Data in the Wild

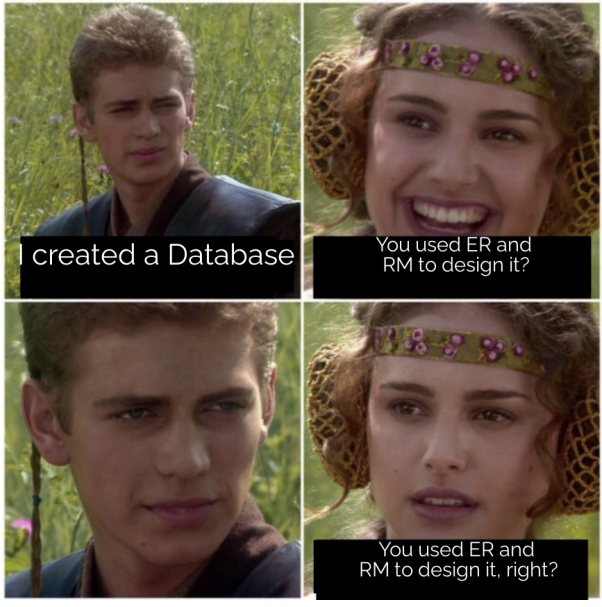
VL Big Data Engineering
(vormals Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

14. Juli 2022

Uni vs Realität



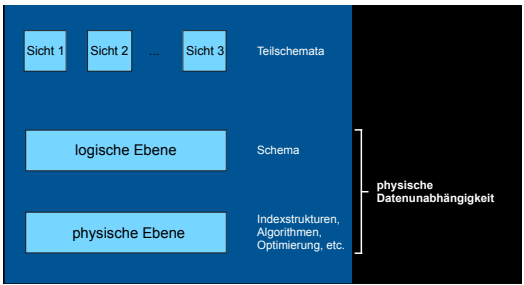
Uni vs Realität (Teil 1)

Uni	Realität	Hinweis
ER	keine Modellierung, keine Iteration mit Kunden, lieber direkt loslegen; alternativ: ein anderes Datenmodell: XML, JSON, OO, Graph; keine Teilschemata, keine logische Datenunabhängigkeit	logische Datenunabhängigkeit (LDUA)
RM	Tabellen, iterativ über Jahre erweitert (Attribute und Tabellen drangeflanscht), teilweise versteckt durch object-relational mapping (ORM)-Wrapper (wie z.B. in Django, kann man aber auch trennen!), viele Redundanzen, keine Normalisierung, viele Altlasten (wer benutzt dieses Feld eigentlich?), keine Sichten, zu weit gefasste Domains	nachträgliche Normalisierung, ORM, Django, LDUA
RA	irgendeine Programmierbibliothek, die das Rad neu erfindet, basierend auf CSV-Dateien; Anfragen prozedural in Bib formuliert, hard-codierte Anfragepläne	hard-codiert vs Spark

Physische Datenunabhängigkeit

Physische Datenunabhängigkeit

Das Datenbankschema ist unabhängig von seiner physischen Realisierung. Änderungen an der physischen Repräsentation der Daten (Hardware, Indexe, etc.) haben keine Auswirkungen auf das Datenbankschema.



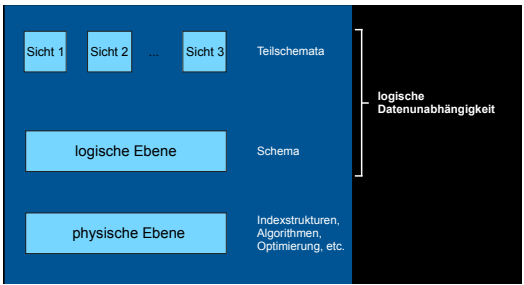
Vorteile:
physische Ebene kann nachträglich geändert werden

Nachteile:
Aufwand für Abstraktion (kein direktes Hard-coden physischer Aktionen)

Logische Datenunabhängigkeit

Logische Datenunabhängigkeit

Die Sichten der Endbenutzer sind unabhängig vom Datenbankschema. Änderungen am Datenbankschema haben keine Auswirkungen auf die Sichten der Nutzer.



Vorteile:
logische Ebene kann nachträglich geändert werden

Nachteile:
Aufwand für Sichtenerstellung,
Mögliche Probleme bei Updates durch Sichten hindurch

ORM

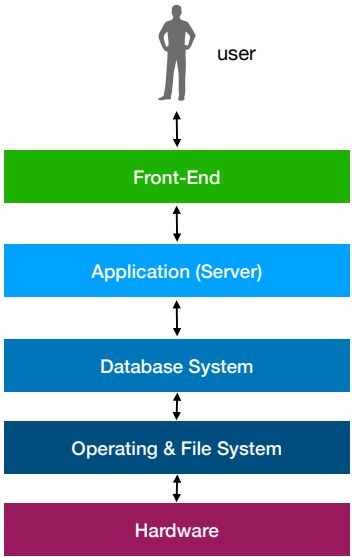
ORM: Object Relational Mapper

Ein objektrelationaler Mapper bildet die Daten aus einer objektorientierten Programmiersprache direkt in Relationen einer Datenbank ab. Die Schnittstellen des DBMS werden teilweise oder komplett vom ORM versteckt. Das ORM übersetzt "Anfragen" aus einer OO Syntax zu SQL und Ergebnisse zurück zu Objekten.

Vorteile:
kein Impedance Mismatch, d.h. keine Reibungsverluste durch das manuelle Abbilden von Daten aus der OO-Welt auf die relationale Welt

Nachteile:
möglicherweise Kontrollverlust über die Datenbank (da versteckt durch den ORM)

Standard DBMS-Architektur



examples:

web browser

map-browsing (e.g. Google Maps),
image collection (e.g. Lightroom),
data analytics software (e.g. Tableau)

PostgreSQL, MySQL,
Oracle, SQLite

Linux, Windows, OS X,
Android, iOS

CPU, DRAM, SSD, hard
disk

ORM-Architektur

user

examples:

Front-End — web browser

Application (Server) — map-browsing (e.g. Google Maps), image collection (e.g. Lightroom), data analytics software (e.g. Tableau)

ORM — Django

Database System — PostgreSQL, MySQL, Oracle, SQLite

Operating & File System — Linux, Windows, OS X, Android, iOS

Hardware — CPU, DRAM, SSD, hard disk

ORM Best Practices

Hybrides ORM

1. DB (zum Teil) selbst erstellen/manipulieren (insbesondere das Schema, Constraints, etc.) **und:**
2. ORM nutzen

Beispiel:
`https://www.djangoproject.com/`

Normalisierung

Normalisierung

Theorie zur Bestimmung der Güte von Relationen mit Hilfe funktionaler Abhängigkeiten.
Algorithmen zur Verbesserung der Güte von Relationen.

Beispiel für Verletzung von 3NF

PersonenProjekte

persnr integer	name character varying	vorname character varying	geburtsdatum date	projektnr integer	pname character varying	prioritaet integer
1	Schweitzer	Albert	1973-03-01	5	Unis	7
2	Carlos	Rob	1975-07-12	1	Data Center	10
2	Carlos	Rob	1975-07-12	3	Lobbyisten	8
2	Carlos	Rob	1975-07-12	6	Kaninchenzüchter	2
3	Mueller	Peter	1963-10-09	2	Hasenzüchter	3
3	Mueller	Peter	1963-10-09	4	Politiker	5

funktionale Abhängigkeiten

{persnr} → {name, vorname, geburtsdatum}

{projektnr} → {pname, prioritaet}

{projektnr} → {name, vorname, geburtsdatum}

Vorteile:
gutes Werkzeug

Nachteile:
etwas in die Jahre gekommen,
zu stark orientiert an atomaren Domänen (SQL 92),
erste Normalform bereits ein Widerspruch zu modernem SQL

Hard-codiert vs Spark

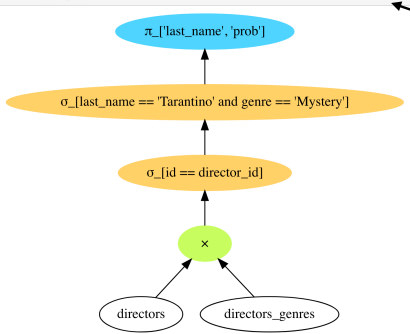
Prozedurale vs deklarative Leseweise

Nur weil es so aussieht, als wäre es ein prozedurales Programm, heißt das noch lange nicht, dass wir es auch in dieser Reihenfolge ausführen müssen.

In [4]:
```
# Unoptimized plan
cp = Cartesian_Product(directors, directors_genres)
sel1 = Selection(cp, "id == director_id")
sel2 = Selection(sel1, "last_name == 'Tarantino' and genre == 'Mystery'")
proj = Projection(sel2, ['last_name', 'prob'])

graph = proj.get_graph()
Source(graph)
```

Out[4]:



Interpretation 1
Dies ist prozeduraler Code, der genau in dieser Reihenfolge ausgeführt werden soll/muss.

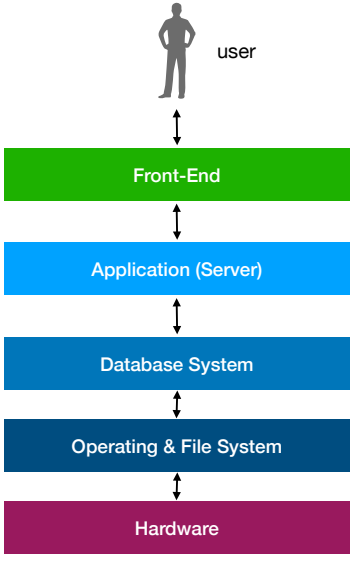
Interpretation 2
Dies ist nur eine spezielle Form deklarativer Programmierung. Auch wenn es aussieht wie ein prozedurales Programm, können wir machen, was wir wollen, solange sich am berechneten Ergebnis nichts ändert.

(ähnliche Bedingung wie bei Isolation)

Uni vs Realität (Teil 2)

Uni	Realität	Hinweis
SQL	nur SQL 92, und das nur partiell; Anwendungslogik wird nur zum Einlesen der Relationen in die Anwendung benutzt, erfindet Teile von SQL und Anfrageoptimierer neu, schlechte Skalierbarkeit; SQL-hints, materialisierte Sichten	Anwenderlogik Horror-Story, Schnitt zwischen Vor- und Nachverarbeitung
Datensparsamkeit	meist kein Thema, im Gegenteil: Datensammelwut ist der Standard	
A	oft nur pro Tupel genutzt: key/value-Stores, NoSQL	KV-Semantik
C	kaum genutzt jenseits von Foreign Keys; insbesondere Trigger kaum genutzt; oder schlicht gar keine C-Bedingungen; TA-Einstellungen des DBMS falsch genutzt (transaktionale Trigger sind nicht ganz einfach)	Trigger
I	alle möglichen Abschwächungen; mit oder ohne Wissen um die Konsequenzen, siehe A	Eventual Consistency
D	irgendwie Teil der Backup-Strategie	

Wo welche Funktionalität implementieren?



Beispiel: filter()

filter(): alle Daten werden bis zum Nutzer hochgeschickt

filter(): alle Daten werden bis zum Front-End hochgeschickt

filter(): alle Daten werden bis zum App-Server hochgeschickt…

filter(): alle Daten werden bis zur DB hochgeschickt…

filter(): auf dem Speichermedium, nur Ergebnisse oder geeignete Obermenge wird verschickt

Anwenderlogik: Wo welche Funktionalität implementieren?

Datenintensive Operationen

Generell gilt: Funktionalität soweit unten im Stack wie möglich ausführen.

Vorteile: weniger Daten werden durch die Gegend geschickt

Nachteile:
nicht immer ganz einfach zu realisieren
zum Entwickeln oft nicht notwendig, da kein Performance-Problem

DB-Funktionalität

Gehört im Zweifel ins DBMS oder (partiell) in tiefere Schichten. DB-Funktionalität in höheren Schichten birgt das Risiko, irgendwann zu einem Performance und/oder anderen Problem zu werden, dass durch das DBMS gelöst werden könnte.

Beispiel:
Transaktionslogik im Anwendungs-Server

Key/Value-Stores

Key/Value-Stores

Ein Key/Value-Store erlaubt

1. das effiziente Speichern und Anfragen von Schlüsseln, die auf beliebige Werte abgebildet werden.
2. Transaktionen über mehrere Keys werden dabei typischerweise nicht unterstützt.
3. Funktionalität und Anfragen, die nicht auf eine Key/ValueSemantik abgebildet werden können, werden meist nur ungenügend unterstützt.

Beispiele:
URL $\mapsto$ Inhalt einer Webseite (das Internet)
hierarchischer Dateipfad $\mapsto$ Inhalt einer Datei (Dateisystem)
ID $\mapsto$ JSON-Dokument (Dokument-Store, z.B. MongoDB)

Key/Value-Stores

Vorteile: sehr effizient und ausreichend, falls keine anderen Zugriffsmuster benötigt werden
Produkte bieten meistens auch scale-out (Verteilung auf mehrere Server)

Nachteile:
sehr eingeschränkter Use-Case
sehr langsam bei allen Anfragen, die nicht über einen Schlüssel anfragen
im Grunde nichts anderes als **ein** Index/Hash-Map

ECA-Regeln und Trigger (1/2)

Event Condition Action (ECA)-Regeln

1. **E**vent: spezifiziert ein Ereignis (event)
2. **C**ondition: spezifiziert eine Bedingung, die für ein Ereignis überprüft wird
3. **A**ction: spezifiziert eine Aktion, die ausgeführt wird, falls die Bedingung erfüllt ist

Datenbanktrigger

Ein Datenbanktrigger erlaubt das Formulieren von ECA-Regeln direkt im DBMS.

ECA-Regeln und Trigger (2/2)

Beispiel:

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

Rufe eine Funktion, die updates logged.
Aber nur, wenn sich etwas geändert hat.

[https://www.postgresql.org/docs/13/sql-createtrigger.html]

Video: https://youtu.be/aTeRR9XmPWE

Vorteile:
sehr mächtiges Werkzeug,
sehr geeignet für komplexe Konsistenzbedingungen, Tracing, Event-basiertes Ändern von Relationen,
d.h. diese Dinge können dort implementiert werden, wo sie meistens auch hingehören: ins DBMS!

Nachteile:
nicht ganz einfach zu nutzen (unterschiedliche Syntax je nach DBMS),
schwierig zu debuggen,
möglicherweise unerkannte Endlosschleifen,
eher langsam

Uni vs Realität (Teil 3)

Uni	Realität	Hinweis
Sicherheit	nachträglich konzipiert oder reingehackt	
Privatheit	nachträglich konzipiert oder reingehackt, falls es überhaupt Thema ist	
QualSicherung	kein Staging-Environment	Staging
Automatisches Testen	App-Testen vs Konsistenz der DB (siehe Trigger)	Trigger vs Test
Dokumentation	nichtssagende Namen von Tabellen und Attributen; Semantik unklar; Nutzer (Lesen und/oder Schreiben) der Tabellen unklar; Effekte über Tabellen hinweg unklar	LDU nachträglich
Erweiterbarkeit	schwierig bis unmöglich durch fehlende Interfaces und Sichten; unbekannte Abhängigkeiten im Code	LDU nachträglich
Physisches Design	neue Hardware kaufen? neues DBMS kaufen? Indexe? Datenbankstatistiken aufgefrischt?	Physical Design Advisory, Performance Tuning

Deployment Environment/Staging

Deployment Environment

In einem Deployment Environment werden mehrere Versionen desselben Systems in einer Pipeline angeordnet.

1. Entwicklung: System zum Entwickeln neuer Features auf beliebigen Beispieldaten
2. Test: System zum Testen neuer Features, typischerweise durch Continuous Integration angebunden, sollte geeignete Benchmarks und Szenarien testen
3. Staging: System zum Testen neuer Features, das exakt dem Produktionssystem entspricht insbesondere was den Zustand der Daten angeht. D.h. idealerweise eine Replika des Produktionssystems.
4. Produktion: Produktionssystem mit echten Daten, echten Kunden

[https://en.wikipedia.org/wiki/Deployment_environment]

Beispiel:

Erinnern Sie sich an die Motivation aus dem Foliensatz “Banken...”? Raten Sie mal, was der Grund war für den Crash einer dieser Szenarios einer dieser Banken...

Physical Design Advisory und Performance Tuning

Physische Datenunabhängigkeit ist toll, aber: irgendwer muss letztendlich festlegen und konfigurieren, wie die Daten physisch abgelegt werden (Indexe, Hardware, Datenverteilung, etc.). Dies macht typischerweise ein Datenbankadministrator (DBA). Hierfür stellen viele DBMS umfangreiche Werkzeuge zur Verfügung.

Physical Design Advisory

Werkzeug zum halbautomatischen Festlegen der physischen Konfiguration eines DBMS.

Beispiel:
Datenbankoptimierungsratgeber https://docs.microsoft.com/de-de/sql/tools/dta/tutorial-database-engine-tuning-advisor?view=sql-server-ver15

Autotuning

Werkzeug zum vollautomatischen Festlegen der physischen Konfiguration eines DBMS.

Beispiel:
https://ottertune.com/

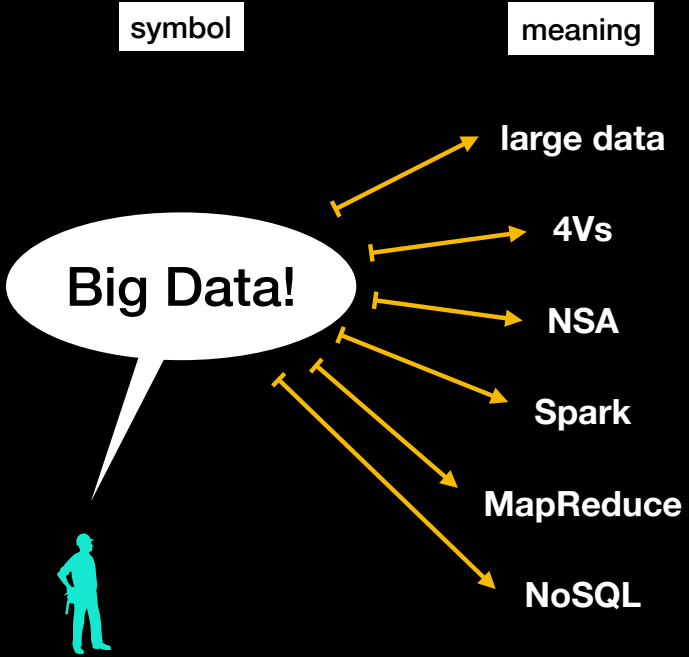
Uni vs Realität (Teil 4)

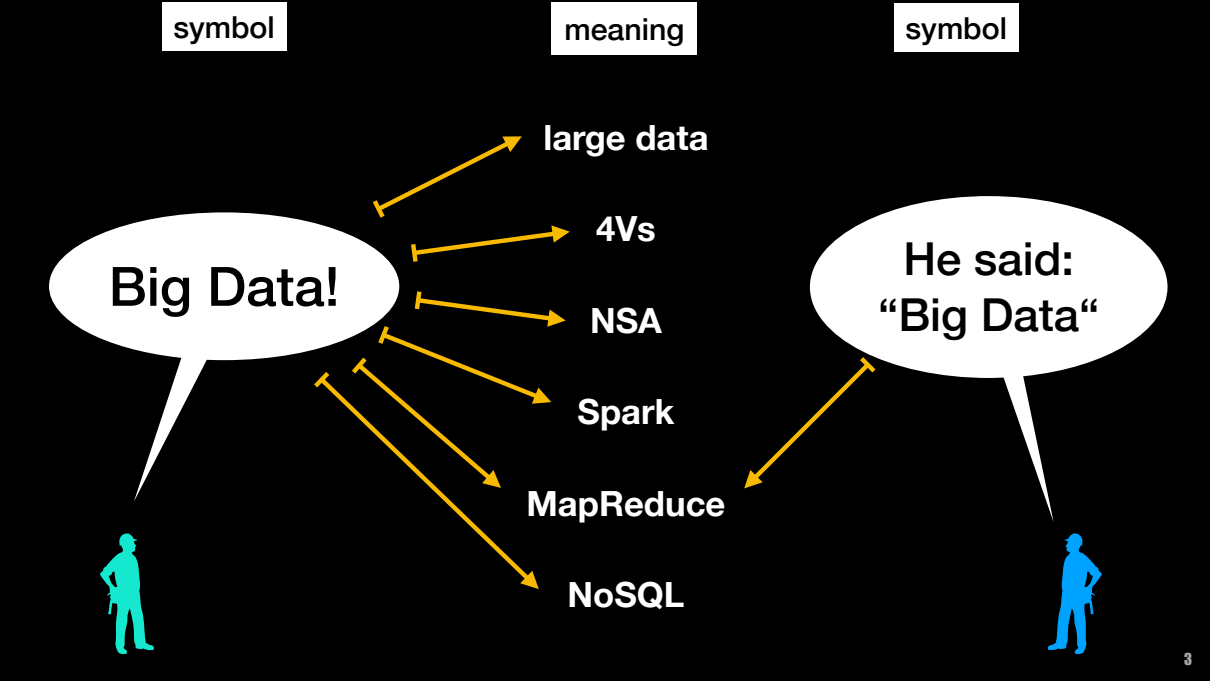
Uni	Realität	Hinweis
Datenhaltung	oft Abgleich mit Organisationshierarchie notwendig, soziale Aspekte insbesondere Claims; verteilte Datenhaltung und Anfrageverarbeitung entlang der Organisationshierarchie	Unternehmensprozesse vs Architektur der IT
Echtes Wissen, Kompetenz	gefühltes Wissen bzw. aktives Kaschieren und/oder Ignorieren des eigenen Nichtwissens (aka Klugscheißerei und Dummschwatz)[1]	Abgrenzung des eigenen Wissens
Konzepte	spezielle Implementierungen, Werkzeuge und Produkte, die auch irgendwie bestimmte uralte Konzepte nutzen (aber dies nicht unbedingt klarmachen)	Konzept vs Abbildung auf Technologie
Fachtermini	Buzzwords (unbewusst oder bewusst eingesetzt)	Buzzword Bullshit Bingo

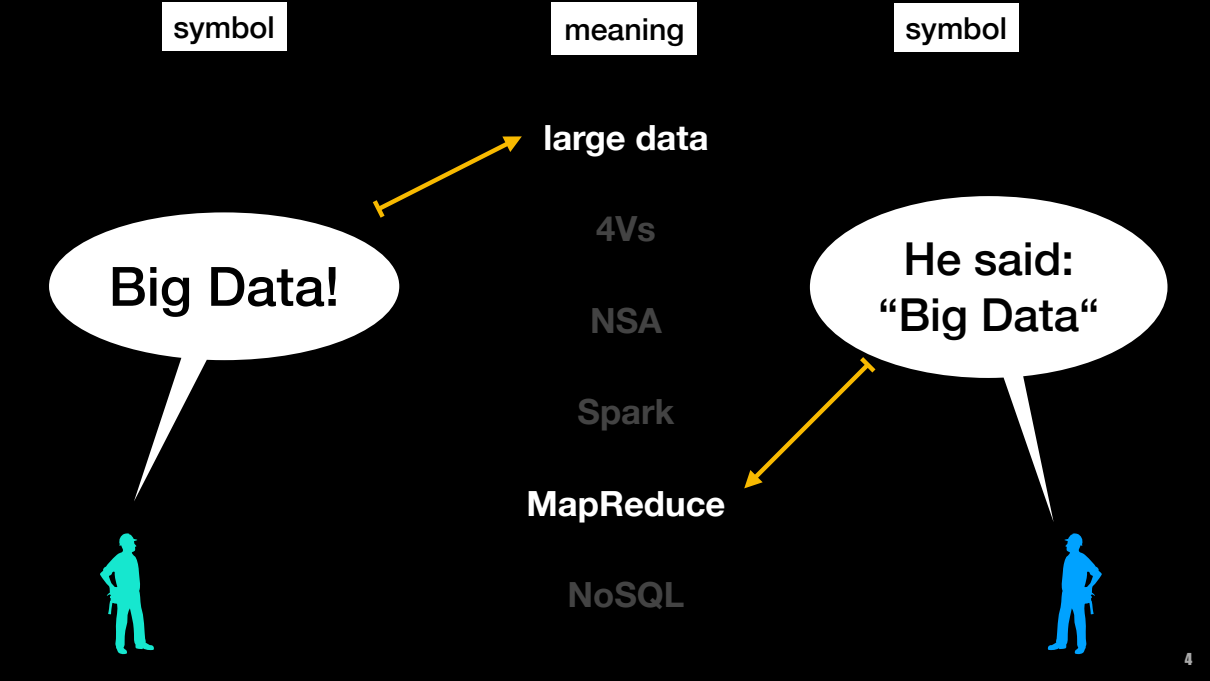
[1] https://thedailywtf.com/articles/classic-wtf-the-mainframe-database

Buzzword Bullshit Bingo

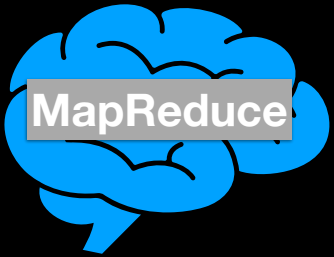
Problem 1:
ambiguous communication


symbol
meaning
large data
4Vs
Big Data!
NSA
Spark
MapReduce
NoSQL


symbol
meaning
symbol
large data
4Vs
NSA
Spark
MapReduce
NoSQL
Big Data!
He said:
“Big Data“


symbol
meaning
symbol
large data
4Vs
NSA
Spark
MapReduce
NoSQL
Big Data!
He said:
“Big Data“

translated to:



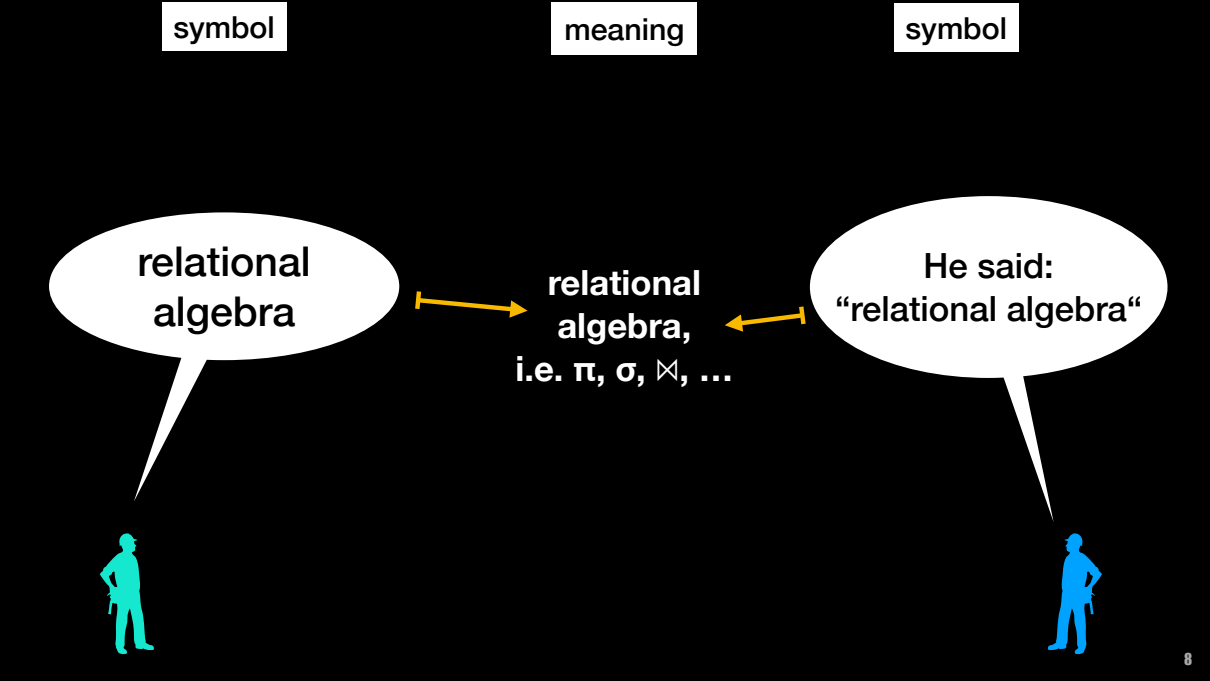
clear communication:

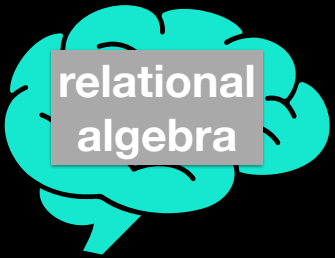
symbol

meaning

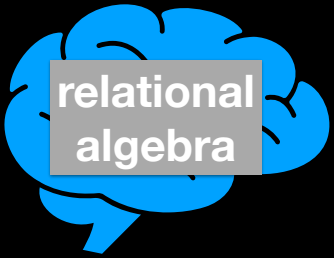
relational algebra

relational algebra, i.e. π, σ, ⋈, …

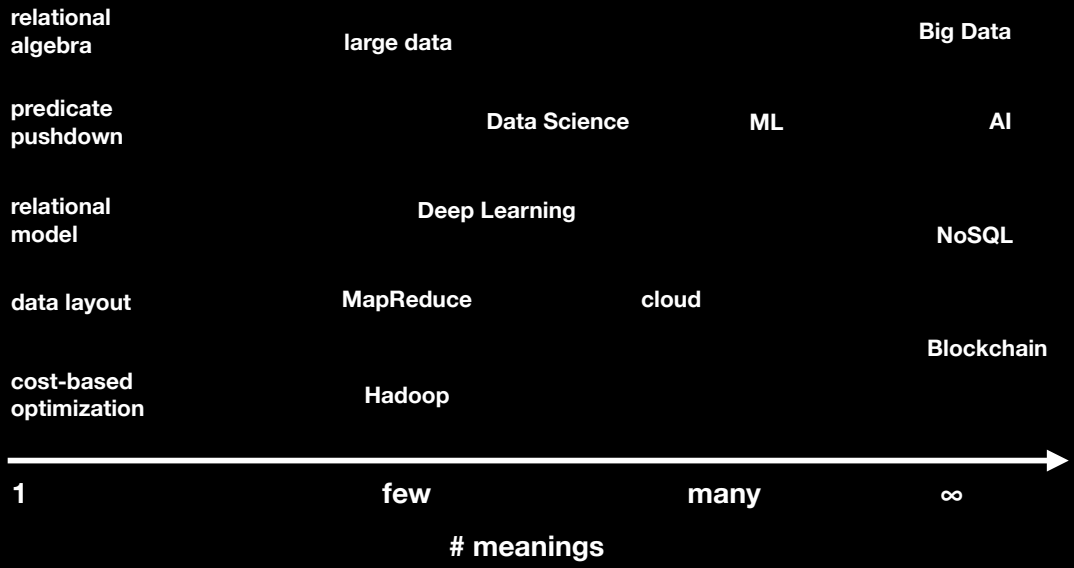

symbol
meaning
symbol
relational algebra
relational algebra, i.e. π, σ, ⋈, …
He said: “relational algebra“



translated to:



The Buzzword Bullshit Bingo Landscape



Konzept vs Technologie

Problem 2:
confusion of dimensions

dimension 1:
fancy sounding buzzwords
(labels & terms)

dimension 2:
technical principles
and patterns
(concepts, best
practices)

predicate pushdown

relational model

relational algebra

data layouts,
e.g. column vs row

cost-based optimization

compress to save I/O

dimension 2: technical principles and patterns (concepts, best practices)

symbol	meaning
predicate pushdown	“filter and project data as early as possible“

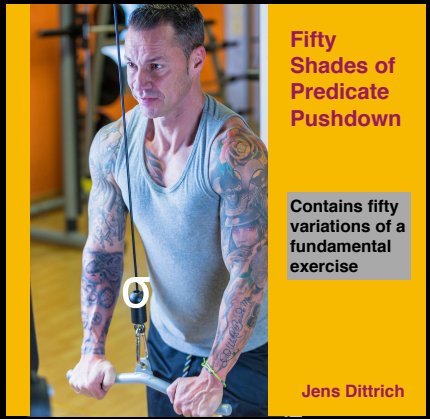
relational model

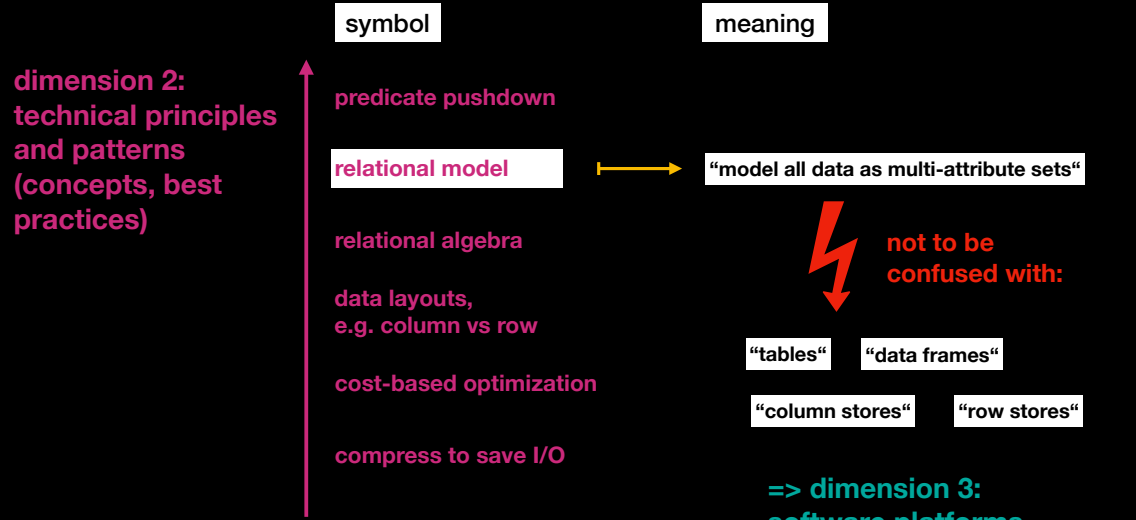
relational algebra

data layouts, e.g. column vs row

cost-based optimization

compress to save I/O




symbol
meaning
dimension 2:
technical principles
and patterns
(concepts, best
practices)
predicate pushdown
relational model
relational algebra
data layouts,
e.g. column vs row
cost-based optimization
compress to save I/O
“model all data as multi-attribute sets“
not to be
confused with:
“tables“
“data frames“
“column stores“
“row stores“
=> dimension 3:
software platforms
(concrete implementations
& frameworks)

symbol

meaning

dimension 2: technical principles and patterns (concepts, best practices)

predicate pushdown

relational model

relational algebra

“query those sets through a combination of simple set-valued functions“

data layouts, e.g. column vs row

cost-based optimization

compress to save I/O

dimension 2:
technical principles
and patterns
(concepts, best
practices)

predicate pushdown

relational model

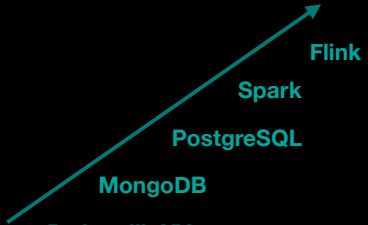
relational algebra

data layouts,
e.g. column vs row

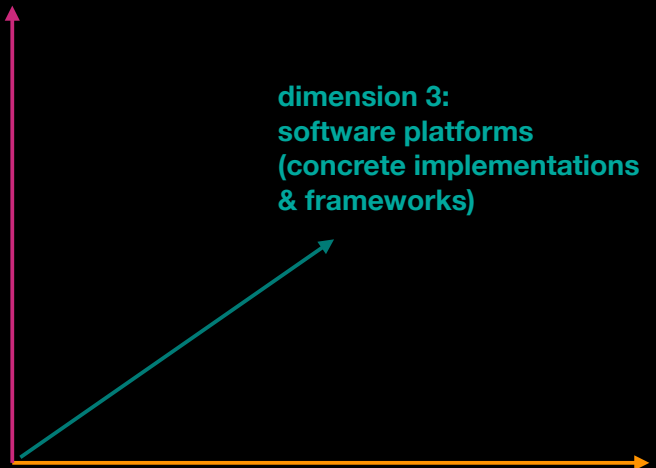
cost-based optimization

compress to save I/O

dimension 3:
software platforms
(concrete implementations
& frameworks)



dimension 2:
technical principles
and patterns
(concepts, best
practices)



dimension 3:
software platforms
(concrete implementations
& frameworks)

dimension 1:
fancy sounding buzzwords
(labels & terms)

Beispielberatungsszenario

(gestern erlebt...)

Problem (Teil 1)

- Große Webapplikation mit schnarchlahmer Performance: **Minuten** statt Millisekunden
- d.h. wir reden hier über Performance-Probleme in der Größenordnung **Faktor 10.000 und mehr langsamer!!!**
- SAP-System mit ca. 100.000 Tabellen, anscheined keine bis kaum Sichten?

völlig absurde Werte!

Lösungsraum

- Laufzeit der Applikation vs Laufzeit der Datenbankanfrage
- Laufzeit der Datenbankanfrage auf System mit und ohne Last
- Start: DB-Performance auf unausgelastetem System (Test oder Entwicklungssystem) profilen und wo immer möglich beheben
- physisches Design, inklusive Indexe und Partitionierung

Beispielberatungsszenario

Problem (Teil 2)

- mehr als 30,000 Nutzer*innen

Lösungsraum

- Lastverteilung klären (**nachdem** Sie ausgeschlossen haben, dass es an einzelnen Anfragen ohne Last liegt)
- falls Appserver und Datenbankserver auf selbem Server, klären ob das ein Lastproblem ist, falls ja: Datenbankserver auf eigenen Server packen
- klares Performance-Monitoring: wann führt welcher Teil des Systems zu einem Problem? SLAs!
- KIWI: Kill it with iron, SSDs vs hard disks

Beispielberatungsszenario

Problem (Teil 3)

- Setup: drei „Firmen“ an der Entwicklung beteiligt, jede mit mehreren Managern, Projektleitern, Teilprojektleitern und Entwicklern, mit sehr unterschiedlichem Datenbankwissen: von Top (der DB-Admin) zu einzelnen Managern, Projektleitern, Anwendungsentwicklern (vermutlich eher mittel bis wenig)
- Fingerpointing: „die anderen sind Schuld!“

Lösungsraum

- Profilen einer Anfrage kann sehr schnell lokalisieren, wo das Problem liegt (DB oder App)
- Problem hier: mehrere Performance-Probleme kombiniert
- Wissensdefizite bei Mitarbeitern identifizieren und beheben: entweder die Leute entsprechend ausbilden oder feuern (das meine ich so, wie ich es sage)
- klare Rolle/Zuständigkeit festlegen für physisches Design der Datenbank

Zusammenfassung

Problem: fehlende Passung von Ausbildung und Aufgabe

- Datenbanktechnologie wird in der Praxis oft von Leuten eingesetzt, die keine wirkliche Ahnung haben
- verschärfend kommt hinzu, dass diese Leute oft glauben, dass sie Ahnung hätten
- zusätzlich sind diese Leute auch oft noch beratungsresistent
- diese ungute Mischung führt zu einer Vielzahl von:
  - Performance-Probleme
  - Sicherheitsproblemen
  - architektonischer Fehlentscheidungen (inklusive Kaufentscheidungen)
  - enorm hohen Kosten

Würden Sie ihre Zähne von jemandem behandeln lassen, der eigentlich Bäcker gelernt hat?
Würden Sie sich von jemandem operieren lassen, der eigentlich Schlachter gelernt hat?

Lösung

Sicherstellen, dass diejenigen, die ein Werkzeug benutzen, dieses Werkzeug auch verstehen.

Noch etwas pointierter formuliert

Wenn Sie von jemandem gemanged werden, der nur betriebswirtschaftliches aber kein technisches Wissen hat, dann laufen Sie!