Optimale Performance durch Constraints im Data Warehouse

Dani Schnider
Trivadis AG
Zürich/Glattbrugg, Schweiz

Einleitung

Die Frage, ob und in welchem Umfang Datenbankconstraints in einem Data Warehouse eingesetzt werden sollen, wird in vielen Projekten kontrovers diskutiert. Obwohl Constraints ein wichtiges Hilfsmittel beim physischen Datenbankdesign sind, werden sie oft weggelassen, sei es aus Architekturüberlegungen, aus „Performancegründen" oder einfach aus Unwissenheit.[1]

A sagt: *„In unserem Data Warehouse verzichten wir auf Constraints. Sie führen zu Fehlern und Abhängigkeiten in den ETL-Prozessen und sind mühsam zu definieren".*
B meint: *„Constraints sind nicht nur wichtig zur Überprüfung der Datenintegrität und als Dokumentation des Datenmodells, sondern vor allem auch für gute Performance im Data Warehouse".*
C fragt: *„Was sind Constraints?"*

Dass Constraints nicht nur für die Datenintegrität und die Verständlichkeit des Datenmodells wichtig sind, sondern auch helfen, die Performance von ETL-Prozessen und Abfragen im Data Warehouse zu verbessern, soll hier anhand von konkreten Beispielen aufgezeigt werden. Wichtig ist dabei jedoch, dass die Constraints so angelegt werden, dass sie die Performance nicht verschlechtern, sondern eben verbessern. Doch dazu später.

Constraints im Data Warehouse

Datenbankconstraints werden hauptsächlich dazu verwendet, um die Datenintegrität in der Datenbank zu prüfen. In OLTP-Systemen mag das zweckmässig sein, aber ist es sinnvoll, dies in einem Data Warehouse zu tun, wenn die Korrektheit der Daten bereits beim Laden überprüft wird? Angenommen, all unsere Dimensionstabellen enthalten als Primärschlüssel eine Sequenznummer. Beim Laden der Fakten wird für jeden Dimensionsschlüssel ein „Key Lookup" durchgeführt. Somit ist durch den ETL-Prozess sichergestellt, dass nur gültige Dimensions-IDs ermittelt werden. Weshalb sollen wir dies also nochmals zusätzlich mittels Foreign Key Constraints überprüfen?

Bleiben wir gleich bei diesem Beispiel: Wird kein passender Dimensionseintrag gefunden, kann durch entsprechende Fehlerbehandlung im ETL-Tool oder direkt in SQL implementiert werden, dass entweder ein NULL-Wert (nicht zu empfehlen) oder ein Verweis auf einen Default-Eintrag („Singleton") eingefügt wird. Dies ist möglich, ohne dass Constraints auf der Zieltabelle notwendig sind. Trotzdem ist es wichtig, diese zusätzlichen Integritätsregeln in Form von Constraints zu definieren. Im konkreten Fall: Ein Primary Key Constraint auf jeder Dimensionstabelle, Foreign Key Constraints zwischen Fakten- und Dimensionstabellen sowie NOT NULL-Constraints für alle Dimensionsschlüssel der Faktentabelle.

[1] Bequemlichkeit könnte ein weiterer Grund sein, aber das wollen wir niemandem unterstellen.

Die Constraints erhöhen nicht nur die Lesbarkeit des Datenmodells (bzw. ermöglichen Abfragen auf den Data Dictionary, um Abhängigkeiten zwischen den Tabellen zu ermitteln), sondern stellen dem Optimizer zusätzliche Informationen zur Verfügung, die für die Ermittlung von Execution Plan für SQL-Abfragen und ETL-Prozesse verwendet werden können.

Der Oracle Query Optimizer ist in der Lage, aufgrund vorhandener Constraints eine Abfrage zu vereinfachen oder umzuschreiben. Dies ermöglicht, dass überflüssige Schritte im Execution Plan weggelassen werden und dadurch die Abfrage beschleunigt werden kann. In den hier vorgestellten Fällen ist der Performancegewinn bescheiden, doch in komplexen Abfragen auf viele Tabellen mit grossen Datenmengen können die Unterschiede in den Antwortzeiten beträchtlich sein, je nachdem ob Constraints vorhanden sind oder nicht. Weitere Beispiele dazu werden am Vortrag gezeigt.

Den Optimizer informieren: NOT NULL Constraints

Das erste Beispiel[2] zeigt das unterschiedliche Verhalten des Optimizers bei Abfragen auf mehrere Bitmap Indizes. Auf der Tabelle *CUSTOMERS* sollen alle Kunden mit Jahrgang 1965 ermittelt werden, die nicht männlich sind. Auf den Attributen *CUST_GENDER* und *CUST_YEAR_OF_BIRTH* ist je ein Bitmap Index vorhanden. Listing 1 zeigt die entsprechende Abfrage mit dem zugehörigen Execution Plan.

```
SELECT * FROM customers
WHERE cust_gender != 'M'
AND cust_year_of_birth = 1965

-------------------------------------------------------------
| Id  | Operation                     | Name                |
-------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |
|   1 |  TABLE ACCESS BY INDEX ROWID  | CUSTOMERS           |
|   2 |   BITMAP CONVERSION TO ROWIDS |                     |
|   3 |    BITMAP MINUS               |                     |
|   4 |     BITMAP MINUS              |                     |
|*  5 |      BITMAP INDEX SINGLE VALUE| CUSTOMERS_YOB_BIX   |
|*  6 |      BITMAP INDEX SINGLE VALUE| CUSTOMERS_GENDER_BIX|
|*  7 |     BITMAP INDEX SINGLE VALUE | CUSTOMERS_GENDER_BIX|
-------------------------------------------------------------

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

   5 - access("CUST_YEAR_OF_BIRTH"=1965)
   6 - access("CUST_GENDER" IS NULL)
   7 - access("CUST_GENDER"='M')
```

Listing 1: Abfrage auf mehrere Bitmap Indizes mit BITMAP MINUS

Interessant dabei ist, dass der Bitmap Index auf *CUST_GENDER* zweimal gelesen wird. Schauen wir uns diesen Execution Plan etwas genauer an:

[2] Das Verhalten des Cost-based Optimizer in dieser Situation wird detailliert in **Fehler! Verweisquelle konnte nicht gefunden werden.** anhand eines ähnlichen Beispiels erklärt.

- Zuerst werden alle Kunden mit Jahrgang 1965 ermittelt, für die das Attribut *CUST_GENDER* nicht auf NULL gesetzt ist. Dies wird mit einer *BITMAP MINUS* Operation auf die zwei Bitmap Indizes ermittelt (Zeilen 4 bis 6).
- Nun werden über einen weiteren Zugriff auf den gleichen Bitmap Index alle männlichen Kunden ermittelt (Zeile 7). Die entsprechenden Datensätze werden über eine weitere *BITMAP MINUS* Operation von der ursprünglichen Resultatmenge eliminiert (Zeile 4).

Warum so umständlich? Kein Mensch käme auf die Idee, die Abfrage über diese Zwischenschritte zu lösen. Wir wissen ja, dass im Attribut CUST_GENDER immer entweder ein M (male) oder F (female) steht, dass das Feld also nie leer ist. Der Optimizer scheint dies nicht zu wissen – und zwar aus dem einfachen Grund, weil wir es ihm nicht mitgeteilt haben.

Falls ein Attribut immer gefüllt ist – und dies sollte in einem Data Warehouse durch den Einsatz von Singletons immer gewährleitet werden können – muss dies durch einen NOT NULL Constraint entsprechend definiert werden. Mit dieser zusätzlichen Information ist der Optimizer in der Lage, einen einfacheren Execution Plan zu generieren, der mit einem *BITMAP MINUS* auskommt. Die Ermittlung der NULL-Werte entfällt, weil durch den Constraint sichergestellt ist, dass es keine geben kann. Listing 2 zeigt die gleiche Abfrage, nachdem das Attribut *CUST_GENDER* auf NOT NULL gesetzt wurde.

```
ALTER TABLE customers MODIFY (cust_gender NOT NULL);

SELECT * FROM customers
WHERE cust_gender != 'M'
AND cust_year_of_birth = 1965

-------------------------------------------------------------
| Id  | Operation                    | Name                 |
-------------------------------------------------------------
|   0 | SELECT STATEMENT             |                      |
|   1 |  TABLE ACCESS BY INDEX ROWID | CUSTOMERS            |
|   2 |   BITMAP CONVERSION TO ROWIDS|                      |
|   3 |    BITMAP MINUS              |                      |
|*  4 |     BITMAP INDEX SINGLE VALUE| CUSTOMERS_YOB_BIX    |
|*  5 |     BITMAP INDEX SINGLE VALUE| CUSTOMERS_GENDER_BIX |
-------------------------------------------------------------

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

   4 - access("CUST_YEAR_OF_BIRTH"=1965)
   5 - access("CUST_GENDER"='M')
```

Listing 2: Abfrage auf mehrere Bitmap Indizes mit BITMAP MINUS und NOT NULL Constraint

Unnötige Joins vermeiden: Join Elimination

Das zweite Beispiel zeigt eine Abfrage auf ein Star Schema. Es soll der Umsatz pro Produktkategorie für den März 2014 ermittelt werden. Listing 3 zeigt die entsprechende Abfrage und den zugehörigen Execution Plan.

```
SELECT p.prod_cat_desc,
       SUM(s.amount_sold)
  FROM sales s
  JOIN products p  ON (s.prod_id = p.prod_id)
  JOIN customers c ON (s.cust_id = c.cust_id)
  JOIN times t     ON (s.time_id = t.time_id)
 WHERE t.calendar_month_desc = '2014-03'
GROUP BY p.prod_cat_desc;

-------------------------------------------------------------------
| Id  | Operation                              | Name            |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |                 |
|   1 |  HASH GROUP BY                         |                 |
|   2 |   NESTED LOOPS                         |                 |
|*  3 |    HASH JOIN                           |                 |
|   4 |     TABLE ACCESS FULL                  | PRODUCTS        |
|   5 |     NESTED LOOPS                       |                 |
|   6 |      NESTED LOOPS                      |                 |
|*  7 |       TABLE ACCESS FULL                | TIMES           |
|   8 |       PARTITION RANGE ITERATOR         |                 |
|   9 |        BITMAP CONVERSION TO ROWIDS     |                 |
|* 10 |         BITMAP INDEX SINGLE VALUE      | SALES_TIME_BIX  |
|  11 |      TABLE ACCESS BY LOCAL INDEX ROWID | SALES           |
|* 12 |    INDEX UNIQUE SCAN                   | CUSTOMERS_PK    |
-------------------------------------------------------------------
```

Listing 3: Abfrage auf Star Schema ohne Foreign Key Constraints

Wenn wir uns die SQL-Abfrage und den Execution Plan genauer ansehen, stellen wir fest, dass unnötigerweise die Dimensionstabelle *CUSTOMERS* gelesen und dazu gejoined wird, obwohl für das Resultat gar keine Attribute daraus verwendet werden. Solche Situationen sind nicht ungewöhnlich, da die Abfragen auf ein Star Schema häufig von einem Reporting- oder OLAP-Tool generiert wird und somit teilweise überflüssige Tabellen mitselektiert werden.

Der Optimizer ist jedoch in der Lage, durch eine Transformation („Join Elimination“) solche Abfragen zu vereinfachen und unnötige Joins wegzulassen. Voraussetzung dafür ist, dass zwischen den Tabellen Foreign Key Constraints definiert sind.[3] In Listing 4 wird die gleiche Abfrage ausgeführt, jedoch mit vorhandenen Foreign Key Constraints zwischen Faktentabelle und Dimensionen. Der Execution Plan zeigt, dass der Optimizer in der Lage ist, die unnötige Tabelle *CUSTOMERS* wegzulassen und somit ein Join weniger ausgeführt werden muss.

[3] Die Constraints müssen entweder mit *ENABLE* aktiviert (aber nicht zwingend validiert) werden oder mittels *RELY* als „vertrauenswürdig“ deklariert werden. Das Verhalten unterscheidet sich jedoch in Oracle 11g und 12c, siehe [3].

```
SELECT p.prod_cat_desc,
       SUM(s.amount_sold)
  FROM sales s
  JOIN products p  ON (s.prod_id = p.prod_id)
  JOIN customers c ON (s.cust_id = c.cust_id)
  JOIN times t     ON (s.time_id = t.time_id)
 WHERE t.calendar_month_desc = '2014-03'
GROUP BY p.prod_cat_desc;

-------------------------------------------------------------------
| Id  | Operation                          | Name            |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |                 |
|   1 |  HASH GROUP BY                     |                 |
|*  2 |   HASH JOIN                        |                 |
|   3 |    TABLE ACCESS FULL               | PRODUCTS        |
|   4 |    NESTED LOOPS                    |                 |
|   5 |     NESTED LOOPS                   |                 |
|*  6 |      TABLE ACCESS FULL             | TIMES           |
|   7 |      PARTITION RANGE ITERATOR      |                 |
|   8 |       BITMAP CONVERSION TO ROWIDS  |                 |
|*  9 |        BITMAP INDEX SINGLE VALUE   | SALES_TIME_BIX  |
|  10 |     TABLE ACCESS BY LOCAL INDEX ROWID| SALES         |
-------------------------------------------------------------------

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

   2 - access("S"."PROD_ID"="P"."PROD_ID")
   6 - filter("T"."CALENDAR_MONTH_DESC"='2014-03')
   9 - access("S"."TIME_ID"="T"."TIME_ID")
```

Listing 4: Abfrage auf Star Schema mit Foreign Key Constraints

Durch die Definition der Foreign Key Constraints hat der Query Optimizer zusätzliche Informationen zur Verfügung, die es ihm ermöglichen, die nicht benötigte Tabelle in der Abfrage wegzulassen und somit einen effizienteren Ausführungsplan zu ermitteln.

Abkürzungen während den Abfragen: Query Rewrite

Zur Performanceoptimierung werden in Data Warehouses oft Materialized Views und Query Rewrite verwendet. Beim Query Rewrite überprüft der Optimizer bei jeder SQL-Abfrage, ob eine passende Materialized View zur Verfügung steht, welche die erforderlichen Informationen liefern kann. Ist dies der Fall, wird der SQL-Befehl so umgeschrieben, dass er auf die Materialized View zugreift.

In einfachen Fällen funktioniert das Query Rewrite auch ohne Constraints, beispielsweise dann, wenn die SQL-Abfrage identisch ist mit dem SELECT-Statement der Materialized View („Full Text Match“). Es gibt aber Situationen, in denen der Optimizer eine Materialized View nicht verwenden kann, weil die Beziehungen zwischen den Tabellen nicht definiert oder nicht validiert sind. Dies kann insbesondere in folgenden Situationen vorkommen:

- Query Rewrite mit Join Eliminiation: Nicht alle Tabellen der Materialized View werden in der Abfrage verwendet.
- Query Rewrite mit Join-Back: Zur Ermittlung von zusätzlichen Dimensionsattributen wird die Materialized View mit einer oder mehreren Dimensionstabelle gejoined.

Auch dazu ein Beispiel: Die Materialized View *MV_PRSUBCAT_MONTH_SALES* enthält die aggregierten Verkaufszahlen pro Produkt-Subkategorie und Monat. Wir möchten den Gesamtumsatz pro Subkategorie ermitteln, ohne Einschränkung oder Gruppierung nach Monaten. Das Query Rewrite funktioniert in diesem Fall offenbar nicht, wie die Analyse in Listing 5 zeigt:

```
VAR query VARCHAR2(1000)

EXEC :query := 'SELECT   p.prod_subcategory, '|| -
               '         sum(s.amount_sold) dollars '|| -
               'FROM     sales s, products p '|| -
               'WHERE    s.prod_id = p.prod_id '|| -
               'GROUP BY p.prod_subcategory'

EXEC dbms_mview.explain_rewrite(:query, 'MV_PRSUBCAT_MONTH_SALES')

SELECT message
FROM   rewrite_table
WHERE  mv_name = 'MV_PRSUBCAT_MONTH_SALES';

MESSAGE
--------------------------------------------------------------------------------
QSM-01150: query did not rewrite

QSM-01110: query rewrite not possible with materialized view
MV_PRSUBCAT_MONTH_SALES because it contains a join between tables
(SALES and TIMES) that is not present in the query and that
potentially eliminates rows needed by the query

QSM-01052: referential integrity constraint on table, SALES, not
VALID in ENFORCED integrity mode
```

Listing 5: Analyse einer SQL-Abfrage mit dbms_mview.explain_rewrite

Die Fehlermeldungen, die hier angezeigt werden, haben alle die gleiche Ursache: Im verwendeten Star Schema sind keine Foreign Key Constraints definiert. Dies verhindert die Verwendung von Query Rewrite. Nach dem Erstellen der fehlenden Foreign Key Constraints und der erneuten Ausführung von *dbms_mview.explain_rewrite* ist nun ein Query Rewrite möglich, wie Listing 6 zeigt.

```
ALTER TABLE sales
ADD CONSTRAINT sales_product_fk (prod_id)
REFERENCES products (PROD_ID);

ALTER TABLE sales
ADD CONSTRAINT sales_time_fk (time_id)
REFERNCES times (time_id);
```

```
DELETE FROM rewrite_table;
EXEC dbms_mview.explain_rewrite(:query, 'MV_PRSUBCAT_MONTH_SALES')

SELECT message
FROM   rewrite_table
WHERE  mv_name = 'MV_PRSUBCAT_MONTH_SALES';

MESSAGE
--------------------------------------------------------------------------------
QSM-01151: query was rewritten
QSM-01033: query rewritten with materialized view,
V_PRSUBCAT_MONTH_SALES
```

Listing 6: Erneute Analyse mit dbms_mview.explain_rewrite

Die Bremsen lösen: Reliable Constraints

Kritische Stimmen werden nun argumentieren: *„Das ist ja alles nett, aber die Performance der ETL-Prozesse wird schlechter, weil die Constraints während des Ladens geprüft werden müssen“*. Dies ist nicht der Fall, wenn diese als „Reliable Constraints“ definiert werden.

Eine übliche Vorgehensweise in Oracle Data Warehouses ist es, die Foreign Key Constraints auf *ENABLE NOVALIDATE* zu setzen. Damit sind die Fremdschlüsselbeziehungen zwischen den Tabellen dokumentiert, werden aber nicht vom Datenbanksystem überprüft. Damit die Constraints trotzdem zur Abfrageoptimierung verwendet werden können, müssen die Foreign Keys und die zugehörigen Primary Key Constraints auf *RELY* gesetzt werden, wie in Listing 7 anhand eines Beispiels einer Kopf- und Versionstabelle im Core aufgezeigt wird.

```
-- Primary Key Constraint auf DWH_HEAD_ID der Kopftabelle
ALTER TABLE cor_product
ADD CONSTRAINT cor_product_pk
PRIMARY KEY (dwh_head_id) RELY;

-- Zusätzlicher Unique Constraint auf Business Keys
ALTER TABLE cor_product
ADD CONSTRAINT cor_product_uk
UNIQUE (supplier_bk, product_bk);

-- Primary Key Constraint auf DWH_VERS_ID der Versionstabelle
ALTER TABLE cor_product_vers
ADD CONSTRAINT cor_product_vers_pk
PRIMARY KEY (dwh_vers_id);

-- Foreign Key Constraint von Versions- auf Kopftabelle
ALTER TABLE cor_product_vers
ADD CONSTRAINT cor_prodv_head_fk
FOREIGN KEY (dwh_head_id) REFERENCES cor_product (dwh_head_id)
RELY ENABLE NOVALIDATE;
```

Listing 7: Definition von Constraints auf Core-Tabellen

Jeweils vor Ausführung der ETL-Prozesse werden nun alle Foreign Key Constraints der Zieltabellen ausgeschaltet. Damit werden unnötige Prüfungen der Constraints während des Ladevorgangs vermieden. Vor allem bei der Verwendung von künstlichen Schlüsseln und Lookup-Operatoren in den ETL-Prozessen ist es überflüssig, einen soeben ermittelten Surrogate Key beim Einfügen in die Datenbank nochmals zu überprüfen. Durch das Ausschalten der Fremdschlüssel können die Ladezeiten verringert werden. Dach Beendigung des Ladevorgangs werden die Constraints wieder eingeschaltet, jedoch nicht überprüft. Listing 8 zeigt ein entsprechendes Beispiel.

```
-- Foreign Key Constraint ausschalten
ALTER TABLE cor_product_vers
MODIFY CONSTRAINT cor_prodv_head_fk
DISABLE NOVALIDATE;

-- ETL-Prozess für COR_PRODUCT_VERS ausführen
BEGIN
   Load_COR_PRODUCT_VERS;
END;

-- Foreign Key Constraint ausschalten
ALTER TABLE cor_product_vers
MODIFY CONSTRAINT cor_prodv_head_fk
ENABLE NOVALIDATE;
```

Listing 8: Aus- und Einschalten von Foreign Key Constraints während ETL-Prozess

Das Einschalten der Constraints ist nicht zwingend notwendig. Die gezeigten Optimierungsmöglichkeiten (Join Elimination, Query Rewrite, Join Back) funktionieren auch, wenn die Foreign Key Constraints auf *DISABLE* gesetzt sind. Wichtig ist jedoch, dass die Constraints auf *RELY* gesetzt sind. Dadurch wird dem Query Optimizer mitgeteilt, dass er sich auf die Constraint-Definitionen verlassen kann, obwohl die Datenintegritiät nicht von der Datenbank geprüft wurde.

Ein weiterer Vorteil von Reliable Constraints besteht darin, dass Abhängigkeiten in der Ladereihenfolge vermieden werden können. So werden zum Beispiel in Data Vault 2.0 Hash-Keys anstelle von Sequenznummern verwendet, um die verschiedenen Arten von Tabellen (Hubs, Links und Satellites) ohne Key Lookups und somit unabhängig voneinander laden zu können. Mit eingeschalteten Foreign Key Constraints ist dies nicht möglich. Werden hingegen die Constraints mit *RELY DISABLE NOVALIDATE* angelegt, bestehen zum Ladezeitpunkt keine Abhängigkeiten zwischen den Tabellen. Trotzdem kann der Query Optimizer die Constraints zur Optimierung von Abfragen und nachfolgenden ETL-Prozessen verwenden. Dadurch und durch die Unabhängigkeit der einzelnen ETL-Prozesse können die Ladezeiten im Data Warehouse zusätzlich verkürzt werden. Constraints sind also keine „Bremse“ beim Laden, sondern eine Hilfe bei der Performanceoptimierung – sowohl beim Laden ins Data Warehouse als auch bei Abfragen auf die Data Marts.[4]

Wo sollen welche Constraints erstellt werden?

Der Einsatz der verschiedenen Constrainttypen ist je nach DWH-Schicht, Modellierungsmethode und gewählter Architektur unterschiedlich. Weitere Informationen dazu sind in [4] zu finden.

[4] Weitere Informationen zur Definition und Verwendung von Reliable Constraints sind im Blog Post [2] beschrieben.

- In der Staging Area werden üblicherweise keine Constraints eingesetzt.
- Primary Key Constraints werden in der Cleansing Area, im Core und in den Data Marts erstellt. Auf den Faktentabellen wird üblicherweise auf einen Primary Key Constraint verzichtet .
- Im Core werden zusätzlich Unique Constraints auf die fachlichen Schlüssel erstellt.
- Foreign Key Constraints werden teilweise in der Cleansing Area, sicher aber im Core und in den Data Marts erstellt. Je nach gewählter Architektur und Design der ETL-Prozesse können diese als Reliable Constraints definiert werden.
- NOT NULL Constraints werden außer in der Staging Area immer angelegt. Insbesondere Fremdschlüsselattribute (z.B. Dimensionsschlüssel in Faktentabellen) sollten als NOT NULL definiert werden, um Outer Joins bei Abfragen zu vermeiden.
- Check Constraints dienen zur Überprüfung der eingegebenen Werte und werden in Data Warehouses selten eingesetzt. Diese Art von Konsistenzprüfungen werden durch die ETL-Prozesse oder Qualitäts-Checks in der Cleansing Area durchgeführt.

Fazit

Constraints werden im Data Warehouse nicht primär für ihren ursprünglichen Zweck – nämlich die Sicherstellung der Datenintegrität – eingesetzt, sondern hauptsächlich zur Dokumentation der Abhängigkeiten im Datenmodell sowie zur Performanceoptimierung. Durch die Definition von Constraints werden dem Query Optimizer zusätzliche Informationen zur Verfügung gestellt, die zur Optimierung von SQL-Befehlen für ETL-Prozesse und Abfragen verwendet werden können. Durch die Verwendung von Reliable Constraints besteht zusätzlich die Möglichkeit, Foreign Key Constraints so anzulegen, dass die Ladeprozesse nicht beeinträchtigt werden.

Quellen und weitere Informationen

[1] Jonathan Lewis: Cost-Based Oracle Fundamentals, Seiten 195 – 198
Apress, 2006, ISBN 978-1590596364

[2] Foreign Key Constraints in an Oracle Data Warehouse
https://danischnider.wordpress.com/2015/12/01/foreign-key-constraints-in-an-oracle-data-warehouse/

[3] Join Elimination: Difference between Oracle 11g and 12c
https://danischnider.wordpress.com/2015/06/29/join-elimination-difference-in-oracle-11g-and-12c/

[4] D. Schnider, C. Jordan, P. Welker, J. Wehner: Data Warehouse Blueprints
Hanser Verlag, 2016, ISBN 978-3446450752

Kontaktadresse:

Dani Schnider
Trivadis AG
Sägereistrasse 29
CH-8152 Glattbrugg

Telefon: +41 58 459 50 81
Fax: +41 58 459 56 95
E-Mail dani.schnider@trivadis.com
Internet: www.trivadis.com