Composite Key Generation on a Shared-Nothing Architecture

Marie Hoffmann[1], Alexander Alexandrov[1], Periklis Andritsos[2], Juan Soto[1], and Volker Markl[1]

[1] Technische Universität Berlin, DIMA, Einsteinufer 17, 10587 Berlin, Germany
[2] Université de Lausanne, Institut des Systèmes d'Information, Bâtiment Internef, 1015 Lausanne, Switzerland

Abstract. Generating synthetic data sets is integral to benchmarking, debugging, and simulating future scenarios. As data sets become larger, real data characteristics thereby become necessary for the success of new algorithms. Recently introduced software systems allow for synthetic data generation that is truly parallel. The systems use fast pseudorandom number generators and can handle complex schemas and uniqueness constraints on single attributes. Uniqueness is essential for forming keys, which identify single entries in a database instance. The uniqueness property is usually guaranteed by sampling from a uniform distribution and adjusting the sample size to the output size of the table such that there are no collisions. However, when it comes to *real* composite keys, where only the combination of the key attribute has the uniqueness property, a different strategy is required. In this paper, we present a novel approach on how to generate composite keys within a parallel data generation framework. We compute a joint probability distribution that incorporates the distributions of the key attributes and use the unique sequence positions of entries to address distinct values in the key domain.

1 Introduction

When big data systems have to be debugged or their performance needs to be analyzed, large test data sets are required. Due to privacy restrictions or locality of data sets, it is not always feasible to ship and share the original data. Additionally, one might be interested in the analysis of data patterns that are not present in current real world data. These are use cases where synthetic data generation plays a vital role.

Synthetic generation of giga- and terabyte data sets becomes scalable if data is generated truly in parallel and not only distributed, i.e., a data generating process does not need to communicate or synchronize with other processes. This is facilitated by a recent trend towards massively parallel shared-nothing architectures where processes have no common resources and communication is typically expensive. It is important that frameworks designed for such architectures take the distribution of resources into account.

Traditional data generators, like `dbgen` from TPC-H,[3] are hand-tuned scripts dedicated to produce data for a specific schema. In the schema of TPC-H all unique attributes are primary keys of type integer ranging from 1 to n, where n is the final table size. By simply partitioning the sequence of integers and drawing all other attributes independently and randomly from predefined distributions, tuples can be generated in parallel. Process i will thereby generate the primary keys $[(i-1)\cdot\frac{n}{N}, ..., i\cdot\frac{n}{N}]$ where N is the number of child processes. It is the low complexity of the schema of TPC-H that enables parallel execution. The fixed schemata are of few inter- and intra-column dependencies and are common to most standard benchmarks. However, this does not suffice to perform tasks like validation of techniques that are sensitive to specific data patterns.

In contrast, flexible toolkits, like Myriad [1] or PDGF [11], enable the implementation of use-case tailored data generators. Generators thereby offer the opportunity to not only produce data sets for benchmarks but also for debugging or testing purposes. Most importantly, Myriad and PDGF follow a parallel execution model for shared-nothing architectures. That is, independent from value domains and column constraints they split the data generation process for tables row- or column-wise. By assigning distinct substreams of the *pseudorandom number generator* (PRNG), which serves for sampling, to each node, inter-process communication is avoided.

Through a hierarchical seeding strategy any value of the final data set can be computed locally, which is integral for resolving data dependencies. To have constant execution times, the involved class of PRNGs must be non-recursive. Examples of commonly used PRNGs with constant lookup times are *explicit inverse congruential generators* (EICGs) [5], *compound* EICGs [4], or *hash function* based PRNGs [9], [10].

Pseudo-random data generators (PRDGs) use the uniformly distributed output of PRNGs to produce user-defined domain values for arbitrary distribution functions through *inverse transform sampling* (ITS, see Section 2.2) or *dictionary lookups*. ITS is sufficient to generate values for a single column for which uniqueness holds. Consequently, these generators support all key constraints where at least one *simple key* is involved. Simple keys are single attributes, whose values uniquely identify a row. However, this approach fails if it comes to the generation of composite keys for which only the combination of all key attributes is unique (i.e., we cannot ensure uniqueness if no simple key is involved). Unfortunately, all parallel data generators we are aware of suffer from this constraint.

In this paper we present a novel approach on how to overcome this limitation. Our basic approach is to use the unique row identifiers to address distinct key tuples in the discretized space of all possible keys.

The rest of the paper is organized as follows. In Section 2 we give a more formal description of the problem of parallel composite key generation. In Section 3 we present our algorithm with an accompanying example followed by an evaluation part (Section 4) and a discussion (Section 7). Notations and additional examples are provided in the appendix.

[3] http://www.tpc.org/tpch/

2 Composite Keys

2.1 Definition

Our composite key generation approach is explained and evaluated based on the relational model [3], although our approach is not restricted to this particular data model. We use the term 'table' as a synonym for a relation $\mathcal{R}$ whose columns are a set of attributes $\mathcal{A}$ and whose rows are tuples. Single tuples are addressed by keys, a subset of attributes for which uniqueness must hold. This constraint is checked by the database management system when tuples are inserted or modified. Although a table may have many columns, we only consider the d columns that are relevant when forming a key, i.e., $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_d\}$.

In the context of databases, *composite keys* are identifiers of two or more attributes ($d > 1$) for which at least one attribute does not make up a *simple key*. In other words, for composite keys there exists at least one key attribute that is not unique. In contrast, *compound keys* are keys of two or more attributes where *each* attribute makes up a simple key in its own right. To make this distinction clearer, examples are given in the appendix (Section A).

2.2 Problem Statement

Our goal is to generate a table of n key tuples, each of them forming a 'real' composite key of d attributes. Moreover, the resulting key set should respect intra-table dependencies and satisfy attribute distributions in expectation.

To understand why guaranteeing uniqueness on a combination of non-unique attributes is technically more challenging than producing a single, unique column we will first describe how synthetic data is generated by parallel data generator tools (PDGs) like Myriad or PDGF. Typical demands on the output of PDGs are:

i) attribute values are distributed according to arbitrary, derived or user-defined distributions;
ii) attribute values may conditionally depend on other attributes;
iii) concrete attribute values are not correlated to process identifiers or their sequence position in the generation process, unless it is intended.

The above requirements can be satisfied if data generators use a class of PRNGs whose sequence can be accessed randomly and that show a high degree of randomness, i.e., they pass various tests for randomness. These properties ensure that we can partition the sequence of random numbers without the need to compute previous values, i.e., we have constant access time. We can therefore recompute values from other substreams in constant time to resolve attribute dependencies without querying remote processes – a key property for parallelization. Compound EICGs and some hash functions are PRNGs with these properties. Both are implemented in Myriad. Panneton et al. [10] give concrete examples of hash functions passing several tests for randomness.

PRNGs provide the input for pseudorandom data generators. The uniformly distributed output is normalized to $[0, 1]$ and mapped to the target domain via *inverse transform sampling*. Given a continuous or discrete distribution function $f : \mathcal{X} \mapsto [0, 1]$, with $\mathcal{X}$ being a continuous, discrete (numerical or categorical) domain, we compute its cumulative distribution function (CDF) $F_X : \mathcal{X} \mapsto [0, 1], F_X(x) = P(X \leq x)$ on a random variable $X \in \mathcal{X}$ and take the inverse (see Figure 1 as an illustrative example).

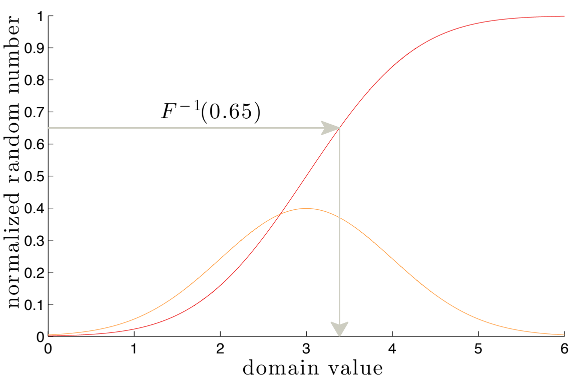


Fig. 1. Illustration of the inverse transform sampling technique. Given a set of uniformly distributed values from $[0, 1]$, we want to simulate normally distributed values $f \propto \mathcal{N}(x|\mu = 2, \sigma = 1)$ (orange) in $[0, 6]$. Putting a normalized value $y \in Y$ from a uniform distribution into the inverse of its CDF $F_X = \int_{-\infty}^{X} f_X(t)dt$ (red) produces a domain value that is distributed according to the target distribution function (gray path).

From the sampling perspective, composite keys fall into two groups. The first group exhibits at least one unique attribute (see example schema `Composite1` in the appendix), while the second group exhibits none (see schema `Composite2`), which means that uniqueness holds exclusively for the join of all key columns. Data according to the first group can be generated with the same approach that applies to simple keys (i.e., we produce a unique attribute column which in turn ensures uniqueness for the whole key tuple). The other key attributes are sampled according to their target distributions.

For the second group of composite keys this approach is not applicable. Each key attribute may contain repeated values according to its particular distribution function (e.g., the first attribute may follow a uniform distribution, the second attribute a normal distribution, and so forth). A naïve approach would be to sample attribute-wise conditioned on already generated domain values. However, this approach does not scale, since it requires parsing all of the data. On a shared-nothing architecture this would be prohibitively expensive.

We will now describe an approach for generating composite keys that is applicable within the parallel data generation frameworks for shared-nothing architectures described introductorily. From now on, we use the term *composite key* as a synonym for the aforementioned second group that contains no unique single attribute.

3 Algorithm

3.1 Preliminaries

We describe the composite key generation problem from the perspective of a data generating process running on a shared-nothing architecture. In total we have N processes that together produce n unique composite key tuples.

We will later see that we do not sample, but rather permute indices to keep the uniqueness property of tuple identifiers. Thus, a node has instances of pseudorandom permutation functions (PRPs) that shuffle an input integer in a unique manner. The PRPs are keyed functions whose output domain size is adjustable. This is an essential property as explained in the next section. We will denote a PRP by

$$\pi : \mathcal{K} \times \mathcal{E} \times \mathcal{D} \mapsto \mathcal{D}.$$

where $\mathcal{K}$ is the key space for seeding the PRP, $\mathcal{D}$ the target size for the output, and $\mathcal{E}$ the input. We address elements of a PRP in an array-like fashion, notated as $[\cdot]$. For the same key and domain size, $\pi_{k,d}[i] \neq \pi_{k,d}[j]$ holds for all i, j iff $i \neq j$.

As initial input the process receives a process identifier in $[0..N-1]$, and the total table size n. It also receives an instance of a PRP function together with a seed $k \in \mathcal{K}$ which enables each process to use the same permutation. As part of the configuration, a process has information about the attribute domains $\mathcal{A}_i$ and their target distributions, given attribute-wise as discrete histograms $\mathcal{B}_i$. Attribute domains may be nominal or real-valued.

3.2 Idea

The core idea of our algorithm is to produce a bijective mapping between a set of unique tuple identifiers and the tuple space for each data generating process. The tuple identifiers are given by the non-intersecting ranges of indices that are assigned to the data generating processes. For example, process N_1 produces table entries with identifiers $[1, 2, ..., r_1]$, process N_2 for identifiers $[r_1 + 1, r_1 + 2, ..., r_2]$, and so forth.

To be able to produce a unique mapping we assume that the domains of the attributes of interest are discretizable and their particular distributions are given in terms of histograms (univariate or conditioned). From the set of histogram distributions, we form a multidimensional joint histogram. However, instead of sampling with the aid of PRNGs, identifier ranges are mapped to bins such that the relative range corresponds to a particular bin height of joint histogram.

In doing so, we respect the joint probability distribution of the key attributes. Given a unique tuple identifier, we first assign a bin and then pick a tuple within the bin. Algorithm 1 summarizes these steps:

Algorithm 1: Composite Key Generation.

```
    **Input**: nodeID, N, n, $\mathcal{A}$, $\mathcal{B}$
    **Output**: Data
1   // Compute joint histogram
2   $C$ := jointHistogram($\mathcal{B}$)
3   // Generate n/N composite keys
4   for $id \leftarrow (nodeID - 1)\cdot n/N$ to $nodeID\cdot n/N$ do
5       // Find bin for current tuple identifier
6       binID := findBin($id$, $C$)
7       // Convert scalar to tuple
8       Data[id] = id2Tuple($id$, $\mathcal{A}$, $\mathcal{C}_{\mathrm{binID}}$)
9   end
```

Usually, tuple identifiers are distributed in a sequential manner among data generating nodes, like described previously with processes N_1 and N_2. Since these identifiers will later correspond to tuple indices of the ordered set of all possible tuples, which is much larger, we break the correlation by introducing shuffling. Shuffling of identifiers will be applied preliminary to the bin assignment step (line 6 of Algorithm 1) and to the assignment of a multidimensional tuple within a bin (line 8 of Algorithm 1). After introducing the accompanying example, we will explain in more detail the computation of the joint histogram, the bin assignment, and the tuple assignment step.

3.3 Accompanying Example

For the purposes of illustration, we will consider an example from biochemistry. Assume we would like to generate a table of composite keys of two attributes: proteins and amino acids. The building blocks of proteins are amino acids and their derivates. We restrict our example to three common proteins, i.e., *collagen* (c), *actin* (a), and *hemoglobin* (h) and six amino acids, i.e., *Glycine* (G), *Proline* (P), *Alanine* (A), *Glutamine* (Q), *Arginine* (R), and *Aspartic acid* (D). For all compounds we use the one-letter notation.

$$\begin{aligned}\mathcal{A} &= \{A_1, A_2\}\\ A_1 &= \{\mathrm{c}, \mathrm{a}, \mathrm{h}\}\\ A_2 &= \{\mathrm{G}, \mathrm{P}, \mathrm{A}, \mathrm{Q}, \mathrm{R}, \mathrm{D}\}\end{aligned}$$

The normalized frequencies of the three proteins in mammal tissues and their composition from amino acids are given in the tables below:

Table 1. Proteins and their normalized frequencies.

protein	frequency
c	0.7
a	0.2
h	0.1

Table 2. Six amino acids and their normalized frequencies in collagen, actin, and hemoglobin.

protein	amino acid frequency					
	G	P	A	Q	R	D
c	0.40	0.23	0.15	0.09	0.07	0.07
a	0.17	0.10	0.18	0.24	0.11	0.20
h	0.28	0.09	0.23	0.17	0.06	0.17

3.4 Joint Histogram

Let us fix our notations. For a set of attribute domains $\mathcal{A}$, we have a corresponding set of histograms $\mathcal{B}$ reflecting the discretized distributions of the attributes. For a single histogram B_i, we denote with $\gamma_{\mathcal{B}_i}$ the number of bins, $b_{i,j}^{low}$ the lower bound, and with $b_{i,j}^{up}$ the upper bound (both inclusive of the bin), and $f_{i,j}$ the relative frequency of values from $\mathcal{A}_i$ that fall into the boundaries of the j-th bin:

$$\begin{aligned}\mathcal{B} &= \{B_1, B_2, ..., B_d\}\\ B_i &= [(b_{i,1}^{low}, b_{i,1}^{up}, f_{i,1}), ..., (b_{i,\gamma_{\mathcal{B}_i}}^{low}, b_{i,\gamma_{\mathcal{B}_i}}^{up}, f_{i,\gamma_{\mathcal{B}_i}})], \quad i \in [1..d]\end{aligned}$$

For our running example, we have

$$\begin{aligned}\mathcal{B} &= \{B_1, B_2\}\\ B_1 &= [(\mathrm{c}, \mathrm{c}, 0.7), (\mathrm{a}, \mathrm{a}, 0.2), (\mathrm{h}, \mathrm{h}, 0.1)]\\ B_2 &= [(\mathrm{G}, \mathrm{A}, 0.67), (\mathrm{Q}, \mathrm{D}, 0.33)]\end{aligned}$$

Each interval $[b_{i,j}^{low}..b_{i,j}^{up}]$ represents a disjoint subset for attribute $\mathcal{A}_i$. The joint histogram is the set of all combinations $B_i \otimes B_j$ of intervals taken from B_1 to B_d. If we assume independence of the attributes, we can compute the joint frequencies ϕ for the combined histogram as the product of all one-dimensional bin probabilities. In our example the amino acid frequencies depend on the protein. Table 2 displays the conditional probabilities of amino acids given a protein. Hence, the joint probability for our two-dimensional case is

$$Pr[c_1 \in B_{1,i}, c_2 \in B_{2,j}] = \sum_{c_1 \in B_{1,i}} Pr[c_1] \cdot \big(\sum_{c_2 \in B_{2,j}} Pr[c_2|c_1] \big). \tag{1}$$

Using Eq. 1, we receive for our running example the following joint probabilities:

$$\begin{aligned}Pr[c \in C_1] &= Pr[c_1 \in B_{1,1}] \cdot Pr[c_2 \in B_{2,1}|c_1 \in B_{1,1}] = 0.546\\ Pr[c \in C_2] &= Pr[c_1 \in B_{1,2}] \cdot Pr[c_2 \in B_{2,2}|c_1 \in B_{1,2}] = 0.09\\ Pr[c \in C_3] &= Pr[c_1 \in B_{1,3}] \cdot Pr[c_2 \in B_{2,1}|c_1 \in B_{1,3}] = 0.06\\ Pr[c \in C_4] &= Pr[c_1 \in B_{1,1}] \cdot Pr[c_2 \in B_{2,2}|c_1 \in B_{1,1}] = 0.161\\ Pr[c \in C_5] &= Pr[c_1 \in B_{1,2}] \cdot Pr[c_2 \in B_{2,1}|c_1 \in B_{1,2}] = 0.11\\ Pr[c \in C_6] &= Pr[c_1 \in B_{1,3}] \cdot Pr[c_2 \in B_{2,2}|c_1 \in B_{1,3}] = 0.04\end{aligned}$$

For d attributes the tensor Φ of joint probabilities has d dimensions. When attributes are independent, it is the result of a series of products[4] of bin frequencies:

$$\Phi = \oplus_{i=1}^{d} f_i$$

Each entry $\Phi_{i_1,i_2,..,i_d} = f_{1,i_1} \cdot f_{2,i_2} \cdot .. \cdot f_{d,i_d}$ represents the relative frequency of a multidimensional bin with a set of lower and upper bin edges. For variables conditioned on others, we may replace some f_i by the conditional probability tables. In order to avoid lists of indices, we use a one-dimensional representation. We apply a reshaping $\rho : \mathbb{R}^{m_1 \times .. \times m_d} \mapsto \mathbb{R}^{\prod m_i}$ on the tensor Φ. The reshaping logically arranges the tensor as a vector:

$$\phi := \rho(\Phi).$$

In our example, the reshaping of joint probabilities would be a row-wise concatenation. Let C denote the set of d-dimensional bins with frequencies ϕ:

$$\begin{aligned} C &= [C_1, C_2, ..., C_{\gamma_C}], \quad \gamma_C = |B_1| \cdot ... \cdot |B_d| \\ C_i &= (\boldsymbol{c_i}^{low}, \boldsymbol{c_i}^{up}, \phi_i) \in B_1 \times B_2 \times ... \times B_d \\ \boldsymbol{c_i}^{low} &= [c_{i,1}^{low}, c_{i,2}^{low}, ..., c_{i,d}^{low}] \\ \boldsymbol{c_i}^{up} &= [c_{i,1}^{up}, c_{i,2}^{up}, ..., c_{i,d}^{up}] \end{aligned}$$

If we use the above notation for our example, the joint histogram is:

$$\begin{aligned} C_1 &= ([\mathrm{c}, \mathrm{G}], [\mathrm{c}, \mathrm{A}], 0.546) & C_4 &= ([\mathrm{a}, \mathrm{Q}], [\mathrm{a}, \mathrm{D}], 0.11) \\ C_2 &= ([\mathrm{c}, \mathrm{Q}], [\mathrm{c}, \mathrm{D}], 0.161) & C_5 &= ([\mathrm{h}, \mathrm{G}], [\mathrm{h}, \mathrm{A}], 0.06) \\ C_3 &= ([\mathrm{a}, \mathrm{G}], [\mathrm{a}, \mathrm{A}], 0.09) & C_6 &= ([\mathrm{h}, \mathrm{Q}], [\mathrm{h}, \mathrm{D}], 0.04) \end{aligned}$$

The ordering of the proteins and amino acids will be kept throughout the whole paper (e.g., C_3 represents actin combined with the three amino acids Glutamine, Arginine, and Aspartic acid).

3.5 Bin Assignment

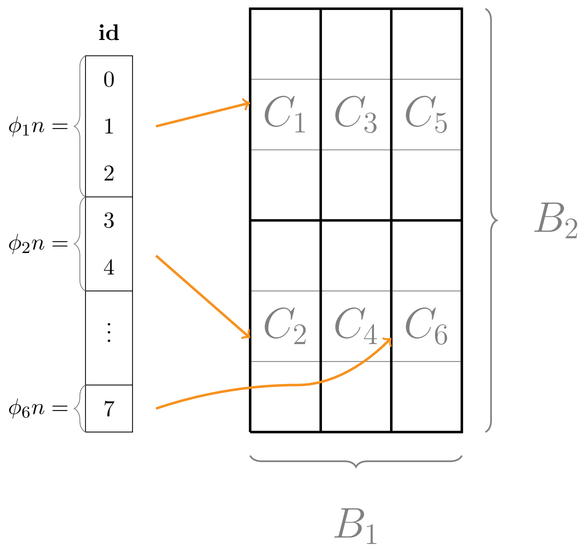
Following the construction of a multidimensional histogram, the next step is to guarantee that in expectation $\phi_i \cdot n$ of the n key tuples will lie in bin C_i. We construct a step function that maps tuple identifiers to bin indices with ranges adjusted according to ϕ:

$$\text{findBin} : [0..n-1] \mapsto [0..\gamma_C - 1]$$

[4] denoted by $\oplus$

$$\text{findBin}(id) = \begin{cases} 1, & \text{if } 0 \leq id < n\phi_1 \\ 2, & \text{if } n\phi_1 \leq id < n(\phi_1 + \phi_2) \\ \vdots & \vdots \\ k, & \text{if } n\sum_{i=1}^{k-1} \phi_i \leq id < n\sum_{i=1}^{k} \phi_i \\ \vdots & \vdots \\ \gamma_C, & \text{if } id \geq n\sum_{i=1}^{\gamma_C - 1} \phi_i \end{cases}$$

The following figure illustrates the bin assignment step for our joint histogram of the example:



Due to the sequential assignment of identifiers to the data generating nodes, we obtain a clustered bin assignment. Hence, each process will generate keys that lie in the same or in adjoined bins. We can decorrelate node and bin identifiers by first applying a keyed permutation function that shuffles the tuple identifier:

$$\text{binID} := \text{findBin}(\pi_n[id], C)$$

In our toy example, we would like to produce $n = 8$ composite keys by $N = 2$ processes. N_1 generates keys for initial identifiers in $[0..3]$ and N_2 for $[4..7]$. To shuffle the indices, we simply XOR the identifiers with random pad k, say 5.

$$\pi_{k=5,D=8}[id] = id \oplus k.$$

The step function for bin assignment and the assignment of identifiers to bin IDs are given below. Note that bin C_3 and C_6 are empty due to rounding and the small output size n.

$$\text{findBin}_C(id) = \begin{cases} 1, \text{ if } 0 \leq id < 3 \\ 2, \text{ if } 3 \leq id < 6 \\ 4, \text{ if } 6 \leq id < 7 \\ 5, \text{ else} \end{cases}$$

	id	$\pi_{5,8}[id]$	binID = $findBin(\pi_{5,8}[id])$
N_1	0	5	2
	1	4	2
	2	7	5
	3	6	4
N_2	4	1	1
	5	0	1
	6	3	2
	7	2	1

3.6 Tuple Assignment

Given the bin identifier i of the joint histogram from the previous step, we again make use of the uniqueness of id to compute a *relative* tuple index tupleID in C_i. This can then be used to map attribute-wise to the output domain. The relative position of id is obtained by subtracting the lowest tuple identifier that is assigned to C_i. This information is given by the step function of the bin assignment step, i.e.,

$$\text{tupleID} := id - \min(\text{findBin}_C^{-1}(id)).$$

Note that we use here a simplified description, since id may not directly be used, but its shuffled value ($\pi[id]$ instead of id). The relative tuple identifier is in $[0..\phi_i \cdot n)$. If we think of the output domain values as an ordered set, we address the first $\phi_i \cdot n$ tuples. Backmost tuples that correspond to identifiers in $[\phi_i \cdot n, ..., \gamma_{C_i})$ are missed. We can hit any tuple in C_i with equal probability if we shuffle the indices. Hence, we again shuffle identifiers before converting the scalars to values in the output domain. The parameter D for the output domain of the shuffle function is the bin cardinality of C_i, tID $= \pi_{D=C_i}[\text{tupleID}]$. To produce a key $= (a_1, a_2, ..., a_d) \in A_1 \times A_2 \times .. \times A_d$ from the shuffled scalar tuple identifier, we iteratively compute integer division of a rest and the total cardinality of the subsequent dimensions (see Algorithm 2). This last conversion step corresponds to the procedure `id2Tuple` listed in Algorithm 1 and is shown below:

Table 3 shows the final result for our accompanying example. To permute the relative tuple identifier, we use π_3 – a shift by one, i.e., $\pi_3[i] = (i+1) \bmod 3$.

Algorithm 2: Conversion of scalar to tuple of output domain.

```
   Input: tupleID, $\mathcal{A}$, $C_i$
   Output: $a$
 1 tID := $\pi_{D=C_i}$[tupleID]
 2 rem := tID
 3 for $k = 1..d-1$ do
 4     // Bin cardinality for subsequent dimensions
 5     $\gamma := \prod_{j=k+1}^{d} |\{a \in A_j | c_{ij}^{low} \leq a \leq c_{ij}^{up}\}|$
 6     // Compute absolute index
 7     pos := $\lfloor rem/\gamma \rfloor + \mathcal{A}_k$.indexOf($c_{ik}^{low}$)
 8     $a_k := \mathcal{A}_k$[pos]
 9     rem := rem mod$\gamma$
10 end
11 pos := rem + $\mathcal{A}_k$.indexOf($c_{dk}^{low}$)
12 $a_d := \mathcal{A}_d$[pos]
```

Table 3. Conversion of *id* and binID to tuples. The last two steps are performed by Algorithm 2

	id	binID	tupleID = id $-$ min(findBin^{-1}(binID))	tID = π_3[tupleID]	tuple
N_1	5	3	0	1	(a,P)
	4	2	1	2	(c,D)
	7	5	0	1	(h,P)
	6	4	0	1	(a,R)
N_2	1	1	1	2	(c,A)
	0	1	0	1	(c,P)
	3	2	0	1	(c,R)
	2	1	2	0	(c,G)

4 Evaluation

Implementation into Myriad. For shuffling tuple identifier we implemented the permutation scheme proposed in Section 7.1. The algorithm for synthetic composite key generation was tested on a numerical data set – the stellar data set from the Sloan Digital Sky Survey (SDSS). After determining the feature set for the composite key attributes for both data sets we proceed as follows:

i) Computation of the histograms $\mathcal{B}$ for each feature.
ii) Execution of the composite key generation algorithm with different scaling factors or partition schemes.
iii) Optional testing for duplicates.
iv) Computation of distribution error (see Eq. 2) by comparing initial histograms f and histograms $\tilde{f}$ computed on output files.

$$error(f,\tilde{f}) = \frac{1}{|\mathcal{A}|}\sum_{i=1}^{|\mathcal{A}|}\frac{1}{|\mathcal{B}_i|}\sum_{j=1}^{|\mathcal{B}_i|}\left(f_{ij}-\widetilde{f_{ij}}\right)^2. \tag{2}$$

All tests were performed with two Intel Xeon Processors E5620 (12M Cache, 2.40 GHz, 5.86 GT/s Intel QPI) and 50 GB main memory.

4.1 Sloan Digital Sky Survey Data Set

The Sloan Digital Sky Survey (SDSS) data set contains the positions of celestial bodies, i.e. spherical coordinates right ascension (ra) and declination (dec) and the amount of light emitted at different wavelengths. Since no two objects have the same position, ra and dec form a composite key. We queried the data of 1000 objects in the field of the galaxy NGC 2967 (see SQL query below) and computed histograms of four bins per dimension. The values were discretized by cutting their floating point values after five decimal places.

```
SELECT top 1000 p.objid, p.ra, p.dec
FROM galaxy p,
     dbo.fgetNearByObjEq(145.514,0.336,4) n
WHERE p.objid=n.objid
```

For the first test we launched the binary with varying number of processes N, i.e. 1 to 16 processes generating a table of 1 GB size. The execution time includes the writing of data to hard disk, see Table 4:

Table 4. Error and execution times [s] with varying partition schemes and constant table size of 1 GB.

N	1	2	4	8	16
Error	0.593e-4	0.593e-4	0.593e-4	0.593e-4	0.593e-4
Time	155	80	45	35	25

As a second test we varied the data set size from 100 MB to 100 GB, but kept the partition scheme fixed. Since the first test already covered testing for duplicates, we omitted the writing of data to hard disk and computed the histograms on-the-fly:

Table 5. Error and execution times [s] with varying table sizes and 8 processes.

Size	100 MB	1 GB	10 GB	100 GB
Error	0.8264e-4	0.8265e-4	0.8238e-4	0.8244e-4
Time	0.34	1.8	6.1	9.4

5 Results

The resulting data sets were sorted and checked for duplicates by the Linux command line tools `sort` and `comm`. For all tests and partition schemes the pairwise comparison of the resulting data files showed that no duplicates had been generated.

The experimental results in Table 4 show that the partitioning has no influence on the data quality – the averaged errors between the initial histograms and the ones computed on the resulting data sets are constant. The experiment also shows that our composite key generation algorithm is parallelizable, since the execution times decrease when more processors are initiated. The second test (see Table 5) points out that with larger scalings the initial distributions are still respected and data is not skewed. Note, that errors of Table 4 and Table 5 are not comparable since different bucket widths were used.

6 Related Work

There is much work done in synthetic data generation. Gray et al. [6] were one of the first to propose strategies for scalable data generation. For example, they propose a forking scheme where each process generates a partition of each table. They also showed how to use multiplicative groups to produce sequences of numbers that are dense and unique.

Bruno and Chaudhuri developed a flexible data generation tool [2] including a Data Description Language (DGL). The basic construct for generating values is the iterator. Iterators can be combined, nested or concatenated to produce complex types. DGL also allows for querying existing data during the generation of new data. Iterators produce their output sequentially and may be consumed by other iterators. However, their buffering technique via shared memory makes their approach unscalable.

In his PhD thesis, Hoag [7], [8] presented a parallel synthetic data generator (PSDG) along with an XML-based data description language. The synthetic data specification accepts five types of generation constraints: uniform, histogram, value sequences, and formular constraints involving operators, constants, built-in functions, or field values.

Rabl et al. [11] presented a parallel data generation framework (PDGF) which is Java-based like PSDG, but has an execution model close to the one of Myriad. It uses a hash function based pseudorandom number generator with constant access time which enables efficient substream partition and recomputation between nodes.

In contrast to PDGF Myriad employs an XML-to-source compilation technique and makes extensively use of C++ templates. This ensures a minimal amount of expensive virtual function calls inside the generation loop and offers extensibility at code-level.

7 Discussion

7.1 Permutation

The most dominant operation affecting the execution time is the querying of the PRP to achieve a uniform distribution within a bin. For each tuple that is written, there is one PRP call. Below we show how we constructed a pseudo-random permutation function π within the Myriad framework, which exploits the already implemented PRNG. Under the condition that the PRNG produces numbers with *multiplicative prediction resistance*, we can simply concatenate random numbers and the prediction resistance scales with it in a multiplicative way. For example, concatenating two 64-bit samples that are prediction resistant under multiplication results in a 128-bit number with multiplicative prediction resistance. Let PRNG^{mult} denote the multiplicative prediction resistant generator

$$\mathrm{PRNG}^{mult} : \{0,1\}^q \to \{0,1\}^r.$$

We can construct a PRNG with arbitrary domain size n and seed s by calling the PRNG $\lceil \frac{\log_2 n}{r} \rceil$ times and concatenate its bit representation

$$\mathrm{PRNG}_s^{mult,gen} : \mathbb{N} \to \mathbb{N}, n \mapsto (\mathrm{PRNG}^{mult}(j))_{j=s..\lceil \frac{\log_2 n}{r} \rceil}$$

We can shuffle or permute the virtual tuple identifier in $\Theta(\lceil \frac{\log_2 n}{r} \rceil)$ time by XORing the identifier with the pad resulting from PRNG$_s^{mult,gen}$:

$$\pi : \{0,1\}^{\log_2 n} \to \{0,1\}, (n,i) \mapsto \mathrm{PRNG}_s^{mult,gen}(n) \oplus i. \tag{3}$$

In the field of cryptography, this encryption scheme is known as Vernam's cipher or *one-time pad*, which obtains perfect secrecy [5]. In our case this means that we obtain a shuffle that cannot be distinguished from a 'real' shuffle.

Usually, the domain cardinalities do not exceed 64 bits. Thus, we have to query the PRNG only once and have in fact constant computation time for shuffling a single number. Since the initial seed s and the total number of tuples n are distributed among all nodes, the permutation of the assigned identifier range can be computed locally.

7.2 Discretization

For our composite key generation algorithm we assume as input discretized distributions. Discretizing the target distributions is not restrictive, and natural in two ways. Firstly, when replicating a database instance column distributions are read out in form of histograms collected by the optimizer of a database management system. Secondly, during the data generation process the bit representation of numbers fixes the number of decimal places. In this way, the number of items in a bin is countable, which is exploited by some sampling algorithms.

[5] Under the condition that the pad is used only once and not known to the adversary

8 Conclusion

We gave a description of how to generate tuples with attribute values for which uniqueness holds only on their combination. In the context of databases, this is relevant for a subclass of composite keys or user-defined constraints. Our key generation algorithm therefore extends the capabilities of the already existing parallel data generation frameworks to more complex data. It is completely parallel and can be implemented such that PRNGs with constant access times are utilized.

9 Acknowledgements

The first author would like to thank Christian Lessig for his valuable assistance in editing. Furthermore, we thank the anonymous reviewers for their input that helped to improve the quality of the paper.

References

1. A. Alexandrov, K. Tzoumas, and V. Markl. Myriad: scalable and expressive data generation. *Proceedings of the VLDB Endowment*, 5(12):1890–1893, 2012.
2. N. Bruno and S. Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1097–1107. VLDB Endowment, 2005.
3. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
4. J. Eichenauer-Herrmann. Explicit Inversive Congruential Pseudorandom Numbers: The Compound Approach. *Computing*, 51(2):175–182, June 1993.
5. J. Eichenauer-Herrmann. Statistical independence of a new class of inversive congruential pseudorandom numbers. *Mathematics of Computation*, 60(201):375–384, 1993.
6. J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *ACM SIGMOD Record*, volume 23, pages 243–252. ACM, 1994.
7. J. E. Hoag. *Synthetic Data Generation: Theory, Techniques and Applications.* PhD thesis, University of Arkansas, 2007.
8. J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *ACM SIGMOD Record*, 36(1):19–24, Mar. 2007.
9. G. Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.
10. F. Panneton and P. L'ecuyer. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, Oct. 2005.
11. T. Rabl and M. Poess. Parallel Data Generation for Performance Analysis of Large, Complex RDBMS. *DBTest*, pages 1–6, 2011.

A Composite Keys

Listing 1.1 shows four SQL statements for creating simple schemata. For the sake of simplicity the statements only declare key columns. Table `Simple` has one column `protein` which is declared as primary key and is necessarily unique, i.e. `protein` makes up a simple key. Table `Compound` has two attributes, each making up a simple key in its own right, since they are declared as unique. Tables `Composite1` and `Composite2` are examples of composite key declarations. `Composite1` has only one key attribute which makes up a simple key. Table `Composite2` has even two attributes for which uniqueness exclusively holds for their combination. Possible instances of all four relations are shown below.

Listing 1.1. Table creation in SQL

```
CREATE TABLE Simple(
        protein VARCHAR(50) PRIMARY KEY
);

CREATE TABLE Compound(
        protein VARCHAR(50) UNIQUE,
        aminoacid CHAR(3) UNIQUE,
        PRIMARY KEY(protein, aminoacid)
);

CREATE TABLE Composite1(
        protein VARCHAR(50) UNIQUE,
        aminoacid CHAR(3),
        PRIMARY KEY(protein, aminoacid)
);

CREATE TABLE Composite2(
        protein VARCHAR(50),
        aminoacid CHAR(3),
        PRIMARY KEY(protein, aminoacid)
);
```

`Simple`

rowid	**protein**
1	collagen
2	hemoglobin
3	actin
4	myosin
5	kinesin

`Compound`

rowid	**protein**	**aminoacid**
1	collagen	Gly
2	hemoglobin	Pro
3	actin	Ala
4	myosin	Gln
5	kinesin	Arg

`Composite1`

rowid	**protein**	**aminoacid**
1	collagen	Gly
2	hemoglobin	Gly
3	actin	Pro
4	myosin	Ala
5	kinesin	Pro

`Composite2`

rowid	**protein**	**aminoacid**
1	collagen	Gly
2	collagen	Pro
3	kinesin	Pro
4	collagen	Ala
5	myosin	Ala