University of Stuttgart
Germany

Lukas Epple

Distributed Persistent Objects

June 27, 2023

University of Stuttgart, IPVS

Everyone uses databases.

Why?

Consistent access to persistent data

$$\text{Database} \implies \text{Transactional data}$$

Are databases always the best solution?

Let's take a step back!

$$\text{Database} \overset{?}{\Longleftarrow} \text{Transactional data}$$

Motivation

- can we integrate the persistency into the programs?
- same object representation
- avoid query processing
- developers are already familiar with object-oriented data access

Persistent Objects

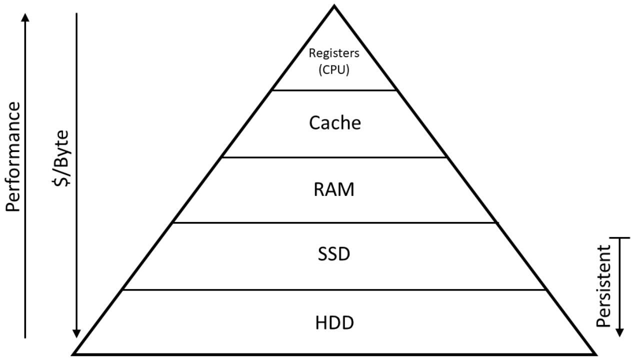
Goal: Persistent Objects

- persistency
- objects of any kind
- transactional behavior
- speed

First Step: Persistent Storage Hardware

Perfect hardware would be

- cheap
- fast
- persistent
- durable

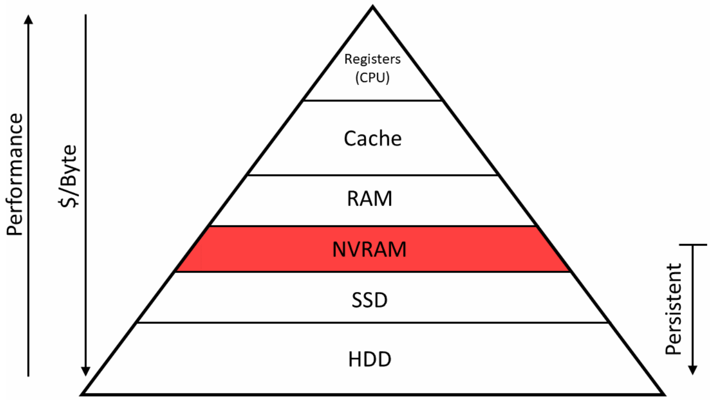


Intel® Optane™

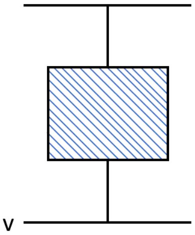
- Released on March 19, 2017
- SSD and DIMM

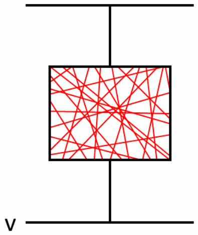
Random Write	Intel Optane	DRAM	SSD	
Latency	0.17/0.3	0.08	60	[µs]
Bandwidth	1.4	5.6	0.2	[$GB\,s^{-1}$]
Durability	360	-	0.7	[PBW]

NVRAM in the Memory Hierarchy

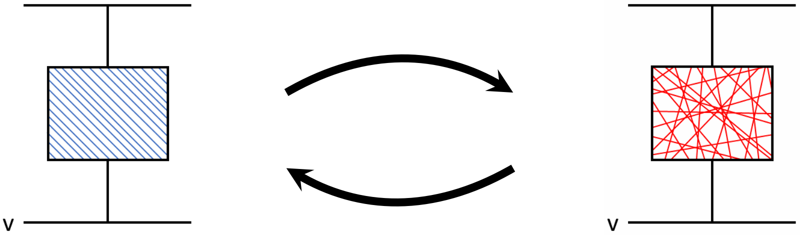


How does Intel® Optane™ work?

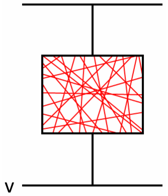


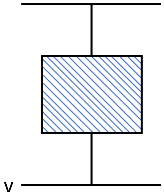


How does Intel® Optane™ work?

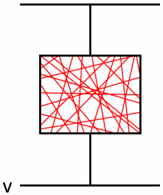


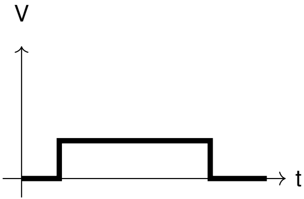
Writing

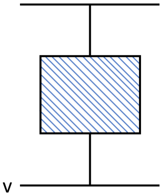




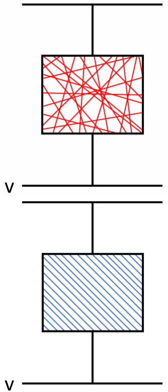
Writing

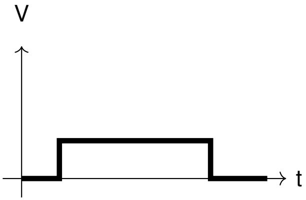


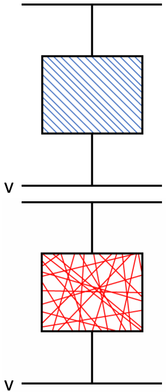




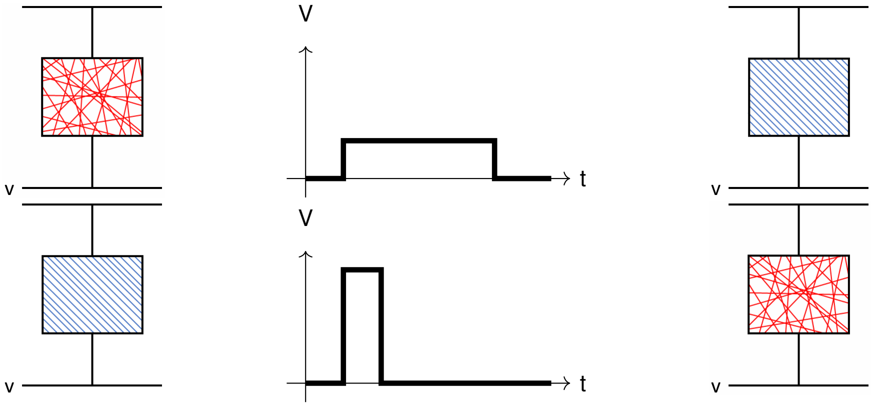
Writing



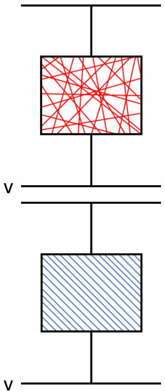




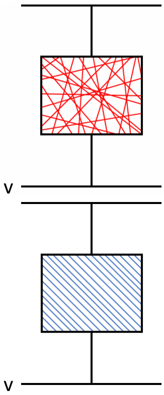
Writing

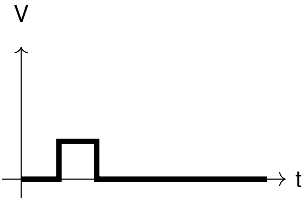


Reading

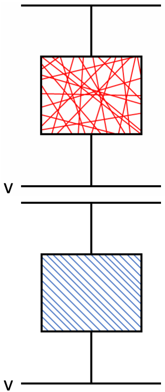


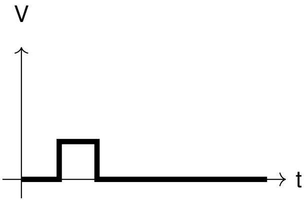
Reading





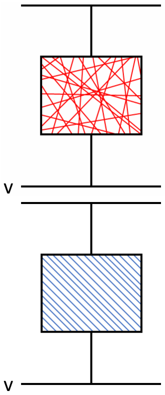
Reading

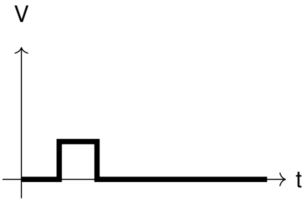




0

Reading





0

1

But is it that simple?

Optane seems to be that magic memory!

- latency comparable to DRAM
- bandwidth almost as high as DRAM
- more durable than SSDs

But is it that simple?

Optane seems to be that magic memory!

- latency comparable to DRAM
- bandwidth almost as high as DRAM
- more durable than SSDs

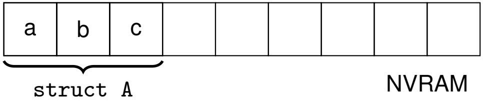
Is that enough?

Remaining problems:

- Optane only guarantees atomic write for 8 Bytes
- Working with data on persistent storage is not trivial

Objects on Persistent Storage: Pitfalls

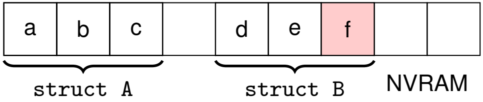
```
struct A {
  int a;
  int b;
  int c;
}

A *obj = nv_alloc(sizeof(A));
```



Objects on Persistent Storage: Pitfalls

```
struct A {
  int a;
  int b;
  int c;
}

A *obj = nv_alloc(sizeof(A));
```

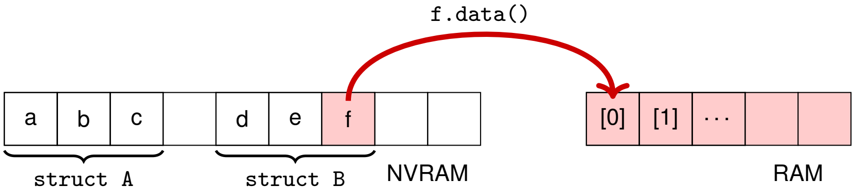
```
struct B {
  int d;
  int e;
  std::vector<int> f;
}

B *obj = nv_alloc(sizeof(B));
```



Objects on Persistent Storage: Pitfalls

```
struct A {
  int a;
  int b;
  int c;
}

A *obj = nv_alloc(sizeof(A));
```

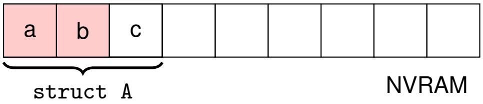
```
struct B {
  int d;
  int e;
  std::vector<int> f;
}

B *obj = nv_alloc(sizeof(B));
```



Objects on Persistent Storage: Pitfalls

```
1 struct A {
2   int a;
3   int b;
4   int c;
5 }
6 A *obj = nv_alloc(sizeof(A));
7
8 obj->a = 5;
9 obj->b = 6;
```

What if the computer crashes between line 8 and 9?



We need Software that Manages Persistent Storage

Intel created the `libpmemobj-cpp` library for Optane

- provides transactional behavior with undo logs, and
- persistent data structures

We need Software that Manages Persistent Storage

Intel created the `libpmemobj-cpp` library for Optane

- provides transactional behavior with undo logs, and
- persistent data structures

But there are problems:

- we want no restrictions on the objects: Intel's data structures too limited
- translation between persistent and volatile objects

We need Software that Manages Persistent Storage

Intel created the `libpmemobj-cpp` library for Optane

- provides transactional behavior with undo logs, and
- persistent data structures

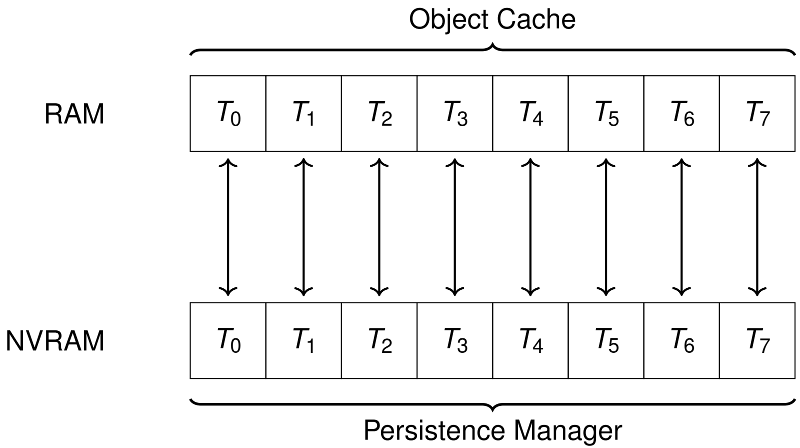
But there are problems:

- we want no restrictions on the objects: Intel's data structures too limited
- translation between persistent and volatile objects
- Optane was discontinued in 2022: we need a technology-independent solution

What do we need?

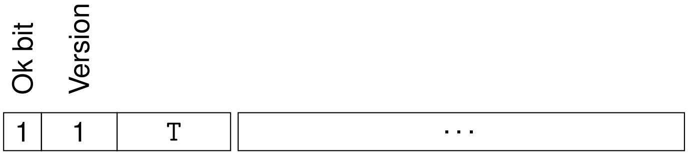
- Persistent storage
- Memory mapped files for technology-independence
- Translation between volatile and persistent data structures
- Object Cache in the volatile RAM for accelerated reads
- Transactional writes

Object Organization



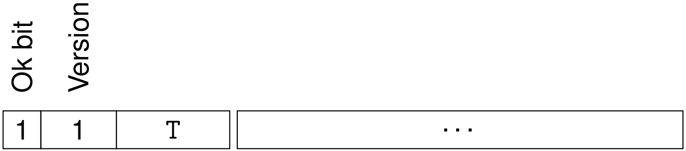
Transaction Execution

Initial memory layout



Transaction Execution

Initial memory layout



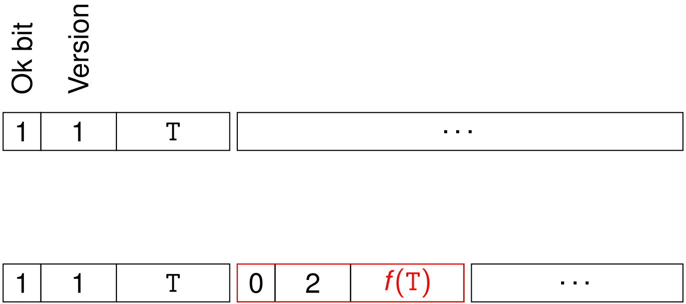
Execute f on `T` in cache

Transaction Execution

Initial memory layout

Execute f on `T` in cache

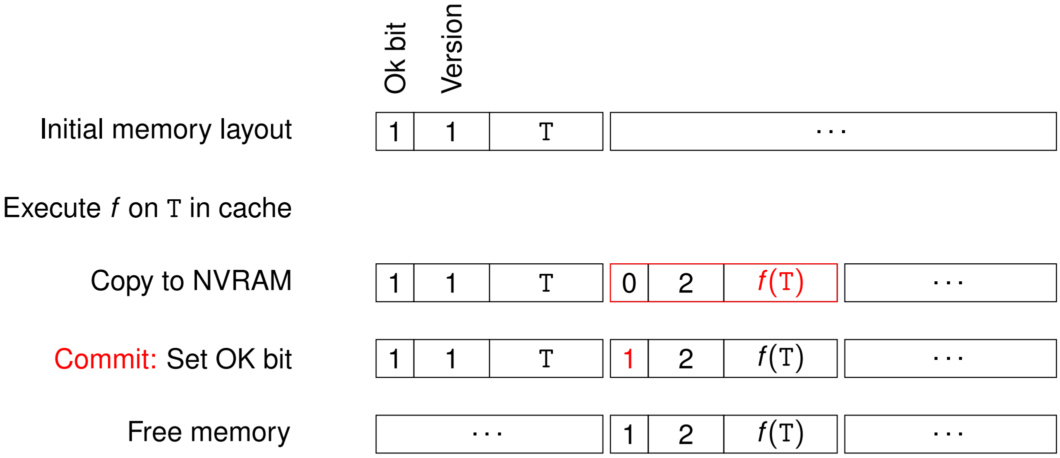
Copy to NVRAM



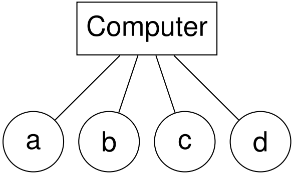
Transaction Execution

	Ok bit	Version					
Initial memory layout	1	1	T	· · ·			
Execute f on T in cache							
Copy to NVRAM	1	1	T	0	2	$f(T)$	· · ·
Commit: Set OK bit	1	1	T	1	2	$f(T)$	· · ·

Transaction Execution



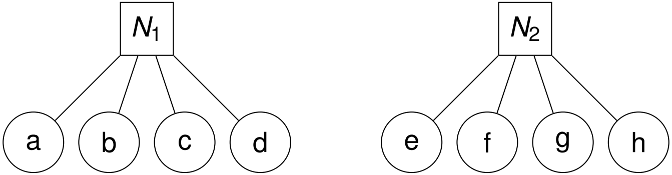
Until now: Local Objects



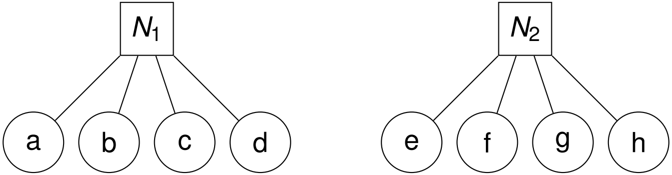
This gives us persistency and transactions but we are limited in capacity, access, . . .

Distributed Persistent Objects

Distributed Persistent Objects



Distributed Persistent Objects



With Distributed Persistent Objects we need a way to

- reference non-local objects,
- select subsets of all objects,
- query for objects, and
- execute transactions on sets of objects

The Basic Concept

Scala:

```
numbers
.filter(_ < 5)
.map(x => x * x)
.foldLeft(0)((acc, current) => max(acc, current))
```

The Basic Concept

Scala:

```
numbers
.filter(_ < 5)
.map(x => x * x)
.foldLeft(0)((acc, current) => max(acc, current))
```

Even modern C++23:

```
numbers
| views::filter([](const auto& x){
return x < 5;
})
| views::transform([](const auto& x){
return x * x;
})
| ranges::fold_left(0, std::max<int>);
```

Our Adaption: Views and Actions

This pattern is extremely powerful even on the local machine.
Nothing prevents us from using it in a distributed setting!

Our Adaption: Views and Actions

This pattern is extremely powerful even on the local machine.
Nothing prevents us from using it in a distributed setting!

Views: access objects

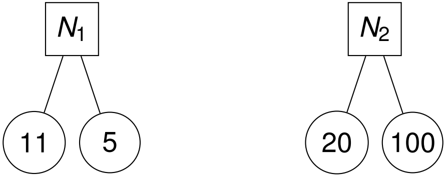
- `map`
- `filter`
- `elem`

Actions: operations on objects

- `reduce`
- `transact`

Views: Example

- Two nodes, each stores a set of objects with `int` values.
- Select objects whose string representation is two characters long.



Views: Example (contd.)

View<int> 11 5 20 100

map λx: x.str()

filter λx: |x| == 2

elem()

Views: Example (contd.)

`View<int>`	11	5	20	100
map λx: x.str()	"11"	"5"	"20"	"100"
filter λx: |x| == 2				
elem()				

Views: Example (contd.)

View<int>	11	5	20	100
map λx: x.str()	"11"	"5"	"20"	"100"
filter λx: |x| == 2	"11"		"20"	
elem()				

Views: Example (contd.)

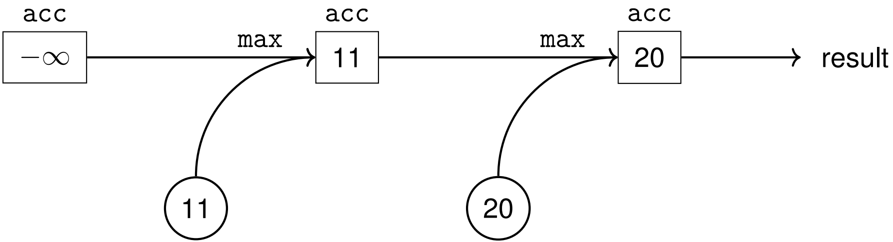
View<int>	11	5	20	100
map λx: x.str()	"11"	"5"	"20"	"100"
filter λx: |x| == 2	"11"		"20"	
elem()	11		20	

Views: Example (contd.)

- Nodes N_1, N_2 can calculate the view independently
- How do Actions work with the result of a view?

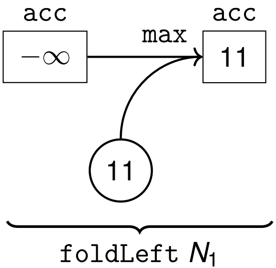
Actions: Fold Left Operation

- `def foldL[B](z: B)(op: (B, A) => B): B`
- Calculate the maximum of all values in the View and return it

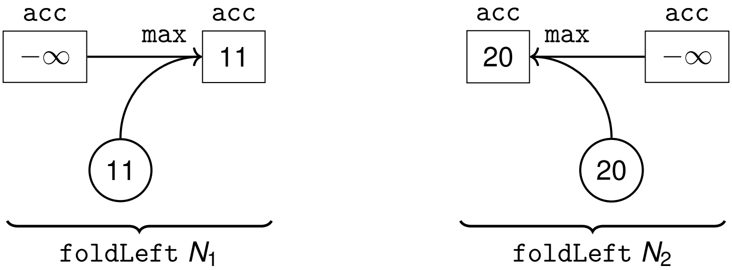


Problem: Objects are not on the same node!

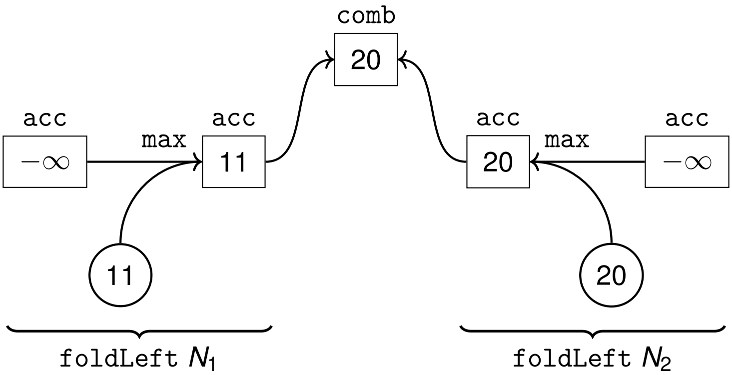
Actions: Reduce Operation



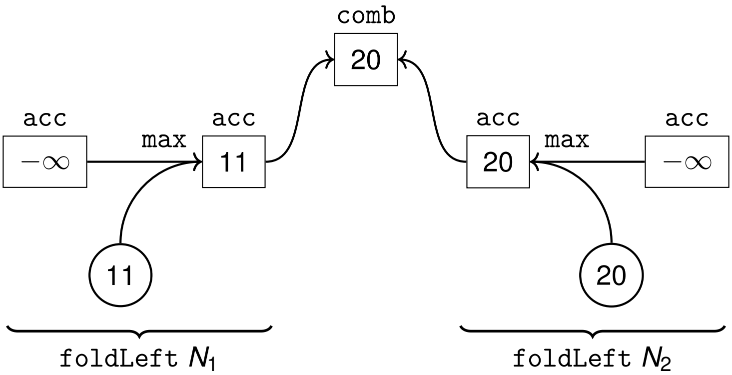
Actions: Reduce Operation



Actions: Reduce Operation



Actions: Reduce Operation



- `def reduce[B](z: B)(op: (B, A) => B)(comb: (B, B) => B): B`
- `reduce` operation allows parallelization of `foldLeft`

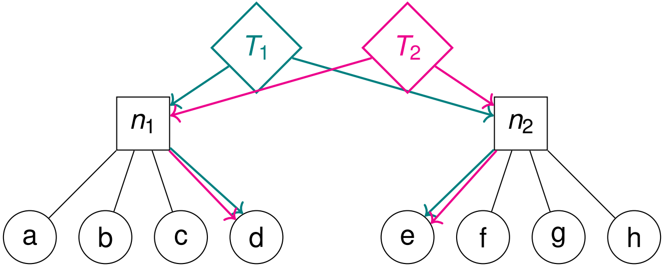
Code

The previous example could be implemented like this

```
View<int>::create()
  .map(std::to_string<int>)
  .filter([](const auto& x){
      return x.length() == 2;
    })
  .elem()
  .reduce(
    0,             // initial accumulator value
    std::max<int>, // foldLeft
    std::max<int>  // reduce
  )
  .build();
```

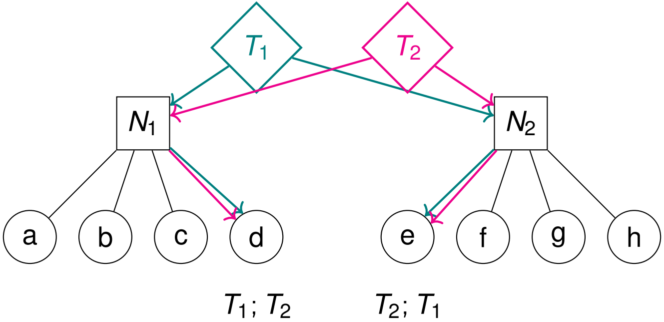
What About Write Operations?

- create a View to select objects
- use the `transact` Action on the view



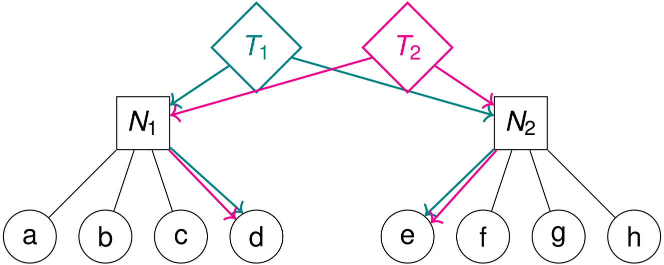
What About Write Operations?

- How do we ensure the order of independent transactions?



What About Write Operations?

- How do we ensure the order of independent transactions?
- we only have *local* atomicity



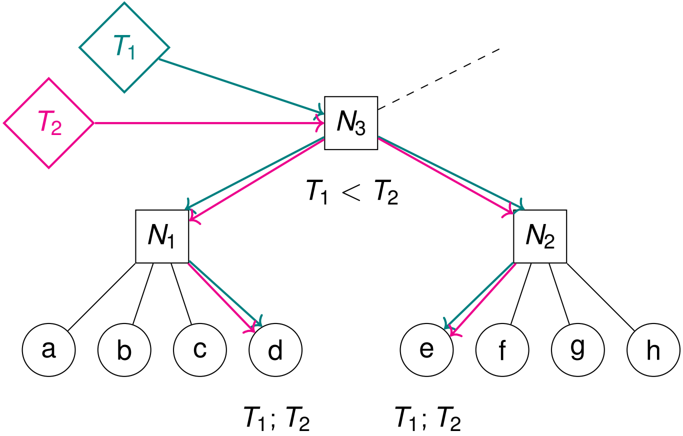
T_1; T_2 ✗ T_2; T_1

Solution

- build a tree of nodes
- every transaction is sent to the lowest common parent node of all objects involved
- this node decides the order of transactions

Sequential Consistency of Distributed Transactions

- N_3 is the lowest common parent of N_1, N_2



Conclusion

- NVRAM is a promising technology
- We can execute distributed transactions without databases
- Views are a powerful abstraction for the interaction

Thank you for your attention!

For further inquiries, contact:

Lukas Epple	lukas.epple@ipvs.uni-stuttgart.de
Simon König	st156571@stud.uni-stuttgart.de
Joel Waimer	st167572@stud.uni-stuttgart.de