Firebird auf der Suche nach dem Orakel

Fyracle

Über das RDBMS Firebird [1] wurde in diesem Magazin bereits einiges berichtet. Die finale Firebird-2.0-Version wurde auf der 4. Internationalen Firebird Konferenz im November 2006 in Prag offiziell präsentiert. Bei dieser Veranstaltung wurde den Teilnehmern auch ein Überblick über die zu erwartenden Neuerungen in Firebird 2.1 und 3.0 gegeben. Einige Dinge, die in Firebird 2.1 enthalten sein werden, sind bereits jetzt in Fyracle verfügbar, einem Produkt des eigenständigen Entwicklungszweiges der Firma Janus Software [2]. Sehen wir uns dieses ehrgeizige Projekt nun etwas genauer an, denn es ermöglicht den Betrieb des Firebird-Servers in einem Oracle-Kompatibilitätsmodus.

von Thomas Steinmaurer

Quellcodezweige

Neben dem eigentlichen Hauptentwicklungszweig von Firebird gibt es noch weitere Quellcodezweige (Branches) im Firebird-Projekt, in denen neue Funktionalitäten umgesetzt und zu einem späteren Zeitpunkt mit dem Hauptentwicklungszweig (HEAD-Branch) zusammengeführt werden. Die Entwicklung in separaten Quellcodezweigen hat den Vorteil, auch umfangreichere Features, die einen längeren Entwicklungs- und Testaufwand zur Folge haben, umsetzen zu können, ohne hierbei die Hauptentwicklung negativ zu beeinflussen. Aber nicht nur für die Entwicklung des Servers, der dann letztendlich in Form von Binärdateien zum Download zur Verfügung steht, ist dieses Konzept von großem Nutzen, sondern auch für jeden, der sich seinen eigenen Firebird-Server mit erweiterter Funktionalität erstellen will. So geschehen bei Fyracle. Dabei handelt es sich kurz gesagt um Firebird 1.5 mit Erweiterungen für die Kompatibilität zu bestimmten Oracle-SQL-Konstrukten und der Unterstützung von Oracle PL/SQL. Aber alles der Reihe nach.

Entstehungsgeschichte von Fyracle

Paul Ruizendaal, der Gründer von Janus Software, war 2003 schon seit längerer Zeit auf der Suche nach einem Datenbankprodukt für die Verwendung in Kombination mit Phoenix Object Basic, einer Visual-Basic-ähnlichen Entwicklungsumgebung, allerdings mit Cross-Plattform-Unterstützung. Es sollte etwas Vergleichbares zur Microsoft-Jet-Datenbankengine sein, aber unter Windows und Linux laufen. Des Weiteren suchte Paul bereits 2002 ein ERP-System und stieß schon bald auf ein Produkt namens Compiere [3]. Dem einen oder anderen ist dieses Produkt vermutlich bekannt. Es handelt sich um ein mittlerweile sehr beliebtes, in Java entwickeltes Open-Source-ERP-Paket, das damals allerdings die Einschränkung hatte, Oracle als DBMS-Backend vorauszusetzen. Bemühungen seitens der Compiere-Entwickler, eine Portierung auf PostgreSQL durchzuführen, schlugen fehl, da PostgreSQL zu diesem Zeitpunkt keine Sequences, keine geschachtelten Transaktionen (Savepoints) und kein Oracle-PL/SQL unterstützte.

Für beide Anwendungsszenarien schien Firebird bereits 2003 ein geeigneter Kandidat zu sein, allerdings mussten noch ein paar (harte) Nüsse geknackt werden. Firebird bot zwar eine prozedurale Sprache PSQL für serverseitige Codemodule an, dessen Syntax war aber nicht mit PL/SQL von Oracle kompatibel. Sequences, in Firebird auch besser bekannt als Generatoren, waren seit Firebird 1.0 verfügbar. Eine Implementierung von Savepoints befand sich zu diesem Zeitpunkt bereits im Entwicklungszweig von Firebird 1.5.

Bei einem Abendessen am Rande der FOSDEM-Konferenz 2003 [4] in Brüssel, waren sich Paul und das Team von IBPhoenix [5] nach umfangreichem Ideenaustausch einig, dass es möglich sein sollte, Firebird einen Oracle-Kompatibilitätsmodus zu „verpassen“, um damit auch Compiere verwenden zu können. Wohlgemerkt mit dem Ziel, so wenig wie möglich in Compiere direkt ändern zu müssen. Die Idee zu Fyracle war geboren.

Die ersten Schritte

Relativ schnell war klar, dass ein SQL Compiler entwickelt werden musste, der in einem ersten Schritt aus einem Lexer und einem rekursiven Parser bestand. Der entwickelte Compiler war rasch in der Lage, Oracle-ähnliche DDL-Anweisungen nach Firebird zu konvertieren. Er wurde dann dahingehend erweitert, auch eine Untermenge des DML-Sprachumfangs von Oracle zu unterstützen. Als erste Hürde stellte sich die Oracle-eigene Join-Syntax heraus, die den (+)-Operator für die Definition eines Outer-Joins verwendet. Der Compiler musste in der Lage sein, folgende *SELECT*-Anweisung

```
select c.cust_no, s.po_number, s.cust_no
from customer c, sales s
where c.cust_no = s.cust_no(+)
```

wie folgt zu transformieren:

```
select c.cust_no, s.po_number, s.cust_no
from customer c left outer join sales s on (s.cust_
                                             no = c.cust_no)
```

Fyracle war zu diesem Zeitpunkt noch ein textbasiertes Tool, das eine Oracle-

ähnliche SQL-Anweisung entgegennahm und daraus eine Anweisung erzeugte, die für die Ausführung an die Firebird-Engine weitergeleitet werden konnte.

Compiere-Fyracle-Verbindungsstack

Mit dem entwickelten Compiler hatte man eine entsprechende Ausgangsbasis, um über weitere Schritte für die Anbindung an das ERP-Paket Compiere nachdenken zu können. Für die Zusammenarbeit von Fyracle und Compiere war ein JDBC-Treiber notwendig, der die Anbindung übernimmt. Die Firebird-Datenbankengine ist unter anderem deswegen so beliebt, weil es für beinahe jede Zugriffsschnittstelle (ODBC, JDBC, .NET usw.) eine Implementierung gibt. So auch für JDBC, in Form des JDBC-Treibers „Jaybird", der direkt vom Firebird-Projekt kommt und ebenfalls kostenlos sowohl für kommerzielle als auch nichtkommerzielle Projekte eingesetzt werden kann. Mit der Hilfe von Roman Rokytskyy, dem Lead-Developer des Jaybird-Projekts, war die Anbindung von Compiere an Fyracle sehr rasch umgesetzt. Fyracle stellte hierfür die eigene Clientbibliothek *fyracle.dll* mit allen Firebird-Client-APIs zur Verfügung. Jaybird wurde dahingehend erweitert, dass die Kommunikation mit dem Datenbankserver als JDBC-Type-II-Treiber über die *fyracle.dll*-Bibliothek, und *fyracle.dll* selbst wiederum über die Standard-Firebird-Clientbibliothek *fbclient.dll*, erfolgte. Abbildung 1 zeigt den schematischen Verbindungsstack. An diesem Punkt war das motivierende Ergebnis wie folgt:

Listing 1

Einfache Stored Procedure p_easyblr

```
create procedure p_easyblr
as
declare variable i integer;
begin
  i = 1;
end
```

Listing 2

Generierte BLR für p_easyblr

```
blr_version5,
blr_begin,
  blr_message, 1, 1,0,
    blr_short, 0,
  blr_begin,
    blr_declare, 0,0, blr_long, 0,
    blr_assignment,
      blr_null,
      blr_variable, 0,0,
    blr_stall,
    blr_label, 0,
      blr_begin,
        blr_begin,
          blr_assignment,
            blr_literal, blr_long, 0, 1,0,0,0,
            blr_variable, 0,0,
          blr_end,
        blr_end,
    blr_end,
  blr_send, 1,
    blr_begin,
      blr_assignment,
        blr_literal, blr_short, 0, 0,0,
        blr_parameter, 1, 0,0,
      blr_end,
  blr_end,
blr_eoc
```

- Man konnte über Oracle-DDL-Anweisungen die Firebird-Datenbank erstellen.
- Man konnte sich über einen Java-Client mit der Firebird-Datenbank verbinden und Oracle-ähnliche Anweisungen absetzen.
- Oracle-ähnliche Anweisungen wurden „on-the-fly" in Firebird-äquivalente Anweisungen umgewandelt und an die Firebird-Engine weitergeleitet.
- War das Resultat eine Ergebnismenge, so wurde diese zum Java-Client zurücktransferiert.

Oracle-PL/SQL vs. Firebird-BLR

Firebird stellt für die Entwicklung von serverseitigem Code in Stored Procedures und Triggern eine prozedurale Sprache PSQL (Procedural SQL) zur Verfügung. Hierbei handelt es sich um SQL mit typischen Erweiterungen, die man von einer prozeduralen Sprache erwartet. So zum Beispiel Kontrollstrukturen, Schleifen, Cursor-Deklarationen, Exception-Handling und vieles mehr. Beim Ausführen einer *CREATE PROCEDURE* oder *CREATE TRIGGER*-DDL-Anweisung wird der PSQL-Code syntaktisch analysiert, optimiert und in eine für die Firebird-Engine verständliche Form (Binary Language Representation = BLR) übersetzt und in den Systemtabellen *RDB$PROCEDURES* bzw. *RDB$TRIGGERS* abgespeichert. Die Speicherung als BLR ist auch der Grund, warum man den Quellcode von

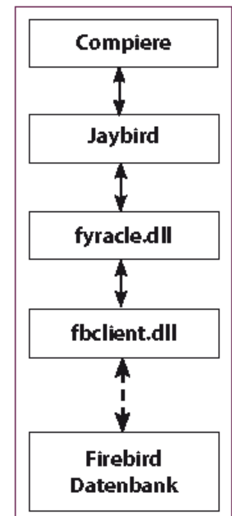

Abb. 1: Schematischer Verbindungsstack

Stored Procedures und Triggern, zwecks Schutz des eigenen geistigen Eigentums, aus den Systemtabellen löschen kann. Stored Procedures und Trigger bleiben noch funktionsfähig, da die Firebird-Engine die kompilierte BLR und nicht den eigentlichen Quellcode für die Ausführung verwendet.

Auf der Suche nach dem „Oracle" galt es mit der Unterstützung von Oracle PL/SQL eine viel härtere Nuss zu knacken. Eine erste Idee bestand darin, PL/SQL direkt auf die BLR-Verben von Firebird abzubilden. Theoretisch wäre dies vielleicht möglich, doch dieser Ansatz war praktisch nicht durchführbar, weil dies ein komplettes Reengineering der BLR nach sich gezogen hätte. Um einen Eindruck darüber zu gewinnen, was man unter BLR verstehen kann, wird für die sehr einfache und zugegeben nutzlose Stored Procedure aus Listing 1 die dafür generierte BLR in Listing 2 veranschaulicht. Es gibt nur sehr wenige Personen, die die Firebird-BLR „fließend sprechen". Für die Unterstützung von Oracle PL/SQL-Konstrukten mittels der BLR war dies aber kein gangbarer Weg.

Byte-Code-Interpreter

Paul wählte einen komplett anderen, sehr interessanten Ansatz. Da die Wurzeln der PL/SQL-Syntax in der Algol-Sprachfamilie liegen, musste ein Algol-ähnlicher Byte-Code-Interpreter für prozedurale Sprachen her, der zwar als solches eigenständig, aber trotzdem eng mit der relationalen Engine von Firebird verbunden sein musste. Die Idee zur einer Byte-Code-Engine bzw.

einer prozeduralen Engine in Form einer Virtual Machine war geboren.

Der bereits vorhandene Lexer und SQL Parser wurde entsprechend um die Unterstützung für Ausdrücke und Kontrollstrukturen erweitert. Der geparste Syntaxbaum wurde von einer weiteren Komponente verwendet, um ausführbaren Code für die Byte-Code-Engine zu erstellen. Die technische Machbarkeit wurde anhand von clientseitigen Tools überprüft. Was noch fehlte, war eine enge Bindung der prozeduralen Engine mit der relationalen Engine von Firebird, mit der Möglichkeit des gegenseitigen Aufrufs über ein API.

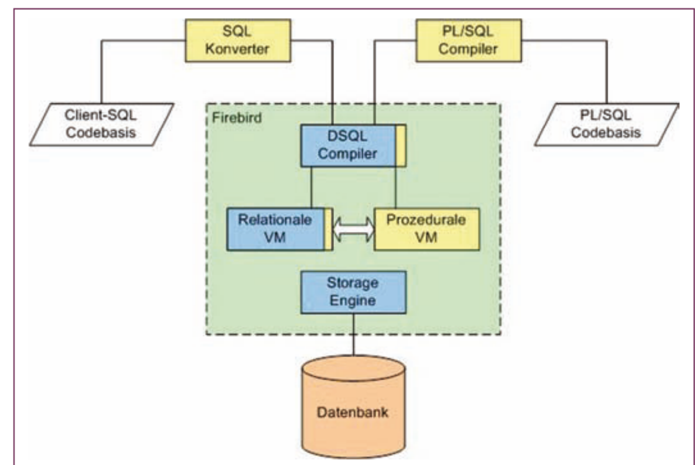

Abb. 2: Die Fyracle-Gesamtarchitektur

Im November 2003 waren die wichtigsten Komponenten fertig gestellt. Ein Java-Client konnte sich mit Fyracle verbinden und Oracle-ähnliche Anweisungen ausführen. Ein Compiler konnte den Quellcode von PL/SQL Stored Procedures, Triggern und Stored Functions in einen Byte-Code übersetzen, den die Byte-Code-Engine ausführte. Abbildung 2 zeigt die Fyracle-Gesamtarchitektur. Die gelben Blöcke veranschaulichen die Erweiterungen durch Fyracle im Vergleich zur Firebird-Codebasis. Erstmals konnte Compiere mit Fyracle als DBMS-Backend gestartet werden. Es folgte allerdings nun ein beschwerlicher Weg der Fehlerbeseitigung und der Implementierung von fehlenden Funktionen, die Oracle standardmäßig unterstützt.

Hierarchische Abfragen

Firebird unterstützt zwar hierarchische Abfragen, jedoch über Umwege. Man muss sich für eine Anforderung eine rekursive Selectable Stored Procedure entwickeln. Dies ist allerdings für den Betrieb von Compiere keine wirkliche Alternative, da das Ziel in Fyracle immer war, in Compiere so wenige Änderungen wie möglich vornehmen zu müssen. Compiere setzt hierarchische Abfragen über die Oracle-*CONNECT BY*-Syntax voraus. Der SQL-Standard sieht für hierarchische Abfragen die so genann-

Anzeige

ten Common Table Expressions (CTE) vor, die viel mächtiger sind als die *CONNECT BY*-Syntax. Die Syntax für CTEs ist in Listing 3 angeführt.

Fyracle wurde mit der Unterstützung für CTEs erweitert und verwendete diese Implementierung auch für *CONNECT BY*-Konstrukte. Somit erschlug man gleich zwei Fliegen mit einer Klappe, denn man hatte eine Implementierung für die im SQL-Standard definierten CTEs und unterstützte gleichzeitig auch SQL-Anweisungen mit der Oracle-*CONNECT BY*-Syntax. Beispielhaft können hiermit alle Unterabteilungen beliebiger Schachtelungstiefe für eine Hauptabteilung mit *DEPT_NO = '000'* auf zwei unterschiedliche Arten (Listing 4 und Listing 5) ermittelt werden.

Listing 3

Common-Table-Expressions-Syntax

```
<with statement> ::=
WITH [RECURSIVE] <common table expression>
  [, <common table expression>...] <select_statement>

<common table expression> ::=
  expression name [(column name [,...n])]
 AS
  (CTE query definition)
```

Listing 4

Hierarchische Abfrage – Oracle-CONNECT BY-Syntax

```
select dept_no, department from department
start with dept_no = '000'
connect by prior dept_no = head_dept;
```

Listing 5

Hierarchische Abfrage – Common-Table-Expression-Syntax

```
with recursive d(dept_no, department) as
 (select dept_no, department
   from department
   where dept_no = '000'
  union all
  select d2.dept_no, d2.department
   from department d2, d
   where d.dept_no = d2.head_dept)
select dept_no, department from d;
```

Listing 6

Global Temporary Tables – Syntax

```
CREATE GLOBAL TEMPORARY TABLE <table_name>
                                  <table_elements>
[ON COMMIT {PRESERVE | DELETE} ROWS]
```

Global Temporary Tables

Ein weiteres interessantes Feature in Fyracle sind so genannte Global Temporary Tables (globale temporäre Tabellen = GTT). Es handelt sich dabei um Tabellen, deren Metadaten permanent in den Systemtabellen gespeichert werden, deren Daten allerdings temporär für die Lebensdauer der Transaktion bzw. Datenbankverbindung verfügbar sind. Jede Datenbanksession kann die Definition einer GTT verwenden, um, für andere Benutzer isoliert, Daten in einer GTT einzufügen, zu löschen, zu ändern und abzufragen. Die Syntax für die Definition einer GTT ist in Listing 6 abgebildet.

Ein einfaches Beispiel soll das Verhalten in Bezug auf temporäre Daten im Kontext einer Transaktion verdeutlichen. Wir erstellen eine globale temporäre Tabelle *t1_on_delete*. Anschließend wird ein Datensatz eingefügt. Diese Einfügeoperation wird nicht mit einem *COMMIT* bestätigt. Die nachfolgende Abfrage liefert einen Datensatz zurück. Nun wird die laufende Transaktion mit einem *COMMIT* beendet. Bei einer normalen Tabelle wäre die Einfügeoperation nun persistent und für andere Transaktionen, auch im Kontext anderer Datenbankverbindungen, sichtbar. Nicht so bei unserer GTT, weil das Charakteristikum einer GTT ist, dass Daten nur temporär verfügbar sind. Da wir bei der Definition der GTT die *ON COMMIT DELETE ROWS*-Klausel verwendet haben, werden beim COMMIT der Transaktion alle sich darin befindenden Daten automatisch gelöscht. Das komplette Beispiel finden Sie in Listing 7.

Listing 7

Beispiel zu GTT mit ON COMMIT DELETE ROWS

```
create global temporary table t1_on_delete (
 id integer
)
on commit delete rows;

commit;
insert into t1_on_delete values (1);
Inserted 1 records

select * from t1_on_delete;
        ID
--------------------
         1

Selected 1 records

commit;
select * from t1_on_delete;
        ID
--------------------

Selected 0 records
```

Sollen die Daten für die Dauer der Datenbankverbindung erhalten bleiben, muss die GTT mit der *ON COMMIT PRESERVE ROWS*-Option erstellt werden. Spätestens beim Beenden einer Datenbankverbindung werden die Daten in allen GTT dann im Kontext dieser Verbindung automatisch entfernt. Für die Speicherung von temporären Daten bieten GTT gegenüber normalen Tabellen den Vorteil, dass das Entfernen von Datensätzen viel schneller erfolgen kann, da bei GTT unter anderem keine Garbage Collection notwendig ist. Für die Optimierung des Zugriffs auf GTT ist es möglich, Indizes auf Feldern in einer GTT anzulegen.

Java User-Defined-Functions und Stored Procedures

Ein sehr interessanter Punkt an Fyracle ist die Unterstützung von User-Defined-

Listing 8

Java-UDF

```
DECLARE FUNCTION currentTimeMillis
RETURNS
 BIGINT
LANGUAGE JAVA
EXTERNAL NAME 'java.lang.System.currentTimeMillis';
```

Listing 9

Java Stored Procedure

```
CREATE PROCEDURE fieldCountExtern (
 dburl VARCHAR(256),
 user_name VARCHAR(256),
 passwrd VARCHAR(256)
) RETURNS(
 V1 VARCHAR(50),
 V2 INTEGER
)
LANGUAGE JAVA
EXTERNAL NAME 'firebird.java.example.Examples.
                                   fieldNames';
```

Functions (UDF) und Stored Procedures, dessen eigentlicher Code in Java entwickelt wird. Eine detailliertere Diskussion würde den Rahmen sprengen, allerdings möchte ich zwei kurze Beispiele (Listing 8 und Listing 9) für die Deklaration einer UDF und einer Stored Procedure zeigen, dessen eigentliche Implementierung in einer Java-Klasse erfolgte.

Fyracle vs. Firebird

Firebird ist ein Open-Source-Projekt, das für jegliche Einsatzzwecke kostenlos ist. Fyracle hingegen ist ein kommerzielles Produkt, somit fallen für den Einsatz Lizenzkosten an, die allerdings sehr gering sind. Nun kann man sich die Frage stellen, ob es lizenztechnisch erlaubt ist, den Firebird-Quellcode herzunehmen und daraus mit proprietären Erweiterungen ein kommerzielles Produkt zu erstellen? Kurze Antwort: ja. Einzig und allein Änderungen, die am Firebird-Quellcode vorgenommen wurden, müssen dem Firebird-Projekt zur Verfügung gestellt werden. So können Erweiterungen und Bugfixes wieder in das offizielle Firebird-Produkt integriert werden. Davon ausgenommen sind Erweiterungen, die nicht im direkten Zusammenhang mit dem Firebird-Quellcode stehen, zum Beispiel der PL/SQL Compiler oder die Byte-Code-Engine in Fyracle.

Für beide Produkte ist es eine Win-Win-Situation. Ohne Firebird würde es Fyracle nicht geben. Auf der anderen Seite werden Erweiterungen wie Global Temporary Tables, Common Table Expressions und Java Stored Procedures/User Defined Functions früher oder später auch in einem offiziellen Build von Firebird verfügbar sein. Teile davon bereits in Firebird 2.1.

Andere Oracle-Mode-Initiativen

Fyracle erlangte die wohlverdiente Aufmerksamkeit bei der offiziellen Präsentation auf der Firebird-Konferenz 2004 in Fulda. Im selben Jahr wurde Fyracle auch auf Slashdot diskutiert [6]. Computer Associates kündigte im selben Jahr einen Millionen-Dollar-Deal für die Entwicklung von Open-Source-Lösungen basierend auf Ingres an, um die Freigabe des vormals kommerziellen DBMS als Open-Source-Produkt zu pushen. Der Hauptgewinner war eine Lösung, die den Betrieb von Ingres in einem Oracle-Kompatibilitätsmodus benutzte [7]. Was aus der Ingres-Open-Source-Bewegung geworden ist, wissen Sie vielleicht. Im selben Jahr sprang auch EnterpriseDB [8] auf diesen Zug auf und startete mit einer ähnlichen Initiative für das DBMS PostgreSQL [9]. Wie es scheint, sprechen sich gute Ideen schnell herum. 2004 waren jedenfalls einige auf der Suche nach dem Orakel, oder besser gesagt Oracle.

Ausblick

Fyracle ist eine sehr interessante Alternative zu Oracle für Oracle-basierte Anwendungen, die nicht den vollen Sprachumfang von Oracle voraussetzen. So basiert der aktuelle Fyracle PL/SQL Compiler grob gesagt auf der Syntax von Oracle 8i, allerdings mit ein paar Oracle-9i-Erweiterungen. Einige Dinge fehlen daher im PL/SQL Compiler von Fyracle noch, die aber in zukünftigen Versionen zur Verfügung stehen sollten. Auch auf der Seite der relationalen Engine stehen Punkte wie Statement-Trigger bzw. System-Level-Trigger sowie Deferred Constraints, Materialized Views und ein 128-bit Integer auf der Wunschliste der Entwickler.

Aber nicht nur zu Oracle ist Fyracle eine Alternative, sondern auch zu Firebird, falls man die in diesem Artikel diskutierten Funktionalitäten bereits jetzt benötigt. Auf der anderen Seite basiert Fyracle auf dem Quellcode von Firebird 1.5. Das bedeutet, dass viele der Neuerungen in Firebird 2.0 in Fyracle noch nicht verfügbar sind. Es bleibt auf jeden Fall interessant und spannend, wie die Entwicklungen weitergehen.

Links

www.firebirdsql.org

[2] www.janus-software.com

[3] sourceforge.net/projects/compiere

[4] www.fosdem.org

[5] www.ibphoenix.com

[6] it.slashdot.org/article.pl?sid=04/09/26/1845230

[7] www3.ca.com/Press/PressRelease.aspx?CID=69484

[8] www.enterprisedb.com

[9] www.postgresql.org

Anzeige