CS165 Project Report

Yifan Wu

April 2014

1 Performance

1.1 Methodology

I ran all the queries three times and took the average. The variance is not too large but might matter when the results are close.

I chose to run on the VM because the limited computation power offsets the size of the data tested, and it has less of other things running which could cause a lot of noise for the data.

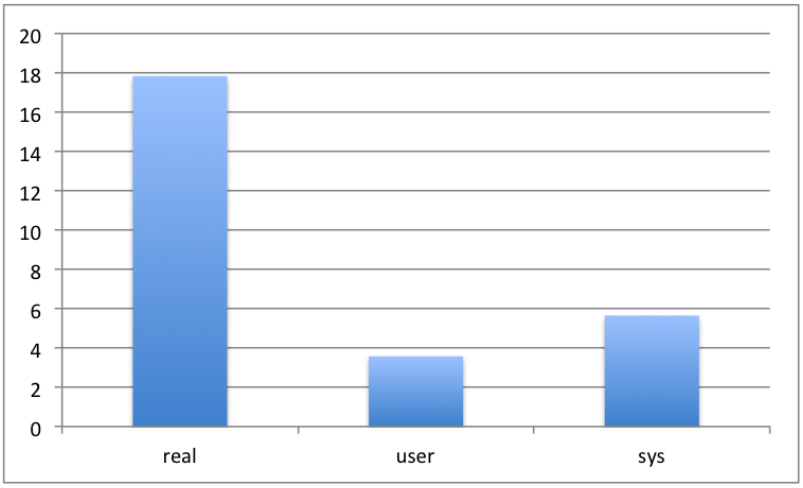
MariaDB (variation of MySQL) was used and the test scripts located under the folder **p3tests**.

1.2 Load

First test is loading a large file into system without any operations (unsorted) as a baseline. Since I load the whole line into a big enough buffer there is no page size at play here, and the relative numbers are:

Type	Time (s)
real	17.84
user	3.58
sys	5.65

The SQL time that took to load was about 6 seconds. Not sure about the detailed breakdown but seems to me about the same.



1.3 Select

1.3.1 Sorted

Query times are a lot faster than the loads, mostly because IO is the most expensive operation, and this is nicely confirmed by the fact that the gap between the user time and the real time is a lot smaller, in that the wait on IO is a lot less, but relatively the difference is pretty big. So I wondered if I could improve performance by reading block at a time (to reproduce just add the flag **NOBLOCKREAD** through make). And below are the comparisons:

Type	B 0 Time (s)	B 64 Time (s)	B 128 Time (s)	B 512 Time (s)
real	0.079	0.052	0.024	0.063
user	0.001	0.000	0.000	0.000
sys	0.002	0.001	0.000	0.001

We could see that reading block at a time improves initially compared to no block, then as the block gets larger, the performance decreases a little, potentially due to higher level caching effects. Not going to test against SQL since MariaDB does not provide the option.

1.3.2 Unsorted

Type	Load Time	Fetch Time
real	8.39	0.028
user	1.58	0.001
sys	2.47	0.001

The SQL result took a grand total of **0.83 s**. Mine did a lot better!

1.3.3 BTree

To simulate sorted in SQL, I add a layer of index over the first column, in MariaDB, it's just specified as BTREE.

The same data as before, but with the first column as b plus tree, and we get

Type	Load Time	Fetch Time
real	17.2	0.028
user	3.9	0.000
sys	4.8	0.001

Which is a little bit unfortunate because the time is about the same. My theory is that the cost to the computation is so small that it doesn't make a difference. In MariaDB, after the index was created, the same call only took **0.01 sec**. And the time for the index was **2.35 seconds** (upfront cost). Overall MariaDB had a better bptree algorithm than mine I suppose.

1.4 Hash

Since my previous test was apparently too little computation requirement to make a big difference, I made the hash tests a bit harder.

On the SQL side, the following was performed: SELECT * FROM p2test3 INNER JOIN p2test3b ON (p2test3.t4a = p2test3b.b4a) With index, it took **0.16 sec** (21688 rows).

For my different hashes:

TYPE	LOOP	HASH	TREE	SORTED
real	39.380	18.623	7.393	24.532
user	0.010	0.014	0.009	0.012
sys	0.060	0.052	0.051	0.087

We could see that for the specific cases tree did really well, followed by hash, then sorted and the worst performance as elected is loop.

2 Virtual Machine Specs

Thru the results of **lscpu**

Architecture	i686
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	1
On-line CPU(s) list	0
Thread(s) per core	1
Core(s) per socket	1
Socket(s)	1
Vendor ID	GenuineIntel
CPU family	6
Model	58
Model name	Intel(R) Core(TM) i7-3540M CPU @ 3.00GHz
Stepping	9
CPU MHz	2992.865
BogoMIPS	5985.73
Hypervisor vendor	VMware
Virtualization type	full
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	4096K

3 Design Doc for P3

3.1 Aggregation Implementation

The overall task is that the aggregator will materialize the result and perform the relevant operations. One potential optimization is to cache the result in the variable in case another access is made, and remove the result after sometime (probably will not have time). Probably will not have enough time for this.

The implementation should be different from the math operators *if* this were a real DB, where I would be mostly streaming info. However since the DB will just be lugging the values around it is not that different.

Interface:

```
\begin{lstlisting}typedef enum {MAX, MIN, AVG} aggregate_op;
```

---

i

3.1.1 Math Operators

Assuming we only support two variables at the same time

Interface for things to implement:

```
Add new enum:

\begin{lstlisting}typedef enum {ADD, SUB, DIV, MUL} math_op;
```

Parsing does not change much for the previous two — just need to add to regex

3.2 Join Implementation

I realized that row IDs must be enabled to support joins. I thought about doing bucket join so as to increase locality, but with the current implementation it's actually not possible since we don't have control over both of the vectors (instead just one).

3.2.1 loopjoin

Outer loop the larger column, inner loop on the other, and mark the two bit vectors when there is an equal. I will stream this and not load all at the same time.

```
for(i=0; i < c1.m.size; i += PAGE_SIZE) {
  load_block(c1.fp, val_buf1, i*PAGE_SIZE, PAGE_SIZE);
  for(j=0; j < c2.m.size; j += PAGE_SIZE) {
    load_block(c2.fp, val_buf2, j*PAGE_SIZE, PAGE_SIZE);
    for(r=0; r < PAGE_SIZE; r++) {
      for(m=0; m < PAGE_SIZE; m++) {
        if (val_buf1[r] == val_buf2[m]) {
          mark_bv(c1->bv, i*PAGE_SIZE+r);
          mark_bv(c2->bv, j*PAGE_SIZE+m);
        }
      }
    }
  }
}
```

3.2.2 sort join

Similar loop idea, just 1) need to be sorted 2) slightly index processing. Also depending on the column type, might need to create another sorted array in memory: if it is already sorted, load it in memory anyways so that it is consistent!.

Interface for the logic:

```
while(i > 0 || j > 0)
{
```

```
  if (c1v[i] < c2v[j]) {
    i ++;
  }
  else if (c1v[i] > c2v[j]) {
    j++;
  }
  else {
    mark_bv(c1->bv, i);
    mark_bv(c2->bv, j);
    // but we need to also check previous values since there might be duplicates
    itr_i = i;
    while(c1v[itr_i-1] == c1v[itr_i]) {
      itr_i --;
      mark_bv(c1->bv, itr_i);
    }
    // do the same for c2

    itr_j = j;
    while(c2v[itr_i-1] == c2v[itr_j]) {
      itr_j --;
      mark_bv(c2->bv, itr_j);
    }
  }
}
Put your code here.
```

3.2.3 hashjoin

I will be implementing static hashing with 2 passes. Basically sweep thru once and have an int array the size of the hash buckets then random access via array index. Initially everything is defined as 0. Then do a second sweep to get an update on the corresponding positions of the hash values. To remap the number to the new value, I will just implement a array of tuples and check. Look up will be linear to the number of buckets. Then I will have a tuple of the original location and the new location based off the array. This will be sorted so look up is log(num buckets).

This is probably easier (or the same) as a dumb linked list join so will impelemnt this

---

```
void load_bloc(FILE *f, int *buf, uint32_t pos, uint32_t size);

// FIRST PASS, fill in hash_count
int *hash_count = malloc(sizeof(int) * HASH_SIZE);
// c_h for column hashed
for(i=0; i < c_h.m.size; i += PAGE_SIZE) {
  load_block(c1.fp, val_buf1, i*PAGE_SIZE, PAGE_SIZE);
  for(m = 0; m < PAGE_SIZE; m ++) {
    hash_count[hash_func(val_buf1[m])] ++;
  }
}
```

```
int sum =0;
for(i=0; i< HASH_SIZE; i++)
{
  // destructive update is fine
  hash_count[i] = sum;
  sum += hash_count[i];
}

int *hash_seen = malloc(sizeof(int) * HASH_SIZE);
int *hash_table = malloc(sizeof(int) * c_h.m.size);
// SECON PASS, actually build the hash table now
for(i=0; i < c_h.m.size; i += PAGE_SIZE) {
  load_block(c1.fp, val_buf1, i*PAGE_SIZE, PAGE_SIZE);
  for(m = 0; m < PAGE_SIZE; m ++) {
    hash_table[hash_count[hash_func(val_buf1[m])] + hash_seen] = val_buf1[m];
    hash_seen[hash_func(val_buf1[m])] ++;
  }
}

// Last step check against the other column
```

3.2.4 treejoin

This join is utilizing the existing tree structure of the column, whereby it first searches in the tree and then check the search against the requirement of the column by checking against the bit vector. Note that I'm not currently supporting row IDs, which based on my current implementation would only happen if we join on top of joins.

4 Known Issues

4.1 Valgrind

I have not cut through all the issues with valgrind, and depending on the machine and the size of the data, sometime the memory crashes. I'm still in the process of cleaning these up. Hopefully by project 4.

4.2 Memory Limitation

So right now the variables are all kept in memory. By design I don't materialize most of them, however bit vectors could still be of decent size. If we keep on using the server, it's eventually going to run out of memory. I might be able to implement some kind of cleaner mechanism that deletes the variables if they haven't been used for a while.