GSoC 2018 Project Proposal

Description:

Implement the project idea sorting algorithms benchmark and implementation (2018)

Applicant Information:

Name: Kefan Yang

Country of Residence: Canada

University: Simon Fraser University

Year of Study: Third year

Major: Computing Science

Self Introduction:

I am Kefan Yang, a third-year computing science student from Simon Fraser University, Canada. I have rich experience as a full-stack web developer, so I am familiar with different kinds of database, such as PostgreSQL, MySQL and MongoDB.

I have a much better understand of database system other than how to use it. In the database course I took in the university, I implemented a simple SQL database. It supports basic SQL statements like select, insert, delete and update, and uses a B+ tree to index the records by default. The size of each node in the B+ tree is set the disk block size to maximize performance of disk operation. Also, several kinds of merging algorithms are used to perform a cross table query. More details about this database project can be found here.

Additionally, I have very solid foundation of basic algorithms and data structure. I've participated in division 1 contest of 2017 ACM-ICPC Pacific Northwest Regionals as a representative of Simon Fraser University, which clearly shows my talents in understanding and applying different kinds of algorithms. I believe the contest experience will be a big help for this project.

Benefits to the PostgreSQL Community:

Sorting routine is an important part of many modules in PostgreSQL. Currently, PostgreSQL is using median-of-three quicksort introduced by Bentley and Mcllroy in 1993 [1], which is somewhat outdated. Using a more optimized sorting algorithm could bring a performance gain for many parts of PostgreSQL.

Apart from this, the PostgreSQL community is using Larson's linear hashing [2], which is outperformed by some modern hashing implementations like the cuckoo hashing [3]. A better hashing table implementation will probably bring better overall performance.

Finally, to find out the best sorting algorithm to use in PostgreSQL, I need to implement a benchmark for various kinds of sorting algorithm. This benchmark can be used to test future patches.

Deliverables:

- A benchmark for sorting algorithms
- Benchmark implementation for the introsort, timsort, dual-pivot quicksort and radixsort
- The winner under the given benchmark
- Industrial implementation for the selected sorting algorithm in `pg_qsort()`, `pg_qsort_args()` and `gen_qsort_tuple.pl`
- New hashing table implementation
- Review of other's patches

TimeLine:

Community Bonding Period (April 23 – May 14)

- Get in touch with mentors and other community members. Try to get suggestions on this proposal as soon as possible.
- Set up the development environment and get more comfortable with vim.
- Get more familiar with codebase. Fully understand the usage of `pg_qsort()`, `pg_qsort_args()` and `gen_qsort_tuple.pl.`
- Understand the implementation of `pgbench`
- Discuss on implementation of the benchmark - what is the “average” use case? What is expected from the community?
- Read research papers about sorting algorithms and hashing table implementation. Be prepared for coding.

Week 1 (May 14 – May 20)

- Read papers about introsort in details
- Implement the benchmark version of introsort

Week 2 (May 21 – May 27)

- Read papers about timsort in details
- Implement the benchmark version of timsort

Week 3 – 4 (May 28 – June 10)

- Implement the benchmark version of dual-pivot quicksort and radixsort (this should not take much time since we can find some c implementation already)
- Implement the sorting benchmark (Phase 1 and 2)

First Evaluation Period Deliverables

- Benchmark implementation of introsort, timsort, dual-pivot quicksort and radixsort
- Phase 1 and 2 of the benchmark implementation

Week 5 (June 11 – June 17)

- Review feedbacks from the first evaluation and make some changes accordingly
- Adjust plans for the following weeks if necessary

Week 6 (June 18 – June 24)

- Test the sorting algorithms under phase 1 and 2
- Phase 3 of the benchmark implementation

Week 7 – 8 (June 25 – July 8)

- Select the winner and implement its industrial version for `pg_qsort()` and `pg_qsort_args()`

Second Evaluation Period Deliverables

- Complete benchmark implementation
- Winner algorithm and some analysis
- New implementation of `pg_qsort()` and `pg_qsort_args()`

Week 9 (July 9 – July 15)

- Review feedbacks from the second evaluation and make some changes accordingly
- Industrial implementation for `gen_qsort_tuple.pl` (may encounter some challenges with Perl script)

Week 10 (July 16 – July 22)

- Test new hash table implementation
- Industrial implementation of hash table

Week 11 – 12 (July 23 – Aug 6)

- Final buffer period
- Review other's patches
- Prepare for the final submission

Implementation:

Benchmark:

After the discussion with the mentors and other community members, I had a new design of benchmark implementation. Basically, I will evaluate the performance of a sorting algorithm through three phases:

Phase 1: Test on random arrays

Input: sorting routine, data size, data type

Output: average runtime measured in CPU clock cycles

The main purpose is to evaluate the average performance of the sorting algorithms. Test cases will be generated using different array sizes and data types (integers, strings and tuples). The performance of each algorithm is measure in CPU clock cycles.

Phase 2: Test on worst (bad) case

Input: sorting routine, worst (bad) case dataset

Output: average runtime measured in CPU clock cycles

In this case, I plan to use pre-generated integer arrays to test the worse (bad) case performance of each algorithm. The worst (bad) case scenario I currently designed for each sorting algorithm is listed below:

introsort: Introsort uses quicksort at the beginning, so its worst-case scenario should be very similar with quicksort.

timsort: Timsort derives from mergesort. We need a specially designed case which will maximize the number of comparisons.

dual-pivot quicksort: The worst case occurs if you choose the first and last elements as pivots and the array is already sorted or reverse-sorted.

radixsort: According to how this algorithm works, it should have rather bad performance with very large data size. (radixsort consumes a lot of memory, which may cause page fault when quicksort only get cache misses)

In this phase, we can also use the median-of-three killer sequence to show the drawback of current quicksort implementation.

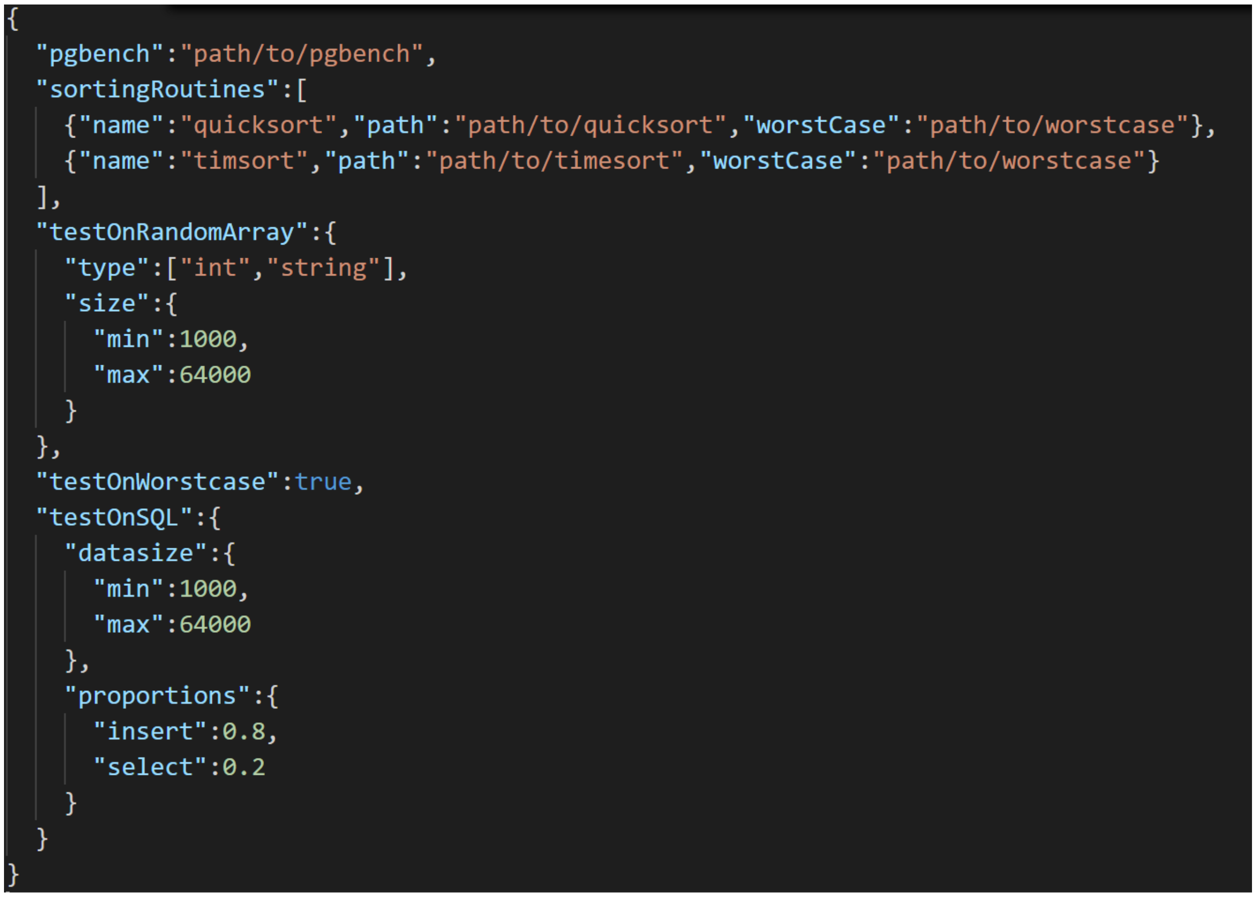
Phase 3: Test on SQL statement

Input: sorting routine, data size, proportions of different types of SQL statements

Output: average transaction rate (transaction per second)

After we get a winner from the first two phases, I am going to test it using SQL statements and compare it with current median-of-three quicksort implementation. The pgbench seems to be enough for this. In practice, we can assign different proportions to each type of SQL statement to simulate a variety of scenarios, such as select intensive or insert intensive cases.

An example config file for this sorting benchmark looks like (may not use JSON syntax in practice):



```json
{
  "pgbench":"path/to/pgbench",
  "sortingRoutines":[
    {"name":"quicksort","path":"path/to/quicksort","worstCase":"path/to/worstcase"},
    {"name":"timsort","path":"path/to/timesort","worstCase":"path/to/worstcase"}
  ],
  "testOnRandomArray":{
    "type":["int","string"],
    "size":{
      "min":1000,
      "max":64000
    }
  },
  "testOnWorstcase":true,
  "testOnSQL":{
    "datasize":{
      "min":1000,
      "max":64000
    },
    "proportions":{
      "insert":0.8,
      "select":0.2
    }
  }
}
```

Benchmark implementation of sorting algorithms:

Implement the pseudo-code provided by the research papers in C.

introsort: There is a highly optimized cpp implementation in STL (generally 20% faster than hand-coded cpp implementation). I will try to implement it in C to see if there's a performance gain.

timsort: Some implementations can be found in the source code of Python and Java SE 7. I can rewrite them in C.

dual-pivot quicksort: C implementation is given in some open source projects.

radixsort: C implementation is given in this research paper [4].

Industrial implementation of selected sorting algorithm:

The industrial version is basically an optimization based on the benchmark implementation. Some of the optimizations I come up with now are:

1. Check if input array is already sorted
2. Applying insertion sort directly for short arrays.
3. Convert recursions to iterations.
4. Optimize loops to avoid potential cache misses.
5. Use key abbreviation for string sorting.

But I think the method here highly depends on the actual implementation of that algorithm and it’s difficult for me to decide in the proposal stage.

New hashing table implementation:

I plan to test the cuckoo hashing. Since my time is quite limited, I will use the source code in other open source project and test the performance of this algorithm using pgbench. If this algorithm works well, I will implement an optimized version based on the research paper by myself.

Reference:

[1] Bentley, Jon L., and M. Douglas McIlroy. "Engineering a sort function." Software: Practice and Experience 23.11 (1993): 1249-1265.

[2] Larson, Per-Åke (April 1988), "Dynamic Hash Tables", Communications of the ACM, 31 (4): 446–457, doi:10.1145/42404.42410

[3] A cool and practical alternative to traditional hash tables, U. Erlingsson, M. Manasse, F. Mcsherry, 2006.

[4] McIlroy, Peter M., Keith Bostic, and M. Douglas McIlroy. "Engineering radix sort." Computing systems 6.1 (1993): 5-27.