Understanding Eventual Consistency

Hongseok Yang

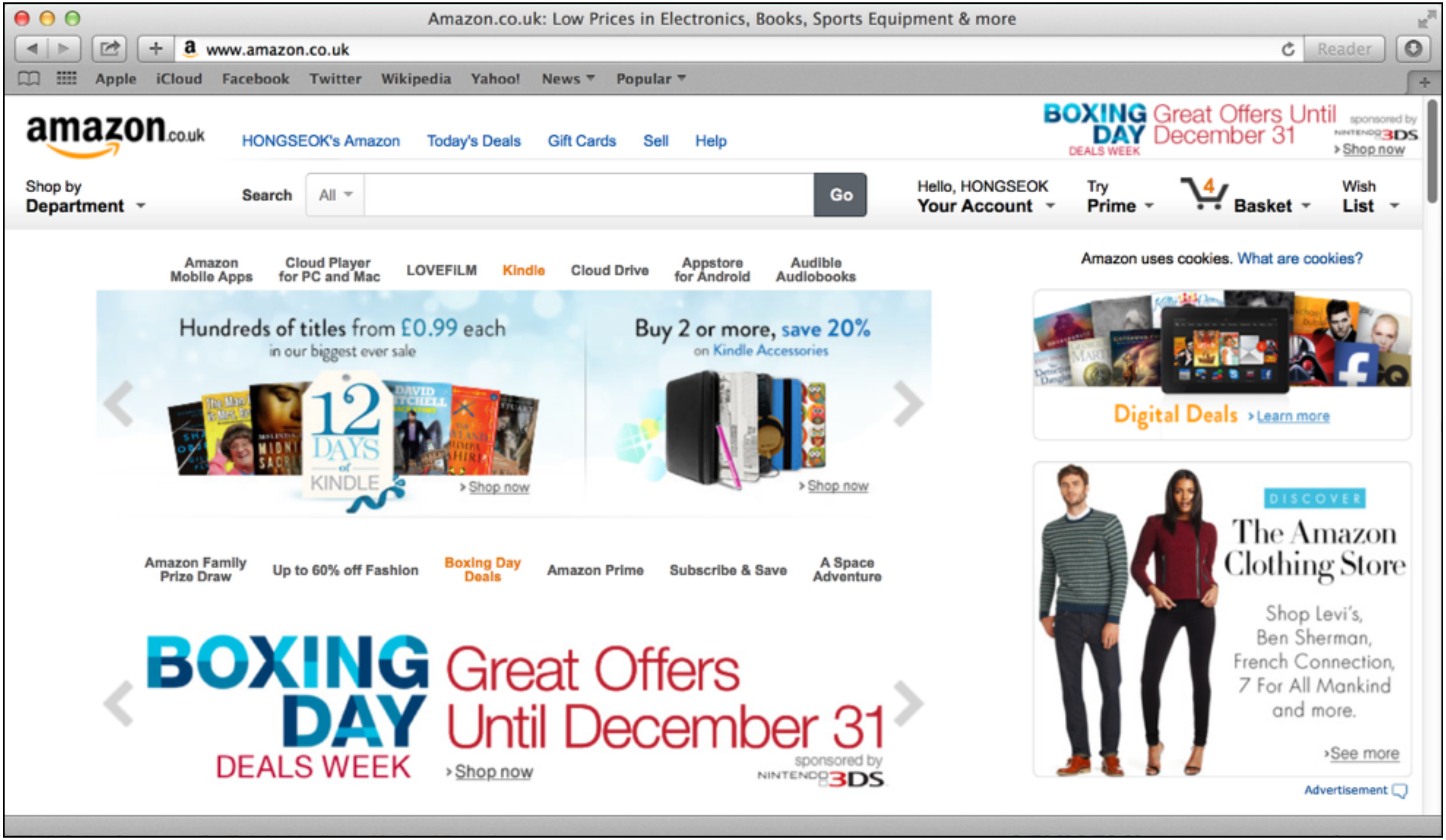
University of Oxford

Joint work with Sebastian Burckhardt (MSR Redmond), Alexey Gotsman (IMDEA) and Marek Zawirski (UPMC)

Who heard about “eventual consistency” before?

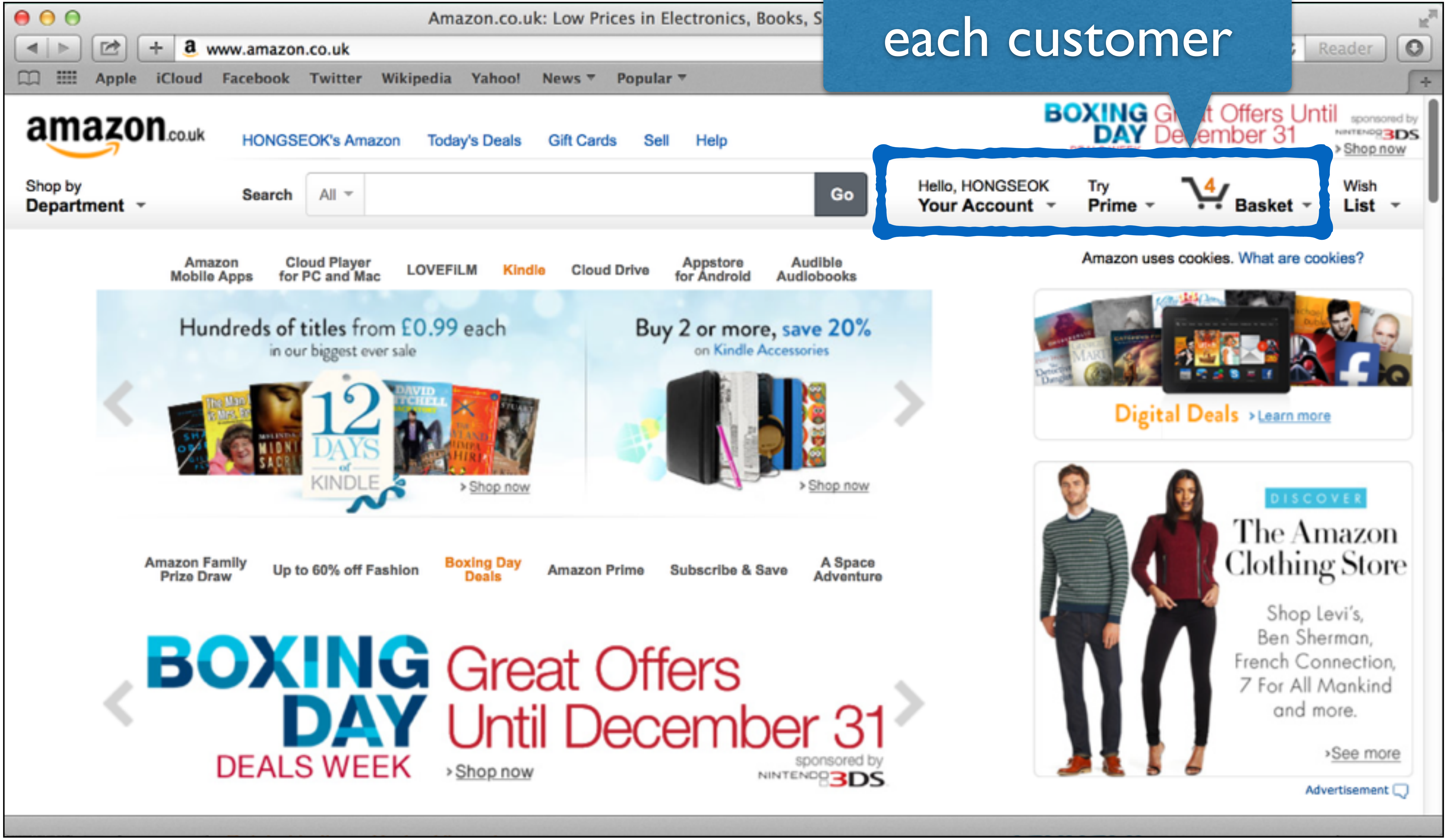
Learning outcome

To understand eventual consistency and research opportunities about it.



Amazon uses cookies. What are cookies?

Information about each customer

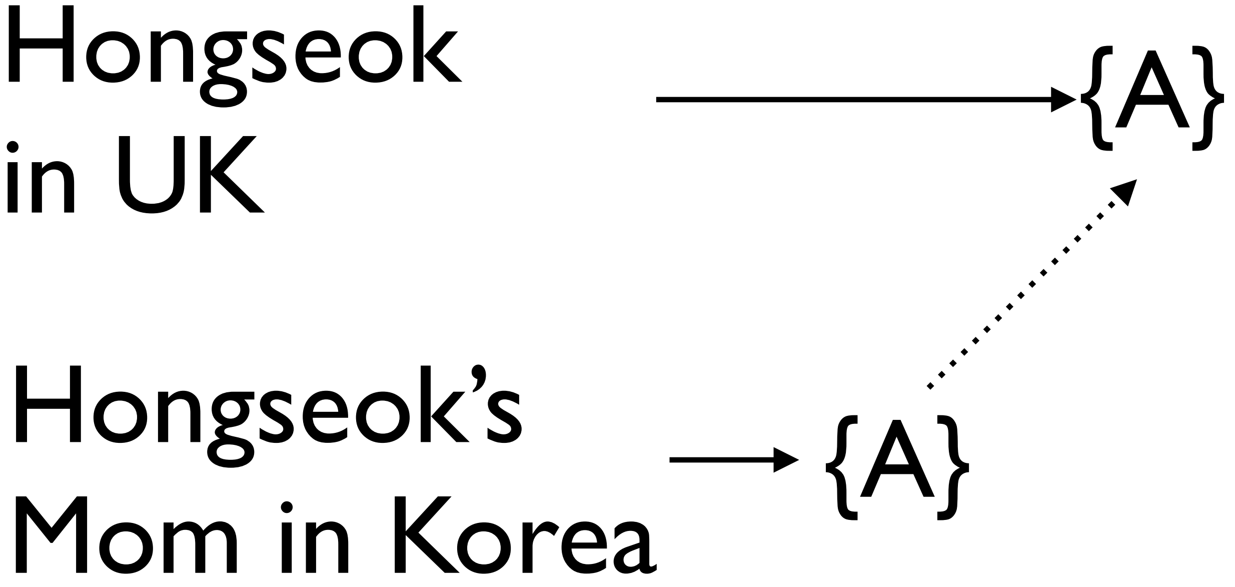


Amazon shopping cart (in 2008)

Hongseok
in UK

Hongseok's
Mom in Korea → {A}

Amazon shopping cart (in 2008)



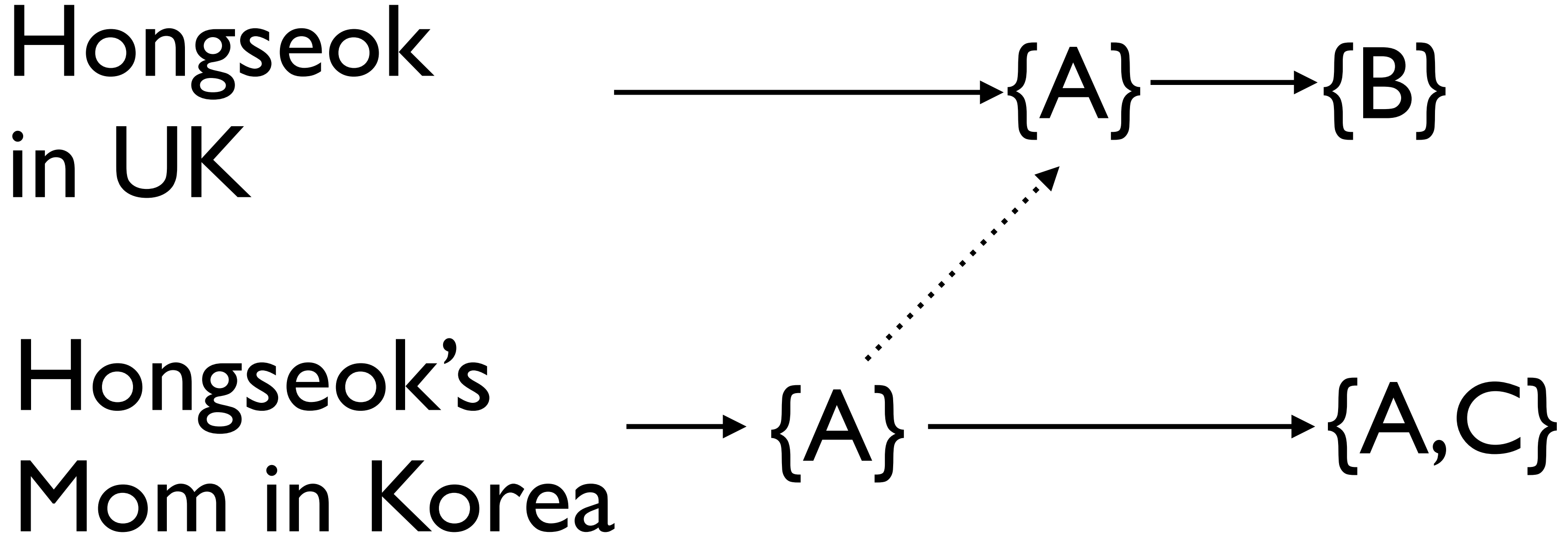
Hongseok
in UK

{A}

Hongseok’s
Mom in Korea

{A}

Amazon shopping cart (in 2008)



Hongseok in UK

{A}

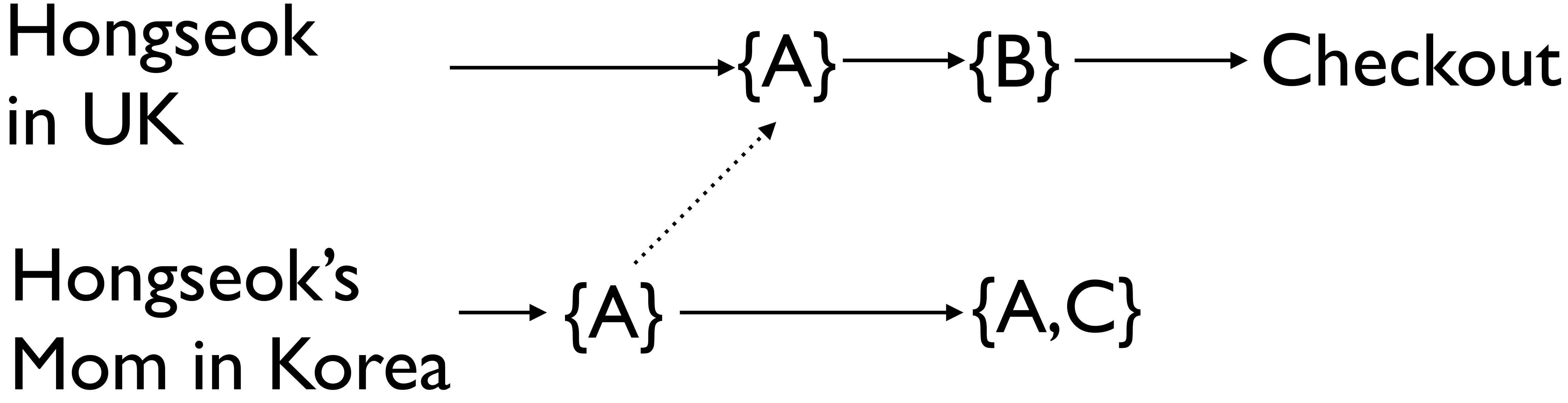
{B}

Hongseok's Mom in Korea

{A}

{A,C}

Amazon shopping cart (in 2008)



Hongseok in UK

{A}

{B}

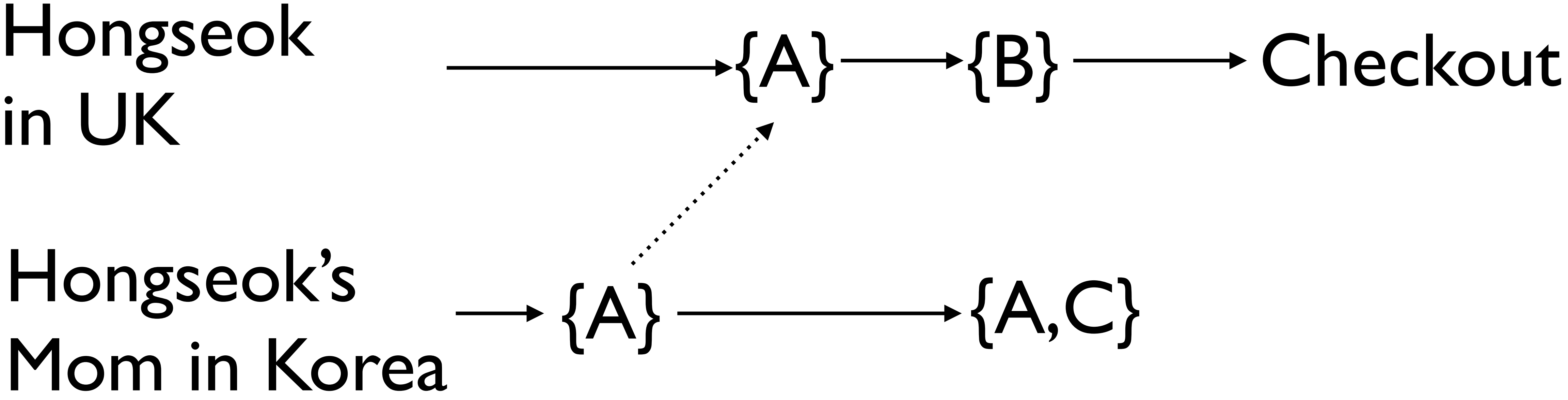
Checkout

Hongseok's Mom in Korea

{A}

{A,C}

Amazon shopping cart (in 2008)



Hongseok in UK

{A}

{B}

Checkout

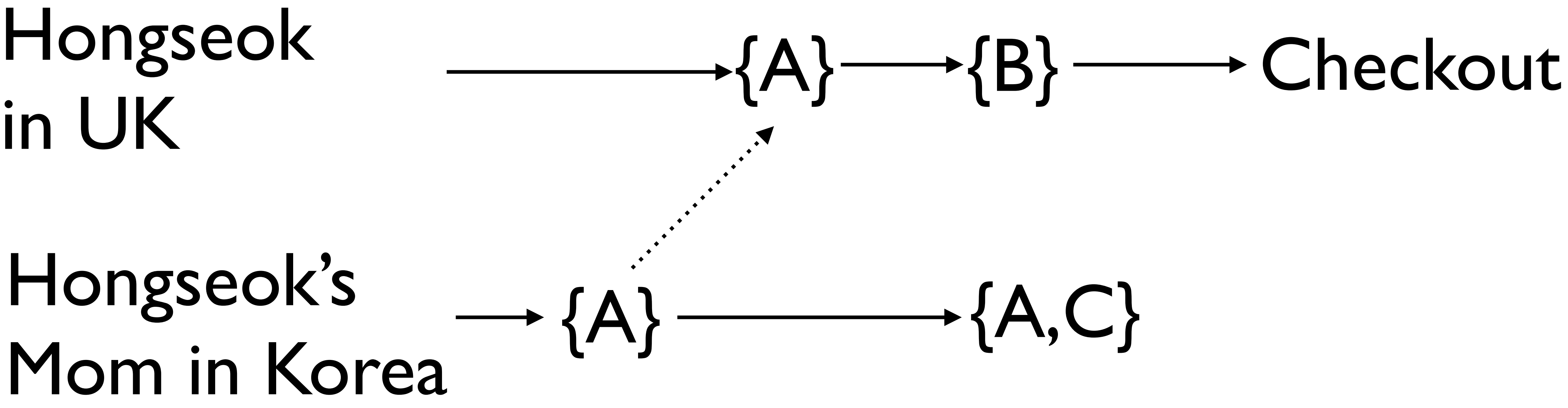
Hongseok's Mom in Korea

{A}

{A,C}

[Q] What could the shopping cart have at checkout?
1. {B} 2. {A,C} 3. {}

Amazon shopping cart (in 2008)



Hongseok in UK

{A}

{B}

Checkout

Hongseok's Mom in Korea

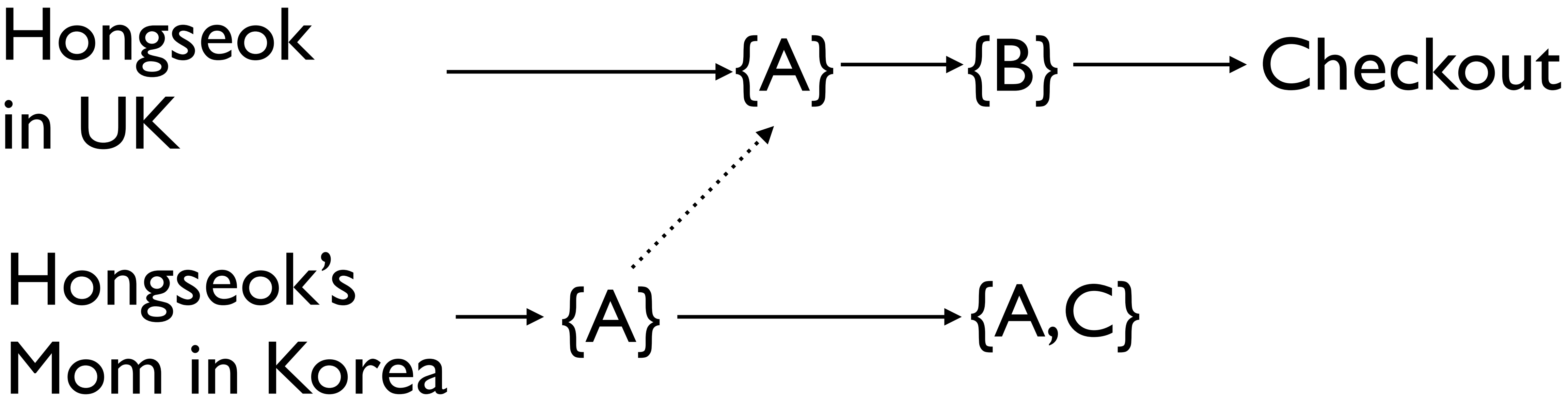
{A}

{A,C}

[Q] What could the shopping cart have at checkout?

1. {B} 2. {A,C} 3. {}

Amazon shopping cart (in 2008)



Hongseok in UK

{A}

{B}

Checkout

Hongseok's Mom in Korea

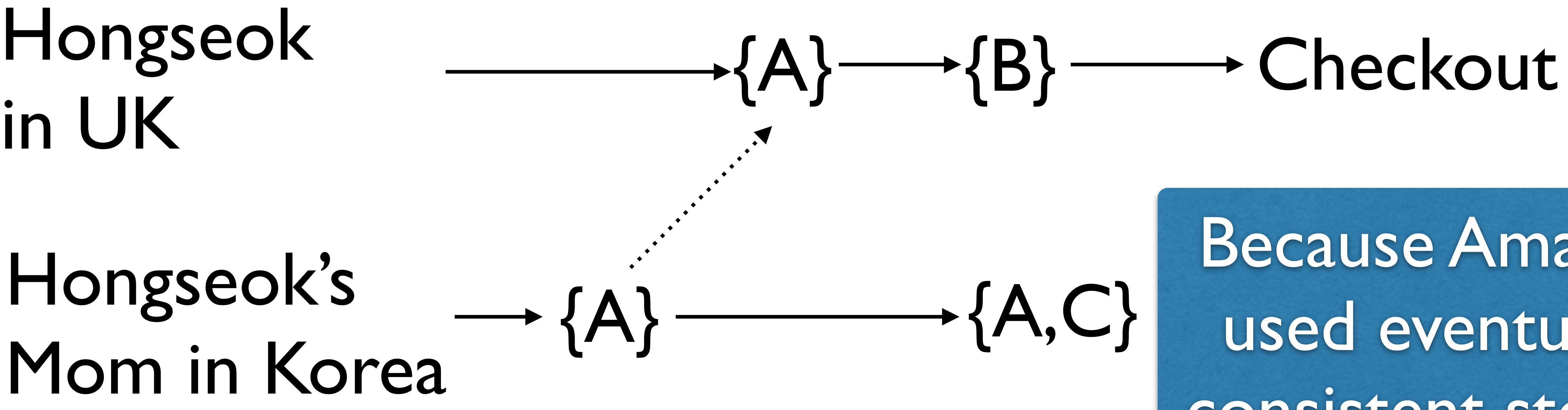
{A}

{A,C}

[Q] What could the shopping cart have at checkout?

1. {B} 2. {A,C} 3. {} 4. {A,B,C}

Amazon shopping cart (in 2008)



Hongseok in UK

{A} {B} Checkout

Hongseok's Mom in Korea

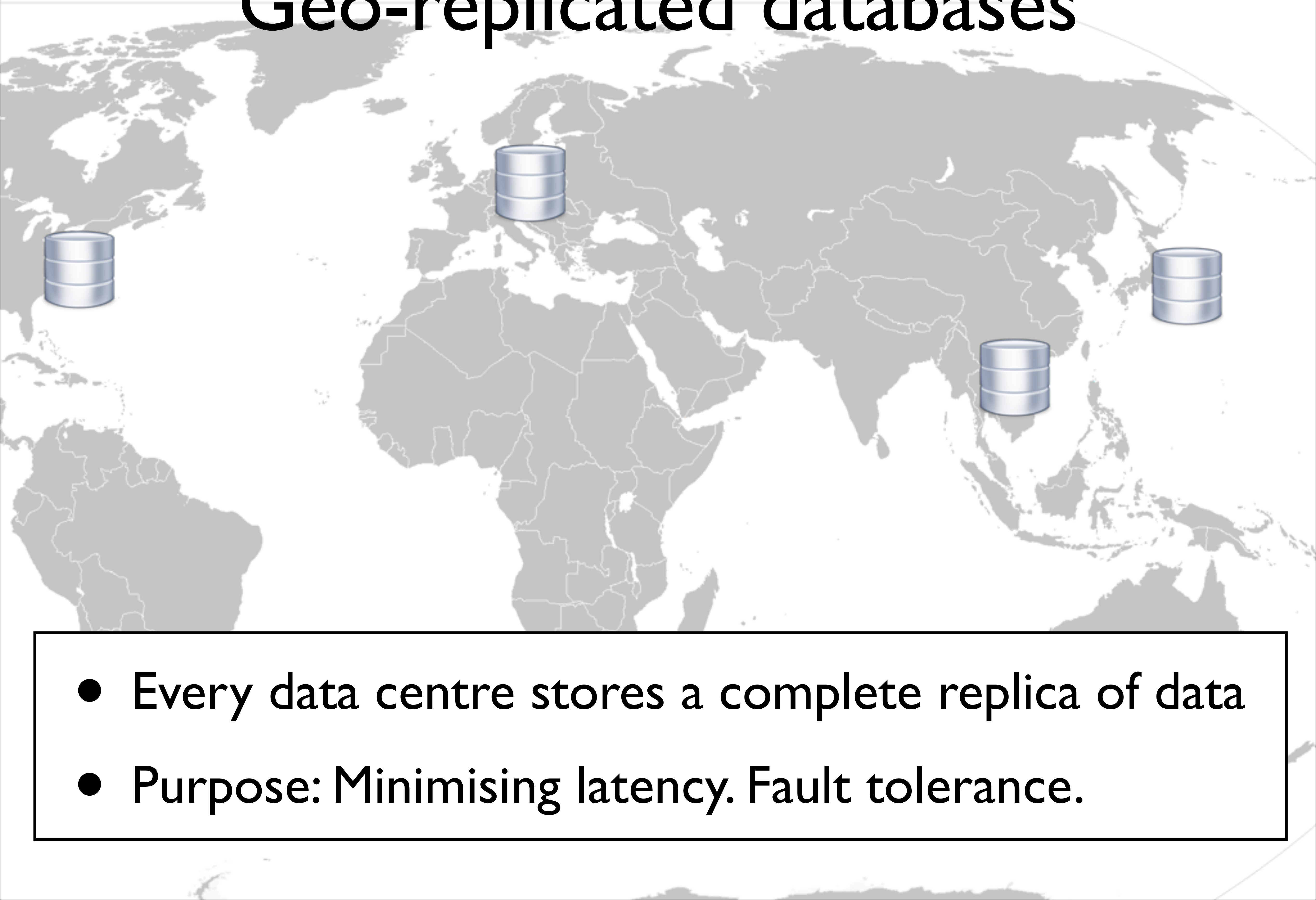
{A} {A,C}

Because Amazon used eventually consistent stores.

[Q] What could the shopping cart have at checkout?

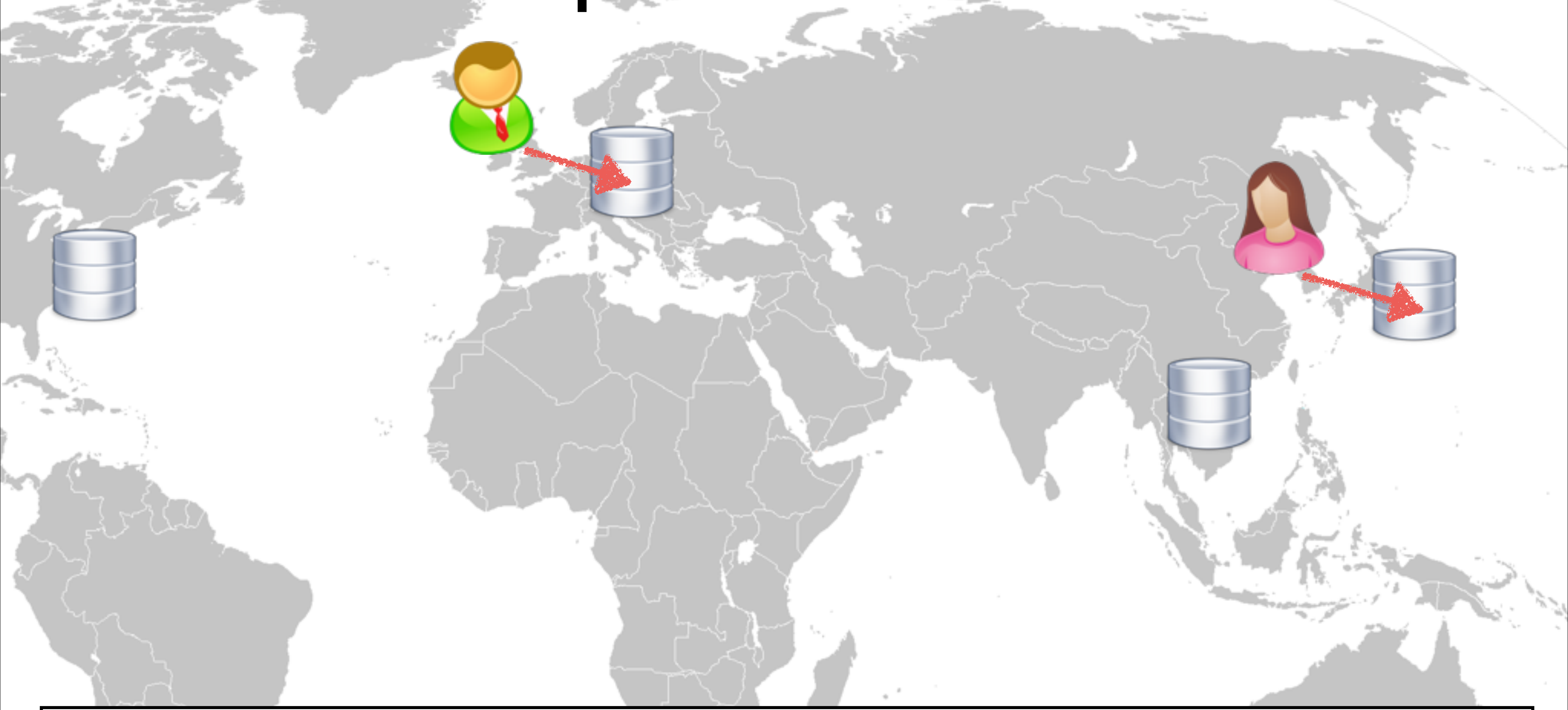
1. {B} 2. {A,C} 3. {} 4. {A,B,C}

Geo-replicated databases



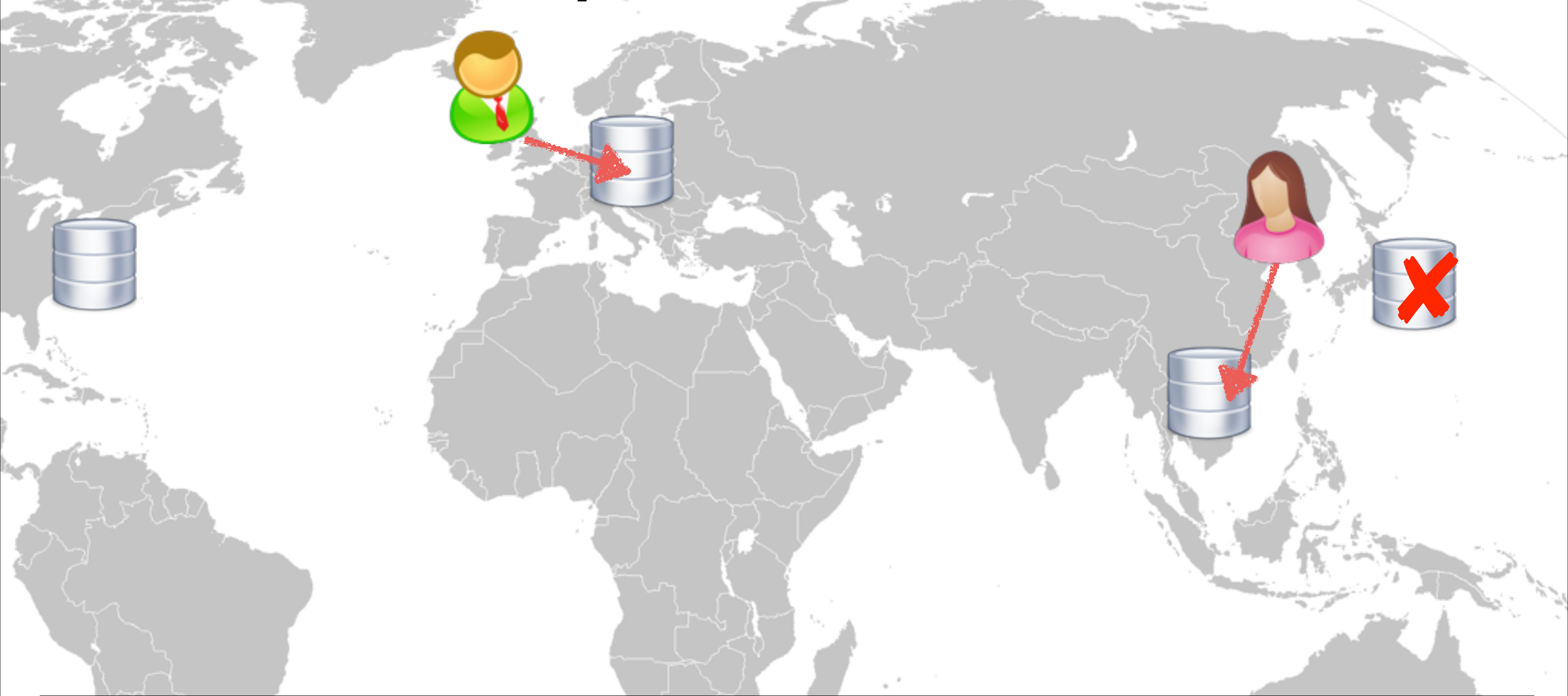
- Every data centre stores a complete replica of data
- Purpose: Minimising latency. Fault tolerance.

Geo-replicated databases



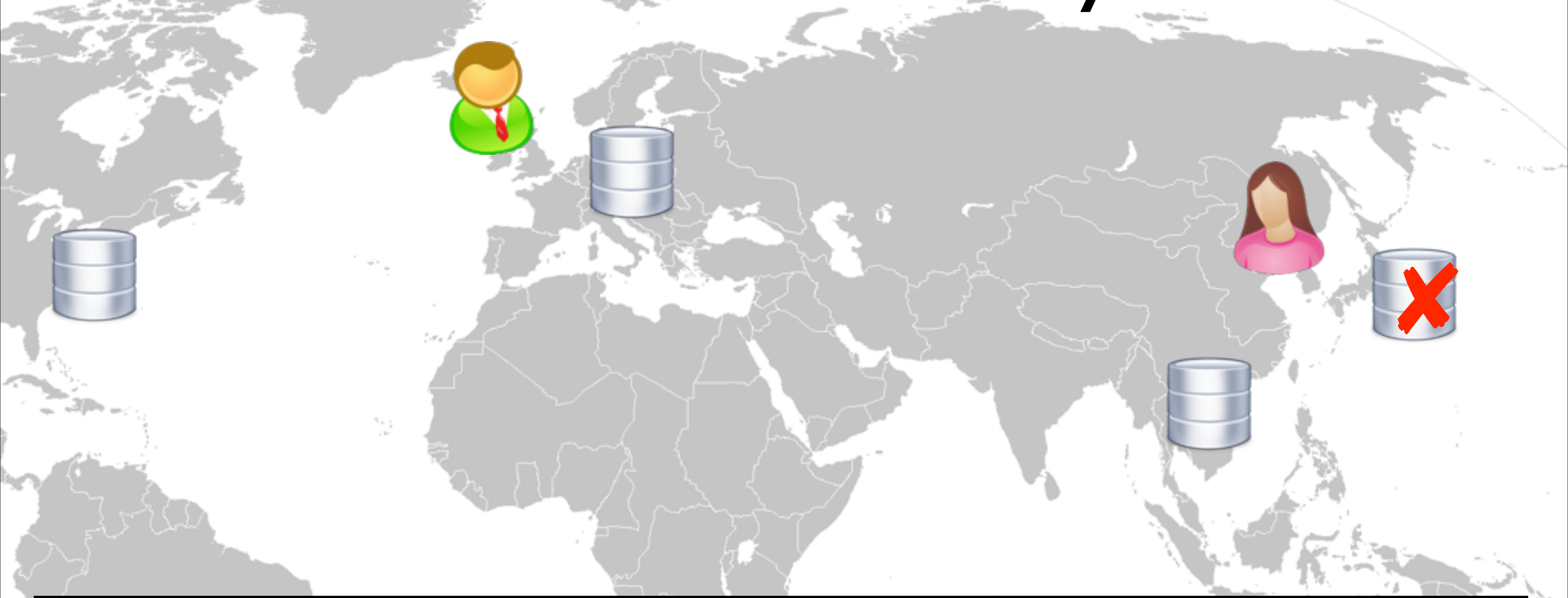
- Every data centre stores a complete replica of data
- Purpose: Minimising latency. Fault tolerance.

Geo-replicated databases



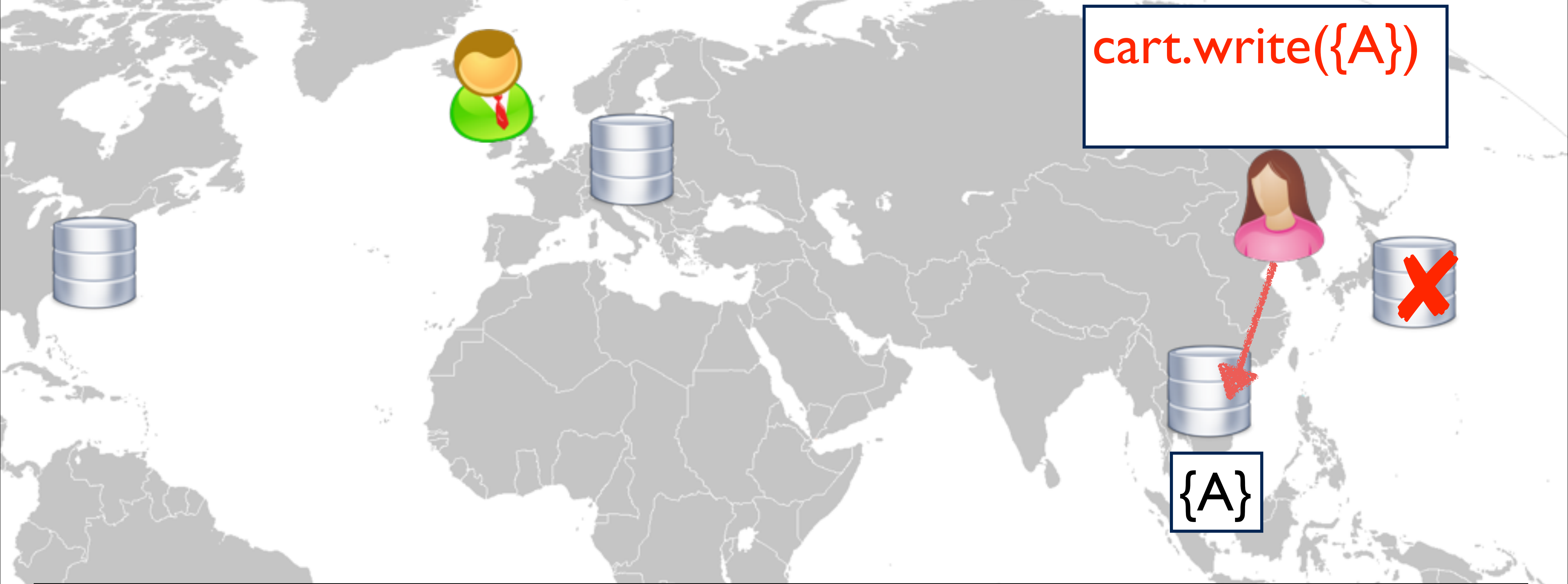
- Every data centre stores a complete replica of data
- Purpose: Minimising latency. Fault tolerance.

Data consistency



- Do data centre stores behave as if a single store?
- Strong consistency: Yes.
- Weak consistency: No.

Data consistency

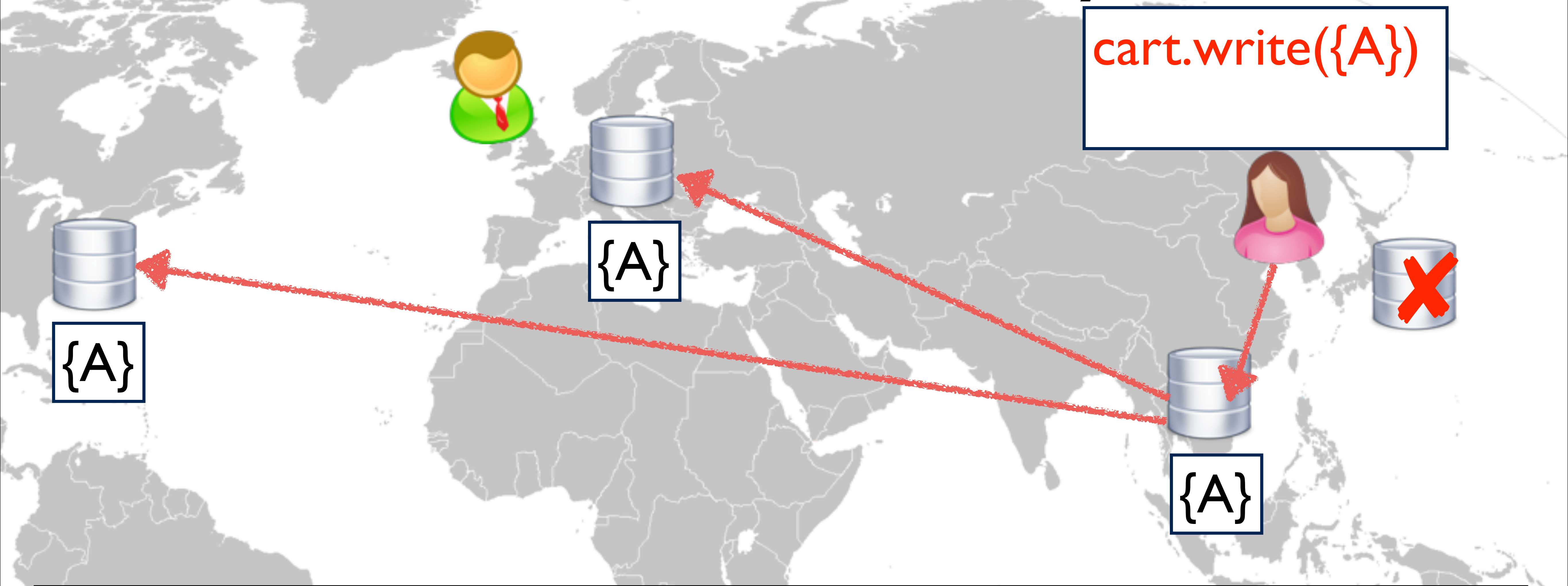


cart.write({A})

{A}

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



cart.write({A})

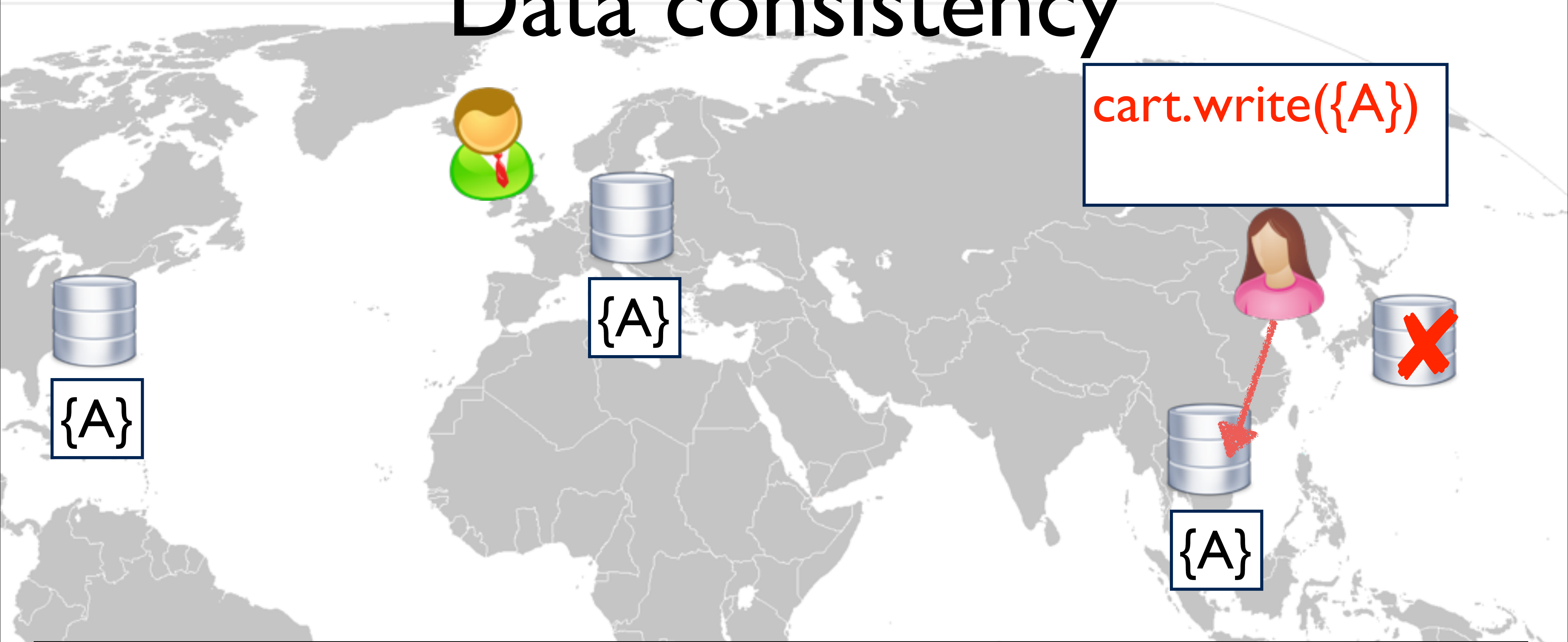
{A}

{A}

{A}

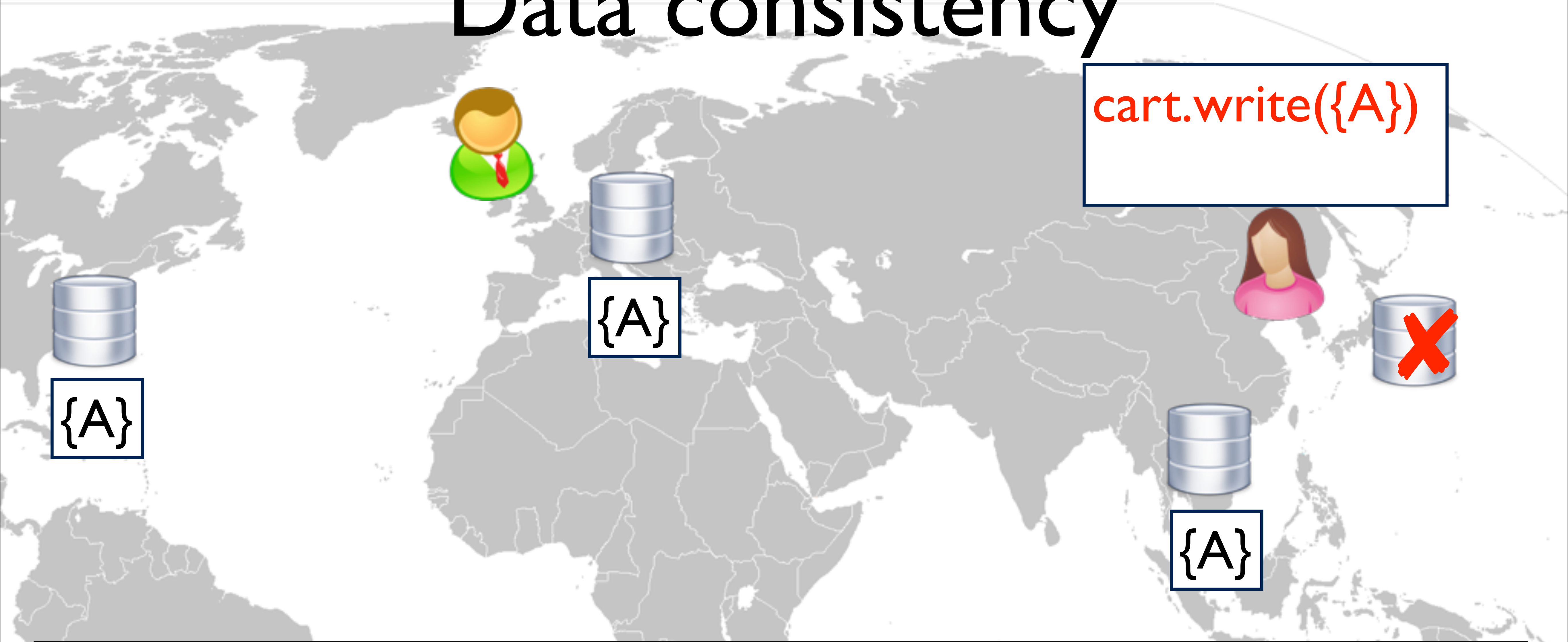
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



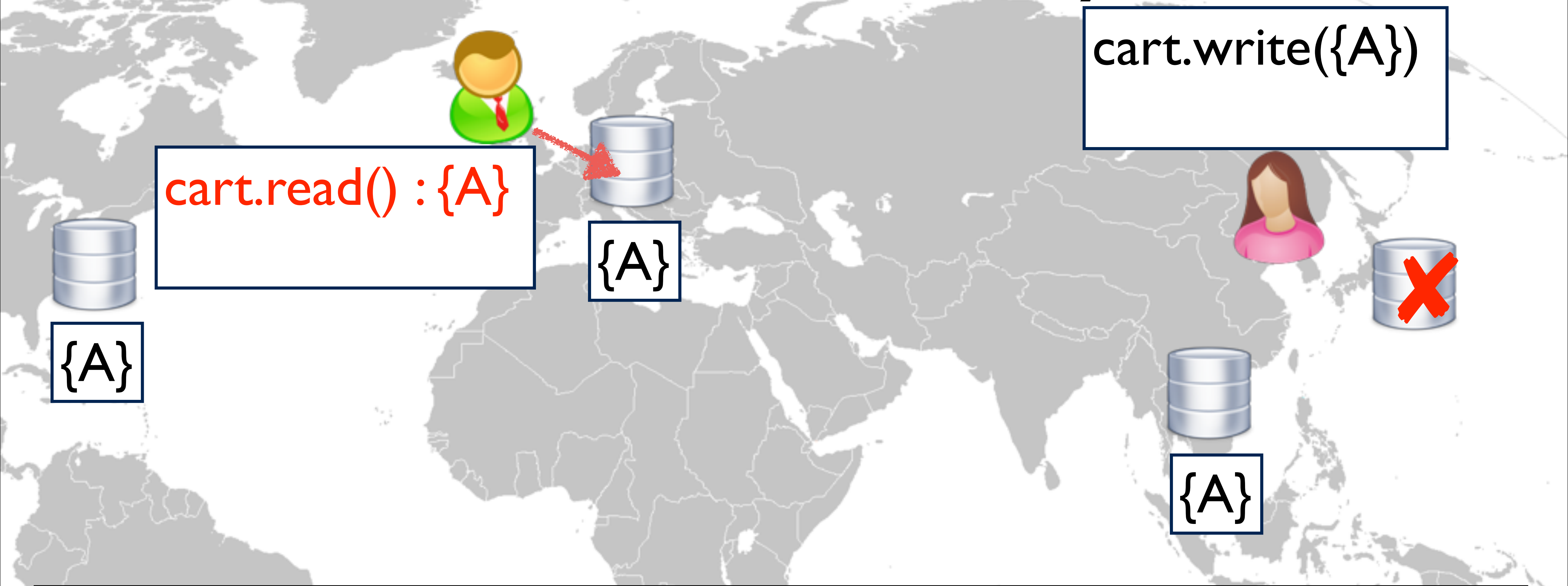
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency

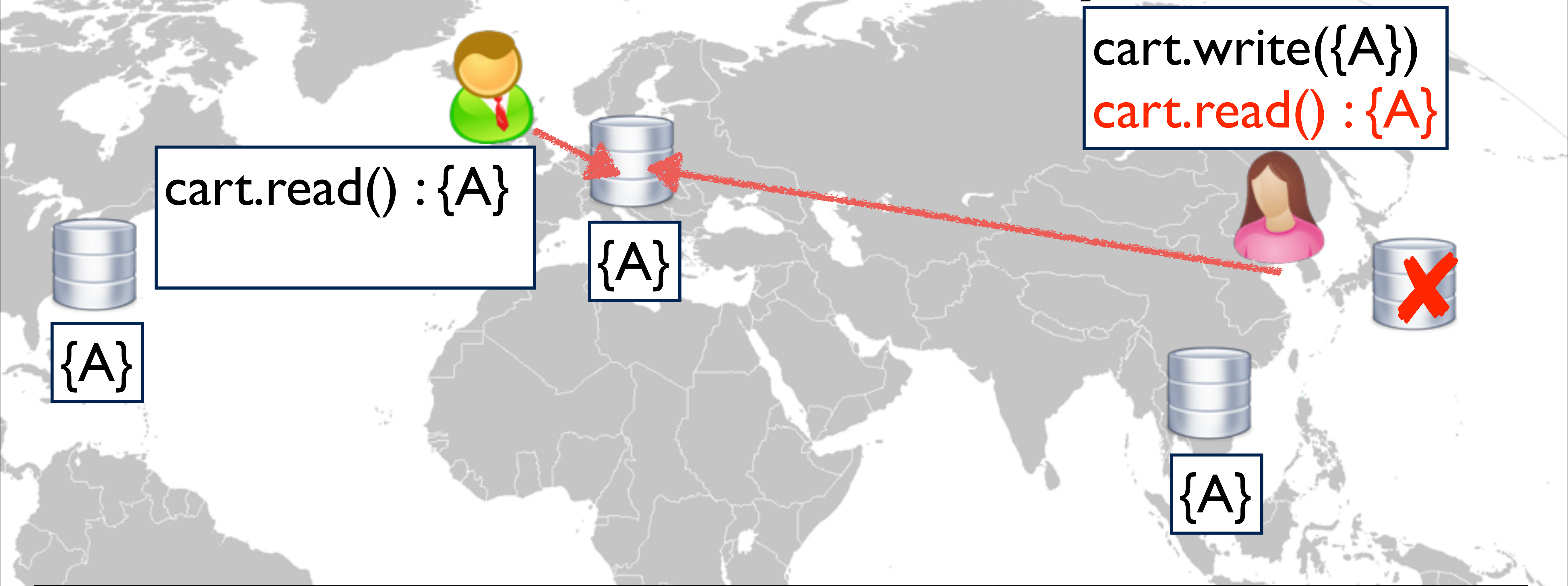


cart.write({A})

cart.read() : {A}

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



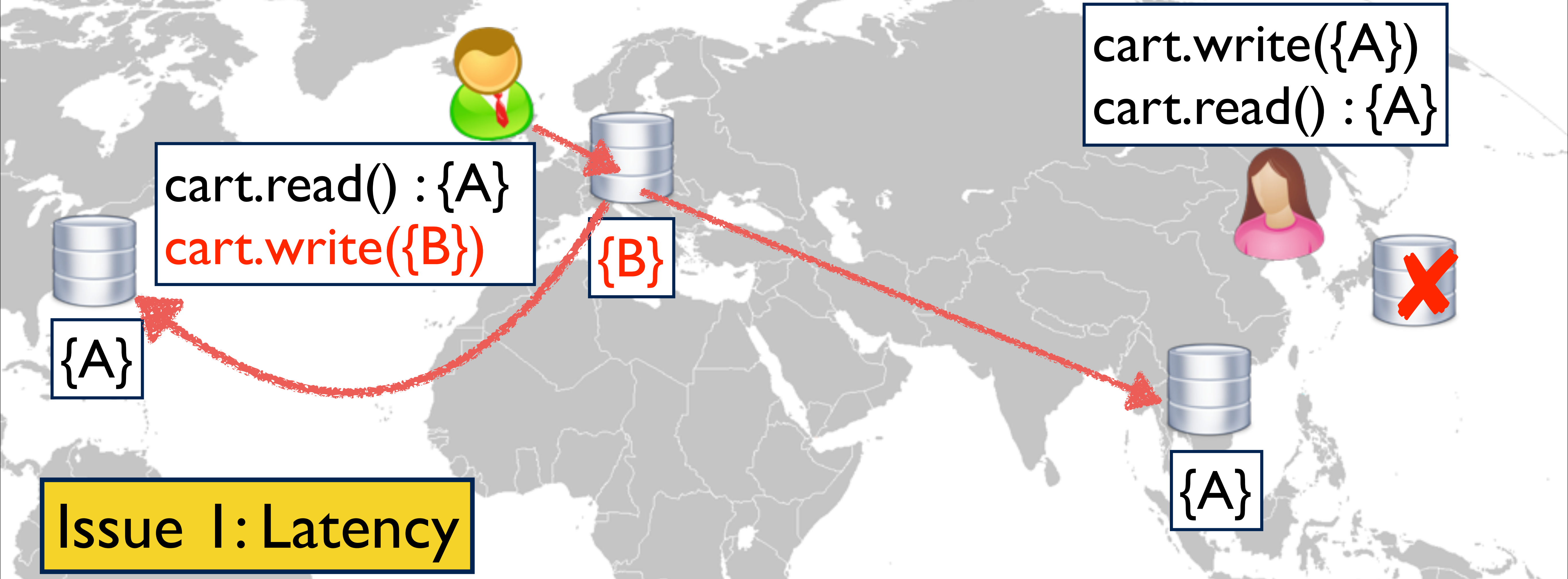
cart.write({A})

cart.read() : {A}

cart.read() : {A}

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



cart.write({A})
cart.read() : {A}

cart.read() : {A}
cart.write({B})

{A}

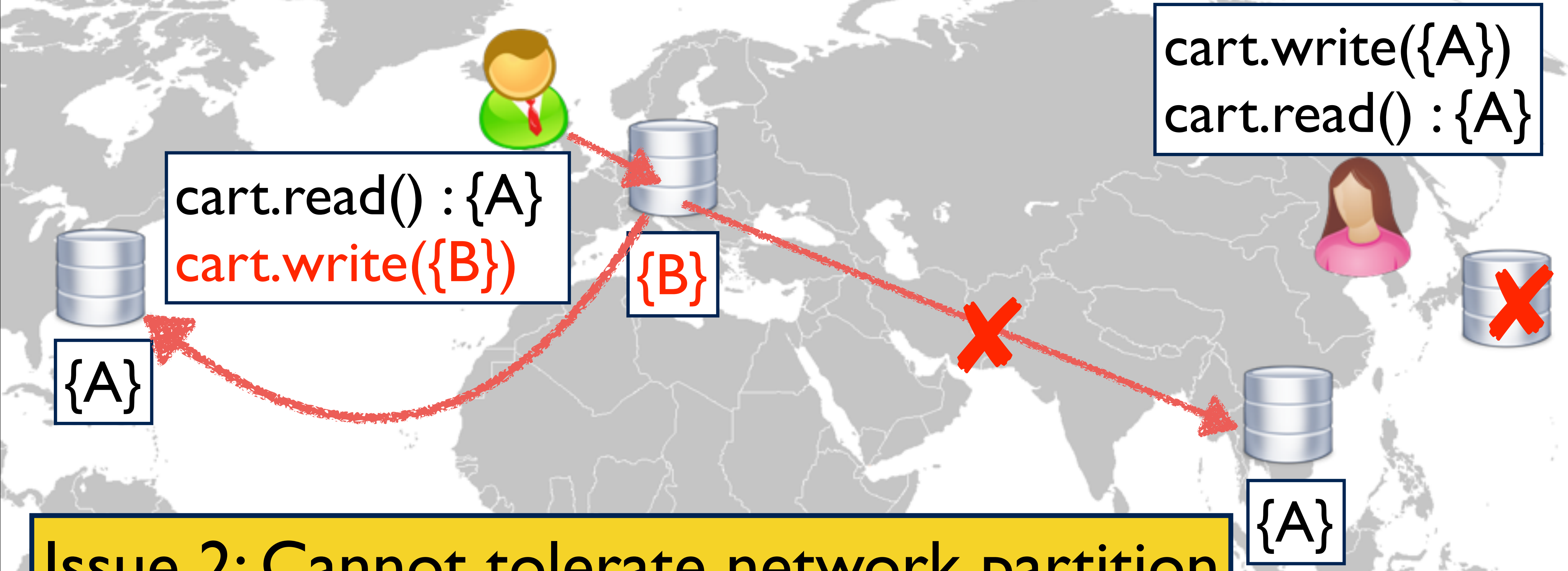
{B}

{A}

Issue 1: Latency

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



cart.write({A})
cart.read() : {A}

cart.read() : {A}
cart.write({B})

{A}

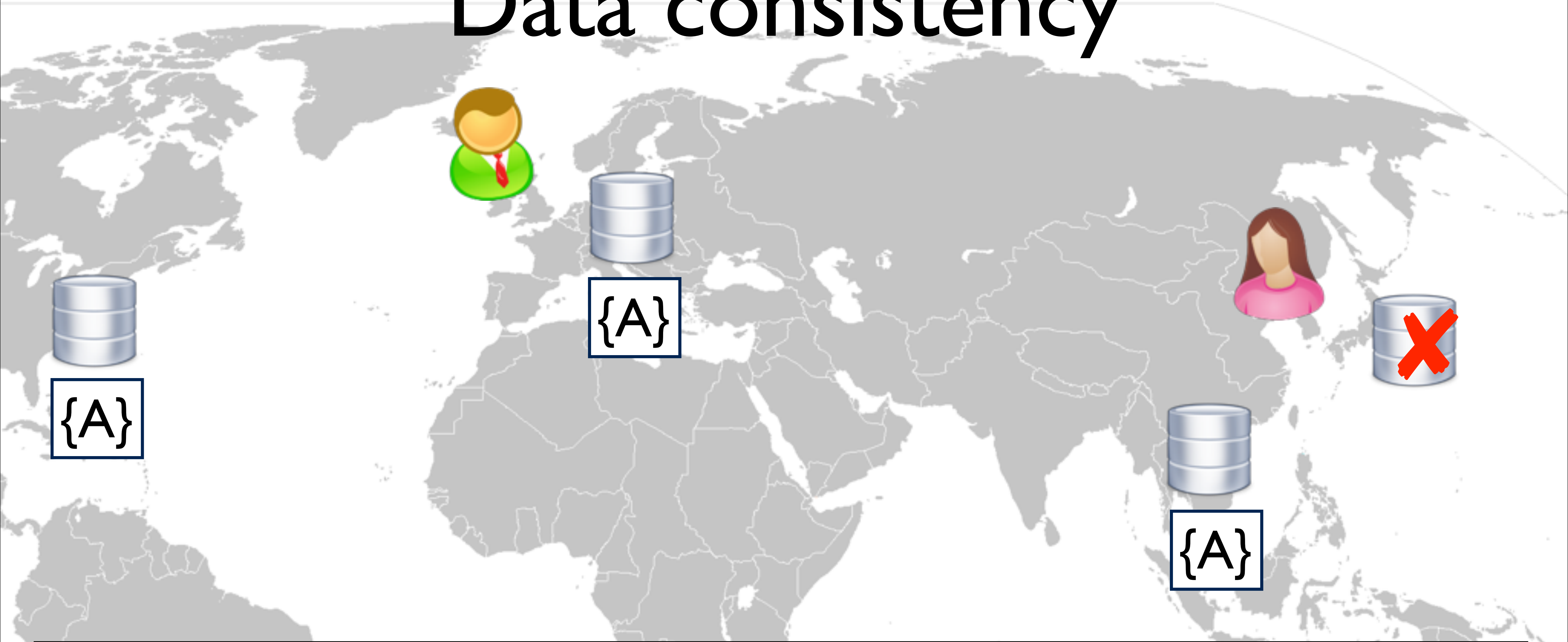
{B}

{A}

Issue 2: Cannot tolerate network partition

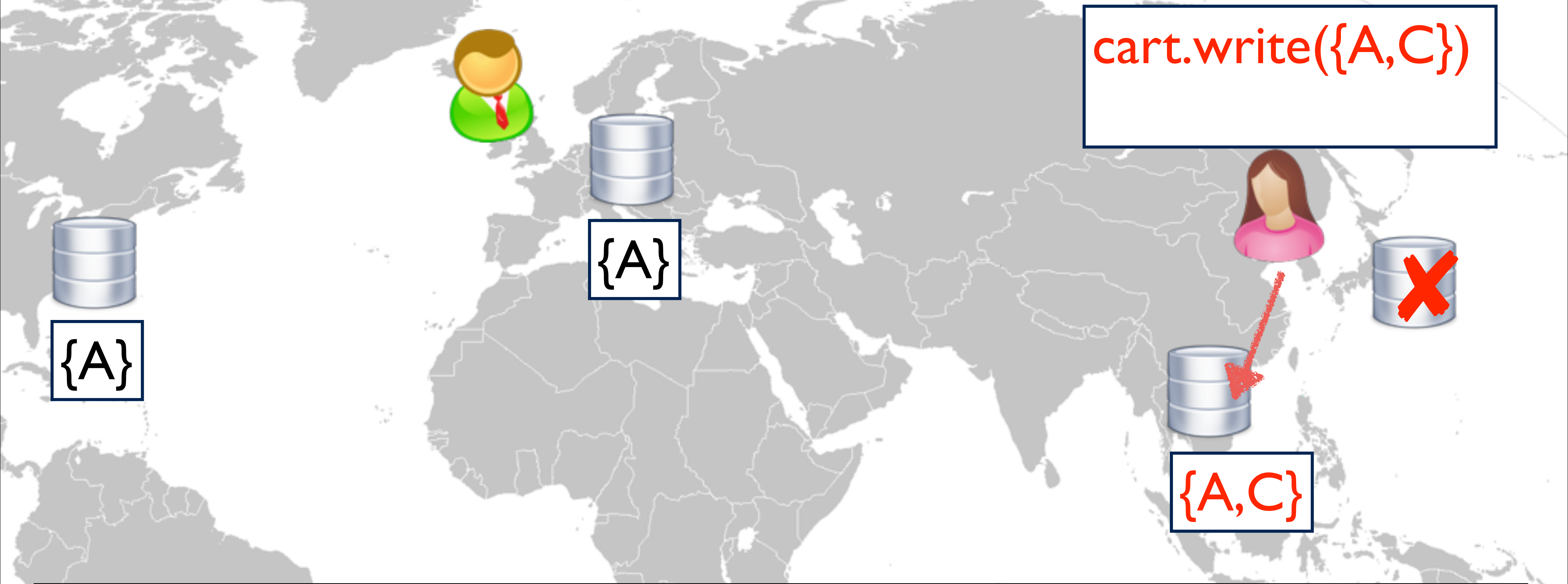
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No.

Data consistency



- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



cart.write({A,C})

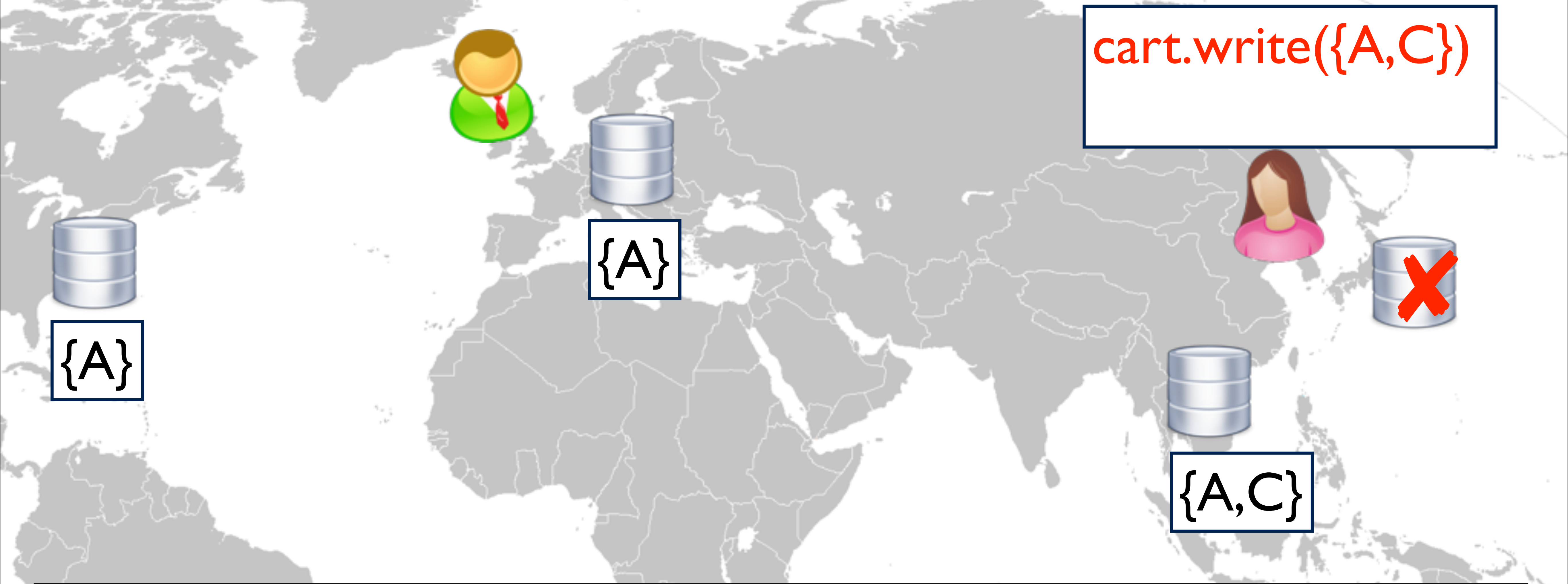
{A}

{A}

{A,C}

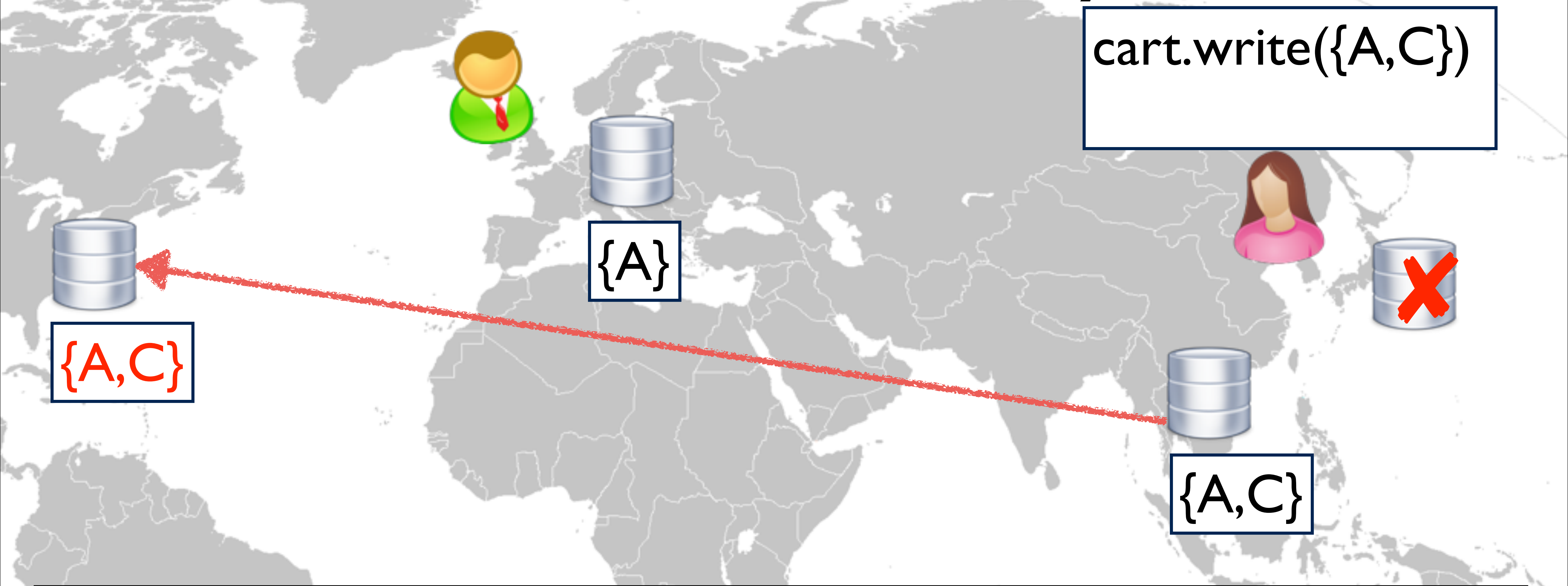
- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



cart.write({A,C})

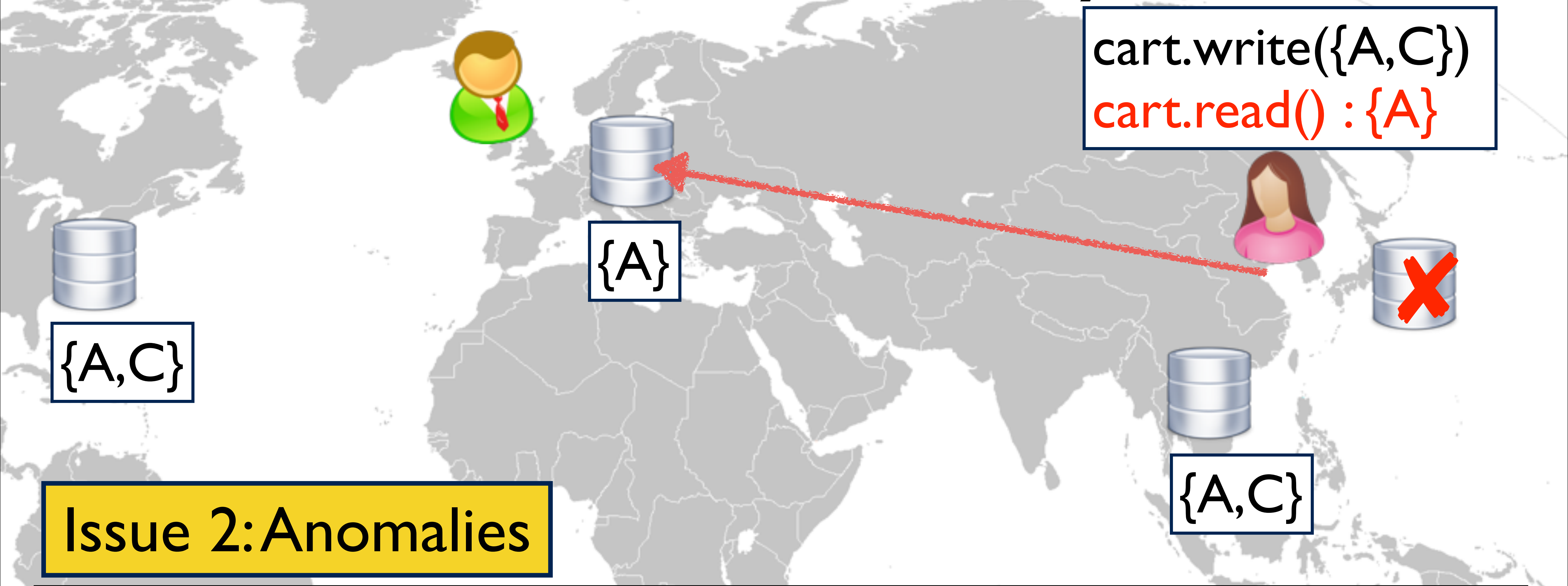
{A}

{A,C}

{A,C}

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



cart.write({A,C})
cart.read() : {A}

{A}

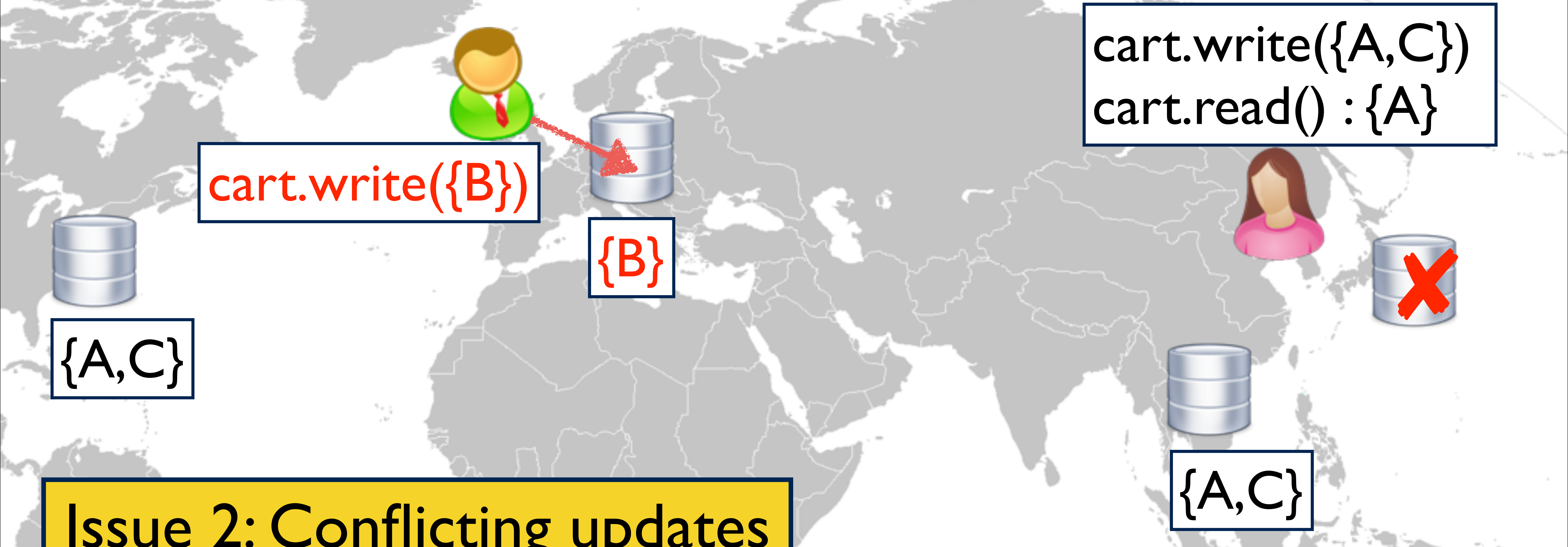
{A,C}

{A,C}

Issue 2: Anomalies

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

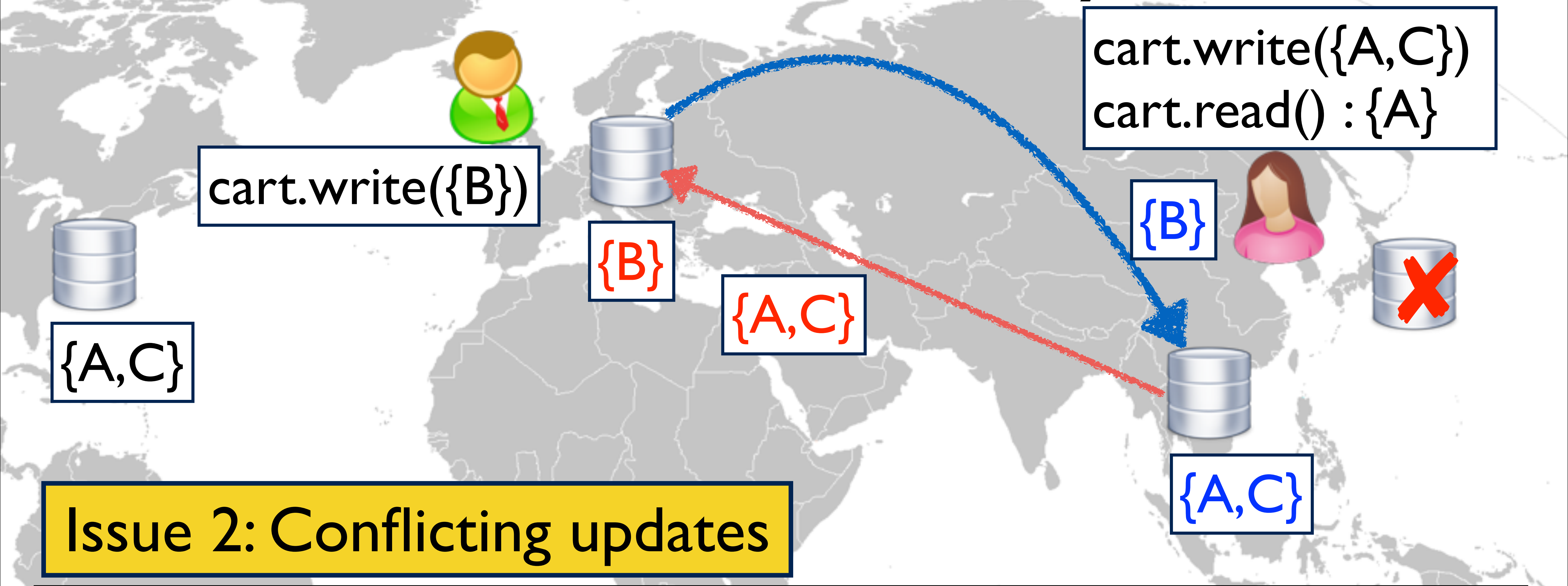
Data consistency



Issue 2: Conflicting updates

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Data consistency



Issue 2: Conflicting updates

- Do data centre stores behave as if a single store?
- Strong consistency: Yes. Block until all get updated.
- Weak consistency: No. First update. Then propagate.

Popularity of weak consistency

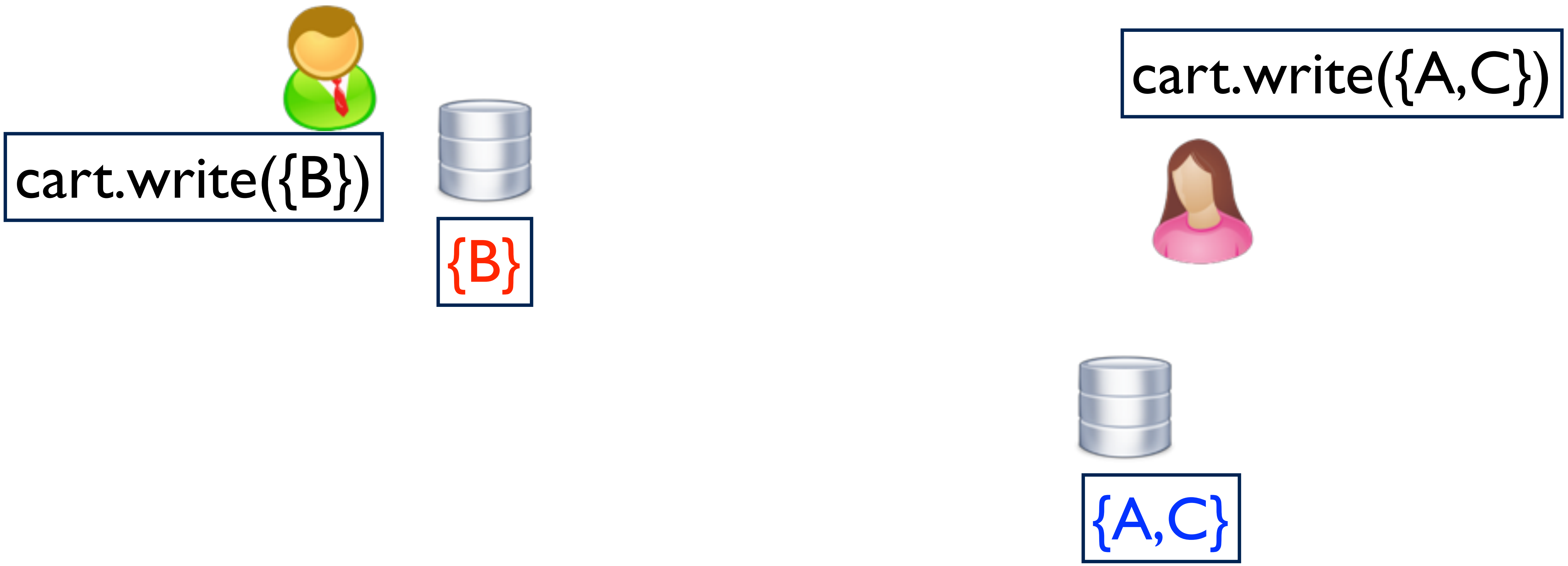
- Dynamo/Simple DB (Amazon), Riak, Cassandra (Facebook/Twitter), CouchDB, Google Doc,etc
- Due to better responsiveness and availability.
- CAP theorem [Brewer, Lynch&Gilbert]: Impossible to achieve all of strong consistency, availability and tolerance to network partition.

Eventual consistency

- One of the weakest consistency conditions.
- Intuitively, it says that replicas never diverge.
- Requires conflict detection and resolution.

Eventual consistency

- One of the weakest consistency conditions.
- Intuitively, it says that replicas never diverge.
- Requires conflict detection and resolution.



cart.write({B})

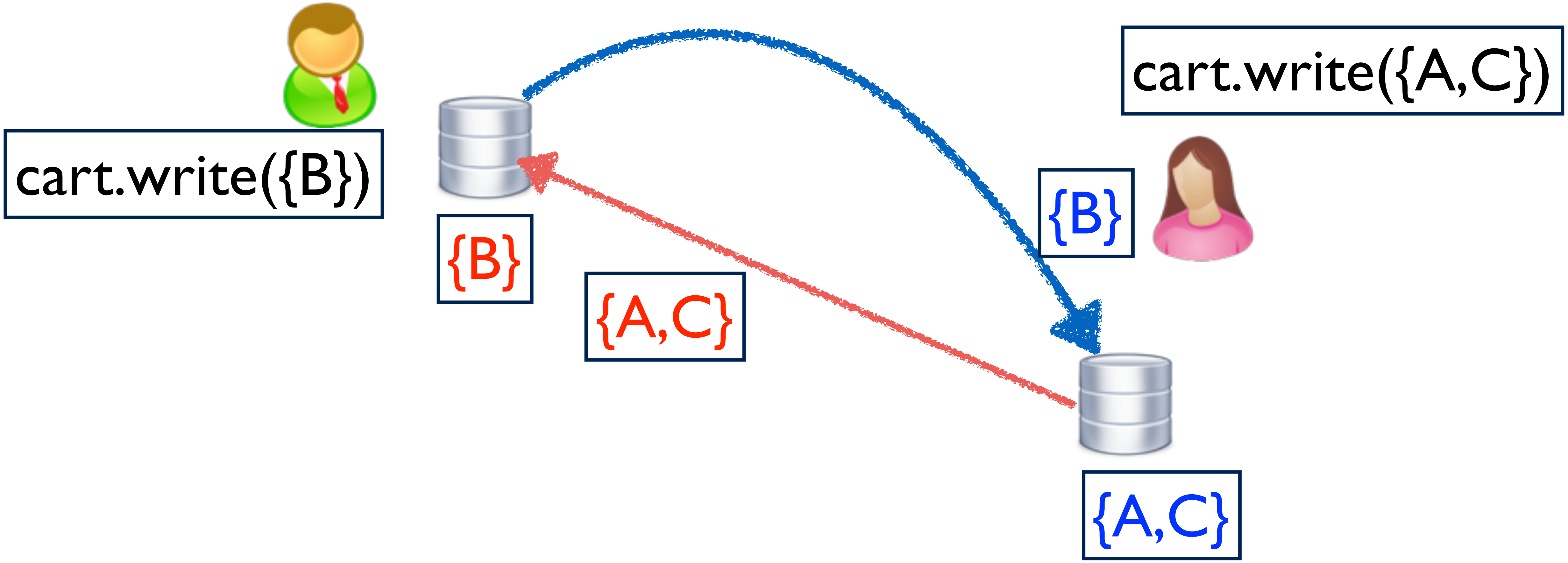
{B}

cart.write({A,C})

{A,C}

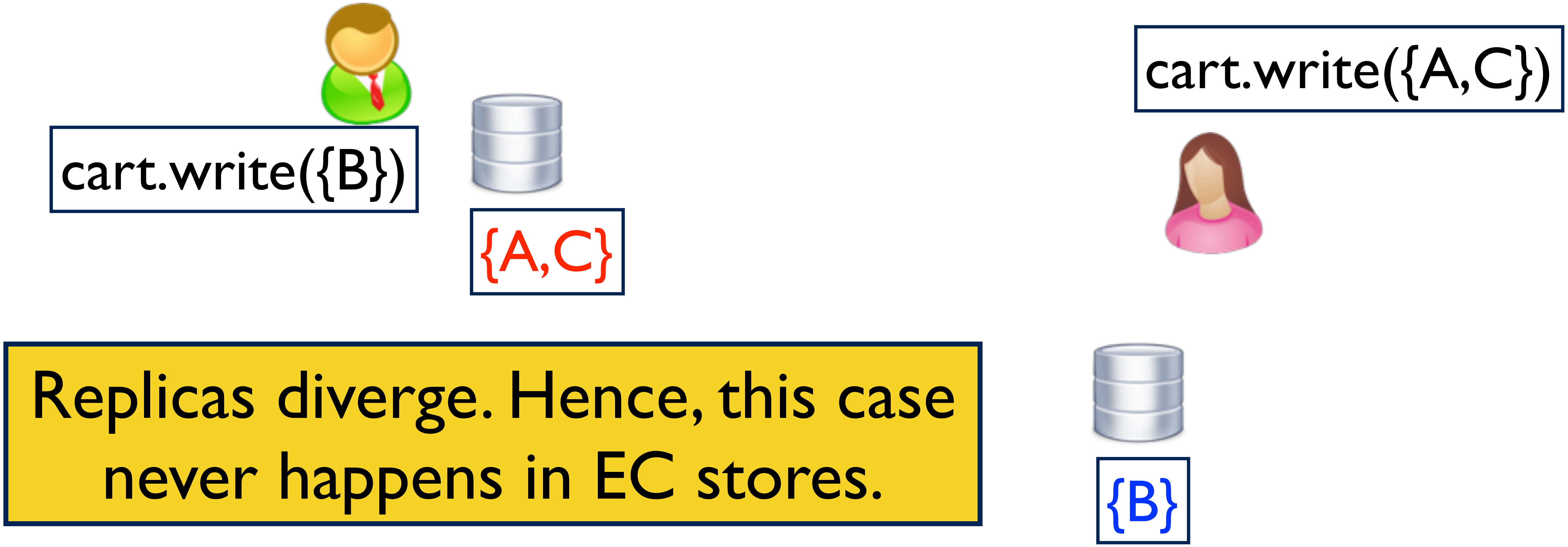
Eventual consistency

- One of the weakest consistency conditions.
- Intuitively, it says that replicas never diverge.
- Requires conflict detection and resolution.



Eventual consistency

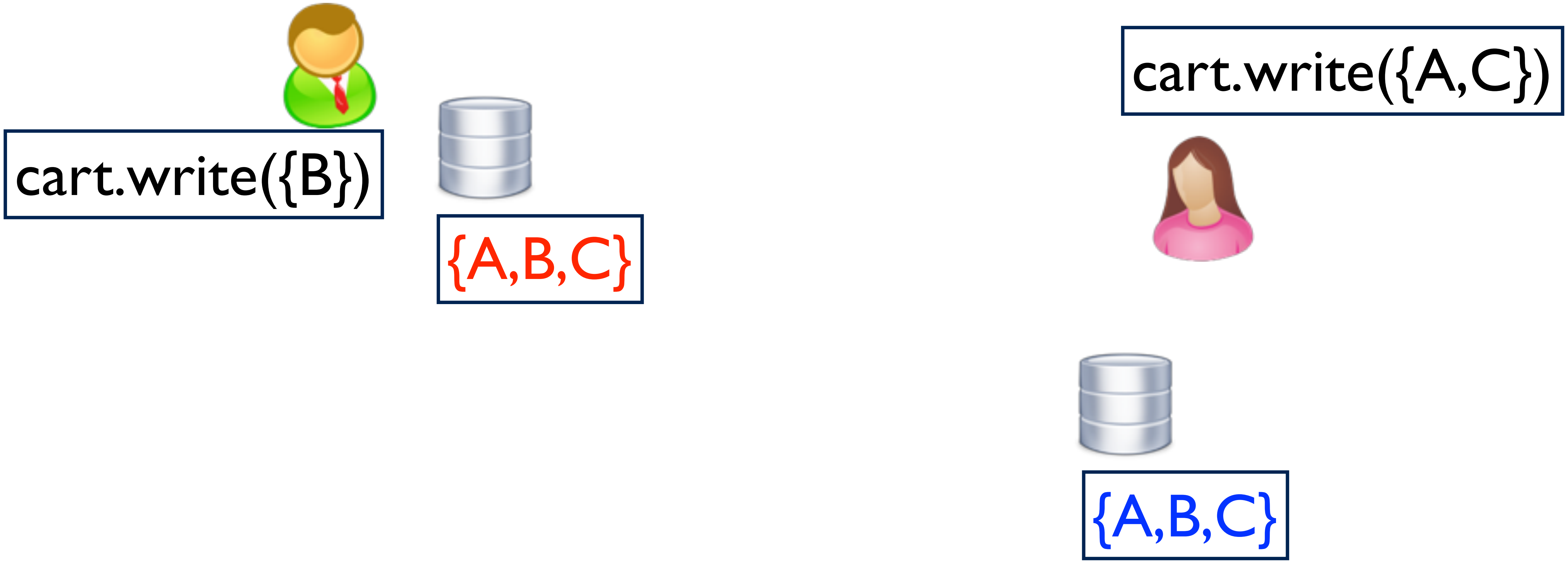
- One of the weakest consistency conditions.
- Intuitively, it says that replicas never diverge.
- Requires conflict detection and resolution.



Replicas diverge. Hence, this case never happens in EC stores.

Eventual consistency

- One of the weakest consistency conditions.
- Intuitively, it says that replicas never diverge.
- Requires conflict detection and resolution.



cart.write({B})

{A,B,C}

cart.write({A,C})

{A,B,C}

Challenge

To develop correct, efficient distributed applications on top of EC stores.

Research opportunities for EC

- Formalisation and foundation.
- New EC store that provides different levels of consistency on demand.
- Design a new programming language.
- Static analysis.
- Efficient data structures with intuitive conflict resolution policies.

Research opportunities for EC

- Formalisation and foundation.
- New EC store that provides different levels of consistency on demand.
- Design a new programming language.
- Static analysis.
- Efficient data structures with intuitive conflict resolution policies.

Convergence property.

If no new updates are made to the store, then replicas will eventually reach a consistent state.

practice

DOI:10.1145/1435417.1435432

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

BY WERNER VOGELS

Eventually Consistent

AT THE FOUNDATION of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost-effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when many widespread distributed systems provide an *eventual consistency* model in the context of data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. Here, I present some of the relevant background that has informed our approach to delivering reliable distributed systems that must operate on a global scale. (An earlier version of this article appeared as a posting on the "All Things Distributed" Weblog and was greatly improved with the help of its readers.)

Historical Perspective

In an ideal world there would be only one consistency model: when an update is made all observers would see that update. The first time this surfaced as difficult to achieve was in the database systems of the late 1970s. The best "period piece" on this topic is "Notes on Distributed Databases" by Bruce Lindsay et al.[5] It lays out the fundamental principles for database replication and discusses a number of techniques that deal with achieving consistency. Many of these techniques try to achieve *distribution transparency*—that is, to the user of the system it appears as if there is only one system instead of a number of collaborating systems. Many systems during this time took the approach that it was better to fail the complete system than to break this transparency.[3]

In the mid-1990s, with the rise of larger Internet systems, these practices were revisited. At that time people began to consider the idea that availability was perhaps the most impor-

Convergence property.

If no new updates are made to the store, then replicas will eventually reach a consistent state.

But updates never stop.

Not useful for developing application programs.

practice

DOI:10.1145/1435417.1435432

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

BY WERNER VOGELS

Eventually Consistent

AT THE FOUNDATION of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost-effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

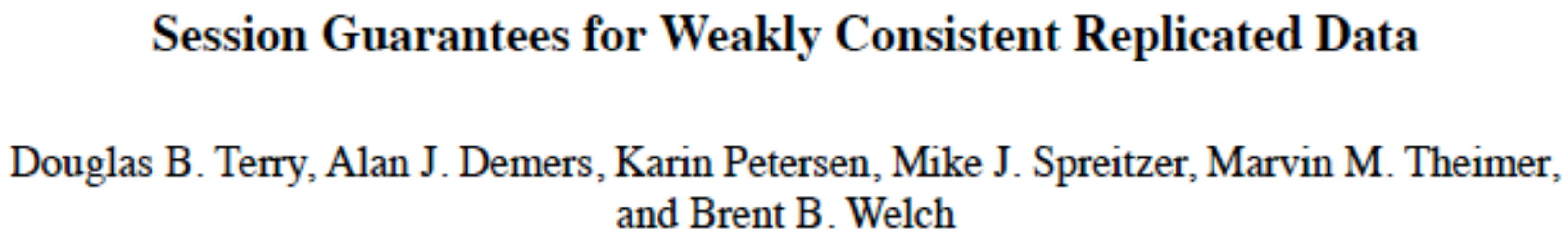
One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when many widespread distributed systems provide an *eventual consistency* model in the context of data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. Here, I present some of the relevant background that has informed our approach to delivering reliable distributed systems that must operate on a global scale. (An earlier version of this article appeared as a posting on the "All Things Distributed" Weblog and was greatly improved with the help of its readers.)

Historical Perspective

In an ideal world there would be only one consistency model: when an update is made all observers would see that update. The first time this surfaced as difficult to achieve was in the database systems of the late 1970s. The best "period piece" on this topic is "Notes on Distributed Databases" by Bruce Lindsay et al.[5] It lays out the fundamental principles for database replication and discusses a number of techniques that deal with achieving consistency. Many of these techniques try to achieve *distribution transparency*—that is, to the user of the system it appears as if there is only one system instead of a number of collaborating systems. Many systems during this time took the approach that it was better to fail the complete system than to break this transparency.[2]

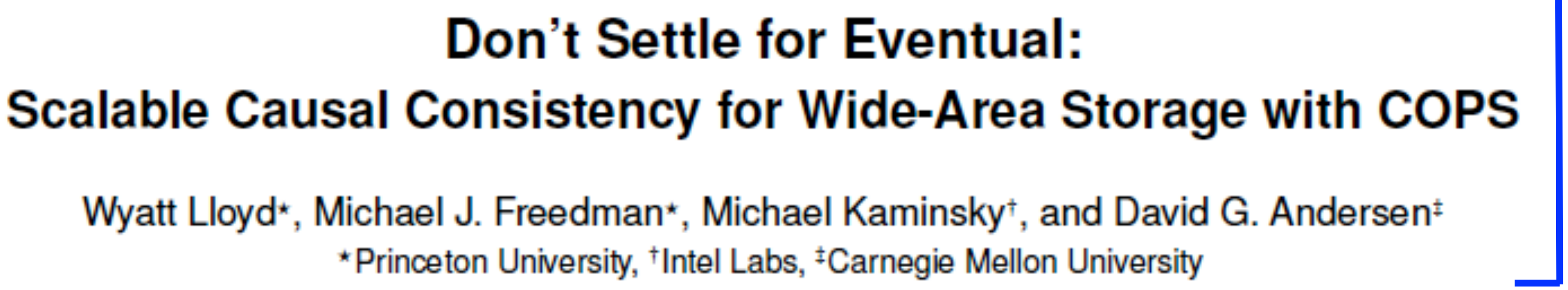
In the mid-1990s, with the rise of larger Internet systems, these practices were revisited. At that time people began to consider the idea that availability was perhaps the most impor-

50 shades of eventual consistency



Session Guarantees for Weakly Consistent Replicated Data

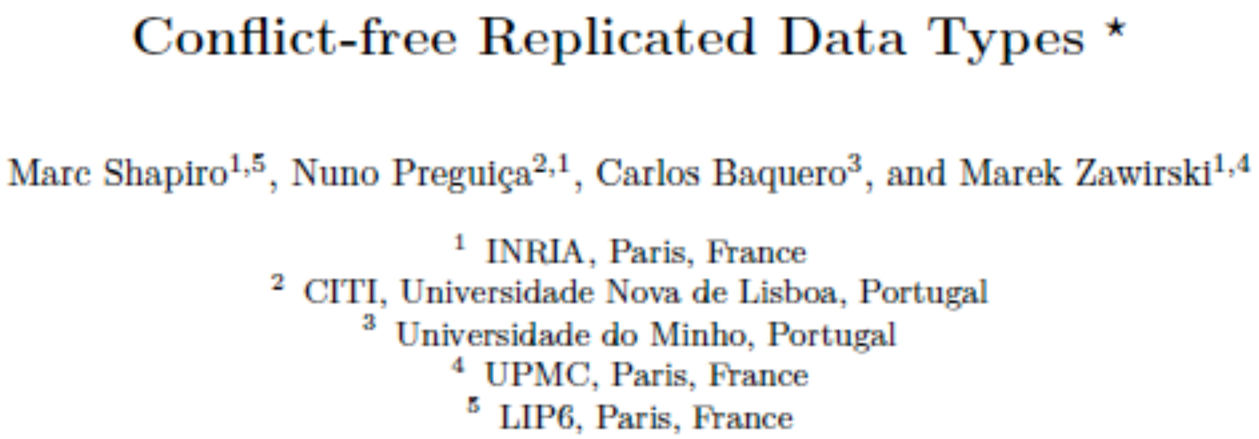
Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer,
and Brent B. Welch



Don't Settle for Eventual:
Scalable Causal Consistency for Wide-Area Storage with COPS

Wyatt Lloyd[*], Michael J. Freedman[*], Michael Kaminsky[†], and David G. Andersen[‡]
[*]Princeton University, [†]Intel Labs, [‡]Carnegie Mellon University

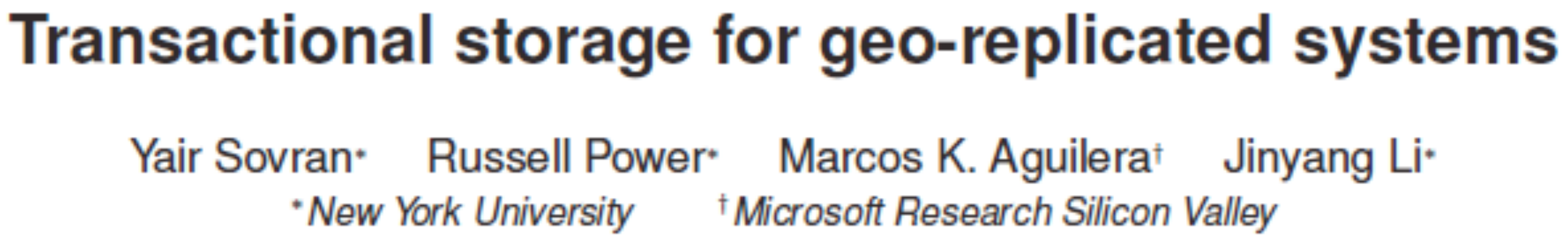
Strengthen consistency, somewhat.



Conflict-free Replicated Data Types *

Marc Shapiro[1,5], Nuno Preguiça[2,1], Carlos Baquero[3], and Marek Zawirski[1,4]

[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France

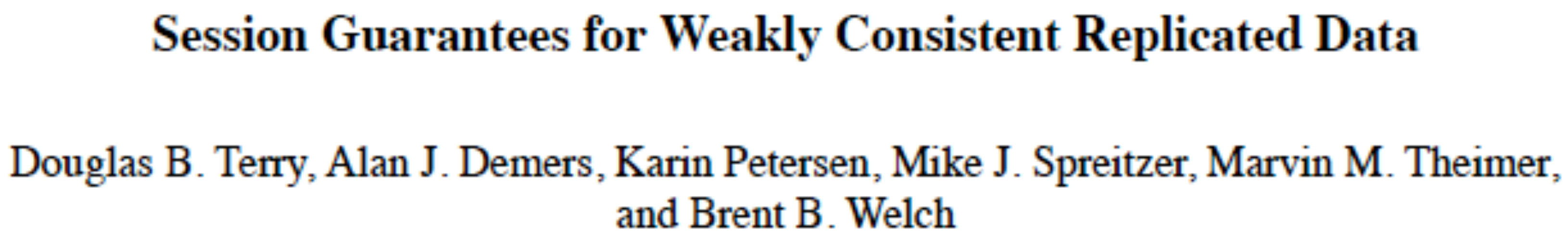


Transactional storage for geo-replicated systems

Yair Sovran[*] Russell Power[*] Marcos K. Aguilera[†] Jinyang Li[*]
[*]*New York University* [†]*Microsoft Research Silicon Valley*

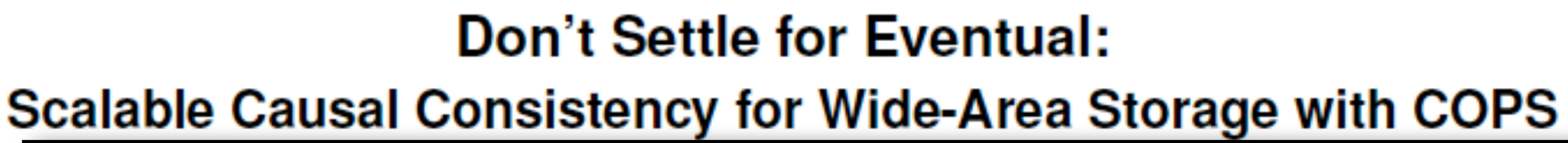
Add features that make coping with weak consistency easier.

50 shades of eventual consistency

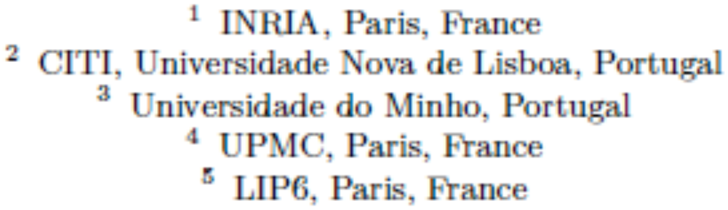


Session Guarantees for Weakly Consistent Replicated Data

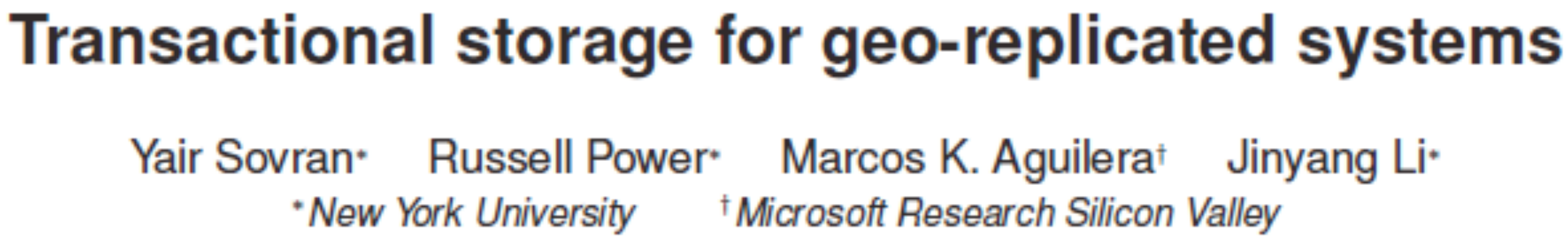
Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch



Don’t Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS



[1] INRIA, Paris, France
[2] CITI, Universidade Nova de Lisboa, Portugal
[3] Universidade do Minho, Portugal
[4] UPMC, Paris, France
[5] LIP6, Paris, France



Transactional storage for geo-replicated systems

Yair Sovran* Russell Power* Marcos K. Aguilera† Jinyang Li*
*New York University †Microsoft Research Silicon Valley

Strengthen consistency, somewhat

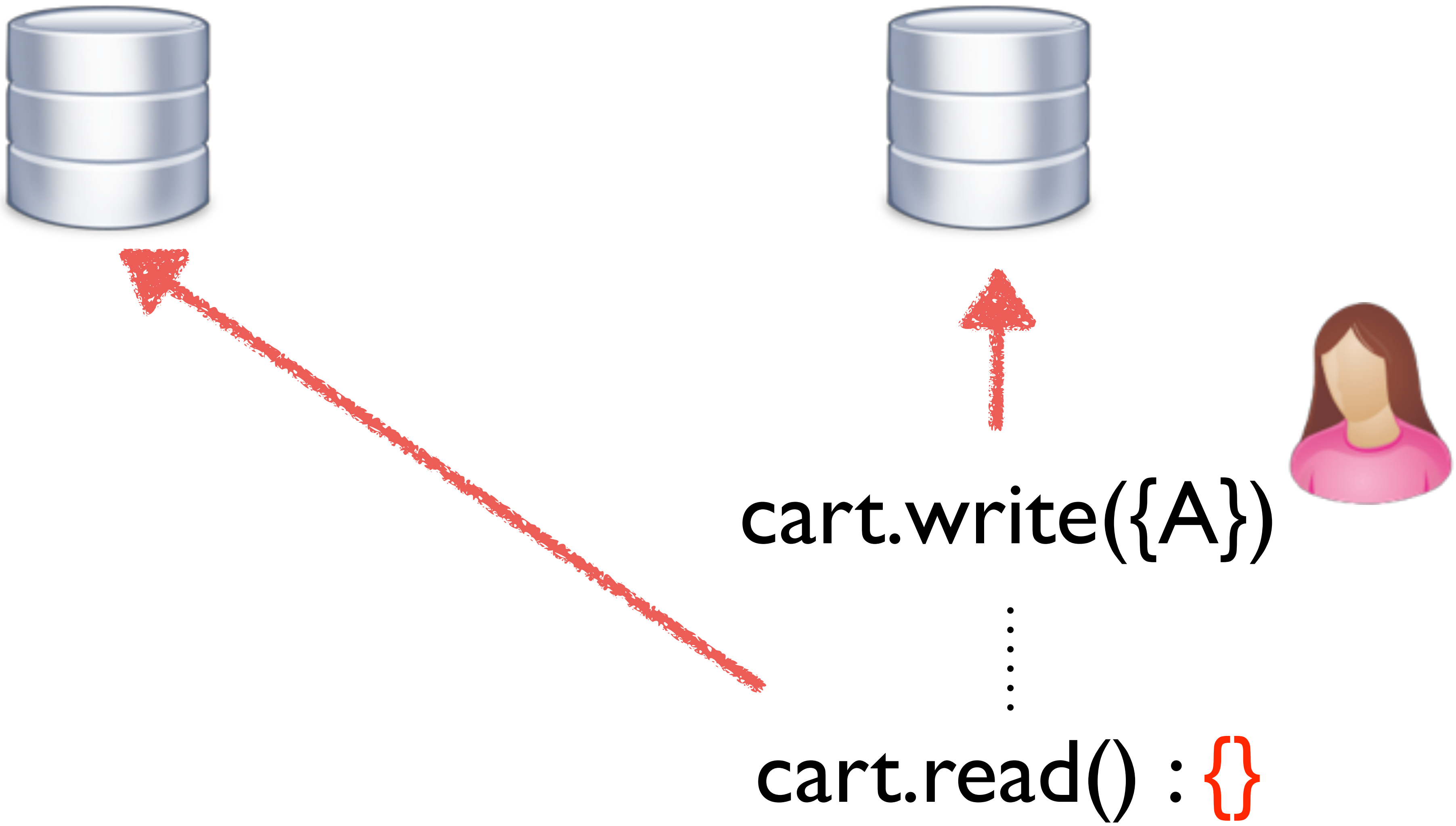
Add features that make coping with weak consistency easier

- Very low-level formalisms → difficult reasoning.
- Wildly different → hard to explore design space.

Key issues beyond ‘eventual’

1. Database eventually consistent if updates stop.

Which anomalies can we see before this?



cart.write({A})

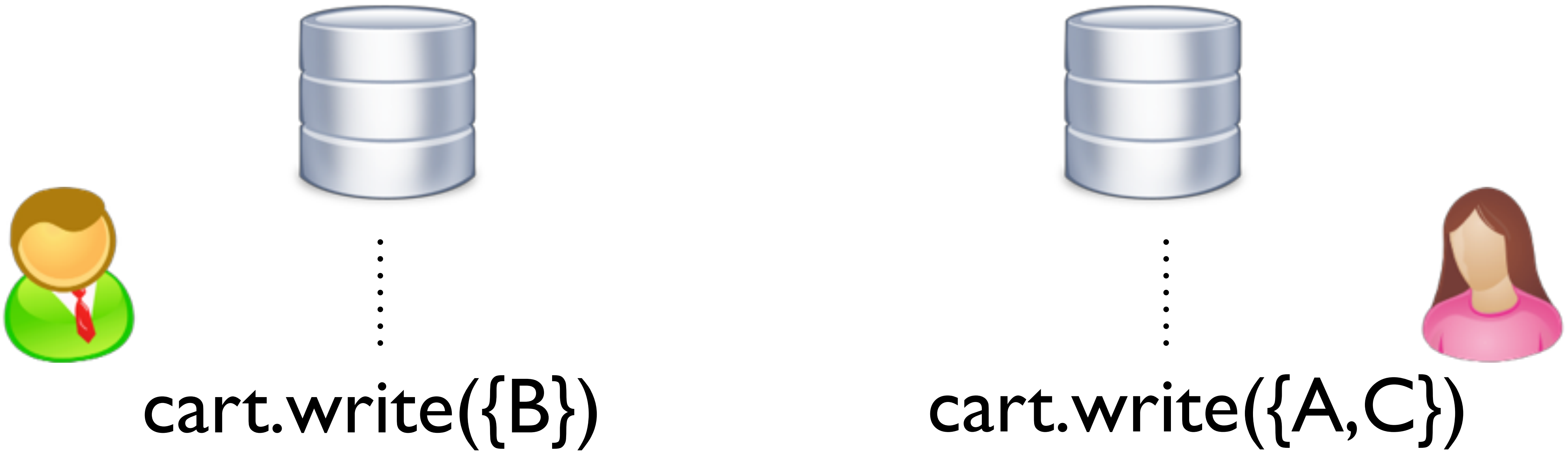
cart.read() : {}

Key issues beyond ‘eventual’

2. Users can make conflicting updates at different replicas.

How do we resolve conflicts?

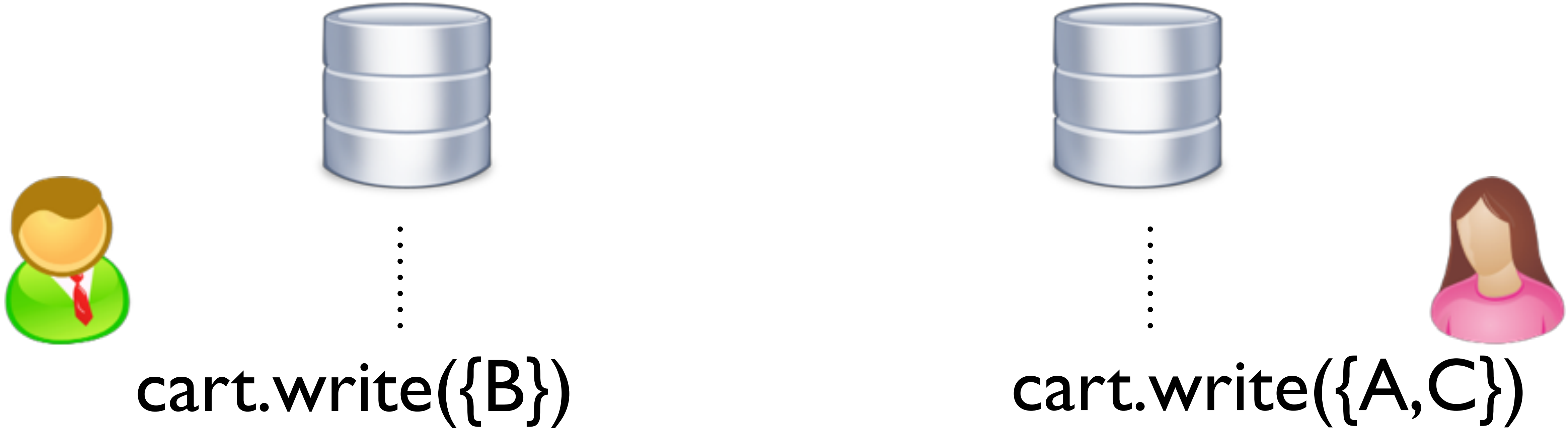
= Which state do the replicas converge to?



cart.write({B})

cart.write({A,C})

Divergence problem

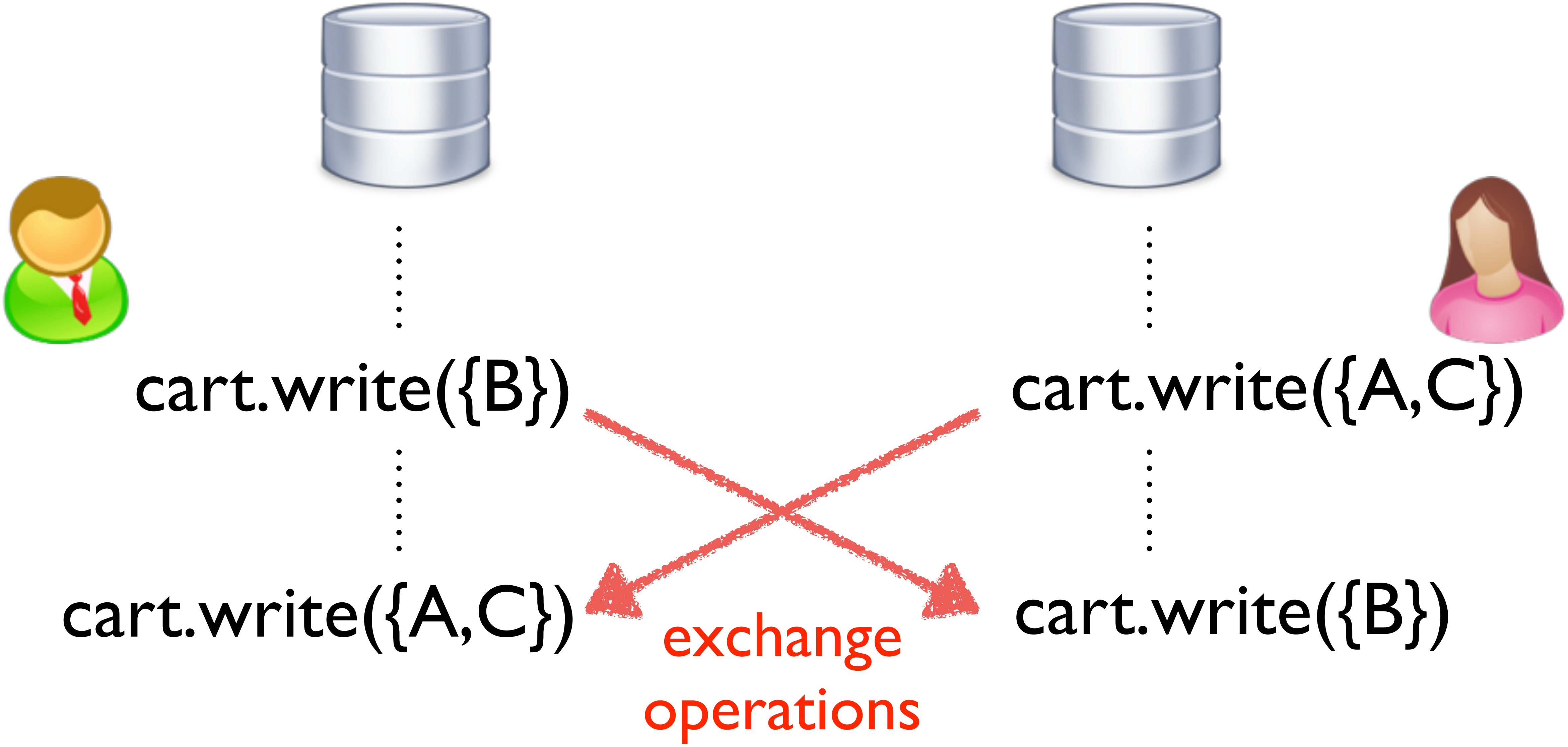


cart.write({B})

cart.write({A,C})

Update the replica you are connected to now, propagate to others later.

Divergence problem



cart.write({B})

cart.write({A,C})

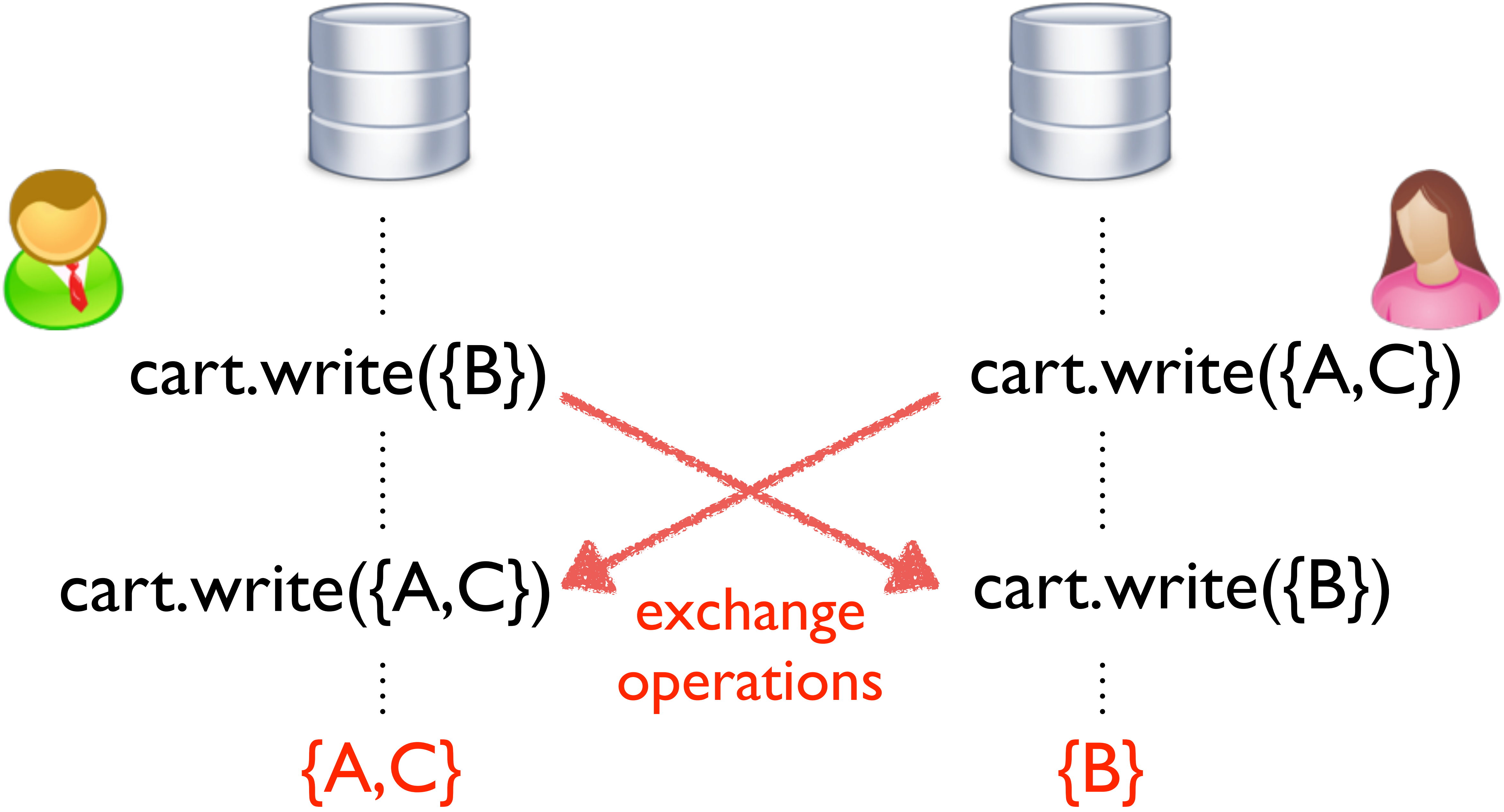
cart.write({A,C})

cart.write({B})

exchange operations

Update the replica you are connected to now, propagate to others later.

Divergence problem



cart.write({B})

cart.write({A,C})

cart.write({A,C})

cart.write({B})

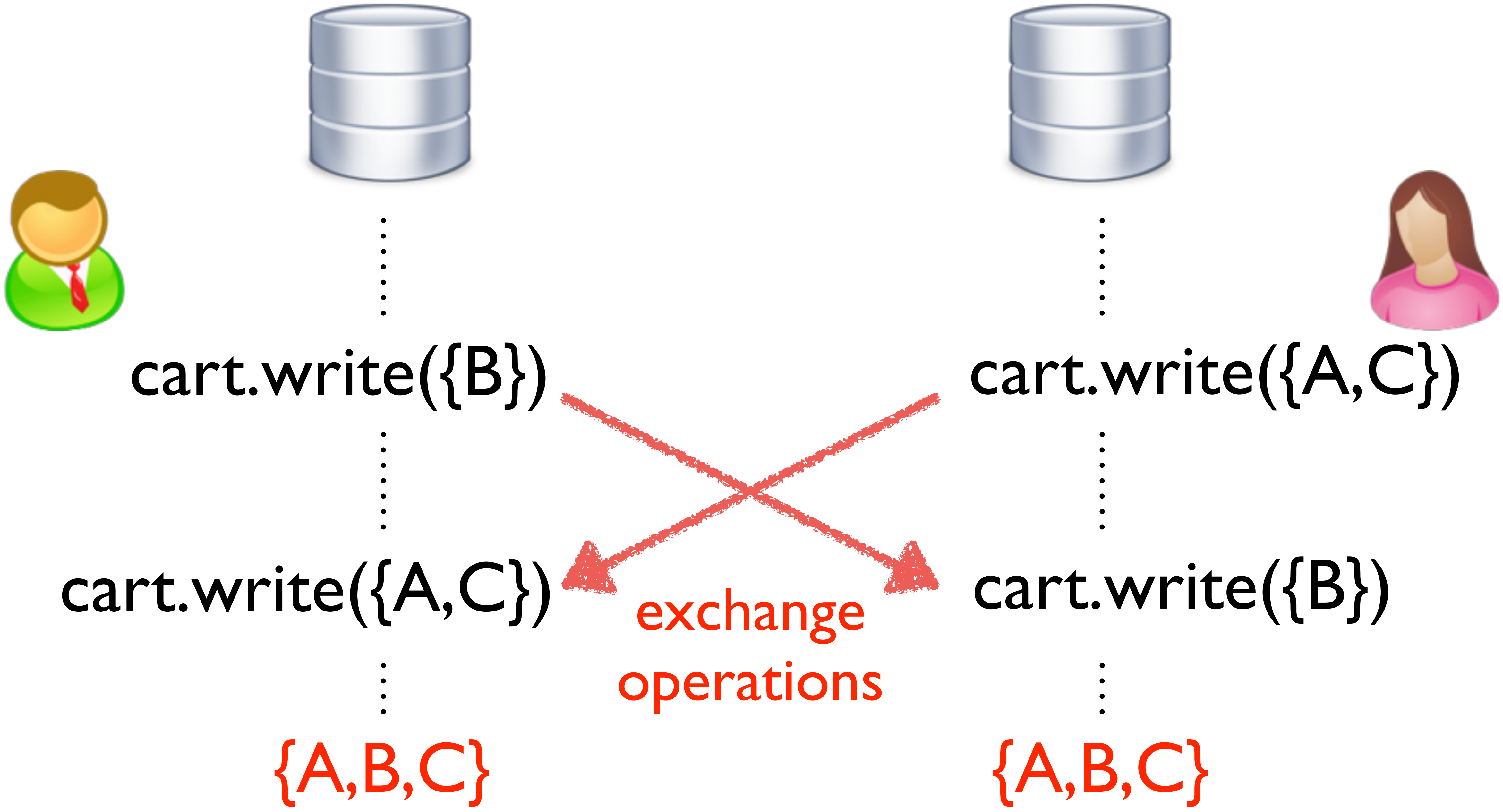
exchange operations

{A,C}

{B}

Update the replica you are connected to now, propagate to others later.

Multi-value policy



cart.write({B})

cart.write({A,C})

cart.write({A,C})

cart.write({B})

exchange operations

{A,B,C}

{A,B,C}

Keep all the values. In this case, take the union.

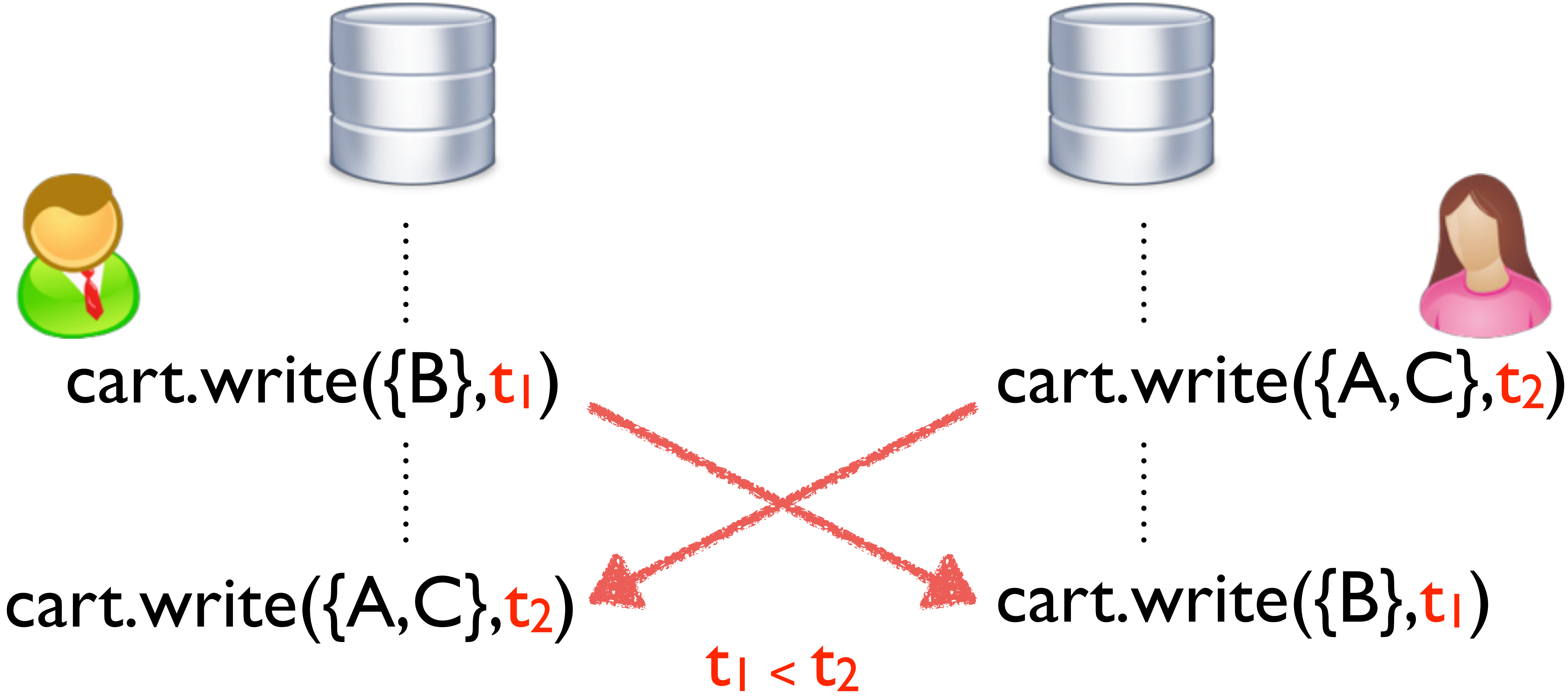
Last-writer-win policy

cart.write({B},t_1)

cart.write({A,C},t_2)

Use a distributed global clock, such as Lamport clock.

Last-writer-win policy



cart.write({B},t_1)

cart.write({A,C},t_2)

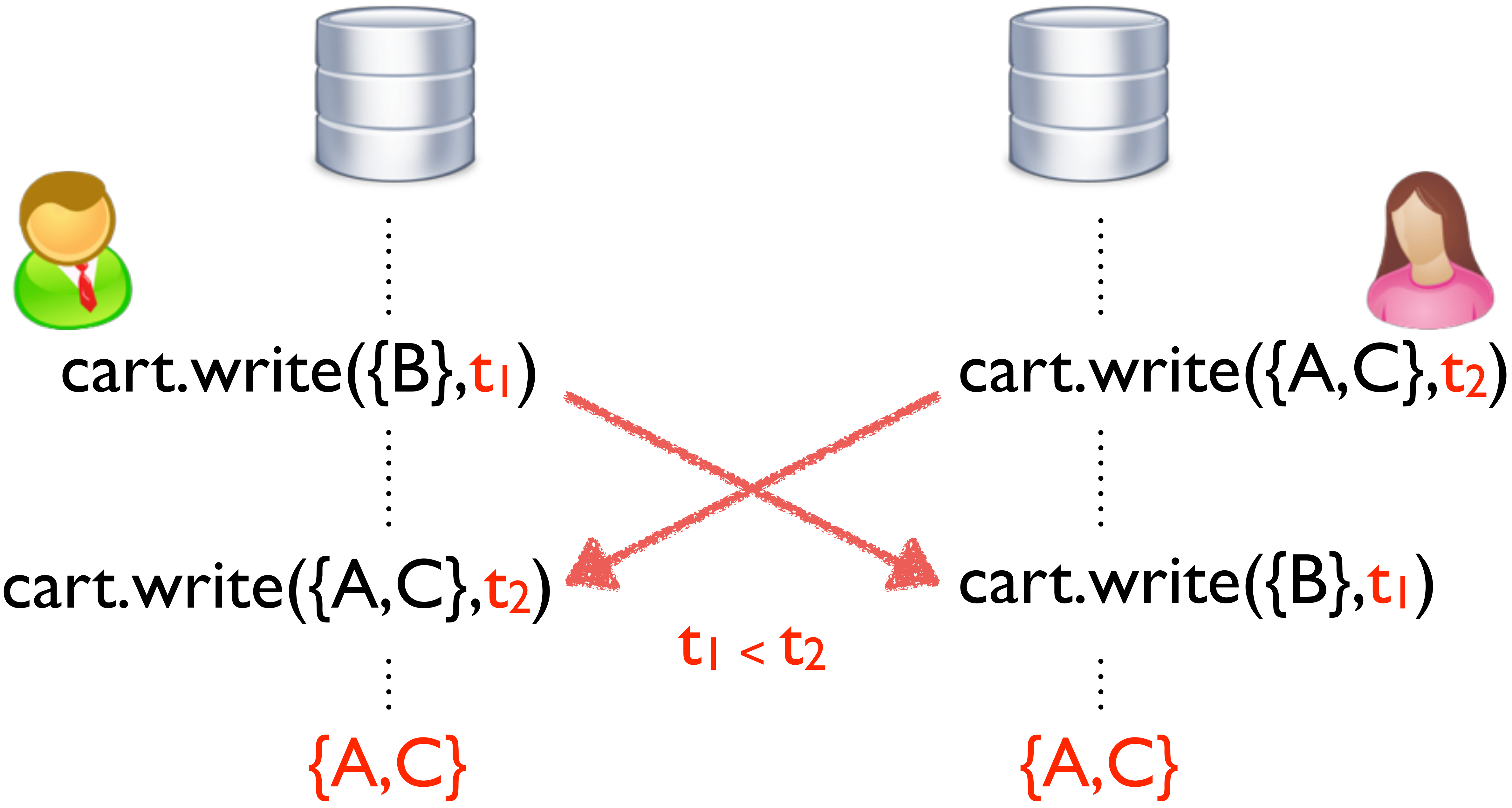
cart.write({A,C},t_2)

cart.write({B},t_1)

$t_1 < t_2$

Use a distributed global clock, such as Lamport clock.

Last-writer-win policy



cart.write({B},t_1)

cart.write({A,C},t_2)

cart.write({A,C},t_2)

cart.write({B},t_1)

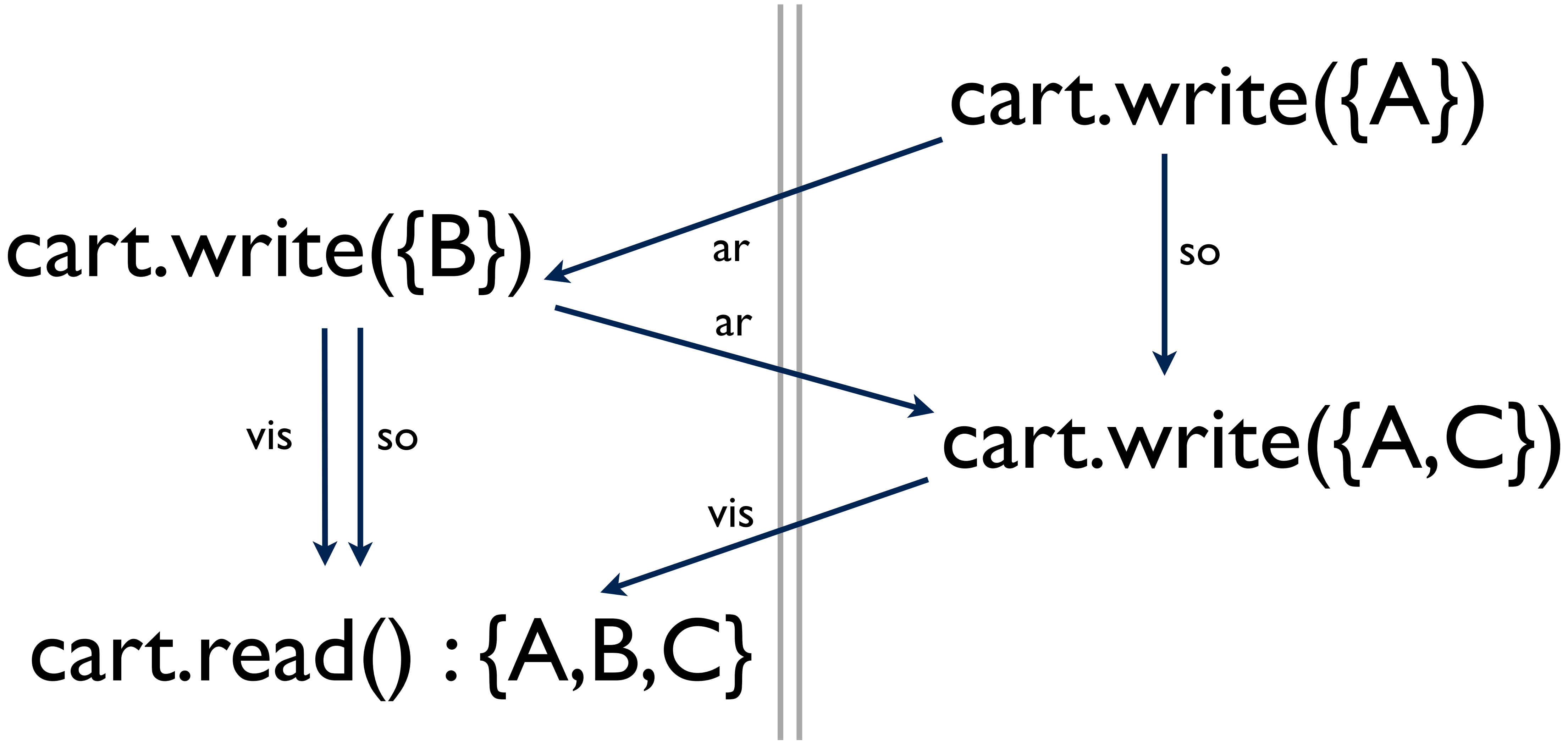
$t_1 < t_2$

{A,C}

{A,C}

Use a distributed global clock, such as Lamport clock.

Execution: (E, so, vis, ar)



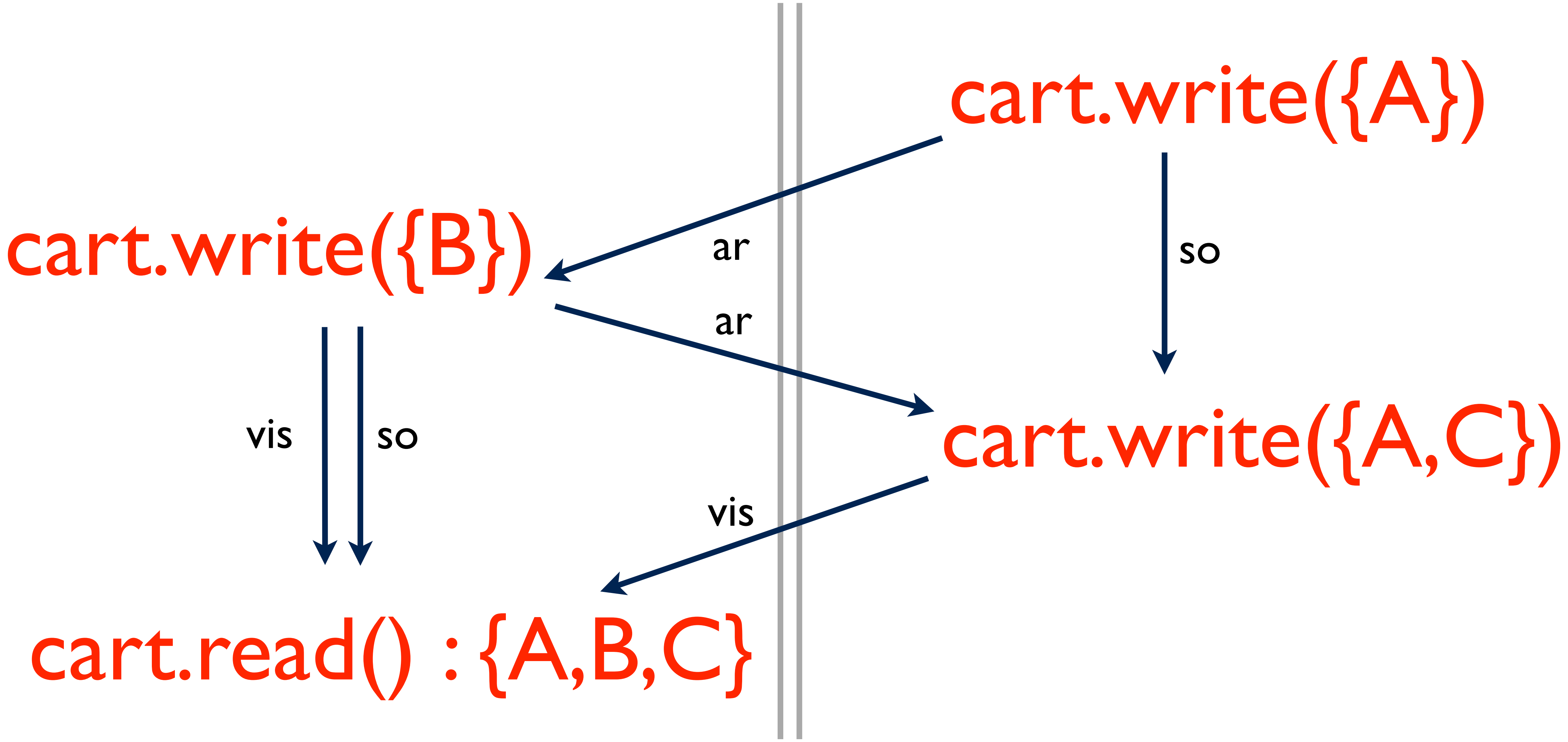
cart.write({A})

cart.write({B})

cart.write({A,C})

cart.read() : {A,B,C}

Execution: (E, so, vis, ar)



cart.write({A})

cart.write({B})

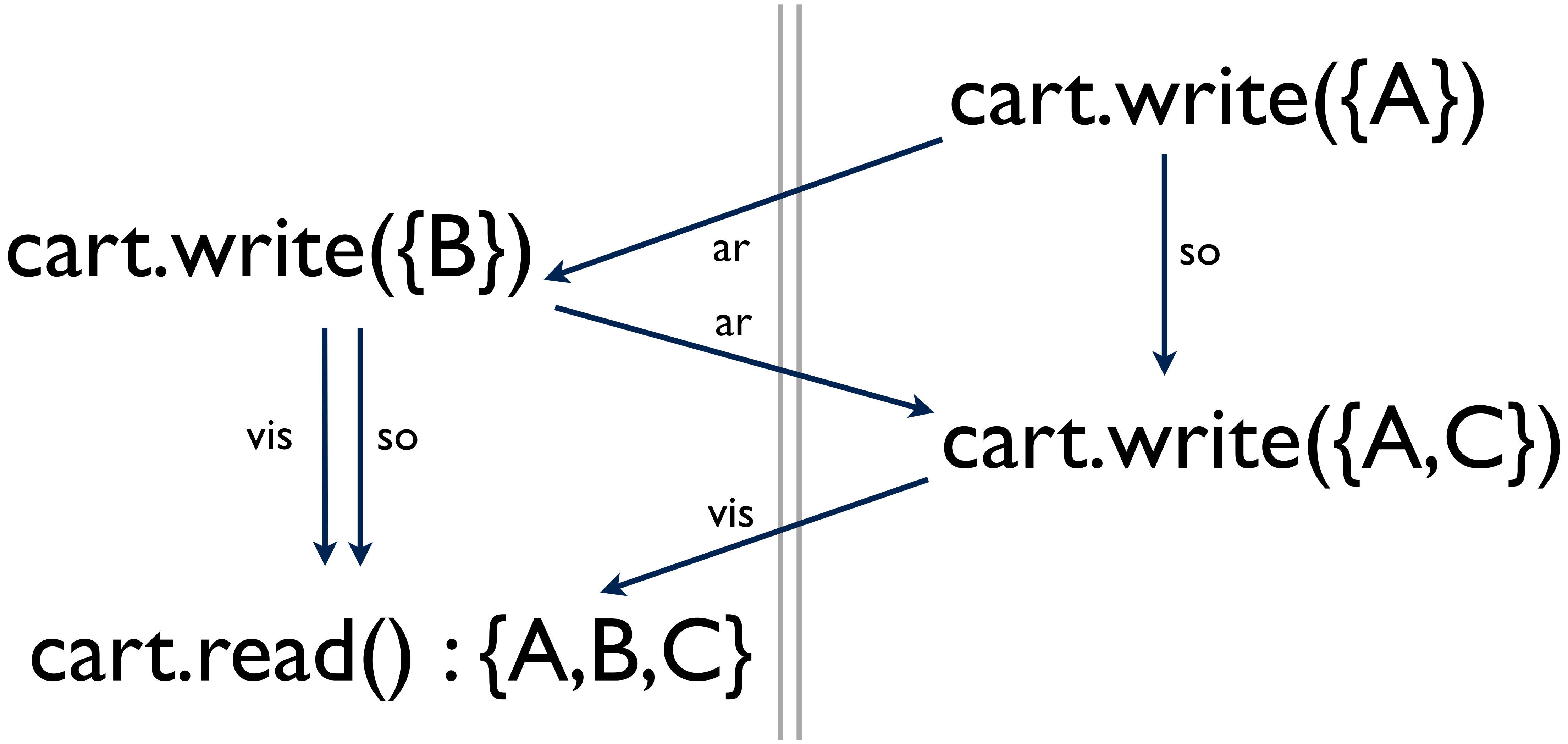
cart.write({A,C})

cart.read() : {A,B,C}

Events:

Describe what happened between stores and clients.

Execution: (E, so, vis, ar)



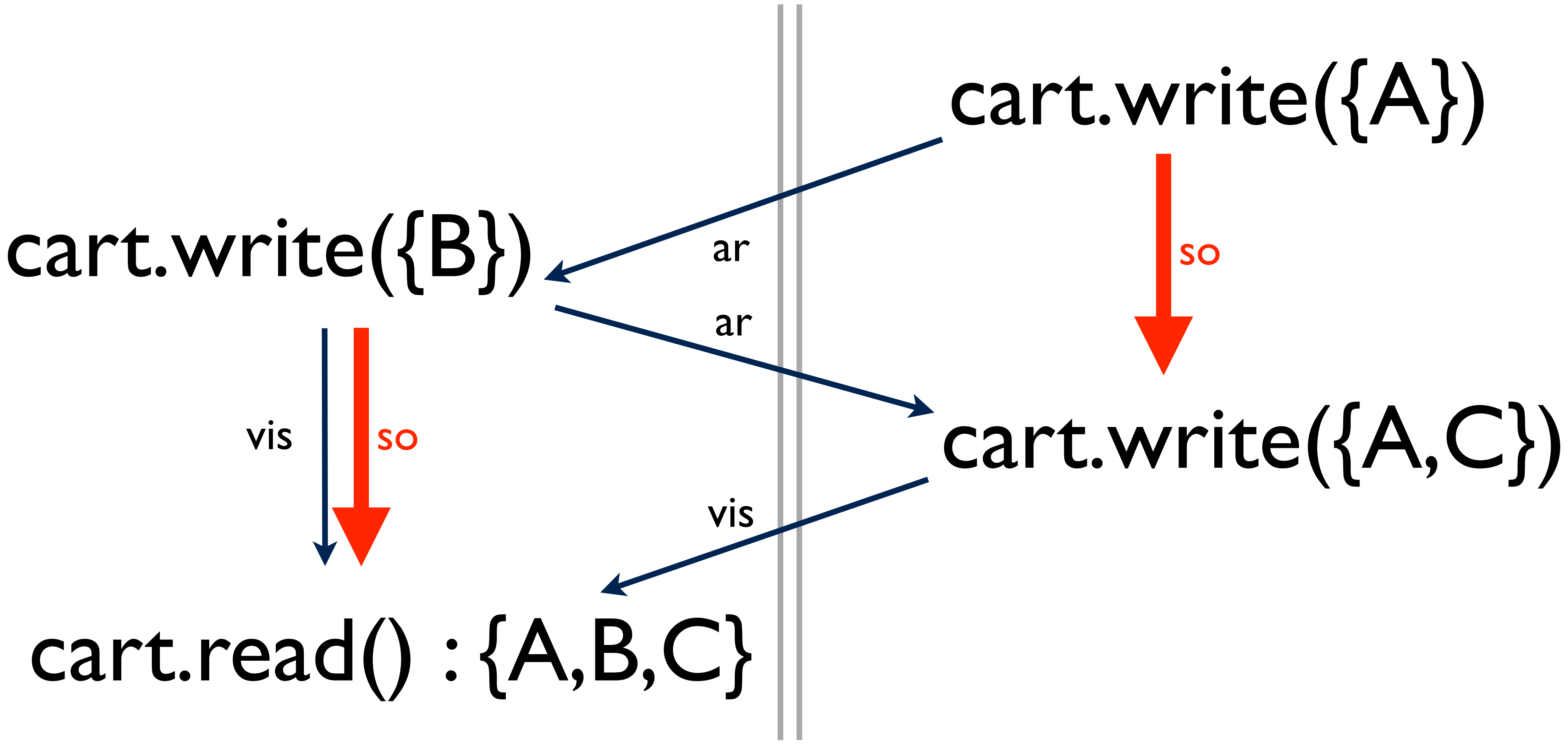
cart.write({A})

cart.write({B})

cart.write({A,C})

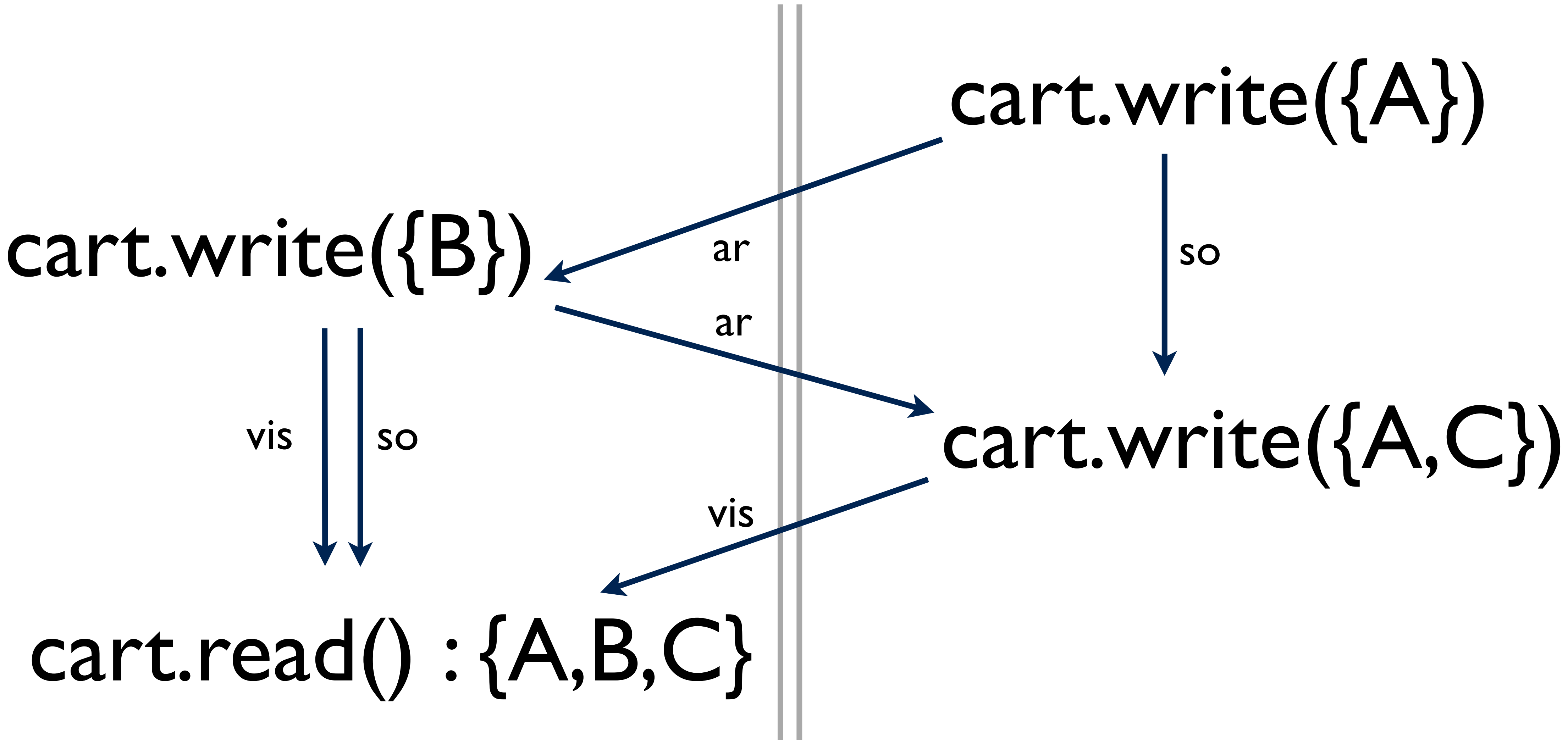
cart.read() : {A,B,C}

Execution: (E, so, vis, ar)



Session order:
Models processes.

Execution: (E, so, vis, ar)



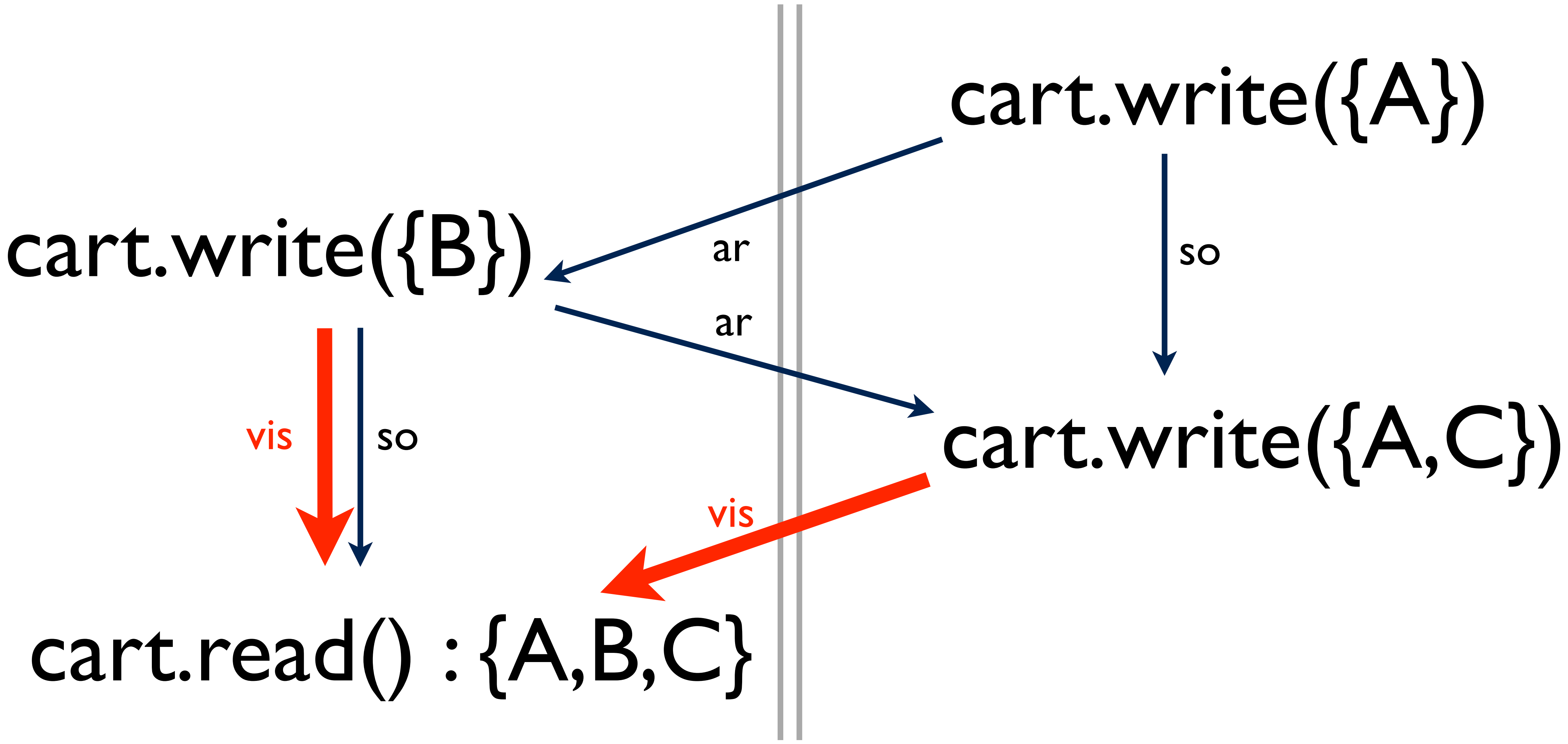
cart.write({A})

cart.write({B})

cart.write({A,C})

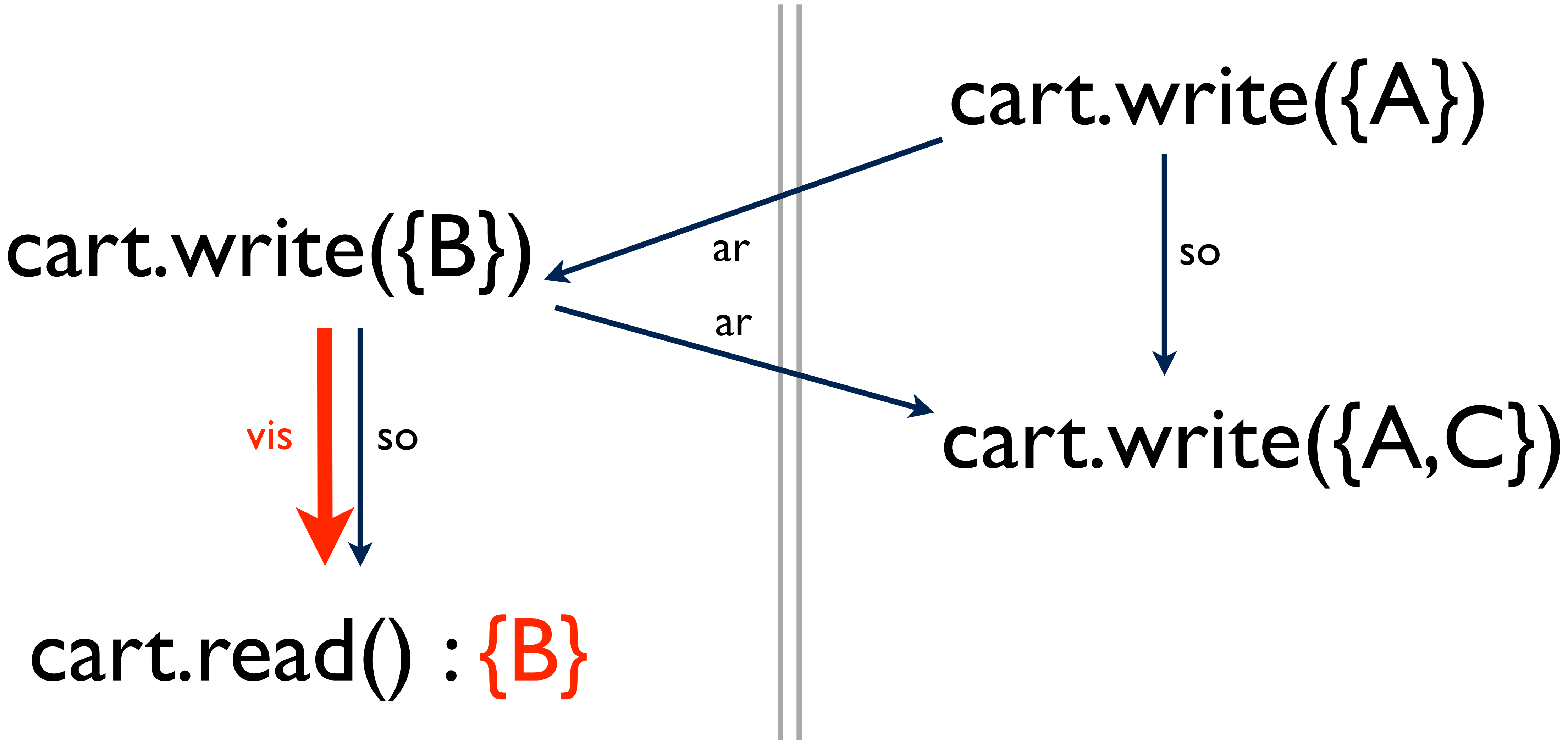
cart.read() : {A,B,C}

Execution: (E, so, vis, ar)



Visibility relation:
Models updates visible to each operation.

Execution: (E, so, vis, ar)



cart.write({B})

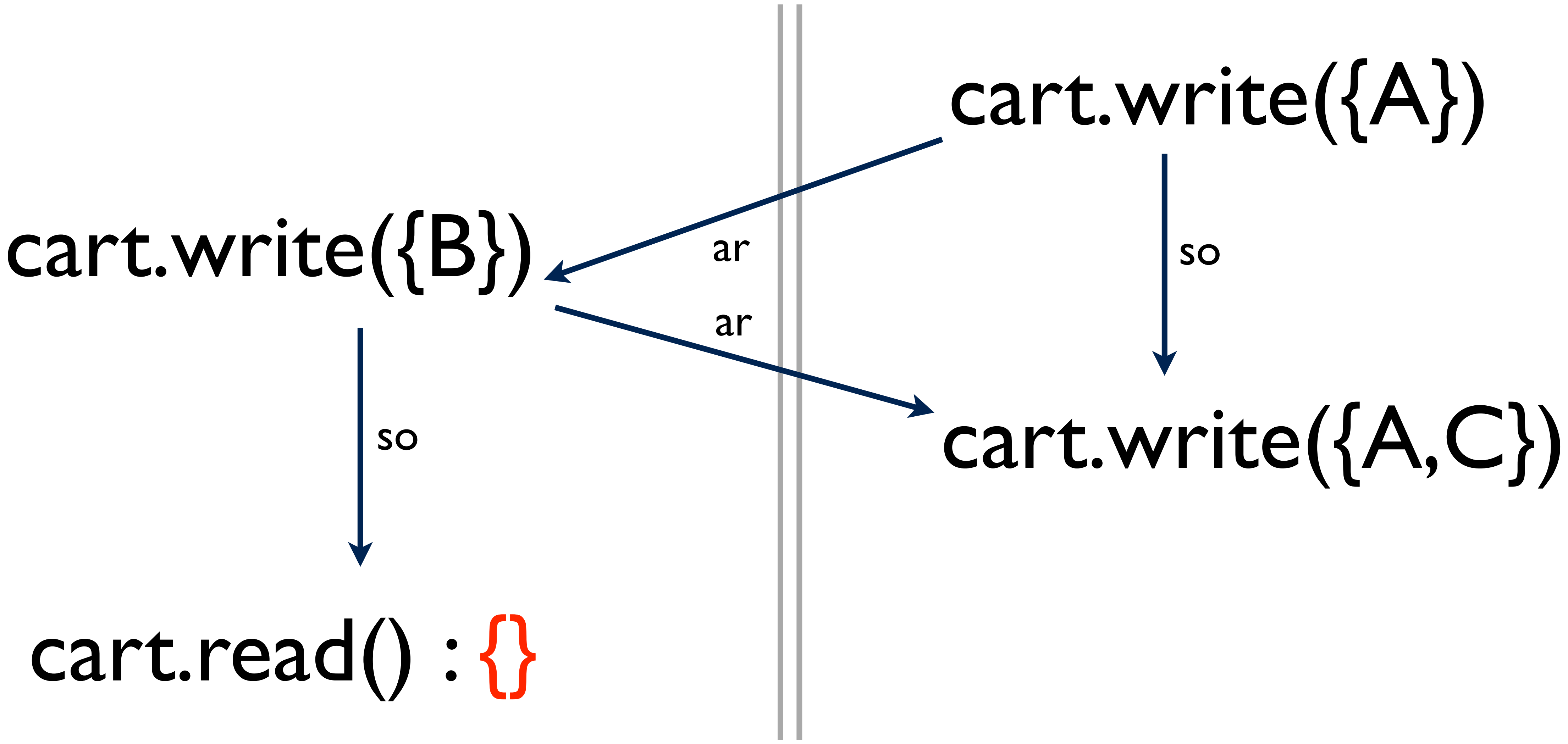
cart.read() : {B}

cart.write({A})

cart.write({A,C})

Visibility relation:
Models updates visible to each operation.

Execution: (E, so, vis, ar)



cart.write({A})

cart.write({B})

ar

ar

so

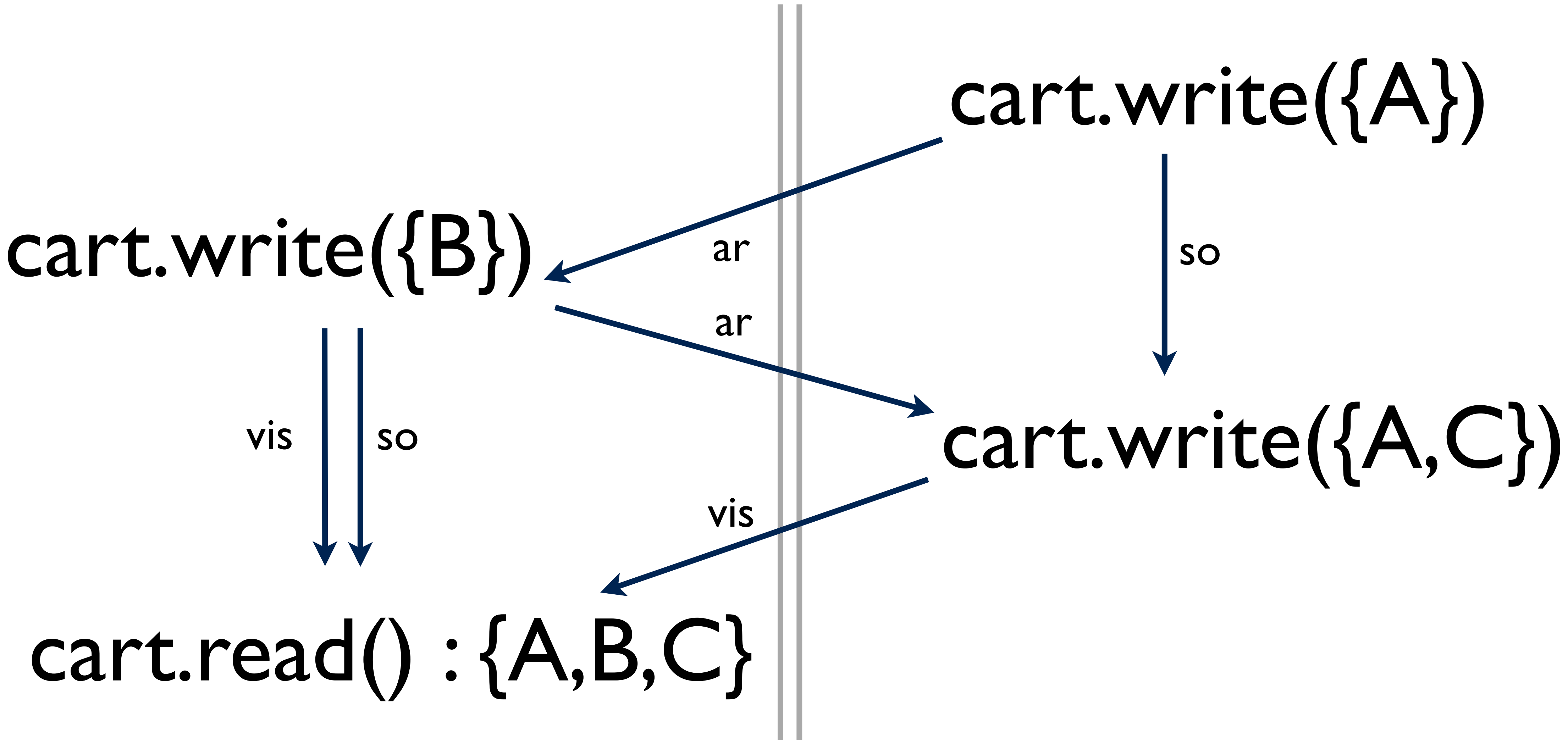
cart.write({A,C})

so

cart.read() : {}

Visibility relation:
Models updates visible to each operation.

Execution: (E, so, vis, ar)



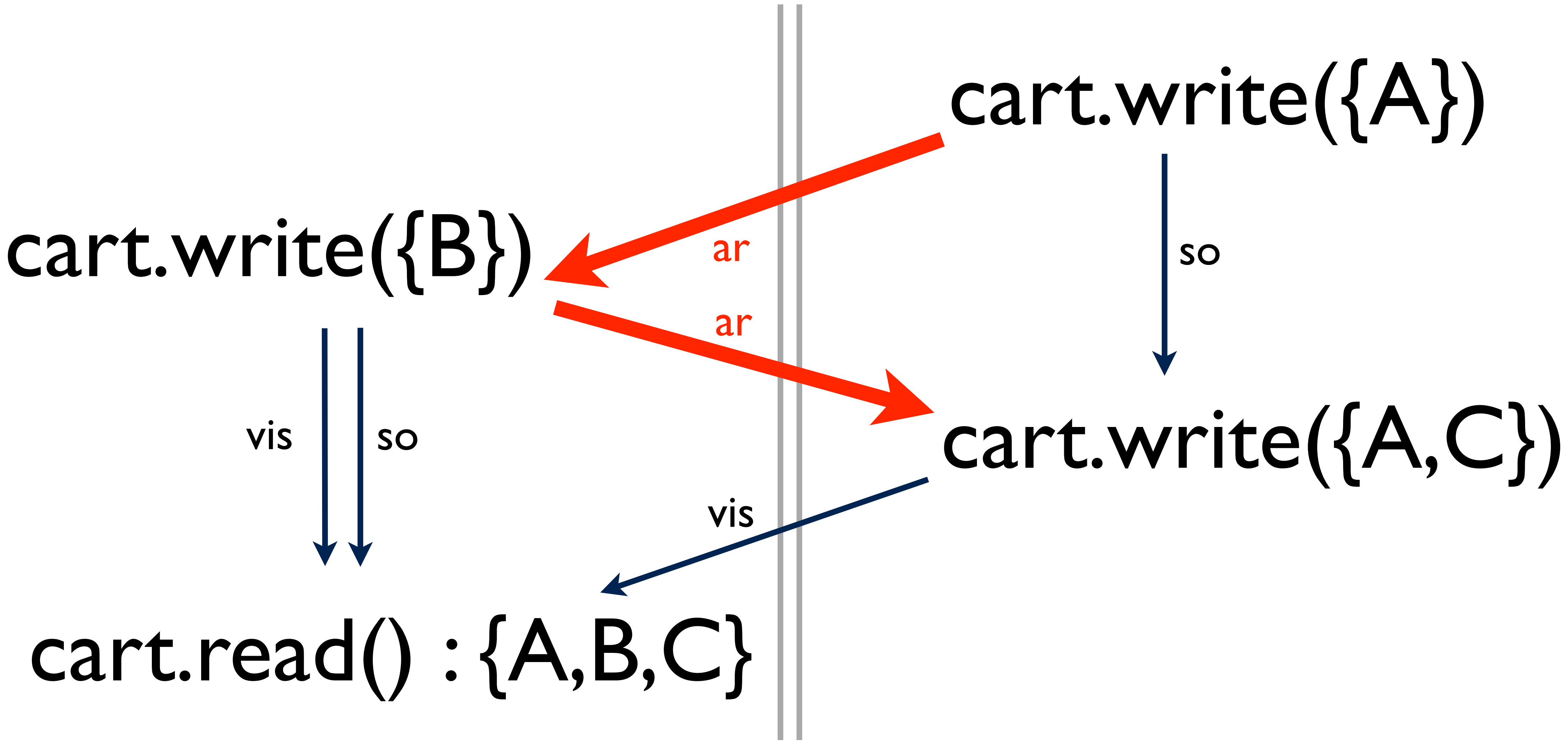
cart.write({A})

cart.write({B})

cart.write({A,C})

cart.read() : {A,B,C}

Execution: (E, so, vis, ar)



Arbitration relation:
Represents global timestamps abstractly.

Our formalism

- We specify replicated stores in terms of possible executions with them.
- Our specification is based on axioms that executions should satisfy.

Basic eventual consistency

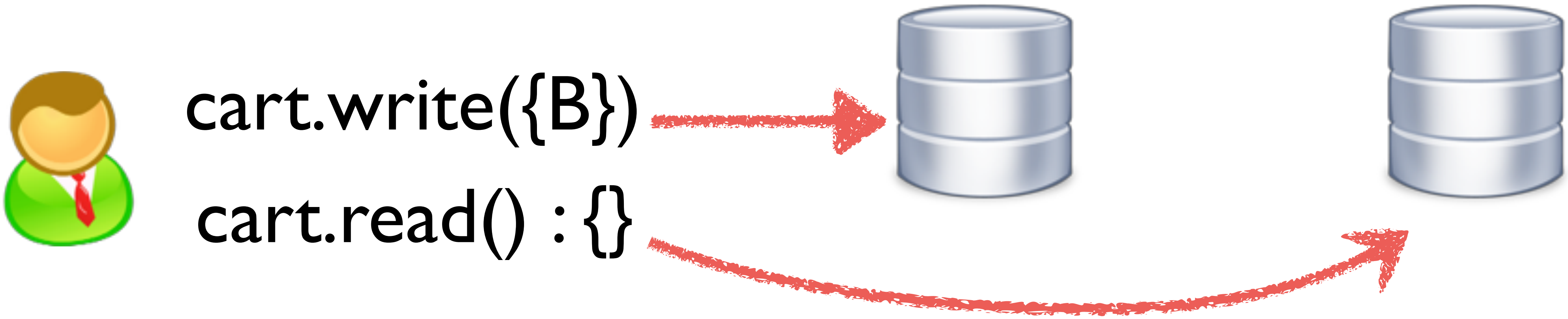
EVENTUAL:

$$\forall e \in E.\, \neg(\exists \text{ infinitely many } f \in E.\, \mathsf{sameobj}(e, f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f))$$

Basic eventual consistency

EVENTUAL:

$$\forall e \in E.\, \neg(\exists \text{ infinitely many } f \in E.\, \mathsf{sameobj}(e, f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f))$$



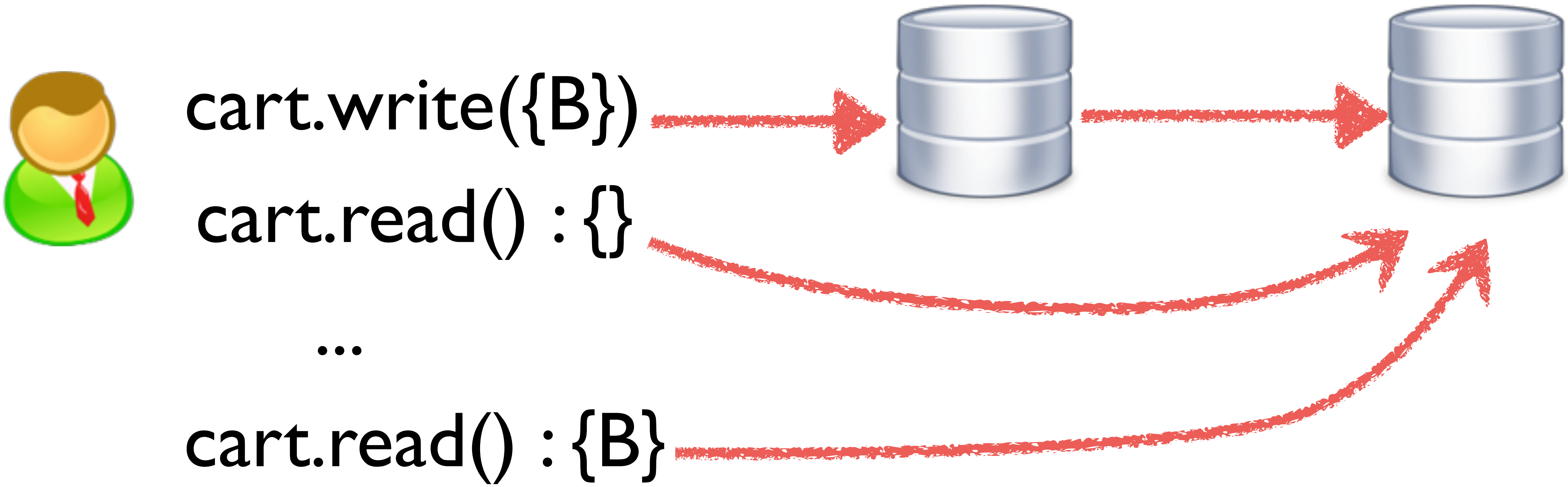
cart.write({B})

cart.read() : {}

Basic eventual consistency

EVENTUAL:

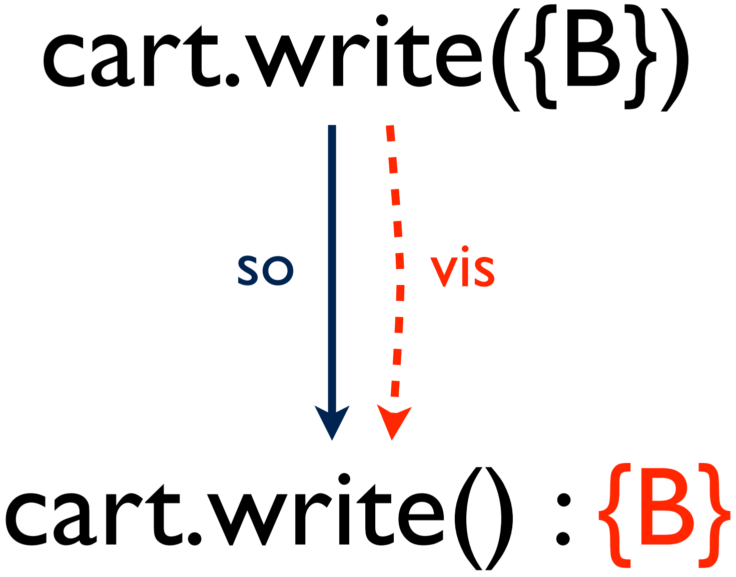
$$\forall e \in E.\, \neg(\exists \text{ infinitely many } f \in E.\, \mathsf{sameobj}(e, f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f))$$



Forbidding anomalies

We require that certain events be visible by including additional edges into vis.

Read Your Writes: so ∩ sameobj ⊆ vis



cart.write({B})

cart.write() : {B}

Data type specifications

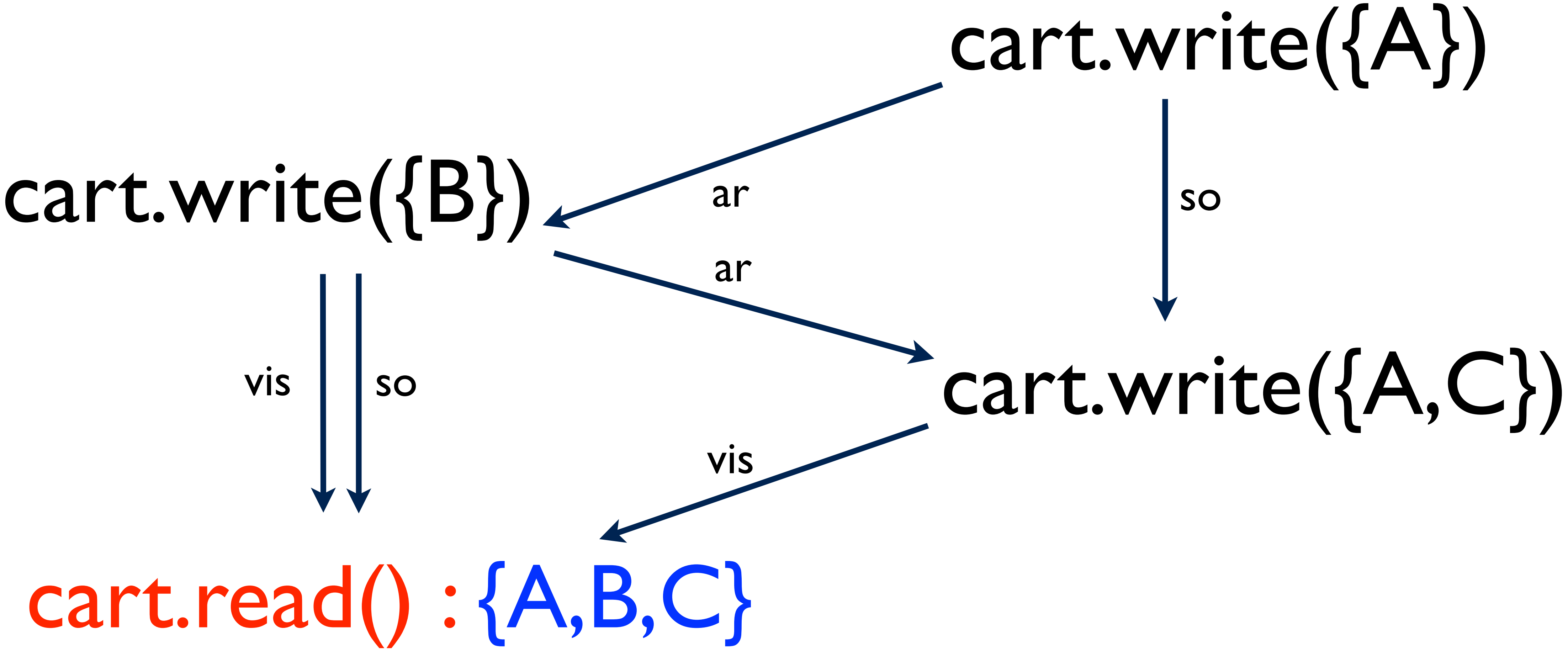
- Conflict resolution policies are implemented in data types [Shapiro+2011].
- Data type specifications in our formalism express these policies.
- They determine the return value of each operation in executions.

Data type specifications

$$\boxed{\begin{array}{c} F_\tau : \text{OperationContexts} \to \text{Values} \\ \textsc{Data} : \forall e \in E.\, \mathsf{retval}(e) = F_{\mathsf{type}(e)}(\mathsf{ctxt}(e)) \end{array}}$$

Data type specifications

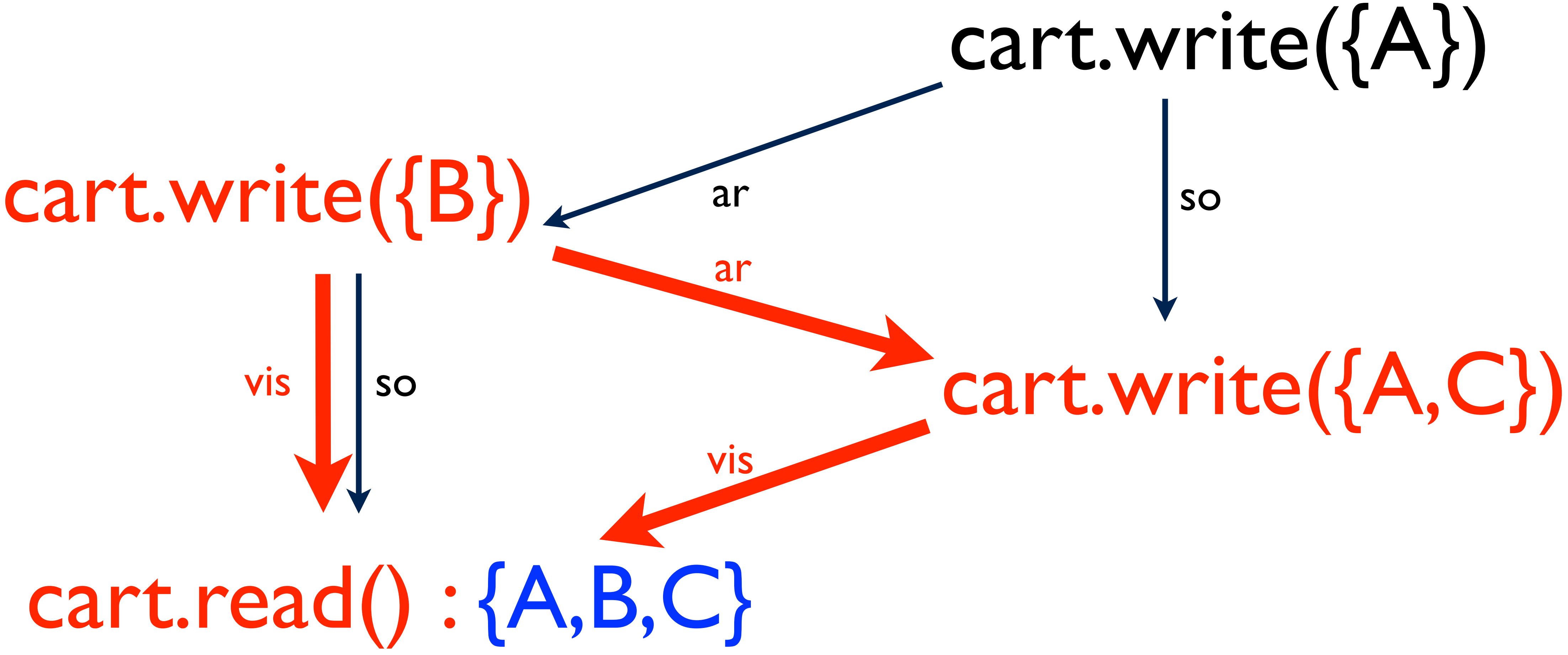
$$\boxed{\begin{array}{r}F_\tau : \mathrm{OperationContexts} \to \mathrm{Values} \\ \textsc{Data} : \forall e \in E.\, \underset{\{A,B,C\}}{\underline{\mathsf{retval}(e)}} = F_{\mathsf{type}(e)}(\mathsf{ctxt}(e))\end{array}}$$



Data type specifications

$$\boxed{\begin{array}{r}F_\tau : \mathrm{OperationContexts} \to \mathrm{Values} \\ \textsc{Data} : \forall e \in E.\, \underline{\mathsf{retval}(e)} = \underline{F_{\mathsf{type}(e)}(\mathsf{ctxt}(e))}\end{array}}$$

{A,B,C}



cart.write({A})

cart.write({B})

ar

so

ar

cart.write({A,C})

vis

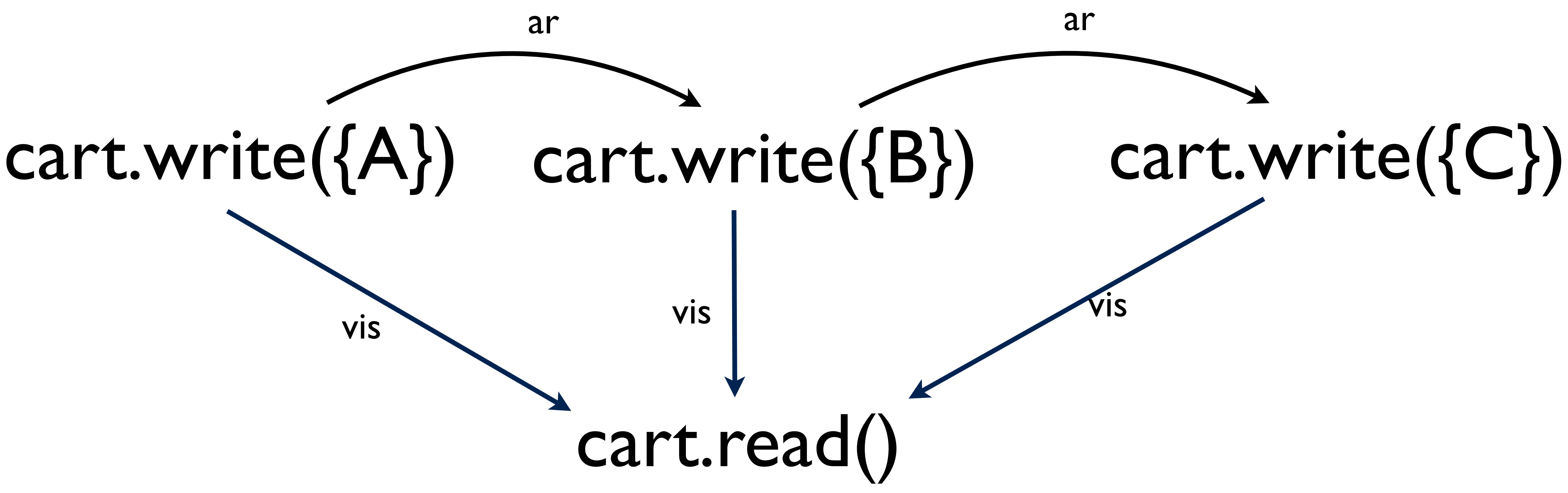
so

vis

cart.read() : {A,B,C}

Multi-valued register

- F_{mv} collects all values written by concurrent operations.



cart.write({A})

cart.write({B})

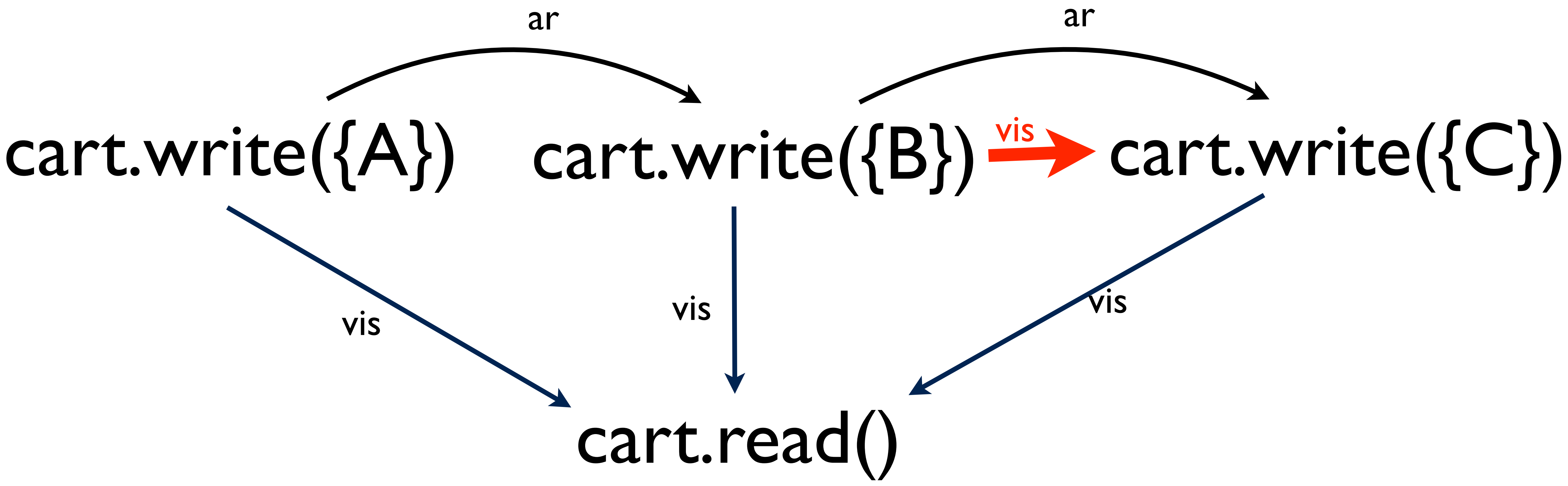
cart.write({C})

cart.read()

$F_{mv}(ctxt(cart.read)) = \{A,B,C\}$

Multi-valued register

- F_{mv} collects all values written by concurrent operations.



cart.write({A})

cart.write({B})

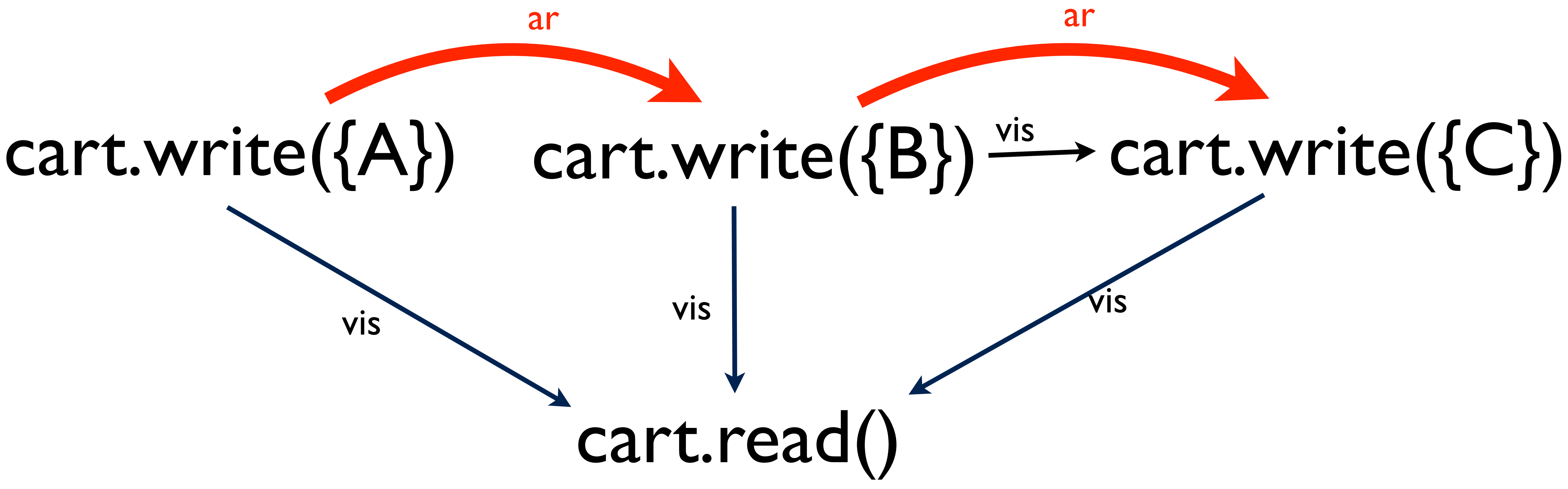
cart.write({C})

cart.read()

$F_{mv}(ctxt(cart.read)) = \{A,C\}$

Last-writer-win Register

- F_{last} selects the value written by the last update according to the ar relation.



ar

ar

cart.write({A})

cart.write({B})

vis

cart.write({C})

vis

vis

vis

cart.read()

$$F_{last}(ctxt(cart.read)) = \{C\}$$

Application 1

Use F to specify a new data type.

Application 2

Verify that a data-type implementation satisfies a given specification F.

Application 3

Obtain a lower bound on the size of the meta data for any data type satisfying F.

Research opportunities for EC

- Formalisation and foundation.
- New EC store that provides different levels of consistency on demand.
- Efficient data structures with intuitive conflict resolution policies.
- Design a new programming language.
- Static analysis.