Set Theory: the Method To Database Madness

Vaidehi Joshi · Follow
Published in basecs · 11 min read · May 1, 2017

Now that we're finally a third of the way through this series, things are finally starting to come together. Sure, we know about quite a few different data structures, how they work, which ones are fast, and how certain ones are more helpful for solving specific problems than others.

But there's almost no point to knowing any of that if we don't have a sense of how they're actually used in real life. It's like learning geometry: it probably all seemed pointless until one day, you woke up and realized that you were actually going to have to calculate the square footage of a room because you wanted to re-carpet the floor! (Okay, I've never actually had to do that, but I can imagine that geometry as a concept would generally be pretty helpful if I ever have to.)

A lot of things are going to come together today, because we're going to learn about a data structure that is almost dogmatic in its theory, but incredibly ubiquitous in practice. In fact, you've probably already worked with this structure in some shape or form, and you were likely introduced to it in your middle school math class.

So, what data structure am I talking about? Why, I'm referring to the all-powerful **set**, of course!

No fear set theory

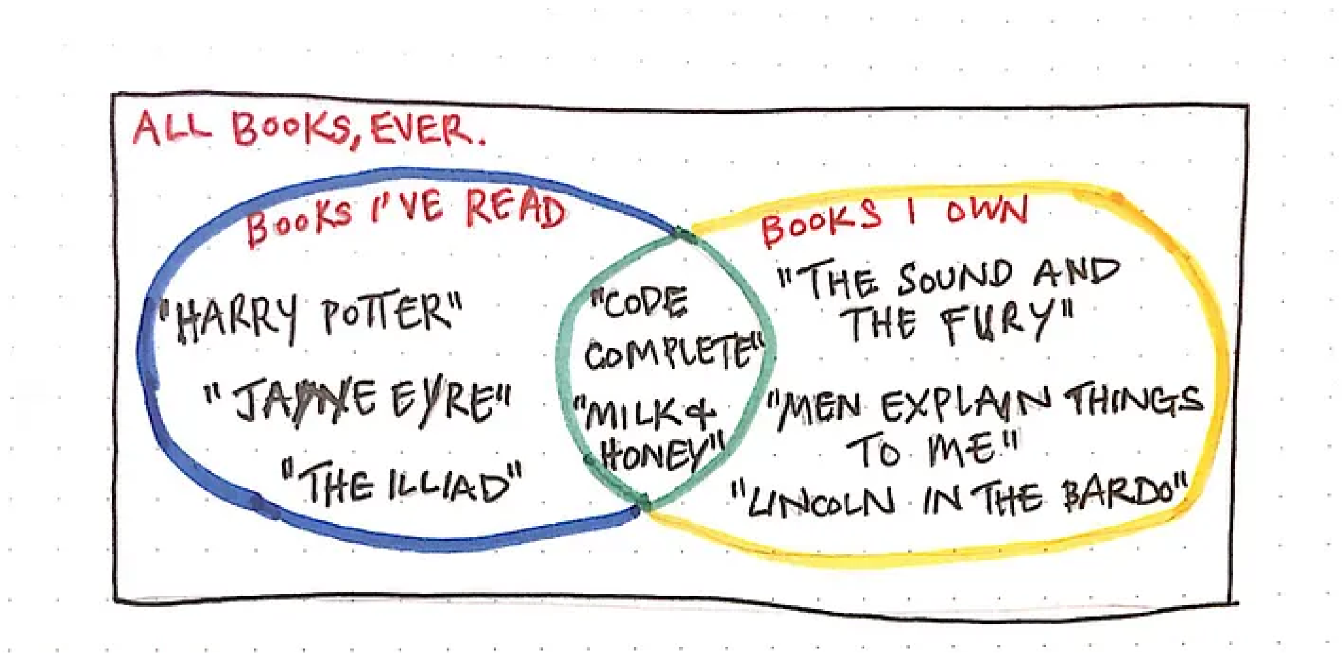
Before we get into the actual implementation of sets, we first need to understand where on earth they come from. This means it's time for us to dive into theory — set theory! But, fear not: there's a good chance you've used set theory in some capacity or another. In fact, you probably know set theory by another name: **a Venn diagram**. The Venn diagram was actually only incorporated into the "set theory curriculum" in the 1960's because it was such an effective way of illustrating simple relationships between sets.

Okay, now that we're sure that set theory won't be that scary, we'd better figure out what it is, exactly! A *set* is actually a mathematical concept, and the way that we relate sets to one another is referred to as **set theory**.

> A set is nothing more than an unordered collection of elements with absolutely no duplicates.

There are three important pieces to that definition: ***unordered, elements,*** and ***no duplicates***. Actually, those three words encompass pretty much the entire definition of a set; if we can remember that, we'll basically know everything about how sets work.

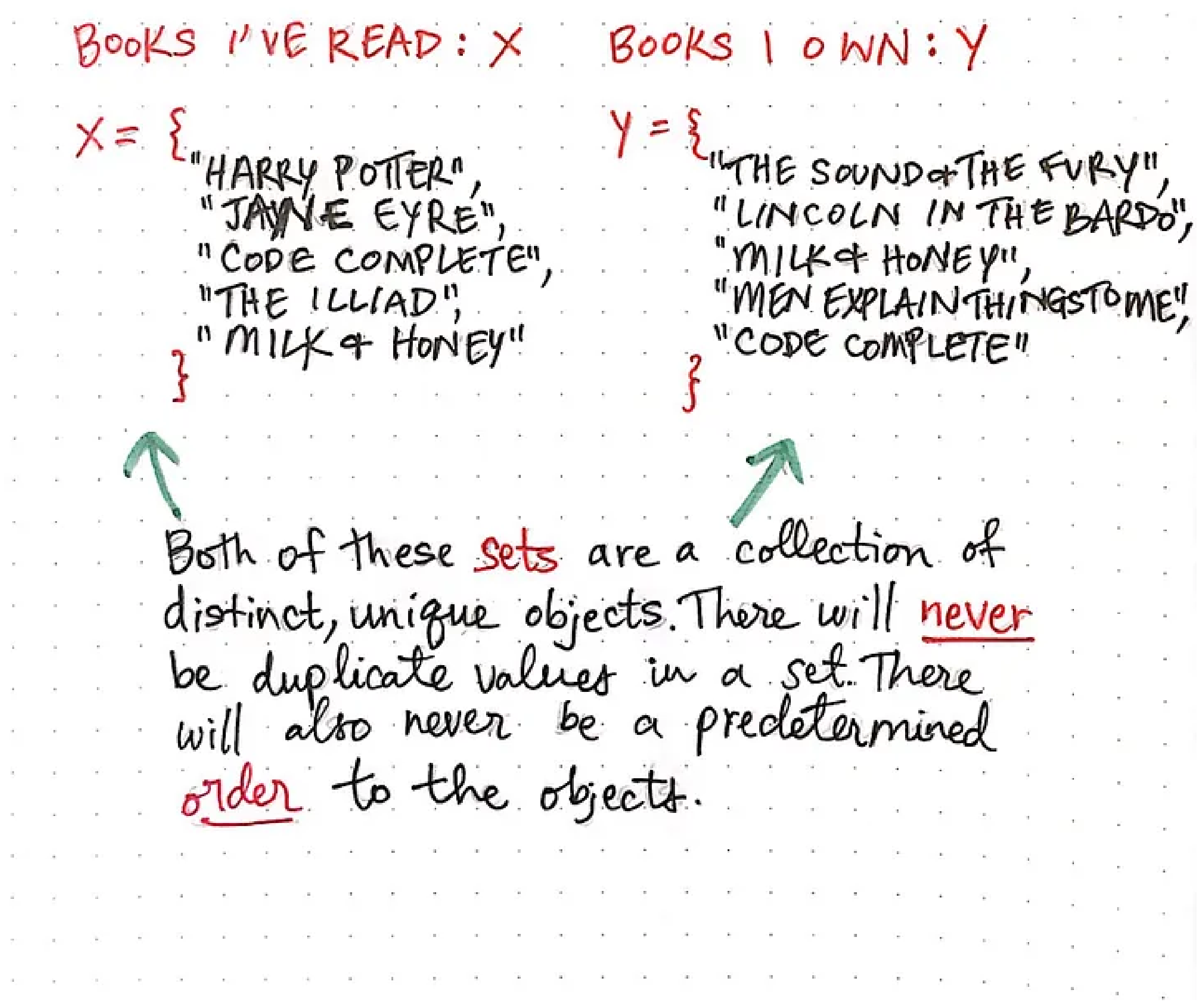
But we'll come back to why that's important in a bit. First, let's look at some sets in action. We know that sets are well-represented by Venn diagrams, so for our example, we'll look at two sets: a set of some of the books I've read, and a set of some of the books I own.



Books I've read vs. books I own

Since we're familiar with the concept of Venn diagrams, we know that the center section in green (where the two sets intersect) represents books that I have both read *and* books that I own. We also know that the two sets drawn above exist within the larger group of all the books in the world!

The Venn diagram is a good introduction to set theory, because it makes the next part a lot easier to explain. Imagine that we wanted to represent these two sets of data in some sort of structure. Well, we already know that we need to divide them into two groups: *books I've read*, and *books I own*. To make it a little easier, we'll call *books I've read* as **set X**, and *books I own* as **set Y**. Reconfigured into data structures, those two sets in the Venn diagram could also be rewritten to look like this:

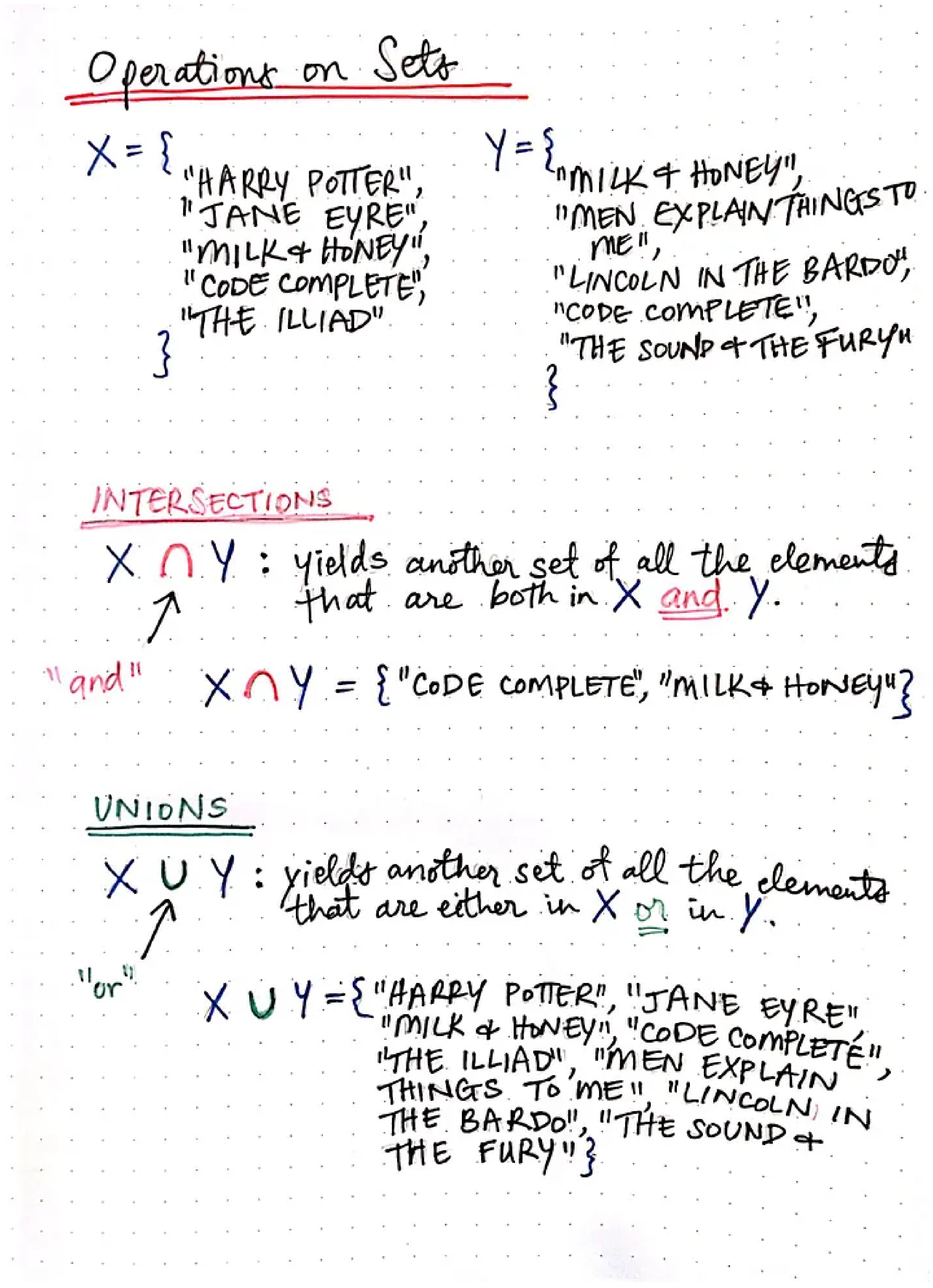

BOOKS I'VE READ: X
BOOKS I OWN: Y
X = {"HARRY POTTER", "JAYNE EYRE", "CODE COMPLETE", "THE ILLIAD", "MILK & HONEY"}
Y = {"THE SOUND & THE FURY", "LINCOLN IN THE BARDO", "MILK & HONEY", "MEN EXPLAIN THINGS TO ME", "CODE COMPLETE"}
Both of these sets are a collection of distinct, unique objects. There will never be duplicate values in a set. There will also never be a predetermined order to the objects.

Sets are a collection of distinct, unique objects that never contain duplicated values.

We'll notice that both set X and set Y look a little bit like objects or hashes: the elements don't have indexes or any sort of order. They also don't have any repeated values, which is part of what makes them a set. Remember that a set is a collection of unique, unordered objects, which means that we'll never find duplicated values within a set.

Pain-free (set) operations

So, what can we do with these sets now that we have them written in data structure format? Well, we can perform some operations on them! The two most important operations that are performed on sets are **intersections** and **unions.**



Operations on Sets

$X = \{$ "HARRY POTTER", "JANE EYRE", "MILK & HONEY", "CODE COMPLETE", "THE ILLIAD" $\}$

$Y = \{$ "MILK & HONEY", "MEN EXPLAIN THINGS TO ME", "LINCOLN IN THE BARDO", "CODE COMPLETE", "THE SOUND & THE FURY" $\}$

INTERSECTIONS

$X \cap Y$: yields another set of all the elements that are both in X and Y.

"and"

$X \cap Y = \{$ "CODE COMPLETE", "MILK & HONEY" $\}$

UNIONS

$X \cup Y$: yields another set of all the elements that are either in X or in Y.

"or"

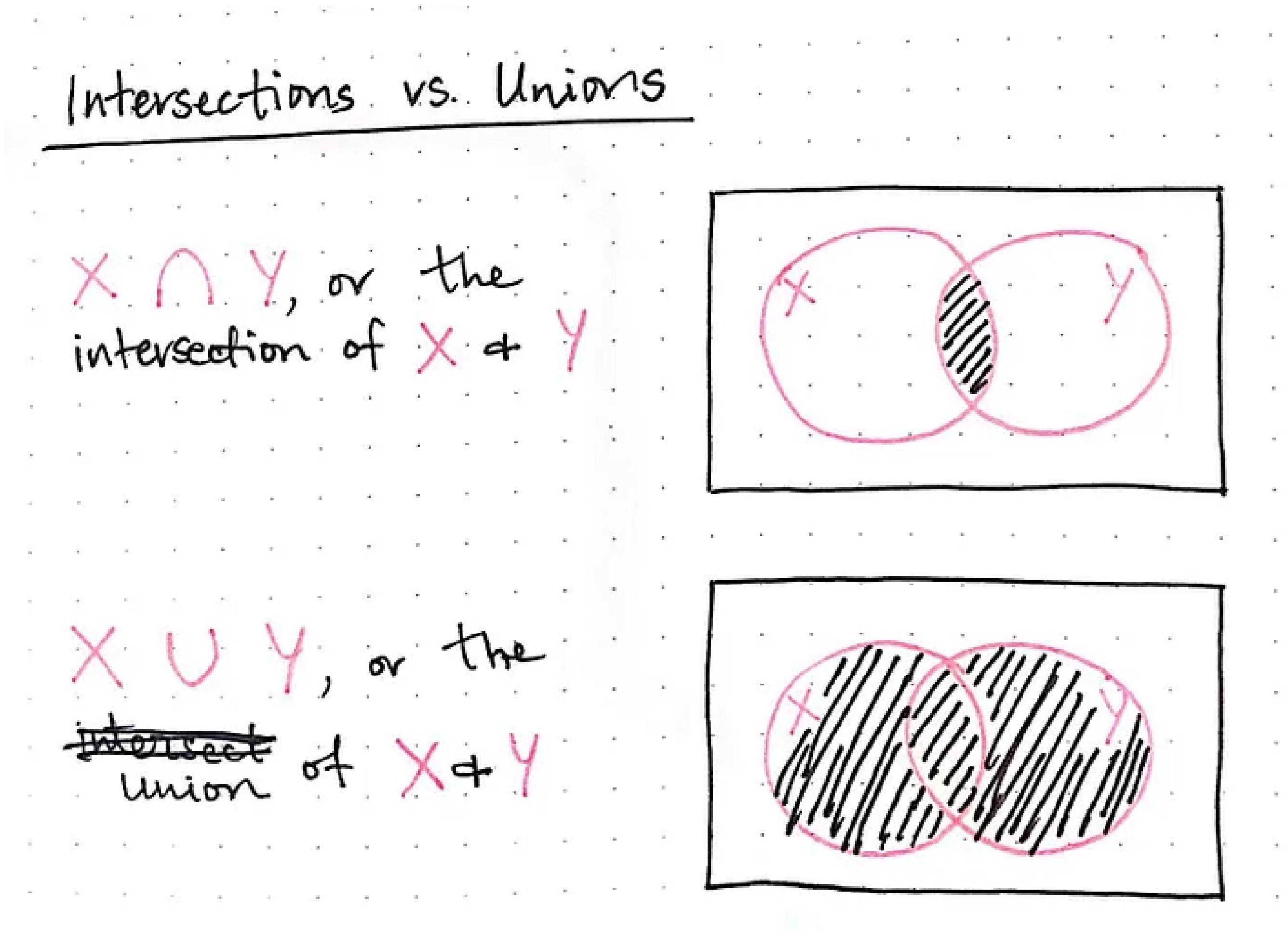
$X \cup Y = \{$ "HARRY POTTER", "JANE EYRE" "MILK & HONEY", "CODE COMPLETÉ", "THE ILLIAD", "MEN EXPLAIN THINGS TO ME", "LINCOLN IN THE BARDO", "THE SOUND & THE FURY" $\}$

Basic operations on sets

The intersection of two sets is often denoted in shorthand like this: ***X* ∩ *Y***. The intersection represents where two sets — you guessed it — *intersect*! In other words, it yields all of the elements that exist within *both* of the sets. In our example, the intersection of set X and set Y are all of the elements that exist in both of them. A good way keyword to remember how intersections work is the word **and**: the elements that exists in both X and Y. In this case, that would mean “Code Complete” and “Milk & Honey”. Even though they exist in both sets, since sets can only ever contain *unique* values, we don’t repeat them; each of these book titles exists only once in the set.

The union of two sets is often denoted in shorthand like this: ***X* ∪ *Y***. The union represents the entirety of two sets, or the two sets when they’ve been *united* together. In other words, it yields all of the elements that exist in *either* of the two sets. A good way keyword to remember how intersections work is the word **or**: the elements that exists in both X or Y. In this case, that would mean all of the eight book titles! The important thing to remember is that even though “Code Complete” and “Milk & Honey” exist in both sets, they can only ever appear once in the union of set X and set Y, since sets can only have distinct values and can never contain duplicates.

If we were to apply intersections and unions to our Venn diagram from earlier, our diagrams would look like this:


Intersections vs. Unions
X ∩ Y, or the intersection of X + Y
X
Y
X ∪ Y, or the ~~intersect~~ union of X + Y
X
Y

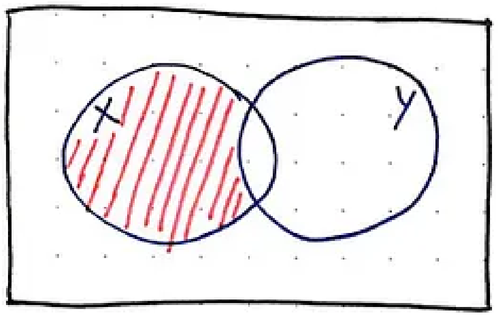
Intersections vs. unions

Okay, time to complicate things a little bit more! Intersections and unions are great, but they're only scratching the surface of set theory. For our purposes, we'll need to be familiar with some other operations as well. As it turns out, there are two operations that turn up quite a bit in computer science: **set differences** and **relative complements.** We'll learn how they both play a role in the next section, but first, let's figure out how they work!

Slightly More Complex Set Operations

SET DIFFERENCE

X − Y: yields the difference between two sets, or all the elements in set **X** that are not in set **Y**.



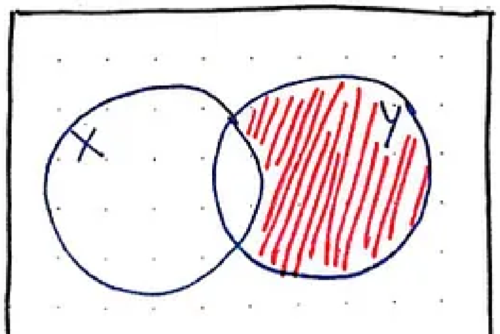
X = { m, L, A, C, Z, W }

Y = { X, N, O, L, A, m }

X − Y = { C, Z, W }

RELATIVE COMPLEMENT

Y \ X: yields a set with all of the elements in set **Y** that do not also exist in set **X**.



X = { 9, 13, 43, 6 }

Y = { 9, 2, 6 }

Y \ X is the same as

Y − X = { 2 }

Set differences and relative complements

Set differences are how we can figure out the difference between two sets. In other words, we can determine what a set looks like without any of the elements that are contained in the *other* set. Another way to write this is **X — Y**. In the example shown here, the difference between set X and set Y results in all of the elements that exist in set X but do not exist in set Y, or the letters *C, Z,* and *W*.

Relative complements are basically the opposite of set differences. For example, the relative complement of Y as compared to X will return all of the elements in set Y that don't appear in set X. We can denote the relative complement by using the short hand **Y \ X**, which is actually results in the exact same returned set as **Y— X**. In our example, the set Y is smaller than the set X. In our example, the only thing that exists in Y that doesn't exist in X is the number *2*.

> Effectively, we're simply subtracting set X from set Y, and answering the question: what exists in Y that doesn't exist in X?

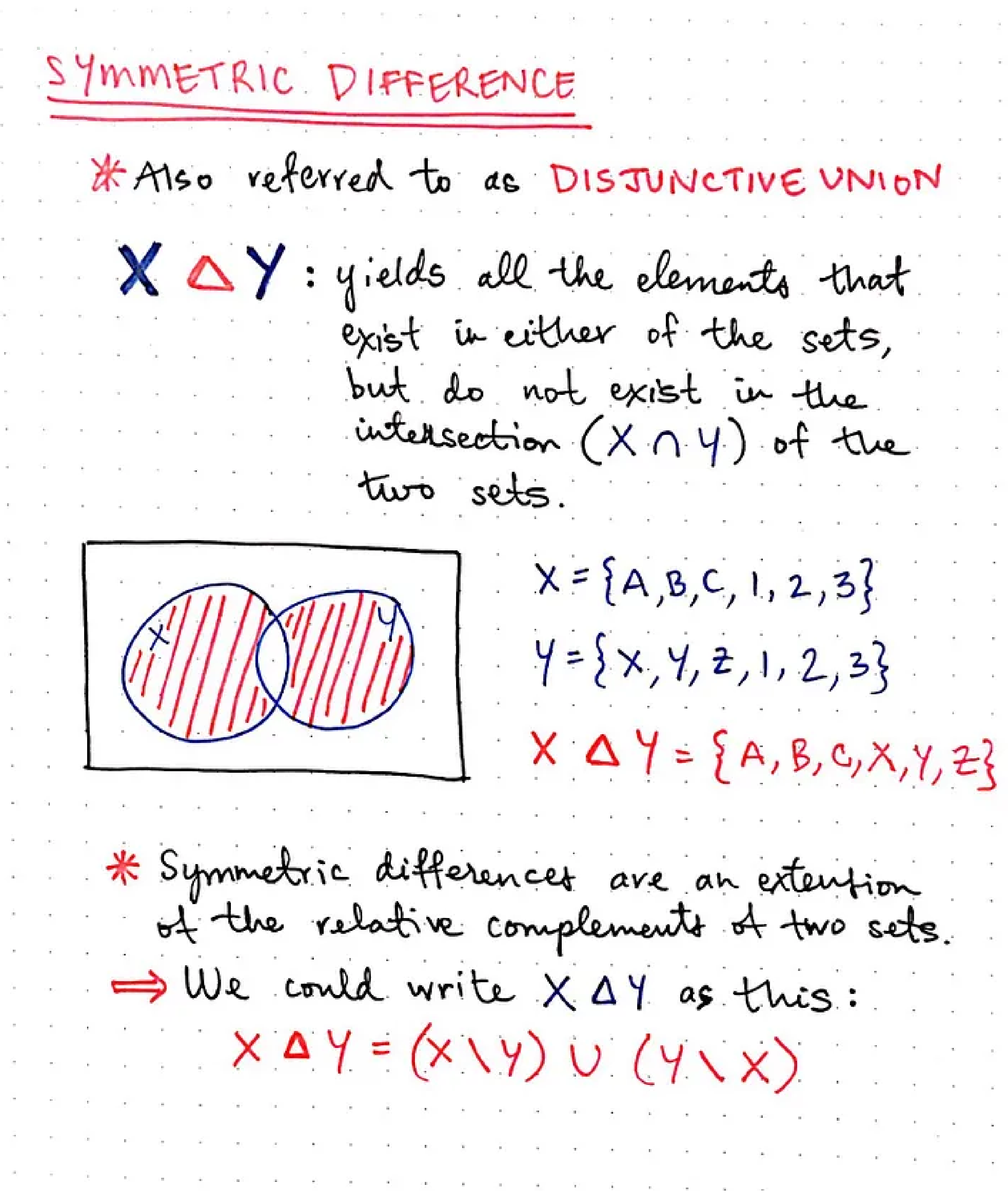
You might have noticed that in some of the examples, we're dealing with strings, and in others, the elements are letters, and sometimes even numbers. This brings up an important point: sets can contain literally any kind of element or object! You can think of them as hashes in that way: they can hold any item, as long as it only occurs once within the set.

Alright, let's look at one last operation — the most complicated one of them all. But we can handle it!

Sometimes, when we have two sets, we might want to find the opposite of the intersection of the two sets. In other words, I might want to find all of the books that I own, and all of the books that I have read, but *none* that intersect between the two. What would we call that subset? And how would we find it?

Well, the proper term for what we're looking for in this case is something called the **symmetric difference** of our two sets, which is also sometimes referred to as the **disjunctive union**. The symmetric difference yields all of the elements that exist within either of the two sets, but *do not* exist at the intersection ($X \cap Y$) of them.

Let's look at an example, which should help clarify what I mean:



SYMMETRIC DIFFERENCE

* Also referred to as DISJUNCTIVE UNION

$X \triangle Y$: yields all the elements that exist in either of the sets, but do not exist in the intersection ($X \cap Y$) of the two sets.

$X = \{A, B, C, 1, 2, 3\}$

$Y = \{X, Y, Z, 1, 2, 3\}$

$X \triangle Y = \{A, B, C, X, Y, Z\}$

* Symmetric differences are an extention of the relative complements of two sets.

$\Longrightarrow$ We could write $X \triangle Y$ as this:

$$X \triangle Y = (X \setminus Y) \cup (Y \setminus X)$$

Symmetric differences of sets

In the example above, the symmetric difference is basically the same as find the relative complement of set X and set Y. If we super mathematical about it, finding the symmetric difference is the same as finding the union of relative complements of set X and of set Y. We could write that out as: **X ▵ Y= (X \ Y) ∪ (Y \ X).**

But don't let that confuse you!

> All we really need to do in order to find the symmetric difference of two sets is ask ourselves: what elements exist in set X that don't exist in set Y, and which elements exist in Y that don't exist in X? In other words: which elements are unique to each set, and don't occur within both of them?

In the example above, the numbers *1, 2,* and *3* occur within both sets. However, the letters *A, B, C* and *X, Y, Z* occur only within set X or set Y, and are therefore the symmetric difference of the two sets.

Okay, that was a *lot* of theoretical stuff. Let's see this theory play out in practice, shall we?

Sets all around us

Hopefully by now, you're wondering what the point of learning sets is. I don't blame you: it's a good question! And it's finally time to answer it.

Guess what? Sets are *everywhere*. They're actual data structures that we can use whenever we want in Java, Python, Ruby, and even JavaScript! You might even be able to guess some of the methods that each of these languages allows us to perform on sets.

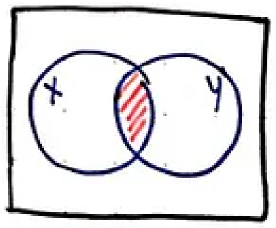
Let's take a quick look, using JavaScript as an example:

```
var s = new Set();

s.add(2);
// Set { 2}
s.add(45);
// Set { 2, 45}
s.add(45);
// Set { 2, 45}
s.add('hello world!');
// Set { 2, 45, 'hello world!' }

s.has(2);
// true
s.has('cats');
// false

s.size;
// 3

s.delete(45);
// Set { 2, 'hello world!' }

s.has(45);
// false
```

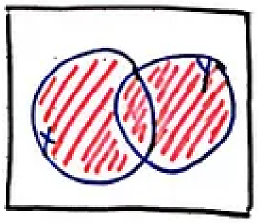
Obviously, some of the method names will change from language to language. For example, Ruby's implementation of `has` is called `include?`, but the idea is fairly similar from one language to another. Python's version of sets actually allows you to call methods like `intersection`, `union`, and `symmetric_difference`!

But, what good are sets, anyways? I mean, we can use them in all of these languages, but when are they useful?

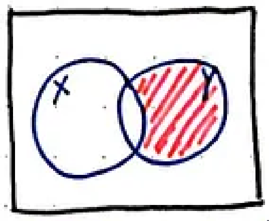
Time-Complexity of Sets



INTERSECTION



UNION



DIFFERENCE/ COMPLEMENT

The time-complexity of all of these set operations is O(length(X) + length(Y)).

Imagine two sets, X + Y:

X = {A, B, C, 1, 2, 3}   Y = {X, Y, Z, 1, 2, 3}

In order to find the intersection, union, or difference/complement of these two sets, we must traverse through the entire length of both set **X** and set **Y**.

A = {5, 2, 9, J, Z, 4, Q}] basic operations can be done in constant time!

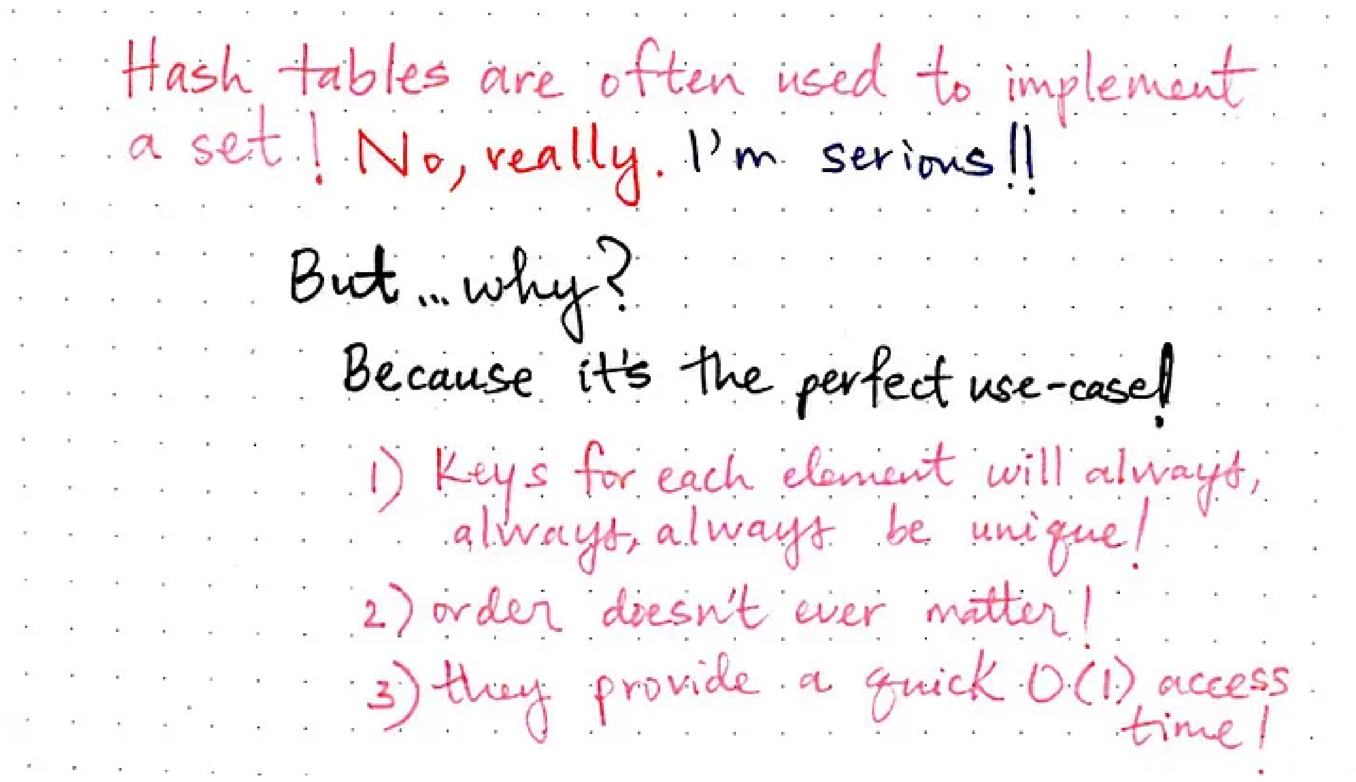
add an element to set A $\longrightarrow$ O(1)
remove element from set A $\longrightarrow$ O(1)
find an element within A $\longrightarrow$ O(1)
determine length of set A $\longrightarrow$ O(1)

Well, for one thing, they can be pretty time-efficient.

Remember all of those complicated operations like `intersection`, `union`, and `difference`? Well, guess what? The amount of time it takes for us to run any of those complex operations depends purely on the length of the two sets. This is because in order for us to find the intersection, union, or difference/complement of these two sets, we have to effectively traverse through the entire length of the two sets being compared. Usually, even giant sets still won't take that much time to traverse.

But what about basic operations? What about adding an element to one of those sets, removing it, or even finding an element within it? Well, all of those operations take **constant time**, or *O(1)*. This can be incredibly powerful, and often means that a set might be a better structure than a dictionary or a hash!

But, wait a second: how is it possible that all of these set operations are so fast?! How does that even happen? Well, as it turns out, many sets are actually implemented by hash tables under the hood! (See, I promised you it was all going to come together!) We already know about hash tables, but why do they make for good skeletons for set implementations?


Hash tables are often used to implement a set! No, really. I'm serious!!
But... why?
Because it's the perfect use-case!
1) Keys for each element will always, always, always be unique!
2) order doesn't ever matter!
3) they provide a quick O(1) access time!

Hash tables are often used to implement sets!

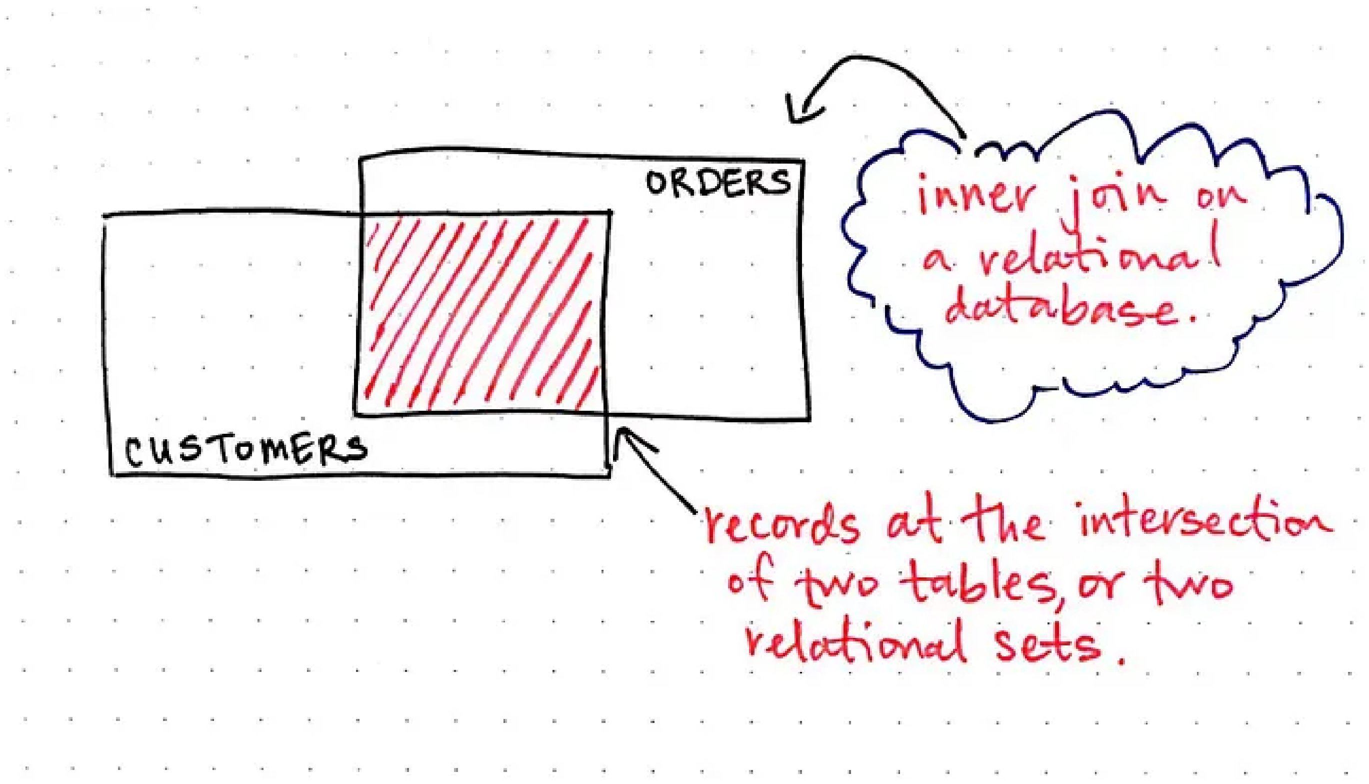
Well, there are a few good reasons: **first**, given what we know about hash

hash tables provide a *O(1)* constant access time, which is what ideal for basic operations performed on a set.

Alright, so hash tables make for good sets. And sets are data structures that most languages give us for free. But when I started this post, I told you that sets were *everywhere*, right? I feel like I should probably let you in on a little secret that's going to (hopefully) blow your mind:

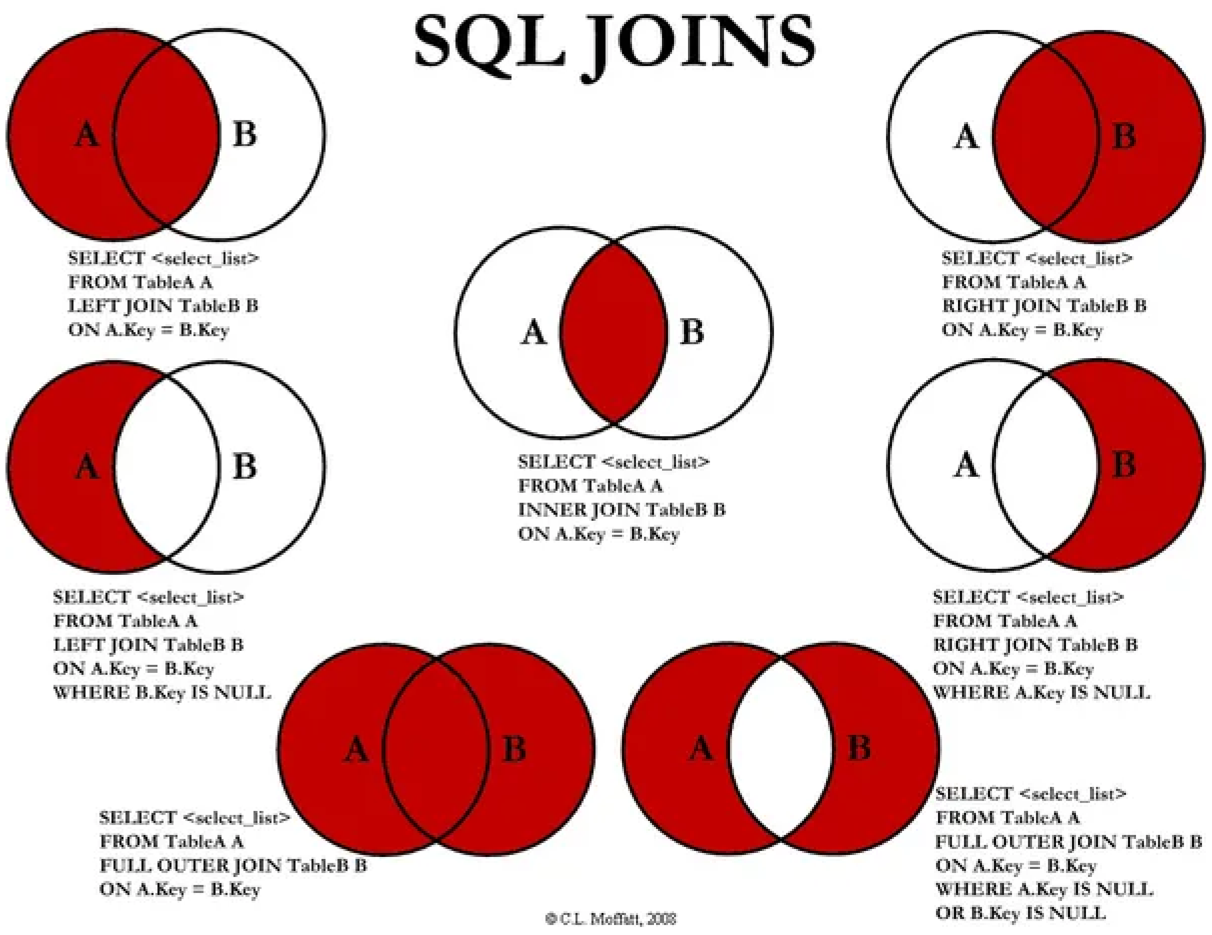
> Relational databases are based almost entirely upon set theory.

In fact, if you've ever worked with or queried a database, or had to write SQL, you're probably familiar with the idea of finding records at the intersection of a table.


ORDERS
CUSTOMERS
inner join on a relational database.
records at the intersection of two tables, or two relational sets.

Relational databases are based entirely upon set theory

This is nothing more than an abstraction of the Venn diagram version of sets that we started off this post with! In fact, even the most complicated SQL statements are nothing more than operations on sets.



http://stackoverflow.com/questions/406294/left-join-vs-left-outer-join-in-sql-server

A SQL `INNER JOIN` is just the *intersection* of two sets.

Finding the `LEFT JOIN` of two tables is nothing more than finding the *set difference* or the *relative complement* of the two tables.

A SQL query that calls for a `FULL OUTER JOIN` is merely returning the *union* of two sets.

And that super complicated `FULL OUTER JOIN` where one keys on both tables is `NULL`? (Also known as the bane of my existence when it comes to writing SQL statements?) That's just the *symmetric difference* or *disjunctive union* of two tables.

How amazing is that?! All of that seemingly boring set theory, when put into practice, makes databases seem like the *coolest things ever*. And that, my friend, is a truly a feat in and of itself!

Resources

Set theory is fairly widespread in various parts of computer science, from its use in relational databases, to a data structure that exists in various languages, including Python, Ruby, JavaScript, and Java. There are plenty of good resources out there to help you get a better understanding of set theory; here are a few to get you started!

1. LEFT JOIN vs. LEFT OUTER JOIN in SQL Server, StackOverflow
2. Set Theory, MariaDB
3. Complexity of Python Operations, Professor Richard E. Pattis
4. Set Theory — Design of Computer Programs, Udacity
5. Set Theory | Introduction, GeeksforGeeks

Programming Database Computer Science Code Data Structures