BHT Berliner Hochschule für Technik

Analyse und Optimierung der Datenbankschicht des Content-Management-Systems TYPO3

Unter besonderer Berücksichtigung von rekursiven Tabellenausdrücken

Bachelor-Thesis

vorgelegt von

Adrian Schönfeld

dem Fachbereich Informatik und Medien
der Berliner Hochschule für Technik Berlin
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

im Studiengang

Medieninformatik

Tag der Abgabe 30. Dezember 2024

Studiere Zukunft

Betreuer
Sebastian Kreideweiß, M.Sc. — Technische Hochschule Brandenburg
Roland Petrasch, Dr. rer. nat. — Berliner Hochschule für Technik

Gutachter
Alexander Löser, Prof. Dr.-Ing. habil. — Berliner Hochschule für Technik

Kurzfassung

Die vorliegende Arbeit ist eine Analyse und Optimierung der Datenbankschicht, des Content-Management-System TYPO3. Das Ziel ist es, die Performance von TYPO3 zu verbessern. Dabei liegt der Fokus besonders auf der Verwendung von rekursivem SQL. Für die Optimierung werden zwei Methoden aus dem TYPO3 Core, in einer eigenen Extension, deutlich performanter umgesetzt. Die Vergleichsmessungen wurden mit zuvor festgelegten Messkriterien und Messverfahren durchgeführt, welche die Performance beschreiben. Die Arbeit stellt ein allgemeines Konzept zur Nutzung von rekursiven SQL-Statements in TYPO3 dar und lässt sich für andere Anwendungsfälle und andere CMS leicht adaptieren. Der entstandene Prototyp und alle dazugehörigen Dateien stehen unter *https://github.com/ASchoenfeld-BHT/TYPO3_recursive_cte* zur Verfügung.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und noch nicht veröffentlicht.

Berlin, 30.12.2024
Adrian Schönfeld

Inhaltsverzeichnis

Abbildungsverzeichnis

Formelverzeichnis

Codeverzeichnis

Tabellenverzeichnis

Glossar

Begri	Beschreibung
Common table expression	Eine CTE ist ein temporärer, benannter Ergebnissatz eines Views oder SQL-Statements, z.B. mittels *WITH [RECURSIVE]*.
Content-Management-System	Ein CMS ist eine Software zur Erstellung, Verwaltung, Bearbeitung und Ausgabe von Inhalten.
Datenbankmanagementsystem	Ein Datenbankmanagementsystem ist eine Software zur Erstellung und Verwaltung von Datenbanken, z.B. MySQL und PostgreSQL.
DDEV	DDEV ist ein Tool für PHP, zum Ausführen von lokalen Entwicklungsumgebungen mittels Docker.
depth-rst search	Die depth-first search oder auch Tiefensuche ist ein Verfahren zum Suchen von Knoten in einem Graphen. (vgl. [1])
Docker	Docker ist eine Software zur Isolierung von Anwendungen. Die Isolierung erfolgt dabei auf mittels Containervirtualisierung und nicht, wie z.B. bei VMware mittels Betriebssystemvirtualisierung.
Doctrine DBAL	Doctrine DBAL ist eine PHP Bibliothek zur Verwaltung von Datenbankverbindungen und Ausführung von SQL-Queries, über verschiedene DBMS hinweg.
Extension	Im Kontext von TYPO3 sind Extensiond Erweiterungen des CMS, welche die Funktionen des CMS erweitern.
Government Site Builder	Der Government Site Builder ist die zentrale CMS-Lösung der deutschen Bundesverwaltung.
MySQL Workbench	MySQL Workbench ist eine Software mit grafische Benutzeroberfläche um mit MySQL-Datenbanken zu arbeiten.
Query Builder	Der Query Builder ist ein Teil von Doctrine DBAL um SQL-Queries an das jeweilige DBMS zu senden.
Round Robin	Das Round-Robin-Verfahren ist ein Rundlaufverfahren, das allen Elementen einer Gruppe gleichmäßig und in einer bestimmten Abfolge, Zugriff auf eine Ressource gibt. (vgl. [2])

Begri	**Beschreibung**
Seite	Eine Seite, oder auch Page, ist ein Element im TYPO3 Seitenbaum und beinhaltet Contentelemente oder dient zur Strukturierung des Seitenbaums, z.B. als Trennlinie oder Ordner.
Seitenbaum	Der Seitenbaum in TYPO3 listet die Hierarchie der Seiten/Pages auf.
SQL-Query	Eine SQL-Query ist ein Block aus einem oder mehreren Befehlen, der von einem Datenbankmanagementsystem verarbeitet wird.
SQL-Statement	Ein SQL-Statement ist ein einzelner Befehl, der von einem Datenbankmanagementsystem verarbeitet wird.
Table Con guration Array	Das Table Configuration Array ist ein Array in PHP, mit dem die Struktur von Datenbanktabellen und -feldern in TYPO3 definiert werden.
TYPO3 Instanz	Eine TYPO3 Instanz ist eine lauffähige CMS Anwendung auf Basis von TYPO3
Windows-Subsystem für Linux	Das Windows-Subsystem für Linux ist ein Feature von Windows, welches ein Linux-System virtualisiert.
Workspace	Ein Workspace im Kontext von TYPO3, ist eine separate Arbeitsumgebung, welche den Inhalt getrennt von der Live-Umgebung und anderen Arbeitsumgebungen verwaltet.

Abkürzungsverzeichnis

Begri□	Beschreibung
CMS	Content-Management-System
CTE	common table expression
DBAL	Database Abstraction Layer
DBMS	Datenbankmanagementsystem
DFS	depth-first search
GSB	Government Site Builder
pid	parent identifier
TCA	Table Configuration Array
uid	unique identifier
WSL	Windows-Subsystem für Linux

Kapitel 1

Einleitung

In dieser Arbeit wird die Entwicklung und Implementierung einer TYPO3 Extension zur rekursiven Abfrage des TYPO3 Seitenbaumes, mittels recursive CTE (common table expression), vorgestellt.

Die entwickelte Extension soll die praktische Umsetzbarkeit von recursive CTE in TYPO3 aufzeigen. Zudem soll die Performance der Extension mit ähnlichen Methoden des TYPO3 Cores verglichen werden.

1.1 Motivation

Content-Management-Systeme (CMS) bilden heutzutage mit fast 70% die Basis aller Webanwendungen [3]. Mit ihnen ist es Betreibern von Webseiten möglich, ohne Programmierkenntnisse, Inhalte zu erstellten und zu verwalten. Im Bereich der öffentlichen Verwaltung hat sich, in Deutschland, das CMS TYPO3 seit vielen Jahren etabliert [4]. TYPO3 ist ein open source CMS, das seit 2001 entwickelt wird.

TYPO3 wird künftig auch die Basis des Government Site Builder (GSB) bilden. Der GSB ist die zentrale Content-Management-Lösung für die Webangebote der deutschen Bundesverwaltung [5]. Er basierte zunächst auf einem kommerziellen CMS. Der GSB 11, mit TYPO3 als Basis, befindet sich aktuell in der Entwicklung.

Daten, die durch TYPO3 verwaltet werden, werden in einer Datenbank gespeichert. Die Datensätze weisen dabei an mehreren Stellen hierarchische Strukturen auf. Diese Strukturen können dazu führen, dass die jeweilige TYPO3 Instanz schlecht skaliert, da diese Strukturen rekursiv auflöst werden müssen. Aktuell werden hierarchische Datenbankstrukturen durch TYPO3 selbst aufgelöst, d.h. das Ergebnis eines Datenbankzugriffs wird von TYPO3 verarbeitet und es erfolgt ein weiterer Datenbankzugriff auf die nächste Rekursionsebene. Je mehr Hierarchien existieren, desto mehr Datenbankzugriffe werden durchgeführt. Dadurch kann es zu merklichen Performanceeinschränkungen kommen.

Es ist davon auszugehen, dass TYPO3 Instanzen in Zukunft immer größere Datensätze verwalten werden. Ebenso ist anzunehmen, dass die deutsche Bundesverwaltung große Datenmengen mit dem GSB 11, mit TYPO3 als Basis, verwalten wird. Dies führt bei Anwendungen mit vielen hierarchischen Datenstrukturen bzw. Zugriffen auf diese Strukturen zu Performanceeinschränkungen und einer sinkenden Akzeptanz von TYPO3.

In TYPO3 werden bereits an verschiedenen Stellen Gegenmaßnahmen zu den Performanceeinschränkungen ergriffen, wie z.B. Caching, aber es lassen sich nicht immer Wartezeiten aufgrund von Datenbankzugriffen vermeiden.

1.2 Zielsetzung

Das Ziel ist es, die Umsetzbarkeit von recursive CTE in TYPO3 aufzuzeigen und zugleich einen deutlichen Performancegewinn zu erreichen. Die Umsetzung soll später in den produktiven Code von TYPO3 übernommen werden.

Für die Umsetzung sollen im Quellcode von TYPO3 Methoden identifiziert werden, die auf hierarchische Strukturen in der Datenbank zugreifen. Eine identifizierte Methode soll beispielhaft in einer eigenen TYPO3 Extension, unter Verwendung einer recursive CTE, umgesetzt werden. Der Aufwand zum Auflösen der hierarchischen Strukturen soll dadurch in die Datenbank verlagert werden. Dadurch erhält die Methode das rekursiv aufgelöste Ergebnis von der Datenbank und muss dieses nicht selbst mit vielen einzelnen SQL Abfragen umsetzen. Die Anzahl der Datenbankzugriffe soll dadurch deutlich verringert werden und es soll ein Performancegewinn erreicht werden.
Die Auswirkungen auf die Performance soll anhand zuvor definierter Parameter untersucht und beziffert werden.

Die Arbeit ist ein Baustein, um die Akzeptanz von TYPO3 als open source CMS zu steigern. Dadurch profitieren alle Projekte, die auf dem CMS aufbauen, wie z.B. der GSB 11.

Ziel ist es, eine allgemeingültige Performancesteigerung zu erreichen, die sich auf die meisten TYPO3 Instanzen anwenden lässt. Es ist nicht das Ziel Hierarchien im Datenbestand einer TYPO3 Instanz aufzulösen um die Anzahl der SQL Abfragen zu reduzieren. Ebenso sollen keine Datensätze gelöscht werden um den Datenbestand zu verringern.

1.3 Hypothese

Die Verwendung einer recursive CTE soll beim Auflösen von hierarchischen Datenbankstrukturen deutlich performanter sein als die bisherigen Implementierungen in TYPO3, welche mehrere einzelne SQL-Statements nutzen. Es wird vermutet, dass sich durch die Verwendung von recursive CTE ein merklicher Performancegewinn der TYPO3 Instanz einstellt.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen zu SQL und zum Content Management System TYPO3 erläutert.

2.1 SQL

Im Folgenden finden sich Definitionen von Begriffen. Zudem werden die Unterschiede zwischen den Befehlstypen in SQL aufgezeigt. Die Konzepte von prepared Statements und rekursivem SQL werden ebenfalls vorgestellt.

2.1.1 SQL Begri□lichkeiten

Für diese Arbeit ist ein gemeinsames Verständnis der SQL Begrifflichkeiten wichtig. Oftmals werden Begriffe als Synonym verwendet oder gänzlich anders verstanden. Aufgrund dessen folgt eine Erläuterung der wichtigsten Begriffe, so wie sie für diese Arbeit zu verstehen sind.

Statement

Ein SQL-Statement ist ein einzelner Befehl. Der Befehl beginnt mit dem ersten Wort und endet mit einem Semikolon (;). Es können auch mehrere Statements in einer Zeile ausgeführt werden, die mit einem Semikolon getrennt werden. (vgl. [6])

```
1 SELECT 1;
```

Code 2.1: *Beispiel SQL-Statement*

Command/Befehl

Ein Command ist jede Art von Aktion, die für ein SQL Objekt ausgeführt wird. Dazu zählen auch Statements. Commands unterteilen die structured query language nochmals in Sublanguages. Jede Sublanguage ist dabei für bestimmte Aktionen zuständig. Die vier Hauptkategorien sind in Tabelle 2.1 dargestellt. (vgl. [6])

Sublanguage	Erläuterung
Data Definition Language (DDL)	Befehle um Datenbankobjekte zu erstellen oder zu modifizieren.
Data Manipulation Language (DML)	Befehle um Werte abzufragen, einzufügen oder zu löschen.
Data Control Language (DCL)	Befehle um Berechtigungen zu vergeben oder zu löschen.
Transaction Control Language (TCL)	Befehle um Datenbankobjekte zu speichern oder wiederherzustellen.

Tab. 2.1: *SQL-Sublanguages [6]*

In der Tabelle 2.2 sind einige Commands ihren Sublanguages zugeordnet. So kann man sagen, dass die SELECT, UPDATE, und DELETE Statements zur Gruppe der DML Commands gehören. Jedes Statement ist also auch ein Command.

DDL	DML	DCL	TCL
CREATE	SELECT	GRANT	COMMIT
ALTER	INSERT	REVOKE	ROLLBACK
DROP	UPDATE		SAVEPOINT
TRUNCATE	DELETE		SET TRANSACTION
COMMENT	MERGE		
RENAME	CALL		
	EXPLAIN PLAN		
	LOCK TABLE		

Tab. 2.2: *Zuordnung von Befehlen zu SQL-Sublanguages [6]*

Query

Eine Query ist jede Art von Aktion, die für eine Datenbank oder ein Datenbankobjekt ausgeführt wird. Die Query wird als Block ausgeführt und kann dabei z.B. aus mehreren Statements bestehen. Die folgenden Statements können zusammen als Query ausgeführt werden. Sie können aber auch einzeln ausgeführt werden und dabei als zwei separate Queries angesehen werden. Jedes Statement ist also auch eine Query. (vgl. [6])

```
1 SELECT 1;
2 SELECT 2;
```

Code 2.2: *SQL Beispielquery*

Clause/Klausel

Eine Klausel ist eine Bedingung innerhalb eines Statements. Die Klausel wird meistens verwendet um die Daten zu filtern. Zu den Klauseln gehören z.B. FROM, WHERE, HAVING, GROUP BY, LIKE oder AND. (vgl. [6])

2.1.2 Command Types

SQL-Statements werden in den Logs von MariaDB in verschiedene *command_types* kategorisiert. So kann in den Logs besser unterschieden werden um welche Art Statement es sich handelt, z.B. den Aufbau einer Datenbankverbindung, das Ausführen eines Statements oder das Beenden einer Datenbankverbindung. In den Logs ist jedes Statement aufgeführt das die Datenbank ausgeführt hat. Die Art und Anzahl der ausgeführten Statements wird in dieser Arbeit vor allem für den

Vergleich zwischen verschiedenen TYPO3 Methoden benötigt. Die wichtigsten *command_types* in MariaDB sind in Tabelle 2.3 aufgeführt.

command_type	Erläuterung
Connect	Verbindungsaufbau zur Datenbank.
Quit	Verbindung zur Datenbank wird geschlossen.
Query	SQL-Statement wird direkt ausgeführt (unprepared Statement).
Prepare	Prepared Statement wird erstellt.
Execute	Prepared Statement wird ausgeführt.
Close stmt	Prepared Statement wird geschlossen/gelöscht.

Tab. 2.3: *Übersicht wichtiger SQL command_types [7]*

2.1.3 Prepared Statements

Ein SQL Befehl kann auf verschiedene Arten ausgeführt werden. Als command_type "Query" wird das Statement direkt von der Datenbank ausgeführt. Sobald der Befehl eine Variable beinhaltet, ist diese Variante anfällig für SQL-Injections. Die Variable könnte bei der Eingabe zum Beispiel so gewählt werden, dass die Benutzernamen und Passwörter der Datenbankbenutzer ausgegeben werden. Der Code 2.3 veranschaulicht eine solche SQL-Injection.

```
1      // Parameter mit SQL-Injection
2      $param = "12 UNION SELECT username, password FROM be_users";
3
4      // Die Query erwartet nur einen Zahlenwert für $param
5      $stmt_injected = $pdo->query("SELECT uid, title FROM pages WHERE uid=$param")
           ;
```

Code 2.3: *Beispielcode für SQL-Injection*

command_type	argument
Query	SELECT uid, title FROM pages WHERE uid=12 UNION SELECT username, password FROM be_users

Tab. 2.4: *Generierte Query aus Code 2.3*

uid	title
12	1.2.3
admin	$argon2i$v=19$m=65536,t=16,p=1$WHFta3lCR0pGCm1VNEJ...
cli	$argon2i$v=19$m=65536,t=16,p=1$RGHFQUgwaTI2WVVpT0x...

Tab. 2.5: *Ausgabe von Code 2.3*

Um die Gefahr von SQL-Injections deutlich zu reduzieren werden prepared Statements eingesetzt. Diese können im Gegensatz zu einer Query zwischen Werten und Operanten unterscheiden. Dafür muss zunächst das prepared Statement mit Platzhaltern, für die späteren Werte, in der Datenbank **erstellt** werden. Danach kann das prepared Statement mehrmalig mit den gewünschten Variablen **ausgeführt** werden. Sollten die Variablenwerte SQL-Statements enthalten, werden diese nicht ausgeführt. Zum Schluss werden die prepared Statements **geschlossen**.
Es werden also insgesamt **mindestens 3 Statements an die Datenbank gesendet** um ein prepared Statement zu nutzen.

```php
// Parameter mit SQL-Injection
$param = "12 UNION SELECT username, password FROM be_users";

// Prepared statement erstellen
$stmt_prepared = $mysqli->prepare("SELECT uid, title FROM pages WHERE uid = ?
    ");

// $param als Wert übergeben
$stmt_prepared->bind_param("i", $param); // "i"=integer

// Prepared Statements ausführen
$stmt_prepared->execute();

// Prepared statement schließen
$stmt_prepared->close();
```

Code 2.4: *Beispielcode für prepared Statement*

command_type	argument
Prepare	SELECT uid, title FROM pages WHERE uid = ?
Execute	SELECT uid, title FROM pages WHERE uid = 12
Close stmt	

Tab. 2.6: *Generierte Statements aus Code 2.4*

uid	title
12	1.2.3

Tab. 2.7: *Ausgabe von Code 2.4*

Auch wenn Doctrine DBAL in TYPO3 für die Datenbankkommunikation genutzt wird, müssen die Variablen der SQL-Befehle als prepared Statement formuliert werden, um Sicherheitsrisiken durch SQL-Injections zu minimieren.

2.1.4 Rekursives SQL

Mit dem SQL Standard 1999 (SQL3) wurde die *WITH [RECURSIVE]*-Klausel eingeführt. Mit der WITH-Klausel können eine oder mehrere **benannte, temporäre Tabellen** im DBMS angelegt werden. Auf diese Tabellen kann wie auf herkömmliche Tabellen referenziert werden. Die WITH-Klausel wird deswegen auch als common table expression (CTE) bezeichnet. Es können auch mehrere CTE's ineinander verschachtelt werden. Die Tabellen existieren nur innerhalb des Statements und werden nach der Ausführung gelöscht. Sie können nicht von anderen Statements genutzt werden.
Die WITH RECURSIVE-Klausel ist eine Sonderform der WITH-Klausel. Mit ihr können ebenfalls eine oder mehrere benannte, temporäre Tabellen erstellt werden, die zudem auf sich selbst verweisen können (aber nicht müssen). Dadurch können solche Tabellen ein Ergebnis liefern, das wiederum als Ausgangstabelle für die nächste Abfrage genutzt wird. Diese Rekursion endet erst, sobald eine Ergebnistabelle leer ist, also den Wert "NULL" zurück gibt.

Als Beispiel dient die fiktive Fußballtabelle 2.8, in der mehrere Ligen zusammengefasst sind. Um den Verein mit der besten Tordifferenz, aus der ersten Liga zu ermitteln, kann z.B. eine CTE genutzt werden.

ID	Verein	Tordi□erenz	Liga
17	Eintracht Frankfurt	2	1
5	Borussia Dortmund	4	1
13	FC Schalke 04	10	2
43	Hansa Rostock	6	3
9	VfL Bochum 1848	2	1
8	Hamburger SV	2	2
19	Saarbrücken	5	3
12	VfB Stuttgart	8	1
76	SC Paderborn 07	5	2
54	VfL Osnabrück	1	3

Tab. 2.8: *Beispieltabelle für rekursives SQL*

```
1  -- RECURSIVE wird angegeben, aber nicht angewendet
2  WITH RECURSIVE
3     -- temporäre Tabelle 'Liga1' wird erstellt
4     Liga1 AS (
5        SELECT * FROM liveDemo_fussball WHERE Liga = 1
6     ),
7     -- temp. Tabelle 'Beste_Tordifferenz' wird erstellt
8     Beste_Tordifferenz AS (
9        -- Referenz auf temp. Tabelle 'Liga1'
10       SELECT * FROM Liga1 WHERE Tordifferenz = (SELECT MAX(Tordifferenz) FROM
             Liga1)
11    )
12 -- Ausgabe der temp. Tabelle 'Beste_Tordifferenz'
13 SELECT * FROM Beste_Tordifferenz;
```

Code 2.5: *Unterteilung von Tabelle 2.8 mittels WITH RECURSIVE-Klausel*

ID	Verein	Tordi□erenz	Liga
12	VfB Stuttgart	8	1

Tab. 2.9: *Ausgabe von Code 2.5*

2.2 TYPO3

Im Folgenden werden Konzepte von TYPO3 vorgestellt, die für diese Arbeit relevant sind.

2.2.1 Seitenbaum

Um Inhalte in TYPO3 zu verwalten werden Seiten/Pages genutzt. Eine Page kann z.B. Textelemente und Bilder enthalten. Pages können auch ineinander verschachtelt werden um z.B. Inhalte zu gruppieren oder Berechtigungskonzepte abzubilden. Es gibt dabei nicht nur klassische (Web)Seiten, auf denen Inhalte platziert werden können, sondern auch Seiten die der Organisation im Backend dienen, z.B. optische Trennlinien und Ordner. Alle Pages ergeben zusammen den Seitenbaum, der in TYPO3 über das Page-Modul bearbeitet werden kann.

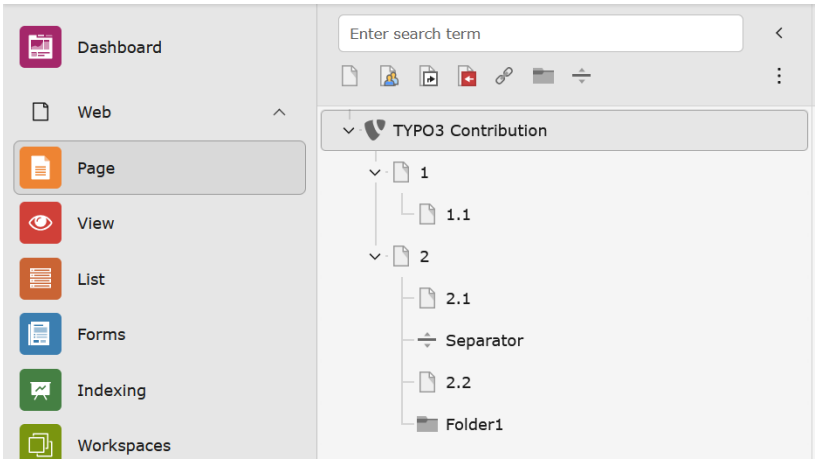


Abb. 2.1: *TYPO3 Seitenbaum*

2.2.2 Workspaces

Für die Versionierung und Veröffentlichung von Inhalten bietet TYPO3 das Feature der Workspaces. Ein Workspace stellt eine eigene Version des Inhaltes inkl. des Seitenbaumes dar. In TYPO3 können somit mehrere Seitenbäume existieren und müssen für den jeweiligen Workspace geladen werden. In diesem können Pages bearbeitet, verschoben, gelöscht oder erstellt werden, ohne den Live-Workspace zu verändern (vgl. [8]). Wenn ein Workspace ganz oder teilweise veröffentlicht werden soll, kann dieser in den Live-Workspace überführt werden. Ein Workspace kann im weitesten Sinne mit einem Branch in GIT verglichen werden.

2.2.3 Extension

Als open source CMS lässt sich TYPO3 an die eigenen Bedürfnisse anpassen. Vereinfacht wird dies durch Funktionspakete, die als Extensions/Erweiterungen installiert werden können. Die Extensions werden durch die TYPO3 Community gepflegt und zentral auf *https://extensions.typo3.org* bereitgestellt. Von dort aus können Funktionen wie z.B. ein “News System” schnell installiert werden, ohne diese selbst programmieren zu müssen. Aktuell werden für die Version 13 ca. 460 Extensions unterstützt [9]. Selbst die Grundinstallation einer TYPO3 Instanz besteht aus Extensions wie “TYPO3 CMS Core” oder “TYPO3 CMS Backend”.

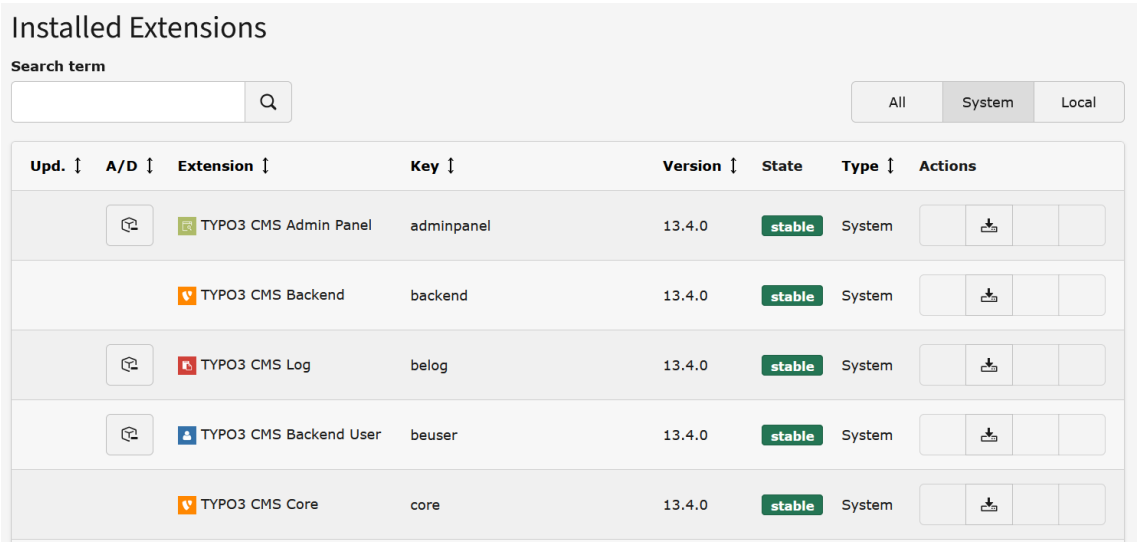


Abb. 2.2: *Extensions einer TYPO3 Core Installation*

2.3 Aufbau der TYPO3 Datenbankschicht

In TYPO3 werden Anwendungsdaten in relationalen Datenbanken gespeichert. In der Version 13.4 werden MariaDB, MySQL, PostgreSQL und SQLite unterstützt [10]. Extensions können ebenfalls eigene Datentabellen enthalten, die in der Systemdatenbank oder einer separaten Datenbank gespeichert werden. Diese Tabellen können auch ganz oder teilweise in separate Datenbanken ausgegliedert werden.

2.3.1 Datenbankschema

Die Tabellen in der TYPO3 Systemdatenbank sind vorwiegend nach ihrer Funktion benannt und besitzen meistens einen Gruppenpräfix, z.B. “be_” für Backend-Tabellen oder “cache_” für Caching-Tabellen.

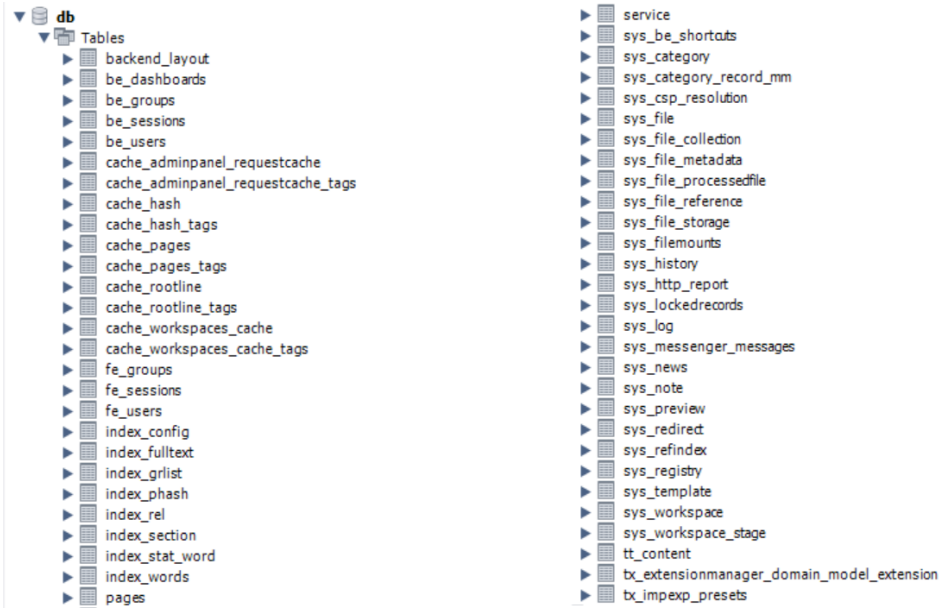


Abb. 2.3: *Übersicht TYPO3 Datenbanktabellen*

Alle Tabellen besitzen einen primary key aber **keinen foreign key**. Somit gibt es auf Datenbankebene keine Beziehungen zwischen den Tabellen um die referentielle Integrität sicherzustellen. Ein Inhaltselement aus der Tabelle *tt_content* enthält z.B. Informationen zu seiner zugehörigen Seite aus der Tabelle *pages*, aber es ist nicht davon abhängig und könnte auch ohne die Seite existieren. Um die Integrität der Datensätze trotzdem zu gewährleisten, werden Beziehungen zwischen Tabellen innerhalb von TYPO3 definiert.

Dafür wird das globale Array $GLOBALS['TCA'] verwendet. In diesem Array wird das Schema der Datenbanktabellen definiert und unter anderem die foreign keys, Feldnamen und Datentypen konfiguriert [11].

Durch die $GLOBALS['TCA'] kennt die Datenbank selbst zwar keine foreign keys und sie verliert die wichtige Aufgabe der referentiellen Integritätsprüfung, aber sie lässt sich flexibler in TYPO3 verwalten. Dies ist für die Entwicklung von Extensions von Vorteil, da die Entwickler recht einfach ein Tabellenschema anlegen können ohne Detailkenntnisse der Datenbank zu benötigen. Datenbankupdates können ebenfalls einfacher gestaltet werden, da hierfür keine Abhängigkeiten innerhalb der Datenbank aufgelöst werden müssen.
Die Tabellen sind aus Sicht der Datenbank nur einzelne Container, in denen jede Art von Daten gespeichert sein könnte. Für eine Überarbeitung des Datenbankschemas reicht z.B. ein ER-Modell alleine nicht aus. Dafür sind zusätzliche, tiefgreifende Kenntnisse von TYPO3 und dessen Funktionen notwendig.

2.3.2 Tabelle pages

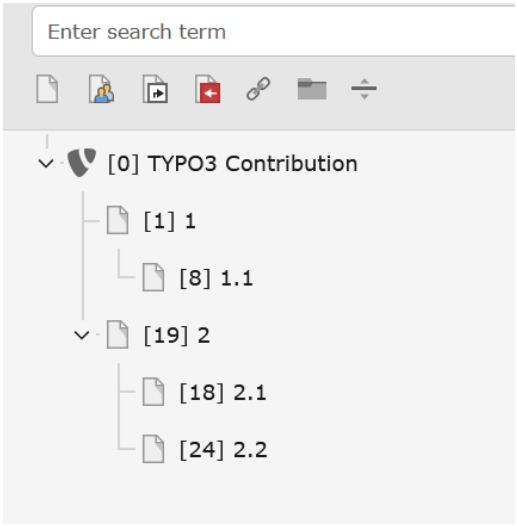
Die Tabelle *pages* umfasst alle Seiten, des TYPO3 Seitenbaumes. Dabei ist jede Seite einer anderen Seite untergeordnet. Die oberste Seite, unter der sich immer alle anderen Seiten befinden, besitzt die uid 0. Diese root-Seite besitzt aber keinen Eintrag in der pages-Tabelle.

Hierarchie

Jede Seite ist im Seitenbaum einer anderen Seite untergeordnet. Die Verknüpfung, zwischen über- und untergeordneter Seite, wird über die Attribute *uid* (unique identifier) und *pid* (parent identifier) hergestellt. Die pid entspricht dabei der uid der übergeordneten Seite. Der Anzeigename der Seite wird im Attribut *title* gespeichert.
In der Abbildung 2.4 ist der Zusammenhang von uid und pid dargestellt. Die uid ist dabei in eckigen Klammern, im Seitennamen dargestellt.

uid	pid	title
1	0	1
8	1	1.1
19	0	2
18	19	2.1
24	19	2.2

pages-Tabelle



Seitenbaum

Abb. 2.4: *Zusammenhang von uid und pid*

Sortierung

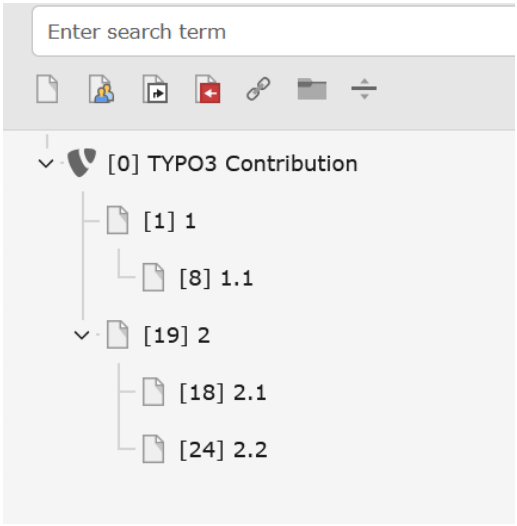
Der Seitenbaum wird TYPO3 intern als Array verarbeitet. Damit er über dieses Array korrekt abgebildet werden kann, ist die Reihenfolge der Elemente von entscheidender Bedeutung. Die Reihenfolge der Arrayelemente muss der Anordnung des Seitenbaumes, von oben nach unten, entsprechen.

Das Attribut *uid* kann nicht zur Sortierung des Arrays genutzt werden, da die Seiten beliebig im Seitenbaum verschoben werden können und dabei ihren primary key (uid) nicht ändern. Die Sortierung wird deswegen über das separate Attribut *sorting* geregelt. Dessen Wert wird je nach Sortierreihenfolge von TYPO3 angepasst. Kleinere *sorting*-Werte werden dabei höher im Seitenbaum einsortiert (näher an uid 0) als höhere Werte.

Das *sorting*-Attribut ist kein globaler Wert, der über alle Seiten in der pages-Tabelle genutzt werden kann. **Die *sorting*-Werte können immer nur zwischen Seiten mit der gleichen *pid* verglichen werden**. Seiten aus anderen Zweigen können völlig andere *sorting*-Werte besitzen (größer, kleiner oder gleich) und haben keine Beziehung zueinander. Andernfalls müssten beim Verschieben einer Seite alle Einträge in der pages-Tabelle angepasst werden.
In der Abbildung 2.5 ist der Zusammenhang von sorting-Attribut und Sortierung dargestellt.

uid	pid	sorting	title
1	0	257	1
8	1	1420	1.1
19	0	514	2
18	19	64	2.1
24	19	96	2.2

pages-Tabelle



Seitenbaum

Abb. 2.5: *Zusammenhang sorting-Attribut und Sortierung*

Versionierung / Workspaces

Einem Workspace sind zunächst keine Seiten zugeordnet. Erst wenn eine Seite im Workspace erstellt, bearbeitet oder gelöscht wird, wird dafür ein **zusätzlicher Eintrag** in der pages-Tabelle erstellt. Der Seitenbaum des Workspace setzt sich also aus dem Live-Workspace und einzelnen, zusätzlichen Seiten zusammen. Für die Workspaces werden die Attribute *t3ver_oid*, *t3ver_wsid*, *t3ver_state* und *t3ver_stage* verwendet. Das Attribut *t3ver_oid* verweist z.B. auf die Live-Version der Seite, wenn diese in einem Workspace geändert wurde.

Der Umstand, dass ein Workspace nur bearbeitete Seiten enthält, ist für die spätere rekursive Umsetzung von großer Bedeutung. Wenn im Workspace z.B. die oberste Seite im Seitenbaum gelöscht wird, wird nur dafür **ein** zusätzlicher Eintrag in der pages-Tabelle erzeugt. Im Seitenbaum müssen aber alle untergeordneten Seiten, für die nur Live-Seiten existieren, ausgeblendet werden.
Wird hingegen eine Seite im Live-Workspace gelöscht, wird das *deleted*-Attribut für **alle** Unterseiten gesetzt.

Im nachfolgenden Beispiel wurde die Seite mit der uid 19 in einem Workspace gelöscht. Die Live-Seite, mit der uid 19 wurde nicht geändert. Es wurde ein neuer Eintrag in der page-Tabelle 2.10 angelegt. Der neue Eintrag besitzt die uid 32 und markiert die Seite mit der uid 19 als gelöscht (t3ver_state=2). Alle Unterseiten müssen im Seitenbaum des Workspace (Abb. 2.6) ebenfalls ausgeblendet werden, ohne dass dafür ein separates Attribut in der Datenbank existiert. Es kann nur aus dem Kontext erschlossen werden, dass die nachfolgenden Seiten in dem Workspace gelöscht wurden.

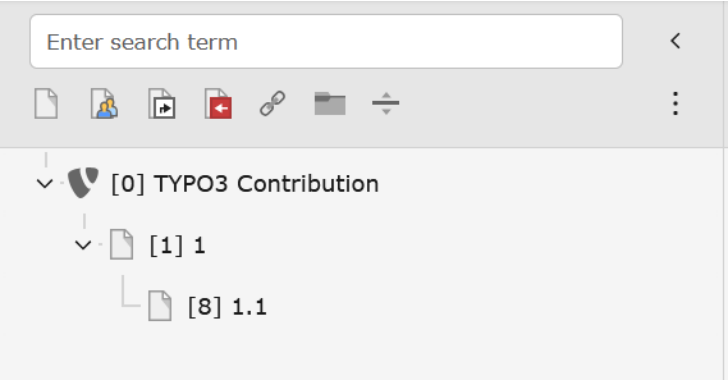


Abb. 2.6: *Seitenbaum für Workspace aus Tab. 2.10*

uid	pid	sorting	title	deleted	t3ver_oid	t3ver_wsid	t3ver_state
1	0	257	1	0	0	0	0
8	1	1420	1.1	0	0	0	0
19	0	514	2	0	0	0	0
18	19	64	2.1	0	0	0	0
24	19	96	2.2	0	0	0	0
32	0	514	2	0	19	9	2

Tab. 2.10: *Pages-Tabelle mit gelöschter Seite im Workspace*

2.3.3 Doctrine DBAL / Querybuilder

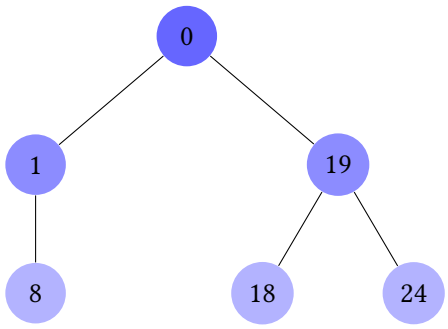
Um die Integration von verschiedenen Datenbanksystemen zu vereinfachen, stellt TYPO3 eine API auf Basis von Doctrine DBAL (Database Abstraction Layer) bereit. Doctrine DBAL ist eine objektorientierte API, um die Kommunikation mit verschiedenen Datenbanksystemen und deren Verbindungen zu vereinheitlichen. Durch diese Abstraktionsebene erhalten Entwickler konsistente Datenbankzugriffe, ohne Kenntnis der verwendeten Datenbank/en und deren Anbindung zu benötigen. Doctrine DBAL nutzt zum Ausführen von SQL-Statements den sog. Querybuilder. Um ein SQL-Statement auszuführen, muss es zuvor mit einer eigenen Syntax an den Querybuilder übergeben werden. Der Querybuilder erstellt daraus das korrekte SQL-Statement, für das verwendete DBMS und führt es aus.

2.3.4 Gewurzelter Baum

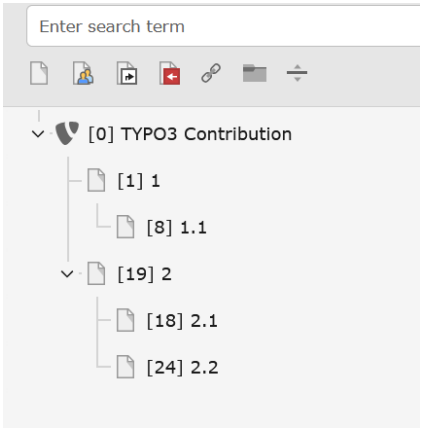
Die Datenstruktur des TYPO3 Seitenbaumes entspricht der, eines *gewurzelten Baumes*.

"*Ein gewurzelter Baum ist in der Graphentheorie ein Baum, dessen Kanten eine ausgezeichnete Richtung besitzen, so dass im Gegensatz zum ungerichteten Baum ein Knoten als Wurzel identi ziert werden kann.*" ([12])

In TYPO3 ist die Wurzel des Seitenbaums die Seite mit der uid 0. Sie ist immer vorhanden. In Abbildung 2.7 ist ein TYPO3 Seitenbaum als Graph dargestellt. Die Knotennamen entsprechen den uid's der Seiten, die in eckigen Klammern, im Seitennamen dargestellt sind.



Seitenbaum als Graph



Seitenbaum

Abb. 2.7: *TYPO3 Seitenbaum als Graph*

In dem Graph erkennt man, dass jeder Teilbaum ebenfalls einem gewurzeltem Baum entspricht. Würde man z.B. die Wurzel 0 entfernen erhält man zwei neue gewurzelte Bäume mit der uid 1 bzw. 19 als neue Wurzel. Dies gilt für alle Knoten in dem Baum.

Abb. 2.8: *Teilbäume aus Abb. 2.7*

2.3.5 Rekursion mittels depth-first search (DFS)

Um jeden Knoten des Baumes zu ermitteln, kann der Baum rekursiv durchwandert werden. Dies kann auf verschiedene Arten vorgenommen werden, z.B. jede Ebene nacheinander (breadth-first search (BFS)). Um den Seitenbaum für TYPO3 zu bilden wird die **depth-first search (DFS)** durchgeführt. Dabei wird der Baum solange in Teilbäume zerlegt, bis dieser nur noch einen einzelnen Knoten besitzt.
Das Vorgehen von DFS kann wie folgt beschrieben werden:

1. Erstelle eine leere Liste, in der die Knoten-ID's gespeichert werden.
2. Füge die ID der Wurzel zur Liste hinzu und markiere sie als "erfasst".
3. Betrachte alle Kindknoten als Wurzel neuer Bäume.
4. Führe Schritt 2 mit allen Kindknoten aus, beginnend beim kleinsten Knoten, der nicht als "erfasst" markiert ist.
5. Fahre fort, bis alle Knoten als "erfasst" markiert sind.

Für das obige Beispiel, aus Abb. 2.7, würden die Knoten in der folgenden Reihenfolge erfasst werden [0, 1, 8, 19, 18, 22]. Die so erstellte Liste deckt sich in der Reihenfolge mit dem TYPO3 Seitenbaum, wenn dieser von oben nach unten gelesen wird.

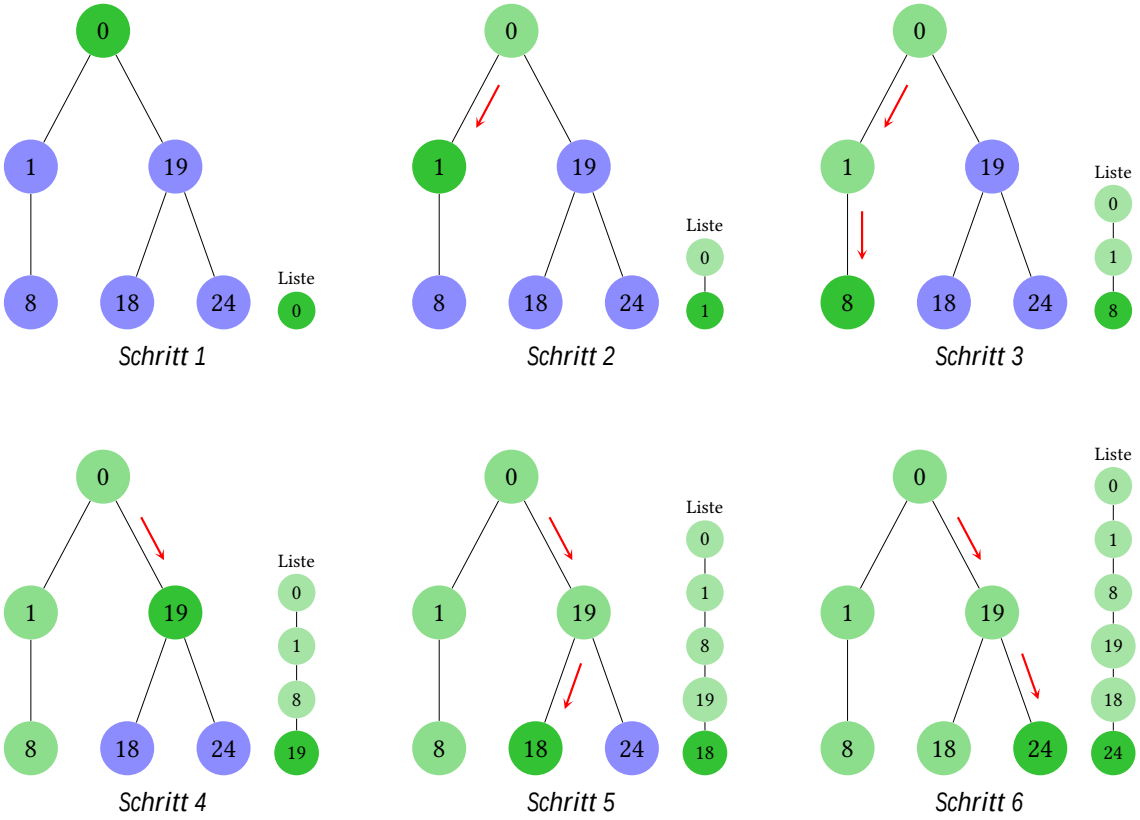


Abb. 2.9: *Gewurzelter Baum als Liste, mittels DFS*

Kapitel 3

Methodik

In diesem Kapitel wird beschrieben, mit welcher Methodik die Hypothese überprüft wird, dass rekursive SQL-Statements die Performance einer TYPO3 Instanz erhöhen. Dazu wird eine TYPO3 Extension entwickelt, welche ausgewählte Methoden aus dem TYPO3 Core mit einem rekursiven SQL-Statement umsetzt. Für dieses Statement wird zunächst ein Schema entwickelt, das sich leicht für andere rekursive Methoden adaptieren lässt.
Die Bestandsmethoden, welche in der Extension mit recursive CTE umgesetzt werden, werden ebenso beschrieben, wie der Aufbau der Test- und Entwicklungsumgebung. Auf die verwendeten Analysetools und Testdaten wird ebenfalls eingegangen.

3.1 Test- und Entwicklungsumgebung

Es wurden zwei lokale TYPO3 Instanzen, in der Version 13.4.0, als Test- und Entwicklungsumgebung installiert. In der Testinstanz wurde ein großer Testdatensatz importiert, welcher sich entgegen den Datensätzen in der Entwicklungsinstanz, nicht verändert. Die Installation der Instanzen erfolgte mittels DDEV als Docker Container [13]. Als Betriebssystem wurde Windows10 mit WSL2 verwendet. Die Nutzung von Docker hat den Vorteil, dass kein separater Webserver und keine Datenbank installiert werden müssen, sondern als Container gestartet werden können. Der Betrieb von TYPO3 inkl. Datenbank als Docker Container hat den Nachteil, dass die Performance darunter leiden kann und sollte deswegen nicht im Live-Betrieb genutzt werden.
Die verminderte Performance der Docker Container ist für diese Arbeit nicht relevant. Die Auswirkung der rekursiven Datenbankabfragen werden in Relation zu den vorhandenen Methoden gestellt, die sich in der selben Docker Instanz befinden.

Software	Version
Windows10	22H2 Build 19045.5131
WSL	2.3.26.0
WSL-Distribution	Ubuntu-20.04
WSL-Kernelversion	5.15.167.4-1
Docker	27.1.1
Docker Desktop	27.1.1
DDEV	v1.23.3
Apache Server (Container)	2.4.59 (Debian)
PHP (Container)	8.2
NodeJS (Container)	22
mariaDB (Container)	10.11
MySQL Workbench	8.0.38 Community
Visual Studio Code	1.95.3

Tab. 3.1: *Softwareübersicht der Entwicklungsumgebung*

3.2 Analysetools

Als Analysetools werden MySQL Workbench und die PHP Funktion *hrtime()* eingesetzt.
Mit MySQL Workbench wird das Datenbankschema analysiert. Das Logging der Datenbank wird aktiviert und in der separaten Loggingtabelle mysql.general_log gespeichert. Diese Logs werden ebenfalls mit MySQL Workbench ausgewertet.
Um die Laufzeit der TYPO3 Core Methoden mit der erstellten Extension zu vergleichen, wird die PHP Funktion hrtime() genutzt. Diese ist in PHP seit Version 7.3 verfügbar und somit auch in den TYPO3 Instanzen mit PHP 8.2.

3.3 Testdatensatz

Für die Performancetests wird ein Datensatz genutzt, der von Christian Kuhn bereitgestellt wurde. Er stammt aus einer produktiven TYPO3 Instanz und beinhaltet nur die pages-Tabelle. Der Datensatz stammt aus dem Jahr 2015 und besitzt ein Schema, dass sich von der TYPO3 Version 13.4.0 unterscheidet. Zudem wurden in dieser Instanz keine Workspaces genutzt. Das Schema des Datensatzes wurde mit dem TYPO3 Feature “Analyze Database Structure” für die Version 13.4.0 angepasst. Außerdem wurde der Datensatz zuvor anonymisiert.

3.4 Fachgespräche mit TYPO3 Entwicklern

Im Rahmen dieser Bachelorarbeit wurden Fachgespräche mit Entwicklern des TYPO3 Cores geführt. Ziel dieser Gespräche war es Detailinformationen zur Funktionsweise von TYPO3 zu gewinnen. Die Gespräche wurden vornehmlich mit Christian Kuhn und Stefan Bürk geführt, die über ein umfangreiches Fachwissen verfügen und seit vielen Jahren bei der Entwicklung des TYPO3 Cores mitwirken.

Die gewonnenen Informationen haben besonders dazu beigetragen die WITH[RECURSIVE] Funktion des Querybuilders nachzuvollziehen. Diese Funktion existiert nicht in der offiziellen Version von Doctrine DBAL und ist von TYPO3 aktuell weder zur allgemeinen Verwendung vorgesehen, noch dokumentiert.

3.5 Bestandsmethoden

In TYPO3 sind bereits mehrere Methoden implementiert, welche den Seitenbaum rekursiv durchwandern. Die Rekursionsebene bezieht sich dabei auf die PHP-Methode, welche für jede Rekursion mindestens ein SQL-Statement ausführt. Falls die Methode weitere Informationen aus der Datenbank abfragt, können auch mehrere SQL-Statements pro Rekursion ausgeführt werden.

Von den TYPO3 Methoden sind vor allem die Methoden *PageRepository->getPageIdsRecursive()* und *PageTreeView->getTree()* interessant. Beide Methoden haben eine ähnliche Funktion und greifen auf wenig andere Methoden innerhalb von TYPO3 zu [14][15]. Somit dienen diese zwei Methoden als Referenz, mit der die Extension später verglichen wird. *PageTreeView->getTree()* wird z.B. vom PermissionController genutzt, um die Seitenberechtigungen im Backend aufzulisten.

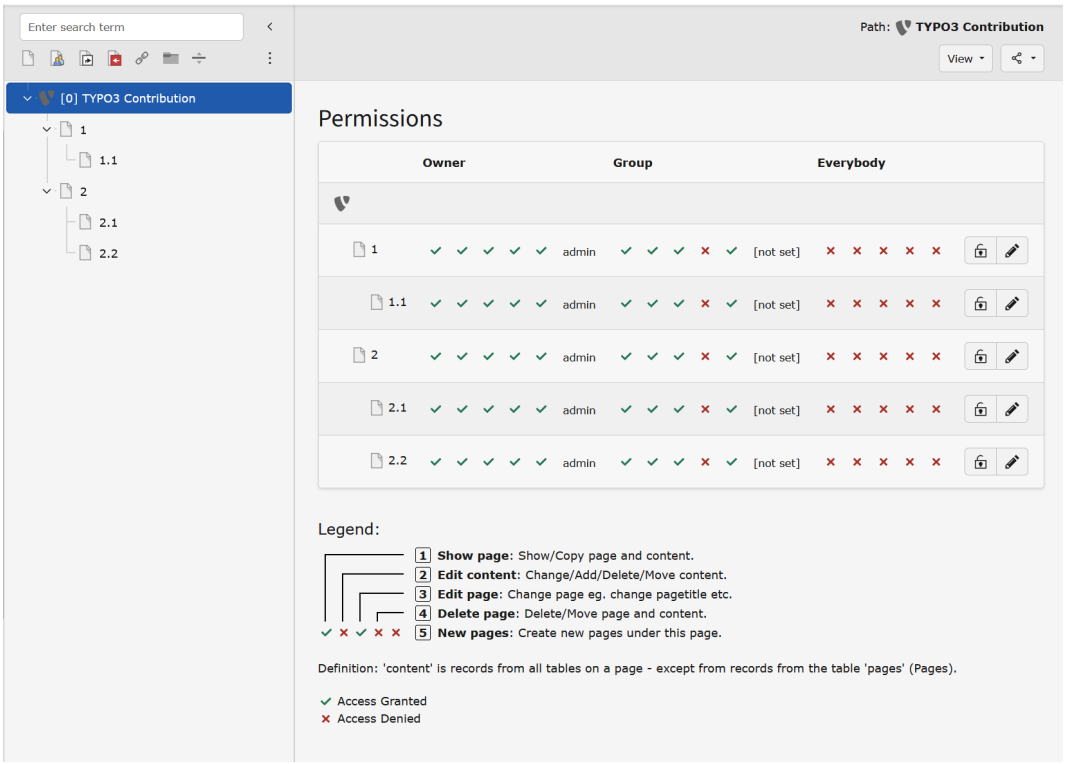


Abb. 3.1: *Permissions-Modul*

3.6 Rekursives SQL-Statement

Aufgrund der Struktur der pages-Tabelle werden mehrere Select-Statements benötigt, um alle Datensätze für einen Workspace zu ermitteln. Wenn diese Selects direkt im rekursiven Teil der CTE verwendet werden, wäre das Statement unübersichtlich und nur schwer wartbar. Deswegen wird die pages Tabelle mit der WITH-Klausel in mehrere, temporäre Tabellen unterteilt.

Die Unterteilung erfolgt dabei in drei Schritten. Im ersten Schritt wird eine **bereinigte Datengrundlage** erstellt. Anschließend lassen sich aus dieser Datengrundlage die benötigten **Teildaten bilden**. Im dritten Schritt werden die Teildaten zu einer **Datentabelle** zusammengefügt. Diese Datentabelle wird anschließend als Input für die rekursive Abfrage genutzt.

Dieses Unterteilungsschema lässt sich flexibel anpassen, sodass neben den Workspaces auch andere Unterteilungskriterien berücksichtigt werden können, z.B. Berechtigungen. Außerdem kann

durch diese Aufteilung besser auf Änderungen im Datenbankschema reagiert werden, ohne das Statement komplett zu überarbeiten.

3.6.1 Bereinigte Datengrundlage erstellen

Das Ziel bei der Erstellung der Datengrundlage ist es, eine **vollständige Auswahl** der relevanten Datensätze zu erfassen. In dieser Arbeit sind die Workspaces das zentrale Auswahlkriterium. Es werden also nur Datensätze des aktuellen Workspace und des Live-Workspace (Workspace 0) in temporären Tabellen gespeichert. Alle anderen Workspaces werden nicht erfasst, ebenso wie gelöschte Live-Pages.

In der Datengrundlage wird bereits festgelegt, welche Felder im Endergebnis ausgewählt werden können. Zur Vereinfachung werden hier alle relevanten Felder für diese Arbeit ausgewählt. In der TYPO3 Extension werden später nur notwendige Felder selektiert. Der aktuelle Workspace wird hier mit der Variabel *@selectedWS* simuliert.

```
1  SET @selectedWS = CAST(3 AS int);
2
3  WITH
4    -- Step 1: Get all live pages
5    getAllLivePages AS (
6       SELECT uid, pid, title, sorting, t3ver_oid, t3ver_wsid, t3ver_state,
            t3ver_stage, crdate, deleted
7       FROM pages
8       WHERE deleted = 0 AND t3ver_wsid=0
9    ),
10   -- Step 1: Get all workspace pages
11   getAllWsPages AS (
12      SELECT uid, pid, title, sorting, t3ver_oid, t3ver_wsid, t3ver_state,
            t3ver_stage, crdate, deleted
13      FROM pages
14      WHERE deleted = 0 AND t3ver_wsid=@selectedWS
15   ),
16   ...
```

Code 3.1: *Erstellung der bereinigten Datengrundlage*

3.6.2 Teildaten bilden

Im zweiten Schritt werden aus der bereinigten Datengrundlage die Datensätze ausgewählt, die später im rekursiven Teil verarbeitet werden. Durch die Struktur der pages-Tabelle können die benötigten Datensätze in drei verschiedenen Ausprägungen vorkommen. Deswegen werden hierfür drei temporäre Tabellen gebildet. Für den nächsten Schritt ist es wichtig, dass alle drei Tabellen die **gleichen Felder in der gleichen Reihenfolge** besitzen.

Die temporäre Tabelle “getAllLivePages_exclWsEdit” bildet alle Live-Pages ab, für die es keine Änderung im jeweiligen Workspace gibt.

Die temporäre Tabelle “getAllWsPages_exclUnchangedLivePages” erfasst alle Pages, die im jeweiligen Workspace geändert wurden. Außerdem wird das Feld *uid* mit dem Feld *t3ver_oid* überschrieben. Dadurch wird die Page aus dem Workspace wie eine Live-Page behandelt. Das gleiche Verfahren wird auch in TYPO3 angewendet.

Die temporäre Tabelle “getAllWsPages_onlyNew” enthält alle Pages die nur im jeweiligen Workspace existieren, also im Workspace angelegt wurden.

In allen Tabellen wird ein zusätzliches Feld namens *__ORIG_UID__* angelegt, welches die ursprüngliche *uid* der Page enthält. Funktional ist dies zwar nicht notwendig, aber für künftige Anwendungen könnte es hilfreich sein, die ursprüngliche uid der Page zu kennen.

```
WITH
   ...
   -- Step 2: Get live pages excluding pages changed/new in workspace
   getAllLivePages_exclWsEdit AS (
      SELECT a.uid AS '__ORIG_UID__', a.uid, a.pid, a.title, a.sorting,
          a.t3ver_oid, a.t3ver_wsid, a.t3ver_state, a.t3ver_stage, a.crdate,
          a.deleted
      FROM getAllLivePages a
      LEFT JOIN getAllWsPages b ON a.uid = b.t3ver_oid
      WHERE b.t3ver_oid IS NULL
   ),
   -- Step 2: Get workspace overlay for changed/moved/deleted pages
   getAllWsPages_exclUnchangedLivePages AS (
      SELECT a.uid AS '__ORIG_UID__', a.t3ver_oid AS 'uid', a.pid, a.title,
          a.sorting, a.t3ver_oid, a.t3ver_wsid, a.t3ver_state, a.t3ver_stage,
          a.crdate, a.deleted
      FROM getAllWsPages a
      JOIN getAllLivePages b ON a.t3ver_oid = b.uid
      WHERE a.t3ver_oid!=0 -- 'WHERE a.t3ver_oid!=0' not really necessary but
          added for safety
   ),
   -- Step 2: Get new pages and placeholders in workspace (created in ws, but
       not present in live data)
   getAllWsPages_onlyNew AS (
      SELECT uid AS '__ORIG_UID__', uid, pid, title, sorting, t3ver_oid,
          t3ver_wsid, t3ver_state, t3ver_stage, crdate, deleted
      FROM getAllWsPages
      WHERE t3ver_oid=0 AND (t3ver_state=1 OR t3ver_state=-1)
   ),
   ...
```

Code 3.2: *Bildung der Teildaten*

3.6.3 Datentabelle erstellen

Im dritten Schritt werden alle Teildaten mit der *UNION*-Klausel zu der Datentabelle "data" zusammengefügt. Sollten die Tabellen unterschiedliche Felder enthalten, müssten die Felder an dieser Stelle einheitlich selektiert werden.

```
WITH
   ...
   -- Step 3: Merge all data parts from step 2
   data AS (
      SELECT * FROM getAllLivePages_exclWsEdit
      UNION
      SELECT * FROM getAllWsPages_exclUnchangedLivePages
      UNION
      SELECT * FROM getAllWsPages_onlyNew
   ),
   ...
```

Code 3.3: *Erstellung der Datentabelle*

3.6.4 Rekursives Statement

Das rekursive Statement wird, der Logik folgend, ebenfalls als temporäre Tabelle angelegt. Dazu wird die Tabelle “result” definiert, die wiederum die rekursiv erstellte Tabelle “rcte” enthält. Die Datentabelle für rcte ist “data”, welche zuvor in Code 3.3 erstellt wurde.

In der rekursiven Abfrage wird nun eine Startpage selektiert von der alle Childpages ermittelt werden (Startpage.uid = Childpage.pid). Danach wird von jeder Childpage, dessen Childpages ermittelt usw. Neben den Feldern aus *data* werden die Felder *__CTE_PATH__* und *__CTE_TITLE__* hinzugefügt, die technisch nicht notwendig, aber hilfreich beim Testen sind.

Das rekursive Statement erfüllt zudem zwei grundlegende Anforderungen. Es ermöglicht die korrekte Sortierung des Ergebnisses mit dem Feld *__CTE_SORTING__* und definiert Abbruchbedingungen.

```
1  SET @maxTraversalLevel = CAST(20 AS int);
2  SET @selectedPid = CAST(0 AS int);
3
4  WITH
5    ...
6    result AS (
7      WITH RECURSIVE rcte AS (
8        SELECT
9          D.uid,
10            ...
11         1 as '__CTE_LEVEL__',
12         CAST(LPAD(D.sorting,4,'0') AS CHAR(200)) as '__CTE_SORTING__',
13         CAST(D.uid AS CHAR(100)) as '__CTE_PATH__',
14         D.title as '__CTE_TITLE__'
15       FROM data D
16       WHERE D.pid = @selectedPid AND D.t3ver_state <> 2 -- IF @selectedPid=0
             THEN selector=D.pid ELSE selector=D.uid
17       UNION ALL
18       SELECT
19         D.uid,
20         ...
21         R.__CTE_LEVEL__+1,
22         CONCAT(R.__CTE_SORTING__,'/',LPAD(D.sorting,4,'0')) as
               '__CTE_SORTING__',
23         CONCAT(R.__CTE_PATH__,'/',D.uid) as '__CTE_PATH__',
24         CONCAT(REPEAT(' ', R.__CTE_LEVEL__), D.title) as '__CTE_TITLE__'
25       FROM data D
26       INNER JOIN rcte R ON R.uid = D.pid
27         WHERE D.t3ver_state <> 2 -- Escape condition for recursion (deleted
               parent page)
28         AND R.__CTE_LEVEL__ < @maxTraversalLevel -- Escape condition for
               maximum traversal level (max. parent pages)
29     )
30       SELECT * FROM rcte
31   )
32    ...
```

Code 3.4: *Rekursives SQL-Statement*

Sortierung

Das Ergebnis der rekursiven Abfrage wird in einer übergeordneten, temporären Tabelle gespeichert. Deswegen ist eine Sortierung in der rekursiven Abfrage selbst nicht sinnvoll. Stattdessen wird ein zusätzliches Feld *__CTE_SORTING__* angefügt. Dieses Feld enthält eine zusammengesetzte Zeichenkette aus den *sorting*-Feldern der übergeordneten Pages. Dieses Feld kann auch außerhalb der rekursiven Abfrage **lexikalisch** sortiert werden. Wichtig ist, dass jedes *sorting*-Feld die gleich Länge hat, z.B. 4.

Für den Seitenbaum aus Abb. 2.4 würde das Ergebnis der Sortierung wir folgt aussehen:

uid	__CTE_SORTING__	__CTE_TITLE__
1	0257	1
8	0257/1420	1.1
19	0514	2
18	0514/0064	2.1
24	0514/0096	2.2

Tab. 3.2: *Beispielausgabe für lexikalische Sortierung*

Abbruchbedingungen

Es werden zwei Abbruchbedingungen definiert. Zum einen wird ein zusätzliches Feld namens *__CTE_LEVEL__* definiert, dass pro Rekursionsebene hochgezählt wird. Im rekursiven Teil der Abfrage kann mit der Variable *@maxTraversalLevel* die maximale Rekursionstiefe festgelegt werden und so eine Endlosschleife unterbrochen werden.

Zum anderen wird mit der Bedingung *D.t3ver_state <> 2* festgelegt, dass bei gelöschten Pages im Workspace gestoppt werden soll.

3.6.5 Sortierte Ausgabe aller Pages

Um alle Pages in der richtigen Reihenfolge zu erhalten und damit den korrekten Seitenbaum darzustellen, wird die Tabelle *result* des äußersten WITH-Statement selektiert. Im letzten Schritt wird diese Tabelle lexikalisch nach dem Feld *__CTE_SORTING__* sortiert.

```
1 WITH
2   ...
3    result AS (
4      WITH RECURSIVE rcte AS (
5        ...
6      )
7       SELECT * FROM rcte
8   )
9 select * from result ORDER BY __CTE_SORTING__ ASC;
```

Code 3.5: *Sortierung der recursive CTE*

3.7 TYPO3 Extension

Um die recursive CTE aus Kapitel 3.6 in TYPO3 zu nutzen, wird eine Extension erstellt, welche das Statement als Service umsetzt. Das Ziel ist es, die Umsetzbarkeit von recursive CTE in TYPO3

aufzuzeigen und die Performanceunterschiede zu ausgewählten Core Methoden zu untersuchen. Die Extension wird keine grafischen Elemente im Front- oder Backend besitzen und ist nur über einen Service, mittels Terminal ausführbar. Der Service kann in Folgeprojekten als Grundlage für die Integration in den TYPO3 Core dienen. Die Umsetzung erfolgt mit Hilfe von Doctrine DBAL, um unabhängig vom verwendeten DBMS zu sein.

3.7.1 Grundgerüst

Das Grundgerüst der Extension bildet die *"TYPO3 Basics - testing-framework integration demo extension"* [16] von Stefan Bürk. Diese Extension dient eigentlich als Beispiel, um Unit- und Functional-Tests in Extensions einzubinden. Sie lässt sich aber gut nutzen, um einen TYPO3 Service zu konfigurieren.

3.7.2 Aufbau und Funktionsweise

Der Service besteht aus einer Klasse und ähnelt vom Aufbau stark dem SQL-Statement.

Mit der Methode *getAllWsPages()* wird die **bereinigte Datengrundlage** erstellt. Die jeweilige Workspace uid wird als Parameter übergeben, sodass alle Pages des Workspace und alle Live-Pages (Workspace 0) ermittelt werden.

Um die **Teildaten** zu ermitteln, werden die Methoden *getAllLivePages_exclWsEdit()*, *getAllWsPages_onlyNew()* und *getAllWsPages_exclUnchangedLivePages()* genutzt. Sie erfassen die gleichen Datensätze wie die temporären Tabellen aus Schritt zwei des SQL-Statements. Die Methoden greifen auf die Datengrundlage von *getAllWsPages()* zu.

Mit *getAllPages()* werden die Teildaten zur **Datentabelle** zusammengefasst.

Die Methode *buildRcte()* erstellt letztendlich das Statement, welches den Seitenbaum zurück gibt. Als Parameter wird die jeweilige Workspace ID übergeben. Von buildRcte() aus wird wiederum auf getAllPages() zugegriffen um die Datentabelle zu erhalten.

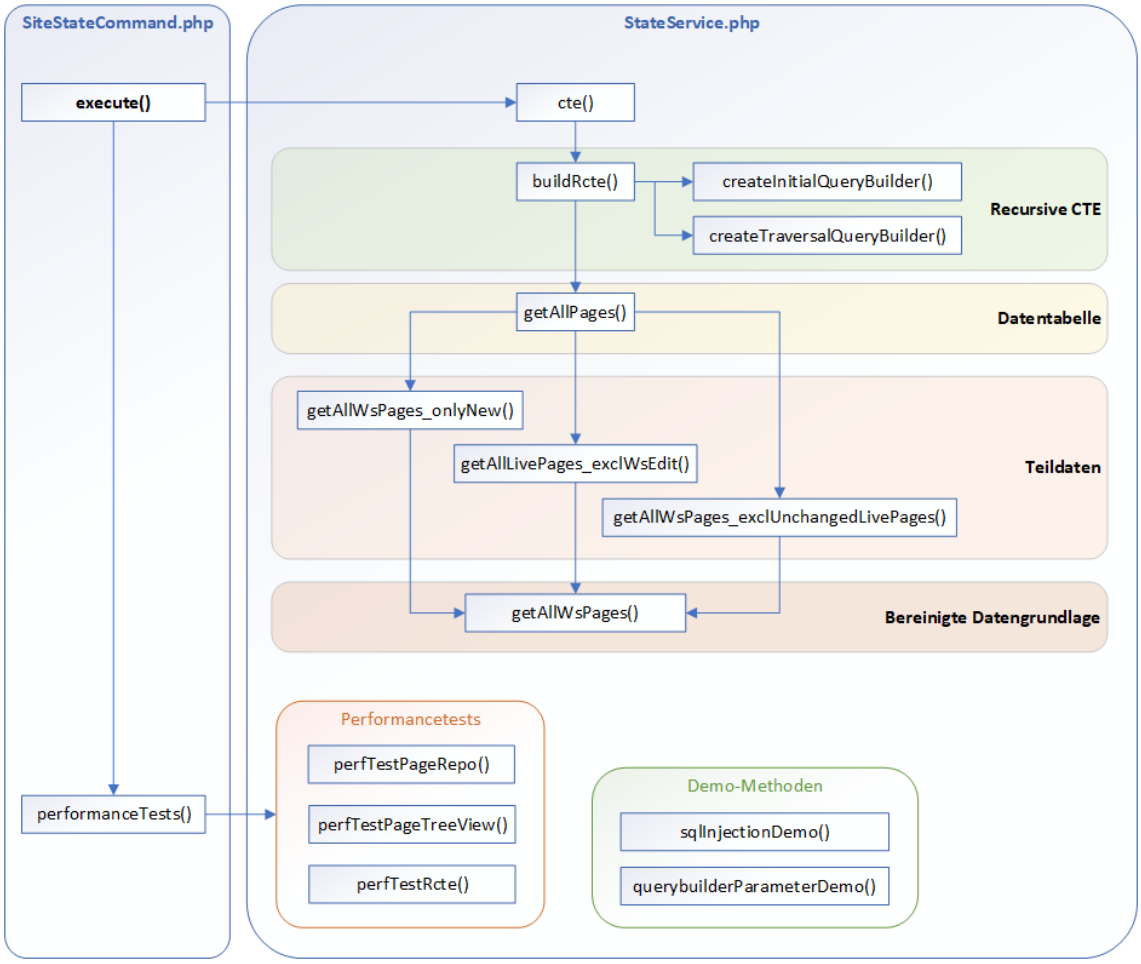


Abb. 3.2: *Aufbau der Extension*

3.7.3 Besonderheiten

Ein Ziel bei der Erstellung des Services war es, diesen möglichst universell zu halten. In TYPO3 sind bereits mehrere Methoden vorhanden, die einen Seitenbaum erstellen. Diese Methoden liefern aber unterschiedliche Rückgabewerte und sind nicht untereinander austauschbar.

Selektive Feldauswahl

Der erstellte Service bietet die Möglichkeit den Seitenbaum für einen Workspace, mit beliebigen Feldern der pages-Tabelle zu erhalten. Die Feldnamen müssen dafür als Parameter übergeben werden. Das Besondere daran ist, dass die Selektion der Felder nicht in der letzten SELECT-Klausel des SQL-Statements vorgenommen wird, sondern bereits bei der Erstellung der einzelnen temporären Tabellen. Dadurch wird verhindert, dass Querybuilder Instanzen mehr Informationen beinhalten als notwendig.

Dafür sind in jeder Methode Tabellenfelder definiert, die für die Funktionsweise zwingend nötig sind, die *"mandatory elds"*. Diese Felder sind für den Anwender transparent, sodass dieser kein Wissen über die notwendigen Felder der Methode benötigt. Aus den *mandatory elds* und den Übergabeparametern wird anschließend die **Datengrundlage der Funktion** erstellt. Diese Datengrundlage ist eine Tabelle, von der wiederum nur die Felder aus den Übergabeparametern als Ergebnis selektiert werden.

Als Beispiel dient eine vereinfachte Version von *getAllWsPages_onlyNew*:

```php
$output = getAllWsPages_onlyNew(['title'], 2);

protected function getAllWsPages_onlyNew(array $outputFields, int $workspace)
{
    $mandatoryDataFields = ['t3ver_state', 't3ver_oid'];
    $dataFields = array_merge($outputFields, $mandatoryDataFields);

    $subSelect_ws_all = $this->getAllWsPages($dataFields, $workspace);

    $query = $queryBuilder
        ->typo3_with(
            'data_ws_all',
            $subSelect_ws_all,
            $dataFields,
        )
        ->select(...$outputFields)
        ->from('data_ws_all')
        ->where('t3ver_oid=0')
        ->andWhere("t3ver_state=1 OR t3ver_state = -1");

    return $query;
}
```

Code 3.6: *Vereinfachte Version von getAllWsPages_onlyNew()*

Zeile	Erläuterung
1	Es soll nur der Titel aller neuen Pages, des Workspaces 2 ermittelt werden.
5	Für die eigentliche Funktion werden die Felder *t3ver_state* und *t3ver_oid* benötigt.
6	Die Datengrundlage besteht aus den mandatory fields und den übergebenen Feldern.
8	Die Datengrundlage der Methode wird erstellt.
16	Das Statement selektiert nur die übergebenen Felder ('title').
18+19	Für die Bedingung müssen die *mandatory elds* in der Datengrundlage vorhanden sein.

Tab. 3.3: *Erläuterung von Code 3.6*

Meta□elds

Neben *mandatory elds* beinhalten die Methoden auch *meta elds*. Dies sind Felder die kein Bestandteil der pages Tabelle sind, aber in dem jeweiligen Statement erstellt werden. Darunter fallen z.B. die in der rekursiven Abfrage zwingend notwendigen Felder *__CTE_LEVEL__* und *__CTE_SORTING__*. Die Felder werden standardmäßig nicht ausgegeben, können aber beim Methodenaufruf mit dem Parameter *metaFields* zum ausgebenden SELECT-Statement hinzugefügt werden. Im Moment werden auch optionale *meta elds* in den Methoden erstellt. Dies ist wie bei dem reinen SQL-Statement hilfreich fürs Debuggen, könnte in Zukunft aber aus dem Service entfernt werden.

Als Beispiel dient wieder die vereinfachte Version von *getAllWsPages_onlyNew*:

```php
$output = getAllWsPages_onlyNew(['title'], 2, true);

protected function getAllWsPages_onlyNew(array $outputFields, int $workspace,
    bool $metaFields=false)
{
```

```
5       $mandatoryDataFields = ['t3ver_state', 't3ver_oid'];
6
7       if($metaFields) {
8           $outputFields[] = 'uid as __ORIG_UID__';
9           $mandatoryDataFields[] = 'uid';
10      }
11
12      $dataFields = array_merge($outputFields, $mandatoryDataFields);
13
14      $subSelect_ws_all = $this->getAllWsPages($dataFields, $workspace);
15
16      $query = $queryBuilder
17          ->typo3_with(
18              'data_ws_all',
19              $subSelect_ws_all,
20              $dataFields,
21          )
22          ->select(...$outputFields)
23          ->from('data_ws_all')
24          ->where('t3ver_oid=0')
25          ->andWhere("t3ver_state=1 OR t3ver_state = -1");
26
27      return $query;
28  }
```

Code 3.7: *Vereinfachte Version von getAllWsPages_onlyNew()*

Zeile	Erläuterung
1	Aufruf der Methode mit *true* als Wert für *metaFields*.
5	Für die eigentliche Funktion werden die Felder *t3ver_state* und *t3ver_oid* benötigt.
12	Die Datengrundlage besteht aus den *mandatory fields* und den übergebenen Feldern.
14	Die Datengrundlage der Methode wird erstellt.
22	Das Statement selektiert nur die übergebenen Felder ('title').
24+25	Für die Bedingung müssen die *mandatory fields* in der Datengrundlage vorhanden sein.

Tab. 3.4: *Erläuterung von Code 3.7*

3.7.4 Einschränkungen des Querybuilders

Bei der Arbeit mit dem Querybuilder, von Doctrine DBAL, müssen einige Einschränkungen für diese Arbeit beachtet werden.

WITH [RECURSIVE]

Eine große Einschränkung für diese Arbeit ist, dass der Querybuilder von Doctrine DBAL keine Unterstützung für die WITH bzw. WITH RECURSIVE-Klausel bietet. In der Version 13 wurde von TYPO3 eine eigene Umsetzung des Statements in den Querybuilder integriert. Diese Integration ist kein Bestandteil von Doctrine und im Moment nicht dokumentiert.

Um das Statement zu nutzen wurde mit den TYPO3 Core Entwicklern Rücksprache gehalten und der Sourcecode des "RootlineUtility" untersucht. Dort befindet sich die bisher einzige Umsetzung des Statements mittels Querybuilder. Sie wurde als Inspiration für diese Arbeit genutzt.

Es ist davon auszugehen, dass Doctrine künftig die WITH [RECURSIVE]-Klausel offiziell unterstützen wird und die TYPO3 eigene Integration ablösen wird.

UNION

Die UNION-Funktion des Querybuilders gibt ein SQL-Statement zurück, welches die einzelnen SELECT-Statements mit Klammern umschließt. Für einfache SELECT-Anweisungen stellen die zusätzlichen Klammern kein Problem dar. Für benannte SELECT-Statements, wie WITH [RECURSIVE] führen diese Klammern zu einer ungültigen SQL Syntax, da die WITH-Klausel zwingend am Anfang des (Sub) Selects stehen muss.

Das folgende Querybuilder Objekt generiert das nachfolgende Statement:

```php
$output = $connection->createQueryBuilder()
    ->union("SATEMENT")
    ->addUnion("SATEMENT",UnionType::ALL);
```

Code 3.8: *Verwendung der UNION-Klausel mit dem Querybuilder*

```
(SATEMENT) UNION ALL (SATEMENT);
```

Code 3.9: *SQL-Statement aus Code 3.8*

In allen Methoden werden temporäre Tabellen mit der WITH-Klausel erstellt. Diese Tabellen müssen in der Methode *getAllPages()*, mittels UNION-Klausel, zu einer *Datentabelle* zusammengefasst werden. Zu diesem Zweck wird im UNION-Teil ein *"SELECT * FROM Teildaten"* ausgeführt. Diese Lösung ist etwas unschön, beinhaltet aber keine überflüssigen Tabellenfelder, da in den Teildaten nur die nötigsten Felder selektiert wurden.

```php
$query = $queryBuilder
    ->typo3_with(
        'allLivePages_exclWsEdit',
        $allLivePages_exclWsEdit,
        $outputFields,
    )
    ->typo3_addwith(
        'allWsPages_exclUnchangedLivePages',
        $allWsPages_exclUnchangedLivePages,
        $outputFields,
    )
    ->typo3_addwith(
        'allWsPages_onlyNew',
        $allWsPages_onlyNew,
        $outputFields,
    )
    ->union("select * FROM allLivePages_exclWsEdit")
    ->addUnion("select * FROM allWsPages_exclUnchangedLivePages", UnionType::ALL)
    ->addUnion("select * FROM allWsPages_onlyNew", UnionType::ALL);
```

Code 3.10: *Erstellung der Datentabelle*

Parameterübergabe

Um die Gefahr von SQL-Injections zu minimieren, verwendet die Extension prepared Statements. Dafür werden die einzelnen SQL-Statements als Querybuilder-Objekte formuliert. Bei

einzelnen, abgeschlossenen Statements entstehen dabei keine Probleme. Wenn aber, wie in der Extension, Querybuilder-Objekte ineinander verschachtelt werden, müssen **zwei große Probleme** gelöst werden. Zum einen können Querybuilder-Objekte, bei der Ausführung, nicht auf Parameter anderer Querybuilder-Objekte zugreifen. Zum anderen werden die Parameter in jedem Querybuilder-Objekt nach dem gleichen Namensschema benannt. Dadurch können zwei Querybuilder-Objekte den gleichen Parameternamen für unterschiedliche Werte besitzen. Nach der Zusammenfassung von solchen Objekten kann nicht mehr bestimmt werden, welcher Wert zu welchem Parameter gehört.

In Code 3.11 wird dieses Problem demonstriert.

```
// Different parameters for $queryBuilder_A and $queryBuilder_B
$param_A = 1;
$param_B = 2;

// Create $queryBuilder_A
$queryBuilder_A
   ->select('uid')
   ->from('pages')
   ->where(
      $queryBuilder_A->expr()->eq(
         'uid',
         $queryBuilder_A->createNamedParameter($param_A, Connection::PARAM_INT)
      )
);

// Create $queryBuilder_B
$queryBuilder_B
   ->select('uid')
   ->from('pages')
   ->where(
      $queryBuilder_B->expr()->eq(
         'uid',
         $queryBuilder_B->createNamedParameter($param_B, Connection::PARAM_INT)
      )
);

/*
Problem with same parameter name (:dcValue1) of generated statements
Same SQL-Statement for A and B:
Statement_A:
  SELECT `uid` FROM `pages` WHERE (`uid` = :dcValue1) AND (`pages`.`deleted` = 0)
Statement_B
  SELECT `uid` FROM `pages` WHERE (`uid` = :dcValue1) AND (`pages`.`deleted` = 0)
*/
```

Code 3.11: *Gleiche Parameternamen für Querybuilderobjekte*

Zeile	Erläuterung
2+3	Variablen, die in den SQL-Statements prepared werden müssen.
6+17	Die Querybuilder-Objekte werden erstellt.
12	Variable param_A wird **für queryBuilder_A** prepared.
23	Variable param_B wird **für queryBuilder_B** prepared.
31+33	Die Erstellten SQL-Statements nutzen den selben Parameternamen. Beim Zusammenfügen der Querybuilder-Objekte wird der Parameter *:dcValue1* in jedem Fall einen falschen Wert für eines der Statement annehmen (1 oder 2).

Tab. 3.5: *Erläuterung von Code 3.11*

Um dieses Problem zu lösen werden die Parameter in der Extension in einem separaten Querybuilder prepared/registriert. Während der Erstellung werden die Parameter von den Statements getrennt. Das bedeutet, dass in dem finalen SQL-String des Querybuilders zwar Parameternamen vorhanden sind, aber der Querybuilder selbst dessen Werte nicht kennt. So wird das Problem der gleichen Parameternamen gelöst. Am Anfang wird ein "Paramater-Querybuilder" erstellt, der bei jedem Methodenaufruf übergeben wird. Wenn innerhalb der Methode ein prepared Statement erstellt wird, werden dessen Parameter nicht in dem erstellten Statement registriert, sondern in dem "Paramater-Querybuilder". Doctrine DBAL erkennt, dass dieser Querybuilder bereits Parameter besitzt und fügt neue Parameter am Ende hinzu. Gleichzeitig wird dieser neue, hochgezählte Parameter in dem jeweiligen Statement hinterlegt.

Das Beispiel aus Code 3.11 würde damit wie folgt abgewandelt werden.

```
1  // Different parameters for $queryBuilder_A and $queryBuilder_B
2  $param_A = 1;
3  $param_B = 2;
4
5  // Create $queryBuilder_A
6  $queryBuilder_A
7     ->select('uid')
8     ->from('pages')
9     ->where(
10       $queryBuilder_A->expr()->eq(
11          'uid',
12          // Note the qb is different from above
13          $queryBuilder_param->createNamedParameter($param_A, Connection::
                PARAM_INT)
14       )
15 );
16
17 // Create $queryBuilder_B
18 $queryBuilder_B
19    ->select('uid')
20    ->from('pages')
21    ->where(
22       $queryBuilder_B->expr()->eq(
23          'uid',
24          // Note the qb is defferent from above
25          $queryBuilder_param->createNamedParameter($param_B, Connection::
                PARAM_INT))
26 );
27
28 $queryBuilder_New->setParameters($queryBuilder_param->getParameters());
29 $queryBuilder_New
30    ->union($queryBuilder_A)
```

```
31      ->addunion($queryBuilder_B)
32  ;
33
34
35  /*
36  Solves the problem 2 querybuilders having the same parameters
37  Statement_New:
38  (SELECT ‘uid‘ FROM ‘pages‘ WHERE (‘uid‘ = :dcValue1) AND (‘pages‘.‘deleted‘ = 0))
39  UNION
40  (SELECT ‘uid‘ FROM ‘pages‘ WHERE (‘uid‘ = :dcValue2) AND (‘pages‘.‘deleted‘ = 0))
41  */
```

Code 3.12: *Unterschiedliche Parameternamen für Querybuilderobjekte*

Zeile	Erläuterung
13	Variable param_A wird in queryBuilder_param prepared.
25	Variable param_B wird in queryBuilder_param prepared.
28	Parameterübergabe an **übergeordnetem Querybuilder** *$queryBuilder_New*.
29-32	Statement bestehend aus 2 Querybuilder-Objekten.
38-40	Parameternamen werden im SQL-Statement sauber unterschieden.

Tab. 3.6: *Erläuterung von Code 3.12*

In der Extension wird das selbe “Querybuilder-Objekt” an alle Methoden übergeben, sodass dieses Objekt am Ende alle Parameter (wie im Beispiel) enthält. Dadurch können im finalen SQL-Statement alle Parameternamen einem eindeutigen Wert zugeordnet werden. Bevor das final generierte SQL-Statement ausgeführt werden kann, werden die Parameter aus dem “Paramater-Querybuilder” mit der Methode *setParameters()* in den “Statement-Querybuilder” übertragen. Dies wird wie in Zeile 28 im Beispiel vorgenommen.

Kapitel 4

Analyse

Zur Überprüfung der Hypothese, dass die Performance einer TYPO3 Instanz mit recursive CTE gesteigert werden kann, werden Methoden aus dem TYPO3 Core mit dem Service der erstellten Extension verglichen. Die Methoden aus dem TYPO3 Core nutzen keine recursive CTE's, im Vergleich zum Service der Extension. Für die Tests wird eine separate TYPO3 13.4.0 Instanz genutzt. Diese wurde ebenfalls per Docker und DDEV installiert und nutzt ebenfalls MariaDB als DBMS. Die Testdaten wurden mittels *ddev import-db* importiert. Die erstellte Extension aus der Entwicklungsinstanz wird ohne Änderungen in die Testinstanz übernommen.

4.1 Analysekriterien

Es muss zunächst eine Vergleichbarkeit zwischen der erstellten Extension und den TYPO3 Core Methoden geschaffen werden. Dazu werden gemeinsame Messgrößen festgelegt, mit denen die Performance definiert wird.

4.1.1 Ausführungszeit

Das Hauptkriterium um die Performance zu beschreiben ist die Ausführungszeit. Hier werden Anwender größere Unterschiede am stärksten wahrnehmen. Es wird gemessen, wie lange es dauert um den Seitenbaum als Array zu erhalten. Gemessen wird dabei nur vom Aufruf der eigentlichen Methode bis zum Erhalt des Arrays. Formatierungen oder Ausgaben zum Debuggen werden von der Zeitmessung ausgenommen.

4.1.2 Anzahl der SQL-Statements

Die Anzahl der SQL-Statements wird ebenfalls als Messgröße hinzugezogen. Bei den Tests wird die TYPO3 Instanz auf dem selben System betrieben wie die Datenbank. Sollte die TYPO3 Datenbank aber auf einem separaten Server betrieben werden, könnten viele SQL-Statements über das Netzwerk zusätzliche Wartezeiten verursachen.

Anzahl der Datenbankverbindungen

Es wird vermutet, dass das Erstellen und Beenden von Datenbankverbindungen relativ viel Zeit benötigt. Deswegen wird die Anzahl der Datenbankverbindungen (Sessions) ebenfalls als Messgröße hinzugezogen. Eine hohe Anzahl von Datenbankverbindungen würde auf eine schlechte Nutzung des Querybuilders hinweisen und müsste genauer untersucht werden.

4.2 Messverfahren

Die Messungen umfassen zwei Hauptbereiche, die Zeitmessung der TYPO3 Methoden und die Messung der SQL-Statements.

4.2.1 Zeitmessung

Für die Zeitmessung werden drei Methoden in der Extension verwendet, **perf TestPageRepo, perf TestPageTreeView und perf TestRcte**. Diese Methoden werden nur für die Performancetests genutzt und können für eine produktive Umsetzung entfernt werden.

Jede der drei Methoden erstellt ein Array, dass den Seitenbaum abbildet. Für die Erstellung werden verschiedene Methoden aus dem TYPO3 Core oder der Extension genutzt. Zum Debuggen kann den Methoden ein Parameter zum Ausgeben des Seitenbaums übergeben werden. Die jeweilige Ausführungszeit wird als Rückgabewert verwendet.

Die Zeitmessung wird mit der PHP Funktion *hrtime()* vorgenommen. Die Funktion hrtime() gibt die aktuelle "high resolution time" des Systems in Nanosekunden wieder und ist in PHP für Zeitmessungen vorgesehen. Neben hrtime() bietet PHP mit *microtime()* eine ähnliche Funktionalität zur Zeitmessung an. Mit microtime() erhält man den aktuellen UNIX-Zeitstempel, der nicht für präzise Zeitmessungen verwendet werden sollte.

Die Zeitmessung erfolgt in den drei Testmethoden jeweils direkt vor und nach der Erstellung des Seitenbaum-Arrays. Die Differenz aus beiden Messungen ergibt die Ausführungszeit. Die Erstellung der Klassenobjekte, die Differenzbildung der Zeitmessungen und evtl. Ausgaben werden nicht in die Messungen einbezogen.

Als Beispiel dient die Methode *perf TestPageRepo()*:

```php
public function perfTestPageRepo(array $parentIds, bool $output=false): int
{
    $pageRepo = \TYPO3\CMS\Core\Utility\GeneralUtility::makeInstance(
        PageRepository::class);

    $pagerepoStart = hrtime(TRUE);
        $pageRepoOutput = $pageRepo->getPageIdsRecursive($parentIds, self::
            MAX_CTE_TRAVERSAL_LEVELS);
    $pagerepoStop = hrtime(TRUE);

    $time = $pagerepoStop - $pagerepoStart;

    if ($output) {
        echo "\npageRepo:\n";
        foreach ($pageRepoOutput as $page) {
            echo $page . "\n";
        }

        echo "PageRepo: " . ($time) / 1000000 . "\n";      // Milliseconds
    }

    return $time;
}
```

Code 4.1: *Performancetest mittels perf TestPageRepo()*

Zeile	Erläuterung
3	Erstellung des PageRepository-Instanz.
5	Start der Zeitmessung.
6	Erstellung des Seitenbaum-Arrays.
7	Ende der Zeitmessung.
9	Differenz der Zeitstempel bilden (Ausführungszeit).
11+18	Output zum Debuggen.
20	Rückgabe der Ausführungszeit.

Tab. 4.1: *Erläuterung von Code 4.1*

Um die drei Performancetest-Methoden aufzurufen, wird die Wrapper-Methode *performanceTests()* in SiteStateCommand.php genutzt. Über diese Methode wird gesteuert, welche Performancetests ausgeführt werden und mit welchen Parametern, z.B. mit welchen Page-uid's. Außerdem wird in der Wrapper-Methode die Ausführungszeit auf der Konsole ausgegeben.
Beim Aufruf von performanceTests() kann zudem angegeben werden, wie oft die zu testenden Methoden ausgeführt werden. Mit Mehrfachmessungen lassen sich Schwankungen bei der Ausführungszeit besser ausgleichen um ein konsistentes Testergebnis zu erhalten.

Zur Verdeutlichung die vereinfachte Form der Wrapper-Methode *performanceTests()*:

```
1  protected function performanceTests(int $times=1, bool $pageRepo=true): void
2  {
3      ...
4      if ($pageRepo) {
5          for ($i=1; $i<=$times; $i++) {
6              echo "PageRepo ($i): " . ($this->stateService->perfTestPageRepo(
                   $parentIds, $output)) / 1000000 . "\n";      // Milliseconds
7          }
8      }
9      ...
10 }
```

Code 4.2: *Vereinfachte Version von performanceTests()*

Zeile	Erläuterung
1	Parameter $times für die Anzahl der Ausführungen.
6	Aufruf der Performancetest-Methode *perfTestPageRepo()* und Ausgabe der Ausführungszeit.

Tab. 4.2: *Erläuterung von Code 4.2*

Um zwischen der Ausführung des eigentlichen Services und den Performancetests zu wechseln müssen die Aufrufe in der execute-Methode in SiteStateCommand.php ein- bzw. auskommentiert werden.

```
1  protected function execute(InputInterface $input, OutputInterface $output): int
2  {
3      // Make sure the _cli_ user is loaded
4      Bootstrap::initializeBackendAuthentication();
5      $io = new SymfonyStyle($input, $output);
6
7      // Production part
8      $this->stateService->cte();
9
```

```
10      // Performance testing part
11      $times=3;
12      $pageRepo=true;
13      $pageTreeView=true;
14      $rCte=true;
15
16      $this->performanceTests($times, $pageRepo, $pageTreeView, $rCte);
17
18      return Command::SUCCESS;
19  }
```

Code 4.3: *Einstiegsmethode der Extension*

Zeile	Erläuterung
1	Hauptmethode, die vom Service ausgeführt wird.
8	Aufruf der funktionalen Umsetzung des Services.
11-14	Parameter für den Performancetest werden initialisiert.
16	Aufruf der Performancetest.
18	Rückgabewert um den Aufruf des Services ordentlich zu beenden.

Tab. 4.3: *Erläuterung von Code 4.3*

4.2.2 Anzahl SQL-Statements

Um zu Bestimmen welche SQL-Statements vom DBMS empfangen und verarbeitet werden, werden diese über das Logging der Datenbank ermittelt. Dadurch wird sichergestellt, dass alle Statements inkl. Verbindungsaufbau und Verbindungsabbau erfasst werden.

Die Logs werden über MySQL Workbench aktiviert und in die Tabelle *mysql.general_log* geschrieben. Alle anderen Einstellungen der Datenbank bleiben unberührt.

```
1  SET GLOBAL general_log = 1;
2  SET GLOBAL log_output = 'Table';
```

Code 4.4: *Konfiguration des Datenbankloggings*

Um für eine Methode die Anzahl der ausgelösten SQL-Statements zu ermitteln, wird als Grundvoraussetzung die ungefähre **Ausführungszeit** und der **Start- oder Endzeitpunkt** der Ausführung benötigt.

Die Ausführungszeit wird ermittelt, indem die zu testende Methode über die Wrapper-Methode *performanceTests()* ausgeführt wird. Die Ausführungszeit wird anschließend der Ausgabe auf der Konsole entnommen. Im Beispiel, Abb. 4.1, beträgt die Ausführungszeit von *PageRepository->getPageIdsRecursive()* etwa 38,4 Sekunden.

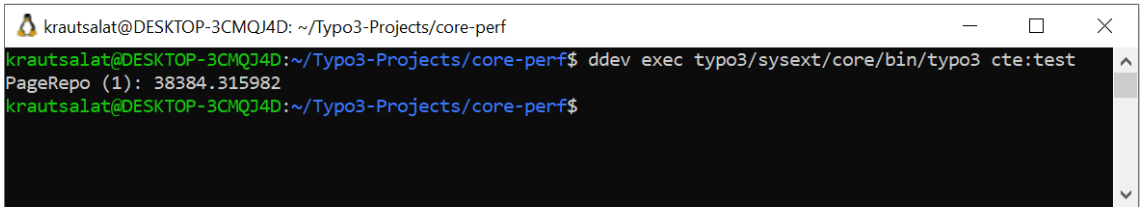


Abb. 4.1: *Beispielausgabe für performanceTests()*

Der Start- bzw. Endzeitpunkt der Ausführung wird als Variable in MySQL Workbench kurz vor oder nach der Ausführung von performanceTests() deklariert. Der Zeitstempel wird in der Variable "@timestmp" gespeichert, indem folgendes Statement manuell ausgeführt wird.

```
1 SET @timestmp = CURRENT_TIMESTAMP;
```

Code 4.5: *SQL-Statement zum Festlegen des Messzeitpunktes*

Der Zeitstempel wird als Variable nur in der jeweiligen Datenbank Session gespeichert. Sobald die Session beendet wird geht der Zeitstempel verloren. Dadurch, dass die TYPO3 Methoden für den selben Seitenbaum immer die gleichen SQL-Statements erzeugen, ist eine dauerhafte Speicherung für diese Arbeit nicht notwendig. Andernfalls kann der Wert von "@timestmp" auch ausgegeben und separat gespeichert werden.

```
1 SELECT @timestmp;
```

Code 4.6: *SQL-Statement zum Ausgeben des Messzeitpunktes*

Zum Auswerten der Log-Tabelle wird diese mit der WITH-Klausel in mehrere temporäre Tabellen unterteilt. Der Aufbau gleicht dabei dem Schema aus dem Kapitel 3.6. Auch hier wird zuerst eine **bereinigte Datengrundlage** erstellt, aus der **Teildaten gebildet** werden, um diese zu einer **Datentabelle** zusammenzufügen.

```
1  WITH
2    -- Datengrundlage
3    data_time AS (
4      SELECT * FROM mysql.general_log WHERE `event_time` > SUBTIME(@timestmp,
           "0:1:0") AND `event_time` < SUBTIME(@timestmp, "-0:1:0")
5    ),
6     -- Teildaten
7     data_prepare AS (
8      SELECT * FROM data_time WHERE `command_type` = 'Prepare'
9    ),
10    data_execute AS (
11     SELECT * FROM data_time WHERE `command_type` = 'Execute'
12   ),
13    data_close AS (
14     SELECT * FROM data_time WHERE `command_type` = 'Close stmt'
15   )
16    data_query AS (
17     SELECT * FROM data_time WHERE `command_type` = 'Query'
18   ),
19    data_quit AS (
20     SELECT * FROM data_time WHERE `command_type` = 'Quit'
21   ),
22    data_connect AS (
23     SELECT * FROM data_time WHERE `command_type` = 'Connect'
24   ),
25    -- Datentabelle
26    statistics AS (
27     SELECT
28       (SELECT SUBTIME(@timestmp, "0:1:0")) AS Timerange_Start,
29       (SELECT SUBTIME(@timestmp, "-0:1:0")) AS Timerange_End,
30       (SELECT COUNT(*) FROM data_connect) AS Connect_Count,
31       (SELECT COUNT(*) FROM data_quit) AS Quit_Count,
32       (SELECT COUNT(*) FROM data_prepare) AS Prepare_Count,
33       (SELECT COUNT(*) FROM data_execute) AS Execute_Count,
```

```
34          (SELECT COUNT(*) FROM data_close) AS Close_Count,
35          (SELECT COUNT(*) FROM data_query) AS Query_Count
36      )
37  -- Ausgabe der Daten
38  SELECT * FROM statistics;
```

Code 4.7: *Statement zum Auswerten der SQL-Logs*

Die Datengrundlage bildet die temporäre Tabelle *data_time*, welche alle Logeinträge zum Zeitpunkt von *@timestmp* +- der Ausführungszeit erfasst (1 Minute im Code 4.7). Aus der Datengrundlage werden die Teildaten nach *command_type* getrennt, z.B. die Tabelle *data_query*. In der Datentabelle *statistics* werden die Datensätze in den Teildaten gezählt und gemeinsam mit dem Zeitstempel ausgegeben. Die Datentabelle wird ebenfalls als Ausgabetabelle der WITH-Klausel verwendet.

Timerange_Start	Timerange_End	Connect_Count	Quit_Count	Prepare_Count	Execute_Count	Close_Count	Query_Count
2024-12-01 23:41:28	2024-12-01 23:43:28	3	3	39	39	39	9

Tab. 4.4: *Beispielausgabe für Code 4.7*

4.3 Testdaten

Für die Messungen wird ein anonymisierter Datensatz einer produktiven TYPO3 Instanz verwendet. Er nutzt keine Workspaces und beinhaltet nur die pages-Tabelle. Alle weiteren Tabellen der TYPO3 Instanzt bleiben unverändert.

Der Datensatz umfasst 31.880 Einträge und repräsentiert damit eine sehr große TYPO3 Instanz. Instanzen von vergleichbarer Größe stellen keine Ausnahmen dar und sind auch in der Praxis, z.B. beim BMUV (Abb. 4.2) zu finden.

Um die Größenordnung von TYPO3 Instanzen einschätzen zu können, soll die Tabelle 4.5 als Beispiel dienen. Sie verdeutlicht, dass die Anzahl der Seiten stark von der Anzahl der Ebenen im Seitenbaum abhängt.

Seiten / Instanzgröße	Gesamt	Ebene 0	Ebene 1	Ebene 2	Ebene 3	Ebene 4
Groß	**30.000**	5	10	10	10	6
Mittel	**2.500**	5	10	10	5	
Klein	**125**	5	5	5		

Tab. 4.5: *Einordnung von großen TYPO3 Instanzen*

Mit dieser Betrachtungsweise kann die Webseite des Bundesministeriums für Umwelt und Verbraucherschutz als große TYPO3 Instanz angesehen werden. Die Seitennavigation in Abbildung 4.2 zeigt zudem nur Seiten, die im Frontend, für jede Person zugänglich sind. Deaktivierte Seiten oder Seitentypen die zur Strukturierung dienen, z.B. Trennlinien und Ordner werden ebenso wenig aufgeführt wie evtl. zugriffsgeschützte Seiten.

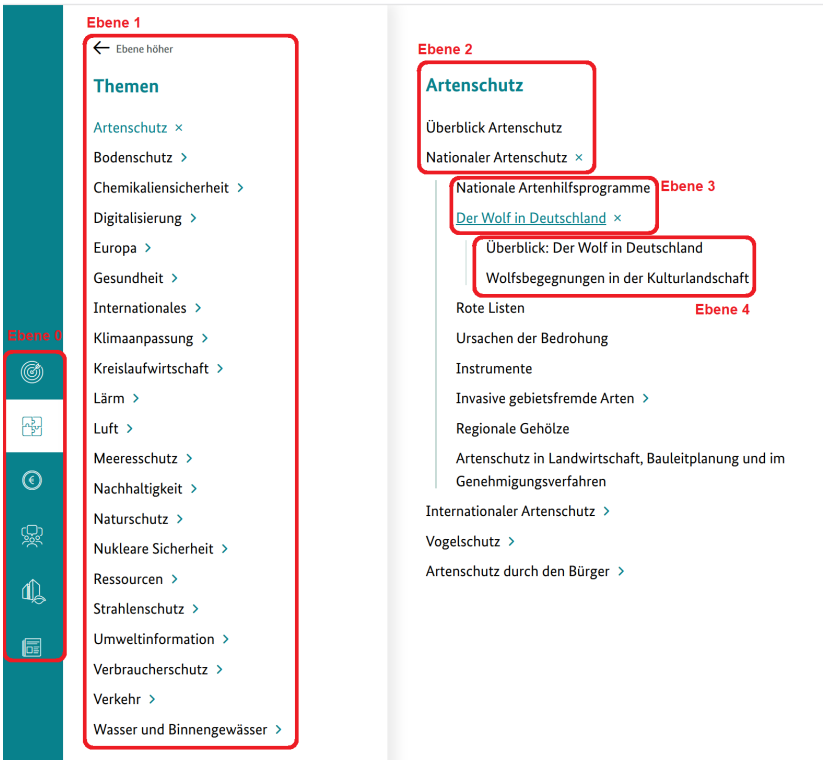


Abb. 4.2: *Seitennavigation des BMUV [17]*

4.4 Messergebnisse

Für die Performancemessungen werden die Methoden *PageTreeView->getTree()* und *PageRepository->getPageIdsRecursive()* aus dem TYPO3 Core und die Methode *buildRcte()* aus dem Service der erstellten Extension verglichen. Die Messungen werden in einer separaten TYPO3 Instanz vorgenommen, deren Datenbank mit Datensätzen aus einer produktiven TYPO3 Instanz gefüllt wurde. Der Seitenbaum umfasst 31.880 Pages und wird für die Messungen komplett durchlaufen, um das entsprechende Array zu erhalten. Als DBMS wird MariaDB verwendet.

4.4.1 Ausführungszeit

Um die Ausführungszeiten zu vergleichen, wird für jede Methode eine Messreihe aus zehn Messungen erstellt. Damit soll sichergestellt werden, dass einzelne Extremwerte das Endergebnis möglichst wenig beeinflussen. Das Logging der Datenbank wird ebenfalls deaktiviert und die Log-Tabelle geleert um die Ausführungszeiten nicht durch das Loggen zu verfälschen. Die Messungen für die Methoden *PageTreeView->getTree()*, *PageRepository->getPageIdsRecursive()* und *buildRcte()* werden zudem im Round-Robin-Verfahren ausgeführt. Die Methoden werden also im Wechsel ausgeführt und nicht zehn mal direkt hintereinander. Dies soll verhindern, dass eine Testreihe übermäßig stark von Performanceschwankungen des Testsystems betroffen ist. Zu erwähnen ist, dass die Methode *buildRcte()* nur das Ergebnis aus der Datenbankabfrage enthält. Für den produktiven Einsatz kann es nötig sein, alle Arrayeinträge durchlaufen zu müssen, um weitere TYPO3 Methoden darauf anzuwenden. In der Testumgebung dauert ein solcher Durchlauf für 30.276 Einträge nur ca. 14ms und wird deshalb in den Messergebnissen nicht berücksichtigt.

Die einzelnen Messergebnisse sind in der Tabelle 4.6 und als Graph in Abb. 4.3 dargestellt. Zu beachten ist dabei, dass die Tabellenwerte in Millisekunden und die Werte im Graph in Sekunden aufgeführt sind. Die Ausführungszeiten der TYPO3 Core Methoden weichen deutlich von

denen der Extension-Methode ab. Die Methoden *PageTreeView->getTree()* und *PageRepository->getPageIdsRecursive()* benötigen für die Erstellung des Seitenbaum Arrays ca. 55 Sekunden. Die Erstellung mittels recursive CTE benötigt ca. 0,9 Sekunden. Bevor die Messergebnisse bewertet werden können, muss sichergestellt werden, dass diese frei von Messfehlern sind. Es kann aber bereits erahnt werden, dass die Erstellung des Seitenbaum-Arrays deutlich schneller mit recursive CTE, als ohne vorgenommen wird.

Ausführung / Methode	1	2	3	4	5	6	7	8	9	10
PageRepository ->getPageIdsRecursive()	51955	49233	53169	61006	59127	65724	59568	64348	67295	63859
PageTreeView ->getTree()	45274	53551	45466	51436	59263	64986	57613	56355	58699	46926
buildRcte()	1014	840	789	915	948	973	819	787	939	1148

Tab. 4.6: *Ergebnisse der Zeitmessungen in Millisekunden*

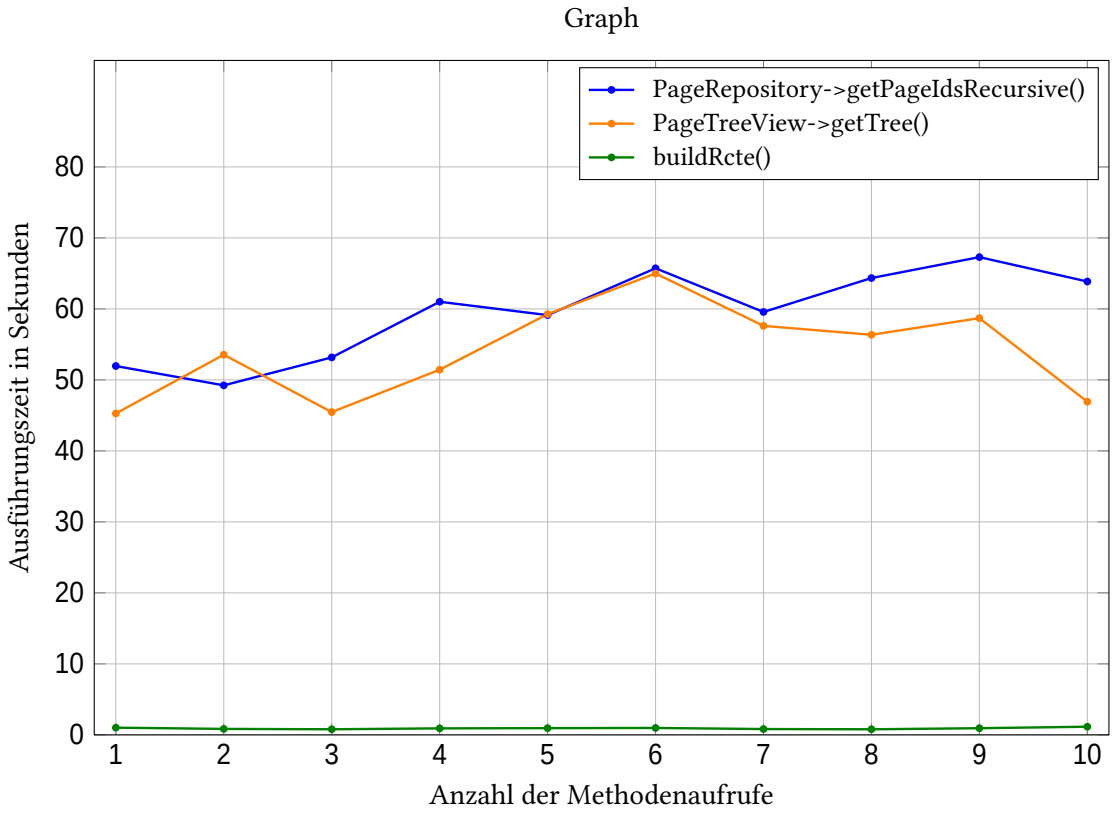


Abb. 4.3: *Einzelergebnisse der Zeitmessungen*

Die Testumgebung befindet sich zwar in einem separaten Docker Container und ist logisch vom Hostsystem isoliert, aber der Container nutzt die selben Hardwareressourcen wie das Hostsystem. Die Testumgebung befindet sich auf einem privatem Laptop, der seit mehreren Jahren in Benutzung ist und auf dem diverse Programme/Tools installiert wurden. Es wäre durchaus möglich, dass die Messungen von außen durch z.B. Softwareupdates oder Virenscans beeinträchtigt wurden.

Um diese Beeinflussung auszuschließen wird für jede Messreihe zunächst die **Standardabweichung** ermittelt. Die Standardabweichung gibt Aufschluss darüber, wie sehr die Messergebnisse von der durchschnittlichen Ausführungszeit, dem **Mittelwert**, abweichen. Je größer die Standardabweichung einer Messreihe ist, desto stärker weichen die einzelnen Messergebnisse vom

Mittelwert bzw. Durchschnittswert ab. Die jeweils zehn Ausführungszeiten pro Methode sollten im besten Fall möglichst gleichbleibend sein und so zu einer geringen Standardabweichung führen.
Die Standardabweichung ist ein Mittel um Messfehler in einer Messreihe zu identifizieren. Sie liefert allerdings absolute Werte, mit denen verschiedene Messreihen nicht sinnvoll miteinander verglichen werden können. Dafür wird der **Variationskoe zient** ermittelt. Dieser Wert drückt die Standardabweichung in Prozent aus. Damit lässt sich die Streuung der Messwerte auch für Methoden vergleichen, die unterschiedlich lange Ausführungszeiten haben. Ein Variationskoeffizient von 0,05 würde bedeuten, dass die Messergebnisse um 5% vom Mittelwert/Durchschnitt abweichen und damit ein guter Wert wären.

Die Formeln des Mittelwertes, der Standardabweichung und des Variationskoeffizient lauten wie folgt.

$$= \frac{1}{n} \sum_{i=1} x_i \qquad (4.1)$$

Formel 4.1: *Formel Mittelwert*

$$= \sqrt{\sum_{i=1} (x_i \quad)^2} \qquad (4.2)$$

Formel 4.2: *Formel Standardabweichung*

Wobei den Mittelwert darstellt, die Standardabweichung, n die Anzahl der Methodenaufrufe und x_i die Ausführungszeit der einzelnen Methodenaufrufe.

$$V = — \qquad (4.3)$$

Formel 4.3: *Formel Variationskoe zient.*

Wobei V den Variationskoe zient darstellt, die Standardabweichung und den Mittelwert.

Nachfolgend wird aus den Messwerten für jede Methode der Mittelwert, die Standardabweichung und der Variationskoeffizient berechnet. Diese werden anschließend in der Tabelle 4.7 gegenübergestellt.

PageRepo

$$\mathbf{59528} = _a = \frac{51955 + 49233 + 53168 + 61006 + 59127 + 65724 + 59568 + 64348 + 67295 + 63859}{10} \qquad (4.4)$$

Formel 4.4: *Mittelwertberechnung von PageRepo() (Werte in Millisekunden).*

$$\mathbf{5889} = {}_a = \sqrt{\frac{1}{10}\begin{array}{l}(51955 \quad {}_a)^2 + (49233 \quad {}_a)^2 + (53168 \quad {}_a)^2 + (61006 \quad {}_a)^2 + (59127 \quad {}_a)^2 \\ +(65724 \quad {}_a)^2 + (59568 \quad {}_a)^2 + (64348 \quad {}_a)^2 + (67295 \quad {}_a)^2 + (63859 \quad {}_a)^2\end{array}} \tag{4.5}$$

Formel 4.5: *Berechnung der Standardabweichung von PageRepo() (Werte in Millisekunden).*

$$\mathbf{0,099} = V_a = \frac{5889}{59528} \tag{4.6}$$

Formel 4.6: *Berechnung des Variationskoe zient von PageRepo() (Werte in Millisekunden).*

PageTreeView

$$\mathbf{53957} = {}_b = \frac{45274 + 53551 + 45466 + 51436 + 59263 + 64986 + 57613 + 56355 + 58699 + 46926}{10} \tag{4.7}$$

Formel 4.7: *Mittelwertberechnung von PageTreeView() (Werte in Millisekunden).*

$$\mathbf{6284} = {}_b = \sqrt{\frac{1}{10}\begin{array}{l}(45274 \quad {}_b)^2 + (53551 \quad {}_b)^2 + (45466 \quad {}_b)^2 + (51436 \quad {}_b)^2 + (59263 \quad {}_b)^2 \\ +(64986 \quad {}_b)^2 + (57613 \quad {}_b)^2 + (56355 \quad {}_b)^2 + (58699 \quad {}_b)^2 + (46926 \quad {}_b)^2\end{array}} \tag{4.8}$$

Formel 4.8: *Berechnung der Standardabweichung von PageTreeView() (Werte in Millisekunden).*

$$\mathbf{0,116} = V_b = \frac{6284}{53957} \tag{4.9}$$

Formel 4.9: *Berechnung des Variationskoe zient von PageTreeView() (Werte in Millisekunden).*

rCTE

$$\mathbf{917} = {}_c = \frac{1014 + 840 + 789 + 915 + 948 + 973 + 819 + 787 + 939 + 1148}{10} \tag{4.10}$$

Formel 4.10: *Mittelwertberechnung von buildRcte() (Werte in Millisekunden).*

$$\mathbf{108} = \sigma_c = \sqrt{\frac{1}{10}\left(\begin{array}{l}(1014 - \mu_c)^2 + (840 - \mu_c)^2 + (789 - \mu_c)^2 + (915 - \mu_c)^2 + (948 - \mu_c)^2 \\ +(973 - \mu_c)^2 + (819 - \mu_c)^2 + (787 - \mu_c)^2 + (939 - \mu_c)^2 + (1148 - \mu_c)^2\end{array}\right)} \tag{4.11}$$

Formel 4.11: *Berechnung der Standardabweichung von buildRcte() (Werte in Millisekunden).*

$$\mathbf{0{,}118} = V_c = \frac{108}{917} \tag{4.12}$$

Formel 4.12: *Berechnung des Variationskoeffizient von buildRcte() (Werte in Millisekunden).*

Bewertung der Messergebnisse

In der folgenden Tabelle sind die Mittelwerte der Messungen, die Standardabweichungen und die Variationskoeffizienten zusammengefasst.

Methode	**Ausführungszeit** (Mittelwert in Sekunden)	**Standardabweichung** (in Sekunden)	**Variationskoeffizient**
PageRepository ->getPageIdsRecursive()	59,528	5,889	9,9 %
PageTreeView ->getTree()	53,957	6,284	11,6 %
buildRcte()	0,917	0,108	11,8 %

Tab. 4.7: *Zusammenfassung der Messergebnisse (Zeit)*

Für die Erstellung des Seitenbaum-Arrays benötigten die Methoden des TYPO3 Kernels deutlich länger als die Extension. Mit einer mittleren Ausführungszeit von 59,528 Sekunden benötigte *PageRepository->getPageIdsRecursive()* am längsten. Mit 53,957 Sekunden benötigte die Methode *PageTreeView->getTree()* etwas weniger Zeit. Am schnellsten wurde das Seitenbaum-Array mit *buildRcte()* in 0,917 Sekunden erzeugt. Damit ist die **recursive CTE ca. 59 bzw. 65 mal schneller als die TYPO3 Core Methoden**.

Die Tabelle zeigt ebenfalls, dass die Standardabweichung nicht als Maß geeignet ist um die Streuung der Messergebnisse zu vergleichen. Bei einer Ausführungszeit von 0,9 Sekunden wird *buildRcte()* keine Standardabweichung von mehr als 5 Sekunden erreichen, wie die anderen Methoden.
Der Variationskoeffizient zeigt aber, dass die Streuung der Messergebnisse bei allen Methoden bei ca. 10-12% lag. Der Wert ist zwar nicht ideal, aber durchaus als gut anzusehen. Vor allem ist er für alle Methoden konstant und deutet darauf hin, dass es während der Messungen keine Beeinflussungen gab, die sich nur auf eine Testreihe ausgewirkt hat.

Die Messergebnisse sind zur Visualisierung im folgenden Graph dargestellt. Der Variationskoeffizient ist als kleine Linie an den Endpunkten der Balken dargestellt. Aufgrund der Skalierung wurde der Variationskoeffizient bei *buildRcte()* weggelassen, da dieser zu klein wäre um Werte sinnvoll ablesen zu können.

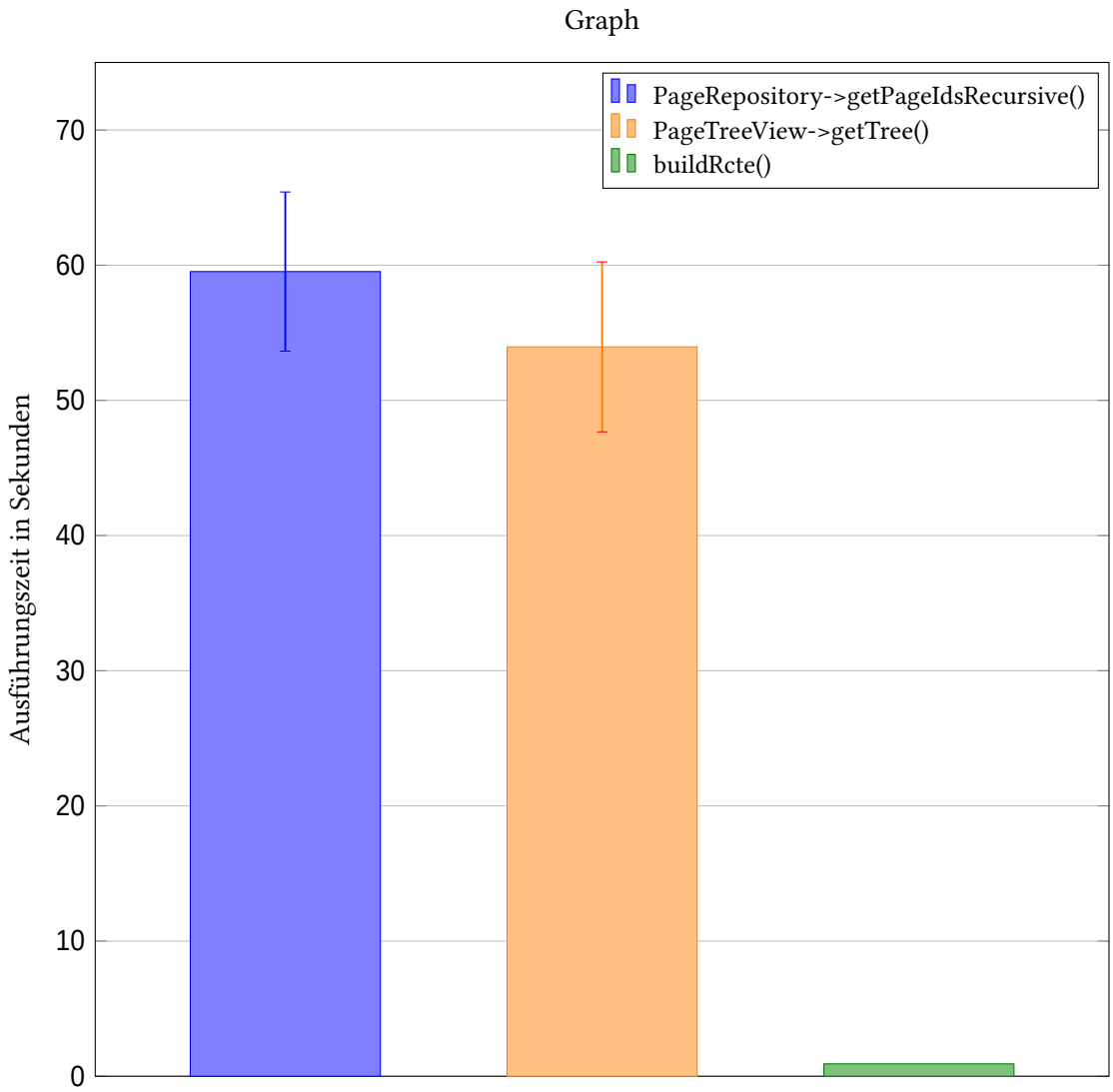


Abb. 4.4: *Ergebnisse Ausführungszeiten*

Der Performancegewinn, der von *buildRcte()* mittels recursive CTE erreicht wird, ist deutlich zu sehen. Die Visualisierung des Variationskoeffizient zeigt zudem, dass die Ausführungszeit von *PageRepository->getPageIdsRecursive()* bei einer vorteilhaften Messung kürzer sein kann, als von *PageTreeView->getTree()*. Ebenso bei einem unvorteilhaften Messwert für *PageTreeView->getTree()*. Die Ausführungszeit von *buildRcte()* können allerdings beide Methoden des TYPO3 Cores, auch bei einer vorteilhaften Messung, nicht erreichen.

4.4.2 SQL-Statements

Um die ausgeführten SQL-Statements der Methoden zu erfassen, werden die Methoden nacheinander ausgeführt. Anschließend werden die Statements über das Logging der Datenbank analysiert. Als Output für das Logging wird die Datenbank verwendet. Dadurch werden die Einträge in eine Loggingtabelle geschrieben und können mit SQL-Statements ausgewertet werden. Andernfalls müssten, neben MySQL Workbench, weitere Tools genutzt werden um die Logs von MariaDB auszuwerten.

Um die SQL-Statements zu analysieren, werden die Testmethoden einzeln ausgeführt. Vor jedem Test wird die execute-Methode in SiteStateCommand.php manuell angepasst, sodass bei

der Ausführung nur eine Testmethode ausgeführt wird. Die Ausführung wird über die boolschen Variablen *$pageRepo*, *$pageTreeView* und *$rCte* gesteuert.

```
1  protected function execute(InputInterface $input, OutputInterface $output): int
2  {
3      ...
4      // Production part
5      //$this->stateService->cte();
6
7      // Performance testing part
8      $times=1;
9      $pageRepo=true;
10     $pageTreeView=false;
11     $rCte=flase;
12
13     $this->performanceTests($times, $pageRepo, $pageTreeView, $rCte);
14     ...
15 }
```

Code 4.8: *Vereinfachter Aufbau von execute()*

Zeile	Erläuterung
5	Hauptmethode ist auskommentiert.
8	Nur eine Ausführung der Testmethode.
9	Variable *$pageRepo* wird auf *true* gesetzt. Bedeutet, *PageRepository->getPageIdsRecursive()* wird ausgeführt.
10+11	Variablen *$pageTreeView* und *$rCte* werden auf *false* gesetzt. Bedeutet, *PageTreeView->getTree()* und *buildRcte()* werden nicht ausgeführt.
13	Aufruf der Wrapper Testmethode.

Tab. 4.8: *Erläuterung von Code 4.8*

Die ermittelten Daten der SQL-Satements sind in der folgenden Tabelle zusammengefasst. Die Spalten mit den Start- und Endzeitpunkten der Messungen wurden zur besseren Übersicht entfernt.

Methode	Connect_Count	Quit_Count	Prepare_Count	Execute_Count	Close_Count	Query_Count
PageRepository->getPageIdsRecursive()	2	2	30267	30267	30267	4
PageTreeView->getTree()	2	2	30236	30236	30236	4
buildRcte()	2	2	2	2	2	4

Tab. 4.9: *Zusammenfassung der Messergebnisse (SQL)*

Die Werte unterscheiden sich zwischen den Methoden nur in der Anzahl der prepare, execute und close Statements. Erwartungsgemäß haben die TYPO3 Core Methoden für jeden Page Eintrag ein separates Statement ausgeführt. Dadurch sind mit über 90.000 Statements bedeutend mehr Statements ausgeführt worden, als durch *buildRcte()*.

Prepare und execute Statements im Detail

Es könnte erwartet werden, dass *buildRcte()* nur ein prepare/execute/close Statement auslöst. Die Ursache für das zusätzliche Statement ist, dass TYPO3 auch für Datenbankzugriffe durch Extensions eine Benutzerprüfung durchführt. Dies wird deutlich, indem man den Inhalt der Statements

ansieht. Dazu wird das SQL-Script für die Auswertung wie folgt angepasst und nochmals ausführt. Die Variable *@timestmp* bleibt dabei unverändert.

```
1 WITH
2    ...
3 -- Ausgabe der Daten
4 SELECT event_time, command_type, argument FROM data_prepare
5 UNION
6 SELECT event_time, command_type, argument FROM data_execute
7 UNION
8 SELECT event_time, command_type, argument FROM data_close
9 ORDER BY event_time ASC;
```

Code 4.9: *Anpassung der SQL-Auswertung für buildRcte()*

event_time	command_type	argument
2024-12-07 17:48:14.691324	Prepare	SELECT * FROM ‘be_users‘ WHERE (‘username‘ = ?) AND...
2024-12-07 17:48:14.691550	Execute	SELECT * FROM ‘be_users‘ WHERE (‘username‘ = ’_cli_’)...
2024-12-07 17:48:14.692465	Close stmt	
2024-12-07 17:48:14.760135	Prepare	WITH RECURSIVE data AS (WITH allLivePages_exclWsEdit...
2024-12-07 17:48:14.760248	Execute	WITH RECURSIVE data AS (WITH allLivePages_exclWsEdit (...
2024-12-07 17:48:14.760634	Close stmt	

Tab. 4.10: *Detaillierte SQL-Auswertung für buildRcte()*

In den ersten drei Zeilen erkennt man, dass für die TYPO3 interne Verarbeitung alle Attribute des Benutzers abgefragt werden. Das Feld *‘username‘* ist dabei die Variable, die später den Benutzernamen enthält. Erkennbar ist dies daran, dass im prepared Statement die Variable username den Wert ? besitzt (*‘username‘* = ?). Im execute Statement wird dieser Wert durch _cli_ ersetzt (*‘username‘* = *‘_cli_‘*). Anschließend wird das prepared Statement geschlossen.
Der zweite prepare/execute/close Block ist die erwartete recursive CTE.

In den Messergebnissen in Tabelle 4.9 fällt zudem auf, dass sich die Anzahl der prepare/execute/close Statements zwischen den TYPO3 Core Methoden unterscheidet. Obwohl die Methoden den gleichen Seitenbaum ermittelt haben, wurden von *PageRepository->getPageIdsRecursive()* 31 prepare/execute/close Statements mehr ausgelöst.

Die Ursache kann durch den Aufbau des Seitenbaums und durch die Datenbanklogs nachvollzogen werden. Das SQL-Script für die Auswertung wird dazu nochmals anpasst.

Abb. 4.5: *Seitenbaum Testdatensatz*

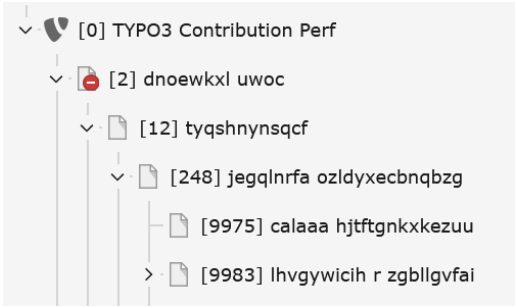


Abb. 4.6: *Pages im Seitenbaum des Testdatensatzes*

```
WITH
    ...
-- Ausgabe der Daten
SELECT event_time, command_type, argument FROM data_execute
ORDER BY event_time ASC;
```

Code 4.10: *Anpassung der SQL-Auswertung für getPageIdsRecursive()*

event_time	command_type	argument
2024-12-07 22:52:08.696151	Execute	SELECT ‘uid‘ FROM ‘pages‘ WHERE (‘uid‘ = 2) AND (‘pages‘.‘deleted‘ = 0)
2024-12-07 22:52:08.696957	Execute	SELECT ‘uid‘, ‘pid‘, ‘doktype‘, ‘mount_pid‘, ‘mount_pid_ol‘, ‘t3ver_state‘, ‘l10n_parent‘ FROM ‘pages‘ WHERE (‘uid‘ = 2) AND (‘pages‘.‘deleted‘ = 0)
2024-12-07 22:52:08.698193	Execute	SELECT * FROM ‘pages‘ WHERE (‘pid‘ = 2) AND...
2024-12-07 22:52:08.700187	Execute	SELECT * FROM ‘pages‘ WHERE (‘pid‘ = 12) AND...
2024-12-07 22:52:08.701517	Execute	SELECT * FROM ‘pages‘ WHERE (‘pid‘ = 248) AND...
...	...	...
2024-12-07 22:53:04.128300	Execute	SELECT ‘uid‘ FROM ‘pages‘ WHERE (‘uid‘ = 38182) AND (‘pages‘.‘deleted‘ = 0)
2024-12-07 22:53:04.129458	Execute	SELECT ‘uid‘, ‘pid‘, ‘doktype‘, ‘mount_pid‘, ‘mount_pid_ol‘, ‘t3ver_state‘, ‘l10n_parent‘ FROM ‘pages‘ WHERE (‘uid‘ = 38182) AND (‘pages‘.‘deleted‘ = 0)
2024-12-07 22:53:04.130994	Execute	SELECT * FROM ‘pages‘ WHERE (‘pid‘ = 38182) AND ...
2024-12-07 22:53:04.133089	Execute	SELECT * FROM ‘pages‘ WHERE (‘pid‘ = 38183) AND ...
2024-12-07 22:53:04.135111	Execute	SELECT * FROM ‘pages‘ WHERE (‘pid‘ = 39348) AND ...
...	...	...

Tab. 4.11: *Detaillierte SQL-Auswertung für getPageIdsRecursive()*

Der Methode *PageRepository->getPageIdsRecursive()* wird ein Array mit den Page-uid’s als Parameter übergeben. Für jede übergebene Page-uid werden zwei Statements ausgeführt um Informationen zu Löschstatus, Workspaceversion usw. zu erhalten. Erst danach werden die Statements für die untergeordneten Pages ausgeführt. Der Testdatensatz besitzt 16 Pages auf oberster Ebene, die als Parameter übergeben werden. Dies bedeutet, dass 32 zusätzliche Statements ausgeführt werden. Da der Methode *PageTreeView->getTree()* die Seiten nicht als Parameter übergeben werden, wird dort ein separates Statement ausgeführt um alle Pages auf der obersten Ebene zu ermitteln. Dadurch entsteht insgesamt eine Differenz von 31 Statements zwischen den Core Methoden.

event_time	command_type	argument
2024-12-23 20:26:37.421949	Execute	SELECT ‘uid‘... FROM ‘pages‘ WHERE (‘pid‘ = 0) AND...
2024-12-23 20:26:37.434136	Execute	SELECT ‘uid‘... FROM ‘pages‘ WHERE (‘pid‘ = 2) AND...
2024-12-23 20:26:37.435566	Execute	SELECT ‘uid‘... FROM ‘pages‘ WHERE (‘pid‘ = 12) AND...
2024-12-23 20:26:37.436740	Execute	SELECT ‘uid‘... FROM ‘pages‘ WHERE (‘pid‘ = 248) AND...
...	...	...

Tab. 4.12: *Detaillierte SQL-Auswertung für getTree()*

Bewertung der Messergebnisse

Die Auswertung der SQL-Statements entspricht im wesentlichen den erwarteten Ergebnissen. Ohne eine recursive CTE wird für jedes Objekt im Seitenbaum ein separates prepare, execute und close Statement ausgeführt. Dadurch entstehen durch die **Core Methoden 30.236 bzw. 30.267 prepare/execute/close Statements** die von TYPO3 und der Datenbank verarbeitet werden müssen. Mit der erstellten Extension werden mittels **recursive CTE hingegen nur 2 prepare/execute/close Statements** ausgeführt.
In jedem Testlauf wurden 2 Datenbankverbindungen durch den Querybuilder erstellt und beendet. Sie geben keinen Hinweis auf die Performanceunterschiede bei den Zeitmessungen. Ebenso wurden je Testlauf 4 Queries ausgeführt, die für die Performance nicht relevant sind. Sie beinhalten nur verbindungsspezifische Optionen, wie z.B. die Konfiguration des “Character Set”.

Die Verarbeitung von über 90.000 SQL-Statements (Summe aus prepare, execute und close) kann durch die erstellte Extension auf 6 SQL-Statements reduziert werden. Dabei ist es egal, wie groß die pages-Tabelle ist. Dieser Unterschied kann besonders zum Tragen kommen, wenn die Datenbank auf einem separaten Server betrieben wird und die einzelnen Statements über das Netzwerk übertragen werden müssen.

Kapitel 5

Zusammenfassung

Als Ergebnis dieser Arbeit steht ein Prototyp in Form einer TYPO3 Extension zu Verfügung, welcher die Umsetzbarkeit von rekursiven SQL-Statements in TYPO3 aufzeigt. Die Umsetzung erfolgt dabei für die Erstellung des TYPO3 Seitenbaums. Außerdem wurde ein allgemeines Schema zur Erstellung von rekursiven SQL-Statements entwickelt. Das Schema teilt die komplexe Struktur eines rekursiven SQL-Statements in kleinere Teile auf und macht sie dadurch besser nachvollziehbar und deutlich wartungsfreudiger. Das Schema wird auch durch die Methoden der Extension umgesetzt, siehe Abbildung 3.2.

Auf dem Weg dahin wurden anhand der Literaturrecherche die **Gundlagen** zum Thema SQL und dessen Verwendung in TYPO3 recherchiert. Daraus wurde ersichtlich, dass die Nutzung von rekursiven SQL-Statements besonders für große Datenmengen einen Performancevorteil gegenüber iterativen SQL-Statements bieten kann. Voraussetzung dafür ist, dass die Datensätze eine rekursive Struktur besitzen. Hinweise auf solche Strukturen ließen sich im TYPO3 Seitenbaum, im Kapitel 2.3.2 erkennen. Dort sind einzelne Seiten immer Unterseiten von anderen Seiten. Der GSB 11, als zentrale Content-Management-Lösung für die Webangebote der deutschen Bundesverwaltung, wird ebenfalls auf TYPO3 basieren. Es wird angenommen, dass der GSB 11 große Datenmengen verwalten wird und entsprechend große Seitenbäume aufweisen wird.

Um genauere Erkenntnisse zur Datenbankstruktur von TYPO3 und der Nutzung von rekursiven SQL-Statements zu erhalten, wurde als **Methodik** eine TYPO3 Instanz installiert, die analysiert wurde. Für die Installation wurden DDEV und Docker in Verbindung mit WSL verwendet. Als Analysetools wurden VSCode für den Quellcode und MySQL Workbench für die Datenbank genutzt, siehe Kapitel 3.1. Während des Projektes wurde ebenfalls mit TYPO3 Core Entwicklern Rücksprache gehalten um ein besseres Verständnis für TYPO3 eigene Methoden zu erhalten. Dadurch konnten Erkenntnisse zu Methoden gewonnen werden, die bisher weder zur freien Verwendung in TYPO3 vorgesehen, noch dokumentiert sind.

Die Erkenntnisse aus der Analyse der TYPO3 Datenbank haben gezeigt, dass eine Änderung des Datenbankschemas alleine keinen Performancegewinn bringen würde. Auf Datenbankebene besitzen die einzelnen Tabellen keine Verbindung untereinander, durch z.B. foreign keys. Die Integrität der Daten wird in TYPO3 selbst durch das *Table Con guration Array* sichergestellt, siehe Kapitel 2.3. Das Array bietet den Vorteil, das Datenbankschema sehr flexibel zu gestalten und Änderungen besser umsetzen zu können, da keine Beziehungen in der Datenbank aufgelöst werden müssen. Außerdem bietet es Entwicklern von Extensions eine einfache Möglichkeit zusätzliche Tabellen/Attribute zur TYPO3 Datenbank hinzuzufügen. Eine Änderung an diesem Konzept würde ein grundlegendes Prinzip von TYPO3 ändern und würde den Rahmen dieser Bachelorarbeit überschreiten.

Die Analyse der Datenbank hat auch gezeigt, dass die Datenstruktur des Seitenbaumes rekursiv

aufgebaut ist. In der Tabelle “pages” sind alle Seiten des Seitenbaumes enthalten. Jede Seite ist über das Attribut “pid (parent identifier)” mit dem Attribut “uid (unique identifier)” der übergeordneten Seite verknüpft.

Aus der Analyse des TYPO3 Quellcodes ging hervor, dass die Bestandsmethoden keine rekursiven SQL-Statements nutzen um den Seitenbaum zu erstellen bzw. das Array, das diesen repräsentiert.
Im Rahmen dieser Bachelorarbeit wurde daher eine Extension entwickelt, welche den Seitenbaum mittels rekursive CTE erstellt. Dafür wurde zunächst ein reines SQL-Statement in Kapitel 3.6 erstellt. Das SQL-Statement erstellt mehrere temporäre Tabellen und ist nach einem Schema aufgebaut, dass sich leicht für andere Anwendungsfälle anpassen lässt. Dafür wird zunächst eine **bereinigte Datengrundlage** erstellt, die aus einer Teilmenge der pages-Tabelle besteht. Von dieser Datengrundlage werden anschließend **Teildaten gebildet**. Die Ermittlung der Teildaten ist nötig um alle Seiten eines Workspace korrekt abzubilden. Danach wird aus den Teildaten die **Datentabelle** erstellt. Diese temporäre Tabelle wird als Input für den rekursiven Teil der CTE genutzt. Um die Ergebnistabelle in der korrekten Reihenfolge zu sortieren, wird das Feld *__CTE_Sorting__* im rekursiven Teil der CTE erstellt.

Um das SQL-Statement in TYPO3 nutzen zu können, wurde eine TYPO3 Extension erstellt und im Kapitel 3.7 vorgestellt. Die Extension stellt einen Service bereit, der über ein Terminal ausführbar ist. Der Service besteht aus Methoden, welche die einzelnen, temporären Tabellen des SQL-Statements erstellen. Die Kommunikation mit der Datenbank erfolgt dabei über den Querybuilder von Doctrine DBAL und ist somit unabhängig von der Syntax des verwendeten DBMS. Das SQL-Statement, das am Ende an die Datenbank gesendet wird, enthält nur die absolut nötigen Tabellenfelder. Den Servicemethoden können aber weitere Tabellenfelder als Parameter übergeben werden.

In der **Analyse** wurden die TYPO3 Core Methoden *PageRepository->getPageIdsRecursive()* und *PageTreeView->getTree()* mit der Extension verglichen. Es konnte nachgewiesen werden, dass das Seitenbaum-Array mit der Extension deutlich schneller erstellt wird, als mit den Core Methoden. Die Ausführungszeiten konnten, für eine große TYPO3 Instanz, **von über 53 Sekunden auf weniger als 1 Sekunde** reduziert werden, siehe Abbildung 4.4. Dieser Zeitunterschied stellt nicht nur einen Messunterschied dar, sondern ist auch für Anwender deutlich wahrnehmbar.
Zudem wurde in Tabelle 4.9 aufgezeigt, dass sich die Anzahl der SQL-Statements **von über 90.000 Statements auf 6 Statements** reduziert. Dies stellt ebenfalls eine deutliche Verbesserung zu den TYPO3 Core Methoden dar. Der Vorteil wird besonders dann zu tragen kommen, wenn sich die Datenbank auf einem separaten Server befindet und jedes SQL-Statement Verzögerungen durch die Netzwerkübertragung verursacht.

5.1 Kritische Betrachtung

Die Extension unterscheidet bei der Erstellung des Seitenbaums zwischen verschiedenen Workspaces. Andere Bedingungen, wie z.B. der Seitentyp oder Berechtigungen, werden nicht überprüft.

Es ist anzunehmen, dass sich die Ausführungszeit der Extensionmethoden minimal verlängert, wenn diese zu einem vollwertigem Ersatz der TYPO3 Core Methoden ausgebaut wird. Auch wenn der Unterschied voraussichtlich nur wenige Millisekunden betragen wird, ist dies in den aktuellen Ergebnissen zu berücksichtigen.
Für die Messungen wurde eine lokale, mit Docker virtualisierte Testumgebung genutzt. Datenbanken wird in diesem Szenario eine schlechte Performance nachgesagt. Die Größenunterschiede in den Ausführungszeiten könnten in einer nicht virtualisierten Umgebung ggf. geringer ausfallen.

In den Messungen wurde zudem nur das Szenario einer lokalen MariaDB Instanz nachgestellt. Messungen mit allen, von TYPO3 unterstützen DBMS, sowohl lokal als auch remote, konnten aus Zeitgründen leider nicht vorgenommen werden. Sie würden ein umfassenderes Bild der Performanceunterschiede aufzeigen.

Rekursive SQL-Statements können zwar einen deutlichen Performancegewinn bedeuten, aber sie bringen immer eine gewisse Komplexität mit sich. Besonders als open source CMS muss der Quellcode von TYPO3 für fremde Entwickler verständlich und nachvollziehbar sein. Rekursive SQL-Statements sollten daher nur selektiv eingesetzt werden und nur nachdem der Nutzen mit der künftigen Wartbarkeit abgewogen wurde. Ergänzend sollten auch andere Techniken in Betracht gezogen werden, wie zum Beispiel einem partiellem Laden oder Cachingstrategien. Letztere werden ohnehin schon an vielen Stellen in TYPO3 vorgenommen, sodass der Seitenbaum nicht in jeder Situation komplett erstellt werden muss.

5.2 Ausblick

Die erstellte Extension ist ein Prototyp, der die Umsetzbarkeit von rekursivem SQL für den TYPO3 Seitenbaum aufzeigt. Um diesen Prototypen in einer produktiven TYPO3 Instanz einsetzen zu können, sind noch Überarbeitungen vonnöten.

Es fehlen noch Berechtigungsprüfungen, die eingepflegt werden müssen. Dies betrifft sowohl die Bedingungen der SQL-Statements, als auch die Initialisierung der Querybuilder-Objekte. Anpassungen am SQL-Statement müssen auch vorgenommen werden, wenn den Zielen von “link”-Seiten gefolgt werden soll. Als Sicherheitsaspekt sollte zudem die Übergabe der SQL-Parameter überprüft werden, um die Gefahr von SQL-Injections zu minimieren.
Die Erstellung von Tests ist für eine Integration in TYPO3 ebenfalls von Nöten. So wird auch sichergestellt, dass der Seitenbaum von allen unterstützen DBMS korrekt erstellt wird.

Mit der TYPO3 eigenen Erweiterung des Querybuilders können zum ersten Mal rekursive SQL-Statements mit diesem umgesetzt werden. Diese Arbeit schafft in der TYPO3 Community ein Bewusstsein für diese Möglichkeit und stellt zugleich eine mögliche Herangehensweise zur Nutzung dieses neuen Features vor. Während der Erstellung des Prototyps wurden erste Zwischenergebnisse auf dem “TYPO3camp Berlin-Brandenburg 2024” vorgestellt [18], was die Bedeutung von recursive CTE’s ebenfalls hervorgehoben hat.

Neben dem Seitenbaum können auch andere, rekursive Datenstrukturen in TYPO3 für eine Umsetzung mittels recursive CTE in Betracht gezogen werden. Die Anwendungsmöglichkeiten beschränken sich dabei nicht nur auf eine TYPO3 Core Instanz. Auch in Extensions, die eine eigene Datenbankstruktur besitzen, können rekursive SQL-Statements eingesetzt werden.

Die Möglichkeit rekursive SQL-Statements mit dem Querybuilder zu erstellen besteht aktuell nur in TYPO3 Version 13.4. Seitens TYPO3 wird angestrebt die eigene Umsetzung auch in die offizielle Version von Doctrine DBAL zu übertragen. Dadurch würden auch andere PHP-Projekte von diesem Feature und den Erkenntnissen aus dieser Arbeit profitieren. Dazu zählen unter anderem Symfony, Drupal und Laravel [19].

Anhang A

Projektdateien

Die erstellten Dateien der Extension befinden sich in der angehängten Archivdatei, im Ordner "*/Code/Extension/*" und online unter https://github.com/ASchoenfeld-BHT/TYPO3_recursive_cte. Die erstellten SQL-Scripts befinden sich im Ordner "*/Code/SQL-Scripts/*" und ebenfalls unter dem o.g. Link.

Für die Testdaten liegt aktuell noch keine allgemeine Freigabe vor, weswegen diese vorerst nur im Anhang unter "*/Code/Testdata/pages.sql*" zur Verfügung stehen. Sobald die Freigabe durch das TYPO3 Core Team erteilt wird, werden diese ebenfalls in das GIT Repository hochgeladen.

Dateistruktur

```
/
├── Bachelorarbeit_Schoenfeld_TYPO3.pdf
└── Code
    ├── Extension
    │   └── typo3conf
    │       └── ext
    │           └── cte_page_tree
    │               ├── Classes
    │               │   ├── Command
    │               │   │   └── SiteStateCommand.php
    │               │   └── Services
    │               │       └── StateService.php
    │               ├── Configuration
    │               │   └── Services.yaml
    │               ├── composer.json
    │               └── ext_emconf.php
    ├── SQL-Scripts
    │   ├── Performancemessung_v1.1.sql
    │   └── rCTE_v1.4.sql
    └── Testdata
        └── pages.sql
```

Literatur

1. WIKIPEDIA: *Tiefensuche*. 2023. Auch verfügbar unter: https://de.wikipedia.org/wiki/Tiefensuche. Abgerufen am: 28. Dezember 2024. [Online].
2. HANNA, K. T.: *Rundlaufverfahren (Round Robin)*. 2024. Auch verfügbar unter: https://www.computerweekly.com/de/definition/Rundlaufverfahren-Round-Robin. Abgerufen am: 28. Dezember 2024. [Online].
3. W³TECHS: *Usage Statistics and Market Share of Content Management Systems*. 2024. Auch verfügbar unter: https://w3techs.com/technologies/overview/content_management. Abgerufen am: 25. Dezember 2024. [Online].
4. KÖLBEL, J.: *Statistische Verteilung von Content-Management-Systemen im ö entlichen Bereich in Deutschland*. 2022. Auch verfügbar unter: https://cmscensus.eu/fileadmin/introduction/content/BA_JudithKoelbel_2022.pdf. Berliner Hochschule für Technik.
5. ITZBUND: *Government Site Builder 11*. 2024. Auch verfügbar unter: https://produkt.gsb.bund.de/gsb11. Abgerufen am: 25. Dezember 2024. [Online].
6. LOVE, A.: *SQL Clause, Statement, Command, Expression and Batch De ned*. 2023. Auch verfügbar unter: https://www.mssqltips.com/sqlservertip/7575/sql-clause-statement-command-expression-batch-definition/. Abgerufen am: 25. Dezember 2024. [Online].
7. MARIADB CORPORATION: *Thread Command Values*. 2014. Auch verfügbar unter: https://mariadb.com/kb/en/thread-command-values. Abgerufen am: 25. Dezember 2024. [Online].
8. TYPO3 DOCUMENTATION TEAM: *Versioning and Workspaces*. 2024. Auch verfügbar unter: https://docs.typo3.org/m/typo3/reference-coreapi/main/en-us/ApiOverview/Workspaces/Index.html. Abgerufen am: 25. Dezember 2024. [Online].
9. TYPO3 COMMUNITY: *TYPO3 Extensions*. 2024. Auch verfügbar unter: https://extensions.typo3.org/. Abgerufen am: 25. Dezember 2024. [Online].
10. TYPO3 COMMUNITY: *TYPO3 Version 13 - System Requirements*. 2024. Auch verfügbar unter: https://get.typo3.org/version/13#system-requirements. Abgerufen am: 25. Dezember 2024. [Online].
11. TYPO3 COMMUNITY: *TCA - Table Con guration Array*. 2024. Auch verfügbar unter: https://docs.typo3.org/m/typo3/reference-tca/main/en-us/Introduction/Index.html. Abgerufen am: 25. Dezember 2024. [Online].
12. HÖGEL, B.: *Gewurzelter Baum*. 2020. Auch verfügbar unter: https://www.biancahoegel.de/mathe/graph/baum_gewurzelt.html. Abgerufen am: 27. Dezember 2024. [Online].
13. TYPO3 COMMUNITY: *Contribution Work ow - TYPO3 Documentation*. 2024. Auch verfügbar unter: https://docs.typo3.org/m/typo3/guide-contributionworkflow/main/en-us/Index.html. Abgerufen am: 25. Dezember 2024. [Online].

14. TYPO3 COMMUNITY: *PageRepository::getPageIdsRecursive - TYPO3 API Documentation*. 2024. Auch verfügbar unter: `https://api.typo3.org/main/classes/TYPO3-CMS-Core-Domain-Repository-PageRepository.html#method_getPageIdsRecursive`. Abgerufen am: 25. Dezember 2024. [Online].

15. TYPO3 COMMUNITY: *AbstractTreeView::getTree - TYPO3 API Documentation*. 2024. Auch verfügbar unter: `https://api.typo3.org/main/classes/TYPO3-CMS-Backend-Tree-View-AbstractTreeView.html#method_getTree`. Abgerufen am: 25. Dezember 2024. [Online].

16. BUERK, S.: *tf-basics-extension*. 2024. Auch verfügbar unter: `https://github.com/sbuerk/tf-basics-extension`. Abgerufen am: 25. Dezember 2024. [Online].

17. BUNDESMINISTERIUM FÜR UMWELT, NATURSCHUTZ UND NUKLEARE SICHERHEIT: *Themen A-Z*. 2023. Auch verfügbar unter: `https://www.bmuv.de/themen/themen-a-z`. Abgerufen am: 29. Dezember 2024. [Online].

18. T3CB TEAM: *Mitmachen - T3CB*. 2024. Auch verfügbar unter: `https://www.t3cb.de/mitmachen/`. Abgerufen am: 25. Dezember 2024. [Online].

19. DOCTRINE PROJECT: *Doctrine Project*. n.d. Auch verfügbar unter: `https://www.doctrine-project.org/`. Abgerufen am: 25. Dezember 2024. [Online].