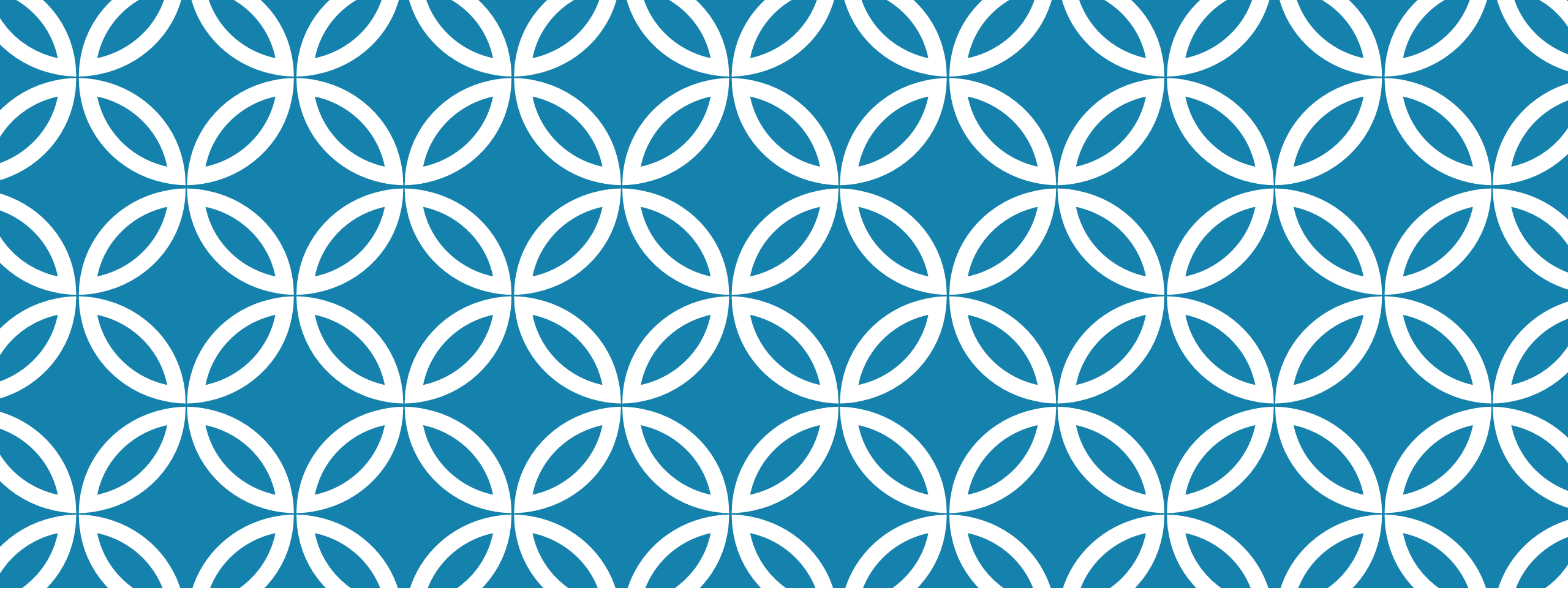


CS 5412/LECTURE 18 ACCESSING COLLECTIONS

Ken Birman

Spring, 2021

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).

JSON FILES

The cloud has a standard way of representing this kind of structured data, in a file format called JSON.

It looks like a web page, with text fields, field = “value”;

These can nest, using a simple bracket notation. You can open a JSON file as a text file, but more often you use a JSON file reader. It will automatically convert the file into a “collection” of fields (which can also be collections, because JSON allows nested declarations).

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!

WHAT ABOUT NON-JSON FILES?

Many file format can also be treated like collections. For example, .csv files (spreadsheets in comma-separated form).

Some scanner libraries can deal with many formats all using the same scanner library – and again, you end up with collections that could perhaps have nested collections (fields that hold a collection).

A collection is like a list where the file itself determines the items and order

PROGRAMMING LANGUAGE CONCEPTS

Modern object oriented programming languages have outstanding support for the idea of collections of objects. This is *not* found in languages that lack object orientation. But for users of Java, C++, Python with its object features, etc, collections are very easy to access.

A collection is a *set* type. You can think of it as a list of elements.

For the cases that arise in the cloud, each element is a key-value pair.

ITERATORS

Modern object oriented languages allow for loops to scan collections, or subsets of them. The scan will be in the order of the collection itself.

This is done using an "iterator" object. Often the syntax hides the object from you if you just plan to scan the entire collection.

An iterator object represents some portion of the collection. It has a **begin** point, a **next** operator, and an **end** point.

EXAMPLES

My photo meta-data was a table, but I can think of it as a collection of rows, one row per meta-data item.

Every row always has a unique row key. The value is the row contents: a struct or array or an object with one field per column.

To scan the full row, a for loop will begin with the first row and scan to the last row: **begin… next… next…. End.**

EXAMPLES

You would see code like this in C++:

```
for(auto row = table.begin(); row = row.next(); row != table.end())
{
    do something with this row
}
```

EXAMPLES

You would see code like this in C++:

```
for(auto row: table)
{
        do something with this row
}
```

MANY CLOUD STORAGE LAYERS PROVIDE ITERATORS AS "CONNECTORS"

Suppose you have data in a cloud DHT, database, file system, etc.

You can generally access your data by:

- Opening the storage system using a library method that returns an iterator. By default it will iterate over all of your content in the service.
- Then you can apply a filter to "focus" the iterator on just certain items.

A filter will select certain items, but skip others.

EXAMPLES

You would see code like this in C++:

```
for(auto row: table)
{
    if(row.cow_id = 1471)
    {
        do something with this row
    }
}
```

BUT WE CAN DO EVEN BETTER!

Languages like C++ have built-in libraries that do this form of selection for you, in a few lines of code:

```
// … code to connect src to some collection hosted on Azure ….
auto src = …;   // details depend on the particular service

// now I can iterate over the collection

auto my_rows = from(src).where([ ](row r) { return r.cowid == 1417 });
```

The first line binds to the service. The second scans data.

BUT WE CAN DO EVEN BETTER!

… or even more compactly, in a single line:

```
auto my_rows = table.where([ ](row r) { return r.cowid == 1417 });
```

Notice how the select operation is using a C++ lambda that decides, row by row, which rows to include into my_rows. Here is picks rows r with r.cowid = 1417.

NOTICE THE STRANGE "METHODS" USED HERE

This example used ".where" – an SQL operation. In fact all of SQL is available!

Collections support a number of these kinds of methods. They tend to come in two flavors: functions that will test an item and return true or false, like where, and functions that transform the whole collection, like count or distinct.

The true/false ones are usually employed to select a subset of items without changing them. The transformer functions create a new collection (perhaps just one element) from an input collection.

LINQ EXAMPLES

Things to notice:

- Code is very "succinct"
- Lots of use of lambdas
- Very powerful
- Mixes with normal C++ (in fact, is a C++ library)

Double the odd numbers, then keep those in the range [3,11]:

```
int src[] = {1, 2, 3, 4, 5, 6, 7, 8};
auto dst = from(src)
        .where( [](int a) { return a % 2 == 1; })  // 1, 3, 5, 7
        .select([](int a) { return a * 2; })       // 2, 6, 10, 14
        .where( [](int a) { return a > 2 && a < 12; }) // 6, 10
        .toStdVector();     // dst will be a std::vector with 6, 10
```

Order descending all the distinct numbers from an array of integers, transform them into strings and print the result.

```
int numbers[] = {3, 1, 4, 1, 5, 9, 2, 6};
auto result = from(numbers)
          . distinct()
          . orderby_descending([](int i) {return i;})
          . select([](int i){std::stringstream s; s<<i; return s.str();})
          . toStdVector();
for(auto i : result)
    std::cout << i << std::endl;
```

Visit booling on GitHub to download, learn more

EXAMPLE WITH STRUCTS

In a list of friends, find the subset who are under age 18, order them by age, then return their names.

```
struct Friends { std::string name; int age; };

Friends src[] = {
   {“Kevin”, 14}, {“Anton”, 18}, {“Agata”, 17}, “Saman”, 20}, {“Alice”, 15},
};

auto dst = from(src).where([](const Friends & who) { return who.age < 18; })
                    .orderBy([](const Friends & who) { return who.age; })
                    .select( [](const Friends & who) { return who.name; })
                    .toStdVector();

// dst type: std::vector<:string>…  items: “Kevin”, “Agata”, “Alice”
```

EXAMPLE WITH STRINGS

In a list of text messages, count the number of messages to Dennis by sender:

```
struct Message {  std::string PhoneA; std::string PhoneB;  std::string Text;  };

Message messages[] = {
    {“Anton”,  “Troll”, “Hello, friend!”},
    {“Denis”,  “Mark”,  “Join us to watch the game?"},
    {“Anton”,  “Sarah”, “OMG! ”},
    {“Denis”,  “Jimmy", “How r u?”},
    {“Denis”,  “Mark",  “The night is young!”},
};

int DenisUniqueContactCount =
    from(messages)
        .where([](const Message & msg) { return msg.PhoneA == “Denis”; })
        .distinct([](const Message & msg) { return msg.PhoneB; })
        .count();
```

SOME LINQ OPERATORS

Filters and reorders:

- `where(predicate)`, `where_i(predicate)`
- `take(count)`, `takeWhile(predicate)`, `takeWhile_i(predicate)`
- `skip(count)`, `skipWhile(predicate)`, `skipWhile_i(predicate)`
- `orderBy()`, `orderBy(transform)`
- `distinct()`, `distinct(transform)`
- `append(items)`, `prepend(items)`
- `concat(linq)`
- `reverse()`
- `cast()`

Transformers:

- `select(transform)`, `select_i(transform)`
- `groupBy(transform)`
- `selectMany(transfom)`

`Bits and Bytes:`

- `bytes(ByteDirection?)`
- `unbytes(ByteDirection?)`
- `bits(BitsDirection?, BytesDirection?)`
- `unbits(BitsDirection?, BytesDirection?)`

Aggregators:

- `all()`, `all(predicate)`
- `any()`, `any(lambda)`
- `sum()`, `sum()`, `sum(lambda)`
- `avg()`, `avg()`, `avg(lambda)`
- `min()`, `min(lambda)`
- `max()`, `max(lambda)`
- `count()`, `count(value)`, `count(predicate)`
- `contains(value)`
- `elementAt(index)`
- `first()`, `first(filter)`, `firstOrDefault()`, `firstOrDefault(filter)`
- `last()`, `last(filter)`, `lastOrDefault()`, `lastOrDefault(filter)`
- `toStdSet()`, `toStdList()`, `toStdDeque()`, `toStdVector()`

Coming soon:

- gz(), ungz(), leftJoin, rightJoin, crossJoin, fullJoin

YET THIS IS CALLED THE "NOSQL MODEL"

Why "No" SQL? SQL is the full database model including ACID transactions for updates.

NoSQL is used in read-only settings, and doesn't have full SQL guarantees... But it *does* have all the SQL operators and because read-only data isn't changing, you don't need the full ACID model.

We split the updates away from the queries. The updates are done "in the background".

SO…

We can iterate over data already in memory, in any data structure compatible with this notion of collections (interface Icollections, in C++).

And we can also iterate over data hosted externally, in some sort of cloud repository like a database, a csv file, etc.

There are many things we can do at this point – including any kind of database SQL query, expressed in this notation.

REQUIRED STEPS

Application requires a *binding* to the data source, such as the file system or perhaps the key-value store.

This is similar to saying "to read a file, first the application must know the file name and be able to open it."

The difference is that a binding connects to a service that could be sharded, whereas a file is a single thing in a file system or key-value store.

EACH TYPE OF CLOUD HAS ITS OWN WAY OF EXPRESSING BINDINGS

In some cloud systems bindings are very static. But fancier clouds, like Azure and AWS, allow you to express a binding to a service that might not be running.

The "app service" would then launch your required service on demand. But startup could take 30s or more for a complex service. *Pre-binding* is important if you care about performance.

Once launched, you would want the service to remain active for a while!

SEQUENCE THAT WE END UP WITH?

Application A will pull data in from service S using an SQL-like notation.

1. You build application A using a package, perhaps PyLINQ or Pandas
2. You create a container and install A on the cloud using the hybrid cloud App Manager Service. You tell the App Manager Service that A requires a binding to S.
3. At runtime, when A is started, App Manager will check that S is running, and will start it if not (if you requested this).
4. Now A begins to execute and should be able to bind its iterators to S as a data source, enabling A to do runtime access to data in S.

COMMON WAYS THIS CAN FAIL?

When debugging A you probably ran your own version of S on your own computer. Moving A to the cloud requires you to express this binding obligation, to properly tell the App Service where S is supposed to be running, and to have runtime permissions to talk to S.

Any of these steps could fail or be misconfigured. Then when A is launched in the cloud, it will crash with some form of binding error.

Best to find examples of how to do it, e.g. in docs.Microsoft.com, and then modify those to create your application and service bindings.

OK, NOW A CAN TALK TO S!

(In practice, it can take weeks to get this right… like withCosmosDB in assignment 2 – that was an example of a "binding" challenge)

Now what could go wrong?

A very common issue is that because cloud-scale DHT data stores are huge, you are very likely to see issues you didn't see in your test setup!

MISSING VALUES

Recall that unstructured data converts to structured data but with gaps. Most kinds of objects are *nullable* – a *null* represents a gap.

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 to know which service is hosting your data, and then you have to "bind" to it and obtain the proper kind of iterator.

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

VISITING THREE GOOD WEB SITES

Boost LINQ for C++:
k06a/booling: Simplest C++ header-only LINQ template library (github.com)

Pandas, for Python:

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

C# LINQ (a language like Java):

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

Be aware of the limitations of NoSQL services that offer SQL APIs but in fact don't offer ACID guarantees.

CosmosDB and DynamoDB and Cassandra and RocksDB are all examples.

Often, you end up with a genuine database in the back-end, and updates go to it. Then periodically, you copy a checkpoint into CosmosDB or your favorite NoSQL solution. And queries run against the NoSQL system.

SQL AND NoSQL

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

If you use the wrong model for a given task, you could have visible bugs or inconsistencies, and this can cause your applications to crash.

NoSQL scales and performs much better for read-only uses.

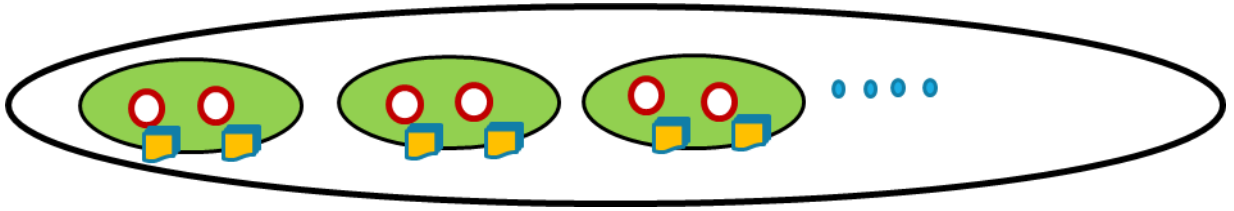
HOW DO YOU UPDATE NOSQL DATA?

Some NoSQL systems do allow updates, but in a restricted form.

These generally are systems that support "versioned" objects. Derecho's object store (Cascade) does this; the data is indexed by time, which can be very helpful in IoT applications.

You obtain the equivalent of an "append only file". You can't change the *existing* data but can add new versions by appending 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 big and often 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 to take less-structured forms of data, like web pages, and turn them into more structured summary data objects.