Effizienzsteigerung durch OO-Programmierung

Thomas Geisel
SYMAX Business Software AG
D-Wiesbaden

Schlüsselworte

PL/SQL, Objektorientierung, komplexe Abfragen, bessere Wartbarkeit, bessere Performance, Caching

Einleitung

In realen Datenbankanwendungen kommt es häufig zu Abfragen, die JOINs, UNIONs, Sub-SELECTs Netsed-SELECTs, usw. schnell eine grenzwertige Komplexität erreichen. Nicht nur das SELECT-Statement wird immer größer und damit schlechter wart- und erweiterbar, sondern auch die Gefahr, dass ein Ausführungsplan plötzlich "kippt" steigt auf ein unvorhersehbares Niveau und Abfragen, die gestern noch vertretbare Antwortzeiten hatten, dauern heute plötzlich ewig.

Durch den Einsatz der seit Oracle 8i zur Verfügung stehenden objektorientierten Programmierelemente in PL/SQL können derartige komplexe Abfrage in kleinere, wesentlich besser strukturierte und damit einfacher wartbare Teile zerlegt werden. Diese simpleren SELECT´s können gezielt optimiert werden und so i.d.R. von der Datenbank in Summe schneller und ressourcenschonender verarbeitet werden, als ein riesiges SELECT-Statement (in Anlehnung an Rechnerarchitektur spreche ich hier gerne von "CISC vs. RISC")

Zudem ist es sehr einfach, wenn man die Vorarbeit in OO geleistet hat, ein Session-bezogenes oder auch Session-übergreifendes Caching zu implementieren, was die Ausführung wiederkehrender Abfragen im den Bereich von zehntel-Sekunden beschleunigt.

Dieser Vortrag erklärt Schritt für Schritt, wie man Objekte erzeugt, mit Methoden versieht, vererbt diese anwendet, in Collections zu Listen bzw. Tabellen zusammenfasst, die Collections in SELECT-Statements nutzt und zeigt abschließend, wie mit wenig Zusatzaufwand sowohl das Session-basierte als auch Session-übergreifende Caching implementiert wird.

Der Vortrag behandelt nicht das Thema "Objektrelationale Datenbank", sondern nutzt lediglich die damit eingeführten Features, um Abfragen gegen relationale effizienter zu gestalten.

Die Problemstellung: komplexe Abfragen sind schlecht wartbar und u.U. sehr langsam

In zahlreichen Kundenprojekten, die wir in den vergangenen 2 Jahrzehnten realisiert haben, sind wir immer wieder mit seitenlangen SELECT-Statements konfrontiert worden. Entweder sollten diese optimiert werden, da inzwischen die Antwortzeiten ein nicht mehr vertretbares Niveau erreichten, oder es sollten noch weitere Fälle implementiert werden, die erneut zu einer weiteren Komplexitätssteigerung der Abfrage geführt hätte.

Dabei ist es letztlich gleich, ob die Abfrage direkt von der Anwendung so ausgeführt wird, oder ob diese schon in einer View zusammengefasst wurde: das Problem der ressourcenfressenden Ausführung und schlechten Les- und Wartbarkeit wird dabei lediglich von einer Codestelle an eine andere verlagert.

Und bei derartig komplexen Abfragen kommt es naturgemäß zu praktisch nicht mehr interpretierbaren Ausführungsplänen, womit auch die Gefahr steigt, dass diese zu unvorhersehbaren Zeitpunkten plötzlich "kippen" können. Eine gezielte Optimierung ist dann, bei solch monströsen Konstrukten, kaum noch möglich, sondern häufig wird nur im "trial-and-error" Verfahren optimiert … was dann u.U. nach wenigen Wochen plötzlich wieder nicht mehr passt.

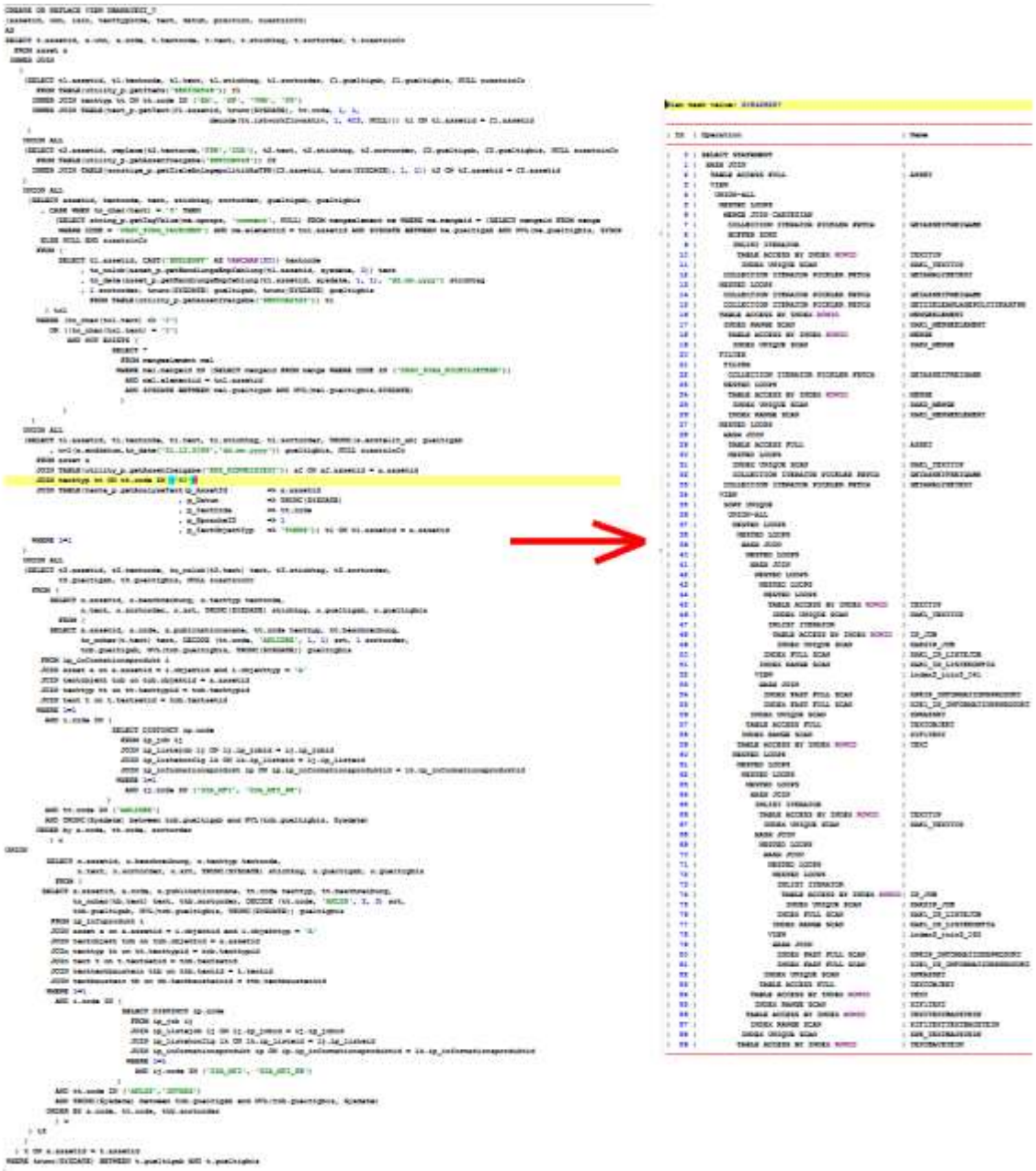

Abb. 1: Seitenlange Abfragen sind kompliziert und führen zu komplexen Ausführungsplänen

Der Lösungsansatz: komplexe Abfragen mit Hilfe der OO-Features in kleine Teile zerlegen

Anstatt ein riesiges SELECT-Statement zu formulieren, ist es u.U. sinnvoll, die einzelnen Teilaufgaben und Sub-SELECTs durch Funktionen von Objekten ausführen zu lassen.

Jeder dieser Teilschritte ist dann in einer relativ überschaubaren Funktion hinterlegt und die einzelnen Abfragen sind klar und präzise und können somit auch zielgerichtet optimiert werden.

Hierzu müssen die zuständigen Objekttypen allerdings in der Datenbank deklariert und bei der Anwendung davon Objektinstanzen angelegt werden.
Ein sehr einfacher Objekttyp wird z.B. mit folgendem Code angelegt:

```
CREATE OR REPLACE TYPE T_ASSET AS OBJECT
(
   assetid INTEGER
)
```

Damit wird der Objekttyp "T_ASSET" deklariert, welcher in diesem einfachen Fall keine Methoden bereitstellt und lediglich über das Attribut "assetid" verfügt.
("Asset" steht in diesem Beispiel für ein Finanzkonstrukt, wie z.B. eine Aktie, eine Anleihe, einen Fonds, usw.)
Verwendet wird dieser Objekttyp dann z.B. wie folgt:

```
DECLARE
   o t_asset;
BEGIN
   o := t_asset(assetid => 0);
   dbms_output.put_line( o.assetid );
END;
```

Hierbei wird die Variable o als Variable vom soeben erzeugten Objekttyp deklariert, danach wird ihr mit dem Standard-Konstruktor eine Instanz zugewiesen und dabei gleich der Wert des Attributs "assetid" belegt, um direkt danach die Ausgabe derselben vorzunehmen.

Bis hier hin wenig spektakulär und auch weitgehend identisch mit den bekannten RECORDs.
Aber wenn das Objekt neben weiteren Attributen dann auch noch um Methoden (Procedures und/oder Functions) erweitert wird, offenbart sich schon langsam, welches Potential darin steckt:

```
CREATE OR REPLACE TYPE T_asset AS OBJECT
(
   assetid INTEGER,
   isin VARCHAR2(50),
   pubname VARCHAR2(200),
   m_assettypname VARCHAR2(50),

   CONSTRUCTOR FUNCTION T_asset
      RETURN SELF AS RESULT,

   CONSTRUCTOR FUNCTION T_asset(p_fromAssetId INTEGER)
      RETURN SELF AS RESULT,

   MEMBER FUNCTION toString(SELF IN OUT t_asset)
      RETURN VARCHAR2,

   MEMBER FUNCTION getPerformance(p_betweenDate DATE,
                                  p_andDate DATE) RETURN FLOAT
)
```

Diese Objekttyp-Deklaration definiert nun neben weiteren Attributen noch
- einen parameterlosen Konstruktor, mit welchem sich gerade bei größeren Objekten eine einfachere Instanziierung durchführen lässt,
- einen Konstruktor, der die Attribute der erzeugten Objektinstanz unmittelbar aus der Tabelle anhand einer AssetID befüllt
- eine Funktion, die eine individuelle String-Repräsentation des geladenen Assets liefert
- eine Funktion, welche die Wertentwicklung des einer zugrundeliegenden Tabelle ermittelt, bzw. adhoc berechnet

Gerade die beiden letzten Funktionen können bzw. müssen unterschiedlich implementiert werden, je nachdem, um welchen Assettyp es sich handelt. So ergibt sich bei einer Aktie die Wertentwicklung einfach aus dem Kursunterschied zwischen 2 Zeitpunkten, bei einem Fonds müssen Ausschüttungen bzw. Thesaurierung berücksichtigt werden, bei einer (Index-)Benchmark müssen die Indexwerte gegenübergestellt werden, usw.

Spätestens jetzt würde eine herkömmliche View derart komplex werden und in reichlich OR-Bedingungen sowie Sub- oder Nested-Selects münden.
Bei Objekten hingegen kann das weiter alles einfach gehalten werden, in dem vom Urtyp geerbt und das (vom Standard) abweichende Verhalten individuell in überschaubaren Methoden neu implementiert wird:

```
CREATE OR REPLACE TYPE T_Benchmark UNDER T_Asset
(
   CONSTRUCTOR FUNCTION T_benchmark
      RETURN SELF AS RESULT,

   CONSTRUCTOR FUNCTION T_benchmark(p_fromAssetId INTEGER)
      RETURN SELF AS RESULT,

   MEMBER FUNCTION useCount
      RETURN INTEGER,

   OVERRIDING MEMBER FUNCTION toString(SELF IN OUT t_benchmark)
      RETURN VARCHAR2

   OVERRIDING MEMBER FUNCTION getPerformance(p_betweenDate DATE,
                                  p_andDate DATE) RETURN FLOAT
);
```

Die vorstehende Deklaration führt den Objekttyp T_Benchmark als Nachfahre von T_Asset ein, welcher zwei eigene Konstruktoren erhält, da eine Benchmark im aktuellen Beispiel anders instanziiert und aus der Datenbank geladen werden muss.

Zudem führt der Objekttyp T_Benchmark eine neue Funktion useCount ein, welche die Häufigkeit der Verwendung der jeweiligen Benchmark zurückgibt und erst auf dieser Ebene, nicht jedoch auf der Ebene von T_Asset Sinn macht.

Die letzten beiden Routinen, die bereits mit T_Asset eingeführt wurden, werden hier per Override neu definiert, was dem Entwickler die Möglichkeit gibt, diese individuell für Benchmarks auszuformulieren und den damit anderen Anforderungen Rechnung zu tragen.

Objekte per Collections und Table-Functions in normalen SELECT-Anweisungen nutzen

Mit einem einzelnen Objekt kann man zwar auch schon sehr vielseitige Dinge tun, aber für die Vereinfachung der eingangs erwähnten komplexen Abfragen, braucht man Listen von Objekten, die sich per SELECT auswerten und bei Bedarf auch mit anderen Tabellen und Views JOINen lassen.

Um dies zu erreichen werden Collections definiert, die als Listen von Objekten zu sehen sind:

```
CREATE OR REPLACE TYPE T_AssetList AS TABLE OF T_Asset;
```

Damit wird T_AssetList als eine (Speicher-)Tabelle definiert, die Objekte vom Typ T_Asset und all´ seiner Nachfahren beinhalten kann.

Um eine solche Collection zu befüllen, wird diese initialisiert und entweder per BULK-COLLECT aus den Ergebnissen einer SELECT-Abfrage beladen, oder einzeln durch das Erweitern der Liste und das Einhängen eines bereitzustellenden Objekts:

```
al := t_assetlist();
al.extend;
al(al.last) := t_asset(p_fromAssetID => 1000);
```

Das vorstehende Code-Fragment zeigt, wie eine solche Collection initialisiert wird, wie dann der Platz für ein weiteres Element geschaffen und auf diesem Platz dann ein neues Objekt eingehängt wird.

Üblicherweise erfolgt eine derartige Befüllung in einer FOR-Loop, die alle für einen bestimmten Anwendungsfall relevanten Elemente identifiziert.

Eine derartige Collection kann theoretisch auch direkt in SQL-Statements verwendet werden, allerdings ist in normalen SELECT-Anweisungen das Initialisierung und Befüllen nicht möglich.
Daher wird für solche Zwecke eine Table-Function dazwischen geschaltet.
So eine Table-Function liefert dann genau die gewünschte Collection zurück und nimmt 0 bis n Parameter an, mit welchen z.B. das innere Verhalten und das Filtern der zutreffenden Daten gesteuert wird:

```
FUNCTION getAssetList(p_AssetIDFilter VARCHAR2 DEFAULT NULL
                    , p_AssetTypFilter VARCHAR2 DEFAULT NULL  )
      RETURN T_AssetList;
```

Diese Funktion nimmt 2 optionale Parameter an, mit welchen die Datenmenge gefiltert werden kann und liefert eine befüllte T_AssetList zurück. In einer SELECT-Abfrage kann diese Table-Function dann z.B. so angewendet

```
SELECT *
   FROM TABLE(fnd_demo.getAssetList(
      p_AssetTypFilter => 'SHR,BM'));
```

und bei Bedarf auch mit anderen Tabellen, Views oder Table-Functions geJOINed werden.

Das Sahnehäubchen: ein eigenes Caching-System

Die Oracle bietet mit result_cache einen eigenen Mechanismus, um Resultate von Funktionsaufrufen bei gleichen Eingangsparametern zu puffern und bei erneutem Aufruf schnell zurückgeben zu können.

Allerdings hat man als Entwickler relativ wenig Einflussmöglichkeit auf die Internas dieser Funktionalität und häufig variiert ein Funktionsergebnis auch bei gleichen Eingangsparametern in Abhängigkeit und inneren Gesetzmäßgikeiten.

In diesem Fall hilft ein eigenes Caching-System, welches mit nur wenigen Handgriffen implementiert werden kann, wenn man bereits OO-Techniken bis hin zu Collections und Table-Functions anwendet.

Für ein Session-bezogenes Caching reicht dazu ein assoziatives Array (map) aus, welches als Nutzlast eben die Collection (hier T_AssetList) und als Schlüssel z.B. einen String verwendet.

```
TYPE t_assetlistmap_s IS TABLE OF t_assetlist INDEX BY VARCHAR2(100);
```

Der Schlüssel kann hier ein bis zu 100 Zeichen langer String sein, der aufgrund der aktuellen Zustände beliebig zusammengesetzt wird. Üblicherweise verpackt man in den Schlüssel Filterparameter, den Auswertungszeitraum, u.ä.
Zu jedem Schlüssel wird dann beim ersten Aufruf eine komplett gefüllt Assetliste hinterlegt.
Beim zweiten Aufruf kann abgefragt werden, ob es zu diesem Schlüssel bereits einen Eintrag gibt und liefert diesen im Positivfall sofort zurück … alle weitere Abfragen oder Datenaufbereitungen entfallen.

Allerdings ist der Inhalt nur innerhalb derselben Session gültig, d.h., für andere Benutzer, die exakt die selbe Abfrage tätigen, greift dieser Cache nicht und die erste Berechnung wird für diese wieder erneut ausgeführt.

Bei dem Session-übergreifenden Caching wird statt einer im Speicher vorgehaltenen map eine Tabelle genutzt, welche neben dem Schlüssel die AssetList in einer NESTED-TABLE ablegt.

```
CREATE TABLE assetlistcache (
      ckey  VARCHAR2(20),
      assetlist t_assetlist)
   NESTED TABLE assetlist STORE AS assetlistcache_coll;
```

Anstatt beim ersten Durchlauf die resultierende AssetList-Collection nur im Speicher zu puffern, wird diese nun persistent in die Tabelle geschrieben und steht damit systemweit zur Verfügung, bis die Cache-Einträge explizit gelöscht werden.

Fazit

Die Nutzung von OO-Features bedeutet zwar etwas höheren Coding-Aufwand, eröffnet aber die Möglichkeit, komplexe Abfragen deutliche kleinteiliger (und damit besser wart- und optimierbar) zu formulieren, sehr einfach Erweiterungen und individuelle Fälle einzubauen und schließlich auch, ein Caching zu implementieren, welches dramatische Geschwindigkeitsverbesserungen bei wiederkehrenden Abfragen bringt.

Kontaktadresse:

Thomas Geisel

SYMAX Business Software AG

Parkstr. 22

D-65189 Wiesbaden

Telefon: +49 (0) 611 900 3640

Fax: +49 (0) 611 900 3642

E-Mail tgeisel@symax.de

Internet: www.symax.de / www.smaxt.de