HARVARD
School of Engineering and Applied Sciences

SRI International®

Declaratively Processing Provenance Metadata

Scott Moore
Joint work with Ashish Gehani and Natarajan Shankar
TaPP 2013
2 April 2013

Processing Provenance Streams

- Systems that collect low-level provenance generate a lot of data…

Processing Provenance Streams

- Systems that collect low-level provenance generate a lot of data…
- SPADE collects I/O system calls
  - Thousands of I/O related edges during process
  - Need some way to manage all that data!
    - Reduce storage requirements, simplify queries, …

Processing Provenance Streams

- Systems that collect low-level provenance generate a lot of data…
- SPADE collects I/O system calls
  - Thousands of I/O related edges during process
    - Need some way to manage all that data!
      - Reduce storage requirements, simplify queries, …
- SPADE uses *provenance filters*
  - Running example: (Gehani et al, POLICY 2010)
    Aggregate I/O events for consecutive sequences of reads or writes (an I/O Run)

Aggregating File I/O Provenance

```
public void putEdge(AbstractEdge incomingEdge) {
  if (incomingEdge instanceof Used) {
    Used usedEdge = (Used) incomingEdge;
    String fileVertexHash = usedEdge.getDestinationVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(usedEdge.getSourceVertex().hashCode());
    if (!reads.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = new HashSet<String>();
      tempSet.add(processVertexHash);
      reads.put(fileVertexHash, tempSet);
    } else {
      HashSet<String> tempSet = reads.get(fileVertexHash);
      if (tempSet.contains(processVertexHash)) {
        vertexBuffer.remove(usedEdge.getDestinationVertex());
        return;
      } else { tempSet.add(processVertexHash); }
    }
    vertexBuffer.remove(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge);
    if (writes.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = writes.get(fileVertexHash);
      tempSet.remove(processVertexHash);
    }
  } else if (incomingEdge instanceof WasGeneratedBy) {
    WasGeneratedBy wgb = (WasGeneratedBy) incomingEdge;
    String fileVertexHash = wgb.getSourceVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(wgb.getDestinationVertex().hashCode());
    if (!writes.containsKey(fileVertexHash)) {
```

Aggregating File I/O Provenance

```
public void putEdge(AbstractEdge incomingEdge) {
  if (incomingEdge instanceof Used) {
    Used usedEdge = (Used) incomingEdge;
    String fileVertexHash = usedEdge.getDestinationVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(usedEdge.getSourceVertex().hashCode());
    if (!reads.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = new HashSet<String>();
      tempSet.ad
      reads.put(
    } else {
      HashSet<St
      if (tempSet.contains(processVertexHash)) {
        vertexBuffer.remove(usedEdge.getDestinationVertex());
        return;
      } else { tempSet.add(processVertexHash); }
    }
    vertexBuffer.remove(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge);
    if (writes.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = writes.get(fileVertexHash);
      tempSet.remove(processVertexHash);
    }
  } else if (incomingEdge instanceof WasGeneratedBy) {
    WasGeneratedBy wgb = (WasGeneratedBy) incomingEdge;
    String fileVertexHash = wgb.getSourceVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(wgb.getDestinationVertex().hashCode());
    if (!writes.containsKey(fileVertexHash)) {
```

What does this do?

Aggregating File I/O Provenance

```
public void putEdge(AbstractEdge incomingEdge) {
  if (incomingEdge instanceof Used) {
    Used usedEdge = (Used) incomingEdge;
    String fileVertexHash = usedEdge.getDestinationVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(usedEdge.getSourceVertex().hashCode());
    if (!reads.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = new HashSet<String>();
      tempSet.ad
      reads.put(
    } else {
      HashSet<St
      if (tempSet.contains(processVertexHash)) {
  vertexBuffer.remove(usedEdge.getDestinationVertex());
  putInNextFilter(usedEdge.getDestinationVertex());
  putInNextFilter(usedEdge);
  if (writes.containsKey(fileVertexHash)) {
    HashSet<String> tempSet = writes.get(fileVertexHash);
    tempSet.remove(processVertexHash);
  }
} else if (incomingEdge instanceof WasGeneratedBy) {
  WasGeneratedBy wgb = (WasGeneratedBy) incomingEdge;
  String fileVertexHash = wgb.getSourceVertex().getAnnotation(artifactKey);
  String processVertexHash = Integer.toString(wgb.getDestinationVertex().hashCode());
  if (!writes.containsKey(fileVertexHash)) {
```

What does this do?

What does the provenance it records mean?

Aggregating File I/O Provenance

```
public void putEdge(AbstractEdge incomingEdge) {
  if (incomingEdge instanceof Used) {
    Used usedEdge = (Used) incomingEdge;
    String fileVertexHash = usedEdge.getDestinationVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(usedEdge.getSourceVertex().hashCode());
    if (!reads.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = new HashSet<String>();
      vertexBuffer.remove(usedEdge.getDestinationVertex());
      putInNextFilter(usedEdge.getDestinationVertex());
      putInNextFilter(usedEdge);
      if (writes.containsKey(fileVertexHash)) {
        HashSet<String> tempSet = writes.get(fileVertexHash);
        tempSet.remove(processVertexHash);
      }
    } else if (incomingEdge instanceof WasGeneratedBy) {
      WasGeneratedBy wgb = (WasGeneratedBy) incomingEdge;
      String fileVertexHash = wgb.getSourceVertex().getAnnotation(artifactKey);
      String processVertexHash = Integer.toString(wgb.getDestinationVertex().hashCode());
      if (!writes.containsKey(fileVertexHash)) {
```

We propose a declarative programming language for processing provenance metadata.

Datalog + Time: a Simple Event Logic

- Datalog + enough to process events
  - Declarative networking: OverLog, Dedalus/Bloom, etc.
  - Synchronous programming: ESTEREL, LUSTRE, etc.

Datalog + Time: a Simple Event Logic

- Datalog + enough to process events
  - Declarative networking: OverLog, Dedalus/Bloom, etc.
  - Synchronous programming: ESTEREL, LUSTRE, etc.
- SEL programs operate on streams of events
  - Events are Datalog facts at an instant in time
  - Inference rules generate new facts

Datalog + Time: a Simple Event Logic

- Datalog + enough to process events
  - Declarative networking: OverLog, Dedalus/Bloom, etc.
  - Synchronous programming: ESTEREL, LUSTRE, etc.
- SEL programs operate on streams of events
  - Events are Datalog facts at an instant in time
  - Inference rules generate new facts
- A single temporal operator 'previously' (written '**?**')

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

Modules operate on streams of events and generate new streams of events.

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

A change event happens when a new value, `value(Cur)`, is different than an old one, `?value(Old)`

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

The first value/1 event is received. There is no previous value, so the inference rule doesn't fire.

```
value(0).
```

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

When the next event is received, the inference rule for change fires and generates a new event.

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

When the next event is received, the inference rule for change fires and generates a new event.

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

‘Previously’ refers to the immediately preceding time step.

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

‘Previously’ refers to the immediately preceding time step.

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```

SEL by Example

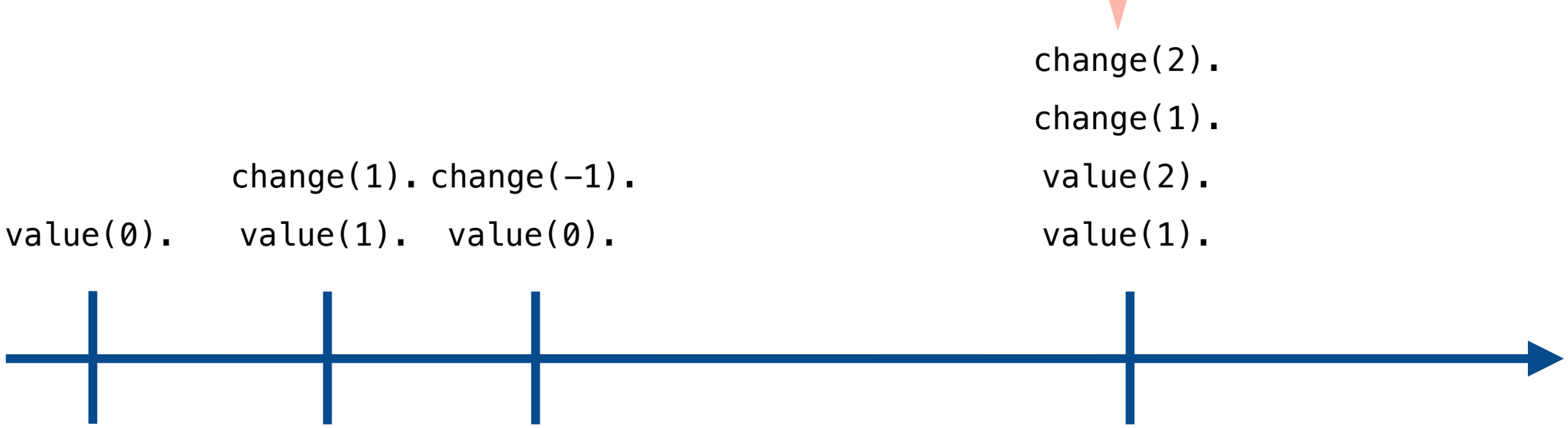
```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur) @ old(Old),
    Cur != Old,
    Change = Cur - Old.

end delta.
```

Multiple input events can happen simultaneously.
Like Datalog, SEL computes all possible facts.

SEL by Example

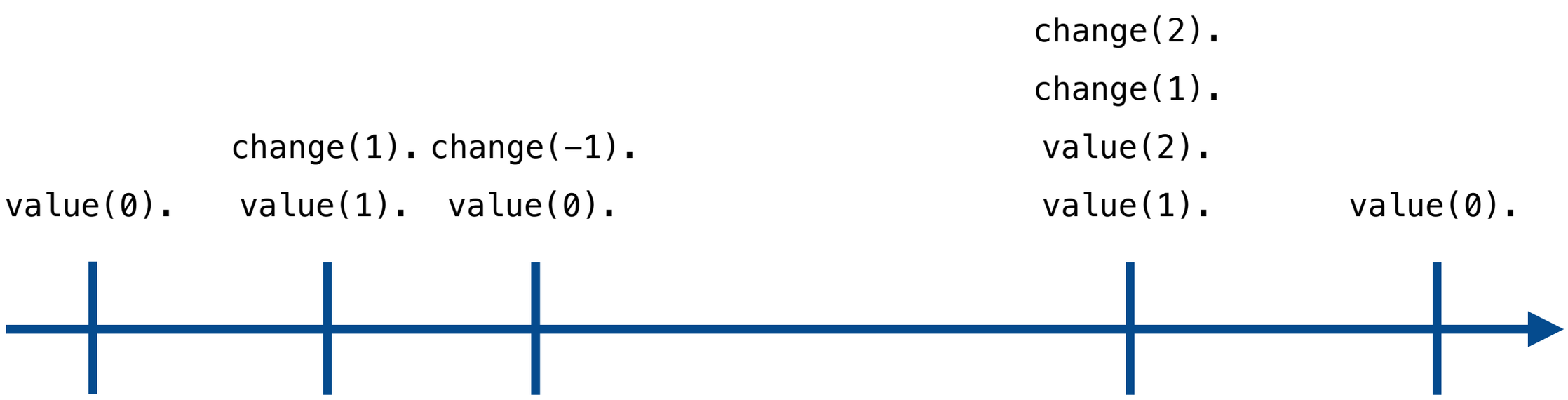
```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur) @ old(Old),
    Cur != Old,
    Change = Cur - Old.

end delta.
```

Multiple input events can happen simultaneously.
Like Datalog, SEL computes all possible facts.



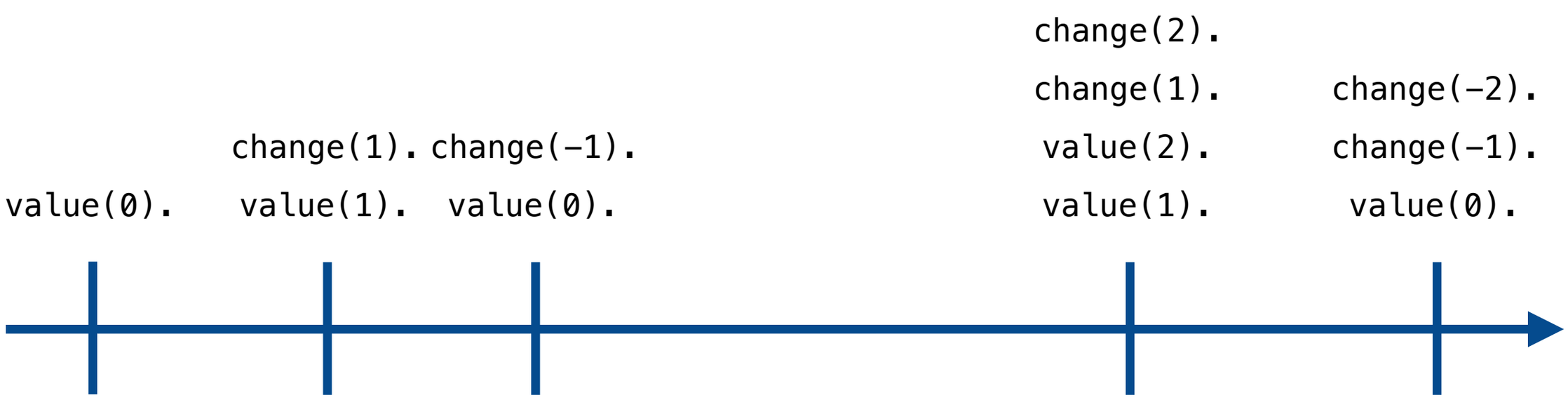
SEL by Example

```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```



SEL by Example

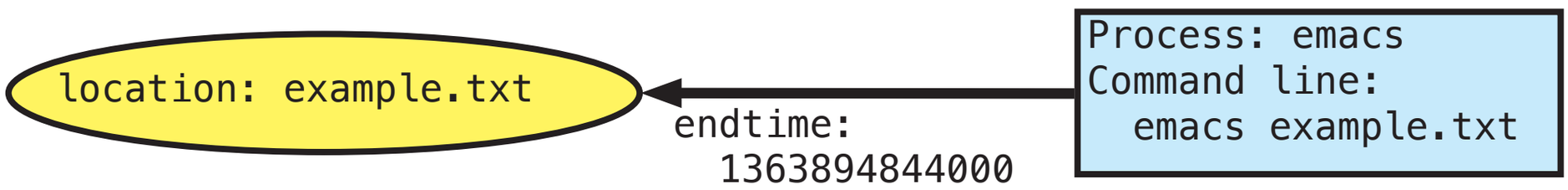
```
module delta.
input value/1.
output change/1.

change(Change) :-
    value(Cur), ?value(Old),
    Cur != Old, Change = Cur - Old.

end delta.
```



Encoding Provenance

- Represent provenance as edges and vertices in a provenance graph with attributes
- Similar to Missier and Belhajjame's PROV encoding (IPAW '12)

```
process(42,process).
attr(42,commandline,"emacs example.txt").
vertex(43,artifact).
attr(43,location,"example.txt").
edge(44,used,42,43).
attr(44,endtime,1363894844000).
```



Aggregating File I/O Provenance

```
public void putEdge(AbstractEdge incomingEdge) {
  if (incomingEdge instanceof Used) {
    Used usedEdge = (Used) incomingEdge;
    String fileVertexHash = usedEdge.getDestinationVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(usedEdge.getSourceVertex().hashCode());
    if (!reads.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = new HashSet<String>();
      tempSet.add(processVertexHash);
      reads.put(fileVertexHash, tempSet);
    } else {
      HashSet<String> tempSet = reads.get(fileVertexHash);
      if (tempSet.contains(processVertexHash)) {
        vertexBuffer.remove(usedEdge.getDestinationVertex());
        return;
      } else { tempSet.add(processVertexHash); }
    }
    vertexBuffer.remove(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge.getDestinationVertex());
    putInNextFilter(usedEdge);
    if (writes.containsKey(fileVertexHash)) {
      HashSet<String> tempSet = writes.get(fileVertexHash);
      tempSet.remove(processVertexHash);
    }
  } else if (incomingEdge instanceof WasGeneratedBy) {
    WasGeneratedBy wgb = (WasGeneratedBy) incomingEdge;
    String fileVertexHash = wgb.getSourceVertex().getAnnotation(artifactKey);
    String processVertexHash = Integer.toString(wgb.getDestinationVertex().hashCode());
    if (!writes.containsKey(fileVertexHash)) {
```

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

Module for buffering and combining provenance events. Controlled by `emit/1`, `aggr/2`, and `drop/1`.

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

Immediately output edges and vertices for provenance not related to File I/O.

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

A `Process` reads a `File` (represented by an `Artifact`) if it `used` it.

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

A Process is reading a File (an I/O run) if after the first read in a sequence occurs.

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

A reading I/O run continues until there is a write (or the file is closed, etc.)

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

If a read happens during an I/O run, aggregate the provenance edges.

Aggregating File I/O Provenance

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    ?reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

Output the edges in the buffer associated with an I/O run when it ends.

Benefits of Declarative Programming

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

Benefits of Declarative Programming

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

- Easy to understand
  - DSL for events
  - Abstractions for buffering, etc.

Benefits of Declarative Programming

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

- Easy to understand
  - DSL for events
  - Abstractions for buffering, etc.
- Built in query language
  - Reason about provenance as you process it!

Benefits of Declarative Programming

```
import Aggregate.

emit(ID) :- vertex(ID,Type),
    Type != artifact.
emit(ID) :- edge(ID,Type,_,_),
    Type != used, Type != wasGeneratedBy.

read(ID,Process,Artifact,File) :-
    edge(ID,used,Process,Artifact),
    attr(Artifact,location,File).

reading(ID,Process,Artifact,File) :-
    read(ID,Process,Artifact,File),
    ¬?reading(_,Process ,_,File).
reading(ID,Process ,Artifact,File) :-
    reading(ID,Process ,Artifact,File),
    ¬write(_,_,_,File).

aggr(ReadSeries,Read) :-
    reading(ReadSeries,Process,_,File),
    read(Read,Process,_,File).

emit(Artifact) :-
    reading(_,_,Artifact ,File),
    write(_,_,_,File).
emit(Read) :-
    reading(Read,_,_,File),
    write(_,_,_,File).
```

- Easy to understand
  - DSL for events
  - Abstractions for buffering, etc.
- Built in query language
  - Reason about provenance as you process it!
- Simple formal semantics
  - Provenance of provenance?

Other applications

- Filters for provenance collection
  - Filtering application-level provenance
  - Aggregating provenance from multiple sources

Other applications

- Filters for provenance collection
  - Filtering application-level provenance
  - Aggregating provenance from multiple sources
- Monitoring provenance for security (Rachapalli et al, TaPP 2012)

Other applications

- Filters for provenance collection
  - Filtering application-level provenance
  - Aggregating provenance from multiple sources
- Monitoring provenance for security (Rachapalli et al, TaPP 2012)

```
tainted(Arifact,File) :-
    attr(Artifact,location,File), secret(File).

tainted(Process,Secret) :-
    edge(_,used,Process,Artifact), tainted(Artifact,Secret).

tainted(Artifact,Secret) :-
    edge(_,wasGeneratedBy,Artifact,Process), tainted(Process,Secret).

tainted(Node,Secret) :-
    ?tainted(Node,Secret), ¬attr(Node,‘endtime’,_).

leak(Secret,Connection) :-
    edge(_,wasGeneratedBy,Connection,Process),
    attr(Conntection,subtype,network),
    tainted(Process,Secret).
```

Other applications

- Filters for provenance collection
  - Filtering application-level provenance
  - Aggregating provenance from multiple sources
- Monitoring provenance for security (Rachapalli et al, TaPP 2012)

```
tainted(Arifact,File) :-
    attr(Artifact,location,File), secret(File).

tainted(Process,Secret) :-
    edge(_,used,Process,Artifact), tainted(Artifact,Secret).

tainted(Artifact,Secret) :-
    edge(_,wasGeneratedBy,Artifact,Process), tainted(Process,Secret).

tainted(Node,Secret) :-
    ?tainted(Node,Secret), ¬attr(Node,‘endtime’,_).

leak(Secret,Connection) :-
    edge(_,wasGeneratedBy,Connection,Process),
    attr(Conntection,subtype,network),
    tainted(Process,Secret).
```

- Other online provenance queries?

Conclusions

- Declarative programming makes processing provenance metadata much easier

Conclusions

- Declarative programming makes processing provenance metadata much easier
- Many interesting opportunities for processing provenance as it is generated!