Current Research

SSSG Seminar
Sven Stork
17th November 2008

Outline

- Motivation
- Petri Nets
- Dataflow Architectures
- My Idea
- Summary

Motivation

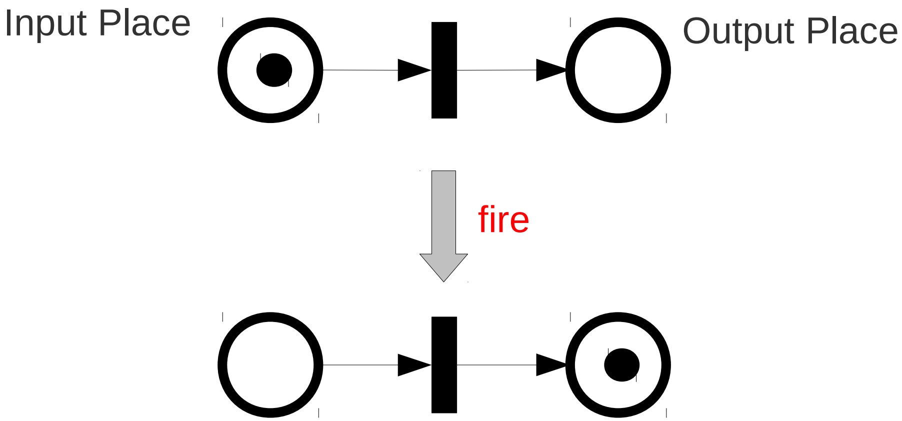
- Writing concurrent applications is **hard** and **error prone**
  - User needs to deal with concurrency at low level abstraction (e.g. threads, locks, ...)
  - Current programming languages were designed
  - User needs to consider all possible executions paths/combinations
- **Question**
  Is there an easier way to write a concurrent programs without dealing of such drawbacks ?

Outline

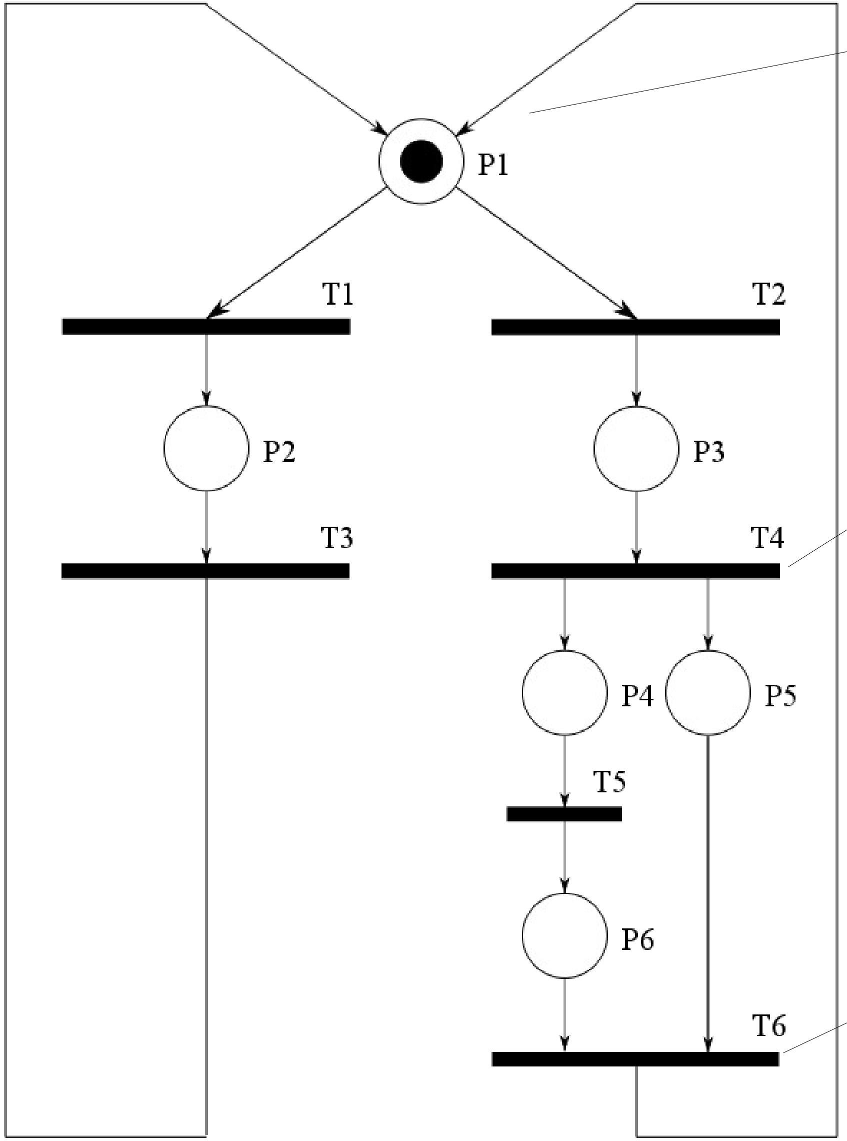
- Motivation
- Petri Nets
- Dataflow Architectures
- My Idea
- Summary

Petri Nets

- A Petri net is a directed bipartite graph
- A Petri net consists of :
  - Tokens ●
  - Places ○
  - Transitions ▮



Petri Nets



Non-Determinism

Trigger multiple actions

Wait for all previous actions to finish

http://de.wikipedia.org/w/index.php?title=Bild:Petrinetz.svg&filetimestamp=20080911233209

Outline

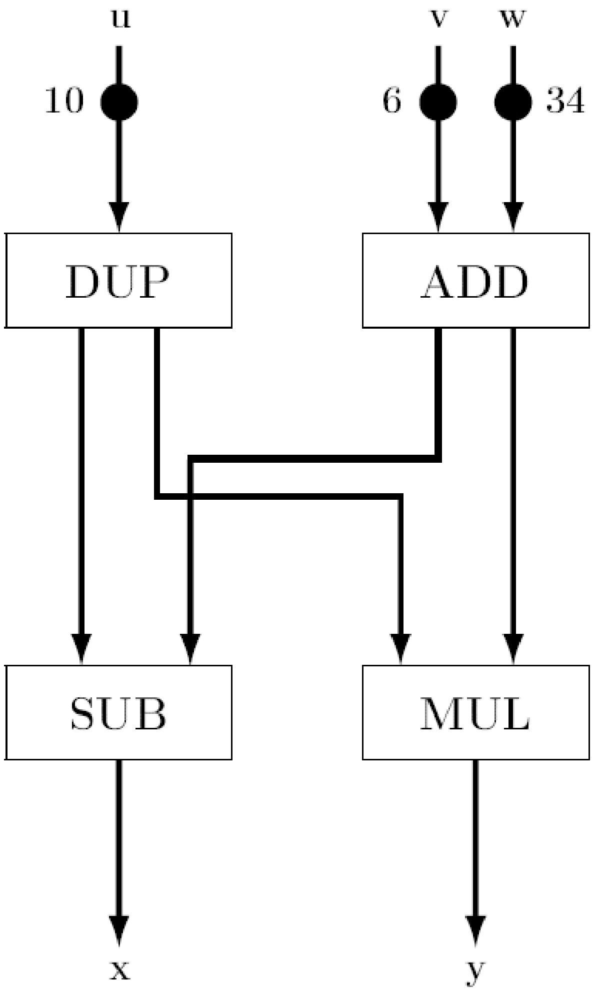
- Motivation
- Petri Nets
- Dataflow Architectures
- My Idea
- Summary

Dataflow Architecture

- ~ deterministic Petri Net
- Express data dependencies between statements
  - Program is data dependency graph
- Execution of functions depends on availability of their input data
- At every moment of time all possible execution units are know
  - **MAXIMUM parallelism**

Dataflow Architecture

```
Input: u, v, w;
  x = u - (v + w);
  y = u * (v + w);
Output: x, y;
```



http://de.wikipedia.org/w/index.php?title=Bild:Datenflussgraph.jpg&filetimestamp=20080425160503

Dataflow

Pro

- Expresses all the parallelism in a program
- No shared data
  - Data is consumed and produced
  - No synchronisation required

Contra

- Hard to write programs
- Inefficient
  - Always creating and consuming data is expensive

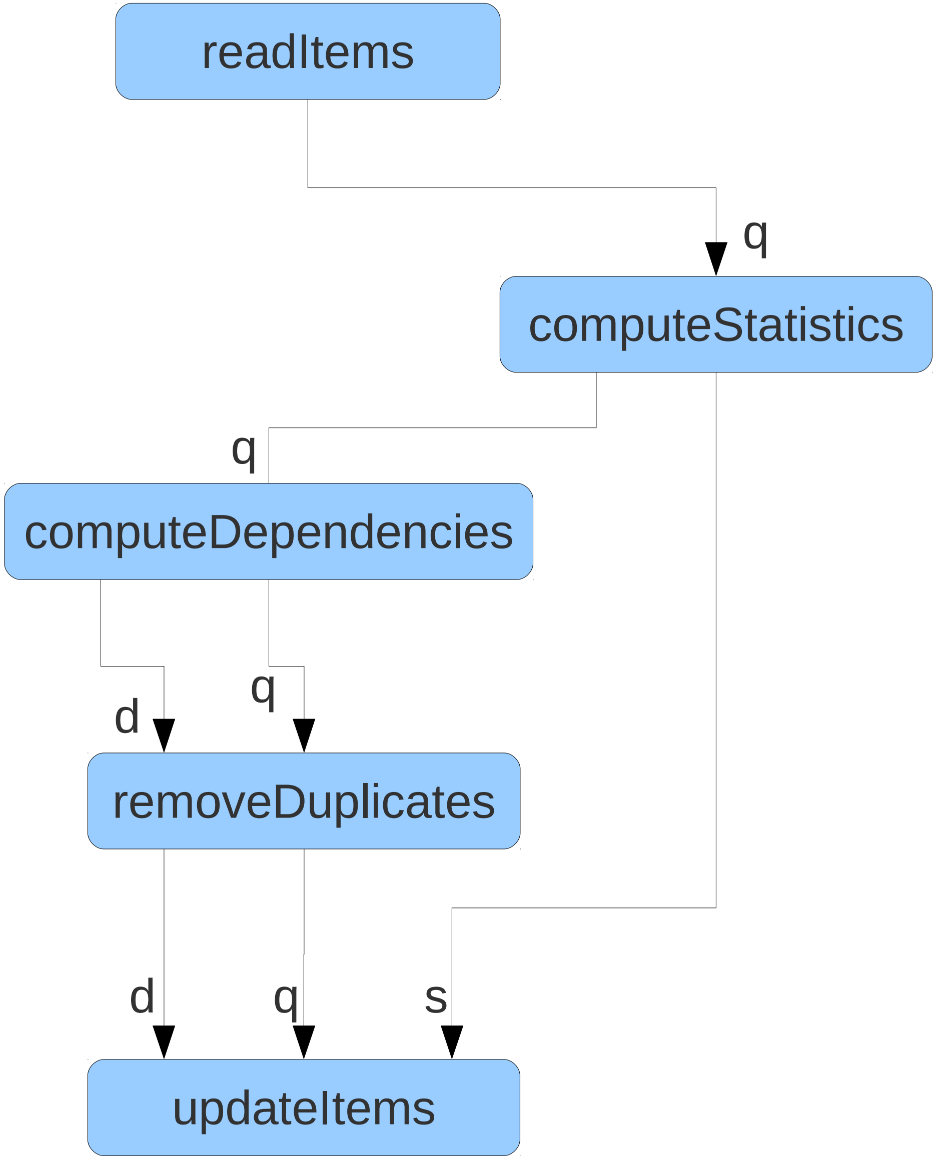
Outline

- Motivation
- Petri Nets
- Dataflow Architectures
- My Idea
- Summary

My Idea

```
void readItems(Queue q) { ... }

void updateItems(Queue q,
                 Deps d,
                 Stats s) { ... }

void removeDuplicates(Queue q,
                      Deps d){ ... }

Deps computeDependencies(Queue q) { ... }

Stats computeStatistics(Queue q) { ... }

void main () {
   Queue q = new Queue();

   readItems(q);
   Stats s = computeStatistics(q);
   Deps s  = computeDependencies(q);
   removeDuplicates(q, d);
   updateItems(q, s d);
}
```

My Idea


readItems
q
computeStatistics
q
computeDependencies
d
q
removeDuplicates
d
q
s
updateItems

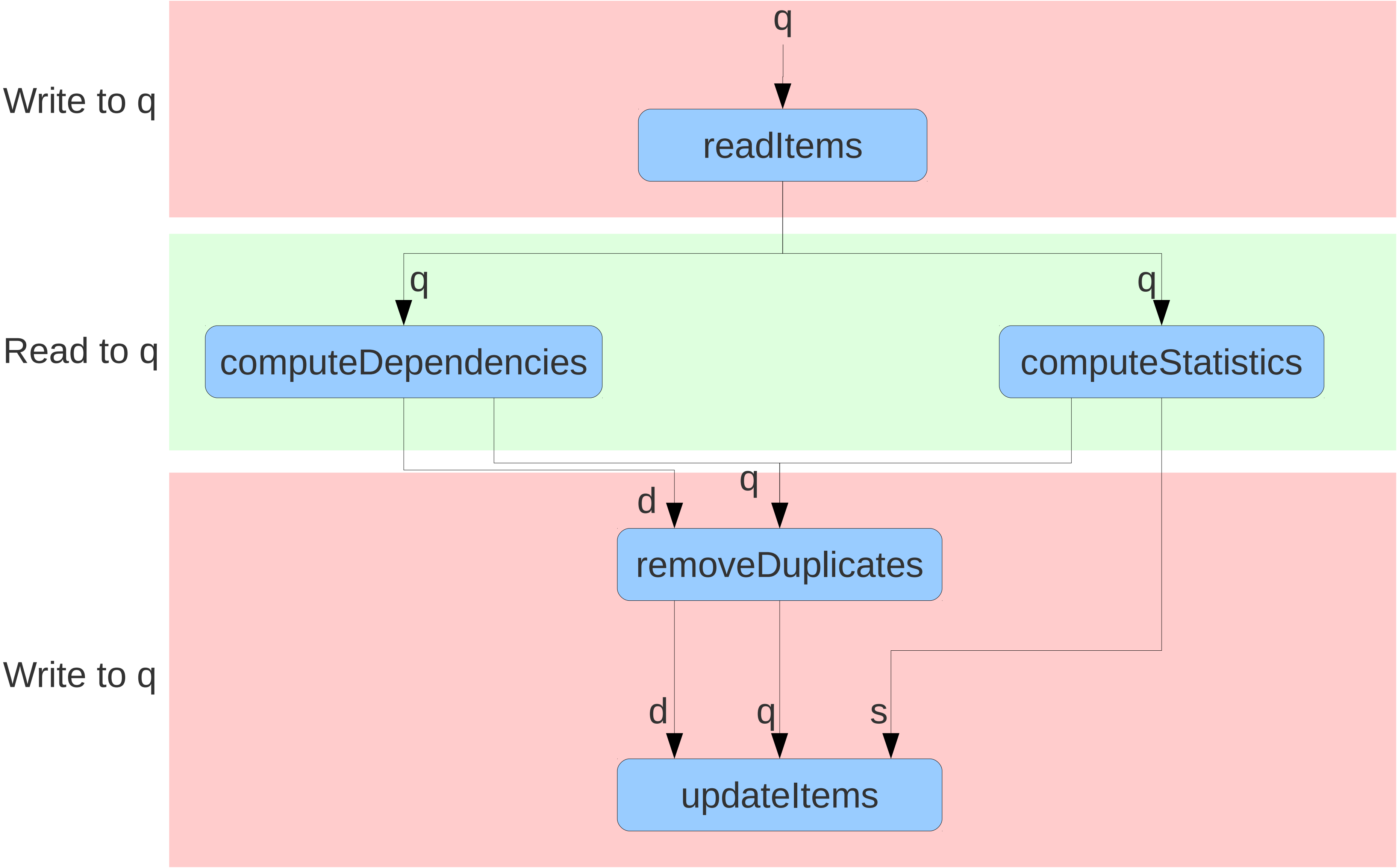
My Idea

- **Requirements of operations on data**:
- Access
  - Read
    - Function only read the specified object
  - Write
    - Function read+write the specified object

My Idea

```
void readItems(@Write Queue q) { ... }

void updateItems(@Write Queue q,
                 @Read Deps d,
                 @Read Stats s) { ... }

void removeDuplicates(@Write Queue q,
                      @Read Deps d){ ... }

Deps computeDependencies(@Read Queue q) { ... }

Stats computeStatistics(@Read Queue q) { ... }

void main () {
   Queue q = new Queue();

   readItems(q);
   Stats s = computeStatistics(q);
   Deps s  = computeDependencies(q);
   removeDuplicates(q, d);
   updateItems(q, s d);
}
```

My Idea


Write to q
q
readItems
Read to q
q
q
computeDependencies
computeStatistics
Write to q
d
q
removeDuplicates
d
q
s
updateItems

My Idea

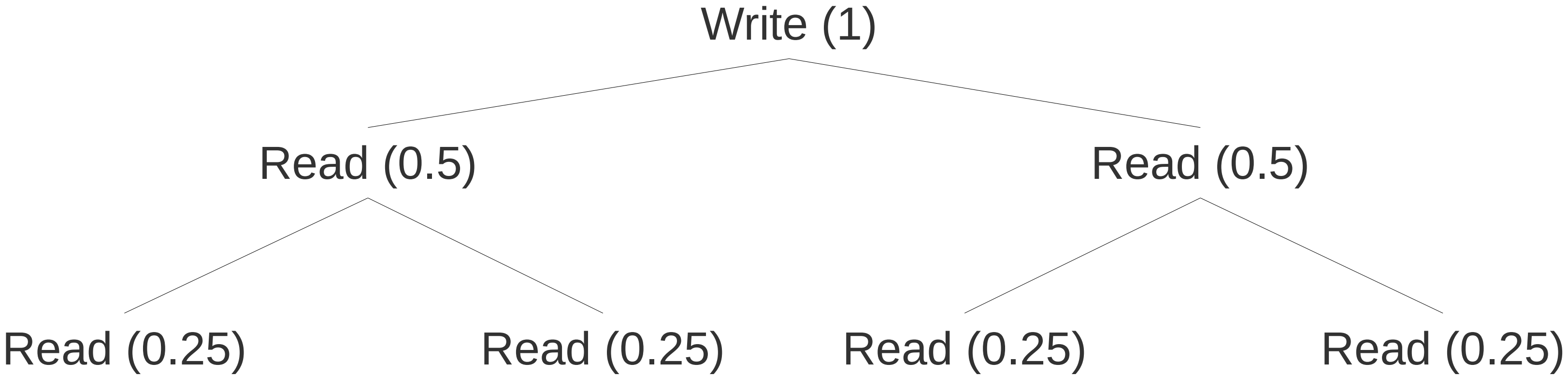
- Ordering
  - Lexical defined

```
...
removeDuplicates(q, d);
updateItems(q, s d);
```

$\overset{?}{=}$

```
...
updateItems(q, s d);
removeDuplicates(q, d);
```

  - By permission splitting/joining



Write (1)

Read (0.5)

Read (0.5)

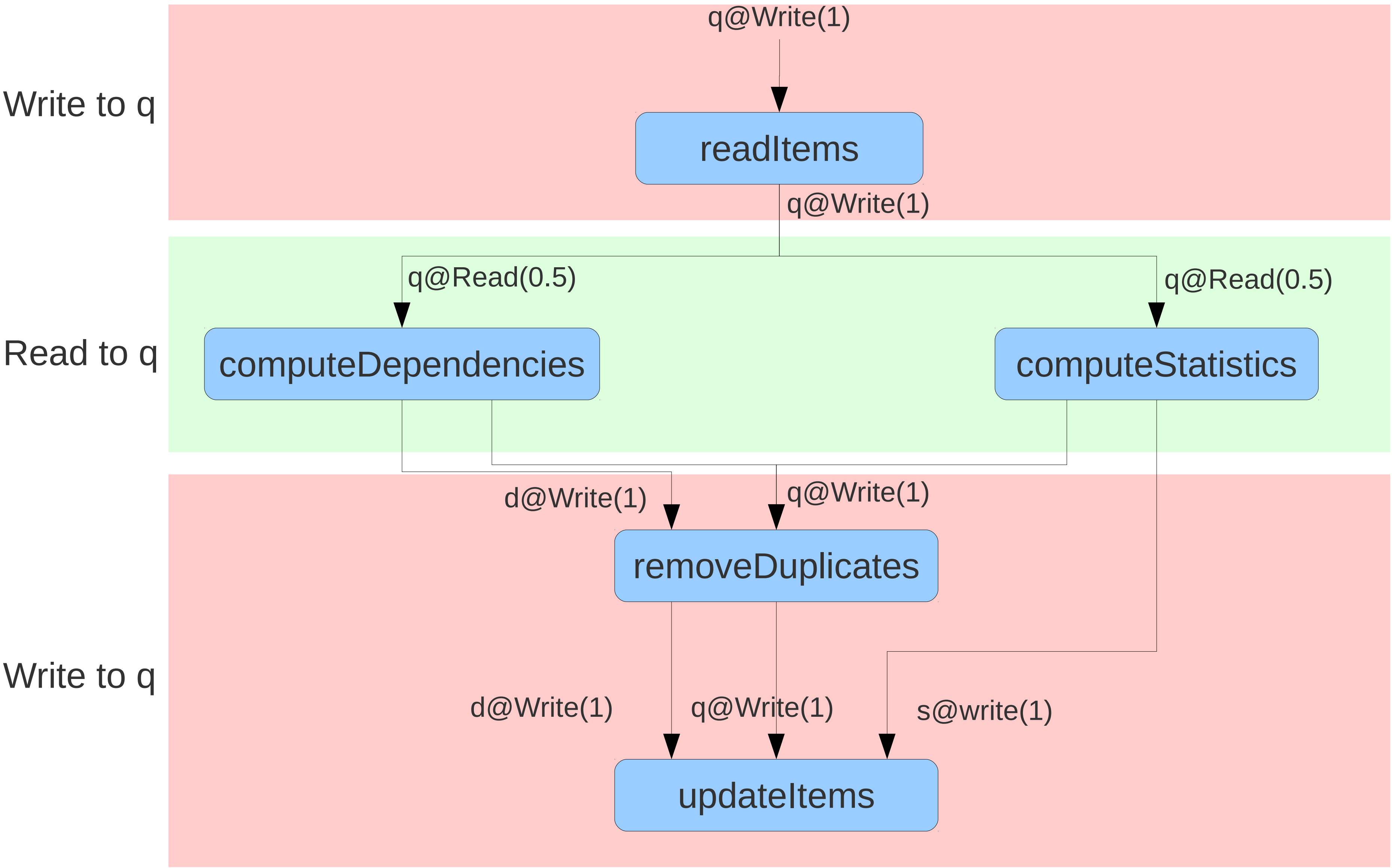
Read (0.25)

Read (0.25)

Read (0.25)

Read (0.25)

My Idea



Write to q

q@Write(1)

readItems

q@Write(1)

Read to q

q@Read(0.5)

q@Read(0.5)

computeDependencies

computeStatistics

Write to q

d@Write(1)

q@Write(1)

removeDuplicates

d@Write(1)

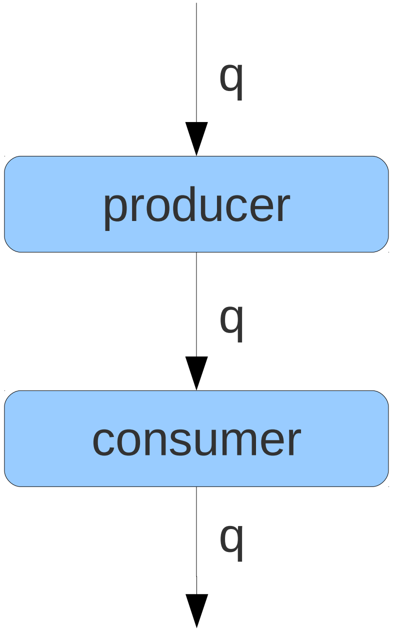
q@Write(1)

s@write(1)

updateItems

My Idea

```
void producer(@Write Queue q) {
   while (condition1) {
      ...
   }
}

void consumer(@Write Queue q) {
   while (condition2) {
      ...
   }
}

void main () {
   Queue q = new Queue();

   producer(q);
   consumer(q);
}
```



My Idea

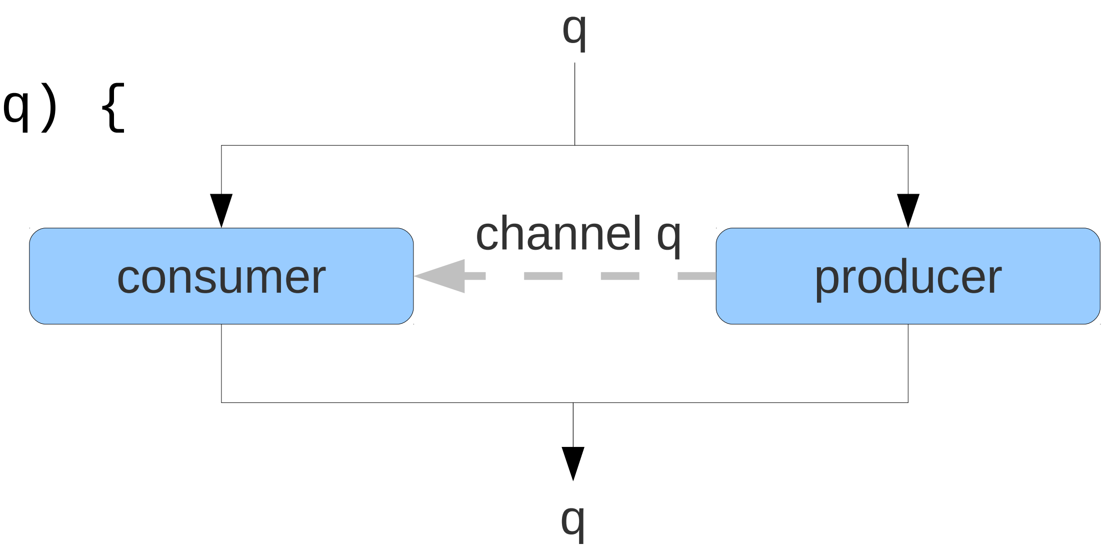
- **Problem:**
  Write permissions force strict sequentially

- What if the consumer could start to work on the already produced items ?
  - Pipelining
- What if the producer never stops ?
  - Example: contiguous reads data from a sensor

- Solution ==> Introduce @Shared
  (allow only “protected” access)

My Idea

```
void producer(@Shared Queue q) {
   while (condition1) {
      ...
   }
}

void consumer(@Shared Queue q) {
   while (condition2) {
      ...
   }
}

void main () {
   Queue q = new Queue();

   producer(q);
   consumer(q);
}
```



My Ideas

- Access-Matrix

	Access	
	Reading	Writing
Single	ReadOnly (RO)	Write (W)
Concurrent	ReadOnly (RO)	Shared (S)

My Idea

- Granularity
  - Data
    - Collection of Objects
    - Objects
    - Partitions of Objects
    - (Every Field of an Object)
  - Code
    - Methods
    - Blocks
    - Statements

My Idea

- Interesting Questions
  - What to do at runtime/compile time ?
  - How to avoid deadlocks ?
  - How and when to select granularity ?
- Runtime System
  - How to represent dependencies ?
  - How to deal with blocking operations ?
  - How to efficiently implement synch. tasks ?

Summary

- Dataflow Architectures are nice for representing concurrency
- Use Permissions to data to derive dataflow graph from a program
    - Permissions = tokens
- A lot of open questions ...