Deliverable 2.3.2
Second Maintenance and Update Report of the HOBBIT Platform

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Due Date of Deliverable</td>
<td>Month 36, 30/11/2018</td>
</tr>
<tr>
<td>Actual Submission Date</td>
<td>Month 36, 30/11/2018</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP2 - Benchmarking Platform</td>
</tr>
<tr>
<td>Task</td>
<td>T2.3</td>
</tr>
<tr>
<td>Type</td>
<td>Report</td>
</tr>
<tr>
<td>Approval Status</td>
<td>Final</td>
</tr>
<tr>
<td>Version</td>
<td>0.1</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>20</td>
</tr>
</tbody>
</table>

Abstract: This deliverable presents the final maintenance and update report of the HOBBIT platform.

The information in this document reflects only the author’s views and the European Commission is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/her sole risk and liability.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 688227.
History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2018-10-08</td>
<td>First draft created</td>
<td>Michael Röder</td>
</tr>
<tr>
<td>0.2</td>
<td>2018-10-10</td>
<td>Updated TWIG description</td>
<td>René Speck</td>
</tr>
<tr>
<td>0.3</td>
<td>2018-11-06</td>
<td>Reviewed version created</td>
<td>Gayane Sedrakyan</td>
</tr>
<tr>
<td>0.9</td>
<td>2018-11-23</td>
<td>Beta version created</td>
<td>Michael Röder</td>
</tr>
<tr>
<td>1.0</td>
<td>2018-11-27</td>
<td>Final version created</td>
<td>Michael Röder</td>
</tr>
</tbody>
</table>

Author List

<table>
<thead>
<tr>
<th>Organization</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfAI</td>
<td>Michael Röder</td>
<td><a href="mailto:roeder@informatik.uni-leipzig.de">roeder@informatik.uni-leipzig.de</a></td>
</tr>
<tr>
<td>AGT</td>
<td>Pavel Smirnov</td>
<td><a href="mailto:psmirnov@agtinternational.com">psmirnov@agtinternational.com</a></td>
</tr>
<tr>
<td>imec</td>
<td>Ruben Taelman</td>
<td><a href="mailto:ruben.taelman@ugent.be">ruben.taelman@ugent.be</a></td>
</tr>
<tr>
<td>imec</td>
<td>Gayane Sedrakyan</td>
<td><a href="mailto:Gayane.Sedrakyan@ugent.be">Gayane.Sedrakyan@ugent.be</a></td>
</tr>
<tr>
<td>InfAI</td>
<td>Axel-Cyrille Ngonga Ngomo</td>
<td><a href="mailto:ngonga@informatik.uni-leipzig.de">ngonga@informatik.uni-leipzig.de</a></td>
</tr>
<tr>
<td>InfAI</td>
<td>Rene Speck</td>
<td><a href="mailto:speck@informatik.uni-leipzig.de">speck@informatik.uni-leipzig.de</a></td>
</tr>
</tbody>
</table>
Executive Summary

This document is the final maintenance and update report of the HOBBIT platform. It describes the monitoring that is used for the online instance of the platform, the updates created for the platform source code and the usage of the platform’s online instance during the first months. After that, the mimicking algorithms that have been implemented for supporting the development of the HOBBIT benchmarks are described.
Contents

1 Introduction 4

2 Monitoring 4

3 Updates 4
   3.1 Benchmark report 5
   3.2 AWS-extended Platform 5
   3.3 AWS Controller 7

4 Platform Usage 7

5 Mimicking Algorithms 7
   5.1 PoDiGG 8
      5.1.1 Stops Generator 8
      5.1.2 Edges Generator 9
      5.1.3 Routes Generator 9
      5.1.4 Trips Generator 9
   5.2 TWIG 10
      5.2.1 Crawling Twitter 10
      5.2.2 Model Analysis 11
         5.2.2.1 Tweets per User 11
         5.2.2.2 Tweets per Daytime 12
         5.2.2.3 Predecessors and Successors of Words 12
      5.2.3 Generate Synthetic Tweets 12
   5.3 IT Monitoring data 13
   5.4 Printing Machines 15
   5.5 Weidmüller molding machines 18
   5.6 Floating car data 19

References 19
1 Introduction

This document is the final maintenance and update report of the Hobbit platform. The platform serves as a framework for benchmarking Big Linked Data systems. Both benchmarks focusing on the evaluation of the quality of a system using single consecutive requests as well as benchmarks aiming at the efficiency (e.g., by generating a lot of parallel requests leading to a high work load) can be run on the platform. Especially for the latter case, the platform supports the handling of Big Linked Data to make sure that even for scalable systems a maximum load can be generated.

The Hobbit platform included in the Hobbit project aims at two goals. Firstly, we offer an open source evaluation platform that can be downloaded and executed locally. For this open source project, several updates have been developed since the release of the platform. These updates are described in Section 3.

Secondly, we offer an online instance of the platform for a) running public challenges and b) making sure that people without the required infrastructure are able to run the benchmarks they are interested in. For the online instance and the services that support the Hobbit project, we introduced a monitoring solution that is described in Section 2. Section 4 lists the usage of the online instance during the first months.

Finally, Section 5 describes the mimicking algorithms that have been developed throughout the Hobbit project to create the data needed for the benchmarks implemented in the work packages 3 - 6. Throughout the document, the term "system" refers to the system that is benchmarked and an "experiment" is a single execution of a benchmark to evaluate a single system.

2 Monitoring

The monitoring of the provided services as well as the online instance of the Hobbit platform are crucial to a) offer those in a reliable way and b) investigate problems as soon as they occur. Therefore, we introduced a monitoring solution during the first half of the project. Since no major changes were necessary its description can be found in Deliverable D2.3.1.

3 Updates

The Hobbit platform has been developed as open source project that is hosted together with other projects of the Hobbit project on Github.1 Table 1 lists the open source projects that are maintained by T2.3.

Maintaining the open source projects includes a) updating and extending the tutorials and explanations in the projects documentation page,2 b) answering technical questions of interested 3rd party representatives as well as c) working on issues that are raised by users or project members. For the latter, the issue system of Github is used. A newly added issue is marked with one or several categories and a priority level. Based on their priority level, the issues are processed and closed after the problem has been solved. Until now, 271 issues have been reported and 178 of them have been closed. From the remaining 92 issues, 29 are marked as bugs while the remaining issues are either questions raised

1https://github.com/hobbit-project
2https://hobbit-project.github.io
by users or documenting enhancements which could not be implemented during the project because of limited resources.

During the runtime of T2.3 several enhancements have been added to the platform. Some of them are described in the following.

### 3.1 Benchmark report

A report page has been added to the graphical user interface. This page gives a different way to access the benchmark results compared to other views implemented before. The user can choose a benchmark from the list. After that, the information about the last successful benchmark runs are loaded. Additionally, the page offers the visualization of experiment results in combination with benchmark parameter values in scatter plots. Figure 1 shows an example of such a plot for the Odin benchmark version 2.

### 3.2 AWS-extended Platform

A cloud extended version of the HOBBIT platform was developed to conduct experiments using dynamically configurable AWS-based infrastructure. The goal of the extension is to add the following capabilities to the platform:

1. Ability to perform performance and scalability benchmarks using different infrastructure setups (i.e., switching between different types and amounts of EC2 machines for the benchmark and system containers).

2. Ability to reduce costs of owning a benchmarking infrastructure - the HOBBIT platform is intended to be deployed on a low performance machine (e.g. laptop or minimal EC2 instance), but

---

3The AWS-extended version of the platform can be found at [https://github.com/hobbit-project/platform/tree/cloud](https://github.com/hobbit-project/platform/tree/cloud)
Figure 1: Example of a scatter plot on the benchmark report page.
still be able to use the unlimited computational power of cloud resources. The "Pay per use" costs model applied for cloud-based resources might be more preferable for certain benchmarking purposes.

Technically the AWS-extension to the platform is based on a standalone component named AWS Controller and includes cloud-extended subcomponents (such as DockerClient, ContainerManager, QueueManager, Swarm Cluster Manager, etc.). The extended platform controller requires the standard number of services (Docker, RabbitMQ, Redis, Virtuoso, GUI) to be running, but the swarm cluster for containers is dynamically deployed in the AWS cloud depending on a current demand (i.e., the experiments submitted into the queue). The configuration of desired EC2 instances might be specified per experiment in JSON format or a default cluster configuration is used. Experiments in the queue are grouped by cluster their configurations, so the time gaps required to reconfigure the cluster are minimized. The overall cluster deployment in the AWS cloud takes 10-12 minutes while the cluster reconfiguration between two experiments takes around 3 minutes (if it is necessary). After the experiment queue is empty the cluster can be switched off after a maximum waiting time to reduce costs.

3.3 AWS Controller

Amazon Web Services Controller is a software library, which provides an ability to deploy and manage computational resources in the AWS cloud. Firstly, the controller includes clients to a number of Amazon services (Cloud Formation, S3, EC2, Autoscaling Groups, etc.) integrated into a unified interface used for all types of HOBBIT-related tasks. Secondly, the controller provides a cluster manager interface for designing use-case specific implementations. For example the Swarm Cluster Manager and Neptune Cluster Manager have been designed for the Extended Platform Controller and Neptune System Adapter respectively. Thirdly, the controller includes the necessary SSH-client tools to establish hierarchical SSH-tunnels to access remotely deployed clusters and services not exposing them to the public internet (the port forwarding principle is used).

Being implemented as a standalone library the AWS Controller is a good basis for new generations of cloud-oriented HOBBIT-compatible system adapters, which would open opportunities to benchmark cloud-based services with a pay per use model.5

4 Platform Usage

The online instance of the HOBBIT platform has 250 user accounts. 337 projects have been created in the platform’s GitLab. Therefore, we can assume that more than 300 components of benchmarks or systems have been uploaded. More than 12800 experiments have been carried out on the platform. Figure 2 shows the growth of the number of experiments over time.

5 Mimicking Algorithms

This section briefly describes the latest status of the mimicking algorithms that have been implemented to be used by the different benchmarks created in the HOBBIT project.

---

4 AWS Neptune System Adapter https://github.com/hobbit-project/neptune-system-adapter
5 AWS Marketplace https://aws.amazon.com/marketplace
5.1 PoDiGG

PoDiGG is a public transport RDF dataset generator that can be used to mimic public transport data based on a given population distribution.\(^6\) It is based on the assumption that areas with a more dense population are expected to have more nearby and more frequent access to public transport. Based on the population distribution, PoDiGG generates a transit network and its scheduling in the following four steps.

1. Generate the positions of stops
2. Connects stops with edges
3. Generate routes in the generated network
4. Schedule timely trips over routes

Finally, all this data is serialized to RDF using the Linked Connections\(^7\) and GTFS\(^8\) ontologies. These four steps will be explained in more detail hereafter.

5.1.1 Stops Generator

The goal of the first step is to generate a realistic placement of stops in a two-dimensional area subdivided in discrete cells of equal size. For a preconfigured number of stops, we iteratively tag random cells as stops, where each stop is given a size equal to the predefined population value. This random selection is based on a Zipf distribution where cells with a higher population value have a

---

\(^6\)A more detailed description can be found in [3, 2].

\(^7\)http://semweb.mmlab.be/ns/linkedconnections

\(^8\)http://vocab.gtfs.org/terms
higher chance to be tagged as a stop than cells with a low population value. This distribution can be scaled to select minimum and maximum population values at which to tag stops.

5.1.2 Edges Generator

After stops have been generated, edges are created in order to form paths between stops. This generator will always create one connected transit network graph, where all stops are reachable from all other stops by any given path consisting of edges. In order to do this, the generator consists of two clustering phases and one so-called “loose stops”-phase.

The first agglomerative hierarchical clustering phase first considers all stops to be part of a different cluster. Each cluster always has a center point representing the average location of all stops in that cluster. This phase loops until all clusters have an inter-cluster distance larger than a pre-configured value. These distances are always calculated using Euclidian distances of the cells in their two-dimensional area. After this step, one or more clusters will exist, where each stop will be connected with zero or more other stops.

After this first step, we have clusters of stops that lie close to each other. In the next step, we consider border stops as the only stops in a cluster that can be used to transfer to (border) stops in other clusters. This step ensures that all clusters eventually form one connected network.

The final step aims to resolve the problem of loose stops, which means that the two previous steps result in a significant amount of stops that are connected with only one other stop, i.e., they have a degree of one, which does not occur frequently in reality. This step aims to resolve this problem by detecting these loose stops, searching for other stops in the opposite direction of the single present edge, and adding an edge for the first stop that can be found in a given area.

5.1.3 Routes Generator

Once we have a network of stops connected by edges, we can start the placement of routes over this network. In order to simulate long and short distance routes, this generation phase is divided into two steps: first we create long routes connecting large stops, and after that, we create smaller routes connecting smaller stops.

For the first step, we create a list of the largest stops, where the size of this list is configurable. For each of these largest stops, the shortest path in the network to all other largest stops is calculated using the A* search algorithm and instantiated as a route. All sufficiently large stops on that route are also included as stops for that route. Next, smaller routes are generated by iteratively selecting two random large stations, connecting them through a heuristically determined shortest path, and including all passed stops in the route. This is done until a preconfigured number of routes is generated.

5.1.4 Trips Generator

Finally, we generate trips based on the created routes. This is done by continuously picking a random route, with a larger chance on longer routes, and instantiating that route in time as a trip. This is done until a pre-configured number of trips has been created. The instantiation of a trip is done by choosing a random starting time of the trip, based on a preconfigured time distribution. The default time distribution is based on the logs of the route planning api (iRail\(^{9}\)) in Belgium[1]. The stop times

\(^{9}\)https://irail.be/
are calculated for each stop in the trip by estimating the time it takes for a train to go from one stop to the next, including preconfigured speedup times, maximum vehicle speeds and required waiting times at stops.

5.2 TWIG

During the last decade, Twitter has become one of the most important microblogging services for online news and social networking on the Web with around 310 million monthly active users in March 2016. The increasing popularity of Twitter as a data source for a multitude of applications ranging from entity extraction to sentiment analysis around products makes it an important dataset for benchmarking. For this reason, a large number of reference datasets based on Twitter (e.g., the Twitter7 dataset with 476 million tweets from 20 million users covering a six-month period from June 1 2009 to December 31 2009) was created. However, a request from Twitter, made the Twitter7 dataset and similar datasets no longer available for public use. Within Hobbit, we circumvent the problem of generating Twitter-like data by providing TWIG, the Twitter Benchmark Generator.

TWIG is a collection of algorithms to 1) crawl Twitter that includes unique IDs for users, user tweet times and the tweets themselves as well as transform the crawled data anonymized into RDF based on the TWIG ontology, 2) analyze the distributions of the data and 3) mimic a Twitter network by generating new tweets based on the analyzed distributions.

With TWIG, we provide synthetic data that is very similar to real Twitter data and can be used to benchmark storage systems w.r.t. their performance when faced with Twitter streams. Using TWIG however has the main advantage of leading to highly controllable, scalable and open data that also means clear and comparable benchmarking.

Our collection of the implemented algorithms is open source and can be found together with the documentation at https://github.com/dice-group/TWIG.

5.2.1 Crawling Twitter

We crawl Twitter and store the crawled data that includes unique IDs for users, user tweet times and the tweets themselves as well as transform the crawled data anonymized into an RDF serialization. Although in the Hobbit project we are using only data that has already been anonymized. The TWIG parsing algorithm includes an anonymization step enabling its usage for other datasets. This anonymization step adds a random 32 Byte long value to each user name before replacing the whole string with its MD5-Hash value.

Let \( n \) be the anonymized user name, TWIG will represent the user as RDF resource (Listing 1).

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix twig: <http://aksw.org/twig#> .
twig:n a owl:NamedIndividual, twig:OnlineTwitterAccount ;
twig:sends twig:n_t_s .
```

Listing 1: Stored user data.

Each user has a list of Tweets sent by this user. A tweet sent by user \( n \) is represented by \( n\_t\_s \) where \( t \) is the tweets timestamp in the format \( yyyy-mm-ddThh:mm:ss \) and \( s \) is a random seed for the pseudorandom number generator (PRNG) provide to TWIG through its arguments. Let \( m \) be the content of the message, TWIG will represent a tweet as well as RDF resource (Listing 2).
Listing 2: Stored tweet data.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix twig: <http://aksr.org/twig#> .
twig:n_t_s a owl:NamedIndividual, twig:Tweet ;
twig:tweetContent "m" ;
twig:tweetTime "t" xsd:dateTime .
```

Figure 3: Excerpt of the TWIG ontology.

Every matching of the regular expression \([a-zA-Z0-9_]{1,15}\) in m is the reference to another user. For every reference to another user \(n^\prime\), TWIG will add the triple in Listing 3 to the tweet \(n_t_s\).

```
twig:n_t_s twig:mentions twig:n'.
```

Listing 3: User mention data.

The result of the parsing process is a single RDF model containing all users and their tweets.

5.2.2 Model Analysis

TWIG offers the analysis of an RDF twitter model as it has been generated in the step before, regarding the following three different distributions.

5.2.2.1 Tweets per User

For this analysis, TWIG iterates over all users in the model, counts the number of tweets that have been sent by the single users and records the difference between the first and the last day the user has sent at least one message. Let \(n\) be a user, \(t \in \mathbb{N}\) a number of days used for normalization and \(t'\) the number of days a user has sent messages, the normalized number of tweets is calculated as

\[
n_t = \frac{n}{t} \times t' \tag{1}
\]

From the normalized number of tweets, a frequency distribution \(H_{T,t}: \mathbb{N}^+ \rightarrow \mathbb{N}\) is determined, where \(H_{T,t}(j) = k\) expresses that \(k\) users have send \(j\) tweets in \(t\) days. Based on \(H_{T,t}\), a discrete prob-
ability distribution $P_{T,t} : \mathbb{N}^+ \to [0,1]$ can be determined. For every element $j \in \mathbb{N}^+$, the distribution gives the probability of a user sending $j$ tweets during $t$ days.

### 5.2.2.2 Tweets per Daytime

TWIG determines the frequency distribution $H_D : \Gamma \to \mathbb{N}$ where $\Gamma = \{00 : 01, 00 : 02, \ldots, 23 : 59\}$ is the set of minutes on a single day. Based on the frequencies gathered from the data the probability distribution $P_D : \Gamma \to [0,1]$ is generated. For every minute of a day it offers a probability that a given tweet is send during this minute.

### 5.2.2.3 Predecessors and Successors of Words

To create synthetic Tweets, TWIG makes use of a first order Markov Chain which describes the probability for a word being followed by another word. To create the underlying transition probability matrix of this chain, TWIG analyses the stored Tweet messages to create a frequency distribution over all words in all Tweets.

### 5.2.3 Generate Synthetic Tweets

TWIG is using the three distribution functions to create synthetic tweets. TWIG expects several parameter for the scalability of the data. Parameter $n$ for the number of users, $d$ defines the start date, $t$ the duration of the mimicking time in days, $s$ the pseudo random number generator seed that is needed for variations of the mimicking approach. Thus, TWIG is a deterministic algorithm, it creates with the same parameter and data always the same synthetic Tweets by the same users.

TWIG creates $n$ random users with IDs. With the help of the Tweets per User distribution, TWIG creates the number of Tweets each user sends per day. For each Tweet, TWIG creates the time when the Tweet is send with the help of the Tweets per Daytime distribution. At least, TWIG creates the Tweet itself with the help of the Predecessors and Successors of Words.

The generate synthetic Tweets are stored in a RDF model (Listing 4).

```sparql
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix twig: <http://aksw.org/twig#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

twig:65abfcd2e9cbbb788fa0089450d2939
a owl:NamedIndividual , twig:OnlineTwitterAccount ;
twig:sendsto:twig:65abfcd2e9cbbb788fa0089450d2939_2009-10-01T00:28:23_1 ,
twig:65abfcd2e9cbbb788fa0089450d2939_2009-10-01T00:28:23_1
a owl:NamedIndividual , twig:Tweet ;
twig:tweetContent "I'm in pain" ;
twig:tweetTime "2009-10-01T00:28:23"^^xsd:dateTime .

twig:65abfcd2e9cbbb788fa0089450d2939_2009-10-01T00:02:12_1
a owl:NamedIndividual , twig:Tweet ;
twig:mentions twig:2ad4278a2003050bed:330b4f22fd25 ;
twig:tweetContent "I'm writing my own Twitter client @2ad4278a2003050bed:330b4f22fd25";
twig:tweetTime "2009-10-01T00:02:12"^^xsd:dateTime .

Listing 4: Generated synthetic example data.
```

Listing 5 shows the command to start the model analysis and to store the models. Thereby is <command> a placeholder and should be substituted with MessageCounterHandler to analyze the Tweets per User, with TimeCounterHandler to analyze the Tweets per Daytime or with
WordMatrixHandler for the Predecessors and Successors of Words model. The parameter -in points to the input data generated in the first step (Section 5.2.1) and the parameter -out points to a folder where the model will be stored.

```
java -jar twig.jar <command> --out=ana --in=data
```

Listing 5: Model analysis command of TWIG.

Listing 6 shows an example execution command to generate synthetic Tweets. Automaton is thereby the mimicking command for TWIG, the next three parameters are the created models: frequency distribution over all words, Tweets per user and Tweets per daytime. In this example execution, TWIG generates synthetic Tweets for 10 users within 2 days starting at 2009-09-29 with 1 as PRNG seed and the output folder out where the generated synthetic Tweets will be stored.

```
j ava twig.jar Automaton ana/word.obj ana/message.obj ana/time.obj 10 2 2009-09-29 1 out
```

Listing 6: Example execution to generate synthetic tweets with TWIG.

### 5.3 IT Monitoring data

The USU Big-Data platform allows processing of large amounts of data for several different use-cases. Compute nodes (cf. Figure 4) process the data and storage nodes are used to persist the results. One of the challenges is to identify bottle necks within a set-up. Therefore a monitoring node constantly monitors the platform.

Measurement variables of storage nodes monitored are e.g. network traffic, disk space, and CPU utilization, having the read and write jobs coming from compute nodes as well as maintenance jobs performed by the storage nodes database Cassandra.\(^{10}\) The IT dataset represents technical performance measurements of a Cassandra database cluster as it can be found in the USU Big-data platform. The features that are monitored and recorded are listed in Table 2.

The general process of mimicking data used for the IT-data is shown in Figure 5. The dependencies and patterns within the captured data are analysed and the analysis results are validated by experts. From these findings models for stochastic and deterministic simulation of the data is generated.

In our model, we assume that all measured variables (cf. Table 2) are exclusively determined by

\(^{10}\)http://cassandra.apache.org/
### Measured variable Description

<table>
<thead>
<tr>
<th>Measured variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Traffic</td>
<td>Amount of data moving across storage nodes (Cassandra) and compute nodes.</td>
</tr>
<tr>
<td>SSTable Count</td>
<td>Number of immutable sorted strings tables (SSTables) on disk.</td>
</tr>
<tr>
<td>Bytes Compacted</td>
<td>Size of compacted SSTables.</td>
</tr>
<tr>
<td>Job Monitor</td>
<td>Number of successfully finished and failed jobs.</td>
</tr>
<tr>
<td>CPU Utilization</td>
<td>CPU load of a Cassandra storage node.</td>
</tr>
</tbody>
</table>

Table 2: The features that are measured for a single Cassandra storage node.

Figure 5: General Process for Determining the Mimicking Algorithm.

The jobs running on a single Cassandra storage node. The jobs in our description are strongly simplified i.e. they are solely determined by the lifetime $\tau$ and the job type. All parameter values, defining the job type, the job lifetime as well as the job submission times, were chosen such that the simulation results and the real time courses coincide qualitatively.

In our description we distinguish between long-lasting and brief jobs. The lifetime $\tau$ of a job is randomly chosen from normal distributions with mean $\mu = 84000s$ and standard deviation $\sigma = 6000s$ and mean $\mu = 4500s$ and standard deviation $\sigma = 2000s$, respectively. Jobs can be divided into three groups in relation to their submission times: The first group of jobs, a mixture of brief and long lasting jobs, is placed in the job queue every day at 10:00pm. Jobs of the second group, exclusively brief jobs, are randomly placed in the queue over the whole day. Brief jobs for SSTable compaction, defining the third group, are exceptional in that they are only submitted every week on Tuesday at 12:00am. To prevent that all jobs of a group start simultaneously, a randomly chosen time offset is added to the job submitting times. The offsets are uniformly distributed over intervals depending on the job group and the job type. For brief jobs in the first group, the interval $I_{1,b}$ is equal to $[0s, 6000s]$, for jobs in the second group, the interval $I_2$ is equal to $[0s, 10s]$, for long lasting jobs in the first group the interval $I_{1,l}$ is equal to $[0s, 4000s]$ and finally for compaction jobs, the interval $I_c$ is equal to $[0s, 20000s]$. 

- Analysis of dependencies and patterns
- Expert knowledge

Models for stochastic and deterministic simulations
Each running job increases the CPU utilization by $\Delta u_{CPU} = 1\% + \eta$, where $\eta$ is white Gaussian noise with standard deviation $\sigma = 0.2\%$. We expect that only short jobs increase the network traffic by $\Delta u_{net}$, where $\Delta u_{net}$ is a normal distributed random variable with mean $\mu = 10\text{Mbps}$ and standard deviation $\sigma = 2\text{Mbps}$. In order to generate the time course of the disk space, we assume that brief jobs write and delete on average $0.2\text{GB}$ on the hard disk. The deletion occurs at starting point, the writing occurs successively during runtime. The creation and deletion of SSTables is determined by a large amount of processes, which we don’t consider in our description. Instead we use a simplified model for SSTables creation and deletion in that the number of SSTables is increased at constant rate $\omega_{g,SST} = 7.5E - 3\text{s}^{-1}$ and is shrinked at constant rate $\omega_{g,SST} = 7.5E - 6\text{s}^{-1}$. Compaction jobs raise the shrinking rate by $1.5E - 7h/(sx\text{Bytes})xc$, where $c$ is the total compaction executed by running compaction jobs. In order to make the time courses look more realistic, we add weak Gaussian noise to the parameters describing the temporal evolution of the SS-tables count. The compaction per job is Gaussian distributed with mean $50\text{GBytes/h}$ and standard deviation $\sigma = 20\text{GBytes/h}$. Whether a job ends successfully or not is chosen at random with the probabilities $P(\text{success}) = 0.95$ and $P(\text{fail}) = 0.05$, respectively. In our simulation, we successively update the system time $t$ by the temporal resolution $\Delta t$ and determine the number of running jobs and their states after each time step. Because the measured variables are exclusively defined by the number of active jobs and their states, we are able to compute the temporal evolutions of the considered variables, see Figure 6.

5.4 Printing Machines

The dataset of the production industry domain represents events logged by production machines. The concrete machine used here is the printing machine. A printing machine processes different printing jobs. A printing job starts with the initial “Starting the print job” event and stops with the “Finished the print job” event. In between these start and finish events there are several other events. The sequence of events can include the events listed in Table 3 which are currently covered by the mimicking algorithms.

Some of the events might be correlated with other events. Others might be uncorrelated. A
Table 3: Event types of a printing machine.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Starting the print job</td>
</tr>
<tr>
<td>2</td>
<td>Finished the print job</td>
</tr>
<tr>
<td>3</td>
<td>Washing the ink rollers</td>
</tr>
<tr>
<td>4</td>
<td>Washing the impression cylinder</td>
</tr>
<tr>
<td>5</td>
<td>with washing ink fountain roller</td>
</tr>
<tr>
<td>6</td>
<td>with washing plates</td>
</tr>
<tr>
<td>7</td>
<td>Changing the plate</td>
</tr>
<tr>
<td>8</td>
<td>Washing the blanket</td>
</tr>
<tr>
<td>9</td>
<td>Sheet is wrongly aligned</td>
</tr>
<tr>
<td>10</td>
<td>Sheet is missing</td>
</tr>
<tr>
<td>11</td>
<td>Side guide warning</td>
</tr>
<tr>
<td>12</td>
<td>Sheet is early</td>
</tr>
<tr>
<td>13</td>
<td>Double sheet</td>
</tr>
<tr>
<td>14</td>
<td>Performance</td>
</tr>
<tr>
<td>15</td>
<td>Partially completed operation</td>
</tr>
<tr>
<td>16</td>
<td>Printing interval</td>
</tr>
<tr>
<td>17</td>
<td>Production good sheet</td>
</tr>
</tbody>
</table>

A printing job can be of different sizes. Some might run for some minutes, others for longer time. Therefore each job varies in the number of log entries it involves.

The general process for determining the mimicking algorithm used for the printing machine data is shown in Figure 7. The dependencies and patterns within the captured data are analysed and the analysis results are validated by experts. From these findings models for stochastic and deterministic simulation of the data is generated.

The main idea behind the simulation model for event data is to generate an event stream consisting of correlated and unrelated events. Figure 8 shows an event sequence that contains the events A, B and C.

Event A precedes event B with a certain time lag $t_{A,B}$, where $t_{A,B}$ is a random variable with the probability distribution function $f(t)$. An example is the normal distribution (cf. Figure 9):

$$f : \mathbb{R} \to \mathbb{R}, \quad f(t) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(t-t_{A,B})^2}{2\sigma^2}}$$

In some cases the probability distribution function might be more complex or even simpler. Some events might occur more often when the machine is started. In this case the distribution would rather look like the one in Figure 10.
In order to implement the mimicking algorithm we follow a generic approach that can be adopted to other event data as well. We define event sequences as graphs with events as nodes and distribution functions, calculating the lag time $t_{A,B,n}$ between the events, as edges of the graph (cf. Figure 11).

Figure 11 shows a simple example of an event graph where events D and E are not related to any other events and event C is correlated with event B which is correlated with event A. What we need to make sure is that a succeeding event should not occur before its ancestor. This means that the value of the probability distribution cannot be negative. The general approach allows us to easily represent arbitrary event networks in different domains.
5.5 Weidmüller molding machines

The Weidmüller dataset comprises 120-dimensional vectors. Basically, the dataset consists of readings taken from sensors deployed on a plastic injection molding machine. The sensors can measure various parameters of the production process like distance, pressure, time, frequency, volume, temperature, time, speed or force. Each measurement is a 120 dimensional vector consisting of values of different types, like fractional or decimal, and has a timestamp.

The first step of our mimicking approach is to automatically classify all dimensions in three groups: constant, trending phases and stateful. A constant dimension has only one value for all data instances. A trending phase is the dimension that exhibits ascending or descending growth. All the other dimensions are considered stateful.

On the second step we take each individual dimension and apply a mimicking technique based on how the dimensions have been classified. For a constant dimension we take a random constant which is not far away from the original dimension’s value. For a trending phase we take the first value, an increment value and produce every next value as the previous value + increment. At random time moments we subtract some other value.

For a stateful dimension we follow a more sophisticated scheme that allows us to mimic states and state transitions found in the original data. This scheme includes the following steps:

1. Clustering of the dimension with the k-means algorithm using automatically computed k for this dimension. This allows us to assign a cluster to each data instance in the dimension.

2. For each cluster we compute mean value and standard deviation.

3. Iterating throughout the dimension we compute one step cluster transition probabilities (Markov model).

We use this model to generate simulated data by randomly walking through the Markov model. When...
we enter a new vertex in the Markov model we generate a data instance using the cluster parameters (mean value and standard deviation) associated with the visited vertex. That data instances constitute simulated data and the algorithm is able to produce as much data as needed.

5.6 Floating car data

In the real world Floating Car data are collected by connected cars, navigation devices and apps. This means practically, that based on the requirements, settings and technical infrastructure different ways of collecting and storing is used. If you are looking for information to detect traffic jams in real time the measuring interval of the GPS positions is less important as it is easy to estimate the travel time in road networks; but if you want to analyze acceleration models at intersections it is sufficient to collect them accumulated but with a higher precision at a later date. However, as all these data are subject to strict rules on data privacy and data protection it was required to provide artificial floating car data to fulfill the requirements.

Each single Point (GPS-Fixes) is usually stored in the following form:

```
<unix time stamp [ms]> <longitude [°]> <latitude [°]> <speed [m/s]>
```

The traces are usually stored in the following form, and one trip containing the single entries are stored in one file.

```
1305093212000 13.587170 52.425710 8.33
1305093213000 13.587030 52.425690 8.33
1305093214000 13.586920 52.425690 8.33
1305093215000 13.586810 52.425660 8.33
1305093216000 13.586730 52.425650 3.89
1305093217000 13.586680 52.425650 3.89
1305093218000 13.586620 52.425640 3.89
1305093219000 13.586580 52.425630 3.89
1305093220000 13.586530 52.425630 3.89
1305093221000 13.586470 52.425630 3.89
1305093222000 13.586370 52.425600 3.89
1305093223000 13.586310 52.425590 3.89
1305093224000 13.586250 52.425580 3.89
1305093225000 13.586200 52.425580 3.89
1305093226000 13.586140 52.425570 3.89
1305093227000 13.586090 52.425640 12.50
1305093228000 13.586040 52.425740 12.50
```

For the implementation of the mimicking algorithm it is required to have a map containing a routable road network graph and a routing algorithm. In order to configure the algorithm it is required to provide some information on origin and destination areas and the number of trips. These statistics are provided as histograms and are based on data that are similar to real world distributions that have been generated from the historical data archive.

The trip generator starts with a random origin node and a destination node based on a sample distance. For this A-B relation a fastest route is calculated, and afterwards a trip is simulated with a given start time. During the trip simulation that is near to some characteristics of real world data the GPS noise is simulated in the data and the trip information are written to the files system. The output format can be either csv, kml or RDF.
Figure 12: An example route with single data points

References

