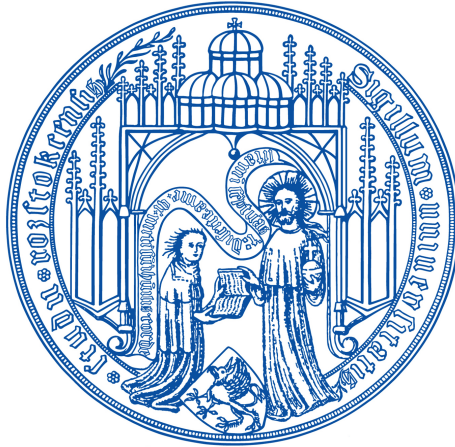




Universität Rostock

Fakultät für Informatik und Elektrotechnik

Institut für Informatik

Implementierung, Optimierung und Evaluation von HPLM in SQL

Bachelorarbeit

vorgelegt von: Amer Badralden

Matrikelnummer: 221200767

geboren am: 26.11.1995

Erstgutachter: Herr Dr.-Ing. Hannes Grunert

Zweitgutachter: Herr Dr.-Ing. Holger Meyer

Rostock, den 19.08.2026

Inhaltsverzeichnis

1. Einleitung	7
1.1. Zielsetzung der Arbeit	7
1.2. Aufbau der Arbeit	8
2. Stand der Technik und theoretische Grundlagen	9
2.1. Piecewise Linear Models (PLM)	9
2.2. Hierarchical Piecewise Linear Models (HPLM)	10
2.2.1. Funktionsweise des HPLM an einem durchgerechneten Beispiel . .	11
2.2.2. Übertragung in relationale Datenbanken und Kopplungstechniken	15
2.2.3. Statistische Gütemaße und Signifikanz	16
2.3. Verwandte Arbeiten zur Kopplung statistischer Methoden mit SQL . . .	17
3. Konzept der In-Database Einzelfallanalyse	19
3.1. Systemarchitektur und Lösungsansatz	19
3.2. Datenvorbereitung und Dummy-Coding	20
3.2.1. Eingabe: Rohdaten und SQL-Aufruf	20
3.2.2. Ausgabe: Dummy-Kodierung und ihre Semantik	22
3.2.3. Bezug zur Hierarchie und Literatur	23
3.3. Mehrebenenaggregation und statistische Kennzahlen	24
3.3.1. Abbildung der Mehrebenenstruktur durch GROUPING SETS . .	25
3.3.2. Berechnung statistischer Gütemaße und algebraische Formalisierung	26
3.3.3. Approximation des p-Werts bei kleinen Stichproben	28
4. Implementierung des HPLM in PostgreSQL	30
4.1. Datenbankschema	30
4.2. Hilfsfunktion zur Approximation des p-Werts	32
4.3. Aggregat-basierte HPLM-Berechnung	33
4.3.1. Zustandsstruktur und Transitions-Funktion	33
4.3.2. Berechnung der HPLM-Kennwerte	35
4.4. Hierarchische Auswertung mit GROUPING SETS	36
5. Evaluation	38
5.1. Validierung gegen die Referenzimplementierung	38
5.1.1. Versuchsaufbau	38
5.1.2. Ergebnisse der SQL-Implementierung	39

5.1.3.	Vergleich mit <code>scan</code> auf der Individualebene	39
5.1.4.	Vergleich auf der Aggregatebene	40
5.1.5.	Hinweis zu Standardfehlern und p -Werten	41
5.1.6.	Interpretation der Ergebnisse	41
5.1.7.	Zusätzliche Validierung an einem unabhängigen Datensatz	41
5.2.	Skalierungs-Benchmark auf einem realistischen Datensatz	44
5.3.	Empirische Gegenüberstellung: GROUPING SETS vs. Einzelabfragen	46
5.4.	Alternative Kopplungsansätze	53
5.4.1.	Realisierung als SQL-Artefakte: Trade-offs zwischen Laufzeit, Speicher und Wartbarkeit	53
5.4.2.	Externe prozedurale Spracherweiterungen: PL/R und PL/Python	55
6.	Zusammenfassung und Ausblick	60
6.1.	Zusammenfassung der Ergebnisse	60
6.2.	Methodische Limitationen	62
6.2.1.	Pooling-Charakter der relationalen Aggregation	62
6.3.	Ausblick	65
A.	R-Skript zur Validierung	67
B.	Einzelmessungen des GROUPING-SETS-Vergleichs	69
C.	Einzeldurchläufe der Benchmarks	71
	Literaturverzeichnis	75
	Selbstständigkeitserklärung	77

Abbildungsverzeichnis

2.1. Visualisierung des durchgerechneten Beispiels	15
3.1. Systemarchitektur der In-Database-Berechnung	20
5.1. Grafische Darstellung der Anfragepläne aus pgAdmin	48

Tabellenverzeichnis

2.1. Dummy-Kodierung der acht Messwerte der ersten Person	12
2.2. Übergang vom PLM zum HPLM	14
3.1. Eingabe: Auszug der Rohdaten aus <code>messwerte</code>	22
3.2. Ausgabe: Dummy-Kodierung der Phaseninformation	23
5.1. HPLM-Kennwerte der PostgreSQL-Implementierung	39
5.2. Trend-Vergleich: <code>scan::plm()</code> und SQL	39
5.3. Vergleich der gepoolten Schätzungen	40
5.4. Vergleich mit <code>scan</code> für <code>exampleAB</code>	42
5.5. In SQL abgeleitete Modellvarianten	43
5.6. Auszug des SMARD-Benchmark-Datensatzes	44
5.7. Ausführungszeit der HPLM-Aggregatabfrage	45
5.8. GROUPING SETS gegenüber Einzelabfragen	49
5.9. Auszug des erweiterten Benchmark-Datensatzes	51
5.10. Laufzeit bei wachsender Ebenenzahl	51
5.11. Trade-offs der vier SQL-Artefakte	54
6.1. Methodische Gegenüberstellung mit <code>scan::hplm()</code>	63
6.2. Verhalten bei wenigen Messpunkten in Phase A	64
B.1. Einzelmessungen des GROUPING-SETS-Vergleichs	69
B.2. Einzeldurchläufe des <code>work_mem</code> -Experiments	70
C.1. Einzeldurchläufe des Ebenen-Benchmarks	71
C.2. Einzeldurchläufe der Artefakt-Messungen	72
C.3. Einzeldurchläufe des Skalierungs-Benchmarks	73
C.4. Laufzeitvergleich SQL-Aggregat und PL/Python	74

Listings

3.1. Allgemeine Struktur des HPLM-Aufrufs	21
3.2. Konzeptioneller SQL-Aufruf	21
3.3. Abstraktes Template für die hierarchische HPLM-Auswertung	25
4.1. Datenbankschema mit drei normalisierten Tabellen	31
4.2. Approximation des zweiseitigen p -Werts	32
4.3. Transitions-Funktion <code>hplm.sfunc</code>	33
4.4. Definition des benutzerdefinierten Aggregats	34
4.5. Berechnung der HPLM-Kennwerte	35
4.6. Hierarchische HPLM-Auswertung mit <code>GROUPING SETS</code>	37
5.1. Anfrageplan auf der größten Skalierungsstufe	46
5.2. Einzelne <code>GROUPING-SETS</code> -Abfrage über drei Ebenen	46
5.3. Drei separate Aggregatabfragen	47
5.4. Anfrageplan der <code>GROUPING-SETS</code> -Abfrage	48
5.5. Anfrageplan der Einzelabfrage	48
5.6. Anfrageplan der <code>GROUPING-SETS</code> -Abfrage über fünf Ebenen	52
5.7. Die vier untersuchten SQL-Artefakte	54
5.8. Prototypische OLS-Regression der Phase A über PL/Python.	56
5.9. Lineare Modellanpassung über PL/R	56
A.1. R-Skript zur HPLM-Validierung	67

1. Einleitung

Die systematische Auswertung von Leistungs- und Verhaltensdaten spielt in zahlreichen wissenschaftlichen und praktischen Disziplinen eine zentrale Rolle; für die Einzelfallforschung liegen hierzu inzwischen etablierte methodische Standards vor [1]. Um den Erfolg gezielter Maßnahmen und Interventionen messbar zu machen, kommen zunehmend Methoden der Einzelfallanalyse (Single-Case Data Analysis) zum Einsatz [2, 3]. Ein etabliertes statistisches Verfahren in diesem Bereich ist das Hierarchical Piecewise Linear Model (HPLM) [4]. Es ermöglicht die präzise Untersuchung von Interventionseffekten, indem es den natürlichen Verlauf (Phase A) von Leistungsveränderungen nach einem Training oder einer Behandlung (Phase B) isoliert.

Bisher werden solche komplexen Regressionen vorwiegend in spezialisierter Statistiksoftware wie R durchgeführt. In der gängigen Praxis liegen die zugrundeliegenden Rohdaten dabei häufig nur semistrukturiert vor – beispielsweise manuell gepflegt in Tabellenkalkulationsprogrammen wie Excel. Um tiefergehende Analysen durchzuführen, müssen diese Daten oft mühsam aufbereitet und in externe Statistik-Tools überführt werden. Dieser Prozess der externen Datenverarbeitung ist insbesondere bei wachsenden Datenmengen ineffizient und fehleranfällig – ein Befund, der die Entwicklung datenbankinterner Analyseverfahren maßgeblich motiviert hat [5].

Insbesondere bei der Erfassung von hochfrequenten Leistungsdaten, wie beispielsweise bei der kontinuierlichen Messung von Reaktionszeiten im Millisekundenbereich über längere Trainingsphasen, entstehen rasant wachsende Datensätze. Um diese Größenordnungen künftig ohne Systembrüche und performant auswerten zu können, bedarf es skalierbarer Lösungen direkt auf Datenbankebene.

1.1. Zielsetzung der Arbeit

Das Ziel dieser Bachelorarbeit ist es, die mathematische Logik des HPLM direkt in ein relationales Datenbanksystem zu überführen. Anhand der Datenbank PostgreSQL soll demonstriert werden, wie das statistische Modell mittels SQL und der prozeduralen Erweiterung PL/pgSQL implementiert werden kann. Dadurch wird ein direkter, datenbankinterner Analyseprozess (In-Database Analytics) geschaffen. Die Anwendung muss

in der Lage sein, wesentliche Parameter des HPLM – wie den Startwert (Intercept), den natürlichen Verlauf (Trend), den unmittelbaren Interventionssprung (Level) sowie die veränderte Steigung nach dem Training (Slope) – zu berechnen. Idealerweise sollen darüber hinaus auch statistische Signifikanzmaße wie p -Werte und weitere Gütekriterien ausgegeben werden, ohne dass externe Softwarelösungen in Anspruch genommen werden müssen.

Um dieses übergeordnete Ziel zu erreichen, werden in der vorliegenden Arbeit die folgenden zentralen Forschungsfragen formuliert und untersucht:

1. **Umsetzbarkeit.** Inwiefern lässt sich die komplexe mathematische Logik eines HPLM, einschließlich statistischer Signifikanztests (p -Werte), vollständig in nativen SQL- und PL/pgSQL-Operationen abbilden?
2. **Abstraktion und Skalierbarkeit.** Wie muss die datenbankinterne Architektur (beispielsweise durch den Einsatz von benutzerdefinierten Aggregatfunktionen und `GROUPING SETS`) gestaltet sein, um datensatzunabhängig und performant über verschiedene Hierarchieebenen hinweg zu aggregieren?
3. **Validität.** Inwieweit stimmen die berechneten Ergebnisse des entwickelten In-Database-Ansatzes mit den etablierten Referenzwerten externer Standard-Statistiksoftware (wie R) überein?

1.2. Aufbau der Arbeit

Die vorliegende Arbeit ist in sechs Kapitel gegliedert. Nach dieser Einleitung werden in Kapitel 2 die theoretischen Grundlagen des HPLM sowie der verwendeten Datenbanksysteme erläutert. Kapitel 3 widmet sich dem Konzept der Übertragung des statistischen Modells in eine relationale Struktur durch Dummy-Coding. In Kapitel 4 wird die konkrete SQL-Implementierung in PostgreSQL vorgestellt. Kapitel 5 validiert diese Implementierung gegen die Referenzimplementierung `scan`, untersucht ihr Skalierungsverhalten und vergleicht alternative Kopplungsansätze. Die Arbeit schließt in Kapitel 6 mit einer Zusammenfassung der Ergebnisse und einem Ausblick ab.

2. Stand der Technik und theoretische Grundlagen

In diesem Kapitel werden die theoretischen Konzepte und statistischen Methoden vorgestellt, die das Fundament dieser Arbeit bilden. Um Leistungsdaten im zeitlichen Verlauf und unter dem Einfluss von Interventionen – wie beispielsweise gezielten Trainingsmaßnahmen – korrekt analysieren zu können, reicht eine einfache lineare Regression oftmals nicht aus. Daher wird im Folgenden zunächst das Konzept der Piecewise Linear Models (PLM) erläutert, bevor dieses zum Hierarchical Piecewise Linear Model (HPLM) erweitert wird.

2.1. Piecewise Linear Models (PLM)

Die klassische lineare Regression geht von einem durchgehenden, konstanten Trend über den gesamten Betrachtungszeitraum aus. In der Praxis, insbesondere bei der Untersuchung von Interventionen (wie dem Start eines neuen Trainingsprogramms), kommt es jedoch häufig zu Brüchen in der Entwicklung. Ein Modell, das eine durchgehende Regressionsgerade durch alle Datenpunkte legt, würde diese wichtigen Veränderungen verschleiern. An dieser Stelle setzen *Piecewise Linear Models* (stückweise lineare Modelle), auch bekannt als Segmented Regression, an, deren statistische Grundlagen maßgeblich von Quandt eingeführt wurden [6].

Ein PLM unterteilt den gesamten Zeitverlauf an definierten Zeitpunkten – den sogenannten Knoten (Knots) oder Interventionszeitpunkten – in mehrere separate Phasen oder Stücke (Pieces). Für jedes dieser Stücke wird eine eigene lineare Regression berechnet, wobei die Übergänge zwischen den Stücken das Kerninteresse der Analyse darstellen. Im Kontext der Einzelfallanalyse (Single-Case Data Analysis) mit einer einzigen Intervention wird der Datensatz in der Regel in zwei Phasen unterteilt [3]:

- **Phase A (Baseline):** Der natürliche Leistungsverlauf der Testperson vor der Intervention.

- **Phase B (Intervention):** Der Leistungsverlauf der Testperson nach dem Start der Intervention.

Je nach Studiendesign kann eine dritte Phase hinzukommen: eine nachgelagerte Evaluationsphase (häufig als Phase E oder Follow-up bezeichnet), die nach Abschluss der Intervention und einem zeitlichen Abstand erhebt, ob der beobachtete Effekt anhält. Formal ändert sich dadurch nichts am Prinzip – es kommt lediglich ein weiterer Knoten und damit ein weiteres Paar aus Level- und Slope-Term hinzu. Die vorliegende Arbeit betrachtet den in der Einzelfallforschung verbreiteten Zwei-Phasen-Fall (A–B); die Erweiterung der Implementierung auf zusätzliche Phasen ist konzeptionell geradlinig, da die Transitions-Funktion lediglich um weitere Summenbereiche je Phase zu ergänzen wäre.

Das PLM formuliert dies mathematisch als multiple lineare Regression, indem es sogenannte Dummy-Variablen nutzt, um die Phasen logisch voneinander zu trennen [4]:

$$Y_t = \beta_0 + \beta_1 \cdot Trend_t + \beta_2 \cdot Level_t + \beta_3 \cdot Slope_t + e_t \quad (2.1)$$

Die Parameter interpretieren sich dabei wie folgt:

- **β_0 (Intercept):** Der Startwert (Y-Achsenabschnitt) zu Beginn der Messung in Phase A.
- **β_1 (Trend):** Die Steigung der Leistungskurve in Phase A (natürliche Entwicklung).
- **β_2 (Level-Effekt):** Der abrupte Niveau-Unterschied (Sprung) genau zum Zeitpunkt des Wechsels von Phase A zu Phase B.
- **β_3 (Slope-Effekt):** Die Differenz der Steigungen zwischen Phase B und Phase A. Ein veränderter Wert würde hier bedeuten, dass die Messwerte nach der Intervention schneller oder langsamer sinken bzw. steigen als in der natürlichen Entwicklung zuvor.
- **e_t (Fehlerterm):** Das Residuum zum Zeitpunkt t . Es beschreibt die zufällige Abweichung der tatsächlichen Beobachtung vom durch das Modell vorhergesagten Wert.

2.2. Hierarchical Piecewise Linear Models (HPLM)

Während ein klassisches PLM hervorragend geeignet ist, um die Daten einer einzelnen Person zu analysieren, stößt es an seine Grenzen, wenn ganze Gruppen – wie etwa eine

Kohorte von mehreren Probanden – gleichzeitig ausgewertet werden sollen. Hier kommt das *Hierarchical Piecewise Linear Model* (HPLM) zum Einsatz. Ein HPLM verbindet die stückweise lineare Regression (PLM) mit den Konzepten der Mehrebenenanalyse (Multilevel Modeling). Datenstrukturen in Interventionsstudien sind klassischerweise hierarchisch verschachtelt (nested data): Die einzelnen Subjekte oder Probanden (Level 1) sind innerhalb einer übergeordneten Gesamtgruppe (Level 2) gruppiert. Das HPLM berücksichtigt diese Hierarchie und löst zwei wesentliche Probleme:

1. **Individuelle Varianzen:** Nicht jedes Subjekt startet mit dem gleichen Ausgangswert (β_0) und nicht jedes Subjekt reagiert gleich stark auf die Intervention (β_2, β_3). Das HPLM erlaubt es, diese Werte als sogenannte „Random Effects“ zu betrachten, die für jedes Individuum variieren dürfen.
2. **Gepoolte Effekte (Fixed Effects):** Gleichzeitig kann das Modell übergreifende, allgemeingültige Effekte für die gesamte Gruppe berechnen, um festzustellen, ob die Intervention *im Durchschnitt* über alle Subjekte hinweg gewirkt hat.

2.2.1. Funktionsweise des HPLM an einem durchgerechneten Beispiel

Um die im vorigen Abschnitt formal eingeführten Effekte greifbar zu machen, wird die Berechnung im Folgenden an einem kleinen, vollständig nachrechenbaren Beispiel demonstriert. Dabei wird zunächst die Auswertung einer einzelnen Person gezeigt (die Ebene des PLM, Level 1) und anschließend die Zusammenführung mehrerer Personen zu einem Gruppeneffekt (die hierarchische Ebene des HPLM, Level 2).

Schritt 1: Kodierung der Modellprädiktoren

Betrachtet werden für die erste Person acht Messzeitpunkte; die Intervention setzt am Tag 5 ein (`start_tag = 5`), sodass die Tage 1 bis 4 zur Phase A (Baseline) und die Tage 5 bis 8 zur Phase B (Intervention) gehören. Zunächst werden die Rohdaten in die drei Prädiktoren *Trend*, *Level* und *Slope* übersetzt (vgl. die Definition in Abschnitt 2.1). Der Prädiktor *Trend* entspricht `tag - 1`, *Level* ist ein binärer Indikator für Phase B, und *Slope* zählt die Tage seit Interventionsbeginn. Streng genommen ist allein *Level* eine echte Dummy-Variable und *Slope* ein daraus gebildeter Interaktionsterm; *Trend* ist hingegen keine kodierte Größe, sondern unmittelbar der (um eins verschobene) Messzeitpunkt. Als Effekt β_1 bleibt der *Trend* dennoch Bestandteil des Modells – er beschreibt die Steigung der Baseline –, wird hier aber nicht eigens kodiert, sondern direkt aus der Zeitachse übernommen.

Tag	Y (Messwert)	Phase	Trend	Level	Slope
1	10	A	0	0	0
2	12	A	1	0	0
3	14	A	2	0	0
4	16	A	3	0	0
5	30	B	4	1	0
6	31	B	5	1	1
7	32	B	6	1	2
8	33	B	7	1	3

Tabelle 2.1.: Dummy-Kodierung der acht Messwerte der ersten Person. Am Phasenübergang (Tag 5) springt Level von 0 auf 1; Slope beginnt erst danach zu zählen.

Die Spalte *Level* übernimmt in der folgenden Rechnung die Rolle des Phasentrenners: Zeilen mit *Level* = 0 bilden die Datengrundlage der Phase-A-Regression, Zeilen mit *Level* = 1 die der Phase-B-Regression. Zugleich sind *Level* und *Slope* die Prädiktoren, deren Koeffizienten β_2 und β_3 aus Gleichung (2.1) in Schritt 3 bestimmt werden.

Schritt 2: Getrennte Regression beider Phasen

Für jede Phase wird eine eigene lineare Regression des Messwerts Y auf den Trend ($x = \text{tag} - 1$) berechnet. Steigung und Achsenabschnitt einer solchen Regressionsgeraden ergeben sich aus den zentrierten Streuungssummen

$$S_{xx} = \sum_i (x_i - \bar{x})^2, \quad S_{xy} = \sum_i (x_i - \bar{x})(y_i - \bar{y}) \quad (2.2)$$

über $\beta_1 = S_{xy}/S_{xx}$ und $\beta_0 = \bar{y} - \beta_1 \bar{x}$; die formale Herleitung dieser Größen folgt in Abschnitt 3.3. Für die Baseline (Phase A) mit den vier Punkten $x = 0, 1, 2, 3$ und $y = 10, 12, 14, 16$ gilt $\bar{x}_A = 1,5$ und $\bar{y}_A = 13$. Daraus ergeben sich

$$S_{xx} = (0-1,5)^2 + (1-1,5)^2 + (2-1,5)^2 + (3-1,5)^2 = 5, \quad (2.3)$$

$$S_{xy} = (0-1,5)(10-13) + (1-1,5)(12-13) + (2-1,5)(14-13) + (3-1,5)(16-13) = 10 \quad (2.4)$$

und damit:

$$\beta_{1,A} = \frac{S_{xy}}{S_{xx}} = \frac{10}{5} = 2,0, \quad \beta_{0,A} = \bar{y}_A - \beta_{1,A} \cdot \bar{x}_A = 13 - 2,0 \cdot 1,5 = 10,0. \quad (2.5)$$

Die Baseline beginnt also bei einem Startwert von 10 und steigt pro Tag um 2 Einheiten. Analog ergeben sich für die vier Punkte der Phase B ($x = 4, 5, 6, 7$ und $y = 30, 31, 32, 33$)

mit den Mittelwerten $\bar{x}_B = 5,5$ und $\bar{y}_B = 31,5$ die Streuungssummen $S_{xx} = 5$ und $S_{xy} = 5$, und damit:

$$\beta_{1,B} = \frac{S_{xy}}{S_{xx}} = \frac{5}{5} = 1,0, \quad \beta_{0,B} = \bar{y}_B - \beta_{1,B} \cdot \bar{x}_B = 31,5 - 1,0 \cdot 5,5 = 26,0. \quad (2.6)$$

Schritt 3: Ableitung der HPLM-Effekte für die erste Person

Aus den Koeffizienten beider Phasen werden die vier Kennwerte abgeleitet. Intercept und Trend stammen direkt aus Phase A:

$$Intercept = \beta_{0,A} = 10,0, \quad Trend = \beta_{1,A} = 2,0. \quad (2.7)$$

Der **Level-Effekt** beschreibt den abrupten Niveausprung am Interventionszeitpunkt. Er ergibt sich als Differenz der beiden Vorhersagewerte am Tag 5 (also bei $x = 4$). Die Baseline würde am Tag 5 den Wert $10 + 2,0 \cdot 4 = 18$ vorhersagen; die Phase-B-Gerade sagt an diesem Punkt $26 + 1,0 \cdot 4 = 30$ voraus. Damit gilt:

$$Level = 30 - 18 = 12,0. \quad (2.8)$$

Der Messwert springt am Interventionszeitpunkt also um 12 Einheiten nach oben, die über den natürlichen Trend hinausgehen. Der **Slope-Effekt** beschreibt schließlich die Veränderung der Steigung von Phase A zu Phase B:

$$Slope = \beta_{1,B} - \beta_{1,A} = 1,0 - 2,0 = -1,0. \quad (2.9)$$

Die Person verbessert sich nach der Intervention also weiterhin, aber um 1 Einheit pro Tag langsamer als zuvor.

Damit schließt sich der Kreis zur Kodierung aus Schritt 1: Die abgeleiteten Effekte sind genau die Koeffizienten der Spalten *Level* und *Slope*, also $\beta_2 = 12,0$ und $\beta_3 = -1,0$ in Gleichung (2.1). Setzt man die vier geschätzten Koeffizienten zusammen mit den kodierten Prädiktorwerten in die Modellgleichung ein, werden in diesem konstruierten Beispiel alle acht Messwerte exakt reproduziert – etwa für Tag 6 ($Trend = 5$, $Level = 1$, $Slope = 1$):

$$\hat{Y}_6 = 10 + 2,0 \cdot 5 + 12,0 \cdot 1 + (-1,0) \cdot 1 = 31. \quad (2.10)$$

Die kodierten Werte aus Tabelle 2.1 gehen somit unmittelbar in die Modellgleichung ein; die getrennten Phasen-Regressionen sind lediglich der Rechenweg, über den ihre Koeffizienten bestimmt werden.

Schritt 4: Vom PLM zum HPLM – Aggregation über mehrere Personen

Bis hierher wurde ausschließlich eine einzelne Person betrachtet; dies entspricht einem reinen PLM auf Ebene 1. Das *Hierarchical* Piecewise Linear Model entsteht erst, wenn mehrere Personen zu übergeordneten Effekten zusammengefasst werden (Ebene 2). Zur Veranschaulichung dient eine zweite Person mit identischem Interventionszeitpunkt, deren Einzelauswertung nach demselben Verfahren erfolgt. Tabelle 2.2 stellt die individuellen Kennwerte beider Personen sowie den daraus abgeleiteten Gruppeneffekt gegenüber.

Ebene	Intercept	Trend	Level	Slope
Person 1 (Level 1)	10,0	2,0	12,0	-1,0
Person 2 (Level 1)	20,0	1,0	16,0	1,0
Gruppe (Level 2)	15,0	1,5	14,0	0,0

Tabelle 2.2.: Übergang vom PLM (Einzelperson, Level 1) zum HPLM (Gruppeneffekt, Level 2). Der Gruppeneffekt entsteht durch Pooling der Messwerte beider Personen.

Der Gruppeneffekt auf Ebene 2 wird in der vorliegenden Arbeit dadurch gebildet, dass die Messwerte beider Personen zusammengefasst und gemeinsam ausgewertet werden – ein Vorgehen, das in der Mehrebenenstatistik als *Complete Pooling* bezeichnet wird. Im vorliegenden balancierten Fall (beide Personen haben gleich viele Messpunkte) entspricht das Ergebnis genau dem Mittelwert der Einzelschätzungen: Der Gruppen-Trend von 1,5 liegt mittig zwischen den individuellen Werten 2,0 und 1,0, und der gegenläufige Slope beider Personen (-1,0 und +1,0) mittelt sich zu 0,0.

Damit ist das Grundprinzip des HPLM vollständig illustriert: Auf Ebene 1 erhält jede Person ihre eigene, individuell kodierte Phasentrennung und Regression; auf Ebene 2 werden diese Einzelergebnisse zu Kohorteneffekten verdichtet. Es sei angemerkt, dass etablierte Statistikpakete wie `scan` auf Ebene 2 ein verfeinertes Verfahren (*Partial Pooling* mittels REML) einsetzen, das individuelle Schätzungen gegeneinander abwägt; der hier gezeigte und in dieser Arbeit umgesetzte Complete-Pooling-Ansatz ist die in einem relationalen Aggregat unmittelbar abbildbare Variante. Der genaue Unterschied beider Strategien wird in Abschnitt 6.2 eingeordnet. Wie die Aggregation über mehrere Ebenen hinweg in SQL umgesetzt wird, behandelt Kapitel 4.

Abbildung 2.1 veranschaulicht das Beispiel grafisch: Teilabbildung (a) zeigt die vier Effekte für Person 1, Teilabbildung (b) den Übergang zur Gruppenebene durch Complete Pooling.

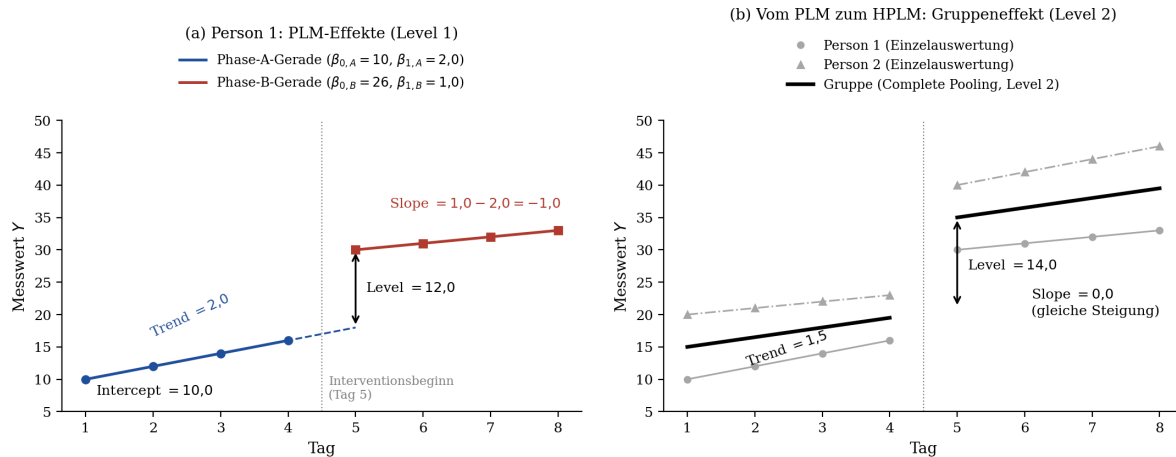


Abbildung 2.1.: Visualisierung des durchgerechneten Beispiels. (a) PLM der ersten Person mit den vier Effekten Intercept, Trend, Level und Slope; die gestrichelte Linie setzt die Baseline-Gerade kontrafaktisch fort. (b) Aggregation beider Personen zum Gruppeneffekt (Complete Pooling): Der Gruppen-Trend von 1,5 und der Level-Sprung von 14,0 entsprechen den Werten aus Tabelle 2.2; die gegenläufigen Slope-Effekte mitteln sich zu 0 (parallele Geraden).

2.2.2. Übertragung in relationale Datenbanken und Kopplungstechniken

Traditionell werden hierarchische Modelle in spezialisierter Statistiksoftware (wie R oder SPSS) berechnet, indem Iterationsverfahren wie *Restricted Maximum Likelihood* (REML) angewendet werden [7]. Um die dafür notwendigen Rohdaten aus relationalen Datenbanksystemen zu extrahieren, kommen klassische Kopplungstechniken der Datenbankprogrammierung zum Einsatz [8]. Üblicherweise geschieht dies über Call-Level Interfaces (CLI) wie ODBC oder JDBC. Dabei werden SQL-Anweisungen aus der Programmiersprache der Anwendung (*Host Language*) an die Datenbank gesendet und die vollständigen Ergebnismengen über das Netzwerk zurückgeliefert.

Dies erfordert jedoch den Export massiver Datenmengen aus den operativen Systemen. Dieser netzwerkbasierter Datentransfer führt unweigerlich zu Performance-Engpässen durch hohen Kommunikationsaufwand im Client-Server-Kontext und dem sogenannten *Impedance Mismatch*: Die Datenbank denkt in Mengen (eine Abfrage liefert eine ganze Ergebnisrelation), die Programmiersprache hingegen in einzelnen Sätzen und Objekten, die nacheinander verarbeitet werden. An der Schnittstelle müssen die Ergebnisse daher Zeile für Zeile in die Datenstrukturen der Anwendungssprache umgewandelt werden – ein Konvertierungsaufwand, der bei großen Ergebnismengen erheblich ins Gewicht fällt. Ein

moderner informationstechnischer Ansatz zur Lösung dieses Problems ist das Paradigma der *In-Database Analytics* [5].

Dabei wird die statistische Berechnung direkt an den Ort der Datenhaltung verlagert. Technisch entspricht dies der Kopplungsmethode der „Datenbanksprache + Erweiterung“. Hierbei werden prozedurale Erweiterungen der Datenbanksprache (wie PL/pgSQL in PostgreSQL) nativ im Datenbankkern ausgeführt, was eine räumliche Nähe von Daten und Anwendungslogik schafft. Die hierarchische Struktur des HPLM lässt sich hier theoretisch durch relationale Gruppierungsoperationen abbilden. Durch den Einsatz von massenverarbeitenden SQL-Konstrukten (wie der `GROUP BY`-Klausel) kann die Level-2-Hierarchie (die individuellen Subjekte) in der Datenbank partitioniert werden. Für jede dieser Partitionen berechnet die Datenbank die PLM-Koeffizienten (Level 1) eigenständig. Dieser Ansatz vermeidet Systembrüche durch externe Datentransfers, umgeht den *Impedance Mismatch* durch ein einheitliches Typsystem und bildet die theoretische Grundlage für performante, skalierbare Auswertungen auf großen Datensätzen.

2.2.3. Statistische Gütemaße und Signifikanz

Um die Qualität und Verlässlichkeit der berechneten Regressionsmodelle zu bewerten, werden in der Statistik etablierte Gütemaße und Signifikanztests herangezogen. Da das Ziel dieser Arbeit die vollständige datenbankinterne Abbildung der HPLM-Analyse ist, müssen auch diese Metriken theoretisch erfasst werden, um sie später algorithmisch umsetzen zu können:

- **Bestimmtheitsmaß (R^2):** Diese Kennzahl gibt an, welcher Anteil der Varianz der abhängigen Variablen durch das Regressionsmodell erklärt wird. Ein Wert nahe 1 signalisiert eine sehr hohe Anpassungsgüte (Model Fit) [9].
- **p-Wert (Signifikanz):** Der p-Wert dient der Überprüfung der statistischen Signifikanz der berechneten Effekte – beispielsweise ob der Interventionssprung (β_2) verlässlich auf die Intervention zurückzuführen ist. Er gibt an, wie wahrscheinlich es ist, unter Gültigkeit der Nullhypothese ein mindestens so extremes Ergebnis zu beobachten wie das vorliegende. In der gängigen Praxis wird ein Schwellenwert von $p < 0,05$ herangezogen, um ein Ergebnis als statistisch signifikant zu betrachten [9].

2.3. Verwandte Arbeiten zur Kopplung statistischer Methoden mit SQL

Um den aktuellen Stand der Forschung im Bereich der datenbankinternen Statistik und der Einzelfallanalyse systematisch zu erfassen, wurde im Vorfeld dieser Arbeit eine strukturierte Literaturrecherche durchgeführt. Als primäre wissenschaftliche Literaturdatenbank diente Google Scholar. Die Suche erfolgte iterativ mit zunehmend spezifischen Suchausdrücken. Eine breite initiale Suche ohne Phrasenbindung nach den Einzelbegriffen *in-database analytics* lieferte zunächst eine sehr umfangreiche Treffermenge. Die Bindung an die exakte Phrase „*in-database analytics*“ grenzte diese bereits drastisch ein. Durch Verknüpfung mehrerer Phrasen mit dem booleschen Operator **AND**, etwa „*in-database analytics*“ **AND** „*user-defined aggregates*“, sowie die domänenspezifische Suche „*single-case data analysis*“ **AND** „*piecewise linear*“ wurde die Literatur systematisch auf die zentralen Referenzwerke dieser Arbeit fokussiert.

Die resultierenden Publikationen wurden in einem zweistufigen Filterprozess (Screening von Titel und Abstract, gefolgt von einer systematischen Volltextanalyse) selektiert. Arbeiten, die sich ausschließlich mit NoSQL-Systemen, maschinellem Lernen ohne Regressionsfokus oder vollständig datenbankunabhängigen Statistiksoftware-Suiten befassten, wurden aus der engeren Auswahl ausgeschlossen. Der Fokus der finalen Selektion lag somit auf nativen SQL-Erweiterungen sowie auf den für das HPLM verwendeten Referenzimplementierungen, die den methodischen Maßstab für diese Arbeit bilden.

Die Idee, statistische Auswertungsmethoden direkt mit relationalen Datenbanksystemen zu koppeln, wurde in der identifizierten Literatur bereits in unterschiedlichen Architekturen untersucht. Eine zentrale Vorarbeit ist die MADlib-Bibliothek von Hellerstein et al. [10], die eine umfangreiche Sammlung statistischer Verfahren – darunter lineare und logistische Regression, k-Means-Clustering und verschiedene Klassifikationsalgorithmen – als Open-Source-Erweiterung für PostgreSQL und Greenplum bereitstellt. MADlib realisiert die Kopplung über benutzerdefinierte Aggregatfunktionen (User-Defined Aggregates, UDA), die in einem dreistufigen Muster aus *Transition Function*, *Merge Function* und *Final Function* arbeiten. Auf diese Weise können große Datensätze zeilenweise gestreamt und im Datenbankkern verarbeitet werden, ohne dass sie vollständig in den Arbeitsspeicher geladen werden müssen.

Einen ergänzenden Ansatz verfolgt das Bismarck-Framework von Feng et al. [11], das eine vereinheitlichte Architektur für In-RDBMS-Analytik auf Basis des inkrementellen Gradientenabstiegs (Incremental Gradient Descent, IGD) bereitstellt. Die Autoren zeigen, dass eine Vielzahl unterschiedlicher statistischer Modelle – darunter logistische Regression, Support Vector Machines und Matrix-Faktorisierungen – über einen einzigen, generischen IGD-Kern realisiert werden kann, der ebenfalls als UDA in der Datenbank implementiert ist. Damit wird der Implementierungsaufwand für neue Verfahren erheblich reduziert.

Beiden Ansätzen ist ein grundlegendes Muster gemein: Die Kopplung erfolgt nativ im Datenbankkern und umgeht damit die in Abschnitt 2.2.2 diskutierten Probleme externer Kopplungsschnittstellen [5].

Während MADlib und Bismarck somit eine breite Palette *allgemeiner* statistischer Verfahren in SQL bereitstellen, fehlt in diesen Systemen eine spezialisierte Unterstützung für die hierarchische, stückweise lineare Regression (HPLM). Verfahren wie die einfache lineare oder logistische Regression aus MADlib lassen sich nicht direkt auf die HPLM-Logik anwenden, da diese eine phasenabhängige Dummy-Kodierung sowie die getrennte Schätzung von Intercept, Trend, Level und Slope erfordert.

Für die spezifische Domäne der Einzelfallanalyse (Single-Case Data Analysis) existieren HPLM-Implementierungen daher bislang ausschließlich außerhalb von Datenbanksystemen. Als Referenzimplementierung dient in dieser Arbeit das R-Paket `scan` (*Single-Case Data Analyses for Single and Multiple Baseline Designs*) [12], ein frei verfügbares, quell-offenes Paket zur Visualisierung und statistischen Auswertung von Interventionsstudien. Die Wahl gerade dieses Pakets als Vergleichsmaßstab ist dadurch begründet, dass es die hier untersuchte Funktion `hplm()` direkt bereitstellt, über die offizielle Paketdistribution CRAN beziehbar ist und aktiv weiterentwickelt wird. Die Kernfunktion `hplm()` modelliert die hierarchische Struktur der Daten, indem sie intern auf das R-Paket `nlme` (*Nonlinear Mixed-Effects Models*) zurückgreift. Die methodische Fundierung dieses Pakets lässt sich an einem objektiven Kriterium festmachen: `nlme` zählt zu den sogenannten *recommended packages*, die fester Bestandteil jeder offiziellen R-Distribution sind und einer gesonderten Pflege durch das R-Core-Team unterliegen. Über dieses Paket wird das iterative *Restricted Maximum Likelihood*-Verfahren (REML) angewendet, um die Varianzkomponenten und Zufallseffekte (*Random Effects*) der individuellen Probanden zu schätzen.

Diese Auswertung setzt jedoch voraus, dass die Rohdaten zunächst aus dem operativen Datenbanksystem exportiert und in die R-Umgebung geladen werden – es manifestiert sich also genau das eingangs beschriebene Grundproblem der externen Datenverarbeitung. Eine native SQL-Umsetzung des HPLM, die ohne diesen externen Datenexport auskommt, ist nach aktuellem Kenntnisstand bislang nicht in der Literatur dokumentiert. Genau diese Lücke adressiert die vorliegende Arbeit.

Bevor die konkrete Implementierung vorgestellt werden kann, muss geklärt werden, wie sich die eingeführte Modelllogik in relationale Strukturen übersetzen lässt. Genau dies entwickelt Kapitel 3: von der Gesamtarchitektur über das Dummy-Coding und die Abbildung der Mehrebenenstruktur bis zur datenbankinternen Berechnung der statistischen Gütemaße.

3. Konzept der In-Database Einzelfallanalyse

Die Überführung des HPLM in eine relationale Datenbankumgebung erfordert zwei Schritte: die Transformation der zeitbasierten Rohdaten in ein für SQL-Aggregate verarbeitbares Format und die Abbildung der Mehrebenenstruktur auf relationale Gruppierungsoperationen. Bevor beide im Detail entwickelt werden, wird im Folgenden zunächst die Gesamtarchitektur des Systems vorgestellt.

3.1. Systemarchitektur und Lösungsansatz

Um den Prozess der *In-Database Analytics* zu realisieren, wurde eine Architektur entworfen, die den gesamten Analyseworkflow – von der Datenspeicherung bis zur statistischen Auswertung – innerhalb der PostgreSQL-Datenbank kapselt (siehe Abbildung 3.1). Die Basis bilden die rohen Leistungsdaten sowie zugehörige Metadatentabellen. Diese Metadaten umfassen neben den individuellen Interventionszeitpunkten der Probanden auch demografische und gruppenspezifische Merkmale (wie Alter, Geschlecht oder Spielzugehörigkeit). Ein zentrales SQL-Skript (bzw. eine PL/pgSQL-Funktion) greift auf diese Tabellen zu, führt das sogenannte Dummy-Coding (die zeitliche Transformation) durch und berechnet mittels nativer Aggregatfunktionen und `GROUPING SETS` die Regressionskoeffizienten für die verschiedenen Hierarchieebenen. Das finale Ergebnis wird als relationale Tabelle ausgegeben und kann anschließend von BI-Tools (Business Intelligence) oder Dashboards ohne weiteren Berechnungsaufwand visualisiert werden.

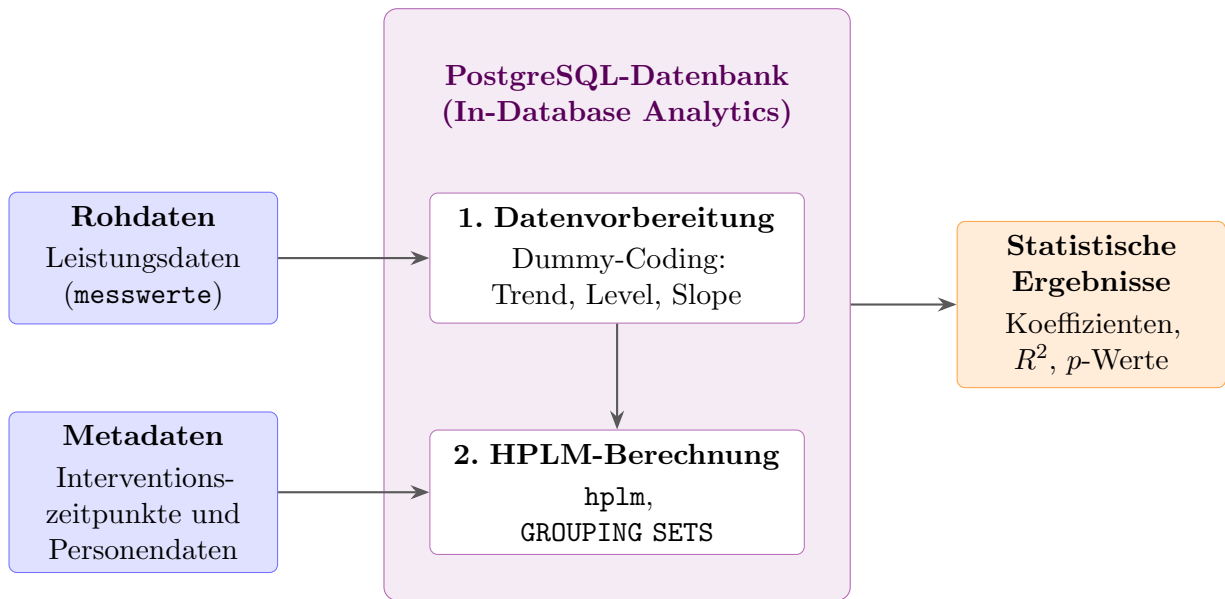


Abbildung 3.1.: Systemarchitektur: Datenfluss von den Rohdaten zur In-Database-Berechnung des HPLM.

3.2. Datenvorbereitung und Dummy-Coding

Da relationale Datenbanksysteme wie PostgreSQL keine spezialisierten Funktionen für stückweise lineare Regressionen besitzen, muss die kategoriale Phaseninformation (Phase A vs. Phase B) in numerische Prädiktoren umgewandelt werden. Diese Umwandlung wird in der Statistik als *Dummy-Coding* bezeichnet. Die in Abschnitt 2.1 mathematisch eingeführten Prädiktoren $Trend_t$, $Level_t$ und $Slope_t$ aus Gleichung (2.1) werden in der SQL-Umsetzung konkret wie folgt aus den gespeicherten Spalten `tag` und `start_tag` abgeleitet:

- $Trend = tag - 1$
- $Level = 1$, falls $tag \geq start_tag$; sonst 0
- $Slope = tag - start_tag$, falls $tag \geq start_tag$; sonst 0

3.2.1. Eingabe: Rohdaten und SQL-Aufruf

Um das Prinzip der relationalen Datenvorbereitung zu veranschaulichen, wird im Folgenden ein konkretes Beispiel herangezogen. Dieses basiert auf dem später in der Validierung

(vgl. Abschnitt 5.1) genutzten Datensatz zur Messung kognitiver Reaktionszeiten. Aus Normalisierungsgründen sind die Rohdaten auf zwei zentrale Tabellen aufgeteilt: Die Tabelle `messwerte` enthält die kontinuierlich erfassten Reaktionszeiten (`reaktionszeit_ms`) der Probanden an den jeweiligen Messtagen (`tag`). Die Metadatatabelle `interventionen` speichert hingegen pro Person den individuellen Tag, an dem die Trainingsphase (Phase B) begonnen hat (`start_tag`).

Da die statistische HPLM-Berechnung auf die zeilenweise Kombination von Messzeitpunkt und Interventionsstart angewiesen ist, erfolgt die Zusammenführung beider Tabellen nativ in der Datenbank. Um diese Transformation greifbar zu machen, zeigt das folgende Listing den konzeptionellen Kern des SQL-Aufrufs. Aus Gründen der Übersichtlichkeit beschränkt sich dieses einführende Beispiel auf die Auswertung pro Person; die vollständige, hierarchische Implementierung mittels `GROUPING SETS` über alle Aggregationsebenen folgt in Kapitel 4.

Der grundlegende Aufbau dieses Aufrufs ist unabhängig vom konkreten Datensatz und lässt sich als allgemeine Syntax-Schablone wie folgt darstellen:

```
1 SELECT
2     <gruppierungsattribute>,
3     (hplm(<messwert>, <zeitpunkt>, <start_tag>)).*
4 FROM <tabellen>
5 GROUP BY <gruppierungsattribute>;
```

Listing 3.1: Allgemeine Struktur des HPLM-Aufrufs.

Dabei stehen `<messwert>`, `<zeitpunkt>` und `<start_tag>` für die drei Argumente der Aggregatfunktion, `<tabellen>` für die Datenquelle einschließlich eventueller Verknüpfungen, während `GROUP BY <gruppierungsattribute>` die Ebene festlegt, auf der ausgewertet wird. Der Aufruf entspricht damit syntaktisch dem einer nativen Aggregatfunktion wie `sum()` oder `regr_slope()`; der Zusatz `.*` sorgt lediglich dafür, dass die von der Funktion zurückgegebenen Kennwerte in einzelne Ergebnisspalten aufgeteilt werden. Angewendet auf den konkreten Datensatz dieser Arbeit ergibt sich der in Listing 3.2 dargestellte Aufruf:

```
1 SELECT
2     m.person_id,
3     (hplm(m.reaktionszeit_ms, m.tag, i.start_tag)).*
4 FROM messwerte m
5 JOIN interventionen i ON m.person_id = i.person_id
6 GROUP BY m.person_id;
```

Listing 3.2: Konzeptioneller SQL-Aufruf zur Verknüpfung der Tabellen und Parameterübergabe.

Dieser Aufruf verdeutlicht den Datenfluss vor der eigentlichen Dummy-Kodierung: Über einen `JOIN` werden Messwerte und Interventionsmetadaten auf Basis der `person_id` verbunden. Für jede Person iteriert die Aggregatfunktion `hplm` anschließend über alle zugehörigen Zeilen. Sie erhält in jedem Schritt drei Argumente: den aktuellen Messwert (`reaktionszeit_ms`) und den aktuellen Tag (`tag`) aus der Tabelle `messwerte` sowie den für diese Person konstanten Interventionsstart (`start_tag`) aus der Tabelle `interventionen`. Auf Basis dieser Argumente entscheidet die Funktion zeilenweise über die Phasenzuordnung und akkumuliert die benötigten Summen – die konkrete Umsetzung dieser Logik wird in Kapitel 4 im Detail erläutert.

Die nachfolgende Tabelle 3.1 zeigt einen exemplarischen Ausschnitt der Rohdaten rund um den Phasenwechsel zweier Probanden; der vollständige Datensatz umfasst pro Person je zehn Messungen in Phase A und Phase B (vgl. Abschnitt 5.1).

<code>person_id</code>	<code>tag</code>	<code>reaktionszeit_ms</code>
1	10	251
1	11	220
1	12	215
⋮	⋮	⋮
2	10	270
2	11	240
2	12	235
⋮	⋮	⋮

Tabelle 3.1.: Eingabe: Auszug der Rohdaten aus `messwerte` für zwei Probanden. Die Auslassungszeichen deuten weitere Messzeitpunkte je Person an; die Phasenzuordnung ist nicht persistent gespeichert.

3.2.2. Ausgabe: Dummy-Kodierung und ihre Semantik

Die SQL-Implementierung leitet aus diesen Rohdaten zur Laufzeit die drei numerischen Prädiktoren ab. Tabelle 3.2 zeigt das resultierende Dummy-Coding für beide Probanden im Übergang von Phase A zu Phase B.

person_id	tag	Y	Phase	Trend	Level	Slope
1	10	251	A	9	0	0
1	11	220	B	10	1	0
1	12	215	B	11	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2	10	270	A	9	0	0
2	11	240	B	10	1	0
2	12	235	B	11	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Tabelle 3.2.: Ausgabe: Dummy-Kodierung der Phaseninformation in die drei numerischen Prädiktoren Trend, Level und Slope. Die Auslassungszeichen deuten weitere Messzeitpunkte je Person an.

Wie in der Tabelle ersichtlich, nehmen die drei Prädiktoren konkrete semantische Bedeutungen an:

- **Trend:** Zählt die kontinuierlichen Messtage ab Beginn der Aufzeichnung (`tag - 1`) und bildet damit die natürliche Leistungsentwicklung (Reifung) der Probanden ab.
- **Level:** Fungiert als binärer Indikator (0 für Phase A, 1 für Phase B), der exakt am individuellen Interventionszeitpunkt umschaltet – im vorliegenden Beispiel bei `start_tag = 11`. Dieser Wert ist nicht hartcodiert, sondern wird pro Person dynamisch aus der Tabelle `interventionen` gelesen und kann daher von Proband zu Proband variieren.
- **Slope:** Misst die verstrichenen Tage *seit* dem Start der Intervention, um eine mögliche Veränderung der Steigung (Lernkurve) in Phase B zu erfassen.

In der konkreten PostgreSQL-Implementierung findet sich exakt diese Zuweisungslogik in der Transitions-Funktion `hplm_sfunc` wieder (vgl. das Listing in Abschnitt 4.3). Dort werden *Trend*, *Level* und *Slope* nicht als persistente Spalten auf die Festplatte geschrieben, sondern zur Laufzeit über eine `IF t < start_tag`-Verzweigung abgeleitet, um die für die Regression benötigten Quadratsummen in getrennte Array-Indizes des Aggregatzustands zu leiten.

3.2.3. Bezug zur Hierarchie und Literatur

Die Notwendigkeit dieser spezifischen Dummy-Notation leitet sich direkt aus den in Kapitel 2 diskutierten Referenzwerken ab. Wie die einschlägige Fachliteratur zur Einzel-

fallanalyse [4] belegt, ist eine klassische, durchgehende lineare Regression unzureichend, um Interventionseffekte messbar zu machen. Die formale Trennung in *Trend*, *Level* und *Slope* ist erforderlich, um den natürlichen Leistungsverlauf (Phase A) isoliert vom Behandlungseffekt (Phase B) betrachten zu können.

Die Dummy-Kodierung selbst findet auf der untersten Datenebene (Level 1) statt, da sie pro Messzeitpunkt und individuell pro Person berechnet wird. Genau diese individualisierte Vorverarbeitung ist die Voraussetzung für das *H* (Hierarchical) im HPLM: Erst weil jede Person ihre eigene, passgenaue Phasentrennung erhält, kann die Datenbank im nächsten Schritt mittels **GROUPING SETS** valide übergeordnete Kohorteneffekte (Level 2) aggregieren. Die Hierarchie entsteht somit nicht durch die Kodierung selbst, sondern durch die anschließende Aggregation der individuell kodierten Daten über mehrere Ebenen hinweg.

3.3. Mehrebenenaggregation und statistische Kennzahlen

Anstatt die Rohdaten für eine hierarchische Auswertung aus dem Datenbanksystem zu exportieren und in externe Statistikumgebungen zu überführen, nutzt das hier vorgestellte Konzept konsequent die native Mengenorientierung von SQL (*In-Database Analytics*). Die methodische Mehrebenenhierarchie des HPLM (Level 1: Individuum, Level 2: Kohorten) wird dabei direkt über relationale Gruppierungsoperationen abgebildet. Die demografischen und versuchsspezifischen Merkmale der Probanden (wie Alter, Geschlecht oder das untersuchte Spiel) dienen dabei als Kriterien, nach denen aggregiert wird. Dieser Abschnitt erläutert zunächst, wie diese Mehrebenenstruktur über **GROUPING SETS** realisiert wird, und anschließend, wie die statistischen Gütemaße datenbankintern berechnet werden. Das konkrete relationale Datenbankschema mit seinen Tabellen wird in Abschnitt 4.1 vorgestellt.

Obwohl Teile der Struktur einer klassischen hierarchischen Abfolge entsprechen (Individuum → Kohorte), liegen mit Attributen wie **spiel** und **geschlecht** parallele, voneinander unabhängige Dimensionen vor. Für den rein hierarchischen Anteil ließe sich die Aggregation alternativ kompakt über den **ROLLUP**-Operator angeben, der genau die absteigende Folge der Ebenen erzeugt. **ROLLUP** ist hier jedoch auf einen einzelnen, linearen Hierarchiepfad beschränkt und könnte die parallel verlaufenden, voneinander unabhängigen Auswertungen (etwa separat nach Spiel und nach Geschlecht) nicht in einem Schritt abbilden. Der **CUBE**-Operator wiederum erzeugt automatisch *sämtliche* Kombinationen der Dimensionen; er ist ebenfalls einsetzbar, liefert für die vorliegende Auswertung aber auch Ebenen, die inhaltlich nicht benötigt werden.

Sowohl ROLLUP als auch CUBE sind letztlich Kurzschreibweisen, die intern auf GROUPING SETS zurückgeführt werden und daher gegeneinander austauschbar sind. In dieser Arbeit wird bewusst die explizite GROUPING SETS-Notation verwendet, da sie die Definition genau der benötigten – auch nicht-hierarchischen – Kombinationen erlaubt und die einzelnen Aggregationsebenen transparent im Quelltext dokumentiert.

3.3.1. Abbildung der Mehrebenenstruktur durch GROUPING SETS

Für die Auswertung der hierarchischen Struktur (Level 1: Individuum, Level 2: Kohorten) kommt die SQL-Klausel GROUPING SETS zum Einsatz. Anstatt für jede Analyseebene eine isolierte Datenbankabfrage abzusetzen, erlaubt dieser Operator die parallele Berechnung mehrerer Aggregationsebenen innerhalb eines einzigen Tabellendurchlaufs (engl. *Table Scan*). Das Konzept sieht vor, dass das Regressionsmodell simultan auf der feingranularsten Ebene (pro `person_id`) sowie auf aggregierten Meta-Ebenen (etwa pro `spiel`, pro `geschlecht` oder über die gesamte Kohorte) berechnet wird. Dieser Ansatz ist darauf ausgelegt, die Anzahl der notwendigen Tabellendurchläufe und damit den I/O-Aufwand gegenüber mehreren separaten Einzelabfragen zu reduzieren. Der grundlegende Aufbau dieser Abfrage lässt sich unabhängig vom konkreten Datensatz als Schablone darstellen:

```

1 SELECT
2     <dimension_individuum>, <dimension_gruppe>,
3     (hplm(<y_wert>, <zeitpunkt>, <interventionsstart>)).*
4 FROM <faktentabelle>
5 JOIN <dimensionstabelle> ON ...
6 GROUP BY GROUPING SETS (
7     (<dimension_individuum>, <dimension_gruppe>), -- Level 1: Individuum
8     (<dimension_gruppe>), -- Level 2: Kohorte
9     () -- Gesamtpopulation
10 );

```

Listing 3.3: Abstraktes Template für die hierarchische HPLM-Auswertung.

In dieser Schablone legt die GROUPING SETS-Klausel fest, welche Aggregationsebenen in einem einzigen Durchlauf berechnet werden: die Kombination aus Individuum und Gruppe (Level 1), die Gruppe allein (Level 2) sowie die leere Menge () für die Gesamtpopulation. Die konkrete Anwendung dieser Schablone auf den Datensatz dieser Arbeit erfolgt in Kapitel 4 (vgl. Abschnitt 4.4).

3.3.2. Berechnung statistischer Gütemaße und algebraische Formalisierung

Um der Forderung nach wissenschaftlicher Validität in der Einzelfallforschung nachzukommen, reicht die reine Schätzung der Regressionskoeffizienten nicht aus. Das Konzept sieht daher die vollständige datenbankinterne Berechnung statistischer Güte- und Signifikanzmaße vor.

Hierbei stellt sich auf konzeptioneller Ebene die Frage, warum die Regression nicht durch getrennte Aufrufe nativer SQL-Funktionen separat für Phase A und Phase B durchgeführt wird. PostgreSQL stellt hierfür neben den Punktschätzern `regr_slope` und `regr_intercept` auch die zentrierten Quadratsummen über `regr_sxx`, `regr_syy` und `regr_sxy` sowie das Bestimmtheitsmaß über `regr_r2` nativ bereit; eine phasenetrennte Auswertung ließe sich somit im Prinzip auch aus diesen Bausteinen zusammensetzen.

Die vorliegende Arbeit entwickelt dennoch eine benutzerdefinierte Aggregatfunktion (engl. *User-Defined Aggregate*, UDA), und zwar aus drei Gründen: Erstens benötigt das HPLM pro Phase sechs Grundgrößen, sodass mit nativen Funktionen mehr als zwölf separate Aggregataufrufe erforderlich wären, deren Ergebnisse anschließend in einer nachgelagerten Abfrageschicht zu den phasenübergreifenden Effekten (Level, Slope), Standardfehlern, Konfidenzintervallen und p -Werten kombiniert werden müssten.

Zweitens stellt PostgreSQL im Standardumfang keine Verteilungsfunktion zur Bestimmung des p -Werts bereit. Spezialisierte Erweiterungen wie MADLIB schließen diese Lücke und berechnen Regressionsstatistiken einschließlich der p -Werte datenbankintern – bezeichnenderweise ebenfalls auf Basis benutzerdefinierter Aggregate [10]. Da MADLIB jedoch als externe C++-Bibliothek in das System integriert werden muss, während der hier verfolgte Ansatz allein mit nativem PL/pgSQL auskommt, bleibt eine eigene, abhängigkeitsfreie Implementierung der p -Wert-Approximation (vgl. Abschnitt 3.3.3) notwendig, um Systembrüche zu vermeiden.

Drittens kapselt das UDA die gesamte Berechnungslogik – von der dynamischen Phasenzuordnung über die Akkumulation bis zur Ableitung aller Kennwerte – hinter einem einzigen Funktionsaufruf, was dem Aggregat-Muster der verwandten Arbeiten (vgl. Abschnitt 2.3) entspricht und den hierarchischen Aufruf mit `GROUPING SETS` kompakt hält.

Daher erfordert der Lösungsansatz die Entwicklung einer benutzerdefinierten Aggregatfunktion (User-Defined Aggregate, UDA). Diese sammelt während eines einzigen Tabellendurchlaufs strikt getrennt für beide Phasen ($p \in \{A, B\}$) die elementaren Summen: $\sum x_p$, $\sum y_p$, $\sum x_p^2$, $\sum y_p^2$, $\sum x_p y_p$ sowie den Stichprobenumfang n_p . Der Prädiktor x entspricht dabei dem in Abschnitt 3.2 eingeführten Trend (`tag - 1`).

Aus diesen isoliert akkumulierten Rohsummen werden in der nachgelagerten Abschlussfunktion zunächst die zentrierten Quadratsummen für jede Phase abgeleitet:

$$S_{xx,p} = \sum x_p^2 - \frac{(\sum x_p)^2}{n_p}, \quad S_{yy,p} = \sum y_p^2 - \frac{(\sum y_p)^2}{n_p}, \quad S_{xy,p} = \sum x_p y_p - \frac{\sum x_p \sum y_p}{n_p} \quad (3.1)$$

Daraus ergeben sich Steigung und Achsenabschnitt der jeweiligen Phasen-Regression in geschlossener Form:

$$\beta_{1,p} = \frac{S_{xy,p}}{S_{xx,p}}, \quad \beta_{0,p} = \bar{y}_p - \beta_{1,p} \bar{x}_p. \quad (3.2)$$

Erst durch das explizite Vorhalten dieser algebraischen Zwischengrößen lassen sich im anschließenden Berechnungsschritt die erforderlichen Metriken mathematisch korrekt ableiten:

- **Bestimmtheitsmaß (R^2):** Gibt an, welcher Anteil der Streuung der Messwerte durch die Regressionsgerade erklärt wird – ein Wert nahe 1 bedeutet, dass die Gerade die Punkte fast vollständig beschreibt. Es wird für beide Phasen separat ermittelt, um die Anpassungsgüte des Modells in Baseline und Intervention getrennt bewerten zu können:

$$R_p^2 = \frac{(S_{xy,p})^2}{S_{xx,p} \cdot S_{yy,p}}. \quad (3.3)$$

- **Standardfehler (SE) und Konfidenzintervalle:** Der Standardfehler beziffert, wie stark die geschätzte Steigung bei wiederholter Messung streuen würde – je kleiner er ist, desto verlässlicher ist die Schätzung. Über die Residuenquadratsumme ergibt sich der Standardschätzfehler $RMSE_p$ und daraus der Standardfehler der Steigung $SE(\beta_{1,p})$:

$$SSE_p = S_{yy,p} - \beta_{1,p} \cdot S_{xy,p}, \quad RMSE_p = \sqrt{\frac{SSE_p}{n_p - 2}}, \quad SE(\beta_{1,p}) = \frac{RMSE_p}{\sqrt{S_{xx,p}}}. \quad (3.4)$$

Dieser Standardfehler bildet die Basis für das 95%-Konfidenzintervall.

- **Signifikanztests (p -Werte):** Durch die Division eines Koeffizienten durch seinen Standardfehler wird der empirische t -Wert ermittelt:

$$t = \frac{\beta_{1,p}}{SE(\beta_{1,p})}. \quad (3.5)$$

Da relationale Datenbanken keine Standardbibliotheken für Verteilungsfunktionen bereitstellen, wird das Konzept um eine numerische Approximation der t -Verteilung ergänzt (konkret nach Pólya mit Li-De-Moor-Korrektur, vgl. Abschnitt 3.3.3), um den p -Wert direkt in SQL abzuleiten.

Aus den Koeffizienten beider Phasen werden schließlich die beiden HPLM-spezifischen Effekte gebildet – der Slope-Effekt als Differenz der Steigungen, der Level-Effekt als Differenz der Phasen-Vorhersagen am Interventionszeitpunkt t_{int} :

$$Slope = \beta_{1,B} - \beta_{1,A}, \quad (3.6)$$

$$Level = (\beta_{0,B} + \beta_{1,B}(t_{int} - 1)) - (\beta_{0,A} + \beta_{1,A}(t_{int} - 1)). \quad (3.7)$$

Damit wird der gesamte Auswertungsprozess – von der dynamischen Phasenbestimmung über die algebraische Parameterschätzung bis hin zur Signifikanzbewertung – am Ort der Datenhaltung (In-Database Analytics) formal vollständig beschrieben [5].

3.3.3. Approximation des p-Werts bei kleinen Stichproben

Der im vorigen Abschnitt eingeführte t -Wert $t = \beta_1 / SE(\beta_1)$ muss abschließend in einen p -Wert überführt werden. Dafür wird die kumulative Verteilungsfunktion der t -Verteilung benötigt, die PostgreSQL nicht nativ bereitstellt und daher numerisch angenähert werden muss. Das hierfür gewählte Verfahren wird im Folgenden erläutert; seine Umsetzung in PL/pgSQL erfolgt in Abschnitt 4.2.

Als Grundlage dient eine von Pólya [13] eingeführte geschlossene Näherung der kumulativen Standardnormalverteilung $\Phi(z)$:

$$\Phi(z) \approx \frac{1}{2} \left(1 + \sqrt{1 - \exp\left(-\frac{2}{\pi} z^2\right)} \right). \quad (3.8)$$

Der Vorteil dieser Formel für den SQL-Kontext liegt darin, dass sie ohne Tabellen auskommt und nur elementare Funktionen (Wurzel, Exponential) benötigt, die nativ verfügbar sind.

Die Pólya-Näherung approximiert allerdings die *Normal*verteilung. Bei den in der Einzelfallforschung typischen kleinen Stichproben weicht die t -Verteilung jedoch durch schwerere Verteilungsenden (*heavy tails*) von ihr ab. Eine reine Normalapproximation würde die Wahrscheinlichkeit in diesen Rändern unterschätzen und damit die Signifikanz von Effekten systematisch überschätzen. Aus diesem Grund wird der t -Wert vor dem Einsetzen mit einem Korrekturfaktor nach Li und De Moor [14] skaliert, der von den Freiheitsgraden ν abhängt:

$$\lambda(t, \nu) = \frac{4\nu + t^2 - 1}{4\nu + 2t^2}. \quad (3.9)$$

Anstelle von t wird der adjustierte Wert $z = t \cdot \lambda(t, \nu)$ in die Pólya-Näherung eingesetzt. Bei kleinen Freiheitsgraden zieht dieser Faktor den Wert so weit zusammen, dass die schwereren Enden der t -Verteilung berücksichtigt werden; mit wachsendem ν nähert sich

λ dem Wert 1, sodass die Korrektur bei großen Stichproben verschwindet. Die Korrektur reduziert den t -spezifischen Anteil des Approximationsfehlers auf $O(1/\nu^2)$; es verbleibt der von ν unabhängige Eigenfehler der Pólya-Näherung von rund 0,006 im zweiseitigen p -Wert. Der ausgewiesene p -Wert liegt dabei systematisch geringfügig unter dem exakten Wert, sodass Ergebnisse knapp unterhalb von $\alpha = 0,05$ zurückhaltend zu interpretieren sind. Das Verfahren ist dabei für $\nu \geq 3$ gültig; unterhalb dieser Grenze würde die unkorrigierte Normalnäherung die Signifikanz drastisch überschätzen; die Implementierung gibt daher für $\nu < 3$ konsistent NULL zurück, anstatt einen unzuverlässigen Wert auszuweisen.

Im Kontext der linearen Regression entsprechen die Freiheitsgrade ν der Anzahl der Messpunkte einer Phase abzüglich der zwei geschätzten Parameter (Achsenabschnitt und Steigung). Da die ausgegebenen p -Werte primär die Signifikanz von Achsenabschnitt und Trend der Baseline bewerten, wird konkret $\nu = n_A - 2$ verwendet, wobei n_A den Stichprobenumfang der Phase A bezeichnet. Daraus ergeben sich zwei unterschiedliche Grenzen: Die Punktschätzung selbst erfordert mindestens drei Messpunkte je Phase, da erst ab $n_A = 3$ ein positiver Freiheitsgrad ($\nu \geq 1$) vorliegt; ein p -Wert wird wegen der Gültigkeitsgrenze $\nu \geq 3$ dagegen erst ab $n_A = 5$ ausgewiesen und ist für $n_A \in \{3, 4\}$ NULL.

Damit ist das Konzept vollständig beschrieben: von der Architektur über die Dummy-Kodierung und die Mehrebenenaggregation bis zur Approximation der Signifikanzmaße. Kapitel 4 setzt diese Konzeption in PostgreSQL um und prüft anschließend empirisch, ob die datenbankinterne Berechnung mit der Referenzimplementierung übereinstimmt und wie sie mit wachsenden Datenmengen skaliert.

4. Implementierung des HPLM in PostgreSQL

In diesem Kapitel wird das in Kapitel 3 vorgestellte Konzept konkret in PostgreSQL umgesetzt. Die Implementierung folgt dem Muster der verwandten Arbeiten aus Abschnitt 2.3 – insbesondere dem Aggregat-basierten Ansatz von MADlib [10] – und besteht aus vier Schritten: dem Datenbankschema, einer Hilfsfunktion zur Approximation des p -Werts, dem eigentlichen HPLM-Aggregat sowie dem abschließenden `SELECT` mit `GROUPING SETS`.

4.1. Datenbankschema

Bevor das konkrete Schema dieser Arbeit vorgestellt wird, lässt sich die benötigte Datenstruktur zunächst abstrakt beschreiben. Für eine hierarchische Einzelfallanalyse werden grundsätzlich drei Arten von Informationen benötigt, die sich auf drei getrennte Relationen abbilden lassen:

- **Eine Subjekt-Relation** hält die Stammdaten der untersuchten Einheiten (Probanden) sowie deren Gruppierungsmerkmale, nach denen später aggregiert wird.
- **Eine Konfigurations-Relation** hält pro Subjekt den individuellen Interventionszeitpunkt, der den Wechsel von Phase A zu Phase B markiert.
- **Eine Mess-Relation (Faktentabelle)** hält die eigentlichen Längsschnittdaten, also pro Subjekt und Zeitpunkt einen Messwert.

Diese Dreiteilung folgt dem Prinzip der Normalisierung: Stammdaten, Konfigurationsdaten und Messwerte werden getrennt gehalten, sodass sie unabhängig voneinander gepflegt werden können und Redundanzen vermieden werden. Insbesondere die Auslagerung des Interventionszeitpunkts in eine eigene Relation sorgt dafür, dass der Phasenwechsel nicht im Programmcode festgeschrieben, sondern datengetrieben ist.

Angewendet auf den konkreten Datensatz dieser Arbeit ergeben sich daraus die drei in Listing 4.1 gezeigten Tabellen `personen`, `interventionen` und `messwerte`, welche die in Abschnitt 3.1 skizzierte Architektur umsetzen.

```
1 CREATE TABLE personen (  
2     person_id    INT PRIMARY KEY,  
3     name         VARCHAR(50),  
4     spiel        VARCHAR(50),  
5     geschlecht   VARCHAR(10),  
6     alter_jahre  INT  
7 );  
8  
9 CREATE TABLE interventionen (  
10    person_id INT PRIMARY KEY REFERENCES personen(person_id),  
11    start_tag INT NOT NULL  
12 );  
13  
14 CREATE TABLE messwerte (  
15    mess_id       SERIAL PRIMARY KEY,  
16    person_id     INT REFERENCES personen(person_id),  
17    tag           INT NOT NULL,  
18    reaktionszeit_ms NUMERIC,  
19    UNIQUE (person_id, tag)  
20 );
```

Listing 4.1: Datenbankschema mit drei normalisierten Tabellen.

Die Zuordnung von der abstrakten Struktur zum konkreten Schema ist dabei unmittelbar: Die Tabelle `personen` ist die Subjekt-Relation und enthält neben dem Primärschlüssel die Gruppierungsmerkmale `spiel`, `geschlecht` und `alter_jahre`, die später als Aggregationskriterien in den `GROUPING SETS` dienen. Die Tabelle `interventionen` ist die Konfigurations-Relation und speichert für jede Person den individuellen Interventionszeitpunkt (`start_tag`); dadurch ist der Phasenwechsel zwischen Phase A und Phase B nicht im SQL-Code hartkodiert, sondern wird zur Laufzeit dynamisch aus den relationalen Daten abgeleitet. Die Tabelle `messwerte` ist schließlich die Faktentabelle, in der jede Zeile eine einzelne Beobachtung (Reaktionszeit an einem Tag) für eine Person repräsentiert.

Die Integritätsbedingungen sichern dabei die Voraussetzungen der Auswertung ab: Der Primärschlüssel auf `interventionen` garantiert höchstens einen Interventionszeitpunkt je Person, sodass der `JOIN` keine Messzeilen vervielfacht, und die `UNIQUE`-Bedingung auf `messwerte` schließt doppelte Messungen zum selben Zeitpunkt aus. Beides ist für die Korrektheit der Freiheitsgrade und damit der Signifikanztests wesentlich.

4.2. Hilfsfunktion zur Approximation des p-Werts

Die theoretische Grundlage der hier verwendeten p -Wert-Approximation – die Pólya-Näherung sowie die Li-De-Moor-Korrektur – wurde bereits in Abschnitt 3.3.3 hergeleitet. Da PostgreSQL keine native Verteilungsfunktion der t -Verteilung bereitstellt, setzt die Funktion `approx_pwert` dieses Verfahren direkt in PL/pgSQL um. Listing 4.2 zeigt die Implementierung.

```
1 CREATE OR REPLACE FUNCTION approx_pwert(t_wert NUMERIC, df INT)
2 RETURNS NUMERIC AS $$
3 DECLARE
4     t_abs      FLOAT8;
5     lambda     FLOAT8;
6     z          FLOAT8;
7     p_approx   FLOAT8;
8 BEGIN
9 IF t_wert IS NULL OR df IS NULL OR df < 3 THEN
10     RETURN NULL;
11 END IF;
12 t_abs := ABS(t_wert::FLOAT8);
13 IF t_abs > 30 THEN
14     RETURN 0.0001;
15 END IF;
16 -- Korrekturfaktor nach Li & De Moor (1999) fuer kleine
17   Freiheitsgrade
18 lambda := (4.0 * df + t_abs * t_abs - 1.0)
19           / (4.0 * df + 2.0 * t_abs * t_abs);
20 z := t_abs * lambda;
21 p_approx := 2.0 * (1.0 - (0.5 * (1.0 +
22     SQRT(1.0 - EXP(-2.0 / PI() * z * z)))));
23 RETURN ROUND(GREATEST(0.0001, LEAST(1.0, p_approx))::NUMERIC, 4);
24 END;
25 $$ LANGUAGE plpgsql;
```

Listing 4.2: Approximation des zweiseitigen p -Werts mit Li-De-Moor-Korrektur.

Die Funktion setzt die in Gleichung 3.9 beschriebene Skalierung unmittelbar um: Für Freiheitsgrade $\nu \geq 3$ ($df \geq 3$) wird der adjustierte Wert $z = t \cdot \lambda(t, \nu)$ gebildet und in die Pólya-Näherung eingesetzt; für $\nu < 3$ gibt die Funktion konsistent zur Gültigkeitsgrenze des Verfahrens (vgl. Abschnitt 3.3.3) NULL zurück, anstatt einen unzuverlässigen Wert auszuweisen. Die abschließende Begrenzung über `GREATEST` und `LEAST` stellt sicher, dass der zurückgegebene p -Wert stets im gültigen Intervall $[0,0001; 1]$ liegt und auf vier Nachkommastellen gerundet wird.

4.3. Aggregat-basierte HPLM-Berechnung

Das Kernstück der Implementierung ist ein benutzerdefiniertes Aggregat (engl. *User-Defined Aggregate*, UDA), das die HPLM-Berechnung in zwei Phasen zerlegt: Eine *Transitions-Funktion* akkumuliert zeilenweise die benötigten Quadratsummen, während eine im Aggregat verankerte *Abschlussfunktion* daraus die Regressionskoeffizienten und Gütemaße berechnet. Dieser zweistufige Aufbau entspricht dem klassischen UDA-Muster, das auch in MADlib verwendet wird [10], und ermöglicht eine streamingfähige Verarbeitung großer Datenmengen.

4.3.1. Zustandsstruktur und Transitions-Funktion

Der Zustand des Aggregats wird als 13-elementiges `FLOAT8`-Array modelliert. Die Indizes 1–6 enthalten die Quadratsummen der Phase A ($\sum x$, $\sum y$, $\sum xy$, $\sum x^2$, $\sum y^2$, n), die Indizes 7–12 enthalten die entsprechenden Summen der Phase B, und Index 13 hält den dynamischen Interventionszeitpunkt fest.

```
1 CREATE OR REPLACE FUNCTION hplm_sfunc(  
2     state FLOAT8[], y NUMERIC, t INT, start_tag INT  
3 ) RETURNS FLOAT8[] AS $$  
4 DECLARE  
5     px FLOAT8;  
6     py FLOAT8;  
7 BEGIN  
8     IF y IS NULL THEN  
9         RETURN state;  
10    END IF;  
11    px := (t - 1)::FLOAT8;  
12    py := y::FLOAT8;  
13    IF start_tag > state[13] THEN  
14        state[13] := start_tag;  
15    END IF;  
16    IF t < start_tag THEN  
17        state[1] := state[1] + px;  
18        state[2] := state[2] + py;  
19        state[3] := state[3] + px * py;  
20        state[4] := state[4] + px * px;  
21        state[5] := state[5] + py * py;  
22        state[6] := state[6] + 1;  
23    ELSE  
24        state[7] := state[7] + px;  
25        state[8] := state[8] + py;  
26        state[9] := state[9] + px * py;  
27        state[10] := state[10] + px * px;  
28        state[11] := state[11] + py * py;  
29        state[12] := state[12] + 1;
```

```

30     END IF;
31     RETURN state;
32 END;
33 $$ LANGUAGE plpgsql;

```

Listing 4.3: Transitions-Funktion `hplm_sfunc` (auf die Phasenzuordnung gekürzt).

Für jede einzelne Zeile entscheidet die Bedingung `t < start_tag`, ob der Messwert zu Phase A oder Phase B gehört. Diese Phasenzuordnung ist genau die in Abschnitt 3.2 beschriebene Dummy-Kodierung. Zeilen mit fehlendem Messwert (`y IS NULL`) werden dabei übersprungen, ohne den Aggregatzustand zu verändern – relevant etwa für Datensätze mit Lücken in der Erfassung. Da die Funktion algorithmisch ausschließlich auf Additionen und Multiplikationen basiert, skaliert sie linear mit der Anzahl der verarbeiteten Datensätze. Das Aggregat selbst wird über folgenden Befehl registriert:

```

1 CREATE AGGREGATE hplm(NUMERIC, INT, INT) (
2     SFUNC      = hplm_sfunc,
3     STYPE      = FLOAT8[],
4     FINALFUNC  = hplm_ffunc,
5     INITCOND   = '{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}'
6 );

```

Listing 4.4: Definition des benutzerdefinierten Aggregats `hplm`.

Die Klausel `FINALFUNC` verankert die Abschlussfunktion im Aggregat selbst, sodass bereits der Aggregataufruf die fertigen Kennwerte liefert und sich das HPLM nach außen wie eine native Aggregatfunktion verwenden lässt (vgl. Listing 3.1). Da PostgreSQL als Rückgabebetyp einer `FINALFUNC` keinen anonymen Verbund akzeptiert, kapselt die Funktion `hplm_ffunc` die in Abschnitt 4.3.2 beschriebene Berechnung und gibt das Ergebnis als benannten Verbundtyp zurück.

Die Aggregatdefinition verzichtet bewusst auf eine Kombinier-Funktion (`COMBINEFUNC`), über die PostgreSQL Teilzustände parallel arbeitender Prozesse zusammenführen könnte. Damit ist die Auswertung stets seriell und die Additionsreihenfolge der `FLOAT8`-Summen fest – wiederholte Läufe liefern bitidentische Ergebnisse. Der Determinismus der Berechnung ist somit konstruktionsbedingt garantiert; die mit einer Kombinier-Funktion mögliche parallele Aggregation wird im Ausblick als Erweiterung benannt, da sie durch die dann variable Summationsreihenfolge minimale Gleitkomma-Abweichungen zwischen Läufen einführen kann.

4.3.2. Berechnung der HPLM-Kennwerte

Aus den vom Aggregat gesammelten Summen werden in der Funktion `berechne_hplm_vollstaendig` die HPLM-Kennwerte abgeleitet. Diese Funktion wird von der Abschlussfunktion `hplm_ffunc` aufgerufen, die das Ergebnis in einen benannten Verbundtyp verpackt (vgl. Listing 4.4). Die Funktion erhält das Summen-Array als Eingabe und liefert über OUT-Parameter alle relevanten Werte: Intercept, Trend, Level-Sprung, Slope-Änderung, die Bestimmtheitsmaße R^2 beider Phasen, den Standardfehler des Trends, das 95 %-Konfidenzintervall, die approximierten p -Werte sowie die Stichprobenumfänge. Die dabei verwendeten Formeln wurden bereits in Abschnitt 3.3 hergeleitet (vgl. die Gleichungen 3.1 bis 3.7): Aus den getrennt akkumulierten Summen beider Phasen werden zunächst die Streuungssummen S_{xx} , S_{yy} und S_{xy} gebildet, daraus die Koeffizienten β_0 und β_1 , und schließlich die Gütemaße sowie die phasenübergreifenden Effekte Level und Slope. Die Implementierung setzt diese algebraischen Gleichungen direkt in PL/pgSQL um; das folgende Listing zeigt die zentralen Berechnungsschritte.

Das Listing ist auf die strukturtragenden Schritte gekürzt: Die Deklarationen der Hilfsvariablen sowie die zu Phase A analogen Berechnungen der Phase B sind mit `-- ...` angegeben. Zwei Details verdienen Erwähnung: Das Bestimmtheitsmaß wird als $1 - SSE/S_{yy}$ berechnet, was bei der einfachen linearen Regression algebraisch mit Gleichung 3.3 übereinstimmt, numerisch aber stabiler ist. Der Standardfehler des Achsenabschnitts folgt der Formel $SE(\beta_0) = RMSE \sqrt{1/n + \bar{x}^2/S_{xx}}$ und geht in den p -Wert des Intercepts ein. Die `GREATEST`-Aufrufe fangen negative Rundungsartefakte in der Residuenquadratsumme ab, die bei nahezu perfekter Modellanpassung entstehen können.

```
1 CREATE OR REPLACE FUNCTION berechne_hplm_vollstaendig(  
2     state FLOAT8[],  
3     OUT intercept NUMERIC, OUT trend NUMERIC,  
4     OUT level_sprung NUMERIC, OUT slope_aenderung NUMERIC,  
5     OUT r2_phase_a NUMERIC, OUT se_trend NUMERIC,  
6     OUT p_wert_trend NUMERIC, ...      -- weitere OUT-Parameter  
7 ) RETURNS record AS $$  
8 DECLARE  
9     sum_xa FLOAT8 := state[1]; -- ... Summen der Phase A  
10    na     FLOAT8 := state[6];  
11    sum_xb FLOAT8 := state[7]; -- ... Summen der Phase B  
12    nb     FLOAT8 := state[12];  
13    t_int  FLOAT8 := state[13]; -- Interventionszeitpunkt  
14    -- ... weitere Hilfsvariablen  
15 BEGIN  
16     -- Mindestens drei Messpunkte je Phase (vgl. Abschnitt 3.3.3)  
17     IF state IS NULL OR na < 3 OR nb < 3 THEN RETURN; END IF;  
18     -- Zentrierte Streuungssummen der Phase A (Gleichung 3.1)  
19     mean_xa := sum_xa / na;  
20     mean_ya := sum_ya / na;  
21     sxx_a   := sum_x2a - na * mean_xa * mean_xa;  
22     syy_a   := sum_y2a - na * mean_ya * mean_ya;
```

```

23  sxy_a    := sum_xya - na * mean_xa * mean_ya;
24  IF sxx_a <= 0 THEN RETURN; END IF;
25  -- Regressionskoeffizienten der Phase A (Gleichung 3.2)
26  b1_a := sxy_a / sxx_a;
27  b0_a := mean_ya - b1_a * mean_xa;
28  -- Guetemasse und Standardfehler (Gleichungen 3.3 und 3.4)
29  sse_a := GREATEST(syy_a - b1_a * sxy_a, 0);
30  r2a    := CASE WHEN syy_a > 0 THEN 1 - sse_a / syy_a ELSE 0 END;
31  rmse_a := SQRT(sse_a / GREATEST(na - 2, 1));
32  se_t    := rmse_a / SQRT(sxx_a);
33  se_b0    := rmse_a * SQRT(1.0 / na + (mean_xa * mean_xa) / sxx_a);
34  -- Phase B analog: sxx_b, sxy_b, b1_b, b0_b, r2b
35  -- ...
36  -- Phasenubergreifende Effekte (Gleichungen 3.6 und 3.7)
37  level_sprung := ROUND(((b0_b + b1_b * (t_int - 1))
38                        - (b0_a + b1_a * (t_int - 1)))::NUMERIC, 4);
39  slope_aenderung := ROUND((b1_b - b1_a)::NUMERIC, 4);
40  -- Konfidenzintervall (Abschnitt 3.3.2) und p-Werte (Abschnitt 4.2)
41  ki_trend_unten := ROUND((b1_a - 2.0 * se_t)::NUMERIC, 4);
42  ki_trend_oben  := ROUND((b1_a + 2.0 * se_t)::NUMERIC, 4);
43  p_wert_trend := approx_pwert((b1_a / NULLIF(se_t, 0))::NUMERIC,
44                               (na - 2)::INT);
45  -- Rundung und Zuweisung an die OUT-Parameter
46  intercept := ROUND(b0_a::NUMERIC, 4);
47  trend      := ROUND(b1_a::NUMERIC, 4);
48  r2_phase_a := ROUND(r2a::NUMERIC, 4);
49  se_trend    := ROUND(se_t::NUMERIC, 4);
50  -- ...
51  END;
52  $$ LANGUAGE plpgsql;

```

Listing 4.5: Berechnung der HPLM-Kennwerte aus den akkumulierten Summen (auf die zentralen Schritte gekürzt).

4.4. Hierarchische Auswertung mit GROUPING SETS

Der eigentliche Aufruf des HPLM erfolgt über eine einzige **SELECT**-Anweisung, die mehrere Aggregationsebenen kombiniert. Angewandt auf das konkrete Datenmodell dieser Arbeit (Reaktionszeiten von Probanden) ergibt sich der in Listing 4.6 dargestellte SQL-Aufruf, der die in Listing 3.3 eingeführte Schablone konkretisiert. Über zwei **JOINS** werden die drei Tabellen verbunden; anschließend werden die Daten gleichzeitig auf vier Ebenen ausgewertet.

```

1 SELECT
2     p.person_id,
3     p.spiel,
4     p.geschlecht,
5     (hplm(m.reaktionszeit_ms, m.tag, i.start_tag)).*
6 FROM messwerte m
7 JOIN personen p ON m.person_id = p.person_id
8 JOIN interventionen i ON m.person_id = i.person_id
9 GROUP BY GROUPING SETS (
10    (p.person_id, p.spiel, p.geschlecht),
11    (p.spiel),
12    (p.geschlecht),
13    ()
14 );

```

Listing 4.6: Hierarchische HPLM-Auswertung mit `GROUPING SETS` (konkretes Beispiel).

Konkret werden in diesem Aufruf die folgenden vier Auswertungsebenen parallel berechnet:

1. **Individualebene** (`person_id`, `spiel`, `geschlecht`) – für jeden einzelnen Probanden wird ein eigenständiges HPLM geschätzt. Auf dieser Ebene werden keine Informationen zwischen den Personen geteilt (Level 1).
2. **Spielebene** (`spiel`) – die Messungen aller Probanden eines Spiels werden zu einem gepoolten HPLM zusammengefasst (Level 2).
3. **Geschlechterebene** (`geschlecht`) – analog werden die Messungen nach Geschlecht zu einem gepoolten HPLM verdichtet (Level 2).
4. **Gesamtebene** (`()`) – alle Messungen fließen in ein einziges HPLM über die gesamte Kohorte ein.

Das Aggregat `hplm` akkumuliert dabei pro Gruppe die benötigten Summen (über die Transitions-Funktion `hplm_sfunc`) und überführt sie in der Abschlussfunktion in die finalen Kennwerte (R^2 , p -Werte, Konfidenzintervalle). Diese Konstruktion bildet den zentralen strukturellen Vorteil gegenüber einer iterativen Verarbeitung in externer Statistiksoftware: Anstelle von vier separaten Anfragen mit jeweils eigenem Tabellendurchlauf führt PostgreSQL alle vier Aggregationen innerhalb eines einzigen Scans durch. Damit ist die Forschungsfrage zur *Umsetzbarkeit* aus Kapitel 1 positiv beantwortet: Die vollständige mathematische Logik eines HPLM – einschließlich der Berechnung von Konfidenzintervallen und der numerischen Approximation von p -Werten – lässt sich in nativen SQL- und PL/pgSQL-Operationen abbilden. Die in Kapitel 5 dokumentierte Evaluation zeigt zudem, dass die datenbankintern erzeugten Schätzer mit den Ergebnissen der etablierten Referenzimplementierung `scan::hplm()` übereinstimmen.

5. Evaluation

Dieses Kapitel prüft die in Kapitel 4 entwickelte Implementierung in drei Richtungen: Abschnitt 5.1 validiert die berechneten Kennwerte gegen die Referenzimplementierung `scan`, die Abschnitte 5.2 und 5.3 untersuchen das Laufzeitverhalten bei wachsender Datenmenge und wachsender Zahl an Aggregationsebenen, und Abschnitt 5.4 grenzt den gewählten Weg gegen alternative Realisierungsformen ab.

5.1. Validierung gegen die Referenzimplementierung

Zur Beantwortung der dritten Forschungsfrage aus Kapitel 1 – der *Validität* der berechneten Ergebnisse – wird die SQL-Implementierung in diesem Abschnitt mit dem etablierten R-Paket `scan` (Version 0.68.1) verglichen, das in der Einzelfallforschung als methodischer Goldstandard gilt (vgl. Abschnitt 2.3). Als Referenzen dienen die Funktionen `plm()` (Piecewise Linear Model je Fall) und `hplm()` (Hierarchical Piecewise Linear Model über mehrere Fälle).

5.1.1. Versuchsaufbau

Der Testdatensatz besteht aus vier Probanden (zwei *Starcraft*- und zwei *Call of Duty*-Spieler (CoD), jeweils ein Mann und eine Frau), für die jeweils 20 Reaktionszeitmessungen vorliegen. Der Interventionszeitpunkt liegt bei allen Probanden einheitlich bei `start.tag = 11`, sodass Phase A jeweils zehn und Phase B ebenfalls zehn Messungen umfasst.

Diese balancierte Datenstruktur ist methodisch wichtig: Wie in Abschnitt 6.2 diskutiert, schätzt der SQL-Ansatz die Parameter über die Methode der kleinsten Quadrate (Ordinary Least Squares, OLS), während `scan` intern auf das iterative *Restricted Maximum Likelihood*-Verfahren (REML) zurückgreift. Bei streng balancierten Daten liefern beide Verfahren für die *Fixed Effects* mathematisch identische Schätzungen [15]; bei Varianzkomponenten und Standardfehlern sind methodisch bedingte Abweichungen zu erwarten. Das verwendete R-Skript ist in Anhang A dokumentiert.

5.1.2. Ergebnisse der SQL-Implementierung

Tabelle 5.1 zeigt zunächst die vollständigen HPLM-Kennwerte, die die SQL-Implementierung über alle Auswertungsebenen hinweg berechnet. Die ersten vier Zeilen enthalten die individuellen Schätzungen je Proband (Level 1), die mittleren vier die nach Spiel und Geschlecht gepoolten Kohorteneffekte und die letzte Zeile das Ergebnis über die gesamte Stichprobe (Level 2). Sämtliche neun Ergebniszeilen entstehen dabei in einem einzigen Tabellendurchlauf durch die `GROUPING SETS`-Klausel aus Listing 4.6; die folgenden Abschnitte vergleichen diese Werte schrittweise mit den Ergebnissen von `scan`.

Gruppe	Intercept	Trend	Level	Slope	R^2 A
Player_1 (Starcraft, m)	251,9818	-0,2182	-31,8545	-3,6364	0,0234
Player_2 (CoD, m)	299,1818	-3,0848	-28,3333	-1,9152	0,9739
Player_3 (Starcraft, w)	268,2909	-0,9758	-30,3152	-3,8061	0,2921
Player_4 (CoD, w)	310,1636	-1,8364	-33,4545	-3,0182	0,9006
Starcraft (gesamt)	260,1364	-0,5970	-31,0848	-3,7212	0,0463
CoD (gesamt)	304,6727	-2,4606	-30,8939	-2,4667	0,4008
Männlich (gesamt)	275,5818	-1,6515	-30,0939	-2,7758	0,0656
Weiblich (gesamt)	289,2273	-1,4061	-31,8848	-3,4121	0,0418
Gesamtkohorte	282,4045	-1,5288	-30,9894	-3,0939	0,0458

Tabelle 5.1.: HPLM-Kennwerte aus der PostgreSQL-Implementierung über alle neun Ergebniszeilen. „ R^2 A“ bezeichnet das Bestimmtheitsmaß der Phase A.

5.1.3. Vergleich mit scan auf der Individualebene

Tabelle 5.2 stellt den Trend (den zentralen, modellunabhängigen Phase-A-Koeffizienten) der vier Probanden dem Ergebnis von `scan::plm()` gegenüber.

Proband	Trend (scan)	Trend (SQL)	Differenz
Player_1	-0,218	-0,2182	0,0002
Player_2	-3,085	-3,0848	0,0002
Player_3	-0,976	-0,9758	0,0002
Player_4	-1,836	-1,8364	0,0004

Tabelle 5.2.: Vergleich des Trend-Koeffizienten zwischen `scan::plm()` und der SQL-Implementierung. Die Differenzen sind ausschließlich rundungsbedingt: `scan` gibt in der Standardausgabe drei Nachkommastellen aus, die SQL-Implementierung vier.

Die verbleibenden Differenzen sind allein der unterschiedlichen Ausgabegenauigkeit geschuldet: `scan` rundet in der Standardausgabe auf drei Nachkommastellen, während die SQL-Implementierung vier ausgibt. Innerhalb der von `scan` ausgewiesenen Genauigkeit

stimmen die Werte exakt überein. Dies bestätigt, dass die in den Abschnitten 3.2 und 4.3 entwickelte SQL-Logik die zugrundeliegende Regressionsmathematik korrekt umsetzt. Bei den Effekten Intercept und Level treten dagegen systematische Differenzen auf, deren Ursache im folgenden Zusatztest (Abschnitt 5.1.7) analysiert wird; sie gehen, wie dort gezeigt wird, nicht auf Rechenfehler, sondern auf eine unterschiedliche Dummy-Kodierung zurück.

5.1.4. Vergleich auf der Aggregatebene

Auf der Aggregatebene wird `scan::hplm()` (REML) über alle vier Probanden mit der Gesamtebene der SQL-Auswertung verglichen:

Fixed Effect	scan (B&L-B)	SQL (W)	SQL nach Transf.	Abw.
Intercept	283,933	282,4045	283,9333	< 0,001
Trend	-1,529	-1,5288	-1,5288	< 0,001
Level	-27,895	-30,9894	-27,8955	< 0,001
Slope	-3,094	-3,0939	-3,0939	< 0,001

Tabelle 5.3.: Vergleich der gepoolten Schätzungen über alle vier Probanden zwischen `scan::hplm()` und der SQL-Implementierung. Die vierte Spalte zeigt die SQL-Werte nach Anwendung der Kodierungstransformation; `scan` rechnet in der Voreinstellung im B&L-B-Modell, die SQL-Implementierung im W-Modell. Die letzte Spalte gibt die Abweichung zwischen den Spalten *scan (B&L-B)* und *SQL nach Transf.* an, also den Vergleich innerhalb desselben Kodierungsmodells.

Auch hier zeigt sich das erwartete Muster: Trend und Slope stimmen praktisch exakt überein – erwartungsgemäß, da OLS und REML bei balancierten Daten für diese Fixed Effects identische Schätzer liefern [15]. Die Differenzen bei Intercept und Level gehen dagegen auf die unterschiedliche Kodierung zurück: `scan::hplm()` rechnet in der Voreinstellung im B&L-B-Modell, die SQL-Implementierung im W-Modell (vgl. Abschnitt 5.1.7). Nach der dort hergeleiteten Transformation stimmen alle vier Fixed Effects auf unter 0,001 überein; die Differenz zwischen der `scan`-Spalte und der untransformierten SQL-Spalte beträgt 1,53 beim Intercept und 3,09 beim Level und entspricht damit dem Betrag des Trends beziehungsweise des Slopes. Der Unterschied zwischen Complete Pooling und Partial Pooling wirkt sich bei diesem balancierten Design somit nicht auf die Punktschätzer aus, sondern auf Standardfehler und Varianzkomponenten; er ist in Abschnitt 6.2 eingeordnet.

5.1.5. Hinweis zu Standardfehlern und p -Werten

Die SQL-Implementierung verwendet eine numerische Approximation der t -Verteilung nach Pólya [13] mit Li-De-Moor-Korrektur [14] (vgl. Abschnitt 3.3.3), während `scan` auf die in R nativ verfügbaren exakten Verteilungsfunktionen zurückgreift. Bei kleinen Stichprobenumfängen (im Testdatensatz $n_A = 10$, also $\nu = 8$ Freiheitsgrade) führt dies zu Abweichungen in den p -Werten. Die Standardfehler weichen davon unabhängig ab, da die SQL-Variante sie aus der phasenweisen Regression ableitet, `scan` dagegen aus dem gemeinsamen Mehrebenenmodell. Beide Abweichungen betreffen nicht die Punktschätzer, sondern allein deren Inferenzstatistik.

5.1.6. Interpretation der Ergebnisse

Für die Bewertung der Implementierung ist nicht die inhaltliche Deutung der Messwerte relevant, sondern die Frage, ob das SQL-Aggregat die vier HPLM-Effekte korrekt voneinander separiert. Genau diese Trennleistung lässt sich an den R^2 -Werten der Phase A ablesen: Bei zwei Probanden liegt $R^2 \approx 0,02$ bzw. $0,29$ (kein nennenswerter Baseline-Trend), bei den beiden anderen $R^2 \approx 0,97$ bzw. $0,90$ (starker Trend bereits in Phase A). Dieser Kontrast ist aus algorithmischer Sicht der entscheidende Testfall: Liegt bereits in der Baseline ein ausgeprägter Trend vor, muss das Modell den nachfolgenden Level-Sprung sauber von diesem vorbestehenden Trend abgrenzen. Dass die SQL-Implementierung in beiden Konstellationen denselben Trend-Koeffizienten wie `scan::plm()` liefert, belegt, dass die phasenbezogene Dummy-Kodierung und die getrennte Schätzung der Phasen-Koeffizienten korrekt arbeiten.

5.1.7. Zusätzliche Validierung an einem unabhängigen Datensatz

Um auszuschließen, dass die Übereinstimmung der Ergebnisse auf Besonderheiten des selbst erstellten Datensatzes beruht, wurde die SQL-Implementierung an einem zweiten, unabhängigen Datensatz geprüft. Hierfür wurde der im `scan`-Paket fest eingebaute Beispieldatensatz `exampleAB` herangezogen, der drei Fälle (Johanna, Karolina, Anja) umfasst. Im Gegensatz zum balancierten Hauptdatensatz ist `exampleAB` *unbalanciert*: Auf fünf Messungen in Phase A folgen fünfzehn Messungen in Phase B. Dieser Datensatz ist damit ein bewusst gewählter, anspruchsvollerer Testfall.

Befund. Tabelle 5.4 stellt die Ergebnisse von `scan::plm()` und der SQL-Implementierung für die drei Fälle gegenüber; `scan` wurde dabei mit der Option `model = "W"`

aufgerufen. In allen drei Fällen stimmen sämtliche vier Kennwerte – Intercept, Trend, Level und Slope – bis auf die dritte Nachkommastelle überein; die ausgewiesenen Differenzen liegen sämtlich unterhalb von 0,001 und sind allein der unterschiedlichen Zahl ausgegebener Nachkommastellen geschuldet. Die zuvor am Hauptdatensatz beobachteten Differenzen bei Intercept und Level sind somit keine Rechenfehler, sondern allein eine Frage der Modellwahl, wie der folgende Unterabschnitt zeigt.

Fall	Kennwert	scan (W)	SQL	Diff.
Johanna	Intercept	54,400	54,4000	< 0,001
	Trend	0,100	0,1000	< 0,001
	Level	7,858	7,8583	< 0,001
	Slope	1,525	1,5250	< 0,001
Karolina	Intercept	49,000	49,0000	< 0,001
	Trend	1,400	1,4000	< 0,001
	Level	3,892	3,8917	< 0,001
	Slope	0,539	0,5393	< 0,001
Anja	Intercept	56,400	56,4000	< 0,001
	Trend	−1,400	−1,4000	< 0,001
	Level	16,967	16,9667	< 0,001
	Slope	2,500	2,5000	< 0,001

Tabelle 5.4.: Vergleich zwischen `scan::plm()` mit `model = "W"` (Version 0.68.1) und der SQL-Implementierung für `exampleAB`.

Verhältnis zur Modellwahl in scan Die exakte Übereinstimmung wirft die Frage auf, welchem der in `scan` vorgesehenen Kodierungsmodelle die SQL-Implementierung entspricht. `scan` erlaubt die Wahl des Dummy-Kodierungsmodells über den Parameter `model`; verfügbar sind „W“, „H-M“ (Huitema-McKean), „B&L-B“ und „JW“, deren Kodierungsschemata sämtlich auf Huitema und McKean [16] zurückgehen. Diese Modelle unterscheiden sich nicht in der Modellanpassung selbst – die geschätzten Regressionsgeraden, die Vorhersagewerte und das Bestimmtheitsmaß sind identisch –, sondern allein im Bezugspunkt, an dem die Effekte *Level* und *Slope* abgelesen werden.

Die in dieser Arbeit gewählte Operationalisierung – zwei getrennte Phasen-Regressionen mit dem Slope-Effekt als Differenz der Steigungen und dem Level-Effekt als Differenz der Phasen-Vorhersagen am Interventionszeitpunkt – entspricht exakt dem W-Modell. Der Vergleich in Tabelle 5.4 wurde daher mit `model = "W"` durchgeführt und zeigt vollständige Übereinstimmung in allen vier Kennwerten.

Wertet man `scan` dagegen in seiner Voreinstellung `model = "B&L-B"` aus, so weichen Intercept und Level um einen festen, modellbedingten Betrag ab – der Intercept um den Wert des Trends, der Level um den Wert des Slopes –, während Trend und Slope unverändert bleiben. Diese Differenz ist somit keine Frage der Korrektheit, sondern ausschließlich der Wahl des Bezugspunktes; sie verschwindet vollständig, sobald beide Implementierungen dasselbe Modell verwenden.

Umsetzung der Modellvarianten in SQL. Da sich die verschiedenen Modellierungsansätze ausschließlich in der Definition ihres zeitlichen Bezugspunktes unterscheiden, lassen sich die scan-Varianten „H-M“ und „B&L-B“ ohne erneute Regression unmittelbar aus den Kennwerten des relational geschätzten W-Modells ableiten. Die eigentliche Modellanpassung an die Datenpunkte – und damit die modellinvarianten Größen *Trend* und *Slope* – bleibt über alle Varianten hinweg identisch. Lediglich die Lageparameter *Intercept* und *Level* verschieben sich um einen festen, modellbedingten Betrag:

$$Intercept_{H-M} = Intercept_W - Trend, \quad Level_{B\&L-B} = Level_W - Slope. \quad (5.1)$$

Das Modell „B&L-B“ kombiniert dabei beide Transformationen: Es übernimmt die Intercept-Korrektur des „H-M“-Ansatzes und korrigiert zusätzlich den *Level*-Parameter. Diese Transformation wurde im Rahmen dieser Arbeit als eigene SQL-Funktion abgebildet, die aus den W-Kennwerten die Werte der übrigen Modellvarianten erzeugt. Das in der aktuellen `scan`-Version als *deprecated* (veraltet) markierte Modell „JW“ wird dabei bewusst ausgeklammert.

Tabelle 5.5 stellt die datenbankintern abgeleiteten Werte den Ausgaben von `scan` gegenüber. Für alle drei untersuchten Fälle und alle Modellvarianten stimmen die über SQL transformierten Kennwerte exakt mit den `plm()`-Ergebnissen überein.

Fall	Modell	Intercept	Level
Johanna	W	54,400	7,858
	H-M	54,300	7,858
	B&L-B	54,300	6,333
Karolina	W	49,000	3,892
	H-M	47,600	3,892
	B&L-B	47,600	3,352
Anja	W	56,400	16,967
	H-M	57,800	16,967
	B&L-B	57,800	14,467

Tabelle 5.5.: In SQL abgeleitete Intercept- und Level-Werte der drei scan-Modelle für die exampleAB-Fälle. Die Werte sind mit den `plm()`-Ausgaben von `scan` identisch. Trend und Slope sind modellinvariant und daher nicht aufgeführt (Johanna: 0,100 / 1,525; Karolina: 1,400 / 0,539; Anja: −1,400 / 2,500).

5.2. Skalierungs-Benchmark auf einem realistischen Datensatz

Zur Beurteilung des Laufzeitverhaltens wurde die HPLM-Implementierung auf einem großen, realistischen Datensatz getestet. Als Grundlage dienten die stündlichen Werte der realisierten Stromerzeugung aus dem SMARD-Portal der Bundesnetzagentur [17]. Jede Erzeugungsart (Biomasse, Wind, Photovoltaik, Braunkohle usw.) wird dabei als ein Fall aufgefasst, jede Stunde als ein Messzeitpunkt; der Interventionszeitpunkt wurde künstlich auf die Mitte der Zeitreihe gesetzt, da der Benchmark die Ausführungszeit und nicht die inhaltliche Interpretation misst.

Tabelle 5.6 zeigt einen Auszug des so aufgebauten Grunddatensatzes rund um den künstlich gesetzten Interventionszeitpunkt (Stunde 125): Jede Zeile enthält den Fall (`fall_id`), den Messzeitpunkt (`tag`) sowie die in dieser Stunde realisierte Erzeugung (`wert`). Die Spalte `tag` trägt dabei ihren generischen Namen aus dem Schema der Faktentabelle `messwerte` (vgl. Tabelle 3.1), bezeichnet in diesem Datensatz jedoch die fortlaufende *Stunde* der Zeitreihe (1 bis 250). Da das HPLM-Aggregat lediglich einen ganzzahligen Zeitindex erwartet, ist die zeitliche Granularität der Messpunkte für die Berechnung unerheblich, und die Tabellenstruktur kann unverändert übernommen werden.

<code>fall_id</code>	<code>tag</code>	<code>wert</code>
101	124	3 621,99
101	125	3 655,90
101	126	3 730,80
102	124	7 661,40
102	125	7 666,57
102	126	7 682,95

Tabelle 5.6.: Auszug des SMARD-Benchmark-Datensatzes rund um den Interventionszeitpunkt (Stunde 125). Jede Erzeugungsart bildet einen Fall (`fall_id`); `tag` bezeichnet hier den fortlaufenden Stundenindex, `wert` die realisierte Erzeugung in MWh.

Um das Skalierungsverhalten über mehrere Größenordnungen zu prüfen, wurde dieser Grunddatensatz durch Vervielfältigung der Fälle auf vier Stufen hochskaliert – von 2 668 bis über eine Million Messpunkte. Zwar stellt das SMARD-Portal von sich aus ausreichend große Datenmengen bereit (ein volles Jahr entspricht in Stundenauflösung bereits rund 115 000, in Viertelstundenauflösung über 450 000 Tupeln), doch würde eine Verlängerung des Abrufzeitraums die Zeitreihen *je Fall* verlängern, während die Zahl der Fälle konstant bei den elf Erzeugungsarten bliebe. Für die HPLM-Auswertung ist jedoch die Skalierung über die *Fälle* die relevante Dimension: Der Anwendungsfall der

Einzelfallanalyse besteht aus vielen Subjekten mit Zeitreihen moderater Länge, nicht aus wenigen extrem langen Reihen, und auch die Gruppierung (`GROUP BY fall_id`) partitioniert die Last über die Fälle. Die Vervielfältigung der Fälle skaliert daher genau diese Dimension und hält zugleich Serienlänge, Interventionszeitpunkt und Datensatzstruktur über alle vier Stufen konstant, sodass gemessene Laufzeitunterschiede allein auf die Datenmenge zurückgehen. Jede Stufe wurde zehnmal ausgeführt und die Laufzeit über `clock_timestamp()` gemessen (PostgreSQL 18); ergänzend wurde jede Stufe einmalig mit `EXPLAIN (ANALYZE, BUFFERS)` ausgewertet, um den Ausführungsplan zu analysieren (vgl. den Absatz zur Planstabilität). Alle Messungen wurden auf einem System mit AMD Ryzen 5 5600X (6 Kerne, 3,70 GHz), 16 GB Arbeitsspeicher und Windows 11 Pro (64-Bit) durchgeführt; PostgreSQL wurde dabei in der Standardkonfiguration betrieben (`work_mem` = 4 MB). Das zusammengesetzte Aggregatergebnis wird dabei in einer CTE einmalig berechnet und erst anschließend entpackt; dies verhindert eine wiederholte Auswertung der Abschlussfunktion je Ausgabespalte und misst somit allein die eigentliche HPLM-Last.

Hypothese: Aus dem Entwurf des Aggregats ergibt sich eine klare Erwartung: Da die für die Regression benötigten Größen als laufende Summen in einem einzigen Tabellendurchlauf gebildet werden – ohne mehrfaches Lesen oder Sortieren der Daten – wird eine *lineare* Skalierung der Laufzeit mit der Zahl der Messpunkte ($O(n)$) und damit eine nahezu konstante Bearbeitungszeit je Zeile erwartet. Tabelle 5.7 zeigt die gemessenen Ausführungszeiten.

Messpunkte	Laufzeit (10 Läufe)	pro Zeile
2 668	$6,2 \pm 2,1$ ms	$2,31 \mu\text{s}$
10 672	$22,3 \pm 3,4$ ms	$2,09 \mu\text{s}$
101 384	$196,2 \pm 3,7$ ms	$1,93 \mu\text{s}$
1 000 500	$1\,919,6 \pm 59,9$ ms	$1,92 \mu\text{s}$

Tabelle 5.7.: Ausführungszeit der HPLM-Aggregatabfrage auf hochskalierten SMARD-Daten als Mittelwert $\pm$ Standardabweichung über zehn Messläufe (gemessen mit `clock_timestamp()`; die Einzeldurchläufe sind in Anhang C dokumentiert). Die Zeit je Zeile (aus dem Mittelwert) bleibt über vier Größenordnungen nahezu konstant.

Lineare Skalierung. Die Messung bestätigt diese Hypothese: Die Ausführungszeit pro Zeile bleibt über alle vier Stufen nahezu konstant bei rund $2 \mu\text{s}$. Der leicht fallende Wert bei den größeren Stufen erklärt sich dadurch, dass sich fixe Kosten (Anfrageplanung, Initialisierung des Aggregatzustands je Gruppe) dort auf deutlich mehr Zeilen verteilen. Die Gesamtlaufzeit wächst damit streng linear mit der Zahl der Messpunkte ($O(n)$). Dieses Verhalten ist eine direkte Folge des Aggregat-Entwurfs: Die für die Regression benötigten Größen werden als laufende Summen in einem einzigen Tabellendurchlauf gebildet, sodass die Daten weder mehrfach gelesen noch sortiert werden müssen.

```

1 CTE Scan on agg (actual time=2142.133..2218.562 rows=4125)
2   CTE agg
3     -> HashAggregate (actual time=2142.113..2214.057 rows=4125)
4       Group Key: smard_1m.fall_id
5       Batches: 1  Memory Usage: 1169kB
6     -> Seq Scan on smard_1m (rows=1000500)
7 Planning Time: 0.480 ms
8 Execution Time: 2220.037 ms

```

Listing 5.1: Anfrageplan der HPLM-Aggregatabfrage auf der größten Skalierungsstufe (smard_1m, gekürzt).

Planstabilität. Listing 5.1 zeigt den mit EXPLAIN (ANALYZE, BUFFERS) erhobenen Anfrageplan der größten Stufe; die Pläne der drei kleineren Stufen sind strukturell identisch und unterscheiden sich nur in den Zeilenzahlen. Der Ausführungsplan bleibt somit über alle Datengrößen unverändert: ein Seq Scan gefolgt von einem HashAggregate. Die Gruppierung erfolgt dabei stets in einem einzigen Batch (Batches: 1); selbst bei einer Million Zeilen verbleibt die Hash-Tabelle mit rund 1,2 MB (Memory Usage: 1169kB) vollständig im Arbeitsspeicher und wird nicht auf die Festplatte ausgelagert. Der Optimierer wählt somit über vier Größenordnungen hinweg dieselbe, vorhersagbare Strategie. Die im Listing ausgewiesene Gesamtzeit liegt über dem Mittelwert aus Tabelle 5.7, da die zeilengenaue Instrumentierung von EXPLAIN ANALYZE einen eigenen Messaufwand hinzufügt.

5.3. Empirische Gegenüberstellung: GROUPING SETS vs. Einzelabfragen

Um den angeführten I/O-Vorteil zu prüfen, wurde der GROUPING-SETS-Ansatz gegen die naheliegende Alternative gemessen: eine separate Abfrage je Aggregationsebene. Als Grundlage diente der hochskalierte SMARD-Datensatz mit rund einer Million Tupeln (vgl. Abschnitt 5.2), ergänzt um eine Gruppendimension, die jede Erzeugungsart einer der beiden Kategorien „erneuerbar“ oder „konventionell“ zuordnet. Verglichen wurde eine einzelne GROUPING-SETS-Abfrage über drei Ebenen mit drei separaten Aggregatabfragen, die dieselben Ebenen erzeugen: die einzelne Erzeugungsart (Individuum, hier die fall_id), die übergeordnete Kategorie (Gruppe) sowie die Gesamtheit aller Erzeugungsarten (Gesamtpopulation). Die GROUPING-SETS-Variante berechnet alle drei Ebenen in einer einzigen Anweisung. Die Messungen entstanden vor der Zusammenfassung von Transitions- und Abschlussfunktion zum Aggregat hplm; die hier gezeigte zweistufige Schreibweise ist dazu rechnerisch identisch:

```

1 EXPLAIN (ANALYZE, BUFFERS)
2 WITH agg AS (
3     SELECT fall_id, gruppe,
4           berechne_hplm_vollstaendig(
5               hplm_akkumulieren(wert, tag, 125)) AS hplm
6     FROM smard_1m
7     GROUP BY GROUPING SETS ((fall_id, gruppe), (gruppe), ())
8 )
9 SELECT fall_id, gruppe, (hplm).* FROM agg;

```

Listing 5.2: Einzelne GROUPING-SETS-Abfrage über drei Ebenen.

Die Vergleichsvariante erzeugt dieselben drei Ebenen über drei separate Abfragen, deren Laufzeiten und gelesene Blöcke anschließend summiert werden:

```

1  -- Ebene 1: je Erzeugungsart und Gruppe (Individuum)
2  EXPLAIN (ANALYZE, BUFFERS)
3  WITH agg AS (
4      SELECT fall_id, gruppe,
5            berechne_hplm_vollstaendig(
6                hplm_akkumulieren(wert, tag, 125)) AS hplm
7      FROM smard_1m GROUP BY fall_id, gruppe)
8  SELECT fall_id, gruppe, (hplm).* FROM agg;
9
10 -- Ebene 2: je Gruppe (erneuerbar / konventionell)
11 EXPLAIN (ANALYZE, BUFFERS)
12 WITH agg AS (
13     SELECT gruppe,
14           berechne_hplm_vollstaendig(
15               hplm_akkumulieren(wert, tag, 125)) AS hplm
16     FROM smard_1m GROUP BY gruppe)
17 SELECT gruppe, (hplm).* FROM agg;
18
19 -- Ebene 3: Gesamtpopulation
20 EXPLAIN (ANALYZE, BUFFERS)
21 WITH agg AS (
22     SELECT berechne_hplm_vollstaendig(
23         hplm_akkumulieren(wert, tag, 125)) AS hplm
24     FROM smard_1m)
25 SELECT (hplm).* FROM agg;

```

Listing 5.3: Drei separate Aggregatabfragen für dieselben Ebenen.

Die von EXPLAIN (ANALYZE, BUFFERS) gelieferten Anfragepläne machen den strukturellen Unterschied beider Varianten sichtbar. Listing 5.4 zeigt den Plan der GROUPING-SETS-Abfrage, Listing 5.5 den Plan der ersten Einzelabfrage; die beiden übrigen Einzelabfragen erzeugen Pläne desselben Typs. Abbildung 5.1 stellt beide ergänzend in der grafischen Notation von pgAdmin dar. Die Listings sind auf die strukturtragen-

den Knoten gekürzt; entfernt wurden die Kostenschätzungen des Optimierers und die Puffer-Statistiken je Knoten, deren Block-Zahlen aggregiert in Tabelle 5.8 stehen.

```

1 CTE Scan on agg (actual time=659.065..6348.641 rows=4128)
2   CTE agg
3     -> GroupAggregate (actual time=659.053..6340.938 rows=4128)
4         Group Key: smard_1m.gruppe, smard_1m.fall_id
5         Group Key: smard_1m.gruppe
6         Group Key: ()
7     -> Sort (actual time=652.123..787.811 rows=1000500)
8         Sort Key: smard_1m.gruppe, smard_1m.fall_id
9         Sort Method: external merge  Disk: 38976kB
10    -> Seq Scan on smard_1m (rows=1000500)
11 Execution Time: 6360.346 ms

```

Listing 5.4: Anfrageplan der GROUPING-SETS-Abfrage (gekürzt).

```

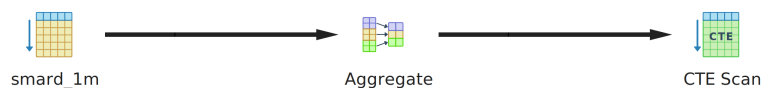
1 CTE Scan on agg (actual time=2276.018..2349.253 rows=4125)
2   CTE agg
3     -> HashAggregate (actual time=2276.013..2344.886 rows=4125)
4         Group Key: smard_1m.fall_id, smard_1m.gruppe
5         Batches: 1  Memory Usage: 1297kB
6     -> Seq Scan on smard_1m (rows=1000500)
7 Execution Time: 2349.632 ms

```

Listing 5.5: Anfrageplan der Einzelabfrage für die Ebene (fall_id, gruppe) (gekürzt).



(a) GROUPING-SETS-Abfrage: Seq Scan → Sort → Aggregate



(b) Einzelabfrage: Seq Scan → Aggregate (ohne Sortierung)

Abbildung 5.1.: Grafische Darstellung der Anfragepläne aus pgAdmin (Datenfluss von links nach rechts). Der GROUPING-SETS-Plan (a) enthält den zusätzlichen **Sort**-Knoten, der die externe Sortierung verursacht; der Plan der Einzelabfrage (b) aggregiert direkt über eine Hash-Tabelle.

Die Pläne belegen die Mechanik im Detail. In beiden Fällen beginnt die Auswertung mit einem **Seq Scan** über alle 1 000 500 Zeilen der Faktentabelle – der Unterschied liegt in der Aggregationsstrategie darüber. Der **GROUPING-SETS**-Plan enthält ein einziges **GroupAggregate** mit *drei* **Group Key**-Einträgen ((**gruppe**, **fall_id**), (**gruppe**) und ()): PostgreSQL berechnet alle drei Aggregationsebenen tatsächlich in einem einzigen Durchlauf über die Daten. Ein sortbasiertes **GroupAggregate** setzt jedoch sortierte Eingabedaten voraus, weshalb dem Aggregat ein **Sort**-Knoten über den gesamten Datenbestand vorgeschaltet ist. Dessen Zeile **Sort Method: external merge** **Disk: 38976kB** zeigt den Preis dieses Vorgehens: Die Sortierung von rund einer Million Zeilen übersteigt das verfügbare **work_mem** (4MB) um ein Vielfaches und wird als External-Merge-Sortierung temporär auf die Festplatte ausgelagert – daraus stammen die in Tabelle 5.8 ausgewiesenen temporären Sortierblöcke.

Der Plan der Einzelabfrage kommt dagegen ohne Sortierung aus: PostgreSQL wählt ein **HashAggregate**, das die Gruppen während des Scans in einer Hash-Tabelle im Arbeitsspeicher bildet. Die Zeile **Batches: 1** **Memory Usage: 1297kB** zeigt, dass diese Hash-Tabelle mit rund 1,3 MB vollständig im Speicher verbleibt und kein einziges Byte ausgelagert wird. Dafür muss jede der drei Einzelabfragen die Faktentabelle erneut vollständig lesen, sodass sich die Scan-Kosten verdreifachen. Genau dieses Wechselspiel – ein Scan mit teurer externer Sortierung gegenüber drei Scans mit jeweils speicherinterner Hash-Aggregation – erklärt, warum sich die beiden Varianten in der Gesamtlaufzeit trotz völlig unterschiedlicher Pläne nahezu ausgleichen (vgl. Tabelle 5.8).

Variante	Durchläufe	Faktentab. (Blöcke)	Sort (Blöcke)	Laufzeit (10 Läufe)
GROUPING SETS	1	12 839	9 737	6631,2 ± 120,1 ms
3 Einzelabfragen	3	38 517	0	6824,6 ± 173,6 ms

Tabelle 5.8.: Vergleich von **GROUPING SETS** und separaten Einzelabfragen auf rund einer Million Tupeln. Angegeben sind die gelesenen 8-kB-Blöcke der Faktentabelle, die temporär auf die Festplatte ausgelagerten Sortierblöcke sowie die Laufzeit als Mittelwert ± Standardabweichung über zehn Messläufe (laut **EXPLAIN (ANALYZE, BUFFERS)**). Die Block-Zahlen sind über alle Läufe konstant; die Laufzeit der Einzelabfragen ist die Summe der drei Teilabfragen.

Die Messung bestätigt den erwarteten Effekt nur teilweise. Wie vorhergesagt liest die **GROUPING-SETS**-Variante die Faktentabelle nur einmal (12 839 Blöcke) statt dreimal (38 517 Blöcke); der Aufwand für das wiederholte Lesen der Basisdaten sinkt damit auf rund ein Drittel. Bei der Laufzeit schlägt sich dieser I/O-Vorteil jedoch kaum nieder: Über zehn Messläufe liegen beide Varianten mit 6631,2 ± 120,1 ms (**GROUPING SETS**) gegenüber 6824,6 ± 173,6 ms (Summe der Einzelabfragen) praktisch gleichauf – der Unterschied liegt im Bereich der Messstreuung. Die Laufzeit jedes einzelnen der zehn Läufe ist in Anhang B aufgeführt. Der Grund ist, dass PostgreSQL eine Mehrebenen-Gruppierung über ein sortbasiertes **GroupAggregate** auswertet und dafür den gesamten

Datenbestand sortieren muss. Diese Sortierung übersteigt den Arbeitsspeicher und wird temporär als external-merge-Sortierung auf die Festplatte ausgelagert; die Spalte „Sort (Blöcke)“ in Tabelle 5.8 gibt die Zahl der dabei gelesenen temporären Blöcke an. Die drei Einzelabfragen nutzen demgegenüber jeweils ein speicherinternes `HashAggregate`, das ohne Festplatten-Sortierung auskommt. Der eingesparte I/O-Aufwand der GROUPING-SETS-Variante und ihr zusätzlicher Sortieraufwand heben sich somit weitgehend auf.

Der praktische Nutzen von GROUPING SETS liegt damit weniger in der reinen Ausführungszeit als in der Reduktion der Tabellendurchläufe und darin, die gesamte mehrstufige Auswertung als eine einzige, atomare Anweisung über einen konsistenten Datenbestand auszudrücken.

Naheliegender wäre die Vermutung, dass der I/O-Vorteil unmittelbar zutage träte, wenn die Sortierung im Arbeitsspeicher stattfinden könnte. Der Parameter `work_mem` legt fest, wie viel Speicher PostgreSQL einer einzelnen Sortier- oder Hash-Operation zugesteht, bevor auf die Festplatte ausgelagert wird. Diese Vermutung wurde daher experimentell geprüft: Dieselbe Messung wurde mit `work_mem` von 64 MB und von 256 MB wiederholt (je zehn Läufe; die Einzeldurchläufe sind in Tabelle B.2 in Anhang B dokumentiert).

Das erste Ergebnis überrascht: 64 MB genügen nicht, obwohl der Plan nur 38 MB Festplattenbedarf ausweist. Der Grund ist, dass diese Angabe die kompakte *externe* Darstellung der Sortierdaten beschreibt; im Arbeitsspeicher benötigt dieselbe Sortierung wegen des Verwaltungsaufwands je Zeile rund 71 MB. Erst bei 256 MB wechselt PostgreSQL auf eine speicherinterne Sortierung (`Sort Method: quicksort Memory: 71264kB`).

Das zweite Ergebnis ist das entscheidende: Auch mit speicherinterner Sortierung wird GROUPING SETS nicht schneller. Mit $5824,0 \pm 250,0$ ms liegt es weiterhin gleichauf mit den Einzelabfragen ($5757,9 \pm 152,0$ ms). Der Engpass dieser Auswertung ist also nicht die Ein-/Ausgabe, sondern die Rechenlast des Aggregats selbst: Bei drei Aggregationsebenen führen beide Varianten in Summe dieselbe Zahl an Aufrufen der Transitions-Funktion aus, denn jede der eine Million Zeilen fließt in drei Ebenen ein. Diese CPU-Kosten dominieren die Gesamtlaufzeit – unabhängig davon, wo sortiert wird. Der Vorteil des einmaligen Tabellendurchlaufs kommt folglich erst dann zum Tragen, wenn der wiederholte Scan selbst zum relevanten Kostenfaktor wird, wie der folgende Versuch mit wachsender Ebenenzahl zeigt.

Verhalten bei wachsender Ebenenzahl. Der bisherige Vergleich betrachtet drei Aggregationsebenen. Um zu untersuchen, wie sich das Verhältnis beider Ansätze bei *mehr* Gruppierungen entwickelt, wurde der Datensatz um zwei zusätzliche, künstliche Dimensionen (`region`, `block`) ergänzt und die Messung systematisch für zwei bis fünf Ebenen wiederholt. Die drei Dimensionen werden dabei deterministisch aus der `fall_id` abgeleitet: `gruppe` ordnet die ersten sechs Erzeugungsarten jedes Vervielfältigungsblocks

(`fall_id mod 100 ≤ 6`) der Kategorie „erneuerbar“ zu, alle übrigen der Kategorie „konventionell“; `region` alterniert über `fall_id mod 2` zwischen „Nord“ und „Sued“, und `block` verteilt die Fälle über `fall_id mod 3` auf drei Blöcke („B1“ bis „B3“). Die drei Dimensionen sind dabei rein künstliche Partitionslabels für den Benchmark: Da die Fallnummerierung alphabetisch über die Erzeugungsarten erfolgt, entspricht insbesondere die Kategorie „erneuerbar“ nicht der realen energiewirtschaftlichen Klassifikation (so fällt etwa Braunkohle in diese Kategorie, vgl. Tabelle 5.9). Für den Zweck des Benchmarks – die Messung des Laufzeitverhaltens bei wachsender Ebenenzahl – ist allein die Kardinalität der Gruppen relevant, nicht ihre inhaltliche Bedeutung. Tabelle 5.9 zeigt einen Auszug des so erweiterten Datensatzes.

<code>fall_id</code>	<code>tag</code>	<code>wert</code>	<code>gruppe</code>	<code>region</code>	<code>block</code>
101	124	3 621,99	erneuerbar	Sued	B3
101	125	3 655,90	erneuerbar	Sued	B3
101	126	3 730,80	erneuerbar	Sued	B3
102	124	7 661,40	erneuerbar	Nord	B1
102	125	7 666,57	erneuerbar	Nord	B1
102	126	7 682,95	erneuerbar	Nord	B1

Tabelle 5.9.: Auszug des um die Dimensionen `gruppe`, `region` und `block` erweiterten Benchmark-Datensatzes (`smard_1m`) rund um den Interventionszeitpunkt (Stunde 125). Die Dimensionswerte sind je Fall konstant, da sie deterministisch aus der `fall_id` abgeleitet werden.

Jede Variante wurde zehnmal ausgeführt; Tabelle 5.10 berichtet Mittelwert und Standardabweichung der Laufzeit.

Ebenen	GROUPING SETS	Einzelabfragen
2	3833,8 ± 117,4 ms	3677,2 ± 99,6 ms
3	5707,2 ± 27,1 ms	5649,5 ± 24,6 ms
4	8029,0 ± 287,4 ms	8126,0 ± 653,8 ms
5	9556,3 ± 179,0 ms	9895,9 ± 338,9 ms

Tabelle 5.10.: Laufzeit von GROUPING SETS und der Summe separater Einzelabfragen bei wachsender Zahl an Aggregationsebenen (Mittelwert ± Standardabweichung über zehn Läufe, rund eine Million Tupel). Die Messung erfolgte über wiederholte Ausführung mit `clock_timestamp()`; die zehn Einzeldurchläufe jeder Messreihe sind in Anhang C dokumentiert.

Das Ergebnis zeigt einen klaren Trend: Bei nur zwei Ebenen ist die Summe der Einzelabfragen noch rund 150 ms schneller, da der zusätzliche Sortieraufwand von GROUPING SETS hier den geringen I/O-Vorteil überwiegt. Bei drei Ebenen liegen beide Varianten noch nahezu gleichauf; ab vier Ebenen kehrt sich das Verhältnis um, und bei fünf Ebenen ist GROUPING SETS mit rund 9 556 ms gegenüber 9 896 ms etwa 340 ms schneller.

Mit jeder weiteren Ebene benötigt die Einzelabfragen-Variante einen zusätzlichen vollständigen Tabellendurchlauf, während GROUPING SETS die Faktentabelle unverändert nur einmal liest. Der Vorteil von GROUPING SETS wächst somit mit der Zahl der Aggregationsebenen, da sein Aufwand wesentlich vom einmaligen Tabellenscan bestimmt wird, während die separaten Abfragen linear mit der Ebenenzahl skalieren. Für Auswertungen über viele Ebenen ist GROUPING SETS damit nicht nur konzeptionell kompakter, sondern auch in der Laufzeit überlegen.

Die Anfragepläne erklären diesen Umschwung auch strukturell. Bei zwei und drei Ebenen erzeugt der Optimierer Pläne desselben Typs wie in Listing 5.4: ein sortbasiertes **GroupAggregate** über einer einzigen Sortierung. Das funktioniert, weil die Ebenen ineinander geschachtelt sind: Sind die Daten nach (**gruppe**, **fall_id**) sortiert, liegen automatisch auch alle Zeilen derselben **gruppe** hintereinander – eine einzige Sortierung bedient also alle drei Ebenen.

Die Ebenen (**region**) und (**block**) brechen diese Schachtelung: Zeilen derselben Region sind in der Sortierung nach (**gruppe**, **fall_id**) über den gesamten Datenbestand verstreut. Für solche Ebenen wechselt PostgreSQL zu einem **MixedAggregate** (Listing 5.6): Die sortierkompatiblen Ebenen werden weiterhin aus der Sortierung bedient (**Group Key**-Einträge), während die querliegenden Ebenen *gleichzeitig* über Hash-Tabellen im Arbeitsspeicher aggregiert werden (**Hash Key**-Einträge) – mit lediglich 40 kB Speicherbedarf, da **region** und **block** nur zwei bzw. drei Gruppen bilden.

Der Plan belegt damit den Mechanismus hinter diesem Umschwung: Auch bei fünf Ebenen bleibt es bei *einem* Tabellendurchlauf und *einer* Sortierung; jede zusätzliche, nicht sortierkompatible Ebene kostet nur eine kleine Hash-Tabelle, während die Einzelabfragen-Variante für jede Ebene einen weiteren vollständigen Scan ausführen muss. Die im Listing ausgewiesene Gesamtzeit ist nicht direkt mit Tabelle 5.10 vergleichbar, da die zeilengenaue Instrumentierung von EXPLAIN ANALYZE bei diesem Plantyp erheblichen eigenen Messaufwand hinzufügt.

```
1 CTE Scan on agg (actual time=2765.037..36446.932 rows=4133)
2   CTE agg
3     -> MixedAggregate (actual time=2764.814..36393.020 rows=4133)
4         Hash Key: smard_1m.block
5         Hash Key: smard_1m.region
6         Group Key: smard_1m.gruppe, smard_1m.fall_id
7         Group Key: smard_1m.gruppe
8         Group Key: ()
```

```

9      Batches: 1   Memory Usage: 40kB
10     -> Sort (actual time=2727.668..3935.454 rows=1000500)
11         Sort Key: smard_1m.gruppe, smard_1m.fall_id
12         Sort Method: external merge   Disk: 46848kB
13         -> Seq Scan on smard_1m (rows=1000500)
14 Execution Time: 36492.721 ms

```

Listing 5.6: Anfrageplan der GROUPING-SETS-Abfrage über fünf Ebenen (auf die strukturtragenden Knoten gekürzt): ein `MixedAggregate` kombiniert sortbasierte (`Group Key`) und hashbasierte (`Hash Key`) Aggregation in einem einzigen Durchlauf.

5.4. Alternative Kopplungsansätze

Die vorliegende Arbeit realisiert die HPLM-Berechnung über ein benutzerdefiniertes Aggregat in der nativen prozeduralen Sprache PL/pgSQL und gibt das Ergebnis über eine `SELECT`-Anweisung aus. Beide Festlegungen sind jedoch nicht alternativlos. Dieser Abschnitt untersucht daher zwei Gestaltungsdimensionen und grenzt sie gegen den gewählten Weg ab: Zum einen die *Form*, in der das berechnete Ergebnis innerhalb der Datenbank als SQL-Artefakt bereitgestellt wird (Abschnitt 5.4.1), zum anderen die *Sprache*, in der die statistische Logik ausgeführt wird (Abschnitt 5.4.2). Während die erste Dimension empirisch anhand von Laufzeit-, Speicher- und Wartbarkeitsmessungen untersucht wird, wird die zweite konzeptionell diskutiert und durch einen Laufzeitvergleich sowie eine Erprobung unter Linux ergänzt.

5.4.1. Realisierung als SQL-Artefakte: Trade-offs zwischen Laufzeit, Speicher und Wartbarkeit

Das in dieser Arbeit entwickelte HPLM-Aggregat liefert seine Kennwerte zunächst als flüchtiges Abfrageergebnis. Unter *SQL-Artefakten* werden hier persistente Datenbankobjekte verstanden, die die Analyselogik dauerhaft im Schema verankern: Sichten, materialisierte Sichten, Tabellen mit Constraints, Stored Procedures und Trigger. Während eine ad-hoc abgesetzte Abfrage nach ihrer Ausführung verschwindet, bleiben diese Objekte als benannte Einträge im Datenbankkatalog bestehen und können von jeder Anwendung und jedem Nutzer wiederverwendet werden. In der Praxis stellt sich jedoch die Frage, in welcher *Form* dieses Ergebnis innerhalb der Datenbank bereitgestellt und vorgehalten wird. PostgreSQL bietet hierfür mehrere SQL-Artefakte an, die alle dieselbe Kernabfrage kapseln (vgl. Abschnitt 4.4), sich aber in Laufzeit, Speicherbedarf und Wartbarkeit deutlich unterscheiden. Um diese Trade-offs messbar zu machen, wurden vier

Realisierungen implementiert und gegeneinander vermessen. Als Datengrundlage diene die dritte Skalierungsstufe des in Abschnitt 5.2 beschriebenen Benchmark-Datensatzes: Die Tabelle `smard_100k` entstand aus den stündlichen SMARD-Erzeugungsdaten (vgl. Tabelle 5.6), indem die ursprünglichen Erzeugungsarten durch Vervielfältigung der Fälle auf 418 Fälle mit insgesamt 101 384 Messpunkten hochskaliert wurden; die Messungen liefen auf demselben System wie der Skalierungs-Benchmark.

```

1  -- 1) Sicht: rechnet bei JEDER Abfrage neu, belegt keinen Speicher
2  CREATE VIEW hplm_sicht AS <HPLM-Abfrage>;
3
4  -- 2) Materialisierte Sicht: rechnet einmal, speichert das Ergebnis
5  CREATE MATERIALIZED VIEW hplm_matsicht AS <HPLM-Abfrage>;
6
7  -- 3) Ergebnistabelle (mit PRIMARY-KEY-Constraint) + Stored Procedure
8  CREATE TABLE hplm_ergebnis (fall_id INT PRIMARY KEY, ...);
9  CREATE PROCEDURE hplm_berechnen() LANGUAGE plpgsql AS $$
10 BEGIN
11     TRUNCATE hplm_ergebnis;
12     INSERT INTO hplm_ergebnis <HPLM-Abfrage>;
13 END $$;
14
15 -- 4) Trigger: aktualisiert die materialisierte Sicht nach jeder
16   Aenderung
17 CREATE TRIGGER trg_hplm_aktuell
18 AFTER INSERT OR UPDATE OR DELETE ON smard_100k
   FOR EACH STATEMENT EXECUTE FUNCTION hplm_matsicht_aktualisieren();

```

Listing 5.7: Die vier untersuchten SQL-Artefakte (Kernabfrage als `<HPLM-Abfrage>` abgekürzt).

Tabelle 5.11 fasst die Messungen zusammen. Alle Zeiten wurden über `clock_timestamp()` als Ausführungsdauer der jeweiligen Anweisung erhoben; jede Messung wurde zehnmal wiederholt.

Artefakt	Lesen (ms)	Aktualisierung (ms)	Speicher
Sicht (View)	210,1 ± 14,2	entfällt (stets aktuell)	0 Bytes
Materialisierte Sicht	0,067 ± 0,152	220,9 ± 13,1 (REFRESH)	80 kB
Ergebnistab. + Prozedur	0,060 ± 0,131	223,7 ± 17,9 (CALL)	120 kB
Trigger (+ Mat. Sicht)	0,067 ± 0,152	autom., +220,7 ± 12,7 je INSERT	80 kB

Tabelle 5.11.: Gemessene Trade-offs der vier SQL-Artefakte auf `smard_100k` (101 384 Messpunkte, 418 Fälle). Angegeben sind Mittelwert ± Standardabweichung über zehn Läufe (gemessen mit `clock_timestamp()`); die Einzeldurchläufe sind in Anhang C dokumentiert. Die erhöhte Streuung der beiden Lesezeiten im Sub-Millisekunden-Bereich geht allein auf den jeweils ersten Lauf zurück, der den einmaligen Planungsaufwand des Erstaufrufs enthält (Läufe 2–10: konstant $\approx 0,017$ ms).

Laufzeit. Die Sicht rechnet bei jeder Abfrage die vollständige HPLM-Aggregation neu; ihre Lesezeit von rund 210 ms liegt in derselben Größenordnung wie der Skalierungs-Benchmark bei gleicher Datengröße (196,2 ms, vgl. Tabelle 5.7); die verbleibende Differenz von rund 14 ms liegt im Bereich der Messstreuung beider Reihen und erklärt sich durch den Cache-Zustand der wiederholten Messung. Die Sicht fügt der Abfrage also keinen messbaren Mehraufwand hinzu – sie ist die HPLM-Abfrage in gekapselter Form.

Speicherbedarf. Die Sicht belegt keinen zusätzlichen Speicher, da sie nur die Abfragedefinition vorhält. Die persistenten Varianten kosten 80 kB (materialisierte Sicht) bzw. 120 kB (Ergebnistabelle mit Index). Entscheidend ist dabei, dass dieser Speicher mit der Zahl der *Fälle* wächst (eine Ergebniszeile je Fall), nicht mit der weit größeren Zahl der Messpunkte; bei 418 Fällen bleibt er daher marginal.

Wartbarkeit. Hier liegt der deutlichste Trade-off. Der Trigger hält das Ergebnis ohne manuelles Zutun stets aktuell, erkaufte dies aber mit rund 221 ms zusätzlicher Latenz pro Änderung an der Faktentabelle (vgl. Tabelle 5.11) – bei schreibintensiven Lasten ein erheblicher Aufschlag, da jede einzelne Datenmanipulation einen vollständigen REFRESH auslöst. Die Stored Procedure verzichtet auf diese Automatik und überlässt der Anwendung die Kontrolle, *wann* neu berechnet wird; das ist effizienter bei häufigen Schreibzugriffen, erfordert aber Disziplin, da das Ergebnis zwischen zwei Aufrufen veralten kann. Der PRIMARY-KEY-Constraint auf `fall_id` sichert dabei zu, dass je Fall genau eine Ergebniszeile existiert, und garantiert so die Korrektheit und deterministische Reproduzierbarkeit des materialisierten Ergebnisses.

Einordnung. Es gibt somit kein universell überlegenes Artefakt, sondern eine vom Lese-Schreib-Verhältnis abhängige Wahl: Bei überwiegend lesenden, selten geänderten Daten amortisieren materialisierte Sicht oder Trigger ihren Aufbauaufwand vielfach; bei häufigen Schreibzugriffen sind die stets aktuelle Sicht oder eine gezielt aufgerufene Prozedur überlegen. Die in dieser Arbeit für die Validierung gewählte direkte SELECT-Form entspricht dabei funktional der Sicht.

5.4.2. Externe prozedurale Spracherweiterungen: PL/R und PL/Python

PostgreSQL erlaubt die Einbettung weiterer prozeduraler Sprachen direkt in den Datenbankserver (engl. *procedural languages*). Neben dem mitgelieferten PL/Python (Erweiterung `plpython3u`) existiert mit PL/R eine externe Erweiterung, die einen vollständigen R-Interpreter in die Datenbank einbindet. Im Unterschied zum nativen PL/pgSQL setzen

beide eine eigenständige, außerhalb der Datenbank installierte Laufzeitumgebung voraus – sie werden daher im Folgenden als *externe Spracherweiterungen* bezeichnet. Beide Ansätze verfolgen dasselbe Grundziel wie diese Arbeit – die Berechnung am Ort der Datenhaltung – gehen es aber anders an: Statt die Statistik in SQL nachzubilden, rufen sie eine etablierte Statistiksprache *innerhalb* des Datenbankprozesses auf.

Um diese Variante konkret umzusetzen, zeigt Listing 5.8 eine prototypische Implementierung der Phase-A-Regression in PL/Python. Der Zugriff auf die Tabellendaten erfolgt über das interne Modul `plpy`; Intercept und Trend werden nach demselben Verfahren wie im nativen SQL-Aggregat berechnet (vgl. Abschnitt 4.3).

```
1 CREATE OR REPLACE FUNCTION hplm_phase_a_python(fall INT)
2 RETURNS TABLE(intercept FLOAT8, trend FLOAT8)
3 LANGUAGE plpython3u AS $$
4     rows = plpy.execute(
5         "SELECT tag, wert FROM smard_100k "
6         "WHERE fall_id = %d AND tag < 125 ORDER BY tag" % fall
7     )
8     n = len(rows)
9     if n < 3: return []
10    xs = [r["tag"] - 1 for r in rows]
11    ys = [float(r["wert"]) for r in rows]
12    mx, my = sum(xs) / n, sum(ys) / n
13    sxx = sum((x - mx) ** 2 for x in xs)
14    sxy = sum((x - mx) * (y - my) for x, y in zip(xs, ys))
15    b1 = sxy / sxx
16    b0 = my - b1 * mx
17    return [(b0, b1)]
18 $$;
```

Listing 5.8: Prototypische OLS-Regression der Phase A über PL/Python.

Diese Funktion wurde auf dem Testsystem (Windows 11, PostgreSQL 18, EDB-Distribution) lauffähig in Betrieb genommen und gegen die native SQL-Implementierung validiert: Für Fall 101 liefern beide Varianten identische Phase-A-Koeffizienten (Intercept 3 639,3985, Trend 1,3614; die PL/Python-Variante gibt dabei die ungerundete Gleitkommadarstellung zurück). Die Variante über eine externe Spracherweiterung ist damit funktionsfähig und korrekt; der für ihre Inbetriebnahme unter Windows erforderliche Konfigurationsaufwand wird am Ende dieses Abschnitts dokumentiert.

Ein theoretisch noch mächtigerer Ansatz ergäbe sich über PL/R (Listing 5.9), da sich hier direkt die R-interne Funktion `lm()` oder sogar das `scan`-Paket aufrufen ließe. PL/R ist jedoch eine von der PostgreSQL-Distribution unabhängige Drittanbieter-Erweiterung [18] und war in der eingesetzten EDB-Installation nicht enthalten. Das folgende Listing wurde daher – wie am Ende dieses Abschnitts dokumentiert – in einer Linux-Umgebung in Betrieb genommen und validiert.

```

1 CREATE OR REPLACE FUNCTION hplm_phase_a_r(fall INT)
2 RETURNS TABLE(intercept FLOAT8, trend FLOAT8)
3 LANGUAGE plr AS $$
4     sql <- sprintf("SELECT (tag - 1) AS x, wert AS y FROM smard_100k
5                     WHERE fall_id = %d AND tag < 125", fall)
6     daten <- pg.spi.exec(sql)
7     if (nrow(daten) < 3) return(NULL)
8     modell <- lm(y ~ x, data = daten)
9     k <- coef(modell)
10    return(data.frame(intercept = k[1], trend = k[2]))
11 $$;

```

Listing 5.9: Konzeptioneller Entwurf: direkter Aufruf der linearen Modellanpassung über PL/R.

Der Vorteil dieser Ansätze liegt auf der Hand: Über PL/R ließe sich `scan::hplm()` unmittelbar nutzen, ohne die Regressionsmathematik manuell in SQL zu reimplementieren. Das in dieser Arbeit diskutierte REML-Verfahren mit Partial Pooling (vgl. Abschnitt 6.2) stünde damit ohne Eigenimplementierung zur Verfügung. Diesem Vorteil stehen jedoch gewichtige Nachteile gegenüber, die – wie der am Ende dieses Abschnitts dokumentierte Befund empirisch zeigt – die Wahl von PL/pgSQL in dieser Arbeit begründen:

- **Sicherheit:** PL/Python ist ausschließlich als *untrusted* Sprache verfügbar (gekennzeichnet durch das u in `plpython3u`). Wer entsprechende Prozeduren anlegen darf, erhält damit vollen Zugriff auf das Hostsystem mit den Rechten des Datenbankprozesses. Auch PL/R gilt als *untrusted*. Die in dieser Arbeit verwendete Sprache PL/pgSQL ist hingegen eine *trusted* Sprache und in PostgreSQL standardmäßig installiert.
- **Abhängigkeiten und Portabilität:** PL/R und PL/Python setzen eine vollständige, korrekt konfigurierte R- bzw. Python-Laufzeitumgebung auf dem Datenbankserver voraus. Auf Windows-Systemen muss die Datenbank-Engine zudem exakt gegen die DLL-Bibliotheken der installierten Sprachversion abgestimmt sein; ist diese Abstimmung fehlerhaft, schlägt die Ausführung fehl oder bringt den Serverprozess zum Absturz (vgl. den nachfolgenden Erfahrungsbericht). Die PL/pgSQL-Lösung kommt ohne jede externe Laufzeitabhängigkeit aus und ist auf jeder Standard-PostgreSQL-Installation lauffähig.
- **Integration in den Auswertungsplan:** Da PL/pgSQL-Aggregate nahtlos in reguläre SQL-Konstrukte wie `GROUPING SETS` eingebettet werden können, fügt sich die Berechnung direkt in den mengenorientierten Verarbeitungsfluss der Datenbank ein. Ein eingebetteter R- oder Python-Aufruf bleibt demgegenüber ein in sich geschlossener Fremdkörper, dessen Datenübergabe gesondert organisiert werden muss.

Praktische Erfahrung: Inbetriebnahme unter Windows. Der Weg zur lauffähigen PL/Python-Umgebung illustriert exemplarisch den Konfigurationsaufwand, den eine externe Spracherweiterung unter Windows verlangt. Die EDB-Distribution liefert den Python-Interpreter nicht mit; die Installationshinweise nennen Python 3.13.12 als getestete Version, die separat zu installieren und für den Serverprozess auffindbar zu machen ist [19]. Nach der Installation dieser Version schlug die Interpreter-Initialisierung zunächst dennoch reproduzierbar fehl – der Serverprozess wurde beim ersten Aufruf einer PL/Python-Funktion beendet, im Serverlog dokumentiert als `Fatal Python error: Failed to import encodings module`. Als Ursache erwies sich keine Fehlkonfiguration der Suchpfade, sondern ein Zugriffsrechteproblem: Der PostgreSQL-Dienst läuft unter einem eigenen Dienstkonto, das auf die im Benutzerprofil liegende Python-Standardbibliothek keinen Lesezugriff hat. Erst nachdem die Python-Installation an einen für das Dienstkonto lesbaren Ort (`C:\Python313`) kopiert und die Pfadkonfiguration entsprechend angepasst wurde, ließ sich der Interpreter initialisieren. Der Befund zeigt: Die Lauffähigkeit externer Spracherweiterungen hängt von einer extern bereitzustellenden, für den Dienstprozess korrekt zugänglichen Laufzeitumgebung ab. Unter Linux entfällt dieser Aufwand, da Interpreter und Erweiterung konsistent über die Paketverwaltung installiert werden (etwa `apt install postgresql-plpython3-18`); bei der abhängigkeitsfreien PL/pgSQL-Lösung stellt sich die Frage konstruktionsbedingt gar nicht.

Laufzeitvergleich. Um die Integrations-Frage auch quantitativ zu beantworten, wurden beide Varianten auf `smard_100k` (418 Fälle, 101 384 Messpunkte) gegeneinander gemessen: das native SQL-Aggregat mit `GROUP BY fall_id` gegen den Aufruf von `hplm_phase_a_python` je Fall (über `LATERAL`), jeweils zehn Läufe mit `clock_timestamp()` (Einzeldurchläufe in Anhang C). Das Ergebnis fällt eindeutig aus: Das SQL-Aggregat benötigt $306,1 \pm 19,3$ ms, die PL/Python-Variante $1970,3 \pm 118,1$ ms – rund das 6,4-Fache –, obwohl das Aggregat sogar den größeren Funktionsumfang berechnet (beide Phasen einschließlich R^2 , Konfidenzintervallen und p -Werten, während die Python-Funktion nur Intercept und Trend der Phase A schätzt). Die Ursache liegt im Verarbeitungsmodell: Das Aggregat läuft in einem einzigen Tabellendurchlauf im mengenorientierten Ausführungsplan mit, während die PL/Python-Funktion für jeden der 418 Fälle eine eigene Abfrage über `plpy.execute` absetzt und die Ergebniszeilen einzeln zwischen SQL-Engine und Python-Interpreter übergeben werden. Der im dritten Aufzählungspunkt benannte „Fremdkörper“-Charakter schlägt sich damit unmittelbar in der Laufzeit nieder: Für die mengenorientierte Auswertung vieler Fälle ist die native Aggregat-Lösung der Kopplung über eine externe Spracherweiterung auch empirisch deutlich überlegen.

Erprobung von PL/R unter Linux. Um den PL/R-Entwurf aus Listing 5.9 dennoch praktisch zu verifizieren, wurde ergänzend eine Linux-Umgebung herangezogen: ein Docker-Container mit PostgreSQL 16, in dem sich PL/R samt R-Laufzeitumgebung über einen einzigen Paketbefehl installieren ließ (`apt-get install postgresql-16-plr`).

Nach dem Anlegen der Erweiterung (`CREATE EXTENSION plr`) wurde die Funktion `hplm_phase_ar` unverändert übernommen und auf den Phase-A-Daten des Falls 101 ausgeführt. Das Ergebnis bestätigt die Korrektheit des Entwurfs: Mit einem Intercept von 3 639,3985 und einem Trend von 1,3614 liefert die über `lm()` gerechnete R-Variante exakt dieselben Koeffizienten wie das native SQL-Aggregat und die PL/Python-Funktion. Alle drei Kopplungsvarianten sind damit wechselseitig validiert. Zugleich bestätigt der Versuch den Portabilitätsbefund dieses Abschnitts empirisch: Was unter Windows an fehlenden Binärpaketen scheitert, ist unter Linux mit einem einzigen Paketbefehl betriebsbereit.

6. Zusammenfassung und Ausblick

Die vorliegende Arbeit hat untersucht, inwieweit sich das Hierarchical Piecewise Linear Model vollständig in ein relationales Datenbanksystem überführen lässt. Dieses abschließende Kapitel bündelt die Ergebnisse: Abschnitt 6.1 fasst die zentralen Befunde entlang der drei Forschungsfragen aus Kapitel 1 zusammen, Abschnitt 6.2 ordnet die methodischen Grenzen des gewählten Ansatzes ein, und der Ausblick benennt Anknüpfungspunkte für weiterführende Arbeiten.

6.1. Zusammenfassung der Ergebnisse

Ausgangspunkt dieser Arbeit war die Frage, ob sich die statistische Auswertung von Interventionsstudien mittels HPLM ohne Export der Daten in externe Statistiksoftware direkt im Datenbanksystem durchführen lässt. Die drei in Kapitel 1 formulierten Forschungsfragen lassen sich nach den Ergebnissen der Kapitel 3 bis 5 wie folgt beantworten.

Die erste Forschungsfrage nach der **Umsetzbarkeit** kann uneingeschränkt bejaht werden. Die mathematische Logik des HPLM wurde vollständig in native SQL- und PL/pgSQL-Operationen überführt: Ein benutzerdefiniertes Aggregat (`hplm`) akkumuliert die benötigten Quadratsummen beider Phasen in einem einzigen Tabellendurchlauf, eine Abschlussfunktion (`berechne_hplm_vollstaendig`) leitet daraus alle Kennwerte ab – Intercept, Trend, Level und Slope ebenso wie Bestimmtheitsmaße, Standardfehler und Konfidenzintervalle. Auch die Signifikanzbewertung gelingt datenbankintern: Da PostgreSQL keine Verteilungsfunktion der t -Verteilung bereitstellt, wurde der p -Wert über die Pólya-Näherung mit Li-De-Moor-Korrektur approximiert und vollständig in PL/pgSQL implementiert. Die Lösung kommt dabei ohne externe Bibliotheken oder untrusted-Spracherweiterungen aus.

Die zweite Forschungsfrage nach **Abstraktion und Skalierbarkeit** beantwortet die entwickelte Architektur auf zwei Ebenen. Die Datensatzunabhängigkeit wird durch das Zusammenspiel von generischer Aggregat-Schnittstelle und datengetriebenem Interventionszeitpunkt erreicht: Der Phasenwechsel ist nicht im Code festgeschrieben, sondern wird zur Laufzeit aus einer Konfigurationsrelation gelesen, und dieselbe Aggregatfunktion ließ sich unverändert sowohl auf die Reaktionszeitdaten des Hauptdatensatzes als auch

auf die SMARD-Energiedaten anwenden. Die Mehrebenenstruktur des HPLM wird über **GROUPING SETS** abgebildet, wodurch alle Aggregationsebenen in einer einzigen Anweisung über einen konsistenten Datenbestand berechnet werden. Der Skalierungs-Benchmark bestätigte das erwartete lineare Laufzeitverhalten: Über vier Größenordnungen hinweg blieb die Bearbeitungszeit je Zeile nahezu konstant bei rund $2\mu\text{s}$, und eine Million Messpunkte wurden in knapp zwei Sekunden vollständig datenbankintern ausgewertet. Die empirische Untersuchung der Auswertungsstrategien zeigte darüber hinaus ein differenziertes Bild: Bei wenigen Aggregationsebenen liegen **GROUPING SETS** und separate Einzelabfragen gleichauf, da der Engpass nicht die Ein-/Ausgabe, sondern die Rechenlast des Aggregats ist; mit wachsender Ebenenzahl kehrt sich das Verhältnis jedoch zugunsten von **GROUPING SETS** um, da der Optimierer querliegende Ebenen über ein **MixedAggregate** nahezu kostenfrei mitberechnet, während jede Einzelabfrage einen weiteren vollständigen Tabellendurchlauf erfordert. Der Vergleich der SQL-Artefakte ergab schließlich, dass die Wahl der Bereitstellungsform vom Lese-Schreib-Verhältnis abhängt: Persistente Artefakte lesen um mehr als drei Größenordnungen schneller als die stets aktuelle Sicht, verlagern den Aufwand aber in Aufbau- bzw. Aktualisierungsschritte.

Die dritte Forschungsfrage nach der **Validität** wurde durch den systematischen Vergleich mit der Referenzimplementierung **scan** beantwortet. Auf der Individualebene stimmen die Ergebnisse exakt überein: Für den unabhängigen, unbalancierten Datensatz **exampleAB** reproduziert die SQL-Implementierung alle vier Kennwerte – Intercept, Trend, Level und Slope – aller drei Fälle bis auf die dritte Nachkommastelle im W-Modell nach Huitema und McKean; die übrigen Kodierungsvarianten lassen sich über geschlossene Transformationen aus den W-Kennwerten ableiten. Auf der Aggregatebene gilt dasselbe: Alle vier Fixed Effects stimmen bei balancierten Daten mit **scan::hplm()** überein, sobald dieselbe Kodierung zugrunde gelegt wird – die zunächst sichtbaren Differenzen bei Intercept und Level gehen allein darauf zurück, dass **scan** in der Voreinstellung im B&L-B-Modell rechnet. Der methodische Unterschied zwischen dem Complete Pooling der SQL-Implementierung und dem Partial Pooling des REML-Verfahrens wirkt sich bei balancierten Daten somit nicht auf die Punktschätzer aus, sondern auf Standardfehler und Varianzkomponenten. Diese Abgrenzung – und die daraus folgende Einordnung als skalierbare Alternative für ausreichend dichte Datensätze, nicht als vollständiger REML-Ersatz – wird im folgenden Abschnitt 6.2 vertieft. Ergänzend zeigte der Laufzeitvergleich mit der Kopplung über eine externe Spracherweiterung, dass die native Aggregat-Lösung einer PL/Python-Implementierung um rund den Faktor sechs überlegen ist.

Insgesamt belegt die Arbeit damit, dass eine vollständige, validierte und skalierbare In-Database-Umsetzung des HPLM mit ausschließlich nativen PostgreSQL-Mitteln möglich ist – und schließt damit die in Abschnitt 2.3 identifizierte Lücke zwischen allgemeinen In-Database-Statistik-Bibliotheken und den bislang rein externen HPLM-Implementierungen.

Bezogen auf die in der Aufgabenstellung benannten Ergebnisse liefert die Arbeit damit beides: Kapitel 3 und 4 bilden mit Schema, Hilfsfunktion, Aggregat und hierarchischem

Aufruf einen dokumentierten Implementierungsleitfaden, der die Umsetzung auf beliebigen PostgreSQL-Installationen reproduzierbar macht; die empirischen Gegenüberstellungen in den Abschnitten 5.3 und 5.4.1 münden in Design-Empfehlungen für den produktiven Einsatz – von der Wahl der Auswertungsstrategie in Abhängigkeit von der Ebenenzahl bis zur Wahl des SQL-Artefakts in Abhängigkeit vom Lese-Schreib-Verhältnis.

6.2. Methodische Limitationen

Trotz der mathematischen Validität und der Performance-Vorteile der vorgestellten Implementierung verbleiben methodische Limitationen, die für eine wissenschaftlich differenzierte Einordnung der Ergebnisse offengelegt werden müssen. Die wesentliche Einschränkung betrifft die statistische Abgrenzung des relationalen Gruppierungsansatzes gegenüber einem simultan geschätzten Mehrebenenmodell: Der folgende Abschnitt ordnet ein, welche Pooling-Strategie die SQL-Implementierung tatsächlich realisiert, wo sie sich vom REML-Verfahren der Referenzimplementierung unterscheidet und welche Konsequenzen dies für spärlich besetzte Datensätze hat.

6.2.1. Pooling-Charakter der relationalen Aggregation

In der Multilevel-Statistik werden drei Pooling-Strategien unterschieden [20]:

- **No Pooling:** Für jede Person wird eine eigenständige Regression berechnet, ohne dass Informationen zwischen Personen geteilt werden.
- **Complete Pooling:** Alle Beobachtungen werden in einer einzigen Regression zusammengefasst; individuelle Unterschiede werden ignoriert.
- **Partial Pooling:** Individuelle Schätzungen werden als *Random Effects* modelliert. Unsichere Schätzungen kleiner Stichproben werden über den sogenannten Shrinkage-Effekt zum Gruppenmittel hingezogen und dadurch stabilisiert.

Das in dieser Arbeit umgesetzte SQL-Aggregat führt durch die `GROUPING SETS`-Klausel auf der Individualebene ein *No Pooling* und auf den aggregierten Ebenen ein *Complete Pooling* durch. Ein echtes Partial Pooling, wie es `scan::hplm()` [12] über das iterative *Restricted Maximum Likelihood*-Verfahren (REML) realisiert, lässt sich nicht in geschlossener Form mit relationalen Standardaggregaten abbilden. Der Grund liegt in einem grundlegenden Gegensatz zwischen der Funktionsweise von REML und dem Verarbeitungsmodell von SQL-Aggregaten:

- **Einmaliger Durchlauf statt Iteration:** Der hier verwendete OLS-Schätzer besitzt eine geschlossene Lösung ($\beta_1 = S_{xy}/S_{xx}$) und benötigt daher nur einen einzigen Durchlauf durch die Daten. Ein SQL-Aggregat ist genau dafür ausgelegt: Es akkumuliert zeilenweise Summen in einem einzigen Tabellen-Scan und liefert anschließend ein Ergebnis. REML hingegen schätzt die Varianzkomponenten *iterativ*, indem es einen Optimierungsschritt so lange wiederholt, bis die Lösung konvergiert. Ein Standardaggregat kennt kein „Wiederhole, bis konvergiert“; es kann sich nicht selbst erneut über die Daten aufrufen.
- **Globale statt zeilenweiser Datensicht:** Ein SQL-Aggregat sieht in jedem Schritt nur eine einzelne Zeile und führt unabhängige Summen fort. REML benötigt dagegen in jeder Iteration die gesamte Datenstruktur gleichzeitig, um die Varianz-Kovarianz-Struktur zwischen den Personen zu schätzen.
- **Fehlende Matrixoperationen:** REML löst in jedem Schritt Gleichungssysteme und invertiert Kovarianzmatrizen. Standard-SQL stellt hierfür keine nativen Operationen bereit.

Zwar ließe sich REML mit prozeduralen PL/pgSQL-Schleifen und Hilfstabellen prinzipiell auch in der Datenbank nachbilden; dies würde jedoch das mengenorientierte Aggregat-Paradigma verlassen und damit gerade jene Skalierbarkeitsvorteile aufgeben, die den Kern des Ansatzes ausmachen. Die Beschränkung auf OLS ist somit keine willkürliche Vereinfachung, sondern die Folge der Entscheidung, die Auswertung vollständig im nativen SQL-Aggregatmodell zu halten.

Tabelle 6.1 fasst die methodischen Unterschiede zusammen:

Kriterium	SQL-Implementierung	scan::hplm()
Pooling-Strategie	No Pooling (individuell) + Complete Pooling (aggregiert)	Partial Pooling mit Shrinkage
Schätzverfahren	Geschlossene OLS-Lösung pro Partition	Iterativer REML-Algorithmus
Laufzeit	Linear $\mathcal{O}(N)$ in der Gesamtzeilenzahl N , ein einziger Tabellendurchlauf	Überlinear: kubische Matrixinversion je Iterationsschritt, mehrfach wiederholt bis zur Konvergenz
Spärliche Daten	Keine Schätzung bei $n_A < 3$ oder $n_B < 3$	Schätzbar durch Informationsaustausch (Shrinkage)

Tabelle 6.1.: Methodische Gegenüberstellung der SQL-Implementierung und der Referenzimplementierung `scan::hplm()`.

Zur Einordnung der Laufzeit ist eine klare Begriffsabgrenzung nötig: Die SQL-Implementierung skaliert linear mit $\mathcal{O}(N)$, wobei N die Gesamtzahl aller Messzeilen in der Faktentabelle bezeichnet. Da die benötigten Summen in einem einzigen Tabellendurchlauf akkumuliert werden, wächst der Aufwand proportional zur Datenmenge. Der iterativ arbeitende REML-Algorithmus in `scan::hplm()` invertiert dagegen in jedem Optimierungsschritt Kovarianzmatrizen; dieser Schritt skaliert überlinear (kubisch in der Anzahl der zu schätzenden Varianzkomponenten) und wird mehrfach wiederholt, bis das Verfahren konvergiert. Wichtig ist dabei die Klarstellung: Lineare Komplexität bedeutet nicht, dass der Aufwand konstant bleibt, sondern dass er *proportional* zur Zeilenzahl wächst – im Gegensatz zum überproportionalen Wachstum des REML-Verfahrens.

Den zweiten Punkt – die Stabilität bei spärlichen Daten – hat die Arbeit empirisch überprüft. Hierzu wurden vier Testpersonen mit unterschiedlich vielen Messpunkten in Phase A ($n_A \in \{2, 3, 4, 5\}$) durch die SQL-Implementierung ausgewertet. Tabelle 6.2 zeigt das Ergebnis.

n_A	Intercept	Trend	SE (Trend)	Ergebnis
2	—	—	—	kein Ergebnis (NULL)
3	100,50	0,50	0,8660	berechnet ($\nu = 1$)
4	100,30	0,80	0,4243	berechnet ($\nu = 2$)
5	100,60	0,50	0,3000	berechnet ($\nu = 3$)

Tabelle 6.2.: Verhalten der SQL-Implementierung bei abnehmender Zahl von Messpunkten in Phase A. Unterhalb von drei Punkten liefert die Funktion kein Ergebnis; der Standardfehler des Trends sinkt mit zunehmender Zahl der Messpunkte. Für $n_A \in \{3, 4\}$ werden die Koeffizienten zwar berechnet, ein p -Wert wird wegen $\nu < 3$ jedoch nicht ausgewiesen.

Der Test bestätigt eine klare Grenze: Da pro Phase zwei Parameter (Achsenabschnitt und Steigung) geschätzt werden, benötigt die isolierte OLS-Schätzung mindestens drei Messpunkte pro Phase ($n_A \geq 3$ bzw. $n_B \geq 3$), um überhaupt einen positiven Freiheitsgrad ($\nu = n - 2 \geq 1$) zu erhalten. Bei nur zwei Punkten gibt die Implementierung bewusst kein Ergebnis zurück (die Funktion fängt diesen Fall ab, bevor eine ungültige Berechnung entstehen kann). Bei genau drei Punkten ist eine Schätzung zwar möglich, beruht aber nur auf einem einzigen Freiheitsgrad und ist damit sehr unsicher, was sich am hohen Standardfehler (0,8660) ablesen lässt. Mit zunehmender Zahl der Messpunkte sinkt dieser Standardfehler erwartungsgemäß (0,4243 bei vier, 0,3000 bei fünf Messpunkten), die Schätzung wird also stabiler.

Genau an dieser Stelle liegt der konzeptionelle Vorteil des Partial Pooling von `scan`: Für Probanden mit sehr wenigen Messpunkten – etwa solche, die eine Studie früh abbrechen – kann das Shrinkage-Verfahren die unsichere Einzelschätzung durch Information aus der Gesamtgruppe stabilisieren. Der hier verfolgte No-Pooling-Ansatz auf der Individualebene besitzt diese Möglichkeit nicht.

Zwei weitere Einschränkungen betreffen die Inferenzstatistik. Erstens verwendet die Implementierung für das 95 %-Konfidenzintervall den festen Faktor 2,0 anstelle des exakten t -Quantils (bei $\nu = 8$ etwa 2,306); das ausgewiesene Intervall ist damit bei kleinen Stichproben geringfügig zu schmal und als bewusste Näherung zu verstehen. Zweitens werden Standardfehler und p -Werte nur für Intercept und Trend der Phase A berechnet; für die phänomenologisch interessantesten Effekte Level und Slope, die sich aus beiden Phasen-Regressionen zusammensetzen, weist die Implementierung keine Signifikanzmaße aus. Beide Punkte werden im Ausblick als Erweiterungen aufgegriffen.

Diese Einschränkung schmälert den Beitrag der Arbeit nicht, sondern verortet ihn präzise: Die vorgestellte SQL-Lösung ist eine auf Skalierbarkeit ausgelegte *architektonische Alternative* für ausreichend dichte Datensätze, jedoch kein vollständiger Ersatz für REML bei stark fragmentierten oder sehr kurzen Einzelfallreihen.

6.3. Ausblick

Aus den Ergebnissen und Limitationen dieser Arbeit ergeben sich mehrere Anknüpfungspunkte für weiterführende Arbeiten. Sie betreffen vier Bereiche: die statistische Modellierung, die zugrunde liegende Schätzstrategie, die technische Umsetzung im Datenbanksystem sowie die Anwendung auf reale Interventionsdaten. Die folgenden Abschnitte skizzieren diese Richtungen jeweils ausgehend von einem konkreten Befund der vorliegenden Arbeit.

Die inhaltlich naheliegendste Erweiterung betrifft die in Abschnitt 6.2 erörterte Pooling-Strategie. Die vorliegende Implementierung realisiert No Pooling auf der Individualebene und Complete Pooling auf den aggregierten Ebenen; ein echtes Partial Pooling mit Shrinkage bleibt dem REML-Verfahren der Referenzimplementierung vorbehalten. Eine künftige Arbeit könnte untersuchen, inwieweit sich zumindest eine Näherung datenbank-intern umsetzen lässt – etwa über ein zweistufiges Verfahren, das im ersten Durchlauf die Einzelschätzungen und deren Standardfehler berechnet und im zweiten Durchlauf präzisionsgewichtete Gruppenschätzer bildet. Ein solches Verfahren bliebe mit zwei Tabellendurchläufen im mengenorientierten Paradigma und könnte den Stabilitätsvorteil des Shrinkage für spärlich besetzte Fälle zumindest teilweise erschließen, ohne den iterativen REML-Algorithmus nachbilden zu müssen.

Ein zweiter Anknüpfungspunkt liegt in der statistischen Modellierung selbst. Die hier verwendete OLS-Schätzung setzt unkorrelierte Residuen voraus; Zeitreihen weisen jedoch häufig eine Autokorrelation erster Ordnung auf – benachbarte Messwerte hängen also systematisch voneinander ab, weil etwa ein guter Tag den nächsten beeinflusst. `scan` kann dies über ein optionales autoregressives Fehlermodell erster Ordnung (AR(1)) berücksichtigen, das genau diese Abhängigkeit zwischen aufeinanderfolgenden Residuen

mitschätzt. Da die dafür benötigte Größe – die Korrelation aufeinanderfolgender Residuen – aus laufenden Summen berechenbar ist, erscheint eine Erweiterung des Aggregatzustands um die entsprechenden Kreuzprodukte grundsätzlich möglich und würde die statistische Aussagekraft der datenbankinternen Inferenz weiter erhöhen. In dieselbe Richtung weist eine exaktere Berechnung des p -Werts: Die verwendete Pólya-Näherung mit Li-De-Moor-Korrektur könnte gegen eine datenbankinterne Implementierung der unvollständigen Betafunktion ausgetauscht werden, um auch bei sehr kleinen Freiheitsgraden exakte Werte zu liefern. Gleiches gilt für die Ausweitung der Inferenz auf die Effekte Level und Slope sowie die Verwendung exakter t -Quantile für die Konfidenzintervalle.

Auf der Systemseite betrifft die nächstliegende Erweiterung das Aggregat selbst: Durch Ergänzung einer Kombinier-Funktion (`COMBINEFUNC`), die zwei Teilzustände elementweise addiert, ließe sich die Auswertung von PostgreSQL parallelisieren – das dreistufige MADlib-Muster aus Transition, Merge und Final (vgl. Abschnitt 2.3) würde damit vervollständigt. Der Preis wäre der in Abschnitt 4.3 diskutierte strikte Determinismus: Die variable Summationsreihenfolge paralleler Prozesse kann minimale Gleitkomma-Abweichungen zwischen Läufen einführen. Darüber hinaus hat die Erprobung von PL/R unter Linux (Abschnitt 5.4.2) gezeigt, dass sich etablierte Statistikpakete grundsätzlich in den Datenbankserver einbetten lassen. Ein systematischer Folgeversuch könnte `scan::hplm()` selbst über PL/R aufrufen und damit REML-Schätzungen am Ort der Datenhaltung ermöglichen – um den Preis der in Abschnitt 5.4.2 diskutierten Sicherheits- und Portabilitätsnachteile sowie des dort gemessenen Laufzeitaufschlags der Kopplung über externe Spracherweiterungen. Der Vergleich beider Wege – native Näherung gegen eingebettete Referenzimplementierung – wäre eine lohnende Fortsetzung der hier begonnenen Gegenüberstellung.

Ebenfalls auf der Systemseite liegt die gezielte Steuerung des Anfrageoptimierers: Die Untersuchung in Abschnitt 5.3 hat gezeigt, dass die Wahl zwischen sortbasierter und hashbasierter Aggregation das Laufzeitverhalten maßgeblich prägt. Mit Erweiterungen wie `pg_hint_plan` [21] ließe sich diese Wahl über Hints explizit vorgeben und systematisch untersuchen, ob eine erzwungene Hash-Strategie für GROUPING SETS die beobachteten Sortierkosten vermeidet.

Schließlich eröffnet die Datenseite weiteres Potenzial. Der SMARD-Datensatz diene in dieser Arbeit als reiner Laufzeit-Benchmark mit künstlich gesetztem Interventionszeitpunkt; inhaltlich tragfähige Analysen würden reale Interventionsereignisse – etwa regulatorische Eingriffe in den Strommarkt oder die Inbetriebnahme großer Erzeugungskapazitäten – als Phasenübergänge nutzen. In Verbindung mit der gezeigten GROUPING-SETS-Architektur ließen sich solche Ereignisse simultan über Erzeugungsarten, Regionen und weitere Dimensionen auswerten. Damit würde der hier entwickelte Ansatz von einem validierten Werkzeug der Einzelfallanalyse zu einem allgemeinen Baustein für die datenbankinterne Interventionsanalyse auf großen, operativen Datenbeständen.

A. R-Skript zur Validierung

Das folgende R-Skript wurde zur unabhängigen Validierung der SQL-Implementierung (vgl. Abschnitt 5.1) verwendet. Es nutzt das etablierte Paket `scan` (Version 0.68.1) mit den Funktionen `plm()` für die einzelfallbezogene und `hplm()` für die hierarchische Auswertung. Die Reaktionszeiten der vier Probanden werden als `scdf`-Objekte (Single-Case Data Frame) definiert; der Interventionszeitpunkt liegt einheitlich bei Messzeitpunkt 11 (`B.start = 11`).

```
1 # -----
2 # HPLM-Analyse mit scan::hplm() (kompatibel mit scan 0.68.1)
3 # -----
4 library(scan)
5
6 # 1) Daten je Spieler als scdf-Objekt; Phase B beginnt bei
7   Messzeitpunkt 11
8 p1 <- scdf(
9   c(255, 248, 252, 245, 260, 250, 249, 253, 247, 251,
10     220, 215, 210, 205, 200, 198, 195, 190, 188, 185),
11   B.start = 11, name = "Player_1"
12 )
13 p2 <- scdf(
14   c(300, 295, 290, 292, 288, 285, 280, 278, 275, 270,
15     240, 235, 230, 225, 220, 215, 210, 205, 200, 195),
16   B.start = 11, name = "Player_2"
17 )
18 p3 <- scdf(
19   c(270, 265, 268, 260, 272, 258, 263, 267, 255, 261,
20     230, 225, 218, 212, 208, 203, 199, 194, 191, 187),
21   B.start = 11, name = "Player_3"
22 )
23 p4 <- scdf(
24   c(310, 308, 305, 307, 302, 299, 303, 298, 295, 292,
25     260, 254, 248, 243, 238, 233, 229, 224, 220, 216),
26   B.start = 11, name = "Player_4"
27 )
28 # Alle Faelle zu einem Mehrebenen-Datensatz zusammenfuehren
29 spieler <- c(p1, p2, p3, p4)
30
31 # 2) HPLM ueber alle vier Spieler (Restricted Maximum Likelihood)
32 hplm(spieler, method = "REML", random.slopes = FALSE)
33
```

```

34 # 3) Einzelfall-PLM je Spieler (Piecewise Linear Model)
35 plm(p1)
36 plm(p2)
37 plm(p3)
38 plm(p4)
39
40 # 4) HPLM je Spiel (Subgruppen ueber Index)
41 #   Player_1 + Player_3 = Starcraft ; Player_2 + Player_4 = CoD
42 hplm(spieler[c(1, 3)], method = "REML", random.slopes = FALSE) #
   Starcraft
43 hplm(spieler[c(2, 4)], method = "REML", random.slopes = FALSE) # CoD
44
45 # 5) HPLM je Geschlecht
46 #   Player_1 + Player_2 = maennlich ; Player_3 + Player_4 = weiblich
47 hplm(spieler[c(1, 2)], method = "REML", random.slopes = FALSE) #
   maennlich
48 hplm(spieler[c(3, 4)], method = "REML", random.slopes = FALSE) #
   weiblich
49
50 # 6) Zusatzvalidierung am eingebauten Datensatz exampleAB
51 plm(exampleAB$Johanna, model = "W")
52 plm(exampleAB$Karolina, model = "W")
53 plm(exampleAB$Anja, model = "W")
54 hplm(exampleAB, method = "REML", random.slopes = FALSE)

```

Listing A.1: R-Skript zur HPLM-Validierung mit `scan::hplm()` und `scan::plm()`.

Wie in Abschnitt 5.1.7 erläutert, verwendet `scan` in der Voreinstellung die Kodierung nach dem B&L-B-Modell, während die SQL-Implementierung dem W-Modell entspricht. Die modellinvarianten Größen Trend und Slope stimmen daher unabhängig von der Modellwahl überein; Intercept und Level weichen nur gegenüber der Voreinstellung ab und stimmen bei Auswertung mit `model = "W"` ebenfalls exakt überein.

B. Einzelmessungen des GROUPING-SETS-Vergleichs

Die folgende Tabelle listet die Laufzeit jedes einzelnen der zehn Messläufe aus Abschnitt 5.3 auf. Die Spalten B1, B2 und B3 bezeichnen die drei separaten Einzelabfragen (Ebene (`fall_id`, `gruppe`), Ebene (`gruppe`) und Gesamtpopulation); ihre Summe je Lauf entspricht der Vergleichsgröße der GROUPING-SETS-Variante. Die Blockzahlen (gelesene Faktentabellen- und Sortierblöcke) sind über alle Läufe konstant und daher hier nicht wiederholt.

Lauf	GROUPING SETS	B1	B2	B3	Summe B1–B3
1	6441,7	2431,9	2228,9	2018,1	6678,8
2	6733,4	2943,8	2227,2	2071,3	7242,4
3	6649,4	2395,7	2308,1	2074,6	6778,4
4	6722,1	2425,1	2263,2	2054,4	6742,7
5	6586,8	2469,2	2318,8	2047,0	6835,0
6	6568,1	2327,3	2296,1	2068,5	6691,9
7	6681,9	2557,1	2245,5	2099,9	6902,5
8	6646,6	2412,8	2262,3	2042,6	6717,8
9	6459,6	2400,0	2211,1	2091,9	6703,0
10	6822,5	2480,6	2269,1	2204,0	6953,6
∅	6631,2	2484,3	2263,0	2077,2	6824,6
Std	120,1	172,5	36,1	50,6	173,6

Tabelle B.1.: Laufzeit (in ms) jedes einzelnen Messlaufs des GROUPING-SETS-Vergleichs aus Abschnitt 5.3. B1, B2 und B3 sind die drei separaten Einzelabfragen; ihre Summe je Lauf ist die Vergleichsgröße zur GROUPING-SETS-Abfrage. Die letzten beiden Zeilen geben Mittelwert und Standardabweichung über die zehn Läufe an.

Tabelle B.2 dokumentiert die Einzeldurchläufe des `work_mem`-Experiments aus Abschnitt 5.3 (drei Ebenen, rund eine Million Tupel, Messung über `clock_timestamp()`).

Lauf	work_mem = 64 MB		work_mem = 256 MB	
	GS	EA	GS	EA
1	6363,2	5657,7	6131,7	5642,2
2	6160,6	5652,8	5927,4	5873,2
3	6107,2	5629,2	5645,7	5912,0
4	5993,3	5575,2	5662,1	5761,8
5	5929,0	5579,8	5627,3	6075,2
6	5856,4	5621,9	5691,2	5674,4
7	6025,1	5590,7	6062,6	5742,2
8	5781,5	5673,7	6253,1	5629,4
9	5657,4	5605,2	5714,1	5598,3
10	5660,9	5574,4	5524,9	5670,1
Ø	5953,5	5616,1	5824,0	5757,9
Std	224,6	36,7	250,0	152,0
Δ GS–EA	+6,0 %		+1,1 %	

Tabelle B.2.: Einzeldurchläufe des `work_mem`-Experiments aus Abschnitt 5.3 in Millisekunden (GS = GROUPING SETS, EA = Summe der drei Einzelabfragen). Die letzte Zeile gibt die prozentuale Differenz der Mittelwerte von GS gegenüber EA an: GROUPING SETS bleibt auch bei speicherinterner Sortierung (256 MB) geringfügig langsamer.

C. Einzeldurchläufe der Benchmarks

Tabelle C.1 dokumentiert die zehn Einzeldurchläufe jeder Messreihe aus Abschnitt 5.3 (Tabelle 5.10). Alle Angaben in Millisekunden; GS bezeichnet die GROUPING-SETS-Variante, EA die Summe der separaten Einzelabfragen je Durchlauf.

Lauf	2 Ebenen		3 Ebenen		4 Ebenen		5 Ebenen	
	GS	EA	GS	EA	GS	EA	GS	EA
1	4097,5	3641,0	5714,4	5675,3	7529,2	8321,6	9450,1	9819,3
2	3944,0	3647,7	5711,1	5613,8	7717,9	8511,4	9476,3	10807,3
3	3907,3	3638,8	5724,9	5700,2	7931,2	8051,0	9413,7	10034,8
4	3789,5	3644,8	5731,0	5646,5	7823,7	8746,5	9452,4	9749,5
5	3726,2	3655,1	5670,2	5650,0	7991,8	9528,6	9449,7	9732,2
6	3765,5	3629,4	5735,2	5648,7	8088,7	7747,5	9394,6	9891,8
7	3833,7	3709,6	5692,8	5651,4	8333,8	7576,3	9652,8	9763,7
8	3728,1	3637,3	5694,8	5650,3	8325,9	7605,2	9578,5	9606,9
9	3761,4	3616,2	5659,9	5621,8	8414,0	7557,2	9953,8	9789,9
10	3784,8	3951,8	5737,7	5636,6	8134,0	7614,3	9741,0	9763,4
∅	3833,8	3677,2	5707,2	5649,5	8029,0	8126,0	9556,3	9895,9
Std	117,4	99,6	27,1	24,6	287,4	653,8	179,0	338,9
Δ GS-EA	+4,3 %		+1,0 %		-1,2 %		-3,4 %	

Tabelle C.1.: Einzeldurchläufe des Ebenen-Benchmarks in Millisekunden (GS = GROUPING SETS, EA = Summe der Einzelabfragen). Die letzten Zeilen zeigen Mittelwert und Standardabweichung jeder Messreihe sowie die prozentuale Differenz der Mittelwerte von GS gegenüber EA (positive Werte: GS langsamer, negative Werte: GS schneller). Der Vorzeichenwechsel zwischen drei und vier Ebenen markiert den Punkt, ab dem GROUPING SETS die Einzelabfragen überholt.

Tabelle C.2 dokumentiert analog die zehn Einzeldurchläufe der Artefakt-Messungen aus Abschnitt 5.4.1 (alle Angaben in Millisekunden).

Lauf	Sicht	Mat. Sicht	Ergebnistab.	REFRESH	CALL	INSERT
1	211,6	0,498	0,433	220,9	225,9	209,7
2	242,0	0,028	0,025	217,2	223,4	213,8
3	200,9	0,020	0,019	209,1	223,8	200,5
4	214,3	0,018	0,017	255,6	233,8	228,1
5	210,6	0,017	0,017	223,0	267,1	227,8
6	203,7	0,017	0,017	213,0	224,0	241,2
7	201,2	0,017	0,017	218,6	219,1	234,8
8	193,6	0,017	0,017	220,5	207,0	222,7
9	224,0	0,017	0,017	210,0	205,4	220,0
10	199,4	0,017	0,017	221,5	207,6	208,3
Ø	210,1	0,067	0,060	220,9	223,7	220,7
Std	14,2	0,152	0,131	13,1	17,9	12,7

Tabelle C.2.: Einzeldurchläufe der Artefakt-Messungen in Millisekunden. Die letzten beiden Zeilen geben Mittelwert und Standardabweichung über die zehn Läufe an. Der erste Lauf der beiden Lesemessungen (Mat. Sicht, Ergebnistabelle) enthält den einmaligen Planungsaufwand des Erstaufrufs; die scheinbar hohe relative Streuung dieser beiden Spalten geht allein auf diesen ersten Lauf zurück (Läufe 2–10: konstant $\approx 0,017$ ms).

Tabelle C.3 dokumentiert die zehn Einzeldurchläufe des Skalierungs-Benchmarks aus Abschnitt 5.2 (alle Angaben in Millisekunden).

Lauf	Messpunkte			
	2 668	10 672	101 384	1 000 500
1	11,9	20,5	199,7	1965,8
2	6,4	20,2	195,7	2021,7
3	6,1	22,0	190,2	2012,9
4	5,4	25,3	195,6	1921,3
5	4,9	22,7	194,8	1876,9
6	4,9	30,6	199,7	1856,5
7	4,8	20,2	200,1	1898,5
8	7,0	20,0	191,7	1871,1
9	5,3	21,3	193,7	1877,8
10	5,0	20,1	200,4	1893,1
Ø	6,2	22,3	196,2	1919,6
Std	2,1	3,4	3,7	59,9

Tabelle C.3.: Einzeldurchläufe des Skalierungs-Benchmarks in Millisekunden, gruppiert nach der Zahl der Messpunkte. Der jeweils erste Lauf der kleinsten Stufe enthält den einmaligen Planungsaufwand des Erstaufrufs (11,9 ms gegenüber ≈ 5 ms in den Folgeläufen).

Tabelle C.4 dokumentiert die zehn Einzeldurchläufe des Laufzeitvergleichs zwischen nativem SQL-Aggregat und PL/Python aus Abschnitt 5.4.2 (alle Angaben in Millisekunden, smard_100k).

Lauf	SQL-Aggregat	PL/Python
1	333,4	2257,0
2	347,1	2024,4
3	308,0	2059,9
4	306,2	1956,9
5	302,1	1906,3
6	294,6	1878,6
7	294,5	1949,5
8	291,0	1881,8
9	290,6	1895,7
10	293,1	1893,2
Ø	306,1	1970,3
Std	19,3	118,1
Δ PL/Python–SQL	+543,7 % (Faktor 6,4)	

Tabelle C.4.: Einzeldurchläufe des Laufzeitvergleichs aus Abschnitt 5.4.2 in Millisekunden (SQL-Aggregat: vollständige HPLM-Auswertung über `GROUP BY fall_id`; PL/Python: Phase-A-Regression je Fall über `LATERAL`). Die letzte Zeile gibt den prozentualen Mehraufwand der PL/Python-Variante gegenüber dem SQL-Aggregat an.

Literaturverzeichnis

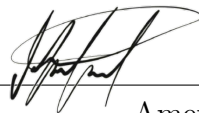
- [1] Thomas R. Kratochwill, John H. Hitchcock, Robert H. Horner, Joel R. Levin, Samuel L. Odom, David M. Rindskopf und William R. Shadish. „Single-Case Intervention Research Design Standards“. In: *Remedial and Special Education* 34.1 (2013), S. 26–38. DOI: [10.1177/0741932512452794](https://doi.org/10.1177/0741932512452794).
- [2] William R. Shadish, Thomas D. Cook und Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Boston, MA: Houghton Mifflin, 2002. ISBN: 978-0395615560.
- [3] Gene V. Glass, Victor L. Willson und John M. Gottman. *Design and analysis of time-series experiments*. Boulder: Colorado Associated University Press, 1975.
- [4] Mariola Moeyaert, John M. Ferron, S. Natasha Beretvas und Wim Van den Noortgate. „From a single-level analysis to a multilevel analysis of single-case experimental designs“. In: *Journal of School Psychology* 52.2 (2014), S. 191–211.
- [5] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein und Caleb Welton. „MAD Skills: New Analysis Practices for Big Data“. In: *Proceedings of the VLDB Endowment* 2.2 (2009), S. 1481–1492.
- [6] Richard E. Quandt. „The estimation of the parameters of a linear regression system obeying two separate regimes“. In: *Journal of the American Statistical Association* 53.284 (1958), S. 873–880.
- [7] Stephen W. Raudenbush und Anthony S. Bryk. *Hierarchical linear models: Applications and data analysis methods*. Thousand Oaks: Sage, 2002.
- [8] Holger Meyer und Florian Rose. „Datenbankanwendungsprogrammierung (DBAP)“. Vorlesungsskript, Universität Rostock, Institut für Informatik. 2025.
- [9] Andy Field. *Discovering Statistics Using IBM SPSS Statistics*. London: Sage, 2013.
- [10] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li und Arun Kumar. „The MADlib Analytics Library: Or MAD Skills, the SQL“. In: *Proceedings of the VLDB Endowment* 5.12 (2012), S. 1700–1711.
- [11] Xixuan Feng, Arun Kumar, Benjamin Recht und Christopher Ré. „Towards a Unified Architecture for in-RDBMS Analytics“. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. New York, NY: ACM, 2012, S. 325–336.

- [12] Jürgen Wilbert und Timo Lueke. *scan: Single-Case Data Analyses for Single and Multiple Baseline Designs*. R package, Version 0.68.1. 2024. URL: <https://cran.r-project.org/package=scan> (besucht am 12.05.2026).
- [13] George Pólya. „Remarks on Computing the Probability Integral in One and Two Dimensions“. In: *Proceedings of the First Berkeley Symposium on Mathematical Statistics and Probability*. Berkeley: University of California Press, 1949, S. 63–78.
- [14] Bin Li und Bart De Moor. „A Corrected Normal Approximation for the Student’s t Distribution“. In: *Computational Statistics & Data Analysis* 29.2 (1999), S. 213–216.
- [15] Paul A. Murtaugh. „Simplicity and Complexity in Ecological Data Analysis“. In: *Ecology* 88.1 (2007), S. 56–62.
- [16] Bradley E. Huitema und Joseph W. McKean. „Design Specification Issues in Time-Series Intervention Models“. In: *Educational and Psychological Measurement* 60.1 (2000), S. 38–58.
- [17] Bundesnetzagentur. *Realisierte Erzeugung [Datensatz]*. Kategorie: Stromerzeugung; Region: Deutschland; Auflösung: Stunde; Zeitraum: 6.–17. Juni 2026; Format: CSV. SMARD – Strommarktdaten. 2026. URL: <https://www.smard.de/home/downloadcenter/download-marktdaten/> (besucht am 17.06.2026).
- [18] Joseph E. Conway. *PL/R – R Procedural Language for PostgreSQL*. 2026. URL: <https://github.com/postgres-plr/plr> (besucht am 05.07.2026).
- [19] PostgreSQL Global Development Group / EDB. *PostgreSQL 18 Installation Notes: Procedural Languages*. Bestandteil der EDB-Windows-Distribution, lokal unter `doc/installation-notes.html`; nennt Python 3.13.12 als getestete Version. 2026. (Besucht am 05.07.2026).
- [20] Andrew Gelman und Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge: Cambridge University Press, 2007.
- [21] PG Hint Plan Development Group. *pg_hint_plan: Hint Plan Extension for PostgreSQL*. 2026. URL: https://github.com/oss-db/pg_hint_plan (besucht am 10.08.2026).

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Rostock, den 19.08.2026


Amer Badralden