**Codd Paper Notes**

**Section 1.2: Old approach**

Applications should not have to assume data are ordered (or unordered) in files. Neither should apps be required to know the structures of the files (e.g. record order in the files’ internal representation), but tree-structured and “network” data management systems impose this requirement. Changing the file structure means rewriting the application.

Indexes should be independent of internal file structures, but aren’t always.

**Section 1.3: Relational approach**

A mathematical structure, but also an abstraction to isolate apps from file structures and data formats.

Implies the need for a DBMS (abstractor) that provides data to apps in a consistent way regardless of changes to underlying file structures.

Column ordering

In this paper, relations can (but don’t always) have ordering to the domains (columns). Column ordering is necessary when more than one domain (column) in a relation (table) has the same name but distinct meanings in the relation.

Basic operations on n-tuples (rows) in a relation (table) of degree n (# of columns) are insert, delete, and update.

For users, it is easier to use a relationship model; relationships are relations with unordered domains (columns). The idea is that users (and by extension, applications) should only have to know the (meaningful) names of the columns to retrieve and manipulate the data—not column ordering and certainly not row ordering.

Codd anticipates as well something like Oracle’s “describe” command to show the columns names in case a user doesn’t know them in advance (subject to security and privacy controls).

Primary key

A primary key is any column or combination of columns in a table whose values are (collectively, if more than one column) unique and which can therefore be used to distinguish between different rows. Columns which are not necessary to establish uniqueness cannot be part of a nonredundant primary key (which is usually the kind you want). One column or combination of columns is then chosen as the primary key.

Foreign key

A foreign key of a relation R is a column (or columns) that is/are not the primary key, but its elements are values of the primary key in some relation S (with S=R being one possible case). The columns that constitute a foreign key do not have to be all of the columns of a multi-column primary key. Hence the values in a foreign key column are not necessarily unique within that column.

For example, if the primary key is three columns (meaning all three are required for uniqueness), any one or two of those columns (or a subset of the elements of those one of two columns) in the same or another table is a foreign key.

Nonatomic domains (nested tables)

It is possible that a domain (column) could consist of sub-columns which themselves form relations. I.e. nested tables.

**Section 1.4: Normal form**

One goal of normalization is to remove compound columns from tables so that each relation can be expressed as a simple (two dimensional) array with no “nested tables”; it’s simpler to store and transfer data this way.

To normalize: Put relations in a hierarchical tree, then expand it top to bottom so that no multipart columns exist (they become relations themselves) and each dependent relation’s primary key includes not only the natural primary key column for that relation but also its parents’ primary key columns.

**Section 1.5: Data interaction language**

Use formalized methods (predicate calculus) to derive a language that is capable of describing and modifying relations.

Codd anticipates the use of triggers (as in Oracle).

We want a language in which a user can express a query, update, or deletion using any number or combination of columns (any subset of columns in any order, by any name) as well as the name of the relation (table) rather than restricting the user to working with, say, just two columns at a time and then having to combine the results to get the set of columns for which they were really interested in seeing values.

The language is descriptive, not programmatic/computational.

A set retrieved by a query may then be modified for update in the database.

The language allows the user to obtain unknown column values by supplying known column values in any combination.

**Section 1.6: Named and expressible sets**

The “named set” of relations consists of those relations explicitly named by users.

The “expressible set” of relations consists of all possible relations that could be formed from the data in the database using expressions in the data language operating on the named set.

The database system must be able to retrieve data stored in a variety of particular representations (types?) in response to queries in a high-level language.

**Section 2.1.1: Permutations**

A permutation of a relation merely changes the order of the columns.

An n-ary relation has n! permutations possible.

**Section 2.1.2: Projection and selection**

A projection of a relation is obtained by striking out certain columns and then removing any duplicated rows in the result.

“Selection” performs permutation and/or projection operations. π12, for example, selects columns 1 and 2 from a relation.

**Section 2.1.3: Joins**

A “join” combines relations on columns.

The “natural join” of two binary (i.e. two-column) relations R and S is a ternary relation.

R\*S = {(a,b,c) : R(a,b) ∧ S(b,c)}

where R(a,b) is true if (a,b) is a member of R

and S(b,c) is true if (b,c) is a member of S.

Other joins are also possible which may not show all combinations of related values across the joined relations. These other joins exist when an element in the domain on which the join is occurring has multiple relatives (mappings) in both R and S. This element is called a “point of ambiguity,” as it may show relationships that aren’t real.

However, another independent relation that shows relationships between domains other than the one on which the join is made can be used to resolve the ambiguity. For example, an independent relation on domains a and c can resolve ambiguity where there are multiple mappings from elements in domain b to a in R and from elements in domain b to c in S.

Given R joinable with S and S with T, we can form a relation U that is a 3-way join between them. If R and T have a domain in common, we can form a cyclic 3-join between R, S, and T in which the “ends” are “tied” together (where Rab, Sbc, Tca, and Uabc and the domains are a, b, c).

If we use the natural join operator (\*) to form a cyclic 3-join, then we obtain the natural cyclic 3-join

γ(R\*S\*T) (the tie operator)

**Section 2.1.4: Composition**

We can form the composition of relations, similar to composition of functions. If U is a join of binary relations R and S such that π13(U)=T, then T is a composition of R and S.

The natural composition results if U is the natural join: R•S = π13(R\*S)

Composition maps one domain in one relation to another domain in another relation through an intermediate domain on which the two relations can be joined. a→c where R = Rab and S = Sbc.

**Section 2.1.5: Restriction**

The restriction operation forms a subset (R′, which itself is a relation) of a relation R by requiring the elements of a chosen list of domains in that subset to match the elements of a corresponding set of domains in another relation S. S thus restricts, or limits, what mappings will exist in R′ in the common list of domains (where R may have additional mappings in those domains).

**Section 2.2 Derivability**

In order to derive relations from other relations, we need the following operations: projection, natural join, tie, and restriction.

**Section 2.2.1 Strong redundancy**

Strong redundancy between relations in a database (or set of relations) exists when at least one relation has a projection that is derivable from other projections of relations in the database for all times. We may wish to store such redundant projections anyway, as having them already available on disk (rather than having to be computed on demand) can speed up query processing.

**Section 2.2.2: Weak redundancy**

A binary relation is complex if neither it nor its converse is a function. The joining of complex relations causes points of ambiguity to arise, resulting in multiple joins being possible, which in turn may mean that some relations may not be derivable from others using projection, natural join, tie, and restriction, even when those relations collectively share common domains. Yet if some of the relations can be obtained through non-natural joins of others, then that indicates that there is still some weak redundancy in the system. This weak redundancy may be useful to clarify real relationships between the data.

**Section 2.3: Consistency**

Given any collection C of time-varying relations and a set Z of logical constraints on the data, we can state whether a snapshot in time V of C satisfies all constraints Z. If so, then we may say that the state of the data is consistent. Constraints and a need to enforce consistency may be necessary where either strong or weak redundancy exists in a database. Otherwise, conflicts or ambiguities in the data relationships may arise, as for example when a user updates a relation without making the corresponding update to a second relation derived in part from the first, or that logically depends in part on the first.

Consistency (checking against constraints) is something that may be checked in real time with every insertion, deletion, or update, or only at specified times via, e.g., a daily batch job.