ORACLE

ORACLE DOJO NR. 11

C. CZARSKI & B. HAMMERSCHMIDT

Arbeiten mit JSON und Oracle12c

Document Store und SQL – zwei Seiten einer Datenbank

Oracle Dojo ist eine Serie von Heften, die Oracle Deutschland B.V. zu unterschiedlichsten Themen aus der Oracle-Welt herausgibt.

Der Begriff Dojo [‘do:dʒo] kommt aus dem japanischen Kampfsport und bedeutet Übungshalle oder Trainingsraum. Als „Trainingseinheiten“, die unseren Anwendern helfen, ihre Arbeit mit Oracle zu perfektionieren, sollen auch die Oracle Dojos verstanden werden. Ziel ist es, Oracle-Anwendern mit jedem Heft einen schnellen und fundierten Überblick zu einem abgeschlossenen Themengebiet zu bieten.

Im *Oracle Dojo Nr. 11* beschäftigen sich Carsten Czarski, Senior Leitender Systemberater innerhalb der BU Datenbank und Beda Hammerschmidt, führender Entwickler der Oracle-DB JSON-Funktionalität mit den neuen Möglichkeiten in Oracle12c zum Umgang mit JSON-Dokumenten, flexiblen Daten und SQL-Funktionen.

ORACLE®

Inhalt

ORACLE

ORACLE DOJO NR. 11

CARSTEN CZARSKI & BEDA HAMMERSCHMIDT

Arbeiten mit JSON und Oracle12c

Document Store und SQL – zwei Seiten einer Datenbank

VORWORT DES HERAUSGEBERS

Datenbanken ohne Anwendungen sind sinnlos!

Nicht umsonst hat Oracle seit den ersten Code-Zeilen Ende der 1970er-Jahre enorm viele Funktionen und Technologien für Entwickler in die Datenbank gepackt. Angefangen mit SQL – das gerade eine Hyper-Renaissance erlebt – über PL/SQL und Java in der Datenbank, Text-Search-Technologien, Spatial, RDF für semantische Systeme, XML, die Sprache ‚R', unzählige analytische Funktionen, Schnittstellen zu Java, .NET, RESTful Service ... und natürlich JSON.

Entwickler stehen für uns im Fokus des Interesses und das nicht erst seit APEX die Entwicklergemeinde rockt!

Schön und gut mag der eine oder die andere sagen, auch wenn SQL wieder im Aufwind ist, und auch wenn immer mehr „noSQL"-Datenbanken versuchen, SQL zu sprechen, so haben sie Dinge im Portfolio, die gestandene Datenbanken nicht können, für die die Enterprise-DBs einfach zu alt und zu schwerfällig sind.

Meine Antwort: Alle Funktionen – alle guten Funktionen der Newcomer-DBs, um genauer zu sein – werden schneller als die meisten denken, in Oracle implementiert werden. Das war bei der objektorientierten-Phase, bei der XML-Phase

und bei der InMemory-Phase der Fall, und die noSQL- und JSON-Phase wird keine Ausnahme bilden.

Mehr noch: Bereits die erste Version einer solchen Technologieabsorption wird deutlich mehr können als die „One-Trick-Ponies". JSON innerhalb einer Oracle-DB ist keine isolierte, funktionsarme Insel, sondern ist integraler Bestandteil einer Oracle-Datenbank-Infrastruktur und mit allen Sicherheits-, Skalierungs- und Funktions-Genen einer Oracle-Datenbank ausgestattet.

Ich freue mich sehr, dass sich mit Carsten Czarski, Senior Leitender Systemberater innerhalb der BU Datenbank und Beda Hammerschmidt, dem führenden Entwickler der Oracle-DB JSON-Funktionalität, zwei hochkarätige Spezialisten bereit erklärt haben, in dieses extrem wichtige Thema einzuführen.

YeSQL mit Oracle und JSON!

Ich wünsche Ihnen viel Spaß beim Lesen und beim Testen.

Ihr Günther Stürner
Vice President Sales Consulting

PS: Wir sind an Ihrer Meinung interessiert. Anregungen, Lob oder Kritik gerne an barbara.frank@oracle.com. Vielen Dank!

1 Einleitung

Die Entwicklung von Software hat in den letzten Jahren vielfältige Änderungen erfahren: Früher ging jeder Anwendung stets die Phase der Datenmodellierung voraus – alle Entitäten und Attribute werden in einem Schema beschrieben und dieses wird komplett modelliert, bevor die Applikation implementiert wird. In einer relationalen Datenbank findet sich das Schema in Form von Tabellen, Spalten, Views und Geschäftslogik in Stored Procedures wieder. Das Reflektieren neuer Attribute oder veränderter Entitäten bedeutet eine Änderung am Schema, was in vielen Fällen zu einer Downtime der Anwendung führt.

In den letzten Jahren stehen Agilität und Flexibilität weit mehr im Vordergrund als früher – der Bedarf nach neuen Attributen oder neuen Strukturen entsteht schnell – Zeit für eine aufwendige Änderung der Datenstrukturen ist aber oft nicht vorhanden. Auch Schnittstellen ändern sich in der Regel schnell: Nutzt man eine öffentliche API aus dem Internet wie beispielsweise Google oder Twitter, so erfährt man von Änderungen oft erst im Nachhinein, wenn überhaupt. Ab einem Zeitpunkt X liefert die API bestimmte Attribute nicht mehr aus und neue kommen hinzu. Es ist nahezu unmöglich, zu Beginn ein Schema zu modellieren, welches alle Eventualitäten berücksichtigt.

Bei all diesen Überlegungen kommt der Datenbank eine Schlüsselrolle zu – denn während sich Middleware-Komponenten leicht austauschen lassen, sind gespeicherte Daten langlebiger – die Entscheidung für eine bestimmte Form der Datenspeicherung wirkt im Unternehmen meist über Jahre oder gar Jahrzehnte nach.

Das ist auch für die Überlegung wichtig, wie man Daten flexibel abspeichern möchte. Oft werden dafür Spezial- oder NoSQL-Datenbanken favorisiert – die Restriktionen und Besonderheiten, die eine solche Wahl mit sich bringt, wirken dann sehr lange nach und man „bezahlt“ die Flexibilität, indem man andere, wichtige Datenbank-Funktionen aufgibt: Transaktionen, referenzielle Integrität und Konsistenz sind Beispiele, die von NoSQL-Datenbanken nicht oder nur teilweise unterstützt werden. Abfragen sind entweder nur mit einem eindeutigen Schlüssel oder mit stark limitierten Abfragesprachen möglich, die nicht von der physikalischen Repräsentation der Daten abstrahieren können. Joins und Views gibt es dann nicht, sodass Abfragen, die mit SQL sehr einfach und in wenigen Zeilen formulierbar sind, aufwendig programmiert oder die Daten sogar bewegt werden müssen, was langwierig und kostenintensiv ist.

Gefragt sind Technologien, die einerseits mit der erwähnten Flexibilität und Geschwindigkeit Schritt halten können, andererseits aber in der Lage sind, den logischen Zugriff auf die

Daten von der physikalischen Speicherung zu abstrahieren. Das relationale Modell ist nach wie vor eine hervorragende Grundlage, die sich um Möglichkeiten zur flexiblen Datenhaltung sehr gut ergänzen lässt.

Mit der im Release 12.1.0.2 eingeführten JSON-Unterstützung bietet die Oracle-Datenbank dem Entwickler einen solchen Ansatz an: Nach wie vor verfolgt Oracle das relationale Modell – Daten werden in Tabellen und Spalten abgelegt. Zusätzlich gibt es nun aber die Möglichkeit, flexible Daten im JSON-Format abzulegen. Für diese Daten ist dann kein Schema mehr erforderlich – der Entwickler lässt JSON-Dokumente einfach „in die Datenbank fallen". Erweiterungen der Abfragesprache SQL erlauben es dann, komplexe Abfragen auf die so gespeicherten JSON-Daten zu stellen, diese mit bestehenden relationalen Daten zu kombinieren und den gesamten Funktionsumfang von SQL zu nutzen.

Auf der anderen Seite steht mithilfe des REST-Standards eine einfache Schnittstelle für Entwickler bereit, die wiederum vollkommen von SQL abstrahiert. Entwickler erzeugen Collections, speichern JSON-Dokumente darin und rufen sie wieder ab – tatsächlich sind die JSON-Dokumente in einer Tabellenspalte abgelegt, auf die man mit SQL-Funktionen zugreifen kann. Diese SODA (*Simple Oracle Document Access*) Schnittstelle wird auch für den Java-Programmierer

angeboten. So bietet die Oracle-Datenbank beiden Seiten die jeweils bevorzugte Schnittstelle an. Die Daten bleiben in einem offenen Format gespeichert, können bei Bedarf in klassische relationale Tabellen konvertiert werden und auch neue, zu Beginn des Projekts noch unbekannte, Anforderungen lassen sich effizient und schnell umsetzen.

Dieses Dojo stellt die neuen Möglichkeiten in Oracle12c zum Umgang mit JSON-Dokumenten, flexiblen Daten und SQL-Funktionen vor. Zunächst wird JSON aus der Sicht des relationalen Datenbanksystems betrachtet – wie es in Tabellen gespeichert und mit SQL-Funktionen abgefragt werden kann. Danach folgt die Sicht des (NoSQL)-Entwicklers: Vorgestellt wird, wie sich Collections und Dokumente in der Oracle-Datenbank mit einer REST-Schnittstelle verwalten lassen, ohne dass der Entwickler mit Tabellen, View oder SQL in Berührung kommt.

2 JSON in der Oracle-Datenbank: SQL/JSON

2.1 JSON IN TABELLEN SPEICHERN

Natürlich kann JSON prinzipiell in allen Oracle-Datenbankversionen in Tabellen gespeichert werden, denn JSON sind zunächst einfache Textdaten, die sich auch als solche ablegen lassen. In diesem Dojo ist aber mehr gemeint: Es geht darum, dass die Datenbank-Funktionen anbietet, die es erlauben, mit JSON zu arbeiten.

- Validierung: Liegt syntaktisch korrektes JSON vor oder nicht?
- Abfragen: Extraktion einzelner Attribute aus dem JSON
- Abfragen: Projektion von JSON-Attributen als SELECT-Ergebnismenge
- Indizieren von JSON-Daten

In der Oracle-Datenbank werden solche Funktionen ab der Version 12.1.0.2 angeboten.

2.1.1 DATENTYPEN

Für JSON wurde, im Gegensatz zur XML-Unterstützung vor einiger Zeit, kein eigener Datentyp eingeführt – JSON wird in Spalten vom Typ VARCHAR, CLOB oder BLOB abgelegt. Das hat den Vorteil, dass externe Programme und Werkzeuge die JSON-Funktionen unmittelbar nutzen können, ohne eine spezielle Datentyp-Unterstützung mitbringen zu müssen. Tabellen mit JSON-Spalten werden also recht einfach angelegt und mit Daten befüllt, wie das folgende Beispiel zeigt.

```
create table dojo_json (
    id        number(10)   generated always as identity,
    created   date         default sysdate not null,
    json      clob
);

insert into dojo_json (json) values (
'{"typ": "dojo", "thema": "JSON"}');
```

Wenn schon im Vorfeld sicher ist, dass die JSON-Dokumente nicht größer als 32 Kilobyte werden, kann man auch mit dem VARCHAR2-Datentypen arbeiten; beim Speichern und Abrufen von Daten ist der typischerweise performanter als ein Large Object. Darüber hinaus kann JSON auch als BLOB (Binary Large Object) abgelegt werden.

```
create table dojo_json (
    id        number(10)   generated always as identity,
    created   date         default sysdate not null,
    json      blob
);

insert into dojo_json (json) values (
utl_raw.cast_to_raw('{"typ": "dojo", "thema": "JSON"}')
);
```

JSON-Dokumente sind immer im Unicode-Zeichensatz kodiert. Im Idealfall benutzt die Oracle-Datenbank daher ebenfalls diesen (AL32UTF8), damit keine Umwandlung der JSON-Dokumente zum Datenbank-Zeichensatz erfolgen muss. An dieser Stelle sei noch eine Besonderheit des CLOB-Datentypen erwähnt: Auch in einer AL32UTF8-Datenbank sind diese im UCS-2 Format kodiert – für CLOBs findet also (auch in einer AL32UTF8-Datenbank) stets eine Zeichensatz-Konvertierung statt.

Wird das JSON als BLOB abgelegt, trifft das natürlich nicht zu – denn dieser speichert die Bytes so, wie sie sind – der Datenbank-Zeichensatz spielt keine Rolle. BLOB-Daten verbrauchen aus diesem Grund in der Regel weniger Speicherplatz und sind etwas performanter als CLOBs, da keine internen Zeichensatz-Konversionen (UCS-2 zu UTF-8) stattfinden müssen. Für kleinere JSON-Dokumente bleibt VARCHAR2 die beste Wahl.

2.1.2 SQL-FUNKTION IS JSON

Da für JSON kein besonderer Datentyp verwendet wird, lassen sich JSON-Daten in einer Tabellenspalte beliebig mit Nicht-JSON-Daten mischen. Dann stellt sich natürlich die Frage, wie man die Nicht-JSON-Daten erkennen kann – hierzu dienen die für JSON eingeführte SQL-Operatoren IS JSON und IS NOT JSON, die in SQL-Abfragen wie folgt eingesetzt werden können.

```
insert into dojo_json (json) values ('{"typ": "dojo", "thema":
"JSON"}');
insert into dojo_json (json) values ('Das ist kein JSON');
select id, json from dojo_json where json IS JSON;

        ID  JSON
---------- ----------------------------------
         1  {"typ": "dojo", "thema": "JSON"}

select id, json from dojo_json where json IS NOT JSON;

        ID  JSON
---------- ----------------------------------
         2  Das ist kein JSON
```

Neben diesen eindeutigen Fällen gibt es im JSON-Umfeld allerdings auch „Grauzonen“, in denen man nicht so leicht

sagen kann, ob das Dokument korrektes JSON ist oder nicht. Die folgenden Dokumente sind Beispiele dafür: Legt man die Syntax streng aus, so liegt kein korrektes JSON vor; in der Praxis wird solches JSON allerdings oft verwendet und auch erfolgreich verarbeitet.

```
insert into dojo_json (json) values ('{typ: "dojo", thema: "JSON"}');
insert into dojo_json (json) values ('{"typ": "dojo", "typ":"buch"}');
insert into dojo_json (json) values ('{typ: "dojo", typ: "buch"}');
```

Im ersten JSON-Dokument sind die Attributnamen nicht von Anführungszeichen eingeschlossen, was zwar ein Syntaxverstoß ist, in der Praxis aber recht häufig vorkommt. Das zweite JSON-Dokument enthält ein Attribut doppelt – auf gleicher Hierarchieebene. Das dritte schließlich kombiniert beide Varianten. Der IS JSON Operator wird alle drei Dokumente als korrektes JSON klassifizieren, da die SQL/JSON-Funktionen mit allen drei Fällen umgehen können. Allerdings kann man den Operator mit zusätzlichen Schlüsselworten zu strengeren Prüfungen zwingen, wie der folgende Code zeigt.

```
select id, json from dojo_json where json is json;

        ID  JSON
---------- ----------------------------------
         1  {typ: "dojo", thema: "JSON"}
```

```
         2  {typ: "dojo", typ: "buch"}
         3  {"typ": "dojo", "typ": "buch"}

3 Zeilen ausgewählt.

select id, json from dojo_json where json is json strict;

        ID  JSON
---------- ----------------------------------
         3  {"typ": "dojo", "typ": "buch"}

1 Zeile wurde ausgewählt.

select id, json from dojo_json where json is json strict with unique keys;

Es wurden keine Zeilen ausgewählt
```

IS JSON STRICT erzwingt die strikte Syntaxprüfung für JSON-Dokumente; alle Attributnamen müssen also in Anführungszeichen eingeschlossen sein. Darüber hinaus müssen die Schlüsselworte **true**, **false** und **null** in der strikten Syntax kleingeschrieben sein und das Format für numerische Werte ist strikter definiert. IS JSON LAX erlaubt Abweichungen von den strikten Syntaxregeln, die in der Praxis gelegentlich auftauchen und welche die Oracle-Datenbank verarbeiten kann. Verwendet man weder STRICT noch LAX, so arbeitet die Datenbank standardmäßig mit LAX.

Die Klausel WITH UNIQUE KEYS ist etwas anders gelagert. Das JSON-Dokument mit der ID 3 in obigem Beispiel ist syntaktisch korrekt – hat aber ein Problem: Das Attribut **typ** kommt zweimal vor.

```
insert into dojo_json (json) values ('{typ: "dojo", typ: "buch"}');
```

Fragt man nun den Wert von **typ** ab, so kann man ohne Weiteres nicht sagen, welcher der beiden Werte zurückgeliefert wird. Der Umgang mit nicht-eindeutigen Attributen ist nicht definiert und unterschiedliche Programme gehen damit unterschiedlich um; in der Regel wird eines der Attribute ignoriert. Möchte man diese Fälle ausschließen, so setzt man die Klausel WITH UNIQUE KEYS ein. Allerdings führt diese Klausel zu erhöhtem CPU-Bedarf bei Ausführung des Operators – man sollte sie also nur einsetzen, wenn man sie wirklich braucht. In der Praxis tauchen JSON-Daten mit doppelten Attributnamen nur selten auf.

2.1.3 JSON SYNTAX MIT CHECK-CONSTRAINT ERZWINGEN

Setzt man den IS JSON Operator als Check-Constraint ein, so lässt sich sicherstellen, dass eine Tabelle nur syntaktisch korrekte JSON-Dokumente enthält. Auch bei der Nutzung als Check-Constraint können die Klauseln STRICT und WITH UNIQUE KEYS verwendet werden. Das folgende Syntaxbeispiel zeigt, wie man nicht nur syntaktisch korrektes

JSON erzwingen kann, sondern darüber hinaus auch noch die strikte Syntax und eindeutige Attributnamen durchsetzt.

```
alter table dojo_json
add constraint check_json CHECK (json IS JSON STRICT WITH UNIQUE
KEYS);
```

Wie bei allen Check-Constraints wird das Kommando wird nur dann erfolgreich ausgeführt, wenn alle bereits befindlichen JSON-Dokumente dem Kriterium entsprechen. Falls nicht, wird eine Fehlermeldung ausgelöst.

2.1.4 DATA DICTIONARY VIEWS FÜR JSON

Ein weiterer Vorteil des Check-Constraints ist, dass die Tabellenspalten dann in die Data Dictionary Views USER_JSON_COLUMNS, ALL_JSON_COLUMNS und DBA_JSON_COLUMNS aufgenommen werden. Es wird so wesentlich leichter, sich über das Vorhandensein von JSON-Daten in der Oracle-Datenbank zu informieren.

```
SQL> desc user_json_columns
 Name                                      Null?    Typ
 ----------------------------------------- -------- ------------
 TABLE_NAME                                NOT NULL VARCHAR2(128)
 COLUMN_NAME                               NOT NULL VARCHAR2(128)
 FORMAT                                             VARCHAR2(9)
 DATA_TYPE                                          VARCHAR2(13)
```

2.1.5 EXTERNE TABELLEN: BEISPIELDATEN „PURCHASEORDERS“

JSON-Daten können auch als externe Tabelle in die Datenbank eingebunden werden. Für diesen Fall enthält die Installation der Datenbank sogar eine Beispieldatei. Im Verzeichnis **${ORACLE_HOME}/demo/schema** */order_entry* auf dem Datenbankserver befindet sich die Datei **PurchaseOrders.dmp**. Für diese Datei lässt sich das mit den folgenden Kommandos erzeugen. Zum Ausführen des CREATE DIRECTORY Befehls braucht man aber das Systemprivileg CREATE ANY DIRECTORY.

Externe Tabelle

```
create directory DIR_POJSON as '/opt/oracle/product/12.1.0/
dbhome_1/demo/schema/order_entry';

create table exttab_pojson (json clob)
ORGANIZATION EXTERNAL (
 type oracle_loader
 default directory DIR_POJSON
 access parameters (
  records delimited by newline
  nologfile nobadfile
  fields missing field values are NULL(JSON char(10000))
 )
 location ('PurchaseOrders.dmp')
)
reject limit unlimited;
```

Anschließend kann die Tabelle wie jede andere selektiert werden. Mit dem IS JSON-Operator kann geprüft werden, ob sich in allen Zeilen der Tabelle korrektes JSON befindet.

```
select count(*) from exttab_po_json where json IS NOT JSON;
 COUNT(*)
----------
     0

select count(*) from exttab_po_json where json IS JSON;

 COUNT(*)
----------
   10000
```

2.1.6 BEISPIELDATEN: EARTHQUAKES

Ein anderer Weg, an Beispieldaten zu kommen, ist das Internet. Das USGS (United States Geological Survey) hält JSON-Feeds mit Erdbebendaten bereit. So liefert der URL **http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_week.geojson** ein JSON-Dokument mit allen registrierten Erdbeben der letzten sieben Tage zurück. Dieses kann mit dem folgenden Code leicht in die Datenbank geladen werden.

```
create table dojo_usgs_earthquake_stage (
```

```
id      number(10)     generated always as identity primary key,
        created        date   default sysdate not null,
        json           clob
);

create or replace procedure dojo_grab_earthquakejson is
begin
 -- Falls Datenbank hinter einer Firewall:
 -- Proxyserver hier an eigene Umgebung anpassen.
 utl_http.set_proxy('{proxyserver.meinefirma.de}:{proxyport}');

 insert into dojo_usgs_earthquake_stage (json) values (
  httpuritype('http://earthquake.usgs.gov/earthquakes/'||
        'feed/v1.0/summary/all_month.geojson').getclob()
 );
end dojo_grab_earthquakejson;
/

begin
 dojo_grab_earthquakejson;
end;
/
```

2.2 JSON-DOKUMENTE ABFRAGEN: SQL/JSON-FUNKTIONEN

Sobald JSON-Dokumente in einer „normalen“ oder externen Tabelle vorliegen, können diese mit den SQL/JSON-

Funktionen abgefragt werden. In der Oracle-Datenbank gibt es vier SQL/JSON-Funktionen: JSON_VALUE, JSON_QUERY, JSON_EXISTS und JSON_TABLE.

Was alle vier Funktionen gegenüber anderen SQL-Funktionen auszeichnet, ist die Tatsache, dass man einen *Error-Handler* festlegen kann. Im Fehlerfall, also wenn ein JSON-Dokument in der Tabelle ungültig ist, liefern die SQL/JSON-Funktionen standardmäßig SQL NULL zurück. Alternativ kann man mit der ON ERROR-Klausel festlegen, dass tatsächlich ein Fehler ausgelöst oder dass ein Standardwert (Default) zurückgegeben werden soll.

2.2.1 JSON-PFADAUSDRÜCKE

Um Werte innerhalb eines Objekts zu selektieren, werden sogenannte Pfadausdrücke verwendet. Ähnlich den aus XML bekannten XPath-Ausdrücken, definieren diese den Weg zum selektierten Attribut. Pfadausdrücke für JSON in der Oracle-Datenbank sind der JavaScript-Syntax entlehnt und beginnen mit dem $-Symbol. Mittels eines Punktes wird innerhalb eines Objektes navigiert: **$.typ** selektiert also den Wert des Attributes **typ**. Innerhalb von JSON-Arrays navigiert man mit eckigen Klammern: **$[0]** selektiert den ersten Wert eines Arrays. Zur Navigation in geschachtelten JSON-Daten können Pfadausdrücke aus mehreren Schritten bestehen: **$.Addressen[0].Stadt**

selektiert zuerst das Attribut Adressen, welches ein Array ist. Danach wird dessen erster Wert mit **[0]** angesprochen und von diesem wiederum das Attribut **Stadt** selektiert.

2.2.2 JSON_VALUE

Mit der SQL/JSON-Funktion JSON_VALUE kann man einzelne Attribute aus JSON-Dokumenten extrahieren. Das extrahierte Attribut muss aber ein skalarer Wert sein, eine JSON-Unterstruktur (Objekt) kann von JSON_VALUE nicht gelesen werden.

Das folgende Codebeispiel zeigt, anhand der Beispieltabelle EXTTAB_POJSON (Abschnitt 2.1.4), wie das JSON-Attribut Reference für jedes JSON-Dokument ausgelesen wird.

```
select json_value(json, '$.Reference')
from exttab_pojson;

JSON_VALUE(JSON,'$.REFERENCE')
--------------------------------------------------------------
MSULLIVA-20141102
MSULLIVA(-20141113
TRAJS-20140518
:

10000 Zeilen ausgewählt.
```

Mit der RETURNING-Klausel kann der zurückgegebene Datentyp näher bestimmt werden.

```
select json_value(json, '$.Reference' RETURNING VARCHAR2(20))
from exttab_pojson;
```

Zielt man dagegen auf ein JSON-Objekt, so wird SQL NULL zurückgeliefert.

```
select json_value(json, '$.ShippingInstructions')
from exttab_pojson;

JSON_VALUE(JSON,'$. SHIPPINGINSTRUCTIONS)
------------------------------------------------------------
- NULL -
- NULL -
- NULL -
:

10000 Zeilen ausgewählt.
```

Der Grund dafür wird nochmals deutlich, wenn man einen Error-Handler festlegt.

```
select json_value(json, '$.ShippingInstructions' ERROR ON ERROR)
from exttab_pojson;

*
```

```
FEHLER in Zeile 1:
ORA-29913: Fehler bei der Ausführung von Aufruf ODCIEXTTABLEFETCH
ORA-40456: JSON_VALUE wurde als nicht-skalarer Wert ausgewertet
```

Gerade wenn man SQL-Abfragen mit JSON_VALUE oder anderen SQL/JSON-Funktionen entwickelt, können Error-Handler sehr hilfreich sein: Anstelle eines einfachen SQL NULL im Ergebnis bekommt man eine etwas aussagekräftigere Fehlermeldung. Wie schon beschrieben, kann für den Fehlerfall auch ein Default-Wert festgelegt werden.

```
select json_value(json, '$.ShippingInstructions' DEFAULT 'ERROR'
ON ERROR)
from exttab_pojson;

JSON_VALUE(JSON,'$.SHIPPINGINSTRUCTIONS)
------------------------------------------------------------
ERROR
ERROR
ERROR
:

10000 Zeilen ausgewählt.
```

Die Möglichkeiten, die sich mit Error-Handlern ergeben, sind sehr nützlich, wenn eine Tabelle

- durchgängig korrektes JSON enthält,
- ein bestimmtes Attribut in manchen Dokumenten skalare Werte, in anderen Dokumenten aber Objekte enthält.

JSON-Attribute können Boolean-Werte (**true** oder **false**) enthalten. Da Oracle SQL keinen Boolean-Datentypen kennt, müssen diese Werte auf einen anderen Datentype abgebildet werden. Der Default ist hierbei VARCHAR2, das heißt der Boolesche JSON-Wert wird auf die Strings „**true**“ und „**false**“ abgebildet. Alternativ kann die Abbildung auf NUMBER gewählt werden; dann steht die **1** für **true** und die **0** für **false**.

```
select json_value(
  '{"bool": true}',
  '$.bool' returning number(1)
) json_boolean from dual;

JSON_BOOLEAN
------------
           1
```

2.2.3 JSON_EXISTS

JSON_EXISTS wird nur in der WHERE-Klausel einer SQL-Abfrage verwendet und liefert einen Treffer zurück, wenn das betreffende JSON-Attribut im Dokument existiert.

Die folgende Abfrage liefert alle Dokumente zurück, die das Attribut **Reference** enthalten.

```
select count(*) from exttab_pojson
where json_exists(json, '$.Reference');
```

Im Datenbankrelease 12.1.0.2 kann mit JSON_EXISTS nur die JSON-Struktur, also das reine Vorhandensein eines Attributs, berücksichtigt werden. Hat man dagegen die aktuellsten JSON-Patches von My Oracle Support heruntergeladen und eingespielt, so kann JSON_EXISTS auch testen, ob das Attribut einen bestimmten Wert hat. Möchte man feststellen, in welchen JSON-Dokumenten das Attribut **Reference** den Wert **MSULLIVA-20141102** hat, geht man wie folgt vor.

```
select count(*) from exttab_pojson
where json_exists(json, '$?($.Reference=="MSULLIVA-20141102")');
```

Alternativ kann man diese Abfrage auch „relational“ mit JSON_VALUE durchführen.

```
select count(*) from exttab_pojson
where json_value(json, '$.Reference') = 'MSULLIVA-20141102';
```

2.2.4 JSON_QUERY

JSON_QUERY ist gewissermaßen das Gegenstück zu JSON_VALUE. Mit dieser Funktion kann man Objekte aus einem JSON-Dokument extrahieren. Das folgende Beispiel extrahiert das Objekt **ShippingInstructions**, welches mit JSON_VALUE nicht angesprochen werden konnte.

```
select json_query(json, '$.ShippingInstructions')
from exttab_pojson;

JSON_QUERY(JSON,'$.SHIPPINGINSTRUCTIONS')
------------------------------------------------------------
{"name":"Martha Sullivan","Address":{"street":"200 Sporting Green"
,"city":"South San Francisco","state":"CA","zipCode":99236,"coun
try": "United States of America"},"Phone":[{"type":"Office","numb
er":"979-555-6598"}]}
{"name":"Trenna Rajs","Address":{"street":"200 Sporting
Green","city":"South San Francisco","state":"CA","zipCode":99236,"c
ountry":"United States of America"},"Phone":[{"type":"Office","numb
er":"905-555-5489"}]}
:
10000 Zeilen ausgewählt.
```

Das JSON-Fragment wird stets als VARCHAR2 zurückgegeben, kann also nicht größer als 32KB sein. Mit dem Schlüsselwort **PRETTY** sorgt man für eine leichter lesbare Ausgabe.

```
select json_query(json, '$.ShippingInstructions' PRETTY)
from exttab_pojson;

JSON_QUERY(JSON,'$.SHIPPINGINSTRUCTIONS')
------------------------------------------------------------
 {
  "name" : "Martha Sullivan",
  "Address" :
  {
   "street" : "200 Sporting Green",
   "city" : "South San Francisco",
   "state" : "CA",
   "zipCode" : 99236,
   "country" : "United States of America"
  },
  "Phone" :
  [
   {
    "type" : "Office",
    "number" : "979-555-6598"
   }
  ]
 }
```

Im Fehlerfall reagiert JSON_QUERY analog zu JSON_VALUE. Der Versuch, mit JSON_QUERY ein Attribut mit skalarem Wert auszulesen, führt also zu SQL NULL als Ergebnis.

```
select json_query(json, '$.REFERENCE')
from exttab_pojson;

JSON_QUERY(JSON,'$.REFERENCE')
--------------------------------------------------------------------
- NULL -
- NULL -
- NULL -
:

10000 Zeilen ausgewählt.
```

JSON_QUERY akzeptiert, analog zu JSON_VALUE, einen Error-Handler.

```
select json_query(json, '$.Reference' ERROR ON ERROR)
from exttab_pojson;

*
FEHLER in Zeile 1:
ORA-29913: Fehler bei der Ausführung von Aufruf ODCIEXTTABLEFETCH
ORA-40480: Ergebnis kann nicht ohne Array-Wrapper zurückgegeben
werden
```

Allerdings ist die Fehlermeldung eine andere – es wird von einem *Array-Wrapper* gesprochen. Die Idee ist, dass es recht einfach ist, aus einem skalaren Wert eine JSON-Struktur zu machen – rahmt man es in eckige Klammern ein, so entsteht ein JSON-Array mit genau einem Element. Und genau das ist der Array-Wrapper, von dem in der Fehlermeldung die Rede ist.

```
select json_query(json, '$.Reference' WITH WRAPPER)
from exttab_pojson;

JSON_QUERY(JSON,'$.REFERENCE'WITHWRAPPER)
--------------------------------------------------
["MSULLIVA-20141102"]
["MSULLIVA(-20141113"]
["TRAJS-20140518"]
:
```

Wird ein CONDITIONAL WRAPPER angefordert, so macht die Datenbank nur dann ein Array, wenn der Wert des Attributs auch wirklich skalar ist – für JSON-Dokumente, in denen das Attribut ein Objekt oder Array ist, wird kein Wrapper angewendet.

2.2.5 JSON_TABLE

JSON_TABLE ist die mächtigste SQL/JSON-Funktion. Sie erlaubt es, mehrere JSON-Attribute auf einmal, als relationales Abfrageergebnis, zu projizieren. Zusätzlich zum JSON-Pfadausdruck enthält JSON_TABLE die COLUMNS-Klausel, welche JSON-Attribute auf die Spalte der Ergebnismenge abbildet. Das folgende Beispiel bildet drei Attribute aus jedem JSON-Dokument und projiziert diese als relationale Ergebnismenge.

Wie man am folgenden Code-Beispiel sieht, wird JSON_TABLE etwas anders eingesetzt als JSON_QUERY oder JSON_VALUE. JSON_TABLE ist eine sog. *Row Source*, wird also wie eine Tabelle, View oder Subquery in der FROM-Klausel der SQL-Abfrage verwendet.

```
Select
 j.reference,
 j.costcenter,
 j.requestor
from exttab_pojson, JSON_TABLE(
 json, '$'
 COLUMNS (
  reference varchar2(20) path '$.Reference',
  costcenter varchar2(5) path '$.CostCenter',
  requestor varchar2(30) path '$.Requestor'
  )
) j;
```

```
REFERENCE                 COSTC     REQUESTOR
--------------------      -----     --------------------
MSULLIVA-20141102         A50       Martha Sullivan
MSULLIVA-20141113         A50       Martha Sullivan
TRAJS-20140518            A50       Trenna Rajs
:                         :         :

10000 Zeilen ausgewählt.
```

JSON_TABLE ist auch in der Lage, geschachtelte Strukturen zu verarbeiten. Innerhalb der COLUMNS-Klausel lässt sich der Ausdruck NESTED PATH verwenden, um in ein JSON-Array einzusteigen. Nested Paths lassen sich beliebig schachteln, um JSON Daten mit mehreren Hierarchiestufen in einem Schritt relational abzubilden. JSON_TABLE generiert dann für jedes JSON-Dokument so viele Ergebniszeilen, wie es Array-Elemente gibt.

```
Select
 j.reference, j.costcenter,
 j.descr, j.unitprice
from exttab_pojson, JSON_TABLE(
 json, '$'
 COLUMNS (
  reference    varchar2(20)  path      '$.Reference',
  costcenter   varchar2(5)   path      '$.CostCenter',
```

```
nested path '$.LineItems[*]' columns (
   descr       varchar2(35)  path      '$.Part.Description',
   unitprice   number        path      '$.Part.UnitPrice',
   quantity    number        path      '$.Part.Quantity'
  )
 )
) j;

REFERENCE           COST   DESC                          UNITPRICE
------------------  ----   ------------------            -----------
DGREENE-20140701    A80    Will Smith: Video Compilation 19.95
DGREENE-20140701    A80    The Color Purple              19.95
DGREENE-20140701    A80    Riders of Destiny             27.95
DGREENE-20140708    A80                                  19.95
:                   :      :                             :

45260 Zeilen ausgewählt.
```

Wie man an diesem Beispiel sieht, werden die Werte der Attribute **Reference** und **CostCenter** außerhalb des Nested Paths wiederholt.

JSON_TABLE vereinigt die Funktionalität von JSON_VALUE, JSON_QUERY und JSON_EXISTS in einer Funktion. Das folgende Beispiel zeigt, wie sich JSON-Fragmente oder die Existenz eines JSON-Attributs in die Ergebnismenge von JSON_TABLE integrieren lassen.

```
Select
 j.reference, j.ship_instr, j.many_items
from exttab_pojson, JSON_TABLE(
 json, '$'
 COLUMNS (
  reference    varchar2(20)                path  '$.Reference',
  ship_instr   varchar2(4000) format json  path  '$.ShippingInstructions',
  many_items   varchar2(5)  exists         path  '$.LineItems[4]'
  )
) j

REFERENCE           SHIP_INSTR                        MANY_ITEMS
------------------  --------------------------------  ----------
MSULLIVA-20141102   {"name":"Martha Sullivan","Address":{"st true
         reet":"200 Sporting Green","city":"South
          San Francisco","state":"CA","zipCo …
MSULLIVA-20141113   {"name":"Martha Sullivan","Address":{"st true
         reet":"200 Sporting Green","city":"South
          San Francisco","state":"CA","zipCo …
TRAJS-20140518      {"name":"Trenna Rajs","Address":{"street false
         ":"200 Sporting Green","city":"South San
          Francisco","state":"CA","zipCode":9 …
```

FORMAT JSON aktiviert die Funktionalität von JSON_QUERY; das Schlüsselwort **EXISTS** die von JSON_EXISTS. Gibt man nichts an, so wird JSON_VALUE als Default genutzt. Alle

Parameter für die drei SQL/JSON-Funktionen (Error-Handler, PRETTY, WRAPPER) können hier für jede Spalte einzeln definiert werden. Zudem lässt sich ein globaler Error-Handler für die ganze Funktion definieren.

Sind von einer SQL-Abfrage mehrere JSON-Attribute betroffen oder sollen die Funktionen JSON_QUERY, JSON_EXISTS und JSON_VALUE gemeinsam in einer Abfrage eingesetzt werden, sollte man JSON_TABLE in Erwägung ziehen, da sich die ganze Aufgabe hier mit einem einzigen Funktionsaufruf erledigen lässt.

2.2.6 SIMPLIFIED QUERY SYNTAX

Wenn man sich, während der Entwicklungszeit, noch mit dem JSON-Dokument und seinen Attributen vertraut macht, kann das ständige Formulieren von JSON_VALUE, JSON_QUERY oder JSON_TABLE-Funktionsaufrufen recht mühsam sein.

Um dem Entwickler das Leben etwas leichter zu machen, hat Oracle die sog. Simplified Query Syntax eingeführt. JSON-Attribute, die keine Arrays sind, können, wie in einer Programmiersprache, mit der „Punkt-Notation“ angesprochen werden. Die folgende Abfrage liefert das gleiche Ergebnis wie die erste JSON_TABLE-Abfrage in Abschnitt 2.2.4, ist von der Syntax her aber wesentlich einfacher.

```
select e.json.Reference, e.json.CostCenter, e.json.Requestor
from exttab_pojson e;

REFERENCE                COSTCENTER        REQUESTOR
-------------------      -----------       --------------------
MSULLIVA-20141102        A50               Martha Sullivan
MSULLIVA(-20141113       A50               Martha Sullivan
TRAJS-20140518           A50               Trenna Rajs
TRAJS-20140520           A50               Trenna Rajs
:                        :                 :

10000 Zeilen ausgewählt.
```

Damit diese Syntax funktioniert, muss die Tabellenspalte mit einem ein Check-Constraint IS JSON versehen werden (Abschnitt 2.1.3) – die Datenbank muss sich darauf verlassen können, dass die Spalte tatsächlich JSON enthält. Ohne Check-Constraint führt das Verwenden der Simplified Syntax zu einer Fehlermeldung.

JSON-Arrays können mit der vereinfachten Syntax zwar angesprochen, aber nicht, wie bei JSON_TABLE, in einzelne Tabellenzeilen zerlegt werden. Spricht man per Punktnotation ein Array-Attribut an, so liefert Oracle hierfür auch ein Array zurück.

```
select e.json.Reference, e.json.LineItems.Part.UnitPrice
from exttab_pojson e;

REFERENCE            LINEITEMS
-------------------- ----------------------------------------
MSULLIVA-20141102    [19.95,19.95,19.95,19.95,19.95]
MSULLIVA-20141113    [19.95,19.95,27.95,27.95,19.95]
TRAJS-20140518       [19.95,19.95,19.95]
TRAJS-20140520       [19.95,27.95,19.95]
:                    :

10000 Zeilen ausgewählt.
```

Das Datentyp-Mapping und das Verhalten im Fehlerfall können hier nicht beeinflusst werden; Ziel ist hier die sehr einfach nutzbare Syntax. Möchte man hier ein anderes Verhalten haben, so muss man die SQL/JSON-Funktionen JSON_VALUE, JSON_QUERY oder JSON_TABLE nutzen.

2.2.7 PERFORMANCE-ASPEKTE

Beim Arbeiten mit JSON in der Datenbank entfällt der größte Aufwand auf JSON-Parsing. Da das JSON als Text gespeichert wird, muss der JSON-Parser bei jedem Zugriff aktiv werden, den Text verarbeiten und die angefragten Attribute extrahieren. Der JSON-Parser in der Oracle-Datenbank ist

ein sehr effektiver *Streaming Parser*, das bedeutet, dass die JSON-Inhalte *nicht* komplett im Hauptspeicher aufgebaut werden. Vielmehr wird das Dokument der Reihe nach gelesen; sobald der Parser auf ein angefragtes Attribut stößt, wird ein „Match" ausgelöst. Wenn der Parser sich sicher ist, dass die Abfrage ausgeführt wurde, obwohl das Dokument noch *nicht* komplett verarbeitet wurde, bricht er ab – es wird also kein Aufwand verschwendet. Aus diesem Grund sind JSON-Operationen schneller, wenn die abgefragten Attribute zu Beginn des JSON-Dokuments zu finden sind.

Zunächst sollte man sicherstellen, dass man so wenig Parsing-Vorgänge durchführt, wie möglich. Das folgende Beispiel illustriert das. Werden zwei Attribute jeweils mit JSON_VALUE abgefragt und verwendet man JSON_EXISTS in der WHERE-Klausel, so führt dies zu drei Parsing-Vorgängen.

```
select
 json_value(json, '$.Reference') as ref,
 json_value(json, '$.Requestor') as req
from exttab_pojson
where json_exists(json, '$.LineItems[6]')
```

Erledigt man den gleichen Job mit JSON_TABLE, so werden die JSON-Dokumente nur noch einmal geparst.

```
select ref, req
from exttab_pojson, json_table(
 json,
 '$'
 columns (
  ref       varchar2(30)          path    '$.Reference',
  req       varchar2(30)          path    '$.Requestor',
  li_gt5    varchar2(5) exists    path    '$.LineItems[6]'
 )
)
where li_gt5 = 'true'
```

Nutzt man JSON_TABLE mit NESTED PATHs so ist es vorteilhaft, wenn alle Werte außerhalb des NESTED PATHs zuerst geparst werden können, damit alle Spalten eine Zeile sofort bekannt sind und zurückgegeben werden können. Ist das nicht der Fall, so muss der JSON-Parser Informationen zwischenpuffern. Dazu ein Beispiel: **{a:1, b:[1,2,3]}** ist besser als **{b:[1,2,3], a:1}**. Im zweiten Fall müssen zuerst alle Werte für **b** gelesen und gepuffert werden, bevor der Wert für **a** bekannt ist und die Ergebniszeile aufgebaut werden kann.

Stellt man fest, dass man immer wieder die gleiche Abfrage auf eine Tabelle mit JSON-Dokumenten durchführt, so ist die Übernahme in eine relationale Tabelle oder das Erzeugen einer Materialized View empfehlenswert. Danach entfallen die Parsing-Vorgänge ganz.

Auch für die in Abschnitt 2.1.2 vorgestellte IS JSON-Funktion, die auf der Tabelle als Check-Constraint eingesetzt werden kann, gibt es Performanceaspekte zu beachten. Wie bereits erwähnt, kann mit der Klausel WITH UNIQUE KEYS geprüft werden, ob in den JSON-Dokumenten doppelte Attributnamen vorkommen. Da die Attributnamen als Schlüssel zum Abrufen der Werte verwendet werden, können doppelte Schlüssel zu Problemen führen.

Das Sicherstellen eindeutiger Attributnamen kostet allerdings etwas, wie der folgende Vergleich zeigt: Die erste Abfrage stellt nur fest, welche JSON-Dokumente syntaktisch korrekt sind – die zweite führt die Prüfung WITH UNIQUE KEYS durch.

```
select count(*) from exttab_pojson
where json is json strict;

 COUNT(*)
----------
   10000

1 Zeile wurde ausgewählt.

Abgelaufen: 00:00:01.34
```

```
select count(*) from exttab_pojson
where json is json strict with unique keys;

 COUNT(*)
----------
   10000

1 Zeile wurde ausgewählt.

Abgelaufen: 00:00:04.63
```

2.3 INDIZIERUNG

Mit Indizes können SQL-Abfragen auf JSON-Dokumente beschleunigt werden. Allerdings beschleunigt ein Index stets die Selektion der SQL-Abfrage, also den in der WHERE-Klausel verwendeten Operator. Funktionen, die in der Projektion, also der SELECT-Liste verwendet werden, lassen sich mit einem Index nicht beschleunigen. Das ist auch bei der Anwendung für JSON-Dokumente wichtig: Eine SQL-Abfrage, die mit JSON_TABLE eine relationale Sicht auf eine JSON-Tabelle bereitstellt, kann durch Indizes nicht beschleunigt werden. Geht es jedoch darum, anhand eines Kriteriums bestimmte JSON-Dokumente in einer Tabelle zu finden (zu selektieren), ist ein Index eine gute Maßnahme zur Verbesserung der Performance.

Indizes können jedoch nicht auf externe Tabellen erzeugt werden; um die Purchaseorder-Dokumente, mit denen in den bisherigen Abschnitten gearbeitet wurde, indizieren zu können, müssen diese also von der externen Tabelle exttab_pojson in eine „normale“ Tabelle kopiert werden.

```
create table tab_pojson as select * from exttab_pojson;
```

2.3.1 FUNKTIONSBASIERENDE INDIZES

Sollen JSON-Dokumente stets anhand eines bestimmten Attributs gefunden werden, so kann es mit einem einfachen JSON_VALUE-Ausdruck indiziert werden. Das folgende Beispiel zeigt dies.

```
create index idx_pojson_reference
on tab_pojson(json_value(json, '$.Reference'));
```

Fragt man das Attribut **Reference** nun gezielt ab, so wird der Index zur Abfragebeschleunigung verwendet, wie der Ausführungsplan zeigt.

```
select * from tab_pojson
where json_value(json, '$.Reference') = 'TRAJS-20140520';
```

```
-------------------------------------------------------------------------
| Id  | Operation                           | Name                | Rows|
-------------------------------------------------------------------------
|  0  | SELECT STATEMENT                    |                     |     |
|  1  |  TABLE ACCESS BY IDX ROWID BATCHED | TAB_POJSON          |   1 |
|* 2  |   INDEX RANGE SCAN                  | IDX_POJSON_REFERENCE |   1 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  2 - access("TAB_POJSON"."SYS_NC00002$"='TRAJS-20140520')
```

Technisch ist dieser Index ein *funktionsbasierter Index*. Alle skalaren JSON-Attribute können auf diese Art und Weise indiziert werden (nicht möglich ist diese Indizierung für JSON-Arrays). Wenn das Attribut, anhand dessen man suchen möchte, im Vorfeld nicht bekannt ist, sieht die Datenbank den JSON-Volltextindex vor.

2.3.2 ORACLE JSON SEARCH INDEX

Mit der Volltexttechnologie Oracle TEXT lassen sich JSON-Dokumente komplett indizieren; ein so erstellter Index erfasst alle JSON-Attribute im Dokument. Folgerichtig ist anschließend eine komplett freie Suche anhand beliebiger Attribute möglich. Ein JSON-Volltextindex wird wie folgt erzeugt.

```
create index idx_pojson_search on tab_pojson (json)
INDEXTYPE is CTXSYS.CONTEXT
parameters (
 'section group CTXSYS.JSON_SECTION_GROUP
  sync (on commit)'
);
```

Volltextindizes unterscheiden sich technisch sehr stark von normalen Indizes – eine vollständige Abhandlung über alle Unterschiede würde den Rahmen dieses Dojos bei Weitem sprengen. Hier sei nur erwähnt:

- Volltextindizes sind prinzipiell asynchron und es wird eigens festgelegt, wann der Index mit Änderungen an der Tabelle synchronisiert werden soll – im Beispiel oben ist das ON COMMIT. Es sind aber auch andere Varianten möglich.
- Der Volltextindex selbst wird in Tabellen gespeichert, die von Oracle generiert und verwaltet werden. Man erkennt sie an dem Präfix DR$. Man kann diese Tabellen ansehen, Änderungen daran sollte man nicht vornehmen.
- Volltextindizes können umfangreich parametrisiert werden; das obige CREATE INDEX-Kommando ist ein Minimalbeispiel. Auch dies würde im Rahmen dieses Dojo zu weit gehen.

2.3.2.1 ABFRAGE MIT JSON_TEXTCONTAINS

Wurde der Index erfolgreich erstellt, kann er für die SQL/JSON-Funktionen JSON_EXISTS und JSON_TEXTCONTAINS herangezogen werden. Zunächst ein Beispiel mit JSON_EXISTS.

```
select count(*) from tab_pojson
where json_exists (json, '$.ShippingInstructions.Address.zipCode');

 COUNT(*)
----------
   6369
```

Der Ausführungsplan zeigt, dass der neue Index zum Einsatz kam.

```
-----------------------------------------------------
| Id | Operation          | Name                | Rows |
-----------------------------------------------------
|  0 | SELECT STATEMENT   |                     |      |
|  1 |  SORT AGGREGATE    |                     |    1 |
|* 2 |   DOMAIN INDEX     | IDX_POJSON_SEARCH   |    5 |
-----------------------------------------------------
```

Wesentlich mächtiger ist die SQL/JSON-Funktion JSON_TEXTCONTAINS; diese erlaubt das Auffinden von JSON-Dokumenten anhand beliebiger Attribute: Ob Array oder nicht, spielt keine Rolle.

```
select json_value(t.json, '$.Reference' error on error) ref
from tab_pojson t
where json_textcontains(
  t.json,
   '$.LineItems.Part.Description',
   'Magic'
);

REF
--------------------------------
OTUVAULT-20141031
NCAMBRAU-20141104
JFLEAUR-20141106
:
```

Interessant ist auch hier der Blick in den Ausführungsplan – und besonders in den Abschnitt der Predicates.

```
-----------------------------------------------------------
| Id  | Operation                    | Name               |
-----------------------------------------------------------
|   0 | SELECT STATEMENT             |                    |
|   1 |  TABLE ACCESS BY INDEX ROWID | TAB_POJSON         |
|* 2  |   DOMAIN INDEX               | IDX_POJSON_SEARCH  |
-----------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
2 - access("CTXSYS"."CONTAINS"("T"."JSON",'{Magic}
            INPATH(/LineItems/Part/Description)')>0)
```

Der Zugriff auf den Domain-Index erfolgt mithilfe der Funktion CTXSYS.CONTAINS. Langjährige Nutzer des Volltextindex Oracle TEXT kennen diese Funktion sehr gut, denn sie wird stets verwendet, wenn mit Oracle TEXT eine Volltextrecherche gemacht wird. Offensichtlich werden die Parameter von JSON_TEXTCONTAINS in die Oracle TEXT-Syntax mit der Funktion CONTAINS übersetzt.

2.3.2.2 ORACLE TEXT-ABFRAGEN MIT CONTAINS

Das bedeutet aber, dass man die Oracle TEXT-Syntax auch direkt verwenden kann. Die folgende Abfrage liefert das gleiche Ergebnis wie die obige.

```
select json_value(t.json, '$.Reference' error on error) ref
from tab_pojson t
where CONTAINS(json, '{Magic} INPATH (/LineItems/Part/
Description)') > 0;
```

Allerdings bietet Oracle TEXT noch weitere Möglichkeiten an. So gibt es den FUZZY-Operator, der eine Ähnlichkeitssuche möglich macht. Man findet also auch JSON-Dokumente mit falsch geschriebenen Wörtern.

```
select json_value(t.json, '$.Reference' error on error) ref
from tab_pojson t
where CONTAINS(json, '?{Magie} INPATH (/LineItems/Part/
Description)') > 0;
```

Suchausdrücke lassen sich auch kombinieren.

```
select json_value(t.json, '$.Reference' error on error) ref
from tab_pojson t
where CONTAINS(
 json,
 '?{Magie} INPATH (/LineItems/Part/Description) or
  ?{San Franzisko} INPATH (/ShippingInstructions)'
) > 0;
```

2.4 EIN UMFASSENDES BEISPIEL: ERDBEBEN

Anhand der in Abschnitt 2.1.5 vorgestellten Beispieldaten „Earthquakes“ wird der Einsatz der verschiedenen SQL/JSON-Funktionen nun nochmals vorgestellt.

Das USGS stellt verschiedene JSON-Feeds für Erdbebendaten bereit. Neben den Feeds mit den Erdbeben der letzten Stunde oder des letzten Tags wird auch ein Feed mit allen registrierten Erdbeben der letzten 30 Tage bereitgestellt. Der folgende Aufruf lädt dieses (größere) JSON-Dokument in einen CLOB. Wenn die Datenbank hinter einer Firewall steht, muss der Proxy-Server vorher gesetzt werden.

```
begin
 utl_http.set_proxy('proxyserver.meinefirma.de:80');
end;

create table usgs_earthquake_stage as
select httpuritype(
 'http://earthquake.usgs.gov/earthquakes/feed/v1.0/'||
 'summary/all_month.geojson'
).getclob() as json from dual;
```

Anschließend befindet sich in der Tabelle ein einziges JSON-Dokument mit der folgenden Struktur:

```
{
 "type": "FeatureCollection",
 "metadata": {...},
 "features": [
  {"id":  1, "properties": {...},
  {"id":  2, "properties": {...},
  :
  {"id": 8192, "properties": {...}
 ]
}
```

Die einzelnen Erdbeben liegen also als Array-Elemente vor. Würde man den gleichen Feed nun eine Woche später nochmals abrufen, würde die gleiche Struktur mit den Erdbeben der dann letzten 30 Tage zurückgegeben. Beide JSON-Dokumente hätten eine Schnittmenge von etwa drei Wochen.

Um damit umgehen zu können, zerlegen wir das große Dokument zunächst in einzelne JSON-Dokumente – eines pro Erdbeben. Weiterhin hat das USGS jedes Erdbeben mit einem eindeutigen Schlüssel versehen, welches im Attribut **id** jedes Array-Elements enthalten ist. Damit kann eine Tabelle erstellt und aus der Staging-Tabelle heraus befüllt werden.

```
create table usgs_earthquake_json(
 usgs_id       varchar2(20) primary key,
 usgs_json     varchar2(4000)
);
insert into usgs_earthquake_json (
 select
  b.id,
  b.json
 from usgs_earthquake_stage a, json_table(
  a.json,
  '$.features[*]'
  columns (
   id       varchar2(20)        path '$.id',
   json     varchar2(4000) format json path '$'
   )
 ) b
);
```

Jedes Erdbeben ist dann genau einmal in der Tabelle vorhanden. Mithilfe der Tabellenspalte **USGS_ID**, dem JSON-Attribut **id** und dem SQL MERGE-Kommando können künftige JSON-Daten nun bequem in die Tabelle „hineingemischt" werden. Das folgende Codebeispiel mischt den Feed mit den Erdbeben der letzten Stunde in die Tabelle. Da das JSON-Format verwendet wird, bleibt der Setup stabil, auch wenn das USGS sich entscheiden sollte, neue Attribute einzuführen oder bestehende wegfallen zu lassen.

```
merge into usgs_earthquake_json dst
using (
 select b.id, b.json
 from json_table(
  httpuritype(
   'http://earthquake.usgs.gov/earthquakes/feed/v1.0/' ||
   'summary/all_hour.geojson'
  ).getclob(),
  '$.features[*]' columns (
   id  varchar2(20)        path '$.id',
   json varchar2(4000) format json path '$'
   )
 ) b
) src on (src.id = dst.usgs_id)
when not matched then insert values (src.id, src.json);
```

Nun können mit der Funktion JSON_TABLE verschiedenste Abfragen durchgeführt werden. Das folgende Beispiel listet die Orte aller Erdbeben mit einer Stärke von 6.0 und höher.

```
select place, mag, unix_time
from usgs_earthquake_json, json_table(
 usgs_json,
 '$'
 columns (
  unix_time number          path    '$.properties.time',
  mag       number          path    '$.properties.mag',
  place     varchar2(40)    path    '$.properties.place'
 )
) where mag >= 6;
```

```
PLACE                                  MAG   UNIX_TIME
-------------------------------------- ----- -------------------
71km SSW of Redoubt Volcano, Alaska    6.6     1438137359000
93km S of Krajan Tambakrejo, Indonesia 6.0     1437894308730
230km W of Abepura, Indonesia          7.0     1438033281440
:                                      :       :
```

Ein Unix-Datum in ein Oracle DATE umzuwandeln, ist recht einfach ...

```
select place, mag, date'1970-01-01' + (unix_time / 86400000) datetime
from usgs_earthquake_json, json_table(
 :
) where mag >= 6;

PLACE                                  MAG   DATETIME
-------------------------------------- ----- -------------------
71km SSW of Redoubt Volcano, Alaska    6.6   29.07.2015 02:35:59
93km S of Krajan Tambakrejo, Indonesia 6.0   26.07.2015 07:05:09
230km W of Abepura, Indonesia          7.0   27.07.2015 21:41:21
:                                      :     :
```

Die Daten des USGS enthalten (natürlich) auch eine Geokoordinate für das Erdbeben – diese ist im JSON-Attribut **geometry** enthalten. Also kann die COLUMNS-Klausel im JSON_TABLE entsprechend erweitert werden.

```
select place, …, lon, lat
from usgs_earthquake_json, json_table(
 usgs_json,
 '$'
 columns (
  :
  lon   number   path   '$.geometry.coordinates[0]',
  lat   number   path   '$.geometry.coordinates[1]'
 )
) where mag >= 6;
```

```
PLACE                        MAG  DATETIME             LON         LAT
-------------------------- ----- -------------------  ----------- ---
71km SSW of Redoubt Volca  6.6   29.07.2015 02:35:59  -153.2020   59.8963
93km S of Krajan Tambakre  6.0   26.07.2015 07:05:09  112.6938    -9.2469
230km W of Abepura, Indon  7.0   27.07.2015 21:41:21  138.5084    -2.6829
:                          :     :                    :           :
```

Nimmt man die Geodaten-Funktionen der Datenbank hinzu, so lässt sich die Entfernung der Erdbeben zu einem gegebenen Punkt („München“) sehr leicht errechnen.

```
select
 place,
 mag,
 date'1970-01-01' + (unix_time / 86400000) datetime,
 sdo_geom.sdo_distance(
  sdo_geometry(2001, 4326, sdo_point_type(lon, lat, null), null,
null),
  sdo_geometry(2001, 4326, sdo_point_type(11.5, 48.2, null), null,
null),
  1, 'unit=km'
 ) as distanz_km
from usgs_earthquake_json, json_table(
 :
) where mag >= 6;
```

```
PLACE                       MAG   DATETIME             DISTANZ_KM
--------------------------- ----- -------------------  ----------
71km SSW of Redoubt Volca   6.6   29.07.2015 02:35:59  7941
93km S of Krajan Tambakre   6.0   26.07.2015 07:05:09  11600
230km W of Abepura, Indon   7.0   27.07.2015 21:41:21  12886
:                           :     :                    :
```

Darüber hinaus sind mit den Mitteln von SQL natürlich noch zahlreiche weitere Auswertungen möglich.

3 Oracle als Document Store: Simple Oracle Document Access (SODA)

Gemeinsam mit dem neuen JSON-Datenbank-Feature stellt Oracle eine Dokument-API für JSON-Daten zur Verfügung: die *Simple Oracle Document Access* API oder kurz *SODA*.

SODA ist eine Dokumenten-Schnittstelle, das heißt sie erlaubt Operationen auf einem oder mehreren *Dokumenten*. Dokumente können hierbei JSON- oder auch andere Binärdaten (Bilder, PDF-Dokumente und mehr) sein. Dokumente werden in einer Collection abgelegt. Eine *Collection* besitzt einen eindeutigen Namen und darin enthaltene Dokumente sind über einen eindeutigen Schlüssel (ID) identifizierbar. Darüber hinaus müssen keine Eigenschaften definiert werden, es gibt also kein Schema, keine Constraints und auch keine Data Definition Language.

Der eindeutige Schlüssel wird automatisch von der Datenbank erstellt und zusammen mit dem Dokument, einem Zeitstempel und einer Versionsnummer gespeichert. Das Dokument kann dann anhand der ID selektiert, gelöscht oder verändert werden. Zusätzlich zu diesen Funktionen, die man auch als *CRUD* bezeichnet (*Create, Read, Update, Delete*) können Dokumente auch anhand ihrer Eigenschaften selektiert werden („Alle Kunden einer bestimmten Postleitzahl“).

Hierfür gibt es eine neue, einfach erlernbare und auf JSON basierende „Abfragesprache“.

Collections und *Dokumente* sind Begriffe der SODA API; über die konkrete Implementierung sagt das noch nichts aus. In der Oracle-Datenbank wird eine Collection auf eine *Tabelle* abgebildet. Ein Dokument ist demzufolge eine *Zeile* mit dem konkreten Dokument als BLOB-Spalte und den Metadaten als zusätzliche Spalten. Während die SODA API dem Entwickler dann einen NoSQL- beziehungsweise Document-Store-ähnlichen Zugriff auf diese Tabelle bietet, kann der SQL-Entwickler darauf die im Abschnitt 2 erläuterten SQL/JSON-Funktionen anwenden und die Daten so umfangreich analysieren. Die SODA API bietet folgende Funktionen an.

- Erstellen einer Collection
- Löschen einer Collection
- Auflistung aller Collections
- Einfügen eines Dokuments in eine Collection
- Löschen eines Dokuments anhand der ID
- Ändern eines Dokuments anhand der ID
- Selektion eines Dokuments anhand der ID

- Finden aller Dokumente
- Finden von Dokumenten anhand einer Abfrage bzw. eines Filters
- BULK-Operationen und Erstellen von Indizes.

SODA wird zur Zeit in zwei Implementierungen (REST und Java) bereitgestellt – weitere Implementierungen sind geplant.

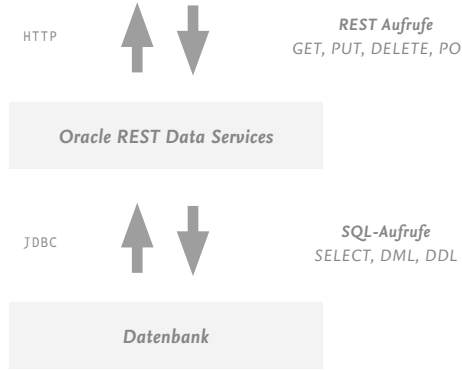
3.1 ORACLE REST DATA SERVICES (ORDS)

REST ist ein de facto Standard, um Dienste mit einer parametrisierten URL über HTTP-Verbindungen (Endpoints) aufzurufen und Inhalte (Ressourcen) zurückzubekommen. Im Grunde genommen ist jede Website ein REST-Service, denn es wird mit einer HTTP-Methode (GET) eine URL (Endpoint) abgerufen und man bekommt eine HTML-Ressource zurück. Diese wird dann im Browser für den Menschen lesbar aufbereitet. Wenn Softwaresysteme miteinander kommunizieren, wird kein HTML, sondern eher JSON oder XML ausgetauscht, das Prinzip ist jedoch das gleiche. Neben GET stehen dann noch andere HTTP-Methoden (beispielsweise PUT, DELETE oder POST) zur Verfügung. Ein aktueller Trend sind sogenannte *Microservice-Architekturen*, bei der eine Anwendung in verschiedene, kleinteilige

Dienste aufgeteilt wird, die über REST kommunizieren. Da REST von allen Programmiersprachen unterstützt wird, können Dienste verschiedener Sprachen problemlos miteinander kommunizieren.

Oracle Rest Data Service (ORDS) stellt als Web-Listener REST-Services auf Basis der Oracle-Datenbank zur Verfügung. ORDS ist Teil der Datenbanklizenz ab der Standard Edition und bietet ab Release 3.0 eine REST-Schnittstelle für die SODA API an. ORDS ist in Java implementiert und wird als „Listener" vor die Datenbank gestellt. Die auf dem Listener entstehende Last ist sehr gering, da die Logik der REST-Dienste größtenteils in der Datenbank abläuft. Insofern hat man die freie Wahl, ob man ORDS auf der gleichen Maschine wie die Datenbank oder auf einer anderen installiert.

Produktive ORDS-Instanzen sollten in einen Java-Container (Tomcat, Glassfisch oder Weblogic) installiert werden (*Deployment*). Für Entwicklungszwecke reicht die *Standalone Variante* völlig aus; in diesem Fall kann das Java-Archiv *ords.war* direkt gestartet werden – ORDS bringt dann seinen eigenen Webserver mit. Die Installation von ORDS und die Aktivierung des SODA-Dienstes wird hier aus Platzgründen nicht beschrieben. Das Blog **JSON, REST und mehr - mit der Oracle Datenbank** (siehe Weitere Informationen) beschreibt den Vorgang im Detail. Auch im Oracle SQL Developer ist ORDS integriert – dort kann er per Mausklick gestartet werden.



Das Diagramm veranschaulicht die Rolle von ORDS: Clients authentifizieren sich bei ORDS und rufen einen REST-Service auf. Nach erfolgreicher Autorisierung werden die dem REST-Dienst zugeordneten SQL-Operationen in der Datenbank ausgeführt.

Um die SODA-API für ein Datenbankschema freizuschalten, muss – im Datenbankschema selbst – folgender PL/SQL-Aufruf ausgeführt werden. Für den Rest des Kapitels wird angenommen, dass mit dem Datenbankschema SCOTT gearbeitet wird – der PL/SQL-Aufruf muss also als SCOTT gemacht werden.

```
begin
 ORDS.ENABLE_SCHEMA(
  p_enabled =>       true,
  p_schema =>       'SCOTT',
  p_url_mapping_type =>  'BASE_PATH',
  p_url_mapping_pattern => 'scott',
  p_auto_rest_auth =>   false
 );
end;
/

commit
/
```

ORDS.ENABLE_SCHEMA gibt REST-Dienste für ein Datenbankschema frei. Das Datenbankschema **SCOTT** wird dabei auf den URL-Präfix **/scott** abgebildet. Für Test und Entwicklung sei die Authentifizierung zunächst mit dem Parameter **p_auto_rest_auth** abgeschaltet (**false**).

Weiterhin wird (für Test und Entwicklung) auch die Autorisierung (Wer darf was?) mit dem folgenden Aufruf deaktiviert.

```
begin
 ORDS.DELETE_PRIVILEGE_MAPPING (
  p_privilege_name => 'oracle.soda.privilege.developer',
  p_pattern =>    '/soda/*'
 );
end;
```

```
/

commit
/
```

Ob ORDS und das Datenbankschema erfolgreich eingerichtet sind, lässt sich durch einen einfachen URL-Aufruf in einem Webbrowser überprüfen. Hierbei ist <hostname> durch den Rechnernamen oder die IP-Adresse des Rechners zu ersetzen, auf dem ORDS läuft. 8080 ist der Standard-TCP/IP-Port von ORDS. Wurde bei der Installation von ORDS eine andere Port-Nummer angegeben, muss man natürlich diese nehmen.

```
http://<hostname>:8080/ords/scott/soda/latest
```

Als Antwort sollte lediglich ein leeres JSON-Array erscheinen, welches besagt, dass es derzeit noch keine SODA-Collections in diesem Schema gibt.

```
[]
```

Bevor die einzelnen SODA-Operationen vorgestellt werden, seien hier die einzelnen Komponenten der URL beschrieben. Der URL-Aufbau folgt dem Oracle-Standard für REST und ist dementsprechend allgemein.

http://	Der Zugriff nutzt das HTTP-Protokoll.
<hostname>	Der Name/die IP-Adresse der Rechners auf dem ORDS läuft.
8080	Port-Nummer auf dem ORDS Anfragen entgegen nimmt.
ords	URL-Präfix for Oracle REST Data Services; in einem Java-Container könnten neben ORDS noch andere Dienste laufen.
scott	Name des Datenbankschemas (Nutzer). In diesem Schema werden die Collections (Tabellen) angelegt. Es können verschiedene Schemata genutzt werden (auch gleichzeitig).
soda	Referenziert die SODA-API. Aus einem Datenbankschema können mehrere REST-Dienste (auch PL/SQL basierende) bereitgestellt werden. **soda** spricht immer die SODA-API an.
latest	Die Version des Dienstes. Da sich der Funktionsumfang eines Dienstes im Laufe der Zeit ändern kann, ermöglicht die Versionsnummer eine genaue Spezifizierung. **latest** steht für die aktuellste Version. In diesem Dojo wird nur **latest** verwendet.

3.2 REST-CLIENTS

Der REST-Client ist das Gegenstück zum REST-Server. REST-Clients bauen eine HTTP oder HTTPS-Verbindung zum Server auf und rufen dort REST-Services, mit den verschiedenen

HTTP-Methoden (GET, POST), auf. Einfache REST-Dienste könnte man sogar nur mit dem Browser aufrufen, allerdings unterstützt dieser nur die HTTP GET-Methode – zum Testen anderer Methoden wie PUT oder DELETE braucht es eigene REST-Clients, die oft als Browser-Add-ons geliefert werden. Beispiele sind **Postman** für Google Chrome oder **RESTClient** für Firefox. Zum Testen und Ausprobieren sind diese Clients sehr hilfreich, da man dann kein eigenes Programm schreiben und ggfs. kompilieren muss.

Alternativ lassen sich REST-Dienste auch mit der Kommandozeile testen – hierzu ist das freie Werkzeug **curl** die beste Alternative, die in diesem Dojo von nun an auch verwendet wird.

3.3 OPERATIONEN MIT DER SODA API

3.3.1 ERZEUGEN EINER NEUEN COLLECTION

Im Folgenden wird eine neue Collection zum Speichern der im vorigen Kapitel vorgestellten Erdbebendaten erzeugt. Diese Collection soll **earthquakes** heißen. Dazu muss ein PUT-Request ausgeführt werden. Der Name der Collection wird einfach an die SODA-URL angehängt. Der **curl** Parameter **–I** sorgt dafür, dass der vom Server zurückgegebene *HTTP-Status* angezeigt wird.

```
$ curl --request PUT localhost:8080/ords/scott/soda/latest/
earthquakes -I

HTTP/1.1 201 Created
Cache-Control: private,must-revalidate,max-age=0
Location: http://localhost:8080/ords/scott/soda/latest/
earthquakes/
Content-Length: 0
```

Führt man den gleichen Request noch einmal aus, so ändert sich der Status Code – denn die Collection existiert bereits.

```
HTTP/1.1 200 OK
Cache-Control: private,must-revalidate,max-age=0
Content-Length: 0
```

Nach Ausführung dieser Operation kann dieselbe URL nochmal mit einem GET aufgerufen werden – dann werden Angaben zum Inhalt der Collection zurückgegeben. Im Moment ist diese natürlich noch leer.

```
$ curl --request GET localhost:8080/ords/scott/soda/latest/
earthquakes

{
  "items": [],
  "hasMore": false,
```

```
  "count": 0,
  "offset": 0,
  "limit": 100,
  "totalResults": 0,
  "links": []
}
```

3.3.2 ANZEIGEN EXISTIERENDER COLLECTIONS

Dieser Aufruf wurde im Kapitel 3.1 bereits zum Testen des ORDS-Setups verwendet. Nachdem eine Collection erzeugt wurde, wird nun ein anderes Ergebnis zurückgegeben.

```
$ curl --request GET localhost:8080/ords/scott/soda/latest
```

```
{
 "items": [
  {
   "name": "earthquakes",
   "properties": {
    "schemaName": "SCOTT",
    "tableName": "earthquakes",
    "keyColumn": {
     "name": "ID",
     "sqlType": "VARCHAR2",
     "maxLength": 255,
```

```
    "assignmentMethod": "UUID"
   },
   "contentColumn": {
    "name": "JSON_DOCUMENT",
    "sqlType": "BLOB",
    "compress": "NONE",
    "cache": true,
    "encrypt": "NONE",
    "validation": "STANDARD"
   },
   "versionColumn": {
    "name": "VERSION",
    "type": "String",
    "method": "SHA256"
   },
   "lastModifiedColumn": {
    "name": "LAST_MODIFIED"
   },
   "creationTimeColumn": {
    "name": "CREATED_ON"
   },
   "readOnly": false
  },
  "links": [{
   "rel": "canonical",
   "href": http://localhost:8080/ords/scott/soda/latest/
```

```
earthquakes
   }]
  }
 ],
 "more": false
}
```

Dieses, schon etwas längere JSON, listet nicht nur die Namen der existierenden Collections, es liefert vielmehr zusätzliche Informationen über die dahinter liegende Tabelle mit ihren Spalten. Man erfährt, dass die Tabelle den Namen **earthquakes** hat und dass die JSON-Daten in eine BLOB-Spalte mit Namen **JSON_DOCUMENT** abgelegt werden. Zusätzlich gibt es Informationen über die Metadaten, wie Version- und Änderungs-Zeitstempel. Diese Metadaten werden beim Speichern eines Dokuments vom System generiert und müssen nicht mit dem REST-Request angegeben werden. Möchte man das dennoch tun, so kann dies mit speziellen Angaben im REST-Request erreicht werden. Hierzu finden sich nähere Angaben in der Dokumentation zu SODA (siehe Weitere Informationen).

Diese Informationen zu den Tabellen- und Spaltennamen helfen auch beim Arbeiten mit SQL. Das Besondere der Oracle-Datenbank ist ja gerade, dass (auf die gleichen JSON-Daten) Schnittstellen in beide Richtungen bereitgestellt werden – SODA liefert einen Document-Store-Sicht, SQL liefert die RDBMS-Sicht.

3.3.3 LÖSCHEN EINER COLLECTION

Nur der Vollständigkeit halber sei erwähnt, wie sich Collections löschen lassen – ebenfalls per REST-Request. Nun wird aber die HTTP-Methode **DELETE** verwendet.

```
$ curl --request DELETE localhost:8080/ords/scott/soda/latest/
earthquakes
```

Die Datenbank führt bei dieser Operation ein **DROP TABLE** aus, die in der Collection/Tabelle enthaltenen Daten sind danach nicht mehr verfügbar. Innerhalb einer gewissen Zeitspanne kann sie allerdings aus dem Recycle Bin der Datenbank wiederhergestellt werden – dazu ist aber SQL erforderlich.

```
SQL> flashback table "earthquakes" to before drop;
```

Nun kann die wiederhergestellte Tabelle unbenannt, die Collection **earthquakes** neu erzeugt und schließlich die Daten von der alten in die neue Tabelle kopiert werden.

3.3.4 EINFÜGEN EINES DOKUMENTS

Nun kann der erste Datensatz als JSON eingefügt werden. Die Collection wird wiederum in der URL angesprochen; die HTTP-Methode **POST** definiert, dass Daten eingefügt werden sollen. Die einzufügenden Daten werden nicht in der URL kodiert, sondern im *Request-Body* mitgeschickt. Wichtig ist es, den HTTP-Header **Content-Type** mit **application/json** zu setzen. Das Kommando-

zeilenwerkzeug **curl** erlaubt es, die JSON-Daten direkt als Parameter zu übergeben (1) oder auf eine Datei zu verweisen (2).

1.

```
curl
 localhost:8080/ords/scott/soda/latest/earthquakes
 --request POST
 -H "Content-Type:application/json"
 -d '{"type": "FeatureCollection", …}'
```

2.

```
curl
 localhost:8080/ords/scott/soda/latest/earthquakes
 --request POST
 -H "Content-Type:application/json"
 --data @earthquake.json
```

Das erfolgreiche Einfügen des JSON-Dokuments wird mit einer Rückmeldung bestätigt.

```
{
 "items": [
  {
   "id": "566A9487558C474EB8622046157EB9EC",
   "etag": "44136FA355B3678A1146AD16F7E864 …",
   "lastModified": "2015-10-08T00:03:01.232000Z",
```

```
   "created": "2015-10-08T00:03:01.232000Z"
  }
 ],
 "hasMore": false,
 "count": 1
}
```

Folgende Informationen werden nach dem Einfügen zurückgeliefert.

id	Mit der ID wird das gerade eingefügte Dokument identifiziert.
etag	Hash-Wert, der für das Caching des Dokuments verwendet werden kann. Verändert sich das Dokument (durch ein Update) so verändert sich auch das **etag**. Vergleicht man **etag** aus dem Cache mit dem der Datenbank, so lässt sich leicht herausfinden, ob das Dokument in der Datenbank verändert wurde und somit der Cache nicht mehr aktuell ist. Die Verwendung eines Caches ist optional.
lastModified, created	Zeitstempel beim Einfügen des Dokuments.
hasMore, count	Die Anzahl der zurückgegebenen Elemente und ob weitere Elemente vorhanden sind, wird bei *jedem* REST-Request zurückgegeben. Das ist insbesondere wichtig, wenn eine Abfrage viele Dokumente selektiert.

Da HTTP und REST zustandslose Protokolle sind, wird das in der Datenbank obligatorische COMMIT automatisch ausgeführt. Beim Einfügen des Dokuments wird das JSON bereits geparst und validiert, wozu der in Abschnitt 2.1.3 erwähnte Check-Constraint zum Einsatz kommt. Ungültiges JSON wird also zurückgewiesen und der REST-Request wird mit einer Fehlermeldung beantwortet.

Wie man an diesem Beispiel sieht, wird der Schlüssel des Dokuments von der Datenbank vergeben und in der Antwort auf dem REST-Request an den Aufrufer zurückgegeben. In manchen Fällen möchte der Nutzer den Schlüssel jedoch selbst vergeben. Auch das ist möglich, jedoch muss in diesem Fall muss der Nutzer jedoch für die Eindeutigkeit des Schlüssels sorgen, damit nicht etwa versehentlich bereits vorhandene Daten überschrieben werden. Ob die ID vom Nutzer oder automatisch von der Datenbank vergeben wird, wird bei Erstellung der Collection festgelegt. Standardmäßig übernimmt die Datenbank diese Aufgabe – was in den meisten Fällen auch zu empfehlen ist.

3.3.5 SELEKTION EINES DOKUMENTS ANHAND DER ID

Anhand der beim Einfügen generierten ID lässt sich das Dokument schnell wiederfinden. Die ID wird einfach an die URL angefügt und ein HTTP GET-Request ausgeführt. Das

vormals eingefügte JSON-Dokument wird 1:1 von der Datenbank zurückgegeben.

```
$ curl
 --request GET
 localhost:8080/ords/scott/soda/latest/earthquakes/566A9487 …
```

3.3.6 ÜBERSCHREIBEN EINES DOKUMENTS

Schickt man an die auf ein Dokument zeigende URL einen POST-Request mit einem neuen Dokument, so wird das alte überschrieben. Wieder wird das neue Dokument im Request Body übergeben. Existiert das in der URL angesprochene Dokument nicht, so wird eine Fehlermeldung (HTTP-404) zurückgegeben.

```
$ curl
 localhost:8080/ords/scott/soda/latest/earthquakes/566A9487 …
 --request POST
 -H "Content-Type:application/json"
 -d '{"type": "FeatureCollection", …}'
```

3.3.7 LÖSCHEN EINES DOKUMENTS

Das Entfernen eines Dokuments aus einer Collection funktioniert mit einem HTTP DELETE-Request. Existiert das Dokument nicht, so gibt es eine Fehlermeldung (HTTP-404).

```
$ curl
 --request DELETE
 localhost:8080/ords/scott/soda/latest/earthquakes/566A9487 …
```

3.3.8 ALLE DOKUMENTE DER COLLECTION ANZEIGEN

Ein einfacher GET-Request mit der URL einer Collection liefert den Inhalt derselben zurück. Der einfachste Aufruf findet ohne Parameter statt.

```
$ curl
 --request GET
 localhost:8080/ords/scott/soda/latest/earthquakes
```

Bei diesem Aufruf werden sowohl die Metadaten, als auch die Dokumente selbst zurückgegeben. Die Ausgabe wird standardmäßig auf 100 Elemente beschränkt. Mit einem erneuten Request lassen sich dann die nächsten 100 Elemente laden – was man solange macht, bis man alle Dokumente abgerufen hat. Für dieses Blättern (*Paging*) gibt es eigene Parameter, die an die URL angehängt werden. Möchte man die zweite „Ergebnisseite“ laden, so setzt man die URL-Parameter **limit** und **offset** – das sieht wie folgt aus.

```
$ curl
 --request GET
 localhost:8080/ords/scott/soda/latest/earthquakes?offset=100&lim
it=100
```

Möchte man nur die Schlüssel abrufen (um diese ggfs. später weiterverarbeiten zu können), so lassen sich die konkreten Daten unterdrücken, indem man den Parameter **fields** verwendet. Das folgende Beispiel ruft nur Metadaten ab, überspringt die ersten 20 Einträge und ruft die dann folgenden nächsten 10 Einträge ab.

```
$ curl
 --request GET
 localhost:8080/ords/scott/soda/latest/earthquakes?
  fields=id&offset=20&limit=10
{
 items: [
 {
 id: "1FC039A8E67E457FB92537C6530BD89F",
 etag:015ABD7F5CC57A2DD94B7590F04AD8084273905EE33EC5CEBEAE62276A9
7F862",
 lastModified: "2015-10-09T05:39:47.266000Z",
 created: "2015-10-09T05:39:47.266000Z"
 },
 // weitere 9 Objekte wie oben
 ],
 hasMore: true,
 count: 10,
 offset: 20,
 limit: 1
}
```

Am Attribut **hasMore** in der JSON-Ausgabe kann man als Client erkennen, ob noch weitere Daten vorhanden sind und ggfs. ein erneuter Request abgesetzt werden muss. Die nächste Seite würde man dann mit **offset=30** abrufen. Diese Flexibilität ist sehr wichtig, wenn man Applikationen für verschiedene Endgeräte schreibt – steht nur wenig Bandbreite zur Verfügung (beispielsweise für mobile Geräte), so ist es sinnvoller, mit kleineren Ergebnismengen zu arbeiten.

3.3.9 ABFRAGEN MIT QUERY BY EXAMPLE (QBE)

Wie gezeigt wurde, kann ein Dokument anhand des eindeutigen Schlüssels leicht wiedergefunden werden; viel wichtiger ist es aber oft, ein oder mehrere Dokumente anhand ihrer Eigenschaften aufzufinden – zum Beispiel Kunden anhand ihres Umsatzes oder ihres Wohnortes.

Da die Dokumente als JSON in der Oracle-Datenbank gespeichert sind, kann man sie natürlich mit den in Abschnitt 2.2 vorgestellten SQL/JSON-Funktionen (JSON_VALUE, JSON_TABLE) abfragen. Jedoch müsste man hierfür SQL-Anfragen schreiben und diese eigens per REST-Schnittstelle publizieren. Das ist mit ORDS zwar möglich, bei vielen unterschiedlichen Abfragen aber doch recht aufwändig. Einfacher geht es ohne SQL mit *QBE-Abfragen*. **QBE** steht für *Query by Example* und bedeutet, das *die Anfrage selbst*

in JSON formuliert wird. Am leichtesten lässt sich dies mit Beispielen verdeutlichen; wir beziehen uns weiterhin auf die Erdbebendaten.

Schaut man sich die Erdbebendaten genauer an, so bemerkt man, dass jedes Erdbeben ein eigenes JSON-Feld namens **id** besitzt (nicht zu verwechseln mit der von der SODA-API automatisch vergebenen ID beim Einfügen). Möchte man ein Dokument anhand dieser (vom USGS vergebenen ID) selektieren, so kann man eine QBE-Abfrage in JSON wie folgt formulieren.

```
{"id": "nn00514488"}
```

Die Anfrage wird per REST-Request mit der HTTP-Methode POST ausgeführt. Die QBE-Abfrage selbst wird als *Request Body* übergeben. Daraufhin wird das Dokument mit dem Wert **nn00514488** im JSON-Attribut **id** zurückgegeben.

```
$ curl
 localhost:8080/ords/scott/soda/latest/earthquakes?action=query
 --request POST
 -H "Content-Type:application/json"
 -data '{"id": "nn00514488"}'
```

Um starke Erdbeben zu finden, nutzt man eine *Range-Query*: Es sind Erdbeben mit einer Stärke von 5 oder höher gefragt.

Im JSON-Dokument ist die Stärke des Erdbebens im Attribut **properties.mag** zu finden. Es ergibt sich folgende QBE-Abfrage.

```
{"properties.mag" : {"$gt": 5}}
```

Das Schlüsselwort **$gt** steht für *greater than* (>); bedeutet also, dass der Wert von **properties.mag** größer als 5 sein muss. Andere Schlüsselwörter sind bspw. **$lt** (*less than*), **$gte** (*greater than or equal*) oder **$eq** (*equal*).

Selektiert die Abfrage mehrere Dokumente, so wird der bereits vorgestellte Mechanismus der *Pagination* genutzt, das heißt das Ergebnis-JSON hat ein Attribut **hasMore**, in welchem der Client erkennen kann, ob eine weitere Abfrage sinnvoll ist oder nicht.

Auch mehrere Filter können in einer QBE-Abfrage kombiniert werden. Sollen alle starken Erdbeben gefunden werden, die zusätzlich einen Tsunami ausgelöst haben, so lautet die QBE wie folgt.

```
{"properties.mag":{"$gt": 5}, "properties.tsunami":1}
```

Wird keine besondere Angabe gemacht, so werden die beiden Filter UND-verknüpft. Ist ein logisches ODER gefragt, so muss dies explizit angegeben werden: Das Schlüsselwort **$or** enthält ein Array mit Filterkriterien im JSON-Format – eine davon muss erfüllt sein, damit der **$or**-Block erfüllt ist.

Analog dazu gibt es auch die Schlüsselworte **$and**, **$not** und **$nor**. Die Operatoren können auch geschachtelt werden, sodass sich recht komplexe QBE ergeben können.

```
{
 "$or": [
   {"properties.mag": {"$gt": 5}},
   {"properties.tsunami": 1}
 ]
}
```

Teilstringsuche ist möglich mit **$startsWith** und **$regex**. Erdbeben in **Kalifornien** könnten mit folgendem regulären Ausdruck selektiert werden.

```
{"properties.place": {"$regex": ".*California.*"}}"
```

Manchmal ist man nicht am Inhalt eines JSON-Attributs interessiert, sondern nur an der Existenz (oder Nicht-Existenz). Hierfür gibt es das Schlüsselwort **$exists**. Das folgende Beispiel selektiert alle Erdbeben, denen keine Stärke zugeordnet wurde.

```
{"properties.mag": {"not": {"$exists":1}}}
```

Neben den erwähnten, aber noch nicht vollständig aufgezählten, Operatoren zum Selektieren, gibt es auch Optionen, um die Sortierung der Ergebnismenge zu steuern. Das folgende QBE liefert alle Erdbeben in Kalifornien, absteigend sortiert

nach der Stärke. Hier wird die QBE erstmals aufgeteilt: Der *Query-Teil* enthält Angaben zur Abfrage, der *Order-By*-Teil legt die Sortierung fest.

```
{
 "$query":  {"properties.place": {"$regex":".*California.*"}},
 "$orderby": {"properties.mag": -1}
}
```

Der Funktionsumfang der QBE geht weit über die hier vorgestellten Operatoren hinaus. Eine vollständige Zusammenstellung findet sich in der Dokumentation für SODA im Abschnitt Weitere Informationen .

3.3.10 ERSTELLUNG VON INDIZES MIT REST

Die im Abschnitt 2.3 vorgestellten Indizierungstechniken für JSON werden auch von der SODA API unterstützt. Indizes nicht nur per SQL und dem **CREATE INDEX**-Kommando, sondern auch per REST-Request erstellt werden. Die für den Index nötigen Angaben werden mit dem REST-Request gesendet, SODA generiert das entsprechende **CREATE INDEX**-Kommando und führt es in der Datenbank aus. Indizes werden für SODA QBE ebenso genutzt wie für SQL-Abfragen.

Angenommen, es sollen häufig Erdbeben anhand ihrer Stärke selektiert werden, eine häufig gestellte Abfrage sei

die nach allen Erdbeben mit einer Stärke von 4.1 bis 4.2. Ohne Index müssen alle Dokumente in der Collection geparst werden. Liegt jedoch ein Index vor, so können die entsprechenden Dokumente über diesen schnell identifiziert und zurückgegeben werden.

Um also einen Index (*Function Based Index*) auf das JSON-Attribut **properties.mag** zu erzeugen, braucht es zunächst die folgende *Index-Spezifikation* im JSON-Format.

```
{
 "name":  "INTENSITAET_INDEX",
 "unique": false,
 "fields": [
  {
   "path"   : "properties.mag",
   "datatype" : "number"
  }
 ]
}
```

Diese Spezifikation wird dann im *Request-Body* eines POST-Requests an die URL der Connection mit dem Parameter **action=index** geschickt.

```
$ curl
 localhost:8080/ords/scott/soda/latest/earthquakes?action=index
 --request POST
```

```
 -H "Content-Type:application/json"
 -data '{"name": "INTENSITAET_INDEX", "unique": false … }]}'
```

Auch der in Abschnitt 2.3.2 vorgestellte *JSON Search Index* kann mit einem REST-Request erzeugt werden. Im Gegensatz zu den funktionsbasierten Indizes indiziert der JSON Search Index das gesamte Dokument. Die Index-Spezifikation ist folgerichtig sehr einfach.

```
{
 "name":  "EVERYTHING_INDEX",
 "unique": false
}
```

An dieser Stelle sei die Einführung in die REST-SODA API beendet. Es wurden bei weitem nicht alle Fähigkeiten der Collection-API und der QBE-Abfrageoperatoren vorgestellt; der interessierte Leser findet weitere Informationen in der Dokumentation (\Weitere Informationen).

3.4 SODA FÜR JAVA

Auch für Java-Programmierer gibt es eine SODA-API mit noch einem etwas größeren Umfang als die REST-API. Da Java das HTTP-Protokoll unterstützt, sind REST-Aufrufe natürlich auch von aus Java möglich. Die native Unterstützung der SODA-API in Java ist jedoch einfacher zu nutzen und

unterstützt, im Gegensatz zu REST, auch *Transaktionen* und *Cursor*. Beides kann REST wegen seiner Zustandslosigkeit nicht leisten.

SODA für Java ist Open Source und kann mitsamt Beispielen von Github heruntergeladen werden (siehe Weitere Informationen). Das folgende Beispielprogramm illustriert die Nutzung der SODA API für Java.

```
// Erzeuge Client und Verbindung zur Oracle Datenbank
Connection con = DriverManager.getConnection("jdbc:oracle:th
in:...");
OracleRDBMSClient cl = new OracleRDBMSClient();
OracleDatabase db = cl.getDatabase(conn);

// Erzeuge Collection fuer Erdbeben "Earthquakes".
OracleCollection col = db.admin().createCollection("Earthquakes");

// Erzeuge JSON-Document.
OracleDocument doc = db.createDocumentFromString(
 "{\"title\": \"Loma Prieta Earthquake\"}"
);

// Einfuegen des Dokuments in Collection liefert die Datenbank-
erzeugte Id
String id = col.insertAndGet(doc).getKey();
```

```
// Finde Dokument anhand ID. getOne() liefert genau ein Dokument
zurueck
System.out.println (
 "Inserted content:" + col.find().key(id).getOne().
getContentAsString()
);

// Finde Dokumente anhand ihrer Eigenschaften (QBE).
OracleDocument qbe = db.createDocumentFromString(
 "{\"title\" : {\"$startsWith\" : \"Loma\" }}"
);

OracleCursor c = col.find().filter(qbe).getCursor();

// Iteriere ueber alle Dokumente der Ergebnismenge
while (c.hasNext()) {
 OracleDocument resultDoc = c.next();
 // Print the document key and content.

 System.out.println ("Id:      " + resultDoc.getKey());
 System.out.println ("Inhalt:  " + resultDoc.
getContentAsString());
}
```

QBE-Syntax und Index-Spezifikationen sind identisch zur SODA-API für REST. Die die Collections letztlich auf Tabellen und Zeilen in der Oracle-Datenbank abgebildet werden, sind Collections der REST-API mit denen der Java-API. SQL/JSON erlaubt zusätzlich SQL-Abfragen auf die Daten der Collections, sodass sich auch komplexere Analysen und der volle Funktionsumfang der Datenbank anwenden lässt.

4 Fazit

Ab Version 12.1.0.2 steht in der Oracle-Datenbank eine umfassende JSON-Unterstützung bereit. Die Kombination von SQL/JSON-Funktionen mit der dokumentorientierten SODA-API erlaubt es, Anwendungen schneller und flexibler zu programmieren. Wie in einem NoSQL Document Store ist kein Schema nötig; Daten können ohne vorherige Definition gespeichert und wiedergefunden werden.

Im Gegensatz zu NoSQL-Datenbanken muss man jedoch nicht auf bewährte Konzepte relationaler Datenbanken verzichten: SQL-Anfragen und Projektionen der JSON-Daten sind weiterhin möglich; relationale Daten und JSON-Daten können koexistieren und in Abfragen sogar miteinander kombiniert werden. Alle Datenbankfunktionen wie Stored Procedures, Replikation, Partitioning, Verschlüsselung stehen auch für JSON-Dokumente zu Verfügung und müssen nicht eigens programmiert werden.

5 Weitere Informationen

- Oracle-Dokumentation: SQL/JSON: Oracle XML DB Developers' Guide 12.1: Chapter 39 – JSON in Oracle Database

  http://docs.oracle.com/database/121/ADXDB/json.htm

- Oracle-Dokumentation: Oracle REST Data Services: Oracle REST Data Services Documentation – Release 3.0

  https://docs.oracle.com/cd/E56351_01/index.html

- SODA API für Java auf Github:

  https://github.com/oracle/SODA-FOR-JAVA

- Deutschsprachiges Blog: JSON, REST und die Oracle-Datenbank:

  http://json-rest-oracledb.blogspot.com

- Englischsprachiges Blog zu JSON in der Oracle-Datenbank:

  http://blogs.oracle.com/jsondb

Die Autoren dieses Dojo sind Beda Hammerschmidt und Carsten Czarski. Beda Hammerschmidt hat als Entwickler in den Oracle Headquartes in Redwood Shores, USA wesentlichen Anteil an der Implementierung der JSON-Unterstützung in Oracle12c, Carsten Czarski ist bei der ORACLE Deutschland B.V. & Co KG als Ansprechpartner für die Entwicklercommunity tätig.

Beda Hammerschmidt ist auf Twitter unter dem Kürzel **bch_t** und Carsten Czarski unter **cczarski** zu finden.

Zugriff auf die komplette
Oracle Dojo-Bibliothek unter
http://tinyurl.com/dojoonline

ORACLE®

Herausgeber: Günther Stürner, Oracle Deutschland B.V.
Design: volkerstegmaier.de // Druck: Stober GmbH, Eggenstein