Myriad – Parallel Data Generation on Shared-Nothing Architectures

Alexander Alexandrov*1 Berni Schiefer†2 John Poelman‡3
Stephan Ewen*1 Thomas O. Bodner*4 Volker Markl*1

*TU Berlin
Germany
[1]firstname.lastname@tu-berlin.de
[4]thomas.o.bodner@campus.tu-berlin.de

†IBM Toronto Lab
Canada
[2]lastname@ca.ibm.com

‡IBM Silicon Valley Lab
United States
[3]lastname@us.ibm.com

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 Big Data problems. As synthetic data model specifications evolve over time, the data generator programs implementing these models have to be adapted continuously – a task that often becomes more tedious 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 quickly produce very large datasets in a shared-nothing parallel execution environment, while at the same time preserve with 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.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Software Engineering, Testing and Debugging, Testing Tools, Scalable Data Generation

Keywords

Scalable Data Generation Myriad Parallel Data Generator Toolkit

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 architecture [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 and execution runtime [1, 2].

Since all these tools are still in active development, there is a clear need to test, analyze, and evaluate them. But developing realistic use cases for such systems can be a challenging task, especially if the Big Data for the use case is not available, e.g. due to privacy concerns or because at this level of magnitude data transfer over the Internet is simply too expensive. What is normally well known, though, is the schema for a particular use case. In case of absent real-world datasets, developers often assume some statistical data model and generate corresponding synthetic data that is used for performance testing. However, developing scalable and efficient data generators is a non-trivial task that may consume a lot of programming effort, especially as it requires careful consideration of advanced programming aspects such as concurrency, synchronization, and sampling from various probability distributions.

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 components and extension points that can be reused in order to minimize the time and effort required for the implementation of data generator programs for user-defined data models. The produced generators implement a partition-based execution model that enables linear scale-out in a shared-nothing environment. We use a special class of pseudo-random number generators to facilitate parallelization and ensure that even complex statistical constraints such as correlated values between records can be achieved without moving data across nodes.

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 pseudo-random sequences of user-defined domain types. 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASBD October 10, 2011, Galveston Island

Copyright 2012 ACM 978-1-4503-1439-8/12/04 ...$15.00.

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

We start with a brief review of pseudo-random number generators, and then show how these gan be generalized to pseudo-random sequences of arbitrary domain types.

A *pseudo-random number generator (PRNG)* is a sequence of integers $s_i \in \mathfrak{R}$, 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 upper bound (modulus) of the function m_f, i.e. $r_i = s_i/m_f$. From a statistical perspective the main requirement for a good PRNG algorithm is the uniform distribution of the r_i values for arbitrary cardinality $i_{\max}$ and initial seed s_0. For algorithms consuming a large number of random numbers the length of the r_i cycle is of critical importance for the quality of the algorithm results.

To formulate a theoretical framework for the supported user-defined data generator programs, we embed the PRNGs concept into a broader set of pseudo-random sequences for arbitrary user-defined domain types. Let T be a user-defined record type with l scalar fields, C_T be the desired sequence cardinality, and r_0^T be the initial seed of the associated underlying PRNG subsequence. We define the *pseudo-random domain type generator for T* (denoted PRDG_T) as the sequence of T records $(t_i)_{i \in 0 \leq i < C_T}$ obtained by mapping adjacent fixed-length PRNG chunks from r^T to the random field values of the corresponding t record.

We refer to the mapping function $\mathbf{g}^T : \mathfrak{R} \to T$ as the *value setter chain* for T. Value setter chains are constructed as concatenation of primitive *value setters*, where each value setter $g_i^T : T \times \mathfrak{R} \to T \times \mathfrak{R}$ consumes a fixed number of elements from the PRNG structure $r^T \in \mathfrak{R}$, synthesizes a field in the record t and returns the updated (t, r^T) arguments pair. Formally, this means that $\mathbf{g}^T = \pi_T \circ g_{l-1}^T \circ \ldots \circ g_0^T \circ \mathbf{new}_T$, where $\mathbf{new}_T$ is the T record constructor and π_T projects the first component of the obtained $T \times \mathfrak{R}$ pair.

3. EXECUTION MODEL

This section covers key runtime-related aspects of the Myriad toolkit. We explain our scalable parallel execution strategy and present two complementary techniques that facilitate the lightweight implementation of complex data generation constraints.

3.1 Parallelization Strategy

Myriad uses horizontal partitioning for parallel data generation. Assume a parallel setup with n data generator nodes. For each data type T, the corresponding PRDG_T sequence is range-partitioned into n equally long adjacent subsequences. During the the initialization phase, all data generator nodes compute the boundaries of their subsequences and upon that generate them completely independently from each other. For an efficient implementation of this partition-based parallelization strategy we use only PRNGs that support direct computation of r_i for an arbitrary index i, i.e. where r_i is derived directly from i rather than implicitly by i applications of the transition function f on the initial seed s_0. This means we can adjust the starting positions of all generated PRDG subsequences for a particular node in constant time.

Consider a domain model with two data types U and V and cardinality C_U and C_V. The PRDG sequences for both types are divided into equally large subsequences of size $c_U := C_U/n$ and $c_V := C_V/n$, where n is the number of data generator nodes, and then assigned to a generator node according to the following schema:

Node#	PRDG_U subsequence	PRDG_V subsequence
1	$[u_0, u_{c_U-1}]$	$[v_0, v_{c_V-1}]$
$\vdots$	$\vdots$	$\vdots$
i	$[u_{(i-1)c_U}, u_{ic_U-1}]$	$[v_{(i-1)c_V}, v_{ic_V-1}]$
$\vdots$	$\vdots$	$\vdots$
n	$[u_{(n-1)c_U}, u_{C_U-1}]$	$[v_{(n-1)c_V}, v_{C_V-1}]$

All generator nodes will adjust the starting position of their local PRNG components r^U and r^V before they enter the generation loops for the two PRDG sequences.

3.2 Re-Computing Referenced Records

To realize certain constraints the value setter chain for a sequence PRDG_U may be required to access referenced records from an associated sequence PRDG_V that in general are generated on a different node. Consider an example where you want to set proper foreign keys to V in U. A naive implementation could achieve this in a two passes: generate the two sequences (without the FKs), shuffle each u to its associated v record, and finally read all v records and set the FKs in the co-located u records. This solution is inefficient because the shuffle step causes extra I/O and network overhead and enforces a global synchronization barrier.

Myriad overcomes this problem with the help of random access PRNGs. The foreign key setter logic is implemented in a single pass in Myriad as follows: (1) sample an index j for the referenced record v_j; (2) adjust the r^V position to $o(j)$ – the offset that marks the start of the PRNG chunk responsible for v_j; (3) use the value setter chain for V to locally re-compute the v_j, i.e. set $v_j = \mathbf{g}^V(r^V_{o(j)})$. This strategy requires a single random access operation on r^V in step (2), but avoids the additional network or I/O overhead which we have in the naive two pass implementation.

The sample technique for local re-computation of referenced records can be used to realize more complex data distribution constraints, for example correlations between record fields that belong to connected record pairs.

3.3 Clustered Sequences

The second technique is used in cases where the set of records eligible for reference sampling is restricted by a particular field value. Think about modeling user preferences when generating orders for a retailer domain model (as the one presented in Figure 1). To generate a *lineitem* we first sample an enclosing *order* and then pick a random *product offer*. Each *order* is associated with a particular *customer*, and each *customer* has a preference type that specifies the *product classes* in which he or she is interested. To ensure that the generated *lineitems* respect this constraint, we have to restrict the sampling range for the *product offers* based

on the preference type of the current *customer*[1]. The most efficient way to achieve that is to cluster the *product offers* PRDG sequence by *product class*, so we can identify the *product offers* with the desired *product class* at runtime.

The major drawback of this strategy is the unrealistic clustering side-effect imposed on the clustered sequence (for example, the user might not want the product offers clustered by product class). In such situations, the user can 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 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 $1 \ldots p-1$. We can choose an appropriate generator element g for the group cycle $x_i = g^i$, enumerate g^i as extra field in the clustered t_i and then sort on this field to randomize the clustered values.

4. ARCHITECTURE

Myriad's extensible architecture consists of six separate modules. These are *record objects* for the domain model, a *generator subsystem* that produces random record sequences, a *configuration module*, *mathematical tools* such as PRNGs and probability distributions, an *I/O subsystem* that formats the generated records and a simple *CLI frontend* for multi-process execution. This section describes each but the last component.

Record Objects provide an object-oriented view of the domain model. All records share 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). There is one record for each type in the domain model.

The Generator Subsystem handles the creation of all domain type sequences. Besides for *random*, we also provide support for *deterministic* and *static* record sequences. The value setter chains for the random sequences are defined as a chain of *value setters* in line with the PRDG framework presented in Section 2. The overall generation process for each node is supervised by a *driver program*.

The Config Module parses configuration files and extracts user-defined data generation parameters, static record sets, and probabilities. The information is used by the generators to derive sequence cardinalities, subsequence boundaries, and to instantiate the value setter chains.

The Math Tools provide common probability functions (e.g. Pareto and Gaussian) and a standard PRNG component with.

The I/O Subsystem receives synthesized records and writes them to an output stream. The output stream and record format are 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 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 fil-

[1]To obtain the referenced *order* and *customer* records we use the re-computation technique described in Section 3.2

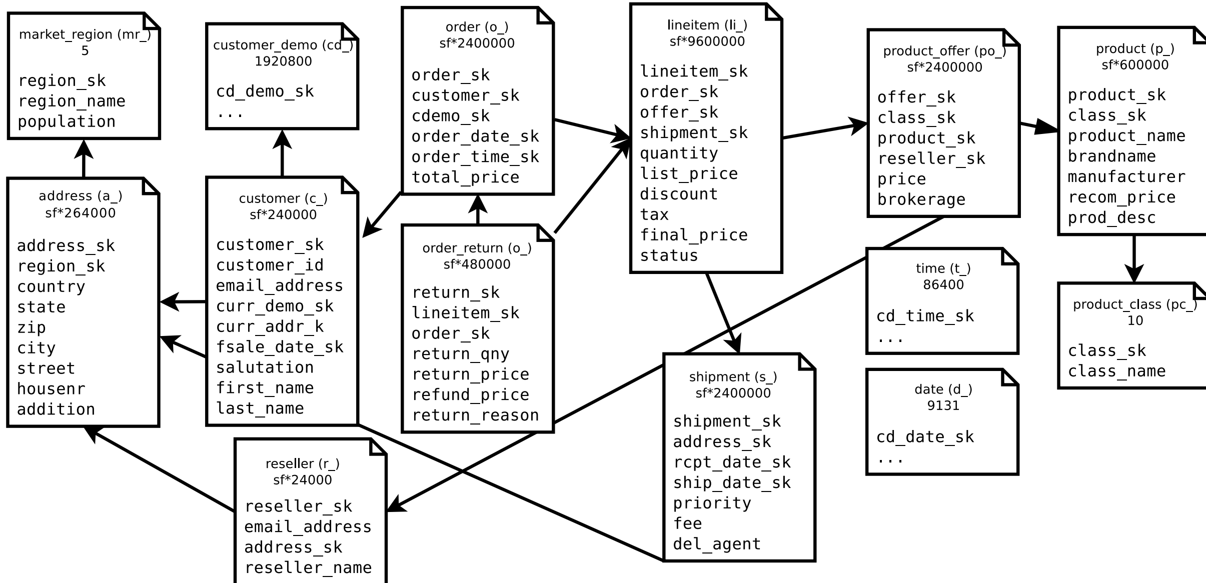

Figure 1: Schema for the Use Case Benchmark

tering 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 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 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 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*.

8. REFERENCES

[1] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. 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] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[3] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. E. C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 2011.

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

[5] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *ACM SIGMOD Conference*, 1994.

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

[7] A. Mahout.

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

[9] F. Xu, K. Beyer, V. Ercegovac, P. J. Haas, and E. J. Shekita. $\text{E} = \text{MC}^3$: managing uncertain enterprise data in a cluster-computing environment. In *ACM SIGMOD Conference*, 2009.