Die neuen analytischen Funktionen und deren Anwendung

Firebird-3-Vorschau – Teil 3

Im dritten Teil dieser Firebird-3-Artikelreihe beschäftigen wir uns mit den hinzugekommenen SQL-Spracherweiterungen zur einfacheren Beantwortung von analytischen Fragestellungen.

von Thomas Steinmaurer

Teil 1 dieser Artikelreihe beschäftigte sich mit dem neuen Datentyp *BOOLEAN*, der vereinfachten Anlage von *Auto Increment*-Spalten, *Stored Functions* und *Packages* [1]. Der zweite Teil [2] widmete sich ausschließlich *DDL Triggern*. In diesem Artikel werden wir uns mit den neuen analytischen Funktionen in Firebird 3 in Form der *Window Functions* und deren Anwendung vertraut machen.

Aggregatsfunktionen

Jeder Datenbankentwickler hat vermutlich schon mal etwas mit Aggregatsfunktionen (*COUNT, SUM, MIN, MAX, AVG*) zu tun gehabt. Mit diesen Funktionen können Datensätze nach angegebenen Feldern gruppiert und Aggregate als separate Spalten in der Ergebnismenge berechnet werden. Als Ausgangsdatenmenge (**Abb. 1**) für die in diesem Artikel erstellten Beispiele dient eine abgespeckte Version der Faktentabelle *FACT_MEASURED_VALUE* des Data Warehouses aus meinem Artikel zu Aggregatstabellen in Firebird [3]. In dieser Tabelle sind Spannung, Strom und Temperatur für drei Geräte/Sensoren gespeichert. Obwohl in einer realen Umsetzung auch noch der Zeitstempel und andere Daten gespeichert sind, verzichte ich in diesem Artikel bewusst auf diese Informationen, um die Beispiele übersichtlich zu gestalten. Eine klassische Fragestellung in unserem Beispiel ist nun der durchschnittliche Messwert je Werttyp (Spannung, Strom, Spannung) über alle Geräte. Die dafür benötigte SQL-Abfrage ist in Listing 1 bzw. die Ergebnismenge in **Abbildung 2** dargestellt.

FACT_MEASURED_VALUE_ID	DEVICE_NAME	VALUE_TYPE_NAME	MEASURED_VALUE
1	Device 1	Voltage	3,3283
2	Device 1	Current	0,0083
3	Device 1	Temperature	-29,4964
4	Device 2	Voltage	2,1403
5	Device 2	Current	0,0081
6	Device 2	Temperature	45,6498
7	Device 3	Voltage	3,6052
8	Device 3	Current	0,0097
9	Device 3	Temperature	-7,4452

Abb. 1: Ausgangsdatenmenge

Die Verwendung einer Aggregatsfunktion wie *AVG* erfordert die Angabe einer *GROUP BY*-Klausel, die die Gruppierung/Partitionierung für die Berechnung des aggregierten Wertes definiert. Das heißt, die Aggregatsberechnung erfolgt immer auf den Daten dieser Gruppierungsdefinition. Das ist auch bereits ein großer Nachteil, wenn man innerhalb einer Abfrage einen aggregierten Wert haben möchte, der in Bezug auf die Gruppierungsdefinition unabhängig von der zugrunde liegenden Ergebnismenge sein soll. In analytischen Fragestellungen gibt es allerdings eine Vielzahl an Beispielen, wo eine Aggregatsberechnung in einem separaten Feld keine Filterung der Ergebnismenge zur Folge haben soll. Ein guter Zeitpunkt um die Brücke zu den neuen analytischen Funktionen zu bauen.

Inline Select

Angenommen wir wollen die Anzahl der Datensätze der Ergebnismenge in einem separaten Feld anzeigen. Dies kann nicht mehr über ein *COUNT* in Kombination mit einem *GROUP BY* erfolgen, da dies die bereits erwähn-

te Filterung der Ergebnismenge zur Folge haben würde. Ohne die analytischen Funktionen in Firebird 3 ist ein „Inline Select“ notwendig, das die Ermittlung der Anzahl der Datensätze durchführt. Siehe dazu die SQL-Abfrage in Listing 2 mit der Ergebnismenge in **Abbildung 3**.

Es wurden 81 *Non-Indexed* und 9 *Indexed* Reads benötigt. Diese Statistik ist für den weiteren Verlauf interessant, wenn wir uns die Umsetzung derselben Fragestellung mit den neuen analytischen Funktionen ansehen werden. Zugegeben, das hier vorgestellte Beispiel ist sehr vereinfacht und auch konstruiert, aber gut genug, um die Idee hinter der Verwendung eines *Inline Selects* zu verstehen.

Etwas umfangreicher ist die SQL-Abfrage in Listing 3, die für alle Datensätze vom Werttyp *Voltage* den durchschnittlichen Spannungswert ermittelt und die prozentuale Abweichung des gemessenen Spannungswertes zum ermittelten Spannungsmittelwert berechnet. Das Ergebnis ist in **Abbildung 4** dargestellt. Der Ausführungsplan für diese Abfrage sieht wie folgt aus:

```
PLAN (F1 NATURAL)
PLAN (F1 NATURAL)
PLAN (FMV ORDER PK_FACT_MEASURED_VALUE)
```

Es waren 54 *Non-Indexed* und 9 *Indexed* Reads auf der Faktentabelle notwendig, um die Abfrage zu bearbeiten.

Wirklich unübersichtlich wird es, wenn die Abfrage nicht nur auf Spannungswerte, sondern für alle verfügbaren Werttypen anwendbar sein soll (Listing 4, **Abb. 5**). Nicht nur dass die SQL-Abfrage umfangreicher wird, sondern die notwendige Überprüfung auf die Konstanten *Voltage*, *Current* und *Temperature* hat zur Folge, dass beim Hinzukommen eines neuen Typs die SQL-Abfrage angepasst werden muss.

VALUE_TYPE_NAME	AVG
Current	0,0087
Temperature	2,9027
Voltage	3,0246

Abb. 2: Ergebnismenge – Durchschnittlicher Messwert je Werttyp über alle Geräte

FACT_MEASURED_VALUE_ID	DEVICE_NAME	VALUE_TYPE_NAME	MEASURED_VALUE	MV_COUNT
1	Device 1	Voltage	3,3283	9
2	Device 1	Current	0,0083	9
3	Device 1	Temperature	-29,4964	9
4	Device 2	Voltage	2,1403	9
5	Device 2	Current	0,0081	9
6	Device 2	Temperature	45,6498	9
7	Device 3	Voltage	3,6052	9
8	Device 3	Current	0,0097	9
9	Device 3	Temperature	-7,4452	9

Abb. 3: Ergebnismenge – Inline Select zur Ermittlung der Datensatzanzahl

FACT_MEASURED_VALUE_ID	DEVICE_NAME	VALUE_TYPE_NAME	MEASURED_VALUE	MV_AVG	MV_PERC_AVG
1	Device 1	Voltage	3,3283	3,0246	110,0399
4	Device 2	Voltage	2,1403	3,0246	70,7640
7	Device 3	Voltage	3,6052	3,0246	119,1961

Abb. 4: Ergebnismenge – Inline Selects zur Durchschnittsermittlung und Berechnung der prozentualen Abweichung für den Messwerttyp Spannung

Listing 1: SQL-Abfrage – Durchschnittlicher Messwert je Werttyp über alle Geräte

```
select
  fmv.value_type_name
  , avg(measured_value)
from
  fact_measured_value fmv
group by
  fmv.value_type_name
order by
  fmv.value_type_name
```

Listing 2: Inline Select zur Ermittlung der Datensatzanzahl

```
select
  fmv.*
  , (select count(*) from fact_measured_value) mv_count
from
  fact_measured_value fmv
order by
  fmv.fact_measured_value_id

//Mit folgendem Ausführungsplan:

PLAN (FACT_MEASURED_VALUE NATURAL)
PLAN (FMV ORDER PK_FACT_MEASURED_VALUE)
```

Listing 3: Inline Selects zur Durchschnittsermittlung und Berechnung der prozentualen Abweichung für den Messwerttyp Spannung

```
select
  fmv.*
  , (select
      avg(f1.measured_value)
    from
      fact_measured_value f1
    where
      f1.value_type_name = 'Voltage'
  ) as mv_avg
  , fmv.measured_value / (
      select
        avg(f1.measured_value)
      from
        fact_measured_value f1
      where
        f1.value_type_name = 'Voltage'
  ) * 100 as mv_perc_avg
from
  fact_measured_value fmv
where
  fmv.value_type_name = 'Voltage'
order by
  fmv.fact_measured_value_id
```

FACT_MEASURED_VALUE_ID	DEVICE_NAME	VALUE_TYPE_NAME	MEASURED_VALUE	MV_AVG	MV_PERC_ARG
1	Device 1	Voltage	3,3283	3,0246	110,0399
2	Device 1	Current	0,0083	0,0087	95,1052
3	Device 1	Temperature	-29,4964	2,9027	-1.016,1653
4	Device 2	Voltage	2,1403	3,0246	70,7640
5	Device 2	Current	0,0081	0,0087	93,2818
6	Device 2	Temperature	45,6498	2,9027	1.572,6566
7	Device 3	Voltage	3,6052	3,0246	119,1961
8	Device 3	Current	0,0097	0,0087	111,6130
9	Device 3	Temperature	-7,4452	2,9027	-256,4913

Abb. 5: Ergebnismenge – Inline Selects zur Durchschnittsermittlung und Berechnung der prozentuellen Abweichung für alle drei Messwerttypen

Auch die Datenbank-Engine muss mehr arbeiten, um an das Ergebnis heranzukommen, wie der Ausführungsplan zeigt:

```
PLAN (F1 NATURAL)
PLAN (F1 NATURAL)
PLAN (F1 NATURAL)
PLAN (F1 NATURAL)
PLAN (F1 NATURAL)
PLAN (F1 NATURAL)
PLAN (FMV ORDER PK_FACT_MEASURED_VALUE)
```

Durch die unzähligen Inline Selects waren auch 162 *Non-Indexed* und 9 *Indexed* Reads notwendig.

OVER()-Klausel

Sehen wir uns nun die Umsetzung der Beispiele mit den neuen analytischen Möglichkeiten in Firebird 3 näher an. Aggregatsfunktionen wurden dahingehend erweitert, dass nun eine *OVER()*-Klausel mit den optionalen Subklauseln *PARTITION BY* bzw. *ORDER BY* mitangegeben werden kann. Die formale syntaktische Erweiterung für Aggregatsfunktionen in Firebird 3 sieht wie folgt aus:

```
<aggregate function>([<expr>])
OVER ([PARTITION BY <expr> [, <expr> ...]]
[ORDER BY <expr> [, <expr> ...]])
```

Um die Verwendung etwas klarer zu machen, habe ich das Beispiel aus Listing 2 mit *OVER()* umgesetzt (Listing 5).

Durch die Angabe von *OVER()* arbeitet die im Beispiel verwendete Aggregatsfunktion *COUNT* auf der abgefragten Ergebnismenge, ohne dabei ein *GROUP*

Listing 4: Inline Selects zur Durchschnittsermittlung und Berechnung der prozentualen Abweichung für alle drei Messwerttypen

```
select
  fmv.*
  , case
     when fmv.value_type_name = 'Voltage' then
       (select avg (f1.measured_value) from fact_measured_value f1
          where f1.value_type_name = 'Voltage')

     when fmv.value_type_name = 'Current' then
       (select avg (f1.measured_value) from fact_measured_value f1
          where f1.value_type_name = 'Current')

     when fmv.value_type_name = 'Temperature' then
       (select avg (f1.measured_value) from fact_measured_value f1
          where f1.value_type_name = 'Temperature')

  end as mv_avg
  , case
     when fmv.value_type_name = 'Voltage' then
       (select (fmv.measured_value / avg (f1.measured_value)) * 100
          from fact_measured_value f1 where f1.value_type_name = 'Voltage')

     when fmv.value_type_name = 'Current' then
       (select (fmv.measured_value / avg (f1.measured_value)) * 100
          from fact_measured_value f1 where f1.value_type_name = 'Current')

     when fmv.value_type_name = 'Temperature' then
       (select (fmv.measured_value / avg (f1.measured_value)) * 100
          from fact_measured_value f1 where f1.value_type_name = 'Temperature')
  end as mv_perc_avg
from
  fact_measured_value fmv

order by
  fmv.fact_measured_value_id
```

Listing 5: Ermittlung der Datensatzanzahl mit OVER ()

```
select
  fmv.*
  , count(fmv.measured_value) over() as mv_count
from
  fact_measured_value fmv
order by
  fmv.fact_measured_value_id
```

Listing 6: Durchschnittsermittlung und Berechnung der prozentualen Abweichung mit OVER(PARTITION BY...)

```
select
  fmv.*
  , avg(fmv.measured_value) over(partition by fmv.value_type_name)
      as mv_avg
  , (fmv.measured_value / avg(fmv.measured_value)
      over(partition by fmv.value_type_name)) * 100 as mv_perc_arg
from
  fact_measured_value fmv
order by
  fmv.fact_measured_value_id
```

BY und implizit dadurch eine Filterung durchführen zu müssen. Folgender Ausführungsplan wird für diese SQL-Abfrage vom Optimizer erzeugt:

```
PLAN SORT (FMV NATURAL)
```

Dies spiegelt sich auch in der verringerten Anzahl der benötigten Leseoperationen wider. Mit *OVER()* werden nunmehr 9 *Non-Indexed* Reads benötigt. Die beiden Beispiele zur prozentualen Abweichung des Messwerts zum Durchschnitt aus Listing 3 und Listing 4 können mit den neuen analytischen Möglichkeiten in nur einer Abfrage formuliert werden (Listing 6).

Durch die Verwendung von *PARTITION BY* auf *VALUE_TYPE_NAME* erfolgt die Durchschnittsermittlung je Typ (Spannung, Strom, Temperatur) vollautomatisch, ohne hier Rücksicht auf die vorhandenen Werttypen nehmen zu müssen. Diese Form der Abfrage ist somit auch resistent gegen das Hinzufügen von neuen Werttypen. Im Beispiel ohne *OVER()* aus Listing 3 wurde explizit auf Datensätze vom Typ *Voltage* abgefragt. Mit *PARTITION BY* kann somit eine Aggregatsermittlung basierend auf einer Gruppierung/Partitionierung nach Feldern definiert werden, die keine Filterung der Ergebnismenge nach sich zieht. Die Ergebnismenge für Listing 6 ist in **Abbildung 5** ersichtlich. Trotz der gegebenen Flexibilität bei der Formulierung der Abfrage und des höheren Informationsgewinns in Bezug auf die automatische Unterstützung aller Werttypen bleibt der Ausführungsplan mit *PLAN SORT (FMV NATURAL)* sehr kompakt. Es waren auch hier wiederum nur 9 *Non-Indexed* Reads notwendig. Der neue interne Hash-Join-Algorithmus leistet bei der Verwendung von *OVER()* ausgezeichnete

FACT_MEASURED_VALUE_ID	DEVICE_NAME	VALUE_TYPE_NAME	MEASURED_VALUE	MV_CUM
2	Device 1	Current	0.0083	0.0083
5	Device 2	Current	0.0081	0.0164
8	Device 3	Current	0.0097	0.0262
3	Device 1	Temperature	-29.4964	-29.4964
6	Device 2	Temperature	45.6498	16.1534
9	Device 3	Temperature	-7.4452	8.7082
1	Device 1	Voltage	3.3283	3.3283
4	Device 2	Voltage	2.1403	5.4686
7	Device 3	Voltage	3.6052	9.0738

Abb. 6: Ergebnismenge – Kumulierter Wert mit der OVER()-Subklausel ORDER BY

FACT_MEASURED_VALUE_ID	DEVICE_NAME	VALUE_TYPE_NAME	MEASURED_VALUE	LAG	LEAD
1	Device 1	Voltage	3.3283	<null>	0.0083
2	Device 1	Current	0.0083	3.3283	-29.4964
3	Device 1	Temperature	-29.4964	0.0083	2.1403
4	Device 2	Voltage	2.1403	-29.4964	0.0081
5	Device 2	Current	0.0081	2.1403	45.6498
6	Device 2	Temperature	45.6498	0.0081	3.6052
7	Device 3	Voltage	3.6052	45.6498	0.0097
8	Device 3	Current	0.0097	3.6052	-7.4452
9	Device 3	Temperature	-7.4452	0.0097	<null>

Abb. 7: Ergebnismenge – Beispiel zur Verwendung von LAG und LEAD

VALUE_TYPE_NAME	ROWNUM_GLOBAL	ROWNUM_PARTITIONED	DENSE_RANK	RANK
Current	1	1	1	1
Current	2	2	1	1
Current	3	3	1	1
Temperature	4	1	2	4
Temperature	5	2	2	4
Temperature	6	3	2	4
Voltage	7	1	3	7
Voltage	8	2	3	7
Voltage	9	3	3	7

Abb. 8: Ergebnismenge: Beispiel zur Verwendung von ROW_NUMBER/ DENSE_RANK/RANK

Listing 7: Kumulierter Wert mit der OVER()-Subklausel ORDER BY

```
select
  fmv.*
  , sum(fmv.measured_value) over(partition by fmv.value_type_name
      order by fmv.value_type_name, fmv.fact_measured_value_id) as mv_cum
from
  fact_measured_value fmv
order by
  fmv.value_type_name
  , fmv.fact_measured_value_id
```

Listing 8: Beispiel zur Verwendung von LAG und LEAD

```
select
  fmv.*
  , lag(fmv.measured_value) over (order by fmv.fact_measured_value_id) as lag
  , lead(fmv.measured_value) over (order by fmv.fact_measured_value_id) as lead
from
  FACT_MEASURED_VALUE fmv
order by
  fmv.fact_measured_value_id
```

Listing 9: Beispiel zur Verwendung von ROW_NUMBER/DENSE_RANK/RANK

```
select
  fmv.value_type_name
  , row_number() over (order by fmv.value_type_name) as rownum_global
  , row_number() over (partition by fmv.value_type_name order by
      fmv.value_type_name) as rownum_partitioned
  , dense_rank() over (order by fmv.value_type_name)
  , rank() over (order by fmv.value_type_name)
from
  fact_measured_value fmv
order by
  fmv.value_type_name
```

Arbeit. Dies ist ein wichtiger Aspekt wenn man fernab von unserer Tabelle mit neun Datensätzen eine Tabelle mit Millionen von Datensätzen abfragt.

Neben der Partitionierung zur Aggregatsermittlung mit *PARTITION BY* steht auch noch *ORDER BY* als Subklausel für *OVER()* zur Verfügung. Hiermit kann eine benutzerdefinierte Reihenfolge bei der Anwendung der Aggregationsfunktion angegeben werden. Dies kann zum Beispiel für die Kumulierung von Werten verwendet werden, wo die Reihenfolge der zu verarbeitenden Datensätze eine Rolle spielt. Angenommen, man will in einem separaten Feld den kumulierten Messwert je Messwerttyp ermitteln, so kann dies mit der Abfrage aus Listing 7 erfolgen.

Mit ORDER BY kann eine benutzerdefinierte Reihenfolge angegeben werden.

In **Abbildung 6** ist die Aufsummierung (Kumulierung) des Messwertes je Typ zu sehen. Die Sortierung erfolgt nach dem Typ und der Einfügereihenfolge (definiert durch den Primärschlüsselwert).

Die Erweiterung des SQL-Sprachumfangs mit *OVER()* bei der Verwendung von Aggregatsfunktionen ist nur eine Neuerung. Des Weiteren sind auch noch neue Funktionen hinzugekommen, die wiederum in Kombination mit *OVER()* verwendet werden können. Diese lassen sich in zwei Kategorien einordnen, die im abschließenden Teil dieses Artikels diskutiert werden:

- Navigation
- Ranking

Navigation/Ranking

In den Snapshot Builds von Firebird 3 standen seit Februar 2011 die Navigationsfunktionen *LAG* und *LEAD* sowie die Ranking-Funktionen *ROW_NUMBER*, *DENSE_RANK* und *RANK* zur Verfügung. Beginnen wir mit einem Beispiel zu den Navigationsfunktionen. Hierbei geht es um den Zugriff auf Feldwerte innerhalb eines Datensatzes, die sich in Datensätzen vor bzw. nach dem aktuellen Datensatz befinden. Darum auch der Name „Navigationsfunktion“ (Listing 8).

Wie in **Abbildung 7** zu sehen ist, liefert *LAG* den Messwert des Vorgängerdatensatzes. Dieser ist *NULL* im Falle des ersten Datensatzes. *LEAD* ermittelt den Messwert des Nachfolgerdatensatzes. Auch hier, im Falle des letzten Datensatzes, ist der Zustand *NULL*. Die Schrittweite bzw. der Default-Wert, im Falle, dass der angesprochene Datensatz nicht vorhanden ist, kann mit weiteren Parametern der *LAG/LEAD*-Funktion definiert werden. Die syntaktische Verwendung dieser beiden Funktionen sieht wie folgt aus:

```
LAG(<expr>, <increment>, <default>)
LEAD(<expr>, <increment>, <default>)
```

Werden *<increment>* und *<default>* beim Aufruf nicht angegeben, dann wird ein *<increment>* von 1 bzw. als *<default>* *NULL* verwendet.

Im Gegensatz dazu ermöglichen die Ranking-Funktionen *ROW_NUMBER*, *DENSE_RANK* und *RANK*, Datensätze zu nummerieren. Natürlich auch in Kombination mit *PARTITION BY*, um eine Nummerierung je Gruppe zu erreichen. Das Beispiel in Listing 9 soll die Verwendung dieser Funktionen verdeutlichen (**Abb. 8**).

ROWNUM_GLOBAL liefert eine Nummerierung in Bezug auf alle Datensätze. *ROWNUM_PARTITIONED* erzeugt eine aufsteigende Nummerierung beginnend mit 1 je neuem Messwerttyp durch die Verwendung der *PARTITION BY*-Klausel. *DENSE_RANK* und *RANK* weisen jedem gleichen Messwerttyp dieselbe Nummer zu, allerdings mit dem Unterschied, dass *RANK* Lücken zwischen den einzelnen Messwerttypen entstehen lässt, abhängig davon, wie viele Datensätze vor einer Messwerttypänderung verarbeitet wurden. Bezüglich dieses Unterschieds kann man sich mit einer Eselsbrücke helfen. *RANK* ist konform einer Ergebnisliste von Teilnehmern einer Sportveranstaltung im Falle von Ex-aequo-Platzierungen.

Fazit

Der Mehrwert der neuen analytischen Möglichkeiten innerhalb des SQL-Sprachumfangs ist enorm. Hiermit lassen sich Abfragen für analytische Fragestellungen nicht nur eleganter formulieren, sondern auch die Ausführungszeit verbessert sich im Vergleich zum bis dato herkömmlichen Weg über Inline Selects drastisch.

Thomas Steinmaurer ist wissenschaftlicher Mitarbeiter am Software Competence Center Hagenberg (http://www.scch.at) mit Schwerpunkt Datenmanagement und Data Warehousing im industriellen Umfeld. Des Weiteren ist er für die LogManager-Produktreihe und FB TraceManager bei Upscene Productions (http://www.upscene.com) verantwortlich.

Links & Literatur

[1] Entwickler Magazin 4.2011: Firebird 3 Vorschau, Teil 1

[2] Entwickler Magazin 5.2011: Firebird 3 Vorschau, Teil 2

[3] Entwickler Magazin 5.2008: Aggregatstabellen mit Firebird