

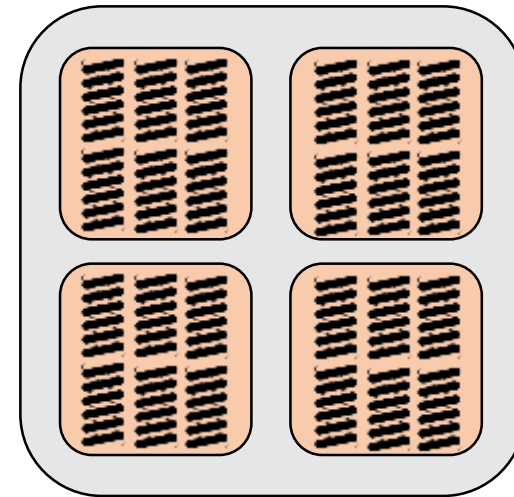
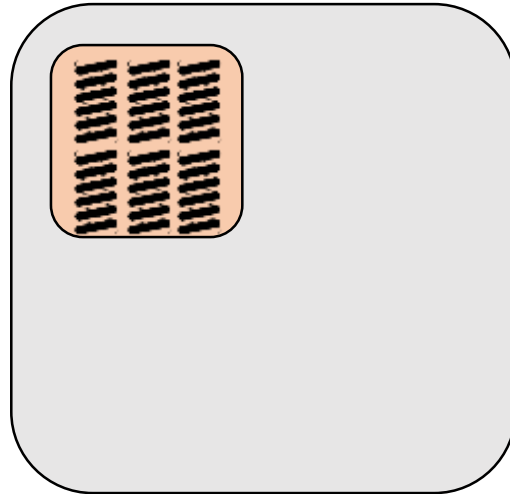
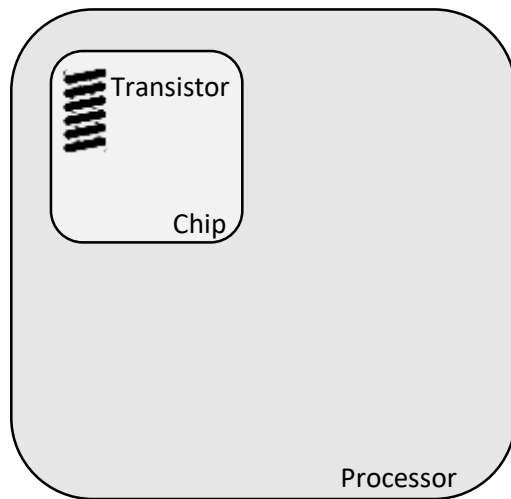
# Degradation: an effective approach to working with shared objects

Boubacar Kane

Supervisor : Pierre Sutra, Denis Conan

# Context

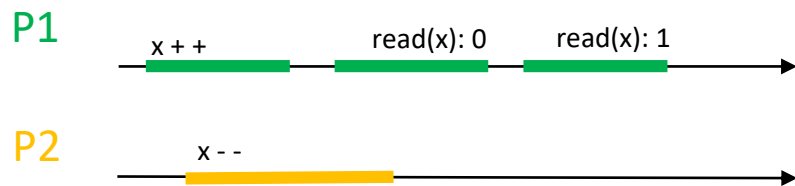
- Since about fifteen years, programmers have been using more and more parallel programs.



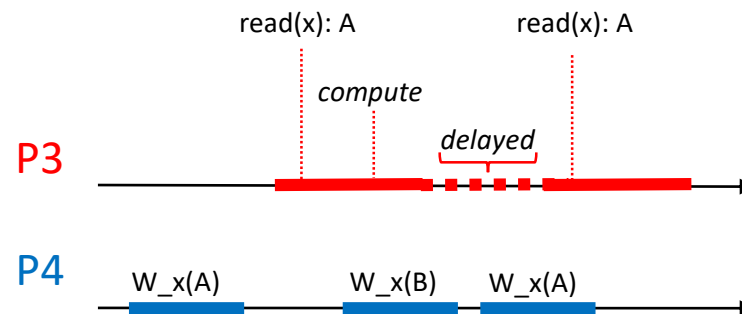
# Context

Working in parallel can bring some issues.

- Data consistency problems (ex: non-monotonic read)

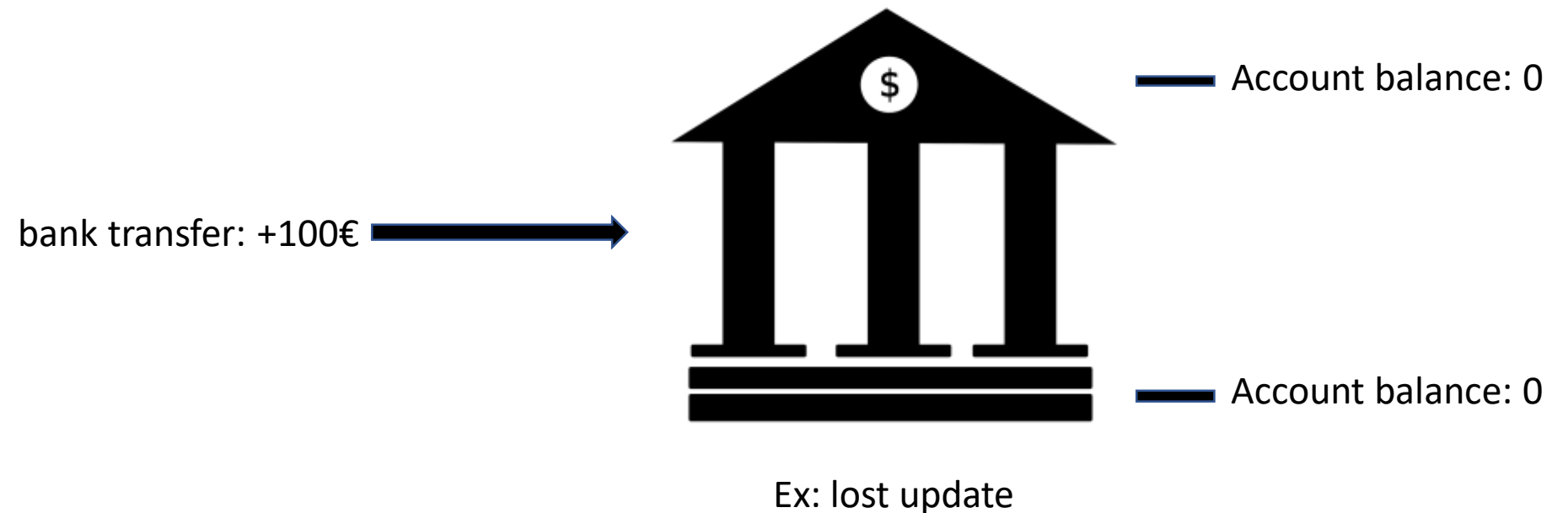


- Concurrent access problems (ex: ABA problem)



# Context

- Weak consistency :
  - Pros: Fast execution
  - Cons: Difficult to program with



# Context

- Strong consistency
  - Pros: Simplicity
  - Cons: Slower because we need to synchronize

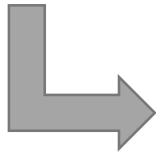
# Outline

- Motivation
- Degradable objects:
  - In distributed systems
  - In shared memory systems
- Indistinguishability graphs
- Future work

# Motivation

- Some programs do not use the whole sequential specification of the objects used.

The return value  
is not used



```
1 public class ScheduledMessageForwardingProcessor extends ScheduledMessageProcessor
2 {
3     private volatile AtomicInteger sendAttempts = new AtomicInteger(0);
4     ...
5
6     public int getSendAttemptCount() {
7         return sendAttempts.get();
8     }
9
10    public void incrementSendAttemptCount() {
11        sendAttempts.incrementAndGet();
12    }
13
14    public void resetSentAttemptCount(){
15        sendAttempts.set(0);
16    }
17    ...
18 }
```

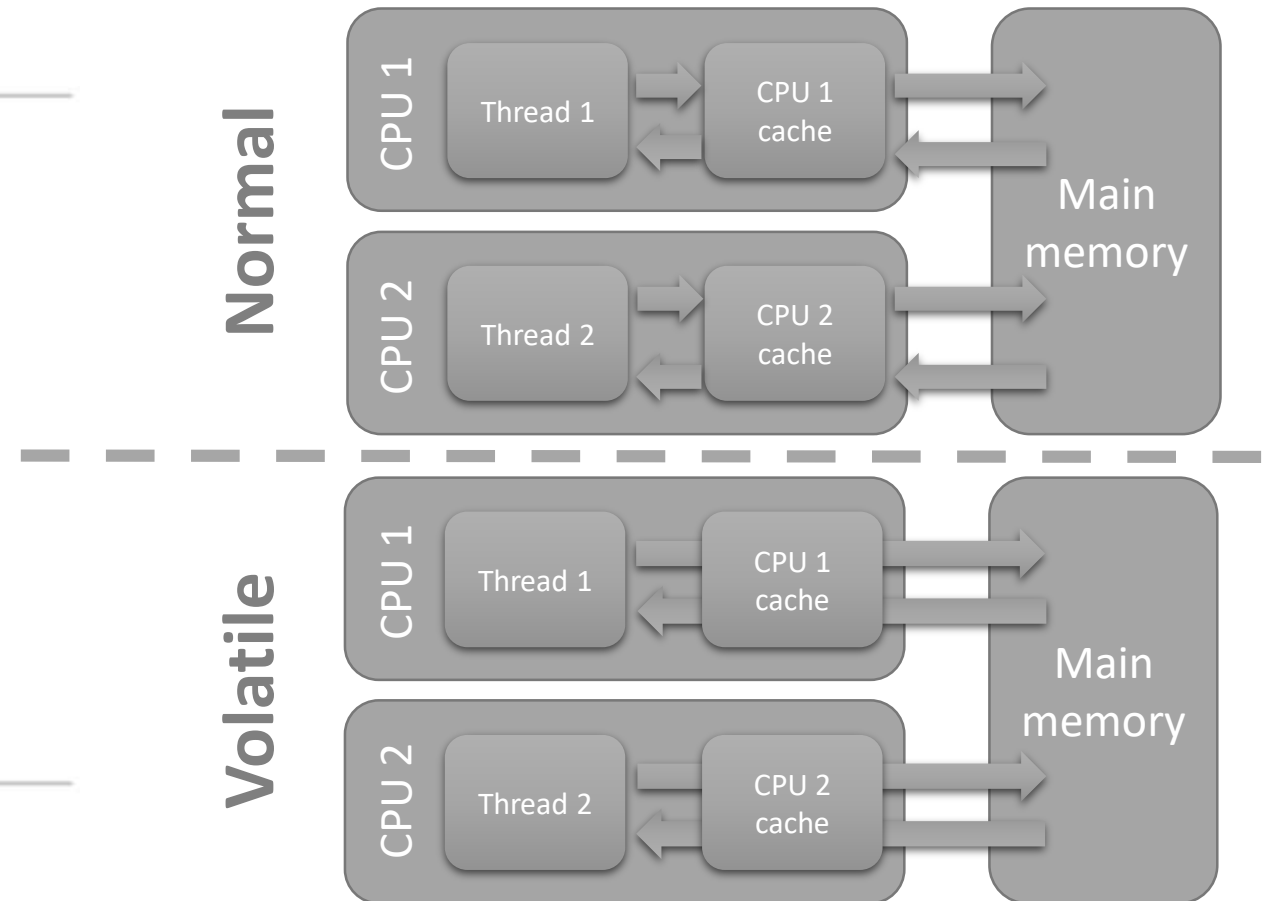
We could replace the  
AtomicInteger

# Motivation

- This method is already used in many programs

```
1 public class AtomicWriteOnceReference<T> implements Serializable {  
2     private transient T _cachedObj = null;  
3     private volatile T _obj = null;  
4  
5     ...  
6  
7     public T get() {  
8         if (_cachedObj != null)  
9             return _cachedObj;  
10  
11         _cachedObj = _obj;  
12         return _cachedObj;  
13     }  
14  
15     public void set(T value) {  
16         if (!trySet(value)) {  
17             throw new IllegalStateException("Value has already been set");  
18         }  
19     }  
20     ...  
21 }
```

Here, with an intermediate variable, we don't need to read the main memory each time.

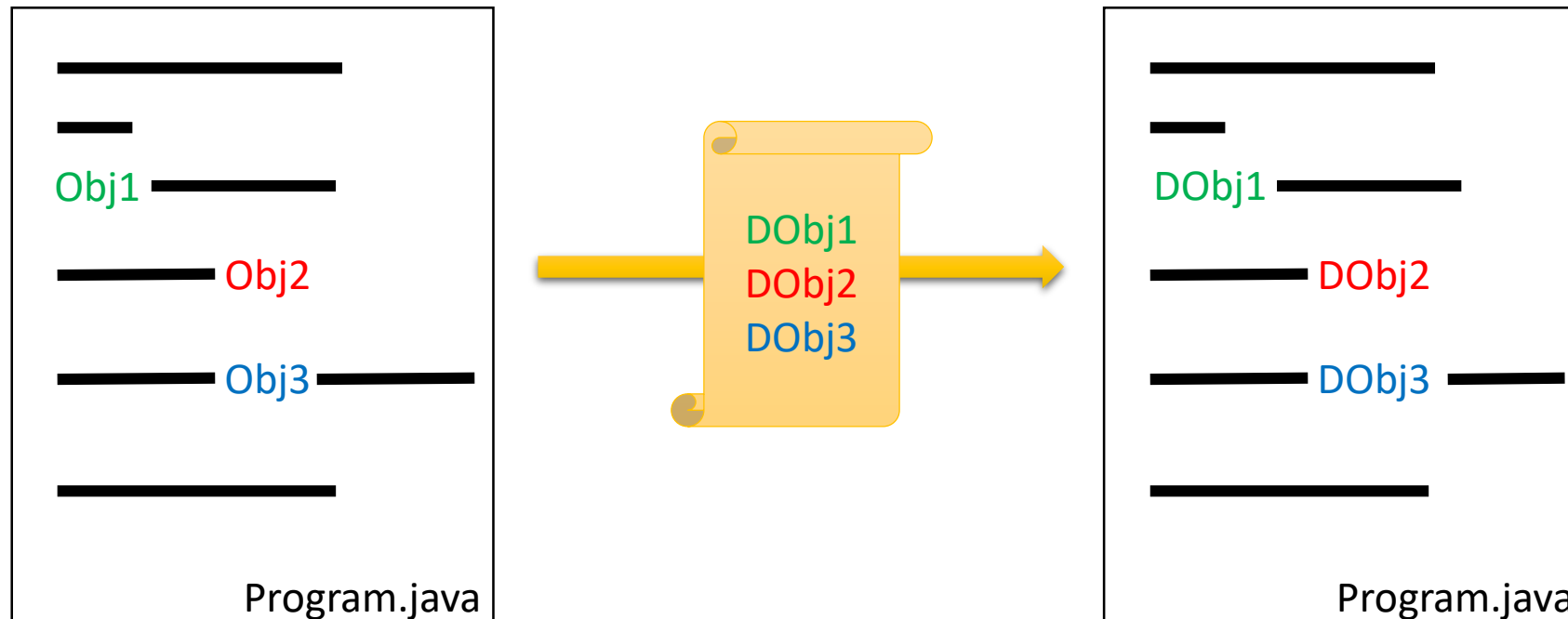




# Motivation

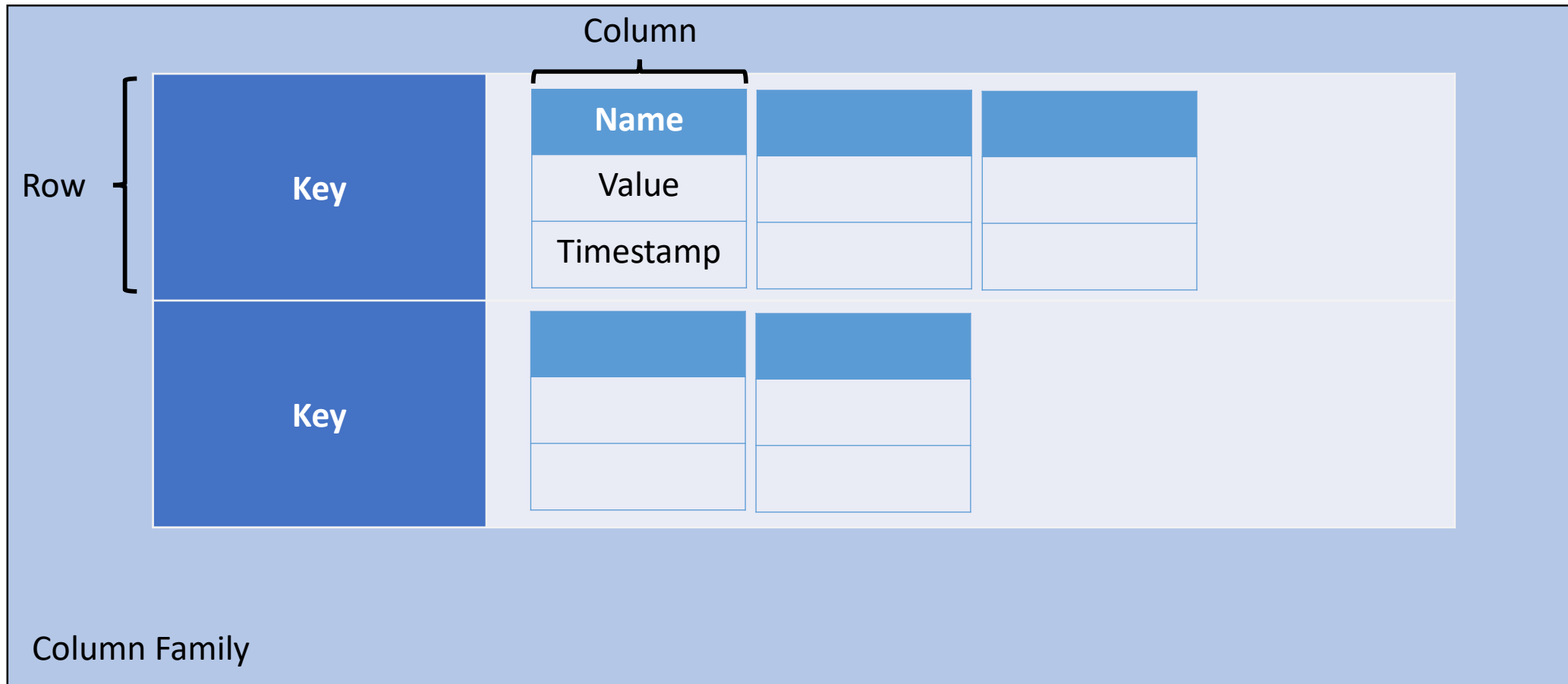
To improve the performance of a program without refactoring

- Our goal is to provide programmers with a library of degradable objects.



# Degradable object in distributed systems

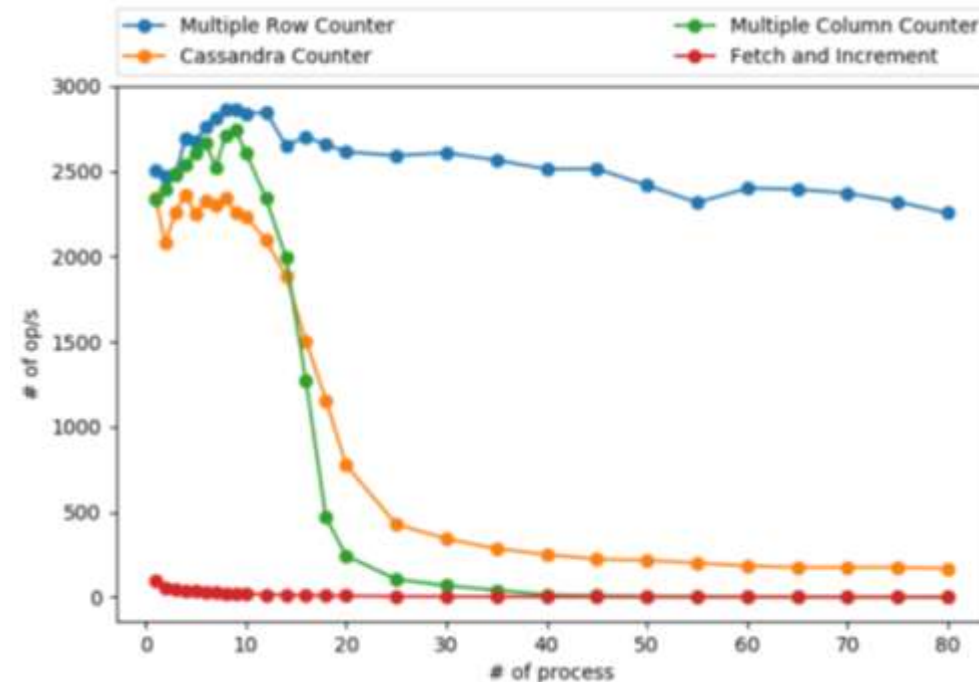
- Cassandra is a NoSQL database designed to handle large amounts of data



# Degradable object in distributed systems

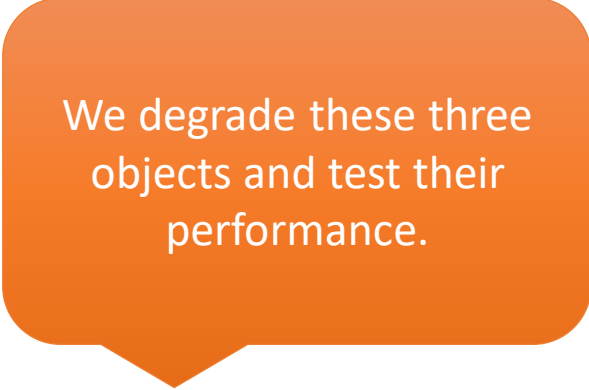
- Multiple Row vs Multiple Column
- Cassandra's Counter vs Degradable Counter

We want to know what is the best way to parallelize access to Cassandra. Then, we degrade cassandra's counter and see if we have better performance.



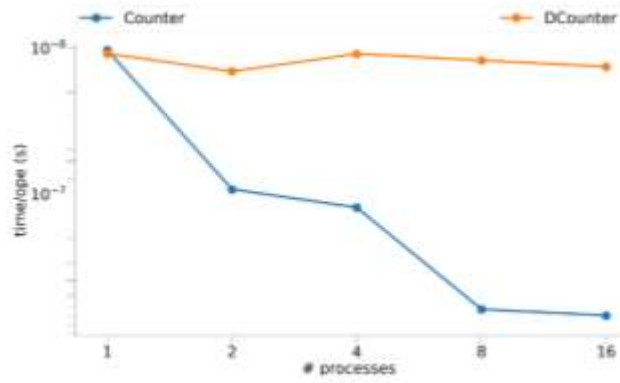
# Degradable object in shared memory systems

- Counter -> AtomicInteger
  - increment() and read()
- List -> ConcurrentLinkedQueue
  - add(x), remove(x) and contains(x)
- Set -> ConcurrentSkipListSet
  - add(x), remove(x) and contains(x)

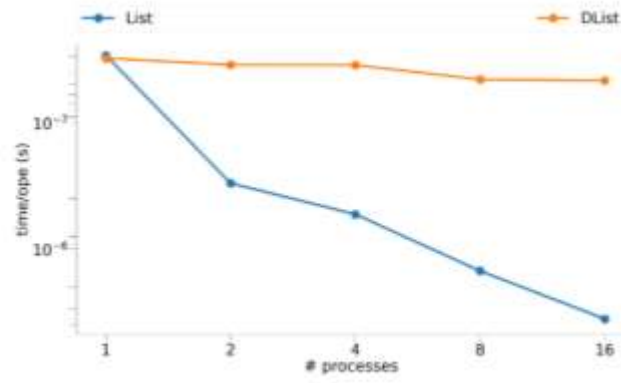
An orange callout box with rounded corners and a downward-pointing tail, containing text.

We degrade these three objects and test their performance.

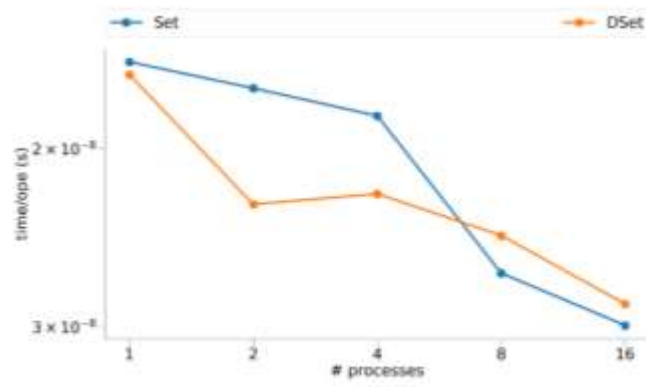
# Degradable object in shared memory systems



(a)



(b)



(c)

- We managed to obtain a factor of  $10^2$  for both the counter (a) and the list (b).

It was not possible to improve the performance of the set (c) since the ConcurrentSkipListSet object of the java library handles the contention correctly.

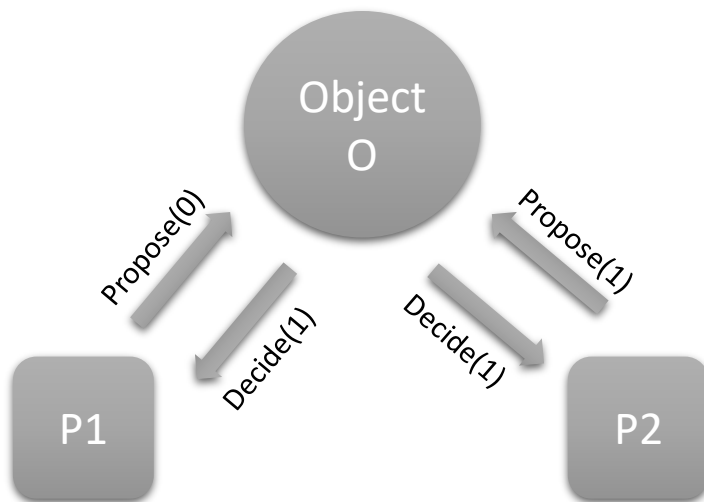
# Indistinguishability graphs

- Synchrony power: the less the better

We focused on objects with low synchrony power because they offer better performance.

# Indistinguishability graphs

- Consensus number introduced by Herlihy
  - Consensus protocol:
    - Consistent
    - Wait-free
    - Valid



Objects with a consensus number of 1 are at the bottom of the hierarchy and thus have the lowest level of synchrony. This is why it is interesting to make parallel accesses with these objects.

# Indistinguishability graphs

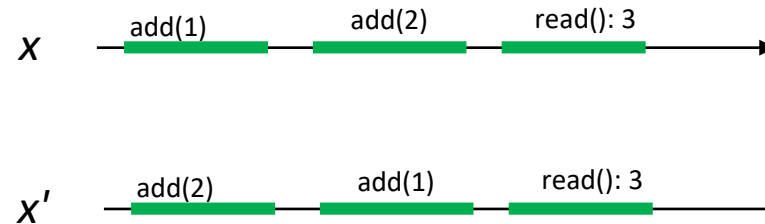
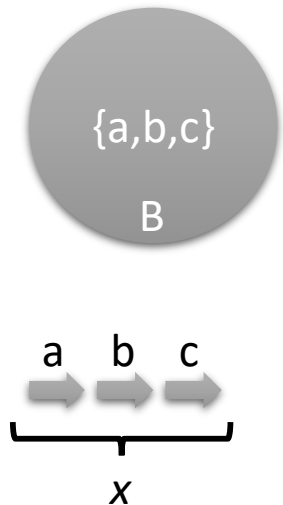
- There are objects with consensus numbers of 1 that have different complexity:
  - Register:  $O(1)$
  - Counter:  $\Omega(n^2)$

The counter object uses a snapshot algorithm. It is known that the complexity of such an algorithm is  $\Omega(n^2)$ .

This is why we are looking for a new way to measure the synchronic power and therefore the level of parallelism.



# Indistinguishability graphs

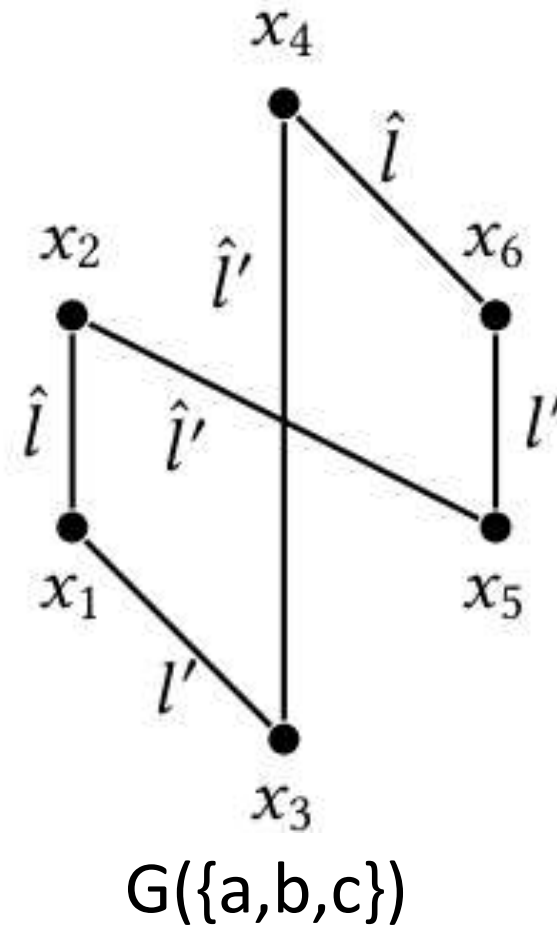


$\text{add}(1): a$   
 $\text{add}(2): b$   
 $\text{read}(): c$

$x$  and  $x'$  are indistinguishable for  $c$  because  $c$  has the same return value and there exists a common state in both executions.

# Indistinguishability graphs

$x_1 = abc$	$l = \{a, b\}$
$x_2 = acb$	$l' = \{c\}$
$x_3 = bac$	$l'' = \{a, b, c\}$
$x_4 = bca$	$\hat{l} = \{a\}$
$x_5 = cab$	$\hat{l}' = \{b\}$
$x_6 = cba$	



Indistinguishability graph for a counter with these three operations:

- $a = \text{increment}(1)$
- $b = \text{increment}(3)$
- $c = \text{increment}(5)$

Here the increment operations return the value of the counter after performing the operation. It is interesting to note that, if no operation has a return value, then we have a complete graph.

# Future work

- Provide an accurate measure of the level of parallelism through the indistinguishability graph.
- Test the performance of our objects in an application.
- Carry out this study in a model of lower consistency

