RULES ABOUT RELATIONS

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

1. What exactly is 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 early 1970s, IBM's **Edgar Codd** published a paper called "Relational completeness of data base sublanguages" [1], which eventually would change the way 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 the languages of **relational algebra** (the basis of SQL) or **relational calculus** could be used to interact with this data in a "complete" way. In comparison to the file-system model of data storage, the relational model presented a "logical" view of data, in which those who interacted with the database didn't need to know anything about the manner in which the data was stored. This would eventually make it much more possible for people who were NOT experts on computers to make use of the ever-increasing amounts of data produced by business, government, and scientific research.

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 2-dimenionsal table with the **rows** representing individual entities and the columns representing different attributes of these entities. In the music database shown 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 exactly one title, and all titles must be "strings" of one or more characters. AlbumId and ArtistId, by contrast, are integers.
    a. The attribute types are specified by a **data dictionary** that fully describes the parameters for each attribute for every table in the database. DBMs will automatically generate a **system catalog** that provides an up-to-date, computer version of this.
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*. This involves the use of **keys,** about which we'll say more in the next section. For example, the "primary key" of the Album table is AlbumId, while the attribute ArtistId is a foreign key that *references* the primary key of a different table (in this case, the Artist table).

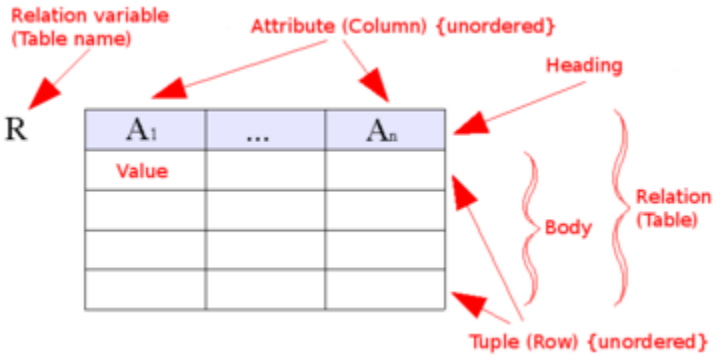

Figure 1 From wikipedia.com

Example: Database. To illustrate what we've talking about so far, 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

[1] E. F. Codd, "Relational Completeness of Data Base Sublanguages," in *Database Systems* (Prentice-Hall, 1972), 65–98.

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

WHAT IS A KEY? WHY ARE THEY IMPORTANT?

A key is a set of one or more attributes that serves 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:

- **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 actually *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 set of attributes that serve to *uniquely* identify 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." That is, if we *knew* an animal was a tiger, we would know it was a mammal. This would NOT, however, allow us to determine attributes such as "Sex" (male or female). So, this wouldn't a superkey. By contrast, if each we associated a unique "id" number with each email, this would be a superkey.
- A **candidate key** is a superkey with no unnecessary attributes included. That is, the other attributes have full functional dependence on this attribute (or set of attributes). ONE such 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**) that are 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 of some type (such as the starId for MinnState students), but something like "email address" will function as a logical key.

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

- Keys ensure **referential integrity,** by making sure that any attribute that we have told it 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 example, 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,** which requires that every row 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, given a "B-tree" index, instead of going through 300,000,000 items (which might the case *without* an index), we may need only to access the *log* of this (around 9!).

CODD'S 12 (ACUTALLY, 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, once we start allowing selected users to "break" the rules when manipulating data, we risk losing this advantage. With this mind, Codd proposed "12 rules" of relational databases, which were intended as a warning 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 a good relational database "should" be. My descriptions here are adapted 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 data base management system, that system must be able to manage data bases 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 data base 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*: Each and every datum (atomic value) in a relational data base 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 data base 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 anything 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 do things like add new tables or attributes to a database, this should NOT impair any *current* uses of database.
- **Rule 10:** *Integrity independence*: Integrity constraints specific to a particular relational data base 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).

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 contains the user name, among other thing. 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 least FIVE attributes that might included on the User table? Three that might be included on the Post table? What type of data are 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?