Myriad: Scalable and Expressive Data Generation

Alexander Alexandrov[*] Kostas Tzoumas[*] Volker Markl[*]

Technische Universität Berlin, Germany
[*]firstname.lastname@tu-berlin.de

ABSTRACT

The current research focus on Big Data systems calls for a rethinking of data generation methods. The traditional sequential data generation approach is not well suited to large-scale systems as generating a terabyte of data may require days or even weeks depending on the number of constraints imposed on the generated model. We demonstrate *Myriad*, a new data generation toolkit that enables the specification of semantically rich data generator programs that can scale out linearly in a shared-nothing environment. Data generation programs built on top of *Myriad* implement an efficient parallel execution strategy leveraged by the extensive use of pseudo-random number generators with random access support.

1. INTRODUCTION

The vast penetration of online social media and next generation mobile technologies in large areas of our life have, among other factors, caused dramatic growth in the amount of produced digital data in recent years. A recent McKinsey report on Big Data suggests that "the amount of data in the world has been expanding rapidly and will continue to grow exponentially for the foreseeable future" [8]. Since classic database architectures can scale only up to a few hundreds of nodes, a large part of the current industrial and academic research is exploring alternative computational models and architectures that can scale out beyond thousands of nodes a massively parallel environment.

A thorough evaluation of these new large-scale data management concepts entails a substantial amount of performance testing. To this purpose, the use of realistic datasets is of critical importance for the credibility of the test results. Unfortunately, gaining access to real-world datasets can be difficult for various reasons, e.g. privacy protection or data transfer costs. Therefore, many test and benchmarking suites rely on the use of synthetically generated data modeled with realistic statistical characteristics and integrity constraints.

While standard benchmarks like TPC-H and TPC-C [3] provide data generators that have been widely adopted for proof-of-concept experimentation by the database community, analyzing unexpected behavior of production systems often calls for the use of tailor-made test data. In recent years, a number of tools that can generate synthetic data based on a formal input specification have been proposed [5, 6].

Unfortunately, the design of most data generation tools entails a trade-off between the expressiveness of the modeling language (i.e. the ability to synthesize data using various inter-dependant sampling processes) and the efficient scale-out of the data generation process. Obviously, such a trade-off is undesired in the context of Big Data, as it imposes either extremely long generation times (sophisticated data models have to be synthesized on a single machine) or too restrictive data modeling language (e.g. models without cross-table dependencies).

In an attempt to solve this problem we developed *Myriad* – a new data generation toolkit that enables the specification of semantically rich data generator programs that can scale out linearly in a shared-nothing environment. The parallel execution scheme implemented by *Myriad* relies on a subclass of pseudo-random number generators (PRNGs) that support random access of the produced pseudo-random number sequence. The use of these special PRNGs allows distribution of the generation process across arbitrarily many nodes and ensures that they can run completely independent from each other, without imposing any restrictions on the data modeling language.

In the remainder of the paper we provide a brief overview of the core features and architecture of *Myriad* (Section 2) and outline a demonstration proposal that shows how *Myriad* can be used for the definition and generation of both relational and non-relational data (Section 3).

2. MYRIAD: BACKGROUND

This section provides a brief overview of the *Myriad* parallel data generation toolkit. For a more detailed introduction, please refer to our recent paper [4] or visit the toolkit website [2].

2.1 Pseudo-Random Number Generators

Reproducible random sampling methods are based on uniform samples drawn from an underlying *pseudo-random number generator (PRNG)*. From a mathematical point of view, a PRNG is a random walk over the elements of a finite algebraic structure $\mathcal{R}$ (a field or a ring), typically defined recursively by a transition function $s_i = f(s_{i-1})$ and an initial seed s_0[1]. In practice, the s_i values often are normalized to the $[0, 1)$ interval by dividing each number by the modulus of the algebraic structure M_f, i.e. $r_i = s_i / M_f \in [0, 1)$.

Although the definition of most PRNG algorithms suggests only sequential access of the generated random sequence through repeated application of the transition function f, some algorithms

Proceedings of the VLDB Endowment, Vol. X, No. Y

[1] Observe that in these cases obviously $s_i = f^i(s_0)$.

allow skips of arbitrary length (SeedSkips) in constant time, e.g. if f^i can be computed directly or if the sequence is defined directly as a function of the element position $s_i = \tilde{f}(i)$. If sequential and random steps take the same time, we say that the PRNG is *symmetric*, otherwise we say that the PRNG is *asymmetric*. We explain how to exploit PRNGs with SeedSkip support when generating sequences of random domain types in Section 2.3.

2.2 Pseudo-Random Data Generators

Pseudo-random number generators can be extended into pseudo-random sequences of arbitrary domain types.

In *Myriad*, a *domain type* T is a record consisting of l_T base type fields. We support common simple types (integer, double, date, string) as well as lists over simple types and references to other domain types.

For a domain type T with output cardinality C_T and an associated PRNG sequence $(s_i)_{\,i\in\mathbb{N}}$, the *pseudo-random domain type sequence for* T, denoted PRDG_T, is the finite sequence of T-values $(t_i)_{\,0\le i<C_T}$ defined by the mapping of consecutive fix-length chunks of (s_i) to the random values of the corresponding t records.

Let new_T denote the operator that creates a new T-record and $h_{T,i}$ a family of *value setter* functions. The mapping function $\mathbf{h}_T$ can be defined as a chain of k value setters:

$$
\begin{aligned}
h_{T,i} : T \times \mathcal{R} \to T \times \mathcal{R},\ (t,s) \mapsto h_{T,i}(t,s)\\
\mathbf{h}_T : \mathcal{R} \to T,\ s \mapsto h_{T,k-1} \circ \ldots \circ h_{T,0}(\mathrm{new}_T, s)
\end{aligned}
$$

Conceptually, a value setter function h_T receives a partially generated record t and the current state of the underlying PRNG s, draws a fixed number of elements from the PRNG and based on the values of these, it synthesizes the values of one or more fields of t. The function returns the partially overridden t record and the new PRNG state s'.

Obviously, if the underlying PRNG_T supports random access (i.e. arbitrary seed skips), this property propagates to the PRDG_T sequence as we can compute the offset $o(i)$ of the i-th PRNG chunk in constant time and apply the setter chain to obtain $t_i = \mathbf{h}_T(\mathrm{new}_T, s_{o(i)})$.

2.3 Execution Model

Having formalized the required theoretical foundations we now explain how they can be applied in a scalable and efficient data generation process. Random access PRNG and PRDGs are fundamental for shared-nothing execution as they allow us to (a) partition the PRDG sequences horizontally, and (b) decouple the synthesis of two sequences PRDG_A and PRDG_B, even if type A has to sample and access referenced records of type B. These two properties are critical for scalability as they ensure that we can generate arbitrary referenced sequences in a massively parallel environment in a single pass and without communication between the data generator nodes.

Let $(T_j)_{\,0\le j<m}$ denote the family of domain types in the specified domain model. We partition a single underlying PRNG stream (s_i) into j non-overlapping and sufficiently large substreams $(s_{j,i})$ and use them to instantiate the associated family of pseudo-random domain type generators $(\mathrm{PRDG}_{T_j})_{\,0\le j<m}$.

Our execution model is based on the independent execution of N generator nodes. In order to start a generator node the user needs to provide values for the following arguments: (1) the total number of nodes N involved in the parallel generation process, (2) the index of the current node $i, 0 \le i < N$, and (3), the scaling factor s used to determine the actual cardinality of the generated PRDG_{T_j} sequences.

Based on the parameter values we first compute the actual cardinality C_{T_j} of each sequence PRDG_{T_j} and then split each PRDG_{T_j} horizontally into N equally large subsequences of length C_{T_j}/N. The sequences are then distributed across the N generator nodes, with node i responsible for the generation of the i-th subsequence of PRDG_{T_j} for all T_j. Depending on the generator configuration the PRDG_{T_j} subsequences inside a single node can be generated simultaneously (using one thread per subsequence), or in a staged manner.

Figure 1 illustrates the described partitioning scheme for a domain model consisting of two types A (circles) and B (triangles) and an execution environment consisting of three nodes (purple, green and yellow).

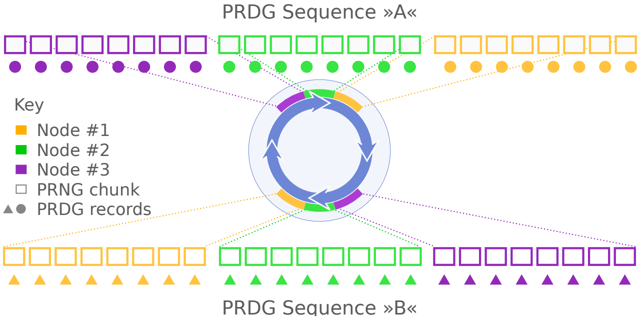

Figure 1: *Myriad* sequence partitioning example

The presented parallelization model is similar to the one proposed by Rabl et al. in [9]. The main difference is in the organization of the PRNG substreams. In [9], a separate substream is maintained per column, which enforces one SeedSkip operation per generated column value. In contrast, our model allocates fix-length chunks from a single PRNG to each record. These are then shared between all setters in the value setter chain, which reduces the total number of SeedSkip operations per record and is more robust against use of PRNGs with asymmetric SeedSkip.

References in the domain model can cause problems if the synthesis of local record fields depends on the field values of the referenced record (e.g. referential integrity constraints on foreign keys). To solve these problems data generators typically fall back to nested generation schemes (i.e. generate all children together with their parent) or multi-pass methods (sample an index from the referenced sequence and join the corresponding record in a second pass). Both solutions have serious disadvantages – nested sampling introduces unwanted artifacts in the generated PRDG sequences (e.g. clustering on the parent key) and can handle only a single parent per child, while multi-pass methods can hamper the data generation efficiency in a large scale parallel setting because of the data-intensive shuffle steps required for the join pass.

In *Myriad*, sampling of referenced records is not restricted by the above partitioning scheme in any way because all PRDG_{T_j} sequences support random access. In general, sampling of referenced records is performed in two steps: (1) sample an index of an element from the referenced PRDG sequence, and (2) recompute the referenced record on the local node. The random access property of the referenced PRDG sequence ensures that all interesting fields of the referenced record can be recomputed on demand on the local node at a constant cost.

Figure 2 illustrates the reference sampling process implemented by *Myriad*. Assume we want to implement the constraint $a.x = b.y$ for a right-unique relation $A \bowtie B$. To implement this constraint, for each each record $a_i \in \mathrm{PRDG}_A$ we first sample the position

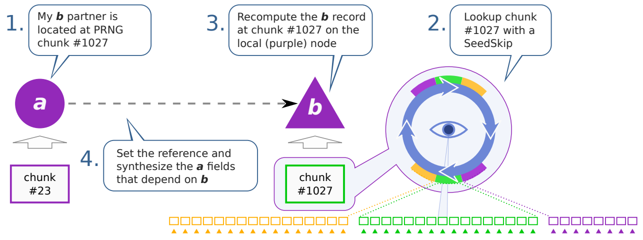
1.
My *b* partner is located at PRNG chunk #1027
3.
Recompute the *b* record at chunk #1027 on the local (purple) node
2.
Lookup chunk #1027 with a SeedSkip
a
b
4.
Set the reference and synthesize the *a* fields that depend on *b*
chunk #23
chunk #1027

Figure 2: *Myriad* reference sampling example

j of the related record $b_j \in \mathrm{PRDG}_B$ (Step 1 in the figure). The b_j must not belong to the PRDG_B-subsequence generated on the same (purple) node as the a_i. We merely adjust the underlying PRNG to the corresponding offset $o(j)$ (Step 2) and compute a consistent clone $\tilde{b}_j = b_j = \mathbf{h}_B(\mathrm{new}_B, o(j))$ locally (Step 3) before we can finally set $a.x := \tilde{b}_j.y$ (Step 4). This way, we can generate correlated values for $a.x$ and $b.y$ in different nodes without transferring data between them.

2.4 Usage Scenario

Figure 3 outlines the standard usage scenario for the *Myriad* toolkit. The user provides a declarative specification of the data generation model using a predefined XML syntax. The core of the XML specification contains the definitions of the generated domain types and the associated value setter chains, i.e. all the information that is required to specify the PRDG sequences described in the previous section. Based on this description the *Myriad* compiler creates a set of C++ classes extending the *Myriad* runtime library. Each PRDG type induces the generation of of corresponding *record class*, *setter chain* and *record generator* extensions. The generated C++ extensions are linked against the core library to produce the final generator executable which represents a single data generation node.

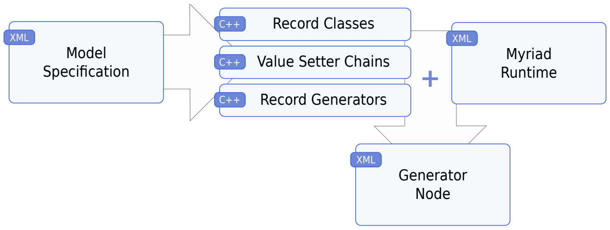

Figure 3: *Myriad* usage scenario

While *Myriad* aims to provide a set of built-in value setters that is expressive enough for most use cases, some generation models might require special value synthesis logic. For these cases, the toolkit provides means to implement this logic directly at the code level by extending the generated record classes and value setter chains.

2.5 Performance

We illustrate the scalability of our parallel execution model with a small experiment using a naive graph generator. In this experiment, we benchmarked the runtime of two alternative PRNG algorithms with random access support against an increasing degree of parallelism on a single machine with 8 CPU cores. Figure 4 shows the results and compares them against a baseline sequential implementation that uses the C++ stdlib *rand* function. The results show linear scale-out for both PRNGs (the sub-linear factor is due to increased I/O congestion and increased node setup over execution time factor for higher N). For *PRNG#2* the payoff against the reference implementation comes for $N > 4$, while *PRNG#1* is faster even for $N = 1$. The reason for this is that *PRNG#2* has a 192 bit cycle and requires more complex computations than *PRNG#1*, which performs only elementary bit-wise operations but is constrained by a 64 bit state.

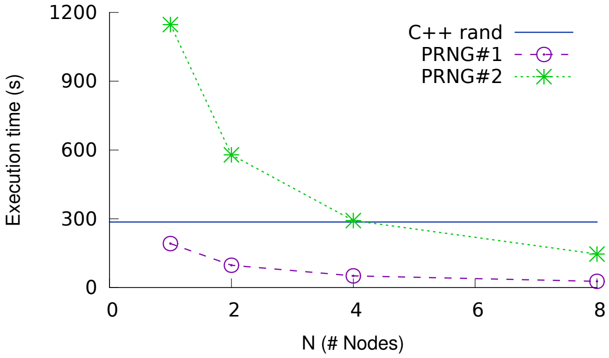

Figure 4: Scale-out for a simple graph model

3. DEMONSTRATION

To showcase the benefits of the described scalable data generation approach we propose a demonstration of the core features provided by the *Myriad* toolkit. As a running example, we use a toy model that combines relational and graph data. First, we show how to specify a generator prototype for a given domain schema and a set of informal model constraints. Upon that, we explain how the generated sources can be extended with user-defined code, e.g. for complex generation semantics that cannot be expressed with the prototype specification language. Finally, we demonstrate how the compiled program can be executed in a shared-nothing environment in two different ways – as a stand-alone application orchestrated by a lightweight coordination service or as a UDF code called by the map tasks in a massively-parallel execution platform like Hadoop [1].

3.1 Generator Model Specification

Figure 5 depicts the toy domain model for our running example. The domain represents a subset of a classical OLAP-style star schema with two fact tables (*Order* and *Lineitem*) and two dimension tables (*Customer* and *Product*). In addition to these relations, the model also incorporates a non-relational extension – a social graph between customers stored in an adjacency matrix format.

Assume that the domain model from Figure 5 imposes the following constraints on the generated data[2]:

1. Both *C.name* and *C.surname* are sampled from fixed name lists according to a predefined discrete probability distribution functions. $P[\mathrm{C.name}]$ is conditioned on *C.gender*.

2. *C.age* has an active domain $[18 : 88]$ and is sampled according to a predefined probability distribution $P[C.age]$.

[2] For all other attributes, assume uniform sampling from a known finite active domain.

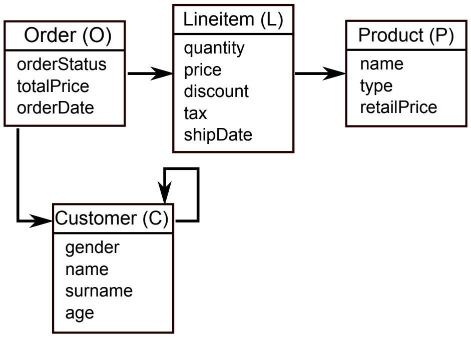

Figure 5: Example schema

3. The references from *Order* to *Lineitem* and *Customer* are sampled independently according to the parametric distributions $P[L \bowtie O]$ (Pareto) and $P[O \bowtie C]$ (normal). The reference from *Lineitem* to *Product* is sampled according to a highly skewed (Pareto or Zipf) distribution $P[L \bowtie P]$.

4. *L.discount* is zero, ten, twenty five or fifty percent of the referenced *P.retailPrice* sampled according to $P[\mathit{L.discount}]$.

5. *L.price* is computed as $(1+\mathit{L.tax})*(\mathit{P.retailPrice}-\mathit{L.discount})$.

6. *O.totalPrice* is computed as the sum of all referenced *L.price* values.

7. *O.orderDate* is a random date from 1 Jan to 31 June 2012 sampled from $P[\mathit{O.orderDate}]$.

8. *L.shipDate* is has a random offset between 1 and 14 days from the associated *O.orderDate*.

9. The *Customer* adjacency matrix produces a graph with realistic features (e.g. small diameter, long tail vertex degree distribution).

During the first part of the demonstration, we show how to translate constraints C1-C8 into a formal *Myriad* specification.

We first explain how our specification language decouples random from functional value synthesis and show how to overcome this restriction with additional domain variables, e.g. *L.discountRate* for C5 or *L.shipDateOffset* for C8. Upon that, we provide a brief overview of the available value setter primitives and show how these can be used to define the value setter chains for the different domain types in line with the specified model constraints (Figure 6).

The final artifact of this part of the demonstration is the XML specification provided to the *Myriad* compiler.

3.2 Code-Level Extensions

Upon creation of the generator source files from our prototype specification, we show the prototype logic can be extended using method overriding at the source code level. To demonstrate this, we show how to augment the record synthesis logic in order to implement the Kronecker graph generation method proposed in [7] in order to fulfill the graph requirements stated in C9.

3.3 Running the Data Generator

We conclude the demonstration by running the data generation program on a small cluster. We use multiple runs with different degrees of parallelism to show the linear scale-out of the produced data generator.

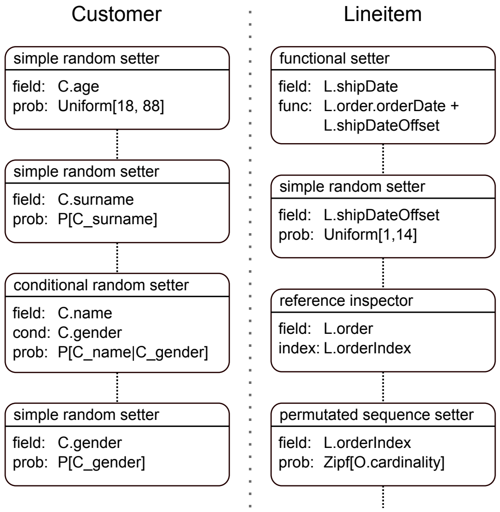

Figure 6: Value setter chains for the Customer and Lineitem domain types (excerpt)

4. CONCLUSION

We demonstrate *Myriad* – a toolkit for expressive data generator programs that can be executed in a massively parallel manner. The demonstration covers the specification, refinement, compilation, and parallel execution of a small OLAP model. We showcase the usage of the toolkit step-by-step and use visualization tools to provide insight into the artifacts produced after each step as well as to show the linear speed-up caused by increasing the degree of parallelism.

5. ACKNOWLEDGMENTS

We thank the IBM Centers for Advanced Studies for supporting this work as part of an ongoing collaboration between TU Berlin and IBM CAS Canada.

6. REFERENCES

[1] Apache Hadoop. URL: `http://hadoop.apache.org/`.

[2] Myriad Parallel Data Generator Toolkit. URL: `http://www.myriad-toolkit.com/`.

[3] Transaction Processing Performance Concil. URL: `http://www.tpc.org/`.

[4] A. Alexandrov, B. Schiefer, J. Poelman, S. Ewen, T. Bodner, and V. Markl. Myriad - Parallel Data Generation on Shared-Nothing Architectures. In *First Workshop on Architectures and Systems for Big Data (ASBD)*, 2011.

[5] J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *ACM SIGMOD Record*, 36(1), 2007.

[6] K. Houkjær, K. Torp, and R. Wind. Simple and Realistic Data Generation. In *VLDB*, pages 1243–1246. ACM, 2006.

[7] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *PKDD*, 2005.

[8] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.

[9] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A Data Generator for Cloud-Scale Benchmarking. In *TPCTC*, 2010.