The Java Concurrency API and Deadlock Prevention in a RETE Rules Engine to Implement a Pricing Service

Elie Levy

elie.levy@zilonis.org

BOF-7793

Goal of the Talk

What You Will Learn

How the Java platform can be used to write a Concurrent RETE Forward Production System:

The Zilonis Rules Engine

Agenda

Pricing Service in Retail

Rules Engine (RETE)

Concurrency in the Rules Engine

Deadlock Prevention

Other Optimizations

Demo!

Agenda

Pricing Service in Retail

Rules Engine (RETE)

Concurrency in the Rules Engine

Deadlock Prevention

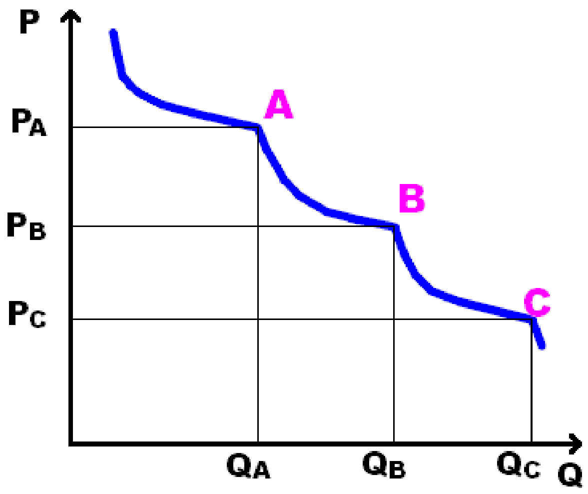
Other Optimizations

Demo!

Pricing Service in Retail

It is more complicated than what it seems at first

- Cost Plus vs. List Minus
- Marketing Campaigns
- Bulk Pricing
- Different Providers/Vendors
- Zone, Geo Location
- Price Discrimination/Contracts
- Competition



Agenda

Pricing Service in Retail

Rules Engine (RETE)

Concurrency in the Rules Engine

Deadlock Prevention

Other Optimizations

Demo!

Rules Engine

The value in a system like a pricing engine

- Understandability:
  - Declarative, well defined rules
  - Easy to read and understand
  - Business engagement early on the process (KPLM)
- We can get a clear explanation of why a result was given
- Agile Maintainability
- Time to market

Rules Engine

The Structure

- Working Memory
  - A set of Facts: Working Memory Elements (WME)
- Production Memory
  - A set of Rules: Productions

Rules Engine

Working Memory

- Internal Representation is in 3-Tuples: Triples
- Complex Structures can be mapped to Triples

```
(Order (sku 3155123) (quantity 2) (channel web))
           W1:(1 clazz Order)
           W2:(1 sku 3155123)
           W3:(1 quantity 2)
           W4:(1 channel web)
```

- RDF uses the same approach (SPO) in the Semantic Web

Rules Engine

Production Memory

```
(Name-of-Production
   LeftHandSide     /* one or more conditions */
=>
   RightHandSide    /* one or more actions */
```

Rules Engine

Production Memory

```
(defrule retailPricingRule
  (order (channel web)
         (sku ?skuId))
  (item (sku ?skuId)
        (retailPrice ?price))
=>
  (assert (methodResult (price ?price)
                        (method retailPrice))))
```

Rules Engine

Complexity of the Match Phase

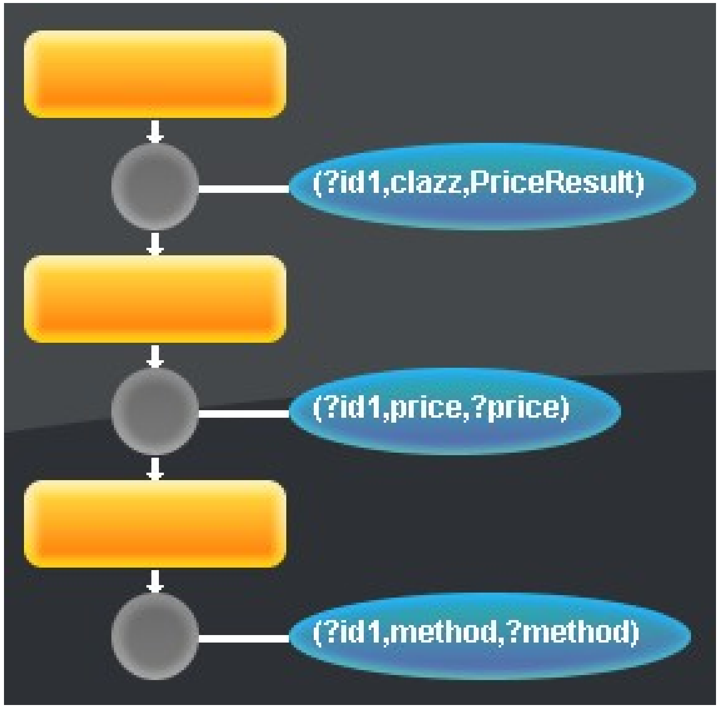
- Consider a production system of:
  - 1,000 rules with 3 conditions each
  - 1,000 Working Memory Elements (WME)
- Naive implementation:
  - Each production is matched against all tuples of size 3 from working memory (WM)
  - Over a trillion ($1{,}000 \times 1{,}000^3$) match operations per cycle
- Even specialized algorithms take 90% of time in this phase

RETE Algorithm

Version used in Zilonis and Soar

- Dataflow network to represent the conditions

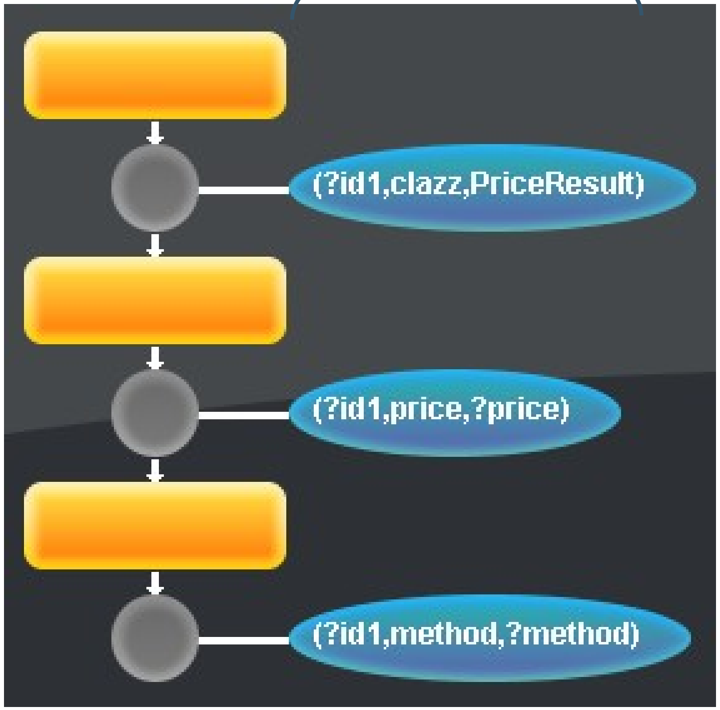
- The network has 2 parts:
  - Alpha Network
  - Beta Network



RETE Algorithm

The Alpha Network

Alpha Memory




```
(defrule PriceResultRule
    (?id clazz PriceResult)
    (?id price ?price)
    (?id method ?method)
=>
    (print “result: ?1 method ?2”
                        ?price ?method))
```

Results of C1

Results of C2

Results of C3

RETE Algorithm

The Beta Network

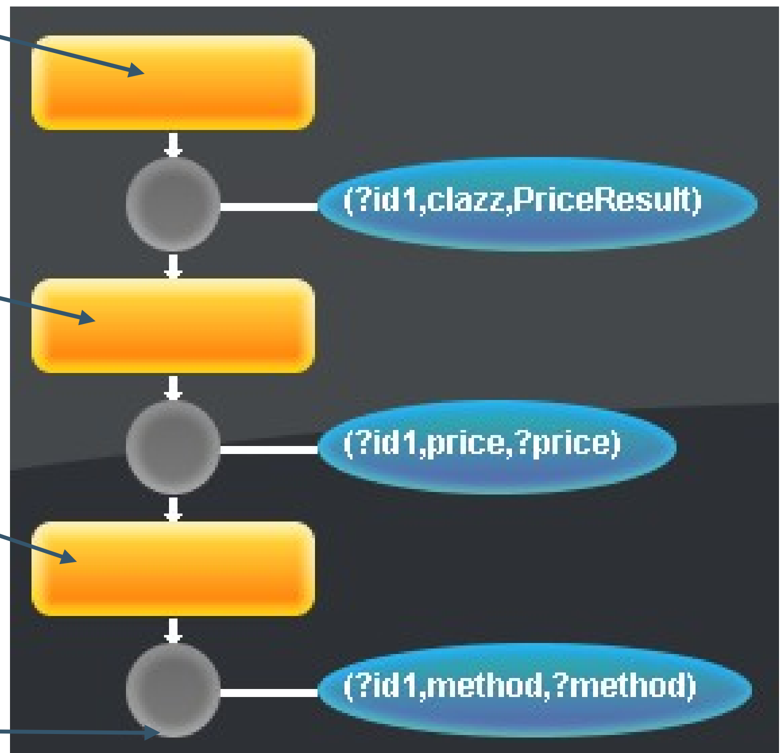
```
(PriceResultRule
    (?id clazz PriceResult)
    (?id price ?price)
    (?id method ?method)
=>
    RHS
```

Dummy Beta Memory

Results of C1

Results of C1 and C2

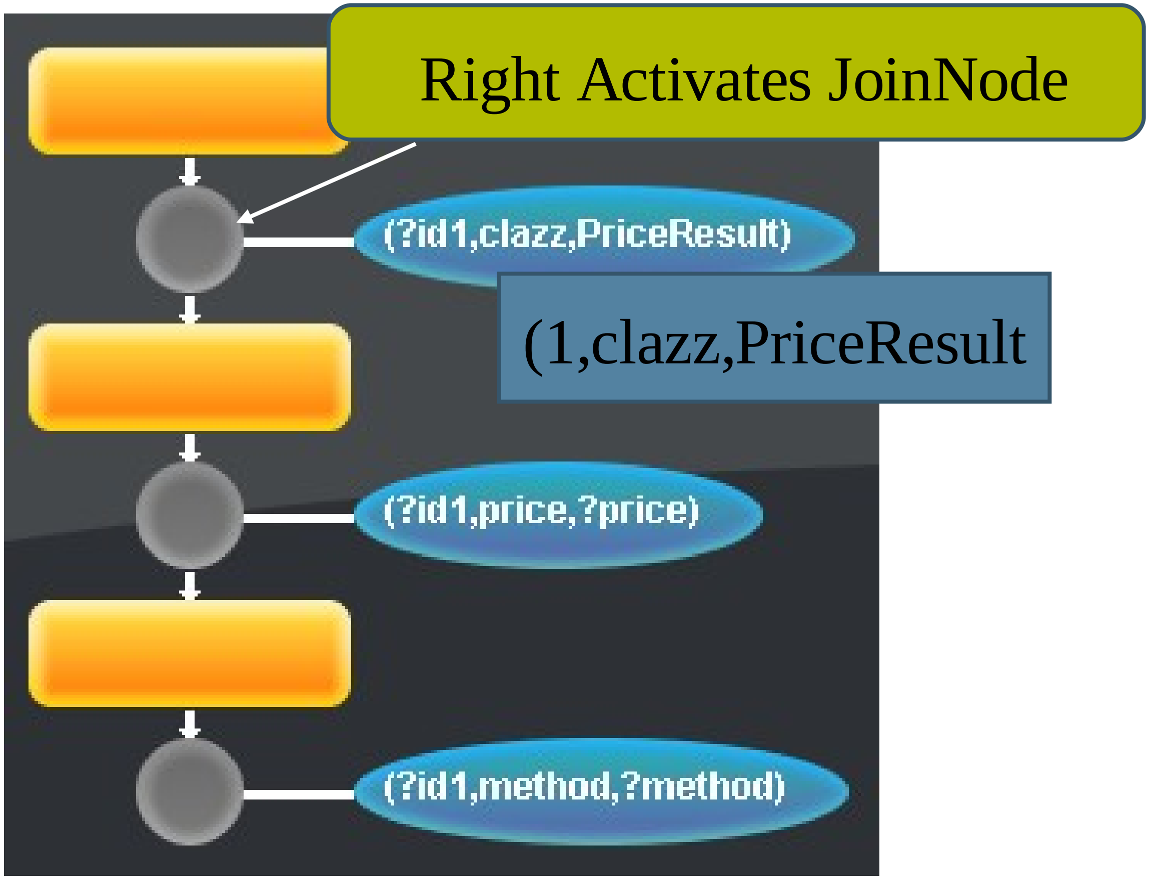
Results of C1 and C2 and C3



RETE Algorithm

Putting it all together

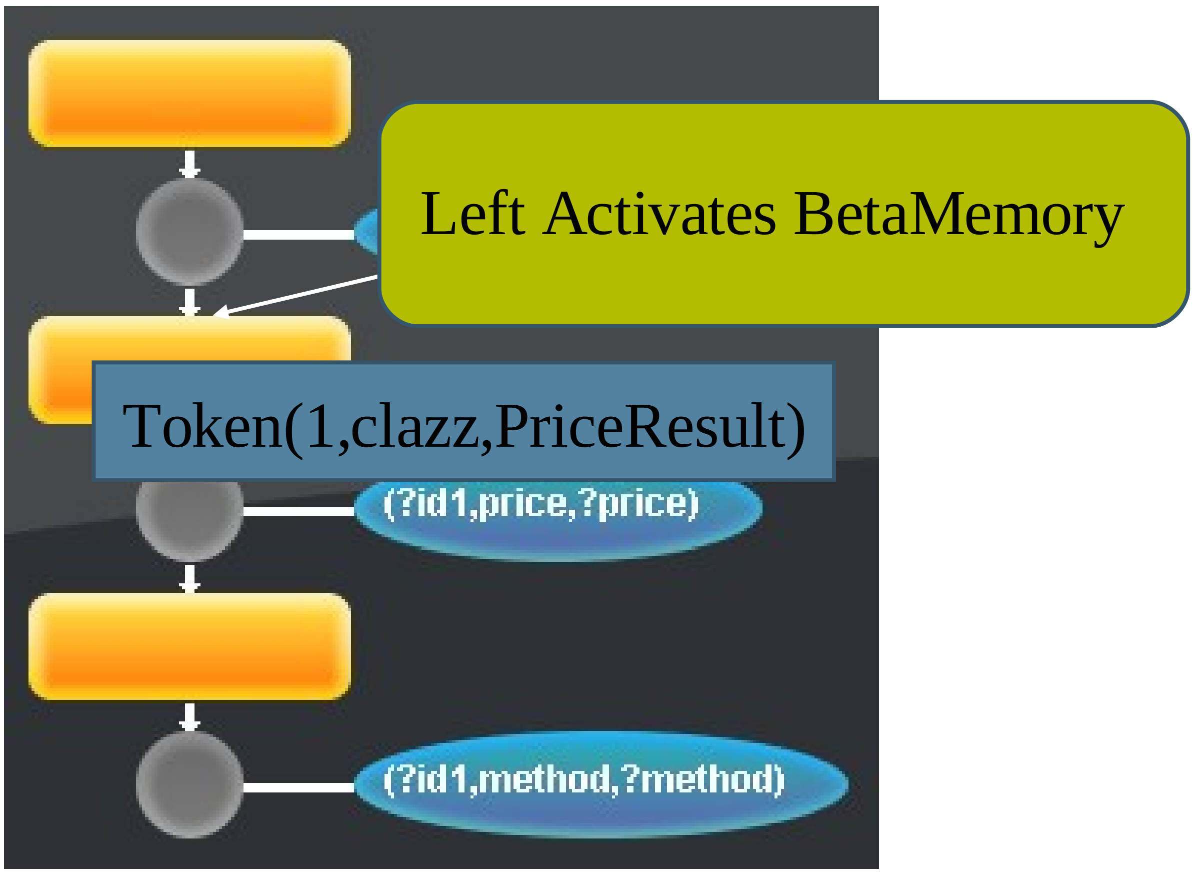
Assert: (1,clazz,PriceResult)



RETE Algorithm

Putting it all together

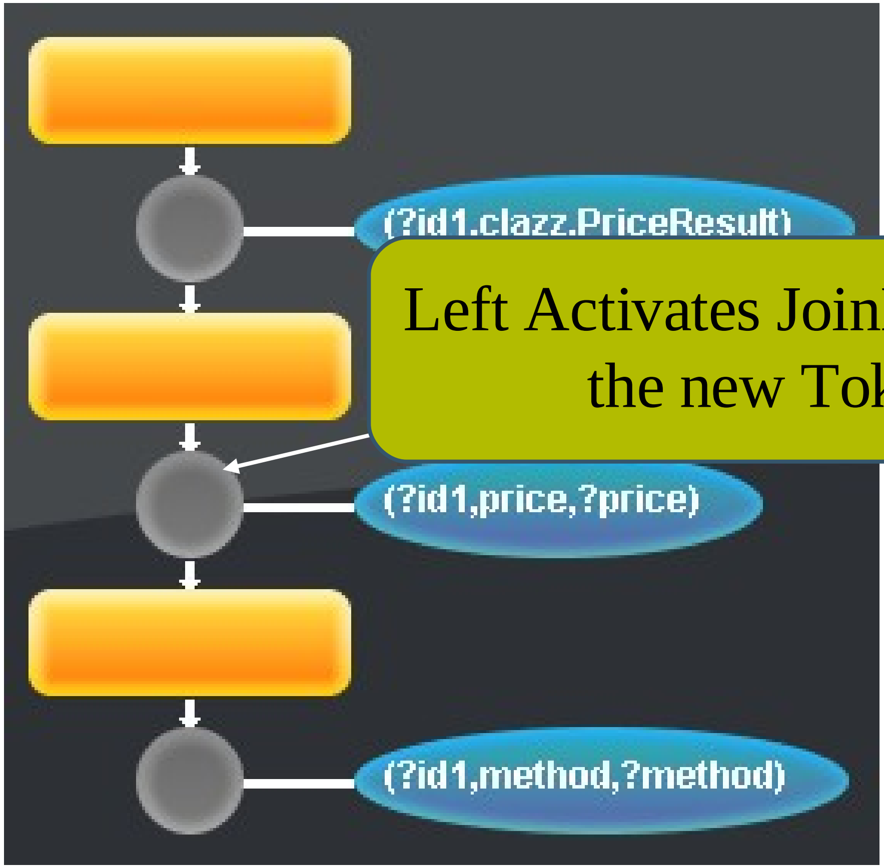
Assert: (1,clazz,PriceResult)



RETE Algorithm

Putting it all together

Assert: (1,clazz,PriceResult)

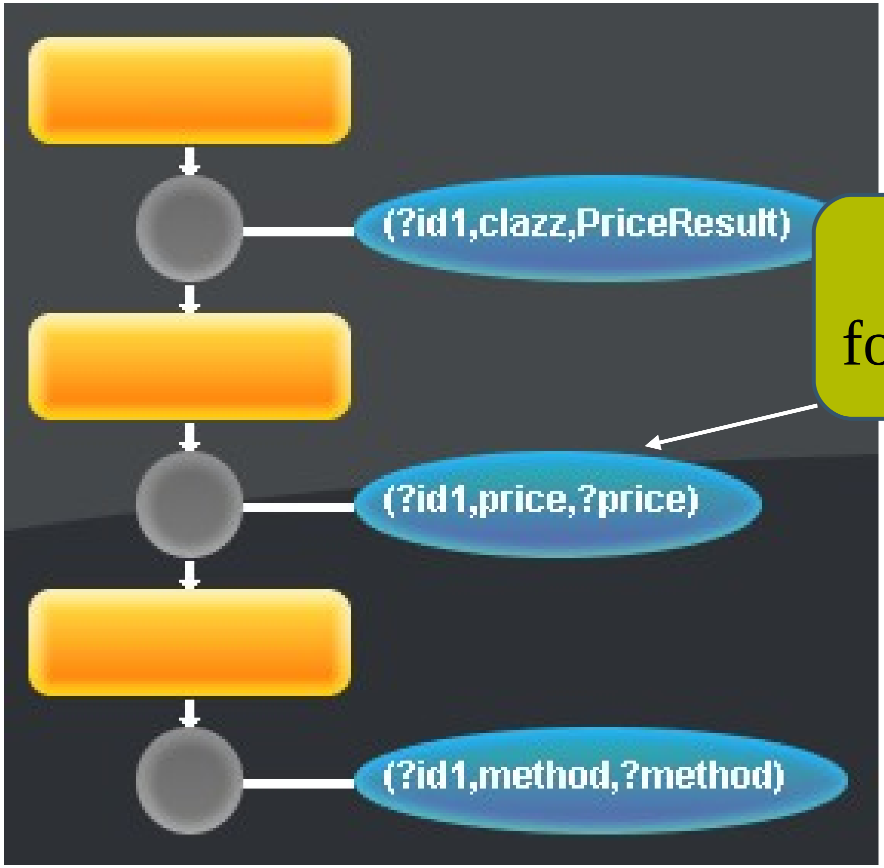


Left Activates JoinNode with the new Token

RETE Algorithm

Putting it all together

Assert: (1,clazz,PriceResult)



Queries AlphaMemory for WMEs and Tries to Match

Demo: Let’s see it in the Analysis Tool

RETE Algorithm

Main advantages over the naive algorithm

- State-saving reduces calculation time
  - Changes in WM: Are saved in Alpha and Beta Memories
  - No need to recalculate all the different possibilities

- Sharing of nodes
  - Alpha Memory
    - when two or more productions have similar conditions
  - Beta Memory
    - when the first few conditions of two or more productions are similar

RETE Algorithm

Why it is not widely used in E-Commerce?

- The traditional algorithm is not Thread-Safe
- Some of the available implementations are aware of the multithreaded challenges
  - They lock the entire engine, similar to what java.util.Hashtable does

RETE Rules Engine

Why it is not widely used in E-Commerce?

- Option #1: Create a RETE instance per Thread
  - When tried to load 20,000 products (close to 250k WMEs) the engine died.
  - Can not even dream creating an instance of the engine per Thread, just one does not work
- Option #2: One single instance, serial access to it
  - Doesn't take advantage of the multiprocessor/multi-core architectures
  - Does not scale to the throughput needs

Agenda

Pricing Service in Retail

Rules Engine (RETE)

Concurrency in the Rules Engine

Deadlock Prevention

Other Optimizations

Demo!

Thread Safety

- Managing Access to Shared Mutable State

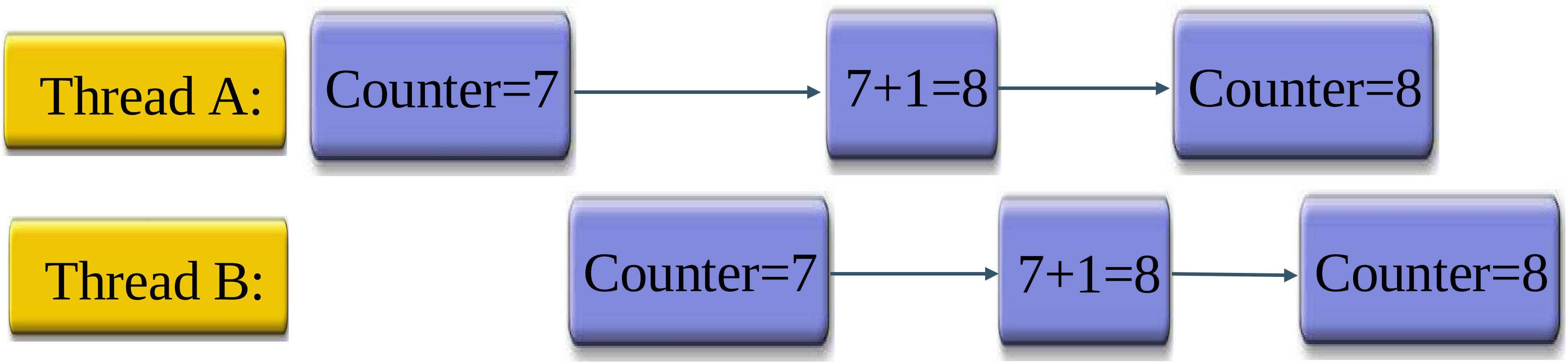
- Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access using Synchronization
  - synchronized
  - volatile variables
  - explicit locks
  - atomic variables

Concurrency Challenges

Atomicity

```
// Susceptible to lost updates
private long counter=0;

public void execute() {
        counter++;
        System.out.println(counter);
}
```



Concurrency Challenges

Atomicity: the java.util.concurrent.atomic API

```
public class SafeCounter {

    private final AtomicLong counter = new
AtomicLong(0);
    public long getCounter() { return count.get(); }

    public void execute() {
        counter.incrementAndGet();
    }
}
```

Concurrency Challenges

Race Conditions: The need for Locks

```
public class NotSafeTransfer {

    private final AtomicLong checkingBalance =
                                 new AtomicLong(0);
    private final AtomicLong savingsBalance =
                                 new AtomicLong(0);

    public void transfer100() {
        checkingBalance.addAndGet(100);
        savingsBalance.addAndGet(-100);
    }

}
```

Concurrency Challenges

Intrinsic Locks: enforcing atomicity

```
public class SafeTransfer {

    private long checkingBalance = 0;
    private long savingsBalance = 0;

    public synchronized void transfer100() {
        checkingBalance+=100;
        savingsBalance-=100;
    }
}
```

Concurrency Challenges

- Modern Compiler and Processor:
  - Speculative Execution
  - Instruction Scheduling
  - Register Allocation
  - Common Sub-expression Elimination
  - Redundant Read Elimination
- Multiprocessor Systems
  - Each processor has its own cache

} Visibility

Understanding Visibility and the JMM

Without the proper synchronization

```
public class Unpredictable {
    private static boolean ready;
    private static int number;
    private static class CheckReady extends Thread {
        public void run() {
                while (!ready)
                        Thread.yield();
                System.out.println(number);
        }
    }
    public static void main(String[] arg) {
        new CheckReady().start();
        number = 42;
        ready = true;
    }
}
```

RETE Rules Engine

Principles to Implement a Concurrent Version

#1: Keep it simple, Keep it simple

Risks of Threads

Using Volatile

```
public class Predictable {

   private static volatile boolean ready;

   private static class CheckReady extends Thread {
      public void run() {
            while (!ready)
                  Thread.yield();
      }
   }
   public static void main(String[] arg) {
      new CheckReady().start();
      ready = true;
   }
}
```

RETE Rules Engine

Principles to Implement a Concurrent Version

#2: Don't reinvent the wheel:

- Create a RETE with the optimizations of one of the best algorithms available in the open source community (SOAR).

#3: Analyze alternatives to make it multithreaded:

- Reusing the RETE Dataflow Network across multiple Threads
- The Rules Developer should not worry about Session Ids

RETE Rules Engine

Principles to Implement a Concurrent Version

#4: **Use of Locks:**

- **Partitioning the way threads access our Alpha and Beta Memories**
- **With the right level of granularity**
- **Allowing multiple threads to operate in a thread-safe way**

RETE Rules Engine

Feasible Solution: Implement a Concurrent Version

Principles:

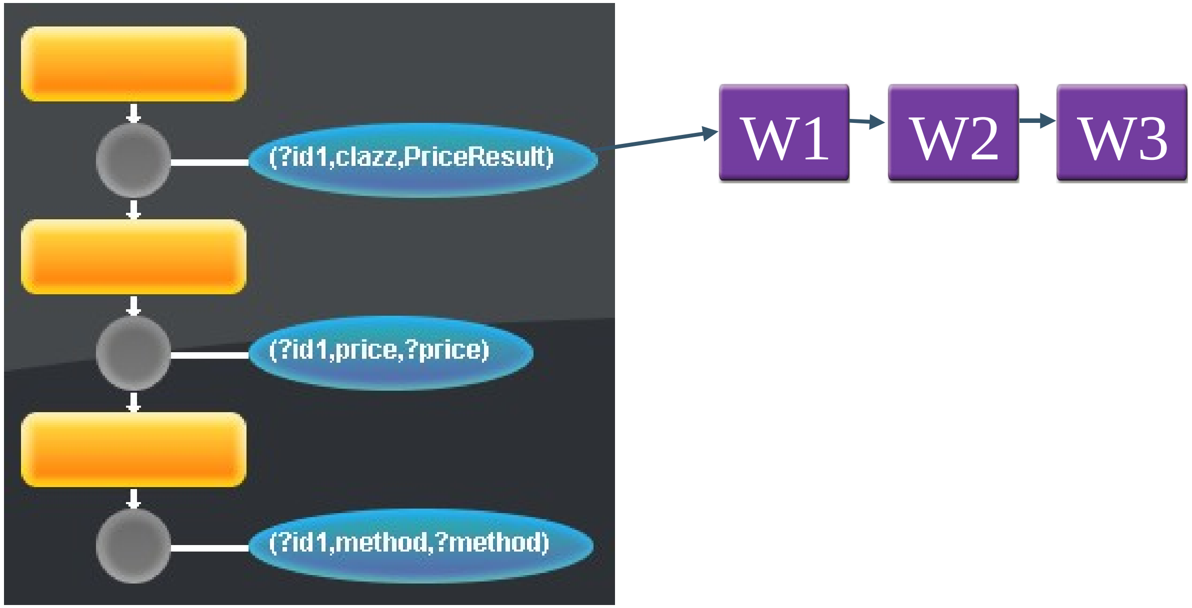
#5: **Follow a strict set of rules to obtain the Locks as a way to Preventing Deadlocks**

Zilonis RULES ENGINE

Multithreaded Rules Engine

Observation:

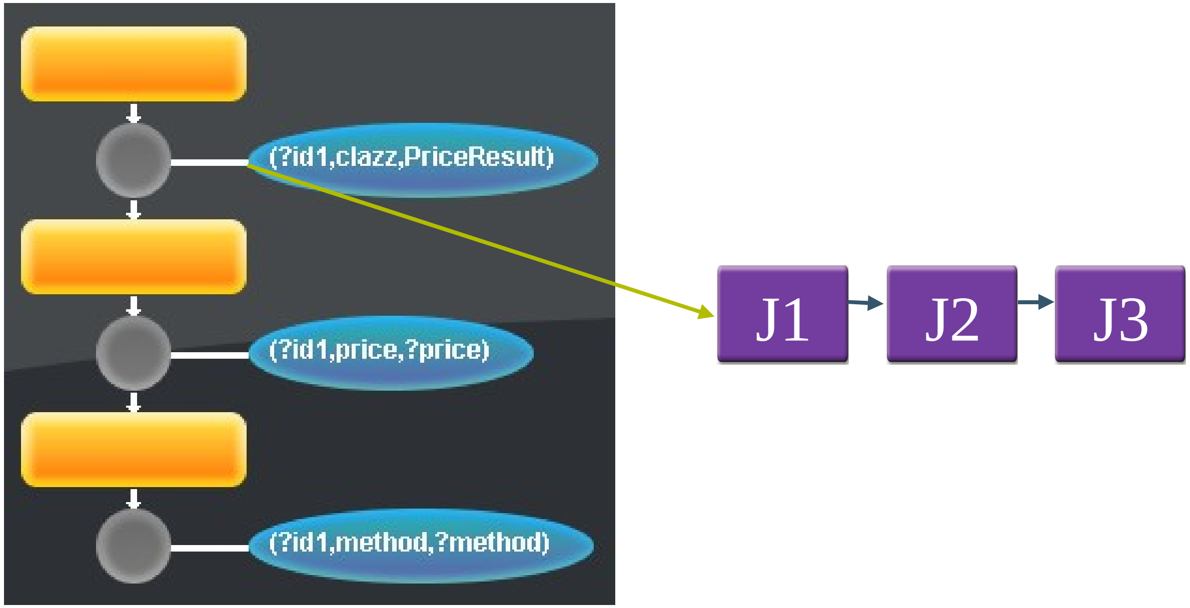
#1: Linked Lists are used to represent the set of WME in Alpha Nodes



Multithreaded Rules Engine

Observation:

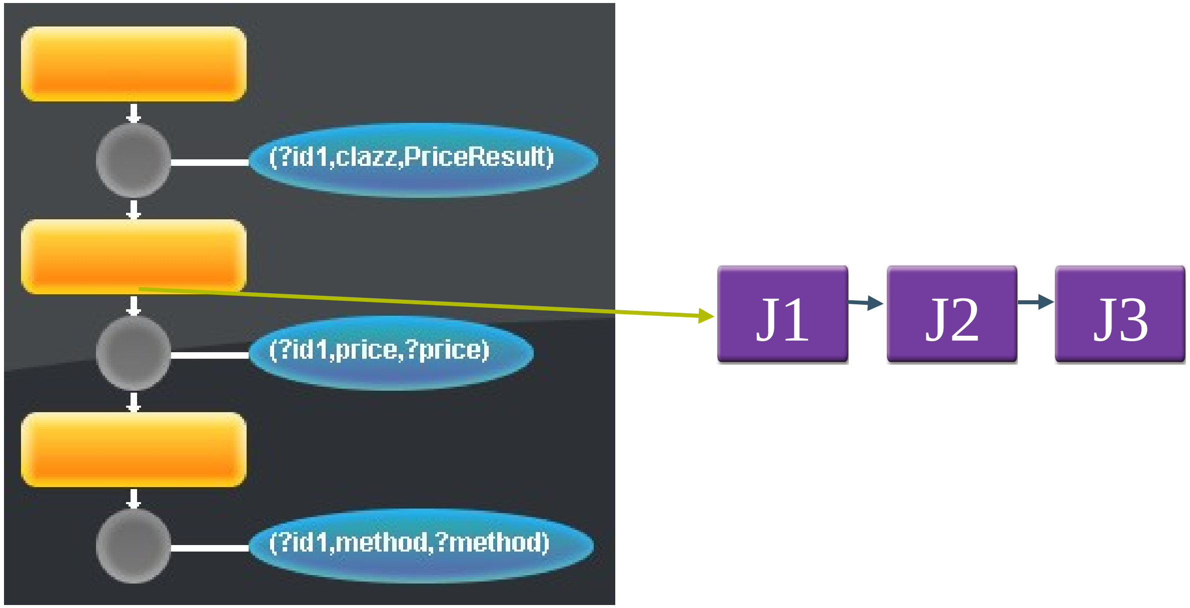
#2: Linked Lists are used to represent the set of Join Nodes in Alpha Nodes



Multithreaded Rules Engine

Observation:

#3: Linked List are used to represent the set of Join Nodes in Beta Memories

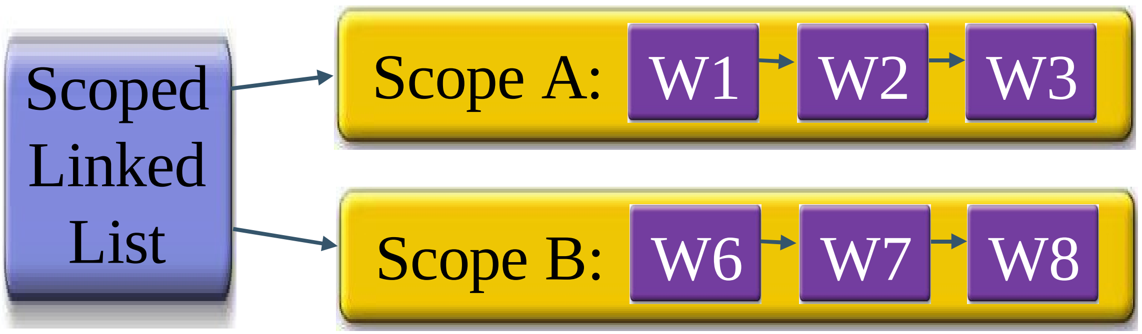


How the Concurrent Users would be using those Linked Lists?

- There is a part of them that is common, and does not change
  - That part can be safely shared
  - In out Pricing Engine example: 20,000 SKUs, and Pricing Rules
- Each User can have its own scope where only one user modifies the state at a time
  - Facts specific to that user
  - In our example: the information about the user, and the SKUs that wants to buy

Multithreaded Rules Engine

Building Block: The ScopedLinkedList

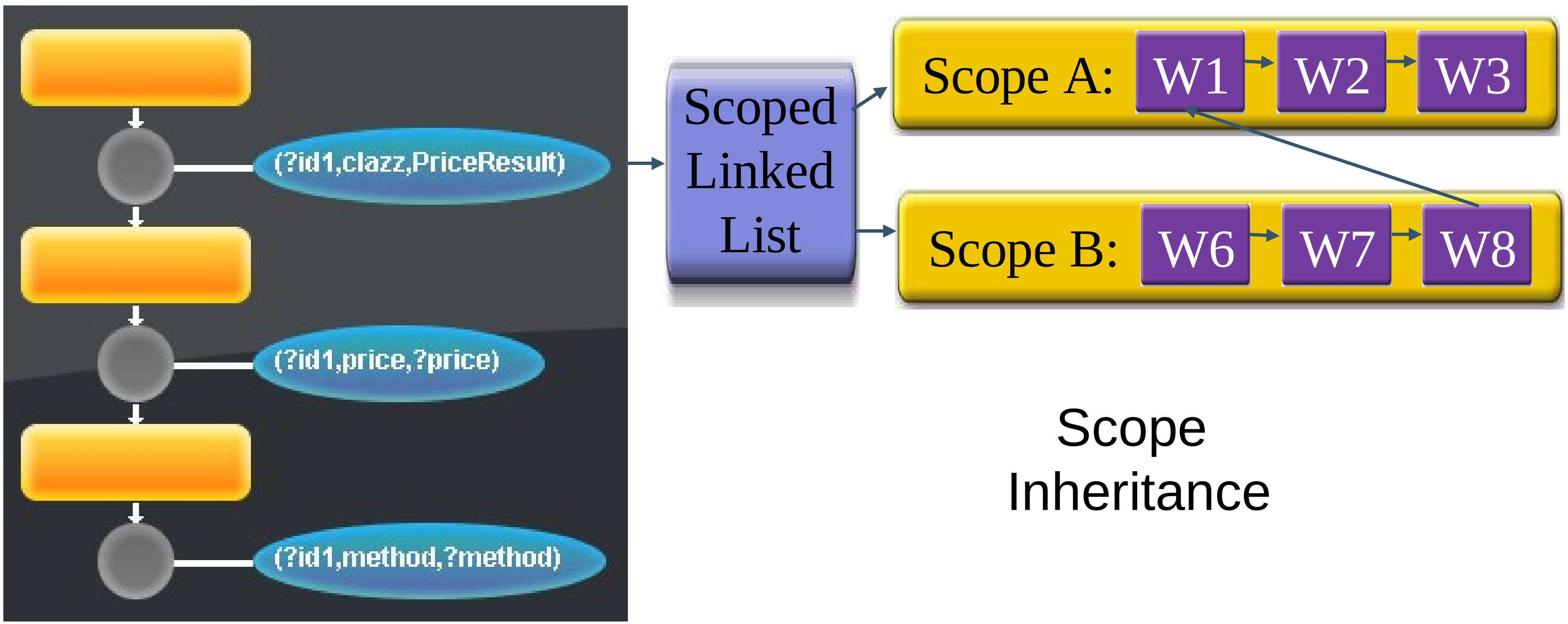


Lock on the Scope

A Thread can only hold the lock of a Scope at a time

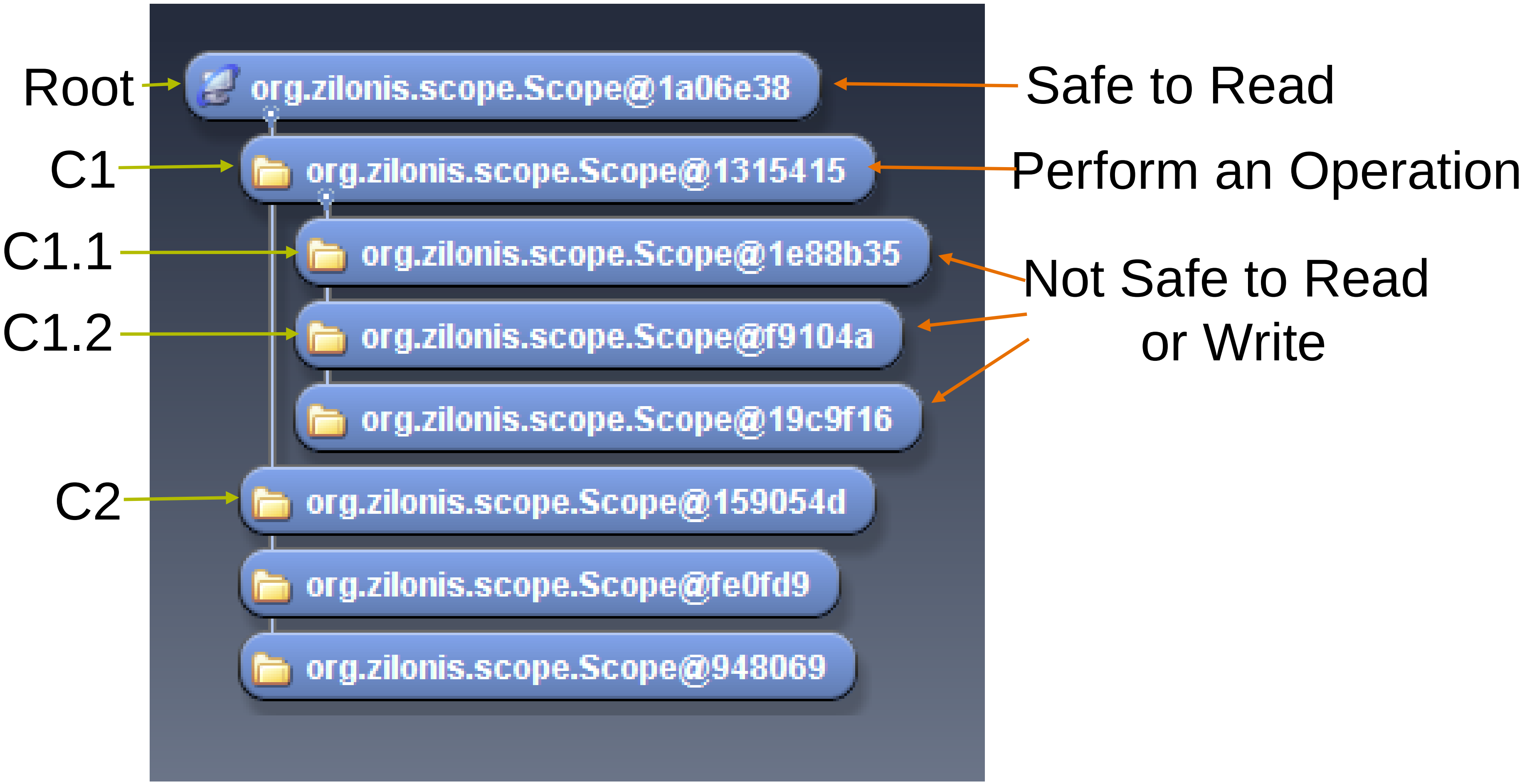
Multithreaded Rules Engine

Using the ScopedLinkedList



Scope Inheritance

Scopes and Concurrency



Scope class

```
public class Scope {
    private ReentrantReadWriteLock lock;
    private Scope parent;
    private LinkedList<Scope> children;

// The rest of the class implementation

}
```

Agenda

Pricing Service in Retail

Rules Engine (RETE)

Concurrency in the Rules Engine

Deadlock Prevention

Other Optimizations

Demo!

Deadlock

Necessary and Sufficient Conditions

- Mutual Exclusion Condition
- Hold and Wait Condition
- No-Preemptive Condition
- Circular Wait Condition

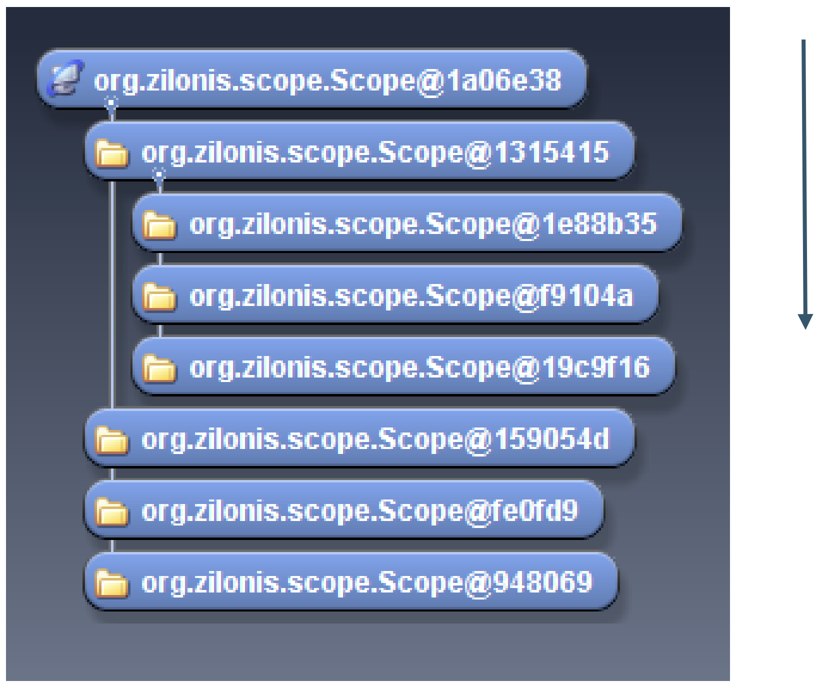
Deadlock Prevention

How can we safely Lock without the Deadlock fears

- Our Scopes are defined in a Tree structure (Graph without Cycles)
- Make sure we only get locks in one direction (Avoiding Circular Wait)

Deadlock Prevention

- Rule: To get the lock of a Scope, we need to get the Lock of the Parent Scope First



Zilonis RULES ENGINE

Scope class

```
public class Scope {
    . . .

    public void lock() {
        if (parent != null)
                parent.getReadLock();
        getWriteLock();
    }

    private void getWriteLock() {
        lock.writeLock().lock();
        for (Scope child : children)
                child.getWriteLock();
        }
}
```

Scope class

```
public class Scope {
    . . .

    private void getReadLock() {
        if (parent != null)
                parent.getReadLock();
        lock.readLock().lock();
}
```

ScopedLinkedList

```
public class ScopeLinkedList<Element> {
    private ConcurrentHashMap<Scope, SubList> map;

    public void add(Scope scope, Element element) {
        SubList subList = map.get(scope);
        if (subList == null) {
                sublist = new SubList(scope);
                map.put(scope, subList);
        }
        subList.add(element);
    }

    public Iterator<Element> iterator(Scope scope) {
        return new ScopedIterator(scope);
    }

}
```

Agenda

Pricing Service in Retail

Rules Engine (RETE)

Concurrency in the Rules Engine

Deadlock Prevention

Other optimizations

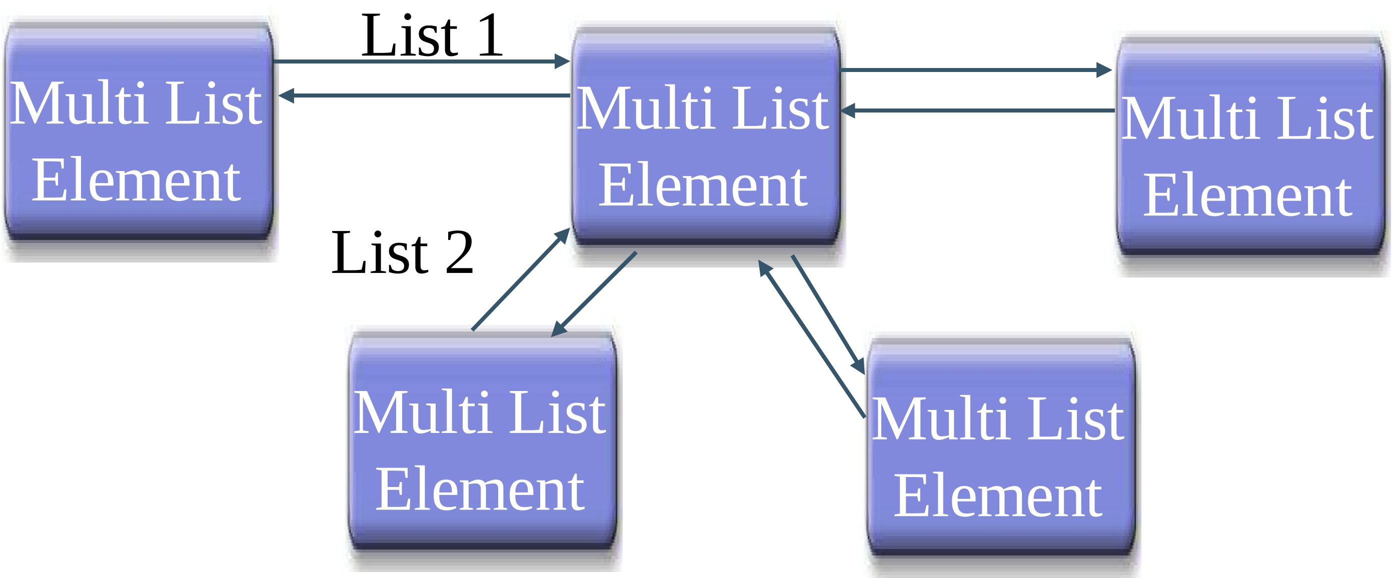
Demo!

Other Optimizations

- Implementation of the Entry in the linked list for fast removals
- Use of two global indexes:
  - Nodes in alpha memories
  - Nodes in beta memories

Implementation of Dobly Linked List

- The WME are in 2 lists
- The Tokens are in several lists also



MultiListElement

```
public class MultiListElement implements NextHolder,
                                  IMultiListElement {

    private IMultiListElement next[];

    private NextHolder prev[];

    . . .
    public void remove(int list) {
        // check for null references
        next[list].setPrev(prev[list]);
        prev[list].setNext(next[list]);
    }

}
```

WME and Token

```
public class Token extends MultiListElement {

    public Token(Token parent, WME wme) {
        super(NUMBER_OF_LISTS);
        . . .
    }
}
```

Searching for WMEs and Tokens

In Alpha and Beta Networks

```
public abstract class Index<Type extends IMultiListElement>
                                 implements NextHolder {

    private static final int LOG2_INDEX_SIZE = 13;
    private static final int INDEX_SIZE = (((int) 1) <<
                                 LOG2_INDEX_SIZE);
    private static final int INDEX_MASK = (INDEX_SIZE - 1);

    final IMultiListElement index[];

    . . .
}
```

One global index for all scopes

We need to lock here

Synchronization in the Index

In Alpha and Beta Networks

```
public abstract class Index<Type extends IMultiListElement>
                                  implements NextHolder {
   . . .
   private static final int SEGMENT_SIZE =
                     (((int) 1) << LOG2_SEGMENT_SIZE);
   private static final int SEGMENT_MASK =
                     (SEGMENT_SIZE - 1);

   final ReentrantLock segment[];
   final IMultiListElement index[];

   . . .
}
```

Follow the same pattern that **ConcurrentHashMap** uses

Some results

- Tested 20,000 Products, with a significant amount of pricing rules
- We achieved response times of 5 to 10 msec per request
- Up to 600 Req/Sec on just a Core 2 Duo Machine, with 1.5GB of RAM in the Heap of the VM

Roadmap

- Production Version 1.0
  - Full Multithreaded Support
  - CLIPS like language
  - Full JSR 094 Support
  - Embedded and WAR Deployable Service
  - Zilonis Analysis Tool
  - Good Documentation
- Next Version
  - Rules Management Tool with JSR 208: Java Business Integration (JBI) Support
  - Natural Language Generation (GATE, KPLM)
  - XML-Rules Language
  - Support for SBVR

For More Information

- http://www.zilonis.org
- http://weblogs.java.net/blog/elevy
- Production Matching for Large Learning Systems
  - Robert B. Doorenbos
- SOAR Project:
  - http://sitemaker.umich.edu/soar/home
- JSR 094: Java Rule Engine API
  - http://jcp.org/aboutJava/communityprocess/review/jsr094/

Let’s see a Complete Demo!

Q&A