PDGF

Parallel Data Generation Framework

-a very short Handbook-

by Michael Frank
v.1.0
24.06.2010
intoccabile@gmx.de

Inhaltsverzeichnis

1.Die Befehle....2
2.Datenbankschemaabbildung....3
3.Schreiben eines Plugins....4
3.1.Überblick....4
3.2.Schreiben eines Generator plugins....6
a)Anzeigen bereits existierender Generatoren und informaionen zu ihen....6
b)Erstellen eines Plugin Projekts....6
c)Das erste simple plugin....8
d)hinzufügen eigener nodeParser....9
e)bestehende Parser konfigurieren....11
f)initialize()....11
4 Verteiltes starten....12
5 JVM options....12
6 Bekannte Probleme....13

1. Die Befehle

Ausgabe von java -jar pdgf.jar -help

```
Command line usage:
-------------------------------------------------------------------------------
pdgf.jar -load <path to project config xml> -load <path to node config xml> -clo
seWhenDone -start

pdgf.jar -load <path to project config xml> -nodeNumber 1 -nodeCount 10 -noShel
l-closeWhenDone -start

Note: some commands depend on other commands. For example: you can not start da
tageneration if the config files are not loaded. The command line parameters ar
e processed sequentially. So the -start command must allways be the last one.

Shell usage:
-------------------------------------------------------------------------------
load          <Filename>
                read and Parse a XML configuration file (node or project config
                ).
                NOTE: Replaces previously imported configuration without asking
                .

nodeNumber    <number>
                number of this Data Generation Node (instances of PDGF)

nodeCount     <number>
                number of participating Data Generation Nodes (instances of PDG
                F)

workers       <number>
                number of threads/workers to be used (optional). Overrides auto
                matic worker determination by cpu count

start
                starts data generation on this node

stop
                stops data generation on this node

closeWhenDone
                automatic quit PDGF when datageneration is done

plugins       <type>
                displays available plugins. Permitted values for <type>: { all
                | generator | distribution | output | scheduler | random}

noShell
                non interactive mode: disables PDGF shell

plugin        <name>
                displays information on a plugin. Parameter <name>: the plugin
                name (like myProject.generator.MyCustomGenerator)

help
                displays all aviable commands with a short description and need
```

```
                ed parameters

testgen       <TableName> <FieldName><iterations>(optional)

                Tests the generator (configured via xml) for the specified  fie
                ld in specified table

benchGen      <TableName> <FieldName> <time>(optional) OR ALL <time>

                Benchmarks a generator for a specified field in a table (must b
                e configured via xml) or if the first parameter equals "ALL", g
                enerators of all fields and tables are benchmarkt. For both use
                s you can specified the  <time> in ms a benchmark should run


exit
                quits PDGF shell, stops all workers and exits programm
```

2. Datenbankschemaabbildung

Das Datenbankschema wird als xml Datei beschrieben.

Dabei wird für jedes Feld (Spalte) in einer Datenbanktabelle angegeben mit welchem Generator die Werte generiert werden und wie diese verteilt sein sollen. Wird keine Verteilung angegeben so sind die Werte uniform verteilt.

Zunächst ein Beispiel einer simplen XML config

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="min">

      <!--
            Wenn größe der Table "foo" size=100 und sf 10 werden 1000 Zeilen
            generiert
      →
      <scaleFactor>10000</scaleFactor>

      <seed>1234567890</seed>
      <rng name="PdgfDefaultRandom"></rng><!-- DEFAULT -->
      <scheduler name="FixedJunkScheduler"></scheduler><!-- DEFAULT -->
      <output name="CSVRowOutput"><!-- DEFAULT -->
            <outputDir>/tmp/someDir/</outputDir>
            <fileEnding>.csv</fileEnding>
            <delimiter>;</delimiter><!-- Felder Trennzeichen in CSV File-->

            <!--
            sollte der clustersize des verwendeten dateisystems entsprechen,
            oder  ein vielfaches davon
            -->
            <bufferSize></bufferSize>
            <compress>false</compress><!--file ending +=".zip"-->
            <compressionLevel>1</compressionLevel> <!--1 niedrigstes, 9
höchstes-->
      </output>
      <tables>
            <table name="tabelle1">
```

```
                <size>1000</size>
                <fields>
                        <field name="id">
                                <type>java.sql.Types.INTEGER</type>
                                <primary>true</primary>
                                <unique>true</unique>
                                <size></size>
                                <generator name="IdGenerator">
                                        <distribution name=""></distribution>
                                        <rng name=""></rng>
                                </generator>
                        </field>
                        <field name="name">
                                <type>java.sql.Types.VARCHAR</type>
                                <size>20</size>
                                <generator name="DictList">
                                        <distribution name=""></distribution>
                                        <size>2</size> <!-- two names! and name1 !=
name2-->
                                        <unique>true</unique>
                                        <file>dicts/Given-Names.dict</file>
                                </generator>
                        </field>
                        <field name="Familyname">
                                <type>java.sql.Types.VARCHAR</type>
                                <size>20</size>
                                <generator name="DictList">
                                        <distribution name=""></distribution>
                                        <file>dicts/Family-Names.dict</file>
                                </generator>
                        </field>
                        <field name="age">
                                <type>java.sql.Types.INTEGER</type>
                                <size>20</size>
                                <generator name="IntGenerator">
                                        <distribution name="LogNormal">
                                                <mu>7.60021</mu>
                                                <sigma>1.40058</sigma>
                                        </distribution>
                                        <min>1</min>
                                        <max>100</max>
                                </generator>
                        </field>
                </fields>
        </table>
    </tables>
</project>
```

3. Schreiben eines Plugins

3.1. Überblick

PDGF ist größtenteils ein Blackbox Pluginframework.

„Größtenteils“ weil es um ein Plugin zu schreiben zwar nicht notwendig, über die vorhanden Schnittellen jedoch möglich ist in den Tiefen von PDGF Einstellungen vorzunehmen.

Die Pluginschnitstellen befinden sich gesammelt im Package pdgf.plugin

- **pdgf.plugin.Generator**

  Das wichtigste Plugin. Jedes Feld (Spalte) einer Datenbanktabelle benötigt einen Generator der die Werte dafür generiert.

  Hier muss sich ein Pluginschreiber meist keine sorgen um Parallelisierung oder Concurency machen.

- **pdgf.plugin.Distribution**

  für die Unterstützung zusätzlicher Verteilungsfunktionen

  Hier muss sich ein Pluginschreiber meist keine sorgen um Parallelisierung oder Concurency machen.

- **pdgf.plugin.AbstractPDGFRandom**

  Falls ein eigener Zufallszahlengenerator benötigt wird.

  Hier muss sich ein Pluginschreiber meist keine sorgen um Parallelisierung oder Concurency machen.

- **pdgf.plugin.Output**

  Wenn die generierten Daten anders ausgegeben werden sollen als in csv dateien. (beispielsweise als insert statements)

  Im Gegensatz zu den oberen Pluginklassen ist es hier nicht so trivial ein eigenes Plugin zu schreiben.

  Ein Output Plugin kann und wird sehr schnell zum Flaschenhals des Frameworks werden! Alle Worker liefern ihre Daten beim zentralen Outputplugin ab.

  Je nach eingesetzter Hardware und Generatoren kann die Datenmenge zwischen ~10Mb/s bis 200Mb/s liegen.

  Zur Synchronisierung der Schreibvorgänge zwischen den Workern übergibt jeder worker seine eindeutige ID beim Aufruf der Schreibmethode:

  ```
  public abstract void write(RowDataDTO r, int threadID)
  ```

  Aus performance gründen sind die DTO´s (data transfer object) sog. Shared reusable objects. Das heist dass von ihnen nur eine instanz existiert und sie, sobald sie vom Output plugin abgearbeitet wurden, von den Generatoren sofort wieder verwendet, und die alten Werte mit neuen überschrieben werden.

- **pdgf.plugin.Scheduler**

  Andere Aufteilung der Arbeit auf die Worker eines Knotens (beispielsweise kleinere WorkUnits oder Netzwerksupport)

  Im Gegensatz zu den oberen Pluginklassen ist es hier ebenfalls so trivial ein eigenes Plugin zu schreiben. Alle workerthreads fordern beim Scheduler nach producer consumer pattern parallel workunits an.

  Wie auch beim Output plugin sind die WorkUnits shared reusable objects.

3.2. Schreiben eines Generator plugins

Bereits vorhandene Generatoren können im interaktiven Modus von PDGF (Standard) in der Shell mit dem Befehl: `plugins generator` angezeigt werden.

Details zu einem einzelnen Generator können mit plugin <package.GeneratorName> angezeigt werden.

a) Anzeigen bereits existierender Generatoren und Informationen zu ihnen.

Beispiel:

pdgf> plugin pdgf.generator.IntGenerator

```
Details for: pdgf.generator.IntGenerator
-------------------------------------------------------------------------------
Tag usage: <generator name="pdgf.generator.IntGenerator">

Supported tag attributes:
-------------------------
Needed |Attribute name     |Description
-------------------------------------------------------------------------------
true   |name=".."          |DefaultParser. (Class)Name of this element. Used to
                            identify plugin Class. Full name is required. Examp
                            le: com.en.myPluginPackage.myPuginClass

Description:
------------
Generates an integer value between <min> and <max>, distributed as spcified in <
distribution>
-------------------------------------------------------------------------------

Registred Child Node Parsers:
-----------------------------
Needed |Tag name           |Description
-------------------------------------------------------------------------------
false  |<seed>             |DefaultParser. Random number generator seed of this
                            Element. Overrides default seeding behaviour.
false  |<rng>              |DefaultParser. Name of random number generator class
                             to be used for calculations in this element.  Exam
                            ple: com.en.myRNG
false  |<size>             |DefaultParser. Size of this element. For tables this
                             is the number of rows.
true   |<min>              |DefaultParser. Min Value of this element. Can be use
                            d by Field description or in Plugins.
true   |<max>              |DefaultParser. Max Value of this element. Can be use
                            d by Field description or in Plugins.
false  |<file>             |Default for all Generators. A path to a file to be u
                            sed by a generator. See pdgf.generator.DictList sou
                            rce for a example.
false  |<distribution>     |Distribution to be used by a generator when calculat
                            ing a value
```

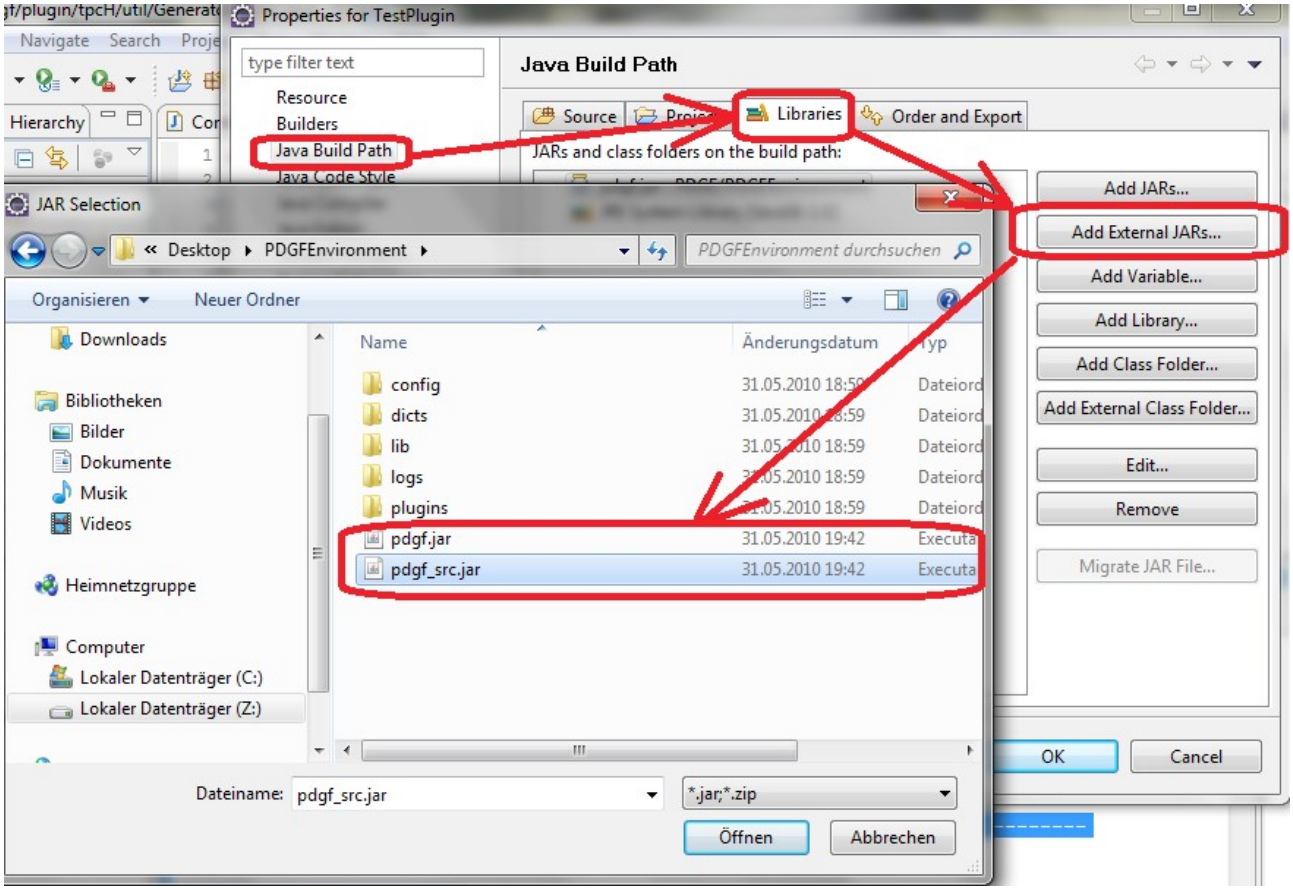
b) Erstellen eines Plugin Projekts

Um einen zusätzlichen eigenen Generator zu implementieren reicht es eine neue Klasse anzulegen

und pdgf.plugin.Generator zu erweitern.

Am besten ist man legt zuerst mal (in eclipse) ein neues Projekt an.

Dort passt man zunächst den Build Path an und bindet die pdgf.jar aus dem Root folder des PDGFEnvironment ein(oder zum testen die pdgf_src.jar) ein.



c) Das erste simple plugin

Nachdem erzeugen des Grundgerüsts für ein Plugin sollte unbedingt der javadoc text zu getNextValue gelesen werden!

Deer Generator Plugin Writer:
Most of the time you do not need to worry about concurrency issues because an instance of your generator is created per thread/worker. The instances of your generator are saved as virtual child's of parent field where Child ID == thread/worker ID

Some warnings!
Be very careful when using the "static" modifier. Most JDK utility classes (like SimpleDate or other formatter classes) are NOT threadsafe!. Its wise to use "static" only for primitives or own classes where you know exactly how they work.
If you really need a static "state" between your instances (f.e. caching of generated values) you may consider to link it with a specific thread/worker! Perhaps one state or one buffer per thread! The id of the calling thread can be found in "generationContext" GenerationContext.getWorkerID() "generationContext". You may consider the use of ThreadLocal for this purpose:

```
// a ReferenceState per thread
static ThreadLocal<ReferenceState> perThreadReferenceState = new
ThreadLocal<ReferenceState>() {
protected ReferenceState initialValue() {
return new ReferenceState();
} };
```

The abstract Class Generator is the basic class for all specific data generators. It offers methods for parsing often needed parameters. Further parameters can be included via the xml-subtree. Every implementation of Generator is responsible for parsing is own aditional needed parameters.

Overrides: getNextValue(...) in Generator

Parameters:
rng the Pseudo Random number generator to be used if needed. If a distribution is used, this rng should be passed down to distribution.nextValue()

generationContext Containing information about the current state of the invoking Worker. (f.e. the current row and id of the worker thread)

currentFieldValue Shared data object! put your generated data in here with: currentFieldValue.setValue(Generated result)

Normalerweise muss man beim schreiben eines Generator Plugins nicht viel beachten. Paralleliserungs- und Concurencyprobleme wurden weitestgehend abstrahiert.

Von einem Generator werden beim Start der Datengenerierung für jeden Worker eine eigene Instanz erzeugt. Besondere Vorsicht ist daher bei der Verwendung des „static“ modifiers geboten.

Die wichtigste Eigenheit ist jedoch der Methodenparameter `FieldValueDTO currentFieldValue` DTO steht für „data transfer object“. Dieses Object existiert pro workerinstanz nur einmal! Für alle Generatoren. Es wird also ständig wiederverwendet. Diese Abweichung von der strikten Objektorientierten Vorgehensweise hat Performancegründe.

Für den Pluginschreiber ist nur zu beachten dass er seinen Generierten Wert in diese Shared object zu speichern hat mit `currentFieldValue.setValue(value);` die übrigen Felder dieses Objects jedoch unangetastet lässt.

Ein Beispiel für ein erstes Plugin:

```
package myPlugins;

public class MyFirstGeneratorPlugin extends Generator{

	public MyFirstGeneratorPlugin() {
		super("Eine zwingend vorgeschriebene Beschreibung!");
	}

	@Override
	public void getNextValue(AbstractPDGFRandom rng,
			GenerationContext generationContext, FieldValueDTO
currentFieldValue) {

		String value = "some generated value...do your magic here";
		value += " or add some random value to it between, as example: [0
currentRow]";
		value+= (rng.nextLong()% generationContext.getCurrentRow());

		//safe your generated value in shared object.
		currentFieldValue.setValue(value);
	}
}
```

Einbinden in die config file:

```
<field name="myFirstOwnField">
	<type>java.sql.Types.VARCHAR </type>
	<size>200</size>
	<generator name="myPlugins.MyFirstGeneratorPlugin">
		<distribution name=""></distribution>
	</generator>
</field>
```

d) hinzufügen eigener nodeParser

Oft werden jedoch für ein Plugin noch zusätzliche Informationen benötigt die man am liebsten über die XML datei einlesen lassen möchte.
Dies ist ebenfalls kein Problem. Man kann jederzeit eigene Nodes hinzufügen und einen Entsprechenden Parser bei der Oberklasse registrieren.

Als Beispiel der Quellcode des Simplen Standardgenerators: StaticValueGenerator.
Dieser macht nichts anderes als einen Festen Text ohne Modifikation als Ergebniss zurück zu geben. Dieser text soll in einer xml node names <value> stehen.

Dazu muss ein eigener Parser als interne klasse für diese node integriert werden mit

class ValueNodeParser extends Parser<Generator> {…}

und bei der oberklasse registiert werden:

```
protected void configParsers() {
		super.configParsers(); // muss vor eigenen Parsern stehen.
		addNodeParser(new ValueNodeParser(true, this));
}
```

Beispielcode:

```
package pdgf.generator;

public class StaticValueGenerator extends Generator {

	public static final String NODE_PARSER_value = "value";
	private char[] value = null;

	public StaticValueGenerator() {
		super("For all rows the same value is used as specified in <"
				+ NODE_PARSER_value + ">The value to be used</"
				+ NODE_PARSER_value + ">");
	}

	@Override
	public void getNextValue(AbstractPDGFRandom rng,
			GenerationContext generationContext, FieldValueDTO
currentFieldValue) {

		// set next value result
		currentFieldValue.setValue(value);

	}

	protected void configParsers() {
		super.configParsers();
		addNodeParser(new ValueNodeParser(true, this));

	}

	private class ValueNodeParser extends Parser<Generator> {

		public ValueNodeParser(boolean required, Generator parent) {
			super(required, NODE_PARSER_value, parent,
					"The static value to be used for all rows.");

		}

		@Override
		protected void parse(Node node) throws XmlExecption {
			String nodeText = null;
			if (node == null || (nodeText = node.getTextContent()) == null
					|| nodeText.isEmpty()) {
				if (this.isRequired()) {
					throw new XmlExecption(getNodeInfo() + "<" +
this.getName()
							+ "> must not be empty.");
				}
			} else {
```

```
                        value = nodeText.toCharArray();
                    }
                }
        }
}
```

Aussehen des entsprechenden config file gegenparts

```
<field name="myFirstOwnField">
      <type>java.sql.Types.VARCHAR </type>
      <size>200</size>
      <generator name="StaticValueGenerator">
            <distribution name=""></distribution>
            <value>SOME VALUE</value>
      </generator>
</field>
```

e) bestehende Parser konfigurieren

Es existieren bereits einige bereits integierte Parser auf die zurück gegriffen werden kann.

Diese können ebenfalls über die configParsers() configuriert werden. Benötigt ein Generator beispielsweise int oder long werte zwischen min und max können die entsprechenden parser dafür auf „required“ geschalten werden.

```
@Override
protected void configParsers() {
      super.configParsers();

      getNodeParser(NODE_PARSER_min).setRequired(true);
      getNodeParser(NODE_PARSER_max).setRequired(true);

}
```

Der Zugriff auf die entsprechenden Werte erfolgt in der Regel über die jeweiligen getter mit dem selben Namen:

```
getMax();
getMin();
```

f) initialize()

Sollte nach! dem parsen der Config file und vor! Start der Datengenerierung noch Konfigurationsarbeit geleisted werden (Vorberechnung irgendwelcher Werte oder Check von Werten/Nodes die sich gegenseitig bedingen, so kann dies über die methode:

```
@Override
public synchronized void initialize(int workers)
            throws ConfigurationExecption, XmlExecption {
      super.initialize(workers);

      //your initialze code here
}
```

geschehen. Bitte verwende dazu wirklich diese Methode, da dies der einzige Weg ist, sicherzustellen dass entsprechende Werte bereits geladen, geparst wurden.

Methodenaufrufhirachie:

- `configParsers()`
- `Subclass constructor`
- `(internal: load config file)`
- `initialize(int workers)`
- `getNextValue(AbstractPDGFRandom, GenerationContext, FieldValueDTO)`

4 Verteiltes starten

PDGF benötigt keine Synchronisierung/NetzwerkIO zwischen den Rechnern auf die der Generierungsprozess aufgeteilt wird wenn als Scheduler „FixedJunkScheduler“ verwendet wird.

Daher ist bisher auch kein Netzwerksupport in PDGF vorgeshen.

Dies hat jedoch zur Folge, dass der Anwender sich selbst um das verteilte Starten von PDGF kümmern muss. Dazu muss PDGF nebst benötigter Ressourcen (dictionarys, config files, etc) natürlich auf jedem Rechner (Node) verfügbar sein.

Wichtig ist, dass jeder PDGF Instanz mitgeteilt wird für welchen Teil der gemeinsamen Arbeit sie zuständig ist.

Mit -nodeCount wird angegeben auf wie viele Rechner der Generierungsprozess aufgeteilt werden soll.

Mit -nodeNumber wird die Node ID [1, nodeCount] der Instanz angegeben.

Im Environment unter PDGFEnvironment\shellScripts\ finden sich einige Beispiele für das verteilte Starten und managen von PDGF unter Linux.

Diese wurden speziell für den Multimediacluster des Lehrstuhls Kosch der Universität Passau geschrieben und starten PDF auf den Rechnern „node1“ bis „node16“ per ssh.

Generell ist es beim Einsatz von PDGF über Scripte empfehlenswert den interaktiven Modus, die interne Shell, mit -noShell abzuschalten.

Grundgerüsts für Aufruf aus scripts heraus:

java $VMOPTIONS -server -jar pdgf.jar -load $PROJECTCFG -nodecount $NODECOUNT -nodenumber $CURENTNODE -workers $WORKERS -closeWhenDone -noShell -start

5 JVM options

Prinzipiell verbraucht PDGF nur sehr wenig Speicher. Werden jedoch große/viele dictionary Files geladen so wird auch mehr Speicher benötigt da diese aus Performance gründen vollständig in den Speicher gepuffert werden.

Bei Performance Tests haben sich auf einem Dual Prozessor System bestehend aus zwei Quad Core Intel(R) Xeon(R) CPU E5405 @ 2.00GHz unter Kernel 2.6.22.17-0.1-default #1 x86_64 GNU/Linux
folgende JVM Startoptionen bewährt:

-server -Xms512m -Xmx1024m -XX:SurvivorRatio=16 -XX:PermSize=128m -Xmn256m -XX:+AggressiveOpts -XX:+UseFastAccessorMethods

Diese Einstellungen sind keinesfalls optimal, bringen auf dem beschriebenen System gegenüber den „defaults“ ca 10-20% Leistungszuwachs.

Die server VM sollte auf jeden Fall der client VM vorgezogen werden da Startzeiten können bei langen Datengenerierungen vernachlässigt werden können.
In der aktuell verwendeten JRE version: jre1.6.0_20 ist die 32Bit version signifikant schneller als die 64Bit version.

Weiterhin ist es empfehlenswert die Größe des new Generation Heaps zu vergrößern da dort die Garbage Collection günstiger ist. Falls im Generierungsprozess neue Objekte angelegt werden so sind dieses nur von kurzer Lebensdauer. Daher ist die Größe und Garbagecollection des new Generation heaps eine der wichtigsten tuning Einstellungen.

6 Bekannte Probleme

- Pausieren/Wiederaufnahme der Datengenerierung nicht implementiert im FixedJunkScheduler. Worker sollten dazu in der Lage sein, wobei dieses Feature bis jetzt nicht getestet wurde.

- Verteilte Generierung im interaktiven Modus (mit shell) führt zu Endlosschleifen da das Startscript ständig Returns in die PDGF shell piped.

  Lösung:

  Starten mit -noShell

- <Distribution> ist ein Feature das in den Standardgeneratoren nur unzureichend implementiert ist. Einige der Standardgeneratoren beachten <Distribution> gar nicht. Bei anderen liefert die Ausgabe nicht das gewünschte Ergebniss da beispielsweise min/max Grenzen in Zusammenhang mit <Distribution> nicht beachtet werden.

- Die Generatoren des TPCH Plugins beachten die <Distribution> nodes überhaupt nicht da sie für standard TPC-H nicht benötigt wurden.

- Ausgabe in .csv Dateien ist unsortiert da sie zeilenweise von den Threads in der Reihenfolge geschrieben werden in der sie ankommen.

- Verwendung auf Mehrprozessoren Systemem (beispielsweise der Cluster des Lehrstuhls kosch). Damit ist gemeint dass ein Mainboard über 2 Sockel für CPU´s verfügt. Es ist der Java VM nicht möglich ihre Threads auf 2

physikalisch getrennte CPUs zu verteilen. Daher muss auf solchen Systemen zwei Instanzen von PDGF gestartet werden um das gesamte System auszulasten.