

Big Data Engineering

Konstruktion datenintensiver Systeme

christian.zirpins@hs-karlsruhe.de

Big Data Paradigma



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES



Einordnung in der Fakultät IWI (Informatik)

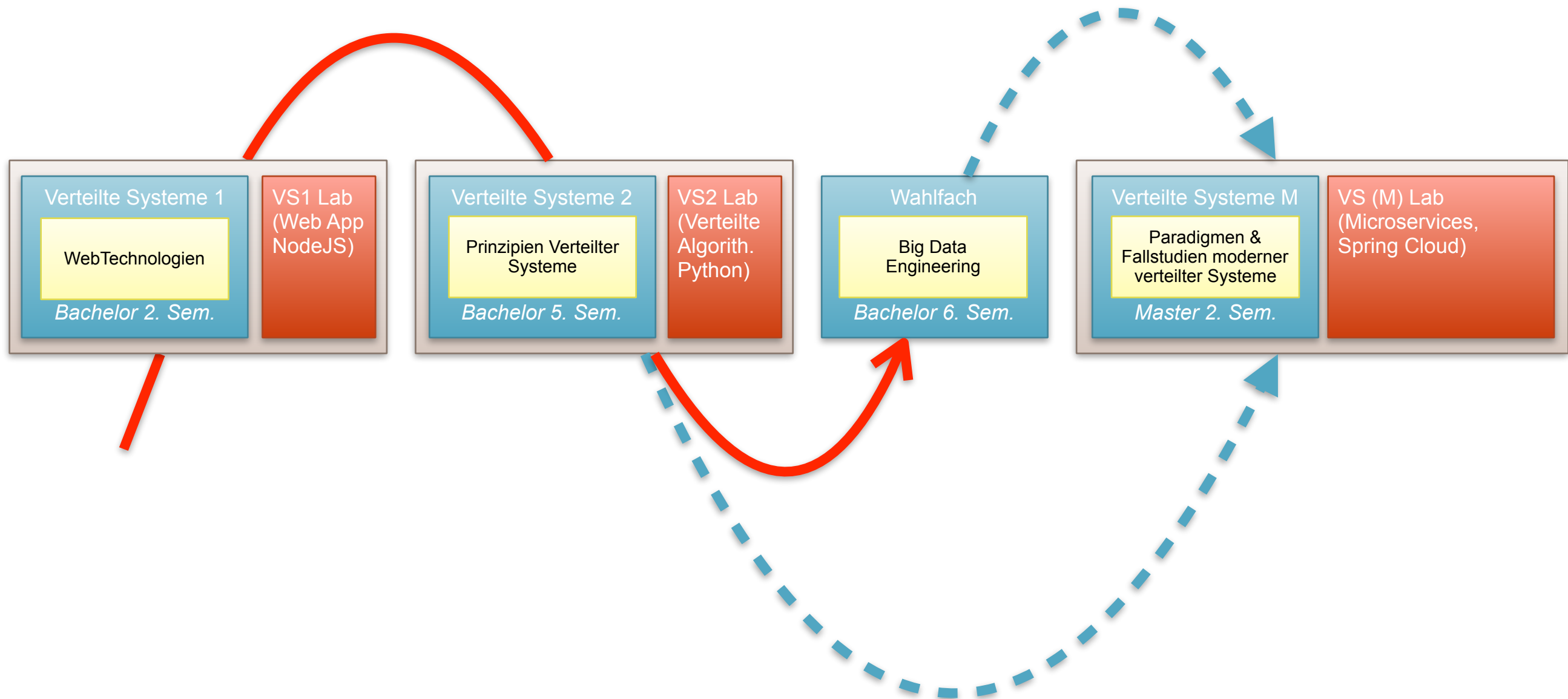
Dozent: Prof. Dr. rer. nat. Dipl.-Inf. **Christian Zirpins**

- Seit 2015 Professor für verteilte Systeme an der HsKA

Bereich Verteilte Systeme (VSYS)

- Schwerpunkte in Forschung und Lehre:
 - *Web Engineering*
 - *Datenintensive Systeme ("Big Data")*
 - *Service Computing ("Microservices")*
- Mit Bezügen zu...
 - *Cloud Computing, Internet of Things, Smart Systems u.a.*

Lehrpfade "Verteilte Systeme"



Mit den Worten des WEF

Nach dieser Einführung können Sie...

- den **Big Data** Begriff **einordnen**.
- **Fallstricke der Skalierung** von Datenbanken **verstehen**.
- die **Grundprinzipien von Datensystemen** **benennen**.
- die **Eigenschaften von Datensystemen** **bewerten**.
- das Paradigma der **Lambda Architektur** **erklären**.
- existierende **Big Data Tools** **klassifizieren**.

Was ist Big Data?

Viel (Volume)

Anzahl von Datensätzen und Dateien

Zettabytes | Yottabytes | Exabytes |
Petabytes | Terabytes

Divers (Variety)

- Fremddaten (Web etc.)

- Firmendaten

(unstrukturiert, semistrukturiert, strukturiert)

Slides | Texte | Video | Bilder | Tweets | Logs |
Kommunikation zwischen Maschinen

Big Data

Schnell (Velocity)

- Schnelle Datengenerierung

- Permanente Datenströme

Echtzeit | Milliseconds | Seconds | Minutes
Hours

Analytics

Erkennung von Zusammenhängen,
Bedeutungen, Mustern:

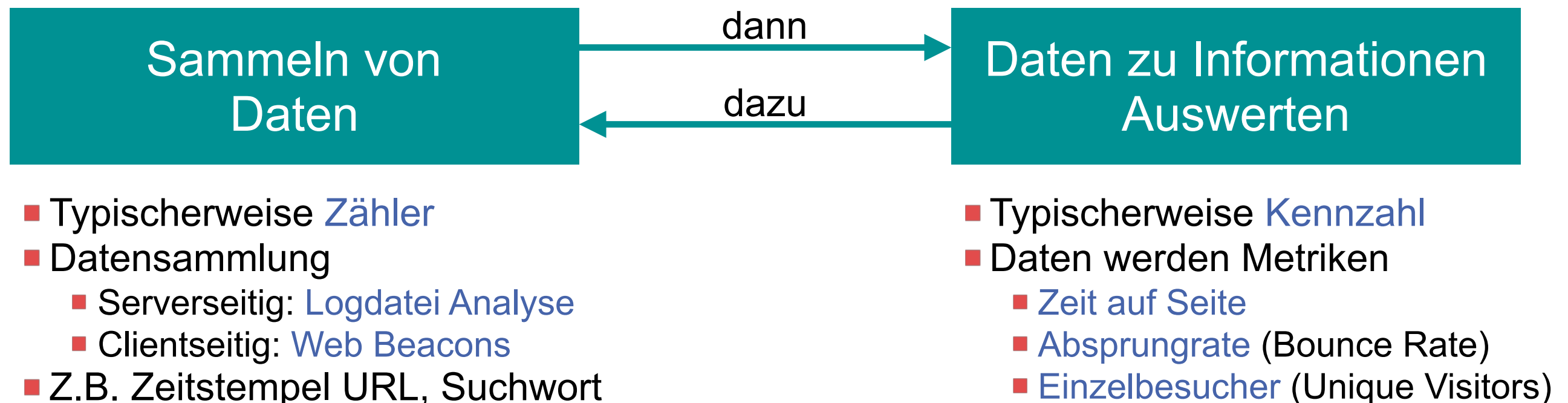
Vorhersagemodelle, Data Mining, Text
Mining, Bildanalytik, Visualisierung, Realtime

Gartner (2012): *"Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization."*

Ein Beispiel: Web Analytics


Web Analytics aka Clickstream-Analyse, Datenverkehrsanalyse, Traffic-Analyse, Web-Analyse, Web-Controlling, Webtracking

*Sammlung von Daten und deren Auswertung bzgl. Verhalten von Besuchern auf Websites. Ein **Analytic-Tool** untersucht typischerweise, woher die Besucher kommen, welche Bereiche auf einer Internetseite aufgesucht werden und wie oft und wie lange welche Unterseiten und Kategorien angesehen werden. (wikipedia)*



SuperWebAnalytics.com

- **SuperWebAnalytics** wird unser Beispiel (ähnlich Google Analytics)
 - Ziel: *Milliarden Pageviews pro Tag verfolgen*
- Wir wollen diverse Metriken in Echtzeit unterstützen; von einfachen Zählern zu komplexen Analysen der Navigation:
 - Pageviews pro URL in Zeitintervallen
 - Einzelbesucher (Unique Visitors) pro URL in Zeitintervallen
 - Bounce-Rate Analyse



Wieviele
Einzelpersonen
haben die Domain in
2010 besucht?

Wieviele Perso-
nen besuchen nur
eine Seite?

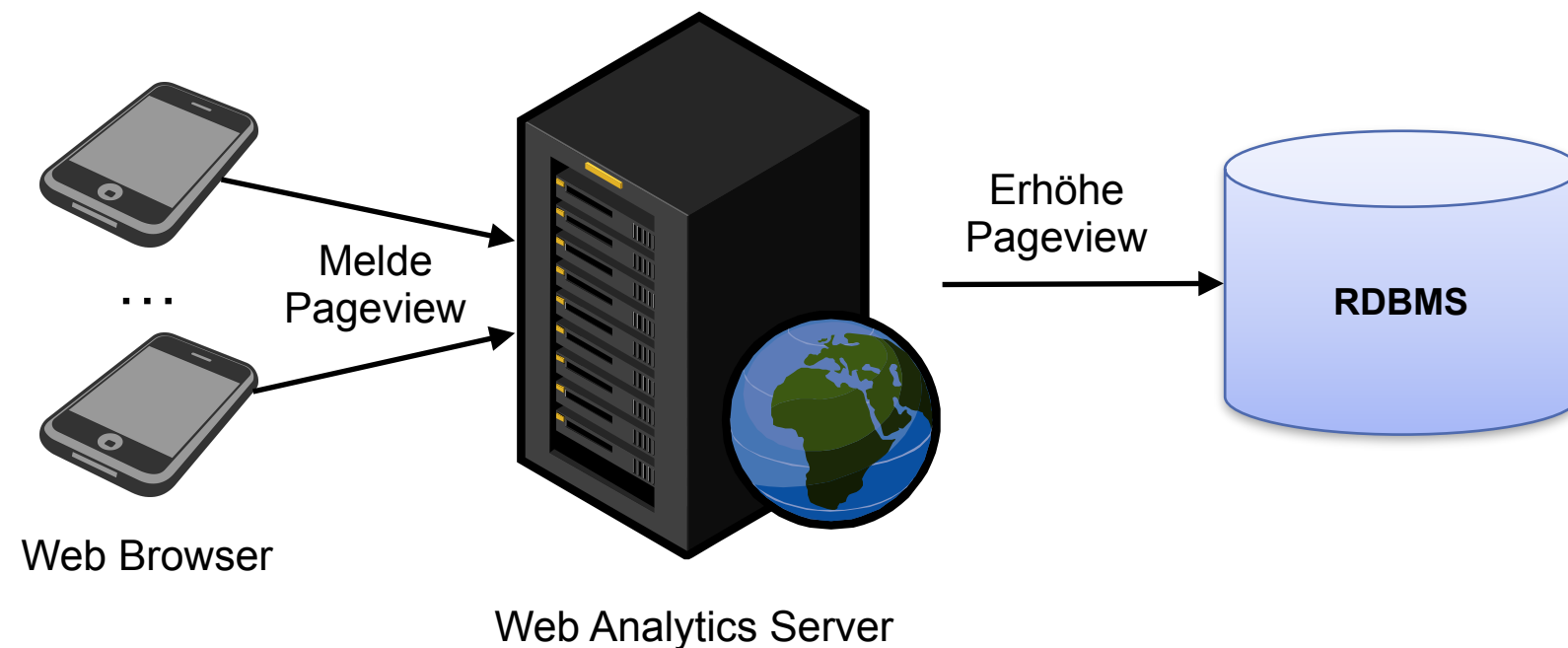
Wie viele Page-
views gab es in den
letzten 12 Stunden?

Wieviele
Einzelpersonen
haben die Domain
die letzten drei Tagen
stündlich besucht?

Grenzen der Skalierbarkeit am Beispiel: Web Analytics mit klassischer Datenbank

Web Analytics auf traditionelle Weise

- Soll die Anzahl der **Seitenzugriffe** für beliebige URLs verfolgen
- Basiert auf **relationaler Datenbank** mit einfacher **Tabelle**
- Webseite ruft den Server auf; der **inkrementiert** den Zähler der entsprechenden Zeile



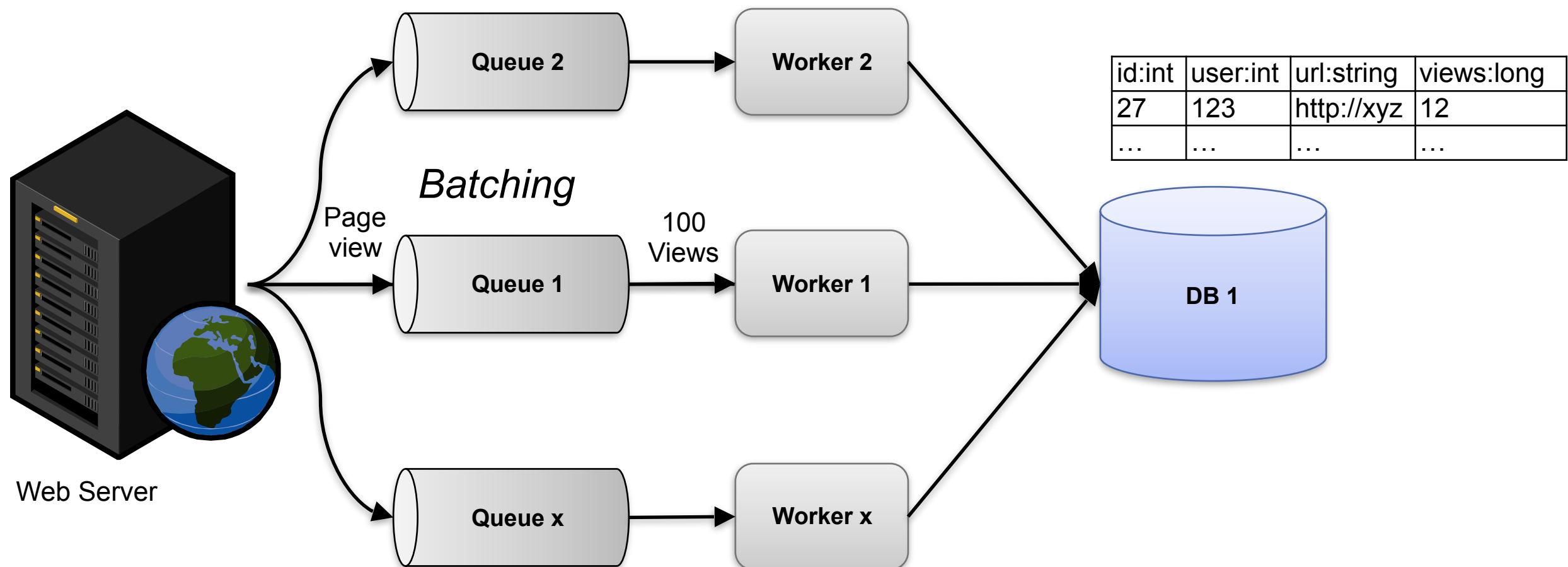
Spaltenname	Typ
id	integer
user_id	integer
url	varchar(255)
pageviews	bigint

Anzahl der URLs steigt: **timeout error on inserting to db**

1. Schritt: Warteschlange(n)

- Eine Operation pro Anfrage ist ineffizient
- **Warteschlange** bündelt (Batch) 100 Pageviews pro DB Request
- Pageviews überleben DB Timeouts

Frage: helfen mehr Queues?



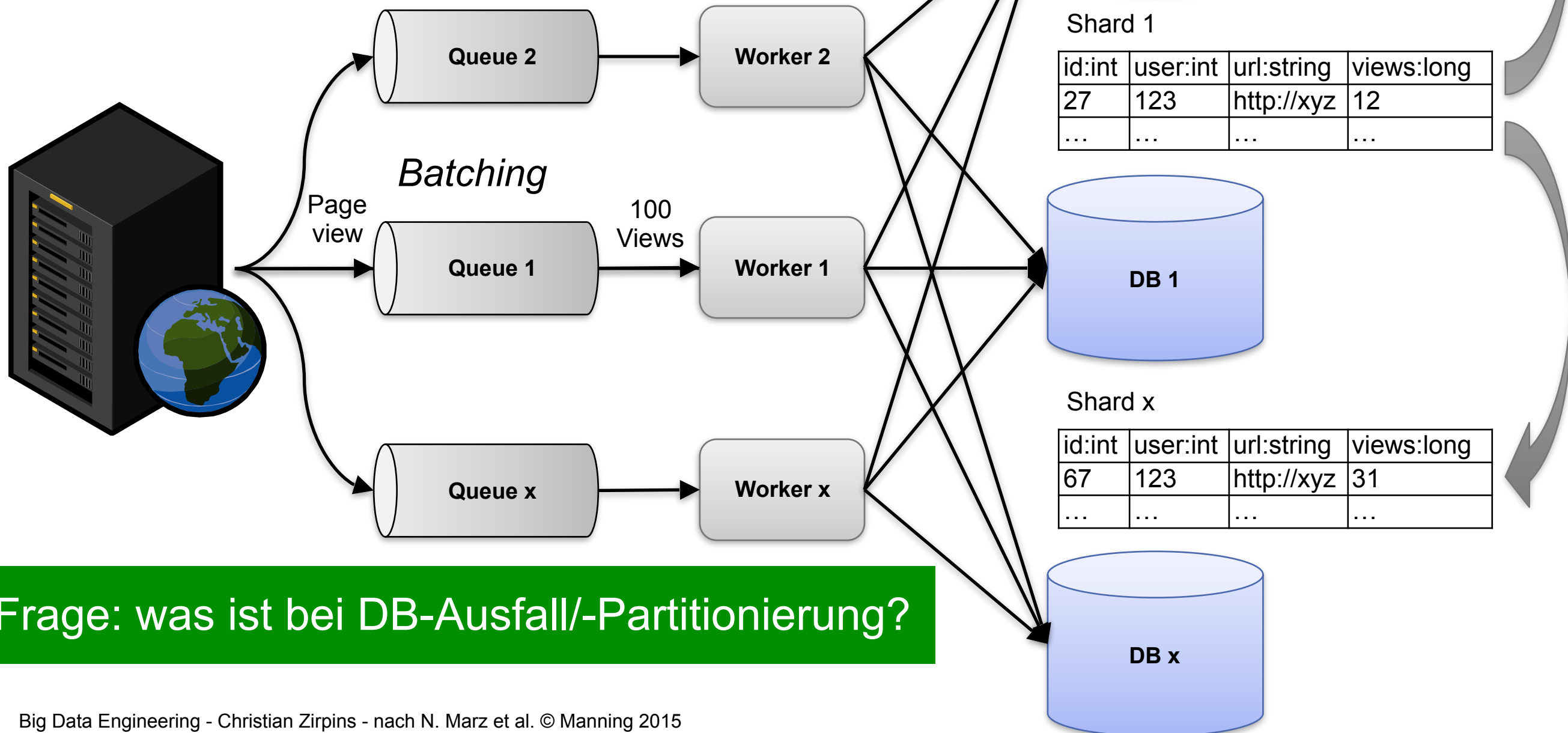
Anzahl der URLs steigt: `timeout error on inserting to db`

2. Schritt: Sharding

- Horizontale Skalierung durch **Sharding**: Jeder Server schreibt/liest nur Teilmenge

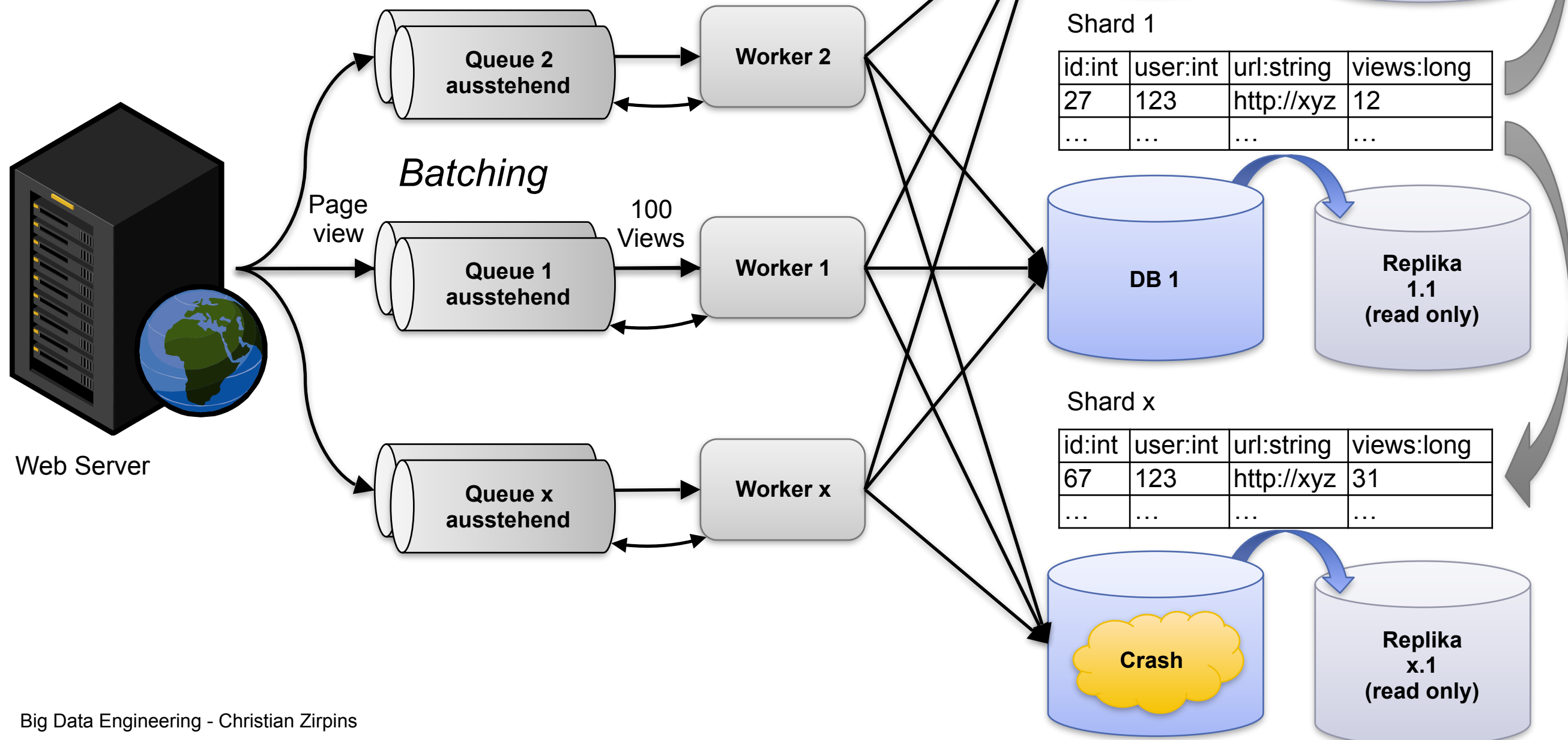
- $ShardId = hash(key) \bmod \#Shards$

Client Code ändern!



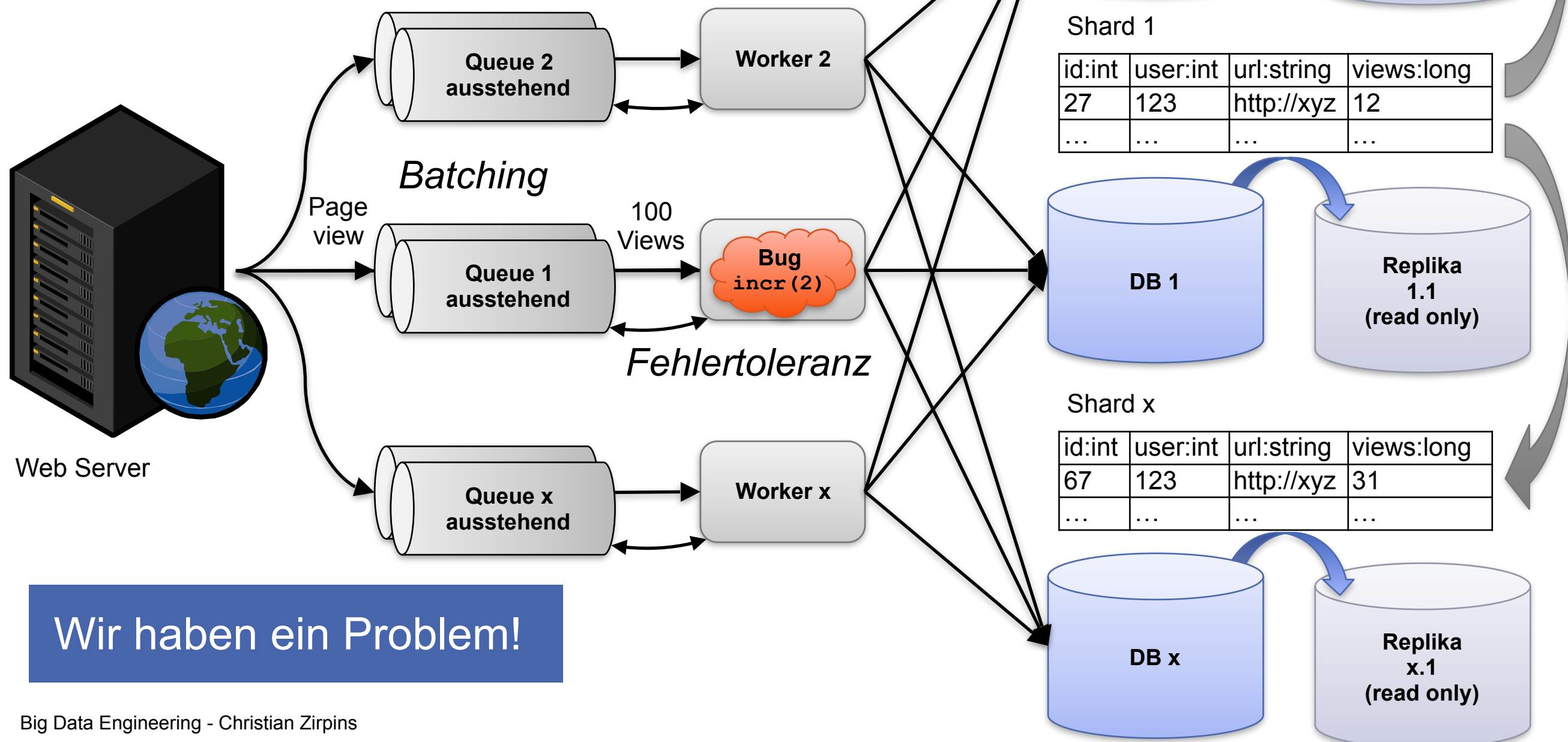
3. Schritt: Fehlertoleranz

- Viele DB-Server steigern das **Ausfallrisiko**
- **DB-Replikate** für Reads, nicht für Writes!
- Writes gehen auf eine **zweite Queue**



4. Schritt: "Human Fault-Tolerance"

- Menschliche Fehler sind unvermeidbar
- Bugs können die Datenbasis verfälschen
- Inkrementelle Updates mutierbarer Daten:
Fehler sind nicht rückgängig zu machen.



Was ist das Problem?

- Das System ist **unüberschaubar komplex**: *Queues, Shards, Replikas Resharding Skripte, usw.*
 - Die Datenbank kapselt nicht die verteilte Implementierung, sondern überläßt die Komplexität der Anwendungsebene
- Das System ist nicht für **menschliche Fehler** ausgelegt
 - Komplexität fördert menschliche Fehler

Human Fault Tolerance: nicht optional sonder *essentiell*.

Was können wir tun?

Vor allem zwei Dinge

- Skalierbarkeit und Komplexität gemeinsam adressieren

Systemkomponenten sollen ihre Verteilung verbergen und durch hinzufügen neuer Knoten skalieren (**Shared-Nothing**)

- Menschliche Fehler tolerieren

Rohe Daten sollten **unveränderbar (immutable)** gehalten werden. Falscher Code kann niemals "gute Daten" zerstören.

Bringt NoSQL die Lösung?

- Es gibt mittlerweile viele **skalierbare Datensysteme** u.a. *MapReduce Cluster* und *NoSQL DBs*
- Aber sie sind speziell; entscheidend ist die richtige Verwendung!

Hadoop kann umfangreiche Batch Berechnungen auf sehr großen Datenmengen parallelisieren, aber diese haben hohe Latenz.

Cassandra erreicht Skalierbarkeit durch ein stark limitiertes Datenmodell. Weil Daten veränderbar sind, ist das Modell intolerant bei menschlichen Fehlern.

**Noch mal ganz von vorn:
Was sind eigentlich **Datensysteme**?**

Was ist ein Datensystem?

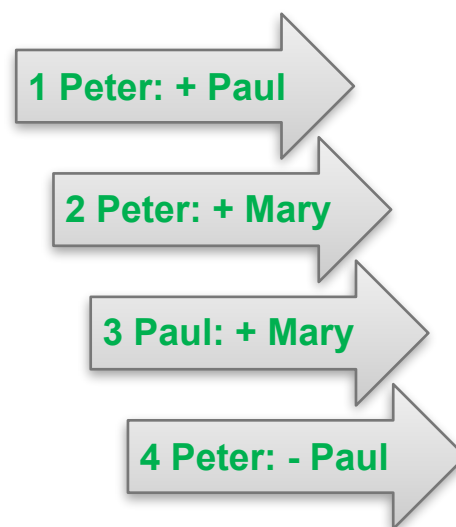
Ein **Datensystem** beantwortet Fragen basieren auf Informationen, die in der Vergangenheit bis in die Gegenwart erworben wurden.

- Im sozialen Netzwerk...
 - *"Wie lautet der Name der Person?"*
 - *"Wie viele Freunde hat diese Person?"*
- In der Banking App...
 - *"Was ist mein aktueller Kontostand?"*
 - *"Welche Transaktionen haben vor kurzem stattgefunden?"*

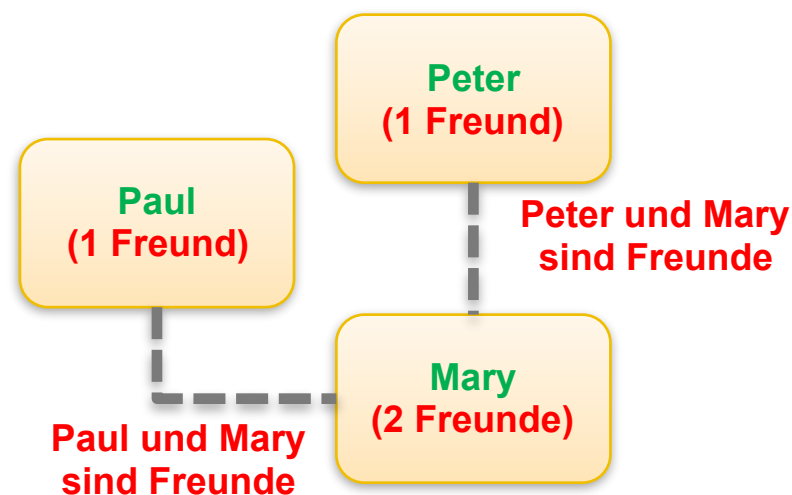
Fakten und Ableitungen

Ein Datensystem gibt nicht nur gelernte **Fakten** wieder, sondern kombiniert sie auch, um Antworten **abzuleiten**.

Transaktionen



Profile



Grün: *Fakten*,
Rot: *Ableitungen*

Ein soziales Netzwerk

Fakten sind “Rohdaten”, die sich nicht weiter ableiten lassen.
Wir nennen sie nachfolgend einfach **Daten**.

Ein funktionales Modell für Datensysteme

- Datensysteme beantworten Fragen (bzw. Queries) durch **Berechnung von Funktionen über Daten.**

In einem **generischen Datensystem** kann jede mögliche Anfrage durch Ausführung einer Funktion über dem kompletten Datenraum beantwortet werden!

$$Query = Funktion (Datenraum)$$

Wie sollen Big Data Systeme beschaffen sein?

Robust und fehlertolerant

Einfach in Funktion und Aufbau, um Korrektheit sicherzustellen.
Unveränderbar und Neuberechenbar, um menschliche Fehler zu tolerieren.

Performant bei Reads und Updates

Geringe **Read Latenz** (wenige hundert Millisekunden)
Geringe **Update Latenz** (bei Bedarf) bis hin zur Echtzeit

Skalierbar

Systemleistung soll bei mehr Daten/Last durch *hinzufügen neuer Knoten* **horizontal skalieren**.

Wie sollen Big Data Systeme beschaffen sein?

Generisch

Unterstützung vieler Arten von Anwendungen.

Finanzmanagement, Social Media Analyse, wissenschaftliche Anwendungen, soziale Netze usw.

Erweiterbar und flexibel

Leichtes Ergänzen von Funktionen; **Datenmigration** durch Neuberechnung.

Explorative Analysen mit **ad hoc Queries** (ungeachtet Latenz).

Wie sollen Big Data Systeme beschaffen sein?

Einfach in der Wartung

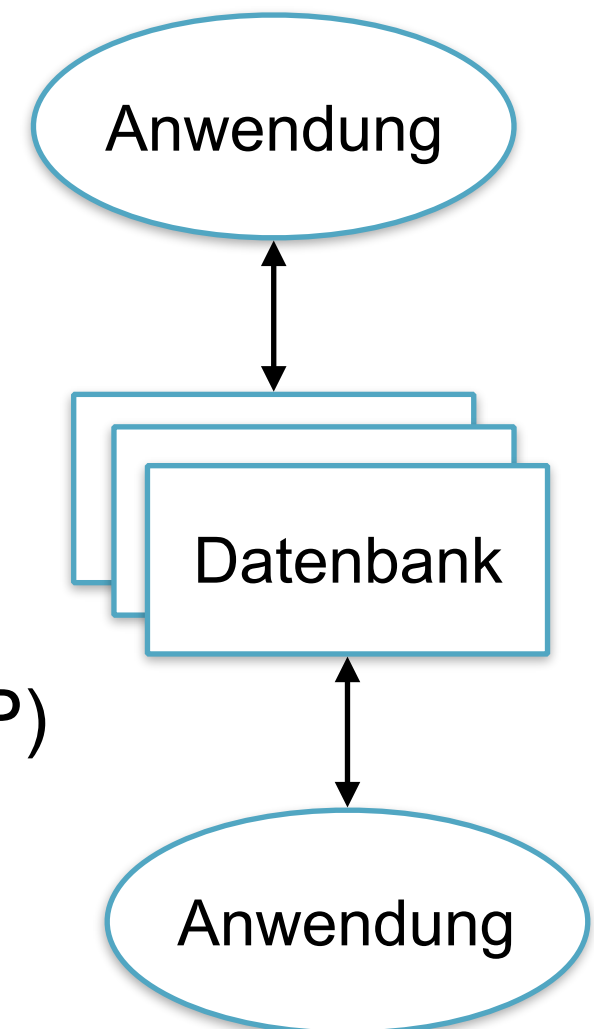
Geringer Aufwand um, ein System reibungslos am laufen zu halten.
Komponenten und Funktionen sollen einfach sein.
Komplizierte Teile (z.B. Random Read/Write DBs) nur in unkritischen Bereichen.

Einfach in der Fehlersuche

Für jeden Wert muss erkennbar sein, wie er zustande kam.
Vereinfacht durch *funktionales System* und *Neuberechnung*.

Vollinkrementelle Datensysteme

- Herkömmliche Datensysteme sind gekennzeichnet durch
 - Nutzung von Read/Write Datenbanken
 - Inkrementelles Fortschreiben des Zustands
 - Z.B. Ereigniszähler: bei Ereignis ändern
- Gilt für *relationale* und *nichtrelationale* DBMS
- Ansatz leidet unter vielen Problemen
 - Komplexität im Betrieb (z.B. Compactions)
 - Komplexität bei Konsistenzmechanismen (z.B. CAP)
 - Keine Toleranz bei menschlichen Fehlern



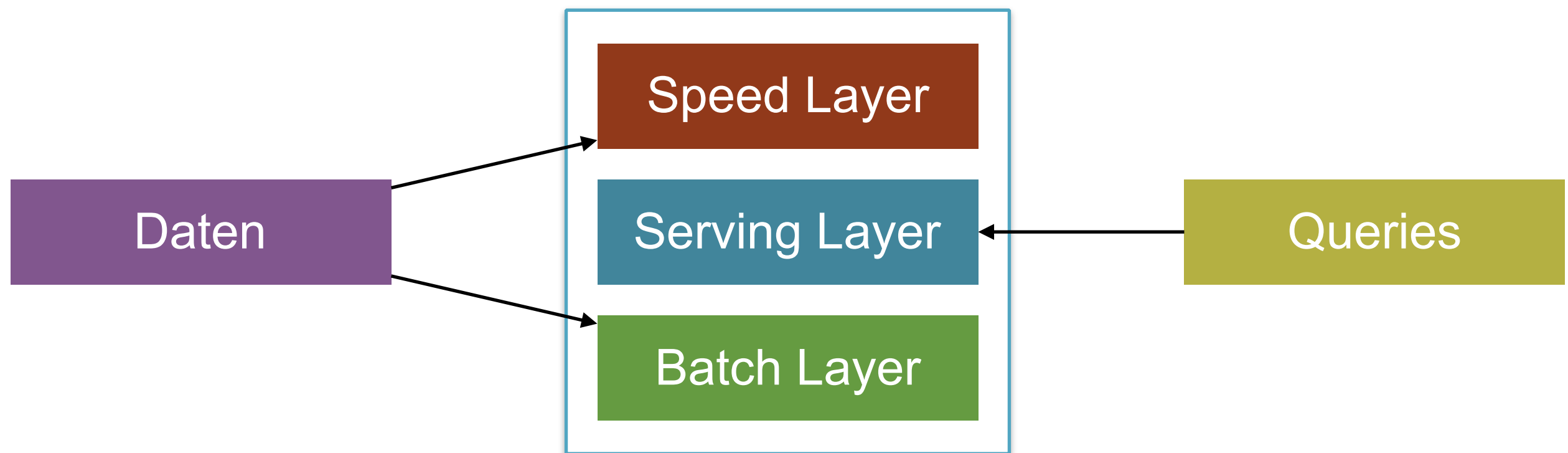
Das **CAP Theorem** besagt, dass verteilte (DB) Systeme nicht gleichzeitig *konsistent*, *verfügbar* und *tolerant gegenüber Netzausfall* (Partitionierung) sein können.

Die **Lambda Architektur**

Ein Paradigma für Big Data

Lambda Architekturen

- Ziel der **Lambda Architektur**: *Big Data Systeme zur Berechnung beliebiger Funktionen auf beliebigen Daten in Echtzeit.*
- Kombination mehrerer verschiedener Tools und Techniken auf drei hierarchischen Ebenen.



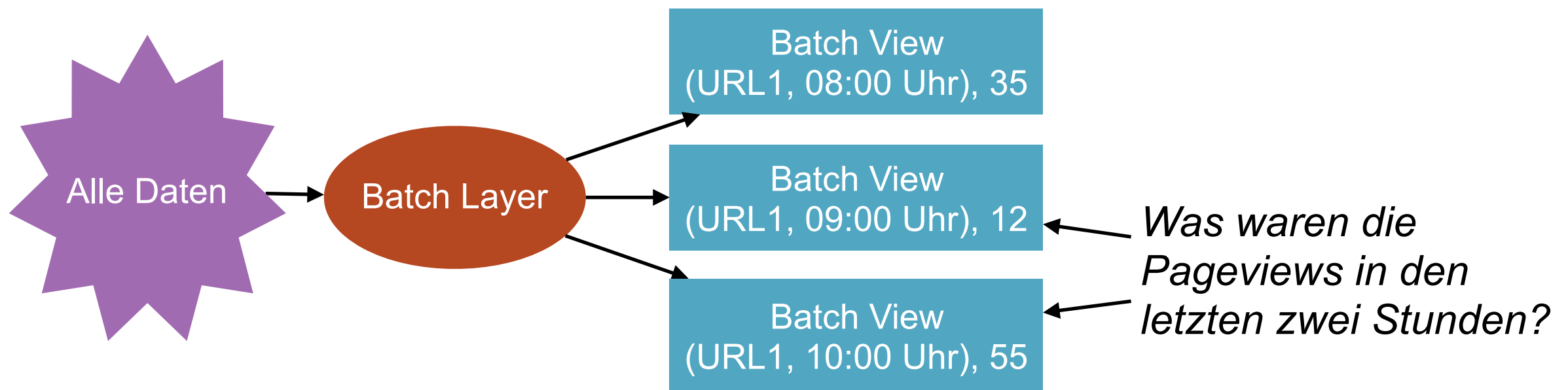
- Jede Ebene realisiert spezifische Eigenschaften des Datensystems
- Ebenen ergänzen sich (nutzen jeweils die nächste)

Vorausberechnung von Abfragen

- Ziel der **Lambda Architektur**: *query = function(all data)*
- Zur Abfragezeit ist es für Berechnung zu spät: **Vorausberechnen!**

Die *indizierte* vorausberechnete Query heißt **Batch View**.
batch view = function (all data), query = function (batch view)

- Abfragen mit einfacher Funktion auf dem Batch View Index.
- SuperWebAnalytics: *pageviews(Url, Stunde)*



Batch Layer

- Der Batch Layer hat zwei kontinuierliche Aufgaben
 - Speicherung des unveränderbaren **Master Datensatzes**
 - Vorausberechnung von Batch Views auf diesem Datensatz

```
function runBatchLayer() :  
    while(true) :  
        recomputeBatchViews()
```

- Berechnung prädestiniert für **Batch Verarbeitung**
 - **Hadoop** ist Industriestandard für Batch-Processing Systeme
 - Batch Berechnungen: einfach wie Single-Thread Programme; Parallelisierung erfolgt automatisch
 - Skaliert durch Hinzufügen von Maschinen
- Zur Speicherung reicht dann ein **Dateisystem**; z.B. **HDFS**
 - nur serielles Schreiben (*Write-Append*) und Lesen (*Batch Reads*)

Serving Layer

- Der Serving Layer hat eine kontinuierliche Aufgabe
 - Laden der Batch Views zur Abfrage durch Anwendungen
- Es genügt eine einfache verteilte Datenbank
 - Queries bedingen wahlfreies Lesen (*Random Reads*)
 - Update neuer Batch Views durch Überschreiben (*Batch Update*)
 - komplizierte wahlfreie Writes (*Random Writes*) unnötig
 - Z.B. **ElephantDB**¹
- Batch und Serving Layer reichen oft schon
 - Hadoop HDFS/MapReduce mit ElephantDB ist robust, fehlertolerant, skalierbar, generisch, erweiterbar, flexibel, wart- und debugbar
 - Aber: hohe **Update Latenz** durch Berechnung von Batch Views
 - Für Realtime Abfragen reicht es nicht

¹<https://github.com/nathanmarz/elephantdb>

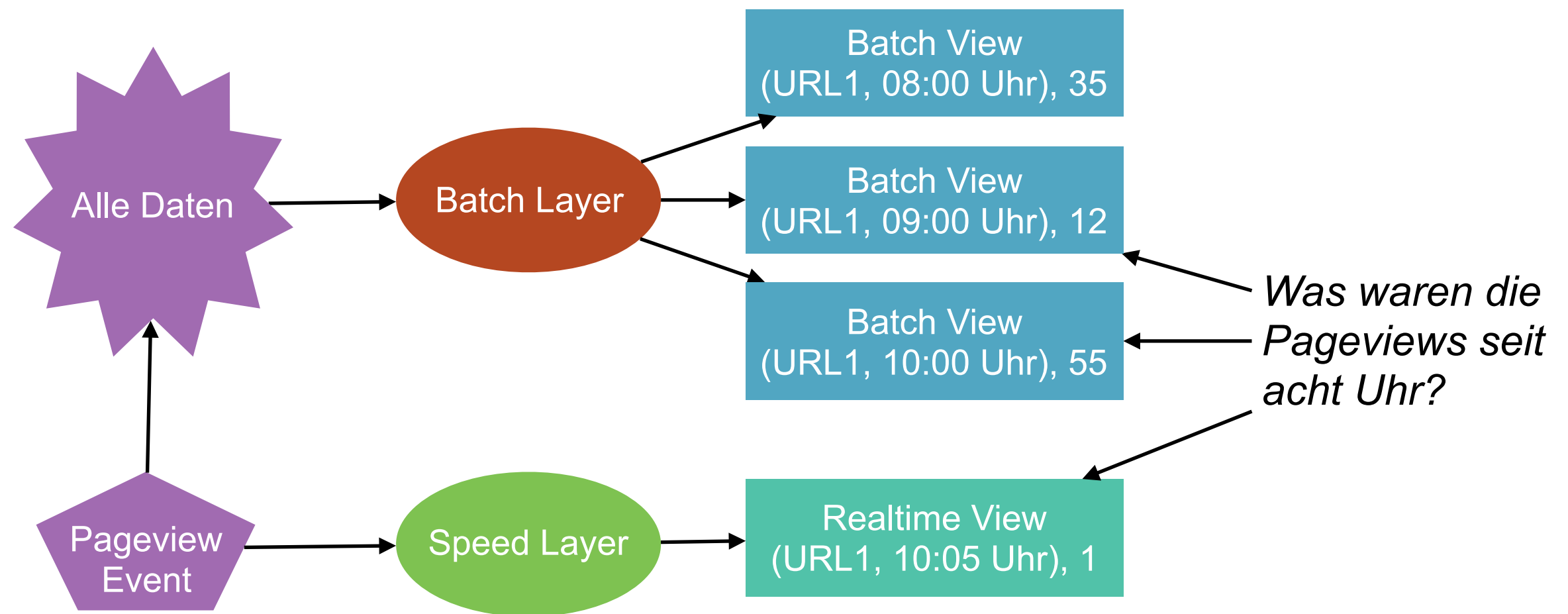
Speed Layer

- Der Speed Layer hat eine kontinuierliche Aufgabe
 - Sofortige inkrementelle Vorausberechnung beliebiger Funktion für alle neuen Datenelemente (genannt **Realtime View**)
- Realtime Views kompensieren Latenz der Batch Berechnung
 - Bei Abfrage des Batch Views fehlen neue Daten während Berechnung
 - Echtzeitabfragen kombinieren Batch und Realtime Views
 - Update des Batch View ersetzt entsprechenden Realtime View

batch view = *function(all data)*
realtime view = *function(realtime view, new data)*
query = *function(batch view, realtime view)*

Komplette Lambda Architektur als Gleichungssystem

SuperWebAnalytics Realtime Pageviews



Speed Layer

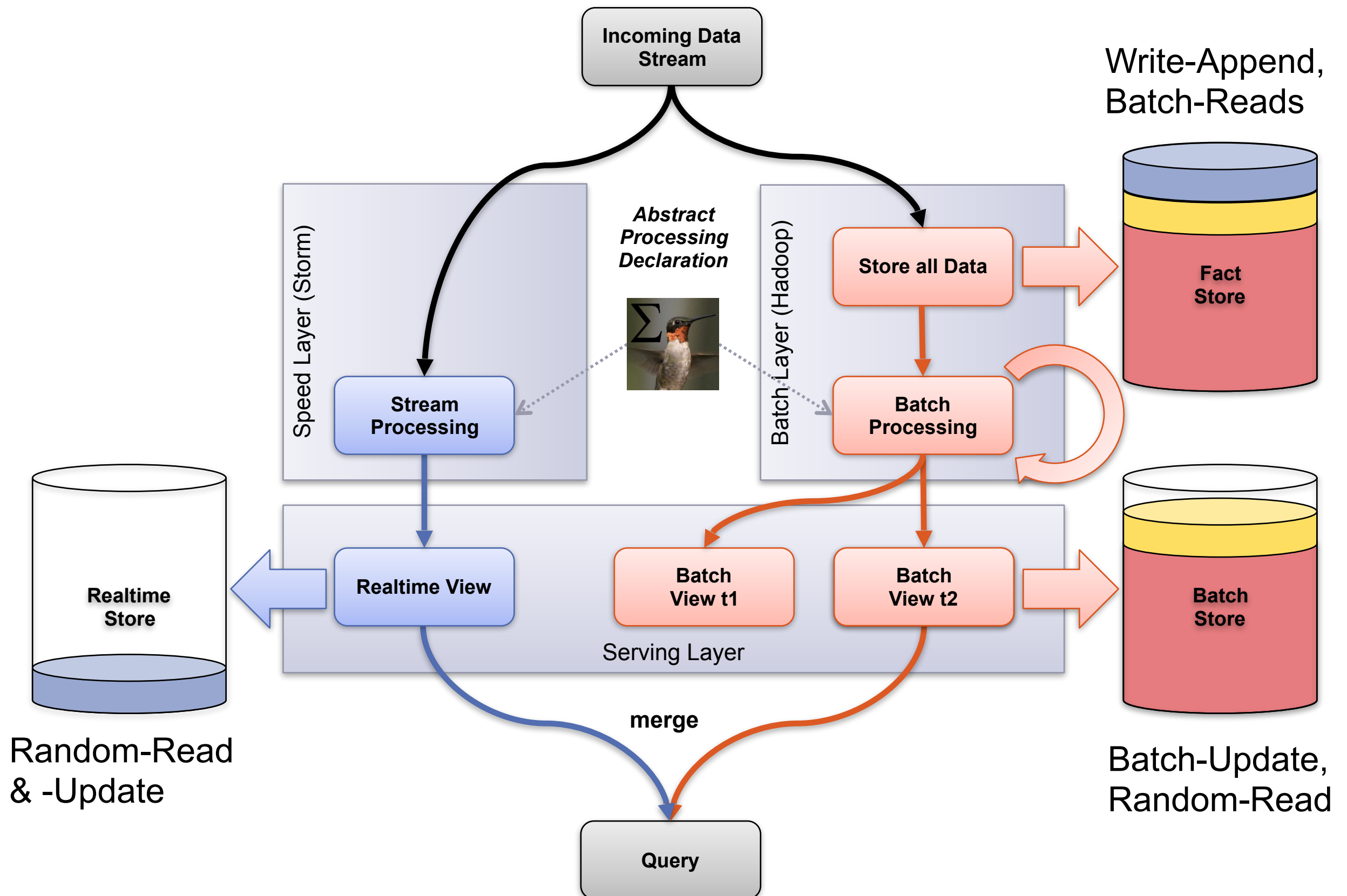
- Berechnung prädestiniert für **Stream Verarbeitung**
 - **Apache Storm** ist Industriestandard für Stream-Processing Systeme
 - Inkrementelle Berechnungen z.T. nur durch Approximation
 - Skaliert durch Hinzufügen von Maschinen

Eventual Accuracy: approximierte Realtime Views werden regelmäßig durch deterministische Batch Views ersetzt

- Speicherung benötigt vollinkrementelles DBMS
 - Wahlfreies Lesen (*Random Reads*) und Schreiben (*Random Updates*)
 - Eine mögliche Technik ist **Apache Cassandra**

Complexity Isolation: Komplexität beschränkt sich auf Ebene mit temporären Ergebnissen

Zusammenspiel der Lambda Layer



Eigenschaften von Lambda Architekturen

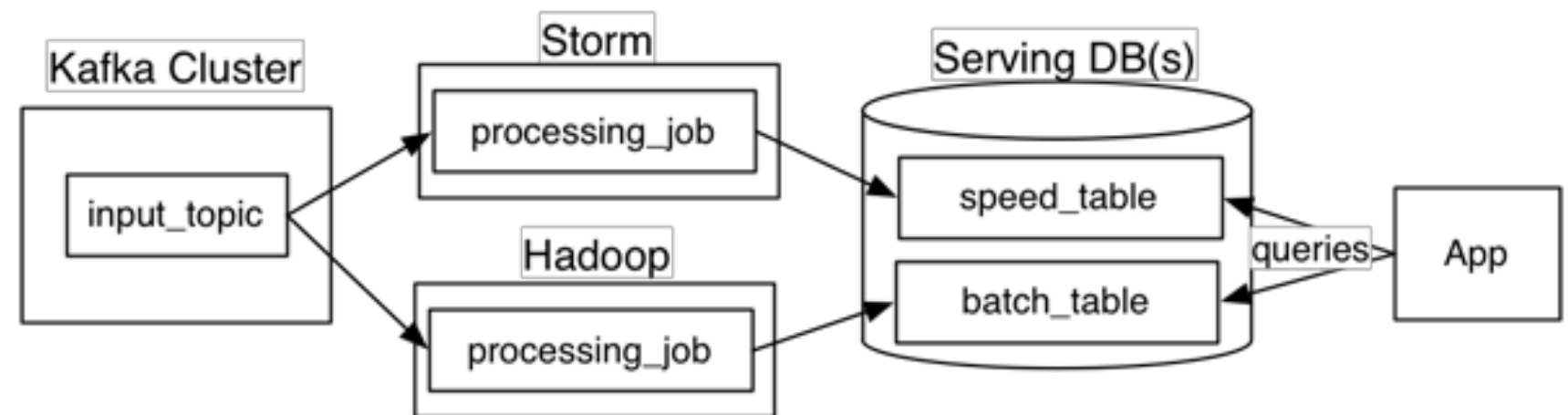
- Durch Serving- und Speed-Layer erfolgen Reads und Updates mit geringer Latenz.
- Alle Teile skalieren linear mittels zusätzlicher Hardware.
- Anwendungen sind robust und hochverfügbar durch Replikation und Failover-Mechanismen.
- Der Wartungsaufwand ist gering, da die Komplexität der Komponenten minimiert und isoliert ist.
- Anwendungen sind tolerant gegenüber menschlichen Fehlern durch unveränderliche Datenbasis und Neuberechnung.
- Fehler lassen sich leicht analysieren, da Ausgangswerte und Ergebnisse vorliegen.
- Der Ansatz ist generisch, da beliebige Funktionen auf beliebigen Daten ausgeführt werden können.
- Anwendungen sind erweiterbar, da sich Queries und Daten jederzeit (auch ad hoc) ändern lassen.

Andere Big Data Architekturen

Niemand hat gesagt es ist leicht... mehr dazu später.

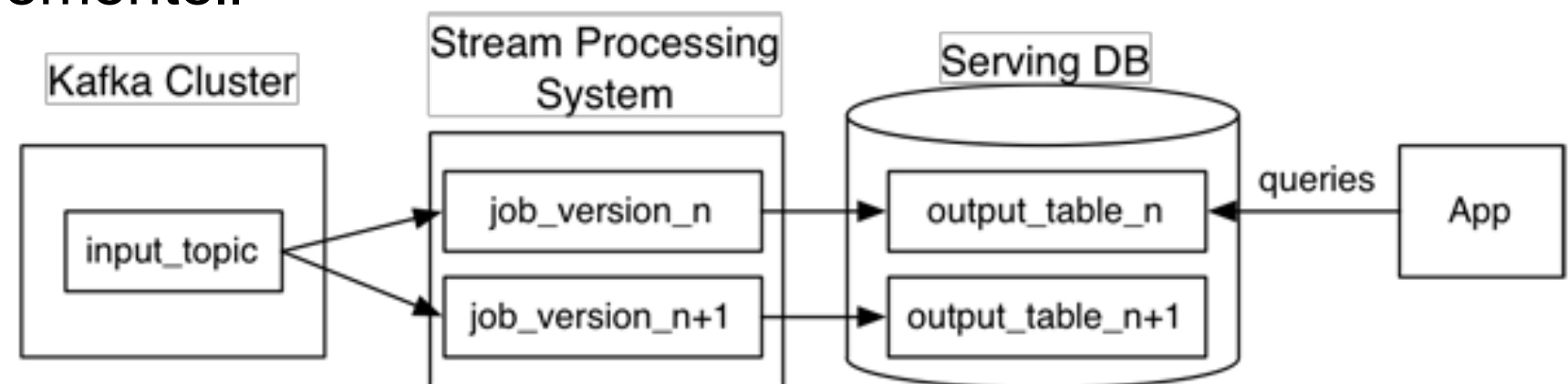
■ Lambda Architektur Kritik

- Synchronisierung von zwei Processing Frameworks (Batch, Stream)
- Potentiell **kompliziert** in Entwicklung und Betrieb



■ Kappa Architecture¹

- Nur Stream Processing für kontinuierliche Verarbeitung (job n)
- Bei Code Änderung: neuer Job $n+1$ wiederholt Stream
- Serving DB vollinkrementell

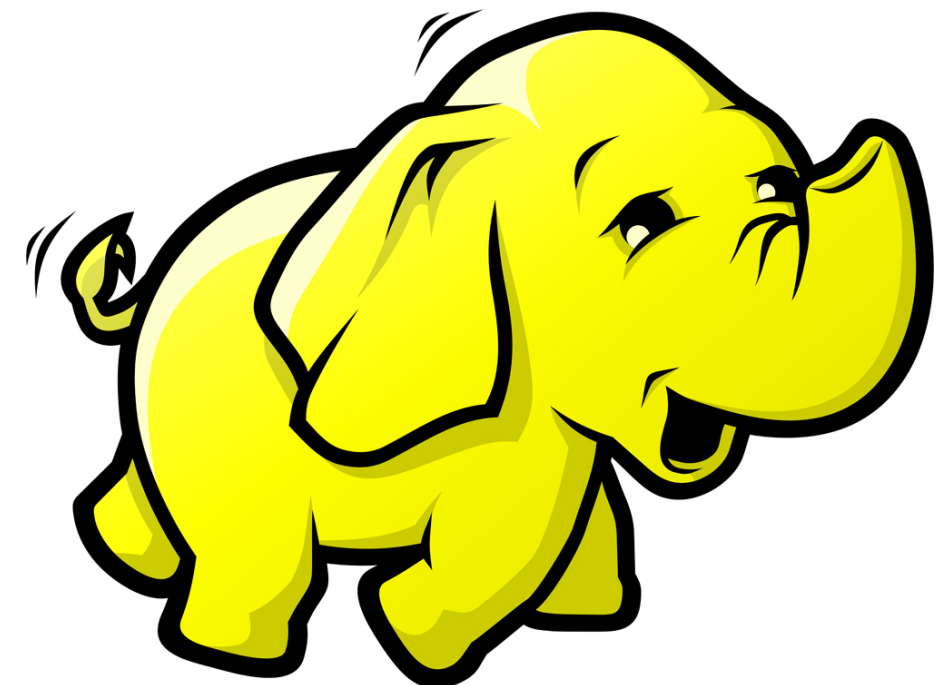


¹ <https://www.oreilly.com/ideas/questioning-the-lambda-architecture>

Big Data **Technologien und Open Source Projekte**

Batch-Verarbeitung

- **Batch Verarbeitungssysteme** liefern hohen Durchsatz bei hoher Latenz. Nahezu beliebige Berechnungen sind möglich, können aber Stunden oder Tage dauern. **Hadoop** ist Industriestandard
- Das **Apache Hadoop-Projekt** hat zwei Teilprojekte:
 - Das Hadoop **Distributed File System (HDFS)** ist ein verteiltes, fehlertolerantes Speichersystem, das auf Petabyte Daten skaliert
 - Hadoop **MapReduce** ist ein Programmiermodell und ein horizontal skalierbares Framework für Batch Verarbeitung, das mit HDFS integriert ist
- <http://hadoop.apache.org>



Serialisierungs-Frameworks

- **Serialisierungs-Frameworks** dienen zur Übertragung und Verwendung von Objekten zwischen Systemen und Sprachen
 - Können aus jeder Sprache Objekte in *Byte-Arrays* **serialisieren** und dann wieder in Objekte jeder Sprache **deserialisieren**
 - **Schema Definition Language** zur Definition von Objekten
 - Mechanismen zur **Versionierung**
- Beispiele: **Thrift, Protocol Buffers** und **Avro**
 - <https://thrift.apache.org>



Random-Access-NoSQL-Datenbanken

- Setzen andere Schwerpunkte als relationale Datenbanken
 - Spezialisierte Operationen statt SQL Ausdruckskraft
 - Hochverfügbarkeit statt strenger Konsistenz
- Beispiele: **Cassandra**, **HBase**, **MongoDB**, **Voldemort**, **Riak**, **CouchDB** u.a.
 - Alle haben unterschiedliche Semantik und sollten für ihre speziellen Zwecke verwendet werden.
- Das **Apache Cassandra Projekt** eignet sich gut für Big Data
 - Wide Column Store (zweidimensionaler Key-Value-Store)
 - Sehr effizient z.B. bei Zeitreihen Daten
 - <http://cassandra.apache.org>



Messaging- / Warteschlangen-Systeme

- Um Nachrichten zwischen Prozessen *fehlertolerant* und *asynchron* zu senden und zu konsumieren
- **Nachrichtenwarteschlangen (Queues)** sind Schlüsselkomponenten für Echtzeit-Verarbeitung
- Das **Apache Kafka Projekt** ist sehr etabliert
 - Verteilt, skalierbar und hoch performant
 - Oft zusammen mit Storm verwendet
 - <http://kafka.apache.org>



Stream-Verarbeitung

- **Echtzeit-Systeme** mit hohem Durchsatz bei niedriger Latenz.
- Nicht alle Berechnungen von Batch-Systemen sind möglich!
- Verarbeiten Nachrichten extrem schnell
- Das **Apache Storm Projekt** ist weit verbreitet
 - Storm-Topologien sind einfach zu schreiben und skalieren
 - <http://storm.apache.org>



Zusammenfassung und Ausblick

Die heutigen Themen

- Der Big Data Begriff
- Grenzen der Skalierbarkeit: Web Analytics mit klassischer DB
- Noch mal ganz von vorn: Was sind eigentlich Datensysteme?
- Die Lambda Architektur: Ein Paradigma für Big Data
- Big Data Technologien und Open Source Projekte

BDE-Termine — Winter 2018

#	KW	Termin	Block	Thema	Raum
	40	8.10.			
1	41	15.10.		Big Data Paradigma	Hörsaal
2	42	22.10.	Batch Layer	Big Data Modellierung	Hörsaal
3	43	29.10.		Big Data Speicherung	Hörsaal
4	44	5.11.		<i>Übung: Hadoop DFS/Thrift</i>	<i>Labor</i>
5	45	12.11.		Batch Verarbeitung	Hörsaal
6	46	19.11.		<i>Übung: Hadoop Map Reduce</i>	<i>Labor</i>
7	47	26.11.	Serving Layer	Big Data Bereitstellung	Hörsaal
8	48	3.12.		<i>Übung: Cascalog</i>	<i>Labor</i>
9	49	10.12.	Speed Layer	Big Data in Echtzeit	Hörsaal
10	50	17.12.		Queues und Stream Verarbeitung	Hörsaal
	51	24.12.			
	52	31.12.			
11	1	7.1.		<i>Übung: Apache Kafka/Storm</i>	<i>Labor</i>
12	2	14.1.		Micro-Batch Stream Verarbeitung	Hörsaal
13	3	21.1.		Lambda Architektur	Hörsaal
14	4	28.1.		Outro / Q&A / Abgabe	<i>Labor</i>

Raumplan

Hörsaal	E 303
Labor	LI137

Ressourcen und Unterstützung

Foliensätze / Begleitmaterial etc. auf ILIAS

- Anmeldung in der Vorlesungszeit (pwd 'ic4ip')

<http://bit.ly/2dsLZr3>

Email Kontakt

- Individuelle Fragen werden gerne per Email beantwortet.
- christian.zirpins@hs-karlsruhe.de

Literatur

- Nathan Marz, James Warren, "Big Data: Principles and best practices of scalable realtime data systems", Manning, 2015
- Martin Kleppmann, "Designing Data-Intensive Applications", O'Reilly, 2014 (Early Release)
- Tom White, "Hadoop: the definitive guide: storage and analysis at internet scale", 4. ed., O'Reilly, 2015, ISBN: 978-1-491-90163-2
- Michael Frampton, "Big Data Made Easy: A Working Guide to the Complete Hadoop Toolset", Apress, 2015
- Vivek Mishra, "Beginning Apache Cassandra Development", Apress, 2014
- Weitere Literatur und Online Material zu den Veranstaltungen

