TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Datenbanken

Belegarbeit

KONZEPTION UND IMPLEMENTIERUNG VON WORKLOADMODELLEN IM MULTI-TENANCY DATABASE BENCHMARK FRAMEWORK (MULTE)

Hendrik Schön

Matr.-Nr.: 3703748

Betreut durch:

Dipl.-Inf. Tim Kiefer

Eingereicht am 10. Mai 2015

ERKLÄRUNG

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, 10. Mai 2015

ZUSAMMENFASSUNG

Ein gutes Modell eines Workloads ist essentieller Bestandteil eines Benchmark-Systems, um realistische und zuverlässige Werte als Resultate zu erhalten. Sie spiegeln die Möglichkeiten und den Umfang sowie die Art und Weise wider, inwieweit der durch den Benchmark repräsentierte Nutzer mit dem zu testenden System interagiert.
In dieser Arbeit wird sich mit den konzeptionellen Modellen von Workloads für Benchmarks befasst. Dabei ist die Thematik dieser Arbeit darauf ausgelegt, ein neues und komplexeres Modell für Tenant-Workloads zu konzipieren, es mit anderen möglichen Modellen zu vergleichen und es dann in geeigneter Form als Prototyp in das Multi-Tenancy Benchmark Framework *MulTe* zu implementieren. Eine kurze Analyse der Ergebnisse und deren Evaluierung soll die gewählte Implementierung begründen sowie die Thematik abrunden. Zum Schluss wird noch kurz auf Verbesserungen eingegangen, welche in Zukunft die Möglichkeiten des entwickelten Modells noch steigern könnten.

INHALTSVERZEICHNIS

1 Einleitung **9**

1.1 Einführung in Benchmarks und Workloads 9

1.2 Motivation für Nutzermodelle 10

2 Grundlagen von Benchmarks und Workloads **13**

2.1 Benchmark-Systeme und Funktionsweisen 13

2.2 Das *MulTe*-Framework 15

2.3 Ähnliche und Weiterführende Arbeiten 19

3 Formales Workload-Modell eines Nutzers **23**

3.1 Das Konzept der Intensität 23

3.2 Evaluierung verschiedener Modelle 25

3.3 Konstruktion des neuen *MulTeX*-Modells 28

4 Technische Umsetzung in *MulTe* **31**

4.1 Konzeptionelle Umsetzung 31

4.2 Technische Umsetzung 34

4.3 Benutzung der Erweiterung 42

5 Evaluierung der neuen *MulTe*-Erweiterung **43**

5.1 Log-Schnittstellen in *MulTe* 43

5.2 Testumgebung und -Szenarien 44

5.3 Auswertung der Tests 49

5.4 Evaluierung der Umsetzung des Modells 54

6 Zusammenfassung und Ausblick **57**

7 Anhang **59**

1 EINLEITUNG

1.1 EINFÜHRUNG IN BENCHMARKS UND WORKLOADS

Um ein bestehendes oder neu entworfenes Software-System zu evaluieren, müssen konkrete Testfälle generiert werden. Das Ziel eines solchen Tests ist immer die Einschätzung des Systems gegenüber einer funktionalen oder nicht-funktionalen Eigenschaft, welche im Idealfall erfüllt sein sollte. Solche Eigenschaften eines Systems sind beispielsweise die Stabilität, die Verfügbarkeit, Effizienz und Effektivität, Sicherheit oder auch Performance. Für jede dieser Eigenschaft sind mehrere Test-Verfahren etabliert, genannt seien hier nur beispielsweise die klassischen jUnit-Tests oder auch die Regressions-Tests. Um die Performance eines Systems zu testen, bietet sich ein Benchmark-Tests an. Ein Benchmark-Tests hat zweierlei Vorteile. Erstens evaluiert ein solcher Test das System auf die Performance-Eigenschaft und gibt Auskunft über das Verhalten des Systems während des Tests. Zweitens ist ein Benchmark-Tests meist derart standardisiert, sodass man sein eigenes getestetes System mit anderen Systemen vergleichen kann, wenn diese ebenfalls mit den gleichen Benchmark-Test evaluiert wurden. Somit gibt also ein Benchmark-Test nicht nur Auskunft über die Performance an sich, sondern stellt dazu noch geeignete Werte dar, um diese Performance mit der anderer Systeme effizient zu vergleichen. Der Benchmark an sich ist dabei nur ein Skala, ein Wertebereich, an dem sich das System messen lassen muss. Der Benchmark kann sehr umfangreich sein, oder aber auch sehr leichtgewichtig. Ein solcher Benchmark-Tests läuft vollkommen automatisiert ab und versucht, je nach seinem genutzten internen Benchmark, das zu testende System derart zu strapazieren, um eine valide Aussage über die Performance zu bekommen. In den meisten Benchmark-Tests bedeutet dies, dass der Benchmark bestimmte Aufgaben bereit hält, welche das System durch den Benchmark-Tests zum Lösen gefordert bekommt. Je nachdem, wie schnell das System dann die gelösten Aufgaben an den Benchmark-Tests zurückliefert, wird die Einschätzung der Performance des Systems vorgenommen. Wird demzufolge der gleiche Benchmark gegen ein anderes System evaluiert, ergibt sich zwangsweise ein Vergleich, welches System, unter dem durch den Benchmark definierten Kontext, das performantere von beiden ist und demzufolge mehr Leistung bietet.

In dieser Arbeit wird das Testen von Datenbanken vorgenommen. Konkret bedeutet dies, dass die Management-Systeme der Datenbanken (ab hier DBMS genannt) durch den Benchmark-Test

gestellte SQL-Transaktionen lösen und eine korrekte Antwort zurückliefern müssen. Um diese sinnvoll gegeneinander evaluieren zu können, benötigt man einen Benchmark, welcher die dazu nötigen Aufgaben bereit hält. Ein prominentes Beispiel wäre hier der *TPC-H*-Benchmark, welcher unter [1] frei verfügbar ist. Der *TPC-H*-Benchmark ist ein Satz von Definitionen für Datenbank-Schemata, -Constraints und -Anfragen. Er definiert zum Beispiel, welche Tabellen das zu testende System bereithalten muss, wie viele Datensätze darin enthalten sind und mit welchen Primär- und Sekundär-Schlüsseln diese Tabellen miteinander verknüpft sind. Allein dieser *TPC-H*-Benchmark reicht allerdings nicht aus, denn er definiert hierbei nur die Grundlage des Tests. Um den Test automatisch führen zu lassen, benötigt man eine weitere Kompontente, das Benchmark-Tool (oder -Toolkit), beziehungsweise ein Benchmark-Framework. Dieses generiert auf Grundlage des Benchmarks (in dem Beispiel *TPC-H*) automatisch die benötigten Anfragen und schickt sie an das DBMS. In den meisten Fällen übernimmt das Tool ebenfalls noch die Analyse der Antworten des Systems, um daraus einen geeigneten Wert als Vergleichswert zu bilden. Korrekterweise müsste man ab hier von einem Nutzer, beziehungsweise einem Tenant, sprechen, den das Benchmark-Tool gegenüber dem DBMS darstellt, denn die Beschaffenheit des Benchmarks ist stark an reale Nutzung angelehnt, um möglichst originalgetreue Werte in realistischen Situationen messen zu können.

Um den Benchmark-Tests vergleichbar machen zu können, fehlt jedoch noch eine wichtige Komponente in dem Test. Das Tool, oder das Framework, führt bisher die Anfragen an das DBMS aus und stützt sich dabei auf die Vorgaben des eigentlichen Benchmarks. Jedoch wird bislang in keiner Weise spezifiziert, wie die Anfragen an sich aussehen. Da der Benchmark nur über die Beschaffenheit der Daten Auskunft gibt, um den Test später vergleichbar zu machen, muss auch die Art der Anfragen vergleichbar gemacht werden, damit sich später eine geeignete Evaluation mit anderen DBMS bestimmen lässt. Um die Anfrage-Spezifikation kümmert sich ein Workload, welcher definiert, welche Anfrage mit welchen Parametern wie oft und wann genau ausgeführt wird. Ist diese Definition durch einen Workload gegeben, kann das Benchmark-Tool oder -Framework genau vordefinierte Transaktionen an das DBMS absetzen und dessen Antwort abwarten. Ab diesen Punkt ist ein Benchmark-Test jederzeit vergleichbar wiederholbar, da alle Parameter für den synthetisierten Nutzer, den das Tool darstellt, festgelegt sind. Eine zusammengefasste Übersicht aller Komponenten eines Benchmark-Tests findet sich in Abbildung 1.1. Zu beachten ist hier, dass der *Nutzer* nicht gleich dem eigentlichen *Benutzer* ist, sondern nur ein möglichst realistisch wirkender und durch das Benchmark-Tool synthetisierter Nutzer mit dem Benchmark als Datenbasis und dem Workload als Verhaltens-Spezifikation. Daher besteht der Nutzer als Ganzes immer aus diesen zwei Komponenten, das Nutzerverhalten jedoch nur aus dem Workload. Da der *TPC-H*-Benchmark keinen Workload beinhaltet, werden diese meist durch die Tools oder Frameworks mitgeliefert.

1.2 MOTIVATION FÜR NUTZERMODELLE

Wie oben beschrieben, ist ein Benchmark wie *TPC-H* bereits realitätsnah angefertigt und kann als kompletter Satz heruntergeladen werden. Es stellt sich dabei aber zwangsläufig die Frage nach der Funktionsweise und Beschaffenheit des Workloads, den das Tool gegenüber dem System anwendet. Es wäre hier von enormen Vorteil, ebenfalls einen realitätsnahen Workload zu schaffen, sodass Benchmark und Workload in Kombination einen möglichst realistischen synthetisierten Nutzer darstellen. Viele Parameter spielen bei einem Workload für einen Benchmark eine Rolle,

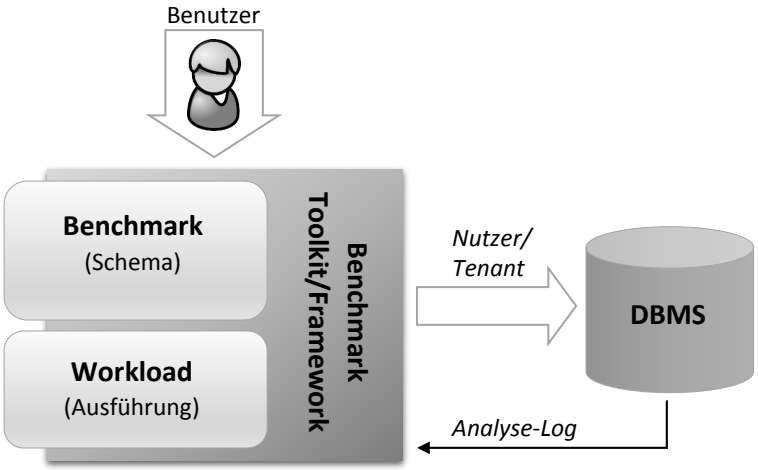


Abbildung 1.1: Zusammenhänge im Benchmark-Umfeld

zum Beispiel die Interaktionsmöglichkeiten des Nutzers mit dem System, die Möglichkeit der Variabilität der Anfragen des Nutzers und anderes. Je mehr Parameter das Tool zur Darstellung des Nutzerverhaltens bekommt, umso geeigneter kann es einen realen Nutzer nachahmen und umso realistischer kann das DBMS am Ende getestet werden. Hierbei gibt es jedoch gravierende Unterschiede, inwieweit ein Workload ein geeignetes Nutzerverhalten darstellen kann oder nicht. So besitzt ein einfaches „An/Aus"-Nutzerverhalten (mehr dazu in den folgenden Kapiteln) in den meisten Fällen weniger realistische Züge, als ein anderer Nutzer, welcher aus einem Pool von Variationen und Parametern unterschiedlichste Verhaltensweisen abseits von „An/Aus" synthetisieren kann. Die großen Schwierigkeiten bei solch einem verbesserten Nutzerverhalten sind insbesondere:

1. die Modellierung des realistischen Nutzerverhaltens kann aufgrund vieler Parameter recht schnell komplex werden,
2. die Umsetzung der gegebenen Parameter in einer sinnvollen und realistischen Weise gegenüber dem System kann sehr anspruchsvoll werden.

Eine Besonderheit bilden hierbei gewisse *Multi-Tenancy Benchmarks*, welche dafür ausgelegt sind, möglichste viele synthetisierte Tenants gleichzeitig an einem DBMS testen zu können. Dabei beinhaltet ein Tenant mehrere Nutzer, und der Tenant stellt die übergeordnete Einheit dar, die abstrakte Ebene zwischen Workload und Nutzer, wobei die einzelnen Tenants untereinander absolut unabhängig sind. Dieses Vorgehen eignet sich besonders gut, um Multi-Nutzer-Applikationen testen zu können, wo gleichzeitig mehrere Nutzer auf eine Datenbasis (beziehungsweise ein DBMS) zugreifen werden. Dieses Szenario besitzt dahingehend große Unterschiede zu einem Einzelnutzer-Szenario, dass hier die Fähigkeiten und Performance des DBMS einspielen, welche das Verwalten von mehreren Nutzern sowie deren Transaktionen gleichzeitig angeht. Dies äußert sich zum Beispiel in der *Load Balancing*-Fähigkeit eines DBMS, welches sicherstellen soll, dass die freien Ressourcen des Systems und die durch die Nutzer erzeugte Last auf dem System gleichmäßig verteilen werden. Aber auch bei diesen Multi-Tenancy-Modellen stellen sich die gleichen Fragen wie bei einem Einzelnutzer-Test, inwieweit man durch den Workload eines jeden einzelnen Tenants möglichst verschiedenes, aber dennoch realitätsnahes Nutzerverhalten modellieren kann.

Aus diesem Grund befasst sich diese Arbeit damit, einen (komplexen) Workload zu entwickeln, der die Möglichkeit bietet, eine möglichst realistisches Nutzerverhalten darzustellen. Es besteht der Anspruch, den neuen Workload als formal parametrierbares Modell eines Nutzerverhaltens darzustellen. Es wird untersucht, inwieweit man einen komplexeren Workload schaffen kann und woran man sich dabei orientiert. Es werden dabei die dazu notwendigen Schritte aufgezeigt, um ein neues Modell zu konzipieren und später damit ein Nutzerverhalten formal beschreiben zu können. Es werden ebenso verschiedene Modelle aufgezeigt, welche sich in dieser Hinsicht eignen würden und die entsprechenden Anforderungen erfüllen. Nach der begründeten Auswahl des endgültigen Modells wird vorgestellt, inwiefern dieses ausgearbeitete Workload-Modell in das bereits bestehende Multi-Tenancy Benchmark Framework *MulTe* [6] als Erweiterung einzubauen ist. Es wird in einer Darstellung der Resultate und deren Evaluierung aufgezeigt, inwiefern das entwickelte komplexe Nutzerverhalten als Workload-Modell sich in den resultierenden Ausgaben des Benchmarks äußert. Am Schluss wird noch ein Ausblick auf zukünftige Arbeiten zu dieser Thematik geworfen und Anhaltspunkte gegeben, inwieweit man den Prototypen der Implementierung sowie das eigentliche Modell noch verfeinern kann.

2 GRUNDLAGEN VON BENCHMARKS UND WORKLOADS

2.1 BENCHMARK-SYSTEME UND FUNKTIONSWEISEN

Das vorherige Kapitel hat bereits die Grundkonzepte eines Benchmark genauer beschrieben. Um weiterhin mit den gleichen Begrifflichkeiten zu arbeiten, ist in Tabelle 2.1 eine Übersicht zu finden, in welcher dargestellt wird, wie genau welcher Begriff in dieser Arbeit benutzt wird. Die Funktionsweise ist bei den meisten Tools gleich. Es wird ein zugrundeliegender Benchmark (zum Beispiel wieder *TPC-H*) genutzt, um eine gemeinsame Datenbasis für alle Benchmark-Tests zu schaffen. Im Falle von *TPC-H* wird das gesamte logische Datenbankdesign spezifiziert, welches unter anderem Entitäten, Relationen, Typen und Tabellen beinhaltet. Damit ist bereits sichergestellt, dass alle anderen Benchmark-Tests die gleiche Grundlage der Datenbank besitzen. Für den Workload sind die meisten Benchmark-Tools selbst zuständig, da hier keine genauen Spezifikationen vorliegen. Es ist also nötig, um Benchmark-Tests vergleichbar zu halten, auch hierbei einen gemeinsamen Punkt zu finden. *TPC-H* selbst bringt dazu aber keine Workload-Spezifizierung mit, weshalb Tools oder Frameworks wie *MulTe* selbst derartige Verhaltensregeln mitliefern müssen.

Was genau in diesen Workloads beschrieben werden kann, hängt stark von der eingesetzten Technik ab. In *MulTe* ist dies zum Beispiel eine einzelne XML-Datei pro Tenant, welche die Parameter bereit stellt, um das Nutzerverhalten exakt beschreiben zu können. Solche Parameter können, je nach Komplexität des eingesetzten Workloads, den kompletten Umfang eines Nutzerverhaltens beschreiben oder auch nur grundlegende Eigenschaften. Einer der einfachsten Workloads, eine einfache Ansammlung von Queries, könnte zum Beispiel mit folgendem Pseudo-Code für den Workload ausgeführt werden:

```
for each QUERY in QUERIES: do USER.TRANSACTION( QUERY );
```

Begriff	Erklärung
Tool / Framework	das zugrundeliegende Test-Programm, welches der Benutzer steuert und welches Benchmark sowie Workload vereint
Engine	der Kern des Tools, welches die Tenants gegen das DBMS ausführt
Benutzer	der eigentliche Benutzer des Benchmark-Tools, spezifiziert den Benchmark sowie den Workload
Nutzer	ein durch den Benchmark-Test repräsentierter Nutzer aus Sicht des Systems, welcher das System „virtuell“ testet
Tenant	eine gekapselte und parametrisierte Einheit aus Benchmark und Workload, *stellt einen oder mehrere Nutzer intern physisch dar*
Nutzerverhalten	das Verhalten eines Nutzers gegenüber dem System, beschrieben durch den Tenant-Workload
Nutzer(verhaltens)modell	ein formal beschreibbares Modell, mit dem sich das *Nutzerverhalten* definiert
Workloadmodell	gleich wie *Nutzermodell*, aber bereits umgesetzt als spezifizierbarer Workload (= *Tenant-Definition*)
Last / Load	auf einem System anliegende Last, verursacht durch einen oder mehrere Nutzer, äußert sich in zB. CPU, RAM, ...
Query	ein einfaches SQL-Statement
Transaktion	ein Vorgang zum Verarbeiten von Queries, inklusive aller nötigen technischen Einheiten, zB. Synchronisation, Locking, ...

Tabelle 2.1: Erklärung der Begrifflichkeiten

Mit der Voraussetzung, dass hier ein entsprechender Benchmark sowohl die Beschaffenheit der Queries als auch die Tabellen und Daten definiert hat, besteht der Workload hier nur aus der Aktion eines Nutzers `USER`, welcher für jede Query `QUERY` aus allen möglichen Queries `QUERIES` genau eine nach der anderen mittels einer Transaktion an das DBMS absetzt: `TRANSACTION( QUERY )`.

Multi-Tenancy Database Benchmarks Diese Art von Benchmarks sind im Kern analog den normalen Benchmarks aufgebaut, auch hier benötigt man neben dem Benchmark zusätzlich einen Workload sowie ein Tool, welches den Test auf das System überträgt. Der große Unterschied hierbei liegt in der Anzahl der einzelnen Nutzer, die das Tool gegenüber dem DBMS darstellen soll. Während es bei klassischen Benchmark-Tools nur ein einzelner ist, wird hier mit mehreren unabhängigen Tenants und Nutzern gleichzeitig hantiert. Darüber hinaus kann ein jeder Tenant mehrere Nutzer beinhalten und repräsentieren. Daraus resultiert, dass das Tool dazu in der Lage sein muss, mehrere Tenants unabhängig voneinander verwalten zu können, sodass kein Nutzer den anderen in seiner Ausführung auf der Benchmark-Test-Seite beeinträchtigt. Im Gegenzug dazu eignet sich ein Multi-Tenancy-Test, um zu testen, inwieweit sich mehrere gleichzeitige Nutzer auf dem System verhalten und wie das System an sich darauf reagiert. Auf der Seite des Systems kann es dadurch durchaus gegenseitige Beeinflussungen der einzelnen Nutzer geben, was sich dann wiederum in der Test-Auswertung widerspiegelt. Solche Multi-Tenancy Benchmarks eigenen sich deswegen auch sehr gut, um das Verhalten des Systems bei Überforderung zu evaluieren, aber auch, um eine Auskunft darüber zu erhalten, wie das zu testende System mit der Verteilung von freien Ressourcen und der Bevorzugung einzelner Nutzer umgeht. Beides sind in der heutigen Technologie sehr wichtige Punkte, weswegen diese Art von Tests dementsprechend

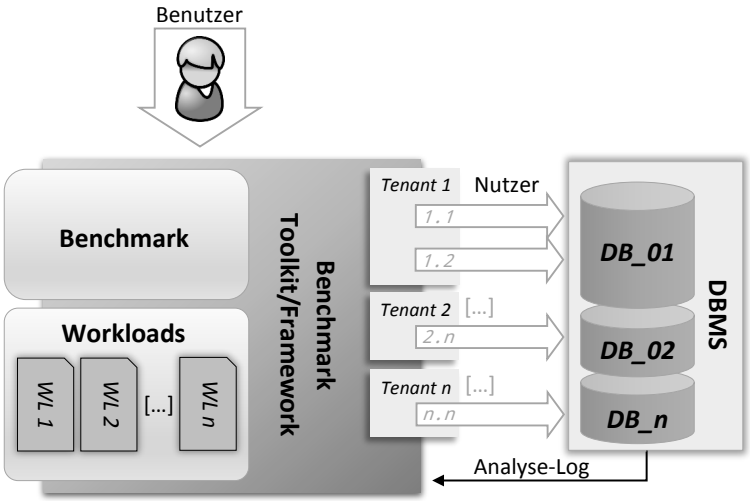


Abbildung 2.1: Zusammenhänge im Multi-Tenancy Benchmark-Umfeld

relevant für kritische und nicht-kritische Systeme ist. Es haben sich im gleichem Zuge auch die DBMS dahingehend weiterentwickelt und neue interne Datenbank-Technologien entwickelt, um sicherzustellen, dass alle Nutzer voneinander getrennt gehandhabt werden. Deswegen besitzen die entsprechenden DBMS meist entsprechende Werkzeuge, um nativ Multi-Tenancy zu unterstützen.

Für das Tool oder Framework bedeutet das Unterstützen von mehreren gleichzeitigen und unabhängigen Tenants, dass hierbei auch mehrere Workloads pro Benchmark zur Anwendung kommen können, was ebenfalls einen enorm höheren Verwaltungsaufwand der einzelnen Tenants mit sich bringt. Eine entsprechend geänderte Darstellung des Benchmark-Umfeldes ist in Abbildung 2.1 gegeben. Wie man anhand der Grafik erkennen kann, benötigt das Tool, um eine geeignete Anzahl verschiedener Tenants zu generieren, auch eine entsprechend hohe Anzahl an Tenant-Definitionen, beziehungsweise deren Workloads (in der Grafik mit „WL“ dargestellt). Die generierten Tenants stellen dann jeweils wieder eigene Nutzer bereit. Dadurch ist es möglich, jedem einzelnen Nutzer ein eigenes Verhalten zu implementieren. Dabei ist immer zu beachten, dass die einzelnen Workloads und Tenants sich auf Tool-Seite nicht gegenseitig beeinflussen, sondern jeweils unabhängige Entitäten darstellen.

2.2 DAS *MULTE*-FRAMEWORK

Das Grundgerüst dieser Arbeit bildet das Multi-Tenancy Benchmark Framework *MulTe* [6], welches von einer Gruppe um Tim Kiefer an der *Technischen Universität Dresden* entwickelt wurde.

Funktionsweise Das *MulTe*-Framework ist ein Benchmark-System, bestehend sowohl aus Python- als auch Java-Komponenten. Man kann *MulTe* hinreichend gut parametrisieren, um ein geeignetes und einfaches Nutzermodell gegenüber einer Datenbank darzustellen. Dabei liegt der Fokus von *MulTe* nicht direkt auf der komplexen Modellierung von Nutzerverhalten, sondern auf der möglichst unkomplizierten Ausführung und Simulation von mehreren unabhängigen Tenants,

welche wiederum jeweils eigene Nutzer und Eigenschaften widerspiegeln können. Das zum Testen benutzte DBMS als Benchmark-Tool-Ziel ist dabei generisch über *JDBC* in das Framework eingebunden, sodass nur minimaler Zusammenhang zwischen dem Framework und dem DBMS besteht und dabei stets einfach um weitere DBMS erweiterbar ist.

Die einzelnen Komponenten Das *MulTe*-Framework besteht aus genau zwei Hauptkomponenten, welche die benötigten Werkzeuge für den Benchmark bereithalten: das Python-Framework und das Java-Framework. Eine Übersicht ist in Abbildung 2.2, entnommen aus der originalen Veröffentlichung von *MulTe* [6], gegeben.

Die erste Hauptkomponente von *MulTe* ist das Python-Framework, welches die eigentliche Vorarbeit der Tests übernimmt. Dieses Framework besteht aus vielen Python-Modulen und nutzt einen eingestellten Benchmark (zumeist *TPC-H*) um die Vorbereitungen zu treffen.
Im ersten Schritt werden die entsprechenden Tenants gebildet. Dazu liegen in *MulTe* vorgefertigte Templates von Workloads bereit, die durch den Benutzer systematisch befüllt werden. Das Python-Framework übernimmt dann die Arbeit, die entsprechend befüllten Workloads auf die einzelnen Tenant-Strukturen zu verteilen, sodass im Endeffekt jeder einzelne Tenant einen anderen Workload benutzen kann. Wie genau ein solcher Workload in *MulTe* definiert ist und welche Möglichkeiten sich dem Benutzer bieten, ist in den folgenden Ausführungen auf Seite 17 beschrieben. Entsprechend der Spezifikation des benutzten Benchmarks werden nun die Datenbanken des DBMS mit Strukturen und Daten befüllt. Dabei wird pro Tenant eine eigene, von den anderen unabhängige Datenbank im DBMS angelegt und befüllt. Die dabei bedingten Constraints und Vorgaben des Benchmarks werden beibehalten und umgesetzt. Als letztes werden noch systembedingte Parameter in eine Struktur gebracht, welche Daten enthalten, die für den Gesamtablauf des Tests nötig sind. Dazu gehören DBMS-spezifische Angaben wie Typ, Nutzername, Passwort und Port, aber auch Test-bedingte Angaben wie Gesamtlaufzeit oder Ausgabeverzeichnis. Damit ist die Vorbereitung des Benchmark-Tests abgeschlossen.

Es folgt die zweite Komponente, das Java-Framework, welche die durch das Python-Framework generierten Tenants mit deren Workloads ausführt. Dabei werden die Tenants als voneinander unabhängige Entitäten betrachtet, welche jeweils selbst ein oder mehrere Nutzer beinhalten. Die komplette Engine für das Ausführen der Tenants mit ihren jeweils eigenen Workloads ist ein eigenständiges Java-Programm (`WorkloadDriver`), eingebettet in den gesamten *MulTe*-Workflow. Dabei werden die in der Tenant-Struktur gesetzten Parameter ausgewertet und final umgesetzt. Bis zu diesem Zeitpunkt kann man noch jeden einzelnen Parameter von Hand ausrichten, alles Folgende, was die Engine passiert und an das DBMS gereicht wird, ist nicht mehr änderbar. Im Kern verwendet die Engine pro Tenant einen eigenen Thread, welcher wiederum einen oder mehrere mehrere Nutzer (ebenfalls als eigenen Thread) spawnt, die dann alle gleichzeitig eine separate Verbindung zu dem DBMS aufnehmen. Zu beachten ist, dass jeder Tenant eine eigene Datenbank im DBMS besitzt und alle Nutzer eines Tenants auf die gleiche Datenbank zugreifen, niemals jedoch zwei Tenants eine gemeinsame Datenbank nutzen. Je nach Konfiguration des Benutzers Evaluiert *MulTe* nun das DBMS in Hinblick auf Multi-Tenancy-Fähigkeiten. Es ist durchaus möglich, die Workload-Engine dem Gesamtworkflow zu entnehmen und separat zu handhaben und damit die Python-Komponente zu überspringen, was für eine Prototyp-Implementierung dieser Arbeit gelegen kam, um den Gesamtaufwand zu minimieren.

Um die durch die zweite Komponente generierten Ergebnisse auszuwerten, wird wieder die ers-

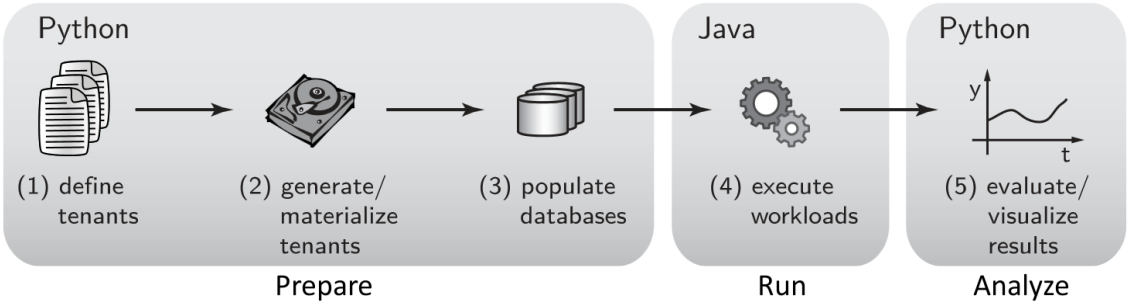


Abbildung 2.2: Workflow von MulTe (aus dem original Paper)

te Komponente, das Python-Framework benutzt, welche einen Interpreter bereit stellt, um die entsprechen angefallenen Daten im CSV-Format auszuwerten und als statistisches Modell wiederzugeben. Dabei wird das Ergebnis nach Tenants, Queries oder beiden gruppiert, sortiert und in ein einheitliches Format gebracht. Man kann diesen Teil ebenso ausklammern, wenn man nicht zwingend eine statistische Analyse der Ergebnisse benötigt oder dritte Programme (wie zum Beispiel *Microsoft Excel* oder *R*) für die Auswertung der Ergebnis-Logs nutzen möchte.

Benutzung und Workflow Das Konzept des *MulTe*-Frameworks ist sehr Workflow-orientiert. Es unterscheidet zwischen den drei Phasen: Prepare, Run und Analyze. Jede der Phasen hat dabei ihre eigene Vorgehensweise und Art der Handhabung, eine Übersicht findet sich ebenfalls in Abbildung 2.2 wieder. Dabei werden die oben beschriebenen Komponenten von *MulTe* in einen Gesamt-Workflow gebracht, an denen der Nutzer an vordefinierten Stellen seine Anpassungen vornehmen kann. Jede Phase kann unabhängig von einer anderen ausgeführt werden. Im Idealfall kann der Benutzer einfach alle notwendigen Informationen zu Beginn eingeben und den Test starten. Alles andere ist durch den Workflow der Daten definiert.

Die Prepare-Phase setzt dabei vollkommen auf die Python-Framework-Komponente und erledigt die gesamte Vorbereitung, wie es oben beschrieben wurde. Anschließend folgt die Run-Phase, welche hauptsächlich die Engine steuert und die eigentliche Test-Arbeit übernimmt. Als letzte folgt die Analyze-Phase, welche die statistische Auswertung der in der Run-Phase erzeugten Ergebnisse aus CSV-Logs auswertet.

Um Eingriffe in den Workload vorzunehmen, wie es die Intention dieser Arbeit ist, muss gleichzeitig der Workflow von *MulTe* erhalten bleiben. Man muss also davon ausgehen, dass der Benutzer wie vorher nur seine initialen Spezifikationen für den Tests eingeben muss, der Rest sollte dann immer noch weitestgehend automatisch durch *MulTe* geschehen.

Ausdrucksmächtigkeit des Workloads Der bisherige Workload lässt einige Spezifikationen für das Verhalten eines Tenants zu. Die wichtigen davon sind in Tabelle 2.2 gelistet. Diese Parameter kann der Benutzer am Anfang seiner Tests in den Templates spezifizieren, welche dann durch die Prepare-Phase auf die einzelnen Tenants verteilt werden. Die Auswirkung der einzelnen Parameter auf das DBMS als theoretische Last-Kurve kann in der Abbildung 2.3 nachvollzogen werden. Der Workload selbst ist nach der Template-Übertragung im Kern ein XML-Dokument mit einer internen Tenant-Struktur. Ein Beispiel für diese Struktur ist in Kapitel 4 gegeben, wo auch näher auf die technischen Details eingegangen wird.

Ebene	Param.	Funktion	Wertebereich
Workload	name	Name des Tenants	$A \ldots Z, a \ldots z, _, \mathbb{N}$
	database	Name der Datenbank des Tenants	$A \ldots Z, a \ldots z, _, \mathbb{N}$
	parallelusers	Anzahl der Nutzer des Tenants, die gleichzeitig mit dem DBMS agieren	$\mathbb{N}^+$
	when_active	Art und Umfang der Beschränkung von Tenants	$seconds, transactions, \mathbb{N}$
	meansleeptime	Zeit in Sekunden zwischen der erneuten Aktivierung des Tenants	$\mathbb{N}$
	thinktime	Zeit in Millisekunden für Pause zwischen einzelnen Transaktionen	$\mathbb{N}$
	transactions	Definition für den Pool an verfügbaren Queries für den Tenant	(eigener XML-Knoten)

Tabelle 2.2: Parameter-Spezifizierung im originalen *MulTe*-Tenant-Definitionen

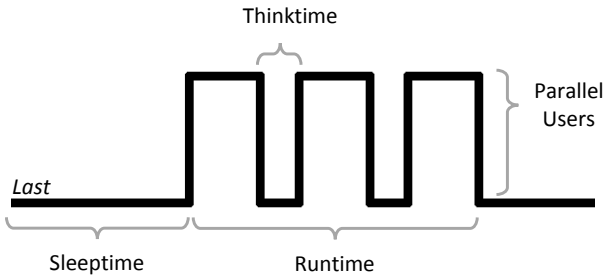


Abbildung 2.3: Darstellbares Nutzerverhalten durch den bisherigen *MulTe*-Workload

Das bisherige Nutzermodell von *MulTe* besteht hauptsächlich aus Pausen- und Denkzeiten für das Ausführen von Transaktionen, die eigentliche Last pro Tenant bestimmt die Anzahl an parallelen Nutzern. Die in Abbildung 2.3 aufgezeigte Kurve *eines* Tenants entstand dadurch, dass der `when_active` Parameter auf drei Transaktionen gesetzt wurde. Währenddessen verbinden sich eine gewissen Anzahl mit dem DBMS und führen ihre Transaktionen aus. Dieses Verhalten wird auch als „An/Aus"-Modell bezeichnet, da hier der modellierte Tenant entweder in Sleep- oder Active-Modus verweilt und wechselt, ohne seine eigentliche Last während der Laufzeit aktiv zu ändern. Dies ist eine valide, wenn auch recht einfache Möglichkeit, ein Nutzerverhalten zu beschreiben. Weitere Informationen zu diesem Modell und dessen Möglichkeiten sowie Vor- und Nachteile finden sich in Kapitel 3.2, wo dieses und weitere Modelle unter dem Gesichtspunkt der Thematik dieser Arbeit analysiert werden. Mit diesem bisherigen Nutzermodell lassen sich gute grundlegende Tenant-Workloads generieren und damit auch das benutzte DBMS analysieren. Jedoch ist dieses Nutzermodell in seinen Möglichkeiten eingeschränkt, einem Workload mittels formalem Modell derart zu konstruieren, dass situationsspezifische Verhaltensmuster (bisher) nicht darstellbar sind. Nichtsdestotrotz ist das bisherige zugrundeliegende „An/Aus"-Workload-Modell der Ausgangspunkt für ein neues, komplexeres Modell eines Workloads und damit einen umfangreicheren synthetisierten Nutzer.

Erweiterung des Nutzermodells Die Kern dieser Arbeit ist die Modellierung und Beschreibung eines neuen Nutzerverhaltens durch komplexe Workloads. Das *MulTe*-Framework bietet hierbei Gelegenheiten, neu entwickelte Ansätze für komplexere Nutzerverhaltensmodelle einzubauen. Einerseits ist es durch den modularen Aufbau vom *MulTe* relativ einfach, Erweiterungen oder Änderungen vorzunehmen. Für viele Aspekte gibt es einen lokalisierten und speziellen Einsprungpunkt, um seine Erweiterung einzubauen. Andererseits unterstützt auch die Workflow-Orientierung des Frameworks spezielle Anpassungen in der Kette der Ausführungen. Ausführliche Details dazu sind in [7] zu finden. Als Beispiel sei die Entwicklung eines alternativen Nutzermodells in *MulTe* gewählt. Ein Ansatz wäre hier, sich das Bauen der Tenants durch die Templates vorzunehmen, sodass neue oder geänderte Parameter als Input vom Benutzer des Benchmarks genutzt werden können und anschließend die Umsetzung dieser Parameter durch die Workload-Engine vorgenommen wird. Zusammengefasst wäre hier die neue Kern-Logik einzig und allein in der Java-Framework-Komponente, also durch Erweiterung der Workload-Engine zu realisieren.

2.3 ÄHNLICHE UND WEITERFÜHRENDE ARBEITEN

Viele bereits existierende Arbeiten befassen sich direkt oder indirekt mit Nutzerverhalten von Benchmarks, allerdings existiert nach bestem Wissen des Autors derzeit keine äquivalente Arbeit zu dem Thema dieser Arbeit.

Das *MulTe*-Framework Angefangen wird hier mit der Grundlage dieser Arbeit [6], welches das Fundament (das *MulTe*-Framework) stellt und ein eigenes Workload-Modell bereitstellt, was bereits in den vorherigen Kapiteln ausführlicher beschrieben wurde. Dieses deckt bereits viele Parameter für einen synthetisierten Nutzer ab, allerdings nicht umfangreich genug um ein wirklich realitätsnahen Nutzer und dessen Workload zu schaffen. Da in diesem Framework allerdings bereits Grundzüge für ein komplexeres Nutzermodell vorhanden sind, wurde es als Grundlage für das weitere Vorgehen für diese Arbeit ausgewählt. Da *MulTe* bereits vorgestellt wurde, wird hier nicht weiter darauf eingegangen.

Die Modellierung von Spikes Weiterhin befasst sich die Arbeit [2] mit der Modellierung von Spikes in das bestehende Benchmark-Systeme.
Es wird in dort zwischen Daten- und Volumen-Spikes unterschieden. Daten-Spikes sind solche, welche ein hohes Aufkommen von Anfragen auf bestimmte Daten konzentriert darstellen. Diese werden in der Arbeit „Hot Objects“ genannt, welche als Grundlage für den Spike für einen Moment vermehrt Aufgerufen werden. Als Volumen-Spikes sind solche definiert, wenn ein kurzes und starkes Aufkommen von Anfragen generell auftritt. Zwischen den beiden Spike-Arten ist der konkrete Unterschied, dass sich bei einem Daten-Spike die eigentliche Anzahl der Nutzeranfragen nicht verändert, sondern sich nur signifikant auf bestimmte Daten, den „Hot Objects“, verlagert. Im Gegensatz dazu wird bei einem Volume-Spike die echte Anzahl an Nutzeranfragen während des Spikes erhöht.

Für die Modellierung eines Spikes werden in der Arbeit sieben verschiedene Parameter eingeführt, welche den Umfang und die Machart eines Spikes sehr genau beschreiben können. Dazu

zählen Parameter wie *Number of Hot Objects*, *Entropy*, *Spatial Locality*, *Duration* oder auch *Relative Increase*, mit denen man sehr detailgetreu einen Spike beschreiben kann. Durch diese exakte Beschreibung und Benutzung eines Spikes durch diese sieben Parameter werden realistischere Spikes von Nutzern simuliert, die auf ein bestimmtes Event reagieren und damit im System eine kurzzeitig hohe Auslastung erzeugen. Es wird dadurch ebenfalls ein Workload-Modell konstruiert, jedoch nur in Hinblick auf Modellierung von Spikes. Während die Autoren der Arbeit dieses Ziel gut umsetzen können, geht ihre Arbeit leider nicht über die Modellierung von Spikes hinaus. Es fehlen dem Workload dabei weitere Mechanismen, um auch abseits der Spikes diverse Muster darstellen zu können. Dennoch legt die Arbeit [2] über die Modellierung von Spikes eine der beiden wichtigen Grundlagen und Ideen für diese Arbeit.

Die Modellierung von Burstiness in *TPC-W* Eine weitere Arbeit [9] befasst sich mit der direkten Injektion von realistischer „Burstiness“ in ein Workload gegenüber dem System. Dafür stellt die Arbeit eine Methode für die Generierung von Workloads vor.
Es wird anhand von *TPC-W*, einem weit verbreiteten Benchmark neben *TPC-H*, aufgezeigt, inwieweit man durch die Injektion von *Thinktime*, also Zeiten zwischen einzelnen Transaktionen, die Art der *Burstiness* steuern kann. Dafür wird ein Markov-Model als Übergangsmodell geschaffen, in welchem diverse Wahrscheinlichkeiten definiert sind, um in den jeweils anderen State überzugehen. Diese States werden als *Normal Traffic* mit langer *Thinktime*, und als *Traffic Surge* mit einer sehr kurzen *Thinktime* bezeichnet.

Zur Steuerung der Übergänge wird dafür ein Parameter namens *Index of Dispersion* benutzt [4], welcher in einer kontrollierbaren Art die Injektion von Bursts in den Datenstream des *TPC-W* Benchmarks handhabt, indem die modellierte *Thinktime* zwischen einzelne Transaktionen eingefügt wird, bevor sie das DBMS erreichen. Auch wenn diese Arbeit den Zweck eines komplexeren Nutzermodells erfüllt, ist das Ziel, einen formal beschriebenen Nutzer zu erzeugen, nicht komplett verwirklicht. Zum Teil fehlen hier noch die Ansätze, die „Burstiness“ genauer zu definieren, beziehungsweise so zu beschreiben, dass man mehr Formen von Nutzermodellen erzeugen kann. Dabei reicht eine einfache Modellierung von *Thinktime* nicht aus, auch wenn der Ansatz, das Transaktionsaufkommen mittels Denkpausen zu steuern, funktioniert. Zudem sei hier kurz festgestellt, dass die Modellierung des Nutzers durch den *Index of Dispersion* nicht wirklich ein Nutzermodell ist, da hierbei die eigentliche Modellierung erst nach dem Benchmark durch Injektion in den Datenstream geschieht, nicht jedoch bei der Definition des eigentlichen Tenants im Workload.

Das Timeseries-Modell Die Arbeit [5] hat zwar nicht direkt mit Nutzermodellen von Benchmarks zu tun, allerdings bildet das darin vorgestellte Verfahren eines der beiden Grundlagen für das in dieser Arbeit entwickelte Modell eines Workloads.
Die Arbeit befasst sich eigentlich mit dem Beschreiben von Kurven im „Forecasting“ mittels dem *Holt-Winters Exponential Smoothing* („exponentielle Glättung“) Verfahren, beziehungsweise dem *Timeseries*-Modell. Das Modell besitzt genau drei Stufen: die einfache, die doppelte sowie die dreifache exponentielle Glättung. In der einfachen Glättung wird angenommen, dass Werte einer Kurve alle um einen Mittelwert herum liegen, weswegen nur ein *Base* festgelegt wird, welcher den Mittelwert der Kurve entspricht. In der doppelten Glättung wird eine weitere Annahme getroffen, in welcher der Verlauf der Werte einem Trend entspricht, dementsprechend auch ein *Trend* in das Kurven-Modell hinzugefügt wird, was sich in einem Anstieg der Kurve äußert. Die höchs-

te Form der Glättung enthält neben *Base* und *Trend* auch ein *Season*-Modell. Die *Season* ist eine bestimmte vordefinierte Folge von Werten, welche entweder additiv oder multiplikativ auf die Kurve hinzugerechnet werden und somit saisonale Unterschiede dargestellt werden können.

Die Arbeit beschreibt weiterhin die Möglichkeiten, mit den drei Parametern eine wohldefinierte Kurve für zeitliche Abläufe zu bilden und damit Forecasting-Analyse zu betreiben, was jedoch für diese Arbeit nicht weiter relevant ist. Dieses Verfahren, ursprünglich in [5] für *Forecasting* im „Big Data"-Bereich gedacht, wird in dieser Arbeit auf das Erstellen von Nutzerverhalten übertragen.

Weitere thematisch ähnliche Arbeiten Am Rande sind noch die folgenden Arbeiten relevant, allerdings nicht direkt zu der Thematik dieser Arbeit. Dennoch gaben sie alle den notwendigen Input für Entwicklungen von Konzepten dieser Arbeit.

[3] Die Arbeit beschäftigt sich mit dem Management mehrerer Datenbanken eines Servers, je nach realem Workload, welcher gerade anliegt, und die zugehörige sinnvolle Verteilung von Ressourcen des Servers, um möglichste effizient zu arbeiten. Weiterhin versucht man, die analysierten Workloads dazu zu nutzen, um Vorhersagen im Bereich Anfrageoptimierung zu treffen sowie Load Balancing ohne Performance-Einbußen zu bewerkstelligen.

[8] In dieser Arbeit geht man zwar nicht direkt auf die Problematik von Benchmark und Workloads ein, erhält dafür aber einen tieferen Blick in die Untersuchung von Anomalien von Anfragen auf Datenbanken.

[10] Hier betrachtet und analysiert man das dynamische Verhalten von Erzeugung und Modifikation von Web-Inhalten auf Servern sowie den Zugang durch Clients.

3 FORMALES WORKLOAD-MODELL EINES NUTZERS

3.1 DAS KONZEPT DER INTENSITÄT

Der Begriff „komplex“ ist naturgemäß recht unscharf gefasst und nicht für sich selbst stehend. Ein komplexerer Zustand ist immer nur relativ zu sehen und benötigt in jedem Kontext eine andere Definition. Die Grundlage dieser Arbeit ist ein einfaches Nutzermodell, eingebettet in das Benchmark-Framework *MulTe* (siehe Kapitel 2). Dadurch ist dies der Bezugspunkt in Relation zu „komplex“ zu verstehen. Legt man dieses grundlegende Modell eines Workloads zugrunde, definiert sich ein komplexeres Modell eines Workloads durch folgende Punkte:

1. das komplexe Modell sollte generell mehr Möglichkeiten zur Steuerung besitzen,
2. das komplexe Modell sollte den anfänglichen Mehraufwand beim Initialisieren als geeigneteres Resultat am Ende auch widerspiegeln (ein gutes Kosten-Nutzen-Verhältnis ist angebracht),
3. das komplexe Modell sollte in das bisherigen *MulTe*-Framework eingebunden werden und mit allen anderen Komponenten kompatibel bleiben (mehr Komplexität im Sinne einer Erweiterung, nicht einer Veränderung oder Neuentwicklung).

Das bisherige Workload-Modell von *MulTe* geht von dem eingangs erwähnten „An/Aus“-Modell aus, gepaart mit mehreren Parametern zum Steuern der Zeiten zwischen einzelnen Queries und großer Pausen. Ein interessanter Parameter ergibt sich, da *MulTe* ein Multi-Tenancy-Framework ist. Der Parameter *Parallel Users*, also eine Anzahl von parallelen Nutzern, wird ebenfalls pro einzelnen Tenant ausgewertet. Das wiederum bedeutet, dass ein Tenant in *MulTe* mehrere Nutzer haben kann, die alle den Workload von dem einen Tenant ausführen. Wie viele dieser „kleinen“

Nutzer dabei genutzt werden, liegt in der Hand dieses Parameters. Anders als in Single-Tenant Benchmark-Systemen wird bei *MulTe* also nicht ein Workload pro Nutzer angelegt, sondern pro Tenant, wo dementsprechend der Tenant einen übergeordneten Nutzer darstellt, eine Gruppe von Nutzern, zusammengefasst in einem Workload-Modell. Nichtsdestotrotz ist dieser Parameter statisch eingebunden und kann damit weder zur Laufzeit geändert werden noch besitzt das aktuelle *MulTe*-Benchmark-System eine andere Interpretation für diesen Wert. Nach derzeitigem Stand dieser Arbeit wird der Parameter im Auslieferungszustand von *MulTe* so interpretiert, dass genauso viele Java-Threads gestartet werden, wie der Parameter an Nutzern angibt, womit dann zeitgleich mehrere Threads (in Äquivalenz zu den Nutzern) einen Tenant-Workload ausführen.

Zwei weitere Begriffe, welche im Zusammenhang mit dieser Arbeit eine wesentliche Rolle spielen und für formale Definitionen benötigt werden, sind „Last“ und „Intensität“.

Last ist der Wert in einer fiktiven oder beliebigen Einheit, der nach dem Benchmark an dem DBMS anliegt. Ein DBMS hat logischerweise meist unter höherer Last mehr interne Probleme zu bewältigen als unter niedriger Last. Dabei spielt die Last hier eine analytische Rolle in Relation zu dem DBMS. Man kann die Last zwar durch gezielten Input variieren, jedoch gilt die Last meist nur als Messfunktion gegenüber dem DBMS. Man kann, wenn man dies aus allgemeinwissenschaftlicher Sicht betrachtet, die Last als Input erster Stufe bezeichnen, da sie im direkten Zusammenhang mit der Schnittstelle zwischen Nutzer und DBMS besteht. Das Workload-Modell, in *MulTe* das Tenant-Workload-Modell, erzeugt durch konfigurierbare Parameter den benötigten Input und sorgt damit für die Variation der Last-Kurve am DBMS. In *MulTe* wäre einer der Parameter für den Input erster Stufe zum Beispiel wieder *Parallel Users*. Man kann die Last als Input erster Stufe auch direkt auf bestimmte Werte der Hardware festlegen, beispielsweise der CPU-Auslastung oder der Speicherverbrauch des Hauptspeichers. Oder man nutzt zusammengesetzte und berechnete Werte als Einheit für die Last-Kurve des DBMS, zum Beispiel wie in *DB2* von *IBM* mit den „Timerons“, welches eine Aussage über die Last trifft, formal berechnet aus allen möglichen Werten von *DB2* und darunterliegender Hardware.

Der Begriff der Intensität, stellt einen abstrakteren Zustand dar. Wenn man die Bezeichnung von oben übernimmt, wäre die Intensität der Input zweiter Stufe, sozusagen der abstrahierte Teil der Last. Dabei ist die Intensität nicht unbedingt auf die Schnittstelle zwischen Benchmark-Tool und DBMS festgelegt, sondern vielmehr auf die Relation von dem synthetisierten Nutzer und dem DBMS. Der synthetisierte Nutzer besitzt demnach keine Last, er erzeugt allerdings welche, je nach seiner Intensität und Intention, mit der er mit dem DBMS interagiert. Da die Intention eines Nutzers nicht darstellbar ist, bleibt nur die Intensität übrig. Je höher die Intensität des Nutzers ist, desto mehr Last erzeugt er theoretisch am DBMS. Allerdings ist dabei die Last von der Intensität getrennt und jeder einzelne Nutzer kann eigene Intensitäten besitzen, welche, durch geeignete algorithmische Umrechnungen, dann kombiniert die entsprechende Lastkurve ergeben. Dabei verhält sich Intensität direkt proportional zur Last. Es ist nötig, die Intensität von der Last zu unterscheiden, da man einen entsprechenden Wert als Nutzerdefinition benötigt, wenn man ein komplexes Nutzermodell aufstellen möchte. In einem „An/Aus“-Modell, wie original in *MulTe* implementiert, gibt es nur 100% Intensität (An) sowie 0% Intensität (Aus) des Nutzers. Die Last am DBMS kann sich dennoch nicht-linear ändern (verursacht durch zum Beispiel einem Caching des DBMS), auch wenn der entsprechende Nutzer gerade durchgängig 100% an Intensität anliegen hat. Möchte man nun ein komplexeres Nutzermodell schaffen, benötigt man zwangsweise Abstufungen der Intensität zwischen dem An und Aus.

Anders als Last kann man die Intensität nicht direkt an dem DBSM ablesen oder berechnen und auch nicht sonderlich gut darstellen, denn ein synthetisierter Nutzer hat keine CPU-Auslastung und keinen Speicher. Eine Möglichkeit die Intensität vom Nutzer darzustellen, wie sie auch in dieser Arbeit gewählt wurde, ist die Einführung einer neuen entsprechend definierten Einheit, der Einfachheit halber gleich *Intensity* genannt. Sie beschreibt die Interaktions-Intensität des Nutzers in relativer Beziehung. Dabei ist $Intensity = 0$ gleichzusetzen mit „es erfolgt keinerlei Interaktion des Nutzers mit dem System“, vergleichbar mit 0% aus dem „An/Aus-Modell“. Je nach System muss noch definiert werden, was $Intensity = 1$ bedeutet, vergleichbar mit 100% aus dem „An/Aus-Modell“. Sobald dies spezifiziert wurde, kann ein Modell auf Basis der Intensität festgelegt werden. Dabei kann $Intensity = 1$ eine absolute Anzahl an Aufgaben sein, eine zeitlich begrenzte Ausführung von Aufgaben oder beides gleichzeitig. Mit der Definition der beiden Werte $Intensity = 0$ und $Intensity = 1$ kann man nun aufbauend ein Modell definieren, welches verschiedene Nutzer-Intensitäten wiedergibt. Die Intensität muss anschließend noch in eine konkrete, für das Benchmark-System interpretierbare Größe übersetzt werden, das heißt, die Input zweiter Stufe muss in den Input erster Stufe übersetzt werden. Bei dieser Arbeit wurde als Beispiel die Intensität (als Input zweiter Stufe) per Algorithmus in die Anzahl an parallelen Nutzer-Threads sowie in die Pausenzeiten zwischen Transaktionen in Last übersetzt (als Input erster Stufe), was allerdings stark *MulTe*-abhängig ist. Weitere Ausführungen dazu folgen in Kapitel 4.

3.2 EVALUIERUNG VERSCHIEDENER MODELLE

Bei der Auswahl des Modells ging es um die Frage, welches davon geeignet wäre, die Intensität des Nutzers geeignet beschreiben zu können. Dabei ist in erster Linie die Mächtigkeit relevant, welche ein Modell bereitstellt, bestimmte Modell-Muster oder Kurven zu erzeugen. Im Folgenden werden einzelne Modelle aufgezeigt sowie deren Vor- und Nachteile benannt.

Das „An/Aus“-Modell Das bereits erwähnte „An/Aus“-Modell ist relativ simpel und benötigt nur wenige Parameter. Möchte man dieses Modell per Intensität beschreiben, gibt es nur zwei Zustände: $Intensity = 0$ und $Intensity = 1$. Diese sind fest eingearbeitet und der Benchmark-Benutzer definiert nur noch Übergänge. Das können Pausenzeiten sein, aber auch Gruppierung und eine Art Blockabfertigung der Aufgaben. Dadurch entstehen Intensitäts-Kurven, welche einzeln getrennte Plateaus bilden (siehe Abbildung 3.1). Die konkreten Parameter sind zur Laufzeit des Benchmarks komplett statisch und unveränderbar. Bei diesem Modell fehlen allerdings sämtliche weiteren Möglichkeiten der Modellierung der Kurve, wie zum Beispiel Anstiege, Spitzen, Kurven, Zyklen und anderes. Dieses Modell ist der derzeitige Standard in *MulTe*, wenn man den aktuellen Build herunterlädt und benutzt, wie es bereits auf Seite 17 aufgeschlüsselt worden ist. Dieses Modell ist aber gerade wegen seiner Einfachheit in der Entwicklung und Benutzung auch bei anderen Tools und Frameworks sehr beliebt.

Das „Chaotic Random“-Modell Ein weiteres mögliches Modell ist das „Chaotic Random“-Modell. Dieses besitzt fast keine Parameter, ist daher also besonders einfach zu pflegen. Dabei werden komplette (eigentlich vorhandene, aber nicht direkt für den Benutzer verfügbare) Parameter mit zufälligen Werten befüllt und an den Benchmark durchgereicht, welche sich dann auch

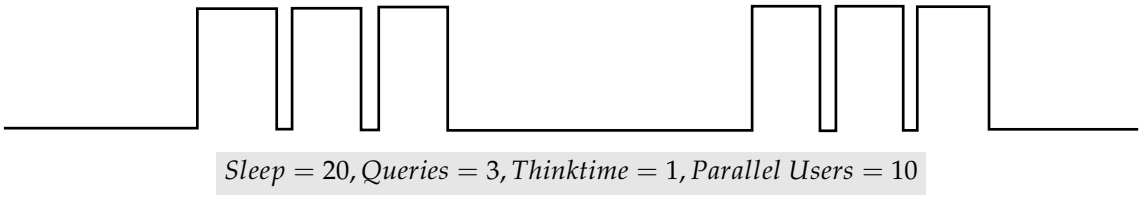


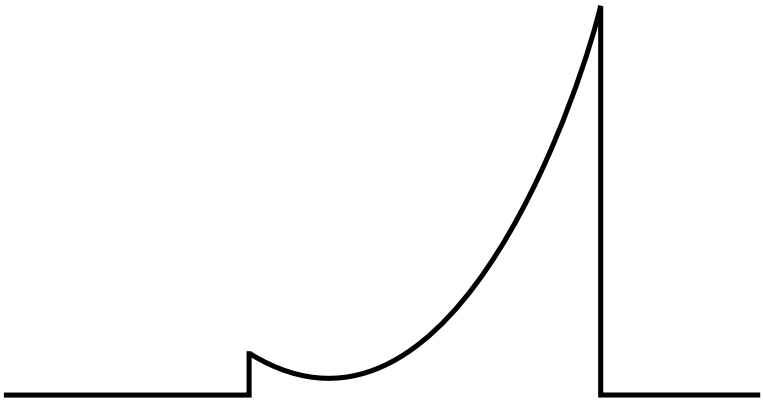
Abbildung 3.1: Beispiel-Kurve für ein simples „An/Aus"-Modell

zufällig zur Laufzeit ändern, was den großen Vorteil darstellt. Im Gegensatz zu dem „An/Aus"-Modell muss man hier nicht einmal die Pausenzeiten oder ähnliches definieren. Das ergibt als Kurve dann eine deutliche Steigerung der Möglichkeiten von Ausprägungen als beim „An/Aus"-Modell. Im Umkehrschluss bietet dieses Modell daher aber keinerlei Kontrolle durch den Benutzer und ist damit fast gänzlich ungeeignet für ein komplexeres Nutzermodell.

Das „Class Random"-Modell Das „Class Random"-Modell basiert auf dem „Chaotic Random"-Modell und versucht, die Nachteile der kompletten Zufälligkeit zu beseitigen. Dazu benutzt dieses Modell sogenannte Klassen von Intensitäten, welche durch den Benutzer steuerbar sind. Damit wird festgelegt, inwieweit der Spielraum für die Zufälligkeit der Parameter zur Laufzeit sein kann. Dabei sind Werte für *Intensity* von 0 und 1 möglich, aber auch höhere ganzzahlige Werte als Klassen-Intensität. Die Intensität wird hier in die Zufälligkeit der einzelnen Parameter umgerechnet. Am Ende bleibt aber auch hier keine direkte Möglichkeit, komplexe Kurven-Muster zu erzeugen, höchstens Plateaus (wie in dem „An/Aus"-Modell) mit unterschiedlicher zufälliger, aber begrenzter Höhe.

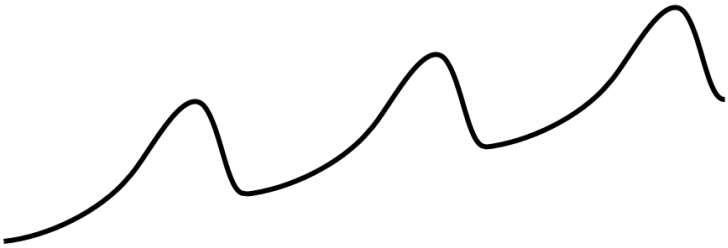
Das „Spike"-Modell Die Arbeit [2] inspirierte zu einer Möglichkeit eines „Spike"-Modells (ein Modell zur Modellierung von Spitzen in der Kurve). Man nutzt hier verschiedene Parameter, um einen Spike in der Kurve zu modellieren. Kombiniert man diese mit dem „An/Aus"-Modell, hat man bereits eine gute Möglichkeit, mit einigen Zusatzparametern mehr Muster in der Intensitäts-Kurve zu beschreiben. Nachteilig ist, dass dieses Modell zu kurz gefasst ist. Zwar werden die Spikes überaus gut beschrieben, jedoch alles andere außen vor gelassen. Zudem ist die Workload-Beschreibung der Spikes, wie sie in [2] vorgestellt wurde, nicht intuitiv genug, um den durchschnittlichen Benutzer damit ein gutes Werkzeug in die Hand zu geben. Es eignet sich für die professionelle Spike-Untersuchung, jedoch nicht konsequent für den Einsatz in einer einfach gehaltenen Tenant-Definition.

Das „Function"-Modell Man kann ebenfalls ein „Function"-Modell nutzen, welches aus der Mathematik bekannt ist. Dazu wird, um entsprechende Muster auf der Intensitäts-Kurve zu erzeugen, eine Kurve oder ein Graph per Funktion (auch höheren Grades) beschrieben und dann direkt die Werte $y = f(x)$ als Intensität abgetragen. Dies gibt einen immensen Vorteil: man kann fast alle Funktions-Kurven als Intensitäts-Kurve darstellen und der Benutzer benötigt als Eingabe nur eine mathematische Funktion. Es wäre eine der idealen Lösungen für die Modell-Wahl, denn man könnte nicht nur Anstiege modellieren, sondern auch Zyklen, Sinus-Kurven (als Tag-Nacht-Simulation der Kurve) oder auch systematisch unterschiedliche Ausreißer. Ein grafisches Beispiel ist in Abbildung 3.2 gegeben, mitsamt dazugehöriger Funktion. Der große Nachteil ist hier der zu hohe Aufwand, den der Benutzer bewältigen muss. Man geht davon aus, dass der Benutzer



$Intensity = f(x) = 0.5x^3 + 2x^2 - 1x + 4$ im Intervall $[0, 3]$

Abbildung 3.2: Beispiel-Kurve für ein „Function“-Modell



$Base = 1, Trend = 0.2, Season = [0, 0, 1, 3, 4, 5]$ (approximiert)

Abbildung 3.3: Beispiel-Kurve für ein „Timeseries“-Modell

seine Ziel-Kurve ungefähr von Aussehen her kennt. Er müsste dann eine entsprechende Funktion finden, die seine Ziel-Kurve hinreichend approximiert, was bei komplexen Kurven einen erheblichen Aufwand darstellen kann. Der zweite große Nachteil wäre hier, dass man Probleme bei Kurven bekommt, die sich nicht per einzelner Funktion darstellen lassen, da sie zu große Ungereimtheiten enthalten, wie zum Beispiel ein per Funktion nicht systematisch darstellbares „Loch“ mit $Intensity = 0$ in der Kurve.

Das „Timeseries“-Modell Das letzte große Modell, was hier vorgestellt wird, ist das aus [5] bekannte „Timeseries“-Modell. Inspiriert durch die Arbeit [5] kann man diese Technik auch auf die Intensitäts-Kurve übertragen. Hauptsächlich basiert das „Timeseries“-Modell darauf, eine Kurve mithilfe von drei Parametern zu beschreiben, recht ähnlich einer Funktion, allerdings darauf spezialisiert, unregelmäßige Muster in der Kurve darstellen zu können. Wie in der Arbeit beschrieben, nutzt man dieses Modell, um bisherige Kurven-Verläufe zu analysieren, um damit später *Forecasting* betreiben zu können. Die drei benötigten Parameter teilen sich wie folgt ein:

1. die *Base* ist der Wert, welcher allen anderen zugrunde liegt und auf welchen alle anderen Parameter relative Bezug nehmen,

2. der *Trend* ist der Wert, welcher den Anstieg beschreibt, den die Kurve im Laufe der Zeit vornimmt,

3. die *Season* ist ein Muster, welches sich ständig wiederholt und kontinuierlich auf den Kurvenverlauf hinzu addiert oder multipliziert wird.

Mit diesen drei Parametern ist es möglich, eine simple $f(x) = mx + n$ Kurve darzustellen, wobei das m der Trend wäre, und n die Base. Hier liegt auch der große Unterschied zu dem „Function"-Modell, welches deutlich mehr Funktionen als diese einfache Funktion darstellen kann. Zusätzlich zu dieser Funktion definiert die Season das Muster, welches sich wiederholend auf die Kurve legt, was wiederum nicht mehr geeignet per Funktion darstellbar wäre, dafür aber viele praktische Vorteile besitzt. Mit der Season könnte man beispielsweise (wie der Name bereits aussagt) ein Muster generieren, welches äquivalent für die Unterschiede in den Quartalen eines Jahres steht; dann wäre ein mögliches Muster der Season die Folge $2, 1, 1, 4$ wobei die 2 den gesteigerten Umsatz zu Ostern und die 4 den gesteigerten Umsatz zu Weihnachten darstellt. Theoretisch kann man mit einer Season mit unendlicher Folge eine komplette Kurve beliebig weit exakt vordefinieren. Praktisch würde man sich ein Muster aussuchen, welches sich in der Kurve des Öfteren wiederholt und dieses einfach wiederholt anwenden. Im Zusammenspiel mit den anderen beiden Parametern Base und Trend kann man also sehr gut eine Kurve im Sinne der Intensität darstellen, da der Benutzer direkt und ohne große mathematische Vorkenntnisse seine Intensitäts-Kurve damit beschreiben kann. Ein Beispiel für ein „Timeseries"-Modell mit den Parametern Base, Trend und Season findet sich in Abbildung 3.3. Zu beachten ist, dass die dazugehörigen Werte für Base, Trend und Season nicht exakt diese Kurve widerspiegeln sondern in diesem Falle nur als Anhaltspunkt gedacht sind.

Es ist recht eindeutig, dass es noch viele weitere Modell-Kandidaten gibt, womit man eine Intensitäts-Kurve geeignet darstellen kann. Darauf jedoch lag nicht der Fokus dieser Arbeit. Die bisher gezeigten Modelle geben einen vorerst guten Überblick, wie man an das Thema der Modellierung herankommt und welche Entscheidungskriterien bei der endgültigen Wahl berücksichtigt wurden.

3.3 KONSTRUKTION DES NEUEN *MULTEX*-MODELLS

Letzten Endes wurde in dieser Arbeit eine Kombination aus zwei Modellen generiert. Der Einfachheit halber wurde es mit dem Namen „MulTeX"-Modell benannt, um das entstandene Modell weiterhin in dieser Arbeit bei einem Namen nennen zu können, wobei das „X" nur den Zustand einer Erweiterung darstellen soll.

Dem Modell liegt das umfangreiche „Timeseries"-Modell zugrunde, welches, wie weiter oben beschrieben, sich gut für den Zweck der Intensitäts-Modellierung eignet. Im Detail wurde die Methode der dreifachen Glättung mit additiver Season gewählt, um möglichst viel Möglichkeiten zu haben. Ein Aspekt, den das „Timeseries"-Modell in Hinblick auf Benchmarking nicht abdecken kann, ist der Einfluss von zufälligen Ereignissen, welche das Verhalten der Nutzer drastisch ändern können. Um solche Ereignisse dennoch abbilden zu können, wurde zusätzlich zu dem „Timeseries"-Modell das „Spike"-Modell angewendet. Diese zusätzliche Komponente von Spikes macht es dem Benutzer möglich, zufällige Ereignisse in das Modell zu streuen, welche einen mehr oder weniger starken Einfluss auf die Intensitäts-Kurve besitzen. Jedoch ist der in [2] gewählte Ansatz nicht ideal, da es zu viele Parameter für den Benutzer bereit hält, die am Ende jedoch nicht wirklich aussagekräftig für ihn sind. Deswegen wurde dieses „Spike"-Modell angepasst und mit

dem „Timeseries“-Modell kombiniert, um eine gute Mischung zwischen Ausdrucksmächtigkeit und Benutzbarkeit zu bekommen, gleichzeitig aber vordefinierte Kurven zusammen mit zufälligen Ereignissen darstellen zu können. Das dadurch erstellte „MulTeX“-Workload-Modell dient dem Zweck, einen möglichst dynamischen synthetisierten Nutzer darzustellen. Die Rolle der Intensität des Nutzers wurde bereits weiter oben behandelt, weswegen hier nicht weiter darauf eingegangen wird. Der Nutzer besitzt als Basis das „Timeseries“-Modell, welches dafür sorgt, eine Grundlage des Intensitäts-Verhaltens zu schaffen. Gleichzeitig wird dabei das „Spike“-Modell additiv hinzugefügt, welches anormales Verhalten zu (pseudo-)zufälligen Zeiten modelliert. Durch diese Kombination ist es möglich, fast jede beliebige Struktur zu erschaffen.

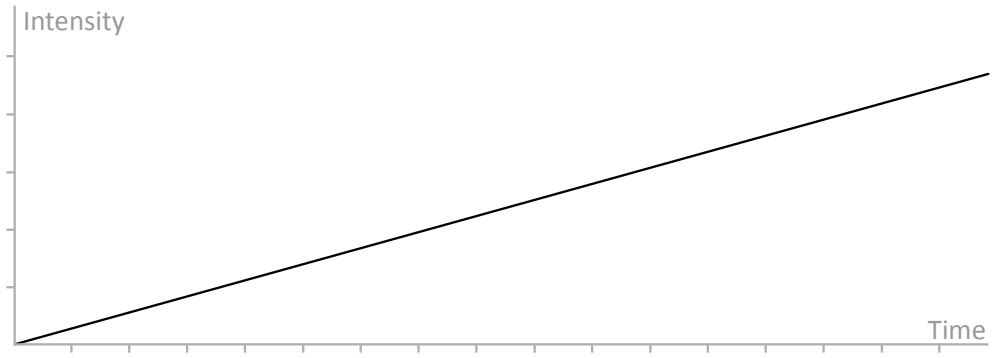
Das entstandene Modell wurde einer Vereinfachung unterzogen, um es für die Allgemeinheit zugänglicher werden zu lassen. Es verbleiben nur noch kombiniert sechs Parameter, welche der Benutzer spezifizieren muss. Die ersten drei sind für die Darstellung von „Timeseries“ gedacht und gleichen der Beschreibung von weiter oben: 1) die *Base*, 2) der *Trend*, und 3) die *Season*. Die anderen drei Parameter gelten für das vereinfachte „Spike“-Modell und betreffen folgende Eigenschaften: 1) *Spike Random* als Aussage über die Auftrittswahrscheinlichkeit eines Spikes während des Verlaufs der Kurve, 2) *Spike Markedness* als Parameter für die absolute Ausprägung (oder „Stärke“) des Spikes, und 3) *Spike Markedness Random* als zufällige Komponente, relativ zu der totalen Ausprägung (äquivalent zu: $Spike\ Markedness \pm Spike\ Markedness\ Random$).

Mit diesen neuen „MulTeX“-Modell ist es möglich, zu jedem Zeitpunkt $t \in \mathbb{N}$ eine sinnvolle Berechnung der Intensität durchzuführen:

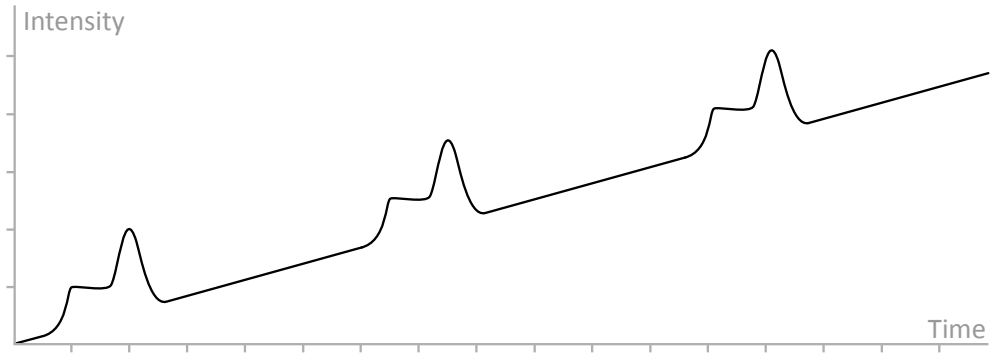
$$Intensity(t) = Base + (Trend * t) + Season[t] + (SR[t] * (SM \pm SMR))$$

SR	Spike Random
SM	Spike Markedness
SMR	Spike Markedness Random

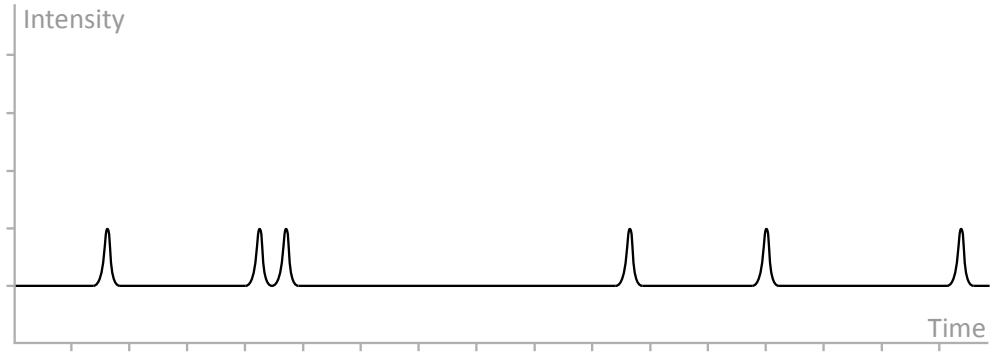
wobei $Season[t]$ die entsprechende Stelle im modularen Season-Muster-Ring liefert und SR (beziehungsweise $Spike\ Random$) eine Funktion ist, welche entweder 1 oder 0 zurück liefert, je nach Ausgang der Zufälligkeit (unter Einbeziehung des t als definierter Seed für Pseudo-Zufälligkeit). Als grafisches Beispiel dient Abbildung 3.4 zum Veranschaulichen der Wirkungsweise der einzelnen Parameter.



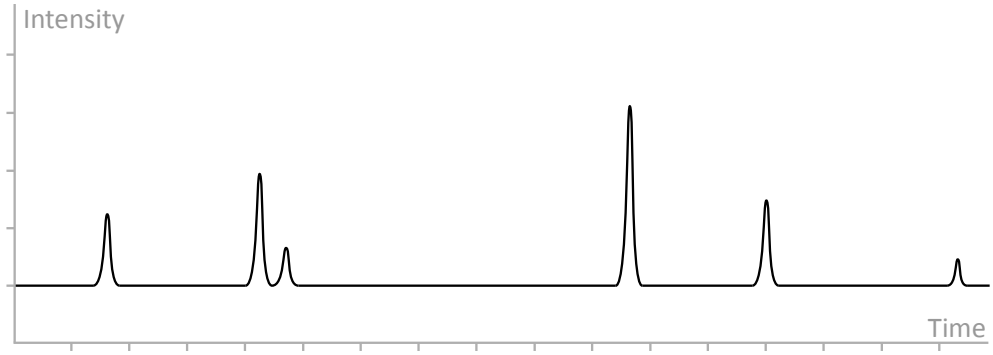
(a) Base + Trend



(b) Base + Trend + Season



(c) Base + Spike Random + Spike Markedness



(d) Base + Spike Random + Spike Markedness + Spike Markedness Random

Abbildung 3.4: Beispiel-Kurven für ausgewählte Parameter im „MulTeX“-Modell

4 TECHNISCHE UMSETZUNG IN *MULTE*

In dem folgenden Kapitel wird die Entwicklung des in Kapitel 3 eingeführten Modells als Prototyp realisiert. Die Umsetzung und spätere Implementierung als Erweiterung in *MulTe* dienen in diesem Fall nicht als Schwerpunkt dieser Arbeit, sondern sollen eher der Veranschaulichung und Machbarkeit der Entwicklungen aus Kapitel 3 aufweisen, gleichzeitig das *MulTe*-Framework dahingehend erweitern.

4.1 KONZEPTIONELLE UMSETZUNG

Einbindung in den Workflow

Der Workflow von *MulTe* ist sehr stark in einzelne Komponenten unterteilt. Wie bereits in Kapitel 2 erwähnt, basiert der Aufbau von *MulTe* auf den drei Phasen Prepare, Run und Analyze. Um das erarbeitete Modell geeignet in das Framework einzubauen, muss man in die beiden Schritte Prepare und Run eingreifen. Es ergab sich, dass für diesen Prototyp einer Erweiterung die Hauptaufgaben in der Run-Phase lagen, weswegen die Prepare weiterhin außen vor blieb. Entsprechend sollte die Erweiterung auch kompatibel mit den aktuell von dem Framework genutzten Konzepten bleiben. Im Idealfall ist die Erweiterung optional, was bedeutet, man kann sie je nach Belieben benutzen oder nicht. Um mit dem Workflow konform zu bleiben, wurde bei der Einbettung der Erweiterung darauf geachtet, eventuelle nötige Änderungen am Workflow insofern nur soweit vorzunehmen, dass am Ende der Workflow nach der Umlenkung wieder zusammenlaufen kann und kein neuer „Workflow-Zweig“ mit separatem Ende entsteht. Damit bleiben der Einsprungpunkt und Endpunkt für die Erweiterung faktisch unangetastet und durch eine einzelne Bedingung eine nötige Umlenkung eingeleitet, je nachdem, wie der Benutzer seinen Workload einstellt. In der aktuellen Umsetzung wird, sobald ein Element des neuen „MulTeX“-Modells im Workload entdeckt wird, diese Bedingung bereits erfüllt und die notwendigen Schritte intern eingeleitet. Dem Benutzer wird dadurch, außer seinen vorgenommenen Spezifikationen im Workload, kei-

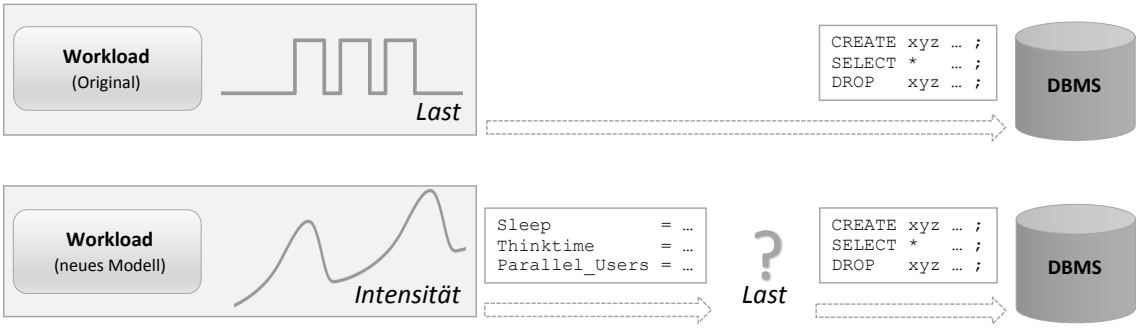


Abbildung 4.1: Funktionsweise des neuen *MulTe*-Workload durch Intensitäts-Modell

nerlei Unterschied im internen Ablauf bewusst. Er kann so alle Elemente des Workflows nutzen wie vorher, mit dem Vorteil, einfach im Workload sein neues Modell zu spezifizieren, womit die komplette Abarbeitung intern auf die Ausführung der Modell-Erweiterung umgelenkt wird.

Wichtig ist hier noch das Konzept, welches genutzt wird, um die vom Benutzer durch den neuen Workload spezifizierte Intensität in geeignete Parametrisierung von *MulTe* umzusetzen. Gerade dadurch, dass der Workflow nicht unterbrochen werden sollte, müssen alle konfigurierten Parameter des neuen Modells am Ende wieder auf originale *MulTe*-Parameter abgebildet werden. Um dennoch die Umsetzung vom Input zweiter Stufe von dem erster Stufe zu trennen, geschieht in der Engine (während der Run-Phase) eine interne Umrechnung von Intensität in gängige *MulTe*-Parameter. Abbildung 4.1 zeigt auf, inwieweit die durch das Workload-Modell gebildete Intensität auf die ursprünglichen *MulTe*-Parameter abgebildet werden. Während im normalen Workflow der Workload nur bestimmte Parameter für Transaktionen annimmt (siehe dazu mehr in Kapitel 2.2), wird in der Erweiterung die Intensität aus dem Workload-Modell berechnet, anschließend wird diese in die *MulTe*-Parameter algorithmisch umgerechnet, welche dann die einzelnen Transaktionen ausführen. Dabei ist der große Vorteil, dass die originalen *MulTe*-Parameter, auf denen die Intensität abgebildet wird, damit zur Laufzeit änderbar werden und nicht von vornherein durch den Workload auf nur einen Wert festgelegt werden. Das ist auch der Grund, warum sich durch diese Erweiterung eine Intensität umsetzen lässt.

Modell, Sessions und Events

MulTe besitzt in der Prepare-Phase mitunter die Aufgabe, aus den Templates die eigentlichen Tenants zu generieren und diese dann zu materialisieren. Um die Machbarkeit dieser Arbeit aufzuzeigen, war ein Prepare nicht zwingend notwendig, weswegen die Tenants einfach manuell erstellt wurden und nur mit der Run-Phase gearbeitet wurde. Warum dies funktioniert, zeigt der Quellcode-Ausschnitt [4.1] kurz auf.

Dieser XML-Ausschnitt aus einer Tenant-Definition zeigt, was die Prepare-Phase aus den Templates generiert hat. Das Prepare übernimmt ein Template, befüllt es mit vom Benutzer spezifizierten Werten und legt es unter einer eigenen Struktur ab. Man kann sozusagen auf die Prepare-Phase verzichten, wenn man sich dieses Tenant-XML vornimmt und per Hand die Werte verteilt. Definiert wird hier der Tenant durch diverse Parameter wie zum Beispiel die Anzahl an Parallelität (`parallelusers`) oder der Pool von zur Verfügung stehenden SQL-Queries. Durch die ersten vier Parameter ([4.1]: Zeile 5-8) wird ein einfaches „An/Aus“-Modell generiert: man definiert die Last

```
<?xmlversion="1.0"encoding="UTF-8"?>
<suitename="exampleWorkloadSuite" xmlns:xsi=[...]>
<workload name="tpch query 1" xmlns:xsi=[...]>
  <database>tenant_0000</database>
  <parallelusers>2</parallelusers>
  <when_active constraint="seconds">20</when_active>
  <thinktime>500</thinktime>
  <meansleeptime>10</meansleeptime>

  <transactions>
    <!--#### transaction 01 ####-->
    <transaction weight="1" name="PricingSummaryReport">
    <location>
      /home/multeuser/multe/tenants/tenant_0000/queries/01.sql
    </location>
    <parameters>
      <parameter>
        <uniformint min="60" max="120"/>
      </parameter>
    </parameters>
    </transaction>
  </transactions>
</workload>
```

Quellcode 4.1: Einfache Tenant-Definition

durch die Höhe der Parallelität sowie die Pausenzeiten zwischen den einzelnen Transaktionen. Die genauen Parameterfunktionen finden sich in Tabelle 2.2 im Kapitel 2.2.

Der Kern des *MulTe*-Frameworks ist die Workload-Engine, welche Teil der Run-Phase ist. Diese Engine musste dem neuen Modell angepasst werden, um die Umlenkung des Workflows zu gewährleisten. Sie nimmt die Workload-Definitionen der Tenants entgegen und führt diese gegenüber dem DBMS aus. Um das komplexe Modell als Erweiterung in *MulTe* einzubauen, ist es notwendig gewesen, den Input zweiter Stufe (die neue Intensität) auf den Input erster Stufe (Last) herunter zu brechen, das heißt, auf Parameter, welche MulTe bereits kennt. Es ist leider so, dass *MulTe* beim Benutzen des aktuellen Builds nicht in der Lage ist, seine Parameter während der Laufzeit zu verändern, was aber die Voraussetzung dafür wäre, Intensität darzustellen. Deswegen wurden Zwischenstadien erschaffen, welche die Kurve diskret auf einzelne Konfigurationen abbilden. Daher wurde entschieden, dass ein „MulTeX“-Modell in *MulTe* eine Liste vieler einzelner Modelle darstellt. Jedes dieser kleinen Modelle (auch *Sessions* genannt) ist selbst ein „MulTeX“-Modell, mit jeweils eigenen sechs Modell-Parametern (auch *Konfiguration* genannt). Weiterhin wird eine Session in viele Events unterteilt. Das bedeutet, dass jede Session mit seiner eigenen Konfiguration eine Kurve bildet, welche dann diskret auf viele Events abgebildet wird, und somit die eigentlich nicht zur Laufzeit änderbaren *MulTe*-Parameter letztlich doch veränderbar werden, indem eine Reihe von Events gekoppelt wird, die das Framework mit jeweils eigenen *MulTe*-Parametern versorgen. Abbildung 4.2 zeigt dafür eine geeignete grafische Darstellung von Modell, Sessions und Events. Dabei ist es wichtig zu erkennen, dass eine Session immer den Input zweiter Stufe und ein Event immer den Input erster Stufe hält, und durch algorithmische Umrechnung eine Konfiguration der Sessions in *MulTe*-Parameter der Events geschieht, um die dynamische Anpassung zur Laufzeit zu erreichen. Man sollte beachten, dass die Umrechnung immer noch im Voraus geschieht, sich die Parameter damit zwar zur Laufzeit diskret

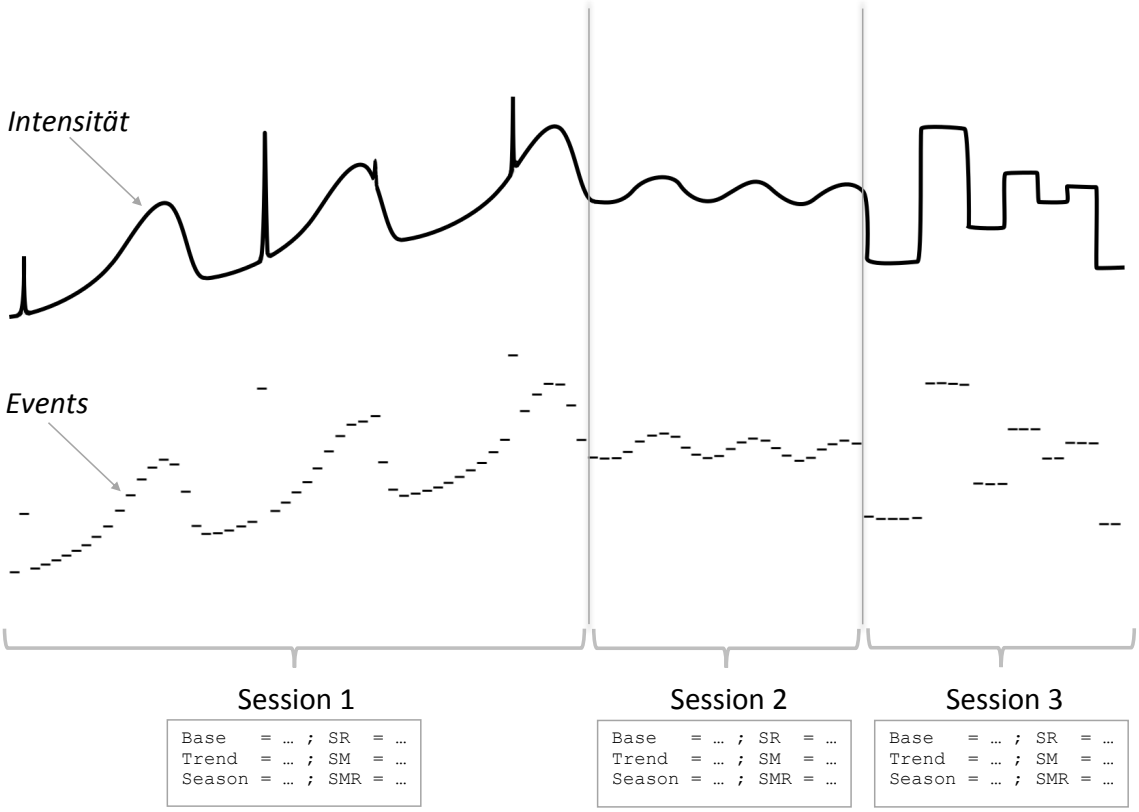


Abbildung 4.2: Diskretisierung der Intensitäts-Kurve durch Sessions und Events

ändern, allerdings feststehen und nicht mehr durch den Benutzer änderbar sind. Der Vorteil liegt dann darin, dass zur Laufzeit mehrere Werte für einen Parameter genutzt werden können, anstatt nur einen. Der Umweg über die Zersplitterung in Sessions und Events geschieht daher nur, weil *MulTe* im Originalzustand keine Variabilität der Parameter während eines Benchmark-Testlaufs vorsieht.

Durch diese Aufteilung der eigentlichen Kurve in kleine Abschnitte ist es möglich, ganz verschiedene Kurven aneinanderzuhängen und damit weitaus mehr Modellierungsmöglichkeiten zu nutzen, als es ein einzelnes Modell ohnehin schon kann. Durch das „MulTeX"-Modell an sich und die Umsetzung in *MulTe* ist es nun möglich, so gut wie jede Kurve geeignet und gut lesbar mit relativ wenigen Parametern darzustellen. Durch Base und Trend sind Verläufe in der Höhe der Intensität kein Problem mehr, durch die Season kann man zudem ganz spezielle Muster mit der generischen Kurve überlagern und so gewünschte Anpassungen vornehmen, und mit den Spikes kann man zufällige Ereignisse hinzufügen, welche durch geeignete Belegung der Konfiguration eine weitreichende Form annehmen können, von konstantem Rauschen bis hin zu positiven oder negativen Extrem-Ausreißern.

4.2 TECHNISCHE UMSETZUNG

Je nach Benchmark muss dieses Modell anders implementiert werden. Durch den speziellen Aufbau von *MulTe* ist es möglich, den ursprünglichen Ansatz beizubehalten und das neue „MulTeX"-

```
<?xmlversion="1.0"encoding="UTF-8"?>
<suitename="exampleWorkloadSuite" xmlns:xsi=[...]>
<workload name="tpch query 1" xmlns:xsi=[...]>
  <database>tenant_0000</database>
  <parallelusers>2</parallelusers>
  <when_active constraint="seconds">20</when_active>
  <thinktime>500</thinktime>
  <meansleeptime>10</meansleeptime>

  <!--#### NEW SESSION NODE ####-->
  <sessions loop="-1" sleep="60">
          <session id="1" name="s_test" slots="15" sleep="10"
           loop="0" seed="1337">
                  <base>5</base>
                  <trend>1</trend>
                  <season>2 1 1 4</season>
                  <spk_prb>0.2</spk_prb>
                  <spk_mrk>5</spk_mrk>
                  <spk_mrk_rnd>2</spk_mrk_rnd>
          </session>
  </sessions>

  <transactions>
     [...]
  </transactions>
</workload>
```

Quellcode 4.2: XML-Knoten für die Modell-Definition

Modell als Erweiterung einzubauen. Leider kann dies nicht komplett modular geschehen, da dafür noch zu viele Eingriffe in diverse Aspekte von *MulTe* nötig sind. Es sei hier nochmals erwähnt, dass diese technische Umsetzung als Prototyp gilt, um die Machbarkeit aufzuzeigen. Viele Optimierungen können weiterhin vorgenommen werden (siehe Kapitel 6).

Modell

Da bei *MulTe* die Workload-Engine nur mit den fertigen Tenant-Definitionen arbeitet, ausliest und auswertet, ist es möglich, eine für den Benutzer einfach lesbare Spezifikation für das Modell in eben diese Tenant-Struktur einzubauen, welche dann ausgewertet wird. Die bisherige und ursprüngliche Definition, wie sie in Quellcode [4.1] beispielhaft aufgezeigt wurde, erlaubt eine XML-Erweiterung im Sinne der Einführung eines neuen XML-Knotens, welcher alle benötigten Parameter für das „MulTeX“-Modell aufnimmt. Der durch die „MulTeX“-Erweiterung eingeführte neue XML-Knoten `<sessions>`, der durch den Benutzer (und später durch vorgefertigte Templates automatisiert) erstellt wird, sähe dann wie in Quellcode [4.2] aus. An diesem Beispiel wird die Einordnung der neuen Konfiguration einer Session ersichtlich. Wie bereits erwähnt, wurde aus Gründen der Flexibilität ein Modell in mehrere Sessions aufgeteilt, wobei eine Session seine eigene Konfiguration hält. In Quellcode [4.2] bestünde das Modell nur aus einer einzelnen Session mit dem Namen `s_test`. Man erkennt leicht, dass neben der Konfiguration der Session noch weitere, für die Verarbeitung notwendige oder optionale Parameter angegeben werden. Auf der

Ebene	Param.	Funktion	Wertebereich
sessions	loop	Anzahl der Wiederholungen für das gesamte Modell (alle Sessions); wenn -1, dann unendliche Wiederholung; wenn 0, dann keine Wiederholung	$\mathbb{N}, -1$
	sleep	Zeit in Millisekunden für Pause *nach* dem Modell; nur wenn $\#Sessions > 1$	$\mathbb{N}$
session	id	ID der Session, bestimmt die Reihenfolge der Ausführung aller Sessions	$\mathbb{N}$
	name	Name der Session, wird für die Auswertung in Logs etc. benötigt	$A \ldots Z, a \ldots z, _, \mathbb{N}$
	slots	Anzahl der für die Session generierten Events („Vorausberechnung“)	$\mathbb{N}$
	sleep	Zeit in Millisekunden für Pause *nach* der Session	$\mathbb{N}$
	loop	Anzahl der Wiederholungen für die Session; wenn 0, dann keine Wiederholung	$\mathbb{N}$
	seed	Seed für wiederholbare Zufälligkeit (von Spikes) in der Session	$A \ldots Z, a \ldots z, \mathbb{N}$

Tabelle 4.1: Parameter-Spezifizierung im neuen *MulTe*-XML-Session-Knoten

obersten Ebene `sessions` finden sich zum Beispiel zwei Parameter für Modell-Einstellungen in Bezug auf Wiederholungen und Pausenzeit nachdem das Modell einmal durchlaufen wurde. Die vollen Details der einzelnen System-Parameter für das neue „MulTeX“-Modell finden sich in der Tabelle 4.1. Die Parameter für die `session` selbst sind größtenteils gleich dem „MulTeX“-Modell benannt, das heißt, dass `base` auch die Base des Modells darstellt. Der einzige kleine Unterschied liegt in `spk_prb`, was eher historisch bedingt nach (engl.) „Spike Probability“ benannt wurde und deswegen diese Abkürzung trägt, funktional allerdings exakt gleich mit dem Modellparameter *SR* ist. Ansonsten stellen `spk_mrk` das *SM*, und `spk_mrk_rnd` das *SMR* des Modell dar.

Die Auswertung des Modells, genauer gesagt des `sessions`-Knotens mit allen enthaltenen Sessions, obliegt dem `WorkloadDriver`, der Workload-Engine von *MulTe*. Dieser hatte bisher nur die Aufgabe, parallele Nutzer zu starten, welche dann die Tenant-Definition auf das DBMS umsetzten. Dieser Ablauf wurde umgelenkt, sodass die Engine nun nicht mehr die Nutzer sondern die Sessions startet und verwaltet. In Abbildung 4.3 ist noch einmal grafisch der neue Ablauf innerhalb der Workload-Engine dargestellt. Im Hintergrund des gesamten Engine-Workflows läuft ein Timer, mit dem die gesamte Ausführung überwacht wird und bei Ablauf des Timers unterbrochen wird. Dieser Timer ist ein Standard in *MulTe* und kann ebenfalls auch die Ausführung des „MulTeX“-Modells handhaben. Das gesamte Modell kann sich durch den Parameter `loop` mit dem Wert `-1` unendlich oft wiederholen lassen, oder durch einen Wert mit natürlicher Zahl auch nur endlich oft. Wie man ebenfalls in dem Beispiel erkennen kann, sind alle anderen Parameter, die *MulTe* bereits vorher kannte (um das „An/Aus“-Modell zu realisieren) immer noch enthalten.

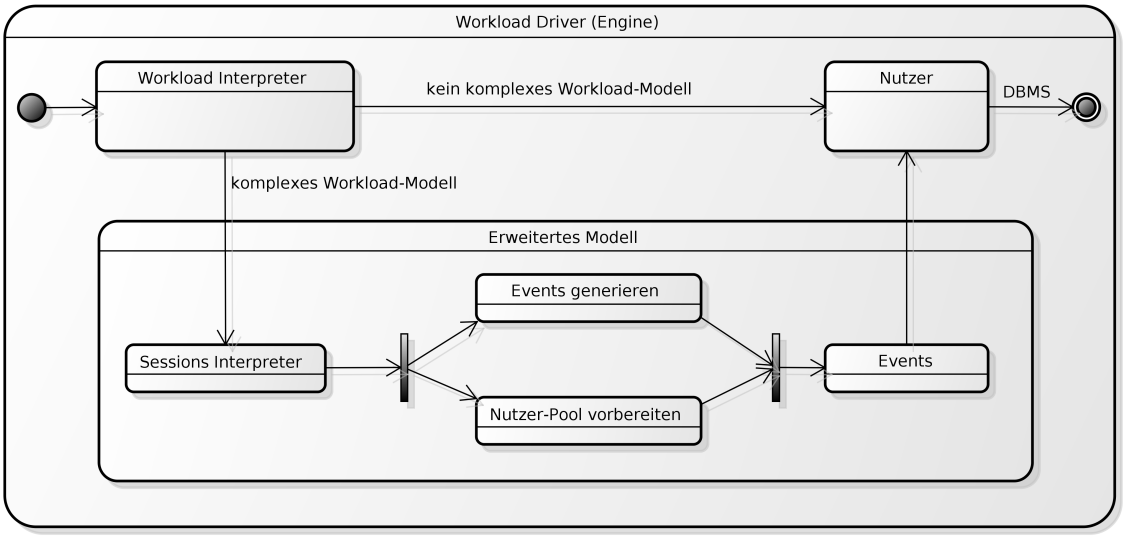


Abbildung 4.3: Umlenkung des Ablaufes in *MulTe*-Workflow der Engine

Das dient dazu, dass der Benutzer jederzeit zwischen dem einfachen Modell und der erweiterten Version umschalten können soll. Sobald im Workload der `sessions`-Knoten entdeckt wird, geht *MulTe* davon aus, dass die Erweiterung mit dem „MulTeX"-Modell genutzt werden soll, und ignoriert die dafür unnötigen Parameter. Im folgenden wird davon ausgegangen, dass der Benutzer das erweiterte Modell benutzen möchte und hat dementsprechend einen Knoten für die Sessions definiert. Das führt dazu, dass fast alle alten Parameter von *MulTe* ([4.2]: Zeile 5-8) nicht wiederverwertet werden können. Einzig der Parameter `when_active` wird für das neue Modell benutzt, jedoch nicht in der ursprünglich angedachten Weise, dazu später noch mehr.

Bevor in den folgenden Absätzen auf die einzelnen Implementierungsschritte genauer eingegangen wird, steht in Abbildung 4.4 noch ein vereinfachtes UML-Diagramm der implementierten Erweiterung mit den wichtigsten Methoden und Parametern zur Verfügung, anhand denen die nachfolgenden Absätze nachvollzogen werden können.

Session

Alle Sessions werden von dem `WorkloadDriver` nach `id` sortiert und dann nacheinander und unabhängig voneinander abgearbeitet. Jede Session bekommt durch die Tenant-Definition durch den Benutzer eine eigene Konfiguration zugewiesen, welche die Session dann per Algorithmus in eine *MulTe*-Parametrisierung umwandelt. Dies ist nötig, da das neue Modell als Erweiterung als Basis immer noch die Ausführungen und Parameter von *MulTe* benutzen muss. Der derzeit implementierte Algorithmus [4.3] ist recht einfach und bildet die Basis-Umrechnung konsistent sowie Monotonie-erhaltend ab.

Anders als ein gesamtes Modell (das hieße ohne Session-Unterteilung), wo die Laufzeit uninteressant ist und nur durch den Engine-Timer begrenzt ist, muss mit einer Unterteilung in einzelne Sessions definiert werden, wann die aktuelle Session beendet ist und die nächste gestartet werden kann. Dafür bietet die Session den Parameter `slots` an, welcher verfügbare Plätze für Events vordefiniert. Da jede Session eigen konfiguriert werden kann, ist es nötig, die Länge des zu be-

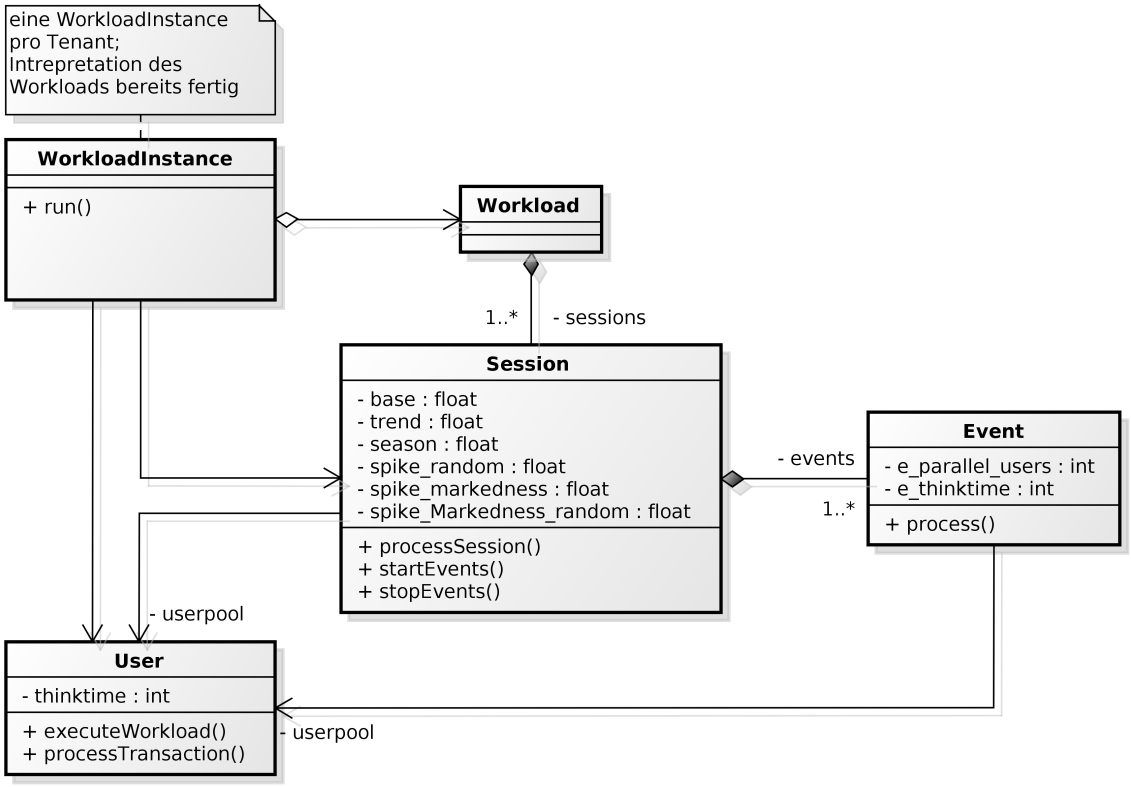


Abbildung 4.4: Erweiterung der Engine für das Interpretieren des neuen Modells

rechneten Modells bereitzustellen, bevor die nächste Session gestartet wird. Die Session bietet durch diesen Parameter die Möglichkeit, die Anzahl der Schritte i anzugeben, für die die Session dann das entsprechende Modell auf Basis der Konfiguration vorausberechnet. Dabei ist das hier Schritt i äquivalent zu dem Zeitpunkt t aus der *Intensity*-Funktion aus Kapitel 3.3 auf Seite 29.

Zu dem Berechnen der Slots verfügt jede Session über die Methode `processSession()`, welche intern zuerst die jeweilige Intensität auf Basis der *Intensity*-Funktion berechnet und anschließend den gezeigten Algorithmus [4.3] nutzt, um für jeden Slot die Konfiguration in eine *MulTe*-Parametrisierung umzusetzen. Dazu wird die Berechnung für Schritt i in zwei Teile zerlegt: 1) die Berechnung der „Timeseries"-Komponente in `calcTimeseries(i)`, und 2) die Berechnung der „Spike"-Komponente in `calcSpikes()`. Da das i für die Spikes bereits in dem Zufallszahlengenerator `rng` indirekt enthalten ist, benötigt nur noch die erstgenannte Komponente den entsprechenden Schritt. Wie in der Funktion für die Berechnung der Intensität gegeben, wird hier der erste Teil der Intensität äquivalent berechnet:

```
base + trend*i + season[i]
```

(vereinfacht dargestellt). Der zweite Teil der Intensität wird durch die Berechnung der Spikes vorgenommen und ist analog äquivalent zur *Intensity*-Funktion:

```
if(spk_prb) rng.nextBoolean() ? (spk_mrk + rngFloat) : (spk_mrk - rngFloat)
```

wobei `spk_prb` das SR, und `spk_mrk + rngFloat` sowie `spk_mrk - rngFloat` den letzten Teil $(SM \pm SMR)$ darstellt, wobei je nachdem wie `rng.nextBoolean()` auswürfelt entweder addiert

```
private Random rng;

public void processSession(/*[...]*/){
  //[...]
  float ts  = calcTimeseries(i);
  float spk = calcSpikes();
  Float intensity = Math.max(0, new Float(ts + spk) );
  //[...]
  // create event here
}

private float calcTimeseries(int step) {
  return base + ((step-1) * trend)
              + season.get( (step-1) % season.size() );
}

private float calcSpikes() {
  float prb = rng.nextFloat();
  if(spk_prb >= prb) {
    float rngFloat = new Float( (rng.nextFloat() * spk_mrk_rnd) );
    return rng.nextBoolean() ?
              (spk_mrk + rngFloat) : (spk_mrk - rngFloat);
  }
  return 0;
}
```

Quellcode 4.3: Einfacher Algorithmus zur Konvertierung von Konfiguration

oder subtrahiert wird. Wurden nun für jeden Slot die beiden einzelnen Intensitäten jeweils berechnet, wird eine Gesamt-Intensität pro Slot gebildet, welche die Grundlage für die einzelnen Events darstellt. Nun kommt der bereits genannte Algorithmus [4.3] zu Tragen, indem die Session pro Slot aus der vorberechneten Intensität ein Event generiert. Da die Event-Generierung sehr einfach aufgebaut ist, lässt er sich wie folgt mit Worten beschreiben:

1. die ganze Zahl der Intensität wird (aufgerundet) in parallele Nutzer umgewandelt, welche später gleichzeitig ihre Transaktionen ausführen,
2. die Nachkommastelle wird indirekt proportional zu 1000 Millisekunden als „Denkpause“ für einen Nutzer bestimmt.

So werden zum Beispiel für $Intensity = 3.6$ durch den Algorithmus vier parallele Nutzer erstellt, welche für den Tenant arbeiten, gleichzeitig bekommt einer von ihnen 400 Millisekunden Pause zwischen einzelnen Transaktionen, um den „halben“ Nutzer noch in geeigneter Monotonie darzustellen. So generiert theoretisch ein Tenant mit $Intensity = 3.6$ am Ende mehr Last auf dem DBMS als ein Tenant mit nur $Intensity = 2.0$ oder auch $Intensity = 3.5$.

Ist die Zuweisung von Intensität zu *Multe*-Parametern nach oben beschriebenem Verfahren erfolgt, ist die Methode `processSession()` abgeschlossen. Die Workload-Engine iteriert nun, nachdem jede Session präpariert wurde, erneut über alle Sessions und startet per `startEvents()`-Methode jede einzelne Session. Wichtig wäre hier noch zu erwähnen, dass bei einer Wiederholung des gesamten Modells (das heißt durch den `loop`-Parameter auf `sessions`-Ebene in der

Tenant-Definition) technisch bedingt eine erneute Berechnung aller Intensitäten vorgenommen und somit ein neues Modell generiert wird, währenddessen eine Wiederholung einer Session (das heißt durch den `loop`-Parameter auf `session`-Ebene) nur die Ausführung der Slots, beziehungsweise zugehöriger Events, wiederholt wird, nicht jedoch die Berechnung der Slots selbst. Damit ist zwar jede erneute Ausführung aller Sessions technisch gesehen immer ein neues Modell mit neu berechneter Intensität, allerdings wird durch gleiche „MulTeX"-Konfiguration, inklusive des Random-Seeds, auch immer ein komplett gleiches Modell generiert, unabhängig der Wiederholungen des kompletten Modells.

Event

Die Events werden, wie vorher beschrieben, durch vorherige Berechnungen der Session kreiert und verwaltet. Durch die Session-Methode `startEvents()` werden alle Events sequentiell ausgeführt, die von der Session generiert wurden. Die Events haben nun die Funktion angenommen, welche die Workload-Engine vorher hatte: das Verwalten der einzelnen parallelen Nutzer-Threads. Je nachdem, welchen Slot in der Session ein Event belegt und was die Session aus der berechneten Intensität für diesen Slot gemacht hat, besitzt das Event nun eine bestimmte Anzahl an Nutzern mit zugehörigen Denkpausen. Ein Event stellt nun den atomaren Bestandteil der Diskretisierung einer Intensitäts-Kurve dar, ist deswegen auch nur von sehr kurzer Dauer. Während eines Events ändern sich die *MulTe*-Parameter nicht mehr. Da mehrere Events eine Session füllen, ist die Session gerade noch in den Parametern quasi „zur Laufzeit" variabel, spätestens bei der Ausführung eines Events ist dies *MulTe*-systembedingt nicht mehr der Fall.

Die Laufzeit eines Events hängt von den eingangs erwähnten, im originalen *MulTe* bereits vorhandenen Parameter `when_active` ab. Dies ist der einzige Parameter, der aus der originalen Tenant-Definition mit genutzt wird. Alle anderen für das neue „MulTeX"-Modell benötigten Parameter werden durch den `sessions`-Knoten bestimmt. Die beiden Möglichkeiten, die der Parameter `when_active` einnehmen kann, sind:

1. gebunden an ein beschränktes Zeitfenster,
2. gebunden an eine bestimmte Anzahl an Transaktionen.

Je nachdem, welche Variante der Benutzer wählt, kann er ein Event beispielsweise derart spezifizieren, dass ein Event fünf Sekunden lang beliebig definierte Transaktionen ausführen lassen soll, oder aber auch, dass ein Event komplette fünf Transaktionen ausführen lassen soll.

Durch die Session wurde jedem Event eine gewisse Nutzerzahl mitsamt Denkpausen zugewiesen. Die eigentlichen Nutzer stellen jeweils einzelne Threads dar, welche von dem Event (je nach durch die Session zugewiesener Nutzeranzahl des entsprechenden Events) genutzt werden, um Verbindungen mit dem DBMS herzustellen. Ist ein Event vorbei, das heißt die zeitliche oder absolute Transaktionsanzahl wurde erreicht, wird das nächste Event innerhalb der Session gestartet. Aus Performance-gründen wurde ein Pool von Nutzer-Threads angelegt, deren Eigenschaften stetig variabel sind, sodass ein Event nicht alle Nutzer neu erstellen muss, sondern einfach nur eine benötigte Anzahl Nutzer-Threads aus dem Pool entnimmt und deren Eigenschaften neu belegt. Das eigentliche Erstellen der Nutzer übernimmt nur noch die Session direkt nach dem Erstellen der Events und lagert diese in den Pool ein. Ein Event lässt sich technisch bedingt nicht direkt

unterbrechen. Sollte der globale Engine-Timer ablaufen, wird das aktuelle Event noch beendet, bevor das Ausführen des nächsten Events unterbunden wird. Interessant ist der Vergleich, dass an diesem Punkt, wo das Event mit dem DBMS kommuniziert, das Konzept der Ausführung wieder dem originalen *MulTe* gleicht, da im Kontext des einzelnen Events wieder die gleichen Parameter umgesetzt werden, wie es zuvor im gesamten *MulTe* der Fall war. Durch das Management durch die Sessions und des Modells hingegen ist die notwendige Dynamik für das „MulTeX"-Modell jedoch auch durch die Erweiterung in *MulTe* gegeben.

Eine Besonderheit stellen hier Events dar, welche eine $Intensity = 0$ repräsentieren. Da in diesem Falle kein einziger Nutzer aus dem Pool entnommen werden müsste, besitzt jede Session einen `IdleUser`, welcher nur dafür da ist, eine entsprechend niedrige Intensität darzustellen. Ein Event mit $Intensity = 0$ kann jederzeit diesen speziellen Nutzer beanspruchen. Er baut, wie alle anderen Nutzer auch, ebenfalls eine Verbindung zu dem DBMS auf, nutzt diese aber nicht und verarbeitet auch keine Transaktionen. In der Praxis liegt der `IdleUser` nur für die entsprechende Beschränkung (zeitlich oder absolut) im Hintergrund und verfährt so, als ob er mit dem DBMS arbeiten würde, was natürlich nicht der Fall ist, da er einen inaktiven Nutzer ohne jegliche Intensität darstellen soll.

User

Die Nutzer sind zwar das kleinste Element, gehören aber theoretisch gesehen nicht mehr zu der „MulTeX"-Modell-Umsetzung. Sie repräsentieren nur noch die Transaktionen und den Aufwand (Last) gegenüber dem DMBS. Sie werden von den einzelnen Events gesteuert und mit den benötigten Variablen belegt (wie zum Beispiel Denkpausen für bestimmte Nutzer) und dann gestartet. Um die Einhaltung der zeitlichen oder absoluten Transaktions-Beschränkung kümmern sich die Nutzer selbst, da diese ebenfalls Zugriff auf den `when_active`-Parameter besitzen. Sind alle Nutzer fertig, wird dem zugehörigen Event das Ende signalisiert, woraufhin es sich beendet und das nächste Event sich die entsprechenden Nutzer aus dem Pool entnimmt und sich dahingehend einrichtet. Dabei sind dem Quellcode von Nutzern nur sehr wenig Änderung widerfahren, da diese streng konform mit der originalen Ausführung von *MulTe* bleiben mussten, sodass der Benutzer jederzeit zwischen der Erweiterung und der normalen Ausführung wechseln kann. Die Nutzer stellen dementsprechend den Ausgang der Modell-Umleitung dar.

Um sicherzustellen, dass sich die Nutzer-Threads nicht beenden, nachdem sie ihre Aufgabe erledigt haben und weiterhin im Pool für andere Events verfügbar sind, wurde ein *Sleep* implementiert, welches immer dann aktiviert wird, wenn der Nutzer-Thread nicht gebraucht wird. Der Quellcode-Ausschnitt [4.4] zeigt die entsprechende Stelle in dieser Klasse. Sollte `pause` auf `false` gesetzt werden, wird der Nutzer wieder aktiv, bis `pause` wieder auf `true` gesetzt wird. Ebenso wird diese gesamte Schleife verlassen, sobald `terminate` auf `true` erkannt wird. Das ist nötig, damit am Ende des Modells die einzelnen Nutzer geregelt beendet werden können.

Dadurch, dass die Nutzer in ihrer Funktionsweise fast nicht abgeändert wurden, bleibt auch die Analyze-Phase gleich und die Nutzer produzieren immer noch den gleichen Output wie das Standard-*MulTe* ohne Erweiterung.

```
while(!terminate) {
  if(!pause) {
    executeWorkload();  // commit transactions
    pause = true;
  }
  Thread.sleep(500);
}
```

Quellcode 4.4: Eine Ausführung von Sleep wenn ein Nutzer inaktiv ist

4.3 BENUTZUNG DER ERWEITERUNG

Für den Benutzer des Benchmarks ändert sich nur sehr wenig in seinen Aufgaben. Für ihn ist lediglich ein neuer Knoten im XML zu befüllen, über dessen Werte er sich Gedanken machen müsste. Sobald (mit der Erweiterung) ein `sessions`-Knoten in der Tenant-Definition vom *MulTe*-System entdeckt wird, kommt das neue Modell sofort zur Anwendung. Überflüssige Angaben in der Tenant-Definition werden dann einfach ignoriert. Sollten mehrere Tenants (mit je eigenem Workload) definiert sein, hängt es von jedem einzelnen Workload ab, ob der zugehörige Tenant das neue Modell nutzt oder die originale *MulTe*-Ausführung beibehält. Somit können mehrere Tenants nebenher existieren, welche jedoch unterschiedliche Ausführungen vollführen, da für jeden Tenant der Workload separat ausgewertet und dadurch seine Ausführung definiert wird.

Dabei bleiben für den Benutzer die eigentlichen Phasen scheinbar unangetastet, auch auch die spätere Analyse-Datei gibt nicht sofort Hinweise darauf, dass hier ein anderes Nutzermodell zur Anwendung gekommen ist, da die einzelnen Nutzer, wie oben beschrieben, den gleichen Output erzeugen, basierend auf den Input erster Stufe, den Events (welche die normale *MulTe*-Parametrisierungen besitzen). Um einzuschätzen, ob das neue Modell etwas bewirkt hat, das heißt um den Input zweiter Stufe (der Sessions und deren Modell-Konfiguration) zu beurteilen, damit beschäftigt sich diese Arbeit in Kapitel 5 ausführlicher.

5 EVALUIERUNG DER NEUEN *MULTE*-ERWEITERUNG

Um die Evaluierung der Intensität vornehmen zu können, wurde ein geeigneter Output zu *MulTe* hinzugefügt, da, wie bereits weiter oben beschrieben, die Intensität nicht direkt am eigentlichen Output von *MulTe* ablesbar ist. Um in diesem Fall die Intensität geeignet Auslesen zu können, werden nicht die einzelnen SQL-Query-Zeiten benötigt, sondern eher eine absolute Kenngröße, wie zum Beispiel „Queries pro Sekunde“ oder „pro Event“. Dadurch kann abgelesen werden, inwieweit die Intensität in das letzte Glied der Kette (also in SQL-Queries) umgesetzt wurde. Die dazu implementierte Schnittstelle in der neuen *MulTe*-Erweiterung erfasst dazu, während eine Transaktion ausgeführt wird, diverse Parameter und schreibt dazu eine Zeile pro Transaktion in den Output-Log. Zu den erfassten Parametern gehören Name und Startzeit der Transaktion, aber auch Namen des Events und des Nutzers sowie die zu dem Event gehörige Intensität und parallelen Nutzer insgesamt.

5.1 LOG-SCHNITTSTELLEN IN *MULTE*

Für die Auswertung des durch die Intensitäts-Schnittstelle generierten Outputs kamen unterschiedliche Tools zum Einsatz. Ein Ausschnitt der ersten Zeilen des generierten Outputs könnte wie in Tabelle 5.1 gegeben aussehen.

```
s_hill_event1,2.0,2,0,2015-04-30 14:41:44,2,2
s_hill_event1,2.0,2,0,2015-04-30 14:41:45,2,1
s_hill_event1,2.0,2,0,2015-04-30 14:41:45,2,0
s_hill_event1,2.0,2,0,2015-04-30 14:41:45,2,1
s_hill_event1,2.0,2,0,2015-04-30 14:41:45,1,3
s_hill_event1,2.0,2,0,2015-04-30 14:41:45,1,0
```

Tabelle 5.1: Log zur Bestimmung der Intensitäten und der Evaluierung des Modells

Dabei sind die Spalten dieser CSV-Datei folgendermaßen eingeteilt: Name des Events, Intensität des Events, parallele Nutzer des Events, Denkzeit des Events, Zeitpunkt der Ausführung der

Gruppierung nach Event	*Gruppierung nach Sekunde*
`s_hill_event1,10`	`2015-04-30 16:34:00,5`
`s_hill_event2,15`	`2015-04-30 16:34:01,5`
`s_hill_event3,15`	`2015-04-30 16:34:02,11`
`s_hill_event4,20`	`2015-04-30 16:34:03,2`
`s_hill_event5,20`	`2015-04-30 16:34:04,2`
`s_hill_event6,25`	`2015-04-30 16:34:05,3`

Tabelle 5.2: Log mit Gruppierung nach Spalte und Häufigkeit des Auftretens

Transaktion, Nummer des ausführenden Nutzers und Nummer der ausgeführten Transaktion.

Da *MulTe* durch den `when_active`-Parameter in der Tenant-Definition sowohl zeitliche als auch absolute Begrenzung von Transaktionen zulässt, müssen beide Verfahren getrennt ausgewertet werden. Beide Methoden geben dabei einen Überblick, wie oft das DBMS mit Transaktionen beansprucht wurde und versuchen so, die Umsetzung von Intensität in Last darzustellen. Es ist dagegen nicht so einfach, einzelne Denkpausen von Nutzern nachzuweisen. Da aber die Denkpausen nicht derart ausschlaggebend sind wie die totale Anzahl an Nutzern, wurde in dieser Analyse darauf verzichtet. Sollte man in zukünftigen Arbeiten den Algorithmus zur Umsetzung von Intensität in Last abändern und mehr Mechanismen nutzen als nur die Anzahl an parallelen Nutzern und die Denkpausen, muss dabei bedacht werden, wie die Umsetzung dieser neuen Parameter geeignet evaluieren kann.

Jede Zeile im Intensitäts-Log steht für eine Query, die abgesetzt wurde. Um die entsprechenden Spalten zu gruppieren und deren Häufigkeit zu zählen, wurde im Rahmen dieser Arbeit ein kleines Tool in *Java* geschrieben, um ein möglichst einfaches Transformieren zu ermöglichen. Eine beispielhafte Ausgabe des Tools in einer separaten CSV-Datei sieht dann wie in Tabelle 5.2 aufgezeigt aus.

Die Spalten des neuen Logs beinhalten dann nur noch Werte für: Event und Häufigkeit, beziehungsweise Sekunde (Zeitstempel) und Häufigkeit. Dies reicht jedoch aus, um eine Analyse der durchgeführten Anfragen durchzuführen, indem man die Gruppierung in einem Diagramm auf der X-Achse, sowie die zugehörige aggregierte Anzahl an Anfragen auf der Y-Achse abträgt. Für die Erstellung dieser Diagramme kann die statistische Programmiersprache *R* oder auch *Microsoft Excel* benutzt werden. Im Rahmen dieser Arbeit wurde sich für letzteres entschieden, jedoch wird es bei späteren Analysen des Modells außerhalb des Umfangs dieser Arbeit durchaus sinnvoll sein, sich der mächtigeren Statistik-Umgebung in *R* zu widmen.

5.2 TESTUMGEBUNG UND -SZENARIEN

Hardware Bei den Tests kam ein einfaches System mit einer *Intel Core i3-3110M* @2.4GHZ Dual-Core-CPU zum Einsatz. Der Hauptspeicher besaß während des Testzeitraums 4GB Kapazität. Man kann leicht erkennen, dass dieses System nur eingeschränkte Rechenkraft besitzt, jedoch für einfache Analysen des erstellten Modells ausreichend sein sollte.

Software Neben dem *MulTe*-Framework wurde für die Tests das DBMS *MySQL* in der Version 5.5.43 benutzt. Das darunterliegende Betriebssystem war ein 64bit *Ubuntu 14.10* mit einem Linux-Kernel der Version 3.13.0-49 und allen notwendigen Paketen, um *MulTe* (inklusive *Python* und *Java 1.8*) sowie *MySQL* lauffähig zu machen. Während der Tests konnte der Test-Rechner ohne Unterbrechung anderer Anwendungen laufen und war auch keinen anderen Einflüssen, welche die Performance zusätzliche beeinträchtigen könnten, ausgesetzt.

Kontext und Vorgehensweise Für alle Test-Szenarien kam der gleiche Workload-Header zum Einsatz, welcher hauptsächlich die Länge eines einzelnen Events spezifiziert und in [5.1] aufgezeigt ist. Das `"seconds"` wird, je nachdem was benötigt wird, entweder so belassen (bei zeitlicher Transaktionsbeschränkung) oder gegen ein `"transactions"` ausgetauscht, wenn eine absolute Transaktionsbeschränkung vorliegen soll. Jedoch wird immer die entsprechende Bedingung mit dem Wert 5 versehen, was bedeutet, dass ein Event entweder 5 *Seconds* oder 5 *Transactions* läuft, beziehungsweise beinhaltet, um eine möglichst feinkörnige Diskretisierung der Kurve zu bewerkstelligen. Die gesamte Modell-Laufzeit betrug immer genau 30 Minuten. Meistens wurde in dieser Zeit das Modell ungefähr drei- bis viermal wiederholt. Es wurde nach jeder Session eine Pause von 10 Sekunden eingefügt, um die Trennung der einzelnen Sessions später im Log einfacher zu gestalten. Zum Schluss wurde noch ein Pool von vier verschiedenen Queries definiert, um eventuelle Caching-Effekte des DBMS etwas zu erschweren. Diese vier Queries waren recht unterschiedlicher Beschaffenheit und die Auswahl erfolgte zufällig pro Nutzer. In den Auswertungen der Ergebnisse, insbesondere bei den Diagrammen, ist immer nur die erste durchlaufene Session-Iteration des Modells aufgezeigt, da alle weiteren Sessions recht äquivalent ausfallen und dadurch eine gewisse Übersichtlichkeit für die Auswertung erzeugt wird.

Das einfache Test-Szenario Für die grundlegenden Tests wurde eine einfache Tenant-Definition gewählt, welche aus genau drei Sessions besteht, welche in Quellcode-Ausschnitt [5.2] parametrisiert sind. Dieser Test soll grundlegende Aussagen darüber treffen, ob man verschiedene Intensitäten an der Last, beziehungsweise am Query-Aufkommen, wiedererkennt.

Die erste Session *s_extreme_low* besteht aus einem Modell, welches nur sehr geringe konstante Nutzeraktivität darstellen soll. Mit einer Wahrscheinlichkeit von 20% kommt ein Spike hinzu, welcher relativ zur Base von 5 einen Intensitäts-Wert im Bereich von 1 bis 3 hinzu addiert. Mit 30 Slots, beziehungsweise 30 vorberechneten Events, bildet diese Session genau ein Drittel der gesamten Laufzeit. Die zweite Session *s_extreme_high* ist analog der ersten, nur dass hier die Parameter verzehnfacht wurden, das heißt, die Nutzerintensität steigt stark an, relativ gesehen zur ersten Session. Das bedeutet, Base liegt nun bei 50 und mit einer Wahrscheinlichkeit von 20% wird ein Spike mit einer Intensität im Bereich von 10 bis 30 hinzu addiert. Durch die Wahl des

```
<database>tenant_1000</database>

<!-- seconds or transactions -->
<when_active constraint="seconds">5</when_active>
<parallelusers>0</parallelusers>
<meansleeptime>0</meansleeptime>
<thinktime>0</thinktime>
```

Quellcode 5.1: Tenant-Definitions-Header

```
<!-- #### complex model ####-->
</sessions>
  <session id="1" name="s_extreme_low" slots="30"
           sleep="10" loop="0" seed="1337">
    <base>5</base>
    <trend>0</trend>
    <season>0</season>
    <spk_prb>0.2</spk_prb>
    <spk_mrk>2</spk_mrk>
    <spk_mrk_rnd>1</spk_mrk_rnd>
  </session>
  <session id="2" name="s_extreme_high" slots="30"
           sleep="10" loop="0" seed="1337">
    <base>50</base>
    <trend>0</trend>
    <season>0</season>
    <spk_prb>0.2</spk_prb>
    <spk_mrk>20</spk_mrk>
    <spk_mrk_rnd>10</spk_mrk_rnd>
  </session>
  <session id="3" name="s_extreme_hill" slots="30"
           sleep="10" loop="0" seed="1337">
    <base>0</base>
    <trend>2</trend>
    <season>0</season>
    <spk_prb>0</spk_prb>
    <spk_mrk>0</spk_mrk>
    <spk_mrk_rnd>0</spk_mrk_rnd>
  </session>
</sessions>
```

Quellcode 5.2: Tenant-Definition (Session-Knoten) des einfachen Test-Szenarios

gleichen Seeds für den Zufallsgenerator existieren die Spikes auch an der gleichen Stelle der Ausführung wie in *s_extreme_low*. Auch diese Session nimmt 30 Slots ein und bildet damit das zweite Drittel des Gesamtmodells. Die letzte Session mit dem Namen *s_extreme_hill* bildet den Übergang zwischen geringer und hoher Auslastung durch den Nutzer. Angefangen mit einer Base von 0, steigert sich die Intensität des Nutzer im Verlauf von 30 Slots auf 58, entsprechend dem Trend von 2. Hierzu wurden keinerlei Spikes genutzt. Die genauen Intensitäten der einzelnen Sessions und deren Events kann man in Tabelle 5.3 ablesen.

Das erweiterte Test-Szenario Bei diesem Szenario wurden fünf Sessions definiert, welche im Quellcode-Ausschnitt [5.3] zu sehen sind. Hier wurden, im Gegensatz zu dem einfachen Szenario, verschiedene Möglichkeiten der Modellierung getestet. Alle Sessions, bis auf die letzte, besaßen 15 Slots und wiederholten sich nicht selbst.

Die erste Session *s_hill* bildet einen leichten Anstieg mit einer rationalen Zahl als Trend. Das beeinflusst erstmalig die Denkzeit, da bei jeder nicht-ganzzahligen Intensität ein weiterer Nutzer mit Denkzeit (in diesem Testfall 500 Millisekunden) eingefügt wird, um den „halben“ Nutzer darzustellen. Zu 10% Wahrscheinlichkeit werden hier mittlere Spikes hinzugefügt. Die zweite Session *s_curve* sollte einen Tagesablauf darstellen, indem unter Season ein Muster hinterlegt wurde, welches an- und abschwellende Intensität hinzufügt. Das Muster reicht von -5 bis $+5$ und wie-

```xml
<!-- #### complex model ####-->
<sessions loop="-1" sleep="60">
  <session id="1" name="s_hill" slots="15"
           sleep="10" loop="0" seed="1337">
    <base>2</base>
    <trend>0.5</trend>
    <season>0</season>
    <spk_prb>0.1</spk_prb>
    <spk_mrk>5</spk_mrk>
    <spk_mrk_rnd>2</spk_mrk_rnd>
  </session>
  <session id="2" name="s_curve" slots="15"
           sleep="10" loop="0" seed="1337">
    <base>3</base>
    <trend>0</trend>
    <season>0 1 2 3 4 5 4 3 2 1 0
            -1 -2 -3 -4 -5 -4 -3 -2 -1</season>
    <spk_prb>0</spk_prb>
    <spk_mrk>0</spk_mrk>
    <spk_mrk_rnd>0</spk_mrk_rnd>
  </session>
  <session id="3" name="s_flat" slots="15"
           sleep="10" loop="0" seed="1337">
    <base>5</base>
    <trend>0</trend>
    <season>0 0 0 0 5 5 5 5 0 0 20 20</season>
    <spk_prb>0</spk_prb>
    <spk_mrk>0</spk_mrk>
    <spk_mrk_rnd>0</spk_mrk_rnd>
  </session>
  <session id="4" name="s_spikey" slots="15"
           sleep="10" loop="0" seed="1337">
    <base>0</base>
    <trend>0</trend>
    <season>0 -4 -10</season>
    <spk_prb>1</spk_prb>
    <spk_mrk>15</spk_mrk>
    <spk_mrk_rnd>3</spk_mrk_rnd>
  </session>
  <session id="5" name="s_single" slots="3"
           sleep="10" loop="4" seed="1337">
    <base>0</base>
    <trend>0</trend>
    <season>0 -4 -10</season>
    <spk_prb>1</spk_prb>
    <spk_mrk>15</spk_mrk>
    <spk_mrk_rnd>3</spk_mrk_rnd>
  </session>
</sessions>
```

Quellcode 5.3: Tenant-Definition (Session-Knoten) des erweiterten Test-Szenarios

Session	Slots	Event Intensity
s_extreme_low	30	[5 , 7.689 , 5 , 5 , 5 , 5 , 7.641 , 5 , 6.979 , 5 , 5 , 7.637 , 6.073 , 5 , 6.879, 5 , 5 , 5 , 7.743 , 5 , 7.571 , 5 , 5 , 5 , 5 , 5 , 5 , 5 , 5 , 5]
s_extreme_high	30	[50 , 76.892 , 50 , 50 , 50 , 50 , 76.411 , 50 , 69.789 , 50 , 50 , 76.368, 60.727 , 50 , 68.794 , 50 , 50 , 50 , 77.435 , 75.714 , 50 , 50 , 50 , 50, 50, 50 , 50 , 50]
s_extreme_hill	30	[0 , 2 , 4 , 6 , 8 , 10 , 12 , 14 , 16 , 18 , 20 , 22, 24 , 26 , 28 , 30 , 32 , 34, 36 , 38, 40 , 42 , 44 , 46 , 48 , 50 , 52 , 54 , 56 , 58]

Tabelle 5.3: Sessions, Slots und Events des einfachen Szenarios

Session	Slots	Event Intensity
s_hill	15	[2 , 2.5 , 3 , 3.5 , 4 , 4.5 , 5 , 5.5 , 6 , 6.5 , 7 , 7.5 , 8 , 14.138 , 9]
s_curve	15	[3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1, 0, 0]
s_flat	15	[5 , 5 , 5 , 5 , 10 , 10 , 10, 10 , 5 , 5 , 25 , 25 , 5 , 5 , 5]
s_spikey	15	[2.174 , 2.883 , 1.188 , 2.942 , 1.979 , 1.187 , 1.113 , 2.495 , 1.879, 1.056 , 2.743 , 2.184 , 2.988 , 2.441 , 2.647]
s_single	3	[15.552 , 13.65 , 2.563] (5 $\times$ ausgeführt)

Tabelle 5.4: Sessions, Slots und Events des erweiterten Szenarios

derholt sich ständig. Da diese Session aber nur 15 Slots zur Verfügung hatte, wurde das Muster nur einmal durchlaufen, bis hin zu dem Element -4, welche wiederum durch eine leicht erhöhte (aber begrenzte) Base von 3 zu *Intensity* 0 transformiert wurden, da Intensitäten kleiner als 0 nicht erlaubt sind. Zum besseren Verständnis sei hier nochmals auf Tabelle 5.4 verwiesen, wo erkennbar ist, dass die letzten beiden Elemente von *s_curve* eine 0 darstellen, obwohl hier theoretisch nach Modell die Zahlen 0 und -1 stehen müssten. Für die dritte Session *s_flat* wurden nur verschiedene Höhen definiert, um die schnellen hohen Unterschiede innerhalb einer Session zu modellieren. Für die nächste Session *s_spikey* ist eine Spezifikation getroffen worden, welche ein konstantes Rauschen simulieren soll. Dazu wurden Base, Trend, und Season jeweils auf 0 gesetzt und einzig die Spike-Konfiguration vorgenommen. Dies äußert sich darin, dass durch `Spike_Random=1` (beziehungsweise das `spk_prb`) permanent ein Spike generiert wird. Die Spike-Höhen an sich sind relativ klein, alle zwischen 1 und 3. Die letzte Session mit dem Namen *s_single* weicht von den anderen etwas ab. Sie besitzt nur drei Slots und dafür vier Wiederholungen, also läuft fünf mal durch. Modelliert wird hier ein einzelner großer Spike mit anschließender kurzer Rücklaufphase gefolgt von zehn Sekunden Pausenzeit. Damit ist es vom Verhalten ähnlich einem *Heartbeat* oder *Ping*. Die Spike-Parameter beschrieben drei zu 100% auftretende Spikes mit jeweils einer Intensität von 15, der um genau 3 abweichen kann. Danach wurde Base und Trend auf 0 gesetzt und für Season ein Muster für den entsprechenden großen Spike-Rücklauf darübergelegt. So wird der erste der drei Spikes original zugelassen, der zweite erhält einen Abzug von 4 Intensität und der letzte einen Abzug von 10 Intensität. Alle Sessions zusammen ergeben das erweitere Szenario. Die einzelnen belegten Slots mit den berechneten Events sind der Tabelle 5.4 zu entnehmen.

5.3 AUSWERTUNG DER TESTS

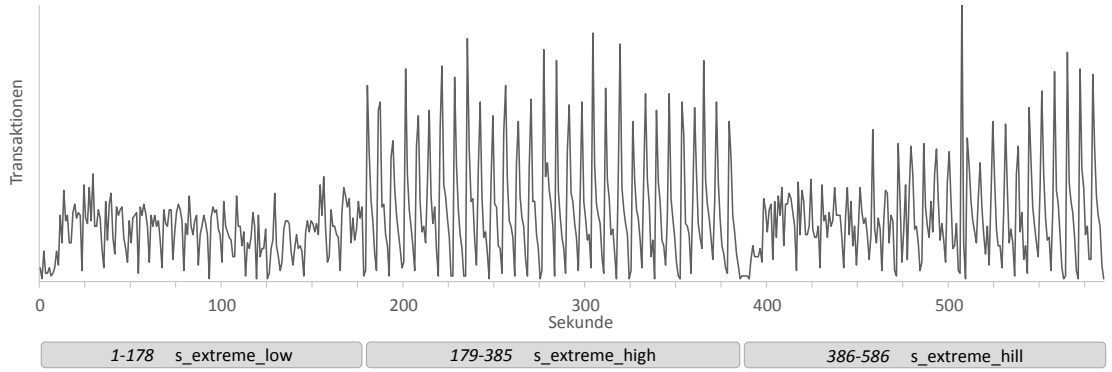
Wie eingangs erwähnt, unterscheidet man hier zwischen der Analyse von Ergebnissen bei absoluter und zeitlicher Beschränkung von Transaktionen gegen das DBMS. Im generellen Kontext kann bereits gesagt werden, dass eine zeitliche Beschränkung auf Diagramme der Event-Gruppierung nicht wirklich sinnvoll ist, hier jedoch trotzdem vorgenommen wird, um dieses Phänomen auch aufzuzeigen. Analog dazu ist es ebenfalls eigentlich nicht sinnvoll, eine absolute Beschränkung auf einem Diagramm mit Zeit-Gruppierung vorzunehmen, allerdings wird auch dieses abgehandelt. Zu beiden wird später näher darauf eingegangen. Bei einer zeitlichen Beschränkung von Transaktionen bedeutete dies konkret, dass jeder von einem Event generierter Nutzer ganze fünf Sekunden für seine Transaktionen besaß, wobei bei einer absoluten Beschränkung ein Nutzer immer garantiert fünf Transaktionen ausführen konnte. Dementsprechend unterschiedlich verhalten sich auch die gruppierten Last-Kurven in den kommenden Auswertungen.

Es ist in jedem Falle zu beachten, dass die Implementation und deren Auswertung eine Machbarkeitsstudie ist und es für groß angelegte Analysetätigkeiten in diesem Bereich eine eigene Arbeit nötig wäre. Es wird hier lediglich die Auswertung und richtige Umsetzung des Modells getestet, nicht jedoch die eigentliche Performance oder die Optimierung. Auch wird hier nur, wenn überhaupt, ein Vergleich zu dem derzeitig originalem Nutzer-Modell von *MulTe* vorgenommen, nicht jedoch zu einem der anderen in Kapitel 3 genannten Modelle.

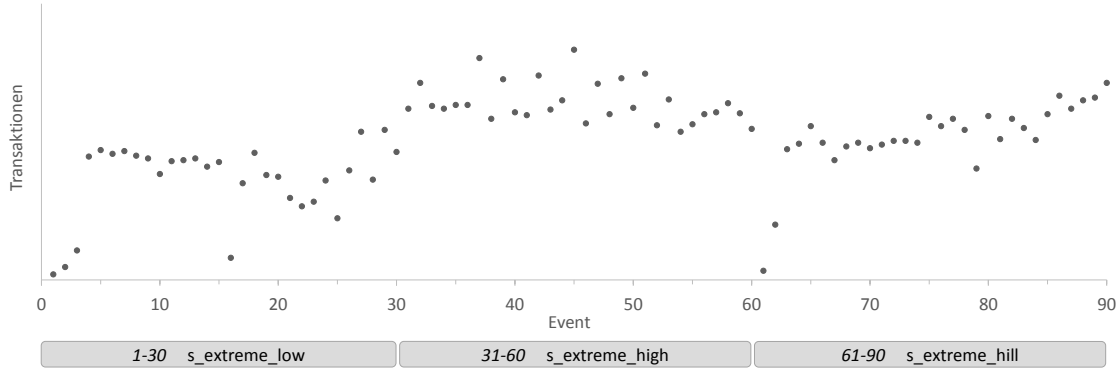
Auswertung des einfachen Szenarios

Zeitlich beschränkte Transaktionen Durch die Gruppierung nach Zeit enthält der Log nur noch die einzelnen Sekunden sowie die Häufigkeiten der in dieser Sekunde ausgeführten Transaktionen. Die entsprechende Last, welche durch die Intensität generiert wurde, zeigt sich in Abbildung 5.1. Man erkennt leicht das eingangs erwähnte Phänomen, dass, wenn man zeitlich gruppiert, auch das zeitliche Last-Diagramm 5.1a bessere Aussagen in Bezug auf die Intensität liefert als das nach Event gruppierte Diagramm 5.1b. Die grauen Balken stellen jeweils die zeitliche Ausprägung der jeweiligen Session dar. Man kann sehr gut erkennen, wie sich die in der Tenant-Definition spezifizierten Sessions im Verlauf des Tests widerspiegeln. Die erste Session synthetisierte eine geringe Intensität, welche sich dann in wenigen Nutzern mit dementsprechend wenigen Transaktionen äußerte. In der zweiten Session verhielt es sich genau umgekehrt und entsprechend viele Nutzer führten ihre Transaktionen in den zur Verfügung stehenden fünf Sekunden pro Event aus. In der letzten Session kann man sehr gut nachvollziehen, wie die Last an das System durch immer mehr Transaktionen anstieg, genauso wie es in der Tenant-Definition beschrieben wurde. Die einzelnen Spitzen in den Kurven entstehen durch die Laufzeit der einzelnen Threads, welche alle gleichzeitig eine Transaktion starten, und dann auf die Antwort des DBMS warten, wodurch nur in dem ersten Teil der Transaktions-Laufzeit wirklich Last generiert wurde.

Da jeder Nutzer vom Event immer fünf Sekunden zur Verfügung gestellt bekam, um seine Transaktionen auszuführen, sind dagegen im zweiten Diagramm 5.1b die einzelnen Sessions nicht ganz so gut voneinander trennbar. Durch unterschiedliche Laufzeiten der Queries kann es also vorkommen, dass ein Event durchaus signifikant mehr Transaktionen absetzt als ein anderes Event



(a) Last in Queries pro Sekunde

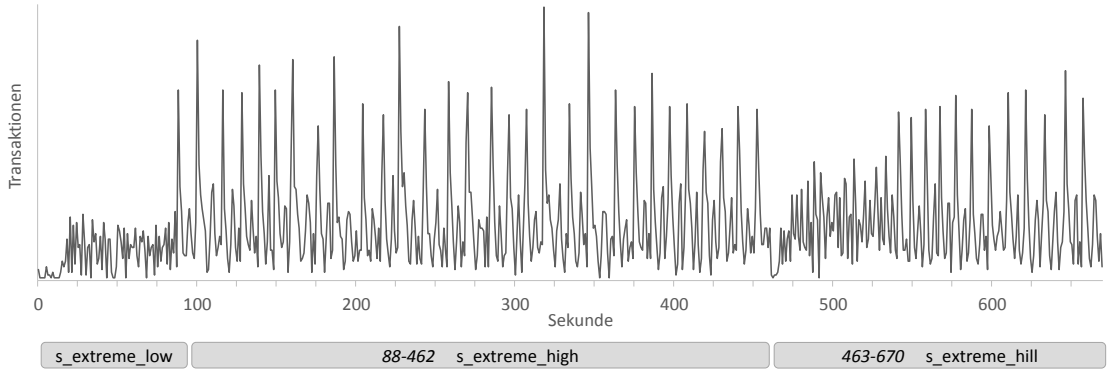


(b) Last in Queries pro Event

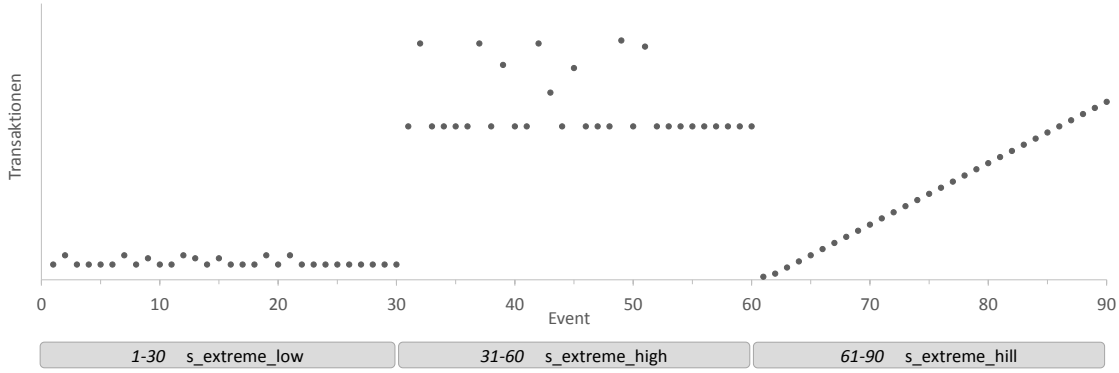
Abbildung 5.1: Umgesetzte Intensität in Last im einfachen Szenario mit zeitlicher Beschränkung

mit einer eigentlich höheren Intensität. Dennoch kann man ungefähr die einzelnen Sessions voneinander abtrennen, wieder gekennzeichnet durch die grauen Balken unter dem Diagramm.

Absolut beschränkte Transaktionen Anders verhält es sich bei einer absoluten Beschränkung von Transaktionen pro Nutzer im Event. Hier hatte ein Nutzer immer die Möglichkeit, fünf komplette Transaktionen auszuführen, bevor das nächste Event startete. Dementsprechend sehen hier die Diagramme anders aus als bei einer zeitlichen Beschränkung. Die entsprechend ausgewerteten Kurven kann man in Abbildung 5.2 betrachten. Das Diagramm 5.2a, dessen Werte nach Zeit gruppiert wurden, ist hier weniger aussagekräftig als 5.2b, dessen Werte nach Event gruppiert wurden. Das liegt sehr wahrscheinlich daran, dass die Events nun (bedingt durch tatsächlich mehr oder weniger Transaktionen pro Event) von unterschiedlicher Dauer waren, was dazu führte, dass eine Gruppierung nach Zeit nicht mehr einheitlich ist. Man kann es an der Dauer der ersten und zweiten Session erkennen (beziehungsweise an der Länge der grauen Balken darunter), wie sehr sich diese beiden unterscheiden. Das die Session *s_extreme_low* nur von sehr kurzer Dauer ist und dagegen die zweite Session *s_extreme_high* um ein vielfaches der Zeit andauert, liegt einfach daran, dass bei der ersten Session die Events recht schnell vorbei waren, da die wenigen Nutzer, welche pro *s_extreme_low*-Event existierten, ihre vorgegebenen fünf Transaktionen schnell erledigt hatten. Dagegen waren in der zweiten Session viel mehr Nutzer aktiv, welche das DBMS zu verarbeiten hatte, und da jeder Nutzer die vorgegebenen fünf Transaktionen zu erledigen hatte und damit die Gesamtanzahl an Transaktionen pro Event stieg, wurden die einzelnen Events einfach in die Länge gezogen und damit auch die gesamte Session. Dennoch kann man hier, unter Beachtung der zeitlichen Verschiebungen, die einzelnen definierten Sessions aus dem



(a) Last in Queries pro Sekunde



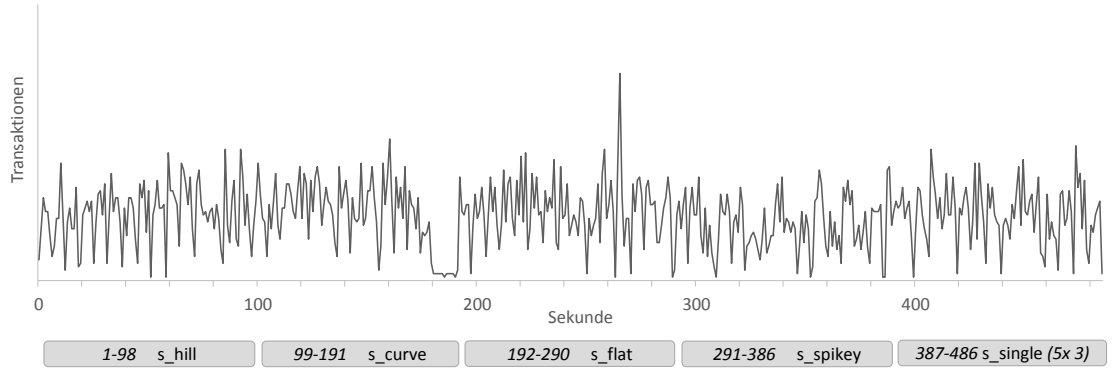
(b) Last in Queries pro Event

Abbildung 5.2: Umgesetzte Intensität in Last im einfachen Szenario mit absoluter Beschränkung

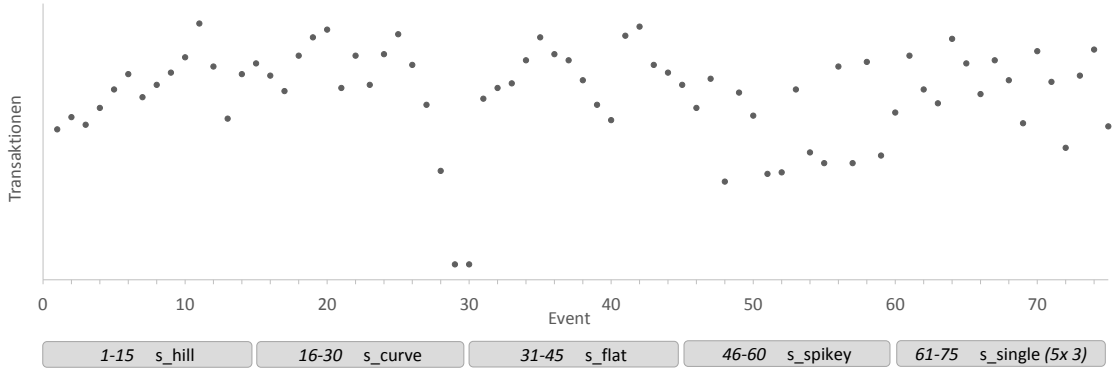
Workload gut ausmachen und auch deren Eingenschaften erkennen: die geringe Intensität, die hohe Intensität und auch der Anstieg der Intensität in der letzten Session.

Dennoch ist das zweite Diagramm 5.2b vorteilhafter, da hier die zeitliche Komponente keinerlei Rolle mehr spielt. Da nur die totale Anzahl an Transaktionen pro Event erfasst wird, spiegelt dieses Diagramm auch exakt das definierte Verhalten wider. Das resultiert daraus, dass, im Gegensatz zu zeitlicher Beschränkung von Events, hier bereits von vornherein fest steht, wie viele Transaktionen ein Nutzer (und damit auch ein Event) ausführen wird. Ein Event wird garantiert genau 15 Transaktionen ausführen, wenn man den Rahmen auf fünf Transaktionen pro Nutzer festlegt und es genau drei Nutzer in diesem Event gibt. Dabei spielt die Laufzeit oder Beschaffenheit einer einzeln Transaktion keinerlei Rolle mehr, da zuletzt nach Event und nicht nach Zeit gruppiert wird. So werden die normalen Flach-Kurven aus den ersten zwei Sessions, aber auch die Anstiegs-Kurve aus der dritten Session exakt nach Definition umgesetzt. Die einzelnen Ausreißer bei den Flach-Kurven resultieren aus den generierten Spikes, die man bei diesem Diagramm ebenfalls sehr gut erkennen kann. Ebenso ist der immer gleich benutzte Seed für den Zufallsgenerator ersichtlich, welcher dadurch die gleichen Spikes-Parameter erzeugt.

Zusammenfassung Auch wenn man pro Einstellung von `when_active` in der Tenant-Definition eine andere Analyse-Methodik anwenden muss, zeigt sich doch, dass die für dieses einfache Szenario gewählte Intensität geeignet in unterschiedliche Last für das DBMS umgesetzt wird. Damit ist zumindest eine gewisse Sicherheit gegeben, dass sich das entwickelte „MulTeX“-Modell in den Grundzügen auf das *MulTe*-Framework anwenden lässt. Die einzeln definierten Höhen an



(a) Last in Queries pro Sekunde



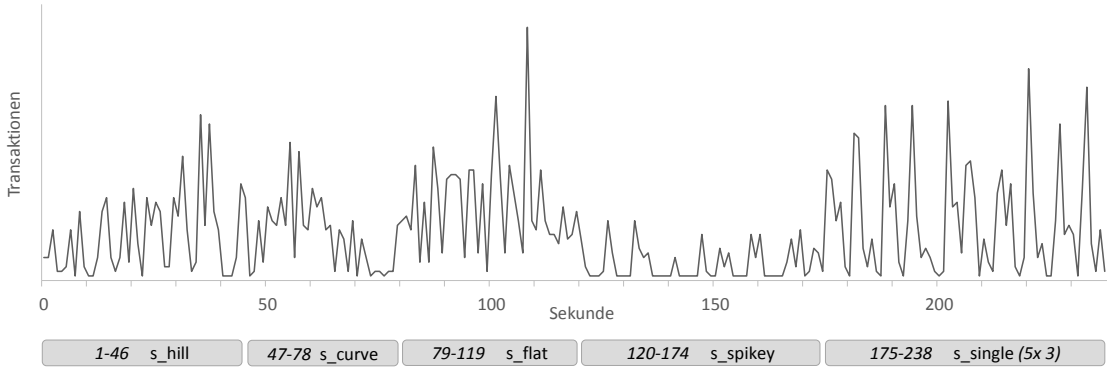
(b) Last in Queries pro Event

Abbildung 5.3: Umgesetzte Intensität in Last im erweiterten Szenario mit zeitlicher Beschränkung

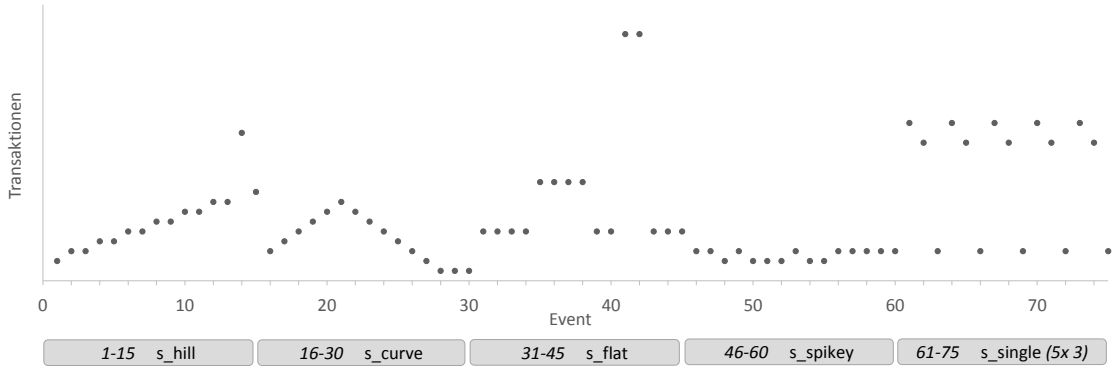
Intensität ließen sich gut voneinander trennen und die Charakteristik der Session konnte ebenfalls nachvollzogen werden. Die nächsten Auswertungen werden zeigen, ob auch feinere Kurven und speziellere Situationen mit der Erweiterung in *MulTe* möglich sind.

Auswertung des erweiterten Szenarios

Zeitlich beschränkte Transaktionen Hier zeigt sich, im Gegensatz zu dem einfachen Szenario, ein etwas anderes Bild auf. Das erweiterte Szenario ist bei einer zeitlichen Beschränkung von fünf Sekunden pro Nutzer getestet worden. Die entsprechenden Diagramme mit den Ergebnissen finden sich in Abbildung 5.3. Leider ist das Ergebnis bei der zeitlichen Beschränkung nicht mehr eindeutig als Output der gegebenen Intensität wiederzuerkennen, und wenn, dann nur durch grobe Übereinstimmungen. Gruppiert nach Transaktionen pro Sekunde, entsteht der Graph 5.3a, welcher nur an einem Punkt genau mit dem eingegebenen Modell übereinstimmt, das Ende von der zweiten Session *s_curve*, wo die Last kurzzeitig auf 0 fällt. Die anderen Session sind, selbst mit dem Wissen über deren theoretisches Aussehen, nicht aus der Lastkurve wiedererkennbar. Eine mögliche Erklärung für dieses Verhalten wäre, dass die Differenzen in den Intensitäten zu gering waren, um sich stark in dem Verlauf der Kurve widerzuspiegeln. Eine andere Erklärung wäre, dass die genutzte Testumgebung nicht den Anforderung für dieses Szenario besaß. Sicher ist jedoch, dass jedes einzelne Event in jeder Session seine Intensität umgesetzt hat, wie es das Modell vorgab, was man an den normalen Debug-Output von *MulTe* während des Tests ablesen konnte. Ein Ausschnitt dieses Logs findet sich im Anhang unter [7.1], wo man den Start einzelner



(a) Last in Queries pro Sekunde



(b) Last in Queries pro Event

Abbildung 5.4: Umgesetzte Intensität in Last im erweiterten Szenario mit absoluter Beschränkung

Sessions und Events erfassen kann, sowie die Ausführungen aller durch das Event gestarteten Nutzer. Warum dies dennoch nicht an dem Transaktionsaufkommen im Diagramm ablesbar ist, müsste separat in einer besser aufgestellten Test-Umgebung nochmal untersucht werden.

Ebenfalls nicht sonderlich gut erkennbar ist die ursprüngliche Intensität im Diagramm 5.3b, welches eine Gruppierung nach Events darstellt. Nach vorheriger Erklärung im einfachen Szenario ist es hier jedoch, im Gegensatz zu der Gruppierung nach Zeit, nicht sonderlich verwunderlich, dass eine Gruppierung nach Events in einer zeitlich begrenzten Ausführung nicht äquivalent wiedergegeben wird, da hier einfach die Laufzeiten der einzelnen verschiedenen Transaktionen sehr stark in die Statistik einspielten und diese verzerren. Auch hier ist der einzige übereinstimme Punkt von Intensität und Last das Ende der zweiten Session.

Absolut beschränkte Transaktionen Bei den absoluten beschränkten Transaktionen bestand, wie auch in den entsprechenden vorherigen Tests, ein Event aus fünf zugesicherten Transaktionen pro vom Event benötigten Nutzer. Die entsprechenden Kurven sind in Abbildung 5.4 dargestellt. Im Gegensatz zu dem letzten Test stellt sich hier der Verlauf des Intenitäts-Modells wieder in der Lastkurve dar, selbst in der Gruppierung nach Zeit (Sekunden) 5.4a. Man kann hier mithilfe der grauen Balken unter dem Diagramm die Laufzeit der einzelnen Sessions ablesen, aber auch ohne diese Balken kann man einigermaßen gut die einzelnen Charakteristiken der Sessions wiedererkennen. Auch hier ist die X-Achse, welche den zeitlichen Verlauf wiedergibt, gestreckt oder gestaucht je nachdem, wie lange die einzelnen Transaktionen benötigten und somit das Event fertig wurde und das nächste Event beginnen konnte. Man kann gut erkennen, wie das Trans-

aktionsaufkommen sich den entsprechenden Tenant-Definitionen anpasst. Während anfangs ein Anstieg, gefolgt von einer Tagesablauf-Kurve, zu erkennen ist, sind auch die größeren und kleineren Unterschiede der letzteren Sessions in der Intensität auch in der Lastkurve sichtbar. Nur der mittlere Teil, genauer gesagt die dritte Session *s_flat* ist hier nicht eindeutig wahrzunehmen, ebenso ist das entstandene Muster aus der vierten Session *s_spikey* nicht sonderlich gut als Rauschen identifizierbar, wenn man nach Zeit gruppiert.

Die nach Event gruppierte Kurve 5.4b ist wie erwartet genau äquivalent der eingehenden Modell-Intensität, da auch hier die eigentliche Anzahl an genutzten Transaktionen pro Event schon vorher berechnet werden können. Man kann hier faktisch jede einzelne Session ohne graue Balken und nur aus dem Kurvenverlauf punktgenau identifizieren. Man kann ebenso genau erfassen, wo einzelnen Spikes über die „Timeseries"-Komponente gelegt wurden.

Zusammenfassung Anders als im einfachem Szenario ist im erweiterten Szenario die Eindeutigkeit der Umsetzung nicht sofort zu erkennen. Das kann an der komplizierteren Konfiguration der Session liegen, oder aber auch an einer nicht ausreichend geeigneten Test-Umgebung. In der Mehrheit der Fälle ist jedoch davon auszugehen, dass die modellierte Nutzer-Intensität korrekt in *MulTe* umgesetzt werden kann. Bei diesem Tests wäre für zukünftige Arbeiten eine erweiterte Analyse sinnvoll, um das volle Potential der *MulTe*-Erweiterung zu erfassen, was im Rahmen dieser Arbeit nicht möglich war.

5.4 EVALUIERUNG DER UMSETZUNG DES MODELLS

Insgesamt hinterlassen die geführten Tests und deren Auswertung eine positive Wirkung. Viele der durchgeführten, auch die abseits der hier dargestellten Tests bestätigten, dass die *MulTe*-Umsetzung des entworfenen „MulTeX"-Modells gut in eine entsprechende Last für das DBMS umgesetzt werden kann. Durch die Beibehaltung der Monotonie-Eigenschaft des Intensitäts-Wertes in *MulTe* ist zudem sichergestellt, dass verschiedenen Intensitäten auch erwartungsgemäß umgesetzt werden können.

Leider konnten nicht alle Tests durchaus positiv gewertet werden, da bislang unbekannte Ursachen für unerwartete Outputs sorgten. Dies benötigt weitere Analysen, um den kompletten Sachverhalt dahinter zu erklären. Mindestens ein Tests ist demzufolge nicht wie erwartet verlaufen, wobei auch dort eine gewisse Grundsicherung gegeben war, dass wenigstens von *MulTe* versucht wurde, den modellierten Nutzer umzusetzen.

Im Vergleich zu einem „An/Aus"-Modell hat die neue Erweiterung in *MulTe* viele Vorteile, hinsichtlich des Modellierens der Workloads. Durch diese neue Erweiterung ist es möglich, Abstufungen und vorausberechnete Modelle im Workload zu hinterlegen, welche durch *MulTe* entsprechend der existierenden Möglichkeiten umgesetzt werden können. Wie die Tests zeigen, resultiert dies direkt, bedingt durch die in 4 gewählte Implementierung, in einer diskreten Kurve von gegen das DBMS abgesetzten Transaktionen pro Intervall. Welche der beiden Methoden für die Intensitäts-Analyse von *MulTe* infrage kommt, entweder *Gruppierung nach Event* oder *Gruppierung nach Zeit*, hängt aber stark von den gewählten Parametern im Workload ab. Für weitere Tests sollte eine erweiterte Umsetzung des „MulTeX"-Modells in *MulTe* in Betracht gezogen wer-

den. Ein Beispiel dafür wäre die Umsetzung des Intensitäts-Wertes nicht nur in parallele Nutzer und Denkpausen, sondern auch in die Beschaffenheit von Queries oder deren Parameter. In Zuge dessen könnten dann erweiterte Tests angewendet werden. Jedoch ist, im Hinblick auf den Fokus dieser Arbeit, die erste und eigentliche Intention erfüllt, das erweiterte Workload-Modell zum besseren und feineren Modellieren eines synthetisierten Nutzers zu benutzen.

6 ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit wurden verschiedene Konzepte von Modellen vorgestellt, um ein Nutzerverhalten in einem Workload einzubringen. Dazu wurde das Konzept der Nutzer-Intensität vorgestellt und die Art des Einflusses, wie die Intensität sich auf den Workload auswirkt. Abgesehen von einfachsten Ansätzen, wie zum Beispiel das „An/Aus"-Modell, ist bisher, nach bestem Wissen des Autors dieser Arbeit, noch nicht direkt ein wie in dieser Arbeit entwickeltes ganzheitliches Nutzermodell für Workloads in einer anderen Arbeit beschrieben worden, auch wenn andere Arbeiten durchaus als Grundlage für das neue Modell dienten. Als Proof-of-Concept wurde eine entsprechende Erweiterung in das Multi-Tenancy Benchmark Framework *MulTe* entwickelt, welche das formale Modell geeignet umsetzt.

Allgemein gab es bisher nur einfachste oder spezialisierte Ansätze, eine Art von Verhalten in den Workload einzubauen, da solche Konzepte nicht im Fokus der Benchmark-Entwickler lagen, ebenso nicht im Fokus der Entwickler von Benchmark-Tools oder -Frameworks wie *MulTe*. Jedoch haben gerade letztere durchaus einen Grund, ein Nutzerverhalten formal in den Workload einzubauen, um den eigentlichen Benchmark-Tests flexibler und realistischer zu gestalten. Diverse Ansätze ergaben sich durch Arbeiten, welche das Modellieren von Spikes oder „Burstiness" beschrieben, damit jedoch nicht weit genug in ein Verhaltensmodell von Tenants eingreifen konnten.

Um die Funktionalität des einfachen „An/Aus"-Modells zu erweitern, wurde sich an verschiedenen Modellierungsmethoden der Mathematik, Statistik sowie anderen IT-Arbeiten bedient, welche die grundlegende Idee hinter dem neuen Modell darstellen. Es wurden davon einige geeignete Konzepte gewählt und dahingehend vereinheitlicht, dass nur noch wenige, dafür aussagekräftige Parameter für eine Konfiguration einer Intensitäts-Kurve notwendig sind. Durch die Kombination dieser verschiedenen Methoden kann mit dem neuen Modell nun aus einem umfangreichen Katalog von Kurvenverläufen und Nutzer-Szenarien gewählt werden. Das erhöht den Mehrwert von *MulTe* beträchtlich, da eine Vielzahl an nun möglichen Situationen dadurch darstellbar ist.

Zuletzt wurde das in *MulTe* umgesetzte Modell mit verschieden konfigurierten Modell-Szenarien

getestet und ausgewertet. Dabei konnte in vielen Teilen eine nachvollziehbare Umsetzung des Modells wahrgenommen werden, womit die Prototyp-Entwicklung beendet wäre und an einer optimierten Version der Erweiterung gearbeitet werden kann. Allerdings wären hier weitere Analysen mit besseren Test-Umgebungen und -Voraussetzungen von Vorteil, um etwaige Umsetzungs-Fehler des Modells in *MulTe* zu korrigieren und daraus resultierend die gesamte Umsetzung der Erweiterung im *MulTe* zu verbessern.

Des weiteren besteht noch Optimierungsbedarf in *MulTe*-Erweiterung bei diversen Implementationen. So könnte die Performance zwischen dem Umschalten der nacheinander folgenden Sessions und Events verbessert werden, um die dafür notwendige, technisch bedingte, Pausenzeit von einigen Sekunden so gut wie möglich zu entfernen. Dies könnte durchaus in einer Zusammenlegung von nacheinander folgenden Sessions und Events resultieren oder durch Einsparung einiger Routinen im Ablauf (zum Beispiel das Beenden und Neustarten des User-Pools). Ebenso denkbar wäre eine erweiterte dynamischere Anpassung der Parameter zur Laufzeit, hier sei besonders der Zufallsgenerator genannt, um bei Wiederholungen einer Session variabel zu bleiben. Weiterhin wäre eine komplette Einbindung in den *MulTe*-Workflow sinnvoll, inklusive aller vorherigen Python-Komponenten und Template-Generierungen.

Zukünftige Arbeiten an dieser Thematik könnten sich mit dem funktionalen Ausbau der neuen *MulTe*-Erweiterung beschäftigen. Es wäre denkbar, neue Konzepte in das neue Modell zu übernehmen, wie zum Beispiel die in der Arbeit [2] genannten Daten-Spikes durch Definition von „Hot Objects“. Aber auch derzeit existierende Konzepte könnten verbessert werden, zum Beispiel eine bessere Parametrisierung von Spikes im Workload. Auch eine GUI-Entwicklung mit einer Echtzeitberechnung des durch den Benutzer eingegebene Session-Konfiguration beschriebenen Modells wäre eine große Verbesserung der neuen *MulTe*-Erweiterung und würde die Benutzbarkeit dieser enorm steigern. Neben den Verbesserungen für dieses Modell wären auch weitere Modelle für *MulTe* denkbar, welche nebeneinander existieren und dem Benutzer die Auswahl lassen, welcher Tenant welches Modell nutzen soll.

7 ANHANG

Folgendes [7.1] stellt einen gekürzten Log-Auszug aus dem originalem *MulTe* dar, enstanden mit Debug-Ausgaben. Man kann erkennen, dass alle notwendigen Infos anfangs geparst werden ([7.1]: Zeile 1 - 16). Es folgen Ausschnitte von den Starts der Session *s_hill* sowie *s_curve*, mit jeweils zugeordneten Events 1 bis 15, welche wiederum selbst Transaktionen an das DBMS absetzen.

```
1  () - workload found
2  () - initialize WorkloadParameters for workload
3  () - workload name: tpch query 2
4  () - database: tenant_1000
5  () - parallel users: 0
6  () - seconds active, when on: 5
7  () - mean sleep time: 0
8  () - think time: 0
9  () - set session
10 () - set session
11 () - set session
12 () - set session
13 () - set session
14 () - reading input files containing possible parameter values
15 () - Create output directory
16 () - Run workloads for  30 minute(s)
17 () - ------------------------------------------------------------
18 () - ------------------------------------------------------------
19 () - start new model round, -1 rounds left
20 () - ####### start processing session s_hill with 15 slots #######
21 () - => calculated intensity: 2.0 [2.000 ; 0.000]
22 () - => calculated intensity: 2.5 [2.500 ; 0.000]
23 () - => calculated intensity: 3.0 [3.000 ; 0.000]
24 () - => calculated intensity: 3.5 [3.500 ; 0.000]
25 () - => calculated intensity: 4.0 [4.000 ; 0.000]
```

```
() - => calculated intensity: 4.5 [4.500 ; 0.000]
() - => calculated intensity: 5.0 [5.000 ; 0.000]
() - => calculated intensity: 5.5 [5.500 ; 0.000]
() - => calculated intensity: 6.0 [6.000 ; 0.000]
() - => calculated intensity: 6.5 [6.500 ; 0.000]
() - => calculated intensity: 7.0 [7.000 ; 0.000]
() - => calculated intensity: 7.5 [7.500 ; 0.000]
() - => calculated intensity: 8.0 [8.000 ; 0.000]
() - => calculated intensity: 14.138  [8.500 ; 5.638]
() - => calculated intensity: 9.0 [9.000 ; 0.000]
() - start new round for this session: 1/1
() - creating 15 normal users
() - start all slots (events)
() - ------------- start users for s_hill_event1 [intensity: 2.0]
() - establish database connection
() - establish database connection
() - prepare statements
() - prepare statements
() - unpause s_hill_user(2) and continue executing
() - unpause s_hill_user(1) and continue executing
() - execute workload, bound by time (5s) with thinktime (0ms)
() - execute workload, bound by time (5s) with thinktime (0ms)
() - terminate concurrent user, because time is up
() - terminate concurrent user, because time is up
() - ------------- start users for s_hill_event2 [intensity: 2.5]
() - establish database connection
() - prepare statements
() - unpause s_hill_user(3) and continue executing
() - unpause s_hill_user(2) and continue executing
() - unpause s_hill_user(1) and continue executing
() - execute workload, bound by time (5s) with thinktime (0ms)
() - execute workload, bound by time (5s) with thinktime (500ms)
() - execute workload, bound by time (5s) with thinktime (0ms)
() - terminate concurrent user, because time is up
() - terminate concurrent user, because time is up
() - terminate concurrent user, because time is up
() - ------------- start users for s_hill_event3 [intensity: 3.0]
() - unpause s_hill_user(2) and continue executing
() - unpause s_hill_user(1) and continue executing
() - unpause s_hill_user(3) and continue executing
() - execute workload, bound by time (5s) with thinktime (0ms)
() - execute workload, bound by time (5s) with thinktime (0ms)
() - execute workload, bound by time (5s) with thinktime (0ms)
() - terminate concurrent user, because time is up
() - terminate concurrent user, because time is up
() - terminate concurrent user, because time is up
```

```
[...]

() - terminate all the 15 old users and 1 idle user at session end
() - sleep for ~10 seconds and continue with sessions afterwards
() - ####### start processing session s_curve with 15 slots ######
() - => calculated intensity: 3.0 [3.000 ; 0.000]
() - => calculated intensity: 4.0 [4.000 ; 0.000]
() - => calculated intensity: 5.0 [5.000 ; 0.000]
() - => calculated intensity: 6.0 [6.000 ; 0.000]
() - => calculated intensity: 7.0 [7.000 ; 0.000]
() - => calculated intensity: 8.0 [8.000 ; 0.000]
() - => calculated intensity: 7.0 [7.000 ; 0.000]
() - => calculated intensity: 6.0 [6.000 ; 0.000]
() - => calculated intensity: 5.0 [5.000 ; 0.000]
() - => calculated intensity: 4.0 [4.000 ; 0.000]
() - => calculated intensity: 3.0 [3.000 ; 0.000]
() - => calculated intensity: 2.0 [2.000 ; 0.000]
() - => calculated intensity: 1.0 [1.000 ; 0.000]
() - => calculated intensity: 0.0 [0.000 ; 0.000]
() - => calculated intensity: 0.0 [-1.000 ; 0.000]
() - start new round for this session: 1/1
() - creating 8 normal users
() - start all slots (events)
() - ------------- start users for s_curve_event1 [intensity: 3.0]
() - establish database connection
() - establish database connection
() - establish database connection
() - prepare statements
() - unpause s_curve_user(2) and continue executing
() - prepare statements
() - unpause s_curve_user(3) and continue executing
() - prepare statements
() - unpause s_curve_user(1) and continue executing
() - execute workload, bound by time (5s) with thinktime (0ms)
() - execute workload, bound by time (5s) with thinktime (0ms)
() - execute workload, bound by time (5s) with thinktime (0ms)
() - terminate concurrent user, because time is up
() - terminate concurrent user, because time is up
() - terminate concurrent user, because time is up
() - ------------- start users for s_curve_event2 [intensity: 4.0]

[...]
```

Quellcode 7.1: Ein Log des originalen Outputs *MulTe* für das erweiterte Szenario

LITERATURVERZEICHNIS

[1] TPC-H Homepage. `http://www.tpc.org/tpch/`. [Online; Mai 2015].

[2] Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In Proceedings of the 1st ACM symposium on Cloud computing, pages 241–252, 2010.

[3] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In SIGMOD '11 Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, pages 313–324, 2011.

[4] R. Gusella. Characterizing the variability of arrival processes with indexes of dispersion. In Selected Areas in Communications, pages 203–211, 1991.

[5] Prajakta S. Kalekar. Time series forecasting using holt-winters exponential smoothing. 2004.

[6] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. Multe: A multi-tenancy database benchmark framework. In Selected Topics in Performance Evaluation and Benchmarking, pages 92–107, 2012.

[7] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. Multe: A multi-tenancy database benchmark framework — implementation details. Technical report, Technische Universität Dresden, 2012.

[8] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In SIGCOMM '04 Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, pages 219–230, 2004.

[9] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In ICAC '09 Proceedings of the 6th international conference on Autonomic computing, pages 149–158, 2009.

[10] Venkata N. Padmanabhan and Lili Qiu. The content and access dynamics of a busy web server: Findings and implications. Technical report, Microsoft Research, 2000.

ABBILDUNGSVERZEICHNIS

1.1 Zusammenhänge im Benchmark-Umfeld 11

2.1 Zusammenhänge im Multi-Tenancy Benchmark-Umfeld 15

2.2 Workflow von MulTe (aus dem original Paper) 17

2.3 Darstellbares Nutzerverhalten durch den bisherigen *MulTe*-Workload 18

3.1 Beispiel-Kurve für ein simples „An/Aus“-Modell 26

3.2 Beispiel-Kurve für ein „Function“-Modell 27

3.3 Beispiel-Kurve für ein „Timeseries“-Modell 27

3.4 Beispiel-Kurven für ausgewählte Parameter im „MulTeX“-Modell 30

4.1 Funktionsweise des neuen *MulTe*-Workload durch Intensitäts-Modell 32

4.2 Diskretisierung der Intensitäts-Kurve durch Sessions und Events 34

4.3 Umlenkung des Ablaufes in *MulTe*-Workflow der Engine 37

4.4 Erweiterung der Engine für das Interpretieren des neuen Modells 38

5.1 Umgesetzte Intensität in Last im einfachen Szenario mit zeitlicher Beschränkung . 50

5.2 Umgesetzte Intensität in Last im einfachen Szenario mit absoluter Beschränkung . 51

5.3 Umgesetzte Intensität in Last im erweiterten Szenario mit zeitlicher Beschränkung 52

5.4 Umgesetzte Intensität in Last im erweiterten Szenario mit absoluter Beschränkung 53

TABELLENVERZEICHNIS

2.1 Erklärung der Begrifflichkeiten 14

2.2 Parameter-Spezifizierung im originalen *MulTe*-Tenant-Definitionen 18

4.1 Parameter-Spezifizierung im neuen *MulTe*-XML-Session-Knoten 36

5.1 Log zur Bestimmung der Intensitäten und der Evaluierung des Modells 43

5.2 Log mit Gruppierung nach Spalte und Häufigkeit des Auftretens 44

5.3 Sessions, Slots und Events des einfachen Szenarios 48

5.4 Sessions, Slots und Events des erweiterten Szenarios 48

QUELLCODEVERZEICHNIS

4.1 Einfache Tenant-Definition 33

4.2 XML-Knoten für die Modell-Definition 35

4.3 Einfacher Algorithmus zur Konvertierung von Konfiguration 39

4.4 Eine Ausführung von Sleep wenn ein Nutzer inaktiv ist 42

5.1 Tenant-Definitions-Header 45

5.2 Tenant-Definition (Session-Knoten) des einfachen Test-Szenarios 46

5.3 Tenant-Definition (Session-Knoten) des erweiterten Test-Szenarios 47

7.1 Ein Log des originalen Outputs *MulTe* für das erweiterte Szenario 59