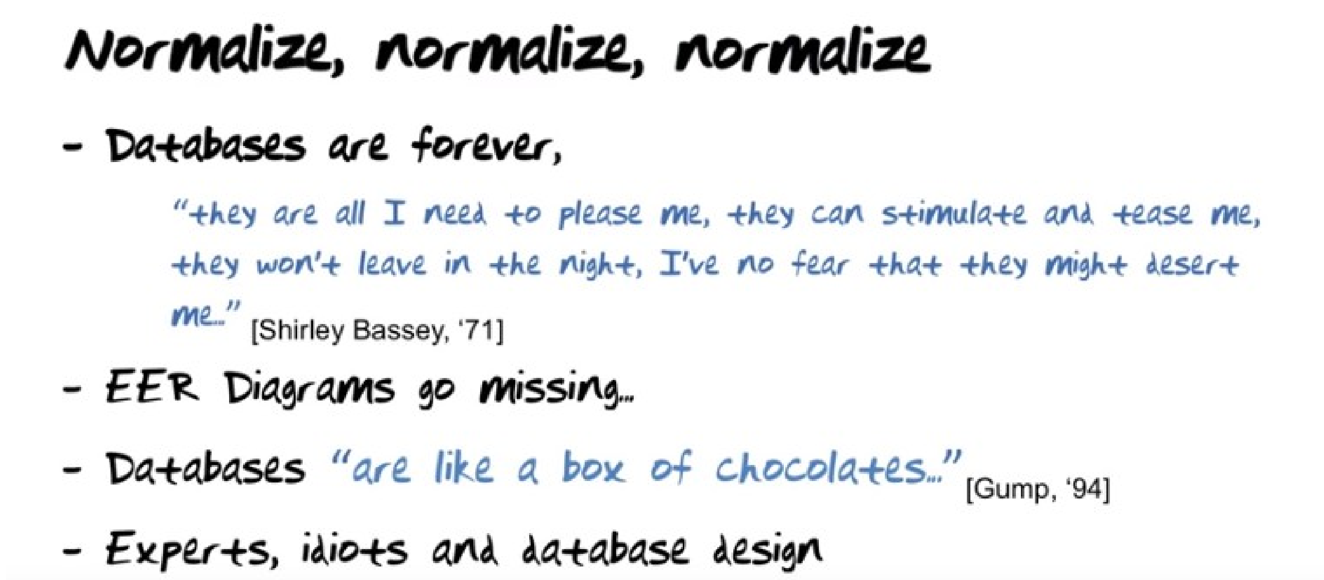
From online:

If you've been working with databases for a while, chances are you've heard the term normalization. Perhaps someone's asked you "Is that database normalized?" or "Is that in BCNF?" All too often, the reply is "Uh, yeah." Normalization is often brushed aside as a luxury that only academics have time for. However, knowing the principles of normalization and applying them to your daily database design tasks really isn't all that complicated and it could drastically improve the performance of your DBMS.

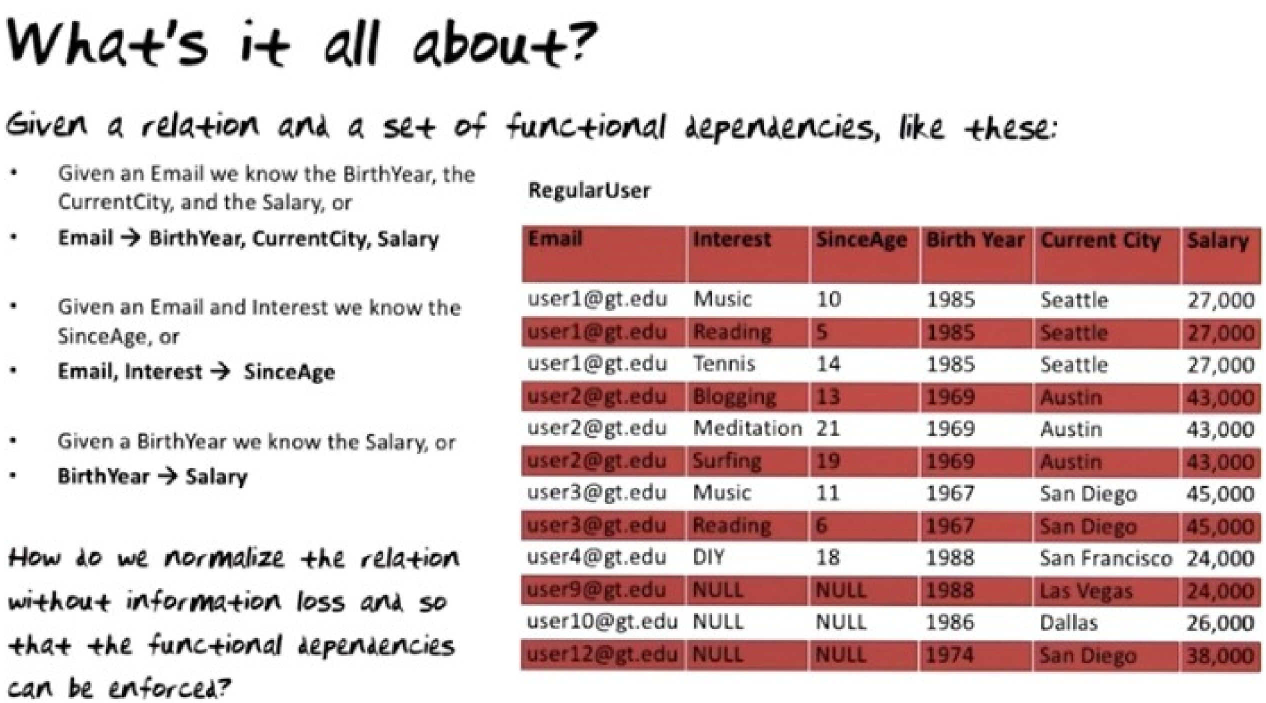
What is Normalization?
Normalization is the process of efficiently organizing data in a database. There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a table).

From Wiki:

Normalization involves decomposing a table into less redundant (and smaller) tables without losing information; defining foreign keys in the old table referencing the primary keys of the new ones. The objective is to isolate data so that additions, deletions, and modifications of an attribute can be made in just one table and then propagated through the rest of the database using the defined foreign keys.



The above figure: The mapping of EER diagrams to relations will in all cases result in a database which is normalized. In spite of that, we still have to learn about normalization. 圖中列的幾點是 the reasons yo need to know how to normalize database.



What's it all about?

Given a relation and a set of functional dependencies, like these:

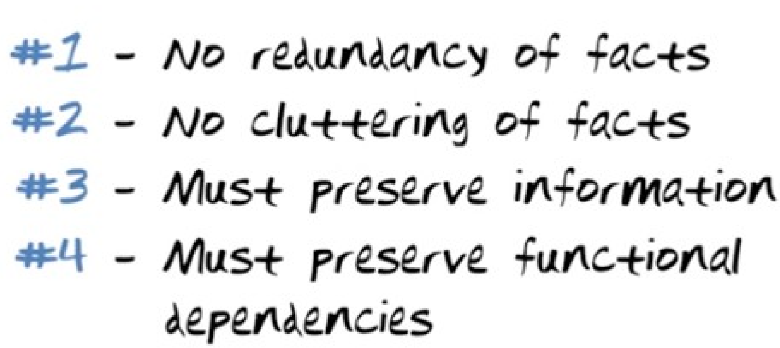
- Given an Email we know the BirthYear, the CurrentCity, and the Salary, or
- **Email → BirthYear, CurrentCity, Salary**
- Given an Email and Interest we know the SinceAge, or
- **Email, Interest → SinceAge**
- Given a BirthYear we know the Salary, or
- **BirthYear → Salary**

How do we normalize the relation without information loss and so that the functional dependencies can be enforced?

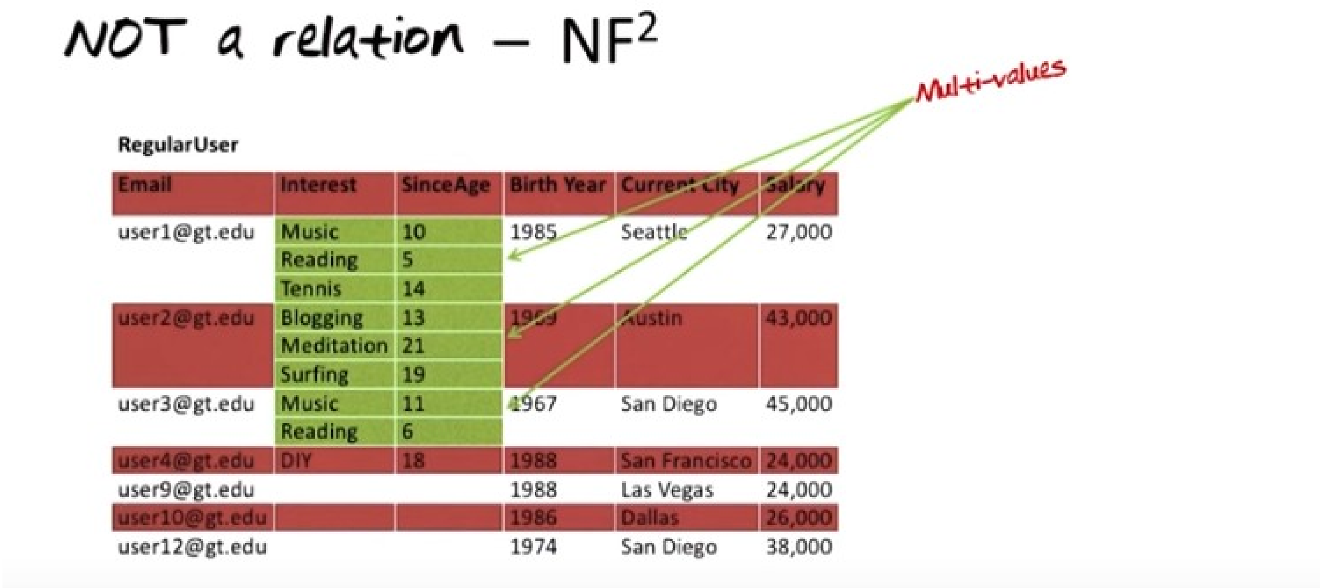
RegularUser

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

上圖表中的三個 User1(兩白一紅)是同一個人, 分成三行列出來是因為他的不同 interest.



The above figure: Here are the rules you must obey when you normalize relation.



Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
	Reading	5			
	Tennis	14			
user2@gt.edu	Blogging	13	1969	Austin	43,000
	Meditation	21			
	Surfing	19			
user3@gt.edu	Music	11	1967	San Diego	45,000
	Reading	6			
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu			1988	Las Vegas	24,000
user10@gt.edu			1986	Dallas	26,000
user12@gt.edu			1974	San Diego	38,000

The above figure: you may end up with a data structure that's not a relation, or data structure that we call a non first normal form (NF^2) data structure (haha 處). 表中的 user1 是管 Music-10, Reading-5, Tennis-14 這三行的, user2 是管 Bloggin-13, Meditation-21, Surfing-19 這兩行的, user3... So take a data structure like this, which is non first normal form and create a relation from it, there is one obvious way we could try to do that: so what if I simply repeat user1 for each one of these rows, …, then I will actually have a relation, 即前面給出的那個有多行 user1 的表, 也即下圖中的那個表.

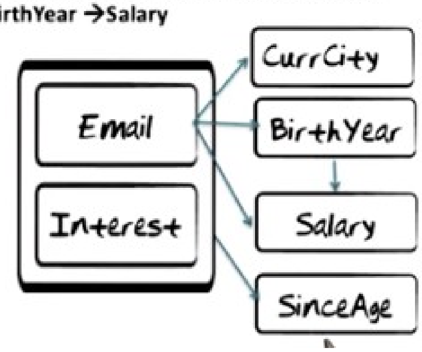
Relation with problems

Functional Dependencies:

Given an Email we know the BirthYear, the CurrentCity, and the Salary, or **Email → BirthYear, CurrentCity, Salary**

Given an Email and Interest we know the SinceAge, or **Email, Interest → SinceAge**

Given a BirthYear we know the Salary, or **BirthYear →Salary**



RegularUser

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

Relation with problems: Redundancy

For each Email the same BirthYear, CurrentCity, and Salary are repeated

For each BirthYear the same Salary is repeated

RegularUser

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

The above figure: redundancy potentially leads to inconsistency. 比如上表中只將第一個 user1 的城市從 Seattle 換成了 Atlanta, 而其它兩個 user1 忘了換.

Relation with problems: Insertion anomaly

If we insert a new RegularUser without any Interest, then we must insert NULL values for Interest and SinceAge

If we insert a pair of BirthYear and Salary then we must insert NULL values for Email, Interest, SinceAge and CurrentCity

RegularUser

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000
NULL	NULL	NULL	1970	NULL	42,000

Relation with problems: Deletion anomaly

If we delete user12@gt.edu, then we lose the fact that when the BirthYear is 1974 then the Salary is 38,000

RegularUser

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

Relation with problems: Update anomaly

If we update the CurrentCity of a RegulatUser, the we must do it in multiple places

If I update BirthYear, then we must update Salary

RegularUser

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

上圖中的"If I update BirthYear, then we must update Salary"的意思是: if you want to update the salaries that are made by people with BirthYear 1974, you gonna have to remember to do it everywhere.

Information loss

If we decompose RegularUser into two relations, then we could get too many rows back when we recombine them

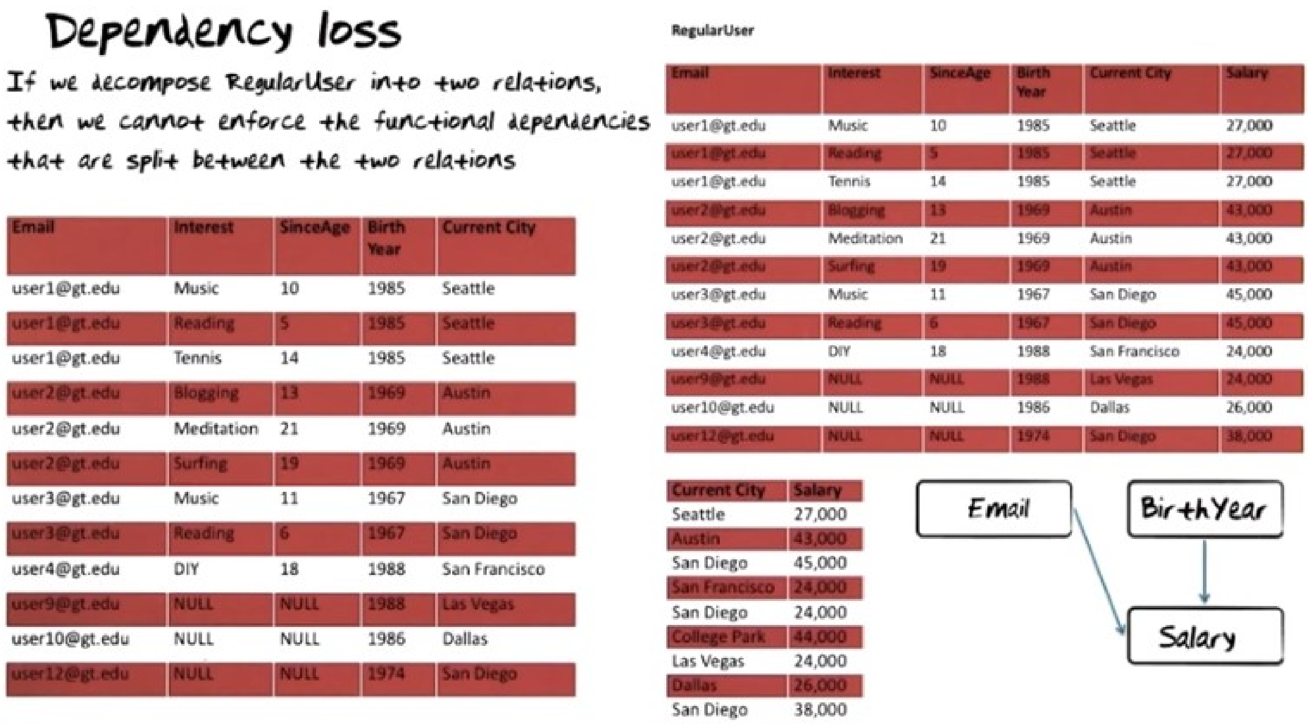
RegularUser

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

Email	Interest	SinceAge	Birth Year	Current City
user1@gt.edu	Music	10	1985	Seattle
user1@gt.edu	Reading	5	1985	Seattle
user1@gt.edu	Tennis	14	1985	Seattle
user2@gt.edu	Blogging	13	1969	Austin
user2@gt.edu	Meditation	21	1969	Austin
user2@gt.edu	Surfing	19	1969	Austin
user3@gt.edu	Music	11	1967	San Diego
user3@gt.edu	Reading	6	1967	San Diego
user4@gt.edu	DIY	18	1988	San Francisco
user9@gt.edu	NULL	NULL	1988	Las Vegas
user10@gt.edu	NULL	NULL	1986	Dallas
user12@gt.edu	NULL	NULL	1974	San Diego

Current City	Salary
Seattle	27,000
Austin	43,000
San Diego	45,000
San Francisco	24,000
Las Vegas	24,000
Dallas	26,000
San Diego	38,000

上圖的意思是將兩個小表合並時, 對於 San Diego, 有兩種連法, 一種是依按之前的表, 將 user3 的兩個 San Diego 連到 4500, 將 user12 的 San Diego 連到 38000. 另一種是反過來的(如圖中綠線所示), 將 user3 的兩個 San Diego 連到 38000, 將 user12 的 San Diego 連到 45000. 這兩種連法都會被包含进 合並後的表 中. So there are two additional tuples added to the result of the joint.... So in other words, when we join these two together, we actually create three additional tuples into RegularUser table that were not there before. This phenomenon is called information loss. You may say: come on, that's a bad name for it because I got more information. Well, getting more information that's not reflecting fact of reality is actually information loss, because now you gonna have three additional rows here and those rows do not relfect facts in reality. Therefore, you have lost your ability to distinguish between the true facts that you started out with and the false facts that were inadvertently(不注意地) added. This is called information loss.

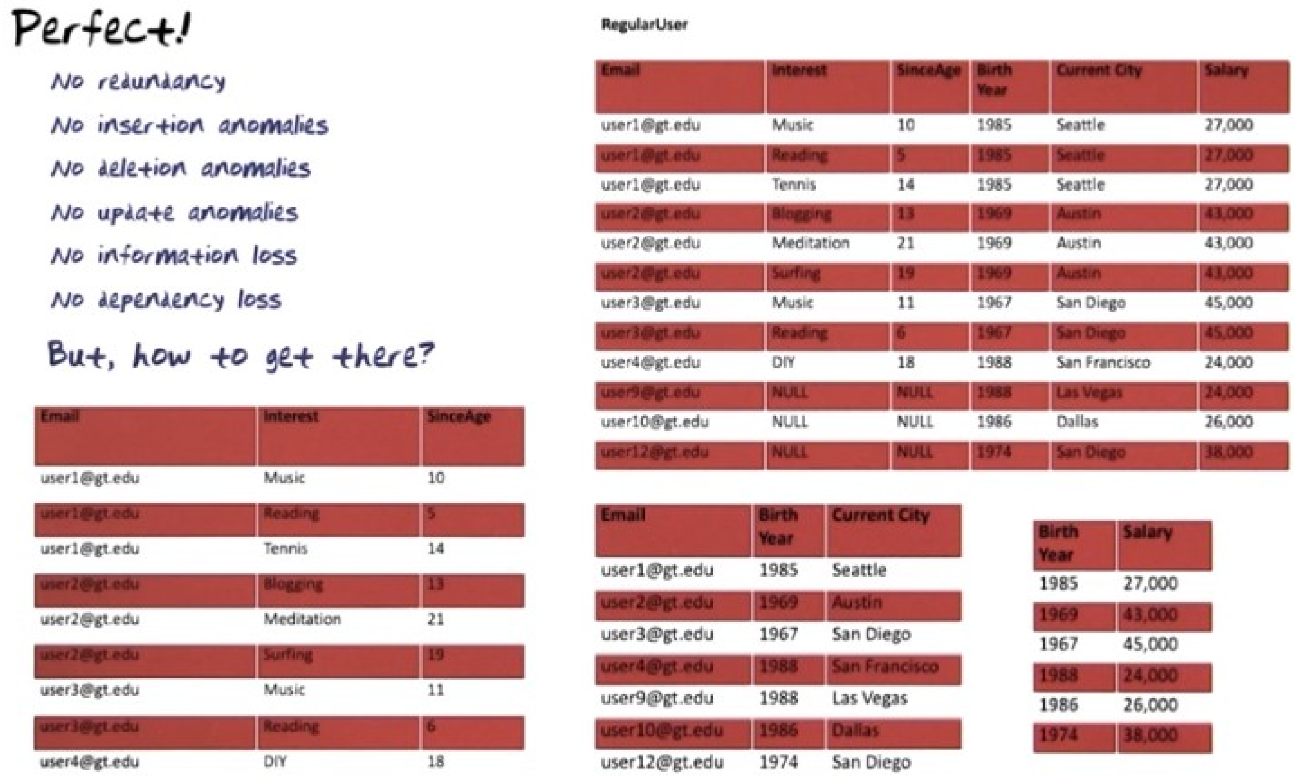


Email	Interest	SinceAge	Birth Year	Current City
user1@gt.edu	Music	10	1985	Seattle
user1@gt.edu	Reading	5	1985	Seattle
user1@gt.edu	Tennis	14	1985	Seattle
user2@gt.edu	Blogging	13	1969	Austin
user2@gt.edu	Meditation	21	1969	Austin
user2@gt.edu	Surfing	19	1969	Austin
user3@gt.edu	Music	11	1967	San Diego
user3@gt.edu	Reading	6	1967	San Diego
user4@gt.edu	DIY	18	1988	San Francisco
user9@gt.edu	NULL	NULL	1988	Las Vegas
user10@gt.edu	NULL	NULL	1986	Dallas
user12@gt.edu	NULL	NULL	1974	San Diego

Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

Current City	Salary
Seattle	27,000
Austin	43,000
San Diego	45,000
San Francisco	24,000
San Diego	24,000
College Park	44,000
Las Vegas	24,000
Dallas	26,000
San Diego	38,000

The above figure: We had a functional dependency from Email to Salary. Since Email and salary do not appear together in any of these two new tables. We cannot enforce it within a table. Similary, BirthYear determined Salary. Since the two do not coexist in a table, we cannot enforce them in the table.



Perfect!

No redundancy
No insertion anomalies
No deletion anomalies
No update anomalies
No information loss
No dependency loss

But, how to get there?

RegularUser

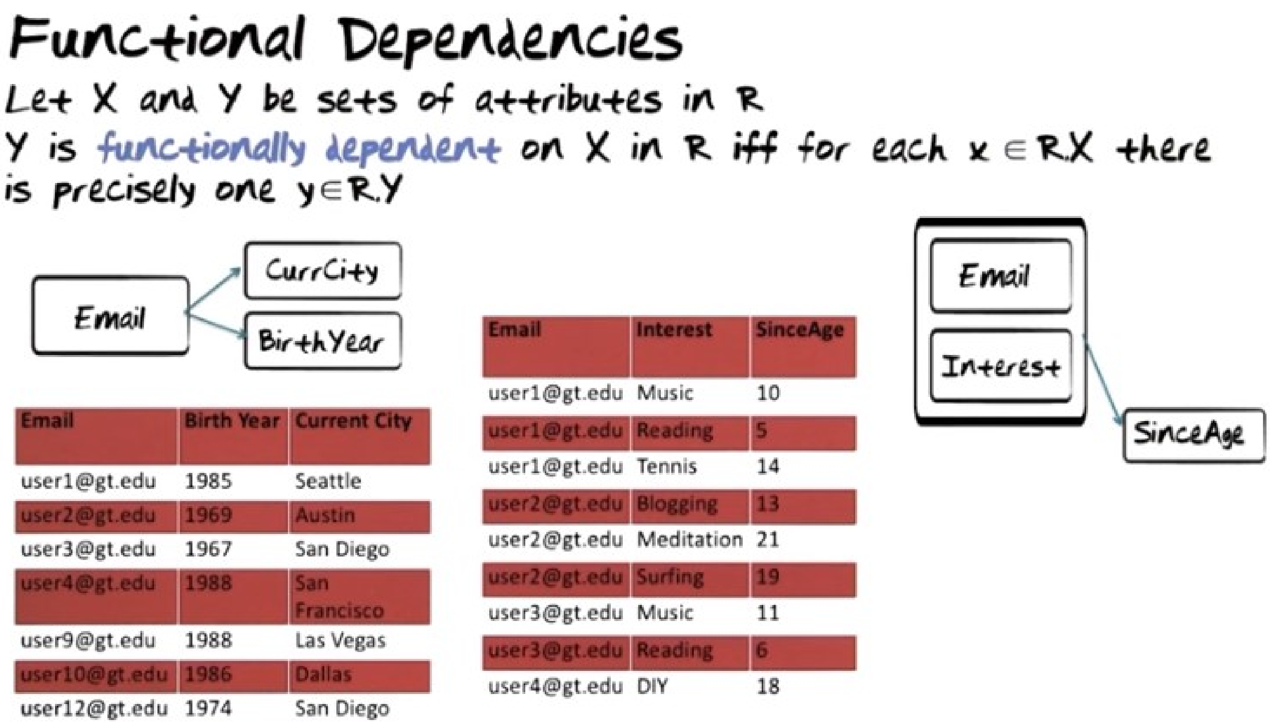
Email	Interest	SinceAge	Birth Year	Current City	Salary
user1@gt.edu	Music	10	1985	Seattle	27,000
user1@gt.edu	Reading	5	1985	Seattle	27,000
user1@gt.edu	Tennis	14	1985	Seattle	27,000
user2@gt.edu	Blogging	13	1969	Austin	43,000
user2@gt.edu	Meditation	21	1969	Austin	43,000
user2@gt.edu	Surfing	19	1969	Austin	43,000
user3@gt.edu	Music	11	1967	San Diego	45,000
user3@gt.edu	Reading	6	1967	San Diego	45,000
user4@gt.edu	DIY	18	1988	San Francisco	24,000
user9@gt.edu	NULL	NULL	1988	Las Vegas	24,000
user10@gt.edu	NULL	NULL	1986	Dallas	26,000
user12@gt.edu	NULL	NULL	1974	San Diego	38,000

Email	Interest	SinceAge
user1@gt.edu	Music	10
user1@gt.edu	Reading	5
user1@gt.edu	Tennis	14
user2@gt.edu	Blogging	13
user2@gt.edu	Meditation	21
user2@gt.edu	Surfing	19
user3@gt.edu	Music	11
user3@gt.edu	Reading	6
user4@gt.edu	DIY	18

Email	Birth Year	Current City
user1@gt.edu	1985	Seattle
user2@gt.edu	1969	Austin
user3@gt.edu	1967	San Diego
user4@gt.edu	1988	San Francisco
user9@gt.edu	1988	Las Vegas
user10@gt.edu	1986	Dallas
user12@gt.edu	1974	San Diego

Birth Year	Salary
1985	27,000
1969	43,000
1967	45,000
1988	24,000
1986	26,000
1974	38,000

The above figure: Big question is: 我們怎樣才能弄出上面那樣的 perfect decomposition?



Functional Dependencies

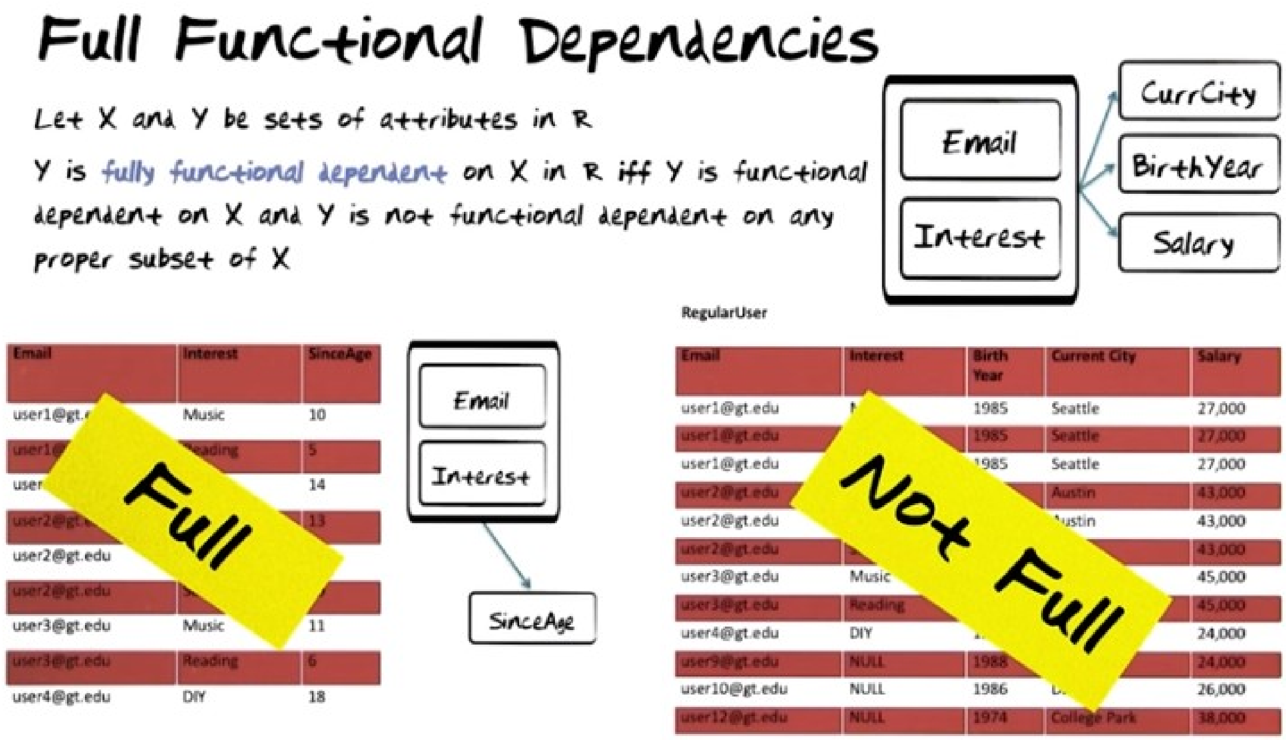
Let X and Y be sets of attributes in R

Y is functionally dependent on X in R iff for each x ∈ R.X there is precisely one y∈R.Y

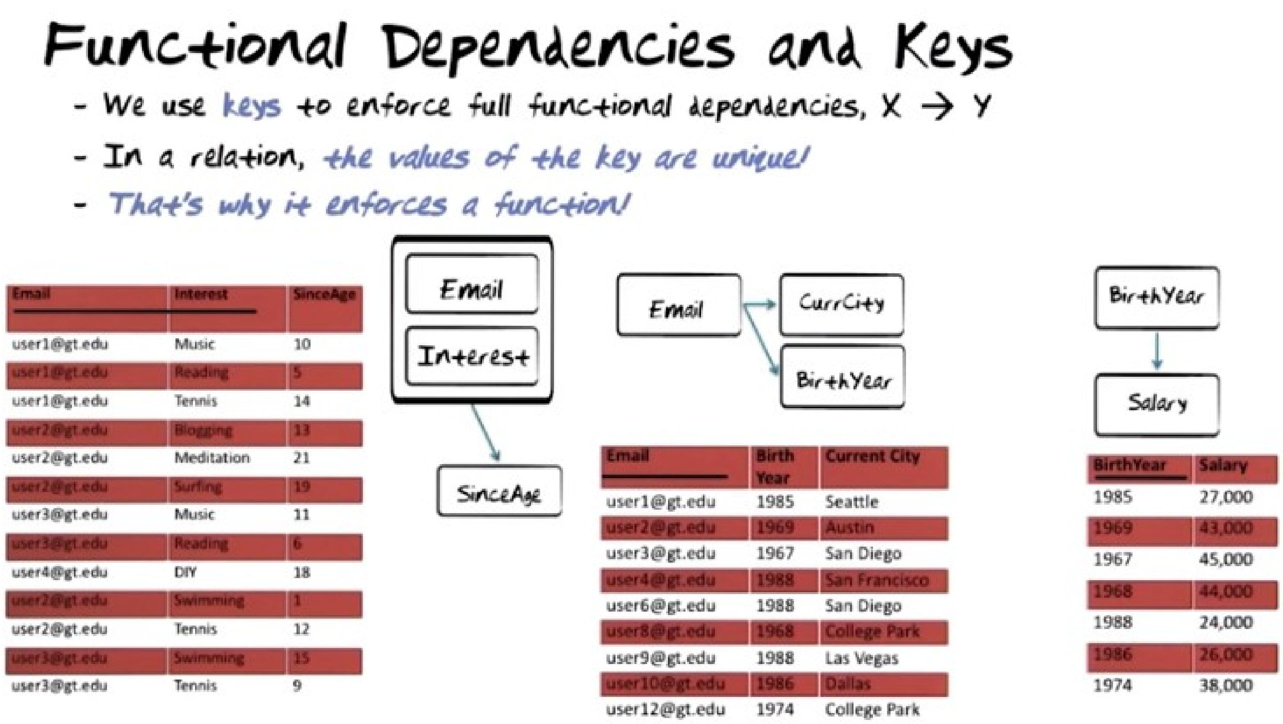
Email	Birth Year	Current City
user1@gt.edu	1985	Seattle
user2@gt.edu	1969	Austin
user3@gt.edu	1967	San Diego
user4@gt.edu	1988	San Francisco
user9@gt.edu	1988	Las Vegas
user10@gt.edu	1986	Dallas
user12@gt.edu	1974	San Diego

Email	Interest	SinceAge
user1@gt.edu	Music	10
user1@gt.edu	Reading	5
user1@gt.edu	Tennis	14
user2@gt.edu	Blogging	13
user2@gt.edu	Meditation	21
user2@gt.edu	Surfing	19
user3@gt.edu	Music	11
user3@gt.edu	Reading	6
user4@gt.edu	DIY	18

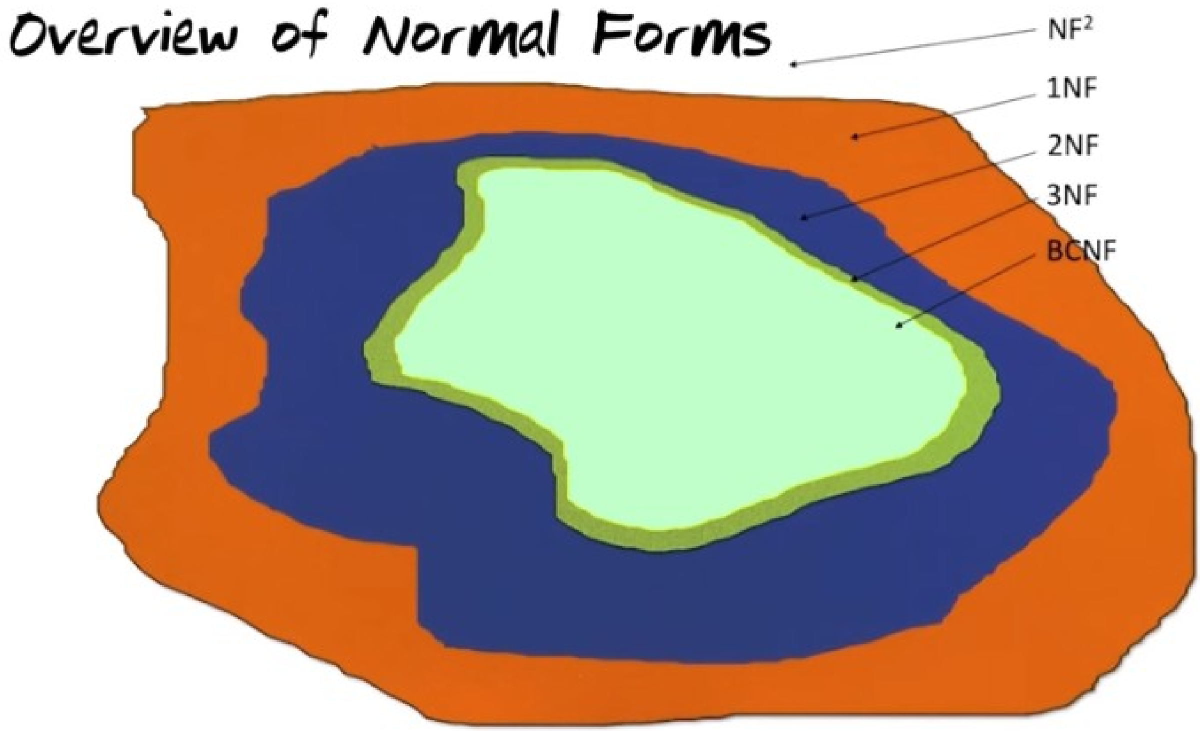
上圖中的 iff 是 if and only if 的意思. MO: Functional dependencies 的定義其實就是函數的定義, 即 y 是 x 的函數. 所以它叫 Functional dependencies.



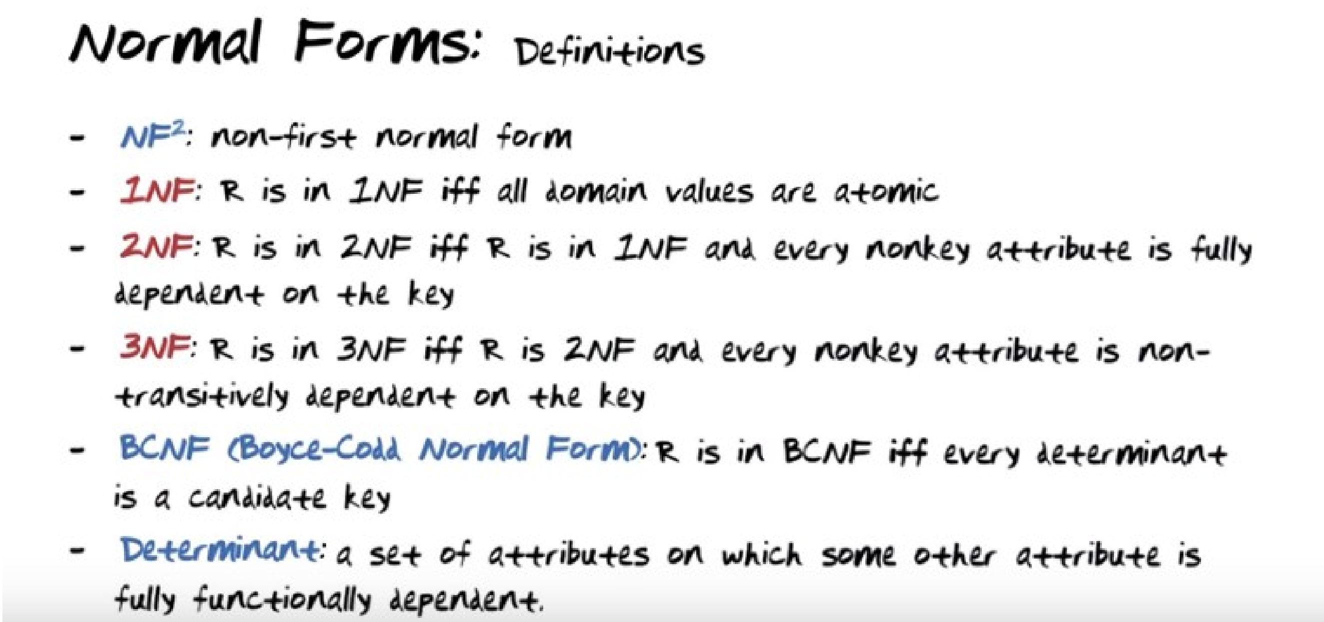
上圖中右邊, Current City is not fully functional dependent on Email and Interest. Why not? Because it's dependent on email alone. You do not need email and interest together to determine a city.



The above figure: So back to the question, then how do you enforce functional dependencies in a relation? The trick simply is that you use keys. So if you want to enforce a functional dependency that X determines Y, then all you need to do is you need to make X the key in that relation.



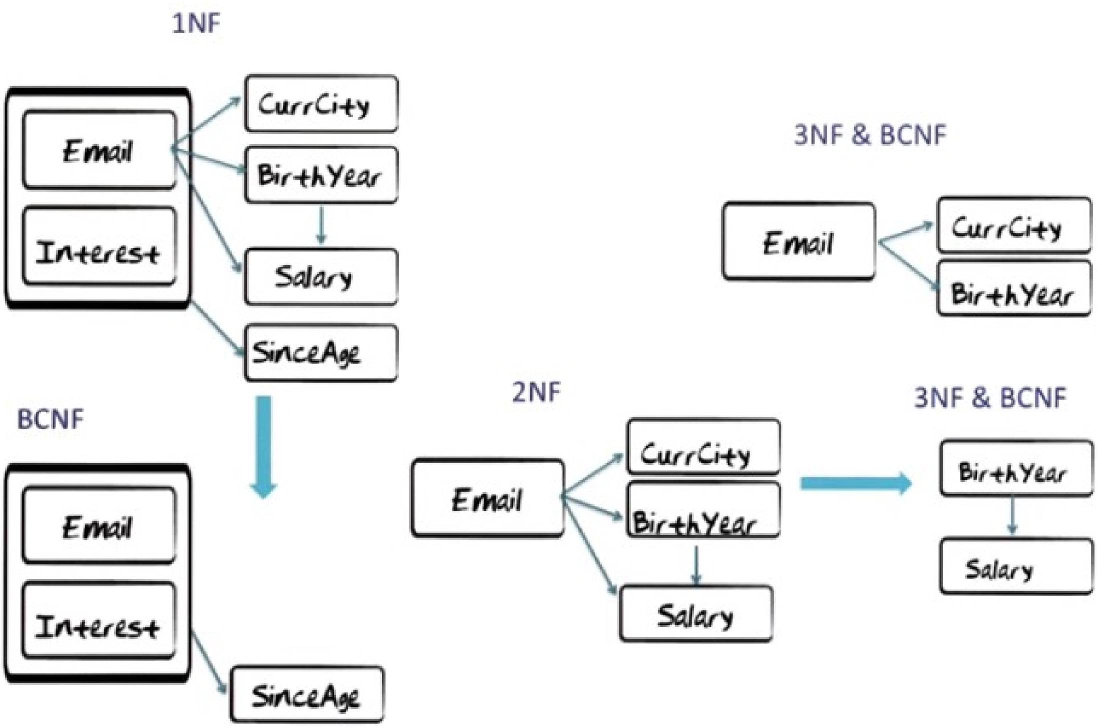
The above figure: <u>Non first normal form(NF^2)</u>之意思見 前面的 haha 處. The <u>Boyce-Codd normal form (BCNF</u>, 其意思見下) relations are the relations we are aiming at.... As illurstrated by this small small number of relations that are <u>third normal form(3NF</u>, 其意思見下) but not in BCNF. It is theoretically possible that you have a relation in 3NF that's not also in BCNF. However, in practice what happens when you normalize relations from second normal form to 3NF is that you get lucky and you actually end up in where we want to be, namely BCNF.



注意上圖中的比如 2NF, 是指某<u>一個</u>表是不是 2NF. 上圖的定義太 formal, 下圖的通俗定義更好懂, 但上圖也

有用, 比如若有 transitive dependency, 則最多只能是 2NF, 不會是 3NF. We defined a relation as a data structure where the domain values are pulled from set of atomic values. So in other words all realtions are automatically born in 1NF. Non-transitively: 非可遞.

> "All attributes must depend on the key (1NF), the whole key (2NF), and nothing but the key (3NF), so help me Codd!" [Kent, 83; Diehr, 89]



The above figure: so let's take a look at how we can work through these normal forms (即如何把之前的那個表拆成一堆 BCNF 表). 中間那個是 2NF, 是因為 they are transitive dependencies: Email determines birth year, which in turn determines salary. 右上那個是 BCNF, 是因為 the only determinant in this relation is Email, and Email of course gonna have to be the key to enforce those two functional dependencies. 同理, 右下那個也是 BCNF.

How to Compute with Functional Dependencies: Armstrong's rules

reflexivity: if Y is part of X, then X→Y

Email, Interest → Interest

augmentation: if X→Y, then WX→WY

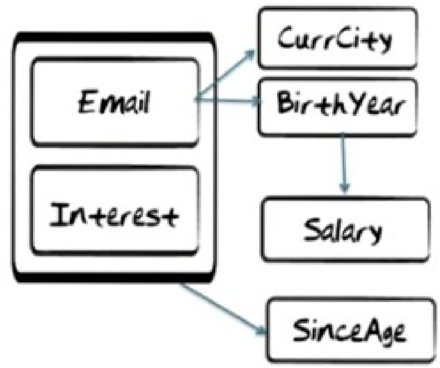
If Email → BirthYear, then

Email, Interest→ BirthYear, Interest

transitivity: if X→Y and Y→Z, then X→Z

Email → BirthYear and BirthYear → Salary, then

Email → Salary



The above figure: to make sure that we do not lose information and that we preserve the functional dependencies when we decompose relations, we need to be able to compute well-meaning(音). The rules for doing that are called Armstrong's rules.

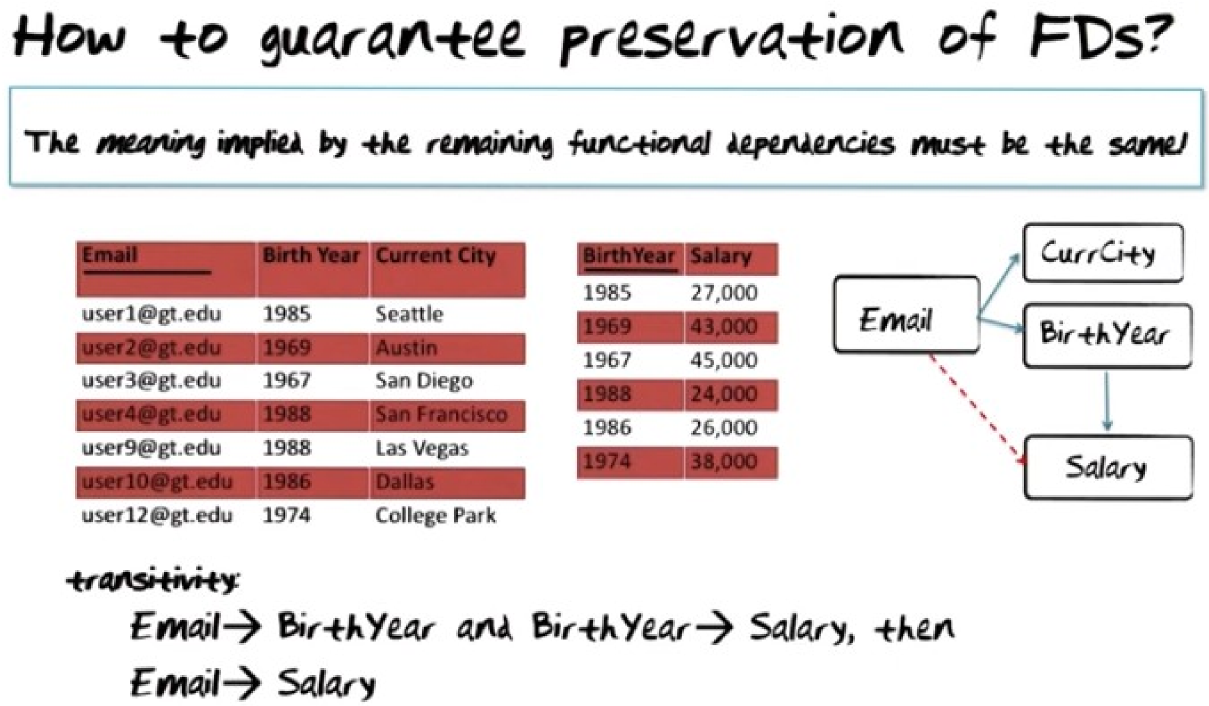
How to guarantee lossless joins?

The join field must be a key in at least one of the relations!

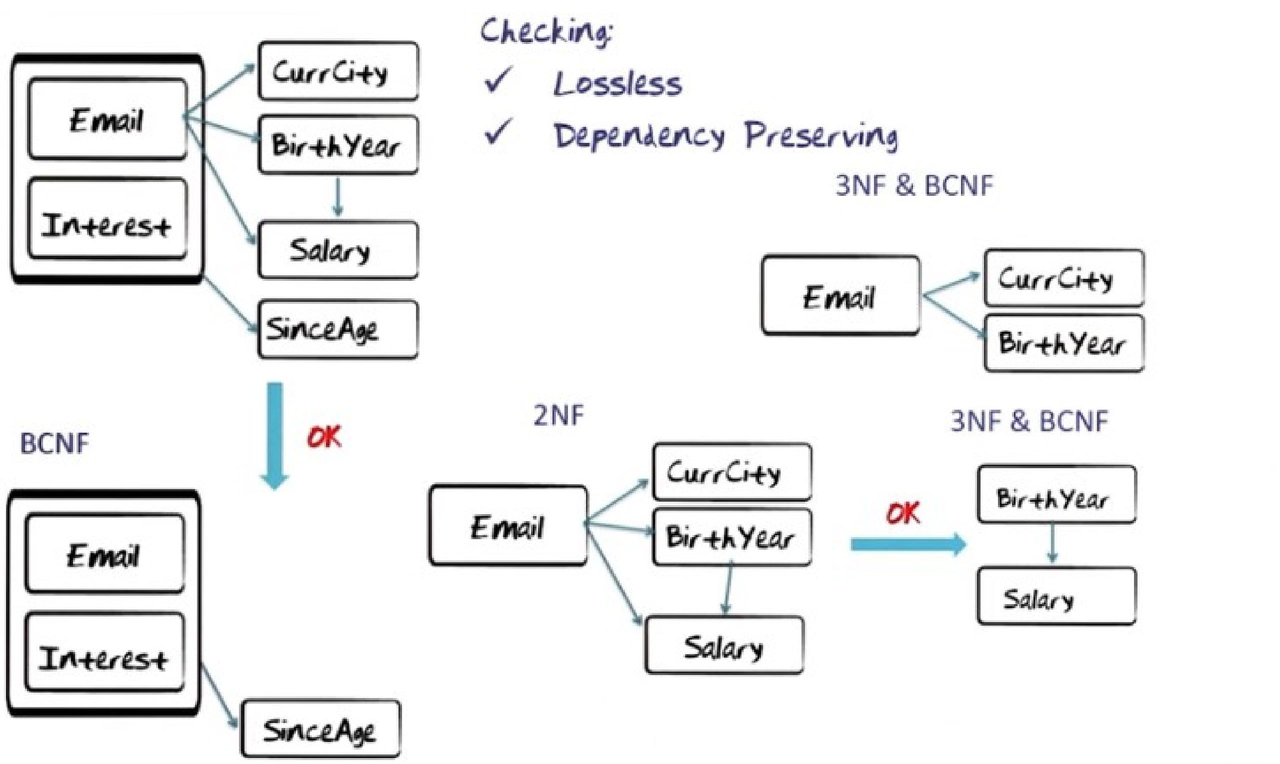
Email	Interest	SinceAge
user1@gt.edu	Music	10
user1@gt.edu	Reading	5
user1@gt.edu	Tennis	14
user2@gt.edu	Blogging	13
user2@gt.edu	Meditation	21
user2@gt.edu	Surfing	19
user3@gt.edu	Music	11
user3@gt.edu	Reading	6
user4@gt.edu	DIY	18

Email	Birth Year	Current City
user1@gt.edu	1985	Seattle
user2@gt.edu	1969	Austin
user3@gt.edu	1967	San Diego
user4@gt.edu	1988	San Francisco
user9@gt.edu	1988	Las Vegas
user10@gt.edu	1986	Dallas
user12@gt.edu	1974	College Park

The above figure: when you look at this decomposition here of relation in two relations, then the joint field (joint 是指, 我們最開始將初始的表拆成了兩個表, 這兩個表合成為初始的表時, 需要 joint, 這就是這裡提到的 joint) between these two relations obviously is Email. If Email is a key in one of the two relations as it is here, then we are guaranteed not to loose information from doing this decomposition. In other words, when we joint these two relations together again over Email, then we are guaranteed to not create a disen(音) tuples that were not there in the relation we started out with.



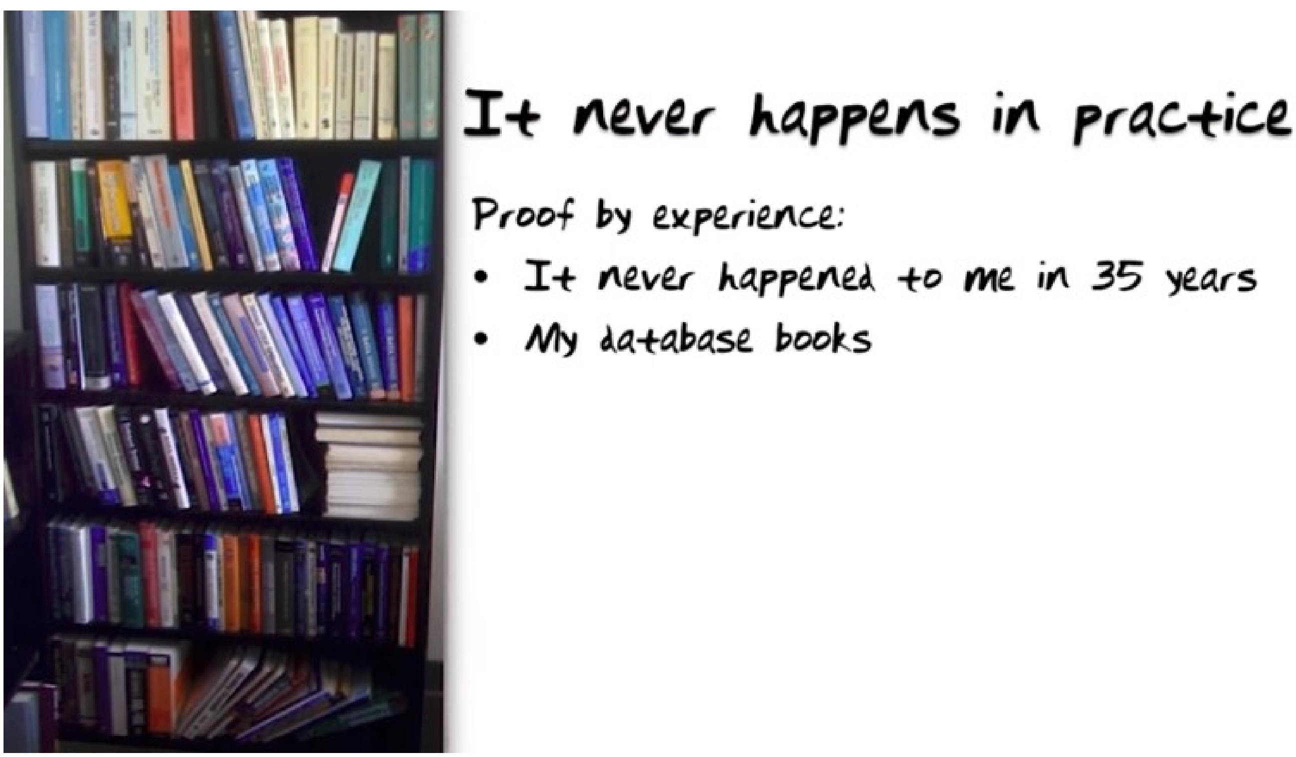
上圖的 FD 即 functional dependency. 方框中完整的句子為: the meaning implied by the remaining functional dependencies must be the same as the meaning that was implied by the original set.



上圖的操作是兩步, 第一步是將初始的表拆為下面那個 BCNF 和 2NF 表, 第二步是將那個 2NF 表拆為右邊的

那兩個 3NF&BCNF 表. 每一步操作時, 都要 check 該步是否 OK(即滿足 lossless 和 dependency preserving). 即整個 normalize 的過程中, 每一次拆表都要滿足 lossless 和 dependency preserving.

上圖第一點是強調 can only be decomposed to 3NF.



As illurstrated by the Venn diagram this(沒說 this 指甚麼, 但可從下文知它指甚麼) really never happens in practice. I'm gonna show you a proof by experience. Proof has two steps to it. Number one, it never happened to me in 35 years. I've just designed a lot of database in industry, in government, and in the university, and it never happened to me. As a second part of the proof, take a look at one of the bookcases in my office. So this one book case contains database textbooks only. Every single one of these textbooks has in it a chapter on database normalization. They will show you 1NF, 2NF, 3NF and

BCNF. And then coming to the bottom of the right page, it says there does exist relations that can only be decomposed to 3NF and not BCNF. With shaking fingers you turn to page to see such an example, and they changed the example on you. Why is that the case? Because it takes a really sick brain to construct an example like that. So be confident if you follow the steps and bring relations through first second third normal form, you will be lucky and you will end up in BCNF in practice.