SQL – C.J. Dates Empfehlungen

In seinem Buch „SQL and Relational Theory: How to Write Accurate SQL Code“, erschienen 2009 bei O'Reilly, setzt sich C.J. Date mit SQL auseinander. Kurz gesagt: es ist beklagenswert, wie weit entfernt SQL von einer Sprache ist, die man zurecht *relational* nennen könnte. Viele Probleme mit Datenbankmanagementsystemen gibt es nur deshalb, weil sie das relationale Datenmodell *nicht* implementieren.

Date wird im Internet schon mal gerne nachgesagt, er sei der „Gralshüter“ des relationalen Modells und ereifere sich unnötig und etwas übertrieben. Nichts könnte verkehrter sein. In seinem Buch argumentiert Date fundiert und beweist jede seiner Aussagen!

Das Buch enthält immer wieder Empfehlungen, wie man SQL verwenden sollte und wie man SQL *nicht* verwenden sollte. Folgender Text gibt diese Empfehlungen wieder und kommentiert sie aus meiner Sicht. Die Seitenangaben beziehen sich auf Dates Buch.

(Manche der Empfehlungen wird vielleicht in der Kürze in diesem Papier nicht unmittelbar klar – eine Aufforderung in Dates Buch nachzulesen. Dort wird jede Empfehlung ausführlich begründet und oft durch klare Beispiele illustriert.)

1. Man kann (zum Beispiel in JDBC) ein SQL-System prozedural verwenden, in dem man Datensätze liest, vergleicht und so fort. Man verwendet dann SQL, um in der Datenbank zu navigieren. SQL (bzw. das relationale Modell) wurde eigens erfunden, damit man dies nicht tun muss – man sagt *was* man wissen will, nicht *wie* es gefunden werden soll. Solch prozedurales Vorgehen widerspricht dem Prinzip der *Datenunabhängigkeit*. [S. 15]

2. Physische Datenunabhängigkeit bezeichnet die Immunität von Anwendungen gegenüber Änderungen an der Implementierung des Datenbankmanagementsystems. Daraus folgt, dass man keine Eingenschaften von Datenbankmanagementsystemen einsetzen sollte, für die man Kenntnis von physischen Zugriffspfaden oder anderer Implementierungsdetails braucht. Da der SQL-Standard in diesem Punkt dem Konzept des relationalen Modells entspricht, kann man diese Empfehlung dadurch einhalten, indem man sich an den Standard hält. [S. 16]

3. Eine Relation ist eine Menge von Tupeln von Werten gleichartiger Struktur zusammen mit dem Relationskopf. Eine Relation kann auftreten als Basisrelation, die gespeichert ist, als View oder auch

als Ergebnis einer Abfrage. Im SQL-Jargon steht Tabelle für Relation, wird aber oft so verstanden, dass es sich um eine Basistabelle handelt, so als ob eine View *keine* Tabelle wäre. Date empfiehlt bei Relation oder Tabelle nicht automatisch an Basisrelation oder Basistabelle zu denken. [S. 21]

4. SQL macht automatisch Typumwandlungen (Coercion), die zu ungewollten Ergebnissen führen können. Man vermeide deshalb solche Umwandlungen. Insbesondere sollten Spalten mit demselben Namen stets denselben Typ haben. [S. 45]

5. Der Vergleich von Strings wird in SQL durch sogenannte *Collations* bestimmt. Dabei wird auch festgelegt, wie Strings verglichen werden, die sich nur durch zusätzliche Leerzeichen unterscheiden: Gibt man der Collation die Eigenschaft `PAD SPACE`, dann werden z.B. die beiden String „AB“ und „AB␣“ dann als gleich betrachtet. Date empfiehlt niemals die Option `PAD SPACE` zu verwenden, sondern immer `NO PAD`. [S. 46]

6. Im SQL-Standard gibt es Stellen, in denen von nichtdeterministischen Ergebnissen von Ausdrücken die Rede ist. Damit ist gemeint, dass eine Wiederholung der Auswertung des Ausdrucks unter identischen Vorbedingungen möglicherweise nicht wieder dasselbe Ergebnis zur Folge hat. Solche Ausdrücke sollte man vermeiden. [S. 47]

7. SQL kennt eine Art Pointer, der im Standard *reference value* genannt wird und die Adresse einer Zeile in einer Tabelle repräsentiert. Dieses Konzept widerspricht dem relationalen Modell grundlegend. Denn im relationalen Modell bestehen Beziehungen zwischen Tupeln in Relationen ausschließlich durch die Übereinstimmung von Werten von Attributen. Deshalb sollte man SQL *reference values* grundsätzlich nicht verwenden. [S. 48]

8. Der Ausdruck „null-Wert“ ist irreführend, weil es sich ja in SQL gerade *nicht* um einen *Wert* handelt. Man spreche also stattdessen stets von „null“. [S. 56]

9. In SQL kann man Tupel vergleichen, dieses geht auch mit den Vergleichsoperatoren `<`, `>` usw. Solche Vergleiche werden lexikographisch durchgeführt, hängen also von der Reihenfolge der Werte in den Tupeln ab – und ihr Ergebnis kann verwirrend sein. Deshalb verwende man Tupelvergleiche nur mit den Vergleichsoperatoren `=` und `<>`. [S. 59]

10. In SQL ist es möglich, Tabellen als Ergebnis einer Abfrage zu erzeugen, in denen Spalten keinen Namen haben. Ein Beispiel ist etwa:

```
select sum(Preis)
  from RechnungsPosition where RNr = 123456
```

In dieser Situation sollte man stets einen Namen vergeben und außerdem darauf achten, dass niemals zwei Spalten einer Tabelle denselben Namen haben. In unserem Beispiel:

```
select sum(Preis) as "Gesamtpreis"
  from RechnungsPosition where RNr = 123456
```

[S. 66]

11. Es gibt relationale Operatoren wie „natural join“ oder „union“, bei denen die Namen der Attribute der Relation eine wesentliche Rolle spielen. Deshalb sollte man stets zwei Spalten in SQL, die „dieselbe Art von Information“ repräsentieren auch denselben Namen geben, wenn dies möglich ist. [S. 67]

12. In SQL gibt es viele Ausdrücke, die von der Reihenfolge von Spalten abhängen, wie z.B.

    ```
    select * from Artikel
    insert into Artikel values( 200001, 'Pinot noir', ...)
    ```

    Man sollte niemals solche Ausdrücke verwenden, sie verletzen das Prinzip der Datenunabhängigkeit. Richtig ist stattdessen:

    ```
    select ArtNr, Bez, Weingut, ... from Artikel
    insert into Artikel( ArtNr, Bez, ... )
      values( 200001, 'Pinot noir', ...)
    ```

    [S. 68 und S. 93]

13. Das relationale Modell sieht Relationen als *Mengen* von Tupeln, es kann also keine Duplikate geben. In SQL ist dies nicht so, eine Tabelle in SQL ist eine *Multimenge* von Zeilen. Manche Operatoren in SQL führen automatisch eine Duplikatelimination durch, z.B. „union“, andere nicht, z.B. eine Projektion. Also muss man (1) wissen, wie SQL mit Duplikaten verfährt, (2) achtgeben, ob Duplikate in der eigenen Anwendung eine Rolle spielen und (3) `distinct` verwenden, um unerwünschte Duplikate zu eliminieren. Und `all` sollte niemals vorkommen! [S. 78]

14. Der Umgang mit „null“ in SQL ist delikat und kann zu unerwarteten Effekten führen. Deshalb sollte man „null“ möglichst vermeiden. Dies bedeutet:
    - In der Definition von Basistabellen sollte man stets `not null` angeben. Dies bedeutet insbesondere, den Entwurf des Datrenbankschemas zu überdenken, wenn man denkt genötigt zu sein „null“ zu erlauben.

- Man vermeide „outer joins“.
- Man verwende `is true` usw. nicht, denn wenn „nulls“ im Spiel sind, erhält man erstaunliche Ergebnisse.
- Man verwende die Funktion `coalesce`, um definiert mit „nulls“ umzugehen.
[S. 83]

15. SQL ermöglicht die Modifikation von Daten durch einen Cursor, diese Technik nennt man „positionierte Updates“. Diese Möglichkeit sollte man nicht verwenden, sondern stets die Anweisungen, die in der `where`-Klausel die zu ändernden Zeilen spezifizieren. [S. 91]

16. Ein Schlüssel heißt irreduzibel, wenn jede Teilmenge der Schlüsselattribute nicht mehr nur eindeutige Werte erlaubt. Man sollte nie einen Primärschlüssel oder die Einschränkung `unique` verwenden, wenn man weiß, das der dadurch definierte Schlüssel nicht irreduzibel ist. [S. 95]

17. In Basistabellen sollte man stets einen Primärschlüssel oder wenigstens eine Einschränkung `unique` spezifizieren, um sicherzustellen, dass jede Basistabelle einen Schlüssel hat. [S. 96]

18. Wenn möglich sollte man in Fremdschlüsselbeziehungen die beteiligten Spalten gleich nennen. [S. 99]

19. Jede relationale Operation ist eine Mengenoperation. Deshalb sollten auch Trigger an ein komplettes Statement und nicht an die Veränderung einzelner Zeilen gebunden werden. Deshalb vermeide man bei der Definition von Triggern die Klausel `for each row`. [S. 100]

20. Algebren und ihre Operatoren haben die Eigenschaft der *Abgeschlossenheit*: Ergebnis eines Operators auf Elementen der Algebra ist wieder ein Element der Algebra. Im relationalen Modell ist dies mit den Relationen und den relationalen Operatoren ebenso. In SQL kann man leicht Ergebnisse produzieren, die nicht wirklich wieder eine Tabelle (im engeren Sinne) ist: man kann etwa leicht unbenannte Spalten haben oder Duplikate in den Spaltennamen. Durch etwas Disziplin in der Verwendung von SQL kann man diese Merkwürdigkeiten in vielen Fällen vermeiden. Man sollte also bestrebt sein, SQL so zu verwenden, dass „aus Tabellen wieder Tabellen und nichts als Tabellen“ entstehen. [S. 110]

21. Der vorige Punkt bedeutet insbesondere, dass man stets die Benennung von Spalten einer Ergebnistabelle mit dem Schlüsselwort `as` verwenden sollte. [S. 112]

22. Für die Formulierung des Verbunds empfiehlt Date (1) die Verwendung von *natural join* und (2) die Vermeidung von *join ... on*. Er weist dabei natürlich auf die dann notwendige Sorgfalt mit Spaltennamen hin. Ich kann der Empfehlung (1) nicht folgen, weil es leicht vorkommen kann, dass eine Basistabelle um Spalten ergänzt wird (unabhängig von unserer Anwendung und seiner SQL-Abfrage) und dadurch möglicherweise ein anderes Resultat beim natürlichen Verbund entsteht. Dagegen hilft nur, die gewünschten Ergebnisspalten *explizit* anzugeben und die korrespondierenden Spalten für den Verbund ebenfalls *explizit* zu nennen. [S. 118]

    Bemerkung: Date setzt sich mit meiner Überlegung auf S. 128 auseinander und skizziert ein Konzept, das Datenunabhängigkeit mitsamt dem *natural join* gestattet. SQL unterstützt dieses Konzept jedoch nicht.

23. Bei der Formulierung einer *union* sollte man die korrespondierenden Spalten in derselben Reihenfolge angeben oder aber das Schlüsselwort `corresponding` verwenden. [PostgreSQL hat `union corresponding` nicht implementiert.] Die Vereinigung *union* führt in SQL zu einer Ergebnismenge (also ohne eventuelle Duplikate). In der Dokumentation von PostgreSQL wird (genau entgegengesetzt zur Auffassung von Date) empfohlen `union all` zu verwenden, wodurch Duplikate im Ergebnis der Vereinigung vorkommen, sofern die vereinigten Mengen identische Zeilen enthielten. Als Grund wird bei PostgreSQL die Performance angegeben. Date hat freilich recht, dass dies kein sonderlich gutes Argument ist: Performance ist eine Implementierungfrage und sollte im Geiste der Datenunabhängigkeit *keinen* Einfluss auf das Konzept der Datenbank*sprache* haben. [S. 119]

24. Bei der Verwendung von `group by` und `having` muss man achtgeben, dass man die wirklich gewünschte Tabelle für die Gruppierung verwendet. Unerwartete Ergebnisse können auftreten, wenn `null`s im Spiel sind, also sollte man `coalesce` einsetzen. [S. 150]

25. Man sollte bei der Definition von Tabellen stets bekannte Integritätsbedingungen, wie Beschränkungen von Wertebereichen, Eindeutigkeit und so fort angeben. [S. 166]

26. In SQL kann man Transaktionen so einstellen, dass *innerhalb* der Transaktion Integritätsbedingungen *nicht* erfüllt sein müssen. Das ist manchmal in der Tat notwendig, weil SQL nicht die Möglichkeit hat, mehrere Zuweisungen in einer Anweisung zu machen. Wenn dies notwendig ist, muss der Verwender sicherstellen, dass die Konsistenz der Datenbank gewahrt bleibt und keine ungewollten Seiteneffekte entstehen. [S. 177]

27. Views sind „virtuelle“ Tabellen und sollten sich aus der externen Sicht so verhalten, dass der Anwender gar nicht weiß, ob er eine Basistabelle oder eine View verwendet. Deshalb sollte man in SQL alle Views, deren Daten via View modifizierbar sind mit der Klausel `with check option` versehen, die eben dies spezifiziert. [PostgreSQL kennt diese Option nicht.] Außerdem soll man bei der Definition der View durch die Select-Anweisung die Spalten mit ordentlichen Namen versehen. [S. 187]

28. Aus der (relationalen) Sicht, in der Views Tabellen sind, sollte es für Views natürlich auch Integritätsbedingungen geben, wie etwa die Eindeutigkeit von Werten o.ä. In SQL ist dies nicht möglich, Date empfiehlt deshalb wenigstens als Kommentar die Integritätsbedingung anzugeben. [S. 193]

29. Es wird oft argumentiert, dass es doch ganz offensichtlich sei, dass manche Views, wie z.B. solche, die durch einen Verbund zustandekommen, nicht modifizierbar sind. Schaut man mit Date genauer hin, merkt man, dass dieses Argument nicht wirklich überzeugt. Auch die Veränderung an Basistabellen kann scheitern, wenn z.B. eine Integritätsbedingung wie eine Fremdschlüsselbeziehung nicht erfüllt ist. Genauso könnte man von einem relationalen System durchaus verlangen, dass es aus der Definition einer View alle Integritätsbedingungen ableitet und überprüft.

    Wenn die View sich nur auf eine Basistabelle bezieht, kann SQL solche Views modifizieren, es entsteht aber durch ein Insert eventuell ein „Phantom“, weil SQL nur die Integritätsbedingungen der Basistabelle überprüft, nicht aber die Definition der View. Wenn die View etwa alle Artikel mit Preis über 10 € zeigt und man fügt einen neuen Artikel zu 5 € ein, dann geht das, ist aber durch die View nicht sichtbar – merkwürdig! Dies kann man in SQL vermeiden, wenn man die Klausel `with cascaded check option` angibt. [`cascaded` ist Default, d.h. `with check option` genügt.] [S. 197]

30. Zitat: „Lobby the SQL vendors to improve their support for view updating as soon as possible.“ [S. 198]

31. In manchen Situation, z.B. beim Data Warehousing, kann man Schnappschüsse (*snapshot*s) brauchen: Kopien von Datein erzeugt durch Select-Anweisungen. In der Literatur wird so was auch materialisierte View (*materialized view*) genannt. Dies ist jedoch eine Contradictio in adjecto, denn Views sind per Definition *virtuell*, beziehen sich also stets auf die Basistabellen, die ihrer Definition zugrundeliegen. Deshalb verwechsle man nie Views mit *snapshots* und benutze den Ausdruck *materialized view* gar nicht! [S. 201]

32. Man kann in SQL Vergleiche machen, bei den eine Subanweisung mit `all` oder `any` verglichen wird. Etwa:

```
select distinct A1.Bez from Artikel as A1
  where A1.Preis < all ( select A2.Preis
                         from Artikel as A2
                         where A2.Weingut='Cave Bellenda' )
```

    Dabei ergibt der Vergleich mit `all` true, genau dann, wenn der Vergleich für jedes der Ergebnisse der Subanweisung true ergibt. Der Vergleich mit `any` ergibt genau dann true, wenn er für ein Ergebnis der Subanweisung true ergibt. Date empfiehlt, `all` und `any` zu vermeiden, weil umgangssprachlich „any“ gerade da verwendet wird, wo im Vergleich `all` stehen muss – also fehleranfällig. [S. 250]

33. Wiederholung: Man vermeide `select * ...`, es sei denn in einer Anweisung mit `exists`, denn da spielt ja die Struktur des Ergebnisses keine Rolle. [S. 256]

34. Es gibt zwei Formen, mit Relationen zu arbeiten: die relationale Algebra und das relationale Kalkül. Date widmet letzterem ein ganzes Kapitel in seinem Buch. Kurz gesagt ist das relationale Kalkül die Anwendung der Prädikatenlogik auf Datenbanken. Es ist in seiner Ausdruckskraft der relationalen Algebra gleichmächtig. SQL mischt Elemente der relationalen Algebra mit denen des relationalen Kalküls. Deshalb hat man in SQL so viele Möglichkeiten ein und dieselbe Frage unterschiedlich zu formulieren. Im relationalen Kalkül kommen Variablen (*range variables*) vor, die für ein Tupel einer Relation stehen. So ist in

```
select X.Preis
  from Artikel as X
  where X.Weingut = 'Louis Max'
```

    X eine Variable, die alle Tupel von Artikel „durchläuft“.

    In manchen Büchern zu SQL wird eine solche Variable auch als „Alias“ bezeichnet, so als ob sie einfach ein alternativer Name für die Tabelle wäre. Diese Denkweise verkennt den (oben skizzierten) Hintergrund für Variablen komplett! Date empfiehlt, explizit solche Variablen einzuführen, wenn man komplexere Ausdrücke hat. [S. 259]

Burkhardt Renz
Technische Hochschule Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 2.0 – 3. März 2011