Database and Information System

Performance Analysis of B+ tree and Buffer

Harsh Parmar 130050017

Utkarsh Mall 130050037

Mayank Sahu 130050038

Introduction

In modern times processor performance has increased many folds due advancements in nanotechnology however further increase in the processor performance will not increase the performance of a Database system since memory system are lagging much behind in access speeds and have become bottleneck in Database's performance. Lots of research work and resources are being spent to improve the performance of memory systems.

The performance of any database system depends on two major factors. First factor on paged file layer as **buffer replacement schemes**. The second factor on at the access method layer on the **B+ tree structure**. Our aim is to study, how these factors affect performance of a database system in real time.

Objectives

On Paged File Layer, We analyzed the performance of different buffer replacement schemes. We implemented 3 of these schemes:

1. Least Recently Used(LRU)
2. Most Recently Used(MRU)
3. Random Replacement(RAN)

Block nested join is one of the main data operations, which requires same block of data more than once in its execution. So we simulated a Block nested join operations, with these three replacement schemes.

On the Access Method layer, we analyzed performance of B+ tree under several parameters that influence the access performance such as key size, node size, number of records, data orderliness and so on.

To analyze the dependence of AM layer on PF layer, we analyzed how B+ tree performance is affected by the buffer replacement scheme.

Experiment Design for buffer miss analysis

First we implement some of the sophisticated buffer replacement schemes, mainly Least recently used(**LRU**) and Least frequently used(**LFU**) buffer replacement scheme.

Next task would be to analyze how these schemes affect the miss ratio for some standard record sets under some specific operations such as join and B+ tree insertion.

Nested Block Join :

Experiment : We analyze the number of page faults (misses) taking place when we join two relations of different sizes with different buffer replacement schemes and compare the time taken in processing for these misses. All the work in section of experiment has to be done in PF layer of ToyDB.

For this experiment we took outer relation with size 10,000 whereas for inner relation we did the experiment for two sizes, 30 and 40.

Result : MRU performed better in both experiment, where as LRU performed the worst.

Random buffer replacement scheme always remained in between LRU and MRU.

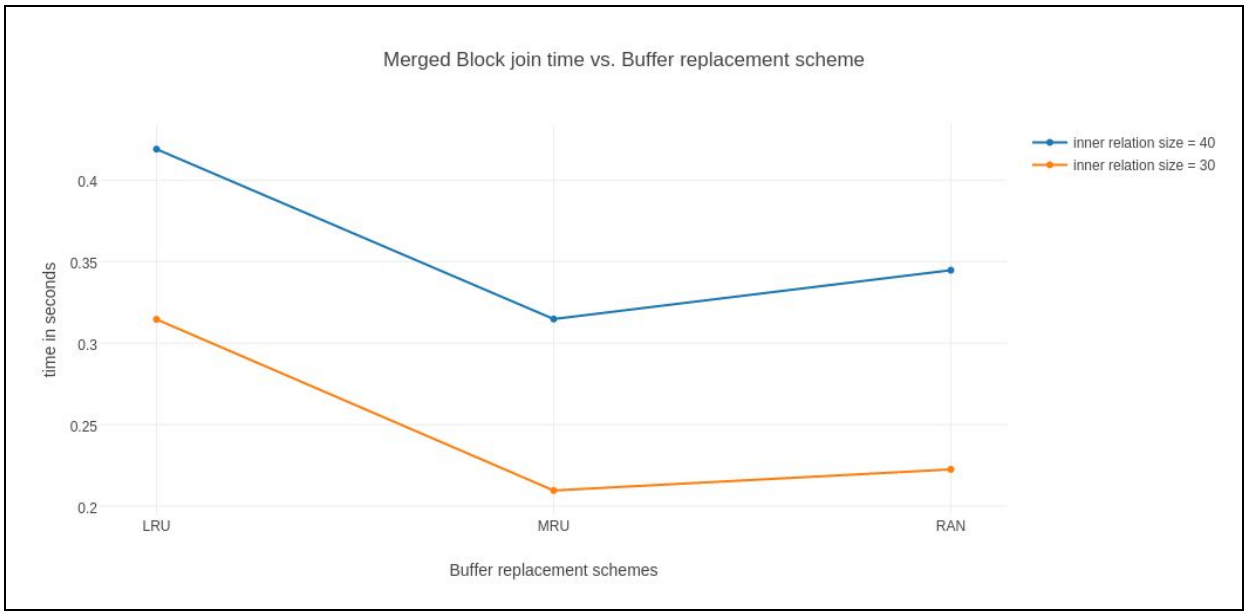

Fig 1 : Merged block join time vs. buffer replacement scheme. Blue line shows the variation when inner relation is of size 40 and orange shows that when inner size is 30.

B+ tree insertion time :

Experiment : We wanted to analyze the effect of buffer replacement scheme on the B+ tree insertion operation. We insert new records in the created index which may be in ascending or descending order or may be randomly ordered under different buffer replacement scheme(LRU, MRU, RAN). For this we changed buffer size to 100.

We inserted 1,000,000 values for our experiment.

Result : Random buffer replacement outperforms the other two in case of random data insertion. MRU comes second in performance and LRU performs the worse.

However in case of increasing and decreasing data insertion LRU performs the best out of three and Random performs worse. However there is not much difference in performance when the data is inserted in increasing or decreasing order.

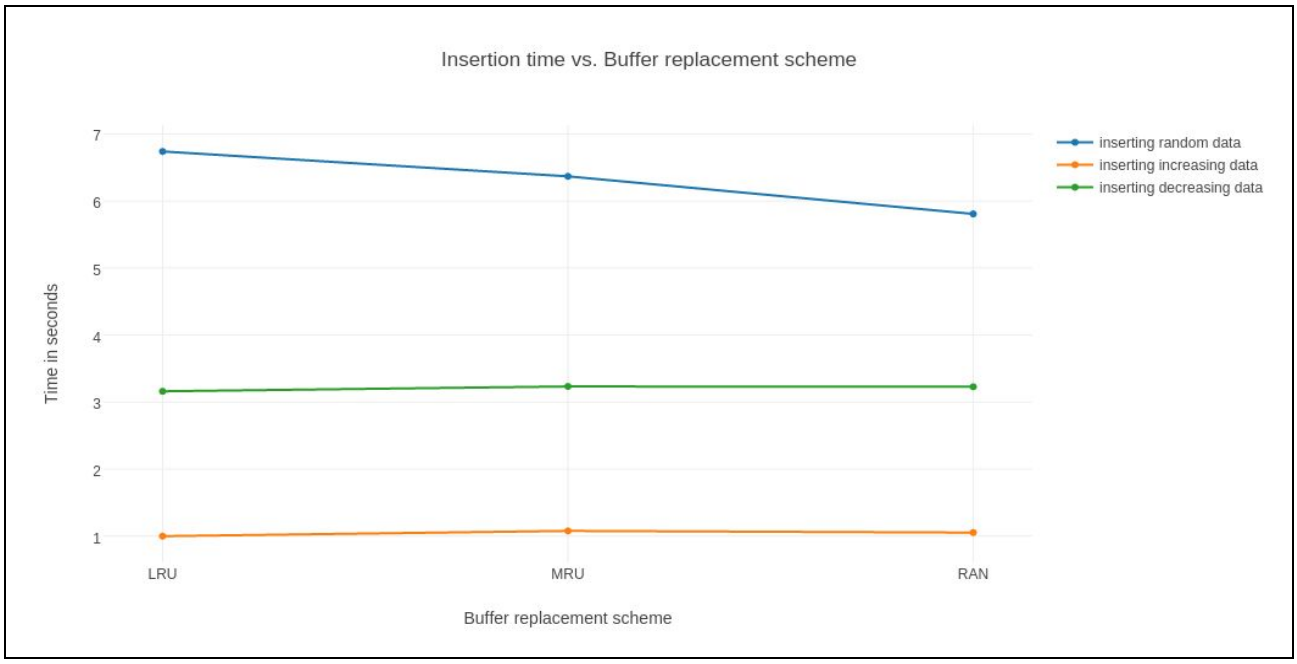

Fig 2 : B+ insertion time vs. Buffer replacement schemes. Blue line depicts the variation when data is inserted in random order. green for decreasing data and orange for increasing data.

B+ tree analysis

Record number

Experiment: The same AM layer code runs on relations of different record number sizes. First a record of some cardinality is inserted in an index file. Then the index file is accessed for point querying for a large number of times. The time/misses utilized by this querying accesses is compared for different record sizes.

We perform this test over records of size 10,000/100,000/1,000,000 sizes, and randomly query 1 million times over this tree(after clearing cache).

Results: Record number has a linear impact to the number of leaf nodes, and thus has a logarithmic impact to the tree depth. The point query performance decreases with the increasing number of record, especially when record number is greater than 1,00,000. The cache miss ratio (both L1 and TLB) increases with the record size.

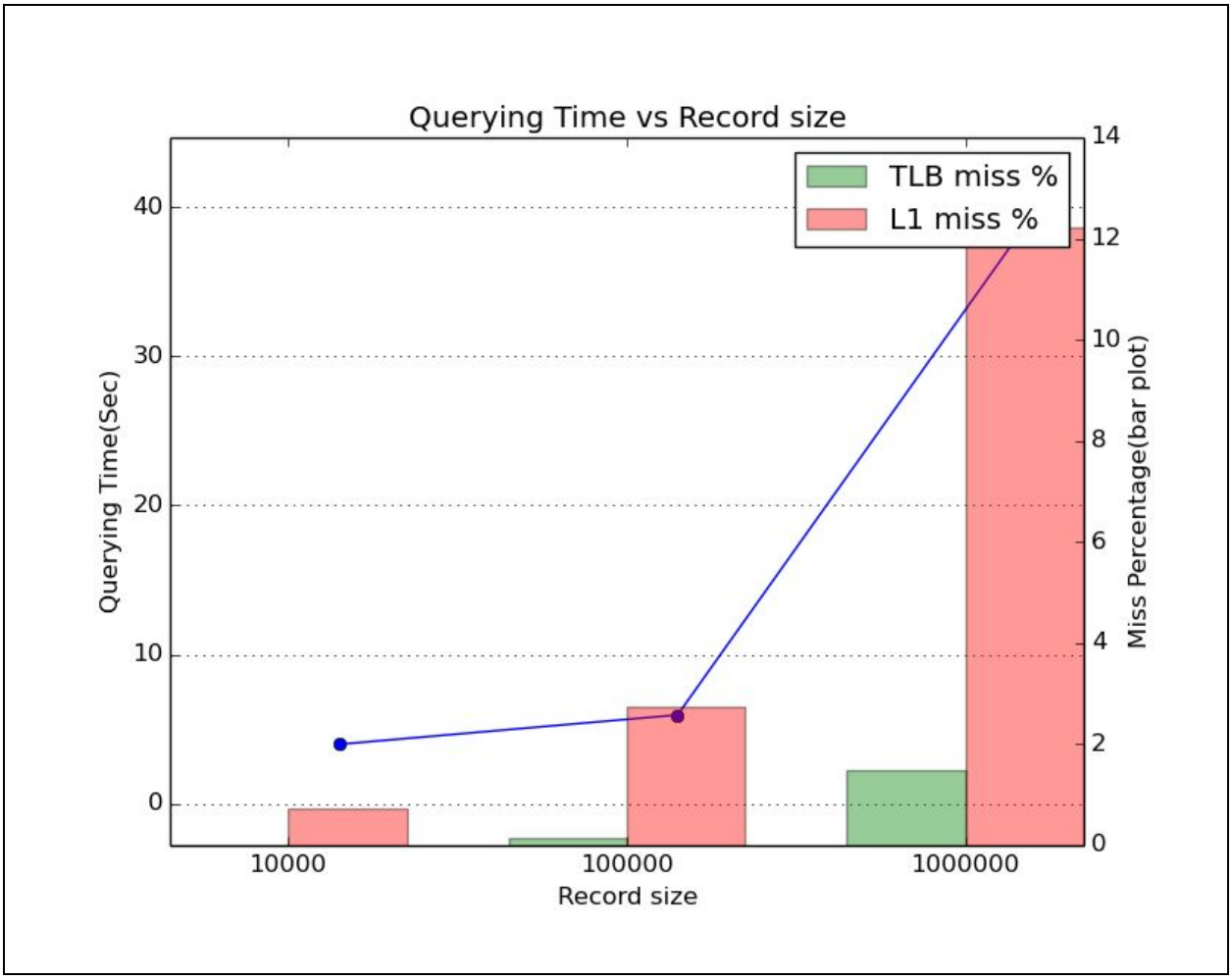

Fig 1. Querying time vs record size. The blue line is the time in 1 million point querying over relations of varying sizes. The effect on time is visible largely after 100,000 record number. But the TLB/L1 miss ratios are increasing along with it.

Key size

Experiment: For this we make different indices on relation with keys of different sizes. First a record of some with some key size is inserted into the index file. Then the index file is accessed for point querying for a large number of times. The time/misses utilized by this querying accesses is compared for different key sizes.

We perform this test over records with keys of size 32B/64B/128B , and randomly query 1 million times over this tree(after clearing cache).

Results: Varying sizes of key, keeping the node size constant, the tree fanout should decrease and hence the depth of B+ tree should increase. This should lead to more time required in accessing a tuple. The TLB/L1 miss ratio is not changed much.

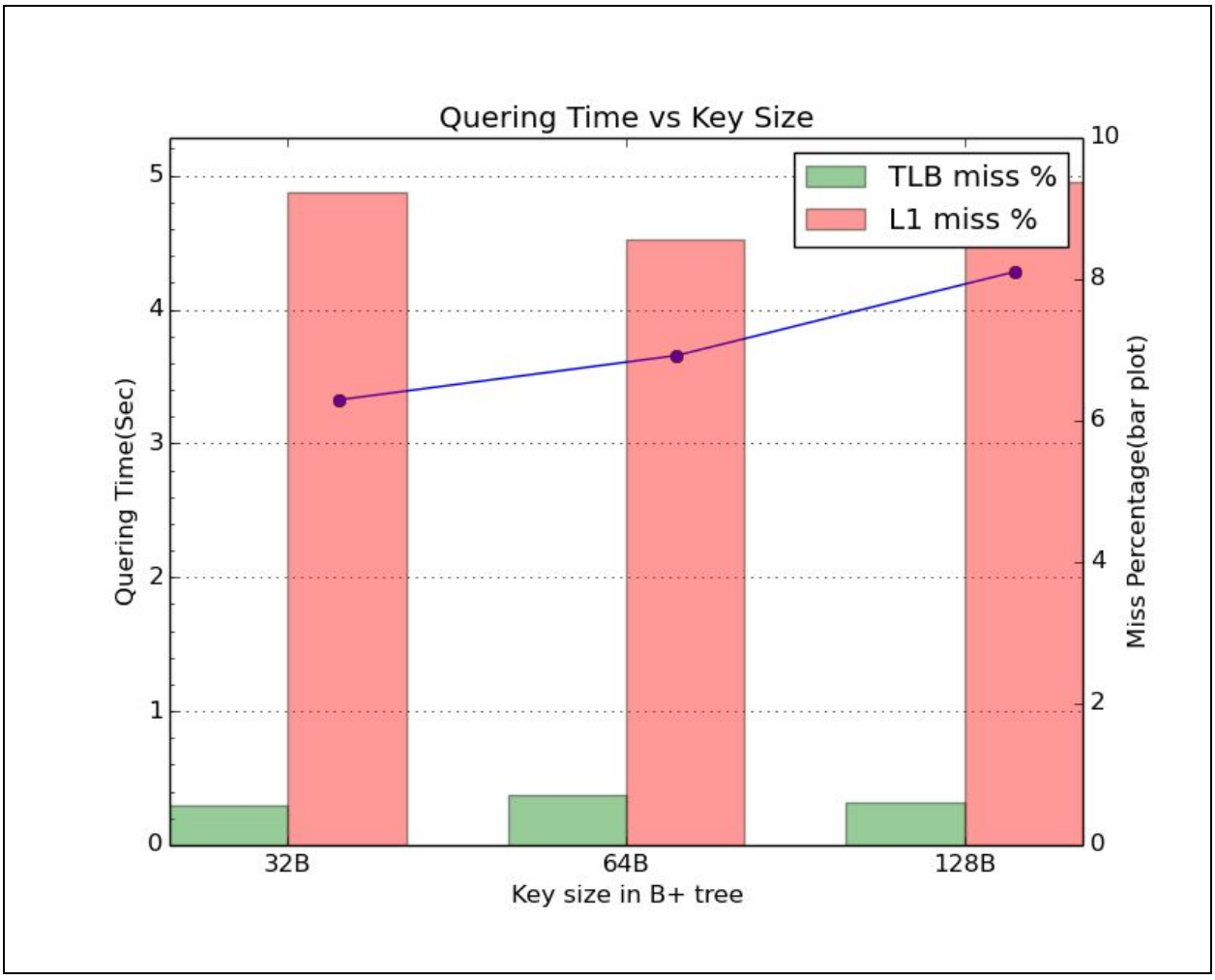

Fig 2. Querying time vs key size. The blue line is the time of 1 million point querying over relation with varying key sizes, but the TLB/L1 miss ratios more or less constant.

Node size

Experiment: For this we make different indices on relation with varying node sizes. First a record is inserted into the index file of some fixed page size. Then the index file is accessed for point querying for a large number of times. The time/misses utilized by this querying accesses is compared for different node sizes.

We perform this test over records with nodes of size 1KB/2KB/4B , and randomly query 1 million times over this tree(after clearing cache). For Varying node size we changed internal source code of toyDB.

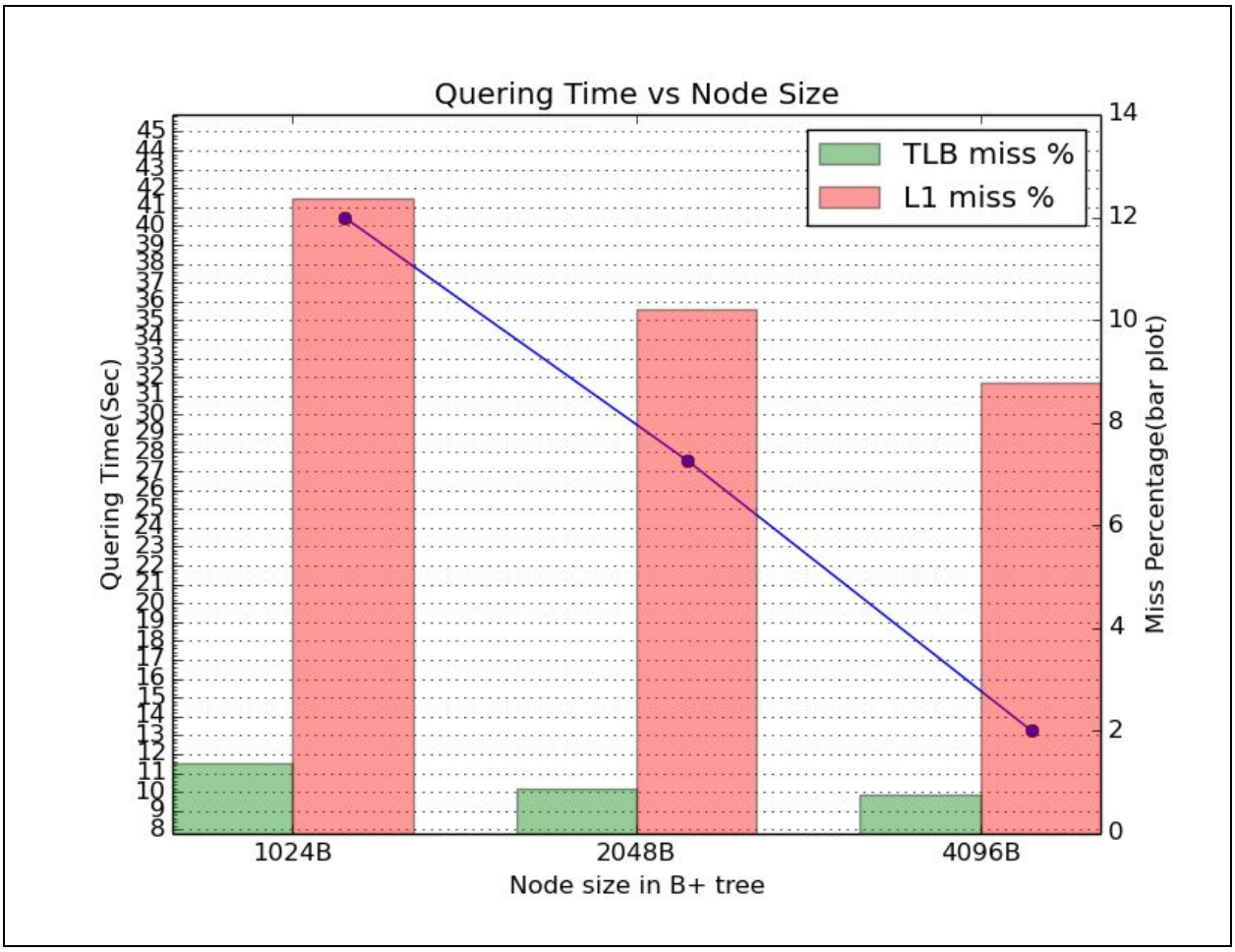

Fig 3. Querying time vs node size. The blue line is the time of 1 million point querying over relation with varying key sizes. The time and misses are decreasing with node size.

Result: Decreasing node size will increase leaf node numbers and hence increase the tree depth logarithmically. Hence the point querying time should decrease and it decreases as expected.

Orderliness

Experiment: Since orderliness does not affect much on the B+ tree structure, but impacts on the B+ tree operations we look for insertion time. We insert new records in the created index which may be in ascending or descending order or may be randomly

ordered. We insert 1 million records in index file in order and out of order and compare time/misses.

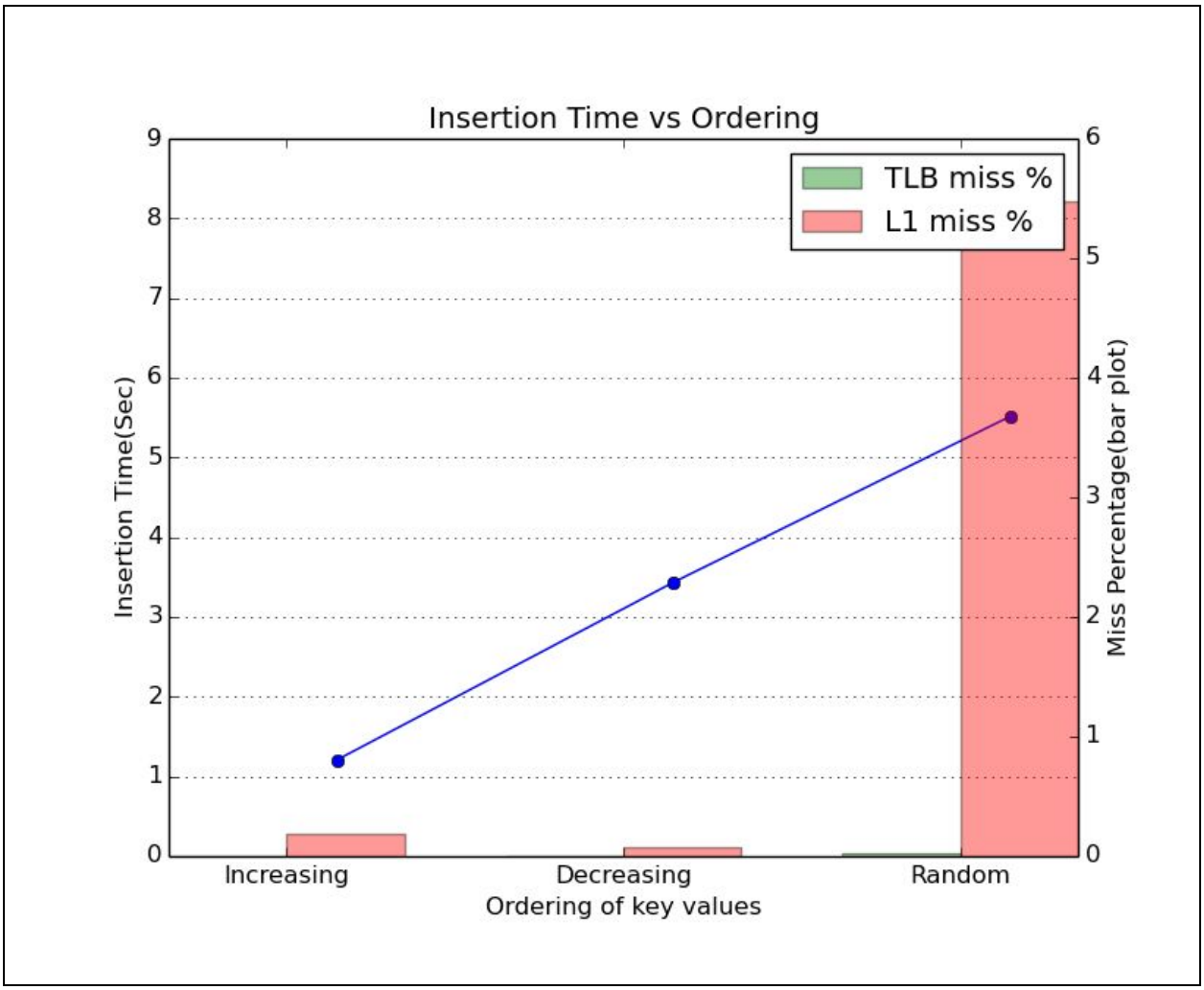

Fig 3. Insertion time vs ordering. The blue line is the time of insertion of 1 million tuples. The time is least for increasing and most for random insertions(see result subsection for reason). Misses are more or less equal for increasing and decreasing, but are very high for random insertions. Here the misses are store misses.

Result: If records are inserted in ascending order, B+ tree doesn't need to move data and should have best performance. However in case of records being inserted in decreasing order data needs to moved around the node which would cause a slight deterioration in performance. The worst performance is encountered when records are inserted randomly as random keys can't share caches between operations. These expectations are indeed reflected by the graph of experimental results.

Conclusion

To conclude, at PF layer, we saw how buffer replacement schemes affect basic most relational operation Nested Block join. Why MRU is better than LRU in this operation even though LRU is assumed to be better than any other replacement scheme.

For B+ tree we found out that all the factors node size, key size, ordering of insertion and record size plays key role in efficiency of execution. In orderliness L1 cache size is the main factor for inefficiency of insertion, whereas in key size even though miss ratio is large it is not a key factor.

We saw interrelation between PF and AM layer, how changing the buffer replacement scheme changes the B+ tree insertion time.

Instruction

For B+ Tree analysis :

- Open /toydb/B+tree/*

- execute the command “sudo bash execute.sh” or switch to sudo user and run “bash execute.sh” (this will run whole experiment and draw graphs for it.)

For Buffer Miss analysis :

- Open /toydb/bufferhit/*

- execute the command “sudo bash execute.sh” or switch to sudo user and run “bash execute.sh” (this will run whole experiment and draw graphs for it.)

Remarks

The experiments for B+ tree might be affected by the page fault in reading datafiles and other external factors too.

Algorithm

The buffer replacement schemes are needed to be implemented to analyze their performance, mainly.

1. LRU Buffer replacement scheme

```
if using_buffer_size < max_buffer_size:
        return new buffer space from empty list
else:
        maxage=0
        newbuffer=Null
        for all B in Buffer pages:
                if(B.age>maxage and B is unfixed):
                        maxage=B.age
                        newbuffer=B
        writeback newbuffer to disk
        return newbuffer
for all B in buffer pages:
        B.age++ if(age>1000)age=0 //for stopping overflow
```

2. MRU Buffer replacement scheme

```
if using_buffer_size < max_buffer_size:
        return new buffer space from empty list
else:
        minage=1001
        newbuffer=Null
```

```
        for all B in Buffer pages:
                if(B.age<minage and B is unfixed):
                        minage=B.age
                        newbuffer=B
        writeback newbuffer to disk
        return newbuffer
for all B in buffer pages:
        B.age++ if(age>1000)age=0
```

3. Random Buffer replacement scheme

```
if using_buffer_size < max_buffer_size:
        return new buffer space from empty list
else:
        writeback and return one of unfixed Buffers uniformly randomly
```

Changes in ToyDB code

In PF layer: We modified *'buf.c'* to imple,ment *'bufLRU.c'*, '*bufMRU.c'*, '*bufRAN.c'* which are various buffer replacement schemes. We also changed *'pftypes.h'* to change the buffer size for B+ tree insertion experiment.

In AM layer: We modified *'pf.h'* to change node size (in node size experiment).

Programming Languages and Tools

Programming language:

1. C
2. Bash and python scripts

Tools:

1. ToyDB C library
2. perf (Linux tool for analyzing hits/misses/time of different caches and buffers)
3. MatplotLib python plotting library