Automating Semantic Data Production Pipelines

David Booth, Ph.D.

PanGenX

Semantic Technology Conference

San Francisco

June 2012

- DRAFT -

Please download the latest version of these slides:

http://dbooth.org/2012/pipeline/

Speaker background

- **David Booth, PhD:**
  - **Software architect, PanGenX**
  - **Cleveland Clinic 2009-2010**
  - **HP Software & other companies prior**
  - **Focus on semantic web architecture and technology**

PanGenX: Enabling personalized medicine

- **Pharmacogenetics is key**
- **Knowledge-as-a-Service**
- **A proprietary, scalable knowledgebase, analytics engine, and decision-support tool**
- **Cloud accessible**
- **Suitable for many applications**
- **Customizable for each therapeutic area**

Architectural strategy for semantic data integration

1. Data production pipeline

- **E.g., Genomic, phenotypic, drug, patient records, outcomes, etc.**

2. Use RDF in the middle; Convert to/from RDF at the edges

- **Good for integration, inference and context/provenance (with named graphs)**

3. Use ontologies and rules for semantic transformations

- **SPARQL is convenient as a rules language**

What is the RDF Pipeline Framework?

- **Framework for automated data production pipelines**
- **Intended for big data integration**
- **Designed for RDF, but data agnostic**
- **Open source – Apache 2.0 license**

Not:

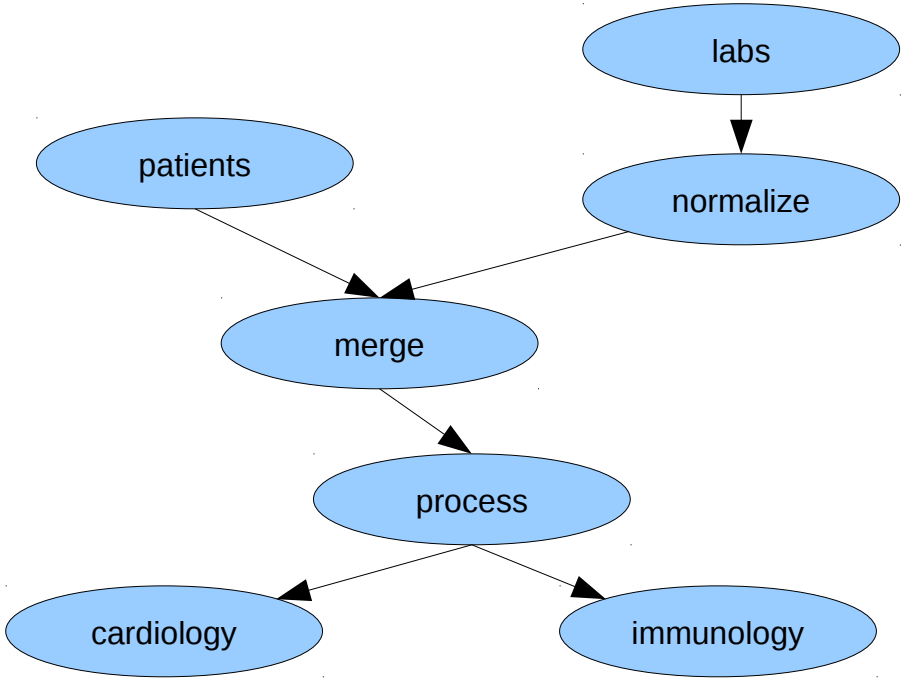
- **A universal data model approach**
- **A workflow language**
  - **No flow-of-control operators**

Major features

- **Distributed, decentralized**
- **Loosely coupled**
  - **Based on RESTful HTTP**
- **Easy to use**
  - **No API calls (usually)**
- **Efficient**
  - **Automatic caching**
  - **Dependency graph avoids unnecessary data regeneration**
- **Programming language agnostic**

Caveat: Still in development. No official release yet.

Example pipeline



- **Pipeline: set of nodes in a data flow graph**
- **Nodes process and store data**

Example pipeline definition (in Turtle)

```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/node/> .
3.  :patients a p:FileNode .
4.  :labs a p:FileNode .
5.  :normalize a p:FileNode ;
6.      p:inputs ( :labs ) .
7.  :merge a p:FileNode ;
8.      p:inputs ( :patients :normalize ) .
9.  :process a p:FileNode ;
10.     p:inputs ( :merge ) .
11.  :cardiology a p:FileNode ;
12.     p:inputs ( :process ) .
13.  :immunology a p:FileNode ;
14.     p:inputs ( :process ) .
```

Related work

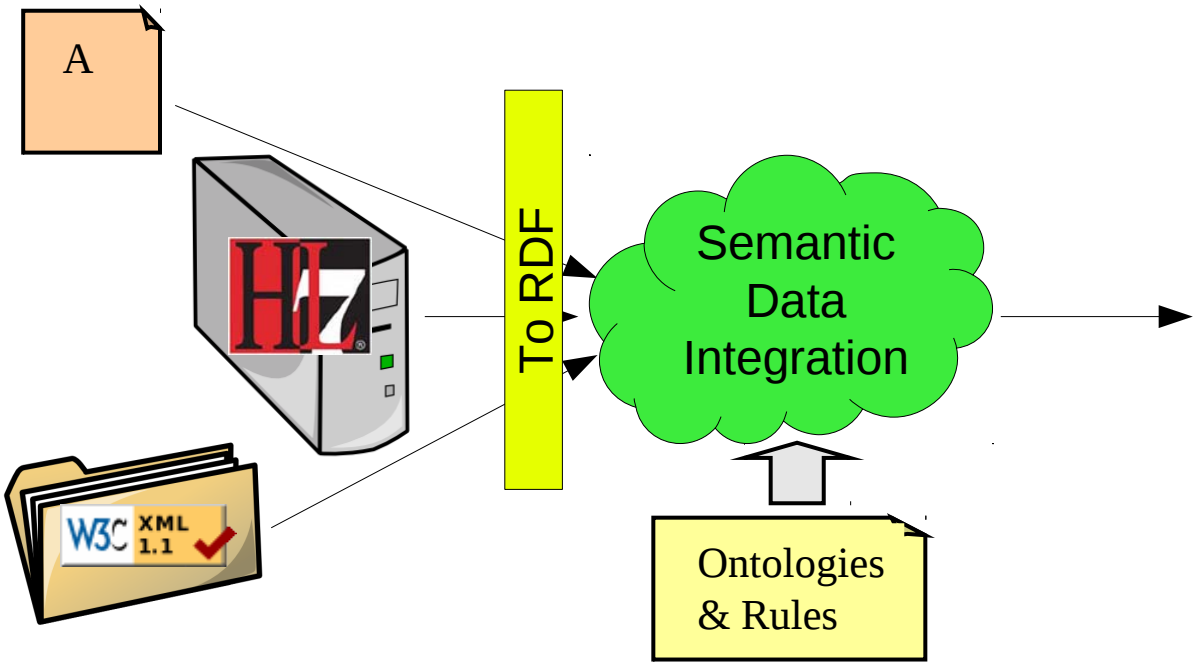
- **Linked Data Integration Framework (LDIF)**
  - **“Translates heterogeneous Linked Data from the Web”**
  - **http://www4.wiwiss.fu-berlin.de/bizer/ldif/**
  - **Similarities: Pipeline framework for RDF data**
  - **Differences: Central control; Synchronous; Hadoop; Oriented toward Linked Data**
- **Sparql Motion, from Top Quadrant**
  - **A “visual scripting language for semantic data processing”**
  - **http://www.topquadrant.com/products/SPARQLMotion.html**
  - **Similarities: Easy to visualize; Easy to build a pipeline**
  - **Differences: Central control & execution; Not cache oriented**
- **DERI Pipes**
  - **A “paradigm to build RDF-based mashups”**
  - **http://pipes.deri.org/**
  - **Similarities: Very similar goals**
  - **Differences: XML pipeline definition; Central control; Not cache oriented**
- **Hadoop**
  - **Implementation of map-reduce algorithm**
  - **http://hadoop.apache.org/**
  - **Similarities: Distributed data production**
  - **Differences: Much more mature; For parallelizing big data analysis; Java based**
- **Oozie**
  - **“Workflow/coordination service to manage data processing jobs for Apache Hadoop™”**
  - **http://rvs.github.com/oozie/index.html**

Related work (cont.)

- **NetKernel**
  - An “implementation of the resource oriented computing (ROC)” – think REST
  - http://www.1060research.com/netkernel/
  - Similarities: Based on REST (REpresentation State Transfer)
  - Differences: Lower level; Expressed through programming language bindings (Java, Python, etc.) instead of RDF
- **Propagators, by Gerald Jay Sussman and Alexey Radul**
  - Scheme-based programming language for propagating data through a network
  - http://groups.csail.mit.edu/mac/users/gjs/propagators/revised-html.html
  - Similarities: Auto-propagation of data through a network
  - Differences: Programming language; Finer grained; Uses partial evaluation; Much larger paradigm shift
- **Enterprise Service Bus (ESB)**
  - http://soa.sys-con.com/node/48035#
  - Similarities: Similar problem space
  - Differences: Central messaging bus and orchestration; Heavier weight; SOA, WS*, XML oriented; Different cultural background
- **Extract, Transform, Load (ETL)**
  - http://www.pentaho.com/
  - Similarities: Also used for data integration
  - Differences: Central orchestration and storage; Oriented toward lower level format transformations

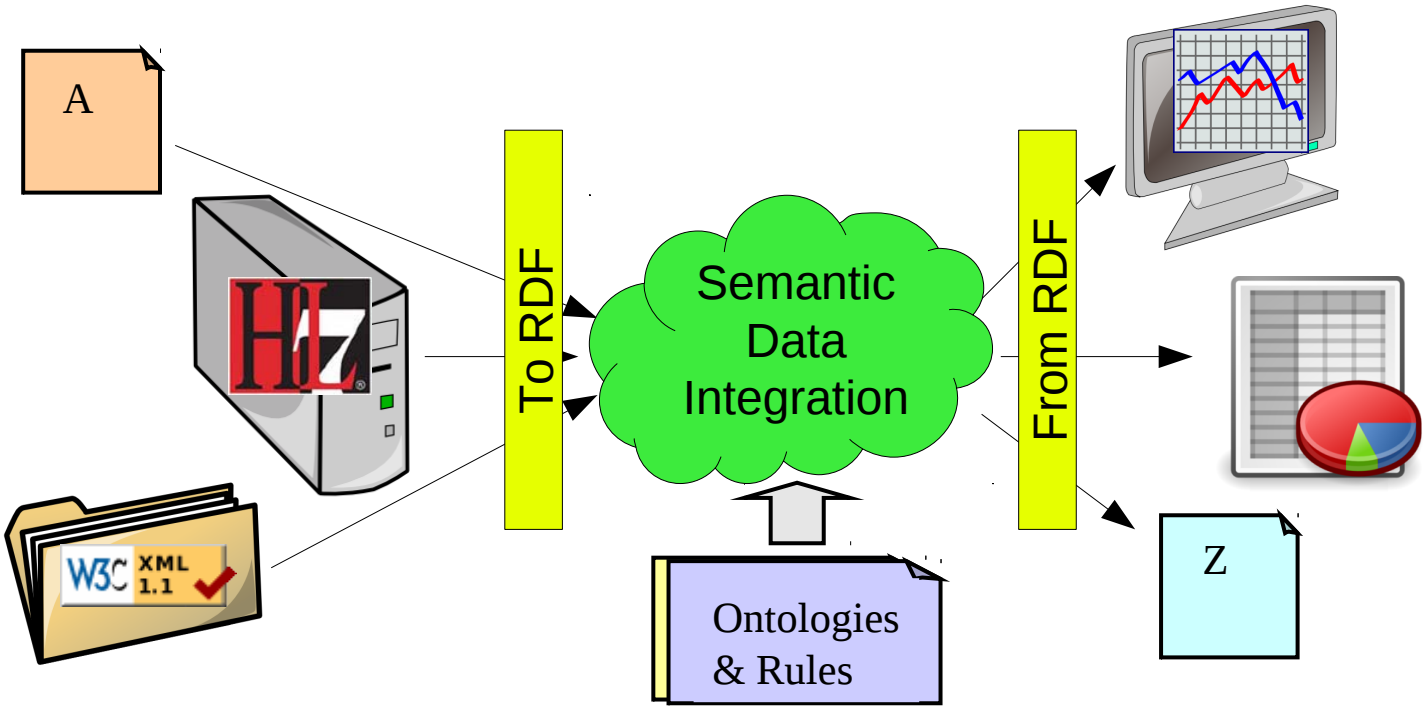
Why a data production pipeline?

Problem 1: Multiple, diverse data sources



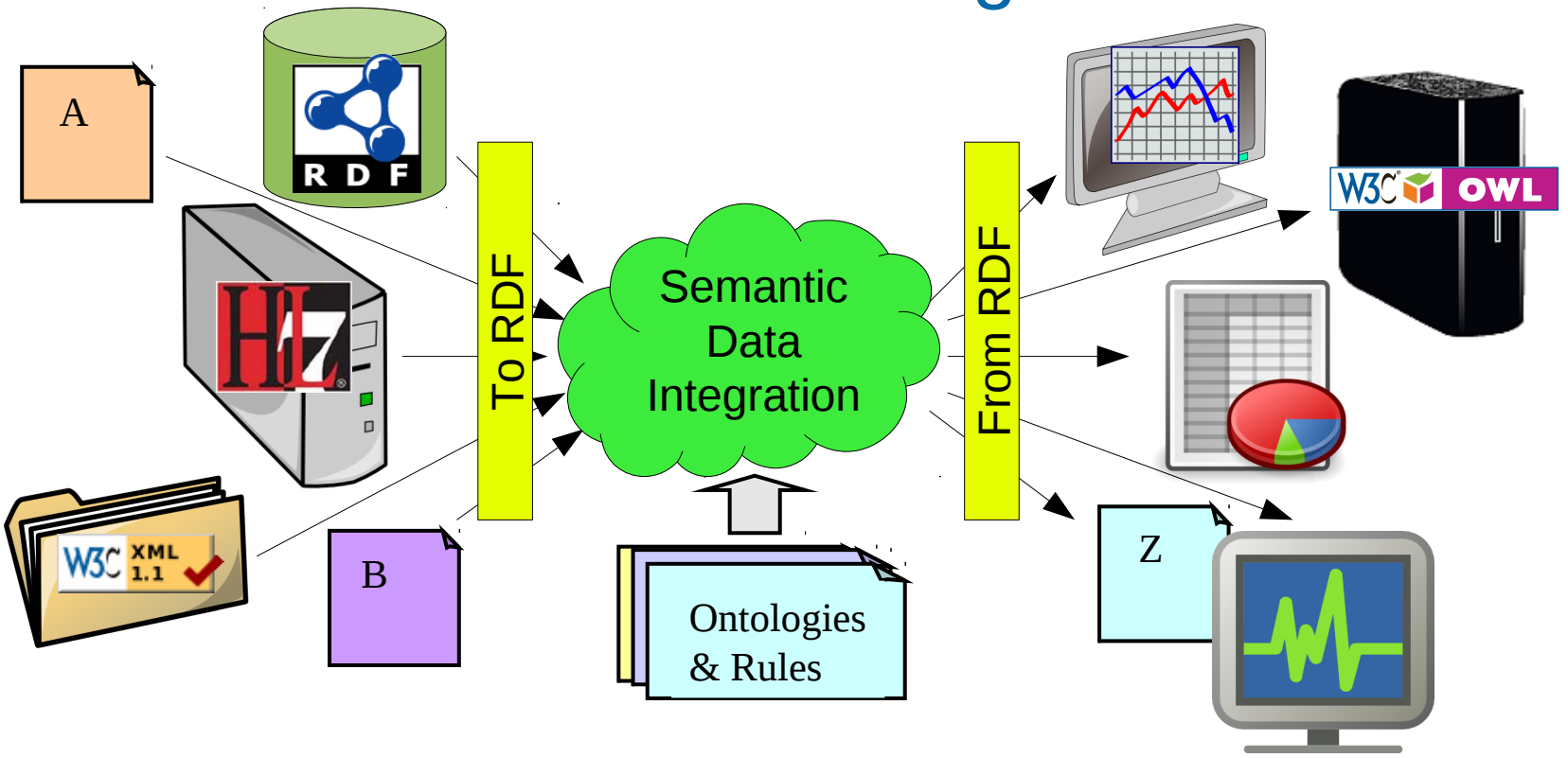
- **Different technologies, protocols and vocabularies**
- **Solution:**
  - **Convert to RDF at the edges**
  - **Use ontologies and rules to transform to hub ontology**

Problem 2: Multiple, diverse applications



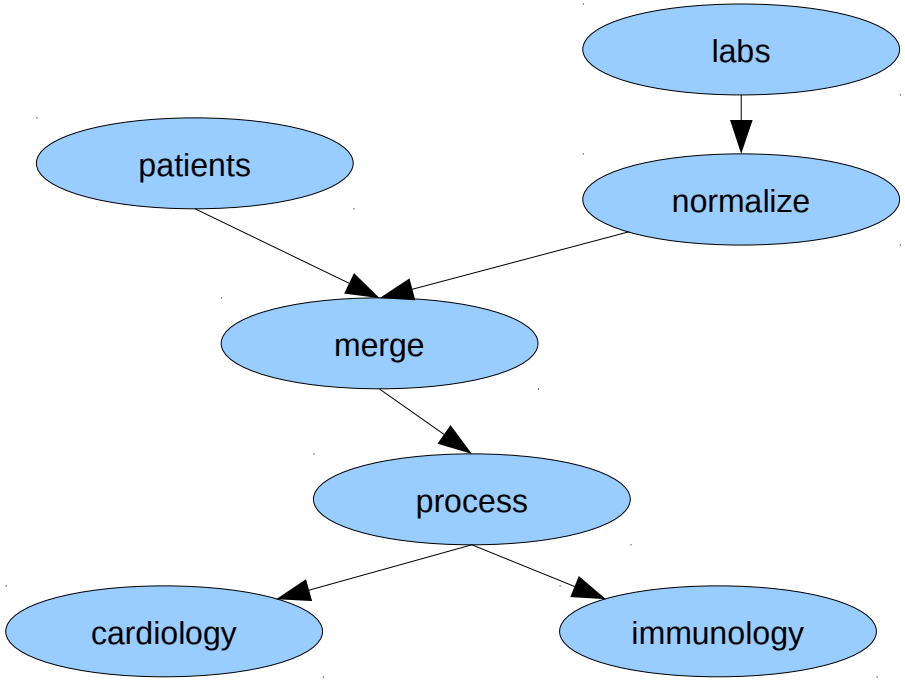
- **Often must support multiple, diverse data sinks**
- **Solution:**
  - **Use ontologies and rules to transform to presentation ontologies**
  - **Convert from RDF at the edges**

Problem 3: Evolving needs



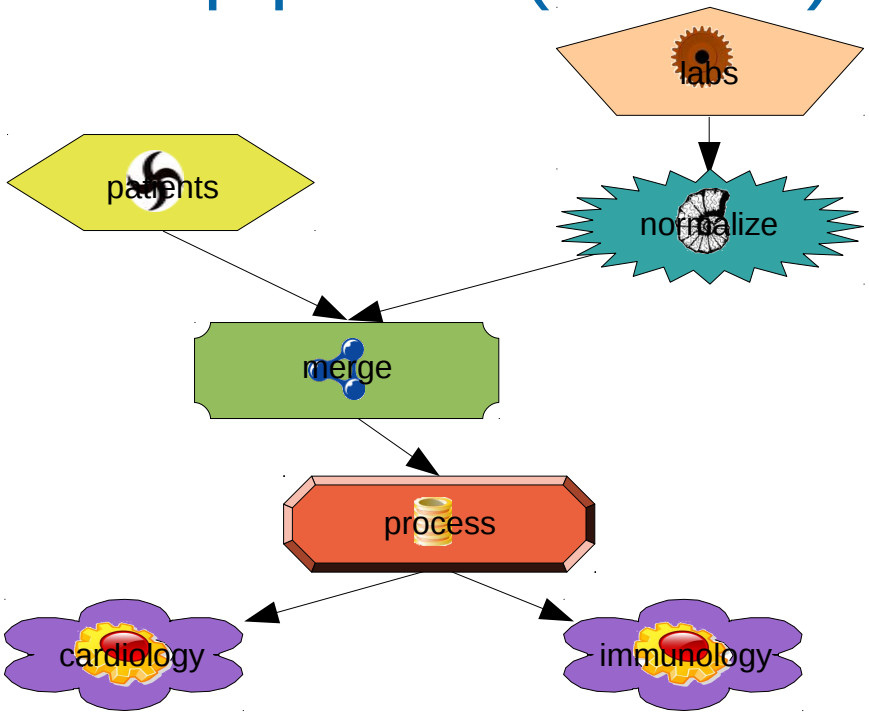
- **New data sources and sinks get added**
- **Transformations get complex, involve several steps**
- **Solution: Data pipeline . . .**

Conceptual data pipeline



- **Pipeline: set of <u>nodes</u> in a data flow graph**
- **Nodes process and store data**
- ***But the reality is often different . . .***

Data pipeline (ad hoc)

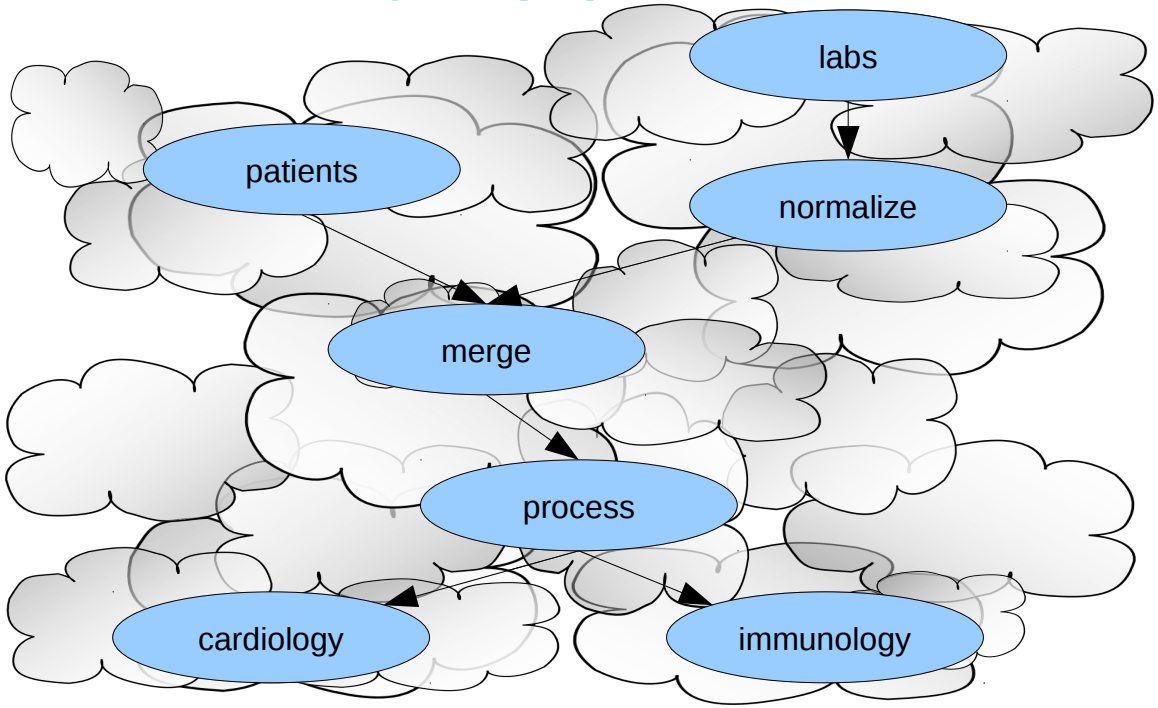


- **Typically involves:**
  - **Mix of technologies: shell scripts, SPARQL, databases, web services, etc.**
  - **Mix of formats – RDF, relational, XML, etc.**
  - **Mix of interfaces: Files, WS, HTTP, RDBMS, etc.**

Problem 4: Ad hoc pipelines are hard to maintain

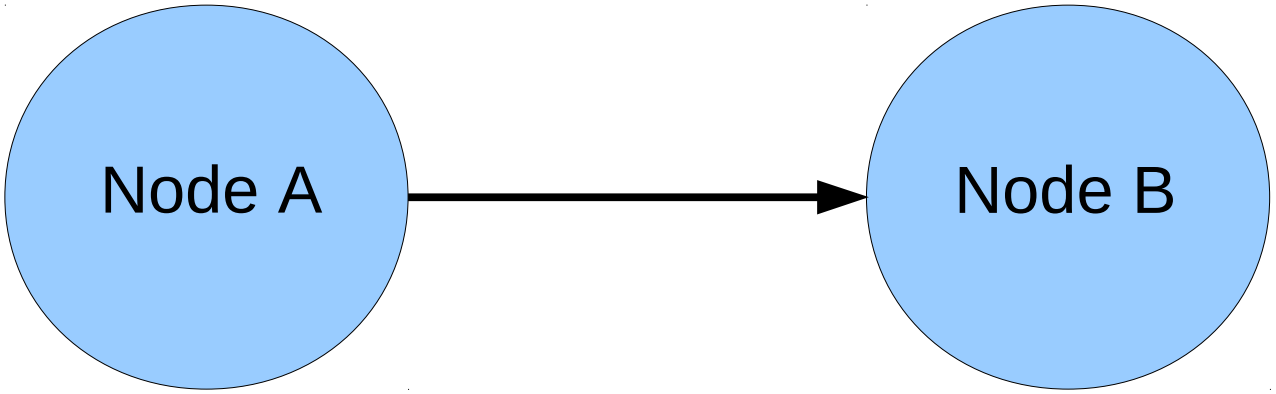
- **Pipeline topology is implicit, deep in code**
  - **Often spread across different computers**
- **Hard to get the “big picture”**
- **Complex & fragile**
- **Solution: Pipeline language / framework**

Pipeline using a pipeline framework



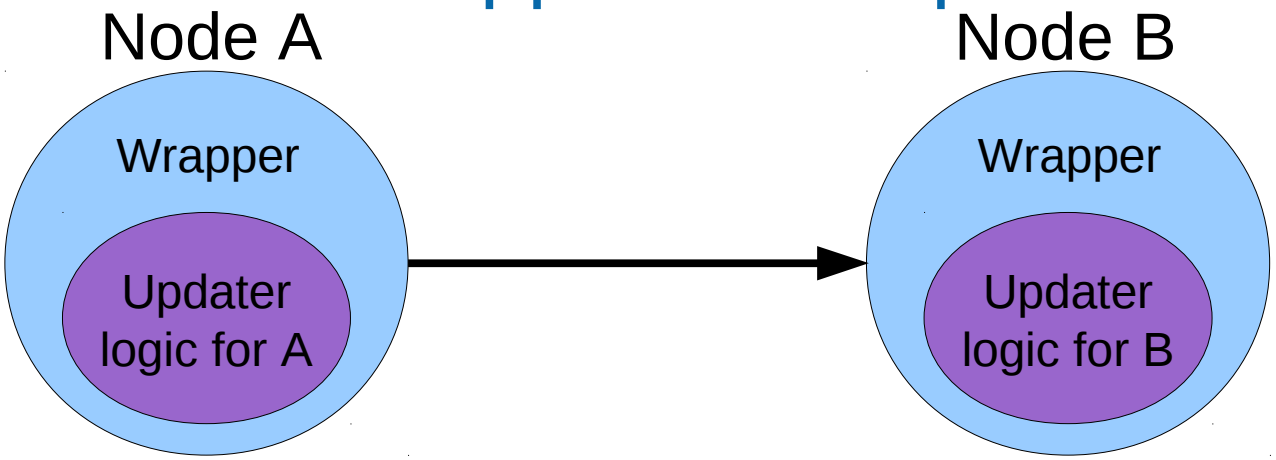
- **Pipeline is described explicitly**
- **Framework runs the pipeline**
- **Easy to visualize – generate a picture!**
- **Easy to see dependencies**
- **Easier to maintain**

Problem 5: Diverse processing needs



- **Nodes perform arbitrary processing**
- **RDF is not the only tool in your toolbox**
- **Diverse data objects: RDF graphs, files, relational tables, Java object, etc.**
- **Diverse processing languages: Java, shell, SPARQL, etc.**
- **Need to choose the best tool for the job**
- **Solution: Node wrappers**

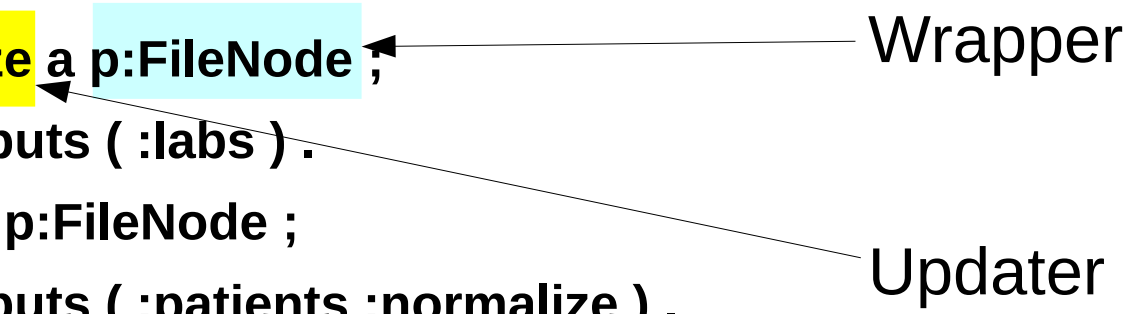
Node wrappers and updaters



- **Node consists of:**
  - **Wrapper – supplied by framework (or add your own)**
  - **Updater – your custom code**
- **Wrapper handles:**
  - **Inter-node communication**
  - **Updater invocation**
- **Updater can use any kind of data object or programming language, given an appropriate wrapper**

Example pipeline definition (in Turtle)

```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/node/> .
3.  :patients a p:FileNode .
4.  :labs a p:FileNode .
5.  :normalize a p:FileNode ;
6.      p:inputs ( :labs ) .
7.  :merge a p:FileNode ;
8.      p:inputs ( :patients :normalize ) .
9.  :process a p:FileNode ;
10.     p:inputs ( :merge ) .
11.  :cardiology a p:FileNode ;
12.     p:inputs ( :process ) .
13.  :immunology a p:FileNode ;
14.     p:inputs ( :process ) .
```

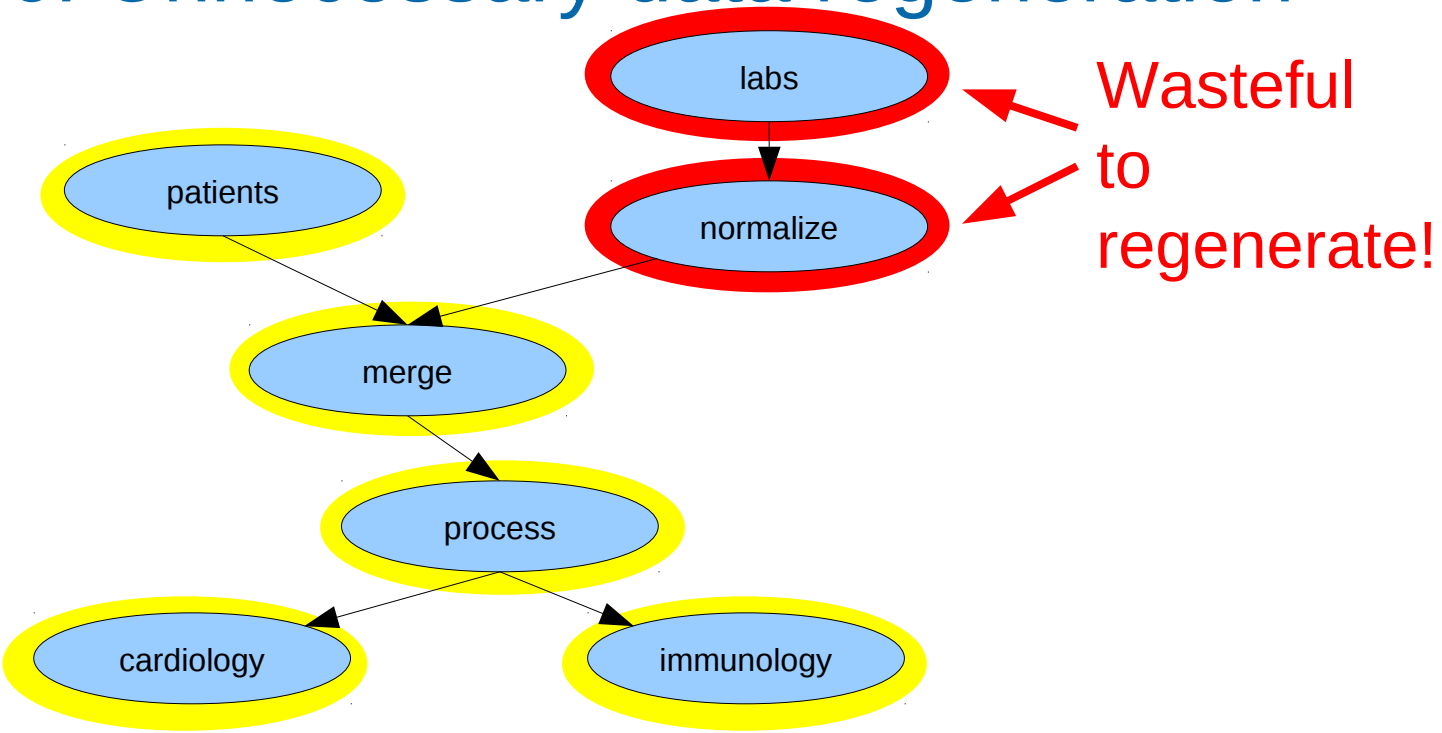


Basic wrappers

- **FileNode**
  - **Updater is a command that generates a file**
  - **E.g., shell script**
- **GraphNode**
  - **Updater is a SPARQL Update operation that generates an RDF named graph**
  - **E.g., INSERT**
- **JavaNode**
  - **Updater is a Java class that generates a Java object**

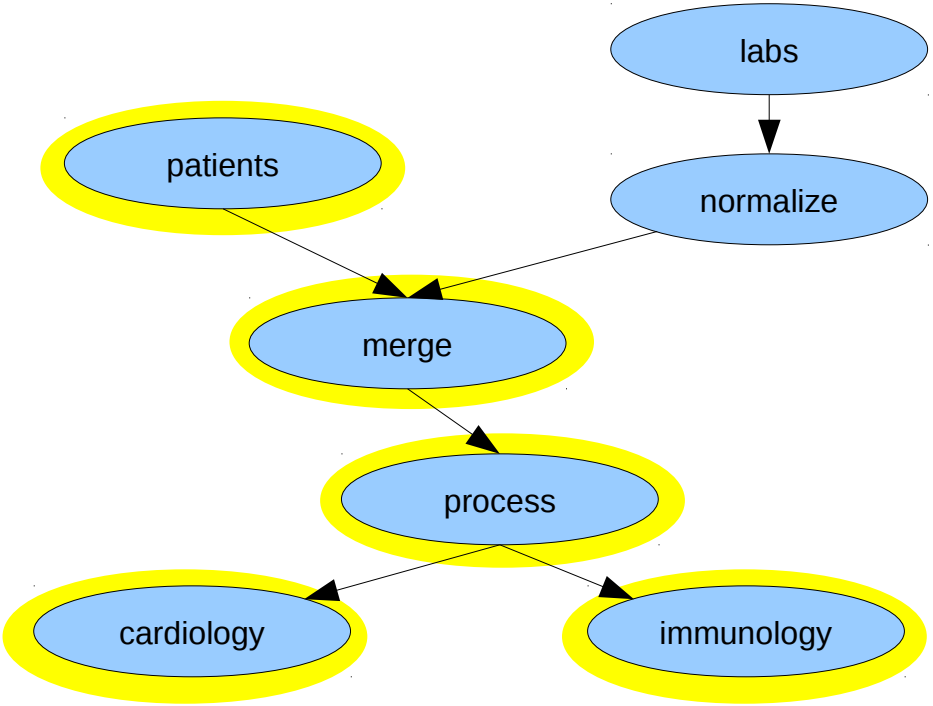
Other wrappers can also be plugged in, e.g., Python, MySql, etc.

Problem 6: Unnecessary data regeneration



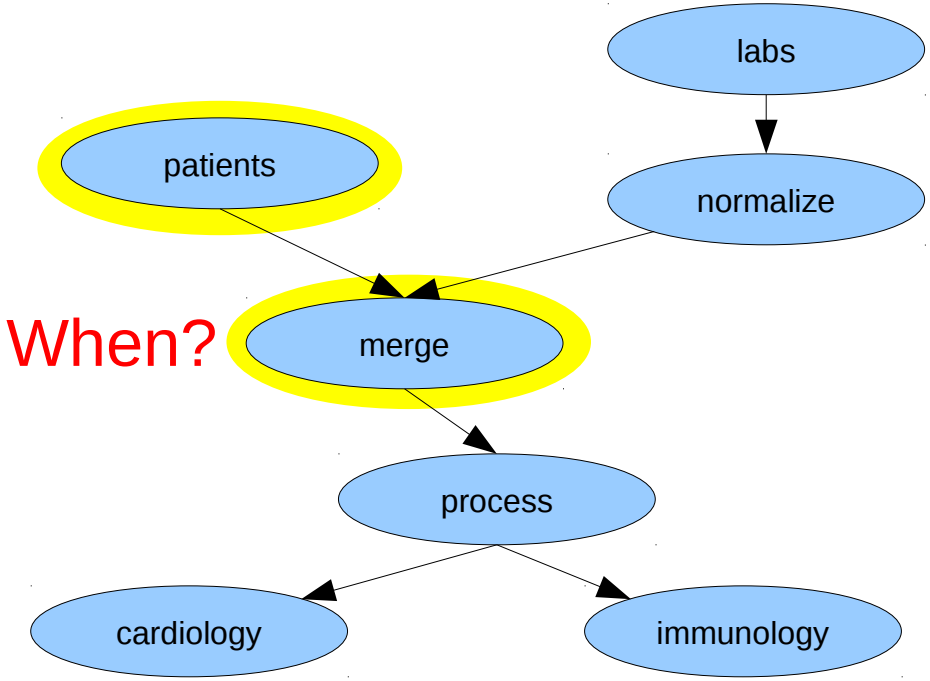
- **Wasteful and slow to re-run the entire pipeline when only one branch changed**
- **Solution: Use the dependency graph!**

Avoiding unnecessary data regeneration



- **Data is automatically cached at every node**
- **Pipeline framework updates only what <u>needs</u> to be updated**
  - **Think “Make” or “Ant”, but <u>distributed</u>**
- **Updater stays simple**

Problem 7: When should a node be updated?



- **Whenever any input changes? (Eager)**
- **Only when its output is requested? (Lazy)**
- **Trade-off: Latency versus processing time**
- **Solution: Update policy**

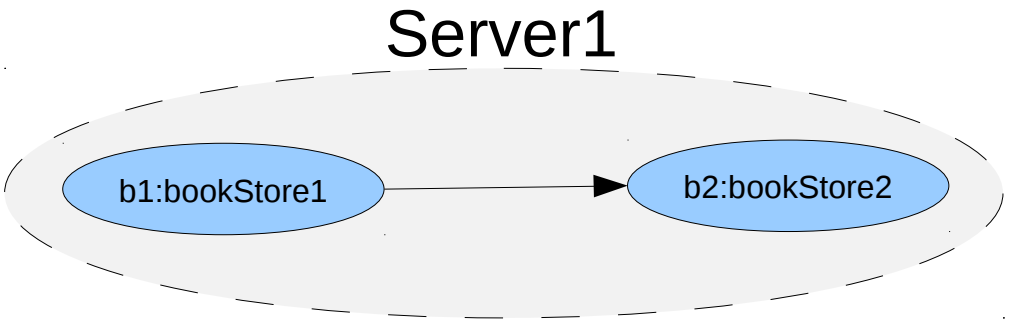
Update policy

- **Controls <u>when</u> a node's data is updated:**
  - **Lazy – Only when output is requested**
  - **Eager – Whenever the node's input changes**
  - **Periodic – Every *n* seconds**
  - **EagerThrottled – When an input changes but not faster than every *n* seconds**
  - **Etc.**
- **Handled by wrapper – independent of updater**
  - **Updater code stays simple – unpolluted**

Specifying an update policy

```
4.  . . .
5.  :normalize a p:FileNode ;
6.        p:updatePolicy p:eager ;
7.        p:inputs ( :labs ) .
8.  . . .
```

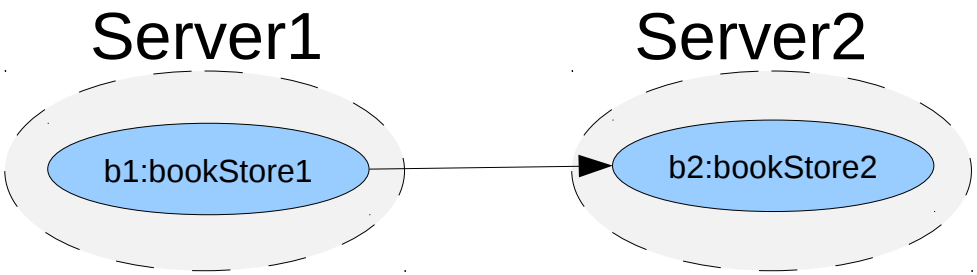
Problem 8: Distributing the processing




```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix b1: <http://server1/> .
3.  @prefix b2: <http://server1/> .
4.  b1:bookStore1 a p:JenaNode .
5.  b2:bookStore2 a p:JenaNode ;
6.      p:inputs ( b1:bookStore1 ) .
```

Same server

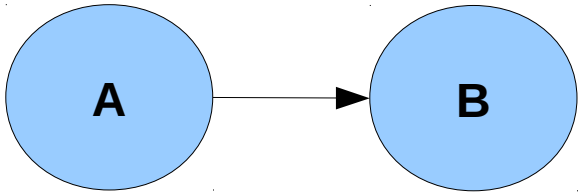
Distributed pipeline




```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix b1: <http://server1/> .
3.  @prefix b2: <http://server2/> .
4.  b1:bookStore1 a p:JenaNode .
5.  b2:bookStore2 a p:JenaNode ;
6.      p:inputs ( b1:bookStore1 ) .
```

Different server

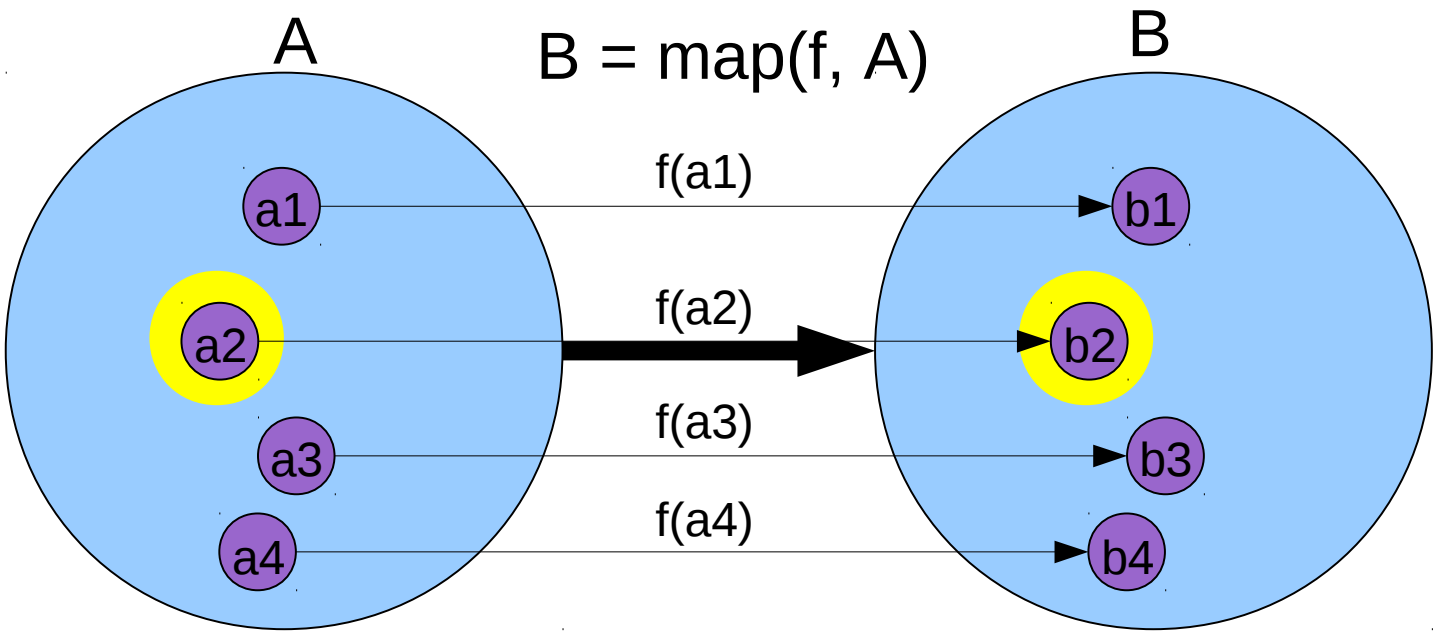
Problem 9: Efficiently scaling for big data



- **Problem: Big datasets take too long to generate**
  - **Wasteful to regenerate when only one portion is affected**
- **Observation: Big datasets can often be naturally subdivided into relatively independent chunks, e.g.:**
  - **Patient records in hospital dataset**
- **Solution: Map operator**

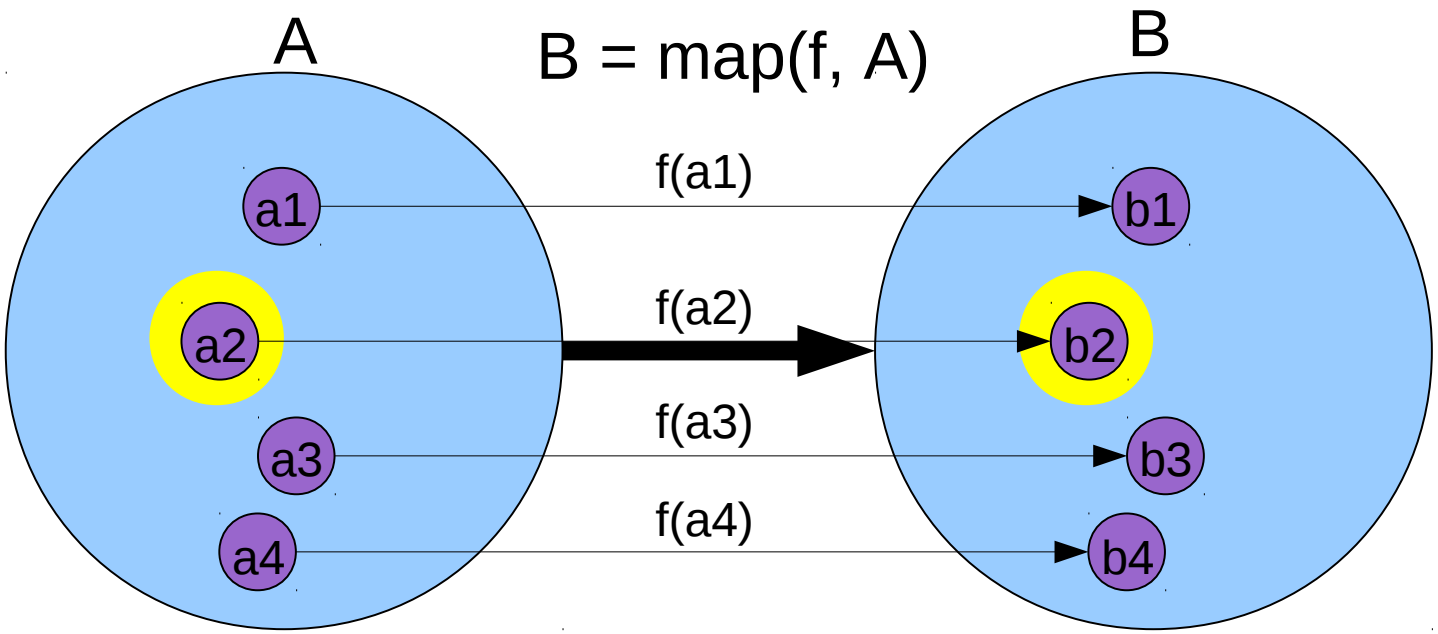
Caveat: Not yet implemented!

Generating one graph collection from another



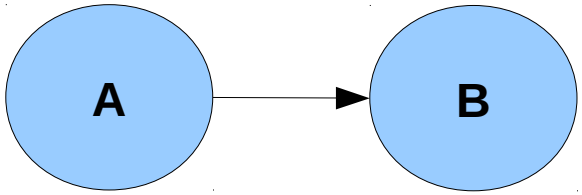
- **A and B contain a large number of items (chunks)**
- **Each item in B corresponds to one item in A**
- **The same function f creates each bi from ai:**
  - **foreach i, bi = f(ai)**

Benefits of map operator



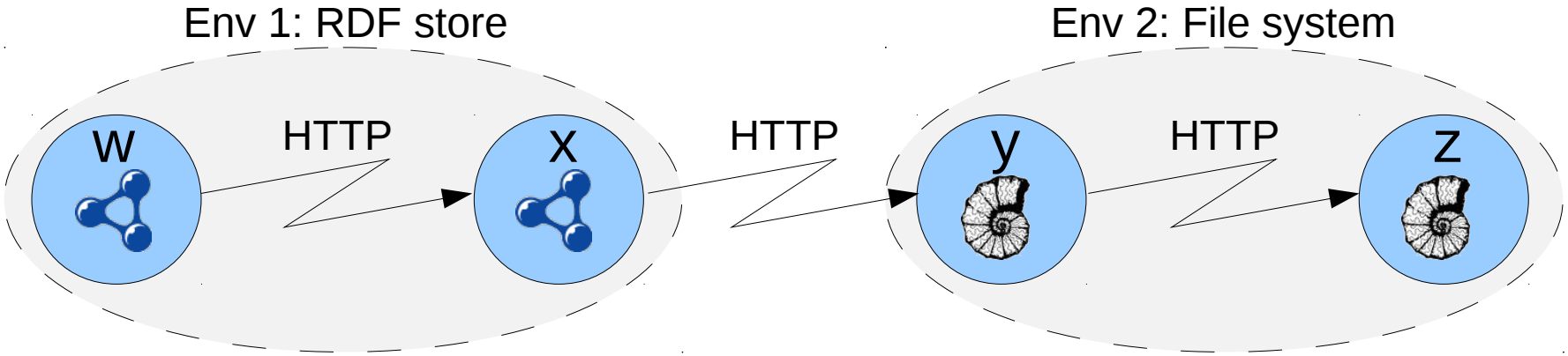
- **Work can be distributed across servers**
- **Framework only updates chunks that need to be updated**
- **Updater stays simple:**
  - **Only needs to know now to update one chunk**
  - **Unpolluted by special API calls**

Pipeline definition using map operator



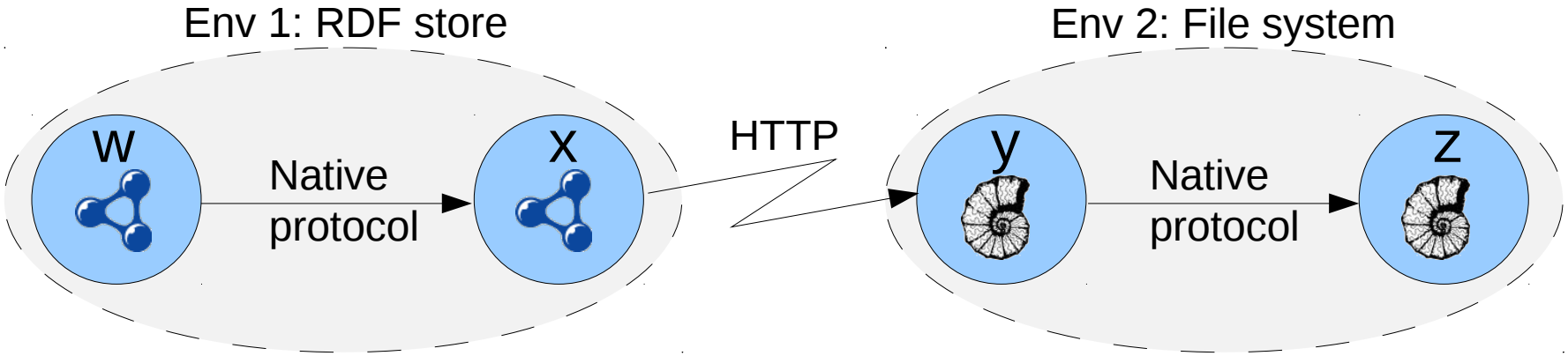

```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/> .
3.  :A a p:FileNode .
4.  :B a p:FileNode ;
5.      p:inputs ( ( p:map :A ) ) ;
6.      p:updater "B-updater" .
```

Problem 10: Optimizing local communication



- **Communication defaults to RESTful HTTP**
- **Inefficient to use HTTP between local objects, e.g.:**
  - **Files on the same server**
  - **Named graphs in the same RDF store**
  - **Java objects in the same JVM**

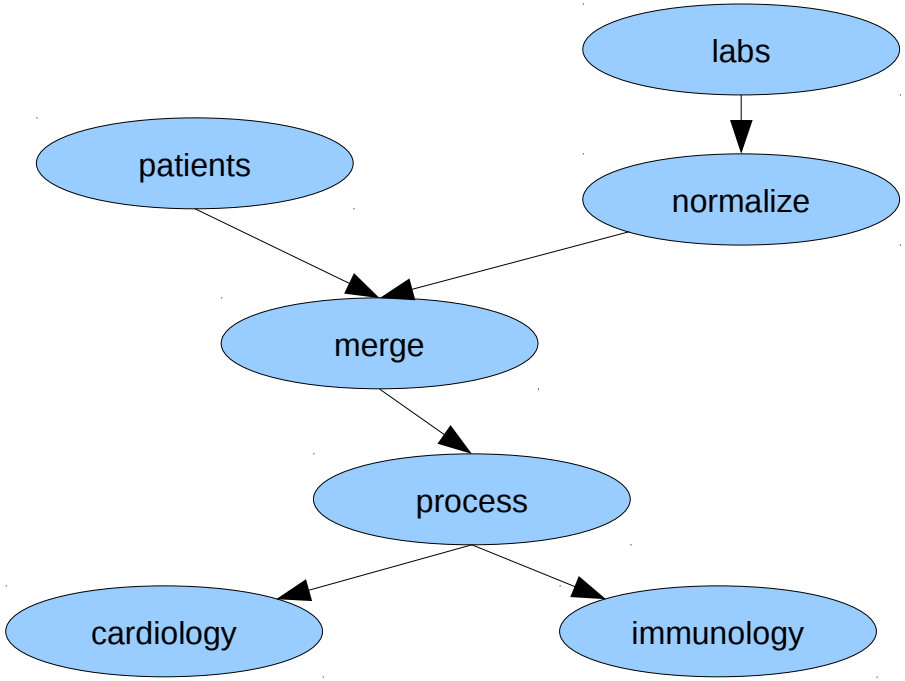
Physical pipeline communication: efficiency



- **Solution: Framework uses native protocols between local objects**
- **Wrappers automatically:**
  - **Use native protocols <u>within</u> an environment**
  - **Use HTTP <u>between</u> environments**
- **Updater stays simple: always thinks it's talking locally**

DEMO

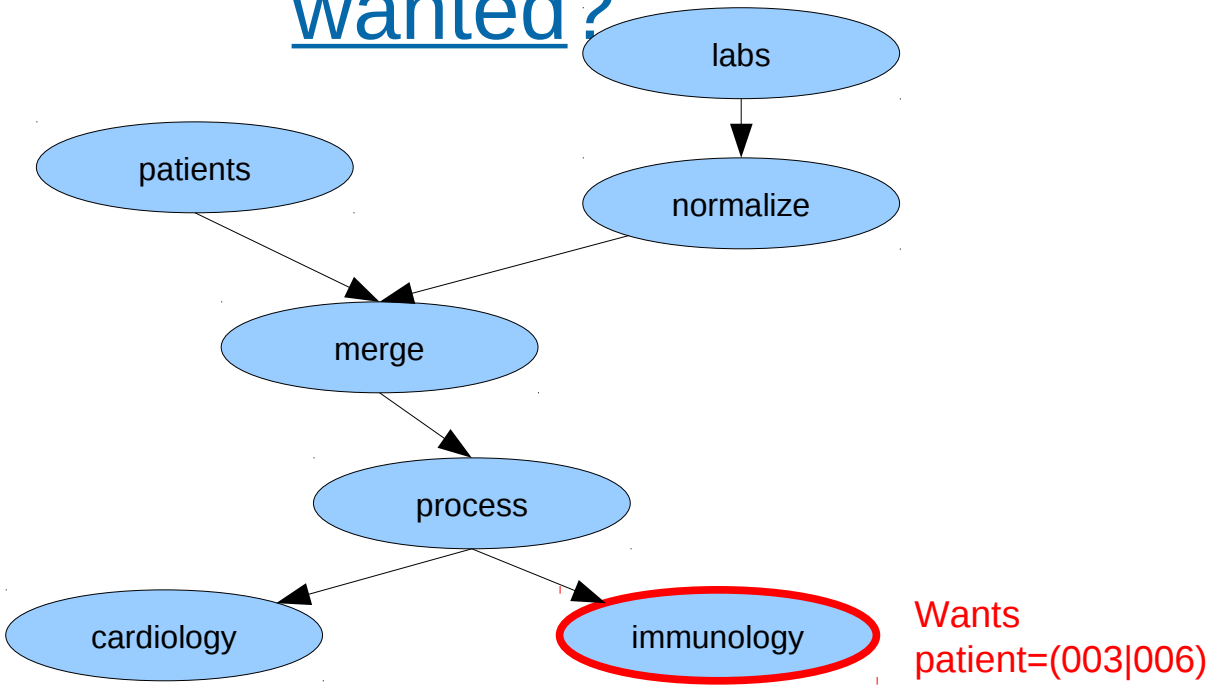
Demo example



Demo URLs

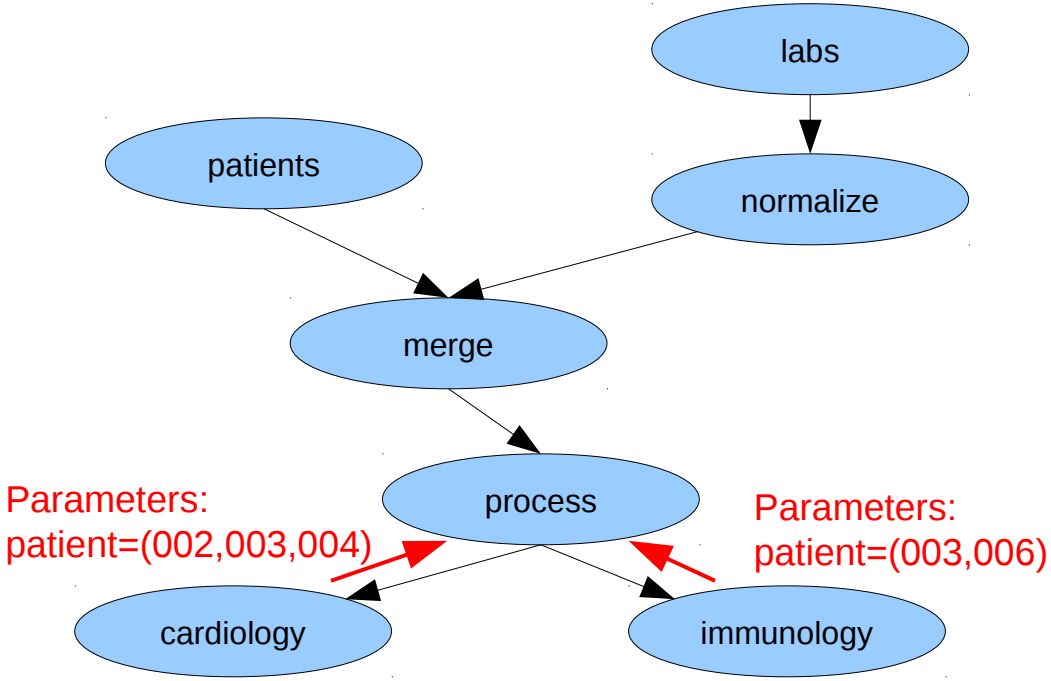
- **http://localhost/node/patients**
- **http://localhost/node/labs**
- **http://localhost/node/normalize**
- **http://localhost/node/merge**
- **http://localhost/node/cardiology**
- **http://localhost/node/immunology**
-

Problem 11: How to indicate what data is wanted?



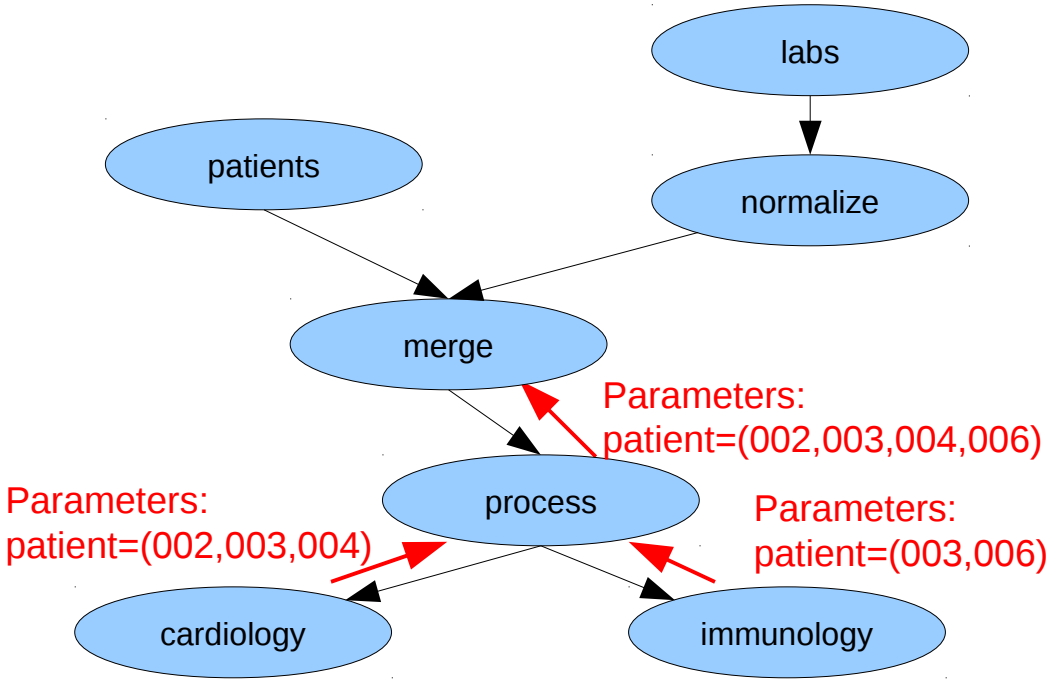
- **immunology only needs a subset of process**
- **Wasteful to generate all possible records**
- **How can immunology tell process which records it wants?**

Solution: Propagate parameters upstream

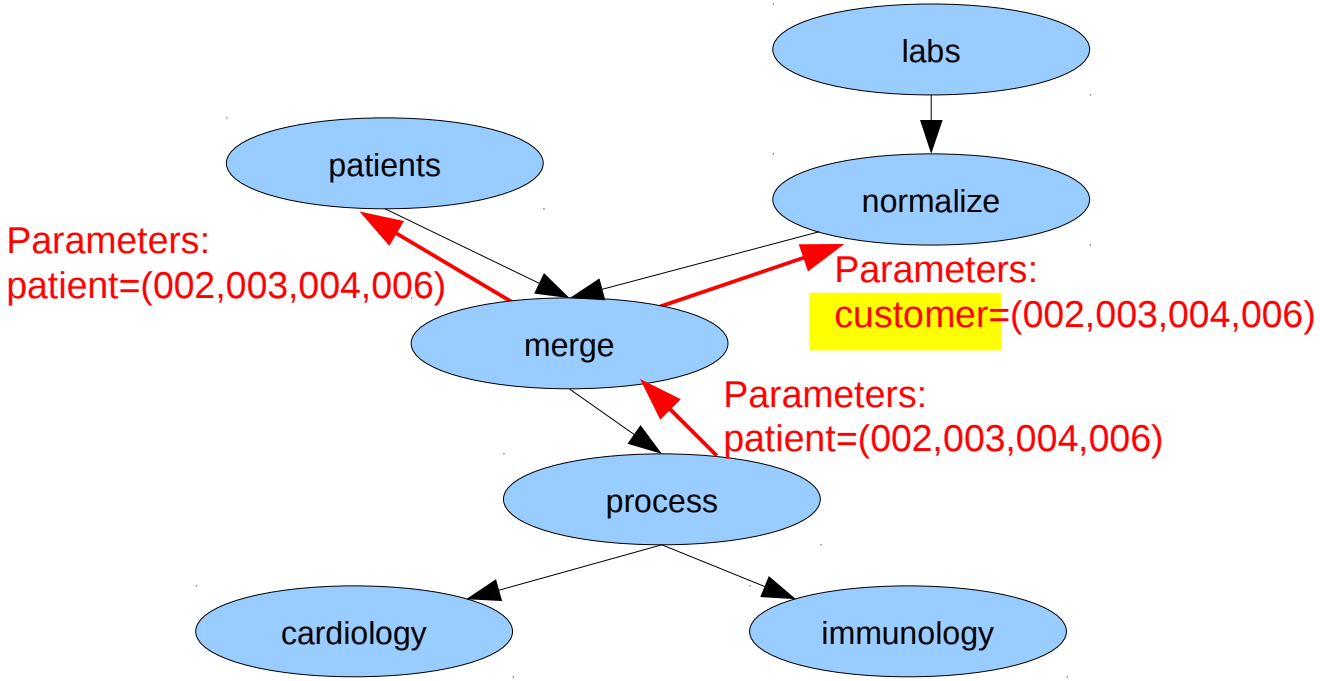


- **Patient ID parameters are passed <u>upstream</u>**

Merging parameters



Passing different parameters to different inputs



- **Different inputs may need different parameters**
- **Node's p:parametersFilter can transform parameters**

DEMO

Demo URLs

- **http://localhost/node/patients**
- **http://localhost/node/labs**
- **http://localhost/node/normalize**
- **http://localhost/node/merge**
- **http://localhost/node/cardiology**
- **http://localhost/node/immunology**
- **http://localhost/node/cardiology?id=(002,003,004)**
- **http://localhost/node/immunology?id=(003,006)**
-

Summary

- **Efficient**
  - **Updates only what needs to be updated**
  - **Caches automatically**
  - **Communicates with native protocols when possible, RESTful HTTP otherwise**
- **Flexible:**
  - **Any kind of data – not only RDF**
  - **Any kind of custom code (using wrappers)**
- **Easy:**
  - **Easy to implement nodes (using standard wrappers)**
  - **Easy to define pipelines (using a few lines of RDF)**
  - **Easy to visualize**
  - **Easy to maintain – very loosely coupled**

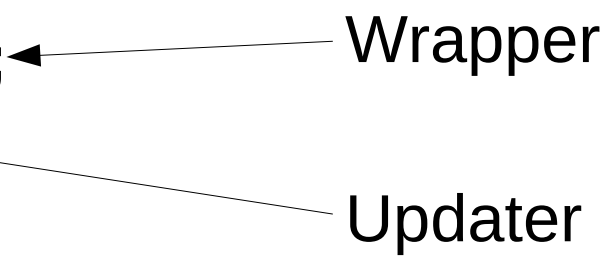
Questions?

BACKUP SLIDES

Wrappers and updaters (in Turtle)

With implicit updater:

```
1.  . . .
2. :normalize a p:FileNode ;
3.     p:inputs ( :labs ) .
4.  . . .
```



Wrappers and updaters (in Turtle)

With implicit updater:

```
1.  . . .
2. :normalize a p:FileNode ;
3.     p:inputs ( :labs ) .
4.  . . .
```

Wrapper

Updater

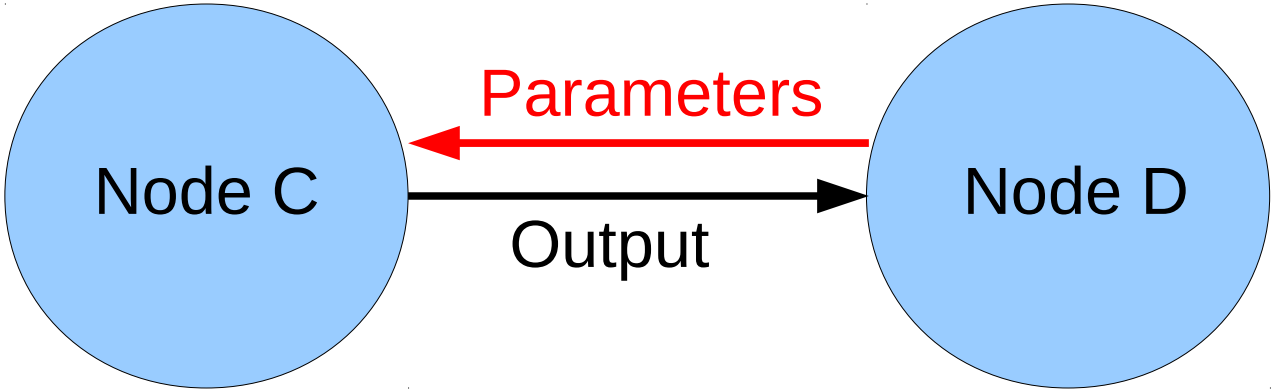
With explicit updater:

```
5.  . . .
6. :normalize a p:FileNode ;
7.     p:inputs ( :labs ) ;
8.     p:updater "normalize-updater" .
9.  . . .
```

Wrapper

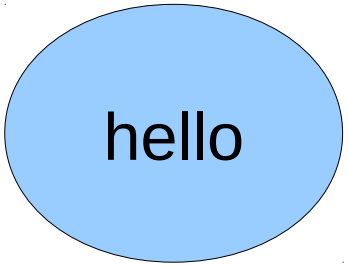
Updater

Terminology: output versus parameters



- **Output flows <u>downstream</u>**
- **Parameters flow <u>upstream</u>**

Example one-node pipeline definition: “hello world”




```
1. @prefix p: <http://purl.org/pipeline/ont#> .
2. @prefix : <http://localhost/> .
3. :hello a Node ;
4.      p:updater "hello-updater" .
```

Output can be retrieved from http://localhost/hello

Implementation of “hello world” Node

Code in hello-updater:

```
1.   #!/bin/bash -p
2.   echo Hello from $1 on `date`
```

- **hello-updater is then placed where the wrapper can find it**
  - **E.g., Apache WWW directory**

Invoking the “hello world” Node

When URL is accessed:

```
http://localhost/hello
```

If $cacheFile is stale, wrapper invokes the updater as:

```
hello-updater > $cacheFile
```

Wrapper serves $cacheFile content:

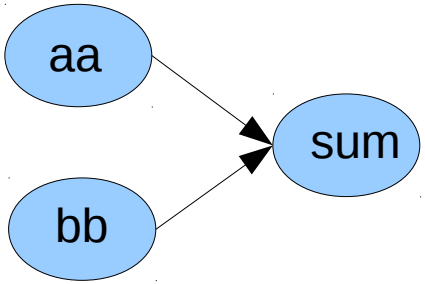
```
Hello on Wed Apr 13 14:54:57 EDT 2011
```

What do I mean by “cache”?

- ~~**Meaning 1: A local copy of some other data store**~~
  - ~~**I.e., the same data is stored in both places**~~

- **Meaning 2: Stored data that is regenerated when stale**
  - **Think: caching the results of a CGI program**
  - **Results can be served from the cache if inputs have not changed**

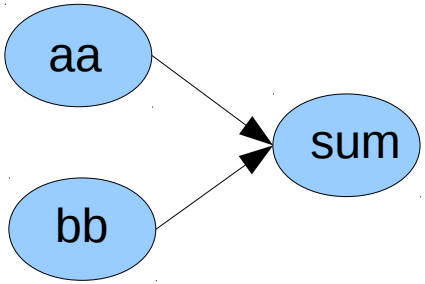
Example pipeline: sum two numbers



Pipeline definition:

```
1.@prefix p: <http://purl.org/pipeline/ont.n3#> .
2.@prefix : <http://localhost/> .
3.:aa a p:Node .
4.:bb a p:Node .
5.:sum a p:Node ;
6.        p:inputs ( :aa :bb ) ;
7.        p:updater "sum-updater" .
```

sum-updater implementation



Node implementation (in Perl):

```
1.#! /usr/bin/perl -w
2.# Add numbers from two nodes.
3.my $sum = `cat $ARGV[1]` + `cat $ARGV[2]`;
4.print "$sum\n";
```

aa cache

bb cache

Why SPARQL?

- **Standard RDF query language**
- **Can help bridge RDF <--> relational data**
  - **Relational --> RDF: mappers are available**
    http://www.w3.org/wiki/Rdb2RdfXG/StateOfTheArt
  - **RDF --> relational: SELECT returns a table**
- ***Also* can act as a rules language**
  - **CONSTRUCT or INSERT**

SPARQL CONSTRUCT as an inference rule

- **CONSTRUCT creates (and returns) new triples if a condition is met**
  - **That's what an inference rule does!**
- **CONSTRUCT is the basis for SPIN (Sparql Inference Notation), from TopQuadrant**
- **However, in standard SPARQL, CONSTRUCT only *returns* triples (to the client)**
  - **Returned triples must be inserted back into the server – an extra client/server round trip**

SPARQL INSERT as an inference rule

- **INSERT creates and asserts new triples if a condition is met**
  - **That's what an inference rule does!**
- **Single operation – no need for extra client/server round trip**

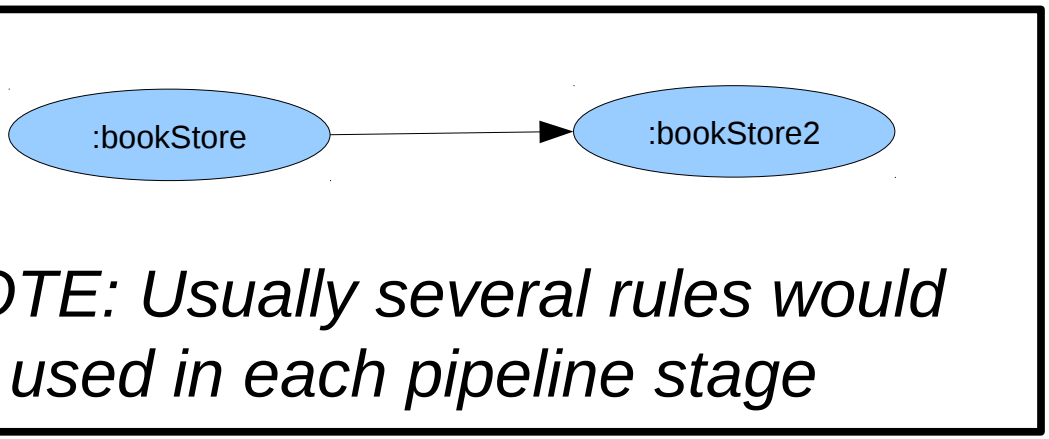
- **Issue: How to apply inference rules repeatedly until no new facts are asserted?**
  - **E.g. transitive closure**
  - **cwm --think option**
  - **SPIN**
- **In standard SPARQL, requested operation is only performed once**
- ***Would be nice to have a SPARQL option to REPEAT until no new triples are asserted***

SPARQL bookStore2 INSERT example

```
1. # Example from W3C SPARQL Update 1.1 specification
2. #
3. PREFIX dc:  <http://purl.org/dc/elements/1.1/>
4. PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
5.
6. INSERT
7.   { GRAPH <http://example/bookStore2> { ?book ?p ?v } }
8. WHERE
9.   { GRAPH  <http://example/bookStore1>
10.      { ?book dc:date ?date .
11.        FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
12.        ?book ?p ?v
13.  } }
```

BookStore2 INSERT rule as pipeline

```
1. # Exa
2. #
3. PREF
4. PREF
5.
6. INSERT
7.   { GRAPH <http://example/bookStore2> { ?book ?p ?v } }
8. WHERE
9.   { GRAPH  <http://example/bookStore1>
10.      { ?book dc:date ?date .
11.        FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
12.        ?book ?p ?v
13.  } }
```



BookStore2 pipeline definition

```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/> .
3.  :bookStore1 a p:JenaNode .
4.  :bookStore2 a p:JenaNode ;
5.      p:inputs ( :bookStore1 ) ;
6.      p:updater “bookStore2-updater.sparql” .
```

SPARQL INSERT as a reusable rule: bookStore2-updater.sparql

```
1. # $output will be the named graph for the rule's results
2. # $input1 will be the input named graph
3. PREFIX dc:  <http://purl.org/dc/elements/1.1/>
4. PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
5.
6. INSERT
7.   { GRAPH <http://example/bookStore2> { ?book ?p ?v } }
8. WHERE
9.   { GRAPH <http://example/bookStore1>
10.      { ?book dc:date ?date .
11.        FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
12.        ?book ?p ?v
13.  } }
```

SPARQL INSERT as a reusable rule: bookStore2-updater.sparql

```
1. # $output will be the named graph for the rule's results
2. # $input1 will be the input named graph
3. PREFIX dc:  <http://purl.org/dc/elements/1.1/>
4. PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
5.
6. INSERT
7.   { GRAPH          $output          { ?book ?p ?v } }
8. WHERE
9.   { GRAPH          $input1
10.      { ?book dc:date ?date .
11.        FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
12.        ?book ?p ?v
13.  } }
```

Issue: Need for virtual graphs

- **How to query against a large collection of graphs?**
- **Some graph stores query the merge of all named graphs by default**
  - **Virtual graph or “view”**
  - **sd:UnionDefaultGraph feature**
- ***BUT* it only applies to the default graph of the entire graph store**

- ***Conclusion: Graph stores should support multiple virtual graphs***
  - ***Some do, but not standardized***

Rough sketch of pipeline ontology: ont.n3 (1)

```
1.@prefix p: <http://purl.org/pipeline/ont#> .
2.@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3.
4.######### Example Node types ##########
5.p:Node a rdfs:Class .
6.p:CommandNode  rdfs:subClassOf p:Node .  # Default Node type
7.p:JenaNode      rdfs:subClassOf p:Node .
8.p:SesameNode    rdfs:subClassOf p:Node .
9.p:PerlNode      rdfs:subClassOf p:Node .
10.p:MysqlNode rdfs:subClassOf p:Node .
11.p:OracleNode  rdfs:subClassOf p:Node .
```

Rough sketch of pipeline ontology: ont.n3 (2)

```
12.######### Node properties ##########
13.p:inputs    rdfs:domain p:Node .
14.p:parameters   rdfs:domain p:Node .
15.p:dependsOn    rdfs:domain p:Node .
16.
17.# p:state specifies the output cache for a node.
18.# It is node-type-specific, e.g., filename for FileNode .
19.# It may be set explicitly, otherwise a default will be used.
20.p:state  rdfs:domain p:Node .
21.
22.# p:updater specifies the updater method for a Node.
23.# It is node-type-specific, e.g., a script for CommandNode .
24.p:updater   rdfs:domain p:Node .
25.
26.# p:updaterType specifies the type of updater used.
27.# It is node-type-specific.
28.p:updaterType rdfs:domain p:Node .
```

Rough sketch of pipeline ontology: ont.n3 (3)

```
29.######### Rules ##########
13.# A Node dependsOn its inputs and parameters:
14.{ ?a p:inputs ?b . }     => { ?a p:dependsOn ?b . } .
15.{ ?a p:parameters ?b . } => { ?a p:dependsOn ?b . } .
```

Nodes

- **Each node has:**
  - **A URI (to identify it)**
  - **One output "state"**
  - **An update method ("updater") for refreshing its output cache**
- **A node may also have:**
  - **Inputs (from upstream)**
  - **Parameters (from downstream)**

(Demo 0: Hello world)

Example GraphNode pipeline (one node)

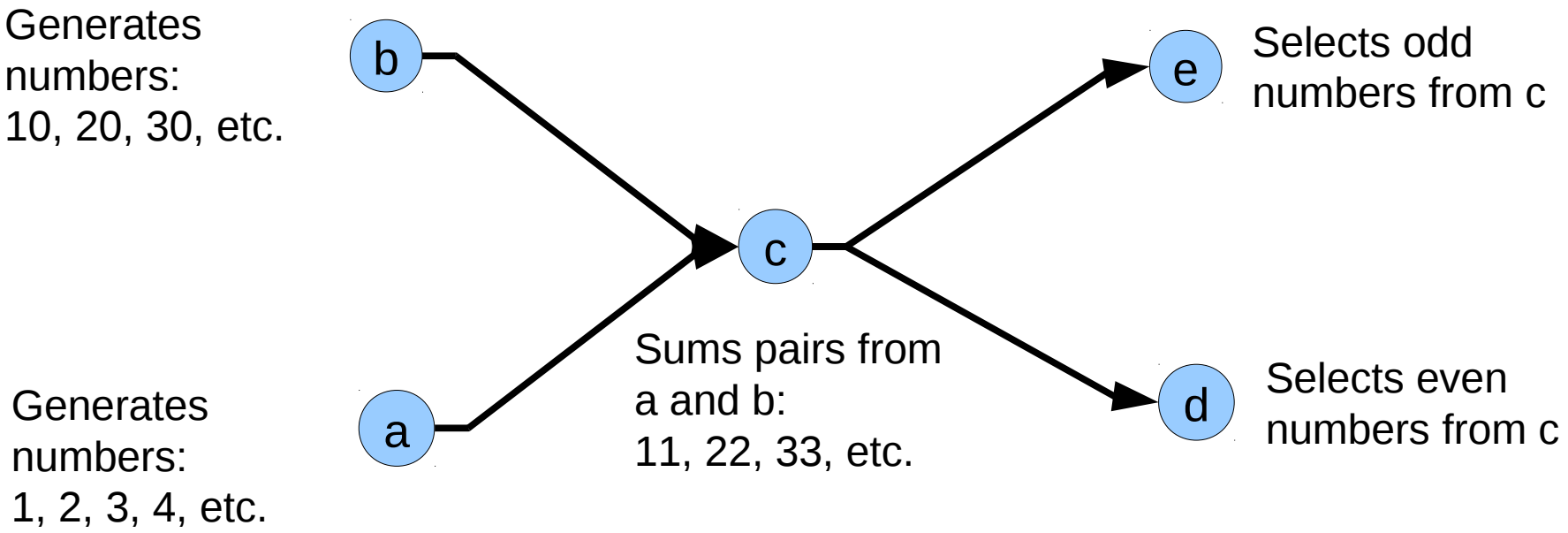
```
@prefix p: <http://purl.org/pipeline/ont#> .
@prefix : <http://localhost/> .
:e a :GraphNode ;
   p:updater “e-updater.sparql” .
```

File example-construct.txt

```
# Example from SPARQL 1.1 spec
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX vcard:   <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT { ?x  vcard:N _:v .
            _:v vcard:givenName ?gname .
            _:v vcard:familyName ?fname }
WHERE
 {
   { ?x foaf:firstname ?gname } UNION  { ?x foaf:givenname   ?gname } .
   { ?x foaf:surname   ?fname } UNION  { ?x foaf:family_name ?fname } .
 }
```

(Demo: Sparql INSERT)

Example 1: Multiple nodes



- **Node c consumes records from a & b**
- **Nodes d & e consume records from c**

Data in node a

<s01> <a1> 111 .

<s01> <a2> 121 .

<s01> <a3> 131 .

<s02> <a1> 112 .

<s02> <a2> 122 .

<s02> <a3> 132 .

<s03> <a1> 113 .

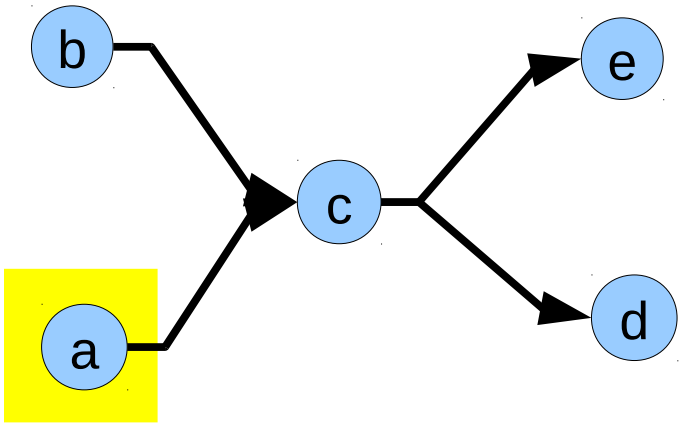
<s03> <a2> 123 .

<s03> <a3> 133 .

<s04> <a1> 114 .

. . .

<s09> <a3> 139 .



Data in node b

<s01> <b1> 211 .

<s01> <b2> 221 .

<s01> <b3> 231 .

<s02> <b1> 212 .

<s02> <b2> 222 .

<s02> <b3> 232 .

<s03> <b1> 213 .

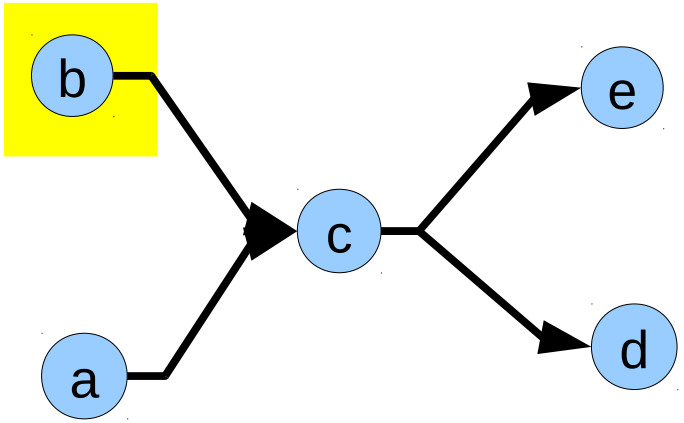
<s03> <b2> 223 .

<s03> <b3> 233 .

<s04> <b1> 214 .

. . .

<s09> <b3> 239 .



Data in node c

<s01> <a1> 111 .
<s01> <a2> 121 .
<s01> <a3> 131 .
<s01> <b1> 211 .
<s01> <b2> 221 .
<s01> <b3> 231 .

Merged triples

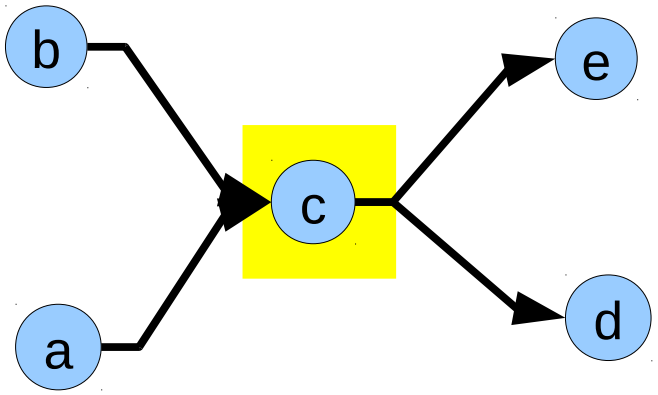
<s01> <c1> 111211 .
<s01> <c2> 121221 .
<s01> <c3> 131231 .

Inferred triples

<s02> <a1> 112 .

. . .

<s09> <c3> 139239 .



Data in nodes d&e: same as c

<s01> <a1> 111 .

<s01> <a2> 121 .

<s01> <a3> 131 .

<s01> <b1> 211 .

<s01> <b2> 221 .

<s01> <b3> 231 .

<s01> <c1> 111211 .

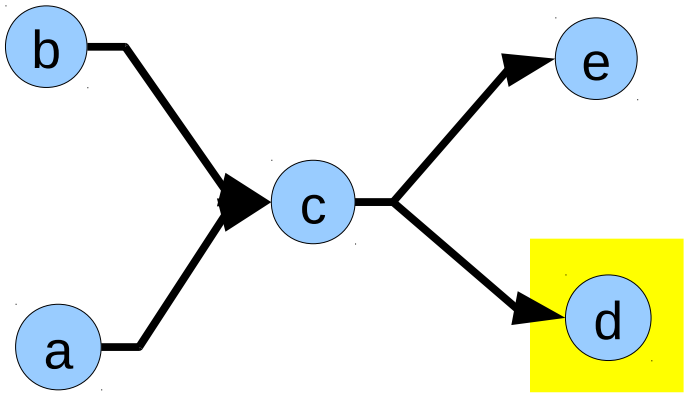
<s01> <c2> 121221 .

<s01> <c3> 131231 .

<s02> <a1> 112 .

. . .

<s09> <c3> 139239 .



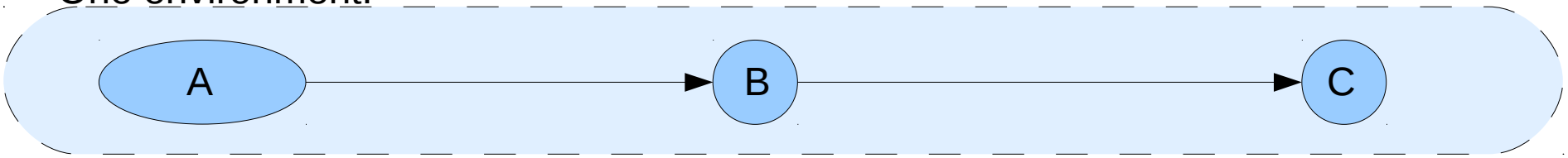
Example 2: Multiple node pipeline in N3

```
# Example 1: Multiple nodes

@prefix p: <http://purl.org/pipeline/ont#> .
@prefix : <http://localhost/> .

:a a p:Node .
:a    p:updater "a-updater" .

:b a p:Node .
:b    p:updater "b-updater" .

:c a p:Node .
:c    p:inputs ( :a :b ) .
:c    p:updater "c-updater" .

:d a p:Node .
:d    p:inputs ( :c ) .
:d    p:updater "d-updater" .

:e a p:Node .
:e    p:inputs ( :c ) .
:e    p:updater "e-updater" .
```

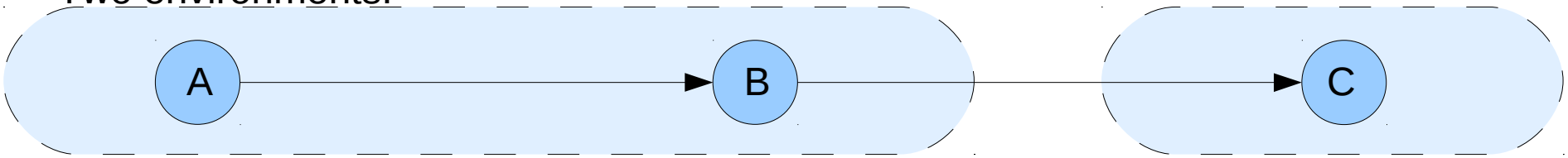
(Demo 1: Multiple node pipeline)

Pipelines and environments

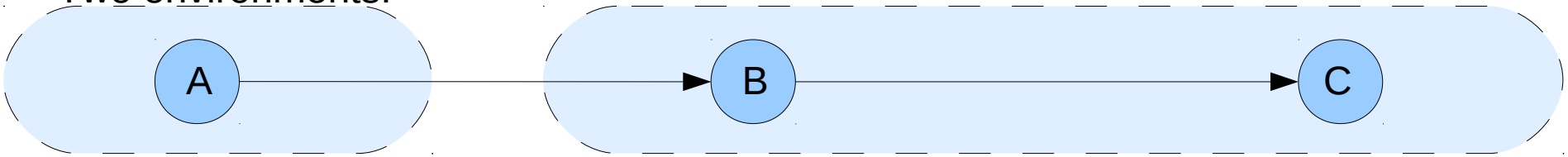
One environment:



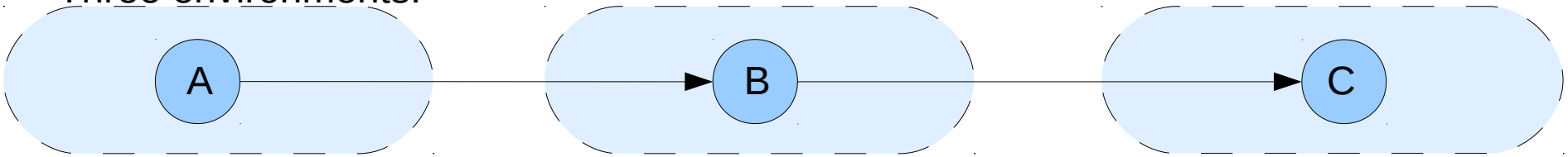
Two environments:



Two environments:

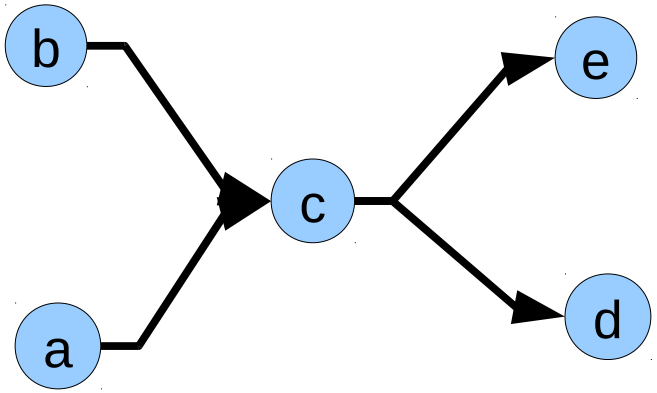


Three environments:

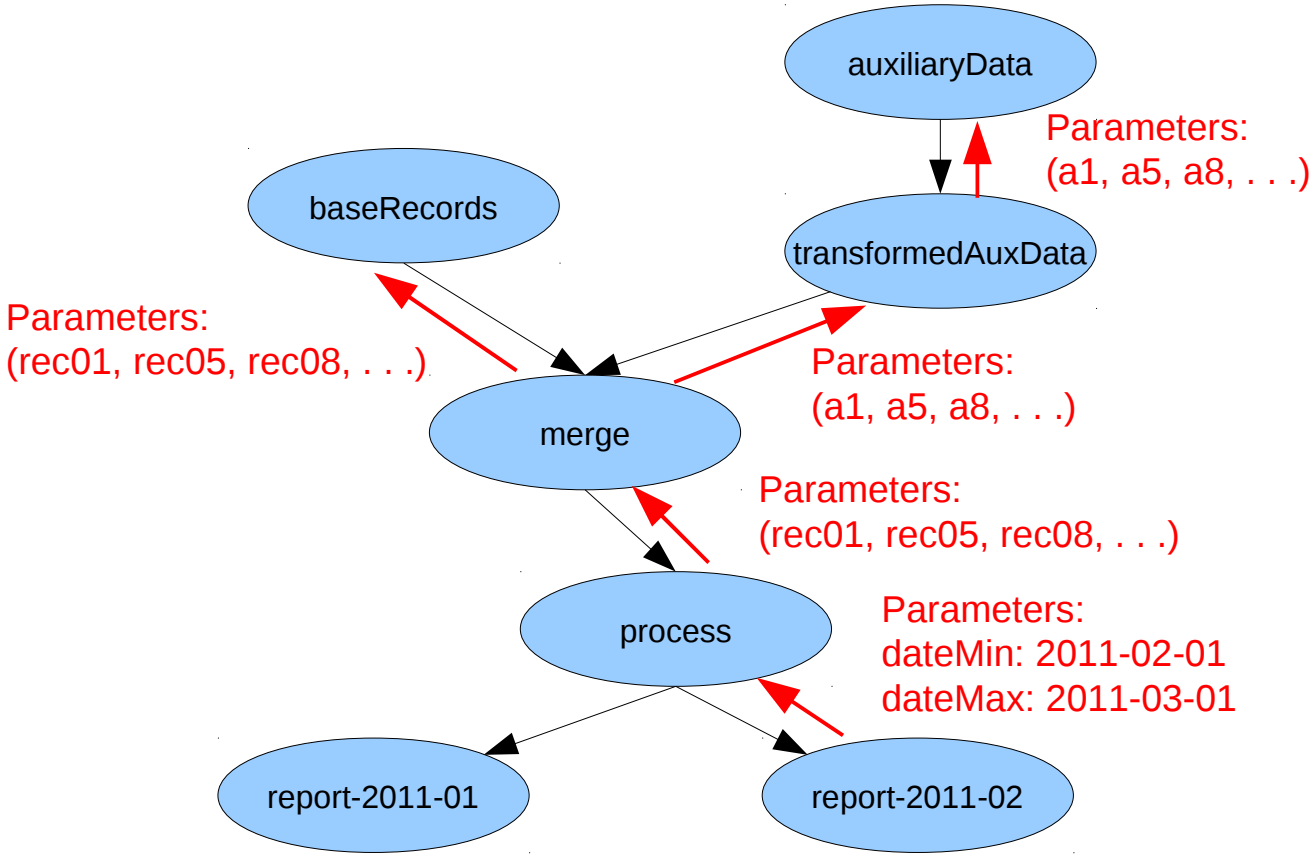


- **Reminder: Environment means server and node type**

Small diagram

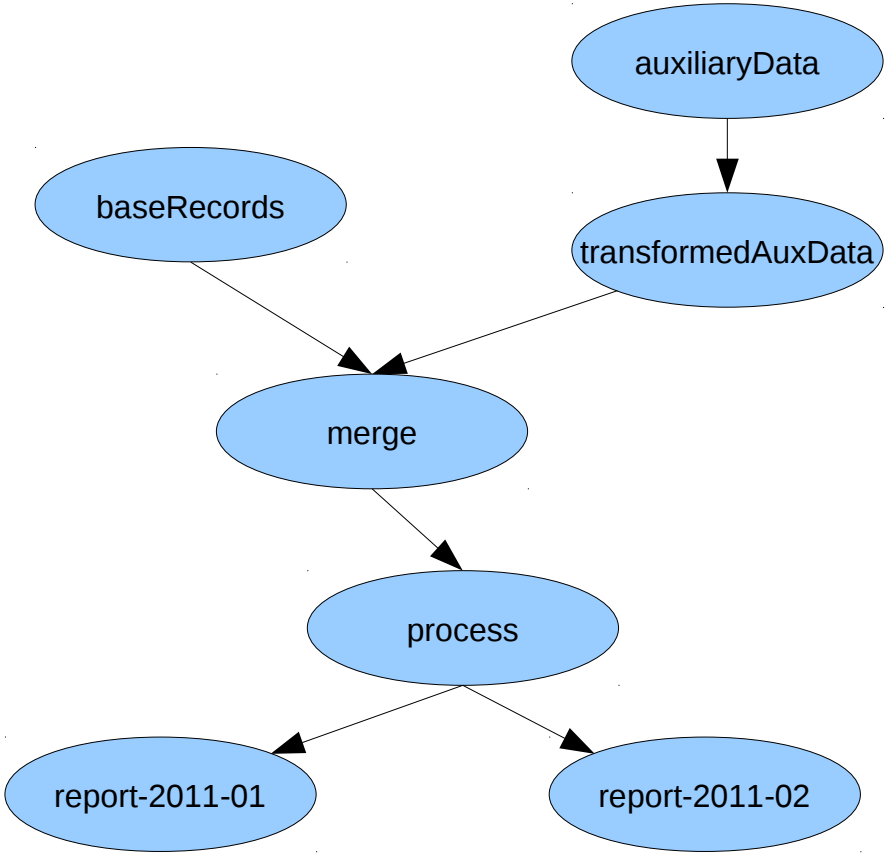


Example: Monthly report



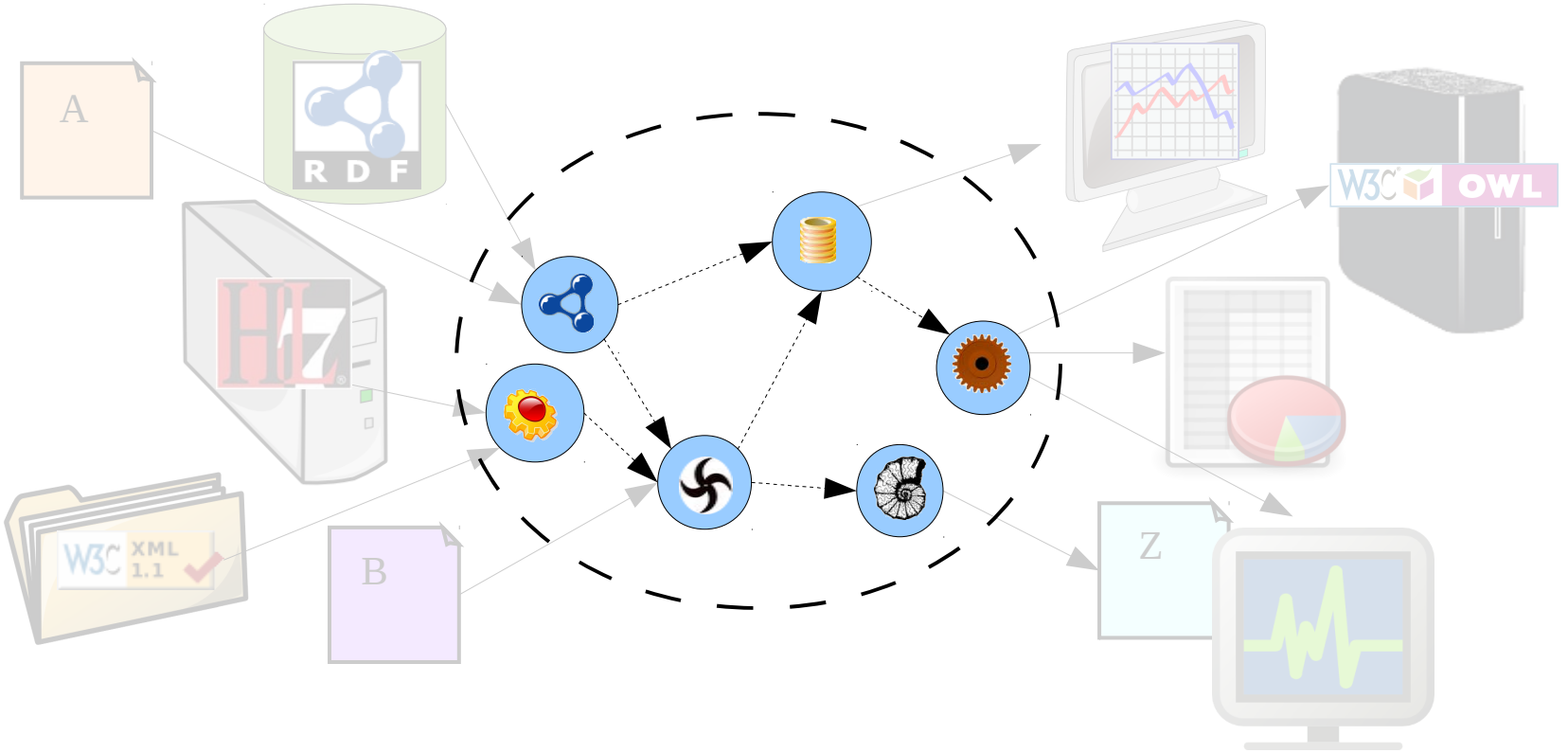
- **Downstream reports should auto update when baseRecords change**

Staleness



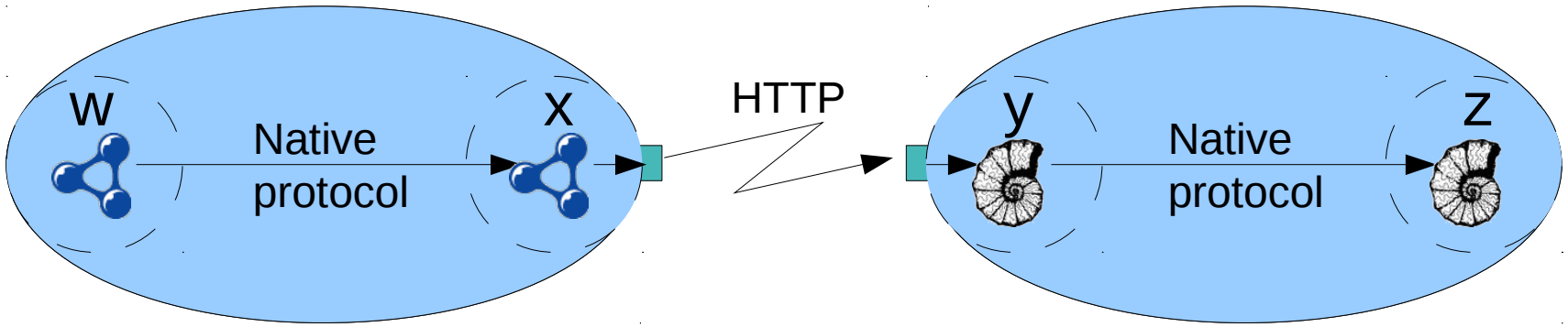
- **A node's state cache becomes stale if an input node changes**
  - **The node's update method must be invoked to refresh it**
- **E.g., when baseRecords is updated, merge becomes stale**

Option 3: RDF data pipeline framework



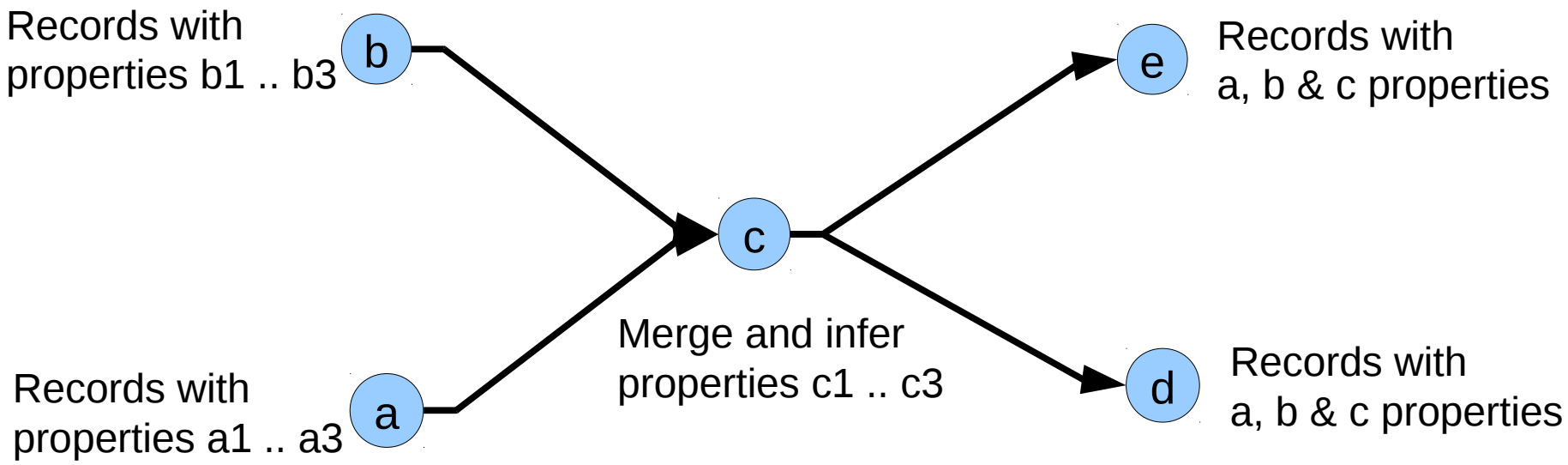
- **Uniform, distributed, data pipeline framework**
- **Custom code is hidden in standard wrappers**
- **Pros: Easy to build and maintain; Can leverage existing integration tools; Low risk - Can grow organically**
- **Cons: Can grow organically – No silver bullet**

Physical view - Optimized



- **But nodes that share an implementation environment communicate directly, using native protocol, e.g.:**
  - **One NamedGraphNode to another in the same RDF store**
  - **One TableNode to another in the same relational database**
  - **One Node to another on the same server**
- **Wrappers handle both native protocol and HTTP**

Example 1: Multiple nodes



- **Five nodes: a, b, c, d, e**
- **Node c merges and augments records from a & b**
- **Nodes d & e consume augmented records from c**

Data in node a

<s01> <a1> 111 .

<s01> <a2> 121 .

<s01> <a3> 131 .

<s02> <a1> 112 .

<s02> <a2> 122 .

<s02> <a3> 132 .

<s03> <a1> 113 .

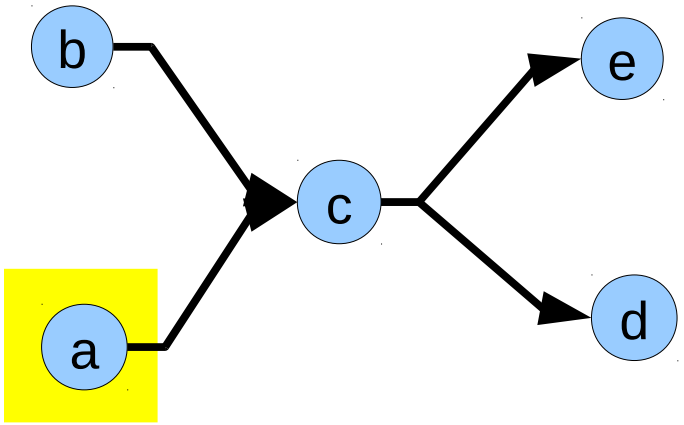
<s03> <a2> 123 .

<s03> <a3> 133 .

<s04> <a1> 114 .

. . .

<s09> <a3> 139 .



Data in node b

<s01> <b1> 211 .

<s01> <b2> 221 .

<s01> <b3> 231 .

<s02> <b1> 212 .

<s02> <b2> 222 .

<s02> <b3> 232 .

<s03> <b1> 213 .

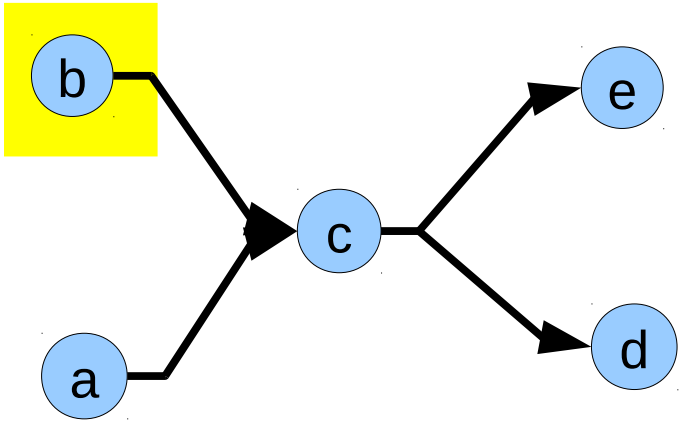
<s03> <b2> 223 .

<s03> <b3> 233 .

<s04> <b1> 214 .

. . .

<s09> <b3> 239 .



Data in node c

<s01> <a1> 111 .
<s01> <a2> 121 .
<s01> <a3> 131 .
<s01> <b1> 211 .
<s01> <b2> 221 .
<s01> <b3> 231 .

Merged triples

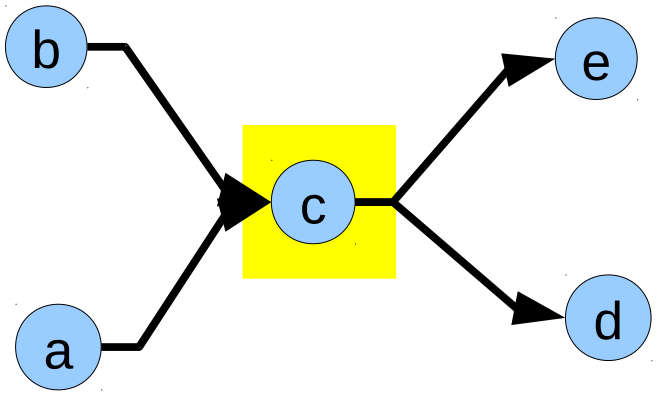
<s01> <c1> 111211 .
<s01> <c2> 121221 .
<s01> <c3> 131231 .

Inferred triples

<s02> <a1> 112 .

. . .

<s09> <c3> 139239 .



Data in nodes d&e: same as c

<s01> <a1> 111 .

<s01> <a2> 121 .

<s01> <a3> 131 .

<s01> <b1> 211 .

<s01> <b2> 221 .

<s01> <b3> 231 .

<s01> <c1> 111211 .

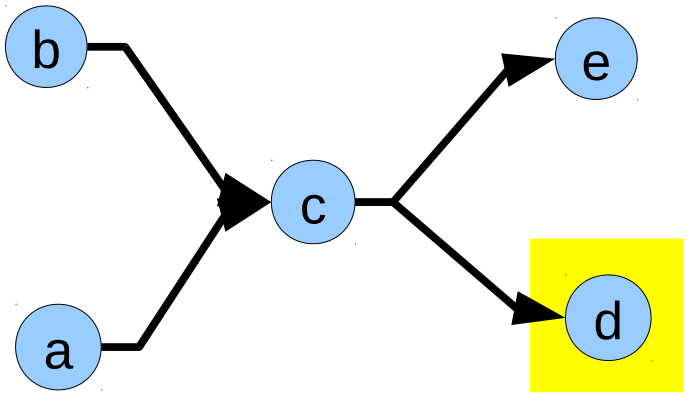
<s01> <c2> 121221 .

<s01> <c3> 131231 .

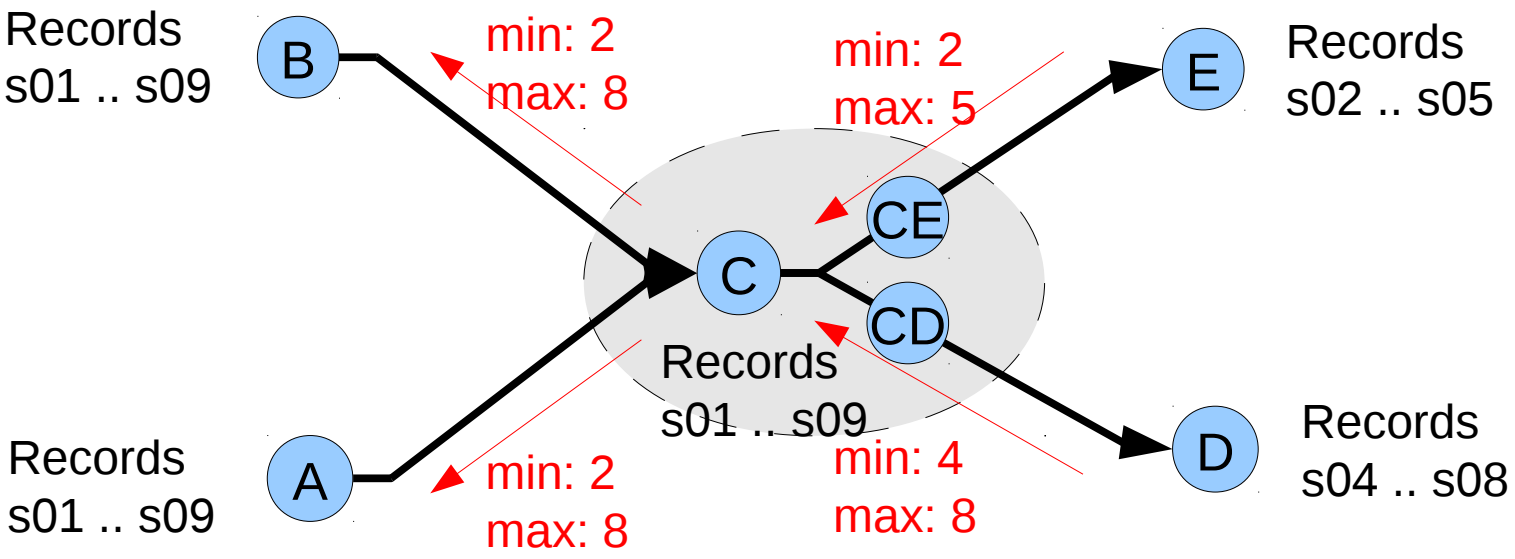
<s02> <a1> 112 .

. . .

<s09> <c3> 139239 .

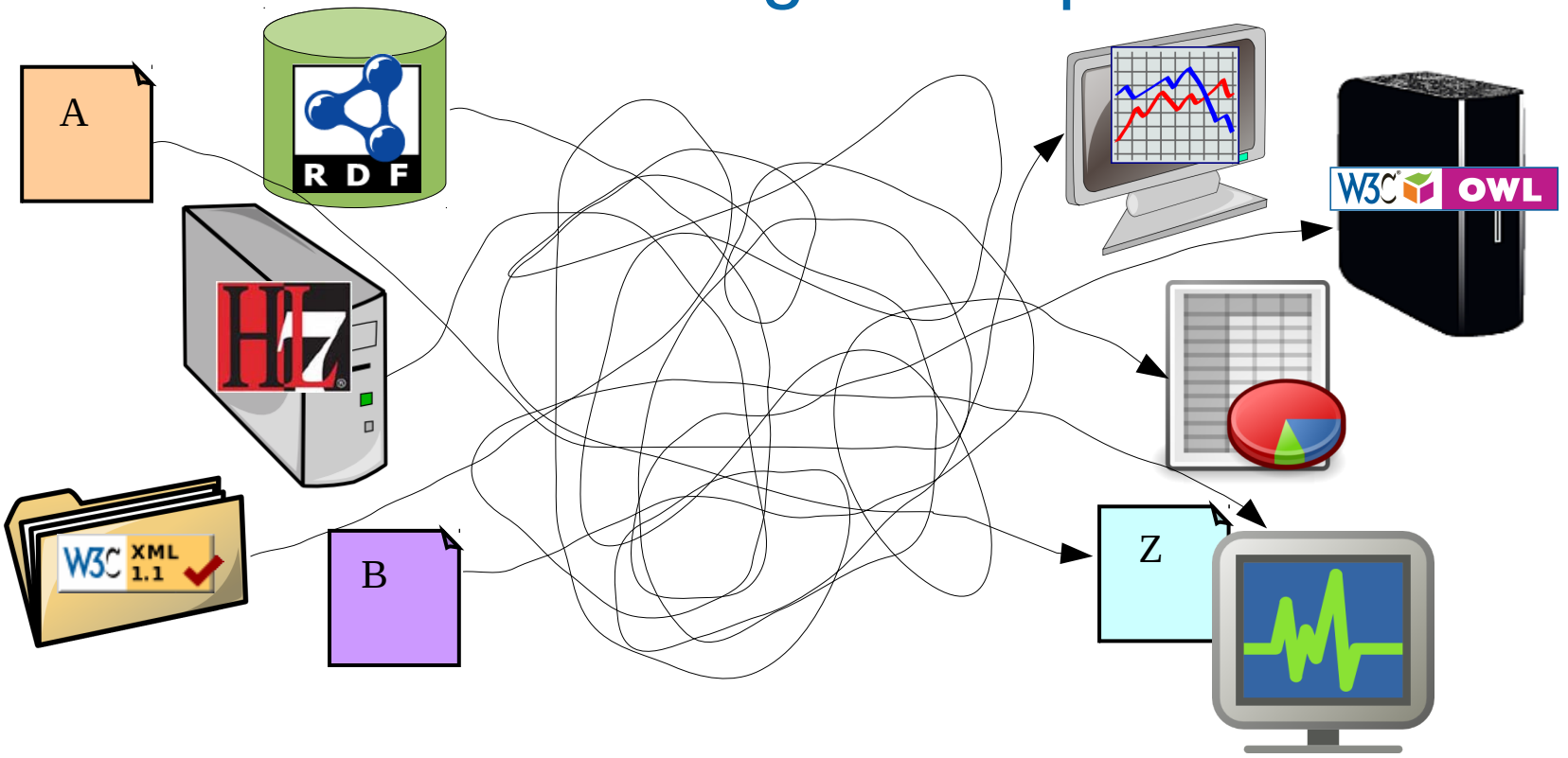


Example 2: Passing parameters upstream



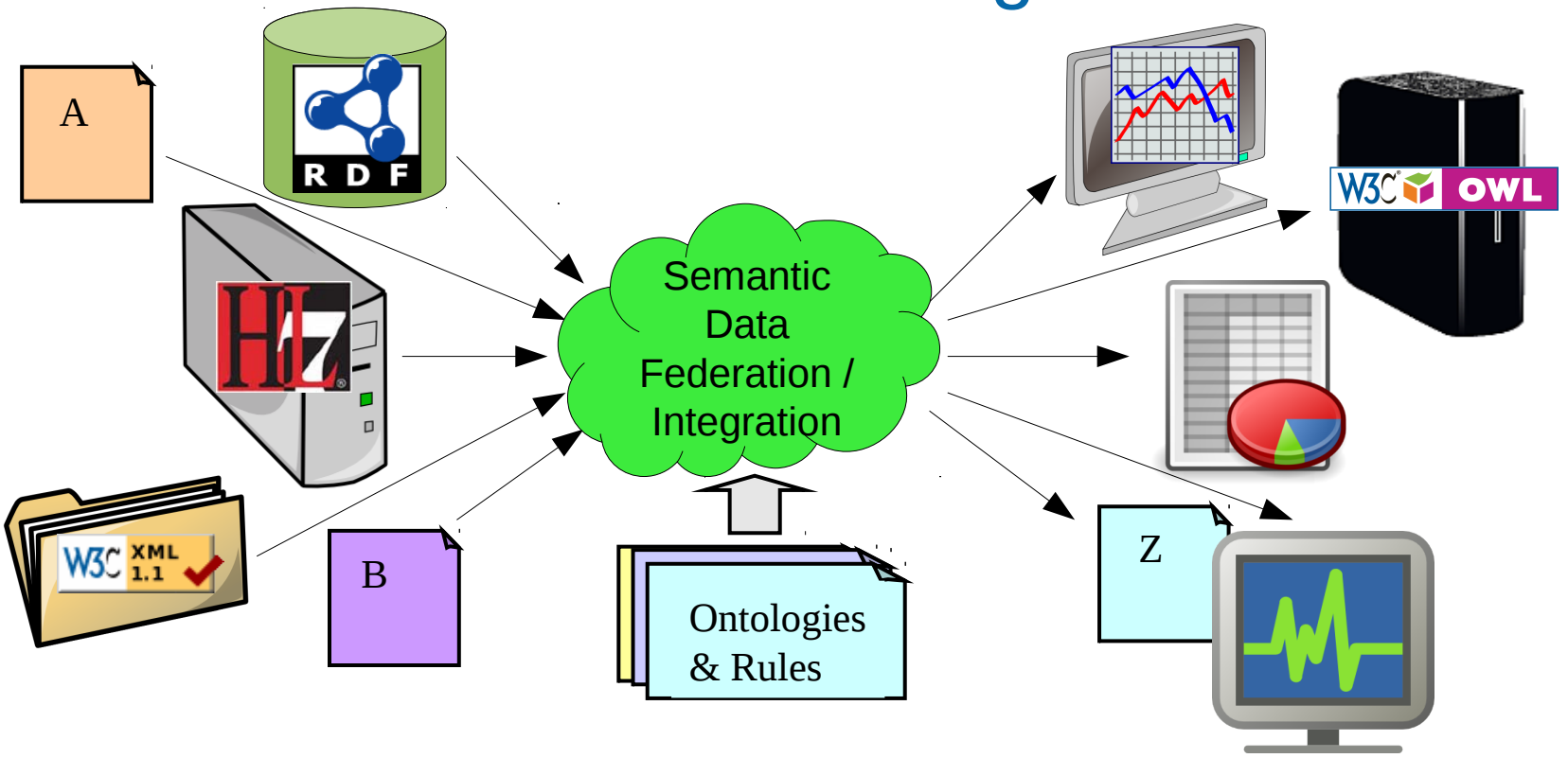
- **Node C may hold more records than D&E want**
- **Nodes D&E pass parameters upstream:**
  - **Min, max record numbers desired**
- **Node C supplies the union of what D&E requested**
- **Nodes D&E select the subsets they want: s04..s08 and s02..s05**
- **Node C, in turn, passes parameters to nodes A&B**

Data in a large enterprise



- **Many data sources and applications**
- **Each application wants the illusion of a single, integrated data source**

Semantic data integration



- **Many data sources and applications**
- **Many technologies and protocols**
- **Goal: Each application wants the illusion of a single, unified data source**
- **Strategy:**
  - **Use ontologies and rules for semantic transformations**
  - **Convert to/from RDF at the edges; Use RDF in the middle**

Example pipeline definition (in N3)

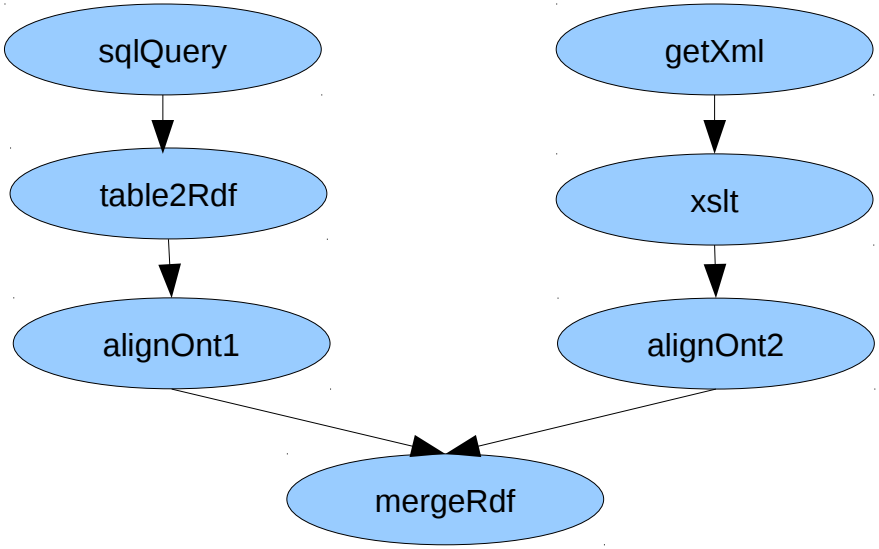
```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/> .
3.  :patients a p:Node .
4.  :labs a p:Node .
5.  :normalize a p:Node .
6.  :merge a p:Node .
7.          p:inputs ( :patients :normalize ) .
8.  :process a p:Node .
9.          p:inputs ( :merge ) .
10.  :report-2011-jan a p:Node .
11.          p:inputs ( :process ) .
12.  :report-2011-feb a p:Node .
13.          p:inputs ( :process ) .
```

Example pipeline definition (in N3)

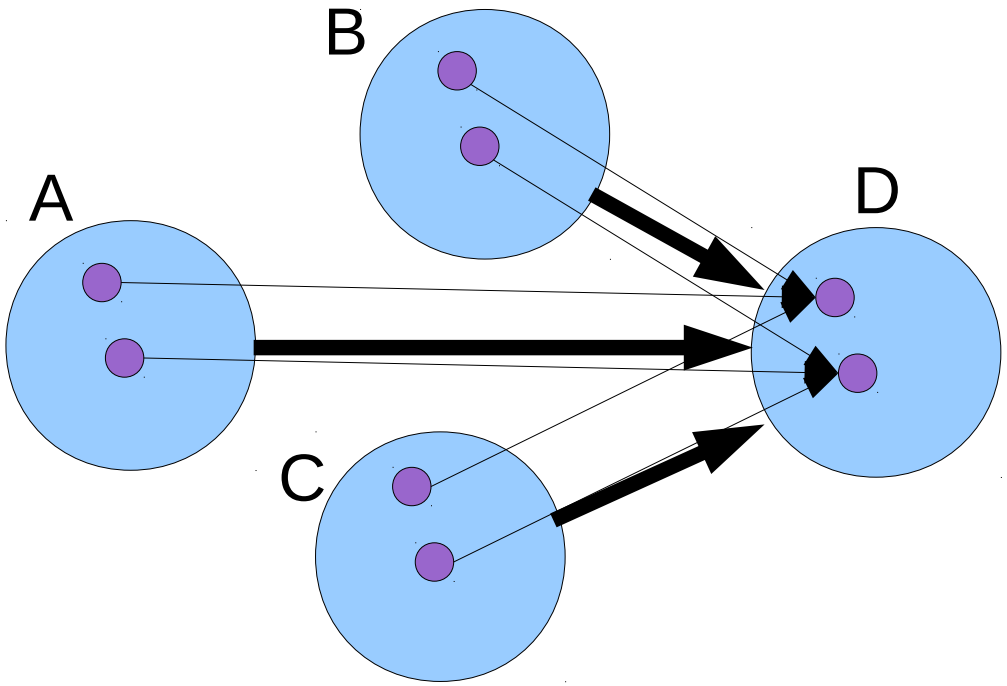
```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/> .
3.  :patients a p:Node .
4.  :labs a p:Node .
5.  :normalize a p:Node .
6.  :merge a p:JenaNode .
7.          p:inputs ( :patients :normalize ) .
8.  :process a p:JenaNode .
9.          p:inputs ( :merge ) .
10.  :report-2011-jan a p:Node .
11.          p:inputs ( :process ) .
12.  :report-2011-feb a p:Node .
13.          p:inputs ( :process ) .
```

Example

```
:sqlQuery a p:FileNode ;
  p:updater “sqlQuery-
  updater” .
:table2Rdf a p:FileNode ;
   p:inputs ( :sqlQuery ) ;
   p:updater “table2Rdf-
  updater” .
```



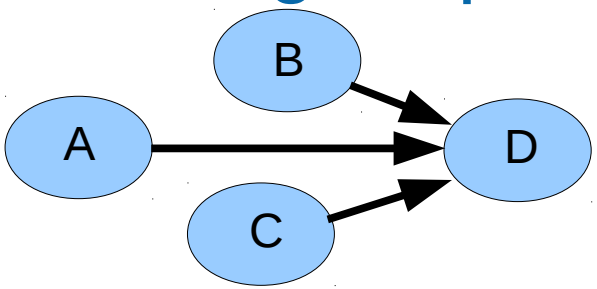
Map with multiple inputs



- **Map can also be used with multiple inputs**
- **D is updated by map(f, A, B, C):**

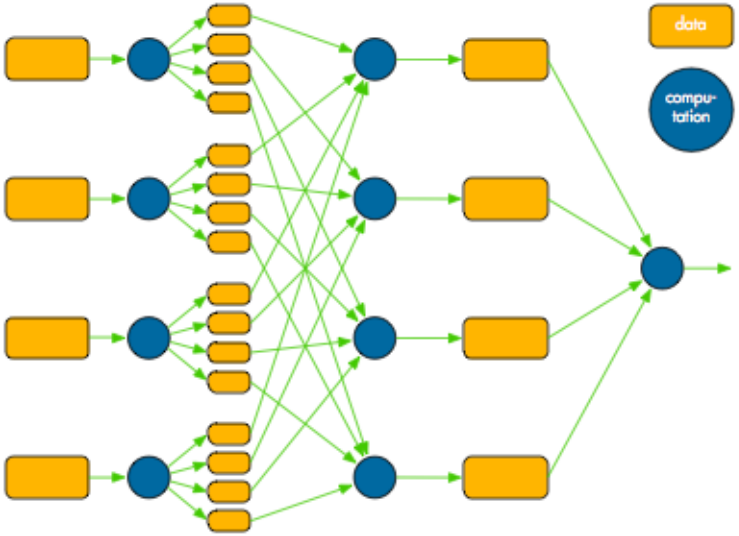
  For each i, di = f(ai, bi, ci)

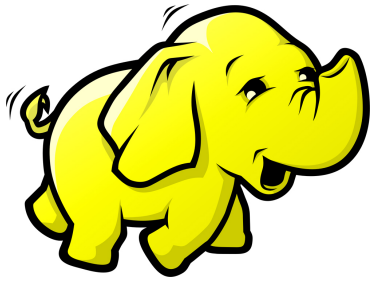
Pipeline definition using map with multiple inputs




```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/> .
3.  :A a p:SesameNode .
4.  :B a p:SesameNode .
5.  :C a p:SesameNode .
6.  :D a p:SesameNode ;
7.      p:inputs ( :A :B :C ) ;
8.      p:updater ( p:mapcar "D-updater.sparql" ) .
```

Comparison with Hadoop





- **Hadoop:**
  - **Available and mature**
  - **Many more features (e.g., fault tolerance)**
  - **For Java**
  - **Processing is much more tightly coupled to Hadoop**

Example pipeline definition (in Turtle)

```
1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/> .
3.  :patients a p:FileNode .
4.  :labs a p:FileNode .
5.  :normalize a p:FileNode ;
6.          p:inputs ( :labs ) .
7.  :merge a p:FileNode ;
8.          p:inputs ( :patients :normalize ) .
9.  :process a p:FileNode ;
10.         p:inputs ( :merge ) .
11.  :report-2011-jan a p:FileNode ;
12.         p:inputs ( :process ) .
13.  :report-2011-feb a p:FileNode ;
14.         p:inputs ( :process ) .
```