Enhancing Data Generation in TPCx-HS with a Non-uniform Random Distribution

Raghunath Nambiar[1(✉)], Tilmann Rabl[2], Karthik Kulkarni[1], and Michael Frank[3]

[1] Cisco Systems, Inc., 275 East Tasman Drive, San Jose, CA 95134, USA
{rnambiar, kakulkar}@cisco.com

[2] University of Toronto, 27 King's College Circle, Toronto, ON M5S, Canada
tilmann.rabl@utoronto.ca

[3] Bankmark, Bahnhofstrasse 10, 94032 Passau, Germany
michael.frank@bankmark.de

Abstract. Developed by the Transaction Processing Performance Council, the TPC Express Benchmark™ HS (TPCx-HS) is the industry's first standard for benchmarking big data systems. It is designed to provide an objective measure of hardware, operating system and commercial Apache Hadoop File System API compatible software distributions, and to provide the industry with verifiable performance, price-performance and availability metrics [1, 2]. It can be used to compare a broad range of system topologies and implementation methodologies of big data systems in a technically rigorous and directly comparable and vendor-neutral manner. The modeled application is simple and the results are highly relevant to hardware and software dealing with Big Data systems in general. The data generation is derived from TeraGen [3] which uses uniform distribution of data. In this paper the authors propose normal distribution (Gaussian distribution) which may be more representative of real life datasets. The modified TeraGen and complete changes required to the TPCx-HS kit are included as part of this paper.

Keywords: TPC · Big data · Industry standard · Benchmark

1 TPCx-HS Current State

As Big Data technologies like Hadoop have become an integral part of enterprise IT ecosystem across all major industry verticals, industry standard benchmarks that can fairly compare technologies and products are critical [4]. Big Data was identified as critical area for benchmarking at conferences such as TPCTC 2013 [5]. Keep this in mind the Transaction Processing Performance Council (TPC) developed TPC Express Benchmark™HS (TPCx-HS) that provides an objective measure of hardware, operating

© Springer International Publishing Switzerland 2016

DOI: 10.1007/978-3-319-31409-9_7

system and commercial Apache Hadoop File System API compatible software distributions. TPCx-HS provides the industry with verifiable performance, price-performance and availability metrics. The benchmark models a continuous system availability of 24 h a day, 7 days a week [1, 6, 7].

Even though the modeled application is simple, the results are highly relevant to hardware and software dealing with Big Data systems in general. The TPCx-HS stresses both hardware and software including Hadoop runtime, Hadoop File System API compatible systems and MapReduce layers. This workload can be used to asses a broad range of system topologies and implementation of Hadoop clusters. The TPCx-HS can be used to assess a broad range of system topologies and implementation methodologies in a technically rigorous and directly comparable, in a vendor-neutral manner [4–6].

TPCx-HS workload is based on TeraSort [1] designed to evaluate the sorting performance of a system-under-test (SUT) [2]. This is highly relevant for every big data system because sorting is a basic operation required in many high level abstractions like ordering, grouping, and joining. The dataset of TPCx-HS consists of records of 100 Byte length where the first 10 Bytes of each record is the sorting key. The keys are distributed uniformly and randomly over the key space. Because of the large key space (2^80 possible keys), duplicate keys are very unlikely and the key space is sparsely populated. In most real world data sets keys are either of sequential type indexing the row of a table or they are referencing other tuples. In both cases, keys are almost never uniformly distributed over the key space. Depending on the data set properties, keys are dense in some areas (e.g., a sequence with some missing keys) and sparse in others or some keys occur more often than others (e.g., in a purchase table one product ID will be bought more frequently than others). The underlying system often is not aware of the nature of data, but its properties like ordering, density, sparsity, and duplication are important to perform efficient sorting.

2 Extending TPCx-HS

TPCx-HS uses 100 Byte records, where the first 10 Bytes are the keys on which the data is sorted and the remaining 90 Bytes are random payload. TPCx-HS has two implementations –for MapReduce 1 (MR1) and for MapReduce 2 (MR2). They differ in the way random numbers are generated and in the key layout.

2.1 MR1 Implementation

The MR1 version uses a 64 Bit linear congruential generator (LCG) based random number generator. To generate each key, three random numbers are drawn. Each 64 Bit random number is split into four bytes. The last random number only populates two bytes, resulting in a total of 10 bytes per key. During the split the byte value is mapped to the range of the 95 printable ASCII characters. Because of this reduction the total number of distinct keys is only $10^{95} \cong 2^{66}$.

2.2 MR2 Implementation

The MR2 version uses a 128 Bit LCG random number generator. To generate a key, a single 128 Bit random number is drawn. From that 128 Bit random number the highest 80 Bits are used for the key. Unlike the MR1 version, the keys remain in the binary format and cover the whole 2^{80} key range.

Both versions split the random number sequence based on the current mapper row for parallelization. The sequence is pre-split and the pre-calculated seeds for these splits are stored inside TPCx-HS' random number generator. This splitting schema only works if the same amount of random numbers is generated for each row. The MR1 version requires exactly three 64 Bit random numbers and the MR2 version exactly one 128 Bit random number to generate a key.

2.3 Proposed Changes

Main change proposed is the key distribution from uniform distribution to normal distribution (Gaussian distribution). The normal distribution is typically used to describe independent random processes such as growth, income, and measurement errors. A typical example for a uniformly distributed random event is a single fair roll of a die, each side of the die has the same probability and, thus, the results will be uniformly distributed. However, the probability distribution of the sum of multiple rolls converges to a normal distribution. As few natural observations consist of a single random event but rather of a process of random events, the normal distribution is more frequently found than a uniform distribution in real and is the most important distribution in statistics[1].

In order to change the key distribution in TeraGen, an interface was introduced that enables plugging in and parametrizing different ways of generating the key. This was done in the following classes and functions, `HsGen.SortGenMapper.addKey()` in the MR1 version and `HsGen.SortGenMapper.map()` and `GenSort.GenerateRe-cord()` in the MR2 version.

The code of the existing uniform key-generation implementation was copied into a separate plugin. The uniform key generation plugin is used as default, if no other key generation strategy is specified. Additional changes were made to enable the specification and configuration of the key generation strategy via command line. An example

[1] Cf., http://www.itl.nist.gov/div898/handbook/pmc/section5/pmc51.htm.

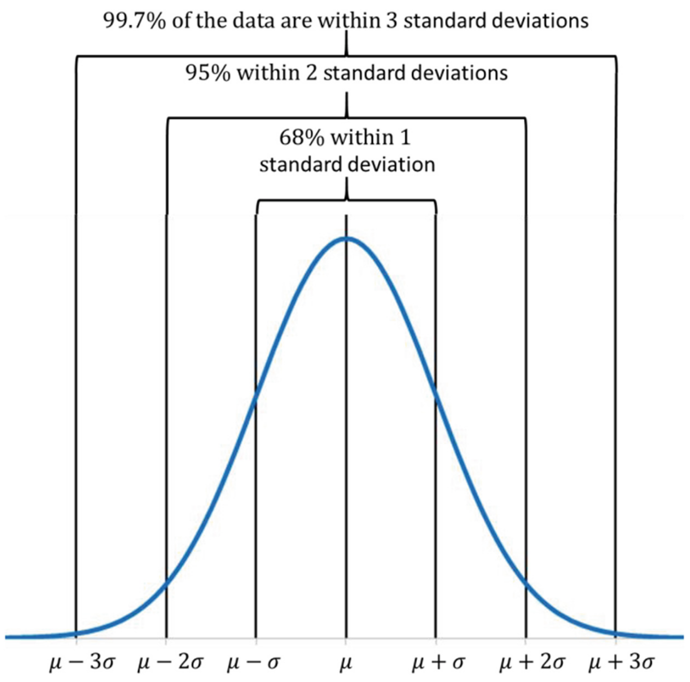

Fig. 1. Normal distribution, its parameters and their implications (Based on Dan Kernler, http://en.wikipedia.org/wiki/File:Empirical_Rule.PNG)

will be given below. Besides the existing default implementation for uniform key generation, an implementation to produce normal distributed keys (Gaussian distribution) was added for both the MR1 and MR2 version of TPCx-HS HsGen:

- `org.tpc.hs.hsgen.mr1.dist.NormalDistributionBigInt`
- `org.tpc.hs.hsgen.mr2.dist.NormalDistributionBigInt`

A normal distribution is parametrized by two values. A mean value μ (mu) and the standard deviation σ (sigma). A visual representation of the normal distribution and the meaning of its parameters can be seen in Fig. 1. The implementation uses a standard polar Box-Muller algorithm for generating normally distributed values using two values sampled from a uniform random number source. Modifications had to be made to scale the algorithm producing 64 Bit double values to the required 80 Bit BigInteger values.

```
double SCALE_D = Long.MAX_VALUE / 10;
BigInteger SCALE_BI = BigInteger.valueOf(Long.MAX_VALUE / 10);

public static BigInteger normalDistributedBigInteger(BigInteger mu_bi,
              BigInteger sd_bi,BigInteger maxValue, HelperPRNG rng) {
 do {
  // Polar box-muller
  double x, y, r, z;
  do {
   x = 2.0 * rng.nextDouble() - 1.0;
   y = 2.0 * rng.nextDouble() - 1.0;
   r = x * x + y * y;
  } while (r >= 1.0);
  z = Math.sqrt(-2.0 * Math.log(r) / r);
  double normalDistNumber = y * z; // range: ~ -5.0/+5.0

  // scale "normalDistNumber" with specified mean (mu)
  //and standard deviation (sd) values

  // naive version, using BigDecimal
  // Formula: res = mu + sd * normalDistNumber
  // Range of mu and sd is [0, 2^80]
  // res = new BigDecimal(mu_bi).add(new BigDecimal(sd_bi)
           .multiply(BigDecimal.valueOf(normalDistNumber))).toBigInteger();

  // we can convert the number to long by scaling up
  // to Long.MaxValue. As the value range of normalDistNumber is
  // ~ -5.0/+5.0 we can optimistically multiply normalDistNumber with:
  // Long.MaxValue/10. After the calculation we have to revert the
  // scaling by dividing by Long.MaxValue/10.
  BigInteger normScaled = BigInteger.valueOf((long)(normalDistNumber * SCALE_D));
  res = mu_bi.add(sd_bi.multiply(normScaled).divide(SCALE_BI));
 } while (res.compareTo(KEY_RANGE) > 0 || res.compareTo(BigInteger.ZERO) < 0);
 return res;
}
```

Listing 1 NormalDistribution – Polar box-muller method

```
long s0, s1; //internal state of HelperPRNG
public void seed(long... seeds) {
 s0 = seeds[0];
 s1 = seeds[1];
  nextLong();
}
/*
 * Xor-shift PRNG with a period of 2^128
 * (c) by Sebastiano Vigna (vigna@acm.org)
 * To the extent possible under law, the author has dedicated all copyright
 * and related and neighboring rights to this software to the public domain
 * worldwide. This software is distributed without any warranty.
 * See <http://creativecommons.org/publicdomain/zero/1.0/>.
 * source: http://xorshift.di.unimi.it/xorshift128plus.c
 */
public long nextLong() {
  final long s0 = this.s0;
  long s1 = this.s1;
  this.s0 = s1;
  s1 ^= s1 << 23;
  return (this.s1 = (s1 ^ s0 ^ (s1 >>> 17) ^ (s0 >>> 26))) + s0;
}
double TWO_POW_64_INV = (1.0 / Math.pow(2, 64));
public double nextDouble() {
  return nextLong() * TWO_POW_64_INV + 0.5:
}
```

Listing 2 HelperPRNG – xorShift128

The nature of this algorithm is to randomly require more the two random values, if certain conditions are not met (see Listing 1 NormalDistribution – Polar box-muller method). Because of this requirement, the default random number generators of TPCx-HS cannot be used, as they only work for if a predefined amount of random numbers is drawn per row as described earlier. Because of this issue, a helper random number generator is used (see Listing 2 HelperPRNG – xorShift128), which is seeded with the random number for each row, obtained from the main random number generator. This allows to draw an arbitrary amount of random samples in each row, as required by the normal distribution algorithm. The implementation was verified to work correctly and the verification results can be seen in Fig. 2. The plot shows the accumulated density distribution of $\sim 6*10^7$ samples, drawn from the implemented normal distribution with parameters mu: 2^{79} and sigma: 2^{76}. To be able to count and plot the values, they were quantized to the range [0, 65536] by using the highest 16 Bit of the full 80 Bit value, resulting in mu: 32768 and sd: 4096. Additionally a wrapper was added for all the tools in the tpcx-hs.jar to comfortably run the TPCx-HS benchmark end to end. Available are the following modules, where 1 stands for the MR1 version and 2 for the MR2 version:

- hsgen1
- hsgen2

- hssort1
- hssort2
- hsvalidate1
- hsvalidate2

An example on how to start and parametrize the generation of normally distributed keys can be seen in the following listing:

```
hadoop jar tpcx-hs.jar hsgen1
  -D Hssort.distribution=org.tpc.hs.hsgen.mr1.dist.NormalDistributionBigInt
  -D Hssort.distribution.mu=29936846961918945312
  -D Hssort.distribution.sd=1000000000 100000000000 tpcxhs/terasort-input
```

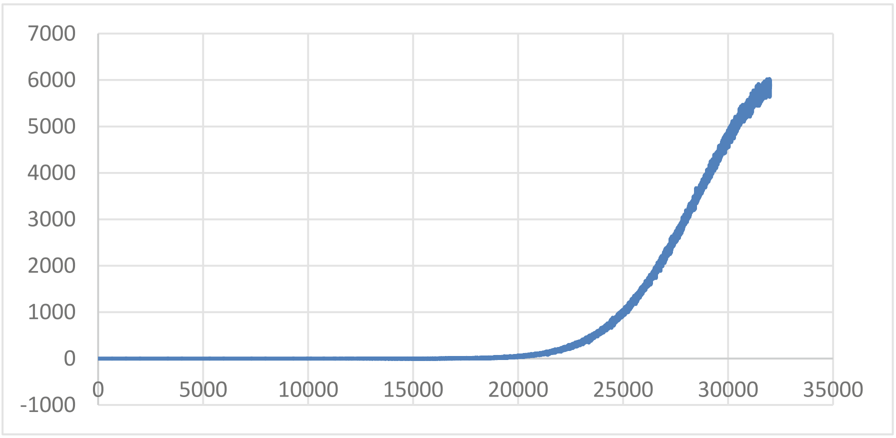

Fig. 2. Accumulated density with mu: 2^{79} sigma: 2^{76} quantized from highest 16 bit of the full 80 bit value to mu: 32768 and sigma: 4096

3 Test Results

We compared the impact on data generation and sorting performance using uniformly and normally distributed keys. The tests where done both for the MR1 and MR2 version of TPCx-HS. We ran a test with a data set size of 9,09 TB (100.000.000.000 records).

For the test runs using the new normal distributed key schema we used the following parameters:

Mu	29.936.846.961.918.900.000
Sigma	1.000.000.000

SD is chosen to be 1/100 of the number of records, to ensure a high number of duplicates within the generated keys. All tests were performed on the same 16 node cluster with the following specification:

Servers: 16 Cisco UCS C240M3 Rack Server
CPU: 2 × Intel® Xeon® Processor E5-2650 v2 (20 M Cache, 2.60 GHz)
Memory: 256 GB
Storage Controller: LSI MegaRAID SAS 9271-8i
Disk: 12 × 3 TB Large Form Factor HDD
Network: Cisco UCS VIC 1225 2 10GE SFP+

Software configuration:

- Disabled SElinux on all the nodes
- Disabled iptables on all the nodes
- NTP configured
- Ulimit set to 64000

A default Hadoop installation using Yarn was used. Looking at the CPU resources, it is obvious that generating non-uniform values is more CPU intensive than generating uniform random values, as shown in Fig. 4. The MapReduce 1 version is more expensive to generate than the MapReduce 2 version because of the additional step of converting the binary key into an ASCII representation. However, there was not much difference in real clock time as shown in Fig. 3, because the cluster was I/O bound the entire time.

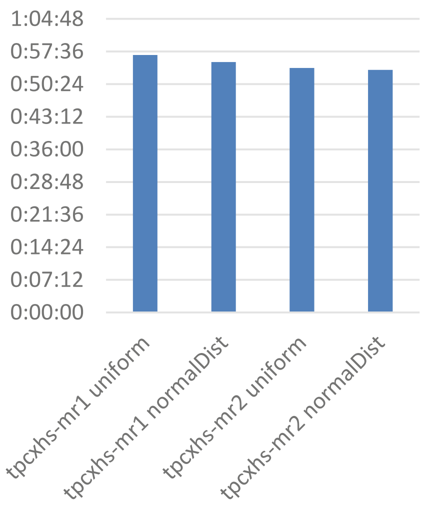

Fig. 3. HSGen wall clock time

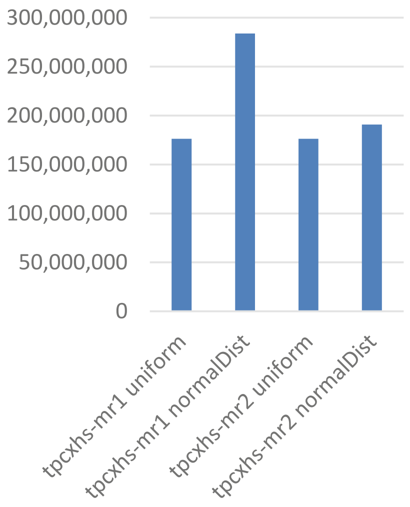

Fig. 4. HSGen CPU time

The interesting result is the impact of the non-uniform data on the sort performance of TeraSort (Fig. 5). It shows that for both the MR1 and MR2 versions, the skewed dataset takes more time to sort than the uniformly distributed dataset. The difference is most notable when comparing the CPU times in Fig. 6. The impact was not as severe as initially expected. TPCx-HS employs techniques to mitigate skewed datasets. It does so by employing a custom split format and partitioning logic. This logic draws random

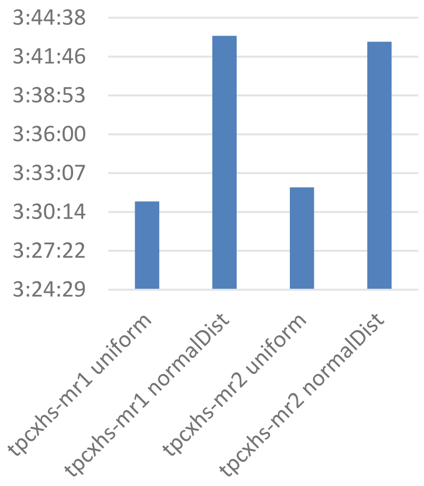

Fig. 5. HSSort wall clock time

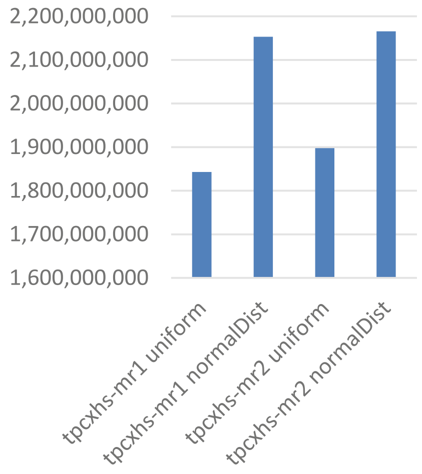

Fig. 6. HSSort CPU time

samples from the generated data and sorts the samples locally to get an estimate of the key distribution. Based on this sample, TPCx-HS' sorting logic pre-partitions the data and distributes it to the mappers.

4 Conclusion

In this paper, the authors propose a potential enhancement to TPCx-HS benchmark with a normally distributed keys. The impact on data generation and sorting performance is compared against uniformly distributed keys.

The test results demonstrate that the distribution of keys has an impact on the data generation and data sorting performance of the system. The authors believe that non-uniform data distributions are more representative of real life datasets and a good enhancement to the next generation of TPCx-HS.

5 Future Work

TTPCx-HS can further extend with more distributions that fit data skew commonly found in production workloads. An example might be the Zipfian distribution, which models an access pattern where some few keys are accessed very often and most of the keys rarely to almost never. Another interesting addition would be the simulation of "null" values or more generally a single key that accounts for a high percentage of the total keys. Null values impose a great scaling problem, for example, in Hive queries, if not addressed properly because many parallel algorithms rely on distributing the values by key. If a single key occurs in, for example, 25 % of all data sets, a single mapper/reducer ends up processing 25 % of the dataset by itself, greatly reducing speedup in a large cluster.

Acknowledgements. The authors thank the contributors of the original TPCx-HS development committee, Andrew Bond (Red Hat), Andrew Masland (NEC), Avik Dey (Intel), Brian Caufield (IBM), Chaitanya Baru (SDSC), Da Qi Ren (Huawei), Dileep Kumar (Cloudera), Jamie Reding

(Microsoft), John Fowler (Oracle), John Poelman (IBM), Karthik Kulkarni (Cisco), Meikel Poess (Oracle), Mike Brey (Oracle), Mike Crocker (SAP), Paul Cao (HP), Reza Taheri (VMware), Simon Harris (IBM), Tariq Magdon-Ismail (VMware), Wayne Smith (Intel), Yanpei Chen (Cloudera), Michael Majdalany (L&M), Forrest Carman (Owen Media), and Andreas Hotea (Hotea Solutions). Thanks to Manankumar Trivedi for his support with benchmark testing and analysis.

Authors also thank Satinder Sethi for his guidance and support with this effort.

Appendix

Command lines used to conduct the experiments

Hsgen MR1 normal distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hsgen1 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hsgen__tpcxhs_mr1_NormalDistributionBigInt_2993684696
1918945312_1000000000 -D mapred.map.tasks=512 -D
Hssort.distribution=org.tpc.hs.hsgen.mr1.dist.NormalDistributionBigIn
t -D Hssort.distribution.mu=29936846961918945312 -D
Hssort.distribution.sd=1000000000 100000000000 tpcxhs/terasort-input
```

Hsgen MR1 uniform distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hsgen1 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hsgen__tpcxhs_mr1_UniformDistribution -D
mapred.map.tasks=512 -D
Hssort.distribution=org.tpc.hs.hsgen.mr1.dist.UniformDistribution -D
Hssort.distribution.mu=29936846961918945312 -D
Hssort.distribution.sd=1000000000 100000000000 tpcxhs/terasort-input
```

Hsgen MR2 normal distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hsgen2 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hsgen__tpcxhs_mr2_NormalDistributionBigInt_2993684696
1918945312_1000000000 -D mapred.map.tasks=512 -D
Hssort.distribution=org.tpc.hs.hsgen.mr2.dist.NormalDistributionBigIn
t -D Hssort.distribution.mu=29936846961918945312 -D
Hssort.distribution.sd=1000000000 100000000000 tpcxhs/terasort-input
```

Hsgen MR1 uniform distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hsgen2 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hsgen__tpcxhs_mr2_UniformDistribution -D
mapred.map.tasks=512 -D
Hssort.distribution=org.tpc.hs.hsgen.mr2.dist.UniformDistribution -D
Hssort.distribution.mu=29936846961918945312 -D
Hssort.distribution.sd=1000000000 100000000000 tpcxhs/terasort-input
```

Hssort MR1 normal distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hssort1 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hssort__tpcxhs_mr1_NormalDistributionBigInt_299368469
61918945312_1000000000 -D mapred.reduce.tasks=512 tpcxhs/terasort-
input/ tpcxhs/terasort-sorted/
```

Hssort MR1 uniform distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hssort1 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hssort__tpcxhs_mr1_UniformDistribution -D
mapred.reduce.tasks=512 tpcxhs/terasort-input/ tpcxhs/terasort-
sorted/
```

Hssort MR2 normal distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hssort2 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hssort__tpcxhs_mr2_NormalDistributionBigInt_299368469
61918945312_1000000000 -D mapred.reduce.tasks=512 tpcxhs/terasort-
input/ tpcxhs/terasort-sorted/
```

Hssort MR1 uniform distribution

```
hadoop jar /root/tpcx-hs/tpcx-hs.jar hssort2 -D
mapreduce.map.speculative=false -D mapreduce.reduce.speculative=false
-D dfs.client.block.write.replace-datanode-on-failure.policy=NEVER -D
mapred.job.name=hssort__tpcxhs_mr2_UniformDistribution -D
mapred.reduce.tasks=512 tpcxhs/terasort-input/ tpcxhs/terasort-
sorted/
```

```
package org.tpc.hs.hsgen;

import org.apache.hadoop.util.ProgramDriver;

/**
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 27.02.2015
 */
public class Driver {

 public static void main(String argv[]) {
  int exitCode = -1;
  ProgramDriver pgd = new ProgramDriver();
  try {
   pgd.addClass("hsgen1",
     org.tpc.hs.hsgen.util.mr1.HSGen.class,
     "Generate data for the hssort using mr1");
   pgd.addClass("hssort1",
     org.tpc.hs.hsgen.util.mr1.HSSort.class,
     "Run the hssort using mr1");
   pgd.addClass("hsvalidate1",
     org.tpc.hs.hsgen.util.mr1.HSValidate.class,
     "Checking results of hssort using mr1");
   pgd.addClass("hsgen2",
     org.tpc.hs.hsgen.util.mr2.HSGen.class,
     "Generate data for the hssort using mr2");
   pgd.addClass("hssort2",
     org.tpc.hs.hsgen.util.mr2.HSSort.class,
     "Run the hssort using mr2");
   pgd.addClass("hsvalidate2",
     org.tpc.hs.hsgen.util.mr2.HSValidate.class,
     "Checking results of hssort using mr2");
   exitCode = pgd.run(argv);
  } catch (Throwable e) {
   e.printStackTrace();
  }
  System.exit(exitCode);
 }
}
```

```
package org.tpc.hs.hsgen.util;
/**
 * TeraGen random number system does not allow to sample
 * arbitrary numbers of random values for a single record. Because of
this we
```

```
 * use the limited available random number as a seed for this helper
PRNG
 *
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 27.02.2015
 */
public class HelperPRNG {

 private static final double TWO_POW_64_INV = (1.0 / Math.pow(2,
64));

 private long s0, s1; // internal state

 /**
  * seed array, arbitrary length but only first 2 bytes are used
  *
  * @param seeds
  * @return this
  */
 public HelperPRNG seed(long... seeds) {
  // seed =seeds[0];
  s0 = seeds[0];
  s1 = seeds[1];
  // apply one round of mixing
  // in case of bad input seed entropy (e.g. never negative), first
  // rand128() result would also be never negative
  rand128();
  return this;
 }

 public long nextLong() {
  return rand128();
 }

 /*--
  * Xor-shift PRNG with a period of 2^128
  * (c) by Sebastiano Vigna (vigna@acm.org)
  * To the extent possible under law, the author has dedicated all
copyright
  * and related and neighboring rights to this software to the public
domain
  * worldwide. This software is distributed without any warranty.
  * See <http://creativecommons.org/publicdomain/zero/1.0/>
  * http://xorshift.di.unimi.it/xorshift128plus.c
  */
 private long rand128() {
  final long s0 = this.s0;
  long s1 = this.s1;
  this.s0 = s1;
  s1 ^= s1 << 23;
  return (this.s1 = (s1 ^ s0 ^ (s1 >>> 17) ^ (s0 >>> 26))) + s0;
 }

 public static final double randLongToDouble(long r) {
  return r * TWO_POW_64_INV + 0.5;
 }

 public double nextDouble() {
  return randLongToDouble(nextLong());
 }
}
```

```
package org.tpc.hs.hsgen.util;
import java.math.BigInteger;

/**
 * <DISTRIBUTION> Helper for HSGen to do BigInteger conversion
 *
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 02.03.2015
 */
public class HsgenHelper {
 public static final BigInteger LONG_MAX_BI =
BigInteger.valueOf(Long.MAX_VALUE);
 public static final int KEY_SIZE_IN_BYTES = 10;
 public static final BigInteger KEY_RANGE_BINARY_BI =
BigInteger.ONE.shiftLeft(
   KEY_SIZE_IN_BYTES * 8).subtract(BigInteger.ONE);

 public static final int KEY_TO_STRING_RADIX = 95;
 public static final BigInteger KEY_TO_STRING_RADIX_BI =
BigInteger.valueOf(KEY_TO_STRING_RADIX);
 public static final BigInteger KEY_RANGE_ASCII_BI =
KEY_TO_STRING_RADIX_BI.pow(
   HsgenHelper.KEY_SIZE_IN_BYTES).subtract(BigInteger.ONE);
 /**
  * Convert a BigInteger value to bytes in radix 95 and + offset ' '
Storage
  * of the 10 key bytes in Little endian format. e.g.
  *
  * <pre>
  * <code>
  * value =
  *         95^9 * keyBytes[9]
  *       + 95^8 * keyBytes[8]
  *       + 95^7 * keyBytes[7]
  *       + 95^6 * keyBytes[6]
  *       + 95^5 * keyBytes[5]
  *       + 95^4 * keyBytes[4]
  *       + 95^3 * keyBytes[3]
  *       + 95^2 * keyBytes[2]
  *       + 95^1 * keyBytes[1]
  *       + 95^0 * keyBytes[0]
  * </code>
  * </pre>
  *
  * @param key
  * @param res
  */
 public static void toRadix_LE(BigInteger res, byte[] keyBytes) {
  BigInteger tmp = res;

  for (int i = 0; i < HsgenHelper.KEY_SIZE_IN_BYTES; i++) {
   BigInteger[] qrem =
tmp.divideAndRemainder(HsgenHelper.KEY_TO_STRING_RADIX_BI);
   // System.out.println(i + ": "+res+" divmod " + RADIX_BI +
   // " = "
   // + qrem[0] + " rem: "
   // + qrem[1] + " asint: " + qrem[1].intValue() + " val: "
   // + (char) (' ' + qrem[1].intValue()));
   tmp = qrem[0];
   int rem = qrem[1].intValue();

   keyBytes[i] = (byte) (' ' + rem);
  }
 }

 /**
  * Convert a BigInteger value to bytes in radix 95 and + offset ' '
Storage
  * of the 10 key bytes in BigEndian format. e.g.
  *
  * <pre>
  * <code>
  * value =
  *         95^9 * keyBytes[0]
  *       + 95^8 * keyBytes[1]
  *       + 95^7 * keyBytes[2]
  *       + 95^6 * keyBytes[3]
  *       + 95^5 * keyBytes[4]
  *       + 95^4 * keyBytes[5]
  *       + 95^3 * keyBytes[6]
  *       + 95^2 * keyBytes[7]
  *       + 95^1 * keyBytes[8]
  *       + 95^0 * keyBytes[9]
  * </code>
  * </pre>
  *
  * @param res
  * @param key
```

```
   */
  public static void toRadix_BE(BigInteger res, BigInteger radix,
byte[] resultBuffer,
    int maxLen, char addToRadixOffset) {
   int KEY_TO_STRING_RADIX = radix.intValue();
   BigInteger tmp_BI = res;

   long tmp_l;
   int i = maxLen - 1;

   // convert radix 10 value to radix 95 bytes
   // use BigInteger arithmetic only as long as needed
   // if tmp value can fit into a long, switch to faster native
   // algorithm
   for (; i >= 0 && tmp_BI.compareTo(LONG_MAX_BI) > 0; i--) {
    BigInteger[] qrem = tmp_BI.divideAndRemainder(radix);
    tmp_BI = qrem[0];
    int rem = qrem[1].intValue();

    resultBuffer[i] = (byte) (addToRadixOffset + rem);
   }

   if (i >= 0) {
    tmp_l = tmp_BI.longValue();
    for (; i >= 0; i--) {
     long rem = tmp_l % KEY_TO_STRING_RADIX;
     tmp_l = tmp_l / KEY_TO_STRING_RADIX;
     resultBuffer[i] = (byte) (addToRadixOffset + rem);
    }
   }
  }
}
```

```
package org.tpc.hs.hsgen.distribution;

import java.math.BigInteger;

import org.tpc.hs.hsgen.util.HelperPRNG;

/**
 * @author Michael
 * @version 1.0 01.11.2015
 */
public class Normal {

 // static helper values for normal distribution
 private static final long SCALE_L = Long.MAX_VALUE / 10;
 private static final double SCALE_D = SCALE_L;
 private static final BigInteger SCALE_BI =
BigInteger.valueOf(SCALE_L);
 private static final BigInteger LONG_MAX_BI =
BigInteger.valueOf(Long.MAX_VALUE);

 public static BigInteger normalDistributedBigInteger(BigInteger
mu_bi, BigInteger sd_bi,
   BigInteger maxValue, HelperPRNG rng) {

  BigInteger res;
  do {
   /*
    * assumption is: 64 Precision for calculating the standard
    * deviation should still be sufficient
    */
   // Polar box-muller
   double x, y, r, z;
   do {

    x = 2.0 * rng.nextDouble() - 1.0;
    y = 2.0 * rng.nextDouble() - 1.0;
    r = x * x + y * y;
   } while (r >= 1.0);
   z = Math.sqrt(-2.0 * Math.log(r) / r);
   // res = mean + sd * y * z;
   double normalDistNumber = y * z; // range: -5.0/+5.0

   // ##################################################
   // scale normalDistNumber with specified mean (mu) and
   // standard deviation (sd) values
   // ##################################################

   // Variant one: naive version
   // Formula: res = mu + sd * normalDistNumber
   // Range of mu and sd is [0, 2^80] -> use BigDecimal
   // res = new BigDecimal(mu_bi).add(new
   //
BigDecimal(sd_bi).multiply(BigDecimal.valueOf(normalDistNumber))).toB
igInteger();

   /*
    * Optimization: As the range of mu and sd is [0, 2^80] we would
    * need to use BigDecimal to calculate:
    *
    * res=mu + sd * normalDistNumber
    *
    * But we can convert the number to long by scaling up (normalize)
    * to Long.MaxValue. As the value range of normalDistNumber is ~
    * -5.0/+5.0 we can optimistically multiply normalDistNumber with:
    * Long.MaxValue/10. After the calculation we have to revert the
    * scaling (de-normalize) by dividing by Long.MaxValue/10.
    *
    * Hint: Implementing and using a UInt128 Datatype (or usage of
the
    * Unsigned16 class from the hsgen.mr2 package) did not show
    * performance gains, as BigIntegers algorithms are very well
    * optimized
    */
   BigInteger normScaled = BigInteger.valueOf(((long)
(normalDistNumber * SCALE_D)));

   // scale to mu and sd values
   res = mu_bi.add(sd_bi.multiply(normScaled).divide(SCALE_BI));
  } while (res.compareTo(maxValue) > 0 ||
res.compareTo(BigInteger.ZERO) < 0);
  return res;

 }
}
```

```
package org.tpc.hs.hsgen.mr1.dist;

import java.util.Arrays;
import java.util.List;

import org.apache.hadoop.mapred.JobConf;

/**
 *
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 27.02.2015
 */
public class DistributionFactory {

 public static class DistributionException extends Exception {
  public DistributionException(String msg, Throwable cause) {
   super(msg, cause);
  }
 }

 // CMD_LINE/JobConf key
 public static final String HSSORT_DISTRIBUTION =
"Hssort.distribution";

 // ##############################################################
 // Add statically know distributions here:
 // you can always load any distribution class via reflection
 // #############################################################
 public static final List<Class<? extends Distribution>> dists = Ar-
rays.asList(
   UniformDistribution.class, NormalDistributionBigInt.class);

 public static Distribution getDistribution(JobConf job) throws
DistributionException {
  String distributionClassName = Distribu-
tion.getStringFromConf(HSSORT_DISTRIBUTION,
    NormalDistributionBigInt.class.getName(), job);

  System.out.println("getConf(\"" + HSSORT_DISTRIBUTION + "\", " +
dists.get(0).getName()
    + ")=" + distributionClassName);
  System.out.println("loading distribution: " + distributionClassName
+ " ...");

  Distribution dist;
  try {
   dist = (Distribution)
Class.forName(distributionClassName).newInstance();
  } catch (Exception e) {
   throw new DistributionException("Could not load distribution
class. Cause:  "
     + e.getClass().getName() + ": " + e.getMessage() + "\nAvailable
distributions:"
     + dists, e);
  }

  dist.initialize(job);
  System.out.println("loaded distribution " + dist.toString());
  return dist;
 }
}
```

```
package org.tpc.hs.hsgen.mr1.dist;

import java.math.BigInteger;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.JobConf;
import org.tpc.hs.hsgen.mr1.HSGen;

/**
 *
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 27.02.2015
 */
public abstract class Distribution {
 public static BigInteger getBigIntegerFromConf(String name,
BigInteger defaultValue, JobConf job) {
  String option = job.get(name, defaultValue.toString());
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return new BigInteger(option);
 }

 public static double getDoubleFromConf(String name, double
defaultValue, JobConf job) {
  double option = job.getDouble(name, defaultValue);
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return option;
 }

 public static boolean getBooleanFromConf(String name, boolean
defaultValue, JobConf job) {
  boolean option = job.getBoolean(name, defaultValue);
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return option;
 }

 public static String getStringFromConf(String name, String
defaultValue, JobConf job) {
  String option = job.get(name, defaultValue.toString());
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return option;
 }

 public abstract void initialize(JobConf c);

 /**
  * Contract of HSGen.RandomGenerator rand is: make exactly! 3 calls
to
  * rand.next(). If you require more random numers, use a helper PRNG
e.g.:
  *
  * <pre>
  * HelperPRNG helperRNG;
  * helperRNG.seed(rand.next(), rand.next(), rand.next()); </ore>
  *
  * @param key
  * @param rand
  */
 public abstract void generateKey(Text key, HSGen.RandomGenerator
rand);
}
```

```
package org.tpc.hs.hsgen.mr1.dist;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.JobConf;
import org.tpc.hs.hsgen.mr1.HSGen;
import org.tpc.hs.hsgen.util.HsgenHelper;

/**
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 27.02.2015
 */
public class UniformDistribution extends Distribution {

 public UniformDistribution() {
 }

 // reuseable buffer
 private byte[] keyBytes = new byte[HsgenHelper.KEY_SIZE_IN_BYTES +
2];

 @Override
 public void generateKey(Text key, HSGen.RandomGenerator rand) {
  // generates 12 bytes, later truncate to 10
  for (int i = 0; i < 3; i++) {
   long temp = rand.next() / 52;
   keyBytes[3 + 4 * i] = (byte) (' ' + (temp %
HsgenHelper.KEY_TO_STRING_RADIX));
   temp /= HsgenHelper.KEY_TO_STRING_RADIX;
   keyBytes[2 + 4 * i] = (byte) (' ' + (temp %
HsgenHelper.KEY_TO_STRING_RADIX));
   temp /= HsgenHelper.KEY_TO_STRING_RADIX;
   keyBytes[1 + 4 * i] = (byte) (' ' + (temp %
HsgenHelper.KEY_TO_STRING_RADIX));
   temp /= HsgenHelper.KEY_TO_STRING_RADIX;
   keyBytes[4 * i] = (byte) (' ' + (temp %
HsgenHelper.KEY_TO_STRING_RADIX));
  }
  key.set(keyBytes, 0, HsgenHelper.KEY_SIZE_IN_BYTES);
 }

 @Override
 public void initialize(JobConf c) {
  // nothing to do
 }

 @Override
 public String toString() {
  return this.getClass().getSimpleName() + " []";
 }
}
```

```
package org.tpc.hs.hsgen.mr1.dist;

import java.math.BigInteger;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.JobConf;
import org.tpc.hs.hsgen.distribution.Normal;
import org.tpc.hs.hsgen.mr1.HSGen;
import org.tpc.hs.hsgen.util.HelperPRNG;
import org.tpc.hs.hsgen.util.HsgenHelper;

/**
 * NormalDistribution or GussianDistribution
 * http://en.wikipedia.org/wiki/Normal_distribution
 *
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 27.02.2015
 */
public class NormalDistributionBigInt extends Distribution {
 public static final String HSSORT_DISTRIBUTION_mu =
"Hssort.distribution.mu";
 public static final String HSSORT_DISTRIBUTION_sd =
"Hssort.distribution.sd";

 /*
  * mean value (the most common key / the position of tip of the bell
shaped
  * distribution curve in the keyspace)
  */
 private BigInteger mu_bi;
 public static final BigInteger DEFAULT_MU =
HsgenHelper.KEY_RANGE_ASCII_BI.divide(BigInteger
   .valueOf(2));

 /*-
  * standard deviation
  * about 68% of values drawn from a normal distribution are within
one standard deviation σ away from the mean
  * about 95% of the values lie within two standard deviations and
  * about 99.7% are within three standard deviations.
  *
http://en.wikipedia.org/wiki/Normal_distribution#Standard_deviation_a
nd_tolerance_intervals
  */
 /*
```

```
  * The default value of KEY_RANGE/8 was chosen because the distribu-
tion will
  * still spread over the whole key range. Every key is possible to
be
  * generated. However you may want to significantly decrease the SD
value to
  * achieve a narrower bell shaped curve. You should consider choos-
ing the SD
  * parameter based on the number of keys/records (aka. data size).
E.g. a sd
  * of 1/100 of the number of records will give you 5% unique keys
and 95%
  * dupplicates of keys.
  */
 private BigInteger sd_bi;
 public static final BigInteger DEFAULT_SD =
HsgenHelper.KEY_RANGE_ASCII_BI.divide(BigInteger
   .valueOf(8));

 private byte[] keyBytes = new byte[HsgenHelper.KEY_SIZE_IN_BYTES];

 private HelperPRNG helperRNG = new HelperPRNG();
 private Normal normalDist = new Normal();

 public NormalDistributionBigInt() {
  this(DEFAULT_MU, DEFAULT_SD);
 }

 public NormalDistributionBigInt(BigInteger mu2, BigInteger sd2) {
  set(mu2, sd2);
 }

 private void set(BigInteger mu2, BigInteger sd2) {
  validate(mu2, sd2);
  this.mu_bi = mu2;
  this.sd_bi = sd2;
 }

 private void validate(BigInteger mu2, BigInteger sd2) {
  if (mu2.add(sd2).compareTo(HsgenHelper.KEY_RANGE_ASCII_BI) > 0) {
   throw new IllegalArgumentException("mu(" + mu2 + ") + sd(" + sd2 +
") = "
     + mu2.add(sd2) + " > KEY_SIZE = " +
HsgenHelper.KEY_RANGE_ASCII_BI);
  }
 }

 @Override
 public void initialize(JobConf c) {
  set(getBigIntegerFromConf(HSSORT_DISTRIBUTION_mu, DEFAULT_MU, c),
    getBigIntegerFromConf(HSSORT_DISTRIBUTION_sd, DEFAULT_SD, c));
 }

 @Override
 public void generateKey(Text key, HSGen.RandomGenerator rand) {
  /*
   * key consists of 10 bytes from ' ' to ' '+95. 95^12 == 2^78 val-
ues =>
   * requiring ~78 bits
   */

  /*
   * Contract of HSGen.RandomGenerator is: call next() 3 times per
key.
   * Unfortunately most distribution algorithms require n*2 random
numbers
   * with an unknown number of n (in case the conditional while loops
of
   * e.g. a NormalDistribution doesn't get results meeting the condi-
tions)
   * Because of this, the 3 random numbers are used as a seed for a
custom
   * helper PRNG.
   */
  helperRNG.seed(rand.next(), rand.next(), rand.next());

  BigInteger normalDistValue =
normalDist.normalDistributedBigInteger(mu_bi, sd_bi,
    HsgenHelper.KEY_RANGE_ASCII_BI, helperRNG);

  // convert normal distributed value to radix 95 using BigInteger
  // arithmetic, because mr1 expects
  // ASCII keys in range [' ', ' '+95]
  HsgenHelper.toRadix_BE(normalDistValue,
HsgenHelper.KEY_TO_STRING_RADIX_BI, keyBytes,
    HsgenHelper.KEY_SIZE_IN_BYTES, ' ');
  key.set(keyBytes, 0, HsgenHelper.KEY_SIZE_IN_BYTES);

 }

 @Override
 public String toString() {
  return this.getClass().getSimpleName() + " [mu=" + mu_bi + ", sd="
+ sd_bi + "]";
 }
}
```

```
package org.tpc.hs.hsgen.mr1;
```

```
// [..] imports
import org.tpc.hs.hsgen.mr1.dist.Distribution;
import org.tpc.hs.hsgen.mr1.dist.DistributionFactory;

// [..]
public class HSGen extends Configured implements Tool {

// [..]

 /**
  * The Mapper class that given a row number, will generate the ap-
propriate
  * output line.
  */
 public static class SortGenMapper extends MapReduceBase implements
   Mapper<LongWritable, NullWritable, Text, Text> {

  private Distribution keyDistribution;
  private Text key = new Text();
  private Text value = new Text();
  private RandomGenerator rand;

  private byte[] spaces = "          ".getBytes();
  private byte[][] filler = new byte[26][];
  {
   for (int i = 0; i < 26; ++i) {
    filler[i] = new byte[10];
    for (int j = 0; j < 10; ++j) {
     filler[i][j] = (byte) ('A' + i);
    }
   }
  }

  /**
   * <DISTRIBUTION> added to load and instantiate chosen distribution
on
   * mappers
   */
  @Override
  public void configure(JobConf job) {
   super.configure(job);
   try {
    keyDistribution = DistributionFactory.getDistribution(job);
   } catch (Exception e) {
    // should not happen! pre-checked in HSGen.run() method
    e.printStackTrace();
   }
  }

  /**
   * Add a random key to the text
   *
   * @param key
   * @param rand
   *
   * @param rowId
   */
  private void addKey(Text key, RandomGenerator rand) {
   // <DISTRIBUTION> original code for uniform key generation relo-
cated
   // to org.tpc.hs.hsgen.mr1.dist.UniformDistribution
   keyDistribution.generateKey(key, rand);
  }

  /**
   * Add the rowid to the row.
   *
   * @param rowId
   * @param value
   * @param value
   */
  private void addRowId(long rowId, Text value) {
   byte[] rowid = Integer.toString((int) rowId).getBytes();
   int padSpace = 10 - rowid.length;
   if (padSpace > 0) {
    value.append(spaces, 0, 10 - rowid.length);
   }
   value.append(rowid, 0, Math.min(rowid.length, 10));
  }

  /**
   * Add the required filler bytes. Each row consists of 7 blocks of
10
   * characters and 1 block of 8 characters.
   *
   * @param rowId
   *           the current row number
   */
  private void addFiller(long rowId, Text value) {
   int base = (int) ((rowId * 8) % 26);
   for (int i = 0; i < 7; ++i) {
    value.append(filler[(base + i) % 26], 0, 10);
   }
   value.append(filler[(base + 7) % 26], 0, 8);
  }

  public void map(LongWritable row, NullWritable ignored,
OutputCollector<Text, Text> output,
    Reporter reporter) throws IOException {
   long rowId = row.get();
   if (rand == null) {
    // we use 3 random numbers per a row
    rand = new RandomGenerator(rowId * 3);
   }
   addKey(key, rand);
   value.clear();
   addRowId(rowId, value);
   addFiller(rowId, value);
   output.collect(key, value);
  }

 }

 private static void usage() throws IOException {
  System.err.println("HSGen <num rows> <output dir>");
 }

 /**
  * @param args
  *            the cli arguments
  * @throws ParseException
  */
 public int run(String[] args) throws Exception {
  if (args.length < 2) {
   usage();
   return 1;
  }
  JobConf job = (JobConf) getConf();

  /*
   * <DISTRIBUTION> Test if chosen distribution exists and parameters
are
   * correct. Eagerly testing this here prevents later exceptions in
   * mappers
   */
  DistributionFactory.getDistribution(job);

  setNumberOfRows(job, Long.parseLong(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  // if job name not overridden by cmd line
  if (job.getJobName() == null || job.getJobName().isEmpty())
   job.setJobName("HSGen");
  job.setJarByClass(HSGen.class);
  job.setMapperClass(SortGenMapper.class);
  job.setNumReduceTasks(0);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(Text.class);
  job.setInputFormat(RangeInputFormat.class);
  job.setOutputFormat(HSOutputFormat.class);
  JobClient.runJob(job);
  return 0;
 }
 public static void main(String[] args) throws Exception {
  int res = ToolRunner.run(new JobConf(), new HSGen(), args);
  System.exit(res);
 }
}
```

```
package org.tpc.hs.hsgen.mr2.dist;

import java.util.Arrays;
import java.util.List;

import org.apache.hadoop.conf.Configuration;

/**
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 27.02.2015
 */
public class DistributionFactory {

 public static class DistributionException extends Exception {
  public DistributionException(String msg, Throwable cause) {
   super(msg, cause);
  }
 }

 // CMD_LINE/JobConf key
 public static final String HSSORT_DISTRIBUTION =
"Hssort.distribution";

 // ############################################################
 // Add statically know distributions here:
 // you can allways load any distribution class via reflection
 // ###########################################################
 public static final List<Class<? extends Distribution>> dists = Ar-
rays.asList(
   UniformDistribution.class, NormalDistributionBigInt.class);

 public static Distribution getDistribution(Configuration configura-
tion)
   throws DistributionException {

  String distributionClassName = Distribu-
tion.getStringFromConf(HSSORT_DISTRIBUTION, dists
    .get(0).getName(), configuration);

  System.out.println("loading distribution: " + distributionClassName
+ " ...");
```

```
  Distribution dist;
  try {
   dist = (Distribution)
Class.forName(distributionClassName).newInstance();
  } catch (Exception e) {
   throw new DistributionException("Could not load distribution
class. Cause:  "
     + e.getClass().getName() + ": " + e.getMessage() + "\nAvailable
distributions:"
     + dists, e);
  }

  dist.initialize(configuration);
  System.out.println("loaded distribution " + dist.toString());
  return dist;

 }
}
```

```
package org.tpc.hs.hsgen.mr2.dist;

import java.math.BigInteger;

import org.apache.hadoop.conf.Configuration;
import org.tpc.hs.hsgen.mr2.Unsigned16;

/**
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 02.03.2015
 */
public abstract class Distribution {

 public static BigInteger getBigIntegerFromConf(String name,
BigInteger defaultValue,
   Configuration job) {
  String option = job.get(name, defaultValue.toString());
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return new BigInteger(option);
 }

 public static double getDoubleFromConf(String name, double
defaultValue, Configuration job) {
  double option = job.getDouble(name, defaultValue);
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return option;
 }

 public static boolean getBooleanFromConf(String name, boolean
defaultValue, Configuration job) {
  boolean option = job.getBoolean(name, defaultValue);
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return option;
 }

 public static String getStringFromConf(String name, String
defaultValue, Configuration job) {
  String option = job.get(name, defaultValue.toString());
  System.out.println("getConf(\"" + name + "\", " + defaultValue +
")=" + option);
  return option;
 }

 public abstract void initialize(Configuration configuration);

 public abstract void generateKey(byte[] recBuffer, Unsigned16 rand);

 public abstract BigInteger generateKey(Unsigned16 rand);

}
```

```
package org.tpc.hs.hsgen.mr2.dist;

import java.math.BigInteger;

import org.apache.hadoop.conf.Configuration;
import org.tpc.hs.hsgen.mr2.Unsigned16;
import org.tpc.hs.hsgen.util.HsgenHelper;

/**
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 02.03.2015
 */
public class UniformDistribution extends Distribution {

 @Override
 public void generateKey(byte[] recBuffer, Unsigned16 rand) {
  for (int i = 0; i < HsgenHelper.KEY_SIZE_IN_BYTES; ++i) {
   recBuffer[i] = rand.getByte(i);
  }
 }

 @Override
 public void initialize(Configuration configuration) {
  // nothing to do
 }

 @Override
 public String toString() {
  return this.getClass().getSimpleName() + " []";
 }

 @Override
 public BigInteger generateKey(Unsigned16 rand) {
  return rand.toBigInteger();
 }
}
```

```
package org.tpc.hs.hsgen.mr2.dist;

import java.math.BigInteger;

import org.apache.hadoop.conf.Configuration;
import org.tpc.hs.hsgen.distribution.Normal;
import org.tpc.hs.hsgen.mr2.Unsigned16;
import org.tpc.hs.hsgen.util.HelperPRNG;
import org.tpc.hs.hsgen.util.HsgenHelper;

/**
 * NormalDistribution or GussianDistribution
 * http://en.wikipedia.org/wiki/Normal_distribution
 *
 * @author Michael Frank (Michael.Frank@bankmark.de)
 * @version 1.0 02.03.2015
 */
public class NormalDistributionBigInt extends Distribution {

 public static final String HSSORT_DISTRIBUTION_mu =
"Hssort.distribution.mu";
 public static final String HSSORT_DISTRIBUTION_sd =
"Hssort.distribution.sd";

 /*
  * mean value (the most common key / the position of tip of the bell
shaped
  * distribution curve in the keyspace)
  */
 private BigInteger mu_bi;
 public static final BigInteger DEFAULT_MU =
HsgenHelper.KEY_RANGE_BINARY_BI.divide(BigInteger
   .valueOf(2));

 /*-
  * standard deviation
  * about 68% of values drawn from a normal distribution are within
one standard deviation σ away from the mean
  * about 95% of the values lie within two standard deviations and
  * about 99.7% are within three standard deviations.
  *
http://en.wikipedia.org/wiki/Normal_distribution#Standard_deviation_a
nd_tolerance_intervals
  */
 /*
  * The default value of KEY_RANGE/8 was chosen because the distribu-
tion will
  * still spread over the whole key range. Every key is possible to
be
  * generated. However you may want to significantly decrease the SD
value to
  * achieve a narrower bell shaped curve. You should consider choos-
ing the SD
  * parameter based on the number of keys/records (aka. data size).
E.g. a sd
  * of 1/100 of the number of records will give you 5% unique keys
and 95%
  * dupplicates of keys.
  */
 private BigInteger sd_bi;
 public static final BigInteger DEFAULT_SD =
HsgenHelper.KEY_RANGE_BINARY_BI.divide(BigInteger
   .valueOf(8));

 private HelperPRNG helperRNG = new HelperPRNG();
 private Normal normalDist = new Normal();

 public NormalDistributionBigInt() {
  this(DEFAULT_MU, DEFAULT_SD);
 }

 public NormalDistributionBigInt(BigInteger mu2, BigInteger sd2) {
  set(mu2, sd2);
 }

 private void set(BigInteger mu2, BigInteger sd2) {
  validate(mu2, sd2);
  this.mu_bi = mu2;
  this.sd_bi = sd2;
 }

 private void validate(BigInteger mu2, BigInteger sd2) {
  if (mu2.add(sd2).compareTo(HsgenHelper.KEY_RANGE_BINARY_BI) > 0) {
   throw new IllegalArgumentException("mu(" + mu2 + ") + sd(" + sd2 +
") = "
     + mu2.add(sd2) + " > KEY_SIZE = " +
HsgenHelper.KEY_RANGE_BINARY_BI);
  }
 }

 @Override
 public void initialize(Configuration c) {
  set(getBigIntegerFromConf(HSSORT_DISTRIBUTION_mu, DEFAULT_MU, c),
    getBigIntegerFromConf(HSSORT_DISTRIBUTION_sd, DEFAULT_SD, c));
 }

 @Override
 public BigInteger generateKey(Unsigned16 rand) {
  /*
   * Contract of Unsigned16 rand is: we have a single 128bit random
number
   * to randomly calculate a distributed key. Unfortunately most
   * distribution algorithms require n*2 random numbers with an un-
known
   * number of n (in case the conditional while loops of e.g. a
   * NormalDistribution doesn't get results meeting the conditions)
   * Because of this, the random number 'rand' is used as a seed for
a
   * custom helper PRNG.
   */
  helperRNG.seed(rand.getHigh8(), rand.getLow8());
  BigInteger normalDistValue =
normalDist.normalDistributedBigInteger(mu_bi, sd_bi,
    HsgenHelper.KEY_RANGE_BINARY_BI, helperRNG);
  return normalDistValue;
 }

 @Override
 public void generateKey(byte[] recBuf, Unsigned16 rand) {
  BigInteger normalDist = generateKey(rand);
  copyToResult(recBuf, normalDist, HsgenHelper.KEY_SIZE_IN_BYTES);
 }

 private static void copyToResult(byte[] recBuf, BigInteger res, int
KEY_SIZE_BYTES) {
  byte[] keyBytes = res.toByteArray();
  for (int i = 0; i < 10 - keyBytes.length; i++) {
   recBuf[i] = 0;
  }
  int start = Math.max(0, keyBytes.length - KEY_SIZE_BYTES);
  int startRec = KEY_SIZE_BYTES - Math.min(KEY_SIZE_BYTES,
keyBytes.length);
  int copyLen = Math.min(KEY_SIZE_BYTES, keyBytes.length);
  System.arraycopy(keyBytes, start, recBuf, startRec, copyLen);
 }

 @Override
 public String toString() {
  return this.getClass().getSimpleName() + " [mu=" + mu_bi + ", sd="
+ sd_bi + "]";
 }
}
```

```
package org.tpc.hs.hsgen.mr2;

//[..]import
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.tpc.hs.hsgen.mr2.dist.Distribution;
import org.tpc.hs.hsgen.mr2.dist.DistributionFactory;
import
org.tpc.hs.hsgen.mr2.dist.DistributionFactory.DistributionException;
import org.tpc.hs.hsgen.util.HsgenHelper;

//[..]
public class HSGen extends Configured implements Tool {

//[..]

/**
  * The Mapper class that given a row number, will generate the ap-
propriate
  * output line.
  */
 public static class SortGenMapper extends Mapper<LongWritable,
NullWritable, Text, Text> {

  private Text key = new Text();
  private Text value = new Text();
  private Unsigned16 rand = null;
  private Unsigned16 rowId = null;
  private Unsigned16 checksum = new Unsigned16();
  private Checksum crc32 = new PureJavaCrc32();
  private Unsigned16 total = new Unsigned16();
  private static final Unsigned16 ONE = new Unsigned16(1);
  private byte[] buffer = new byte[HSInputFormat.KEY_LENGTH +
HSInputFormat.VALUE_LENGTH];
  private Counter checksumCounter;

  private Distribution keyDistribution;

  /**
```

```
   * <DISTRIBUTION> added to load and instantiate chosen distribution
on
   * mappers
   */
  @Override
  protected void setup(Mapper<LongWritable, NullWritable, Text,
Text>.Context context)
    throws IOException, InterruptedException {
   super.setup(context);
   try {
    keyDistribution =
DistributionFactory.getDistribution(context.getConfiguration());
   } catch (Exception e) {
    // should not happen! pre-checked in HSGen.run()
    e.printStackTrace();
   }
  }

  public void map(LongWritable row, NullWritable ignored, Context
context)
    throws IOException, InterruptedException {
   if (rand == null) {
    rowId = new Unsigned16(row.get());
    rand = Random16.skipAhead(rowId);
    checksumCounter = context.getCounter(Counters.CHECKSUM);
   }

   Random16.nextRand(rand);

   // <DISTRIBUTION> pass key distribution instance to record
   // generation
   // logic
   GenSort.generateRecord(buffer, rand, rowId, keyDistribution);
   key.set(buffer, 0, HSInputFormat.KEY_LENGTH);
   value.set(buffer, HSInputFormat.KEY_LENGTH,
HSInputFormat.VALUE_LENGTH);
   context.write(key, value);
   crc32.reset();
   crc32.update(buffer, 0, HSInputFormat.KEY_LENGTH +
HSInputFormat.VALUE_LENGTH);
   checksum.set(crc32.getValue());
   total.add(checksum);
   rowId.add(ONE);
  }

  @Override
  public void cleanup(Context context) {
   if (checksumCounter != null) {
    checksumCounter.increment(total.getLow8());
   }
  }
```

```
 }

//[..]
}
```

References

1. Nambiar, R., Poess, M., Dey, A., Cao, P., Magdon-Ismail, T., Qi Ren, D., Bond, A.: Introducing TPCx-HS: The First Industry Standard for Benchmarking Big Data Systems. In: Nambiar, R., Poess, M. (eds.) TPCTC 2014. LNCS, vol. 8904, pp. 1–12. Springer, Heidelberg (2015)
2. TPCx-HS Specification. www.tpc.org
3. O'Malley, O.: TeraByte sort on apache hadoop (2008)
4. Nambiar, R., Poess, M.: Keeping the TPC relevant! PVLDB **6**(11), 1186–1187 (2013)
5. Nambiar, Raghunath, Poess, Meikel (eds.): TPCTC 2013. LNCS, vol. 8391. Springer, Heidelberg (2014)
6. Nambiar, R.: A standard for benchmarking big data systems. In: BigData Conference 2014, pp. 18–20 (2014)
7. Nambiar, R.: Benchmarking big data systems: introducing TPC express benchmark HS. In: Rabl, T., Sachs, K., Poess, M., Baru, C., Jacobson, H.-A. (eds.) WBDB 2014. LNCS, vol. 8991, pp. 24–28. Springer, Heidelberg (2015)