University of Magdeburg

School of Computer Science

Master's Thesis

Configurable Testbeds for Cloud Data Management Systems

Author:

Sebastian Schultz

December 11, 2014

Advisors:

Dr. Ing. Eike Schallehn

Database and Software Engineering Workgroup

Schultz, Sebastian:
Configurable Testbeds for Cloud Data Management Systems
Master's Thesis, University of Magdeburg, 2014.

Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BASE	Basically Available, Soft state, Eventually consistent
C*	Apache Cassandra
CDMS	Cloud Data Management System
CQL	Cassandra Query Language
DBMS	Database Management System
i.i.d.	identically distributed
JSON	JavaScript Object Notation
PRNG	Pseudo-Random Number Generator
PyPI	Python Package Index
RDBMS	Relational Database Management System
RDG	Random Data Generator
REST	Representational State Transfer
RNG	Random Number Generator
SCLA	Strongly Consistent, Loosely Available
SQL	Structured Query Language
TPC	Transaction Processing Performance Council
tps	transactions per second
UML	Unified Modeling Language

UUID	Universally Unique Identifier
YAML	YAML Ain't Markup Language
YCSB	Yahoo! Cloud Serving Benchmark

1. Introduction

The recently increasing collection of all kinds of data (e.g. from websites, mobile phones and sensors) naturally led to increased amounts of data (also known as *big data*) that need to be managed. As this amount of data often poses great challenges to traditional Relational Database Management System (RDBMS), and as other restrictions to the management of such data have emerged (e.g. increased availability instead of imperative consistency), new approaches to manage such data were needed. From these new challenges the NoSQL movement arose, aiming to provide scalable Cloud Data Management Systems (CDMSs).

Developing performant schemata has not been easy for traditional databases, and it got even more complicated with the data models introduced by the new kinds of CDMSs. Not only does a database have to represent all participating entities, it also has to be adapted to the access schemata that are intended to be used. Furthermore, it has to be scalable to store terabytes of data and still offer good performance.

Benchmarks are used to gain insight of which database performs best for the task at hand. Benchmarking of traditional databases is possible because most of them use the same relational data model and a similar query language to access data. This property changed for CDMSs, as not only their used data models often differs greatly, but also the way data can be stored and accessed. Also, the different CDMS implementations aim for different goals. While some might focus on availability, others might focus on scalability. Hence, a comparison of the performance of different CDMSs is difficult and still a research field with many open questions and problems.

Load testing individual schemata designed for specific application scenarios was already difficult for RDBMSs (e.g. because of complex dependencies within the data), but got even more difficult for systems managing vast amounts of data. The newly developed CDMSs are designed to handle high data amounts, hence they have to be tested using such hight amounts of data. Furthermore, the repeatability of tests needs to be retained, and realistic data with possibly complex dependencies has to be used.

Goal of this Thesis

It is the goal of this thesis to design a framework for load testing CDMSs using application specific schemata and workloads. Furthermore, it could be used in research for reproducible experiments or for benchmarking different CDMSs, though the latter poses some additional requirements that are not exhaustively considered in this thesis.

The developed framework will be trying to support different kinds of data models. Furthermore, it will be designed to allow extensions, which could be used to support further emerging data models. Also, it will be scalable, as it aims to stress scalable data management systems. Therefore, it will be determined which requirements such a tool has to fulfill, and what pitfalls might have to be avoided. The described approaches will also be evaluated on the posed requirements by using an implemented prototype, which will also be compared to a related stress tool that has been published in the course of this thesis. The developed prototype is also freely available[1] for further studies, extensions, and general usage.

Structure of the Thesis

First, the knowledge, terms and techniques required for the understanding of this thesis will be explained briefly. Afterwards, a basic concept for the design of a configurable testbed for scalable data management systems will be presented. This chapter is followed by descriptions of the implemented prototype. Lastly, the prototype will be evaluated, after which conclusions of this thesis will be drawn and open questions for future work will be described.

[1] https://github.com/causa-prima/COLT

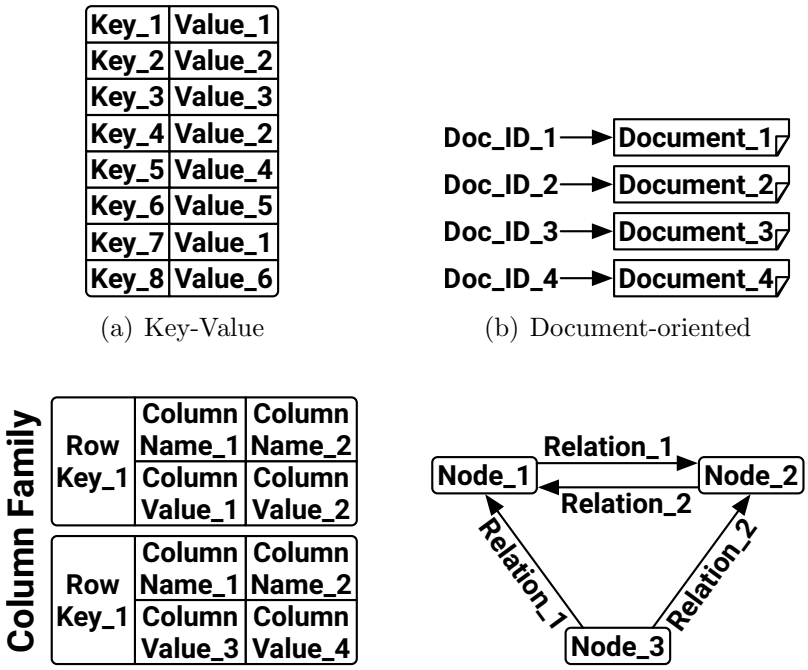
2. Background

This chapter briefly covers the required knowledge, terms and techniques for understanding the following chapters. It starts by explaining the motivation for and the mechanics of NoSQL systems, with finer details on the database used in this thesis. Then, it is looked at the field of Database Management System (DBMS) benchmarking, which is closely related to the work of this thesis. Finally, the generation of random numbers will be explained, as it is at the heart of the developed prototype.

2.1 NoSQL

In the past decade the continuously growing number of internet users and the associated growth of web services like Facebook, Twitter, Spotify, eBay et cetera increased the need for a new approach to data storage. New requirements, like massive data volume, scalability, availability, fault tolerance and elasticity, all while maintaining high performance, pose a challenge to traditional RDBMSs [Gro+13]. From this demand the so-called NoSQL movement emerged, promising to provide the following properties often attributed to them [HJ11; SF12; Coo+10; Kon+11; Cat10]:

- **Horizontal scaling:** The data store should be able to scale horizontally (i.e. over many commodity servers), providing data scaling and/or read/write scaling.
- **Non-relational data model:** Providing flexible schemata or even schemalessness to handle a wide variety of data structures.
- **Partition tolerance:** Because many NoSQL data stores aim to support highly distributed scenarios, they often choose to increase availability by sacrificing consistency.
- **Weaker consistency model:** The Atomicity, Consistency, Isolation, Durability (ACID) approach is typically replaced by Basically Available, Soft state, Eventually consistent (BASE) [Pri08], Strongly Consistent, Loosely Available (SCLA) or even tunable consistency.



(a) Key-Value (b) Document-oriented

(c) Wide Column (d) Graph

Figure 2.1: Different types of NoSQL data models, adaption of [Gro+13, Figure 1].

It is not mandatory for a database system to provide all of these properties to be categorized as NoSQL system as there is no agreement on what exactly constitutes such systems [Gro+13]. However, most of them possess one or more of these characteristics.

A widely-used approach to classify NoSQL systems is using their different data models [Gro+13; HJ11; MBS12], illustrated in figure 2.1:

Key-value: A simple data model in which a key is used to uniquely identify a value as well as to store it in and retrieve it from the data store. Examples: Redis, Memcached

Document-oriented: Similar to the key-value model keys are used to locate documents inside the data store, mostly using JavaScript Object Notation (JSON) or derived formats as document representation. Examples: MongoDB, CouchDB

Wide Column: Contrary to the row-oriented model of traditional RDBMSs, data is stored in a column-oriented way. Examples: Cassandra, HBase

Graph: Based on the mathematical concept of graphs, objects are denoted as vertices (or nodes) and links (or edges) describing relationships among them. Examples: Neo4j, Titan

To achieve scalability, key oriented NoSQL systems use partitioning (also known as sharding) on key level [HJ11; MBS12; Cat10]. Using range based partitioning, the

key space is divided into intervals, which each get assigned to one specific partition. Alternatively, the output of a hash function applied on a key determines the partition of the key – a technique called consistent hashing [Kar+99].

For communication with the database there are query languages and Application Programming Interfaces (APIs) that differ in richness of their provided functionalities [HJ11]. For key-value stores a query language would be unnecessary overhead, as they only provide key based put, get and delete operations that can be handled by a simple API. The more complex column oriented stores offer APIs and Structured Query Language (SQL)-like query languages with range queries, operations such as "in", "where", "and/or" and regular expressions, but only if they are applied on row keys or indexed values. In contrast, document stores offer range queries on values, secondary indexes, operations like "and/or", between and querying nested documents via APIs or Representational State Transfer (REST) interfaces. A query for a graph database can be done using graph pattern matching strategies or graph traversals, both of which can be performed using program language specific interfaces, store specific interfaces or REST APIs.

2.1.1 Apache Cassandra

> *"Apache Cassandra is an open-source, distributed, decentralized, elastically, scalable, highly available, fault-tolerant, tuneably consistent, column-oriented database that bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable. Created at Facebook, it is now used at some of the most popular sites on the Web."* [Hew10]

Originating from ideas used in Amazon Dynamo and Google Bigtable, and initially developed at Facebook [LM10], Apache Cassandra (C*) is now an open source project curated by the Apache Foundation. Due to its decentralized design, in which every node in the cluster fulfills the same tasks, it has no single points of failure. As a distributed system for multi-node multi-datacenter deployment it is tailored towards redundancy, for failover and disaster recovery. Configurable replication is done automatically and possible across multiple data centers. This allows for failed nodes to be replaced with no downtime. As new machines are added to a C* cluster (also with no downtime), both read and write throughput increase linearly.

C*'s eventual consistency is tunable on a cluster, data center, or individual I/O operation basis. Different consistency levels are provided for read and write operations, each ranging from "`ALL`" (all replica nodes in the cluster must agree) to "`ANY`" (at least one replica node must agree). In addition, C* 2.0 introduced linearizable consistency called "lightweight transactions"[1].

[1] http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0, accessed 2014-10-19

The C* data model, belonging to the wide column model depicted in figure 2.1(c) on the preceding page, is best described by the Apache-approved documentation from DataStax[2]:

> *"Cassandra's data model is a partitioned row store with tunable consistency. Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Other columns can be indexed separately from the primary key. Tables can be created, dropped, and altered at runtime without blocking updates and queries."* [3]

For accessing and manipulating tables and data C* initially only offered an API called *Thrift* that provides a low-level interface to the database. Later, the Cassandra Query Language (CQL) was introduced, offering a simpler and more intuitive SQL-like interface for data and table manipulation. A short example showing table creation, data insertion and table querying using CQL is presented in listing 2.1.

It should be emphasized that C* does not support joining tables, as it does not make use of normalization. Hence, this task is given to the user, who is encouraged to make use of schema denormalization and data duplication for high performance.

The CQL data types cover basic types for strings (`ascii`, `text`, `varchar`), numbers (`bigint`, `counter`, `decimal`, `double`, `float`, `int`, `varint`), truth values (`boolean`), binary objects (`blob`), and points in time (`timestamp`). In addition, it has types for IP addresses (`inet`) and Universally Unique Identifiers (UUIDs) (`uuid`, `timeuuid`). Furthermore, it has special types for collections of any of these basic data types, namely `list`, `set` and `map`, with which schemata can easily be de-normalized further.

2.2 Benchmarking of DBMSs

Benchmarks are used to answer the common question about the best system in a given domain. In this context, *"the best"* usually means a superior performance in one or more of the desired properties of the given domain, e.g. the fastest algorithm to sort an array, or the shortest response times to queries [FK09]. This implies that a benchmark has to choose a scenario that is representative for the given domain by defining the rules for construction and running of the scenario [Fol+13].

In the field of DBMSs, increasing standardization of relational systems led to performance as main differentiating factor among vendors [SKS11]. While the NoSQL approach introduced new distinctive features, their performance is still a main factor for comparison. These needs for grading different DBMSs led to the development of many benchmarks, two of which will be presented in section 2.2.3 on page 9. But first, general benchmark requirements will be presented shortly, followed by a description of the overall DBMS benchmark process in section 2.2.2 on the following page.

[2] http://www.datastax.com/docs, accessed 2014-10-19
[3] http://datastax.com/documentation/cql/3.1/cql/ddl/ddl_intro_c.html, accessed 2014-10-19
[4] http://www.datastax.com/dev/blog/cql-in-2-0-6, accessed 2014-11-17

```
1  CREATE TABLE timeline (
2      day TEXT,
3      hour INT,
4      min INT,
5      sec INT,
6      value TEXT,
7      PRIMARY KEY (day, hour, min, sec)
8  );
9
10 INSERT INTO timeline (day, hour, min, sec, value)
11                VALUES ('12␣Jan␣2014', 3, 43, 12, 'event1');
12 INSERT INTO timeline (day, hour, min, sec, value)
13                VALUES ('12␣Jan␣2014', 3, 52, 58, 'event2');
14 INSERT INTO timeline (day, hour, min, sec, value)
15                VALUES ('12␣Jan␣2014', 4, 37, 01, 'event3');
16 INSERT INTO timeline (day, hour, min, sec, value)
17                VALUES ('12␣Jan␣2014', 4, 37, 41, 'event3');
18
19 SELECT * FROM timeline WHERE day='12␣Jan␣2014'
20                        AND (hour, min) >= (3, 50)
21                        AND (hour, min, sec) <= (4, 37, 30);
22 /* Output:
23  *   day          | hour | min | sec | value
24  *  ---------------+------+-----+-----+--------
25  *   12 Jan 2014 |    3 |  52 |  58 | event2
26  *   12 Jan 2014 |    4 |  37 |   1 | event3
27  */
```

Listing 2.1: CQL example, excerpt from blog posting[4]

2.2.1 General Benchmark Requirements

There has been a number of publications trying to provide guidelines for benchmark design and implementation [Gra93; Kou05; Sac+09; Hup09; Sac10]. Based on this work, [Fol+13] define the following three groups of requirements:

1. **General Requirements:**
   - *Strong Target Audience:* A target audience of considerable size interested to obtain the information is needed.
   - *Relevant:* The performance of the typical operation within the problem domain has to be measured.
   - *Economical:* It should be affordable to run the benchmark.
   - *Simple:* understandable benchmarks create trust.
2. **Implementation Requirements:**
   - *Fair and Portable:* All compared systems can participate equally.

- *Repeatable*: Rerunning the benchmark under similar conditions produces the same results.
- *Realistic and Comprehensive:* all features typically used in the target applications are exercised.
- *Configurable:* A flexible performance analysis framework should be provided to allow users the configuration and customization of the workload.

3. **Workload Requirements:**

   - *Representativeness:* The benchmark should be based on a workload scenario that contains a representative set of interactions
   - *Scalable:* Scalability should be supported in a manner that preservers the relation to the real-life business scenario modeled.
   - *Metric:* To report about the tested application's reaction to the load a meaningful and understandable metric is required

As this thesis does not only cover the development of benchmarks for DBMSs, but a superset of that, these general requirements will be used to evaluate the chosen approaches in section 5.1 on page 49. However, as this thesis does not aim to provide comparability between different kinds of databases, the implementation requirement to treat different DBMSs fairly is not of great importance, as a comparability between those is not a focus of this work. However, it should be noted that, if carefully designed, specific workloads *could* be used to compare different kinds of DBMSs, but it would be difficult to guarantee that these comparisons are fair as the optimality of the implemented modules used for each database would have to be proven.

2.2.2 DBMS Benchmark Process

The process of benchmarking a DBMS can be divided into several steps [BK98]:

1. **Objective definition:** As objectives may vary according to particular requirements of databases, users, and other components of the database environment, the benchmark's objectives are defined.
2. **Customization:** The test environment (e.g. database and benchmarking application) are configured according to the defined benchmark test objectives.
3. **Setting execution parameters:** Arguments for the benchmarking process (e.g. termination conditions, database size) are defined and applied by the application.
4. **Transaction generation:** The queries needed by the program are generated.
5. **Communication:** Generated transactions are submitted to the database.
6. **Gather information:** Generate and store transaction information, then loop back to step 4 as many times as necessary.

7. **Generate reports:** The information wanted for analysis is generated by processing the gathered data, and is presented to the user afterwards.

8. **Analysis:** The user analyzes the benchmarking output.

According to Rabl and Poess [RP11] it is becoming increasingly difficult to run realistic benchmarks to test the performance of today's systems because of the exponential growth in the amount of data retained by them, the number of systems that need to participate in the data generation, and the complex structure of the data. They also state that modern data generators are required to generate data deterministically, in parallel processes/threads, in parallel across address spaces of multiple systems, and generate complex data sets. In addition, data generators need to adapt to different schemata and need to be easy to set up. As a basis for synthetic data generation, all data types are generated using pseudo-random numbers (see section 2.3 on page 12), which can be generated deterministically. Also, some of these generators have been developed to be parallelizable [RP11; Sal+11], which is a feature availed to efficiently generate data for large data sets.

2.2.3 Examples of DBMS Benchmarks

In this section, two benchmarks for DBMSs are described. The first (TPC) is a set of well-known and extensively used benchmarks developed for relational DBMSs. The second (YCSB) is a new contestant developed with the goal of facilitating performance comparisons of the new generation of cloud data serving systems.

TPC Benchmarks

Founded in 1988 to define transactional processing and database benchmarks and to disseminate objective, verifiable performance data from those benchmarks to the industry, the Transaction Processing Performance Council (TPC)[5] has defined a series of benchmarks standards for database systems. For each benchmark they define the set of relations and the size of tuples. The number of tuples in the relations are defined as a multiple of the number of claimed transactions per second to reflect the likely correlation of a larger rate of transactions with a larger number of accounts [SKS11]. The used metrics usually include throughput, e.g. expressed as *transactions per second (tps)*, and cost, e.g. in terms of *price per tps*, while other metrics are defined if they are useful for the benchmarked domain. A notable attribute of the TPC benchmarks is that a company cannot claim TPC benchmark numbers for its system without an external audit ensuring that the benchmark's definition has been exactly followed.

Since its establishment, the TPC defined different benchmarks, some of which are now obsolete. Hence, only current ones will be presented, with descriptions taken from the TPC's benchmark overview[6]:

[5] http://www.tpc.org, accessed 2014-11-15
[6] http://www.tpc.org/information/benchmarks.asp, accessed 2014-11-15

- **TPC-C:** Centered around the principal activities of an order-entry environment, where transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses.

- **TPC-DS:** Decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance.

- **TPC-E:** Models a brokerage firm with customers who generate transactions related to trades, account inquiries, and market research while the brokerage firm in turn interacts with financial markets to execute orders on behalf of the customers and updates relevant account information.

- **TPC-H:** Decision support benchmark consisting of a suite of business oriented ad-hoc queries and concurrent data modifications, illustrating decision support systems that examine large volumes of data, execute queries with a high degree of complexity and give answers to critical business questions.

- **TPC-VMS:** Intended to represent a virtualization environment where three data"-base workloads are consolidated onto one server.

- **TPCx-HS:** Developed to provide an objective measure of hardware, operating system and commercial Apache Hadoop File System API compatible software distributions.

As an example, the schema of TPC-H is given in figure 2.2. Also, one of the twenty-two queries of that benchmark is shown in listing 2.2.

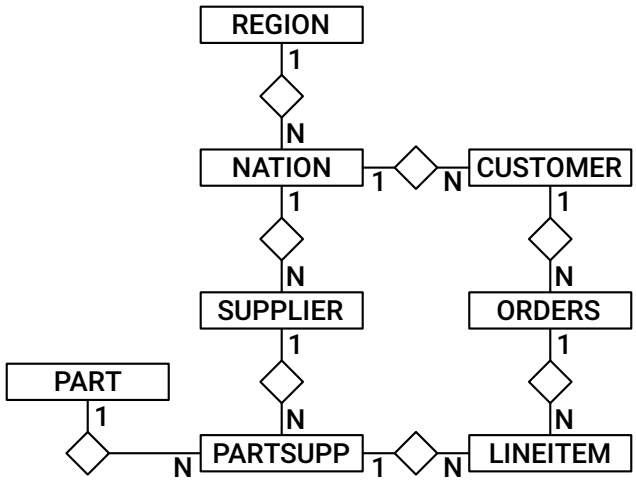


Figure 2.2: Schema of TPC-H

```
1  select 100.00 * sum(case
2        when part.type like 'PROMO%'
3        then lineitem.extendedprice*(1-lineitem.discount)
4        else 0
5      end) / sum(lineitem.extendedprice * (1-lineitem.discount))
6      as promo_revenue
7  from lineitem, part
8  where
9    lineitem.partkey = part.partkey
10   and lineitem.shipdate >= date '[DATE]'
11   and lineitem.shipdate < date '[DATE]' + interval '1' month;
```

Listing 2.2: TPC-H query 14, slightly adapted for better undrestanding. `[DATE]` is the first day of a month randomly selected from a random year within [1993 .. 1997], e.g. 1995-09-01.

YCSB

The lack of apples-to-apples performance comparisons for the branch of cloud systems led Cooper, Silberstein, Tam, Ramakrishnan, and Sears [Coo+10] from Yahoo! Research to develop the Yahoo! Cloud Serving Benchmark (YCSB) to enable an understanding of the tradeoffs between these systems and the workloads for which they are suited. They further motivate their work with a hard performance comparability of those systems, as they made different decisions about access optimization, data partitioning and placement, replication, transactional consistency, et cetera. They provide an extensible benchmarking framework to assist in the evaluation of different cloud systems and in addition supply a core set of workloads for it.

The developed tool to execute the YCSB benchmark is a Java program called the *YCSB Client*, which generates the data to be loaded to the database and the operations which make up the workload. The architecture of the client is shown in figure 2.3 on the next page. The tool works in the following way, as described by its authors in [Coo+10]:

> *"The basic operation is that the workload executor drives multiple client threads. Each thread executes a sequential series of operations by making calls to the database interface layer, both to load the database (the load phase) and to execute the workload (the transaction phase). The threads throttle the rate at which they generate requests, so that we may directly control the offered load against the database. The threads also measure the latency and achieved throughput of their operations, and report these measurements to the statistics module. At the end of the experiment, the statistics module aggregates the measurements and reports average, 95th and 99th percentile latencies, and either a histogram or time series of the latencies."*

The operation of the client can be defined by a series of properties (name/value pairs), divided into *workload properties*, which define the workload, independent of a given

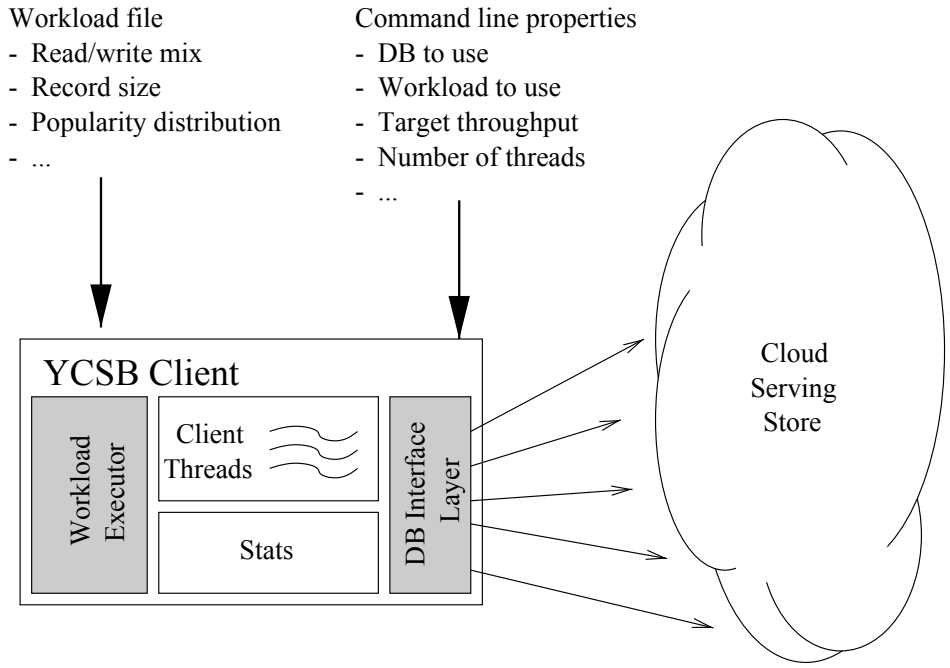


Figure 2.3: YCSB client architecture, taken from [Coo+10, Figure 2].

database (e.g. read/write mix of the database, distributions to use), and *runtime properties*, which are properties specific to a given experiment (e.g. number of client threads, database layer to use).

A primary goal in the development of YCSB was extensibility, which is supported by replacing the components shown as shaded boxes in figure 2.3. This especially means that the tool allows the definition of own workloads, as well as the support of new databases by writing a class implementing `read`, `insert`, `update`, `delete` and `scan` operations for the new database.

The tool tests the performance of all databases using a very simple schema with a single table consisting of ten string attributes (each with variable length), one of which is the row key used to identify an item. As some cloud systems do not support item selection using multiple attributes, all created queries only use the row key to identify items.

2.3 Random Number Generation

Random numbers are used in many fields, such as the creation of random data sets, data encryption, simulating and modeling complex phenomena, or for games and gambling. Drawn random numbers must both be statistically independent and be a subject to a specific distribution. Random sequences are distinguished between *truly* random and *pseudo-random* sequences. True random numbers are extracted from physical phenomena expected to be controlled by randomness, like coin flipping, dicing, atmospheric noise or radioactive decay, while pseudo-random numbers are generated using com-

pletely deterministic algorithms. For computational purposes, the use of truly random numbers has been practically abandoned for various reasons [BFS83; LEc90].

Random Number Generators (RNGs) are procedures that produce an infinite stream $U_1, U_2, U_3, ... \overset{i.i.d.}{\sim} Dist$ of random variables that are identically distributed (i.i.d.) to some probability distribution [KTB13]. Because this concept of an infinite i.i.d. stream of random variables may not be possible to implement, it is tried to produce a sequence with statistical properties indistinguishable in feasible time from those of a truly random sequence [LEc94; KTB13]. Such an RNG is called *Pseudo-Random Number Generator (PRNG)*, which will be described in section 2.3.1. However, even many scientific publications lack the accurate distinction between these terms, often using the term RNGs when referring to PRNGs.

To use a random number in the production of synthetic data (e.g. for DBMS benchmarking), it is either used directly or mapped to a different domain via some predefined function. For example, to map a random number x from the domain of real values $[0, 1]$ to the domain of real values $[y, z]$, the function $f(x) = y + x \cdot (z - y)$ could be applied. As a second example, a random string could be constructed iteratively by first determining a random integer number n and then picking the n^{th} character from a set of characters. As a final example, realistic data, e.g. for name generation, can be based on a dictionary lookup by picking an entry based on a generated random number [RP11].

In order to generate data for large data sets, it is desired to generate this data in parallel. Therefore, after showing how non-parallelized generators work in the following sections, it is shown how to parallelize them in section 2.3.3 on page 17.

2.3.1 Pseudo-random Generators

A *PRNG* is defined by [LEc94] as a tuple (S, s_0, f, U, g), where

- S is a finite set of *states*,
- $s_0 \in S$ is the *initial State* (called the *seed*),
- $f : S \to S$ is the *transition function*,
- U is a finite set of *output* symbols, and
- $g : S \to U$ is the *output function*.

The PRNG then uses the following deterministic algorithm [LEc94]: Start from the initial state s_0 and let $u_0 := g(s_0)$. Then, for $i := 1, 2, ...$ let $s_i := f(s_{i-1})$ and $u_i = g(s_i)$. This produces a sequence $U_1, U_2, U_3, ...$ of *pseudo-random numbers*. As [KTB13; LEc94] state, starting from a certain seed, the sequence of states (and hence of random numbers) must repeat itself, because the state space is finite. The smallest number of steps taken before entering a previously visited state is called the *period length* of the generator.

There are desirable properties that researchers (i.e. [Hel98b; KTB13; LEc94]) agree to be essential properties of a good PRNG: good statistical properties, theoretical support, reproducibility, long period, speed and efficiency, and splitting facilities.

Different approaches for generating pseudo-random sequences have been engineered in the past decades, leading to a plethora of methods that differ largely in quality of the wanted properties. This led to the fact that several of the standard generators shipped by popular programming languages can be "painfully poor" [LEc01] or "badly flawed" [Pre+07]. Hence, the user of PRNGs has to be aware of the fact that his choice might have undesired effects on the results and correctness of his algorithm.

Example: Wichmann-Hill

To demonstrate how a specific PRNG works, the process shall be illustrated with a generator commonly known as *Wichmann-Hill*. The PRNG originally called *AS 183* by its authors in [WH82] has been widely used since its presentation, but has been shown to be inadequate by today's standards and hence got mostly replaced by others (e.g. the *Mersenne Twister* [MN98]). It is one of the earliest examples of *combined generators*. Those PRNGs use the output of several PRNGs, which may be of poor quality, to combine their output, resulting in a PRNG of superior quality [KTB13].

The approach of Wichmann and Hill for example combines three *linear congruential generators*[7] to obtain the values for X_t, Y_t and Z_t. In short, a linear congruential generator is a PRNG with state $S_t = X_t \in \{0, ..., m-1\}$ for some strictly positive integer m called the modulus, and state transitions $X_t = (aX_{t-1}+c)\ mod\ m,\ t = 1, 2, ...$, where the multiplier a and the increment c are integers. Applying the modulo-m operator means that $aX_{t-1} + c$ is divided by m, and the remainder is taken as the value for X_t [KTB13].

The Wichmann-Hill PRNG uses an algorithm easy to explain, where first three random integers are computed and the results are then combined into a single random number i.i.d. U[0,1]:

$$
\begin{aligned}
X_t &= (171 \cdot X_{t-1} \bmod 30269) \\
Y_t &= (172 \cdot Y_{t-1} \bmod 30307) \\
Z_t &= (170 \cdot Z_{t-1} \bmod 30323) \\
U_t &= (\frac{X_t}{30269} + \frac{Y_t}{30307} + \frac{Z_t}{30323}) \bmod 1.
\end{aligned}
$$

[7] There are other types of generators as well, but explaining all of them would go far beyond the scope of this short overview. Hence, the interested reader is referred to [KTB13] for exhaustive explanations.

If this algorithm was initialized with the values $X_{t-1} = 1$, $Y_{t-1} = 2$ and $Z_{t-1} = 3$, the first produced random value would be computed as follows:

$$\begin{aligned} X_t &= (171 \cdot 1 \bmod 30269) = 171 \\ Y_t &= (172 \cdot 2 \bmod 30307) = 344 \\ Z_t &= (170 \cdot 3 \bmod 30323) = 510 \\ U_t &= (\frac{171}{30269} + \frac{344}{30307} + \frac{510}{30323}) \bmod 1 \approx 0.03381877363047378 \end{aligned}$$

The next random number generation would then use the previously computed values for X_t, Y_t and Z_t as values for X_{t-1}, Y_{t-1} and Z_{t-1}, resulting in the computation

$$\begin{aligned} X_t &= (171 \cdot 171 \bmod 30269) = 29241 \\ Y_t &= (172 \cdot 344 \bmod 30307) = 28861 \\ Z_t &= (170 \cdot 510 \bmod 30323) = 26054 \\ U_t &= (\frac{29241}{30269} + \frac{28861}{30307} + \frac{26054}{30323}) \bmod 1 \approx 0.7775418875596665 \end{aligned}$$

and so forth for the next random numbers. This algorithm results in a period length of $(30269 - 1) \cdot (30307 - 1) \cdot (30323 - 1) \div 4 \approx 2^{42.66}$, which is atomically short when compared to the Mersenne Twister's period length of $2^{19937} - 1$ [MN98].

One can easily see the PRNG's state transition, and thus how previously generated values can be re-generated. Since PRNGs use information from their last state to generate a new value, all that has to be done is to set the variables used to values of the desired state.

2.3.2 Probability Distributions

The application areas of random numbers can demand for variables underlying a specific probability distribution (e.g. normal, exponential, Bernoulli, Poisson, et cetera). Methods designed to generate those random variables assume in turn the availability of a source of continuous random variables distributed uniformly over the real interval [0,1] (i.i.d. U[0, 1], for short)[LEc94]. A generator producing such variables is called *uniform random generator*, and is at the heart of the random number generation processes. An example showing how such a generator might work was given in section 2.3.1 on the facing page. Following the algorithms from [KTB13], uniform random generators are either used directly when computing a variable underlying another distribution, or can be used in the production of needed generators producing variables underlying yet another distribution. This can be seen in the following two examples, taken from [KTB13, chapter 4].

Example: Exponential Distribution

The first example will show how a uniform random generator is used to produce variables underlying an *exponential* distribution. It is a *continuous* distribution, which is

used primarily as a model to answer the question about the duration of random time intervals, e.g. the time until a radioactive particle decays, or for risk management. Its probability density function, which is a function describing the relative likelihood for a random variable to take a given value, is defined as $f(x, \lambda) = \lambda e^{-\lambda x}, x \geqslant 0$, where λ is called the *rate* parameter, describing the expected number of events occurring in a unit interval. Algorithm 2.1 shows how a uniform random generator can be used in the production of a variable underlying an exponential distribution.

Algorithm 2.1. *Exponential Distribution Generator Exp(λ)*

1. *Draw* $U \sim \mathrm{U}(0, 1)$.
2. *Output* $X = -\frac{1}{\lambda}\ ln\ U$.

This simple example shows the direct usage of a uniform random generator when producing variables for a exponential distribution.

Example: Geometric Distribution

The second example shows a random number generator for a *geometric* distribution, which is a *discrete* distribution again. It can be produced relying on the exponential generator described in the previous example, and hence relies indirectly on the uniform random generator. The probability density function for the geometric distribution is given by $f(x; p) = (1 - p)^{x-1}p,\ x = 1, 2, 3, \ldots$, where $0 \leq p \leq 1$. It is used to describe the time of first success in an infinite sequence of independent Bernoulli trials with success probability p. The used algorithm is described in algorithm 2.2.

Algorithm 2.2. *Geometric Distribution Generator Geom(p) (I)*

1. *Generate* $Y \sim \mathrm{Exp}(-ln(1 - p))$.
2. *Output* $X = \lceil Y \rceil$.

This could also be written using a uniform random generator directly, giving algorithm 2.3.

Algorithm 2.3. *Geometric Distribution Generator Geom(p) (II)*

1. *Generate* $U \sim \mathrm{U}(0, 1)$.
2. *Output* $X = \lceil \frac{ln(U)}{ln(1-p)} \rceil$.

2.3.3 Parallel PRNGs

The exponential growth of the number of transistors on a chip predicted by Moore's law has recently not delivered increasing clock speeds, but more parallelism. But because they are designed as sequentially dependent state transformations, most PRNGs scale poorly to massively parallel high-performance computation [Sal+11]. To solve this problem, *parallel PRNG* have been proposed, most of which parallelize a sequential generator by distributing the elements of the sequence of generated pseudo-random numbers among different processors via three basic ways [Cod97; Sal+11]:

- **Independent sequences:** choosing the initial seed for each processor in such a way as to produce long period independent subsequences on each processor.
- **Leapfrog:** partitioning the sequence in turn among the processors in the same way a deck of card is dealt to card players.
- **Sequence splitting:** partitioning the sequence by splitting it into contiguous non-overlapping sections.

All these methods have inherent problems [Cod97; Sal+11]. The approach with independent sequences should only be used with families of PRNGs having large, easily enumerable sets of "known good" parameters, which have proven elusive. The last two techniques require a PRNG with large period and a mechanism for partitioning the underlying sequence of the used generator, which only a few allow. Also, the method used to distribute the sequence may amplify any small correlations existing in the used PRNG or even introduce inter-processor correlations [DP88; DP90; Hel98a]. Given a PRNG with a very long period, the starting points for the parallel sequences could be chosen at random with only a negligible chance of overlap between subsequences because of the large state space. However, even with non-parallel PRNGs failing at the nontrivial task of properly initializing the state of long-period PRNGs can result in significant defects [Mat+07], so this is an even greater risk when using many of them in parallel.

These problems led to some interesting approaches to generate random numbers in parallel, like counter-based PRNGs using block ciphers [Sal+11], stream ciphers [NA11], or elliptic curves over a prime field [NA14]. However, L'Ecuyer, Oreshkin, and Simard [LOS14] conclude that in order to construct efficient and reliable parallel PRNGs, there is still work that remains to be done.

3. Concept for Configurable Testbeds

This chapter will explain the basic concepts for the design of a configurable testbed for scalable data management systems. First, general requirements and preliminary considerations will be stated, after which an overview of the conceptual design will be given. The chapter is closed by taking a look at the topic of seed management.

3.1 Requirements

The target of this master's thesis is the development of a prototype that allows stress testing an instance of a NoSQL database, using specific schemata and workloads defined by the user. Therefore, and in accordance with the requirements stated in section 2.2.1 on page 7, a user of this program has to be able to define

- **Database schemata:** To emulate a specific use case it is inevitably important to be able to define the schemata of that scenario.
- **Workloads:** Related sequences of queries, which are expected to occur in a use case that is to be tested, must be definable.
- **Ratios between workloads:** When multiple workloads have been defined, it should be possible to state that e.g. `WORKLOAD A` needs to represent 20% of all executed queries, while `WORKLOAD B` and `WORKLOAD C` each represent 40% of all executed queries.
- **Value domains:** For each schema defined the user should be able to state domains for each defined attribute, i.e. that `COLUMN A` holds values between -5 and 42, or the boolean values in `COLUMN B` will be true with a chance of 57%.

- **Termination conditions:** Since the test should be completed when the maximum performance of the database was ascertained, the user must be able to determine when that goal has been achieved, as a database under heavy load might still respond to queries though its performance decreases.

Whenever possible the program should derive needed metadata automatically, i.e. data types, table and column names, to spare the user from non-essential definitions.

The output of the prototype aims to enable the user to rate the performance of his database by providing information necessary for that task. Unfortunately, the tight time frame does not allow providing detailed in-depth information for each query within each workload. Instead, just aggregated values of the overall performance will be provided. However, it will also be tried to assure that acquisition of detailed information is not impeded.

It should be noted that the resulting program is not conceived as a benchmarking tool to compare the performance of DBMSs on different computer systems. However, when run with the same configuration on different machines, it could be used as such, though the actual target is the superset of that: testing any configuration on any system.

3.1.1 Preliminary Considerations

In this subsection requirements of the program's different key aspects will be thought through. Also, it will be stated which desirable properties can not be implemented due to the tight timeframe of this thesis.

Scalability and Extensibility

Since the goal of this program should be to test the performance of scalable NoSQL databases, the program itself should be able to scale, too. It should, when provided with the needed computing resources, automatically increase load to adapt to the capabilities of the tested database and be able to explore its performance limits. Hence, the implementation has to be designed to support parallelization and allow easy load scaling by just adding more parallel computing power. But again, the tight timeframe does not allow for complete support of all kinds of parallelization, so that only support for parallel processing on a single machine will be implemented.

Because of the variety of NoSQL databases (described in section 2.1 on page 3) and their ongoing development, it is not possible to implement a tool supporting all approaches in the scope of this master's thesis. Hence, it is only implemented supporting Apache Cassandra. The reasons that led to choose C* are given in section 4.1.2 on page 37.

The factors mentioned in the previous two paragraphs lead to the decision that a modular design has to be used. This allows for fast and easy replacement of relevant parts in case single modules do not provide required functions, e.g. support of previously unsupported databases or increased parallelization.

Data Generation

At the core of this program, the generation of random data has to allow testing arbitrary schemata and equally arbitrary queries. As the aim is to utilize the full capacity of the tested databases, random data generation must be equally efficient, fast, and scalable. For the latter reasons the processes need to be parallelizable, while simultaneously synchronization of shared objects has to be assured. Since queries that read, update or delete entries often require scope-constraining data, it must also be possible to regenerate previously created random data.

For the solution of both problems the nature of PRNGs can be utilized. The definition from section 2.3.1 on page 13 shows that each state determines the output of a PRNG. This property easily enables parallelization by ensuring that each state is only used once, namely to produce a distinct entry, hence using one of the approaches presented in section 2.3.3 on page 17. However, the implemented approach differs from that and will be described in section 3.3 on page 28.

The one-state-per-entry property also enables regeneration of previously produced data and allows to guarantee that read-, update- and delete-queries can be filled with previously produced content. A prerequisite, however, is the knowledge of what data has previously been produced, so this information needs to be stored and synchronized between running processes accessing this data. However, the synchronization effort has to be kept minimal in order to achieve maximal scalability.

Even very different approaches on *how* to store data still store the same *types* of data. This results in the fact that the implemented random generator must be able to produce all (major) data types, including composite data types like set, list, and map, which are frequently used in NoSQL databases. Additionally, the program must support the generation of additional data types used by the database under test. Again, this should be implemented using modules for the basic data types and database-specific types respectively.

Although in reality data between and even within tables can have complex relations, it will not be tried to model this property in the generation process. This problem alone would pose enough problems and pitfalls for a separate thesis, hence it can not be considered in this thesis.

Configuration

To support repeated execution with the same chosen settings, a file-based storing of runtime configurations should be used. This also allows for fast testcase adjustments as well as easy exchange of settings, even between users and different hardware configurations. Also, previously tested approaches can be archived and reused later without taking notes about the used settings manually.

If all needed modules are implemented, the program execution should not call for user action other than providing a reference to a configuration file with all needed data.

Hence, in accordance with the requirements stated in section 3.1 on page 19 this file must provide at least the following data in a form that is processable by the used modules:

- **Schemata:** A set of table definitions, identifiable by name, each possibly with metadata for each column (e.g. value domains) supported by the used Random Data Generator (RDG).
- **Workloads:** A list of workloads, identifiable by name, each with a probability of execution and a sequence of queries belonging to that workload.
- **Connection establishment data:** Arbitrary data needed by the module responsible for connection establishment, e.g. network address.
- **Termination conditions:** The prerequisites that have to be fulfilled to terminate the program, e.g. maximum latency.

As those data can not always have the same format if all types of databases should be supported, the processing of the configuration file must be implemented modularly, too. Furthermore, it should be attempted to keep the needed configuration data minimal to provide user-friendliness, with sane default settings chosen where possible to keep the need for configuration slight.

As the configuration file is the intended interface for the user to manipulate the program, it calls for comfort. For this reason the configuration of the program should be done using an approach that is easy to write and read for users. Also, as the user will be requested to specify different data structures to state his preferences, it should represent this approach and offer a natural way to describe these data structures.

3.1.2 Other Considerations

Although it will be attempted to implement the tool in a way that it reaches maximal performance, not each of its aspects can be exhaustively optimized to reach this goal because of the complexity of that task alone. Thus, determining performance-critical parts and thorough optimization of those parts has to be done in future work, as it unfortunately can not be accomplished within the narrow time frame of this thesis.

It should also be tried to thoroughly document the source code. This supports further adaptions, code extensions providing new features, error corrections, and the implementation of modules needed to support other types of NoSQL databases (and maybe other types as well).

3.2 Overall Design

In this section there will first be an overview on the proposed architecture and workflow for a configurable testbed for scalable data management systems. Afterwards, a more detailed look at the testing process will be given, highlighting the specific part of parallel data generation, querying and execution data acquirement. Finally, a proposed configuration file format will be explained.

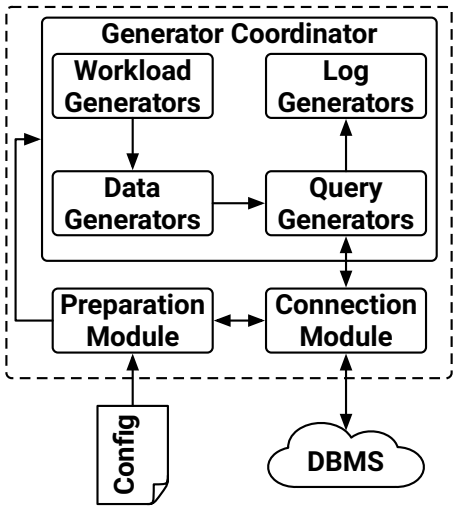


Figure 3.1: High-level program architecture

3.2.1 Overview on Architecture and Workflow

Following the requirements stated in the preceding sections of this chapter, the high-level architecture shown in figure 3.1 has been designed. It depicts the major components taking part in the execution of the program, where the program's own parts are framed by a dashed line.

An external configuration file is used as input for a preparation module, which reads and processes this file. This preparation module serves information to a coordinator, which coordinates a set of generators for workloads, data, queries and logs. All of these generators work in parallel and communicate with each other sequentially via one-way channels between subsequent generators. Both the preparation module and the query modules have a two-way connection to share information with a connection module. This module in turn has a two-way connection for communication with an external DBMS.

The chosen workflow can also be followed in that figure: at first, a file with runtime configurations is read and processed by a preparation module, including the establishment of the environmental variables (e.g. metadata for the data generation process) and the preparation of the testing environment (e.g. schemata creation). To create the needed schemata and possibly obtain more metadata about it, the preparation module communicates with the DBMS via a connection module. Afterwards, the determined data is passed on to a generator coordinator, which uses this information in the construction and supervision of generators. After their initialization, these generators each take part in a particular step of the testing phase (see section 3.2.2 on the following page). This approach has been chosen to allow for easy expansions and adaptions.

The demanded ability to parallelize this process (see section 3.1 on page 19) can easily be obtained by using separate processes for each generator, for parts of the generator chain (hence two or more consecutive generators of the generator series), or for a complete

sequence of all four generator types. However, possibly shared objects often have some consistency restrictions and thus might require locking. Thus, communication among consecutive generators as well as between processes and the coordinator has to be swift. For reasons of simplicity the approach chosen in this thesis is to use separate processes for each generator. Also, because of the just stated considerations it was tried to minimize the need for inter-process communication and the amount of shared objects.

3.2.2 Testing Process

After the parsing and processing of a given configuration file, the obtained information is passed on to a coordinator, whose purpose is to create and manage all processes and objects needed in the further testing process. As working in parallel to achieve scalability is a requirement (see section 3.1.1 on page 20), it generates (possibly multiple) processes for each generator class. Furthermore, the coordinator also has to assign communication channels (e.g. queues or signals) to the created processes and manage the information shared between them (e.g. synchronize information about already created data). It also might have the ability to create more processes later on to further utilize available resources.

In the testing process, data flows through specific instances of each generator class in the way depicted in figure 3.2. This figure is an overview for the general process, leaving out smaller steps and loops to support comprehensibility. Also, the figure does not show that possibly multiple generators of each class might exist and work in parallel, as this is not relevant for the general data flow.

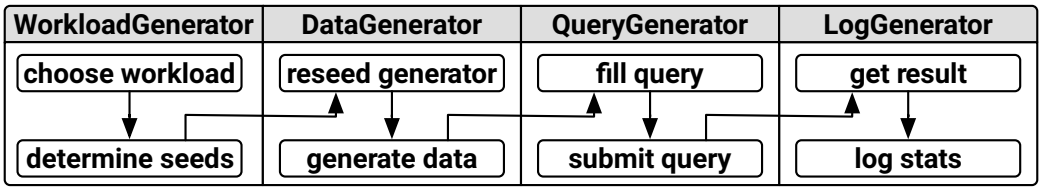


Figure 3.2: Workflow between generators

In the first step, workload generators choose the workload (a sequence of queries) to execute next from the given set of workloads and determine metadata (e.g. the seeds to use, datatypes, or constraints to the data) for the production of the needed data items. This information is passed to the data generators. These produce the just described items with respect to the passed metadata (especially the seed to use), and pass them on to a query generator. Those in turn fill the chosen queries with the produced data and then send the resulting query to the database via the connection module. On receiving an answer from the DBMS, data about the query's execution is handed over to the log generators. They aggregate this data and give it back to the coordinator, where it is presented to the user.

During all these processes the fulfillment of given termination conditions has to be checked. If they are found to be true, the testing process has to be terminated.

If it shall be guaranteed that queries reading, updating, or deleting data only operate on items already present in the database, information about which data items have already been put into the database has to be stored. Furthermore, this information has to be synchronized between processes, and a system for how to regenerate these has to be developed. As a solution to these problems is highly specific to the approach chosen for parallel random data generation, it will not be elaborated further here. However, in section 3.3 on page 28 the approach chosen for the implemented prototype will be described.

3.2.3 Configuration File

In order to run, the program needs a file with runtime configurations provided by the user. Bringing together the demands from section 3.1.1 on page 21 and using the YAML Ain't Markup Language (YAML) format (see section 4.1.3 on page 37 for the reasons that led to that choice), a configuration file template for testing C* instances was created. It is divided into three parts: schemata, workloads, and general configurations. For each of these parts the proposed template and an instantiation of this template using the example from listing 2.1 on page 6 will be shown.

`schemata` part

```
1  schemata:
2    <keyspace_name>:
3      definition: <keyspace_definition>
4      tables:
5        <table_name>:
6          definition: <table_definition>
7          distributions:          # optional
8            <column_name>: <generator_args>
9            <column_name>: <generator_args>
10           ...
11       <table_name>:
12         definition: <table_definition>
13   <keyspace_name>:
14     definition: <keyspace_definition>
15     tables:
16       ...
```

Listing 3.1: Configuration file template, schemata part

The schemata part of a C*-specific configuration file (see listing 3.1 for the template and listing 3.2 on the next page for an example) consists of definitions for each keyspace[1], which in turn consist of table definitions. For each `<keyspace_name>` (e.g. 'events' in the instantiated example) and each `<table_name>` (e.g. 'timeline' in the instantiated example) within the keyspace's `tables` section there has to be a definition in form of

[1] a top-level namespace allowing configurations (e.g. replica placement strategy class, replication factor) for a set of tables

```
schemata:
  events:                       # multiline strings in YAML that should
    definition: |               # keep line breaks are denoted with a |
                  CREATE KEYSPACE IF NOT EXISTS events
                  WITH REPLICATION = {'class':'SimpleStrategy',
                                      'replication_factor':3}
    tables:
      timeline:
        definition: |
                      CREATE TABLE events.timeline (
                        day TEXT,
                        hour INT,
                        min INT,
                        sec INT,
                        value TEXT,
                        PRIMARY KEY (day, hour, min, sec))
        distributions:          # time values have certain restrictions
          day: {size: 11}
          hour: {low: 0, high: 23}
          min: &id001 {low: 0, high: 59} # using an anchor
          sec: *id001                    # referencing the anchor
```

Listing 3.2: Configuration file example, schemata part

a valid CQL query. For each table the properties of data within each column can be specified in the `distributions` section, using arguments supported by the used RDG. There can possibly be multiple keyspace specifications, each possibly with multiple table definitions, each possibly having multiple distribution specifications.

`workloads` part

```
workloads:
  <workload_name>:
    queries:
      - query: <query>
        chance: <value>         # only needed if query type is 'insert'
      - query: <query>
      ...
    ratio: <value>
  <workload_name>:
    queries:
      ...
```

Listing 3.3: Configuration file template, workloads part

In the `workloads` part (see listing 3.3 for the template and listing 3.4 on the next page for an example) each workload is identified by a `<workload_name>` (e.g. 'insert_values' and 'query_timeframes' in the instantiated example). Each workload has a sequence of `queries` and a `ratio` value that determines the probability to choose this workload

```
workloads:
  insert_data:
    queries:
      - query: |
                INSERT INTO events.timeline (day, hour, min, sec, value)
                                VALUES (?, ?, ?, ?, ?)
        chance: .001            # many inserts per day, as the chance to
    ratio: 1300                 # create a new partition is very low
  query_timeframes:
    queries:
      - query: |
                  SELECT * FROM events.timeline WHERE day= ?
                            AND (hour, min) >= (9, 00)
                            AND (hour, min, sec) <= (11, 59, 59)
      - query: |
                  SELECT * FROM events.timeline WHERE day= ?
                            AND (hour, min) >= (15, 00)
                            AND (hour, min, sec) <= (17, 59, 59)
    ratio: 10
```

Listing 3.4: Configuration file example, workloads part

for execution. For each query within a workload there has to be a definition in form of a `query`, which has to be in valid CQL for the example. Queries that insert data into the database also need to specify a `chance` value that determines whether the new item will be inserted into an already existing partition or create a new partition. In the example the items are partitioned by the `day` attribute, hence the `chance` parameter determines the probability of inserting a new day instead of adding data to an already existing day. The values for `chance` should be defined in the range $[0, 1]$, while the `ratio` values can be arbitrary numbers. This approach has been chosen to spare the user from error-prone manual calculations of `chance` values for each workload. It also enables easily changing the tested scenario by adding and/or removing workloads without the need to recalculate chances manually, as these calculations are done by the program. In contrast, the `chance` value for a specific query only determines whether a new partition will be created, a decision that is not influenced by other queries and hence not enabling automatic calculations.

`config` part

In the `config` part (see listing 3.5 on the following page for the template and listing 3.6 on the next page for an example) the value for the `type` of the `database` (which is `Cassandra` in the example) determines which modules will be used to process the configuration file, and hence the other modules to use as well. Also, there is an optional possibility to specify `connection arguments` (e.g. the protocol version to use in the example configuration). Furthermore, this part is where the `termination conditions` are specified. This section was not yet developed thoroughly, as there are only very few options available to automatically terminate the program's execution.

```
config:
  database:
      type: <database_type>
      connection arguments: <arguments>       #optional
  termination conditions:
    latency:
      max: 1000 # Value in ms
      consecutive: 5
    queries:
      max: 10000
      consecutive: 5
```

Listing 3.5: Configuration file template, configuration part

```
config:
  database:
      type: Cassandra
      connection arguments: {protocol_version: 3}
  termination conditions:
    latency:
      max: 1000               # value in ms
      consecutive: 5
    queries:
      max: 10000
      consecutive: 5
```

Listing 3.6: Configuration file example, configuration part

Right now the only choices are specifying a maximum value for the average `latency` and number of `queries` per second, each with the possibility to state the number of `consecutive` times this values has to be exceeded. These possibilities should be extended (e.g. percentiles for latencies) and made optional to allow for more kinds and combinations of termination conditions.

3.3 Seed Management

To enable regeneration of previously created data items some form of seed management is needed, as information about the usage of seeds has to be stored. As parallelization is a key element to scalability, this information also has to be synchronized between processes generating data items. Also, the amount of seeds used is potentially huge. These properties call for a memory efficient storage with fast access to specific seed information, including

- what kind of item has been produced using this seed (e.g. a key item, sub-key item, or an attribute), called the *level* of an item from now on,
- whether the item produced with this seed was updated and what seed was used to do so,

- whether the item produced with this seed was deleted.

As possibly a huge number of items need to be created, the memory consumption for a single produced item is of great interest. In terms of memory required, the deletion information is two-valued and thus needs only one bit of memory. If n different values for the level of an item are possible, this information needs at least $\lceil log_2(n) \rceil$ bits to be stored. Knowing whether an item has been updated again cost only 1 bit.

Often all of this information is needed when trying to recreate an item.Thus, it can be grouped together into a seed bitmap, where every created item is represented by $1 + 1 + \lceil log_2(n) \rceil$ sequential bits. Storing this information in separate structures would be possible, too, but would often require locking all of these objects when trying to acquire information. This would be much more computationally expensive and also introduce possible deadlocks.

If an item has been updated, the information about which seed was used to update which other seed consumes the most bits per item, and is heavily dependent on the maximum seed size. It calls for an associative array to map a given seed to another, which should be as memory efficient and as fast as possible.

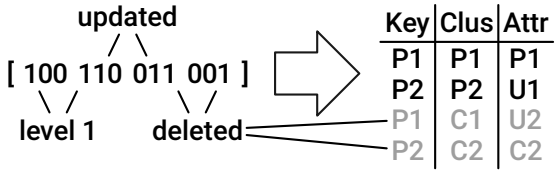


Figure 3.3: Bitmap with four entries and corresponding table

For these reasons the prototype uses two separate data structures: a bitmap for basic information about an item (as shown in figure 3.3), and an associative array to store the information about seeds used to update a previously generated item. Furthermore, these structures are maintained on a per-table basis to keep the need for access synchronizations small. A proposed associative array is the Judy array[2], as it has excellent properties regarding memory usage as well as data access and manipulation.

3.3.1 Ascertaining Seed Information

When determining information about the seeds to use for data generation, two major cases have to be distinguished: the creation of a new data item, or obtaining (and possibly manipulation of) an already created item. In the following an approach handling both cases will be described, making use of the fact that in this approach all seeds are used sequentially, i.e. seed x is used first, then seed $x+1$, and so on. It should be noted that this approach enables parallelization without shared states using the *sequence splitting* method described in section 2.3.3 on page 17. To keep the explanations simple, it

[2] http://judy.sourceforge.net, accessed 2014-11-29

is also assumed that only two different item levels exist (called *partition* and *cluster*). Also, it is assumed that update operations always update all possible attributes (and hence no parts of the key), resulting in four possible configurations for seeds used in the production of a data item, which are shown in table 3.1. It shows that items of level *partition* can only consists of data produced with either just the *partition seed*, or, if they have been updated by some later workload, can have non-key attributes that used the *update seed* for data generation. Similarly, an item of level *cluster* only used the *partition seed* and the *partition seed* when created, and, if it was updated later on, used *update seed* for the generation of all non-key attributes.

partition key	cluster key	other attributes
partition seed	partition seed	partition seed
partition seed	partition seed	update seed
partition seed	cluster seed	cluster seed
partition seed	cluster seed	update seed

Table 3.1: Possible seed configurations used to produce data for different attributes

Furthermore, slightly adapted code examples taken from the developed prototype written in Python (see section 4.1.1 on page 35 for the reasons leading to that choice) are shown in listing 3.7 on the facing page and listing 3.8 on page 32 to illustrate the descriptions.

3.3.2 Creating a new item

When having to create a new item, the first thing to do is to determine what seed will be used. This is easily computed by looking at the length of the seed bitmap for the table the item will be written into, and dividing that value by the length of each entry in the bitmap. As composite keys consisting of a *partition* and a *cluster* part have to be determined, it has to be found out if the new item will be a sub-item of an existing partition item or not. Hence, the random generator is seeded with the just computed seed, and a random value is drawn and compared to the `chance` value (ranging from 0 to 1) of the given query. If this results in the creation of an item at the *partition* level, the previously determined seed will be used to produce all attributes of this item. Otherwise, the determined seed will be used as *cluster seed*, and a seed that has been used as *partition seed* has to be determined.

In the proposed approach this is done by iteratively drawing a random number between 0 and the current seed value (no created item can have used another seed), and looking up the level of the item produced with this seed in the bitmap. If it was on the *partition* level, the wanted seed was found. If not, the next random number is drawn and the level of the item produced with that seed is determined, and so on.

It should be noted that this iteration takes $\frac{1}{chance}$ ($chance \in [0, 1]$) steps on average, hence for small `chance` values it can take many iterations to determine an already

```
# Each item in the bitmap has three bits:
# - the first is set if the seed produced an item on partition level
# - the second is set if the item was update_seed
# - the third is set if the item was deleted
# First, determine the seed to use.
cluster_seed = bitmap.length()/3
partition_seed = cluster_seed
# Seed the generator to produce regeneratable results.
generator.seed(partition_seed)
# Determine if this seed will generate
# a new cluster for an old partition.
new_cluster = query['chance'] <= generator.random()
if new_cluster and partition_seed > 0:
    # An old seed that created a completely new item is needed,
    # so randomly iterate over old keys until one is found.
    while True:
        partition_seed = self.generator.randrange(0, cluster_seed)
        if partition_seed == 0 or bitmap[partition_seed*3]:
            break
# Append information about the created item to the bitmap.
bitmap.extend((not(new_cluster) or partition_seed == cluster_seed, 0, 0))
```

Listing 3.7: Python code determining seeds for the creation of a new item

created partition. Nevertheless, this approach has been chosen as the following thought-off alternatives are inferior.

Storing a list of all seed used for the creation of a new partition would allow very fast determination of such a seed, but would cost approximately $\frac{1}{chance} \cdot itemcount \cdot seedsize$ bits of memory. Considering the possibly large number of items to generate and seeds needing 64 bits each, this would quickly result in vast amounts of used memory.

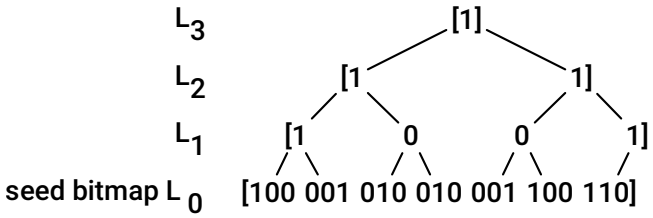


Figure 3.4: Tree-like bitmap structure for seven created items with two possible levels

As an interesting alternative, a tree-like bitmap structure (as shown in figure 3.4) could be used. It would consist of $\lceil log_2(n) \rceil$ bitmaps, where n is the number of created items (both partition and cluster items). An entry k in bitmap l would be true, if at least one of the entries $(2k, 2k-1)$ in bitmap $l-1$ would be true. A special case would be bitmap 0, which would be the seed bitmap and would require additional logic, as each entry consists of more than one bit. If now a partition seed has to be determined, the tree would be traversed from item $l[k]$ by choosing indifferently between $l-1[2k]$ and $l-1[2k-1]$ if both are true, or choosing the one that is true. On the assumption

that drawing a random number is as expensive as traversing one level in this 'tree' of bitmaps, this approach would be superior if $\frac{1}{chance} < \lceil log_2(n) \rceil$. Nevertheless, this approach also has some disadvantages:

- a higher memory consumption (at least $n - 1$ additional bits),
- possibly worse if $\frac{1}{chance} > \lceil log_2(n) \rceil$,
- difficult (or maybe even impossible) support for seeds underlying distributions other than uniform distribution.

Assuming high numbers of data items (and thus a high value for n), this approach quickly gets inferior to the iterative random approach.

3.3.3 Obtaining a previously created item

```
1  # Determine the highest used seed.
2  ks_length = bitmap.length()/3
3  was_deleted = True
4  # Search for a seed that has not been deleted.
5  while was_deleted:
6      cluster_seed = generator.randrange(0, ks_length)
7      is_primary, was_updated, was_deleted =\
8          tuple(bitmap[cluster_seed*3:cluster_seed*3+3])
9
10 partition_seed = cluster_seed
11 # If this seed did not produce a new item on partition level
12 # the partition key it did produce a new cluster for is needed
13 if not is_primary:
14     generator.seed(partition_seed)
15     # When generating a new cluster it is first tested if
16     # a new cluster will be generated by using a random
17     # number, so we need to advance the generator one step.
18     generator.random()
19     # Now it can be searched for the seed that was used.
20     while True:
21         partition_seed = generator.randrange(0, cluster_seed)
22         is_primary, was_updated, was_deleted =\
23             tuple(bitmap[cluster_seed*3:cluster_seed*3+3])
24         if (not was_deleted) and\
25                 (is_primary or partition_seed == 0):
26             break
27 update_seed = cluster_seed
28 # If the item has been updated get the seed used for that.
29 if was_updated:
30     update_seed = update_dict[cluster_seed]
```

Listing 3.8: Python code determining seeds for reading, updating and deleting an item

Reading, updating, and deleting a previously generated item from the database requires information about the seeds used to create and possibly update a specific item. As shown in table 3.1 on page 30, this might demand to determine three different seeds: a *partition seed*, a *cluster seed*, and an *update seed*. The first two can be found within the seed bitmap. Therefore, the same procedure used to generate items will be used.

First, the number of seeds is determined by looking at the length of the bitmap and dividing that number by the length of each entry. Next, a random number between 0 and the number of used seeds is drawn. Afterwards, the information of that seed is looked up in the bitmap. If the item created with that seed has been deleted, a new item will be computed by drawing a random number again until an undeleted item was found.

As soon as a non-deleted item has been found, the level of the chosen item has to be checked. If it is a *partition* item, all key seeds have been determined. If it is a *cluster* item, only the *cluster seed* has been determined yet and is used to seed the RNG as it was done when creating that item. Following the same steps used in the item's creation, the *partition seed* for that item is obtained by iteratively drawing a random number between 0 and *cluster seed* and checking the level of the item created with that seed.

As a last step, it is checked if the item produced using *cluster seed* (which is identical to *partition seed* if the item's level was *partition*) has been updated. Therefore, the associative array of update seeds is checked for the entry *cluster seed*. If there is a value associated with this seed, the *update seed* has been found. Otherwise, the *update seed* defaults to *cluster seed*.

Naturally, if the operation to perform is the update or the deletion of an item, this has to be recorded in the according data structure.

3.3.4 Further remarks

A possible drawback of the chosen approach has to be noted: as update operations might not change every attribute of an item, the information about a seed used to update an item might be useless and possibly even result in wrong items being queried. However, storing the information about which attributes have been updated with which seed is only possible if the set of attributes is fixed. Then, the *level* part of the seed bitmap could indicate which attributes have been changed. To determine all information needed for a given query this includes which of its attributes have been updated and which seed was used to do so. This would require additional data structures and program logic, hence possibly massively slowing down the process. Also, many cloud data management systems do not require a fixed set of attributes, hence the described solution might not work for them. However, many databases used to store huge amounts of data do not allow item selections using arbitrary attributes, but only attributes that are part of a key. Hence, the proposed approach should be sufficient for most cases, especially the ones of interest.

Also, some other approaches for data generation using PRNGs have been proposed. However, the described approaches were chosen, as they fulfill all major requirements and are simple enough to be implemented in a prototype.

4. Implementation

This chapter will give a detailed look at what has been implemented in the prototype and how this was done. For that, at first general choices are going to be explained. Afterwards, the application schema will be described. Lastly, the process of the generation of an item will be followed through its different stages.

From this chapter on names of classes, methods, functions and variables from the actual implementation will be used. The applied convention is to highlight them in `typewriter typeface` (except inside figures), using `CamelCase` for class names, while `lowercase_with_underscores` is used for the names of methods, functions and variables. Variable arguments of methods, functions and data structures are additionally set in *`italics`* to further differentiate them from normal variables.

The developed prototype is also freely available[1] for further studies, extension, and general usage.

4.1 Implementation Choices

In this section decisions made about the programming language, NoSQL database and configuration file format to use will be explained.

4.1.1 Programming Language

Early on in the course of this work it was decided to implement this project with Python (version 2.7). The reasons leading to that choice were

- **Syntax:** Python is a language with a clean and clear syntax and a basic tool set that can be comprehended and utilized within very short time. This enables writing more code with fewer lines and hence more code in less time. Also, its readability makes it easy to adapt or extend the code for own needs.

[1] https://github.com/causa-prima/COLT

- **Comprehensive library:** Python's "batteries included" philosophy led to sophisticated and robust capabilities of its larger packages for many fields, making many tasks almost trivial.[2] Furthermore, the Python Package Index (PyPI)[3] offers more than fifty thousand additionally available packages to further extend the capabilities of the language.

- **Cross-platform:** The platform independence of Python allows using the program in most environments, especially the targeted environments.

- **Popularity:** It is consistently ranked among the ten most popular languages in major indexes (Tiobe Index[4], PYPL Index[5], RedMonk Programming Language Rankings[6]).

It should be noted that Python is rather slow when compared to other languages, while at the same time needing much less code for the same tasks[7]. While the latter is clearly beneficial for coding speed and often also for code comprehensibility, the former clearly is unfavorable when writing programs that demand the highest performance possible. But what first seems like a contradiction - a slow language for a speed demanding task - gets qualified by the following facts:

- **Parallelization:** Because the code is thought to be parallelizable, adding more computing power should ideally let the program's performance scale in the same manner.

- **Compilers:** Other compilers enable Python code to be executed much faster than with the standard compiler, though sometimes calling for slight code adaptions. However, they won't be tested or even used for interoperability reasons.

- **Libraries:** If some used data structures prove to be slow, there are often plenty of others to be chosen from PyPI that might solve the tasks more efficient.

- **Extensions:** If all else fails, Python has the ability to invoke C or C++ functions out of the box. If some code turns out to be performance critical, it could still be implemented in those languages to heavily speed up the algorithms, while still providing the comfortability of Python code for the other parts of the code base.

[2] for details about included packages see https://docs.python.org/2/library, accessed 2014-10-25
[3] https://pypi.python.org, accessed 2014-10-25
[4] http://www.tiobe.com/index.php/content/paperinfo/tpci, accessed 2014-10-25
[5] https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language, accessed 2014-10-19
[6] http://redmonk.com/sogrady/category/programming-languages, accessed 2014-10-25
[7] for detailed data compare http://benchmarksgame.alioth.debian.org, accessed 2014-10-25

4.1.2 NoSQL Database

As stated before, the variety of different NoSQL databases makes it impossible to develop a single tool to support all approaches in the tight timeframe of a master's thesis. Therefore, it was decided to implement the tool just for C*, but with other NoSQL-approaches kept in mind by making use of modularity, i.e. with inheritance from base classes for unavoidably database-specific parts.

The decision for C* was made for a variety of reasons:

- **Mature status:** Though still under active development, the main features and concepts are fixed. Therefore, the development is based on stable ground.
- **Performance:** The capabilities of C* to perform both reads and writes fast, as well as its ability to scale easily, make it a good choice to test the limits of the chosen algorithms.
- **Data model:** The wide column data model allows to easily adapt to the data model of key-values stores as well as document-oriented databases. This is due to the fact that both other models use one single key to identify objects within the database, while C* uses composite keys and thus being able to emulate the other data models.
- **Flexibility:** Its design allows for a broad field of use cases, hence also for a broad field of test cases.
- **APIs:** C*'s interfaces allow working with most major programming languages, including C++, C#, Go, Haskell, Java, Perl, PHP, Python, R, et cetera.
- **Thoroughly documented:** Well-recorded instructions for its use allow for fast induction to C*. The open source nature of itself and most, if not all, of its APIs even allow exploring the source code when in need for further insights of used methods.
- **Well-established:** Since its release, C* attracted a large base of users, hence the application will have a large base of potential users.

4.1.3 Configuration Format

As a result of the preliminary considerations regarding the configuration in section 3.1.1 on page 21, a decision on a file format was made in favor of the mature and widely-used YAML format. As *"a human-friendly, cross language, Unicode based data serialization language"* [8], data types common to most high-level languages, such as sequences (arrays/lists), mappings (hashes/dictionaries) and scalars (strings/numbers), are designed to be easily mappable. Regarding the read- and writabilty,

[8] http://yaml.org/spec, accessed 2014-10-26

> *"YAML achieves a unique cleanness by minimizing the amount of structural characters and allowing the data to show itself in a natural and meaningful way. For example, indentation may be used for structure, colons separate key: value pairs, and dashes are used to create 'bullet' lists."* [9]

It should also be noted that YAML is a superset of the popular and widely-used JSON format. As JSON has a much stricter format, it is not as easily read- and writable for humans, which disqualified it in comparison with YAML.

4.1.4 Data structures

Python's own data structures have been used for most objects. However, as determined in section 3.3 on page 28, the seed management calls for data structures that are highly memory efficient and allow swift access to their items.

As Python itself does not offer bitmaps, it was chosen to use a package available in the PyPI called "bitarray"[10]. It "provides an object type which efficiently represents an array of booleans" and behaves "very much like a list object, in particular slicing (including slice assignment and deletion) is supported". All of its functionality is implemented in C with minimal memory usage, and a single bitarray object can contain up to 2^{63} elements (limited of course by the available memory).

To store the information which seed was used in an update for an item produced by another seed, a Python wrapper for Judy arrays (which are written in C) was chosen. Because of the lack of documentation of the package available in the PyPI, it was decided to use a freely available library called PyJudy[11]. This library allows handling Judy objects much like Python dictionaries and hence was easy to learn and utilize. The memory efficiency of Judy arrays is astounding, yet still offer fast access to (and manipulation of) stored data items. As this thesis does not allow the explanation of how and why this data structure is so efficient, the interested reader must be referred to online resources[12] explaining the used techniques in great detail.

4.2 Application schema

In this section the implemented classes will be presented in the order of their usage within the workflow of the prototype (see section 3.2 for an overview). For each class and its subclasses there is an Unified Modeling Language (UML)-like class diagram and a description of their main methods and attributes. Notice that both diagrams and descriptions might leave out certain minor methods or attributes to focus on the main functions.

[9] http://yaml.org/spec/1.2/spec.html#Introduction, accessed 2014-10-26
[10] https://pypi.python.org/pypi/bitarray/, accessed 2014-11-30
[11] http://www.dalkescientific.com/Python/PyJudy.html, accessed 2014-11-30
[12] http://judy.sourceforge.net/, accessed 2014-11-30

Also, the configuration file parsing is not part of any class, but is done in the start script. All that this script does is loading the configuration file and passing it to a YAML parser and then choosing the right `PreparationInterface` subclass to use for further processing. Hence, once the right preparation class is chosen, it is called with the parsed configuration file as only argument.

4.2.1 Preparation

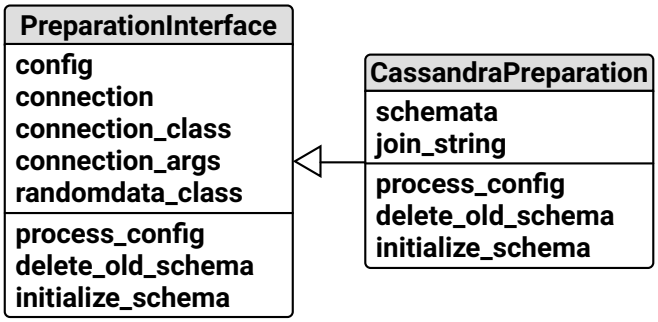


Figure 4.1: Preparation classes

To process the runtime configurations specified, an instantiated subclass of a `PreparationInterface` for the chosen database is needed. When created, this base class will automatically call the child classes methods `delete_old_schema`, `initialize_schema` and `process_config`, which therefore have to be implemented in its subclasses. But before that it instantiates a suitable `ConnectionInterface` subclass, which is needed to delete and recreate the schemata. While the methods `delete_old_schema` and `initialize_schema` should ensure a 'clean state' of the test environment, the `process_config` method has to process the parsed configuration file for the information needed by consecutive processes, putting the results into the corresponding attributes. After the subclass finished its `process_config` method, the base class constructs a `GeneratorCoordinator` and calls its `start` method.

For the `CassandraPreparation` class the `process_config` method determines further metadata about tables and columns, assuming that within the configuration file the keyspace- and tablenames used in their definitions and within workload queries correspond to the names used for the corresponding `<keyspace_name>` and `<table_name>`. Furthermore, it sets the `randomdata_class` attribute to `CassandraTypes`, the `connection_class` attribute to `CassandraConnection` and the `connection_args` attribute to the values specified in the configuration file.

4.2.2 Connection

Although the connection to the dabase is one of the most important parts of the testing process, the `ConnectionInterface` implementation within the program is rather simple. This is due to the fact that it mainly has to provide a mapping from a database

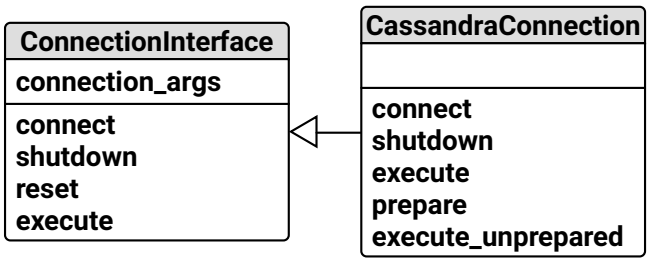


Figure 4.2: Connection classes

specific package implemented by some external provider to some basic functions: `connect`, `shutdown`, `reset` and `execute`. While the function of the first three methods should be obvious, the `execute` method has to do more than one might expect. It must bind given parameters to an also-given query, and then execute the resulting query without blocking the calling process. On the arrival of a result to the query, it has to write information about the execution time into a given queue.

4.2.3 Coordinator

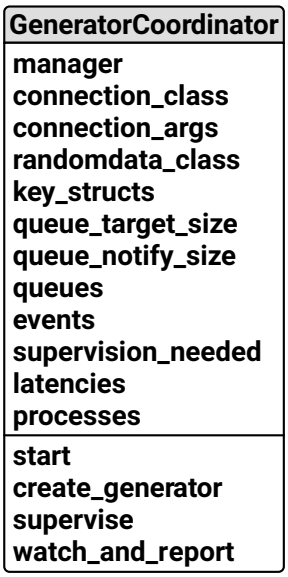


Figure 4.3: Coordinator class

As indicated by its name and already roughly described in section 3.2.1 on page 23, the `GeneratorCoordinator` class manages a set of generators. It possesses many objects, most of which are shared with or passed to managed generators:

- **`manager`:** An object controlling a server process which holds Python objects and allows other processes to manipulate them using proxies. It controls all queues, the `key_structs` attribute and the `latencies` attribute.

- **queues:** Consecutive generators communicate via queues, which are maintained in this attribute.
- **queue_target_size and queue_notify_size:** A limit for the size of the output queue and a minimum for the input queue specifying when it is allowed to notify the coordinator about the need for more input data.
- **events:** Signals enabling child processes to signal the need for supervision, e.g. the need for more input data. All events are also intertwined into the `supervision_needed` attribute to allow for easy polling of all events simultaneously.
- **key_structs:** For each table information about the seeds used for the creation of data for that table are stored within this attribute.
- **connection_class, connection_args and random_class:** The two `class` attributes are references to the connection and random classes needed, while the `connection_args` attribute provides connection argument for the `connection_class`.
- **processes:** A list of created subprocesses.
- **latencies:** Store for log data of execution times.

In contrast to its many attributes, the `GeneratorCoordinator` class has relatively few methods. When called, the `start` method creates processes for each generator from the generator chain and starts them. For the generator creation, it calls the `create_generator` method, which returns a chosen generator, instantiated by passing needed objects and references. Additionally, the `start` method creates a separate process to run the `watch_and_report` method independent from the `GeneratorCoordinator`. This method's purpose is solely to periodically check the `latencies` attribute, check for the fulfillment of termination conditions and report results to the user.

After all processes have been created and started, the `supervise` method is entered, which is an endless loop to periodically check if any class of generators needs more input. To do so, the `supervision_needed` event is checked periodically, and if it is set, it is determined by whom it was set and needed generators are created. As not all systems running this tool are supposed to have an unlimited amount of processors, the maximum number of processes can be chosen, but, due to some Python restrictions, will at least be six processes: one for each generator, one for the `GeneratorCoordinator`, and one for the `watch_and_report` method.

The only way to exit the endless loops of the `GeneratorCoordinator` and its child processes is setting the shutdown signal contained in the `events` attribute, which is treated by all processes with an ordered self-termination.

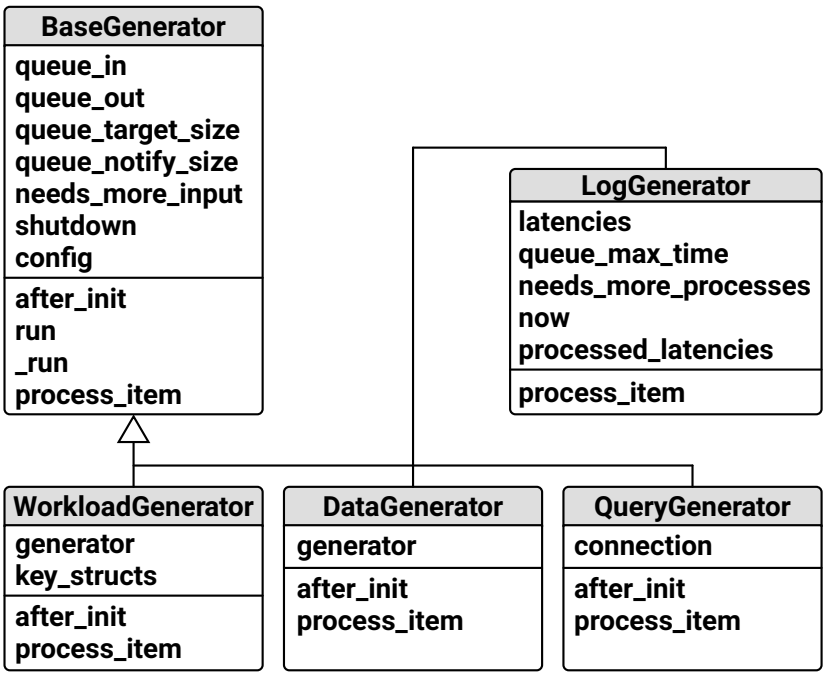


Figure 4.4: Generator classes

4.2.4 Generators

The `WorkloadGenerators`, `DataGenerators`, `QueryGenerators` and `LogGenerators` created by the `GeneratorCoordinator` all inherit from the base class `BaseGenerator` as shown in figure 4.4, because all of them need the same basic attributes:

- **`queue_in` and/or `queue_out`:** an input and/or output queue, where the data to process has to be received from and/or put into
- **`queue_target_size`:** wanted size of the input queue, that, while exceeded, lets the generator stop producing more output
- **`queue_notify_size`:** size of the input queue that has to be undershot to raise a notification
- **`needs_more_input`:** event that will be set when `input_queue` of the generator holds too few items to process, so the `GeneratorCoordinator` can react and create more processes feeding that queue
- **`shutdown`:** event that, when set, causes the generator to terminate itself
- **`config`:** dictionary of the parsed and processed configuration file

The `BaseGenerator` class also implements the methods `run` and `_run` and defines the abstract methods `process_item` and `after_init`. After initialization, the first

method called is the run method, which is first calling after_init and then _run. This approach was chosen to enable the child processes to construct objects in their own memory space by constructing them within their after_init method.

The _run method is an endless loop that will only be left if the shutdown event occurs. Within this loop the number of items in queue_out is checked first. If there are more than queue_target_size items in queue_out, no action is taken until more data is needed. Otherwise, it is checked whether there are less than queue_notify_size items present in queue_in, setting the needs_more_input event if this holds true. These approaches were chosen to save computing power by only generating data items on demand, and enabling automatic adaption to the computing power needed by single generator classes.

After checking for the aforementioned conditions, the process_item method is called, which therefore has to be implemented in all generators inheriting from BaseGenerator. The description of the child classes implementations of this method can be found in section 4.3 on the following page, where each part explains the work done within this method by the named class of generators. Basically, each generator takes a single item from its queue_in, processes it, puts the results into its queue_out, and then returns control to the endless loop of the _run method.

Workloadgenerators and DataGenerators each have a private generator, which is an instance of PythonTypes or one of its subclasses. Additionally, all Workloadgenerators share the key_structs attribute, which hold information about the seeds used for the creation of data for each table. The QueryGenerators each have a private connection, which is an instance of a ConnectionInterface subclass. Lastly, all LogGenerators share their latencies attribute, in which log data of execution times is stored. Each LogGenerator also has needs_more_processes event to signal the need for more LogGenerators, and a queue_max_time value, which determines the maximum time an item of its queue_in is allowed to be within that queue before raising the needs_more_processes signal. As the LogGenerators only synchronize their data once every second, their private now attribute holds the timestamp of the last report. This periodic synchronization, which was chosen to decrease the amount of needed synchronizations, also demands temporarily storing the accumulated data in each LogGenerators private processed_latencies attribute.

4.2.5 Randomdata

The PythonTypes base class used for the production of random data, despite its many methods, is a rather simple one. All methods execpt the lcg_random method are producing data in the type of their name (e.g. pyint produces integers, pystring produces strings). All of these generators accept parameters to determine the domain of their result (e.g. the range of dates, or length of a list). To improve the usability and code clarity, the base class also has a methods_switch-object, which maps the type names to the type functions. This makes a call to any generator a simple methods_switch[*type*](**arguments*) statement, which renders a long, unnatural and

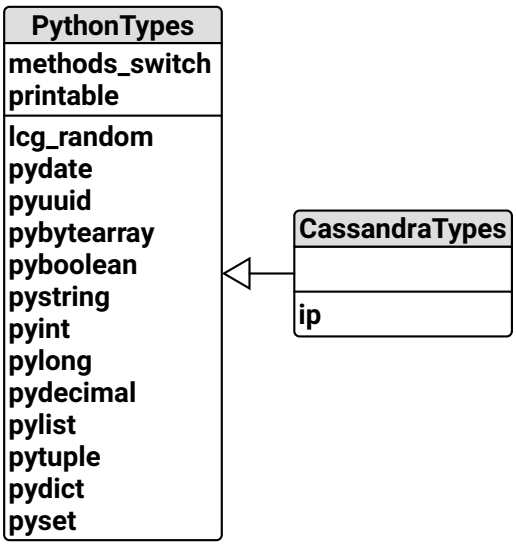


Figure 4.5: Randomdata classes

space-wasting `if-then-elif-else` block unnecessary for methods providing multiple possible output types.

The `PythonTypes` class itself is a child class of Python's built-in `Random` class and hence inherits all of its methods, especially the `seed` method. This method is one of the key methods of the data generation process, as it enables data regeneration through reseeding the generator with a specific seed before the generation of a data item (as required in section 3.1.1 on page 20).

The `CassandraTypes` RDG class inheriting from the `PythonTypes` base class extends it with a method for the C*-specific `inet` type, and also extends the `methods_switch`-object with its own type names, e.g. mapping the C* type `text` to `pystring`. Support for types of other databases can be added in the same simple manner.

4.3 Following a Workload Through the Process

In this section, the implementation of the whole process from the choice of a workload to outputting the momentary performance to the user will be explained. An overview for this process was already given in section 3.2.2 on page 24. To aid the explanations, the configuration file shown in listing 4.1 on the next page (which is the combination of the example pieces from section 3.2.3 on page 25 using the YAML format) will be used to illustrate the actions taken in each step with a specific example.

4.3.1 `WorkloadGenerator`

First step of the process are the `WorkloadGenerators`. Following the chance each workload has to be executed, they choose a workload from the set of possible workloads at random. Then, they iterate over all queries within that workload and determine the RNG seeds needed to create the data items for each query (for details on

```
schemata:
  events:                       # multiline strings in YAML that should
    definition: |               # keep line breaks are denoted with a |
                  CREATE KEYSPACE IF NOT EXISTS events
                  WITH REPLICATION = {'class':'SimpleStrategy',
                                      'replication_factor':3}
    tables:
      timeline:
        definition: |
                      CREATE TABLE events.timeline (
                        day TEXT,
                        hour INT,
                        min INT,
                        sec INT,
                        value TEXT,
                        PRIMARY KEY (day, hour, min, sec))
        distributions:          # time values have certain restrictions
          day: {size: 11}
          hour: {low: 0, high: 23}
          min: &id001 {low: 0, high: 59} # using an anchor
          sec: *id001                    # referencing the anchor

workloads:
  insert_data:
    queries:
      - query: |
                INSERT INTO events.timeline (day, hour, min, sec, value)
                              VALUES (?, ?, ?, ?, ?)
        chance: .001            # many inserts per day, as the chance to
    ratio: 1300                 # create a new partition is very low
  query_timeframes:
    queries:
      - query: |
                  SELECT * FROM events.timeline WHERE day= ?
                            AND (hour, min) >= (9, 00)
                            AND (hour, min, sec) <= (11, 59, 59)
      - query: |
                  SELECT * FROM events.timeline WHERE day= ?
                            AND (hour, min) >= (15, 00)
                            AND (hour, min, sec) <= (17, 59, 59)
    ratio: 10

config:
  database:
      type: Cassandra
      connection arguments: {protocol_version: 3}
  termination conditions:
    latency:
      max: 1000                 # value in ms
      consecutive: 5
    queries:
      max: 10000
      consecutive: 5
```

Listing 4.1: Configuration file example

that particular stage see section 3.3 on page 28). Finally, they put a tuple (`workload_name, queries`) into their shared `queue_out`, where `workload_name` is the chosen workload's name, and `queries` is a list. Each item in `queries` represents one query of the chosen workload, and is composed of a tuple (*`new, query_data`*). The variable *`new`* is a boolean value, which is *`True`* if a new data item will be inserted into the database. The variable *`query_data`* again is a list, composed of tuples (*`attribute_type,seed,generator_arguments`*) for each attribute in a query.

Example. *As the `WorkloadGenerator` can choose between the two workloads `insert_data` and `query_timeframes`, it will most likely choose the first one due to its high ratio value. Assuming that choice, it then iterates over all queries in that workload, which is only one `INSERT` query. For each attribute needed to fill this query it determines the seed to use for generating that data item, e.g. 123 for `day`, 456 for `hour`, 789 for `min`, 101 for `sec`, and 234 for `value`. Also, for each attributes the data type and generator arguments are determined using the data from the `conf` attribute. The boolean value for `new` is easily determined using the type of the query, which also can be found in `conf`.*

The described process results in the following tuple that will be put into `queue_out`:

```
('insert_data', [(True, [
    ('text', 123, {size : 11}),
    ('int', 456, {low : 0, high: 23}),
    ('int', 789, {low : 0, high: 59}),
    ('int', 101, {low : 0, high: 59}),
    ('int', 234, {})
])])
```

4.3.2 `DataGenerator`

Taking an item from their shared `queue_in`, the `DataGenerator`s unpack it and iterate over all queries for that workload, again iterating over all attributes of each query. For each attribute they first seed their RDG with the *`seed`* chosen by some up-chain `WorkloadGenerator`. Then they call their RDG's method that corresponds to *`attribute_type`*, passing the arguments in *`generator_arguments`*. The resulting value will be appended to a list of all data for the currently processed query. The resulting list will again be put into a list for all queries of that workload (called *`workload_data`*), coupled with the *`new`* generated by some up-chain `WorkloadGenerator`. When all data for the chosen workload was generated, the `DataGenerator`s put a tuple (*`workload_name,workload_data`*) into their shared `queue_out`.

Example. *After unpacking the tuple, the `DataGenerator` takes the first-and-only query and iterates over it's attributes. For each attribute, the private `generator` is reseeded (e.g. with seed 123 for the `date` attribute) before calling the `generator` method for the data type and using the generator arguments determined by the preceding `WorkloadGenerator` (e.g. the `pystring` method with parameters `{size : 11}` in case of the `date` attribute). As described for the general case, the resulting data is then packed*

into a tuple again, which for the chosen example might be `('insert_data', [(True, ['*+NCuWcon=P', 3, 2, 39, 5389159256853512406])])`.

4.3.3 QueryGenerator

For the QueryGenerators the first step is again taking an item from their queue_in and unpacking it. For each query in the chosen workload they look up the prepared_statement in the config using the workload_name received from up-chain Generators. They then call the execute method of their connection object for this query, handing over the prepared *statement* for this query, the query's *attributes*, a reference to the QueryGenerators shared *queue_out*, and the *metadata* tuple (*workload_name*, *query_num*, *new*). The variable *workload_name* again is the name of that workload, *query_num* is the sequence number of the executed query within that workload, and *new* the boolean marker for newly generated data items. The called execute method will itself run the query asynchronously, recording the time point of the query start. When the database answers to that query, a tuple (*result*,*start*,*end*,*mdata*) will be put into the QueryGenerators shared queue_out, where *result* is either uninitialized or some error data, *start* and *end* are the time points of query execution and result receiving, and *mdata* is the metadata tuple explained earlier in this paragraph.

Example. *Looking up the* prepared_statement *in the* config *would yield some object reference to the prepared query, which then be used as the value of the* query *parameter when calling the* execute *method. For the chosen example, the other parameters for that call would be* `['*+NCuWcon=P', 3, 2, 39, 5389159256853512406]` *for* attributes*, a reference to the* queue_out *for* queue_out*, and* `('insert_data', 0, True)` *for the* metadata *tuple. After receiving a result to this query, the following item might have been put into* queue_out*:*

```
(None,
datetime.datetime(2014, 11, 24, 15, 19, 4, 238896),
datetime.datetime(2014, 11, 24, 15, 19, 4, 236140),
('insert_data', 0, True))
```

where datetime.datetime *is a Python datetime object which stores data about a specific timestamp.*

4.3.4 LogGenerator

The final generators are the LogGenerators. They are the only generators allowed to signal the need for more generators of their type. After taking a tuple from their shared queue_in and unpacking it, they compute the time the item stayed in the queue by comparing the *end* variable with the current time. If the difference exceeds the threshold stored in their max_queue_time attribute , they set their needs_more_processes event, signaling the GeneratorCoordinator a need for more LogGenerators. Also, they determine the current second's integer value and compare it to the one stored

in their now attribute. If there is a difference, they synchronize their private processed_latencies attribute with the shared latencies attribute for the last second, then set now to the current second and empty their processed_latencies. Then, if the processed query did not result in an error, they add the acquired metadata of that query to their private processed_latencies attribute. If the query did result in an error, they raise a warning to the user.

Example. *With the input tuple described in the last example, the following tuple would first be stored in the* `processed_latencies` *attribute and then eventually be synchronized between the processes:* `(datetime.timedelta(0, 0, 2756),'insert_data', 0)`*, where* `datetime.timedelta(0, 0, 2756)` *is a Python* `datetime.timedelta` *object representing the response time of the query,* `'insert_data'` *is the workload's name, and* `0` *is the sequence number of the executed query from the chosen workload.*

4.3.5 GeneratorCoordinator's watch_and_report() method

The final processing step is the watch_and_report() method of the GeneratorCoordinator class. This method also operates in an endless loop. It is a process running separated from its calling GeneratorCoordinator, but still sharing needed attributes with its creator. Hence, it also has access to the GeneratorCoordinator's latencies object, which, together with the shutdown event, are the only attributes it needs to operate. When running, it first determines the current time. Then from the latencies attribute it gets the execution information of all queries executed in the last second. It computes the average response time over all queries and reports both that value and the number of queries of the last second to the user. Afterwards it checks for the fulfillment of termination conditions, setting the shutdown event if needed. Finally, it waits for one second before processing the next time slot.

Example. *Assuming that only the described workload was executed within the last second, the output to the user would be:*

```
2014-11-24 15:19:04   queries/sec:     1     avg latency:     2.77 ms
```

5. Evaluation

In this chapter the implemented prototype will be evaluated, first using some general benchmark requirements, and then using own requirements for a configurable testbed for scalable data management systems. Afterwards, results of performance tests will be given, including a comparison to a related tool.

5.1 Fulfillment of General Benchmark Requirements

The first thing that should be evaluated is whether the implemented prototype fulfills the general benchmark requirements stated in section 2.2.1 on page 7.

5.1.1 General Requirements

***Strong Target Audience*:** The flexibility of the proposed approach regarding DBMSs and workloads naturally implies a strong target audience, yet support for different data models needs to be examined further.

***Relevant*:** This requirement is partly fulfilled. Though the prototype does not define standard workloads with typical operations for specific problem domains, these can easily be created. Also, as the tool is aimed to test workload scenarios specific for a single user, these would have to be implemented by them anyways.

***Economical*:** As the developed tool is open source and free of charge, there are no acquisition costs. As the prototype is not yet ready for application to production systems (see the further sections of this chapter for reasons why), other costs can not yet be determined.

***Simple*:** The minimal configuration interface, using real schemata in the language of the database to test, and the clear output of the tool prove that this requirement has been a goal of the implemented prototype, though there is still room for improvements.

5.1.2 Implementation Requirements

***Fair and Portable*:** Again, this requirement does not hold as much as for a benchmarking tool, as the tool is not aimed to provide comparisons between different DBMSs.

***Repeatable*:** Although it was tried to provide the repeatability of results, this requirement is not yet reached completely. While a seedable PRNG is used for choosing the workload to execute next, the PRNG is not yet seeded deterministically before that choice. This should be the only source of randomness within the prototype, yet it could be easily removed and only has not been done because of a lack of time. Also, because of the random choice of workloads using specified chances, this has just a minor influence on the results the longer the testing process takes.

***Realistic and Comprehensive*:** The nature of this tool implies that all features typically used in the target applications are exercised, hence this requirement is fully met.

***Configurable*:** Again, this feature is fulfilled by the nature of the tool, though many improvements can be made (e.g. value distributions, data dependencies).

5.1.3 Workload Requirements

***Representativeness, Scalable*:** As workloads used in the testing process should be written by a user, it is naturally based on a workload scenario containing a representative set of interactions, and is scalable preserving the relation to the real-life business scenario modeled.

***Metric*:** Only two metrics are reported to the user yet (queries per second and average latency), both of which are easily understandable. Of course there are other metrics worth reporting as well (e.g. percentiles for both latencies and queries per second), but there was not enough time to implement these.

5.2 Fulfillment of Self-elaborated Requirements

Next, the prototype will be evaluated using the requirements from section 3.1 on page 19, which were derived from the just evaluated requirements from section 2.2.1 on page 7. All of them are basically fulfilled: it is possible to specify schemata and workloads as well as ratios between those workloads. Also, it is possible to specify value domains and termination conditions, although these features could and should be extended much further, e.g. by allowing different distributions and specifying termination conditions relying on other statistical properties. Furthermore, metadata is derived by the prototype where possible, hence delivering the user from non-essential definitions.

The program is able to scale when given more parallel computing power, as the performance test in section 5.3 on the next page will show. Its extensibility remains to be investigated further, as the tight timeframe did not allow implementing support for

more than one type of database. However, the program's modular design should allow many types of extensions.

As required, the data generation is parallelizable and allows regeneration of previously generated content, while it has also been tried to keep synchronization effort between generators minimal.

Furthermore, the prototype uses a file-based storage of runtime configurations, in which all needed data is stored. This supports repeated execution with the same chosen settings as well as fast testcase adjustments and easy exchange of settings, even between users and different hardware configurations. This also includes no need for user interaction other than stating the configuration file to use when starting the program.

To support adaptions, extensions, and error corrections, it was tried to thoroughly document the source code using comments. However, some rework might be needed, as not all comments are up to date and document behavior of past versions. Still, most functions and interfaces are described correctly.

5.3 Performance Evaluation

To evaluate the performance of the prototype, a set of tests has been conducted. Two aspects have been investigated: whether the prototype is able to scale when given more parallel computing power, and how strong the influence of the `chance` parameter is because it heavily affects the number of recursions taken for each seed determination. The test environment used to conduct all test consisted of a desktop computer equipped with an Intel Core i7-4770K, 8GB DDR3-1600 RAM and a Samsung 256GB SSD 840, running a 64-bit version of Arch Linux[1] with the latest packages on the day of test conduction (2014-11-28). To set the prototyp's processor affinity the `taskset` command was used, which is part of many major Linux distributions.

As all used configuration files differ in just one line for this experiment, only one example is given in listing 5.1. Line 23 shows that this configuration file was used for the test with `chance`$= \frac{1}{4}$. This is the only line that has been changed to conduct the other tests to compare the influence of the `chance` parameter.

Furthermore, the configuration file shows workloads with `ratio: 0`, hence workloads that will never be executed. This is due to the fact that the same configuration file was used to perform the tests for section 5.4.2 on page 56 with ratio values differing from the ones shown. This also exemplifies how easy test configurations can be altered by simply changing some values.

```
1 schemata:
2   testkeyspace:
3     definition: |
4                   CREATE KEYSPACE IF NOT EXISTS testkeyspace
5                   WITH REPLICATION = {'class':'SimpleStrategy',
```

[1] https://www.archlinux.org, accessed 2014-12-01

```
6                                       'replication_factor':1}
7     tables:
8       typestest:
9         definition: |
10                        CREATE TABLE testkeyspace.typestest (
11                              name text, choice boolean, date timestamp,
12                              address inet, dbl double, lval bigint,
13                              ival int, uid timeuuid, value blob,
14                              PRIMARY KEY((name,choice),
15                              date, address, dbl, lval, ival, uid))
16  workloads:
17    create:
18      queries:
19        - query: |
20                    INSERT INTO testkeyspace.typestest
21                    (name,choice,date,address,dbl,lval,ival,uid,value)
22                    VALUES (?,?,?,?,?,?,?,?,?)
23          chance: 0.25
24      ratio: 1
25    read:
26      queries:
27        - query: |
28                    SELECT * FROM testkeyspace.typestest where name = ?
29                    AND choice = ? LIMIT 100
30      ratio: 0
31    update:
32      queries:
33        - query: |
34                    UPDATE testkeyspace.typestest SET value=?
35                    WHERE name=? AND choice=? AND date=? AND address=?
36                    AND dbl=? AND lval=? AND ival=? and uid=?
37      ratio: 0
38    delete:
39      queries:
40        - query: |
41                    DELETE FROM testkeyspace.typestest
42                    WHERE name = ? AND choice = ? AND date = ?
43      ratio: 0
44  config:
45    database:
46        type: Cassandra
47        connection arguments: {protocol_version: 3}
48    termination conditions:
49      latency:
50        max: 1000               # value in ms
51        consecutive: 5
52      queries:
53        max: 10000
54        consecutive: 5
```

Listing 5.1: Example of the configuration files used for evaluation

Also, the results where obtained with the `QueryGenerators` `connection` not querying against a real database. Instead, the `execute` function was rewritten to directly put data into the `queue_out`, hence simulating an immediate response. This approach has been chosen to cancel out a possible influence of the database on the measurements taken.

Also, the prototype was only allowed to create a maximum of eight child processes. This can be configured when constructing the `GeneratorCoordinator` class, but is not yet configurable in the configuration file.

Within this section the results of the conducted test shown in table 5.1 will be used to evaluate the performance properties of the implemented prototype. Furthermore, these results are plotted into figure 5.1 on the following page and figure 5.2 on page 55.

# Cores	chance=1	chance=$\frac{1}{4}$	chance=$\frac{1}{16}$	chance=$\frac{1}{64}$
1	1265 q/s	1275 q/s	950 q/s	559 q/s
2	1903 q/s	1700 q/s	1350 q/s	604 q/s
4	2582 q/s	2162 q/s	1683 q/s	800 q/s
8	2620 q/s	2265 q/s	1608 q/s	720 q/s

Table 5.1: Performance data of the prototype with different `chance` values, q/s meaning queries per second

5.3.1 Scalability

Figure 5.1 on the following page illustrates the prototypes ability to scale vertically to the given parallel computing power up to four cores. The minimal performance gains when doubling the number of cores available to the prototype from four to eight are probably caused by the fact that the used processor only has four physical cores and provides two virtual cores for each physical core via a technique called Hyper-Threading. As the prototype does not use threads, but multiple processes (due to restrictions in Python), it might not be able to take advantage of this technique. Hence, to gain conclusive results the prototype should be tested again with a machine providing 8+ physical cores. Sadly, no such machine was available to conduct these tests, hence this remains an open question.

However, the acquired results indicate that the chosen approach is able to scale vertically. The scale factor ranges from 1.25 to 1.5 most of the time, with exceptions for the step from four to eight cores and the step from one to two cores with `chance`= $\frac{1}{64}$. It should also be noted that the prototype is mostly unoptimized and hence might be improved significantly.

5.3.2 Influence of the `chance` Parameter

As figure 5.2 on page 55 shows, the `chance` parameter has heavy influence on the performance of the prototype, as the output of the prototype decreases exponentially.

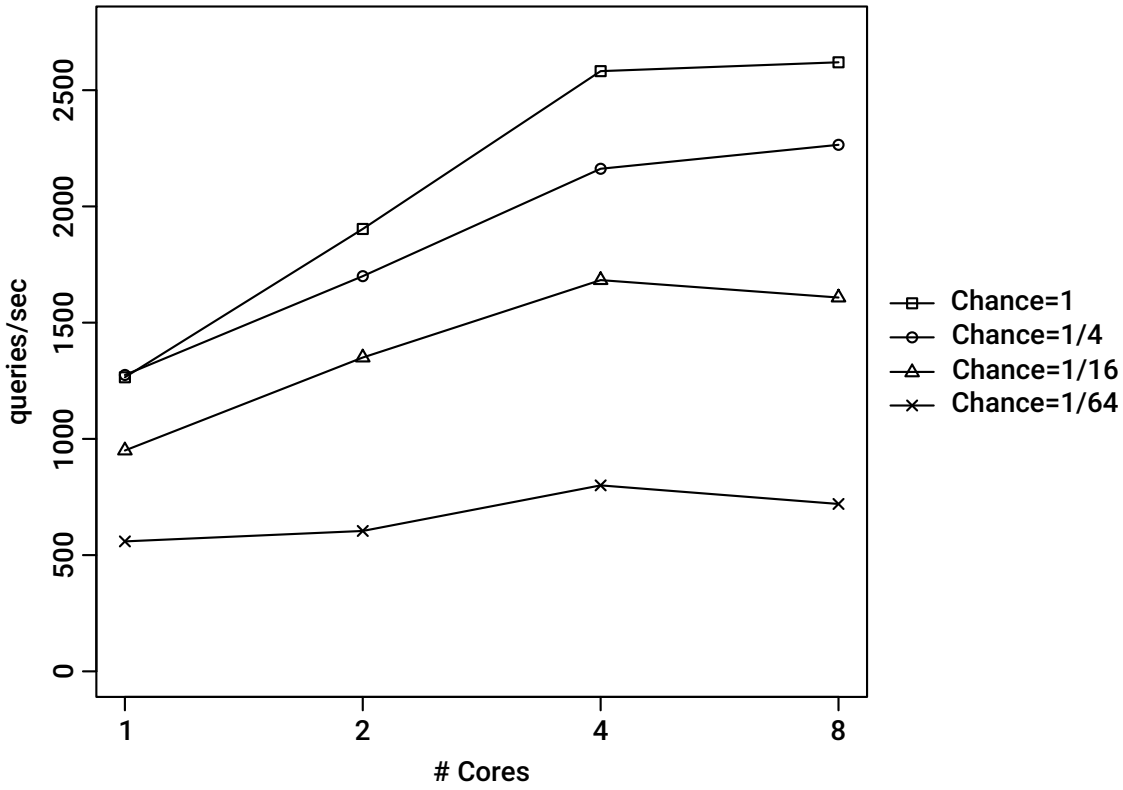


Figure 5.1: Scalability of the prototype

This behavior was already predicted in section 3.3.2 on page 30 and shows a clear weakness of the chosen approach. It remains to be tested whether the bitmap-tree approach described in the same section outperforms the chosen approach in most cases and does not show other defects. However, this defect is only valid for data models with more than one key level, hence other types can be tested with no restrictions. All other users should be advised to avoid small `chance` values when configuring the prototype.

Also, the nearly identical performance when using four and eight cores easily recognizable in this figure nicely illustrates the assumption that the prototype can not make use of the Hyper-Threading offered by the processor.

5.4 Comparison with Cassandra's Included Stress Tool

Some weeks after this thesis' key points had been determined, Apache published version 2.1 of C*. Among many other improvements, this version included an improved stress tool, which allows the testing of arbitrary tables (with minor restrictions to the data types). As it was already to late to use this tool as a basis for this master's thesis, it was only used as an inspiration for some parts, especially the configuration file. However, there are major similarities and differences that will be investigated in this section.

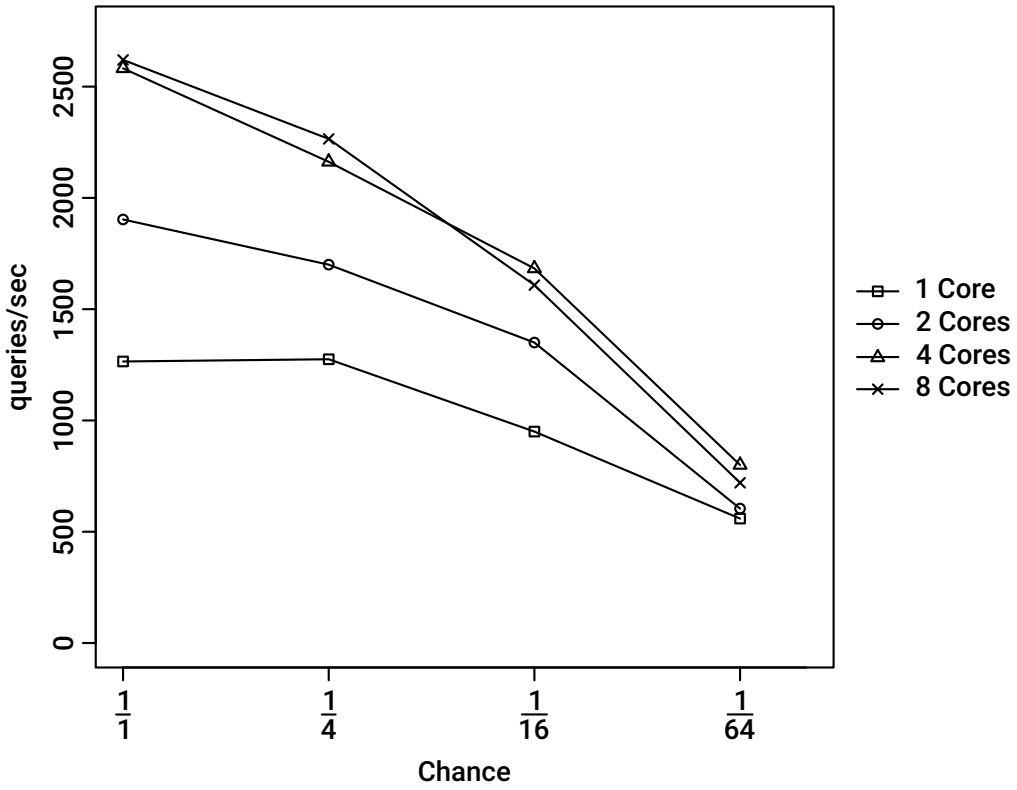


Figure 5.2: Influence of the chance parameter

Also, there will be a performance comparison of the prototype and C*'s stress testing tool named *cassandra-stress*.

5.4.1 Similarities and Differences

An overview of the similarities and differences of the compared tools is given in table 5.2 on the next page, which will be elaborated further in the following.

Both tools aim to stress test a specific database instance. While cassandra-stress was designed to only test C* instances, the prototype's concept is to allow testing of arbitrary databases. Also, the prototype is able to perform tests on complex schemata with multiple tables, cassandra-stress only allows to test one specific table. Furthermore, it understands workloads as a single query, while the prototype sees them as a sequence of queries. While both allow specifying a workload's probability of being executed, cassandra-stress can not perform queries in fixed sequences. Both approaches are modular and thus extensible, but use different programming languages (cassandra-stress uses Java). Furthermore, cassandra-stress can not yet create map types, while the prototype can produce all basic data types. Cassandra-stress allows fine-grained definitions of data distributions and even provides different types of distributions, while the prototype only allows specifying limits for values or sizes and only supports uniform distributions. Also, cassandra-stress tests iteratively, increasing the count of used threads

	cassandra-stress	prototype
databases	Cassandra	Cassandra (but extensible)
schemata	single table	multiple tables
workloads	single queries	multiple queries
modular & extensible	yes, but C* centered	yes
programming language	Java	Python
supported data types	all, but maps	all basic data types, including maps
supported distributions	multiple	only uniform (but extensible)
testing process	iterative with increasing thread count	single run with fixed maximum number of processes

Table 5.2: Feature comparison of the prototype and cassandra-stress

for each test until some termination condition is met. The prototype just conducts one test where it creates a predefined maximum number of child processes.

5.4.2 Performance Comparison

All tests have been conducted with the same configuration as in section 5.3 on page 51. The only difference are changed values of the configuration file used to test the prototype (see listing 5.1 on page 51), namely the `chance` value of 1, the `ratio` value of 2 for workload `create`, and a `ratio` value of 1 for all other workloads.

Cassandra-stress used the configuration file shown in listing 5.2, which was constructed in a way that both tools should perform nearly the same operations. However, as the implementation of cassandra-stress is very complex, it can not be guaranteed that both tools performed the same operations in the same way. Furthermore, cassandra-stress was called with the additional parameters `ops(create=2, read=1, update=1, delete=1)`, which specifies the ratio with which the stated workloads will be executed, and `no-warmup`, which prevents the tool from executing a certain number of iterations for each workload before the actual performance test.

```
1  keyspace: testkeyspace
2  keyspace_definition: |
3    CREATE KEYSPACE testkeyspace WITH
4    replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
5
6  table: typestest
7  table_definition: |
8    CREATE TABLE typestest (
9          name text, choice boolean, date timestamp,
10         address inet, dbl double, lval bigint,
11         ival int, uid timeuuid, value blob,
12         PRIMARY KEY((name,choice), date, address, dbl, lval, ival, uid))
13
14 columnspec:
```

```
  - name: name
    size: fixed(10)
  - name: value
    size: fixed(50)

queries:
   create:
      cql: |
                  INSERT INTO testkeyspace.typestest
                  (name,choice,date,address,dbl,lval,ival,uid,value)
                  VALUES (?,?,?,?,?,?,?,?,?)
      fields: samerow
   read:
      cql: |
                  SELECT * FROM testkeyspace.typestest where name = ?
                  AND choice = ? LIMIT 100
      fields: samerow
   update:
      cql: |
                  UPDATE testkeyspace.typestest SET value=?
                  WHERE name=? AND choice=? AND date=? AND address=?
                  AND dbl=? AND lval=? AND ival=? and uid=?
      fields: samerow
   delete:
      cql: |
                  DELETE FROM testkeyspace.typestest
                  WHERE name = ? AND choice = ? AND date = ?
      fields: samerow
```

Listing 5.2: Configuration file used for cassandra-stress

All tests were conducted with a Cassandra (2.1.2) instance running on the same machine, as no other machine with enough computing power was available to run C* on.

The results of this test are shown in table 5.3 and are plotted in figure 5.3 on the following page. As cassandra-stress outputs values for all test runs with different thread counts, it was chosen to always take values of runs with a mean latency of 0.5 milliseconds. It should be noted that the values for the chosen latency where among the best of all test runs, so this choice does only slightly diminish the values, if does so at all.

# Cores	prototype	cassandra-stress
1	1033 q/s	37165 q/s
2	1356 q/s	45518 q/s
4	1970 q/s	48304 q/s
8	1987 q/s	44916 q/s

Table 5.3: Performance comparison data of the prototype and cassandra-stress, q/s meaning queries per second

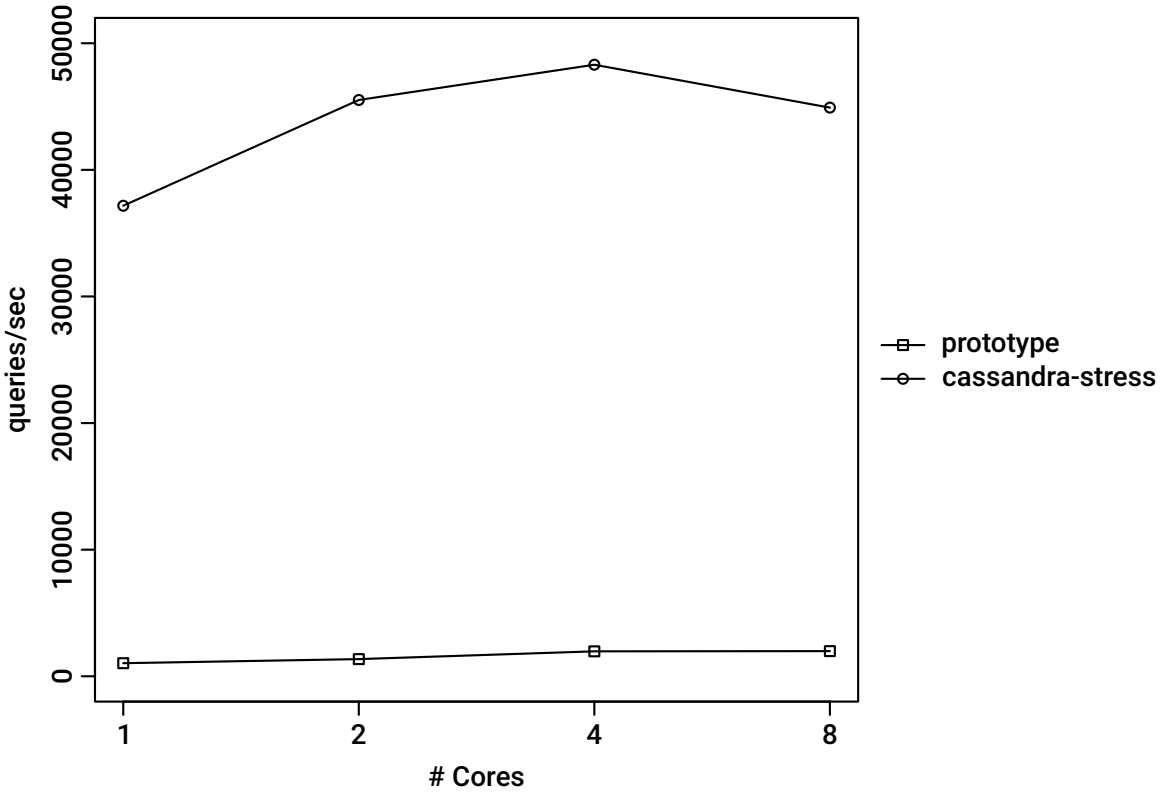


Figure 5.3: Performance and scaling comparison with cassandra-stress

The results show that cassandra-stress has a superior performance. This is probably first and foremost caused by the usage of Java instead of Python, which allows threading and hence might benefit strongly from the processor used. This assumption is supported by the observation that cassandra-stress fully utilized all cores, while the prototype used different amounts of available computing power, depending on the number of cores (from about 100% when testing with a single core, to about 65% per core when testing with eight cores). Also, as stated in section 4.1.1 on page 35, Python is rather slow when compared to other languages. As a popular benchmark website for different programming languages[2] shows, the median performance of Python is 27 times slower than Java, which matches the obtained values. Furthermore, the prototype has not been optimized in any way, and is still what it is – a prototype. In contrast, cassandra-stress is a rather mature tool that has been developed over years.

Again, the data shows that the prototype scales linearly, and it even suggests that the prototype scales much better than cassandra-stress. As the C* instance used was running on the same machine as the program trying to stress it, the assumption should not be made. It is rather assumed that the database was already limiting cassandra-stress's results, as otherwise the tool would show very poor scaling.

[2] http://benchmarksgame.alioth.debian.org, accessed 2014-12-01

A further thing to notice is that the prototype's latency values got better the more cores it was allowed to use. It remains unclear why this behavior was observed, as the tests conducted in section 5.3 on page 51 did not show such properties. Contrary, no latencies have been seen at all, although only the actual querying to the database was short-circuited. Maybe the driver used to communicate with C* (the Python driver from DataStax) introduced some of these latencies, but further investigation has to be done before any final statement can be given on that matter.

Summing up the results of the comparison with cassandra-stress it can be ascertained that while the prototype is currently too slow, it provides a more flexible approach as it allows testing more configuration. Cassandra-stress on the other hand provides a superior performance, but does not allow specifying workloads consisting of more than one query and also does not allow using more than one table at a time.

5.5 Additional Remarks

As stated in section 4.1.1 on page 35, it is often possible to strongly increase the performance of Python programs by using different Python implementations. This has been tested shortly by using PyPy[3]. However, the results with that implementation even showed decreased performance. This behavior sadly has been observed by others when using the `multiprocessing` package and thus might be a characteristic of that implementation. It remains to be investigated if other Python implementations might deliver a performance improvement.

[3] http://pypy.org, accessed 2014-12-01

6. Conclusion and Future Work

This chapter will give an overview of the work done in this thesis. Afterwards, questions and problems of the presented approach that still need to be investigated are presented.

6.1 Conclusion

Testing an application specific database schema for its performance under heavy load and its scalability for increasing load and vast amounts of data is a difficult task. However, it is a property worthwhile to evaluate, as it helps to prevent future problems and probably expensive accommodations. As the area of NoSQL systems is very inhomogeneous and still yields new approaches, developing a tool to test these is a difficult task.

This thesis contributed to that task by investigating how such a tool should be designed. It first covered the required knowledge, terms and techniques to understand the backgrounds of the field. Therefore, it explained the reasons that led to the development of NoSQL systems and showed their different data models. Then, it was focused on the related field of database benchmarking by pointing out its targets and general requirements. Also, as data to test databases needs to be generated somehow, it has been shown how pseudo-random number generators work and how they can produce arbitrary pseudo-random data.

In chapter 3 on page 19 a basic concept for the design of a configurable testbed for scalable data management systems was proposed. First, the basic requirements where stated, which are user-definable schemata, workloads, ratios between those workloads, value domains and termination conditions. It was also stated that basic requirements of such a framework are scalability and extensibility, which can only be achieved through a highly modular and parallelizable design. Furthermore, it was shown that the generation of random data has to be efficient, fast, scalable, and repeatable. The proposed solution is to use the nature of PRNGs, which means to use each state of a PRNG for the

production of a distinct data item and sharing the information about which and how seeds were used with all data creating processes. The final stated requirement was allowing users to repeatedly execute the conducted tests by using a file-based storing of configuration files. This also allows fast testcase adjustments, archiving, and easy exchange of settings, even between users and different hardware settings.

After determining requirements, the proposed overall design was presented. It consists of different modules with specific functions in the testing process, such as a preparation module, a coordinator that manages all generators producing data in the testing process, and a connection module, which acts as an abstraction layer between the framework and specific database instances. Next, the testing approach was described, where first a workload is chosen and the seeds to use are identified. Afterwards, the needed data is generated, which is then used to fill the queries of the chosen workload. The filled queries are then send to the database. Next, execution information for stated queries is aggregated and presented to the user, until some predefined termination conditions are fulfilled. As a closure to that chapter, a specific approach to the management of seeds was presented.

Chapter 4 on page 35 gave a detailed look at what has been implemented in the prototype, and how this was done. For that, first the choices for Python as programming language, Apache Cassandra as NoSQL database to use, YAML as configuration file format, and some used data structures were explained. Afterwards, the application schema was described by an explanation of the main attributes and methods of the implemented classes and their instances. Finally, the whole process from the choice of a workload to outputting momentary performance data to the user was explained by following a workload through the process.

Evaluations of the developed prototype were made in chapter 5 on page 49. Conclusions of this chapter were that it fulfills most stated requirements, though there are still some properties that could be improved. The comparison to a related tool showed that the performance has to be improved greatly, but the overall approach seems to be able to scale with increasing parallel computing power.

6.2 Future Work

This section will first cover questions and problems of the presented approach that still have to be investigated. Afterwards, open problems of the implemented prototype will be presented. Readers interested in implementing the proposed enhancements are referred to the freely available source code[1] of the prototype.

6.2.1 Open Questions and Problems of the General Approach

Although data might have complex dependencies, it was chosen not to represent these dependencies in the chosen approach, as it is a complex question itself. However, some

[1] https://github.com/causa-prima/COLT

work on this topic has already been done ([RP11; FPR12]), and it should be investigated if and how these approaches could be included into the designed framework.

Also, it was not considered that data might have other special characteristics, like always increasing values (e.g. for date values or counters). However, these properties have heavy influence e.g. on range queries, and thus it is important to examine how this property could be introduced to the framework.

Furthermore, it is not yet regarded that workloads might shift over time. While an application might experience a heavy amount of new data in the start and relatively few read queries, this might shift to the exact opposite. Hence, it might be worth investigating how such behavior could be included into the presented approach, which would allow modeling the life cycle of a database.

Further minimizations of synchronization points are also of great interest, as this would increase the scalability. Also, this would probably facilitate parallelization across different machines, which is yet another field that should be investigated.

As the approach was only implemented using a specific data model, it remains to be determined if the approach is suited for different data models as well.

6.2.2 Possible Enhancements of the Prototype

As the prototype has been implemented within a very tight timeframe, it has not been optimized in any way. As the results of chapter 5 on page 49 show, this has to be done. Therefore, profiling of the code should be done, identifying the cumbering parts and optimizing them. This profiling should also be done to answer whether the program itself distorts the latency values and to minimize that distortion.

It also might increase the prototype's performance to group a complete chain of generators into a single thread, as this could probably facilitate communication between them. Furthermore, it has to be investigated whether the `watch_and_report` method should be converted to a generator, as this method is not yet able to scale.

More and detailed execution information should be provided to the user, e.g. percentiles for latencies, or even latencies per workload. This would enable the user to gain a better understanding of his schema's performance and also allow for more options to terminate the testing.

Also, it is clearly desirable to implement data distributions other than uniform distribution, and make these available to the user. Therefore, the random data generator would probably have to be modularized further.

Lastly, it would be beneficial for some user to be able to start the testing on an already filled database, hence this would be another desirable feature.

Bibliography

[BFS83] Paul Bratley, Bennett L Fox, and Linus E Schrage. *A guide to simulation*. 2nd. Vol. 2. Springer, 1983 (cit. on p. 13).

[BK98] Richard N. Bromberg and Richard W. Kupcunas. "Application and method for benchmarking a database server". US Patent 5,819,066. Oct. 6, 1998. URL: https://www.google.com/patents/US5819066 (cit. on p. 8).

[Cat10] Rick Cattell. "Scalable SQL and NoSQL Data Stores". In: *ACM SIGMOD Record* 39.4 (Dec. 2010), pp. 12–27. DOI: 10.1145/1978915.1978919 (cit. on pp. 3, 4).

[Cod97] Paul D. Coddington. *Random Number Generators for Parallel Computers*. NPAC Technical Report. Northeast Parallel Architectures Center, 1997. Published in The NHSE Review, 1996 Volume, Second Issue (cit. on p. 17).

[Coo+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154. DOI: 10.1145/1807128.1807152 (cit. on pp. 3, 11, 12).

[DP88] A. De Matteis and Simonetta Pagnutti. "Parallelization of random number generators and long-range correlations". English. In: *Numerische Mathematik* 53.5 (1988), pp. 595–608. ISSN: 0029-599X. DOI: 10.1007/BF01397554 (cit. on p. 17).

[DP90] A. De Matteis and Simonetta Pagnutti. "Long-range correlations in linear and non-linear random number generators". In: *Parallel Computing* 14.2 (1990), pp. 207–210. ISSN: 0167-8191. DOI: 10.1016/0167-8191(90)90108-L (cit. on p. 17).

[FK09] Daniela Florescu and Donald Kossmann. "Rethinking Cost and Performance of Database Systems". In: *SIGMOD Rec.* 38.1 (June 2009), pp. 43–48. ISSN: 0163-5808. DOI: 10.1145/1558334.1558339 (cit. on p. 7).

[Fol+13] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. "Benchmarking in the Cloud: What It Should, Can, and Cannot Be". In: *Selected Topics in Performance Evaluation and Benchmarking* (2013), pp. 173–188. ISSN: 1611-3349. DOI: 10.1007/978-3-642-36727-4_12 (cit. on p. 7).

[FPR12] Michael Frank, Meikel Poess, and Tilmann Rabl. "Efficient Update Data Generation for DBMS Benchmarks". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. Boston, Massachusetts, USA: ACM, 2012, pp. 169–180. ISBN: 978-1-4503-1202-8. DOI: 10.1145/2188286.2188315 (cit. on p. 63).

[Gra93] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1558601597 (cit. on p. 7).

[Gro+13] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. "Data management in cloud environments: NoSQL and NewSQL data stores". In: *Journal of Cloud Computing: Advances, Systems and Applications* 2.1 (2013), p. 22. DOI: 10.1186/2192-113X-2-22 (cit. on pp. 3, 4).

[Hel98a] Peter Hellekalek. "Don't Trust Parallel Monte Carlo!" In: *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*. PADS '98. Banff, Alberta, Canada: IEEE Computer Society, 1998, pp. 82–89. ISBN: 0-8186-8457-7. DOI: 10.1145/278008.278019 (cit. on p. 17).

[Hel98b] Peter Hellekalek. "Good random number generators are (not so) easy to find". In: *Mathematics and Computers in Simulation* 46 (5–6 June 1998), pp. 485–505 (cit. on p. 14).

[Hew10] Eben Hewitt. *Cassandra: The Definitive Guide*. First Edition. O'Reilly Media, Dec. 2010. ISBN: 9781449396640 (cit. on p. 5).

[HJ11] Robin Hecht and Stefan Jablonski. "NoSQL evaluation: A use case oriented survey". In: *Proceedings of the 2011 International Conference on Cloud and Service Computing*. CSC '11. IEEE Computer Society. IEEE Computer Society, 2011, pp. 336–341. DOI: 10.1109/CSC.2011.6138544 (cit. on pp. 3–5).

[Hup09] Karl Huppler. "The Art of Building a Good Benchmark". English. In: *Performance Evaluation and Benchmarking*. Ed. by Raghunath Nambiar and Meikel Poess. Vol. 5895. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 18–30. ISBN: 978-3-642-10423-7. DOI: 10.1007/978-3-642-10424-4_3 (cit. on p. 7).

[Kar+99] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. "Web caching with consistent hashing". In: *Computer Networks* 31.11–16 (May 1999), pp. 1203–1213. DOI: 10.1016/S1389-1286(99)00055-9 (cit. on p. 5).

[Kon+11] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. "On the Elasticity of NoSQL Databases over Cloud Management Platforms". In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM '11. ACM, 2011. DOI: 10.1145/2063576.2063973 (cit. on p. 3).

[Kou05] Samuel Kounev. "Performance Engineering of Distributed Component-based Systems". PhD thesis. 2005 (cit. on p. 7).

[KTB13] Dirk P. Kroese, Thomas Taimre, and Zdravko I. Botev. *Handbook of Monte Carlo Methods*. Wiley Series in Probability and Statistics. John Wiley & Sons, 2013. ISBN: 978-1-118-01495-0 (cit. on pp. 13–15).

[LEc01] Pierre L'Ecuyer. "Software for uniform random number generation: distinguishing the good and the bad". In: *Proceeding of the 2001 Winter Simulation Conference* (2001), pp. 95–105. DOI: 10.1109/wsc.2001.977250 (cit. on p. 14).

[LEc90] Pierre L'Ecuyer. "Random Numbers for Simulation". In: *Communications of the ACM* 33.10 (Oct. 1990), pp. 85–97. ISSN: 0001-0782. DOI: 10.1145/84537.84555 (cit. on p. 13).

[LEc94] Pierre L'Ecuyer. "Uniform random number generation". English. In: *Annals of Operations Research* 53.1 (1994), pp. 77–120. ISSN: 0254-5330. DOI: 10.1007/BF02136827 (cit. on pp. 13–15).

[LM10] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922 (cit. on p. 5).

[LOS14] Pierre L'Ecuyer, Boris Oreshkin, and Richard Simard. "Random Numbers for Parallel Computers: Requirements and Methods". In: *Mathematics and Computers in Simulation* (2014). Pre-published (cit. on p. 17).

[Mat+07] Makoto Matsumoto, Isaku Wada, Ai Kuramoto, and Hyo Ashihara. "Common Defects in Initialization of Pseudorandom Number Generators". In: *ACM Trans. Model. Comput. Simul.* 17.4 (Sept. 2007). ISSN: 1049–3301. DOI: 10.1145/1276927.1276928 (cit. on p. 17).

[MBS12] Siba Mohammad, Sebastian Breß, and Eike Schallehn. "Cloud Data Management: a Short Overview and Comparison of Current Approaches". In: *Proceedings of the 24st Workshop "Grundlagen von Datenbanken 2012" (GvD 2012)*. Ed. by Ingo Schmitt, Sascha Saretz, and Marcel Zierenberg. Vol. 850. CEUR-WS, 2012, pp. 41–46 (cit. on p. 4).

[MN98] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995 (cit. on pp. 14, 15).

[NA11] Samuel Neves and Filipe Araujo. "Fast and Small Nonlinear Pseudorandom Number Generators for Computer Simulation". English. In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski. Vol. 7203. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 92–101. ISBN: 978-3-642-31463-6. DOI: 10.1007/978-3-642-31464-3_10 (cit. on p. 17).

[NA14] Samuel Neves and Filipe Araujo. "Engineering Nonlinear Pseudorandom Number Generators". English. In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 96–105. ISBN: 978-3-642-55223-6. DOI: 10.1007/978-3-642-55224-3_10 (cit. on p. 17).

[Pre+07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007. ISBN: 0521880688 (cit. on p. 14).

[Pri08] Dan Pritchett. "BASE: An Acid Alternative". In: *Queue* 6.3 (May 2008), pp. 48–55. DOI: 10.1145/1394127.1394128 (cit. on p. 3).

[RP11] Tilmann Rabl and Meikel Poess. "Parallel Data Generation for Performance Analysis of Large, Complex RDBMS". In: *Proceedings of the Fourth International Workshop on Testing Database Systems*. DBTest '11. Athens, Greece: ACM, 2011, 5:1–5:6. ISBN: 978-1-4503-0655-3. DOI: 10.1145/1988842.1988847 (cit. on pp. 9, 13, 63).

[Sac+09] Kai Sachs, Samue Kounev, Jean Bacon, and Alejandro Buchmann. "Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark". In: *Performance Evaluation* 66.8 (2009). Selected papers of the Fourth European Performance Engineering Workshop (EPEW) 2007 in Berlin, pp. 410–434. ISSN: 0166-5316. DOI: 10.1016/j.peva.2009.01.003 (cit. on p. 7).

[Sac10] Kai Sachs. "Performance Modeling and Benchmarking of Event-Based Systems". PhD thesis. TU Darmstadt, 2010 (cit. on p. 7).

[Sal+11] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. "Parallel Random Numbers: As Easy As 1, 2, 3". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. New York, NY, USA: ACM, 2011, pp. 1–12. DOI: 10.1145/2063384.2063405 (cit. on pp. 9, 17).

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012 (cit. on p. 3).

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 6th Edition. Connect, learn, succeed. McGraw-Hill, 2011 (cit. on pp. 7, 9).

[WH82] Brian A. Wichmann and I. David Hill. "Algorithm AS 183: An efficient and portable pseudo-random number generator". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 31.2 (1982), pp. 188–190 (cit. on p. 14).

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den