New Sorting Patch Test Report

1. Environment Specification

Amazon EC2 Instance

instance type: t2.micro

vCPUs: 1

RAM: 1.0 GiB

OS: Ubuntu 16.04.3 LTS

2. Test Steps

Standalone Test

1. Clone the *Dual* branch from https://github.com/Strider-Alex/PostgreSQLSorting
2. Compile the source code and run the executable.
3. The executable generates test data files and sorts arrays of different patterns using original qsort and the new qsort implemented with intro sort. The minimum and maximum data size is specified in *benchmark.h*.

Array patterns specification:

- Sorted: already sorted array
- Random: random array
- Reversed: reversely sorted array
- Mostly sorted: divide a sorted array into bins of size 10, then randomly shuffle these bins
- Most reversed: divide a reversed array into bins of size 10, then randomly shuffle these bins
- Killer sequence: a specially generated sequence that makes qsort reach n^2 time complexity

Pgbench Test

1. Generate initialization SQL script from the test data files in the Standalone Test. These initialization files create a test table and insert simple records with only one integer field into the table. Or you can use the SQL scripts I provided.
2. Clone the source code for PostgreSQL from https://github.com/postgres/postgres
3. Compile the *master* branch, install and start the server.
4. Create database *test* if not exits.
5. Run an initialization SQL script on database *test* by the following command:
   psql -d test -f <init SQL script>
6. Benchmarking the test script on data base *test* by the following command:
   pgbench test -c 10 -f <test script>
   The test script simply selects all entries from the test table and order them by values.
7. Apply the patch of the new sorting routine, then re-compile, install and restart server.

8. Apply Step 5 and 6.

(I may write a shell script to make previous steps easier)

3. Test Result

Notice: green indicates better performance; red indicates worse.

Standalone test – intro sort – CPU clocks

	Sorted	Random	Reversed	Mostly sorted	Mostly reversed	Killer sequence
N=1000	5.26	166.18	141.76	130.60	134.00	381.52
N=10000	51.00	2326.80	1716.89	1573.55	1565.61	5578.46
N=100000	505.54	29299.04	15225.53	17743.78	17758.34	72358.99

Standalone test – pg qsort – CPU clocks

	Sorted	Random	Reversed	Mostly sorted	Mostly reversed	Killer sequence
N=1000	5.56	170.35	126.84	131.28	137.85	412.30
N=10000	50.91	2384.32	1563.97	1545.40	1527.69	33236.13
N=100000	514.35	29311.91	15768.52	16542.25	16760.67	2264115.75

Pgbench – intro sort – transaction per second (excluding connections establishing)

	Random	Killer sequence
N=1000	1571.911076	1360.387390
N=10000	215.382229	158.025610
N=100000	15.325185	16.671237

Pgbench – pg qsort – transaction per second (excluding connections establishing)

	Random	Killer sequence
N=1000	1535.237516	1236.217073
N=10000	204.818081	40.568045
N=100000	15.295583	16.600637

Notice: when N=100000, tuplesort switches to external sorting algorithms instead of using qsort, so the result is very similar.

4. Conclusion

New intro sort-based sorting routine has slightly better performance than pg_qsort on sorting random data. Also, since it's has guaranteed $O(nlogn)$ worst case time complexity, the new sorting routine dramatically outperforms pg_qsort on specially generated killer sequence.

On the other hand, pg_qsort seems to be slightly better on sorting mostly sorted data. This is probably because it checks if the array is pre-sorted on every recursion. However, in the new sorting routine, we only check if the array is pre-sorted once on the whole array, which gives us better performance on random data.