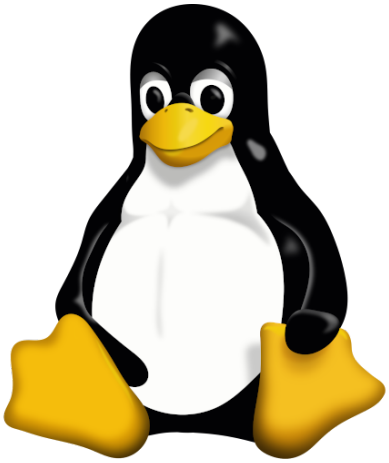


Data-Science

Wissenschaftliches Rechnen innerhalb von Datenbanken

Viterbi-Algorithmus

Daniel Dietrich

16. November 2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Bei der Überwachung und Steuerung von Systemen kommen häufig Sensoren zum Einsatz, die durch Messung physikalischer Größen Zustände quantitativ erfassen. Eine Analyse der Messergebnisse ermöglicht es, in geeigneter Weise auf Zustandsänderungen zu reagieren oder Zusammenhänge bzw. Ausreißer zu erkennen. Dieser Prozess produziert zum Teil große Datenmengen, die selektiert, aggregiert, gespeichert und analysiert werden sollen. Dafür kommen meist Datenbanken zur Speicherung und Verfahren des maschinellen Lernens und des Data-Minings zur Analyse zum Einsatz.
Da für die Vorhersage von Events oder Handlungen sowie der Erkennung von Clustern oder Muster durch Algorithmen (Machine Learning) oft Minimierungsverfahren und Statistik Anwendung finden, sind für viele dieser Analyse-Verfahren und auch anderer Operationen wissenschaftlichen Rechnens Vektorraumoperationen notwendig.
Eine Verarbeitung der Daten direkt im Datenbanksystem bietet gegenüber anderen Lösungen einige Vorteile. Dazu zählt die Nachhaltigkeit von SQL als „intergalactic dataspeak“ (SRL$^+$90). Ferner ist durch die Standardisierung von SQL als Anfragesprache[1] zu erwarten, dass die Anfragen sowohl weitgehend unabhängig vom verwendeten Datenbanksystem sind, als auch in absehbarer Zukunft noch funktionieren werden. Als ein weiterer Vorteil kann die Sicherheit der Daten betrachtet werden, da die Daten das Datenbanksystem zur Analyse nicht verlassen müssen. Allerdings ist SQL als mengenorientierte und deklarative Sprache untypisch für Vektorraumoperationen, was möglicherweise zu einem Abbildungsproblem der Algorithmen in die Sprache oder einem Performanz-Verlust gegenüber etablierten Umgebungen für wissenschaftliches Rechnen, Machine Learning und Big Data wie beispielsweise R, TensorFlow oder Hadoop-basierten Systemen wie Apache Flink oder Apache Spark führen kann. Frühere Untersuchungen (SAD$^+$10, DFS$^+$18) zeigten bereits, dass Datenbanken den speziellen Tools nicht prinzipiell unterlegen sind.
In dieser Arbeit möchte ich einen Überblick über die mathematischen Grundlagen (Kapitel 2), Technologien (Kapitel 3) sowie einige konkrete Systeme (Kapitel 4) geben.

[1] ISO/IEC TR 19075

Kapitel 2

Grundlagen

Dieses Kapitel enthält Definitionen und Modelle, die für die weiteren Kapitel notwendig sind. Im Abschnitt 2.1 werden zwei Arten von Präfixen vorgestellt, zwischen denen in dieser Arbeit unterschieden wird. Anschließend führt Abschnitt 2.2 Elemente und Operationen der linearen Algebra ein. Für einen praxisnäheren Vergleich soll die Auswertung eines Hidden Markov Models genutzt werden. Dieser und damit verbundene Problemklassen werden im Abschnitt 2.3 aufeinander aufbauend definiert. Im Abschnitt 2.4 wird ein Algorithmus betrachtet, der ein Problem der zuvor vorgestellten Problemklassen löst. Dieser Algorithmus wird zunächst hergeleitet und am Ende so modifiziert, dass dieser numerisch stabil ist.

2.1 Präfixe

In dieser Arbeit werden sowohl IEC-Präfixe (Binärpräfixe) gemäß (IEC00, Prz05) als auch SI-Präfixe (Dezimalpräfixe) nach (TM01) verwendet. Tabellen und Beispiele dazu befinden sich in Tabelle A.1.

2.2 Lineare Algebra

Dieser Abschnitt definiert Elemente und Operationen der linearen Algebra basierend auf (Fis17).

2.2.1 Vektorraum

Sei $(K, +, \cdot)$ ein Körper. Eine algebraische Struktur $(V, \oplus, \odot)$ mit $\oplus\colon V \times V \to V$ und $\odot\colon K \times V \to V$ heißt Vektorraum über K, falls

- $(V, \oplus)$ eine abelsche Gruppe, sodass $\forall u, v, w \in V$:

(V_1)	$v \oplus u = u \oplus v$	Kommutativgesetz
(V_2)	$u \oplus (v \oplus w) = (u \oplus v) \oplus w$	Assoziativgesetz
(V_3)	$0_V \in V$ mit $v \oplus 0_V = 0_V \oplus v = v$	neutrales Element
(V_4)	$-v \in V$ mit $v \oplus (-v) = (-v) \oplus v = 0_V$	inverses Element

- und $\forall \alpha, \beta \in K, \quad \forall u, v, w \in V$:

(S_1)	$\alpha \odot (u \oplus v) = (\alpha \odot u) \oplus (\alpha \odot v)$	Distributivgesetz 1
(S_2)	$(\alpha + \beta) \odot v = (\alpha \odot v) \oplus (\beta \odot v)$	Distributivgesetz 2
(S_3)	$(\alpha \cdot \beta) \odot v = \alpha \odot (\beta \odot v)$	Assoziativgesetz
(S_4)	$1 \in K$ mit $1 \odot v = v$	neutrales Element

Zusammen mit den folgenden Definitionen ist $(V, +, \cdot)$ mit $V = K^n$ und $n \in \mathbb{N}^+$ ein Vektorraum über K:

- Vektoraddition:
$$+ \colon V \times V \to V, \quad (a, b) \mapsto a + b = c, \quad c_i = a_i + b_i$$
- Skalarmultiplikation:
$$\cdot \colon K \times V \to V, \quad (\lambda, a) \mapsto \lambda \cdot a = b, \quad b_i = \lambda \cdot a_i$$

Die Elemente von V heißen Vektoren und einen Vektorraum über $\mathbb{R}$ wird als reellen Vektorraum bezeichnet. In dieser Arbeit sind Vektoren Elemente des endlichdimensionalen, reellen Vektorraums $(V, +, \cdot)$ mit $V = \mathbb{R}^n$ und $n \in \mathbb{N}^+$.

2.2.2 Matrix

Eine $m \times n$-Matrix A ist ein Element der Menge $M^{m \times n}$ mit $m, n \in \mathbb{N}^+$ und kann wie folgt dargestellt werden:

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}$$

Sei $(R, +, \cdot)$ ein Ring und $l, m, n \in \mathbb{N}^+$, dann sind folgende Operatoren definiert:

- Matrixaddition:
$$\oplus \colon R^{m \times n} \times R^{m \times n} \to R^{m \times n}, \quad (A, B) \mapsto A \oplus B := (a_{ij} + b_{ij}) = \begin{pmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{pmatrix}$$
- Matrixprodukt:
$$\odot \colon R^{l \times m} \times R^{m \times n} \to R^{l \times n}, \quad (A, B) \mapsto A \odot B = C = (c_{ij}), \quad c_{ik} := \sum_{j=1}^{m} a_{ij} \cdot b_{jk}$$

Blockmatrix

Sei $A \in R^{m \times n}$ eine $m \times n$-Matrix mit $m = \sum_{i=1}^{r} m_i$ und $n = \sum_{j=1}^{s} n_j$, dann kann A in die Blöcke $A_{ij} \in R^{m_i \times n_j}$ partitioniert werden

$$A = \left(\begin{array}{c|c|c} A_{11} & \dots & A_{1s} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{r1} & \dots & A_{rs} \end{array}\right)$$

Seien $A \in R^{m \times n}$, $B \in R^{n \times p}$ zwei Blockmatrizen mit $A_{ij} \in R^{m_i \times n_j}$, $B_{jk} \in R^{n_j \times p_k}$, so kann das Matrixprodukt $A \odot B = C \in R^{m \times p}$ blockweise berechnet werden, wobei $C_{ik} = \sum_{j=1}^{n_j} A_{ij} \cdot B_{j_k}$ mit $1 \leq i \leq m_i$, $1 \leq j \leq n_j$ und $1 \leq k \leq p_k$.

Dünnbesetzte Matrix

Eine Matrix $A \in M^{m \times n}$ heißt dünnbesetzt (Bar15), falls für die Anzahl T_N (abhängig von m und n) der von Null verschiedenen Einträge gilt:

$$T_N \in \mathcal{O}(N \cdot \log(N)) \text{ mit } N = \max\{m, n\}$$

Bandmatrix

Seien $p, q \in \mathbb{N}_0$, so ist die dünnbesetzte Matrix $A \in M^{m \times n}$ eine Bandmatrix der Bandbreite $l = p + q + 1$, falls für ihre Einträge a_{ij} gilt:

$$a_{ij} = 0 \quad \text{für } j + p < i \text{ oder } i + q < j$$

Neben der Hauptdiagonale sind also nur p untere und q obere Nebendiagonalen besetzt.

$$A = \begin{pmatrix} a_{11} & \dots & a_{1(q+1)} & 0 & \dots & \dots & \dots & 0 \\ \vdots & \ddots & & \ddots & \ddots & & & \vdots \\ a_{(p+1)1} & & \ddots & & \ddots & \ddots & & \vdots \\ 0 & \ddots & & \ddots & & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & & \ddots & & \ddots & 0 \\ \vdots & & \ddots & \ddots & & \ddots & & a_{(n-q)n} \\ \vdots & & & \ddots & \ddots & & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 & a_{n(n-p)} & \ddots & a_{nn} \end{pmatrix}$$

2.3 Hidden Markov Model

Mit Hidden Markov Modelle können spezielle Vorgänge aus der realen Welt modelliert werden. Erzeugt ein realer Vorgang eine Sequenz von beobachtbaren Symbolen, so wird diese Emission genannt. Die Herausforderung besteht darin, ein Modell zu entwickeln, welches die Emissionen erklärt oder mit einem solchen Modell bestimmte Muster in den Emissionen zu erkennen. Das Hidden Markov Model ist ein Ansatz für ein solches Modell, welches im Folgenden basierend auf (RJ86) vorgestellt wird.

Zeitdiskreter stochastischer Prozess

Ein stochastischer Prozess mit der Indexmenge T heißt zeitdiskreter stochastischer Prozess, falls $|T| \leq \aleph_0$. In dieser Arbeit werden nur zeitdiskrete stochastische Prozesse mit $T = \mathbb{N}^+$ betrachtet.

Markov-Kette k-ter Ordnung

Sei X eine Menge von Zuständen. Ein stochastischer Prozess, bei dem die Übergangswahrscheinlichkeit von einem Zustand X_{t+k-1} in einen Folgezustand X_{t+k} nur von den letzten $k \in \mathbb{N}^+$ Zuständen $X_{t+k-1}, \ldots, X_t$ abhängt, heißt Markov-Kette k-ter Ordnung. Diese Eigenschaft wird auch Markov-Eigenschaft genannt.

$$P\left(X_{t+k} = x_{i_{t+k}} \,\middle|\, \bigwedge_{j=1}^{t+k-1} X_j = x_{i_j}\right) = P\left(X_{t+k} = x_{i_{t+k}} \,\middle|\, \bigwedge_{j=t}^{t+k-1} X_j = x_{i_j}\right)$$

Gedächtnislose Markov-Kette

Eine Markov-Kette 1. Ordnung heißt gedächtnislos. Dabei gilt:

$$P\left(X_{t+1} = x_{i_{t+1}} \,\middle|\, \bigwedge_{j=1}^{t+k-1} X_j = x_{i_j}\right) = P\left(X_{t+1} = x_{i_{t+1}} \,|X_t = x_{i_t}\right)$$

Endliche Markov-Kette

Eine Markov-Kette heißt endlich, falls die Menge möglicher Zustände endlich ist.

$$X = \{x_1, \ldots, x_n\}, \quad n \in \mathbb{N}^+$$

Anfangsverteilung

Die Verteilung $\pi \in \mathbb{R}^n, \quad n \in \mathbb{N}^+$ mit $\forall i\colon \pi_i = P(X_1 = x_i) \geq 0$ und $\sum_{i=1}^{n} \pi_i = 1$ gibt für jeden Zustand x_i an, mit welcher Wahrscheinlichkeit er als Startzustand X_1 beobachtet werden kann. Sie wird auch als Startverteilung oder Anfangsverteilung bezeichnet.

Homogene Markov-Kette

Eine gedächtnislose Markov-Kette heißt homogen oder zeitinvariant, falls die Übergangswahrscheinlichkeit vom Zustand x_i in den Zustand x_j unabhängig vom Zeitpunkt t ist.

$$\forall i, j \colon P(X_{t+1} = x_j | X_t = x_i)(t) = a_{ij}$$

Übergangsmatrix

Für gedächtnislose, homogene, zeitdiskrete Markov-Ketten mit endlichem Zustandsraum können die Übergangswahrscheinlichkeiten a_{ij} vom Zustand x_i in den Zustand x_j in einer Matrix $A \in \mathbb{R}^{n \times n}$ mit konstanter Zeilensumme 1 dargestellt werden, das heißt:

$$\sum_{j=1}^{n} a_{ij} = 1, \quad \text{wobei } a_{ij} \geq 0 \text{ für } 1 \leq i, j \leq n.$$

Diese Matrix wird Übergangsmatrix oder auch Transitionsmatrix genannt.

Beobachtungsmatrix

Die Einträge $b_{ij} = P(Y_t = y_j | X_t = x_i)$ mit

$$\sum_{j=1}^{n} b_{ij} = 1, \quad \text{wobei } b_{ij} \geq 0 \text{ und } \quad 1 \leq i, j \leq n$$

der Beobachtungsmatrix $B \in \mathbb{R}^{n \times m}$ geben die Wahrscheinlichkeit an, im Zustand x_i die Emission y_j zu erhalten.

Hidden Markov Model

Ein stochastischer Prozess wird Hidden Markov Model (HMM) genannt, falls:

- das durch System eine Markov-Kette mit unbeobachteten Zuständen modelliert wird
- und mit bestimmten Wahrscheinlichkeiten zustandsabhängige Emissionen erzeugt.

Ein HMM lässt sich als 5-Tupel $\lambda = (X, Y, A, B, \pi)$ beschreiben, wobei:

- $X = \{x_1, \ldots, x_n\}$ Menge der Zustände
- $Y = \{y_1, \ldots, y_m\}$ Menge der Emissionen
- $A \in \mathbb{R}^{n \times n}$ Übergangsmatrix
- $B \in \mathbb{R}^{n \times m}$ Beobachtungsmatrix
- $\pi \in \mathbb{R}^{n}$ Anfangsverteilung

Außerdem seien $H = (X_1, \ldots, X_l)$ und $O = (Y_1, \ldots, Y_l)$ Folgen der Länge l mit Zuständen bzw. Emissionen.
Die Abbildung 2.1 veranschaulicht einen solchen Prozess. Dabei sei $Y = \{y_1, \ldots, y_m\}$ mit $m \in \mathbb{N}^+$ die Menge aller Emissionen.

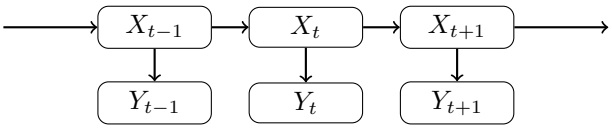


Abbildung 2.1: Hidden Markov Model

Ein Hidden Markov Model besitzt folgende Eigenschaften:

1. Markov-Eigenschaft:
Der Zustand X_{t+k} hängt nur von den letzten k Zuständen $X_{t+k-1}, \ldots, X_t$ ab:

$$P\left(X_{t+k} = x_{i_{t+k}} \,\middle|\, \bigwedge_{j=1}^{t+k-1} (X_j = x_{i_j}, Y_j = y_{i_j})\right) = P\left(X_{t+k} = x_{i_{t+k}} \,\middle|\, \bigwedge_{j=t}^{t+k-1} X_j = x_{i_j}\right)$$

2. Markov-Eigenschaft:
Die Emission Y_{t+k-1} hängt nur von den letzten k Zuständen $X_{t+k-1}, \ldots, X_t$ ab:

$$P\left(Y_{t+k-1} = y_{i_{t+k-1}} \,\middle|\, \bigwedge_{j=1}^{t+k-1} X_j = x_{i_j}, \bigwedge_{j=1}^{t+k-2} Y_j = y_{i_j}\right) = P\left(Y_{t+k-1} = y_{i_{t+k-1}} \,\middle|\, \bigwedge_{j=t}^{t+k-1} X_j = x_{i_j}\right)$$

Markov-Ketten in dieser Arbeit

Für die Algorithmen, die in dieser Arbeit betrachtet werden, ist die Existenz einer Übergangsmatrix notwendig. Daher werden im weiteren Verlauf nur gedächtnislose, homogene, zeitdiskrete Markov-Ketten mit endlichem Zustandsraum betrachtet.

Anwendungen

Hidden Markov Modelle werden häufig in der Mustererkennung eingesetzt, beispielsweise in der Spracherkennung oder der Bioinformatik zum Auffinden von CpG-Inseln in DNA-Sequenzen. Die verborgenen Zustände repräsentieren meist semantische Aussagen. Oftmals soll aus der Sequenz von Emissionen die wahrscheinlichste Aussage über die verborgenen Zustände abgeleitet werden.

2.3.1 Problemklassen

Im Zusammenhang mit HMMs gibt es drei Klassen von Problemen, die ebenfalls in (RJ86) beschrieben werden:

Evaluation-Problem

Gegeben seien eine Beobachtungssequenz O und ein HMM λ. Die Wahrscheinlichkeit $P(O|\lambda)$, dass das Modell diese Sequenz erzeugt, wird gesucht. Falls es verschiedene Modelle für eine Beobachtungssequenz gibt, wird jenes Modell gesucht, welches die Sequenz mit größter Wahrscheinlichkeit erzeugt. Dieses Problem löst der Forward-Backward-Algorithmus, der hier allerdings nicht näher betrachtet werden soll.

Decoding-Problem

Gegeben seien eine Beobachtungssequenz O und ein HMM λ. Gesucht ist die wahrscheinlichste Zustandsfolge E, mit der $P(E|O,\lambda)$ maximal wird. Dieses Problem wird vom Viterbi-Algorithmus gelöst, der in Abschnitt 2.4 im Detail betrachtet wird.

Learning-Problem

Gegeben seien eine Beobachtungssequenz O und ein HMM λ. Es werden die Modellparameter gesucht, die die Wahrscheinlichkeit $P(O|\lambda)$ maximieren. Wenn ein Modell gut zu den Trainingsdaten passt, sind auch möglichst gute Ergebnisse mit echten Daten zu erwarten. Der hier nicht näher betrachtete Baum-Welch-Algorithmus löst dieses Problem.

Beispiel

Gesucht ist die wahrscheinlichste Reiseroute E, ausgehend von täglichen Fotos in einem sozialen Netzwerk. Dies ist ein Decoding-Problem, welches mit Hilfe des Viterbi-Algorithmus gelöst werden kann.

$$
\begin{aligned}
\text{Städte:} \quad & X = \{\text{Berlin, Hamburg, München, Rostock}\} \\
\text{Aktivitäten:} \quad & Y = \{\text{Einkaufen, Fahrradtour, Geocaching, Kneipe, Schwimmen, Tanzen}\} \\
\text{Reisenwahrscheinlichkeit:} \quad & A = \begin{pmatrix} 0.5 & 0.1 & 0.1 & 0.3 \\ 0.1 & 0.5 & 0.1 & 0.3 \\ 0.3 & 0.1 & 0.5 & 0.1 \\ 0.1 & 0.3 & 0.1 & 0.5 \end{pmatrix} \\
\text{Aktivitätswahrscheinlichkeit:} \quad & B = \begin{pmatrix} 0.3 & 0.2 & 0 & 0.2 & 0 & 0.3 \\ 0.1 & 0.5 & 0 & 0.1 & 0.2 & 0.1 \\ 0.3 & 0.1 & 0.3 & 0.2 & 0 & 0.1 \\ 0.3 & 0.1 & 0.1 & 0.1 & 0.3 & 0.1 \end{pmatrix} \\
\text{Start-Stadt-Wahrscheinlichkeit:} \quad & \pi = (0.4, 0.1, 0.2, 0.3) \\
\text{Fotos mit Aktivitäten:} \quad & O = (\text{Kneipe, Geocaching, Fahrradtour, Kneipe, Schwimmen})
\end{aligned}
$$

Die Lösung zum Beispiel ist in Abschnitt 2.4.1 auf Seite 13 zu finden.

2.4 Viterbi-Algorithmus

Bei dem Viterbi-Algorithmus handelt es sich um den Standard-Algorithmus zur Lösung des Decoding-Problems eines Hidden Markov Modells.(RJ86)

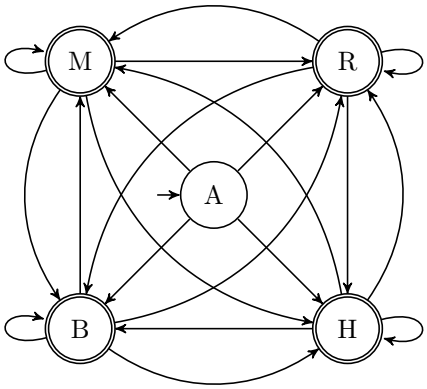


Abbildung 2.2: Zustandsübergänge

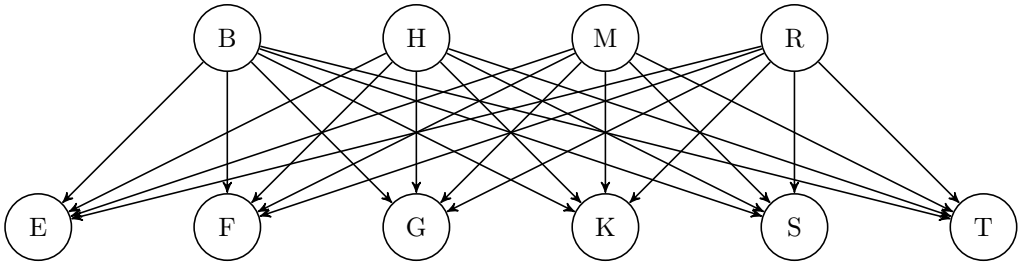


Abbildung 2.3: Emissionen

2.4.1 Algorithmus

Gesucht ist die Zustandsfolge $E = (X_1, \ldots, X_l)$, welche die Wahrscheinlichkeit $P(E|O, \lambda)$ maximiert, dass ein gegebenenes Hidden Markov Model $\lambda = (X, Y, A, B, \pi)$ eine bestimmte Beobachtungssequenz $O = (Y_1, \ldots, Y_l)$ erzeugt.
Die triviale, aber wenig effiziente Lösung des Problems besteht in der Berechnung der Pfadwahrscheinlichkeiten für sämtliche Zustandsfolgen $(X_1, \ldots, X_l)$, welche die Emissionsfolge $(Y_1, \ldots, Y_l)$ erzeugen und die Zustandsfolge mit maximaler Pfadwahrscheinlichkeit Δ auszuwählen:

$$\Delta = \max_{1 \leq v \leq n} P\left(\bigwedge_{t=1}^{l} (X_t = x_v, Y_t = O_t)\right)$$

Der Viterbi-Algorithmus basiert auf diesem Ansatz, allerdings wird die Lösung schrittweise berechnet. Anstelle des globalen Maximums P_{max} wird für jeden Zustand das Maximum zum Zeitpunkt t mit $1 \leq t \leq l$ gesucht:

$$\delta_t(i) = \max_{1 \leq v \leq n} P\left(\bigwedge_{u=1}^{t-1} X_u = x_v, X_t = x_i, \bigwedge_{u=1}^{t} Y_u = O_u\right)$$

Dieser Ansatz ermöglicht es, die Suche auf wahrscheinlichste Teilpfade zu beschränken. Ein Pfad von X_{t-1}

nach X_t braucht nur dann betrachtet werden, wenn bereits der Pfad von X_1 nach X_{t-1} optimal war. Dafür ist es jedoch notwendig, für jeden Zeitpunkt t und jeden Zustand X_i die maximale Pfadwahrscheinlichkeit $\delta_t(i)$ und den besten Vorgänger $\Psi_t(i)$ zu kennen. Verfolgt man anschließend den Pfad vom wahrscheinlichsten Endzustand X_l über die besten Vorgänger zurück, erhält man die Folge $E = (X_1, \ldots, X_l)$ mit den Zuständen, die am wahrscheinlichsten die Emissionsfolge O erzeugt haben.
Als Erstes wird daher in der Initialisierung für jeden Zustand $x_i \in X$ ermittelt, mit welcher Wahrscheinlichkeit dieser Startzustand X_1 ist und die Emission O_1 erzeugt:

$$\begin{aligned}\delta_1(i) &= P(X_1 = x_i, Y_1 = O_1) \\ &= \underbrace{P(X_1 = x_i)}_{=\pi_i} \cdot \underbrace{P(Y_1 = O_1 | X_1 = x_i)}_{=b_i(O_1)}\end{aligned}$$

Im Anschluss werden alle weiteren Pfadwahrscheinlichkeiten und die besten Vorgänger im Schritt der Rekursion ermittelt. Um diese Berechnungen für $2 \leq t \leq l$ effizient durchführen zu können, ist folgende Umformung in die rekursive Form sinnvoll.

$$\delta_t(i) := \max_{1 \leq v \leq n} P\left(\bigwedge_{u=1}^{t-1} X_u = x_v, X_t = x_i, \bigwedge_{u=1}^{t} Y_u = O_u\right) \tag{2.1}$$

$$= \max_{1 \leq v \leq n} P\left(\bigwedge_{u=1}^{t-1} X_u = x_v, X_t = x_i, \bigwedge_{u=1}^{t-1} Y_u = O_u\right) \cdot \tag{2.2}$$

$$\max_{1 \leq v \leq n} P\left(Y_t = O_t \,\middle|\, \bigwedge_{u=1}^{t-1} X_u = x_v, X_t = x_i, \bigwedge_{u=1}^{t-1} Y_u = O_u\right)$$

$$= \max_{1 \leq v \leq n} P\left(\bigwedge_{u=1}^{t-1} X_u = x_v, X_t = x_i, \bigwedge_{u=1}^{t-1} Y_u = O_u\right) \cdot \underbrace{P\left(Y_t = O_t \,|X_t = x_i\right)}_{=b_i(O_t)} \tag{2.3}$$

$$= \max_{1 \leq j \leq n} \max_{1 \leq v \leq n} P\left(\bigwedge_{u=1}^{t-2} X_u = x_v, X_{t-1} = x_j, X_t = x_i, \bigwedge_{u=1}^{t-1} Y_u = O_u\right) \cdot b_i(O_t) \tag{2.4}$$

$$= \max_{1 \leq j \leq n} \max_{1 \leq v \leq n} P\left(\bigwedge_{u=1}^{t-2} X_u = x_v, X_{t-1} = x_j, \bigwedge_{u=1}^{t-1} Y_u = O_u\right) \cdot \tag{2.5}$$

$$\max_{1 \leq v \leq n} P\left(X_t = x_i \,\middle|\, \bigwedge_{u=1}^{t-2} X_u = x_v, X_{t-1} = x_j, \bigwedge_{u=1}^{t-1} Y_u = O_u\right) \cdot b_i(O_t)$$

$$= \max_{1 \leq j \leq n} \underbrace{\max_{1 \leq v \leq n} P\left(\bigwedge_{u=1}^{t-2} X_u = x_v, X_{t-1} = x_j, \bigwedge_{u=1}^{t-1} Y_u = O_u\right)}_{=\delta_{t-1}(j)} \cdot \underbrace{P\left(X_t = x_i | X_{t-1} = x_j\right)}_{=a_{ij}} \cdot b_i(O_t) \tag{2.6}$$

$$= \max_{1 \leq j \leq n} a_{ij} \delta_{t-1}(j) \cdot b_i(O_t) \tag{2.7}$$

Von (2.1) nach (2.6) werden X_t und Y_t mit Hilfe des Multiplikationssatz $P(A \cap B) = P(A) \cdot P(B|A)$ separiert und die dazugehörigen bedingten Wahrscheinlichkeiten mit a_{ij} respektive $b_i(O_t)$ identifiziert. Dazu sind in (2.3) und (2.6) die Markov-Eigenschaften notwendig. In (2.3) wird die 2. Markov-Eigenschaft

genutzt, da Y_t nur von X_t abhängt und der Term $P\left(Y_t = O_t \left| \bigwedge_{u=1}^{t-1} X_u = x_v, X_t = x_i, \bigwedge_{u=1}^{t-1} Y_u = O_u \right.\right)$ so auf $P(Y_t = O_t | X_t = x_i) = b_i(O_t)$ schrumpft. Analog reduziert die 1. Markov-Eigenschaft den Term in (2.6) von $P\left(X_t = x_i \left| \bigwedge_{u=1}^{t-2} X_u = x_v, X_{t-1} = x_j, \bigwedge_{u=1}^{t-1} Y_u = O_u \right.\right)$ auf $P(X_t = x_i | X_{t-1} = x_j) = a_{ij}$. Da sowohl a_{ij} als auch $b_i(O_t)$ unabhängig von v sind, entfallen die Maximierungen.
Der beste Vorgänger ist daher jener Zustand X_i, der mit größter Wahrscheinlichkeit zum aktuellen Zustand X_j führt und folglich $\delta_{t-1}(j)a_{ij}$ maximiert, d.h.

$$\Psi_{t-1}(i) = \underset{1 \le j \le n}{\operatorname{argmax}} \delta_{t-1}(j)a_{ij}$$

In der Termination wird das Maximum der Pfadwahrscheinlichkeit unter allen Endzuständen gesucht und der entsprechende Vorgänger vermerkt, also

$$\Delta = \max_{1 \le i \le n} \delta_l(i), \quad \Psi_l = \underset{1 \le i \le n}{\operatorname{argmax}} \delta_l(i)$$

Zum Schluss findet die Rückverfolgung statt, bei dem ausgehend von allen Endzuständen X_{l_i} mit der größten Pfadwahrscheinlichkeit Δ die gesamten Pfade rückwärts aufgebaut werden, dass heißt

$$X_t = \Psi_{t+1}(\Psi_l), \quad 1 \le t \le l-1$$

Algorithmus

function VITERBI $(X, Y, A, B, \pi, O) : X_1$
# Initialisierung:
  for Zustand $x_i \in X, \quad 1 \le i \le n$ do
    $\delta_1(i) = \pi_i \cdot b_i(O_1)$
  end for
# Rekursion:
  for Beobachtung $O_t, \quad 2 \le t \le l$ do
    for Zustand $x_i \in X, \quad 1 \le i \le n$ do
      $\delta_t(i) = \max_{1 \le j \le n} a_{ij}\delta_{t-1}(j) \cdot b_i(O_t)$
      $\Psi_{t-1}(i) = \underset{1 \le j \le n}{\mathtt{argmax}} \delta_{t-1}(j)a_{ij}$
    end for
  end for
# Termination:
  $\Delta = \max_{1 \le i \le n} \delta_l(i)$
  $\Psi_l = \underset{1 \le i \le n}{\mathtt{argmax}} \delta_l(i)$
# Verfolgung der Zustandssequenz:
  for Zeitpunkt $t, \quad l-1 \ge t \ge 1$ do
    $X_t = \Psi_{t+1}(\Psi_l)$
  end for

```
  return X_1
end function
```

Viterbi-Algorithmus am Beispiel

Die Tabelle 2.1 enthält noch einmal die Ausgangssituation des Problems.

Städte:	$X = \{\text{Berlin, Hamburg, München, Rostock}\}$
Aktivitäten:	$Y = \{\text{Einkaufen, Fahrradtour, Geocaching, Kneipe, Schwimmen, Tanzen}\}$
Reisewahrscheinlichkeit:	$A = \begin{pmatrix} 0.5 & 0.1 & 0.1 & 0.3 \\ 0.1 & 0.5 & 0.1 & 0.3 \\ 0.3 & 0.1 & 0.5 & 0.1 \\ 0.1 & 0.3 & 0.1 & 0.5 \end{pmatrix}$
Aktivitätswahrscheinlichkeit:	$B = \begin{pmatrix} 0.3 & 0.2 & 0 & 0.2 & 0 & 0.3 \\ 0.1 & 0.5 & 0 & 0.1 & 0.2 & 0.1 \\ 0.3 & 0.1 & 0.3 & 0.2 & 0 & 0.1 \\ 0.3 & 0.1 & 0.1 & 0.1 & 0.3 & 0.1 \end{pmatrix}$
Start-Stadt-Wahrscheinlichkeit:	$\pi = (0.4, 0.1, 0.2, 0.3)$
Fotos mit Aktivitäten:	$O = (\text{Kneipe, Geocaching, Fahrradtour, Kneipe, Schwimmen})$

Tabelle 2.1: Beispiel Viterbi-Algorithmus

In der Initialisierung werden für jeden Zustand die Wahrscheinlichkeit die Emission O_1 zu erzeugen und die Wahrscheinlichkeit Anfangszustand zu sein, multipliziert.

$$\begin{pmatrix} 0.2 \cdot 0.4 \\ 0.1 \cdot 0.1 \\ 0.2 \cdot 0.2 \\ 0.1 \cdot 0.3 \end{pmatrix} = \begin{pmatrix} 0.08 \\ 0.01 \\ 0.04 \\ 0.03 \end{pmatrix}$$

Als nächstes folgt die Rekursion, wodurch sich die Matrix δ schrittweise mit neuen Spalten füllt.

$$\delta = \begin{pmatrix} \mathbf{0.08} & 0 & \mathbf{0.00036} & \mathbf{0.0000360} & 0 \\ 0.01 & 0 & \mathbf{0.00036} & 0.0000180 & 0.00000180 \\ 0.04 & \mathbf{0.0060} & 0.00030 & 0.0000300 & 0 \\ 0.03 & 0.0024 & 0.00012 & 0.0000108 & \mathbf{0.00000324} \end{pmatrix}$$

Im Schritt der Termination wird in der letzten Spalte von δ nach dem Maximum und dem dazugehörigen Zustand gesucht. Dieses ist im Beispiel der Wert 0.00000324 für Rostock.
Zuletzt werden von diesem Zustand ausgehend all jene Vorgänger mit der größten Wahrscheinlichkeit gesucht. Dies führt dann schließlich zum Ergebnis.

$$\text{Berlin} \rightarrow \text{Rostock} \rightarrow \text{Berlin oder Hamburg} \rightarrow \text{Berlin} \rightarrow \text{Rostock}$$

Numerische Stabilität

Der Viterbi-Algorithmus erfordert viele Multiplikationen von Zahlen zwischen 0 und 1, sodass eine naive Implementierung numerisch instabil ist und viele Werte produziert, die sich asymptotisch der Null nähern. Um diesem Problem entgegenzuwirken, wird die Monotonie des Logarithmus genutzt (RJ86) und die

Wahrscheinlichkeiten werden logarithmiert. Dabei werden die Werte betragsmäßig größer und aus den Multiplikationen werden einfachere Additionen. Daraus folgt demnach

$$\begin{aligned}\hat{\delta}_t(i) &:= \max_{1\leq v\leq n} \log P\left(\bigwedge_{u=1}^{t-1} X_u = x_v, X_t = x_i, \bigwedge_{u=1}^{t} Y_u = O_u\right) \\ &= \max_{1\leq j\leq n} \left(\log\left(a_{ij}\right) + \delta_{t-1}(j)\right) + \log b_i(O_t)\end{aligned}$$

Der Fall $log(p = 0)$ wird von vornherein ausgeschlossen, da solche unmöglichen Pfade nicht zielführend sind.

Kapitel 3

Technologien

In diesem Kapitel wird ein kurzer Überblick über einige Datenbank-Technologien und Plattformen gegeben. Dazu wird im Abschnitt 3.1 auf zwei verschiedene Typen relationaler Datenbanken eingegangen, die sich hinsichtlich der Art, wie Daten gespeichert werden, unterscheiden. Im Abschnitt 3.2 werden Architekturen betrachtet, die auf mehr als einem Rechner laufen. Dies führt zwangsläufig auch zur Fragestellung, wie die Daten über die verschiedenen Rechner verteilt werden können. Abschnitt 4.1 gibt einen Überblick über die Entwicklung und den Zusammenhang dieser Plattformen. Danach werden die konkreten Plattformen Actian Vector in Hadoop in Abschnitt 4.2, Postgres-XL in Abschnitt 4.3 und Apache Spark in Abschnitt 4.4 vorgestellt.

3.1 Relationale Datenbank

In den 1960er Jahre wurden Daten üblicherweise in elementaren Dateien abgelegt mit anwendungsspezifischer Datenorganisation. Diese Art der Datenspeicherung führte leicht zu inkonsistenten, geräteabhängigen und redundanten Datenbeständen. Als Lösung wurde 1970 von Codd[1] das relationale Datenbankmodell (Cod70) vorgeschlagen. Auf dessen Grundlage wurden im folgenden Jahrzehnt erste Datenbanksysteme entwickelt, die für eine redundanzfreie und konsistente Datenhaltung in einer zentralisierten, relationalen Datenbank sorgten. Somit konnten unterschiedliche Anwendungen und Geräte gleichzeitig auf einen gemeinsamen, konsistenten Datenbestand zugreifen. Diese Art der Datenbankverarbeitung ist schematisch in Abbildung 3.1 dargestellt. Die ersten kommerziellen, relationalen Datenbanksysteme waren System R von Codd und Ingres von Stonebraker[2]. Mit den Codd'schen Regeln (Cod82) wurden 1982 wesentliche Eigenschaften von Datenbankmanagementsystemen definiert.(Lam94, SSH18)
Als relationale Datenbank versteht man ein weit verbreitetes Datenbankmodell. Die theoretische Grundlage für dieses Modell bildet eine relationale Algebra (SSH18). Auf dieser lassen sich verschiedene Operationen definieren, die auf Mengen von Relationen angewendet werden können. Die Abgeschlossenheit der Relationenalgebra garantiert dabei, dass das Resultat einer Mengenoperation wieder eine Relation ist. Anfragen an die Datenbank lassen sich in einer deklarativen Sprache wie SQL formulieren.

[1] Edgar Frank „Ted“ Codd, `https://amturing.acm.org/award_winners/codd_1000892.cfm`
[2] Mike Stonebraker, `https://www.csail.mit.edu/person/michael-`

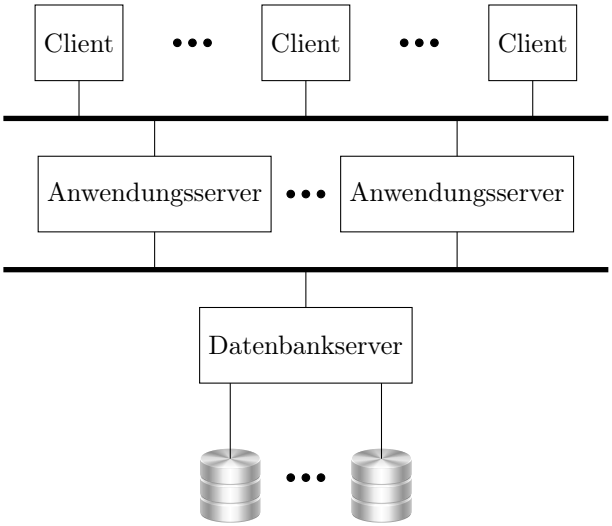


Abbildung 3.1: Zentralisierte Datenbankverarbeitung(Kud15)

Anschaulich ist eine relationale Datenbank nichts Anderes als eine Ansammlung von Tabellen. Die erste Zeile einer Tabelle, auch Kopf genannt, repräsentiert die Struktur der Tabelle. Die Spaltenüberschriften werden Attribute genannt und einzelne Zeilen heißen Tupel.(SSH18, SSH11)

Beispiel:

Lesegeschwindigkeiten

Hardware	Lesegeschwindigkeit in MB/s	Zugriffszeit in μs
Festplatte	210	12 000
RAM	25 600[3]	0.06
Prozessor-Cache		0.001

Tabelle 3.1: Beispiel Tabelle „Lesegeschwindigkeit“(SSH11, wd)

Zur Speicherung der Datenbank-Tabellen gibt es zwei unterschiedliche Möglichkeiten. Die Tabelle kann entweder zeilen- oder spaltenweise in den Speicher geschrieben werden. Dabei bedeutet eine zeilenweise Speicherung, dass alle Datenwerte einer Zeile hintereinander im Speicher liegen. Bei der spaltenweise Speicherung werden hingegen alle Werte eines Attributes hintereinander im Speicher abgelegt. Eine Datenbank, die ihre Tabellen zeilenweise speichert, heißt zeilenorientierte Datenbank. Analog heißt eine Datenbank spaltenorientiert, wenn sie ihr Tabellen spaltenweise in den Speicher schreibt. Die Wahl der Speicherart sollte vom typischen Einsatzzweck der Datenbank abhängig sein.

[3]DDR4-3200: 400 MHz · 64 Bit = 25 600 MB/s

3.1.1 Zeilenorientierte Datenbank

Die zeilenorientierte Architektur basiert auf der Idee, dass zusammenhängende Daten auch zusammenhängend gespeichert werden. Bezogen auf das Beispiel aus Tabelle 3.1 würden die Daten als folgende Tupel gespeichert
(Festplatte; 210; 12 000), (RAM; 25 600; 0.06), (Prozessor-Cache;;0.001)

3.1.2 Spaltenorientierte Datenbank

Spaltenorientierte Datenbanken speichern ihre Daten spaltenweise, sodass ein Zugriff auf alle Werte eines Attributes schnell ist. Dies ist besonders dann wichtig, wenn man sich nicht für den einzelnen Datensatz, sondern für ein Aggregat eines Attributes interessiert. Anders als bei zeilenorientierten Datenbanken können alle Werte des entsprechenden Attributes direkt hintereinander gelesen und verarbeitet werden, was die Performanz entsprechender Abfragen steigert.

Beispiel:

(Festplatte; RAM; Prozessor-Cache), (210; 25 600;), (12 000; 0.06; 0.001)

3.1.3 Vergleich

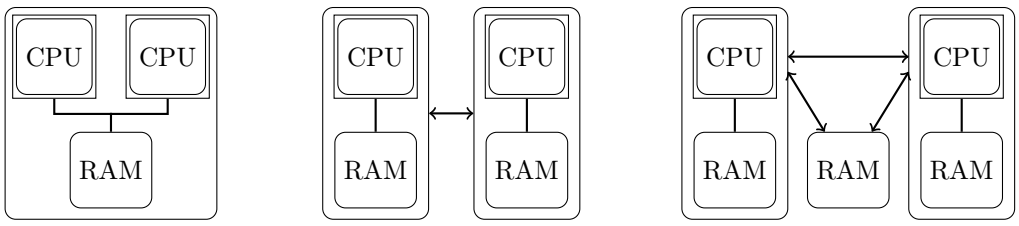
Bei der Verarbeitung großer Datenmengen kommt es vor allem auf die Effizienz des Datenzugriffes an. Insbesondere der Zugriff auf Sekundärspeicher ist vergleichsweise langsam, wie die Übersicht in Tabelle 3.1 zeigt. Das Vorhalten möglichst vieler, relevanter Daten im Primärspeicher oder idealerweise im Prozessor-Cache kann die Verarbeitung enorm beschleunigen.
Da sämtliche Spaltendaten den Datentyp des Attributes besitzen, gibt es für spaltenorientierten Datenbanken die Möglichkeiten einer effektiven Kompression. Diese lässt sich durch eine Sortierung und Ausnutzung von Index-Strukturen wie einem Bitmap-Index verbessern und führt zu einer Reduzierung des Speicherverbrauchs. Die CPU muss häufig auf die langsamen Speicher warten. Daher kann sich eine nicht zu starke Kompression positiv auf die Performanz auswirken, weil mehr Daten in kürzerer Zeit gelesen werden können. Wichtig dafür ist jedoch, dass die Dekomprimierung der Daten nicht länger als das Lesen des nächsten Datensatz von der Platte dauert.
Die spaltenorientierte Speicherarchitektur hat daher Vorteile für Anwendungen, bei denen Operationen auf vielen oder allen Tupeln angewendet oder Aggregate über Attribute gebildet werden, da nicht alle Spalten, sondern nur die entsprechen Spalten gelesen werden müssen. Insbesondere ist diese für OLAP-Aufgaben geeignet, da diese durch eine kleine Anzahl sehr komplexen Abfragen über alle Datensätze charakterisiert sind.
Zeilenorientierte Datenbanksysteme sind effizienter, wenn ein Zugriff auf vielen Spalten, aber nur wenigen Zeilen notwendig ist, da das gesamte Tupel mit einem einzigen Plattenzugriff gelesen werden kann. Auch beim Einfügen eines neuen Tupels sind zeilenorientierte Systeme schneller, da alle Daten dieser Zeile mit einem einzigen Plattenzugriff geschrieben werden können. Diese Systeme sind daher besonders gut für OLTP-Aufgaben mit vielen interaktiven Transaktionen geeignet.

3.2 Mehrrechner-Datenbanksysteme

Es gibt Umgebungen, in denen relationale Datenbanksysteme auf *einem* zentralen System nicht die gewünschten Anforderungen hinsichtlich Leistungsfähigkeit, Verfügbarkeit, Skalierbarkeit oder Verteilung der Daten erfüllen können. Dies kann beispielsweise in dezentralen Organisationsstrukturen der Fall sein, wenn die Daten nur in dem Unternehmensteil gespeichert werden dürfen, in dem sie anfallen oder eine Analyse der Daten zeitkritisch ist und ein einzelnes System nicht die notwendige Performanz bietet.
In solchen Fällen kommen Datenbanksysteme zum Einsatz, an denen mehrere Rechner beteiligt sind. Diese werden **Mehrrechner-Datenbanksysteme** (MRDBS) genannt und können durch Verteilung von Daten und Berechnungen eine höhere Gesamtleistung erreichen oder zu einer Reduzierung von Kosten führen. Der komplexere Aufbau eines MRDBS führt allerdings auch zu einem gewissen Kommunikationsaufwand zwischen den Rechnern und zu einem erhöhten Systemadministrationsaufwand, der Zeit kosten kann. Des Weiteren ist eine Lastbalancierung sinnvoll, damit vorhandene Ressourcen effizient genutzt werden können. MRDBS werden sowohl hinsichtlich ihrer räumlichen Verteilung als auch ihrer Architektur unterschieden.(RSS15, Kud15)
Die folgenden Unterabschnitte und Abbildungen basieren auf (Kud15).

Kopplungsarten

Rechnerkopplungen werden in enge, lose und nahe Kopplungen unterschieden. Eine enge Kopplung ist durch einen gemeinsamen Hauptspeicher charakterisiert, wie es für Multikern-Prozessoren typisch ist. Eine lose Kopplung zeichnet sich dadurch aus, dass jeder Prozessor seinen eigenen Hauptspeicher hat. Eine nahe Kopplung ist eine Mischform aus enger und loser Kopplung, bei der sowohl jeder Prozessor seinen eigenen Hauptspeicher besitzt als auch auf einen gemeinsamen Hauptspeicher zugreifen kann. Abbildung 3.2 stellt diese drei Kopplungsarten schematisch dar.



Enge Kopplung Lose Kopplung Nahe Kopplung

Abbildung 3.2: Kopplungsarten

Stonebraker-Klassifikation

Die Stonebraker-Klassifikation unterscheidet drei grundlegende Architekturen von Mehrrechner-Datenbanksysteme anhand der Kopplungsart und der Anbindung an den Sekundärspeicher. Diese sind in Tabelle 3.2 aufgeführt.

Klassifikation	Rechnerkopplung	Sekundärspeicher
Shared Everything	eng	gemeinsam
Shared Disks	nahe oder lose	gemeinsam
Shared Nothing	nahe oder lose	eigene

Tabelle 3.2: Stonebraker-Klassifikation

Diese Architekturen werden in Abbildung 3.3 veranschaulicht. Die Abbildung 3.4 zeigt einen Entscheidungsbaum zur Klassifizierung der Architekturen, wobei ortsverteilte MRDBS gerade die verteilten Datenbanken und lokal verteilte MRDBS die parallelen Datenbanken sind.

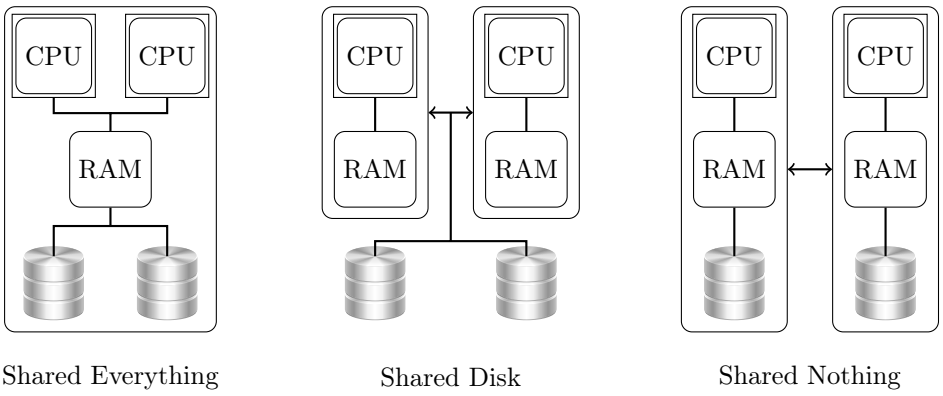


Abbildung 3.3: Architekturen nach der Stonebraker-Klassifikation

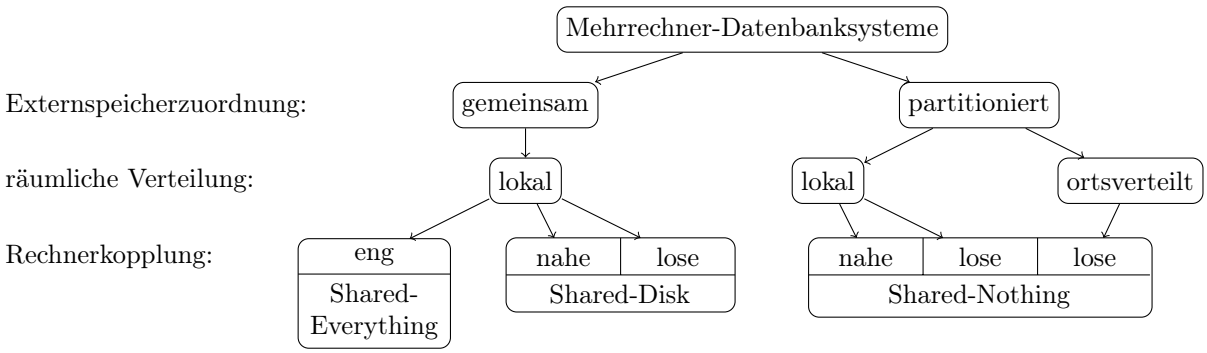


Abbildung 3.4: Entscheidungsbaum zur Klassifikation der MRDBS

3.2.1 Parallele Datenbanken

Parallele Datenbanken speichern Daten *lokal* über mehrere Rechner verteilt oder repliziert. Außerdem laufen notwendige Berechnungen möglichst parallel ab. Sie werden verwendet, um eine Leistungssteigerung durch die Verwendung von Parallelrechner zu erzeugen. Dabei wird das Verhältnis der Antwortzeiten bei sequenzieller Verarbeitung gegenüber paralleler Verarbeitung auf n Prozessoren als Speedup(n) bezeich-

net. Analog wird die Steigerung der Transaktionsraten durch n Parallelrechner im Folgenden Scaleup(n) genannt.
In Abbildung 3.5 ist eine parallele Datenbankverarbeitungen mit Shared Disk schematisch dargestellt. Analog zeigt Abbildung 3.6 eine mit Shared Nothing.

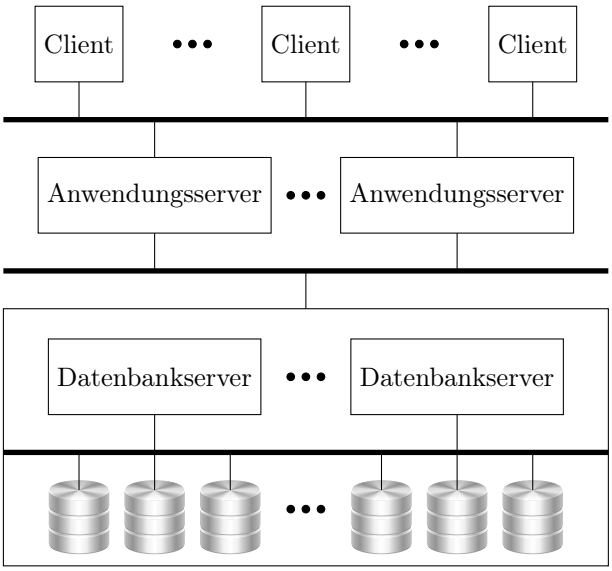


Abbildung 3.5: Parallele Datenbankverarbeitung mit Shared Disk

Parallelverarbeitung

Die Datenverarbeitung in parallelen Datenbanken kann auf unterschiedlichen Ebenen parallelisiert werden. Im wesentlichen wird zwischen Transaktions-, Anfrage- und Operatorparallelität unterschieden. Diese können jedoch durch Pipelining, Datenpartitionierung und parallele Ein-/Ausgabe weiter optimiert werden. Abbildung 3.7 veranschaulicht die Ebenen der Parallelverarbeitung.
Als *Inter-Transaktionsparallelität* wird die Eigenschaft eines DBMS bezeichnet, mehrere unabhängige Transaktionen parallel ausführen zu können. Dies ermöglicht sowohl einen Mehrbenutzerbetrieb als auch ein akzeptables Durchsatzverhalten und wird bereits von zentralisierten Datenbanksystemen und allen MRDBS unterstützt.
Die Ausführung von komplexen Transaktionen oder Operationen auf sehr großen Datenmengen kann viel Zeit kosten. Zur Reduzierung dieser Antwortzeiten können parallele Datenbanksysteme Aktionen innerhalb von Transaktionen parallelisieren. Diese Eigenschaft heißt *Intra-Transaktionsparallelität.*
Intra-Transaktionsparallelität lässt sich in Inter-Query-Parallelität und Intra-Query-Parallelität unterteilen. Dabei spricht man von *Inter-Query-Parallelität*, wenn sequenzielle Queries einer Transaktion durch parallel ausführbare Teiltransaktionen ersetzt werden. Die Parallelisierbarkeit wird dabei durch Ausführungsabhängigkeiten und nicht parallelisierbare Operationen eingeschränkt. Werden die Operationen innerhalb einer Query parallelisiert, spricht man von *Intra-Query-Parallelität.*
Eine Query kann aus mehreren Basisoperatoren wie Selektion, Projektion oder Join bestehen, deren

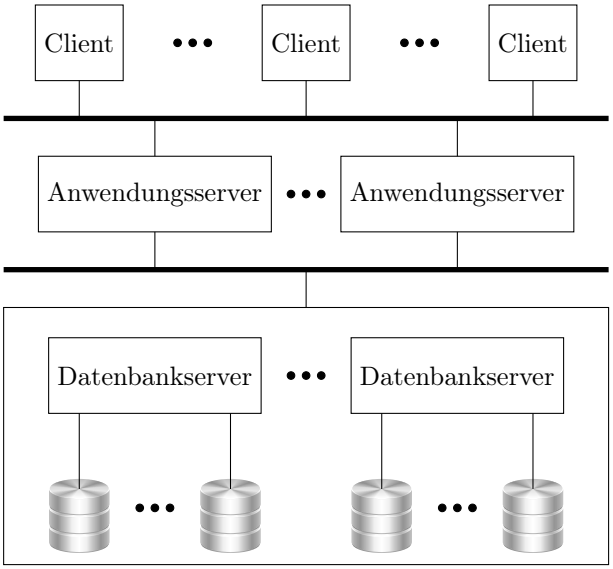


Abbildung 3.6: Parallele Datenbankverarbeitung mit Shared Nothing

Ausführungsreihenfolge durch einen Operatorbaum beschrieben werden kann. Die Parallelisierung mehrerer Operatoren innerhalb einer Query wird als *Inter-Operatorparallelität* bezeichnet. Wird hingegen ein Basisoperator parallelisiert, heißt dies *Intra-Operatorparallelität*.
Zusätzlich können die Daten partitioniert werden, sodass parallel auf den Datenpartitionen gearbeitet werden kann. Dies wird *Datenparallelität* oder *horizontale Datenflussparallelität* genannt. Der Parallelitätsgrad kann dabei sehr gut zur Relationengröße skaliert werden.(RSS15)
Bei der *Pipelineparallelität*, oder auch *vertikale Datenflussparallelität* genannt, werden verschiedene Operatoren überlappend ausgeführt, sodass die Ausführungszeit kürzer als bei einer sequenziellen Ausführung ist. Die Ausgaben eines Operators werden im Datenflussprinzip direkt an einen anderen weitergeleitet. Es wird nicht gewartet, bis der Erzeuger-Prozess die gesamte Ausgabe produziert hat, sondern produzierte Ergebnisse werden fortlaufend weitergeleitet, um eine frühzeitige Weiterverarbeitung zu ermöglichen. Dabei kann es jedoch zu einem erheblichen Kommunikationsoverhead kommen.(RSS15)

3.2.2 Verteilte Datenbanken

Verteilte Datenbanken (VDBS) gehören zur Klasse ortsverteilter Shared-Nothing-Systeme und kommen zum Einsatz, wenn Daten *nicht lokal* verteilt oder repliziert werden sollen. Dies können beispielsweise personenbezogene oder ortsgebundene Daten sein, die in der Region gespeichert werden sollen, in der diese benötigt werden oder anfallen. Ein anderer, wichtiger Anwendungsfall ist die Datenredundanz, bei der zur Ausfallsicherung die Daten über mehrere Rechenzentren gespiegelt werden. Die Datenbank bleibt so selbst dann noch arbeitsfähig, wenn ein Rechenzentrum ausfällt. Die Abbildung 3.8 stellt eine verteilte Datenverarbeitung schematisch dar.
In Tabelle 3.3 sind Bewertungen wesentlicher Eigenschaften paralleler und verteilter Datenbanksysteme

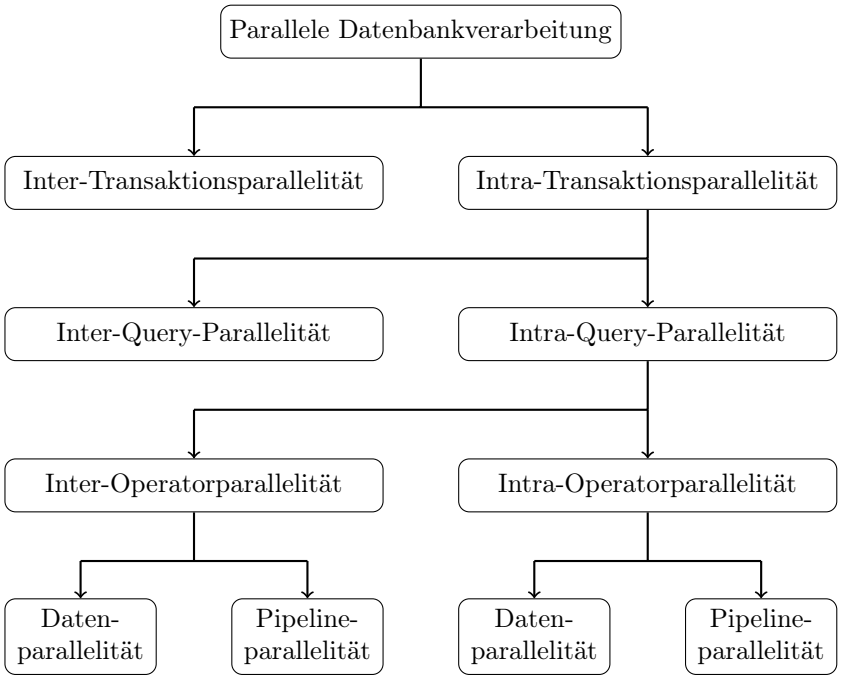


Abbildung 3.7: Arten der parallelen Datenbankverarbeitung

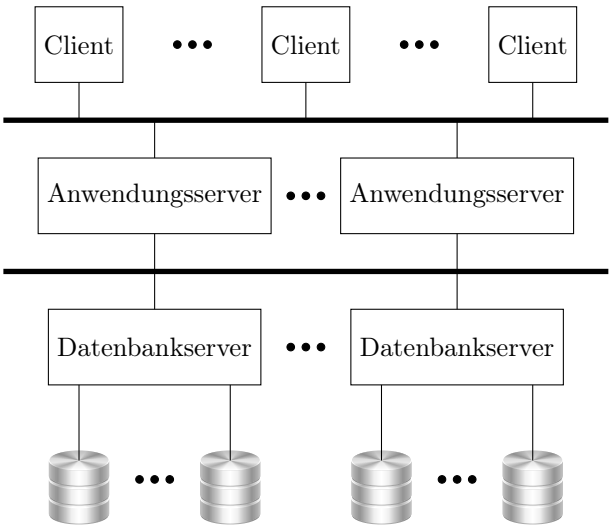


Abbildung 3.8: Verteilte Datenbankverarbeitung

	Parallele DBS	Verteilte DBS
Hohe Transaktionsraten	++	○/+
Intra-Transaktionsparallelität	++	○/+
Skalierbarkeit	+	○/+
Verfügbarkeit	+	+
Verteilungstransparenz	++	+
Geografische Verteilung	-	+
Knotenautonomie	-	○
Heterogene Daten	-	-
Administration	○	-

Tabelle 3.3: Grobe Bewertung der Architekturalternativen (RSS15)

nach (RSS15) enthalten. Für Performanz-Tests sind daher besonders die parallelen Datenbanksysteme von Interesse.

3.2.3 Fragmentierung und Replikation

Horizontale Fragmentierung

Als *horizontale Fragmentierung* wird die Zerlegung einer Relation durch Selektion in Teilrelationen unter Wahrung von Rekonstruierbarkeit, Vollständigkeit und Disjunktheit verstanden.

Vertikale Fragmentierung

Die Zerlegung einer Relation durch Projektion in Teilrelationen unter Wahrung von Rekonstruierbarkeit und Vollständigkeit heißt *vertikale Fragmentierung*. Die Rekonstruierbarkeit kann dabei durch zusätzliche Übernahme des Primär- oder Surrogatschlüssels in allen Teilrelationen erzeugt werden. Die entstehenden Teilrelationen sind disjunkt, abgesehen von den redundanten Identifikatoren für die Rekonstruierbarkeit.

Replikation

Unter einer Replikation versteht man das redundante Vorhalten der Daten in mehreren Datenbankinstanzen.

Kapitel 4

Systeme

In diesem Kapitel werden konkrete Plattformen vorgestellt.
Als zeilenorientiertes System wird Postgres-XL[1] verwendet, da das System sowohl frei, quelloffen, weit verbreitet, als auch nahe am SQL-Standard ist und viel Einblicke in die interne Arbeit zulässt. Postgres-XL basiert auf PostgreSQL, welches sich selbst als „die fortschrittlichste Open-Source-Datenbank der Welt“ (pos) bezeichnet. Actian Vector in Hadoop[2] wird als spaltenorientiertes System zum Einsatz kommen. Als schnellstes DBMS auf dem Markt (tpc, avh, Bon), ist es ein guter Vergleichskandidat. Ein Vergleich beider Systeme mit Apache Spark als typische Plattform für Big-Data-Analyse kann in späteren Arbeiten sinnvoll sein. Daher wird in Abschnitt 4.4 die grobe Architektur von Apache Spark vorgestellt.

4.1 Entwicklung von Actian Vector

Actian Vector stammt im Wesentlichen von zwei unterschiedlichen Projekten ab, wie es in Abbildung 4.1 dargestellt ist. Eines davon ist MonetDB. Es ist ein leistungsstarkes, spaltenorientiertes RDBMS, welches seit 2002 zunächst als Monet im Rahmen eines Forschungsprojektes an der Universität Amsterdam entwickelt und 2004 als Open-Source Software MonetDB frei zugänglich gemacht wurde. In den Jahren von 2003 bis 2008 wurden für MonetDB im X100-Projekt verschiedene Technologien zur Leistungsoptimierung auf modernen Prozessoren erprobt. Als Resultat verfügte MonetDB/X100 über eine beschleunigte Abfrageverarbeitung unter Ausnutzung der vertikalen Speicherfragmentierung und des CPU-Caches. Daraus entstand im Jahr 2008 ein Startup, welches MonetDB/X100 ab 2010 als kommerzielles Produkt unter dem neuen Namen VectorWise vertrieb. Bereits 2011 wurde VectorWise von der Ingres Corporation übernommen und integrierte VectorWise in die eigene Ingres-Datenbank. Das resultierende Produkt wurde 2014 in Actian Vector umbenannt. Im Jahr 2015 folgte mit Actian Vector in Hadoop (VectorH) eine spezielle Version für Computercluster, die für Massiv-Parallel-Processing (MPP) ausgelegt war und auf Hadoop bzw. HDFS aufbaut.(mon, Bon, HNZB07, ZNB08, IZB11, SBZ12)
Die Entwicklungslinien von Actian Vector in Hadoop, Postgres-XL und Apache Spark werden in der

[1] `https://www.postgres-xl.org/`
[2] https://www.actian.com/analytic-database/vectorh-sql-hadoop/

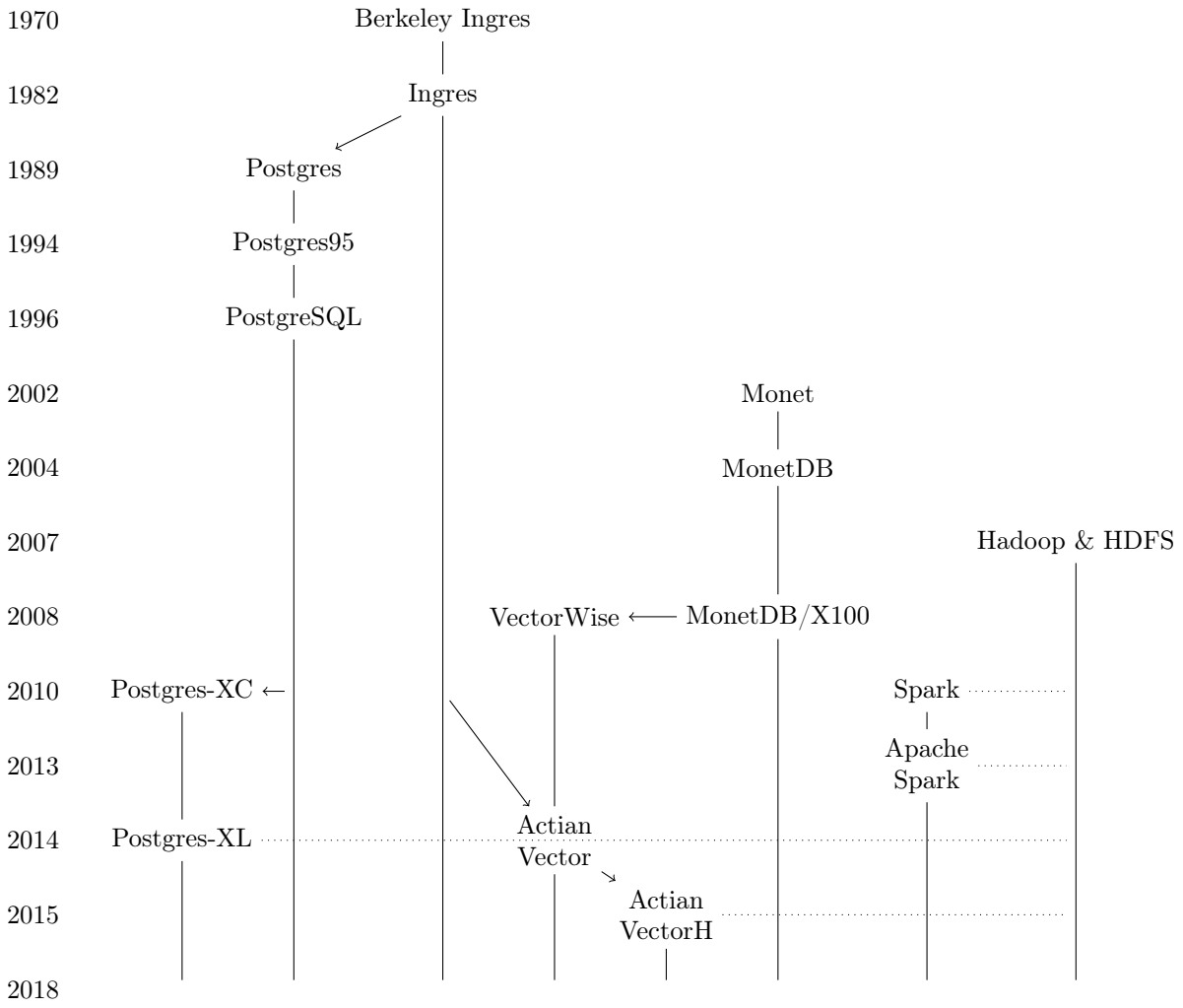


Abbildung 4.1: Entwicklungsgeschichte der Plattformen nach (Nau, mon, Bon, IZB11)

Genealogie in Abbildung 4.1 dargestellt. Zusätzlich zeigt Abschnitt C.1 auf Seite 46 die Entwicklungsgeschichte relationaler Datenbanken nach (Nau).

4.2 Actian Vector und Vector in Hadoop

Actian Vector (früher VectorWise) ist ein leistungsstarkes (tpc), spaltenorientiertes, relationales SQL-Datenbank-Managementsystem der Firma Actian (Palo Alto, Kalifornien), das für eine hohe Leistung in analytischen Datenbankanwendungen entwickelt wurde. Es nutzt eine vektorisierte Abfrageausführung und ein mehrstufiges In-Memory-Datenmanagement zur Optimierung der analytischen Aufgaben. Dank der X100-Abfrage-Engine (MBK00, BZN05) wird ein Interpretations-Overhead reduziert und es werden leistungskritische Merkmale moderner Prozessoren ausgenutzt wie Superskalarität und Single Instruction, SIMD-Anweisungen[3]. Eine SIMD-Anweisung wendet eine einzelne Operation auf einen Vektor von mehreren Datenwerten in einem einzigen Befehlszyklus an. Dies wird in Actian Vector genutzt, um mit einem Operator einen Teil einer Spalte auf einmal in sogenannten Chunks zu verarbeitet (SSH11). Dies beschleunigt die Verarbeitung der Daten enorm (avh). Eine bessere Ausnutzung des Prozessor-Caches führt zur Reduzierung des Datenaustausches mit dem Hauptspeicher, was besonders wichtig bei modernen Vielkern-Prozessoren ist.(ZNB08, IZB11, HNZB07, SBZ12, mon, Bon)
Actian Vector in Hadoop (Actian VectorH) ist die Variante von Actian Vector, die nativ auf einem Hadoop-Cluster läuft. Actian VectorH unterstützt eine Integration von Apache Spark mittels Spark-Vector Konnektor und kann direkt das HDFS als Speicher nutzen.(avh)

4.3 PostgreSQL und Postgres-XL

Als zeilenorientierte relationale Datenbankmanagementsysteme kommen PostgreSQL und Postgres-XL[4] zum Einsatz. Postgres-XL ist die lokal verteilte Variante (paralleles RDBMS) von PostgreSQL (pgx), welches sich selbst als „die fortschrittlichste Open-Source-Datenbank der Welt“ (pos) bezeichnet und weit verbreitet ist. Da beide Systeme sowohl frei, quelloffen, als auch nahe am SQL-Standard sind und viele Einblicke in die interne Arbeit zulassen, eignen sie sich besonders gut für die nicht-kommerzielle Nutzung wie in dieser Arbeit.
Die folgenden Unterabschnitte basieren auf (EH13).

4.3.1 Multiversion Concurrency Control (MVCC)

Multiversion Concurrency Control ermöglicht es, dass jeder Benutzer zu einem bestimmten Zeitpunkt einen eigenen Snapshot der Datenbank sieht (Stu16). Dazu werden intern mehrere Versionen eines Objektes gehalten. Diese werden durch fortlaufend erhöhte Transaktionsnummern voneinander unterschieden. Dabei gelten folgende Eigenschaften.

[3] Single Instruction, Multiple Data
[4] `https://www.postgres-xl.org/`

Standard:	32 MB
Minimal:	128 kB oder 16·max_connections
Maximal:	SHMMAX[5]
Empfohlen:	10 bis 25 % des Arbeitsspeicher

Tabelle 4.1: Werte für den Shared-Buffer-Pool

- Lesezugriff blockiert niemals parallelen Schreibzugriff
- Lesezugriff muss niemals auf parallelen Schreibzugriff warten

Intern wird jeder Tabelle ein *Xmin* und ein *Xmax* zugewiesen. Für jedes Tupel wird in *Xmin* die Transaktionsnummer gespeichert, welche das Tupel erzeugt hat. *Xmax* ist solange *Null*, wie das Tupel gültig ist. Andernfalls enthält *Xmax* die Transaktionsnummer, welche das Tupel wegen einer Update- oder Delete-Operation als ungültig markiert hat. Wenn ein Tupel durch eine Operation mit der Transaktionsnummer *Xid* geändert wird, wird bei dem alten Tupel *Xmax* auf *Xid* gesetzt und eine neue Version des Tupels mit *Xmin*=*Xid* und *Xmax*=*Null* erzeugt.
Für eine Transaktion mit Transaktionsnummer *Xid* ist ein Tupel t sichtbar, falls

$$t[Xmin] \leq Xid$$

und

$$(x[Xmax] \text{ IS NULL oder } t[Xmax] > Xid)$$

Das Tupel t wurde vor oder in Transaktion *Xid* erzeugt und aus Sicht der Transaktion *Xid* noch nicht gelöscht.
Durch MVCC werden viele Datensätze gespeichert, die bereits nicht mehr gültig sind. Diese werde mit VACUUM endgültig gelöscht. Dieser wird daher periodisch ausgeführt, kann aber auch manuell angestoßen werden. Vacuum kümmert sich auch um die korrekte Verwaltung der Transaktionsnummern. Diese werden als 32 bit Zahl gespeichert. Dies kann den Effekt haben, dass Tupel, welche sehr alt sind, plötzlich aus der Zukunft zu kommen scheinen. VACUUM markiert diese alten Tupel, sodass sie beim Sichtbarkeitscheck richtig behandelt werden.

4.3.2 Parameter

In der Konfigurationsdatei von PostgreSQL und Postgres-XL gibt es eine Vielzahl von möglichen Parametern zur Anpassung des Systems auf die konkrete Umgebung. Im Folgenden werden einige wichtige Parameter erläutert.

shared_buffers

Mit dem Parameter shared_buffers kann die Größe des Shared-Buffer-Pools eingestellt werden. Die Tabelle 4.1 fasst die Eckdaten für die Einstellung des Shared-Buffer-Pools zusammen.

[5] Maximale Speichergröße, die von einem Prozess reserviert werden darf

Standard:	8 MB
Empfohlen:	8 MB

Tabelle 4.2: Werte für den Buffer-Pool für temporäre Tabellen

Standard:	1 MB
Empfohlen (OLTP-Anwendung):	8 MB
Empfohlen (OLAP-Anwendung, Data Mining):	>8 MB

Tabelle 4.3: Werte für den work_mem

Hierbei sollte beachtet werden, dass 32 MB nur für Test- und Minimalsysteme geeignet sind. Für nur lesende Aufgaben reichen 10% des Arbeitsspeichers aus. Je höher die Schreiblast, desto höher sollte dieser Wert gewählt werden. Schreibvorgänge in Datenbanktabellen und -indexe können gepuffert und häufiges Ausschreiben vom Shared-Buffer-Pool auf das Dateisystem vermieden werden. Ein zu hoher Wert sorgt dafür, dass wenig Speicher für das Betriebssystem für Dateisystempuffer übrig bleibt.

temp_buffers

Mit dem Parameter temp_buffers kann der Buffer-Pool für temporäre Tabellen eingestellt werden. Dieser Wert gibt die Obergrenze je Datenbankverbindung für temporäre Tabellen als Pufferspeicher an. Der Speicher wird dabei nach Bedarf in 8 kB Blöcken genommen. Die sinnvollen Größen sind in Tabelle 4.2 enthalten.
Dieser Wert sollte so klein wie möglich gewählt werden, allerdings immer so, dass die Menge der Daten, die in temporäre Tabellen abgelegt werden soll, inklusive entsprechender Indexe, vollständig hinein passen. Falls der Buffer-Pool leer ist, kann dieser im laufenden Betrieb mit *SET temp_buffers TO 'VALUE';* beliebig angepasst werden.

work_mem

Mit Obergrenze der Größe des zur Verfügung stehenden Hauptspeichers für Datenbankoperationen wie Sortieren, bestimmte Verknüpfungsalgorithmen oder Filter je Operation. Wird mehr Speicher benötigt, wird dieser von der Festplatte genommen. Ein zu kleiner Wert führt zu einer erhöhten Nutzung temporärer Dateien und belastet das Dateisystem. Diese Option legt auch die Größe des zur Verfügung stehenden Speichers für Sortieroperationen wie ORDER BY, DISTINCT, MERGE-JOIN, Hash-Tabellen wie sie für einen Hash-Join, Hash-Aggregierungen oder Hash-basierte IN-Operationen verwendet werden und Bitmap Index Scans zur Verfügung. Ein zu hoher Wert kann durch viele Operationen je Abfrage und viele gleichzeitige Benutzer schnell den Arbeitsspeicher aufbrauchen. Die Tabelle 4.3 enthält empfohlene Werte.
Mit trace_sort kann ermittelt werden, wie viel Speicher tatsächlich genutzt wird und die Einstellung entsprechend benötigten Speichers optimiert werden. Tabelle 4.4 enthält dafür ein Beispiel.

```
SET work_mem TO '8MB';
SET trace_sort TO on;
SET client_min_message TO LOG;
ANFRAGE
LOG: begin tuple sort: nkeys = 2, workMem = 8192, randomAccess = f
LOG: performsort starting: CPU 0.00s/0.00u sec elapsed 0.00 sec
LOG: performsort done: CPU 0.00s/0.00u sec elapsed 0.00 sec
LOG: internal sort ended, 31 kB used: CPU 0.00s/0.00u sec elapsed 0.00 sec
```

Tabelle 4.4: Beispiel für trace_sort

Standard:	16 MB
Empfohlen:	>16 MB

Tabelle 4.5: Empfehlung für den Parameter maintenance_work_mem

maintenance_work_mem

Der Parameter *maintenance_work_mem* legt die Obergrenze je Datenbanksitzung an Arbeitsspeicher für Verwaltungsoperationen zur Änderung und Erzeugung von Datenbankobjekten oder Garbage Collection fest. Dies wird von CREATE INDEX, ALTER TABLE, VACUUM und CLUSTER verwendet. Die Tabelle 4.5 enthält den Standardwert sowie die Empfehlung mehr Speicher zu nutzen.

Vacuum und Autovacuum

Die *Vacuum*-Befehle untersuchen Tabellen auf gelöschte oder aktualisierte Zeilenversionen und geben toten Speicher frei. Dabei kann mit *autovacuum* das automatische Ausführung des *Vacuum*-Befehls aktiviert werden, was sehr empfehlenswert ist.

4.4 Apache Spark

Apache Spark ist ein Open-Source Framework für Big-Data-Analysen. Auf der Projektseite (spa) wird es als blitzschnelle, einheitliche Analyse-Engine[6] bezeichnet. Seinen Ursprung hatte Apache Spark als Forschungsprojekt am AMPLab der University of California in Berkeley. Nach seiner Veröffentlichung 2010 unter einer Open-Source-Lizenz wurde es 2013 von der Apache Software Foundation weitergeführt und 2014 dort zu einem Top Level Projekt.

4.4.1 Aufbau

Apache Spark besitzt eine sprachübergreifende API, die mit Java, Scala, Python, R und SQL genutzt werden kann. Der Spark Core kann über Konnektoren auf verschiedene Datenquellen zugreifen. Dies ist in Abbildung 4.2 dargestellt.
Die zentrale Datenstruktur innerhalb von Apache Spark ist das Resilient Distributed Dataset (RDD). Dabei handelt es sich um eine Teilmenge von Daten, die über mehrere Rechner verteilt sein kann.

[6] eigene Übersetzung: "Lightning-fast unified analytics engine"

4.4.2 Module

Auf dem Spark Core bauen verschiedene, spezialisierte Module auf:

Modul	Beschreibung
Spark Streaming	Verarbeitung von Datenströmen durch Micro Batching
MLlib Machine Learning Bibliothek	Funktionsbibliothek mit Machine-Learning-Algorithmen
GraphX	Framework für Graph-Algorithmen
Spark SQL	Spark Modul für die Verarbeitung strukturierter Daten
Sonstige Pakete	Weitere oder eigene Module, die genutzt werden können

Tabelle 4.6: Apache Spark Module

4.4.3 Transformationen und Aktionen

Transformations-Funktionen sind zentrale Funktionen innerhalb von Apache Spark, um die RDDs, Datasets oder DataFrames zu verarbeiten. Das Resultat von Transformationen sind neue RDDs. Aktionen sind Funktionen, die keine RDDs als Ergebnis liefern.

4.4.4 Spark SQL

Spark SQL ist ein Spark Modul für die Verarbeitung strukturierter Daten, welches mit SQL und der Dataset-API verwendet werden kann. Die Informationen über Struktur der Daten und die durchzuführenden Berechnungen ermöglichen weitere Optimierungen. Dabei werden RDDs zu Data Frames als temporäre Tabelle mit benutzerdefinierten Tabellennamen für SQL-Abfragen. SQL kann direkt über die Kommandozeile oder über JDBC/ODBC verwendet werden. Als Resultat erhält man Datasets oder DataFrames.
Ein Dataset ist eine verteilte Sammlung von Daten, welches die Vorteile von RDDs (starke Typisierung, Lambda-Funktionen) mit der optimierten Ausführungs-Engine von Spark SQL verbindet. Ein Dataset kann aus JVM-Objekten erstellt und anschließend mit Transformationen verändert werden.
Ein DataFrame ist ein Dataset mit benannten Spalten. Es ist konzeptionell äquivalent zu einer Tabelle in einer relationalen Datenbank oder einem Data Frame in R oder Python, jedoch mit einigen Optimierungen.

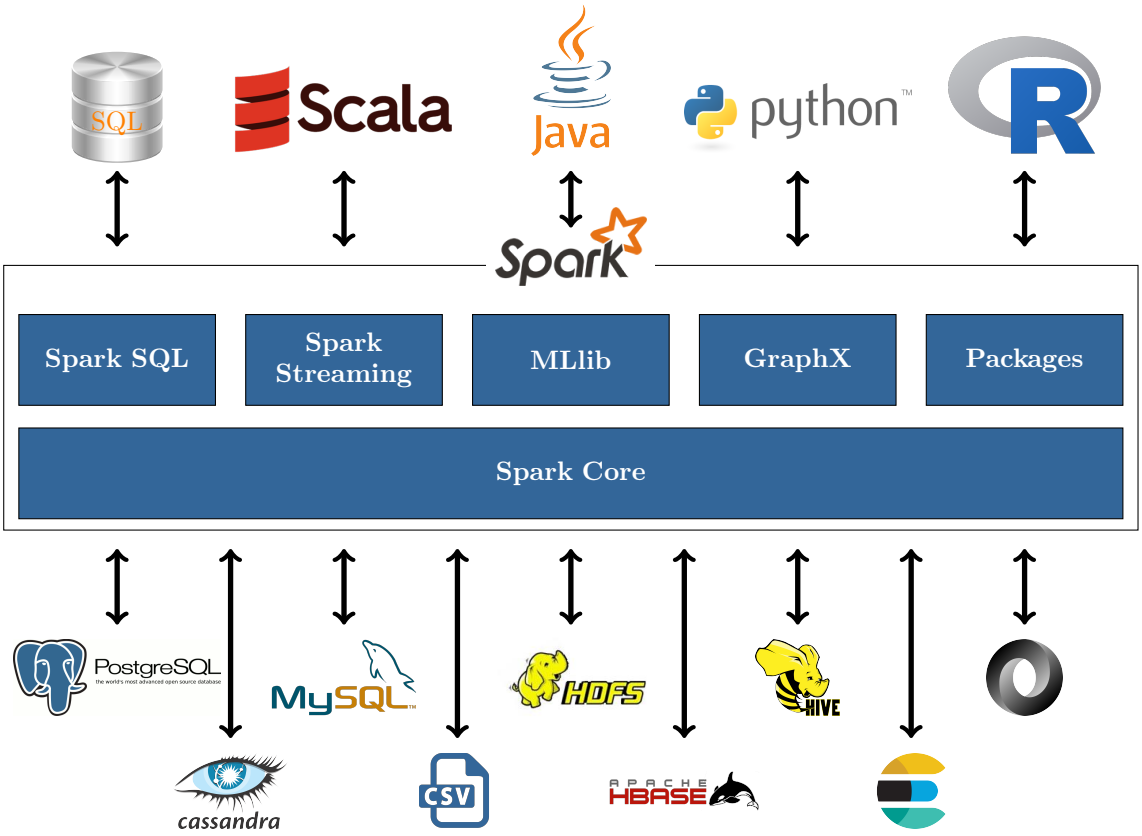


Abbildung 4.2: Schematischer Aufbau von Apache Spark. Die Rechte an den verwendeten Logos liegen bei den entsprechenden Organisationen oder Unternehmen.

Literaturverzeichnis

[avh] *Actian Vector in Hadoop: Scale the fastest SQL analytics engine on Apache Hadoop for industrial-strength deployment.* `https://www.actian.com/wp-content/uploads/2017/03/DS14-0316-US-ActianVectorinHadoop.pdf`, Abruf: 2018-11-14

[Bar15] Bartels, Sören: *Numerik 3x9: Drei Themengebiete in jeweils neun kurzen Kapiteln (Springer-Lehrbuch) (German Edition).* Springer Spektrum, 2015. – ISBN 978–3–662–48203–2

[Bon] Boncz, Peter A.: *Homepage of Peter Boncz: Projects.* `https://homepages.cwi.nl/~boncz/`, Abruf: 2018-11-14

[BZN05] Boncz, Peter A. ; Zukowski, Marcin ; Nes, Niels: MonetDB/X100: Hyper-Pipelining Query Execution. In: *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, www.cidrdb.org, 2005, 225–237

[Cod70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. In: *Commun. ACM* 13 (1970), Juni, Nr. 6, 377–387. `http://dx.doi.org/10.1145/362384.362685`. – DOI 10.1145/362384.362685. – ISSN 0001–0782

[Cod82] Codd, E. F.: Relational Database: A Practical Foundation for Productivity. In: *Commun. ACM* 25 (1982), Februar, Nr. 2, 109–117. `http://dx.doi.org/10.1145/358396.358400`. – DOI 10.1145/358396.358400. – ISSN 0001–0782

[DFS+18] Dietrich, Daniel ; Fenske, Ole ; Schomacker, Stefan ; Schweers, Philipp ; Heuer, Andreas: Stonebraker versus Google: 2-0 Scores in Rostock - A Comparison of Big Data Analytics Environments (Stonebraker gegen Google: Das 2: 0 fällt in Rostock). In: Klassen, Gerhard (Hrsg.) ; Conrad, Stefan (Hrsg.): *Proceedings of the 30th GI-Workshop Grundlagen von Datenbanken, Wuppertal, Germany, May 22-25, 2018.* Bd. 2126, CEUR-WS.org, 2018 (CEUR Workshop Proceedings), 101–107

[EH13] Eisentraut, Peter ; Helmle, Bernd: *PostgreSQL-Administration.* 3. O'Reilly Media, 2013 `https://books.google.de/books?id=SGzWDgAAQBAJ`. – ISBN 978–3–95561–159–0

[Fis17] Fischer, Gerd: *Lernbuch Lineare Algebra und Analytische Geometrie: Das Wichtigste ausführlich für das Lehramts- und Bachelorstudium (German Edition).* Springer Spektrum, 2017. – ISBN 978–3658181901

[HNZB07] Héman, Sándor ; Nes, Niels ; Zukowski, Marcin ; Boncz, Peter: Vectorized Data Processing on the Cell Broadband Engine. In: *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2007, S. 1–6

[IEC00] Norm IEC 60027–2 2000. *Letter symbols to be used in electrical technology – Part 2: Telecommunications and electronics*

[IZB11] Inkster, Doug ; Zukowski, Marcin ; Boncz, Peter: Integration of Vectorwise with Ingres. In: *SIGMOD Rec.* 40 (2011), November, Nr. 3, 45–53. `http://dx.doi.org/10.1145/2070736.2070747`. – DOI 10.1145/2070736.2070747. – ISSN 0163–5808

[Kud15] Kudrass: *TB Datenbanken, 2.A.* Carl Hanser Verlag GmbH & Co, 2015. – ISBN 978–3–446–43508–7

[Lam94] Lamersdorf, Winfried: *Datenbanken in verteilten Systemen - Konzepte, Lösungen, Standards.* Vieweg, 1994 (Datenbanksysteme). – ISBN 978–3–528–05467–0

[MBK00] Manegold, Stefan ; Boncz, Peter A. ; Kersten, Martin L.: Optimizing database architecture for the new bottleneck: memory access. In: *VLDB J.* 9 (2000), Nr. 3, S. 231–246. `http://dx.doi.org/10.1007/s007780000031`. – DOI 10.1007/s007780000031

[mon] *A short history about us.* `https://www.monetdb.org/AboutUs`, Abruf: 2018-11-14

[Nau] Naumann, Felix: *RDBMS Genealogy.* `https://hpi.de/naumann/projects/rdbms-genealogy`, Abruf: 2018-11-14

[pgx] *Postgrs-XL: Open Source Scalable SQL Database Cluster.* `https://www.postgres-xl.org/`, Abruf: 2018-11-15

[pos] *PostgreSQL: The world's most advanced open source database.* `https://www.postgresql.org/`, Abruf: 2018-11-14

[Prz05] Przedmiotow, Strony: *IEC 60027. Review of Content Standard. Letter symbols to be used in electrical technology.* `http://fiona.dmcs.pl/ak/IEC_60027-SIUnits.pdf`. Version: 09 2005

[RJ86] Rabiner, Lawrence R. ; Juang, Biing-Hwang: An introduction to hidden Markov models. In: *ieee assp magazine* 3 (1986), Nr. 1, S. 4–16

[RSS15] Rahm, Erhard ; Saake, Gunter ; Sattler, Kai-Uwe: *Verteiltes und Paralleles Datenmanagement: Von verteilten Datenbanken zu Big Data und Cloud (eXamen.press) (German Edition).* Springer Vieweg, 2015. – ISBN 978–3–642–45241–3

[SAD+10] Stonebraker, Michael ; Abadi, Daniel J. ; DeWitt, David J. ; Madden, Samuel ; Paulson, Erik ; Pavlo, Andrew ; Rasin, Alexander: MapReduce and parallel DBMSs: friends or foes? In: *Commun. ACM* 53 (2010), Nr. 1, S. 64–71. `http://dx.doi.org/10.1145/1629175.1629197`. – DOI 10.1145/1629175.1629197

[SBZ12] Switakowski, Michal ; Boncz, Peter A. ; Zukowski, Marcin: From Cooperative Scans to Predictive Buffer Management. In: *PVLDB* 5 (2012), Nr. 12, 1759–1770. `http://dx.doi.org/10.14778/2367502.2367515`. – DOI 10.14778/2367502.2367515

[spa] *Apache Spark™– Unified Analytics Engine for Big Data.* `https://spark.apache.org/`, Abruf: 2018-11-15

[SRL+90] Stonebraker, Michael ; Rowe, Lawrence A. ; Lindsay, Bruce G. ; Gray, Jim ; Carey, Michael J. ; Brodie, Michael L. ; Bernstein, Philip A. ; Beech, David: Third-generation database system manifesto. In: *ACM sIGMOD record* 19 (1990), Nr. 3, S. 31–44

[SSH11] Saake, Gunter ; Sattler, Kai-Uwe ; Heuer, Andreas: *Datenbanken - Implementierungstechniken (3. Aufl.)*. MITP, 2011 `http://www.it-fachportal.de/shop/buch/Datenbanken/detail.html,b98855`. – ISBN 978–3–8266–9156–0

[SSH18] Saake, Gunter ; Sattler, Kai-Uwe ; Heuer, Andreas: *Datenbanken - Konzepte und Sprachen*. 6. mitp, 2018. – ISBN 978–3–95845–776–8

[Stu16] Studer, Thomas: *Relationale Datenbanken: Von den theoretischen Grundlagen zu Anwendungen mit PostgreSQL (eXamen.press) (German Edition)*. Springer Vieweg, 2016. – ISBN 978–3–662–46570–7

[TM01] Taylor, Barry N. ; Mohr, Peter J.: The role of fundamental constants in the International System of Units (SI): present and future. In: *IEEE Trans. Instrumentation and Measurement* 50 (2001), Nr. 2, S. 563–567. `http://dx.doi.org/10.1109/19.918192`. – DOI 10.1109/19.918192

[tpc] *TPC-H – Top Ten Performance Results.* `http://www.tpc.org/tpch/results/tpch_perf_results.asp`, Abruf: 2018-11-14

[wd] *WD Black PC DH Series Specification Sheet.* `https://www.wd.com/content/dam/wdc/website/downloadable_assets/deu/spec_data_sheet/2879-771434.pdf`, Abruf: 2018-11-16

[ZNB08] Zukowski, Marcin ; Nes, Niels ; Boncz, Peter A.: DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In: Luo, Qiong (Hrsg.) ; Ross, Kenneth A. (Hrsg.): *4th Workshop on Data Management on New Hardware, DaMoN 2008, Vancouver, BC, Canada, June 13, 2008*, ACM, 2008, S. 47–54

Anhang A

Beispiele

A.1 Einheiten

Name	Symbol	Wert
Kibi	Ki	$2^{10} = 1024^1 = 1\,024$
Mebi	Mi	$2^{20} = 1024^2 = 1\,048\,576$
Gibi	Gi	$2^{30} = 1024^3 = 1\,073\,741\,824$
Tebi	Ti	$2^{40} = 1024^4 = 1\,099\,511\,627\,776$
Pebi	Pi	$2^{50} = 1024^5 = 1\,125\,899\,906\,842\,624$
Exbi	Ei	$2^{60} = 1024^6 = 1\,152\,921\,504\,606\,846\,976$
Zebi	Zi	$2^{70} = 1024^7 = 1\,180\,591\,620\,717\,411\,303\,424$
Yobi	Yi	$2^{80} = 1024^8 = 1\,208\,925\,819\,614\,629\,174\,706\,176$

Name	Symbol	Wert
Kilo	k	$10^3 = 1000^1 = 1\,000$
Mega	M	$10^6 = 1000^2 = 1\,000\,000$
Giga	G	$10^9 = 1000^3 = 1\,000\,000\,000$
Tera	T	$10^{12} = 1000^4 = 1\,000\,000\,000\,000$
Peta	P	$10^{15} = 1000^5 = 1\,000\,000\,000\,000\,000$
Exa	E	$10^{18} = 1000^6 = 1\,000\,000\,000\,000\,000\,000$
Zetta	Z	$10^{21} = 1000^7 = 1\,000\,000\,000\,000\,000\,000\,000$
Yotta	Y	$10^{24} = 1000^8 = 1\,000\,000\,000\,000\,000\,000\,000\,000$

Tabelle A.1: Präfixübersicht

Beispiel: $3\,\mathrm{TiB} = 3 \cdot 2^{40}\,\mathrm{Byte} \approx 3{,}3\,\mathrm{GB}$

A.2 Lösung Viterbi Beispiel

```
INSERT INTO a VALUES (1,1,0.5),(1,2,0.1),(1,3,0.1),(1,4,0.3),
    (2,1,0.1),(2,2,0.5),(2,3,0.1),(2,4,0.3),
    (3,1,0.3),(3,2,0.1),(3,3,0.5),(3,4,0.1),
    (4,1,0.1),(4,2,0.3),(4,3,0.1),(4,4,0.5);
INSERT INTO b VALUES
(1,1,0.3),(1,2,0.2),(1,3,0),(1,4,0.2),(1,5,0),(1,6,0.3),
    (2,1,0.1),(2,2,0.5),(2,3,0),(2,4,0.1),(2,5,0.2),(2,6,0.1),
    (3,1,0.3),(3,2,0.1),(3,3,0.3),(3,4,0.2),(3,5,0),(3,6,0.1),
    (4,1,0.3),(4,2,0.1),(4,3,0.1),(4,4,0.1),(4,5,0.3),(4,6,0.1);
INSERT INTO pi VALUES (1,0.4),(2,0.1),(3,0.2),(4,0.3);
INSERT INTO o VALUES (1,4),(2,3),(3,2),(4,4),(5,5);
SELECT viterbi2();
 viterbi2
----------
 (1,1) -- Berlin
 (2,4) -- Rostock
 (3,1) -- Berlin
 (3,2) -- Hamburg
 (4,1) -- Berlin
 (5,4) -- Rostock
```

Anhang B

Viterbi-Algorithmus

B.1 PostgreSQL

```
DROP TABLE IF EXISTS a, b, o, pi;
CREATE TABLE a(i int, j int, v double precision);
CREATE TABLE b(i int, j int, v double precision);
CREATE TABLE o(i int, v int);
CREATE TABLE pi(i int, v double precision);

\copy a (i, j, v) from a.txt delimiter ' ';
\copy b (i, j, v) from b.txt delimiter ' ';
\copy o (i, v) from o.txt delimiter ' ';
\copy pi (i, v) from pi.txt delimiter ' ';

CREATE INDEX ON a USING btree ( i );
CREATE INDEX ON b USING btree ( j, i );

CREATE OR REPLACE FUNCTION viterbi()
RETURNS TABLE (i int, v int)
AS $$
DECLARE
    rec RECORD;
BEGIN
--  INITIALISIERUNG
--  δ̂_ij
    DROP TABLE IF EXISTS delta;
    CREATE TEMP TABLE delta (i int, j int, v double precision);
    INSERT INTO delta SELECT b.i, 1, log(b.v) + log(p.v) as v
    FROM b, o, pi as p
```

```
    WHERE o.i = 1
    AND b.j = o.v
    AND p.i = b.i
    AND b.v <> 0
    AND p.v <> 0;
    CREATE INDEX ON delta USING btree (j);
--  REKURSION
--  ψ_{t-1}(i)
    DROP TABLE IF EXISTS psi;
    CREATE TEMP TABLE psi (i int, j int, v int);
--  Zwischenergebnis ad_i = max_{1≤j≤n} a_ij δ_{t-1}(j)
    DROP TABLE IF EXISTS ad;
    CREATE TEMP TABLE ad (i int, v double precision);
    FOR rec IN SELECT o.i, o.v
        FROM o
        WHERE o.i>1
        ORDER BY o.i
    LOOP
        TRUNCATE ad;
--      ad_i = max_{1≤j≤n} a_ij δ_{t-1}(j)
        INSERT INTO ad
            SELECT a.j, MAX(log(a.v)+d.v) as v
            FROM a, delta d
            WHERE a.i=d.i AND d.j=rec.i-1
            AND a.v <> 0
            GROUP BY a.j;
--      δ_t(i) = max_{1≤j≤n} a_ij δ_{t-1}(j) · b_i(O_t) = ad · b_i(O_t)
        INSERT INTO delta
            SELECT b.i, rec.i as j, ad.v+log(b.v) as v
            FROM ad, b
            WHERE ad.i=b.i AND b.j=rec.v
            AND b.v <> 0;
--      Ψ_{t-1}(i) = argmax_{1≤i≤n} δ_{t-1}(i) a_ij
        INSERT INTO psi
            SELECT a.j, rec.i-1 as i, a.i as v
            FROM a, delta d, ad
            WHERE a.i=d.i AND d.j=rec.i-1 AND log(a.v)+d.v=ad.v
            AND a.v <> 0;
    END LOOP;
```

```
--  TERMINATION + Zustandssequenz
--  l
    DROP TABLE IF EXISTS l;
    CREATE TEMP TABLE l (i int);
    INSERT INTO l SELECT COUNT (*) FROM o;
--  Ψ_l
    DROP TABLE IF EXISTS Psi_l;
    CREATE TEMP TABLE Psi_l (i int);
    INSERT INTO Psi_l SELECT d.i
        FROM delta d
        WHERE d.j IN (SELECT * FROM l) AND d.v IN (
--  Δ
            SELECT MAX(d.v)
            FROM delta d
            WHERE d.j IN (SELECT * FROM l)
        );
--  Ausgabe von X_1...X_l
    RETURN QUERY
--  X_1...X_{l-1}
    SELECT psi.j, psi.v
    FROM psi
    WHERE psi.i IN (SELECT * FROM Psi_l)
--  X_l
    UNION (SELECT l.i, Psi_l.i FROM l, Psi_l);
END
$$
LANGUAGE plpgsql;
EXPLAIN ANALYSE SELECT viterbi();
```

B.2 Postgres-XL

```
CREATE TEMP TABLE IF NOT EXISTS a(i int, j int, v double precision)
DISTRIBUTE BY HASH (i);
CREATE TEMP TABLE IF NOT EXISTS b(i int, j int, v double precision)
DISTRIBUTE BY HASH (j);
CREATE TEMP TABLE IF NOT EXISTS pi(i int, v double precision)
DISTRIBUTE BY REPLICATION;
CREATE TEMP TABLE IF NOT EXISTS o(i int, v int) DISTRIBUTE BY
REPLICATION;

CREATE TEMP TABLE IF NOT EXISTS delta (i int, j int, v double
precision) DISTRIBUTE BY HASH (i);
CREATE INDEX ON delta USING btree (j);
CREATE TEMP TABLE IF NOT EXISTS psi (i int, j int, v int) DISTRIBUTE BY
HASH (i);
CREATE TEMP TABLE IF NOT EXISTS ad (i int, v double precision)
DISTRIBUTE BY HASH (i);
CREATE TEMP TABLE IF NOT EXISTS l (i int);
CREATE TEMP TABLE IF NOT EXISTS Psi_l (i int);

\copy a (i, j, v) from a.txt delimiter ' ';
\copy b (i, j, v) from b.txt delimiter ' ';
\copy o (i, v) from o.txt delimiter ' ';
\copy pi (i, v) from pi.txt delimiter ' ';

create index on a using btree ( i );
create index on b using btree ( j, i );

CREATE OR REPLACE FUNCTION viterbi()
RETURNS TABLE (i int, v int)
AS
DECLARE
    rec RECORD;
BEGIN
--  INITIALISIERUNG
RAISE NOTICE 'INIT';
    TRUNCATE TABLE delta;
    INSERT INTO delta SELECT b.i, 1, log(b.v) + log(p.v) as v
    FROM b, o, pi as p
    WHERE o.i = 1
```

```
    AND b.j = o.v
    AND p.i = b.i
    AND b.v <> 0
    AND p.v <> 0;
--  REKURSION
RAISE NOTICE 'REKURSION';
    TRUNCATE TABLE psi;
    TRUNCATE TABLE ad;
    FOR rec IN SELECT o.i, o.v
        FROM o
        WHERE o.i>1
        ORDER BY o.i
    LOOP
        TRUNCATE ad;
        INSERT INTO ad
            SELECT a.j, MAX(log(a.v)+d.v) as v
            FROM a, delta d
            WHERE a.i=d.i AND d.j=rec.i-1
            AND a.v <> 0
            GROUP BY a.j;
        INSERT INTO delta
            SELECT b.i, rec.i as j, ad.v+log(b.v) as v
            FROM ad, b
            WHERE ad.i=b.i AND b.j=rec.v
            AND b.v <> 0;
        INSERT INTO psi
            SELECT a.j, rec.i-1 as i, a.i as v
            FROM a, delta d, ad
            WHERE a.i=d.i AND d.j=rec.i-1 AND log(a.v)+d.v=ad.v
            AND a.v <> 0;
    END LOOP;
RAISE NOTICE 'TERMINATION';
--  TERMINATION + Zustandssequenz
    TRUNCATE TABLE l;
    INSERT INTO l SELECT COUNT (*) FROM o;
    TRUNCATE TABLE Psi_l;
    INSERT INTO Psi_l SELECT d.i
        FROM delta d
        WHERE d.j IN (SELECT * FROM l) AND d.v IN (
            SELECT MAX(d.v)
            FROM delta d
```

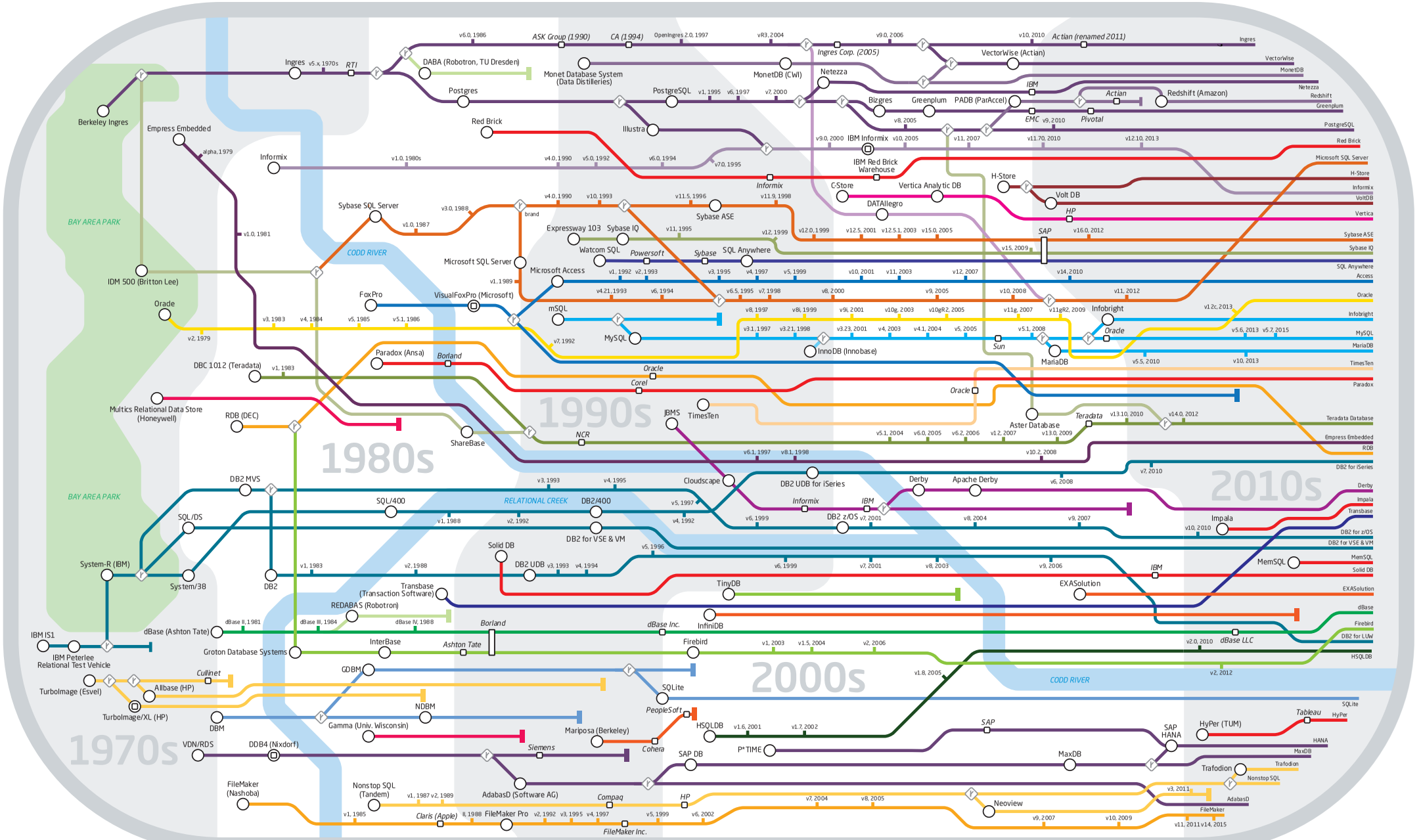
```
            WHERE d.j IN (SELECT * FROM l)
        );
    RETURN QUERY
    SELECT psi.j, psi.v
    FROM psi
    WHERE psi.i IN (SELECT * FROM Psi_l)
    UNION (SELECT l.i, Psi_l.i FROM l, Psi_l);
END

LANGUAGE plpgsql;
EXPLAIN ANALYSE SELECT viterbi();
```

Anhang C

Weitere Materialien

C.1 RDBMS Genealogie

Genealogy of Relational Database Management Systems



Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Key to lines and symbols

DBMS name (Company) | *Acquisition* | v9.2006 Versions | Discontinued | Branch (intellectual and/or code) | Crossing lines have no special semantics

Felix Naumann, Jana Bauckmann, Claudia Exeler, Jan-Peer Rudolph, Fabian Tschirschnitz
Contact - Hasso Plattner Institut, University of Potsdam, felix.naumann@hpi.de
Design - Alexander Sandt Grafik-Design, Hamburg

Version 6.0 - October 2018
https://hpi.de/naumann/projects/rdbms-genealogy.html