RDF Data Pipelines for Semantic Data Federation

David Booth, Ph.D. (Contractor)
Christopher Pierce, Ph.D. (Cleveland Clinic)

Semantic Technology Conference
San Francisco
8-Mar-2011

Please download the latest version of these slides:
http://dbooth.org/2011/pipeline/
Who am I?

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

• **Christopher Pierce, PhD:**
  – Manager of Informatics, Cleveland Clinic
  – Pioneered use of RDF for patient data
  – W3C case study:
    http://www.w3.org/2001/sw/sweo/public/UseCases/ClevelandClinic/
What is this about?

• **Vision for multi-stage data production pipelines**
  – Dependency networks of nodes that process/store data
  – Intended for semantic data federation or integration
• **Light weight, decentralized, very loosely coupled**
  – Point-to-point communication
• **Designed for RDF data, but data agnostic**
• **Based on:**
  – RDF pipeline descriptions
  – HTTP dependency graphs
  – SPARQL
• **Cache oriented**
• **Updates only what needs to be updated**
Related work

- **Sparql Motion, from Top Quadrant**
  - A “visual scripting language for semantic data processing”
  - 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

- **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
What this is not

• Not a universal data model approach
  – No automatic data model/format translation

• Not a centralized approach
  – No central server or controller
    • Each node acts independently
  – But all nodes share the same RDF pipeline definition

• Not a workflow language
  – No flow-of-control operators
  – Focus is on data production pipelines
Where did this come from?

• Ideas originated while at HP Software
• Motivated by the need to manage RDF data production in a scalable way
• Ideas further extended from Cleveland Clinic work
  – Large amounts of patient data, lab data, etc. to be integrated and transformed
Why?

• **Flexible:**
  - Any kind of data – not only RDF
  - Any kind of custom code (using wrappers)
  - Internal homogeneous pipelines
  - Distributed heterogeneous pipelines

• **Efficient**
  - Updates only what needs to be updated
  - Communicates with native protocols when possible, HTTP otherwise

• **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
Caveat

• This is an architectural approach – not a product
• *Interested in your feedback!*
Semantic data federation / 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
How?

- 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: Monthly report pipeline

- Pipeline of multiple data sources and data production stages
  - A directed graph of nodes
  - Each node is one stage: processing and/or data storage
How?

• Pipeline of multiple data sources and data production stages
  – A directed graph of nodes
  – Each node is one stage: processing and/or data storage
Ad hoc data pipeline

• 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.
Pros and cons of ad hoc data pipeline

- **Pros**: Low initial risk; Can be built incrementally from existing pieces
- **Cons**: High long term cost; Fragile; Difficult to understand & maintain
Vision: RDF data pipeline

- Pipeline defined in RDF
  - An HTTP dependency graph
- Uses a uniform interface: RESTful HTTP
- Uses wrappers to handle:
  - Inter-node communication
  - Node update invocation
Flexibility retained

- Still permits:
  - Any technology inside nodes: shell scripts, SPARQL, databases, web services, etc.
  - Any data format between nodes – RDF or other
Example pipeline definition (in N3)

1.  @prefix p: <http://purl.org/pipeline/ont#> .
2.  @prefix : <http://localhost/> .
4.  :labData a p:Node .
5.  :transformedLabData a p:Node ;
   p:inputs ( :labData ) .
6.  :augmentedRecords a p:Node ;
   p:inputs ( :patientRecords :transformedLabData ) .
7.  :processedRecords a p:Node ;
   p:inputs ( :augmentedRecords ) .
   p:inputs ( :processedRecords ) .
   p:inputs ( :processedRecords ) .
Node wrappers

- Nodes may be implemented in arbitrary ways
  - Command script, SPARQL rules, HTTP web service, Relational database, etc.
- Custom node logic (“updater”) is hidden in wrapper
  - Wrappers provided for common node types
- Wrappers handle:
  - Inter-node communication (HTTP and potentially other protocols)
  - Node invocation
Example node wrapper types

- CommandNode is the default Node type
Example one-node pipeline definition:

“hello world”

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

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

Wrapper invokes the updater as:

hello-updater http://localhost/hello > /.../hello-stdout.txt

Wrapper serves /.../hello-stdout.txt content:

Why RDF pipeline definition?

- Directed graphs are natural to RDF
- Permits inferencing
- Easy visualization . . .
Visualizing ad hoc pipelines

- Ad hoc pipelines are difficult to figure out
  - Definition is spread around in source files
  - Big picture is obscured
- Difficult to visualize
Automatic pipeline visualization

- RDF pipeline definition permits visualization to be auto-generated
- Self-documenting
Why a dependency graph?

- Wrappers can:
  - Keep track of node dependencies
  - Invoke a node automatically as needed

- Think Ant or Make
Why cache oriented?

- **Node is updated only** if one of its inputs changed
  - Otherwise cached output is used
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
Why a uniform interface?

Simplifies implementation

Same interface for both:

• Internal / homogeneous pipelines
• Distributed / heterogeneous pipelines . . .
Internal / homogeneous versus distributed / heterogeneous pipeline

• Internal / homogeneous:
  – Same server
  – Same processing environment
  – E.g. named graphs within the same Java RDF store

• Distributed / heterogeneous:
  – Different server
  – Different processing environment
  – E.g., Java RDF store on one server to relational database on another
Why HTTP?

• Simple, ubiquitous protocol
• Allows any data format (RDF or other)
• Built-in cache support: Last-Modified, ETag, etc.
• Easy testing
Example pipeline: sum two numbers

Pipeline definition:

1. @prefix p: <http://purl.org/pipeline/ont.n3#> .
2. @prefix : <http://localhost/> .
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.
4. print "\$sum\n";`
Why SPARQL?

- Standard RDF query language
- Can help bridge RDF <-> relational data
  - Relational --> RDF: mappers are available
    [http://www.w3.org/wiki/Rdb2RdfXG/StateOfTheArt](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 bookStore22 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
8. WHERE
9. { GRAPH <http://example/bookStore1>
11.     FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
13. } }
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. INSERT


7. WHERE

8. { GRAPH <http://example/bookStore1>


10.  FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )


12. } }
BookStore2 pipeline definition

1. @prefix p: <http://purl.org/pipeline/ont#> .
2. @prefix : <http://localhost/> .
4. :bookStore2 a p:JenaNode ;
   p:inputs ( :bookStore1 ) ;
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
8. WHERE
9. { GRAPH <http://example/bookStore1> 
11. FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime ) 
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
11.     FILTER ( ?date > "1970-01-01T00:00:00-02:00"^^xsd:dateTime )
13.  } }
Why RDF pipeline definition?

• Graphs are natural to RDF
• Permits inferencing
• Easy visualization

• Efficiency . . .
Logical pipeline communication

- Uniform interface: RESTful HTTP
• **Wrappers can transparently:**
  – Use native protocols *within* an environment
  – Use HTTP *between* environments

• **Example:**
  – Inferencing from one named graph to another in an RDF store
1. @prefix p: <http://purl.org/pipeline/ont#> .
2. @prefix b1: <http://server1/> .
3. @prefix b2: <http://server2/> .
5. b2:bookStore2 a p:JenaNode ;
6. p:inputs ( b1:bookStore1 ) ;
1. @prefix p: <http://purl.org/pipeline/ont#> .
2. @prefix b1: <http://server1/> .
3. @prefix b2: <http://server1/> .
5. b2:bookStore2 a p:JenaNode ;
6. p:inputs ( b1:bookStore1 ) ;
Incremental update of graph collections

• Problem: Big datasets take too long to re-generate
  – E.g., ~200k patient records can take many hours
  – Want to update only what needs to be updated

• Big datasets are often composed of many (independent) subgraphs
  – E.g., one named graph per patient record

• One solution: Update only the subgraphs that changed

• How?
Generating one graph collection from another

- A and B contain a large number of items
- Each item in A corresponds to one item in B
- The same function $f$ creates each $b_i$ from $a_i$
- Wasteful to regenerate every $b_i$ when only a few $a_i$'s have changed
Collection generation as a mapping

- “Map” function applies f to each item in A
- B is updated from A by map(f, A):
  For each i, bi = f(ai)
Pipeline definition using map

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

Updater needs no logic for incremental update!
• Map can also be used with multiple inputs
• D is updated by map(f, A, B, C):
  For each i, \( d_i = f(a_i, b_i, c_i) \)
Pipeline definition using map with multiple inputs

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

• When should a node be updated? E.g., processedRecords
  – Whenever patientRecords or labData changes? (Eager)
  – Only when a report is requested? (Lazy)

• Trade-off: Latency versus processing time
Why wrappers? Update policies

• **Update policy controls** *when* a node's data is updated:
  - lazy – When output is requested
  - eager – When any of the node's inputs changes
  - periodic – Every $n$ seconds
  - eagerThrottled – When an input changes and the node has not been updated within the past $n$ seconds
  - Etc.

• **Handled by wrapper** – independent of node update logic
Problem: How to indicate what data is wanted?

- report-2010-feb only needs a subset of processedRecords
- How can it tell processedRecords what date range it wants?
Solution: Propagate parameters upstream

- dateMin and dateMax parameters are passed upstream

Parameters:
- dateMin: 2011-02-01
- dateMax: 2011-03-01
Propagating parameters upstream

- Different parameters may be needed by different stages
Propagating parameters upstream

- Different parameters may be needed by different inputs

Parameters:
- $(\text{rec}01, \text{rec}05, \text{rec}08, \ldots)$
- $(a1, a5, a8, \ldots)$
Output flows downstream
Parameters flow upstream

How?
Parameter nodes

- Parameters can be achieved by an extra node
  - Virtual node D consists of two physical nodes: d, dp
- Parameter node (dp) is no different than other nodes, but used as a parameter node by C.
- Parameter nodes are like additional input nodes
Pipeline definition with parameter

1. @prefix p: <http://purl.org/pipeline/ont#> .
2. @prefix : <http://localhost/> .
4. :c a p:Node ;
5. p:inputs ( :a ) ;
6. p:parameters ( :dp ) ;
7. p:updater "c-updater" .
8. :d a p:Node ;
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
12.############ Node properties ################
16.
17.# p:output 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.
21.
22.# p:updater specifies the updater method for a Node.
23.# It is node-type-specific, e.g., a script for CommandNode .
25.
26.# p:updaterType specifies the type of updater used.
27.# It is node-type-specific.
29.######### Rules #########

13.# A Node dependsOn its inputs and parameters:
Summary

• Flexible:
  – Any kind of data – not only RDF
  – Any kind of custom code (using wrappers)
  – Internal homogeneous pipelines
  – Distributed heterogeneous pipelines

• Efficient
  – Updates only what needs to be updated
  – Communicates with native protocols when possible, HTTP otherwise

• 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
Nodes

• Each node has:
  – A **URI** (to identify it)
  – One output “**cache**”
  – An **update method** ("updater") for refreshing its output cache

• A node may also have:
  – **Inputs** (from upstream)
  – **Parameters** (from downstream)
Basic node functions

• **Update cache**
  – Triggered by an input or parameter change
  – Changes the state of the node
  – Handled by custom logic “updater” method

• **Serve an output request**
  – Triggered by GET request
  – Normally handled by wrapper
  – Does not (normally) change the state of the node
Output cache

- One per node
  - All downstream nodes see the same data
- Logical data store, e.g.:
  - Named graph within an RDF store
  - File
  - Database
- Not necessarily physical
  - Different nodes may share the same physical store
- Has an associated lastModified datetime
- Allows the node to serve data without re-running its updater
Example: Node

- Updater is an arbitrary command script
- Output data cached as a file
- Command script is invoked as:
  
  \texttt{cmd \ thisUri \ [ \ i1 \ i2 \ ... \ ] \ [ \ p1 \ p2 \ ... \ ] > cacheFile}

- Where:
  - \texttt{cmd} – Command to invoke to update \texttt{cacheFile}
  - \texttt{thisUri} – URI of this node
  - \texttt{i1, i2, ...} – Cache filenames from input nodes
  - \texttt{p1, p2, ...} – Cache filenames from parameter nodes
  - \texttt{cacheFile} – Cache file for thisUri node
(Demo 0: Hello world)
Example: JenaNode

- Output data cached as a **named graph**
- Updated by:
  - Sparql INSERT
  - Rules
  - Reasoner
  - Java function
- `p:updaterType` can specify the type of updater used
Potential JenaNode definition

@prefix p: <http://purl.org/pipeline/ont#> .
@prefix : <http://localhost/> .
:e a :JenaNode ;
  p:updater "e-updater.sparql" .
# 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
{
}
Parameter nodes are data sources for two purposes:

1. Additional input to regular node (in computing output)
2. Propagating parameters farther upstream
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
Example 2: Passing parameters upstream

• Legend:
  → Regular node output to regular node input
  ← Param node output to param node input
  ← Param node output to regular node param

Diagram:
- b → cp
- cp → e
- cp → d
- b → a
- cp → d
- ep → c
- e → c
- dp → c
- d → c
- 4 → c

Node labels:
- b
- cp
- c
- e
- dp
- d
- ep
- a
- 4
Example 2: Pipeline with parameters in N3

:a p:cache "a-cache.txt".
a p:updater "a-updater".
a p:parameters ( :cp ).

:b p:cache "b-cache.txt".
b p:updater "b-updater".
b p:parameters ( :cp ).

:c p:cache "c-cache.txt".
c p:updater "c-updater".
c p:inputs (:a :b).
:cp p:cache "cp-cache.txt".
:cp p:updater "cp-updater".

d p:cache "d-cache.txt".
d p:updater "d-updater".
d p:inputs ( :c ).
:dp p:cache "dp-cache.txt".

e p:cache "e-cache.txt".
e p:updater "e-updater".
e p:inputs ( :c ).
:ep p:cache "ep-cache.txt".
(Demo: Sparql INSERT)
Example 1: Multiple nodes

Generates numbers: 10, 20, 30, etc.

Generates numbers: 1, 2, 3, 4, etc.

- Node c consumes records from a & b
- Nodes d & e consume records from c
Data in node a

\[
\begin{align*}
<s01> <a1> & 111 . \\
<s01> <a2> & 121 . \\
<s01> <a3> & 131 . \\
<s02> <a1> & 112 . \\
<s02> <a2> & 122 . \\
<s02> <a3> & 132 . \\
<s03> <a1> & 113 . \\
<s03> <a2> & 123 . \\
<s03> <a3> & 133 . \\
<s04> <a1> & 114 . \\
\ldots
<s09> <a3> & 139 .
\end{align*}
\]
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

Diagram:
- Node b
- Node c
- Node d
- Node e
- Edges connecting b to c, c to d, and c to e
Data in node c

\[\langle s01 \rangle \langle a1 \rangle 111.\]
\[\langle s01 \rangle \langle a2 \rangle 121.\]
\[\langle s01 \rangle \langle a3 \rangle 131.\]
\[\langle s01 \rangle \langle b1 \rangle 211.\]
\[\langle s01 \rangle \langle b2 \rangle 221.\]
\[\langle s01 \rangle \langle b3 \rangle 231.\]
\[\langle s01 \rangle \langle c1 \rangle 111211.\]
\[\langle s01 \rangle \langle c2 \rangle 121221.\]
\[\langle s01 \rangle \langle c3 \rangle 131231.\]
\[\langle s02 \rangle \langle a1 \rangle 112.\]
\[\ldots\]
\[\langle s09 \rangle \langle c3 \rangle 139239.\]
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 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)
Optimizing internal communication
Inter-node communication: Logical view

- Nodes pass data from one to another . . .
  - But *how*?
• Framework handles inter-node communication
  – Uniform virtual interface makes communication easy
• By default, nodes use HTTP
  – Common denominator
But nodes that share an implementation environment communicate directly, using native protocol, e.g.:
- One SesameNode to another in the same RDF store
- One Node to another on the same server

Wrappers handle both native protocol and HTTP
Optimizing external communication
Optimizing HTTP GET with parameter node

- Suppose node d has parameter node dp
- When d needs to GET data from c, c must first GET parameter data from dp:
  1. Request: d sends GET request to c
  2. Request: c sends GET request to dp
  3. Response: dp responds to c
  4. Response: c responds to d

Extra round trip
Optimized HTTP GET with parameter node

To optimize, d can send dp response *preemptively* to c with its GET request

- Query parameters can include:
  - Node URI of dp
  - Last-Modified, ETag, Content-Type, Body, etc.
- i.e., the same response info as if c had issued a GET request to dp

[Thanks to Steve Battle for inspiring this optimization]
fromPairs and toPairs

- Transformation from fromPairs to toPairs
Logic for mapcar update

1. function MapcarUpdate(Method method, 
2. Pairs toPairs, Pairs fromPairs) {
3.   foreach Key k in keys of fromPairs {
4.     if !exists(toPairs{k})
5.       || fromPairs{k}.updateTime > toPairs{k}.updateTime {
6.       Update(toPairs{k});
7.     }
8.   }
9. }
Eager update logic

1. /* Called after parent is updated */
2. function EagerUpdate(PCache parent) {
3.     foreach PCache child that depends on parent {
4.         child.update();
5.         EagerUpdate(child);
6.     }
7. }
Lazy update logic

1. /* Called before getting data from child */
2. function LazyUpdate(PCache child) {
3.   /* “contributes to” is the inverse of “depends on” */
4.   foreach PCache parent that contributes to child {
5.       LazyUpdate(parent);
6.   }
7.   if IsOutOfDate(child) then child.update();
8. }
Example 2: merging, inferring

- Node c merges and augments records
- Nodes d&e select subsets
Semantic Data Federation

• Integrating data from diverse:
  – vocabularies, formats and data sources

• Producing data for diverse:
  – vocabularies, formats and applications
Semantic Data Federation

- Integrating data from diverse:
  - vocabularies, formats and data sources
- Producing data for diverse:
  - vocabularies, formats and applications
Semantic Data Federation

- Does transformations, caching, etc.
- Different sources use different vocabularies/ontologies
- Different consumers use different vocabularies/ontologies
- See also:
Persistent Caching

- Semantic Data Federation does **persistent caching**
- Many pcaches may be used
- Each should be updated automatically
Persistent Cache (pcache)

• Each pcache can be regenerated based on its
  – Update method (e.g., SPARQL rules)
  – Update policy (eager, periodic, lazy, etc.)
  – Dependencies (other pcaches, data sources, ontologies, rules)

• Pcache update is like running a makefile:
  – Dependencies are analyzed
  – Each out-of-date pache is updated based on its update method and update policy
• child1 dependsOn parent1 and parent2
• Inverse: parent1 contributesTo child1
• or maybe: parent1 isRequiredBy/supports/supplies/influences/affects child1
Inside a pcache

- Each pcache has an update method, a dataset, an update policy and other metadata, e.g., provenance, updateTime
Eager update

- If parent1 or parent2 are updated, then run child1's update method, and so on recursively
- In general: if any parent is updated, update the child
• Each pcache has a lastUpdateTime

• If child1 is requested but out of date, then:
  – Recursively make sure parent1 and parent2 are up to date
  – Run child1's update method
Example: Monthly report

- Downstream reports should auto update when baseRecords change

Diagram:

- baseRecords
  - Parameters: (rec01, rec05, rec08, ...)
- transformedAuxData
  - Parameters: (a1, a5, a8, ...)
- augmentedRecords
  - Parameters: (rec01, rec05, rec08, ...)
- processedRecords
  - Parameters: dateMin: 2011-02-01, dateMax: 2011-03-01
- report-2011-01
- report-2011-02
A node's output 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, augmentedRecords 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

\[
\begin{align*}
&lt;s01&gt; &lt;a1&gt; & 111 . \\
&lt;s01&gt; &lt;a2&gt; & 121 . \\
&lt;s01&gt; &lt;a3&gt; & 131 . \\
&lt;s01&gt; &lt;b1&gt; & 211 . \\
&lt;s01&gt; &lt;b2&gt; & 221 . \\
&lt;s01&gt; &lt;b3&gt; & 231 . \\
&lt;s01&gt; &lt;c1&gt; & 111211 . \\
&lt;s01&gt; &lt;c2&gt; & 121221 . \\
&lt;s01&gt; &lt;c3&gt; & 131231 . \\
&lt;s02&gt; &lt;a1&gt; & 112 . \\
\ldots \\
&lt;s09&gt; &lt;c3&gt; & 139239 .
\end{align*}
\]
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 .
\ldots
<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.
Option 1: Monolithic, big bang process

- One monster process that handles all vocabularies, formats, data sources and applications
- Pros: Highest potential processing efficiency
- Cons: Huge complex ontology; Very risky to build (requirements evolve); Difficult to maintain
Semantic Data Federation

- Need to integrate and generate data from distributed, diverse:
  - vocabularies, formats and data sources
- Producing data for distributed, diverse:
  - vocabularies, formats and applications
- While each data consumer sees a single data source
A node's output cache becomes **stale** if any of its input nodes change
- E.g., B's cache becomes stale if A's cache changes

Updater can refresh it

NOTE: Because different nodes may have different clocks (clock skew), the technique for determining staleness is slightly different from that used by Make
Data in a large enterprise

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

- Easy to create nodes
  - Node may be written in any convenient language/environment
  - Any kind of data and storage – not only RDF
  - Node does not need to know how other nodes are implemented
- Easy to connect nodes
  - Add a few lines of RDF
- Parameters can be passed upstream
- Nodes are invoked automatically, based on dependencies, to update node data
- Flexible node data update policies
  - E.g., eager, lazy, periodic
- Efficient
  - Updates only what should be updated
  - Low node communication overhead
Example pipeline definition (in N3)

1. @prefix p: <http://purl.org/pipeline/ont#> .
2. @prefix : <http://localhost/> .
4. :labData a p:Node .
5. :transformedLabData a p:Node .
8. :processedRecords a p:Node .
11. p:inputs ( :processedRecords ) .
Example pipeline definition (in N3)

1. @prefix p: <http://purl.org/pipeline/ont#> .
2. @prefix : <http://localhost/> .
4. :labData a p:Node .
5. :transformedLabData a p:Node .
8. :processedRecords a p:JenaNode .
11. p:inputs ( :processedRecords ) .