Dependencies

Remy Wang

April 2026

A key difference between a DB and a spreadsheet is that, in a DB, data is stored in multiple, interconnected tables. This raises the question of how to organize data across tables.

Let's consider our favorite pets again:

ID	name	age	food
1	Casa	9	chicken
1	Casa	9	grass
2	Kira	7	fish
2	Kira	7	grass

Here we have a new column storing their favorite food. But because each of them likes more than one kinds of food, we need two rows per cat, duplicating their name and age.

You may be tempted to replace the duplicated data with `NULL`s to save space:

ID	name	age	food
1	Casa	9	chicken
1	NULL	NULL	grass
2	Kira	7	fish
2	NULL	NULL	grass

But now the natural query looking for cats who like grass breaks!

```
SELECT name
  FROM pets
 WHERE food = 'grass'
```

So let's go back to duplicating the entries.

ID	name	age	food
1	Casa	9	chicken
1	Casa	9	grass
2	Kira	7	fish
2	Kira	7	grass

Can you think of a way to break up the data into multiple tables to remove the redundancy?

We can have two tables, one storing name and age, and another storing food:

ID	name	age
1	Casa	9
2	Kira	7

ID	food
1	chicken
1	grass
2	fish
2	grass

Now *name* and *age* are no longer repeated! You may notice the duplicated IDs in the second table, but that's unavoidable, and an integer takes up little space.[1]

[1] There's also a duplicate of *grass* which can be eliminated. Try to do that after you learn about decomposition.

Taking a step back, we see the root cause of the data redundancy is that although ID *uniquely determines* name and age, we repeat the information for every occurance of ID:

ID	name	age	food
1	Casa	9	chicken
1	Casa	9	grass
2	Kira	7	fish
2	Kira	7	grass

Formally, there is a **functional dependency** (FD) ID $\rightarrow$ name, as well as ID $\rightarrow$ age.

In general, an FD $X \rightarrow Y$ (where X and Y are sets of attributes) holds for a table t, if the value of X uniquely determines the value of Y.

In our example, we also have ID $\rightarrow$ name, age, but ID $\rightarrow$ food does not hold.

For an example with multiple attribute on the left, consider a table storing GPS coordinates and zipcode:

long.	lat.	zip
34N	118W	90095
47N	122W	98195

Neither the longitude nor the latitude along uniquely determines the zipcode, but they do together, so we have long., lat. $\rightarrow$ zip

There's also some "trivial" FDs:

- For any attribute x, $x \to x$ holds
- For any sets $Y \subseteq X$, $X \to Y$ holds

We say k is a *key* if $k \to x$ holds for every attribute x in the table.[2] In our example, ID is a key after we break the table up, but it was not when we had one big table.

ID	name	age
1	Casa	9
2	Kira	7

ID	food
1	chicken
1	grass
2	fish
2	grass

[2] rigorously, a key must be *minimal*, otherwise it's a *superkey*.

"Breaking the table up" is formally known as *decomposition* or *normalization*, and the goal is to exploit FDs to remove redundancy.

In our example, we decomposed with the FD ID $\rightarrow$ name, age.

In general we want to decompose so that for every FD $X \rightarrow Y$, X is a key.[3][4]

[3] Or $X \rightarrow Y$ is *trivial*, meaning $Y \subseteq X$.

[4] More rigorously, X is a *superkey* meaning it contains a key.

But we could have decomposed in 2 steps: first with ID $\rightarrow$ name, then with ID $\rightarrow$ age:

ID	name
1	Casa
2	Kira

ID	age
1	9
2	7

ID	food
1	chicken
1	grass
2	fish
2	grass

This however takes up more space because the ID column is now duplicated.

Intuitively, when we break up, we want to take as much as possible, i.e., we want to decompose with an FD that has many attributes on the right.

To do this, we compute the *FD closure*: given a set of FDs Φ, the *closure* X^+ of a set X is the largest set such that $X \to X^+$ follows from Φ

Here, an FD ϕ *follows from* Φ if ϕ holds in any table t satisfying Φ.

That's all very abstract, so let's look at some cats. Suppose we have a table with these columns:

name	birth day	age	astrology sign	personality
...	...	...	...	...

Then we have:

$$\text{bday} \to \text{age}$$
$$\text{bday} \to \text{astro}$$
$$\text{astro} \to \text{personality}$$

We can deduce bday $\to$ personality by *composing* the FDs bday $\to$ astro and astro $\to$ personality.

In general, we can deduce additional FDs using the "Armstrong axioms":

- **Reflexivity**: $X \to Y$ if $Y \subseteq X$
- **Augmentation**: if $X \to Y$, then $XZ \to YZ$
- **Transitivity**: if $X \to Y$ and $Y \to Z$, then $X \to Z$

We've seen examples of reflexivity and transitivity. Augmentation is also intuitive: from bday $\to$ age we can conclude
bday, name $\to$ age, name.

Actually, you don't need to remember the rules. There's a even simpler way to directly compute X^+. The intuition can be explained with the *tableux method*

Given the FDs bday $\rightarrow$ age, bday $\rightarrow$ astro, and astro $\rightarrow$ pers., let's compute bday$^+$. First we write down a table with just two rows:

name	birth day	age	astrology sign	personality
Casa	3/20/2017	9	Pisces	dreamy
?	3/20/2017	?	?	?

Then, we *apply* each FD to see if we can infer the unknown values.

Step 1: Apply bday $\rightarrow$ age. Both rows share the same birthday, so the unknown age must equal 9.

name	birth day	age	astrology sign	personality
Casa	3/20/2017	9	Pisces	dreamy
?	3/20/2017	**9**	?	?

Step 2: Apply bday $\rightarrow$ astro. Both rows share the same birthday, so the unknown astrology sign must equal Pisces.

name	birth day	age	astrology sign	personality
Casa	3/20/2017	9	Pisces	dreamy
?	3/20/2017	9	**Pisces**	?

Step 3: Apply astro $\rightarrow$ pers. Both rows now share the same astrology sign, so the unknown personality must equal dreamy.

name	birth day	age	astrology sign	personality
Casa	3/20/2017	9	Pisces	dreamy
?	3/20/2017	9	Pisces	**dreamy**

No more FDs can fill in new values — we've reached a *fixpoint*. The final result is:

name	birth day	age	astrology sign	personality
Casa	3/20/2017	9	Pisces	dreamy
?	3/20/2017	9	Pisces	dreamy

This tell us the closure $\text{bday}^+ = \{\text{bday, age, astro., pers.}\}$, i.e., birthday uniquely determines all of these attributes.

More abstractly, the algorithm works like this:

```
def clos(FDs, V):
  C = V
  while C changes:
    for (X, Y) in FDs:
      if C contains X:
        add Y to C
  return C
```

In words: we start with the set V, then keep applying each FD to grow the set, until doing so doesn't change it anymore.

We can use the closure algorithm to check if a given FD $\varphi = X \to Y$ follows from a given set of FDs Φ, denoted $\Phi \vdash \varphi$:

1. Compute the closure X^+ of X over Φ
2. Check if $Y \subseteq X^+$
3. If so, $\Phi \vdash \varphi$, otherwise $\Phi \nvdash \varphi$

That's a lot of new concepts! Let's step back and think about our motivation.

Our original table had *redundancy*, which we now understand is caused by *functional dependencies*:

ID	name	age	food
1	Casa	9	chicken
1	Casa	9	grass
2	Kira	7	fish
2	Kira	7	grass

In particular, we have FDs like ID $\rightarrow$ name where the left-hand-side ID is *not* a key.

To fix the redundancy, we break the table up according to the FDs:

ID	name	age
1	Casa	9
2	Kira	7

ID	food
1	chicken
1	grass
2	fish
2	grass

We also want to "take as much as possible" when breaking up, so we want to compute the *closure* which tells us *all* the dependent attributes.

So our algorithm so far looks like this:

- Find any FD $X \to Y$ such that $Y \not\subseteq X$ and X does not contain a key
- Compute the closure X^+ of X
- Move X^+ to a new table while keeping X in the old one

Let's also clarify some terminology:

- A single attribute k is a *key* if $k \rightarrow y$ for every other attribute y

Let's also clarify some terminology:

- A single attribute k is a *key* if $k \rightarrow y$ for every other attribute y
- We can also have multi-attribute keys (composite keys):

Let's also clarify some terminology:

- A single attribute k is a *key* if $k \rightarrow y$ for every other attribute y
- We can also have multi-attribute keys (composite keys):
  - K is a *superkey*[5] if $K \rightarrow y$ for every y

[5] superset of a key

Let's also clarify some terminology:

- A single attribute k is a *key* if $k \to y$ for every other attribute y
- We can also have multi-attribute keys (composite keys):
  - K is a *superkey*[5] if $K \to y$ for every y
  - A superkey K is a *key* if no $K' \subset K$ is a superkey, i.e. K is *minimal*

[5] superset of a key

Let's also clarify some terminology:

- A single attribute k is a *key* if $k \to y$ for every other attribute y
- We can also have multi-attribute keys (composite keys):
  - K is a *superkey*[5] if $K \to y$ for every y
  - A superkey K is a *key* if no $K' \subset K$ is a superkey, i.e. K is *minimal*

[5] superset of a key

Let's also clarify some terminology:

- A single attribute k is a *key* if $k \to y$ for every other attribute y
- We can also have multi-attribute keys (composite keys):
  - K is a *superkey*[5] if $K \to y$ for every y
  - A superkey K is a *key* if no $K' \subset K$ is a superkey, i.e. K is *minimal*

The idea is, once we have a composite key K, we can always keep adding stuff to K to get another superkey, but K is more interesting because it doesn't contain "junk".

[5] superset of a key

Let's practice this on the larger table as well:

name	birth day	age	astrology sign	personality
...	...	...	...	...

1. We have bday $\rightarrow$ age yet bday is not a key

Let's practice this on the larger table as well:

name	birth day	age	astrology sign	personality
...	...	...	...	...

1. We have bday $\rightarrow$ age yet bday is not a key
2. Compute $\text{bday}^+ = \{\text{bday}, \text{age}, \text{astro}, \text{pers.}\}$

Let's practice this on the larger table as well:

name	birth day	age	astrology sign	personality
...	...	...	...	...

1. We have bday $\rightarrow$ age yet bday is not a key
2. Compute bday$^+ = \{\text{bday}, \text{age}, \text{astro}, \text{pers.}\}$
3. Break the table into two

Let's practice this on the larger table as well:

name	birth day	age	astrology sign	personality
...	...	...	...	...

1. We have bday $\rightarrow$ age yet bday is not a key
2. Compute bday$^+ = \{\text{bday}, \text{age}, \text{astro}, \text{pers.}\}$
3. Break the table into two

Let's practice this on the larger table as well:

name	birth day	age	astrology sign	personality
...	...	...	...	...

1. We have bday $\rightarrow$ age yet bday is not a key
2. Compute bday$^+ = \{\text{bday}, \text{age}, \text{astro}, \text{pers.}\}$
3. Break the table into two

name	birth day
...	...

birth day	age	astrology sign	personality
...	...	...	...

But we're not done! In the second table, we still have redundancy:

birth day	age	astrology sign	personality
3/20/2017	9	**Pisces**	**dreamy**
2/19/2016	10	**Pisces**	**dreamy**

But we're not done! In the second table, we still have redundancy:

birth day	age	astrology sign	personality
3/20/2017	9	**Pisces**	**dreamy**
2/19/2016	10	**Pisces**	**dreamy**

To fix this, we *repeat the same process* to decompose this table

1. astro $\rightarrow$ pers. yet astro is not a key

But we're not done! In the second table, we still have redundancy:

birth day	age	astrology sign	personality
3/20/2017	9	**Pisces**	**dreamy**
2/19/2016	10	**Pisces**	**dreamy**

To fix this, we *repeat the same process* to decompose this table

1. astro $\rightarrow$ pers. yet astro is not a key
2. Compute astro$^+ = \{\text{astro}, \text{pers.}\}$

But we're not done! In the second table, we still have redundancy:

birth day	age	astrology sign	personality
3/20/2017	9	**Pisces**	**dreamy**
2/19/2016	10	**Pisces**	**dreamy**

To fix this, we *repeat the same process* to decompose this table

1. astro $\rightarrow$ pers. yet astro is not a key
2. Compute astro$^+ = \{\text{astro}, \text{pers.}\}$
3. *Factor out* $\{\text{astro}, \text{pers.}\}$

We end up with 3 tables that cannot be decomposed further:

name	birth day
...	...

birth day	age	astrology sign
...	...	...

astrology sign	personality
...	...

The general decomposition algorithm is also a *fixpoint* algorithm:

```
def decompose(R):
  S = attributes of R
  find nontrivial X->Y s.t. X contains no key
  if not found:
    return
  else:
    C = clos(X)
    break R into R1(C), R2(S - (C - X))
    decompose(R1)
    decompose(R2)
```

At the end of this process, the tables are in *Boyce-Codd Normal Form*, meaning that any FD $X \to Y$ that still hold in any table will satisfy either:

- $Y \subseteq X$ ("trivial" FD)
- Some $K \subseteq X$ is a key (X is a *superkey*)

So far, we've been focusing on redudancy caused by functional dependencies, which can be addressed by decomposing according to the FDs.

However, this can at most save us $O(N)$ space (why?). Let's now consider a more severe form of redundancy caused by *independence*.

Suppose we have a single table storing the favorite toy and food for each cat:

name	toy	food
casa	ball	chicken
casa	bag	chicken
casa	TP	chicken
casa	ball	grass
casa	bag	grass
casa	TP	grass

The issue here is that **toy** is independent of **food** for each cat, yet we store them in the same table.

The solution is once again breaking the table up:

name	toy
casa	ball
casa	bag
casa	TP

name	food
casa	chicken
casa	grass

Formally, the redundancy can be described with (conditional) independence. For that, let's review some basic probability theory.

A *distribution* assigns a probability to each event. If we draw a row uniformly at random from our data, there's $1/2$ chance we get a row with food $=$ chicken or food $=$ grass, So the distributions over **food** and **toy** are:

food	p
chicken	$1/2$
grass	$1/2$

toy	p
ball	$1/3$
bag	$1/3$
TP	$1/3$

A *joint distribution* describes the probablities of different types of events occuring at the same time. If we draw a random row, the probability of toy $=$ ball $\wedge$ food $=$ chicken is $1/6$, and so is for any other pairing of toy and food:

name	toy	food	p
casa	ball	chicken	$1/6$
casa	bag	chicken	$1/6$
casa	TP	chicken	$1/6$
casa	ball	grass	$1/6$
casa	bag	grass	$1/6$
casa	TP	grass	$1/6$

The *conditional* distribution focuses a "slice" of the larger distribution. For example, $P(\text{toy}|\text{food} = \text{chicken})$ is:

toy	food	p
ball	chicken	$1/3$
bag	chicken	$1/3$
TP	chicken	$1/3$

Two things happened to obtain this distribution:

1. We are focusing only on the rows where food = chicken
2. We *re-normalized* the probabilities so that they sum to 1

Two types of events are *independent* if all conditional distributions agree with the unconditional ones:

$$\forall b : P(A|B = b) = P(A)$$

In the example above, $P(\text{toy}) = P(\text{toy}|\text{food} = \text{chicken})$, so knowing the cat likes chicken doesn't tell us anything about the toy preference.

Note that this also needs to hold for food = grass to establish the independence.

Another way to check independence is to see if the joint distribution is simply a product of the single ones[6]:

$$P(\text{toy}, \text{food}) = P(\text{toy}) \times P(\text{food})$$

[6] A.k.a. marginal distributions.

If you are traumatized by probabilities, here's a different way to understand independence. To check if toy is independent from food, first `GROUP BY` one of them, say food:

food	toy
chicken	ball
chicken	bag
chicken	TP

food	toy
grass	ball
grass	bag
grass	TP

We see that across different food groups, the set of different values for toy remains the same. This tells us that the toy preference is not affected by the food preference.

You can also see that the large table is the *cartesian product* of the small ones:

toy	food	p
ball	chicken	
bag	chicken	
TP	chicken	
ball	grass	
bag	grass	
TP	grass	

toy
ball
bag
TP

food
chicken
grass

This is why redundancy caused by independence costs a lot more space than dependencies.

But we're not done yet. I've been having some relationship issues with my cats lately, so I gathered some data:

love_c	love_k
5	4
5	5
4	5
4	4
1	2
1	1
2	1
2	2

It looks like Kira's love towards me is depedent upon Casa's love! How can this be?

It turns out they love me more when they are hungry:

hungry?	love_c	love_k
y	5	4
y	5	5
y	4	5
y	4	4
n	1	2
n	1	1
n	2	1
n	2	2

In statistics, hungry is called a *confounding factor*.

Now, if we *condition* on hunger, we will see the cats become independent again:

hungry?	love_c	love_k
y	5	4
y	5	5
y	4	5
y	4	4

hungry?	love_c	love_k
n	1	2
n	1	1
n	2	1
n	2	2

Formally, we say two types of events X, Y are *conditionally independent* on another type Z, if they are independent for every case of Z, written $X \perp Y \mid Z$.

When we have conditional independence in a table, we can break it up along the column conditioned on:

hungry?	love_c
y	5
y	4
n	1
n	2

hungry?	love_k
y	5
y	4
n	1
n	2

But we need to be careful! Suppose we have another column that somewhat depends on the rest:

hungry?	love_c	love_k	happy_r
y	5	4	2
y	5	5	1
y	4	5	2
y	4	4	3
n	1	2	2
n	1	1	1
n	2	1	2
n	2	2	3

This shows I'm happy if my cats love/hate me a little bit, but not too much.

Suppose we decompose and take out love_k:

hungry?	love_c	happy_r
y	5	?
y	4	?
n	1	?
n	2	?

Now since we have two rows with $\text{love}_c = 5$ but different happy_r, we can't collapse them any more.

But now we're in trouble: if we join these tables back together, we don't get back the original one!

hungry?	love_c	happy_r
y	5	2
y	5	1
y	4	2
y	4	3
n	1	2
n	1	1
n	2	2
n	2	3

hungry?	love_k
y	5
y	4
n	1
n	2

The reason is that, although love_c and love_k are conditionally independent, happy_r depends on both of them in a nontrivial way, and to represent this dependency faithfully, we *have* to put them in the same table.

Another example is to think about science experiments:

alt.	temp.	pressure	color
...	...	...	...

Although temperature and pressure are independent for each altitude, the color of your material depends on both, and you certainly don't want to decouple the pressure data in your records!

So we only use a conditional independence $X \perp Y \mid Z$ to decompose tables, if $X \cup Y \cup Z$ cover *all* table attributes.

This also explains the traditional definition of *multi-valued dependency*:

$$R \models Z \twoheadrightarrow X \iff R \models X \perp Y \mid Z$$

$$\text{where } Y = \Sigma(R) - (X \cup Z)$$

In words: there's a MVD from Z to X[7], if X is conditionally independent with the other attributes (Y) given Z.

[7] Or, X multi-value depends on Z