Lecture 3: The Relational Model

Database and SQL: Course Notes | Brendan Shea, Ph.D. (Brendan.Shea@rctc.edu)

1 CONTENTS

2 RULES ABOUT RELATIONS

In this lesson, we'll be answering the following questions

1. What exactly are a **relation** and the **relational database model?**
2. What are the different types of **keys,** and why are keys important?
3. What are **indexes,** and why are they useful?
4. What are **Codd's 12 rules** for relational databases?

In the early 1970s, IBM's Edgar Codd published a paper called "Relational completeness of data base sublanguages" , which eventually would change how the world stores and accesses data. Codd's paper used ideas from set theory and first-order logic to describe the structure of what we know call the **relational database** model. Codd then showed how we could use **relational algebra** (the basis of SQL) to interact with this data. Compared to the file-system data storage model, the relational model presented a "logical" view of data. Those who interacted with the database didn't need to know anything about how the data was stored. The relational model helped make it possible for people who were NOT computer experts to use the ever-increasing amounts of data produced by businesses, governments, and scientific research.

2.1 WHAT IS A RELATION?

The basic ideas of the relational model are as follows:

1. Data can be seen as organized into **relations** of arbitrary size. A relation can be visualized as a two-dimensional table with the **rows** representing individual entities and the columns representing different attributes of these entities. In the music database below, I've shown tables representing albums, artists, and tracks.
    a. More technically, we might think of a "table" as a *persistent* relation (one whose data in stored).
2. Each intersection between a column and row (a "cell") contains ONE value, and all entries within a single column must have the *same* data type and fall within the allowed values for the **attribute domain**. So, for example, each album can have precisely one title, and all titles must be "strings" of one or more characters. AlbumId and ArtistId, by contrast, are integers.
    a. A data dictionary fully specifies information about tables and their attribute types. DBMs will automatically generate a **system catalog** that provides an up-to-date computer version.
3. The order of the columns and rows is irrelevant to the DBMS. So, for example, the artists here are listed alphabetically, but the DBMS could just as easily show them to you in reverse alphabetical order, etc.
4. Each row must be *uniquely identifiable*. We ensure uniqueness using **keys,** which we'll discuss in the next section. For example, the "primary key" of the Album table is AlbumId. Meanwhile, the attribute ArtistId is a foreign key that *references* the primary key of a different table (in this case, the Artist table).

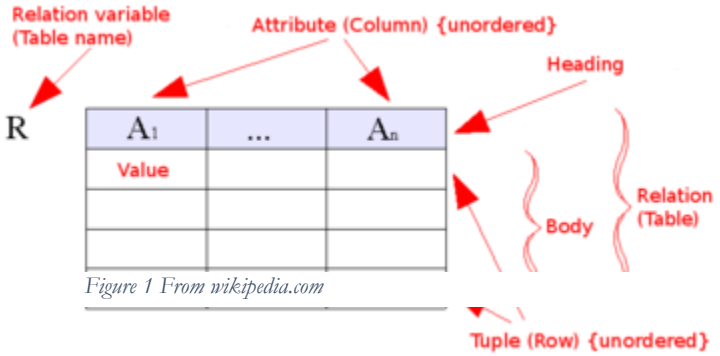


Figure 1 From wikipedia.com

Example: Database. To illustrate what we're talking about, let's consider some rows from two different tables from the public domain "Chinook" music database, one for "Album" and one for "Relation."

"Album" Table (or Relation)		
AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
4	Let There Be Rock	1
2	Balls to the Wall	2
3	Restless and Wild	2
5	Big Ones	3
6	Jagged Little Pill	4

"Artist" Table (or Relation)	
Name	ArtistId
AC/DC	1
Accept	2
Aerosmith	3
Alanis Morissette	4

5. And here are a few rows from the "Track" table:

TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice
1	For Those About To Rock (We Salute You)	1	1	1	Angus Young	343719	11170334	0.99
2	Balls to the Wall	2	2	1		342562	5510424	0.99
3	Fast As a Shark	3	2	1	F. Baltes	230619	3990994	0.99
4	Restless and Wild	3	2	1	F. Baltes	252051	4331779	0.99
5	Princess of the Dawn	3	2	1	Deaffy & R.A. Smith-Diesel	375418	6290521	0.99

2.2 WHAT IS A KEY? WHY ARE THEY IMPORTANT?

A key is a set of one or more attributes that serve to *determine* the value of the other attributes. In other words, if you know the value of the key attributes for a given entity, you are in a position to retrieve *other* information (regarding at least some other attributes) that you'd like to know. More technically, we can describe keys and their related concepts as follows:

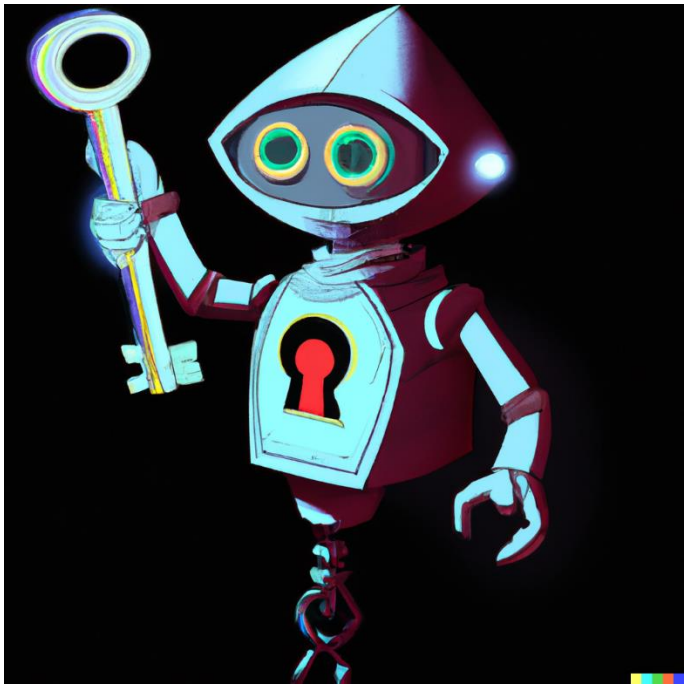


Figure 2 Good keys are important! (Brendan Shea x Dall-E)

- **Determination and Functional Dependence.** We say that attribute set A *determines* attribute set B (or that B is *functionally dependent* on A) if and only knowing the value of A allows you to determine the value of B. For example, in a government database, the attribute SocialSecuityNumber for the relation/table Person might serve to determine attributes such as FirstName, LastName, and so on. However, the converse does not hold—for example, knowing a person's first name does NOT allow you to determine their social security number (after all, many people have the same first name!).
- **Full Functional Dependence** requires that every member of A is *necessary* to determine B. For example, the attribute set (SocialSecurityNumber, FirstName) determines (LastName). However, we don't *need* FirstName to do this since SocialSecurityNumber would be enough. So, while LastName is functionally dependent on the larger set, it has full functional dependence on only SocialSecurityNumber.
- **Keys** serve to determine other attributes. Keys may be simple (one attribute) or **composite** (sets of multiple attributes). A **superkey** is a set of attributes that uniquely determine ALL attributes in a table. So, for example, if we had a Zoo table (for animals in a Zoo), knowing the value of the attribute "Species" would allow us to determine attributes such as "IsMammal." If we *knew* an animal was a tiger, we would know it was a mammal. However, knowing it was a tiger would not allow us to determine attributes such as "Sex" (male or female). So, Species wouldn't a superkey. By contrast, if each animal were associated a unique "id" number with each email, this id would be a superkey.
- A **candidate key** is a superkey with no unnecessary attributes included. The other attributes fully depend on this attribute (or set of attributes). One candidate key will be chosen as the **primary key** for the table. On other tables, we can include a **foreign key** attribute that allows us to reference this. In the music database, for example, the Track table uses "TrackId" as a primary key, while AlbumId, MediaTypeId, and GenreId are all foreign keys. There may also be **secondary keys** (or **logical keys**) used to look up data, but are not the primary key. For example, in many web applications, the *primary* key will be a unique number (such as the starId for MinnState students). Meanwhile, attributes such as "email address" will function as logical keys.

Keys are crucial to the functioning of relational databases in several ways:

- Keys ensure **referential integrity** by ensuring that any attribute defined is a "foreign key" does NOT point to a non-existent "primary key" and produce an error. The foreign key value will EITHER be a valid primary key OR a **null** ("no reference"). So, for example, in the Track table, a relational DBMS will ensure that GenreId, AlbumId are valid. (It might do this, for instance, by refusing to allow us to delete an Album until we've first deleted the tracks, or even doing this all for us).
- Primary keys ensure **entity integrity** by requiring every row to have a unique key value AND that there are no "nulls." Again, a relational DBMS will simply not allow us to violate this rule (and will, for example, return an error if you attempt to insert two different rows with the same primary key value).
- Keys are used to create **indexes** into the database, which make CRUD (create, retrieve, update, delete) much faster. For example, if we create an index on SocialSecurityNumber, we can use this knowledge to *quickly* retrieve data on a person, given their social security number. For example, a "B-tree" index might allow us to replace a line-by-line ("linear") search of 300,000,000 items with an operation accessing only *log* of this number (around 9!).

2.3 CODD'S 12 (ACTUALLY, 13!) COMMANDMENTS FOR RELATIONAL DATABASES

The power of the relational database lies in its ability to present users with a "logical" view of data, where users don't need to worry about all the messy details regarding hard drives, networks, or underlying file formats or organizations. However, the only way this can be achieved is to place *constraints* on what users can do. In particular, we risk losing this advantage once we start allowing selected users to "break" the rules when manipulating data. With this in mind, Codd proposed "12 rules" of relational databases, which were intended to warn against database software that tried to be "all things to all people." While no software package realizes these rules perfectly, they are now generally recognized as good statements of what an ideal relational database "should" be. I've adapted the descriptions here from Wikipedia's treatment (you can find dozens of such lists).

- **Rule 0:** The *foundation rule*: For any system that is advertised as, or claimed to be, a relational database management system, that system must be able to manage databases entirely through its relational capabilities. In other words, users should be able to use the "logical" language (of SQL or relational algebra) without anything else!
- **Rule 1:** The *information rule*: All information in a relational database is represented explicitly at the logical level and in exactly one way – by values in tables. A relational database just IS a collection of tables.
- **Rule 2:** The *guaranteed access rule*: Every datum (atomic value) in a relational database is guaranteed to be logically accessible by resorting to a combination of table name, primary key value, and column name. Hence, keys!
- **Rule 3:** *Systematic treatment of null values*: **Null values** (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type. We don't leave it to *users* to define NULL, since this can lead to bad results (for example, one person may code "missing data" as -1, but later users think this is a number!).
- **Rule 4:** *Dynamic online catalog based on the relational model*: The database description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data. In relational DBMSs, we can ask questions "about" the system catalog using the same language we use to ask questions about the data stored in the database. Neat!
- **Rule 5:** The *comprehensive data sublanguage rule*: A relational system may support several languages. However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and which supports core database capacities. For modern relational databases this language is SQL.

- **Rule 6:** The *view updating rule*: All views (i.e., way of "presenting" data to different users) that are theoretically updatable can be updated through the DBMS. Again, the idea is that all core database functionalities CAN be carried out using a logical language of SQL and do NOT require users to know more about how this particular DBMS works.
- **Rule 7:** *Possible for high-level insert, update, and delete*: Create, Retrieve, Delete, Update can be done using the DBMS using SQL or a similar language (and not require a separate program).
- **Rule 8:** *Physical data independence*: Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods. That is, users shouldn't notice any change in function when underlying data is moved from one drive to another, or when the files are renamed, etc.
- **Rule 9:** *Logical data independence*: Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables. For example, if we add new tables or attributes to a database, this should NOT impair any current database uses.
- **Rule 10:** *Integrity independence*: Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.
- **Rule 11:** *Distribution independence.* The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only, and not have to worry about hard drives, networks, or the like.
- **Rule 12:** The *nonsubversion rule*: If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

2.4 REVIEW QUESTIONS AND ACTIVITIES

1. Without looking at the handout, define the following terms: **relation, super key, primary key, candidate key,** and **referential integrity.**
2. Suppose that a social media website has a database has a User table, which records data on its users. This table contains the user name, among other things. It also has a Post table, which contains information on posts (in particular, the text "content" of the post, and the user who made it). Produce a data dictionary by answering the following questions.
    a. What are some attributes that might be included in the User table? That might be included on the Post table? What type of data is represented by each attribute?
    b. Are there any superkeys on each table? (If not, you'll need to add one!).
    c. What are the "candidate keys" for each table? Of these, which would you choose as the primary key?
    d. Are there any foreign keys present?
3. Choose a few of Codd's "12 rules" and give examples of data models that might violate these rules.

3 RELATIONAL ALGEBRA: LOGIC, NOT MAGIC!

In this lesson, we'll be answering the following questions:

1. What is **relational algebra,** and why is it so important to the relational model? What does it mean to say it is **complete?**
2. How do the **select, project,** and **rename** operators work?
3. How do the set **union** and set **difference** operators work?
4. How does the **cross-product** operator work, and how does this relate to **full joins, natural joins,** and **outer joins?**

You just learned that the relational model is based on the idea that data users can interact with data from the "logical" viewpoint of tables and rows. **Relational algebra** is the language to express this logic. Until Codd's paper (which launched the relational model), relational algebra was mostly an obscure language, of interest mainly to set theorists and logicians. Codd, however, showed that it had a practical use: so long as users understood just a few (five!) basic operations, they could combine these questions to produce ANY database query they liked! Here, we'll be looking at the basics of Codd's insight, without *too* much concern about becoming experts on it (database professionals rarely use relational algebra in "practical" applications). Later, when we turn our attention to Structured Query Language (SQL), we'll see many of these ideas again.

The basic idea behind relational algebra is straightforward. We take one or more relations as inputs, do something to them, and then produce a *different* relation as an output (so, the initial relations are untouched!). Relations are **closed** under the operators of relational algebra, which means that we will ALWAYS get a relation back (which can then be used as the input for more relational algebra!). In relational calculus (not covered here, but which provides equivalent functionality), we will always get back a TRUE or FALSE to a correctly formulated query.

3.1 THE BASICS: SELECT, PROJECT, RENAME, UNION, DIFFERENCE

Let's suppose that Tony Stark (the "Iron Man" character, apparently modeled after Oracle founder and database superstar Larry Ellison) has the following two relations R1 and R2. R1 contains facts about some of his friends, while R2 records various superheroes' favorite colors and whether or not they scare Tony Stark.

Relation R1			
first_name	**last_name**	**secret_identity**	**Age**
Peter	Parker	Spiderman	16
Wanda	Maximoff	Scarlett Witch	35
Thor	Odinson	Thor	1500

Relation R2		
secret_indentity	**favorite_color**	**scare_me**
Hulk	Green	True
Spiderman	Red	False
Scarlett Witch	Red	True
Black Widow	Black	False

We are now in a place to introduce the basic operators of relational algebra.

Selection $\sigma_{condition}Relation$ is used to select tuples (rows) from a relation (table) that satisfy some condition. So, $\sigma_{age > 30}R1$ will return just those rows from relation R1 where the age attribute has a value of greater than 30. You can also write this as SELECT age > 30 FROM R1 in **syntactically sugared relational algebra** (basically, relational algebra with less math-y symbols).

SELECT age > 30 FROM R1			
first_name	**last_name**	**secret_identity**	**Age**
Wanda	Maximoff	Scarlett Witch	35
Thor	Odinson	Thor	1500

Rename ($\rho_{"new\ table\ name"}Relation$) renames a table as "X." This can be super convenient since it allows us to reference the results of previous relational algebra operations easily. So, for example, I could rename the

result of the previous SELECT operation (which found "old" superheroes) to be the OldHeroes. There are several different notations for this. These all mean the same thing!

- $\rho_{OldHero}(\sigma_{\text{age}>30}R1)$
- $OldHero \leftarrow (\sigma_{\text{age}>30}R1)$
- SELECT age>30 FROM R1 AS OldHero

Projection ($\Pi_{\text{list of attributes}}\boldsymbol{Relation}$) is used to "project" selected attributes from a table, and ignore the others. So, $\Pi_{secret\ identity,scare\ me}R2$ or PROJECT secret_identity, scare_me FROM R2 returns a new table with just these attributes (and excluded favorite_color).

PROJECT secret_identity, scare_me FROM R2	
secret_identity	scare_me
Hulk	True
Spiderman	False
Scarlett Witch	True
Black Widow	False

Union. ($\boldsymbol{FirstRelation} \cup \boldsymbol{SecondRelation}$) "combines" the tuples/rows from two relations so long as they have (1) the same number of attributes and (2) these attributes have the same type (for example, we can combine strings with strings, but not strings with numbers). Trying to do something like $R1 \cup R2$ would fail because R1 has four attributes, while R2 has three. We could, however, create a union of just the "secrect_identity" attribute from both tables like this:

$$(PROJECT\ secret\_identity\ FROM\ R1) \cup (PROJECT\ \text{secret\_identity}\ FROM\ R2)$$

secret_identity
Hulk
Spiderman
Thor
Scarlett Witch
-Black Widow

Notice that even if some identities appeared in both tables (such as Spiderman) these would appear only once in the resulting table.

Difference ($\boldsymbol{FirstRelation} - \boldsymbol{SecondRelation}$) returns the tuples/row that appear in the first relation but do not appear in the second. For example,

$$(PROJECT\ \text{secret\_identity}\ FROM\ R1) - (PROJECT\ \text{secret\_identity}\ FROM\ R2)$$

secret_identity
Scarlett Witch
Thor

3.2 PRODUCTS AND JOINS

Cartesian Product (or "Cross Join"). The real power of relational algebra comes from the ability to combine (or **join)** different tables. The operation underlying this ability in relational algebra is called the **Cartesian product** (or **cross-product).** This operation returns *every possible combination of rows from R1 and R2.* So, if we cross R1 (which has three rows) with R2 (which has four rows), this yields a new relation with 12 total rows. Many of these rows make no intuitive "sense". For example, the first row contains information about two different superheroes (Hulk and Spiderman) and doesn't seem to represent anything "real" in the superhero world (that is, there is no Hulk-Spiderman combination).

For this reason, you rarely want to use this sort of operation in real-life operations. However, it's nevertheless a crucially vital concept to understand, because other (more useful!) joins are ways of taking the results of this sort of join and "filtering" its results. You want to make really, really sure that you don't accidentally make the database produce unfiltered cartesian products unless you need them. Cartesian products are big, slow to compute, and often yield "results" that are useless.

$\boldsymbol{R1 \times R2}$ **(R1 CROSS JOIN R2)**

first_name	last_name	R1.secret_identity	age	R2.secret_indentity	favorite_color	scare_me
Peter	Parker	Spiderman	16	Hulk	Green	True
Peter	Parker	Spiderman	16	Spiderman	Red	False
Peter	Parker	Spiderman	16	Scarlett Witch	Red	True
Peter	Parker	Spiderman	16	Black Widow	Black	False
Wanda	Maximoff	Scarlett Witch	35	Hulk	Green	True
Wanda	Maximoff	Scarlett Witch	35	Spiderman	Red	False
Wanda	Maximoff	Scarlett Witch	35	Scarlett Witch	Red	True
Wanda	Maximoff	Scarlett Witch	35	Black Widow	Black	False
Thor	Odinson	Thor	1500	Hulk	Green	True
Thor	Odinson	Thor	1500	Spiderman	Red	False
Thor	Odinson	Thor	1500	Scarlett Witch	Red	True
Thor	Odinson	Thor	1500	Black Widow	Black	False

A **natural join** takes the results of the above Cartesian join and throws away any tuples where (1) R1 and R2 have an attribute of the same name and domain, such as secret_identity, and (2) these shared attributes have different values. So, for example, a natural join will throw away rows where R1.secret_idenity = "Spiderman" while R2.secret_identity = "Thor." Finally, a natural join will keep only ONE copy of the shared attribute. Natural joins are the most common sorts of joins in database operations. A natural join of R1 and R2 looks like this.

R1 ⋈ R2 (R1 NATURAL JOIN R2)					
first_name	**last_name**	**secret_identity**	**Age**	**favorite_color**	**scare_me**
Peter	Parker	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	35	Red	True

Notice that a natural join includes *only* those rows with secret_identity in *both* tables. An **equijoin** is very similar to a natural join. However, the "shared column" (in this case, secret_identity) appears twice (both times with the same values.

	R1 EQUIJOIN JOIN R2					
first_name	**last_name**	**R1.secret_identity**	**R2.secret_identity**	**age**	**favorite_color**	**scare_me**
Peter	Parker	Spiderman	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	Scarlett Witch	35	Red	True

3.2.1 Outer Joins

Sometimes, though, you'll want to include rows even when they DON'T have matches in the other table. This is called an **outer join.** Outer joins can be used to explore issues regarding: "referential integrity" (when an attribute in one table doesn't "refer" to the corresponding attribute in another table). There are several types of outer joins:

An **outer left join** between R1 and R2 returns every row that the natural join does plus "unmatched" rows from the "lefthand" relation (in this R1). The unmatched rows will contain "null" for any value that is missing:

R1 OUTER LEFT JOIN R2					
first_name	**last_name**	**secret_identity**	**Age**	**favorite_color**	**scare_me**
Peter	Parker	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	35	Red	True
Thor	Odinson	Thor	1500	NULL	NULL

A **right join** does just the opposite: it returns a natural join plus any unmatched rows from the "righthand" relation (in this case, R2):

R1 OUTER RIGHT JOIN R2					
first_name	**last_name**	**secret_identity**	**age**	**favorite_color**	**scare_me**
NULL	NULL	Hulk	NULL	Green	True
Peter	Parker	Spiderman	16	Red	False
Wanda	Maximoff	Scarlett Witch	35	Red	True
NULL	NULL	Black Widow	NULL	Black	False

3.3 PUTTING IT ALL TOGETHER: RELATIONAL ALGEBRA IN ACTION

Now that we've seen how this works in theory, let's try putting it together. Here, I've also used a few *logical operators* such as **OR, AND,** and **NOT** that will be familiar to those of you who've taken classes in programming, discrete mathematics, or formal logic (and here you thought you'd never get to use these skills!). I've also included some Here are some sample queries Iron Man might run. I've described what they *mean;* however, I'll leave it up to you to determine what they will return.

English Statement	Relational Algebra Statement	Notes
"Show me just the last names and ages from R1"	PROJECT(last_name, age) FROM R1	To project two attributes, we use commas
"Which people from R1 are named 'Pater' or 'Wanda'?"	SELECT(first_name = Peter **OR** first_name = Wanda)	Here we have two "terms," and we want to return rows that satisfy *either* term.
"Which people from R2 do NOT have red as their favorite color?"	SELECT(**NOT**(favorite_color = red)) FROM R2	NOT(x = y) is often written as x !=y
"Give me a list of the people who appear on both tables, with everything I know about them."	R1 NATURAL JOIN R2	This is a typical query! In fact, it is *so* common that people sometimes forget that the "default" join is often cross-join.
"List the secret identities of just those people that scare me AND whose favorite color is green from R2"	PROJECT secret_identity FROM (SELECT(scare_me=TRUE AND favorite_color=green) FROM R2)	Pay attention to the nesting here. We PROJECT from the result of the SELECT query.

3.4 REVIEW PROBLEMS: PRACTICE WITH RELATIONAL ALGEBRA[i]

Consider a database with the following schema:

Relations	**Keys**
Person (<u>name</u>, age, gender)	name is a key
Frequents (<u>name</u>, <u>pizzeria</u>)	(name, pizzeria) is a key
Eats (<u>name</u>, <u>pizza</u>)	(name, pizza) is a key
Serves (<u>pizzeria</u>, <u>pizza</u>, price)	(pizzeria, pizza) is a key

1. Write relational algebra expressions for the following nine queries. (Warning: some of the later queries are a bit challenging.)

a. Find all pizzerias frequented by at least one person under the age of 18.
b. Find the names of all females who eat either mushroom or pepperoni pizza (or both).
c. Find the names of all females who eat both mushroom and pepperoni pizza.
d. Find all pizzerias that serve at least one pizza that Amy eats for less than $10.00.
e. Find all pizzerias that are frequented by only females or only males.
f. For each person, find all pizzas the person eats that are not served by any pizzeria the person frequents. Return all such person (name) / pizza pairs.
g. Find the names of all people who frequent only pizzerias serving at least one pizza they eat.
h. Find the names of all people who frequent every pizzeria serving at least one pizza they eat.
i. Find the pizzeria serving the cheapest pepperoni pizza. In the case of ties, return all of the cheapest-pepperoni pizzerias.

2. Consider a schema with two relations, R(A, B) and S(B, C), where all values are integers. Make no assumptions about keys. Consider the following three relational algebra expressions:

a. $\pi_{A,C}(R \bowtie \sigma_{B=1}S)$

b. $\pi_{A}(\sigma_{B=1}R) \times \pi_{C}(\sigma_{B=1}S)$

c. $\pi_{A,C}(\pi_{A}R \times \sigma_{B=1}S)$

Two of the three expressions are equivalent (i.e., produce the same answer on all databases), while one of them can produce a different answer. Which query can produce a different answer? Give the simplest database instance you can think of where a different answer is produced.

4 OPTIONAL READING: EDGAR F CODD: A TRIBUTE AND PERSONAL MEMOIR (BY CJ DALE)[ii]

[Brendan's Note: Many of the ideas we've been exploring in this class so far, especially as they relate to the relational model, trace back to the work of Edgar Codd. This piece—published shortly after his death in 2003—gives an overview of his contributions.]

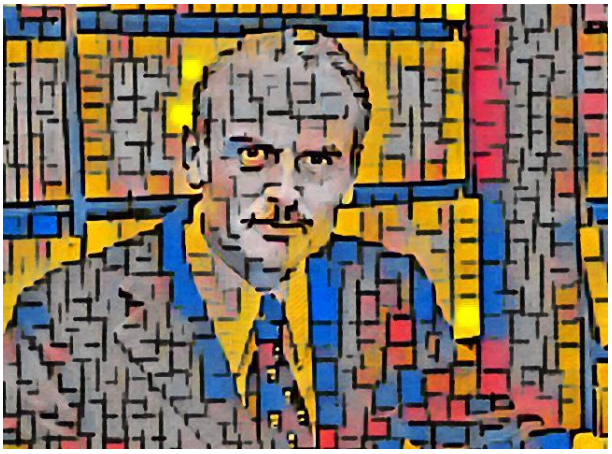
This is an expanded version of a piece that originally appeared [on the SIGMOD website as an obituary]. Here's the opening paragraph from that earlier piece:

> *By now there cannot be many in the database community who are unaware that, sadly, Dr. E. F. Codd passed away on April 18th, 2003. He was 79. Dr. Codd, known universally to his colleagues and friends—among whom I was proud to count myself—as Ted, was the man who, singlehanded, put the field of database management on a solid scientific footing. The entire relational database industry, now worth many billions of dollars a year, owes the fact of its existence to Ted's original work, and the same is true of all of the huge number of relational database research and teaching programs under way worldwide in universities and similar organizations. Indeed, all of us who work in this field owe our career and livelihood to the giant*

contributions Ted made during the period from the late 1960s to the early 1980s. We all owe him a huge debt. This tribute to Ted and his achievements is offered in recognition of that debt.

However, I've discovered I was operating on a false assumption when I wrote this paragraph. To be more specific, I've found—and it's a sad comment on the state of our field that I feel I have to say this—that many database practitioners, and even some researchers, really don't know who Ted was or what he did. In my own day-to-day database activities (for example, on the seminars I teach), I often encounter people who've never even heard of him! So I thought it would be a good idea to amplify and republish my original tribute: in particular, to elaborate briefly on the nature of some of Ted's numerous technical contributions. I'd also like to say a little more about "Ted the man"—i.e., Ted Codd as I personally remember him, and what he meant to me.

5 BACKGROUND



Ted Codd was a native of England and a Royal Air Force veteran of World War II. He moved to the United States after the war and became a naturalized US citizen. He held MA degrees in mathematics and chemistry from Oxford University and MS and PhD degrees in communication sciences from the University of Michigan.

Ted began his computing career in 1949 as a programming mathematician for IBM on the Selective Sequence Electronic Calculator. He subsequently participated in the development of several important IBM products, including the 701, IBM's first commercial electronic computer, and STRETCH, which led to IBM's 7090 mainframe technology. Then, in the late 1960s, he turned his attention to the problem of database management—and over the next several years he created the invention with which his name will forever be associated: **the relational model of data**.

5.1 DATABASE CONTRIBUTIONS

Ted Codd's relational model is widely recognized as one of the great technical innovations of the 20th century. Ted described it and explored its implications in a series of research papers — staggering in their originality—that he published during the period from 1969 to 1981. The effect of those papers was twofold:

- First, they changed for good the way the IT world perceived the database management problem.
- Second, as already mentioned, they laid the foundation for a whole new industry.

In fact, they provided the basis for a technology that has had, and continues to have, a major impact on the very fabric of our society. It's no exaggeration to say that Ted is the intellectual father of the modern database field.

To give some idea of the scope and extent of Ted's accomplishments, I'd like to highlight in this section what seem to me the most significant of his database contributions. Of course, the biggest of all was, as already mentioned, to make database management into a science (and thereby to introduce a welcome and sorely needed note of clarity and rigor into the field): The relational model provides a theoretical framework within

which a variety of important problems can be attacked in a scientific manner. Ted first described his model in 1969 in an IBM Research Report:

- "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599 (August 19th, 1969)

He also published a revised version of this paper the following year:

- "A Relational Model of Data for Large Shared Data Banks," *CACM 13,* No. 6 (June 1970) and elsewhere[1].

(This latter——i.e., the 1970 paper——is usually credited with being the seminal paper in the field, though this characterization is a little unfair to its 1969 predecessor.) Almost all of the novel ideas described in outline in the remainder of this section, as well as numerous subsequent technical developments, were foreshadowed or at least hinted at in these first two papers; in fact, some of those ideas remain less than fully explored to this day. In my opinion, everyone professionally involved in database management should read, and reread, at least one of these papers every year. (It's true they're not all that ***easy*** to read, being fairly abstract and somewhat mathematical in tone. On the other hand, they stand up extremely well to being read, and indeed repeatedly reread, over 30 years after they were written! How many technical papers can you say that of?)

Incidentally, it's not as widely known as it should be that Ted not only invented the relational model in particular, he invented the whole concept of a ***data model*** in general. See his paper:

- "Data Models in Database Management," *ACM SIGMOD Record 11,* No. 2 (February 1981)

This was the first of Ted's papers to include a definition of the term ***data model***. It also addressed the question: What purposes are data models in general, and the relational model in particular, intended to serve? And it then went on to offer evidence in support of the claim that, contrary to popular belief, the relational model was the first data model to be defined. (The socalled hierarchic and network models, which are often thought to have predated the relational model, were actually defined after the fact by a process of induction——in this context, a polite word for guesswork——from existing implementations.)

In connection with both the relational model in particular and data models in general, the paper also stressed the importance of the distinction, regrettably underappreciated even today, between a data model and its physical implementation.

Ted also saw the potential of using ***predicate logic*** as a foundation for a database language. He discussed this possibility briefly in his 1969 and 1970 papers, and then, using the predicate logic idea as a basis, went on to describe in detail what was probably the very first relational language to be defined, ***Data Sublanguage ALPHA,*** in:

- "A Data Base Sublanguage Founded on the Relational Calculus," ***Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control***, San Diego, Calif.

  (November 1971)

ALPHA as such was never implemented, but it was extremely influential on certain other languages that were, including in particular the Ingres language QUEL and (to a much lesser extent) the IBM language SQL as well.

[1] Many of Ted's papers were published in several places. Here I'll just give the primary sources.

If I might be permitted a personal anecdote at this point, I remember being invited, in 1974, to Anaheim, California, to present the basic ideas of the relational model to the GUIDE Database Language Working Group. (I probably don't have that title quite right, but it was a project within GUIDE, the IBM user group, to examine possible programming language extensions for database access.) Members of the working group had coded five sample applications using various database products of the time (TOTAL, IDS, IMS, etc.), and the meeting began with those various solutions being presented. I distinctly remember the group's reaction (chiefly amazement, mingled with delight) when I was able to show that each of their applications—which I came to stone cold, never having seen them before that afternoon—involved ***just one line*** of ALPHA code! At that point, the IBM representative to the group took me off to one side and told me "not to come on too strong about this relational stuff." I said: "Why not?" He said: "Because we don't have a product." To which I answered: "Then maybe we should get one." SQL/DS was announced a mere seven years later ...

Back to Ted. Ted subsequently defined the ***relational calculus*** more formally, as well as the ***relational algebra,*** in:

- "Relational Completeness of Data Base Sublanguages," in Randall J. Rustin (ed.), ***Data Base Systems: Courant Computer Science Symposia Series 6*** (Prentice-Hall, 1972)

Very loosely speaking, relational calculus provides a notation for ***defining*** some desired relation

(typically the result of some query) in terms of others, while relational algebra provides a set of operators for ***computing*** some desired relation from others. Clearly, each could be used as a basis on which to define a query language. As I've already indicated, QUEL is an example of a language that's based on the calculus. SQL, by contrast, is a mixture: It includes some elements that are calculus-based, others that are algebra-based, and still others that are neither.

As the title indicates, the paper also introduced the notion of ***relational completeness*** as a basic measure of the expressive power of a database language. Essentially, a language is said to be relationally complete if it's as powerful as the calculus. To quote: "A query language ... which is claimed to be general purpose should be at least relationally complete in the sense defined in this paper. [Such a language need never resort] to programming loops or any other form of branched execution—an important consideration when interrogating a [database] from a terminal."

The same paper also described an algorithm—***Codd's reduction algorithm***—for transforming an arbitrary expression of the calculus into an equivalent expression in the algebra, thereby (a) proving the algebra is relationally complete and (b) providing a basis for implementing the calculus.

Ted also introduced the concept of ***functional dependence*** and defined the first three ***normal forms*** (1NF, 2NF, 3NF). See the papers:

- "Normalized Data Base Structure: A Brief Tutorial," ***Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control***, San Diego, Calif. (November 11th-12th, 1971)

- "Further Normalization of the Data Base Relational Model," in Randall J. Rustin (ed.), ***Data Base Systems: Courant Computer Science Symposia Series 6*** (Prentice-Hall, 1972)

Simplifying considerably, we can say that:

- Given relation *R*, attribute *B* of *R* is functionally dependent on attribute *A* of *R*—— equivalently, the functional dependence (FD) $A \rightarrow B$ holds in *R*——if and only if, whenever two tuples of *R* have the same value for *A*, they also have the same value for *B*.

- Relation *R* is in third normal form, 3NF, if and only if the only FDs that hold in *R* are of the form $K \rightarrow B$, where *K* is a key. (In the interests of accuracy, I should say that this is really a definition of what subsequently became known as *Boyce/Codd normal form*, BCNF, not 3NF as such. Part of the confusion arises from the fact that Ted himself referred to BCNF as "third" normal form for some years.)

Between them, Ted's two normalization papers laid the foundations for the entire field of what is now known as *dependency theory*, an important branch of database science in its own right. Among other things, that theory serves as a basis for a truly scientific approach to the problem of logical database design.

As an aside, I note that the first of the two papers, the tutorial one, also includes arguments for excluding pointers from the user's view of the database. The relational model's total ban on pointers is just one of many factors that sharply distinguish it from its competitors. Indeed, any language that exposes pointers to the user, in any shape or form, forfeits all right to be called relational——and the introduction of pointers into the SQL:1999 standard in particular can thus only be deplored. But I digress.

There's something else I'd like to say here. Normalization is sometimes criticized on the grounds that "it's all really just common sense"; any competent professional would "naturally" design databases to be fully normalized anyway, without having to know anything about dependency theory at all. In other words, the problems with less than fully normalized designs are "obvious," and common sense is sufficient to tell us that fully normalized designs are better. But what do we mean by "common sense"? What are the *principles* (inside the designer's brain) that the designer is applying when he or she chooses a fully normalized design? The answer is, of course, that they're exactly the principles of normalization. In other words, those principles are indeed just common sense——but (and here comes the point) they're *formalized* common sense. The whole point is to try to identify such principles and formalize them——which isn't an easy thing to do! But if it can be done, then we can *mechanize* them; in other words, we can get the machine to do the work. Critics of normalization usually miss this point; they claim, quite rightly, that the ideas are all basically common sense, but they typically don't realize that it's a significant achievement to state what "common sense" means in a precise and formal way.

Yet another crucial notion introduced by Ted was that of *essentiality*. Ted defined that notion in:

- "Interactive Support for Nonprogrammers: The Relational and Network Approaches," *Proc. ACM SIGMOD Workshop on Data Description, Access, and Control, Vol. II*, Ann Arbor, Michigan (May 1974)

This paper was Ted's principal written contribution to "The Great Debate." The Great Debate——the official title was *Data Models: Data-Structure-Set vs. Relational*——was a special event held at the 1974 SIGMOD Workshop; it was subsequently characterized in *CACM* by Robert L. Ashenhurst as "a milestone event of the kind too seldom witnessed in our field." (I should explain that "the data-structure-set model" is just the CODASYL network model by another name.)

The concept of essentiality, introduced by Ted in this debate, is a great aid to clear thinking in discussions regarding the nature of data and DBMSs. Basically, an "information carrier" (in other words, a construct for the representation of data) is said to be *essential* if and only if its removal would cause a loss of information.

Now, the relational model provides just one way to represent data——namely, by means of relations themselves——and the sole essential information carriers in a relational database are thus necessarily relations, *a fortiori*. By contrast, other data models typically provide many distinct ways to represent data (lists, bags, links, sets, arrays, and so on), and all or any of those ways can be used as essential information carriers in a nonrelational database. One way of representing data is both necessary and sufficient; more than one introduces complexity, but no additional power.

As a matter of fact, it's the concept of essentiality that forms the underpinning for the wellknown, and important, *Information Principle:*

> *The entire information content of a relational database is represented in one and only one way: namely, as attribute values within tuples within relations.*

I heard Ted refer to this principle on more than one occasion as ***the*** fundamental principle underlying the relational model. (I remark in passing that a better name for it might be ***The Principle of Uniform Representation***.)

It seems appropriate to round out this discussion with a succinct, reasonably formal definition of the relational model. (This definition doesn't come from any of Ted's papers——it's my own attempt to capture the essence of what Ted was talking about in all of the papers I've been discussing, as well as many others.) Briefly, the relational model consists of five components:

1. An open-ended collection of **scalar types** (including in particular the type ***boolean*** or ***truth value***)
2. A **relation type generator** and an intended interpretation for relations of types generated thereby
3. Facilities for defining **relation variables** of such generated relation types
4. A **relational assignment** operation for assigning relation values to such relation variables
5. An open-ended collection of generic **relational operators** ("the relational algebra") for deriving relation values from other relation values

Let me conclude this section by observing that Ted's work on the relational model didn't just set the entire field of database management on a solid scientific footing——it also provided the foundations for the careers of numerous people, myself not least (and doubtless many readers of this tribute could say the same). Not to mention the fact that there is, and continues to be, tremendous intellectual excitement, rigor, and integrity, in those foundations that Ted laid down. (By the way, I don't want to give the impression that all of the foundation-level problems have been solved. They haven't. That's partly why the field is still so rewarding!) So those of us who have continued in Ted's footsteps haven't just had a career and a livelihood, as I said before——we've had work that was, and continues to be, both interesting and useful. We all owe Ted an enormous debt of gratitude.

5.2 PERSONAL MEMORIES

I first met Ted in 1971. I'd been doing some database work myself for IBM in England in 197071, as a result of which I entered into a correspondence with Ted, and he invited me to the US to present my ideas at various locations within IBM. So I got to meet Ted at the IBM San Jose Research Laboratory, where he was working at the time. I was immediately struck by his energy, and in particular by the care and precision with which he expressed himself. He was always careful never to overclaim. For example, in his paper "Further Normalization of the Data Base Relational Model," he states that one advantage of normalization is that it "[tends] to capture some aspects of the semantics (minor, of course)." I love that parenthetical remark! What a contrast to some of the overblown claims we so often encounter elsewhere in the literature.

Following our meeting in 1971, Ted and I began a cooperation—at first informal, later more formal—that lasted essentially throughout the remainder of his career. The precise nature of that cooperation varied over time, but the essence of it was that Ted continued to do research on his model and I took it on myself to present and explain his ideas at technical conferences, user group meetings, and the like. In this connection, Ted was always encouraging, supportive, and generous to me, as indeed he was to all of his friends and coworkers. For example (this is from a letter to me dated April 7th, 1972): "I am pleased to learn you are contemplating writing a [database book] ... The kind of book you propose is much needed ... I feel confident that you can produce an excellent book. You may make use of any of the material in my papers for this purpose."

Other people who had the opportunity to work with or for Ted over the years have confirmed to me what an inspiration he could be in their own endeavors.

Ted was easily one of the most unforgettable characters I ever met. He was a genius, of course—and like all geniuses he "stood at a slight angle to the rest of the universe." It wasn't just database, either. Something else he displayed his genius in was *losing things* ... When we were working together in Codd and Date Inc. (a consulting company we and Sharon Weinberg, later Sharon Codd, started in the early 1980s), I once personally saw him lose a crucial document as he crossed from one side of our office to the other. I didn't see it go—I just saw him start off with it on one side of the room and arrive at the other side without it. I don't think we ever found it again.

Another way his genius manifested itself was in a somewhat idiosyncratic sense of humor (again I suspect this is a property that is common to many geniuses). Things that most people didn't think were funny at all he would sometimes find incredibly funny; conversely, he would fail, sometimes, to see the funny side of things that other people thought were hilarious. When Sharon, Ted, and I were forming our consulting company, the question of a company logo came up, for use on business cards and the like. I said I thought the logo should incorporate a stylized picture of a table. Sharon agreed. Ted said: "I don't get it—we're not in the furniture business, are we?"

There are many, many Ted Codd stories, but I think one more here has to suffice. (I like this one, though actually I wasn't there when it happened. So it might be apocryphal. But it does have the ring of truth about it.) Apparently, when Ted was made an IBM Fellow—see the next section—he said:

"It's the first time I recall someone being made an IBM Fellow for someone else's product."

6 CONCLUDING REMARKS

Ted was a genuine computing pioneer. And he achieved more in his lifetime than most of us ever will or could, if we had ten lifetimes. Look at the honors he was awarded (and this is only a partial list):

- IBM Fellow
- ACM Fellow
- Fellow of the British Computer Society
- Member of the National Academy of Engineering
- Member of the American Academy of Arts and Sciences
- 1981 ACM Turing Award
- Outstanding recognition award from IEEE
- First annual achievement award from IDUG

- 2001 Hall of Fame Lifetime Achievement Award from DAMA

All thoroughly deserved, of course.

Ted was an inspiration and a good friend to all of us who had the privilege, fortune, and honor to know him and work with him. It's a particular pleasure to be able to say that he was always scrupulous in giving credit to other people's contributions. Moreover, as I've already mentioned (and despite his huge achievements), he was also careful never to overclaim; he would never claim, for example, that the relational model could solve all possible problems or that it would last forever. And yet those who truly understand that model do believe that the class of problems it can solve is extraordinarily large and that it will endure for a very long time. Systems will still be being built on the basis of Codd's relational model for as far out as anyone can see.

Ted is survived by his wife Sharon; a daughter, Katherine; three sons, Ronald, Frank, and David; his first wife Libby, mother of his children; and six grandchildren. He also leaves other family members, friends, and colleagues all around the world. He is mourned and sorely missed by all.

—C. J. Date

Healdsburg, California, 2003

---

[i] These questions are adapted from Stanford's "Database" taught by Jennifer Wisdom in 2011.

[iiii] From SIGMOD Record, Vol. 32, No. 4, December 2003.