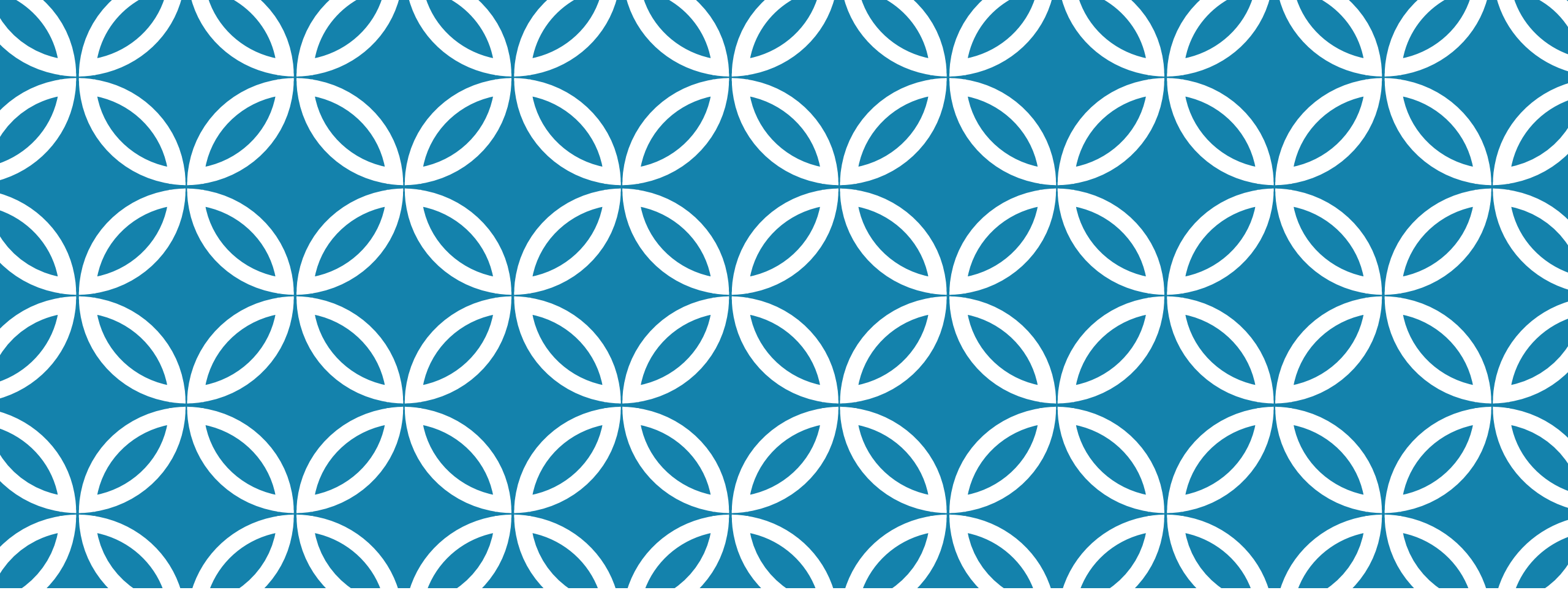


CS 5412/LECTURE 18 ACCESSING COLLECTIONS

Ken Birman

Spring, 2022

STRUCTURED AND UNSTRUCTURED DATA

It is common to say that cloud data is **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, or a list of items in a similar format such as (key,value) tuples in a collection

A TABLE

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

Notice that this table has an error: Sally isn't a male cow. "Milking" should be N. And we are missing weight data for Clover.

Often the first step is to clean up missing data, visibly incorrect data, etc.

STRUCTURED AND UNSTRUCTURED DATA

There are many tools to convert unstructured data to structured data.

For example, we can take a photo and extract the photo meta-data. This would initially be a list of (key,value) pairs. The values would be byte arrays

If we deserialize the values, we obtain some form of structure, and the fields in the structure become the "columns" in our row

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.

AUTOMATED PIPELINES

In fact the big cloud companies have huge automated pipelines that do exactly this task.

Photos are uploaded into, say, Facebook. Then in big batches they are auto-segmented, tagged to identify the people, and this in turn allows them to repost to the feeds of friends of those people.

AUTOMATED PIPELINES

Notice that the sequence would have a database query in it: first, the people in a photo are often friends of the person who uploaded it.

… so the autotagger would want a list of those friends as an input.

Then the autotagger would probably want to find prior photos of those individuals: a second query that returns a list of photos.

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

NOTICE THAT THE META DATA HAD MORE THAN ONE SOURCE

Some meta-data fields were put there by the camera, but other applications could add more tags

Here, some were added by photo analysis applications. The extra meta-data includes information about a series of objects identified by some sort of computer vision software.

Each type of meta-data would have its own columns.

JSON FILES

The cloud has a standard way of representing things like tags, in a file format called JSON.

```
{"widget": {
    "debug": "on",
    "window": {
        "title": "Sample Konfabulator Widget",
        "name": "main_window",
        "width": 500,
        "height": 500
    },
    "image": {
        "src": "Images/Sun.png",
        "name": "sun1",
        "hOffset": 250,
        "vOffset": 250,
        "alignment": "center"
    },
    "text": {
        "data": "Click Here",
        "size": 36,
        "style": "bold",
        "name": "text1",
        "hOffset": 250,
        "vOffset": 100,
        "alignment": "center",
        "onMouseUp": "sun1.opacity = (sun1.opacity / 100) * 90;"
    }
}}
```

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

These can nest, using a simple bracket notation.

STRUCTURED AND UNSTRUCTURED DATA

Now, imagine that we actually had many photos

We could do this same process photo by photo.

We end up with one row per photo. The photo name or id is just one more column!

OUR CHALLENGE: TRANSFORM DATA SETS

From unstructured to structured

From a collection of photos to a single table describing the collections, with one row per photo

From a table with missing data to one with all records filled in (or rows with missing data deleted).

EXAMPLE: A COMPOSITE TABLE

Photo-id	GPS	Camera orientation
27	4°26'26.31 N -76°29'47.80	NNE
28	42°26'26.27 N -76°29'47.84	
29	42°26'26.28 N -76°29'47.72	SW
30	42°26'27.11 N -76°29'47.15	N

A STRUCTURED WORLD!

We can start with almost any information, even unstructured information, and convert that information into tables.

Then can view almost everything as a table or a multi-dimensional "tensor" (means a d-dimensional matrix).

But how can a program deal with data of unknown dimensionality and format? This leads to the concept of collections

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!

STRUCTURED DATA CAN BE ACCESSED AS A COLLECTION

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

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 list of elements.

For the cases that arise in the cloud, elements are often key-value pairs.

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.

THINGS TO BE AWARE OF

Notice that whereas a database select has two clauses – one to pick the rows ("where"), and one to decide what to keep ("select"), our where clause just picks the rows to keep.

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

NOTICE THE STRANGE "METHODS" USED HERE

Each language has its own notation. C++ uses anonymous methods – lambdas –declared inline. This one used returns true or false.

In C++, this is a list of variables from the surrounding scope used inside the lambda

The argument will be our iterator variable

```
[ ](row& r) { return r.cowid == 1417 });
```

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 Microsoft LINQ for .NET documentation to 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”, “Alice”, “Agata”
```

Visit Microsoft LINQ for .NET documentation to learn more

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();
```

Visit Microsoft LINQ for .NET documentation to learn more

AZURE C#

Microsoft actually has a favorite LINQ language: C# (a cousin of Java)

Using the Mono compiler C# is also available on Linux, including all LINQ elements

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}
```

SOME BOOLINQ.H OPERATORS. THE FULL MICROSOFT LINQ HAS MORE!

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()`

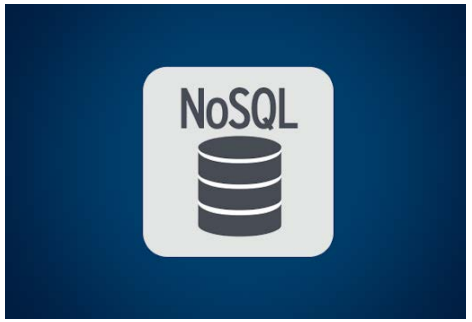
Fancy stuff:

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

Visit Microsoft LINQ for .NET documentation to learn more

THINK OF LINQ AS A NOSQL TECHNOLOGY

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

VISITING THREE GOOD WEB SITES

LINQ, on Microsoft .NET: LINQ overview - .NET | Microsoft Docs.
Supported in 44 languages! Nice slide set: here

Examples of LINQ queries: Write LINQ queries in C# | Microsoft Docs

Pandas, for Python:

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

SAVING OUTPUT FROM A LINQ PROGRAM

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.

STRANGE HDFS LIMITATION

When using the HDFS file system there is an extra issue to know about.

HDFS only allows file **creation, replace** and **append** – to update a file, you must delete the old copy and replace it with a new version!

This is tied to the way that Apache Hadoop handles rollback in MapReduce jobs. But it fits nicely with systems like CosmosDB and Cascade that have a concept of versions – you can't change a version but you can create a new one.

TEMPORARY DATA? PERSISTENT? OR BOTH?

A curious thing about the cloud is that we often do almost all our computing on temporary data!

Original input is normally held for a fixed period. But anything derived from the input is viewed as temporary: we can create it again if necessary!

So, most cloud computing frameworks expect you to specify which data is temporary and which will be persistent.

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.