Performante Abfragen in OLAP-Szenarien mit Aggregatstabellen

Aggregatstabellen mit Firebird

Welche Stückzahl eines bestimmten Produkts wurde im Quartal 1/2008 in Europa verkauft? Welchen Umsatz hatte mein Tochterunternehmen auf den Fiji-Inseln im Jahr 2007? Dies sind typische unternehmenskritische Fragen, die das Management von der Controlling-Abteilung einfordern wird. Der für solche Fragestellungen relevante Datenbestand kann sich schon mal im Bereich von über 100 Millionen Datensätzen bewegen, vor allem wenn es sich um ein größeres Unternehmen handelt.

von Thomas Steinmaurer

Die in der Einleitung formulierten Abfragen an ein DBMS-gestütztes System sind typisch für OLAP- (Online-Analytical-Processing-)Szenarien, deren technische Implementierung in der Regel ein Data Warehouse (DWH) [1] darstellt. In solch einem System spielen Aggregate eine sehr wichtige Rolle, da bei einer OLAP-Abfrage eine aggregierte Sicht auf einen existierenden Datenbestand eine wichtige Eigenschaft verkörpert. Aus SQL sind vermutlich die folgenden Aggregatsfunktionen bekannt: *COUNT*, *SUM*, *AVG*, *MIN* und *MAX*.

Wie man aggregierte Abfragen mithilfe von voraggregierten Datenbeständen um ein Vielfaches beschleunigen kann, welche Konzepte es für die Voraggregierung in Form von Aggregatstabellen gibt, und dass es nicht immer um Verkaufszahlen gehen muss, wenn von einem DWH die Rede ist – dies alles wird der vorliegende Artikel zeigen. Verwendet werden dazu das Open Source DBMS Firebird [2] und Mondrian [3], ein Open Source OLAP Server. Mondrian besitzt keine eigene (multidimensionale) Storage-Engine, sondern folgt dem relationalen OLAP- (ROLAP-)Ansatz und

Quellcode

Der Quellcode zum Artikel befindet sich auf der CD und auf www.entwickler-magazin.de.

greift auf eine existierende relationale Datenbank zu.

DWH im industriellen Umfeld

Der am meisten verbreitete Anwendungsbereich für ein Data Warehouse ist wohl noch immer der Bereich der Verkaufszahlen. Es gibt jedoch auch andere Einsatzbereiche – Paradebeispiel für eine „etwas andere" Domäne ist die industrielle Fertigung. Die Fülle an anfallenden Prozess- und Steuerdaten muss in einem DWH integriert und für Analysen aufbereitet und zugänglich gemacht werden. Wichtige Ziele für den Einsatz eines DWH in diesem Anwendungsbereich sind:

- die Verbesserung des Fertigungsprozesses, um den Ausschuss von fehlerhaft produzierten Teilen zu minimieren bzw. fehlerhafte Teile nicht an den Kunden auszuliefern
- Trendanalysen für die Lebensdauer von elektrischen Geräten, die sich bereits beim Kunden im Einsatz befinden

In beiden Fällen müssen die benötigten Daten in einem DWH gesammelt und für die darauf auszuführenden Datenanalysen entsprechend aufbereitet werden. Ein Beispiel ist die Aufgabe: „Ermittle den Verlauf der an einem Gerät gemessenen durchschnittlichen Temperatur für das Jahr 2008, aggregiert nach dem Quartal". Bedeutet die daraus gewonnene Erkenntnis, dass der Messwert über das Jahr immer nahe an der maximalen Betriebstemperatur lag, so wird sich dies nicht förderlich auf die Lebensdauer des Geräts auswirken.

Anybody in the DWH?

Die Aufgabenstellung im letzten Absatz deutet bereits auf unterschiedliche Formen an Daten hin, die für diese Analyse relevant sind:

- ein eindeutig identifizierbares Gerät
- der Messwerttyp (z.B. Temperatur, Strom, Spannung)
- der Messwert (z.B. 60 Grad Celsius)
- Datum/Zeit (z.B. 11.11.2007/15:34:32)

Diese Daten werden in einem DWH als Star-Schema modelliert, wie in Abbildung 1 dargestellt. Die Bedeutung der Tabellen wird in Tabelle 1 näher erläutert.

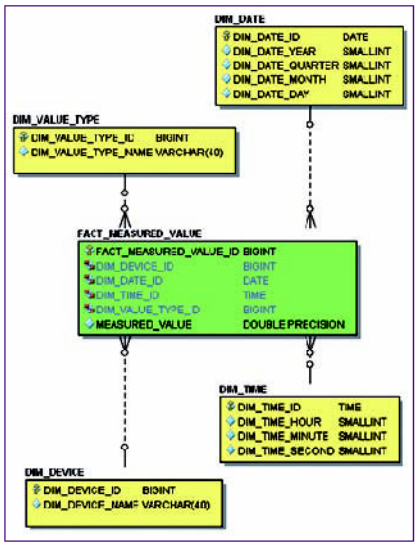
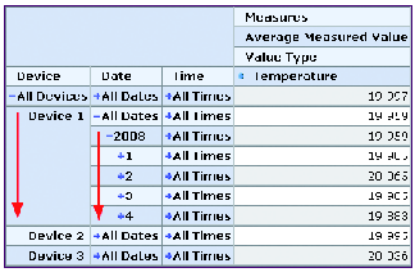


Abb. 1: Star-Schema

Da sich die Anforderungen an eine OLTP- und OLAP-Datenbank in der Regel im Hinblick auf die Anzahl der gleichzeitigen Datenbankverbindungen, Verfügbarkeit und Konfiguration unterscheiden, werden diese beiden Systeme in je einer eigenen Datenbank betrieben.



			Measures
			Average Measured Value
			Value Type
Device	Date	Time	Temperature
All Devices	All Dates	All Times	19 207
Device 1	All Dates	All Times	19 [illegible]
	2008	All Times	19 359
	1	All Times	19 [illegible]
	2	All Times	20 365
	3	All Times	19 [illegible]
	4	All Times	19 363
Device 2	All Dates	All Times	19 [illegible]
Device 3	All Dates	All Times	20 036

Abb. 2: Webbasierter OLAP Client mit Drill-Down

Die Befüllung der Dimensionstabellen und der Faktentabelle erfolgt durch einen ETL-Prozess (Extraktion, Transformation, Laden), der die benötigten Daten aus dem operativem OLTP-System *extrahiert*, anhand definierter Regeln für das Zielsystem *transformiert* und schließlich in die OLAP-Datenbank *lädt*.

Für die Befüllung der OLAP-Datenbank wird hier auf einen ETL-Prozess verzichtet. Stattdessen wurde für jede Dimensionstabelle sowie für die Faktentabelle je eine Firebird Stored Procedure geschrieben, mit der eine konfigurierbare Befüllung durchgeführt werden kann.

Um die Unterschiede in der Ausführungszeit und der *Non-Indexed vs. Index-Reads* ohne und mit Aggregatstabellen besser veranschaulichen zu können, darf sich schon eine größere Anzahl an Datensätzen in diesen Tabellen, vor allem aber in der Faktentabelle *FACT_MEASURED_VALUE*, befinden. Tabelle 2 gibt einen Überblick über das Ergebnis nach der Befüllung.

Um die oben gestellte Aufgabe zu lösen, kann ein Datenbankentwickler eine entsprechende SQL-Abfrage an die OLAP-Datenbank formulieren, oder er verwendet einen OLAP-Client, der eine visuelle Abfragendefinition per Benutzerinteraktion unterstützt. Der Benutzer kann sich per Drill-Down entlang der Dimensionen Gerät und Datum zum gewünschten Ergebnis navigieren. So zum Beispiel mit einer JPivot-basierten Webanwendung (Abbildung 2), die Bestandteil der Mondrian-Distribution ist. Die roten Pfeile stellen den für den Benutzer möglichen Drill-Down-Navigationspfad in den beiden Dimensionen dar. Anhand des in Mondrian aktivierten Statement Tracings kommt man sehr rasch zu der ausgeführten SQL-Abfrage, die in einer ersten Version noch keine Aggregatstabelle verwendet (Listing 1).

Führt man diese Abfrage mit einem SQL-Tool aus, erhält man in der Desktop-PC-Umgebung das Ergebnis in ca. einer Minute und 17 Sekunden. Die Indexed vs. Non-Index Reads für diese Abfrage zeigen ein interessantes Ergebnis (vgl. Abbildung 3), nämlich eine Unmenge an indizierten Leseoperationen in den unterschiedlichen Tabellen. Man muss kein Experte sein, um zu sehen, dass hier das DBMS einiges an Arbeit zu verrichten hat, um an das gewünschte Ergebnis zu kommen.

Tabelle	Zweck
DIM_DATE	Dimensionstabelle Datum: Ein Datensatz entspricht einem Datum entlang der Hierarchien Jahr, Quartal, Monat, Tag.
DIM_TIME	Dimensionstabelle Zeit: Ein Datensatz entspricht einer Uhrzeit entlang der Hierarchien Stunde, Minute, Sekunde.
DIM_DEVICE	Dimensionstabelle Gerät: Ein Datensatz identifiziert eindeutig ein Gerät (Bauteil) im DWH.
DIM_VALUE_TYPE	Dimensionstabelle Messwerttyp: Ein Datensatz entspricht einem Typ eines Messwerts, zum Beispiel: Temperatur, Strom oder Spannung.
FACT_MEASURED_VALUE	Faktentabelle: Speichert das zu analysierende Faktum, auch „Measure" genannt. In unserem Fall einen Messwert eines bestimmten Typs (DIM_VALUE_TYPE_ID) für ein bestimmtes Gerät (DIM_DEVICE_ID) zu einem bestimmten Zeitpunkt (DIM_DATE_ID und DIM_TIME_ID).

Tab. 1: Tabellen des Star-Schemas

Tabelle	Zweck
DIM_DATE	Befüllung für das Jahr 2008 = 366 Datensätze
DIM_TIME	Befüllung für einen Tag mit der Hierarchie „Sekunde" als kleinste Einheit = 24 * 60 * 60 = 86.400 Datensätze
DIM_DEVICE	Drei Geräte = 3 Datensätze
DIM_VALUE_TYPE	Drei Messwerttypen: Spannung, Strom und Temperatur = 3 Datensätze
FACT_MEASURED_VALUE	Für das gesamte Jahr 2008 wird pro Minute für jedes Gerät für jeden Messwerttyp ein Messwert erzeugt = 366 * 24 * 60 * 3 * 3 = 4.743.360 Datensätze

Tab. 2: Befüllungsstatistik der Tabellen des Star-Schemas

Aggregatstabellen als Nachbrenner

Aggregatstabellen sind keine neue Errungenschaft im Bereich der Datenbanktechnologien. „Enterprise-fähige" DBMS wie Oracle, DB2 oder Microsoft SQL Server unterstützen die Persistie-

Listing 1

OLAP-Abfrage ohne Aggregatstabelle

```
select
  "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" as "c0",
  "DIM_DATE"."DIM_DATE_YEAR" as "c1",
  "DIM_DATE"."DIM_DATE_QUARTER" as "c2",
  "DIM_DEVICE"."DIM_DEVICE_NAME" as "c3",
  avg("FACT_MEASURED_VALUE"."MEASURED_VALUE")
                                        as "m0"
from
  "DIM_VALUE_TYPE" "DIM_VALUE_TYPE",
  "FACT_MEASURED_VALUE" "FACT_MEASURED_VALUE",
  "DIM_DATE" "DIM_DATE",
  "DIM_DEVICE" "DIM_DEVICE"
where
  "FACT_MEASURED_VALUE"."DIM_VALUE_TYPE_ID" =
        "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_ID" and
  "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" =
                                        'Temperature' and
  "FACT_MEASURED_VALUE"."DIM_DATE_ID" =
                        "DIM_DATE"."DIM_DATE_ID" and
  "DIM_DATE"."DIM_DATE_YEAR" = 2008 and
  "FACT_MEASURED_VALUE"."DIM_DEVICE_ID" =
                    "DIM_DEVICE"."DIM_DEVICE_ID" and
  "DIM_DEVICE"."DIM_DEVICE_NAME" = 'Device 1'
group by
  "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME",
  "DIM_DATE"."DIM_DATE_YEAR",
  "DIM_DATE"."DIM_DATE_QUARTER",
  "DIM_DEVICE"."DIM_DEVICE_NAME"
```

Anzeige

Table	Non-Indexed	Indexed	Updates	Deletes	Inserts
DIM_DATE		527.040			
FACT_MEASURED_VALUE		1.561.120			
DIM_VALUE_TYPE		1.561.120			
DIM_DEVICE		1			

Abb. 3: Indexed vs. Non-Indexed Reads ohne Aggregatstabellen

rung der Ergebnismenge einer Abfrage, inklusive unterschiedlicher Strategien zur Aktualisierung, falls sich in den Basistabellen an den Ausgangsdaten etwas geändert hat. Diesen Mechanismus bezeichnen die Hersteller oft als *Materialized Views* oder auch *Indexed Views*. In diesem Artikel wird der Begriff „Aggregatstabelle" verwendet, sofern auf keine DBMS-spezifische Implementierung Bezug genommen wird.

Die Hauptaufgabe einer Aggregatstabelle ist es, basierend auf einem Zugriffsmuster die Anzahl der Datensätze im Vergleich zur Basistabelle um ein Vielfaches zu reduzieren und die Ergebnismenge zu persistieren. Im vorliegenden Fall ist das Zugriffsmuster die Aggregierung der Faktentabelle über die Dimension Datum bis zur Hierarchie Quartal. Zusätzlich existiert für unser Beispiel seitens der Qualitätssicherungsabteilung die Anforderung, dass auch Abfragen bis zur Hierarchie Monat performant sein müssen. Das hierfür notwendige Aggregatsschema kann, wie in Abbildung 4 dargestellt, modelliert werden.

Bei der Aggregatstabelle *AGG_FACT_FMV_MONTH* handelt es sich um eine abgewandelte Version der Faktentabelle aus dem Star-Schema. Die Dimension Datum ist nun bis zur Hierarchie Monat vollständig in die Aggregatstabelle gewandert und wird nicht mehr über eine Fremdschlüsselbeziehung zur Dimensionstabelle *DIM_DATE* modelliert. Des Weiteren werden die berechneten Aggregate *MIN, MAX, AVG* und *SUM* gespeichert. Zusätzlich wird die Anzahl der betroffenen Datensätze in der Faktentabelle für das Aggregat abgespeichert. Auf die Tabelle *FMV_DELTA* komme ich später noch zu sprechen, wenn ich unterschiedliche Befüllungs- bzw. Aktualisierungsstrategien vorstellen werde. Setzen wir zu diesem Zeitpunkt eine korrekt befüllte Aggregatstabelle voraus, dann sieht die Performance für unsere Fragestellung mit gleicher Ergebnismenge schon bedeutend besser aus. Listing 2 zeigt die von Mondrian generierte SQL-Abfrage.

In dieser Abfrage ist zu sehen, dass nun die Aggregatstabelle *AGG_FACT_FMV_MONTH* und nicht die Faktentabelle *FACT_MEASURED_VALUE* verwendet wird, um für dasselbe Drill-Down das gewünschte Ergebnis zu erhalten. Wird diese Anweisung wiederum mit einem SQL-Tool ausgeführt, erhält man die Ergebnismenge in 30 ms. Die drastisch reduzierte Anzahl an Leseoperationen ist in Abbildung 5 zu sehen. Nicht sehr überraschend – ist doch die Aggregatstabelle auf 12 * 3 * 3 = 108 Datensätze im Vergleich zu 4.743.360 Datensätzen in der Faktentabelle geschrumpft.

Listing 2

OLAP-Abfrage mit Aggregatstabelle

```
select
  "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" as "c0",
  "AGG_FACT_FMV_MONTH"."DIM_DATE_YEAR" as "c1",
  "AGG_FACT_FMV_MONTH"."DIM_DATE_QUARTER" as "c2",
  "DIM_DEVICE"."DIM_DEVICE_NAME" as "c3",
  sum("AGG_FACT_FMV_MONTH"."AVG_MEASURED_VALUE" * "AGG_FACT_FMV_MONTH"
                    ."COUNT_MEASURED_VALUE") / sum("AGG_FACT_FMV_MONTH".
                                                  "COUNT_MEASURED_VALUE") as "m0"
from
  "DIM_VALUE_TYPE" "DIM_VALUE_TYPE",
  "AGG_FACT_FMV_MONTH" "AGG_FACT_FMV_MONTH",
  "DIM_DEVICE" "DIM_DEVICE"
where
  "AGG_FACT_FMV_MONTH"."DIM_VALUE_TYPE_ID" = "DIM_VALUE_TYPE".
                                              "DIM_VALUE_TYPE_ID" and
  "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME" = 'Temperature' and
  "AGG_FACT_FMV_MONTH"."DIM_DATE_YEAR" = 2008 and
  "AGG_FACT_FMV_MONTH"."DIM_DEVICE_ID" = "DIM_DEVICE"."DIM_DEVICE_ID" and
  "DIM_DEVICE"."DIM_DEVICE_NAME" = 'Device 1'
group by
  "DIM_VALUE_TYPE"."DIM_VALUE_TYPE_NAME",
  "AGG_FACT_FMV_MONTH"."DIM_DATE_YEAR",
  "AGG_FACT_FMV_MONTH"."DIM_DATE_QUARTER",
  "DIM_DEVICE"."DIM_DEVICE_NAME"
```

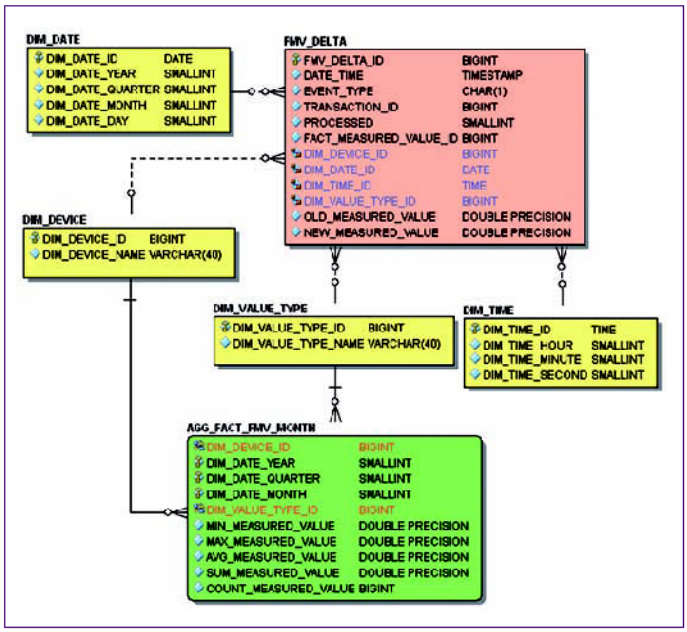
Listing 3

Stored Procedure für die Snapshot-Aktualisierungsstrategie

```
create procedure p_ssload_agg_fact_fmv_month (
 delete_table SmallInt)
as
begin
 if (delete_table = 1) then
 begin
  delete from agg_fact_fmv_month;
 end

 insert into agg_fact_fmv_month
 select
  fmv.DIM_DEVICE_ID
  , dd.dim_date_year
  , dd.dim_date_quarter
  , dd.dim_date_month
  , fmv.DIM_VALUE_TYPE_ID
  , min(fmv.MEASURED_VALUE)
  , max(fmv.MEASURED_VALUE)
  , avg(fmv.MEASURED_VALUE)
  , sum(fmv.MEASURED_VALUE)
  , count(fmv.MEASURED_VALUE)
 from
  fact_measured_value fmv join dim_date dd on
                    (fmv.DIM_DATE_ID = dd.DIM_DATE_ID)
 group by
  fmv.DIM_DEVICE_ID
  , dd.dim_date_year
  , dd.dim_date_quarter
  , dd.dim_date_month
  , fmv.DIM_VALUE_TYPE_ID
 ;
end
```

Abb. 4: Aggregatsschema



Eine sehr gelungene Optimierung für dieses OLAP-Szenario.

Query Rewriting

Mondrian muss natürlich in einer so genannten „Cube-Definitions-Datei" bekannt gemacht werden, dass zusätzlich zur Faktentabelle auch noch ein oder mehrere Aggregatstabellen existieren. Wurde diese Definition korrekt durchgeführt, ist der OLAP-Server in der Lage, ausgehend von der Basisabfrage auf die Faktentabelle diese so umzuwandeln, dass die Aggregatstabelle verwendet wird. Diesen Mechanismus nennt man *Query Rewriting*. Er stellt einen wichtigen Bestandteil bei der Verwendung von Aggregatstabellen dar, um die Transparenz für den Benutzer zu gewährleisten. Je nach Cleverness dieser Komponente ist es sogar möglich, dass die Aggregatstabelle nicht 1:1 auf die aggregierte Abfrage abgebildet werden können muss. So können zum Beispiel in einer Abfrage Hierarchien einer Dimension in einer höheren Hierarchiestufe mit Aggregatstabellen niedrigerer Hierarchiestufe abgebildet werden. So geschehen in obigem Beispiel, in dem die Drill-Down-Analyse auf das Quartal durchgeführt wurde, der OLAP-Server jedoch in der Lage war, die auf den Monat basierte Aggregatstabelle zu verwenden.

Wird kein Query Rewriting unterstützt, ist die Verwendung einer Aggregatstabelle für den Benutzer nicht mehr transparent, sondern ihm muss bekannt sein, welche Aggregatstabellen mit welchem Inhalt existieren, um diese entsprechend in den SQL-Anweisungen explizit zu verwenden. Allerdings bieten die DBMS-Hersteller Query Rewriting in der Regel erst in den teuren Enterprise-Editionen an, trotz der Unterstützung von Materialized bzw. Indexed Views in kostengünstigeren Editionen!

Firebird als DBMS bietet kein Query Rewriting an – im vorliegenden Fall übernimmt das eine andere Komponente in der DWH-Architektur, nämlich der OLAP-Server. Der Benutzer braucht hier von den Aggregatstabellen nichts zu wissen und profitiert bei demselben Drill-Down-Verhalten von der besseren Performance.

Aktualisierungsstrategien

Mit einem guten Aggregatsschemadesign kann man sehr viel an Performance gewinnen. Hier heißt es aber, einen Trade-Off zu finden und nicht für jede beliebige Abfrage eine entsprechende Aggregatstabelle zu erstellen, sondern selektiv vorzugehen und zum Beispiel für die zehn meistverwendeten OLAP-Abfragen in einem DWH entsprechende Aggregatstabellen vorzusehen.

IMPRESSUM

Verlag:
Software & Support Verlag GmbH

Anschrift der Redaktion:
Entwickler Magazin
Software & Support Verlag GmbH
Geleitsstraße 14
D-60599 Frankfurt am Main
Tel. +49 (0)69 630089 0
Fax. +49 (0)69 630089 89
redaktion@entwickler-magazin.de
entwickler-magazin.de

Chefredakteur: Masoud Kamali
Advisory Board: Holger Flick, Rudolf Jansen, Olaf Monien
Redaktion: Nicole Bechtel, Robert Lippert, Markus Zeischke

Autoren dieser Ausgabe: Ben Chelf, Joachim Dürr, Marco Gerlach, Andreas Holubek, Rudolf Jansen, Thomas Kaufmann, Daniel Magin, Thomas Meinike, Christian Metzger, Bernd Ott, Thomas Pfister, Carsten Rogas, Stefan Rojacher Uwe Schirmer, Felix Schrader, Holger Seubert, Thomas Steinmaurer, Ralph Steyer, Dapeng Wang, Tobias Wassermann, Christian Weber,
Chefin vom Dienst: Nicole Bechtel
Schlussredaktion: Nicole Bechtel, Katharina Klassen, Frauke Pesch
Leiter Grafik/Produktion: Jens Mainz
Layout, Titel: Kristin Brockmann, Jessica Demirkaya
CD/DVD-Erstellung: Daniel Zuzek, Özkan Peksan

Anzeigenverkauf:

Entwickler Magazin
Patrik Baumann
Software & Support Verlag GmbH
Tel. +49 (0)69 630089 20
pbaumann@entwickler-magazin.de

Hanno Arens
Software & Support Verlag GmbH
Tel. +49 (0)69 630089 29
harens@entwickler-magazin.de

Open Source
Verlagsbüro Ohm-Schmidt
Osmund Schmidt
Schneckenburger Str. 22
30177 Hannover
Tel. +49 (0)511 2354164
Fax: +49 (0)1805 06033695669
E-Mail: osmund@ohm-schmidt.de

Es gilt die Anzeigenpreisliste Nr. 13

Pressevertrieb:
DPV Network GmbH
Tel.:+49(0)40 23711 0
www.dpv-network.de

Druck: PVA, Landau

Abo-Service:
Software & Support Verlag GmbH
Tel.: +49 (0)69 630089 0
Fax.: +49 (0)69 630089 89
entwickler-magazin.de/service

Abonnementpreise der Zeitschrift (inkl. Leser-CD):

Inland:	6 Ausgaben	€ 35,00
Studentenpreis:	6 Ausgaben	€ 28,50
europ. Ausland:	6 Ausgaben	€ 45,00
Stud. europ. Ausland:	6 Ausgaben	€ 38,50

Abonnementpreise der Zeitschrift (inkl. Leser-CD plus Profi-CD):

Inland:	6 Ausgaben & CD	€ 99,-
Studentenpreis:	6 Ausgaben & CD	€ 80,-
europ. Ausland:	6 Ausgaben & CD	€ 109,-
Stud. europ. Ausland:	6 Ausgaben & CD	€ 90,-

Abonnementpreise der Profi-CD:

Inland:	6 CD-ROM	€ 72,-
Studentenpreis:	6 CD-ROM	€ 62,-
europ. Ausland:	6 CD-ROM	€ 82,-
Stud. europ. Ausland:	6 CD-ROM	€ 72,-

ISSN: 1619-7941

Erscheinungsweise: zweimonatlich

Table	Non-Indexed	Indexed	Updates	Deletes	Inserts
DIM_DEVICE		36			
AGG_FACT_FMV_MONTH		36			
DIM_VALUE_TYPE		1			

Abb. 5: Indexed vs. Non-Indexed Reads mit Aggregatstabellen

Mit der Einführung von Aggregatstabellen wird man unmittelbar mit dem Thema *Redundanz* konfrontiert. Im Fall von Aggregatstabellen ist der wohl wichtigste Faktor die Aktualität der voraggregierten Daten im Vergleich zur On-The-Fly-Aggregierung durch eine OLAP-Abfrage auf die Faktentabelle. Werden die Daten in einer Faktentabelle geändert, weil neue Datensätze durch den ETL-Prozess geladen worden sind, so ist der Inhalt der Aggregatstabelle bereits wieder veraltet. Dies hat unmittelbar zur Folge, dass eine Abfrage auf die Aggregatstabelle nicht notwendigerweise dieselbe Ergebnismenge wie eine Abfrage auf die Faktentabelle zurückliefert. Hier heißt es, anforderungsgetrieben eine geeignete Strategie für die Aktualisierung der Aggregatstabelle(n) zu finden.

Im Wesentlichen gibt es drei unterschiedliche Aktualisierungsstrategien, denen man in der Literatur und in der Praxis begegnet: *Snapshot*, *Eager* und *Lazy*. Bei einer *Snapshot-Aktualisierung* wird die Aggregatstabelle immer vollständig entleert und neu aufgebaut. Die Implementierung ist einfach, allerdings steigt die Serverauslastung für die Aggregierung mit der Anzahl der Datensätze in der Faktentabelle. Eine Snapshot-Aktualisierung wird in der Regel periodisch durchgeführt, zum Beispiel automatisch nach dem Laden der Faktentabelle durch den ETL-Prozess. Bei der *Eager-Strategie* befindet sich auf der Faktentabelle für jede davon abhängige Aggregatstabelle ein Delete/Insert/Update Trigger, der quasi in Echtzeit eine inkrementelle „Nachaggregierung" vornimmt. Hierbei wird die Aggregatstabelle nicht vollständig neu aufgebaut, sondern der Trigger aktualisiert die bereits durchgeführte Aggregierung mit den neuen Werten aus der Faktentabelle. Die Implementierung der inkrementellen Aggregierung ist etwas komplexer. Der Nachteil dieses Ansatzes ist, dass sich zum Beispiel die Ausführungszeit des ETL-Prozesses verlängert, da bei jedem *COMMIT* der Trigger auf der Faktentabelle angestoßen und die inkrementelle Aggregierung sofort durchgeführt wird.

Der *Lazy-Ansatz* hat wiederum einen Trigger auf der Faktentabelle, der jede Änderung in eine eigene Tabelle protokolliert (siehe Tabelle *FMV_DELTA* in Abbildung 4). In regelmäßigen Abständen wird eine Stored Procedure angestoßen, die alle noch nicht abgearbeiteten Datensätze in der Protokolltabelle durchläuft

Snapshot, Eager und Lazy

und für jeden Datensatz den bereits beschriebenen Eager-Mechanismus ausführt. Auch hier sind bei jeder Änderung in der Faktentabelle zusätzliche Schreiboperationen notwendig (nämlich in die Protokolltabelle), aber im Unterschied zum Eager-Mechanismus kann die inkrementelle Aktualisierung zu einem späteren, womöglich günstigeren Zeitpunkt durchgeführt werden.

Eine Implementierung aller Aktualisierungsstrategien für eine Firebird-2.1-Datenbank soll hier ebenfalls gezeigt werden. Bei DBMS, die Stored Procedure und Trigger unterstützen, kann eine Implementierung ähnlich aussehen. In Listing 3 ist die Stored Procedure abgebildet, die die Aggregatstabelle nach der Snapshot-Aktualisierungsstrategie befüllt. Hier handelt es sich um ein einfaches *INSERT INTO* mit einer aggregierten *SELECT*-Abfrage auf die Faktentabelle. Listing 4 (zu finden auf der Heft-CD) zeigt die Stored Procedure und den Delete/Insert/Update Trigger für den Eager-Mechanismus. In diesem Fall ist die Stored Procedure etwas komplexer ausgefallen, da sich die Aktualisierungslogik je nach Art der schreibenden Operation (*Delete/Insert/Update*) unterscheidet. Listing 5 (zu finden auf der Heft-CD) zeigt den einfach zu realisierenden Protokollierungs-Trigger und die Stored Procedure für den Lazy-Modus. Die Stored Procedure ruft für jeden protokollierten Datensatz die Stored Procedure *P_EAGLOAD_AGG_FACT_FMV_MONTH* aus Listing 4 (Heft-CD) auf. Danach werden die Protokolldatensätze als verarbeitet markiert bzw. gelöscht, abhängig vom Wert des Eingabeparameters der Stored Procedure.

Fazit

Aggregatstabellen sind eine sehr interessante und effektive Möglichkeit, um die Antwortzeit aggregierter Abfragen in OLAP-Szenarien drastisch zu verkürzen. DBMS wie Oracle, DB2 und Microsoft SQL Server bieten hierfür schon Implementierungen in Form von Materialized Views bzw. Indexed Views an. Firebird in der Version 2.1 und andere Open Source DBMS wie PostgreSQL und MySQL bieten aktuell noch kein vergleichbares Feature an, allerdings kann man mit Firebird-Bordmitteln unter Verwendung von Stored Procedures und Triggern selbst Hand anlegen und die gängigsten Aktualisierungsstrategien implementieren. Man erhält ein Aggregatsschema, das zwar Redundanzen aufweist, die allerdings gut beherrschbar sind. In Kombination mit einem OLAP-Server wie Mondrian, der Query Rewriting unterstützt, hat man eine Low-Cost-Lösung basierend auf Open-Source-Software zur Verfügung, die durchaus für umfangreichere DWH-Projekte eingesetzt werden kann.

Thomas Steinmaurer *ist wissenschaftlicher Mitarbeiter am Software Competence Center Hagenberg (SCCH) [4] in Österreich und beschäftigt sich mit Datenmanagement und Data Warehousing im industriellen Umfeld. Des Weiteren ist er verantwortlich für die LogManager Series bei Upscene Productions [5]. Er ist Mitbegründer der Firebird Foundation [6] und regelmäßiger Autor im Entwickler Magazin. Der Autor ist unter folgender E-Mail-Adresse erreichbar: thomas.steinmaurer@scch.at.*

Links & Literatur

[1] de.wikipedia.org/wiki/Data-Warehouse

[2] www.firebirdsql.org

[3] mondrian.pentaho.org

[4] www.scch.at

[5] www.upscene.com

[6] www.firebirdsql.org/index.php?op=ffoundation