Myriad – Parallel Data Generation on Shared-Nothing Architectures

Alexander S. Alexandrov[*] Berni Schiefer[†] John Poelman[‡] Stephan Ewen[§]
Thomas O. Bodner [¶] Volker Markl[‖]

Abstract

The need for efficient data generation for the purposes of testing and benchmarking newly developed massively-parallel data processing systems has increased with the emergence of BigData problems. As synthetic data model specifications evolve over time, the data generator programs implementing these models have to be adapted continuously – a task that might become complex as the set of model constraints grows. In this paper we present Myriad - a new parallel data generation toolkit. Data generators created with the toolkit can produce very large datasets by exploiting a completely parallel execution model, while at the same time maintain cross-partition dependencies, correlations and distributions in the generated data. In addition, we report on our efforts towards a benchmark suite for large-scale parallel analysis systems that uses Myriad for the generation of OLAP-style relational datasets.

1 Introduction

In recent years, due to the exponential growth of the volume of Internet traffic, managing and processing large data volumes have become increasingly important both for business and research. Currently, a wide range of new projects influenced by Google's MapReduce [4] are offering a scalable, fault tolerant and cost effective solution to these problems. Some projects provide a programming abstraction in terms of a high-level language [3] or a domain specific API [7] while other extend the underlying parallel programming model [1, 2].

Since all these tools are still in active development, there is a clear need to test, analyze, and evaluate them for both business and scientific reasons. But developing realistic use cases for such systems is challenging, especially as the data required for large-scale testing is either not available due to privacy issues or too large to transfer. What is normally well known or can be estimated, though, is the schema of the data involved in a particular use case scenario. In case of absent real-world data sources, the developer often assumes some statistical model for the data and, based on that, generates synthetic datasets that are used as input. However, developing scalable and efficient data generators is a non-trivial task that may consume a lot of programming effort, especially as a single threaded generation approach is infeasible for parallel setups due to lack of scaling.

To support the developer in the process of specifying and implementing use case specific data generators from scratch, we developed the Myriad parallel data generator toolkit. The toolkit provides a set of reusable components and extension points in order to minimize the time and effort required for the implementation of data generator programs for arbitrary data models. The produced generators support a partition-based execution model which

[*] alexander.s.alexandrov@campus.tu-berlin.de
[†] schiefer@ca.ibm.com
[‡] poelman@us.ibm.com
[§] stephan.ewen@tu-berlin.de
[¶] thomas.o.bodner@campus.tu-berlin.de
[‖] volker.markl@tu-berlin.de

We thank IBM for supporting this work in the course of its Centres for Advanced Studies (CAS) program.

enables linear scale-out on a shared-nothing architecture while preserving the ability for cross-partition sampling of referenced objects. We use a special class of pseudo-random number generators to ensure that no data needs to be moved across nodes even for the realization of complex statistical constraints that require additional information about the referenced records (e.g. correlated values).

The remainder of this paper is structured as follows: Section 2 introduces the basic mathematical concepts behind pseudo-random number generators in the context of generating typed random data. Section 3 describes the fundamentals behind our parallel execution model and section 4 provides an architectural overview of the system. Section 5 presents a benchmark draft for large-scale analysis systems which also served as a first use case for the data generator toolkit. Finally, section 6 provides an overview of the related work and Section 7 concludes with ideas for future development.

2 Technical Background

Before we start with the presentation of the execution model, we introduce some necessary formal background.

A *pseudo-random number generator (PRNG)* is a sequence of integers s_i, typically defined recursively by a transition function $s_n = f(s_{n-1})$ and an initial seed s_0. Since for virtually all transition functions the produced sequence is cyclic, the s_i values are often normalized to the $[0, 1)$ interval by dividing each number by the lower bound (modulus) of the function m_f, i.e. $r_i = \frac{s_i}{m_f}$. From the statistical perspective, the minimal requirement for a useful PRNG algorithm is the uniform distribution of the sequence of r_i values for arbitrary cardinality $i_{\max}$ and initial seed s_0. For applications consuming a large number of random numbers the length of the cycle s_i is of critical importance for the quality of the produced results.

Having formalized pseudo-random number generators we can now introduce the concept of a generator sequence. For an arbitrary data type T with l randomly generated scalar fields, a desired output cardinality C_T and an initial seed $s_{T,0}$, the *generator sequence for* T is a sequence of T values t_i, $i \in 0 \leq i < C_T$ defined by the mapping of subsequent chunks of the PRNG sequence r_i (each of length l) to the random values of the corresponding t record. The mapping can be specified as the product of l *generator functions* (one for each random field):

$$g_T : [0, 1)^l \to T,$$

$$g_T(r_i, \ldots r_{i+l}) = g_{T,0}(r_i) \times \ldots \times g_{T,l-1}(r_{i+l}) \mapsto t$$

3 Execution Model

This section presents some key execution model decisions implemented by the toolkit. We discuss the parallelization strategy behind our scalable generation process. We also present a random record inspection approach used for complex data model constraint types and a randomized de-clustering technique that fits in situations requiring reference sampling from a context-constrained subset.

3.1 Parallelization Strategy

The parallel execution strategy implemented by the toolkit is based on a horizontal partitioning scheme. For each data type, non-overlapping subsequences of the generator sequence are assigned to a single partition. The created partitions can be generated completely independent from each other. The key idea behind our parallelization approach is to use a PRNG algorithm that supports fast computation of arbitrary s_i values, i.e. generator that can derive s_i directly from i instead of applying the transition function f i times on the initial seed s_0.

Consider a model consisting of two data types U and V with cardinality C_U and C_V. Since all the information about the execution environment is available in the bootstrap phase, the driver application can choose how many data generator instances to start and where to place them. The generator sequences for U and V are divided into equally large subsequences of size respectively $c_U := \frac{C_U}{N}$ and $c_V := \frac{C_V}{N}$, where N is the degree of parallelism.

Each subsequence is then assigned one of the N instances according to the following scheme:

#	U-sequence	V-sequence
1	$u_0 : u_{c_U-1}$	$v_0 : v_{c_V-1}$
...		
i	$u_{(i-1)c_U} : u_{ic_U-1}$	$v_{(i-1)c_V} : v_{ic_V-1}$
...		
N	$u_{(N-1)c_U} : u_{C_U-1}$	$v_{(N-1)c_V} : v_{C_V-1}$

In order to ensure that the generated sequences remain invariant to the offered degree of parallelism, all generator instances adjust the starting index for the local s_U and s_V components before they enter the corresponding sequence generation loop.

3.2 Random Field Inspection

In order to realize certain model constraints, the local generator is required to read random values of a sampled referenced record. In the most basic case U references V and the primary key of V is randomly generated by the generator function $g_{V,0}$. The generation logic for the foreign keys in U can be executed in the following three steps: 1) sample an index j for the referenced record v_j; 2) compute $r_{V,j'}$ where j' is the offset of the r_V chunk responsible for the random values of v_j and finally 3) compute the primary key of v_j as $g_{V,0}(r_{V,j'})$. This comes at the cost of a single random s_V access but does not cause any additional network or I/O transfer. Moreover, the basic field inspection logic can be easily generalized and used in more complex scenarios involving multiple fields (e.g. multivariate correlations).

3.3 Clustered Sequences

The last technique we present is used in cases where the sampling range for referenced records is restricted by some random property of the local record. Think about modeling user preferences when generating order lineitems for a retailer data model (as the one presented in Figure 1). We generate a lineitem by first sampling the user and then the product offer. Each user has a random preference type that specifies in which product classes the user is interested. In order to ensure that the generated lineitems comply to this statistical constraint, for each lineitem we have to restrict the product offer sampling range based on the current user preference type[1]. The most efficient way to achieve that is to cluster the product offers sequence by product class. That way we can efficiently exclude product offer subsequences with inappropriate product class entries at runtime.

The major drawback of this clustering strategy is the unrealistic side-effect that is imposed on the generated data sequence (the user might not want the product offers clustered by product class). In such cases, it is advisable to de-cluster the data in a post-processing step. To do this, we reuse a method relying on multiplicative groups to generate permutated sequences of numbers of predefined cardinality [5]. The basic idea is to pick a prime number p slightly larger then the desired cardinality and then to work with the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^{\times}$. Since p is prime, the group is guaranteed to be cyclic and to have exactly $p-2$ elements corresponding to the integers $1 \ldots p-1$. Moreover, the random properties of the group cycle defined by $x_i = g^i$ are strongly influenced by the choice of the generator element g. We can choose an appropriate g, enumerate the values for g^i as an extra field in the clustered sequence and then sort on this field in order to randomize the clustered values. We are currently working on ways to determine good values for g.

4 Architecture

Myriad's extensible architecture consists of six separate modules. These are record objects that serve as data holders, a generator subsystem that produces random record sequences, a configuration module, mathematical tools such as PRNGs and probability distributions, an I/O subsystem exposing the generated records to the client and a simple CLI frontend to control the data generation process. This section describes each but the last component.

Record Objects provide an object-oriented view of the data model. All records share

[1]which can be re-computed for each lineitem using the inspection technique described in Section 3.2

a common *base record type* and contain only simple getter and setter methods (similar to the data transfer objects known from application programming design patterns). Typically, there is one record type for each data source in the data model.

The Generator Subsystem handles the creation of generation sequences for each record type. We provide three *base generator types* for deterministic, static and random sequences of records. The iteration strategy used by the generators is decoupled in the form of *generator tasks* and the random value generation logic for a single record is configured declaratively using a chain of predefined so called *hydrator objects*. The overall generation process for each partition is supervised by a driver program.

The Config Module parses configuration files and extracts data generation parameters, static object sets, and defined probabilities. The information is used by the generators to derive sequence cardinalities, subsequence boundaries, and for hydrator logic setup.

The Math Tools provide common probability functions (e.g. Pareto and Gaussian) and a standard PRNG component. The PRNG implementation currently shipped with the system uses the same algorithm for sequential and random access. This means that the random record inspection described in section 3.2 has the same cost as generating an extra random field per record inside the local sequence.

The I/O Subsystem receives the values of each record and writes them to an output stream. The output stream and data format are thereby fully exchangeable so that the data can either be loaded directly into the target system, e.g. a large-scale data processor or a classic RDBMS, or persistently stored on a local or distributed file system.

5 Use Case

There is a big trend in industry to complement the traditional analysis of relational data in databases with deep analysis of the relational data together with semi-structured as well as unstructured data from additional sources. The latter is typically done with tools like MapReduce. Together, the diverse characteristics of the data and queries pose a variety of different challenges to the analytical systems. To our knowledge, no benchmark today reflects those diverse characteristics and is able to define the sweet spot of different systems, like RDBMSs or MapReduce, in a coherent way.
We used the Myriad toolkit to devise a benchmark scenario, addressing that issue. Due to space constraints, this section gives only a rough overview. For details, we are making the full specification available online. In its core, the benchmark builds upon the TPC-DS benchmark specification. We simplified it and added new tables and different queries. TPC-DS reflects the relational part of the scenario. Our modified version represents a web retailer scenario with customers, orders, order-returns, external re-sellers, web-logs, recommendations, and fraud detection. Figure 1 shows the resulting schema. It contains a representative relational part (star schema warehousing), which also provides data for additional deep analytical queries, like clustering of re-sellers by their profile of offers, or collaborative filtering to recommend products to customers. For the non-relational part, we added server logs, which are frequently used to analyze the searching, browsing, and decision making process of customers. We divided the queries into *six* different categories. Each category contains queries that pose a different kind of challenge to the analytical system:

Embarrassingly Parallel Queries run parallel instances of the query without communicating or exchanging data between them. The result of the query is the union of the different instances' result. We count also queries in this category that require a simple finalizing step, such as the computation of a total sum from partial sums.

Parallel Aggregations require to establish a partitioning on the grouping keys, because either the aggregation functions cannot be decomposed into pre-aggregation and final aggregation, or the number of distinct groups is too

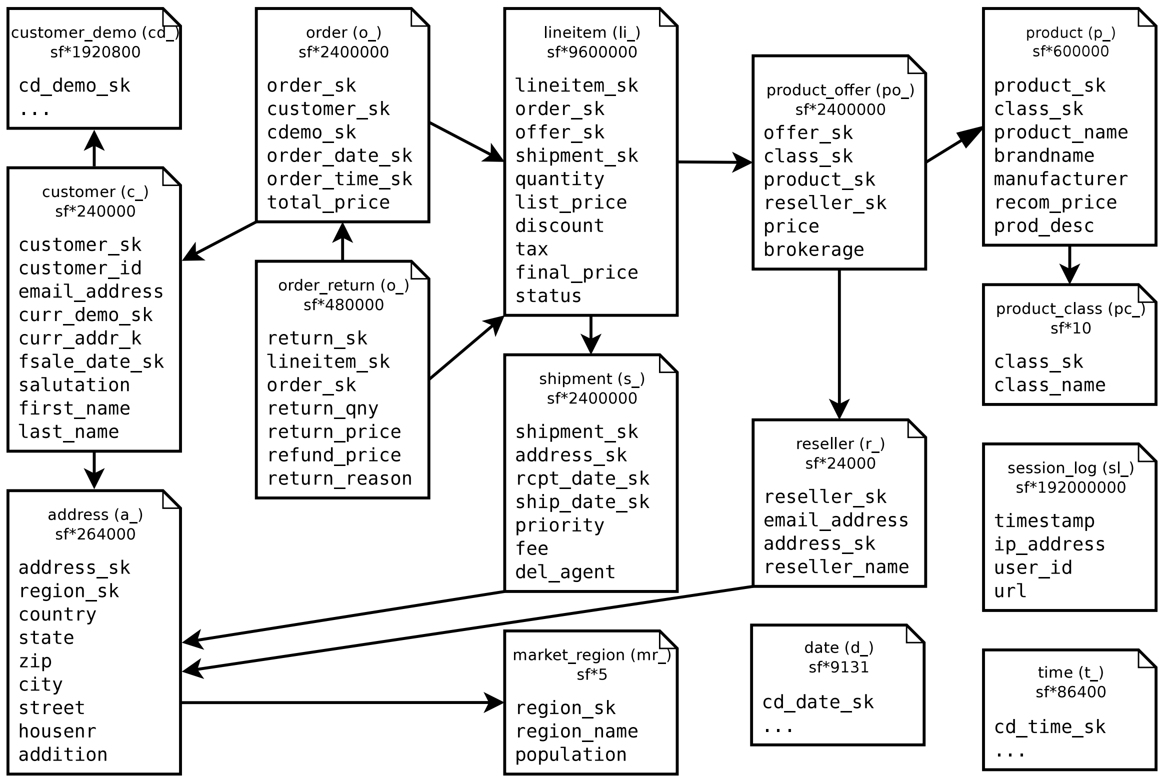

Figure 1: The Benchmark Schema

large to be finally aggregated in an efficient way by a single node.

Parallel Joins come in a variety of flavors. This category contains single joins with varying characteristics, including symmetric and asymmetric input sizes, varying result set sizes and different opportunities to reuse partitionings in the input.

Multi-Join BI Queries extend the previous category, by testing how well the system handles compositions of joins and aggregations. The queries contain longer pipelines and require optimization across multiple joins or aggregations. Examples for such queries are manifold in the original TPC-DS or in the TPC-H specification.

Borderline relational Queries are queries that can be expressed in SQL in a cumbersome way, but don't perform well in parallel. On the other hand, they are expressible for example in MapReduce with relatively little effort. An example is the decomposition of click-streams into sessions, which are the consecutive activity of a single user with a specified maximal delay between two clicks.

Non-relational Queries operate on non-relational data and frequently perform operations that are not expressible in terms of relational operators. In our example, this category contains machine-learning queries for collaborative filtering via matrix factorization, as well as certain clustering algorithms.

6 Related Work

Gray et al. [5] were the first to discuss strategies for scalable parallel data generation, including a scheme for dense unique random data generation which we reuse for the purpose of randomizing clustered values (see Section 3.3). Hoag et al. [6] present a parallel synthetic data generator and an XML-based synthetic data description language (SDDL) similar to the one we intend to implement in our system, but their parallel execution approach is limited by the lack of remote field inspection support. More recently Rabl et al. [8] introduced a parallel data generation framework (PDGF) which implements a similar execution model, most notably the use of a fast SeedSkip operation for sequence partitioning and recomputation of referenced remote fields.

The idea to employ generators with fast SeedSkip support in order to partition the data was inspired by the work of Xu et al. [9] who use a similar technique to facilitate parallel Monte-Carlo simulations in the evaluation of queries on uncertain data in a cluster-computing environment.

7 Conclusion

The data generator provides an easy and efficient means to generate large amounts of data with certain statistical features. It is, however, still quite a challenge to identify and define relevant features, whether in the course of designing a benchmark dataset, or when creating a synthetic dataset that is supposed to reflect the statistical properties of an existing one.

In the future, we want to improve the data generator suite in a two fold way: First, we will add a lightweight way of specifying those distributions and correlations in a profile, such that no code has to be written to adopt the generator. Second, we plan to add a tool that, given a dataset and workload, automatically extracts the relevant statistical properties of the dataset and generates a profile for the data generator. The generator can use that profile to create a synthetic dataset that reflects the statistical features of the original dataset.

The current version of the Myriad toolkit is available at *http://www.myriad-toolkit.com*.

References

[1] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[2] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[3] Kevin Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh Carl-Christian Kanne, Fatma Ozcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 2011.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[5] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *ACM SIGMOD Conference*, 1994.

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

[7] Apache Mahout. URL `http://mahout.apache.org`.

[8] Tilmann Rabl, Michael Frank, Hatem Mousselly Sergieh, and Harald Kosch. A Data Generator for Cloud-Scale Benchmarking. In *TPCTC*, 2010.

[9] Fei Xu, Kevin Beyer, Vuk Ercegovac, Peter J. Haas, and Eugene J. Shekita. $E = MC^3$: managing uncertain enterprise data in a cluster-computing environment. In *ACM SIGMOD Conference*, 2009.