Fine Grained PL/SQL and SQL Dependency Analysis

Philipp Salvisberg
Trivadis AG
Glattbrugg (Zürich)

Schlüsselworte
PL/SQL, SQL, Dependency Analyse

Abstrakt
Oracle Data Dictionary Views wie DBA_IDENTIFIERS und DBA_DEPENDENCIES sind in vielen Fällen ausreichend, um statischen PL/SQL und SQL Code in der Datenbank zu analysieren. Was aber, wenn ausführlichere Analysen gefordert sind, wie beispielswiese die Verwendung von Tables oder Columns in PL/SQL Package Units, in SQL Statements oder in SQL Statement Klauseln? Dieser Vortrag zeigt anhand von Praxisbeispielen, wie PL/SQL und SQL Parser von Oracle und Dritten eingebunden werden können, um diese und ähnliche Fragestellungen zu beantworten.

Gründe für PL/SQL und SQL Analysen
Es gibt viele Gründe, um PL/SQL und SQL Code zu analysieren. Ein häufiger Grund ist die Auswirkungen einer Änderung abzuschätzen. Bei der Änderung oder Erweiterung eines Datenmodells oder einer PL/SQL Package Function stellt sich die Frage, welche Code Stellen betroffen und ebenfalls anzupassen sind. Nicht in allen Fällen führen Änderungen zu Kompilationsfehlern und um Fehler nicht erst zur Laufzeit zu entdecken, sind diese Analysen hilfreich. Statische Code Analysen helfen auch die Auswirkungen von Deployments und damit verbundenen Re-Kompilationen und Locking Situationen abzuschätzen. Wechseln die Verantwortlichkeiten für eine Applikation können statische Code Analysen dem neuen Verantwortlichen dabei helfen die Applikation besser zu verstehen. Zu guter Letzt eignet sich diese Form der Analyse auch, um die Einhaltung von Standards zu prüfen. Beispielsweise ob Tables nur in zentralen APIs geändert werden.

Anwendungsbereich

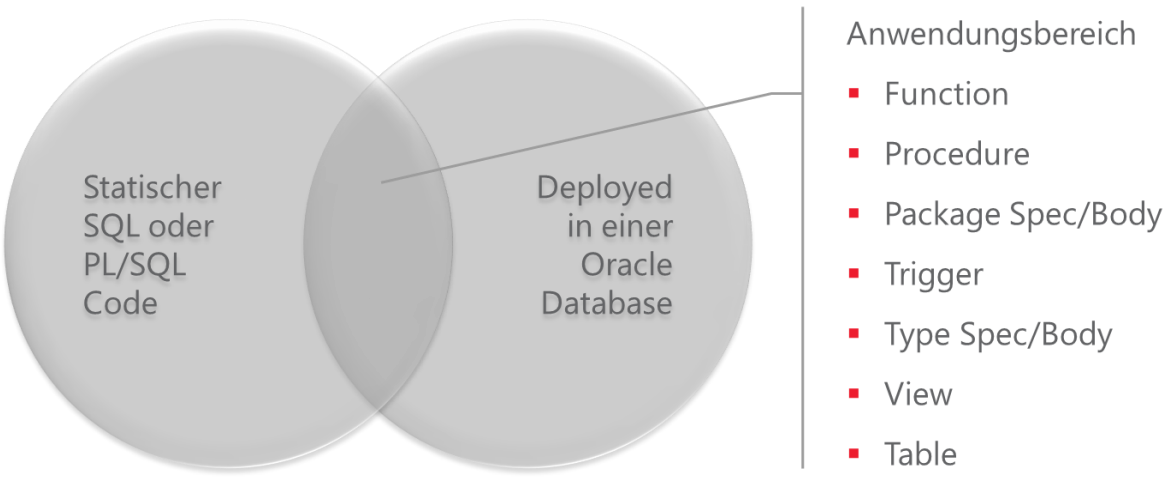

Abb. 1: Primärer Anwendungsbereich der Oracle Data Dictionary basierter Dependency Analyse

Der hier vorgestellte Ansatz eignet sich primär dazu statischen SQL und PL/SQL Code, der in einer Oracle Datenbank installiert ist zu analysieren (siehe Abb. 1). Für dynamischen SQL und PL/SQL

Code ist ein anderer resp. abgewandelter Lösungsansatz zu wählen, z.B. mit Hilfe eines Datensammlers, der ausgeführte Statements angereichert um Kontext-Informationen in Oracle Tables speichert. Die Kontext-Informationen (USER, OSUSER, MODULE, ACTION, CLIENT_INFO, etc.) werden benötigt, um eine Zuordnung zur ausführenden Applikation resp. deren Komponenten machen zu können. Sobald die zu analysierenden Statements persistiert in der Datenbank zur Verfügung stehen, können Analysen mit einem SQL und PL/SQL Parser durchgeführt werden.

Oracle Data Dictionary Views und ihre Einschränkungen
Seit vielen Releases bietet Oracle die Data Dictionary Views DBA_DEPENDENCIES, DBA_SOURCE, DBA_PROCEDURES, DBA_ARGUMENTS, DBA_TYPES, DBA_TYPE_METHODS, DBA_TRIGGERS, DBA_VIEWS, DBA_TABLES, etc. für statische Analysen an. Natürlich können statt den DBA-Views auch die ALL- respektive USER-Views verwendet werden, allerdings ist bei Analysen immer darauf zu achten, dass keine relevanten Informationen aufgrund fehlender Berechtigungen verloren gehen.

DBA_DEPENDENCIES führt Abhängigkeiten zwischen Objekten. Objekte sind Tables, Views, Types, Trigger, Packages, Procedures, Functions, etc. Sind Verwendungen auf einer tieferen Ebene zu analysieren, wie z.B. Columns oder lokale Procedures oder Functions, so ist dies mit den Oracle Data Dictionary Views nicht möglich.

Auch wenn Rob van Wijk in seinem Blog aufzeigt, wie eine DBA_DEPENDENCY_COLUMNS View basierend auf zusätzlichen Informationen in SYS.DEPENDENCY$ erstellt werden kann, um die Granularität der Analysen zu erhöhen, sind auch damit keine Analysen auf SQL Statement Ebene mit Hilfe von Oracle Data Dictionary Views möglich.

Im PL/SQL Bereich ist mit 11*g* Release 1 die View DBA_IDENTFIERS dazugekommen, welche die Analyse von PL/SQL Programmcode erlaubt, sofern auf Session- oder Systemebene der Parameter PLSCOPE_SETTINGS auf 'IDENTIFIERS:ALL' gesetzt ist und der zu analysierende PL/SQL Code nach Änderung des Parameters recompiliert worden ist. Abhängigkeiten aller PL/SQL Komponenten lassen sich mit dieser View analysieren. PL/SQL Komponenten sind z.B. Variables, Constants, Cursors, Procedures, Functions, etc. Allerdings gehören SQL Statements wie SELECT, INSERT, UPDATE, DELETE und MERGE nicht dazu. D.h. die Verwendung von bestimmten Views oder Tables in SQL Statements kann mit Hilfe von DBA_IDENTIFIERS nicht analysiert werden.

Erweiterung des Oracle Data Dictionarys
Die Oracle Data Dictionary Views sind für viele Auswertungen gut geeignet. Wenn immer möglich sollten diese Views für die Analysen verwendet werden. Informationen, die der Oracle Data Dictionary nicht hergibt, sind mit einem entsprechenden Prozess zu sammeln und in einer dedizierten Oracle Tables bereitzustellen. Diese zusätzliche Table habe ich TVD_PARSED_OBJECTS_T genannt. Nachfolgend die Struktur der Table:

```
SQL> desc tvd_parsed_objects_t

Name          Type
------------- ------------------
OBJECT_ID     NUMBER
OWNER         VARCHAR2(30 CHAR)
OBJECT_NAME   VARCHAR2(128 CHAR)
OBJECT_TYPE   VARCHAR2(30 CHAR)
LAST_DDL_TIME DATE
DDL_SOURCE    CLOB
PARSE_TREE    XMLTYPE
```

Die Columns OBJECT_ID, OWNER, OBJECT_NAME, OBJECT_TYPE und LAST_DDL_TIME haben dieselbe Semantik wie die Columns in DBA_OBJECTS. Die Column DDL_SOURCE enthält das Ergebnis des DBMS_METADATA.GET_DDL Function-Calls. Die Column PARSE_TREE enthält den Parse-Tree im XML Format als Ergebnis eines SQL & PL/SQL Parser Calls. Die nachfolgende Abbildung verdeutlicht die Ermittlung des XML Parse-Trees.

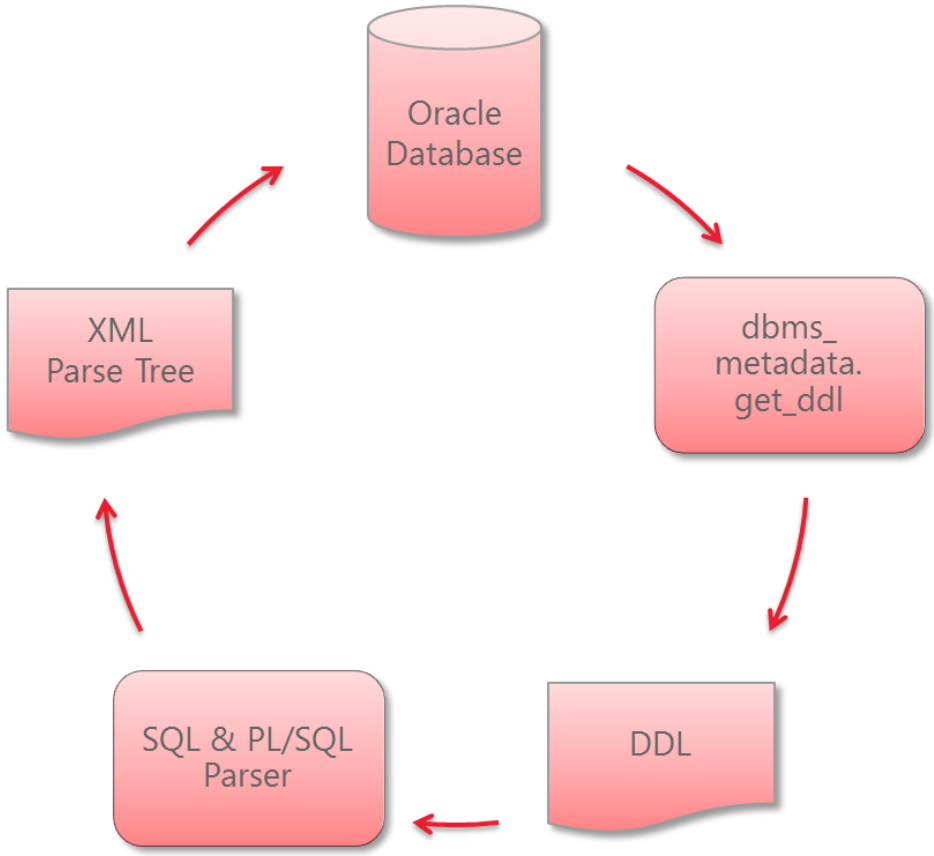

Abb. 2: Ermittlung XML Parse Tree – Refresh TVD_PARSED_OBJECTS_T

Für Java-Umgebungen sind verschiedene SQL & PL/SQL Parser Implementationen unter http://www.antlr.org/grammar/list kostenlos verfügbar und relativ einfach integrierbar. Der Ladeprozess kann je nach Bedürfnissen mit Hilfe einer PL/SQL Package und einem Web-Service oder einer eigenständigen Applikation erfolgen. Ich verwende für diesen Zweck den Trivadis SQL & PL/SQL Parser als Web-Service. Folgende zwei Signaturen kommen dabei zur Anwendung:

```
CREATE OR REPLACE PACKAGE tvd_parser_pkg IS
   FUNCTION parse_plsql(p_source_in CLOB) RETURN XMLTYPE;
END tvd_parser_pkg;
/

CREATE OR REPLACE PACKAGE tvd_parsed_objects_pkg  authid CURRENT_USER IS
   PROCEDURE refresh(p_owner_in       IN VARCHAR2 DEFAULT NULL,
                     p_object_type_in IN VARCHAR2 DEFAULT NULL,
                     p_object_name_in IN VARCHAR2 DEFAULT NULL);
END tvd_parsed_objects_pkg;
/
```

Die Procedure REFRESH kann anhand der Column LAST_DDL_TIME sehr effizient feststellen, ob ein erneuter Parse-Call notwendig ist.

Analyse #1 – SQL Statements mit Hints
Das Ziel dieser Analyse ist es, Hints in SQL Statements auszuweisen. Die Logik ist über eine View zu kapseln. Die nachfolgende Query verdeutlicht die Anwendung:

```
SQL> SELECT object_name, object_type, position, hints, usage
  2    FROM tvd_object_hint_usage_v t
  3   WHERE owner = 'TVDCC';

OBJECT_NAME        OBJECT_TYPE  POSITION HINTS    USAGE
------------------ ------------ -------- -------- ---------------
TVD_SAMPLE_PACKAGE PACKAGE BODY        1 /*+ a */ DeleteStatement
TVD_SAMPLE_PACKAGE PACKAGE BODY        2 /*+ b */ InsertStatement
TVD_SAMPLE_PACKAGE PACKAGE BODY        3 /*+ c */ QueryBlock
TVD_SAMPLE_PACKAGE PACKAGE BODY        4 /*+ d */ UpdateStatement
TVD_SAMPLE_PACKAGE PACKAGE BODY        5 /*+ e */ MergeStatement
…
```

Die hierfür notwendige View basierend auf dem Trivadis SQL & PL/SQL Parse-Tree sieht wie folgt aus:

```
CREATE OR REPLACE VIEW tvd_object_hint_usage_v AS
SELECT tobj.owner,
       tobj.object_name,
       tobj.object_type,
       tab.position,
       tab.hints,
       tab.usage
  FROM tvd_parsed_objects_t tobj,
       XMLTABLE('for $i in //hints
                 where substring($i/text(),3,1) = "+"
                 return <hints
            usage="{$i/ancestor::node()[1]/@xsi:type}">{$i/text()}</hints>'
        PASSING tobj.parse_tree
        COLUMNS "HINTS" VARCHAR2(1000) PATH 'text()',
                "USAGE" VARCHAR2(30)   PATH 'substring(@usage,7)',
                position FOR ORDINALITY) tab;
```

Die View basiert auf der relationalen Table TVD_PARSED_OBJECTS_T, so dass Filter-Prädikate auf OWNER, OBJECT_NAME und OBJECT_TYPE effizient appliziert werden können. Die HINTS und USAGE Column werden mit Hilfe einer XQuery innerhalb der XMLTABLE Function ermittelt.

Analyse #2 – Tables und Views in Queries
Das Ziel dieser Analyse ist es, alle Tables und Views auszuweisen, welche in Queries, d.h. in SELECT Statements oder in Subqueries anderer SQL Statements verwendet werden. Die Logik ist über eine View zu kapseln.Die nachfolgende Query verdeutlicht die Anwendung:

```
SQL> SELECT object_type, object_name, position AS pos, procedure_name,
  2         table_owner AS t_own, table_name
  3    FROM tvd_object_query_usage_v t
  4   WHERE owner = 'TVDCC'
  5   ORDER BY object_name, object_type;
```

```
OBJECT_TYPE  OBJECT_NAME              POS PROCEDURE_NAME   T_OWN TABLE_NAME
------------ ------------------------ --- ---------------- ----- --------------------
VIEW         TVD_OBJECT_QUERY_USAGE_V   1                        TVD_PARSED_OBJECTS_T
PACKAGE BODY TVD_PARSED_OBJECTS_PKG     1 REFRESH          SYS   DBA_OBJECTS
PACKAGE BODY TVD_PARSED_OBJECTS_PKG     2 REFRESH                TVD_PARSED_OBJECTS_T
PACKAGE BODY TVD_PARSED_OBJECTS_PKG     3 REFRESH                DBA_OBJECTS
FUNCTION     TVD_SAMPLE_FUNCTION        1 INNER_PROCEDURE  TVDCC TVD_PARSED_OBJECTS_T
…
```

Die hierfür notwendige View sieht wie folgt aus:

```
CREATE OR REPLACE VIEW tvd_object_query_usage_v AS
SELECT tobj.object_id,
       tobj.owner,
       tobj.object_type,
       tobj.object_name,
       'SELECT' AS operation,
       tab.position,
       upper(tab.procedure_name) AS procedure_name,
       upper(tab.table_owner) AS table_owner,
       upper(tab.table_name) AS table_name
  FROM tvd_parsed_objects_t tobj,
       xmltable('for $i in //queryTableExpression[@qteName]
                 let $items := $i/ancestor::items[not(@xsi:type =
                   "plsql:CursorDeclarationOrDefinition") and position()=1]
                 let $elements := $i/ancestor::elements[not(@xsi:type =
                                    "plsql:TableReference") and position()=1]
                 let $proc := if ($items) then
                                 concat($items/heading/sqlObject/@value,
                                        $items/heading/function/@value)
                              else
                                 concat($elements/function/function/@value,
                                      $elements/procedure/procedure/@value,
                                        $elements/datatype/@value)
            return <result table="{$i/@qteName}" schema="{$i/@schema}"
                     proc="{$proc}"/>'
        passing tobj.parse_tree
        columns "TABLE_NAME"  VARCHAR2(30) path '@table',
                "TABLE_OWNER" VARCHAR2(30) path '@schema',
                "PROCEDURE_NAME" VARCHAR2(30) PATH '@proc',
                position FOR ordinality) tab;
```

Das Prinzip ist dasselbe wie in der Analyse #1. Die XQuery ist in diesem Fall komplexer, da die Verwendung einer Table oder View – ausgewiesen als PROCEDURE_NAME – an unterschiedlichsten Stellen erfolgen kann. Beispielsweise ist die Verwendung in Cursors, Types sowie lokalen Functions und Procedures entsprechend zu ermitteln.

Die Implementation ist kompakt, effizient und die bereitgestellt View kann ohne spezielle Kenntnisse der XML Strukturen für Analysen genutzt werden.

Analyse #3 – Tables und Views in DML Statements
Das Ziel dieser Analyse ist es, alle Tables und Views auszuweisen, welche in DML Statements verwendet werden. Die Logik ist über eine View zu kapseln. Die nachfolgende Query verdeutlicht die Anwendung:

```
SQL> SELECT object_type, object_name, operation AS op, position AS pos, procedure_name,
  2         table_owner AS t_own, table_name
  3    FROM tvd_object_dml_usage_v t
  4   WHERE owner = 'TVDCC';

OBJECT_TYPE  OBJECT_NAME            OP     POS PROCEDURE_NAME       T_OWN TABLE_NAME
------------ ---------------------- ------ --- -------------------- ----- --------------------
PACKAGE BODY TVD_PARSED_OBJECTS_PKG INSERT   1 REFRESH              TVDCC TVD_PARSED_OBJECTS_T
FUNCTION     TVD_SAMPLE_FUNCTION    INSERT   1 INNER_PROCEDURE      TVDCC TVD_PARSED_OBJECTS_T
FUNCTION     TVD_SAMPLE_FUNCTION    INSERT   2 INNER_FUNCTION       TVDCC TVD_PARSED_OBJECTS_T
FUNCTION     TVD_SAMPLE_FUNCTION    INSERT   3                      TVDCC TVD_PARSED_OBJECTS_T
PACKAGE BODY TVD_SAMPLE_PACKAGE     INSERT   1 MOST_INNER_PROCEDURE TVDCC TVD_PARSED_OBJECTS_T
…
```

Die View zur Erzeugung dieses Resultats ist ähnlich aufgebaut wie diejenige der Analyse #2. Gelöst wurde dies hier mit vier Queries und zwar für INSERT, UPDATE, DELETE und MERGE, welche mit UNION ALL verbunden sind. Nachfolgend der Ausschnitt für den ersten Teil, welche Verwendungen in INSERT Statements identifiziert:

```
CREATE OR REPLACE VIEW TVD_OBJECT_DML_USAGE_V AS
SELECT tobj.object_id,
       tobj.owner,
       tobj.object_type,
       tobj.object_name,
       'INSERT' AS operation,
       tab.position,
       upper(tab.procedure_name) AS procedure_name,
       upper(tab.table_owner) AS table_owner,
       upper(tab.table_name) AS table_name
  FROM tvd_parsed_objects_t tobj,
       xmltable('for $i in //dmlExpressionClause[dmlName/@value
                and ancestor::statements/@xsi:type="plsql:InsertStatement"]
                 let $items := $i/ancestor::items[1]
                 let $elements := $i/ancestor::elements[1]
                 let $proc := if ($items) then
                                 concat($items/heading/sqlObject/@value,
                                        $items/heading/function/@value)
                              else
                                 concat($elements/function/function/@value,
                                      $elements/procedure/procedure/@value,
                                        $elements/datatype/@value)
             return <result table="{$i/dmlName/@value}"
schema="{$i/@schema}" proc="{$proc}"/>'
        passing tobj.parse_tree
        columns "TABLE_NAME"  VARCHAR2(30) path '@table',
                "TABLE_OWNER" VARCHAR2(30) path '@schema',
                "PROCEDURE_NAME" VARCHAR2(30) PATH '@proc',
                position FOR ordinality) tab …
```

Die Views der Analyse #2 und #3 haben dieselbe Struktur. Dadurch ist es leicht eine einzige View zu erstellen – mit Hilfe von UNION ALL – welche sämtliche Verwendungen von Tables und Views ermittelt.

Analyse #4 – Table Column in View Column Expressions
Das Ziel dieser Analyse ist es die Verwendung von schützenswerten Columns zu analysieren. Mit dieser Analyse sollen Fragen wie die folgende beantwortet werden: In welcher View wird die Column UNIT_COSTS der Table COSTS des Schemas SH angezeigt?

In den nachfolgenden zwei Views ist die direkte und indirekte Verwendung von UNIT_COSTS hervorgehoben:

```
CREATE OR REPLACE VIEW PROFITS AS
SELECT s.channel_id,
       s.cust_id,
       s.prod_id,
       s.promo_id,
       s.time_id,
       c.unit_cost,
       c.unit_price,
       s.amount_sold,
       s.quantity_sold,
       c.unit_cost * s.quantity_sold TOTAL_COST
 FROM  costs c, sales s
 WHERE c.prod_id = s.prod_id
   AND c.time_id = s.time_id
   AND c.channel_id = s.channel_id
   AND c.promo_id = s.promo_id;

CREATE OR REPLACE VIEW GROSS_MARGINS AS
WITH gm AS
 (SELECT time_id, revenue, revenue - cost AS gross_margin
    FROM (SELECT time_id,
                 unit price * quantity_sold AS revenue,
                 total_cost AS cost
            FROM profits))
SELECT t.fiscal_year,
       SUM(revenue) AS revenue,
       SUM(gross_margin) AS gross_margin,
       round(100 * SUM(gross_margin) / SUM(revenue), 2)
            AS gross_margin_percent
  FROM gm
 INNER JOIN times t ON t.time_id = gm.time_id
 GROUP BY t.fiscal_year
 ORDER BY t.fiscal_year;
```

Die nachfolgende Query zeigt, wie die Frage nach der Verwendung einer Column beantwortet werden soll:

```
SQL> SELECT schema_name, view_name, column_name FROM
  2  TABLE(tvd_coldep_pkg.get_dep('sh', 'costs', 'unit_cost'));

SCHEMA_NAME VIEW_NAME     COLUMN_NAME
----------- ------------- --------------------
SH          PROFITS       UNIT_COST
SH          PROFITS       TOTAL_COST
SH          GROSS_MARGINS GROSS_MARGIN
SH          GROSS_MARGINS GROSS_MARGIN_PERCENT
```

Die obige Query verwendet eine Pipelined Table Function. Hierfür sind die nachfolgenden DDL Statements erforderlich:

```
CREATE OR REPLACE TYPE tvd_coldep_typ FORCE IS OBJECT (
      schema_name VARCHAR2(30),
      view_name   VARCHAR2(30),
      column_name VARCHAR2(30)
);
/

CREATE OR REPLACE TYPE tvd_coldep_l FORCE IS TABLE OF tvd_coldep_typ;
/

CREATE OR REPLACE PACKAGE BODY tvd_coldep_pkg IS

   FUNCTION get_dep(p_schema_name IN VARCHAR2,
                    p_object_name IN VARCHAR2,
                    p_column_name IN VARCHAR2) RETURN tvd_coldep_l
      PIPELINED IS
   BEGIN
      -- query dictionary dependencies
      FOR v_dep IN (SELECT d.owner      AS schema_name,
                           d.name       AS view_name,
                           v.parse_tree AS parse_tree
                      FROM all_dependencies d
                     INNER JOIN tvd_parsed_objects_t v
                        ON v.owner = d.owner
                           AND v.object_name = d.name
                           AND v.object_type = d.type
                     WHERE d.referenced_owner = upper(p_schema_name)
                           AND d.referenced_name = upper(p_object_name)
                           AND d.type = 'VIEW')
      LOOP
         -- process every fetched view
         FOR v_views IN (SELECT VALUE(pv) coldep
                           FROM TABLE(process_view(v_dep.schema_name,
                                                   v_dep.view_name,
                                                   p_column_name,
                                                   v_dep.parse_tree)) pv)
         LOOP
            -- return column usages in v_dep.view_name
            PIPE ROW(v_views.coldep);
            -- get column usages of views using v_dep.view_name (recursive calls)
            FOR v_recursive IN (SELECT VALUE(dep) coldep
                                  FROM TABLE(get_dep(v_views.coldep.schema_name,
                                                     v_views.coldep.view_name,
                                                     v_views.coldep.column_name)) dep)
            LOOP
               -- return column usages of recursive call
               PIPE ROW(v_recursive.coldep);
            END LOOP;
         END LOOP;
      END LOOP;
   END get_dep;

   FUNCTION process_view(p_schema_name IN VARCHAR2,
                         p_view_name   IN VARCHAR2,
                         p_column_name IN VARCHAR2,
                         p_parse_tree  IN xmltype) RETURN tvd_coldep_l IS
      v_search_l       tvd_coldep_l := tvd_coldep_l(tvd_coldep_typ(NULL,
                                                                   NULL,
                                                                   p_column_name));
      v_previous_count INTEGER := 0;
      v_coldep_l       tvd_coldep_l := tvd_coldep_l();
   BEGIN
      -- get inline dependencies from secondary select lists
      -- TODO: handle table/view source and wildcard properly
      WHILE v_previous_count < v_search_l.count
      LOOP
```

```
         v_previous_count := v_search_l.count;
         FOR v_secondary IN (SELECT DISTINCT nvl(alias_name,
                                                 column_reference) AS alias_name
                               FROM xmltable('for    $i in //selected//*
                                              where  ($i/ancestor::fromList or
                                                      $i/ancestor::subqueryFactoringClause)
                                                     and $i/@value and not($i/self::alias)
                                              return <ret column="{$i/@value}"
                                            alias="{$i/ancestor::selected[1]//alias/@value}"/>'
                                             passing p_parse_tree columns
                                             column_reference VARCHAR2(1000) path
                                             '@column',
                                             alias_name VARCHAR2(30) path
                                             '@alias') x
                              WHERE upper(column_reference) IN
                                    (SELECT upper(column_name)
                                       FROM TABLE(v_search_l))
                                    AND upper(alias_name) NOT IN
                                    (SELECT upper(column_name)
                                            FROM TABLE(v_search_l)))
         LOOP
            -- add internal column usage
            v_search_l.extend;
            v_search_l(v_search_l.count) := tvd_coldep_typ(NULL,
                                                           NULL,
                                                           v_secondary.alias_name);
         END LOOP;
      END LOOP;
      -- analyze primary select list
      -- TODO: handle table/view source and wildcard properly
      FOR v_primary IN (SELECT DISTINCT x.column_id, atc.column_name
                          FROM xmltable('for    $i in //selected//*
                                         where  not($i/ancestor::fromList or
                                                    $i/ancestor::subqueryFactoringClause)
                                                and $i/@value and not($i/self::alias)
                                         return <ret column="{$i/@value}"
                                          id="{count($i/ancestor::selected/preceding-sibling::*)+1}"/>'
                                                passing p_parse_tree columns
                                                column_reference VARCHAR2(1000) path
                                                '@column',
                                                column_id NUMBER path '@id') x
                         INNER JOIN all_tab_columns atc
                            ON atc.owner = p_schema_name
                               AND atc.table_name = p_view_name
                               AND atc.column_id = x.column_id
                         WHERE upper(x.column_reference) IN
                               (SELECT upper(column_name)
                                  FROM TABLE(v_search_l))
                         ORDER BY x.column_id)
      LOOP
         -- add external column usage
         v_coldep_l.extend;
         v_coldep_l(v_coldep_l.count) := tvd_coldep_typ(p_schema_name,
                                                        p_view_name,
                                                        v_primary.column_name);
      END LOOP;
      -- return column dependencies
      RETURN v_coldep_l;
   END process_view;
END tvd_coldep_pkg;
/
```

Die obige PL/SQL Package ist zweifellos komplex und benötigt zur Erstellung und Wartung gute SQL, PL/SQL und XML Kenntnisse sowie ein gutes Verständnis des XML Parse-Trees.

Die Views und Pipelined Table Functions zur statischen SQL und PL/SQL Analyse sollten deshalb durch entsprechende Spezialisten bereitgestellt werden. - Die Anwendung dieser Views und Pipelined Table Functions hingegen ist einfach und bedingt nur grundlegende SQL Kenntnisse.

Fazit
Der Oracle Data Dictionary unterstützt derzeit keine Column basierte Analysen und PL/Scope ist für Analysen von SQL ungeeignet.

Erweitern Sie den Oracle Data Dictionary, um den XML Parse-Tree für alle zu analysierenden Objekte.

Abfragen basierend auf dem erweiterten Oracle Data Dictionary erlauben feingranulare, statische SQL und PL/SQL Analysen.

Vereinfachen Sie Analysen, indem Sie für komplexe Abfragen Views und Pipelined Table Functions definieren.

Kontaktadresse:
Philipp Salvisberg
Trivadis AG
Europastrasse 5
CH-8152 Glattbrugg (Zürich)

Telefon: +41-44-808 70 20
Fax: +41-44-808 70 21
E-Mail philipp.salvisberg@trivadis.com
Internet: www.trivadis.com