PL/SQL oder JavaScript in der Oracle-Datenbank 23ai?

Philipp Salvisberg, Grisselbav

Die erste unterstützte Sprache der Multilingualen Engine (MLE) in der Oracle Datenbank 23ai ist JavaScript. Durch weitere Sprachen in der Oracle-Datenbank können wir einerseits existierende Bibliotheken in der Datenbank nutzen und anderseits den Einstieg in die Datenbankentwicklung für jene ohne PL/SQL-Kenntnisse erleichtern. – War das nicht auch schon das Argument für Java in der Datenbank? Was ist jetzt mit JavaScript einfacher und besser als mit Java? Wie performant sind JavaScript-Module? Wann ist JavaScript eine gute Alternative zu PL/SQL und wann eher nicht?

Warum brauchen wir Code in der Datenbank?

Ich sehe hierfür folgende Gründe.

1. Wir bringen den Code zu den Daten statt die Daten zum Code. Damit können wir die Daten effizient auf dem Datenbank-Server verarbeiten und das Ergebnis in wenigen Netzwerk-Roundtrips dem Client bereitstellen. Dies benötigt weniger Ressourcen, ist dadurch kostengünstiger und vor allem performanter, als wenn wir die Daten zum Client transportieren und dort verarbeiten würden.
2. Wir nehmen die Datenbank in die Verantwortung für die Qualität der Daten, die sie speichert. Typischerweise schreiben wir Daten einmal und lesen sie häufig wieder aus. Darum sollten wir Daten korrekt speichern, so dass sich die Konsumenten beim Lesen auf die Daten verlassen können. So gesehen gehört die Logik zur Validierung der Daten in die Datenbank. Diese Logik ist oftmals umfangreicher als das, was die heutigen Datenbank-Constraints hergeben. Anders ausgedrückt: wir brauchen Code in der Datenbank als Teil einer API, um unsere Daten konsistent respektive korrekt zu halten.

Selbst wenn Ihre Datenbank-Applikationen nicht den Prinzipien von SmartDB oder PinkDB folgen sollten, bringt der gezielte Einsatz von Code in der Datenbank Vorteile. Falls Ihre Unternehmensrichtlinien die Verwendung von Code in der Datenbank kategorisch verbieten, ist es wahrscheinlich an der Zeit, diese Richtlinien zu überdenken.

PL/SQL ohne SQL

Nehmen wir an, wir benötigen eine Funktion, um einen Zeitstempel in Unixzeit umzuwandeln. Die deutsche Wikipedia-Seite definiert die Unixzeit wie folgt *[2]*:

Die Unixzeit zählt die vergangenen Sekunden seit Donnerstag, dem 1. Januar 1970, 00:00 Uhr UTC. Das Startdatum wird auch als The Epoch bezeichnet. Die Umschaltung von einer Sekunde zur nächsten ist synchron zur UTC. Schaltsekunden werden ignoriert, […].

Listing 1 zeigt, wie wir dies in PL/SQL implementieren können.

Die Funktion *to_epoch_plsql* erwartet einen „timestamp“. Besser wäre ein „timestamp with timezone“. Wir verzichten hier darauf, um das Beispiel möglichst einfach zu halten. Auch wenn die Lösung recht einfach erscheinen mag, implementieren wir eine bestehende Funktionalität neu. Wir mussten uns informieren, wie die Unixzeit genau funktioniert, welche Rolle Zeitzonen spielen, was es mit Schaltsekunden auf sich hat und haben gelernt, dass die Unixzeit tatsächlich in Millisekunden und nicht etwa Sekunden erfasst wird.

Wäre es nicht schön, wenn wir auf eine bestehende, getestete Funktionalität in der Datenbank zurückgreifen und so die Erweiterung unserer Applikation auf ein Minimum beschränken könnten? Auch wenn es keine *to_epoch-Funktion* in SQL gibt, bietet das Java Development Kit (JDK) eine solche Funktionalität an und die JDK-Version 11 ist als Embedded Oracle Java Virtual Machine (OJVM) ein Teil der Oracle- Datenbank 23.5.

Java ohne SQL

Die Oracle-Datenbank unterstützt Java Stored Procedures seit der Version 8i Release 1. Damit können wir eine *to_epoch_java*-Funktion wie in *Listing 2* zur Verfügung stellen.

Für Java müssen wir einerseits eine Klasse und andererseits eine Call-Specification erstellen. Die Call-Specification hat unter anderem die Aufgabe Ein- und Ausgabe-Datentypen zwischen SQL und Java zu mappen – so beispielsweise den Return-Wert von *java.lang.long* auf NUMBER.

Der Code enthält jetzt nicht mehr die Formel zur Umrechnung eines Zeitstempels auf die Unixzeit, ist aber verhältnismässig umfangreich. Geht das mit JavaScript einfacher?

JavaScript ohne SQL

Mit der Oracle-Datenbank 23ai können wir JavaScript-Module erstellen.

Die Implementation in *Listing 3* von *to_epoch_js* ist ähnlich zu *to_epoch_java.* Ein Modul in JavaScript und dann eine MLE-Call-Specification. Für JavaScript ist es nicht mehr notwendig die Eingabe-Datentypen vollständig zu mappen, da die Oracle-Datenbank Standardwerte für das Mapping aller Datentypen definiert. Diese können übersteuert werden, müssen aber nicht wie bei Java explizit definiert werden. Das Mapping des Ausgabe-Datentyps ist nicht möglich. Der Rückgabewert aus JavaScript muss sich in diesem Fall auf „number“ umwandeln lassen, ansonsten gibt es einen Laufzeitfehler.

Nichtsdestotrotz ist die Implementation für diesen einfachen Fall recht umfangreich. Das hat sich wohl auch Oracle gedacht und eine alternative Möglichkeit bereitgestellt *(siehe Listing 4)*.

Die Funktion *to_epoch_js2* in *Listing 4* ist gleichwertig zu *to_epoch_js*, aber deutlich knackiger als alle anderen Varianten. Allerdings ist eine inline MLE-Call-Specification nur für JavaScript-Code ohne Abhängigkeiten zu anderen Modulen anwendbar, also für Module ohne Import-Befehle.

Performance von to_epoch…

Wer schon mit Java Stored Procedures in der Datenbank gearbeitet hat, weiß, dass das Initialisieren der OJVM in einer neuen Datenbank-Session die Antwortzeit spürbar verlangsamt. Dies ist mit der MLE nicht der Fall, da hierfür ein GraalVM Native Image verwendet wird. Vereinfacht gesagt, wird nur der Speicherinhalt von einer Datei eingelesen, so ähnlich, als wenn Sie ihren Laptop aus dem Ruhezustand aufwecken. Dies ermöglicht es, ein Java-Programm innerhalb von einer Millisekunde zu starten. In der Datenbank wird das Native Image als Shared Library ($ORACLE_HOME/lib/libmle.so) eingebunden. Das heißt, die MLE stellt JavaScript zwar via Java zur Verfügung, ist allerdings komplett unabhängig von der OJVM.

In *Abbildung 1* vergleichen wir die Laufzeiten von 100.000 Funktionsaufrufen. Statt Sekunden verwenden wir eine normalisierte Zeiteinheit, was Vergleiche vereinfacht und die Ergebnisse unabhängiger vom verwendeten Hardware-Stack macht.

Alle Experimente wurden mit der Oracle Database 23.5 Free Edition auf einem System mit einem AMD Ryzen

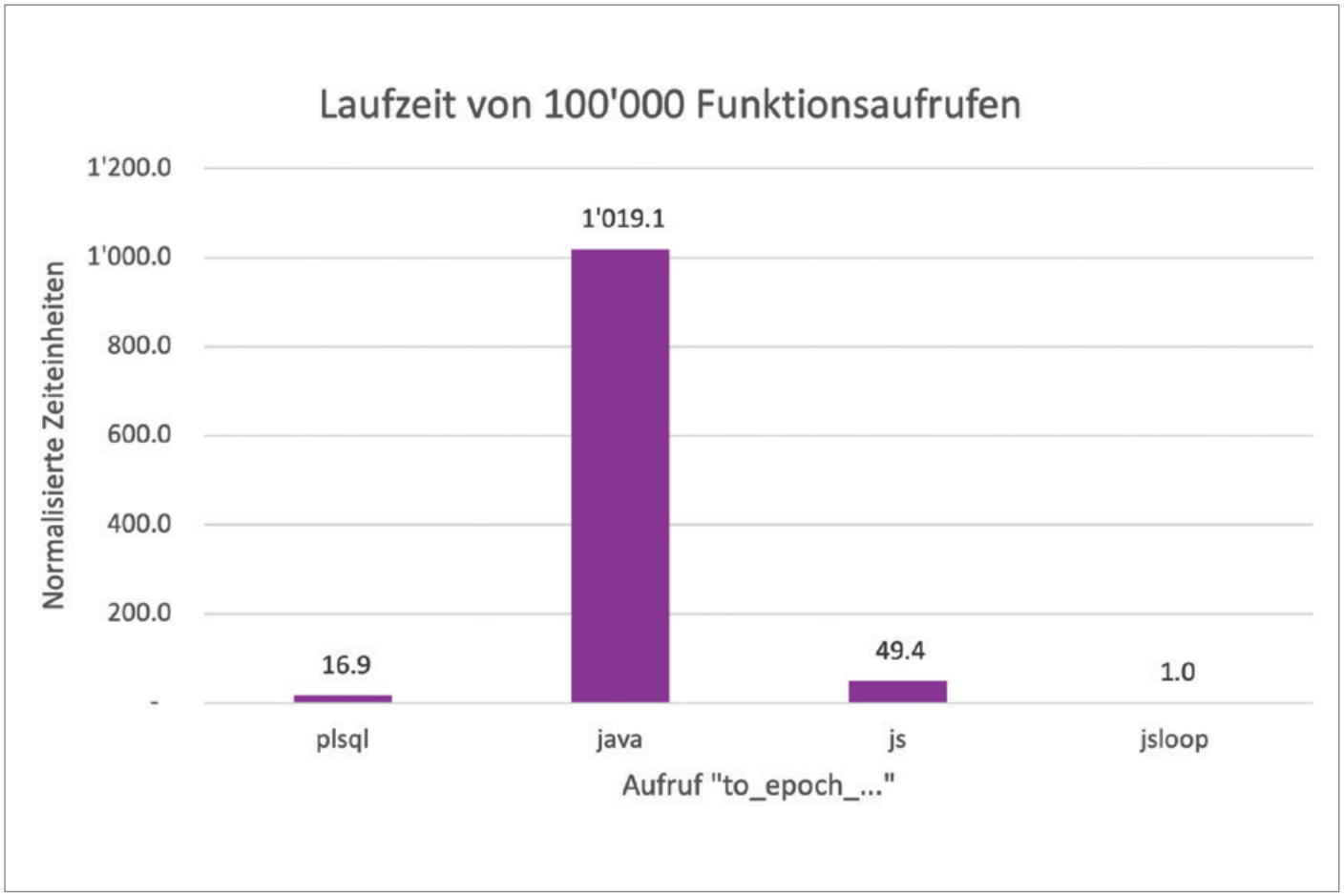

Abbildung 1: Laufzeit von 100.000 Funktionsaufrufen (Quelle: Philipp Salvisberg)

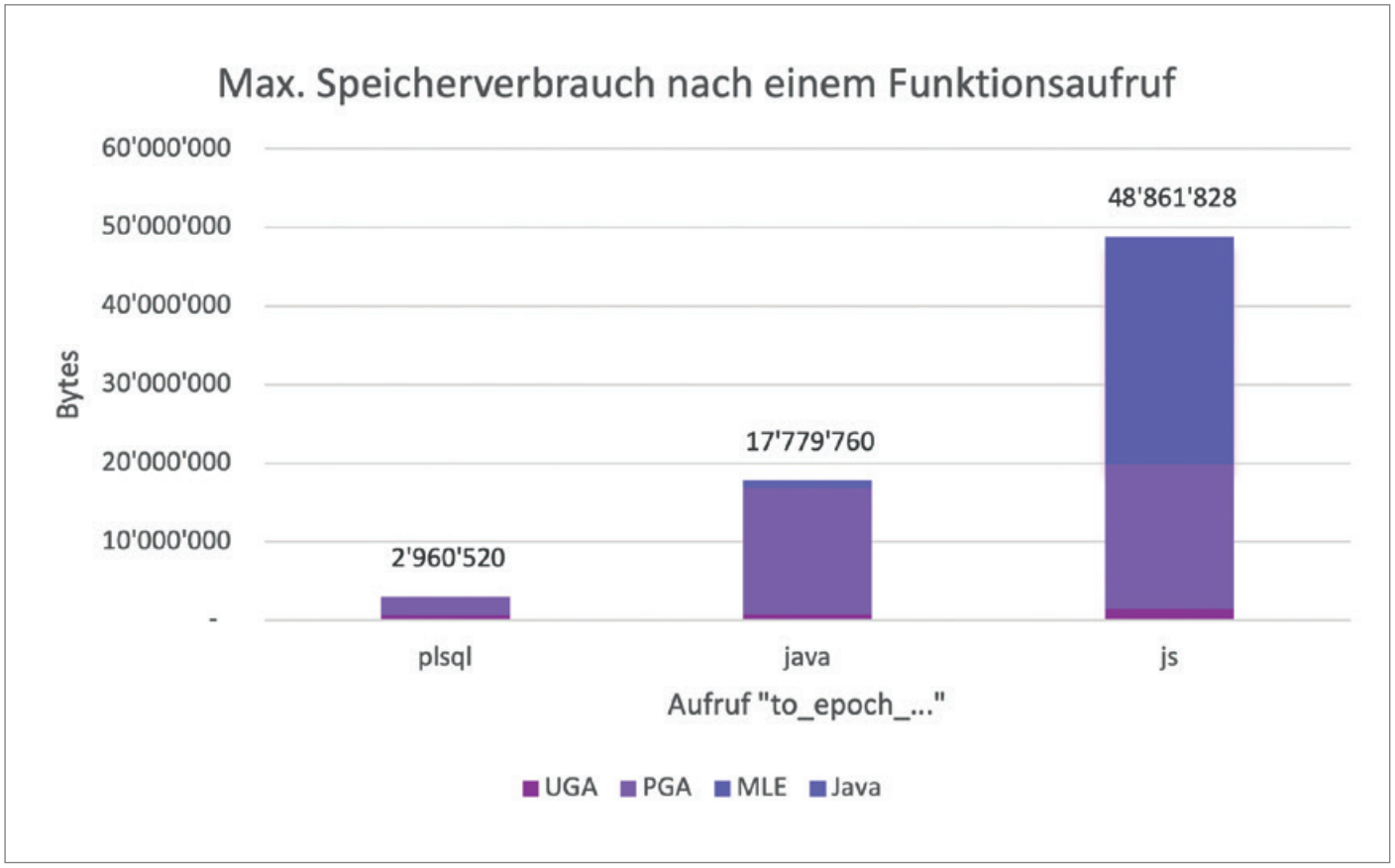

Abbildung 2: Maximaler Speicherverbrauch nach einem Funktionsaufruf (Quelle: Philipp Salvisberg)

R1600-Prozessor durchgeführt. Von fünf Wiederholungen wurde die jeweils kürzeste Zeit berücksichtigt. Sie können diese Experimente anhand der Skripte im GitHub Repository *[1]* nachvollziehen.

Bei den Varianten *to_epoch_plsql*, *to_epoch_java* und *to_epoch_js* erfolgt der Aufruf aus einer PL/SQL Loop. Dies bedeutet 100.000 Kontextwechsel zwischen PL/SQL und Java respektive JavaScript. In der vierten Variante *to_epoch_jsloop* erfolgt der Aufruf von *to_Epoch* aus einer JavaScript Loop. Der Kontextwechsel zwischen PL/SQL und JavaScript macht die Verarbeitung in diesem Fall circa um Faktor 50 langsamer.

Basierend auf diesen Ergebnissen sollten wir, wenn möglich Kontextwechsel zwischen PL/SQL und JavaScript vermeiden. Die Performance von JavaScript in der Datenbank überzeugt in diesem Fall – ganz anders als die OJVM.

Speicherverbrauch von to_epoch...

Abbildung 2 zeigt den maximal belegten Speicher am Ende eines *to_epoch*...-Funktionsaufrufs. Der Aufruf erfolgte jeweils in einer neuen Datenbank-Session und enthält auch den Speicherbedarf der Messinstrumente.

JavaScript benötigt deutlich mehr Speicher als PL/SQL. Wenn Sie dies beim Sizing des Datenbank-Servers und der Connection-Pools berücksichtigen, sollte dies heutzutage kein Problem darstellen.

JavaScript-Bibliothek verwenden

Nehmen wir an, wir möchten E-Mail-Adressen in unserer Datenbank validieren, ohne tatsächlich eine Test-E-Mail zu

```
create or replace function to_epoch_plsql(
   in_ts in timestamp
) return number is
   co_epoch_date constant timestamp with time zone :=
      timestamp '1970-01-01 00:00:00 UTC';
   l_interval    interval day(9) to second (3);
begin
   l_interval := in_ts - co_epoch_date;
   return 1000 * (extract(second from l_interval)
         + extract(minute from l_interval) * 60
         + extract(hour from l_interval) * 60 * 60
         + extract(day from l_interval) * 60 * 60 * 24);
end;
/
```

Listing 1: to_epoch_plsql

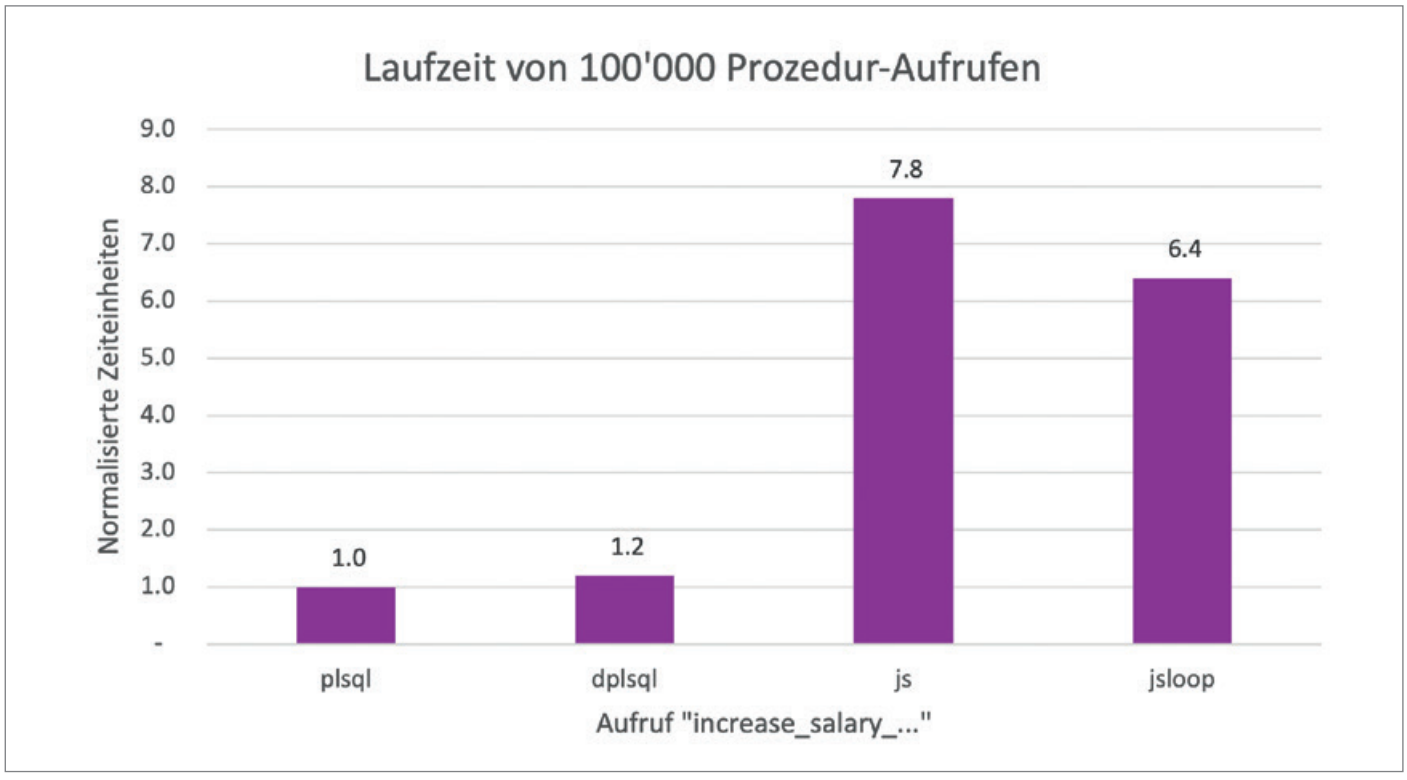

Abbildung 3: Laufzeit von 100.000 Prozedur-Aufrufen (Quelle: Philipp Salvisberg)

versenden. Die Regeln für eine gültige E-Mail-Adresse *[3]* sind relativ umfangreich. Im JavaScript-Ökosystem finden wir für solche Anforderungen Open-Source-Bibliotheken, welche sich mit Hilfe der MLE in der Datenbank unverändert einsetzen lassen. Für dieses Beispiel verwenden wir *validator.js [4]*, welche neben E-Mail-Adressen auch Kreditkarten-Nummern, EAN, IBAN und vieles mehr validieren kann. Mit Hilfe des „script"-Befehls in SQLcl können wir dieses npm-Modul direkt in die Datenbank laden *[5] (siehe Listing 5)*.

Das auszuführende Script in *Listing 5* wird nicht wie üblich vom lokalen Dateisystem gelesen, sondern via einer URL von GitHub. Das Script erstellt ein JavaScript-Modul namens VALIDATOR_MOD mit dem Inhalt der URL *https://esm.run/validator@13.12.0*, eine minimierte, für den Browser optimierte Variante des validator.js-Moduls in der Software Registry npm. Der letzte Parameter „13.12.0" bezeichnet die Version des Moduls im Oracle Data Dictionary.

In *Listing 6* erstellen wir die MLE-Call-Specification in einer PL/SQL Package. Die Funktion *is_mail* akzeptiert als Parameter nur einen String. Die Optionen für den Validator sind im Package Body definiert. Dies vereinfacht die einheitliche Verwendung in der Datenbankapplikation.

Listing 7 zeigt die Verwendung des Validators in SQL. Die zweite E-Mail-Adresse ist wegen der *allow_display_name*-Option ungültig. Die dritte E-Mail-Adresse ist formal korrekt, aber sie verwendet eine Domäne, welche unter *host_blacklist* aufgeführt ist.

JavaScript mit SQL

Die MLE stellt eine global Variable „session" vom Typ Connection zur Verfügung, um mit der aktuellen Datenbank-Session zu kommunizieren. *Listing 8* zeigt ein Beispiel eines einfachen Update-Statements mit Bind-Variablen.

PL/SQL mit SQL

Listing 9 zeigt das PL/SQL-Pendant zum JavaScript Code in *Listing 8*. Es verwendet dynamisches SQL mit Bind-Variablen.

Erfahrene PL/SQL-Entwickler würden das nicht so schreiben, da Syntax-Fehler und semantische Fehler erst zur Ausführungszeit geworfen werden. Außerdem ist es für die Oracle-Datenbank aufwändiger, dynamisches SQL auszuführen und die Verwendung der Datenbank-Objekte wird nicht im Oracle Data Dictionary abgelegt. Stattdessen verwenden erfahrene PL/SQL-Entwickler immer dann statisches SQL, wenn es möglich und sinnvoll ist. Der Code ist kürzer und verhindert SQL Injection. *Listing 10* zeigt die Variante mit statischem SQL.

Performance von increase_salary_

In *Abbildung 3* vergleichen wir die Laufzeiten von 100.000 Prozedur-Aufrufen in einer PL/SQL Loop. Einzig *increase_salary_jsloop* verwendet eine JavaScript Loop. Dadurch lassen sich 100.000 Kontextwechsel zwischen PL/SQL und JavaScript vermeiden. Das heißt, die Differenz zwischen *increase_salary_js* und i*ncrease_salary_jsloop* zeigt die Kosten von 100.000 Kontextwechseln.

In der Oracle-Datenbank Version 23.5 ist JavaScript in diesem Beispiel etwa 5- bis 6-mal langsamer als PL/SQL, wenn wir dynamisches SQL verwenden. Mit der Oracle- Datenbank-Version 23.3 lag die Differenz noch bei Faktor 7. Dies stimmt mich optimistisch, sodass wir hier mit weiteren Performance-Verbesserungen in künftigen Versionen rechnen dürfen.

Es ist schwierig, basierend auf diesen Experimenten generelle Aussagen zu den Performance-Unterschieden zwischen PL/SQL und JavaScript zu treffen. Tendenziell scheint es aber schon so zu sein, dass PL/SQL Code mit SQL-Statements gegenüber JavaScript im Vorteil ist.

MLE-Environment

Der Zugriff auf das Netzwerk ist über das JavaScript Fetch API möglich, wenn die entsprechenden Berechtigungen mit dem PL/SQL Package DBMS_NETWORK_ACL_ADMIN erteilt wurden. Allerdings ist aus Sicherheitsgründen der Zugriff auf das Dateisystem des Datenbankservers via JavaScript nicht erlaubt. In JavaScript-Laufzeitumgebungen wie Node.js, Deno, Bun oder auch Web-Browsern wird ein solcher Zugriff benötigt, um mit dem Import-Befehl andere JavaScript-Module einzubinden. Damit ein Import in der MLE funktioniert, braucht es ein MLE-Environ-

```
create or replace and compile java source named "Util" as
public class Util {
   public static long toEpoch(java.sql.Timestamp ts) {
      return ts.getTime();
   }
}
/
create or replace function to_epoch_java(in_ts in timestamp)
  return number is language java name
    'Util.toEpoch(java.sql.Timestamp) return java.lang.long';
/
```

Listing 2: to_epoch_java

```
create or replace mle module util_mod language javascript as
export function toEpoch(ts) {
   return ts.valueOf();
}
/
create or replace function to_epoch_js(in_ts in timestamp)
   return number is
      mle module util_mod
      signature 'toEpoch(Date)';
/
```

Listing 3: to_epoch_js

```
create or replace function to_epoch_js2("in_ts" in timestamp)
   return number is
      mle language javascript ' return in_ts.valueOf();';
/
```

Listing 4: to_epoch_js2 – inline MLE-Call-Specification

```
script https://raw.githubusercontent.com/PhilippSalvisberg/mle-sqlcl/main/mle.js install validator_mod
https://esm.run/validator@13.12.0 13.12.0

select version, language_name, length(module)
  from user_mle_modules
 where module_name = 'VALIDATOR_MOD';

VERSION    LANGUAGE_NAME       LENGTH(MODULE)
---------- ---------------- -----------------
13.12.0    JAVASCRIPT                 123260
```

Listing 5: validator.js von npm als validator_mod in die Datenbank laden

```
create or replace package validator_api is
   function is_email(
      in_email in varchar2
   ) return boolean deterministic;
end validator_api;
/
create or replace package body validator_api is
   function is_email_internal(
      in_email   in varchar2,
      in_options in json
   ) return boolean deterministic as mle module validator_mod
   signature 'default.isEmail(string, any)';

   function is_email(
      in_email in varchar2
   ) return boolean deterministic is
   begin
      return is_email_internal(
                in_email   => in_email,
                in_options => json('
                   {
                      "allow_display_name": false,
                      "allow_undescores": false,
                      "require_display_name": false,
                      "allow_utf8_local_part": true,
                      "require_tld": true,
                      "allow_ip_domain": false,
                      "domain_specific_validation": false,
                      "blacklisted_chars": "",
                      "ignore_max_length": false,
                      "host_blacklist": ["dubious.com"],
                      "host_whitelist": []
                   }
                ')
             );
   end is_email;
end validator_api;
/
```

Listing 6: PL/SQL Package validator_api

```
select e_mail, validator_api.is_email(e_mail) as is_valid
  from (values
          ('esther.muster@example.com'),
          ('Esther Muster <esther.muster@example.com>'),
          ('esther.muster@dubious.com')
       ) test_data (e_mail);

E_MAIL                                     IS_VALID
---------------------------------------- ----------
esther.muster@example.com                         1
Esther Muster <esther.muster@example.com>         0
esther.muster@dubious.com                         0
```

Listing 7: E-Mail-Adressen validieren.

```
create or replace mle module increase_salary_mod
language javascript as
export function increase_salary(deptno, by_percent) {
   session.execute(`
      update emp
         set sal = sal + sal * :by_percent / 100
         where deptno = :deptno`, [by_percent, deptno]);
}
/
create or replace procedure increase_salary_js(
   in_deptno     in number,
   in_by_percent in number
) as mle module increase_salary_mod
signature 'increase_salary(number, number)';
/
```

Listing 8: increase_salary_js

```
create or replace procedure increase_salary_dplsql(
   in_deptno     in number,
   in_by_percent in number
) is
begin
   execute immediate '
      update emp
         set sal = sal + sal * :by_percent / 100
      where deptno = :deptno' using in_by_percent, in_deptno;
end increase_salary_dplsql;
/
```

Listing 9: increase_salary_dplsql

```
create or replace procedure increase_salary_plsql(
   in_deptno     in number,
   in_by_percent in number
) is
begin
   update emp
      set sal = sal + sal * in_by_percent / 100
    where deptno = in_deptno;
end increase_salary_plsql;
/
```

Listing 10: increase_salary_plsql

ment. *Listing 11* zeigt wie ein MLE-Environment erstellt und verwendet wird.

Das MLE-Environment demo_env mappt den Import-Namen *increase_salary* auf das MLE-Modul *increase_salary_mod.* Im MLE-Modul *increase_salary_loop_mod* wird dieser Import verwendet, aber das MLE-Environment wird hier noch nicht zugewiesen. Dies geschieht erst in der MLE-Call-Specification *increase_salary_loop.*

MLE-Environments erlauben es den JavaScript-Code innerhalb und außerhalb der Datenbank auf gleiche Art und Weise zu strukturieren. Vermutlich wird in vielen Fällen ein MLE-Environment für eine Applikation ausreichen. Mehrere MLE-Environments braucht es, wenn unterschiedliche Sprachoptionen je Modul verwendet oder wenn mit dem gleichen Import-Namen unterschiedliche Versionen eines Moduls geladen werden sollen.

Ist das Mantra von Tom Kyte noch gültig?

Tom Kyte ist unter anderem für sein Mantra berühmt. Davon gibt es verschiede Variationen, aber immer mit der gleichen Botschaft. Diese Variante stammt aus Expert Oracle Database Architecture [6]. Auf Seite 3 schreibt er:

I have a pretty simple mantra when it comes to developing database software, one that has been consistent for many years:

- You should do it in a single SQL statement if at all possible. And believe it or not, it is almost always possible. This statement is even truer as time goes on. SQL is an extremely powerful language.
- If you can't do it in a single SQL Statement, do it in PL/SQL — as little PL/SQL as possible! Follow the saying that goes "more code = more bugs, less code = less bugs."
- If you can't do it in PL/SQL, try a Java stored procedure. The times this is necessary are extremely rare nowadays with Oracle9i and above. PL/SQL is an extremely competent, fully featured 3GL.
- If you can't do it in Java, do it in a C external procedure. This is most frequently the approach when raw speed or using a third-party API written in C is needed.

```
create or replace mle env demo_env
    imports(
        'increase_salary' module increase_salary_mod,
        'validator'       module validator_mod,
        'util'            module util_mod
    )
    language options  'js.strict=true, js.console=false,
                       js.polyglot-builtin=true'
;
create or replace mle module increase_salary_loop_mod
language javascript as
import {increase_salary} from "increase_salary";
export function increase_salary_loop(deptno,by_percent,times){
   for (let i=0; i<times; i++) {
      increase_salary(deptno, by_percent);
   }
}
/
create or replace procedure increase_salary_jsloop(
   in_deptno     in number,
   in_by_percent in number,
   in_times      in number
) as mle module increase_salary_loop_mod env demo_env
signature 'increase_salary_loop(number, number, number)';
/
```

Listing 11: Verwendung eines MLE-Environments

- If you can't do it in a C external routine, you might want to seriously think about why it is you need to do it.

Mit der Oracle-Datenbank 23ai würde ich JavaScript tendenziell auf die gleiche Stufe wie PL/SQL setzen. In jedem Fall vor Java. Außerdem geht es nicht nur darum, ob etwas in SQL oder PL/SQL machbar ist. Wenn eine benötigte Funktionalität im JavaScript-Ökosystem vorhanden ist, können wir diese jetzt in der Datenbank nutzen und sollten die Funktionalität nicht in SQL oder PL/SQL neu implementieren, nur weil es möglich ist. Es geht am Ende auch um die Wartbarkeit der Applikation und die technische Schuld, die wir uns aufladen.

Fazit

Die MLE wurde als experimentelles Feature auf der Oracle Open World 2017 vorgestellt. Seither wurde die MLE und die darunterliegende GraalVM-Technologie stetig weiterentwickelt und hat einen guten, produktionsreifen Stand in der Oracle- Datenbank 23ai erreicht. Sie ist bestens geeignet, um bestehende, getestete Funktionalität im JavaScript-Ökosystem in der Oracle-Datenbank einzubinden.

Wie genau wir JavaScript-Code mit SQL-Code am besten entwickeln, testen, debuggen und deployen, werden wir noch herausfinden müssen. In jedem Fall ist JavaScript eine echte Alternative zu PL/SQL, auch wenn PL/SQL mit statischem SQL und der besseren Performance punktet.

Quellen

[1] *https://github.com/PhilippSalvisberg/js23c*
[2] *https://de.wikipedia.org/wiki/Unixzeit*
[3] *https://de.wikipedia.org/wiki/E-Mail-Adresse*
[4] *https://www.npmjs.com/package/validator*
[5] *https://www.salvis.com/blog/2023/11/26/installing-mle-modules-in-the-oracle-database/*
[6] Tom Kyte, Expert Oracle Database Architecture, Third Edition, 2014.

Über den Autor

Philipp Salvisberg ist Inhaber der Grisselbav GmbH und ein Oracle ACE Pro. Seit 1988 fokussiert er sich auf datenbankbasierte Lösungen und unterstützt Kunden beim Design, Aufbau und der Optimierung ihrer datenbankzentrischen Lösungen. Softwarequalität und Automatisierung haben dabei einen hohen Stellenwert. Philipp hat ein Faible dafür, möglichst viel in einer einzigen SQL-Anweisung auszuführen und ist an so ziemlich allem interessiert, was hilft, die Datenbank so effizient wie möglich zu nutzen.

Philipp Salvisberg
philipp.salvisberg@grisselbav.com