Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Roman Betík

Automatic Generation of Synthetic XML Documents

Department of Software Engineering

Supervisor of the master thesis:	Doc. RNDr. Irena Holubová, Ph.D.
Study programme:	Software Systems
Specialization:	Software Engineering

Prague 2015

I would like to thank my supervisor Doc. RNDr. Irena Holubová, Ph.D. for leading my work. Then I would like to thank RNDr. Jakub Yaghob, Ph.D. for creating virtual servers that I used during my experiments. I would also like to thank my family, especially my parents and sister, my colleagues and my girlfriend Veronika for all the support during my studies.

Název práce: Automatické generování umělých XML dokumentů

Autor: Roman Betík

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: doc. RNDr. Irena Holubová, Ph.D.

Abstrakt: Cílem této práce je prozkoumat možnosti a omezení v generování umělých XML a JSON dokumentů používaných v oblasti Big Data. První část práce zkoumá vlastnosti nejpoužívanějších XML generátorů, Big Data a JSON generátorů a porovnává jejich vlastnosti. Další část práce popisuje návrh vlastního algoritmu na generování semistrukturovaných dat. Hlavní zaměření algoritmu je paralelní vykonávání procesu generování se zachovaním možností na kontrolu obsahu generovaných dokumentů. Generátor umožňuje využít vzorky skutečných dat v procesu generování dat umělých a je také schopen automaticky generovat jednoduché odkazy mezi výstupními dokumenty ve formátu JSON. Poslední část práces poskytuje výsledky experimentů s generátorem při testování databáze MongoDB, popisuje jeho přínos a porovnává ho s jinými řešeními.

Klíčová slova: XML, JSON, Big Data, generátor, testování, benchmark, umělá data, NoSQL

Title: Automatic Generation of Synthetic XML Documents

Author: Roman Betík

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D.

Abstract: The aim of this thesis is to research the current possibilities and limitations of automatic generation of synthetic XML and JSON documents used in the area of Big Data. The first part of the work discusses the properties of the most used XML data generators, Big Data and JSON generators and compares them. The next part of the thesis proposes an algorithm for data generation of semistructured data. The main focus of the algorithm is on the parallel execution of the generation process while preserving the ability to control the contents of the generated documents. The data generator can also use samples of real data in the generation of the synthetic data and is also capable of automatic creation of simple references between JSON documents. The last part of the thesis provides the results of experiments with the data generator exploited for the purpose of testing database MongoDB, describes its added value and compares it to other solutions.

Keywords: XML, JSON, Big Data, generator, testing, benchmark, synthetic data, NoSQL

Contents

Introduction

The original aim of this thesis was to analyze existing XML [1] data generators, their options and limitations. The main target of the work was a proposal of custom XML data generator with the focus on the main structural properties of XML data like the number of elements, number of attributes, fan-out etc. This generator should have had a set of simple configuration options which affected the generated data. However, we have found out that semi-structured data in XML or JSON [2] format is currently popular especially in the area of Big Data [3]. In addition, there are not many data generators for this type of workloads, so we changed our focus to this new and challenging area.

Nowadays, the computer systems of many companies work with very large data sets. This is mostly due to the fact that this data is being gathered and generated by cheap and information-capturing mobile devices, software logs, cameras, satellites, different sensors, users of social media and networks and many more. This introduced many new problems into how this data is stored, processed, searched, transferred, visualized etc. The term Big Data is closely connected to NoSQL databases which are used as data stores.

In our context of semi-structured data we especially focus on document databases. Document databases are one of NoSQL [4] database types. They store data in the form of semi-structured or structured documents, in formats like XML or JSON. Usual use cases for these databases include event logging, e-commerce applications, real-time analytics, content management systems and many more. Document databases are currently popular storage type for many new applications. There are several implementations and the developers must choose which one they are going to use in their solution. They usually involve large data sets which affects the performance of the system.

Testing such systems usually involves using some type of test data. It can be either the real-world data from a similar system or synthetically generated data. Obtaining real-world data is often difficult due to its size and/or privacy concerns. Synthetic data generation is then often the only option. Therefore, we are going to analyze all the suitable XML data generators, Big Data and JSON data generators and describe their options, advantages and disadvantages. Our main focus, however, will be generation of documents in JSON format. We will ten study and demonstrated the abilities and advantages of the generator using MongoDB database. This particular database is currently popular in the Big Data area and it is also a good candidate where to test our solution. We will also describe options how to extend our results to other formats or databases.

Data generation of large data sets involves a new set of challenges. Many of the currently available data generators do not scale well to produce data sets in parallel. We will also describe a method which uses existing data to produce synthetic data of larger proportions. We will also investigate available options in the generation of references between JSON documents. Last, but not least, we will create an experimental version of the data generator and test this solution on a cluster of servers to show its capabilities.

Aim of this Thesis

We have two aims in this thesis. The first one is to analyse existing data generators (XML, JSON and also special Big Data generators). Many of these are a part of some benchmark. We want to describe the properties of each of these data generators and compare them mutually.

Currently available data generators capable of producing structured documents are in most cases web applications and therefore not suitable for generation of bigger data sets. Data generators for Big Data in general focus only on selected specific use cases and cannot generate different data sets without complex configurations or reprogramming. Therefore, the main aim of this thesis is to propose a solution for Big Data generator producing documents for document databases, in our case MongoDB. The data generator should be able to run in parallel on multiple servers and be able to produce different volumes of data at different data generation rates. It should make it easy to customize the contents of the generated data.

After describing the proposed solution we want to show how this data generator can be used in practice via experiments and how it compares to other available solutions.

Structure of the Document

Chapter 1 contains the definitions of the most important terms used throughout the rest of the text.

Chapter 2 contains the introduction to Big Data, NoSQL databases and related terminology.

Chapter 3 contains the analysis of the existing Big Data, JSON and XML data generators.

Chapter 4 contains description of the MongoDB database.

Chapter 5 discusses the architecture of the proposed solution and implementation details of the data generator.

Chapter 6 describes the user interface of the data generator, its installation, usage and configuration options.

Chapter 7 contains the description of the experiments we executed, the results and their meaning.

Chapter 8 summarises the work and proposes some improvements and ideas that can be further explored.

1. Definitions

This chapter contains descriptions of the most important terms and constructs which we use throughout the thesis. Most of the XML definitions are from [6].

1.1 XML Definitions

The XML (Extensible Markup Language) [1] is a markup language that defines rules for encoding documents in human and machine-readable format. The design goals for XML were simplicity, generality and usability across the Internet. It is defined by the World Wide Web Consortium in XML 1.0 Specification [1]. XML format is commonly used as an interchange data format.

Definition 1. An XML document *is a finite ordered tree $T = (\Sigma, N, E, r)$ where Σ is a finite alphabet, N is a set of nodes, E is a set of edges and $r \in N$ is a special node called root node (element). Each node has an associated type which can be element, attribute, text, processing instruction or comment. Nodes with type of either element or attribute have a node label l from the alphabet Σ which is called element or attribute name. For simplicity, we refer to nodes with types element or attribute simply as element or attribute.*

Sometimes we refer to tree T as XML tree.

Listing 1 contains a short example of an XML document. The root element is *restaurant_menu* which contains all other elements. Example of an attribute with the name *id* is on the element *menu_item*. Text content 'Hamburger' has element *name*, for example.

```xml
<?xml version="1.0"?>
<!DOCTYPE restaurant_menu SYSTEM "menu.dtd">
<restaurant_menu>
  <menu_item id="1">
    <name>Hamburger</name>
    <price>250 CZK</price>
    <description>
      Beef hamburger, served on crispy bun with fresh vegetables, Cheddar cheese
      and bacon, accompanied by home-made french fries.
    </description>
    <calories>1200</calories>
  </menu_item>
  <!-- Comment: other items were omitted -->
</restaurant_menu>
```

Listing 1: Example of an XML document

Figure 1.1 shows the simplified view of the corresponding XML tree for the XML document shown in Listing 1. It shows elements, comments and text contents.

Definition 2. A DTD (Document Type Definition) *is a collection of declarations in the form $e \to \alpha$ where $e \in \epsilon$ is the name of an element and α is its content model (which is a regular expression over ϵ). The content model α of an element e is*

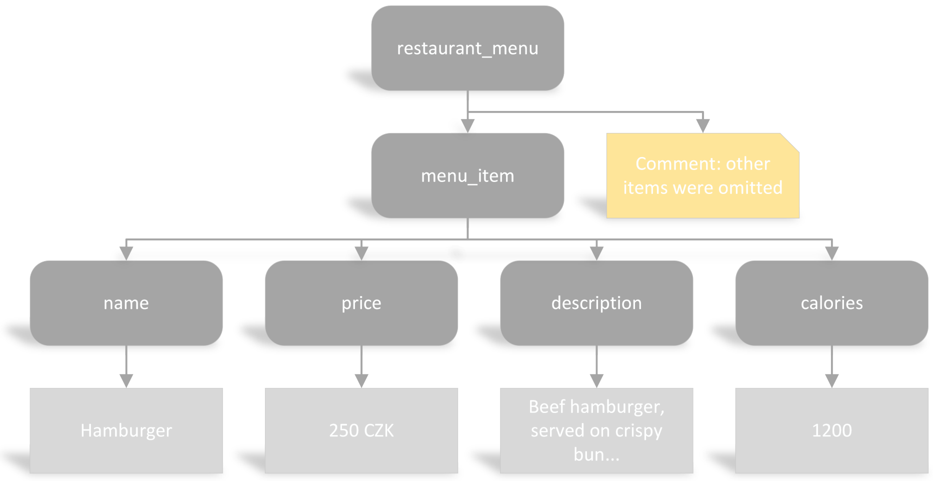

restaurant_menu
menu_item
Comment: other items were omitted
name
price
description
calories
Hamburger
250 CZK
Beef hamburger, served on crispy bun...
1200

Figure 1.1: XML tree example

$\alpha = \epsilon \mid text \mid f \mid (\alpha_1, \alpha_2, ..., \alpha_n) \mid (\alpha_1|\alpha_2|...|\alpha_n) \mid \beta^* \mid \beta+ \mid \beta?$. ϵ is empty content model, text is text content, f means single element, "," means concatenation of content models, "|" means union of content models. "", "+", "?" mean zero or more, one or more, optional occurrence (of content model β). One of the element names $s \in \epsilon$ is called* a start symbol.

Listing 2 shows the DTD for the XML document shown in Listing 1. There are two ways to reference a DTD in XML document. It is either by using the internal DTD declaration where DTD is inside the XML file or by using an external DTD declaration where the DTD is declared in an external file. Both ways are described in [9]. Our example shows the contents of the file menu.dtd which is referenced in XML document in Listing 1.

```
<!ELEMENT restaurant_menu (menu_item+)>
<!ELEMENT menu_item (name, price, description, calories)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT calories (#PCDATA)>

<!ATTLIST menu_item id ID #REQUIRED>
```

Listing 2: DTD example

Definition 3. *Content model α is called* mixed *when $\alpha = (\alpha_1 \mid ... \mid \alpha_n \mid text)^* \mid (\alpha_1 \mid ... \mid \alpha_n \mid text)+$ where $n \geq 1$ and for i: $1 \leq i \leq n$ content model $\alpha_i \neq \epsilon \vee \alpha_i \neq text$.*

An element is called mixed-content element *when its content model is mixed.*

There is an example of mixed-content element in Listing 3. Element *description* contains text and element *bold* with another text in it.

```xml
<restaurant_menu>
  <menu_item id="1">
    <!-- ... -->
    <description>
      Beef hamburger, served on <bold>crispy</bold> bun with fresh vegetables,
      Cheddar cheese and bacon, accompanied by home-made french fries.
    </description>
  </menu_item>
  <!-- ... -->
</restaurant_menu>
```

Listing 3: Example of element with mixed content

1.2 JSON and BSON

JSON [2] (JavaScript Object Notation) is a data-interchange format. It is easy to read and write for humans and also easy to parse and generate for computers. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 [73].

JSON is based on two basic structures: a collection of key-value pairs and an ordered list of values (array). These structures are universal and almost all modern programming languages support them in some form.

JSON supports the following basic types: `Number`, `String`, `Boolean`, `Array`, `Object` and `null`. Listing 4 shows a simple example of JSON document which consists of several fields (or keys). The type of values of the fields `first_name`, `last_name` and `email` is `String`. The type of the value of the `Address` field is `Object` and `favourite_colours` field has an `Array` value type.

```json
{
        "first_name": "John",
        "last_name": "Doe",
        "address": {
                "street": "High Street",
                "city": "Rome",
                "state": "Italy"
                },
        "email": "john.doe@email.com",
        "favourite_colours": ["blue", "red", "green"]
}
```

Listing 4: JSON example

JSON is promoted [5] as a low-overhead alternative to XML and XML documents can be converted into JSON (and vice-versa). Listing 5 shows the same structure as in Listing 4 expressed in XML.

BSON [74] (Binary JSON) is a binary-encoded serialization of JSON-like documents. BSON contains extensions that allow it to represent data types that are not a part of JSON (e.g. BSON has a `date` type and `BinData` type. Other types available in BSON include `string`, textttinteger (32- or 64-bit), `double`, `date`, `byte array` (`BinData`), `boolean`, `null`, `BSON object`, `BSON array`.

BSON is designed to have minimum spatial overhead which is useful for network transfers. However, BSON representation can sometimes be bigger than the

```xml
<person>
  <first_name>John</first_name>
  <last_name>Doe</last_name>
  <address>
    <street>High Street</street>
    <city>Rome</city>
    <state>Italy</state>
  </address>
  <email>john.doe@email.com</email>
  <favourite_colours>
    <colour>blue</colour>
    <colour>red</colour>
    <colour>green</colour>
  </favourite_colours>
</person>
```

Listing 5: JSON example in XML format

corresponding JSON file. This is due to another design goal which is traversability. BSON contains additional data like length of strings or sub-objects which makes traversal faster. This property is vital for databases like MongoDB [45] for which this is the storage and network transfer format. BSON is also designed to be fast to decode and encode.

Listing 6 shows very simple JSON document and the Listing 7 its BSON representation.

```json
{"hello": "world"}
```

Listing 6: JSON syntax

```
\x16\x00\x00\x00                   // total document size
\x02                               // 0x02 = type String
hello\x00                          // field name
\x06\x00\x00\x00world\x00          // field value
\x00                               // 0x00 = type EOO ('end of object')
```

Listing 7: BSON representation

2. Big Data Introduction

Big Data is a relatively new term that emerged around year 2000. There is no formal definition yet. This term is usually used for data sets that are so large that common software tools and methods are unable to work with them effectively in acceptable time and new tools and forms of processing are required. There is a proposal of the formal definition in [3] which defines Big Data as follows:

> *"Big Data represents the Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value."*

In terms of size Big Data currently ranges from terabytes to petabytes and even more (the upper bound is not restricted). However, this size increases from year to year. There are also techniques and technologies that allow us to dive into large, diverse and complex data sets, e.g. NoSQL databases, cloud computing etc. Examples of Big Data sources are various logs, data from sensors, various scientific data, social media data, medical records, surveillance data, data from e-commerce applications and many others. These types vary widely, however in all cases, this data holds valuable information for the owner of the data.

There are four basic characteristics of Big Data (4 V properties) [72]: volume, variety, velocity and veracity. Volume is the size of the data. This determines its value and potential and also whether it can be considered as Big Data. Variety of data deals with various formats, types, forms and structures of data. These can be from unstructured, semi-structured to structured data. Velocity of Big Data talks about the speed at which the data is being generated and processed. Veracity of Big Data deals with uncertainty in data. The quality of the data can vary greatly. This can be due to inconsistencies, ambiguities or incompleteness of the data which depends on the veracity of the source of the data.

The following sections describe basic database types used for Big Data and explain some additional Big Data terminology.

2.1 NoSQL Databases

Traditional technologies like relational database management systems are usually not able to handle the size and velocity of Big Data which resulted in new technologies for batch processing and interactive processing of such data.

One of these technologies are NoSQL databases which are used for interactive processing of very large data sets. It provides mechanisms for storage and retrieval of data that is modeled in other ways than traditional relations. There is no single formal definition and the definitions vary. It is often characterized [4] as next generation databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable (which means to add more nodes to a system, e.g. new computers to a distributed system where load is distributed among all the nodes). These systems usually do not provide full ACID (atomicity, consistency, isolation, durability) properties but provide distributed and fault-tolerant architecture.

There are several data models used in NoSQL systems. These are different in key-value, document, column-family and graph databases. The first three models are oriented on aggregates. An aggregate is a unit of related data with complex structure. This data we usually want to treat as one unit. Relational database management systems and graph databases are aggregate-ignorant. This allows different views on data easily but at the higher computational costs.

In the following paragraphs we will describe each category of NoSQL database types briefly.

2.1.1 Key-value Databases

This is the simplest type of NoSQL databases. It provides a simple hash table addressed via a primary key. We would model it with two columns, e.g. ID and VALUE in relational database. ID column stores the key and the VALUE contains the associated value. It supports only basic operations like getting the value for the specified key, inserting the value for a key and a deletion of such key-value pair. Its simplicity provides great performance and is highly scalable, however it does not support complex queries.

The representatives of this type of NoSQL database are Riak [40], Redis [41], MemcachedDB [42], Hamster DB [43], Project Voldemort [44] etc.

This type of database is commonly used for storing session information, user profiles and preferences, shopping cart data etc. It is fast, as it works with a single object that contains all the information.

2.1.2 Document Databases

The main concept of this type of NoSQL database are documents which are stored, retrieved and managed. Documents are usually in XML or JSON format and thus have hierarchical tree structures self-describing the contents. We usually expect them to be similar but not exactly the same (the schema might differ). Document databases contrast strongly with relational databases which are strongly typed during the database creation. Every instance of the data has the same format as the other and it is often difficult to change this. Document databases get the type information from the data itself. They store the related data together and allow every instance of the data to be different. This property makes it easier and more flexible to deal with changes.

The representatives of this type of NoSQL databases are MongoDB [45], RavenDB [47], CouchDB [46], OrientDB [48] etc.

Document databases are useful for event logging, content management systems, blogging platforms, e-commerce applications, real-time analytics and more.

2.1.3 Column-Family Databases

This type of database is column-oriented. Column family is a collection of similar rows. Each row is a collection of columns associated with a key, i.e. it is a tuple (key-value pair), where the key is mapped to a value that is a set of columns. Column is a basic unit which consists of a triple: name-value pair and a timestamp. Column families are groups of related data that is often accessed together. In relational databases, a column family would be a table, each key-value pair being a row. In relational databases, each row must have the same columns whereas in column-family databases, this is not the case.

The representatives of this type of NoSQL databases are Google Cloud Bigtable [49], Apache HBase [50], Apache Cassandra [51], Hypertable [52], Amazon SimpleDB [53] etc.

Column-family databases are suitable for event logging, content management systems, blogging platforms etc.

2.1.4 Graph Databases

Graph databases store entities and relationships between them. Basic concepts are nodes (instances of objects) and edges (correspond to relationships between nodes). Nodes have properties (like name) and edges have types (like 'is a friend of'). In a relational database we can model a single type of relationship whereas adding a new type of relationship requires many changes in existing schema. Nodes in graph databases can have many different types of relationships. The number of these relationships is not limited and relationships can be easily added or removed.

The representatives of this type of NoSQL database are Neo4j [54], Infinite-Graph [55], OrientDB [48], FlockDB [56], etc.

Graph databases are suitable for connected data like social networks. Another area where this type of database is useful are routing, dispatch and location-based services. They can also be used for recommendation engines.

2.2 Common Big Data Terminology

There are several important common terms used in the Big Data world. We also often refer to them in the rest of the thesis. The explanation of the most important ones is included in the following paragraphs.

2.2.1 Horizontal and Vertical Scaling

Large data sets and high query rates can exceed the capacity of a single machine, either the CPU, RAM or disk drives. There are two basic approaches to mitigating these issues: *vertical* and *horizontal* scaling.

Vertical scaling adds more CPUs, RAM and disk drives to increase the capacity of the system. However, this scaling has its limitation. Systems with many CPUs and large amounts of RAM are disproportionately more expensive than smaller systems. In practice, there is also a maximum capability for vertical scaling.

Horizontal scaling divides the data sets over multiple hosts. Each host is an independent physical database. All these hosts compose a logical database. Horizontal scaling reduces the amount of data each host has to store and also reduces the number of operations it has to handle. Thus it is possible to store and handle very large data sets using this approach. This is the reason why this method is used in databases that store large datasets. Horizontal scaling has also its disadvantages. For example adding many inexpensive computers to a cluster might mean additional licensing costs, additional power consumption, large footprint in datacenter, increased management complexity or issues with throughput and latency between the nodes.

2.2.2 Strong and Eventual Consistency

Large datasets often span over multiple nodes and are also replicated to ensure high availability of the system. Data modifications are a common scenario and desired property of a database system is for all data readers to see the same data. *Strong consistency* guarantees that after an update, all accesses to the changed value will return the same value. Achieving this property in practice, however, is difficult since there are many computers and network systems involved.

Eventual consistency [7] is a model of consistency used in distributed computing to achieve high availability. It informally guarantees that if there are no more updates to the given data item, eventually all accesses to it will return the last updated value. A system that has achieved this state is usually said to have converged or achieved replica convergence. Eventual consistency is a weak guarantee and is often criticized because it does not guarantee safety. System that is eventually consistent can return any value before it converges. However, these inconsistencies must be tolerated mainly for two reasons: the improvement of read and write performance and cases where a majority model makes the part of the system unavailable.

2.2.3 High Availability

High availability is a characteristic of a system or a component that is in operational state for a desirably long length of time. This property is is achieved through elimination of single points of failures, e.g. by adding redundancy. It is also necessary to ensure reliable crossover from the the failed entity to the operational one and that also means being able to detect the failure has occurred.

2.2.4 Automatic Scaling

Automatic scaling is the ability to add or remove capacity based on the actual usage without human intervention.

2.2.5 Sharding

Sharding is a database scalability technique that partitions very large data sets into smaller parts called *shards*. These smaller parts are usually located on independent database servers. These smaller parts are usually faster and more easily managed. One of the benefits if this technique is that is allows to scale out the database to potentially very large data sets.

2.2.6 Cluster

Cluster is a set of loosely or tightly connected computers working together. A cluster might often be viewed as a single system. The main reason a cluster is used is to improve the computational, data storage capacity of the system or to increase the availability.

2.2.7 Replication

Replication is an act of sharing information to ensure consistency between redundant resources (either software or hardware components). Replication is usually done to improve reliability, fault-tolerance or accessibility of the resources.

In database systems, the replication usually involves data synchronization across multiple servers (i.e. creates multiple copies of the same data). This provides redundancy and increases availability of the data. Replication protects database from the loss of a server and allows recovery from hardware failures or service interruptions. Sometimes it even increases read capacity.

A *replica* is one copy of the replicated resource, e.g. copy of the database.

3. Data Generation

As the amounts of data generated and captured daily increases, new challenges for processing such huge amounts of data arise. New technologies and products were developed to make it possible to work with these big and complex data sets. Naturally, it is necessary to have also systems that compare these software products. Big Data benchmarks for comparing and performance testing of these systems were developed, are still under development or researched.

In order to test the Big Data systems using benchmarks, it is often necessary to use data sets similar to those used in production systems. One of the options is to use real-world data sets as workload inputs. However, obtaining these data sets is often difficult due to confidentiality of the data. Another issue is that these data sets have a fixed size and thus it would be difficult to alter it for various types of test scenarios. One of the problems is also how to transfer such huge volumes of data via the Internet (downloading terabytes, petabytes or even bigger volume of data might take quite a long time).

The obvious idea to solve this problem is to use a synthetically generated data based on real-world data. Optimal data generator should keep the 4 V properties of Big Data: (1) high volume of data and various (2) velocities can be generated; (3) different varieties of data sources should be supported and the (4) veracity of the original data should be preserved.

Currently there are two Big Data generators (frameworks) which we will describe in the following sections. The first one is called Parallel Data Generation Framework (PDGF) [57] and the second one is called Big Data Generator Suite (BDGS) [58].

We focus on document databases in this thesis, therefore we will also include a description of several JSON and XML data generators.

3.1 Parallel Data Generation Framework

The Parallel Data Generation Framework (PDGF) [57] is a flexible, generic data generator capable of generating large amounts of relational data. It is implemented in Java and is fully platform-independent. PDGF uses parallel random number generation for independent data generation which enables parallel data generation needed for large data sets.

PDGF was originally built for relational data. However, it contains a post-processing module that enables mappings to other data formats like XML. PDGF was used as data generator in the BigBench big data analytics benchmark [60]. PDGF was used for data generation of semi- and unstructured data sources.

The data is specified in two XML configuration files: the schema configuration and generation configuration. The schema configuration file specifies the schema of the generated file similar to the relational schema. The generation configuration contains additional post-processing options like data formatting, table merging and splitting, etc.

3.2 Big Data Generator Suite

Big Data Generator Suite (BDGS) [58] is a scalable data generator suite which contains six data generators for three data types (structured, semi-structured and unstructured) and three data sources (text, graph and table data).

BDGS preserves the 4 V properties of the data. Data generators in BDGS are designed for various application classes (search engine, e-commerce, social network). BDGS is implemented as a component of the BigDataBench [61], which is an open-source Big Data benchmarking project.

4 V properties of the generated data are preserved in the following manner: (1) the volume of the data can be adjusted via a configuration of each data generator; (2) different velocities can be achieved by deploying various numbers of parallel data generators; (3) different kinds of data generators produce a variety of data sets which have different types and sources; (4) synthetic data is generated based on real-world data sets which preserves the data veracity.

Data generation in BDGS uses small real-world data sets. The Wikipedia entries, Google Web Graph, Facebook Social Graph are used for unstructured data. E-commerce transaction data is used for structured data. Amazon Movie Reviews and Personal Resumes are used for semi-structured data.

The generator uses fixed data sources and consists of multiple data generators each of which has different properties and usage. There are six data generators implemented in BDGS, each belonging to one of the following types: Text Generator, Graph Generator and Table Generator. All the data generators from BDGS are written in C++ for Linux environment.

3.3 DataGenerator

This data generator [59] is a Java library for systematically producing large volumes of data. This framework is designed to produce large data sets very quickly (TB in minutes according to their website). It is designed to help with the generation of patterns based on a given model. It is not a random data generator. It supports parallel execution either locally (using separate threads) or it can be distributed to multiple nodes.

There are several steps during the data generation. First, it is necessary to create a model for the data generation. Then it is necessary to write Java application that calls the library which loads the model, executes the generation and uses a writer for the generated patterns.

The data model is expressed in SCXML [84] (State Chart eXtensible Markup Language). The data is represented as states which can set output variables to certain values. The transitions optionally contain conditions where the user can control the data values.

3.4 JSON Data Generators

Document databases usually use JSON data format for the documents. We reviewed several data generators capable of creating synthetic JSON documents.

Most of them are online tools so their usage is limited to manual interaction. We are going to briefly describe the following generators:

- json-generator.com [62]
- generatedata.com [63]
- mockaroo.com [64]
- MongoDB-Datasets [65]

3.4.1 json-generator.com

This web application [62] is a personal project of Vazha Omanashvili. It allows the user to specify a template using special tags and this template is transformed into JSON document. This template must contain a valid JavaScript. The tool provides several template tags for various data types, e.g. integer, floating, bool, date, lorem, company, country, street, etc. It is possible to generate even larger files, these can be downloaded from the web page. The application also contains useful manual for each of the templates so writing a custom template is easy.

An example of the template is shown in Listing 8. The resulting document is shown in Listing 9.

This application is not open source and it is currently not possible to deploy it to another server and extend or modify it.

3.4.2 generatedata.com

This is another web application [63] for data generation. It can create various output types, whereas JSON is one of the options. Other options include CSV, Excel, HTML, SQL, XML and others. The data is generated using a table where it is possible to specify columns and their data types. There are several built-in data types the user can choose from. The limit of this application is that it is not possible to nest the records and only flat structure can be created (like in relational databases).

The online application is a demo version of the application that can be downloaded and deployed to custom server. It is a script written in JavaScript, PHP and MySQL.

3.4.3 Mockaroo

This web application [64] is similar to generatedata.com in the sense that it is capable of generating only flat structure. Its primary function is to generate realistic data for relational databases. It also contains various custom data types (more than 80) and supports generation of strings based on regular expressions. This tool can generate various output formats as well, e.g. CSV, SQL, JSON, DBUnit XML. The web application has a public REST API which can be called for data generation. This service is limited to 200 requests per day for free version, more requests per day are payed.

```
[
  '{{repeat(1, 1)}}',
  {
    _id: '{{objectId()}}',
    index: '{{index()}}',
    name: '{{firstName()}} {{surname()}}',
    gender: '{{gender()}}',
    email: '{{email()}}',
    phone: '+1 {{phone()}}',
    address: '{{integer(100, 999)}} {{street()}}, {{city()}}, {{state()}},
    {{integer(100, 10000)}}',
    about: '{{lorem(1, "paragraphs")}}',
    tags: [
      '{{repeat(7)}}',
      '{{lorem(1, "words")}}'
    ],
    friends: [
      '{{repeat(3)}}',
      {
        id: '{{index()}}',
        name: '{{firstName()}} {{surname()}}'
      }
    ],
    greeting: function (tags) {
      return 'Hello, ' + this.name + '! You have ' + tags.integer(1, 10)
      + ' unread messages.';
    },
    favoriteFruit: function (tags) {
      var fruits = ['apple', 'banana', 'strawberry'];
      return fruits[tags.integer(0, fruits.length - 1)];
    }
  }
]
```

Listing 8: JSON Generator sample

3.4.4 MongoDB-Datasets

MongoDB-Datasets [65] is an open source Node.js [66] application designed to generate data based on a template (which is similar to the template used in json-generator.com). Node.js is a platform built on V8 JavaScript Engine [67]. MongoDB-Datasets is therefore written in JavaScript and exploits this in its templates. The template is a JSON document with the parts written in JavaScript. Simple example is shown in Listing 10. This template defines 4 fields to be generated. The values for those fields can be any primitive data types (boolean, number, array, object). When the value is a string (as in our example), it is possible to specify a JavaScript expression which gets evaluated during the data generation. This expression is surrounded by double curly braces: `{{ expression }}`. Random data generation is achieved using additional libraries that produce random data: change.js [68] and faker.js [69].

The example in Listing 10 uses built-in `counter()` function for the `_id` field which increases its value every time it is evaluated. The next two fields use functions from the Chance library and the last one uses built-in function to sample a value from the enumerated values. Other options and capabilities of the templates

```
[
  {
    "_id": "5592dcd636f9a1dc697702e7",
    "index": 0,
    "name": "Cruz Wynn",
    "gender": "male",
    "email": "cruzwynn@imkan.com",
    "phone": "+1 (989) 401-2133",
    "address": "109 Clifford Place, Belgreen, Marshall Islands, 2271",
    "about": "Consequat ea cillum cupidatat Lorem aliquip reprehenderit
    nisi officia veniam id dolor commodo. Qui occaecat officia tempor
    sit dolor fugiat velit deserunt et eu magna culpa aliqua proident.\r\n",
    "tags": [
      "officia",
      "ex",
      "in",
      "anim",
      "ullamco",
      "laborum",
      "non"
    ],
    "friends": [
      {
        "id": 0,
        "name": "Colon Wilcox"
      },
      {
        "id": 1,
        "name": "Jenkins Rice"
      },
      {
        "id": 2,
        "name": "Nielsen Malone"
      }
    ],
    "greeting": "Hello, Cruz Wynn! You have 1 unread messages.",
    "favoriteFruit": "banana"
  }
]
```

Listing 9: JSON Generator result

can be found in [65].

```
{
  "_id": "{{counter()}}",
  "name": "{{chance.name()}}",
  "phones": [ 3, "{{chance.phone()}}" ],
  "title": "Software {{util.sample(['Engineer', 'Programmer'])}}"
}
```

Listing 10: Simple template for MongoDB-Datasets generator

MongoDB-Datasets is invoked from the command line and can generate any number of documents based on the supplied template. These documents can be saved to a file or printed to standard output (or piped to other application).

This data generator is very similar to json-generator.com. Its big advantage is that it is possible to invoke it locally as opposed to json-generator.com. This generator is therefore more suitable for testing with large data sets. However, in its current implementation, there are also some drawbacks. It is not multithreaded which limits its usage on multiprocessor environments. Another drawback is that the implementation outputs generated documents inside a JSON array either to a file or to a standard output and not to MongoDB directly. Therefore, it is necessary to alter the application itself or build another application which would use MongoDB-Datasets as a module.

3.4.5 Summary of Big Data and JSON Data Generators

PDGF [57] data generator focuses on relational data, requires two configuration files and can generate large amounts of data. Its advantage is also that it is platform-independent. BDGS [58] data generator focuses on selected specific use cases and uses six different data generators for specific types of data. It preserves the 4V properties of the generated data as well. The last described big data generator [59] is a general Java framework which requires additional programming which makes it the least usable despite its speed, in our opinion.

JSON data generators currently known to us can be divided into two groups: web applications and a standalone application. Web applications are not suitable for the generation of large data sets. They can be used only for generation of small sample documents. This makes them unusable for Big Data generation. Applications belonging to this category are json-generator.com [62], generate-data.com [63] and mockaroo.com [64]. MongoDB-Datasets [65] is a standalone application and can be used for generation of large data sets only if there is additional work done (e.g. changes in its source code to support multithreading, additional applications for storage of generated data etc.). The usage pattern of these generators (i.e. templates) is, in our opinion, simple and effective.

Selected properties of the described data generators are compared in Table 3.1.

	PDGF	BDGS	Data Generator	json-generator.com	generatedata.com	mockaroo.com	MongoDB-Datasets
Source code available	Not to our knowledge	yes	yes	no	yes	no	yes
Programming language	Java	C++	Java	unknown	PHP and JavaScript	unknown	JavaScript
User interaction mode	Unknown	command line	depends on the application	web application	web application	web application	command line
Input parameters	2 XML configuration files	depends on the data generator, usually configuration file	SCXML model	JSON template	Data set specification in the web interface	Data set specification in the web interface	JSON template
Primary focus	Generation of large data sets of relational data	Generation of large data sets while preserving the 4 V properties	Producing large volumes of data	Generation of sample JSON documents	Generation of test data in various formats	Generation of realistic test data	Generation of test data for MongoDB

Table 3.1: Summary of Big Data and JSON data generators

3.5 XML Data Generators

There are various XML data generators available at the present moment. Some are simple and some are quite complex. Many of them are a part of a more complex solution (for instance many XML benchmarks contain their own XML data generator).

We found several data generators which have different properties and usage. As the first step, we will have a closer look at a data generator *xmlgen* which is a part of the *XMark benchmark* [12]. The next one will be the *genxml* tool from the *XOO7 benchmark* [17]. Then we will describe the *XMach1 generator* [21], the generator used in the *Michigan benchmark* [22], the data generator from the *ToXgene* [27]. Then we will describe a solution called *Complex-structured XML data generator* [31]. The next group of data generators are generators found in different editors or applications whose primary function is not XML data generation. These applications are <oXygen/> XML Editor [32], Liquid XML Studio 2013 [34], Altova XMLSpy [36], Eclipse [38] and Microsoft Visual Studio [39].

Every section briefly describes the data generator (or benchmark, if applicable) and for each of the generators there is a brief summary which answers the following questions:

- Is there a source code available for the XML data generator?
- What is the programming language of the data generator? What are the platforms for which this program is available?
- It is a command line program or does it have some kind of user interface?
- How many input parameters are there? Is it easy to use?
- What is the primary focus of the data generator?
- Is there a predefined schema for the output XML documents?
- Is it possible to alter this schema?
- Is it possible to scale the resulting XML documents in a simple manner?

After this summary we add also notes specific to the current data generator, if appropriate.

3.5.1 XMark Benchmark

XMark benchmark [12] provides a framework to test the abilities of an XML database to cope with different query types that are typical in real-world scenarios. This benchmark aims to help both the implementors and users to compare various XML databases. The benchmark also offers a set of queries where each query targets a different part of the XML query processor. There are 20 XQuery [14] queries [12] which focus on various parts of XML query processor, e.g. exact match, ordered access, casting, regular expressions, aggregations, joins, sorting etc. This XML benchmark belongs to one of the most commonly used XML benchmarks [13].

Data Generator from XMark Benchmark

The XML data generator called *xmlgen* is very simple, with very few options to change (the following paragraph describes all the options it offers). It produces

XML document(s) with predefined content (which was set by its authors). The DTD that can be generated by *xmlgen* (or downloaded here [16]) describes an Internet auction. Complete DTD is shown in Listing 52 in Appendix B. The text nodes are generated from the most frequently occurring words in the works of Shakespeare. The default document size is about 150MB (when scaling factor is set to 1).

According to the website of the data generator [15] the main features of the data generator are the following:

- Generation of well-formed, valid and meaningful XML data.
- Efficient, scalable generation of XML documents the size of several GB.
- Observing of referential constraints concerning ID/IDREF pairs.
- Low, constant memory requirements, independent of the size of the generated document.

XML Data Generator Parameters

The data generator has the following options which affect the generated document:

- *scaling factor of the document*: 0 - produces the minimal file, 1 - produces the default size (about 150MB), it can be set to any float number
- *name of the output file*
- *whether to use doctype preamble*
- *number of elements a single file should contain* - it can split the document into smaller chunks

Other options are for debugging or profiling purposes. The generator is available as a binary for Windows, Linux, Solaris and IRIX platforms.

Data Generator Summary

- The source code is available.
- It is written in C, available for Windows, Linux, Solaris, IRIX.
- It is a command line program.
- The usage is very simple, there are only four input parameters and they all are easy to understand.
- The generator focuses on the generation of well-formed, valid and meaningful XML data while keeping low memory requirements even for large documents.
- The DTD for the output documents is available.
- It is not possible to alter the schema of the output documents (without altering the source code of the application).
- It is possible to scale the output documents via input parameter.

This data generator is simple; the user cannot change the structure of the generated data which simplifies the process a lot. Basically the only thing a user can affect is the size of the document. This data generator is therefore not very useful for usage scenarios where a different structure is needed or desired. Despite this fact it is one of the commonly used [13].

3.5.2 XOO7 Benchmark

XOO7 benchmark [17] is an XML version of original OO7 benchmark [19] for object-oriented database management systems. It was enriched with relational, document and navigational queries that are specific and critical for XML databases. The data generator called *genxml* which is part of the benchmark is able to generate data sets for the benchmark according to the user-defined parameters. The authors of this benchmark proposed three types of data sets (small, medium, large) with predefined values for the parameters of the data generator. This benchmark also contains 18 XQuery queries [18] divided into three groups: traditional database queries (joins, aggregation, sorting, etc.), navigational queries (exploiting references and links) and document queries (focusing on element ordering).

XML Data Generator from XOO7 Benchmark

The XML data generator *genxml* used in the XOO7 benchmark is slightly more complex than the data generator from the XMark benchmark. There is a configuration file which controls several aspects of the generated XML documents, such as depth of the document tree, fan-out or amount of textual data. This generator also uses a predefined DTD [20], its complete listing is shown in Listing 11.

XML Data Generator Parameters The parameters in the configuration file are related to the structure of the generated document. The whole description of the generated data and each of the elements mentioned in the configuration file is in [18]. The configuration file contains the following parameters:

- *NummAssmPerAssm:* specifies the number of assembly elements inside an assembly element (either Complex or Base)
- *NumCompPerAssm:* specifies the number of Composite elements inside the assembly element
- *NumCompPerModule:* specifies the number of unique CompositePart elements per Module
- *NumAssmLevels:* specifies the number of nested assembly elements (there are NumAssmLevels - 1 levels of ComplexAssembly elements and then one BaseAssembly element)
- *NumAtomicPerComp:* specifies the number of unique Atomic elements per CompositePart
- *NumConnPerAtomic:* specifies the number of connections per Atomic element (the total number of connections inside the CompositePart element is equal to NumAtomicPerComp * NumConnPerAtomic)
- *DocumentSize:* length of the contents of the Document element inside the CompositePart element
- *ManualSize:* length of the contents of the Manual element

It is possible to control the number of elements at various levels inside the document tree which then also influences the total document size. Textual content stays always the same, only the length changes according to the set parameters.

```
<!ELEMENT Module   (Manual, ComplexAssembly)>
<!ATTLIST Module  MyID    NMTOKEN  #REQUIRED
      type     CDATA    #REQUIRED
      buildDate   NMTOKEN    #REQUIRED>
<!ELEMENT Manual   (#PCDATA)>
<!ATTLIST Manual  MyID     NMTOKEN    #REQUIRED
      title     CDATA    #REQUIRED
      textLen   NMTOKEN    #REQUIRED>
<!ELEMENT ComplexAssembly (ComplexAssembly+ | BaseAssembly+)>
<!ATTLIST ComplexAssembly
      MyID     NMTOKEN    #REQUIRED
      type     CDATA    #REQUIRED
      buildDate   NMTOKEN    #REQUIRED>
<!ELEMENT BaseAssembly (CompositePart+)>
<!ATTLIST BaseAssembly
      MyID     NMTOKEN    #REQUIRED
      type     CDATA    #REQUIRED
      buildDate   NMTOKEN    #REQUIRED>
<!ELEMENT CompositePart (Document, Connection+)>
<!ATTLIST CompositePart  MyID     NMTOKEN    #REQUIRED
      type     CDATA    #REQUIRED
      buildDate   NMTOKEN    #REQUIRED>
<!ELEMENT Document   (#PCDATA | para)+>
<!ATTLIST Document  MyID     NMTOKEN    #REQUIRED
      title     CDATA    #REQUIRED>
<!ELEMENT para     (#PCDATA)>
<!ELEMENT Connection   (AtomicPart, AtomicPart)>
<!ATTLIST Connection  type     CDATA    #REQUIRED
      length     NMTOKEN    #REQUIRED>
<!ELEMENT AtomicPart   EMPTY>
<!ATTLIST AtomicPart  MyID     NMTOKEN    #REQUIRED
      type     CDATA    #REQUIRED
      buildDate  NMTOKEN    #REQUIRED
      x     NMTOKEN    #REQUIRED
      y     NMTOKEN    #REQUIRED
      docId     NMTOKEN    #REQUIRED>
```

Listing 11: XOO7 Data Generator DTD

Runtime parameters specify the name of the configuration file, the name of the output file and the number of the Module elements to generate (it appends this number to the root element and also sets it as an id of this element).

The data generator is provided as a source code written in C. In order to run it on an OS X 10.8.2 we had to modify the source code, mostly correcting the compilation errors which originated due to the age of the code. After these corrections were done, we were able to run the data generator successfully.

Data Generator Summary

- The source code is available.
- It is written in C, it should be possible to run it on any platform with C runtime.
- It is a command line program.
- There are eight input parameters in the configuration file, quite confusing. We did not find the proper documentation for these parameters.

- It seems that this generator is tailored only for the benchmark itself with no further usage.
- DTD of the output documents is available.
- It is not possible to alter the schema of the output documents (without altering the source code itself).
- It is possible to scale the output documents via the configuration file (although there is not one single argument for this purpose).

3.5.3 XMach-1 Benchmark

Xmach-1 [21] is a multi-user benchmark focused on evaluating the performance of XML data management systems. The previously described benchmarks assumed single-user approach. The benchmark allows to identify advantages and shortcomings of various XML data management approaches.

The benchmark is based on a web-oriented usage scenario (idea of a web application). It has four parts: XML database, application servers, loaders and browser clients. The application servers interact with the backend XML database and serve to process XML documents. The loaders load and alter various XML data in the database using the application servers. Browser clients retrieve and query the stored XML data.

This benchmark contains a data generator and XQuery queries too. This data generator can produce schema-based documents (where the documents conform to a pre-defined DTD) or the schema-less documents (however the structure of the XML documents is the same, the difference is that the DTD is not stored in the XML database). There is also a directory of the stored XML documents.

There are 8 XQuery queries in this benchmark and also 3 operations that manipulate with the data. The queries are similar to the ones found in previous benchmarks, e.g. text retrieval query, navigation through document tree, sorting, counting, joins etc. Data manipulation involves inserting a document to the database, deleting it from the database and update of the information in the directory entry of stored documents.

The XML database is firstly populated with an initial number of documents and then the benchmark is executed.

XML Data Generator from XMach-1 Benchmark

This XML data generator is also quite simple. The generated documents are valid against a predefined DTD which is shown in Listing 12. As we mentioned earlier, even the schema-less documents conform to this DTD.

The structure of the generated XML document represents a document with chapters, sections, paragraphs and so on. We will discuss the various parameters that can be set to alter the generated output in the following section.

Every document in the benchmark has a unique URL and document id. URL consists of three host elements and one to four path elements. The result conforms to the following regular expression:

```
ahost{1-3}. bhost{1-(N/100)}. chost{1-5}/[ apath{1-3}/
[ bpath{1-3}/[ cpath{1-3}]]] NAME
```

```
<!ELEMENT documentXX (titleXX, chapterXX+) >
<!ATTLIST documentXX
  author CDATA #IMPLIED
  doc_id ID    #IMPLIED >
<!ELEMENT titleXX    (#PCDATA) >
<!ELEMENT chapterXX  (author?, headXX, sectionXX+) >
<!ATTLIST chapterXX
    id     ID    #REQUIRED >
<!ELEMENT author     (#PCDATA) >
<!ELEMENT sectionXX  (headXX, paragraph+, sectionXX*) >
<!ATTLIST sectionXX
    id     ID    #REQUIRED >
<!ELEMENT headXX     (#PCDATA) >
<!ELEMENT paragraph  (#PCDATA | link)* >
<!ELEMENT link       EMPTY >
<!ATTLIST link
    xlink:type (simple)  #FIXED "simple"
    xlink:href CDATA     #REQUIRED>
```

Listing 12: XMach-1 Data Generator DTD

Curly braces contain range numbers, square brackets denote optional parts, '.' and '/' are separators. N is the initial number of documents in the database.

Textual data comes from a list of the 10,000 most common English words. These words are chosen randomly using the Zipf's law [28] and appended together.

XML Data Generator Parameters The configuration is specified using a configuration file. There are the following parameters:

- *number of sections per document*: 5 - 150
- *number of paragraphs per section*: 1 - 15
- *number of sentences per paragraph*: 2 - 30
- *number of words per sentence*: 3 - 30
- *probability of having an author attribute/element*: 0.5
- *number of words per head or title element*: 2 - 12
- *probability of having a phrase within a sentence*: 0.01
- *probability of having a link element within a paragraph*: 0.05
- *number of documents per DTD*: 2 - 100

Data Generator Summary

- The source code is available.
- It is written in Java, available for any platform capable of running Java.
- It is a command line program.
- There are nine input parameters which are quite simple and easy to understand.
- The XML generator focuses on generating the document which has a predetermined structure.
- DTD is available.
- Is it not possible to alter the schema of the output documents (without altering the source code of the application).

- Is it possible to scale the resulting documents via input parameters.

This generator seems to have the same single purpose usage as previously described generators. It is possible to change the various aspects of the structure, however it is not possible to change the overall structure (there is always a document which has paragraphs which has sentences etc.). This limits its usage in general.

3.5.4 Michigan Benchmark

The Michigan Benchmark [22] represents a different kind of benchmark. The previous benchmarks and XML data generators focused on a specific application or domain whereas the Michigan Benchmark is a so called *micro-benchmark*. Its purpose is to test the impact of the basic query operations (selections, joins, aggregations etc.) on the performance of the tested XML systems. The previously mentioned benchmarks provide a good measure of the tested system performance against data sets (and queries) in their target XML application. However, it is difficult to use these results for a different domains and applications. This benchmark tries to overcome this shortcoming.

This benchmark consists of a data set and a set of queries. The data set is created according to an XML Schema (XSD) which is shown in Listing 13. The selection queries are divided into 4 groups [23]: simple selection queries, value-based join queries, pointer-based join queries and aggregate queries. The rest of the queries consists of update queries (insertion, deletion, load, etc.).

XML Data Generator from Michigan Benchmark

XML data generator in the Michigan Benchmark generates a data set which focuses on two structural parameters important to tree structures: depth and fan-out. The approach used in the data generator creates one big data set that contains various combinations of these two parameters at different levels of the tree. This way the benchmark can be run on a single document or on a part of this document and these results can be then compared. The basic data set has the depth of 16 and the fan-out differs at each level according to predefined parameters (details can be found in [24]).

The document scaling in this generator is achieved by keeping the tree-depth constant for all scaled versions, the difference is in the fan-out of the nodes at a few specific levels (levels 5-8).

The schema of the generated data is specified in XSD shown in Listing 13.

String Data Generation An interesting point about this data generator is the way how it generates its string attributes and elements. The data generator uses synthetically generated strings. It creates a pool of a specific number of synthetic words which are then divided into 16 buckets (with exponentially growing bucket occupancy). That means that the bucket i has $2i$ words. The words contain information about the bucket to which they belong and the word number in the bucket. As an example, "15twentynineB14" means that this is the 1,529th word from the 14th bucket. The last bucket contains words derived from words

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.eecs.umich.edu/db/mbench/bm.xsd"
  xmlns="http://www.eecs.umich.edu/db/mbench/bm.xsd"
  elementFormDefault="qualified">
  <xsd:complexType name="BaseType" mixed="true">
    <xsd:sequence>
      <xsd:element name="eNest" type="BaseType" maxOccurs="unbounded">
        <xsd:key name="aU1PK">
          <xsd:selector xpath=".//eNest"/>
          <xsd:field xpath="@aUnique1"/>
        </xsd:key>
        <xsd:unique name="aU2">
          <xsd:selector xpath=".//eNest"/>
          <xsd:field xpath="@aUnique2"/>
        </xsd:unique>
      </xsd:element>
      <xsd:element name="eOccasional" type="OccasionalType" minOccurs="0">
        <xsd:keyref name="aU1FK" refer="aU1PK">
          <xsd:selector xpath="../eOccasional"/>
          <xsd:field xpath="@aRef"/>
        </xsd:keyref>
      </xsd:element>
    </xsd:sequence>
    <xsd:attributeGroup ref="BaseTypeAttrs"/>
  </xsd:complexType>
  <xsd:complexType name="OccassionalType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="aRef" type="xsd:integer" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <xsd:attributeGroup name="BaseTypeAttrs">
    <xsd:attribute name="aUnique1" type="xsd:integer" use="required"/>
    <xsd:attribute name="aUnique2" type="xsd:integer" use="required"/>
    <xsd:attribute name="aLevel" type="xsd:integer" use="required"/>
    <xsd:attribute name="aFour" type="xsd:integer" use="required"/>
    <xsd:attribute name="aSixteen" type="xsd:integer" use="required"/>
    <xsd:attribute name="aSixtyFour" type="xsd:integer" use="required"/>
    <xsd:attribute name="aString" type="xsd:string" use="required"/ >
  </xsd:attributeGroup>
</xsd:schema>
```

Listing 13: XML Schema of the Michigan Benchmark Data Set

in other buckets by adding the suffix "ing" so the entire vocabulary is kept at roughly 30,000 words. The whole algorithm is described in more detail in [25].

XML Data Generator Parameters The data generator is available as a source code for Unix and Windows. The program is written in C++, however it is already outdated. When we tried to compile this data generator, we encountered compilation errors which had to be corrected. The main problem was the usage of old standard libraries. After correcting these errors we were able to run the data generator and test it. The program options are as follows [26]:

- **-sf=scale factor:** Valid scale factors are {0.1, 1, 10, 100}, [$default = 1$]
- **-n=doc name:** Set document file name, [$default =' doc'$]
- **-s | -S:** Turn off schema support [$default = on$]
- **-c | -C:** Turn off element content printed out [$default = on$]
- **-d | -D:** Turn on DTD support [$default = off$]
- **-v | -V:** Turn on verbose message printing [$default = off$]
- **-h | -H:** Print available options

Data Generator Summary

- The source code is available.
- It is written in C++, available for UNIX and Windows.
- It is a command line program.
- Low number of input parameters and those are easy to understand.
- Focuses on two structural parameters: depth and fan-out.
- XSD is available.
- It is not possible to alter the schema of the output XML document (without altering the source code).
- It is possible to scale the resulting XML document via the input parameter.

This data generator produces a bit different data when compared to the data generated from the previously described generators. It contains subtrees with different properties (mainly fan-out) which can be used as separate documents for testing. On the other hand, the names of the elements in the document are predefined and user cannot change this.

3.5.5 ToXgene Data Generator

ToXgene is a template-based generator for large, consistent collections of XML documents. This generator was developed as a part of the *ToX* [29] project. ToX is a heterogeneous repository for XML data and metadata being developed at the Database Group of the University of Toronto. ToXgene was developed to enable its users to create data fast and to produce quite complex XML content. The main features [27] are:

- Generation of complex XML content
- Use of skewed distributions
- Element sharing
- Integrity constraints
- Modularity
- Reuse of existing data
- Extensibility
- Scalability

This generator uses its own language for template specification called *TSL* (Template Specification Language). TSL is a subset of XML Schema [10] notation with annotations for specifying certain properties of the data to be generated (e.g. value distributions, CDATA vocabulary etc.). This generator is much more complex than any other generator we described so far.

ToXgene Template Specification Language

The basis for this language is XML Schema. It is enriched with different annotations which specify how to generate documents, special types, elements, probability distributions of occurrences of elements and attributes. They also define element sharing, integrity constraints and some randomness in the structure of the data.

Basic blocks of TSL are types and genes [30]. A *type* specifies a valid XML content (either a simple or complex type), an example can be seen in Listing 15. There is a declaration of the *departmentType* which consists of element department. This element contains one child element *classes* and two attributes (*name* and *employees*). Values of these attribute are generated using a gibberish string generator and exponential distribution generator respectively.

A *gene* is a specification of either an element (an *element gene*) or an attribute (an *attribute gene*) and contains a name and a type. In Listing 15 the specification of the element department is an element gene. The specification of the attribute *name* is an attribute gene.

Some randomness in generated data can be brought in by the probability distributions. These can be specified by separate annotations and referenced later via their names. These distributions can be used to determine the number of elements and attributes, the length of strings and to generate numbers. ToXgene now supports uniform, normal, exponential and log-normal distributions. It also supports user-defined discrete distributions, where the user provides all the possibilities with their respective probabilities. An example in Listing 14) shows the declaration of the exponential distribution which can be used for generation of numerical values. This generator is referenced in Listing 15.

Element sharing is another interesting feature of ToXgene. It is possible to share elements within or across documents. This feature can be used to generate XML documents that share the same contents and can be joined by these values later. Element sharing is achieved by using the tox-lists (list declaration can be seen in Listing 16 and its usage in Listing 17). This is a special annotation which specifies shared data. These lists can be then queried using another annotation.

Irregular structures can be specified using the *if-then-else* statements or using a "lottery" statement (an example of tox-alternatives is shown in Listing 18). Another way to generate irregular structure is to use element recursion in the genes.

```
<tox-distribution name="d1" type="exponential" mean="4" minInclusive="2"
  maxInclusive="10" />
```

Listing 14: Distribution Declaration in TSL

Extending ToXgene

ToXgene was created with extensibility in mind. It is possible to extend it by providing custom simpleType generators. There is an interface that has to be implemented. It can also be coupled with external tools to read and store lists from files.

```xml
<complexType name="departmentType">
  <element name="department" minOccurs="5" maxOccurs="15">
    <complexType>
      <element name="classes" type="classType"/>
      <attribute name="name">
        <simpleType name="nameType">
          <restriction base="string">
            <tox-string type="gibberish" maxLength="50" />
          </restriction>
        </simpleType>
      </attribute>
      <attribute name="employees">
        <simpleType name="employeesType">
          <restriction base="float">
            <tox-number tox-distribution="d1" />
          </restriction>
        </simpleType>
      </attribute>
    </complexType>
  </element>
</complexType>
```

Listing 15: Type Declaration in TSL

```xml
<tox-list name="courseList">
  <element name="course" minOccurs="20" maxOccurs="20">
    <complexType>
      <element name="id" type="idType" />
      <element name="title">
        <simpleType>
          <restriction base="string">
            <tox-string type="text" minLength="10" maxLength="20" />
          </restriction>
        </simpleType>
      </element>
    </complexType>
  </element>
</tox-list>
```

Listing 16: List Declaration in TSL

Data Generator Summary

- The source code is not publicly available.
- It is written in Java.
- It is a command line program.
- There are several input parameters, however they are intuitive. The difficult part is in writing the TSL template.
- It focuses on generation of large, consistent collections of synthetic XML documents.
- The schema is set by user in the TSL template.
- Altering the schema is possible, simply by editing the template.
- Document scaling must be done inside the template.

```
<complexType name="classType">
  <element name="class" minOccurs="1" maxOccurs="20"
    tox-distribution="classesDistribution">
    <complexType>
      <tox-scan path="[courseList/course]" name="a">
        <element name="title" type="string">
          <tox-expr value="[$a/title]" />
        </element>
        <element name="id" type="string">
          <tox-expr value="[$a/id]" />
        </element>
      </tox-scan>
    </complexType>
  </element>
</complexType>
```

Listing 17: List Usage in TSL

```
<tox-alternatives>
  <tox-option odds="70">
    <element name="required">
      <simpleType>
        <restriction base="string">
          <tox-value>true</tox-value>
        </restriction>
      </simpleType>
    </element>
  </tox-option>
  <tox-option odds="10">
    <element name="optional">
      <simpleType>
        <restriction base="string">
          <tox-value>true</tox-value>
        </restriction>
      </simpleType>
    </element>
  </tox-option>
</tox-alternatives>
```

Listing 18: Random Structures in TSL

This data generator overcomes all the shortcomings of the previously described generators. It is very general and can produce almost any structure the user wants. It is possible to create very simple structures and also very complex ones. Creating the TSL template, however, is not very easy and it can take some time to learn the language itself. We encountered some problems when debugging the template. After some experimenting we were able to create a simple template which produced an XML document we wanted.

3.5.6 Complex-Structured XML Data Generator

Paper [31] describes an XML data generator which has several input parameters affecting the resulting document. As the author states, the main purpose of this approach is to generate XML data sets which can have widely varying characteristics by changing the input parameters.

This generator can be used to generate synthetic XML data that resembles real data, however this is not its goal. The XML documents are created by generating a tree, a so-called path tree, which represents the structure of the data. The generator then assigns element names to this tree and specifies the frequency distribution of the XML elements. This information is then used to generate the resulting XML document. However it omits generation of attributes.

Element Name and Value Generation

The generator uses a different method for generating tag names. It simply uses letters A, B, C, ..., Z, AA, AB, AC, etc. For element values, it uses similar approach, in generates words tw1, tw2, tw3, etc.

Data Generator Parameters

The input parameters include the following:

- the number of XML documents to generate
- the total number of text words to generate in all the documents
- the total number of distinct text words to generate
- the Zipfian skew parameter of the text word distribution
- the number of levels in the path tree
- the minimum and maximum number of children for nodes at every level (except the lowest level)
- parameters affecting the direct and indirect recursion, number of repeated tag names among internal path tree nodes and among leaf nodes of the path tree
- the total number of XML elements to generate
- the skew parameter of frequencies of the path tree nodes
- the parameter affecting the non-determinism of the generated data

Data Generator Summary

- Source code is not available
- Unknown, probably not implemented
- User interaction: not applicable.
- More than 10 input parameters which are intuitive.
- Focus on generation of documents with widely varying characteristics.
- No schema available.
- No schema available - it is not possible to change it.
- Scaling is achieved via the input parameters.

We do not have a working code which implements this proposed algorithm however it seems to be an interesting way to generate documents. It uses simple input parameters which then affect the output of the generation to a large extent. That means the structure is not set beforehand but the user can affect it simply by changing some input parameters.

3.5.7 <oXygen/> XML Editor

<oXygen/> XML Editor [32] is one of the most advanced XML editors available. It supports many XML technologies and offers editors for plain XML, XML Schema, XSL/XSLT, WSDL, RelaxNG, XQuery, JSON etc. The comprehensive list of its features can be found in [33]. One of the features of this suite is an XML generator. This generator is XML Schema based which means the user supplies file with his desired schema, modifies the generator properties and runs the generator.

Data Generator Parameters

Parameters of this generator are set using a graphical user interface. It is divided into three main parts: schema, options and advanced.

Schema

- *URL*: path to the XML Schema file
- *Namespace*: namespace of the selected schema
- *Root Element*: selection of the root element from all the candidates for root element
- *Output folder*: path where the generated instances are saved
- *Filename prefix and Extension*: names of the generated instances are created using the prefix, number and the extension
- *Number of instances*: number of instances to generate

Options

- *Namespace/Element table*: allows to set a namespace for each element name and it is possible to further customize generation attributes using the following options:
  - *Generate optional elements*: when this is checked, all elements are generated, including the optional ones
  - *Generate optional attributes*: when this is checked, all attributes are generated, including the optional ones
  - *Values of elements and attributes*: controls the content of generated elements and attributes with the following choices:
    - *None*: no content is generated
    - *Default*: inserts a default values which depends on the data type of the particular element or attribute, however type restrictions are ignored. The default value can be either the data type name or an incremental name of the attribute or element.
    - *Random*: inserts a random value depending on the data type of the particular element or attribute
  - *Preferred number of repetitions*: sets the preferred number of repeating elements related with minOccurs and maxOccurs facets.
    - If the value set is between *minOccurs* and *maxOccurs*, then this value is used

        * If the value set is less than *minOccurs*, then the *minOccurs* is used
        * If the value set is greater than *maxOccurs*, the the *maxOccurs* is used
    - *Maximum recursion level*: when there is a recursion, this sets the maximum allowed depth of the same element
    - Choice strategy: sets the strategy used in case of *xs:choice* or *substitutionGroup* elements:
        * *First*: the first branch of the *xs:choice* or the head element of the *substitutionGroup* is always used
        * *Random*: a random branch of *xs:choice* or a substitute element or the head element of a *substitutionGroup* is used
    - *Generate the other options as comments*: it is possible to generate other choices or substitutions as comments so that they can be later uncommented and used
    - *Load/export settings*: saves/loads the settings
    - *Element values*: this allows to add custom values for particular elements; if there are more than one value, the values are randomly selected
    - *Attribute values*: this allows to add custom values for particular attributes; in case of more than one value, random values is selected

Advanced

- *Use incremental attribute/element names as default*: if checked, the value of an element or attribute starts with the name of that element or attribute, e.g. for element a the generated values are a1, a2, etc. If not checked, the value is the name of the type of that element or attribute, e.g. string, integer, etc.
- *Maximum length*: the maximum length of string values generated for elements and attributes
- *Discard optional elements after nested level*: optional elements that exceed this specified level are not generated anymore

Data Generator Summary

- Source code is not available, it is a commercial tool
- Written in Java, binaries available for Windows, Mac OS X and Linux, also available as a plugin for Eclipse
- The whole application has a graphical user interface, the data generator is available as a command line tool as well
- Input parameters are simple and well described in the application
- The primary focus of the generator is to create sample documents from the provided XML Schema file
- Output XML documents entirely rely on the input XML Schema file and set parameters
- Output document schema corresponds to the input XML Schema file (however there are special cases when this is not true)

- Scaling of the generated files is achieved by altering the parameters of the generator; the scaling is not direct, i.e. there is no parameter like total number of elements or size in bytes etc.

3.5.8 Liquid XML Studio 2013

Liquid XML Studio [34] is an advanced graphical XML development environment. It comes in two editions (Starter and Designer) which slightly differ in their capabilities. The more advanced version contains graphical editor fox XML files, XML Schema files, WSDL files and text editors for XPath, XQuery, DTD, CSS, XDR and more. Full list of its features can be found here [35].

One of its features is also a data generator. This generator uses an XML Schema file as the main input. The parameters are then set via wizard. The first step is the schema file section, then the root element. The next step is the setup of the basic options. The last step involves the namespace options.

Data Generator Parameters

The options are set using a graphical wizard. The available options are:

- *Root element*: first step offers the list of all the possible root elements in the schema file and lets the user to select one
- *Include a 'schemaLocation' attribute*: when checked then the schemaLocation or noNamespaceSchemaLocation attribute is added to the root element
- *Force the creation of optional entities to a depth of*: whenever there is an element with minOccurrs 0 or an attribute with use=optional, there is a 50% chance that this entity will appear in the resulting XML; this option forces the appearance of these entities until the nesting depth reaches this value, then the chance this optional entity appears is 50%
- *Stop creating optional items after a depth of*: this options prevents the appearance of optional entities in the resulting document after the depth reaches this value
- *Generate attributes for <xs:anyAttribute>*: when checked the generator creates attributes for elements with this specification
- *Generate elements for <xs:any>*: when checked the generator creates elements when it encounters this declaration in the schema file
- *Namespace options*: controls the default namespace of the generated XML document and also the namespace aliases used in the document

Data Generator Summary

- Source code is not available, it is a commercial tool
- Requires .NET Framework, binaries are available only for Windows, it is also available as a plugin for Microsoft Visual Studio
- The whole application has graphical user interface, including the data generator
- There are few input parameters which are simple
- Primary focus of the generator is to create example documents from the provided XML Schema file

- Output XML documents entirely rely on the input XML Schema file and set parameters
- Output document schema corresponds to the input XML Schema file (however there are special cases when this is not true)
- Scaling of the generated files is achieved by altering the parameters of the generator; the scaling is not direct, i.e. there is no parameter like total number of elements or size in bytes etc.

3.5.9 Altova XMLSpy

Altova XMLSpy [36] is an XML editor for modeling, editing, transforming and debugging XML-related technologies. It has a support for XSLT, XPath, XQuery, SOAP, WSDL, XBRL, JSON, Office Open XML etc. More detailed list of features can be found in [37]. Its XML editor also contains a sample document generator. It is possible to generate sample XML files either from DTD or XSD.

When generating sample from XML Schema file or DTD, the user can alter the following options:

- *Choice generation* - it is possible to select among three options:
  - First branch of choice
  - All branches of choice
  - Branch of choice with the smallest number of elements
- *Number of repeatable elements*: it is possible to specify how many elements should be generated for repeatable elements
- General generation options - each of these is a checkbox when user can check the desired options:
  - Generate non-mandatory elements
  - Generate non-mandatory attributes
  - Fill elements with data
  - Fill attributes with data
  - Treat element content of nillable elements as non-mandatory
  - For elements with an abstract type, try to use a non-abstract type for xsi:type
- *Schema assignment*: user can choose whether the generated document contains path to its schema (either absolute, relative or no path)
- *Usage of manually added sample values*: it is possible to use manually added sample values, if available and these can be selected randomly, sequentially or using the first value; facet window enables user to manually specify some sample values for each element which can have text value, generator then can use these values and create more realistic samples
- *Root selection*: user can select the desired root element for the generated document

Data Generator Summary

- Source code is not available, it is a commercial tool
- Written in C++ and designed only for Microsoft Windows

- Whole application has graphical user interface, including the data generator
- There are few input parameters which are simple
- Primary focus of the generator is to create sample documents from the provided XML Schema or DTD file
- Output XML documents entirely rely on the input XML Schema file and on the preset parameters
- Output document schema corresponds to the input XML Schema file
- Scaling of the generated files is achieved by altering the parameters of the generator; the scaling is not direct, i.e. there is no parameter like total number of elements or size in bytes etc. The scaling is not very flexible, the only option that alter the size of the generated document is the number of repeatable elements and this is a single number for the whole document.

3.5.10 Eclipse

Eclipse [38] is a multi-language integrated development environment for many programming languages and has a rich plug-in system which allows customization. One of the features is a simple data generator which is capable of creating sample documents from DTD files or from XML Schema files. The generation process is straightforward. The whole process starts with opening the DTD or XSD file, right-clicking on it and choosing Generate -> XML File... from the context menu. Then it is possible to change the following options:

- *Root element*: user can select from the list of elements eligible as root elements
- *Create optional attributes*: if checked, optional attributes get generated
- *Create optional elements*: if checked, optional elements get generated; there is also an option to limit the depth for optional elements
- *Create first choice of required choice*: if checked, the first element from choice gets selected during generation
- *Fill elements and attributes with data*: if checked, generator creates data for elements and attributes
- *Namespace Information*: it is possible to select prefixes for various namespaces

The similar options are available when the generation is based on DTD file (except the namespace options).

Data Generator Summary

- Source code is available
- Written mostly in Java, runs on Windows, Mac OS X, Linux
- Whole application has graphical user interface, including the data generator
- There are only few input parameters which are simple
- Primary focus of the generator is to create example documents from the provided XML Schema or DTD file
- Output XML documents entirely rely on the input XML Schema file and set parameters

- Output document schema corresponds to the input XML Schema file
- Scaling of the generated files is achieved by altering the parameters of the generator; the scaling is not direct, i.e. there is no parameter like total number of elements or size in bytes etc.

3.5.11 Microsoft Visual Studio

Microsoft Visual Studio [39] is another integrated development environment we analyze. It is a mature application used for various purposes. It can be used for application development ranging from console applications, through classic desktop applications, web applications, mobile applications to name just a few. It also has an XML editor and is capable of data generation. This data generator is only XML Schema based and offers no options to customize the output. Its sole purpose is to generate a sample from the schema so the user can validate the schema is correct.

3.5.12 Summary of XML Data Generators

In this chapter we described several data generators with different approaches to data generation. Many of them use a predefined schema and change only the text content or the number of elements in the result. The majority of these data generators are not very well suited for the purpose of using these data sets as input for testing any application working with XML. Two exceptions are the ToXgene and the Complex-structured XML data generator. ToXgene is capable of creating almost any XML data and, therefore, it is the best from this point of view. Complex-structured XML data generator can also produce different data sets, however it must be altered in some way to offer a greater power to the user. Our solution will be based on ideas used in this data generator.

The rest of the described solutions has XML data generation as a secondary function. They accept an XML schema as an input and they produce sample XML documents based on a set of predefined parameters (which are different for each solution). These applications differ mainly in what parameters they provide for the user to change. Tables 3.2 and 3.3 contain the summary of described XML data generators.

	XMark	XOO7	XMach-1	Michigan Benchmark	ToXgene	Complex-structured generator
Source code available	yes	yes	yes	yes	no	no
Programming language	C (Windows, Linux, Solaris, IRIX)	C	Java	C++ (Unix, Windows)	Java	unknown
User interaction mode	command line	command line	command line	command line	command line	not applicable
Input parameters	4 input parameters, easy to understand	8 input parameters, confusing	9 simple parameters	2 simple input parameters	main input is template in TSL	more than 10 intuitive input parameters
Primary focus	generation of well-formed, valid and meaningful XML data	generation of data sets for the benchmark	generation of data for the benchmark with predetermined structure	data generation focused on depth and fan-out	generation of large, consistent collections of synthetic XML documents	generation of documents with widely varying characteristics
Output schema	DTD of the generated XML is available	DTD of the generated XML is available	DTD of the generated XML is available	XSD of the generated XML is available	schema is created by the user	schema not available
Output schema alteration	not possible	not possible	not possible	not possible	no single schema, template affects the output	schema not available
Document scaling	possible via input parameters	possible indirectly via input parameters	possible via input parameters	possible via the input parameter	possible inside the template	possible via input parameters

Table 3.2: Summary of XML data generators

	oXygen XML Editor	Liquid XML Studio 2013	Altova XMLSpy	Eclipse	Microsoft Visual Studio
Source code available	no	no	no	yes	no
Programming language	Java	Unknown (requires .NET Framework)	C++ (Windows)	Java	C++
User interaction mode	graphical user interface	graphical user interface	graphical user interface	graphical user interface	graphical user interface
Input parameters	simple input parameters, easy to understand	simple input parameters, easy to understand	simple input parameters, easy to understand	simple input parameters, easy to understand	no parameters
Primary focus	generation of sample documents from the schema	generation of sample documents from the schema	generation of sample documents from the schema	generation of sample documents from the schema	generation of sample documents from the schema
Output schema	output document schema corresponds to the input schema	output document schema corresponds to the input schema	output document schema corresponds to the input schema	output document schema corresponds to the input schema	output document schema corresponds to the input schema
Output schema alteration	via input schema	via input schema	via input schema	via input schema	via input schema
Document scaling	possible indirectly via input parameters	possible indirectly via input parameters	possible indirectly via input parameters	possible indirectly via input parameters	not possible

Table 3.3: Summary of XML data generators #2

4. MongoDB

MongoDB [45] is an open-source, document database. It provides high performance, high availability and automatic scaling. MongoDB stores documents which are data structures composed of key and value pairs. The documents are JSON objects stored in BSON format. The fields (keys) in documents may have values that are other documents, arrays, arrays of documents etc. Listing 19 shows a simple document that can be stored inside MongoDB.

```
{
        "_id": 1,
        "first_name": "John",
        "last_name": "Doe",
        "address": {
                "street": "High Street",
                "city": "Rome",
                "state": "Italy"
                },
        "email": "john.doe@email.com",
        "age": 30,
        "favourite_colours": ["blue", "red", "green"]
}
```

Listing 19: MongoDB document example

4.1 Key Features of MongoDB

There are several advantages of documents. Documents (can also be seen as objects) correspond to data types in many programming languages. Embedded documents reduce the need for expensive joins. The schema of the documents is dynamic so the users can modify their structure as they wish.

High performance is achieved via support for embedded data models which reduces I/O activity in database system and also by indexing. In order to support high availability, MongoDB provides replication facilities called replica sets. These provide automatic failover and data redundancy. Finally, automatic scaling is achieved using a function called automatic sharding. This distributes data across a cluster thus providing a horizontal scalability. Replica sets can also provide eventually-consistent reads.

4.2 CRUD Operations in MongoDB

MongoDB stores all its data in the form of documents. These documents associate keys with values, e.g. dictionaries, hash maps, associative arrays etc. Every document in MongoDB is stored in BSON format. These documents are stored in collections. A collection is a group of similar documents.

4.2.1 Read Operations

Read operations or queries retrieve data from the database. In MongoDB, results of queries are documents selected from a collection of documents. As usual, queries contain conditions identifying the documents to return. MongoDB also offers an option to specify projections that specify which fields from documents to return. This way it is possible to limit the amount of transferred data.

MongoDB provides a `db.collection.find()` method for queries. It is possible to specify both the query criteria and projections as parameters. Optionally, it is possible to specify limits, skips and it is also possible to sort the result. The result of the method is a cursor to the matching documents. Listing 20 contains a sample query. This query looks for documents which have the field `last_name` equal to `Doe` and the `favourite_colours` array contains `red` and `blue`, and the `age` field has the value between 30 and 40 (including 30 and 40). It returns only the field `first_name`. Results are sorted in descending order by the `age` field and the query returns only 10 documents.

```
db.people.find(
        {
                last_name: "Doe",
                favourite_colours: { $all: ["red", "blue"] },
                age: {$gte : 30, $lte: 40}
        }, // query criteria
        { first_name: 1 } // projection
).sort({'age': -1}) // sorting in descending order
.limit(10) // cursor modifier
```

Listing 20: Sample query

Queries in MongoDB always target a single collection. The order of documents is not defined unless there is a `sort` modifier specified. There is also a special method `db.collection.findOne()` that returns a single document.

There are many query selectors and operators available in MongoDB. They can be divided by their type. The following list summarizes the available options:

Comparison operators

- `$eq` - matches values equal to the specified value
- `$gt` - matches values greater than the specified value
- `$gte` - matches values greater than or equal to the specified value
- `$lt` - matches values less than the specified value
- `$lte` - matches values less than or equal to the specified value
- `$ne` - matches values not equal to the specified value
- `$in` - matches any values specified in an array
- `$nin` - matches none of the values specified in an array

Logical operators

- `$or` - joins clauses with a logical OR
- `$and` - joins clauses with a logical AND

- `$not` - inverts a query clause
- `$nor` - joins clauses with a logical NOR

Element operators

- `$exists` - matches documents that have the specified field
- `$type` - selects documents where a field has the specified type

Evaluation operators

- `$mod` - modulo operation
- `$regex` - selects documents where values match a specified regular expression
- `$text` - performs text search
- `$where` - matches documents that satisfy a JavaScript expression or a full JavaScript function

Geospatial operators

- `$geoWithin` - selects geometries within a bounding GeoJSON geometry
- `$geoIntersects` - selects geometries that intersect with a GeoJSON geometry
- `$near` - returns geospatial objects in proximity to a point (requires a geospatial index)
- `$nearSphere` - returns geospatial objects in proximity to a point on a sphere (requires a geospatial index)

Array operators

- `$all` - matches arrays that contain all elements specified in the query
- `$elemMatch` - selects documents where an element in the array field matches all the specified $elemMatch conditions
- `$size` - selects documents where the array field has a specified size

MongoDB offers four projection operators:

- `$` - projects the first element in an array that matches the condition
- `$elemMatch` - selects the first element in an array that matches the specified $elemeMatch condition
- `$meta` - projects the score of the document assigned during the $text operation
- `$slice` - limits the number of projected elements from an array (supports skips and limits)

4.2.2 Write Operations

Write operation is an operation in MongoDB that creates or modifies data. Write operations also target a single collection. Every write operation is atomic at the level of a single document.

MongoDB offers three basic write operations: insert, update and remove. No insert, update or remove operation can affect more than one document atomically. Update and remove operations use the same syntax for specifying which documents to alter.

Insert operation in MongoDB is done via calling the `db.collection.insert()` method. This method adds new documents to a collection. Listing 21 shows an example of the insert method call. Every document in MongoDB contains `_id` field which must be a unique identifier within a collection. If the users adds new document without this field, MongoDB adds one and populates it with a unique `ObjectId` (`ObjectId` is a 12-byte BSON type that guarantees uniqueness within the collection).

```
db.people.insert(
        {
                "first_name": "Jane",
                "last_name": "Doe",
                "address": {
                        "street": "Low Street",
                        "city": "Budapest",
                        "state": "Hungary"
                        },
                "email": "jane.doe@email.com",
                "favourite_colours": ["green"]
        }
)
```

Listing 21: Sample insert in MongoDB

Update operation in MongoDB is performed using the `db.collection.update()` method. This method accepts a criteria which determines which documents to update and also contains the update action. There is also an update option affecting the behaviour of the update, e.g. `multi` option which causes updates of multiple documents. Listing 22 contains an example of the update operation. Update operation method affects only a single document by default. To update multiple documents, it is necessary to set the `multi` option to `true`.

```
db.people.update( // specifies the collection and operation
        { "last_name": { $eq: "Doe"} }, // search criteria
        { $set: { "favourite_colours": ["black"] } }, // update action
        { multi: true } // update option
)
```

Listing 22: Sample update in MongoDB

Another option available for `update()` method is `upsert`. This option (if set to `true`) specifies that if no documents match the query portion of the update, the update operation should create a new document. Otherwise it updates the matching documents.

In MongoDB, there is a method called `db.collection.remove()` for deletion of specified documents from a collection. This method also accepts a query criteria which specifies which documents to remove. Listing 23 contains an example of the remove operation in MongoDB. Default behaviour of the `remove()` method is to remove all documents matching the query criteria. It is also possible to set a flag to remove only a single document. This flag is called `justOne`.

```
db.people.remove( // specifies the collection and operation
        { "last_name": "Doe" } // search criteria
)
```

Listing 23: Sample delete in MongoDB

Write Concern

Write concern [91] in MongoDB describes the kind of guarantee that MongoDB provides when reporting the result of the write operation. The strength of this write concern determines the level of the guarantee. Write operations with weak write concern return quickly but in case of a failure, these operations might not get persisted.

There are 4 levels in MongoDB:

- *Unacknowledged*: MongoDB does not acknowledge the receipt of write operations.
- *Acknowledged*: MongoDB confirms the receipt of the write operation and applies the change to the in-memory view of data. This is the default write concern in MongoDB.
- *Journaled*: MongoDB acknowledges the write operation only after committing the data to the journal thus ensuring the ability to recover the data after shutdown or power interruption.
- *Replica Acknowledged*: this level concerns replica sets. With replica acknowledged write concern, there is a guarantee that the write operation propagates to additional members of the replica set.

4.3 Aggregation in MongoDB

Aggregation is an operation that processes data records and returns some computed results. There are several aggregation operations available in MongoDB and we will briefly describe the options. Aggregations in MongoDB use collections of documents as inputs and return one or several documents as output.

There are several options for aggregation. There is a framework called aggregation pipeline in MongoDB, map-reduce operations and single purpose aggregation operations. All these options can be used for data processing.

4.3.1 Aggregation Pipeline

Aggregation pipeline [75] is data aggregation framework based on the concept of data processing pipelines. All the input documents enter the processing pipeline where there are performed various transform operations which create an aggregated output.

Aggregation pipeline consists of multiple stages. Every document is being transformed as it passes these stages. A method for aggregation pipeline in MongoDB is called `db.collection.aggregate()`.

4.3.2 Map-Reduce

Map-Reduce [76] is a data processing style that works with large data sets aggregating them into smaller results. MongoDB provides the `mapReduce` command for this type of aggregation. This processing operation consists of two phases: map and reduce. Let us consider an example of the `mapReduce` operation (taken from [76]) in Listing 24.

```
db.orders.mapReduce(
        function() { emit( this.cust_id, this.amount ); }, // map function
        function(key, values) { return Array.sum( values ) }, // reduce function
        {
                query: { status: "A" }, // query
                out: "order_totals" // output
        }
)
```

Listing 24: Sample map-reduce operation in MongoDB

The map operation is applied to documents matching the query. The map operation emits key-value pairs. If there are multiple values for the same key, the reduce operation is applied. This reduce phase sums the values of the amounts. The result is stored in the output collection.

4.3.3 Single Purpose Aggregation Operations

In addition to aggregation pipeline and map-reduce operation, MongoDB contains a number of single-purpose aggregation operations [77]. These are applied over a collection of documents returning some result. The operations available are: `count()`, `distinct()` and `group()`.

4.4 Indices in MongoDB

Efficient execution of queries in MongoDB is achieved by using indices. Without them, the database has to perform a collection scan, i.e. iterating through every document and selecting the documents matching the query. With index, MongoDB can limit the set of documents it has to go through.

Indices in MongoDB are defined at the collection level. Indices can be specified for any field or sub-field of the documents in the collection. They store the value of field or fields, ordered by the value of the field. This ordering supports efficient equality matches and range-based queries. MongoDB can also return sorted results using the ordering saved in the index.

4.4.1 Index Types

There are several index types in MongoDB for different types of data and queries.

Default Index

Every collection has an index on the `_id` field which exists for every document in a collection.

Single Field Index

MongoDB supports a user-defined index (ascending or descending) on a single field of a document. The sort order of the index key does not matter in this case because MongoDB can traverse this index in both directions.

An index is created using a `createIndex` method on a specific collection, an example is in Listing 25.

```
db.people.createIndex( { "first_name" : 1 } )
```

Listing 25: Singe field index creation example

Compound Index

Another type of user-defined index is compound index which is defined on multiple fields. The order of fields in this index is significant. Listing 26 shows creation of index for fields `first_name` (ascending - `1`) and `last_name` (descending - `-1`). The index contains information for documents sorted first by the values of `first_name` field and within each value of the `first_name` sorted by the `last_name` field. This order of fields is important for sorting, queries matching on all the index fields, as well as for matching based on the prefixes of the index.

```
db.people.createIndex( { "first_name" : 1, "last_name" : -1 } )
```

Listing 26: Compound index creation example

Compound indices can be used also for queries on the prefixes of the index. In our example, the index can also be used for queries on the `first_name` field alone. If the index contained more than two fields, queries on all the prefixes of the compound index would be supported by this index.

Multikey Indices

MongoDB supports indices on fields that hold an array value. These indices are called multikey [78] and can be constructed over arrays that hold scalar values and also nested documents. For multikey index creation, the same method `db.collection.createIndex()` is used. Listing 27 shows a multikey index creation which is the same as for other index types.

```
db.people.createIndex( { "favourite_colours" : 1} )
```

Listing 27: Multikey index creation example

Compound multikey indices have a limitation that each indexed document can have at most one indexed field with an array value.

Geospatial Indices

MongoDB offers three types of indices for geospatial data [79]. There is a `2dsphere` index for queries calculating geometries on an earth-like sphere. For flat surfaces, there is a `2d` index. The last index type for geospatial data is a special index optimized for small areas called `geoHaystack` index. The creation of these index types also uses the `db.collection.createIndex()` method with additional fields specified during the index creation.

Text Indices

There is also a support for text search inside string content in documents in MongoDB. This is achieved using a `text` index type [80]. This type of index can be used on any field with string value or with an array of string values. The collection, however, can have at most one `text` index. The creation of `text` index is similar to previous cases, only the `text` literal is used for indexed field, as shown in Listing 28.

```
db.people.createIndex( { "notes" : "text"} )
```

Listing 28: Text index creation example

For cases when documents contain highly unstructured data, there is an option to index all fields with string content. This is done using the wildcard text indices. The syntax for this type of index is shown in Listing 29.

```
db.people.createIndex( { "$$**" : "text"} )
```

Listing 29: Wildcard text index creation example

Hashed Indices

The last type of index in MongoDB is hashed index [81]. This type of index maintains entries with hashes of the indexed field values. The hashing function collapses embedded documents and computes a hash value for the entire document. Multikey indices (for arrays) are not supported. Hashed indices support sharding (for more information on sharding, see Section 4.6) a collection using a hashed shard key. It is not possible to create compound indices that have hashed index fields and it is not possible to create unique constraint on a hashed index. The creation of the hashed index is shown in Listing 30.

```
db.people.createIndex( { "last\_name" : "hashed"} )
```

Listing 30: Hashed index creation example

4.5 Replication in MongoDB

MongoDB supports data replication [82]. In MongoDB, a replica set is a group of `mongod` instances that all contain the same data. One instance is the primary one. This primary instance receives all write operations. The rest of the instances (secondaries) receive the operations from the primary which they apply to the data so that they have the same data set. This replication is done asynchronously.

If the primary instance does not communicate with other members of the replica set for more than ten seconds, the replica set tries to select another member of the set to be the primary. This is how the automatic failover is done in MongoDB.

4.6 Sharding

Sharding is a method for storing data across multiple servers. MongoDB uses this technique to support storage of very large data sets and high throughput.

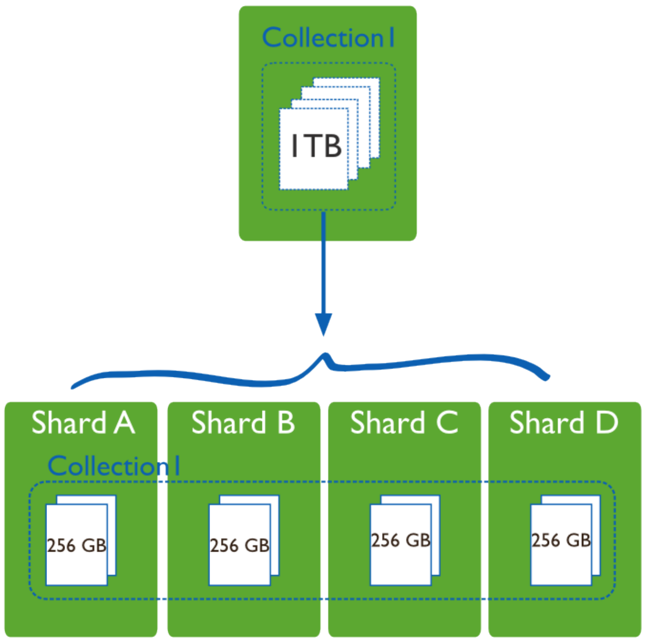


Figure 4.1: Sharded collection in MongoDB

Horizontal scaling, or sharding, divides the data sets over multiple hosts, or shards. Every single shard is an independent database. These shards together compose a logical database. An example is shown in Figure 4.1 (taken from [83]).

This approach has several advantages. It reduces the number of operations each shard handles as the number of them grows. This means that the whole cluster can increase its capacity and throughput horizontally. Sharding also reduces the amount of data each shard stores as the number of shards in a cluster grows.

Sharded cluster in MongoDB has three main components: shards, query routers and config servers. Shards are data stores. Each shard can be represented by a replica set. Query routers are components that clients communicate with. Query routers processes the operations from clients, direct them to shards and return results back to clients. Config servers are metadata stores of the cluster. It contains mappings of data set to the shards. Query routers use this metadata to target operation to the right shards. There are exactly three config servers for production clusters.

4.6.1 Data Partitioning

Data distribution among shards is done using the shard key at the collection level. Shard key is either an indexed field (single or compound) that exists in every document in the collection. MongoDB divides the values of the shard key into *chunks* and these chunks are then distributed evenly across the shards. Division into chunks is done either using the range-based partitioning or hash-based partitioning.

In range-based partitioning, the whole data set is split into parts that are determined by the shard key values, i.e. to which range of all the values the keys fit. For example, for numeric shard key, the whole key space is divided into several non-overlapping ranges. Each key then belongs to exactly one of these ranges. In this partitioning scheme, documents with close shard key values are likely to be in the same chunk.

In hash-based partitioning a hash of the value of the field is computed. MongoDB then uses these hashes to create chunks. In this case, when two documents have close shard key values, they are unlikely to be part of the same chunk. This creates more random distribution of the data in the cluster.

New data or new server additions can cause imbalances in data distribution, i.e. one shard contains significantly more data than the others. MongoDB resolves this using two processes: splitting and balancing. More about these processes can be found in MongoDB manual [83].

4.7 Database References

There is no support for joins in MongoDB. Data in MongoDB is usually stored in denormalized state or it is stored with related data in documents. This removes the need for joins. However, there are cases when the storage of related information in separate documents makes sense.

MongoDB supports two methods for creating relationships between documents [71]:

- *Manual references* where the application saves the `_id` field of one document in another document as a reference. In the example in Listing 31 we insert an address into `addressBook` collection and we are using its `_id` field when inserting the information about the person. If we want to retrieve this referenced document later, the application has to run an additional query to get the referenced document.
- *DBRefs* are references from one document to another using the value of the `_id` field of the referenced document, collection name, and, optionally, its database name. The application must also resolve these references by performing additional queries. However, many drivers have helper methods

to create queries that retrieve these references. These drivers do not automatically resolve DBRefs into documents. An example of how this reference looks in a document is shown in Listing 32.

```
original_id = ObjectId()

db.addressBook.insert({
    "_id": original_id,
    "street": "Diagonal Alley",
    "city": "London"
})

db.people.insert({
    "name": "Ron",
    "address_id": original_id
})
```

Listing 31: Random field definition

```
{
  "_id" : ObjectId("1236aaf64aed4daf9f1ab771"),
  // .. other fields
  "address" : {
                  "$ref" : "addressBook",
                  "$id" : ObjectId("1236bc054aed3d169e2ab155"),
                  "$db" : "people"
               }
}
```

Listing 32: DBRef example

5. Data Generator Architecture and Implementation

One of the aims of this thesis is to design and implement a data generator for document databases. This data generator should be able to generate large amounts of documents for a document database - in our case - MongoDB. We chose this database due to its popularity and maturity. There are also many tools, libraries and resources available for this database system.

The proposed data generator should have the following properties:

- The data generator will be able to to generate large data sets in parallel, i.e. on multiple computers concurrently.
- Data sets will consist of large numbers of small JSON documents (i.e. several kB).
- The data generator will be able to generate references between documents.
- It will be possible to change the number of generated documents.
- It will be possible to change the schema of the generated documents.
- It will be possible to use an existing set of JSON documents as a base for data generation.
- The usage of the generator should be easy.

5.1 Possible Solutions for the Architecture

In the following paragraphs we will discuss the options for the data generator and the decisions we made during the design phase.

We designed the proposed data generator with the analysis of existing data generators in mind. We considered different approaches to data generation, their options, capabilities and we based parts of our solution on some of their properties and ideas which we considered useful.

Our aim is to generate documents for MongoDB which natively use JSON format. Despite this we also considered many XML data generators as well. The XML format can be easily transformed into JSON. Therefore it should be possible to generate XML documents and transform them to JSON format at the final stage of data generation.

The XML data generators we analyzed used either predefined fixed structures (schemas) for their outputs or a schema in XSD format. The fixed schema approach is not suitable for our requirements. It has its usage for specialized benchmarks as we described in Chapter 3.

The approach used in the ToXgene [27] data generator is more suitable for our requirements. It can generate basically any structure including very complicated ones. However, the TSL format is quite complicated and might discourage the potential users from using it.

The idea of basing the data generation on templates, on the other hand, seems very useful. The user can express the desired schema as closely as he wants and the template more or less resembles the output. We found this idea also in other data generators (specifically in JSON data generators).

JSON generators we analyzed use templates as well. Three of them (json-generator.com, generatedata.com and Mockaroo) are web applications which makes them unusable for parallel data generation. However, they also use a template-based approach. MongoDB-Datasets application uses similar approach as json-generator.com, but it is not a web application and there is also the source code available.

Data stored in document databases usually do not have a fixed schema. It is possible and it is usually the case that there are different documents stored together (in one collection in MongoDB). However, these documents are very often similar, i.e. there is a common set of attributes they all share. For example, when we store information about users, there are some required fields we have for every user but there are also some optional ones, like e.g. interests or favourite movies which we do not require the user to type in, so there is no need to save them if they are missing.

Based on these observations we decided to generate JSON data directly as this is natively used in MongoDB. We also decided to use the template-based approach used in MongoDB-Datasets application mostly due to its expressive power and simplicity for the users. These templates are very similar to the generated data so creating them is easy and moreover, we also designed a way to create this template automatically if the user has some representative documents.

However, this generator could potentially be used also for XML data generation via the transformation of JSON documents to XML documents. We did not pursue this approach but it is a possibility.

5.2 Design of the Architecture

The required properties also influenced how we designed the overall architecture of the data generator. The first property requires parallel execution but not only within a single application but across possibly many computers. The generation of references requires some form of communication between the instances of the data generator and this can be solved in many different ways.

The data generator we designed and implemented is called *JsonGenerator*. We will describe its main components in the following paragraphs and explain how it helps to achieve the required properties.

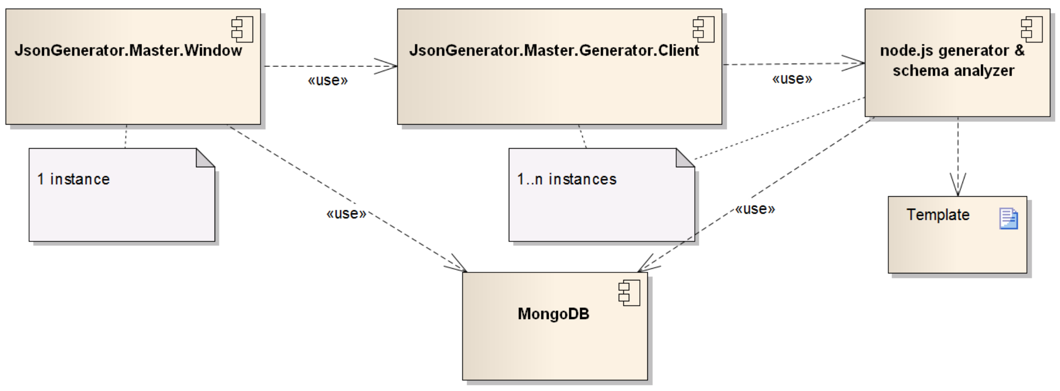

JsonGenerator.Master.Window
«use»
JsonGenerator.Master.Generator.Client
«use»
node.js generator & schema analyzer
1 instance
«use»
1..n instances
«use»
Template
MongoDB

Figure 5.1: Main components of the proposed data generator

The main components of the *JsonGenerator* are shown in Figure 5.1.

5.2.1 JsonGenerator.Master.Window

The main role in the data generator has the component called `JsonGenerator.-Master.Window` (we will use the term `Master` from now on) which coordinates the data generation and it is also a main interaction point for the user. It is a place where the data generation process starts and where the properties of the data generation process are configured. It also shows the progress of the data generation.

5.2.2 JsonGenerator.Master.Generator.Client

The component called `JsonGenerator.Master.Generator.Client` (`Client` from now on) is an application which will run on every server responsible for data generation. That means that there will be multiple instances of this application running at the same time. The role of this application is to receive tasks from the `Master` and delegate it to actual data generator and then notify the `Master` about the results.

5.2.3 The Data Generator and Schema Analyzer

The actual data generator and schema analyzer component (we will refer to it simply as data generator component) is also present in multiple instances and has one-to-one relationship with the `Client` application. That means that every instance of the `Client` application uses its own instance of the data generator component. This component performs the data generation and also infers the schema from the collection of existing documents.

From the standpoint of the architecture, it would be better to have the `Client` and `data generator` as one component, because it is logically one component (this is the reason why we use the one-to-one relationship between them). However, due the technologies we later chose for the implementation we had to separate this into two separate modules.

This component could be implemented also in such a way that the output is not only JSON but also XML or similar format. This would require a module that would transform the generated data to the required format.

5.2.4 Data Store

An important part of the data generator is the MongoDB database. This database is used as a storage place for input documents and also for the generated documents. This decision allows us to implement the references between documents without a special communication protocol between individual instances of the generator.

However, this decision directly limits our ability to test another database. There is a solution to this problem. It is necessary to abstract the interface to the database so we can perform the lookup for references independently of the datastore we use. This same solution can be applied to the document saving stage. The document saving stage can also be solved using another method. The output of the data generator can be a file. Then it is possible to write a program which imports the files into the database. It could also perform all the necessary transformations of the generated documents to the format specific for the used database.

5.2.5 The Architecture Summary

How this architecture fulfills the required properties? The fact that the generator can run on several computers means the data generation is parallel. Usage of the MongoDB database as a storage place enables us to use this for the generation of references (see Section 4.7). The user will input the number of desired documents in the `Master` component and this will be used for the data generation process as a parameter. We did not describe the data generation process yet but it will be possible to change the schema of the resulting documents via a template (see Section 5.3.2).

5.3 Design and Implementation Details

We described the overall structure of the data generator. Now we are going to focus on the specific parts more closely.

5.3.1 Technologies

The Master application is written in C# for the .NET 4.5.1 [87] platform. It is an application with simple user interface which controls the server part of the generator, data generation and schema inferring.

The Client application is written also in C#. It is a console application, because users do not have to interact with it. For our testing purposes we used it as a standalone application. For regular usage we would suggest to deploy it as a Windows service.

The Node.js data generator and schema analyzer is written in JavaScript and can run as an independent console application or can be hosted in different servers.

5.3.2 Templates

JsonGenerator uses templates as an input for data generation. These are the same templates as used in the described *MongoDB-Datasets* application. We are not going to describe them as they are already described in Chapter 3 and there is also a detailed documentation available online [65].

However, there are additional options we added to the templates for the *JsonGenerator*. The original template specifies only how the document should look like, we added additional meta data for the generator so that it knows where to store the generated data. This is necessary for the case when the generator generates data using multiple templates and the user would have to add a mapping between the templates and collections inside MongoDB. Listing 33 shows this meta data. This is written as the value of the field `_$db`.

These settings instruct the data generator where to save the results of the data generation. This structure can be extended if the data generator needed

```
{
        "_$db": {
                "sourceDb": "local",
                "sourceCollection": "test",
                "targetDb": "test",
                "targetCollection": "people"
        },
        "firstName": "{{ chance.first() }}",
        // ...rest of the template...
}
```

Listing 33: Meta data in template

additional data during the data generation phase. This structure is not written to the output document. This is also true for any other field that starts with the characters `_$`.

5.3.3 Schema Inferring

Writing a template for data generator can be sometimes a tedious work. When there are already some existing documents present, we can infer their schema and generate the template automatically. This template can be then further fine-tuned manually.

Our solution uses a small Node.js application called *mongodb-schema* [85]. This application produces a probabilistic schema for a collection inside MongoDB. Figure 5.2 depicts the class diagram of the data structure it produces (the figure is taken from [85]).

The inferred information includes the names and types of all the fields and also the probability of field being included in the document. There are also all possible values stored that were present in the scanned documents.

These classes correspond to parts of the JSON document, e.g. `Field` class contains information about one field. This field can have different types which are expressed as separate classes.

We use this inferred information to construct a new template in the format suitable for MongoDB-Datasets. This template construction involves the generation of the right data generator for each of the fields. JSON has only basic types like `number`, `boolean` and `string`, so it is not directly possible to transform the inferred data type into the right data generator. For instance, only the possibilites we have for string fields are basically limitless. We can use any function from the Chance [68] or faker.js [69] library. These libraries can produce various random data types. These include names of people, words, sentences, various dates like birthdays, numbers, URLs and many more. If we included additional libraries or written custom string generators to the data generator, it would also be possible to use them.

Our implementation uses `chance.word()` for string types, `chance.floating()` with `min` and `max` values specified for numeric types (or `chance.integer()` if all the present values in scanned documents were integers). Similarly, for boolean types we use `chance.bool()`, for dates `faker.Date.recent()` and finally `ObjectID-(chance.hash({length: 12}))` for objectid type.

It is possible to include additional logic in this template generator which would

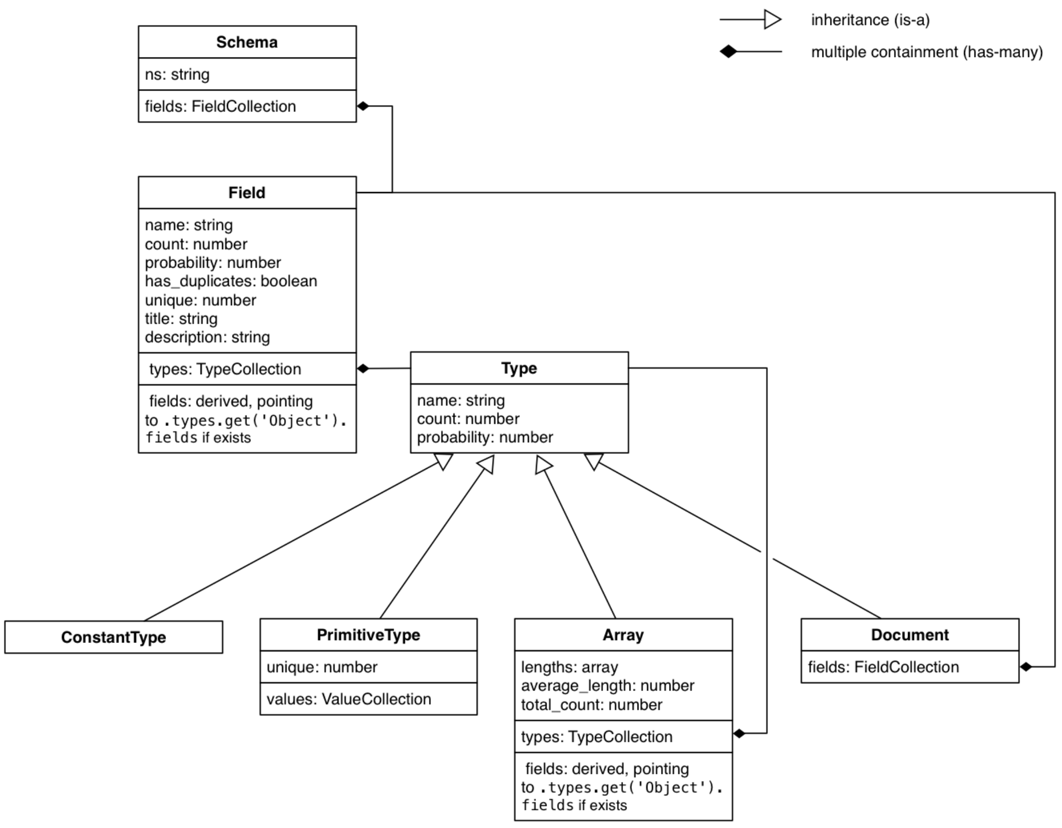


Figure 5.2: High-level view of the class interaction in mongodb-schema

be capable of detecting various data types using regular expressions, machine learning or similar methods.

Other possible improvement we did not implement is to generate coniditons for optional fields based on the analyzed data. Out algorithm adds template for every field so the generated documents always include the union of all fields present in the sample set. Our implementation leaves this step to user which can add additional conditions for optional elements.

Listing 34 contains one sample document from the collection of similar documents. After the schema of the document has been inferred, our application creates a result as shown in Listing 35. All `string` fields contain the `word()` generator from Chance library. For numeric types, the template contains corresponding data types based on the data found in the source documents. For instance, the field `pop` has been identified as an integer with values between 0 and 65046.

Our implementation limits the number of analyzed documents to 1000. This constant can be easily changed to any other desired number.

5.3.4 Data Generation

As we have mentioned, the data generator is written in JavaScript using Node.js framework. It uses a template from which it constructs an in-memory representation of the document, i.e. parses the JSON template and constructs the tree structure of JavaScript objects which also contains the part of the tem-

```json
{
    "_id" : "01002",
    "city" : "CUSHMAN",
    "loc" : [
        -72.5156499999999940,
        42.3770170000000020
    ],
    "pop" : 36963,
    "state" : "MA"
}
```

Listing 34: Sample document to analyze

```json
{
  "_$db": {
    "sourceDb": "local",
    "sourceCollection": "zips",
    "targetDb": "local",
    "targetCollection": "zips"
  },
  "_id": "{{chance.word()}}",
  "city": "{{chance.word()}}",
  "loc": [
    "{{_$config}}",
    {
      "size": [
        2,
        2
      ]
    },
    "{{chance.floating({min: -73, max: 46})}}"
  ],
  "pop": "{{chance.integer({min: 0, max: 65046})}}",
  "state": "{{chance.word()}}"
}
```

Listing 35: Example of the generated template

plate written inside the double curly braces `{{}}`. These parts of the template are compiled into evaluable JavaScript functions using the underscore.js library [70]. It is therefore possible to inject any kind of JavaScript code into template if the data generator references those functions.

Current implementation supports only functions from Chance and faker.js libraries and utility methods that are part of the generator.

The use of JavaScript and this particular type of template evaluation enables further enhancements of the capabilities of the data generator without changes in the core of the data generator.

The following list summarizes the capabilities of the data generator:

- The data generator can generate basic JSON data types.
- The data generator can generate random data using third party random data generators.
- The data generator provides type conversion specifically for MongoDB (Date, Number, ObjectID, Timestamp) so the generated value stored in MongoDB

is not just string.

- It is possible to use `this` keyword in template expressions to reference other generated values.
- It is possible to hide or show fields based on a JavaScript expression.
- It provides the following utility methods:
  - `_$size` - the total number of generated docs in the current run
  - `_$index` - the index of the current document, starting from 0
  - `counter([id], [start], [step])` - the underlying counts are accessible anywhere in the outmost document so that it is possible to use the same counter consistently regardless of its position
    - `id` - the index of the counter to use, default is 0
    - `start` - the first count, default is 0
    - `step` - increment of each count, default is 1
  - `util.sample(list, [n])` - chooses n items from the supplied list of values (used in arrays)

We included some of the various options available in templates in Listing 36. The example contains a usage of basic type, random data type, type conversion, usage of the keyword `this` and the field visibility.

```
{
        "basicType" : true, // boolean
        "randomSentence" : "{{ faker.Lorem.sentence() }}", // string
        "typeConversion" : {
                "randomDate" : "{{ Date(chance.date()) }}", // this will become
                // ISODate() in MongoDB
        },
        "this" : {
                "ten": 10,
                "eleven": "{{ Number(this.ten + 1) }}" // this will produce 11
        },
        "fieldVisibility" : {
                "speed": "{{ util.random(40, 200) }}",
                "warning": "Slow down! {{ hide(this.speed < 130) }}"
                // hides the warning if the speed is under 130
        }
}
```

Listing 36: Various options in template

Current implementation generates documents either directly to MongoDB or it can save it to files on disk where each line contains a single JSON document. This format can be easily imported into MongoDB with their `mongoimport` tool. Each of the generated files has different size based on the random number that the client generator chose during the generation process. It would be possible to have this constant but we decided to leave it random so it would behave exactly the same as in the case the output is database.

Document References

The JSON format does not support references to other documents natively. However, MongoDB offers two methods we have already described in Section 4.7.

MongoDB website [71] suggests the use of manual references. Therefore, these references are also supported in our data generator.

MongoDB-Datasets does not support the generation of the references between documents so we had to add this implementation to the node.js application. We created a module `Reference` with function `LinkDocument` which can be used as shown in Listing 37.

```
{{ ObjectID(reference.LinkDocument('referenced DB', 'referenced collection') }}
```

Listing 37: Linking documents

The data generator then chooses a random document from the referenced collection and writes its ObjectId to the referencing document.

In order to support the random document selection, target collection must have a special field `rnd` defined in its template as shown in Listing 38. If the collection was not being generated using this data generator, it is possible to update the documents manually by adding this field with random numeric values (64-bit integers).

The method we described so far will generate references to the whole collection. If we wanted to limit the set of documents which get referenced, it is possible to alter the generation of the `rnd` field so that this field is hidden based on some arbitrary condition. Then the data generator chooses only from this restricted set of documents. An example of this is also shown in Listing 38, where the `rnd` field is hidden for people with the title 'Ing.' so candidates for references will not include these people.

```
{
        "rnd" : "{{NumberLong(chance.natural()}}"
        // ... the rest of the template
        // or optional rnd field
        "rnd": "{{NumberLong(chance.natural())}}{{ hide(this._$title == 'Ing.') }}",
        "_$title": "{{ util.sample(['Ing.', 'Mgr.', 'Bc.', 'MuDr.', 'JuDr.', '']) }}"
}
```

Listing 38: Random field definition

JsonGenerator supports the generation of references only when it is used with MongoDB. Output to files requires that the template contains no references.

JsonGenerator does not support automatic infering of the references between documents when analyzing existing data sets. There is no standard way how references are implemented in JSON documents so only a heuristic method can be used. We suggest the method as described in the following paragraph.

Inferring References First, the user specifies which collections he wants to use for the analysis. The algorithm would analyze each collection. For each collection, the algorithm would mark every field with `ObjectId` type as a possible candidate for a document reference. Then for each document in the collection and each candidate field it would try to get the document with the corresponding `_id` from all the other collections in the analyzed set. If all of the ObjectIds (or a majority) would point to the same collection, the algorithm would assume this is a reference to that particular collection. Corresponding reference in the template would be created. The user could then verify this result and correct it manually, if necessary.

5.3.5 Task Delegation and Parallelism

The important part of our work was to enable parallel data generation. We wanted to have independent data generation units which could be run in many instances. These have to be somehow managed and hence our data generator would have one main control part. We decided to use this as a data generation coordinator.

Then we had to solve the problem of how we would control these data generation units. It is possible to implement custom protocols to send messages and tasks to workers or use a message passing library like MPI.NET [88]. We wanted only a simple mechanism for message passing so we decided to abandon specialized libraries for parallel programs for the purposes of our testing implementation. Another reason was that MPI.NET has not been updated since 2008.

Instead we used a framework for cross-platform interprocess communication called Eneter Messaging Framework [89] which offers several message passing mechanisms and is easy to use. It supports not only the .NET Framework but also Java, Windows Phone, Mono, Android and Javascript platforms.

Supported protocols are for example TCP, WebSocket, HTTP, UDP, Shared Memory, Named Pipes etc. There are also several routing mechanisms available, e.g. Message Bus, Broker, Dispatcher, Router, Load Balancer etc.

We used the broker service to create a publish-subscribe model where clients subscribe to the server and notify it that they are ready to receive tasks. The server then sends messages to all the clients at once for simplicity. The task delegation is solved in application layer.

The whole process has the following steps (it is also shown in simplified form in Figure 5.3 as a sequence diagram):

- The Master application starts (the server) and creates the broker.
- Client applications start and subscribe to several messages that the broker sends. These include the request for the data generation, the hello world message (for testing purposes) and the template synchronization message. The template synchronization is a convenient way to copy the templates to all the client data generators, otherwise the user would have to copy them manually.
- The Master application is informed about the subscribed clients and keeps the list of them.
- Client applications wait for incoming messages.
- When the user of the data generator starts the data generation process and wants to generate 1 000 000 documents (it is necessary to specify the batch size, i.e. the number of documents each client application will generate as a single task), the following happens:
  - The Master creates a queue of tasks based on the total number of documents and the batch size. For example, if we are generating 1

000 000 documents and the batch size is set to 100 000 documents, the Master application creates 10 tasks which are then distributed to the connected clients in the way that depends on the speed of the Client application. If one instance of the Client application runs on a faster server with more CPU cores, it finishes the task sooner and can start processing the next task sooner. The Master sends only 1 task to a client at once. Only after the client responds that the task has been finished, it receives another task to process. This mechanism was implemented to prevent the overload of the Client. Another reason was that it is difficult to predict what computing power there is available for the Client and we might end up waiting for the slowest one.

– Client application registers the task and creates even smaller tasks for the Node.js data generator. MongoDB has an insertion limit of 1000 documents in one batch so we implemented the Client application so that it assigns a random number of documents to a single Node.js data generator with a predefined maximum (we used 10 000 in our tests). Each client application sends as many tasks to Node.js generator as there are CPU cores available. Node.js data generator is implemented so that it creates the same number of worker threads as there are CPU cores. This way we can utilize the most of the CPU.
– After all the tasks have been finished by the Node.js data generator, the Client application generates a response for the Master application and informs it the task has been finished.
– The Master application receives this response and if there are any unfinished tasks, it removes one from the queue and assigns it to this Client application.
– This process is repeated until there are no more tasks to process and the data generation process ends.

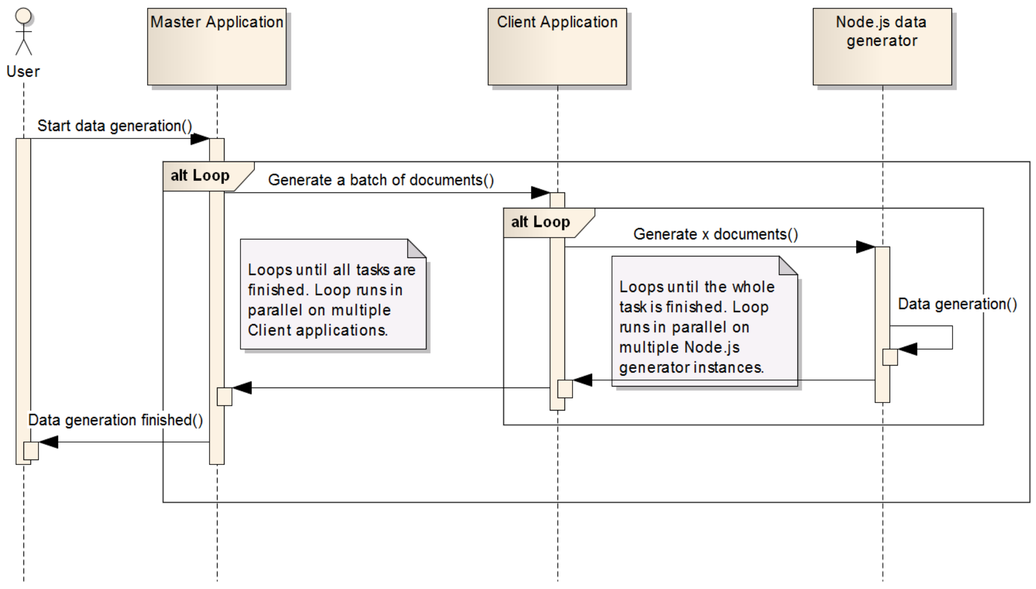

User
Master Application
Client Application
Node.js data generator
Start data generation()
alt Loop
Generate a batch of documents()
alt Loop
Generate x documents()
Loops until all tasks are finished. Loop runs in parallel on multiple Client applications.
Loops until the whole task is finished. Loop runs in parallel on multiple Node.js generator instances.
Data generation()
Data generation finished()

Figure 5.3: Diagram of task delegation in Json Generator

5.4 Implementation Summary

In this chapter we described the architecture of our data generation and also some of the implementation details. The following list summarizes the main points:

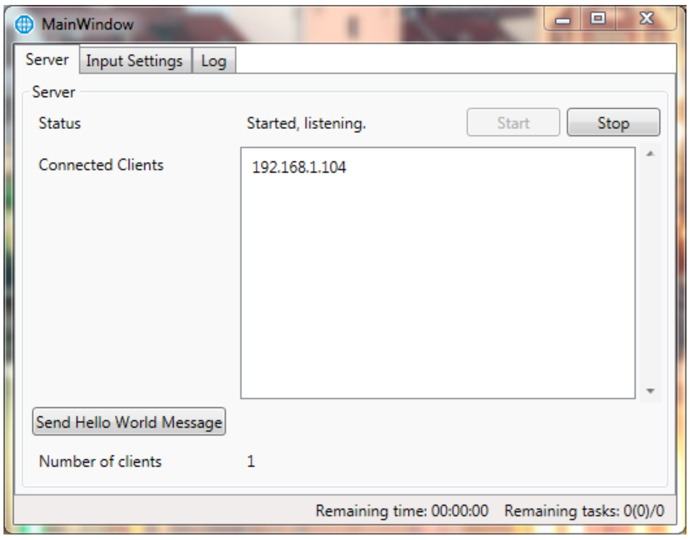
- We created a data generator that can run on multiple servers in parallel and is therefore capable to increase its data generation rate at the expense of adding additional hardware.
- The data generator is capable of infering the schema from the set of existing documents which it transforms to the template that can be then used for data generation.
- The data generation possibilities are very broad due to the use of the JSON templates and random data generation libraries capable of producing various data types.
- The data generator is also capable of creating simple manual references between the documents.
- The data generator can output the data directly to MongoDB or to text files saved on disk.

6. Data Generator User Manual

This chapter contains information about the user interface of the *JsonGenerator*, its installation, usage and configuration options.

6.1 JsonGenerator.Master.WindowApp

`JsonGenerator.Master.WindowApp` is the main control point of the *JsonGenerator*. It is a window application with simple controls which allow users to generate templates from collections in MongoDB and also start the data generation process. Figure 6.1 shows the first tab of the application. It is possible to start and stop the server component of the generator, see the list of connected client generators and also send the Hello world message (which is intended only as a communication test).



Status Started, listening.

Connected Clients 192.168.1.104

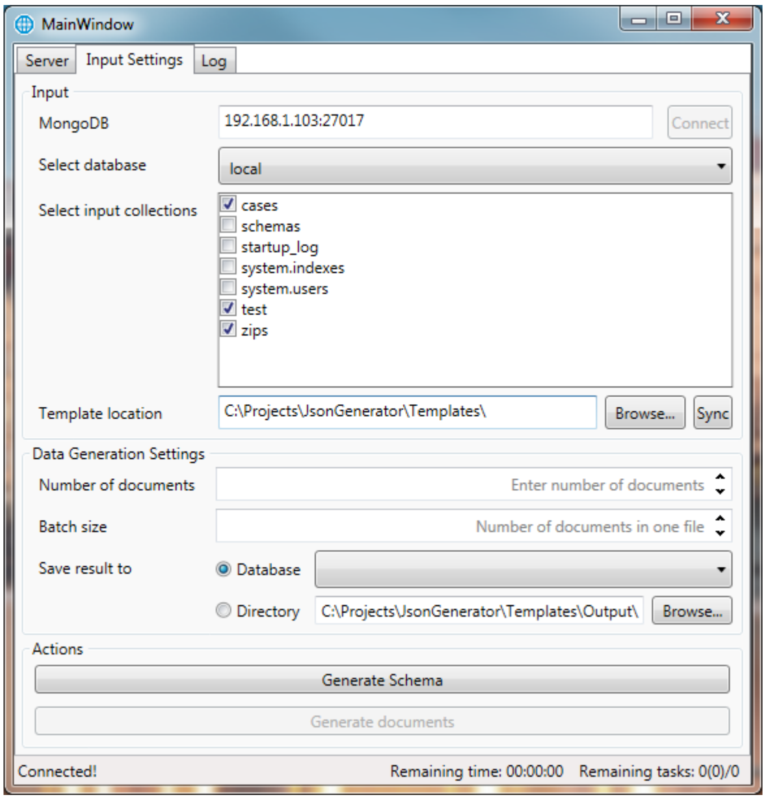
Number of clients 1

Remaining time: 00:00:00 Remaining tasks: 0(0)/0

Figure 6.1: Server tab of JsonGenerator

The *Input Settings* tab shown in Figure 6.2 is divided into two sections. The top part contains the list of databases from the currently connected MongoDB. After the database is selected, the list underneath contains all the collections. It is possible to check some of them and generate their schema, which also generates the template and saves it to the location specified in *Template* location box. The *Sync* button synchronizes the templates to all the connected clients. Is copies all the files with `json` file extension from the current computer to all the connected clients. Before it copies the templates, it also deletes all `json` files previously present in that folder.

The *Data Generation Settings* part of the tab controls the options for data generation. It is possible to specify the total number of documents to generate, the batch size and the target of the data generation (which is either database or a directory). If there are multiple templates present in the *Template* location during the generation process, every template in this location is used for data generation sequentially and the settings specified in the application are applied again. That means that if we specify to generate 1 000 000 documents, the data



MongoDB: 192.168.1.103:27017
Select database: local
Select input collections: [✓ cases] schemas startup_log system.indexes system.users [✓ test] [✓ zips]
Template location: C:\Projects\JsonGenerator\Templates\
Number of documents: Enter number of documents
Batch size: Number of documents in one file
Save result to: [✓ Database] Directory C:\Projects\JsonGenerator\Templates\Output\
Connected! Remaining time: 00:00:00 Remaining tasks: 0(0)/0

Figure 6.2: Input Settings tab of JsonGenerator - template generation

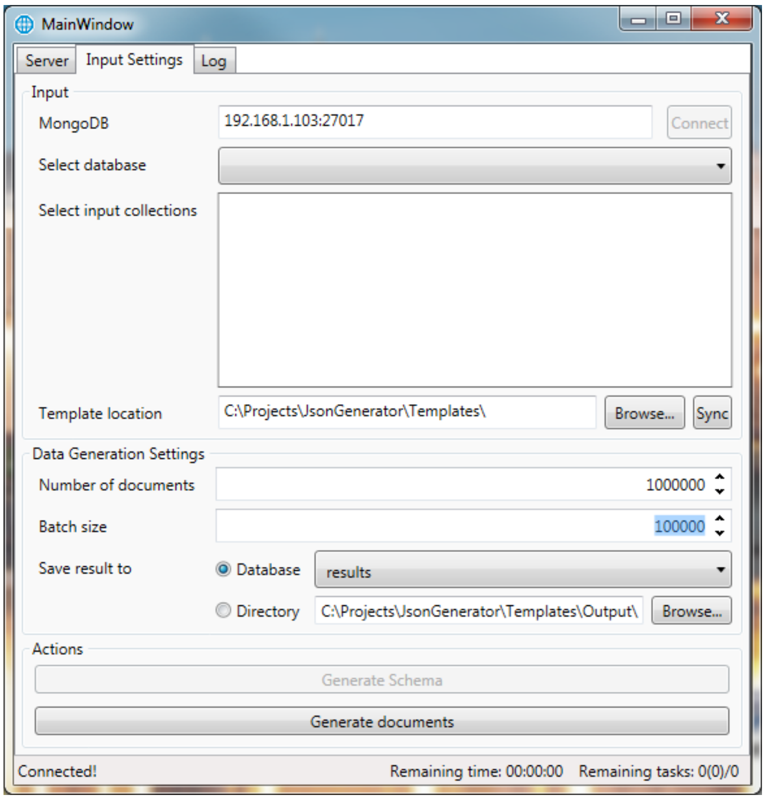
generator will generate 1 000 000 documents for each of the found templates. Figure 6.3 shows the case when data generation is chosen.

Figure 6.4 shows the application while the data generation is in progress. The status bar shows the duration of the last operation on the left side. The next information is the name of the currently processed template, estimated remaining time for the generation using the current template, average time for one batch and the number of remaining tasks (batches) in the format `#RemainingTasks (#CurrentlyExecutingTasks) #TotalTasks`. The last information shows the elapsed time of the data generation process for the current template. All this information is reset after data using each template is generated.

The last tab shows only log information, e.g. messages from the client generators were received, duration of the data generation etc.

6.2 Installation

To install the *JsonGenerator.Master.WindowApp* application, simply copy the binary files present on the attached DVD to the desired location. The application requires .NET 4.5.1 installed on the machine as a prerequisite. The same steps are required for the *JsonGenerator.Master.Generator.Client* application. The Node.js data generator and the schema analyzer requires the installation of the Node.js framework. For development of the JavaScript part, we also had to install Python 2.7. All the used libraries and software is also included on the attached DVD. We also included all the scripts we used for automatic deployment on our cluster.



Input
MongoDB: 192.168.1.103:27017
Select database:
Select input collections:
Template location: C:\Projects\JsonGenerator\Templates\

Data Generation Settings
Number of documents: 1000000
Batch size: 100000
Save result to: [✓ Database] results
Directory: C:\Projects\JsonGenerator\Templates\Output\

Actions

Connected!   Remaining time: 00:00:00   Remaining tasks: 0(0)/0

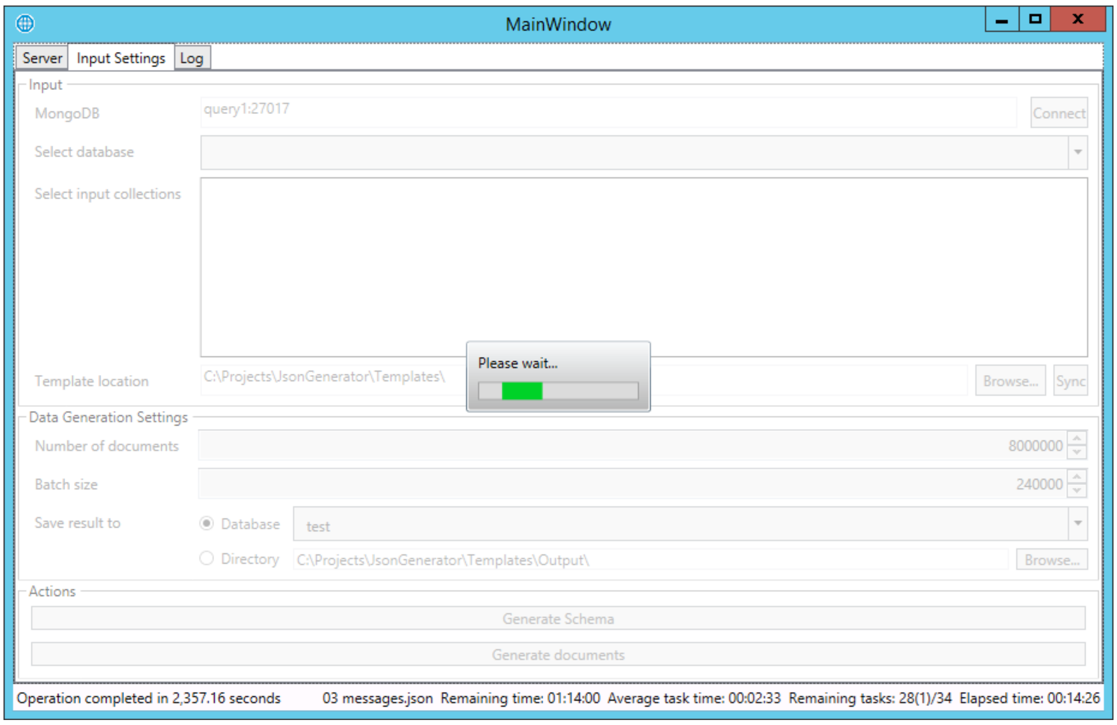
Figure 6.3: Input Settings tab of JsonGenerator - data generation

6.3 Configuration

The configuration of the .NET applications is stored in their `.config` files. For *JsonGenerator.Master.WindowApp* it is the file `JsonGenerator.Master.WindowApp-.exe.config`. Listing 39 shows all available configuration options with the description.

```
<appSettings>
  <!-- template folder chosen after application startup -->
  <add key="DefaultTemplateFolder" value="C:\Projects\JsonGenerator\Templates\"/>
  <!-- output folder chosen after application startup -->
  <add key="DefaultOutputFolder" value="C:\Projects\JsonGenerator\Templates\Output\"/>
  <!-- server port -->
  <add key="server.port" value="8091"/>
</appSettings>
```

Listing 39: Configuration options for JsonGenerator.Master.WindowApp


MainWindow

Server | Input Settings | Log

Input

MongoDB: query1:27017 — Connect

Select database:

Select input collections:

Template location: C:\Projects\JsonGenerator\Templates\ — Browse... — Sync

Please wait...

Data Generation Settings

Number of documents: 8000000

Batch size: 240000

Save result to: [● Database] test

○ Directory: C:\Projects\JsonGenerator\Templates\Output\ — Browse...

Actions

Generate Schema

Generate documents

Operation completed in 2,357.16 seconds   03 messages.json Remaining time: 01:14:00 Average task time: 00:02:33 Remaining tasks: 28(1)/34 Elapsed time: 00:14:26

Figure 6.4: Actual data generation performed during our experiments

```xml
<appSettings>
  <!-- address of the node.js server -->
  <add key="node.js.server.address" value="http://localhost:8081"/>
  <!-- address of the master application -->
  <add key="master.server.ipAddress" value="tcp://10.3.4.110:8091/" />
  <!-- default batch size maximum -->
  <add key="default.batchSize" value="10000" />
  <!-- maximum duration in ms for which the client generator
    waits in case of timeout from node.js before it performs next
    call to node.js server -->
  <add key="node.js.sleepTime" value="60000" /> <!-- 1 minute -->
  <!-- timeout for REST call to node.js server -->
  <add key="restCall.timeout" value="1500000"/> <!-- 25 minutes -->
</appSettings>
```

Listing 40: Configuration options for JsonGenerator.Master.Generator.Client

7. Experiments

JsonGenerator can generate various JSON documents by using templates. In this chapter we show an example of how this generator can be used in conjunction with MongoDB and what it enables us to test. There are, of course, many other ways how to use this data generator.

Our goal in the experiments was to generate different amounts of data for manually written templates. We wanted to see how the varying size influences the performance of several queries we wrote depending on different settings of indices, sharding etc. We were also interested how the number of deployed client generators affects the data generation speed.

We also compare the data generation directly to MongoDB with the data generation to files (7.3).

We performed all the commands and queries using the *Robomongo* [90] application. It is an open-source MongoDB management tool available for multiple platforms (Windows, Linux, Mac OS X).

7.1 Test Cases

We created 3 different templates which model a database of people, their phones and messages sent between them. We decided to create these templates manually because the template builder included in the data generator does not produce different random data generators for various text values.

We also created one template to generate a collection of documents that model a blogging platform. The last template we created models log data. Each of these templates corresponds to one collection. The following list summarizes the generation templates:

- `people`: various personal information about a person, see Listing 41; this template generates the biggest documents from our set of templates.
- `phones`: information about phones with the reference to the owner which points to the collection `people`, see Listing 42; this template was created to test the generation of references and what impact they have on MongoDB performance.
- `messages`: messages between people, again with the reference to the collection `people`, see Listing 43; this template has two references and we are interested how it affects the generation performance.
- `cms`: template for blogging platform where different types of content are stored in the same collection, see Listing 44; this template was created to generate data that could be used in production environment (blog) and we created few sample queries for it.
- `logs`: template for event log data, see Listing 45; this template simulates log data as they could be created in a production environment.

```
{
	"_$db": {
		"sourceDb": "local",
		"sourceCollection": "test",
		"targetDb": "test",
		"targetCollection": "people"
	},
	"rnd": "{{NumberLong(chance.natural())}}",
	"_$title": "{{ util.sample(['Ing.', 'Mgr.', 'Bc.', 'MuDr.', 'JuDr.', '']) }}",
	"title": "{{ this._$title }}{{hide(this._$title == '')}}",
	"firstName": "{{chance.first()}}",
	"lastName": "{{chance.last()}}",
	"fullNameWithTitle": "{{ this.title + ' ' + this.firstName + ' ' + this.lastName }}",
	"dateOfBirth": "{{Date(chance.birthday())}}",
	"e-mail": "{{chance.email()}}",
	"web": "{{chance.domain()}}",
	"address": {
		"street": "{{chance.street()}}",
		"city": "{{chance.city()}}",
		"zip": "{{chance.zip()}}",
		"country": "{{chance.country({ full: true })}}"
	},
	"twitter": "{{chance.twitter()}}",
	"bodyTemperature": "{{ Number(chance.floating({min: 35, max: 42, fixed: 2}))  }}",
	"warning": "Too hot! {{ hide(this.bodyTemperature < 38) }}",
	"personality": {
		"favorites": {
			"number": "{{Number(chance.d100())}}",
			"city": "{{chance.city()}}",
			"radio": "{{chance.radio()}}"
		},
		"violence-rating": "{{Number(chance.d6())}}"
	},
	"friends" : [ "{{_$config}}", { "size": [ 5, 25 ] }, {
		"name": "{{chance.name()}}",
		"phones": [ "{{_$config}}", {}, "{{chance.phone()}}"]
	}]
}
```

Listing 41: Template for people collection

```
{
	"_$db": {
		"sourceDb": "local",
		"sourceCollection": "test",
		"targetDb": "test",
		"targetCollection": "phones"
	},
	"rnd": "{{NumberLong(chance.natural())}}",
	"model": "{{faker.company.companyName()}}",
	"price": "{{Number(faker.commerce.price())}}",
	"phoneNumber": "{{chance.phone()}}",
	"owner": "{{ ObjectID(reference.LinkDocument('results', 'people')) }}"
}
```

Listing 42: Template for phones collection

```
{
	"_$db": {
		"sourceDb": "local",
		"sourceCollection": "test",
		"targetDb": "test",
		"targetCollection": "messages"
	},
	"from":"{{ ObjectID(reference.LinkDocument('results', 'people')) }}",
	"to": "{{ ObjectID(reference.LinkDocument('results', 'people')) }}",
	"message": "{{ chance.paragraph() }}",
	"date": "{{ Date(faker.date.past()) }}"
}
```

Listing 43: Template for messages collection

```
{
        "_$db": {
                "sourceDb": "local",
                "sourceCollection": "cms",
                "targetDb": "test",
                "targetCollection": "cms"
        },
    "nonce": "{{ObjectID(chance.hash({length: 12}))}}",
    "metadata": {
        "type": "{{ util.sample([’basic-page’, ’blog-entry’, ’comment’, ’photo’]) }}",
        "section": "{{ chance.state() }}",
        "slug": "{{chance.word()}}",
        "title": "{{chance.sentence()}}",
        "created": "{{ Date(faker.date.past()) }}",
        "author": {
                        "_id": "{{ ObjectID(reference.LinkDocument(’results’, ’people’)) }}"
                },
        "tags": [ "{{_$config}}", { "size": [ 5, 10 ] }, "{{ util.sample([’abstract photography’,
        ’architectural detail’,’architecture photography’,’astrophotography’,’candid shot’,
        ’food photography’,’glamour photography’,’landscape photography’,’macro photography’,
        ’pet photography’,’portrait photography’,’rural photography’,’sport photography’,
        ’still life photography’,’street photography’,’urban photgraphy’,’wildlife photography’,
        ’back view’,’front view’,’profile’,’side view’,’bokeh’,’lens zoom’,’light trail’,
        ’long exposure’,’panning’,’panorama’,’reflection’,’silhouette’,’star trail’,
        ’vignette’,’close-up’,’environmental’,’full body’,’headshot’,’upper body’,’landscape’,
        ’portrait’,’exterior’,’in-car’,’interior’,’studio’,’black and white’,’color image’,
        ’platinotype’,’selenium toning’,’sepia toning’]) }}"],
        "detail": { "text": "{{chance.paragraph()}}" }
    }
}
```

Listing 44: Template for cms collection

```
{
        "_$db": {
                "sourceDb": "local",
                "sourceCollection": "test",
                "targetDb": "test",
                "targetCollection": "logs"
        },
        "datetime": "{{ Date(faker.date.past()) }}",
        "process": "{{chance.word({syllables: 3})}}",
        "eventDescription": "{{ chance.sentence() }}",
        "duration": "{{ Number(chance.integer({min: 1, max: 500000})) }}",
        "version": "{{ Number(chance.integer({min: 1, max: 100})) }}",
        "level": "{{ util.sample([’Debug’, ’Info’, ’Warning’, ’Error’, ’Fatal’]) }}",
        "keywords": [ "{{_$config}}", {"size": [ 5, 10 ]}, "{{ chance.word({syllables: 3}) }}" ],
        "processId": "{{ Number(chance.natural({min: 1, max: 65536})) }}",
        "threadId:": "{{ Number(chance.natural({min: 1, max: 65536})) }}",
        "computer": "{{ chance.ip() }}",
        "user": "{{ chance.fbid() }}"
}
```

Listing 45: Template for logs collection

We defined 4 different test configurations which alter the creation of collections, sharding settings and indices. The following list summarizes these different scenarios:

- *Test 1*: All the collections are created without any indices except the `rnd` field in `people` collection for random document selection. Sharding is not enabled which means that all the data will be stored on a single server. Listing 46 includes the commands that were used to create the collections in MongoDB for this test.

- *Test 2*: All the collections are created the same way as in *Test 1*. Sharding is enabled using the `_id` field as the shard key. See Listing 47 for the commands used for this test.
- *Test 3*: All the collections were created as in previous cases but we also included hashed indices on their `_id` fields. Sharding is enabled using these fields as shard key with the option `hashed`. This option should give a random distribution of data across the shards and we are interested if this affects either the generation process or the query performance. See Listing 48 for the commands used for this test.
- *Test 4*: All the collections are created the same way as in previous cases. There are several indices created for various fields to suit the some of the test queries for each collection. Sharding is enabled on hashed `_id` fields. See Listing 49 for full commands.

After completion of the data generation for each number of documents and each test, we drop all the collections and create them again. For each generated data set size we executed the queries only once because it does not matter whether the data was created using 1, 2 or 3 instances of the data generator.

```
db.createCollection("people");
db.people.createIndex({rnd: 1});
db.createCollection("phones");
db.createCollection("messages");
db.createCollection("cms");
db.createCollection("logs");
```

Listing 46: Create commands for Test 1

```
db.createCollection("people");
db.people.createIndex({rnd: 1});
db.createCollection("phones");
db.createCollection("messages");
db.createCollection("cms");
db.createCollection("logs");

sh.shardCollection("test.people", {"_id": 1})
sh.shardCollection("test.phones", {"_id": 1})
sh.shardCollection("test.messages", {"_id": 1})
sh.shardCollection("test.cms", {"_id": 1})
sh.shardCollection("test.logs", {"_id": 1})
```

Listing 47: Create commands for Test 2

Each of these tests is performed using 1, 2 and 3 worker generators and for each number of generators we generated 1 000 000 documents for every collection and then we repeated the same process with 8 000 000 documents (as an example of a bigger data set). These numbers were chosen based on our preliminary tests of the data generator with regards to our testing environment.

For every test we measured the generation time, the amount of the generated data and the average object size in database. We also calculated the average data generation rate.

```
db.createCollection("people");
db.createCollection("phones");
db.createCollection("messages");
db.createCollection("cms");
db.createCollection("logs");

db.people.createIndex({rnd: 1});
db.people.createIndex({_id: "hashed"});
db.phones.createIndex({_id: "hashed"});
db.messages.createIndex({_id: "hashed"});
db.cms.createIndex({_id: "hashed"});
db.logs.createIndex({_id: "hashed"});

sh.shardCollection("test.people", {"_id": "hashed"})
sh.shardCollection("test.phones", {"_id": "hashed"})
sh.shardCollection("test.messages", {"_id": "hashed"})
sh.shardCollection("test.cms", {"_id": "hashed"})
sh.shardCollection("test.logs", {"_id": "hashed"})
```

Listing 48: Create commands for Test 3

7.1.1 Test Queries

For every test and every size of the data set we executed the queries shown in Listing 50 three times and measured the execution time. Following list describes these queries:

- `QP1`: Finds people with the `firstName` equal to a certain value. This query contains only one filtering condition and we are interested how the indexing (on/off) and sharding (on/off) affects its performance.
- `QP2`: Finds people with the `title` equal to a certain value. This query was chosen because `title` field is optional.
- `QP3`: Finds people with 10 `friends`. This query was chosen to test the performance in the case where the field is an array.
- `QP4`: Finds people with the field `bodyTemperature` greater than or equal to a certain value. This query was chosen because it queries for a range of values.
- `QP5`: Finds documents having the field `address.country` with a certain value. This query was chosen because it filters documents based on a field inside an object field.
- `QM1`: Finds messages sent `from` a certain person. This query and the next query were selected to test query performance on ObjectId fields.
- `QM2`: Finds messages sent `to` a certain person.
- `QPh1`: Finds phones with the `price` between the specified boundaries.
- `QC1`: Finds documents having all the specified metadata tags. This query tests the query performance when filtering on the contents of an array field.
- `QC2`: Finds the 10 most recent blog entries. This query was selected because it represents a common query in a blogging system.
- `QL1`: Finds the 1000 most recent log entries for the specified date range. This query was also selected because it is a common query for event logs.

```
db.createCollection("people");
db.createCollection("phones");
db.createCollection("messages");
db.createCollection("cms");
db.createCollection("logs");

db.people.createIndex({rnd: 1});
db.people.createIndex({_id: "hashed"});
db.people.createIndex({firstName: 1, lastName: 1});
db.people.createIndex({title: 1});

db.phones.createIndex({_id: "hashed"});
db.phones.createIndex({price: 1});

db.messages.createIndex({_id: "hashed"});
db.messages.createIndex({from: 1});
db.messages.createIndex({to: 1});

db.cms.createIndex({_id: "hashed"});
db.cms.createIndex({'metadata.tags': 1});
db.cms.createIndex({'metadata.created': 1});
db.cms.createIndex({'metadata.type': 1});

db.logs.createIndex({_id: "hashed"});
db.logs.createIndex({datetime: -1});
db.logs.createIndex({level: 1});

sh.shardCollection("test.people", {"_id": "hashed"})
sh.shardCollection("test.phones", {"_id": "hashed"})
sh.shardCollection("test.messages", {"_id": "hashed"})
sh.shardCollection("test.cms", {"_id": "hashed"})
sh.shardCollection("test.logs", {"_id": "hashed"})
```

Listing 49: Create commands for Test 4

7.1.2 Testing Environment

The testing environment we used was composed of 12 virtual servers with the following attributes:

- 3 x application server (App1, App2, App3): each with 12 cores (Intel Xeon @ 2.13GHz), 4GB RAM, Windows Server 2012 R2
- 3 x MongoDB configuration server (Config1, Config2, Config3): each with 2 cores (Intel Xeon @ 2.2GHz), 2GB RAM, Windows Server 2012 R2
- 2 x MongoDB query router (Query1, Query2): each with 2 cores (Intel Xeon @ 2.67GHz), 2GB RAM, Windows Server 2012 R2
- 4 x MongoDB database server (Shard1, Shard2, Shard3, Shard4): each with 4 cores (Intel Xeon @ 2.2GHz), 16GB RAM, Windows Server 2012 R2

The deployment was based on the architecture proposed by the authors of MongoDB [86]. This architecture is depicted in Figure 7.1. The diagram of our actual testing environment is shown in Figure 7.2. Every worker created 12 independent threads and each of these threads was generating data and writing it directly to the database in batches of 1000 documents (which is a limit imposed

```
QP1: db.getCollection('people').find({firstName: 'Walter'}).explain("executionStats")
QP2: db.getCollection('people').find({title: 'Ing.'}).explain("executionStats")
QP3: db.getCollection('people').find(
        {friends: {$size: 10 }}).explain("executionStats")
QP4: db.getCollection('people').find(
        {bodyTemperature: {$gte: 38}}).explain("executionStats")
QP5: db.getCollection('people').find(
        {'address.country': 'Czech Republic'}).explain("executionStats")

QM1:db.getCollection('messages').find(
        {'from': ObjectId("55b625c008f843ac53a53ae9")}).explain("executionStats")
QM2: db.getCollection('messages').find(
        {'to': ObjectId("55b625c008f843ac53a53ae9")}).explain("executionStats")

QPh1: db.getCollection('phones').find(
        {'price': {$lte : 400, $gte: 200 }}).explain("executionStats")

QC1: db.getCollection('cms').find(
        {'metadata.tags': { $all:  ['portrait', 'landscape', 'exterior']}})
        .explain("executionStats")
QC2: db.getCollection('cms').find(
        {'metadata.type':'blog-entry'})
        .sort({'metadata.created':-1})
        .limit(10).explain("executionStats")

QL1: db.getCollection('logs').find(
        {'datetime': {$gte: ISODate("2015-07-01T00:00:00.000Z"),
        $lt: ISODate("2015-07-27T00:00:00.000Z")},'level':'Error'})
        .sort({'datetime':-1}).limit(1000).explain("executionStats")
```

Listing 50: Test queries used in the experiment

by MongoDB). That means that we were able to generate data at most in 36 threads at the same time.

Servers App1, App2 and App3 hosted the Client application together with the Node.js data generator and schema analyzer. We controlled and monitored the data generation proces from the App1 server where we executed the Master application.

The rest of the servers were all part of the MongoDB cluster and we did not modify it during the tests. We used MongoDB version 3.0.4. All insert operations used acknowledged write concern [91].

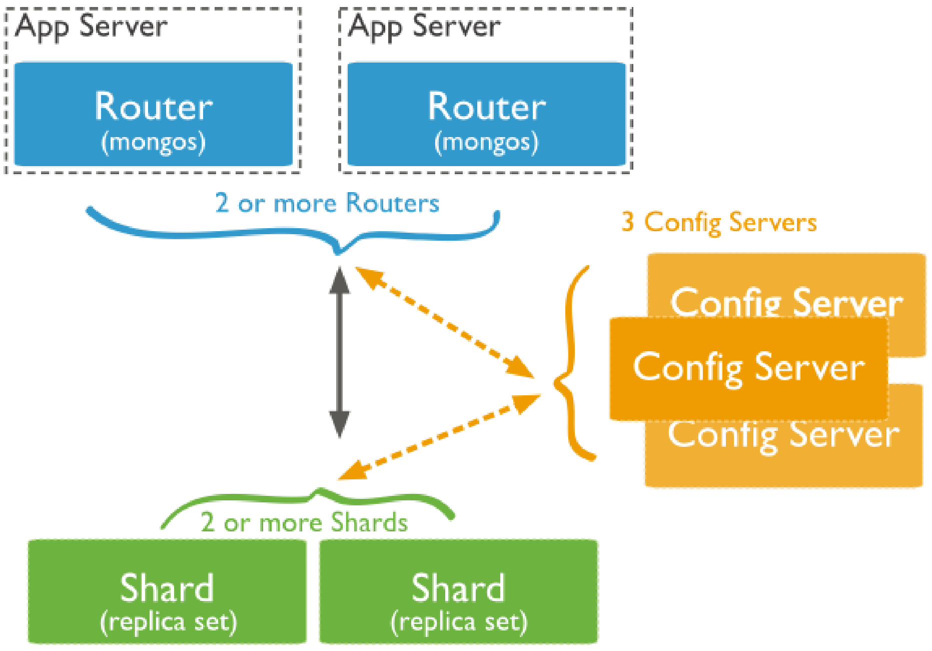


Figure 7.1: Sharded cluster architecture

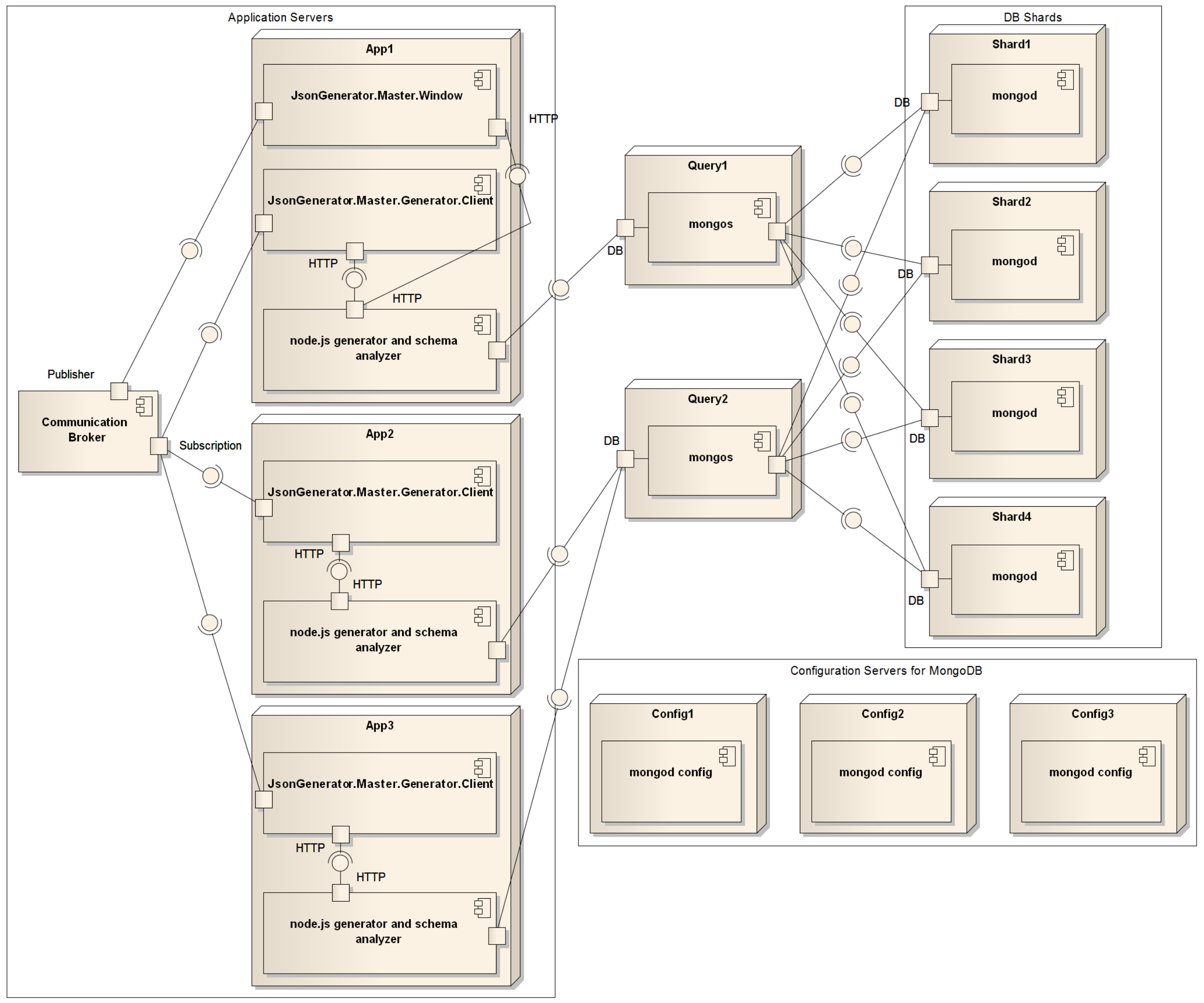

Application Servers
App1
JsonGenerator.Master.Window
HTTP
JsonGenerator.Master.Generator.Client
HTTP
HTTP
node.js generator and schema analyzer
Publisher
Communication Broker
Subscription
App2
JsonGenerator.Master.Generator.Client
HTTP
HTTP
node.js generator and schema analyzer
App3
JsonGenerator.Master.Generator.Client
HTTP
HTTP
node.js generator and schema analyzer
Query1
mongos
DB
Query2
mongos
DB
DB Shards
Shard1
mongod
DB
Shard2
mongod
DB
Shard3
mongod
DB
Shard4
mongod
DB
Configuration Servers for MongoDB
Config1
mongod config
Config2
mongod config
Config3
mongod config

Figure 7.2: Testing deployment diagram

7.2 Test Results

This section summarizes the results of the data generation we performed on our testing environment. During all these tests, the maximum batch size in the *JsonGenerator.Master.Generator.Client* application was set to 10 000.

Before we show the results, we are going to summarize the tests we performed:

- *Test 1*: The purpose of this test was to generate several collections in MongoDB. There were no custom indices created (except one on `rnd` field) and sharding was disabled. That means all the data was stored on a single server. All the operations and queries were services by a single `mongod` instance.
  We were interested in the data generation speed rate and how the server responds to the queries depending on the size of the generated data and the type of the query.
- *Test 2*: This test differs from Test 1 in that we enabled sharding for all the generated collections using the field `_id` as the sharding key. We were again interested in the data generation speed rate and the performance of the queries.
- *Test 3*: This test used hashed indices for `_id` fields and used this as a shard key. Hashed shard key should enforce random distribution of data across the shards and we were again interested how this affected our generator and the performance of the queries.
- *Test 4*: The last test we performed used additional indices that we created for some of the queries we then executed. We were interested in how the creation of the indices affected the data generation performance and also the query performance.

7.2.1 Generation of 1 000 000 Documents

Figure 7.3 summarizes the generation of 1 000 000 documents using a single client generator (i.e. 12 threads). Figure 7.4 shows the results for the case when two client generators were running (i.e. 24 threads) and Figure 7.5 shows the same results for three client generators (i.e. 36 threads). We used the same scale on the y axis so the generation times can be compared visually.

It is obvious that addition of another client generator increased the data generation rate. The biggest difference was between using 1 and 2 client generators (the data generation was approximately twice as fast). Addition of another client data generator sometimes even made it worse as it probably overloaded the database. There were cases where it did improve the generation rate but it was not very significant.

Sharding and indices influenced the data generation rate slightly when it slowed down the data generation in majority of our tests.

Data Generation Rate

Table 7.1 contains the computed generation rates for each test we performed while generating 1 000 000 documents. The maximum rate at which the data generator was able to produce data was around 10MB/s (1 generator, 12 threads).

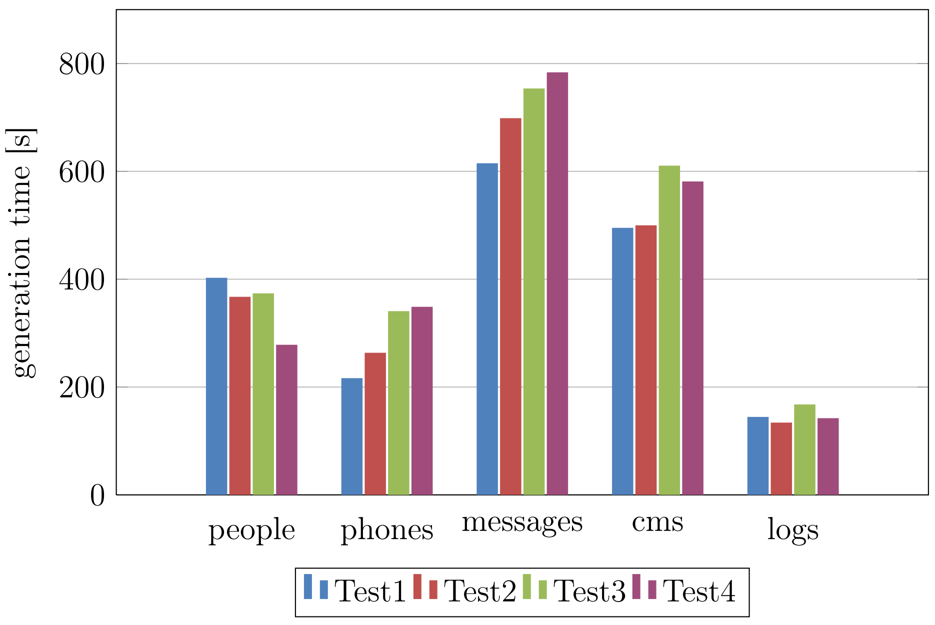


Figure 7.3: 1 000 000 documents, 1 client generator

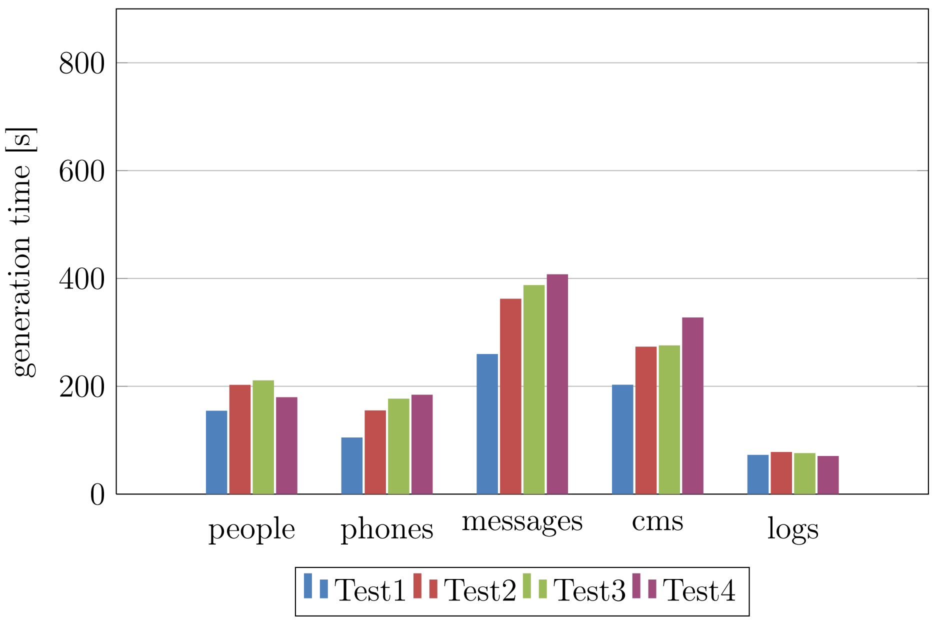


Figure 7.4: 1 000 000 documents, 2 client generators

When the generation process included also references, this rate dropped significantly - in some cases even under 1MB/s.

Additional generator (additional 12 threads) raised the maximum capacity to almost 20MB/s. It also improved the generation with references - it roughly doubled this rate. The third data generator did not improve the overall data generation speed. In some cases, it was even worse.

We also monitored the load on the database server and it seemed that 2 generators were enough to fully load it.

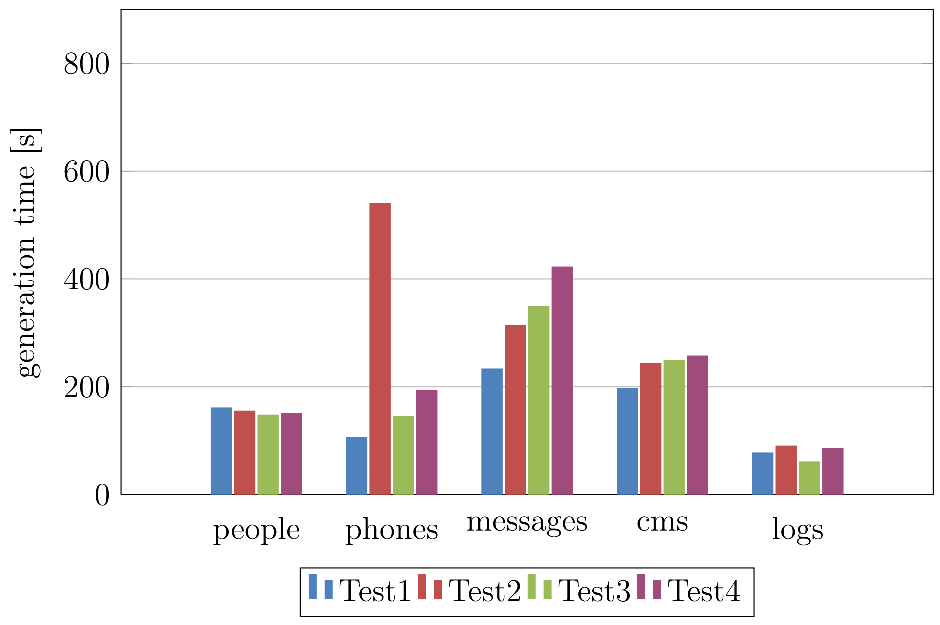


Figure 7.5: 1 000 000 documents, 3 client generators

Collection Name	# of generators	Test 1 [kB/s]	Test 2 [kB/s]	Test 3 [kB/s]	Test 4 [kB/s]
people	1	7909.96	8675.53	8523.13	11442.72
phones	1	1082.66	889.3	687.86	671.9
messages	1	1266.35	1114.56	310.93	993.93
cms	1	2509.14	2485.38	2034.24	2137.06
logs	1	3391.48	3658.4	2924.32	3447.86
people	2	20593.85	15706.42	15096.75	17707.92
phones	2	2230.02	1508.87	1323.54	1271.07
messages	2	2997.97	2149.78	2009.67	1909.62
cms	2	6122.13	4542.10	4504.3	3792.91
logs	2	6734.01	6276.57	6445.93	6935.54
people	3	19700.85	20447.12	21470.02	21005.17
phones	3	2186.95	433.42	1604.74	1206.18
messages	3	3328.29	2476.22	2223.03	1841.14
cms	3	6282.85	5082.09	4981.60	4814.65
logs	3	6270.25	5342.91	7959.14	5678.19

Table 7.1: Data generation rates for 1 million documents

Query Performance

We measured the execution of the sample queries we created and already described. Summarization of the measured values is present in Figure 7.6, Figure 7.7, Figure 7.8 and Figure 7.9.

If we compare the query performance between *Test 1* and *Test 2*, we can see that the sharding helped to speed up the query execution time. It was roughly twice as fast. Query performance measured after *Test 3* also showed overall improvement. An interesting finding, however, is that queries we executed after *Test*

4 were slower in cases when there were no indices created for the queries. Query execution times for fields which were indexed were much faster (the bar in graph is almost not visible).

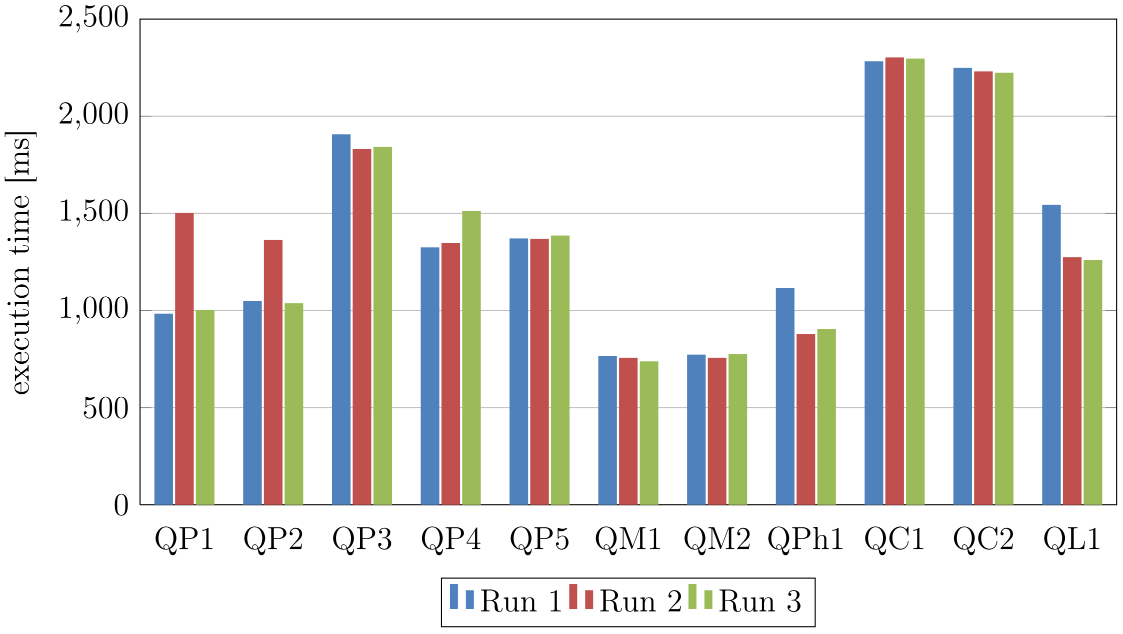


Figure 7.6: Query performance, 1 000 000 documents, Test 1

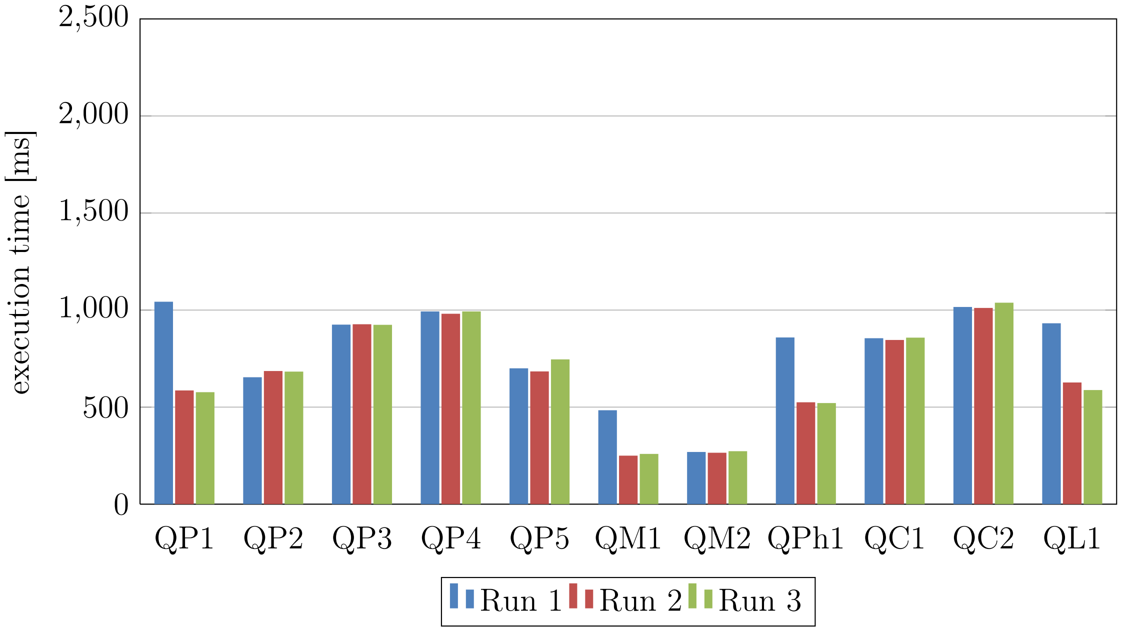


Figure 7.7: Query performance, 1 000 000 documents, Test 2

7.2.2 Generation of 8 000 000 Documents

The next part of the test was to repeat the whole testing process again with the increased number of documents. We used 1, 2 and 3 instances of the client data generator.

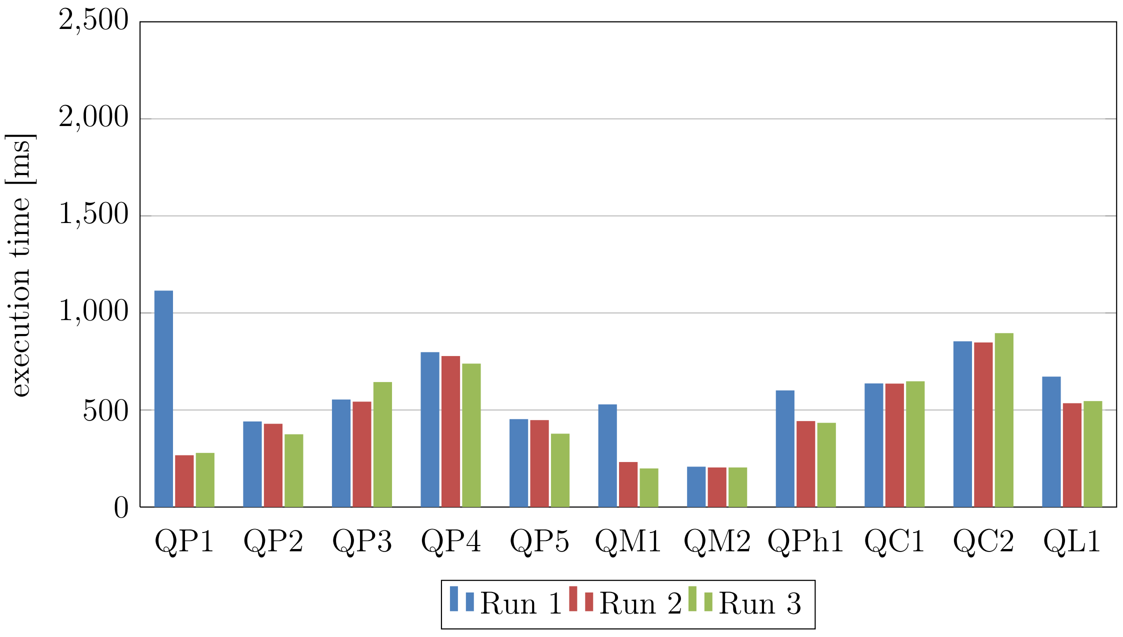


Figure 7.8: Query performance, 1 000 000 documents, Test 3

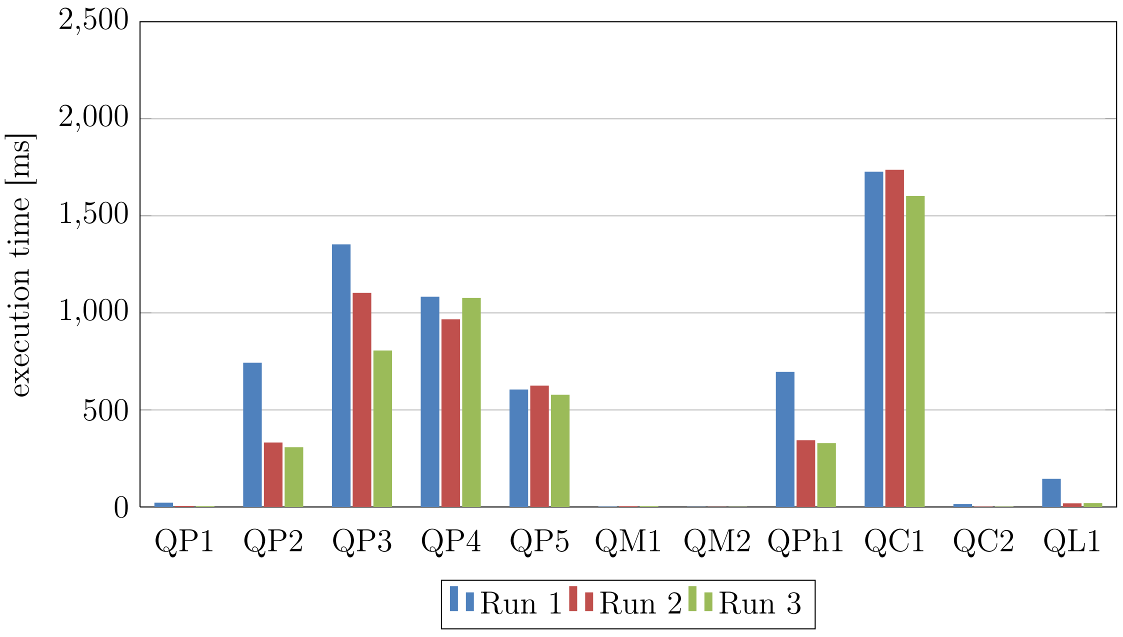


Figure 7.9: Query performance, 1 000 000 documents, Test 4

Graphs in Figure 7.10, Figure 7.11 and Figure 7.12 show an improvement in data generation time when 2 and 3 client data generators were used. It is also obvious that turning on the sharding had a positive impact on data generation which involved references between documents. In other cases the data generation rate was almost identical.

Data Generation Rate

Table 7.2 contains the computed generation rates for each test we performed while generating 8 000 000 documents. The maximum rate at which the data generator was able to produce data was around 10MB/s (1 generator, 12 threads). When the generation process included also references, this rate dropped significantly - in some cases even under 1MB/s.

Additional generator (additional 12 threads) raised the maximum capacity to almost 18MB/s. It also improved the generation with references - it roughly doubled this rate. The third data generator improved the data generation speed in some cases (collection people, around 30% increase) but overall it was not significant.

We also monitored the load on the database server and it seemed that 2 generators were enough to fully load it again.

Collection Name	# of generators	Test 1 [kB/s]	Test 2 [kB/s]	Test 3 [kB/s]	Test 4 [kB/s]
people	1	10985.87	11256.17	10376.65	10555.90
phones	1	370.76	881.27	799.05	795.45
messages	1	437.79	1308.48	1191.40	1172.61
cms	1	925.74	2948.31	2699.86	2495.79
logs	1	2859.65	4875.22	3869.69	2795.47
people	2	18329.13	18740.57	18157.37	15406.84
phones	2	683.22	1689.13	891.12	1258.59
messages	2	870.48	2155.16	1785.5	1654.41
cms	2	1878.01	4711.63	3855.71	3239.07
logs	2	5477.69	7930.85	4997.10	3648.84
people	3	24721.83	23491.77	23265.31	22982.75
phones	3	711	1916.84	819.21	1012.84
messages	3	1078.68	2861.74	2362.22	1966.26
cms	3	2478.05	5884.23	5179.5	4192.38
logs	3	10934.73	8157.5	6593.96	4689.78

Table 7.2: Data generation rates for 8 million documents

7.3 Additional Tests

As a final test, we were interested in data generation performance when generating small and large files and also how it depends on its target place, i.e. whether we save results to MongoDB or to files.

We performed the generation using 1, 2 and 3 client generators again. We used template `logs` for small files and a new template we called `people_large` (see Listing 51) for larger files. We generated 10 000 000 files to MongoDB using the settings of the Test 3 (i.e. hashed indices and sharding using the hashed index). The results of our measurements are shown in Figure 7.17 for the template `logs`. Figure 7.18 contains the results for the `people_large` template. During all these tests, the maximum batch size in the *JsonGenerator.Master.Generator.Client* application was set to 5 000 due to available RAM on our application servers.

Our tests shown that the data generation to files is slightly faster than the data generation directly to MongoDB. However, if we wanted to use the generated

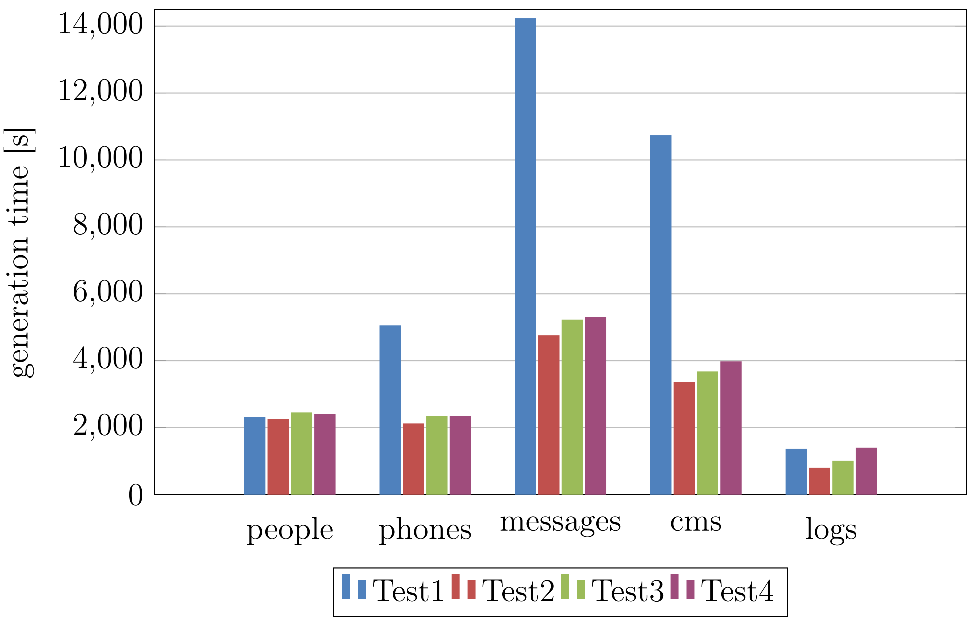


Figure 7.10: 8 000 000 documents, 1 client generator

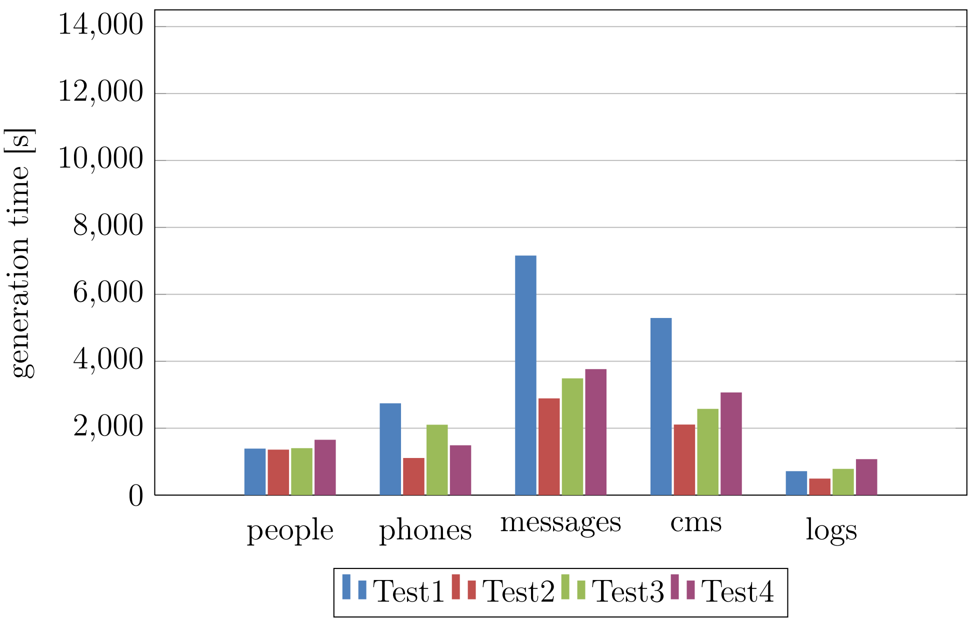


Figure 7.11: 8 000 000 documents, 2 client generators

files in database, we would have to import it anyway, which would take some time. This method could also be used if the target database was not MongoDB.

When using 3 client generators, the overall generation time improved as well. However, when we generated data to files, the generation time was even shorter, as is obvious from the chart.

7.4 Encountered Difficulties

During our experiments we encountered problems with the performance of MongoDB.

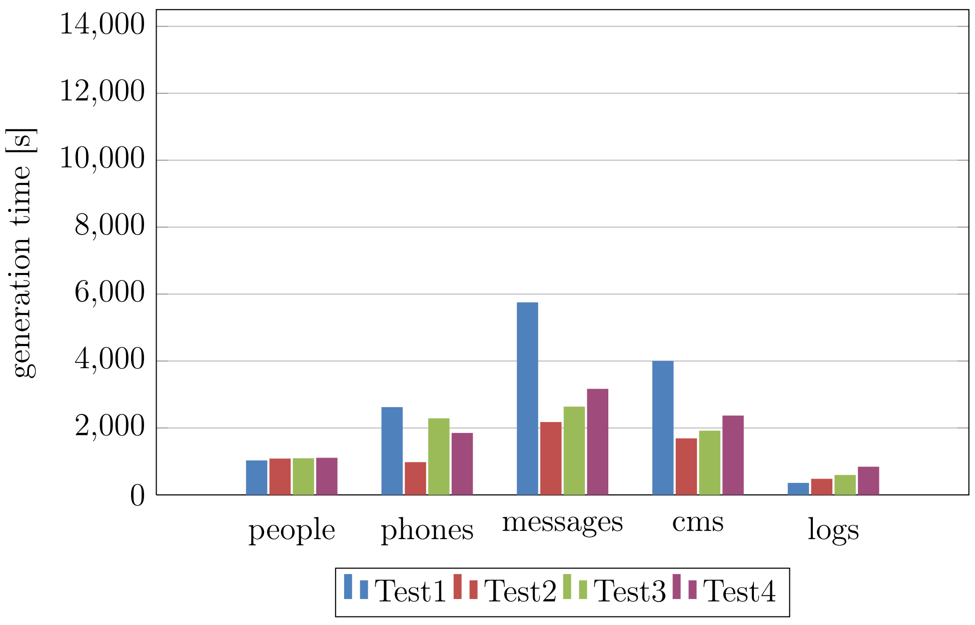


Figure 7.12: 8 000 000 documents, 3 client generators

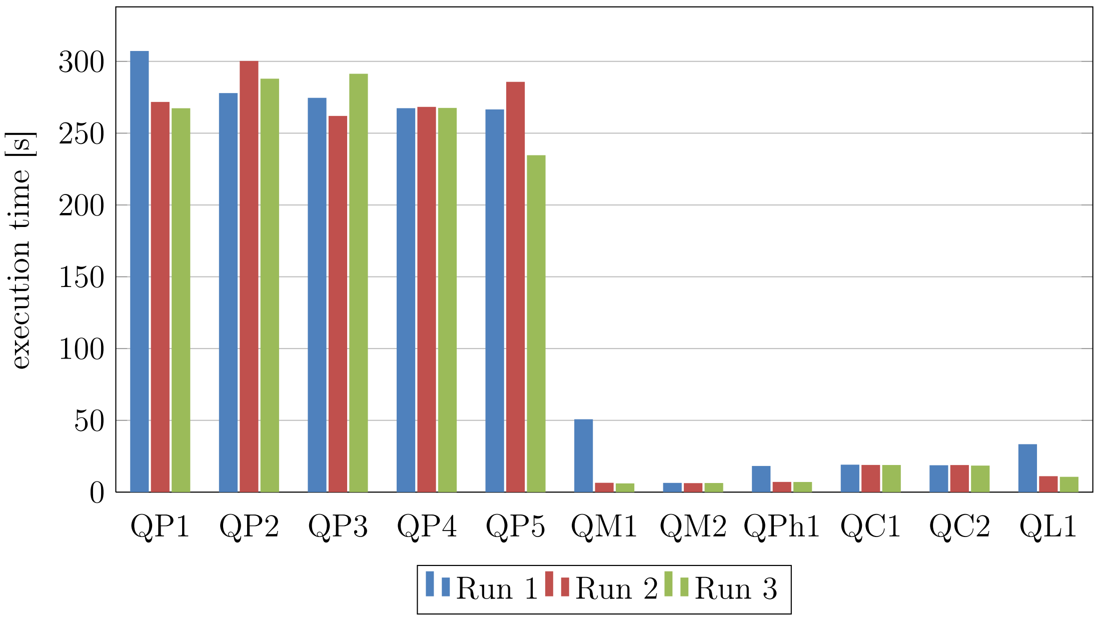


Figure 7.13: Query performance, 8 000 000 documents, Test 1

Our data generator can possibly run on many servers concurrently. We executed it on 1 to 3 servers at the same time. When there were 3 instances of the data generator running concurrently during Test 2, we were able to overload the database so it often did not respond even after 10 minutes, which was our initial timeout for HTTP requests set in Node.js servers. We later increased this timeout to 20 minutes. We found out that if we generated the first collection and let the database redistribute the data, then our data generator did not overload the database. We also restarted query router servers which also seems to help. We did not observe this behaviour in any other tests.

Our suggestion for this problem would be to generate one collection at a time,

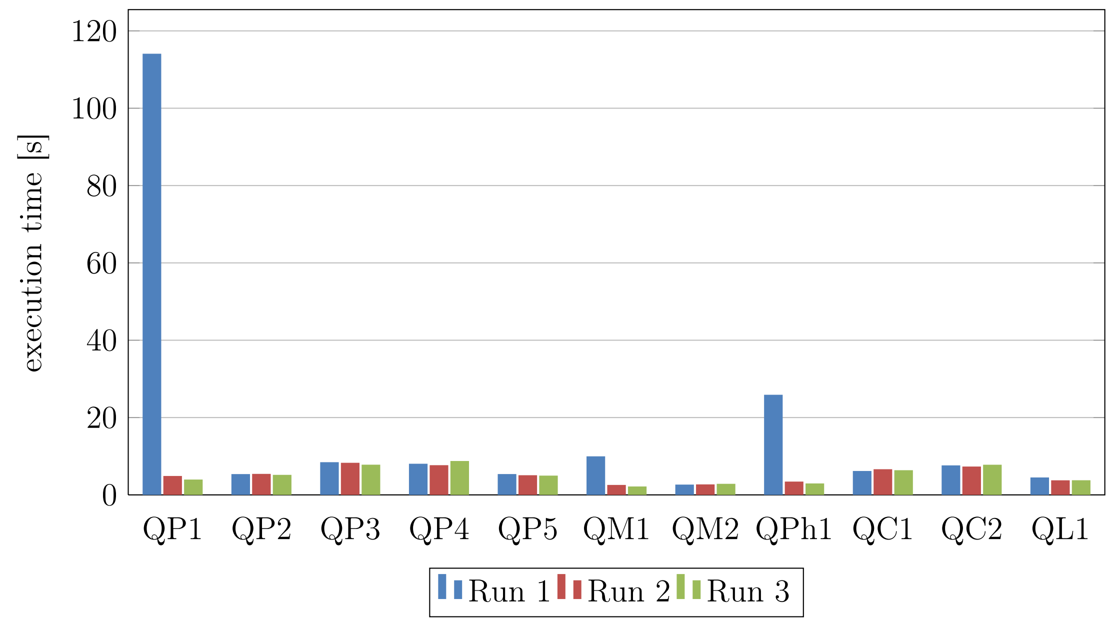


Figure 7.14: Query performance, 8 000 000 documents, Test 2

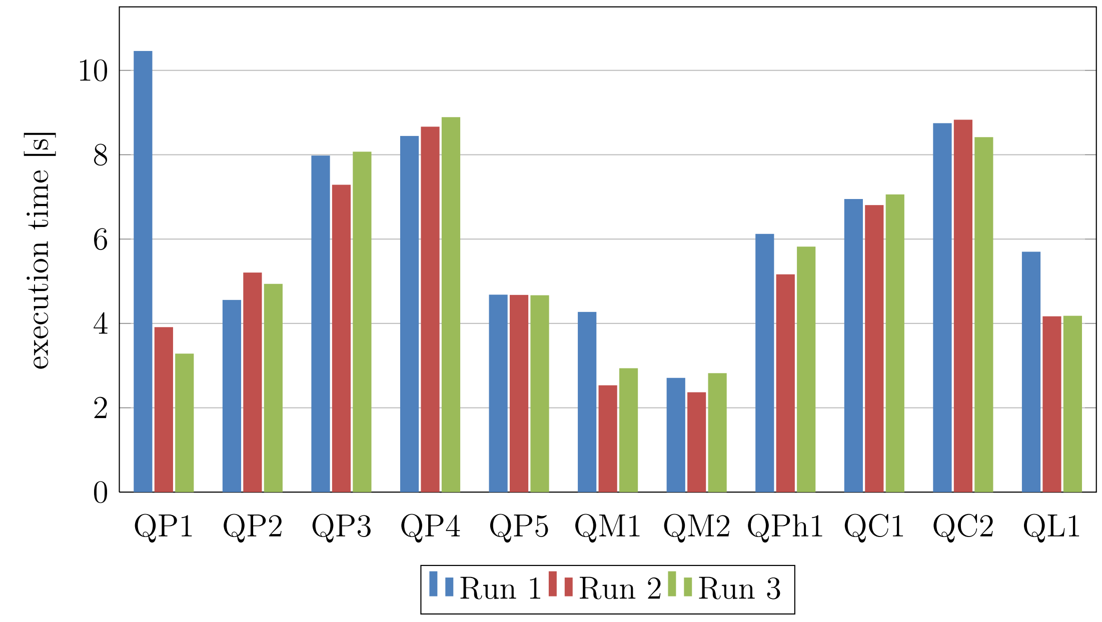


Figure 7.15: Query performance, 8 000 000 documents, Test 3

mostly when there is a template with generation of references.

Another option we tried was to change the write concert during inserts. With *acknowledged* write concern (i.e. `mongod` confirms that it received the write operation and applied the change to the in-memory view of data), the performance of the insertion deteriorated when using more data generators. After clean startup of the database, the insertion performance remained similar until all the available RAM on the shards was filled up. After that we observed longer response times from MongoDB. When we changed the write concern to *unacknowledged* (i.e. `mongod` does not confirm the receipt of write operations), the data generator was not being slowed down by the database that much. For example the insertion of

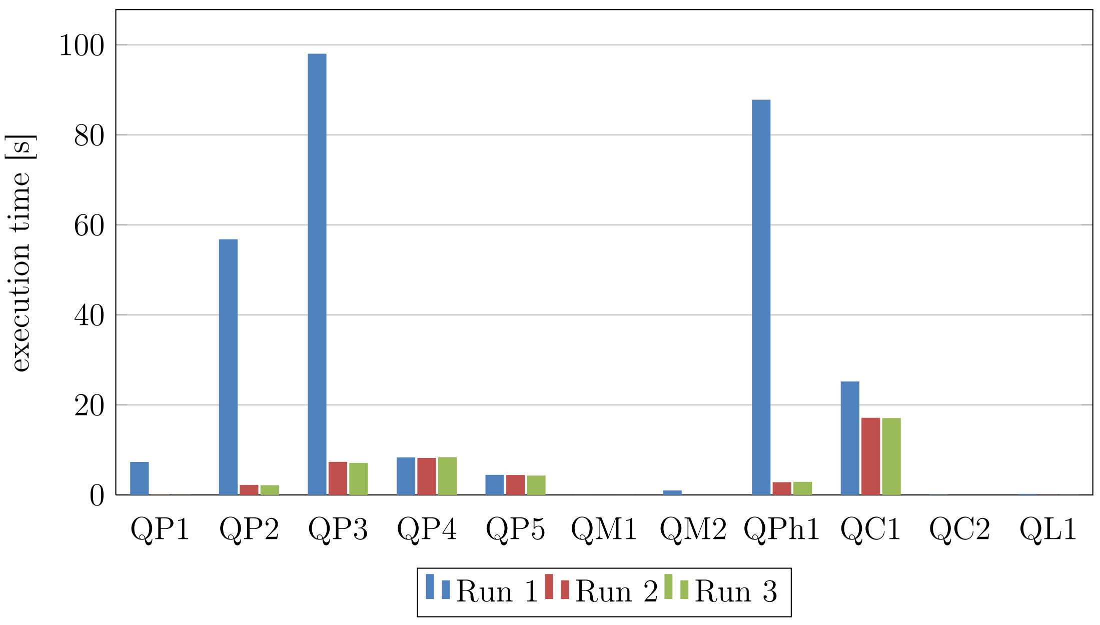


Figure 7.16: Query performance, 8 000 000 documents, Test 4

10 000 000 documents with *acknowledged* write concern using 2 data generators took almost twice as long as when the write concern was set to *unacknowledged*. However, the database was performing inserts and was moving data even after the data generation ended.

7.5 Test Summary

Our experiments showed us the limits of the implemented data generator. We compared the performance of different numbers of instances of the client worker generators under various conditions. Since our data generator can scale horizontally, we were able to test the limits of the database as well. References between documents proved to be a performance hog. This part of the generator is a good candidate for optimizations. A simple example of such optimization could be limiting the number of documents that are used in references which is possible, as we have shown in Listing 38.

We also tested the performance of MongoDB with and without the sharding enabled and we found out that sharding indeed improves the overall performance. It also helps with the query performance. The biggest positive impact on query performance had indices. In most of the cases, the presence of indices slowed down the data generation rate slightly which is understandable due to additional computations the database has to do.

If we wanted to execute similar tests with the generators already available, it would be almost impossible. To the best of our knowledge, currently there is no solution that can be deployed in the similar manner and that could generate structures based on a JSON template either generated automatically or written manually. Many of the XML data generators simply were not created for this type of usage scenario. None of them can run on a cluster and scale like our solution. Similarly, described JSON data generators are web applications and these are not

```
{
        "_$db": {
                "sourceDb": "local",
                "sourceCollection": "test",
                "targetDb": "test",
                "targetCollection": "people"
        },
        "bodyTemperature": "{{Number(chance.floating({min: 35, max: 42, fixed:2}))}}",
        "warning": "Too hot! {{ hide(this.bodyTemperature < 38) }}",
        "array" : [ "{{_$config}}", { "size": [ 5, 50 ] }, {
                "name": "{{chance.name()}}",
                "accounts": [ "{{_$config}}", {}, "{{faker.finance.accountName()}}"]
        }],
        "longText" : "{{ faker.lorem.paragraphs(10) }}",
        "web" : {
                "color": "{{chance.color()}}",
                "domain": "{{chance.domain()}}",
                "email": "{{chance.email()}}",
                "fbid": "{{chance.fbid()}}",
                "googleanalytics": "{{chance.google_analytics() }}",
                "hashtag": "{{chance.hashtag() }}",
                "ip": "{{chance.ip() }}",
                "ipv6": "{{chance.ipv6() }}",
                "klout": "{{chance.klout() }}",
                "twitter": "{{chance.twitter() }}",
                "url" : "{{chance.url()}}"
        },
        "commerce" : {
                "color": "{{faker.commerce.color()}}",
                "department": "{{faker.commerce.department()}}",
                "productName": "{{faker.commerce.productName()}}",
                "price": "{{faker.commerce.price()}}",
                "productAdjective": "{{faker.commerce.productAdjective()}}",
                "productMaterial": "{{faker.commerce.productMaterial()}}",
                "product": "{{faker.commerce.product()}}"
        },
        "company" : {
                "suffixes":"{{faker.company.suffixes()}}",
                "companyName":"{{faker.company.companyName()}}",
                "companySuffix":"{{faker.company.companySuffix()}}",
                "catchPhrase":"{{faker.company.catchPhrase()}}",
                "bs":"{{faker.company.bs()}}",
                "catchPhraseAdjective":"{{faker.company.catchPhraseAdjective()}}",
                "catchPhraseDescriptor":"{{faker.company.catchPhraseDescriptor()}}",
                "catchPhraseNoun":"{{faker.company.catchPhraseNoun()}}",
                "bsAdjective":"{{faker.company.bsAdjective()}}",
                "bsBuzz":"{{faker.company.bsBuzz()}}",
                "bsNoun":"{{faker.company.bsNoun()}}"
        }
}
```

Listing 51: Template for large people collection

suitable for testing large datasets like we performed.

Big Data generators like PDGF [57] or BDGS [58] can scale horizontally and generate data in parallel. It was one of the goals in their design. However, they lack the simplicity in how to change the structure of the generated data, they

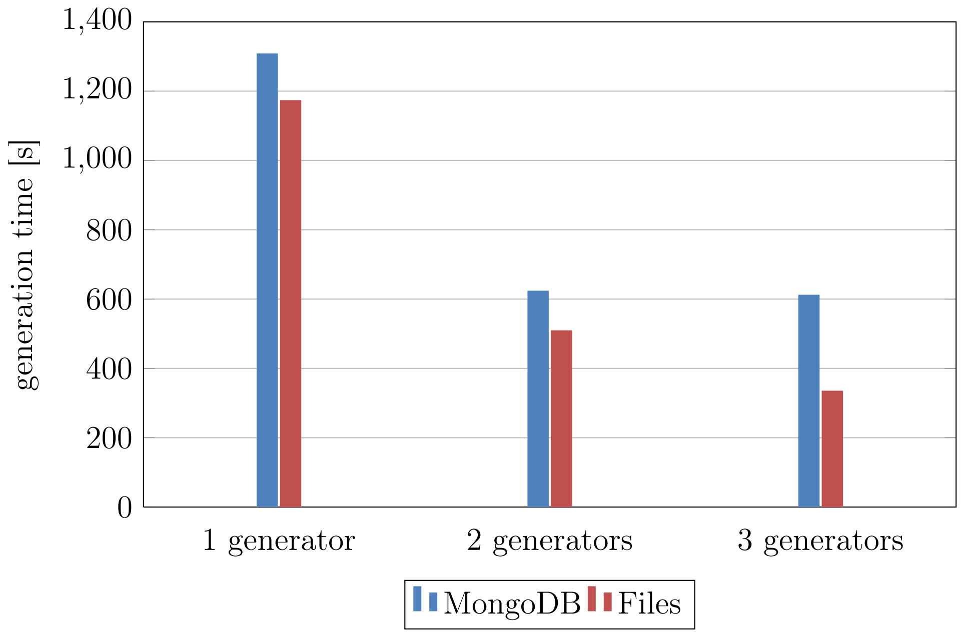


Figure 7.17: 10 000 000 documents, logs template

focus on specific use cases and also lack the inferring abilities. Their advantage is also their data generation speed. Experiments done for BDGS used one node with two 2.4GHz Intel Xeon E5645 processors (i.e. 12 threads) and 32GB RAM with 8X1TB disk. Their data generation rates ranged from 34MB/s (Facebook Data Set) to up to 78MB/s (Amazon Data Set). Our data generator could generate data at a maximum rate around 22MB/s (with 24 threads). However, at most cases it was slower. This could be affected also by the performance of the hardware on which we executed the tests. We used virtual servers and we observed performance issues with disks. This was beyond our control. However, if one wanted to improve the data generation speed with hardware, we suggest the usage of dedicated SSD drives for the database.

The last set of tests we performed were defined to compare the effect of MongoDB on data generator in case of smaller and larger documents. We were also interested how the data generation rate depends on the target of the data generation, i.e. whether it is MongoDB or files on the disk. We found out that the data generation of smaller files is slower and it is also slower when the output is MongoDB. This suggests the use of more smaller shards if the desired target place is MongoDB. If the target is not MongoDB, we recommend using the data generation to files and using an import tool of the desired database.

We also found out that write concern for inserts had an impact on the performance of inserts from the point of the data generator. *Unacknowledged* write concern provides better performance for the data generator but does not ensure the data will be written to disk. However, for the duration of the data generation, this might not be an issue.

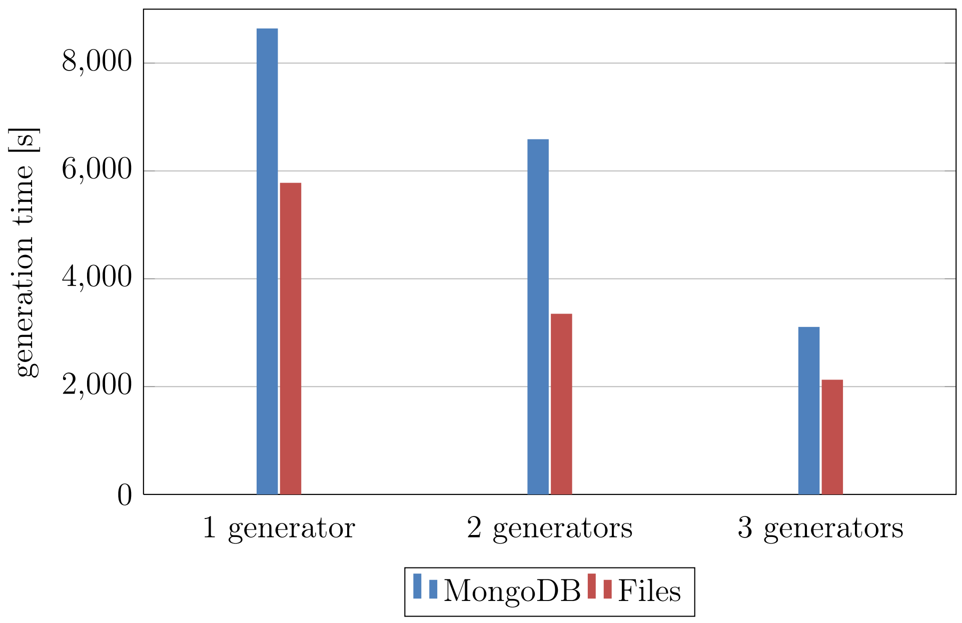


Figure 7.18: 10 000 000 documents, people_large template

8. Conclusion

The first aim of the thesis was to perform a research of the currently available methods in synthetic data generation for semi-structured documents. We focused on XML and JSON format and we also described 3 Big Data generators.

For the XML format, we found several data generators from which the majority was a part of an XML benchmark. In general, there are four distinct methods for data generation in the generators we investigated. The simplest one is generation of documents with predefined properties and contents which can be found, for example, in the generator from XMark benchmark [12]. The most complicated approach is present in ToXgene [27]. It uses its own language for definition of the data to generate. This solution is very capable and powerful. It can generate large and mainly very different XML documents. Its main advantage is, however, its capability to handle very small details in the generation process. We consider this to be its biggest disadvantage too. It is because to be able to define such details, it is necessary to learn this language. The next approach is in the data generator called Complex-structured XML data generator [31]. It first generates a tree structure according to the set of parameters and then assigns artificially created element names and textual values. The last found approach uses schema with a small set of parameters to generate sample XML documents. This approach can be found for example in <oXygen/> XML Editor [32] or Altova XMLSpy [36].

For the JSON format, we found three web applications ([62], [63] and [64]) and one Node.js application [65]. They all use similar method for data generation. The user specifies the output in the form of the template which is then transformed into the resulting documents.

The remaining data generators focus on Big Data (PDGF [57], BDGS [58] and DataGenerator [59]). They focus either on relational data (PDGF), on a specific usage scenarios (BDGS) or require additional programming and creation of the model (DataGenerator). None of these tools provides an easy way to generate large amounts of semi-structured documents with an arbitrary contents. They are all designed to produce data in parallel and can scale horizontally unlike all the other XML or JSON data generators we mentioned.

Our second target was to propose our own algorithm which would focus on Big Data generation for document databases, in our case MongoDB. The generator should be able to run in parallel on multiple servers producing different volumes of data at different data generation rates. The next property we wanted was the ability to infer schema of an existing set of documents and automatic creation of a template for our data generator. We also wanted an ability to create references between the generated documents.

We discussed several options how to design an architecture of such a data generator. We chose an existing solution [65] which we modified for our desired purposes. The main advantage of this solution are the templates in which the user can specify the structure of the documents and he can also use various random data generators to create realistic data. One of the main challenges we faced was to make this single-threaded data generator into a multi-threaded one which could be scaled across several nodes of a cluster (i.e. horizontally). We achieved this by dividing the data generator into two main parts - server and client. The server part is responsible for the division and delegation of the work into smaller tasks to the connected clients. We accomplished this by using a simple communication protocol based on a publisher-subscriber model.

When implementing a functionality to reuse existing data sets for data generation, we used an existing tool [85] which creates a probabilistic schema of the collection of documents in MongoDB. We transformed this information into a template which can be further manually adjusted or directly used for data generation in our generator. We implemented only simple mappings between the found data types and random data generators in the template as these can be added later.

MongoDB does not support joins in its queries. The idea in document databases is to have the data in one place. For the cases when this is not possible, there are two options for referencing other documents: manual and DBRefs. The authors of MongoDB suggest the usage of manual references and therefore we focused on this area. We added a special syntax to the templates which defines the target database and collection of the reference. During the data generation phase, the data generator looks up random documents from the specified collection and saves their `_id` fields to the generated document. This also requires an addition of a special attribute to the referenced collection which enables quick selection of the random document.

The next part of the thesis describes the experiments we performed using the created data generator and MongoDB. We created five different templates to model a simple schema where we store information about people, their phones and messages between them. Additional two templates modeled a simple blogging platform and an event log. These were created so that we could analyze the performance of the data generator when generating documents with and without the references under 4 different conditions regarding indices and sharding. We performed these test using 1, 2 or 3 servers generating data. The size of the data sets we generated were set to 1 000 000 and 8 000 000 documents. We measured the duration of the data generation process and also the size of the resulting data so that we could calculate the data generation rate. The last test we performed was designed to compare the performance of the data generation when the result is saved directly to MongoDB and when it is saved to files on the disk.

Our test results showed us that sharding in MongoDB improved the insert and query performance of the database. We also found out that in our environment, the best number of client generators was 2. Additional generator often overloaded the database and caused timeouts. Query performance was improved when we added appropriate indices which we anticipated. These indices also slowed down the generation process but it was insignificant in general. We also showed that MongoDB was slowing down the data generation process. When we generated the same number of documents and saved them to files, the generation process took shorter amount of time.

In general, we showed how the data generator can be used for testing and that to the best of our knowledge there are no similar solutions that could be used with the same small amount of effort. The data generator can create large numbers of JSON documents with an arbitrary content, at various speeds depending on the number of nodes it runs on. It can also help with creation of the template when the user has a sample data available. It is also possible to create links between the generated data.

8.1 Future Work

There are few areas in which our solution can be improved. First of all, our implementation is a prototype which focuses on the core of the problems (i.e. data generation, references, parallelism), less on the overall usability of the application in terms of the user interface etc. For production environment, these issues should be resolved.

Regarding functionality, we observed slow downs of the data generation process due to massive amount of insertions which our testing MongoDB could not handle. It would be possible to add an additional module which would only import the generated data asynchronously with regards to the data generation. Currently it is possible to do this manually. It would be also good to improve the data generation rates when the data generator creates references as these create additional load on MongoDB (as we have proven in our experiments).

Other area where the data generator could be improved is the support for additional document databases and formats like XML. At the moment, it is possible to convert the generated data manually, however an automatic approach would be preferred.

Another interesting area would be to further investigate and optimize the JavaScript data generator to improve its overall performance.

Bibliography

[1] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. World Wide Web Consortium, 2013. Available from: http://www.w3.org/TR/REC-xml/.

[2] *JSON*. ECMA-404 The JSON Data Interchange Standard, 2015. Available from: http://json.org/

[3] A. De Mauro; M. Greco; M. Grimaldi: *What is big data? A consensual definition and a review of key research topics*. AIP Conference Proceedings 1644: 97-104, Madrid, Spain, 2015. Available from: http://scitation.aip.org/content/aip/proceeding/aipcp/10.1063/1.4907823

[4] *NoSQL Databases*. 2015. Available from: http://nosql-database.org/

[5] *JSON: The Fat-Free Alternative to XML*. json.org, 2015. Available from: http://www.json.org/xml.html

[6] I. Mlynkova, K. Toman, J. Pokorný: *Statistical Analysis of Real XML Data Collections*. Technical report 2006/5. Charles University, Prague, Czech Republic, June 2006, 43 pages. Available from: http://www.ksi.mff.cuni.cz/~mlynkova/doc/tr2006-5.pdf

[7] Vogels, W.: *Eventually consistent*. Communications of the ACM 52: 40. 2009. Available from: http://dl.acm.org/citation.cfm?doid=1435417.1435432

[8] U. Nambiar, Z. Lacroix, S. Bressan, M. L. Lee, Y. G. Li: *Efficient XML Data Management: An analysis*. Proceedings of the Third International Conference on E-Commerce and Web Technologies, p. 87–98, 2002, Springer-Verlag, London, UK.

[9] *Introduction to DTD*. W3Schools, 2013. Available from: http://www.w3schools.com/dtd/dtd_intro.asp

[10] *XML Schema Definition Language (XSD)*. World Wide Web Consortium. 2012. Available from: http://www.w3.org/TR/xmlschema11-1/

[11] XML Schema Definition Tool. Microsoft Developer Network (MSDN). 2012. Available from: http://msdn.microsoft.com/en-us/library/x6c1kb0s(v=vs.110).aspx

[12] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse: *XMark: a benchmark for XML data management*, VLDB '02 Proceedings of the 28th international conference on Very Large Data Bases, p. 974–985, 2002, Hong Kong, China.

[13] I. Mlýnková: *XML Benchmarking: Limitations and Opportunities*, technical report no. 2008/1, Charles University, 23 pages, 2008. Available from: http://www.ksi.mff.cuni.cz/~holubova/doc/tr2008-1.pdf

[14] *XQuery 1.0: An XML Query Language (Second Edition)*, World Wide Web Consortium, 2011. Available from: http://www.w3.org/TR/xquery/.

[15] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. *XMark - An XML Benchmark Project: xmlgen - The Benchmark Data Generator*. Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 2003. Available from: http://www.xml-benchmark.org/generator.html.

[16] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. *XMark - An XML Benchmark Project: DTD of XMark Benchmark Data Generator*. Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 2003. Available from: http://www.ins.cwi.nl/projects/xmark/Assets/auction.dtd.

[17] S. Bressan, M. L. Lee, Y. G. Li, B. Wadhwa, Z. Lacroix, U. Nambiar, and G. Dobbie. *The XOO7 Benchmark*. 2002. Available from: http://www.comp.nus.edu.sg/~ebh/XOO7.html.

[18] S. Bressan, M. L. Lee, Y. G. Li., Z. Lacroix, U. Nambiar. *The XOO7 XML Management System Benchmark*, Technical Report. National University of Singapore and Arizona State University. 2002. Available from: http://www.comp.nus.edu.sg/~ebh/XOO7/download/XOO7_TechReport.pdf.

[19] M. J. Carey, D. J. DeWitt, and J. F. Naughton. *The OO7 Benchmark*. SIGMOD Record (ACM Special Interest Group on Management of Data), 22(2):12–21, 1993.

[20] *The XOO7 Benchmark - DTD of XOO7 Benchmark Data Generator*. National University of Singapore, 2013. Available from: http://www.comp.nus.edu.sg/~ebh/XOO7/download/Module.DTD.

[21] T. Bohme and E. Rahm. XMach-1: *A Benchmark for XML Data Management*. Database Group Leipzig, 2002. Available from: http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html.

[22] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. *The Michigan Benchmark*. Department of Electrical Engineering and Computer Science, The University of Michigan, 2006. Available from: http://www.eecs.umich.edu/db/mbench/description.html.

[23] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. *The Michigan Benchmark - Queries*. Department of Electrical Engineering and Computer Science, The University of Michigan, 2006. Available from: http://www.eecs.umich.edu/db/mbench/description.html#queries.

[24] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. *The Michigan Benchmark - Data Set Description*. Department of Electrical Engineering and Computer Science, The University of Michigan, 2006. Available from: http://www.eecs.umich.edu/db/mbench/description.html#dataSet.

[25] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. *The Michigan Benchmark - Generation of the String Attributes and Elements*. Department of Electrical Engineering and Computer Science, The University of Michigan, 2006. Available from: http://www.eecs.umich.edu/db/mbench/description.html#3.3.

[26] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. *The Michigan Benchmark - Instructions to build the data generator.*. Department of Electrical Engineering and Computer Science, The University of Michigan, 2006. Available from: http://www.eecs.umich.edu/db/mbench/cgi/downloads/installWin.html.

[27] *ToXgene - the ToX XML Data Generator*. University of Toronto, 2005. Available from: http://www.cs.toronto.edu/tox/toxgene/.

[28] Weisstein, Eric W. *Zipf's Law*. MathWorld – A Wolfram Web Resource. Available from: http://mathworld.wolfram.com/ZipfsLaw.html

[29] *Toronto XML Server*. University of Toronto, 2002. Available from: http://www.cs.toronto.edu/tox/.

[30] D. Barbosa, A. Mendelzon, J. Keenleyside, K. Lyons. *ToXgene: An extensible template-based data generator for XML*. In Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002). Madison, Wisconsin - June 6-7, 2002. Available from: http://www.cs.toronto.edu/tox/toxgene/docs/ToXgene.pdf

[31] A. Aboulnaga, J. F. Naughton, C. Zhang: *Generating Synthetic Complex-Structured XML Data*. In WebDB'01: Proc. of the 4th Int. Workshop on the Web and Databases, pages 79-84, Washington, DC, USA, 2001.

[32] *<oXygen/> xml editor*. SyncRO Soft SRL, 2013. Available from: http://www.oxygenxml.com/

[33] *<oXygen/> XML Feature Matrix*. SyncRO Soft SRL, 2013. Available from: http://www.oxygenxml.com/xml_editor/feature_matrix.html

[34] *Liquid Xml Studio*. Liquid Technologies, 2013. Available from: http://www.liquid-technologies.com/xml-studio.aspx

[35] *Liquid XML Studio Edition Comparison*. Liquid Technologies, 2013. Available from: http://www.liquid-technologies.com/xml-studio.aspx?view=editions

[36] *Altova XMLSpy XML Editor*. Altova, 2013. Available from: http://www.altova.com/xmlspy.html

[37] *Altova XMLSpy Edition Comparison*. Altova, 2013. Available from: http://www.altova.com/xmlspy/editions/

[38] *Eclipse Home Page*. The Eclipse Foundation, 2013. Available from: http://www.eclipse.org/

[39] *Microsoft Visual Studio*. Microsoft, 2013. Available from: http://www.microsoft.com/visualstudio

[40] *Riak KV*. Basho Technologies, 2015. Available from: http://basho.com/products/riak-kv/

[41] *Redis*. Redis.io, 2015. Available from: http://redis.io

[42] *MemcacheDB*. MemcacheDB.org, 2015. Available from: http://memcachedb.org

[43] *Hamster DB*. Christoph Rupp, 2015. Available from: http://hamsterdb.com

[44] *Project Voldemort*. Project Voldermort - A distributed database, 2015. Available from: http://www.project-voldemort.com/voldemort/

[45] *MongoDB*. Mongo DB, Inc., 2015. Available from: https://www.mongodb.org

[46] *Apache CouchDB*. The Apache Software Foundation, 2015. Available from: http://couchdb.apache.org

[47] *RavenDB*. Hibernate Rhinos, 2015. Available from: http://ravendb.net

[48] *OrientDB*. Orient Technologies LTD, 2015. Available from: http://orientdb.com/orientdb/

[49] *Google Cloud Bigtable*. Google, 2015. Available from: https://cloud.google.com/bigtable/

[50] *Apache HBase*. The Apache Software Foundation, 2015. Available from: http://hbase.apache.org

[51] *Apache Cassandra*. The Apache Software Foundation, 2015. Available from: http://cassandra.apache.org

[52] *Hypertable*. Hypertable Inc., 2015. Available from: http://hypertable.org

[53] *Amazon SimpleDB*. Amazon Web Services, Inc., 2015. Available from: http://aws.amazon.com/simpledb/

[54] *Neo4j*. Neo Technology, Inc., 2015. Available from: http://neo4j.com

[55] *infiniteGraph*. Objectivity Inc., 2015. Available from: http://www.objectivity.com/products/infinitegraph/

[56] *FlockDB*. Github, Inc., 2015. Available from: https://github.com/twitter/flockdb

[57] Rabl, Tilmann and Jacobsen, Hans-Arno: *Big Data Generation*. Specifying Big Data Benchmarks. WBDB 2012, LNCS 8163. Springer Berlin Heidelberg, pp. 20-27, 2014. Available from: http://dx.doi.org/10.1007/978-3-642-53974-9_3

[58] Ming, Zijian and Luo, Chunjie and Gao, Wanling and Han, Rui and Yang, Qiang and Wang, Lei and Zhan, Jianfeng: *BDGS: A Scalable Big Data Generator Suite in Big Data Benchmarking.* Advancing Big Data Benchmarks. WDBD 2013, LNCS 8585. Springer Internation Publishing Switzerland, pp. 138-154, 2014. DOI: 10.1007/978-3-319-10596-3 11 Available from: `http://dx.doi.org/10.1007/978-3-319-10596-3_11`

[59] *DataGenerator*, version 2.1. 2015. Available from: `http://finraos.github.io/DataGenerator/`

[60] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, H.A. Jacobsen: *BigBench: Towards an industry standard benchmark for big data analytics.* In: Proceedings of the ACM SIGMOD Conference (2013). Available from: `http://dl.acm.org/citation.cfm?id=2463712`

[61] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, WanlingGao, Zhen Jia, Yingjie Shi, Shujie Zhang, Cheng Zhen, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu: *BigDataBench: a Big Data Benchmark Suite from Internet Services.* The 20th IEEE International Symposium On High Performance Computer Architecture (HPCA-2014), February 15-19, 2014, Orlando, Florida, USA. Available from: `http://prof.ict.ac.cn/BigDataBench/wp-content/uploads/2013/10/Wang_BigDataBench.pdf`

[62] V. Omanashvili: *JSON GENERATOR*, 2015. Available from: `http://www.json-generator.com/`

[63] B. Keen: *generatedata.com*, version 3.1.4, 2015. Available from: `http://www.generatedata.com/`

[64] *Mockaroo.* Mockaroo, LLC. 2015. Available from: `https://www.mockaroo.com/`

[65] *MongoDB-Datasets.* Github, 2015. Available from: `https://github.com/mongodb-js/datasets`

[66] *Node.js.* Node.js Foundation, 2015. Available from: `https://nodejs.org/`

[67] *V8 JavaScript Engine.* Google, 2015. Available from: `https://code.google.com/p/v8/`

[68] *Chance.* Victor Quinn, 2015. Available from: `http://chancejs.com/`

[69] *Faker.js.* Matthew Bergman & Marak Squires, Github, 2015. Available from: `https://github.com/FotoVerite/Faker.js`

[70] *Underscore.* 2015. Available from: `http://underscorejs.org/`

[71] *Database References.* MongoDB, Inc., 2015. Availabe from: `http://docs.mongodb.org/manual/reference/database-references/`

[72] *The Four Vś of Big Data.* IBM Big Data & Analytics Hub, IBM, 2015. Available from: `http://www.ibmbigdatahub.com/infographic/four-vs-big-data`

[73] *ECMAScript 2015 Language Specification*. Ecma International, Geneva, 2015. Available from: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[74] *BSON*. Specification Version 1.0, 2015. Available from: http://bsonspec.org/spec.html

[75] *Aggregation Pipeline in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/aggregation-pipeline/

[76] *Map-Reduce in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/map-reduce/

[77] *Single Purpose Aggregation Operations*. Mongo DB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/single-purpose-aggregation/

[78] *Multikey Indexes in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/index-multikey/

[79] *Geospatial Indexes in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/applications/geospatial-indexes/

[80] *Text Indexes in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/index-text/

[81] *Hashed Indexes in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/index-hashed/

[82] *Replication in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/replication-introduction/

[83] *Sharding in MongoDB*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/sharding-introduction/

[84] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. W3C Proposed Recommendation 30 April 2015. World Wide Web Consortium, 2015. Available from: http://www.w3.org/TR/scxml/

[85] *mongodb-schema*. Github, 2015. Available from: https://github.com/mongodb-js/mongodb-schema

[86] *Production Cluster Architecture*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/sharded-cluster-architectures-production/

[87] *.NET Framework 4.5*. Microsoft Developer Network. Available from: http://msdn.microsoft.com/en-us/library/vstudio/w0x726c2.aspx

[88] *MPI.NET: High-Performance C# Library for Message Passing*. The Trustees of Indiana University, 2015. Available from: http://www.osl.iu.edu/research/mpi.net/

[89] *Cross-platform framework for interprocess communication*. Ondrej Uzovic, 2015. Available from: http://www.eneter.net/index.htm

[90] *Robomongo: Shell-centric cross-platform MongoDB management tool*. Paralect, 2015. Available from: http://robomongo.org/

[91] *Write Concern*. MongoDB, Inc., 2015. Available from: http://docs.mongodb.org/manual/core/write-concern/

List of Tables

List of Figures

List of Listings

Appendices

A. DVD Contents

This thesis contains an attached DVD with binaries of *JsonGenerator*, source code of *JsonGenerator*, documentation generated from the source code and an electronic version of this document. We also included the configuration files and other scripts we used in experiments from Chapter 7.

The disc contains the following directories:

- `SourceCode` – contains the complete source code in the form of solution in Visual Studio 2013
- `Documentation` – contains documentation generated from comments in the source code
- `Binaries` – contains the executable binaries of *JsonGenerator* with all required dependencies
- `Test Data` – contains configuration files used in our experiments and also various helper scripts
- `Thesis` – contains this text in PDF format

B. XMark Data Generator DTD

```
<!ELEMENT site          (regions, categories, catgraph, people, open_auctions,
  closed_auctions)>
<!ELEMENT categories    (category+)>
<!ELEMENT category      (name, description)>
<!ATTLIST category      id ID #REQUIRED>
<!ELEMENT name          (#PCDATA)>
<!ELEMENT description   (text | parlist)>
<!ELEMENT text          (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold      (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword    (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph      (#PCDATA | bold | keyword | emph)*>
<!ELEMENT parlist    (listitem)*>
<!ELEMENT listitem      (text | parlist)*>

<!ELEMENT catgraph      (edge*)>
<!ELEMENT edge          EMPTY>
<!ATTLIST edge          from IDREF #REQUIRED to IDREF #REQUIRED>

<!ELEMENT regions       (africa, asia, australia, europe, namerica, samerica)>
<!ELEMENT africa        (item*)>
<!ELEMENT asia          (item*)>
<!ELEMENT australia     (item*)>
<!ELEMENT namerica      (item*)>
<!ELEMENT samerica      (item*)>
<!ELEMENT europe        (item*)>
<!ELEMENT item          (location, quantity, name, payment, description, shipping,
  incategory+, mailbox)>
<!ATTLIST item          id ID #REQUIRED
                        featured CDATA #IMPLIED>
<!ELEMENT location      (#PCDATA)>
<!ELEMENT quantity      (#PCDATA)>
<!ELEMENT payment       (#PCDATA)>
<!ELEMENT shipping      (#PCDATA)>
<!ELEMENT reserve       (#PCDATA)>
<!ELEMENT incategory    EMPTY>
<!ATTLIST incategory    category IDREF #REQUIRED>
<!ELEMENT mailbox       (mail*)>
<!ELEMENT mail          (from, to, date, text)>
<!ELEMENT from          (#PCDATA)>
<!ELEMENT to            (#PCDATA)>
<!ELEMENT date          (#PCDATA)>
<!ELEMENT itemref       EMPTY>
<!ATTLIST itemref       item IDREF #REQUIRED>
<!ELEMENT personref     EMPTY>
<!ATTLIST personref     person IDREF #REQUIRED>

<!ELEMENT people        (person*)>
<!ELEMENT person        (name, emailaddress, phone?, address?, homepage?,
  creditcard?, profile?, watches?)>
<!ATTLIST person        id ID #REQUIRED>
<!ELEMENT emailaddress  (#PCDATA)>
<!ELEMENT phone         (#PCDATA)>
<!ELEMENT address       (street, city, country, province?, zipcode)>
<!ELEMENT street        (#PCDATA)>
<!ELEMENT city          (#PCDATA)>
```

```
<!ELEMENT province        (#PCDATA)>
<!ELEMENT zipcode         (#PCDATA)>
<!ELEMENT country         (#PCDATA)>
<!ELEMENT homepage        (#PCDATA)>
<!ELEMENT creditcard      (#PCDATA)>
<!ELEMENT profile         (interest*, education?, gender?, business, age?)>
<!ATTLIST profile         income CDATA #IMPLIED>
<!ELEMENT interest        EMPTY>
<!ATTLIST interest        category IDREF #REQUIRED>
<!ELEMENT education       (#PCDATA)>
<!ELEMENT income          (#PCDATA)>
<!ELEMENT gender          (#PCDATA)>
<!ELEMENT business        (#PCDATA)>
<!ELEMENT age             (#PCDATA)>
<!ELEMENT watches         (watch*)>
<!ELEMENT watch           EMPTY>
<!ATTLIST watch           open_auction IDREF #REQUIRED>

<!ELEMENT open_auctions   (open_auction*)>
<!ELEMENT open_auction    (initial, reserve?, bidder*, current, privacy?,
  itemref, seller, annotation, quantity, type, interval)>
<!ATTLIST open_auction    id ID #REQUIRED>
<!ELEMENT privacy         (#PCDATA)>
<!ELEMENT initial         (#PCDATA)>
<!ELEMENT bidder          (date, time, personref, increase)>
<!ELEMENT seller          EMPTY>
<!ATTLIST seller          person IDREF #REQUIRED>
<!ELEMENT current         (#PCDATA)>
<!ELEMENT increase        (#PCDATA)>
<!ELEMENT type            (#PCDATA)>
<!ELEMENT interval        (start, end)>
<!ELEMENT start           (#PCDATA)>
<!ELEMENT end             (#PCDATA)>
<!ELEMENT time            (#PCDATA)>
<!ELEMENT status          (#PCDATA)>
<!ELEMENT amount          (#PCDATA)>

<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref, price, date, quantity,
  type, annotation?)>
<!ELEMENT buyer           EMPTY>
<!ATTLIST buyer           person IDREF #REQUIRED>
<!ELEMENT price           (#PCDATA)>
<!ELEMENT annotation      (author, description?, happiness)>

<!ELEMENT author          EMPTY>
<!ATTLIST author          person IDREF #REQUIRED>
<!ELEMENT happiness       (#PCDATA)>
```

Listing 52: XMark Data Generator DTD