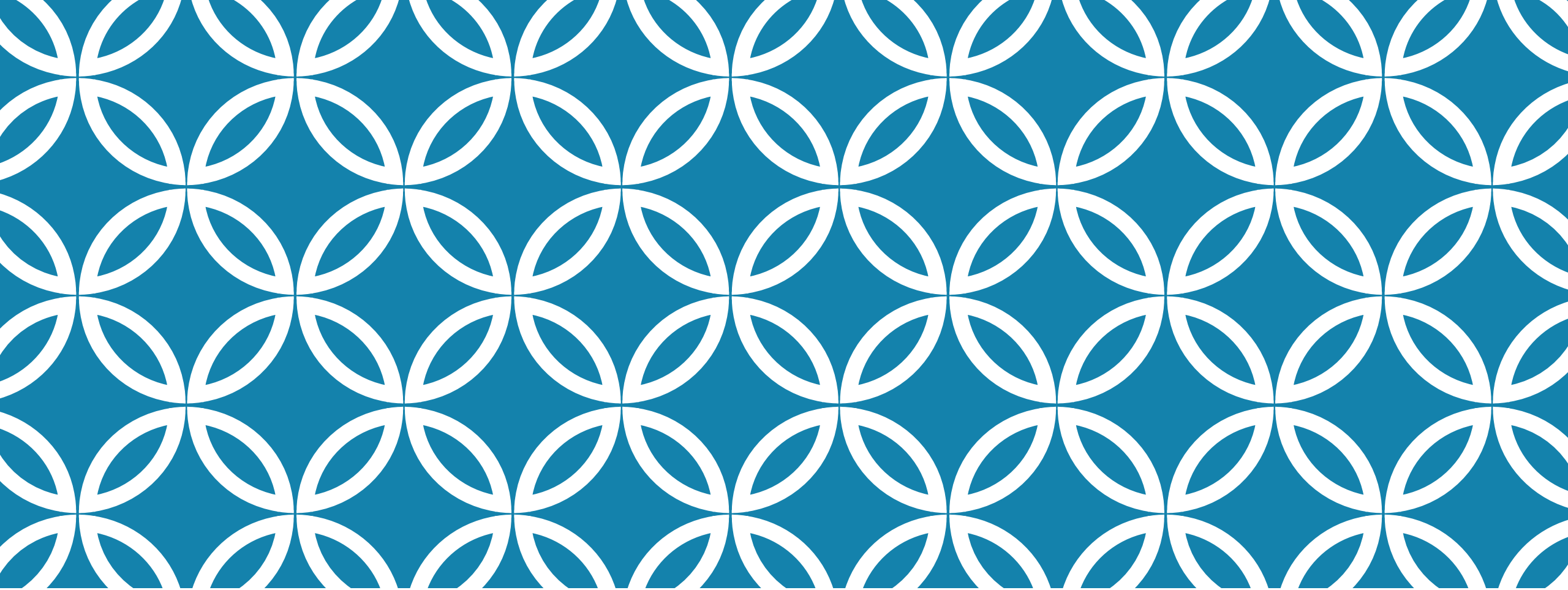


CS 5412/LECTURE 18 ACCESSING COLLECTIONS

Ken Birman

Spring, 2020

STRUCTURED AND UNSTRUCTURED DATA

Earlier we learned that cloud data is generally viewed as structured or unstructured.

Unstructured data means web pages, photos, or other kinds of content that isn't organized into some kind of table.

Structured data means "a table" with a regular structure.

A TABLE

Cow Name	Weight	Age	Sex	Milking?
Bessie	375kg	4	F	Y
Bruno	480kg	3	M	
Clover	390kg	2	F	N
Daisy	411kg	5	F	Y
…				

STRUCTURED AND UNSTRUCTURED DATA

Often we convert unstructured data to structured data.

For example, we could take a set of photos and extract the photo meta-data.

We could create a table: photo-id or name, and then one column per type of tag, and then the value of that tag from the meta-data we extracted.

STRUCTURED AND UNSTRUCTURED DATA

Another example with a photo collection.

We could take a set of photos and *segment* them to outline the objects in the image: fences, plants, cows, dogs, etc.

Then we can tag the objects: this is Bessie the cow, that is Scruffy the dog, over there is the milking barn. And finally, we could make one table per photo with a row for each of the tagged objects within the photo.

A PHOTO AND ITS META-DATA

TAG	VALUE	ADDITIONAL_VALUE
GPS	42°26'26.27" N -76°29'47.80"	DMS
Cow	Bessie	Object #3
Cow	Daisy	Object #4
Dog	Scruffy	Object #5
DATETIME	Jan 15, 2020	10:18.25.821
Bldg	Milking shed	Object #8
Man	Farmer Jim	Object #71
Bldg	Farm House	Object #2
Vehicle	Tractor	Object #33

STRUCTURED AND UNSTRUCTURED DATA

What about missing data?

Often if we convert unstructured data to structured data, not all the fields will be identical!

We could easily end up with "holes" in the table: missing information.

In fact this is exactly what happens. Structured data can have gaps!

A STRUCTURED WORLD!

This lets us start with almost any information, even unstructured information, and convert that information into tables.

For many purposes, we can view almost everything as a table or a multi-dimensional "tensor" (means a d-dimensional matrix).

The most universal perspective is to think about the table itself as a *collection* of tuples (rows).

COLLECTION CONCEPT

A collection is any kind of list of data that has some form of key for each item. The value could be a simple value like a number, or a tuple.

Unlike in cloud storage, collections are a programming concept used inside your code. So the value can also be any form of object, or even another collection!

Now you can think about code that iterates over the (key,value) pairs and even does database-style operations on them!

MISSING VALUES

Most kinds of objects are *nullable*

This means that null is a legal value, and can be used for missing data

Others might have a default value for missing data, like -99

IMPORTANT DATABASE CONCEPTS

In databases we talk about

- A **schema**: This is the layout of our tables (hence, our collections) plus the relationships between them (for example, "cow id" might show up in many different relations).
- Individual **relations**, which just means "tables". Each table is a set of rows and within each row, some column is designated as the primary key
- Often a relation is sorted by **primary key**, and there may be **secondary keys** (sorted indices) as well for other columns (B+ trees)

IMPORTANT DATABASE CONCEPTS

We say that a **select** operation is occurring if we take a row but extract just a few columns, yielding smaller rows. **Project** is similar, but creates a whole new table containing only rows that match some pattern.

A **group-by** operation occurs if we create smaller collections that have the same value in some field, like if we grouped by cow names. Bessie's data would end up in one single group.

A **join** operation occurs if we have two tables and combine data from both, for rows that have matching values in some field.

PROGRAMMING LANGUAGE EMBEDDINGS

The idea is that these collections can be used just like other data structures, and will even be created *automatically* just by opening a particular file or database and saying that you wish to treat it as a collection!

You just need the file name.

Then you can write code that actually has database-style operations in your code – you don't have to implement them yourself.

VISITING TWO GOOD WEB SITES

Pandas, for Python:

https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html

LINQ. The examples here are for C# (like Java or C++):

https://docs.microsoft.com/en-us/dotnet/csharp/linq/query-expression-basics

YOU CAN CREATE NEW PERSISTENT DATA TOO, OR UPDATE EXISTING DATA

These same solutions create new temporary collections as in-memory data objects all the time.

You can just work with them like other in-memory variables, but you can also write them back to storage.

And you can do in-place updates too, but this is not as common. For many reasons the cloud is often a world of "immutable" data (write-once, read as often as you like). New versions are often preferable to updating old versions.

SQL AND NoSQL

We often say that a database permits "SQL programming". In fact there are packages like the ones we just saw for most SQL databases.

A big feature of databases with SQL is consistency: they use a model called ACID that guarantees atomicity for updates. Invisible to you, this requires mechanisms like read/write locking and two-phase commit.

But as we learned from Jim Gray, SQL/ACID doesn't scale well. In fact this is one reason for the immutability model: creating a totally new object doesn't require locking and two-phase commit operations.

SQL AND NoSQL

This is why most cloud computing systems use sharding, no locking, and no two-phase commits.

But the effect is that a database might actually not be consistent. We call this the NoSQL model.

When you write cloud computing code with Pandas or LINQ, it is your responsibility to specify which model you are working with.

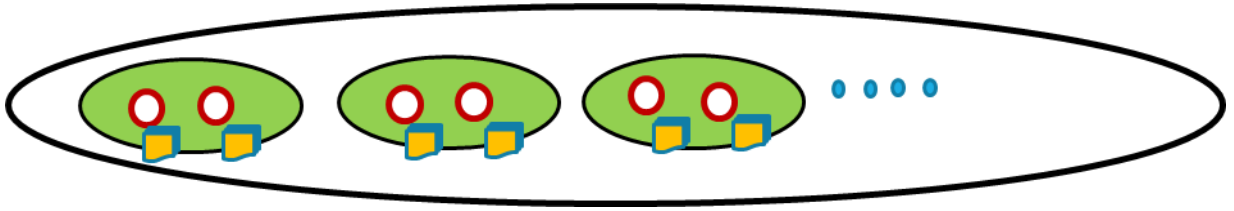
HOW DO YOU UPDATE NOSQL DATA?

Some NoSQL systems favor a model in which you can create objects and delete them, but can't modify them.

Some support "versioned" objects. Derecho's object store does this; the data is indexed by time, which can be very helpful in IoT applications.

Many have a concept of an "append only file". You can't change the existing data but can extend it with new records.

SHARDED DATA



With sharded data, we often take one program, but then *run an instance of it on each shard, one instance per shard.*

If we adopt this approach, we end up with parallel processing: each instance handles a portion of the overall task, just for data in its own local shard. If it generates new tuples to store, we "shuffle" them to the proper locations before the next stage of computing.

We also can use group-by and then some form of aggregation to handle the reduce operation common in MapReduce computational patterns.

TEMPORARY DATA? PERSISTENT? OR BOTH?

A curious thing about the cloud is that we often do almost all our computing on temporary data! Think of the air traffic control example, where the only permanent data was the flight plan database.

In the cloud, the raw IoT input data is permanent, or perhaps held for a fixed period. But with the input we can rerun our task and re-create any needed outputs! And this can be repeated for subsequent stages too!

So, most cloud data is viewed as temporary, but cached (and maybe even persisted on a temporary disk area for fast reloading!) We can always recreate it if necessary. You control when persistent data will be stored.

SUMMARY

In today's cloud platforms, data is sharded all the time.

Tools like Pandas and LINQ make it very easy to compute on this data, especially if we can think of it as have some kind of regular structure.

We haven't yet seen them, but there are also powerful packages totake less-structured forms of data, like web pages, and extract structured data.