

Informationssysteme

Informationssysteme

Skriptum zur Vorlesung - 01.09.2020

Dipl.-Ing. Msc. Paul Panhofer BSc.^{1*}

¹ ZID, TU Wien, Taubstummengasse 11, 1040, Wien, Austria

Abstract: Ein Informationssystem ist ein soziotechnisches System, das die Abarbeitung von Informationsnachfrage zur Aufgabe hat. Es handelt sich um ein Mensch/Aufgabe/Technik-System, das Daten produziert, beschafft, verteilt und verarbeitet.

Daneben bezeichnen Informationssysteme im allgemeineren Sinne Systeme von Informationen, die in einem wechselseitigen Zusammenhang stehen.

Die Begriffe Informationssystem und Anwendungssystem werden häufig synonym verwendet. Dabei werden Informationssysteme im engeren Sinne als computergestützte Anwendungssysteme verstanden. Es ist jedoch wichtig zu verstehen, dass ein Anwendungssystem mit Anwendungssoftware und Datenbank nur Teil eines Informationssystems sind.

MSC: paul.panhofer@gmail.com

Keywords:

Contents	
1. Datenbankentwurf <ul style="list-style-type: none"> 1.1. Datenbankentwurf 10 <ul style="list-style-type: none"> 1.1.1. Phasen des Datenbankentwurfs 10 1.1.2. Externe Phase 11 1.1.3. Konzeptionelle Phase 11 1.1.4. Logische Phase 11 1.1.5. Physische Phase 12 1.2. Entity Relationship Modell 12 <ul style="list-style-type: none"> 1.2.1. ER Modell 12 1.2.2. Entität 13 1.2.3. Beziehung 13 1.3. Relationale Modell 15 <ul style="list-style-type: none"> 1.3.1. Relationale Modell 15 1.3.2. Beziehungen zwischen Tabellen 15 1.3.3. Transformationsregel: Entität 16 	<ul style="list-style-type: none"> 1.3.4. Transformationsregel: 1..1 Beziehung 16 1.3.5. Transformationsregel: 1..n Beziehung 17 1.3.6. Transformationsregel: n..m Beziehung 17 1.3.7. Vererbungskonzept 19 1.3.8. Transformationsregel: Vererbung 19 1.4. Normalisierung 19 <ul style="list-style-type: none"> 1.4.1. Datenredundanz 19 1.4.2. Erste Normalform 21 1.4.3. Zweite Normalform 21 1.4.4. Dritte Normalform 22
2. SQL - Data Query Lanuguage <ul style="list-style-type: none"> 2.1. SQL Grundlagen <ul style="list-style-type: none"> 2.1.1. SQL - Structured Query Language 24 2.1.2. Kategorien von SQL Befehlen 24 2.1.3. DQL - Select Anweisung 25 	24

*E-mail: paul.panhofer@tuwien.ac.at

2.2. Select Klausel - Projektion	26	4.1.1. Funktionstypen	42
2.2.1. Select Klausel	26	4.1.2. Zeilenfunktionen	42
2.2.2. Wiederholte Spaltenausgabe	27	4.1.3. Kategorien von Zeilenfunktionen	43
2.2.3. Spaltenwerte bearbeiten	27	4.2. Datumsfunktionen	43
2.2.4. Spaltenwerte verknüpfen	27	4.2.1. Datumstypen vs. Zeichenketten	43
2.2.5. Spaltenalias	27	4.2.2. Erzeugen eines Datums	44
2.2.6. Pseudospalten	28	4.2.3. Konstrukturfunktionen	44
2.2.7. Pseudospalten: NEXTVAL, CURRVAL	28	4.2.4. Konvertieren von Datumswerten - to_date	44
2.2.8. Pseudospalte: ROWNUM	28	4.2.5. Datumsliterale	44
2.2.9. Pseudospalten: USER, UID	29	4.2.6. Datumswerte umwandeln - to_char()	45
2.2.10. Pseudospalten: SYSDATE, TIMESTAMP	29	4.2.7. Rundungsfunktionen - trunc(), round()	45
2.3. Case Klausel - Kondition	30	4.2.8. Erzeugen von Intervallen	46
2.3.1. Case Klausel	30	4.2.9. Geschachtelte Intervalle	47
2.3.2. Case Klauseln in select Abfragen	30	4.2.10. Datumsarithmetik	47
2.4. Where Klausel - Restriktion	31	4.2.11. Datumsfunktion: add_months()	48
2.4.1. Restriktion	31	4.2.12. Datumsfunktion: next_day()	48
2.4.2. Abfrageobjekt	31	4.2.13. Datumsfunktion: extract()	48
2.4.3. Logische Operatoren - AND, OR, XOR	31	4.2.14. Datumsfunktion: last_day()	48
2.4.4. Logischer Operator - LIKE	32	4.2.15. Datumsfunktion: next_day()	49
2.4.5. Logischer Operator - IN	32	4.2.16. Datumsfunktion: months_between()	49
2.4.6. Logischer Operator - IS	32	4.3. Textfunktionen	50
2.4.7. Logischer Operator - between	32	4.3.1. Textfunktion: instr()	50
2.4.8. Dreiwertige Logik	33	4.3.2. Textfunktionen: trim()	50
2.5. Order by Klausel - Sortierung	34	4.3.3. Textfunktion: length()	51
2.5.1. Order By Klausel	34	4.3.4. Textfunktion: soundex()	51
2.5.2. Sortierreihenfolge - asc/desc	34	4.3.5. Textfunktionen: lower(), upper()	52
2.5.3. Optionen - nulls first und nulls last	34	4.3.6. Textfunktion: replace()	52
2.6. Fetch Klausel - Limitierung	35	4.3.7. Textfunktion: substr()	52
2.6.1. Fetch Klausel	35	4.4. Numerische Funktionen	53
2.7. Offset Klausel - Paginierung	35	4.4.1. Rundungsfunktionen	53
2.7.1. Offset Klausel	35	4.4.2. Konvertierungsfunktionen	54
3. SQL - Datenaggregation	36	4.4.3. Numerische Funktion: abs()	54
3.1. Datenmodellierung	36	4.4.4. Numerische Funktion: exp(), log()	55
3.1.1. Relationale Modellierung	36	4.4.5. Numerische Funktion: mod()	55
3.1.2. Datenaggregation	36		
3.2. From Klausel	37	5. SQL - Aggregatfunktionen	56
3.2.1. Grundlagen	37	5.1. Aggregatfunktionen	56
3.3. Relationale Join	37	5.1.1. Aggregatfunktionen	56
3.3.1. Virtuelle Tabellen	37	5.1.2. Median	57
3.3.2. Jointyp: Inner Join	38	5.1.3. Standardabweichung	57
3.3.3. Jointyp: Natural Join	38	5.1.4. Aggregatfunktionen und NULL Werte	57
3.3.4. Jointyp: Cross Join	39	5.1.5. UNIQUE Operator	57
3.3.5. Jointyp: Full Outer Join	39		
3.3.6. Jointyp: Left-/Right Join	41		
4. SQL - Zeilenfunktionen	42	5.2. Group by Klausel	58
4.1. Zeilenfunktionen - Grundlagen	42	5.2.1. Neustrukturierung von Daten	58

5.2.2. Group by Klausel	58	7.2.8. DDL Befehl: alter table	75
5.2.3. Map Phase	58	7.2.9. Fallbeispiel: alter table	75
5.2.4. Group by Klausel	59	7.3. Datenbankartefakt: Constraint	76
5.2.5. Semantik der Gruppenbildung	60	7.3.1. Datenkonsistenz und Integrität	76
5.2.6. Fallbeispiel: Gruppierung	60	7.3.2. Constraints	76
5.2.7. Aggregate Phase	60	7.3.3. Verwalten von Constraints	77
5.2.8. Having Klausel	60	7.3.4. Constraintmonitoring	77
5.2.9. Null Aggregat	61	7.4. Datenbankartefakt: View	78
6. SQL - Komplexe Abfragen		7.4.1. View	78
6.1. Unterabfragen	62	7.4.2. DDL Befehl: create view	78
6.1.1. Unterabfragen	62	7.4.3. Variable Bedingungen	79
6.1.2. Formen von Unterabfragen	62	7.4.4. DDL Befehl: drop view	79
6.1.3. Subqueryform: Innere View	63	7.5. Datenbankartefakt: Index	79
6.1.4. Subqueryform: Skalare Abfragen	64	7.5.1. Index	79
6.1.5. Subqueryform: Konditionale Abfrage	64	7.5.2. Indextyp: B* Index	80
6.2. With Klausel	65	7.5.3. Indextyp: Funktionsindex	80
6.2.1. WITH Klausel - Strukturierung	65	7.5.4. Indextyp: Unique Index	81
6.2.2. Syntax: WITH Klausel	65	7.5.5. Indextyp: Volltextindex	81
6.3. Mengentheoretische Operationen	66	7.6. Sequenz	82
6.3.1. Mengentheoretische Operationen	66	7.6.1. Anlegen von Sequenzen	82
6.3.2. Union Klausel - Vereinigungsmenge	66	7.6.2. Sequenzen verwenden	82
6.3.3. Intersect Klausel - Durchschnittsmenge	67	8. SQL - Data Manipulation Language	84
6.3.4. Except Klausel - Differenzmenge	67	8.1. DML - Befehlssatz	84
6.4. Hierarchische Abfragen	68	8.1.1. DML Befehlssatz	84
6.4.1. Hierarchische Strukturen	68	8.2. insert Befehl	85
6.4.2. CONNECT BY Klausel - Rekursion	68	8.2.1. INSERT Befehl	85
6.4.3. Hierarchische Abfragen: Pseudospalten	69	8.2.2. INSERT SELECT Befehl	85
6.4.4. Hierarchische Abfragen: Funktionen	69	8.3. update Befehl	86
6.5. Quantoren	70	8.3.1. UPDATE Befehl	86
6.5.1. ALL Operator	70	8.4. delete Befehl	86
6.5.2. EXISTS Operator	70	8.4.1. DELETE Befehl	86
7. SQL - Data Definition Language		8.5. merge Befehl	87
7.1. Datenbankartefakte	72	8.5.1. Fallbeispiel: Verkauf	87
7.1.1. Datenbankartefakte	72	8.5.2. MERGE Befehl	87
7.2. Datenbankartefakt: Tabelle	73	8.5.3. Fallbeispiel: MERGE	88
7.2.1. Tabelle	73	9. SQL - Data Control Language	90
7.2.2. Tabellendefinition	73	9.1. DCL Befehle	90
7.2.3. Schlüsseldefinition	74	9.1.1. DCL Befehlssatz	90
7.2.4. DDL Befehl: create table	74	9.2. create user Befehl	90
7.2.5. Fallbeispiel: create table	74	9.2.1. Benutzerverwaltung	90
7.2.6. DDL Befehl: drop table	74	9.2.2. create User Befehl	91
7.2.7. DDL Befehl: truncate table	75	9.3. grant Befehl	91
		9.3.1. Privilegien zuordnen	91
		9.3.2. Syntax: grant Befehl	91
		9.3.3. grant Befehl	92
		10. PL/SQL - Grundlagen	94
		10.1. PL/SQL Grundlagen	94
		10.1.1. PL/SQL	94
		10.1.2. PL/SQL Sprachspezifikation	95
		10.1.3. Einsatzgebiete von PL/SQL	95

10.1.4. PL/SQL Engine	96	12.4.6. Fehlerbehandlung für forall	117
10.1.5. PL/SQL Block	97	12.5. Kursor	118
10.1.6. Fallbeispiel: PL/SQL Programm	97	12.5.1. Kursor	118
10.2. PL/SQL Block	98	12.5.2. Explizite Cursor	118
10.2.1. PL/SQL Block	98	12.5.3. Datenverarbeitung	119
10.2.2. Anonymer Block	99	12.5.4. BULK FETCH Anweisung	119
10.2.3. Benannte Blocktypen	99	13. PL/SQL - Blocktypen	120
11. PL/SQL - Datentypen	100	13.1. Blocktyp - Stored Procedure	120
11.1. Datentypen und Variablen	100	13.1.1. PL/SQL Prozedur	120
11.1.1. Definition von Variablen	100	13.1.2. Unterprogrammaufruf	121
11.1.2. Gültigkeit von Variablen	101	13.1.3. Parametertyp	121
11.1.3. Qualifier	101	13.1.4. Formen der Parameterzuweisung	122
11.1.4. Datentypen	102	13.1.5. Optionale Parameterzuweisung	122
11.2. Basisdatentypen	102	13.1.6. PL/SQL Funktion	122
11.2.1. Numerische Datentypen	102	13.2. Blocktyp - Package	123
11.2.2. Alphanumerische Datentypen	103	13.2.1. PL/SQL Package	123
11.3. Abgeleitete Datentypen	105	13.2.2. Packagespezifikation	123
11.3.1. Data Dictionary	105	13.2.3. Packageimplementierung	124
11.4. Strukturierte Datentypen	105	13.3. Blocktyp: Trigger	125
11.4.1. Record - Strukturierte Datentypen	105	13.3.1. PL/SQL Trigger	125
11.5. Tabellentypen	106	13.3.2. Anlegen von Triggern	125
11.5.1. Tabellentypen	106	13.3.3. Pseudorecords	126
11.5.2. Virtuelle Tabellen: Strukturierte Daten	106	13.3.4. Namenskonventionen	127
11.5.3. Virtuelle Tabellen	107	14. PL/SQL - SQL Funktionen	128
11.5.4. DQL: Virtuelle Tabellen	107	14.1. SQL Funktionen	128
12. PL/SQL - Befehle	108	14.1.1. SQL Funktionen	128
12.1. SQL Befehle in PLSQL	108	14.1.2. Programmieren von SQL Funktionen	129
12.1.1. DQL - Data Query Language	108	14.1.3. Fallbeispiel: Berechnung der Fakultät	129
12.1.2. DML - Data Manipulation Language	109	15. MongoDB - DDL	132
12.1.3. FORALL Anweisung	109	15.1. Datenkontainer	132
12.1.4. FORALL - RETURNING Klausel	110	15.1.1. Datenkontainer	132
12.1.5. DDL - Data Definition Language	110	15.2. Datenkontainer anlegen	133
12.2. Kontrollstrukturen	111	15.2.1. Datencontainer implizit anlegen	133
12.2.1. IF THEN Block	111	15.2.2. Collections explizit anlegen	133
12.2.2. Kontrollstruktur - Case Block	111	15.3. Collections und Dokumente	134
12.3. Schleifenkonstrukte	112	15.3.1. BSON Datenformat	134
12.3.1. Loop Block	112	15.3.2. Eingebettete Objekte	135
12.3.2. WHILE Block	113	15.4. Dokumentschema	136
12.3.3. FOR Block	113	15.4.1. BSON Schema	136
12.4. Fehlerbehandlung	114	15.4.2. Schemaattribute	136
12.4.1. Exceptionhandling	114	15.5. Schemaelemente	138
12.4.2. Exceptionhandler	114	15.5.1. Schemaelement: Eingebettete Objekte	138
12.4.3. Arten von Exceptions	115	15.5.2. Schemaelement: Enum	138
12.4.4. Anwendungsfehler	115	15.5.3. Schemaelement: Array	139
12.4.5. Unbenannte Fehler	117	15.5.4. Arrays von Objekten	139

15.5.5. Collection mit Schemavalidation	140	17.6. Daten löschen	159
15.6. Views erstellen	140	17.6.1. delete Befehl	159
15.6.1. Views			
15.7. Datencontainer verwalten	141	18. MongoDB - Aggregation von Daten	160
15.7.1. Container verwalten	141	18.1. Methoden und Verfahren	160
16. MongoDB - DQL	142	18.1.1. Aggregatgorithmen	160
16.1. Daten lesen	142	18.2. Abfragemethoden	161
16.1.1. find Befehl	142	18.2.1. Abfragemethode: <code>count()</code>	161
16.2. Kursormethoden	143	18.2.2. Abfragemethode: <code>distinct()</code>	161
16.2.1. Cursormethoden	143	18.2.3. Abfragemethode: <code>group()</code>	161
16.2.2. pretty() Methode	143	18.3. Aggregatframework	162
16.2.3. sort() Methode	144	18.3.1. Aggregationspipeline	162
16.2.4. limit() Methode	144	18.3.2. Aggregatframework	163
16.2.5. skip() Methode	144	18.3.3. Methode: <code>aggregate()</code>	163
16.2.6. forEach() Methode	145	18.3.4. Pipelinestufen	165
16.2.7. batchSize(), objsLeftInBatch() Methode	145	18.3.5. Fallbeispiel: Projects	165
16.3. Query Kriterien - Abfrageobjekt	146	18.4. Dokumentstufen	166
16.3.1. Abfrageobjekt	146	18.4.1. Dokumente filtern - <code>\$match</code>	166
16.3.2. Formen von Bedingungen	146	18.4.2. Dokumente sortieren - <code>\$sort</code>	166
16.3.3. Kurzformen von Abfragen	147	18.4.3. Dokumente entfernen - <code>\$skip</code>	166
16.3.4. Komplexe Bedingungen	147	18.4.4. Arrays auflösen - <code>\$unwind</code>	167
16.3.5. \$in Operator	148	18.4.5. Dokumentenstrom einschränken - <code>\$limit</code>	167
16.3.6. \$where Operator	148	18.4.6. Ergebnis abspeichern - <code>\$out</code>	167
16.4. Arrayabfragen	148	18.5. Strukturstufen	168
16.4.1. \$size Operator	148	18.5.1. Felder hinzufügen - <code>\$addFields</code>	168
16.4.2. \$all Operator	148	18.5.2. Felder projizieren - <code>\$project</code>	170
16.4.3. \$elemMatch Operator	149	18.5.3. Dokumentwurzel ändern - <code>\$replaceRoot</code>	171
16.5. Projektion	149	18.6. Beziehungsstufen	172
16.5.1. Projektionsobjekt	149	18.6.1. Relationen definieren - <code>\$lookup</code>	172
17. MongoDB - DML	150	18.6.2. Unterabfrage definieren - <code>\$lookup</code>	174
17.1. Daten einfügen - insert	150	18.7. Aggregatstufen	176
17.1.1. insertOne(), insertMany() Befehl	150	18.7.1. Aggregatoperatoren	176
17.2. Daten bearbeiten - update	151	18.7.2. Dokumente gruppieren - <code>\$group</code>	176
17.2.1. updateOne() Befehl	151	18.7.3. Fallbeispiel: Gruppierung	177
17.2.2. updateMany Befehl	153	18.7.4. Dokumente gruppieren - <code>\$bucket</code>	177
17.3. Strukturoperatoren	153	18.8. Expression	178
17.3.1. \$set Operator	153	18.8.1. Expressionsyntax	178
17.3.2. \$unset Operator	154	18.9. Arithmetische Operatoren	180
17.3.3. \$rename Operator	154	18.9.1. \$add, \$subtract, \$multiply Operator	180
17.4. Werteoperatoren	155	18.9.2. \$divide Operator	181
17.4.1. \$mul Operator	155	18.9.3. \$abs Operator	181
17.4.2. \$min Operator	155	18.10. Vergleichsoperatoren	181
17.4.3. \$inc Operator	156	18.10.1. Vergleichsoperatoren	181
17.5. Arrayoperatoren	156	18.11. Boolesche Operatoren	182
17.5.1. \$addToSet Operator	156	18.11.1. \$and, \$or Operatoren	182
17.5.2. \$push Operator	157	18.11.2. \$not Operator	182
17.5.3. \$pull, \$pullAll Operator	158	18.12. Kontrolloperatoren	182
17.5.4. \$pop Operator	158	18.12.1. \$cond Operator	182

18.12.2. \$switch Operator	183	19.6.1. XML Technologien	204
18.12.3. \$cmp Operator	184	19.6.2. XML Komponentenbaum	204
18.13. Arrayexpressions	184	19.6.3. Komponentenknoten	205
18.13.1. \$arrayElemAt Operator	184	19.6.4. Knotentypen	205
18.13.2. \$concatArrays Operator	185		
18.13.3. \$in Operator	185	20. Datenformat - XML XPath	206
18.13.4. \$map Operator	186	20.1. XPath - Konzepte	206
18.13.5. \$filter Operator	186	20.1.1. XPath Konzepte	206
18.13.6. \$reduce Operator	187	20.2. Pfadausdrücke in Kurzform	207
18.13.7. \$isArray Operator	187	20.2.1. Vereinfachte Pfadausdrücke	207
18.14. Aggregateexpressions	188	20.2.2. Auswahl von Elementknoten	207
18.14.1. Fallbeispiel: sales	188	20.2.3. Kontextknoten	208
18.14.2. \$addToSet, \$push Operator	188	20.2.4. Descendant Pfadoperator	208
18.14.3. \$min, \$max, \$avg Operatoren	189	20.2.5. Auswahl von Textknoten	208
18.14.4. \$sum Operator	189	20.2.6. Auswahl von Attributknoten	208
18.15. Referenzausdrücke	190	20.3. Lösungsobjekt	209
18.15.1. \$\$ Operator - Systemvariablen	190	20.3.1. Lösungsobjekt - Datentypen	209
18.15.2. \$expr Operator	190	20.4. Prädikat	209
18.15.3. \$let Operator	191	20.4.1. Definieren von Prädikaten	209
18.16. Mengenoperatoren	191	20.5. Pfadausdrücke in Standardform	210
18.16.1. \$setDifference Operator	191	20.5.1. Lokalisierungsstufen	210
18.16.2. \$setIntersection Operator	192	20.5.2. Achsenbezeichner	210
18.16.3. \$setUnion Operator	193	20.5.3. Knotenabfragen	211
18.16.4. \$setIsSubset Operator	193	20.5.4. Formen von XPath Ausdrücken	212
19. Datenformat - XML	196	20.6. XPath Funktionen	212
19.1. XML Grundlagen	196	20.6.1. Kategorien von Funktionen	212
19.1.1. Einsatzgebiete von XML	196	20.6.2. Knotenmengenfunktionen	212
19.1.2. Fallbeispiel: XML Dokumente	196	20.6.3. String Funktionen	213
19.1.3. Aufbau eines XML Dokuments	197	20.6.4. Logische Funktionen	214
19.2. XML Elemente	198	20.7. Fallbeispiel: XPath	215
19.2.1. Struktur eines XML	198	20.7.1. Datei: browser.xml	215
Dokuments	198	20.7.2. XPath Ausdrücke	215
19.2.2. Aufbau eines XML Elements	199	21. Datenformat - XML XSLT	216
19.2.3. Inhaltstypen von XML	199	21.1. XSLT Grundlagen	216
Elementen	199	21.1.1. XSLT Stylesheet	216
19.3. Attribute in Xml Elementen	200	21.1.2. Fallbeispiel: Hello World	217
19.3.1. Xml Attribute - Grundlagen	200	21.2. XSLT Transformationsprozess	217
19.3.2. Vergleich: Elemente vs.	200	21.2.1. Transformationsschritte	217
Attribute	200	21.3. XSLT Programm	218
19.4. Wohlgeformte XML Dokumente	201	21.3.1. XSLT Stylesheet	218
19.4.1. Wohlgeformete XML	201	21.3.2. Templateregel	218
Dokumente	201	21.3.3. XSLT Suchmuster	218
19.4.2. Wurzelement	201	21.3.4. Template Anweisungen	219
19.4.3. Elementtags	201	21.4. Deklarative Verarbeitung	220
19.4.4. Überlappung von XML	201	21.4.1. <xsl:apply-templates>	220
Elementen	201	Element	220
19.5. XML Namensräume	202	21.4.2. <xsl:template> Element	220
19.5.1. Namenskonflikte	202	21.4.3. <xsl:with-param> Element	221
19.5.2. XML Namensräume	202	21.4.4. xsl:mode Attribut	221
19.5.3. Namensraumdefinition	203	21.5. Prozedurale Verarbeitung	222
19.5.4. Standard Namensraum	203	21.5.1. <xsl:variable> Element	222
19.6. Logische Sicht	204	21.5.2. <xsl:value-of> Element	222

21.5.3. <xsl:if> Element	222	23.2.3. Einfache Datentypen	243
21.5.4. <xsl:choose> Element	223	Glossar	246
21.5.5. <xsl:for-each> Element	224	Index	249
21.5.6. <xsl:sort> Element	224		
21.6. Ausgabestream	225		
21.6.1. <xsl:element> Element	225		
21.6.2. <xsl:text> Element	225		
21.6.3. <xsl:attribute> Element	225		
21.6.4. <xsl:attribute-set> Element	226		
21.6.5. <xsl:copy-of> Element	226		
21.6.6. <xsl:output> Element	226		
21.7. Suchanfragen	227		
21.7.1. Auflösen von Templatekonflikten	227		
21.7.2. Default Templates	227		
21.7.3. Roottemplate	227		
21.7.4. Stringtemplate	229		
22. Datenformat - XML Schema	230		
22.1. Schema Grundlagen	230		
22.1.1. Inhaltsmodell	230		
22.1.2. Validierung	231		
22.1.3. XML Schema Regeln	231		
22.2. Struktur: XML Schemas	231		
22.2.1. Fallbeispiel: XML Schema	231		
22.2.2. Aufbau eines XML Schemas	231		
22.2.3. Kategorien von Datentypen	233		
22.3. Einfache Datentypen	233		
22.3.1. Einfache XML Elemente	233		
22.3.2. Benutzerdefinierte Datentypen	234		
22.3.3. Derivation by Restriction	234		
22.3.4. Derivation by Union	235		
22.4. Komplexe Datentypen	235		
22.4.1. Elemente mit komplexen Datentypen	235		
22.4.2. Strukturierung von XML Elementen	236		
22.4.3. Häufigkeitsindikatoren	237		
22.4.4. gemischter Inhalt	238		
22.5. Attribute	238		
22.5.1. Attributedefinitionen - Elemente mit einfacherem Inhalt	238		
22.5.2. Attributedefinitionen - Elemente mit komplexem Inhalt	239		
22.6. Sichtbarkeitskonzepte	239		
22.6.1. globale Datentypdeklarationen	239		
23. Datenformat - JSON	240		
23.1. JSON Datenformat	240		
23.1.1. JSON Grundlagen	240		
23.2. JSON Datentypen	241		
23.2.1. JSON Datentypen	241		
23.2.2. JSON Grundstrukturen	242		

Entwicklung relationaler Datenbanken

August 19, 2020

01

1. Datenbankentwurf

Datenmodellierung

01. Phasen des Datenbankentwurfs	10
02. Entity Relationship Modell	12
03. Relationale Modell	15
04. Normalisierung	19

1.1. Datenbankenentwurf ▾

Der **Datenbankentwurf** beschreibt alle Aufgaben und Tätigkeiten, die zur Erstellung eines physischen Datenbankschemas notwendig sind.

Ein **Datenbankschema** gibt die **physische Struktur** der Daten einer Datenbank vor.

1.1.1 Phasen des Datenbankentwurfs

Der Datenbankentwurf für relationale Datenbanken durchläuft in der Regel 4 Phasen.

► Vorgangsweise: Datenbankentwurf ▾

Externe Phase ▾

Informationsbeschaffung: In der externen Phase wird die Informationsstruktur des Datenmodells definiert. Dazu wird eine **Bestandaufnahme** der Anforderungen des Kunden durchgeführt.

Konzeptionelle Phase ▾

Semantische Modell: Ziel des konzeptionellen Entwurfs ist eine formalisierte Beschreibung der Anforderungen des Kunden.

Dazu werden alle für die Aufgabe essentiellen Gegenstände, Personen, Dienstleistungen und deren Beziehungen untereinander, erfasst und grafisch dargestellt.

Logische Phase: ▾

Logische Datenmodell: Das Ziel der logischen Phase ist die Übertragung des semantischen Modells in ein logisches Datenmodell. Das logische Datenmodell entspricht in seiner Form der **Struktur der Objekte** der Datenbank.

Physische Phase: ▾

Datenbankschema: In der physischen Phase wird aus dem logischen Modell die **physische Struktur** einer Datenbank generiert.



DB Entwicklung



1.1.2 Externe Phase

In der externen Phase wird die **Informationsstruktur** des Datenmodells definiert.

Ziel der externen Phase ist die **Informationsbeschaffung**. Die Anforderungen der Kunden an das Datenmodell werden gesammelt und strukturiert.

► Erklärung: Externe Phase ▾

- In der externen Phase erfolgt eine Bestandaufnahme der Anforderungen des Kunden. Es muss bestimmt werden, welche **Information** das Datenbanksystem generieren soll und welche Information dazu bereitgestellt werden muss.
- Das Ergebnis der externen Phase ist eine informelle **Beschreibung** der Geschäftsprozesse in Form eines Dokuments.

► Vorgangsweise: Externe Phase ▾

- In der externen Phase erfolgt eine Abbildung der **Anforderungen** auf eine Menge von **Objekten**.
- Betrachtet man z.B.: ein System Schule erkennt man gewisse Objekte bzw. Objektmengen wie das Fach Deutsch, ein Schüler namens Max oder einen Lehrer namens Franz.



1.1.3 Konzeptionelle Phase

Ziel der konzeptionellen Phase ist eine **graphische Repräsentation** der informellen Beschreibung der Anforderungen.

► Erklärung: Konzeptionelle Phase ▾

- Das semantische Modell dient zur **Veranschaulichung** des technischen Datenmodells im naiven Kontext des Geschäftsumfelds. Das Modell muß sowohl für Geschäftskunden als auch für Datenbankentwickler verständlich sein.
- Das Ergebnis der konzeptionellen Phase ist die Beschreibung der Geschäftsprozesse als **Entity Relationship Modell**.



1.1.4 Logische Phase

Das Ziel der logischen Phase ist die **Transformation** der Entitäten und Beziehungen des semantischen Modells in eine **Tabellenstruktur**.

► Erklärung: Logische Phase ▾

- Das semantische Modell unterscheidet zwei Strukturelemente: Entitäten und Beziehungen. Das logische Modell beschreibt die Welt hingegen als eine Menge von Tabellen.
- Der logische Entwurf überführt die Entitäten und Beziehungen des semantischen Modells in eine Tabellenstruktur.

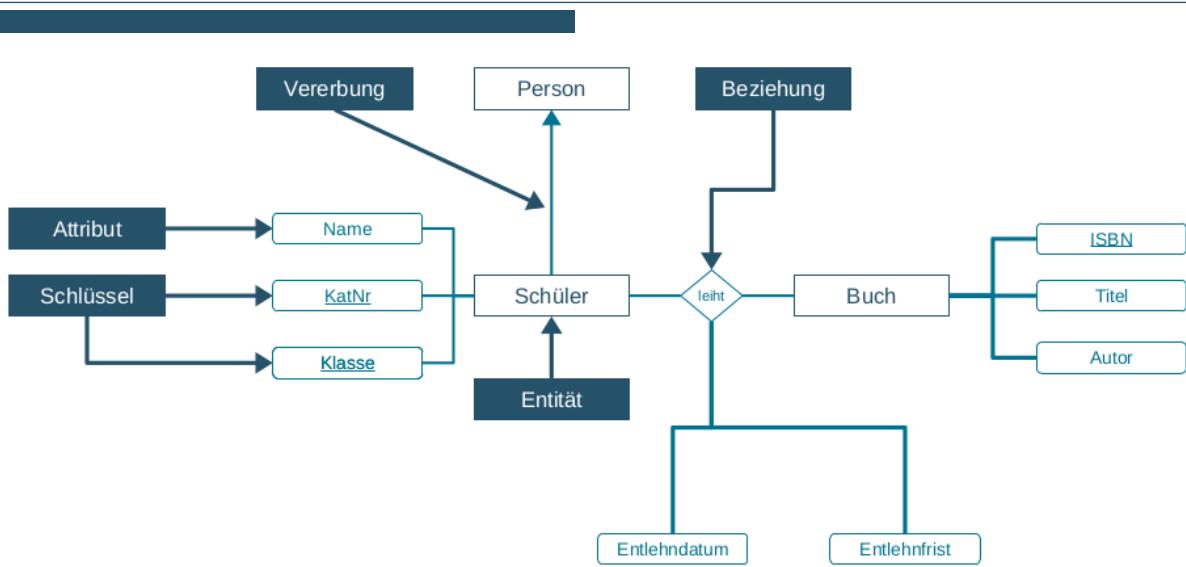


Abbildung 1. Entity Relationship Modell

- Zur Abbildung des Semantischen Modells in eine Tabellenstruktur, wird eine Reihe von **Transformationsregeln** auf die Entitäten und Beziehungen des ER Diagramms angewandt.
- Das Ergebnis der logischen Phase ist die Beschreibung der Geschäftsprozesse als **Relationales Modell**.

1.2. Entity Relationship Modell ▾



ER Modell ▾

Ein Modell beschreibt ein zweckorientiertes vereinfachtes **Abbild** der **Wirklichkeit**.

Zur Beschreibung des semantischen Modells einer relationalen Datenbank, wird das **Entity Relationship Modell** verwendet.

1.1.5 Physische Phase

In der physischen Phase wird aus dem logischen Modell die physische Struktur der Datenbank generiert.

► Analyse: Physische Phase ▾

- Das Ergebnis des physischen Entwurfs ist eine Folge von **Datenbankbefehlen** zum Anlegen des physischen Schemas des Datenmodells.
- Dazu müssen alle notwendigen Datentypen, Wertebereiche, Relationen bzw. Sichten für das Schema definiert werden.
- Das Ziel des physischen Entwurfs ist die Überführung des logischen Modells in eine physisches Datenbankschema.
- Die physische Phase ist der abschließende Schritt im Datenbankentwurfsprozesses.

1.2.1 ER Modell

Das ER Modell beschreibt die grundlegende **Entitäts-** bzw. **Beziehungsstruktur** des semantischen Modells.

► Erklärung: Datenmodell ▾

- Im ER Modell werden alle essentiellen Personen, Dienstleistungen bzw. Gegenstände und deren Beziehungen untereinander graphisch erfasst.
- Betrachtet man z.B.: ein System Schule, erkennt man **Entitäten** wie das Fach Mathematik, einen Schüler Namens Kurt bzw. einen Lehrer namens Hase.
- Das ER Modell unterscheidet dabei zwei grundlegende Elemente: **Entitäten** und **Beziehungen**.

ER Element	Beschreibung	Seite
Entität	Als Entität wird in der Datenmodellierung ein datentragendes Objekt bezeichnet.	13
Entitätstyp	Entitätstypen beschreiben Gruppen von Entitäten, die aufgrund gleicher oder ähnlicher Attributwerte zusammengefaßt werden. Entitätstypen werden im ER Modell durch Rechtecke symbolisiert, zugehörige Attribute durch Ellipsen.	13
Schlüssel	Der Schlüssel einer Entität beschreibt eine Attributkombination die für jede Entität eines Entitätstyps eindeutig ist.	13
Beziehung	Eine Beziehung beschreibt die Art und Weise in der 2 Entitäten miteinander in Verbindung stehen. Dabei handelt es sich in der Regel um eine Zugehörigkeit zu einer Aktion oder einem Sachverhalt. Beziehungen werden im ER Modell durch Rauten symbolisiert.	13

Abbildung 2. Elemente des ER Modells

1.2.2 Entität

Als Entität wird in der Datenmodellierung ein **datentragendes Objekt** bezeichnet.

Eine Entität kann dabei ein materielles oder immateriell, konkretes oder abstraktes Konzept beschreiben.

► **Erklärung: Entität ▾**

- Datentragende Objekte werden durch **Attribute** näher bestimmt. Attribute beschreiben die Merkmale bzw. Eigenschaften einer Entität.
- Zur **thematischen Abgrenzung** werden Entitäten mit gleichen oder ähnlichen Attributausprägungen im Modell zu **Entitätstypen** zusammengefaßt.
- Das ER Modell abstrahiert die Entitätstypen sowie deren Beziehungen in einer **graphischen Repräsentation**. In der Datenbank entsprechen die Entitätstypen den Tabellen, die Tabellenzeilen den Entitäten.
- Der Schlüssel einer Entität beschreibt eine **Attributkombination** die für jede Entität eines Entitätstyps eine andere Ausprägung besitzt.

1.2.3 Beziehung

Das ER Modell stellt im Kontext des Datenbankentwurfs einen **relevanten Ausschnitt** der realen Welt dar.

► **Erklärung: Beziehung ▾**

- Im Modell werden alle essentiellen Objekte und deren Beziehungen untereinander graphisch erfaßt.
- Eine Beziehung beschreibt die Form in der 2 Entitäten miteinander in **Verbindung** stehen. Dabei handelt es sich in der Regel um die Zugehörigkeit zu einer Aktion oder einem Sachverhalt.
- Die Form einer Beziehung wird dabei durch die **Kardinalität** der Beziehung definiert. Die Kardinalität einer Beziehung beschreibt in welcher Form die Entitäten einer Beziehung untereinander assoziiert werden.
- Ein Schüler kann z.B.: ein oder mehrere Bücher ausleihen. Ein Buch kann im Laufe der Zeit wiederum von mehreren Schülern ausgeborgt worden sein. Bei dieser Art von Beziehung handelt es sich um eine n:m Relation weil n Objekte des einen Entitätstyps mit m Objekten des anderen Entitätstyps in Relation stehen.
- Generell werden 3 Formen von Beziehungen unterschieden: 1:1, 1:n und n:m.
- Beziehungen werden im ER Modell durch Rauten symbolisiert, wobei die Kardinalität der Beziehung bei der jeweiligen Entität notiert wird.



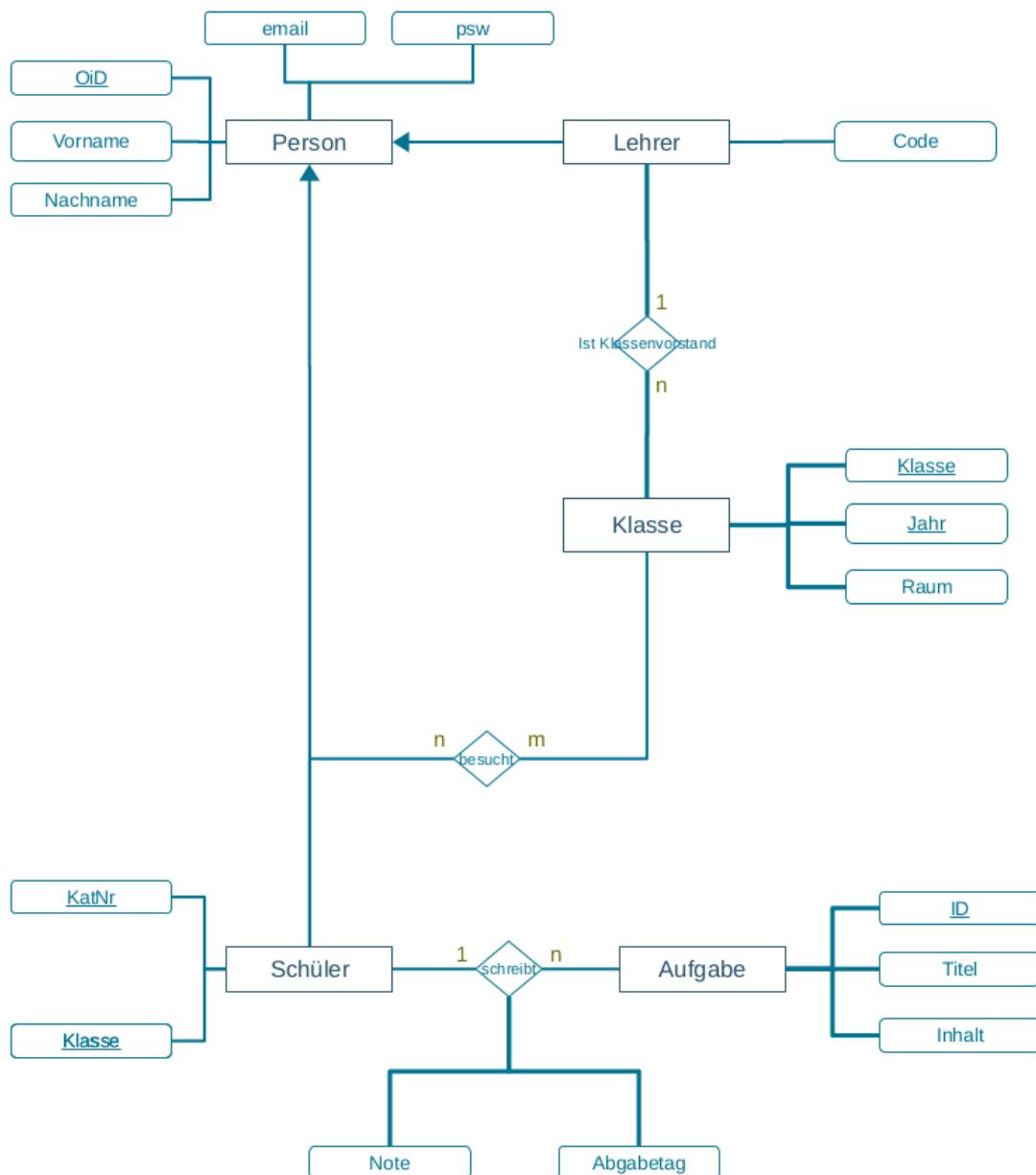


Abbildung 3. ER Modell: Kardinalität

Kardinalität	Beschreibung	Seite
1..1	Bei der 1..1 Beziehung steht ein Objekt des ersten Entitätstyps mit genau einem Objekt des anderen Entitätstyps in Beziehung. z.B.: Ein Schüler hat einen Schullaptop. Ein Schullaptop ist immer genau einem Schüler zugeordnet.	12
1..n	Bei der 1..n Beziehung steht ein Objekt des ersten Entitätstyps mit einem oder mehreren Objekten des anderen Entitätstyps in Beziehung. Ein Schüler verfaßt Aufgaben. Eine Aufgabe ist genau einem Schüler zugeordnet.	12
n..m	Bei der n..m Beziehung steht ein Objekt des ersten Entitätstyps mit einem oder mehreren Objekten des anderen Entitätstyps in Beziehung. Die Beziehung gilt dabei für beide Entitätstypen. z.B.: Eine Schularbeit wird von den Schülern einer Klasse geschrieben. Schüler schreiben mehrere Schularbeiten	12

Abbildung 4. ER Modell: Kardinalität

1.3. Relationale Modell

Das Ziel des logischen Entwurfs ist die Transformation der Entitäten und Beziehungen des semantischen Modells in eine **Tabellenstruktur**.

1.3.1 Relationale Modell

Das relationale Modell basiert auf einer **tabellarischen Organisation** der Daten.

► Erklärung: Relationale Modell ▾

- Das Relationale Modell kennt im Gegensatz zum ER Modell nur ein einzelnes **Strukturelement**: die Tabelle.
- Im Zuge des relationalen Entwurfs wird die Struktur der Tabellen des Datenbankschemas definiert. Dazu wird die Anzahl, der Name und der **Typ** der Spalten einer Tabelle bestimmt.
- Zur **logischen Organisation** der Daten einer Tabelle, wird für jeden Tabelleneintrag ein Schlüssel definiert.
- Dabei beschreibt ein Schlüssel eine **Spaltenkombination**, die für jeden Datensatz einer Tabelle eine unterschiedliche Ausprägung besitzt. Der Schlüssel kann dabei aus einer oder mehreren Spalten bestehen.
- Im relationalen Entwurf werden Tabellenschlüssel auch als **Primaerschlüssel** bezeichnet.

1.3.2 Beziehungen zwischen Tabellen

Eine Beziehung beschreibt die Art und Weise in der **Tabelleneinträge** miteinander in **Relation** stehen.

► Erklärung: Beziehungen ▾

- Zur Definition einer Beziehung im relationalen Modell werden die **Schlüssel** der **Tabelleneinträge** in Relation gesetzt. Dazu wird ein Datensatz um den Primärschlüssel eines anderen Eintrags ergänzt.
- Aus technischer Sicht wird die Tabellenstruktur des datentragenden Objekts, um den Primärschlüssels der gewünschten Entität erweitert.
- Die neu definierten Tabellenspalten werden auch als **Fremdschlüsselspalten** bezeichnet, weil sie auf den Primärschlüssel einer anderen Tabelle beziehen.
- Eine Beziehung besteht dabei für alle Tabelleneinträge die für Primär- und Fremdschlüsselspalten übereinstimmende Einträge besitzen.
- Die **Kardinalität** einer **Beziehung** bestimmt in welcher Form der Fremdschlüssel in die Tabellenstruktur des Datenmodells integriert wird.
- Das relationale Modell definiert dabei eine Reihe von **Transformationsregeln**, um die Elemente des ER Modells in eine Tabellenstruktur zu zwingen.
- Mit dem Relationalen Modell liegt der abzubildende Datenbestand in seiner finalen Form vor. Das Relationalen Modell beschreibt in dieser Form die Struktur des physischen Datenbankschemas.

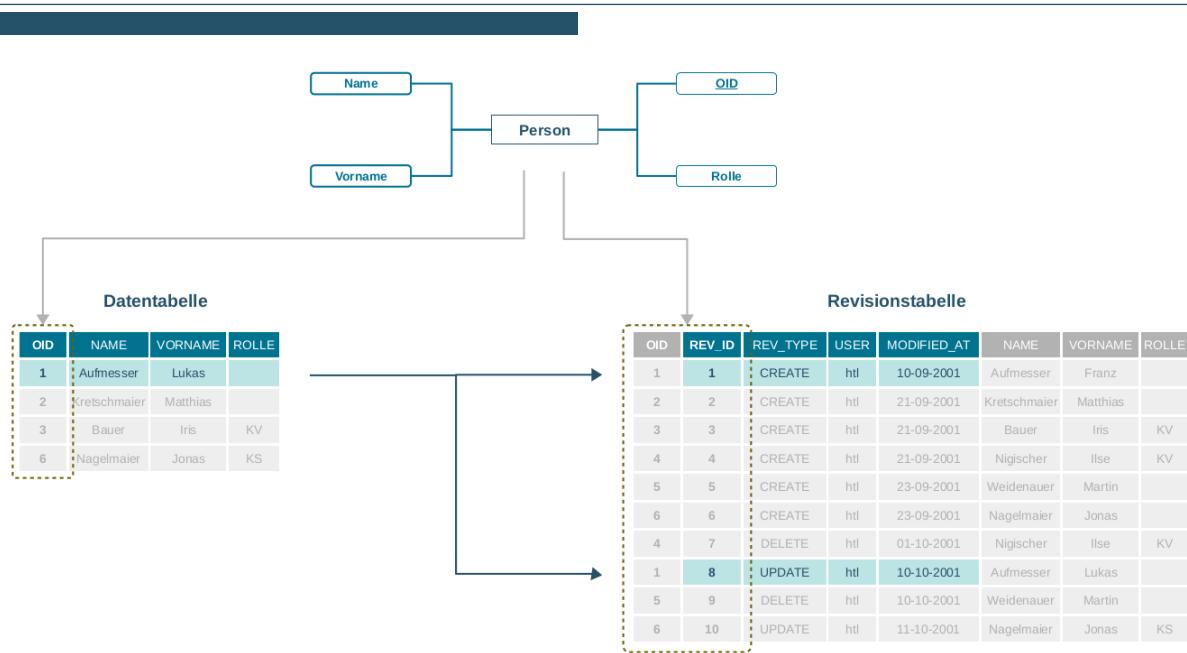


Abbildung 5. Datentabelle vs. Revisionstabelle

1.3.3 Transformationsregel: Entität

Das Relationale Modell kennt im Gegensatz zum ER Modell nur ein einzelnes Strukturelement: die **Tabelle**.

► Transformation: Entitäten ▾

- Das Ziel des logischen Entwurfs ist die Transformation der Entitäten und Beziehungen des semantischen Datenmodells in eine **Tabellenstruktur**.
- Die **Entitäten** desselben Entitätstyps werden dazu zeilenweise in einer **Tabelle** gesammelt.
- Im relationalen Modell werden dazu die Anzahl, die Namen und die Typen der Spalten der Tabellen definiert. Das relationale Modell unterscheidet dabei mehrere **Formen von Tabellen**.
- Datentragende Tabellen werden im Modell als Datentabellen bezeichnet. **Datentabellen** speichern den Grossteil der Daten eines Geschäftsfalls.
- Zur Protokollierung von Änderungen im Datenbestand, werden den Datentabellen **Revisionstabellen** beigestellt.

1.3.4 Transformationsregel: 1..1 Beziehung

Beziehungen beschreiben die Art und Weise in der Tabelleneinträge miteinander in Relation stehen.

► Transformation: 1..1 Beziehung ▾

- Bei einer 1:1 Beziehung steht ein Objekt des ersten Entitätstyps mit genau einem Objekt des anderen Entitätstyps in Beziehung.
- Zur Modellierung einer 1:1 Beziehung, wird eine der **Tabellen**, um den **Schlüssel** der anderen Entität erweitert.
- Welche der Tabellen um Schlüsselspalten erweitert werden soll, liegt dabei im Bemessen des Entwicklers. Handelt es sich bei einer Relation z.B.: um eine **optionale Beziehung** muss die abhängige Entität, um den Schlüssel erweitert werden.



Tabellentyp	Beschreibung	Seite
Datentabelle	Als Datentabellen werden die datentragenden Tabellen eines Modells bezeichnet. Datentabellen halten in der Regel den Grossteil der Daten eines Geschäftsfalls. Benannt sind Datentabellen nach den Entitätstypen auf die sie sich beziehen. z.B.: employees	16
Attributabelle	Konzeptionell können Attributtabellen als Aufzählungstypen verstanden werden. Die zulässigen Werte des Aufzählungstyps bilden dabei den Datenbestand der Tabelle. Benannt sind Attributtabellen nach dem Aufzählungstyp auf den sie sich beziehen. Dem Namen wird zusätzlich ein e vorangestellt. verwendet. z.B.:e_project_types	12
Schlüsseltabelle	m:n Beziehungen werden im relationalen Entwurf mit der Hilfe von Schlüsseltabellen modelliert. Dazu speichert die Schlüsseltabelle für jede an der m:n Relation beteiligte Entität deren Schlüssel. Der Schlüssel der Schlüsseltabelle selbst, ist eine Kombination aller eingetragenen Schlüssel. Benannt sind Schlüsseltabelle nach den Name aller an der Relation beteiligen Entitäten, getrennt durch einen Unterstrich. Dem Namen wird zusätzlich ein jt nachgestellt. z.B.: project_employees_jt	12
Revisionstabelle	Revisionstabellen werden zur Protokollierung von Änderungen am Datenbestand verwendet. Benannt sind Revisionstabellen nach den Entitätstypen auf die sie sich beziehen. Dem Namen wird zusätzlich ein v vorangestellt. z.B.: v_employees	16
Sammeltabelle	Vererbungsbeziehungen zwischen Entitäten werden im Relationalen Modell mit der Hilfe von Sammeltabellen modelliert. Alle Objekte eines Entitätstyps oder davon abgeleiteter Entitätstypen werden in einer einzelnen Tabelle gesammelt. Die Tabellenstruktur einer Sammeltabelle setzt sich aus allen möglichen Attributen der einzutragenden Objekten zusammen. Benannt werden werden Sammeltabellen nach dem Entitätstyp der Basisentität. Dem Namen wird zusätzlich ein st nachgestellt. z.B.: employees_st	16
Basistabelle, Kind-tabelle	Zur Modellierung von Vererbungsbeziehungen können die Werte eines Objekts auf eine Basis- bzw. Kindtabelle verteilt werden. Konzeptionell entspricht der Vorgang der Modellierung von Basis- und Kindklassen der OOP. Benannt werden die Tabellen nach den Entitätstypen auf die sie sich beziehen. Basistabellen wird zusätzlich ein Kürzel bt nachgestellt.	

Abbildung 6. Relationales Modell: Tabellentypen

1.3.5 Transformationsregel: 1..n Beziehung

Beziehungen beschreiben die Art und Weise in der Tabelleneinträge miteinander in Relation stehen.

► Transformation: 1..n Beziehung ▾

- Bei der 1..n Beziehung steht ein Objekt des ersten Entitätstyps mit einem oder mehreren Objekten des anderen Entitätstyps in Beziehung.
- In der Gegenrichtung wird jeder Entität des anderen Entitätstyps genau eine Entität des ursprünglichen Entitätstyps zugeordnet.
- Zur Modellierung einer 1:n Beziehung, wird die **Tabellestruktur** der abhängigen **Entität**, um den Schlüssel der Zielentität erweitert.

1.3.6 Transformationsregel: n..m Beziehung

Beziehungen beschreiben die Art und Weise in der Tabelleneinträge miteinander in Relation stehen.

► Transformation: n..m Relation ▾

- Bei einer n:m Beziehung kann ein Objekt des einen Entitätstyps mit beliebig vielen Objekten des anderen Entitätstyps in Beziehung gesetzt werden.
- Zur Modellierung einer n:m Beziehung werden die Schlüsselwerte der Entitäten in einer eigenen Tabelle in Beziehung gesetzt.
- Zusätzliche Attribute der Beziehung werden ebenfalls in der **Schlüsseltabelle** gespeichert.

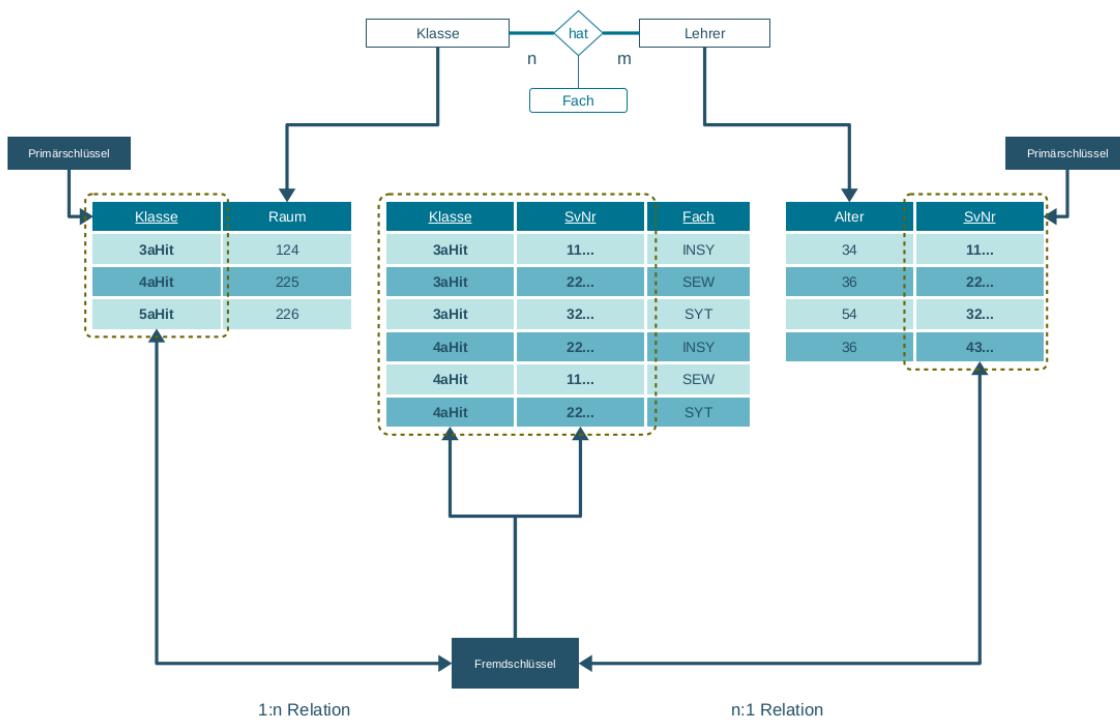
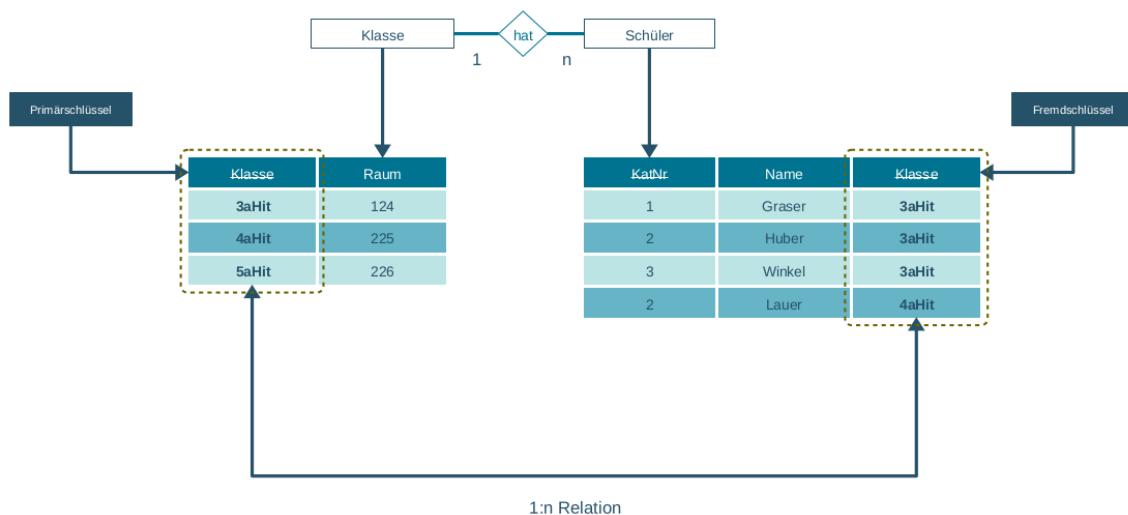


Abbildung 7. Beziehungen: 1..n, n..m

1.3.7 Vererbungskonzept

Zur thematischen Abgrenzung logischer Konzepte werden Entitäten mit gleichen oder ähnlichen Attributen zu **Entitätstypen** zusammengefaßt.

Entitätstypen abstrahieren Geschäftsprozesse in Form einfacher **Netzstrukturen**.

► Erklärung: Vererbungskonzept ▾

- Konzeptionell kann durch die Definition von **Vererbungsbeziehungen** eine weitere strukturelle Vereinfachung erreicht werden.
- Eine **Vererbungsbeziehung** drückt aus, dass 2 Entitätstypen zwar nicht gleich sind, aber eine große Ähnlichkeit haben. Die Vererbung kann dabei zwischen 2 oder mehreren Entitätstypen definiert werden.
- Bei der Vererbung wird dazu eine **Basisentität** bestimmt, die ihre Attribute an alle Subentitäten weitergibt, die von ihr erben.
- Gemeinsame Attribute müssen damit nur einmal modelliert werden.



1.3.8 Transformationsregel: Vererbung

Konzeptionell können **Vererbungsbeziehungen** im Relationalen Modell nicht definiert werden.

► Transformation: Vererbung ▾

- Der relationale Entwurf abstrahiert 2 Formen zur Abbildung von Vererbungsbeziehungen.
- **Single Table Inheritance:** Die Objekte der Basisentität und aller Subentitäten werden gesammelt in einer einzelnen Tabelle eingetragen. Die Tabellenstruktur der **Sammeltabelle** setzt sich dabei aus allen Attribute der einzelnen Objekte zusammen.
- **Joined Table Inheritance:** Die Daten der Objekte werden verteilt auf mehrere Tabellen eingetragen. Die Werte der Basisentität werden dazu in einer eigenen **Basistabelle** gesammelt. Die restlichen Werte werden verteilt auf andere Tabellen gespeichert. Für jeden Subtyp definiert das Modell dazu eine eigene Tabelle.



1.4. Normalisierung

Als **Normalisierung** wird ein Ansatz des Datenbankdesigns zur Vermeidung **redundanter Daten** bezeichnet.

1.4.1 Datenredundanz

Nominell werden Mehrfacheinträge derselben Daten in einem einzelnen Datenbankschema als **Datenredundanz** beschrieben.

Redundanz beschreibt in diesem Zusammenhang die mehrfache Speicherung derselben Information in einer Tabelle.

► Erklärung: Datenredundanz ▾

- **Datenredundanz** verursacht bei der Verarbeitung von Daten **Anomalien**. Solche Inkonsistenzen treten z.B.: dann auf wenn nur eine der Kopien eines redundanten Datensatzes geändert wird.
- Um Datenredundanz zu verhindern werden die Tabellen eines Datenbankschemas **normalisiert**.
- Unter dem Begriff **Normalisierung** wird die Strukturierung einer Datenbasis im Sinne eines konsistenten Verhaltens verstanden.



Normalisierung ▾

Bei der Normalisierung handelt es sich um eine **Strategie**, Redundanzen aus dem Datenbestand eines Datenschemas zu beseitigen.

► Erklärung: Normalisierung ▾

- **Normalisierung** beschreibt dabei den Prozess der **Restrukturierung** und Neuorganisation relationaler Datenbankschema.
- Dazu wird eine Reihe von **Regeln** definiert, die fortlaufend gegen die Datenbasis eines Schemas angewandt werden.
- Im Zuge der Normalisierung erfolgt eine wiederholte **Neustrukturierung** der Tabellen bis keine Datenredundanzen mehr vorhanden sind.
- Für die Normalisierung relationaler Datenmodelle werden 3 grundlegende **Normalformen** definiert. Die Normalformen bauen dabei aufeinander auf. Ist die nte erfüllt sind auch alle darunterliegenden erfüllt.



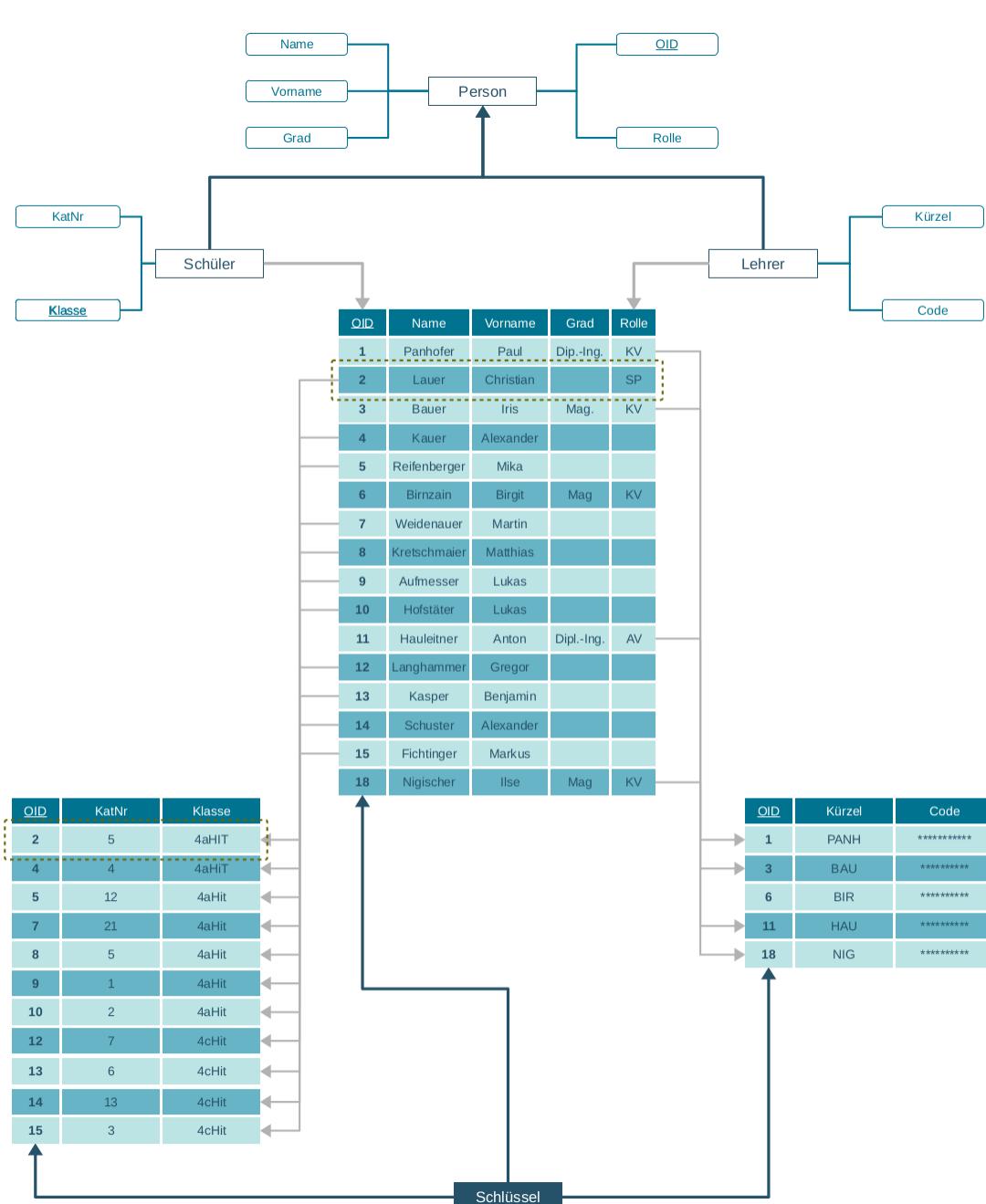


Abbildung 8. Vererbung: Basistabelle, Kindtabelle

Normalform	Beschreibung	Seite
1.Normalform	Eine Tabelle befindet sich in erster Normalform, wenn jede Information innerhalb einer Tabelle eine eigene Tabellenspalte bekommt und zusammenhängende Informationen, wie z.B. die Postleitzahl und der Ort , nicht in einer Tabellenspalte vorliegen.	21
2.Normalform	Tabellen in zweiter Normalform müssen bereits in erster Normalform vorliegen. Zusätzlich dürfen die Datensätze einer Tabelle immer nur einen Sachverhalt abbilden. Abstrahieren die Daten einer Tabelle mehrere Sachverhalte müssen sie in zusätzliche Tabellen zerlegt werden.	21
3.Normalform	Die dritte Normalform gilt als eingehalten, wenn die zweite Normalform erfüllt ist und keine indirekten Abhängigkeiten innerhalb einer Tabelle mehr bestehen.	13

Abbildung 9. Elemente des ER Modells

1.4.2 Erste Normalform



1.4.3 Zweite Normalform



In der Datenbankentwicklung ist die 3 Normalform ausreichend, um die perfekte Balance aus Redundanz, Performance und Flexibilität



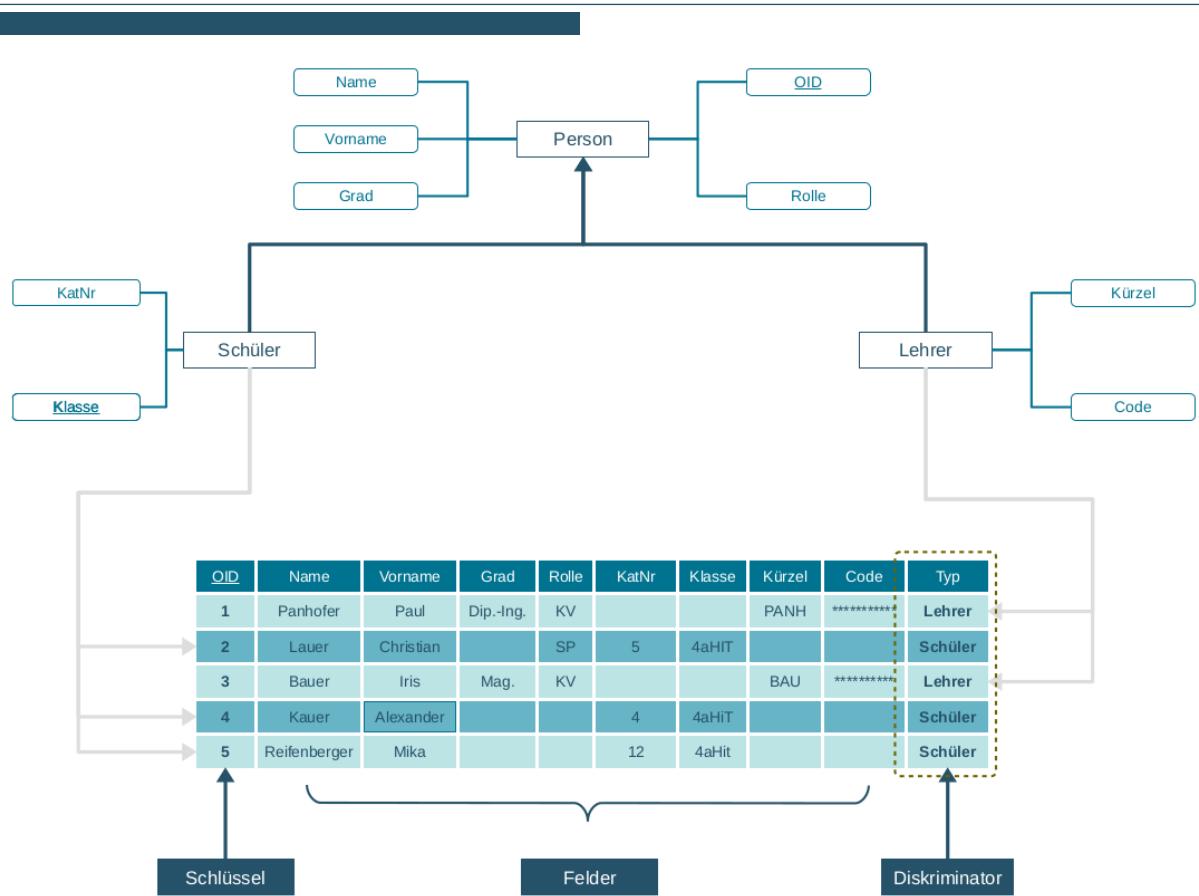


Abbildung 10. Vererbung: Sammeltabelle

1.4.4 Dritte Normalform

SQL - Oracle 19c

August 19, 2020

2. SQL - Data Query Lanuguage

01

SQL Grundlagen

01. SQL Grundlagen	24
02. Projektion - Select Klausel	26
03. Kondition - Case Klausel	30
04. Restriktion - Where Klausel	31
05. Sortierung - Order By Klausel	34
06. Limitierung - Fetch Klausel	35
07. Paginierung - Offset Klausel	35

2.1. SQL Grundlagen

SQL ist eine **Programmiersprache**, mit der relationale Datenbanken erstellt bzw. bearbeitet, sowie Datenbestände abgefragt werden können.

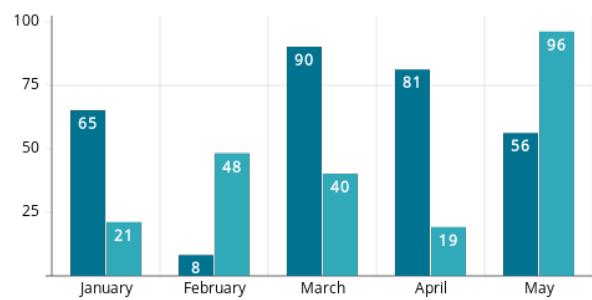
2.1.1 SQL - Structured Query Language

SQL als Programmiersprache folgt dem **deklarativen Programmierparadigma**.

► Erklärung: Deklarative Programmierung ▾

- Während ein prozedurales Programm den Prozess beschreibt, mit dem ein bestimmtes Ergebnis erreicht werden soll, beschränkt sich die deklarative Programmierung auf die Definition eines gewünschten **Endzustands**.

Es ist die Aufgabe der **Datenbankengine** eine entsprechende Lösung zu berechnen.



► Erklärung: SQL Syntax ▾

- SQL als Programmiersprache zeichnet sich durch eine **einfache Syntax** aus. Die Sprache selbst besteht dabei im Wesentlichen aus englischen Sprachelementen.
- SQL als Sprache ist **standardisiert** und wird plattformübergreifend in relationalen Datenbanksystemen eingesetzt. Es existieren jedoch unterschiedliche SQL Dialekte, die eine vollständige Kompatibilität des Standards verhindern.



2.1.2 Kategorien von SQL Befehlen

Der **SQL Befehlssatz** ist in unterschiedliche **Kategorien** unterteilt.

► Auflistung: Kategorien von SQL Befehlen ▾

- **Data Manipulation Language:** Befehle zum Verarbeiten von Daten.
- **Data Definition Language:** Befehle zum Definieren der Struktur einer Datenbank.
- **Data Query Language:** Befehle zum Auslesen von Daten aus einer Datenbank.

► Begriff: DML Befehlsatz ▾

DML Befehle werden zum **Bearbeiten**, **Einfügen** und **Löschen** von Daten verwendet.

```

1 -- Einfuegen von Daten
2 INSERT INTO ...
3 -- Veraendern von Daten
4 UPDATE EMPLOYEES ...
5 -- Loeschen von Daten
6 DELETE FROM ...

```

► Begriff: DDL Befehlsatz ▾

DDL Befehle werden zur **Definition** des **Schemas¹** einer Datenbank verwendet.

```

1 -- Anlegen von Tabellen
2 CREATE TABLE ...
3 -- Aendern von Tabellen
4 ALTER TABLE ...

```

► Begriff: DCL Befehlsatz ▾

DCL Befehle werden zum **Verwalten** von **Rechten** bzw. zur **Steuerung** von **Transaktionen** verwendet.

```

1 -- Weitergabe von Rechten fuer den
2 -- Zugriff auf Tabellen
3 GRANT ON TO ...
4 -- Beenden einer Transaktion
5 COMMIT ...

```

► Begriff: DQL Befehlsatz ▾

DQL Befehle werden zum **Auslesen** von Daten aus der Datenbank verwendet.

```

1 -- Daten aus der Datenbank lesen
2 SELECT EMPLOYEE_ID FROM ...

```

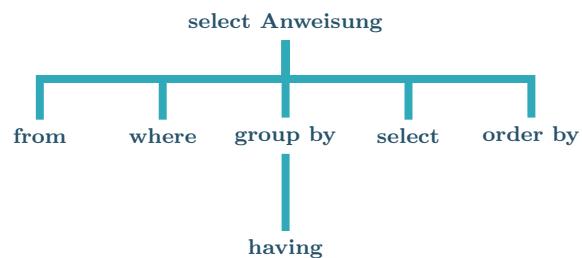
2.1.3 DQL - Select Anweisung

Die **select** Anweisung wird verwendet um Daten aus einer Datenbank zu lesen.



► Erklärung: Select Anweisung ▾

- Strukturell besteht eine **select** Anweisung aus mehreren Teilen, den **Klauseln**. Dabei sind 2 der Klauseln für Abfragen obligatorisch: die **select** Klausel und die **from** Klausel. Alle anderen Klauseln sind **optional**.
- Welche Daten, zu welchem Zeitpunkt, auf welche Weise verarbeitet werden, wird durch die entsprechende **Klausel** bestimmt.
- **Auswertungsreihenfolge von SQL Klauseln:**



► Auflistung: Klauseln der Select Anweisung ▾

From Klausel - Projektion ▾

In der from Klausel wird die **Datenbasis** einer Abfrage festgelegt.

Die Klausel ist **obligatorisch**.

Where Klausel - Restriktion ▾

Die where Klausel wird zur **Filterung** der Datensätze einer Abfrage verwendet.

Die where Klausel ist **optional**.

Group By - Reporting ▾

Mit der group by Klausel kann die **Struktur** der Daten einer Abfrage verändert werden.

Die group by Klausel ist **optional**.

¹ Beschreibung der Struktur der Daten in der Datenbank



Select Klausel - Projektion ▾

In der select Klausel wird definiert welcher Teil der Daten in der **Ergebnismenge** einer Abfrage enthalten sein soll.

Die Klausel ist **obligatorisch**.



Order By Klausel - Sortierung ▾

Mit der order By Klausel kann eine **Sortierung** der Datensätze des Ergebnisses einer Abfrage definiert werden.

Die order by Klausel ist **optional**.



Fetch Klausel - Limitierung ▾

Mit der fetch Klausel kann das Ergebnis einer Abfrage auf eine bestimmte Zahl von Datensätzen **eingeschränkt** werden.

Die fetch Klausel ist **optional**.



Offset Klausel - Ausschluss ▾

Mit der offset Klausel werden die ersten n Datensätze des Ergebnisses einer Abfrage ignoriert .

Die offset Klausel ist **optional**.

2.2. Select Klausel - Projektion ▾

2.2.1 Select Klausel

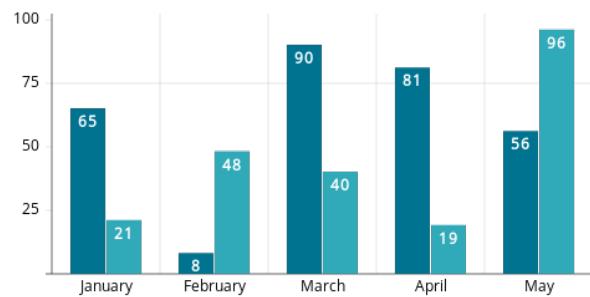
In der **select Klausel** werden jene **Spalten** definiert, deren Werte die Ergebnismenge einer Abfrage bilden.

► Query: Select Anweisung ▾

```
1 -- -----
2 -- Select Anweisung
3 -- -----
4 SELECT * FROM EMPLOYEES;
```

► Analyse: Select Anweisung ▾

- Hier steht gewissermassen der Satz: Wähle alle Spalten der Tabelle `employees` und zeige die Daten als Ergebnis an.
- Am Ende der select Anweisung steht ein **Semikolon**. Das Semikolon ist dabei nicht Teil der select Anweisung, sondern ein **Steuerzeichen** der Datenbank.



► Query: Projektion von Daten ▾

```
1 -- -----
2 -- Projektion von Daten
3 -- -----
4 SELECT DISTINCT FIRST_NAME, LAST_NAME
5 FROM EMPLOYEES;
```

► Erklärung: Projektion von Daten ▾

- Soll für eine SQL Abfrage die **Spaltenauswahl** der Ergebnismenge eingeschränkt werden, wird eine durch Komma getrennte Liste von Spaltenbezeichnern in der select Klausel angeführt.
- Mit dem `distinct` Schlüsselwort kann die Datenbankengine angehalten werden, Duplikate aus dem Abfrageergebnis zu entfernen.

2.2.2 Wiederholte Spaltenausgabe

In der select Klausel kann die Angabe von Spaltenbezeichnern in beliebiger Reihenfolge bzw. in beliebiger Zahl erfolgen.

► Erklärung: Wiederholte Spaltenausgabe ▾

- In einer select Klausel kann der gleiche Spaltenbezeichner mehrfach angegeben werden.
- Die Ausgabe der Daten erfolgt entsprechend der in der Klausel definierten **Reihenfolge** der Spaltenbezeichner.

► Query: Wiederholte Spaltenausgabe ▾

```

1  -- -----
2  -- Wiederholte Spaltenausgabe
3  --
4  SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID,
5      SALARY,
6      SALARY
7  FROM EMPLOYEES;
8
9  SELECT DEPARTMENT_ID, DEPARTMENT_NAME,
10     DEPARTMENT_HEAD,
11     DEPARTMENT_CODE,
12     DEPARTMENT_NAME
13  FROM DEPARTMENTS;
```

2.2.3 Spaltenwerte bearbeiten

Daten können vor ihrer Ausgabe in der select Klausel adaptiert werden.

► Query: Spaltenwerten algebraisch bearbeiten ▾

```

1  --
2  -- Mit Spaltenwerten rechnen
3  --
4  SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME,
5      SALARY * 1.03,
6      SALARY * 1.10,
7      SALARY - (SALARY * 0.8)
8  FROM EMPLOYEES;
```

► Analyse: Spaltenwerte bearbeiten ▾

- Durch die Anweisung wird zusätzlich zum Gehalt das um 3% erhöhte Gehalt ausgegeben.
- Alphanumerische Werte können im Rahmen einer Projektion nicht algebraisch verändert werden.

2.2.4 Spaltenwerte verknüpfen

Zur Bearbeitung alphanumerischer Werte stellt SQL den || Operator zur Verfügung.

► Erklärung: Spaltenwerte verknüpfen ▾

- SQL verwendet den || Operator um alphanumerische Werte miteinander zu verknüpfen.
- Der || Operator kann dabei beliebig oft in einem SQL Ausdruck auftreten.

► Query: Spaltenwerte verknüpfen ▾

```

1  -- -----
2  -- Alphanumerische Werte verknuepfen
3  --
4  SELECT LAST_NAME || ' ' ||
5      FIRST_NAME
6  FROM EMPLOYEES;
```

2.2.5 Spaltenalias

Aliase sind **alternative Namen** für Spaltenbezeichner.

Ein Spaltenalias maskiert den eigentlichen **Spaltennamen** durch eine sprechende Bezeichnung.

► Erklärung: Spaltenalias ▾

- Ein Spaltenalias wird durch ein Leerzeichen getrennt, der Spaltenbezeichnung nachgestellt, deren Bezeichnung maskiert werden soll.
- Klauseln die von der SQL Engine nach der select Klausel ausgewertet werden, besitzen eine **Referenz** auf Spaltenalias.
- Enthält ein Spaltenalias **Sonderzeichen**, muß es unter Hochkomma gesetzt werden.

► Query: Spaltenaliase definieren ▾

```

1  -- -----
2  -- Spaltenaliase definieren
3  --
4  SELECT LAST_NAME NACHNAME, JOB_ID BERUF
5  FROM EMPLOYEES;
6
7  SELECT DEPARTMENT_NAME "Abteilungsendbericht"
8  FROM DEPARTMENTS;
```

2.2.6 Pseudospalten

Pseudospalten stellen eine alternative Form von Spaltenbezeichnern für SQL Abfragen dar.



Pseudospalte ▾

Pseudospalten treten in SQL Abfragen in 2 Formen auf: Funktionen und Spaltenalias:

- **Funktionen:** Die SQL Spezifikation, definiert eine Reihe von parameterlosen Funktionen die in SQL Abfragen als Teil der select Klammer verwendet werden.
- **Spaltenalias:** Ein Spaltenalias ist ein **symbolischer Alias** auf die Spalten einer anderen Tabelle.

Pseudospalten werden in SQL Abfragen wie **gewöhnliche Spaltenbezeichner** verwendet. Es können jedoch keine Werte für diese Art von Spalten eingetragen werden.

► Query: Pseudospalten ▾

```

1 -- -----
2 -- Pseudospalte: ROWNUM
3 --
4 -- Das Ergebnis der Abfrage setzt sich aus
5 -- den ersten 10 Datensaetze des Ergebnisses
6 -- zusammen.

7
8 SELECT * FROM EMPLOYEES WHERE ROWNUM < 10;
9 --
10 -- Pseudospalte: SYSDATE
11 --
12 -- Das Ergebnis der Abfrage enthaelt die
13 -- gegenwaertigen Zeit des Datenbankservers
14
15 -- z.B.: 2017-11-23 23:41:08
16
17
18 SELECT SYSDATE FROM DUAL;
19 --
20 -- Pseudospalte: TIMESTAMP
21 --
22 -- Das Ergebnis der Abfrage enthaelt die
23 -- gegenwaertigen Zeit des Datenbankservers
24 -- auf Millisekunden genau.
25
26 SELECT TIMESTAMP FROM DUAL;
```

2.2.7 Pseudospalten: NEXTVAL, CURRVAL

nextval und currval sind 2 Funktionen zur Verwaltung von Sequenzobjekten.

► Erklärung: Sequenzen in SQL Abfragen ▾

- Sequenzobjekte sind wie Tabellen Datenbankobjekte.
- Sequenzen werden zum **Generieren von Schlüsselwerten** für Datenbanktabellen verwendet.

► Query: NEXTVAL, CURRVAL ▾

```

1 -- -----
2 -- Pseudospalte: NEXTVAL
3 --
4 -- Die NEXTVAL Pseudospalte liefert den
5 -- naechsten Werte der Sequenz.
6
7 SELECT EMPLOYEE_SEQ.NEXTVAL FROM DUAL;
8 --
9 -- Pseudospalte: CURRVAL
10 --
11 -- Die CURRVAL Pseudospalte liefert den akt-
12 -- uellen Wert der Sequenz. Der Wert entspr-
13 -- icht dabei dem Wert, der beim letzten Auf-
14 -- ruf von SEQUENCE_NAME.NEXTVAL generiert
15 -- worden ist.
16
17 SELECT EMPLOYEE_SEQ.CURRVAL FROM DUAL;
```



2.2.8 Pseudospalte: ROWNUM

Die rownum Funktion ordnet jedem **Ergebnisdatensatz** einer SQL Abfrage eine Numer entsprechend seiner Position im Ergebnis einer Abfrage zu.

► Query: ROWNUM ▾

```

1 -- -----
2 -- Pseudospalte: ROWNUM
3 --
4 -- Das Ergebnis der Abfrage wird auf die
5 -- ersten 100 Datensaetze beschraenkt.
6
7 SELECT ROWNUM, LAST_NAME, FIRST_NAME
8 FROM EMPLOYEES
9 WHERE ROWNUM < 101;
```

Operator	Beschreibung	Sql Beispiel
ROWNUM	Jedem Ergebnisdatensatz einer SQL Abfrage wird eine <code>SELECT * FROM employees WHERE ROWNUM < 10</code> Numer entsprechend seiner Position im Ergebnis der Abfrage zugeordnet.	
SYSDATE	Die Pseudospalten SYSDATE Funktion ermöglicht den Zugriff auf die interne Zeit des Datenbankservers.	<code>SELECT SYSDATE FROM DUAL</code>
TIMESTAMP	Die Pseudospalten TIMESTAMP Funktion ermöglicht den Zugriff auf die interne Zeit des Datenbankservers.	<code>SELECT TIMESTAMP FROM DUAL</code>
USER	Die USER Pseudospalte ermöglicht den Zugriff auf den Namen des eingeloggten Benutzers der gegenwärtigen Datenbanksession.	<code>SELECT USER, UID FROM DUAL</code>
UID	Die UID Pseudospalte ermöglicht den Zugriff auf die ID des eingeloggten Benutzers der gegenwärtigen Datenbanksession.	<code>SELECT USER, UID FROM DUAL</code>
CURRVAL	Die CURRVAL Pseudospalte liefert den aktuellen Wert der Sequenz. Der Werte entspricht dabei dem Wert, der beim letzten Aufruf von <code>SEQUENCE_NAME.NEXTVAL</code> generiert worden ist.	<code>SELECT employee_seq.CURRVAL FROM DUAL</code>
NEXTVAL	Die NEXTVAL Pseudospalte liefert den nächsten Wert der Sequenz.	<code>SELECT employee_seq.NEXTVAL FROM DUAL</code>

Abbildung 11. Pseudospalten

2.2.9 Pseudospalten: USER, UID

Die USER bzw. UID Pseudospalten verweisen auf die Daten des eingeloggten Users.

```
▶ Query: USER, UID ▾
1  -- -----
2  --   Pseudospalte: USER
3  -- -----
4  -- Die USER Pseudospalte ermöglicht den
5  -- Zugriff auf den Namen des eingeloggten
6  -- Users.
7
8  SELECT USER, EMPLOYEES_ID FROM EMPLOYEES;
9  -- -----
10 --   Pseudospalte: UID
11 -- -----
12 -- Die UID Pseudospalte ermöglicht den Zu-
13 -- griff auf die ID des eingeloggten Benut-
14 -- zers der gegenwärtigen Datenbanksession.
15
16 SELECT UID, LAST_NAME FROM EMPLOYEES;
```

2.2.10 Pseudospalten: SYSDATE, TIMESTAMP

Die SYSDATE bzw. TIMESTAMP Pseudospalte ermöglicht den Zugriff auf die interne Zeit des Datenbankservers.

```
▶ Query: SYSDATE, TIMESTAMP ▾
1  -- -----
2  --   Pseudospalte: SYSDATE
3  -- -----
4  -- Die SYSDATE Pseudospalte ermöglicht den
5  -- Zugriff auf die interne Zeit des Daten-
6  -- bankservers.
7
8  SELECT SYSDATE FROM DUAL;
9  -- -----
10 --   Pseudospalte: TIMESTAMP
11 -- -----
12 -- Die TIMESTAMP Pseudospalte ermöglicht
13 -- den Zugriff auf die interne Zeit des
14 -- Datenbankservers.
15
16 SELECT TIMESTAMP FROM DUAL;
```

2.3. Case Klausel - Kondition

Mit der **case** Klausel definiert die SQL Spezifikation einen logischen **Operator**, der das Ergebnis einer Abfrage in konditionale Abhängigkeit zum Datenbestand setzt.

Die case Klausel wird stets vor der Klausel ausgewertet, in der sie definiert wurde.



2.3.1 Case Klausel

Case Klausel - Kondition

Die case Klausel ermöglicht eine **konditionale Verarbeitung** des Datenbestands einer SQL Abfrage.

Die Case Klausel zeigt dabei dasselbe Verhalten wie der ?: Operator imperativer Programmiersprachen.

► Syntax: Case Klausel

```

1 -- -----
2 -- Syntax : CASE Klausel
3 --
4 CASE WHEN <condition> THEN <result>
5   WHEN <condition> THEN <result>
6   [WHEN <condition> THEN <result>
7     ...]
8   [ELSE <result>] +
9 END

```

► Erklärung: Case Klausel

- Für eine **Case Klausel** können mehrere **<when then>** Paare definiert werden. Jedes **<when then>** Paar formuliert dabei eine Bedingung. Das Ergebnis der Auswertung der Case Klausel ist das Wert des ersten wahren **<when then>** Paars.
- Syntaktisch gelten für **Bedingungen** der case und where Klausel dieselben Regeln.
- Der else Zweig der case Klausel kommt immer dann zur Ausführung, wenn keine der Bedingungen der vorangegangenen when then Paare zutreffen. Wird kein else Zweig definiert, wird er als **else null** angenommen.

2.3.2 Case Klauseln in select Abfragen

Die case Klausel zeigt im Vergleich zu anderen Klauseln mehr das **Verhalten eines Operators** als einer Klausel.

Ein Aufruf der case Klausel kann nur im Kontext einer anderen **Klausel** erfolgen.

► Query: Case Klausel

```

1 -- -----
2 -- Fallbeispiele : CASE in ORDER BY
3 --
4 SELECT COUNTRY_ID, FIRST_NAME, LAST_NAME
5 FROM EMPLOYEES
6 ORDER BY
7 CASE
8   WHEN LOCATION_ID IS NULL THEN COUNTRY_ID
9   ELSE LOCATION_ID
10 END;
11 --
12 --
13 -- Fallbeispiele : CASE in SELECT
14 --
15 SELECT FIRST_NAME, LAST_NAME,
16 CASE
17   WHEN SALARY <= 1000
18     THEN 'low income'
19   WHEN SALARY BETWEEN 10000 AND 20000
20     THEN 'medium income'
21   WHEN SALARY BETWEEN 20001 AND 40000
22     THEN 'upper income'
23   WHEN SALARY BETWEEN 40001 AND 80000
24     THEN 'ceo'
25   ELSE null
26 END AS INCOME
27 FROM EMPLOYEES
28 ORDER BY LAST_NAME, FIRST_NAME;
29 --
30 --
31 -- Fallbeispiele : CASE in WHERE
32 --
33 SELECT FIRST_NAME, LAST_NAME
34 FROM EMPLOYEES
35 WHERE SALARY >
36 CASE
37   WHEN DEPARTMENT_ID = 106 THEN 1000
38   WHEN DEPARTMENT_ID = 107 THEN 2000
39 END
40 ORDER BY LAST_NAME;

```

2.4. Where Klausel - Restriktion



Where Klausel - Restriktion ▾

Zur **Filterung** des Datenbestandes einer Abfrage können in der where Klausel Bedingungen definiert werden.

Die where Klausel ist **optional**.

2.4.1 Restriktion

Zur **Filterung** der Datensätzen einer Abfrage können in der where Klausel Bedingungen formuliert werden.

Mit der where Klausel können **Restriktionen** für die Daten einer Abfrage definiert werden.

► Erklärung: where Klausel ▾

- Innerhalb der where Klausel werden Bedingungen definiert, die für jeden Datensatz der Datenbasis geprüft werden.
- Evaluiert die Prüfung einer Bedingung für einen Datensatz nicht zu **true**, wird der Datensatz aus der Datenbasis der Abfrage entfernt.

► Query: where Klausel - Datenfilterung ▾

```

1  -- -----
2  -- WHERE Klausel
3  --
4  SELECT LAST_NAME, FIRST_NAME, JOB_ID,
5      DEPARTMENT_ID,
6      SALARY
7  FROM EMPLOYEES
8  WHERE JOB_ID = 'SA_MAN'
9  ORDER BY LAST_NAME, FIRST_NAME;
10
11 SELECT LAST_NAME, FIRST_NAME
12 FROM EMPLOYEES
13 WHERE SALARY > 4000;
```



2.4.2 Abfrageobjekt

Ein **Abfrageobjekt** wird zum Filtern der Daten einer Abfrage verwendet.

Abfrageobjekte formulieren logische Bedingungen in Form von **mathematischen Aussagen**.

► Erklärung: Abfrageobjekt ▾

- Eine Bedingung ist ein Ausdruck, der immer zu wahr oder falsch evaluiert.
- Abfrageobjekte können durch die **Verknüpfung** mathematischer Aussagen weiter präzisiert werden.
- Zur Kombination mathematischer Aussagen werden **logische Operatoren** verwendet.



2.4.3 Logische Operatoren - AND, OR, XOR



Verknüpfungsoperatoren ▾

Verknüpfungsoperatoren werden verwendet um logische **Bedingungen** miteinander zu **verknüpfen**. Die SQL Spezifikation definiert dazu die Operatoren **and**, **or**, **not** und **xor**.

► Erklärung: Logische Operatoren ▾

- Zur Formulierung komplexer Bedingungen können die Operatoren **and**, **or** bzw **not** in einem Abfrageobjekt beliebig kombiniert werden.

► Query: Logische Operatoren: ▾

```

1  -- -----
2  -- logische Operatoren: and, or, not
3  --
4  SELECT LAST_NAME, FIRST_NAME, SALARY,
5      DEPARTMENT_NAME,
6      DEPARTMENT_ID
7  FROM EMPLOYEES
8  WHERE
9  (JOB_ID = 'CLERK' OR JOB_ID = 'MAN')
10 AND
11 (SALARY > 100 OR LAST_NAME = 'Mel')
12 ORDER BY LAST_NAME, FIRST_NAME;
```



2.4.4 Logischer Operator - LIKE

Der like Operator ermöglicht es Zeichenketten auf das Vorhandensein bestimmter **Zeichenfolgen** zu prüfen.

Zur Beschreibung einer Zeichenfolge wird eine sogenannte **Suchmaske** formuliert. Die SQL Spezifikation definiert dazu zwei Platzhaltertoken.

► Erklärung: Platzhaltertoken ▾

- Der % Token steht stellvertretend für eine beliebige Zahl von Zeichen.
- Der _ Token wird als Platzhalter für ein beliebiges Zeichen verwendet.

► Query: Patternmatching ▾

```

1  -- -----
2  -- logischer Operator - like
3  -- -----
4  SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID
5  FROM EMPLOYEES
6  WHERE LAST_NAME LIKE 'M%';
7
8  SELECT LAST_NAME, FIRST_NAME, JOB_ID, SALARY
9  FROM EMPLOYEES
10 WHERE LAST_NAME LIKE '_A%';

```



2.4.5 Logischer Operator - IN

Mit dem in Operator kann geprüft werden, ob ein bestimmter Wert in einer Liste von Werten enthalten ist.

► Query: in Operator ▾

```

1  -- -----
2  -- logischer Operator - in
3  -- -----
4  SELECT LAST_NAME, JOB_ID, SALARY
5  FROM EMPLOYEES
6  WHERE JOB_ID IN ('SA_REP', 'SA_CLERK');

```

► Analyse: in Operator ▾

- Wir suchen alle Mitarbeiter deren Berufsbezeichnung entweder SA_REP oder SA_CLERK ist.
- Der in Operator kann in Verbindung mit numerischen bzw. alphanumerischen Werten verwendet werden.



2.4.6 Logischer Operator - IS

Mit dem is Operator kann geprüft werden, ob der Wert einer bestimmten Spalte bekannt ist.

► Erklärung: is Operator ▾

- Der Vergleich last_name = null wird unabhängig von dem in der last_name Spalte gespeichertem Wert immer false ergeben.
- Der Grund dafür ist das der Vergleich null = null stets zu false evaluiert.
- Zur Prüfung auf null muß der is Operator verwendet werden.

► Query: Prüfung auf null Werte ▾

```

1  -- -----
2  -- logischer Operator - in
3  -- -----
4  SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID
5  FROM EMPLOYEES
6  WHERE COMMISSION_PCT IS NOT NULL;

```



2.4.7 Logischer Operator - BETWEEN

Um zu prüfen ob ein Wert in einem **Intervall** von Werten enthalten ist, wird der between Operator verwendet.

► Query: between Operator ▾

```

1  -- -----
2  -- logischer Operator - between
3  -- -----
4  SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID
5  FROM EMPLOYEES
6  WHERE
7  SALARY BETWEEN 10000 AND 14000;
8
9
10 -- -----
11 -- logischer Operator - between
12 -- -----
13 SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID,
14   DEPARTMENT_ID,
15   SALARY
16 FROM EMPLOYEES
17 WHERE BIRTHDAY BETWEEN
18 TO_DATE('10.01.1950', 'dd.mm.yyyy')
19 AND
20 TO_DATE('10.01.1955', 'dd.mm.yyyy');

```



Operator	Beschreibung	Sql Beispiel
AND	Die Verknüpfung logischer Terme mit einem and wird als <code>first_name like 'A_'</code> and <code>salary > 1000</code> Konjunktion bezeichnet. Eine Konjunktion ist wahr wenn jeder angegebene logische Term wahr ist.	
OR	Die Verknüpfung logischer Terme mit einem or wird als Disjunktion bezeichnet. Ein Disjunktion ist wahr wenn einer der angegebenen Terme wahr ist.	<code>salary < 0 or salary > 20000</code>
NOT	Die Negation ist ein Operator zur Verneinung logischer Aussagen	<code>not salary = 2000</code>
IN	Der in Operator prüft ob ein gegebener Wert in einer Liste von Werten enthalten ist.	<code>department_id in (106, 107, 109)</code>
LIKE	Der like Operator prüft alphanumerische Werte auf das Vorhandensein von Mustern.	<code>first_name like K%</code>
IS	Der is Operator prüft ob der Wert für eine bestimmte Spalte bekannt ist.	<code>department_id is not null</code>
BETWEEN	Der between Operator prüft ob ein Wert in einem Intervall von Werten enthalten ist.	<code>salary between 10000 and 14000</code>

Abbildung 12. Logische Operatoren

2.4.8 Dreiwertige Logik



Dreiwertige Logik ▾

Die SQL Spezifikation kennt zum Unterschied zu anderen Programmiersprachen 3 **Wahrheitswerte**: true, false und **null**.

Der **null Wert** kann mit nichts verglichen werden, er ist weder falsch noch wahr sondern eben einfach unbekannt.

and	t	f	n
t	t	f	n
f	f	f	f
n	n	f	n

or	t	f	n
t	t	t	t
f	f	f	n
n	n	t	n

Mit dem **null Wert** verlassen wir die Welt der intuitiv erfassbaren Logik und führen eine dreiwertige Logik ein, die neben **wahr** und **falsch** noch **unbekannt** als Wahrheitswert kennt. Der null Wert kann mit nichts verglichen werden. Die Negation eines unbekannten Werts ist selbst wieder unbekannt.

► Erklärung: Dreiwertige Logik ▾

- Wir möchten berechnen, wie hoch das tatsächliche Jahresgehalt eines Angestellten ist.
- Die Spalte `commission_pct` enthält null Werte. Berechnungen auf null Werten sind problematisch. Wird z.B.: zu einer Zahl eine unbekannte Zahl addiert, ist das Ergebnis wieder eine unbekannte Zahl.
- Zur Verarbeitung von Nullwerten definiert die SQL Spezifikation die `coalesce` Funktion.

Die **coalesce Funktion** erlaubt die Angabe eines Defaultwerts, der immer dann herangezogen wird, wenn der Spaltenwert eines Datensatzes unbekannt ist.

► Query: Mit Nullwerten arbeiten ▾

```

1 -- -----
2 -- Mit Nullwerten arbeiten
3 -- -----
4 SELECT LAST_NAME, FIRST_NAME,
5      DEPARTMENT_ID,
6      (SALARY * 12) +
7      COALESCE(COMMISSION_PCT, 0.3)
8 FROM EMPLOYEES;

```



2.5. Order by Klausel - Sortierung

Order By Klausel - Sortierung ▾

Mit der order by Klausel kann eine **Sortierung** der Datensätze des Ergebnisses einer Abfrage definiert werden.

Die order by Klausel ist **optional**.

2.5.1 Order By Klausel

Mit der order by Klausel kann eine **Sortierung** der Datensätze einer Abfrage definiert werden.

Beachten Sie, dass nur nach Spalten sortiert werden kann, die Teil des Ergebnisses der Abfrage sind.

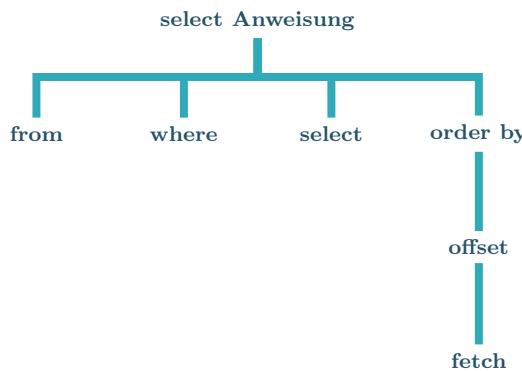
► Query: Einsatz der Order By Klausel ▾

```

1  -- -----
2  -- ORDER BY Klausel
3  --
4  SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID,
5      DEPARTMENT_NAME,
6      JOB_ID
7  FROM EMPLOYEES
8  WHERE DEPARTMENT_ID = 30
9  ORDER BY LAST_NAME;
```

► Analyse: Order Klausel ▾

- Es werden alle Mitarbeiter der Abteilung 30, sortiert nach ihren Nachnamen, ausgeben.
- **Reihenfolge der Auswertung der Klauseln:**



2.5.2 Sortierreihenfolge - asc/desc

Durch die Angabe des **asc** bzw. **desc** Schlüsselworts kann die Art der Sortierung zusätzlich präzisiert werden.

► Query: Sortieren des Abfrageergebnisses ▾

```

1  --
2  -- ORDER BY Klausel - ACS, DESC
3  --
4  SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID,
5      DEPARTMENT_NAME,
6      SALARY
7  FROM EMPLOYEES
8  WHERE DEPARTMENT_ID = 30
9  ORDER BY LAST_NAME ASC, SALARY DESC;
```

► Analyse: Sortierung ▾

- Im Beispiel wird das Abfrageergebnis zuerst aufsteigend nach den Werten der last_name Spalte sortiert.
- Treten in der last_name Spalte Datensätze mit gleichen Werten auf, werden diese Zeilen absteigend nach den Werten der salary Spalte sortiert.



2.5.3 Optionen - nulls first und nulls last

Eine letzte Erweiterung der order by Klausel betrifft die Behandlung von null Werten.

► Erklärung: Null Werte ▾

- Mit dem **nulls first** bzw. **nulls last** Schlüsselwort werden Datensätze, die Null Werte enthalten, summarisch dem Ergebnis entweder voran- bzw. nachgestellt.

► Query: Null Werte ▾

```

1  --
2  -- ORDER BY Klausel - Null Werte
3  --
4  SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID,
5      DEPARTMENT_NAME,
6      JOB_ID,
7  FROM EMPLOYEES
8  WHERE DEPARTMENT_ID = 30
9  ORDER BY LAST_NAME ASC,
10 NULLS FIRST;
```



28	Fareeha Amari	(334) 700-8000	justicerains@overwatch.com	APPLIED	07/12/16 11:55PM	EDIT	DELETE
27	Hana Song	(700) 132-5468	D.va@overwatch.com	APPROVED	07/12/16 9:08PM	EDIT	DELETE
25	Angela Zeigler	(800) 867-5309	mercy@overwatch.com	APPROVED	07/12/16 7:31PM	EDIT	DELETE
24	Mei-Ling Zhou	(300) 666-7777	mei@overwatch.com	DISMISSED	07/12/16 5:40AM	EDIT	DELETE

◀ ◀ 1 2 3 4 5 6 ▶ ▶|

Abbildung 13. Paginierung von Daten

2.6. Fetch Klausel - Limitierung



Fetch Klausel - Limitierung

Mit der fetch Klausel kann das Ergebnis einer Abfrage auf eine bestimmte Zahl von Datensätzen **beschränkt** werden.

Die fetch Klausel ist **optional**.

Der Einsatz der fetch Klausel macht nur dann Sinn, wenn das Abfrageergebnis in **sortierter Form** vorliegt.

2.6.1 Fetch Klausel

Mit der fetch Klausel wird das Ergebnis einer Abfrage auf die ersten N Datensätze **beschränkt**.

► Query: Einsatz der fetch Klausel

```

1 -- -----
2 -- FETCH Klausel - Anzahl der Datensaetze
3 --
4 SELECT LAST_NAME, FIRST_NAME, DEPARTMENT_ID
5 FROM EMPLOYEES
6 ORDER BY SALARY DESC
7 FETCH FIRST 5 ROWS ONLY;
8 --
9 --
10 -- FETCH Klausel - Prozentangabe
11 --
12 SELECT LAST_NAME, FIRST_NAME, JOB_ID
13 FROM EMPLOYEES
14 ORDER BY SALARY DESC
15 FETCH FIRST 20 PERCENT ROWS ONLY;
```

2.7. Offset Klausel - Paginierung

Mit der offset Klausel kann ein seitenweiser Zugriff auf das Ergebnis einer Abfrage definiert werden.

2.7.1 Offset Klausel

Mit der offset Klausel können die ersten n Datensätze aus dem Ergebnis einer Abfrage entfernt werden.

► Query: offset Klausel

```

1 -- -----
2 -- OFFSET Klausel
3 --
4 -- Paginierung des Abfrageergebnisses
5 -- Ergebnis: Datensaetze 21 - 40
6 SELECT LAST_NAME, FIRST_NAME, MIDDLE_NAME
7     DEPARTMENT_NAME,
8     DEPARTEMNT_ID,
9     SALARY
10 FROM EMPLOYEES
11 ORDER BY LAST_NAME ASC,
12         FIRST_NAME ASC
13 OFFSET 20 ROWS
14 FETCH NEXT 20 ROWS ONLY;
```

► Erklärung: Offset Klausel

- Die **offset Klausel** zusammen mit der **fetch Klausel** werden verwendet um eine **Paginierung** des Abfrageergebnisses zu definieren.
- Eine Paginierung des Abfrageergebnisses erlaubt den seitenweisen Zugriff auf die Datensätze einer Abfrage.
- Vermeiden Sie in SQL Abfragen den Zugriff auf alle Datensätze einer Tabelle!

3. SQL - Datenaggregation

02

Datenaggregation

01. Datenmodellierung 36

02. From Klausel 37

03. Relationale Join 37

3.1. Datenmodellierung



Relationale Modell ▾

Das Relationale Modell beschreibt die grundlegende **Tabellen-** bzw. **Beziehungssstruktur** einer relationalen Datenbank.

► Erklärung: Relationale Modellierung ▾

- Bei der Arbeit mit relationalen Datenbanken werden die **Geschäftsobjekte** einer Anwendung auf eine Menge von Tabellen abgebildet.
- Jede **Tabelle** repräsentiert dabei einem bestimmten **Aspekt** der Wirklichkeit.
- Das Aufteilen der Daten auf einzelne Tabellen ermöglicht eine klare Strukturierung der Daten in der Datenbank.
- Beim Formulieren einer SQL Abfrage stehen wir oft vor der Herausforderung, die Daten wieder so zusammenzustellen, dass das ursprüngliche **Geschäftsobjekt** zum Vorschein kommt.



3.1.2 Datenaggregation

Datenaggregation beschreibt den Prozess der **Verdichtung** einzelner Aspekte der **Geschäftdaten** zu neuen Geschäftsbobjekten.

► Erklärung: Datenaggregation ▾

- Mit dem Aufteilen der Daten auf getrennte Aspekte, können Datensätze ihrerseits wiederum zu neuen Geschäftsbobjekten zusammengesetzt werden.
- Durch die geschickte Kombination der Aspekte eines Geschäftsfalls kann aus den gegebenen Daten neue Information gewonnen werden.
- Die technische Grundlage der **Datenaggregation** ist der **relationale Join**. Ein relationaler Join beschreibt die logische Verbindung zweier Datensätzen. Relationale Datenbanken unterstützen dabei mehrere Formen des relationalen Joins.
- In SQL Abfragen werden relationale Joins in der from Klausel definiert.



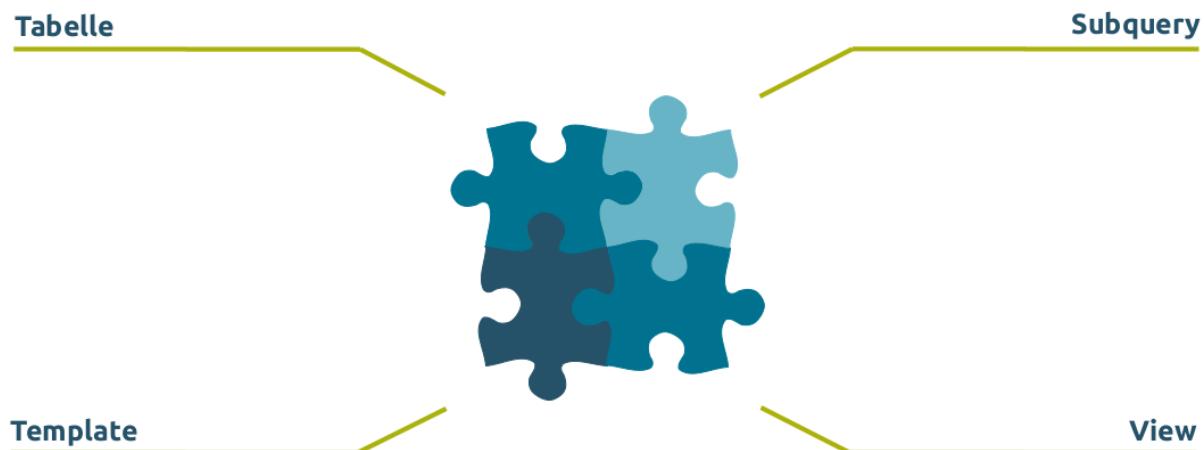


Abbildung 14. Datenbankobjekte der from Klausel

3.2. From Klausel

In der from Klausel einer select Anweisung wird die Datenbasis der Abfrage definiert.

3.2.1 Grundlagen

SQL Abfragen beziehen sich immer auf die in der from Klausel definierte Datenbasis.

► Erklärung: From Klausel ▾

- Für select Anweisungen ist die from Klausel **obligatorisch**. Die Klausel wird vor der Auswertung jeder anderen Klausel einer SQL Abfrage ausgeführt.
- Zur **Definition der Datenbasis** einer Abfrage werden in der from Klausel jene Datenbankobjekte angegeben, die die gewünschten Daten halten. Mögliche Datenbankobjekte sind dabei Tabellen, Views bzw. Subselects.
- In **komplexen Abfragen** werden Datenbankobjekte in der from Klausel in **Relation** gesetzt.

3.3. Relationale Join

Ein relationaler Join definiert die **logische Verknüpfung** zweier Datenbankobjekten.

3.3.1 Virtuelle Tabellen

Das Ergebnis eines Joins wird in einer **virtuellen Tabelle** gespeichert.

► Erklärung: Datenbasis einer Abfrage ▾

- In der from Klausel einer select Anweisung wird die Datenbasis der Abfrage definiert. Für komplexe Abfragen werden mehrere Datenbankobjekte in der from Klausel in Beziehung gesetzt.
- Das Ergebnis der Auswertung eines relationalen Joins wird in einer virtuellen Tabelle gespeichert.
- Eine Virtuelle Tabllen entspricht inhaltlich der **Kombination** der ursprünglichen **Datenbankobjekte**.
- Die select Abfrage kann nach der Join Stufe jedoch nicht mehr auf die Datensätze der ursprünglichen Datenbankobjekte zugreifen. Es wird eine neue **Virtuelle Tabelle** generiert, auf die sich die Abfrage nun bezieht.



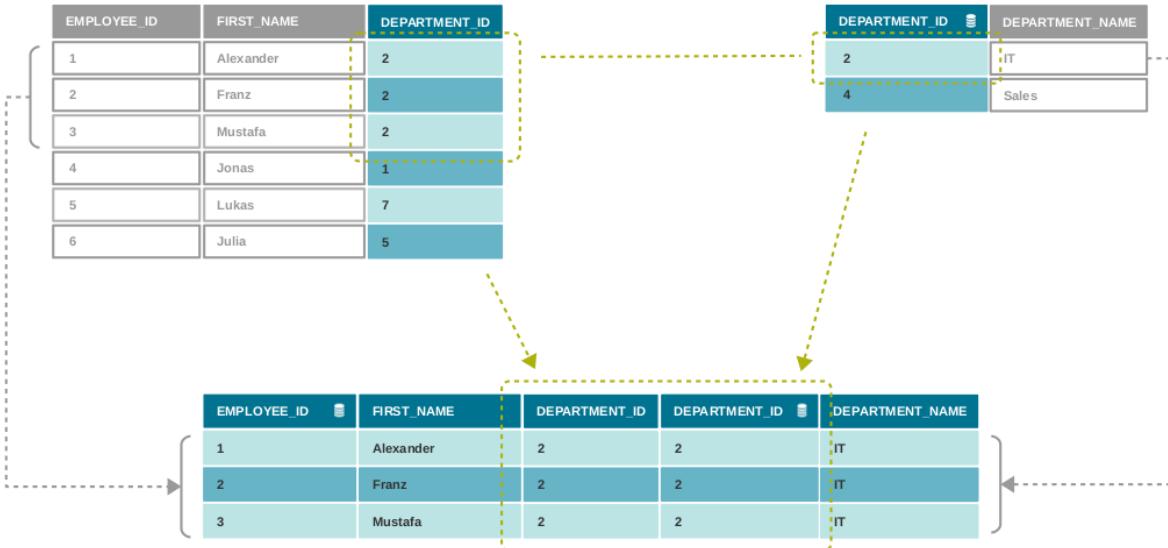


Abbildung 15. Relationale Join: Inner Join

3.3.2 Jointyp: Inner Join

Ein **Inner Join** führt die Datensätze zweier Datenbankobjekte genau dann zusammen, wenn für die Datensätze die angegebene **join Bedingung** erfüllt ist.

► Erklärung: Join Bedingung ▾

- Syntaktisch wird die **Join Bedingung** mit dem Schlüsselwort **on** eingeleitet.
- Das **INNER** Schlüsselwort ist dabei **optional**.

3.3.3 Jointyp: Natural Join

Beim Natural Join handelt es sich um eine besondere Form des **Inner Joins**.

Beim Natural Join werden die Datensätze der Datenobjekte über alle Spalten mit demselben Namen in Beziehung gesetzt.

► Syntax: Natural Join ▾

```

1 -- -----
2 -- Syntax: INNER JOIN
3 -- -----
4 SELECT ...
5 FROM <Table1> [INNER] JOIN <Table2>
6 ON <Bedingung>
7 ...
8
9 -- Beispiel: Inner Join
10 SELECT E.EMPLOYEE_ID, E.LAST_NAME,
11      D.DEPARTMENT_NAME
12 FROM EMPLOYEES E JOIN DEPARTMENTS D
13 ON E.DEPARTMENT_ID = D.DEPARTMENT_ID;

```



Jointyp	Beschreibung	Seite
CROSS JOIN	Ein cross join führt Datensätze zweier Datenbankobjekte zusammen, indem jeder Datensatz des ersten Objekts mit jedem Datensatz des zweiten Objekts in Beziehung gesetzt wird.	39
INNER JOIN	Ein Inner Join führt die Datensätze zweier Datenbankobjekte genau dann zusammen, wenn für die Datensätze die angegebene join Bedingung erfüllt ist.	38
NATURAL JOIN	Beim Natural Join handelt es sich um eine besondere Form des Inneren Joins. Beim Natural Join werden die Datensätze der Datenobjekte über alle Spalten mit demselben Namen in Beziehung gesetzt.	38
LEFT/RIGHT JOIN	Ein Left Join übernimmt alle Datensätze des ersten Datenobjekts und setzt sie mit korrelierenden Datensätzen des zweiten Datenobjekts in Beziehung.	41
FULL OUTER JOIN	Der Full Outer Join entspricht semantisch einer Kombination aus Left- und Rightjoin.	39

Abbildung 16. Jointypen

3.3.4 Jointyp: Cross Join

Ein **cross join** führt Datensätze zweier Datenbankobjekte zusammen, indem jeder Datensatz des ersten Objekts mit jedem Datensatz des zweiten Objekts in Beziehung gesetzt wird.

► Erklärung: Cross Join ▾

- Das Ergebnis eines cross joins enthält damit $m \times n$ Zeilen wenn die eine Tabelle m und die andere Tabelle n Zeilen enthält.

► Syntax: Cross Join ▾

```

1  -- -----
2  -- Syntax: CROSS JOIN
3  --
4  SELECT ...
5  FROM <Table1> CROSS JOIN <Table2>
6  ...
7
8  -- employees.zeile1 <-> departments.zeile1
9  -- employees.zeile1 <-> departments.zeile2
10 --      ... <-> ...
11 --          ... <-> departments.zeileN
12 -- employees.zeile2 <-> departments.zeile1
13 --      ... <-> ...
14 -- employees.zeileM <-> departments.zeileN
15
16 -- Beispiel: Cross Join
17 SELECT E.FIRST_NAME, D.DEPARTMENT_NAME
18 FROM EMPLOYEES E
19 CROSS JOIN DEPARTMENTS D;
```

3.3.5 Jointyp: Full Outer Join

Der **Full Outer Join** entspricht semantisch einer Kombination aus Left- und Rightjoin.

► Erklärung: Full Outer Join ▾

- Mengentheoretisch entspricht der Full outer Join zweier Mengen A und B der **Vereinigungsmenge** der Mengen.
- Der full outer join zweier Mengen als Relation ist **kommutativ**.

► Syntax: Full Outer Join ▾

```

1  -- -----
2  -- Syntax: FULL OUTER JOIN
3  --
4  SELECT ...
5  FROM <Tabelle1> FULL OUTER JOIN <Tabelle2>
6  ON <Bedingung>
7  ...
8
9  --
10 -- Beispiel: FULL OUTER JOIN
11 --
12 SELECT E.LAST_NAME, E.FIRST_NAME,
13     E.LAST_NAME,
14     D.DEPARTMENT_NAME,
15     D.DEPARTMENT_ID
16 FROM EMPLOYEES E
17 FULL OUTER JOIN DEPARTMENTS D
18 ON E.DEPARTMENT_ID = D.DEPARTMENT_ID
19 ORDER BY E.LAST_NAME, E.FIRST_NAME;
```

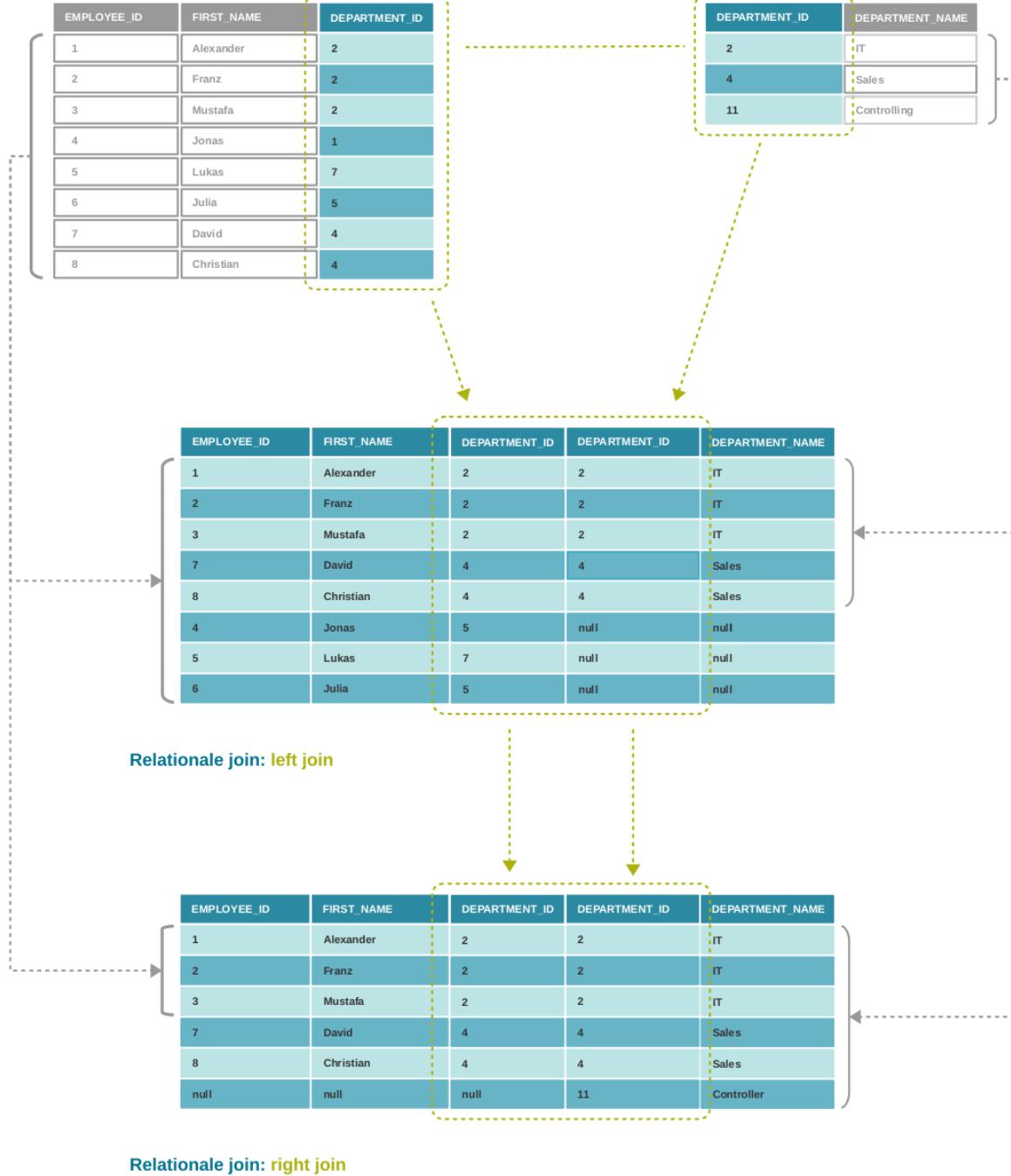


Abbildung 17. Relationale Join: Left/Right Join

3.3.6 Jointyp: Left-/Right Join

Ein Left Join übernimmt alle Datensätze des ersten Datenobjekts und setzt sie mit korrelierenden Datensätzen des zweiten Datenobjekts in Beziehung.

Left- und Right Join beschreiben denselben **Jointyp**, in jeweils gespiegelter Form.

► Erklärung: Left-/Right Join ▾

- Mengentheoretisch entspricht der `left join` zweier Tabellen A und B der Menge A.
- Die Syntax ist dabei bis auf die Schlüsselwörter `left` bzw. `right` identisch zur Syntax des Inner Join.
- Der left- bzw. right Join ist keine kommutative Relation.

► Syntax: Left Join/Right Join ▾

```

1  -- -----
2  -- Syntax: LEFT JOIN
3  --
4  SELECT ...
5  FROM <Table1> LEFT JOIN <Table2>
6  ON <Bedingung>
7  ...
8
9  -- Beispiel: Left Join
10 SELECT E.FIRST_NAME,
11      E.LAST_NAME,
12  FROM EMPLOYEES E
13 LEFT JOIN DEPARTMENTS D
14 ON E.DEPARTMENT_ID = D.DEPARTMENT_ID;
15
16
17 -- -----
18 -- Syntax: RIGHT JOIN
19 --
20 SELECT ...
21 FROM <Table1> RIGHT JOIN <Table2>
22 ON <Bedingung>
23 ...
24
25 -- Beispiel: Right Join
26 SELECT E.FIRST_NAME,
27      E.LAST_NAME
28      E.SALARY,
29  FROM EMPLOYEES E
30 RIGHT JOIN DEPARTMENTS D
31 ON E.DEPARTMENT_ID = U.DEPARTMENT_ID;
```



4. SQL - Zeilenfunktionen

03

Zeilenfunktionen

01. Grundlagen: Zeilenfunktionen	42
02. Datumsfunktionen	43
03. Textfunktionen	50
04. Numerische Funktionen	53

4.1. Zeilenfunktionen - Grundlagen

4.1.1 Funktionstypen

Die SQL Spezifikation definiert 2 Arten von Funktionen.

► Auflistung: Arten von Funktionen ▾



Zeilenfunktionen ▾

Zeilenfunktionen verarbeiten die Daten eines einzelnen **Datensatzes**. Der Rückgabewert einer Zeilenfunktion ist stets ein einzelner Wert.



Aggregatfunktionen ▾

Aggregatfunktionen **verdichten** mehrere **Datensätze** zu einem einzelnen Wert.

4.1.2 Zeilenfunktionen

Syntaktisch gesehen haben alle **Zeilenfunktionen** gemeinsam, dass sie die Daten eines einzelnen Datensatzes, zu einem einzelnen Wert verdichten.

► Erklärung: Zeilenfunktionen ▾

- Die SQL Spezifikation definiert eine Reihe von Zeilenfunktionen.

```

1  -- -----
2  -- Zeilenfunktionen
3  --
4  -- Die Funktion gibt das aktuelle Datum
5  -- des Datenbankservers zurueck
6  -- Da die FROM Klausel fuer eine select
7  -- Abfrage obligatorisch ist wird die
8  -- Pseudotabelle dual verwendet
9  SELECT SYSDATE FROM DUAL;

10
11 -- Funktion zum Berechnen des aktuellen
12 -- Zeitpunkts
13 SELECT SYSTIMESTAMP FROM DUAL;

14
15 -- Funktion zum Berechnen der Laenge eines
16 -- alphanumerischen Wertes
17 SELECT LENGTH(LAST_NAME) FROM EMPLOYEES;

```

- Ein Großteil, der in der SQL Spezifikation definierten Zeilenfunktionen erwartet dabei **Funktionsparameter**.
- **Funktionsparameter** sind Werte, die einer Funktion zur Verarbeitung mitgegeben werden.

```

1  -----
2  --  Funktionsparameter
3  -----
4  SELECT LOWER(ENAME) ERGEBNIS FROM EMP;

```



4.1.3 Kategorien von Zeilenfunktionen

Die SQL Spezifikation unterscheidet mehrere **Kategorien** von Zeilenfunktionen.

► Auflistung: Kategorien von Zeilenfunktionen ▾

- **Datumsfunktionen:** Funktionen zum Bearbeiten zeitbezogener Daten.

```

1  -----
2  --  Datumsfunktion: to_date
3  -----
4  FUNCTION TO_DATE (
5      P_TIME_LITERAL IN VARCHAR2,
6      P_TIME_MASK    IN VARCHAR2
7  )
8  RETURN DATE;

```



- **Zeichenfunktionen:** Funktionen zum Bearbeiten von Zeichenketten.

```

1  -----
2  --  Zeichenfunktion: instr
3  -----
4  FUNCTION INSTR (
5      P_CONTENT      IN VARCHAR2,
6      P_TOKEN        IN VARCHAR2,
7      P_START_INDEX IN INTEGER
8  )
9  RETURN NUMBER;

```



- **Mathematische Funktionen:** Funktionen zur Transformation von Zahlenwerten.
- **Konvertierungsfunktionen:** Funktionen zum **Konvertieren** von Werten eines Datentyps zu einem anderen Datentyp.

4.2. Datumsfunktionen ▾

Datumsfunktionen werden zur Verarbeitung **zeitbezogener Daten** verwendet.

► Auflistung: Datumsfunktionen ▾

```

1  -----
2  --  Datumsfunktionen
3  -----
4  FUNCTION TO_DATE (
5      P_TIME_LITERAL IN VARCHAR2,
6      P_TIME_MASK    IN VARCHAR2
7  )
8  RETURN DATE;
9
10 FUNCTION TO_CHAR(
11     P_DATE       IN DATE,
12     P_TIME_MASK  IN VARCHAR2
13 )
14 RETURN VARCHAR2;
15
16 FUNCTION TRUNC (
17     P_DATE IN DATE,
18     P_MASK IN VARCHAR2
19 )
20 RETURN DATE;

```



4.2.1 Datumstypen vs. Zeichenketten



Datumswerte ▾

Datumswerte werden verwendet um **zeitbezogene Werte** in einer Datenbank abzubilden.

Die SQL Engine speichert Datumswerte dabei als die Anzahl von **Millisekunden**, die seit dem 01.01.1972 vergangen sind.

► Erklärung: Darstellung von Datumstypen ▾

- Bevor ein Datumswert angezeigt werden kann, muß er durch die Datenbankengine entsprechend **formatiert** werden.
- Durch die Angabe sogenannter **Masken** bestimmt der Benutzer die Art der **Formatierung** für die Ausgabe eines Datumwertes.



4.2.2 Erzeugen eines Datums

Zum **Erzeugen** von Datumswerten definiert die SQL Spezifikation 3 Möglichkeiten.

► Auflistung: Erzeugen eines Datums ▾

	Konstruktorfunktionen ▾
	Funktionen zum direkten Erzeugen von Datumswerten .
	Konvertierungsfunktionen ▾
	Funktionen zum Konvertieren von Zeichenketten in Datumswerte .
	Literale ▾
	Erzeugen von Datumswerten aus Literalen .

4.2.3 Konstrukturfunktionen

Die SQL Spezifikation definiert die **sysdate** und **systimestamp** Funktionen zum Erzeugen von Datumswerten.

Die **sysdate** und **systimestamp** Funktionen werden ohne Parameter aufgerufen, und liefern als Ergebnis die aktuelle Zeit des Datenbankservers.

► Query: Konstruktorfunktionen ▾

```

1 -- -----
2 -- Konstruktorfunktionen
3 -- -----
4 -- Generieren eines Date Datumswertes. Es
5 -- wird das aktuelle Datum ausgegeben.
6 SELECT SYSDATE FROM DUAL;
7
8 -- Ausgabe: 2018-10-30 18:18:27
9
10
11 -- Generieren eines Timestamp Datumswertes.
12 -- Es wird das aktuelle Datum ausgegeben.
13 SELECT SYSTIMESTAMP FROM DUAL;
14
15 -- Ausgabe:
16 -- 2018-10-30 18:18:27.590954 +01:00

```

4.2.4 Konvertieren von Datumswerten - `to_date`

Die `to_date()` Funktion konvertiert Zeichenketten in Datumswerte. Die Funktion erwartet beim Aufruf 2 Parameter.

► Parameter: `to_date` Funktion ▾

- `p_time_literal`: Der Parameter beschreibt einen Datumswert.
- `p_time_mask`: Der Parameter beschreibt eine Formatmaske.

► Syntax: `to_date` Funktion ▾

```

1 -- -----
2 -- Syntax: to_date
3 -- -----
4 FUNCTION TO_DATE (
5   P_TIME_LITERAL IN VARCHAR2,
6   P_TIME_MASK     IN VARCHAR2
7 )
8   RETURN DATE;
9
10 -- Aufruf: to_date()
11 SELECT TO_DATE (
12   '15.05.2012 17:30:56',
13   'dd.mm.yyyy hh24:mi:ss'
14 )
15 FROM dual;

```

4.2.5 Datumsliterale

Datumswerte selbst, können ebenfalls aus einem **Literal** generiert werden. Ein **Datumsliteral** muß dabei dem ISO Datumsformat entsprechend aufgebaut sein.

► Query: Datumsliterale ▾

```

1 -- -----
2 -- Literale
3 -- -----
4 -- Literal: yyyy-mm-dd
5 SELECT DATE '2012-05-15' DATUM
6 FROM DUAL;
7
8 -- Literal: yyyy-mm-dd hh24:mi:ss
9 SELECT TIMESTAMP '2012-05-15 15:00:00' DATUM
10 FROM DUAL;

```

Rundungsparameter	Beschreibung	Seite
YEAR, Y	Das Datum wird auf den ersten Tag des Jahres gerundet. Die round Funktion rundet Datumswerte ab dem 1.Juli auf.	45
Q	Das Datum wird auf den ersten Tag des Quartals gerundet in das der Datumswert fällt. Werte ab dem 16ten des zweiten Monats eines Quartals werden von der round Funktion aufgerundet.	45
MONTH, MM	Der Datumswert wird auf den ersten Tag des Monats gerundet. Wert ab dem 16ten des Monats werden von der round Funktion aufgerundet.	45
DAY, D	Der Datumswert wird auf den ersten Tag der Woche gerundet. Die round Funktion rundet Werte vom Donnerstag weg auf.	45
DD	Der Datumswert wird auf den Tag gerundet. Zeitangaben gehen verloren.	45
HH	Der Datumswert wird auf den Stundenanteil gerundet.	45
MI	Der Datumswert wird auf den Minutenanteil gerundet.	45

Abbildung 18. Rundungsparameter: round, trunc

4.2.6 Datumswerte umwandeln - to_char()

Die `to_char()` Funktion konvertiert Datumswerte in Zeichenketten.

► Parameter: `to_char` Funktion ▾

- **p_date:** Der Parameter beschreibt einen Datumswert.
- **p_time_mask:** Der Parameter beschreibt eine Formatmaske zur Formatierung eines Datumswertes.

► Syntax: `to_char` Funktion ▾

```

1  -- -----
2  -- Syntax: to_char
3  --
4  FUNCTION TO_CHAR(
5      P_DATE      IN DATE,
6      P_TIME_MASK IN VARCHAR2
7  )
8  RETURN VARCHAR2;
9
10 --
11 --  Datumswerte umwandeln
12 --
13 SELECT TO_CHAR( HIREDATE, 'dd.mm.yyyy')
14   HIREDATE
15  FROM EMPLOYEES
16 WHERE EMPLOYEE_ID = 7839;
17 -- Ausgabe: 10.09.2008

```

4.2.7 Rundungsfunktionen - `trunc()`, `round()`

Die SQL Spezifikation definiert 2 Funktionen zum Runden von Datumswerten: `trunc()` und `round()`.

Die `round` und `trunc` Funktionen unterscheiden sich ausschließlich in ihrem **Rundungsverhalten**. Beide Funktionen arbeiten mit denselben Parametern.

► Erklärung: `trunc`, `round` Funktion ▾

- Die `trunc()` Funktion wird verwendet um Datumswerte zu runden.
- Das Rundungsverhalten der Funktionen wird dabei über Parameterwerte gesteuert.
 - **Kein Rundungsparameter:** Das Datum wird auf '00:00:00' gesetzt.
 - **MM:** Das Datum wird auf den 1ten des Monats gerundet.
 - **Q:** Das Datum wird auf das **Quartal** zurückgesetzt in das das **Datum** fällt.
 - **Y:** Das Datum wird auf den **1ten** Tag des **Jahres** gerundet.
 - **DD:** Das Datum wird auf den gegenwärtigen Tag des Datums gerundet. Etwaige Zeitangaben gehen verloren.
- Die `round` und `trunc` Funktion sind bis auf ihr **Rundungsverhalten** identisch. Die `round` Funktion rundet im Gegensatz zur `trunc` Funktion kaufmännisch.

Funktion	Beschreibung	Seite
<code>add_months()</code>	Die add_months Funktion arbeitet analog zur Datumsarithmetik und erlaubt die Addition von Monaten zu einem Datumswert.	48
<code>extract()</code>	Die extract Funktion erlaubt die Extraktion bestimmter Teile aus einem Datum.	48
<code>last_day()</code>	Die last_day Funktion berechnet den letzten Tag des Monats in dem das Datum fällt.	48
<code>months_between()</code>	Die months_between Funktion berechnet die Differenz zweier Datumswerte. Der Wert wird dabei auf Monate gerundet. Dabei werden Schaltjahre und Monatslängen berücksichtigt.	49
<code>next_day()</code>	Die next_day Funktion berechnet für ein gegebenes Datum das Datum des nachfolgenden Tages.	49
<code>to_char()</code>	Die to_char() Funktion konvertiert eine Datumsangaben in eine Zeichenketten.	45

Abbildung 19. Datumsfunktionen

► Syntax: `trunc, round` Funktion ▾

```

1  --
2  -- Syntax: trunc(), round()
3  --
4  FUNCTION TRUNC (
5      P_DATE IN DATE,
6      P_MASK IN VARCHAR2 DEFAULT ''
7  )
8  RETURN DATE;
9
10 FUNCTION ROUND (
11     P_DATE IN DATE,
12     P_MASK IN VARCHAR2 DEFAULT ''
13 )
14 RETURN DATE;
15
16 --
17 -- trunc Funktion
18 --
19 SELECT
20     TRUNC(TO_DATE('12.05.2017'))    HIREDATE,
21     TRUNC(TO_DATE('12.05.2017'), 'MM') MONTH,
22     TRUNC(TO_DATE('12.05.2017'), 'Q') QUARTER,
23     TRUNC(TO_DATE('12.05.2017'), 'Y') YEAR
24 FROM DUAL;
25
26
27 -- Ausgabe
28 12.05.2017 00:00:00 -- hiredate
29 01.05.2017 00:00:00 -- month
30 01.04.2017 00:00:00 -- quarter
31 01.01.2017 00:00:00 -- year

```

4.2.8 Erzeugen von Intervallen

Die SQL Spezifikation erlaubt neben dem Verarbeiten von Datumswerten auch das Arbeiten mit **Zeitintervallen**.

► Erklärung: Erzeugen von Intervallen ▾

- Die SQL Spezifikation erlaubt neben der Berechnung von Zeitangaben auch das Arbeiten mit Zeitintervallen.
- Zeitintervalle werden durch die Angabe von Literalen definiert.

► Query: Erzeugen von Intervallen ▾

```

1  --
2  -- Erzeugen von Intervallen
3  --
4  SELECT INTERVAL '3' YEAR JAHRE,
5      INTERVAL '32' MONTH MONATE,
6      INTERVAL '64' DAY TAGE,
7      INTERVAL '36' HOUR STUNDEN,
8      INTERVAL '105' MINUTE MINUTEN,
9      INTERVAL '635' SECOND SEKUNDEN
10 FROM DUAL;
11
12 -- Ausgabe
13 3-0          -- Jahre
14 1-0          -- Monate
15 64 0:0:0.0   -- Tage
16 1 12:0:0.0   -- Stunden
17 0 1:45:0.0   -- Minuten
18 0 0:10:35.0 -- Sekunden

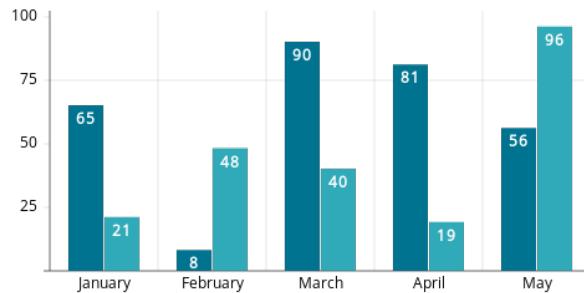
```

Intervall	Beschreibung	Seite
year to month	Das Literal repräsentiert ein Zeitintervall aus Jahren und Monaten.	
day to minute	Das Literal repräsentiert ein Zeitintervall. Der Wert beschreibt ein Zeitintervall von Tagen, Stunden und Minuten.	
day to second	Das Literal repräsentiert ein Zeitintervall. Der Wert beschreibt ein Zeitintervall von Tagen, Stunden, Minuten und Sekunden.	

Abbildung 20. Datumsfunktionen

4.2.9 Geschachtelte Intervalle

Die SQL Spezifikation erlaubt die Definition geschachtelter Intervalle. Die Intervalle werden als **Literale** definiert.



► Query: Geschachtelten Intervallen ▾

```

1  -- -----
2  -- Geschachtelte Intervalle
3  --
4  SELECT
5  INTERVAL '2-11'  YEAR TO MONTH INTERVALL_1,
6  INTERVAL '4 3:30' DAY TO MINUTE INTERVALL_2,
7  INTERVAL
8    '4 3:30:45' DAY TO SECOND INTERVALL_3
9  FROM DUAL;
10
11 -- Ausgabe
12 2-11      -- intervall_1
13 3:30:0.0   -- intervall_2
14 3:30:45.0  -- intervall_3

```

4.2.10 Datumsarithmetik

Das Rechnen mit **Datumswerten** ist dann leicht, wenn wir uns vergegenwärtigen, daß ein Datum eine Zahl darstellt mit einem **Ganzzahlanteil**, der die Anzahl

der Tage seit einem Startdatum wiedergibt, und einen **Nachkommaanteil**, der den Anteil des Tages benennt, der bereits vergangen ist.

► Erklärung: Datumsarithmetik ▾

- Die SQL Spezifikation erlaubt das **Rechnen** mit **Datumswerten**. Durch die Angabe von Intervallen können einfache Berechnungen mit Datumswerten durchgeführt werden.
- Dabei kümmert sich die **Datenbankengine** um Fiddigkeiten wie wieviele Tage ein bestimmtes Monat hat oder ob ein Jahr ein Schaltjahr ist.
- Die Datumsarithmetik unterstützt dabei die Addition bzw. die Subtraktion von Datumswerten ausgehend von einem bestimmten Zeitpunkt.
- Die **Addition** ist dabei eine arithmetische Operation auf einem Datumswert und einem Zeitintervall.
- Die **Subtraktion** ist die einzige sinnvolle direkte arithmetische Operation auf 2 Datumswerten.
 - Im Fall der Subtraktion des Datentyps **date** ist das Ergebnis eine Zahl.
 - Im Fall der Subtraktion des Datentyps **timestamp** ein Intervall.

► Query: Datumsarithmetik ▾

```

1  -- -----
2  -- Datumsarithmetik
3  --
4  SELECT TRUNC(SYSDATE)
5    + INTERVAL '3' MONTH
6    + INTERVAL '1' DAY
7    + INTERVAL '1' DAY
8    + INTERVAL '2 15:20' DAY TO MINUTE
9  FROM DUAL;

```

4.2.11 Datumsfunktion: add_months()

Die `add_months()` Funktion arbeitet analog zur **Datumsarithmetik** und erlaubt die Addition von Monaten zu einem Datumswert.

► Parameter: add_months Funktion ▾

- Die Funktion erwartet 2 Parameter. `add_months` arbeitet mit Datenwerten vom Typ `Date` bzw. `Timestamp` und liefert als Ergebnis jedoch immer einen Wert vom Typ `Date` zurück.
- Ein negativer Wert der **Monatsangabe** subtrahiert die entsprechende Anzahl von Monaten.

► Syntax: add_months Funktion ▾

```

1  -- -----
2  -- Syntax: add_months
3  --
4  FUNCTION ADD_MONTHS (
5      P_DATE IN      DATE,
6      P_MONTH_AMOUNT IN INTEGER
7  )
8  RETURN DATE;
9
10 --
11 -- Aufruf add_months
12 --
13 SELECT ADD_MONTHS(TRUNC(SYSDATE), 3)
14 FROM DUAL;
```



4.2.12 Datumsfunktion: next_day()

Die `next_day()` Funktion berechnet für ein gegebenes Datum das Datum des nachfolgenden Tages.

► Syntax: next_day Funktion ▾

```

1  -- -----
2  -- Syntax: next_day
3  --
4  FUNCTION NEXT_DAY (
5      P_DATE IN DATE,
6      P_DAY  IN VARCHAR2
7  )
8  RETURN DATE_EXP;
9
10 SELECT NEXT_DAY(SYSDATE, 'Freitag')
11 FROM DUAL;
```



4.2.13 Datumsfunktion: extract()

Die `extract()` Funktion ermöglicht die Extraktion bestimmter Teile eines Datumswertes.

► Syntax: extract Funktion ▾

```

1  -- -----
2  -- Syntax: extract
3  --
4  FUNCTION EXTRACT (
5      P_PARAM IN  EXPR
6  )
7  RETURN INTEGER;
8
9  --
10 -- Datumsfunktion: extract
11 --
12 SELECT EXTRACT(YEAR FROM SYSDATE) YEAR,
13       EXTRACT(MONTH FROM SYSDATE) MONTH,
14       EXTRACT(DAY   FROM SYSDATE) DAY
15  FROM DUAL;
16
17 2017    -- year
18 7        -- month
19 30       -- day
```



4.2.14 Datumsfunktion: last_day()

Die `last_day()` Funktion berechnet den letzten Tag des Monats in den der angegebene Datumswert fällt.

► Syntax: last_day Funktion ▾

```

1  -- -----
2  -- Syntax: last_day
3  --
4  FUNCTION LAST_DAY (
5      P_DATE IN DATE
6  )
7  RETURN DATE;
8
9  --
10 -- Datumsfunktion: last_day
11 --
12 SELECT LAST_DAY(SYSDATE) DATE_1,
13       EXTRACT(DAY FROM LAST_DAY(SYSDATE)) DATE_2
14  FROM DUAL;
15
16 31.7.2017 15:11:24 -- date_1
17          31           -- date_2
```



4.2.15 Datumsfunktion: next_day()

Die `next_day()` Funktion berechnet für ein gegebenes Datum, das Datum des nachfolgenden Tages.

► Syntax: next_day Funktion ▾

```
1  -- -----
2  -- Syntax: next_day
3  --
4  FUNCTION NEXT_DAY (
5      P_DATE_1 IN DATE,
6      P_DAY     IN VARCHAR2
7  )
8  RETURN DATE_EXP;
9
10 -- AUFRUF
11 SELECT NEXT_DAY(SYSDATE, 'Freitag')
12 FROM DUAL;
```



4.2.16 Datumsfunktion: months_between()

Die `months_between()` Funktion berechnet wieviele Monate zwischen 2 Datumswerten liegen. Dabei werden Schaltjahre und Monatslängen berücksichtigt.

► Parameter: months_between Funktion ▾

- Als Parameter erwartet die Funktion zwei Datumsangaben, wobei das spätere Datum als erstes übergeben werden sollte.
- Die Funktion eignet sich in erster Linie zur Berechnung von Zeitdauern.

► Syntax: months_between Funktion ▾

```
1  -- -----
2  -- Syntax: months_between
3  --
4  FUNCTION MONTHS_BETWEEN (
5      P_DATE_1 IN DATE,
6      P_DATE_2 IN DATE
7  )
8  RETURN DATE_EXP;
9
10 -- AUFRUF
11 SELECT TRUNC(
12     MONTHS_BETWEEN( SYSDATE, HIREDAT )
13 )
14 FROM EMPLOYEES;
```



4.3. Textfunktionen

Die SQL Spezifikation definiert eine Zahl von Funktionen zur Verarbeitung von Zeichenketten.

4.3.1 Textfunktion: instr()

Die `instr()` Funktion prüft, ob ein **Token** in einer Zeichenkette enthalten ist und gibt die Position bei einer Übereinstimmung zurück.

► Parameter: instr() Funktion ▾

- **p_content:** Der Parameter beschreibt die zu durchsuchende Zeichenkette.
- **p_token:** Der Parameter beschreibt den Token nach dem gesucht wird.
- **p_start_index:** Der Parameter beschreibt die Position ab der die Zeichenkette durchsucht werden soll. Der Parameter ist optional und ist defaultmäßig mit dem Wert 1 initialisiert.
- **p_count:** Der Parameter definiert welches Vorkommen des Tokens ermittelt werden soll. Der Parameter ist optional und ist defaultmäßig mit dem Wert 1 initialisiert.

► Syntax: instr Funktion ▾

```

1  -- -----
2  -- Syntax: instr
3  --
4  FUNCTION INSTR (
5      P_CONTENT      IN VARCHAR2,
6      P_TOKEN        IN VARCHAR2,
7      P_START_INDEX  IN INTEGER DEFAULT 1,
8      P_COUNT        IN INTEGER DEFAULT 1
9  )
10 RETURN NUMBER;
11 --
12 -- Textfunktion: instr
13 --
14 SELECT
15     INSTR('/home/foo.txt', '/', 1, 2) POS_2
16     INSTR('/home/foo.txt', '/') POS_1,
17 FROM DUAL;
18
19 -- Ausgabe
20 6  -- pos_2
21 1  -- pos_1

```

4.3.2 Textfunktionen: trim()

Die `trim()` Funktion wird verwendet um **Token** am Ende bzw. am Anfang einer Zeichenkette zu entfernen. Die SQL Spezifikation definiert mehrere Formen der `trim()` Funktion.

► Syntax: trim(), ltrim(), rtrim() ▾

```

1  -- -----
2  -- Syntax: trim, ltrim, rtrim
3  --
4  FUNCTION ...TRIM... (
5      P_TEXT    IN VARCHAR2,
6      P_TOKEN   IN VARCHAR2
7  )
8  RETURN VARCHAR2;
9
10 --
11 -- Textfunktion: lefttrim, righttrim
12 --
13 SELECT LTRIM('<Das ist ein Element/>', '<')
14 FROM DUAL;
15
16 -- Ausgabe
17 Das ist ein Element/>
18
19 -- rtrim
20 SELECT RTRIM('<Das ist ein Element/>', '/>')
21 FROM DUAL;
22
23 -- Ausgabe
24 <Das ist ein Element
25
26 --
27 -- Textfunktion: trim
28 --
29 SELECT
30     TRIM(BOTH    ' ' FROM '...SMITH...') TEXT_1,
31     TRIM(LEADING ' ' FROM '...SMITH...') TEXT_2,
32     TRIM(TRAILING ' ' FROM '...SMITH...') TEXT_3
33 FROM dual;
34
35 -- Ausgabe
36 SMITH      -- text_1
37 SMITH...    -- text_2
38 ...SMITH   -- text_3
39
40 SELECT TRIM(' SMITH ') FROM DUAL;
41
42 -- Ausgabe
43 SMITH

```

Funktion	Beschreibung	Seite
instr()	Die instr() Funktion prüft, ob ein Token in einer Zeichenkette enthalten ist und gibt die Position bei einer Übereinstimmung an.	50
length()	Mit der length Funktion wird die Länge einer Zeichenkette berechnet. Der Rückgabewert ist dabei als die Anzahl von bytes bzw. die Anzahl von Zeichen zu lesen.	51
lower()	Mit der lower() und upper() Funktion werden Zeichenfolgen transformiert.	52
replace()	Mit der replace Funktion wird eine Zeichenkette in einem Text durch eine andere Zeichenkette ersetzt.	52
soundex()	Die soundex Funktion führt einen Vergleich zwischen Zeichenketten durch. Der Vergleich bewertet die phonetische Ähnlichkeit von Zeichenketten.	51
substr()	Die substr Funktion wird verwendet um aus einer Zeichenkette einen Teilstring zu extrahieren.	52
trim()	Mit den unterschiedlichen Formen der trim Funktion werden am Ende bzw. Anfang einer Zeichenkette Zeichenfolgen entfernt.	50
upper()	Mit der lower() und upper() Funktion werden Zeichenfolgen transformiert.	52

Abbildung 21. Textfunktionen

4.3.3 Textfunktion: length()

Mit Hilfe der **length()** Funktion wird die Länge einer Zeichenkette bestimmt. Der Rückgabewert kann dabei in Form von bytes bzw. als Anzahl von Zeichen ausgegeben werden.

► Syntax: length Funktion ▼

```

1  -- -----
2  -- Syntax: length
3  --
4  FUNCTION LENGTH (
5      P_TEXT IN VARCHAR2
6  )
7  RETURN INTEGER;
8
9  --
10 -- Textfunktion: length
11 --
12 SELECT
13     LENGTH ('Toromtomtom') ZEICHEN,
14     LENGTHB('Toromtomtom') BYTE
15 FROM DUAL;
16
17 -- Ausgabe
18 11  -- zeichen
19 15  -- byte

```

4.3.4 Textfunktion: soundex()

Mit der **soundex()** Funktion wird die phonetische Kennzahl einer Zeichenkette bestimmt werden.

► Syntax: soundex Funktion ▼

```

1  --
2  -- Syntax: soundex
3  --
4  FUNCTION SOUNDEX (
5      P_TEXT IN VARCHAR2
6  )
7  RETURN VARCHAR2;
8
9
10 --
11 -- Textfunktion: soundex
12 --
13 SELECT LAST_NAME, JOB_ID
14 FROM EMPLOYEES
15 WHERE SOUNDEX_GER(LAST_NAME) IN (
16     SOUNDEX_GER('Meier'),
17     SOUNDEX_GER('Mayer'),
18     SOUNDEX_GER('Meyer'),
19     SOUNDEX_GER('Mair'),
20     SOUNDEX_GER('Maier'),
21     SOUNDEX_GER('Mayr'),
22     SOUNDEX_GER('Meir')
23 );

```

4.3.5 Textfunktionen: lower(), upper()

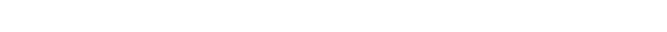
Mit der `lower()` bzw. `upper()` Funktion können Zeichenketten transformiert werden.

► Syntax: `lower()`, `upper()`, `initcap()` ▾

```

1  -- -----
2  -- Syntax: lower, upper
3  --
4  FUNCTION LOWER (
5      P_TEXT IN VARCHAR2
6  )
7  RETURN VARCHAR2;
8
9  --
10 -- Textfunktion: lower, upper
11 --
12 SELECT LOWER(LAST_NAME)      L_NAME,
13          UPPER(LAST_NAME)      U_NAME,
14          INITCAP (LAST_NAME)  IC_NAME
15 FROM EMPLOYEES
16 WHERE LOWER(LAST_NAME) = LOWER('Miller');
17
18 -- Ausgabe
19 miller  -- lower
20 MILLER  -- upper
21 Miller   -- initcap

```



4.3.6 Textfunktion: replace()

Mit der `replace()` Funktion wird ein Token in einem Text durch einen anderen Token ersetzt. Das gilt für jedes Vorkommen der Zeichenkette im Text. Der Token, der als Ersatzzeichen angegeben werden kann ist optional. Wird er nicht angegeben wird der Token lediglich aus dem Text entfernt.

► Query: `replace` Funktion ▾

```

1  -- -----
2  -- Textfunktion: replace
3  --
4  SELECT REPLACE(
5      'Katalog 2014: ICD_2014', '2014', '2015',
6  )
7  FROM DUAL;
8
9  -- Ausgabe
10 Katalog 2015: ICD_2015

```



```

1  -- -----
2  -- replace ohne Ersatzzeichen
3  --
4  SELECT REPLACE('SALEMAN', 'MAN')
5  FROM DUAL;
6
7  -- Ausgabe
8  SALE

```



4.3.7 Textfunktion: substr()

Die `substr` Funktion wird verwendet um aus einer **Zeichenkette** einen **Teilstring** zu extrahieren.

► Parameter: `substr` Funktion ▾

- Die `substr()` Funktion erwartet 3 Parameter: `p_text`, `p_begin` und `p_length`.
- Der `p_length` Parameter ist optional. Wird er weggelassen, wird der String bis zum Ende der Zeichenkette extrahiert.

► Syntax: `substr` Funktion ▾

```

1  -- -----
2  -- Syntax: substr
3  --
4  FUNCTION SUBSTR(
5      P_TEXT    IN VARCHAR2,
6      P_BEGIN   IN INTEGER,
7      P_LENGTH  IN INTEGER DEFALUT LENGTH(P_TEXT)
8  )
9  RETURN VARCHAR2;
10
11 --
12 -- Textfunktion: substr
13 --
14 SELECT SUBSTR('Das ist ein Text', 9, 3)
15 FROM DUAL;
16
17 -- Ausgabe
18 ein
19
20 --
21 -- substr ohne 3ten Parameter
22 --
23 SELECT SUBSTR('Das ist ein Text', 9)
24 FROM DUAL;
25
26 -- Ausgabe
27 ein Text

```



Element	Bemerkung	Beispiel
G	Liefert den Tausendertrenner. Im deutschen Sprachraum wird das Komma verwendet. 9G999G999	
, (Komma)	Wird als Tausendertrenner nach amerikanischem Standard verwendet und kann mehrfach vorhanden sein, allerdings weder als erstes Zeichen noch rechts vom Dezimaltrenner bzw. Punkt.	9,999,999
D	Liefert das Dezimaltrennzeichen. Im deutschen Sprachraum ist das ein .	9G999D00
. (Punkt)	Wird als Dezimaltrenner nach amerikanischem Standard verwendet und darf daher nur einmal vorkommen.	9,999.00
C	Liefert das ISO Währungssymbol an der angegebenen Stelle	C9G990D00
\$	Liefert die Zahl mit einem führenden Dollarzeichen	\$9,990.00
9	Optionale Ziffer. Ist die Ziffer dieser Position nicht vorhanden, wird sie nicht ausgegeben.	9,99
0	Verpflichtend auszugebende Ziffer. Ist die Ziffer dieser Position nicht vorhanden, wird eine 0 ausgegeben.	0.00
FM	Entfernt Leerzeichen aus der Zeichenkette.	FM99G999
FM	Entfernt Leerzeichen aus der Zeichenkette.	FM99G999
FM	Entfernt Leerzeichen aus der Zeichenkette.	FM99G999
B	Liefert Leerzeichen falls die Ziffern des Ganzahlanteils nicht vorhanden sind.	BS99G999

Abbildung 22. Formatoption für numerische Konvertierungsfunktionen

4.4. Numerische Funktionen

Numerische Funktionen werden zur Verarbeitung numerischer Werte in SQL verwendet.

4.4.1 Rundungsfunktionen

Zum **Runden** numerischer Werte stellt die SQL Spezifikation 2 Funktionen zur Verfügung: **trunc** und **round**.

► Analyse: Rundungsfunktionen

- Mit der **trunc** Funktion werden numerische Werte abgerundet, während **round** nach **kaufmännischem Verfahren** auf- bzw. abrundet.
- Der 2te Parameter spezifiziert die Genauigkeit des Rundungsvorgangs. Dazu wird die Anzahl der Nachkommastellen angegeben.
- Zusätzlich kann für den 2ten Parameter ein negativer Wert übergeben werden, wodurch die Funktion auf ganze Zehner, Hunderter bzw. Tausender etc. rundet.

► Syntax: round Funktion

```

1  --
2  -- Syntax: round
3  --
4  FUNCTION ROUND (
5      P_NUMBER IN NUMBER,
6      P_ROUND   IN PLS_INTEGER
7  )
8  RETURN VARCHAR2;
9
10 SELECT ROUND(12345.678) N_1,
11      ROUND(12345.678, 1) N_2,
12      ROUND(12345.678, 2) N_3,
13      ROUND(12345.678, -2) N_6,
14  FROM DUAL;
15
16  -- Ausgabe
17  12346    -- n_1
18  12345.7  -- n_2
19  12345.68 -- n_3
20  12300    -- n_6

```

► Syntax: trunc Funktion ▾

```

1  -- -----
2  --   Syntax: trunc
3  --
4  FUNCTION TRUNC (
5      P_NUMBER  IN NUMBER,
6      P_ROUND    IN PLS_INTEGER
7  )
8  RETURN VARCHAR2;
9
10 --
11 --   Syntax: trunc
12 --
13 SELECT TRUNC(12345.678) N_1,
14      TRUNC(12345.678, 1) N_2,
15      TRUNC(12345.678, 2) N_3,
16      TRUNC(12345.678, -2) N_6,
17      TRUNC(12345.678, -1) N_7
18 FROM DUAL;
19
20
21 -- Ausgabe
22 12345      -- n_1
23 12345.6     -- n_2
24 12345.67    -- n_3
25 12300      -- n_6
26 12340      -- n_7

```

► Query: to_number Funktion ▾

```

1  -- -----
2  --   Syntax: to_number
3  --
4  FUNCTION TO_NUMBER (
5      P_NUMBER_LITERAL IN VARCHAR2,
6      P_MASK          IN VARCHAR2
7  )
8  RETURN NUMBER;
9
10 --
11 --   Konvertierungsfunktion: to_number
12 --
13 SELECT
14      TO_NUMBER('123.45,21', '999G99D99') N_3
15      TO_NUMBER('12345')   N_1,
16      TO_NUMBER('123,45')  N_2,
17  FROM DUAL;
18
19
20 -- Ausgabe
21 12345      -- n_1
22 123.45     -- n_2
23 12345.45   -- n_3

```



4.4.2 Konvertierungsfunktionen

Zum Konvertieren numerischer Wert in **Zeichenketten** wird die **to_char** Funktion verwendet. Dazu werden durch den User **Formatmasken** definiert.

► Query: to_char Funktion ▾

```

1  -- -----
2  --   Konvertierungsfunktion: to_char
3  --
4  SELECT
5      TO_CHAR(123456.89, '999G999D99') S_1,
6      TO_CHAR(12345.67, '999G999D99')  S_2,
7      TO_CHAR(12345.67, '999G999D99C') S_3
8  FROM DUAL;
9
10
11 -- Ausgabe
12 123.456,12    -- s_1
13 12.345,67     -- s_2
14 12.345,67EUR -- s_3

```

4.4.3 Numerische Funktion: abs()

Die **abs** Funktion gibt den absoluten Wert einer Zahl zurück.

► Query: abs Funktion ▾

```

1  -- -----
2  --   Syntax: abs
3  --
4  FUNCTION ABS (
5      P_VALUE IN NUMBER
6  )
7  RETURN NUMBER;
8
9  --
10 -- Example: abs
11 --
12 SELECT
13      ABS(-4) N_1, ABS( 4) N_2
14  FROM DUAL;
15
16 4  -- n_1
17 4  -- n_2

```



Funktion	Beschreibung	Seite
<code>round()</code> , <code>trunc()</code>	Funktionen zum Runden numerischer Werte	53
<code>to_char()</code>	Funktion zum Konvertieren numerischer Werte in Zeichenketten.	54
<code>to_number()</code>	Funktion zum Konvertieren von Zeichenketten in numerische Werte.	54
<code>abs()</code>	Die <code>abs</code> Funktion gibt den absoluten Wert einer Zahl zurück	54
<code>mod()</code>	Die <code>mod</code> Funktion gibt den Rest von m geteilt durch n als Ergebnis zurück.	55
<code>exp()</code>	Die <code>exp</code> Funktion berechnet die mathematische Exponentialfunktion für einen numerischen Wert.	55
<code>log()</code>	Die <code>log</code> Funktion gibt den Logarithmus von n zur Basis m zurück.	55

Abbildung 23. Numerische Funktionen

4.4.4 Numerische Funktion: `exp()`, `log()`

Die `exp` Funktion berechnet die mathematische Exponentialfunktion für einen numerischen Wert. Die `log` Funktion gibt den Logarithmus von n zur Basis m zurück.

► Query: `exp`, `log` Funktion ▾

```

1  -- -----
2  -- Syntax: exp, log
3  --
4  FUNCTION EXP (
5      P_M IN NUMBER
6  )
7  RETURN NUMBER;
8
9  FUNCTION LOG (
10     P_N IN NUMBER,
11     P_M IN NUMBER
12  )
13 RETURN NUMBER;
14
15 --
16 -- Example: exp, log
17 --
18 SELECT
19     EXP(3) N_1,
20     LOG(10, 20) N_2,
21     LOG(100, 1) N_3
22 FROM DUAL;
23
24 -- Ergebnis
25 20.0855369 -- n_1
26 1.30203999 -- n_2

```

4.4.5 Numerische Funktion: `mod()`

Die `mod` Funktion gibt den Rest von m geteilt durch n als Ergebnis zurück.

► Query: `mod` Funktion ▾

```

1  -- -----
2  -- Syntax: mod
3  --
4  FUNCTION MOD (
5      P_M IN NUMBER,
6      P_N IN NUMBER
7  )
8  RETURN NUMBER;
9
10 --
11 -- Example: mod
12 --
13 SELECT
14     MOD(15,    4) N_1,
15     MOD(15,    3) N_2,
16     MOD(15,    0) N_3,
17     MOD(11.6,  2) N_4,
18     MOD(-15,   4) N_5,
19     MOD(-15,   0) N_6,
20     FROM DUAL;
21
22 -- Ergebnis
23 3  -- n_1
24 0  -- n_2
25 15 -- n_3
26 1.1 -- n_4
27 -3 -- n_5
28 -15 -- n_6

```

5. SQL - Aggregatfunktionen

04

Gruppenfunktionen

01. Aggregatfunktionen	56
02. Group By Klausel	58

5.1. Aggregatfunktionen

Funktionen, die eine Menge von Werten zu einem einzelnen Wert verdichten, werden als Aggregatfunktionen bezeichnet.



Aggregation ▾

Aggregation bezeichnet das Zusammenfassen einer Reihe von Werten zu einem einzelnen Wert.

Beispielsweise lässt sich aus einer Menge von Zahlen der Mittelwert, das Minimum bzw. das Maximum oder die Summe der Werte bestimmen.

5.1.1 Aggregatfunktionen

Im Gegensatz zu einer Zeilenfunktion, die für einen Datensatz einen einzelnen Wert berechnet, verdichtet eine Aggregatfunktion mehrere Datensätze zu einem einzelnen Wert.

► Erklärung: Einfache Aggregatfunktionen ▾

- Die SQL Spezifikation definiert eine Reihe von Aggregatfunktionen: `avg`, `max`, `min`, `sum` und `count`.
- Aggregatfunktionen werden in der Regel in Kombination mit der `group by` Klausel verwendet.

```

1  -- -----
2  -- Aggregatfunktionen
3  -- -----
4  -- Berechnet die Anzahl aller Angestellen
5  SELECT COUNT(EMPLOYEE_ID) FROM EMPLOYEES;
6
7  -- Berechnet das kleinste Gehalt das im
8  -- Unternehmen ausbezahlt wird
9  SELECT MIN(SALARY) FROM EMPLOYEES;
10
11 -- Berechnet das hoechste Gehalt das im
12 -- Unternehmen ausbezahlt wird
13 SELECT MAX(SALARY) FROM EMPLOYEES;
14
15 -- Berechnet das Durchschnittsgehalt
16 -- das im Unternehmen ausbezahlt wird
17 SELECT AVG(SALARY) FROM EMPLOYEES;
```



5.1.2 Median



Median ▾

Der **Median** oder **Zentralwert** ist einer der statistischen Mittelwerte.

Zur Berechnung des Medians einer Menge von Zahlen, werden die Werte zunächst sortiert und dann der **mittlere** dieser Werte gewählt.

► Query: Median berechnen ▾

```

1 -- -----
2 -- Median
3 -- -----
4 SELECT
5   MEDIAN(COALESCE(SALARY, 0)) MEDIAN
6 FROM EMPLOYEES;

```

5.1.3 Standardabweichung



Standardabweichung ▾

Die **Standardabweichung** einer Menge gibt an wie stark die Werte der Menge von deren Mittelwert abweichen.

Die Standardabweichung ist ein **Streuungswert** der Statistik.

► Erklärung: Standardabweichung ▾

- Bei einer Menge von Zahlen, die um den Mittelwert herum angeordnet sind, ist es oft von Interesse zu wissen, wie nahe diese Werte am Mittelwert bleiben.
- Als Mittelwert wird in diesem Zusammenhang das **arithmetische Mittel** verwendet.
- Bleiben die Werte dicht am Mittelwert, ist die **Streuung** der Datenwerte gering.
- Der Abstand der Werte vom Mittelwert, wird als Quadrat der Differenz der Einzelwerte berechnet.

```

1 -- -----
2 -- Standardverteilung
3 -- -----
4 SELECT STDEV(SALARY)
5 FROM EMPLOYEES;

```

5.1.4 Aggregatfunktionen und NULL Werte

Bei der Verdichtung von Datensätzen zu einem einzelnen Wert werden **null Werte** ignoriert.

Damit können Aggregatfunktionen auch auf Spalten angewandt werden die null Werte enthalten.

► Query: Aggregatfunktionen und NULL Werte ▾

```

1 -- -----
2 -- Aggregatfunktionen und NULL Werte
3 -- -----
4 SELECT AVG(SALARY), MAX(SALARY)
5 FROM EMPLOYEES;

```

5.1.5 UNIQUE Operator

Für Aggregatfunktionen muß entschieden werden ob alle oder nur alle unterschiedlichen Werte einer Spalte für die Berechnung eines Ergebnisses herangezogen werden sollen.

► Erklärung: Werte einer Spalte zählen ▾

- Durch die Verwendung des **distinct** Schlüsselworts in der select Klausel, wird die **Datenbankengine** angehalten, alle redundanten Datensätze aus dem Ergebnis einer Abfrage zu entfernen.
- Da die group Klausel jedoch vor der select Klausel ausgewertet wird, hat die Verwendung des **distinct** Schlüsselworts keinen Auswirkung auf den Datenbestand der group Klausel.

```

1 -- -----
2 -- distinct operator
3 -- -----
4 SELECT COUNT(DISTINCT JOB) BERUFE
5 FROM EMP;

```

- Sollen nur die disjunktten Werte einer Spalte für die Auswertung einer Aggregatfunktion herangezogen werden, muß das **unique** Schlüsselwort verwendet werden.

```

1 -- -----
2 -- unique operator
3 -- -----
4 SELECT COUNT(UNIQUE JOB) BERUFE
5 FROM EMP;

```

5.2. Group by Klausel

Die **group by** Klausel führt eine **Neustrukturierung** der Daten einer Abfrage durch.

Bislang ist es zwar ganz interessant, das höchste Gehalt bzw. das Durchschnittsgehalt aller Angestellten im Unternehmen zu berechnen, vielmehr würde uns aber der Vergleich der Gehälter der einzelnen Abteilungen im Unternehmen interessieren.

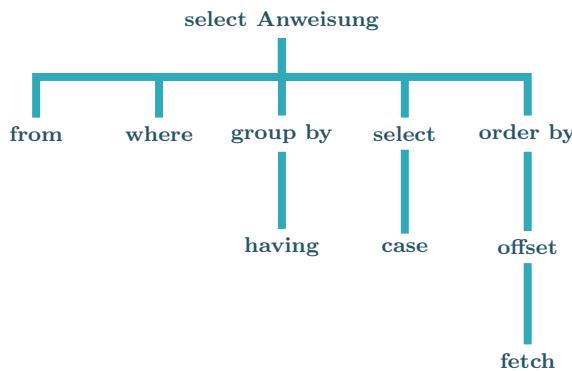
5.2.1 Neustrukturierung von Daten



Group by Klausel ▾

Die group by Klausel führt eine **Neustrukturierung** der Daten einer Abfrage durch.

► Erklärung: Ausführungsreihenfolge der Klauseln ▾



► Beispiel: Group By Klausel ▾

- Wir möchten für jede Abteilung die Anzahl der Mitarbeiter, die dort arbeiten, bestimmen.
- Mit der gegenwärtigen **Struktur** des Datenbestands ist eine Lösung der Aufgabe nicht möglich.
- Zur Lösung der Aufgabe wäre es notwendig die Angestellten jeder Abteilung in einer eigenen Tabelle zu speichern. Die Anzahl der Mitarbeiter pro Abteilung könnte dann durch das Aufsummieren der Datensätze der Tabellen, ermittelt werden.
- Die group by Klausel ermöglicht eine entsprechende **Neustrukturierung** der Daten.

5.2.2 Group by Klausel

Mit der group by Klausel kann eine Neustrukturierung der Daten einer Abfrage durchgeführt werden.

► Erklärung: Neustrukturierung von Daten ▾

- Die durch die group by Klausel angestoßene Neustrukturierung der Daten einer Abfrage erfolgt in 2 Schritten: der **Map Phase** und der **Aggregate Phase**.
- **Map Phase:** In der Map Phase werden die Daten der Datenbasis auf disjunkte Gruppen verteilt.
- **Aggregate Phase:** In der Aggregate Phase werden die Daten jeder Gruppe zu einem einzelnen Datensatz verdichtet. Die Menge aller so bestimmten Datensätze wird zur Datenbasis der Abfrage. Die Struktur des Datenbestandes wurde damit einer Neustrukturierung unterzogen.
- Nach dem Abschluss der Aggregate Phase ist die Neustrukturierung der Daten abgeschlossen.



5.2.3 Map Phase



Map Phase ▾

In der Map Phase werden die Daten einer SQL Abfrage auf disjunkte **virtuelle Tabellen** verteilt.

► Erklärung: Map Phase ▾

- Wir möchten für jede Abteilung eines Unternehmens, die Anzahl der dort beschäftigten Mitarbeiter, bestimmen.
- Für jede Abteilung legt die Datenbankengine eine eigene **virtuelle Tabelle** an. Die Angestelltendaten werden nun in die virtuellen Tabellen der zugehörigen Abteilungen übernommen.
- Nach der Map Phase liegen die Angestelltendaten gruppiert auf ihre Abteilungen vor.
- Mit der group by Klausel wird bestimmt nach welchen Kriterien der Datenbestand nun gruppiert werden soll.

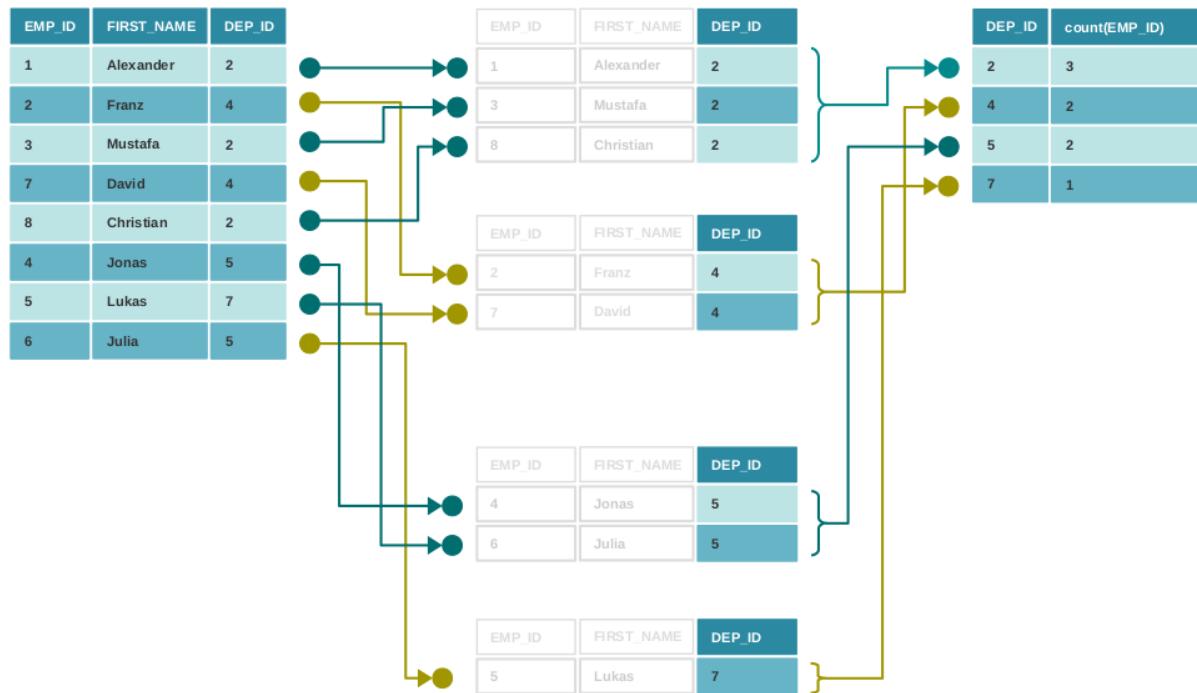


Abbildung 24. Group by: Neustrukturierung von Daten

5.2.4 Group by Klausel

Mit der group by Klausel wird bestimmt nach welchen Kriterien der Datenbestand gruppiert werden soll.

► Erklärung: Group by Klausel ▼

- Die **disjunkten Werte**, der in der group by Klausel angegebenen Spalten, definieren die Gruppen auf die der Datenbestand verteilt werden soll.
- Wird in einer Abfrage beispielsweise nach der **department_id** Spalte gruppiert, erfolgt eine **Verteilung der Datensätze** auf die unterschiedlichen Abteilungen des Unternehmens.

```

1  -- -----
2  -- Group by Klausel
3  -- -----
4  SELECT DEPARTMENT_ID, COUNT(EMPLOYEE_ID),
5      AVG(SALARY)
6  FROM EMPLOYEES
7  GROUP BY DEPARTMENT_ID;

```

- Die Spaltenbezeichner der group by Klausel definieren dabei die Kriterien für die Neustrukturierung der Daten.

► Query: Group by Klausel ▼

```

1  --
2  -- Group by Klausel
3  --
4  -- Wir moechten nun wissen, wieviele Mitarbeiter
5  -- in den einzelnen Abteilungen, der unterschiedlichen Laender, arbeiten.
6
7  SELECT DEPARTMENT_ID, COUNTRY_ID
8      COUNT(EMPLOYEE_ID) EMPLOYEE_COUNT,
9      MAX(SALARY) MAX_SALARY,
10     MIN(SALARY) MIN_SALARY,
11     SUM(SALARY) DEPARTMENT_COST
12  FROM EMPLOYEES
13  JOIN DEPARTMENTS USING(DEPARTMENT_ID)
14  JOIN REGIONS USING(REGION_ID)
15  GROUP BY COUNTRY_ID, DEPARTMENT_ID;

```



5.2.5 Semantik der Gruppenbildung

Wird der Datenbestand einer Abfrage nach mehreren **Kriterien** gruppiert, die im **Kontext** der Anwendung denselben Sachverhalt beschreiben, wird für die **Gruppierung** nur eines der Kriterien herangezogen.

► Query: Semantik der Gruppenbildung ▾

```

1  --
2  --   Semantik der Gruppenbildung
3  --
4  --   Die Spalten department_id und depart-
5  --   ment_name beschreiben im Kontext der
6  --   Anwendung denselben Sachverhalt.

7
8  SELECT DEPARTMENT_ID, DEPARTMENT_NAME
9  FROM EMPLOYEES
10 NATURAL JOIN DEPARTMENTS
11 GROUP BY DEPARTMENT_ID, DEPARTMENT_NAME;
```

5.2.6 Fallbeispiel: Gruppierung

SQL Abfragen mit einer group by Klausel.

► Query: Group by Klausel ▾

```

1  --
2  --   Anzahl der Angestellten pro Abteilung
3  --
4  SELECT DEPARTMENT_ID, COUNT(EMPLOYEE_ID)
5          AVG(SALARY)
6  FROM EMPLOYEES
7  GROUP BY DEPARTMENT_ID;

8
9  --
10 -- Anzahl der Angestellten fuer jedes Land
11 --
12 SELECT COUNTRY_ID, COUNT(EMPLOYEE_ID)
13          AVG(SALARY)
14  FROM EMPLOYEES
15  GROUP BY COUNTRY_ID;

16
17 --
18 -- Anzahl der Angestellten pro Abteilung
19 -- fuer jedes Land
20 --
21 SELECT COUNTRY_ID, COUNT(EMPLOYEE_ID)
22  FROM EMPLOYEES
23  GROUP BY COUNTRY_ID, DEPARTMENT_ID;
```

5.2.7 Aggregate Phase

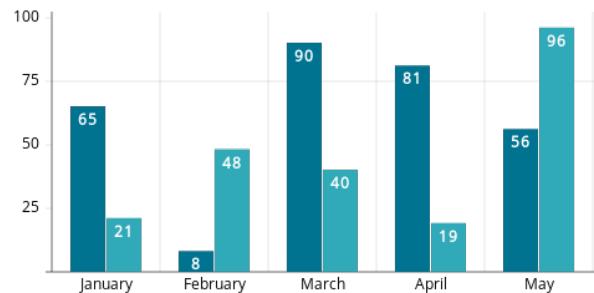


Aggregate Phase ▾

In der Aggregate Phase werden die **Datensätze** der virtuellen Gruppen auf einen einzelnen Datensatz **verdichtet**.

Die Datenbasis der Abfrage besteht nun aus den aggregierten Datensätzen.

Nach der **Map Phase** des Gruppierungsvorgangs, liegen die Datensätze der Abfrage verteilt auf **virtuelle Tabellen** vor.



► Erklärung: Aggregate Phase ▾

- In der Aggregate Phase werden die **Datensätze** der einzelnen virtuellen Tabelle auf einen einzelnen Datensatz **verdichtet**.
- Die Verdichtung der Datensätze erfordert dabei eine **Neustrukturierung** der Daten. Für die Neustrukturierung der Daten gelten bestimmte Vorgaben.
- Umstrukturierte Datensätze dürfen nur Werte enthalten, nach denen ursprünglich gruppiert worden ist.
- Zusätzlich kann einem **Zieldatensatz** das Ergebnis beliebiger **Aggregatfunktionen** zugeordnet werden.
- Klauseln einer select Abfrage, die von der **Datenbankengine** nach der group by Klausel ausgewertet werden, haben nur mehr Zugriff auf die neustrukturierten Daten.

5.2.8 Having Klausel

Mit der **having Klausel** können für die in der group by Klausel definierten Gruppen, **Filterkriterien** definiert werden.

► Analyse: Filter: having Klausel ▾

- Mit der where Klausel kann der Datenbestand einer Abfrage nach bestimmten Kriterien gefiltert werden.
Zeitlich erfolgt die **Auswertung** der where Klausel in SQL Abfragen vor der Auswertung der group by Klausel. Es fehlt damit die Möglichkeit die **virtuellen Gruppen** einer group by Klausel zu filtern.
- Mit der having Klausel stellt die SQL Spezifikation eine Möglichkeit zur Verfügung, die virtuellen Gruppen der group by Klausel zu filtern.
- Wir haben damit eine Möglichkeit an der Hand eine **Selektion** auf Gruppenebene durchzuführen.

```
1  -- -----
2  --   having Klausel
3  --
4  SELECT DEPARTMENT_ID, MIN(SALARY) MIN_SAL
5  FROM EMPLOYEES
6  JOIN DEPARTMENTS USING (DEPARTMENT_ID)
7  GROUP BY DEPARTMENT_ID, JOB_ID
8  HAVING SUM(SALARY) < 10000;
```



5.2.9 Null Aggregat



Null Aggregat ▾

Als **Null Aggregat** werden **virtuelle Gruppen** bezeichnet, die aus Datensätzen bestehen die keiner konkreten Gruppe zugeordnet werden können.



► Erklärung: Null Aggregat ▾

- Alle Datensätze einer SQL Abfrage, die für eines der **Gruppierungskriterien** keinen Wert besitzen, werden in einer eigenen virtuellen Gruppe zusammengefasst: dem **Nullaggregat**.
- Das **NULLS FIRST** bzw. **NULLS LAST** Schlüsselwort bestimmt ob das Null Aggregat dem Ergebnis vorangestellt oder als letzte virtuelle Gruppe im Ergebnis figuriert.



6. SQL - Komplexe Abfragen

05

Komplexe Abfragen

01. Unterabfragen	62
02. With Klausel	65
03. Paarweise Vergleich	??
04. Mengenoperationen	66
05. Quantoren	70

6.1. Unterabfragen



Unterabfragen ▾

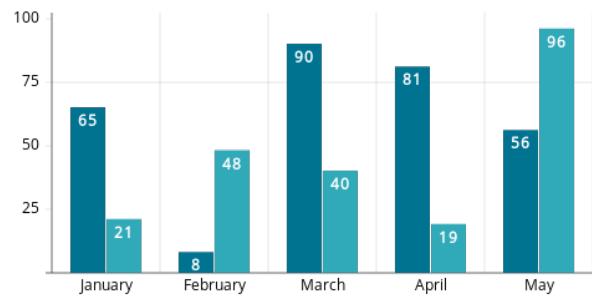
Unterabfragen sind **select Ausdrücke**, die in SQL Abfragen eingebettet werden.

6.1.1 Unterabfragen

Oft gibt es **Fragestellungen**, die auf einfache Weise in SQL nicht zu beantworten sind: z.B.: Wie heissen die Mitarbeiter mit dem geringsten Einkommen im Unternehmen?

► Analyse: Unterabfragen ▾

- Um diese Fragestellung beantworten zu können, müssten wir erst einmal wissen wie hoch das geringste Einkommen im Unternehmen ist, um anschließend zu ermitteln, welche Mitarbeiter ein entsprechendes Einkommen beziehen.
- **Das Problem:** Zur Beantwortung der vorhergehenden Frage müssten 2 Abfragen zur selben Zeit abgesetzt werden können.
- Wir brauchen damit einen neuen Ansatz: **Unterabfragen**.



► Erklärung: Unterabfragen ▾

- Unterabfragen sind **select Ausdrücke** die in SQL Abfragen eingebettet werden. Die SQL **Spezifikation** erlaubt dabei Unterabfragen in der select, from, where bzw. having Klausel.
- Beim **Auswerten einer SQL Abfrage**, berechnet die **Datenbankengine** zuerst das Ergebnis der Unterabfragen. Das Ergebnis der Unterabfrage steht der Abfrage dann in Form einer **virtuellen Tabelle** zur Verfügung.



Unterabfragen



6.1.2 Formen von Unterabfragen

Die SQL Spezifikation definiert mehrere Formen von Unterabfragen.

► Auflistung: Formen von Unterabfragen ▾



Innere View ▾

Unterabfragen in der **from Klausel** werden als **Innere Views** bezeichnet.

In der from Klausel einer Abfrage wird die **Datenbasis** einer Query definiert. Das **Ergebnis** der Inneren View bildet in diesem Fall die Datenbasis der Abfrage.



Skalare Abfragen ▾

Subqueries die nach der Auswertung durch die SQL Engine als Ergebnis lediglich einen **einzelnen Datensatz** enthalten, werden als Skalare Abfragen bezeichnet.

In der **select Klausel** dürfen nur Skalare Abfragen definiert werden.



Konditionale Abfragen ▾

Unterabfragen in der **where Klausel** bzw. **having Klausel** werden als Konditionale Abfragen bezeichnet.

Das Ergebnis einer Konditionale Abfrage wird für die Auswertung der having bzw. where Klausel herangezogen.

6.1.3 Subqueryform: Innere View

Unterabfragen in der **from Klausel** werden als **Innere Views** bezeichnet.

► Erklärung: Innere View ▾

- In der from Klausel einer Abfrage wird die **Datenbasis** einer Query definiert.
-
- Das Ergebnis der Inneren View wird dabei in einer **virtuellen Tabelle** gepeichert.

► Query: Innere View Beispiele ▾

```

1   -- -----
2   -- Subqueryform: Innere View
3   --
4   SELECT E.FIRST_NAME, E.LAST_NAME, E.SALARY
5   FROM (SELECT MAX(SALARY) MAX_SAL,
6           DEPARTMENT_ID
7           FROM EMPLOYEES
8           GROUP BY DEPARTMENT_ID) SUB
9   JOIN EMPLOYEES E
10  ON E.DEPARTMENT_ID = SUB.DEPARTMENT_ID
11  AND E.SALARY = SUB.MAX_SAL;
12
13  SELECT E.FIRST_NAME, E.LAST_NAME, E.SALARY
14      E.DEPARTMENT_ID,
15      E.JOB_ID
16  FROM EMPLOYEES E JOIN
17      (SELECT MIN(SALARY) M_SAL
18      FROM EMPLOYEES) SUB_QUERY
19  ON E.SALARY = SUB_QUERY.M_SAL
20  ORDER BY E.DEPARTMENT_ID;
```



Konzept	Beschreibung	Seite
Innere View	Unterabfragen in der from Klausel werden als Innere Views bezeichnet. Das Ergebnis einer Inneren View bildet in diesem Fall die Datenbasis einer Abfrage.	63
Skalare Abfrage	Subqueries die nach Ihrer Auswertung durch die Datenbankengine als Ergebnis lediglich einen einzelnen Datensatz enthalten, werden als Skalare Abfragen bezeichnet. Skalare Abfragen können in der select, from, where bzw. having Klausel definiert werden.	64
Konditionale Abfrage	Unterabfragen in der where Klausel bzw. having Klausel werden als Konditionale Abfragen bezeichnet. Das Ergebnis einer Konditionale Abfrage wird für die Auswertung der having bzw. where Klausel herangezogen.	64
With Klausel	Mit der with Klausel kann SQL Code einfach und lesbar strukturiert werden. Dazu wird in der with Klausel eine Liste von Queries definiert, die anschließend in der eigentlichen Abfrage referenziert werden können.	65
Union	Die union Klausel wird verwendet, um die Ergebnismengen zweier oder mehrerer select Anweisungen miteinander zu kombinieren.	66
Intersect	Die intersect Klausel wird verwendet, um die Durchschnittsmenge zweier oder mehrerer select Anweisungen zu bilden.	67
Except	Die except Klausel wird verwendet, um die Differenzmenge zweier oder mehrerer select Anweisungen zu bilden.	67
Quantoren	Zur Prüfung einer Menge von Datensätzen auf bestimmte Eigenschaften können die logischen Operatoren ALL, ANY bzw. EXISTS verwendet werden.	70
Hierarchische Abfragen	Hierarchische Abfragen ermöglichen Auswertungen zu Hierarchischen Strukturen im Datenbestand.	68

Abbildung 25. Konzepte komplexer Abfragen

6.1.4 Subqueryform: Skalare Abfragen

Subqueries die nach der Auswertung durch die SQL Datenbankengine im Ergebnis lediglich einen **einzelnen Datensatz** enthalten, werden als Skalare Abfragen bezeichnet.

Unterabfragen in der select Klausel müssen Skalare Abfragen sein.

► Query: Skalare Abfrage ▾

```

1  -- -----
2  -- Subqueryform: Innere View
3  --
4  SELECT E.FIRST_NAME, E.LAST_NAME,
5      (SELECT SUM(SALARY) FROM EMPLOYEES)
6      "company income"
7  FROM EMPLOYEES E JOIN DEPARTMENTS D
8  ON E.DEPARTMENT_ID = D.DEPARTMENT_ID
9  GROUP BY E.DEPARTMENT_ID,
10     D.DEPARTMENT_NAME
11 ORDER BY D.DEPARTMENT_NAME, E.LAST_NAME;
```

6.1.5 Subqueryform: Konditionale Abfrage

Unterabfragen in der **where** bzw. **having** Klausel werden als Konditionale Unterabfragen bezeichnet.

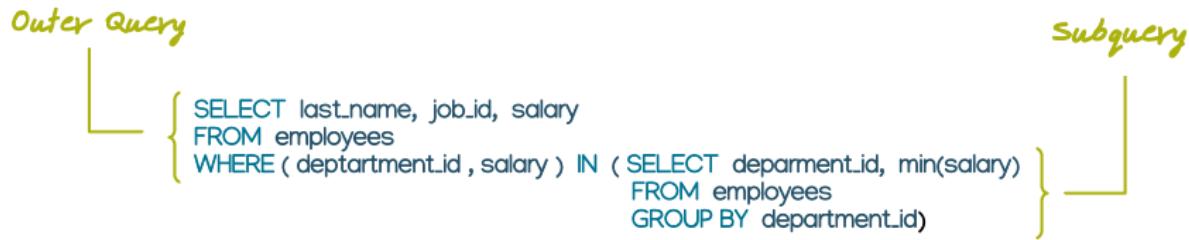
► Erklärung: Konditionale Abfrage ▾

- Das Ergebnis einer Konditionale Abfrage wird für die Auswertung der having bzw. where Klausel herangezogen.
- Verwenden Sie für Vergleiche mit Konditionalen Abfragen den **in** Operator.

► Query: Konditionale Unterabfragen ▾

```

1  -- -----
2  -- Subqueryform: Unterabfragen
3  --
4  SELECT E.FIRST_NAME, E.LAST_NAME
5  FROM EMPLOYEES E
6  WHERE E.SALARY IN
7      (SELECT MIN(SALARY) FROM EMPLOYEES);
```



6.2. With Klausel

Mit Hilfe der With Klausel können SQL Abfragen auf einfache Weise **logisch strukturiert** werden.

6.2.1 WITH Klausel - Strukturierung

Komplexe SQL Abfragen sind oft schwer lesbar bzw. unverständlich.

► Erklärung: Codestrukturierung ▾

- SQL Code wird im Gegensatz zur funktionalen Programmierung nicht über Unterprogrammen, sondern mit der Hilfe von **Unterabfragen** strukturiert.
- Die Strukturierung von SQL Code mit Unterabfragen führt jedoch zu einer **Verschachtelung** des Abfragecodes.
- Abfragen dieser Form müssen von innen nach außen ausgewertet.



WITH Klausel - Strukturierung ▾

Die with Klausel ermöglicht es SQL Code auf einfache Weise logisch zu strukturieren. Dazu wird in der with Klausel eine **Liste von Queries** definiert, die aus der eigentlichen Abfrage heraus **referenziert** werden können.

Die in der with Klausel definierten Abfragen werden als Innere Views referenziert.



6.2.2 Syntax: WITH Klausel

In SQL Abfragen wird die with Klausel syntaktisch vor der select Klausel definiert.

► Syntax: WITH Klausel ▾

```

1  -- -----
2  -- Syntax: with Klausel
3  --
4  WITH query_name1 AS (
5      SELECT ...
6  ), query_name2 AS (
7      SELECT ...
8  ), query_name3 AS (
9      SELECT ...
10 )
11 SELECT ...
12     FROM query_name1 q1
13     JOIN query_name2 q2 on ...
14     ...
15
16 --
17 -- Query: with Klausel
18 --
19 WITH REP_EMPLOYEES AS (
20     SELECT MAX(SALARY) MAX_SAL,
21             E.DEPARTMENT_ID
22     FROM EMPLOYEES E
23     GROUP BY E.DEPARTMENT_ID
24 )
25 SELECT E.DEPARTMENT_ID
26         E.FIRST_NAME
27     FROM EMPLOYEES E
28     JOIN REP_EMPLOYEES R ON
29         E.SALARY = R.MAX_SAL AND
30         E.DEPARTMENT_ID = R.DEPARTMENT_ID
31     JOIN DEPARTMENTS D ON
32         D.DEPARTMENT_ID = E.DEPARTMENT_ID
33     ORDER BY D.DEPARTMENT_NAME;

```

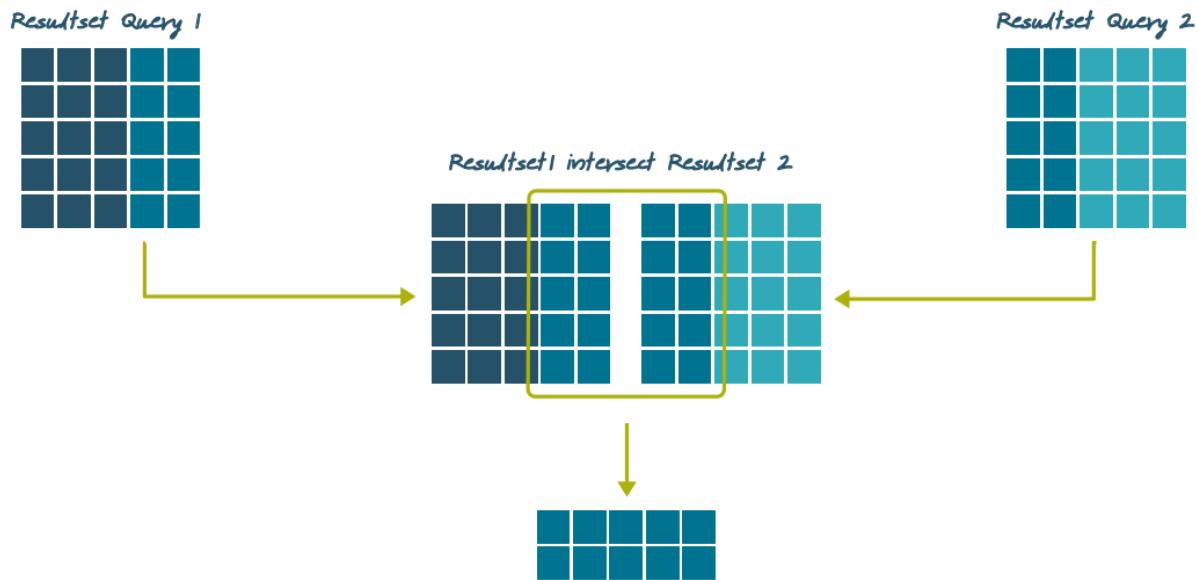


Abbildung 26. Mengentheoretische Operation: Intersect

6.3. Mengentheoretische Operationen ▾



Mengenlehre ▾

Die Mengenlehre ist ein grundlegendes Teilgebiet der Mathematik, das sich mit der **Untersuchung von Mengen** beschäftigt.

Die Mengenlehre ist dabei ein essentieller Bestandteil der SQL Sprachspezifikation.

6.3.1 Mengentheoretische Operationen

SQL als Programmiersprache definiert eine **mengenorientierte Sprachschnittstelle** zur Datendefinition bzw. Datenmanipulation.

► Erklärung: Mengentheoretische Operationen ▾

- Mengentheoretische Operationen sind ein fundamentaler Bestandteil der **SQL Sprachspezifikation**.
- Die Sprache selbst unterstützt dabei alle grundlegenden mengentheoretischen Operationen.
- Die Form einer mengentheoretischen Operation folgt dabei immer derselben Syntax.

6.3.2 Union Klausel - Vereinigungsmenge

Die union Klausel wird verwendet, um die **Ergebnismengen** zweier oder mehrerer select Anweisungen miteinander zu **kombinieren**.

► Erklärung: union, union all Klausel ▾

- Um die Ergebnisse 2er Queries miteinander kombinieren zu können, muss die **Struktur** der **Ergebnisdatensätze** beider Abfragen übereinstimmen.
- Verwenden Sie die union Klausel statt der **union all** Klausel um Kopien des gleichen Datensatzes aus der Ergebnismengen der Abfrage zu entfernen.

► Codebeispiel: union, union all Klausel ▾

```

1  -- -----
2  -- Union, Union all Klausel
3  --
4  SELECT LAST_NAME, FIRST_NAME
5  FROM EMPLOYEES
6  WHERE SALARY=(SELECT MIN(SALARY) FROM EMP)
7  --> UNION ALL <--
8  SELECT LAST_NAME, FIRST_NAME
9  FROM EMPLOYEES
10 WHERE SALARY=(SELECT MAX(SALARY) FROM EMP);

```

Konzept	Beschreibung	Mengenoperation
Schnittmenge	Die Schnittmenge zweier Mengen A und B ist die Menge aller Elemente, die sowohl zu A als auch zu B gehören.	INTERSECT
Vereinigungsmenge	Die Vereinigungsmenge zweier Mengen A und B ist die Menge aller Elemente, die zu A oder zu B oder zu beiden Mengen gehören.	UNION
Differenzmenge	Die Differenzmenge zweier Mengen A und B ist die Menge aller Elemente, die zu A, nicht aber zu B gehören.	EXCEPT

Abbildung 27. Mengentheoretische Operationen

6.3.3 Intersect Klausel - Durchschnittsmenge

Mit der intersect Klausel kann die **Durchschnittsmenge** der Ergebnisdatensätze zweier oder mehrerer select Anweisungen ermittelt werden.

► Syntax: intersect Klausel ▾

```

1  -- -----
2  -- Syntax: intersect Klausel
3  --
4  SELECT Ausdruck1, Ausdruck2, ... Ausdruck_n
5  FROM Tabellen
6  INTERSECT
7  SELECT Ausdruck1, Ausdruck2, ... Ausdruck_n
8  FROM Tabellen;
9
10 --
11 -- intersect Klausel
12 --
13 SELECT P.ID, P.TITLE, P.DESCRIPTION
14     P.PROJECT_STATE,
15     P.PROJECT_TYPE,
16  FROM L_PROJECTS P
17 WHERE ( SELECT SUM(AMOUNT)
18         FROM L_FUNDINGS
19         WHERE PROJECT_ID = P.ID
20         GROUP BY PROJECT_ID ) > 0
21 ORDER BY P.TITLE
22 --> INTERSECT <--
23 SELECT P.PROJECT_ID, P.TITLE, P.DESCRIPTION
24     P.PROJECT_STATE,
25     P.PROJECT_TYPE
26  FROM C_PROJECTS P JOIN C_FUNDING F
27  ON P.PROJECT_ID = F.PROJECT_ID
28 WHERE F.AMOUNT > 0
29 ORDER BY P.TITLE;
```

6.3.4 Except Klausel - Differenzmenge

Mit der except Klausel kann die **Differenzmenge** der Ergebnisdatensätze zweier oder mehrerer select Anweisungen ermittelt werden.

► Syntax: except Klausel ▾

```

1  -- -----
2  -- Syntax: except Klausel
3  --
4  SELECT Ausdruck1, Ausdruck2, ... Ausdruck_n
5  FROM Tabellen
6  EXCEPT
7  SELECT Ausdruck1, Ausdruck2, ... Ausdruck_n
8  FROM Tabellen;
9
10 --
11 -- except Klausel
12 --
13 SELECT P.ID, P.TITLE, P.DESCRIPTION
14     P.PROJECT_STATE,
15     P.PROJECT_TYPE,
16  FROM L_PROJECTS P
17 WHERE ( SELECT SUM(AMOUNT)
18         FROM L_FUNDINGS
19         WHERE PROJECT_ID = P.ID
20         GROUP BY PROJECT_ID ) > 0
21 ORDER BY P.TITLE
22 --> EXCEPT <--
23 SELECT P.PROJECT_ID, P.TITLE, P.DESCRIPTION
24     P.PROJECT_STATE,
25     P.PROJECT_TYPE
26  FROM C_PROJECTS P JOIN C_FUNDING F
27  ON P.PROJECT_ID = F.PROJECT_ID
28 WHERE F.AMOUNT > 0
29 ORDER BY P.TITLE;
```

Pseudospalte	Beschreibung	Seite
CONNECT_BY_ISLEAF	Die connect_by_isleaf Pseudospalte ist ein logischer Verweis auf die hierarchische Abhangigkeit eines Datensatzes. Ein Datensatz ohne Kindelemente hat dabei einen Wert von 1. alle anderen Datensatze haben eine Wert von 0.	69
CONNECT_BY_ISCYCLE	Besteht zwischen abhangigen Datensatzen einer hierarchischen Struktur eine wechselseitige Beziehung hat die connect_by_iscycle Spalte einen Wert von 1 anderfalls 0.	69
CONNECT_BY_ROOT	Die connect_by_root Pseudospalte ist ein logischer Verweis auf die Wurzel der hierarchischen Struktur.	69
LEVEL	Der Level eines Datensatzes beschreibt die Position des Datensatzes innerhalb einer hierarchischen Struktur. Datensatze an der Wurzel eines Baums haben dabei einen Level von 1.	69

Abbildung 28. Hierarchische Abfragen: Pseudospalten

6.4. Hierarchische Abfragen



Hierarchische Abfragen ▾

Bei Hierarchischen Abfragen handelt es sich um eine besondere Form von **SQL Abfragen**. Hierarchische Abfragen werden zur Auswertung **hierarchischer Strukturen** verwendet.

6.4.1 Hierarchische Strukturen

Hierarchische Strukturen gehoren zu den primaren **Datenstrukturen** der Datenverarbeitung.

► Erklarung: Hierarchische Strukturen ▾

- Hierarchische Strukturen werden in der Datenverarbeitung zur Beschreibung einer Vielzahl von **Anwendungsfallen** verwendet: Organigramme, Fhrungsstrukturen, Dateisysteme usw.
- Hierarchische Strukturen werden dazu als logische **Baumstrukturen** abgebildet.
- Relationale Systeme konnen Baumstrukturen jedoch nicht direkt darstellen. Zur Definition einer Hierarchie zwischen 2 Datensatzen, werden die Datensatze miteinander in **Relation** gesetzt. Der Ausgangs- und Zielpunkt der Relation ist in der Regel dieselbe Entitat.

6.4.2 CONNECT BY Klausel - Rekursion

Fur Abfragen auf hierarchischen Strukturen wird in SQL Abfragen die **connect by Klausel** verwendet.

► Erklarung: Connect by Klausel ▾

- In der connect by Klausel werden die Abhangigkeiten der **Eltern- und Kindelementen** einer Baumstruktur beschrieben. Das prior Schlsselwort wird dabei dem Kindelement zugestellt.
- Mit dem starts with Pradikat der connect by Klausel wird der Startpunkt einer hierarchischen Struktur definiert. Eine **Baumstruktur** kann dabei mehrere **Wurzelemente** haben. Wird in der Abfrage kein starts with Pradikat definiert, gelten alle Eintrage der Tabelle als Ausgangspunkt der Rekursion.
- Die SQL Engine wertet die connect by Klausel dabei nach der where und vor der select Klausel aus.

► Codebeispiel: Connect by Klausel ▾

```

1  -- -----
2  -- Fallbeispiel: Connect By Klausel
3  --
4  SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME
5      MANAGER_ID, LEVEL
6  FROM EMPLOYEES
7  STARTS WITH MANAGER_ID IS NULL
8  CONNECTED BY
9  PRIOR EMPLOYEE_ID = MANAGER_ID;

```

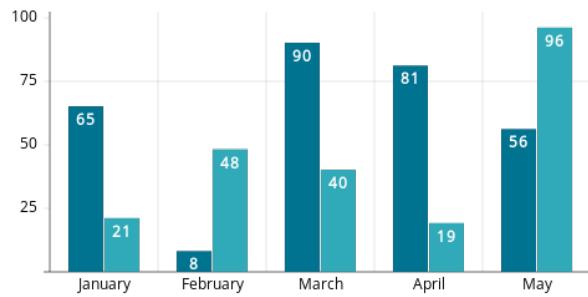
Funktion	Beschreibung	Seite
SYS_CONNECT_BY_PATH	Die SYS_CONNECT_BY_PATH Funktion generiert aus der Liste aller Vorfahren des Datensatzes einen Zeichenkettendarstellung.	69
RPAD	Mit der RPAD Funktion kann eine Zeichenkette rechtsbündig mit einem vorgegebenem Zeichen aufgefüllt werden.	69

Abbildung 29. Hierarchische Abfragen: Funktionen

6.4.3 Hierarchische Abfragen: Pseudospalten ▾

Zusammen mit der connect by Klausel definiert die SQL Spezifikation eine Reihe von **Pseudospalten** zur Formulierung von hierarchischen Abfragen.

Die Pseudospalten der connect by Klausel halten zusätzliche Informationen zur **Beschreibung** hierarchischer Strukturen.



► Codebeispiel: Pseudospalten ▾

```

1 -- -----
2 -- Tabellendefinition: trees
3 --
4 CREATE TABLE TREES (
5   ID NUMBER, PARENT_ID NUMBER,
6   CONSTRAINT TREES_PK PRIMARY KEY (ID),
7   CONSTRAINT TREES_TREES_FK
8     FOREIGN KEY (PARENT_ID)
9     REFERENCES TAB1(ID)
10 );
11
12 SELECT ID, CONNECT_BY_ISLEAF AS LEAF,
13   LEVEL, PARENT_ID
14 FROM TREES
15 START WITH PARENT_ID IS NULL
16 CONNECT BY PARENT_ID = PRIOR ID;
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

```

6.4.4 Hierarchische Abfragen: Funktionen ▾

Zusammen mit der connect by Klausel definiert die SQL Spezifikation eine Reihe von **Funktionen** zur Formulierung von hierarchischen Abfragen.

► Codebeispiel: Funktionen ▾

```

1 -- -----
2 -- Syntax: sys_connect_by_path
3 --
4 -- Die Funktion generiert aus der Liste
5 -- aller Vorfahren des Datensatzes eine
6 -- Zeichenkettendarstellung.
7 SYS_CONNECT_BY_PATH(
8   LABEL IN NAME%TYPE,
9   SEPARATOR IN VARCHAR2
10 );
11
12 -- -----
13 -- Syntax: RPAD
14 --
15 -- Mit der RPAD Funktion kann eine Zeichen-
16 -- kette rechtsbündig mit einem vorgegebenem
17 -- Zeichen aufgefüllt werden.
18 RPAD(
19   VALUE IN VARCHAR2,
20   SIZE IN NUMBER,
21   LITERAL IN CHAR
22 );
23
24
25
26
27
28
29
30
31

```

6.5. Quantoren



Quantoren ▾

Quantoren sind **logische Operatoren** mit denen eine Menge von Werten auf eine bestimmte Eigenschaft geprüft werden kann.

6.5.1 ALL Operator

Beim all Operator handelt es sich um einen **logischen Operator**.

Der all Operator evaluiert zu true wenn alle Elemente einer Menge eine bestimmte Bedingung erfüllen.

► Syntax: All Operator ▾

```

1  -- -----
2  -- Syntax: ALL Operator
3  --
4  SELECT <expression>
5  FORM <expression>
6  WHERE column_name <operator> ALL
7    (SELECT <expression> ... )
8
9  -- -----
10 -- ALL Operator
11 --
12 -- Finden Sie alle Angestellten die das hoe-
13 -- chste Gehalt im Unternehmen beziehen.
14 SELECT E.FIRST_NAME, E.LAST_NAME
15   E.DEPARTMENT_NAME,
16   E.DEPARTMENT_ID,
17   E.SALARY
18 FROM EMPLOYEES E
19 JOIN DEPARTMENT D
20 ON E.DEPARTMENT_ID = D.DEPARTMENT_ID
21 WHERE NOT EXISTS (
22   SELECT E2.LAST_NAME
23   FROM EMPLOYEES E2
24   WHERE E2.SALARY > E.SALARY
25 )
26 ORDER BY LAST_NAME;
27
28 -- Finden Sie alle Projekte die keine
29 -- Foerderung haben.
30
31 SELECT P.TITLE, P.PROJECT_TYPE
32   P.IS_FWF_SPONSORED,
33   P.IS_EU_SPONSORED,
34   P.PROJECT_STATE
35 FROM PROJECTS P
36 WHERE NOT EXISTS (
37   SELECT F.AMOUNT
38   FROM FUNDINGS F
39   WHERE F.PROJECT_ID = P.PROJECT_ID
40 )
41 ORDER BY P.TITLE;
```

6.5.2 EXISTS Operator

Beim exists Operator handelt es sich um einen **logischen Operator**.

Der exists Operator evaluiert zu true wenn zumindestens ein Element einer Menge eine bestimmte Bedingung erfüllt.

► Syntax: Exists Operator ▾

7. SQL - Data Definition Language

06

DDL - Data Definition Language

01. Datenbankartefakte	72
02. Tabelle	73
03. Constraint	76
04. View	78
05. Index	79
06. Sequenz	82

7.1. Datenbankartefakte



Data Definition Language ▾

Mit **DDL Befehle** kann die **Struktur** einer relationalen Datenbank definiert bzw. geändert werden.

7.1.1 Datenbankartefakte

Datenbankartefakte beschreiben die Struktur einer Datenbank.



► Auflistung: Datenbankartefakte ▾

- **Schema:** Ein Schema ist ein logischer **Namensraum** für die Tabellen eines Geschäftsfalls. Im Sprachgebrauch wird ein Schema auch als Geschäftsfall bezeichnet. Ein Datenbankserver kann eine beliebige Zahl von Schemen verwalten.
- **Tabelle:** Ein Datenbankschema ist ein logischer Namensraum für die Daten eines Geschäftsfalls. Tabellen repräsentieren dabei die einzelnen **Aspekte** der Geschäftsdaten. Aus technischer Sicht werden in Tabellen gleichartige **Datensätze** gesammelt.
- **Constraints:** Constraints werden verwendet um die **Konsistenz** der Daten in einer Datenbank sicherzustellen. SQL unterstützt mehrere Formen von Constraints. Damit können die primären **Konsistenzanforderungen** sichergestellt werden.
- **View:** Eine View ermöglicht es Datenbankabfragen wie **Datenbankobjekte** zu behandeln. SQL Queries können so für den späteren Gebrauch gespeichert werden.
- **Sequenz:** Mit der Hilfe einer Datenbanksequenz können Sequenzen aufeinanderfolgender Werte generiert werden. Sequenzen werden in der Regel zur Generierung von Schlüsselwerten definiert.
- **Trigger:** Trigger sind Datenbankroutinen deren Ausführung durch bestimmte **Ereignisse** angestoßen wird.
- **Index:** Ein Datenbankindex ist eine **Datenstruktur** zur Beschleunigung der Suche und Sortierung von Datensätzen.



Befehl	Beschreibung	Seite
CREATE TABLE	Der CREATE TABLE Befehl wird zum Anlegen einer Tabelle in der Datenbank verwendet. Der Befehl besitzt dabei 2 unterschiedliche Ausprägungen um die Struktur von Tabellen zu definieren.	74
ALTER TABLE	Der ALTER TABLE Befehl wird zum Bearbeiten der Struktur einer Tabelle in der Datenbank verwendet. Der Befehl besitzt dabei unterschiedliche Ausprägungen und Optionen zum Löschen bzw. Hinzufügen von Feldern bzw. Constraints.	75
DROP TABLE	Der DROP TABLE Befehl wird zum Löschen einer Tabelle in der Datenbank verwendet.	74
TRUNCATE TABLE	Der TRUNCATE TABLE Befehl wird zum Löschen der Datensätze einer Tabelle verwendet.	75
CREATE VIEW	Der CREATE VIEW Befehl zum Anlegen einer View in der Datenbank verwendet.	78
DROP VIEW	Der DROP VIEW Befehl zum Löschen einer View in der Datenbank verwendet.	79
CREATE SEQUENCE	Der CREATE SEQUENCE Befehl zum Anlegen einer Sequenz in der Datenbank verwendet. Der Befehl besitzt unterschiedliche Optionen um die Sequenz an den geforderten Geschäftsfall anpassen zu können.	??
DROP SEQUENCE	Der DROP SEQUENCE Befehl zum Löschen einer Sequenz aus der Datenbank verwendet.	??

Abbildung 30. DDL Befehle

7.2. Datenbankartefakt: Tabelle

Eine Datenbanktabelle repräsentiert einen einzelnen Aspekt eines spezifischen Geschäftsfalls.

7.2.1 Tabelle

 Datenbanktabelle ▾
Datenbanktabellen bestehen aus Zeilen und Spalten. Daten werden in Form von Datensätzen in einer Tabelle gespeichert.

- **Zeile:** Jede Zeile einer Tabelle speichert einen einzelnen Datensatz.
- **Spalte:** Die Spalten einer Tabelle beschreiben die Attribute der Datensätze.

► Erklärung: Datenbanktabellen ▾

- In einer Datenbanktabelle werden alle gleichartigen **Datensätze** eines Geschäftsfalls gesammelt. Der Einsatz von Tabellen hilft dabei bei der logischen und physischen **Strukturierung** der Daten.

7.2.2 Tabellendefinition



Tabellendefinition ▾

Zur Beschreibung einer Tabelle werden die Eigenschaften der einzelnen Spalten der Tabelle definiert. **Spaltendefinitionen** folgen dabei einem strikt vorgegebenem Muster:

spaltenbezeichner datentyp [attribute+]

► Erklärung: Spaltendefinition ▾

- **Spaltenbezeichner:** Für eine Tabellenspalte kann ein beliebiger **Spaltenbezeichner** gewählt werden. Es ist jedoch darauf zu achten daß der Bezeichner nicht mehr als 32 Zeichen lang sein kann, sowie keine Sonderzeichen enthalten darf.
- **Datentyp:** Tabellenspalten haben immer einen spezifischen Datentyp. Lediglich **Werte** die zum angegebenen **Datentyp** kompatibel sind, können in eine Spalte eingetragen werden.
- **Attribute:** Optional können für eine Spalte eine Reihe von Attributen definiert werden.

7.2.3 Schlüsseldefinition

Eine der Spalten einer Tabelle muß als **Schlüsselspalte** designiert werden.



Primärschlüssel ▾

Anhand eines Primärschlüssels können die in der Tabelle gespeicherten Datensätze von einander **unterschieden** werden.

► Erklärung: Schlüsselspalten ▾

- Der **Schlüssel** einer Tabelle besteht dabei aus einer oder mehreren Spalten der Tabelle. Ein Schlüssel darf jedoch maximal nur soviele Spalten enthalten, wie notwendig sind, um die Datensätze der Tabelle zu differenzieren.
- Bestimmte Spalten einer Tabelle können auch als **Fremdschlüsselspalten** ausgewiesen werden. Fremdschlüssel sind Verweise auf die Primärschlüssel anderer Tabellen. Über Fremdschlüssel werden die Daten unterschiedlicher Tabellen in **Relation** gesetzt.



7.2.4 DDL Befehl: create table

Der create table Befehl wird zum Anlegen von Tabellen in einer Datenbank verwendet.

► Syntax: create table ▾

```

1  -- -----
2  -- Syntax: CREATE TABLE
3  -- -----
4  CREATE TABLE table_name (
5      column1 datatype [NOT NULL|NULL|UNIQUE],
6      ...
7      columnN datatype,
8      PRIMARY KEY (column_name+),
9      [CONSTRAINT fk_name FOREIGN KEY
10         (column_name+)
11         REFERENCES table_name (column_name+)
12         [ON DELETE CASCADE]]
13     );
14
15     CREATE TABLE table_name AS
16         SELECT column1, column2, ...
17         FROM existing_table_name;

```

► DDL: employees, departments ▾

```

1  -- -----
2  -- Fallbeispiel: hr schema
3  -- -----
4  CREATE TABLE EMPLOYEES (
5      EMPLOYEE_ID    NUMBER(19,0) NOT NULL,
6      SALARY         NUMBER(10,0) NOT NULL,
7      JOB_ID         NUMBER(19,0) NOT NULL,
8      FIRST_NAME     VARCHAR(30) NOT NULL,
9      MIDDLE_NAME    VARCHAR(30) NOT NULL,
10     LAST_NAME      VARCHAR(30) NOT NULL,
11     DEPARTMENT_ID  NUMBER(19,0) NOT NULL,
12
13     PRIMARY KEY (EMPLOYEE_ID),
14     CONSTRAINT FK_DEPARTMENTS_DEP_ID
15         FOREIGN KEY (DEPARTMENT_ID)
16             REFERENCES DEPARTMENTS (DEPARTMENT_ID)
17 );
18
19 CREATE TABLE DEPARTMENTS (
20     DEPARTMENT_ID NUMBER(19,0) NOT NULL UNIQUE,
21     LOCATION_ID   NUMBER(19,0) NOT NULL,
22     DEP_NAME       VARCHAR(30) NOT NULL,
23
24     PRIMARY KEY (DEPARTMENT_ID),
25     CONSTRAINT FK_LOCATION_LOCATION_ID
26         FOREIGN KEY (LOCATION_ID)
27             REFERENCES LOCATIONS (LOCATION_ID)
28             ON DELETE CASCADE
29 );

```

7.2.6 DDL Befehl: drop table

Der drop table Befehl wird zum Löschen von Tabellen aus der Datenbank verwendet.

► DDL: Tabellen verwalten ▾

```

1  -- -----
2  -- Befehl: DROP TABLE
3  -- -----
4  -- Befehl: Table Artefakte loeschen
5  DROP TABLE TABLE_NAME;
6
7  DROP TABLE STUDENTS;

```

7.2.7 DDL Befehl: truncate table

Der truncate table Befehl wird zum Löschen aller Datensätze einer Tabelle verwendet.

► DDL: Datensätze löschen ▾

```

1  -- -----
2  --   Befehl: TRUNCATE TABLE
3  -- -----
4  --   Befehl: Table Artefakte loeschen
5  TRUNCATE TABLE TABLE_NAME;
6
7  TRUNCATE TABLE STUDENTS;
```



7.2.8 DDL Befehl: alter table

Der alter table Befehl wird verwendet um die Struktur der Tabellen einer Datenbank zu ändern.

► Syntax: alter table ▾

```

1  -- -----
2  --   Syntax: ALTER TABLE
3  -- -----
4  --   Der ALTER TABLE Befehl besitzt unter-
5  --   schiedliche Ausprägungen und Optionen
6  --   für das Löschen bzw. Hinzufügen von
7  --   Feldern bzw. Constraints.
8
9  ALTER TABLE table_name {
10    RENAME TO new_table_name      |
11    MODIFY (column action)       |
12    ADD (column_name datatype attribute) |
13    DROP (column_name)          |
14    ADD CONSTRAINT fk_name      |
15      FOREIGN KEY (column_name+) |
16      REFERENCES table (column_name+) |
17    RENAME CONSTRAINT old TO new  |
18    DROP CONSTRAINT constraint_name |
19    DROP PRIMARY KEY
20  }
21
22  -- -----
23  --   Fallbeispiel: ALTER TABLE
24  -- -----
25  ALTER TABLE PROJECTS RENAME TO C_PROJECTS;
26
27  ALTER TABLE C_PROJECTS ADD (
28    IS_FFG_SPONSORED NUMBER(1)
29  );
```



7.2.9 Fallbeispiel: alter table

Der alter table Befehl wird verwendet um die Struktur der Tabellen der Datenbank zu verwalten.

► DDL: alter table Beispiele ▾

```

1  -- -----
2  --   Fallbeispiel: ALTER TABLE
3  -- -----
4  --   Befehl: Constraints hinzufügen/lösen
5  ALTER TABLE C_PROJECT_PARTNERS
6  ADD CONSTRAINT FK_PROJECT_PARTNERS
7    FOREIGN KEY (PROJECT_ID)
8    REFERENCES C_PROJECTS (PROJECT_ID);
9
10 ALTER TABLE C_PROJECTS ADD CONSTRAINT
11 UQ_PROJECTS_CODE UNIQUE (PROJECT_CODE);
12
13 ALTER TABLE C_PROJECTS DROP CONSTRAINT
14 UQ_PROJECTS_CODE;
15
16 ALTER TABLE C_PROJECTS ADD CONSTRAINT
17 CHECK_LEGAL_TYPE CHECK
18 (LEGAL_FOUNDATION IN ('P_26', 'P_27'));
19
20 -- Befehl Constraints ein-/ausschalten
21 ALTER TABLE C_PROJECTS DISABLE CONSTRAINT
22 UQ_PROJECTS_CODE;
23
24 ALTER TABLE C_PROJECTS ENABLE CONSTRAINT
25 UQ_PROJECTS_CODE;
26
27 -- Spalteneigenschaften verändern
28 ALTER TABLE C_PROJECTS
29 MODIFY DESCRIPTION VARCHAR2(4000);
30
31 ALTER TABLE C_PROJECTS MODIFY (
32   IS_FFG_SPONSORED NUMBER(1) NOT NULL,
33   IS_FWF_SPONSORED NUMBER(1) NOT NULL,
34   IS_EU_SPONSORED NUMBER(1) NOT NULL
35 );
36
37 -- Spalten hinzufügen
38 ALTER TABLE C_PROJECTS ADD DESCRIPTION
39 VARCHAR(255);
40
41 ALTER TABLE C_PROJECTS ADD (
42   IS_FFG_SPONSORED NUMBER(1),
43   IS_FWF_SPONSORED NUMBER(1),
44   IS_EU_SPONSORED NUMBER(1)
45 );
```



7.3. Datenbankartefakt: Constraint ▾

Datenkonsistenz bzw. Datenintegrität beschreibt die **Korrektheit**, der in einer Datenbank gespeicherten Daten.

7.3.1 Datenkonsistenz und Integrität

Der **Datenbestand** einer Datenbank ist konsistent, wenn die folgenden **Integritätsbedingungen** erfüllt werden können.

► Auflistung: Integritätsbedingungen ▾



Bereichsintegrität ▾

Die möglichen Werte der Attribute eines Datensatzes müssen in einem bestimmten **Wertebereich** enthalten sein.

Durch die Verwendung von **Datentypen** kann für die Spalten von Datenbanktabellen Bereichsintegrität erreicht werden.



Logische Konsistenz ▾

Die logische Konsistenz der Daten wird durch die Vorgaben der Geschäftsprozesse bestimmt.

Durch die Definition von **check** bzw. **unique Constraints** kann logische Konsistenz in einer Datenbank sichergestellt werden.



Referentielle Konsistenz ▾

Beziehungen dürfen für **Datensätzen** nur zwischen logisch assoziierten Objekten definiert werden. Zur Sicherstellung referentieller Konsistenz werden für Tabellen Primär- und Fremdschlüssel definiert.



Entitätsintegrität ▾

Jeder Datensatz des Datenbestands muß eindeutig **identifizierbar** sein. Entitätsintegrität wird durch die Definition von Primärschlüsseln sichergestellt.

7.3.2 Constraints

Constraints definieren **Bedingungen**, die beim Einfügen, Ändern bzw. Löschen von Datensätzen erfüllt sein müssen.

Durch die Verwendung von Constraints kann die **Konsistenz** der Daten einer Datenbank sichergestellt werden.

► Erklärung: Constraints ▾

- Constraints werden der Tabelle zugeschrieben, für die sie mittels des create table bzw. alter table Befehls definiert wurden.
- Constraints sind dabei keine eigenständigen Datenbankobjekte. Wird eine Tabelle gelöscht, werden zusammen mit der Tabelle alle zugehörigen Constraints entfernt.

► Syntax: Constraints anlegen ▾

```

1  -- -----
2  -- CREATE TABLE Befehl
3  -- -----
4  CREATE TABLE table(
5      column datatype [column_constraint],
6      column datatype [column_constraint],
7      ...
8      [table_constraint]
9  );
10 
11 CREATE TABLE EMPLOYEES(
12     EMPLOYEE_ID NUMBER(19,0) NOT NULL
13     FIRST_NAME VARCHAR(30) UNIQUE,
14     DEP_ID      NUMBER(19,0),
15     ...
16     PRIMARY KEY EMPLOYEE_ID,
17     CONSTRAINT FK_DEPARTMENT_DEP_ID
18         FOREIGN KEY (DEP_ID)
19             REFERENCES DEPARTMENTS (DEP_ID)
20 );
21 
22 -- -----
23 -- ALTER TABLE Befehl
24 -- -----
25 ALTER TABLE table_name
26 ADD CONSTRAINT constraint_name <type>
27 
28 ALTER TABLE PROJECTS
29 ADD CONSTRAINT P_DUR CHECK (DURATION >= 1);

```

Constraint	Beschreibung	Integritätsbedingung
not null	Der not null Constraint fordert, dass für eine bestimmte Spalte zwingend ein Wert eingetragen werden muss.	Logische Konsistenz
unique	Der unique Constraint fordert, dass alle für eine Spalte eingetragenen Werte eindeutig sind.	Logische Konsistenz
check	Der check Constraint ermöglicht die Formulierung spezifische Einschränkungen für die Werte einer Spalte.	Logische Konsistenz
key	Durch die Definition von Schlüsseln kann für die Daten einer Datenbank Entitätsintegrität und Referentielle Konsistenz sichergestellt werden.	Referentielle Konsistenz

Abbildung 31. Constrainttypen

7.3.3 Verwalten von Constraints

Zur Verwaltung von Constraints definiert die SQL Spezifikation den alter table Befehl.

► DDL: Verwalten von Constraints ▾

```

1  -- -----
2  -- Verwalten von Constraints
3  --
4  -- Constraint zu einer Tabelle hinzufuegen
5  ALTER TABLE PROJECTS
6  ADD CONSTRAINT UQ_PRO_CODE
7      UNIQUE (PROJECT_CODE);
8
9  -- Constraint deaktivieren
10 ALTER TABLE PROJECTS DISABLE UQ_PRO_CODE;
11
12 -- Constraint aktivieren
13 ALTER TABLE PROJECTS ENABLE UQ_PRO_CODE;
14
15 -- Constraint umbenennen
16 ALTER TABLE PROJECTS RENAME CONSTRAINT
17 UP_PRO_CODE TO UP_PROJECTS_CODE;
18
19 -- Constraint loeschen
20 ALTER TABLE PROJECTS DROP CONSTRAINT
21 UP_PROJECTS_CODE;
```

7.3.4 Constraintmonitoring

Das Constraintartefakt kann zur Laufzeit des Datenbankservers unterschiedliche Zustände haben.

Die user_cons_columns bzw. user_constraints Tabellen werden verwendet um den Zustand von Constraints zu verwalten.

► Query: Constraintmonitoring ▾

```

1  -- -----
2  -- Constraintmonitoring
3  --
4  -- Die user_cons_columns Tabelle ermoeglicht
5  -- eine Einsicht auf die Zugriffsrechte
6  -- fuer Constraints
7  SELECT CONSTRAINT_NAME,
8      TABLE_NAME,
9      COLUMN_NAME,
10     OWNER
11 FROM USER_CONS_COLUMNS
12 WHERE TABLE_NAME = 'PROJECTS';
13
14 -- Die user_constraints Tabelle speichert
15 -- fuer jeden Constraint seine Zugehoerigkeit,
16 -- Namen und Status
17 SELECT CONSTRAINT_NAME,
18     CONSTRAINT_TYPE,
19     LAST_CHANGE,
20     STATUS
21 FROM USER_CONSTRAINTS
22 WHERE TABLE_NAME = 'PROJECTS';
```



7.4. Datenbankartefakt: View ▾

Mit einer View kann eine **SQL Abfrage** als **Objekt** in einer Datenbank gespeichert werden.

7.4.1 View



View ▾

Eine **View** ist eine **SQL Abfrage**, die in der Datenbank als Objekt gespeichert wird.

Die Abfrage selbst wird dabei in Form einer **Zeichenkette** hinterlegt. Erst beim Aufruf der View wird die mit der View assoziierte Abfrage ausgeführt.

Views spielen eine zentrale Rolle in der **Entwicklung** relationaler Datenbanken.

► Auflistung: Einsatz von Views ▾

- **Komplexitätsreduktion:** Mit Views kann der **Zugriff** auf den **Datenbestand** einer Datenbank wesentlich vereinfacht werden.

Normalisierte Datenbankschema abstrahieren ein komplexes Netzwerk von Tabellen und Relationen. SQL Abfragen in solchen Strukturen sind komplex und umfangreich. Views werden verwendet um die Komplexität von Abfragen für den User zu abstrahieren.

- **Datenschutz:** Views können helfen die **Datenstruktur** einer Datenbank vor unbefugten Usern zu maskieren.

In der Regel ist es nicht erwünscht, die Struktur einer Datenbank nach außen hin zu publizieren. Views helfen die Details eines Schemas vor Usern zu verstecken. Durch die Vergabe von Rollen und Rechten kann der Zugriff auf die einzelnen Datenbankobjekte einfach gesteuert werden.

- **Kompilierungssicherheit:** Beim Speichern der View, wird die mit der View assoziierte Abfrage kompiliert und auf **syntaktische Richtigkeit** geprüft. Fehlerhafte SQL Abfragen müssen vor dem Speichern korrigiert werden.

- **Datenbankschnittstelle:** Durch die Verwendung von Views kann auf einfache Weise eine einheitliche Datenbankschnittstelle entworfen werden.

7.4.2 DDL Befehl: create view

Zum Anlegen von Views definiert die SQL Spezifikation den **create view** Befehl.

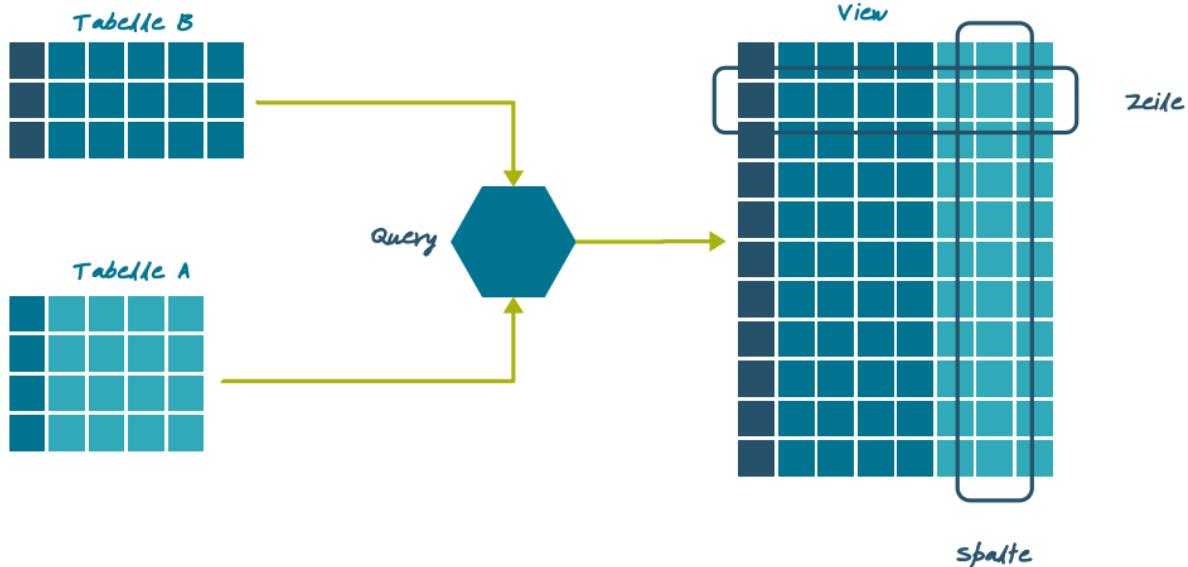
► Erklärung: Restriktionen einer View ▾

- Eine mit einer View assoziierte SQL Abfrage unterliegt in ihrer Form bestimmten Einschränkungen.
- Die **order by** Klausel einer View wird **ignoriert**, wenn in der SQL Abfrage selbst eine Sortierung definiert wird.

► Syntax: create view ▾

```

1  -- -----
2  -- Syntax: create view
3  --
4  CREATE OR REPLACE VIEW view_name
5  [(spalten_namen)+]
6  AS <select Ausdruck>;
7
8  -- -----
9  -- DDL: View anlegen
10 --
11 CREATE OR REPLACE VIEW employees_dep
12 (FIRST_NAME, LAST_NAME, DEPARTMENT_NAME)
13 AS
14 SELECT e.FIRST_NAME,
15       e.LAST_NAME,
16       D.DEPARTMENT_NAME
17 FROM EMPLOYEES E JOIN DEPARTMENTS D
18 USING(DEPARMENT_ID);
19
20
21 CREATE OR REPLACE VIEW MVP_EMPLOYEES
22 AS
23 SELECT E.FIRST_NAME,
24       E.LAST_NAME,
25       E.DEPARTMENT_ID
26 FROM EMPLOYEES E
27 JOIN
28   (SELECT MAX(SALARY) MAX_SALARY,
29    DEPARTMENT_ID
30    FROM EMPLOYEES
31    GROUP BY DEPARTMENT_ID) SUB
32 ON E.SALARY = SUB.MAX_SALARY AND
33 E.DEPARTMENT_ID = SUB.DEPARTMENT_ID;
34
35 -- Query: View ausfuehren
36 SELECT FIRST_NAME, LAST_NAME
37 FROM EMPLOYEES_DEP;
```



7.4.3 Variable Bedingungen

Eine **view** selbst kann keine **variablen Parameter** für die **where** Klausel enthalten.

► DDL: variable Bedingungen ▾

```

1  -- -----
2  --   DDL: variable Bedinugen
3  -- -----
4  CREATE OR REPLACE VIEW EMPLOYEES_DEPARTMENT
5  (FIRST_NAME, LAST_NAME, DEPARTMENT_ID)
6  AS
7  SELECT E.FIRST_NAME, E.LAST_NAME, E.DEP_ID
8  FROM EMPLOYEES E JOIN DEPARTMENTS D
9  USING(DEPARMENT_ID);

10
11 --   Query: View ausfuehren
12 SELECT LAST_NAME, FIRST_NAME
13 FROM EMPLOYEES_DEP
14 WHERE DEPARTMENT_ID = 'IT';

```

7.4.4 DDL Befehl: drop view

► Syntax: drop view ▾

```

1  -- -----
2  --   DDL: View lschen
3  -- -----
4  DROP VIEW EMPLOYEES_DEP;

```

7.5. Datenbankartefakt: Index



Index ▾

Der **Datenbankindex** einer Tabelle, ist eine von der eigentlichen Tabelle getrennte Struktur, zur Beschleunigung der **Suche** bzw. **Sortierung** von Daten.

7.5.1 Index

Ein Index dient dazu, die Werte einer Spalte zu erschließen.

► Erklärung: Datenbankindex ▾

- Ein Index ordnet jedem Eintrag einer indizierten Spalte seine Position im Datenbestand der Tabelle zu.
- Auswertungen auf indizierten Spalten haben im Vergleich zu gewöhnlichen SQL Abfragen eine signifikant verbesserte **Laufzeit**.
- Der Index einer Spalte wächst jedoch mit der Zahl der Einträge auf die er sich bezieht.
- Die Definition eines Indexes ist deshalb nur für jene Spalten einer Tabelle sinnvoll, die Teil der where, group by, having bzw. order by Klausel sind.

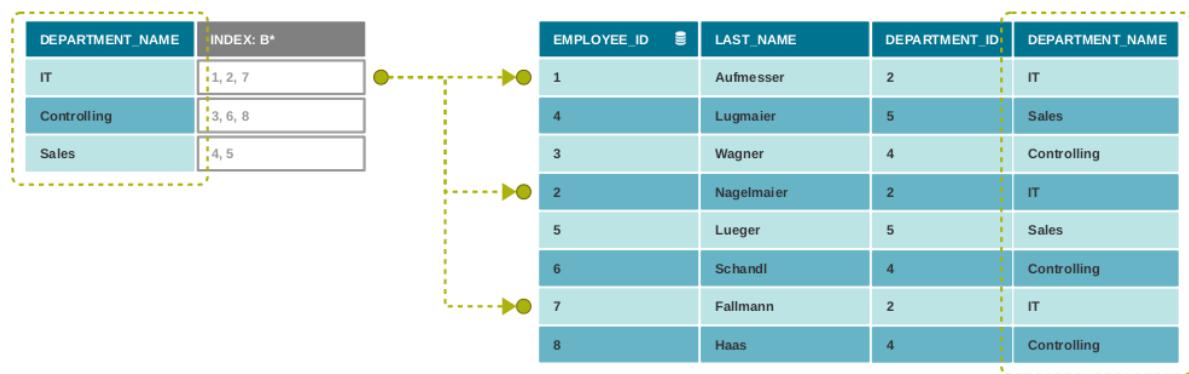


Abbildung 32. Indexstruktur

7.5.2 Indextyp: B* Index

Der B* Index ist der primäre Indextyp in relationalen Datenbanken.

► Erklärung: B* Index ▾

- Die einem B* Index zugrundeliegende **Datenstruktur** ist ein B Baum. B Bäume gehören zu den Datenstrukturen der **balancierte Suchbäume**. Verglichen mit anderen Datenstrukturen ist das Suchen, Einfügen bzw. Löschen von Werten in einem B Baum hoch optimiert.
- Ein B* Index wird dabei in erster Linie für Spalten zur Verwaltung numerischer bzw. alphanumerischer Werte eingesetzt.
- Die Datenbankengine legt zur Beschleunigung von Abfragen für Schlüsselpalten automatisch einen B* Index an.
- Der B* Index unterstützt dabei vordergründig eine **Bereichssuche** im Datenbestand.

7.5.3 Indextyp: Funktionsindex

Ein funktionsbasierter Index stellt insofern eine Besonderheit dar, als nicht ein Spaltenwert indiziert wird, sondern das **Ergebnis einer Funktion**.

► Erklärung: Funktionsbasierter Index ▾

- Für den funktionsbasierten Index wird also für jede zu indizierende Zeile der Funktionswert berechnet und das Ergebnis im Index abgelegt.
- Wird nun in einer Abfrage nach einer indizierten Funktion gesucht, braucht diese Funktion nicht mehr berechnet zu werden, sondern das Ergebnis wird direkt aus dem Index gelesen.

► Syntax: create index ▾

```

1  -- -----
2  --   B* Index
3  -- -----
4  CREATE INDEX IDX_EMP_LAST_NAME
5  ON EMPLOYEES(LAST_NAME);
6
7  -- optimierte Abfrage
8  SELECT FIRST_NAME, LAST_NAME
9  FROM EMPLOYEES
10 ORDER BY LAST_NAME;
11
12
13
14
15
16
17

```

```

1  -- -----
2  -- Funktionsbasierter Index
3  -- -----
4  -- Indexdefinition
5  CREATE INDEX IDX_F_DEPARTMENT_NAME_SUBSTR
6  ON
7  DEPARTMENTS(SUBSTR(DEPARTMENT_NAME,1,1));
8
9  -- SELECT Anweisung
10 SELECT DEPARTMENT_ID, DEPARTMENT_NAME, MAN_ID
11      SUBSTR(DEPARTMENT_NAME, 1,1) ||
12 CASE
13 WHEN INSTR(DEPARTMENT_NAME, ' ', 1,1) >0
14 THEN SUBSTR(DEPARTMENT_NAME,
15             INSTR(DEPARTMENT_NAME, ' ',1,1)+1,1)
16 END
17 FROM DEPARTMENTS;

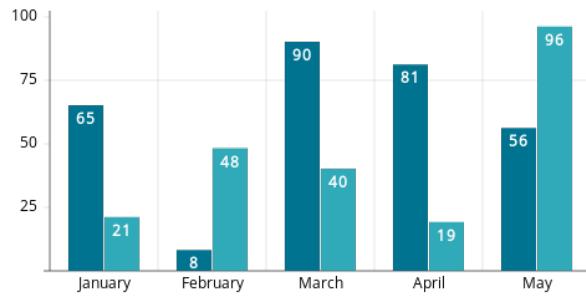
```

Indextyp	Beschreibung	Spaltentypen
B* Index	Datenbankindex zum Indexieren von Spalten mit alphanumerischen bzw. numerischen Werten. z.B.: Nachnamen bzw. Ortsnamen	varchar, number
Unique Index	Datenbankindex zum Indexieren von Spalten mit einem unique Constraint. z.B.: Spalten mit einem Primary Key Constraint.	unique Constraint
Funktionsindex	Datenbankindex zum Indexieren von Spalten deren Werte durch Funktionen transformiert werden. z.B.: lower(first_name)	
Volltextindex	Datenbankindex zum Indexieren von Spalten die umfangreiche Texte speichern.	clob, word, pdf, html

Abbildung 33. Indextypen

7.5.4 Indextyp: Unique Index

Für Datenbankspalten mit einem **unique Constraint**, wird durch die Datenbankengine automatisch ein unique Index angelegt.



► Erklärung: unique Index ▾

- Ein unique Index kann auch über mehrere Spalten hinweg definiert werden. In diesem Fall müssen die kombinierten Werte der Spalten eindeutig sein.
- In der Regel wird dieser Index Typ durch die Datenbankengine angelegt.

► Syntax: create index ▾

```

1  -- -----
2  --  unique Index
3  --
4  CREATE UNIQUE INDEX
5  idx_department_name
6  ON departments(department_name);
7
8  CREATE UNIQUE INDEX
9  IDX_PROJECT_DESCRIPTION
10 ON PROJECTS(TITEL);
```

7.5.5 Indextyp: Volltextindex

Durch die Verwendung eines Volltextindex können Volltextsuchen in relationalen Datenbanken signifikant beschleunigt werden.

Die Volltextsuche wurde entwickelt um Word, Pdf bzw. Html Dokumente, die in der Datenbank gespeichert werden, zu durchsuchen.

► Syntax: create index ▾

```

1  -- -----
2  -- Volltextindizierung: Context
3  --
4  -- Tabellendefinition
5  CREATE TABLE C_PROJECTS (
6      PROJECT_ID      NUMBER(19,0) NOT NULL,
7      MANAGER_ID      NUMBER(19,0) NOT NULL,
8      PARENT_ID       NUMBER(19,0) NOT NULL,
9      IS_FWF_FUNDED  NUMBER(1) NOT NULL,
10     IS_FFG_FUNDED  NUMBER(1) NOT NULL,
11     DESCRIPTION     VARCHAR2(4000),
12     TITLE          VARCHAR(255) NOT NULL UNIQUE,
13     PRIMARY KEY (PROJECT_ID)
14 )
15
16 CREATE INDEX idx_full_text_ctx
17 ON
18 table_name(text)
19 INDEXTYPE IS ctxsys.context;
20
21 CREATE INDEX PRO_DES_INDEX
22 ON
23 C_PROJECTS(DESCRIPTION)
24 INDEXTYPE IS CTXSYS.CONTEXT;
```

7.6. Sequenz



Sequenz ▾

Eine Sequenz ist ein Datenbankobjekt, mit dem **einzigartige** fortlaufende **Werte** generiert werden.

7.6.1 Anlegen von Sequenzen

Sequenzen werden zur **Generierung** von **Primärschlüsseln** verwendet.

► Syntax: create sequence ▾

```

1  -- -----
2  -- Syntax: create sequence
3  --
4 <CREATE SEQUENCE-Anweisung > :=
5   CREATE SEQUENCE Sequencename
6     [ INCREMENT BY integer ]
7     [ START WITH integer ]
8     [ MAXVALUE integer | NOMAXVALUE ]
9     [ MINVALUE integer | NOMINVALUE ]
10    [ CYCLE | NOCYCLE]
11    [ CACHE | NOCACHE]
12    [ ORDER | NOORDER ]
13
14  INCREMENT BY int
15  -- bestimmt die ganze Zahl, um die eine
16  -- SEQUENCE erhöht wird
17
18  START WITH int
19  -- bestimmt die ganze Zahl, mit der die
   -- SEQUENCE
20  -- startet
21
22  MAXVALUE int
23  -- obere Grenze der SEQUENCE
24
25  NOMAXVALUE
26  -- SEQUENCE hat keine obere Grenze
27
28  MINVALUE int
29  -- untere Grenze der SEQUENCE bei absteigenden
30  -- SEQUENZEN. Der DEFAULT für MINVALUE ist 1,
31  -- wenn MINVALUE nicht angegeben ist
32
33  NOMINVALUE
34  -- SEQUENCE hat keine untere Grenze
35
36  CYCLE|NOCYCLE
37  -- bestimmt, ob nach Erreichen des MAXVALUE

```

```

38  -- bzw. MINVALUE zyklisch weiter vergeben wird
39
40  CACHE int|NOCACHE
41  -- bestimmt, wie viele Schlüsselwerte im
42  -- Hauptspeicher bereitgestellt werden.
43  -- Bei NOCACHE werden keine Werte
44  -- vorgehalten

```

► DDL: create sequence ▾

```

1  -- -----
2  -- Beispiele: Sequenzen anlegen
3  --
4  CREATE SEQUENCE project_seq INCREMENT BY 1
5  STARTS WITH 1
6  NOMAXVALUE
7  NOCYCLE
8  CACHE 10;

```

7.6.2 Sequenzen verwenden

Für das Arbeiten mit **Sequenzen** verwenden wir die Attribute **curval** und **nextval**.

► DDL: Sequenzen verwenden ▾

```

1  -- -----
2  -- Sequenzen verwenden
3  --
4  INSERT INTO projects values
5  (project_seq.NEXTVAL, 'Simulation');
6
7  -- Gegenwärtigen Wert der Sequenz abfragen
8  project_seq.CURVAL
9
10 -- Nächsten Wert der Sequenz abfragen
11 project_seq.NEXTVAL

```


8. SQL - Data Manipulation Language

07

DML - Data Manipulation Language

01. DML Befehlssatz	84
02. Insert Befehl	85
03. Update Befehl	86
04. Delete Befehl	86
05. Merge Befehl	87

8.1. DML - Befehlssatz



Data Manipulation Language ▾

Die Data Manipulation Language definiert Befehle zur **Verarbeitung von Daten**.

8.1.1 DML Befehlssatz

Der **DML Befehlssatz** wird zur Verarbeitung von Daten in der Datenbank verwendet.

► Auflistung: **DML Befehlssatz** ▾



insert command ▾

Die insert Anweisung ist ein Befehl zum **Einfügen** von Daten in **Datenbanktabellen**.

Der insert Befehl besitzt mehrere Formen.



update command ▾

Die update Anweisung ist ein Befehl zum **Ändern** von Daten.

Der update Befehle besitzt mehrere Formen.



merge command ▾

Die merge Anweisung ist ein Befehl zum **Verarbeiten** von Daten. Die Anweisung ist im Kern eine Kombination eines insert und update Befehls.



delete command ▾

Die delete Anweisung ist ein Befehl zum **Löschen** von Daten.



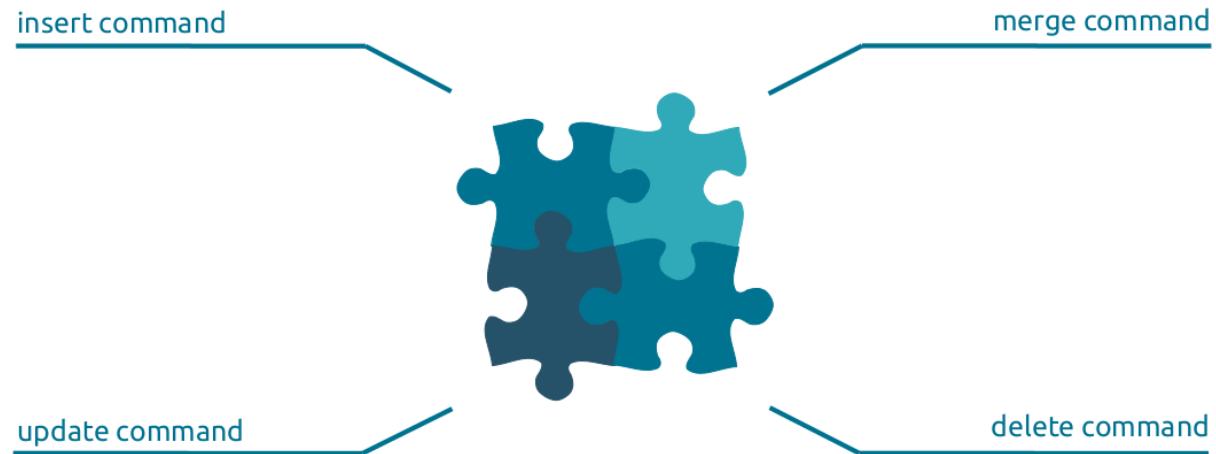


Abbildung 34. DML Befehlssatz

8.2. insert Befehl

Die insert Anweisung ist ein Befehl zum Einfügen von Daten in die Datenbank. Die SQL Spezifikation unterscheidet mehrere Formen des insert Befehls.

8.2.1 INSERT Befehl

Befehl zum **Einfügen** von Datensätzen in die Datenbank.

► DML: insert command ▾

```

1  -- -----
2  -- Syntax: INSERT
3  -- -----
4  -- Syntax 1
5  INSERT INTO table_name (column1, column2, )
6  VALUES (value1, value2, value3, ...);

7
8  -- Syntax 2
9  INSERT INTO table_name
10 VALUES (value1, value2, value3, ...);

11
12 -- Beispiel: INSERT
13
14
15  INSERT INTO groups (group_id, name) VALUES
16  (1, '1 AHIT - 2015');

17
18  INSERT INTO subjects VALUES
19  (1, 'Systemtechnik - Informationssysteme');

```

8.2.2 INSERT SELECT Befehl

Der insert - select Befehl stellt eine Variation des ursprünglichen insert Befehls dar.

Zur Migration von Daten ist es möglich den **insert** Befehl mit der **select** Anweisung zu koppeln.

► DML: insert - select command ▾

```

1  -- -----
2  -- Syntax: INSERT SELECT
3  -- -----
4  INSERT INTO table_name VALUES
5  (column1, column2)
6  SELECT value1, value2
7  FROM table_name;

8
9  -- -----
10 -- Beispiel: INSERT SELECT
11 -- -----
12 INSERT INTO c_projects
13 VALUES (
14     project_description,
15     project_title,
16     project_id,
17     is_fwf_sponsored,
18     is_ffg_sponsored
19 )
20 SELECT id, title, description, ffwf
21 FROM l_projects
22 WHERE project_type = 'REQUEST_FUNDING';

```

8.3. update Befehl

Die update Anweisung ist ein Befehl zum Ändern von Daten.

8.3.1 UPDATE Befehl

Die SQL Spezifikation definiert mehrere Formen des update Befehls.

► DML: update Syntax ▾

```

1  -- -----
2  -- Syntax: UPDATE
3  --
4  -- Andern aller Werte einer Tabelle
5  UPDATE table_name
6  SET column1 = value1, column2 = value2, ...;
7
8  -- Andern bestimmte Werte einer Tabelle
9  UPDATE table_name
10 SET column1 = value1, column2 = value2, ...
11 WHERE predicate;
```

► DML: Fallbeispiel Schule ▾

```

1  -- -----
2  -- Beispiel: UPDATE
3  --
4  UPDATE students SET last_name = 'Lang'
5  WHERE student_id = 1;
6
7  UPDATE students
8  SET last_name, = 'Dreger',
9      first_name = 'Nikolaus'
10 WHERE student_id = 3
11
12 UPDATE employees
13 SET salary = (
14     SELECT avg(salary) FROM employees
15 )
16 WHERE department_id = 103;
17
18 UPDATE employees
19 SET salary = (
20     SELECT max(salary) FROM employees
21 )
22 WHERE department_id IN (
23     SELECT department_id
24     WHERE DEPARTMENTS_ID = 101;
25 )
```

8.4. delete Befehl

Die delete Anweisung wird zum Löschen von Daten verwendet.

8.4.1 DELETE Befehl

Mit der delete Anweisung können ein oder mehrere Datensätze aus einer Tabelle gelöscht werden.

Die **Bedingung** der where Klausel steuert dabei welche Datensätze vom Löschvorgang betroffen sind.

► DML: delete Syntax ▾

```

1  -- -----
2  -- Syntax: DELETE
3  --
4  DELETE FROM table_name
5  WHERE [condition];
```

► DML: Fallbeispiel HR ▾

```

1  -- -----
2  -- Beispiel: DELETE
3  --
4  -- Loeschen Sie den Angestellten mit der
5  -- ID 10
6  DELETE FROM employees
7  WHERE employee_id = 10;
8
9  -- Loeschen Sie alle Angestellten die in
10 -- der SALES Abteilung arbeiten
11 DELETE FROM employees
12 WHERE employee_id in (
13     SELECT employee_id
14     FROM employees e JOIN DEPARTMENTS D
15     ON e.department_id = d.department_id
16     WHERE department_name = 'SALES';
17 );
18
19 -- Loeschen Sie alle Angestellten die in
20 -- Saudi Arabien arbeiten.
21 DELETE FROM employees
22 WHERE employee_id in (
23     SELECT employee_id from employees e
24     JOIN departments d on e.dep_id = d.dep_id
25     JOIN locationos l on d.loc_id = l.loc_id
26     WHERE l.country_name = 'Saudi Arabia'
27 );
```

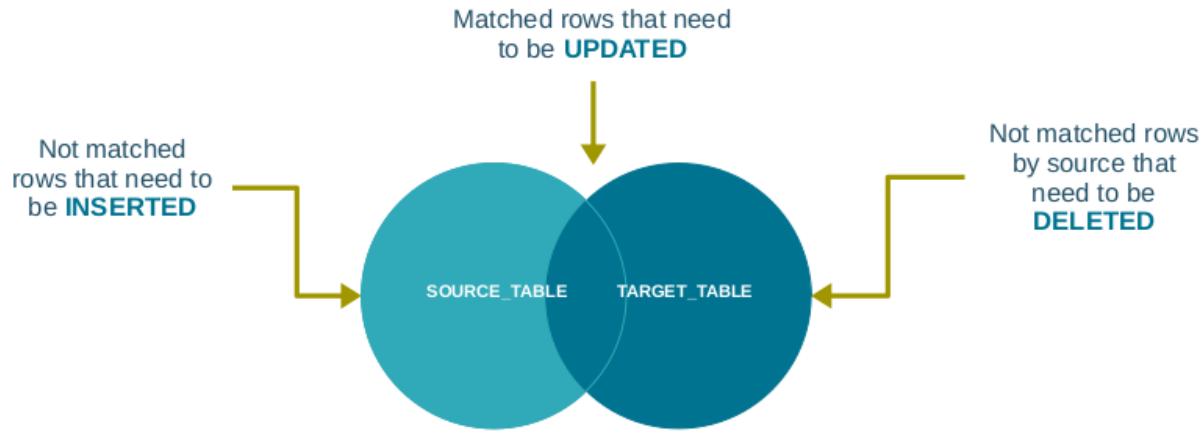


Abbildung 35. MERGE Befehl

8.5. merge Befehl

Mit der merge Anweisung können Daten auf Basis einer Bedingung mittels einer einzigen Anweisung eingefügt, aktualisiert bzw. gelöscht werden.

8.5.1 Fallbeispiel: Verkauf

Für ein Unternehmen soll eine Berichtstabelle der täglichen Verkaufsstatistiken erstellt werden.

► Erklärung: Vorgang Berichtswesen ▾

- Im klassischen Szenario werden die Datensätze dazu zuerst in eine Stagingtabelle geladen.
- Anschließend wird eine Reihe von **DDL Befehlen** ausgeführt, um die täglichen Verkaufsdaten zu aktualisieren.
- Mit der MERGE Anweisung kann die Stagingtabelle mittels einer einzigen Anweisung aktualisiert werden, anstatt verschiedene DDL Anweisungen benutzen zu müssen.

8.5.2 MERGE Befehl

Der Mergebefehl wird zum **Synchronisieren** zweier Datenquellen verwendet.

► DML: merge command ▾

```

1  -- -----
2  -- Syntax: MERGE
3  --
4  MERGE INTO target_table USING source_table
5  ON merge_condition
6  WHEN MATCHED
7      THEN update_statement
8  WHEN NOT MATCHED
9      THEN insert_statement;
10 --
11 --
12 -- Fallbeispiel: MERGE
13 --
14 -- target table
15
16 CREATE TABLE CATEGORIES (
17     category_id    NUMBER(19,0) NOT NULL,
18     category_name VARCHAR(100) NOT NULL,
19     amount         NUMBER(5,2) NOT NULL,
20     PRIMARY KEY (category_id)
21 );

```



8.5.3 Fallbeispiel: MERGE

► DML: merge command ▾

```

1  -- -----
2  -- Fallbeispiel: MERGE
3  --
4  -- source table (staging table)
5  CREATE TABLE CATEGORIES_STAGING (
6      category_id    NUMBER(19,0) NOT NULL,
7      category_name VARCHAR(100) NOT NULL,
8      amount         NUMBER(5,2) NOT NULL,
9      PRIMARY KEY (category_id)
10 );
11
12 INSERT INTO CATEGORIES
13 VALUES(1,'Children Bicycles', 15000),
14      (2,'Comfort Bicycles', 25000),
15      (3,'Cruisers Bicycles', 13000),
16      (4,'Cyclocross Bicycles',10000);
17
18 INSERT INTO CATEGORIES_STAGING
19 VALUES(1,'Children Bicycles',15000),
20      (3,'Cruisers Bicycles',13000),
21      (4,'Cyclocross Bicycles',20000),
22      (5,'Electric Bikes',10000),
23      (6,'Mountain Bikes',10000);
24
25 --
26 -- MERGE Befehl
27 --
28 MERGE CATEGORIES t
29 USING CATEGORIES_STAGING s
30   ON (s.category_id = t.category_id)
31 WHEN MATCHED THEN UPDATE SET
32     t.category_name = s.category_name,
33     t.amount = s.amount
34 WHEN NOT MATCHED THEN
35   INSERT (category_id, category_name, amount)
36   VALUES (
37     s.category_id,
38     s.category_name,
39     s.amount
40 );
41
42 -- 1 15000 Children Bicycles
43 -- 3 13000 Cruisers Bicycles
44 -- 4 20000 Cyclocross Bicycles
45 -- 5 10000 Electric Bikes
46 -- 6 10000 Mountain Bikes

```



9. SQL - Data Control Language

08

DCL- Data Control Language

01. DCL Befehlssatz	90
02. create User	90
03. grant	91

9.1. DCL Befehle



Data Control Language ▾

Die **DCL** definiert Befehle zum Vergeben und Entziehen von Berechtigungen.

DCL ist die **Datenüberwachungssprache** relationaler Datenbanken.

9.1.1 DCL Befehlssatz

Der **DCL Befehlssatz** definiert Befehle zur Verwaltung von Benutzerrechten für die Datenbank.

► Auflistung: DCL Befehlssatz ▾



create user ▾

Befehlsgruppe zum **Anlegen** von Benutzern.



grant ▾

Befehlsgruppe zum **Zuordnen** von Benutzerrechten.



revoke ▾

Befehlsgruppe zum **Zurücknehmen** von Benutzerrechten

9.2. create user Befehl

Befehlsgruppe zum **Anlegen** und Verwalten von Benutzern.

9.2.1 Benutzerverwaltung

Die **Benutzerverwaltung** ist der **Steuerungsprozess**, welche Benutzer sich mit der Datenbank verbinden dürfen und welche **Rechte** sie für die Datenbank haben.

Für ein Unternehmen wäre es schwer ein **Sicherheitssystem** für seine Anwendungen zu implementieren, wenn die Datenbank keine Möglichkeit für die Durchsetzung von Benutzerrechten bieten würde.

```

          user           IP Adresse          Passwort
CREATE USER 'htl-administrator'@'localhost' IDENTIFIED BY 'htl-pa33w09d'
                                         create user Befehl

```

Abbildung 36. create user Befehl

9.2.2 create User Befehl

Der `create user` Befehl besteht aus mehreren Segmenten.

► Erklärung: Segmente `create user` ▾

- **Schlüsselwörter:** Die Schlüsselwortsegmente `create user` und `identified by` trennen das Benutzersegment vom Passwortsegment.
- **Benutzername:** Das Benutzernamesegment besteht aus 2 Teilen: Dem Namen des Benutzers und der IP Adresse des Hosts von dem er sich verbindet.
- **Passwort:** Das Passwortsegment dient zur Angabe des Passworts des Benutzers.

► DCL: Benutzerverwaltung ▾

```

1  --
2  -- Syntax: create User
3  --
4  -- Anlegen des users htl-admin
5  create user 'htl-admin'@'127.0.0.1'
6  identified by 'htl-12345';
7
8  create user 'htl-admin'@'%',
9  identified by 'htl145';

10
11 -- Benutzer anzeigen
12 select * from mysql.user;
13
14 --
15 -- Syntax: mysqladmin
16 --
17 -- Aendern des Passworts des Admins
18 mysqladmin --user=root
19           --password=old
20           password new
21
22 --
23 -- Syntax: drop user
24 --
25 drop user user_name;

```

9.3. grant Befehl

Befehlsgruppe zum Zuordnen von Benutzerrechten.

9.3.1 Privilegien zuordnen

Damit ein **Benutzer** bestimmte **Aktionen** in der Datenbank ausführen kann werden ihm **Privilegien** zugeordnet.

► Auflistung: Privilegien ▾

- **create:** Erlaubt einem Benutzer, neue **Tabellen** zu erstellen.
- **delete:** Erlaubt einem Benutzer, Zeilen in einer **Tabelle** zu löschen.
- **insert:** Erlaubt einem Benutzer, neue **Zeilen** in eine **Tabelle** zu schreiben.
- **select:** **Leseberechtigungen** auf eine Datenbank oder **Tabelle**.
- **update:** Erlaubnis eine Zeile zu aktualisieren.
- **all privileges:** Ein **Wildcard** für alle Rechte auf das gewählte **Datenbankobjekt**.



9.3.2 Syntax: grant Befehl

► DCL: Rechteverwaltung ▾

```

1  --
2  -- Syntax: grant
3  --
4  -- Anatomie des grant Befehls
5  GRANT privilege ON table_name TO user@host

```

9.3.3 grant Befehl

► DCL: Rechteverwaltung ▾

```
1  -- -----
2  -- Syntax: grant
3  --
4  -- Tabellen fuer Datenbank erstellen
5  GRANT create ON projects.* to
6  'htl-admin'@'localhost';
7
8  -- Tabellen auslesen
9  GRANT select ON projects.subprojects to
10 'htl-admin'@'localhost';
11
12 -- Alle Rechte fuer eine Datenbank zuordnen
13 GRANT all privileges ON projects.* to
14 'htl-admin'@'localhost';
15
16 -- -----
17 -- Rechte anzeigen
18 --
19 SHOW GRANTS FOR 'htl-admin'@'localhost';
20
21 -- -----
22 -- Rechte zuordnen
23 --
24 FLUSH PRIVILEGES;
25
26 -- -----
27 -- Rechte zuruecknehmen
28 --
29 REVOKE select ON projects.subprojects FROM
30 'htl-admin'@'localhost';
```



Programmierung relationaler Datenbanken

August 19, 2020

10. PL/SQL - Grundlagen

01

PL/SQL Grundlagen

01. Grundlagen 94

02. Block 98

10.1. PL/SQL Grundlagen ▾

PL/SQL als Programmiersprache stellt eine **prozedurale Erweiterung** der SQL Sprachspezifikation dar.

10.1.1 PL/SQL

PL/SQL kombiniert den **SQL Befehlssatz** mit Strukturen zur Entscheidungsfindung und Schleifenkonstrukten.

PL/SQL Programme haben **direkten Zugriff** auf die Strukturen und Daten einer Datenbank.

► Erklärung: PL/SQL Grundlagen ▾

- Durch die **Integration** des SQL Befehlssatz in der PL/SQL Sprachspezifikation haben PL/SQL Programme direkten Zugriff auf die Daten und Strukturen einer Datenbank.
- Damit wird es möglich Teile der **Anwendungslogik** vom Applikationsserver in den Datenbankserver zu verlagern.
- Die **Verlagerung** der Anwendungslogik in den Datenbankserver, resultiert dabei in einer signifikanten Steigerung der Verarbeitungsgeschwindigkeit datenzentrierter Anwendungen.
- Der **Datenbankzugriff** erfolgt in PL/SQL Programme dabei gekapselt über die PL/SQL Sprachchnittstelle in Form von **Modulen**.
- Logisch zusammenhängende Routinen werden dazu in **Funktionen** und **Prozeduren** zusammengefaßt.
- PL/SQL Programme können direkt in einer Datenbank gespeichert und ausgeführt werden.
- Zur Ausführung von PL/SQL Programmen verwendet eine Datenbank ein eigenes Programm: die **PL/SQL Engine**.
- Gegenwärtig können PL/SQL Programme nur in Oracle Datenbanken ausgeführt werden. Hersteller anderer Datenbanken unterstützen nur Teile des PL/SQL Standards.
- Neben dem direkten Datenzugriff unterstützt PL/SQL eigene Mechanismen zur Transaktionssteuerung, Cursor- und Fehlerbehandlung.



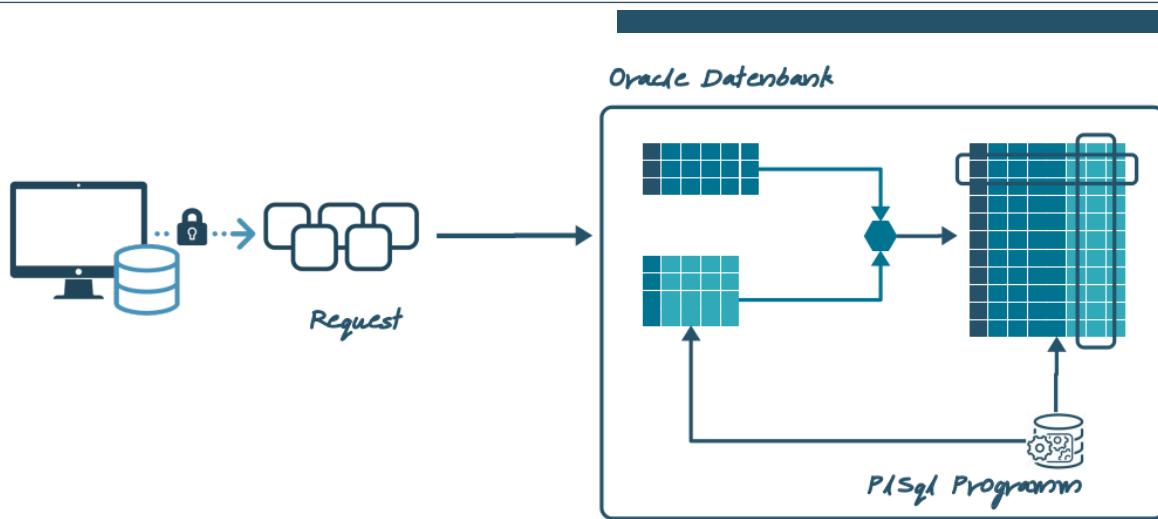


Abbildung 37. PL/SQL Ausführung

10.1.2 PL/SQL Sprachspezifikation

PL/SQL als Programmiersprache wurde für den Einsatz in Datenbanken konzipiert.

► Erklärung: PL/SQL Sprachspezifikation ▾

- Der **Datenbankzugriff** in PL/SQL Programme erfolgt gekapselt über die PL/SQL Sprachschnittstelle in Form von Prozeduren bzw. Funktionen.
- **Deklarative Anweisungen** können in PL/SQL Prgrammen als Programmelemente in einer Datenbank gespeichert und ausgeführt werden. Der **Datenbankzugriff** kann damit als freie Abfolge von Anweisungen und Routinen formuliert werden.
- Ein PL/SQL Programm kann im Grunde als **strukturelle Hülle** des SQL Befehlssatzes abstrahiert werden. Prominentester Befehl eines PL/SQL Programms bleibt die select Anweisung.
- Neben dem direkten Datenzugriff unterstützt PL/SQL eigene Mechanismen zur **Transaktionssteuerung**, Cursor- und Fehlerbehandlung.



10.1.3 Einsatzgebiete von PL/SQL

PL/SQL Programme erfüllen in der Datenbankenentwicklung ein weites Feld von Aufgaben.

► Auflistung: Einsatzgebiete von PL/SQL ▾



Datenbankfunktionalität ▾

Mit PL/SQL Programmen kann die SQL **Sprachspezifikation** um zusätzliche Funktionen erweitert werden.



Datenkonsistenz ▾

Die Sicherstellung der **Konsistenz** des Datenbestandes gehört zu den grundlegenden Aufgaben eines Informationssystems. PL/SQL abstrahiert die **Integrationsprüfung** dazu in Form von ereignisgesteuerten Routinen.



Datensicherheit ▾

PL/SQL stellt dem Datenbankentwickler ein weites Feld von **Authorisierungskonzepten** zur Verfügung.

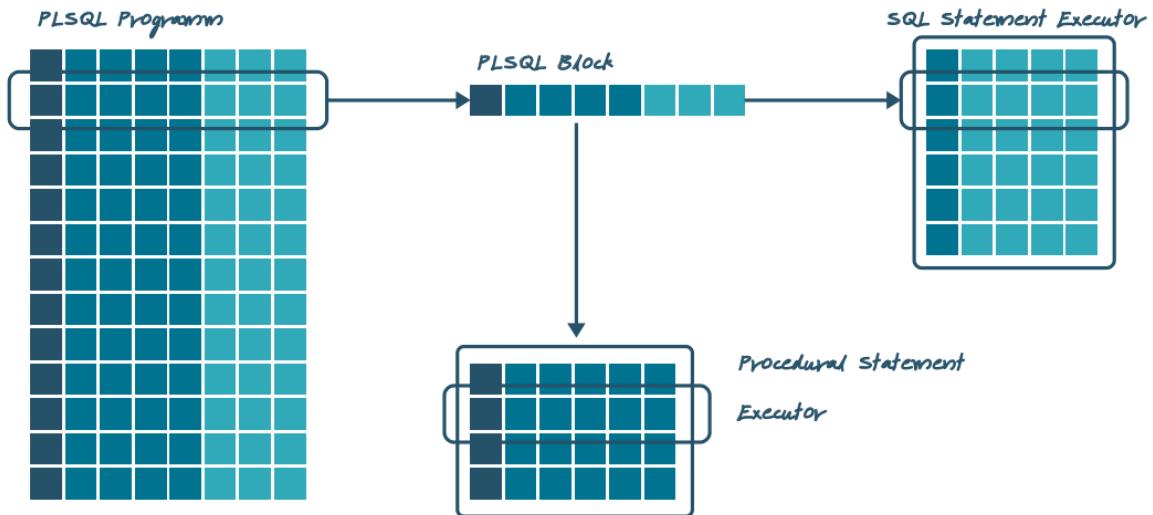


Abbildung 38. Kontextswitch pl/sql Engine

🔧 **Datenzugriff** ▾
 Unterliegt die **Verarbeitung** der Daten einer Anwendung einer komplexen Geschäftslogik, muß der direkte Zugriff auf den Datenbestand verhindert werden.
 Der Datenzugriff erfolgt in solchen Fällen mit **PL/SQL Prozeduren**.

⚙️ **Datenmigration** ▾
 Eine Datenmigration bzw. Portierung beschreibt alle Prozesse die notwendig sind um die Daten eines Schemas in ein anderes Schema zu überführen.

🏢 **Anwendungsentwicklung** ▾
 Die Verantwortung datenzentrierter **Anwendungen** liegt in der Datenerfassung bzw. Datenpräsentation.
 Anwendungen solcher Art können in Gänze in PL/SQL programmiert werden.

10.1.4 PL/SQL Engine

PL/SQL Programme werden in Datenbanken in einem eigenem Prozess, der **PL/SQL Engine** ausgeführt.

► Erklärung: PL/SQL Engine ▾

- Ein PL/SQL Programm stellt eine freie Abfolge von PL/SQL Anweisungen und SQL Befehlen dar.
- PL/SQL Programme können im Grunde als strukturelle Hülle für SQL Befehle verstanden werden.
- Im Gegensatz zu PL/SQL Blöcken, werden SQL Befehl jedoch nicht von der PL/SQL Engine ausgeführt. Die Befehle werden zur Ausführung an einen eigenen Prozess, den **SQL Statement Executor** weitergeleitet.
- Der Aufruf des SQL Statement Executors wird als **Kontextswitch** bezeichnet. Ein Kontextswitch ist dabei mit einem gewissen Ressourcenaufwand verbunden.

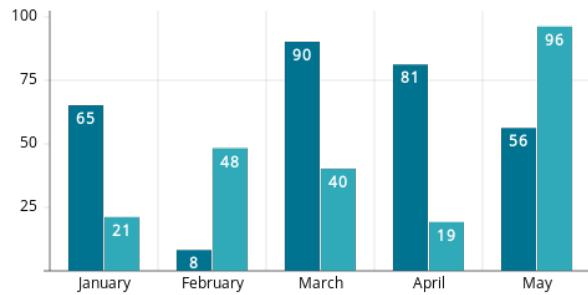


10.1.5 PL/SQL Block

PL/SQL ist eine **blockorientierte** Programmiersprache.

► Erklärung: PL/SQL Block ▾

- Das grundlegende Element eines PL/SQL Programms ist der **PL/SQL Block**.
- PL/SQL Blöcke erfüllen in PL/SQL Programmen 2 Aufgaben:
 - **Codestrukturierung:** PL/SQL Blöcke beschreiben die **Struktur** von PL/SQL Programmen.
 - **Programmausführung:** PL/SQL Blöcke können von der PL/SQL Engine ausgeführt werden.



► Syntax: PL/SQL Block ▾

```

1  -- -----
2  -- Syntax: Anonymer Block
3  --
4  BEGIN
5    Executable Statements
6    ...
7  END;
8
9  --
10 -- Codebeispiel: Block
11 --
12 BEGIN
13   DBMS_OUTPUT.PUT_LINE('Hello World');
14 END;
15
16 BEGIN
17   FOR COUNTER IN 1..10 LOOP
18     DBMS_OUTPUT.PUT_LINE(COUNTER);
19   END LOOP;
20 END;

```

10.1.6 Fallbeispiel: PL/SQL Programm

► Codebeispiel: PL/SQL Programm ▾

```

1  -- -----
2  -- Codebeispiel: plsql Programm
3  --
4  DECLARE
5    -- DEFINITION: Tabellentyp
6    TYPE PROJECT_TABLE_TYPE IS TABLE OF
7      HR.L_PROJECTS%ROWTYPE;
8
9    -- DEKLARATION: Virtuelle Tabelle
10   L_PROJECTS PROJECT_TABLE_TYPE :=
11     PROJECT_TABLE_TYPE();
12
13   -- DEFINITION: Strukturierter Datentyp
14   TYPE PROJECT_REPORT_TYPE IS RECORD (
15     TITLE HR.L_PROJECTS.TITLE%TYPE,
16     DESCRIPTION VARCHAR(4000),
17     PROJECT_ID NUMBER(19,2)
18   );
19
20   -- DEFINITION: Tabellentyp
21   TYPE PR_TABLE_TYPE IS TABLE OF
22     PROJECT_REPORT_TYPE;
23
24   -- DEKLARATION: Virtuelle Tabelle
25   L_PROJECT_REPORTS PR_TABLE_TYPE :=
26     PR_TABLE_TYPE();
27
28   BEGIN
29     SELECT L.ID, L.FUNDING, L.TITLE
30       L.DESCRIPTION, 1
31     BULK COLLECT INTO L_PROJECT_REPORTS
32     FROM HR.L_PROJECTS L
33     WHERE L.PROJECT_TYPE = 'STIPENDIUM';
34
35     FOR I IN 1..L_PROJECT_REPORTS.COUNT
36     LOOP
37       DBMS_OUTPUT.PUT_LINE(
38         'project data: '
39         || L_PROJECTS(I).TITLE
40         || chr(13)
41         || L_PROJECTS(I).DESCRIPTION
42         || chr(13)
43         || L_PROJECTS(I).PROJECT_TYPE
44       );
45     END LOOP;
46
47   END;

```

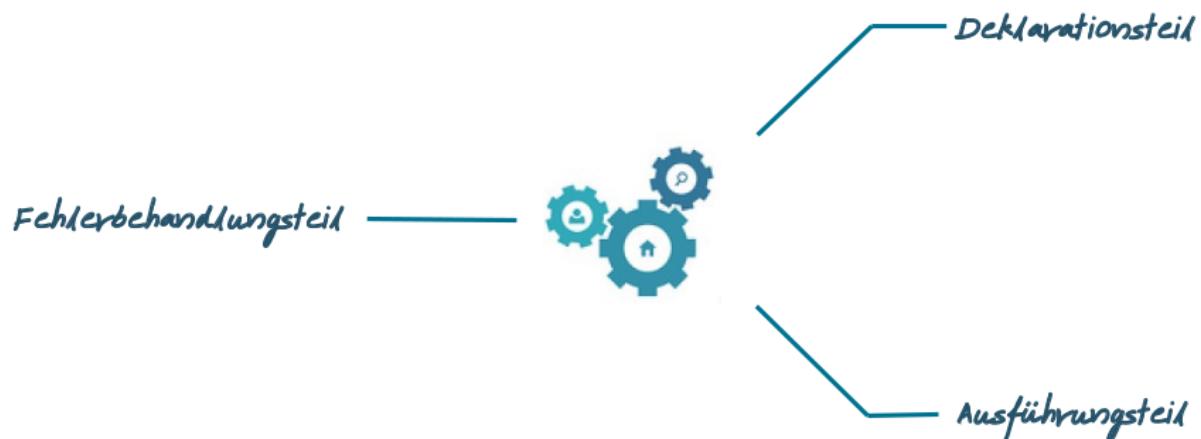


Abbildung 39. Aufbau eines plsql Blocks

10.2. PL/SQL Block

10.2.1 PL/SQL Block

PL/SQL Block ▾

PL/SQL Blöcke sind die grundlegenden Elemente eines PL/SQL Programms:

- PL/SQL Blöcke definieren die **Struktur** von PL/SQL Programmen.
- PL/SQL Blöcke können im Kontext einer Datenbank ausgeführt werden.

Strukturell gliedert sich ein PL/SQL Block in drei Teile.

► Erklärung: Aufbau eines Blocks ▾

- **Deklarationsteil:** Der Deklarationsteil eines Blocks enthält die für ein Programm erforderlichen **Vereinbarungen** in Form von Variablen, Konstanten, Cursor bzw. Fehlerzuständen. Der Deklarationsteil wird mit den Schlüsselwörtern **declare** bzw. **is** oder **as** eingeleitet. Der Deklarationsteil ist ein **optionaler** Bestandteil eines Blocks.
- **Ausführungsteil:** Der Ausführungsteil enthält die eigentliche **Funktionalität** eines PL/SQL Programms als freie Abfolge von Anweisungen und Befehlen. Der Ausführungsteil wird durch das Schlüsselwort **begin** eingeleitet.

Der Ausführungsteil ist **obligatorisch**.

- **Fehlerbehandlungsteil:** Tritt im Ausführungsteil eines PL/SQL Blocks ein **Fehler** auf, verzweigt die Ausführung des PL/SQL Programms in den Fehlerbehandlungsteil des Blocks. Der Fehlerbehandlungsteil wird durch das Schlüsselwort **exception** eingeleitet.

Der Fehlerbehandlungsteil ist **optional**.

► Codebeispiel: Anonymer Block ▾

```

1 -- -----
2 -- Syntax: Anonymer Block
3 --
4 DECLARE
5     Declaration Statements
6 BEGIN
7     Executable Statements
8 EXCEPTION
9     Exception handling Statements
10 END
11 --
12 -- -----
13 -- Codebeispiel: Block
14 --
15 DECLARE
16     L_INDEX NUMBER DEFAULT 10;
17 BEGIN
18     ...
19 EXCEPTION
20     ...
21 END;
22 
```



Baustein	Beschreibung	Vorkommen
Deklarationsteil	Der Deklarationsteil enthält die Vereinbarungen der für das Programm erforderlichen Variablen, Konstante, Cursor bzw. Fehlerzustände.	optional
Ausführungsteil	Der Ausführungsteil enthält die eigentliche Funktionalität eines PL/SQL Programms als freie Abfolge von Anweisungen und Befehlen.	obligatorisch
Fehlerbehandlungsteil	Im Fehlerbehandlungsteil wird die Fehlerbehandlung eines PL/SQL Programms implementiert.	optional

Abbildung 40. Blockstruktur

10.2.2 Anonymer Block

Ein PL/SQL Programm ist grundsätzlich immer als **Blockstruktur** aufgebaut.

► Erklärung: Anonymer Block ▾

- Die PL/SQL Spezifikation unterscheidet 2 Arten von **Blocktypen**: anonyme Blöcke und benannte Blocktypen.
- Ein alterer Block besitzt keine **Definitionsvereinbarung** und keinen Namen und kann daher nicht aus anderen PL/SQL Programmen heraus aufgerufen werden.
- Anonyme Blöcke können in einer Datenbank direkt ausgeführt werden.
- Anonyme Blöcke können im Vergleich zu benannten Blocktypen jedoch nicht in Datenbanken gespeichert werden.

► Codebeispiel: Anonymer Block ▾

```

1  -- -----
2  --  Anonymer Block
3  --
4  DECLARE
5      L_LAST_NAME VARCHAR2(50) := 'Washington';
6      L_FIRST_NAME VARCHAR2(50) := 'Denzel';
7  BEGIN
8      DBMS_OUTPUT.PUT_LINE(
9          'person data: '
10         || L_FIRST_NAME
11         || ', '
12         || L_LAST_NAME
13     );
14 END;

```

10.2.3 Benannte Blocktypen

Die PL/SQL Spezifikation definiert mehrere Typen von benannten Blocken.

► Auflistung: Arten von Blöcken ▾



Package Block ▾

Ein Package faßt logisch zusammenhängende Objekte zu einer modularen Einheit zusammen.

Packages werden zur **Strukturierung** von PL/SQL Programmen verwendet.



Procedure Block ▾

Eine Prozedur beschreibt eine freie Abfolge von Anweisungen und Routinen. Prozeduren können in der PL/SQL Engine ausgeführt werden. Prozeduren werden zur **Strukturierung** von PL/SQL Programmen verwendet.



Function Block ▾

PL/SQL Funktionen sind **Prozeduren**, die einen **Rückgabewert** an den Aufrufer der Funktion zurückgeben.



Trigger Block ▾

Trigger sind eine besondere Form von Prozeduren, die in Reaktion auf ein bestimmtes Datenbankinternes Ereignis ausgelöst werden.

11. PL/SQL - Datentypen

02

PLSQL Datentypen

01. Datentypen und Variablen	100
02. Basisdatentypen	102
03. Abgeleitete Datentypen	105
04. Strukturierte Datentypen	105
05. Tabellentypen	106

11.1. Datentypen und Variablen

Variablen werden in Programmen zur Verwaltung von Daten verwendet.

11.1.1 Definition von Variablen

Variablen haben einen Namen und einen Datentyp.

► Erklärung: Definition von Variablen ▾

- Bevor eine Variable in einem PL/SQL Programm verwendet werden kann, muss sie im Deklarationsteil eines PL/SQL Blocks definiert werden.
- Der Datentyp einer Variable bestimmt dabei den Wertebereich und die Operationen die für die Variable zulässig sind.
- Mit dem Zuweisungsoperators := wird einer Variable ein Wert zugewiesen.

► Codebeispiel: Definition von Variablen ▾

```

1  -- -----
2  -- Definition von Variablen
3  --
4  DECLARE
5      -- Definition einer Variable
6      -- Name: l_tax_rate
7      -- Typ: NUMBER
8      L_TAX_RATE NUMBER;
9
10     -- Definition und Initialisierung
11     -- Name: l_index_high
12     -- Typ: NUMBER
13     -- Initialisierungswert: 10
14     L_INDEX_HIGH NUMBER := 10;
15
16     -- Definition einer Konstante
17     -- Name: l_day_amount
18     -- Typ: NUMBER
19     -- Wert: 30
20     L_DAY_AMOUNT CONSTANT NUMBER := 30;
21 BEGIN
22     DBMS_OUTPUT.PUT_LINE(
23         'data output: '
24         || 'tax rate:' || L_TAX_RATE
25         || L_INDEX_HIGH
26     );
27 END;

```



Datentypen



11.1.2 Gültigkeit von Variablen

Variablen gelten in jenem Block, indem sie definiert wurden.

► Erklärung: Gültigkeit von Variablen ▾

- Blöcke können in PL/SQL Programmen ineinander verschachtelt werden.
- Um **Namenskonflikte** bei der Vergabe von Variablennamen zu vermeiden, verborgen Variablen innerer Blöcke, Variablen umgebender Blöcke.

► Codebeispiel: Gültigkeit von Variablen ▾

```

1  -- -----
2  --  Gueltigkeit von Variablen
3  --
4  DECLARE
5      L_NAME VARCHAR2(20) := 'A. Gusenbauer';
6  BEGIN
7      -- GUELTIGKEIT: Innerer Block
8      DECLARE
9          L_NAME VARCHAR2(20);
10     BEGIN
11         L_NAME := 'Alfred Peter';
12
13         DBMS_OUTPUT.PUT_LINE(
14             'kundenname: ' || L_NAME
15             || chr(13)
16             || chr(13)
17         );
18     END;
19
20     DBMS_OUTPUT.PUT_LINE(L_NAME);
21 END;

```

11.1.3 Qualifier



Qualifier ▾

Ein Qualifier ist ein logischer **Verweis** auf einen PL/SQL Block. Als Qualifier wird der Name des Blocks verwendet.

Der **Zugriff** auf einen Qualifier erfolgt über den Namen eines PL/SQL Blocks.

► Codebeispiel: Qualifier ▾

```

1  -- -----
2  --  Qualifier
3  --
4  DECLARE
5      L_COUNT PLS_INTEGER DEFAULT 0;
6  BEGIN
7      DECLARE
8          L_INNER NUMBER;
9      BEGIN
10         SELECT COUNT(PROJECT_ID)
11         INTO LOCAL_BLOCK.L_INNER
12         FROM PROJECTS;
13
14         INIT.L_COUNT := LOCAL_BLOCK.L_INNER;
15     END LOCAL_BLOCK;
16
17     PROJECT.G_PROJECT_COUNT := INIT.L_COUNT;
18
19     DBMS_OUTPUT.PUT_LINE(
20         'PROJECT COUNT: ' || INIT.L_COUNT
21     );
22 END init;

```

11.1.4 Datentypen

Ein Datentyp bestimmt den **Wertebereich** und die **Operationen**, die für eine Variable zulässig sind.

► Auflistung: Arten von Datentypen ▾



Basisdatentypen ▾

Als Basisdatentypen wird eine Gruppe von **Datentypen**, zur Verwaltung von Zahlen, einzelner Zeichen bzw. logischer Werte bezeichnet.



Abgeleitete Datentypen ▾

Neben der direkten Angabe eines Datentyps, kann in PL/SQL Programmen der Datentyp einer Variable implizit **referenziert** werden.

Wir sprechen in diesem Fall von **abgeleiteten Datentypen**.



Strukturierte Datentypen ▾

Strukturierte Datentypen fassen logisch zusammenhängende Basisdatentypen zu einem **strukturellen Verbund** zusammen.



Tabellentypen ▾

Tabellentypen ermöglichen die Verwaltung von Werten in einer **Tabelle**.

11.2. Basisdatentypen



Basisdatentypen ▾

Als Basisdatentypen wird eine Gruppe von Datentypen zur Verwaltung von Zahlen, einzelner Zeichen bzw. logischer Werte bezeichnet.

Die PL/SQL Spezifikation definiert 4 **Basisdatentypen**.

► Auflistung: Kategorien ▾

- **Numerische Datentypen:** Numerische Datentypen kommen bei der Verarbeitung von Zahlenwerten zum Einsatz.
- **Alphanumerische Datentypen:** Alphanumerische Datentypen kommen bei der Verarbeitung von Zeichenketten zum Einsatz.
- **Zeitzogene Datentypen:** Zeitbezogene Datentypen kommen bei der Verarbeitung von Datumswerten zum Einsatz.
- **Boolsche Datentypen:** Eine boolsche Variable wird verwendet um **Wahrheitswerte** zu verarbeiten.



11.2.1 Numerische Datentypen

Numerische Werte: Number ist der grundlegende Datentyp zur Verwaltung numerischer Datenwerte.

► Codebeispiel: Numerische Datentypen ▾

```

1  -- -----
2  -- Datentyp: numerische Werte
3  --
4  DECLARE
5      L_WEEKS_TO_PAY NUMBER(2) := 52;
6      L_RAISE_FACTOR NUMBER := 0.05;
7      L_SALARY NUMBER(2,9) := 1234567.89;
8      L_FACTOR NATURAL DEFAULT 0;
9  BEGIN
10     DBMS_OUTPUT.PUT_LINE(
11         L_WEEKS_TO_PAY
12         || chr(13)
13         || L_RAISE_FACTOR
14         || chr(13)
15         || L_SALARY
16     );
17 END;

```



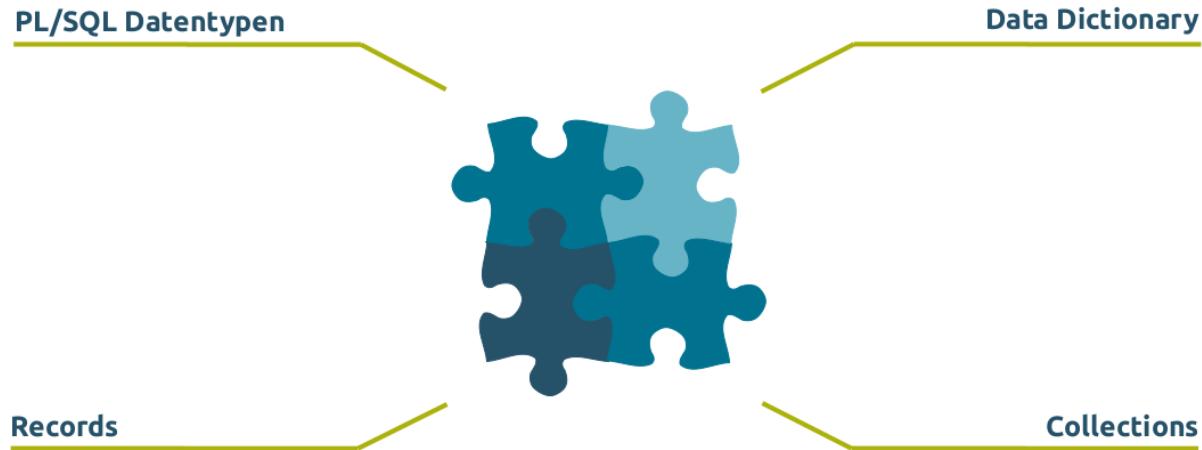


Abbildung 41. Typen von Datentypen

► Codebeispiel: Precision und Scale ▾

```

1  -- -----
2  --  Datentyp: Precision und Scale
3  --
4  DECLARE
5      L_V1_NR NUMBER(10)    := 2.567;
6      L_V2_NR NUMBER(5,2)   := 123.567;
7      L_V3_NR NUMBER        := 2/3;
8      L_V4_NR NUMBER(5,2)   := 2/3;
9
10     L_V5_NR PLS_INTEGER := 201;
11     L_V6_NR BINARY_FLOAT := 323.332;
12     L_V7_NR BINARY_DOUBLE := 323321.32
13
14 BEGIN
15     DBMS_OUTPUT.PUT_LINE(
16         L_V1_NR
17         || 'v2: ' || L_V2_NR
18         || 'v3: ' || L_V3_NR
19         || 'v4: ' || L_V3_NR * 3
20         || 'v5: ' || L_V4_NR * 3
21     );
22
23     -- Ausgabe
24     v1: 3
25     v2: 123,57
26     v3: 2
27     v4: 2.01
28     v5: 201
29     v6: 323.332
30     v7: 323321.32

```

11.2.2 Alphanumerische Datentypen

Alphanumerische Datentypen kommen bei der Verarbeitung von Zeichenketten zum Einsatz.

Verwenden Sie den VARCHAR2 Datentyp zur Verarbeitung von Zeichenketten in PlSql.

► Auflistung: PL/SQL Datentypen ▾

- **Zeichenketten - Char:** Datentypen zur Verwaltung alphanumerischer Werte, besitzen 2 Ausprägungen: Datentypen mit einer variablen und Datentypen mit einer festgelegten Länge.

Char² ist ein Datentyp mit **festgelegter** Länge. Bei einem Datentyp mit festgelgter Länge wird die gespeicherte Zeichenkette solange mit Leerzeichen aufgefüllt bis die Zeichenkette die in der Varaiblendefinition angegebene Länge hat.

- **Zeichenketten - Varchar2:** Varchar2 ist ein Datentyp mit **variabler** Länge. Zeichenketten die einer Variable eines Datentyps variabler Länge zugeordnet werden benötigen in der Datenbank genauso viel Platz wieviele Zeichen sie beinhalten.

Varchar2 ist der wichtigste Datentyp zur Verarbeitung von Zeichenketten.



² Char ist ein veralteter Datentyp und hat in der PlSQL Programmierung keine Bedeutung mehr.

Datentyp	Beschreibung	Zuordnung
NUMBER	Numerischer Datentyp zur Verwaltung von Gleitkommazahlen	Double
NUMBER(10)	Numerischer Datentyp zur Verwaltung ganzzahliger Zahlenwerte - 10 stelliger Integer	Integer
NUMBER(2,9)	Numerischer Datentyp zur Verwaltung von Fixpunktzahl . - Fixpunktzahl mit 7 Stellen vor dem Komma und 2 Stellen nach dem Komma.	Float
PLS_INTEGER	Numerischer Datentyp zur Verwaltung von ganzzahligen Zahlenwerten. PLS_INTEGER ist ein PLSQL interner Datentyp. PLS_INTEGER ist ein binärer Datentyp der weniger Platz beansprucht und schneller verarbeitet werden kann als NUMBER .	Integer
NATURAL	Numerischer Datentyp zur Verwaltung von ganzzahligen positiven Zahlenwerten- positive Integer bzw. dem null Wert. Natural ist PLSQL interner Datentyp.	
BINARY_FLOAT	Numerischer Datentyp zur Verwaltung von Fixkommazahlenwerten. BINARY_FLOAT ist ein PLSQL interner Datentyp. BINARY_FLOAT ist ein binärer Datentyp der weniger Platz beansprucht und schneller verarbeitet werden kann als NUMBER .	Float
BINARY_DOUBLE	Numerischer Datentyp zur Verwaltung von Fixkommazahlenwerten. BINARY_DOUBLE ist ein PLSQL interner Datentyp. BINARY_FLOAT ist ein binärer Datentyp der weniger Platz beansprucht und schneller verarbeitet werden kann als NUMBER .	Double
VARCHAR(255)	Alphanumerischer Datentyp zur Verwaltung von Zeichenketten . Zeichenketten die einer Variable eines Datentyps mit variabler Länge zugeordnet werden nehmen bei der Speicherung genauso viel Platz ein wieviele Zeichen sie beinhalten.	String
CHAR(100)	Alphanumerischer Datentyp zur Verwaltung von Zeichenketten. Char ist ein Datentyp mit festgelegter Länge. Bei einem Datentyp mit festgelgter Länge wird die gespeicherte Zeichenkette solange mit Leerzeichen aufgefüllt bis die Zeichenkette die in der Variablendefinition angegebene Länge hat.	String
VARCHAR(255)	Alphanumerischer Datentyp zur Verwaltung von Zeichenketten. Zeichenketten die einer Variable eines Datentyps mit variabler Länge zugeordnet werden nehmen bei der Speicherung genauso viel Platz ein wieviele Zeichen sie beinhalten.	String
DATE	Datentyp zur Verwaltung von zeitbezogenen Daten	Date
TIMESTAMP	Datentyp zur Verwaltung von zeitbezogenen Daten	Date
BOOLEAN	Datentyp zur Verwaltung von boolschen Wahrheitswerten	Boolean

Abbildung 42. Skalare Datentypen

11.3. Abgeleitete Datentypen

Bei der Definition einer Variable kann alternativ zur Angabe eines Datentyps, der Datentyp einer Variable auch **implizit referenziert** werden.

11.3.1 Data Dictionary

Mit dem `%TYPE` bzw. `%ROWTYPE` Attribut kann der Datentyp für eine Variable implizit referenziert werden.

► Codebeispiel: Ableiten von Datentypen ▾

```

1  -- -----
2  --  Abgeleitete Datentypen
3  --
4  DECLARE
5      -- Implizite Referenzierung
6      L_GIVEN_NAME EMPLOYEES.FIRST_NAME%TYPE;
7      L_SURNAME EMPLOYEES.LAST_NAME%TYPE;
8
9      -- Implizite Referenzierung
10     L_PERSON_REC EMPLOYEES%ROWTYPE;
11
12    BEGIN
13        L_GIVEN_NAME := 'Tobias';
14        L_SURNAME := 'Huber';
15
16        DBMS_OUTPUT.PUT_LINE(L_GIVEN_NAME);
17        DBMS_OUTPUT.PUT_LINE(L_SURNAME);
18
19        SELECT E.*
20        INTO L_PERSON_RES
21        FROM EMPLOYEES E
22        WHERE E.FIRST_NAME = L_GIVEN_NAME AND
23              E.LAST_NAME = L_SURNAME;
24
25        DBMS_OUTPUT.PUT_LINE(
26            'employee data: ,
27            || chr(13)
28            || 'firstname: ,
29            || L_PERSON_REC.FIRST_NAME
30            || chr(13)
31            || 'lastname: ,
32            || L_PERSON_REC.LAST_NAME
33            || chr(13)
34            || 'salary: ,
35            || L_PERSON_REC.SALARY
36            || chr(13)
37        );
38    END;

```

11.4. Strukturierte Datentypen



Strukturierte Datentyp

Strukturierte Datentypen fassen logisch zusammenhängende Variablen zu einem **strukturellen Verbund** zusammen.

11.4.1 Record - Strukturierte Datentypen

Ein Record ist eine Variable mit einem strukturierten Datentyp.

► Erklärung: Komplexe Datentypen ▾

- Records werden zur Verarbeitung von **Datensätzen** verwendet.
- Der Zugriff auf die Felder eines Records erfolgt über den **Punktoperator**.

► Syntax: Record ▾

```

1  --
2  -- Syntax: Strukturierter Datentyp
3  --
4  TYPE <typename> IS RECORD (
5      <fieldname> <type> [NOT NULL {:=} <exp>],
6      ...
7  )
8
9  --
10 -- Datentyp: Strukturierter Datentyp
11 --
12
13 DECLARE
14     -- Typdefinition
15     TYPE L_PERSON_REC_T IS RECORD (
16         FIRST_NAME EMPLOYEES.FIRST_NAME%TYPE,
17         LAST_NAME EMPLOYEES.LAST_NAME%TYPE,
18         ID NUMBER NOT NULL DEFAULT 1,
19         PUBLICATION_DATE DATE
20     );
21
22     -- Record
23     L_PERSON_REC L_PERSON_REC_T;
24
25    BEGIN
26        L_PERSON_REC.ID := 17;
27        ...
28        DBMS_OUTPUT.PUT_LINE (
29            || L_PERSON_REC.FIRST_NAME
30            || L_PERSON_REC.LAST_NAME
31        );
32    END;

```

11.5. Tabellentypen



Tabellentypen ▾

Tabellentypen ermöglichen die Verwaltung von Werten in **Tabellenstrukturen**.

11.5.1 Tabellentypen

Varibalen mit einem Tabellentyp werden als **Virtuelle Tabellen** bezeichnet.

Bevor eine Virtuelle Tabelle definiert werden kann, muss ein entsprechender Tabellentyp definiert werden.

► Syntax: Tabellentypen ▾

```

1  -- -----
2  -- Syntax: Tabellentyp
3  --
4  TYPE <type_name> IS TABLE OF <datatype>
5  --
6  -- Tabellentyp: Basisdatentypen
7  --
8  DECLARE
9  --
10 -- DEFINITION: Tabellentypen
11 TYPE NAME_TABLE_TYPE IS TABLE OF
12   VARCHAR2(10);
13 
14  TYPE CODE_TABLE_TYPE IS TABLE OF
15    NATURAL;
16 
17  -- VARIABLE: Virtuelle Tabelle
18  L_NAMES NAME_TABLE_TYPE :=
19    NAME_TABLE_TYPE(
20      'Franz', 'Josef', 'Daniel'
21    );
22 
23  L_CODES CODE_TABLE_TYPE :=
24    CODE_TABLE_TYPE(12,3,12);
25 
26 BEGIN
27  FOR i IN L_NAMES.FIRST..L_NAMES.LAST
28  LOOP
29    DBMS_OUTPUT.PUT_LINE(
30      L_NAMES(i) || L_CODES(i)
31    );
32  END LOOP;
33 END;

```

11.5.2 Virtuelle Tabellen: Strukturierte Daten

► Codebeispiel: Virtuelle Tabellen ▾

```

1  -- -----
2  -- Tabellendatentypen: Strukturierte D.
3  --
4  DECLARE
5    -- DEFINITION: Tabellentyp
6    TYPE PROJECT_TABLE_TYPE IS TABLE OF
7      HR.L_PROJECTS%ROWTYPE;
8 
9    -- DEKLARATION: Virtuelle Tabelle
10   L_PROJECTS PROJECT_TABLE_TYPE :=
11     PROJECT_TABLE_TYPE();
12 
13   -- DEFINITION: Strukturierter Datentyp
14   TYPE PROJECT_REPORT_TYPE IS RECORD (
15     TITLE HR.L_PROJECTS.TITLE%TYPE,
16     DESCRIPTION  VARCHAR(4000),
17     PROJECT_ID   NUMBER(19,2),
18     FUNDING      NUMBER(10),
19     COMPLEXITY   NUMBER(100)
20   );
21 
22   -- DEFINITION: Tabellentyp
23   TYPE PR_TABLE_TYPE IS TABLE OF
24     PROJECT_REPORT_TYPE;
25 
26   -- DEKLARATION: Virtuelle Tabelle
27   L_PROJECT_REPORTS PR_TABLE_TYPE :=
28     PR_TABLE_TYPE();
29 
30   BEGIN
31     SELECT L.ID, L.FUNDING, L.TITLE
32       L.DESCRIPTION, 1
33     BULK COLLECT INTO L_PROJECT_REPORTS
34     FROM HR.L_PROJECTS L
35     WHERE L.PROJECT_TYPE = 'STIPENDIUM';
36 
37     FOR i IN
38       L_PROJECT_REPORTS.FIRST..L_PROJECT_REPORTS.LAST
39     LOOP
40       DBMS_OUTPUT.PUT_LINE(
41         'project data: '
42         || L_PROJECTS(i).TITLE
43         || chr(13)
44         || L_PROJECTS(i).DESCRIPTION
45         || chr(13)
46       );
47     END LOOP;
48   END;

```

Pseudospalte	Beschreibung	Vorkommen
prior, next	Navigiert zum vorangegangenen oder nachfolgenden Element	
count	Zählt die Einträge	
first, last	Navigiert zum ersten bzw. letzten Element.	
extend	Mit der extend Funktion wird die Größe einer virtuellen Tabelle explizit gesteuert.	

Abbildung 43. Virtuelle Tabellen: Pseudospalten

11.5.3 Virtuelle Tabellen

Mit der **extend** Funktion kann die Größe einer Virtuellen Tabelle explizit gesteuert werden.

► Codebeispiel: Virtuelle Tabellen ▾

```

1  -- -----
2  -- Virtuelle Tabellen
3  --
4  DECLARE
5      -- DEFINITION: Tabellentyp
6      TYPE PROJECT_TABLE_TYPE IS TABLE OF
7          HR.C_PROJECTS%ROWTYPE;
8
9      -- DEKLARATION: Virtuelle Tabelle
10     L_PROJECTS PROJECT_TABLE_TYPE :=
11         PROJECT_TABLE_TYPE();
12
13     P_I PLS_INTEGER DEFAULT 1;
14
15    L_RESEARCH HR.C_RESEARCH_PROJECT%ROWTYPE;
16
17    L_PROJECT HR.C_PROJECTS%ROWTYPE;
18
19    BEGIN
20        FOR PROJECT IN
21            ( SELECT P.* FROM L_PROJECTS)
22        LOOP
23            -- EXPLIZITE AENDERUNG DER GROESSE
24            L_PROJECTS.EXTEND;
25            L_PROJECTS(p_i).TITLE := PROJECT.TITLE;
26            L_PROJECTS(p_i).DESCRIPTION :=
27                PROJECT.DESCRIPTION;
28
29            p_i := p_i + 1;
30        END LOOP;
31
32        DBMS_OUTPUT.PUT_LINE(
33            'project count: ' || l_projects.COUNT
34        );
35    END;

```

11.5.4 DQL: Virtuelle Tabellen

Zur Verarbeitung der Daten einer Virtuellen Tabelle wird die select Anweisung verwendet.

► Codebeispiel: DQL: Virtuellen Tabellen ▾

```

1  -- -----
2  -- DQL: Virtuelle Tabellen
3  --
4  DECLARE
5      -- DEFINITION: Tabellentyp
6      TYPE PROJECT_TABLE_TYPE IS TABLE OF
7          HR.L_PROJECTS%ROWTYPE;
8
9      -- DEKLARATION: Virtuelle Tabelle
10     L_PROJECTS PROJECT_TABLE_TYPE :=
11         PROJECT_TABLE_TYPE();
12
13     TYPE PROJECT_REPORT_TABLE_TYPE IS TABLE
14         OF HR.L_PROJECT_REPORT%ROWTYPE;
15
16     L_PROJECT_REPORTS
17         PROJECT_REPORT_TABLE_TYPE :=
18             PROJECT_REPORT_TABLE_TYPE();
19
20     BEGIN
21         -- Verarbeiten der Daten der Virtuellen
22         -- Tabelle. Speichern des Ergebnisses
23         SELECT L.*
24             BULK COLLECT INTO L_PROJECT_REPORTS
25             FROM HR.PROJECT_REPORT L
26             JOIN TABLE(L_PROJECTS) P
27                 ON L.PROJECT_ID = P.PROJECT_ID
28                 WHERE P.FOUNDING_AMOUNT > 10000;
29
30         DBMS_OUTPUT.PUT_LINE(
31             'project count: ' || L_PROJECTS.COUNT
32         );
33     END;

```

12. PL/SQL - Befehle

03

PLSQL - Befehle

01. SQL Befehle in PL/SQL	108
02. Kontrollstrukturen	111
03. Schleifenkonstrukte	112
04. Fehlerbehandlung	114
05. Cursor	118
06. Transaktionen	??

12.1. SQL Befehle in PLSQL

PL/SQL stellt eine **prozedurale Erweiterung** der SQL Sprachspezifikation dar.

Ein PL/SQL Programm stellt eine freie Abfolge von PL/SQL Anweisungen und SQL Befehlen dar.

12.1.1 DQL - Data Query Language

Die **select** Anweisung wird in PL/SQL Programmen zum Lesen von Daten verwendet.

► Erklärung: select Anweisung ▾

- Gelesene Daten können dabei in der **INTO** Klausel einer Variable zugewiesen werden.
- Besteht das Ergebnis der select Anweisung aus mehreren Datensätzen, wird anstelle der **into** Klausel die **bulk collect into** Klausel verwendet.

► Codebeispiel: select Befehl ▾

```

1  -- -----
2  -- Befehl: DQL Befehle
3  --
4  DECLARE
5      L_TITLE VARCHAR2(50);
6
7      TYPE PROJECT_TABLE_TYPE IS TABLE OF
8          HR.L_PROJECTS%ROWTYPE;
9
10     L_PROJECTS PROJECT_TABLE_TYPE :=
11         PROJECT_TABLE_TYPE();
12
13     BEGIN
14         -- Auslesen eines einzelnen Datensatzes
15         -- Speichern des Ergebnisses in einer
16         -- Variable
17         SELECT P.TITLE INTO L_TITLE
18         FROM HR.L_PROJECTS P
19         WHERE P.PROJECT_ID = 123;
20
21         -- Auslesen mehrerer Datensaetze
22         SELECT L.*
23         BULK COLLECT INTO L_PROJECTS
24         FORM HR.L_PROJECTS L;
25
26         DBMS_OUTPUT.PUT_LINE('select into ...');
27     END;

```

12.1.2 DML - Data Manipulation Language

DML Befehle werden in PL/SQL Programmen zur Verarbeitung von Daten verwendet.

► Codebeispiel: DML Befehle ▾

```

1  -- -----
2  -- Befehl: DML Befehle
3  --
4  DECLARE
5      TYPE PROJECT_TABLE_TYPE IS TABLE OF
6          HR.L_PROJECTS%ROWTYPE;
7
8      TYPE REPORT_TABLE_TYPE IS TABLE OF
9          HR.PROJECT_REPORT%ROWTYPE;
10
11     L_PROJECTS PROJECT_TABLE_TYPE :=
12         PROJECT_TABLE_TYPE();
13
14     L_PROJECT_REPORT
15         HR.PROJECT_REPORT%ROWTYPE;
16
17 BEGIN
18     -- Auslesen der Projektdaten in eine
19     -- virtuelle Tabelle
20     SELECT P.*
21     BULK COLLECT INTO L_PROJECTS
22     FROM HR.L_PROJECTS P;
23
24     -- Loeschen aller Projektreportdaten
25     DELETE FROM HR.PROJECT_REPORT;
26
27     FOR i IN L_PROJECTS.FIRST..L_PROJECTS.LAST
28     LOOP
29         L_PROJECT_REPORT.TITLE :=
30             L_PROJECTS(i).TITLE;
31         L_PROJECT_REPORT.COMPLEXITY := 100;
32
33         INSERT INTO HR.PROJECT_REPORT VALUES
34             L_PROJECT_REPORT;
35     END LOOP;
36
37     -- Aendern der Projektdaten mit einer
38     -- bestimmten ID.
39     UPDATE HR.PROJECT_REPORT
40     SET COMPLEXTY = 200
41     WHERE PROJECT_ID = 23;
42
43 END;
```

12.1.3 FORALL Anweisung

Bei der Ausführung von DML Befehl leitet die PL/SQL Engine die SQL Anweisung an den **SQL Executor** weiter.

► Erklärung: forall Anweisung ▾

- Ein solcher **Kontextswitch** ist immer mit einem bestimmten Resourcenaufwand verbunden.
- Zur Resourcenschonung können mit der **forall** Anweisung mehrere DML Befehle als einzelne PL/SQL Anweisung ausgeführt werden.

► Syntax: forall Anweisung ▾

```

1  -- -----
2  -- Syntax: FORALL
3  --
4  FORALL index IN
5      [ lower_bound .. upper_bound |
6          INDICES OF indexing_collection |
7          VALUES OF indexing_collection
8      ]
9      [ SAVE EXCEPTION ]
10     sql_statement;
11
12 -- -----
13 -- Codebeispiel: FORALL
14 --
15 DECLARE
16     -- DEFINITION: Tabellentyp
17     TYPE PROJECT_REPORT_TYPE IS TABLE OF
18         HR.PROJECT_REPORT%ROWTYPE;
19
20     -- DEKLARATION: Virtuelle Tabelle
21     L_PROJECT_REPORTS PROJECT_REPORT_TYPE :=
22         PROJECT_REPORT_TYPE();
23
24 BEGIN
25     -- Auslesen von Datensaetzen
26     SELECT P.PROJECT_ID, P.TITLE, 0
27     BULK COLLECT INTO L_PROJECT_REPORTS
28     FROM HR.L_PROJECTS P
29
30     -- Schreiben von Datensaetzen in die
31     -- Datenbank
32     FORALL i IN L_PROJECT_REPORTS.FIRST ..
33         L_PROJECT_REPORTS.LAST
34         INSERT INTO HR.PROJECT_REPORT VALUES
35             L_PROJECT_REPORTS(i);
36
37 END;
```

12.1.4 FORALL - RETURNING Klausel

Der Zugriff auf mögliche Rückgabewerte von DML Befehlen erfolgt für **forall** Anweisungen in der **returning** Klausel.

► Codebeispiel: returning Klausel ▾

```

1  -- -----
2  --  Codebeispiel: FORALL RETURNING
3  --
4  DECLARE
5      -- DEFINITION: Tabellentyp
6      TYPE PROJECT_REPORT_TYPE IS TABLE OF
7          HR.PROJECT_REPORT%ROWTYPE;
8
9      -- DEFINITION: Tabellentyp
10     TYPE PROJECT_ID_TYPE IS TABLE OF
11         HR.PROJECT_REPORT.PROJECT_ID%TYPE;
12
13     -- DEKLARATION: Virtuelle Tabelle
14     L_PROJECT_REPORTS PROJECT_REPORT_TYPE :=
15         PROJECT_REPORT_TYPE();
16
17     -- DEKLARATION: Virtuelle Tabelle
18     L_PROJECT_IDS PROJECT_ID_TYPE :=
19         PROJECT_ID_TYPE();
20
21 BEGIN
22     -- Auslesen von Projektdaten
23     SELECT P.PROJECT_ID, P.TITLE, 0
24     BULK COLLECT INTO L_PROJECT_REPORTS
25     FROM HR.L_PROJECTS P;
26
27     -- forall Befehl
28     FORALL i IN L_PROJECT_REPORTS.FIRST..
29         L_PROJECT_REPORTS.LAST
30         INSERT INTO HR.PROJECT_REPORT
31             VALUES L_PROJECT_REPORTS(i)
32
33         -- returning Klausel
34         RETURNING PROJECT_ID
35             BULK COLLECT L_PROJECT_IDS;
36
37         -- Wertausgabe
38         FOR i IN
39             L_PROJECT_IDS.FIRST..L_PROJECT_IDS.LAST
40             LOOP
41                 DBMS_OUTPUT.PUT_LINE(
42                     L_PROJECT_IDS
43                 );
44             END LOOP;
45     END;

```

12.1.5 DDL - Data Definition Language

DDL Befehle werden zum Verwalten der Struktur einer Datenbank verwendet.

DDL Befehle müssen im Kontext der **execute immediate** Anweisung definiert werden.

► Codebeispiel: DDL Befehle ▾

```

1  -- -----
2  --  Befehl: DLL Befehle
3  --
4  DECLARE
5      L_IS_TABLE_PRESENT PLS_INTEGER DEFAULT 0;
6
7      -- Definition eines DDL Befehls in Form
8      -- eines Zeichenkettenliterals
9      L_C_CREATE_TABLE_DDL VARCHAR2(500) :=
10         'CREATE TABLE PROJECT_REPORT (
11             PROJECT_ID INTEGER NOT NULL,
12             TITLE VARCHAR(50) NOT NULL,
13             DESCRIPTION VARCHAR(4000),
14             FUNDING_AMOUNT NUMBER(19),
15             PRIMARY KEY (PROJECT_ID)
16         )';
17
18      -- Definition eines DDL Befehls in Form
19      -- eines Zeichenkettenliterals
20      L_C_DROP_TABLE_DDL CONSTANT VARCHAR2(500)
21          DEFAULT 'DROP TABLE PROJECT_REPORT';
22
23 BEGIN
24     -- Pruefen ob eine bestimmte Tabelle
25     -- bereits existiert.
26     SELECT COUNT(L.TABLE_NAME)
27     INTO L_IS_TABLE_PRESENT
28     FROM USER_TABLES L
29     WHERE L.TABLE_NAME = 'PROJECT_REPORT';
30
31     -- Existiert die Tabelle noch nicht
32     -- wird sie mit der EXECUTE IMMEDIATE
33     -- Anweisung angelegt.
34     IF L_IS_TABLE_PRESENT = 0 THEN
35         EXECUTE IMMEDIATE L_C_CREATE_TABLE_DDL;
36     END IF;
37
38     ...
39
40     -- Die Tabelle wird nach dem Ausfuehren
41     -- des Programms wieder geloescht.
42     EXECUTE IMMEDIATE L_C_DROP_TABLE_DDL;
43 END;

```

12.2. Kontrollstrukturen

Kontrollstrukturen sind Routinen zur Steuerung des Programmflusses.

12.2.1 IF THEN Block

Kontrollstrukturen bestimmen die Reihenfolge, in der die Anweisungen eines Programms ausgeführt werden.

Trifft die Bedingung einer if Anweisung zu, werden die Befehle im **if then** Blocks ausgeführt.

► Syntax: IF THEN ▾

```

1  -- -----
2  -- Syntax: IF THEN
3  --
4  IF condition THEN
5      Statement1;
6      Statement2;
7      ...
8  ELSIF condition2 THEN
9      Statement1;
10     Statement2;
11     ...
12 ELSE
13     StatementN;
14 END IF;
15
16 -- -----
17 -- Beispiel: IF THEN
18 -- -----
19 DECLARE
20     L_EMPLOYEES_COUNT NUMBER := 0;
21     L_NUMBER NUMBER := 8;
22 BEGIN
23     SELECT COUNT(EMPLOYEES_ID)
24     INTO L_EMPLOYEES_COUNT
25     FROM EMPLOYEES;
26
27     IF L_EMPLOYEES_COUNT THEN
28         DBMS_OUTPUT.PUT_LINE(
29             'Inside the if'
30         );
31     END IF;
32
33     DBMS_OUTPUT.PUT_LINE(
34         'Outside the if'
35     );
36 END;

```

12.2.2 Kontrollstruktur - Case Block

Alternativ zur **if** Anweisung steht in PL/SQL Programmen die **case** Anweisung zur Verfügung.

► Syntax: Case Block ▾

```

1  -- -----
2  -- Syntax: CASE Block
3  --
4  CASE
5      WHEN condition1 THEN
6          Statement1;
7          Statement2;
8          ...
9      WHEN condition2 THEN
10         Statement1;
11         ...
12     [ ELSE
13         Statement1;
14         ...
15     ]
16 END CASE;
17
18 -- -----
19 -- Beispiel: CASE Block
20 -- -----
21 DECLARE
22     L_VERSION PLS_INTEGER DEFAULT 12;
23 BEGIN
24     -- read database version
25     SELECT VERSION
26     INTO L_VERSION
27     FROM DMB_SESSION;
28
29     -- case Kontrollstruktur
30     CASE L_VERSION
31         WHEN 11 THEN
32             DBMS_OUTPUT.PUT_LINE(
33                 'no feature support'
34             );
35         WHEN 12 THEN
36             DBMS_OUTPUT.PUT_LINE(
37                 'feature suport'
38             );
39         ELSE
40             DBMS_OUTPUT.PUT_LINE(
41                 'no information'
42             );
43     END CASE;
44
45 END;

```

```

1  -- -----
2  -- Beispiel: CASE Block
3  --
4  DECLARE
5      L_COUNTRY VARCHAR2(30) := &user_input;
6  BEGIN
7      CASE
8          WHEN L_COUNTRY = 'at' THEN
9              DBMS_OUTPUT.PUT_LINE(
10                  'AUSTRIA'
11 );
12         WHEN V_COUNTRY = 'de' THEN
13             DBMS_OUTPUT.PUT_LINE(
14                 'GERMANY'
15 );
16     END CASE;
17     ...
18 END;

```

12.3. Schleifenkonstrukte ▾



Schleifenkonstrukte ▾

Eine Schleife ist eine Programmroutine, die einen bestimmten **Programmabschnitt** solange abarbeitet, bis eine bestimmte Bedingung eintritt.

12.3.1 Loop Block

Der **loop** Block ist die grundlegende Schleifenvariante in PL/SQL. Der Block definiert dabei keine Abbruchbedingung.

► Syntax: Loop Block ▾

```

1  -- -----
2  -- Syntax: Loop Block
3  --
4  LOOP
5      Statement1;
6      ...
7      StatementN;
8  END LOOP;
9
10 --
11 -- Beispiel: Loop Block
12 --
13 DECLARE
14     L_COUNTER NUMBER := 0;
15     L_RESULT NUMBER;
16 BEGIN
17     LOOP
18         IF L_COUNTER > 10 THEN
19             EXIT;
20         END IF;
21
22         L_COUNTER := L_COUNTER + 1;
23         L_RESULT := L_COUNTER * 5;
24
25         RDBMS_OUTPUT.PUT_LINE(
26             '5 * ' || L_COUNTER
27             || ' = '
28             || L_RESULT
29 );
30
31     END LOOP;
32 END;

```

12.3.2 WHILE Block

Mit der **WHILE** Schleifenvariante wird ein bestimmter Programmabschnitt solange wiederholt bis eine bestimmte **Bedingung** eintritt.

► Syntax: WHILE Block ▾

```

1  -- -----
2  -- Syntax: WHILE LOOP
3  --
4  WHILE condition LOOP
5      Statement1;
6      ...
7      StatementN;
8  END LOOP;
9  --
10 --
11 -- Beispiel: WHILE LOOP
12 --
13 DECLARE
14     TYPE PROJECT_TABLE_TYPE IS TABLE OF
15         HR.L_PROJECTS%ROWTYPE;
16
17     L_PROJECTS PROJECT_TABLE_TYPE :=
18         PROJECT_TABLE_TYPE();
19
20     l_i PLS_INTEGER DEFAULT 1;
21
22 BEGIN
23     SELECT L.*
24     BULK COLLECT INTO L_PROJECTS
25     FROM HR.L_PROJECTS L
26     WHERE L.PROJECT_TYPE =
27         'REQUEST_FUNDING_PROJECT';
28
29     WHILE l_i < L_PROJECTS.LAST
30     LOOP
31         DBMS_OUTPUT.PUT_LINE(
32             'project id: '
33             || L_PROJECTS(l_i).PROJECT_ID
34             || CHR(13)
35             || L_PROJECTS(l_i).TITLE
36             || CHR(13)
37             || L_PROJECTS(l_i).DESCRIPTION
38             || CHR(13)
39             || L_PROJECTS(l_i).IS_FWF_FUNDED
40         );
41
42         l_i := l_i + 1;
43     END LOOP;
44 END;

```

12.3.3 FOR Block

► Syntax: FOR Block ▾

```

1  -- -----
2  -- Syntax: FOR LOOP
3  --
4  FOR loop_counter IN [REVERSE]
5      lower_limit .. upper_limit
6  LOOP
7      Statement1;
8      ...
9  END LOOP;
10 --
11 --
12 -- Syntax: FOR SQL LOOP
13 --
14 FOR row IN (sql statement)
15 LOOP
16     Statement1;
17     ...
18 END LOOP;
19 --
20 --
21 -- Beispiel: FOR LOOP
22 --
23 DECLARE
24     TYPE PROJECT_TABLE_TYPE IS TABLE OF
25         HR.L_PROJECTS%ROWTYPE;
26
27     L_PROJECTS PROJECT_TABLE_TYPE :=
28         PROJECT_TABLE_TYPE();
29
30 BEGIN
31     FOR ROW IN (SELECT P.* FROM HR.L_PROJECTS)
32     LOOP
33         DBMS_OUTPUT.PUT_LINE( ROW.PROJECT_ID);
34     END LOOP;
35
36     SELECT L.*
37     BULK COLLECT INTO L_PROJECTS
38     FROM HR.L_PROJECTS;
39
40     FOR i IN l_projects.FIRST..l_projects.LAST
41     LOOP
42         DBMS_OUTPUT.PUT_LINE(
43             L_PROJECTS(i).PROJECT_ID
44         );
45     END LOOP;
46 END;

```

12.4. Fehlerbehandlung

Treten zur **Laufzeit** eines PL/SQL Programms **Fehler** auf, wird das entsprechende Programm bei fehlender Fehlerbehandlung beendet.

12.4.1 Exceptionhandling

Tritt im Ausführungsteil eines PL/SQL Blocks ein **Fehler** auf, verzweigt die Ausführung eines Programms in den Fehlerbehandlungsteil des entsprechenden Blocks.

Die **Fehlerbehandlung** erfolgt in PL/SQL Programmen im Exceptionteil eines Blocks.

► Erklärung: Exceptionhandling ▾

- Für jedes **fehlerbezogene Ereignis**, das in einem PL/SQL Programm auftritt, kann im Exceptionteil eines Blocks, ein **Exceptionhandler** definiert werden.
- Exceptionhandler werden zur **Verarbeitung**, der in einem Programm auftretenden Fehlerzustände definiert.
- **Fehlerzustände** werden in PL/SQL als **Exception**, die Fehlerbehandlung als **Exceptionhandling** bezeichnet.

► Codebeispiel: Exceptionhandling ▾

```

1  -- -----
2  -- Beispiel: Exceptionhandling
3  --
4  DECLARE
5      L_AMOUNT    PLS_INTEGER := 300;
6      L_RELATIVE  PLS_INTEGER := 0;
7      L_RESULT    PLS_INTEGER;
8  BEGIN
9      -- Fehlerzustand: Division by zero
10     L_RESULT := L_AMOUNT/L_RELATIVE;
11
12     -- Programmlogik
13
14     DBMS_OUTPUT.PUT_LINE( L_RESULT );
15 EXCEPTION
16     -- Exceptionhanlder: ZERO_DIVIDE
17     WHEN ZERO_DIVIDE THEN
18     ...
19 END;

```

12.4.2 Exceptionhandler

Mit der Hilfe von Exceptionhandlern kann in PL/SQL Programmen auf eintretende Fehler Bezug genommen werden.

► Erklärung: Exceptionhandler ▾

- Ein Exceptionhandler beginnt mit dem Schlüsselwort **when** und endet mit der letzten Anweisung des zugeordneten **then** Blocks.
- Exceptionhandler werden im **Fehlerbehandlungsteil** eines PL/SQL Blocks definiert. Ein Exceptionteil kann dabei eine freie Abfolge von Exceptionhandlern enthalten.
- Die **Bezeichnung** einer Exception stellt dabei den Bezug zwischen der Exception und einem Exceptionhandler her.
- Wird für einen Fehlerzustand kein Exceptionhanlder definiert, bricht das PL/SQL Programm ohne Fehlerbehandlung ab.

► Codebeispiel: Exceptionhandling ▾

```

1  -- -----
2  -- Beispiel: Exceptionhandler
3  --
4  DECLARE
5      L_AMOUNT    PLS_INTEGER := 300;
6      L_RELATIVE  PLS_INTEGER := 0;
7      L_RESULT    PLS_INTEGER;
8  BEGIN
9      ...
10     L_RESULT := L_AMOUNT/L_RELATIVE;
11
12     ...
13
14 EXCEPTION
15     -- Exceptionhandler: NEGATIVE_BALANCE
16     WHEN NEGATIVE_BALANCE THEN
17         ...
18         -- Exceptionhandler: NO_DATA_FOUND bzw.
19         -- ZERO_DIVIDE
20         WHEN NO_DATA_FOUND OR ZERO_DIVIDE THEN
21             ...
22             -- Exceptionhandler: OTHERS
23             -- Der Exceptionhandler behandelt alle
24             -- Ereignisse fuer die kein Exception-
25             -- handler definiert worden ist.
26             WHEN OTHERS THEN
27                 ...
28 END;

```

Exceptiontyp	Beschreibung	Seite
Standardexception	Standardexceptions sind die durch die PL/SQL Spezifikation definierten internen PL/SQL Exceptions.	114
Anwendungsfehler	Anwendungsfehler sind für Anwendungen spezifische Fehlerzustände.	114
Unbenannte Fehler	Die restlichen Fehler, die keinen vordefinierten Namen besitzen, liefern nur eine Fehlernummer mit zugeordneter Fehlermeldung.	114

Abbildung 44. Typen von Fehlerobjekten

12.4.3 Arten von Exceptions

Die PL/SQL Spezifikation unterscheidet mehrere Formen von Fehlerzuständen.

► Erklärung: Arten von Exceptions ▾

- **Standardexceptions:** Standardexceptions sind Fehlerzustände, die in der PL/SQL Spezifikation definiert werden. Sie beschreiben die grundlegenden in einer Datenbankanwendung auftretenden Fehlerzustände.

Standardexceptions besitzen eine interne Beschreibung und einen Fehlercode.

- **Anwendungsfehler:** Anwendungsfehler sind Fehlerzustände, die explizit für Datenbankanwendungen definiert werden. Sie beschreiben, die für eine Anwendung spezifischen Fehlerzustände.

Anwendungsfehler besitzen eine durch die Anwendung vorgegebene Beschreibung und einen Fehlercode.

- **Unbenannte Fehler:** Unklassifizierte Fehler besitzen in PL/SQL keinen Namen. Unbenannte interne Exceptions müssen in PL/SQL Programmen über die `pragma exception_init` Routine verankert werden.



12.4.4 Anwendungsfehler

Anwendungsfehler sind die durch die **Geschäftslogik** einer Anwendung vorgegebenen Fehlerobjekte.

► Erklärung: Anwendungsfehler ▾

- Bevor ein Anwendungsfehler in einem PL/SQL Programm verwendet werden kann, muß er vorher **deklariert** werden.
- Optional können für ein Fehlerobjekt ein **Fehlercode** und eine **Beschreibung** definiert werden.
- Fehlerzustände werden in PL/SQL Programm über die `raise` Routine ausgelöst.

► Codebeispiel: Applikationsexceptions ▾

```

1  -- -----
2  -- Beispiel: Anwendungsexceptions
3  --
4  DECLARE
5      -- DEKLARATION: Anwendungsexception
6      INCREASE_AMOUNT EXCEPTION;
7
8      -- INITIALISIERUNG: Anwendungsexception
9      pragma exception_init(
10          INCREASE_AMOUNT, -20999
11      );
12 BEGIN
13     -- Anwendungsexception auslösen
14     RAISE INCREASE_AMOUNT;
15     ...
16 EXCEPTION
17     -- EXCEPTIONHANDLING
18     WHEN INCREASE_AMOUNT THEN
19         ...
20 END

```

Exceptionname	Beschreibung	SQL Code
ACCESS_INTO_NULL	Program attempted to assign values to the attributes of an uninitialized object.	ORA-06530
CASE_NOT_FOUND	None of the choices in the <code>WHEN</code> clauses of a <code>case</code> statement were selected ORA-06592 and there is no <code>ELSE</code> clause.	ORA-06592
COLLECTION_IS_NULL	Program attempted to apply collection methods other than <code>EXISTS</code> to an uninitialized nested table or varray, or program attempted to assign values to the elements of an uninitialized nested table or varray.	ORA-06531
CURSOR_ALREADY_OPENED	Program attempted to open an already opened cursor.	ORA-06511
DUP_VAL_ON_INDEX	Program attempted to insert duplicate values in a column that is constrained by a unique index.	ORA-06511
INVALID_CURSOR	There is an illegal cursor operation.	ORA-01001
INVALID_NUMBER	Conversion of character string to number failed.	ORA-01722
NO_DATA_FOUND	Single row <code>SELECT</code> returned no rows or your program referenced a deleted element in a nested table or an uninitialized element in an associative array (index-by table).	100
PROGRAM_ERROR	Plsql has an internal problem.	ORA-06501
ROWTYPE_MISMATCH	The host cursor variable and Plsql cursor variable involved in an assignment have incompatible return types.	ORA-06504
SELF_IS_NULL	A program attempts to call a member method, but the instance of the object type has not been initialized.	5432
STORAGE_ERROR	Plsql ran out of memory or memory was corrupted.	ORA-06500
SUBSCRIPT_BEYOND_COUNT	A program referenced a nested table or varray using an index number larger than the number of elements in the collection.	ORA-06533
SUBSCRIPT_OUTSIDE_LIMIT	A program referenced a nested table or varray element using an index number that is outside the legal range (for example, -1).	ORA-06532
SYS_INVALID_ROWID	The conversion of a character string into a universal rowid failed because the character string does not represent a ROWID value.	ORA-01410
TOO_MANY_ROWS	Single row <code>SELECT</code> returned multiple rows.	ORA-01422
TRANSACTION_BACKED_OUT	The remote portion of a transaction has rolled back.	ORA-00061
VALUE_ERROR	An arithmetic, conversion, truncation, or size constraint error occurred.	ORA-06502
ZERO_DIVIDE	A program attempted to divide a number by zero.	ORA-01476

Abbildung 45. Standardexceptions

12.4.5 Unbenannte Fehler

Unbenannte Fehler sind systeminterne Fehler ohne vordefinierten Namen.

► Erklärung: Unbenannte Fehler ▾

- Fehlerereignisse, die in PL/SQL Programmen nur selten auftreten, wird kein eigener Name zugewiesen.
- Fehler dieser Art werden im `others` Exceptionhandler behandelt.

► Codebeispiel: Unbenannte Fehler ▾

```

1  -- -----
2  -- Beispiel: Unbenannte Fehler
3  --
4  DECLARE
5      -- TYPDEFINITION
6      TYPE PR_TABLE_TPYE IS TABLE OF
7          HR.PROJECT_REPORT%ROWTYPE;
8
9      -- DEKLARATION
10     L_REPORTS PR_TABLE_TYPE := 
11         PR_TABLE_TYPE();
12
13     ERRTEXT VARCHAR2(300);
14
15     BEGIN
16         SELECT L.PROJECT_ID, L.TITLE, 201
17         BULK COLLECT INTO L_REPORTS
18         FROM HR.L_PROJECTS L;
19
20         FORALL i IN
21             L_REPORTS.FIRST..L_REPORTS.LAST
22             INSERT INTO HR.PROJECT_REPORT VALUES
23                 L_REPORTS(i);
24
25     EXCEPTION
26         -- EXCEPTION HANDLER OTHERS: Unbenannte
27         -- Fehler werden im others Exception
28         -- Handler behandelt. Um die Unbenannten
29         -- Fehler unterscheiden zu koennen wird
30         -- das sqlcode Attribut gegen die
31         -- Nummer des Fehlers geprueft.
32         WHEN OTHERS THEN
33             ERRTEXT := SQLERRM(SQLCODE);
34
35             IF SQLCODE = -20981 THEN
36                 DBMS_OUTPUT.PUT_LINE(
37                     'error orccured: ' || errtext
38                 );
39             END IF;
40
41     END

```

12.4.6 Fehlerbehandlung für forall

Die forall Anweisung erlaubt es mehrere DML Anweisungen gekapselt auszuführen.

► Erklärung: Fehlerbehandlung für forall ▾

- Tritt ein Fehler bei der Ausführung bei einer der DML Anweisungen auf, werden die restlichen Anweisungen trotzdem ausgeführt.
- Die `save exceptions` Klausel aggregiert Fehlernachrichten die bei der Ausführung des forall Befehls auftreten.

► Codebeispiel: Fehlerbehandlung für forall ▾

```

1  -- -----
2  -- Beispiel: Fehlerbehandlung fr forall
3  --
4  DECLARE
5      TYPE PR_TABLE_TYPE IS TABLE OF
6          HR.PROJECT_REPORT%ROWTYPE;
7
8      L_REPORTS PR_TABLE_TYPE := PR_TABLE_TYPE();
9
10     BEGIN
11         SELECT L.PROJECT_ID, L.TITLE, 201
12         BULK COLLECT INTO L_REPORTS
13         FROM HR.L_PROJECTS L;
14
15         FORALL i IN L_REPORTS.FIRST..L_REPORTS.LAST
16             SAVE EXCEPTIONS
17             INSERT INTO HR.PROJECT_REPORT VALUES
18                 L_REPORTS(i);
19
20     EXCEPTION
21         WHEN BULK_ERRORS
22             THEN
23                 DBMS_OUTPUT.PUT_LINE(
24                     'UPDATED ' || SQL%ROWCOUNT || ' rows.'
25                 );
26
27         FOR i IN 1..SQL%BUKL_EXCEPTINS.COUNT
28             LOOP
29                 DBMS_OUTPUT.PUT_LINE(
30                     'error ' || i || ' occurred '
31                     SQL%BUKL_EXCEPTIONS(i).ERROR_INDEX
32                 );
33
34             DBMS_OUTPUT.PUT_LINE(
35                 SQLERRM(-1*
36                     SQL%BUKL_EXCEPTIONS(i).ERROR_CODE)
37                 );
38         END LOOP;
39     END

```

12.5. Kursor



Kontextarea ▾

Als Kontextarea wird jener Bereich des **Arbeitspeichers** bezeichnet, der für die Ausführung von SQL Anweisungen reserviert ist.

Bevor die **Daten** einer Datenbank **verarbeitet** werden können, müssen Sie in die Kontextarea der **PL/SQL Engine** geladen werden.

12.5.1 Kursor

Im vereinfachten Sinne stellt ein Kursor eine Referenz auf die Daten der Kontextarea dar.

► Erklärung: Kursor ▾

- Bei Kursorn handelt es sich um benannte **Datenbankobjekte**.
- Kursor sind in PL/SQL Programmen instrumental für den **Zugriff** auf die **Daten** einer Datenbank. Ein Kursor verwaltet dabei die Datenstrukturen, in denen der Datenbankserver seine Daten bewegt.
- Immer dann wenn eine **select** oder DML Anweisung ausgeführt wird, erzeugt die PL/SQL Engine einen Cursor, um die Ergebnismenge zu bestimmen bzw. Daten zu speichern.
- Für Kursor werden dabei 3 **Formen** unterschieden: implizite und explizite Kursor.
- **implizite Kursor:** Implizite Kursor werden bei der Ausführung von SQL Befehlen von der PL/SQL Engine erzeugt und verwaltet.
- **explizite Kursor:** Explizite Kursor ermöglichen PL/SQL Programmen den Zugriff auf die Daten einer SQL Abfrage. Die Steuerung eines expliziten Kursors obliegt dabei zur Gänze dem PL/SQL Programm.
- **cursor for Schleife:** Mit der **cursor for** Schleife bietet PL/SQL eine Möglichkeit auf einfache Weise auf die Kontextarea einer select Anweisung zu zugreifen.
- Die Anzahl gleichzeitig geöffneter Kursor wird dabei nicht durch die Datenbank beschränkt.
- Über den **OPEN_CURSORS** Parameter kann die Anzahl der gleichzeitig geöffneten Kursor einer Datenbank gesteuert werden.

12.5.2 Explizite Cursor

Explizite Kursor ermöglichen PL/SQL Programmen den Zugriff auf die Daten einer SQL Abfrage. Die Verwaltung des Kursors obliegt dabei zur Gänze dem PL/SQL Programm.

Im Zuge der **Verarbeitung** der Daten durchläuft der **Kursor** 4 Phasen.

► Auflistung: Kursorzugriff ▾

- **Kursordeklaration:** Bevor ein Kursor verwendet werden kann muss er **deklariert** werden.
- **Kursorinitialisierung:** Mit der **Initialisierung** eines Kursors werden die mit dem Kursor assoziierten Daten aus der Datenbank in die Kontextarea geladen.
- **Datenzugriff:** Der Zugriff auf die Daten der Kontextarea erfolgt für einen Kursor Zeile für Zeile.
- **Resourcenfreigabe:** Wurden die mit dem Kursor assoziierten Daten verarbeitet, können die vom Kursor verwalteten **Ressourcen** wieder **freigegeben** werden.

► Codebeispiel: Kursorzugriff ▾

```

1  -- -----
2  -- Codebeispiel: expliziter Cursor
3  --
4  DECLARE
5      -- PHASE 1: cursor declaration
6      CURSOR L_EMPLOYEES_CUR IS
7          SELECT E.* FROM EMPLOYEES E;
8
9      L_EMPLOYEE_REC L_EMPLOYEES_CUR%ROWTYPE;
10 BEGIN
11     -- PHASE 2: cursor initialisierung
12     OPEN L_EMPLOYEES_CUR;
13
14 LOOP
15     -- PHASE 3: data access
16     FETCH L_EMPLOYEE_CUR INTO
17         L_EMPLOYEE_REC;
18     EXIT WHEN L_EMPLOYEE_CUR%NOTFOUND;
19
20     DBMS_OUTPUT.PUT_LINE(
21         L_EMPLOYEE_REC.EMPLOYEE_ID
22     );
23
24 END LOOP;
25
26 -- PHASE 4: release cursor ressources
27 CLOSE CURSOR_NAME;
28 END;

```

Kursorattribut	Beschreibung	Rückgabewert
FOUND	Das Attribut speichert den Wert true wenn der vorhergehende fetch Befehl ein Ergebnis zurückgeliefert hat. Anderfalls wird false gespeichert	Boolean
NOTFOUND	Das Gegenteil von found	Boolean
ISOPEN	Mit dem Attribut wird geprüft ob der Cursor offen ist.	Boolean
ROWCOUNT	Gibt die Zahl der Zeilen an die durch die Datenbankoperation bearbeitet worden sind.	Number
BULK_ROWCOUNT	Gibt die Zahl der Zeilen an die durch die Datenbankoperation bearbeitet worden sind.	Number
BULK_EXCEPTION	Gibt an ob bei der Massendatenverarbeitung Fehler aufgetreten sind.	Boolean

Abbildung 46. Kursorattribute

12.5.3 Datenverarbeitung

Ein Kursor ermöglicht PL/SQL Programmen den Zugriff auf die Daten einer Datenbank.

► Erklärung: Datenverarbeitung ▾

- Mit der Initialisierung eines Kursors wird das Ergebnis der mit dem Kursor assoziierten SQL Abfrage in die **Kontextarea** der PL/SQL Engine geladen. Zwischenzeitliche Änderungen am Datenbestand werden jedoch nicht an die Daten der Kontextarea weitergegeben.
- Nach seiner **Initialisierung** referenziert ein Kursor den ersten Datensatz der Ergebnismenge der assoziierten Abfrage.
- Der Zugriff auf die Daten der Kontextarea erfolgt im Kursor Zeile für Zeile. Das Lesen eines Datensatzes aus dem Kursor wird dabei als **Fetch** bezeichnet.
- Für den Datenzugriff verwaltet die PL/SQL Engine eine Reihe von Statusinformationen. Über **Kursorattribute** können diese Daten in PL/SQL Programm abfragt werden.



12.5.4 BULK FETCH Anweisung

Im Zuge eines Fetch können in den Kursor auch mehrere Datensätze geladen werden.

► Codebeispiel: bulk collect ▾

```

1  -- -----
2  -- Codebeispiel: Bulk Collect
3  --
4  DECLARE
5      TYPE EMPLOYEE_TABLE_TYPE IS TABLE OF
6          HR.EMPLOYEES%ROWTYPE;
7
8      CURSOR L_EMPLOYEES_CUR IS
9          SELECT * FROM EMPLOYEES;
10
11     L_EMPLOYEES EMPLOYEE_TABLE_TYPE := 
12         EMPLOYEE_TABLE_TYPE();
13
14     BEGIN
15         OPEN L_EMPLOYEES_CUR;
16
17         LOOP
18             -- DATEN EINLESEN: 100 Datensaetze
19             FETCH L_EMPLOYEES_CUR BULK COLLECT
20                 INTO L_EMPLOYEES LIMIT 100;
21
22             ...
23         END LOOP;
24
25         CLOSE L_EMPLOYEES_CUR;
26     END;

```

13. PL/SQL - Blocktypen

04

PLSQL Blöcke

01. Blocktyp: Stored Prozedur	120
02. Blocktyp: Package	123
03. Blocktyp: Trigger	125
04. Namenskonventionen	125

13.1. Blocktyp - Stored Prozedure ▾

Eine Prozedur beschreibt eine freie Abfolge von Anweisungen und Befehlen.

Prozeduren werden zur **Organisation** von PL/SQL Code in PL/SQL Programmen verwendet.

13.1.1 PL/SQL Prozedur

Eine Prozedur ist ein benannter PL/SQL Block, an den Parameter übergeben werden können.

► Erklärung: PL/SQL Prozedur ▾

- Für Prozeduren wird dabei zwischen der **Spezifikation** und der **Implementierung** unterschieden. Der Prozedurkopf beschreibt die Spezifikation der Prozedur, der Prozedurkörper die Implementierung.

► Syntax: PL/SQL Prozedur ▾

```

1  -- -----
2  -- Syntax: Pl/SQL Prozedur
3  --
4  CREATE [OR REPLACE] PROCEDURE <name>
5      [(Parameter1, [Parameter2, ... ])]
6  IS
7      ...
8  BEGIN
9      Executable statements;
10 END [<name>];
11
12 -- -----
13 -- Codebeispiel: Pl/SQL Prozedur
14 --
15 -- SPEZIFIKATION: update_project
16 CREATE OR REPLACE PROCEDURE UPDATE_PROJECT(
17     P_PROJECT_TITLE IN VARCHAR2,
18     P_PROJECT_ID IN PLS_INTEGER,
19 )
20 -- IMPLEMENTIERUNG: update_project
21 AS
22     L_PROJECT_DESC VARCHAR2(4000);
23 BEGIN
24     UPDATE PROJECTS
25     SET PROJECT_TITLE = P_PROJECT_TITLE,
26         WHERE PROJECT_ID = P_PROJECT_ID;
27 END UPDATE_PROJECT;

```



Blocktyp	Beschreibung	Seite
Prozedur	Eine Prozedur beschreibt eine freie Abfolge von Anweisungen und Routinen. Prozeduren werden zur Organisation von PL/SQL Code in PL/SQL Programmen verwendet.	120
Funktion	PL/SQL Funktionen sind Prozeduren die einen Rückgabewert an den Aufrufer der Funktion zurückgeben.	122
Package	Ein Package faßt logisch zusammenhängende Objekte zu einer modularen Einheit zusammen. Packages werden zur Organisation und Strukturierung von PL/SQL Programmen verwendet.	123
Trigger	Trigger sind eine besondere Form von Prozeduren, die in Reaktion auf ein bestimmtes datenbankinternes Ereignis ausgelöst werden. Trigger werden in Datenbanken zur Wahrung der Datenkonsistenz eingesetzt.	125

Abbildung 47. Kursorattribute

13.1.2 Unterprogrammaufruf

Der Aufruf einer Prozedur erfolgt durch die Angabe des **Prozedurnamen** mit den jeweiligen Parameterwerten.

► Erklärung: **Unterprogrammaufruf** ▾

- Eine wesentliche Eigenschaft von Prozeduren ist die Möglichkeit sie in **kompilierter Form** in einer Datenbank zu speichern. Prozeduren liegen damit im zentralen **Namensraum** einer Datenbank und können aus anderen Blöcken heraus aufgerufen werden.
- Da eine Prozedur kein Ergebnis berechnet, kann sie nicht unmittelbar in einem Ausdruck auftreten. Der Unterprogrammaufruf erfolgt deshalb in Form einer **ausführbaren Anweisung**.
- Hat eine Prozedur keine Parameter, so kann die Parameterliste auch fehlen. In diesem Fall erfolgt der **Aufruf** der Prozedur ohne Angabe von Klammern.
- Bei einem Prozederaufruf, wird die Programmausführung unterbrochen und die **Programmkontrolle** an die Prozedur übergeben.
- Lorem ipsum ipsum bla bla bla.

13.1.3 Parametertyp

Im einfachsten Fall wird ein Parameter durch einen Namen und eine Datentyp definiert.

Zusätzlich zum Namen und Datentyp kann Parametern ein Typ zugeordnet.

► Erklärung: **Parametertyp** ▾

- **Parametertypen** beschreiben wie sich Parameter in Unterprogrammen verhalten.
- **Parametertyp in:** in Parameter verhalten sich in Unterprogrammen wie **initialisierte Konstante**. Im Unterprogramm wird der Parameter wie ein konstanter Wert behandelt, der zwar gelesen, dessen Inhalt aber nicht geändert werden kann.
- **Parametertyp out:** out Parameter verhalten sich in Unterprogrammen wie **nicht initialisierte Variablen**. Der Wert von out Parametern kann im Unterprogramm verändert aber nicht gelesen werden. Typischerweise werden out Parameter zur Rückgabe von berechneten Werten verwendet.
- **Parametertyp in out:** Der Parametertyp vereint die Eigenschaften der anderen Parametertypen. in out Parameter können in Unterprogrammen gelesen und geändert werden.



Parameterzuweisung	Beschreibung	Seite
positionell	Bei der positionellen Parameterzuweisung werden die Werte den Parametern in der selben Reihenfolge zugeordnet, in der sie im Unterprogramm definiert wurden.	122
nominell	Bei der nominellen Parameterübergabe werden die Parameterwerte über den Parameternamen zugeordnet.	122

Abbildung 48. Formen der Parameterzuweisung

13.1.4 Formen der Parameterzuweisung

Parameter können in PL/SQL Programmen auf unterschiedliche Weise zugewiesen werden: nominell bzw. positionell.

► Erklärung: nominelle Parameterübergabe ▾

- Werden die **Parameter** einer Prozedur nominell übergeben, muß beim Aufruf der Prozedur keine Rücksicht auf die **Reihenfolge** der Parameter mehr genommen werden.
- Der **Assoziationsoperator =>** stellt den Parameternamen zum Aufrufzeitpunkt mit dem Parameterwert in Beziehung.

► Codebeispiel: Parameterübergabe ▾

```

1 -- -----
2 -- Codebeispiel: Parameteruebergabe
3 --
4 CREATE OR REPLACE PROCEDURE PRINT_DATE (
5     P_START_DATE IN DATE
6 ) AS
7     L_END_TIME VARCHAR2(25);
8 BEGIN
9     L_END_TIME := TO_CHAR(
10         NEXT_DAY(
11             P_START_DATE, 'MON'
12         ), 'DD.MM.YYYY'
13     );
14     ...
15 END PRINT_RETURN_DATE;
16
17
18 -- Procedure call
19 CALL PRINT_RETURN_DATE(
20     P_START_DATE => SYSDATE
21 );

```

13.1.5 Optionale Parameterzuweisung

Bei seiner Definition kann ein Parameter optional mit einem **Vorgabewert** belegt werden.

► Codebeispiel: Parameterübergabe ▾

```

1 -- -----
2 -- Optionale Parameteruebergabe
3 --
4 CREATE OR REPLACE PROCEDURE PRINT_DATE (
5     P_START_DATE IN DATE := SYSDATE,
6     P_DAY_AMOUNT IN NUMBER := 24
7 );
8
9 CALL PRINT_RETURN_DATE(
10     P_START_DATE => TO_DATE()
11 );

```

13.1.6 PL/SQL Funktion

Eine Funktion ist ein Unterprogramm das eine **Rückgabewert** an den Aufrufer der Funktion zurückgibt.

► Syntax: PL/SQL Funktionen ▾

```

1 -- -----
2 -- Syntax: Funktion
3 --
4 CREATE [OR REPLACE] FUNCTION <name>
5     [(Parameter1, Parameter2, ...)]
6     RETURN datatype IS
7     declare variable, constant, etc. here
8     BEGIN
9         executable statements;
10        RETURN (Return Value);
11    END [<name>];

```

Blocktyp	Beschreibung	Seite
Packagespezifikation	Die Pacakgespezifikation beschreibt die öffentliche Schnittstelle eines PL/SQL Packages.	123
Packageimplementierung	Die Packageimplementierung enthält die Implementierung der in der Spezifikation deklarierten Objekte.	124

Abbildung 49. Packagekomponenten

13.2. Blocktyp - Package



PL/SQL Package

Ein Package faßt logisch zusammenhängende Objekte zu einer **modularen Einheit** zusammen.

PL/SQL Packages werden zur **Organisation** und **Strukturierung** von PL/SQL Programmen verwendet.

13.2.1 PL/SQL Package

Ein Package faßt Objekte wie Funktionen, Prozeduren, Variablen bzw. Typdeklarationen zu einem **Datenbankobjekt** zusammen.

Erklärung: PL/SQL Package

- Packages werden in **kompilierter Form** in einer Datenbank gespeichert. Damit liegen Funktionen und Prozeduren nicht mehr verstreut in einer Datenbank, sondern stehen **logisch strukturiert** in einem Objekt zur Verfügung.
- Packages werden beim **Aufruf** eines Packageobjekts komplett in den Arbeitsspeicher geladen und automatisch nachkompiliert. Bei weiteren Aufrufen kann das Objekt direkt verwendet werden.
- PL/SQL Packages werden in 2 Schritten definiert: **Packagespezifikation** und **Packageimplementierung**.
- **Packagespezifikation:** Die Packagespezifikation beschreibt die **öffentliche Schnittstelle** eines Packages.
- **Packageimplementierung:** Die Packageimplementierung enthält die Implementierung der in der Spezifikation deklarierten Objekte.

13.2.2 Packagespezifikation

Die Packagespezifikation beschreibt die **öffentliche Schnittstelle** eines Packages.

Erklärung: Packagespezifikation

- In der Packagespezifikation werden alle Objekte **deklariert** die nach außen hin sichtbar sein sollen.
- Mit der Deklaration werden die Objekte in den **globalen Kontext** einer Datenbank geladen.

Syntax: Package Spezifikation

```

1 -- -----
2 -- Syntax: Package Spezifikation
3 --
4 CREATE [OR REPLACE] PACKAGE pkg_name
5 AS
6     Deklaration der Package Elemente
7 END [pkg_name];
8 --
9 --
10 -- Codebeispiel: Package Spezifikation
11 --
12 CREATE OR REPLACE PACKAGE TOOLS
13 AS
14     -- DEKLARATION: Konstanten und Variablen
15     GC_MATH_PI CONSTANT NUMBER := 3.1456;
16 
17     TYPE PROJECT_TABLE_TYPE IS TABLE OF
18         HR.PROJECTS%ROWTYPE;
19 
20     PROCEDURE PRINT_DATE (
21         P_START_DATE IN DATE := SYSDATE,
22         P_DAY IN NUMBER := 24
23     );
24 
25 END TOOLS;

```

Packageartefakt	Packagekomponente	Beschreibung	Seite
globale Variable/- Konstante	Spezifikation	Globale Variablen sind Variablen und Konstanten die sich im globalen Kontext einer Datenbank befinden.	100
Prozedurspezifikation	Spezifikation	Mit einer Prozedurspezifikation befindet sich eine Prozedur im globalen Namensraum einer Datenbank.	120
Funktionsspezifikation	Spezifikation	Mit einer Funktionsspezifikation befindet sich eine Funktion im globalen Namensraum einer Datenbank.	122
globale Prozedur	Implementierung	Globale Prozeduren sind Prozeduren die sich im globalen Kontext einer Datenbank befinden.	120
globale Funktion	Implementierung	Globale Funktion sind Funktion die sich im globalen Kontext einer Datenbank befinden.	122
lokale Prozedur	Implementierung	Lokale Prozeduren sind nur im Kontext eines Packages aufrufbar.	120

Abbildung 50. Packageartefakte

13.2.3 Packageimplementierung

Der Packagekörper enthält die **Implementierung** der in der Spezifikation deklarierten Objekte.

► Erklärung: Packageimplementierung ▾

- Packagespezifikation und Packageimplementierung werden über den **Packagenamen** in Bezug gesetzt.
- Die Packageimplementierung kann **lokal** auch eigene Objekte deklarieren. Diese Objekte können jedoch nur innerhalb der Packageimplementierung referenziert werden.

► Syntax: Packageimplementierung ▾

```

1  -- -----
2  -- Syntax: Package Koerper
3  --
4  CREATE OR REPLACE PACKAGE BODY pkg_name
5  AS
6      -- lokale Variablen und Konstanten
7      ...
8      -- lokale und globale Prozeduren
9      -- und Funktionen
10     ...
11 BEGIN
12     -- Initialisierungsteil
13 END [pkg_name];
14
15  -- -----
16  -- Codebeispiel: Packagekoerper
17  --
18  CREATE OR REPLACE PACKAGE BODY TOOLS
19  AS
20      -- IMPLEMENTIERUNG: globale Prozedur
21      PROCEDURE PRINT_DATE (
22          P_START_DATE IN DATE
23      ) AS
24          L_END_TIME VARCHAR2(25);
25      BEGIN
26          L_END_TIME := TO_CHAR(
27              NEXT_DAY(
28                  P_START_DATE, 'MON'
29              ), 'DD.MM.YYYY'
30          );
31      END PRINT_DATE;
32
33  -- DEKLARATION: lokale Prozedur
34  PROCEDURE INITIALIZE AS
35  BEGIN
36      G_STD_DAY_AMOUNT := 24;
37  END INITIALIZE;
38
39  BEGIN
40      INITIALIZE;
41  END TOOLS;
42
43  -- -----
44  
```



Triggerform	Beschreibung	Seite
Zeiletrigger	Ein Zeiletrigger wird für jeden geänderten Datensatz ausgelöst. Zeiletrigger werden eingesetzt, wenn die auszuführenden Aktionen vom Inhalt der einzelnen Datensätze abhängen.	125
Anweisungstrigger	Im Gegensatz dazu reagiert ein Anweisungstrigger unabhängig von der Anzahl der geänderten Datensätze, auf die durchgeführte DML Anweisung. Beispielsweise kann ein Anweisungstrigger verwendet werden, um die Berechtigung eines Benutzers für die auszuführenden Operation zu überprüfen.	125

Abbildung 51. Triggerformen

13.3. Blocktyp: Trigger

Trigger ▾
Trigger sind eine besondere Form von **Prozeduren**, die in Reaktion auf ein bestimmtes datenbankinternes **Ereignis** ausgelöst werden.

Trigger werden in der Datenbank zur Wahrung der **Datenkonsistenz** eingesetzt.

13.3.1 PL/SQL Trigger

Trigger sind Prozeduren die automatisch in Reaktion auf ein bestimmtes Ereignis ausgelöst werden.

► Erklärung: PL/SQL Trigger ▾

- Trigger werden in **kompilierter Form** in Datenbanken gespeichert. Wie Indizes oder Constraints sind Trigger von den Tabellen abhängig für die sie definiert wurden.
- Die Aktivierung eines Triggers erfolgt dabei implizit durch die Datenbank, wahlweise vor oder nach dem Auftreten eines bestimmten **Ereignisses**.
- In Datenbanken werden 2 **Formen von Triggern** unterschieden: Zeiletrigger und Anweisungstrigger.
- **Zeiletrigger:** Ein Zeiletrigger wird für jeden geänderten **Datensatz** ausgelöst. Zeiletrigger werden eingesetzt, wenn die auszuführenden Aktionen vom Inhalt der einzelnen Datensätze abhängen.
- **Anweisungstrigger:** Im Gegensatz dazu reagiert ein Anweisungstrigger unabhängig von der Anzahl der geänderten Datensätze, auf die durchgeführte **DML Anweisung**.

13.3.2 Anlegen von Triggern

Trigger werden in **kompilierter Form** in Datenbanken gespeichert. Zum Anlegen eines Triggers wird der `create` Befehl verwendet.

► Erklärung: Anlegen von Triggern ▾

- Im **Triggerkopf** werden die Eigenschaften eines Triggers definiert.
- Der Kopf enthält den Triggernamen, den Zeitpunkt der Aktivierung, den Triggertyp und die auslösenden Ereignisse.
- Ein Trigger kann mehrere Ereignisse gleichzeitig behandeln.

► Syntax: Anlegen von Triggern ▾

```

1   -- -----
2   -- Syntax: Anlegen von Triggern
3   --
4   CREATE [OR REPLACE] TRIGGER <trigger_name>
5       -- Triggerereignis festlegen
6       [BEFORE|AFTER] <event> ON <table_name>
7       -- Triggerart definieren
8       [FOR EACH ROW]
9       -- Ketten von Triggern definieren
10      [FOLLOWS another_trigger_name]
11      -- Trigger verwalten
12      [ENABLE/DISABLE]
13      -- Konditionale Ausfuehrung
14      [WHEN condition]
15      DECLARE
16      BEGIN
17          DBMS_OUTPUT.PUT_LINE('Trigger example');
18          ...
19      END;

```

Triggerform	Beschreibung	Seite
SQL Befehle	Trigger können im Rahmen ihrer Logik SQL Befehle ausführen. Allerdings existieren aufgrund der Datenkonsistenzregeln gewisse Einschränkungen bezüglich der Tabellen, die gelesen und verändert werden dürfen. Diese Tabellen werden als mutating tables bezeichnet. Für die Daten in Mutating Tables dürfen keine SQL Befehle ausgeführt werden. Die mutating table ist vereinfacht ausgedrückt immer die Tabelle, die einen Trigger auslöst. Handelt es sich um kaskadierende Trigger, existieren mehrere mutating tables.	125
Transaktionssteuerung	Innerhalb eines triggers dürfen keine Befehle zur Transaktionssteuerung verwendet werden, da die Ausführung eines Triggers innerhalb der Transaktionsklammer der auslösenden Tabelle erfolgt. Dies gewährleistet das Trigger die aktuelle Transaktion nicht beenden oder beeinflussen können.	125
long Datentyp	Der long Datentyp darf nicht als VariablenTyp im Deklarationsteil eines Triggers verwendet werden. Attributwerte, die diese Datentypen besitzen, müssen entweder konvertiert werden oder können in SQL Befehlen tatsächlich nicht verwendet werden.	125

Abbildung 52. Triggerprogrammierung

13.3.3 Pseudorecords

Pseudorecord ▾

Trigger haben für geänderte Datensätze Zugriff auf die ursprünglichen bzw. neuen Werte der adaptierten Attribute.

► Erklärung: Pseudorecord ▾

- Der Kontext eines Triggers kapselt jene Daten, die durch die auslösende DML Anweisung verändert wurden.
- Das Präfix :old referenziert dabei den geänderten und das Präfix :new den aktuellen Attributwert. Der Doppelpunkt dient als Separator zwischen Präfix und Attributnamen.
- Zur Referenzierung des auslösenden DML Ereignisses kann in einem Trigger auf die `inserting`, `updating` bzw. `deleting` Bedingungen zugegriffen werden.
- Wurde ein Trigger beispielsweise durch eine `update` Operation ausgelöst, evaluiert die `updating` Bedingung zu true.

► Codebeispiel: Pseudorecords ▾

```

1  -- -----
2  -- Codebeispiel: Log Beispiel
3  --
4  CREATE OR REPLACE TRIGGER tr_project_audit
5    AFTER INSERT OR UPDATE OR DELETE ON
       projects
6    FOR EACH ROW ENABLE
7    DECLARE
8    BEGIN
9      IF INSERTING THEN
10        INSERT INTO PROJECTS_AUDIT VALUES (
11          :NEW.TITLE, NULL,
12          USER, SYSDATE, 'INSERTING'
13        );
14      ELSIF UPDATING THEN
15        INSERT INTO PROJECTS_AUDIT VALUES (
16          :NEW.TITLE, :OLD.TITLE,
17          USER, SYSDATE, 'UPDATING'
18        );
19      ELSIF DELETING THEN
20        INSERT INTO PROJECTS_AUDIT VALUES (
21          NULL, :OLD.TITLE,
22          USER, SYSDATE, 'DELETING'
23        );
24      END IF;
25    END;

```

13.3.4 Namenskonventionen

Datenbankobjekt	Namenskonvention	Beispiel	Seite
Primary Key	<name>_pk	PROJECT_ID, SUBPROJECT_ID, FUNDING_ID	76
Unique Key	<name>_uk	PROJECT_TITLE_UK	76
Foreign Key	<name>_pk	SUBPROJECT_PROJECT_FK	76
Check Constraint	<name>_chk	PROJECTS_FUNDING_CHK, PROJECT_RESEARCH_CHK	76
Sequence	<name>_seq	PROJECTS_PROJECT_ID_SEQ, PROJECTS_SUBPROJECT_ID_SEQ	82
View	<name>_v	PROJECT_FUNDING_V, PROJECT_RESEARCH_V	78
Type	<name>_t	PROJECT_RECORD_T, SUBPROJECT_RECORD_T	105
PL/SQL Package	<name>_pkg	PROJECT_MIGRATION_PKG, LOGGING_PKG	123
PL/SQL Procedure	<name>_prc	RELEASE_RESOURCE_PRC	120
PL/SQL Function	<name>_fun	MIGRATE_PROJECT_DATA_FUN	120
Global Variable	g_<variable_name>	G_PROJECT_DATA	105
Local Variable	l_<variable_name>	L_INDEX	105
Parameter	p_<name>	P_PROJECT	105
Cursor	c_<name>	LC_PROJECTS, LC_SUBPROJECTS	105
Varchar	v_<name>	LV_FIRST_NAME, LV_LAST_NAME	105
Record	r_<name>	LR_PROJECT	105
Datentabelle	<name>	PROJECTS, SUBPROJECTS, EMPLOYEES, DEPARTMENTS	15
Revisionstabelle	v_<name>	V_PROJECTS, V_SUBPROJECTS, V_EMPLOYEES	15
Attributabelle	e_<name>	E_PROJECT_TYPE, E_FUNDING_TYPE	15
Schlüsseltabelle	<name>_jt	PROJECT_EMPLOYEES_JT, PROJECT_FUNDING_JT	15
Sammeltabelle	<name>_st	PROJECTS_ST, EMPLOYEES_ST	15
Basistabelle	<name>_bt	PROJECTS_BT, PROJECTS_BT	15

Abbildung 53. Namenskonventionen

14. PL/SQL - SQL Funktionen

05

SQL Funktionen

01. SQL Funktionen

128

14.1. SQL Funktionen

SQL als Programmiersprache wurde als **Sprach-schnittstelle** für Informationssystemen konzipiert.

14.1.1 SQL Funktionen

Mit PL/SQL Funktionen kann die **SQL Sprachspezi-fikation** um neue Funktionen erweitert werden.

► Erklärung: SQL Funktionen anlegen ▾

- Zur Erweiterung der SQL Sprachspezifikation wird die gewünschte Funktionalität in Form von PL/SQL Funktionen definiert.
- **SQL Funktionen** müssen für ihren Einsatz in einer Datenbank dabei bestimmte Bedingungen erfüllen.
- SQL Funktionen sind **deterministisch**. Als deterministisch werden Vorgänge bezeichnet, die für gleiche Ausgangsbedingungen, zu gleichen Ergebnissen führen.
- Deterministische Funktionen können in einer Datenbank **parallel** ausgeführt werden.
- Zum Anlegen einer SQL Funktion wird der **create function** Befehl verwendet.

► Syntax: create function ▾

```

1  -- -----
2  -- Syntax: SQL Funktion
3  --
4  CREATE [OR REPLACE] FUNCTION <name>
5      [(Parameter1, Parameter2, ...)]
6  RETURN <datatype> DETERMINISTIC
7  PARALLEL_ENABLE
8  IS
9      declare variable, constant, etc. here
10 BEGIN
11     executable statements;
12
13     RETURN (Return Value);
14 EXCEPTION
15     WHEN OTHERS THEN
16         ...
17
18 END [<name>];

```



SQL Funktion



14.1.2 Programmieren von SQL Funktionen

Bei der Programmierung von SQL Funktionen müssen bestimmte Restriktionen eingehalten werden.

► Auflistung: Restriktionen ▾

- **globale Kontext:** Damit ein PL/SQL Funktion in einer **select** Abfrage verwendet werden kann muss sie im globalen Kontext der Datenbank verfügbar sein.
- **Befehle:** In einer SQL Funktionen dürfen keine **DDL** bzw. **DCL** Befehle verwendet werden. Damit wird verhindert dass durch das Ausführen der Funktion Nebeneffekte auftreten.
- **Parameter:** Funktionsparameter in SQL Funktionen müssen einen SQL kompatiblen Datentyp haben. Funktionsparameter dürfen dabei auch keine **out** Parameter sein.
- **null Werte:** Tritt ein **Fehler** bei der Ausführung einer SQL Funktion auf sollte darauf nicht mit dem Werfen einer Exception reagiert werden. Bei der Ausführung einer **select** Anweisung sollte nie ein Fehler auftreten.

Auf Fehlerereignisse sollte immer mit der Rückgabe des null Werts reagiert werden.



14.1.3 Fallbeispiel: Berechnung der Fakultät

► Codebeispiel: Fakultät berechnen ▾

```

1  --
2  -- Fakultaet berechnen
3  --
4  CREATE OR REPLACE FUNCTION FACTORIAL(
5      p_n IN NATURAL
6  )
7  RETURN NUMBER DETERMINISTIC
8  PARALLEL_ENABLE
9  IS
10     NUMERIC_VALUE_EXCEPTION EXCEPTION;
11    pragma exception_init(
12        NUMERIC_VALUE_EXCEPTION, -6502
13    );
14
15    l_n NATURAL RANGE 0 .. 8 := round(p_n);
16    l_result NUMBER := 1;
17
18    BEGIN
19        IF l_n IN (0, 1) THEN
20            l_result := 1;
21        ELSE
22            FOR i = 2 .. p_n LOOP
23                l_result := l_result * i;
24            END LOOP;
25        END IF;
26
27        RETURN l_result;
28    EXCEPTION
29        WHEN numeric_value_exception THEN
30            RETURN null;
31    END FACTORIAL;

```


MongoDB - Dokumentorientierte Datenbanken

August 19, 2020

15. MongoDB - DDL

02

MongoDB - DDL

01. Datenkontainer	132
02. Datenkontainer anlegen	133
03. Collections und Dokumente	134
04. Dokumentschema	136
05. Schemaelemente	138
06. Views ersetzen	140
07. Datencontainer verwalten	141

15.1. Datenkontainer

Die MongoDB Datenbankengine verwaltet **Sammlungen** von **Dokumenten** in logischen Namensräumen.

15.1.1 Datenkontainer

Zur Verwaltung von Daten definiert die MongoDB Spezifikation 5 Typen von Datenkontainern.

Datenkontainer dienen zur **Strukturierung** und **Verwaltung** von Daten.

► Auflistung: Datenkontainer ▾



Database ▾

Als Datenbank werden die Strukturen und Objekte zur Verwaltung der Daten eines logisch **zusammenhängenden Datenbestandes** bezeichnet.

Aus technischer Sicht entspricht eine Datenbank einem logischen Namensraum.



Collection ▾

Collections verwalten **Sammlungen gleichartiger Dokumente**.

Collections können dabei konzeptionell mit den Tabellen Relationaler Datenbanken verglichen werden.



View ▾

Views ermöglichen es **Datenbankabfragen** als Objekte zu speichern.



Capped Collection ▾

Capped Collection sind eine Collection die strukturell wie ein **Ringbuffer** aufgebaut ist. Sie wird verwendet um große Mengen von Daten schnell verarbeiten zu können.



Document ▾

MongoDB speichert **Datensätze** in Form von Dokumenten.

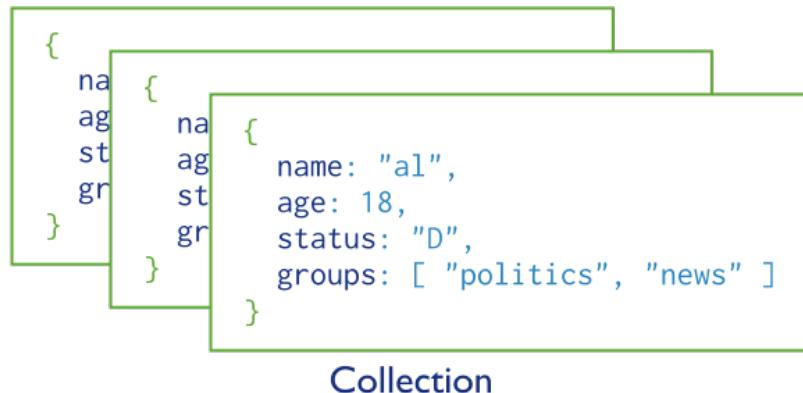


Abbildung 54. Dokumente einer Collection

15.2. Datenkontainer anlegen

Database ▾

Eine Datenbank dient als logischer **Namensraum** für **Collections**.

15.2.1 Datencontainer implizit anlegen

Die MongoDB Datenbankengine verwendet **JavaScript** als **Abfragesprache** und **Datenbanksprache**.

Beim Ausführen bestimmter designierter Befehle werden Datencontainer **implizit** in der Datenbank angelegt.

▶ Codebeispiel: Datenbanken implizit anlegen ▾

```

1 // -----
2 // Datencontainer implizit anlegen
3 // -----
4 // Mit dem <use> Befehl wird der Kontext
5 // einer Datenbank geladen. Existiert die
6 // Datenbank zu diesem Zeitpunkt nicht,
7 // wird sie angelegt.
8 use persons;
9
10 // Mit dem insertOne Befehl koennen Doku-
11 // mente in Collections eingetragen wer-
12 // den. Existiert die Collection zu diesem
13 // Zeitpunkt nicht, wird sie implizit
14 // angelegt.
15 db.projects.insertOne({name : "E.T."});

```

15.2.2 Collections explizit anlegen

Zum expliziten **Anlegen** von **Collections** definiert die MongoDB Spezifikation den `createCollection` Befehl.

▶ Codebeispiel: createCollection ▾

```

1 // -----
2 // Syntax: createCollection
3 // -----
4 db.createCollection(
5   name : <String>,
6   option : <Document>
7 );
8
9
10 db.createCollection(<name>, {
11   capped : <boolean>,
12   autoIndexId : <boolean>,
13   size : <number>,
14   max : <number>,
15   storageEngine : <document>,
16   validator : <document>,
17   validationLevel : <string>,
18   validationAction : <string>,
19   indexOptionDefaults : <document>,
20   viewOn : <string>,
21   pipeline : <pipeline>,
22   collation : <document>,
23   writeConcern : <document>
24 });
25
26
27 db.createCollection("projects");

```

Parameter	Beschreibung	Typ
capped	Optional. To create a capped collection, specify true. If you specify true, you must also set a maximum size in the size field.	boolean
autoIndexId	Optional. Specify false to disable the automatic creation of an index on the _id field.	boolean
size	Optional. Specify a maximum size in bytes for a capped collection. Once a capped collection reaches its maximum size, MongoDB removes the older documents to make space for the new documents. The size field is required for capped collections and ignored for other collections.	number
max	Optional. The maximum number of documents allowed in the capped collection. The size limit takes precedence over this limit. If a capped collection reaches the size limit before it reaches the maximum number of documents, MongoDB removes old documents. If you prefer to use the max limit, ensure that the size limit, which is required for a capped collection, is sufficient to contain the maximum number of documents.	number
validator	Optional. Allows users to specify validation rules or expressions for the collection. For more information, see Schema Validation.	document
validationLevel	Optional. Determines how strictly MongoDB applies the validation rules to existing documents during an update.	string
validationAction	Optional. Determines whether to error on invalid documents or just warn about the violations but allow invalid documents to be inserted.	string
viewOn	The name of the source collection or view from which to create the view. The name is not the full namespace of the collection or view; i.e. does not include the database name and implies the same database as the view to create.	string
pipeline	An array that consists of the aggregation pipeline stage. db.createView creates the view by applying the specified pipeline to the viewOn collection or view.	array
writeConcern	Optional. A document that expresses the write concern for the operation. Omit to use the default write concern.	document

Abbildung 55. createCollection

15.3. Collections und Dokumente

 Collection ▾

Collections verwalten **Sammlungen** gleichartiger Dokumente.

Die MongoDB Datenbankengine speichert Datensätze in Form von **BSON Dokumenten**.

15.3.1 BSON Datenformat

Das BSON Datenformat ist eine Variation des JSON Datenformats.

► Erklärung: BSON Datenformat ▾

- BSON ist ein **binäres Datenformat** zur Speicherung semistrukturierter Daten.
- Das BSON Datenformat kann im Vergleich zu JSON **schneller** von der MongoDB Datenbankengine verarbeitet werden.
- BSON Dokumente speichern Daten dabei in Form von **Key - Value** Paaren.
- Für die Namen der Attribute eines Dokuments müssen folgende Restriktionen eingehalten werden: Der Name **_id** darf nur für das Schlüsselfeld des Dokuments verwendet werden. Attributnamen dürfen nicht mit dem \$ Zeichen beginnen.

BSON Typ	Beschreibung	Key Value Paar im Dokument
object	BSON Objekt	field : {...}
array	Eine Array speichert eine Liste beliebiger Werte	field : [...]
string	UTF 8 codierte Zeichenkette	field : "hallo Mongo"
double	64 Bit Gleitkommazahlen	field : Math.PI
long	64 Bit Integer	field : NumberLong(32353423)
int	32 Bit Integer mit Vorzeichen	field : NumberInt(45)
bool	Datentyp für Wahrheitswerte	field : true
objectId	Eine 12 Byte lange ID zur Identifikation des Dokuments: 24 Bit Hexadecimalzahl	field : ObjectId("312G64AC6543233F3232902")

Abbildung 56. BSON Datentypen

▶ Codebeispiel: BSON Dokument ▾

```

1 //-----
2 // BSON Dokument
3 //-----
4 var project = {
5     // Das _id Attribut darf nur zur Spei-
6     // cherung der ID des Dokuments ver-
7     // wendet werden. Eine ID ist eine 24
8     // bit Hexadecimallzahl
9     _id : ObjectId("5099803df3f4948bd1"),
10    title: "Motorsimulation"
11
12    // Das contact Feld referenzier ein einge-
13    // bettetes Objekt
14    contact : { first: "Alan", last: "Tur" },
15    type : "REQUEST_FUNDING_PROJECT",
16    begin : new Date('Jun 23, 1995'),
17    state : "APPROVED",
18
19    // contribs speichert ein Liste von
20    // Zeichenketten
21    contribs: [
22        "Turing machine",
23        "Turing test"
24    ],
25
26    views      : NumberLong(1250000)
27 }

```

15.3.2 Eingebettete Objekte



Embedded Document ▾

In einem BSON Objekten können andere Objekte gespeichert werden. Eingebettete Objekte werden als **Embedded Objects** bezeichnet.

Eingebettete Objekte können über den **Punktorperator** referenziert werden.

▶ Codebeispiel: embedded documents ▾

```

1 //-----
2 // embedded document
3 //-----
4 var project = {
5     _id : ObjectId("5099803df3f4948bd2"),
6     type : "REQUEST_FUNDING_PROJECT",
7     contact : {
8         name : "Fabricio Caprese",
9         phone : {
10             number : "111-222-3333"
11         }
12     }
13 }
14
15 // Zugriff auf Felder
16 project.contact.phone.number

```

15.4. Dokumentschema ▾

Schemafreiheit beschreibt die Eigenschaft eines Informationssystems, Datensätze beim Einfügen bzw. Ändern keiner Strukturprüfung zu unterziehen.

15.4.1 BSON Schema

In der Regel können Anwendungen Daten nur dann verarbeiten, wenn sie eine bestimmte **Struktur** aufweisen.

Die **BSON Schema Spezifikation** ist ein technischer Standard zur Beschreibung der Struktur von BSON Dokumenten.

► Erklärung: BSON Schema ▾

- Aus fachlicher Sicht macht es keinen Sinn, Geschäftsobjekte unreflektiert in einer Collection zu speichern.
- Mit einem BSON Schema kann definiert werden, wie ein Dokument aufgebaut sein muss, um in eine Collection eingetragen werden zu können.

► Codebeispiel: BSON Schema ▾

```

1 //-----
2 // BSON Dokument
3 //-----
4 var project = {
5   title : "Motorensimulation",
6   type : "REQUEST_PROJECT",
7   views : 10000
8 };
9
10 //-----
11 // BSON Schema
12 //-----
13 var projectSchema = {
14   bsonType : "object",
15   required : ["title", "views"],
16   additionalProperties : true,
17   properties : {
18     title : {
19       bsonType : "string",
20       maxLength : 100
21     },
22     views : {
23       bsonType : "int",
24     }
25   }
26 };
27
28
29
30
31 }
```

15.4.2 Schemaattribute

Ein BSON Schema selbst ist wieder ein **BSON Dokument**.

Zur Definition eines BSON Schemas gibt die Schema Spezifikation eine Zahl von Attributen vor.

► Auflistung: Schemaattribute ▾

- **bsonType**: Das **bsonType** Attribut definiert den Typ des Dokuments bzw. den Typ im Dokument enthaltener Felder.
- **required**: Mit dem **required** Attribut wird angegeben, welche **Felder** ein Dokument umbedingt enthalten muss.
- **properties**: Das **properties** Attribut wird verwendet, um für die Felder des Dokuments **Constraints** zu definieren.

► Codebeispiel: Schemaattribute ▾

```

1 //-----
2 // BSON Dokument
3 //-----
4 var person = {
5   _id : ObjectId("5099803df3f4948391"),
6   lastName : "Turning",
7   contribs : [ "Turing machine", "test" ],
8   major : "Math"
9 }
10
11 //-----
12 // BSON Schema
13 //-----
14 var personSchema = {
15   bsonType : "object",
16   required : [ "major", "lastName" ],
17   properties : {
18     contribs : {
19       bsonType : "array",
20       items : {
21         bsonType : "string"
22       }
23     },
24     major : {
25       enum : [ "Math", "English" ]
26     },
27     lastName : {
28       bsonType : "string"
29     }
30   }
31 }
```

Keyword	Definition	Behavior	Type
bsonType	string alias	Definiert den Typ des Feldes	all types
enum	array of values	Definiert alle möglichen Werte für ein Feld.	all types
multipleOf	number	Field must be a multiple of this value	numbers
maximum	number	Indicates the maximum value of the field	numbers
exclusiveMaximum	boolean	If true and field is a number, maximum is an exclusive maximum. Otherwise, it is an inclusive maximum.	numbers
minimum	number	Indicates the minimum value of the field	numbers
exclusiveMinimum	boolean	If true and field is a number, minimum is an exclusive minimum. Otherwise, it is an inclusive minimum.	numbers
maxLength	integer	Indicates the maximum length of the field	strings
minLength	integer	Indicates the minimum length of the field	strings
pattern	regex string	Field must match the regular expression	strings
required	array of strings	Object's property set must contain all the specified elements in the array	objects
additionalProperties	boolean or object	If true, additional fields are allowed. If false, they are not. If a valid JSON Schema object is specified, additional fields must validate against the schema.	objects
items	bson type	Definiert den Typ der Elemente die im Array gespeichert werden	arrays
maxItems	integer	Indicates the maximum length of array	arrays
minItems	integer	Indicates the minimum length of array	arrays
uniqueItems	boolean	If true, each item in the array must be unique. Otherwise, no uniqueness constraint is enforced.	arrays
title	string	A descriptive title string with no effect	N/A
description	string	A string that describes the schema and has no effect.	N/A

Abbildung 57. BSON Schema Attribute

15.5. Schemaelemente

15.5.1 Schemaelement: Eingebettete Objekte

Für in Objekte eingebettete Objekte erfolgt die Schemadefinition in **rekursiver Weise**.

► Codebeispiel: Eingebettete Objekte ▾

```

1 //-----
2 // BSON Dokument
3 //-----
4 var person = {
5     _id : ObjectId("5099803df3f4948b91"),
6     firstName : "Alan",
7     address : {
8         country : "Austria",
9         location : "1040 Vienna",
10        street : "Taubstummengasse 11"
11    }
12 };
13
14 //-----
15 // BSON Schema
16 //-----
17 var personSchema = {
18     bsonType : "object",
19     required : ["firstName", "address"],
20     properties : {
21         first_name : {
22             bsonType : "string",
23             description : "..."
24         },
25         address : {
26             bsonType : "object",
27             required : [
28                 "country", "street"
29             ],
30             properties : {
31                 country : {
32                     bsonType : "string",
33                     minLength : 2,
34                     maxLength : 100
35                 },
36                 street : {
37                     bsonType : "string"
38                 }
39             }
40         }
41     }
42 };

```

15.5.2 Schemaelement: Enum



Aufzählungstyp ▾

Ein **Aufzählungstyp** ist ein Datentyp mit festgelegten Wertemöglichkeiten.

Enums werden verwendet um die **logische Struktur** und Lesbarkeit von Programmen zu verbessern.

► Erklärung: Aufzählungstypen ▾

- Zur Definition eines Enums, müssen im Schema alle zulässigen Werte des Aufzählungstyps angeführt werden.
- Dabei kann auch eine **Reihenfolge** festgelegt werden, die eine Ordnung der einzelnen Werte bestimmt.

► Codebeispiel: Enums ▾

```

1 //-----
2 // BSON Dokument
3 //-----
4 var project = {
5     name : "Computer Simulation",
6     code : "A-33434-ZTU",
7     projectType : "REQUEST_PROJECT"
8 };
9
10 //-----
11 // BSON Schema
12 //-----
13 var projectSchema = {
14     bsonType : "object",
15     required : [
16         "name", "code", "projectType"
17     ],
18     properties : {
19         name : {
20             bsonType : "string",
21         },
22         projectType : {
23             enum : [
24                 "REQUEST_PROJECT",
25                 "RESEARCH_PROJECT",
26                 "MANAGEMENT_PROJECT"
27             ]
28         }
29     }
30 };

```

15.5.3 Schemaelement: Array

Arrays fassen mehrere gleichartige Werte zu einer Collection zusammen.

► Erklärung: Arraydefinition ▾

- Der BSON Typ eines Arrays ist `array`. Zur Definition eines Arrays muß zusätzlich die Art und der Aufbau der Arrayelemente angegeben werden.
- Im Schema wird dazu das `items` Attribut verwendet.

► Codebeispiel: Arrayschema ▾

```

1 //-----
2 // BSON Dokument
3 //-----
4 var point = {
5     location : [3, 2, 5],
6     props : [
7         "value coordinate",
8         "xyz values",
9     ],
10    description : "pipeline point"
11};
12 //-----
13 // BSON Schema
14 //-----
15 var pointSchema = {
16     bsonType : "object",
17     required : [
18         "location", "props", "dimension"
19     ],
20     properties : {
21         location : {
22             bsonType : "array",
23             minItems : 1,
24             maxItems : 4,
25             items : {
26                 bsonType : "int"
27             }
28         },
29         props : {
30             bsonType : "array",
31             maxItems : 3,
32             items : {
33                 bsonType : "string"
34             }
35         }
36     }
37 };
38 
```

15.5.4 Arrays von Objekten

Zur Definition eines Arrays von Objekten muß das `items` Attribut verwendet werden.

► Codebeispiel: Arrays von Objekten ▾

```

1 //-----
2 // BSON Dokument
3 //-----
4 var project = {
5     title : "Simulationssoftware",
6     subprojects : [
7         {
8             title : "Finite Elemente"
9             isFWFFunded : true,
10            isFFGFunded : false
11        },
12        fundings : [
13            {
14                debtor : "TU Wien",
15                amount : NumberLong(100000)
16            },
17            {
18                debtor : "Simens AG.",
19                amount : NumberLong(250000)
20            }
21        ]
22 };
23 //-----
24 // BSON Schema
25 //-----
26 var projectSchema = {
27     bsonType : "object",
28     required : [ "title", "subprojects" ],
29     properties : {
30         subprojects : {
31             bsonType : "array",
32             items : {
33                 bsonType : "object",
34                 required : [ "title" ],
35                 properties : {
36                     title : {
37                         bsonType : "string",
38                         maxLength : 100
39                     },
40                     isFwfFunded : {
41                         bsonType : bool
42                     }
43                 }
44             }
45         }
46     }
47 };
48 
```

15.5.5 Collection mit Schemavalidation

Sollen Dokumente beim Einfügen in eine Collection einer **Strukturprüfung** unterzogen werden, muss beim Anlegen der Collection eine Schemabeschreibung definiert werden.

► Auflistung: Attribute ▾

- **validationLevel:** Das validationLevel Attribut definiert den **Validierungslevel** einer Collection.
 - **strict:** Für inserts bzw. updates wird eine Strukturprüfung durch die Datenbankengine angestossen.
 - **moderate:** Für bereits gespeicherte Dokumente erfolgt keine Validierung. Ansonsten gelten die strict Regeln.
- **validationAction:** Das validationAction Attribut definiert das Verhalten der Datenbankengine im Falle des Fehlschagens der Strukturprüfung.
 - **error:** Ungültige Dokumente werden nicht in die Datenbank eingetragen. Die Datenbankengine antwortet mit einem **Fehler**.
 - **warn:** Für ungültige Dokumente wird eine Warnung ausgegeben. Das Dokument wird jedoch trotzdem in die Collection geschrieben.

► Codebeispiel: Collection anlegen ▾

```

1 //-----
2 // Collection mit Validierung anlegen
3 //-----
4 db.createCollection( "persons",
5 {
6   validationLevel : "strict",
7   validationAction : "error",
8   validator : {
9     $jsonSchema : {
10       bsonType : "object",
11       required : ["name"],
12       properties : {
13         name : {
14           bsonType : "string",
15           maxLength : 50,
16           description : "..."
17         }
18       }
19     }
20   }
21 };
22 };

```

15.6. Views erstellen

15.6.1 Views



View ▾

Views sind **Sichten** auf Collections.

Auf Views können nur **Leseoperationen** ausgeführt werden.

► Codebeispiel: View anlegen ▾

```

1 //-----
2 // Syntax: createView
3 //-----
4 db.createView(<view>, <source>);
5
6 db.createView("students", "persons");
7
8 //-----
9 // Syntax: createCollection
10 //-----
11 db.createCollection(
12   "persons", {
13     validationLevel : "strict",
14     validationAction : "error",
15     validator : {
16       $jsonSchema : {
17         bsonType : "object",
18         required : ["name",
19           "birth"],
20         additionalProperties : true,
21         properties : {
22           name : {
23             bsonType :
24               "string",
25             minLength : 3,
26             maxLength : 50
27           }
28         birth : {
29           bsonType : "date",
30         },
31         contribs : {
32           bsonType :
33             ["string"]
34         }
35       }
36     }
37   }
38 );

```

15.7. Datencontainer verwalten

15.7.1 Container verwalten

Wir wollen zum Schluß noch die wichtigsten Methoden zur Verwaltung von Datencontainern angeben.

► Codebeispiel: Container verwalten ▾

```

1 //-----
2 // Syntax: getCollection()
3 //-----
4 db.getCollection(<string>);

5
6 var personColl = db.getCollection("persons");

7
8 personColl.insertOne({
9     birth : new Date('Jun 23, 1912'),
10    death : new Date('Jun 07, 1954'),
11    views   : NumberLong(1250000),
12    user_name : "turing",
13    email      : "alan.turing@berkley.com",
14    phone      : "0664/3452372",
15    url       : "http://www.htlkrema.at"
16 });
17
18 //return value
19 var returnValue = {
20     "acknowledged" : true,
21     "insertedId" : ObjectId("569525e144f63")
22 }
23
24 //-----
25 // Syntax: getCollectionNames()
26 //-----
27 //return all collections
28 db.getCollectionNames();

29
30 use school;
31 db.getCollectionNames();

32
33 //return value
34 var returnValue = ["persons", "teachers",
35   "students"]
36
37 // Syntax: drop()
38 //-----
39 db.<collectionName>.drop();
40
41 db.teachers.drop();
42 db.students.drop();

```



04

MongoDB - DQL

01. Daten lesen	142
02. Kursormethoden	143
03. Query Kriterien	146
04. Arrayoperatoren	148
05. Projektion	149

16. MongoDB - DQL

16.1. Daten lesen

Zum **Abfragen** von Daten definiert die MongoDB Spezifikation den **find** Befehl.



16.1.1 find Befehl

Abfragen werden für die Datenbankengine in Form von **BSON Objekten** formuliert. Das Ergebnis einer Abfrage ist ein **Cursor**.

Abfragen können jedoch nur für die Dokumente einer einzelnen **Collection** formuliert werden. Das Konzept eines Joins über mehrere Collections wird nicht unterstützt.

► Codebeispiel: find Befehl ▾

```

1 //-----
2 // Syntax: find
3 //-----
4 var crsr = db.<collection>.find(
5     <query criteria>,
6     <projection>
7 );
8 //-----
9 // Beispiel: find
10 //-----
11 var crsr = db.inventory.find( {
12     status: "A",
13     $or: [
14         { qty: { $lt: 30 } },
15         { item: "p" }
16     ]
17 } );
18 while (crsr.hasNext()){
19     printjson( crsr.next() );
20 }
21
22 }
```

► Erklärung: Methoden Parameter ▾

- **query criteria:** Eine Abfrage selbst wird in Form eines **BSON Objekts** formuliert.
- **projection:** Mit einer Projektion können die Ergebnisdokumente auf eine Untermenge ihrer Attribute beschränkt werden.
- **Cursor:** Das Ergebnis einer Abfrage ist ein **Cursor**.



```
db.projects.find(
  { level : { $gt : 4 } },   ← query criteria
  { name : 1, address : 1 } ← projection
).limit(10);
```

Abbildung 58. find Befehl

16.2. Kursormethoden



ResultSet ▾

Das **Ergebnis** einer Abfrage wird als Resultset bezeichnet. Im Falle einer MongoDB Datenbank ist das eine Sammlung von **Dokumenten**.



Cursor ▾

Ein **Cursor** ist eine **Referenz** auf die Elemente eines Resultsets.

Das Ergebnis der `find()` Methode ist stets ein **Cursor**.

16.2.1 Cursormethoden

Zur Verarbeitung eines Resultsets stellt die MongoDB Spezifikation eine Reihe von **Methoden** zur Verfügung.

▶ Codebeispiel: Cursormethoden ▾

```
1 //-----
2 // Beispiel: find
3 //-----
4 var crsr = db.inventory.find( {
5     status: "A",
6     $or: [
7         { qty: { $lt: 30 } },
8         { item: "p" }
9     ]
10} );
11
12 crsr = crsr.sort({ item: -1 });
13
14 while ( crsr.hasNext() ){
15     printjson( crsr.next() );
16 }
```



```
1 //-----
2 // Ausgabe
3 //-----
4 {
5     _id : ObjectId(2),
6     firstName : "Franz",
7     lastName : "Xaver",
8     roles : [
9         "student",
10        "room manager"
11    ]
12 }
```

16.2.2 pretty() Methode

Die **Darstellung** von JSON Objekte ist in der Shell ab einer gewissen Komplexität schlecht lesbar.

Die `pretty()` Methode optimiert die **Darstellung** von Objekten für die Ausgabe in der Konsole.

▶ Codebeispiel: pretty Methode ▾

```
1 //-----
2 // Cursormethode: pretty
3 //-----
4 db.students.remove();
5
6 db.students.insertOne( {
7     _id      : ObjectId(2),
8     firstName : "Franz",
9     lastName : "Xaver",
10    roles    : ["student", "room manager" ]
11} );
12
13 var crsr = db.students.find();
14 crsr.pretty();
15
16 while ( crsr.hasNext() ){
17     printjson(crsr.next());
18 }
19
20 //-----
21 // Ausgabe
22 //-----
23 {
24     _id : ObjectId(2),
25     firstName : "Franz",
26     lastName : "Xaver",
27     roles : [
28         "student",
29         "room manager"
30     ]
31 }
```

16.2.3 sort() Methode

Durch den Aufruf der `sort()` Methode, werden die Dokumente eines Resultsets einer Sortierung unterzogen.

► Erklärung: sort Methode ▾

- Als Parameter erwartet die Methode ein Objekt, deren Attributen die Felder definieren, nach denen sortiert werden soll.
- Der Wert des Attributes definiert dabei das **Sortierverhalten** an: (1 = aufsteigend, -1 = absteigend).
- Idealerweise sollte nach Feldern sortiert werden, für die ein Index generiert wurde.

► Codebeispiel: sort Methode ▾

```

1 //-----
2 //  Cursormethode: pretty
3 //-----
4 db.points.remove();
5
6 db.points.insertMany(
7     { x : 0, y : 2, z : 4 },
8     { x : 0, y : 3, z : 4 },
9     { x : 1, y : 4, z : 4 },
10    { x : 1, y : 5, z : 5 },
11    { x : 1, y : 6, z : 5 },
12    { x : 2, y : 6, z : 5 }
13 );
14
15 var crsr = db.points.find();
16
17 crsr.sort( {
18     x : -1,
19     y : -1
20 } );
21
22 while ( crsr.hasNext() ){
23     printjson(crsr);
24 }
25
26 //-----
27 //  Ausgabe
28 //-----
29 { x : 2, y : 6, z : 5 }
30 { x : 1, y : 6, z : 5 }
31 { x : 1, y : 5, z : 5 }
32 { x : 1, y : 4, z : 4 }
33 { x : 0, y : 3, z : 4 }
34 { x : 0, y : 2, z : 4 }
```

16.2.4 limit() Methode

Durch die Verwendung der `limit()` Methode wird das Resultsets auf seine ersten n Elemente **beschränkt**.

Ein Limit von 0 zeigt alle Datensätze an. Für negative Werte wird der Betrag berechnet.

► Codebeispiel: limit Methode ▾

```

1 //-----
2 //  Cursormethode: pretty
3 //-----
4 db.points.remove();
5
6 db.points.insertMany(
7     { x : 0, y : 1, z : 4 },
8     { x : 0, y : 2, z : 4 },
9     { x : 0, y : 3, z : 4 }
10 );
11
12 var crsr = db.points.find();
13 crsr.limit(2);
14
15 //-----
16 //  Ausgabe
17 //-----
18 { x : 0, y : 1, z : 4 },
19 { x : 0, y : 2, z : 4 }
```



16.2.5 skip() Methode

Mit der `skip()` Methode kann der Cursor eines Resultsets verändert werden.

► Codebeispiel: skip Methode ▾

```

1 //-----
2 //  Cursormethode: skip
3 //-----
4 var crsr = db.inventory.find( {
5     status: "A",
6     qty: { $lt: 30 },
7     item: "P"
8 } );
9
10 crsr = crsr.skip(100);
11 crsr.limit(20);
```

Methode	Beschreibung	Seite
<code>cursor.forEach()</code>	Applies a JavaScript function for every document in a cursor.	145
<code>cursor.map()</code>	Applies a function to each document in a cursor and collects the return values in an tba array.	tba
<code>cursor.skip()</code>	Returns a cursor that begins returning results only after passing or skipping a number of documents.	144
<code>cursor.limit()</code>	Constrains the size of a cursor's result set.	144
<code>cursor.size()</code>	Returns a count of the documents in the cursor after applying skip() and limit() methods.	tba
<code>cursor.count()</code>	Modifies the cursor to return the number of documents in the result set rather than the documents themselves.	tba
<code>cursor.sort()</code>	Returns results ordered according to a sort specification.	144
<code>cursor.hasNext()</code>	Returns true if the cursor has documents and can be iterated.	tba
<code>cursor.next()</code>	Returns the next document in a cursor.	tba
<code>cursor batchSize()</code>	Controls the number of documents MongoDB will return to the client in a single network message.	145
<code>cursor.close()</code>	Close a cursor and free associated server resources.	tba
<code>cursor.isClosed()</code>	Returns true if the cursor is closed.	tba

Abbildung 59. Kursormethoden

16.2.6 forEach() Methode

Zur weiteren Verarbeitung der Elemente eines Resultsets definiert die MongoDB Spezifikation die `forEach()` Methode.

► Codebeispiel: forEach Methode ▾

```

1 //-----
2 // Cursormethode: forEach
3 //-----
4 var crsr = db.persons.find({});  

5  

6 crsr.forEach(function(doc){  

7     db.persons.updateOne(  

8         { _id : doc._id },  

9         { $set : {  

10             email : doc.lastName;  

11         }  

12     }  

13 });
14 });

```

16.2.7 batchSize(), objsLeftInBatch() Methode

Die Batchsize definiert die maximale Zahl an Elementen, die in einem Resultset enthalten sein können.

► Codebeispiel: batchSize Methode ▾

```

1 //-----  

2 // Cursormethode: batchSize  

3 //-----  

4 // Mit der batchSize()} Methode wird die  

5 // gegenwärtige Batchsize der Datenbankengine  

6 // konfiguriert.  

7 var crsr = db.points.find(  

8     {}, { _id:0 }  

9 );  

10 crsr.batchSize(100);  

11  

12 while( crsr.hasNext() ) {  

13     printjson(crsr.next());  

14     print(crsr.objsLeftInBatch());  

15 }

```



16.3. Query Kriterien - Abfrageobjekt ▾

Zum Abfragen von Daten definiert die MongoDB Spezifikation den `find()` Befehl. Abfragen selbst werden dabei in Form eines **BSON Dokuments** formuliert.

16.3.1 Abfrageobjekt

Ein Abfrageobjekt definiert welche **Eigenschaften** bzw. **Struktur** ein Dokument haben muß, um in die Ergebnismenge einer Abfrage aufgenommen zu werden.

Abfragen dieser Art werden als *Query by Example* bezeichnet.

► Erklärung: Abfrageobjekt ▾

- Ein Abfrageobjekt hat dabei stets die selbe **Struktur**.
- Ein Abfrageobjekt definiert eine Logische Verknüpfung von Bedingungen. Die Logische Verknüpfung erfolgt dabei über einen der Operatoren `$and`, `$or` bzw. `$nor`.

► Syntax: Abfrageobjekt ▾

```

1 //-----
2 // Syntax: Abfrageobjekt
3 //-----
4 db.<collection>.find({
5     $logischerOperator : [
6         { .. mathematische Bedingung .. },
7         { .. mathematische Bedingung .. },
8         { .. mathematische Bedingung .. },
9         ...
10    ]
11 });
12
13 db.projects.find({
14     $and : [
15         {type : {$eq : "FUNDING_PROJECT"}},
16         {state : {$eq : "IN_PROGRESS"}},
17         {isFWFSponsored : {$eq : true}}
18     ]
19 });
20
21 db.projects.find({
22     $nor : [
23         {type : {$eq : "FUNDING_PROJECT"}},
24         {type : {$eq : "REQUEST_PROJECT"}}
25     ]
26 });
27 });

```

16.3.2 Formen von Bedingungen

Als **Bedingung** wird eine **mathematische Aussage** bezeichnet die entweder wahr oder falsch sein kann.

► Auflistung: Formen von Bedingungen ▾

```

1 //-----
2 // Syntax: Bedingung
3 //-----
4 { <field> : { $operator : <value> }}
5
6 {title : {$eq : "Planungssysteme"}}
7 {type : {$ne : "FUNDING_PROJECT"}}
8 {isFFGSponsored : {$eq : true}}
9
10 //-----
11 // $eq - equal, $ne - not equal
12 //-----
13 db.projects.find({
14     $nor : [
15         {type : {$eq : "MANAGEMENT_PROJECT"}},
16         {type : {$eq : "RESEARCH_PROJECT"}},
17         {type : {$eq : "FUNDING_PROJECT"}},
18         {type : {$eq : "STIPENDIUM"}}
19     ]
20 });
21
22 db.projects.find({
23     $or : [
24         {isFFGSponsored : {$ne : false}},
25         {isFWFSponsored : {$ne : false}},
26         {isEUSponsored : {$ne : false}}
27     ]
28 });
29
30 // Einzelne Bedingung
31 db.projects.find({
32     type : { $eq : "RESEARCH_PROJECT" }
33 });
34
35 //-----
36 // $lt, $lte - less than, less than equal
37 // $gt, $gte - greater than
38 //-----
39 db.projects.find({
40     $and : [
41         {review : {$gte : 3}},
42         {review : {$lte : 5}}
43     ]
44 });

```



```
db.<collection>.find({
    $logischerOperator : [
        {}, {}
    ]
});
```

← Query Kriterien
← LogischerOperator

Abbildung 60. Abfrageobjekt

16.3.3 Kurzformen von Abfragen

Für bestimmte Abfrageformen stellt die MongoDB Spezifikation eine **vereinfachte Form** zur Verfügung.

► Codebeispiel: Kurzformen ▾

```
1 //-----
2 // Fallbeispiel: Kurzformen
3 //-----
4 // Kurzform: $eq Operator
5 db.projects.find({
6     type : {$eq : "REQUEST_FUNDING_PROJECT"}
7 });
8 /* ==> */
9 db.projects.find({
10    type : "REQUEST_FUNDING_PROJECT"
11 });
12 // Kurzform: $and Operator
13 db.projects.find({
14     $and : [
15         {isFWFSponsored : true},
16         {isFFGSponsored : true}
17     ]
18 });
19 /* ==> */
20 db.projects.find({
21     isFWFSponsored : true,
22     isFFGSponsored : true
23 });
24 //
25 // Kurzform: Abfragebedingungen
26 db.projects.find({
27     review : {$gt:3}, review : {$lt:10}
28 });
29 /* ==> */
30 db.projects.find(
31     {review : {$gt : 3, $lt : 10}}
32 );
33 );
```

16.3.4 Komplexe Bedingungen

Zur Formulierung **komplexer Abfragen** erlaubt die MongoDB Spezifikation das Schachteln logischer Ausdrücke.

► Codebeispiel: Komplexe Bedingungen ▾

```
1 //-----
2 // Fallbeispiel: komplexe Bedingungen
3 //-----
4 db.projects.find({
5     $or : [
6         {
7             $and : [
8                 {type : {$eq : "FUNDING_PROJECT"}},
9                 {isFFGSponsored : {$eq : true}}
10            ]
11        },
12        {
13            $and : [
14                {type : {$eq : "REQUEST_PROJECT"}},
15                {isFWFSponsored : {$eq : true}}
16            ]
17        }
18    );
19 //
20 // -----
21 // Komplexe Bedingungen : embedded Object
22 // -----
23 db.subprojects.find(
24     $and : [
25         {theoreticalResearch : {$gte : 30}},
26         {
27             "facility.code" : "124.339.125"
28         },
29         {
30             isEUSponsored : true
31         }
32    );
33 );
```

16.3.5 \$in Operator

Mit dem \$in Operator kann geprüft werden, ob ein bestimmter Wert in einer Liste von Werten enthalten ist.

► Codebeispiel: \$in Operator ▾

```

1 //-----
2 //  $in Operator
3 //-----
4 db.inventory.find( {
5     status : {
6         $in [
7             "IN_PROGRESS", "CANCELED",
8             "APPROVED", "FINISHED"
9         ]
10    }
11 } );

```



16.3.6 \$where Operator

Komplexe Abfragen können auch in Form einer **JavaScript Funktion** formuliert werden.

► Erklärung: \$where Operator ▾

- Der \$where Operator speichert dazu die Bedingung in Form einer JavaScript Funktion ab.
- Der this Operator dient in diesem Kontext als **Referenz** auf das eigentliche Dokument.

► Codebeispiel: \$where Operator ▾

```

1 //-----
2 //  $where Operator
3 //-----
4 db.inventory.find( {
5     $where : function(){
6         var flag = true
7
8         if(this.isSmallProject ||
9             this.isEuSponsored)
10            flag = false;
11
12         if(this.fundings.length < 2)
13             flag = flase;
14
15         return flag;
16     }
17 } );

```



16.4. Arrayabfragen

Für Abfragen auf Arraydaten definiert die MongoDB Spezifikation eigene Operatoren.

16.4.1 \$size Operator

Der \$size Operator prüft die Anzahl der Werte in einem Array.

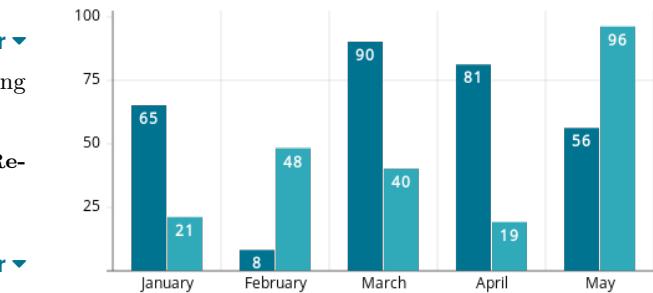
Der \$size Operator erlaubt dabei lediglich Äquivalenzprüfungen.

► Codebeispiel: \$size Operator ▾

```

1 //-----
2 //  $size Operator
3 //-----
4 db.projects.find({
5     $and : [
6         { reviews : { $size : 3 } }
7     ]
8 });

```



16.4.2 \$all Operator

Der \$all Operator prüft ob eine Liste von Werten in einem Array enthalten ist.

► Codebeispiel: \$all Operator ▾

```

1 //-----
2 //  $all Operator
3 //-----
4 db.projects.find({
5     reviews : { $all : [3,6,4] }
6 });
7
8 db.projects.find({ reviews : 5 });

```



Arrayoperator	Beschreibung	Seite
\$all	Der \$all Operator prüft ob eine Liste von Werten in einem Array enthalten ist.	148
\$size	Der \$size Operator prüft die Anzahl der Werte in einem Array.	148
\$elemMatch	Für die Prüfung von Objekten in Arrays wird der \$elemMatch Operator verwendet. Ist im Array ein Objekt gespeichert, das die formulierten Bedingungen erfüllt, wird das Dokument in die Ergebnismenge aufgenommen.	149

Abbildung 61. Arrayoperatoren

16.4.3 \$elemMatch Operator

Zur Prüfung der Eigenschaften von **Objekten in Arrays** wird der \$elemMatch Operator verwendet.

Beinhaltet das Array ein **Element**, das alle Bedingungen erfüllt wird das Dokument in die Ergebnismenge aufgenommen.

► Codebeispiel: \$elemMatch Operator ▾

```

1 //-----
2 //  Abfrageoperatoren: $elemMatch
3 //-----
4 // Der $elemMatch Operator erwartet ein
5 // Abfrageobjekt
6 db.projects.find({
7     fundings : {
8         $elemMatch : {
9             $and : [
10                 {debtorName : "TU Wien"},
11                 {amount : NumberLong(1500)}
12             ]
13         }
14     }
15 });
16
17 db.projects.find({
18     projectStaff : {
19         $elemMatch : {
20             $and : [
21                 {$in : ["MANAGER"]}, //role
22                 {lastName : "Gruber"} //lastName
23             ]
24         }
25     }
26 });

```

16.5. Projektion

Sind für eine Abfrage nicht alle **Attribute** eines Dokuments interessant, wird durch die Angabe einer **Projektion** eine entsprechende Einschränkung definiert.

16.5.1 Projektionsobjekt

Projektionen werden durch die Angabe eines **Projektionsobjekts** definiert.

► Erklärung: Projektionsobjekt ▾

- Die Attribute des Projektionsobjekts definieren die Felder der Projektion. Attributen, die in das Ergebnis aufgenommen werden sollen, wird 1 als Wert zugeordnet.
- Das **_id Attribut** eines Dokuments ist defaultmäßig Teil der Projektion.

► Codebeispiel: Projektion ▾

```

1 //-----
2 //  Beispiel: find
3 //-----
4 var crsr = db.inventory.find(
5     { status: "A",
6         $or: [
7             { qty: { $lt: 30 } },
8             { item: p }
9         ]
10    },
11    { _id : 0 , status : 1, item : 1}
12 );
13
14 while (crsr.hasNext()){
15     printjson( crsr.next() );
16 }

```

17. MongoDB - DML

03

MongoDB - DML

01. Daten einfügen	150
02. Daten bearbeiten	151
03. Strukturoperatoren	153
04. Werteoperatoren	155
05. Arrayoperatoren	156
03. Daten löschen	159

17.1. Daten einfügen - insert

Für das **Einfügen** von Daten stellt die MongoDB Spezifikation mehrere Formen des `insert()` Befehls zur Verfügung.



17.1.1 insertOne(), insertMany() Befehl

Der `insert` Befehl wird immer auf einer einzelnen Collection ausgeführt.

► Codebeispiel: insertOne() Befehl ▾

```

1 //-----
2 // Syntax: insertOne
3 //-----
4 db.<collection>.insertOne(<document>);
5
6 //-----
7 // insertOne: document without _id
8 //-----
9 try {
10   db.products.insertOne(
11     {item:"card", qty:15}
12   );
13 } catch (e) {
14   print (e);
15 }
16
17 // insertOne: document with _id
18 try {
19   db.products.insertOne( {
20     _id : 10, item : "box"
21   });
22 } catch (e) {
23   print (e);
24 }
```

► Erklärung: Nebeneffekte des Insert Befehls ▾

- Beim **Einfügen von Dokumenten** in eine Collection treten in der Regel mehrere **Nebeneffekte** auf.
- **collection:** Wird die `insert()` Methode auf einer nicht existierende Collection aufgerufen, legt die Datenbankengine die Collection implizit an.
- **ObjectID:** Für Dokumente ohne ein `_id` Attribut, **generiert** die Datenbankengine eine eindeutige ID beim Einfügen des Dokuments.



```
db.<collection>.insertOne(
  {
    name : "sue",
    age : 26,
    status : "pending"
  }
);
```

Abbildung 62. insert document

► Codebeispiel: `insertMany()` Befehl ▾

```
1 //-----
2 // Syntax: insertMany
3 //-----
4 db.<collection>.insertMany([<document>, ...]);
5
6 //-----
7 // insertMany: document without _id
8 //-----
9 try {
10   db.products.insertMany([
11     { item: "card", qty: 15 },
12     { item: "envelope", qty: 20 }
13   ]);
14 } catch (e) {
15   print(e);
16 }
17
18 var returnValue = {
19   "insertedIds" : [
20     ObjectId("562a94d381cb9f1cd6eb0e1a"),
21     ObjectId("562a94d381cb9f1cd6eb0e1b")
22   ]
23 };
24
25 //-----
26 // insertMany: document with _id
27 //-----
28 try {
29   db.products.insertMany([
30     { _id: 10, item: "large", qty: 20 },
31     { _id: 11, item: "small", qty: 55 }
32   ]);
33 } catch (e) {
34   print(e);
35 }
```

17.2. Daten bearbeiten - update ▾

Zum Bearbeiten von Dokumenten definiert die MongoDB Spezifikation mehrere Formen des `update()` Befehls.

17.2.1 `updateOne()` Befehl

Für das Bearbeiten eines einzelnen Dokuments, stellt die MongoDB Spezifikation den `updateOne()` Befehl zur Verfügung.

► Codebeispiel: `updateOne()` Befehl ▾

```
1 //-----
2 // Syntax: updateOne
3 //-----
4 db.<collection>.updateOne(
5   <query criteria>, <update operator>
6 );
7
8 try {
9   db.products.updateOne(
10     { "_id" : 1 },
11     { $set: { "sold" : true } }
12   );
13 } catch (e) {
14   print(e);
15 }
```

► Erklärung: `updateOne` Parameter ▾

- **query criteria:** Mit dem Parameter wird definiert, welche Dokumente einer Collection einer Änderung unterzogen werden sollen. Der Parameter erwartet dazu eine **Abfrageobjekt**.
- **update operator:** Der Parameter definiert die Art der Änderung. Die MongoDB Spezifikation definiert dazu 14 unterschiedliche **Operatoren**.

Operator	Beschreibung	Seite
\$set	Mit dem \$set Operator werden ein oder mehrere Felder auf einen konstanten, neuen Wert gesetzt. Felder, die noch nicht existieren, werden angelegt.	153
\$unset	Zum Löschen von Feldern wird der \$unset Operator verwendet. Die Namen der zu löschen- den Felder werden als Schlüssel mit Wert 1 angegeben.	154
\$rename	Eine Umbenennung eines oder mehrerer Felder wird mit dem \$rename Operator durchgeführt. Der Schlüssel des jeweiligen Feldes im Änderungsdokument ist der Name des umzubenen- nenden Feldes, der Wert der neue Name.	154
\$inc	Mithilfe des \$inc Operators lassen sich numerische Felder um einen bestimmten, festen Betrag erhöhen bzw. erniedrigen (bei negativen Werten). Werden Felder angegeben, die es zuvor nicht gab, werden diese auf den entsprechenden Wert gesetzt.	156
\$mul	Der \$mul Operator multipliziert ein numerisches Feld mit dem angegebenen Faktor.	155
\$currentDate	Der \$currentDate Operator wird verwendet um Datumsfelder zu setzen.	tba
\$max	Only updates the field if the specified value is greater than the existing field value.	155
\$min	Only updates the field if the specified value is greater than the existing field value.	155
\$setOrInsert	Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.	tba
\$push	Der \$push Operator hängt an ein Array, ein weiteres Element an falls das Feld noch nicht existiert, wird es angelegt.	157
\$addToSet	Analog zu \$push fügt \$addToSet einem Array einen oder mehrere Werte hinzu, allerdings nur dann, wenn der jeweilige Wert noch nicht in dem Array enthalten ist.	156
\$pop	Der \$pop Operator ist das Gegenstück zum \$push bzw. \$addToSet Operator. Mit ihm wird das letzte bzw. erste Element eines Array gelöscht. Zum Löschen des letzten Elements ist eine 1 zu übergeben, -1 löscht hingegen das erste Element.	158
\$pull	Removes all array elements that match a specified query.	158
\$pullAll	Der \$pullAll Operator hängt an ein Array, ein weitere Elemente an falls das Feld noch nicht existiert, wird es angelegt.	158
\$each	Modifies the \$push and \$addToSet operators to append multiple items for array updates.	??
\$position	Modifies the \$push operator to specify the position in the array to add elements.	??
\$slice	Modifies the \$push operator to limit the size of updated arrays.	??
\$sort	Modifies the \$push operator to reorder documents stored in an array.	??

Abbildung 63. update Operatoren

17.2.2 updateMany Befehl

Die `updateMany` Methode initialisiert eine Änderung mehrerer **Dokumente**.

► Codebeispiel: `updateMany`

```

1 //-----
2 // Syntax: updateMany
3 //-----
4 db.<collection>.updateMany(
5   <query criteria>,
6   <update>,
7 );
8
9 try {
10   db.restaurant.updateMany(
11     { violations: { $gt: 4 } },
12     { $set: { "review" : true } }
13   );
14 } catch (e) {
15   print(e);
16 }
```

► Codebeispiel: `updateMany`

```

1 //-----
2 // Beispiel: updateMany
3 //-----
4 var restaurant2 = {
5   _id      : 2,
6   name    : "Rock A Feller Bar",
7   violations : 2
8 };
9
10 var restaurant3 = {
11   _id      : 3,
12   name    : "Empire State Sub",
13   violations : 5
14 };
15
16 db.restaurant.updateMany(
17   { violations: { $gt: 4 } },
18   { $set: { "review" : true } }
19 );
20
21 var restaurant3 = {
22   _id      : 3,
23   name    : "Empire State Sub",
24   violations : 5,
25   review   : true
26 };
```

17.3. Strukturoperatoren

Mit Strukturoperatoren kann die **Struktur** von Dokumenten geändert werden.

17.3.1 \$set Operator

Der `$set` Operator wird verwendet um den Wert eines Attributes zu ändern.

► Erklärung: `$set` Operator

- Mit dem `$set` Operator werden ein oder mehrere Attribute auf einen neuen, konstanten Wert gesetzt.
- Existiert eines der angegebenen Felder nicht, wird es zusammen mit dem neuen Wert im Dokument angelegt.

► Codebeispiel: `$set` Operator

```

1 //-----
2 // update Operator: $set
3 //-----
4 db.products.insertMany([
5   {
6     _id      : 100,
7     rating  : 4,
8     tags    : [ "apparel", "clothing" ]
9   },
10  {
11    _id      : 101,
12    rating  : 6,
13    tags    : [ "apparel", "clothing" ]
14  }
15]);
16
17 db.products.updateOne(
18   { _id: 100 },
19   {
20     $set : {
21       quantity: 500,
22       tags: [
23         "coats", "outerwear", "clothing"
24       ]
25     }
26   }
27 );
28
29 var product = {
30   _id      : 100,
31   quantity : 500,
32   rating   : 4,
33   tags     : [
34     "coats", "outerwear", "clothing"
35   ]
36 };
```

17.3.2 \$unset Operator



\$unset Operator ▾

Der \$unset Operator wird zum **Löschen** von Attributen verwendet.

► Erklärung: \$unset Operator ▾

- Zu löschende Felder werden als Attribute des **Updateteobjekts** definiert.
- Attribute die nicht im Zieldokument definiert sind, werden ignoriert.

► Codebeispiel: \$unset Operator ▾

```

1 //-----
2 // update Operator: $unset
3 //-----
4 var product = {
5     _id      : 100,
6     sku      : "abc123",
7     quantity : 250,
8     reorder  : false,
9     details  : {
10         model: "14Q2", make: "xyz"
11     },
12     tags    : [ "apparel", "clothing" ],
13     ratings : [
14         { by: "ijk", rating: 4 }
15     ]
16 };
17
18 db.products.updateOne(
19     { _id: 100 },
20     { $unset: {
21         ratings : "",
22         reorder : ""
23     } }
24 );
25
26 var product = {
27     _id      : 100,
28     sku      : "abc123",
29     quantity : 500,
30     details  : {
31         model: "14Q2", make: "xyz"
32     },
33     tags    : [ "apparel", "clothing" ]
34 }
```


17.3.3 \$rename Operator

\$rename Operator ▾

Der \$rename Operator wird zum **Umbenennen** von Attributnamen in Dokumenten verwendet.

► Erklärung: \$rename Operator ▾

- Das **Renameobjekt** enthält dazu den neuen und alten Namen des Attributes in Form eines Schlüssel/Wertepaars.

► Codebeispiel: \$rename Operator ▾

```

1 //-----
2 // update Operator: $rename
3 //-----
4 db.students.insertMany([
5     {
6         alias : [ "The American Cincinnatus" ],
7         nmae  : {
8             first : "george", last : "was"
9         },
10        alias : [ "My dearest friend" ],
11        nmae  : {
12            first : "abigail", last : "adams"
13        }
14    ]);
15
16 db.student.updateMany{
17     {},
18     {$rename : {nmae : "name"}}
19 };
20
21 > Ausgabe
22 {
23     alias : [ "The American Cincinnatus" ],
24     name  : {
25         first : "george",
26         last  : "was"
27     }
28 }
29 }
30 {
31     alias : [ "My dearest friend" ],
32     name  : {
33         first : "abigail",
34         last  : "adams"
35     }
36 }
```

154

```
db.<collection>.updateMany(
  { },
  {
    $updateOperator : {
      ...
      ...
    }
  }
);
```

← Query Kriterien
← Update Operator

Abbildung 64. update Operatoren

17.4. Werteoperatoren

Werteoperatoren werden zum Bearbeiten numerischer Werte verwendet.

17.4.1 \$mul Operator

Der **\$mul Operator** wird verwendet um ein Feld auf ein Vielfaches seines ursprünglichen Wertes zu setzen.

Codebeispiel: \$mul Operator

```
1 //-----
2 // update Operator: $mul
3 //-----
4 var product1 = {
5   _id : 1,
6   price : NumberDecimal("10.99"),
7   qty : 25
8 };
9
10 db.products.updateOne(
11   { _id: 1 },
12   {
13     $mul: {
14       price: NumberDecimal("1.25"),
15       qty : 2
16     }
17   }
18 );
19
20 var updatedProduct = {
21   _id : 1,
22   price : NumberDecimal("14.7375"),
23   qty : 50
24 };
```

17.4.2 \$min Operator

Mit dem **\$min Operator** wird der Wert eines Feldes auf den kleineren 2er Werte gesetzt.

Codebeispiel: \$min Operator

```
1 //-----
2 // update Operator: $min
3 //-----
4 var result = {
5   _id: 1,
6   highScore: 800,
7   lowScore: 200
8 };
9
10 var result = {
11   _id: 2,
12   highScore: 1800,
13   lowScore: 50
14 };
15
16 db.scores.updateOne(
17   { _id: 1 },
18   { $min: { lowScore: 250 } }
19 );
20
21
22 //-----
23 // Ergebnis: $min
24 //-----
25 var updatedResult = {
26   _id: 1,
27   highScore : 800,
28   lowScore : 200
29 };
```

17.4.3 \$inc Operator



\$inc Operator ▾

Der \$inc Operator wird verwendet um **numerische Werte** zu bearbeiten.

► Erklärung: \$inc Operator ▾

- Mithilfe des \$inc Operators werden **numerische Felder** um einen bestimmten, festen Betrag erhöht bzw. erniedrigt.
- Existiert das angegebene Attribute nicht, wird es in das Dokument eingefügt.

► Codebeispiel: \$inc Operator ▾

```

1 //-----
2 // update Operator: $inc
3 //-----
4 var cart = {
5     _id      : 1,
6     sku      : "abc123",
7     default  : "tower",
8     quantity : 10,
9     type     : "showel",
10    metrics : {
11        orders: 2,
12        ratings: 3.5
13    }
14 };
15
16 db.products.updateMany(
17     { sku: "abc123" },
18     { $inc: {
19         quantity: -2,
20         "metrics.orders": 1
21     }
22 }
23 );
24
25 var updatedCart = {
26     _id      : 1,
27     sku      : "abc123",
28     default  : "tower",
29     type     : "showel",
30     quantity : 8,
31     metrics : {
32         orders : 3,
33         ratings : 3.5
34     }
35 };

```

17.5. Arrayoperatoren

Arrayoperatoren werden zur Verarbeitung von Arraydaten verwendet.

17.5.1 \$addToSet Operator

Der \$addToSet Operator wird verwendet um Werte zu einem Array hinzuzufügen.

► Erklärung: \$addToSet Operator ▾

- Werte die bereits in einem Array vorhanden sind werden nicht wiederholt eingefügt.
- Zum Hinzufügen mehrerer Werte muss der \$addToSet Operator mit dem \$each Operator kombiniert werden.

► Codebeispiel: \$addToSet Operator ▾

```

1 //-----
2 // update array Operator: $addToSet
3 //-----
4 var test = {
5     _id: 1, letters: ["a", "b"]
6 };
7
8 db.test.updateOne(
9     { _id: 1 },
10     { $addToSet: { letters: [ "c", "d" ] } }
11 );
12
13 var updatedTest = {
14     _id: 1,
15     letters: [ "a", "b", [ "c", "d" ] ]
16 };
17
18 var test = { _id: 2, letters: [ "a", "b" ] };
19
20 db.test.updateOne(
21     { _id: 2 },
22     { $addToSet: {
23         letters: { $each : [ "c", "d" ] }
24     }
25 };
26
27 var updatedTest = {
28     _id: 2, letters: [ "a", "b", "c", "d" ]
29 };
30

```

Arrayoperator	Beschreibung	Seite
\$addToSet	Der addToSet Operator wird verwendet um zu einem Array einem oder mehreren Werte hinzuzufügen.	156
\$push	Der push Operator wird verwendet um Elemente zu einem hinzuzufügen. Die hinzuzufügenden Werte unterliegen jedoch keiner Prüfung auf Eindeutigkeit.	157
\$pull, \$pullAll	Mit dem \$pull bzw. \$pullAll Operator werden Werte aus einem Array entfernt.	158
\$pop	Der \$pop Operator wird verwendet um Elemente aus einem Array zu löschen.	158

Abbildung 65. Arrayoperatoren

17.5.2 \$push Operator



Spush Operator ▾

Der \$push Operator wird verwendet um Elemente in ein Array **einzzufügen**. Existiert das Array nicht, wird es angelegt.

Der push Operator definiert eine Reihe von **Operatoren** um sein Verhalten zu adaptieren.

► Erklärung: \$slice, \$sort \$position Modifier ▾

- **\$position Modifier:** Der \$position Modifier definiert an welcher **Position** im Array die Daten eingefügt werden sollen.
- **\$slice Modifier:** Der \$slice Modifier limitiert das Array auf die ersten bzw. letzten n Elemente.
- **\$sort Modifier:** Der \$sort Modifier wird verwendet um die Elemente eines Arrays zu sortieren.

► Codebeispiel: \$push Operator ▾

```

1 //-----
2 // update array Operator: $push
3 //-----
4 db.students.updateOne(
5   { _id: 5 },
6   { $push: {
7     quizzes: {
8       $each: [
9         { wk: 5, score: 8 },
10        { wk: 6, score: 7 }
11      ],
12      $sort: { score: -1 }
13    }
14  }
15 );
16
17 var updatedStudent = {
18   quizzes: [
19     { wk: 1, score: 10 },
20     { wk: 2, score: 8 },
21     { wk: 3, score: 5 }
22   ]
23 };
24
25
26
27
28
29
30
31
32
33 }
```

► Codebeispiel: \$slice, \$sort \$position ▾

```

1 //-----
2 // update array Operator: $push
3 //-----
4 var student = {
5   quizzes: [
6     { wk: 1, score: 10 },
7     { wk: 2, score: 8 },
8     { wk: 3, score: 5 },
9     { wk: 4, score: 6 }
10   ]
11 };
12
13 db.students.updateOne(
14   { _id: 5 },
15   { $push: {
16     quizzes: {
17       $each: [
18         { wk: 5, score: 8 },
19         { wk: 6, score: 7 },
20         { wk: 7, score: 6 }
21       ],
22       $slice : 3
23     }
24   }
25 );
26
27
28
29
30
31
32
33 }
```



17.5.3 \$pull, \$pullAll Operator

► Erklärung: \$pull Operator ▾

- Mit dem \$pull bzw. \$pullAll Operator lassen sich gezielt Werte aus einem Array entfernen.
- \$pull löscht alle Vorkommen eines einzelnen Wertes.
\$pullAll löscht alle Vorkommen mehrerer Werte.

► Erklärung: \$pull Operator ▾

```

1 //-----
2 // update array Operator: $pull
3 //-----
4 var store = { _id: 1,
5   fruits: [ "apples", "oranges", "grapes" ],
6   vegetables: [
7     "carrots", "celery", "carrots"
8   ]
9 };
10
11 var store2 = { _id: 2,
12   fruits: [ "oranges", "bananas", "apples" ],
13   vegetables: [
14     "broccoli", "zucchini", "carrots"
15   ]
16 };
17
18 db.stores.updateMany(
19   { },
20   { $pull: {
21     fruits: {
22       $in: [ "apples", "oranges" ]
23     },
24     vegetables: "carrots"
25   },
26   { multi: true }
27 )
28
29
30 var updatedStore = { _id: 1,
31   fruits: [ "grapes" ],
32   vegetables: [
33     "celery", "carrots"
34   ]
35 };
36
37 var updatedStore2 = { _id: 2,
38   fruits: [ "bananas" ],
39   vegetables: [ "broccoli", "zucchini" ]
40 }
```

17.5.4 \$pop Operator



\$pop Operator ▾

Der \$pop Operator wird verwendet um Elemente aus einem Array zu löschen.

► Erklärung: \$pop Operator ▾

- Der \$pop Operator ist das Gegenstück zum \$push bzw. \$addToSet Operator.
- Der \$pop Operator löscht das letzte bzw. erste Element eines Arrays.
- Zum Löschen des letzten Elements wird 1, des ersten Elements -1 übergeben.

► Codebeispiel: \$pop Operator ▾

```

1 //-----
2 // update array Operator: $pop
3 //-----
4 var student = {
5   _id: 1,
6   scores: [ 8, 9, 10 ]
7 };
8
9 db.students.updateOne(
10   { _id: 1 },
11   { $pop: { scores: -1 } }
12 );
13
14 var updatedStudent = {
15   _id: 1,
16   scores: [ 9, 10 ]
17 };
18
19 //-----
20 // update array Operator: $pop
21 //-----
22 var student = { _id: 1, scores: [ 8, 9, 10 ] };
23
24 db.students.updateOne(
25   { _id: 1 },
26   { $pop: { scores: 1 } }
27 );
28
29 var updatedStudent = {
30   _id: 1,
31   scores: [ 8, 9 ]
32 }
```

17.6. Daten löschen

Befehle zum **Löschen** von Dokumenten aus **Collections**.



17.6.1 delete Befehl

MongoDB unterscheidet mehrere Befehle zum **Löschen** von Dokumenten.

► Codebeispiel: `deleteOne`, `deleteMany` Befehl ▾

```

1 //-----
2 // Syntax: deleteOne
3 //-----
4 db.<collection>.deleteOne(
5   <filter>,
6 );
7
8 //-----
9 // Syntax: deleteMany
10 //-----
11 db.<collection>.deleteMany(
12   <filter>,
13 );
14
15 //-----
16 // Beispiel: deleteOne, deleteMany
17 //-----
18 try {
19   db.orders.deleteOne( {
20     "_id" :
21       ObjectId("563237a41a4d68582c2509da")
22   } );
23 } catch (e) {
24   print(e);
25 }
26
27 try {
28   db.orders.deleteOne( {
29     "expiryts" : {
30       $lt: ISODate("2015-11-01T12:40:15Z")
31     }
32   } );
33 } catch (e) {
34   print(e);
35 }
```



18. MongoDB - Aggregation von Daten

05

Aggregation von Daten

01. Methoden und Verfahren	160
02. Abfragemethoden	161
03. Aggregationsframework	162
04. Dokumentstufen	166
05. Expression	178
06. Kontrolloperatoren	182
07. Arrayoperatoren	184
08. Aggregatoperatoren	188

18.1. Methoden und Verfahren

Zur Abbildung geschäftlicher **Kernprozesse**, müssen Datenbanken in der Lage sein komplexe Abfragen durchzuführen.

Aggregationsalgorithmen kommen immer dann zum Einsatz, wenn Auswertungen über die Dokumente mehrerer Collections hinweg, durchgeführt werden müssen.

18.1.1 Aggregatalgorithmen

Zur Definition komplexer Abfragen stellt die MongoDB Spezifikation 3 **Frameworks** zur Verfügung.

► Auflistung: Aggregationsalgorithmen ▾



Abfragemethoden ▾

Abfragemethoden ermöglichen das Aggregieren von Daten für die Dokumente **einer Collection**.



Aggregatframework ▾

Das Aggregatframework ermöglicht die Formulierung komplexer Abfragen für die Dokumente **mehrerer Collections**.



Map Reduce Algorithmus ▾

Der Map Reduce Algorithmus ist ein Verfahren zur **nebenläufigen Verarbeitung** großer Datens Mengen.

► Erklärung: Aggregatalgorithmen ▾

- **Abfragemethoden:** Abfragemethoden erlauben lediglich das **Aggregieren** von Daten für die Dokumente einer einzelnen Collection. Abfragemethoden sind in ihrem Funktionsumfang damit relativ **eingeschränkt**.
- **Aggregatframework:** Das Aggregatframework ermöglicht komplexe Auswertungen für Dokumente in **mehreren Collections**.
- **Map Reduce Verfahren:** Der Map Reduce Algorithmus ist der flexibelste der 3 Aggregationsalgorithmen. Das Verfahren weist bei einer Implementierung jedoch eine hohe **Komplexität** auf.

Abfragemethoden	Beschreibung	Seite
count	Mit der Hilfe der <code>count</code> Methode wird die Anzahl der Dokumente eines Resultsets ermittelt.	161
distinct	Mit der <code>distinct</code> Methode können alle unterschiedliche Werte eines Attributes in Form eines Array ermittelt werden.	161
group	Die <code>group()</code> Methode gruppiert die Daten einer Collection anhand eines Schlüssels. Für die gruppierten Daten können nun einfache Operationen wie das Zählen oder das Aufsummieren von Werten durchgeführt werden.	161

Abbildung 66. Abfragemethoden

18.2. Abfragemethoden

Abfragemethoden ermöglichen das **Aggregieren von Daten** für die Dokumente einer Collection.

18.2.1 Abfragemethode: count()

Mit Hilfe der `count()` Methode kann die Anzahl der **Dokumente** eines Resultsets ermittelt werden.

► Erklärung: count() Methode ▾

- Mit der `count()` Methode kann auf einfache Weise bestimmt werden, wieviele **Dokumente** im Resultset einer **Abfrage** enthalten sind.

► Codebeispiel: count() Methode ▾

```

1 //-----
2 //  count() Methode
3 //-----
4 db.inventory.insertMany([
5   item : "journal",
6   qty  : 25,
7 }, {
8   item : "notebook",
9   qty  : 50,
10 }
11 ]);
12
13 db.inventory.count();
14 > 2
15
16 db.inventory.find(
17   { gty : { $gte : 20}}
18 ).count();
19 > 1

```

18.2.2 Abfragemethode: distinct()

Mit der `distinct()` Methode können die unterschiedlichen **Ausprägungen** eines **Attributes** für die Dokumente einer Collection bestimmt werden.

► Codebeispiel: distinct() Methode ▾

```

1 //-----
2 //  distinct() Methode
3 //-----
4 db.persons.insertMany([
5   { name : "Andreas", age : 21},
6   { name : "Christian", age : 23}
7 ]);
8
9 db.persons.distinct('name');
10 > ["Andreas", "Christian"]

```

18.2.3 Abfragemethode: group()

Die `group()` Methode **gruppiert** die Daten einer Collection anhand eines Schlüssels. Für die gruppierten Daten können einfache Operationen wie das Zählen bzw. das Aufsummieren von Werten durchgeführt werden.

► Auflistung: group() Parameter ▾

- **key:** Das `key` Attribut bestimmt die Werte nach denen die Dokumente einer Collection **gruppiert** werden sollen.
- **reduce:** Die `reduce` Funktion definiert in welcher Form die Dokumente einer Gruppe verarbeitet werden sollen. Die Funktion wird für jedes Dokument einer Gruppe aufgerufen.

- **initial:** Bevor die `reduce` Funktion durch die MongoDB Engine ausgeführt wird, können in einem **Initialisierungsschritt** Variablen definiert und initialisiert werden.
- **finalize:** Müssen nach dem Ausführen der `reduce` Methode weitere Berechnungen durchgeführt werden, können diese optional in Form einer Funktion definiert werden.

► Codebeispiel: `group()` Methode ▾

```

1 //-----
2 // group() Methode
3 //-----
4 db.persons.insertMany([
5   { name : "Andreas", age : 21},
6   { name : "Alexander", age : 21},
7   { name : "Michael", age : 14},
8   { name : "Alexander", age : 24},
9   { name : "Michael", age : 34}
10 ]);
11
12
13 // Fuer jeden Namen soll ermittelt werden
14 // wieoft er vergeben wurde.
15 db.persons.group({
16   reduce : function (curr, result) {
17     result.count++;
18   },
19   initial : { count : 0 },
20   key : { name : 1 }
21 });
22
23 //-----
24 // Ausgabe: group() Methode
25 //-----
26 [
27   {
28     name : "Andreas",
29     count : 1
30   }, {
31     name : "Michael",
32     count : 2
33   }, {
34     name : "Alexander",
35     count : 2
36   }
37 ]

```

18.3. Aggregatframework

Das Aggregatframework ermöglicht die Formulierung komplexer Abfragen für die Dokumente **mehrerer Collections**

Die Basis des Aggregatframeworks ist die **Aggregationspipeline**.

18.3.1 Aggregationspipeline



Datenstruktur Pipeline ▾

Eine Pipeline ist eine **Datenstruktur** zur Verarbeitung von Daten.

Datenpipelines besitzen in der Regel mehrere Stufen. Je nach **Pipelinstufe** werden die Daten der Pipeline unterschiedlich verarbeitet.

Eine Aggregationspipeline definiert eine **Kette von Operationen**.

► Erklärung: Aggregationspipeline ▾

- Eine **Aggregationspipeline** legt eine Kette von Operationen, zur Verarbeitung der Dokumente einer Collection fest. Jede **Operation** wird dabei durch einen **Operator** definiert.
- Die MongoDB Spezifikation sieht **keine Restriktionen** für die Zusammensetzung einer Pipeline vor. Lediglich die `$out` Stufe muß als letzte Stufe einer Pipeline auftreten.
- In der Pipeline werden die **Dokumente** einer Collection von einer Stufe zur nächsten Stufe weitergereicht und verarbeitet. Die Ausgabe einer Pipelinestufe dient dabei als **Eingabe** der nachfolgenden Stufe.
- Dabei muss ein Dokument nicht gezwungenermaßen an die nächste Pipelinestufe weitergereicht werden. In einer Pipelinestufe können neue Dokumente erzeugt bzw. vorhandene Dokumente aus der Pipeline entfernt werden.
- Innerhalb einer Stufe kann die **Struktur** von Dokumenten verändert bzw. die Daten eines Dokuments **geändert** werden.



PIPELINE

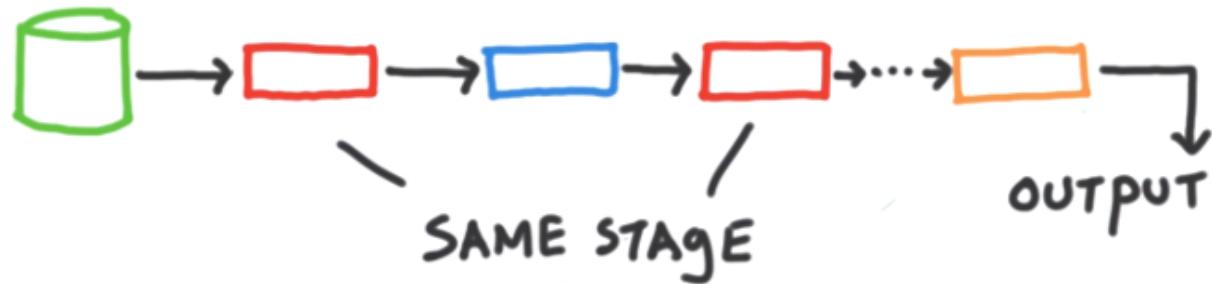


Abbildung 67. Aggregationspipeline

18.3.2 Aggregatframework

Das **Aggregationsframework** basiert auf 2 Grundprinzipien:

► Prinzipien: Aggregatframework ▾



Pipelineverarbeitung ▾

Das Herzstück des Aggregatframework ist die **Aggregationspipeline**.

Die Pipeline legt eine Kette von Operationen, zur sequenziellen Verarbeitung der Dokumente einer Collection fest.



Expressions: ▾

Expressions steuern die Art der **Verarbeitung** innerhalb einer Pipelinestufe.

► Erklärung: Pipelineexpressions ▾

- **Expressions** werden zur **Verarbeitung** von Daten innerhalb einer Pipelinestufe definiert.
- Eine Expression hat in der Regel nur auf die Daten eines einzelnen Dokuments innerhalb einer Pipelinestufe Zugriff.
- Bestimmte Expressions können nur in Kombination mit bestimmten Pipelineoperatoren verwendet werden.

18.3.3 Methode: aggregate()

Durch den Aufruf der `aggregate()` Methode kann die Verarbeitung einer Aggregationspipeline durch die **Datenbankengine** angestoßen werden.

► Syntax: Methode aggregate ▾

```
1 //-----
2 // Syntax: aggregate()
3 //-----
4 db.<collection>.aggregate(<pipeline>);
5
6 //-----
7 // Beispiel: aggregate()
8 //-----
9 db.projects.aggregate([
10   {
11     $match : {
12       $nor : [
13         {type : "RESEARCH_PROJECT"}, 
14         {type : "REQUEST_PROJECT"}
15       ]
16     }
17   },
18   {
19     $project : {
20       type : 1, state : 1,
21       description : 1
22     }
23   },
24   { $sort : { follower_count : -1} }
25 ]);
```

Operator	Beschreibung	Referenz
\$addFields	Adds new fields to documents. Similar to \$project, \$addFields reshapes each document in the stream; specifically, by adding new fields to output documents that contain both the existing fields from the input documents and the newly added fields.	168
\$bucket	Categorizes incoming documents into groups, called buckets, based on a specified expression and bucket boundaries.	177
\$count	Returns a count of the number of documents at this stage of the aggregation pipeline.	168
\$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.	176
\$limit	Passes the first n documents unmodified to the pipeline where n is the specified limit. For each input document, outputs either one document (for the first n documents) or zero documents (after the first n documents).	167
\$lookup	Performs a left outer join to another collection in the same database to filter in documents from the "joined" collection for processing.	172
\$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. \$match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).	166
\$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the \$out stage, it must be the last stage in the pipeline.	167
\$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.	169
\$skip	Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first n documents) or one document (if after the first n documents).	166
\$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.	166
\$unwind	Deconstructs an array field from the input documents to output a document for each element. Each output document replaces the array with an element value. For each input document, outputs n documents where n is the number of array elements and can be zero for an empty array.	167

Abbildung 68. Pipelinestufen

18.3.4 Pipelinestufen

Jede der Pipelinestufe wird durch einen eigenen **Pipelineoperator** definiert.

► Auflistung: Pipelinestufen ▾



match Stufe ▾

Die **match** Stufe wird verwendet um die Dokumente im Dokumentenstrom der Pipeline zu **filtern**.



project Stufe ▾

Die project Stufe der Aggregationspipeline erlaubt die **Manipulation von Dokumenten** durch das Umbenennen, Hinzufügen bzw. Entfernen von Attributen.



limit Stufe ▾

Mit dem **limit** Operator kann die **Anzahl** der Dokumente im Dokumentenstrom der Pipeline limitiert werden.



group Stufe ▾

Mit dem **group** Operator können **Auswertungen** über alle Dokumente einer Pipelinestufe hinweg, durchgeführt werden.



lookup Stufe ▾

Mit dem **lookup** Operator können die Dokumente einer Pipelinestufe mit den Dokumenten anderer Collections in **Relation** gesetzt werden.



unwind Stufe ▾

Mit dem **unwind** Operator können Arrays in Dokumenten zur späteren Verarbeitung aufgelöst werden.



out Stufe ▾

In der **out Stufe** werden die Dokumente der vorhergehenden Pipelinestufe in eine Collection geschrieben.

18.3.5 Fallbeispiel: Projects

Die folgenden Dokumente definieren den Datenbestand der nachfolgenden Abfragen.

► Codebeispiel: projects Dokument ▾

```

1 //-----
2 // Fallbeispiel: projects collection
3 //-----
4 db.projects.insertMany([
5     _id : ObjectId("...1"),
6     title : "Produktionsplanungssysteme",
7     type : "REQUEST_PROJECT",
8     state : "APPROVED",
9     fundings : [
10         {
11             debtorName : "SAP Microsystems",
12             amount : NumberLong(10000)
13         }
14     ],
15     reviews : [ 4,4,3,3,4 ]
16 },{
17     _id : ObjectId("...2"),
18     title : "Finite Elemente",
19     type : "RESEARCH_PROJECT",
20     state : "IN_APPROVEMENT",
21     fundings : [
22         {
23             debtorName : "TU Wien",
24             amount : NumberLong(5000)
25         },
26         {
27             debtorName : "Oracle Systems.",
28             amount : NumberLong(15000)
29         }
30     ],
31     reviews : [ 5,5,3 ]
32 },{
33     _id : ObjectId("...3"),
34     title : "Simulationssysteme",
35     type : "MANAGEMENT_PROJECT",
36     state : "IN_APPROVEMENT",
37     fundings : [
38         {
39             debtorName : "Sun Microsystems",
40             amount : NumberLong(45000)
41         },
42         {
43             debtorName : "Oracle Systems.",
44             amount : NumberLong(15000)
45         }
46     ],
47     reviews : [ 4,5,4,5,5 ]
48 });

```

18.4. Dokumentstufen



Dokumentstufen ▾

Durch die Verwendung von Dokumentstufen wird die **Anzahl der Dokumente** im Dokumentenstrom einer Pipeline gesteuert.

18.4.1 Dokumente filtern - \$match

Die **match** Stufe wird verwendet um **Dokumente** im Dokumentenstrom der Pipeline zu **filtern**.

► Erklärung: \$match Stufe ▾

- Dokumente, die die in der Stufe definierten Filterbedingungen erfüllen, werden unverändert an die nächste Pipelinestufe weitergegeben. Alle anderen Dokumente werden verworfen.
- **Filterbedingungen** werden als **Query Kriterien** definiert.

► Syntax: \$match Operator ▾

```

1 //-----
2 // Syntax: $match Operator
3 //-----
4 db.<collection>.aggregate([
5     { $match : <query criteria>}
6 ]);
7
8 //-----
9 // Beispiel: $match Stufe
10 //-----
11 db.projects.aggregate([
12     { $match : {
13         $nor : [
14             {type: "MANAGEMENT_PROJECT"}, 
15             {type: "REQUEST_PROJECT" },
16         ],
17         state : "IN_APPOINTMENT",
18     }
19 }
20 ]);
21
22 db.projects.aggregate([
23     { $match : {
24         reviews : {$size : 4}
25     }
26 }
27 ]);
28 ]);
```

18.4.2 Dokumente sortieren - \$sort

► Syntax: \$sort Operator ▾

```

1 //-----
2 // Syntax: $sort Operator
3 //-----
4 db.<collection>.aggregate([
5     { $sort : {
6         <sort-key> : <sort-order>
7     }
8 }
9 ]);
10
11 //-----
12 // Beispiel: $sort Stufe
13 //-----
14 db.projects.aggregate([
15     {$match:{type:"REQUEST_PROJECT"}},
16     { $sort :
17         { title : -1 }
18     }
19 ]);
```

18.4.3 Dokumente entfernen - \$skip

Der **\$skip** Operator wird verwendet um Dokumente aus dem Dokumentenstrom der Pipeline zu entfernen.

► Syntax: \$skip Operator ▾

```

1 //-----
2 // Syntax: $skip Operator
3 //-----
4 db.<collection>.aggregate([
5     { $skip : <number_of_documents> }
6 ]);
7
8 //-----
9 // Beispiel: $skip Stufe
10 //-----
11 db.projects.aggregate([
12     { $match : {type : "REQUEST_PROJECT"} },
13     { $skip : 1 }
14 ]);
```

18.4.4 Arrays auflösen - \$unwind

Der **\$unwind** Operator wird verwendet um Arrays strukturell aufzulösen.

Der **\$unwind** Operator ist die Gegenoperation zum **\$group** Operator.

► Erklärung: \$unwind Operator ▾

- Der **\$unwind** Operator wird verwendet um Arrays strukturell aufzulösen.
- Dazu wird für jedes **Arrayelement** eines Datensatzes ein neues Dokument generiert.
- Die verbleibenden Daten des ursprünglichen Dokuments, bleiben dabei von Änderungen unberührt, und werden unverändert in die neu generierten Dokumente übernommen.

► Syntax: \$unwind Operator ▾

```

1 //-----
2 // Syntax: $unwind Operator
3 //-----
4 db.<collection>.aggregate([
5     { $unwind : <fieldname> }
6 ]);
7
8 db.projects.aggregate([
9     { $match : {type:"RESEARCH_PROJECT"}},
10    { $unwind : "reviews" },
11    { $project :
12        { title:1, type: 1, reviews : 1 }
13    },
14    { $out : "projectreport" }
15 ]);
16
17 //-----
18 // Ausgabe: $unwind Stufe
19 //-----
20 {
21     title : "Simulationssysteme",
22     type : "RESEARCH_PROJECT",
23     reviews : 5
24 }, {
25     title : "Simulationssysteme",
26     type : "RESEARCH_PROJECT",
27     reviews : 5
28 }, {
29     title : "Simulationssysteme",
30     type : "RESEARCH_PROJECT",
31     reviews : 3
32 }
```

18.4.5 Dokumentenstrom einschränken - \$limit

Mit dem **\$limit** Operator kann die Zahl der Dokumente im Dokumentenstrom limitiert werden.

► Syntax: \$limit Operator ▾

```

1 //-----
2 // Syntax: $limit Operator
3 //-----
4 db.<collection>.aggregate([
5     { $limit : { <number_of_documents> } }
6 ]);
7
8 //-----
9 // Beispiel: $limit Stufe
10 //-----
11 db.projects.aggregate([
12     { $match : {type : "REQUEST_PROJECT"} },
13     { $limit : 5 }
14 ]);
```



18.4.6 Ergebnis abspeichern - \$out

Der **\$out** Operator finalisiert eine Pipeline und schreibt die Dokumente des Dokumentenstroms in eine Collection.

► Syntax: \$out Operator ▾

```

1 //-----
2 // Syntax: $out Operator
3 //-----
4 db.<collection>.aggregate([
5     { $out : <name_of_collection> }
6 ]);
7
8 //-----
9 // Beispiel: $out Stufe
10 //-----
11 db.projects.aggregate([
12     { $match : {
13         type:"RESEARCH_PROJECT"
14     }
15 }, {
16     $out : "projectreport"
17 }
18 ]);
```

18.5. Strukturstufen



Strukturstufen ▾

Strukturstufen werden verwendet um die **Struktur der Dokumente** im Dokumentenstrom der Pipeline zu ändern.

18.5.1 Felder hinzufügen - \$addFields

Der **\$addFields** Operator wird verwendet um neue Felder zu Dokumenten hinzuzufügen.

Der **\$addFields** Operator ist die Komplementäroperation zum **\$project** Operator.

► Erklärung: \$addFields Operator ▾

- Der **\$addFields** Operator wird verwendet um neue Felder zu Dokumenten hinzuzufügen.
- Die ursprünglichen Daten des Dokuments werden dabei zusätzlich zu den neu hinzugefügten Feldern unverändert übernommen.
- Der **\$addFields** Operator unterstützt eine Reihe von Expression zur Extrahierung von Information aus bestehenden Dokumentdaten.
- Der **\$ Operator** erlaubt dabei den Zugriff auf die Werte der Felder des Dokuments.

► Syntax: \$addFields Operator ▾

```

1 //-----
2 // Syntax: $addFields Operator
3 //-----
4 db.<collection>.aggregate([
5   { $addFields : [
6     { <newField> : <expression> }
7   ]}
8 ]);
9 //-----
10 // Beispiel: Konstante Werte hinzufuegen
11 //-----
12 db.projects.aggregate([
13   { $match : {type: "MANAGEMENT_PROJECT"}},
14   { $addFields : {
15     //Konstante Werte hinzufuegen
16     verified : true,
17     keyword : [ "initialised" ]
18   }}
19 ]);
20 });
21 ]);

```

```

1 //-----
2 // Ergebnis: Konstante Werte hinzufuegen
3 //-----
4 {
5   _id : ObjectId("...3"),
6   title : "Simulationssysteme",
7   type : "MANAGEMENT_PROJECT",
8   state : "IN_APPROVEMENT",
9   fundings : [
10     {
11       debtorName : "Sun Microsystems",
12       amount : NumberLong(45000)
13     },
14     {
15       debtorName : "Oracle Systems.",
16       amount : NumberLong(15000)
17     }
18 ],
19 keyword : ["initialised"],
20 reviews : [ 4,5,4,5,5 ],
21 verified : true
22 }
23 //-----
24 // Beispiel: Variable Werte hinzufuegen
25 //-----
26 db.projects.aggregate([
27   { $match : {type: "MANAGEMENT_PROJECT"}},
28   { $addFields : {
29     //Variable Werte hinzufuegen
30     description : "$title",
31   }}
32 ]);
33 //-----
34 // Ergebnis: Variable Werte hinzufuegen
35 //-----
36 {
37   _id : ObjectId("...3"),
38   title : "Simulationssysteme",
39   type : "MANAGEMENT_PROJECT",
40   state : "IN_APPROVEMENT",
41   fundings : [
42     {
43       debtorName : "Sun Microsystems",
44       amount : NumberLong(45000)
45     },
46     {
47       debtorName : "Oracle Systems.",
48       amount : NumberLong(15000)
49     }
50 ],
51 reviews : [ 4,5,4,5,5 ],
52 description : "Simulationssysteme"
53 }

```

```

1 //-----
2 // Beispiel: Aggregatoperationen
3 //-----
4 db.projects.aggregate([
5   { $addFields : {
6     //Konstante Werte hinzufuegen
7     verified : true,
8     keyword : [ "initialised" ],
9
10    //Aggregatoperatoren auf Arrays
11    voteCount : {
12      $sum : "$reviews"
13    },
14    minVote : {
15      $min : "$reviews"
16    },
17    maxVote : {
18      $max : "$reviews"
19    },
20    groupCount : {
21      $size : "$reviews"
22    }
23  }
24 ]
25 );
26
27 //-----
28 // Ergebnis: Aggregatoperationen
29 //-----
30 {
31   _id : ObjectId("...3"),
32   title : "Simulatinssysteme",
33   type : "MANAGEMENT_PROJECT",
34   state : "IN_APPROVEMENT",
35   fundings : [
36     { debtorName : "Sun Microsystems",
37       amount : NumberLong(45000)
38     },
39     { debtorName : "Oracle Systems.",
40       amount : NumberLong(15000)
41     }
42   ],
43   reviews : [ 4,5,4,5,5 ],
44   keyword : [ "initialised" ],
45   groupCount : 5
46   verified : true,
47   voteCount : 23,
48   minVote : 4,
49   maxVote : 5
50 }

```

```

1 //-----
2 // Ergebnis: Aggregatoperationen
3 //-----
4 {
5   _id : ObjectId("...1"),
6   title : "Produktionsplanungssysteme",
7   type : "REQUEST_PROJECT"
8   state : "APPROVED"
9   fundings : [
10     {
11       debtorName : "SAP Microsystems",
12       amount : NumberLong(10000)
13     },
14     {
15       debtorName : "Oracle Systems",
16       amount : NumberLong(15000)
17     }
18   ],
19   reviews : [
20     4,4,3,3,4
21   ],
22   keyword : [ "initialised" ],
23   groupCount : 5
24   verified : true,
25   voteCount : 18,
26   minVote : 3,
27   maxVote : 4
28 }, {
29   _id : ObjectId("...2"),
30   title : "Finite Elemente im SimuLink",
31   type : "RESEARCH_PROJECT"
32   state : "IN_APROVEMENT"
33   fundings : [
34     {
35       debtorName : "TU Wien",
36       amount : NumberLong(5000)
37     },
38     {
39       debtorName : "Oracle Systems",
40       amount : NumberLong(15000)
41     }
42   ],
43   reviews : [
44     5,5,3
45   ],
46   keyword : [ "initialised" ],
47   groupCount : 3,
48   verified : true,
49   voteCount : 13,
50   minVote : 3,
51   maxVote : 5
52 }

```



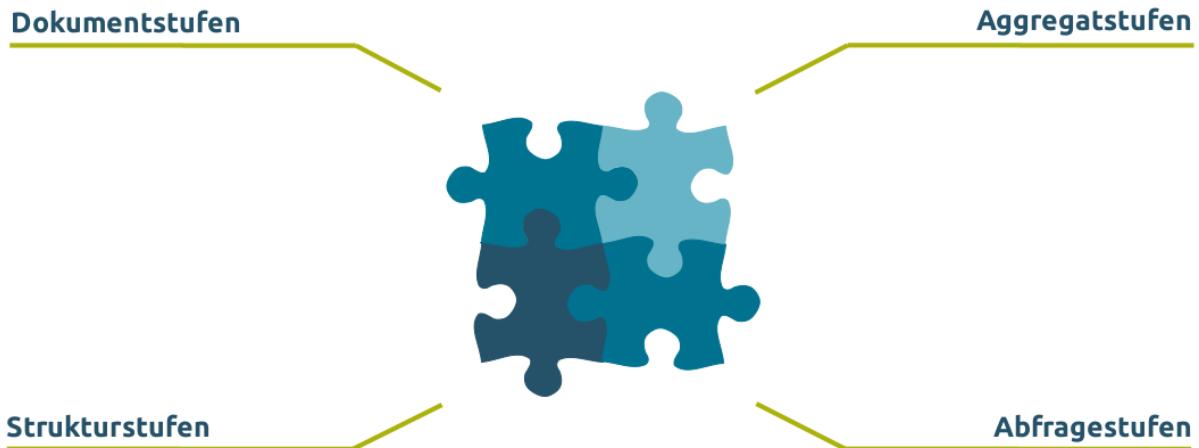


Abbildung 69. Stufentypen

18.5.2 Felder projizieren - \$project

Der **\$project** Operator wird verwendet um die Daten eines Dokuments auf eine **Untermenge seiner Attribute** zu beschränken.

Der **\$project** Operator ist die Komplementaeroperation zum **\$addField** Operator.

► Erklärung: \$project Operator ▾

- Der **\$project** Operator wird verwendet um die Daten eines Dokuments auf eine **Untermenge seiner Attribute** zu beschränken.
- Es werden dabei nur jene Daten eines Dokuments an die nächste Pipelinestufe weitergegeben, die Teil der definierten **Projektion** sind.

► Syntax: \$project Operator ▾

- Dem **\$project** Operator wird ein Objekt als Parameter übergeben.
- Mit dem Objekt wird bestimmt welche Felder eines Dokuments in das Ergebnis der Pipelinestufe übernommen werden sollen.
- Gleichzeitig können neue Attribute zu Dokumenten hinzugefügt werden.
- Dabei erlaubt der **\$** Operator Zugriff auf die Werte der Felder der Eingabedokumente.

► Syntax: \$project Operator ▾

```

1 //-----
2 // Syntax: $project
3 //-----
4 db.<collection>.aggregate([
5   { $project : {<fieldname> : (0|1)> } }
6 ]);
7
8 //-----
9 // Beispiel: $project Stufe
10 //-----
11 db.projects.aggregate([
12   { $match : {type:"MANAGEMENT_PROJECT"} }
13   { $project : {
14     //Felder projizieren
15     title : 1,
16     reviews : 1,
17
18     //Felder umbenennen
19     projectType : "$type",
20     projectState : "$state",
21
22     //Aggregatoperatoren fuer Arrays
23     maxVote : {
24       $max : "$reviews"
25     },
26     minVote : {
27       $min : "$reviews"
28     }
29   } );
30 ]);
```

► Ergebnis: \$project Operator ▾

```
1 //-----
2 // Ergebnis: $project Stufe
3 //-----
4 {
5     title : "Simulationssysteme",
6     projectType : "MANAGEMENT_PROJECT",
7     reviews : [4,5,4,5,5],
8     maxVote : 5,
9     minVote : 4
10 }
```



18.5.3 Dokumentwurzel ändern - \$replaceRoot

Mit dem \$replaceRoot Operator wird die Struktur der Dokumente im Dokumentenstrom der Pipeline verändert.

► Codebeispiel: \$replaceRoot Operator ▾

```
1 //-----
2 // Syntax: $replaceRoot Operator
3 //-----
4 db.<collection>.aggregate([
5     { $replaceRoot : {
6         newRoot : <replacementDocument>
7     }
8 }
9 ]);
10 //-----
11 // Beispiel: $replaceRoot Stufe
12 //-----
13 db.tweets.aggregate([
14     { $match : { lang : "en" } },
15     {
16         $replaceRoot : {
17             newRoot : "$user"
18         }
19     }
20 ]);
21 });
22 > Ausgabe
23 account : {
24     name      : "Jonas Nagelmaier",
25     verified : false,
26     rating   : 5,
27     state    : "initialised"
28 }
```



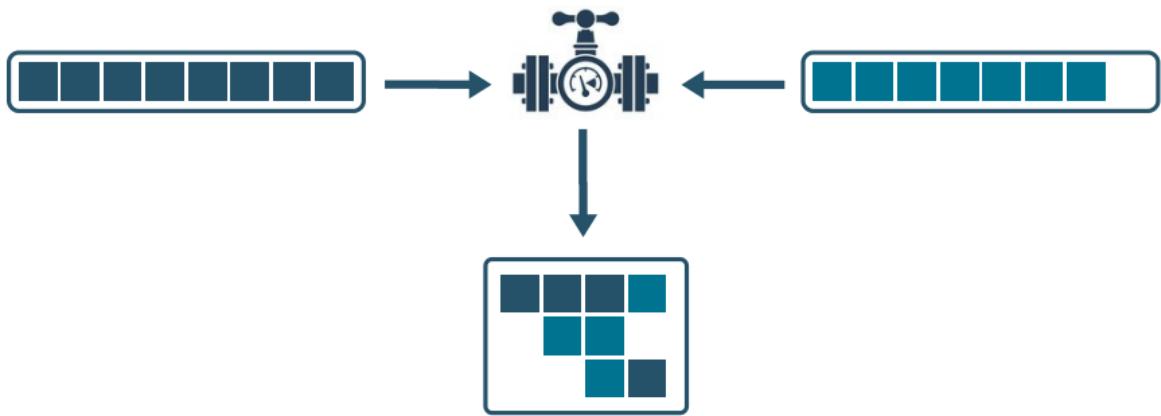


Abbildung 70. Aggregatframework: Beziehungsstufen

18.6. Beziehungsstufen



Beziehungsstufen

Beziehungsstufen werden verwendet um Daten mehrerer **Collections** miteinander in **Relation** zu setzen.

18.6.1 Relationen definieren - \$lookup

Der **\$lookup Operator** wird verwendet um **Relationen** zwischen Dokumenten zu definieren.

Mit dem **\$lookup Operator** können 2 Arten von **Relationen** beschrieben werden.

Erklärung: \$lookup Operator

- Die MongoDB Spezifikationen definiert für den **\$lookup Operators** 2 unterschiedliche Formen.
- Zum einen wird der **\$lookup Operator** verwendet, um einen **left join** zwischen Dokumenten zu definieren.
- Zum anderen können mit dem **\$lookup Operator** 2 Pipelines in Relation gesetzt werden.
- Jede der Formen des **\$lookup Operators** verwendet unterschiedliche Attribute.

Syntax: \$lookup Operator

```

1 //-----
2 // Syntax: $lookup Operator
3 //-----
4 // left join Definition
5 /*
6 * @Param <from>: Der Name der Collection
7 * deren Daten das Ziel der Beziehung sind.
8 *
9 * @Param <localField> : Der Name des Attri-
10 * butes der Joinbedingung der Dokumente
11 * der Basiscollection.
12 *
13 * @Param <foreignField> : Der Name des
14 * Attributes in der zu joinenden Collection.
15 *
16 * @Param <as> : Ein Array das die gejointen
17 * Datensaetze enthaelt.
18 */
20 db.<collection>.aggregate([
21   { $lookup : {
22     from : <collection to join>,
23     localField : <field input document>,
24     foreignField : <extern join field>,
25     as : <output array field>
26   }
27 }
28 ]);
```

Operation	Beschreibung	Referenz
left join	Mit dem \$lookup Operator kann ein left join für die Dokumente 2er Collections definiert werden.	172
Unterabfrage	Der \$lookup Operator kann verwendet werden, um Pipelines miteinander in Relation zu setzen.	174
Hierarchische Abfrage	Mit dem \$Graphlookup Operator können Abfragen auf hierarchischen Strukturen durchgeführt werden.	??

Abbildung 71. Aggregatframework: Beziehungen

► Codebeispiel: subprojects ▾

```

1 //-----
2 // Fallbeispiel : subprojects
3 //-----
4 db.subprojects.insertMany([
5     {
6         _id : ObjectId(...11),
7         appliedResearch : NumberInt(20),
8         focusResearch : NumberInt(80),
9         project_id : ObjectId(...1),
10        title : "ERP SAP",
11        facility : {
12            name : "Softwaretechnikinstitut",
13            _id : ObjectId(...23)
14        },
15        _id : ObjectId(...12),
16        appliedResearch : NumberInt(40),
17        focusResearch : NumberInt(60),
18        project_id : ObjectId(...1),
19        title : "Webbased Systems",
20        facility : {
21            name : "Datenbankeninstitut",
22            _id : ObjectId(...45),
23        }
24    },
25    {
26        _id : ObjectId(...13),
27        appliedResearch : NumberInt(80),
28        focusResearch : NumberInt(10),
29        project_id : ObjectId(...2),
30        title : "Embedded Systems"
31    },
32    {
33        _id : ObjectId(...14),
34        appliedResearch : NumberInt(10),
35        focusResearch : NumberInt(80),
36        project_id : ObjectId(...3),
37        title : "API Design SAP"
38    }
39 ]);

```

► Codebeispiel: \$lookup Operator ▾

```

1 //-----
2 // Beispiel: Relation definieren
3 //-----
4 db.subprojects.aggregate([
5     {
6         $match : {
7             appliedResearch : {
8                 $gt : 10
9             }
10        }
11    },
12    {
13        $lookup : {
14            from : "projects",
15            localField : "project_id",
16            foreignField : "_id",
17            as : "project"
18        }
19    },
20    {
21        $unwind : project
22    },
23    {
24        $project : {
25            appliedResearch : 1,
26            _id : 0, title : 1,
27            focusResearch : 1,
28            project : 1
29        }
30    },
31    {
32        $sort : {
33            title : 1
34        }
35    },
36    {
37        $out : "subprojectReport"
38    }
39 ]);

```

► Codebeispiel: subprojects ▾

```

1 //-----
2 // Ergebnis: $lookup Stufe
3 //-----
4 {
5     _id : ObjectId(...11),
6     appliedResearch : NumberInt(20),
7     focusResearch : NumberInt(80),
8     title : "ERP SAP",
9     project : {
10         id : ObjectId("...1"),
11         title : "Produktionsplanungssysteme",
12         type : "REQUEST_PROJECT",
13         reviews : [
14             4,4,3,3,4
15         ],
16         fundings : [
17             {
18                 debtorName : "..."
19             }
20         },
21         _id : ObjectId(...12),
22         appliedResearch : NumberInt(40),
23         focusResearch : NumberInt(60),
24         title : "Webbased Systems",
25         project : {
26             id : ObjectId("...1"),
27             title : "Produktionsplanungssysteme",
28             type : "REQUEST_PROJECT",
29             reviews : [
30                 4,4,3,3,4
31             ],
32             fundings : [
33                 {
34                     debtorName : "..."
35                 }
36             },
37             _id : ObjectId(...13),
38             appliedResearch : NumberInt(80),
39             focusResearch : NumberInt(10),
40             title : "Embedded Systems"
41             project : {
42                 id : ObjectId("...2"),
43                 title : "Finite Elemente",
44                 type : "RESEARCH_PROJECT",
45                 reviews : [
46                     5, 5, 3
47                 ],
48                 fundings : [
49                     {
50                         debtorName : "..."
51                     }
52                 }
53             }
54         }
55     }
56 }

```

18.6.2 Unterabfrage definieren - \$lookup ▾

Der **\$lookup** Operator kann verwendet werden, um Pipelines miteinander in Relation zu setzen.

► Erklärung: \$lookup Operator ▾

- Mit dem **\$lookup** Operator kann eine Unterabfrage in Form einer Pipeline definiert werden.
- Die Datenströme der Pipelines können dabei mit der Hilfe von Variablen untereinander **Daten austauschen**.
- Der **\$\$ Operator** wird dabei verwendet um in der Unterabfrage auf Variablen zuzugreifen.

► Syntax: \$lookup Operator ▾

```

1 //-----
2 // Syntax: $lookup Operator
3 //-----
4 /*
5 * @Param <from>: Der Parameter bestimmt
6 * fuer welche Collection die Unterabfrage
7 * definiert wird.
8 *
9 * @Param <let>: Fuer den Datenaustausch
10 * koennen im let Attribut Variablen
11 * definiert werden.
12 *
13 * @Param <pipeline>: Der Parameter
14 * referenziert die eigentliche Unterabfrage.
15 *
16 * @Param <as>: Mit dem as Parameter wird
17 * gesteuert in welchem Feld das Ergebnis der
18 * Unterabfrage gespeichert werden soll.
19 */
20
21 db.<collection>.aggregate([
22     { $lookup : {
23         from : <collection to join>,
24         let : {
25             <var_1> : <expression>,
26         },
27         pipeline : [
28             <pipeline to execute on
29             collection>
30         ],
31         as : <output array field>
32     }
33 });

```

► Codebeispiel: \$lookup Operator ▾

```

1  //-----
2  //  Beispiel: $lookup Operator
3  //-----
4  db.debtors.aggregate([
5      {
6          $lookup: {
7              from: "projects",
8              let: {
9                  pid: "$fundings.project_id",
10                 debid: "$_id"
11             },
12             pipeline: [
13                 {
14                     $unwind: "$fundings"
15                 },
16                 {
17                     $match: {
18                         $expr: {
19                             $and: [
20                                 {
21                                     $in: [
22                                         "$_id",
23                                         "$$pid"
24                                     ]
25                                 },
26                                 {
27                                     eq: [
28                                         "$$debitid",
29                                         "$fundings._id"
30                                     ]
31                                 }
32                             ]
33                         }
34                     }
35                 }
36             ],
37             as: "projects"
38         }
39     },
40     {
41         $project: {
42             name: 1,
43             projects: 1
44         }
45     },
46     {
47         $out: "debtorReport2"
48     }
49 ]);

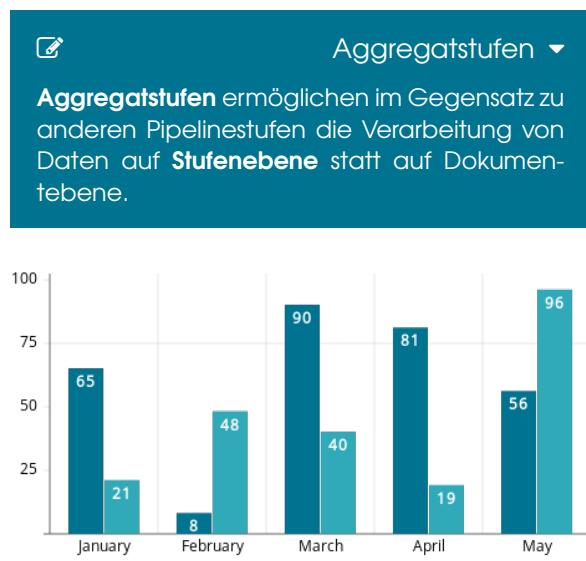
```



Operator	Beschreibung	Referenz
\$group	Der \$group Operator wird verwendet um Dokumente anhand eines Schlüssels zu gruppieren.	176
\$bucket	Der \$bucket Operator wird verwendet um Dokumente anhand eines Wertebereichs zu gruppieren.	177

Abbildung 72. Expressions: Arithmetische Operatoren

18.7. Aggregatstufen



18.7.1 Aggregatoperatoren

Aggregatoperatoren ermöglichen innerhalb einer Aggregatstufe Zugriff auf die Daten **aller Dokumente** der gegenwärtigen Stufe.

► Erklärung: Aggregat- vs. Dokumentstufen ▾

- Der Input einer Pipelinestufe besteht aus den Dokumenten des **Dokumentenstroms** der vorhergehenden Pipelinestufe.
- Die Dokumente des Dokumentenstroms werden getrennt in eine Pipelinestufe eingelesen und verarbeitet. Pipelineoperatoren haben dabei nur auf die Daten eines einzelnen Dokuments Zugriff.
- Aggregatstufen ermöglichen im Vergleich dazu Daten auf **Stufenebene** zu verarbeiten.

18.7.2 Dokumente gruppieren - \$group

Der \$group Operator wird verwendet um Dokumente anhand eines Schlüssels zu **gruppieren**. Die gruppierten Daten können anschließend mit der Hilfe von Aggregatoperatoren verarbeitet werden.

► Syntax: \$group Operator ▾

```

1 //-----
2 // Syntax: $group Operator
3 //-----
4 /* @Param <_id>: Der _id Parameter legt fest
5 * nach welchen Attributen gruppiert werden
6 * soll.
7 *
8 * @Param <aggregated_fieldname>: Zum
9 * Ergebnis der Gruppierung können durch
10 * die Definition von Feldern neue Daten in
11 * hinzugefügt werden.
12 */
13 db.<collection>.aggregate([
14   { $group : {
15     _id : <grouped_by_fieldname>,
16     <aggregated_fieldname> : {expression}
17   }
18 });
19
20 db.projects.aggregate([
21   { $group : {
22     _id : "$type",
23     projectCount : {
24       $sum : 1
25     }
26   }
27 }, {
28   $out : "projectReport"
29 }
30 ]);

```

18.7.3 Fallbeispiel: Gruppierung

Welche Projekte haben die höchste Projektförderung?

► Codebeispiel: Gruppierungsstufen ▾

```

1 //-----
2 // Codebeispiel: Gruppierungsstufen
3 //-----
4 db.projects.aggregate([
5   { $addFields : {
6     projectFunding : {
7       $sum : "$fundings.amount"
8     }
9   },
10  }, {
11    $group : {
12      _id : null,
13      projectFunding : {
14        $max : "$projectFunding"
15      }
16    }
17  },
18  { $lookup : {
19    from : "projects",
20    let : {
21      funds : "$projectFunding"
22    },
23    pipeline : [
24      { $addFields : {
25        funds : {
26          $sum : "fundings.amount"
27        }
28      }
29    },
30    { $match : {
31      $expr : {
32        $eq : [$funds, $$funds]
33      }
34    }
35  },
36  ],
37  as : "projects"
38 },
39  },
40  { $unwind : { path : "$projects" }
41  },
42  { $replaceRoot : {
43    newRoot : "$projects"
44  }
45 });
46 ]);

```

18.7.4 Dokumente gruppieren - \$bucket

Der **\$bucket** Operator wird verwendet um Dokumente anhand eines Werteintervalls zu gruppieren.

Im Gegensatz zum **\$group** Operator wird das **Gruppierungskriterium** für den **\$bucket** Operator in Form eines Intervalls definiert.

► Syntax: \$bucket Operator ▾

```

1 //-----
2 // Syntax: $bucket Operator
3 //-----
4 /*
5  * @Param <groupBy>: Der Parameter definiert
6  * das Kriterium nach dem der Datenbestand
7  * gruppiert werden soll.
8  *
9  * @Param <boundaries>: Mit dem Parameter
10 * werden fuer das Gruppierungskriterien
11 * die Intervallgrenzen definiert.
12 */
13 db.<collection>.aggregate([
14   { $bucket : {
15     groupBy : <expression>,
16     boundaries : [<lowerbound1>, ...],
17     output : {
18       <output1 : {$expression}>,...}
19     }
20   }
21 ]);
22 //-----
23 // Beispiel: $bucket Stufe
24 //-----
25 // Gruppieren Sie Subprojekte entsprechend
26 // ihrem Wert fuer Angewandte Forschung.
27 // appliedResearch Intervalle
28 // 1.Intervall: 0 - 50
29 // 2.Intervall: 51 - 100
30
31 db.subprojects.aggregate([
32   { $bucket : {
33     groupBy : "$appliedResearch",
34     boundaries : [0, 51, 101],
35     output : {
36       count : { $sum : 1},
37       titles : { $push : "$title"}
38     }
39   }
40 });
41 ]);

```



18.8. Expression



Expression ▾

Expressions werden zur **Verarbeitung von Dokumenten** innerhalb einer Pipelinestufe verwendet. Aus formaler Sicht entspricht eine Expression einem **Ausdruck**.

Das **Aggregatframework** basiert auf 2 Grundprinzipien: der Pipelineverarbeitung und den Pipelineexpressions.

► Erklärung: Pipelineexpressions ▾

- Expressions sind **Ausdrücke** zur Verarbeitung von Daten.
- **Expressions** beziehen sich dabei immer auf das aktuell zu verarbeitende Dokument. Es ist nicht möglich, auf Daten anderer Dokumente Bezug zu nehmen.



18.8.1 Expressionsyntax

Für Expressions definiert die MongoDB Spezifikation eine einfache Syntax.

Die Expressionsyntax ist dabei **rekursiv** definiert und ermöglicht dadurch die Formulierung beliebig komplexer Ausdrücke.

► Syntax: Expression ▾

```

1 //-----
2 // Syntax: Expression
3 //-----
4 db.<collection>.aggregate([
5   {
6     <$step> : {
7       <expression>,
8       <expression>
9     }
10   }
11 ]);
12 <expression> = <field path> | <object>
13           | <system variable>
14           | <expression operator>
```

```

1 //-----
2 // Syntax: <Expression operator>
3 //-----
4 <expression> = <expression operator>
5           | <system variable>
6           | <field path> | <object>
7
8
9 <expression operator> = {<operator> : <exp>} |
10   {<operator> = [ <exp>, <exp>, ... ]}
11
12 <operator> = <arithmetic operator>
13           | <array operator>
14           | <boolean operator>
15           | <comparison operator>
16           | <conditional operator>
17           | <accumulators>
18           | <variable operators>
19
20 // Arithmetische Operatoren
21 <arithmetic op.> = $abs | $add | $divide
22           | $multiply
23           | $subtract
24
25 // Arrayoperatoren
26 <array op.> = $arrayElemAt | $map | $in
27           | $concatArrays
28           | $filter
29           | $reduce
30
31
32 // Boolesche Operatoren
33 <boolean op.> = $and | $not | $or
34
35 // Vergleichsoperatoren
36 <comparison op.> = $cmp | $eq | $gt | $lt
37           | $ne
38           | $gte
39           | $lte
40
41 // Konditionale Operatoren
42 <conditional op.> = $cond | $ifNull
43           | $switch
44
45 // Aggregatoperatoren
46 <accumulator> = $addToSet | $avg | $max
47           | $mergeObjects
48           | $min
49           | $push
50           | $sum
```

Operator	Beschreibung	Referenz
\$abs	Der \$abs Operator wird zur Berechnung des Betrags eines numerischen Wertes verwendet.	181
\$add, \$subtract	Der \$add Operator wird verwendet um die Summe 2er oder mehrerer Zahlen zu berechnen.	180
\$addToSet, \$push	Der \$addToSet bzw. \$push Operator ermöglicht die gruppierten Dokumentdaten in einem Array zu verdichten.	188
\$and, \$or, \$not	Boolsche Operatoren werden verwendet um Bedingungen miteinander zu verknüpfen.	182
\$arrayElemAt	Der \$arrayElemAt Operator extrahiert das Arrayelement mit einem bestimmten Index aus einem Array.	184
\$cmp	Mit der Hilfe des \$cmp Operators können 2 Werte miteinander verglichen werden. Das Ergebnis des Vergleichs ist ein Zahlenwert.	184
\$concatArrays	Mit dem \$concatArrays Operator können Elemente aus unterschiedlichen Arrays in ein Zielarray übernommen werden.	185
\$cond	Der \$cond Operator ermöglicht die kontextbasierte Verarbeitung von Daten in Dokumenten.	183
\$divide	Der \$divide Operator wird verwendet um die Division 2er Zahlen zu berechnen.	181
\$eq, \$ne	Vergleichsoperatoren sind zweistellige logische Operatoren. Vergleichsoperatoren werden vor allem in mathematischen Gleichungen verwendet.	181
\$filter	Mit dem \$filter Operator können die Elemente eines Arrays nach bestimmten Kriterien gefiltert werden.	186
\$gt, \$gte, \$lt, \$lte	Vergleichsoperatoren sind zweistellige logische Operatoren. Vergleichsoperatoren werden vor allem in mathematischen Gleichungen verwendet.	181
\$in	Mit dem \$in Operator kann geprüft werden ob ein Element in einem Array enthalten ist.	185
\$map	Der \$map Operator wird verwendet um die Elemente eines Arrays zu verarbeiten.	186
\$max, \$min, \$avg	Mit dem \$min bzw. \$max Operatoren wird für einen bestimmtes Attribut der maximale bzw. minimale Wert auf Collectionebene bzw. Arrayebene ermittelt.	189
\$multiply	Der \$multiply Operator wird verwendet um das Produkt 2er oder mehrerer Zahlen zu berechnen.	180
\$reduce	Mit dem \$reduce Operator können die Elemente eines Arrays auf einen einzelnen Wert verdichtet werden.	187
\$switch	Der \$switch Operator ermöglicht die kontextbasierte Verarbeitung von Daten in Dokumenten.	183

Abbildung 73. Expression: Operatoren

Operator	Beschreibung	Referenz
\$abs	Der \$abs Operator wird zur Berechnung des Betrags eines numerischen Wertes verwendet.	181
\$add, \$subtract	Der \$add Operator wird verwendet um die Summe 2er oder mehrerer Zahlen zu berechnen.	180
\$multiply	Der \$multiply Operator wird verwendet um das Produkt 2er oder mehrerer Zahlen zu berechnen.	180

Abbildung 74. Expressions: Arithmetische Operatoren

18.9. Arithmetische Operatoren ▾



Arithmetische Operatoren ▾

Arithmetische Operatoren verwenden **numerische Werte** als Operanden und geben einen einzelnen numerischen Wert zurück.

18.9.1 \$add, \$subtract, \$multiply Operator

Der \$add Operator bzw. \$subtract wird zur Berechnung der **Addition** bzw. **Subtraktion** 2er oder mehrer Werte verwendet.

► Syntax: \$arithmetische Operatoren ▾

```

1 //-----
2 // Syntax: Arithmetische Operatoren
3 //-----
4 <arithmetic operator> : [
5   <expression1>, <expression2>, ...
6 ]
7
8 db.subprojects.aggregate([
9   { $project : {
10     researchFocus : {
11       $add : [
12         "$theoreticalResearch",
13         "$focusResearch",
14         "$appliedResearch"
15       ]
16     }
17   },
18   { $out : subprojectReport
19   }
20 ]);
21 
```

► Codebeispiel: Arithmetische Operatoren ▾

```

1 //-----
2 // Beispiel: komplexe Ausdruecke
3 //-----
4 value : {
5   $multiply : [ { $add : [3,3] }, 10 ]
6 }
7
8 value : {
9   $subtract : [ "$projectFunding", 10 ]
10 }
11
12 value : {
13   $divide : [
14     { $add : [4, 4] },
15     { $multiply : [2, 3]}
16   ]
17 }
18 //-----
19 // Beispiel: komplexe Ausdruecke
20 //-----
21 db.projects.aggregate([
22   {
23     $addFields : {
24       projectFunding : {
25         $multiply : [
26           {$add : "$projectFunding",10},
27           3
28         ]
29       }
30     }
31   },
32   { $out : projectReport
33   }
34 ]); 
```

Operator	Beschreibung	Referenz
\$eq, \$ne	Vergleichsoperatoren sind zweistellige logische Operatoren. Vergleichsoperatoren werden vor allem in mathematischen Gleichungen verwendet.	181
\$gt, \$gte, \$lt, \$lte	Vergleichsoperatoren sind zweistellige logische Operatoren. Vergleichsoperatoren werden vor allem in mathematischen Gleichungen verwendet.	181

Abbildung 75. Expression: Vergleichsoperatoren

18.9.2 \$divide Operator

Der \$divide Operator wird zur Berechnung der Division 2er Zahlen verwendet.

► Syntax: \$divide Operator ▾

```

1 //-----
2 // Syntax: $divide Operator
3 //-----
4 $divide : [
5   <expression1>, <expression2>
6 ]

```

18.10. Vergleichsoperatoren



Vergleichsoperatoren ▾

Vergleichsoperatoren sind **logische Operatoren**. Vergleichsoperatoren werden vor allem in mathematischen Gleichungen verwendet.

18.9.3 \$abs Operator

Der \$abs Operator wird zur Berechnung des **Betrags** eines numerischen Wertes verwendet.

► Syntax: \$abs Operator ▾

```

1 //-----
2 // Syntax: $abs Operator
3 //-----
4 $abs : <expression>
5
6 db.projects.aggregate([
7   {
8     $project : {
9       projectfunding : {
10         $abs : "$projectFunding"
11       },
12       title : 1
13     }
14   }, {
15     $out : projectReport
16   }
17 ]);

```

18.10.1 Vergleichsoperatoren

Die \$lt bzw \$gt sind zweistellige logische Operatoren.

► Syntax: Vergleichsoperator ▾

```

1 //-----
2 // Syntax: $lt, $gt, $lte, $gte, $eq, $ne
3 //-----
4 <comparison operator> : [
5   <expression1>, <expression2>
6 ]
7
8 db.projects.aggregate([
9   { $match : {
10     $and : [
11       { $eq : [
12         "$type",
13         "REQUEST_FUNDING_PROJECT"
14       ]
15     }, {
16       $lt : [
17         "$projectFunding", 5000
18       ]
19     }
20   }
21 }, {
22   $out : projectReport
23 }]
24 ]);

```

18.11. Boolsche Operatoren



Boolsche Operatoren ▾

Boolsche Operatoren werden verwendet um Bedingungen miteinander zu **verknüpfen**.

18.11.1 \$and, \$or Operatoren

Die \$and und \$or Operatoren werden verwendet um Bedingungen miteinander zu verknüpfen.

► Syntax: \$and, \$or Operator ▾

```

1 //-----
2 // Syntax: $and, $or
3 //-----
4 <boolean op> : [
5     <expression1>, <expression2>, ...
6 ]
7
8 //-----
9 // Beispiel: $and, $or
10 //-----
11 db.projects.aggregate([
12     {
13         $addFields : {
14             projectFunding : {
15                 $sum : "$fundings.amount"
16             }
17         }
18     }, {
19         $match : {
20             $expr : {
21                 $and : [
22                     $eq: [
23                         "$projectFunding", 5000
24                     ]
25                 ], {
26                     $eq : [
27                         "$projectState", "END"
28                     ]
29                 }
30             ]
31         }
32     }, {
33         $out : projectReport
34     }
35 ]);

```

18.11.2 \$not Operator

► Syntax: \$not Operator ▾

```

1 //-----
2 // Syntax: $not
3 //-----
4 $not : [ <expression> ]
5
6 //-----
7 // Beispiel: $and, $or
8 //-----
9 db.projects.aggregate([
10     {
11         $addFields : {
12             verified : { $not : ["$created"] }
13         }
14     }
15 ]);

```

18.12. Kontrolloperatoren



Kontrolloperatoren ▾

Kontrolloperatoren ermöglichen eine **kontextbasierte Verarbeitung** der Daten eines Dokuments.

18.12.1 \$cond Operator

Der \$cond Operator ermöglicht die kontextbasierte Verarbeitung von Daten in Dokumenten.

► Syntax: \$cond Expression ▾

```

1 //-----
2 // Syntax: $cond
3 //-----
4 $cond : {
5     if : <boolean-expression>,
6     then : <true-case>,
7     else : <false-case>
8 }
9
10 $cond : [
11     <boolean-expression>,
12     <true-case>, <false-case>
13 ]

```

Operator	Beschreibung	Referenz
\$cmp	Mit der Hilfe des \$cmp Operators können 2 Werte miteinander verglichen werden. Das Ergebnis des Vergleichs ist ein Zahlenwert.	184
\$cond	Der \$cond Operator ermöglicht die kontextbasierte Verarbeitung von Daten in Dokumenten.	183
\$switch	Der \$cond Operator ermöglicht die kontextbasierte Verarbeitung von Daten in Dokumenten.	183

Abbildung 76. Expression: Kontrollooperatoren

▶ Codebeispiel: \$cond Expression ▾

```

1 //-----
2 // Expression: $cond
3 //-----
4 db.projects.aggregate([
5   { $addField : {
6     verified : {
7       $cond : {
8         if : {
9           $eq : ["$state", "APP"]
10        },
11        then : true,
12        else : false
13      }
14    }
15  }
16]
17]);
18
19 // Kurzform - $cond Operator
20 db.projects.aggregate([
21   { $addField : {
22     verified : {
23       $cond : [
24         "$eq : ['$state', 'APP']",
25         true, false
26       ]
27     }
28   }
29 }, {
30   $project : {
31     title : 1, type : 1, state : 1
32   }
33 }
34]);

```

18.12.2 \$switch Operator

Der \$switch Operator ermöglicht die bedingte Verarbeitung der Daten eines Dokuments.

▶ Syntax: \$switch Expression ▾

```

1 //-----
2 // Syntax: $switch
3 //-----
4 $switch : {
5   branches : [
6     {case : <exp>, then: <exp>},
7     {case : <exp>, then: <exp>},
8     ...
9   ],
10  default : <exp>
11 }
12
13 db.projects.aggregate([
14   $addFields : {
15     projectValue : {
16       $switch : {
17         branches : [
18           {case : {
19             $eq : [
20               "$type",
21               "REQUEST_PROJECT"
22             ]
23           },
24             then : 10
25           }
26         ],
27         default : 0
28       }
29     }
30   });

```

Operator	Beschreibung	Referenz
\$concatArrays	Mit dem \$concatArrays können Elemente aus unterschiedlichen Arrays in ein Zielarray übernommen werden.	184
\$in	Mit dem \$in Operator kann geprüft werden ob ein Element in einem Array enthalten sind.	185
\$map	Der \$map Operator wird verwendet um die Elemente eines Arrays zu verarbeiten.	186
\$filter	Mit dem \$filter Operator können die Elemente eines Arrays nach bestimmten Kriterien gefiltert werden.	186

Abbildung 77. Expressions: Arrayexpressions

18.12.3 \$cmp Operator

Mit der Hilfe des \$cmp Operators können 2 Werte miteinander verglichen werden. Das Ergebnis des Vergleichs ist ein Zahlenwert.

► Erklärung: \$cmp Operator ▾

- Ist der erste Wert größer als der 2te ist das Ergebnis des Vergleichs 1.
- Ist der erste Wert kleiner dann ist das Ergebnis des Vergleichs -1.
- Sind die Werte gleich ist das Ergebnis 0.

► Codebeispiel: \$cmp Expression ▾

```

1 //-----
2 // Beispiel: $cmp
3 //-----
4 db.tweets.insertMany([
5   { _id : 1, item : "abc1", qty: 300 },
6   { _id : 2, item : "abc2", qty: 200 }
7 ]);
8
9 //-----
10 // Expression: $cmp
11 //-----
12 db.tweets.aggregate([
13   {
14     $project : {
15       item : 1, _id : 0,
16       cmpTo250 : {
17         $cmp : [ "$qty", 250 ]
18       }
19     }
20   },
21 ]);
```

18.13. Arrayexpressions



Arrayoperatoren ▾

Arrayoperatoren werden zur Verarbeitung von **Arrays** verwendet.

18.13.1 \$arrayElemAt Operator

Der \$arrayElemAt Operator extrahiert das Arrayelement mit einem bestimmten Index aus einem Array.

► Syntax: \$arrayElemAt ▾

```

1 //-----
2 // Syntax: $arrayElemAt
3 //-----
4 /*
5  * @Param idx: Der Index des gewuenschten
6  * Elements. Fuer negative Werte wird
7  * das Array vom letzten Element weg
8  * durchlaufen.
9 */
10 $arrayElemAt : [
11   <array>, <idx>
12 ]
13
14 db.projects.aggregate([
15   { $project : {
16     _id: 0, title : 1,
17     funding : {
18       $arrayElemAt : ["$fundings", 0]
19     }
20   }
21 ]);
```

18.13.2 \$concatArrays Operator

Mit dem `$concatArrays` Operator können Elemente aus unterschiedlichen Arrays in ein Zielarray übernommen werden.

► Syntax: `$concatArrays` ▾

```

1 //-----
2 // Syntax: $concatArrays
3 //-----
4 $concatArrays : [ <array1> , <array2> ]
5
6 //-----
7 // Beispiel: $concatArrays
8 //-----
9 db.warehouse.insertMany([
10   {
11     instock : ["chocolate"],
12     ordered : ["butter", "apples"]
13   }, {
14     instock : ["apples", "pudding", "pie"]
15   }, {
16     instock : ["ice cream"],
17     ordered : []
18   },
19 ])
20
21 db.warehouse.aggregate([
22   { $project : {
23     items : {
24       $concatArrays : [
25         "$instock", "$ordered"
26       ]
27     }
28   },
29   { $out : "warehouseReport"
30   }
31 ]);
32 );
33
34 > Ausgabe
35
36 {
37   items : [
38     "chocolate", "butter", "apples"
39   ],
40   {
41     items : null
42   },
43   {
44     items : [ "ice cream" ]
45   }

```

18.13.3 \$in Operator

Mit dem `$in` Operator kann geprüft werden ob ein Element in einem Array enthalten ist.

► Syntax: `$in Expression` ▾

```

1 //-----
2 // Syntax: $in Operator
3 //-----
4 $in: [ <expression>, <array expression> ]
5
6 //-----
7 // Beispiel : $in Operator
8 //-----
9 db.warehouse.insertMany([
10   {
11     location : "24th Street",
12     in_stock : [ "apples", "oranges" ]
13   },
14   {
15     location : "36th Street",
16     in_stock : [ "bananas", "pears" ]
17   },
18   {
19     location : "82nd Street",
20     in_stock : [ "apples" ]
21   }
22 ])
23
24 db.warehouse.aggregate([
25   { $project : {
26     storeLocation : "$location",
27     bananaInStock : {
28       $in : [
29         "bananas", "$in_stock"
30       ]
31     }
32   },
33   {
34     out : "warehouseReport"
35   }
36 ]);
37
38 > Ausgabe
39 {
40   storeLocation : "24th Street",
41   bananaInStock : true
42 },
43 {
44   storeLocation : "36th Street",
45   bananaInStock : true
46 },
47 {
48   storeLocation : "82nd Street",
49   bananaInStock : false
50 }

```

18.13.4 \$map Operator

Der `$map` Operator wird verwendet um die Elemente eines Arrays zu verarbeiten.

Vergleichen Sie den `$map` Operator vom Konzept mit der `foreach` Schleife.

► Syntax: \$map Operator ▾

```

1 //-----
2 // Syntax: $map Operator
3 //-----
4 /*
5  * @Param input: Ein Ausdruck der zu einem
6  *               Array evaluiert.
7  *
8  * @Param as: Optional. A name for the
9  *            variable that represents each
10 *           individual element of the array.
11 *           If no name is specified, the
12 *           variable name defaults to this.
13 *
14 * @Param in: An expression that is applied
15 *            to each element of the input array.
16 */
17 $map: {
18     input: <expression>,
19     as: <string>,
20     in: <expression>
21 }
22
23 //-----
24 // Beispiel: $map Operator
25 //-----
26 db.results.insertMany([
27     { _id : 1, values : [ 5, 6, 7 ] },
28     { _id : 2, values : [ ] },
29     { _id : 3, values : [ 3, 8, 9 ] }
30 ])
31
32 db.results.aggregate([
33     $project : {
34         adjustedValues : {
35             $map : {
36                 input : "$values",
37                 as : "value",
38                 in : { $add : [ "$$value", 2 ] }
39             }
40         }
41     }
42 ]);

```

18.13.5 \$filter Operator

Mit dem `$filter` Operator können die Elemente eines Arrays nach bestimmten Kriterien gefiltert werden.

► Syntax: \$filter Expression ▾

```

1 //-----
2 // Syntax: $filter
3 //-----
4 /*
5  * @Param input: Ein Ausdruck der zu einem
6  *               Array evaluiert.
7  *
8  * @Param as: Optional. A name for the
9  *            variable that represents each
10 *           individual element of the array.
11 *           If no name is specified, the
12 *           variable name defaults to this.
13 *
14 * @Param cond: An expression that resolves
15 *              to a boolean value used to determine
16 *              if an element should be included
17 *              in the output array.
18 */
19 $filter : {
20     input : <array>,
21     as   : <string>,
22     cond : <expression>
23 }
24
25 //-----
26 // Beispiel: $filter
27 //-----
28 db.facilities.aggregate([
29     {
30         $addFields : {
31             $filter : {
32                 input : "$projects",
33                 as   : "project",
34                 cond : {
35                     $ne : [
36                         "$$project.type",
37                         "MANAGEMENT_PROJECT"
38                     ]
39                 }
40             }
41         }
42     },
43     {
44         $out : "facilityReport"
45     }
46 ]);

```

18.13.6 \$reduce Operator

Mit dem **\$reduce** Operator können die Elemente eines Arrays auf einen einzelnen Wert **verdichtet** werden.

► Syntax: \$reduce Operator ▾

```

1 //-----
2 // Syntax: $reduce Operator
3 //-----
4 /*
5 * Param input: Ein Ausdruck der zu einem
6 * Array evaluiert.
7 *
8 * Param initialValue: The initial cumulative
9 * value set before in is applied to
10 * the first element of the input array.
11 *
12 * Param in: A valid expression that $reduce
13 * applies to each element in the
14 * input array in left-to-right order.
15 */
16 $reduce: {
17     input: <array>,
18     initialValue: <expression>,
19     in: <expression>
20 }
21
22 //-----
23 // Beispiel: $reduce Operator
24 //-----
25 {
26     $reduce: {
27         input: [ 1, 2, 3, 4 ],
28         initialValue: { sum: 5, product: 2 },
29         in: {
30             sum: {
31                 $add : [
32                     "$$value.sum", "$$this"
33                 ]
34             },
35             product: {
36                 $multiply: [
37                     "$$value.product", "$$this"
38                 ]
39             }
40         }
41     }
42 }
43 // Ergebnis ... { sum : 15, product : 48 }
```

```

1 //-----
2 // Beispiel: $reduce Operator
3 //-----
4 db.getCollection("projects").aggregate([
5     {
6         $addFields : {
7             value : {
8                 $reduce: {
9                     input: ["a", "b", "c"],
10                    initialValue: "",
11                    in: {
12                        $concat : [
13                            "$$value", "$$this"
14                        ]
15                    }
16                }
17            }
18        ]
19    }
20 ]);
```

18.13.7 \$isArray Operator

Der **\$isArray** Operator wird verwendet um zu prüfen ob es sich bei einem Wert um ein Array handelt.

► Syntax: \$isArray Operator ▾

```

1 //-----
2 // Syntax: $isArray Operator
3 //-----
4 $isArray: <expression>
5
6 //-----
7 // Beispiel: $isArray Operator
8 //-----
9 db.getCollection("projects").aggregate([
10     {
11         $addFields : {
12             items : {
13                 $cond : [
14                     { $isArray : "$reviews"}, []
15                 ]
16             }
17         }
18     },
19     {
20         $out : projectReport
21     }
22 ]);
```

Operator	Beschreibung	Referenz
\$addToSet, \$push	Der \$addToSet bzw. \$push Operator ermöglicht die Dokumentdaten der Gruppen in einem Array zu verdichten.	188
\$max, \$min, \$avg	Mit dem \$min bzw. \$max Operatoren wird für einen bestimmtes Attribut der maximale bzw. minimale Wert auf Collectionebene bzw. Arrayebene ermittelt.	189
\$sum	Der \$cond	183

Abbildung 78. Expression: Operatoren

18.14. Aggregateexpressions



Aggregatoperatoren ▾

Aggregatoperatoren haben im Vergleich zu anderen Expressions Zugriff auf die **Daten aller Dokumente** im Dokumentenstrom der gegenwärtigen Pipelinestufe.

18.14.1 Fallbeispiel: sales

Für die nachfolgenden Abfragen bilden die folgenden Dokumente die Datenbasis.

► Codebeispiel: sales collection ▾

```

1 //-----
2 // Collection: sales
3 //-----
4 db.sales.insertMany([
5     {
6         item : "abc1", price : 10
7         category: "BOARD_GAME"
8     },
9     {
10        item : "jkl", price : 20
11        category : "PC_GAME"
12    },
13    {
14        item : "xyz", price : 5
15        category : "BOOK",
16    },
17    {
18        item : "abc2", price : 10
19        category : "BOOK",
20    },
21    {
22        category : "BOARD_GAME",
23        item : "abc3",
24        price : 10
25    }
26]);

```

18.14.2 \$addToSet, \$push Operator

Der \$addToSet bzw. \$push Operator ermöglicht die gruppierten Dokumentdaten in einem Array zu verdichten.

► Syntax: \$addToSet Expression ▾

```

1 //-----
2 // Syntax: $addToSet
3 //-----
4 $addToSet : <expression>
5
6 db.sales.aggregate([
7     {
8         $group: {
9             _id: "$category",
10            itemsSold: {
11                $addToSet: "$item"
12            }
13        },
14        $out : "salesReport"
15    }
16 ]);
17
18 > Ausgabe
19
20 {
21     _id : "BOARD_GAME",
22     itemsSold : [ "abc1", "abc3" ]
23 }, {
24     _id : "BOOK",
25     itemsSold : [ "xyz", "abc2" ]
26 }, {
27     _id : "PC_GAME",
28     itemsSold : [
29         "jkl"
30     ]
31 }

```

18.14.3 \$min, \$max, \$avg Operatoren

Mit dem \$min bzw. \$max Operatoren wird für einen bestimmtes Attribut der maximale bzw. minimale Wert auf Collectionebene bzw. Arrayebene ermittelt.

► Syntax: \$min, \$max, \$avg Operator ▾

```

1 //-----
2 // Syntax: $min, $max Operatoren
3 //-----
4 {$min: <expression>}
5 {$max: <expression>}
6 {$avg: <expression>}
7 //-----
8 // Beispiel: $min, $max Operatoren
9 //-----
10 db.sales.aggregate([
11   {
12     $group: {
13       _id: "$category",
14       minQuantity: { $min: "$quantity" },
15       maxQuantity: { $max: "$quantity" },
16       avgQuantity: { $avg: "$quantity" }
17     }
18   },
19 }, {
20   $sort : {
21     _id : 1
22   }
23 }, {
24   $out : "salesReport"
25 }
26]);
27
28 > Ausgabe
29
30 { _id : "BOARD_GAME",
31   minQuantity: 2,
32   maxQuantity: 10,
33   avgQuantity: 6
34 }, {
35   _id : "BOOK",
36   minQuantity: 5,
37   maxQuantity: 10,
38   avgQuantity: 7
39 }, {
40   _id : "PC_GAME",
41   minQuantity: 1,
42   maxQuantity: 1,
43   avgQuantity: 1
44 }
```

18.14.4 \$sum Operator

Mit dem \$sum Operator ist es möglich für ein bestimmtes Feld über die Dokumente einer Gruppe hinweg die Summe der Werte zu bilden.

► Codebeispiel: \$sum Operator ▾

```

1 //-----
2 // Syntax: $sum Operatoren
3 //-----
4 { $sum: <expression> }
5 //-----
6 // Beispiel: $sum Operatoren
7 //-----
8 db.products.aggregate([
9   {
10     $match : {
11       state : { $ne:"IN_APPROVEMENT" },
12       type : { $ne:"STIPENDIUM" }
13     }
14   },
15   {
16     $group : {
17       _id : "$type",
18       projectCount : {
19         $sum : 1
20       },
21       projects : {
22         $addToSet : "$title"
23       }
24     }
25   },
26   {
27     $out : "projectReport"
28   }
29 ]);
30
31 > Ausgabe
32 [
33   {
34     _id : "REQUEST_PROJECT"
35     projectCount : 1
36     projects : [
37       "Produktionsplanungssysteme"
38     ],
39   },
39   {
40     _id : "RESEARCH_PROJECT"
41     projectCount : 1
42     projects : [
43       "Finite Elemente"
44     ]
44 }
```

18.15. Referenzausdrücke



Referenzausdrücke ▾

Referenzausdrücke ermöglichen die Verwendung von Variablen im Aggregatframework.

18.15.1 \$\$ Operator - Systemvariablen

Die MongoDB Spezifikation definiert eine Reihe eigener **Systemvariablen**.

Für den Zugriff auf **Variablen** wird der **\$\$ Operator** verwendet.

► Codebeispiel: Systemvariablen ▾

```

1 //-----
2 // Systemvariablen: CURRENT
3 //-----
4 // Die CURRENT Systemvariable verweist in
5 // einer Pipelinestufe auf das gegen-
6 // waertige Dokument
7
8 // Bei der Formulierung eines Fieldpaths
9 // wird defaultmaessig die CURRENT System-
10 // variable Pfaden vorangestellt.
11
12 // Folgende Schreibweisen sind aequivalent:
13 // "$<field>" -> "$$CURRENT.<field>"
14
15 db.projects.aggregate([
16   {
17     $match : {
18       $expr : {
19         $eq : [
20           "$$CURRENT.projectType",
21           "MANAGEMENT_PROJECT"
22         ]
23       }
24     }
25   }, {
26     $sort : {
27       "$$CURRENT.projectType" : -1
28     }
29   }, {
30     $out : "projectReport"
31   }
32 ]);

```

18.15.2 \$expr Operator

Ursprünglich erlaubt die \$match Pipelinestufe keine Verwendung von Expressions.

Durch die Verwendung des \$expr Operators ist es möglich Pipelineausdrücke auch in der \$match Stufe zu verwenden.

► Syntax: \$expr Operator ▾

```

1 //-----
2 // Syntax: $expr
3 //-----
4 $expr : {
5   <expression>
6 }
7
8 //-----
9 // Beispiel: $expr
10 //-----
11 db.projects.aggregate([
12   {
13     $addFields : {
14       projectFunding : {
15         $sum : "$fundings.amount"
16       }
17     }
18   }, {
19     $match : {
20       $expr : {
21         $lt : [
22           "$projectFunding", 5000
23         ]
24       }
25     }
26   }, {
27     $project : {
28       projectFunding : 1,
29       projectType : 1,
30       description : 1,
31       title : 1
32     }
33   }, {
34     $sort : {
35       projectFunding : 1,
36     }
37   }, {
38     $out : projectReport
39   }
40 ]);

```

18.15.3 \$let Operator

Der \$let Operator erlaubt die Definition von Variablen in Pipelinestufen.

► Syntax: \$let Operator ▾

```

1 //-----
2 // Syntax: $let
3 //-----
4 /*
5 * @Param <vars>: Assignment block for the
6 *                 variables accessible in the in
7 *                 expression. The variable assign-
8 *                 ments have no meaning outside
9 *                 the in expression.
10 *
11 * @Param <in>: The expression to evaluate.
12 */
13 $let : {
14     vars : {
15         <var1> : <expression>,
16         <var2> : <expression>,
17         ...
18     },
19     in : <expression>
20 }
21
22 //-----
23 // Beispiel: $let
24 //-----
25 db.projects.aggregate([
26     {
27         $addFields : {
28             projectFunding : {
29                 $let : {
30                     vars : {
31                         amount : {
32                             $sum: "$$..."
33                         }
34                     },
35                     in : {
36                         $multiply : [
37                             "$$amount", 1.2
38                         ]
39                     }
40                 }
41             }
42         }
43     }
44 ]);

```

18.16. Mengenoperatoren



Mengenoperatoren ▾

Mengenoperatoren werden verwendet um **Mengenoperationen** auf **Arrays** durchzuführen.

18.16.1 \$setDifference Operator

Mit dem **\$setDifference** Operator wird die **Differenzmenge** der Elemente zweier Arrays ermittelt.

Die Differenzmenge zweier Mengen enthält alle Elemente, die in der ersten Menge enthalten sind und nicht in der zweiten.

► Syntax: \$setDifference Expression ▾

```

1 //-----
2 // Syntax: $setDifference
3 //-----
4 $setDifference: [
5     <expression1>, <expression2>
6 ]
7
8 //-----
9 // Beispiel: $setDifference
10 //-----
11 $setDifference: [
12     [ "a", "b", "c" ], [ "b", "a" ]
13 ]
14
15 // Ausgabe ... [ "c" ]
16
17
18 $setDifference: [
19     [ "a", "b", "a" ], [ "b", "a" ]
20 ]
21
22 // Ausgabe ... []
23
24
25 $setDifference: [
26     [ "a", "b" ], [ [ "b", "a" ] ]
27 ]
28
29 // Ausgabe ... [ "a", "b" ]

```



Abfragemethoden	Beschreibung	Seite
setDifference	Mit dem \$setDifference Operator wird die Differenzmenge der Elemente zweier Arrays ermittelt.	191
setIntersection	Mit dem \$setIntersection Operator wird die Durchschnittsmenge der Elemente zweier Arrays ermittelt.	192
setUnion	Mit dem \$setUnion Operator wird die Vereinigungsmenge der Elemente zweier Arrays ermittelt.	193
setIsSubset	Mit dem \$setIsSubset Operator wird ermittelt ob zwischen 2 Arrays eine Untermengenrelation besteht.	193

Abbildung 79. Expression: Mengenoperationen

▶ Codebeispiel: \$setDifference Operator ▶

```

1 //-----
2 // Beispiel: $setDifference
3 //-----
4 db.experiments.insertMany(
5   { A : [ "red", "blue", "green" ],
6     B : [ "red", "blue" ]
7   },
8   { A : [ "red", "blue" ], B : [ ] }
9 ), {
10   A : [ ], B : [ "red" ]
11 });
12
13 db.experiments.aggregate([
14   { $project: {
15     diff : {
16       $setDifference: [ "$B", "$A" ]
17     },
18     _id : 0, A : 1, B : 1
19   }
20 }
21 ]);
22
23 { A : [ "red", "blue", "green" ],
24   B : [ "red", "blue" ],
25   diff : [ "green" ]
26 }, {
27   A : [ "red", "blue" ],
28   B : [ ], diff : []
29 }, {
30   A : [ ],
31   B : [ "red" ], diff : [ "red" ]
32 }
```

18.16.2 \$setIntersection Operator ▶

Mit dem \$setIntersection Operator wird die **Durchschnittsmenge** der Elemente zweier Arrays ermittelt.

Der Durchschnitt zweier Mengen ist als diejenige Menge definiert, die alle Elemente enthält, die in beiden Mengen vorhanden sind.

▶ Syntax: \$setIntersection Expression ▶

```

1 //-----
2 // Syntax: $setIntersection
3 //-----
4 $setIntersection: [
5   <expression1>, <expression2>
6 ]
7
8 //-----
9 // Beispiel: $setIntersection
10 //-----
11 $setIntersection: [
12   [ "a", "b", "c" ], [ "b", "a" ]
13 ]
14 // Ausgabe ... [ "a", "b" ]
15
16 db.experiments.aggregate([
17   { $project: {
18     A: 1, B: 1,
19     commonToBoth: {
20       $setIntersection: [ "$A", "$B" ]
21     }
22   }
23 }]);
```

18.16.3 \$setUnion Operator

Mit dem `$setUnion` Operator wird die **Vereinigungsmenge** der Elemente zweier Arrays ermittelt.

Die Vereinigung zweier Mengen ist die Menge, die alle Elemente enthält, die wenigstens in einer der beiden Mengen enthalten ist.

► Syntax: \$setUnion Expression ▾

```

1 //-----
2 // Syntax: $setUnion
3 //-----
4 $setUnion: [
5     <expression1>, <expression2>
6 ]
7
8 //-----
9 // Beispiel: $setUnion
10 //-----
11 $setUnion: [
12     [ "a", "b", "c" ], [ "b", "a" ]
13 ]
14
15 // Ausgabe ... ["a", "b", "c"]
16
17
18 $setUnion: [
19     [ "a", "b", "a" ], []
20 ]
21
22 // Ausgabe ... ["a", "b"]
23
24
25 $setUnion: [
26     [ "a", "b" ], [ [ "b", "a" ] ]
27 ]
28
29 // Ausgabe ... ["a", "b", [ "a", "b" ]]
30
31 //-----
32 // Codebeispiel: $setUnion
33 //-----
34 db.experiments.aggregate([
35     { $project: {
36         union : {
37             $setUnion: [ "$A", "$B" ]
38         }
39     }
40 }
41 ]);
```

18.16.4 \$setIsSubset Operator

Mit dem `$setIsSubset` Operator wird ermittelt ob zwischen 2 Arrays eine Untermengen Relation besteht.

► Codebeispiel: \$setIsSubset Operator ▾

```

1 //-----
2 // Beispiel: $setIsSubset Operator
3 //-----
4 $setIsSubset: [
5     <expression1>, <expression2>
6 ]
7
8 //-----
9 // Beispiel: $setIsSubset
10 //-----
11 $setIsSubset: [
12     [ "a", "b", "c" ],
13     [ "b", "a" ]
14 ]
15
16 // Ausgabe ... false
17
18
19 $setIsSubset: [
20     [ "b", "a" ], [ "a", "b", "c" ]
21 ]
22
23 // Ausgabe ... true
24
25
26 $setIsSubset: [
27     [ "a", "b", "c" ], []
28 ]
29
30 // Ausgabe ... false
31
32 //-----
33 // Codebeispiel: $setIsSubset
34 //-----
35 db.experiments.aggregate([
36     {
37         $project: {
38             A: 1,
39             B: 1,
40             subset: {
41                 $setIsSubset: [ "$A", "$B" ]
42             }
43         }
44     }
45 ]);
```


XML Technologien

August 19, 2020

19. Datenformat - XML

01

XML - Grundlagen

01. XML Grundlagen	196
02. XML Elemente	198
03. XML Attribute	200
04. Wohlgeformtheit	201
05. Namensräume	202
06. Logische Sicht	204

19.1. XML Grundlagen



XML Datenformat ▾

XML ist ein grundlegendes **Datenformat** zur Darstellung hierarchisch strukturierter Daten.

Hierarchisch strukturierte Daten bilden die Welt in Form einer hierarchischen **Baumstruktur** ab.

19.1.1 Einsatzgebiete von XML

XML ist ein **Datenformat** zur Darstellung hierarchisch strukturierter Daten.

► Auflistung: Einsatzgebiete von XML ▾



Datenaustauschformat ▾

XML ist ein **Datenformat** zum Austausch von Daten zwischen Anwendungen und Softwareplattformen.

Neben JSON ist XML das wichtigste Datenaustauschformat für Informationssysteme.



Anwendungskonfiguration ▾

In der **Anwendungskonfiguration** gilt XML als de facto Standard für Informationssysteme.



Auszeichnungssprache XML ▾

XML wurde mit der Motivation entwickelt eine universale **Auszeichnungssprache** für das Internet zu schaffen.

Eine Auszeichnungssprache definiert aus welchen Elementen ein Dokument bestehen kann. Die wohl bekannteste Auszeichnungssprache ist HTML.



19.1.2 Fallbeispiel: XML Dokumente

XML wurde mit der Motivation entwickelt eine universale Auszeichnungssprache für das Internet zu schaffen.

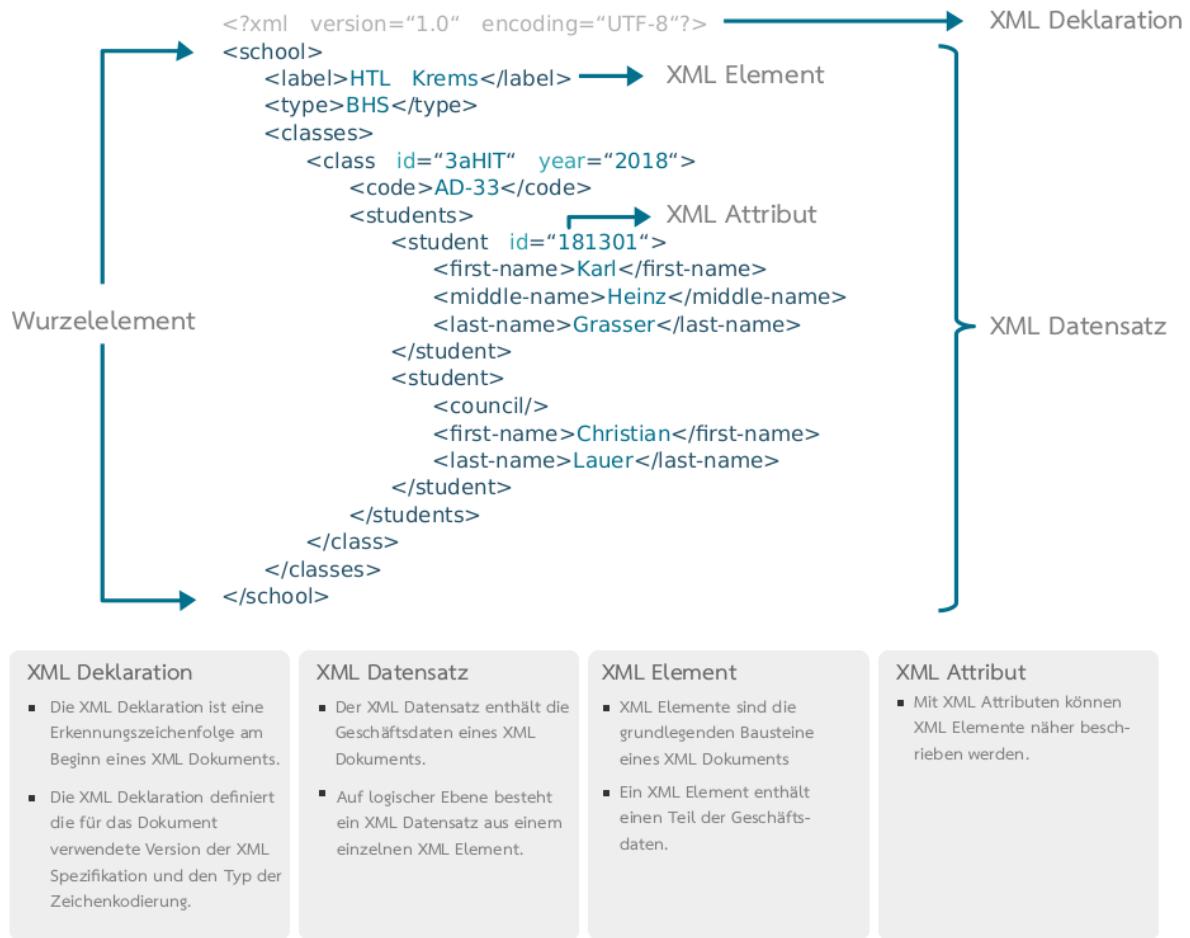


Abbildung 80. Aufbau eines XML Dokuments

► Auflistung: Datenaustauschformate ▾

- proprietäre Formate:** Der Handel unterstützt eine Vielzahl eigener XML Sprachen für den Austausch von Daten zwischen Anwendungen.
 - Officeanwendungen:** Officeanwendungen speichern Dokumente in eigens dafür definierten XML Formaten.
 - Grafikformate:** Oft werden Grafikformate als XML Sprachen definiert. SVG zählt dabei zu den wichtigsten auf XML basierten Formaten.
- SOAP Webservice:** Webservice tauschen Daten oft im XML Format aus. SOAP zählt dabei zu den am häufigsten verwendeten Formaten.

19.1.3 Aufbau eines XML Dokuments

Ein XML Dokument unterliegt einem durch die XML Spezifikation vorgegebenem Aufbau.

► Auflistung: Elemente eines XML Dokuments ▾



XML Deklaration ▾

Eine XML Datei unterteilt sich in die XML Deklaration und den XML Datensatz.

Die XML Deklaration ist eine **Erkennungszeichenfolge** am Beginn eines XML Dokuments. Sie definiert die für das Dokument verwendete Version der XML Spezifikation und den Typ der Zeichenkodierung.



XML Datensatz ▾

Der XML Datensatz enthält die **Geschäftsdaten** des XML Dokuments. Auf logischer Ebene besteht er aus einem einzelnen XML Element - dem **Wurzelement**.



XML Elemente ▾

XML Elemente sind die grundlegenden **Bausteine** eines XML Dokuments. XML Elemente enthalten Teile der Geschäftsdaten.



XML Attribute ▾

Mit XML Attributen können XML Elemente näher **beschrieben** werden.

19.2. XML Elemente ▾

XML Elemente sind die grundlegenden **Bausteine** eines XML Dokuments.

19.2.1 Struktur eines XML Dokuments

XML Elemente werden zur Speicherung und Strukturierung der Daten eines XML Datensatzes verwendet.

Mit XML Elementen wird eine strikte Trennung von Daten und **Strukturinformationen** implementiert.

► Erklärung: Xml Elemente ▾

- Ein XML Datensatz besteht immer aus einem XML Element - dem **Wurzelement**.
- Ein XML Element kann andere XML Elemente bzw. Daten enthalten.
- Enthält ein XML Element andere XML Elemente so können diese beliebig tief geschachtelt sein.

► Codebeispiel: Xml Dokument ▾

```

1  <!-- ----- -->
2  <!-- Beispieldokument -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <school>
6  <!-- Wurzelement des Xml Dokuments -->
7  ...
8  </school>
9
10 <?xml version="1.0" encoding="UTF-8"?>
11 <school>
12   <classes>
13     <class id="3aHIT" year="2018">
14       <student id="181301">
15         <first-name>Karl</first-name>
16         <middle-name>Heinz</middle-name>
17         <last-name>Grasser</last-name>
18       </student>
19       <student id="181303">
20         <first-name>Franz</first-name>
21         <last-name>Hofer</last-name>
22       </student>
23     </class>
24   </classes>
25 </school>
```



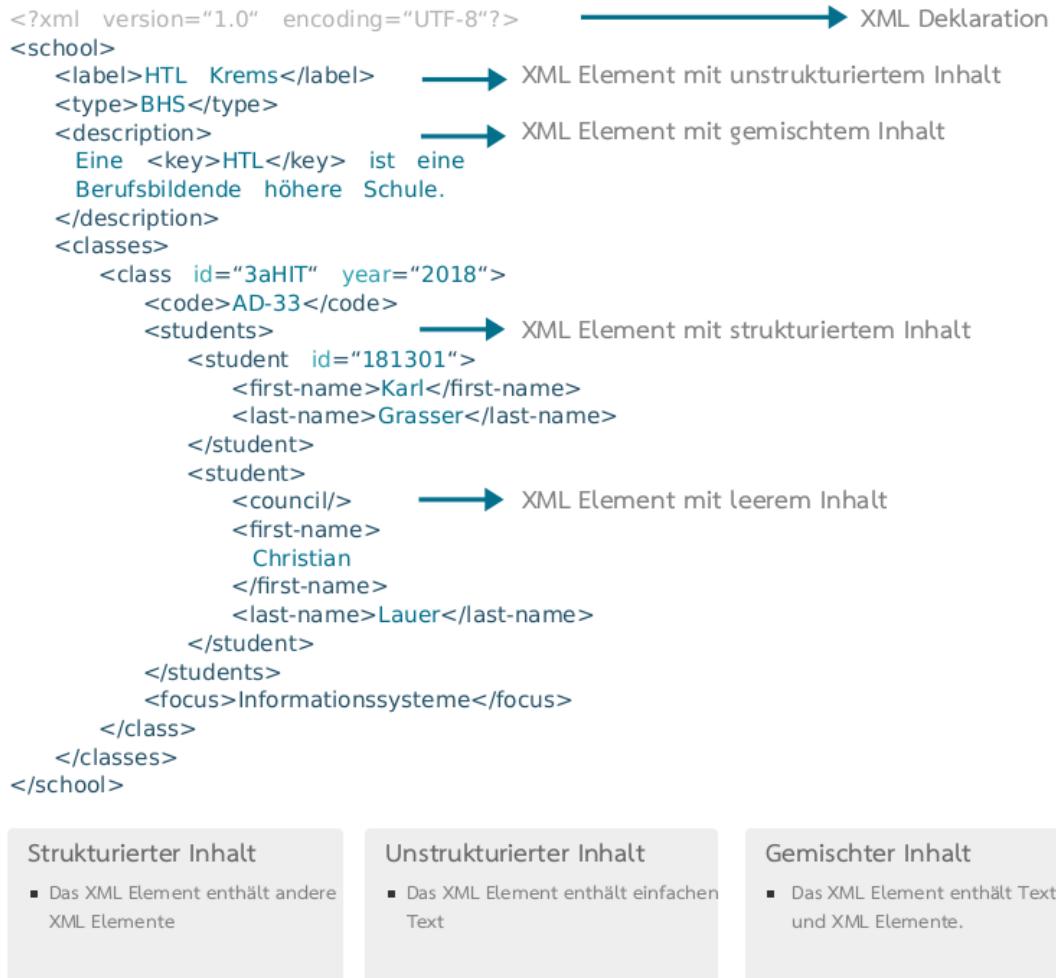


Abbildung 81. Inhaltstypen von XML Elementen

19.2.2 Aufbau eines XML Elements

Ein XML Element besteht immer aus einem **Start-** und einem **Endtag**. Die Bezeichnung des Elements kann dabei beliebig sein.

► Erklärung: Xml Element ▾

- Beispiel: <first-name>John</first-name>
- Das XML Element besteht aus dem Starttag <first-name> und dem dazugehörigen Endtag </first-name>.
- Das Element speichert den Token John.

19.2.3 Inhaltstypen von XML Elementen

XML Elemente können andere XML Elemente bzw. Daten enthalten.

► Auflistung: Inhaltstypen von Xml Elementen ▾

S unstrukturierter Inhalt ▾

Das XML Element enthält Daten in Form einer Zeichenkette. Der Inhaltstyp des XML Elements wird als **unstrukturiert** eingestuft.



strukturierter Inhalt ▾

Das XML Element enthält andere XML Elemente. Der Inhaltstyp des XML Elements wird als **strukturiert** eingestuft.



gemischter Inhalt ▾

Das XML Element enthält Daten und XML Elemente. Der Inhaltstyp des XML Elements wird als **gemischt** eingestuft.



leerer Inhalt ▾

Das XML Element hat **keinen Inhalt**. Der Inhaltstyp des XML Elements wird als **leer** eingestuft.

► Erklärung: Inhaltstypen ▾

- **unstrukturierter Inhalt:** Das XML Element enthält Daten in Form von Zeichenketten.

```
1 <first-name>John</first-name>
```

- **strukturierter Inhalt:** Das XML Element enthält andere XML Elemente.

```
1 <student id="1232345">
2   <first-name>John</first-name>
3   <middle>Fitzgerald Johansen</middle>
4   <last-name>Doe</last-name>
5   <phone>0650/443533</phone>
6   <email>j.doe@htlkrems.at</email>
7 </student>
```

- **gemischter Inhalt:** Das XML Element enthält Daten und andere XML Elemente.

```
1 <name id="1232345">
2   Der Name des Schuelers ist
3   <first>John</first>, mit dem Nachnamen
4   <middle>Fitzgerald Johansen</middle>
5   <last>Doe</last>
6   <phone>0650/443533</phone>
7   <email>j.doe@htlkrems.at</email>
8 </name>
```

- **leerer Inhalt:** Das XML Element hat keinen Inhalt.

```
1 <name/>
```

19.3. Attribute in Xml Elementen ▾

Mit XML Attributen können XML Elemente näher beschrieben werden.

19.3.1 Xml Attribute - Grundlagen

► Erklärung: Xml Attribute ▾

- Für ein XML Element können beliebig viele Attribute definiert werden.
- Die logische Struktur eines XML Attributs entspricht einem Name/Werte Paar.
- Ein XML Element kann nicht 2 Attribute mit dem gleichen Namen haben.

► Codebeispiel: XML Attribute ▾

```
1 <!-- ----- -->
2 <!-- Attribute -->
3 <!-- ----- -->
4 <?xml version="1.0" encoding="UTF-8"?>
5 <school>
6   <class id="3aHIT" semester="8">
7     <student id="181301">
8       <first-name>Karl</first-name>
9       <middle-name>Heinz</middle-name>
10      <last-name>Grasser</last-name>
11    </student>
12  </class>
13 </school>
```

19.3.2 Vergleich: Elemente vs. Attribute

XML Elemente und XML Attribute sind die grundlegenden Artefakte eines XML Dokuments.

► Analyse: Elemente vs. Attribute ▾

- Attribute können Daten nur in Form von Zeichenketten speichern. XML Elemente differenzieren hingegen unterschiedliche Inhaltstypen.
- Die Reihenfolge in der Attribute in einem XML Elementen auftreten sind belanglos. Die Reihenfolge in der XML Elementen in einem XML Dokument auftreten ist signifikant.
- Ein XML Attribut kann auch XML Element dargestellt werden.

19.4. Wohlgeformte XML Dokumente ▾

Das XML Datenformat definiert eine Reihe von Regeln für die Zusammensetzung eines XML Dokuments.

19.4.1 Wohlgeformte XML Dokumente

► Auflistung: Wohlgeformte XML Elemente ▾



Wurzelement ▾

Jedes XML Dokument besteht aus einem einzelnen Wurzelement.



Wohlgeformte Elemente ▾

Jedes Anfangstag muss ein zugehöriges Endtag haben.



Elementstruktur ▾

XML Elemente dürfen sich **nicht überlappen**.

19.4.2 Wurzelement

Jedes **XML Dokument** hat genau ein **Wurzelement**.

► Codebeispiel: Wurzelement ▾

```

1  <!-- ----- -->
2  <!--      Beispieldokument      -->
3  <!-- ----- -->
4  <!-- XML Dokument mit Wurzelement -->
5  <?xml version="1.0"?>
6  <name id="1232345" nickname="Shiny John">
7    <first>John</first>
8    <last>Doe</last>
9  </name>
10 
11 <!-- XML Dokument ohne Wurzelement -->
12 <?xml version="1.0"?>
13 <name id="1232345' nickname='Shiny John">
14   <first>John</first>
15   <middle>Fitzgerald Johansen</middle>
16   <last>Doe</last>
17 </name>
18 <name id="5672' nickname='Stinky Gordan">
19   <first>Gordan</first>
20   <last>Ramsey</last>
21 </name>
```

19.4.3 Elementtags

Jedes **Anfangstag** muss ein zugehöriges **Endtag** haben.

► Codebeispiel: Anfangstag und Endtag ▾

```

1  <!-- ----- -->
2  <!--      Beispieldokument      -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <student>
6    <name id="1232345" nickname="Shiny John">
7      <first>John</first>
8      <last>Silver</last>
9    </name>
10   <course>Informationssysteme</course>
11 </student>
12 
13 
14 <!-- HTML folgt nicht der XML Syntax -->
15 <html>
16   <body>
17     <p>Text
18     <br>More text in the same paragraph
19     <p>Some text in another paragraph</p>
20   </body>
21 </html>
```

19.4.4 Überlappung von XML Elementen

XML Elemente dürfen sich **nicht überlappen**.

► Codebeispiel: Überlappung von Elementen ▾

```

1  <!-- ----- -->
2  <!--      Beispieldokument      -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <name id="1232345" nickname="Shiny John">
6    <first>John</first>
7    <last>Loke</last>
8  </name>
9 
10 
11 <!-- HTML folgt nicht der XML Syntax -->
12 <html>
13   <body>
14     <p>Some <strong>formatted<em>text</strong>
15     , but </em> no grammar no good!
16   </body>
17 </html>
```



19.5. XML Namensräume



Xml Namensraum ▾

Ein XML Namensraum ist ein **logischer Namensraum** mit dem XML Elementen einem logischen Kontext zugeordnet werden können.

Innerhalb eines Namensraums müssen die Namen von XML Elementen eindeutig sein.

► Analyse: Xml Namensraum ▾

- Das Konzept eines XMO Namensraumes kann mit dem `namespace` von C# Klassen verglichen werden.
- Ein XML Namensraum ist ein **logischer Namensraum** mit dem XML Elementen einem logischen Kontext zugeordnet werden können.



19.5.1 Namenskonflikte

Im folgenden Beispiel haben die beiden `title` Elemente zwar denselben Namen, beschreiben aber unterschiedliche Konzepte.

Namensräume helfen XML Elemente mit gleichem Namen aber unterschiedlicher Bedeutung zu differenzieren.

► Codebeispiel: XML Dokument ▾

```

1  <!-- ----- -->
2  <!--      Namensraum      -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <course>
6    <title>Semantic Web</title>
7    <description>Semantic ... </description>
8    <lecturers>
9      <name>
10        <title>Pirv.-Doz. Dr.</title>
11        <last>Staab</last>
12      </name>
13    </lecturers>
14  </course>
```

19.5.2 XML Namensräume

► Erklärung: XML Namensräume ▾

- Zur Auflösung von Namenskonflikten werden XML Elemente in Namensräume zusammengefaßt.
- XML Namensräume zeigen die logische Zusammengehörigkeit von XML Elementen an.
- Namensräume werden durch eine URI identifiziert.
- Jedem Element des Namensraums wird ein Bezeichner vorangestellt um seine Zugehörigkeit zum Namensraum anzuzeigen.

► Codebeispiel: Namensräume ▾

```

1  <!-- ----- -->
2  <!--      Namensraum      -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <c:course xmlns:c="http://www.htl.at/courses">
6    <c:title>Semantic Web</c:title>
7    <c:description>
8      Der Kurs Semantic Web behandelt die
9      Prinzipien semistrukturerter Daten.
10     </c:description>
11     <c:lecturers>
12       <l:lecturer>
13         <l:name>
14           <l:title>Pirv.-Doz. Dr.</l:title>
15           <l:first>Steffen</l:first>
16           <l:last>Staab</l:last>
17         </l:name>
18       </l:lecturer>
19       <l:lecturer>
20         <l:name>
21           <l:first>Gerald</l:first>
22           <l:last>Futschek</l:last>
23         </l:name>
24       <l:contact>
25         <l:email>g.futsch@tuwien.at</l:email>
26         <l:phone>0650/543467</l:phone>
27       </l:contact>
28     </l:lecturer>
29   </c:lecturers>
30 </c:course>
```



19.5.3 Namensraumdefinition

Namensraumdefinitionen müssen global eindeutig sein. XML Namensräume werden aus diesem Grund durch eine **URI** beschrieben.

► Codebeispiel: Namensräume ▾

```

1  <!-- ----- -->
2  <!--      Namensraum      -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <c:course xmlns:c="http://www.htl.at/course">
6    <c:title>Semantic Web</c:title>
7    <c:description>
8      Der Kurs Semantic Web behandelt die
9      semistrukturierter Daten.
10   </c:description>
11   <c:date>23.09.23 16:15:00</c:date>
12   <loc:location
13     xmlns:loc="http://www.htl.at/location">
14     <loc:country>Austria</loc:country>
15     <loc:city>
16       <loc:name>Vienna</loc:name>
17       <loc:code>1040</loc:code>
18     </loc:city>
19     <loc:building>TU Vienna</loc:building>
20     <loc:room>Seminarraum 134</loc:room>
21   </loc:location>
22   <c:lecturers>
23     <l:lecturer
24       xmlns:l="http://www.htl.at/lecturer">
25       <l:name>
26         <l:title>Pirv.-Doz. Dr.</l:title>
27         <l:first>Steffen</l:first>
28         <l:last>Staab</l:last>
29       </l:name>
30       <l:contact>
31         <l:email>s.staab@hugo.com</l:email>
32         <l:phone>0650/543467</l:phone>
33       </l:contact>
34     </l:lecturer>
35     <l:lecturer
36       xmlns:l="http://www.htl.at/lecturer">
37       <l:name>
38         <l:title>Pirv.-Doz. Dr.</l:title>
39         <l:first>Steffen</l:first>
40         <l:last>Staab</l:last>
41       </l:name>
42     </l:lecturer>
43   </c:lecturers>
44 </c:course>
```

19.5.4 Standard Namensraum



Standard Namensraum ▾

Der **Standardnamensraum** gilt für das Element indem er definiert wird, genauso wie für alle Kindelemente dieses Elements.

Für einen **Standardnamensraum** wird kein Prefix definiert.

► Codebeispiel: Standard Namensräume ▾

```

1  <!-- ----- -->
2  <!--      Standard Namensraum      -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <c:course xmlns:c="http://www.htl.at/course">
6    <c:title>Semantic Web</c:title>
7    <c:description>
8      Der Kurs Semantic Web behandelt die
9      semistrukturierter Daten.
10   </c:description>
11   <loc:location
12     xmlns:loc="http://www.htl.at/location">
13     <loc:country>Austria</loc:country>
14   </loc:location>
15   <c:lecturers>
16     <l:lecturer
17       xmlns:l="http://www.htl.at/lecturer">
18       <l:name>
19         <l:title>Pirv.-Doz. Dr.</l:title>
20         <l:first>Steffen</l:first>
21         <l:last>Staab</l:last>
22       </l:name>
23       <l:contact>
24         <l:email>s.staab@hugo.com</l:email>
25       </l:contact>
26     </l:lecturer>
27   <l:lecturer
28     xmlns:l="http://www.htl.at/lecturer">
29     <l:name>
30       <l:title>Pirv.-Doz. Dr.</l:title>
31       <l:first>Steffen</l:first>
32       <l:last>Staab</l:last>
33     </l:name>
34     <l:contact>
35       <l:email>s.staab@hugo.com</l:email>
36     </l:contact>
37   </l:lecturer>
38 </c:lecturers>
39 </c:course>
```

XML Technologie

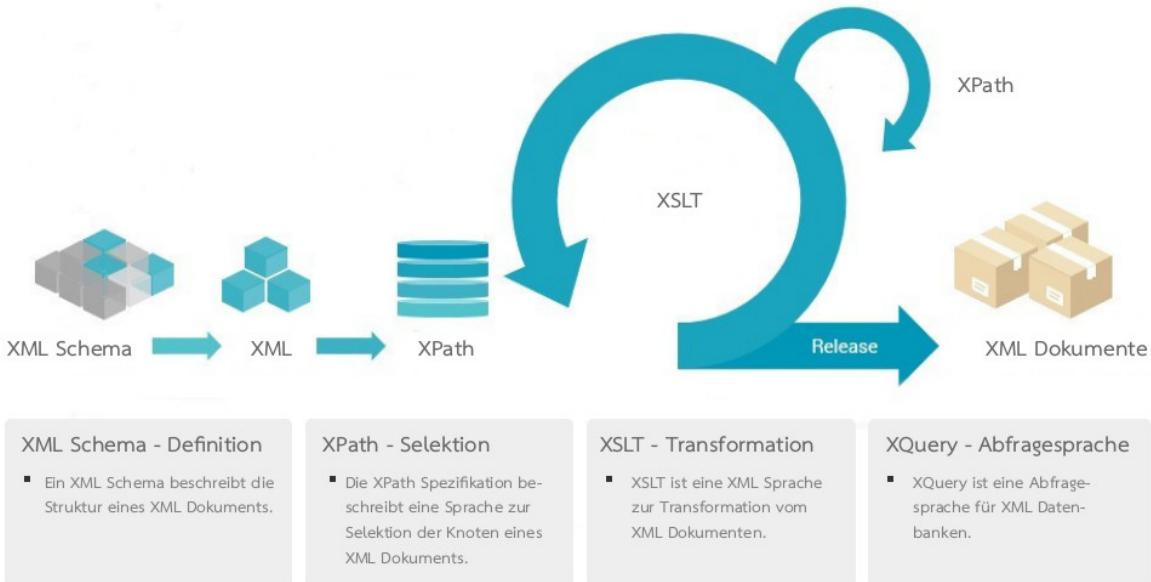


Abbildung 82. XML Technologien

19.6. Logische Sicht

XML wurde mit dem Anspruch entwickelt ein universelles Datenformat für das Internet zu schaffen.

19.6.1 XML Technologien

Zur Verarbeitung von XML Daten wurde eine Reihe eigener Technologien definiert.

► Auflistung: XML Technologien ▾

- **XML:** Die XML Spezifikation definiert ein Datenformat zum Austausch von Daten im Internet.
- **XML Schema:** XML Schemas werden verwendet um die Struktur für XML Dokumente zu definieren.
- **XPath:** Die XPath Spezifikation beschreibt eine Sprache zur **Selektion** von Knoten in XML Dokumenten.
- **XSLT:** XSLT ist eine XML Sprache zur **Transformation** von XML Dokumenten.
- **XQuery:** XQuery ist eine **Abfragesprache** für XML Datenbanken.

19.6.2 XML Komponentenbaum

Bevor ein XML Dokument verarbeitet werden kann muss für das Dokument eine **logische Sicht** generiert werden.



XML Komponentenbaum ▾

Der logisch Aufbau eines XML Dokuments entspricht einer **hierarchischen Baumstruktur**. Die Baumstruktur selbst wird als Komponentenbaum bezeichnet.

Damit existieren für XML Dokumente immer 2 Sichten: die strukturelle- und die logische Sicht.

► Erklärung: XML Komponentenbaum ▾

- Ein XML Komponentenbaum besteht aus Knoten.
- Der Aufbau eines Knotenbaums entspricht dabei der hierarchischen Struktur der strukturellen Sicht des zugehörigen XML Dokuments. Für die unterschiedlichen Elemente im XML Dokument werden im Knotenbaum eigene Knoten integriert.

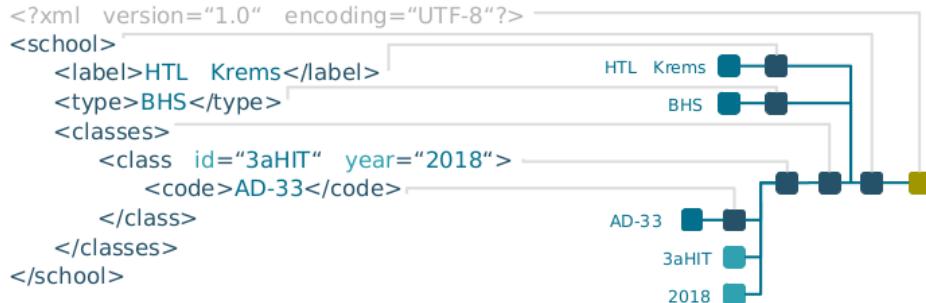


Abbildung 83. strukturelle vs. logische Sicht

19.6.3 Komponentenknoten

Jedes XML Dokument besitzt eine Darstellung als Komponentenbaum.

► Erklärung: Komponentenknoten ▾

- Jedes Element eines XML Dokuments besitzt eine entsprechende Repräsentation als Knoten im Knotenbaum.
- Die XML Spezifikation definiert dabei für folgende Elementtypen eigene **Knotentypen**: XML Deklaration, Kommentare, XML Elemente, XML Attribut.
- Tritt eines dieser Elemente in einem XML Dokument auf wird ein Knoten des entsprechenden Typs im Knotenbaum eingefügt.
- Elemente die in einem XML Element eingebettet sind werden als Kindknoten des entsprechenden Elementknotens dargestellt.

```

1  <!-- ----- -->
2  <!--          Knotenbaum          -->
3  <!-- ----- -->
4  <class id="3aHIT" year="2018">
5    <code>AD-33</code>
6  </class>

```

Das `<class>` Element besitzt im Knotenbaum eine Repräsentation als Elementknoten. Dem Elementknoten sind dabei folgende Kindknoten zugeordnet: Ein Elementknoten für das `<code>` Element zusammen mit 2 Attributknoten für die Attribute des Elements.

- Beachten Sie das Knoten unterschiedlicher Knotentypen Kindelemente eines Elementknoten sein können.

19.6.4 Knotentypen

Die XML Spezifikation definiert für die logische Sicht von XML Daten folgende Knotentypen.

► Auflistung: Knotentypen ▾

	Wurzelknoten
	Der Wurzelknoten ist der primäre Knoten eines Knotenbaums. Der Wurzelknoten enthält alle anderen Knoten des Knotenbaums.
	Elementknoten
	Elementknoten entsprechen der logischen Repräsentation eines XML Elements.
	Attributknoten
	Attributknoten entsprechen der logischen Repräsentation eines XML Attributes. Der Attributknoten wird dem Elementknoten als Kindelement zugeordnet.
	Textknoten
	Die in einem Element enthaltenen Daten werden einem eigenen Textknoten zugeordnet. Der Textknoten wird dem entsprechendem Elementknoten als Kindelement zugeordnet.
	Namensraumknoten
	Der Namensraum eines Elements ist ihm als Elternknoten zugeordnet.

20. Datenformat - XML XPath

03

XPath

01. XPath Konzepte	206
02. Pfadausdrücke in Kurzform	207
03. Lösungsobjekt	209
04. Prädikate	209
05. Pfadausdrücke in Standardform	210
06. XPath Funktionen	212
07. Fallbeispiel: XPath	215

20.1. XPath - Konzepte



XPath - Selektion ▾

XPath ist eine **Adressierungssprache** zur Auswahl von Knoten in XML Dokumenten.

Der **XPath Standard** dient als **Grundlage** für eine Reihe anderer XML Technologien wie XSLT, XML Schema bzw. XQuery.

20.1.1 XPath Konzepte

Der Begriff XPath selbst ist ein Akronym. Die Abkürzung steht stellvertretend für den Ausdruck **X**ml **P**ath Language.

► Erklärung: **XPath Grundlagen** ▾

- XPath selbst ist keine XML Sprache. Der XPath Standard definiert eine eigene Syntax zur Formulierung von **Pfadausdrücken**.
- Mit einem XPath Pfadausdruck kann die Position von Knoten in einem XML Knotenbaum beschrieben werden.

► Erklärung: **Lokalisierungspfade** ▾

- Bevor ein XPath Ausdruck ausgewertet werden kann, muss für die entsprechenden XML Daten die assozierte logische Sicht bestimmt werden. Lokalisierungspfade werden stets in Relation zum **Knotenbaum** eines XML Dokuments ausgewertet.
- Soll beispielsweise auf einen bestimmten Elementknoten eines XML Dokuments zugegriffen werden, wird ein **logischer Pfad** ausgehend vom Wurzelknoten des Knotenbaums zum gewünschten Elementknoten definiert.
- Mit XPath Lokalisierungspfaden kann dabei auf beliebige Teile der XML Daten zugegriffen werden.
- Das Ergebnis der Auswertung eines Lokalisierungspfades wird als **Lösungsobjekt** bezeichnet. Lösungsobjekte können Knotenmengen, Strings, Zahlen bzw. boolesche Werte sein.
- Die XPath Spezifikation unterscheidet für Lokalisierungspfade 2 Formen: Pfadausdrücke in Standardschreibweise bzw. Pfadausdrücke in Kurzschreibweise.

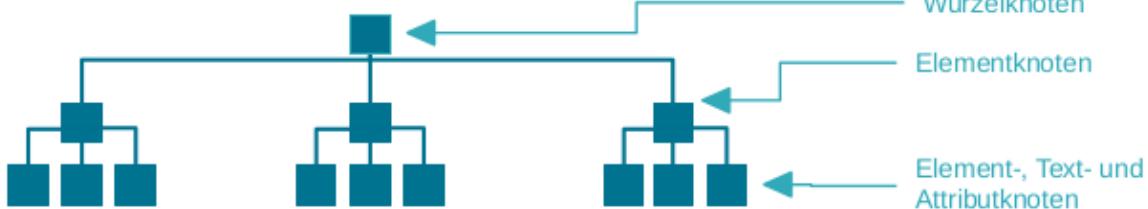


Abbildung 84. Knotenbaum eines XML Dokuments

20.2. Pfadausdrücke in Kurzform ▾

Der grundlegende **Ausdruckstyp** in XPath ist der **Lokalisierungspfad**.

20.2.1 Vereinfachte Pfadausdrücke

Die XPath Spezifikation erlaubt für Lokalisierungspfade eine vereinfachte Schreibweise: die **XPath Kurzform**.

► Erklärung: XPath Kurzform ▾

- Bei der Auswertung von Pfadausdrücken in Kurzform trifft die XPath Engine eine Reihe von **Annahmen**.
- Ihre kompakte Form schuldet die XPath Kurzform dabei den getroffenen Annahmen, das Selektionsverhalten des entsprechenden Pfadausdrucks ist in seiner Reichweite jedoch auf eine Zahl von Anwendungsfällen beschränkt.

► Erklärung: Lokalisierungspfade ▾

- Ein Lokalisierungspfad ist eine Folge von Lokalisierungsstufen. Lokalisierungsstufen sind voneinander durch den / Delimiter getrennt.
- Je nach **Knotentyp** definiert die XPath Spezifikation einen eigenen Operatoren für die **Knotenauswahl**.

```

1 // -----
2 //     XPath: Lokalisierungspfade
3 // -----
4 // Selektion: <student> Elemente
5 // Ergebnis: Alle <student> Elemente
6
7 XPath: /school/classes/class/student

```

20.2.2 Auswahl von Elementknoten

Die Position von Elementknoten in XML Dokumenten kann durch einfache XPath Pfadausdrücke beschrieben werden.

► Erklärung: Knotenpfad ▾

- Für die Auswahl von **Elementknoten** in XML Dokumenten wird ein logischer Pfad ausgehend vom Wurzelknoten des Knotenbaums zum gewünschten Elementknoten definiert.
- Jeder Lokalisierungsschritt des Pfadausdrucks entspricht dabei einem der übergeordneten Elementknoten des gewünschten Elements.

► Codebeispiel: Zugriff auf Elementknoten ▾

```

1 // -----
2 //     XPath: Pfadausdrücke
3 // -----
4 // Selektion: Alle <student> Elemente
5 // Ergebnis: Alle im XML Dokument
6 // enthaltenen <student> Elemente
7
8 XPath: /school/classes/class/student
9
10 // Selektion: Alle <school> Elemente
11 // Ergebnis: Der <school> Wurzel-
12 // elementknoten
13
14 XPath: /school
15
16 // Selektion: Alle <schule> Elemente
17 // Lösung: In den XML Daten sind keine
18 // <schule> Elemente enthalten. Die
19 // Lösungsmenge ist leer.
20
21 XPath: /schule

```

20.2.3 Kontextknoten



Kontextknoten ▾

Der **Kontextknoten**, ist jener Knoten eines XML Dokuments, der zum gegenwärtigen Zeitpunkt von der **XPath Engine** verarbeitet wird.

► Erklärung: Kontextknoten ▾

- Zur Referenzierung des **Kontextknotens** wird der **Punktoperator** verwendet.
- Der Elternknoten des Kontextknotens kann über den **.. Operator** ausgewählt werden.

► Codebeispiel: Punktoperatoren ▾

```

1 // -----
2 //      XPath: Kontextoperatoren
3 // -----
4 // Selektion: Alle <first-name> Kindelemente
5 // des Kontextknotens
6 // Loesungsobjekt: Knotenmenge
7 XPath: ./first-name
8
9 // Selektion: Alle <first-name> Kindelemente
10 // des Elternknotens des Kontextknotens
11 // Lsungsobjekt: Knotenmenge
12 XPath: ../name

```

20.2.4 Descendant Pfadoperator

Der Descendant Pfadoperator ermöglicht einen Knotendurchlauf über alle Nachkommen des Kontextknotens.

► Codebeispiel: Doppelte Pfadoperator ▾

```

1 // -----
2 //      XPath: Descendant Pfadoperator
3 // -----
4 // Selektion: alle Vorkommen des <first-name>
5 // Elements im XML Dokument
6 XPath: //first-name
7
8 // Selektion: alle <first-name> Elemente
9 // die Kindelemente eines <student>
10 // Elements im XML Dokument
11 XPath: //student/first-name

```

20.2.5 Auswahl von Textknoten

Textknoten werden im XML Knotenbaum als Kindknoten von Elementknoten gespeichert.

► Erklärung: Zugriff auf Textknoten ▾

- Textknoten werden in XPath Lokalisierungspfaden über den **text()** Operator ausgewählt.
- Das Lösungsobjekt der **text()** Knotenabfrage ist eine Zeichenkette.

► Codebeispiel: Zugriff auf Textknoten ▾

```

1 // -----
2 //      XPath: text() Knotentest
3 // -----
4 //Selektion: Alle Vornamen der Schueler
5 //Loesungsobjekt: String
6 XPath: //student/first-name/text()

```

20.2.6 Auswahl von Attributknoten

Obwohl die XPath Spezifikation, Elementknoten als Elternknoten ihrer Attributknoten definiert, können Attributknoten nicht als Kindknoten eines Elementknotens angesprochen werden.

► Erklärung: Zugriff auf Attribute ▾

- Für den Zugriff auf die Attributknoten eines Elementknotens wird der **@ Operator** verwendet.
- Das Ergebnis der Auswertung des **@ Operators** ist eine Knotenmenge von Attributknoten.
- Mit der Kombination des **@** und des ***** Operators kann auf alle Attributknoten eines Elements zugegriffen werden.

► Codebeispiel: Zugriff auf Attribute ▾

```

1 // -----
2 //      XPath: @ Operator
3 // -----
4 // Selektion: die id Attribute aller Schueler
5 // Loesungsobjekt: Knotenmenge
6 XPath: //student/@id
7
8 // Selektion: alle Attribute der Schueler
9 // Loesungsobjekt: Knotenmenge
10 XPath: //student/@*

```

20.3. Lösungsobjekt

Das Ergebnis eines XPath Ausdrucks ist ein **Lösungsobjekt**.

20.3.1 Lösungsobjekt - Datentypen

Die XPath Spezifikation definiert 4 Typen von Lösungsobjekten.

► Auflistung: Datentypen für Lösungsobjekte ▾



Knotenmenge ▾

Eine Knotenmenge ist eine **Teilmengen** der Knoten eines Knotenbaums.



String ▾

Ein String ist eine **Zeichenfolge**. Die Zeichenfolge kann dabei auch leer sein.

z.B.: 'Hallo Welt'



Number ▾

Number Objekte sind 64 Bit **Fließkommazahlen**.

z.B.: 34



Boolean ▾

Boolean Objekte nehmen entweder den Wert **true** oder **false** an.

► Codebeispiel: Datentypen ▾

```

1 // -----
2 //   XPath: Lösungsobjekte
3 // -----
4 // Lösungsobjekt: Knotenmenge
5 XPath: //student
6
7 // Lösungsobjekt: String
8 XPath: //student/first-name/text()
9
10 // Lösungsobjekt: Number
11 XPath: count(//person)
12
13 // Lösungsobjekt: Boolean
14 XPath: boolean(/project/title)
```

20.4. Prädikat

Zur Filterung der Knoten einer Knotenmenge können Prädikate definiert werden.

20.4.1 Definieren von Prädikaten



XPath Prädikat ▾

Ein **Prädikat** ist ein **logischer Ausdruck**.

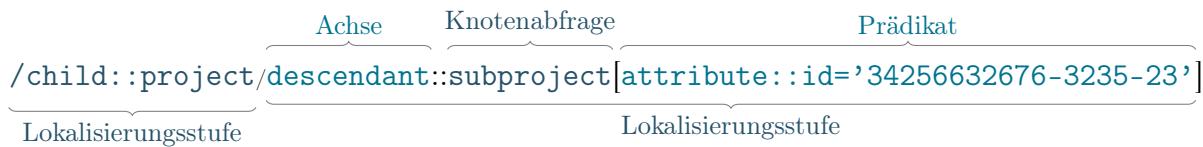
► Erklärung: Filtern mit Prädikaten ▾

- Prädikate werden in eckigen Klammern, am Ende einer Lokalisierungsstufe definiert. Die eckigen Klammern sind dabei ein **Steuerzeichen** der XPath Engine.
- Für eine XPath Lokalisierungsstufe kann eine beliebige Zahl von Prädikaten definiert werden.
- Werden für einen XPath Ausdruck mehrere Prädikate definiert, werde die Prädikate ausgehend vom ersten Prädikat zum letzten hin ausgewertet.
- Prädikatsausdrücke unterstützen die folgenden Vergleichsoperatoren : <, >, >=, <=, =, !=.
- Allerdings muß beachtet werden, dass Operatoren wie <, > nicht unmittelbar in einem XML Dokument erscheinen dürfen, sondern durch die entsprechenden **Entitätsreferenzen** < bzw. > ersetzt werden müssen.

► Codebeispiel: Prädikate ▾

```

1 // -----
2 //   XPath: Prädikate
3 // -----
4 // Alle Projekte mit einer bestimmten id
5 /projects[@id='343225']
6
7 // Alle Personen die ein name Element haben
8 //person[name]
9
10 // Alle subprojekte die einen Partner haben
11 /project[partners/partner]/subprojects/
12 subproject
13
14 // Die 3te Person einer Knotenmenge
15 //person[3]
16
17 // Alle Personen die eine id haben
18 //person[@id]
```



20.5. Pfadausdrücke in Standardform ▾

Die XPath Spezifikation unterscheidet 2 Formen von Lokalisierungspfaden: Lokalisierungspfade in **Standardschreibweise** bzw. Lokalisierungspfade in Kurzform.

Ein Lokalisierungspfad ist eine Folge von **Lokalisierungsstufen**.

20.5.1 Lokalisierungsstufen

Eine Lokalisierungsstufe besteht aus 3 möglichen Segmenten.

► Syntax: Lokalisierungsstufe ▾

```
1 // -----
2 //   XPath: Syntax Lokalisierungsstufe
3 // -----
4 Achse::Knotenabfrage[Prädikat1][...]
```

► Auflistung: Segmente einer Lokalisierungsstufe ▾



Achsenbezeichner ▾

Der Achsenbezeichner definiert die **Richtung** des **Knotendurchlaufs** einer Lokalisierungsstufe. Das Ergebnis eines Knotendurchlaufs ist eine Knotenmenge.

Die Angabe eines Achsenbezeichners ist verpflichtend.



Knotenabfrage ▾

Die Knotenabfrage ermöglicht eine **Vorauswahl** der durch den Knotendurchlauf bestimmten Knoten.

Die Angabe einer Knotenabfrage ist verpflichtend.



Prädikat ▾

Prädikate erlauben die Formulierung komplexer **Filterbedingungen**.

20.5.2 Achsenbezeichner

Achsenbezeichner definieren die **Richtung** eines **Knotendurchlaufs** in Lokalisierungsstufen.



► Erklärung: Achsenbezeichner ▾

- Jede **Lokalisierungsstufe** besteht aus einem Achsenbezeichner, einer Knotenabfrage und gegebenenfalls aus Prädikaten.
- Der Achsenname bestimmt die Richtung, in die der Lokalisierungspfad fortgesetzt werden soll.
- Das Ergebnis eines Knotendurchlaufs enthält alle Knoten der gewählten Achse relativ zum Kontextknoten.

► Auflistung: Achsen ▾

- **self:** Die Achse referenziert den Kontextknoten.
- **child:** Die Achse referenziert alle Kindknoten des Kontextknotens.
- **parent:** Die Achse referenziert den Elternknoten des Kontextknotens.
- **descendant:** Die Achse referenziert alle Nachkommen des Kontextknotens (Kinder, Kindeskinder etc.).
- **ancestor:** Die Achse referenziert alle Vorfahren des Kontextkontexts inc. des Wurzelknotens.
- **following:** Die Achse referenziert alle Nachkommen des Kontextknotens in Dokumentreihenfolge.
- **following-sibling:** Die Achse referenziert alle nachfolgenden **Geschwister** des Kontextknotens.
- **preceding-sibling:** Die Achse referenziert alle vorhergehenden **Geschwister** des Kontextknotens.
- **preceding:** Die Achse referenziert alle vorhergehenden Knoten des Kontextknotens in Dokumentreihenfolge.
- **attribute:** Die Achse referenziert alle Attributknoten des Kontextknotens.

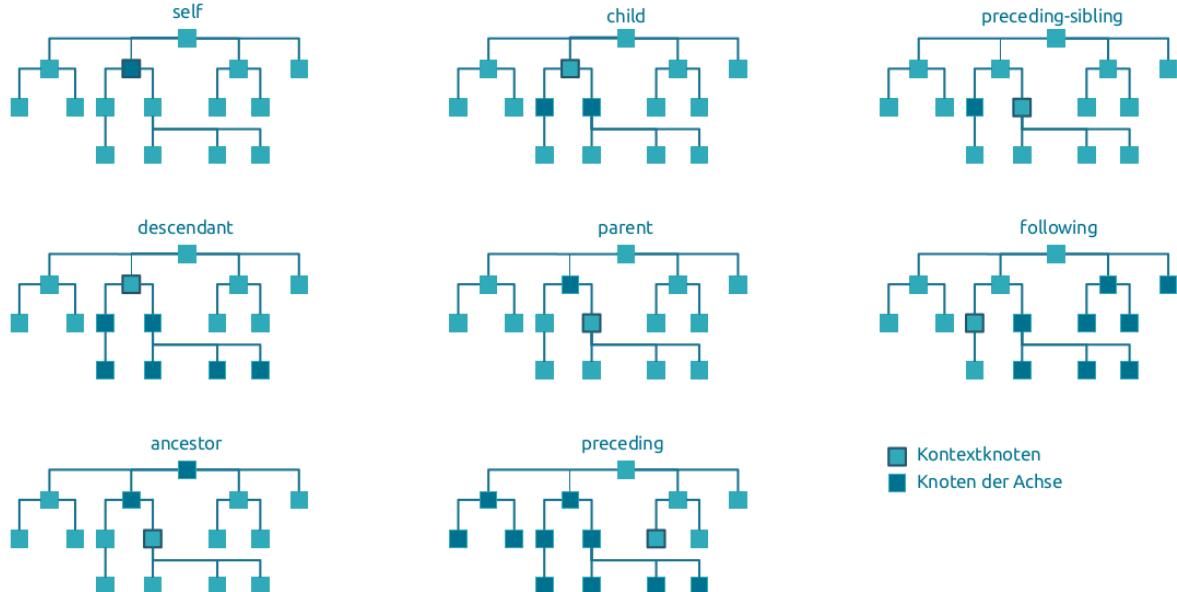


Abbildung 85. Achsen in XPath

20.5.3 Knotenabfragen

Mit einer Knotenabfrage werden die Knoten eines Knotendurchlaufs gefiltert.

► Erklärung: Knotenabfrage ▾

- Die Knotenabfrage selbst wird an den Achsenbezeichner angehängt und ermöglicht eine **Vorauswahl**, der durch den Knotendurchlauf bestimmten Knoten.
- Die Knotenabfrage besteht dabei entweder aus einem **Elementvergleich** bzw. dem Test auf einen bestimmten Knotentypen.
- Ein Elementvergleich wählt alle Knoten des Knotendurchlaufs aus, die einen bestimmten Namen haben.
- Knotentypentests** ermöglichen eine Vorauswahl aller Knoten des Knotendurchlaufs die eine bestimmten Knotentypen haben. Die XPath Spezifikation definiert für Knotentypentests eine Reihe von Operatoren und Funktionen.
- Zur Formulierung komplexer Filterbedingungen können Prädikate³ für XPath Lokalisierungsstufen definiert werden.

► Codebeispiel: Knotenabfrage ▾

```

1 // -----
2 // XPath: Knotenabfrage
3 // -----
4 // Selektion: Alle Elementknoten im Knoten-
5 // durchlauf
6 // Knotentesttyp: Knotentypentest
7 XPath: child::*
8
9 // Selektion: Alle <student> Elementknoten
10 // im Knotendurchlauf
11 // Knotentesttyp: Elementvergleich
12 XPath: parent::student
13
14 // Selektion: Alle Textknoten im Knoten-
15 // durchlaufs
16 // Knotentesttyp: Knotentypentest
17 XPath: ancestor::text()
18
19 // Selektion: Alle Knoten des KD.
20 // Knotentesttyp: Knotentypentest
21 XPath: descendant::node()
22
23 // Selektion: Alle Kommentarknoten
24 // Knotentesttyp: Knotentypentest
25 XPath: parent::comment()

```

³ siehe Kapitel Prädikate

20.5.4 Formen von XPath Ausdrücken

Für Lokalisierungsstufen unterscheidet die XPath Spezifikation 2 Formen: die **Standardschreibweise** bzw. Lokalisierungsstufen in Kurzform.



► Vergleich: Kurz- vs. Standardform ▾

- Lokalisierungspfade können als Mischform der Standardschreibweise bzw. der Kurzform definiert werden.
- Beachten Sie das jede Lokalisierungsstufe in Kurzform in einen äquivalenten XPath Ausdruck in Standardform übergeführt werden kann. Der Umkehrschluß gilt jedoch nicht.

► Codebeispiel: Transformationsregeln ▾

```

1 // -----
2 //  XPath: Standardform vs. Kurzform
3 // -----
4 // Pfadausdrücke
5 Kurzform:    classes/class
6 Standardform: child::classes/child::class
7
8 // Attributknoten
9 KF: //student/@id
10 SF: /descendant::student/attribute::id
11 Mischform: //student/attribute::id
12
13 KF: //student/@*
14 SF: /descendant::student/attribute::node()
15 MF: /descendant::student/@*
16
17 // Kontextknoten
18 KF: .
19 SF: self::node()
20
21 //Elternknoten
22 KF: ..
23 SF: parent::node()
24
25 // Textknoten
26 KF: //first-name/text()
27 SF: /descendant::first-name/child::text()
28 MF: //first-name/child::text()
29
30 KF: //last-name/text()
31 SF: /descendant::last-name/child::text()

```

20.6. XPath Funktionen

Die XPath Spezifikation umfasst eine Reihe von Funktionen zur Bearbeitung von XPath Ausdrücken.

20.6.1 Kategorien von Funktionen

XPath Funktionen werden einer von 4 Kategorien zugeordnet.

► Auflistung: Kategorien von Funktionen ▾



Knotenmengenfunktionen: ▾

Knotenmengenfunktionen werden zur Verarbeitung von Mengen von Knoten eingesetzt.



String Funktionen: ▾

String Funktionen werden zur Verarbeitung von Zeichenketten eingesetzt.



Numerische Funktionen: ▾

Numerische Funktionen werden zur Verarbeitung von Zahlenwerten verwendet.



Logische Funktionen: ▾

Logische Funktionen werden zur Verarbeitung von booleschen Ausdrücken verwendet.

20.6.2 Knotenmengenfunktionen

► Auflistung: Knotenmengenfunktionen ▾

- **last()**, **count()**: Ermittelt die Anzahl von Knoten einer **Knotenmenge**.
- **position()**: Ermittelt die **Position** eines Knotens in einer **Knotenmenge**.
- **local-name()**: Ermittelt den **lokalen Namen** eines **Knotens**.
- **namespace-uri()**: Ermittelt die Namensraum-URI eines Knotens.
- **name()**: Ermittelt den **qualifizierten Namen** eines Knotens.

► Codebeispiel: Knotenmengenfunktionen ▾

```

1 // -----
2 //   XPath Funktion: last
3 // -----
4 /project/subprojects/subproject[last()]
5
6 /* Ermittelt das letzten subproject Element
7   der Knotenmenge */
8
9 // -----
10 //   XPath Funktion: count
11 // -----
12 count(/project/subprojects/subproject)
13
14 /* Ermittelt die Anzahl von Subprojekte in der
15   Knotenmenge */

```

► Codebeispiel: Stringfunktionen ▾

```

1 // -----
2 //   XPath Funktion: string
3 // -----
4 /* xs:string string(
5   items elem
6 ) */
7
8 XPath: string(/project[@id='343225']/name)
9 Ergebnis: 'Simulation'
10
11 XPath: string(/project/@id)
12 Ergebnis: '343225'
13
14 XPath: string(2 = 2)
15 Ergebnis: 'true'
16
17 // -----
18 //   XPath Funktion: concat
19 // -----
20 /* xs:string concat(
21   xs:anyAtomicType* token
22 ) */
23
24 XPath: concat('a', 'b', 'c')
25 Ergebnis: 'abc'
26
27 XPath: concat(
28   /person[last()]/first-name,
29   ',',
30   /person[last()]/middle-name
31   ',',
32   /person[last()]/last-name,
33 )
34
35 Ergebnis: Stefan Jell
36
37 // -----
38 //   XPath Funktion: starts-with
39 // -----
40 /* xs:boolean starts-with(
41   xs:string token1, xs:string token2
42 ) */
43
44 XPath: starts-with('yes', 'yes')
45 Ergebnis: true
46
47 XPath: starts-with(/person[last()]/name, 'J')
48 Ergebnis: true

```

20.6.3 String Funktionen

► Auflistung: String Funktionen ▾

- **string():** Ermittelt die Stringdarstellung der übergebenen Knotenmenge.
- **concat():** Verkettet die übergebenen Zeichenketten.
- **starts-with():** Gibt den Wert wahr, zurück wenn die erste Zeichenkette mit der zweiten Zeichenkette beginnt.
- **contains():** Gibt den Wert wahr zurück, wenn die zweite Zeichenkette in der ersten Zeichenkette enthalten ist.
- **substring-before():** Liefert aus der ersten Zeichenkette ds, was vor dem Teil steht, der mit der zweiten Zeichenkette übereinstimmt.
- **substring-after():** Liefert aus der ersten Zeichenkette das, was hinter dem Teil steht, der mit dem ersten Auftreten der zweiten Zeichenketten übereinstimmt.
- **substring():** Liefert eine Teil einer Zeichenkette, der an der mit dem zweiten Argument angegebenen Position beginnt und die mit dem dritten Argument angegeben Länge hat.
- **string-length():** Liefert die Anzahl der Zeichen in einer Zeichenkette.

► Codebeispiel: Stringfunktionen ▾

```

1 // -----
2 // XPath Funktion: string-length
3 //
4 /* xs:integer string-length(
5     xs:string token
6 ) */
7
8 XPath: string-length('abc')
9 Ergebnis: 3
10
11 // -----
12 // XPath Funktion: contains
13 //
14 /* xs:boolean contains(
15     xs:string param1,
16     xs:string param2
17 ) */
18
19 XPath: contains('Shakespeare', 'spear')
20 Ergebnis: true
21
22 XPath: contains('Shakespeare', '')
23 Ergebnis: true
24
25 XPath: contains('', 'Shakespeare')
26 Ergebnis: false
27
28 XPath: contains(/person[last()]/name, 'J')
29 Ergebnis: true
30
31 // -----
32 // XPath Funktion: substring
33 //
34 /* xs:string substring(
35     xs:string param1,
36     xs:double param2,
37     xs:double param3
38 ) */
39
40 XPath: substring('abcde', 2)
41 Ergebnis: 'bcde'
42
43 XPath: substring('abcde', 2, 2)
44 Ergebnis: 'bc'
45
46 XPath: substring('abcde', 10, 2)
47 Ergebnis: ''
48
49 XPath: substring('abcde', 1, 20)

```

20.6.4 Logische Funktionen

Logische Funktionen werden zur Verarbeitung boolescher Ausdrücke verwendet.

► Auflistung: Logische Funktionen ▾

- **boolean()**: Wandelt das angegebene Objekt in einen logischen Wert um. Nichtleere Kontenmengen, nichtleere String-Werte und Zahlen größer 0 ergeben wahr.
- **not()**: Ergibt wahr, wenn das Objekt falsch ist.
- **true()**: Gibt immer den Wert wahr zurück.
- **false()**: Gibt immer den Wert falsch zurück.

► Codebeispiel: Logische Funktionen ▾

```

1 // -----
2 // XPath Funktion: boolean
3 //
4 /* xs:boolean boolean(
5     items()
6 ) */
7
8 XPath: boolean(/project[@id='1'])
9 Ergebnis: false
10
11 XPath: boolean(/project[@id='343225'])
12 Ergebnis: true
13
14 XPath: boolean(string(/project/name))
15 Ergebnis: true
16
17 // -----
18 // XPath Funktion: not
19 //
20 /* xs:boolean not(
21     item()
22 ) */
23
24 XPath: not(1 = 1)
25 Ergebnis: false
26
27 XPath: not(*)
28 wahr wenn der Kontextknoten keine
29 Kindelemente hat
30
31 XPath: not(name='AI')
32 wahr wenn der Kontextknoten eine name
33 Element hat mit dem Wert AI

```

20.7. Fallbeispiel: XPath

Schreiben Sie für folgende Aufgabenstellung die gefragten **XPath** Ausdrücke.

20.7.1 Datei: browser.xml

► Codebeispiel: Eingabedatein

```

1  <!-- ----- -->
2  <!-- Beispieldokument: browser.xml -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <browser>
6    <tab id="t1" >
7      <link url="http://www.orf.at"/>
8      <content>
9        Berichterstattung
10       </content>
11      <subpage id="s1" topic="wheater">
12        <link url="wheater"/>
13        <content>
14          Das Wetter fuer Oesterreich
15         </content>
16      </subpage>
17      <subpage id="s2" topic="sport">
18        <link url="sport"/>
19        <content>
20          Unglaubliche Sportnachrichten
21         </content>
22      </subpage>
23      <sublink id="s3" topic="society"/>
24    </tab>
25    <tab id="t2">
26      <link url="http://www.wrd.de/reisen"/>
27      <content>
28        Reiseberichte mit Tamia Karens
29       </content>
30      <subpage id="s5">
31        <link url="train"/>
32        <content>
33          Zugreisen in Europa
34         </content>
35      </subpage>
36    </tab>
37    <histor>
38      <sub>s1</sub>
39      <sub>s3</sub>
40      <sub>s5</sub>
41    </history>
42  </browser>
```

20.7.2 XPath Ausdrücke

Aufgabenstellung XPath Ausdrücke:

► Codebeispiel: Lokalisierungspfade

```

1  // -----
2  //   Beispieldokument: browser.xml
3  // -----
4  1. Finden Sie alle <subpage> Elemente
5
6  /browser/subpage
7
8  // -----
9  2. Finden sie das letzte <subpage> Element
10   des t1 <tab>
11  // -----
12
13 /browser/tab[@id='t1']/subpage[last()]
14
15 // -----
16 3. Finden Sie alle <subpage> Elemente die
17   nicht in der <history> enthalten sind
18  // -----
19
20 //subpage[not(@id = //history/sub/text())]
21
22 // -----
23 4. Geben Sie die gesamte url der s1
24   <subpage> an
25  // -----
26
27 concat(//tab[@id='t1']/link/@url, '/',
28 //tab[@id='t1']/subpage[@id='s1']/link/@url)
29
30 // -----
31 5. Geben Sie das erste <tab> aus das sich mit
32   der Berichterstattung befasst.
33  // -----
34
35 //tab[contains(content,
36   'Berichterstattung')][1]
37
38 6. Wie oft wurde <subpage> s5 vom User auf-
39   gerufen
40  // -----
41 count(//sub[.= 's5'])
42
43 count(/descendant::sub[self::node() = 's5'])
```



21. Datenformat - XML XSLT

03

XSLT Stylesheets

01. XSLT Grundlagen	216
02. XSLT Transformationsprocess	217
03. XSLT Programm	218
04. Deklarative Verarbeitung	220
05. Prozedurale Verarbeitung	222
06. Ausgabestream	225
07. Suchanfragen	227

21.1. XSLT Grundlagen



XSLT Standard ▾

Der XSLT Standard ist eine XML Sprache zur **Transformation** und Verarbeitung von XML Dokumenten.

Ein XSLT Programm beschreibt Regeln zur **Transformation** von XML Daten.

21.1.1 XSLT Stylesheet

Ein XSLT Programm wird als **XSLT Stylesheet** bezeichnet.

► Erklärung: XSLT Stylesheet ▾

- Ein XSLT Stylesheet selbst ist ein XML Dokument.
- Ein Stylesheet besteht dabei aus einer freien Abfolge von **Templateregeln**.
- Templateregeln beschreiben wie die **Elemente** eines XML Eingabedokuments verarbeitet werden sollen.

► Codebeispiel: XSLT Stylesheet ▾

```

1  <!-- ----- -->
2  <!--          XSLT Stylesheet          -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <xsl:stylesheet
       xmlns:xsl="http://www.w3.org/Transform">
6
7      <xsl:template match="... ">
8
9        ...
10     </xsl:template>
11
12    <xsl:template match="... ">
13
14      ...
15    </xsl:template>
16
17    <xsl:template match="... ">
18
19      ...
20    </xsl:template>
21
22  </xsl:stylesheet>
```

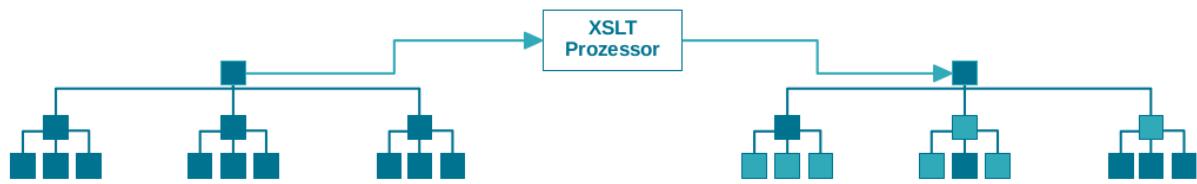


Abbildung 86. Transformation von XML Dokumenten

21.1.2 Fallbeispiel: Hello World

▶ Codebeispiel: Hello World ▾

```

1  <!-- ----- -->
2  <!--      Input: greeting.xml      -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <greeting>
6      Hello, World!
7  </greeting>
8
9  <!-- ----- -->
10 <!--     XSLT Programm: greeting.xsl    -->
11 <!-- ----- -->
12 <?xml version="1.0" encoding="UTF-8"?>
13 <xsl:stylesheet
14     xmlns:xsl="http://www.w3.org/Transform"
15     version="2.0">
16
17     <xsl:template match="greeting">
18         <html>
19             <body>
20                 <h1>
21                     <xsl:value-of select="greeting"/>
22                 </h1>
23             </body>
24         </html>
25     </xsl:template>
26
27 </xsl:stylesheet>
28
29 <!-- ----- -->
30 <!--      Output: greeting.html      -->
31 <!-- ----- -->
32 <html>
33     <body>
34         <h1>Hello, World!</h1>
35     </body>
36 </html>
```

21.2. XSLT Transformationsprozess ▾

Der XSLT Transformationsprozess besteht aus einer Folge von **Schritten**.

21.2.1 Transformationsschritte

▶ Auflistung: Transformationsschritte ▾

Initialisierungsschritt ▾

Die XSLT Engine lädt die XML **Eingabedatei** und das gewünschte **Stylesheet**.

Logische Verarbeitung ▾

Die XSLT Engine generiert für die eingelesenen XML Daten eine **Knotenbaumrepräsentation**.

Kontextschritt ▾

Um Knoten verarbeiten zu können müssen sie in den **Kontext** der XSLT Enginen geladen werden. Es gibt 2 Möglichkeiten um Knoten in den Kontext zu laden:

- Mit dem Abschluss der logischen Verarbeitung wird der Wurzelknoten der XML Eingabedatei in den **Kontext** geladen.
- Durch den Aufruf der `apply-templates` Anweisung können neue Knoten in den Kontext geladen werden.

Verarbeitungsschritt ▾

Für jeden Knoten im Kontext bestimmt die XSLT Engine eine entsprechende **Templateregel**, um den Knoten zu verarbeiten.

21.3. XSLT Programm

21.3.1 XSLT Stylesheet



XSLT Stylesheet ▾

Ein **XSLT Programm** wird als **XSLT Stylesheet** bezeichnet.

► Erklärung: XSLT Stylesheet ▾

- Ein XSLT Stylesheet selbst ist ein **XML Dokument**.
- Ein Stylesheet definiert eine freie Abfolge von **Templateregeln**. Templateregeln beschreiben wie die Elemente einer XML Eingabedatei verarbeitet werden sollen.
- Templateregeln werden in XSLT mit dem `<xsl:template>` Element definiert.

► Codebeispiel: XSLT Stylesheet ▾

```

1  <!-- ----- -->
2  <!--      XSLT Stylesheet      -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <xsl:stylesheet
6      xmlns:xsl="http://www.w3.org/Transform"
7      version="2.0">
8
9      <xsl:template match="...">
10         ...
11         <xsl:apply-template select="..."/>
12     </xsl:template>
13
14     <xsl:template match="...">
15         ...
16         <xsl:apply-template select="..."/>
17     </xsl:template>
18
19
20     <xsl:template match="...">
21         ...
22         <xsl:apply-template select="..."/>
23     </xsl:template>
24
25 </xsl:stylesheet>

```

21.3.2 Templateregel

Templateregeln sind die grundlegenden Bausteine eines XSLT Stylesheets.

► Erklärung: Templateregel ▾

- Templateregeln werden zur Verarbeitung von XML Knoten verwendet.
- Eine Templateregel bezieht sich dabei immer auf ein bestimmtes XML Element der eingelesenen XML Eingabedaten.
- Templateregeln setzen sich aus 2 Teilen zusammen: dem XSLT Suchmuster und dem Template.

► Auflistung: Komponenten einer Templateregel ▾



XSLT Suchmuster ▾

Suchmuster sind **XPath Lokalisierungspfade**. Suchmuster definieren welche Elementknoten einer Eingabedatei, mit einer Templateregel verarbeitet werden können.



Template ▾

Ein Template definiert eine freie Abfolge von **Anweisungen**. Im XSLT Template wird der eigentliche Transformationsprozess beschrieben.

21.3.3 XSLT Suchmuster

Ein XSLT Suchmuster wird durch einen **XPath Lokalisierungspfad** beschrieben.

► Erklärung: XSLT Suchmuster ▾

- Suchmuster werden im `match` Attribut eines `<xsl:template>` Elements definiert.
- Mit einem Suchmuster wird bestimmt, welche Elementknoten der Eingabedatei, mit einer Templateregel verarbeitet werden sollen.
- Damit definiert ein Suchmuster eine **Relation** zwischen den Templateregeln des XSLT Stylesheets und den XML Elementen der XML Eingabedatei.
- Für den Knotendurchlauf im XSLT Suchmuster kann nur auf die `child` bzw `descendant` Achse zurückgegriffen werden.
- Im `match` Attribut können auch mehrere Lokalisierungspfade getrennt durch ein `|` definiert werden.

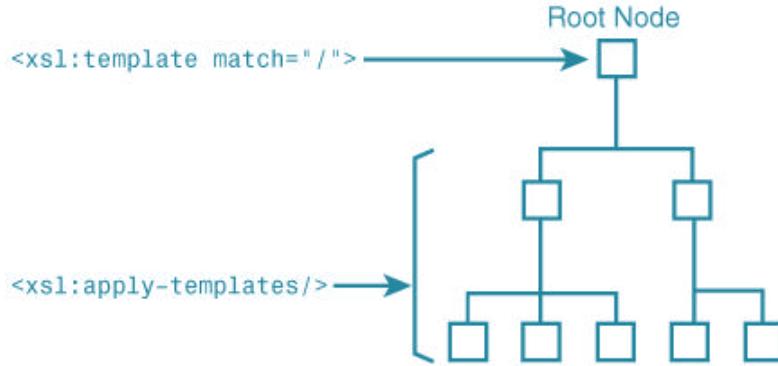


Abbildung 87. XSLT Transformationsprozess

21.3.4 Template Anweisungen

Templates enthalten Anweisungen zur Verarbeitung von XML Daten.

Templates können 2 Arten von **Anweisungen**⁴ enthalten: **XSLT Anweisungen** und **Literale**.

► Auflistung: Template Anweisungen ▾



XSLT Anweisungen ▾

XSLT Anweisungen sind Befehle die vom XSLT **Prozessor** ausgeführt werden. XSLT Anweisungen sind im Stylesheet am `xsl` Namespace erkennbar.



Literale ▾

Literale werden vom XSLT Prozessor nicht verarbeitet. Literale werden direkt in den **Ausgabestrom** des Transformationsprozesses geschrieben.

► Analyse: Template Anweisungen ▾

- **XSLT Anweisungen:** Das folgende Beispiel enthält die folgenden XSLT Anweisungen: `stylesheet`, `output`, `template`, `apply-templates` und `value-of`.
- **Literale:** Die restlichen Anweisungen sind **Literale**. Literale werden direkt in den Ausgabestrom des Transformationsprozesses geschrieben.

► Codebeispiel: Template ▾

```

1  <!-- ----- -->
2  <!--      Input: greeting.xml          -->
3  <!-- ----- -->
4  <?xml version="1.0"?>
5  <greeting>
6      Hello, World!
7  </greeting>
8
9  <!-- ----- -->
10 <!--     xslt file: greeting.xsl       -->
11 <!-- ----- -->
12 <?xml version="1.0" encoding="UTF-8"?>
13 <xsl:stylesheet
14     xmlns:xsl="http://www.w3.org/Transform"
15     version="2.0">
16
17     <xsl:output method="xml" indent="yes"/>
18
19     <xsl:template match="/">
20         <html>
21             <body>
22                 <xsl:apply-templates select="/" />
23             </body>
24         </html>
25     </xsl:template>
26
27     <xsl:template match="greeting">
28         <h1>
29             <xsl:value-of select="greeting"/>
30         </h1>
31     </xsl:template>
32
33 </xsl:stylesheet>

```

⁴ Befehl



21.4. Deklarative Verarbeitung



XSLT Transformationsprozess ▾

Der XSLT Transformationsprozess besteht im wesentlichen aus 2 sich wiederholenden **Schritten**:

- **1.Schritt:** Schreibe die zu verarbeitenden Elemente in den Kontext der XSLT Engine
- **2.Schritt:** Verarbeite die Elemente des Kontext mit Templateregeln.

21.4.1 <xsl:apply-templates> Element

Mit der **apply-templates** Anweisung werden neue XML Elemente in den Kontext der XSLT Engine geschrieben.

► Erklärung: apply-templates Anweisung ▾

- Mit der **apply-templates** Anweisung wird gesteuert, welche Elemente der Eingabedatei in den Kontext geschrieben werden sollen.
- Die **apply-templates** Anweisung wird zur Steuerung des **Transformationsprozesses** verwendet.

► Syntax: <xsl:apply-templates> ▾

```

1  <!-- ----- -->
2  <!--   Syntax: <xsl:apply-templates>   -->
3  <!-- ----- -->
4  <xsl:apply-templates
5      select = Knotenmengenausdruck
6      mode   = QName>
7  </xsl:apply-templates>
8
9  <!-- ----- -->
10 <!--  XSLT: apply-templates Anweisung   -->
11 <!-- ----- -->
12 <?xml version="1.0" encoding="UTF-8"?>
13 <xsl:stylesheet
14     xmlns:xsl="http://www.w3.org/Transform">
15
16     <xsl:template match="... ">
17         ...
18         <!-- bind parameter to element -->
19         <xsl:apply-templates select="... "/>
20             <xsl:with-param name="type">
21                 action
22             </xsl:with-param>
23         </xsl:apply-templates>
24     </xsl:template>
25
26     <!-- Declare template parameter -->
27     <xsl:template match="... ">
28         <xsl:param name="type"/>
29     </xsl:template>
30 </xsl:stylesheet>
```

21.4.2 <xsl:template> Element

Ein XSLT Stylesheet ist eine freie Abfolge von **Templateregeln**.

► Erklärung: Templateregeln ▾

- Templateregeln werden in XSLT mit dem **<xsl:template>** Element definiert.
- Stellt die XSLT Engine für einen Knoten im XSLT Kontext eine Übereinstimmung mit dem **Suchmuster** einer Templateregel fest, wird die entsprechende Templateregel geladen, um den Knoten zu verarbeiten.
- Die Templateregel selbst hat nun Zugriff auf alle Daten des Knotens zusammen mit allen Parametern die im **<xsl:apply-templates>** Element definiert worden sind.

► Syntax: <xsl:template> ▾

```

1  <!-- ----- -->
2  <!--   Syntax: <xsl:template>   -->
3  <!-- ----- -->
4  <xsl:template
5      match = Knotenmengenausdruck
6      mode  = Zeichenkette>
7  </xsl:template>
8
9  <!-- ----- -->
10 <!--  XSLT: param Anweisung   -->
11 <!-- ----- -->
12 <?xml version="1.0" encoding="UTF-8"?>
13 <xsl:stylesheet
14     xmlns:xsl="http://www.w3.org/Transform">
15
16     <xsl:template match="... ">
17         ...
18         <!-- bind parameter to element -->
19         <xsl:apply-templates select="... "/>
20             <xsl:with-param name="type">
21                 action
22             </xsl:with-param>
23         </xsl:apply-templates>
24     </xsl:template>
25
26     <!-- Declare template parameter -->
27     <xsl:template match="... ">
28         <xsl:param name="type"/>
29     </xsl:template>
30 </xsl:stylesheet>
```

21.4.3 `<xsl:with-param>` Element

Mit der `with-param` Anweisung können zusätzliche Informationen mit den Knoten im Kontext der XSLT Engine definiert werden.

► Erklärung: `with-param` Anweisung ▾

- Durch das einbetten der `with-param` Anweisung in `<xsl:apply-templates>` Elementen können zusätzliche Informationen mit Elementen im Knotext assoziiert werden.
- Bei der Verarbeitung dieser Elements, hat die entsprechende Templateregel nun Zugriff auf die mit der `with-param` Anweisung definierten Parameter.

► Syntax: `<xsl:with-param>` ▾

```

1  <!-- ----- -->
2  <!-- Syntax: <xsl:with-param>      -->
3  <!-- ----- -->
4  <xsl:with-param name = QName
5    select = Knotenmengenausdruck>
6  </xsl:with-param>
7
8  <!-- ----- -->
9  <!-- XSLT: with-param      -->
10 <!-- ----- -->
11 <?xml version="1.0" encoding="UTF-8"?>
12 <xsl:stylesheet
13   xmlns:xsl="http://www.w3.org/Transform">
14   <xsl:template match="...">
15     ...
16     <!-- bind parameter to element -->
17     <xsl:apply-templates select="..."/>
18     <xsl:with-param name="type">
19       action
20     </xsl:with-param>
21     </xsl:apply-templates>
22   </xsl:template>
23
24   <xsl:template match="...">
25     <!-- fetch value from context -->
26     <!-- and bind it to param -->
27     <xsl:param name="type"/>
28
29     <!-- work with value -->
30     <xsl:value-of select="$type"/>
31   </xsl:template>
32
33 </xsl:stylesheet>

```

21.4.4 `xsl:mode` Attribut

XSLT Stylesheets können mehrere **Templateregeln** mit demselben XSLT **Suchmustern** enthalten.

Templateregeln mit dem gleichen Suchmuster sind für die XSLT Engine nicht differenzierbar.

► Erklärung: `Modaler Aufruf` ▾

- Der XSLT Prozessor unterscheidet Templateregeln durch den Vergleich ihrer Suchmuster.
- Besitzen 2 oder mehrere Templateregeln dasselbe Suchmuster versucht die XSLT Engine die Templates durch den Vergleich der `mode` Attribute zu differenzieren.
- Das `xsl:mode` Attribut muß dabei sowohl für das `<xsl:apply-templates>` Element als auch dem `<xsl:template>` Element gesetzt sein.

► Codebeispiel: Template modi ▾

```

1  <!-- ----- -->
2  <!-- XSLT: Template Modi      -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8" ?>
5  <xsl:stylesheet
6    xmlns:xsl="http://www.w3.org/Transform">
7
8    <xsl:template match="/">
9      <!-- Verarbeitung durch Template A -->
10     <xsl:apply-templates
11       select="//movie[1]"
12       mode="premiere"/>
13
14      <!-- Verarbeitung durch Template B -->
15      <xsl:apply-templates select="//movie"/>
16    </xsl:template>
17
18    <!-- Template A -->
19    <xsl:template match="movie"
20      mode="premiere">
21      ...
22    </xsl:template>
23
24    <!-- Template B -->
25    <xsl:template match="movie">
26      ...
27    </xsl:template>
28
29  </xsl:stylesheet>

```

21.5. Prozedurale Verarbeitung ▾

Die XSLT Spezifikation definiert eine Reihe von Anweisungen zur **prozeduralen Verarbeitung** von XML Knoten.

21.5.1 <xsl:variable> Element

Für die Verarbeitung von Knoten unterstützt XSLT die Verwendung von Variablen.

► Erklärung: Variablen Deklaration ▾

- Die XSLT Spezifikation definiert **Variablen** jedoch nur in einem sehr eingeschränkten Sinn.
- XSLT **Variablen** kann zwar ein Wert **zugewiesen** werden, der Wert der Variable kann im Laufe der weiteren Verarbeitung jedoch nicht geändert werden.
- Der Geltungsbereich einer Variable beschränkt sich dabei auf die Templateregel, indem sie definiert wurde.

► Syntax: <xsl:variable> ▾

```

1  <!-- ----- -->
2  <!--   Syntax: <xsl:variable>      -->
3  <!-- ----- -->
4  <xsl:variable name = QName select = Ausdruck>
5  <!-- Inhalt: Template -->
6  </xsl:variable>
7
8  <!-- ----- -->
9  <!--   XSLT: variable Anweisung      -->
10 <!-- ----- -->
11 <?xml version="1.0" encoding="UTF-8" ?>
12 <xsl:stylesheet version="2.0">
13
14    <xsl:template match="...">
15      <xsl:value-of select="@id"/>
16      ...
17      <xsl:variable name="code"
18        select="'java'"/>
19
20      <!-- Stringdarstellung -->
21      <xsl:value-of select="$code"/>
22    </xsl:template>
23
24  </xsl:stylesheet>

```



21.5.2 <xsl:value-of> Element

Mit der **value-of** Anweisung kann die **Stringdarstellung** von Variablen, Parametern bzw. Knoten ermittelt werden.

► Syntax: <xsl:value-of> ▾

```

1  <!-- ----- -->
2  <!--   Syntax: <xsl:value-of>      -->
3  <!-- ----- -->
4  <xsl:value-of select = Ausdruck
5    disable-output-escaping = "yes" | "no"
6 </xsl:value-of>
7
8  <!-- ----- -->
9  <!--   XSLT: value-of Anweisung      -->
10 <!-- ----- -->
11 <?xml version="1.0" encoding="UTF-8" ?>
12 <xsl:stylesheet version="2.0">
13
14    <xsl:template match="...">
15      <xsl:value-of select="@id"/>
16      ...
17      <xsl:variable name="code"
18        select="'java'"/>
19
20      <!-- Stringdarstellung -->
21      <xsl:value-of select="$code"/>
22    </xsl:template>
23
24  </xsl:stylesheet>

```



21.5.3 <xsl:if> Element

Die **if** Anweisung ermöglicht den Ablauf der Verarbeitung von Anweisungen innerhalb einer **Templateregel** zu steuern.

► Erklärung: if Anweisung ▾

- Zur Formulierung von Bedingung wird dem im **<xsl:if>** Element definiertem **test** Attribut ein logischer Ausdruck zugeordnet.
- Kann der logische Ausdruck zu true evaluiert werden, werden die im **<xsl:if>** Element enthaltenen Anweisungen ausgeführt.
- Für Logische Ausdrücke gelten dabei dieselben syntaktischen Einschränkungen wie für Prädikate in XPath Ausdrücken.



► Erklärung: Auswertung logischer Ausdrücke ▾

- **Number:** Die numerischen Werte 0 bzw. NaN evaluieren zu **false**. Alle anderen Werte evaluieren zu **true**.
- **node-set:** Leere **node-set** evaluieren zu **false**. **node-set** mit Elementen evaluieren zu **true**.
- **String:** Ein leerer String evaluiert zu **false**. Alle anderen Strings evaluieren zu **true**.

► Syntax: <xsl:if> ▾

```

1  <!-- ----- -->
2  <!-- Syntax: <xsl:if> -->
3  <!-- ----- -->
4  <xsl:if test = "logischer Ausdruck">
5  <!-- Inhalt: Template -->
6  </xsl:if>
7
8  <!-- ----- -->
9  <!-- XSLT: if Anweisung -->
10 <!-- ----- -->
11 <?xml version="1.0" encoding="UTF-8"?>
12 <xsl:stylesheet version="2.0">
13
14   <xsl:template match="...">
15     <xsl:variable name="code"
16       select="'java'"/>
17
18     <xsl:variable name="students"
19       select="/students"/>
20
21     <!-- Vergleich von Zeichenketten -->
22     <xsl:if test="@id = $code">
23
24       ...
25     </xsl:if>
26
27     <!-- Test auf Knotenmengen -->
28     <xsl:if test="$students">
29
30       ...
31     </xsl:if>
32
33   </xsl:template>
34
35 </xsl:stylesheet>
```

21.5.4 <xsl:choose> Element

Neben dem <xsl:if> Element definiert die XSLT Spezifikation das <xsl:choose> Element zur Steuerung des Programmflusses.

► Erklärung: choose Anweisung ▾

- Im <xsl:choose> XML Element können mehrere <xsl:when> Elemente eingebettet werden.
- Evaluierter im **test** Attribut einer **when** Anweisung definierte logische Ausdruck zu **true**, werden die in der Anweisung engebetteten Elemente ausgeführt.
- Es wird jedoch nur die erste **when** Anweisung ausgeführt, deren Bedingung zu **true** evaluiert.
- Zusätzlich kann mit dem <xsl:otherwise> Element ein Default Pfad definiert. Trifft keine der angegebenen Bedingungen zu verzweigt der Programmfluss in den **Defaultzweig**.

► Syntax: <xsl:choose> ▾

```

1  <!-- ----- -->
2  <!-- Syntax: <xsl:choose> -->
3  <!-- ----- -->
4  <xsl:choose>
5    <xsl:when test = "logischer Ausdruck">
6      <!-- Inhalt: Template -->
7      </xsl:when>
8      <xsl:otherwise>
9        <!-- Inhalt: Template -->
10     </xsl:otherwise>
11   </xsl:choose>
12
13 <!-- ----- -->
14 <!-- XSLT: choose Anweisung -->
15 <!-- ----- -->
16 <xsl:template match="table-row">
17   <xsl:choose>
18     <xsl:when test="position() = 0">
19       <xsl:text>papaywhip</xsl:text>
20     </xsl:when>
21     <xsl:when test="position() = 1">
22       <xsl:text>mintcream</xsl:text>
23     </xsl:when>
24     <xsl:otherwise>
25       <xsl:text>whitesmoke</xsl:text>
26     </xsl:otherwise>
27   </xsl:choose>
28 </xsl:template>
```

21.5.5 <xsl:for-each> Element

Zur Verarbeitung von Knotenmengen stellt die XSLT Spezifikation das <xsl:for-each> Element zur Verfügung.

► Erklärung: Iterative Verarbeitung ▾

- Für die iterative Verarbeitung eines **node-sets** stellt XSLT das <xsl:for-each> Element zur Verfügung.
- In jedem **Iterationsschritt** der Schleife wird eines der Elemente des **node-sets** verarbeitet.
- Das <xsl:for-each> Element ist dabei neben dem <xsl:apply-templates> Element die einzige Anweisung, die den Kontext des XSLT Prozessors verändern kann.

► Codebeispiel: for-each Anweisung ▾

```

1  <!-- ----- -->
2  <!-- XSLT: for-each Anweisung -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <xsl:stylesheet version="2.0">
6
7      <!-- Datenformat der Ausgabe -->
8      <xsl:output method="html"/>
9
10     <xsl:template match="/">
11         <cinema>
12             <!-- iterative Verarbeitung -->
13             <xsl:for-each
14                 select="movies/movie">
15                 <xsl:if test="imdbRating[.
16                     &gt; 8]">
17                     <screening>
18                         <title>
19                             <xsl:value-of
20                             select="title"/>
21                         </title>
22                         <link>
23                             <xsl:value-of
24                             select="@url"/>
25                         </link>
26                     </screening>
27                 </xsl:if>
28             </xsl:for-each>
29         </cinema>
30     </xsl:template>
31
32 </xsl:stylesheet>
```

21.5.6 <xsl:sort> Element

Mit dem <xsl:sort> Element können die Knoten einer Knotemenge **sortiert** werden.

► Erklärung: Sortieren von Knotenmengen ▾

- Das <xsl:sort> Element kann dabei als Kind-Element des <xsl:apply-templates> bzw. des <xsl:for-each> Elements auftreten.
- Um eine Knotenmenge nach mehreren Kriterien zu sortieren, wird das <xsl:sort> Element wiederholt eingebettet.

► Syntax: <xsl:sort> ▾

```

1  <!-- ----- -->
2  <!-- Syntax: <xsl:sort> -->
3  <!-- ----- -->
4  <xsl:sort
5      select = Knotenmenge
6      order = "ascending" | "descending"
7 </xsl:sort>
8
9  <!-- ----- -->
10 <!-- XSLT: sort Anweisung -->
11 <!-- ----- -->
12 <?xml version="1.0" encoding="UTF-8"?>
13 <xsl:stylesheet version="2.0">
14
15      <!-- Datenformat der Ausgabe -->
16      <xsl:output method="html"/>
17
18      <xsl:template match="/">
19          <project-staff>
20              <xsl:for-each select="//employee">
21                  <xsl:sort
22                      select="name/last-name"
23                      order="descending"/>
24                  <xsl:sort
25                      select="name/first-name"
26                      order="descending"/>
27                      <last-name>
28                          <xsl:value-of
29                          select="name/last-name"/>
30                      </last-name>
31                      <first-name>unknown</first-name>
32                  </xsl:for-each>
33              </project-staff>
34          </xsl:template>
35
36 </xsl:stylesheet>
```

21.6. Ausgabestream

Die XSLT Spezifikation definiert eine Reihe von Anweisungen zum Schreiben von Daten in den **Ausgabestream**.

21.6.1 <xsl:element> Element

Die **element** Anweisung wird verwendet um ein XML Element in den Ausgabestream zu schreiben.

► Syntax: <xsl:element>

```

1  <!-- ----- -->
2  <!--      Syntax: <xsl:element>      -->
3  <!-- ----- -->
4  <xsl:element
5      name          = QName
6      namespace = URI Referenz
7      use-attribute-set = QName>
8  <!--   Inhalt: Template   -->
9  </xsl:element>

10 <!-- ----- -->
11 <!--      XSLT: element Anweisung      -->
12 <!-- ----- -->
13 <?xml version="1.0" encoding="UTF-8" ?>
14 <xsl:stylesheet version="2.0">

15
16
17     <xsl:template match="/">
18         <kursprogramm>
19             <xsl:apply-templates select="kurs"/>
20         </kursprogramm>
21     </xsl:template>

22
23     <xsl:template match="kurs">
24         <kurs>
25             <xsl:for-each select="@*">
26                 <xsl:element name="{name()}>
27                     <xsl:value-of select=". "/>
28                 </xsl:element>
29             </xsl:for-each>
30         </kurs>
31     </xsl:template>

32
33     <xsl:template match="... ">
34         ...
35
36
37     </xsl:template>
38
39 </xsl:stylesheet>

```

21.6.2 <xsl:text> Element

Die **text** Anweisung wird verwendet um Textzeichen in den Ausgabestream zu schreiben.

► Syntax: <xsl:text>

```

1  <!-- ----- -->
2  <!--      Syntax: <xsl:text>      -->
3  <!-- ----- -->
4  <xsl:text
5      disable-output-escaping = "yes" |
6      "no">
7  <!-- Inhalt: PCDATA -->
8  </xsl:text>

```

21.6.3 <xsl:attribute> Element

Die **attribute** Anweisung wird verwendet um **Attribute** in den Ausgabestream zu schreiben.

► Erklärung: attribute Anweisung

- In ein <xsl:element> Element können mehrere <xsl:attribute> Elemente eingebettet werden.
- Beim Generieren des Ausgabestroms wird ein entsprechendes Attribut dem übergeordneten XML Element hinzugefügt.

► Syntax: <xsl:attribute>

```

1  <!-- ----- -->
2  <!--      Syntax: <xsl:attribute>      -->
3  <!-- ----- -->
4  <xsl:attribute
5      name          = QName
6      namespace = URI Referenz>
7  <!--   Inhalt: Template   -->
8  </xsl:attribute>

9
10 <!-- ----- -->
11 <!--      XSLT: attribute Anweisung      -->
12 <!-- ----- -->
13 <xsl:stylesheet version="2.0">
14     <xsl:template match="kurs">
15         <kurs>
16             <xsl:attribute name="nr">
17                 <xsl:value-of select="position()"/>
18             </xsl:attribute>
19         </kurs>
20     </xsl:template>
21 </xsl:stylesheet>

```

21.6.4 <xsl:attribute-set> Element

Teilen sich mehrere Elemente dieselben Attribute kann eine gemeinsame Attributliste für die entsprechenden Elemente definiert werden.

► Erklärung: attribute-set Anweisung ▾

- In ein <xsl:attribute-set> Element können mehrere <xsl:attribute> Elementen eingebettet werden.
- XML Elemente können Attributsets über das **xsl:use-attribute-sets** Attribut referenzieren.
- Für Elemente die ein Attributset referenzieren werden die entsprechenden Attribute im Ausgabestream generiert.

► Syntax: <xsl:attribute-set> ▾

```

1  <!-- ----- -->
2  <!-- Syntax: <xsl:attribute-set> -->
3  <!-- ----- -->
4  <xsl:attribute-set
5      name          = QName
6      use-attribute-sets = QName
7      namespace = URI Referenz>
8  <!-- Inhalt: <xsl:attribute> -->
9  </xsl:attribute-set>
10 <!-- ----- -->
11 <!-- Beispiel: <xsl:attribute> -->
12 <!-- ----- -->
13 <!-- ----- -->
14 <?xml version="1.0" encoding="UTF-8" ?>
15 <xsl:stylesheet version="2.0">
16
17   <xsl:template match="/">
18     <xsl:attribute-set name="programm">
19       <xsl:attribute name="id">
20         3235
21       </xsl:attribute>
22       <xsl:attribute name="day">
23         Monday
24       </xsl:attribute>
25     </xsl:attribute>
26
27     <kursprogramm
28       xsl:use-attribute-sets="programm">
29       <xsl:apply-templates select="kurs"/>
30     </kursprogramm>
31   </xsl:template>
32 </xsl:stylesheet>
```

21.6.5 <xsl:copy-of> Element

Die <xsl:copy-of> Anweisung kopiert die durch einen XPath Ausdruck definierte **Knotenmenge** in die Ausgabedatei.

► Syntax: <xsl:copy-of> ▾

```

1  <!-- ----- -->
2  <!-- Syntax: <xsl:copy-of> -->
3  <!-- ----- -->
4  <xsl:copy-of select = XPath>
5  <!-- kein Inhalt -->
6  </xsl:copy-of>
```

► Codebeispiel: <xsl:copy-of> ▾

```

1  <!-- ----- -->
2  <!-- Beispiel: <xsl:copy-of> -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8" ?>
5  <xsl:stylesheet version="2.0">
6
7    <xsl:template match="/">
8      <xsl:for-each select="kurs">
9        <info>
10          <xsl:attribute name="nr">
11            <xsl:value-of
12              select="position()"/>
13            </xsl:attribute>
14            <xsl:copy-of select=". "/>
15          </info>
16        </xsl:for-each>
17      </xsl:template>
18
19    </xsl:stylesheet>
```

21.6.6 <xsl:output> Element

Die <xsl:output> Anweisung definiert das **Datenformat** der Ausgabedatei.

► Syntax: <xsl:output> ▾

```

1  <!-- ----- -->
2  <!-- Syntax: <xsl:output> -->
3  <!-- ----- -->
4  <xsl:output
5      method = "xml" | "html" | "text"
6      indent = "yes" | "no"
7      >
8  <!-- kein Inhalt -->
9  </xsl:output>
```

21.7. Suchanfragen

Das **Suchmuster** einer **Templateregel** definiert welche **XML Elementknoten** ein Template verarbeiten kann.

21.7.1 Auflösen von Templatekonflikten

Mit Xslt ist es möglich für denselben **Xml Elementknoten** mehrere **Templateregeln** zu definieren.

► Codebeispiel: Templatekonflikte

```

1  <!-- ----- -->
2  <!--      Templatekonflikte      -->
3  <!-- ----- -->
4  <xsl:stylesheet version="2.0">
5
6      <xsl:template match="/">
7          <cinema>
8              <xsl:apply-templates
9                  select="movies/movie" />
10             </cinema>
11         </xsl:template>
12
13     <xsl:template match="movie">
14         <screening>
15             <xsl:value-of select="title"/>
16         </screening>
17     </xsl:template>
18
19     <xsl:template match="movies/movie">
20         <premiere>
21             <xsl:value-of select="title"/>
22         </premiere>
23     </xsl:template>
24 </xsl:stylesheet>
```

► Erklärung: Templatekonflikte

- Kommen mehrere Templateregeln für die **Verarbeitung** eines Xml Elements in Frage, treten bestimmte **Prioritätsregeln** in Kraft.
- Spezifischere Regeln haben Vorrang vor allgemeineren Regeln. z.B.: `/movies/movie` ist spezifischer als `movie`.
- Suchanfragen mit Wildcard z.B.: `*` oder `@*` sind allgemeiner als entsprechende Muster ohne Wildcards.
- Wenn keines der aufgeführten Kriterien zum Tragen kommt, hat die zuletzt stehende Regel Vorrang.

21.7.2 Default Templates

Die Verarbeitung eines Xslt Stylesheets beginnt stets mit dem **Wurzelknoten** des Eingabedokuments.



Default Template

Default Templates sind implizit im Xslt Prozessor definierte Templates.

► Erklärung: Default Templates

- Der **Xslt Prozessor** lädt als erstes immer die Templateregel zur Verarbeitung des **Wurzelknotens** des Eingabedokuments.
- Es spielt deshalb keine Rolle, in welcher **Reihenfolge** Templateregeln in einem Stylesheet eingetragen sind, der Prozessor sucht immer zunächst nach einer Templateregel, die **explizit** für den **Wurzelknoten** definiert worden ist oder sich dafür verwenden lässt.
- Findet er keine solche Templateregel nutzt er eine entsprechende eingebaute Templateregel.
- Eingebaute Templateregeln werden als **Default Templates** bezeichnet.



21.7.3 Roottemplate

Wenn eine **Templateregel** fehlt, die sich **explizit** auf den Wurzelknoten bezieht, verwendet der Prozessor automatisch das **Roottemplate**.

► Codebeispiel: Roottemplate

```

1  <!-- ----- -->
2  <!-- Defaulttemplate: Roottemplate      -->
3  <!-- ----- -->
4  <xsl:template match="*|/*">
5      <xsl:apply-templates/>
6  </xsl:template>
```

► Erklärung: Roottemplate

- Wir können das testen, indem wir ein Stylesheet auf eine Quelldatei anwenden, das keine Templateregeln enthält.
- Der Prozessor gibt dann einfach die **Stringwerte** aller Elemente des **Eingabedokuments** aus, wobei Attribute ignoriert werden.



Das **Roottemplate** wird jedoch nicht nur auf den Wurzelknoten, sondern auf jedes beliebige Element angewandt.

► Analyse: Roottemplate ▾

- Das **Roottemplate** sorgt damit für eine rekursive Verarbeitung, also dafür, dass, wenn eine **Templateregel** für bestimmte XML Elemente existiert, diese auch ausgeführt wird, auch wenn ihre Anwendung nicht explizit aufgerufen wird.

► Codebeispiel: Roottemplate ▾

```

1  <!-- ----- -->
2  <!-- Stylesheet: movie1.xsl -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8" ?>
5  <xsl:stylesheet version="2.0">
6      <xsl:output method="xml" indent="yes"/>
7  </xsl:stylesheet>
8
9  <!-- ----- -->
10 <!-- Ausgabe: movies1.xml -->
11 <!-- ----- -->
12 <?xml version="1.0" encoding="UTF-8"?>
13     Die Verurteilten
14     1994
15     9.2
16         Tim Robbins
17         Morgan Freeman
18
19     Der Pate
20     1972
21     9.2
22         Marlon Brando
23         Al Pacino
24
25     Twilight Imperium
26     1985
27     5.0
28         Al Pacino
29         Robert De Niro
30
31     <!-- ----- -->
32     <!-- Stylesheet: movie2.xsl -->
33     <!-- ----- -->
34     <?xml version="1.0" encoding="UTF-8" ?>
35     <xsl:stylesheet version="2.0">
36
37         <xsl:template match="movie">
38             <movies/>
39         </xsl:template>
40
41     </xsl:stylesheet>
42
43     <!-- ----- -->
44     <!-- Ausgabe: movies2.xml -->
45     <!-- ----- -->
46     <?xml version="1.0" encoding="UTF-8"?>
47         <movies/>
48         <movies/>
49         <movies/>
50

```

► Codebeispiel: Roottemplate ▾

```

1  <!-- ----- -->
2  <!--      Stylesheet: movie1.xsl      -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8" ?>
5  <xsl:stylesheet version="2.0">

6
7      <xsl:template match="title">
8          <title>
9      </xsl:template>

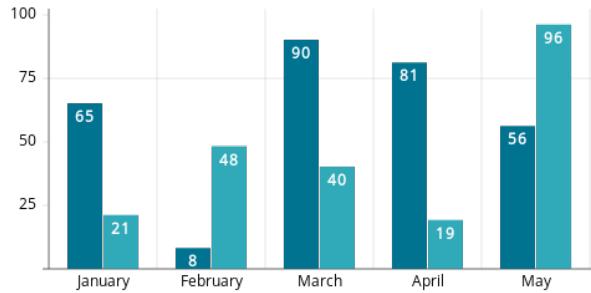
10 </xsl:stylesheet>

12 <!-- ----- -->
13 <!--      Ausgabe: movies1.xml      -->
14 <!-- ----- -->
16 <?xml version="1.0" encoding="UTF-8"?
    <title/>
    1994
    9.2
        Tim Robbins
        Morgan Freeman
    <title/>
    1972
    9.2
        Marlon Brando
        Al Pacino
    <title/>
    1985
    5.0
        Al Pacino
        Robert De Niro

```

► Erklärung: Stringtemplate ▾

- Das **Stringtemplate** kopiert den Text aller **Text-** und **Attributknoten** in das Ausgabedokument.
- Das **Stringtemplate** wird nur angewandt, wenn der Knoten ausgewählt worden ist.
- Alle Default Templates haben eine niedrigere **Priorität** als ein im Stylesheet definiertes Template.



► Codebeispiel: Roottemplate ▾

```

1  <!-- ----- -->
2  <!--      Stylesheet: movie1.xsl      -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <xsl:stylesheet version="2.0">

6
7      <xsl:output method="xml" indent="yes"/>

8
9      <xsl:template match="movie">
10         <xsl:apply-templates select="title"/>
11     </xsl:template>

12
13      <xsl:template match="movie">
14         <xsl:apply-templates select="title"/>
15     </xsl:template>

16
17      <xsl:template match="movie">
18         <xsl:apply-templates select="title"/>
19     </xsl:template>

21 </xsl:stylesheet>

23 <!-- ----- -->
24 <!--      Ausgabe: movies1.xml      -->
25 <!-- ----- -->
26 <?xml version="1.0" encoding="UTF-8"?>
27     Die Verurteilten
28     Der Pate
29     Twilight Imperium

```

21.7.4 Stringtemplate

Das **Stringtemplate** wird immer dann durch den Xslt Prozessor ausgeführt, wenn ein XML Element ausgewählt wird, es aber keine **Templateregel** für seine Verarbeitung gibt.

► Codebeispiel: Stringtemplate ▾

```

1  <!-- ----- -->
2  <!--      Defaulttemplate: Stringtemplate -->
3  <!-- ----- -->
4  <xsl:template match="text()|@*/">
5      <xsl:value-of select=". />
6  </xsl:template>

```

22. Datenformat - XML Schema

02

XML - Schema

01. Schemagrundlagen	230
02. Struktur: XML Schema	231
03. Einfache Datentypen	233
04. Komplexe Datentypen	235
05. Attribute definieren	238
06. Sichtbarkeitskonzepte	239

22.1. Schema Grundlagen

22.1.1 Inhaltsmodell

So wie sich ein Dokument in Einleitung, Hauptteil und Nachwort gliedern lässt, so kann für die Struktur eines Xml Dokuments ein Inhaltsmodell definiert werden.

Das Inhaltsmodell eines Xml Dokuments beschreibt welche Elemente in einem XML Dokument auftreten dürfen.

► Erklärung: Inhaltsmodell ▾

- Damit eine Anwendung die **Gültigkeit** eines Dokuments prüfen kann, muß die Beschreibung des **Inhaltsmodells** selbst in einer maschinell lesbaren Sprache definiert sein.
- Die Beschreibung eines Inhaltsmodells in maschiniell lesbarer Form wird als **Schema** bezeichnet.
- Xml Schemas stellen eine Schemadefinition für XML Dokumente dar.



Inhaltsmodell ▾

Ein Inhaltsmodell beschreibt die **logische Struktur** eines Xml Dokuments.



XML Schema ▾

Xml Schema beschreiben **Inhaltsmodelle** für Xml Dokumente in maschinell lesbarer Form.

► Codebeispiel: XML Schema ▾

```

1  <!-- ----- -->
2  <!--      Schemadefinition      -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <xss:schema xmlns:xss="http://... /XMLSchemax>
6  <xss:element name="person" type="xss:string"/>
7  </xss:schema>
8
9  <!-- ----- -->
10 <!--       XML Dokument       -->
11 <!-- ----- -->
12 <?xml version="1.0" encoding="UTF-8"?>
13 <person>Max Mustermann</person>
```



22.1.2 Validierung

Zur Prüfung der **Gültigkeit** eines XML Dokuments, validiert der XML Prozessor das Dokument gegen das dem Inhaltsmodell des Dokuments zugeordneten Schemas.



Validierung ▾

Die Prüfung eines XML Dokuments auf **Gültigkeit** wird als Validierung bezeichnet.

► Erklärung: Validierung ▾

- XML Schemas definieren **Regeln**, die die Struktur von XML Dokumenten beschreiben.
- Die im XML Schemas definierten Regeln, müssen bei der Erstellung eines XML Dokuments eingehalten werden.
- Bevor ein XML Dokument validiert werden kann, wird es auf **Wohlgeformtheit** geprüft. Ein wohlgeformtes XML Dokument kann dann gegen die im XML Schema definierten Regeln validiert werden.
- Erfüllt ein XML Dokument die Regeln des XML Schemas, wird es als **Instanz** des Schemas bezeichnet.



22.1.3 XML Schema Regeln

XML Schemas definieren Regeln, die die **Struktur** von XML Dokumenten beschreiben.

► Analyse: Regelausprägung ▾

- XML Schemas definieren welche Elemente, in einem XML Dokument enthalten sein dürfen.
- XML Schemas definieren welche Attribute ein XML Element haben kann.
- XML Schemas definieren welche Kindelemente ein XML Element enthalten kann.
- Mit einem XML Schema kann die Reihefolge in welcher die Kindelemente eines Elements auftreten, festgelegt werden.

22.2. Struktur: XML Schemas

■ Ein XML Schema selbst ist ein **XML Dokument**.

22.2.1 Fallbeispiel: XML Schema

Das folgende Beispiel zeigt ein **XML Schema** für ein einfaches XML Dokument.

► Codebeispiel: XML Schema ▾

```
1  <!-- ----- -->
2  <!--      XML Schema: Person           -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <xs:schema
6    xmlns:xs="http://www.w3.org/2001/XMLSchema">
7    <xs:element name="person">
8      <xs:complexType>
9        <xs:sequence>
10          <xs:element name="name"
11            type="xs:string"/>
12        </xs:sequence>
13      </xs:complexType>
14    </xs:element>
15  </xs:schema>
16
17  <!-- ----- -->
18  <!--      XML Dokument: person.xml   -->
19  <!-- ----- -->
20  <?xml version="1.0" encoding="UTF-8"?>
21  <person>
22    <name>Kauer Alexander</name>
23  </person>
```

22.2.2 Aufbau eines XML Schemas

Das **xs:element** Element ist das grundlegende Element der XML Schema Spezifikation. Mit dem Element wird einem XML Element ein Datentyp zugeordnet.

► Codebeispiel: <xs:element> ▾

```
1  <!-- ----- -->
2  <!--      XML Schema: <xs:element/>     -->
3  <!-- ----- -->
4  <xs:element name="age" type="xs:int"/>
```

Datentyp	Beschreibung	Beispiel
string	Zeichenkette	Helmut Meier
token	Zeichenkette	Helmut Meier
language	ISO639-1 Kürzel einer Sprache	de, tr, en
integer	Ganze Zahl	-2143, -3, 0 , 354
positiveInteger	positive ganze Zahl	432
negativeInteger	negative ganze Zahl	-323
nonNegativeInteger	positive ganze Zahl	323
nonPositiveInteger	negative ganze Zahl	-323
byte	8 bit Integer mit Vorzeichen	-1, 124
short	16 bit Integer mit Vorzeichen	23232
int	32 bit Integer mit Vorzeichen	3234
long	64 bit Integer mit Vorzeichen	2323
decimal	gebrochene Dezimalzahl	-1.23, 123.4, 100.0
float	Fließkommazahl	-INF, -1E4, -12.3
double	Fließkommazahl	-INF, -1E4, -12.3
time	Uhrzeit, ggf. mit Zeitzone	11:29:00.000
date	Datum (ISO8601 Format)	24-02-15
gMonth	Monat (ISO8601 Format)	-05-
gYear	Jahr (ISO8601 Format)	1978
gDay	Tag (ISO8601 Format)	—31
gMonthDay	Zeitangabe (ISO8601 Format)	-05-31
gYearMonth	Zeitangabe (ISO8601 Format)	1999-02
datetime	Zeitstempel	1004-02-15T09:00:00
boolean	logischer Wert	true, false, 0, 1
anyURI	URI	http://www.htlkrems.at.ac/welcome

Abbildung 88. Native Datentypen

22.2.3 Kategorien von Datentypen

Die **Xml Schemas Spezifikation** unterscheiden 2 Arten von Datentypen.

► Auflistung: Arten von Datentypen ▾



Einfache Datentyp ▾

Xml Elemente mit **unstrukturiertem** Inhalt haben einen einfachen Datentyp.



Komplexer Datentyp ▾

Xml Elemente mit einem **strukturierten** Inhalt haben einen komplexen Datentyp.

► Codebeispiel: Datentypen ▾

```

1  <!-- ----- -->
2  <!-- Element mit strukturiertem Inhalt -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <person>
6      <name>Max Mustermann</name>
7  </person>
8
9  <?xml version="1.0" encoding="UTF-8"?>
10 <xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
11     <xs:element name="person">
12         <xs:complexType>
13             <xs:sequence>
14                 <xs:element name="name"
15                     type="xs:string"/>
16             </xs:sequence>
17         </xs:complexType>
18     </xs:element>
19 </xs:schema>
20
21 <!-- ----- -->
22 <!-- Element mit unstrukturiertem Inhalt -->
23 <!-- ----- -->
24 <?xml version="1.0" encoding="UTF-8"?>
25 <person>Max Mustermann</person>
26
27 <xs:schema xmlns:xs="http://.../XMLSchema">
28     <xs:element name="person"
29         type="xs:string"/>
</xs:schema>
```

22.3. Einfache Datentypen

Einfache Datentypen der Xml Schema Spezifikation sind vergleichbar mit den einfachen Datentypen von Programmiersprachen wie Java.

XML Elemente mit unstrukturiertem Inhalt und keinen Attributen haben einen einfachen Datentyp.

22.3.1 Einfache XML Elemente

Die Xml Schema Spezifikation definiert 2 Arten von einfachen Datentypen.

► Auflistung: Arten einfacher DatenTypen ▾



Native Datentypen ▾

Die XML Schema Spezifikation definiert eine Reihe **einfacher Datentypen**.



Benutzerdefinierte Datentypen ▾

Durch das **Ableiten** einfacher Datentypen können eigene benutzerdefinierte Datentypen definiert werden.

► Codebeispiel: Einfache Datentypen ▾

```

1  <!-- ----- -->
2  <!-- Element mit strukturiertem Inhalt -->
3  <!-- ----- -->
4  <?xml version="1.0" encoding="UTF-8"?>
5  <xs:schema xmlns:xs="http://... /XMLSchema">
6      <xs:complexType name="personType">
7          <xs:sequence>
8              <!-- Native Datentypen -->
9              <!-- xs:string, xs:int -->
10             <xs:element name="name"
11                 type="xs:string"/>
12             <xs:element name="age" type="xs:int"/>
13         </xs:sequence>
14     </xs:complexType>
15     <xs:simpleType name="nameType">
16         <xs:restriction base="xs:string">
17             <xs:length value="10"/>
18         </xs:restriction>
19     </xs:simpleType>
20 </xs:schema>
```

22.3.2 Benutzerdefinierte Datentypen

Die XML Spezifikation definiert 3 Möglichkeiten einfache Datentypen abzuleiten.

► Auflistung: Ableiten einfacher Datentypen ▾



Derivation by Restriction ▾

Neue Datentypen werden definiert, indem der Wertebereich eines gegebenen Datentyps **eingeschränkt** wird.



Derivation by Union ▾

Neue Datentypen werden definiert indem die Wertebereiche bestehender Datentypen **korreliert** werden.



Derivation by List ▾

Neben atomaren Datentypen können ebenfalls Listen von einfachen Typen, als neue Datentypen definiert werden.

22.3.3 Derivation by Restriction



Derivation by Restriction ▾

Neue Datentypen werden definiert, indem der Wertebereich eines gegebenen Datentyps **eingeschränkt** wird.

Zum Einschränken des Wertebereichs eines Datentyps definiert die XML Schema Spezifikation **Facets**.

► Erklärung: Facets ▾

- Facets definieren Regeln zum Einschränken der Wertebereiche einfacher Datentypen.
- Facets werden eingebettet in `<xs:restriction>` Elemente definiert. Dabei können mehrere Facets in einem restriction Element eingebettet werden.
- Facets beziehen sich dabei immer auf einen bestimmten Datentyp.
- Die XML Schema Spezifikation erlaubt die Definition eigener Facets.

► Codebeispiel: Derivation by Restriction ▾

```

1  <!-- ----- -->
2  <!--     Derivation by Restriction      -->
3  <!-- ----- -->
4  <xs:simpleType name="gFormat">
5      <xs:restriction base="xs:string">
6          <xs:enumeration value="jpeg"/>
7          <xs:enumeration value="svg"/>
8          <xs:enumeration value="png"/>
9          <xs:enumeration value="gif"/>
10     </xs:restriction>
11 </xs:simpleType>
12
13 <xs:element name="extension" type="gFormat"/>
14
15 <extension>gif</extension>
16
17
18 <xs:simpleType name="sCode">
19     <xs:restriction base="xs:string">
20         <xs:length value="8"/>
21     </xs:restriction>
22 </xs:simpleType>
23
24 <xs:element name="matrikelnr" type="sCode"/>
25
26 <matrikelnr>e9756234</matrikelnr>
27
28
29 <xs:simpleType name="userName">
30     <xs:restriction base="xs:string">
31         <xs:minLength value="3"/>
32         <xs:maxLength value="20"/>
33     </xs:restriction>
34 </xs:simpleType>
35
36 <xs:element name="user-name" type="userName"/>
37
38 <user-name>Wladimir</user-name>
39
40
41 <xs:simpleType name="httpURI">
42     <xs:restriction base="xs:anyURI">
43         <xs:pattern value="http://.*"/>
44     </xs:restriction>
45 </xs:simpleType>
46
47 <xs:element name="uri" type="httpURI"/>
48
49 <uri>http://htlkrems.ac.at</uri>

```

Facet	Beschreibung	Typ
<xs:enumeration>	Das Facet erlaubt die Definition einer Liste von möglichen Werten, die das String, Integer Element annehmen kann. Vergleichen Sie das Facet mit einem Enum.	
<xs:length>	Das Facet legt die Länge eines String fest	String
<xs:minLength>	Das Facet legt die minimale Länge eines Strings fest.	String
<xsmaxLength>	Das Facet legt die maximale Länge eines Strings fest.	String
<xs:pattern>	Das Facet definiert ein Muster die ein String haben muss.	String
<xs:minInclusive>	Das Facet definiert den minimalen Wert den ein Integer annehmen kann.	Integer
<xs:minExclusive>	Das Facet definiert den minimalen Wert den ein Integer annehmen kann.	Integer
<xs:maxInclusive>	Das Facet definiert den maximalen Wert den ein Integer annehmen kann.	Integer
<xs:maxExclusive>	Das Facet definiert den maximalen Wert den ein Integer annehmen kann.	Integer

Abbildung 89. Facets

22.3.4 Derivation by Union



Derivation by Union ▾

Neue Datentypen können definiert werden indem die Wertebereiche bestehender Datentypen **korreliert** werden.

```

1  <!-- ----- -->
2  <!-- Fallbeispiel: Enum -->
3  <!-- ----- -->
4  <xs:simpleType name="colorType">
5    <xs:union>
6      <xs:simpleType>
7        <xs:restriction base="xs:int">
8          <xs:enumeration value="0"/>
9          <xs:enumeration value="1"/>
10         </xs:restriction>
11      </xs:simpleType>
12      <xs:simpleType>
13        <xs:restriction base="xs:string">
14          <xs:enumeration value="white"/>
15          <xs:enumeration value="yellow"/>
16        </xs:restriction>
17      </xs:simpleType>
18    </xs:union>
19  </xs:simpleType>

```

22.4. Komplexe Datentypen

Komplexe Datentypen beschreiben Xml Elemente die andere Xml Elemente beinhalten können.

22.4.1 Elemente mit komplexen Datentypen

► Auflistung: Komplexe Datentypen ▾

- Komplexe Datentypen beschreiben Xml Elemente mit strukturiertem Inhalt.
- Komplexe Datentypen beschreiben Xml Elemente mit unstrukturiertem Inhalt mit Attributen.
- Komplexe Datentypen beschreiben Xml Elemente mit gemischem Inhalt.
- Komplexe Datentypen beschreiben Xml Elemente mit leerem Inhalt.

► Codebeispiel: <complexType> Element ▾

```

1  <!-- ----- -->
2  <!-- Komplexe Datentypen -->
3  <!-- ----- -->
4  <xs:complexType name="nameType">
5    ...
6    ...
7    ...
8    ...
9  </xs:complexType>

```

22.4.2 Strukturierung von XML Elementen

Die XML Schema Spezifikation definiert mehrere Möglichkeiten zur Beschreibung des Inhaltsmodells komplexer Datentypen.

Zur Beschreibung des Inhaltsmodells komplexer Datentypen stellt die XML Schema Spezifikation 3 Möglichkeiten zur Verfügung.

► Auflistung: Inhaltsmodelle ▾



Sequenz ▾

Wird das **Inhaltsmodell** eines XML Elements als Sequenz beschrieben, kann die Art und Reihenfolge der im Inhaltsmodell enthaltenen Elemente definiert werden.



Auswahl ▾

Wird das **Inhaltsmodell** eines XML Elements als Auswahl definiert, kann eine Auswahl von Elementen definiert werden, von denen 1 im Inhaltsmodell enthalten sein muß.



Aggregator ▾

Die Elemente eines Aggregators definieren einen **Pool von Elementen** der im Inhaltsmodell des Elements auftreten müssen.

► Codebeispiel: Auswahl ▾

```

1  <!-- ----- -->
2  <!--      Struktur: Auswahl      -->
3  <!-- ----- -->
4  <xs:complexType name="graphType">
5    <xs:sequence>
6      <xs:element name="node" type="nodeType"/>
7    </xs:sequence>
8  </xs:complexType>
9
10 <xs:complexType name="nodeType">
11   <xs:choice>
12     <xs:element name="label" type="xs:string"/>
13     <xs:sequence>
14       <xs:element name="node" type="nodeType"/>
15       <xs:element name="edge" type="xs:string"/>
16       <xs:element name="node" type="nodeType"/>
17     </xs:sequence>
18   </xs:choice>
19 </xs:complexType>
20
21 <xs:element name="graph" type="graphType"/>
22
23 <!-- ----- -->
24 <!--      graph.xml      -->
25 <!-- ----- -->
26 <?xml version="1.0" encoding="UTF-8"?>
27 <graph>
28   <node>
29     <node>
30       <node>
31         <label>A5</label>
32       </node>
33       <edge>a5</edge>
34     <node>
35       <label>A6</label>
36     </node>
37     <edge>a1</edge>
38   <node>
39     <node>
40       <label>A2</label>
41     </node>
42     <edge>a2</edge>
43   <node>
44     <label>A3</label>
45   </node>
46   <node>
47     <node>
48   </node>
49 </graph>
```

► Codebeispiel: Aggregator ▾

```

1  <!-- ----- -->
2  <!--      Struktur: Aggregator      -->
3  <!-- ----- -->
4  <x:complexType name="addressType">
5    <x:all>
6      <x:element tpe="x:string" name="code"/>
7      <x:element type="x:string" name="count"/>
8      <x:element type="x:string" name="street"/>
9      <x:element type="x:string" name="loc"/>
10   </x:all>
11 </x:complexType>

12 <x:element name="address" type="aType"/>
13
14 <!-- ----- -->
15 <!--      address.xml      -->
16 <!-- ----- -->
17 <!-- ----- -->
18 <?xml version="1.0" encoding="UTF-8"?>
19 <address>
20   <street>Alauntalstrasse 11</street>
21   <location>Krems a. Donau</location>
22   <code>3470</code>
23   <country>Austria</country>
24 </address>
25
26 <?xml version="1.0" encoding="UTF-8"?>
27 <address>
28   <country>Austria</country>
29   <code>3470</code>
30   <location>Krems a. Donau</location>
31   <street>Alauntalstrasse 11</street>
32 </address>
33
34 <?xml version="1.0" encoding="UTF-8"?>
35 <address>
36   <code>3470</code>
37   <location>Krems a. Donau</location>
38   <street>Alauntalstrasse 11</street>
39   <country>Austria</country>
40 </address>
41
42 <?xml version="1.0" encoding="UTF-8"?>
43 <address>
44   <code>3470</code>
45   <location>Krems a. Donau</location>
46   <street>Alauntalstrasse 11</street>
47   <country>Austria</country>
48 </address>
```

22.4.3 Häufigkeitsindikatoren

Häufigkeitsindikatoren legen fest wie oft eine Element im Inhaltsmodell eines Xml Elements auftreten darf.

► Erklärung: Häufigkeitindikatoren ▾

- **minOccurs:** Legt fest wie oft ein Element im Inhaltsmodell eines Xml Elements **zumindestens** vorkommen muß.
- **maxOccurs:** Bestimmt wie oft ein Element **maximal** vorkommen darf.

► Codebeispiel: Häufigkeitsindikatoren ▾

```

1  <!-- ----- -->
2  <!--      Struktur: Aggregator      -->
3  <!-- ----- -->
4  <x:complexType name="passengerType">
5    <x:sequence>
6      <x:element name="passenger"
7        tpe="x:string"
8        minOccurs="1" maxOccurs="5"/>
9    </x:sequence>
10   </x:complexType>
11
12 <x:element name="passengers"
13   type="passengerType"/>
14
15 <!-- ----- -->
16 <!--      passengers.xml      -->
17 <!-- ----- -->
18 <?xml version="1.0" encoding="UTF-8"?>
19 <passengers>
20   <passenger>Tsao Tse Long</passenger>
21 </passengers>
22
23 <?xml version="1.0" encoding="UTF-8"?>
24 <passengers>
25   <passenger>Guanyu</passenger>
26   <passenger>Konfuzius</passenger>
27 </passengers>
28
29 <?xml version="1.0" encoding="UTF-8"?>
30 <passengers>
31   <passenger>Guanyu</passenger>
32   <passenger>Konfuzius</passenger>
33   <passenger>Konfuzius</passenger>
34 </passengers>
```

22.4.4 gemischter Inhalt

Xml Elemente deren Inhaltsmodelle, sowohl andere Elemente als auch Text beinhalten können, haben einen gemischten Inhalt.

► Codebeispiel: gemischter Inhalt ▾

```

1  <!-- ----- -->
2  <!--      Struktur: mixed      -->
3  <!-- ----- -->
4  <x:complexType name="reportType"
     mixed="true">
5    <x:sequence>
6      <x:element name="index"
        type="x:string"
        maxOccurs="unbounded"/>
7    </x:sequence>
8  </x:complexType>
9
10 <x:element name="report" type="reportType"/>
11
12 <?xml version="1.0" encoding="UTF-8"?>
13 <text>In einer Hoehle in der Erde, da lebte
14 ein <index>Hobbit</index>. Nicht in einem
15 schmutzigen, nassen <index>Loch</index>,
16 in das die Enden von irgendwelchen Wuermern
17 herabbaumelten und das nach Schlamm und Moder
18 roch.
19 </text>
20

```

22.5. Attribute

Die Attribute eines Xml Elements werden im Inhaltsmodell getrennt von eingebetteten Elementen definiert.

Bevor Attribute für ein XML Element definiert werden können muß der Inhlatstyp des Elements bestimmt werden. Je nach Inhaltstyp werden Attribute unterschiedlich definiert.

22.5.1 Attributedefinitionen - Elemente mit einfacherem Inhalt

► Codebeispiel: Attributdefinition ▾

```

1  <!-- ----- -->
2  <!--      Fallbeispiel: Personendatensatz -->
3  <!-- ----- -->
4  <x:complexType name="bookType">
5    <x:sequence>
6      <x:element name="title" type="tType"/>
7    </x:sequence>
8  </x:complexType>
9
10 <x:complexType name="tType">
11   <x:simpleContent>
12     <x:extension base="x:string">
13       <x:attribute name="lang"
14         type="x:string"/>
15       <x:attribute name="version"
16         type="x:string"/>
17       <x:attribute name="dif"
18         type="x:string"/>
19     </x:extension>
20   </x:simpleType>
21 </x:complexType>
22
23 <?xml version="1.0" encoding="UTF-8"?>
24 <book>
25   <title lang="en" version="1.0">
26     Moby Dick
27   </title>
28   <description>
29     A book about a whale ...
30   </description>
31 </book>

```

22.5.2 Attributedefinitionen - Elemente mit komplexem Inhalt

Für Elemente mit **komplexem Inhalt** gestaltet sich die **Attributdefinition** wesentlich einfacher.

► Codebeispiel: Attributdefinition ▾

```

1  <!-- ----- -->
2  <!--      Attributdefinition      -->
3  <!-- ----- -->
4  <x:complexType name="lType">
5    <x:all>
6      <x:element name="address" type="aType"/>
7    </x:all>
8    <x:attribute name="id" type="xs:string"/>
9    <x:attribute name="code" type="xs:string"/>
10   </x:complexType>
11
12  <x:complexType name="aType">
13    <x:sequence>
14      <x:element name="country"
15        type="xs:string"/>
16      <x:element name="loc" type="xs:string"/>
17      <x:element name="street"
18        type="xs:string"/>
19    </x:sequence>
20  </x:complexType>
21
22  <x:complexType name="bType">
23    <x:sequence>
24      <x:element name="x" type="xs:string"/>
25      <x:element name="y" type="xs:string"/>
26      <x:element name="z" type="xs:string"/>
27      <x:element name="fig" type="xs:int"/>
28    </x:sequence>
29  </x:complexType>
30
31  <!-- ----- -->
32  <!--      Fallbeispiel: Attributdefinition      -->
33  <!-- ----- -->
34  <x:element name="library" type="lType"/>
35
36  <?xml version="1.0" encoding="UTF-8"?>
37  <library id="23" code="htl-323-2dfbfr">
38    <address>
39      <country>Austria</country>
40      <loc>3540 Krems</loc>
41      <street>Alaunthalstrasse 29</street>
42    </address>
43  </library>
```

22.6. Sichtbarkeitskonzepte ▾

Die Xml Schema Spezifikation unterscheidet **lokale** und **globale Datentypdeklarationen**.

22.6.1 globale Datentypdeklarationen

Eine **Datentypdeklaration** wird als **global** klassifiziert wenn die Deklaration als Kindelement des **schema Elements** definiert wird.

Globale Datentypen können von mehreren Elementen verwendet werden.

► Codebeispiel: Globale Datentypen ▾

```

1  <!-- ----- -->
2  <!--  Fallbeispiel: Globale Datentypen  -->
3  <!-- ----- -->
4  <x:complexType name="nameType">
5    <x:sequence>
6      <x:element name="surname"
7        type="xs:string"/>
8      <x:element name="givenname"
9        type="xs:string"/>
10     </x:sequence>
11   </x:complexType>
12
13  <x:element name="user-name" type="nameType"/>
14  <x:element name="pilot" type="nameType"/>
15  <x:element name="programmer"
16    type="nameType"/>
17  <x:element name="student" type="nameType"/>
18
19  <user-name>
20    <surname>Mustermann</surname>
21    <givenname>Max</givenname>
22  </user-name>
23
24  <pilot>
25    <surname>Armstrong</surname>
26    <givenname>Neil</givenname>
27  </pilot>
28
29  <programmer>
30    <surname>Akasha</surname>
31    <givenname>Bhairava</givenname>
32  </programmer>
```

23. Datenformat - JSON

01

JSON

01. JSON Grundlagen	240
02. JSON Datentypen	241

23.1. JSON Datenformat



json Datenformat ▾

JavaScript Object Notation, kurz JSON ist ein Datenformat für die Verarbeitung bzw. dem **Austausch** von **Daten** in Informationssystemen und Datenbankanwendungen.



23.1.1 JSON Grundlagen

Json wird zur **Übertragung** und zum **Speichern** von strukturierten Daten eingesetzt. Es dient dabei in erster Linie für die **Serialisierung** von Daten.

► Erklärung: JSON ▾

- **JSON** ist ein **textbasiertes**, von Menschen lesbares Datenformat, das für die Darstellung einfacher **Datenstrukturen** und **Objekte** verwendet wird.
- **JSON** kommt in Desktop- bzw. serverseitigen **Programmierumgebungen** zum Einsatz.
- **JSON** stammt ursprünglich aus dem **JavaScript** Umfeld. Jedes gültige **JSON Dokument** ist gültiger **JavaScript Code**. Ein JSON Dokument beschreibt ein gültiges⁵ JavaScrit Objekt.
- **JSON** kann von den meisten Programmiersprachen entweder direkt bzw. mit der Hilfe von Programmabibliotheken **verarbeitet** werden.
- **JSON** wurde in erster Linie als **Datenaustauschformat** konzipiert. Dabei zählt JSON zu den unstrukturierten Datenformaten.
- **JSON** ist neben XML das wichtigste Datenformat zur Übertragung von Daten. Wie XML ist es sprachunabhängig, gleichzeitig ist es in der Verwendung jedoch einfacher und effizienter als XML.



⁵ Obwohl JSON von JavaScript abstammt, sind beide nicht zu 100 kompatibel. Ein JSON Objekt kann Vereinfachungen beinhalten, die von einem JavaScript Interpreter nicht verarbeitet werden können.

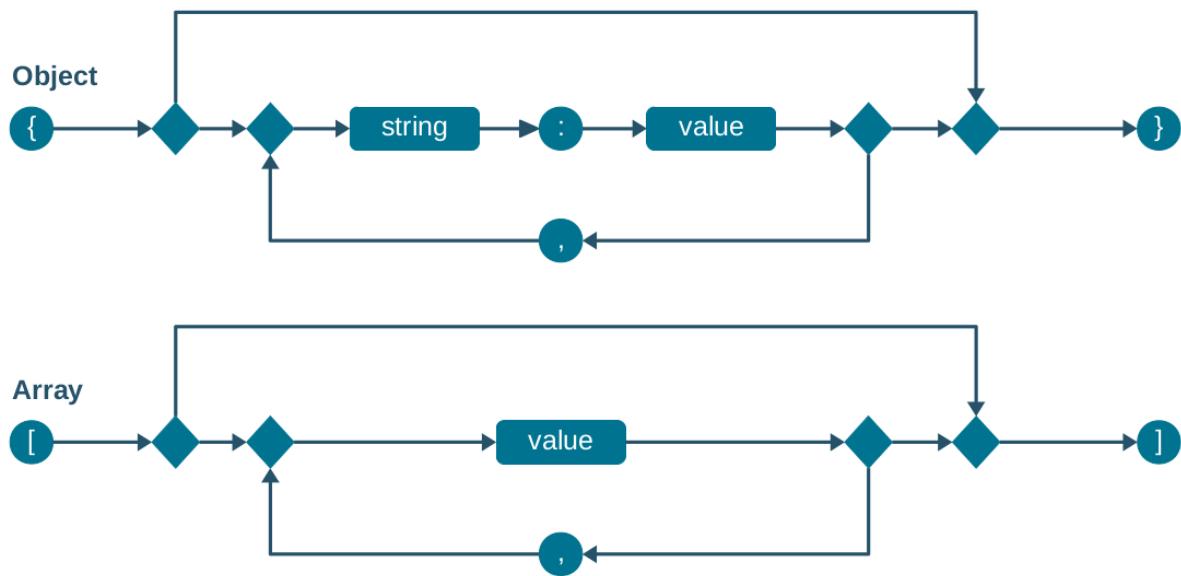


Abbildung 90. JSON: Grundstrukturen

23.2. JSON Datentypen

JSON unterscheidet eine Reihe von **Datentypen**.



23.2.1 JSON Datentypen

► Auflistung: JSON Datentypen ▾

- **Null:** Mit dem Schlüsselwort **null** kann einer Variable der **Nullwert** zugewiesen werden. Der Nullwert zeigt an dass sich kein Wert⁶ in der Variable befindet.
 - **Boolean:** Der Datentyp **Boolean** wird zur Verarbeitung von **Wahrheitswerten** verwendet. Mittels der Schlüsselwörter **true** und **false** kann der Wahrheitswert für eine Variable bestimmt werden.
 - **Zahl:** Für das Arbeiten mit **Zahlenwerten** stellt JSON den Datentyp **Number** zur Verfügung.

- **Zeichenkette:** Für das Arbeiten mit **Zeichenketten** stellt JSON den Datentyp **String** zur Verfügung. Die Zeichenkette wird dabei zwischen 2 Anführungszeichen notiert.

- **Array:** Für die Verarbeitung von **Listen** von **Werten** stellt JSON den Datentyp **array** zur Verfügung.

Um ein **Array** zu deklarieren werden beliebig viele Werte durch Kommas getrennt in eckigen Klammern `[]` notiert. Für jeden Eintrag im Array kann man einen beliebigen, für JSON zugelassenen Typ verwenden.

- **Objekt:** Für die Verarbeitung von komplexen, strukturierten Daten stellt JSON den Datentyp **Object** zur Verfügung

Objekte deklarieren eine beliebige Zahl von Variablen eingeschlossen in geschweiften Klammern. Dabei organisiert das Objekt die Variablen als **Schlüssel/Werte** Paare. Der Schlüssel fungiert als Name einer Variable, der Wert entspricht dem Wert der Variable.

⁶ Verwechseln Sie den Nullwert mit dem Zahlenwert 0 bzw. einem leeren String.

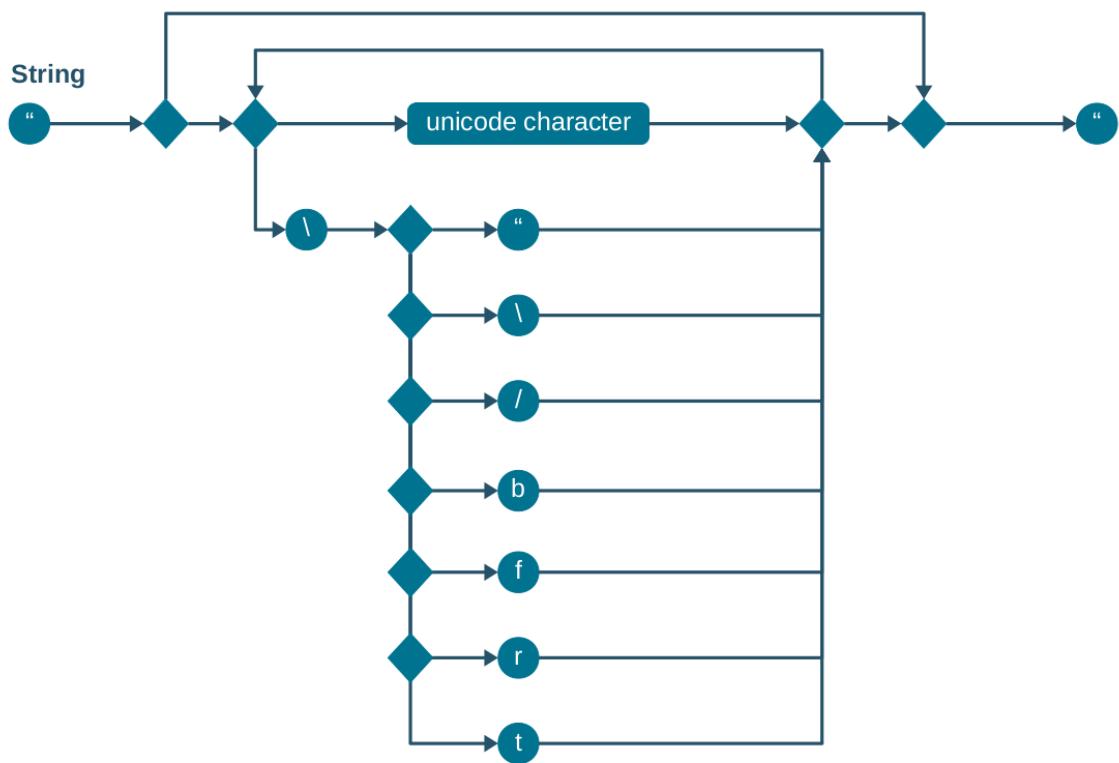


Abbildung 91. JSON: Zeichenketten

23.2.2 JSON Grundstrukturen

JSON unterscheidet 2 **Grundstrukturen** die es dem Datenformat erlauben die Strukturen und Variablen anderer Programmiersprachen abzubilden.

JSON dient in Informationssystemen als austauschbares Datenformat.

► Erklärung: JSON Object ▾

- Als Format zur Übertragung von Daten besteht für JSON die Notwendigkeit die **Datenstrukturen** anderer **Programmiersprachen** abbilden zu können.
- Der **Object** Datentyp wird verwendet um **Objekte**, **Records**, **Structs**, **Dictionaries** und **Hashtable** abzubilden.
- Hierbei handelt es sich um die universellen Datenstrukturen, die in Programmiersprachen unterstützt werden.

► Codebeispiel: JSON Objekt ▾

```

1 //-----
2 // JSON Objekt
3 //-----
4 var person = {
5     <string> : <value>,
6     <string> : <value>,
7     <string> : <value>,
8     <string> : <value>,
9     <string> : <value>,
10    ...
11    <string> : [
12        <string> : <value>,
13        <string> : <value>,
14        <string> : <value>
15        ...
16    ],
17    <string> : <value>
18    ...
19 }

```

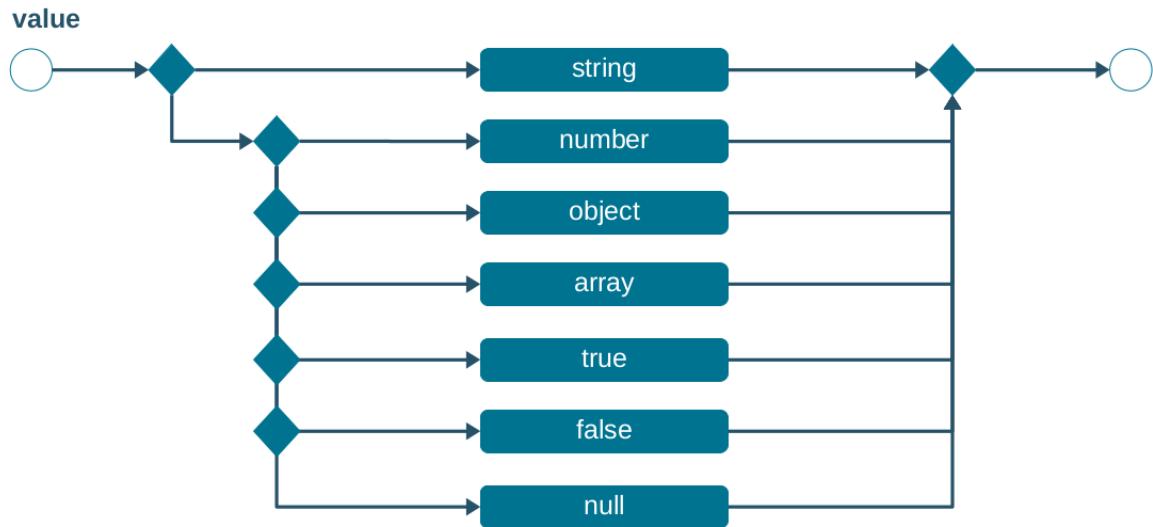


Abbildung 92. JSON: Datentypen

► Erklärung: JSON Array ▾

- Der **Array** Datentyp speichert eine **Liste** von geordneten Werten.
- Der **Array** Datentyp wird verwendet um **Arrays**, **Vektoren**, **Listen** bzw. **Sequenzen** abzubilden.

23.2.3 Einfache Datentypen

► Codebeispiel: String und Werte ▾

```

1 //-----
2 // JSON Array
3 //-----
4 var person = {
5   ...
6   ...
7   ...
8   <string> : [
9     <string> : <value>,
10    <string> : <value>,
11    <string> : <value>,
12    <string> : <value>,
13    <string> : <value>
14   ...
15 ],
16 ...
17 }

```

```

1 //-----
2 // Aufloesen von Strings und Werten
3 //-----
4 var person = {
5   "firstName": "John",
6   "lastName" : "Smith",
7   "isAlive"  : true,
8   "age"      : 45,
9   "height_cm": 1.78,
10  "address" : {
11    "street"     : "21 2nd Street",
12    "city"       : "New York",
13    "postalCode" : "10021-2100"
14  },
15  "children" : [
16    "firstName" : "John",
17    "lastName"  : "Wick",
18    "isAlive"   : true,
19    "age"       : 15
20  ],
21  "spouse"   : null
22 }

```


Glossar

A

Abfrage

Abfrageobjekt

Ein Abfrageobjekt wird zum Filtern von Daten einer Abfrage verwendet.

Abfragesprache

Aggregatfunktion

Funktionen, die eine Menge von Werten zu einem einzelnen Wert verdichten, werden als Aggregatfunktionen bezeichnet.

Alphanumerische Werte

Der Begriff alphanumerische Zeichen wird hauptsächlich im Bereich der Computertechnik bzw. Telekommunikation verwendet. Alphanumerische Werte sind Zeichenketten die aus Ziffern, Buchstaben bzw. Sonderzeichen bestehen können.

Alias

Ein Alias ist ein Pseudonym das anstelle eines bestimmten Bezeichners oder Wertes verwendet werden kann.

Anforderung

Anwendungslogik

Aspekt

Ein Aspekt beschreibt einen einzelnen Gesichtspunkt bei der Betrachtung, Auswertung oder Analyse komplexer Sachverhalte.

B

Baumstruktur

Baustein

Bedingung

Eine Bedingung ist eine Funktion, die immer den Wahrheitswert true oder falsch liefert.

Berichtstabelle

Bereichssuche

D

Datenaggregation

Datenaggregation beschreibt den Prozess des Zusammentragens von Daten aus unterschiedlichen Datenquellen.

Datenbankengine

Die Datenbankengine ist eine zugrundeliegende Softwarekomponente, die ein Datenbankverwaltungssystem verwendet, um Daten einer Datenbank zu erstellen, zu lesen bzw. zu löschen. Datenbankverwaltungssysteme besitzen eine eigene Programmierschnittstelle (API), die es dem Benutzer erlaubt die darunterliegende Engine zu verwenden.

Datenbankobjekt

Datenbankobjekte stellen das Gerüst zur Speicherung und Abfrage von Daten einer Datenbank dar. Beispiele von Datenbankobjekten sind: Tabelle, Datenbankindex, View, Alias, Stored Procedure, Trigger, Constraint usw.

Datenbanksprache

Datenbestand

Datendefinitionssprache

Datenmodell

Datenbankschema

Datenkontainer

Datenredundanz

Datenschema

Ein Datenbankschema beschreibt die physische Struktur der Daten einer Datenbank..

Datenstruktur

Datentyp

Deklarative Programmierung

Die deklarative Programmierung ist ein Programmierparadigma, bei dem die Beschreibung eines Problems im Vordergrund steht. Im Gegensatz zur imperativen Programmierung, bei der ein genauer Lösungsweg beschrieben werden muss, fragt man bei der deklarativen Programmierung was berechnet werden soll.

deterministisch

Differenzmenge

Durchschnittsmenge

E

Entitaet

Exception

F

Formatmaske

Formatmasken geben ein bestimmtes Muster vor, das zur Formatierung von Zeichenketten herangezogen wird. Die Formatmaske selbst ist dabei ebenfalls eine Zeichenkette.

Fremdschluessel

Funktion

Eine Funktion oder Projektion definiert eine Beziehung zweier Mengen, die jedem Element der einen Menge (Ausgangswerte), genau ein Element der anderen Menge (Zielwerte) zuordnet.

Funktionsparameter

Funktionsparameter sind Werte, die einer Funktion zur Verarbeitung mitgegeben werden..

G

Geschaeftsobjekt

Gegenoperation

H

Hierarchische Struktur

I

idempotent

Informationsstruktur

Inkonsistenzt

K

Komplexitaet

Komplementaeroperation

Konsistenz

Konstante

Konstruktionselement

Kontext

Als Kontext ist jede Art von Information gemeint, die für die Interaktion der Objekte eines Systems notwendig ist.

Kontextswitch

Konvertieren

Als Konvertierung wird die Umwandlung des Wertes eines Datentyps in den Wert eines anderen Datentyps bezeichnet.

L

Laufzeit

Laufzeitumgebung

Eine Laufzeitumgebung beschreibt die zur Laufzeit von Programmen verfügbaren Standardbibliotheken, Programmierschnittstellen, Laufzeitvariablen sowie die Hard- und Softwarekomponenten des Betriebssystems..

Literal

Als Literal bezeichnet man in der Programmierung eine Zeichenkette, die zur direkten Darstellung der Werte der Basistypen verwendet werden können.

logische Struktur

Die logische Struktur einer Datenbank beschreibt die Beziehungs- und Speicherstruktur der Datensätze..

M

Modul

N

Namensraum

Neustrukturierung

Normalisierung

P

Paginierung

Als Paginierung wird die Seitennummerierung eines Schriftstückes bezeichnet. In der Datenverarbeitung versteht man unter Paginierung, den seitenweisen Zugriff auf das Ergebnis einer Abfrage.

Patternmatching

Patternmatching beschreibt ein Verfahren zur musterbasierten Suche in einer Zeichenkette. Dazu wird anhand einer Suchmaske geprüft ob eine bestimmte Zeichenfolge in einer Zeichenkette vorhanden ist..

plattformübergreifend

Als Plattformunabhängigkeit wird die Eigenschaft eines Systems beschrieben, dass ein Programm auf unterschiedlichen Plattformen ausgeführt werden kann. Ein Programm wird dazu in einer sogenannten Laufzeitumgebung ausgeführt. Eine Laufzeitumgebung lädt dazu ein Programm und lässt diese auf der gewünschten Plattform ablaufen.

Primaerschlüssel

Programmablauf

Als Programmablauf wird in der IT der Ablauf eines Programms bezeichnet. Zur Steuerung des Programmablaufs werden in der Programmierung Kontroll- bzw. Schleifenkonstrukte verwendet.

Programmkontrolle

Projektion

Eine Projektion oder Funktion definiert eine Beziehung zweier Mengen, die jedem Element der einen Menge (Ausgangswerte), genau ein Element der anderen Menge (Zielwerte) zuordnet..

Q

Query by Example

R	Token
Referenz	Als Token wird eine kurze Zeichenkette beliebiger Zusammensetzung bezeichnet..
Relation	Transformation
Restriktion	
	Als Restriktion wird eine Bedingung bezeichnet, die die Datensätze eines Informationssystems erfüllen müssen..
S	Unterabfrage
Schluessel	
Spezifikation	V
Die Spezifikation eines Softwaresystems beschreibt alle funktionalen und nichtfunktionalen Anforderungen an die Implementierung der Anwendung.	Variable
Stagingtabelle	Variablendeclaration
Strukturierung	
T	W
Tabelle	Wertzuweisung
	Index
ACID Eigenschaften, 136	Atomarität, 136
ACID Transaktionsmodell, 136	
Aggregatfunktionen	Bigdata, 127
count, 53	Variety, 128
max, 53	Velocity, 128
min, 53	Volume, 128
sum, 53	Block, 92
Aggregate Phase, 57	BSON, 149
Aggregatfunktion	CA Systeme, 133
median, 54	CAP Theorem, 131
Aggregatfunktionen, 53	AP Systeme, 134
stdev, 54	Ausfallssicherheit, 131
Aggregation, 22, 53	CA Systeme, 133
Aggregationsframework, 177	CP Systeme, 134
AP Systeme, 134	Konsistenz, 131
asd, 31	Verfügbarkeit, 131

concat(), 228
 Consistent Hashing, 138
 contains(), 228
 count(), 227
 CP Systeme, 134
 Cursor, 113, 158

Daten, 3
 Datenbankartefakte, 69
 Datenformat, 5
 csv Format, 5
 json Format, 6
 xml Format, 5
 Datenmodellierung, 33
 Datensatz, 4
 Datenverarbeitung, 3
 Datumsfunktionen, 40
 add_month, 45
 extract, 45
 last_day, 45
 months_between, 46
 next_day, 46
 sysdate, 41
 systimestamp, 41
 to_date, 41
 Dauerhaftigkeit, 136
 DDL Befehl
 create view, 75
 drop view, 76
 deklarative Programmierung, 21
 desc, 31
 DML Befehl
 insert ... select, 81
 update, 82
 DML Befehlssatz, 81
 Dokumentschema, 151
 Embedded Documents, 150
 Externe Phase, 10

Fetch Klausel, 32
 From Klausel, 34
 Funktion, 117

Group by
 Aggregate Phase, 55
 Map Phase, 55
 Group By Klausel, 22

Having Klausel, 57

Index
 B* Index, 77
 Funktionsbasiert, 77

Volltextindex, 78
 Information, 3
 Informationsmanagement, 3, 21, 33, 131
 Inner Join, 35
 Isolation, 136

Join, 34
 JSON, 255

Konsistenz, 136
 Konsistenzmodell, 135
 eventual consistency, 135
 strict consistency, 135
 Konzeptionelle Phase, 10

last(), 227
 Limitierung, 23
 local-name(), 227
 Logische Phase, 10
 Lokalisierungspfad, 222
 Lösungsobjekt, 224

Map Phase, 140
 Map Reduce
 map, 140
 reduce, 140
 shuffle, 140
 Map Reduce Algorithmus, 139
 Median, 54
 MongoDB, 143
 \$addToSet, 171
 \$limit, 182
 \$match, 181
 \$mul, 170
 \$pop, 173
 \$project, 184
 \$pull, 171
 \$push, 172
 \$rename, 169
 \$reset, 168
 \$set, 168
 \$skip, 181
 \$sort, 181
 \$unwind, 182
 aggregate, 178
 Aggregationsframework, 177
 count, 176
 createCollection, 148, 155
 createView, 155
 Cursor, 158
 deleteMany, 174
 deleteOne, 174
 distinct, 176

drop, 156
getCollection, 156
getCollectionNames, 156
group, 176
insertMany, 166
limit, 159
Pipeline, 178
pretty, 158
sort, 159
updateMany, 168
updateOne, 166
MongoDB\$inc, 171
MonogDB
 \$addFields, 183
Multi Version Concurrency Control, 137
MVCC, 137

name(), 227
namespace-uri(), 227
NoSQL Datenbanken, 130
Numerische Funktionen, 50
 round, 50
 to_char, 51
 trunc, 50

Offset Klausel, 32
Optimistic Locking, 137
Order By Klausel, 23, 31

Physische Phase, 11
Pipeline, 178
PL/SQL
 Block, 92
 Datenbankfunktionalität, 90
 Datenkonsistenz, 90
 Datenmigration, 91
 Datensicherheit, 90
 Datenzugriff, 91
 Funktion, 117
 Prozedur, 115
 Qualifier, 96
 Variablen, 95
PL/SQL Befehl
 begin ... block, 93
 create ... function, 117
 create ... package body, 119
 declare ... block, 93
 end ... block, 93
 exception ... block, 93
PL/SQL Block
 Ausführungsteil, 93
 Deklarationsteil, 93
 Fehlerbehandlungsteil, 93

Funktion, 117
Prozedur, 115
Trigger, 120
PL/SQL Engine, 91
position(), 227
Projektion, 22
Prozedur, 115
Prädikat, 224

Qualifier, 96

Reduce Phase, 140
Right Join, 36

Schemafreiheit, 151
Schlüsselwort
 asd, 31
 between, 29
 create view, 75
 desc, 31
 distinct, 23
 drop view, 76
 fetch, 32
 from, 34
 group by, 55
 having, 57
 inner join, 35
 join, 34
 nulls first, 31
 nulls last, 31
 offset, 32
 order by, 31
 select, 23
 where, 28
 with ties, 32
Select Klausel, 22, 23
Selektion, 22
Sequenz, 79
Shuffle Phase, 140
Sortierung, 23
Sperre, 136
SQL, 21
 DDL, 69
 Kategorien, 21
 DCL, 22
 DDL, 22
 DML, 22
 DQL, 22
 Klausel, 22
SQL Klauseln
 fetch, 32
 from, 34
 group by, 55

having, 57
 offset, 32
 order by, 31
 select, 23
 where, 28
 Standardabweichung, 54
 starts-with(), 228
 string(), 228
 string-length(), 228
 Strukturierung, 22
 Stylesheet, 233
 substring(), 228
 substring-after(), 228
 substring-before(), 228

 Template, 233
 Textfunktionen, 47
 initcap, 49
 instr, 47
 length, 47
 lower, 49
 ltrim, 47
 replace, 49
 rtrim, 47
 soundex, 48
 substr, 49
 trim, 47
 upper, 49
 Transaktion, 136
 ACID Transaktionsmodell, 136
 Atomarität, 136
 Dauerhaftigkeit, 136
 Isolation, 136
 Konsistenz, 136
 Trigger, 120

 Varietät, 128
 Velocity, 128
 Verteilte Datenbank, 129
 Verteilte Datenbanken, 129
 View, 75
 create view, 75
 Virtuelle Tabelle, 34
 Virtuelle Tabelle, 55
 Aggregate Phase, 57
 Map Aggregate, 55
 Volume, 128

 Where Klausel, 22, 28
 Wissen, 3
 with ties, 32

 XML

Attribute, 215
 Element, 213
 Schema, 245
 XPath, 221

 XML Attribute, 215
 XML Element, 213
 XML Schema, 245
 Xml Schema, 245
 XPath, 221
 . Operator, 223
 .. Operator, 223
 // Operator, 223
 concat(), 228
 contains(), 228
 count(), 227
 last(), 227
 local-name(), 227
 Lokalisierungspfad, 222
 Lösungsobjekt, 224
 name(), 227
 namespace-uri(), 227
 position(), 227
 Prädikat, 224
 starts-with(), 228
 string(), 228
 string-length(), 228
 substring(), 228
 substring-after(), 228
 substring-before(), 228

 XSLT, 231
 Stylesheet, 233
 Template, 233
 xsl:apply-templates, 235
 xsl:choose, 238
 xsl:copy, 241
 xsl:element, 240
 xsl:for-each, 239
 xsl:if, 237
 xsl:otherwise, 238
 xsl:param, 236
 xsl:value-of, 237
 xsl:variable, 237
 xsl:when, 238
 xsl:with-param, 236

 Zeilenfunktionen
 add_month, 45
 extract, 45
 initcap, 49
 instr, 47
 last_day, 45
 length, 47
 lower, 49

ltrim, 47
months_between, 46
next_day, 46
replace, 49
round, 50
rtrim, 47
soundex, 48
substr, 49
sysdate, 41
systimestamp, 41
to_char, 51
to_date, 41
trim, 47
trunc, 50
upper, 49