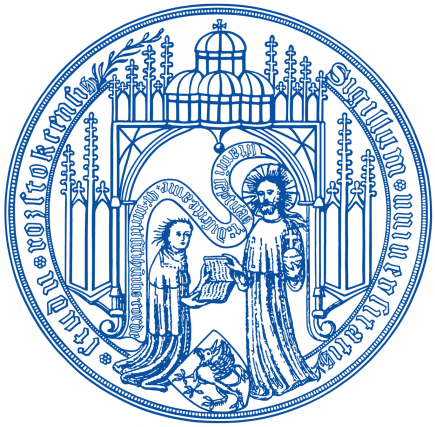
Entwicklung effizienter Zugriffsstrukturen auf großen Messdatenfiles

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von: Mhd Esmail Kanaan
Matrikelnummer: 218203288
geboren am: 01.05.1997 in Damaskus
Erstgutachter: Dr.-Ing. Hannes Grunert
Zweitgutachter: Dipl.-Inf. Ilvio Bruder
Betreuer: Dr.-Ing. Hannes Grunert
Dipl.-Inf. Ilvio Bruder
Prof. Dr. Gerd Baumgarten

Abgabedatum: 29. September 2025

Abstract

Die vorliegende Arbeit beschäftigt sich mit der Entwicklung effizienter Zugriffsstrukturen auf große Messdatenfiles am Beispiel von LIDAR-Daten, die im HDF5-Format vorliegen. Motiviert durch die stetig wachsenden Datenmengen am Leibniz-Institut für Atmosphärenphysik wurde untersucht, wie Abfragen auf Petabyte-großen Datensätzen beschleunigt und gleichzeitig die Speicherorganisation konsistent gehalten werden können. Zu diesem Zweck wurden verschiedene Indexstrukturen untersucht, miteinander verglichen und im Hinblick auf die Anforderungen dieser Arbeit bewertet. Anschließend wurde diskutiert, wie die ausgewählte Struktur in den bestehenden Datenworkflow integriert und für die effiziente Durchführung von Abfragen auf den Messdaten eingesetzt werden kann.
Die durchgeführten Experimente zeigen, dass externe Indexierung konsistent die besten Laufzeiten erzielt. Interne Indexierung bietet zwar Vorteile in Bezug auf die direkte Integration, verursacht jedoch deutlich höhere Laufzeiten aufgrund der Restriktionen des HDF5-Formats. Bei der Variante ohne Indexierung, wobei einfache Pythonskripte genutzt sind, bleiben die Zugriffszeiten wegen der linearen Suche stabil. Insgesamt verdeutlichen die Ergebnisse, dass eine Kombination aus HDF5 als primärem Datenspeicher und einer ergänzenden externen Indexierung den vielversprechendsten Ansatz darstellt, um auch künftig wachsende Datenmengen effizient verwalten und wissenschaftliche Abfragen performant unterstützen zu können.

Inhaltsverzeichnis

Abstract **3**

1 Einleitung **7**
1.1 Problemstellung und Zielsetzung 7
1.2 Aufbau der Arbeit 8

2 Grundlagen **9**
2.1 Das HDF5-Dateiformat 9
2.1.1 Beschreibung des Dateiformats HDF5 9
2.1.2 Aufbau von HDF5-Dateien 9
2.2 Beispiel einer HDF5-Datei aus dem Projektkontext 13
2.3 Vorteile des HDF5-Formats 13
2.4 Anwendungsfelder des HDF5-Formats 14
2.5 Typische Anfragen und Nutzungszenarien 14
2.6 Zugriffsstrukturen in Datenbanksystemen 16
2.6.1 Binärer Suchbaum 17
2.6.2 AVL-Baum 19
2.6.3 B-Baum 20
2.6.4 B^+-Baum 22
2.6.5 KD-Baum 24
2.6.6 KDB-Baum 25
2.6.7 Statisches Hashing 28
2.6.8 Lineares Hashing 29
2.6.9 Erweiterbares Hashing 31
2.6.10 Bitmap-Index 33
2.6.11 Grid-File 34

3 Konzept **39**
3.1 Zusammenfassende Bewertung der Strukturen 39
3.2 Begründung der Wahl des B^+-Baums als Indexstruktur 40
3.3 Art der Integration von Indexstrukturen 40
3.4 Externe Indexierung 41
3.4.1 Szenarien der externen Indexierung 41
3.4.2 Einsatz externer Datenbanksysteme 42
3.4.3 Externe Indexierung in SQLite 42
3.4.4 Wahl des Primärschlüssels 43
3.4.5 Struktur des reduzierten Datenbankschemas 43
3.4.6 Endgültiges Design der externen Indizes 43
3.5 Interne Indexierung 44

3.5.1 Struktur der reduzierten Datei . . . 45
3.5.2 Integration eines benutzerdefinierten Datentyps im HDF5-Dateiformat . . . 45
3.5.3 B^+-Baum als interne Indexstruktur innerhalb des HDF5-Dateiformats . . . 46
3.6 Ohne Indexierung . . . 47
3.7 Allgemeine Entwurfsentscheidungen . . . 48

4 Prototypische Implementierung 49
4.1 Entwicklungsumgebung . . . 49
4.2 Struktur der Implementierung . . . 50
4.3 Implementierungsdokumentation . . . 51
4.3.1 Allgemeine Zellen . . . 51
4.3.2 Besondere Zellen in der externen Indexierung . . . 53
4.3.3 Besondere Zellen in der internen Indexierung . . . 55
4.3.4 Besondere Zellen in der Variante ohne Indexierung . . . 57

5 Evaluation und Ergebnisse 61
5.1 Versuchsaufbau und Szenarien . . . 61
5.2 Messungen und Ergebnisse . . . 62
5.3 Herausforderungen bei der Integration dynamischer Datenstrukturen in HDF5 . . . 64
5.4 Diskussion . . . 64

6 Zusammenfassung und Ausblick 69
6.1 Schlussfolgerung . . . 69
6.2 Ausblick und zukünftige Arbeiten . . . 70

Literaturverzeichnis 71

A Implementierung 73
A.1 Nutzung der Jupyter-Notebooks . . . 73
A.2 Externe Indexierung . . . 74
A.2.1 Externe Indexierungsabfragen . . . 82
A.3 Interne Indexierung . . . 84
A.3.1 Interne Indexierungsabfragen . . . 93
A.4 Variante ohne Indexierung . . . 94
A.4.1 Abfragen der Variante ohne indexierung . . . 98

B Messungendurchläufe 101
B.1 Vergleich der Laufzeit der Rohdatenextraktion bei externer und interner Indexierung . . . 101
B.2 Vergleich der Erstellungszeiten: zusammengefasste HDF5-Datei vs externe SQLite-Datenbank 101
B.3 Vergleich der Erstellungszeiten der Indizes in SQLite und HDF5 . . . 102
B.4 Peak Top-10% – Vollständige Messdurchläufe . . . 102
B.5 Mean 0.1M..0.8M – Vollständige Messdurchläufe . . . 103
B.6 BG Bottom-10% – Vollständige Messdurchläufe . . . 103
B.7 Verwendete SQL-Abfragen . . . 104

C Hierarchische Organisation einer HDF5-Datei 109

Kapitel 1

Einleitung

Mit dem stetigen Wachstum der in Wissenschaft und Technik erzeugten Mess- und Forschungsdaten steigen auch die Anforderungen an deren effiziente Speicherung, Verarbeitung und Analyse. HDF5 bezeichnet ein **Hierarchical Data Format der Version 5**, das entwickelt wurde, um komplexe wissenschaftliche Datensätze langfristig, strukturiert und flexibel abzulegen. Dies betrifft insbesondere große, über längere Zeiträume aufgezeichnete Atmosphärendaten, wie sie etwa am Leibniz-Institut für Atmosphärenphysik erfasst werden, darunter Lidar-Messungen der Troposphäre, unteren Stratosphäre und Mesosphäre[1].
Trotz der vielseitigen Möglichkeiten, die HDF5 bietet, stellt insbesondere der schnelle und gezielte Zugriff auf große Datenmengen weiterhin eine Herausforderung dar. Mit dem exponentiellen Anstieg der Datenvolumina – laut Angaben von Mitarbeitenden des Leibniz-Instituts verdoppelt sich das gespeicherte Datenvolumen etwa alle viereinhalb Jahre – wächst der Bedarf an skalierbaren und leistungsfähigen Zugriffslösungen, die in der Lage sind, den steigenden Anforderungen moderner Datenanalyse-Workflows gerecht zu werden.

1.1 Problemstellung und Zielsetzung

Die zentrale Herausforderung besteht darin, spezifische und voneinander getrennte Informationen aus den vorliegenden Messdaten zu extrahieren, ohne dass der gesamte Forschungsdatenworkflow dadurch übermäßig verlangsamt wird. Zu diesem Zweck sind geeignete Zugriffs- und Indexstrukturen erforderlich, die präzise und schnelle Datenabfragen ermöglichen. Ziel dieser Arbeit ist es, relevante Informationen aus umfangreichen HDF5-Datensätzen effizient und zuverlässig zu gewinnen. In diesem Zusammenhang werden bestehende Methoden, Algorithmen und Zugriffsstrukturen systematisch analysiert und bewertet. Anschließend wird eine geeignete Struktur ausgewählt, weiterentwickelt und prototypisch implementiert, um eine performante und ressourcenschonende Datenabfrage innerhalb des gegebenen Forschungsdatenworkflows zu gewährleisten. Zur weiteren Präzisierung dieses Ziels werden im Rahmen dieser Arbeit die folgenden Forschungsfragen adressiert:

1. Welche Indexstrukturen stehen aktuell zur Verfügung, worin bestehen ihre Unterschiede, und welche eignet sich am besten für den vorliegenden Anwendungsfall?

2. Welche Varianten der Integration dieser Indexstrukturen gibt es, und welche erweist sich im gegebenen Kontext als die geeignetste?

3. Wie unterscheiden sich verschiedene Abfragetypen, wie beispielsweise `Top-N`, `Bottom-N` oder `Range Queries`, in Bezug auf ihre Effizienz in den unterschiedlichen Szenarien?

[1] Die in diesem Abschnitt dargestellten Informationen stammen aus einer persönlichen Mitteilung von Dipl.-Inf. Ilvio Bruder am 19.06.2025.

1.2 Aufbau der Arbeit

Die Arbeit gliedert sich in die folgenden Kapitel:

- **Kapitel 2: Grundlagen** — Darstellung des HDF5-Dateiformats mit Aufbau und typischen Anwendungsfällen, eine Übersicht zentraler Eigenschaften wissenschaftlicher Messdaten sowie eine Einführung in Zugriffsstrukturen von Datenbanksystemen.
- **Kapitel 3: Konzeption der Zugriffsstrukturen** — Vergleich und Bewertung verschiedener Strukturen, Auswahl und Begründung der am besten geeigneten Zugriffsstruktur sowie eine detaillierte Beschreibung der Integration dieser Struktur in den Datenworkflow.
- **Kapitel 4: Prototypische Implementierung** — Darstellung der prototypischen Umsetzung, beginnend mit der Entwicklungsumgebung und dem Aufbau der Implementierung. Anschließend folgt eine umfassende Dokumentation der Implementierung.
- **Kapitel 5: Evaluation und Ergebnisse** — Beschreibung des Versuchsaufbaus und der durchgeführten Messungen, gefolgt von einer detaillierten Analyse der Ergebnisse und einer Diskussion zentraler Beobachtungen und Herausforderungen.
- **Kapitel 6: Zusammenfassung und Fazit** — Abschließende Zusammenfassung und Ausblick auf mögliche zukünftige Arbeiten.

Kapitel 2

Grundlagen

Dieses Kapitel führt in die für die Arbeit relevanten Grundlagen ein. Dazu zählen eine Einführung in das HDF5-Dateiformat mit seinem Aufbau, typischen Nutzungsmöglichkeiten und Vorteilen sowie ein Überblick über grundlegende Zugriffsstrukturen in Datenbanksystemen. Behandelt werden die für diese Arbeit wesentlichen Anfragearten und Komplexitätsmaße im Zusammenhang mit den betrachteten Indexstrukturen.

2.1 Das HDF5-Dateiformat

Hinweis: Die Inhalte dieses Abschnitts orientieren sich überwiegend an der Darstellung in [The25d, The25e, The06].

2.1.1 Beschreibung des Dateiformats HDF5

HDF5 (Hierarchical Data Format Version 5) ist ein Dateiformat zur Speicherung großer und komplexer Datenmengen. Es umfasst eine Sammlung von Softwarekomponenten, darunter Programmbibliotheken, Sprachschnittstellen und Werkzeuge, die die Nutzung und Verarbeitung von HDF5-Daten unterstützen. Das Format wird von der HDF Group betreut und stellt die Weiterentwicklung des früheren HDF4-Standards dar.

2.1.2 Aufbau von HDF5-Dateien

In diesem Abschnitt wird die grundlegende Struktur von HDF5-Dateien beschrieben, die als zentrales Datenformat für die Speicherung der Messdaten im Rahmen dieser Arbeit verwendet wird. Ziel ist es, ein besseres Verständnis der internen Organisation dieser Dateien zu vermitteln. Im Fokus stehen dabei die wichtigsten strukturellen Elemente von HDF5: die Datei selbst (File), Gruppen (Groups), Datensätze (Datasets), Datentypen (Datatypes), Datenräume (Dataspaces), Eigenschaften (Properties) und Attribute.
Datei (File): bildet das oberste Element der hierarchischen Struktur und fungiert als Behälter für alle enthaltenen Datenobjekte wie Gruppen, Datensätze und Attribute u.ä. Die Dateiendung lautet in der Regel `.h5`. Jede Datei beginnt mit einer sogenannten Wurzelgruppe.
Gruppe (Group): ist das zentrale Strukturelement innerhalb einer HDF5-Datei, sie ist vergleichbar mit Ordnern in einem klassischen Dateisystem. Gruppen können sowohl weitere Untergruppen als auch Datensätze enthalten. Von der Wurzelgruppe (/) aus sind alle weiteren Strukturen organisiert. Über sogenannte Pfadnamen können Objekte eindeutig angesprochen werden, z.B. /Messdaten/Temperatur.
Eine typische HDF5-Datei kann mehrere Gruppen enthalten, die jeweils unterschiedliche Arten von Datenobjekten organisieren. Innerhalb einer Gruppe können beispielsweise Bilder, Tabellen oder Arrays

gespeichert sein. Gruppen können auch gemeinsam genutzte Objekte enthalten oder Verlinkungen zu Datenobjekten in anderen HDF5-Dateien aufweisen. Auf diese Weise ermöglicht die hierarchische Struktur von HDF5 eine flexible und modulare Organisation umfangreicher wissenschaftlicher Datensätze.

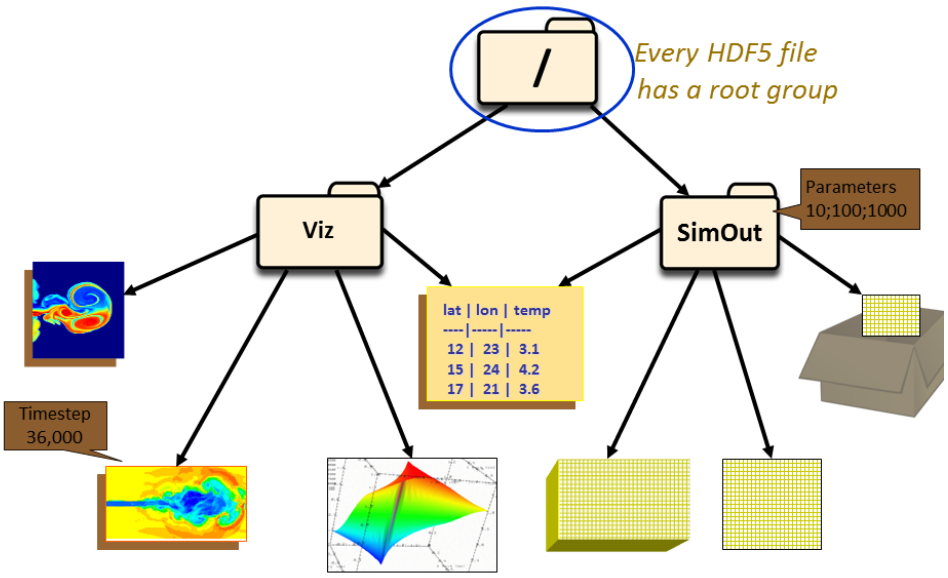


Abbildung 2.1: Beispiel einer HDF5-Gruppenstruktur [The25c]

In der oben dargestellten HDF5-Datei (siehe Abbildung 2.1) befinden sich zwei Hauptgruppen mit den Namen `Viz` und `SimOut`. Die Gruppe `Viz` enthält eine Vielzahl von Bilddaten sowie eine Tabelle, die ebenfalls in der Gruppe SimOut referenziert wird. Die Gruppe SimOut umfasst ein dreidimensionales Array, ein zweidimensionales Array sowie einen Link, der auf ein weiteres zweidimensionales Array in einer externen HDF5-Datei verweist. Objekte innerhalb einer HDF5-Datei werden üblicherweise über ihre vollständigen (absoluten) Pfadnamen identifiziert. So steht `/` für die Wurzelgruppe, `/Viz` für ein direktes Element der Wurzelgruppe mit dem Namen `Viz`, und `/Viz/Bild1` beispielsweise für ein Objekt innerhalb der Gruppe `Viz` [The06].

Datensatz (Dataset): in HDF5 dienen der strukturierten Speicherung von Rohdaten innerhalb einer Datei und sind funktional vergleichbar mit Dateien oder Tabellen in klassischen Datei- oder Datenbanksystemen. Ihre Struktur wird durch mehrere begleitende Objekte definiert – darunter Datentypen, Datenräume, Eigenschaftslisten sowie Attribute. Dabei umfassen Datasets nicht nur die eigentlichen Datenwerte, sondern auch sogenannte Metadaten, die für die Beschreibung und Interpretation der gespeicherten Inhalte unerlässlich sind [The06].

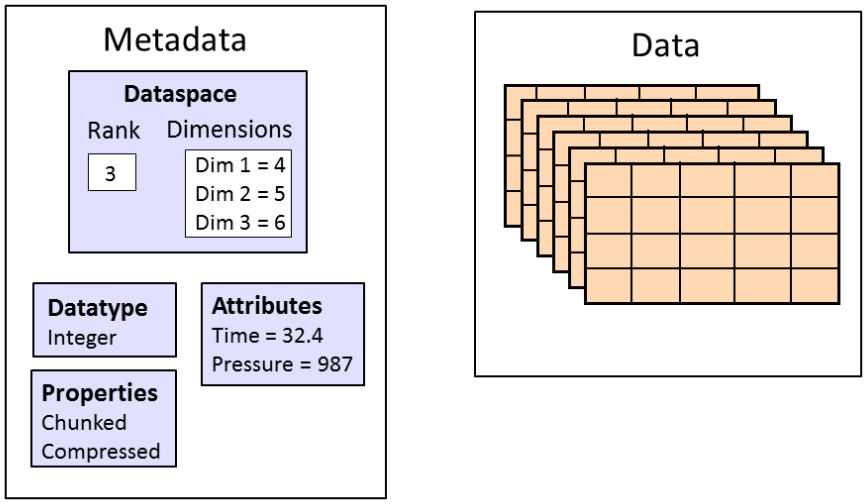


Abbildung 2.2: Schematische Darstellung eines HDF5-Datasets [The25a].

In der dargestellten Abbildung 2.2 wird ein dreidimensionaler Datensatz `(Rank 3)` mit vier Zeilen, fünf Spalten, und sechs Schichten verwendet. Die darin enthaltenen Werte liegen im Integer-Datentyp vor. Zusätzlich ist der Datensatz mit den Attributen Time und Pressure versehen. Es ist zudem sowohl gechunked (in kleine Blöcke unterteilt) als auch komprimiert (platzsparend gespeichert), was eine speichereffiziente und performante Datenverarbeitung ermöglicht.

Datenbereich (Dataspace): beschreibt die Struktur eines Datasets oder Attributs in einer HDF5-Datei. Er definiert die Dimensionen der Daten, also deren Anzahl sowie die Größe jeder einzelnen Dimension (z. B. 3×4 für drei Zeilen und vier Spalten). Bei der Erstellung eines Datasets oder Attributs ist die Angabe eines Datenbereichs erforderlich. Ein typisches Anwendungsszenario für einen Dataspace ergibt sich in der Umweltmesstechnik: Angenommen, eine Wetterstation zeichnet über einen Zeitraum von sieben Tagen stündlich die Temperatur auf. Die gesammelten Daten werden in einer 7×24-Tabelle gespeichert, wobei jede Zeile einem Tag und jede Spalte einer Stunde entspricht. Möchte man nun lediglich die Temperaturwerte der ersten beiden Tage analysieren – etwa um einen plötzlichen Kälteeinbruch zu Beginn des Beobachtungszeitraums zu untersuchen –, müssen nicht alle 168 Werte geladen werden. Stattdessen kann mithilfe geeigneter Programmierfunktionen im Dataspace gezielt nur der entsprechende Ausschnitt (die ersten beiden Zeilen mit jeweils 24 Werten) adressiert und geladen werden. Dies spart Rechenzeit und Arbeitsspeicher – ein wesentlicher Vorteil bei der Arbeit mit großen oder hochfrequenten Datensätzen.

Eigenschaften (Properties): dienen dazu, das Verhalten und die Struktur von Objekten wie Dateien, Datasets oder Attributen gezielt zu steuern. Für viele Anwendungsfälle stellt HDF5 standardisierte Eigenschaften bereit, die gängige Anforderungen abdecken. Diese Voreinstellungen lassen sich jedoch flexibel anpassen, um spezielle Funktionen zu realisieren oder die Leistung zu optimieren.

Datentyp (Datatype): beschreibt in HDF5 die Struktur und Interpretation der einzelnen Datenelemente innerhalb eines Datasets. Er definiert unter anderem den Datentyp (z. B. Integer, Float), die Bitbreite (z. B. 8, 16, 32, 64 Bit), das Vorzeichen (signed/unsigned) sowie die Byte-Reihenfolge (Endianness). In Tabelle 2.1 sieht man eine Liste von den verfügbaren Datentypen.

Datatype	**Beschreibung (einfach)**	**Beispiel**
short	Ganze Zahl (16 Bit)	-32 768, 32 767
unsigned short	Ganze Zahl (16 Bit, ohne Vorzeichen)	0, 65 535
int	Ganze Zahl (typisch 32 Bit)	-2 147 483 648, 2 147 483 647
unsigned int	Ganze Zahl (typisch 32 Bit, ohne Vorzeichen)	0, 4 294 967 295
long	Ganze Zahl (mind. bis zu 32 Bit, evtl. 64 Bit)	-2 147 483 648, 2 147 483 647
unsigned long	Ganze Zahl (ohne Vorzeichen)	0, 4 294 967 295
long long	Ganze Zahl mit erweiterter Genauigkeit (typisch 64 Bit)	-9 223 372 036 854 775 808
unsigned long long	Ganze Zahl ohne Vorzeichen (typisch 64 Bit)	0, 18 446 744 073 709 551 615
Float16	Gleitkommazahl (16 Bit)	6.10×10^{-5}
float	Gleitkommazahl (32 Bit) mit höhere Genauigkeit als Float16	1.18×10^{-38}
double	Gleitkommazahl (64 Bit) mit höhere Genauigkeit als float	2.23×10^{-308}
bitfield	Bitfeld zur Repräsentation von Flags oder Statuswerten	`00001111`
reference	Verweis auf ein anderes Objekt innerhalb der Datei	Referenz auf Dataset X
opaque	Binärdaten ohne bekannte Struktur	Binär-Log oder verschlüsselte Datei
array	Mehrere Werte gleichen Typs in strukturierter Form	{{1, 2}, {3, 4}}
variable length	Daten variabler Länge (Strings, Arrays)	`"Anna"`, `"Mohammad"`, `"Li"`
enumeration	Symbolische Konstanten	{OK=0, WARN=1, ERROR=2}
compound datatypes	Zusammengesetzte Datentypen (ähnlich struct in C)	{id: 1, temp: 20.5, flag: `'Y'`}

Tabelle 2.1: Übersicht wichtiger HDF5-Datentypen mit einfachen Beispielen [The25b].

Ein typisches Beispiel ist die Art der Datenspeicherung: Standardmäßig werden Datasets kontinuierlich im Speicher abgelegt. Bei Datenverarbeitung kann diese Eigenschaft geändert werden – etwa auf eine blockweise Speicherung (`chunked`) oder eine Kombination aus Chunking und Komprimierung. Durch den gezielten Einsatz von Properties lässt sich die Verarbeitung und Ablage von Daten an projektspezifische Anforderungen oder an die jeweilige Systemumgebung anpassen.

Attribute können optional mit HDF5-Objekten verknüpft werden, sie stellen eine Methode dar, um beschreibende Zusatzinformationen direkt an ein Datenobjekt wie ein Dataset oder eine Gruppe anzuhängen. Sie enthalten Metadaten, die Kontext zu den eigentlichen Daten liefern, jedoch kein eigenständiges Dataset darstellen. So lassen sich kleinere Informationsmengen effizient speichern, ohne dafür separate Datasets anlegen zu müssen. Ein typisches Anwendungsbeispiel wäre ein Datensatz mit Messergebnissen, beispielsweise einer Reihe von Temperaturwerten. Anstatt allgemeine Versuchsbedingungen wie Umgebungstemperatur oder Druck in eigenen Datasets zu speichern – was bei solch kleinen Informationsmengen unpraktisch und speicherineffizient wäre – können diese Werte als Attribute direkt dem Datensatz zugewiesen werden. Dabei werden sie in Form von eindeutigen Name/Wert-Paaren gespeichert, was ihre Strukturierung und spätere Identifikation erleichtert. Obwohl Attribute über einen eigenen Datentyp und einen zugehörigen Datenraum verfügen, unterscheiden sie sich in wesentlichen Punkten von klassischen Datasets. So unterstützen sie weder Komprimierung noch das teilweise Lesen oder Schreiben der Daten, lassen sich nicht gemeinsam mit anderen Objekten verwenden und können selbst keine weiteren Attribute besitzen. Ein weiterer zentraler Aspekt besteht darin, dass Attribute im sogenannten Objekt-Header des zugehörigen Objekts gespeichert werden. Dadurch sind sie unmittelbar an das jeweilige Objekt gebunden und existieren nicht eigenständig. Der Zugriff auf ein Attribut erfolgt daher stets über das Objekt, dem

es zugeordnet wurde. Insgesamt bieten Attribute in HDF5 eine kompakte und strukturierte Möglichkeit, kleine, aber bedeutende Metainformationen wie Einheiten, Zeitangaben oder geografische Informationen direkt und effizient mit den entsprechenden Datenobjekten zu verknüpfen.

2.2 Beispiel einer HDF5-Datei aus dem Projektkontext

In diesem Abschnitt wird ein kleiner Ausschnitt der vorliegenden HDF5-Daten vorgestellt. Im Fokus stehen dabei die Inhalte, die für diese Arbeit von zentraler Bedeutung sind. In der folgenden Tabelle 2.2 sind die für diese Arbeit relevanten Datensätze aus den HDF5-Dateien mit ihrer Gruppenzugehörigkeit, den Dimensionen sowie einer kurzen Beschreibung des Inhalts dargestellt.

Gruppe	**Datensatz**	**Form**	**Inhalt**
BeamStabilizer	RMSD	5300×1	Root-Mean-Square-Deviation als Maß für die Stabilität des Strahls.
LISA	mean	5300×4	Mittelwerte der Messsignale.
	peak	5300×4	Spitzenwerte der Messsignale.
	background	5300×4	Hintergrundwerte der Messsignale.
	time	5300×1	Zeitpunkt der Messsignale.

Tabelle 2.2: Auszug relevanter Datensätze aus den HDF5-Dateien mit Gruppen, Dimensionen und Inhalt.

Die in Tabelle 2.2 angegebene Form beschreibt die Dimensionen der jeweiligen Datensätze. So entspricht eine Form von 5300×1 einer Messreihe mit 5300 Zeilen und einer Spalte, während 5300×4 bedeutet, dass der Datensatz aus 5300 Zeilen und vier Spalten besteht. Darüber hinaus umfasst die HDF5-Datei zahlreiche weitere Gruppen und Datensätze, welche zusätzliche Mess- und Metainformationen enthalten. Diese werden im Rahmen dieser Arbeit jedoch nicht im Detail betrachtet, da der Fokus auf den für die Abfrageoptimierung relevanten Strukturen liegt. Eine vollständige Darstellung der hierarchischen Organisation ist im Anhang C enthalten.

2.3 Vorteile des HDF5-Formats

Das HDF5-Format überzeugt insbesondere durch seine Eignung für die Verarbeitung umfangreicher und komplex strukturierter wissenschaftlicher Daten. Ein zentraler Vorteil liegt in der selbstbeschreibenden Struktur von HDF5-Dateien: Sowohl Dateien als auch Gruppen und Datasets können mit eingebetteten Metadaten versehen werden. Dadurch entfällt die Notwendigkeit externer Metadatendateien, da alle relevanten Informationen direkt innerhalb der Datei abgelegt und automatisiert ausgelesen werden können. Ein weiterer Pluspunkt ist die Möglichkeit, Daten komprimiert zu speichern und dennoch effizient auf Teilmengen zuzugreifen. Durch sogenanntes Slicing lassen sich gezielt bestimmte Bereiche eines Datasets extrahieren, ohne dass der komplette Datensatz in den Arbeitsspeicher geladen werden muss – ein entscheidender Vorteil im Umgang mit sehr großen Datenbeständen. HDF5 ermöglicht darüber hinaus die gleichzeitige Speicherung unterschiedlicher Datentypen innerhalb einer einzigen Datei. So können beispielsweise numerische Werte, Texte und Bilddaten gemeinsam strukturiert abgelegt werden. Dies fördert eine zentrale und übersichtliche Organisation komplexer Daten aus Forschungsprojekten. Nicht zuletzt handelt es sich bei HDF5 um ein offenes, plattformunabhängiges Format, das frei verfügbar ist und von vielen gängigen Programmiersprachen und Analysetools unterstützt wird. Dazu zählen unter anderem Python, R, MATLAB sowie geowissenschaftliche Software wie QGIS und ArcGIS. Diese breite Kompatibilität erleichtert den Austausch von Daten und deren Einsatz in interdisziplinären Umgebungen [Lea25].

2.4 Anwendungsfelder des HDF5-Formats

Hinweis: Die Inhalte dieses Abschnitts orientieren sich überwiegend an der Darstellung in [The06].
Das Dateiformat HDF5 wird breitgefächert in naturwissenschaftlichen und technischen Disziplinen eingesetzt. Besonders in der akademischen Forschung wird es aufgrund seiner plattformunabhängigen Struktur und der Fähigkeit zur Speicherung großer, heterogener Datensätze geschätzt. Es bietet eine standardisierte Lösung für die Archivierung und Verarbeitung komplexer Datenbestände und wird von zahlreichen Analyseumgebungen wie `MATLAB`, `Mathematica`, `ParaView`, `Python` und `R` nativ unterstützt.
Ein prominentes Anwendungsfeld stellt die Astronomie dar, in der HDF5 beispielsweise in Observatorien wie `LOFAR` oder `LIGO` zur Speicherung und Auswertung großer, schnell anfallender Datenmengen verwendet wird. Die Möglichkeit, unterschiedliche Datenquellen in einem einheitlichen Format zu integrieren und unabhängig von der zugrundeliegenden Hardwarearchitektur zu verarbeiten – von lokalen Rechnern bis hin zu Hochleistungsclustern – macht HDF5 in dieser Domäne unverzichtbar.
In der Physik erfüllt HDF5 zentrale Anforderungen bei großskaligen Simulationen und Experimenten, etwa in der Plasmaphysik oder an Synchrotronanlagen. Die hohe Schreibgeschwindigkeit, die flexible Datenstruktur sowie die Unterstützung komplexer Datentypen machen das Format besonders geeignet für anspruchsvolles Datenmanagement. Durch sogenannte virtuelle Datasets lassen sich verteilte Datenquellen logisch zusammenfassen, was insbesondere für die Echtzeitsteuerung experimenteller Abläufe von Vorteil ist.
Auch in den Ingenieurwissenschaften – insbesondere in Bereichen wie der Automobiltechnik oder der Luft- und Raumfahrt – ist HDF5 ein essenzielles Werkzeug zur Erfassung, Analyse und Simulation von Sensordaten. Es erlaubt die gleichzeitige Speicherung mehrerer synchroner Datenströme inklusive ihrer Metadaten in strukturierter Form. Industrieprojekte wie `CMORE`, die sich mit autonomen Fahrsystemen befassen, nutzen HDF5 zur effizienten Verwaltung und Auswertung umfangreicher Sensordatenbestände. Diese vielfältigen Anwendungsfelder unterstreichen die hohe Flexibilität und Leistungsfähigkeit des HDF5-Formats im Kontext datenintensiver wissenschaftlicher und technischer Anwendungen.

2.5 Typische Anfragen und Nutzungszenarien

Für die Analyse geeigneter Zugriffsstrukturen ist es zunächst notwendig, die im Kontext der Forschungsfrage relevanten Abfragearten zu identifizieren und hinsichtlich ihrer technischen Anforderungen zu bewerten. Die vom Leibniz-Institut für Atmosphärenphysik (IAP) formulierten Anwendungsbeispiele, die im Rahmen dieser Arbeit durch den Betreuer, Prof. Dr. Gerd Baumgarten, weitergeleitet wurden, verdeutlichen diese Fokussierung. Im Zentrum stehen statistische Auswertungen der Attribute `Peak` und `Background`, die zentrale physikalische Größen der Messverfahren darstellen. Konkret umfassen sie Abfragen, die die obersten beziehungsweise untersten zehn Prozent der Werte dieser Attribute bestimmen (im Folgenden zusammenfassend als Top-N%-Anfragen bezeichnet), sowie Bereichsanfragen auf dem Attribut `Mean`, bei denen alle Werte zwischen zehn und achtzig Prozent des maximalen `Mean`-Werts berücksichtigt werden.
Die Relevanz dieser Abfragen ergibt sich aus der zugrunde liegenden Datenstruktur: Jeder empfangene Laserimpuls wird in vier Kanäle aufgeteilt, die unterschiedlichen Lichtdetektoren zugeordnet sind. Für jeden Kanal werden mehrere Messgrößen erfasst, von denen insbesondere drei Attribute zentral sind. Der `Peak`-Wert beschreibt die maximale Lichtintensität und dient als Indikator für reflektierende atmosphärische Schichten. Der `Background`-Wert misst hingegen die Grundintensität unabhängig vom eigentlichen Signal und ist notwendig, um Rauschkomponenten zu identifizieren und zu kompensieren. Der `Mean`-Wert ergänzt diese Informationen durch eine zeitlich gemittelte Intensität.

Aggregierte Attribute Da diese Messgrößen jeweils für vier parallele Kanäle vorliegen, werden sie in dieser Arbeit zu aggregierten Attributen zusammengeführt: Aus den vier `Peak`-Werten wird das Maximum als `row_max_peak` gebildet, aus den vier `Background`-Werten das Minimum als `row_min`

_background, und aus den vier Mean-Werten wiederum das Maximum als row_max_mean. Im weiteren Verlauf dieser Arbeit beziehen sich die Begriffe row_max_peak, row_min_background und row_max_mean stets auf die zuvor beschriebenen, aus den vier parallelen Kanälen aggregierten Attribute. Die folgende Tabelle 2.3 fasst die Bildung der aggregierten Attribute zusammen.

Attribut	**Berechnungsregel**	**Ergebnis-Spalte**
Peak	Maximum aus vier Kanälen	row_max_peak
Background	Minimum aus vier Kanälen	row_min_background
Mean	Maximum aus vier Kanälen	row_max_mean

Tabelle 2.3: Übersicht über die aggregierten Attribute aus den vier parallelen Kanälen

Relevante Anfragearten Darauf aufbauend ergibt sich ein klarer Fokus auf bestimmte Anfragetypen. Dazu gehören insbesondere klassische Bereichsanfragen, bei denen alle Werte innerhalb eines definierten Intervalls berücksichtigt werden, sowie deren Spezialform, die Top-N%-Anfragen, bei denen gezielt die höchsten oder niedrigsten Werte eines Attributs extrahiert werden. Solche Anfragen erfordern eine Kombination aus Sortierung und Selektion und stellen hohe Anforderungen an die zugrunde liegende Zugriffsstruktur. Darüber hinaus fließen auch Punktanfragen und Aggregationen in die Analyse ein, die zwar nicht Teil der ursprünglichen Anforderungen waren, jedoch ergänzend untersucht werden, um zusätzliche Einblicke in die Leistungsfähigkeit der verschiedenen Zugriffsstrukturen zu ermöglichen. In der folgenden Tabelle 2.4 wird dargestellt, welche Anfragearten in der Arbeit untersucht werden.

Anfrage	**Attribut**	**Definition**
Bereichsanfrage (X % bis Y %)	row_max_mean	Werte im Bereich von X % bis Y % des globalen Maximums von row_max_mean.
Top-N %	row_max_peak	Die größten N % der row_max_peak.
Bottom-N %	row_min_background	Die kleinsten N % der row_min_background.

Tabelle 2.4: Zusammenfassung der in der Arbeit verwendeten Anfragearten; Nullwerte (0) werden bei den Bottom-N %-Anfragen ausgeschlossen.

Relevante Komplexitätsmaße Für die Bewertung der Datenstrukturen sind auch die Komplexitäten der Operationen Zugriff, Suche und Einfügen relevant. Die Zugriffskomplexität spielt eine zentrale Rolle bei Punktabfragen, etwa bei der gezielten Auswahl eines einzelnen Wertes, sowie bei Bereichsanfragen, die typischerweise eine Traversierung benachbarter Dateneinträge erfordern. Eine geringe Zugriffskomplexität ist somit grundlegend für effiziente Aggregationen und Top-N%-Abfragen, die häufig auf solchen Bereichsanfragen aufbauen.
Die Suchkomplexität ist eng damit verbunden, wird jedoch vor allem dann relevant, wenn der Zugriffsschlüssel nicht bekannt ist. Bei Bereichsanfragen oder mehrdimensionalen Zugriffen muss zunächst der Startpunkt lokalisiert und anschließend eine gezielte Traversierung durchgeführt werden. Ist der Schlüssel hingegen bekannt, reduziert sich der Aufwand auf den eigentlichen Zugriff. Auch komplexere Abfragen, wie Top-N%- oder Mittelwertberechnungen über Teilbereiche, profitieren von effizienten Suchstrukturen. Im Gegensatz dazu ist die Einfügekomplexität in diesem Anwendungskontext von untergeordneter Bedeutung. Die betrachteten Messdaten liegen in der Regel als vorab erfasste, unveränderliche Datenmengen vor. Schreiboperationen finden kaum statt, sodass eine höhere Einfügekomplexität in Kauf genommen werden kann. Dasselbe gilt für Löschoperationen, die im Rahmen typischer Analyseworkflows praktisch keine Rolle spielen. Gemeinsam bilden die genannten Anfragearten und die operationellen Komplexitäten Suche, Zugriff und Einfügen das Fundament für die technische Bewertung von Zugriffsstrukturen. Sie sind entscheidend für die Gesamtleistung eines Datenzugriffssystems und dienen als Grundlage für den strukturierten Vergleich der betrachteten Ansätze. Im folgenden Abschnitt werden darauf aufbauend die zentralen Zugriffsstrukturen in Datenbanksystemen vorgestellt.

2.6 Zugriffsstrukturen in Datenbanksystemen

Der Zugriff auf Daten stellt einen der grundlegendsten und zugleich leistungsrelevantesten Aspekte moderner Datenbanksysteme dar. Besonders bei großen und komplexen Datenmengen, wie sie etwa in wissenschaftlichen Messdaten vorkommen, ist eine naive sequentielle Suche nicht praktikabel. Sie führt zu langen Antwortzeiten und ineffizienter Ressourcennutzung. Um diesen Herausforderungen zu begegnen, wurden sogenannte Zugriffsstrukturen entwickelt – spezialisierte Datenstrukturen, die schnellere Datenzugriffe ermöglichen. Dazu zählen unter anderem Indizes, Bäume und Hash-Tabellen. Sie erlauben es, relevante Informationen schnell und ressourcenschonend aufzufinden, indem sie eine logische Struktur über den physisch gespeicherten Daten etablieren.

Bäume

Ein Baum ist eine abstrakte Datenstruktur, die der hierarchischen Organisation von Informationen dient. Er besteht aus sogenannten Knoten, die in einer Eltern-Kind-Beziehung zueinander stehen. Der oberste Knoten eines Baumes wird als Wurzel bezeichnet. Jeder Knoten kann mehrere Kindknoten besitzen; Knoten ohne Kinder werden als Blätter bezeichnet. Knoten, die denselben Elternknoten teilen, heißen Geschwisterknoten. In jedem Knoten befindet sich in der Regel ein Schlüssel, also ein Wert oder eine Zahl, der zum Vergleichen und zur Navigation innerhalb der Struktur dient.
Die Tiefe eines Knotens beschreibt die Anzahl der Kanten von der Wurzel bis zu diesem Knoten. Die Wurzel selbst hat somit eine Tiefe von 0, während jeder Schritt zu einem Kindknoten die Tiefe um 1 erhöht. Im Gegensatz dazu bezeichnet die Höhe eines Knotens die Länge des längsten Pfades von diesem Knoten bis zu einem Blatt. Die Höhe eines Blattes ist folglich 0, während die Höhe der Wurzel der maximalen Tiefe des gesamten Baumes entspricht. In der folgenden Abbildung 2.3 ist ein Beispiel für einen Baum[1] dargestellt, das die genannten Konzepte veranschaulicht.

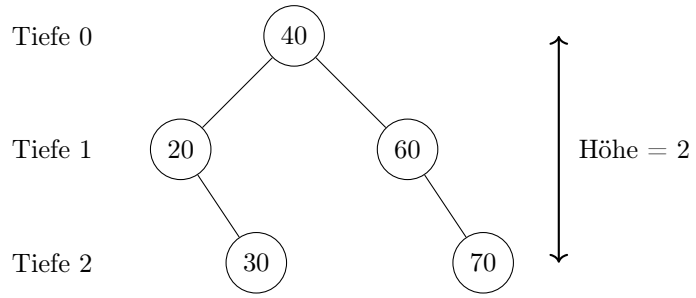


Abbildung 2.3: Baum mit den Knoten 20, 30, 40, 60 und 70, sowie Markierung von Tiefe und Höhe.

Vergleich Balancierter vs. nicht balancierter Baum: Ein balancierter Baum ist eine Baumstruktur, bei der die Höhe der Teilbäume so kontrolliert wird, dass sie sich nur geringfügig voneinander unterscheiden. Dadurch wird gewährleistet, dass die Pfadlängen von der Wurzel zu den Blättern ungefähr gleich sind. Ein unbalancierter Baum hingegen führt keine Maßnahmen durch, um die Höhe der Teilbäume auszugleichen. Durch das Fehlen einer Balancierung können Suchoperationen im Worst Case lineare Zeit erfordern. In den nächsten beiden Abbildungen (2.4 und 2.5) wird ein Vergleich zwischen einem balancierten und einem nicht balancierten Baum dargestellt, um die Unterschiede in der Höhe der Teilbäume und den daraus resultierenden Auswirkungen auf die Effizienz der Operationen zu verdeutlichen.

[1] Bäume haben im Allgemeinen keine Sortierungsregel. In den hier dargestellten Beispielen sind die Knoten jedoch der Anschaulichkeit halber sortiert.

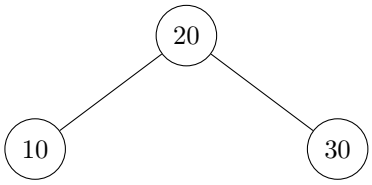


Abbildung 2.4: Balancierter Baum: Höhe der Teilbäume nahezu gleich, effiziente Operationen.

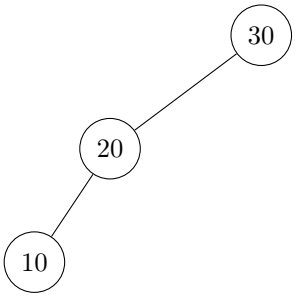


Abbildung 2.5: Nicht balancierter Baum: Lineare Struktur mit den Knoten 10, 20 und 30.

Binärbaum

Eine spezielle Form der Baumstruktur ist der **Binärbaum**, bei dem jeder Knoten höchstens zwei Kindknoten besitzen darf – einen linken und einen rechten. Jeder Knoten enthält typischerweise einen Schlüsselwert. Diese Struktur bildet die Grundlage für viele weiterentwickelte Baumtypen. In der nächsten Abbildung 2.6 ist ein Beispiel für einen Binärbaum dargestellt[2].

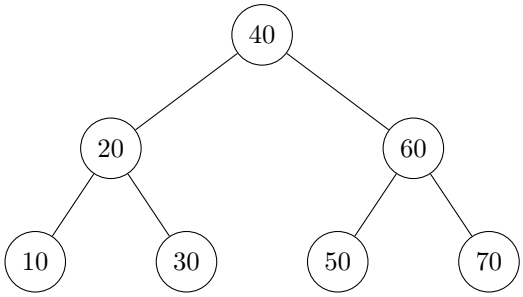


Abbildung 2.6: Binärbaum mit den Knoten 10, 20, 30, 40, 60, und 70.

2.6.1 Binärer Suchbaum

Der binäre Suchbaum, häufig als Binary Search Tree oder BST bezeichnet, ist eine der grundlegenden Datenstrukturen im Bereich der Informatik und bildet das theoretische Fundament zahlreicher komplexerer Baumvarianten. Die Struktur besteht aus Knoten, wobei jeder Knoten einen Schlüssel enthält und maximal zwei Kindknoten besitzt [Hib62]. Die Schlüsselverteilung orientiert sich an einem eindeutigen Prinzip: Im linken Teilbaum befinden sich ausschließlich Schlüssel mit geringerem Wert als derjenige des betrachteten Knotens, im rechten Teilbaum dagegen ausschließlich Schlüssel mit größerem Wert. Diese geordnete Struktur ermöglicht eine effiziente Suche und erleichtert auch das Einfügen sowie das Löschen von Elementen, indem man sich von der Wurzel nach unten bewegt und den Schlüsselvergleich auf jeder Ebene wiederholt. In der Praxis findet der BST in zahlreichen Algorithmen und Systemen Anwendung, etwa bei der Implementierung von Symboltabellen, in elementaren Datenbankindizes oder in Speichermanagementsystemen, wo ein schneller Zugriff auf Schlüsselwerte erforderlich ist. Obwohl der BST konzeptionell einfach ist, bildet er die Grundlage für weiterentwickelte und balancierte Varianten wie AVL-Bäume. Die folgende Abbildung 2.7 zeigt einen binären Suchbaum nach dem Einfügen der Werte 10, 30, 50, 40, 70 und 80.

[2]Binärbäume müssen im Allgemeinen nicht sortiert sein wie Binärsuchbäume. In diesem Beispiel wurden die Knoten jedoch der besseren Anschaulichkeit halber sortiert dargestellt.

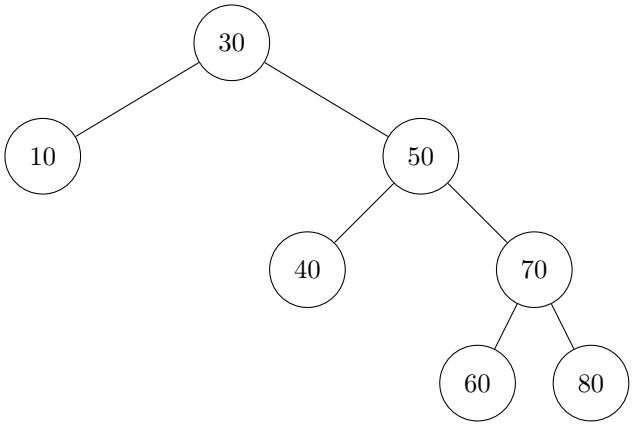


Abbildung 2.7: Binärsuchbaum mit den Knoten 10, 30, 40, 50, 70 und 80.

Dieses Beispiel illustriert die Einfachheit der Funktionsweise, zeigt aber auch die mögliche Problematik: Wenn Werte in streng sortierter Reihenfolge eingefügt werden, degeneriert der Baum[3] zu einer linearen Struktur, was die Suche verlangsamt. Der binäre Suchbaum ist in seiner klassischen Form eine eindimensionale Struktur, da er nur ein einziges Schlüsselattribut zur Ordnung der Knoten verwendet und somit nicht für mehrdimensionale Daten geeignet ist. Hinsichtlich der Balancierung ist der binäre Suchbaum von Natur aus unbalanciert, was bedeutet, dass seine Höhe stark von der Reihenfolge der Einfügungen abhängt. Bei zufälligen Daten ist die Höhe im Durchschnitt logarithmisch zur Anzahl der Elemente, während sie im ungünstigsten Fall linear wächst, wenn die Daten bereits sortiert eingefügt werden. Diese Eigenschaften wirken sich unmittelbar auf die Eignung für verschiedene Anfragearten aus. Für Punktanfragen, also Abfragen nach einem exakten Schlüssel, ist der BST im Durchschnitt gut geeignet, da die Suche im balancierten Fall effizient erfolgt. Bereichsanfragen können ebenfalls durch eine In-Order-Traversierung abgewickelt werden, jedoch hängt die Effizienz stark vom Balancierungszustand ab. Für Top-N%-Abfragen, bei denen die größten oder kleinsten N Elemente benötigt werden, bietet der BST keine native Optimierung. Aggregationsabfragen wie Summen oder Mittelwerte sind ebenfalls nur dann effizient, wenn zusätzliche Metainformationen wie Teilbaumgrößen[4] oder kumulierte Werte[5] gepflegt werden; ohne diese Erweiterungen ist eine vollständige Traversierung erforderlich, was zu linearen Kosten führt.

Parameter	**BST**
Dimension	Eindimensional
Balancierung	Unbalanciert
Punktanfragen	Eingeschränkt
Bereichsanfragen	Eingeschränkt
Top-N%-Anfragen	Eingeschränkt
Aggregationen	Eingeschränkt

Tabelle 2.5: Analyseparameter für den Binären Suchbaum (BST).

[3] Von Degeneration spricht man, wenn der Baum seine Balancierung verliert und im Extremfall einer linearen Liste ähnelt. Dies führt zu einer linearen anstelle einer logarithmischen Laufzeit bei Such- und Einfügeoperationen.

[4] Mit „Teilbaumgrößen“ sind in diesem Zusammenhang zusätzliche Metainformationen gemeint, die in jedem Knoten gespeichert werden und die Anzahl der Elemente im jeweiligen Teilbaum angeben. Dadurch lassen sich beispielsweise Rangabfragen oder Größenbestimmungen von Teilbereichen effizient durchführen, ohne alle Elemente traversieren zu müssen.

[5] „Kumulierte Werte“ bezeichnen vorab berechnete Summen oder andere aggregierte Kennzahlen, die für den jeweiligen Teilbaum gespeichert werden. Dies ermöglicht es, Aggregationsabfragen wie Summen oder Mittelwerte deutlich schneller zu beantworten, da keine vollständige Traversierung des Baums nötig ist.

Die Zeitkomplexität des BST hängt unmittelbar von seiner Höhe ab. Im Durchschnitt, bei zufälliger Einfügereihenfolge, ist die Höhe logarithmisch zur Anzahl der Elemente, sodass Einfügen, Zugriff und Suche jeweils $O(\log n)$ erfordern. Im Worst-Case, wenn die Einfügereihenfolge sortiert ist und der Baum degeneriert, entspricht die Höhe der Anzahl der Elemente und führt zu einer linearen Komplexität von $O(n)$ für alle Operationen. Diese Diskrepanz macht den BST in seiner klassischen Form unvorhersehbar für Anwendungen, die konsistente Antwortzeiten erfordern, und begründet die Entwicklung balancierter Varianten.

BST	**Komplexität (avg / worst)**
Einfügen	$O(\log n)$ / $O(n)$
Suche	$O(\log n)$ / $O(n)$
Zugriff	$O(\log n)$ / $O(n)$

Tabelle 2.6: Komplexitätsparameter für den Binären Suchbaum (BST) [CLRS09].

2.6.2 AVL-Baum

Der AVL-Baum ist eine balancierte Variante des klassischen binären Suchbaums und wurde entwickelt, um die Problematik unbalancierter Bäume und deren mögliche Degeneration zu linearen Strukturen zu vermeiden. Wie beim herkömmlichen Binärbaum basiert auch hier die Struktur auf der Regel, dass alle Schlüssel im linken Teilbaum kleiner und alle Schlüssel im rechten Teilbaum größer als der Schlüssel des aktuellen Knotens sind. Der entscheidende Unterschied liegt in der Balancebedingung: Bei einem AVL-Baum darf sich die Höhe des linken und rechten Teilbaums eines Knotens höchstens um eins unterscheiden. Wird diese Bedingung durch Einfügen oder Löschen von Elementen verletzt, führt der Baum unmittelbar eine oder mehrere Rotationen durch, um das Gleichgewicht wiederherzustellen. Diese Eigenschaft garantiert, dass die Höhe des Baums stets logarithmisch zur Anzahl der gespeicherten Elemente bleibt, was eine vorhersehbare und effiziente Laufzeit für alle grundlegenden Operationen ermöglicht. AVL-Bäume werden typischerweise in Datenbanksystemen und Anwendungen eingesetzt, in denen ein hoher Anteil an Suchoperationen und eine konsistente Performance für Lesezugriffe erforderlich ist [Bro25].
Das folgende Beispiel zeigt die Entwicklung eines AVL-Baums beim sukzessiven Einfügen der Werte 10, 20, und 30. Nach jedem Einfügevorgang wird überprüft, ob die AVL-Bedingung – also die sich höchstens um eins unterscheidende Höhe der Teilbäume – verletzt ist. Falls dies der Fall ist, wird eine Rotation durchgeführt, um die Balance wiederherzustellen. Zunächst werden die Werte 10 und 20 eingefügt. Der Baum bleibt nach diesen beiden Schritten balanciert, da die Höhen der Teilbäume noch im erlaubten Bereich liegen. Nach dem anschließenden Einfügen des Wertes 30 ergibt sich jedoch eine lineare Kette der Knoten (10–20–30), was zu einer nach rechts geneigten Struktur führt und die AVL-Bedingung verletzt. Zur Wiederherstellung der Balance wird eine einfache Linksrotation durchgeführt. Dabei wird der Knoten 20 zur neuen Wurzel verschoben. Die Knoten 10 und 30 werden entsprechend als linkes bzw. rechtes Kind angeordnet. Dies wird in der folgenden Abbildung 2.8 veranschaulicht.

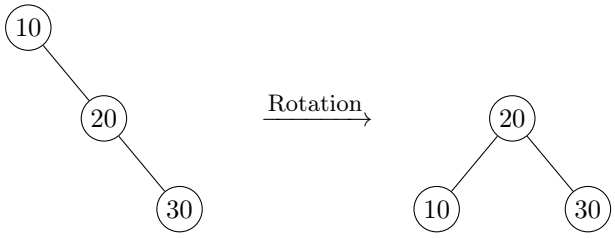


Abbildung 2.8: Links: Ungleichgewicht nach dem Einfügen von 10, 20 und 30. Die Rotation bringt den Baum in einen balancierten Zustand.

Der Endzustand zeigt einen balancierten AVL-Baum mit der Wurzel 20. Die Rotationen während des Einfügeprozesses stellen sicher, dass Such- und Einfügeoperationen auch bei wachsender Knotenzahl effizient mit logarithmischer Komplexität durchgeführt werden können. Der AVL-Baum ist eine eindimensionale Struktur, da er ausschließlich nach einem einzigen Schlüssel organisiert wird und nicht für mehrdimensionale Daten wie zeitlich-räumliche Abfragen optimiert ist. Hinsichtlich der Balancierung ist der AVL-Baum streng balanciert, was bedeutet, dass die Baumhöhe stets innerhalb von $O(\log n)$ bleibt. Diese Eigenschaft macht ihn besonders effizient für Punktanfragen, bei denen nach einem exakten Schlüssel gesucht wird, da sowohl im Durchschnitt als auch im schlechtesten Fall logarithmische Laufzeiten erreicht werden. Bereichsanfragen können ebenfalls effizient abgewickelt werden, da die geordnete Struktur eine sortierte Traversierung der Daten erlaubt; die Effizienz bleibt auch hier logarithmisch für das Finden der Start- und Endpunkte, während die Ausgabe aller Ergebnisse linear in der Anzahl der zurückgegebenen Elemente erfolgt. Top-N%-Abfragen lassen sich mit einer modifizierten Traversierung umsetzen, sind jedoch nicht speziell optimiert und profitieren nicht in besonderem Maße von der Balancierung. Aggregationsabfragen wie Summen oder Mittelwerte sind in der Grundstruktur nicht direkt unterstützt; eine effiziente Umsetzung erfordert zusätzliche gespeicherte Metainformationen wie Teilsummen oder Knotengrößen.

Parameter	**AVL-Baum**
Dimension	Eindimensional
Balancierung	balanciert
Punktanfragen	Gut geeignet
Bereichsanfragen	Gut geeignet
Top-N%-Anfragen	Eingeschränkt geeignet
Aggregationen	Eingeschränkt geeignet

Tabelle 2.7: Analyseparameter für den AVL-Baum.

Durch die strenge Balancierung bleibt die Höhe des AVL-Baums logarithmisch, wodurch alle grundlegenden Operationen wie Einfügen, Zugriff und Suche im Durchschnitt und auch im schlechtesten Fall $O(log\, n)$ erfordern. Beim Einfügen kommen neben dem Finden der richtigen Position zusätzliche Rotationen hinzu, die ebenfalls in logarithmischer Zeit ausgeführt werden. Der Zugriff auf ein Element erfolgt wie eine Suchoperation entlang des Pfads von der Wurzel zu einem Blatt und ist ebenfalls logarithmisch. Da der Baum niemals degeneriert, gibt es keinen Fall, in dem eine Operation lineare Zeit benötigen würde.

AVL-Baum	**Komplexität (avg / worst)**
Einfügen	$O(\log n)$ / $O(\log n)$
Suche	$O(\log n)$ / $O(\log n)$
Zugriff	$O(\log n)$ / $O(\log n)$

Tabelle 2.8: Komplexitätsparameter für den AVL-Baum [Bro25].

2.6.3 B-Baum

Der B-Baum ist eine dynamische und balancierte Baumstruktur, die in Datenbanksystemen häufig als Indexstruktur eingesetzt wird. Es handelt sich dabei um einen sogenannten Mehrwegebau, bei dem jeder Knoten mehrere Schlüsselwerte enthalten kann und entsprechend viele Nachfolger besitzt. Ziel dieser Struktur ist es, einen effizienten Zugriff auf große Datenmengen bei gleichzeitig möglichst wenigen I/O-Operationen zu ermöglichen. Ein B-Baum wird durch eine Ordnung m charakterisiert, wobei m die minimale Anzahl von Schlüsseln pro Knoten (außer der Wurzel) angibt. Jeder Knoten enthält mindestens m und höchstens $2m$ Schlüsselwerte. Ein Knoten mit i Schlüsselwerten verfügt über genau $i + 1$ Nachfolger. Zudem gilt die Eigenschaft, dass alle Blattknoten auf derselben Ebene liegen, was die Ausgeglichenheit des Baums garantiert. Die Höhe des Baumes wächst dabei nur logarithmisch zur Anzahl

der gespeicherten Elemente, wodurch selbst bei sehr großen Datenmengen effiziente Zugriffsmöglichkeiten gewährleistet werden. Die Schlüsselwerte innerhalb des Baums sind stets sortiert. Operationen wie das Suchen, Einfügen und Löschen erfolgen so, dass die Balance-Eigenschaften des Baumes erhalten bleiben [HSS19]. Im folgenden Beispiel 2.9 wird das Endergebnis des B-Baums der Ordnung $m = 1$ nach dem Einfügen der Elemente mit den Werten 10, 20, 30, 40, 50 und 60 veranschaulicht.

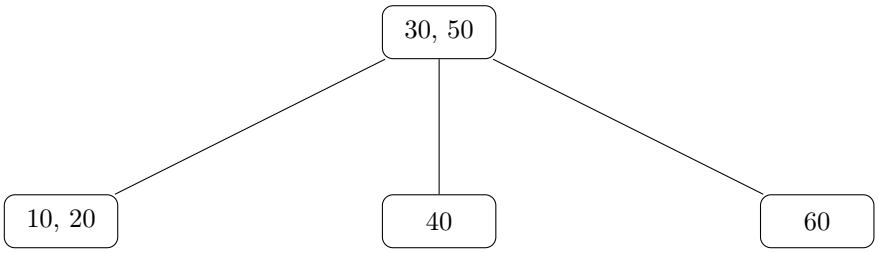


Abbildung 2.9: Endstruktur eines B-Baums der Ordnung 1 nach dem Einfügen der Werte 10, 20, 30, 40, 50 und 60.

Die Struktur eines B-Baums hängt von der Reihenfolge ab, in der die Schlüssel eingefügt werden. Während die Gesamtheit der gespeicherten Schlüssel sowie ihre sortierte Reihenfolge im Baum immer identisch sind, kann sich die konkrete Verteilung der Schlüssel auf die Knoten und damit die Baumstruktur verändern. Dies liegt daran, dass bei jedem Einfügen überprüft wird, ob ein Knoten die maximale Schlüsselanzahl überschreitet; in diesem Fall wird der Knoten gesplittet, wobei der mittlere Schlüssel nach oben wandert. Die Position, an der solche Splits auftreten, ist direkt von der Einfügereihenfolge abhängig.
Das zuvor gezeigte Beispiel entstand durch sukzessives Einfügen der Werte 10, 20, 30, 40, 50 und 60. Wird jedoch dieselbe Menge an Schlüsseln in umgekehrter Reihenfolge – also 60, 50, 40, 30, 20 und 10 – eingefügt, ergibt sich eine andere, aber ebenfalls gültige Baumstruktur, wie in der folgenden Abbildung 2.10 dargestellt.

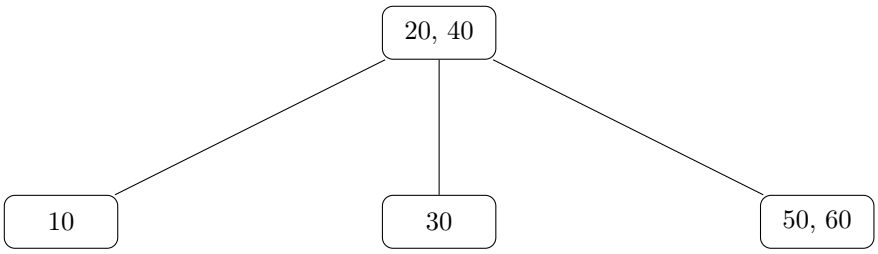


Abbildung 2.10: B-Baum der Ordnung 1 nach Einfügen der Werte 60, 50, 40, 30, 20 und 10 (umgekehrte Reihenfolge).

Auch in diesem Fall erfüllt der Baum alle Eigenschaften eines B-Baums: Die Schlüssel sind sortiert, alle Blätter liegen auf derselben Ebene, und die Höhe wächst nur logarithmisch mit der Anzahl der Elemente. Die unterschiedliche Verteilung der Schlüssel auf die Knoten verdeutlicht jedoch den Einfluss der Einfüge-Reihenfolge.
Der B-Baum ist eine eindimensionale Struktur, da er ausschließlich nach einem einzelnen Schlüsselattribut geordnet ist. Er ist streng balanciert, da alle Blätter auf derselben Höhe liegen und Einfüge- sowie Löschoperationen durch Teilungen und Verschmelzungen sofort eine erneute Balancierung herbeiführen. Für Punktanfragen eignet sich der B-Baum hervorragend, da Suchoperationen durch die geordnete Struktur effizient verlaufen und die Höhe logarithmisch ist. Bereichsanfragen werden ebenfalls gut unterstützt, da eine Traversierung der Blätter in geordneter Reihenfolge möglich ist; dies ist besonders nützlich bei

Datenbanksystemen für Abfragen über Schlüsselintervalle. Top-N%-Abfragen lassen sich effizient umsetzen, indem eine Traversierung der Blattebene beginnt und nach Erreichen der N größten oder kleinsten Schlüssel abgebrochen wird. Aggregationen wie Summen oder Durchschnitte können zwar durch Traversieren der Blattebene durchgeführt werden, erfordern jedoch, ähnlich wie bei AVL-Bäumen, zusätzliche Informationen in den Knoten, wenn sie in sublinearer Zeit berechnet werden sollen.

Parameter	**B-Baum**
Dimension	Eindimensional
Balancierung	Balanciert
Punktanfragen	Gut geeignet
Bereichsanfragen	Gut geeignet
Top-N%-Anfragen	Gut geeignet
Aggregationen	Eingeschränkt geeignet

Tabelle 2.9: Analyseparameter für den B-Baum.

Da der B-Baum balanciert ist und die Höhe logarithmisch mit der Anzahl der gespeicherten Elemente wächst, bleiben die grundlegenden Operationen wie Einfügen, Suchen und Zugriff in allen Fällen innerhalb von $O(\log_m n)$. Einfügeoperationen können zusätzliche Kosten durch das Teilen von Knoten verursachen, doch auch diese Operationen sind durch die logarithmische Höhe beschränkt. Die Suche nach einem Schlüssel verläuft durch seitenweise Vergleiche innerhalb der Knoten, was typischerweise sehr effizient ist, da mehrere Schlüssel pro Seite gespeichert werden und dadurch weniger I/O-Operationen erforderlich sind. Im Gegensatz zu binären Strukturen gibt es keinen Fall, in dem die Komplexität auf lineare Zeit ansteigt, da der B-Baum seine Balance stets aktiv aufrechterhält.

B-Baum	**Komplexität (avg)**
Einfügen	$O(\log_m n)$
Suche	$O(\log_m n)$
Zugriff	$O(\log_m n)$

Tabelle 2.10: Komplexitätsparameter für den B-Baum [HSS19].

2.6.4 B$^+$-Baum

Der B$^+$-Baum ist eine in der Praxis weit verbreitete Variante des klassischen B-Baums und wird insbesondere in Datenbanksystemen zur effizienten Verwaltung großer Datenmengen eingesetzt. Er zeichnet sich durch eine klare Trennung zwischen inneren Knoten, die ausschließlich zur Navigation dienen, und Blattknoten, die die eigentlichen Dateneinträge enthalten, aus. Wie beim B-Baum sind auch im B$^+$-Baum alle Blattknoten auf derselben Höhe angeordnet, was die Balanciertheit der Struktur garantiert. Die inneren Knoten beinhalten nur die Schlüssel, also die Zugriffsattributwerte, und Zeiger auf Kindseiten, während die vollständigen Datensätze beziehungsweise Verweise, etwa TIDs (Tuple Identifiers), ausschließlich in den Blättern abgelegt werden. Ein wesentliches Merkmal des B$^+$-Baums ist die lineare Verkettung der Blattknoten, wodurch effiziente Bereichsanfragen, sogenannte Range-Scans, ermöglicht werden. Im Unterschied zum B-Baum bleiben im B$^+$-Baum beim Löschen von Einträgen die Schlüsselwerte in den inneren Knoten zunächst erhalten, selbst wenn die zugehörigen Datensätze bereits entfernt wurden. Eine Aktualisierung erfolgt nur im Fall eines strukturellen Unterlaufs. Dadurch wird die Komplexität der Änderungsoperationen reduziert. Ein typischer B$^+$-Baum der Ordnung m enthält in jedem inneren Knoten mindestens m und höchstens 2m Schlüssel. Die Anzahl der Zeiger pro innerem Knoten beträgt stets um eins mehr als die Anzahl der Schlüssel. Alle Blattknoten sind durch eine einfach verkettete Liste miteinander verbunden. Die Suche erfolgt stets über die inneren Knoten bis hin zur Blattebene, auch dann, wenn der gesuchte Wert bereits in einem inneren Knoten enthalten ist [HSS19]. Im folgenden

Beispiel 2.11 wird die Endstruktur eines B$^+$-Baums der Ordnung $m = 1$ nach dem Einfügen der Werte 10, 20, 30, 40, 50, 60 und 70 dargestellt.

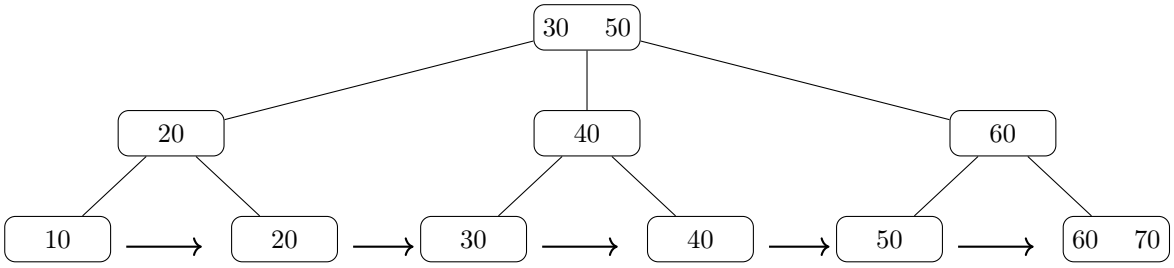


Abbildung 2.11: B$^+$-Baum der Ordnung $m = 1$ nach dem Einfügen der Schlüsselwerte 10 bis 70.

Der B$^+$-Baum ist eine eindimensionale Struktur, da er ausschließlich nach einem einzigen Schlüsselattribut geordnet ist. Er ist streng balanciert, weil alle Blattknoten auf derselben Ebene liegen und Einfügungen sowie Löschungen durch Teilungen und Verschmelzungen sofort eine erneute Balancierung herbeiführen. Punktanfragen können effizient beantwortet werden, da die Suche nach einem Schlüssel in logarithmischer Zeit erfolgt. Bereichsanfragen sind im B$^+$-Baum besonders effizient, da die Blätter nicht nur in sortierter Reihenfolge gespeichert sind, sondern zusätzlich durch Zeiger miteinander verkettet werden. Dadurch muss nach dem Auffinden des Startpunkts nicht erneut der Baum traversiert werden, sondern alle weiteren Elemente können sequenziell und ohne zusätzlichen Suchaufwand gelesen werden. Auch Top-N%-Anfragen profitieren unmittelbar von dieser Eigenschaft: Da die Blätter in einer durchgehenden Sequenz organisiert sind, lassen sich die größten oder kleinsten Werte einfach durch Traversieren der Blattebene bestimmen. Die Traversierung kann zudem genau dann abgebrochen werden, wenn die gewünschte Anzahl von Elementen erreicht ist, was die Abfrage deutlich beschleunigt. Aggregationen wie Summen oder Mittelwerte können durch Traversieren der verketteten Blätter durchgeführt werden.

Parameter	**B$^+$-Baum**
Dimension	Eindimensional
Balancierung	Streng balanciert
Punktanfragen	Gut geeignet
Bereichsanfragen	Sehr gut geeignet
Top-N%-Anfragen	Sehr gut geeignet
Aggregationen	Gut geeignet

Tabelle 2.11: Analyseparameter für den B$^+$-Baum.

Die Komplexität des B$^+$-Baums entspricht weitgehend der des klassischen B-Baums. Die Höhe wächst logarithmisch mit der Anzahl der gespeicherten Elemente, wodurch Einfügen, Suchen und Zugriff im Durchschnitt Fall $O(\log_m n)$ benötigen. Die Verkettung der Blätter ermöglicht es zusätzlich, Bereichsanfragen nach dem initialen Suchschritt mit sequentieller Komplexität in Bezug auf die Anzahl der zurückgegebenen Elemente auszuführen. Da die inneren Knoten nur Schlüssel und keine Daten enthalten, können mehr Schlüssel pro Knoten gespeichert werden, was zu einer geringeren Höhe und damit geringeren Zugriffskosten führt.

Operation	Komplexität (avg)
Einfügen	$O(\log_m n)$
Suche	$O(\log_m n)$
Zugriff	$O(\log_m n)$

Tabelle 2.12: Komplexitätsparameter für den B^+-Baum [HSS19].

2.6.5 KD-Baum

Der KD-Baum (k-dimensionaler Baum) ist eine Datenstruktur, die als Verallgemeinerung des binären Suchbaums für mehrdimensionale Daten entwickelt wurde. Während ein klassischer BST ausschließlich nach einem einzigen Schlüssel organisiert ist, ermöglicht der KD-Baum die Speicherung und effiziente Abfrage von Punkten in einem k-dimensionalen Raum. Die Grundidee besteht darin, die Daten rekursiv anhand wechselnder Dimensionen aufzuteilen: Auf der ersten Ebene wird nach der ersten Dimension sortiert, auf der zweiten Ebene nach der zweiten Dimension und so weiter, bis alle Dimensionen genutzt wurden und der Zyklus erneut beginnt. Diese abwechselnde Partitionierung führt zu einer hierarchischen Struktur, die mehrdimensionale Suchen unterstützt, ohne dass alle Punkte linear durchsucht werden müssen. Typische Anwendungen des KD-Baums finden sich überall dort, wo Abfragen nach nächsten Nachbarn oder nach Bereichen in mehreren Dimensionen vorkommen [Ben80].
Die folgende Abbildung 2.12 zeigt das Endergebnis eines KD-Baums für sieben Messpunkte mit den Attributen Zeit (in Sekunden) und Höhe (in Kilometern): $(10, 40)$, $(20, 30)$, $(15, 35)$, $(30, 20)$, $(25, 50)$, $(35, 45)$ und $(40, 25)$. Die erste Partitionierung erfolgt nach der Zeit, wobei der Medianwert 25 als Wurzel gewählt wird. Die zweite Ebene wird nach der Höhe partitioniert, sodass links der Knoten 15 und rechts der Knoten 40 entsteht. In der dritten Ebene erfolgt erneut eine Trennung nach der Zeit, wodurch die verbleibenden Punkte in Blätter einsortiert werden. Die abwechselnde Nutzung der Dimensionen ermöglicht eine strukturierte und annähernd balancierte Aufteilung des Datenraums Raumaufteilung.

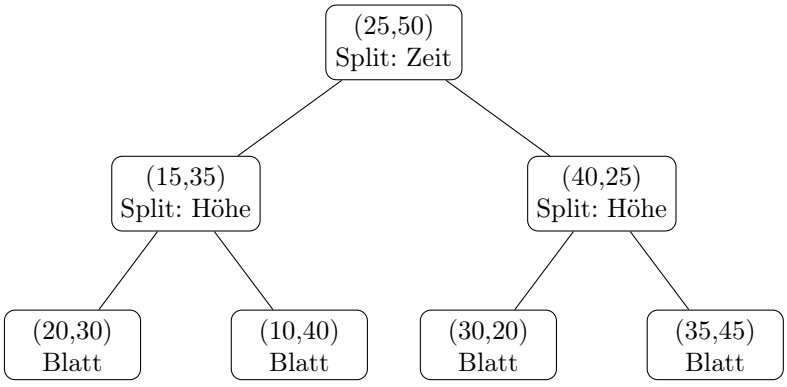


Abbildung 2.12: KD-Baum für sieben Punkte (Zeit, Höhe) mit Median-Split.

Der KD-Baum ist eine mehrdimensionale Indexstruktur, die sich besonders für Daten mit zwei oder mehr Schlüsselattributen eignet. Die Balancierung ist nicht streng garantiert wie bei AVL- oder B-Bäumen, hängt aber oft davon ab, ob beim Aufbau Mediane genutzt werden; wird dies getan, ist der Baum im Durchschnitt halbwegs balanciert. Für Punktanfragen ist der KD-Baum sehr gut geeignet, da durch das Abwechseln der Dimensionen der Suchraum effizient eingegrenzt wird und im Durchschnitt logarithmische Laufzeiten erreicht werden. Bereichsanfragen profitieren ebenfalls von der hierarchischen Partitionierung, allerdings steigt der Aufwand mit zunehmender Dimension stark an (Fluch der Dimensionalität). Top-N%-Anfragen lassen sich nur eingeschränkt effizient umsetzen, da sie häufig eine Kombination aus Bereichssuche und zusätzlicher Traversierung benötigen. Aggregationen wie Summen oder Mittelwerte

sind nicht direkt optimiert und erfordern vollständige Traversierungen oder spezielle Erweiterungen wie Range Trees.

Parameter	**KD-Baum**
Dimension	Mehrdimensional (k-dimensional)
Balancierung	Teilweise balanciert (abhängig von Medianauswahl)
Punktanfragen	Gut geeignet
Bereichsanfragen	Gut geeignet
Top-N%-Anfragen	Eingeschränkt geeignet
Aggregationen	Eingeschränkt geeignet

Tabelle 2.13: Analyseparameter für den KD-Baum.

Die Zeitkomplexität für die Operationen Zugriff, Einfügen und Suche liegt im Durchschnitt bei $O(\log n)$. da bei einer ausgewogenen Medianaufteilung die Höhe logarithmisch wächst. Im schlechtesten Fall, wenn die Daten stark ungleichmäßig verteilt sind oder keine Mediansortierung erfolgt, kann die Komplexität bis auf $O(n)$ ansteigen.

KD-Baum	**Komplexität (avg / worst)**
Einfügen	$O(\log n)$ / $O(n)$
Suche	$O(\log n)$ / $O(n)$
Zugriff	$O(\log n)$ / $O(n)$

Tabelle 2.14: Komplexitätsparameter für den KD-Baum [Ben80].

2.6.6 KDB-Baum

Der KDB-Baum ist ein mehrdimensionales Indexverfahren, das die Konzepte des KD-Baums und des B^+-Baum miteinander kombiniert. Während der KD-Baum von Bentley und Friedman [Rob81] ein binärer Baum ist, der für Hauptspeicherstrukturen entwickelt wurde und Daten anhand mehrerer Attribute hintereinander partitioniert, erweitert der KDB-Baum dieses Prinzip um die Seitenorganisation des B-Baums. Dadurch erhält der KDB-Baum einen höheren Verzweigungsgrad und eignet sich für die Speicherung auf Sekundärspeicher, wie sie in Datenbanksystemen üblich ist.
Jede Indexseite des KDB-Baumes enthält nicht einfach eine Sequenz von Schlüsselwerten, sondern einen kleinen KD-Baum, der Schnittattribute und deren Werte repräsentiert und die Datensätze in zwei Teilräume unterteilt: solche, die kleiner als der Schnittwert sind, und solche, die größer oder gleich sind. Der KDB-Baum in der Ausprägung (b, t) gliedert sich in zwei Arten von Speicherseiten: sogenannte Bereichsseiten, die als Index fungieren, und Satzseiten, welche die Blattebene darstellen. Innerhalb jeder Bereichsseite findet sich ein eingebetteter KD-Teilbaum, dessen Aufbau maximal b innere Verzweigungsknoten umfasst. Diese Trennpunkte können sich auf verschiedene Attribute stützen und dadurch unterschiedliche Dimensionen der Daten separieren. Die Satzseiten können bis zu t Tupel speichern. Der Traversal beginnt an der Wurzel und durchläuft rekursiv die Schnittattribute in einer zyklischen oder selektivitätsbasierten Reihenfolge. Bei jedem Schnittattribut wird entschieden, ob der Pfad über den linken oder rechten Zeiger fortgesetzt wird, bis schließlich eine Blattseite erreicht wird, die die eigentlichen Datensätze enthält.
Die Abbildung 2.13 zeigt ein Beispiel für einen KDB-Baum, der die Attribute `Zeit` und `Höhe` abwechselnd als Trennkriterien verwendet. Die Wurzel des Baums partitioniert die Daten zunächst nach dem Attribut `Zeit` mit dem Schnittwert 25. Alle Punkte mit einem Zeitwert kleiner als 25 werden in den linken Teilbaum eingeordnet, alle übrigen in den rechten Teilbaum.
Im linken Teilbaum erfolgt in der zweiten Ebene eine Partitionierung nach dem Attribut `Höhe` mit dem Schnittwert 40. Punkte mit einer geringeren Höhe werden im linken Teilbaum weiter nach `Zeit` (Schnittwert 15) unterteilt, wodurch die Punkte (10,30) und (20,35) auf getrennte Blätter verteilt werden.

Punkte mit einer Höhe von mindestens 40 – in diesem Fall der Punkt (15,50) – werden zusätzlich nach `Zeit` mit dem Schnittwert 18 partitioniert, was in diesem Beispiel jedoch nur zu einem einzelnen Blatt führt.

Der rechte Teilbaum (Zeit $\geq$ 25) wird ebenfalls in der zweiten Ebene nach dem Attribut `Höhe` mit dem Schnittwert 40 unterteilt. Im linken Teilbaum dieser Ebene (Höhe $<$ 40) erfolgt eine weitere Aufspaltung nach `Zeit` mit dem Schnittwert 35, was zur Trennung der Punkte (32,25) und (42,20) führt. Analog dazu wird im rechten Teilbaum (Höhe $\geq$ 40) nach `Zeit` mit dem Schnittwert 37 partitioniert, wodurch die Punkte (28,45) und (38,55) auf getrennte Blätter verteilt werden. Dieses Beispiel verdeutlicht die zyklische Nutzung mehrerer Attribute in einem KDB-Baum sowie dessen Fähigkeit, Daten hierarchisch in mehreren Dimensionen zu partitionieren. Die folgende Abbildung 2.13 zeigt beispielhaft den Endzustand eines KDB-Baums mit drei Ebenen und zyklischer Attributnutzung.

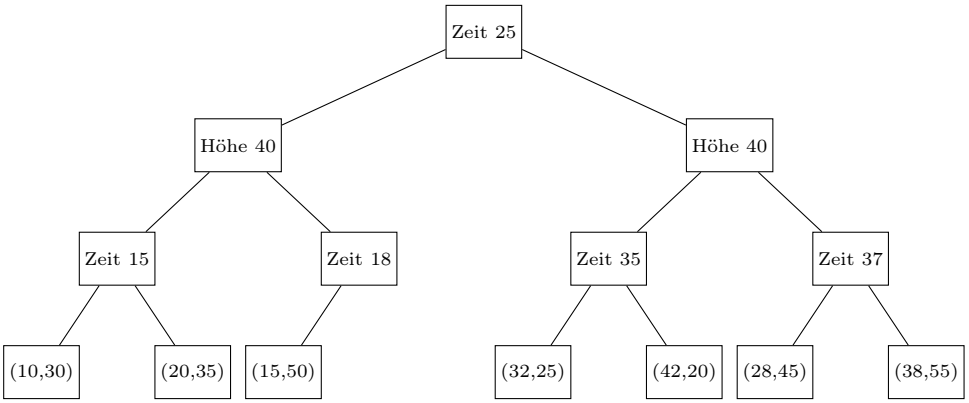


Abbildung 2.13: Endzustand eines KDB-Baums mit drei Ebenen und zyklischer Attributnutzung.

Der KDB-Baum ist mehrdimensional und kann Attribute in beliebiger Reihenfolge als Trennkriterien nutzen. Durch die Kombination mit dem B$^+$-Baum-Prinzip besitzt er eine hohe Fächerung, was ihn für Sekundärspeicher optimiert. Für Punktanfragen ist der KDB-Baum effizient, da ein exakter Schlüsselwert in logarithmischer Zeit gefunden werden kann. Bereichsanfragen profitieren ebenfalls von der hierarchischen Partitionierung, allerdings müssen mehrere Teilbäume durchsucht werden. Dank der Kombination aus der hohen Verzweigungsrate (Fächergrad) des B$^+$-Baums und der mehrdimensionalen Raumaufteilung des KD-Baums, die gezielte Filterung in hochdimensionalen Datensätzen erlaubt, eignet sich der KDB-Baum besonders für Top-N%-Anfragen und komplexe Aggregationen.

Parameter	**KDB-Baum**
Dimension	Mehrdimensional
Balancierung	Balanciert
Punktanfragen	Gut geeignet
Bereichsanfragen	Gut geeignet
Top-N%-Anfragen	Gut geeignet
Aggregationen	Gut geeignet

Tabelle 2.15: Analyseparameter für den KDB-Baum.

Die Zeitkomplexität des KDB-Baums entspricht im exakten Fall der des B-Baums und liegt bei $O(logn)$. Dies gilt für Such-, Einfüge- und Löschoperationen.

Operation	Komplexität (avg)
Einfügen	$O(\log n)$
Suche	$O(\log n)$
Zugriff	$O(\log n)$

Tabelle 2.16: Komplexitätsparameter für den KDB-Baum [HSS19].

Hash

Hashing ist ein grundlegendes Verfahren zur effizienten Speicherung und zum schnellen Zugriff auf Daten. Die zentrale Idee besteht darin, einen Schlüsselwert (zum Beispiel eine Messzeit oder eine Messhöhe) mittels einer sogenannten Hashfunktion in einen festen Adressraum zu transformieren. Diese Hashfunktion berechnet aus dem Schlüssel eine Speicheradresse oder einen Index, unter dem die zugehörigen Daten abgelegt werden. Dadurch entfällt eine aufwändige Suche in sortierten oder unsortierten Datenstrukturen, da der Zugriff direkt über den berechneten Index erfolgt.
Ein zentrales Problem beim Hashing besteht darin, dass mehrere Schlüssel denselben Bucket belegen können, was als *Kollision* bezeichnet wird. Eine verbreitete Strategie zur Behandlung solcher Kollisionen ist die *Verkettung* (*Separate Chaining*) [LC88]. Dabei enthält jeder Bucket einen Zeiger auf eine verkettete Liste, in der alle kollidierenden Einträge gespeichert werden[6]. Diese Methode erlaubt eine flexible Handhabung, da die Listen dynamisch wachsen können. Bei starker Kollision führt sie jedoch zu längeren Zugriffszeiten, da die Elemente innerhalb der Listen sequenziell durchsucht werden müssen.
In dem hier betrachteten Beispiel wird eine Hashfunktion mit zwei Buckets verwendet:

$$h(x) = x \bmod 2$$

Die Schlüsselwerte 2, 4, 6 und 8 werden nacheinander eingefügt. Gemäß der Hashfunktion werden alle Schlüssel dem Bucket mit Index 0 zugeordnet. Zur Veranschaulichung werden die zusätzlichen Einträge (6 und 8) in einem Überlauf-Bucket gespeichert, der mit dem Hauptbucket verknüpft ist. In der folgenden Abbildung 2.14 wird ein vereinfachtes Beispiel dargestellt[7].

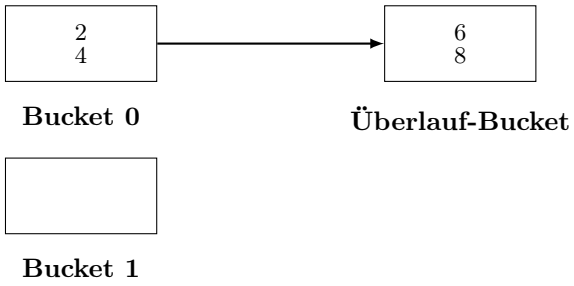


Abbildung 2.14: Vereinfachte Darstellung von Separate Chaining: Hauptbucket mit einem Überlauf-Bucket.

[6]Zur Vereinfachung der Darstellung wird die *verkettete Liste* in dieser Arbeit als *Überlauf-Bucket* (*Overflow Bucket*) mit fester Kapazität visualisiert. In einer realen Implementierung von Separate Chaining handelt es sich jedoch um eine dynamisch wachsende verkettete Liste ohne feste Größenbeschränkung.

[7]In praktischen Anwendungen verteilen Hashfunktionen die Schlüssel in der Regel gleichmäßiger über die Buckets. In diesem Beispiel landen jedoch alle Werte im selben Bucket, um das Prinzip der Verkettung besonders deutlich zu veranschaulichen.

2.6.7 Statisches Hashing

Statisches Hashing ist eine der einfachsten und grundlegendsten Techniken zur Organisation von Daten mittels Hashfunktionen. Dabei wird eine feste Anzahl an Buckets (Seiten) oder Speicherplätzen vorab definiert, und eine Hashfunktion transformiert jeden Schlüssel deterministisch auf einen dieser Buckets. Der zentrale Vorteil besteht darin, dass Such- und Einfügeoperationen im Durchschnitt in konstanter Zeit ausgeführt werden können, da der Speicherort direkt aus dem Schlüssel berechnet wird. Statisches Hashing wird häufig für Datenmengen eingesetzt, deren Größe im Voraus bekannt ist oder sich nur selten verändert. Da die Anzahl der Buckets fix bleibt, führt ein Überlauf der Kapazität nicht zu einer Anpassung der Hashstruktur.

Stattdessen kommen verschiedene Strategien zur Behandlung von Kollisionen zum Einsatz. Eine gängige Variante ist das *lineare Sondieren* [LC88], bei dem im Falle einer Kollision die nächste freie Speicherzelle in linearer Reihenfolge gesucht und belegt wird. Dieses Verfahren ist einfach zu implementieren und benötigt keine zusätzlichen Zeigerstrukturen, kann jedoch bei häufiger Kollision zu längeren Suchketten führen (sogenanntes *Clustering*). Angenommen, wir wollen die Messwerte 2, 4, 6 und 8 speichern. Zu Beginn stehen zwei Buckets (`0` und `1`) zur Verfügung, wobei jeder Bucket maximal zwei Werte aufnehmen kann. Als Hashfunktion wird

$$h(x) = x \bmod 2$$

verwendet. Die Schlüssel 2 und 4 werden zunächst in Bucket 0 einsortiert, da $2 \bmod 2 = 0$ und $4 \bmod 2 = 0$. Da die Kapazität dieses Buckets ausgeschöpft ist, entsteht beim Einfügen des nächsten Wertes (Schlüssel 6) ein Überlauf, weil auch dieser Wert gemäß der Hashfunktion Bucket 0 zugeordnet würde ($6 \bmod 2 = 0$). Beim linearen Sondieren wird in diesem Fall die nächste freie Position in der Tabelle gesucht, sodass Schlüssel 6 in Bucket 1 gespeichert wird. Beim anschließenden Einfügen von Schlüssel 8 tritt dieselbe Situation auf, weshalb auch dieser Wert in Bucket 1 abgelegt wird. Zur Illustration zeigt die folgende Abbildung 2.15 ein einfaches Beispiel[8].

Abbildung 2.15: Speicherlayout beim linearen Sondieren: Überlauf aus Bucket 0 wird in Bucket 1 eingefügt.

Statisches Hashing ist eine eindimensionale Struktur, da die Hashfunktion auf einem einzigen Schlüsselattribut basiert. Die gleichmäßige Verteilung hängt vollständig von der Qualität der Hashfunktion und der Verteilung der Daten ab. Für Punktanfragen ist statisches Hashing besonders gut geeignet, da der Zugriff auf einen Schlüssel durch direkte Berechnung des Buckets sehr schnell ist. Bereichsanfragen hingegen sind ineffizient, da keine Reihenfolge zwischen den Buckets existiert und alle Buckets durchsucht werden müssen. Top-N%-Abfragen lassen sich ebenfalls nur schwer umsetzen, da eine globale Sortierung fehlt. Aggregationen wie Summen oder Mittelwerte erfordern ebenfalls das vollständige Durchlaufen aller Buckets, da die Daten nicht geordnet vorliegen.

[8] In der Praxis verteilen geeignete Hashfunktionen die Schlüssel in der Regel gleichmäßiger auf mehrere Buckets. Hier landen jedoch alle Werte zunächst im selben Bucket, um den Ablauf des linearen Sondierens besonders klar darzustellen.

Parameter	Statisches Hashing
Dimension	Eindimensional
Punktanfragen	Gut geeignet
Bereichsanfragen	Nicht geeignet
Top-N%-Anfragen	Nicht geeignet
Aggregationen	Nicht geeignet

Tabelle 2.17: Analyseparameter für statisches Hashing.

Die Komplexität von Static Hashing ist im Durchschnitt für Einfüge-, Such- und Zugriffsvorgänge konstant, also $O(1)$. Dies liegt daran, dass die Hashfunktion $h(k)$ den Ziel-Bucket direkt identifiziert und dieser in der Regel nur eine primäre Seite enthält, wodurch oft nur ein einziger Plattenzugriff (I/O) erforderlich ist. Im Idealfall können daher sowohl Suche als auch Einfügen und Zugriff in konstanter Zeit erfolgen. Im Worst Case jedoch, wenn durch starkes Datenwachstum lange Overflow-Ketten entstehen, müssen alle Overflow-Seiten des betroffenen Buckets sequenziell durchsucht werden. Dies kann zu einer linearen Laufzeit von $O(n)$ führen, da im Extremfall fast alle Datensätze in einer Overflow-Kette liegen können. Die Performance hängt somit stark von der Verteilung der Schlüssel und der Auslastung der Buckets ab [RG03].

Operation	Komplexität (avg / worst)
Einfügen	$O(1)$ / $O(n)$
Suche	$O(1)$ / $O(n)$
Zugriff	$O(1)$ / $O(n)$

Tabelle 2.18: Komplexitätsparameter für Statisches Hashing [RG03].

2.6.8 Lineares Hashing

Lineares Hashing ist ein dynamisches Hashverfahren, das entwickelt wurde, um eine schrittweise Erweiterung der Speicherstruktur ohne vollständiges Rehashing zu ermöglichen. Im Gegensatz zum statischen Hashing, bei dem die Anzahl der Buckets(seiten) und die Hashfunktion fest vorgegeben sind, erlaubt das lineare Hashing eine kontrollierte Vergrößerung des Adressraums. Die zentrale Idee besteht darin, mit einer anfänglichen Anzahl von Buckets zu starten, die sukzessive erweitert wird, sobald einzelne Buckets ihre maximale Kapazität erreichen. Diese maximale Anzahl von Einträgen pro Bucket wird als Bucket-Kapazität bezeichnet und bleibt während des gesamten Verfahrens konstant, während die Anzahl der Buckets dynamisch ansteigt. Das lineare Hashen basiert auf einer Familie von Hashfunktionen, die durch das sogenannte Level i charakterisiert wird. Diese Familie lässt sich durch die folgende allgemeine Funktion darstellen:

$$h_i(x) = x \bmod \left(2^i \cdot N\right)$$

Hierbei bezeichnet N die anfängliche Anzahl der Buckets, während i das aktuelle Level repräsentiert, das sich nach einer vollständigen Runde von Splits um eins erhöht. Ein zentrales Steuerungselement dieses Verfahrens ist der sogenannte *Split-Zeiger*. Dieser zeigt stets auf den Bucket, der als Nächstes aufgeteilt wird, sobald eine Überlastung auftritt und zusätzlicher Speicher erforderlich ist. Wird das Ende der aktuellen Bucket-Liste erreicht, so wird der Split-Zeiger wieder auf null zurückgesetzt. Bei jeder Split-Operation wird der Split-Zeiger auf das nächste Bucket gesetzt, das Level der Hashfunktion für dieses Bucket um eins erhöht und ein zusätzlicher Bucket angelegt. Anschließend werden Teile der zuvor gespeicherten Schlüssel anhand der neuen Hashfunktion neu verteilt [HSS19]. Ein einfaches Beispiel verdeutlicht dieses Prinzip: Ausgangspunkt sind zwei Buckets mit einer Bucket-Kapazität von zwei Einträgen. Zu Beginn wird die Hashfunktion

$$h_0(x) = x \bmod 2$$

verwendet, da anfangs nur zwei Buckets (0 und 1) existieren. Das aktuelle Level der Hashfunktion ist also $i = 0$, und der *Split-Zeiger* steht zunächst auf Bucket 0. Wir fügen nun die Schlüssel 2 und 4 ein. Beide Werte ergeben unter $h_0(x)$ den Wert 0 und werden daher in Bucket 0 abgelegt. Nach dieser Einfügung ist die Kapazität von Bucket 0 erreicht, sodass ein Split ausgelöst wird. Dabei zeigt der Split-Zeiger auf Bucket 0, welcher nun aufgeteilt wird. Durch den Split wird ein neuer Bucket (Bucket 2) angelegt und die Hashfunktion für die neu gesplitteten Buckets erweitert, sodass nun

$$h_1(x) = x \bmod 4$$

verwendet wird. Nach dem Split zeigt der Split-Zeiger auf den nächsten Bucket in der Reihenfolge, also auf Bucket 1.Mit dieser neuen Hashfunktion werden die bestehenden und zukünftigen Schlüssel neu verteilt. Die Schlüssel 2 und 6 werden dabei in Bucket 2 einsortiert (da $2 \bmod 4 = 2$ und $6 \bmod 4 = 2$), während Schlüssel 4 in Bucket 0 bleibt und Schlüssel 8 in Bucket 0 einsortiert wird ($8 \bmod 4 = 0$). Das Level der Hashfunktion beträgt nach dem Split weiterhin $i = 1$, und der Split-Zeiger verweist nun auf Bucket 1, der als Nächstes gesplittet würde, wenn ein weiterer Überlauf auftritt. Die folgende Abbildung 2.16 zeigt den Endzustand nach allen Splits, inklusive der Position des Split-Zeigers und des aktuellen Levels der Hashfunktion:

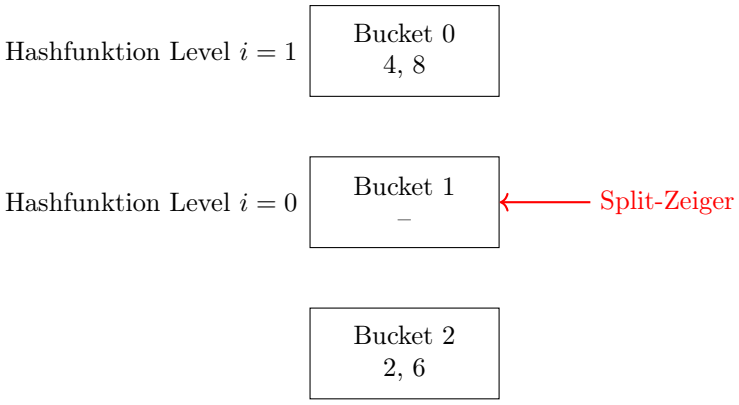


Abbildung 2.16: Darstellung der Buckets mit Hashfunktion Level und Position des Split-Zeigers.

Linear Hashing ist eine eindimensionale Struktur, die ihre Kapazität durch lineare Splits erweitert. Durch die sukzessive Erweiterung wird eine gleichmäßige Verteilung der Schlüssel erreicht. Punktanfragen sind sehr effizient und können in konstanter Zeit durchgeführt werden. Bereichsanfragen sind wie bei anderen Hashverfahren ineffizient, da keine natürliche Ordnung zwischen den Buckets besteht. Top-N%-Abfragen erfordern ein vollständiges Scannen und Sortieren der Daten, ebenso Aggregationen wie Summen oder Mittelwerte.

Parameter	**Lineares Hashing**
Dimension	Eindimensional
Punktanfragen	Gut geeignet
Bereichsanfragen	Nicht geeignet
Top-N%-Anfragen	Nicht geeignet
Aggregationen	Nicht geeignet

Tabelle 2.19: Analyseparameter für Lineares Hashing.

Bei Linear Hashing kostet eine Gleichheitsselektion (*equality selection*) im Durchschnitt nur einen einzelnen Plattenzugriff (I/O), sofern der Bucket keine Overflow-Seiten enthält. In der Praxis liegt der Durchschnittswert bei etwa 1,2 Plattenzugriffen für gleichmäßig verteilte Daten. Im Worst Case kann die Komplexität jedoch linear in der Anzahl der Datensätze $O(n)$ werden, wenn die Datenverteilung stark unausgeglichen (*skewed*) ist und lange Overflow-Ketten entstehen. Einfügeoperationen erfordern im Durchschnitt das Lesen und Schreiben einer einzelnen Seite, sofern kein Split ausgelöst wird [RG03].

Operation	**Komplexität (avg / worst)**
Einfügen	$O(1)$ / $O(n)$
Suche	$O(1)$ / $O(n)$
Zugriff	$O(1)$ / $O(n)$

Tabelle 2.20: Komplexitätsparameter für Lineares Hashing [RG03].

2.6.9 Erweiterbares Hashing

Nachdem das Prinzip des Linearen Hashings erläutert wurde, soll nun das Erweiterbares Hashing vorgestellt werden, das ebenfalls ein dynamisches Hashverfahren darstellt, jedoch eine andere Strategie zur Erweiterung des Adressraums verfolgt. Im Gegensatz zum Linearen Hashing, bei dem Buckets sukzessive und in fester Reihenfolge gesplittet werden, verwendet Extendible Hashing ein Directory, dessen Größe sich bei Bedarf verdoppelt. Dieses Directory enthält Zeiger auf die Buckets und wird mithilfe einer Hashfunktion adressiert, deren Präfixlänge durch die Directory-Tiefe i bestimmt wird.

Ein wesentlicher Unterschied liegt darin, dass nicht alle Buckets linear durchlaufen werden müssen; stattdessen kann durch Verdopplung des Directories ein einzelner Bucket mehrfach referenziert werden. Dadurch können Überläufe gezielt durch lokale Splits behoben werden, ohne dass alle anderen Buckets betroffen sind [HSS19]. Zur Verdeutlichung der Unterschiede wird auch hier das Beispiel mit den Schlüsseln 2, 4, 6 und 8 bei einer Bucket-Kapazität von zwei Einträgen verwendet. Anfangs wird die Hashfunktion

$$h_0(x) = x \bmod 2$$

genutzt. Nach dem Füllen des ersten Buckets mit 2 und 4 wird das Directory verdoppelt und die Hashfunktion auf

$$h_1(x) = x \bmod 4$$

erweitert. Die bestehenden und neuen Schlüssel werden anschließend nach der aktualisierten Funktion verteilt. Das Endergebnis ist in der folgenden Abbildung 2.17 dargestellt. Die Directory-Einträge sind binär nummeriert, während die Buckets selbst mit fortlaufenden Zahlen bezeichnet sind.

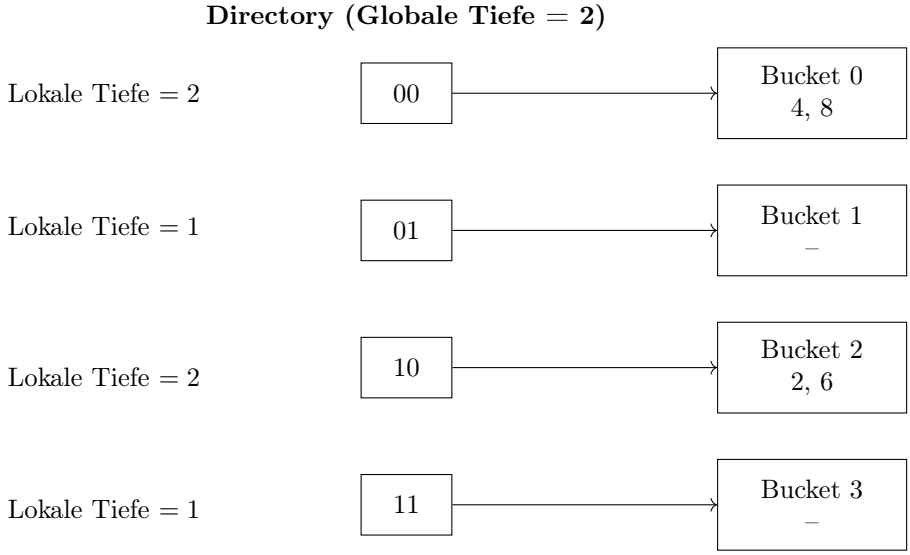


Abbildung 2.17: Endzustand des Erweiterbaren Hashings: Globale Tiefe = 2 und lokale Tiefe = 1 für alle Buckets.

Erweiterbares Hashing ist eine eindimensionale Struktur, da sie nach einem einzigen Schlüssel arbeitet. Durch die Verwendung des Directories wird eine gleichmäßige Verteilung der Schlüssel über die Buckets erreicht, sofern die Hashfunktion gut gewählt ist. Punktanfragen sind sehr effizient, da die Suche direkt über die Bits des Hashwerts erfolgt und in $O(1)$ durchgeführt werden kann. Bereichsanfragen sind jedoch ineffizient, da die Daten nicht sortiert vorliegen und alle Buckets durchsucht werden müssen. Top-N%-Abfragen sind ebenfalls schwierig, da keine Ordnung existiert; sie erfordern ein vollständiges Scannen. Aggregationen wie Summen oder Mittelwerte benötigen ebenso einen vollständigen Durchlauf aller Buckets.

Parameter	**Erweiterbares Hashing**
Dimension	Eindimensional
Punktanfragen	Gut geeignet
Bereichsanfragen	Nicht geeignet
Top-N%-Anfragen	Nicht geeignet
Aggregationen	Nicht geeignet

Tabelle 2.21: Analyseparameter für Erweiterbares Hashing.

Die Komplexität des Erweiterbaren Hashings ist im Durchschnitt für Einfüge-, Such- und Zugriffsvorgänge konstant, also $O(1)$. Dies ist darauf zurückzuführen, dass die Hashfunktion $h(k)$ den Ziel-Bucket direkt identifiziert und der Zugriff in der Regel über das Verzeichnis (Directory) nur einen Plattenzugriff (I/O) erfordert. Befindet sich das Verzeichnis vollständig im Hauptspeicher, ist selbst der Zugriff auf das Directory kostenfrei im Sinne der I/O-Zeit, was die Leistung auf das Niveau von Static Hashing hebt. Im Worst Case kann die Komplexität jedoch auf $O(n)$ ansteigen, etwa wenn sehr viele Datensätze denselben Hashwert besitzen und dadurch Overflow-Seiten benötigt werden, oder wenn stark schiefe (skewed) Datenverteilungen zu einer ungleichmäßigen Belegung der Buckets führen. Zusätzlich können aufwendige Operationen wie das Directory Doubling beim Splitten eines Buckets zu höheren Kosten führen, wobei hier nur die betroffenen Buckets und das Directory angepasst werden, anstatt das gesamte File umzustrukturieren. Somit bietet das erweiterbare Hashing im Vergleich zum statischen Hashing eine flexiblere Speicherverwaltung mit stabiler Durchschnittsleistung, leidet aber in Extremsituationen unter denselben

linearen Laufzeitproblemen [RG03].

Operation	Komplexität (avg / worst)
Einfügen	$O(1)$ / $O(n)$
Suche	$O(1)$ / $O(n)$
Zugriff	$O(1)$ / $O(n)$

Tabelle 2.22: Komplexitätsmaße für das erweiterbare Hashing [RG03].

2.6.10 Bitmap-Index

Ein Bitmap-Index stellt eine spezielle Form von Indexstrukturen dar, die insbesondere in spaltenorientierten Datenbanksystemen und Data-Warehouse-Szenarien von Bedeutung ist. Er eignet sich vor allem für Attribute mit geringer Kardinalität bzw. niedriger Selektivität, bei denen klassische Indexierungsverfahren wie B-Bäume an Effizienz verlieren. Anstelle eines direkten Verweises auf jeden einzelnen Datensatz wird für jeden möglichen Attributwert ein eigener Bitvektor gebildet. Dieser Bitvektor markiert mit gesetzten Bits (1) alle Tupel, die den entsprechenden Attributwert besitzen, und mit nicht gesetzten Bits (0) diejenigen, bei denen dies nicht der Fall ist. Durch diese Art der Darstellung lassen sich sehr effiziente bitweise Operationen wie AND, OR oder NOT durchführen [HSS19].

Ein einfaches Beispiel verdeutlicht das Prinzip: Angenommen, wir haben eine kleine Tabelle mit fünf Datensätzen, die ein Attribut *Farbe* enthalten. Die möglichen Werte sind `Rot`, `Grün` und `Blau`. Die Daten sehen wie folgt aus:

ID	Farbe	Bitmap Rot	Bitmap Grün	Bitmap Blau
1	Rot	1	0	0
2	Blau	0	0	1
3	Grün	0	1	0
4	Rot	1	0	0
5	Blau	0	0	1

Tabelle 2.23: Beispielhafte Bitmap-Darstellung für Farbattribute.

Eine Anfrage wie „alle Datensätze, die *Rot* oder *Blau* sind“ kann durch eine bitweise OR-Operation der beiden entsprechenden Bitmaps beantwortet werden:

$$(1\ 0\ 0\ 1\ 0)\ \text{OR}\ (0\ 1\ 0\ 0\ 1) = 1\ 1\ 0\ 1\ 1$$

Die Analyse dieser Struktur zeigt, dass der Bitmap-Index eindimensional ist, da er sich nur auf ein einzelnes Attribut bezieht. Für Punktanfragen eignet sich der Bitmap-Index sehr gut, weil das Vorhandensein eines Wertes direkt durch Ablesen des entsprechenden Bits überprüft werden kann. Bereichsanfragen lassen sich durch Kombination mehrerer Bitmaps ebenfalls effizient beantworten, besonders wenn der Wertebereich klein ist. Top-N%-Anfragen sind nur eingeschränkt unterstützt, da hierfür zusätzliche Informationen wie Zählwerte oder externe Sortierungen erforderlich sind. Aggregationen wie Summen oder Mittelwerte profitieren hingegen von der Möglichkeit, mehrere Bitmaps gleichzeitig mit bitweisen Operationen auszuwerten, was zu einer hohen Abfragegeschwindigkeit führen kann.

Parameter	**Bitmap-Index**
Dimension	Eindimensional
Punktanfragen	Gut geeignet
Bereichsanfragen	Eingeschränkt geeignet
Top-N%-Anfragen	Eingeschränkt geeignet
Aggregationen	Gut geeignet durch bitweise Operationen

Tabelle 2.24: Analyseparameter für den Bitmap-Index.

Der Zugriff auf einen Bitvektor erfolgt in $O(1)$, da die Bit-Codierung einer Knoten-ID über Hash-Tabellen bezogen wird und somit ohne zusätzlichen Speicheraufwand direkt verfügbar ist. Die Suche benötigt $O(\log n)$, da nach dem Laden der beiden Bitvektoren in $O(1)$ eine bitweise AND-Operation über Vektoren der Länge $O(\log n)$ durchgeführt wird. Während der Aufbau der gesamten BIT-Struktur eine Zeitkomplexität von $O(n \log n)$ aufweist, lässt sich für das Einfügen oder Aktualisieren einzelner Elemente eine Komplexität von $O(\log n)$ ableiten, da lediglich die zugehörigen Bitvektoren angepasst werden müssen. [HCZS24].

Operation	**Komplexität (avg)**
Einfügen	$O(\log n)$
Suche	$O(\log n)$
Zugriff	$O(1)$

Tabelle 2.25: Komplexitäten für Einfügen, Suche und Zugriff beim Bitmap-Index [HCZS24].

2.6.11 Grid-File

Das Grid-File ist eine multidimensionale Indexstruktur, die entwickelt wurde, um Datenzugriffe in mehreren Dimensionen effizient zu unterstützen. Es kombiniert Konzepte von Hashing und Bereichspartitionierung und teilt den Datenraum in ein gleichmäßiges Gitter aus rechteckigen Zellen auf. Jede Dimension des Datenraums wird durch eine Skala repräsentiert, die in Intervalle unterteilt ist. Die Kreuzung dieser Intervalle definiert die sogenannten Grid-Zellen, die jeweils auf einen Datenbereich oder eine Bucket-Seite verweisen. Das zentrale Verzeichnis, das sogenannte Grid-Directory, ordnet jeder Zellkombination einen Zeiger zu. Dieses Verzeichnis kann dynamisch wachsen, indem neue Intervalle hinzugefügt werden, wenn eine bestimmte Region des Datenraums überfüllt ist. In der folgenden Abbildung 2.18 sind die Bestandteile eines Grids dargestellt [HSS19].

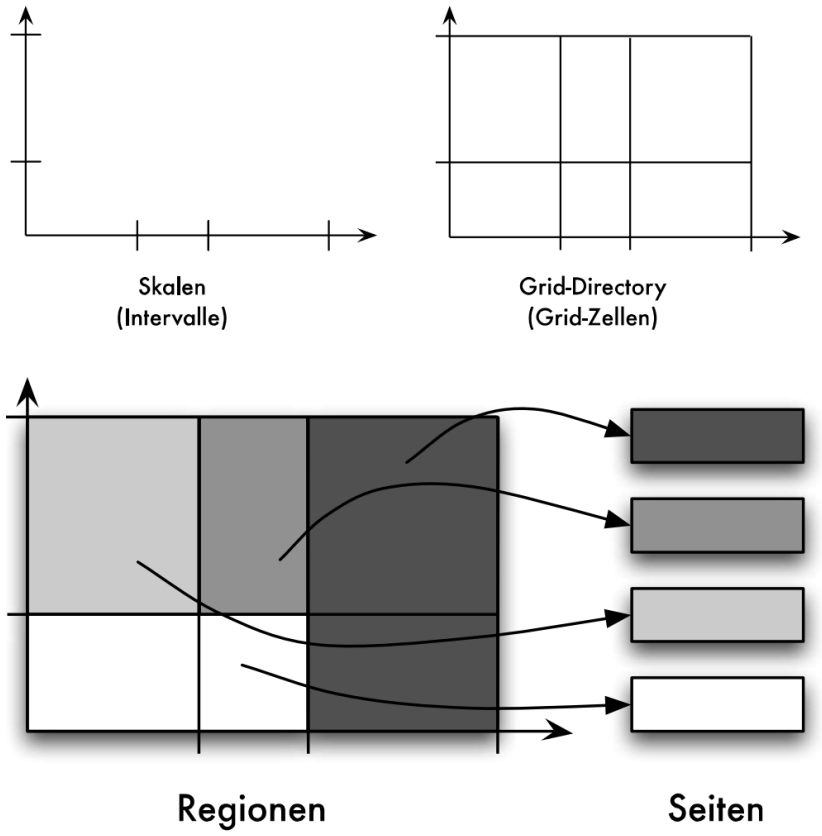


Abbildung 2.18: Bestandteile eines Grid-Files [HSS19].

Ein vereinfachtes Beispiel verdeutlicht die Funktionsweise eines Grid-Files. Insgesamt wollen wir nun die folgenden Datensätze in das Grid-File einfügen:

A1 (Werte)	**A2 (Buchstaben)**
45	D
02	B
87	S
75	M

Tabelle 2.26: Darstellung der vier Datensätze mit den Attributen A1 und A2

Somit ergeben sich die vier Datenpunkte $(45, D)$, $(02, B)$, $(87, S)$ und $(75, M)$. Es wird angenommen, dass jede Seite des Grid-Files maximal drei Datensätze aufnehmen kann. Die ersten drei Datenpunkte $(45, D)$, $(02, B)$ und $(87, S)$ können ohne Probleme in die anfänglich einzige vorhandene Grid-Region eingefügt werden. Beim Einfügen des vierten Datensatzes $(75, M)$ tritt jedoch ein Überlauf auf, da die Kapazitätsgrenze von drei Elementen überschritten wird. Um diesen Überlauf zu beheben, wird die Skala des Attributs A_1 (Zahlenachse) in zwei Intervalle unterteilt. Die Intervallgrenze wird dabei als arithmetisches Mittel der bisher gespeicherten Werte 45, 02 und 87 berechnet und liegt folglich bei 60. Zur Veranschaulichung wird hier ein Wertebereich von 0 bis 100 angenommen. Dadurch entstehen die Intervalle $[0{-}60]$ und $[61{-}99]$. Die Datenpunkte werden anschließend gemäß dieser neuen Aufteilung neu verteilt: Die Werte

$(45, D)$ und $(02, B)$ fallen in das Intervall [0−60], während die Werte $(87, S)$ und $(75, M)$ dem Intervall [61−99] zugeordnet werden. Auf diese Weise entstehen zwei Grid-Zellen, die jeweils einer separaten Seite im Grid-File entsprechen. Der resultierende Zustand ist in der folgenden Abbildung 2.19 dargestellt.

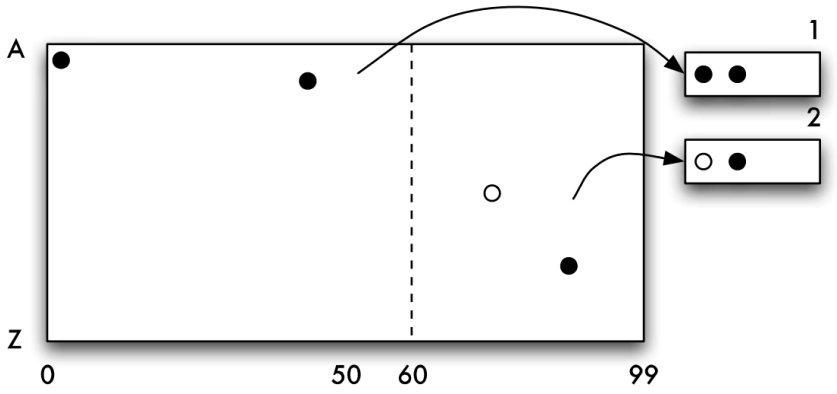


Abbildung 2.19: Grid-File nach dem ersten Split [HSS19].

Die Analyse dieser Struktur zeigt, dass das Grid-File multidimensional ist und besonders für Punkt- und Bereichsanfragen geeignet ist. Da die Daten in einem gleichmäßigen Raster organisiert sind, lassen sich Bereichsanfragen effizient beantworten, indem nur die relevanten Zellen betrachtet werden. Top-N%-Anfragen können ebenfalls unterstützt werden, erfordern jedoch zusätzliches Scannen der Zellen. Aggregationen sind mit Grid Files nur eingeschränkt effizient durchführbar. Zwar können Operationen seitenweise über die relevanten Buckets erfolgen, jedoch fehlt eine interne Ordnung der Daten, sodass alle Einträge vollständig durchsucht werden müssen – insbesondere dann, wenn der Bereich der gesuchten Daten nicht eindeutig definiert ist.

Parameter	**Grid-File**
Dimension	Mehrdimensional
Punktanfragen	Gut geeignet
Bereichsanfragen	Gut geeignet
Top-N%-Anfragen	Eingeschränkt
Aggregationen	Eingeschränkt

Tabelle 2.27: Analyseparameter für das Grid-File.

Die Komplexitäten des Grid File erklären sich wie folgt: Exakte Zugriffe benötigen höchstens zwei Plattenzugriffe (einmal für das Verzeichnis und einmal für die Datenseite). Bereichs- und Teilanfragen durchsuchen nur die vom Suchbereich geschnittenen Zellen, sodass die Kosten bei $O(b)$ mit b betroffenen Seiten liegen. Einfügungen sind in der Regel konstant, lediglich selten ist eine Verzeichnisreorganisation erforderlich, weshalb die Kosten amortisiert mit $O(1)$ angegeben werden [NHS84].

Operation	**Komplexität (avg)**
Einfügen	$O(1)$
Suche	$O(b)$
Zugriff	$O(1)$

Tabelle 2.28: Komplexitäten für Einfügen, Suche und Zugriff beim Grid-File.

Zusammenfassend werden in diesem Kapitel 2 die zentralen Grundlagen für diese Arbeit dargestellt. Neben einer Einführung in das HDF5-Format mit seinem Aufbau, seinen Vorteilen und typischen Einsatzszenarien wurden relevante Anfragearten sowie maßgebliche Komplexitätsmaße erläutert. Darauf aufbauend erfolgte eine systematische Übersicht über gängige Zugriffsstrukturen, darunter verschiedene Baumvarianten (AVL-, B-, B^+- und KD-Bäume), der KDB-Baum sowie unterschiedliche Hash- und Indexierungsverfahren. Damit ist das Fundament gelegt, um im folgenden Kapitel 3 die ausgewählten Strukturen vergleichend zu bewerten, die Wahl des B^+-Baums als Indexstruktur zu begründen und dessen Integration – sowohl extern als auch intern – detailliert zu konzipieren.

Kapitel 3

Konzept

In diesem Kapitel wird das Konzept zur Auswahl geeigneter Zugriffsstrukturen für die im Projekt verwendeten Messdaten vorgestellt. Nach dem Aufbau in den vorherigen Kapiteln sollen die Strukturen hier nun den Anforderungen entsprechend verglichen werden, um eine Entscheidung über die am besten geeignete konzeptionelle Implementierung treffen zu können.

3.1 Zusammenfassende Bewertung der Strukturen

Die folgende Tabelle fasst die untersuchten Strukturen zusammen und bewertet deren Eignung hinsichtlich der im Projekt relevanten Abfragen. Dabei wird deutlich, dass sich einige Strukturen klar voneinander abgrenzen: Während Hashing-Varianten nur für einfache Punktanfragen geeignet sind, zeigen sich deutliche Einschränkungen bei Bereichs- und Aggregationsanfragen. Dagegen liefern insbesondere der B^+-Baum und der KDB-Baum eine konsistente Leistung über alle betrachteten Anfragearten hinweg und erscheinen somit besonders vielversprechend für den Einsatz im Projekt. Diese Übersicht bildet die Grundlage für die detaillierte Diskussion in den folgenden Abschnitten, in denen die ausgewählten Strukturen näher untersucht und praktisch implementiert werden.

Struktur	**Dimension**	**Punkt-anfragen**	**Bereichs-anfragen**	**Top-N%-anfragen**	**Aggregation**
BST	1D	Eingeschränkt	Eingeschränkt	Eingeschränkt	Eingeschränkt
AVL-Baum	1D	Gut	Gut	Eingeschränkt	Eingeschränkt
B-Baum	1D	Gut	Gut	Gut	Eingeschränkt
B^+-Baum	1D	Gut	Sehr gut	Sehr gut	Gut
KD-Baum	KD	Gut	Gut	Eingeschränkt	Eingeschränkt
KDB-Baum	KD	Gut	Gut	Gut	Gut
Statisches Hashing	1D	Gut	Nicht geeignet	Nicht geeignet	Nicht geeignet
Lineares Hashing	1D	Gut	Nicht geeignet	Nicht geeignet	Nicht geeignet
Erweiterbares Hashing	1D	Gut	Nicht geeignet	Nicht geeignet	Nicht geeignet
Bitmap-Index	1D	Gut	Eingeschränkt	Eingeschränkt	Gut
Grid-File	KD	Gut	Gut	Eingeschränkt	Eingeschränkt

Tabelle 3.1: Vergleich der untersuchten Zugriffsstrukturen nach Dimension und Eignung für verschiedene Abfragen.

Wie aus der vorherigen Tabelle 3.1 hervorgeht, eignen sich nahezu alle betrachteten Strukturen mit Ausnahme des Binären Suchbaums gut für Punktanfragen. Deutlich wird jedoch, dass Hash-basierte Verfahren zwar in diesem Szenario überzeugen, ihre Stärken aber ausschließlich auf exakte Schlüsselzugriffe beschränkt bleiben und sie bei komplexeren Abfragearten erheblich an Leistungsfähigkeit einbüßen. Bitmap-Indizes und Grid-Files zeigen ebenfalls Einschränkungen, insbesondere im Hinblick auf Top-N%-Anfragen, da sie für diese Form der Selektion nicht optimiert sind. Für den Kontext dieser Arbeit war daher eine detaillierte Gegenüberstellung insbesondere zwischen dem B-Baum, dem B$^+$-Baum sowie dem KDB-Baum erforderlich, da nur diese Strukturen sowohl für Bereichsanfragen als auch für Top-N%-Anfragen eine hohe Eignung aufweisen – zwei Abfragearten, die für das vorliegende Projekt von zentraler Bedeutung sind.

3.2 Begründung der Wahl des B$^+$-Baums als Indexstruktur

Der B$^+$-Baum stellt im Rahmen dieser Arbeit die besten geeignete Struktur dar, da er im Vergleich zu alternativen Ansätzen wie dem B-Baum oder dem KDB-Baum eine entscheidende Zusatzfunktionalität bietet: Alle Blätter sind durch eine verkettete Liste miteinander verbunden. Diese Eigenschaft erlaubt nicht nur eine effiziente Bearbeitung von Bereichsanfragen, sondern macht auch die Durchführung von Top-N%-Abfragen besonders leistungsfähig, da nach dem Auffinden des Startpunkts die gewünschten Werte sequenziell und ohne zusätzliche Suchoperationen gelesen werden können.
Während der B-Baum und auch der KDB-Baum bei Punktanfragen eine vergleichbare Effizienz aufweisen, müssen sie für Bereichs- oder Top-N%-Abfragen stets die Baumstruktur durchlaufen und jeden relevanten Teilbaum einzeln erreichen. Dies erfordert ein wiederholtes Navigieren über innere Knoten und ist damit deutlich weniger effizient als die direkte Traversierung der verketteten Blattebene im B$^+$-Baum. Der B$^+$-Baum hingegen vereint die Vorteile balancierter Suchbäume mit einer optimierten Traversierung der Blattebene und bietet damit die für dieses Projekt zentrale Grundlage zur Beschleunigung von Bereichs- und Top-N%-Abfragen. Diese Eigenschaften begründen die Wahl des B$^+$-Baums als Indexstruktur.

3.3 Art der Integration von Indexstrukturen

Die Frage, wie Indexstrukturen in ein Datensystem integriert werden, ist von zentraler Bedeutung für Effizienz, Speicherorganisation und Wartbarkeit. Je nachdem, ob die Indizes innerhalb derselben Datenquelle wie die Rohdaten oder in einem separaten System verwaltet werden, ergeben sich unterschiedliche Vor- und Nachteile. Eine geeignete Integrationsstrategie beeinflusst maßgeblich, wie schnell Abfragen beantwortet werden können, wie einfach sich die Datenbank pflegen lässt und welche Flexibilität für zukünftige Erweiterungen besteht. Grundsätzlich lassen sich dabei zwei Ansätze unterscheiden: die interne und die externe Indexierung.
Bei der internen Indexierung werden die Indexstrukturen direkt innerhalb der HDF5-Datei gespeichert. Der Vorteil dieses Ansatzes liegt darin, dass Index und Daten stets gemeinsam abgelegt sind und somit keine Inkonsistenzen entstehen können. Zudem muss lediglich eine Datei verwaltet werden, was die Organisation erleichtert. Ein Nachteil ist jedoch, dass die ohnehin großen Messdateien durch die zusätzlichen Indexinformationen weiter anwachsen. Darüber hinaus wird die gleichzeitige Pflege unterschiedlicher Indextypen erschwert. Befinden sich die HDF5-Dateien auf langsamen Speichermedien (wie im vorliegenden Fall), so wird jede Anfrage durch diese langsamen Zugriffszeiten belastet. Zusätzlich muss bei jeder Anfrage auf die großen HDF5-Dateien zugegriffen werden, um an die darin eingebetteten Indizes zu gelangen.
Bei der externen Indexierung hingegen werden die Indizes in separaten Dateien gespeichert, beispielsweise in einer Datenbank. Der Vorteil dieses Ansatzes besteht darin, dass die Originaldaten unverändert bleiben und die Indizes flexibel auf schnellen Speichermedien abgelegt werden können. Dadurch lassen sich viele Anfragen deutlich schneller beantworten, ohne dass die großen Messdateien direkt eingelesen werden müssen. Zudem erlaubt dieser Ansatz die parallele Erstellung mehrerer spezialisierter Indizes für

unterschiedliche Anfragearten. Nachteile bestehen in der Notwendigkeit, die Indizes regelmäßig zu synchronisieren und zu aktualisieren. Außerdem wird die Anfrageverarbeitung komplexer, wenn ein Zugriff auf die eigentlichen HDF5-Daten nach einer Indexabfrage erforderlich ist.
Für das vorliegende Projekt gilt, dass die HDF5-Dateien einmalig erzeugt und anschließend unverändert gespeichert werden. Dies stellt sowohl für die interne als auch für die externe Indexierung einen Vorteil dar, da der Aktualisierungsaufwand vollständig entfällt. Allerdings sind die Dateien so organisiert, dass jede Datei lediglich eine Messminute umfasst (ca. 200 MB). Für Anfragen, die sich nur auf eine einzelne Minute beziehen (z. B. die Ermittlung der größten 100 Werte innerhalb einer Minute), wäre eine interne Indexierung in Form einer sortierten Struktur effizient. Sobald sich die Anfrage jedoch über mehrere Minuten erstreckt – was in der Praxis der Regelfall ist – müssten mehrere Dateien gleichzeitig durchsucht und deren Werte aufwändig zusammengeführt werden. Dies führt insbesondere bei der gegebenen, nicht optimalen Speicherumgebung zu erheblichen Leistungseinbußen.

3.4 Externe Indexierung

Wie bereits im vorherigen Abschnitt beschrieben, werden bei der externen Indexierung die HDF5-Dateien weiterhin als alleinige Quelle der Rohdaten beibehalten, während außerhalb davon ergänzende Strukturen aufgebaut werden, um das Auffinden von Datensätzen und die Ausführung von Anfragen zu beschleunigen.

3.4.1 Szenarien der externen Indexierung

Ein erstes Szenario bestünde darin, alle Daten aus den HDF5-Dateien vollständig in eine externe Datenbank zu kopieren. Dabei würden sämtliche Spalten und Zeilen übertragen und die Anfragen ausschließlich in dieser Datenbank ausgeführt. In diesem Fall handelt es sich jedoch nicht mehr um eine externe Indexierung im eigentlichen Sinne, da HDF5 praktisch durch ein neues System ersetzt würde. Dies stellt somit keinen Ansatz für unser Projekt dar.
Ein zweites Szenario umfasst die Übernahme nur der inhaltsbasierten Attribute in eine externe Tabelle, beispielsweise `time`, `row_max_peak`, `row_min_background`, `row_max_mean` sowie `RMSD`. Diese Attribute werden im externen Index abgelegt und mit passenden Indexstrukturen versehen, sodass die Anfragen vollständig in der externen Struktur beantwortet werden können, ohne die HDF5-Dateien erneut öffnen zu müssen. Damit entsteht eine schlanke Attributtabelle, die zentrale Anfragen wie `Top-10% Peak`, `Bottom-10% Background` oder `Bereich-Abfragen auf Mean` effizient abdeckt. Der Vorteil liegt in einer erheblich besseren Performance, da die Datenbank klein bleibt und die Operationen (Filtern, Sortieren, Aggregieren) lokal erfolgen. Der Nachteil ist eine partielle Redundanz, da eine verkleinerte Kopie bestimmter Attribute entsteht.
Ein drittes Szenario schließlich sieht einen reinen Zeigerindex vor, in dem keine Werte wie `row_max_peak`, `row_max_mean` oder `row_min_background` gespeichert werden, sondern lediglich Referenzen wie `filename`, `row_idx` und `time`. Gegebenenfalls können zusätzliche Metadaten oder statistische Kennzahlen ergänzt werden. Dieses Vorgehen ist sehr speichereffizient, da keine Werte dupliziert werden und HDF5 die alleinige Quelle bleibt. Der Nachteil besteht allerdings darin, dass jede praxisrelevante Anfrage zweistufig ablaufen muss: Zunächst werden im externen Index die Referenzen ermittelt, anschließend müssen die eigentlichen Werte aus den HDF5-Dateien geladen und weiterverarbeitet werden. Dies erschwert die Abfrageverarbeitung und reduziert den Geschwindigkeitsvorteil erheblich. Besonders Anfragen mit breiten Zeitintervallen können dazu führen, dass große Teile der HDF5-Dateien erneut geladen werden müssen, wodurch der Nutzen der externen Indexierung verloren geht. Darüber hinaus lassen sich Top-N-Abfragen nur eingeschränkt realisieren, da die Werte nicht im Index vorliegen und erst durch zusätzliche Leseoperationen aus HDF5 gewonnen werden können. Im folgenden Vergleich werden die beiden relevanten Ansätze der externen Indexierung, nämlich Szenario 2 (Attributtabelle) und Szenario 3 (Zeigerindex), gegenübergestellt, um ihre jeweiligen Stärken und Schwächen klar herauszuarbeiten.

Aspekt	Szenario 2: Attributtabelle	Szenario 3: Zeigerindex
Gespeicherte Daten	inhaltsbasierte Attribute wie `time`, `row_max_peak`, `row_min_background`, `row_max_mean`, sowie `RMSD`	Nur Referenzen (`filename`, `row_idx`, `time`) ohne Werte
Performance bei Abfragen	Hoch: Anfragen können direkt in SQLite beantwortet werden	Niedrig: Immer zweistufig (Index in HDF5), zusätzlicher I/O bei breiten Zeitintervallen
Speicherbedarf	Mittel: teilweise Redundanz, da Werte kopiert werden	Niedrig: sehr speichereffizient, keine Duplikation der Werte
Eignung für Projekt	Zweckmäßigste Lösung, da Effizienz und Vollständigkeit der Anfragen gegeben sind	Weniger geeignet, da hoher Mehraufwand beim Zugriff auf große HDF5-Dateien

Tabelle 3.2: Vergleich der Szenarien 2 und 3 der externen Indexierung

Für das vorliegende Projekt erweist sich das zweite Szenario als die zweckmäßigste Lösung. Die HDF5-Dateien bleiben dabei die vollständige Referenzbasis und behalten ihre Rolle als Quelle der Rohdaten. Gleichzeitig können alle relevanten Anfragen im externen Index effizient und ohne wiederholten Zugriff auf die großen HDF5-Dateien beantwortet werden. Dies stellt in Anbetracht der Größe der Daten und der hohen Kosten von direkten HDF5-Lesezugriffen einen erheblichen Vorteil dar.

3.4.2 Einsatz externer Datenbanksysteme

Externe Indexierung ermöglicht den Einsatz leistungsfähiger Datenbanksysteme, die spezialisierte Funktionen für Verwaltung, Abfrageoptimierung und Indexstrukturen bereitstellen, wie beispielsweise RocksDB, Cassandra oder SQLite. Für das vorliegende Projekt wurde SQLite ausgewählt, da dieses System intern den `B+-Baum` als Indexstruktur verwendet. Der folgende Abschnitt erläutert näher, was SQLite ist und weshalb es für die externe Indexierung im Projekt verwendet wurde.

3.4.3 Externe Indexierung in SQLite

SQLite ist ein leichtgewichtiges relationales Datenbankmanagementsystem, das sich durch Einfachheit und Portabilität auszeichnet. Im Gegensatz zu komplexeren Systemen wie PostgreSQL oder MySQL basiert SQLite vollständig auf einer einzelnen Bibliothek und speichert alle Daten in genau einer Datei. Es eignet sich insbesondere für Anwendungen, die keine aufwändige Serverinfrastruktur benötigen, und unterstützt die gängigen SQL-Funktionen wie `Transactions`, `Indizes` und Abfragen über mehrere Tabellen. Zudem überzeugt SQLite durch hohe Stabilität und einen sehr geringen Verwaltungsaufwand. Alternative Systeme wie klassische Client-Server-Datenbanken sind zwar leistungsfähig, erfordern aber für dieses Projekt einen unverhältnismäßig hohen Installations- und Wartungsaufwand. Andere Speicherformate wie `CSV` sind zwar einfach und leicht lesbar, bieten jedoch keine ausgereiften Mechanismen für eine umfassende Indexierung. SQLite stellt hier einen idealen Mittelweg dar: Es bleibt leichtgewichtig und portabel, bietet gleichzeitig eine vollständige SQL-Schnittstelle und erlaubt die effiziente Speicherung der aus den HDF5-Dateien extrahierten wichtigsten Informationen in einer separaten, schlanken Indexdatenbank. Ein weiterer entscheidender Vorteil ist, dass SQLite intern auf die für dieses Projekt optimale Datenstruktur zurückgreift, nämlich den B^+-Baum, der bereits zuvor als beste Struktur für die vorgesehenen Anfragen herausgestellt wurde.
Die interne Organisation der Tabellen und Indizes in SQLite basiert auf B^+-Bäumen. Die eigentlichen Datenwerte befinden sich ausschließlich in den Blättern, während die inneren Knoten lediglich zur Navigation dienen. Darüber hinaus integriert SQLite mehrere Optimierungen gegenüber der klassischen Implementierung eines B^+-Baums. So erfolgt die Speicherung in festen Blöcken (`Pages`), was die I/O-Effizienz auf dem Datenträger deutlich verbessert. Zudem ist es möglich, mehrere voneinander unabhängige Indexbäume für unterschiedliche Attribute zu erstellen. Wichtige Erweiterungen umfassen `Covering Indexes`,

die es erlauben, bestimmte Abfragen vollständig im Index zu beantworten, sowie `Partial Indexes`, die nur auf gefilterte Datenbereiche angewandt werden und dadurch Zeit und Speicherplatz sparen. Ergänzend werden statistische Informationen über die Wertverteilungen genutzt, um die Abfrageoptimierung zu verbessern.

3.4.4 Wahl des Primärschlüssels

Als Primärschlüssel wurde die Kombination aus `filename` und `row_idx` gewählt. Diese Kombination stellt einen stabilen und direkten Verweis auf die jeweilige Zeile innerhalb der HDF5-Datei dar und ist unabhängig von der Qualität oder Art der gemessenen Werte. Andere Attribute wie `time` sind für diesen Zweck ungeeignet, da sie fehlende Werte (`NaN`) enthalten oder in unterschiedlichen Formaten gespeichert sein können. Auch die wissenschaftlichen Messgrößen wie `Peak` oder `Mean` wurden nicht als Primärschlüssel herangezogen, da ihre Werte nicht notwendigerweise eindeutig sind und somit Mehrfachvorkommen auftreten können.

3.4.5 Struktur des reduzierten Datenbankschemas

Um die wissenschaftlichen Anfragen zu beschleunigen, wurde eine reduzierte Tabelle in SQLite entworfen, die ausschließlich die wesentlichen Schlüssel- und Attributspalten enthält. Neben dem bereits zuvor erläuterten Primärschlüssel umfasst die Tabelle die abgeleiteten wissenschaftlichen Attribute `row_max_peak`, `row_min_background` und `row_max_mean`. Zusätzlich enthält sie die Spalte `time` als Zeitstempel, die insbesondere für zeitbasierte Abfragen und zur Auflösung von Gleichständen benötigt wird, sowie `RMSD` aus dem `BeamStabilizer`, das von den Physikern als analytisch relevant eingestuft wurde.
Die Reduktion auf diese Struktur bietet mehrere Vorteile. Zum einen steigt die Effizienz, da alle sekundären Spalten aus den HDF5-Dateien ausgeschlossen werden und die Tabelle dadurch kompakter wird, was sowohl das Einfügen als auch die Indexierung beschleunigt. Zum anderen bleibt die notwendige Flexibilität erhalten, da die ausgewählten Attribute die vorgesehenen wissenschaftlichen Anfragen vollständig abdecken. Sollte in Zukunft ein zusätzliches Attribut erforderlich sein, kann die Extraktion entsprechend erweitert werden. Die Gestaltung gewährleistet darüber hinaus Transparenz, da jede Zeile der Tabelle direkt auf eine Zeile in HDF5 verweist und damit leicht nachvollziehbar bleibt.
Zusammenfassend fungiert die reduzierte Tabelle als ideale Zwischenschicht: Sie ist kompakt, enthält alle für die Analyse wesentlichen Attribute und bewahrt gleichzeitig die Referenzierbarkeit der Originaldaten über den Primärschlüssel. Damit bildet sie die Grundlage für den anschließenden Aufbau geeigneter Indexstrukturen. Ein Beispielauszug der Datensätze mit dem gewählten Primärschlüssel ist in folgender Tabelle 3.3 dargestellt.

filename	row_idx	time	row_max_peak	row_max_mean	RMSD
LISA_20241008_0721.h5	0	1728372066	12.5	7.8	0.35
LISA_20241008_0721.h5	1	NaN	11.9	7.5	0.41
LISA_20241008_0721.h5	2	1728372068	0.0	7.6	0.38

Tabelle 3.3: Beispielhafte Datensätze des reduzierten Schemas mit Primärschlüssel und ausgewählten Attributen

3.4.6 Endgültiges Design der externen Indizes

Das endgültige Design der externen Indexstrukturen richtet sich nach den vordefinierten wissenschaftlichen Anfragen und basiert auf einer gezielten Auswahl der zu indizierenden Attribute sowie deren Sortierrichtung. Für den Index auf `row_max_peak` wurde eine absteigende Sortierung gewählt, da die Abfragen vom Typ `Top-N` stets die größten Werte erfordern. Durch die absteigende Ordnung können diese Werte unmittelbar und effizient gefunden werden. Der Index auf `row_min_background` hingegen wurde aufsteigend angelegt, da die Anfragen hier vom Typ `Bottom-N` sind und somit die kleinsten Werte

im Vordergrund stehen. Der Index auf `row_max_mean` dient der Unterstützung von Bereichsanfragen; da in diesem Fall beide Sortierrichtungen gleichermaßen geeignet wären, wurde die aufsteigende Ordnung gewählt, da sie dem Standard vieler Datenbanksysteme entspricht und die Implementierung vereinfacht. Zur Auflösung von Gleichständen wird `time` als sekundäres Attribut genutzt. Im Falle identischer Werte innerhalb eines Indexes erhält dabei der Eintrag mit dem kleineren Zeitstempel Vorrang. Diese zusätzliche Ordnung stellt sicher, dass die Ergebnisse deterministisch und reproduzierbar bleiben, auch wenn mehrere Messungen denselben Wert aufweisen.
Das Konzept basiert auf einem Primärschlüssel aus `filename` und `row_idx`. Darauf aufbauend werden Indizes für die drei zentralen Anfragearten (`Top-N`, `Bottom-N`, `Bereich`) definiert. Es handelt sich um Mehrspaltenindizes, die neben der jeweiligen Metrik (`row_max_peak`, `row_max_mean`, `row_min_background`) auch die Attribute `time`, `filename` und `row_idx` umfassen. Diese Struktur ermöglicht effiziente Abfragen mit zusätzlichen zeitlichen oder dateibezogenen Bedingungen. Der Nachteil besteht in einem höheren Aufwand bei der Indexerstellung, da das Datenbanksystem (SQLite) die Daten über mehrere Spalten hinweg sortieren und organisieren muss.
Obwohl das Design streng an den aktuellen Analysebedarf angepasst ist, bleibt es offen für künftige Erweiterungen. Die reduzierte Tabelle in SQLite kann bei Bedarf problemlos um zusätzliche Attribute ergänzt werden, und neue partielle oder zusammengesetzte Indizes lassen sich ohne grundlegende Umstrukturierungen hinzufügen. Damit bietet das vorliegende Konzept eine robuste Grundlage, die einerseits die aktuellen Abfragen optimal beschleunigt und andererseits genügend Flexibilität für zukünftige Anforderungen bewahrt. Im Folgenden ist ein beispielhafter Index auf der Spalte `row_max_peak` dargestellt.

row_max_peak	**time**	**filename**	**row_idx**
12.5	1728372066	LISA_20241008_0721.h5	0
11.9	NaN	LISA_20241008_0721.h5	1
0.0	1728372068	LISA_20241008_0721.h5	2

Tabelle 3.4: Beispielhafte Darstellung eines Indexes auf der Spalte `row_max_peak`

3.5 Interne Indexierung

Wie bereits zuvor erläutert, bedeutet interne Indexierung, dass die Indizes direkt innerhalb der HDF5-Dateien selbst angelegt werden. Bei der Konzeption dieser Indexierung in HDF5-Dateien lassen sich im Wesentlichen drei Strategien unterscheiden, die sich hinsichtlich Datenvolumen, Abfrageeffizienz, Wartbarkeit und potenziellen Risiken voneinander abgrenzen.
Eine erste Möglichkeit besteht darin, sämtliche verfügbaren HDF5-Dateien in einer einzigen großen Datei zusammenzuführen, die mehrere Dutzend Gigabyte umfassen kann. Innerhalb dieser zentralen Datei wird ein einheitlicher interner Index aufgebaut, der alle Daten abdeckt. Der Vorteil dieser Lösung liegt in der direkten und schnellen Bearbeitung globaler Anfragen, da ein konsistenter Index ohne Wechsel zwischen verschiedenen Dateien zur Verfügung steht. Der Nachteil zeigt sich jedoch in der erschwerten Wartung: Jede Änderung oder Erweiterung der Datenbasis erfordert eine komplette Neugenerierung der großen Datei, was zeit- und ressourcenintensiv ist. Zudem führt eine mögliche Beschädigung der Datei zu einem vollständigen Datenverlust.
Ein zweiter Ansatz sieht vor, dass jede HDF5-Datei über einen eigenen internen Index verfügt, der ausschließlich die zugehörigen Daten erfasst. Damit wird jede Datei zu einer in sich geschlossenen Einheit, die sowohl die Rohdaten als auch den Index enthält. Dies erleichtert Transport und Austausch und reduziert das Wartungsrisiko. Auch die Vergrößerung des Dateiumfangs bleibt durch die Indexinformationen moderat. Problematisch wird dieser Ansatz jedoch bei globalen Abfragen, die mehrere Dateien betreffen, da diese sequentiell durchsucht werden müssen. Noch schwerwiegender ist die Tatsache, dass jede Datei über eine eigene Baumstruktur verfügt, sodass eine globale Anfrage erst durch zusätzliche Mechanismen wie einen übergeordneten externen Index effizient beantwortet werden kann. Rein interne Indexierung ist damit für umfassende Anfragen allein nicht ausreichend.

Ein drittes Szenario kombiniert nur die wichtigsten Daten aus allen Dateien zunächst in einer reduzierten HDF5-Datei und legt darauf einen einheitlichen internen Index, etwa in Form einer B^+-Baum-Struktur, an. Dieser Ansatz verringert den Gesamtspeicherbedarf deutlich, da nur eine komprimierte Repräsentation der Daten abgelegt wird, und ermöglicht zugleich die effiziente Bearbeitung globaler Anfragen auf einer konsistenten Indexbasis. Für das vorliegende Projekt erweist sich das dritte Szenario als die zweckmäßigste Lösung und wurde daher für die Evaluation ausgewählt.

Szenario	**Vorteile**	**Nachteile**
1: Eine zentrale Datei	Konsistenter globaler Index, schnelle Anfragen	Hoher Speicherbedarf, schwierige Wartung, Risiko bei Dateibeschädigung
2: Einzelindex pro Datei	Modular, leicht zu verwalten	Globale Abfragen ineffizient, keine einheitliche Indexbasis
3: Reduzierte kombinierte Datei	Speicher effizient, globale Anfragen machbar	Zusätzlicher Verarbeitungsschritt, erhöhte Komplexität

Tabelle 3.5: Vergleich möglicher Szenarien der internen Indexierung

3.5.1 Struktur der reduzierten Datei

Die reduzierte Datei enthält die gleichen Spalten wie bei der externen Indexierung, nämlich `row_max_peak`, `row_max_mean`, `row_min_background`, `time`, `filename` sowie `row_idx_in_file`. Diese Spalten sind innerhalb einer Gruppe `Measurements` abgelegt. Zusätzlich wird im selben HDF5-File eine weitere Gruppe angelegt, in der die internen Indizes (d. h. die jeweiligen Baumstrukturen) gespeichert werden.
Bei der Konzeption der internen Struktur wurde entschieden, jedes Attribut in einem separaten Datensatz innerhalb einer Untergruppe zu speichern, anstatt einen einzigen zusammengesetzten Datensatz in Tabellenform zu verwenden. Diese Entscheidung beruht auf mehreren Überlegungen. Zum einen entspricht sie der ursprünglichen Organisation der HDF5-Dateien im Institut, in denen jede Messgröße wie `time`, `row_max_peak` oder `row_min_background` bereits als eigenständiger Datensatz vorliegt. Auf diese Weise bleibt die Konsistenz zwischen den Rohdaten und den reduzierten Dateien gewahrt. Zum anderen erleichtert die getrennte Speicherung den direkten Zugriff unmittelbar auf die Zeitwerte, ohne zuvor eine zusammengesetzte Struktur auslesen und daraus die gewünschte Spalte extrahieren zu müssen. Darüber hinaus bietet dieser Ansatz eine hohe Flexibilität für Erweiterungen: Soll später ein zusätzliches Attribut wie `row_min_peak` ergänzt werden, kann dies einfach durch das Anlegen eines neuen Dataset erfolgen, ohne dass die bestehende Struktur oder der zugrunde liegende `dtype` verändert werden muss. Schließlich erlaubt die Aufteilung in mehrere Datasets eine optimale Nutzung der Funktionalitäten von HDF5, da Kompression oder Chunking für jede Messgröße individuell konfiguriert werden können, was eine bessere Anpassung an die jeweilige Datencharakteristik ermöglicht. Demgegenüber hätte die alternative Lösung, ein einziges zusammengesetztes `Compound Dataset` mit allen Attributen zu verwenden, deutliche Nachteile. Sie würde die Erweiterbarkeit einschränken, den Zugriff auf einzelne Spalten erschweren und eine differenzierte Anwendung von Speicher- und Kompressionseinstellungen verhindern. Daher erweist sich die gewählte Struktur als zweckmäßigste Lösung für die Anforderungen des Projekts.

3.5.2 Integration eines benutzerdefinierten Datentyps im HDF5-Dateiformat

Bei der internen Indexierung innerhalb von HDF5-Dateien ergibt sich eine besondere Einschränkung: Neue Strukturen können nur mithilfe von `Compound Datatypes` realisiert werden, die wiederum in `Datensätzen` oder `Gruppen` abgelegt werden müssen, da diese die zentralen Strukturelemente des HDF5-Formats darstellen. Allerdings kennt HDF5 die Semantik eines benutzerdefinierten Datentyps nicht von selbst. Entsprechende Operationen können daher nicht direkt durch HDF5-Befehle ausgeführt werden,

sondern müssen explizit implementiert werden. Zudem muss die Größe jedes Teils dieser neuen benutzerdefinierten Struktur vorab festgelegt werden, was bedeutet, dass dynamisch wachsende Strukturen hier nicht unterstützt werden. Damit unterscheidet sich die interne Indexierung grundlegend von der externen, bei der die Datenbank (z. B. SQLite) die Indexlogik bereits integriert mitliefert. In der folgenden Tabelle wird die finale Struktur der Datensätze innerhalb der reduzierten Datei dargestellt, wobei die Baumstrukturen als Indizes in einer eigenen Gruppe gespeichert sind.

Gruppe	**Datensatz**
`Measurements_short`	- `time` - `row_max_peak` - `row_max_mean` - `row_min_background` - `filename` - `row_idx_in_file` - `RMSD`
`Indexes`	- `bplus_peak` - `bplus_mean` - `bplus_background`

Tabelle 3.6: Übersicht der reduzierten HDF5-Datei mit den abgelegten Messdaten und den darin enthaltenen Indexstrukturen

3.5.3 B$^+$-Baum als interne Indexstruktur innerhalb des HDF5-Dateiformats

Die Integration eines B$^+$-Baums innerhalb von HDF5 wurde auf Basis eines `Compound Dataset` realisiert. Die Knoten enthalten dabei sowohl Schlüssel als auch Zeiger (`Pointers`). Die zentrale Herausforderung bestand darin, die Anzahl der Elemente pro Knoten festzulegen. Theoretisch ist die Kapazität eines Knotens im B$^+$-Baum nicht konstant, sondern hängt von der Ordnung des Baums und der Datenmenge ab. Darüber hinaus musste bestimmt werden, wie viele Schlüssel und Zeiger in jedem Knoten gespeichert werden. Entweder hätte diese Zahl exakt vorab festgelegt werden müssen, oder man hätte einen sehr großen Wert wählen können, der sämtliche Fälle abdeckt. Letzteres hätte jedoch zu erheblicher Speicherineffizienz und spürbaren Leistungseinbußen geführt. Daher wurde ein fester, praxisgerechter Wert gewählt, der die meisten Fälle abdeckt, ohne die Abfragen unnötig zu verlangsamen.

Die endgültige Struktur ist ein flacher Baum, der aus einer Wurzel besteht, die direkt auf die Blätter verweist. Jedes Blatt enthält die Schlüsselwerte der indexierten Spalte, beispielsweise `row_max_peak`, `row_max_mean` oder `row_min_background`, sowie Zeiger auf die tatsächlichen Zeilen im HDF5 über den globalen Zeilenindex. Weitere Messwerte wie `time` oder `RMSD` werden nicht im Baum selbst gespeichert, sondern verbleiben in der Gruppe `Measurements_short` und können über die Zeiger nachgeladen werden. Die Blätter sind zusätzlich über das Feld `NextLeaf` verkettet, sodass Bereichsanfragen effizient durch sequentielles Traversieren der Blätter verarbeitet werden können. Jeder Knoten des Baums besteht aus einer eindeutigen Kennung (`NodeID`), einem Indikator, ob es sich um ein Blatt handelt (`IsLeaf`), einer festen Menge an Schlüsseln (`Keys`), den zugehörigen Zeigern (`Pointers`) sowie einem zusätzlichen Verweis auf das nächste Blatt (`NextLeaf`). Folgende Abbildung zeigt die vereinfachte Struktur des im Projekt implementierten B$^+$-Baums.

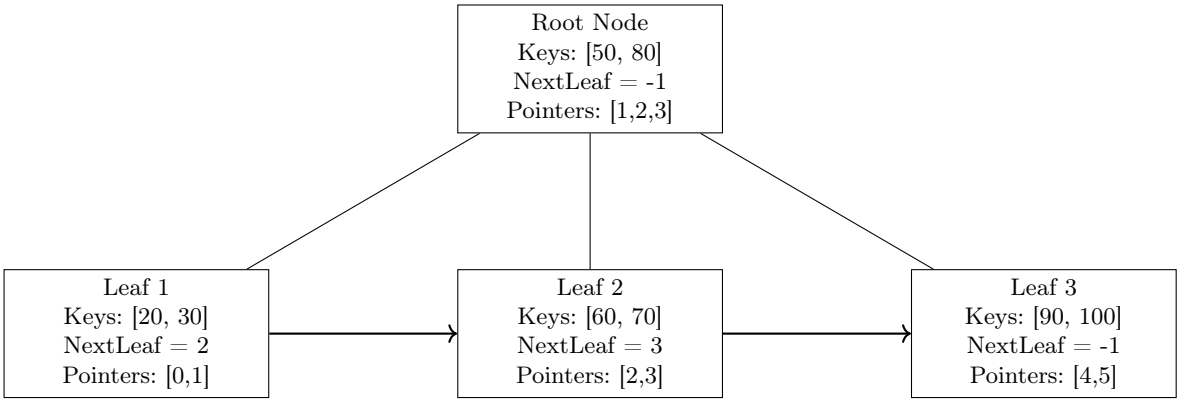


Abbildung 3.1: B^+-Baum-Struktur mit Wurzel und Blättern, wobei die Blätter über `NextLeaf`-Zeiger miteinander verbunden sind.

Bei dieser Indexierung mittels B^+-Baums wird eine lexikographische Sortierung über mehrere Attribute durchgeführt. Dabei erfolgt die primäre Ordnung nach der jeweiligen Metrik (`row_max_peak`, `row_max_mean` oder `row_min_background`), während zusätzliche Attribute wie `time`, `filename` und `row_idx` lediglich als sekundäre Kriterien dienen, um eine eindeutige und konsistente Reihenfolge sicherzustellen. Diese zusätzliche Attribute werden jedoch nicht in der Baumstruktur selbst gespeichert. Die Blätter enthalten ausschließlich die Werte der jeweiligen Metrik als Schlüssel sowie die Zeiger auf den globalen Zeilenindex, die auf die Originalzeilen in der Gruppe `Measurements_short` verweisen.
Mit diesem Aufbau bleibt die Größe der Indexstruktur klein, da keine vollständige Datenkopie erfolgt, sondern lediglich die Schlüsselwerte und Zeiger gespeichert werden. Die Bearbeitung der Anfragen erfolgt in mehreren Schritten: Zunächst wird über die Wurzel des Baums auf die in den Blättern abgelegten Werte zugegriffen. Anschließend werden mithilfe der in den Blättern gespeicherten Zeilenindizes die korrespondierenden Werte aus den ursprünglichen Datensätzen nachgeladen. Auf diese Weise müssen nicht alle relevanten Werte direkt im Index gespeichert werden; es genügt, die Zeilenindizes als Referenzen vorzuhalten. Dieses Vorgehen reduziert den Speicherbedarf erheblich. Damit entspricht die innere Logik dem Prinzip eines B^+-Baums, wird jedoch in einer vereinfachten, flachen Struktur innerhalb eines festen *Compound-Datasets* in HDF5 realisiert.

3.6 Ohne Indexierung

Ohne Indexierung bedeutet, dass keine fertige Indexstruktur zur Informationsgewinnung genutzt wird, sondern der Zugriff direkt auf die HDF5-Dateien selbst erfolgt. Dabei ist ein wichtiger Punkt zu beachten: Aus technischer Sicht stellt das HDF5-Framework (sowohl die ursprüngliche C-Bibliothek als auch Schnittstellen wie `h5py`) keine integrierte Abfrage-Engine bereit. Es ist primär für die hierarchische Speicherung von Daten über Elemente wie `Gruppen`, `Datensätze` und `Datentypen` konzipiert und bietet keine Funktionen, die mit klassischen Datenbanken (z. B. SQL) vergleichbar wären. Daher können die Abfragen nicht direkt durchgeführt werden. Das Maximum, was HDF5 auf dieser Ebene ermöglicht, ist ein teilweiser Datenzugriff mittels Techniken wie `Slicing` sowie die Nutzung von Optimierungsmechanismen wie `Chunking` und `Compression`. Allerdings fehlen eingebaute Funktionen für Filterung, Sortierung oder statistische Berechnungen.
Die Durchführung erweiterter Abfragen erfordert daher das Laden der Daten in den Arbeitsspeicher und deren externe Verarbeitung mit Werkzeugen wie `NumPy` oder `Pandas`, oder mit einer anderen Programmiersprache. Somit hängt die Qualität dieses Vorgangs in erster Linie von der Leistungsfähigkeit der eingesetzten Programmiersprache und deren Werkzeuge ab sowie von der Effizienz, mit der HDF5 den Datenzugriff für diese Werkzeuge bereitstellt. Die folgende Abbildung 3.2 verdeutlicht die einzelnen

Schritte der Variante ohne Indexierung.

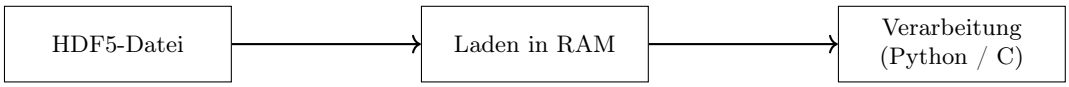


Abbildung 3.2: Datenfluss ohne Indexierung: Laden von HDF5 in den Arbeitsspeicher und Verarbeitung

3.7 Allgemeine Entwurfsentscheidungen

Unabhängig von der gewählten Indexierungsstrategie mussten im Projekt einige grundlegende Entscheidungen getroffen werden, die für alle Ansätze gleichermaßen gelten. Diese betreffen insbesondere den Umgang mit problematischen Werten sowie die Wahl einer konsistenten Zeitreferenz für zeitbasierte Abfragen.

Umgang mit Nullwerten

Bei der Konstruktion der Indizes stellte sich die Frage, wie mit fehlenden oder problematischen Werten in den HDF5-Dateien umzugehen ist. In den Daten treten zwar vereinzelt echte `NULL`-Werte auf, jedoch findet sich in sehr großem Umfang der Wert `0`. Dies erschwert die Interpretation, da `0` einerseits ein gültiges Messergebnis darstellen kann, andererseits aber in vielen Fällen als Platzhalter für einen eigentlich fehlenden Wert dient. Theoretisch wäre es möglich, solche Werte in `NULL` zu überführen, was zu kleineren und schnelleren Indizes führen und statistische Abfragen vereinfachen würde. Allerdings besteht dabei die Gefahr, dass auch tatsächlich gemessene Nullen verloren gehen, da nicht eindeutig zwischen „echten“ und „ersetzten“ Nullen unterschieden werden kann.
Aus diesem Grund wurde entschieden, die `0` Werte grundsätzlich beizubehalten. Dadurch bleibt die wissenschaftliche Aussagekraft der Daten vollständig erhalten. Gleichzeitig bleibt die Flexibilität gewahrt, indem bei Bedarf partielle Indizes eingesetzt werden können, die Nullwerte in bestimmten Abfragen gezielt ausschließen. Auf diese Weise wird ein Kompromiss zwischen wissenschaftlicher Vollständigkeit und technischer Effizienz erreicht[1].

[1] Eine Ausnahme bildet die Berechnung des `Minimum Background`. In diesem speziellen Fall führen Nullen zu einer systematischen Verfälschung des Ergebnisses, weshalb sie dort gezielt ausgeschlossen werden können.

Kapitel 4

Prototypische Implementierung

In diesem Kapitel wird die praktische Umsetzung des zuvor entwickelten Konzepts beschrieben. Der Fokus liegt dabei auf der eingesetzten Entwicklungsumgebung und der Implementierung der Indexstrukturen. Ziel ist nicht die Entwicklung eines produktiven Systems, sondern die prototypische Umsetzung zur Überprüfung der Praxistauglichkeit. Anhand ausgewählter Messdaten wird demonstriert, wie die unterschiedlichen Integrationsarten von Indexstrukturen praktisch realisiert und erprobt werden können.

4.1 Entwicklungsumgebung

Die Entwicklung und Evaluierung des Prototyps erfolgte auf einem Laptop des Typs HP 17-ca1xxx mit installiertem Betriebssystem Windows 10 Home in der 64-Bit-Variante. Das System ist mit einem AMD Ryzen 5 3500U Prozessor mit acht logischen Kernen und einer Taktfrequenz von etwa 2,1 GHz ausgestattet und verfügt über 16 GB Arbeitsspeicher. Für grafische Berechnungen und Visualisierungen steht eine integrierte AMD Radeon Vega 8 Grafikeinheit mit 2 GB dediziertem VRAM zur Verfügung, die insgesamt auf rund 9 GB Grafikspeicher zurückgreifen kann.
Die Implementierung der entwickelten Verfahren wurde in einer modernen Entwicklungsumgebung auf Basis von Visual Studio Code durchgeführt, wobei die Notebook-Funktionalität des Editors genutzt wurde, um Experimente, Code und Auswertungen in strukturierter Form zu kombinieren. Als Programmiersprache kam Python in der Version 3.11.9 zum Einsatz.

Komponente	**Spezifikation**
Laptop	HP 17-ca1xxx
Betriebssystem	Windows 10 Home, 64-Bit
Prozessor	AMD Ryzen 5 3500U, 8 logische Kerne, ca. 2,1 GHz
Arbeitsspeicher	16 GB RAM
Grafik	AMD Radeon Vega 8, 2 GB VRAM (insg. 9 GB nutzbar)
Entwicklungsumgebung	Visual Studio Code (Notebook-Funktionalität)
Programmiersprache	Python 3.11.9

Tabelle 4.1: Übersicht über die eingesetzte Hardware- und Softwareumgebung

Für die Realisierung der Funktionalitäten wurden ausgewählte Standardbibliotheken sowie externe Pakete genutzt. Die relevanten Versionen lauten wie folgt: h5py 3.14.0, numpy 2.2.6, pandas 2.3.1, sqlite3 2.6.0 sowie re 2.2.1. Weitere verwendete Module wie `os`, `glob`, `subprocess`, `time`, `datetime`, `socket` und `pathlib` stammen aus der Standardbibliothek von Python und sind versionsunabhängig an die jeweilige Python-Installation gebunden. Diese Konstellation erlaubte es, die Entwicklungsschritte effizient umzusetzen und die Ergebnisse reproduzierbar zu dokumentieren. Im Folgenden werden die verwendeten

Python-Bibliotheken vorgestellt, die für die Realisierung der Funktionalitäten zum Einsatz kamen (siehe Tabelle 4.2).

Bibliothek	**Version**	**Verwendung**
h5py	3.14.0	Zugriff auf HDF5-Dateien und Arbeit mit hierarchischen Datenstrukturen
numpy	2.2.6	Numerische Berechnungen und effiziente Array-Operationen
pandas	2.3.1	Verarbeitung tabellarischer Daten mithilfe von DataFrames
sqlite3	2.6.0	Einbettung einer relationalen Datenbank (SQLite) zur Indexverwaltung
re	2.2.1	Reguläre Ausdrücke zur Mustererkennung und Textverarbeitung
os	built-in	Schnittstelle zum Betriebssystem, u. a. für Pfad- und Prozessmanagement
glob	built-in	Auffinden von Dateien anhand von Platzhaltermustern
subprocess	built-in	Starten und Steuern externer Prozesse aus Python heraus
time	built-in	Zeitmessung, Pausen und Zeitsteuerung während der Programmausführung
datetime	built-in	Verarbeitung von Datums- und Zeitangaben in verschiedenen Formaten
socket	built-in	Kommunikation über Netzwerke mittels Sockets
pathlib	built-in	Objektorientierte Arbeit mit Dateipfaden und Verzeichnissen

Tabelle 4.2: Übersicht über die verwendeten Python-Bibliotheken und Module

Nutzung der Jupyter-Notebooks in Visual Studio Code Für die Ausführung des im Rahmen dieser Arbeit entwickelten Python-Codes wurden im Anhang detaillierte Hinweise zur Nutzung der Jupyter-Notebooks innerhalb von Visual Studio Code aufgenommen (siehe Anhang A.1). Dort wird Schritt für Schritt erläutert, wie die Notebooks gestartet und verwendet werden können. Diese ergänzenden Informationen dienen dazu, die Reproduzierbarkeit der Experimente sicherzustellen und auch den Zugang zu den Implementierungen zu erleichtern.

4.2 Struktur der Implementierung

Für jeden betrachteten Indexierungstyp – externe, interne sowie die Variante ohne Indexierung – wurde ein eigenes Jupyter Notebook (`.ipynb`) erstellt. Obwohl sich die einzelnen Szenarien inhaltlich unterscheiden, folgte die Struktur der Notebooks einem einheitlichen Aufbau mit klar abgegrenzten Zellen. Die erste Zelle umfasste die notwendigen Importe, allgemeine Einstellungen sowie Hilfsfunktionen für die Extraktion der Daten. In Zelle (2.5) erfolgte die Auswahl der zu verarbeitenden HDF5-Dateien. Anschließend wurden in Zelle (3) die wesentlichen Kenngrößen aus den HDF5-Dateien extrahiert, darunter `row_max_peak` (Maximalwerte über die vier Kanäle im Peak), `row_min_background` (Minimalwerte über die vier Kanäle im Background) sowie `row_max_mean` (Maximalwerte über die vier Kanäle im Mean gemäß Konzept). Die Zelle (4) war vom jeweiligen Szenario abhängig: Bei der externen Indexierung wurde eine separate Datenbank erstellt und die extrahierten Informationen darin abgelegt, während in der internen Indexierung ein reduziertes HDF5-File erzeugt wurde, das ausschließlich die relevanten Daten enthielt[1]. In der Variante ohne Indexierung entfiel dieser Schritt. Zelle (5) definierte die allgemeinen Einstellungen für die Abfragen, insbesondere die Messkriterien und die Anzahl der Wiederholungen. Darauf aufbauend wurden in Zelle (5.2) die Hilfsfunktionen zur Durchführung der Abfragen implementiert. Die anschließenden Zellen enthielten die eigentlichen Abfragen: Zelle (5.5) realisierte die Bereichsanfrage auf dem Mean im Intervall von $10\%M$ bis $80\%M$, Zelle (6) umfasste die Peak-Abfrage nach dem `Top-N%`-Prinzip und Zelle (7) die Background-Abfrage für die kleinsten 10% der Werte unter Ausschluss der Nullen. Die Zelle (7.1) dient zu der Ausgabe und Formatierung der Resultate. Durch diesen einheitlichen Aufbau konnte die Implementierung konsistent dokumentiert und die Ergebnisse der

[1] Bei der externen Indexierung wurden die Indizes direkt innerhalb derselben Zelle der Erstellung der reduzierten Tabelle in der Datenbank (Zelle 4) erzeugt. Bei der internen Indexierung hingegen musste dieser Schritt aufgrund des großen Umfangs in zwei separate Zellen aufgeteilt werden (Zelle 4.5 und Zelle 4.6).

unterschiedlichen Szenarien systematisch miteinander verglichen werden. Die folgende Tabelle zeigt die in den drei Varianten implementierten Zellen übersichtlich.

Zelle	Externe	Interne	Ohne
(1) Importe + Hilfsfunktionen	Ja	Ja	Ja
(2.5) Dateiauswahl (all/range/match)	Ja	Ja	Ja
(3) Extraktion der Metriken	Ja	Ja	Nein
(4) Reduzierte Datei bzw. Tabelle	Ja in (SQLite)	Ja in (HDF5)	Nein
(5) Abfrageeinstellungen (REPEATS, Scopes)	Ja	Ja	Ja
(5.2) Abfragehilfsfunktionen	Ja	Ja	Ja
(5.5) Mean-Bereichsanfragen 10%..80%	Ja	Ja	Ja
(6) Peak-Top-10%-Anfragen	Ja	Ja	Ja
(7) Background-Bottom-10%-Anfragen	Ja	Ja	Ja
(7.1) Speichern der Ergebnisse	Ja	Ja	Ja

Tabelle 4.3: Übersicht der implementierten Zellen in den drei Varianten

4.3 Implementierungsdokumentation

In diesem Abschnitt wird die konkrete Umsetzung der im Kapitel 3 beschriebenen Verfahren dokumentiert. Der Fokus liegt nun auf der technischen Ausgestaltung einzelner Schritte. Dazu zählen ausgewählte, repräsentative Codeausschnitte in Python, die innerhalb der Notebooks implementiert wurden. Die in diesem Kapitel präsentierten Ausschnitte stellen dabei eine bereinigte und gekürzte Version der vollständigen Implementierung dar, die im Anhang A vollständig aufgeführt ist. Zudem werden die hier beschriebenen Notebook-Zellen systematisch mit den ursprünglichen, im Anhang enthaltenen Quellzellen verknüpft, sodass jederzeit ein direkter Bezug zwischen der kompakten Darstellung und der vollständigen Implementierung hergestellt werden kann. Ziel ist es, die Nachvollziehbarkeit der Implementierung sicherzustellen und eine reproduzierbare Grundlage für die anschließende Evaluation zu schaffen.

4.3.1 Allgemeine Zellen

Unabhängig davon, ob eine externe Indexierung, eine interne Indexierung oder die Variante ohne Indexierung gewählt wird, existieren bestimmte Zellen, die in allen Implementierungen identisch oder weitgehend identisch ausgeführt werden. Diese allgemeinen Zellen enthalten grundlegende Einstellungen und Hilfsfunktionen, die für die Durchführung der Abfragen erforderlich sind. Auf diese Weise wird eine einheitliche Grundlage geschaffen, die einen systematischen Vergleich der unterschiedlichen Varianten ermöglicht.

Initialisierung der Arbeitsumgebung und Vorbereitung der Extraktion

Die erste Notebook-Zelle (1) entspricht Zelle A.1 im Anhang und dient der Initialisierung der Arbeitsumgebung. Hier werden die grundlegenden Parameter für die Ausführung definiert, zentrale Hilfsfunktionen implementiert und die notwendigen Module eingebunden. Diese Vorbereitungen bilden die technische Grundlage, um in den folgenden Schritten die relevanten Daten aus den HDF5-Dateien zuverlässig extrahieren zu können. Damit werden die zentralen Verzeichnispfade vorbereitet. Durch den Befehl `mkdir` wird das Ausgabe-Verzeichnis automatisch erzeugt, falls es noch nicht existiert. Außerdem werden Hostname und Zeitstempel gespeichert, um jede Ausführung eindeutig identifizieren zu können.

Auswahl der zu verarbeitenden HDF5-Dateien

Die folgende Implementierung entspricht der Notebook-Zelle (2.5) und ist in Zelle A.2 im Anhang dargestellt. Hier wird festgelegt, welche HDF5-Dateien für die Verarbeitung berücksichtigt werden. Damit wird der Eingabedatenbestand für alle weiteren Schritte bestimmt.

Listing 4.1: Zelle (2.5) Auswahl der HDF5-Dateien

```
SELECTION_MODE = "all"
_all_files = sorted(glob.glob(GLOB_PAT))
_selected = []
if SELECTION_MODE == "all":
    _selected = _all_files

TARGET_FILES = _selected
sel_list_path = os.path.join(RESULTS_DIR, f"selected_files_{RUNSTAMP}.txt")
with open(sel_list_path, "w", encoding="utf-8") as f:
    for fp in TARGET_FILES:
        f.write(fp + "\n")
```

Abhängig vom Modus `SELECTION_MODE` können entweder alle verfügbaren Dateien (`all`), ein bestimmtes Muster (`match`) oder ein definierter Zeitbereich (`range`) ausgewählt werden. Für die Bereichsauswahl werden Zeitstempel aus den Dateinamen extrahiert, sortiert und anschließend mit den gewählten Start- und Endpunkten abgeglichen.

Extraktion der wichtigsten Daten aus den HDF5-Dateien

Die Notebook-Zelle (3) entspricht Zelle A.3 im Anhang und ist für die Extraktion der zentralen Kenngrößen aus den HDF5-Dateien zuständig. Für jedes Gerät werden die relevanten Messreihen geladen, vereinheitlicht und anschließend die abgeleiteten Metriken berechnet. Diese Zelle entfällt bei der Variante ohne Indexierung. Die Funktion `_prep_matrix_keep_zeros()` stellt sicher, dass die Kanalwerte in eine einheitliche Matrixform gebracht werden, wobei Nullen explizit beibehalten werden. Die zentrale Routine `extract_index_for_device()` lädt die Messreihen für ein spezifisches Gerät, vereinheitlicht deren Länge und berechnet anschließend die drei zentralen Metriken: `row_max_peak`, `row_min_background` und `row_max_mean`. Zusätzlich werden Gültigkeits-Flags gesetzt, die kennzeichnen, ob die jeweiligen Werte vorliegen. Anschließend werden alle Ergebnisse geräteweise gesammelt und zu einem Gesamt-DataFrame zusammengeführt. Die gesamte Extraktion und Zusammenführung werden zeitlich gemessen, um die Performanz der Verarbeitung zu dokumentieren.

Allgemeine Einstellungen für die Abfragen

Die Notebook-Zelle (5) entspricht Zelle A.5 im Anhang und definiert allgemeine Parameter, die für alle nachfolgenden Abfragen (Mean, Peak, Background) benötigt werden. Dazu gehören die Auswahl des Gerätes, die Definition von Zeitintervallen, die Anzahl der Wiederholungen sowie unterschiedliche Szenarien. Eine Hilfsfunktion liefert zudem den Tabellennamen, der einem bestimmten Gerät entspricht.

Listing 4.2: Zelle (5) Allgemeine Einstellungen für die Abfragen

```
DEVICE   = 0
T_START  = 1728372066.996737
T_END    = 1728372119.978274
REPEATS  = 5
PERCENT  = 10.0
SCOPES = [ "all",
           "LISA_20241008_072100.h5"]
TIME_SCENARIOS = [ False,
                   True ]
def _table_name_for_device(d):
    return f"entries_dev{d}"
```

Die Variablen DEVICE, T_START, T_END und REPEATS legen fest, welches Gerät betrachtet wird, welcher Zeitbereich bei zeitabhängigen Abfragen gilt und wie oft jede Abfrage wiederholt wird. Mit PERCENT wird der Anteil für Top-N-Prozent-Operationen definiert. Die beiden Listen SCOPES und TIME_SCENARIOS geben an, ob die Abfragen über alle Dateien oder nur über eine Datei laufen und ob eine Zeitbedingung berücksichtigt wird. Die Hilfsfunktion _table_name_for_device(d) generiert den Tabellennamen in der SQLite-Datenbank, der einem spezifischen Gerät entspricht (z. B. entries_dev0). Dadurch wird die spätere Formulierung der SQL-Abfragen vereinfacht.

4.3.2 Besondere Zellen in der externen Indexierung

In diesem Abschnitt werden die besonderen Zellen der externen Indexierung vorgestellt. Zwar existieren in allen betrachteten Varianten ähnliche Zellen, jedoch sind sie nicht identisch, da sich insbesondere der Inhalt der Funktionen unterscheidet. Aus diesem Grund beginnen wir mit der vierten Zelle, in der eine externe Datenbank erstellt wird. Der vollständige Code befindet sich im Anhang A.2.

Erstellung der SQLite-Datenbank und Zeitmessung

Die Notebook-Zelle (4) entspricht Zelle A.4 im Anhang und erstellt für jedes Gerät[2] eine SQLite-Datenbank, in der die extrahierten Daten persistiert und Indizes angelegt werden. Dabei werden mehrere Wiederholungen durchgeführt, um die Zeitmessung für das Erstellen der Tabellen und Indizes zu erfassen. Am Ende wird ein Bericht gespeichert, der sowohl die Einzelergebnisse als auch arithmetische Mittelwerte enthält.

Listing 4.3: Zelle (4) Erstellung der SQLite-Datenbank mit Zeitmessung (gekürzt)

```
def write_sqlite_for_device_with_repeats(df, device, repeats=1):
    # 1) Vorbereitung der Daten (filename, row_idx, time, Peak, Mean, Background, RMSD)
    df_bulk = ...

    with sqlite3.connect(db_path, timeout=10) as conn:
        for rep in range(1, repeats+1):
            # Tabelle pro Device erzeugen
            conn.executescript("CREATE TABLE entries_dev...")

            # Bulk-Insert aller Messwerte
            conn.executemany("INSERT INTO entries_dev...", df_bulk...)

            # Indexe anlegen (z.B. fuer Mean, Peak, Background)
            create_index("CREATE INDEX idx_mean_time_dev...", "Mean", times_idx_mean)
            ...
    # Ergebnisse und Logdateien speichern
    return db_path, log_path, avg_table, avg_peak, avg_bg, avg_mean
```

Die Funktion write_sqlite_for_device_with_repeats() erstellt eine SQLite-Datenbank, befüllt diese mit den extrahierten Messwerten und legt anschließend Indizes für die drei Metriken row_max_peak, row_min_background und row_max_mean an. Jeder Durchlauf kann mehrfach wiederholt werden, sodass Zeitmessungen für das Anlegen der Tabellen und Indizes erfasst und Mittelwerte berechnet werden können. Anschließend ist eine Hilfsfunktion (Listing 4.4) zur Datenbankpflege enthalten:

[2] In den zugrunde liegenden HDF5-Dateien lagen Messungen von zwei Geräten vor (device 0 und device 1), die in unterschiedliche Richtungen ausgerichtet waren. Im Rahmen dieser Arbeit wurde jedoch ausschließlich auf die Messungen eines einzelnen Geräts fokussiert.

Listing 4.4: Bereinigung des SQLite-Speichers innerhalb Zelle (4)

```
def clear_sqlite_cache(db_path):
    try:
        with sqlite3.connect(db_path, timeout=5) as conn:
            conn.execute("PRAGMA shrink_memory;")
            conn.execute("PRAGMA optimize;")
            conn.commit()
    except Exception:
        pass
```

Durch den Aufruf von PRAGMA-Befehlen wird der Arbeitsspeicher von SQLite freigegeben und die Datenbank optimiert, was insbesondere bei wiederholten Abfragen für stabile Laufzeiten sorgt.

Hilfsfunktionen für die Abfragen bei der externen Indexierung

Die Notebook-Zelle (5.2) entspricht Zelle A.6 im Anhang und enthält zentrale Funktionen, die unterschiedliche Abfragetypen realisieren. Für jede Metrik wird ein Schwellenwert bestimmt, anhand dessen die entsprechenden Datensätze selektiert werden. Die im Anhang dokumentierten SQL-Abfragen bilden die konzeptionelle Grundlage für die im Code implementierten Abfragefunktionen (vgl. Abschnitt B.7).

Listing 4.5: Pseudocode der Hilfsfunktionen für Bereichs- und Prozentabfragen Zelle (5.2)

```
Funktion compute_M_for_scope(...)
    Bestimme Maximalwert M fuer row_max_mean
    Rueckgabe: M

Funktion mean_range_query(...)
    1. Bestimme M
    2. Waehle alle Werte im Bereich [0.1M .. 0.8M]
    Rueckgabe: Ergebnistabelle, M

Funktion threshold_for_top_percent(...)
    Bestimme Schwellenwert thr fuer oberstes Prozent
    Rueckgabe: thr

Funktion top_percent_peak(...)
    1. Bestimme Schwellenwert thr
    2. Waehle alle Werte >= thr
    Rueckgabe: Ergebnistabelle, thr

Funktion threshold_for_bottom_percent_bg(...)
    Bestimme Schwellenwert thr fuer unterstes Prozent
    Rueckgabe: thr

Funktion bottom_percent_bg(...)
    1. Bestimme Schwellenwert thr
    2. Waehle alle Werte <= thr
    Rueckgabe: Ergebnistabelle, thr
```

Die Funktion `mean_range_query` bestimmt zunächst den Maximalwert der Spalte `row_max_mean` und liefert anschließend alle Zeilen im Intervall von $10\%M$ bis $80\%M$. Die Funktion `top_percent_peak` berechnet einen Schwellwert, der dem obersten Prozentsatz der Werte in `row_max_peak` entspricht, und gibt alle Werte oberhalb dieses Schwellwerts zurück. Die Funktion `bottom_percent_bg` arbeitet analog, jedoch für die kleinsten Werte in `row_min_background`, wobei Nullwerte ausgeschlossen werden.

Abfragezellen für Mean, Peak und Background

Die folgenden Zellen führen die eigentlichen Abfragen auf den vorbereiteten Tabellen in SQLite aus. Je nach Szenario (alle Dateien oder nur eine Datei, mit oder ohne Zeitbedingung) werden die entsprechenden Datensätze ermittelt. Diese drei Abfragezellen sind im Anhang im Abschnitt A.2.1 dargestellt.

Listing 4.6: Zellen (5.5, 6 und 7) Abfragen für Mean, Peak und Background

```
# Mean Range (0.1M .. 0.8M)
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        with sqlite3.connect(dbp) as conn:
            df, M = mean_range_query(conn, table, scope, use_time, T_START, T_END)

# Peak Top-N%
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        with sqlite3.connect(dbp) as conn:
            df, thr = top_percent_peak(conn, table, PERCENT, scope, use_time, T_START, T_END)

# Background Bottom-N%
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        with sqlite3.connect(dbp) as conn:
            df, thr = bottom_percent_bg(conn, table, PERCENT, scope, use_time, T_START, T_END)
```

Die drei Schleifen repräsentieren die Kernabfragen. Die Funktion `mean_range_query` liefert Werte im Intervall zwischen 10% und 80% der `row_max_mean`. Mit `top_percent_peak` werden die obersten $N\%$ der Werte in `row_max_peak` bestimmt, während `bottom_percent_bg` die kleinsten $N\%$ der Werte in `row_min_background` auswählt, wobei Nullwerte ignoriert werden.

4.3.3 Besondere Zellen in der internen Indexierung

Im Gegensatz zur externen Indexierung, bei der die extrahierten Daten in eine separate Datenbank überführt werden, erfordert die interne Indexierung spezifische Anpassungen innerhalb der HDF5-Dateien selbst. Dadurch entstehen zusätzliche Zellen im Notebook, die ausschließlich für diese Variante relevant sind. In diesem Abschnitt werden jene Implementierungsschritte dokumentiert, die über die allgemeinen und gemeinsamen Abfragen hinausgehen und die Besonderheiten der internen Indexierung abbilden. Der vollständige Code befindet sich im Anhang A.3.

Erstellung einer HDF5-Kurzversion

In der internen Indexierung wird anstelle einer externen Datenbank in der Notebook-Zelle (4) eine einzige HDF5-Datei erzeugt, die nur die für die Indexierung relevanten Spalten enthält. Diese entspricht Zelle A.14 im Anhang. Damit wird eine kompakte Datengrundlage geschaffen, die als Basis für die interne Organisation mit B^+-Bäumen dient.

Listing 4.7: Zelle (4) Erstellung einer reduzierten HDF5-Datei

```
with h5py.File(merged_path, "w") as f:
    meas_group = f.create_group("Measurements_short")
    # Anlage der Kernspalten (time, row_idx_in_file, Peak, Mean, Background, RMSD, filename)
    ...
    idx_group = f.create_group("Indexes")
    # Schleife ueber alle Dateien und Devices
    for fp in TARGET_FILES:
        for d in DEVICES_TO_PROCESS:
            df_one = extract_index_for_device(fp, d)
            # Anhaengen der Werte an die reduzierten Datasets
            ...
test_file = merged_path
```

Die erzeugte Datei enthält nur die wesentlichen Spalten (`time`, `row_idx_in_file`, `row_max_peak`, `row_min_background`, `row_max_mean`, `RMSD`, `filename`). Damit werden alle extrahierten Werte aus den ursprünglichen HDF5-Dateien in kompakter Form zusammengefasst. Diese reduzierte Struktur bildet die Grundlage für den anschließenden Aufbau interner Indizes.

Aufbau von B$^+$-Bäumen mit globalen Indizes

Die Zelle (4.5) dient dem Aufbau von B$^+$-Bäumen, die globale Zeilenindizes (global row index) enthalten, und entspricht Zelle A.15 im Anhang. Diese Struktur wird innerhalb der reduzierten HDF5-Datei abgelegt und bildet die Grundlage für interne Indexabfragen.

Listing 4.8: Zelle (4.5) Aufbau von B$^+$-Bäumen

```
LEAF_CAPACITY = 732
NODE_CAPACITY = 732

def build_bplus_tree(vals, order_keys, ...):
    # 1) Gesamtsortierung nach den order_keys
    # 2) Aufbau der Blaetter mit Keys und globalen Zeiger
    # 3) Erzeugung der Root-Node mit Trennschluesseln
    return np.array([root] + nodes, dtype=...)

with h5py.File(test_file, "r+") as f:
    vals = {...}
    bplus_peak = build_bplus_tree(vals, [...])
    bplus_mean = build_bplus_tree(vals, [...])
    bplus_bg   = build_bplus_tree(vals, [...])
    # Speicherung in Gruppe "Indexes"
    ...
```

Die Funktion `build_bplus_tree` erzeugt für eine gegebene Spalte (z. B. `row_max_peak`) einen B$^+$-Baum, bei dem die Schlüsselwerte sortiert und mit globalen Zeilenindizes verknüpft werden. Die Blätter enthalten direkte Zeiger auf die Zeilen, während die Wurzel die Trennungsschlüssel und Verweise auf die Blätter enthält. Anschließend werden die erzeugten Strukturen in der HDF5-Datei unter der Gruppe `Indexes` gespeichert.

Weitere Hilfsfunktionen für interne B$^+$-Bäume

Die Zelle (4.6) entspricht Zelle A.16 im Anhang und enthält verschiedene Hilfsfunktionen, die für die Arbeit mit den internen B$^+$-Bäumen notwendig sind. Sie ermöglichen insbesondere das Auslesen von Datensätzen, das Iterieren über Blätter und das Abbilden von Intervallen auf bestimmte Teilbereiche des Baumes. So dient die Funktion `_iter_leaves` dem Durchlaufen der Blätter anhand der `NextLeaf`-Zeiger, während `_per_leaf_filter` ein lokales Filtern pro Blatt unter Berücksichtigung von Dateinamen und Zeitintervallen ermöglicht. Mit `_root_child_ids_and_seps` sowie `_leaf_range_for_interval` lassen sich aus den Root-Knoten die relevanten Kinder bestimmen und Intervalle auf konkrete Blattbereiche abbilden. Schließlich erlauben `_iter_leaves_by_pos` und `_iter_leaves_reverse_from_end` die gezielte Iteration über bestimmte Blätterbereiche, entweder in normaler oder in rückwärtiger Reihenfolge. Diese Hilfsfunktionen bilden die Grundlage für das strukturierte Navigieren innerhalb der intern aufgebauten B$^+$-Bäume.

Hilfsfunktionen für Abfragen auf dem B$^+$-Baum

Die Zelle (5.2) entspricht Zelle A.18 im Anhang und implementiert zentrale Funktionen, mit denen Abfragen direkt auf den in der HDF5-Datei gespeicherten B$^+$-Bäumen ausgeführt werden. Damit können Bereichsanfragen sowie Top- und Bottom-Prozent-Abfragen effizient ohne zusätzliche SQL-Indizes realisiert werden. Die Funktion `mean_range_query_bplus` ermittelt den Maximalwert M aus dem Baum für die Spalte `row_max_mean` und liefert anschließend alle Werte im Bereich zwischen 10% und 80% der `row_max_mean`. Die Funktion `top_percent_peak_bplus` bestimmt einen Schwellenwert, der dem obersten Prozentsatz der Werte in `row_max_peak` entspricht, und gibt alle Werte oberhalb dieses Schwellenwerts zurück. Die Funktion `bottom_percent_bg_bplus` berechnet analog den Schwellenwert für die kleinsten Werte in `row_min_background`, wobei (0) werte ausgeschlossen werden.

Ausführung der Abfragen auf dem B$^+$-Baum

Die finale Zelle bei internen Indexierung entspricht Zelle A.19 im Anhang und führt die zuvor definierten Abfragen direkt auf den internen B$^+$-Bäumen aus. Für jede Variante (Mean Range, Peak Top-% und Background Bottom-%) werden die Szenarien mit und ohne Zeitbedingung berücksichtigt. Damit werden die gleichen Abfragetypen wie bei der externen Indexierung abgebildet, jedoch vollständig innerhalb der HDF5-Datei verarbeitet.

Listing 4.9: Endgültige Ausführung der Abfragen

```
# Mean Range (0.1M .. 0.8M)
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        df, M = mean_range_query_bplus(test_file, scope, use_time, T_START, T_END)

# Peak Top-N%
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        df, thr = top_percent_peak_bplus(test_file, PERCENT, scope, use_time, T_START, T_END)

# Background Bottom-N%
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        df, thr = bottom_percent_bg_bplus(test_file, PERCENT, scope, use_time, T_START, T_END)
```

Die Zelle ruft die drei Kernfunktionen `mean_range_query_bplus`, `top_percent_peak_bplus` und `bottom_percent_bg_bplus` auf. Damit wird gezeigt, dass sämtliche Abfragen vollständig über die internen B$^+$-Bäume abgewickelt werden können, ohne auf eine externe Datenbank zurückzugreifen.

Darstellung der Funktionsweise des internen B$^+$-Baums am Beispiel der Bottom-10%-Abfrage auf `row_min_background`

Zu Beginn der Anfrage wird die HDF5-Datei mit `h5py` geöffnet und die Blätter des Baums sequentiell durchlaufen. Dabei werden die gespeicherten Werte und ihre Zeilenindizes extrahiert, während ungültige Einträge (NaN oder Nullwerte) verworfen werden. Anschließend wird auf Basis der Gesamtzahl N die Quantilsposition mit $q = 0.1$ berechnet, um den Schwellenwert `thr` für das untere Dezil zu bestimmen. Dieser Schwellenwert grenzt den relevanten Suchbereich im Baum ein, sodass nur die entsprechenden Blätter traversiert werden müssen. Aus diesen Blättern werden die passenden Zeilen unter Berücksichtigung von Zeit- und Dateifiltern extrahiert und in ein DataFrame überführt, das die wesentlichen Spalten `filename`, `row_idx`, `time` und `row_min_background` enthält. So liefert die Bottom-10%-Anfrage genau die Werte des unteren Zehntels, wobei die vereinfachte B$^+$-Baum-Struktur als interne Indexstruktur innerhalb von HDF5 zum Einsatz kommt.

4.3.4 Besondere Zellen in der Variante ohne Indexierung

In Ergänzung zu den zuvor dargestellten besonderen Zellen der externen und internen Indexierung werden im Folgenden die entsprechenden Zellen für die Variante ohne Indexierung vorgestellt. Auch hier handelt es sich um spezielle Codeabschnitte, die zur Durchführung der Abfragen dienen, jedoch ohne den Einsatz zusätzlicher Datenstrukturen. Dadurch lassen sich die Unterschiede in Aufbau und Ausführung im direkten Vergleich zu den beiden Indexierungsvarianten nachvollziehen. Der vollständige Code befindet sich im Anhang, siehe Anhang A.4.

Hilfsfunktionen für die Variante ohne Indexierung

Die Notebook-Zelle (5.2) entspricht Zelle A.23 im Anhang und stellt Hilfsfunktionen bereit, mit denen die relevanten Spalten direkt aus den HDF5-Dateien gelesen und bei Bedarf zu einem DataFrame zusammengeführt werden. Diese Hilfsfunktionen ermöglichen es, die für die Abfragen benötigten Werte `row_max_mean`, `row_max_peak` und `row_min_background` ohne zusätzliche Indizes zu berechnen.

Listing 4.10: Zelle (5.2) Hilfsfunktionen für die Variante ohne Indexierung

```
def _scan_mean_df(files, device):
    ...
    row_max_mean = np.nanmax(a, axis=1)
    return pd.DataFrame(...)

def _scan_peak_df(files, device):
    ...
    row_max_peak = np.nanmax(a, axis=1)
    return pd.DataFrame(...)

def _scan_bg_df(files, device):
    ...
    row_min_background = np.min(a_nozero, axis=1)
    return pd.DataFrame(...)

def _apply_scope_time(df, scope, use_time, t0, t1):
    ...
```

Die Funktionen `_scan_mean_df`, `_scan_peak_df` und `_scan_bg_df` lesen die Rohdaten direkt aus den HDF5-Dateien, berechnen daraus jeweils die relevanten Metriken und geben die Ergebnisse als DataFrame zurück. Die Funktion `_apply_scope_time` filtert die Daten nach Dateinamen (*scope*) und optional nach einem angegebenen Zeitintervall. Auf diese Weise werden die Abfragen auf denselben Szenarien wie bei den indexbasierten Varianten ausgeführt, jedoch ohne jegliche Indexstrukturen.

Ausführung der Abfragen für die Variante ohne Indexierung

Die letzten drei Notebook-Zellen der Variante ohne Indexierung entsprechen den letzten drei Zellen im Anhang A.4.1 und führen sämtliche in dieser Arbeit definierten Abfragen vollständig aus. Hierbei werden jedoch sämtliche Daten direkt aus den HDF5-Dateien gelesen und gefiltert, ohne dass zusätzliche Indexstrukturen genutzt werden. Dies entspricht einem vollständigen sequentiellen Scan der Daten.

Listing 4.11: Endgültige Ausführung der Abfragen ohne Indexierung

```
# Mean Range (0.1M .. 0.8M)
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        df_all = _scan_mean_df(TARGET_FILES, DEVICE)
        df_scope = _apply_scope_time(df_all, scope, use_time, T_START, T_END)
        ...
        df_res = ...
# Peak Top-N%
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        df_all = _scan_peak_df(TARGET_FILES, DEVICE)
        df_scope = _apply_scope_time(df_all, scope, use_time, T_START, T_END)
        ...
        df_res = ...
# Background Bottom-10%
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        df_all = _scan_bg_df(TARGET_FILES, DEVICE)
        df_scope = _apply_scope_time(df_all, scope, use_time, T_START, T_END)
        ...
        df_res = ...
```

Die drei vorgezeigten Schleifen zeigen, dass die Abfragen auf `row_max_mean`, `row_max_peak` und `row_min_background` vollständig ohne Indexstrukturen ausgeführt werden. Stattdessen erfolgt jedes Mal ein kompletter sequentieller Scan aller relevanten HDF5-Dateien. Dadurch lassen sich die Ergebnisse korrekt bestimmen, allerdings mit deutlich höherem Zeitaufwand im Vergleich zu indexbasierten Varianten. Durch die einheitliche Struktur der Notebooks mit wiederkehrenden Zellen (Extraktion, Aufbau der Datenbasis, Definition von Abfragen und deren Ausführung) konnte sichergestellt werden, dass alle Ansätze unter vergleichbaren Bedingungen evaluiert werden. Die prototypische Implementierung umfasst somit alle drei Varianten — externe Indexierung, interne Indexierung sowie den Zugriff ohne Indexierung — und bildet die Grundlage für die nachfolgende Evaluation. Im nächsten Kapitel 5 werden die entwickelten Konzepte anhand wiederholter Messungen systematisch getestet und hinsichtlich ihrer Effizienz verglichen.

Kapitel 5

Evaluation und Ergebnisse

In diesem Kapitel wird die externe 3.4 und interne 3.5 Indexierung sowie die Variante ohne Indexierung 3.6 evaluiert. Dazu werden zunächst die Szenarien beschrieben, anschließend die Messergebnisse präsentiert und schließlich die Resultate diskutiert.

5.1 Versuchsaufbau und Szenarien

Im Rahmen der Evaluation wurden eine Reihe von exemplarischen Abfragen auf den Messdaten durchgeführt, um die Leistungsfähigkeit der unterschiedlichen Indexierungsarten zu messen. Dabei wurden drei grundlegende Abfragearten ausgewählt: eine `Bereichsanfrage` auf der Spalte `row_max_mean`, wobei alle Werte zwischen 10% und 80% der `row_max_mean` berücksichtigt werden, eine `Top-N`-Abfrage auf der Spalte `row_max_peak` (oberste 10%), sowie eine `Bottom-N`-Abfrage auf der Spalte `row_min_background` (unterste 10%). Jeder dieser Abfragetypen wurde in vier unterschiedlichen Szenarien ausgeführt: einmal auf allen Dateien (`scope=all`), einmal mit einer Bedingung auf die Spalte (`filename`), jeweils mit und ohne zeitbasierte Einschränkung (`use_time=true/false`). Daraus ergibt sich für jede Abfrageart eine Kombination von vier Szenarien und insgesamt zwölf Evaluierungsfälle. Wesentlich ist, dass diese Abfragen nicht nur auf eine einzige Art der Indexierung angewandt wurden, sondern systematisch auf allen drei Varianten: der `internen Indexierung`, der `externen Indexierung` sowie der `ohne Indexierung`[1]. Dadurch entsteht ein vollständiger Vergleich, der die Unterschiede in Bezug auf Effizienz und Antwortzeit transparent macht.

Wiederholungen der Messungen Um belastbare und reproduzierbare Ergebnisse zu erhalten, wurden sämtliche Zeitmessungen in diesem Projekt konsequent mehrfach durchgeführt. Dies betrifft nicht nur die Ausführung der Abfragen in den verschiedenen Szenarien, sondern ebenso die Messungen für die Erstellung der Indizes sowie weitere zeitkritische Teilschritte. Jede Messung wurde insgesamt fünfmal hintereinander wiederholt. Anschließend wurde aus den erhaltenen Werten der arithmetische Mittelwert gebildet, der im Folgenden jeweils als Ergebnis angegeben wird. Durch diese Vorgehensweise lassen sich zufällige Schwankungen reduzieren, sodass die Resultate stabiler und zwischen den verschiedenen Ansätzen besser vergleichbar sind. Die Implementierung und Durchführung aller Messungen erfolgte in Python innerhalb einer Jupyter-Notebook-Umgebung unter Verwendung von Visual Studio Code.

[1] Die in der Evaluation betrachteten Abfragen umfassen implizit auch Punktanfragen und enthalten Aggregationen, da sowohl Extremwertabfragen (Top-N, Bottom-N) als auch Bereichsanfragen auf einer zugrundeliegenden Aggregation oder Selektion basieren.

5.2 Messungen und Ergebnisse

In diesem Abschnitt werden die Laufzeitmessungen vorgestellt, die sowohl die Erstellung der Tabellen als auch den Aufbau der Indizes sowie die Ausführung der Abfragen umfassen. Die Darstellung erfolgt getrennt nach den drei betrachteten Varianten der Indexierung. Die Resultate werden in Tabellen zusammengefasst. Die in den verschiedenen Szenarien ausgeführten Abfragen lieferten detaillierte Zeitwerte, die im Folgenden für die drei Abfragearten – Bereichsanfragen, Top-N-Abfragen und Bottom-N-Abfragen – einzeln dargestellt und anschließend vergleichend diskutiert werden.

Zu Beginn der Evaluation wurde die Zeit gemessen, die benötigt wird, um die relevanten Informationen aus den HDF5-Dateien zu extrahieren und in einem vollständigen `DataFrame` zusammenzuführen. Dabei ist zu berücksichtigen, dass die in der Evaluation verwendeten Attribute `row_max_mean`, `row_max_peak` und `row_min_background` nicht direkt in den Rohdaten vorliegen, sondern zuvor aus den ursprünglichen Messwerten berechnet werden. Da die Originaldaten für jede Messung vier separate Kanäle enthalten, ergeben sich für jedes Attribut vier Einzelwerte pro Zeile, aus denen im Zuge der Extraktion jeweils ein zusammenfassender Wert bestimmt wurde (Maximum für `Peak` und `Mean`, Minimum für `Background`). Dieser Verarbeitungsschritt war integraler Bestandteil der Extraktion und führte zu einem konsolidierten, kompakten Tabellenformat, das als Grundlage für die anschließende Indexerstellung und Abfrageverarbeitung dient. Die dafür erforderliche Laufzeit betrug insgesamt 2517.91 ms. Die folgende Tabelle 5.1 zeigt die Zeit für die Extraktion und den Aufbau eines vollständigen `DataFrame`.

Schritt	**Durchschnittszeit**
Extraktion und Aufbau eines vollständigen `DataFrame`	2517.91 ms

Tabelle 5.1: Laufzeit für die Extraktion der Rohdaten bei externer sowie interner Indexierung. Die detaillierten Wiederholungszeiten sind im Anhang (vgl. B.1) dargestellt.

Im Anschluss an diesen Vorbereitungsschritt mussten die extrahierten und reduzierten Informationen in einer übergreifenden Datenstruktur abgelegt werden. Im Fall der externen Indexierung erfolgte dies durch die Speicherung in einer kompakten Datenbank, während bei der internen Indexierung eine einzelne zusammengefasste HDF5-Datei mit reduzierten Inhalten erstellt wurde. Für die Variante ohne Indexierung hingegen blieb der erzeugte `DataFrame` ausschließlich im Arbeitsspeicher erhalten und wurde nicht in eine separate Struktur übertragen, sodass hierfür keine zusätzliche Laufzeitmessung anfällt. Die folgende Tabelle 5.2 gibt einen Überblick über die entsprechenden Aufwände.

Variante	**Ablage der reduzierten Daten**	**Durchschnittszeit**
Externe Indexierung	Erstellung der reduzierten Tabelle in SQLite	3195.63 ms
Interne Indexierung	Erstellung einer zusammengefassten HDF5-Datei	1285.17 ms

Tabelle 5.2: Laufzeiten für die Ablage der reduzierten Datenbasis in den beiden Indexierungsvarianten. Die detaillierten Wiederholungszeiten sind im Anhang (vgl. B.2) dargestellt.

Nach der Ablage der reduzierten Daten an den entsprechenden Stellen wurden darauf Indizes erstellt, um die Abfragen zu beschleunigen. Es wurde für jedes zentrale Attribut ein eigener Index erstellt. Konkret betrifft dies die Attribute `row_max_mean`, `row_max_peak` und `row_min_background`, da sie die Grundlage für die im Projekt relevanten Abfragen bilden. Im Fall der externen Indexierung wurden die Indizes in einer `SQLite`-Datenbank erzeugt, während sie bei der internen Indexierung direkt innerhalb der HDF5-Dateien abgelegt wurden. Die folgende Tabelle 5.3 fasst die durchschnittlichen Zeiten für die Erstellung der Indizes zusammen und verdeutlicht die Unterschiede zwischen externer und interner Indexierung.

Variante	Index für Peak	Index für Mean	Index für Background
Externe Indexierung	1334.52 ms	1247.35 ms	1190.25 ms
Interne Indexierung	138.85 ms	166.41 ms	128.12 ms

Tabelle 5.3: Durchschnittliche Zeiten für die Indexerstellung. Die detaillierten Wiederholungszeiten sind im Anhang (vgl. B.3) dargestellt.

Im Folgenden sind die Messergebnisse für die `Top-10%`-Abfragen auf der Spalte `row_max_peak` dargestellt. Die Tabelle 5.4 zeigt die durchschnittlichen Laufzeiten in allen vier Szenarien, jeweils getrennt nach externer Indexierung, interner Indexierung und der Variante ohne Indexierung

Peak Top-10%		Externe Indexierung	Interne Indexierung	Ohne Indexierung
Scope	use_time			
`all`	`false`	192.21 ms	15830.91 ms	1506.95 ms
`all`	`true`	296.06 ms	37382.25 ms	1493.93 ms
`filename`	`false`	18.47 ms	37700.16 ms	1518.76 ms
`filename`	`true`	20.99 ms	36604.56 ms	1520.71 ms

Tabelle 5.4: Durchschnittliche Laufzeiten der Top-10%-Abfragen auf `row_max_peak` in den unterschiedlichen Szenarien. Die detaillierten Wiederholungszeiten sind im Anhang (vgl. B.4) dargestellt.

Nachfolgend werden die Messergebnisse für die `Bereichsabfragen` auf der Spalte `row_max_mean` dargestellt. Das Intervall wurde als $0.1M$ bis $0.8M$ definiert, wobei M den größten Wert der Spalte `row_max_mean` bezeichnet. Damit umfasst die Abfrage alle Ergebnisse, die zwischen 10% und 80% des Maximums liegen. Die folgende Tabelle 5.5 fasst die durchschnittlichen Laufzeiten in allen vier Szenarien zusammen und ermöglicht einen direkten Vergleich zwischen externer Indexierung, interner Indexierung und der Variante ohne Indexierung

Mean 10%..80%		Externe Indexierung	Interne Indexierung	Ohne Indexierung
Scope	use_time			
`all`	`false`	1241.40 ms	21025.62 ms	1642.33 ms
`all`	`true`	109.16 ms	20439.09 ms	1499.89 ms
`filename`	`false`	16.02 ms	20126.77 ms	1527.42 ms
`filename`	`true`	18.64 ms	20178.35 ms	1536.91 ms

Tabelle 5.5: Durchschnittliche Laufzeiten der Bereichsabfragen auf `row_max_mean` im Werteintervall von 0.1M bis 0.8M (wobei M den globalen Maximalwert der Spalte `row_max_mean` bezeichnet). Die detaillierten Wiederholungszeiten sind im Anhang (vgl. B.5) dargestellt.

Abschließend werden die Messergebnisse für die `Bottom-10%`-Abfragen auf `row_min_background` dargestellt. Tabelle 5.6 zeigt die durchschnittlichen Laufzeiten in allen vier Szenarien und ermöglicht einen direkten Vergleich zwischen externer, interner und der Variante ohne Indexierung.

Background Bottom-10%		Externe Indexierung	Interne Indexierung	Ohne Indexierung
Scope	use_time			
`all`	`false`	620.68 ms	9976.39 ms	1560.01 ms
`all`	`true`	124.29 ms	8842.38 ms	1520.63 ms
`filename`	`false`	7.77 ms	8695.30 ms	1522.26 ms
`filename`	`true`	7.90 ms	8788.44 ms	1512.16 ms

Tabelle 5.6: Durchschnittliche Laufzeiten der Bottom-10%-Abfragen auf `row_min_background` in den unterschiedlichen Szenarien. Die detaillierten Wiederholungszeiten sind im Anhang (vgl. B.6) dargestellt.

Zusammenfassend lässt sich festhalten, dass die durchgeführten Messungen ein umfassendes Bild über die Laufzeiten der unterschiedlichen Indexierungsarten liefern. Die Ergebnisse verdeutlichen sowohl die Stärken als auch die Schwächen der einzelnen Ansätze und bilden damit die Grundlage für eine vertiefte Analyse. Im nächsten Abschnitt werden die Resultate vergleichend diskutiert, um daraus Schlussfolgerungen für die Eignung der jeweiligen Verfahren im Kontext des Projekts abzuleiten.

5.3 Herausforderungen bei der Integration dynamischer Datenstrukturen in HDF5

Die HDF5-Struktur zählt zu den am häufigsten verwendeten Formaten zur Speicherung großer wissenschaftlicher Datenmengen, da sie eine hohe Flexibilität im Umgang mit Array-Datasets sowie die Unterstützung hierarchischer Strukturen und Metadaten bietet. Dennoch bringt ihre Verwendung beim Aufbau dynamischer Zugriffstrukturen wie Bäumen oder Indizes mehrere grundlegende Herausforderungen mit sich. Die erste Herausforderung besteht in der statischen Natur der internen Struktur von HDF5; jedes Dataset wird mit festen Dimensionen und Datentypen erstellt, die nach der Anlage nicht mehr ohne Weiteres verändert werden können. Dies schränkt die Möglichkeit ein, dynamische Strukturen wie Knoten und Blätter eines B^+-Baums abzubilden, da solche Strukturen in der Regel flexible Mechanismen zum Hinzufügen von Elementen oder zum wiederholten Rebalancieren erfordern.

Die zweite Herausforderung liegt in den Einschränkungen der Leseoperationen, da HDF5 über seine Standardbibliothek h5py verlangt, dass Leseindizes (Indices) aufsteigend sortiert sein müssen. Das bedeutet, dass Zeilen oder Elemente nicht in beliebiger, unsortierter Reihenfolge gelesen werden können, was im Widerspruch zur Natur von Indizes steht, die oft den Zugriff auf verstreute Positionen in einer spezifischen Reihenfolge erfordern. Um diese Einschränkung zu umgehen, ist es notwendig, zusätzliche Hilfsfunktionen zu implementieren, die die Indizes vor dem Lesen sortieren und anschließend die Ergebnisse wieder in die ursprüngliche Reihenfolge zurückführen. Dies erhöht jedoch die rechnerische und zeitliche Komplexität.

Zusätzlich zwingt HDF5 den Benutzer zur Verwendung von Compound Datatypes fester Größe, wenn mehrere Elemente in einer Struktur gespeichert werden sollen. Dies führt zu Speicherverlusten (Padding) und vergrößert die Dateigröße, insbesondere wenn zusammengesetzte Schlüssel aus Text- und numerischen Daten abgelegt werden.

Vor dem Hintergrund dieser Herausforderungen lässt sich feststellen, dass HDF5 zwar in erster Linie für die effiziente Speicherung wissenschaftlicher Daten und deren sequentiellen Zugriff konzipiert ist, jedoch nicht von Natur aus zur Unterstützung komplexer dynamischer Indexstrukturen geeignet ist. Der Aufbau eines internen B^+-Baums innerhalb von HDF5 bleibt daher lediglich eine experimentelle Lösung (Prototyp) mit begrenzter Skalierbarkeit und verdeutlicht den Bedarf an externen oder hybriden Ansätzen zur Überwindung dieser Einschränkungen.

5.4 Diskussion

Die Messergebnisse werden interpretiert und hinsichtlich Effizienz verglichen. Besonderes Augenmerk liegt auf den Unterschieden zwischen externer und interner Indexierung sowie auf dem Vergleich zur Variante ohne Indexierung. Anhand der Darstellung der Laufzeiten für die Erstellung der reduzierten Datenbasis in Tabelle 5.2 folgt eine detaillierte Analyse der Ergebnisse: Die externe Indexierung weist mit durchschnittlich rund 3195.63 ms den höchsten Erstellungsaufwand auf. Dies lässt sich durch die aufwendigen Insert-Operationen in SQLite erklären (u. a. Transaktionsverwaltung, Logging und B^+-Baum-Updates). Im Gegensatz dazu benötigt die interne Indexierung mit 1285.17 ms deutlich weniger Zeit, da die Daten lediglich in einer zusammengefassten HDF5-Datei abgelegt werden, ohne zusätzliche Datenbankoperationen.

Analyse der Indexerstellung

Die in Tabelle 5.3 dargestellten Zeiten verdeutlichen, dass die externe Indexierung beim Aufbau der Indizes auf der reduzierten Datei bzw. Datenbank den höchsten Aufwand verursacht. Für die Metriken ergeben sich dabei Werte von 1334.52 ms für `row_max_peak`, 1247.35 ms für `row_max_mean` sowie 1190.25 ms für `row_min_background`. Dies liegt vor allem daran, dass hier zusammengesetzte Indizes gebildet werden, die nicht nur auf der jeweiligen Metrik, sondern zusätzlich auf den Attributen `time`, `filename` und `row_idx` basieren. Durch diese Mehrspalten-Indizes erhöht sich der Erstellungsaufwand erheblich, da die Datenbank die Daten über mehrere Dimensionen hinweg organisieren muss. Im Gegensatz dazu zeigt die interne Indexierung deutlich geringere Zeiten, nämlich 138.85 ms für `row_max_peak`, 166.41 ms für `row_max_mean` und 128.12 ms für `row_min_background`. Hier werden die B^+-Bäume direkt im Arbeitsspeicher aus vorbereiteten Arrays aufgebaut und anschließend als HDF5-Datasets gespeichert. Dabei wird eine lexikographische Sortierung über mehrere Attribute (Metrik[2], Zeit, Dateiname, Zeilenindex) verwendet, deren Umsetzung wesentlich effizienter erfolgt als die Generierung eines Mehrspalten-Index in SQLite.

Analyse der Top-10%-Abfragen auf `row_max_peak`

Die in Tabelle 5.4 dargestellten Ergebnisse verdeutlichen erhebliche Unterschiede zwischen den drei Ansätzen. Die externe Indexierung liefert durchgehend die besten Laufzeiten: Bei Abfragen über alle Dateien werden im Durchschnitt lediglich 192 ms ohne Zeitfilter und 296 ms mit Zeitfilter benötigt, während die Ausführung mit Bedingung auf `filename` sogar auf etwa 18–21 ms reduziert werden kann. Die interne Indexierung hingegen zeigt extrem hohe Laufzeiten zwischen 15830 ms und 37382 ms. Dies ist auf die zusätzlichen Komplexitäten der internen Indexierung im Zusammenhang mit HDF5 zurückzuführen. Zum einen stellt HDF5 von Haus aus keinen eigenen Query-Mechanismus bereit, sodass sämtliche Indexoperationen manuell über zusätzliche Datasets abgebildet werden müssen. Zum anderen existieren innerhalb von HDF5 keine echten Zeiger auf einzelne Tupel, weshalb die im B^+-Baum gespeicherten Verweise lediglich indirekt über Zeilenindizes oder Offsets aufgelöst werden können. Dies führt dazu, dass jeder Zugriff letztlich erneut über `h5py` erfolgen muss, was die Abfragezeiten deutlich erhöht. wodurch die Methode für Top-N-Abfragen ungeeignet erscheint. Die Variante ohne Indexierung erreicht konstante Laufzeiten von rund 1490–1520 ms, unabhängig davon, ob ein Zeitfilter aktiviert ist oder ob eine Bedingung auf `filename` vorliegt. Damit liegt sie zwar deutlich über den Zeiten der externen Indexierung, bleibt aber unter den Werten der internen Lösung.

Besondere Beobachtung im Peak-Top-10%-Szenario bei der externen und internen Indexierung

Auffällig ist, dass die Abfrage `Peak Top-10%` bei aktivierter Zeitbedingung eine höhere Laufzeit aufweist als ohne Zeitfilter. Der zentrale Grund hierfür liegt in der ermittelten Schwelle (Threshold) für die Top-10%, die im vorliegenden Fall den Wert 0 annahm. Dies bedeutet, dass der überwiegende Teil der Werte in `row_max_peak` gleich 0 ist. Somit weist der Datensatz einen ausgesprochen hohen Anteil an Nullwerten auf. Damit wird die Bedingung „`row_max_peak` ≥ 0“ praktisch nicht selektiv und schließt nahezu alle Tupel ein. Wird zusätzlich eine Zeitbedingung berücksichtigt, führt jedoch zu einem zusätzlichen Verarbeitungsaufwand im Datenbanksystem. Anstatt also die Laufzeit zu verringern, erhöht sich diese, da ein weiteres Prädikat ausgewertet werden muss, ohne dass ein nennenswerter Gewinn an Selektivität entsteht. Zusammenfassend lässt sich festhalten, dass die längeren Laufzeiten im Szenario `Peak all true` nicht auf einen Fehler zurückzuführen sind, sondern auf die geringe Selektivität der berechneten Schwelle in Kombination mit der zusätzlichen Filterbedingung.

[2]Die verwendete Metrik kann je nach Anwendungsfall `row_max_peak`, `row_max_mean` oder `row_min_background` sein.

Analyse der Bereichsabfragen auf `row_max_mean` (Mean 10%..80%)

Die in Tabelle 5.5 dargestellten Ergebnisse zeigen deutliche Unterschiede zwischen den drei Indexierungsstrategien bei Bereichsabfragen auf der Spalte `row_max_mean` im Intervall von (10%..80%). Für die externe Indexierung variieren die Laufzeiten erheblich in Abhängigkeit von den verwendeten Bedingungen. Im Fall `all false` beträgt die durchschnittliche Laufzeit rund 1241 ms, während sie bei aktivierter Zeitbedingung (`all true`) deutlich auf 109 ms reduziert wird. Diese starke Verbesserung ist darauf zurückzuführen, dass die Zeitbedingung den Suchraum erheblich einschränkt und der zusammengesetzte Index in SQLite dadurch effizienter genutzt werden kann. Noch schneller sind die Abfragen mit Bedingung auf `filename`, die mit 16–18 ms abgeschlossen werden, da sich der Suchraum hier auf eine einzige Datei beschränkt. Die interne Indexierung führt dagegen zu keinen spürbaren Verbesserungen. In allen Szenarien bewegen sich die Laufzeiten zwischen 20s und 21s, unabhängig davon, ob eine Zeitbedingung aktiviert ist oder ob die Abfrage auf `filename` eingeschränkt wird. Dies ist auf die zusätzlichen Komplexitäten beim Aufbau und bei der Nutzung der B^+-Bäume in HDF5 zurückzuführen, wodurch die Ausführung von Bereichsabfragen sehr zeitaufwendig bleibt. Bei ohne Indexierung bleiben die Laufzeiten nahezu konstant und liegen zwischen 1500 ms und 1600 ms. Der Grund dafür ist, dass die Ausführung auf einem vollständigen sequentiellen Scan der Daten basiert. Der größte Teil der Zeit wird für das Laden der Daten aus den HDF5-Dateien in den Speicher benötigt, was durch zusätzliche Bedingungen praktisch nicht reduziert wird. Zusammenfassend lässt sich feststellen, dass die externe Indexierung in diesem Szenario die mit Abstand effizienteste Strategie darstellt, während die interne Indexierung ungeeignet bleibt und die Ausführung ohne Indexierung lediglich eine mittlere, aber nicht konkurrenzfähige Leistung erbringt.

Analyse der Bottom-10%-Abfragen auf `row_min_background`

Die in Tabelle 5.6 dargestellten Ergebnisse verdeutlichen die Unterschiede zwischen den drei Indexierungsstrategien bei Bottom-10%-Abfragen auf der Spalte `row_min_background`. Die externe Indexierung zeigt hier durchgehend die besten Laufzeiten. Im Fall `all false` beträgt die durchschnittliche Laufzeit etwa 621 ms, während sie bei aktivierter Zeitbedingung (`all true`) deutlich auf 124 ms sinkt. Diese Verbesserung lässt sich dadurch erklären, dass die Zeitbedingung den Suchraum erheblich einschränkt und der zusammengesetzte Index in SQLite dadurch besonders effizient genutzt werden kann. Noch schneller sind die Abfragen mit Bedingung auf `filename`, die lediglich 7.8 ms benötigen, da der Suchraum auf eine einzige Datei begrenzt ist. Die interne Indexierung weist dagegen deutlich höhere Laufzeiten zwischen 8.6 s und 10 s auf, unabhängig davon, ob eine Zeitbedingung aktiviert ist oder ob die Abfrage auf `filename` eingeschränkt wird. Zwar sind diese Zeiten geringer als bei den Bereichsabfragen auf `row_max_mean` (ca. 20 s), dennoch bleibt die Methode sehr ineffizient. Der Hauptgrund liegt in den Restriktionen von HDF5 (vgl. Abschnitt 5.3): Obwohl B^+-Bäume theoretisch eine effiziente Unterstützung für Bereichsabfragen bieten, führt der Zugriff über h5py und die Struktur von HDF5 zu einem erheblichen Mehraufwand, wodurch das volle Potenzial der internen Indexierung nicht ausgeschöpft werden kann. Bei Ohne Indexierung bleiben die Laufzeiten nahezu konstant und liegen zwischen 1500 ms und 1560 ms. Dies ist darauf zurückzuführen, dass die Ausführung auf einem vollständigen sequentiellen Scan der Daten basiert. Der größte Teil der Zeit entfällt dabei auf das Laden der Daten aus den HDF5-Dateien in den Speicher, was durch zusätzliche Bedingungen kaum reduziert wird. Zusammenfassend lässt sich feststellen, dass die externe Indexierung auch in diesem Szenario die mit Abstand effizienteste Strategie darstellt. Die interne Indexierung leidet hingegen unter den durch HDF5 auferlegten Einschränkungen und bleibt daher ungeeignet, während die Ausführung ohne Indexierung eine mittlere, jedoch nicht konkurrenzfähige Leistung erbringt.

Allgemeine Beobachtungen

Ein Vergleich der Ergebnisse in Tabelle 5.5 und Tabelle 5.6 zeigt ein konsistentes Muster: Die externe Indexierung liefert in beiden Fällen die mit Abstand besten Laufzeiten, insbesondere wenn eine Zeitbedingung oder eine Einschränkung auf `filename` vorliegt. Die interne Indexierung bleibt hingegen in allen Szenarien deutlich langsamer. Obwohl die gemessenen Zeiten bei den `row_min_background`-Abfragen mit rund 9–10 s etwas niedriger ausfallen als bei den `row_max_mean`-Abfragen (ca. 20 s), bleibt die Methode aufgrund der durch HDF5 auferlegten Restriktionen ineffizient(vgl. Abschnitt 5.3). Die Variante ohne Indexierung zeigt in beiden Fällen konstante Laufzeiten um 1.5 s, was auf den dominanten Aufwand für den vollständigen sequentiellen Scan der HDF5-Dateien zurückzuführen ist. Zusammenfassend lässt sich feststellen, dass die externe Indexierung sowohl für Bereichsabfragen auf `row_max_mean` als auch für Abfragen auf `row_min_background` die einzig praktikable Lösung darstellt, während die interne Indexierung durch die HDF5-Beschränkungen stark limitiert bleibt.
Auffällig ist der Unterschied zu den `row_max_peak`-Abfragen: Hier führte die große Anzahl von Nullwerten in den Daten dazu, dass die berechnete Schwelle (Threshold) häufig den Wert 0 annahm, wodurch die Abfragen praktisch keine Selektivität aufwiesen und die Laufzeiten entsprechend ungenau gemacht wurden. Im Gegensatz dazu wurden bei den `row_min_background`-Abfragen die Nullen vor der Schwellenberechnung gefiltert[3], sodass dieses Problem nicht auftrat. Bei den `row_max_mean`-Abfragen spielte das Vorhandensein von Nullen ebenfalls keine Rolle, da sowohl die obere als auch die untere Grenze der Bereichsbedingung notwendigerweise auf Basis des globalen Maximums berechnet werden und daher niemals Null sein können.
Zusammenfassend lässt sich feststellen, dass die externe Indexierung in allen Szenarien die mit Abstand besten Laufzeiten erzielt und insbesondere bei einschränkenden Bedingungen wie Zeit oder `filename` ihre volle Stärke ausspielt. Die interne Indexierung hingegen weist aufgrund der durch HDF5 bedingten Restriktionen sehr hohe Laufzeiten auf und erweist sich damit in der praktischen Anwendung als ungeeignet. Die Variante ohne Indexierung zeigt zwar konstante Laufzeiten von etwa 1.5 s, erreicht jedoch bei weitem nicht die Effizienz der externen Indexierung.

[3] **Prof. Dr. Gerd Baumgarten**, einem der Betreuer dieser Arbeit, eingeführt und ausschließlich vor der Berechnung der unteren Schwelle für `row_min_background` angewandt. Dies war insofern sinnvoll, als dass ansonsten der Wert Null selbst als Grenzwert bestimmt worden wäre, was die Ergebnisse verfälscht und zu einer ähnlichen Problematik wie bei den `row_max_peak`-Abfragen geführt hätte.

Kapitel 6

Zusammenfassung und Ausblick

Dieses Kapitel fasst die zentralen Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Weiterentwicklungen und zukünftige Forschungsarbeiten. Ziel dieser Arbeit war die Entwicklung effizienter Zugriffsstrukturen für große Messdaten im HDF5-Format. Hierfür wurden mehr als zehn verschiedene Datenstrukturen untersucht und miteinander verglichen. Als besonders geeignet erwies sich dabei der B^+-Baum, dessen Integration im Rahmen dieser Arbeit auf zwei Arten betrachtet wurde: erstens durch eine externe Indexierung, die intern B^+-Bäume einsetzt, und zweitens durch eine interne Indexierung mit B^+-Bäumen direkt innerhalb von HDF5. Darüber hinaus wurde auch aufgezeigt, wie Abfragen ohne den Einsatz spezieller Indexstrukturen aussehen und welche Performanz sie im Vergleich erbringen. Die Ergebnisse verdeutlichen eindeutig, dass die externe Indexierung in allen Szenarien die schnellste und zugleich praktikabelste Lösung darstellt. Die interne Indexierung erwies sich dagegen aufgrund der technischen Restriktionen von HDF5 als ungeeignet, während Abfragen ohne Indexierung zwar nahezu konstante, jedoch insgesamt höhere Laufzeiten im Vergleich aufwiesen.

6.1 Schlussfolgerung

Auf Basis der durchgeführten Untersuchungen lässt sich festhalten, dass für das Leibniz-Institut für Atmosphärenphysik (IAP) die externe Indexierung insgesamt die praktikabelste und effizienteste Lösung darstellt. Sie ermöglicht kurze Antwortzeiten bei unterschiedlichen Abfragetypen und überwindet zugleich die durch HDF5 auferlegten Einschränkungen, die bei der internen Indexierung zu erheblichen Laufzeitproblemen geführt haben. Darüber hinaus hat sich gezeigt, dass die interne Indexierung aus mehreren Gründen nicht als optimale Lösung angesehen werden kann. Zum einen arbeitet das Institut typischerweise mit HDF5-Dateien von etwa 200 MB Größe, die jeweils nur Messdaten einer einzigen Minute enthalten. Selbst wenn für einzelne Dateien interne Indizes erstellt würden, könnten globale Anfragen über mehrere Dateien hinweg nur durch zusätzliche externe Strukturen oder den Einsatz von Drittsystemen effizient unterstützt werden. Zum anderen traten während der Arbeit erhebliche Herausforderungen und Komplexitäten beim Aufbau interner Indizes auf. Selbst in einem hypothetischen Szenario, in dem extrem große HDF5-Dateien über viele Jahre hinweg als ein einziges Datenobjekt gespeichert würden, wären die strukturellen Grenzen von HDF5 in Bezug auf Indexierung unübersehbar. Hinzu kommt, dass eine mehrere Petabyte große HDF5-Datei in der Praxis weder realistisch noch wartbar wäre, sodass eine derartige Lösung nicht nur enorme Schwierigkeiten bei der Aktualisierung einzelner Datenbereiche verursachen würde, sondern auch das Risiko birgt, dass im Falle einer Beschädigung die gesamte Datei unbrauchbar wird. Die externe Indexierung hingegen überwindet all diese Restriktionen und eröffnet zusätzliche Vorteile. So können beispielsweise komprimierte oder reduzierte Datenversionen gezielt auf spezialisierten Speichermedien abgelegt werden, die für schnelle Abfragen optimiert sind. Dadurch werden zwei Ziele gleichzeitig erreicht: Zum einen lassen sich Abfragen deutlich effizienter ausführen, und zum anderen entsteht eine

zusätzliche kompakte Sicherungskopie der Daten, die im Falle einer Beschädigung der Originaldateien als Backup genutzt werden kann. Insgesamt zeigt die Arbeit somit, dass die externe Indexierung nicht nur eine praktikable, sondern auch eine zukunftsfähige Lösung darstellt, die als Ausgangspunkt für weitere Forschungs- und Entwicklungsarbeiten im Bereich der effizienten Datenverarbeitung dienen kann.

6.2 Ausblick und zukünftige Arbeiten

Für die Zukunft sollten die durch HDF5 auferlegten Restriktionen 5.3 besonders berücksichtigt werden, um ein Indexierungskonzept zu entwickeln, das sich besser an die spezifischen Eigenschaften von HDF5 anpasst. Dabei erscheint es sinnvoll, alternative Indexstrukturen systematisch zu evaluieren und jene auszuwählen, die trotz der bestehenden Einschränkungen in HDF5 mit hoher Effizienz anwendbar sind. Darüber hinaus sollten komplexere Abfragearten untersucht werden, die sowohl höhere Dimensionalität als auch vielfältigere Szenarien abdecken. Ein weiterer wichtiger Schritt besteht darin, bereits vor der eigentlichen Ausführung von Abfragen geeignete Filtermethoden zu definieren und zu evaluieren, um zu vermeiden, dass flache bzw. wenig aussagekräftige Daten in die Ergebnisse einfließen. Dies würde die Effizienz der nachfolgenden Verarbeitung zusätzlich steigern. Die entwickelten Lösungen sollten zudem auf größere und heterogenere wissenschaftliche Datensätze angewandt werden, um ihre Skalierbarkeit und Praxistauglichkeit zu überprüfen. Da sich in den Experimenten gezeigt hat, dass die externe Indexierung die besten Ergebnisse liefert, wäre es zudem sinnvoll, in zukünftigen Arbeiten leistungsfähigere Datenbanksysteme als SQLite zu evaluieren. Dabei könnte untersucht werden, welche Systeme sich in Abhängigkeit von der Datenmenge, der Datenstruktur sowie den typischen Anfragearten am besten eignen. Schließlich könnte auch die Möglichkeit untersucht werden, die interne Struktur von HDF5 selbst zu erweitern oder anzupassen, um eine engere Integration mit leistungsfähigen Indexierungsverfahren zu ermöglichen.

Literaturverzeichnis

[Ben80] Bentley, Jon Louis: *Multidimensional Divide-and-Conquer*. Communications of the ACM, 23(4):214–229, 1980.

[Bro25] Brown, Russell A.: *Comparative Performance of the AVL Tree and Three Variants of the Red-Black Tree*. Software: Practice and Experience, 55(9):1607–1615, 2025.

[CLRS09] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Binary Search Trees*. In: *Introduction to Algorithms*, Seiten 286–308. MIT Press, Cambridge, MA, 3rd Auflage, 2009.

[HCZS24] Hu, Chuan, Jiawei Cai, Zihao Zhao und Zhihong Shen: *BIT: Using Bitmap Index to Speed Up NCBI Taxonomy Computing*. In: *Proceedings of the 36th International Conference on Scientific and Statistical Database Management (SSDBM)*, Seiten 1–12. ACM, 2024.

[Hib62] Hibbard, Thomas N.: *Some Combinatorial Properties of Certain Trees with Applications to Searching and Sorting*. Journal of the ACM, 9(1):13–28, 1962.

[HSS19] Heuer, Andreas, Gunter Saake und Kai-Uwe Sattler: *Datenbanken: Implementierungstechniken*. MITP-Verlag, 2019.

[LC88] Lewis, Ted G. und Curtis R. Cook: *Hashing for Dynamic and Static Internal Tables*. IEEE Computer, 21(10):45–56, Oktober 1988.

[Lea25] Leah A. Wasser: *Hierarchical Data Formats – What is HDF5?* `https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5`, 2025. Zugriff am 09.06.2025.

[NHS84] Nievergelt, J., H. Hinterberger und K. C. Sevcik: *The Grid File: An Adaptable, Symmetric Multikey File Structure*. ACM Transactions on Database Systems, 9(1):38–71, 1984.

[RG03] Ramakrishnan, Raghu und Johannes Gehrke: *Database Management Systems*. McGraw-Hill Higher Education, 3rd Auflage, 2003.

[Rob81] Robinson, J.: *The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes*. In: *Proceedings of the ACM SIGMOD Conference*, Seiten 10–18, Ann Arbor, MI, 1981.

[The06] The HDF Group: *The HDF Group* . `https://www.hdfgroup.org/`, 2006. Zugriff am 09.06.2025.

[The25a] The HDF Group: *HDF5 Dataset Structure – Illustration*, 2025. Zuletzt abgerufen am 25. Juni 2025.

[The25b] The HDF Group: *HDF5 Datatypes*. `https://support.hdfgroup.org/documentation/hdf5/latest/_l_b_datatypes.html`, 2025. Zugriff am 12. Juni 2025.

[The25c] The HDF Group: *HDF5 Group Structure – Illustration*, 2025. Zuletzt abgerufen am 25. Juni 2025.

[The25d] The HDF Group: *Introduction to HDF5*. https://support.hdfgroup.org/documentation/hdf5/latest/_intro_h_d_f5.html, 2025. Zugriff am 12. Juni 2025.

[The25e] The HDF Group: *Learn HDF5 Basics*. https://support.hdfgroup.org/documentation/hdf5/latest/_learn_basics.html, 2025. Zugriff am 12. Juni 2025.

Anhang A

Vollständiger Implementierungscode

In diesem Anhang wird der vollständige Python-Code der Implementierung aufgeführt. Im Hauptteil der Arbeit wurden die einzelnen Zellen lediglich auszugsweise oder in gekürzter Form dargestellt, um die Lesbarkeit des Fließtextes zu gewährleisten. Der hier bereitgestellte Quellcode dokumentiert sämtliche Varianten (externe Indexierung, interne Indexierung sowie die Verarbeitung ohne Indexierung) in vollständiger Länge und dient damit als Grundlage für die Reproduzierbarkeit und Nachvollziehbarkeit der Experimente. Darüber hinaus wird in diesem Anhang detailliert erläutert, wie die entwickelten Jupyter-Notebooks gestartet und verwendet werden können. Diese Anleitung richtet sich insbesondere an Endanwender ohne tiefergehende Informatikkenntnisse und beschreibt die einzelnen Schritte zur Installation, Ausführung und Nutzung der Notebooks im Detail.

A.1 Nutzung der Jupyter-Notebooks

Die im Rahmen dieser Arbeit entwickelten Prototypen wurden in Form von Jupyter-Notebooks implementiert. Damit auch Endanwender ohne Informatikhintergrund die Ausführung nachvollziehen können, werden im Folgenden die notwendigen Schritte detailliert beschrieben.

Systemanforderungen

- Installiertes Python in der Version 3.11.9 (oder kompatibel).
- Installierte Entwicklungsumgebung Visual Studio Code.
- Installierte Jupyter-Erweiterung innerhalb von Visual Studio Code.

Start der Umgebung

1. Öffnen Sie Visual Studio Code.
2. Laden Sie den Projektordner, in dem die Jupyter-Notebooks abgelegt sind.
3. Stellen Sie sicher, dass die Python-Umgebung (Interpreter) in Visual Studio Code korrekt ausgewählt ist (rechts oben in der Menüleiste).
4. Öffnen Sie die gewünschte Notebook-Datei mit der Endung `.ipynb`.
5. Installation der benötigten Python-Bibliotheken. Dazu kann entweder die Kommandozeile genutzt werden oder eine leere Zelle im Jupyter-Notebook erstellt werden, in die folgender Befehl eingefügt und ausgeführt wird:

```
!pip install h5py==3.14.0 numpy==2.2.6 pandas==2.3.1
```

Dieser Befehl sorgt dafür, dass alle benötigten Bibliotheken automatisch in der richtigen Version installiert werden.

Ausführung der Notebooks

1. Jupyter-Notebooks bestehen aus sogenannten *Zellen*. Jede Zelle enthält entweder Text (Erklärungen) oder Python-Code.
2. Um eine Zelle auszuführen, klicken Sie in die Zelle und drücken `Shift + Enter`.
3. Es wird dringend empfohlen, **alle Zellen in der vorgesehenen Reihenfolge von oben nach unten** auszuführen. Dadurch wird sichergestellt, dass alle Variablen und Zwischenergebnisse korrekt erzeugt werden. Ein Überspringen oder Ausführen in falscher Reihenfolge kann zu Fehlern führen.
4. Nach der Ausführung jeder Zelle erscheinen die Ergebnisse direkt darunter, sodass die Berechnungsschritte leicht nachvollziehbar sind.

A.2 Externe Indexierung

Listing A.1: Externe Indexierung – Zelle 1

```
#%% ================================================================
# (1) Importe + Einstellungen + Hilfsfunktionen fuer die Extraktion
#=================================================================
import h5py
import os, glob, re, time as _time, datetime, sqlite3, socket
import numpy as np
import pandas as pd
import subprocess
from pathlib import Path
pd.set_option("display.float_format", lambda x: f"{x:.6f}")
DATA_DIR    = r"C:\Atmospherdaten"
GLOB_PAT    = os.path.join(DATA_DIR, "*.h5")
RESULTS_DIR = os.path.join(DATA_DIR, "results")
Path(RESULTS_DIR).mkdir(parents=True, exist_ok=True)
ZERO_AS_NAN = False
HOSTNAME    = socket.gethostname()
RUNSTAMP    = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
DEVICES_TO_PROCESS = [0]
#DEVICES_TO_PROCESS = [0,1]
_name_ts_regex = re.compile(r"LISA_(\d{8})_(\d{6})\.h5$", re.IGNORECASE)
def parse_ts_from_name(path_or_name):
    bn = os.path.basename(path_or_name)
    m = _name_ts_regex.search(bn)
    if not m:
        return None
    try:
        return datetime.datetime.strptime(m.group(1) + m.group(2), "%Y%m%d%H%M%S")
    except Exception:
        return None
def _parse_dt_loose(s):
    if s is None:
        return None
    s = s.strip()
    for fmt in ("%Y-%m-%d %H:%M:%S", "%Y-%m-%d %H:%M"):
        try:
```

```
            dt = datetime.datetime.strptime(s, fmt)
            if fmt.endswith("%H:%M"):
                dt = dt.replace(second=0, microsecond=0)
            return dt
        except Exception:
            pass
    raise ValueError(f"Nicht unterstuetztes Datum/Zeit-Format: {s!r}")
def _floor_to_minute(dt):
    return dt.replace(second=0, microsecond=0)
def _coerce_float_no_thousands(series):
    if np.issubdtype(series.dtype, np.number):
        return series.astype("float64")
    s = series.astype(str).str.replace(" ", "")
    s = s.str.replace(".", "", regex=False).str.replace(",", ".", regex=False)
    return pd.to_numeric(s, errors="coerce").astype("float64")
def _safe_read(h5f, path):
    return h5f[path][:] if path in h5f else None
```

Listing A.2: Externe Indexierung – Zelle 2.5

```


#%% ====================================================================
# (2.5) Select HDF5 files to be processed
#=====================================================================
SELECTION_MODE = "all"     # "all" | "range" | "match"
#RANGE_START = "2024-10-08 07:21:00"
#RANGE_END   = "2024-10-08 07:24:00"
MATCH_GLOBS = [
    # "LISA_20241008_0721*.h5",
    # "LISA_20241008_08*.h5",
]
_all_files = sorted(glob.glob(GLOB_PAT))
_selected = []
if SELECTION_MODE == "all":
    _selected = _all_files
elif SELECTION_MODE == "match":
    pooled = []
    for pat in MATCH_GLOBS:
        pooled += glob.glob(os.path.join(DATA_DIR, pat))
    _selected = list(dict.fromkeys(sorted(pooled)))
elif SELECTION_MODE == "range":
    pairs = []
    for fp in _all_files:
        ts = parse_ts_from_name(fp)
        if ts: pairs.append((ts, fp))
    pairs.sort(key=lambda x: x[0])
    if pairs:
        start_dt = _parse_dt_loose(RANGE_START) if RANGE_START else pairs[0][0]
        end_dt   = _parse_dt_loose(RANGE_END)   if RANGE_END   else pairs[-1][0]
        start_floor, end_floor = _floor_to_minute(start_dt), _floor_to_minute(end_dt)
        if end_floor < start_floor:
            raise ValueError(" Ungueltiger Zeitraum.")
        _selected = [fp for ts, fp in pairs if start_floor <= _floor_to_minute(ts) <=
        end_floor]
    else:
        _selected = []
else:
    print(f"[WARN] Auswahlmodus nicht bekannt: {SELECTION_MODE!r} -> all")
    _selected = _all_files
TARGET_FILES = _selected
print("\n[Ausgewaehlte Dateien")
print("Verzichnis:", DATA_DIR)
print("Anzahl der in der Verzeichniss verfuegbare HDF5 Dateien", len(_all_files))
print("Anzahl der ausgewaehlte HDF5 Dateien :", len(TARGET_FILES))
```

```
for i, fp in enumerate(TARGET_FILES[:10], 1):
    print(f"  {i:>2}. {os.path.basename(fp)}")
if not TARGET_FILES:
    raise RuntimeError(" no selected Files")
sel_list_path = os.path.join(RESULTS_DIR, f"selected_files_{RUNSTAMP}.txt")
with open(sel_list_path, "w", encoding="utf-8") as f:
    for fp in TARGET_FILES:
        f.write(fp + "\n")
print("[INFO] Files List :", sel_list_path)
```

Listing A.3: Externe Indexierung – Zelle 3

```

#%% ================================================================
# (3) Extraktion der Wichtigsten Daten aus der HDF5
#==================================================================
def _prep_matrix_keep_zeros(arr, n, cols=4):
    if arr is None:
        return np.full((n, cols), np.nan, dtype="float64")
    a = arr[:n].astype("float64")
    if a.ndim == 1:
        a = a.reshape(-1, 1)
    if a.shape[1] > cols:
        a = a[:, :cols]
    elif a.shape[1] < cols:
        pad = np.full((a.shape[0], cols - a.shape[1]), np.nan, dtype="float64")
        a = np.concatenate([a, pad], axis=1)
    return a
def extract_index_for_device(h5_path, device):
    d = device
    sfx = str(d)
    with h5py.File(h5_path, "r") as f:
        t  = _safe_read(f, f"/LISA/{d}/time")
        pk = _safe_read(f, f"/LISA/{d}/peak")
        bg = _safe_read(f, f"/LISA/{d}/background")
        mn = _safe_read(f, f"/LISA/{d}/mean")
        r  = _safe_read(f, f"/BeamStabilizer/{d}/RMSD")
    if t is None:
        raise RuntimeError(f"time Spalte Fehlt: /LISA/{d}/time in {h5_path}")
    lens = [len(t)]
    for arr in (pk, bg, mn, r):
        if arr is not None:
            lens.append(len(arr))
    n = min(lens)
    t = t[:n].astype("float64")
    r = r[:n].astype("float64") if r is not None else np.full(n, np.nan, dtype="float64")
    pk = _prep_matrix_keep_zeros(pk, n)
    mn = _prep_matrix_keep_zeros(mn, n)
    bg = _prep_matrix_keep_zeros(bg, n)
    row_max_peak = np.nanmax(pk, axis=1)
    bg_nozero = np.where(bg == 0, np.inf, bg)
    row_min_background = np.min(bg_nozero, axis=1)
    row_min_background = np.where(row_min_background == np.inf, np.nan, row_min_background)
    row_max_mean = np.nanmax(mn, axis=1)
    df = pd.DataFrame({
        f"time{sfx}": t,
        "filename": os.path.basename(h5_path),
        f"row_idx_in_file{sfx}": np.arange(n, dtype=int),
        f"RMSD{sfx}": r,
        f"row_max_peak{sfx}": row_max_peak,
        f"row_min_background{sfx}": row_min_background,
        f"row_max_mean{sfx}": row_max_mean,
    })
    df[f"valid_peak{sfx}"] = (~np.isnan(row_max_peak)).astype(int)
    df[f"valid_bg{sfx}"]   = (~np.isnan(row_min_background)).astype(int)
```

```
    df[f"valid_mean{sfx}"] = (~np.isnan(row_max_mean)).astype(int)
    for col in [f"time{sfx}", f"RMSD{sfx}", f"row_max_peak{sfx}", f"row_min_background{sfx}",
    f"row_max_mean{sfx}"]:
        df[col] = _coerce_float_no_thousands(df[col])
    return df



# Daten pro Geraet sammeln (ohne Vermischung)
t0 = _time.perf_counter()
dfs_by_device = {d: [] for d in DEVICES_TO_PROCESS}
print(f"[INFO] Processing {len(TARGET_FILES)} Files x {len(DEVICES_TO_PROCESS)} Device/Devices
.")
for fp in TARGET_FILES:
    for d in DEVICES_TO_PROCESS:
        try:
            dfs_by_device[d].append(extract_index_for_device(fp, d))
        except Exception as e:
            print(f"[WARN] Uebersprungen {os.path.basename(fp)} fuer das Geraet {d}: {e}")

df_all_by_device = {
    d: (pd.concat(dfs_by_device[d], ignore_index=True) if dfs_by_device[d] else pd.DataFrame()
    )
    for d in DEVICES_TO_PROCESS
}
for d in DEVICES_TO_PROCESS:
    if df_all_by_device[d].empty:
        print(f"[WARN] Keine Daten fuer das Geraet{d}.")

elapsed_ms = (_time.perf_counter() - t0) * 1000
print(f"[INFO] Extraktions- und Zusammenfuehrungszeit: {elapsed_ms:.2f} ms")
```

Listing A.4: Externe Indexierung – Zelle 4

```

#%% ================================================================
# (4) Erstellung von SQLite mit Wiederholungen zur Zeitmessung + Speichern eines Berichts
#==================================================================


def write_sqlite_for_device_with_repeats(df, device, repeats=1):
    if df.empty:
        print(f"[INFO] SQLite fuer Geraet {device} uebersprungen (keine Daten).")
        return None
    s = str(device)
    base   = f"index_dev{device}_{HOSTNAME}_{datetime.datetime.now().strftime('%Y%m%d_%H%M%S_%
    f')}"
    db_path = os.path.join(RESULTS_DIR, base + ".db")
    log_path = os.path.join(RESULTS_DIR, base + "_timing.txt")
    times_table = []
    times_idx_peak = []
    times_idx_bg = []
    times_idx_mean = []
    results_log = []
    results_log.append(f"Zeitbericht fuer Geraet dev{s}, Anzahl Wiederholungen = {repeats}")
    results_log.append("="*60)
    df_bulk = pd.DataFrame({
        "filename": df["filename"].astype(str),
        "row_idx":  df[f"row_idx_in_file{s}"].astype(int),
        "time":     pd.to_numeric(df[f"time{s}"], errors="coerce"),
        "row_max_peak":       pd.to_numeric(df[f"row_max_peak{s}"], errors="coerce"),
        "row_min_background": pd.to_numeric(df[f"row_min_background{s}"], errors="coerce"),
        "row_max_mean":       pd.to_numeric(df[f"row_max_mean{s}"], errors="coerce"),
        "RMSD":     pd.to_numeric(df[f"RMSD{s}"], errors="coerce"),
        "valid_peak": df[f"valid_peak{s}"].astype(int),
```

```
31          "valid_bg":   df[f"valid_bg{s}"].astype(int),
32          "valid_mean": df[f"valid_mean{s}"].astype(int),
33      })
34      miss = int(df_bulk["time"].isna().sum())
35      if miss:
36          warn_msg = f"[WARN] dev{s}: {miss} Zeilen verworfen wegen ungueltiger time."
37          print(warn_msg)
38          results_log.append(warn_msg)
39          df_bulk = df_bulk.dropna(subset=["time"])
40      with sqlite3.connect(db_path, timeout=10) as conn:
41          conn.executescript("""
42          PRAGMA journal_mode = WAL;
43          PRAGMA synchronous  = NORMAL;
44          PRAGMA busy_timeout = 30000;
45          """)
46          for rep in range(1, repeats+1):
47              results_log.append(f"\n--- Wiederholung Nr. {rep} ---")
48              t0 = _time.perf_counter()
49              conn.executescript(f"""
50              DROP TABLE IF EXISTS entries_dev{s};
51              CREATE TABLE entries_dev{s}(
52                filename            TEXT    NOT NULL,
53                row_idx             INTEGER NOT NULL,
54                time                REAL    NOT NULL,
55                row_max_peak        REAL,
56                row_min_background  REAL,
57                row_max_mean        REAL,
58                RMSD                REAL,
59                valid_peak          INTEGER NOT NULL,
60                valid_bg            INTEGER NOT NULL,
61                valid_mean          INTEGER NOT NULL,
62                PRIMARY KEY (filename, row_idx)
63              );
64              """)
65              conn.executemany(
66                  f"""INSERT INTO entries_dev{s}
67                  (filename,row_idx,time,row_max_peak,row_min_background,row_max_mean,
68                   RMSD,valid_peak,valid_bg,valid_mean)
69                  VALUES (?,?,?,?,?,?,?,?,?,?)""",
70                  df_bulk[["filename","row_idx","time","row_max_peak","row_min_background","
                    row_max_mean",
71                           "RMSD","valid_peak","valid_bg","valid_mean"]].itertuples(index=False,
                              name=None)
72              )
73              conn.commit()
74              dt_table = (_time.perf_counter()-t0)*1000
75              times_table.append(dt_table)
76              line = f"[rep {rep}] Tabelle erstellt + Einfuegen: {dt_table:.2f} ms"
77              print(line); results_log.append(line)
78              def create_index(sql, label, collector):
79                  t1 = _time.perf_counter()
80                  conn.execute(sql); conn.commit()
81                  dt = (_time.perf_counter()-t1)*1000
82                  collector.append(dt)
83                  msg = f"[rep {rep}] Index {label} erstellt: {dt:.2f} ms"
84                  print(msg); results_log.append(msg)
85              create_index(f"""
86              CREATE INDEX IF NOT EXISTS idx_peak_time_dev{s}
87                ON entries_dev{s}(row_max_peak DESC, time, filename, row_idx)
88                WHERE row_max_peak IS NOT NULL AND valid_peak=1;
89              """, "Peak", times_idx_peak)
90              create_index(f"""
91              CREATE INDEX IF NOT EXISTS idx_bg_time_dev{s}
92                ON entries_dev{s}(row_min_background ASC, time, filename, row_idx)
```

```
              WHERE row_min_background IS NOT NULL AND row_min_background != 0 AND valid_bg=1;
            """, "Background", times_idx_bg)
            create_index(f"""
            CREATE INDEX IF NOT EXISTS idx_mean_time_dev{s}
              ON entries_dev{s}(row_max_mean ASC, time, filename, row_idx)
              WHERE row_max_mean IS NOT NULL AND valid_mean=1;
            """, "Mean", times_idx_mean)
        results_log.append("\n=== Arithmetische Mittelwerte ===")
        avg_table = sum(times_table)/len(times_table)
        avg_peak  = sum(times_idx_peak)/len(times_idx_peak)
        avg_bg    = sum(times_idx_bg)/len(times_idx_bg)
        avg_mean  = sum(times_idx_mean)/len(times_idx_mean)
        results_log.append(f"[AVG] Tabelle erstellt + Einfuegen: {avg_table:.2f} ms")
        results_log.append(f"[AVG] Index Peak: {avg_peak:.2f} ms")
        results_log.append(f"[AVG] Index Background: {avg_bg:.2f} ms")
        results_log.append(f"[AVG] Index Mean: {avg_mean:.2f} ms")
        print("\n=== Arithmetische Mittelwerte ===")
        print(f"[AVG] Tabelle erstellt + Einfuegen: {avg_table:.2f} ms")
        print(f"[AVG] Index Peak: {avg_peak:.2f} ms")
        print(f"[AVG] Index Background: {avg_bg:.2f} ms")
        print(f"[AVG] Index Mean: {avg_mean:.2f} ms")
        preview = pd.read_sql_query(f"SELECT * FROM entries_dev{s} LIMIT 10;", conn)
        print(f"\n[Preview] Erste 10 Zeilen:")
        print(preview)
        results_log.append("\n[Preview] Erste 10 Zeilen:")
        results_log.append(preview.to_string(index=False, float_format="%.6f"))
    with open(log_path, "w", encoding="utf-8") as f:
        for line in results_log:
            f.write(line + "\n")
    print(f"[OK] SQLite (dev{device}): {db_path}")
    print(f"[PATH] Zeitbericht gespeichert in: {os.path.abspath(log_path)}")
    return db_path, log_path, avg_table, avg_peak, avg_bg, avg_mean

db_paths = []
for d, df in df_all_by_device.items():
    res = write_sqlite_for_device_with_repeats(df, d, repeats=1)
    if res:
        db_path, log_path, avg_table, avg_peak, avg_bg, avg_mean = res
        db_paths.append(db_path)
ACTIVE_DB = db_paths[-1] if db_paths else None
print("[INFO] Neueste Datenbank:", ACTIVE_DB)


def clear_sqlite_cache(db_path):
    try:
        with sqlite3.connect(db_path, timeout=5) as conn:
            conn.execute("PRAGMA shrink_memory;")
            conn.execute("PRAGMA optimize;")
            conn.commit()
    except Exception:
        pass
```

Listing A.5: Externe Indexierung – Zelle 5

```

#%% ================================================================
# (5) Allgemeine Einstellungen fuer die Abfragen
#=====================================================================
DEVICE   = 0
T_START  = 1728372066.996737
T_END    = 1728372119.978274
REPEATS  = 1
PERCENT  = 10.0
SCOPES = ["all","LISA_20241008_072100.h5"]
TIME_SCENARIOS = [False,True]
```

```
def _table_name_for_device(d):
    return f"entries_dev{d}"
SAVE_TO_TXT = True
if SAVE_TO_TXT:
    ts_now = datetime.datetime.now().strftime("%Y%m%d_%H%M%S_%f")
    TXT_PATH = os.path.join(RESULTS_DIR,f"index_dev{DEVICE}_{HOSTNAME}_{ts_now}_queries.txt")
    TXT_LOG = []
    try:
        TXT_LOG.append(f"[INFO] Durchschnittliche Zeit zur Erstellung der Tabelle in SQLlite:
        {avg_table:.2f} ms")
        TXT_LOG.append(f"[INFO] Durchschnittliche Zeit zur Erstellung eines Index auf Peak: {
        avg_peak:.2f} ms")
        TXT_LOG.append(f"[INFO] Durchschnittliche Zeit zur Erstellung eines Index auf
        Background: {avg_bg:.2f} ms")
        TXT_LOG.append(f"[INFO] Durchschnittliche Zeit zur Erstellung eines Index auf Mean: {
        avg_mean:.2f} ms")
    except NameError:
        TXT_LOG.append("[WARN] Zelle (4) wurde noch nicht ausgefuehrt -> keine
        Durchschnittszeiten fuer die Erstellung vorhanden.")
else:
    TXT_PATH = None
    TXT_LOG = []
dbp = ACTIVE_DB
table = _table_name_for_device(DEVICE)
```

Listing A.6: Externe Indexierung – Zelle 5.2

```

#%% ======================================================================
# (5.2) Definition der Hilfsfunktionen fuer die Abfragen
#======================================================================
def _compute_M_for_scope(conn, table, scope, use_time, t0, t1):
    base = f"""
        SELECT MAX(row_max_mean) AS M
        FROM {table}
        WHERE row_max_mean IS NOT NULL AND valid_mean=1
    """
    conds, params = [], []
    if scope != "all":
        conds.append("filename = ?")
        params.append(scope)
    if use_time:
        conds.append("time BETWEEN ? AND ?")
        params.extend([t0, t1])
    if conds:
        base += " AND " + " AND ".join(conds)
    return pd.read_sql_query(base, conn, params=params)["M"].iloc[0]
def mean_range_query(conn, table, scope, use_time, t0, t1):
    M = _compute_M_for_scope(conn, table, scope, use_time, t0, t1)
    if M is None or np.isnan(M):
        return pd.DataFrame(), None
    q = f"""
        SELECT filename, row_idx, time, row_max_mean
        FROM {table}
        WHERE row_max_mean BETWEEN ? AND ? AND valid_mean=1
    """
    params = [0.1 * M, 0.8 * M]
    if scope != "all":
        q += " AND filename = ?"
        params.append(scope)
    if use_time:
        q += " AND time BETWEEN ? AND ?"
        params.extend([t0, t1])
    q += " ORDER BY row_max_mean ASC"
    df = pd.read_sql_query(q, conn, params=params)
```

```
39      return df, M
40  def _threshold_for_top_percent(conn, table, col, percent, scope, use_time, t0, t1):
41      base = f"SELECT {col} FROM {table} WHERE {col} IS NOT NULL AND valid_peak=1"
42      conds, params = [], []
43      if scope != "all":
44          conds.append("filename = ?")
45          params.append(scope)
46      if use_time:
47          conds.append("time BETWEEN ? AND ?")
48          params.extend([t0, t1])
49      if conds:
50          base += " AND " + " AND ".join(conds)
51      cnt = pd.read_sql_query(f"SELECT COUNT(*) AS c FROM ({base})", conn, params=params)["c"].
        iloc[0]
52      if not cnt:
53          return None
54      k = max(int(np.floor((percent/100.0) * cnt)) - 1, 0)
55      q = f"{base} ORDER BY {col} DESC LIMIT 1 OFFSET ?"
56      df = pd.read_sql_query(q, conn, params=params + [k])
57      return df[col].iloc[0] if not df.empty else None
58  def top_percent_peak(conn, table, percent, scope, use_time, t0, t1):
59      thr = _threshold_for_top_percent(conn, table, "row_max_peak", percent, scope, use_time, t0
        , t1)
60      if thr is None or np.isnan(thr):
61          return pd.DataFrame(), None
62      q = f"""
63          SELECT filename, row_idx, time, row_max_peak
64          FROM {table}
65          WHERE row_max_peak >= ? AND valid_peak=1
66      """
67      params = [thr]
68      if scope != "all":
69          q += " AND filename = ?"
70          params.append(scope)
71      if use_time:
72          q += " AND time BETWEEN ? AND ?"
73          params.extend([t0, t1])
74      q += " ORDER BY row_max_peak DESC"
75      return pd.read_sql_query(q, conn, params=params), thr
76  def _threshold_for_bottom_percent_bg(conn, table, col, percent, scope, use_time, t0, t1):
77      base = f"""
78          SELECT {col}
79          FROM {table}
80          WHERE {col} IS NOT NULL AND {col} != 0 AND valid_bg=1
81      """
82      conds, params = [], []
83      if scope != "all":
84          conds.append("filename = ?")
85          params.append(scope)
86      if use_time:
87          conds.append("time BETWEEN ? AND ?")
88          params.extend([t0, t1])
89      if conds:
90          base += " AND " + " AND ".join(conds)
91      cnt = pd.read_sql_query(f"SELECT COUNT(*) AS c FROM ({base})", conn, params=params)["c"].
        iloc[0]
92      if not cnt:
93          return None
94      k = max(int(np.floor((percent/100.0) * cnt)) - 1, 0)
95      q = f"{base} ORDER BY {col} ASC LIMIT 1 OFFSET ?"
96      df = pd.read_sql_query(q, conn, params=params + [k])
97      return df[col].iloc[0] if not df.empty else None
98  def bottom_percent_bg(conn, table, percent, scope, use_time, t0, t1):
99      thr = _threshold_for_bottom_percent_bg(conn, table, "row_min_background", percent, scope,
```

```
    use_time, t0, t1)
    if thr is None or np.isnan(thr):
        return pd.DataFrame(), None
    q = f"""
        SELECT filename, row_idx, time, row_min_background
        FROM {table}
        WHERE row_min_background IS NOT NULL
          AND row_min_background != 0
          AND row_min_background <= ? AND valid_bg=1
    """
    params = [thr]
    if scope != "all":
        q += " AND filename = ?"
        params.append(scope)
    if use_time:
        q += " AND time BETWEEN ? AND ?"
        params.extend([t0, t1])
    q += " ORDER BY row_min_background ASC"
    return pd.read_sql_query(q, conn, params=params), thr
```

A.2.1 Externe Indexierungsabfragen

Listing A.7: Externe Indexierung – Zelle 5.5

```
#%% ================================================================
# (5.5) Abfragezelle   Mean Range  10\% bis 80\%
#================================================================
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        header = f"\n[INFO] Mean 0.1M..0.8M | SCOPE={scope}, USE_TIME={use_time}"
        print(header)
        if SAVE_TO_TXT: TXT_LOG.append(header)
        times = []
        df_example = None
        for rep in range(1, REPEATS + 1):
            clear_sqlite_cache(dbp)
            with sqlite3.connect(dbp) as conn:
                t0 = _time.perf_counter()
                df, M = mean_range_query(conn, table, scope, use_time, T_START, T_END)
                dt = (_time.perf_counter() - t0) * 1000
                times.append(dt)
                line = (f"[Mean][scope={scope}][use_time={use_time}][rep {rep}] rows={len(df)}
                  M={M}   time={dt:.2f} ms")
                print(line)
                if SAVE_TO_TXT: TXT_LOG.append(line)
                if rep == 1:
                    df_example = df
        avg_time = sum(times) / len(times)
        avg_line = (f"[AVG][Mean][scope={scope}][use_time={use_time}] Durchschnittliche Zeit
        ueber {REPEATS} Wiederholungen = {avg_time:.2f} ms")
        print(avg_line)
        if SAVE_TO_TXT: TXT_LOG.append(avg_line)
        if df_example is not None and not df_example.empty:
            print(df_example.head(10))
            if SAVE_TO_TXT:
                case_name = f"mean_scope={scope}_use_time={use_time}"
                case_file = os.path.join(RESULTS_DIR,f"index_dev{DEVICE}_{HOSTNAME}_{datetime.
                datetime.now().strftime('%Y%m%d_%H%M%S_%f')}_{case_name}.txt")
                with open(case_file, "w", encoding="utf-8") as f:
                    f.write(df_example.to_string(index=False, float_format='%.6f'))
                print(f"[PATH] Ergebnisse vollstaendig gespeichert: {os.path.abspath(case_file
                )}")
```

Listing A.8: Externe Indexierung – Zelle 6

```
#%% ================================================================
# (6) Abfrage Peak Top-N%
#================================================================
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        header = f"\n[INFO] Peak Top-{PERCENT:.0f}% | SCOPE={scope}, USE_TIME={use_time}"
        print(header)
        if SAVE_TO_TXT: TXT_LOG.append(header)
        times = []
        df_example = None
        for rep in range(1, REPEATS + 1):
            clear_sqlite_cache(dbp)
            with sqlite3.connect(dbp) as conn:
                t0 = _time.perf_counter()
                df, thr = top_percent_peak(conn, table, PERCENT, scope, use_time, T_START,
                T_END)
                dt = (_time.perf_counter() - t0) * 1000
                times.append(dt)
                line = (f"[Peak][scope={scope}][use_time={use_time}][rep {rep}] rows={len(df)}
                  thr={thr}   time={dt:.2f} ms")
                print(line)
                if SAVE_TO_TXT: TXT_LOG.append(line)
                if rep == 1:
                    df_example = df
        avg_time = sum(times) / len(times)
        avg_line = (f"[AVG][Peak][scope={scope}][use_time={use_time}] Durchschnittliche Zeit
        ueber {REPEATS} Wiederholungen = {avg_time:.2f} ms")
        print(avg_line)
        if SAVE_TO_TXT: TXT_LOG.append(avg_line)
        if df_example is not None and not df_example.empty:
            print(df_example.head(10))
            if SAVE_TO_TXT:
                case_name = f"peak_scope={scope}_use_time={use_time}"
                case_file = os.path.join(RESULTS_DIR,f"index_dev{DEVICE}_{HOSTNAME}_{datetime.
                datetime.now().strftime('%Y%m%d_%H%M%S_%f')}_{case_name}.txt")
                with open(case_file, "w", encoding="utf-8") as f:
                    f.write(df_example.to_string(index=False, float_format='%.6f'))
                print(f"[PATH] Ergebnisse vollstaendig gespeichert: {os.path.abspath(case_file
                )}")
```

Listing A.9: Externe Indexierung – Zelle 7

```
#%% ================================================================
# (7) Abfrage Background Bottom-10%
#================================================================
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        header = f"\n[INFO] BG Bottom-{PERCENT:.0f}% | SCOPE={scope}, USE_TIME={use_time}"
        print(header)
        if SAVE_TO_TXT: TXT_LOG.append(header)
        times = []
        df_example = None
        for rep in range(1, REPEATS + 1):
            clear_sqlite_cache(dbp)
            with sqlite3.connect(dbp) as conn:
                t0 = _time.perf_counter()
                df, thr = bottom_percent_bg(conn, table, PERCENT, scope, use_time, T_START,
                T_END)
                dt = (_time.perf_counter() - t0) * 1000
                times.append(dt)
                line = (f"[BG][scope={scope}][use_time={use_time}][rep {rep}] rows={len(df)}
```

```
                thr={thr}   time={dt:.2f} ms")
20              print(line)
21              if SAVE_TO_TXT: TXT_LOG.append(line)
22              if rep == 1:
23                  df_example = df
24          avg_time = sum(times) / len(times)
25          avg_line = (f"[AVG][BG][scope={scope}][use_time={use_time}] Durchschnittliche Zeit
            ueber {REPEATS} Wiederholungen = {avg_time:.2f} ms")
26          print(avg_line)
27          if SAVE_TO_TXT: TXT_LOG.append(avg_line)
28          if df_example is not None and not df_example.empty:
29              print(df_example.head(10))
30              if SAVE_TO_TXT:
31                  case_name = f"bg_scope={scope}_use_time={use_time}"
32                  case_file = os.path.join(RESULTS_DIR,f"index_dev{DEVICE}_{HOSTNAME}_{datetime.
                    datetime.now().strftime('%Y%m%d_%H%M%S_%f')}_{case_name}.txt")
33                  with open(case_file, "w", encoding="utf-8") as f:
34                      f.write(df_example.to_string(index=False, float_format='%.6f'))
35                  print(f"[PATH] Ergebnisse vollstaendig gespeichert: {os.path.abspath(case_file
                    )}")
```

Listing A.10: Externe Indexierung – Zelle 7.1

```
1
2   #%% ================================================================
3   # (7.1) Speichern des Abfrageberichts (Zusammenfassung) in TXT-Datei
4   #====================================================================
5   if SAVE_TO_TXT:
6       with open(TXT_PATH, "w", encoding="utf-8") as f:
7           for line in TXT_LOG:
8               f.write(line + "\n")
9       print(f"[PATH] Zusammenfassungsbericht gespeichert in: {os.path.abspath(TXT_PATH)}")
```

A.3 Interne Indexierung

Listing A.11: Interne Indexierung – Zelle 1

```
1
2   #%% ================================================================
3   # (1) Importe + Einstellungen + Hilfsfunktionen
4   #====================================================================
5   import os, glob, re, socket, datetime
6   import time as _time
7   import h5py
8   import numpy as np
9   import pandas as pd
10  from pathlib import Path
11  import subprocess
12  pd.set_option("display.float_format", lambda x: f"{x:.6f}")
13  # ---------------------------
14  # Allgemeine Pfade
15  # ---------------------------
16  DATA_DIR    = r"C:\Atmospherdaten"
17  GLOB_PAT    = os.path.join(DATA_DIR, "*.h5")
18  RESULTS_DIR = os.path.join(DATA_DIR, "results")
19  Path(RESULTS_DIR).mkdir(parents=True, exist_ok=True)
20  # ---------------------------
21  # Allgemeine Einstellungen
22  # ---------------------------
23  ZERO_AS_NAN = False
24  HOSTNAME    = socket.gethostname()
25  RUNSTAMP    = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
26  DEVICES_TO_PROCESS = [0]   # Geraet oder mehrere
```

```
# ---------------------------
# Regex fuer Dateinamen
# ---------------------------
_name_ts_regex = re.compile(r"LISA_(\d{8})_(\d{6})\.h5$", re.IGNORECASE)
def parse_ts_from_name(path_or_name):
    """Extrahiert den Zeitstempel aus dem Dateinamen (yyyyMMdd_HHmmss)."""
    bn = os.path.basename(path_or_name)
    m = _name_ts_regex.search(bn)
    if not m:
        return None
    try:
        return datetime.datetime.strptime(m.group(1) + m.group(2), "%Y%m%d%H%M%S")
    except Exception:
        return None
def _parse_dt_loose(s):
    """Konvertiert Text zu datetime und akzeptiert mehrere Formate."""
    if s is None:
        return None
    s = s.strip()
    for fmt in ("%Y-%m-%d %H:%M:%S", "%Y-%m-%d %H:%M"):
        try:
            dt = datetime.datetime.strptime(s, fmt)
            if fmt.endswith("%H:%M"):
                dt = dt.replace(second=0, microsecond=0)
            return dt
        except Exception:
            pass
    raise ValueError(f"Nicht unterstuetztes Datum/Zeit-Format: {s!r}")
def _floor_to_minute(dt):
    return dt.replace(second=0, microsecond=0)
def _coerce_float_no_thousands(series):
    """Bereinigt numerische Spalten (z.B. '1.234,56' in float)."""
    if np.issubdtype(series.dtype, np.number):
        return series.astype("float64")
    s = series.astype(str).str.replace(" ", "")
    s = s.str.replace(".", "", regex=False).str.replace(",", ".", regex=False)
    return pd.to_numeric(s, errors="coerce").astype("float64")
def _safe_read(h5f, path):
    """Liest Dataset falls vorhanden, sonst None."""
    return h5f[path][:] if path in h5f else None
```

Listing A.12: Interne Indexierung – Zelle 2.5

```

#%% ================================================================
# (2.5) Auswahl der zu verarbeitenden HDF5-Dateien (all | range | match)
#=================================================================
SELECTION_MODE = "all"
RANGE_START = "2024-10-08 07:21:00"
RANGE_END   = "2024-10-08 07:24:00"
MATCH_GLOBS = []
_all_files = sorted(glob.glob(GLOB_PAT))
_selected = []
if SELECTION_MODE == "all":
    _selected = _all_files
elif SELECTION_MODE == "match":
    pooled = []
    for pat in MATCH_GLOBS:
        pooled += glob.glob(os.path.join(DATA_DIR, pat))
    _selected = list(dict.fromkeys(sorted(pooled)))
elif SELECTION_MODE == "range":
    pairs = []
    for fp in _all_files:
        ts = parse_ts_from_name(fp)
        if ts: pairs.append((ts, fp))
```

```
    pairs.sort(key=lambda x: x[0])
    if pairs:
        start_dt = _parse_dt_loose(RANGE_START) if RANGE_START else pairs[0][0]
        end_dt   = _parse_dt_loose(RANGE_END)   if RANGE_END   else pairs[-1][0]
        start_floor, end_floor = _floor_to_minute(start_dt), _floor_to_minute(end_dt)
        if end_floor < start_floor:
            raise ValueError("Ungueltiger Zeitbereich.")
        _selected = [fp for ts, fp in pairs if start_floor <= _floor_to_minute(ts) <=
        end_floor]
    else:
        _selected = []
else:
    print(f"[WARN] Unbekannter Modus: {SELECTION_MODE!r} -> all")
    _selected = _all_files
TARGET_FILES = _selected
print("\n[Dateiauswahl]")
print("Verzeichnis:", DATA_DIR)
print("Anzahl verfuegbare hf Dateien:", len(_all_files))
print("Anzahl ausgewaehlte hd Dateien:", len(TARGET_FILES))
for i, fp in enumerate(TARGET_FILES[:10], 1):
    print(f"  {i:>2}. {os.path.basename(fp)}")
if not TARGET_FILES:
    raise RuntimeError("Keine Dateien wurden ausgewaehlt.")
sel_list_path = os.path.join(RESULTS_DIR, f"selected_files_{RUNSTAMP}.txt")
with open(sel_list_path, "w", encoding="utf-8") as f:
    for fp in TARGET_FILES:
        f.write(fp + "\n")
print("[INFO] Dateiliste:", sel_list_path)
```

Listing A.13: Interne Indexierung – Zelle 3

```

#%% ==================================================================
# (3) Extraktion der Indizes aus einer Datei + Zeitmessung pro Index
#====================================================================
def _prep_matrix_keep_zeros(arr, n, cols=4):
    if arr is None:
        return np.full((n, cols), np.nan, dtype="float64")
    a = arr[:n].astype("float64")
    if a.ndim == 1:
        a = a.reshape(-1, 1)
    if a.shape[1] > cols:
        a = a[:, :cols]
    elif a.shape[1] < cols:
        pad = np.full((a.shape[0], cols - a.shape[1]), np.nan, dtype="float64")
        a = np.concatenate([a, pad], axis=1)
    return a
def extract_index_for_device(h5_path, device):
    d = device
    with h5py.File(h5_path, "r") as f:
        t  = _safe_read(f, f"/LISA/{d}/time")
        pk = _safe_read(f, f"/LISA/{d}/peak")
        bg = _safe_read(f, f"/LISA/{d}/background")
        mn = _safe_read(f, f"/LISA/{d}/mean")
        r  = _safe_read(f, f"/BeamStabilizer/{d}/RMSD")
    if t is None:
        raise RuntimeError(f"time Spalte fehlt in {h5_path}")
    lens = [len(t)]
    for arr in (pk, bg, mn, r):
        if arr is not None:
            lens.append(len(arr))
    n = min(lens)
    t = t[:n].astype("float64")
    r = r[:n].astype("float64") if r is not None else np.full(n, np.nan, dtype="float64")
    pk = _prep_matrix_keep_zeros(pk, n)
```

```
    mn = _prep_matrix_keep_zeros(mn, n)
    bg = _prep_matrix_keep_zeros(bg, n)
    row_max_peak = np.nanmax(pk, axis=1)
    bg_nozero = np.where(bg == 0, np.inf, bg)
    row_min_background = np.min(bg_nozero, axis=1)
    row_min_background = np.where(row_min_background == np.inf, np.nan, row_min_background)
    row_max_mean = np.nanmax(mn, axis=1)
    df = pd.DataFrame({
        "time": t,
        "filename": os.path.basename(h5_path),
        "row_idx_in_file": np.arange(n, dtype=int),
        "RMSD": r,
        "row_max_peak": row_max_peak,
        "row_min_background": row_min_background,
        "row_max_mean": row_max_mean,
    })
    for col in ["time", "RMSD", "row_max_peak", "row_min_background", "row_max_mean"]:
        df[col] = _coerce_float_no_thousands(df[col])
    return df
```

Listing A.14: Interne Indexierung – Zelle 4

```

#%% ================================================================
# (4) Erstellung einer einzigen HDF5-Kurzversion fuer alle Dateien
#=================================================================
merged_name = f"all_data_short_intern_index_bplus_{RUNSTAMP}.h5"
merged_path = os.path.join(DATA_DIR, merged_name)
if os.path.exists(merged_path):
    os.remove(merged_path)
t0 = _time.perf_counter()
with h5py.File(merged_path, "w") as f:
    meas_group = f.create_group("Measurements_short")
    datasets = {}
    for col in ["time", "row_idx_in_file", "row_max_peak",
                "row_min_background", "row_max_mean", "RMSD"]:
        datasets[col] = meas_group.create_dataset(
            col, shape=(0,), maxshape=(None,), dtype="f8"
        )
    dt_str = h5py.string_dtype(encoding="utf-8")
    datasets["filename"] = meas_group.create_dataset(
        "filename", shape=(0,), maxshape=(None,), dtype=dt_str
    )
    idx_group = f.create_group("Indexes")
    total_rows = 0
    for fp in TARGET_FILES:
        for d in DEVICES_TO_PROCESS:
            try:
                df_one = extract_index_for_device(fp, d)
                if df_one.empty:
                    print(f"[WARN] Keine Daten in {os.path.basename(fp)} fuer Geraet {d}")
                    continue
                n_new = len(df_one)
                for col in ["time", "row_idx_in_file", "row_max_peak",
                            "row_min_background", "row_max_mean", "RMSD"]:
                    old = datasets[col].shape[0]
                    datasets[col].resize((old + n_new,))
                    datasets[col][old:old+n_new] = df_one[col].to_numpy()
                old = datasets["filename"].shape[0]
                datasets["filename"].resize((old + n_new,))
                datasets["filename"][old:old+n_new] = df_one["filename"].astype(str).to_numpy
                ()
                total_rows += n_new
                print(f"[OK] Daten von {os.path.basename(fp)} hinzugefuegt (Zeilen: {n_new})")
            except Exception as e:
```

```
                print(f"[ERROR] Problem in {os.path.basename(fp)} fuer Geraet {d}: {e}")
elapsed_ms = (_time.perf_counter() - t0) * 1000
print(f"\n[INFO] Zusammengefasste Datei erstellt: {merged_path}")
print(f"[INFO] Gesamtanzahl der Zeilen: {total_rows}")
print(f"[INFO] Durchschnittliche Zeit zur Erstellung: {elapsed_ms:.2f} ms")
test_file = merged_path
```

Listing A.15: Interne Indexierung – Zelle 4.5

```
#%% ================================================================
# (4.5) Aufbau von B+ Baeumen mit globalen Indizes (global row index)
#=================================================================
LEAF_CAPACITY = 732
NODE_CAPACITY = 732
def _create_bplus_dtype():
    return np.dtype([
        ("NodeID", np.int32),
        ("IsLeaf", np.bool_),
        ("Keys", np.float64, (NODE_CAPACITY,)),
        ("Pointers", np.int32, (NODE_CAPACITY,)),
        ("NextLeaf", np.int32)
    ])
def build_bplus_tree(vals, order_keys, node_capacity=NODE_CAPACITY, leaf_capacity=
LEAF_CAPACITY):
    sort_arrays = []
    for col, asc in reversed(order_keys):
        arr = vals[col]
        if (not asc) and np.issubdtype(arr.dtype, np.number):
            arr = -arr
        sort_arrays.append(arr)
    order = np.lexsort(sort_arrays)
    n = order.size
    global_idx = order.astype(np.int64)
    ordered = {k: v[order] for k, v in vals.items()}
    dt_bplus = _create_bplus_dtype()
    nodes = []
    nleaves = int(np.ceil(n / leaf_capacity))
    for i in range(nleaves):
        start, end = i * leaf_capacity, min((i+1) * leaf_capacity, n)
        node = np.zeros((), dtype=dt_bplus)
        node["NodeID"] = i+1
        node["IsLeaf"] = True
        keys = np.zeros(node_capacity, dtype=np.float64)
        keys[:(end-start)] = ordered[order_keys[0][0]][start:end]
        node["Keys"] = keys
        ptrs = np.full(node_capacity, -1, dtype=np.int32)
        ptrs[:(end-start)] = global_idx[start:end]
        node["Pointers"] = ptrs
        node["NextLeaf"] = (i+2) if (i < nleaves-1) else -1
        nodes.append(node)
    root = np.zeros((), dtype=dt_bplus)
    root["NodeID"] = 0
    root["IsLeaf"] = False
    rkeys = np.zeros(node_capacity, dtype=np.float64)
    if len(nodes) > 1:
        seps = [nodes[i]["Keys"][0] for i in range(1, len(nodes))]
        rkeys[:len(seps)] = seps
    root["Keys"] = rkeys
    rptrs = np.full(node_capacity, -1, dtype=np.int32)
    rptrs[:len(nodes)] = [n["NodeID"] for n in nodes]
    root["Pointers"] = rptrs
    root["NextLeaf"] = -1
    return np.array([root] + nodes, dtype=dt_bplus)
with h5py.File(test_file, "r+") as f:
    vals = {
```

```
        "row_max_peak":       f["Measurements_short/row_max_peak"][:],
        "row_max_mean":       f["Measurements_short/row_max_mean"][:],
        "row_min_background": f["Measurements_short/row_min_background"][:],
        "time":               f["Measurements_short/time"][:],
        "filename":           f["Measurements_short/filename"][:],
    }
    timings = {}
    t0 = _time.perf_counter()
    bplus_peak = build_bplus_tree(vals, [("row_max_peak", False), ("time", True), ("filename", True)])
    timings["Peak"] = (_time.perf_counter() - t0) * 1000
    t0 = _time.perf_counter()
    bplus_mean = build_bplus_tree(vals, [("row_max_mean", True), ("time", True), ("filename", True)])
    timings["Mean"] = (_time.perf_counter() - t0) * 1000
    t0 = _time.perf_counter()
    bplus_bg = build_bplus_tree(vals, [("row_min_background", True), ("time", True), ("filename", True)])
    timings["Background"] = (_time.perf_counter() - t0) * 1000
    idx_group = f["Indexes"]
    for name, arr in [("bplus_peak", bplus_peak),
                      ("bplus_mean", bplus_mean),
                      ("bplus_background", bplus_bg)]:
        if name in idx_group:
            del idx_group[name]
        dset = idx_group.create_dataset(name, shape=arr.shape, maxshape=(None,), dtype=arr.dtype)
        dset[:] = arr
        print(f"[OK] neu aufgebaut {name}: nodes={len(arr)}")
for k, v in timings.items():
    print(f"[INFO] Durchschnittliche Zeit zur Erstellung eines Index auf {k}: {v:.2f} ms")
```

Listing A.16: Interne Indexierung – Zelle 4.6

```
#%% ================================================================
# (4.6) Weitere + Hilfsfunktionen fuer interne B+Baum
#=====================================================================
def _h5_take(ds, idx):
    idx = np.asarray(idx, dtype=np.int64)
    if idx.size == 0:
        return np.empty((0,), dtype=ds.dtype)
    order = np.argsort(idx, kind="mergesort")
    sorted_idx = idx[order]
    vals_sorted = ds[sorted_idx]
    inv = np.empty_like(order)
    inv[order] = np.arange(order.size)
    return vals_sorted[inv]
def _h5_take_str(ds, idx):
    arr = _h5_take(ds, idx)
    return np.array([x.decode("utf-8") if isinstance(x, (bytes, bytearray, np.bytes_)) else str(x) for x in arr],dtype=object)
def _iter_leaves(f, tree_dset_name):
    dset = f[tree_dset_name]
    cur = 1 if len(dset) > 1 else -1
    while cur != -1:
        node = dset[cur]
        ptrs = node["Pointers"]
        keys = node["Keys"]
        valid = (ptrs != -1)
        if np.any(valid):
            yield keys[valid], ptrs[valid].astype(int)
        cur = int(node["NextLeaf"])
def _per_leaf_filter(f, ptrs, scope, use_time, t0, t1):
    times  = _h5_take(   f["Measurements_short/time"],      ptrs)
    fnames = _h5_take_str(f["Measurements_short/filename"], ptrs)
```

```
    ok = np.ones(ptrs.shape[0], dtype=bool)
    if scope != "all":
        ok &= (fnames == scope)
    if use_time:
        ok &= (times >= t0) & (times <= t1)
    return ok, times, fnames
def _root_child_ids_and_seps(f, tree_dset_name):
    root = f[tree_dset_name][0]
    child_ids = root["Pointers"]
    child_ids = child_ids[child_ids != -1].astype(int)
    used_ptr_cnt = child_ids.size
    used_key_cnt = max(0, used_ptr_cnt - 1)
    seps = root["Keys"][:used_key_cnt].astype(float)
    return child_ids, seps
def _leaf_range_for_interval(seps, lower, upper, order="asc"):
    if seps.size == 0:
        return 0, 0
    if order == "asc":
        start = int(np.searchsorted(seps, lower, side="right"))
        end   = int(np.searchsorted(seps, upper, side="right"))
    else:
        s = -seps
        start = int(np.searchsorted(s, -lower, side="right"))
        end   = int(np.searchsorted(s, -upper, side="right"))
    if end < start:
        return 1, 0
    return start, end
def _iter_leaves_by_pos(f, tree_dset_name, start_pos, end_pos):
    dset = f[tree_dset_name]
    child_ids, _ = _root_child_ids_and_seps(f, tree_dset_name)
    L = child_ids.size
    if L == 0:
        return
    start_pos = max(0, min(start_pos, L-1))
    end_pos   = max(0, min(end_pos,   L-1))
    if start_pos > end_pos:
        return
    for leaf_id in child_ids[start_pos:end_pos+1]:
        node  = dset[leaf_id]
        ptrs  = node["Pointers"]
        keys  = node["Keys"]
        valid = (ptrs != -1)
        if np.any(valid):
            yield keys[valid], ptrs[valid].astype(int)
def _iter_leaves_reverse_from_end(f, tree_dset_name):
    dset = f[tree_dset_name]
    child_ids, _ = _root_child_ids_and_seps(f, tree_dset_name)
    for leaf_id in child_ids[::-1]:
        node  = dset[leaf_id]
        ptrs  = node["Pointers"]
        keys  = node["Keys"]
        valid = (ptrs != -1)
        if np.any(valid):
            yield keys[valid], ptrs[valid].astype(int)
```

Listing A.17: Interne Indexierung – Zelle 5

```

#%% ================================================================
# (5) Allgemeine Einstellungen fuer Abfragen - interne Indexierung (B+ Baum)
#=================================================================
T_START  = 1728372066.996737
T_END    = 1728372119.978274
REPEATS  = 1
PERCENT  = 10.0
```

```
SCOPES = ["all","LISA_20241008_072100.h5"]
TIME_SCENARIOS = [False, True]
SAVE_TO_TXT = True
if SAVE_TO_TXT:
    ts_now = datetime.datetime.now().strftime("%Y%m%d_%H%M%S_%f")
    TXT_PATH = os.path.join(RESULTS_DIR, f"intern_index_queries_{HOSTNAME}_{ts_now}.txt")
    TXT_LOG = []
else:
    TXT_PATH = None
    TXT_LOG = []
```

Listing A.18: Interne Indexierung – Zelle 5.2

```

#%% ================================================================
# (5.2) Hilfsfunktionen fuer Abfragen aber nur auf dem Baum
#=================================================================
def mean_range_query_bplus(test_file, scope, use_time, t0, t1):
    with h5py.File(test_file, "r") as f:
        M = np.nan
        for keys, ptrs in _iter_leaves_reverse_from_end(f, "Indexes/bplus_mean"):
            ok, _, _ = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]
            kvals = kvals[np.isfinite(kvals)]
            if kvals.size:
                M = kvals[-1]
                break
        if np.isnan(M):
            return pd.DataFrame(columns=["filename","row_idx","time","row_max_mean"]), np.nan
        lower, upper = 0.1*M, 0.8*M
        child_ids, seps = _root_child_ids_and_seps(f, "Indexes/bplus_mean")
        start_pos, end_pos = _leaf_range_for_interval(seps, lower, upper, order="asc")
        out = []
        for keys, ptrs in _iter_leaves_by_pos(f, "Indexes/bplus_mean", start_pos, end_pos):
            ok, times, fnames = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]; psel = ptrs[ok]; tsel = times[ok]; fsel = fnames[ok]
            sel = (kvals >= lower) & (kvals <= upper) & np.isfinite(kvals)
            if np.any(sel):
                out.extend(zip(fsel[sel].tolist(),psel[sel].astype(int),tsel[sel],kvals[sel]))
        df = pd.DataFrame(out, columns=["filename","row_idx","time","row_max_mean"])
        if not df.empty:
            df = df.sort_values("row_max_mean", ascending=True, kind="mergesort")
        return df, M
def top_percent_peak_bplus(test_file, percent, scope, use_time, t0, t1):
    q = 1 - percent/100.0
    with h5py.File(test_file, "r") as f:
        N = 0
        for keys, ptrs in _iter_leaves(f, "Indexes/bplus_peak"):
            ok, _, _ = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]
            N += np.sum(np.isfinite(kvals))
        if N == 0:
            return pd.DataFrame(columns=["filename","row_idx","time","row_max_peak"]), np.nan
        pos  = (N - 1) * (1 - q)
        i0   = int(np.floor(pos))
        i1   = int(np.ceil(pos))
        frac = pos - i0
        seen = 0; v0 = None; v1 = None
        for keys, ptrs in _iter_leaves(f, "Indexes/bplus_peak"):
            ok, _, _ = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]; kvals = kvals[np.isfinite(kvals)]
            m = kvals.size
            if m == 0:
                continue
            if v0 is None and i0 < seen + m: v0 = kvals[i0 - seen]
```

```
            if v1 is None and i1 < seen + m: v1 = kvals[i1 - seen]
            seen += m
            if v0 is not None and v1 is not None:
                break
        thr = v0 if i0 == i1 else (v0 + (v1 - v0) * frac)
        child_ids, seps = _root_child_ids_and_seps(f, "Indexes/bplus_peak")
        start_pos, end_pos = _leaf_range_for_interval(seps, lower=thr, upper=thr, order="desc"
        )
        out = []
        for keys, ptrs in _iter_leaves_by_pos(f, "Indexes/bplus_peak", 0, end_pos):
            ok, times, fnames = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]; psel = ptrs[ok]; tsel = times[ok]; fsel = fnames[ok]
            sel = np.isfinite(kvals) & (kvals >= thr)
            if np.any(sel):
                out.extend(zip(fsel[sel].tolist(),psel[sel].astype(int),tsel[sel],kvals[sel]))
        df = pd.DataFrame(out, columns=["filename","row_idx","time","row_max_peak"])
        if not df.empty:
            df = df.sort_values("row_max_peak", ascending=False, kind="mergesort")
        return df, thr
def bottom_percent_bg_bplus(test_file, percent, scope, use_time, t0, t1):
    q = percent/100.0
    with h5py.File(test_file, "r") as f:
        N = 0
        for keys, ptrs in _iter_leaves(f, "Indexes/bplus_background"):
            ok, _, _ = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]; kvals = kvals[np.isfinite(kvals) & (kvals != 0)]
            N += kvals.size
        if N == 0:
            return pd.DataFrame(columns=["filename","row_idx","time","row_min_background"]),
            np.nan
        pos  = (N - 1) * q
        i0   = int(np.floor(pos))
        i1   = int(np.ceil(pos))
        frac = pos - i0
        seen = 0; v0 = None; v1 = None
        for keys, ptrs in _iter_leaves(f, "Indexes/bplus_background"):
            ok, _, _ = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]; kvals = kvals[np.isfinite(kvals) & (kvals != 0)]
            m = kvals.size
            if m == 0:
                continue
            if v0 is None and i0 < seen + m: v0 = kvals[i0 - seen]
            if v1 is None and i1 < seen + m: v1 = kvals[i1 - seen]
            seen += m
            if v0 is not None and v1 is not None:
                break
        thr = v0 if i0 == i1 else (v0 + (v1 - v0) * frac)
        child_ids, seps = _root_child_ids_and_seps(f, "Indexes/bplus_background")
        start_pos, end_pos = _leaf_range_for_interval(seps, lower=thr, upper=thr, order="asc")
        out = []
        for keys, ptrs in _iter_leaves_by_pos(f, "Indexes/bplus_background", 0, end_pos):
            ok, times, fnames = _per_leaf_filter(f, ptrs, scope, use_time, t0, t1)
            kvals = keys[ok]; psel = ptrs[ok]; tsel = times[ok]; fsel = fnames[ok]
            sel = np.isfinite(kvals) & (kvals != 0) & (kvals <= thr)
            if np.any(sel):
                out.extend(zip(fsel[sel].tolist(),psel[sel].astype(int),tsel[sel],kvals[sel]))
        df = pd.DataFrame(out, columns=["filename","row_idx","time","row_min_background"])
        if not df.empty:
            df = df.sort_values("row_min_background", ascending=True, kind="mergesort")
        return df, thr
```

A.3.1 Interne Indexierungsabfragen

Listing A.19: Interne Indexierung – (vereinte Zellen 5.5, 6, 7 und 7.1)

```

#%% ============================================================
# (final) Zellen der Abfragen (vereint aus Zelle 5.5, 6 und 7) #mit gemeinsamer Ausgabe
#================================================================
SAVE_TO_TXT = True
if SAVE_TO_TXT:
    ts_now = datetime.datetime.now().strftime("%Y%m%d_%H%M%S_%f")
    TXT_PATH = os.path.join(RESULTS_DIR,f"intern_index_queries_{HOSTNAME}_{ts_now}.txt")
    TXT_LOG = []
else:
    TXT_PATH = None
    TXT_LOG = []
def _save_case(df, case_name):
    if not SAVE_TO_TXT or df.empty:
        return
    fname = f"intern_index_{HOSTNAME}_{datetime.datetime.now().strftime('%Y%m%d_%H%M%S_%f')}_{case_name}.txt"
    fpath = os.path.join(RESULTS_DIR, fname)
    with open(fpath, "w", encoding="utf-8") as f:
        f.write(df.to_string(index=False, float_format="%.6f"))
    print(f"[PATH] saved full results: {os.path.abspath(fpath)}")
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        times = []; df_example = None; M_last = None
        for rep in range(1, REPEATS+1):
            t0 = _time.perf_counter()
            df, M = mean_range_query_bplus(test_file, scope, use_time, T_START, T_END)
            dt = (_time.perf_counter() - t0) * 1000
            times.append(dt); M_last = M
            line = f"[Mean][scope={scope}][use_time={use_time}][rep {rep}] rows={len(df)} M={M} time={dt:.2f} ms"
            print(line); TXT_LOG.append(line)
            if rep == 1: df_example = df
        avg = sum(times)/len(times)
        avg_line = f"[AVG][Mean][scope={scope}][use_time={use_time}] avg={avg:.2f} ms"
        print(avg_line); TXT_LOG.append(avg_line)
        if df_example is not None and not df_example.empty:
            print(df_example.head(10))
            _save_case(df_example, f"mean_scope={scope}_use_time={use_time}")
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        times = []; df_example = None; thr_last = None
        for rep in range(1, REPEATS+1):
            t0 = _time.perf_counter()
            df, thr = top_percent_peak_bplus(test_file, PERCENT, scope, use_time, T_START, T_END)
            dt = (_time.perf_counter() - t0) * 1000
            times.append(dt); thr_last = thr
            line = f"[Peak Top-{PERCENT:.0f}%][scope={scope}][use_time={use_time}][rep {rep}] rows={len(df)} thr={thr} time={dt:.2f} ms"
            print(line); TXT_LOG.append(line)
            if rep == 1: df_example = df
        avg = sum(times)/len(times)
        avg_line = f"[AVG][Peak Top-{PERCENT:.0f}%][scope={scope}][use_time={use_time}] avg={avg:.2f} ms"
        print(avg_line); TXT_LOG.append(avg_line)
        if df_example is not None and not df_example.empty:
            print(df_example.head(10))
            _save_case(df_example, f"peak_scope={scope}_use_time={use_time}")
for scope in SCOPES:
```

```
    for use_time in TIME_SCENARIOS:
        times = []; df_example = None; thr_last = None
        for rep in range(1, REPEATS+1):
            t0 = _time.perf_counter()
            df, thr = bottom_percent_bg_bplus(test_file, PERCENT, scope, use_time, T_START,
            T_END)
            dt = (_time.perf_counter() - t0) * 1000
            times.append(dt); thr_last = thr
            line = f"[BG Bottom-{PERCENT:.0f}%][scope={scope}][use_time={use_time}][rep {rep}]
             rows={len(df)} thr={thr} time={dt:.2f} ms"
            print(line); TXT_LOG.append(line)
            if rep == 1: df_example = df
        avg = sum(times)/len(times)
        avg_line = f"[AVG][BG Bottom-{PERCENT:.0f}%][scope={scope}][use_time={use_time}] avg={
        avg:.2f} ms"
        print(avg_line); TXT_LOG.append(avg_line)
        if df_example is not None and not df_example.empty:
            print(df_example.head(10))
            _save_case(df_example, f"bg_scope={scope}_use_time={use_time}")
if SAVE_TO_TXT:
    with open(TXT_PATH, "w", encoding="utf-8") as f:
        for line in TXT_LOG:
            f.write(line + "\n")
    print(f"[PATH] saved summary report: {os.path.abspath(TXT_PATH)}")
```

A.4 Variante ohne Indexierung

Listing A.20: Ohne Indexierung – Zelle 1

```

#%% ======================================================================
# (1) Importe + Einstellungen + Hilfsfunktionen
#======================================================================
import h5py, os, glob, re, time as _time, datetime, socket
import numpy as np
import pandas as pd
from pathlib import Path
import subprocess
# Daten- und Ergebnisordner
DATA_DIR    = r"C:\Atmospherdaten"
GLOB_PAT    = os.path.join(DATA_DIR, "*.h5")
RESULTS_DIR = os.path.join(DATA_DIR, "results")
Path(RESULTS_DIR).mkdir(parents=True, exist_ok=True)
HOSTNAME    = socket.gethostname()
RUNSTAMP    = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
# Geraet auswaehlen (0 oder 1)
DEVICE   = 0
# HDF5 Dateinamen extrahieren
_name_ts_regex = re.compile(r"LISA_(\d{8})_(\d{6})\.h5$", re.IGNORECASE)
def parse_ts_from_name(path_or_name):
    bn = os.path.basename(path_or_name)
    m = _name_ts_regex.search(bn)
    if not m:
        return None
    try:
        return datetime.datetime.strptime(m.group(1)+m.group(2), "%Y%m%d%H%M%S")
    except:
        return None
def _parse_dt_loose(s):
    if s is None:
        return None
    s = s.strip()
```

```
    for fmt in ("%Y-%m-%d %H:%M:%S", "%Y-%m-%d %H:%M"):
        try:
            dt = datetime.datetime.strptime(s, fmt)
            if fmt.endswith("%H:%M"):
                dt = dt.replace(second=0, microsecond=0)
            return dt
        except:
            pass
    raise ValueError(f"Nicht unterstuetztes Datum/Zeit-Format: {s!r}")
def _floor_to_minute(dt):
    return dt.replace(second=0, microsecond=0)
```

Listing A.21: Ohne Indexierung – Zelle 2.5

```

#%% ================================================================
# (2.5) Auswahl der zu verarbeitenden HDF5-Dateien
#=================================================================
SELECTION_MODE = "all"     # "all" | "range" | "match"
RANGE_START = "2024-10-08 07:21:00"
RANGE_END   = "2024-10-08 07:24:00"
MATCH_GLOBS = []
_all_files = sorted(glob.glob(GLOB_PAT))
_selected = []
if SELECTION_MODE == "all":
    _selected = _all_files
elif SELECTION_MODE == "match":
    pooled = []
    for pat in MATCH_GLOBS:
        pooled += glob.glob(os.path.join(DATA_DIR, pat))
    _selected = list(dict.fromkeys(sorted(pooled)))
elif SELECTION_MODE == "range":
    pairs = []
    for fp in _all_files:
        ts = parse_ts_from_name(fp)
        if ts: pairs.append((ts, fp))
    pairs.sort(key=lambda x: x[0])
    if pairs:
        start_dt = _parse_dt_loose(RANGE_START) if RANGE_START else pairs[0][0]
        end_dt   = _parse_dt_loose(RANGE_END)   if RANGE_END   else pairs[-1][0]
        start_floor, end_floor = _floor_to_minute(start_dt), _floor_to_minute(end_dt)
        if end_floor < start_floor:
            raise ValueError("Ungueltiger Zeitbereich.")
        _selected = [fp for ts, fp in pairs if start_floor <= _floor_to_minute(ts) <=
        end_floor]
    else:
        _selected = []
else:
    print(f"[WARN] Unbekannter Modus: {SELECTION_MODE!r} -> all")
    _selected = _all_files
TARGET_FILES = _selected
print("\n[Dateiauswahl]")
print("Verzeichnis:", DATA_DIR)
print("Anzahl verfuegbare h5 Dateien:", len(_all_files))
print("Anzahl ausgewaehlte h5 Dateien:", len(TARGET_FILES))
for i, fp in enumerate(TARGET_FILES[:10], 1):
    print(f"  {i:>2}. {os.path.basename(fp)}")
if not TARGET_FILES:
    raise RuntimeError("Keine Dateien wurden ausgewaehlt.")
sel_list_path = os.path.join(RESULTS_DIR, f"selected_files_ohne_index_{RUNSTAMP}.txt")
with open(sel_list_path, "w", encoding="utf-8") as f:
    for fp in TARGET_FILES:
        f.write(fp + "\n")
print("[INFO] Dateiliste:", sel_list_path)
```

Listing A.22: Ohne Indexierung – Zelle 5

```

#%% ======================================================================
# (5) Allgemeine Einstellungen fuer Abfragen
#======================================================================
T_START  = 1728372066.996737
T_END    = 1728372119.978274
REPEATS  = 1
PERCENT  = 10.0   # Top-10%
SCOPES = [
    "all",
    "LISA_20241008_072100.h5"
]
TIME_SCENARIOS = [False, True]
SAVE_TO_TXT = True   # wenn False -> kein Report speichern
if SAVE_TO_TXT:
    ts_now = datetime.datetime.now().strftime("%Y%m%d_%H%M%S_%f")
    TXT_PATH = os.path.join(
        RESULTS_DIR,
        f"ohne_index_{HOSTNAME}_{ts_now}_queries.txt"
    )
    TXT_LOG = []
else:
    TXT_PATH = None
    TXT_LOG = []
```

Listing A.23: Ohne Indexierung – Zelle 5.2

```

#%% ======================================================================
# (5.2) Hilfsfunktionen fuer OHNE INDEX Variante
#======================================================================

def _scan_mean_df(files, device):
    out = []
    for fp in files:
        with h5py.File(fp, "r") as f:
            t  = f[f"/LISA/{device}/time"][:]
            mn = f[f"/LISA/{device}/mean"][:]
        a = mn.astype("float64")
        if a.ndim == 1:
            a = a.reshape(-1, 1)
        if a.shape[1] > 4:
            a = a[:, :4]
        elif a.shape[1] < 4:
            a = np.concatenate([a, np.full((a.shape[0], 4-a.shape[1]), np.nan)], axis=1)
        row_max_mean = np.nanmax(a, axis=1)
        n = len(row_max_mean)
        out.append(pd.DataFrame({
            "filename": os.path.basename(fp),
            "row_idx":  np.arange(n, dtype=int),
            "time":     t.astype("float64"),
            "row_max_mean": row_max_mean
        }))
    df = pd.concat(out, ignore_index=True) if out else pd.DataFrame(
        columns=["filename","row_idx","time","row_max_mean"])
    return df.sort_values(
        by=["row_max_mean", "time", "filename", "row_idx"],
        ascending=[True, True, True, True]
    ).reset_index(drop=True)


def _scan_peak_df(files, device):
    out = []
```

```
    for fp in files:
        with h5py.File(fp, "r") as f:
            t  = f[f"/LISA/{device}/time"][:]
            pk = f[f"/LISA/{device}/peak"][:]
        a = pk.astype("float64")
        if a.ndim == 1:
            a = a.reshape(-1, 1)
        if a.shape[1] > 4:
            a = a[:, :4]
        elif a.shape[1] < 4:
            a = np.concatenate([a, np.full((a.shape[0], 4-a.shape[1]), np.nan)], axis=1)
        row_max_peak = np.nanmax(a, axis=1)
        n = len(row_max_peak)
        out.append(pd.DataFrame({
            "filename": os.path.basename(fp),
            "row_idx":  np.arange(n, dtype=int),
            "time":     t.astype("float64"),
            "row_max_peak": row_max_peak
        }))
    df = pd.concat(out, ignore_index=True) if out else pd.DataFrame(
        columns=["filename","row_idx","time","row_max_peak"])
    return df.sort_values(
        by=["row_max_peak", "time", "filename", "row_idx"],
        ascending=[False, True, True, True]
    ).reset_index(drop=True)

def _scan_bg_df(files, device):
    out = []
    for fp in files:
        with h5py.File(fp, "r") as f:
            t  = f[f"/LISA/{device}/time"][:]
            bg = f[f"/LISA/{device}/background"][:]
        a = bg.astype("float64")
        if a.ndim == 1:
            a = a.reshape(-1, 1)
        if a.shape[1] > 4:
            a = a[:, :4]
        elif a.shape[1] < 4:
            a = np.concatenate([a, np.full((a.shape[0], 4-a.shape[1]), np.nan)], axis=1)
        a_nozero = np.where(a == 0, np.inf, a)
        row_min_background = np.min(a_nozero, axis=1)
        row_min_background = np.where(np.isinf(row_min_background), np.nan, row_min_background
        )
        n = len(row_min_background)
        out.append(pd.DataFrame({
            "filename": os.path.basename(fp),
            "row_idx":  np.arange(n, dtype=int),
            "time":     t.astype("float64"),
            "row_min_background": row_min_background
        }))
    df = pd.concat(out, ignore_index=True) if out else pd.DataFrame(
        columns=["filename","row_idx","time","row_min_background"])
    return df.sort_values(
        by=["row_min_background", "time", "filename", "row_idx"],
        ascending=[True, True, True, True]
    ).reset_index(drop=True)


def _apply_scope_time(df, scope, use_time, t0, t1):
    if scope != "all":
        df = df[df["filename"] == scope]
    if use_time:
        df = df[(df["time"] >= t0) & (df["time"] <= t1)]
    return df
```

A.4.1 Abfragen der Variante ohne indexierung

Listing A.24: Ohne Indexierung – Zelle 5.5

```
1
2  # ----------------------------
3  #  Mean Range: 0.1*M .. 0.8*M     Zelle 5.5
4  # ----------------------------
5  for scope in SCOPES:
6      for use_time in TIME_SCENARIOS:
7          header = f"\n[OHNE][MEAN] SCOPE={scope} USE_TIME={use_time}"
8          print(header)
9          if SAVE_TO_TXT: TXT_LOG.append(header)
10         times = []; df_example = None
11         for rep in range(1, REPEATS+1):
12             t0 = _time.perf_counter()
13             df_all = _scan_mean_df(TARGET_FILES, DEVICE)
14             df_scope = _apply_scope_time(df_all, scope, use_time, T_START, T_END)
15             if df_scope.empty or df_scope["row_max_mean"].isna().all():
16                 M = np.nan; df_res = pd.DataFrame()
17             else:
18                 M = df_scope["row_max_mean"].max()
19                 df_res = df_scope[(df_scope["row_max_mean"] > 0.1*M) & (df_scope["row_max_mean
                   "] < 0.8*M)] \
20
21             dt = (_time.perf_counter()-t0)*1000; times.append(dt)
22             line = f"[rep {rep}] rows={len(df_res)}  M={M}  time={dt:.2f} ms"
23             print(line)
24             if SAVE_TO_TXT: TXT_LOG.append(line)
25             if rep==1: df_example = df_res
26         avg = sum(times)/len(times)
27         avg_line = f"[AVG][OHNE][MEAN][scope={scope}][use_time={use_time}] avg={avg:.2f} ms"
28         print(avg_line)
29         if SAVE_TO_TXT: TXT_LOG.append(avg_line)
30         if df_example is not None and not df_example.empty:
31             print(df_example.head(10))
32             if SAVE_TO_TXT:
33                 case_name = f"mean_scope={scope}_use_time={use_time}"
34                 case_file = os.path.join(
35                     RESULTS_DIR,
36                     f"ohne_index_{HOSTNAME}_{ts_now}_{case_name}.txt"
37                 )
38                 with open(case_file, "w", encoding="utf-8") as f:
39                     f.write(df_example.to_string(index=False, float_format="%.6f"))
40                 print(f"[PATH] volle Ergebnisse gespeichert: {os.path.abspath(case_file)}")
```

Listing A.25: Ohne Indexierung – Zelle 6

```
1
2  # ----------------------------
3  #  Peak Top-N%    Zelle 6
4  # ----------------------------
5  for scope in SCOPES:
6      for use_time in TIME_SCENARIOS:
7          header = f"\n[OHNE][PEAK Top-{PERCENT:.0f}%] SCOPE={scope} USE_TIME={use_time}"
8          print(header)
9          if SAVE_TO_TXT: TXT_LOG.append(header)
10         times = []; df_example = None
11         for rep in range(1, REPEATS+1):
12             t0 = _time.perf_counter()
13             df_all = _scan_peak_df(TARGET_FILES, DEVICE)
14             df_scope = _apply_scope_time(df_all, scope, use_time, T_START, T_END)
15             if df_scope.empty or df_scope["row_max_peak"].isna().all():
16                 thr = np.nan; df_res = pd.DataFrame()
17             else:
```

```
                thr = df_scope["row_max_peak"].quantile(1 - PERCENT/100.0)
                df_res = df_scope[df_scope["row_max_peak"] >= thr]
            dt = (_time.perf_counter()-t0)*1000; times.append(dt)
            line = f"[rep {rep}] rows={len(df_res)}  thr={thr}  time={dt:.2f} ms"
            print(line)
            if SAVE_TO_TXT: TXT_LOG.append(line)
            if rep==1: df_example = df_res
        avg = sum(times)/len(times)
        avg_line = f"[AVG][OHNE][PEAK][scope={scope}][use_time={use_time}] avg={avg:.2f} ms"
        print(avg_line)
        if SAVE_TO_TXT: TXT_LOG.append(avg_line)
        if df_example is not None and not df_example.empty:
            print(df_example.head(10))
            if SAVE_TO_TXT:
                case_name = f"peak_scope={scope}_use_time={use_time}"
                case_file = os.path.join(
                    RESULTS_DIR,
                    f"ohne_index_{HOSTNAME}_{ts_now}_{case_name}.txt"
                )
                with open(case_file, "w", encoding="utf-8") as f:
                    f.write(df_example.to_string(index=False, float_format="%.6f"))
                print(f"[PATH] volle Ergebnisse gespeichert: {os.path.abspath(case_file)}")
```

Listing A.26: Ohne Indexierung – Zelle 7

```
#  Background Bottom-10%  Zelle 7
# ------------------------------
for scope in SCOPES:
    for use_time in TIME_SCENARIOS:
        header = f"\n[OHNE][BG Bottom-10%] SCOPE={scope} USE_TIME={use_time}"
        print(header)
        if SAVE_TO_TXT: TXT_LOG.append(header)
        times = []; df_example = None
        for rep in range(1, REPEATS+1):
            t0 = _time.perf_counter()
            df_all = _scan_bg_df(TARGET_FILES, DEVICE)
            df_scope = _apply_scope_time(df_all, scope, use_time, T_START, T_END)
            df_nz = df_scope.dropna(subset=["row_min_background"])
            df_nz = df_nz[df_nz["row_min_background"] != 0]
            if df_nz.empty:
                thr = np.nan; df_res = pd.DataFrame()
            else:
                thr = df_nz["row_min_background"].quantile(0.10)
                df_res = df_nz[df_nz["row_min_background"] <= thr]
            dt = (_time.perf_counter()-t0)*1000; times.append(dt)
            line = f"[rep {rep}] rows={len(df_res)}  thr={thr}  time={dt:.2f} ms"
            print(line)
            if SAVE_TO_TXT: TXT_LOG.append(line)
            if rep==1: df_example = df_res
        avg = sum(times)/len(times)
        avg_line = f"[AVG][OHNE][BG][scope={scope}][use_time={use_time}] avg={avg:.2f} ms"
        print(avg_line)
        if SAVE_TO_TXT: TXT_LOG.append(avg_line)
        if df_example is not None and not df_example.empty:
            print(df_example.head(10))
            if SAVE_TO_TXT:
                case_name = f"bg_scope={scope}_use_time={use_time}"
                case_file = os.path.join(
                    RESULTS_DIR,
                    f"ohne_index_{HOSTNAME}_{ts_now}_{case_name}.txt"
                )
                with open(case_file, "w", encoding="utf-8") as f:
                    f.write(df_example.to_string(index=False, float_format="%.6f"))
                print(f"[PATH] volle Ergebnisse gespeichert: {os.path.abspath(case_file)}")
```

Listing A.27: Ohne Indexierung – Zelle 7.1

```
#%% ================================================================
# (7.1) Speichern des Zusammenfassungs-Reports in TXT
#=================================================================
if SAVE_TO_TXT:
    with open(TXT_PATH, "w", encoding="utf-8") as f:
        for line in TXT_LOG:
            f.write(line + "\n")
    print
```

Anhang B

Vollständiger Messdurchläufe

In diesem Anhang werden die Ergebnisse der wiederholten Messungen für die Abfragen sowie die Zeiten zur Erstellung der Indizes im Detail dargestellt. In diesem Anhang steht σ für die Standardabweichung der jeweiligen fünf Durchläufe.

B.1 Vergleich der Laufzeit der Rohdatenextraktion bei externer und interner Indexierung

Schritt	**rep 1**	**rep 2**	**rep 3**	**rep 4**	**rep 5**	**AVG**	σ
Extraktion	2549.56	2491.48	2512.59	2376.80	2659.12	**2517.91**	101.96

Tabelle B.1: Laufzeiten für die Extraktion der Rohdaten aus der HDF5-Datei (fünf Wiederholungen, Durchschnitt und Standardabweichung).

B.2 Vergleich der Erstellungszeiten: zusammengefasste HDF5-Datei vs externe SQLite-Datenbank

Variante	**Ablage der reduzierten Daten**	**rep 1**	**rep 2**	**rep 3**	**rep 4**	**rep 5**	**AVG**	σ
Externe Indexierung	Erstellung der reduzierten Tabelle in SQLite	3269.57	2948.52	3312.09	3249.60	3198.36	**3195.63**	144.06
Interne Indexierung	Erstellung einer zusammengefassten HDF5-Datei	1290.26	1281.33	1292.31	1298.77	1263.17	**1285.17**	13.79

Tabelle B.2: Laufzeiten für die Ablage der reduzierten Datenbasis in beiden Varianten (fünf Wiederholungen, Durchschnitt und Standardabweichung).

B.3 Vergleich der Erstellungszeiten der Indizes in SQLite und HDF5

Index	Variante	rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
row_max_Peak	SQLite (extern)	1253.94	1178.00	1430.71	1401.31	1408.64	**1334.52**	110.43
	HDF5 (intern)	141.53	150.97	136.36	133.26	132.12	**138.85**	7.87
row_min_Background	SQLite (extern)	1226.13	1170.77	1212.68	1177.02	1164.64	**1190.25**	24.60
	HDF5 (intern)	122.28	123.65	117.10	118.66	158.89	**128.12**	15.48
row_max_Mean	SQLite (extern)	1225.86	1302.67	1237.02	1252.84	1218.34	**1247.35**	31.77
	HDF5 (intern)	192.84	178.34	146.84	160.26	153.78	**166.41**	16.79

Tabelle B.3: Vergleich der Laufzeiten für die Indexerstellung in SQLite (externe Indexierung) und HDF5 (interne Indexierung) über fünf Wiederholungen, Durchschnitt und Standardabweichung.

B.4 Peak Top-10% – Vollständige Messdurchläufe

Externe Indexierung – Peak Top-10%		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	use_time							
all	false	193.74	187.36	197.53	179.71	202.72	**192.21**	8.64
all	true	339.09	285.78	282.32	290.68	282.41	**296.06**	22.80
filename	false	17.69	17.62	17.66	20.45	18.94	**18.47**	1.08
filename	true	21.40	20.15	17.64	22.80	22.99	**20.99**	2.01

Tabelle B.4: Laufzeiten für die Top-10%-Abfrage (row_max_peak) in der externen Indexierung mit Mittelwert und Standardabweichung.

Interne Ind. - Peak Top-10%		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	use_time							
all	false	15849.79	15797.10	16065.28	15706.33	15736.06	**15830.91**	132.02
all	true	37153.17	37411.02	37561.22	37333.72	37452.14	**37382.25**	148.74
filename	false	37755.48	37666.34	37815.63	37860.07	37403.26	**37700.16**	168.12
filename	true	36677.66	36773.39	36258.70	36648.71	36664.36	**36604.56**	196.79

Tabelle B.5: Laufzeiten für die Top-10%-Abfrage (row_max_peak) in der internen Indexierung mit Mittelwert und Standardabweichung.

Ohne Indexierung – Peak Top-10%		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	use_time							
all	false	1506.48	1505.40	1518.43	1498.69	1505.73	**1506.95**	6.57
all	true	1500.17	1490.55	1499.82	1491.67	1487.42	**1493.93**	5.17
filename	false	1523.45	1517.20	1519.08	1518.54	1515.51	**1518.76**	2.88
filename	true	1520.36	1517.56	1529.55	1522.54	1513.52	**1520.71**	5.62

Tabelle B.6: Laufzeiten für die Top-10%-Abfrage (row_max_peak) ohne Indexierung mit Mittelwert und Standardabweichung.

B.5 Mean 0.1M..0.8M – Vollständige Messdurchläufe

Externe Ind. – Mean 0.1M..0.8M		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	**use_time**							
all	false	1111.75	1250.28	1264.23	1349.02	1231.71	**1241.40**	84.14
all	true	130.45	104.06	104.18	100.53	106.59	**109.16**	11.36
filename	false	17.14	16.81	15.36	15.35	15.47	**16.02**	0.79
filename	true	18.76	21.10	15.98	20.34	17.04	**18.64**	1.89

Tabelle B.7: Laufzeiten für die Bereichsanfrage (Mean 0.1M..0.8M) in der externen Indexierung mit Mittelwert und Standardabweichung.

Interne Ind. - Mean 0.1M..0.8M		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	**use_time**							
all	false	21197.60	20710.69	21238.10	20739.33	21242.36	**21025.62**	259.21
all	true	20773.12	20709.77	20239.04	20131.95	20341.57	**20439.09**	292.73
filename	false	20211.27	20090.47	20360.66	19820.19	20151.25	**20126.77**	192.10
filename	true	20303.70	20082.01	20427.12	19982.11	20096.81	**20178.35**	175.32

Tabelle B.8: Laufzeiten für die Bereichsanfrage (Mean 0.1M..0.8M) in der internen Indexierung mit Mittelwert und Standardabweichung.

Ohne Indexierung – Mean 0.1M..0.8M		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	**use_time**							
all	false	1620.74	1704.03	1638.20	1609.46	1639.21	**1642.33**	34.72
all	true	1493.64	1504.93	1498.56	1498.57	1503.72	**1499.89**	4.46
filename	false	1536.78	1533.60	1531.54	1523.86	1511.32	**1527.42**	9.40
filename	true	1519.61	1536.14	1510.42	1531.44	1586.91	**1536.91**	27.68

Tabelle B.9: Laufzeiten für die Bereichsanfrage (Mean 0.1M..0.8M) ohne Indexierung mit Mittelwert und Standardabweichung.

B.6 BG Bottom-10% – Vollständige Messdurchläufe

Externe Indexierung – BG Bottom-10%		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	**use_time**							
all	false	576.83	588.43	626.29	629.58	682.29	**620.68**	40.63
all	true	129.71	136.35	119.23	119.41	116.74	**124.29**	7.94
filename	false	7.41	6.66	7.19	10.62	6.95	**7.77**	1.47
filename	true	8.51	6.97	7.83	7.78	8.40	**7.90**	0.58

Tabelle B.10: Laufzeiten für die Bottom-10%-Abfrage (`row_min_background`) in der externen Indexierung mit Mittelwert und Standardabweichung.

Interne Ind. – BG Bottom-10%		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	**use_time**							
all	false	9953.60	10167.62	9931.32	9781.18	10048.24	**9976.39**	142.65
all	true	8890.69	9037.09	8662.27	8736.07	8885.78	**8842.38**	134.12
filename	false	9094.94	8587.81	8908.44	8388.19	8497.12	**8695.30**	291.20
filename	true	8868.39	8882.35	8732.30	8731.51	8727.66	**8788.44**	73.50

Tabelle B.11: Laufzeiten für die Bottom-10%-Abfrage (`row_min_background`) in der internen Indexierung mit Mittelwert und Standardabweichung.

Ohne Indexierung – BG Bottom-10%		rep 1	rep 2	rep 3	rep 4	rep 5	AVG	σ
Scope	**use_time**							
all	false	1548.83	1550.63	1567.11	1566.17	1567.34	**1560.01**	9.08
all	true	1489.92	1490.13	1618.50	1513.53	1491.10	**1520.63**	55.96
filename	false	1529.65	1537.27	1513.24	1508.43	1522.71	**1522.26**	10.62
filename	true	1503.32	1516.86	1511.64	1517.58	1511.42	**1512.16**	5.32

Tabelle B.12: Laufzeiten für die Bottom-10%-Abfrage (`row_min_background`) ohne Indexierung mit Mittelwert und Standardabweichung.

B.7 Verwendete SQL-Abfragen

Die in den vorigen Kapiteln ausgewerteten Messungen basieren auf den folgenden SQL-Abfragen. Um identische Ergebnisse und Laufzeiten zu reproduzieren, muss daher das gesamte Python-Skript ausgeführt werden. Eine alleinige Ausführung der unten stehenden Abfragen kann zu abweichenden Messwerten führen.

1) Mean Range (0.1M .. 0.8M)

Listing B.1: Mean | all | use-time = false

```sql
SELECT filename, row_idx, time, row_max_mean
FROM entries_dev0
WHERE row_max_mean BETWEEN (
        0.1 * (SELECT MAX(row_max_mean) FROM entries_dev0)
      )
      AND (
        0.8 * (SELECT MAX(row_max_mean) FROM entries_dev0)
      )
ORDER BY row_max_mean ASC;
```

Listing B.2: Mean | all | use-time = true

```sql
SELECT filename, row_idx, time, row_max_mean
FROM entries_dev0
WHERE row_max_mean BETWEEN (
        0.1 * (SELECT MAX(row_max_mean) FROM entries_dev0)
      )
      AND (
        0.8 * (SELECT MAX(row_max_mean) FROM entries_dev0)
      )
  AND time BETWEEN 1728372066.996737 AND 1728372119.978274
ORDER BY row_max_mean ASC;
```

Listing B.3: Mean | filename | use-time = false

```sql
SELECT filename, row_idx, time, row_max_mean
FROM entries_dev0
WHERE filename = 'LISA_20241008_072100.h5'
  AND row_max_mean BETWEEN (
        0.1 * (SELECT MAX(row_max_mean)
               FROM entries_dev0
               WHERE filename='LISA_20241008_072100.h5')
      )
      AND (
        0.8 * (SELECT MAX(row_max_mean)
               FROM entries_dev0
               WHERE filename='LISA_20241008_072100.h5')
      )
ORDER BY row_max_mean ASC;
```

Listing B.4: Mean | filename | use-time = true

```sql
SELECT filename, row_idx, time, row_max_mean
FROM entries_dev0
WHERE filename = 'LISA_20241008_072100.h5'
  AND row_max_mean BETWEEN (
        0.1 * (SELECT MAX(row_max_mean)
               FROM entries_dev0
               WHERE filename='LISA_20241008_072100.h5')
      )
      AND (
        0.8 * (SELECT MAX(row_max_mean)
               FROM entries_dev0
               WHERE filename='LISA_20241008_072100.h5')
      )
  AND time BETWEEN 1728372066.996737 AND 1728372119.978274
ORDER BY row_max_mean ASC;
```

2) Peak Top-10%

Listing B.5: Peak | all | use-time = false

```sql
SELECT filename, row_idx, time, row_max_peak
FROM entries_dev0
WHERE row_max_peak >= (
        WITH ranked AS (
            SELECT row_max_peak,
                   ROW_NUMBER() OVER (ORDER BY row_max_peak DESC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
        )
        SELECT row_max_peak
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
      )
ORDER BY row_max_peak DESC;
```

Listing B.6: Peak | all | use-time = true

```sql
SELECT filename, row_idx, time, row_max_peak
FROM entries_dev0
WHERE time BETWEEN 1728372066.996737 AND 1728372119.978274
  AND row_max_peak >= (
        WITH ranked AS (
            SELECT row_max_peak,
                   ROW_NUMBER() OVER (ORDER BY row_max_peak DESC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
```

```
            WHERE time BETWEEN 1728372066.996737 AND 1728372119.978274
        )
        SELECT row_max_peak
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
      )
ORDER BY row_max_peak DESC;
```

Listing B.7: Peak | filename | use-time = false

```
SELECT filename, row_idx, time, row_max_peak
FROM entries_dev0
WHERE filename = 'LISA_20241008_072100.h5'
  AND row_max_peak >= (
        WITH ranked AS (
            SELECT row_max_peak,
                   ROW_NUMBER() OVER (ORDER BY row_max_peak DESC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
            WHERE filename='LISA_20241008_072100.h5'
        )
        SELECT row_max_peak
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
      )
ORDER BY row_max_peak DESC;
```

Listing B.8: Peak | filename | use-time = true

```
SELECT filename, row_idx, time, row_max_peak
FROM entries_dev0
WHERE filename = 'LISA_20241008_072100.h5'
  AND time BETWEEN 1728372066.996737 AND 1728372119.978274
  AND row_max_peak >= (
        WITH ranked AS (
            SELECT row_max_peak,
                   ROW_NUMBER() OVER (ORDER BY row_max_peak DESC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
            WHERE filename='LISA_20241008_072100.h5'
              AND time BETWEEN 1728372066.996737 AND 1728372119.978274
        )
        SELECT row_max_peak
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
      )
ORDER BY row_max_peak DESC;
```

3) Background Bottom-10%

Listing B.9: BG | all | use-time = false

```
SELECT filename, row_idx, time, row_min_background
FROM entries_dev0
WHERE row_min_background != 0
  AND row_min_background <= (
        WITH ranked AS (
            SELECT row_min_background,
                   ROW_NUMBER() OVER (ORDER BY row_min_background ASC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
        )
        SELECT row_min_background
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
```

```
      )
ORDER BY row_min_background ASC;
```

Listing B.10: BG | all | use-time = true

```
SELECT filename, row_idx, time, row_min_background
FROM entries_dev0
WHERE row_min_background != 0
  AND time BETWEEN 1728372066.996737 AND 1728372119.978274
  AND row_min_background <= (
        WITH ranked AS (
            SELECT row_min_background,
                   ROW_NUMBER() OVER (ORDER BY row_min_background ASC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
            WHERE time BETWEEN 1728372066.996737 AND 1728372119.978274
        )
        SELECT row_min_background
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
      )
ORDER BY row_min_background ASC;
```

Listing B.11: BG | filename | use-time = false

```
SELECT filename, row_idx, time, row_min_background
FROM entries_dev0
WHERE filename = 'LISA_20241008_072100.h5'
  AND row_min_background != 0
  AND row_min_background <= (
        WITH ranked AS (
            SELECT row_min_background,
                   ROW_NUMBER() OVER (ORDER BY row_min_background ASC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
            WHERE filename='LISA_20241008_072100.h5'
        )
        SELECT row_min_background
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
      )
ORDER BY row_min_background ASC;
```

Listing B.12: BG | filename | use-time = true

```
SELECT filename, row_idx, time, row_min_background
FROM entries_dev0
WHERE filename = 'LISA_20241008_072100.h5'
  AND row_min_background != 0
  AND time BETWEEN 1728372066.996737 AND 1728372119.978274
  AND row_min_background <= (
        WITH ranked AS (
            SELECT row_min_background,
                   ROW_NUMBER() OVER (ORDER BY row_min_background ASC) AS rn,
                   COUNT(*) OVER () AS cnt
            FROM entries_dev0
            WHERE filename='LISA_20241008_072100.h5'
              AND time BETWEEN 1728372066.996737 AND 1728372119.978274
        )
        SELECT row_min_background
        FROM ranked
        WHERE rn = CAST((cnt*10.0/100.0) AS INT)
      )
ORDER BY row_min_background ASC;
```

Anhang C

Hierarchische Organisation einer HDF5-Datei

Die nachfolgende Abbildung zeigt die vollständige hierarchische Struktur einer Projektdatei im HDF5-Format. Während im Hauptteil der Arbeit nur die für die Abfragen relevanten Gruppen und Datensätze betrachtet wurden, ist hier die gesamte Organisation zur Vollständigkeit dargestellt.

Abbildung C.1: Komplette hierarchische Struktur einer HDF5-Projektdatei.

Selbständigkeitserklärung

Im Rahmen der Anfertigung dieser Bachelorarbeit wurden Werkzeuge (z. B. ChatGPT, DeepL) verwendet. Der Einsatz beschränkte sich auf die sprachliche Überprüfung, die Verbesserung der Verständlichkeit sowie die Strukturierung einzelner Textabschnitte. Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, 28. September 2025

Unterschrift