Indizes – Fluch und Segen?

Sebastian Reimann
viadee Unternehmensberatung GmbH
Anton-Bruchausen-Straße 8
48147 Münster
sebastian.reimann@viadee.de

Zusammenfassung

Wie heißt es so schön an vielen Stellen in der SAS Hilfe: Die Eingabetabellen müssen nach den Schlüsselvariablen sortiert oder indiziert sein. Ein DataStep Merge, Update oder Modify ist anders gar nicht möglich. An anderer Stelle verspricht man sich vom Anlegen einiger Indizes deutliche Performance Vorteile, da der aufwendige Full Table Scan bei der Ausführung einer Abfrage unterbunden werden kann. Doch hält der Index, was er verspricht? Und welche Nebenwirkungen hat ein Index? Auf diese Fragestellungen soll im Folgenden anhand von zwei Beispielen eingegangen werden.

Schlüsselwörter: Index, PROC SQL, DataStep Update, DataStep Modify

1 Motivation

Die Performance von SAS Auswertungen ist immer wieder ein spannendes Thema. Gerade wenn bei der Auswertung auf die klassische SAS Datenhaltung (BASE oder SPDE Libraries) gesetzt wird, lohnt es sich, ein paar Performance Aspekte näher zu betrachten.

„Ein Datenbankindex, oder kurz Index, ist eine von der Datenstruktur getrennte Indexstruktur in einer Datenbank, die die Suche und das Sortieren nach bestimmten Feldern beschleunigt.“ [1]

Durch den Aufbau eines Index werden bestimmte, abfragerelevante Felder, redundant zur Datenstruktur in einer weiteren Indexstruktur abgelegt, um somit schneller auf die Werte zugreifen zu können. Dabei werden im Index Zeiger auf die Datenstruktur abgelegt, um vom Index-Eintrag schnell auf den kompletten Datensatz zugreifen zu können.

2 Index zur Steigerung der Abfrageperformance

In normalisierten Datenbankstrukturen werden Informationen möglichst redundanzfrei in mehreren Tabellen abgelegt. Die Beziehung der Datensätze zueinander wird über gemeinsame Schlüsselvariablen hergestellt.

Ein typisches Datenmodell eines Finanzdienstleisters könnte folgenden Aufbau haben:

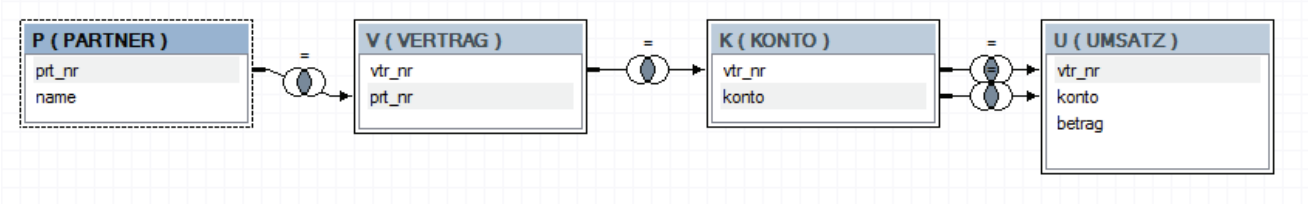

Abbildung 1: Normalisiertes Datenmodell eines Finanzdienstleisters

Ein Partner kann dabei mehrere Verträge abschließen, zu jedem Vertrag wird eine definierte Anzahl an Konten geführt und jedes dieser Konten steht mit einer beliebigen Anzahl an Umsätzen in Verbindung.

Zur Wahrung der referenziellen Integrität der Daten sind i.d.R. Primärschlüssel- und Fremdschlüsselbeziehungen in einer solchen Datenbank definiert. Wenn auf das Datenmodell nur im Rahmen einer dispositiven Verarbeitung zugegriffen wird, wird auf diese Indizes auch gerne verzichtet.

Im Beispiel-Setup haben wir 1.000.000 Partner, die in Summe 10.000.000 Verträge haben. Jeder Vertrag besteht aus bis zu 3 Konten und zu jedem Vertrag werden 10 Buchungen geführt, die auf die 3 Kontoarten verteilt werden.

Möchte man nun zu einer Auswahl von wenigen Partnern die Summe der Umsätze selektieren, hilft ein Tool wie der Enterprise Guide schnell, eine entsprechende SQL Abfrage zu definieren:

```
PROC SQL;
  CREATE TABLE WORK.SUMME_PRT_KLEINER_100 AS
    SELECT P.NAME AS NAME,
          (SUM(U.BETRAG)) AS SUM_BETRAG
    FROM WORK.VERTRAG V,
         WORK.KONTO K,
         WORK.UMSATZ U,
         WORK.PARTNER P
    WHERE V.VTR_NR = K.VTR_NR
      AND K.VTR_NR = U.VTR_NR
      AND K.KONTO = U.KONTO
      AND P.PRT_NR = V.PRT_NR
      AND P.PRT_NR < 100
    GROUP BY P.NAME;
QUIT;

NOTE: SAS threaded sort was used.
NOTE: Table WORK.SUMME_PRT_KLEINER_100 created,
      with 61 rows and 2 columns.

NOTE: PROZEDUR SQL used (Total process time):
      real time           49.21 seconds
      cpu time            1:26.14
```

Bei der Ausgabe stellt sich schnell die Frage, warum die in diesem Fall doch recht triviale Selektion von 61 Zeilen fast 50 Sekunden dauert. Die Vermutung ist, dass ein Index Abhilfe schaffen kann.

Entsprechend des Datenmodells werden folgende Indizes angelegt:

Tabelle 1: Indizes zur Performance-Optimierung

Tabelle	Index
Partner	PRT_NR
Vertrag	PRT_NR
Konto	VTR_NR
Umsatz	KTO=(VTR_NR KONTO)

Die Erstellung der Indizes verursacht natürlich auch Kosten. Im vorliegenden Fall dauert die Erstellung ca. 50 Sekunden. Werden auf die Datenbestände gleichartige Abfragen mehrfach ausgeführt, kann sich dieser Aufwand schnell durch die verbesserte Abfrageperformance rechnen.

Führt man obige Abfrage nach der Indexerstellung erneut aus, stellt man jedoch schnell fest, dass der wirkliche Performance-Schub gar nicht eintritt. Die Abfrage benötigt weiterhin fast 50Sekunden:

```
INFO: Index prt_nr für Optimierung der Where-Bedingung ausgewählt.
NOTE: Table WORK.SUMME_PRT_KLEINER_100 created,
      with 61 rows and 2 columns.

NOTE: PROZEDUR SQL used (Total process time):
      real time           47.87 seconds
      cpu time            1:23.91
```

Über die SAS Option MSGLEVEL=I lässt sich im SASLog ausgeben, welche Indizes zur Verarbeitung der Abfrage herangezogen wurden. Hier ist schnell ersichtlich, dass ausschließlich der Index auf die Partnernummer genutzt wurde.

Dies liegt daran, dass SAS in einem SELECT zur Optimierung nur einen einzelnen Index nutzt. Bei allen weiteren Tabellenverknüpfungen werden die Indizes ignoriert. Um zu einem performanten Ergebnis zu gelangen, muss die Abfrage wie folgt umgestellt werden:

```
PROC SQL;
  CREATE TABLE WORK.ERG1 AS
    SELECT P.NAME,
           P.PRT_NR
    FROM WORK.PARTNER P
    WHERE P.prt_nr < 100;
```

```
  CREATE TABLE WORK.ERG2 AS
    SELECT P.NAME,
           V.VTR_NR
    FROM WORK.ERG1 P,
         WORK.VERTRAG V
    WHERE P.PRT_NR = V.PRT_NR;

  CREATE TABLE WORK.ERG3 AS
    SELECT P.NAME,
           K.VTR_NR,
           K.KONTO
    FROM WORK.ERG2 P,
         WORK.KONTO K
    WHERE P.VTR_NR = K.VTR_NR;

  CREATE TABLE WORK.SUMME_PRT_KLEINER_100_NEU AS
    SELECT P.NAME,
           SUM(U.BETRAG) AS SUM_BETRAG
    FROM WORK.ERG3 P,
         WORK.UMSATZ U
    WHERE P.VTR_NR = U.VTR_NR
      AND P.KONTO = U.KONTO
    GROUP BY P.NAME;
QUIT;

INFO: Index prt_nr für Optimierung der Where-Bedingung ausgewählt.
NOTE: Table WORK.ERG1 created, with 105 rows and 2 columns.

INFO: Index prt_nr of SQL table WORK.VERTRAG (alias = V) selected
      for SQL WHERE clause (join) optimization.
NOTE: Table WORK.ERG2 created, with 1006 rows and 2 columns.

INFO: Index vtr_nr of SQL table WORK.KONTO (alias = K) selected
      for SQL WHERE clause (join) optimization.
NOTE: Table WORK.ERG3 created, with 2109 rows and 3 columns.

INFO: Index kto of SQL table WORK.UMSATZ (alias = U) selected
      for SQL WHERE clause (join) optimization.
NOTE: SAS threaded sort was used.
NOTE: Table WORK.SUMME_PRT_KLEINER_100_NEU created,
      with 61 rows and 2 columns.

NOTE: PROZEDUR SQL used (Total process time):
      real time           1.99 seconds
      cpu time            0.31 seconds
```

Wie man sieht, wurden die gleichen Datensätze selektiert. Nur die Dauer der Ausführung konnte von 50 Sekunden auf 2 Sekunden reduziert werden. Führt man derartige Abfragen auf SAS unter z/OS aus, wo i.d.R. auch die Prozessorzeit berechnet wird, sind die Einsparungen noch drastischer. Bei der CPU Zeit konnte diese von 84 Sekunden auf 0,3 Sekunden reduziert werden.

Da im vorgestellten Testszenario Tabellen ausschließlich mit relevanten Schlüsselspalten genutzt wurden, ist der Performance Gewinn durchaus überschaubar. In realen Da-

tensituationen, bei denen neben den Schlüsselspalten noch eine ganze Menge weiterer Spalten in den Tabellen enthalten sind, kann eine Performance-Verbesserung durchaus von deutlich über 20 Minuten auf unter 10 Sekunden erreicht werden.

3 Indexnutzung beim Merge, Update oder Modify

Der DataStep Merge/Update/Modify wird häufig genutzt, um Datensätze in einer Tabelle zu aktualisieren oder zu ergänzen. Wir alle wissen, dass hierfür die Tabellen, die auf diese Weise zusammengeführt werden sollen, nach den Schlüsselvariablen sortiert sein müssen.

Es besteht jedoch auch die Möglichkeit, die Tabellen nicht zu sortieren und stattdessen einen Index in den Tabellen zu verwenden. Die Auswirkungen dieses Index (gerade bei großen Tabellen) sollen im Folgenden exemplarisch untersucht werden.

Für das Beispielszenario nutzen wir eine große Tabelle mit ca. 6.000.000 Datensätzen. Jeder Datensatz besteht aus einer Vertragsnummer und einem Wert. Weiterhin nutzen wir eine kleine Update-Tabelle, die zu ca. 10.000 Datensätzen neue Werte beinhaltet. Ziel ist es, in der großen Tabelle die Datensätze zu aktualisieren.

Aus verschiedensten Gründen kann es nun erforderlich sein, dass die beiden genutzten Tabellen nicht nach der Schlüsselvariablen sortiert werden sollen. Dies kann bspw. darin begründet sein, dass in der Folgeverarbeitung eine andere Sortierung vorausgesetzt wird und aus diesem Grund auf den zusätzlichen I/O durch die Sortierung verzichtet werden soll.

Um die Daten dennoch aktualisieren zu können, ist es somit erforderlich, auf den beiden Tabellen einen Index zu definieren. Die Indexerstellung ist im Beispielszenario sehr leicht möglich und benötigt auch nicht viel Zeit:

```
proc datasets lib=work;
  modify big;
  index create vtr_nr;
  run;
  modify small;
  index create vtr_nr;
  run;
quit;

NOTE: PROZEDUR DATASETS used (Total process time):
      real time           2.76 seconds
      cpu time            5.21 seconds
```

Auch die Update-Anweisung ist schnell geschrieben. Mit wenigen Zeilen Code werden in der großen Tabelle die Werte anhand der Vertragsnummern der kleinen Tabelle aktualisiert.

```
data big;
  update big small;
  by vtr_nr;
run;

INFO: Index vtr_nr Verarbeitung der BY-Bedingung ausgewählt.
INFO: Index vtr_nr Verarbeitung der BY-Bedingung ausgewählt.
NOTE: There were 6323657 observations read from the data set WORK.BIG.
NOTE: There were 9994 observations read from the data set WORK.SMALL.

NOTE: The data set WORK.BIG has 6323657 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           2:39.30
      cpu time            2:37.03
```

Trotz der vorhandenen Indizes liegt die Verarbeitungszeit mit über 2 Minuten deutlich über den Erwartungen. Betrachtet man die große Tabelle zusätzlich etwas genauer, muss man zwei interessante Beobachtungen machen:

1. Nach der Verarbeitung ist die Tabelle nach Vertragsnummer aufsteigend sortiert obwohl nirgends eine SORT-Anweisung explizit im Code steht.
2. Der Index der großen Tabelle ist entfallen.

Diese Beobachtungen widersprechen deutlich der Eingangsüberlegung, dass auf einen SORT verzichtet wird, um den zusätzlichen I/O der Sortierung zu umgehen. Nach der Aktualisierung ist die Tabelle nun doch anders sortiert und muss wieder umsortiert werden. Verzichtet man hingegen bei der gleichen Datenkonstellation auf die Erzeugung der Indizes und sortiert die Tabellen nach den Schlüsselvariablen, so macht man die nächste interessante Beobachtung:

```
proc sort data=big;
 by vtr_nr;
run;

NOTE: There were 6319856 observations read from the data set WORK.BIG.
NOTE: SAS threaded sort was used.

NOTE: The data set WORK.BIG has 6319856 observations and 2 variables.
NOTE: PROZEDUR SORT used (Total process time):
      real time           4.94 seconds
      cpu time            7.73 seconds

proc sort data=small;
  by vtr_nr;
run;
```

```
NOTE: There were 9994 observations read from the data set WORK.SMALL.
NOTE: SAS sort was used.

NOTE: The data set WORK.SMALL has 9994 observations and 2 variables.
NOTE: PROZEDUR SORT used (Total process time):
      real time           0.00 seconds
      cpu time            0.01 seconds

data big;
  update big small;
  by vtr_nr;
run;

NOTE: There were 6319856 observations read from the data set WORK.BIG.
NOTE: There were 9994 observations read from the data set WORK.SMALL.

NOTE: The data set WORK.BIG has 6323657 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.88 seconds
      cpu time            0.87 seconds
```

Die gesamte Verarbeitung inkl. einer zusätzlichen Sortierung und der Aktualisierung dauert nicht einmal 10 Sekunden. Trotz des zusätzlichen I/Os ist dieser Weg also der Indexnutzung vorzuziehen.

Es bleibt jedoch die Frage, wie sich das eingangs beschriebene Szenario noch lösen lässt, ohne eine Umsortierung der Daten vorzunehmen. Hier bietet sich der Einsatz eines Hash-Objektes an, mit dem auch unsortierte Daten aktualisiert werden können:

```
data big;
  set big;
  if _n_=1 then do;
    declare hash h(dataset:"small");
     h.definekey("vtr_nr");
     h.definedata("Wert");
     h.definedone();
  end;
  rc=h.find();
  drop rc;
run;
```

Über einen klassischen DataStep wird die große Tabelle eingelesen und sequentiell verarbeitet. Im ersten Datensatz wird das Hash-Objekt definiert, welches die kleine Tabelle in den Hauptspeicher lädt und gleichzeitig die Vertragsnummer als Schlüsselvariable und den Wert als Nachlesevariable definiert.

Bei dieser Variante ist zu berücksichtigen, dass die Hash-Tabelle in den Hauptspeicher passen muss. In diesem Fall werden 2 numerische Variablen à 8 Byte für ca. 10.000 Datensätze in den Hauptspeicher gelesen. Es ist somit eine Hauptspeichernutzung von 160 KByte zu erwarten, was sicherlich zu keinem Problem führen wird. Bei größeren

Tabellen oder vielen, teils alphanumerischen Nachlesefeldern ist an dieser Stelle sicherlich etwas genauer zu schauen.

Das Ergebnis des DataSteps ist überzeugend:

```
NOTE: There were 9994 observations read from the data set WORK.SMALL.
NOTE: There were 6323657 observations read from the data set WORK.BIG.

NOTE: The data set WORK.BIG has 6323657 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           1.51 seconds
      cpu time            1.48 seconds
```

Ohne die Sortierung der Daten zu ändern wurde die Aktualisierung der Daten vorgenommen. Dabei wurde nochmals deutlich weniger Zeit benötigt, als beim Update der sortierten Daten.

4 Fazit

Die zwei Beispiele haben gezeigt, dass ein Index in SAS Programmen hilfreich sein kann. Es kommt, wie immer, auf die richtige und gezielte Verwendung an. Alternative Herangehensweisen sollte man jedoch nicht außer Betracht lassen, da auch diese häufig zu einem guten und teilweise auch überraschenden Ergebnis führen können.

Literatur

[1] Datenbankindex: http://de.wikipedia.org/wiki/Datenbankindex (09.03.2015)