

Application Performance Management

Experiment-Design & Benchmarking

Michael Faes

Übersicht Woche 3

1. Übungsbesprechung
2. Performance-Experimente: Grundlagen
3. Experiment-Design
4. Übung: *Benchmarking mit JMH*

Performance-Experimente

Beobachtung und Experiment

Rückblick: Untersuchen von Unbekanntem durch *Scientific Method*



1. Frage
2. Hypothese
3. Vorhersage
4. **Test:** *Beobachtung* oder *Experiment*
5. Auswertung

Beobachtung und *Experiment* sind zwei grundsätzliche Ansätze, Performance zu analysieren. Beide haben Vor- und Nachteile, bzw. unterschiedliche Use Cases.

Beispiel Beobachtung (letzte Woche): Hypothese: Grund für schlechte Performance ist Dateisystem-Cache. Test: *Messen der Cache-Misses*.

Beispiel Experiment:

1. Frage: Warum dauern HTTP-Requests länger von Host A zu Host C als von Host B zu Host C?
2. Hypothese: Grund: A und B sind in unterschiedlichen Datenzentren
3. Vorhersage: Verschieben von Host A in Datenzentrum von Host B behebt Problem.
4. Test: *Host A verschieben* und Anfragezeit messen
5. Analyse: Requests dauern nicht mehr länger – Problem gefunden!

Alternative mit Beobachtung:

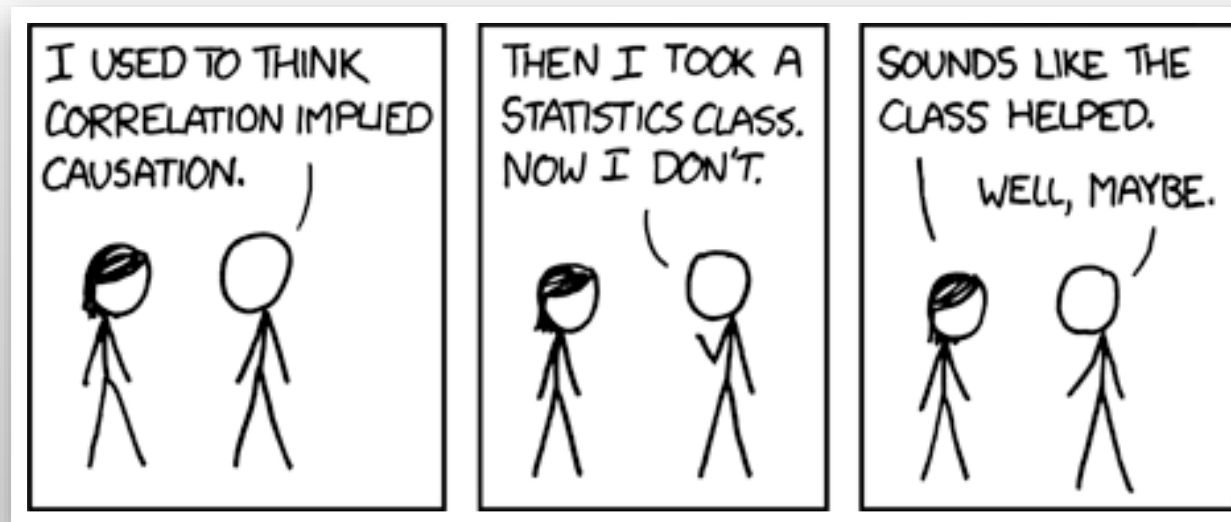
3. Vorhersage: Anfragen von anderen Hosts in Datenzentrum von A dauern ebenfalls länger
4. Test: *Anfragezeit auf anderen Hosts messen*

Korrelation vs. Kausalität

Hauptvorteil von Experiment: Starker Hinweis, dass gefundener Zusammenhang wirklich der Grund für beobachtetes Verhalten ist.

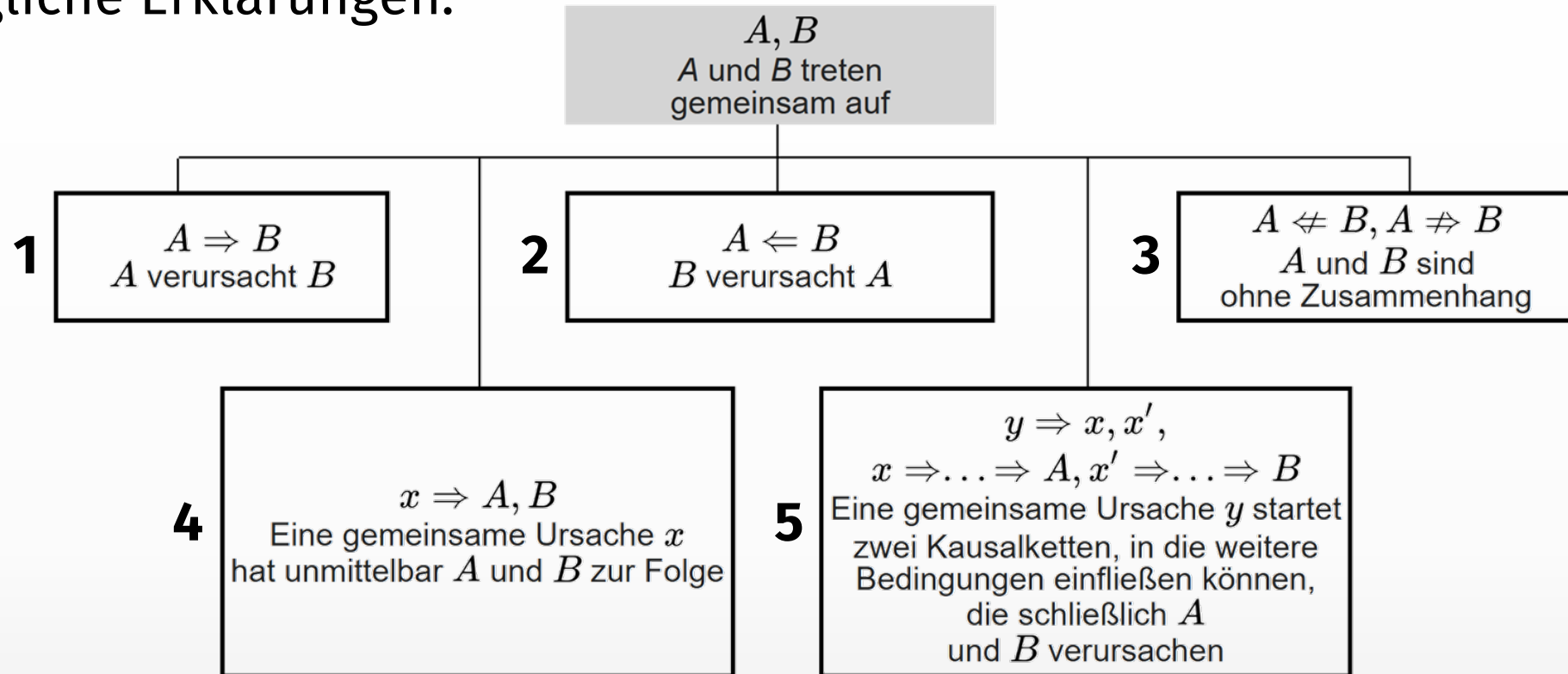
Korrelation: Wenn man A beobachtet, beobachtet man auch B.

Kausalität: A verursacht B.



[xkcd: Correlation](#) (CC BY-NC 2.5)

Mögliche Erklärungen:



https://de.wikipedia.org/wiki/Cum_hoc_ergo_propter_hoc

Meist kann man zumindest **2** ausschliessen, aber zwischen **1, 3, 4 & 5** unterscheiden ist schwierig durch reine Beobachtung.

Experiment: Beobachten nicht nur A und B, sondern **steuern A!**

Performance: Beobachtung vs. Experiment

Vorteile Experiment

- Kontrolliertes Ändern von Parametern möglich
- Erlaubt zuverlässige Schlüsse über Kausalität
- Kann optimalen Wert für Parameter bestimmen
- Ermöglicht Vorhersagen (Was wäre, wenn...?)
- Ist nicht auf Users/reale Last angewiesen

Vorteile Beobachtung

- Keine grosse Beeinflussung von Live-System
- Liefert Daten unter realen Bedingungen
- Users/Last müssen nicht modelliert/simuliert werden
- Ist oft weniger aufwändig
- Kann (besser) automatisiert werden

Experiment-Design

(deutsch: «Statistische Versuchsplanung»...)

Experiment-Design

Experiment-Design:

Welche Experimente, wie oft, in welcher Reihenfolge?

Wenn System *völlig unbekannt*, dann wieder Scientific Method:



Frage, Hypothese & Vorhersage

Passendes Experiment entwerfen, durchführen & auswerten

Oft *nicht effizient*: Wissen/vermuten schon gewisse Dinge über System, z. B. dass CPU & Speicher Einfluss auf Performance haben.

Ziel von Experiment-Design: Maximale Menge an Information mit minimalem Aufwand.


Ziele


Wichtigster Schritt am Anfang jeder Analyse: **Klare Ziele/Fragen.**

Beispiel: Auswählen von SWITCHengine für App

Flavor	CPUs	RAM	Disk
m1.tiny	1	512MB	1GB
m1.small	1	2GB	20GB
m1.medium	2	4GB	40GB
m1.large	4	8GB	80GB
m1.xlarge	8	16GB	160GB

<https://www.switch.ch/de/engines/techspecs/>

 «Welche VM-Konfiguration ist für unsere App *am geeignetsten?*»

 «Welche VM-Konfiguration führt dazu, dass die *häufigsten 80% der Anfrage-Arten* an unsere App die *kürzeste durchschnittliche Antwortzeit* haben?
Und welche anderen Konfig. führen zu einer *max. 20% höheren Antwortzeit?*»

 «Welche VMs führen für [...] zu einer *durch. Antwortzeit von weniger als 100 ms?*»

Begriffe

Antwortvariable (*response variable*): Resultat des Experiments, bzw. der Teil davon, der gemessen wird.

Im Beispiel: durchschnittliche Antwortzeit

Faktoren: Variablen, welche die Antwortvariable beeinflussen könnten.

Im Beispiel: Anzahl CPUs, RAM-Grösse, Disk-Grösse, Anfrage-Art (!)

Stufen (*levels*): Werte, die ein bestimmter Faktor annehmen kann.

Im Beispiel: CPU: 1, 2, 4, 8; RAM: 512 MB, 2 GB, 4 GB, 8 GB, 16 GB;

Disk: 1 GB, 20 GB, 40 GB, 80 GB, 160 GB, Anfrage-Arten: ... (12 Stück)

Replikation: Wiederholung von einigen oder allen Experimenten.

Beispiel: Messungen auf VMs könnten 3 mal wiederholt werden.

Designs

Design (*Versuchsplan*): Besteht aus: Anzahl der Experimente, Definition des Levels für jeden Faktor für jedes Experiment, Anzahl Replikationen für jedes Experiment.

Beispiele:

- Alle Kombinationen von Levels der 4 Faktoren testen und alle Experimente 10× replizieren:

$$4 \times 5 \times 5 \times 12 \times 10 = 12'000 \text{ Experimente...}$$

- Könnten «*typische*» *Konfiguration* auswählen, jeden Faktoren separat variieren. Experimente 3× replizieren:

$$(1 + (4-1) + (5-1) + (5-1) + (12-1)) \times 3 = 69 \text{ Experimente}$$

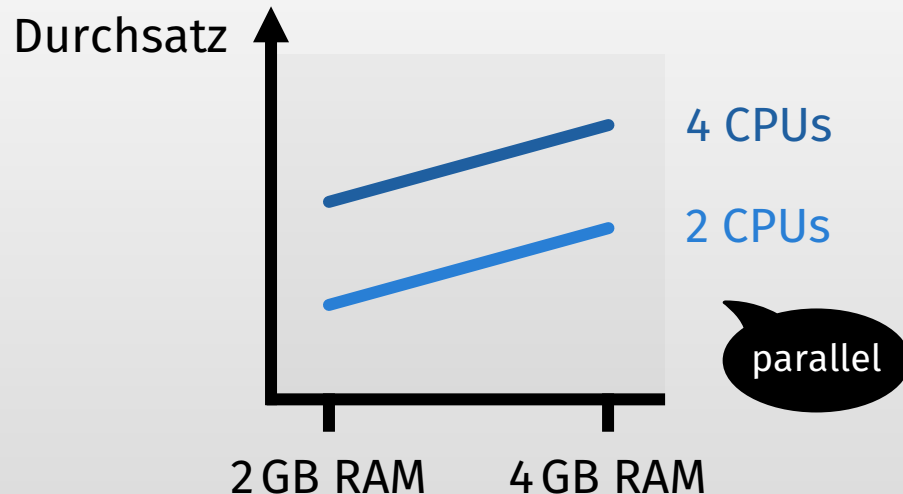
• ...

Interaktion

Interaktion: Zwei Faktoren *interagieren*, falls der Effekt des einen vom Level des anderen abhängig ist.

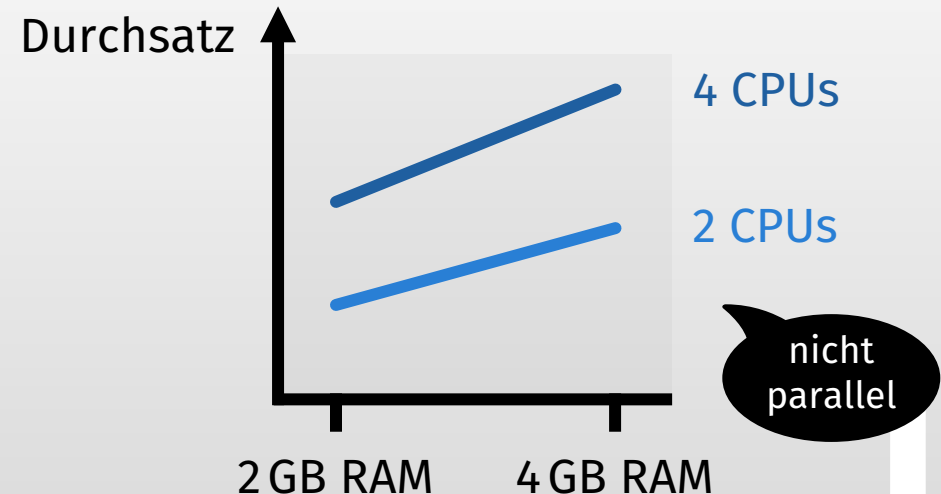
keine Interaktion

	2 GB RAM	4 GB RAM
2 CPUs	100	150
4 CPUs	200	250



Interaktion

	2 GB RAM	4 GB RAM
2 CPUs	100	150
4 CPUs	200	300



Typen von Designs

«**Einen Faktor aufs Mal**»: Wählen «typische» Konfiguration, variieren einen Faktor aufs Mal um dessen Einfluss auf Performance zu sehen.

Anzahl Experimente (ohne Replikation):

$$n = 1 + \sum_{i=1}^k (l_i - 1)$$

n Anzahl Experimente

k Anzahl Faktoren

l_i Anzahl Levels für Faktor i

Nachteile:

- Interaktionen zwischen Faktoren werden nicht berücksichtigt. Falls vorhanden, liefern Experimente falsche Schlüsse.
- Statistisch nicht effizient: Mit gleicher Anzahl von Experimenten könnte mehr Information gewonnen werden. (...)

Full Factorial Design (*Vollständiger Versuchsplan*): Alle möglichen Kombinationen von allen Levels aller Faktoren.

Anzahl Experimente (ohne Replikation):

$$n = \prod_{i=1}^k l_i$$

n Anzahl Experimente

k Anzahl Faktoren

l_i Anzahl Levels für Faktor i

Vorteil: Kann Effekte *aller Faktoren* bestimmen und *alle Interaktionen* (sogar zwischen mehr als zwei Faktoren)

Nachteil: Anzahl Experimente explodiert für grosse Zahl von Faktoren.

Möglichkeiten zur Reduktion:

- Anzahl Faktoren reduzieren
- **Anzahl Levels pro Faktor reduzieren** (...)
- Abgewandeltes Design: *Fractional Factorial Design*

Fractional Factorial Design (*Teilfaktorplan*): Statt *allen* möglichen werden nur *gewisse Kombinationen von Levels* getestet.

Beispiel: VM-Konfigs (vereinf.) ohne Anfrage-Art und ohne Replikation

Full design:

4 CPU-Levels × 4 RAM-Levels × 4 Disk-Levels = 64 Experimente

Fractional design (Beispiel):

Experiment	CPUs	RAM	Disk
1	1	2 GB	20 GB
2	1	4 GB	40 GB
3	2	8 GB	20 GB
4	2	16 GB	40 GB
5	4	2 GB	80 GB
6	4	4 GB	160 GB
7	8	8 GB	80 GB
8	8	16 GB	160 GB

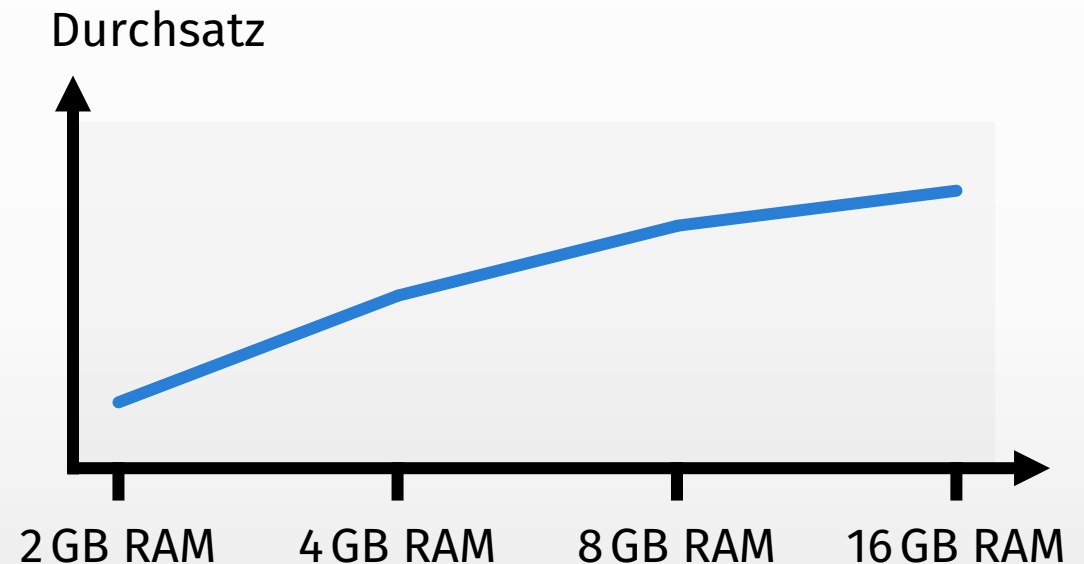
8 Experimente

2^k -Design

Spezialfall von Full Factorial Design: Jeder Faktor hat genau 2 Levels.

Anzahl Experimente: 2^k .

Idee: Wissen oft, dass Einfluss von Faktoren *unidirektional* ist: Durch Erhöhen von Faktor wird Performance monoton besser (oder schlechter).



Mögliches Vorgehen: Zu Beginn einer Analyse nur zwei Levels für jeden Faktor: Minimum und Maximum. Dann entscheiden, ob Performance-Unterschied genügend gross ist, um Faktor weiter zu beachten.

2^k-Design: Effekt-Analyse

Einfachstes Beispiel:

2²-Design ohne Replikation

	2 GB RAM	4 GB RAM
2 CPUs	96	143
4 CPUs	209	296

Durchsatz (in Anfragen/s)

Einfaches Modell: Durchsatz ist *linear abhängig* von Anzahl CPUs, von RAM-Grösse und von Interaktion (Produkt) der beiden.

$$x_C = \begin{cases} -1 & \text{falls 2 CPUs} \\ 1 & \text{falls 4 CPUs} \end{cases}$$

$$x_R = \begin{cases} -1 & \text{falls 2 GB RAM} \\ 1 & \text{falls 4 GB RAM} \end{cases}$$

$$y = q_0 + q_C x_C + q_R x_R + q_{CR} x_C x_R$$

q_0 Durchschnittl. Durchsatz

q_C Effekt von Anzahl CPUs

q_R Effekt von RAM-Grösse

q_{CR} Interaktion von CPU/RAM

Können Messresultate, x_C und x_R in Gleichung einsetzen:

$$\begin{aligned}96 &= q_0 - q_C - q_R + q_{CR} \\209 &= q_0 + q_C - q_R - q_{CR} \\143 &= q_0 - q_C + q_R - q_{CR} \\296 &= q_0 + q_C + q_R + q_{CR}\end{aligned}$$

	2 GB RAM	4 GB RAM
2 CPUs	96	143
4 CPUs	209	296

Auflösen nach q_0 , q_C , q_R und q_{CR} ergibt:

$$y = 186 + 66.5x_C + 33.5x_R + 10x_Cx_R$$

Bedeutet:

- Durchschnittlicher Durchsatz ist 186 Anfragen/s
- Effekt von 4 CPUs gegenüber 2 CPUs ist 66.5 Anfragen/s
- Effekt von 2 GB RAM gegenüber 4 GB RAM ist 33.5 Anfragen/s
- Interaktion zwischen CPU und RAM macht 10 Anfragen/s aus

Vorzeichentabelle

Statt von Hand Gleichungssystem auflösen: Trick mit *Vorzeichentabelle*

\emptyset	CPU	RAM	CPU & RAM	Durchsatz
1	-1	-1	1	96
1	1	-1	-1	209
1	-1	1	-1	143
1	1	1	1	296
744	266	134	40	Total
186	66.5	33.5	10	Total/4

Effekt von Anzahl CPUs: *Durchschnittlicher Unterschied* zwischen Durchsatz mit 2 CPUs und mit 4 CPUs. Analog für RAM.

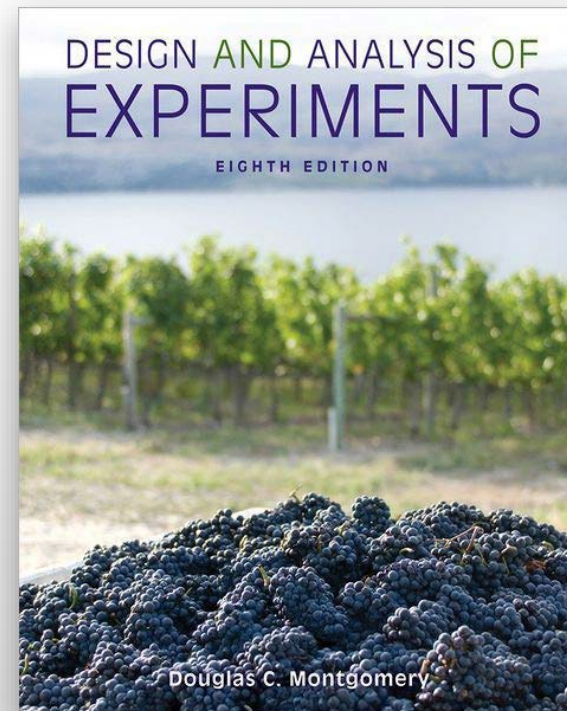
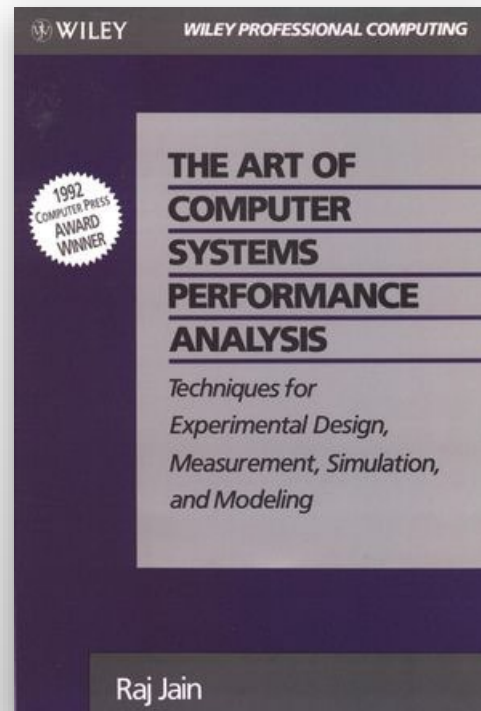
Mit Replikationen: in letzter Spalte einfach *Durchschnittswerte* für entsprechende Experimente einsetzen.

Experiment-Design: Details

Vorsicht: Nicht alle Effekte sind *statistisch signifikant*!

Echte Performance-Studie (z. B. für BA) sollte auch *Varianz durch Messfehler* berücksichtigen. «Fortgeschrittene» statistische Verfahren.

Weitere Details
z. B. in:



einfach online
zu finden...

Benchmarking mit JMH

Benchmarking

Benchmarking: Kontrolliertes Messen von Performance unter definierter Last, Konfiguration, usw.

Ziele:

- Vergleichen von Alternativen (Systeme, Konfigs, Code-Stücke, ...)
- Identifizieren von Performance-Rückschritten (*Regressions*)
- Verstehen von Performance-Grenzen

Arten von Benchmarks:

- *Microbenchmarks*: Messen von einzelnen oder kleinen Serien von Operationen, wie Öffnen einer Datei, Ausführen einer Methode, ...
- *Macrobenchmarks*: Messen der Performance von ganzen Applikationen oder Sammlungen davon

Herausforderungen beim Benchmarking

Herausforderungen beim Benchmarking von (Java-)Code:

- *Variation*: Performance hängt von unkontrollierbaren Umständen ab, die von Ausführung zu Ausführung ändern können.
- Verzerrung der Resultate durch *Overhead der Mess-Infrastruktur*
- *Warmup*: JIT-Compiler optimiert (oder überhaupt kompiliert) Code nicht von Anfang an. Erste n Ausführungen nicht repräsentativ.
- *Ungewollte Optimierungen*: JIT-Compiler könnte Code «zu stark» optimieren, z. B. «ungebrauchte» Teile wegoptimieren.
- *Vermischen von Code-Profilen*: JIT-Compiler optimiert basierend auf bisher gesehener Ausführung (Profiling!). Mehrere Benchmarks innerhalb der gleichen JVM führen zu Vermischung und Interaktion.

Java Microbenchmark Harness

Werkzeug, das diese Herausforderungen berücksichtigt:

Java Microbenchmark Harness (JMH)

Geeignet für alle JVM-Sprachen (Java, Scala, Kotlin, ...) und alle möglichen Benchmarks, von Nano- bis Macro.

Idee: Benchmarks sind Java-Klassen und Teil der Code Base.
Konfiguration über Annotationen, ähnlich wie bei JUnit.

Mehrwert gegenüber *hand-rolled* (z. B. `DocFinderPerfTester`):

- Weniger Code, weniger Fehler
- Eingebaute einfache statistische Analyse
- **Vor allem:** Berücksichtigt Warmup, ungewollte Optimierung, usw.

Übung: Benchmarking mit JMH

Fragen?

