Pearls of Database Literature

Discussion Protocol

A Relational Model of Data for Large Shared Data Banks
(by E. F. Codd)

Daniel Kocher, Reg. No. 0926293

dkocher@cosy.sbg.ac.at

Salzburg, April 9, 2015

1

Question:

What is the problem that Codd tries to solve? What was the state of data base system up to that point and why was that a problem? (*by Alexander*)

Answer:

General problem: the user had to know too much about how the data is represented in the underlying data base system. This was also observable in the applications because the applications needed to know how the data is represented to get access to it (and how to store it back into the data base when a modification was done).

Codd proposed three kinds of data dependencies:

- Ordering Dependence:

  In which ordering is the data physically stored? The problem hereby: Imagine the data was stored ordered by a certain criteria, e.g. the primary key. While the data was stored using this ordering, all the application programmers relied on this fact and wrote their applications to utilize this ordering. On the one hand, this led to better performance (in most cases), but on the other hand, all those applications where only operating correctly as long as the data ordering was kept consistently. If the data was reordered, the applications did not work anymore.

- Indexing Dependence:

  This dependence is somehow related to the Ordering Dependence but on another level, namely the level of indices. An application relying on an index had to be changed if the used index was changed in the data base. Otherwise, the application did not operate correctly. Furthermore, the application relies on a correct structure of the index information to be present. This can lead to further problems, if it is not the case. Codd lists some data systems in his paper and explains if applications remain invariant as indices come and go:

  - TDMS: no problem (unconditionally provided indices)
  - IDS: problem (indexing chains; applications must refer to those by name)

- Access Path Dependence:

  In hierarchical data base systems, the programmers needed to know how to get access the actual data. Therefore, the applications used certain access paths which were at least present when the application was written. Sometimes, a data base system needs a restructuring (and some access paths may be changed or deleted while this is done). If those access paths were deleted, the application was not able to operate correctly anymore, because it only knew the access paths provided by the programmer. Hence, an access path may only be deleted if and only if the last application using this access path, is no longer used.

Additional information:

Data bases from the time Codd wrote this paper were only used as a file system (or similar to a container in e.g. C++; it is taken care of the pointers and its internal organization). In contrast, the programmer needed to know exactly how to access the data.

2

Question:

Describe shortly the relational data model and it's concepts. (*by Vasker*)

Answer:

For a given number n of sets, a relation is a set of n-tuples where the first element is in the first set, the second element is in the second set, and so on. The j-th set is referred to as the j-th domain of the relation.

Primary key: used to uniquely identify a tuple in a relation and may be composed of multiple domains.

Foreign key: not used to uniquely identify a tuple in a relation but a reference to a primary key of another relation.

Simple domain: domain is atomic/nondecomposable, e.g. a first name.

Nonsimple domain: domain consists of another relation (decomposable), e.g. a whole name, which can be decomposed into at least first and last name. These domains would require more complex data structures.

Codd's proposal: Decompose all nonsimple domains into simple domains (First Normal Form).

Domain-ordered: relations; field accessed by the number

Domain-unordered: relationships; field accessed by the name

The user should not be burdened with the ordering of the domain. Hence, Codd proposed to use relationships (the unordered counterparts of relations). Thus, domains must be uniquely identifiable without using the position, so Codd introduced a *role name* to identify the role of a domain in a given relation. All the user needs to know about a relationship is the name of the relationship and its domains (role qualified if necessary).

3

Question:

What are the two proposed features of a universal data sublanguge? (*by Vasker*)

Answer:

Sublanguage: Embedded into another programming language, not inventing an actual new programming language.

1. *First order predicate logic* (is able to describe every tuple sufficiently)
2. *Arithmetic operations*
3. *Insertions* (into set in any order), *Deletions*, *Updates*

Symmetric exploitation:

In hierarchical data base systems: Accessed by using a concatenation as primary key (not symmetric).

In the relational model: The only thing we need to know is the fact that a domain exists (not where exactly). All relations and domains have equal rights.

4

Question:

Which operations on relations are presented in this paper? (*by Stefan*)

Answer:

- *Projection* π: `SELECT * FROM ...;`
- *Permutation*: The ordering how the data is stored physically should not play a role. It is sufficient to store one ordering and to provide a permutation operation.
- *Joins* $\bowtie$: `SELECT * FROM ... WHERE A.ID = B.ID`
  There are multiple possible ways to join relations. *Natural join* as a special case: The rightmost domain of the left relation is joined with the leftmost domain of the right relation. It is assumed that a join partner is found in every case. Both properties are not met in the natural join nowadays (today's natural join is more general and you may lose tuples if no join partner is found).
- *Composition*: A join followed by a projection, $\pi\,(R \bowtie S)$.
- *Restriction* |: joinable attributes only, $(R\ {}_L|_M\ S) \bowtie (S\ {}_M|_L\ R)$.
  Nowadays: Semi-Join, $R \ltimes S$ (all attributes of R which have join partner in S.)

5

Question:

Show two relations are composable if and only if they are joinable. (*by Vasker*)

Answer:

Per definition: composition is a combination of a join and a projection. Hence, if two relations are joinable, they must be composable because a projection is always possible. If two relations are not joinable, the join before the projection cannot be done, thus the composition cannot be done and the relations are not composable.

6

Question:

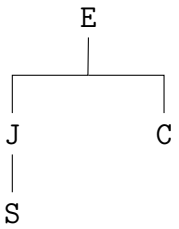
What is normalization (in the context of databases), how is it carried out and what are its advantages? (*by Stefan*)

Answer:

Normalization (in the context of the discussed paper) is done to eliminate nonsimple domains. We are allowed to eliminate them because they are always reconstructable using the join operator. This simplifies the physical data representation in many ways: No pointers are needed because we only use primitive data types. It is also possible without an index. Communication of bulk data: An array is simple to export because it is system-independent. In contrast, pointers in the data representation would be invalidated (they are system-dependent).

Example with four relations:

```
Employee E(man#, name, jobhistory, children)
Jobhistory J(jobdate, title, salaryhistory)
Salaryhistory S(salarydate, salary)
Children C(childname, birthyear)
```



The relations `J` and `C` are nonsimple domains of relation `E`. Relation `S` is a nonsimple domain of relation `J`.

Steps to normalized relations:

1. Add man# to each relation being a nonsimple domain of E (jobhistory and children) and remove nonsimple domains from E:

   ```
   Employee E(man#, name)
   Jobhistory J(man#, jobdate, title, salaryhistory)
   Salaryhistory S(salarydate, salary)
   Children C(man#, childname, birthyear)
   ```

2. Add man# and jodate to each relation being a nonsimple domain of J (salaryhistory) and remove nonsimple domains from J:

   ```
   Employee E(man#, name)
   Jobhistory J(man#, jobdate, title)
   Salaryhistory S(man#, jobdate, salarydate, salary)
   Children C(man#, childname, birthyear)
   ```

Prepending the primary key to the relation being a nonsimple domain is necessary because previously the hierarchy has resolved the references and we removed this hierarchy in order to get simple domains only.

7

Question:

What is the difference between strong and weak redundancy and how are they avoidable? (*by Stefan*)

Answer:

Strong redundancy:

- Can be explained using a formula
- Subsets, e.g. Management $\subseteq$ Name
- Redundancy can always be computed through a formula (deterministic)

Weak redundancy:

- Cannot be characterized by a formula and can be temporarily generated through data in the data base system (not eliminable)
- Reconstructable through Join and Projection
- Non-deterministic because it is not known which Join to use

8

Question:

What are the possible disadvantages of the proposed solution? (*by Alexander*)

Answer:

In contrast to

- Hierarchical data base systems:
  - Performance advantages in contrast to the relational model
  - High throughput for certain operations (because only pointer have to be dereferenced)
- Document-based data base systems:
  - Structure (schema) not necessary (the relational model requires restructuring if e.g. a schema is changed)

9

Question:

Codd's solution is widely adapted in modern DBS. Which parts are implemented, which have changed since, and which ones were never implemented? (*by Alexander*)

Answer:

- Implemented:
  - Normalization (even further than the First Normal Form)
  - Projection (in form of the `SELECT` statement in SQL)
  - Joins and Restrictions (which are Semi-Joins)
- Changed since:
  - the sublanguage (SQL can do much more things nowadays)
- Never implemented:
  - Composition

Notizen zur Diskussion des Papers: A relational model of data for large shared data banks, von E.F.Codd

Note Taker: Philipp Neulinger

25. März 2015

1 What is the problem that Codd tries to solve? What was the state of data base system up to that point and why was that a problem?

z.Dt.: Was ist das Problem das Codd zu Lösen versucht? Was war der damals aktuelle Stand der Technik und was war dabei ein Problem? (Frage von Alexander)

Grundsätzlich war das Problem der damals vorherrschenden Datenbanken, dass die User bzw. die Programmierer viel über die Anordnung und Organisation der Daten wissen mussten. Dabei musste unter anderem genau beachtet werden, wie diese Daten ausgelesen und gespeichert werden sollen.

Während der Diskussion wurde vor allem die verschiedenen Abhängigkeiten, auf die Codd einging, besprochen; diese waren:

- *Access Path Dependence:* Viele der damals wie heute bestehenden Produkte (IMS, TDMS) waren in Baum Strukturen organisiert. Dadurch war es für die Programme, die solche Datenbanken verwendet haben, notwendig sich der Struktur der Datenbank anzupassen. Dies führt zu dem wesentlichen Problem, das Änderungen an der Struktur einer Datenbank teuer sind. Hinzu kommt, dass in Systemen die ein "Network Model" verwenden, der User (bzw. sein Programm) muss die Zugriffswege auf die Daten irgendwie sammeln. Ein wesentliches Problem (neben wachsendem Platzverbrauch) ist dabei, dass diese Zugriffswege für die aktuellen Programme und Terminals immer present sein muss. Eine Lösung für dieses Problem wäre alle Zugriffswege zu löschen, sobald sich das letzte Programm bzw. Terminal abgemeldet hat.
- *Indexing Dependence:* Damit eine Programm einen bestehenden Index verwenden kann, muss dies im Programm festgelegt werden. Hinzu kommt, dass die Wartung der Indizes in hierarchischen Systemen teuer bzw. aufwendig ist. Diese optimieren in den hierarchischen Datenbanken zwar den Zugriff für Lesen und Updates, allerdings erschweren sie das Einfügen und das Löschen

von Daten. Ein weiteres Problem dabei ist, dass das Programm, sofern es einen Index verwendet, davon ausgeht, dass die Struktur korrekt ist.

- *Ordering Dependence:* Die Elemente, welche in einer Datenbank gespeichert werden, können dabei auf verschiedene Arten abgelegt werden, z.B. Ohne oder mit Rücksicht auf eine bestimmte (totale) Ordnung. Dabei wird oftmals ein Attribut zur Ordnung herangezogen, nach dem nun eine solche Tabelle in der Datenbank geordnet wird. Sollte sich nun das primäre Ordnungsattribut ändern, so ist die Neusortierung besonders teuer.

Um es zusammenzufassen, die wesentlichsten Probleme die während der Diksussion deutlich wurden:

- User/Programmierer muss viele Informationen über die zugrunde liegende Datenbank beachten
- Wartung und Änderungen können sehr teuer werden
- War vom Prinzip her einem Filesystem sehr ähnlich

2 Describe shortly the relational data model and it's concepts.

z.Dt.: Beschreiben sie kurz das Relationale Datenbanlmodell und dessen Konzepte? (Frage von Vasker)

Codd hat eine Relation als eine Teilmenge von mehreren Sets bzw. Domains betrachtet. Dabei stellt er das Konzept des zusammengesetzten Primary Keys und des Foreign Keys (ein Primary Key einer Tabelle der sogleich in einer anderen gespeichert wird) vor.

- A Simple Domain: Eine Domain ist bspw. der Preis eines Produkts, da es eine einfache Zahl ist.
- A non simple Domain: Eine Relation innerhalb einer Domain

Domain Ordering und Unordering: Bezieht sich vor allem auf die Ordnung der Attribute und deren Zugriff in weiterem Sinne durch Relations bzw. Relationships. Eine Relation ist Domain ordered, diese greift über die Nummer der Spalte auf die Information zu. Ein Rleationship ist Domain Unordered und greift dabei über den Namen des Attributs auf die Daten zu.
Ein wesentliches Problem hierbei ist die Eindeutigkeit: Sollten mehre Domains denselben Namen und eventuell auch dieselbe Domain haben, so wird durch eine Relation ihre Eindeutige Bedeutung festgelegt. Durch diese Relationen können mehre Datensätze aus verschiedenen Tabellen zusammengefasst bzw. ausgelesen werden können.

3 What are the two proposed features of a universal data sublanguge?

z.Dt.: Was sind die vorgeschlagenen Features einer universelen data sublanguage? (Frage von Vasker)

Table 1: Umgesetzte Operationen nach Codd

Π	Projektion	Select * ...
	Permutation	Select Col1, Col2, Col3, ...
$\bowtie$	Join	Select ... from ... where A.ID = B.ID

Die Data Sublanguage die Codd ebenso vorgeschlagen hat, ist eine eigene Sprache, die in eine Programmiersprache eingebettet werden kann. Es geht darum die Daten, die in einer Datenbank bzw. einer Tabelle o.Ä liegen, möglichst genau zu beschreiben, sodass ein User/ein Programm nur durch das "Beschreiben" der gewünschten Daten zu selbigen kommt.

Wesentlich ist hierbei der Begriff der Symmetrie und der Asymmetrie; klassische hierarchische Datenbanksysteme sind unsymmetrische Systeme, da viel Wissen über die Datenbank vorhanden sein muss, um darauf zugreifen bzw. selbige verwenden zu können, vorhanden sein müssen. Ein symmetrisches Datenbanksystem wäre das heutige SQL in Kombination mit einer Relationalen Datenbank, dabei muss man als Programmierer bzw. User nicht wissen, wo die entsprechenden Domains liegen, sondern beschreibt nur, dass man selbige haben will.

Neben den vorgeschlagenen Features, dass mit solch einer Sprache vor allem Insertions, Updates und Deletes möglich sein sollen, sollen auch Calculus und Arithmetische Funktionen in der Sprache enthalten sein. Dadurch können bestimmte Operationen vereinfacht werden, wie zum Beispiel: Die Summe aller Preise der Produkte eines Lieferanten soll so berechnet werden können.

Hinzu kommt, dass bei solchen Datenbanken nicht wichtig ist, wie die Daten physisch geordnet sind. Also dass bei einem Insert die Daten einfach wirklich nur in die Tabelle geschrieben werden, und über die Relationen bzw. die Sublanguage die Ordnung des Auslesens bestimmt werden. Genauso hat kein Delete zur Folge, dass die Daten neu sortiert werden müssen.

4 Codd's solution is widely adapted in modern DBS. Which parts are implemented, which have changed since, and which ones were never implemented?

z.Dt.: Codds Lösung ist weitestgehend in modernen DBS adaptiert worden. Welche Teile sind bereits implementiert, und welche wurden nie implementiert? (Frage von Alexander)

Bei der Permutation spielt es keine Rolle, in welcher Reihenfolge man die Spalten abfragt. Wenn zwei (oder mehrere) Tabellen eine Domain gemeinsam haben, besitzen diese ein Join Kriterium. Es gibt dabei natürlich mehr als nur diese eine Join Möglichkeit.

Der unterschied zur Relationalen Algebra ist, dass Codd in seiner Arbeit immer nur die letzte Spalte der einen Tabelle mit der ersten Spalte der zweiten Tabelle.

Joinable: In einem Natural Join können Tupel verloren gehen. Es muss nicht gegeben sein, dass in beiden Tabellen entsprechende Join Partner in der Domain vorhanden sind.

Compustion: Wenn man einen Join macht, kann man das Join Kriterium weglassen. Darstellung durch andere Operationen möglich.

$$\Pi(R \times S) \tag{1}$$

Restriction: Operator in der relationalen Algebra, auch als (Left oder Right) Semi Join $R \ltimes S$ oder $R \rtimes S$ bekannt. Dabei nimmt man nur die Attribute von einer Seite.

5 What is normalization (in the context of databases), how is it carried out and what are its advantages?

z.Dt.: Was ist Normalisierung (im Kontext von Datenbanken), wie wird es durchgeführt und was sind die Vorteile dieser Lösung? (Frage von Stefan)

Wesentlicher Vorteil sind non simple Domains können dadurch entfernt werden. Dadurch sind alle Domains simple. Die weiteren Vorteile die Codd nennt wären:

1. Man wäre frei von Pointern (adress-valued oder displacement-valued)
2. Es würde alle Abhängigkeiten hinsichtlich hash Adressierung vermeiden
3. Es würde keine Indices oder geordnete Listen enthalten

Des Weiteren wurde die Normalisierung durch Codd in der Diskussion einmalig durchgespielt, hier die Abbildung aus der Arbeit von Codd 1.

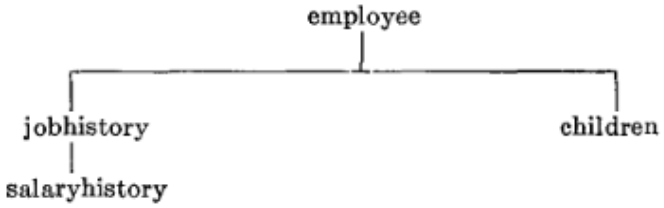
6 What are the possible disadvantages of the proposed solution?

z.Dt.: Was sind die möglichen Nachteile dieses Lösungsansatzes? (Frage von Alexander)

Im Vergleich zu hierarchischen Datenbanken sind die Zugriffszeiten schlechter, auch die Performance eines solchen Systems wie IMS wird bereits sehr hoch sein, da es bereits stark weiterentwickelt wurde. Ein weiterer Nachteil ist, dass wesentlich mehr Plattenzugriffe benötigt werden. Außerdem muss für das Locking von Tabellen eine andere Lösung gefunden werden.

7 What is the difference between strong and weak redundancy and how are they avoidable?

z.Dt.: Was ist der Unterschied zwischen starker und schwacher Redundanz und wie können sie vermieden werden? (Frage von Stefan)



employee (*man#*, name, birthdate, jobhistory, children)
jobhistory (*jobdate*, title, salaryhistory)
salaryhistory (*salarydate*, salary)
children (*childname*, birthyear)

FIG. 3(a). Unnormalized set

employee′ (*man#*, name, birthdate)
jobhistory′ (*man#*, *jobdate*, title)
salaryhistory′ (*man#*, *jobdate*, *salarydate*, salary)
children′ (*man#*, *childname*, birthyear)

FIG. 3(b). Normalized set

Figure 1: Beispiel der Normalisierung aus dem Paper von Codd

- Strong
  Eine starke Redundanz kann durch eine Teilmenge bzw. eine Formel oder Ähnliches dargestellt werden. Ein Beispiel hierfür wäre: Die Namen aller Manager die mit M anfangen - wäre eine Teilmenge der Name der Manager.
- Weak
  Kann auftreten, bezeichnet einen Zustand der temporär mehr Daten anzeigt als gewollt. Ein Beispiel wäre, wenn man einen Join auf zwei Tabellen ausführt, und sich die Keyfelder (welche ident wären) beide male ausgeben lassen würde. Man kann sich hierbei bei der Ausgabe für oder gegen die Anzeige der Duplikate entscheiden.

8 Which operations on relations are presented in this paper?

z.Dt.: Welche Operationen auf Relationen werden in der Arbeit präsentiert? (Frage von Stefan)

Noch nicht umgesetzt ist die Composition, bereits implementiert wurden die Normalisierung, die Projektion und die Joins.

9 Show that two relations are composable if and only if they are joinable.

z.Dt.: Zeigen Sie dass zwei Relations "composable" sind, genau dann wenn sie

joinable sind. (Frage von Vasker)

Join: Angenommen es gibt zwei Relationen R und S. R und S sind joinbar (ohne den Verlust von Informationen) gdw. es eine dritte Relation U gibt, sodass $\pi_{12}(U) = R$ und $\pi_{23}(U) = S$ ist. Eine andere Form wäre natürlich wenn R und S Relationen sind, sodass $\pi_2(R) = \pi_1(S)$.

Seien R und S zwei Relationen. Dass ist T nun eine Composition von R mit S gdw. ein Join U für R mit S existiert sodass $T = \pi_{13}(U)$. Zwei Relationen können also per Definition nur dann Composable sein, wenn sie joinbar sind.