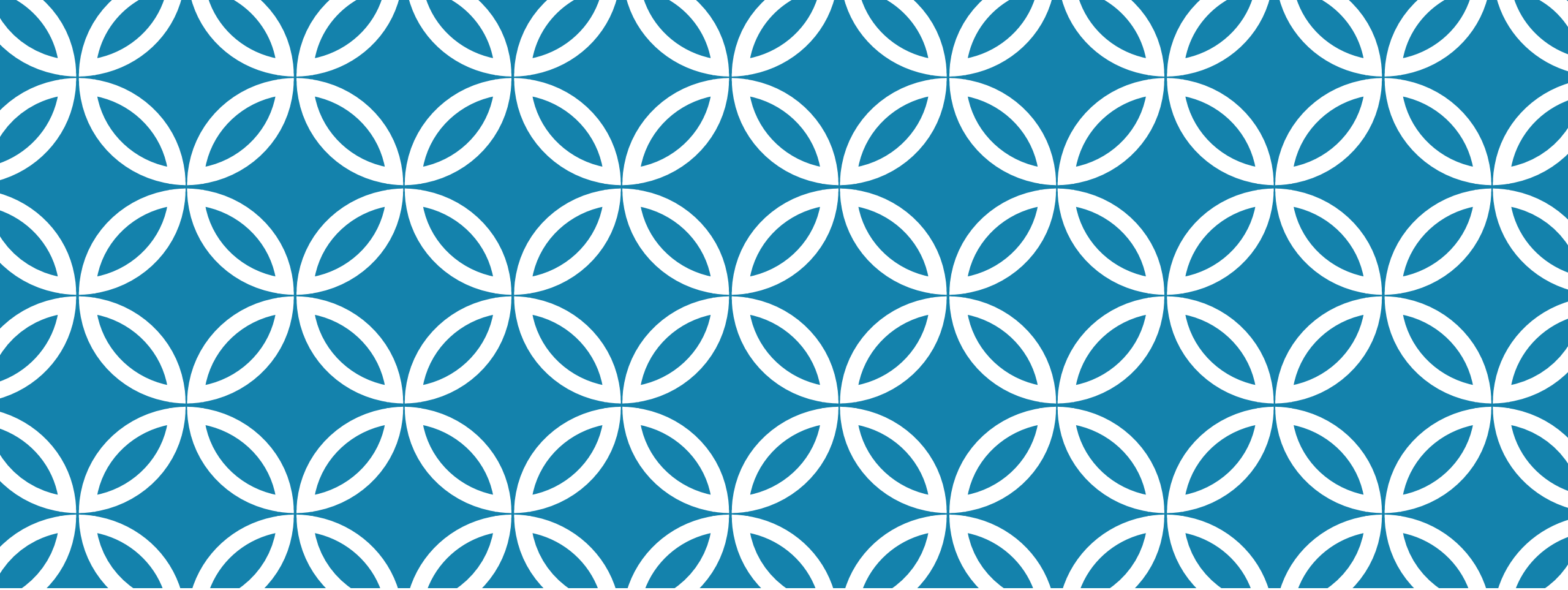
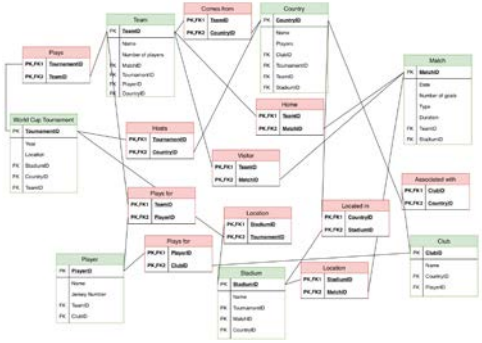


CS 5412/LECTURE 19: ACCESSING COLLECTIONS USING SQL IN LAMBDAS

Ken Birman

Fall, 2022

RECAP: DATABASES



Entity-Relations Graph

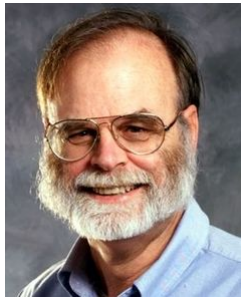
Database systems offer the abstraction of "one big repository

- Data lives in "relational tables": nodes in an entity-relations graph tied together by key-foreign key edges.
- A database uses SQL to express read-only queries and updates.
- Data is often too big to fit in memory… disk I/O is a big consideration.
- ACID concept… transactional model. Transactions run concurrently, using locking to avoid conflicts and 2-phase commit at the end

CAN A DATABASE BE SHARDED?

Yes, and every major SQL product automatically does this sort of thing.
But often, not into totally independent mini-databases.

Sharding of relations (tables) is common, and enables modern SQL databases to hold really huge amounts of data in a single relation!



The scalability of concurrent SQL computing is limited by locking conflicts, timeouts, abort/rollback and retry.

RECAP: KEY-VALUE STORE

Key-value DHTs offer the abstraction of "many pools of memory".

- A pool of memory is called a shard, owned by one or more servers.
- By design, we have enough shards so that data would normally fit in memory, allowing O(1) access. Supports **get**/**put**/**watch**
- Clients issue requests concurrently, but a server normally handles them one by one, enabling it to offer atomicity without needing locking
- Shards implement state machine replication (SMR) for **put.** For **get** you just send an RPC to any random member of the shard

WITH BIG DATA, A KEY-VALUE STORE IS LIKE A COLLECTION OF MINI-DATABASES

We might have gigabytes of data in each shard. An Intel server can host 64GB of memory, and we could fill this completely!

A new generation of "persistent" memory will expand this limit to 256GB soon, and even more down the road, still with DRAM access speed

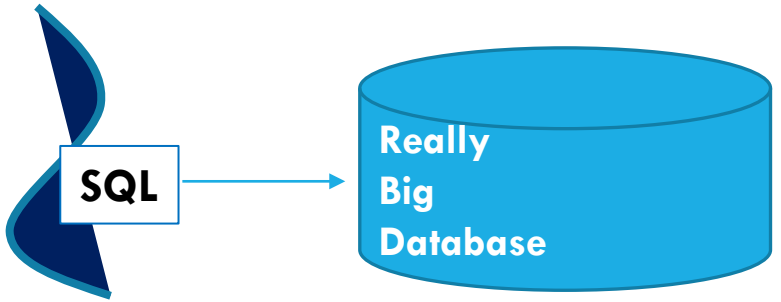
… BUT ARE THEY REAL DATABASES?

These mini-databases offer O(1) in-memory speed!

We hash to figure out which shard… then use an O(1) lookup structure to find the actual object in this massive memory region.

Does it make sense to think of a **get** as doing something more elaborate than just finding the (key,value) tuple and sending it back to the caller?

WHERE DOES A "PROGRAM" RUN?



With SQL, you give your program (transaction) to the SQL database.

It compiles and executes the SQL code – think of the database system as a form of interpreter. Even if the SQL code was submitted from a client program, the SQL execution occurs inside the database.

So the program is run by the database on your behalf, and it can even send tasks to "workers" if the database is spread over multiple machines

LINQ: SQL EMBEDDING INTO CODE

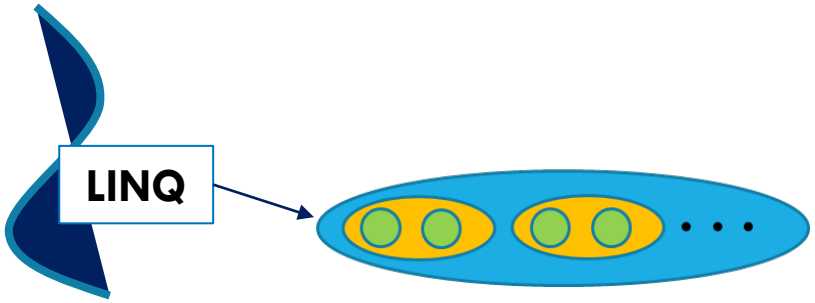
LINQ is the Language Integrated Query package from Microsoft

Allows a program to establish a binding (connection) to services like Cascade, CosmosDB, BLOB store, Databricks. These connectors need to support the concept of "iteration" over a collection (list) of tuples.

- For Cascade: "scan my (key,value) objects hosted locally".
- If your objects have a regular structure, this is exactly like scanning a database – and we can do SQL queries. LINQ offers this model.
- But those queries will run in just he one server

WHERE DOES THE "PROGRAM" RUN?

In this illustration, LINQ is part of some client program, and run entirely inside the client

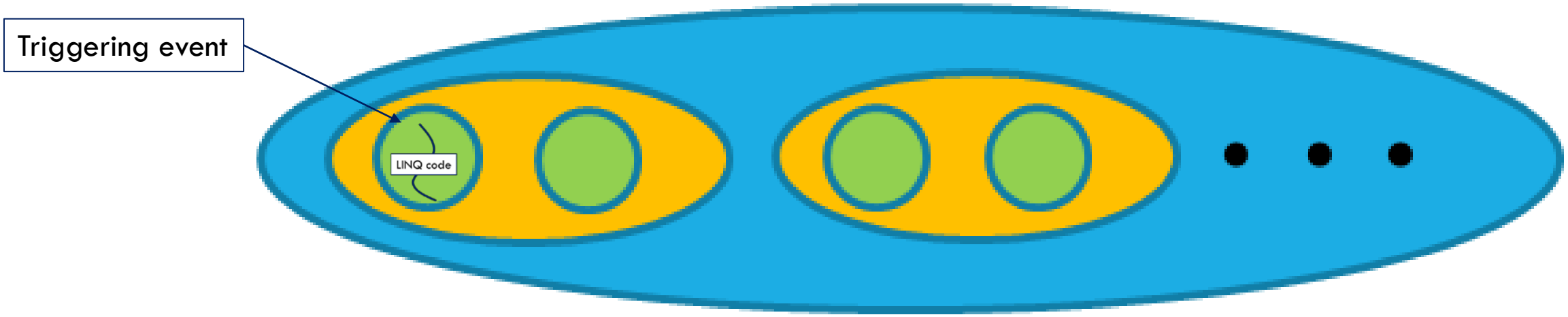


The key-value store is "accessed" purely via **put** and **get**. Any data the LINQ logic "looks at" must be fetched tuple by tuple.

To scan a lot of data items, must fetch them first, one by one.

OTHER OPTIONS TO CONSIDER

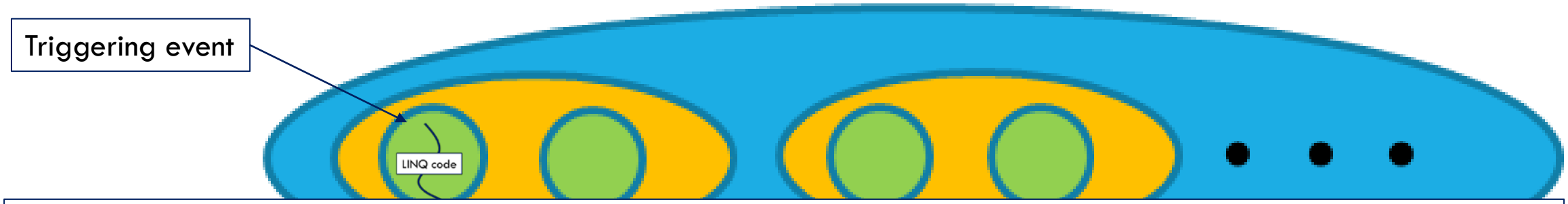
In Cascade, Cornell's experimental key-value store, client code can be recompiled as a "UDL" and dynamically loaded into Cascade itself.



When triggered by some event, the user's UDL runs. Now LINQ will be "talking to the Cascade server" from inside one instance. Any **get** operation that can be handled locally will be very cheap. A **get** that needs data from some other shard will be performed by sending an RPC to fetch it. (Same for **put**)

OTHER OPTIONS TO CONSIDER

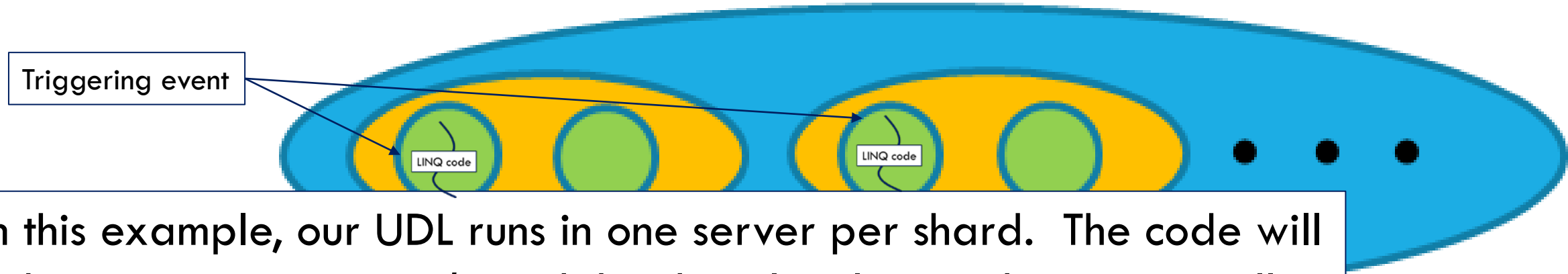
In Cascade, Cornell's experimental key-value store, client code can be recompiled as a "UDL" and dynamically loaded into Cascade itself.



When triggered by some event, the user's UDL runs. Now LINQ will be "talking to the Cascade server" from *inside one instance*. Any **get** operation that can be handled locally will be very cheap. (Remote **get** or **put** will still trigger network I/O)

WE COULD ALSO TRIGGER THE SAME CODE, BUT ON MULTIPLE SHARDS, CONCURRENTLY

The trigger could be a multicast or a series of point-to-point requests



In this example, our UDL runs in one server per shard. The code will be running *concurrently* and the data local to each instance will differ from shard to shard!

HTTP://WWW.CS.CORNELL.EDU/COURSES/CS5412/2022FA 12

… LINQ IS LIKE A SOFTWARE LIBRARY

It is used by a program as it executes, and lets the program treatlists and other collections as in-memory "relational tables"

Most programmers do know SQL and find it powerful and intuitive.
LINQ enables us to use SQL on this internal data structures, too

Same remark applies to Pandas, PyTorch, Tensor Flow, Spark. All of them are drawn to the SQL model because it is convenient.

REVISITING A QUESTION FROM SLIDE 5

We asked if it makes sense to think of a shard as a mini-database

We can see now that sometimes, for a key-value store, this might be a sensible thing to do! (If the key-value store holds multiple kinds of data, you could first filter to focus on "one table"….)

Best to cache that remote data, if you do this!

With LINQ we can run any kind of database-like logic on the local shard, *and it can even use* ***get*** *to fetch some data from remote shards.*

STAYING IN THE SPIRIT OF A KEY-VALUE MODEL

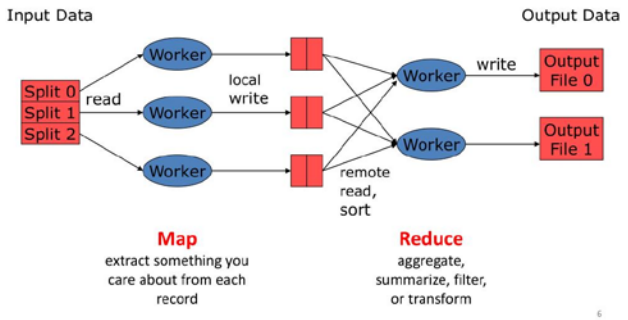
Cloud builders don't view a key-value sharded store as a true database.

- They keep the sharding policy in mind when they mentally visualize their system, and think of NoSQL as a convenient way to express data transformations and filtering
- Fit is great for big collections, social network graphs stored as key-value data
- … but not for a genuine relational database split into lots of shards

With LINQ triggered on multiple shards, the developer is leveraging parallelism.

- Concurrency rules the cloud: "embarrassing parallelism" when searching or transforming the entire big-data collection
- Our goal is really fast individual get operations, parallel search/transformation

THE MAPREDUCE PATTERN



Google was first to explore this idea of running the same code but on sharded data, doing purely local data access in parallel.

They had huge collections of web pages and needed to run algorithms to decide which web sites to return when a user issues a search. Their developers quickly ran into a problem: parallel computing on shards is easy, but sometimes we need a way to merge the subresults.

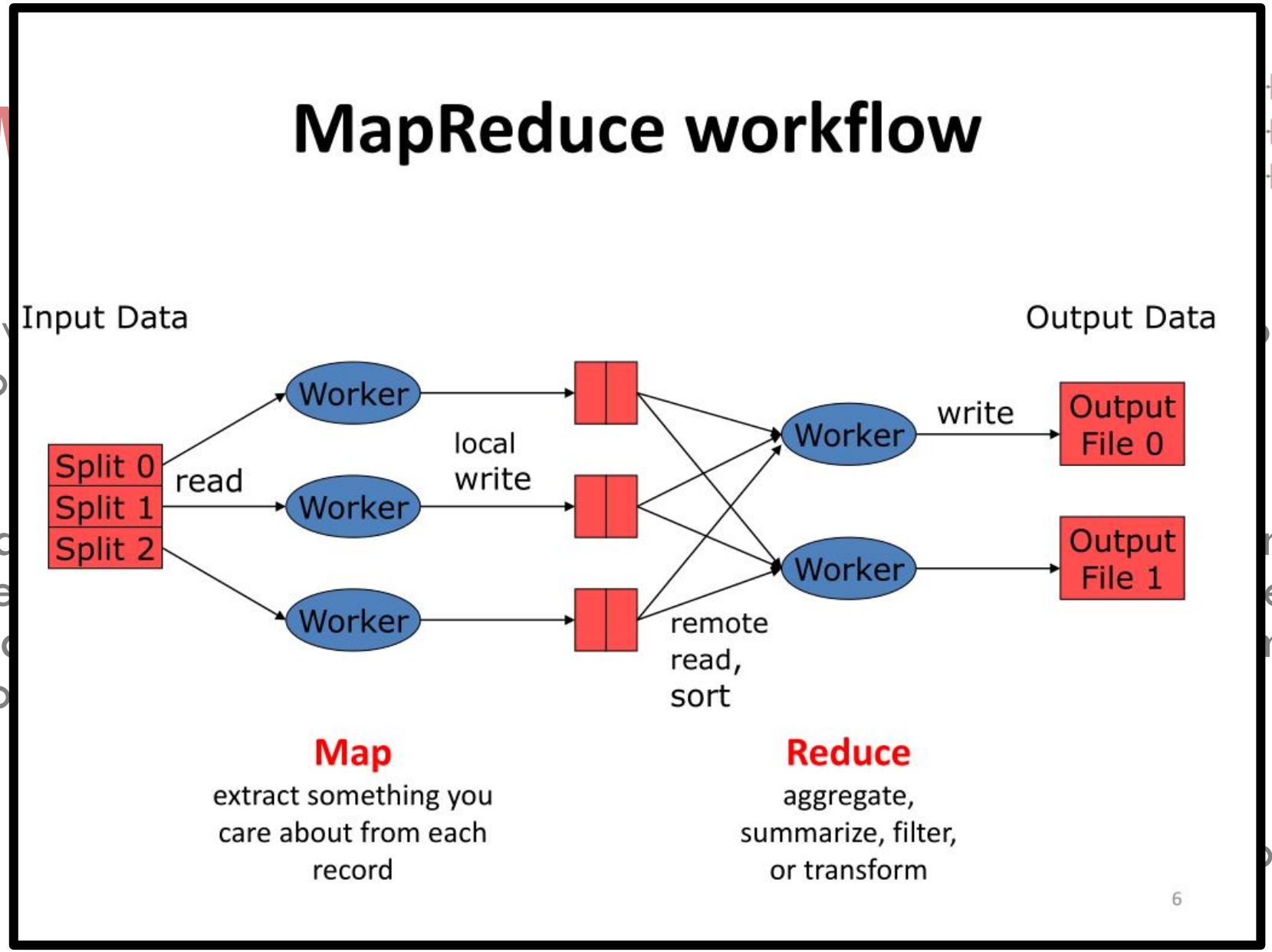
- All at one place? But this could overwhelm that server
- Keep results sharded? They favored this, but it needs a special coding style

THE M

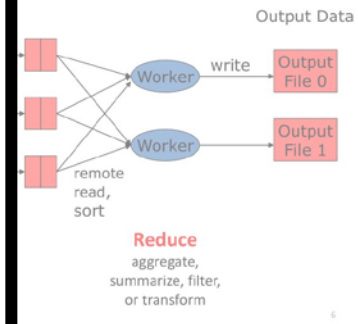
Google v … n sharded data, do …

They hac … ms to decide which we … ers quickly ran into … mes we need a way to …

- All at …
- Keep … oding style

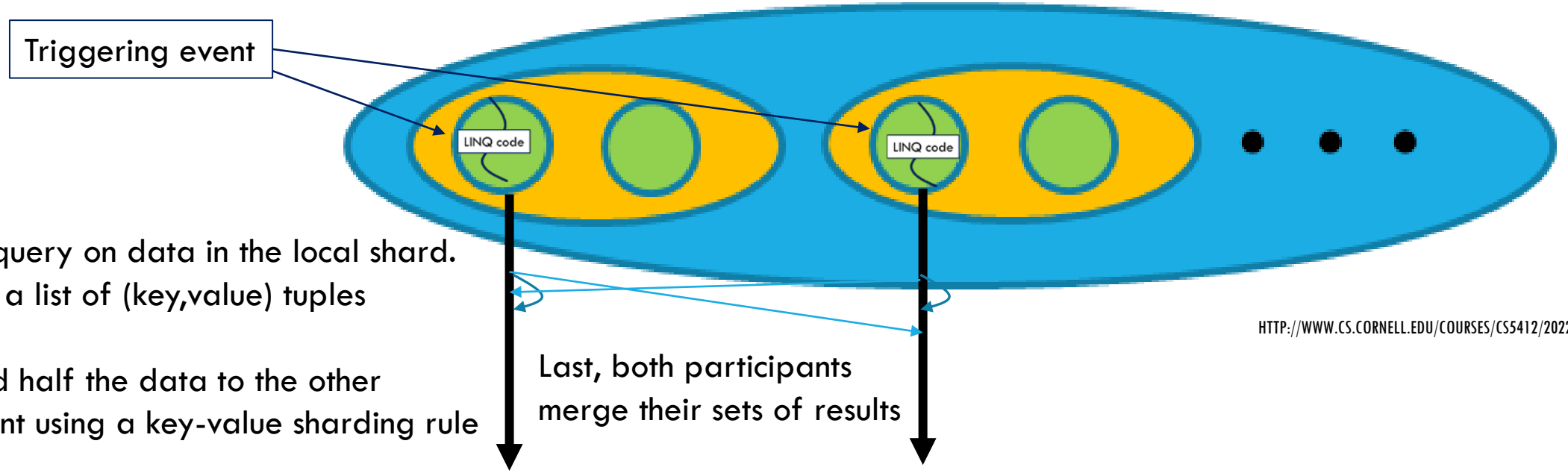


MapReduce workflow



MAPREDUCE ON KEY-VALUE DATA

MapReduce with a simple shuffle, group-by, and reduce step. We will revisit this in full detail two weeks from now.



First run query on data in the local shard. Results in a list of (key,value) tuples

Then send half the data to the other participant using a key-value sharding rule

Last, both participants merge their sets of results

HTTP://WWW.CS.CORNELL.EDU/COURSES/CS5412/2022FA 18

MAPREDUCE SUMMARY

1) [**map**] Request triggers computation by one member of each shard.

2) [**compute**] Each runs code locally, processes their local "share" of the data

3) [**shuffle**] Output is in the form of (key,value) tuples. Hash the keys to figure out which to keep and which to send to some other shard. Called a "shuffle"

4) [**reduce**] Collect all the incoming data, group it by key, then combine ("reduce") the values so that for each key, you keep just one result.

WHEN TO USE MAPREDUCE

Easiest if the computation is embarrassingly parallel:

Apply some function to every data item, group results by a shared key, combine the subresults – and leave all of it sharded.

Much harder for tasks that are inherently entangled. For example, a MapReduce version of binsort/mergesort must estimate bin sizes, and that turns out to be tricky – yet sorting individual bins and merging is easy.

CONCEPT: THE NoSQL MODEL

NoSQL is a term for any system that supports SQL but really operates on sharded data. Examples: AWS DynamoDB, Azure CosmosDB, Cassandra, DB Rocks, MemCacheD.

NoSQL inherently has limitations: These SQL queries are not true SQL on a single database… they do not have full transactional semantics (no ACID), they cannot combine data from multiple shards, no locking occurs.

With **put**, we normally just send the update to both replicas via a state machine replication protocol. They can then perform the identical changes.

THE DATA LINQ OPERATES ON IS IN MEMORY IN YOUR OWN ADDRESS SPACE

With an SQL model, the program describes a transaction on the full database, which is "out there". The database itself plans an execution strategy.

With NoSQL from LINQ or similar packages, you fetch data into your address space using **get** (or perhaps some form of **multi-get** to pull a batch of content all at once), then process it in-memory inside your program. Your LINQ expression determines the execution strategy.

Cascade is unusual in adding an extra option: "send my LINQ code to the servers hosting my data, then run it in one shard, or many shards."

LINQ WON'T NEED LOCKING

Normal SQL needs locks to deal with concurrency, like if we had multiple threads each running SQL code at the same time.

In the LINQ-style NoSQL model, our code is generally not concurrent (not multithreaded) and does the entire operation in one shot.

No locks are needed! Jim Gray's concerns wouldn't apply in this situation.

LINQ REQUIRES "STRUCTURED" 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

STRUCTURED AND UNSTRUCTURED DATA

When we first introduced key-value storage we talked about storing pretty much anything into a DHT. We would consider most data to be unstructured.

But there are many tools to convert unstructured data to structured data.

For example, given a web page, we could extract all the n-grams, or all the URLs we find in it. Given dairy documents, we could extract one-row summaries using an OCR and then a parser.

A TABLE IS STRUCTURED

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. Tools can introduce these sorts of issues

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

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 table

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!

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

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
- Looks just like SQL
- Data could be entirely local or we could use **get** to fetch data from a mix of the local shard and remote shards

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;
```

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”
```

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();
```

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);
}
```

… A SAMPLING OF 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()`

Fancy stuff:

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

JOINS EXIST TOO

The confusing thing to think about is how a JOIN will work if we have sharded data, run our LINQ queries concurrently in each shard, and each query is accessing just the local data.

We would be treating each shard as a distinct mini-database

… but we can do much more if we want to get fancier!

JOINS EXIST TOO

Suppose that we have one table, maybe CName from the lecture last week, and we fully replicate it across all the key-value servers.

Every shard now has a copy of CName.

In this case a *local* LINQ query could still compute the entire JOIN, in pieces: each shard would generate one piece of the full JOIN result.

MORE MISSING THINGS

An aggregation query like SUM or AVG will compute the local answer but not the global one.

Here we need to send our partial results to one node, to combine them.

Easy to do, but only if you can visualize the pattern you are requesting!

REQUIRED STEPS

Application must be expressed in a way where NoSQL is sufficient

There is no automatic transformation from SQL on a full database to a sharded set of NoSQL actions on local portions of the data!

If you can't shard the task and write it with local SQL logic, you won't be able to use this method!

REQUIRED STEPS

Application also 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!

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. Assumes that you understand the semantics of LINQ with null values.

Some databases instead have a default value for missing data, like -99

VISITING FOUR GOOD WEB SITES

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

Complete list of all LINQ operators

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

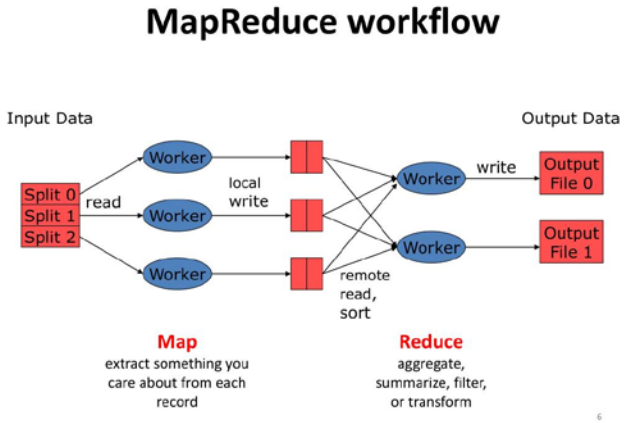
HOW DO LINQ, PANDAS, PYTORCH, SPARK, ETC. RELATE?

Almost every modern programming language or system offers a way to embed some form of NoSQL notation into your code.

The concept is universal: collections (lists, often of tuples, often treated as key-value "pairs" or as relations), transformed in a functional manner.

But the detailed syntax will vary, as will the mechanisms for binding to a storage server that holds data that can be viewed as a collection.

SUMMARY



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

Tools like Pandas, PyTorch and LINQ make it easy to compute on this data, especially if it has a regular structure.

Requires new thinking about what it means for data to be a database. LINQ and other NoSQL (perhaps + MapReduce) are very useful.