XML Data Streams

Data Stream Processing

- What is a data stream?
  - continuous, time-varying data arriving at unpredictable rates
  - continuous updates, continuous queries
  - no stored index is available
- Sought characteristics of stream processing engines
  - real-time processing
  - high throughput, low latency, fast mean response time, low jitter
  - low memory footprint

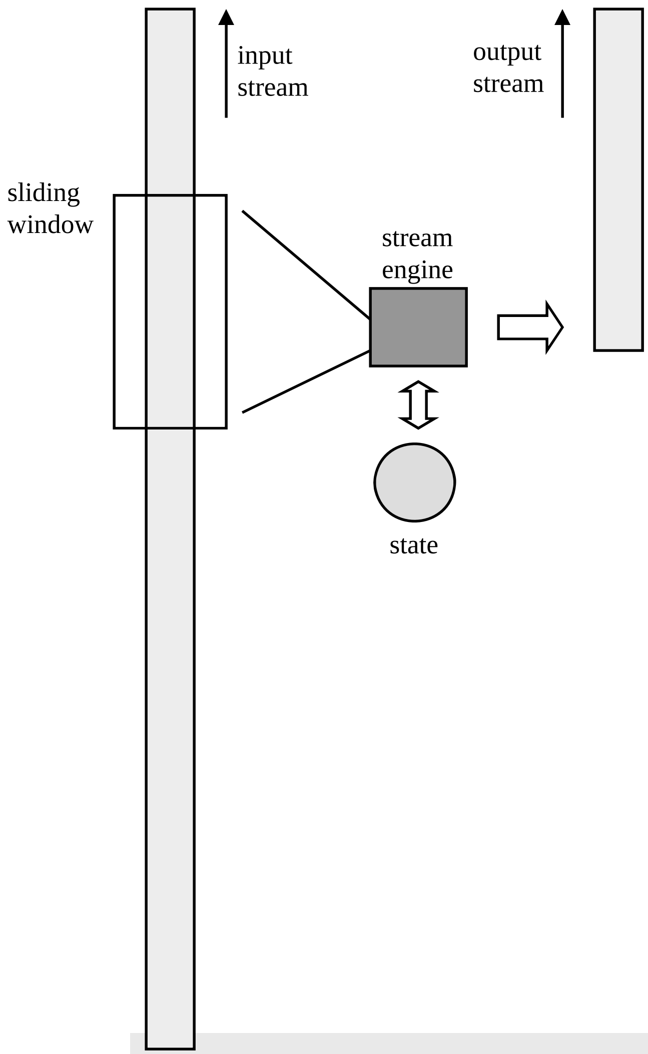
Data Stream Processing

- Why bother?
  - many data are already available in stream form
    - sensor networks, network traffic monitoring, stock tickers
    - publisher-subscriber systems
    - data stream mining for fraud detection
  - data may be too volatile to index
    - continuous measurements

XML Stream Processing

- Various sources of XML streams
  - tokenized XML documents
  - sensor XML data
  - RSS feeds
  - web service results
  - MPEG-7 (binary encoding in XML)
- Granularity
  - XML tokens (events): <tag>, </tag>, “X”, etc
  - region-encoded XML elements
  - XML fragments (hole-filler model)

Traditional Stream Processing



- Typically, a stream consists of numerical values or relational tuples
- Focuses on a sliding window
  - fixed number of tuples, or
  - fixed time span
- Extracts approximate results
- Uses a small (bounded) state
- Examples:
  - top-k most frequent values
  - group-by SQL queries (OLAP)
  - data stream mining

XML Update Streams

- A continuous (possibly infinite) sequence of XML tokens with embedded updates
  - Usually, a finite data stream followed by an infinite stream of updates
  - three basic types of tokens: <tag>, </tag>, text
  - the target of an update is a stream subsequence that contains zero, one, or more "complete" XML elements
  - the source is also a token sequence that contains complete XML elements

XML Update Streams

- the source is also a token sequence that contains complete XML elements
- updates are embedded in the data stream and can come at any time
  - update events can be interleaved with data events and with each other
  - each event must now have an id to associate it with an update
- updated regions can be updated too
- to update a stream subsequence, you wrap it in a Mutable region
- three types of updates:
  - replace
  - insertBefore
  - insertAfter

An Example

id	Event	… equivalent to
1	<a>	<a>
1		
1	StartMutable(2)	<c>
2	<c>	Y
2	X	</c>
2	</c>	<c>
1	EndMutable(2)	X
1		</c>
2	StartInsertBefore(3)	
3	<c>	
3	Y	
3	</c>	
2	EndInsertBefore(3)	
1		

Continuous Queries

- Need to decide: snapshot or temporal stream processing?
  - Snapshot: after a replace update, the replaced element is forgotten
  - Temporal: “some” of the replaced elements are kept
    - we may have repeated updates on a mutable region, forming a history list
    - each version has a time span (valid begin/end times)
    - the versions kept are determined at run time from the temporal components of the query that process that region

Continuous Queries

- Query language: XQuery with temporal extensions

e?t	time projection	“*give me the version before t secs*”
e#v	version projection	“*give me the past v version*”
e?[t]	time sliding window	“*give me all versions the last t secs*”
e#[v]	version sliding window	“*give me the v latest versions*”

- The default is “current snapshot” (version #0 at time 0)
- Much finer grain for historical data than sliding windows

Continuous Results

- One can consider a stream engine is implemented as a pipeline
  - each pipeline stage performs a very simple task
- The final pipeline stage is the **Result Display** that displays the query results continuously
  - the display can be shown as a editable text window (a GUI), where text can be inserted, deleted, and replaced at any point
  - when an update is coming in the input stream, it is propagated through the result display, where it causes an update to the display text!

Snapshot Example

- *XQuery*

```
<books>{
    for $b in stream("books")//biblio[publisher="Wiley"]/books
    where $b/author/lastname="Smith"
    order by $b/price
    return <book>{ $b/title, $b/price }</book>
}</books>
```

Display

```
<books>
    <book><title>All about XML</title><price>35</price></book>
    <book><title>XQuery for Dummies</title><price>58</price></book>
    <book><title>Querying XML</title><price>120</price></book>
…
```

A Temporal Query

- Display all stocks whose quotation increased at least 10% since the last time, sorted by their rate of change:

```
<quotes>{
            for $q in stream("tickers")//ticker
            where $q/quote > $q/quote#1 * 1.1
            order by ($q/quote - $q/quote#1) div $q/quote
            return <quote>{ $q/name, $q/quote }</quote>
}</quotes>
```

Efficient Evaluation of XQuery over Streaming Data

- XPath over Streaming Data
  - XPath is relatively simple
- XQuery over Streaming Data
  - Limited features handled
  - Focus on queries that are written for single pass evaluation
- VLDB 2005
  - Xiaogang Li , Gagan Agrawal

Ideas

- Can the given query be evaluated correctly on streaming data?

  - Only a single pass is allowed

  - Decision made by compiler, not a user

- If not, can it be correctly transformed ?
- How to generate efficient code for XQuery?

  - Computations involved in streaming application are non-trivial

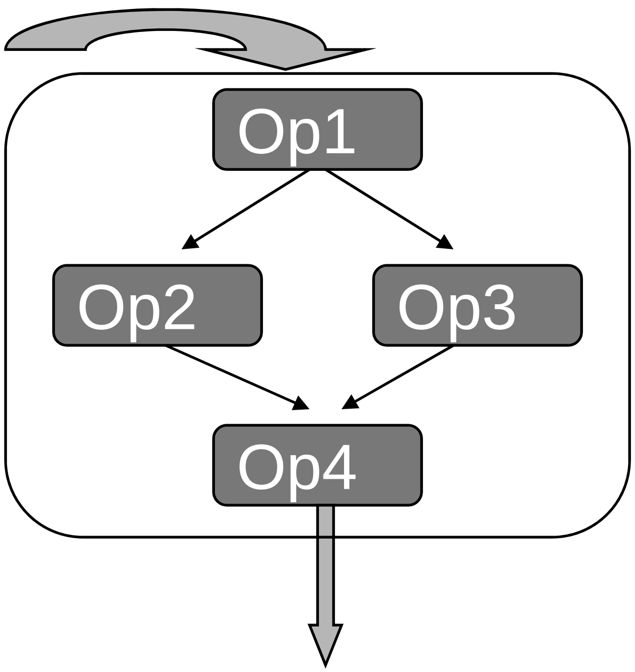
  - Recursive functions are frequently used

  - Efficient memory usage is important

The Approach

- For an arbitrary query, can it be evaluated correctly on streaming data?
  - Construct data-flow graph for a query
  - Static analysis based on data-flow graph
- If not, can it be transformed to do so ?
  - Query transformation techniques based on static analysis
- How to generate efficient code for XQuery?
  - Techniques based on static analysis to minimize memory usage and optimize code
  - Generating imperative code
    -- Recursive analysis and aggregation rewrite

Query Evaluation Model



- Single input stream
- Internal computations
  - Limited memory
  - Linked operators
- Pipeline operator and Blocking operator

Pipeline and Blocking Operators

- Pipeline Operator:

  - each input tuple produces an output tuple independently

  - Selection, Increment etc

- Blocking Operator:

  - Can only compute output after receiving all input tuples

  - Sort, Join etc

- Progressive Blocking Operator:

  (1)|output|<<|input|:   we can buffer the output

  (2) Associative and commutative operation: discard input

  - count(), sum()

Single Pass?

Pixels with x and y

Q1:

```
let $i := …/pixel
    sortby (x)
```

(1) A blocking operator exists

```
Q2:
 let $i := for $p in /pixel
        where $p/x  > ..
 x  =  count(/pixel)
```

(2) A progressive blocking operator is referred by another pipeline operator or progressive operator

Check condition 2 in a query

Challenges in Single-Pass

```
let $b = count(stream/pixel[x>0])
  for $i in stream/pixel
     return $i/x idvi $b
```

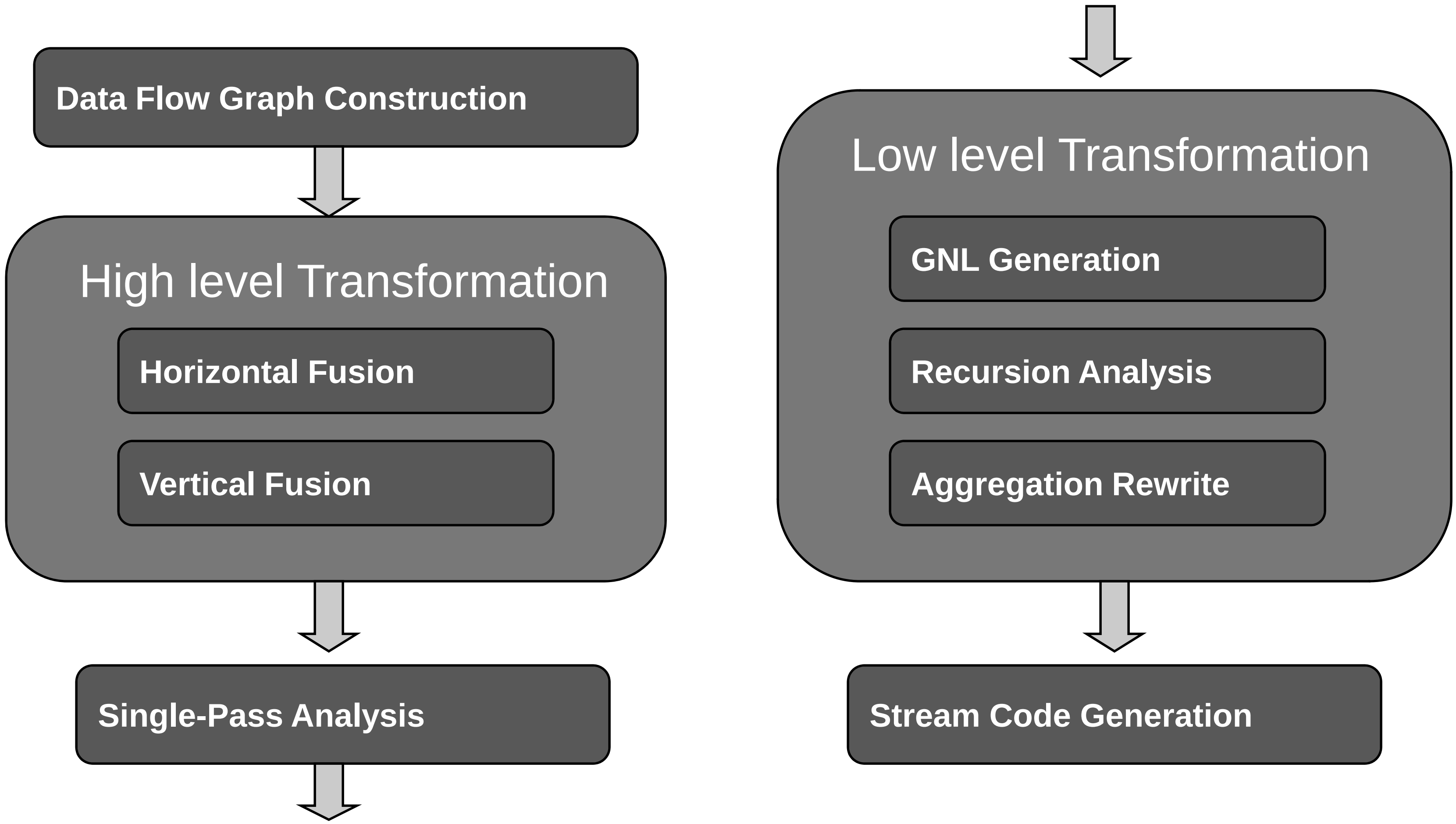
Must Analyze data dependence at expression level

```
let $b: = for $i in  stream/pixel[x>0]
   return $i
for $j in $b/y
   return $j
   where $j = count($b)
```

A Query may be complex

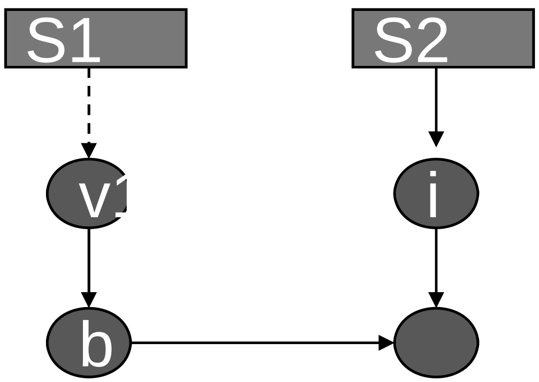
Need a simplified view of the query to make decision

Overall Framework



Stream Data Flow Graph (DFG)

```
let $b = count(stream/pixel[x>0])
  for $i in stream/pixel
     return $i/x idvi $b
```



S1:stream/pixel[x>0]

S2:stream/pixel

V1: count()

- Node represents variable:
  Explicit and implicit
  Sequence and atomic

- Edge: dependence relation
  v1->v2 if v2 uses v1
  Aggregate dependence and flow dependence

- A DFG is acyclic
- Cardinality inference is required to construct the DFG

High-level Transformation

- Goals

  1: Enable single pass evaluation

  2: Simplify the SDFG and single-pass analysis

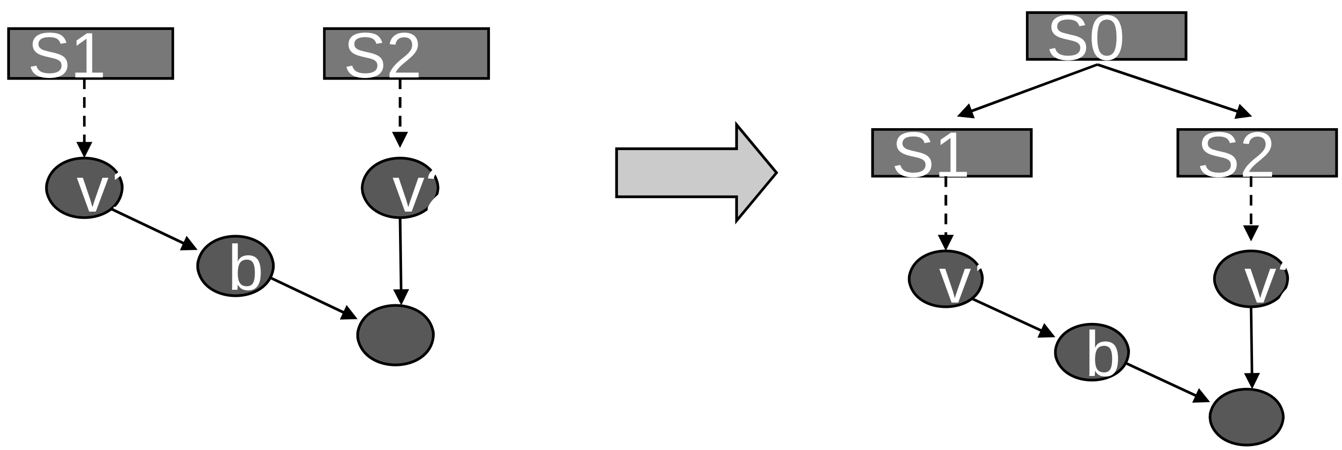
- Horizontal Fusion and Vertical Fusion

  - Based on SDFG

Horizontal Fusion

- Enable single-pass evaluation
  - Merge sequence node with common prefix

```
let $b = count(stream/pixel[x>0])
     return sum(stream/pixel/y) idvi $b
```



S1:stream/pixel[x>0]
S2:stream/pixel/y
V1: count()   V2: sum()

S0:/stream/pixel
S1:[x>0]   S2: /y
V1: count()   V2: sum()

Horizontal Fusion with Nested Loops

- Perform loop unrolling first
- Merge sequence node accordingly

```
unordered(
 for $i in (1 to 2)
   let $b: =//stream/pixel[x=$i]
      return count($b))
```

```
unordered(
   let $b1: =//stream/pixel[x=1]
   let $b2: =//stream/pixel[x=2]
      return count($b1), count($b2)
```

Horizontal Fusion: Side-effect

- May resulted incorrect result due to inter-dependence

```
let $b = count(stream/pixel[x>0])
  for $i in stream/pixel

      return $i/x idiv $b
```

⇨

```
for $i in stream/pixel
    return $i/x idiv  count()
```

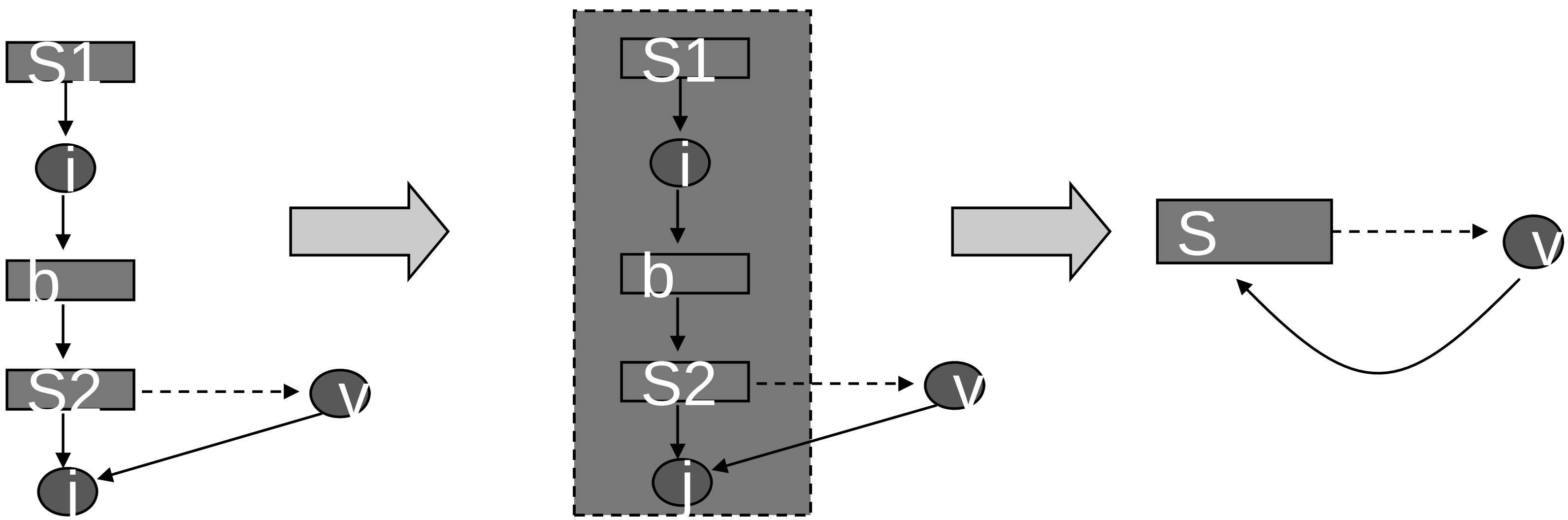
Partial result of count is used to compute output

Will be dealt with at single-pass analysis

Vertical Fusion

- Simplify DFG and single-pass analysis
  - Merge a cluster of nodes linked by flow dependence edges

```
let $b: = for $i in  stream/pixel[x>0]
   return $i
for $j in $b/y
   return $j
   where $j = count($b)
```



Single-pass Analysis

- Can a query be evaluated on-the fly?

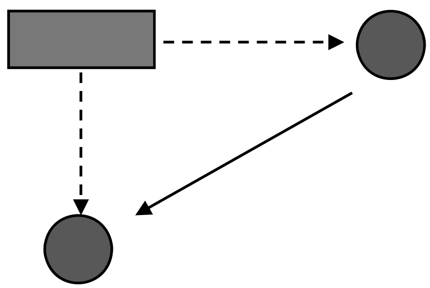
THEOREM 1. *If a query with dependence graph G=(V,E) contains more than one sequence node after vertical fusion, it can not be evaluated correctly in a single pass.*

Reason: Sequence node with infinite length can not be buffered

Single-pass Analysis

THEOREM 2. *Let S be the set of atomic nodes that are aggregate dependent on any sequence node in a stream data flow graph. For any given two elements* s1 *and* s2, *if there is a path between* s1 *and s2, the query may not be evaluated correctly in a single pass.*

Reason: A progressive blocking operator is referred by another progressive blocking operator

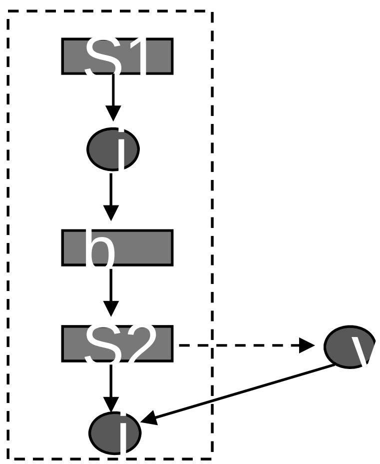


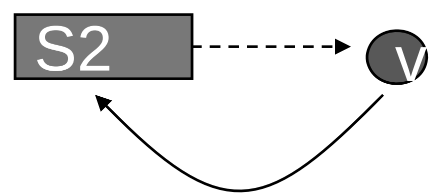
Example : count (pixel)
where /x>0.005*sum(/pixel/x)

Single-pass Analysis

THEOREM 3. *In there is a cycle in a stream data flow graph G, the corresponding query may not be evaluated correctly using a single pass.*

Reason: A progressive blocking operator is referred by a pipeline operator





Single-pass Analysis

- Check conditions corresponding to Theorem 1 2 and 3

  -Stop further processing if any condition is true

- Completeness of the analysis

  - If a query without blocking operator pass the test, it can be evaluated in a single pass

THEOREM 4. *If the results of a progressive blocking operator with an unbounded input are referred to by a pipeline operator or a progressive blocking operator with unbounded input, then for the stream data flow graph, at least one of the three conditions holds true*

Conservative Analysis

- Our analysis is conservative

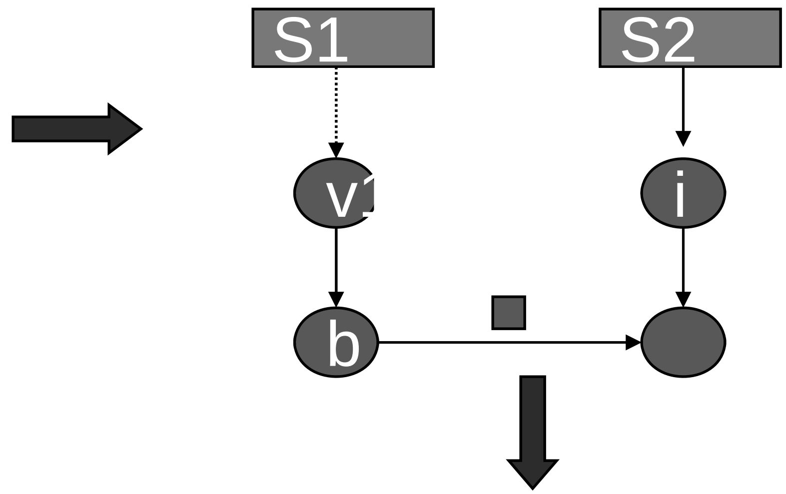
  - A valid query may be labeled as “cannot be evaluated in a single-pass”

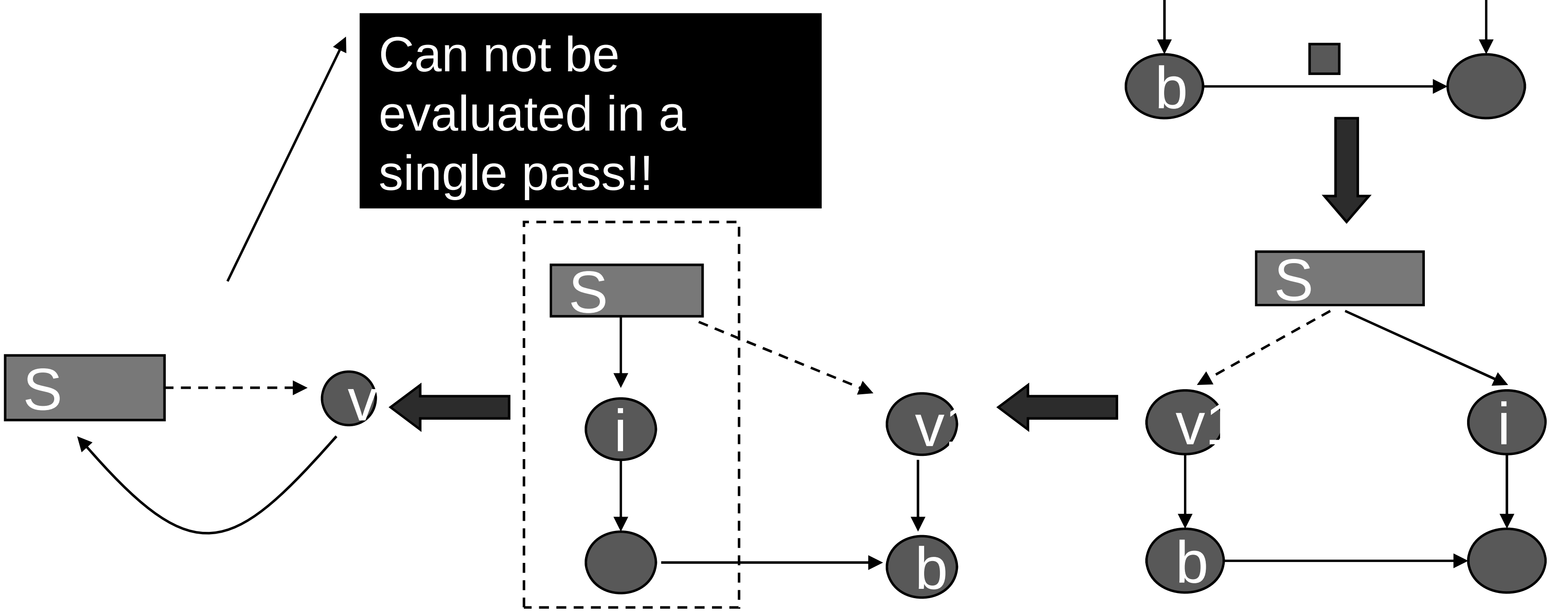
Example:

```
let $p: =  stream/pixel/x
 for $i in $p
   where $i <= max($p)
   return $i
```

The Overall Procedure

```
let $b = count(stream/pixel[x>0])
  for $i in stream/pixel
     return $i/x idvi $b
```



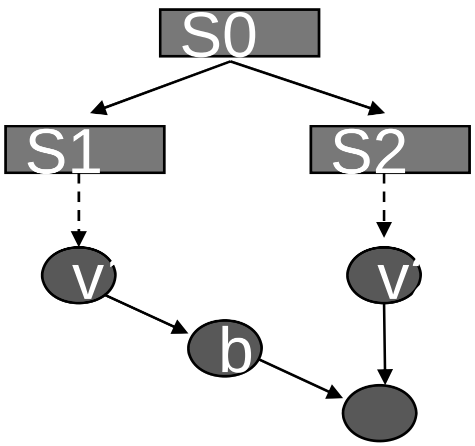


Low-level Transformations

- Use GNL as intermediate representation
  - GNL is similar to nested loops in Java
  - Enable efficient code generation for reductions
  - Enable transformation of recursive function into iterative operation
- From SDFG to GNL
  - Generate a GNL for each sequence node associated with XPath expression
  - Move aggregation into GNL using aggregation rewrite and recursion analysis

GNL Example

```
let $b = count(stream/pixel[x>0])
     return sum(stream/pixel/y) idvi $b
```



for i_1, stream/pixel, --

[for i_2, /x, /x > 0

  [$v_1 = v_1 + 1$

  for i_3, /y, --

  [$v_2 = v_2 + i_3$;

$b = v_1$ return $b \div v_2$

Facilitate code generation for any desired platform

Low-Level Transformations

- Recursive Analysis

  - extract commutative and associative operations from recursive functions

- Aggregation Rewirte

  - perform function inlining

  - transform built-in and user-defined aggregate into iterative operations

Code Generation

- Using SAX XML stream parser

  - XML document is parsed as stream of events

  <x> 5 </x>: startelement <x>, content 5, endelement <x>

  - Event-Driven: Need to generate code to handle each event

- Using Java JDK

  -Our compiler generates Java source code

Code Generation: Example

```
for i1, stream/pixel, --
  [ for i2, /x, /x > 0
    [ v1 = v1 + 1
    for i3, /y, --
    [ v2 = v2 + i3 ;
b = v1 return b ÷ v2
```

```
foreach startElement (e_i) {
   switch( e_i.node)
         x: buffer.add(x)
         y: buffer.add(y)
}

foreach endElement (e_i) {
   switch( e_i.node)
         x: if (buffer.dispatch(x) >0)
            v1 = v1 + 1
         y: v2 = v2+ buffer.dispatch(y)
         root: { b = v1;
            return b / v2
         }
}
```

startElement: Insert each referred element into buffer

endElement: Process each element in the buffer, dispatch the buffer