handled by this function. If order of the root exceeds or underflows the threshold order of the tree after making the modifications, the root is split to accommodate these modifications. This function is executed by only one thread.
Performance:

The performance of PALM was tested on three different machines with varying number of cores. The details about the machines is as follows:

1) Dual Core 2.2 GHz Intel Pentium
2) 4 Core 2.6GHz AMD Opteron
3) 8 Core 1.8GHz Intel Xeon

On each of these machines the time taken for 3000 random inserts was measured. The experiment was repeated with 2, 3, 4 and 5 threads. The results of the experiment is as follows:

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
<th>5 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.011</td>
<td>0.06</td>
<td>0.14</td>
<td>0.3</td>
</tr>
<tr>
<td>4</td>
<td>0.024</td>
<td>0.05</td>
<td>0.06</td>
<td>0.09</td>
</tr>
<tr>
<td>8</td>
<td>0.012</td>
<td>0.02</td>
<td>0.019</td>
<td>0.03</td>
</tr>
</tbody>
</table>

It can be inferred that PALM performs better with increasing number of cores. However, the experiments show that for a given machine with a particular number of cores, the performance deteriorates marginally with increasing number of threads. This could be attributed to the cost of synchronization of the threads in our implementation of the algorithm. With a more efficient implementation significantly better results could be expected.

Conclusion:

As the number of cores increases, usage of locks poses the following problems:
   i) Lock-contention increases and this results in poor scaling.
   ii) The cost of obtaining locks without contention will also increase due to growing costs of cross-core/cross-socket communication.

In addition, the cost of all-to-all barrier synchronization increases with the number of cores as well.

PALM is a lock-free algorithm and eliminates the cost associated with using locks. In addition, PALM also uses point-to-point synchronization and reduces the cost of using all-to-all barriers with increasing number of cores.