Data, Data Storage, Data Collection

Lecture 10: Denormalization and Aggregates

Romain Pascual

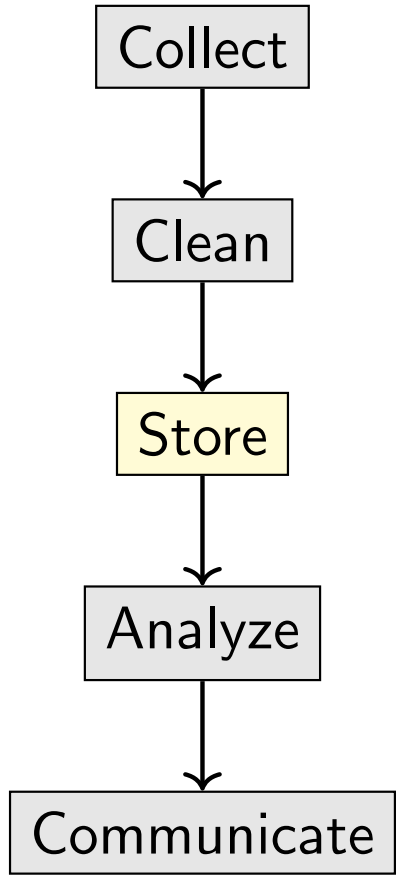
MICS, CentraleSupélec, Université Paris-Saclay

Introduction

Recap: Normalization

1. **Redundancy causes anomalies:** a fact stored with another cannot be inserted, deleted or updated on its own.
2. **Dependencies are rules of the domain:** the rows can refute a dependency, never prove it.
3. **Keys decide the normal form:** find the candidate keys, then check each non-trivial dependency against them.
4. **Normalize by lossless splits:** split along a dependency; the join gives the table back, with no spurious rows.
5. **Not every redundancy is an FD:** independent facts stored as combinations (4NF), several kinds of facts in one table (5NF).
6. **BCNF may not preserve dependencies:** some rules can no longer be checked inside one table.

Within The Lifecycle



Session Objectives

By the end of this session, you should be able to:

- Justify a **denormalization**, and name its cost
- Explain why an **aggregate** leaves the relational model
- Place aggregates on nodes by hashing their key, and say what stays atomic
- Choose **consistency** or **availability** for each operation during a partition

Denormalization

Data Modeling for a Commercial Platform

An online shop: customers place orders, and each order lists products with a quantity.

Customers

CustomerID	Name	City
C01	Alex	Paris
C02	Sam	Lyon
C03	Mia	Lille

Products

ProductID	Product	UnitPrice
P01	Coffee Machine	50
P02	Mug	5
P03	Tea Box	6

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04

Order_Products

OrderID	ProductID	Quantity
O01	P01	1
O01	P02	2
O02	P03	3
O03	P01	1
O04	P03	1

Each fact is stored once and the structure prevents contradictions.

Why Would We Ever *Not* Normalize?

Normalization protects against inconsistencies and modification anomalies.

Why Would We Ever *Not* Normalize?

Normalization protects against inconsistencies and modification anomalies.

But... in practice, some databases are **denormalized**.

Definition (Denormalization)

Intentional storage of redundant data in a normalized database: copies of facts stored elsewhere, or values computed from other data.

It can violate **any normal form**:

`1NF` `2NF` `3NF` `BCNF` `4NF` `5NF`

Why Would We Ever *Not* Normalize?

Normalization protects against inconsistencies and modification anomalies.

But... in practice, some databases are **denormalized**.

Definition (Denormalization)

Intentional storage of redundant data in a normalized database: copies of facts stored elsewhere, or values computed from other data.

It can violate **any normal form**:

`1NF` `2NF` `3NF` `BCNF` `4NF` `5NF`

Why would a designer deliberately accept redundancy and inconsistency risk?

Denormalization due to External Integration

When integrating secondary data from other platforms, the source may provide a single flat table. We have no control over it.

Orders

Customer	City	Order Date	Product	Unit Price	Quantity
Alex	Paris	2025-11-01	Coffee Machine	50	1
Alex	Paris	2025-11-01	Mug	5	2
Sam	Lyon	2025-11-02	Tea Box	6	3
Mia	Lille	2025-11-03	Coffee Machine	50	1
Alex	Paris	2025-11-04	Tea Box	6	1

Denormalization due to External Integration

When integrating secondary data from other platforms, the source may provide a single flat table. We have no control over it.

Orders

Customer	City	Order Date	Product	Unit Price	Quantity
Alex	Paris	2025-11-01	Coffee Machine	50	1
Alex	Paris	2025-11-01	Mug	5	2
Sam	Lyon	2025-11-02	Tea Box	6	3
Mia	Lille	2025-11-03	Coffee Machine	40	1
Alex	Paris	2025-11-04	Tea Box	6	1

 If the source contains inconsistencies, loading it into the normalized schema may fail.

Sometimes denormalization is **not a choice**, but a constraint of data integration.

Denormalization from Evolving Rules

The source table could have been normalized. Why is it not?

Denormalization from Evolving Rules

The source table could have been normalized. Why is it not?

A normalized schema assumes certain rules:

- Each product has a single official price.

Denormalization from Evolving Rules

The source table could have been normalized. Why is it not?

A normalized schema assumes certain rules:

- Each product has a single official price.
- What about discounts, promotions or dynamic pricing?

Orders

Customer	City	Order Date	Product	Unit Price	Quantity
Alex	Paris	2025-11-01	Coffee Machine	50	1
Alex	Paris	2025-11-01	Mug	5	2
Sam	Lyon	2025-11-02	Tea Box	6	3
Mia	Lille	2025-11-03	Coffee Machine	40	1
Alex	Paris	2025-11-04	Tea Box	6	1

 Rule changes make the normalized design obsolete.

Denormalization can be **a design choice**, for anticipating changes.

Logical and Physical Levels[1]

How can denormalization make queries faster?

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04

tables, keys, views, queries

Logical level

- What you see and interact with.
- Guarantees correctness of answers.

[1] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. **Database System Concepts**. 7th ed. McGraw-Hill, 2019.

Logical and Physical Levels[1]

How can denormalization make queries faster?

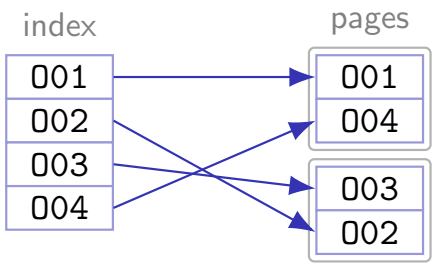
Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04

tables, keys, views, queries

Logical level

- What you see and interact with.
- Guarantees correctness of answers.



storage, execution engine

Physical level

- Controls the physical organization of data.
- Handles the processing of operations.

[1] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. **Database System Concepts**. 7th ed. McGraw-Hill, 2019.

Logical and Physical Levels[1]

How can denormalization make queries faster?

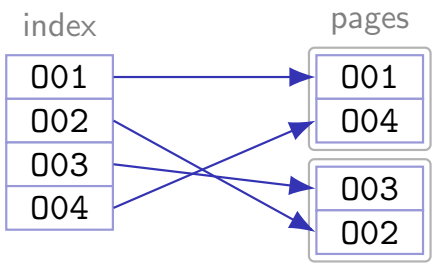
Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04

tables, keys, views, queries

Logical level

- What you see and interact with.
- Guarantees correctness of answers.



storage, execution engine

Physical level

- Controls the physical organization of data.
- Handles the processing of operations.

Performance issues should be addressed at the **physical level**.

[1] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. **Database System Concepts**. 7th ed. McGraw-Hill, 2019.

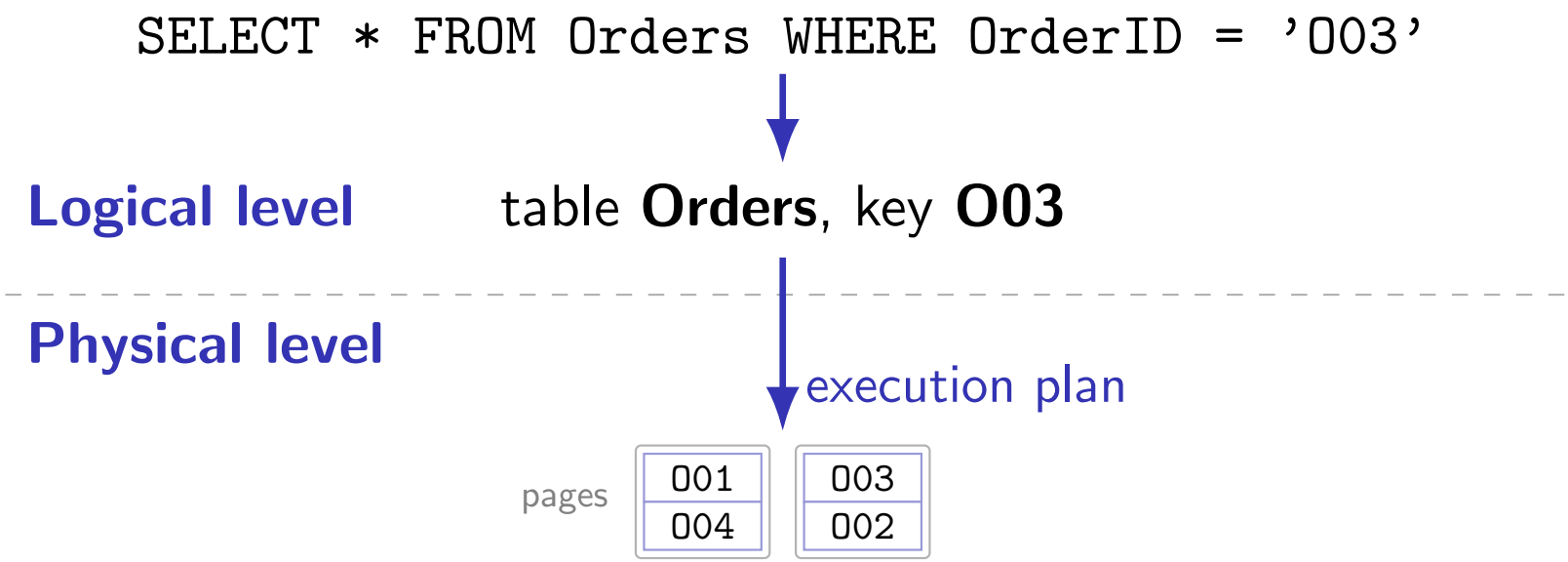
Query Handling

```
SELECT * FROM Orders WHERE OrderID = '003'
```



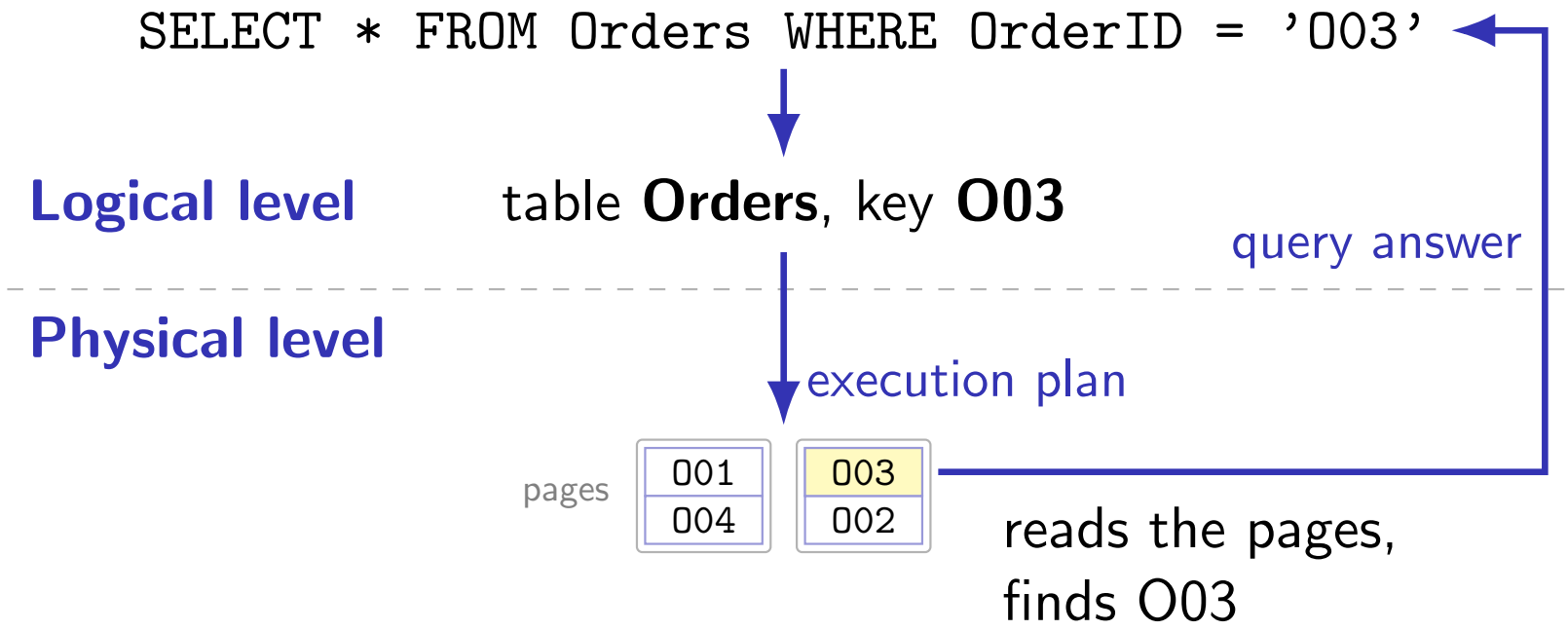
Logical level

Query Handling



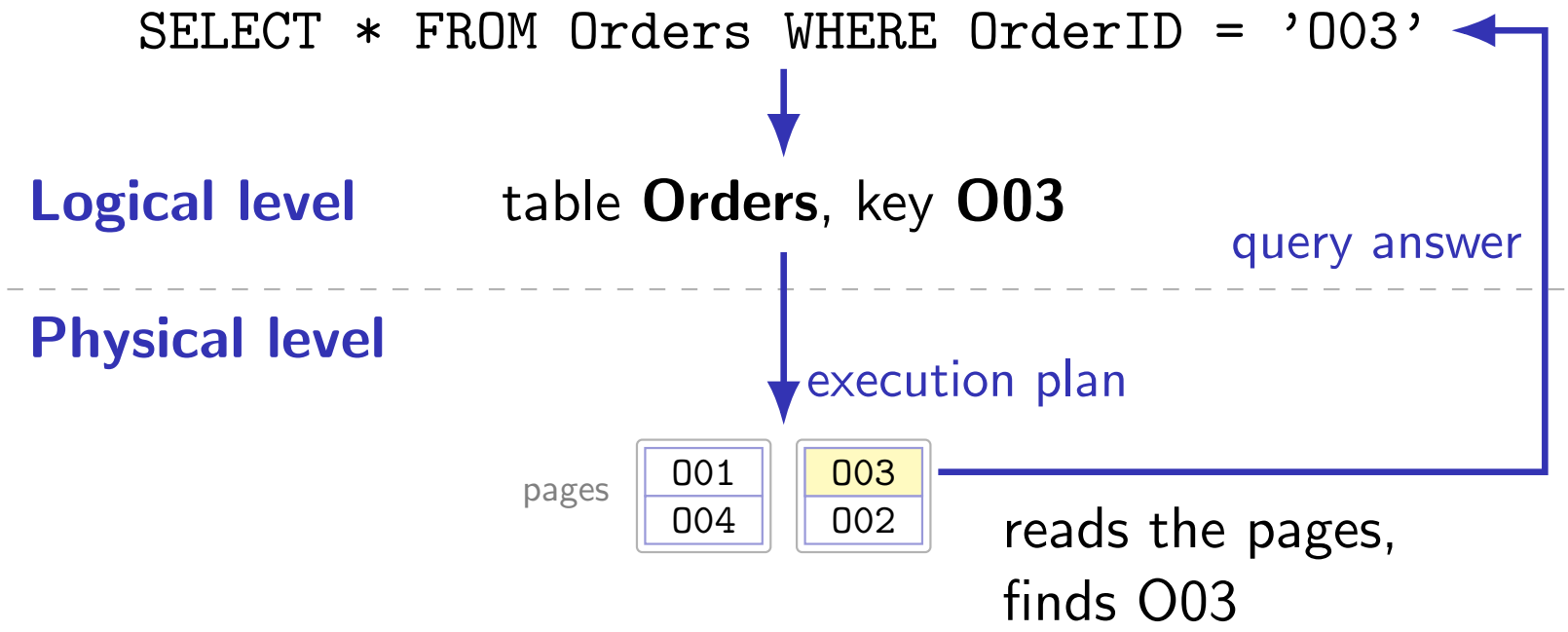
- The **logical level** receives and interprets the query.

Query Handling



- The **logical level** receives and interprets the query.
- The **physical level** optimizes and executes it.

Query Handling



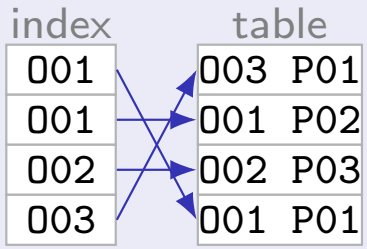
- The **logical level** receives and interprets the query.
- The **physical level** optimizes and executes it.
- Indexes and caching change the storage without changing the logical design: **physical data independence**.

Denormalization is **not** the first solution for performance.

Making a Join Fast

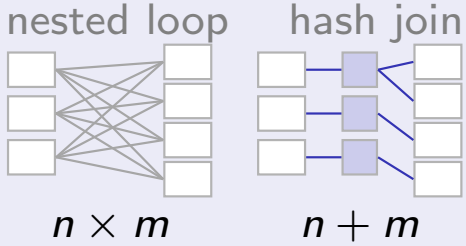
A query that joins two tables is slower than a lookup. The physical level can make it faster without changing the tables.

Index



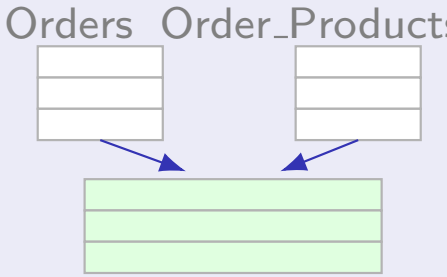
finds rows without scanning the table

Join algorithm



the optimizer picks the cheaper way to pair rows

Materialized view



the join, stored and kept up to date

The index is sorted, not the table.

Denormalization is **not the first fix** for a slow query.

Sorted or Hashed?

Definition (Sorted index)

A sorted index keeps the keys in order, usually in a search tree, so that a key is found in $\mathcal{O}(\log n)$.

Definition (Hash index)

A hash index stores each key in a bucket given by *hash*(*key*), so that a key is found in $\mathcal{O}(1)$ on average.

Sorted index on Date

1 Nov:	001
2 Nov:	002
3 Nov:	003
4 Nov:	004

Hash index on Date

3 Nov:	003	0
2 Nov:	002	1
		2
4 Nov:	004	3
		4
1 Nov:	001	5

Sorted or Hashed?

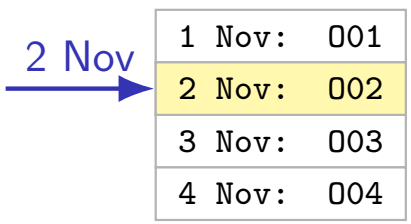
Definition (Sorted index)

A sorted index keeps the keys in order, usually in a search tree, so that a key is found in $\mathcal{O}(\log n)$.

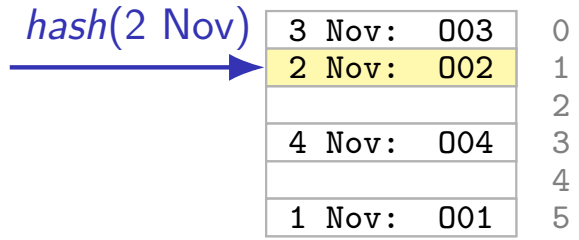
Definition (Hash index)

A hash index stores each key in a bucket given by *hash*(*key*), so that a key is found in $\mathcal{O}(1)$ on average.

Sorted index on Date



Hash index on Date



Sorted or Hashed?

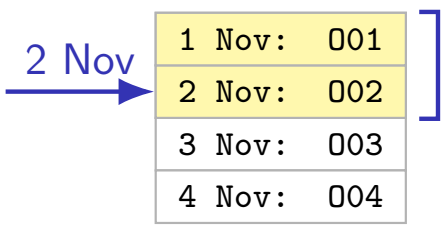
Definition (Sorted index)

A sorted index keeps the keys in order, usually in a search tree, so that a key is found in $\mathcal{O}(\log n)$.

Definition (Hash index)

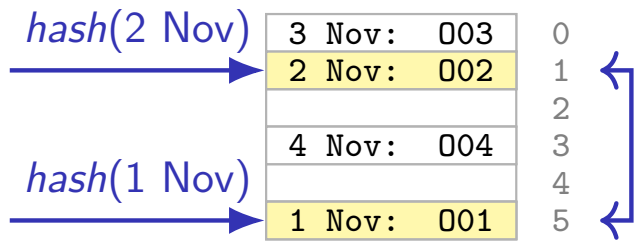
A hash index stores each key in a bucket given by *hash*(*key*), so that a key is found in $\mathcal{O}(1)$ on average.

Sorted index on Date



1 to 2 Nov: side by side

Hash index on Date



1 to 2 Nov: far apart

Sorted or Hashed?

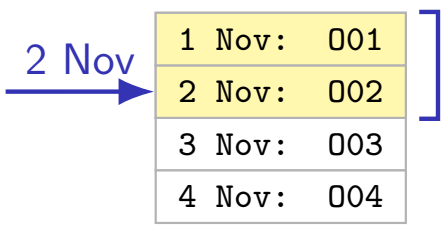
Definition (Sorted index)

A sorted index keeps the keys in order, usually in a search tree, so that a key is found in $\mathcal{O}(\log n)$.

Definition (Hash index)

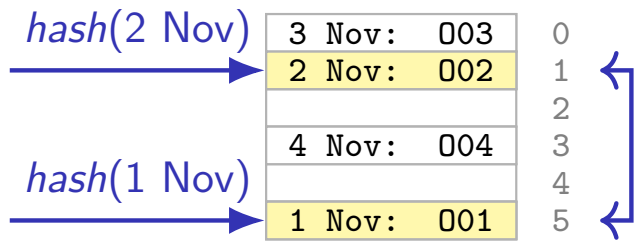
A hash index stores each key in a bucket given by *hash*(*key*), so that a key is found in $\mathcal{O}(1)$ on average.

Sorted index on Date



1 to 2 Nov: side by side

Hash index on Date



1 to 2 Nov: far apart

Hashing finds a key. Sorting also finds its **neighbors**.

When Physical Fixes Are Not Enough

What if the DBMS only provides **limited control** over the physical level?

Denormalization may be the **only option** for:

- Acceptable performance
- Simplified reads

⚠ Inconsistencies become possible, and updates become slower.

Best suited when data is **read often, updated rarely**: OLAP rather than OLTP.

When Physical Fixes Are Not Enough

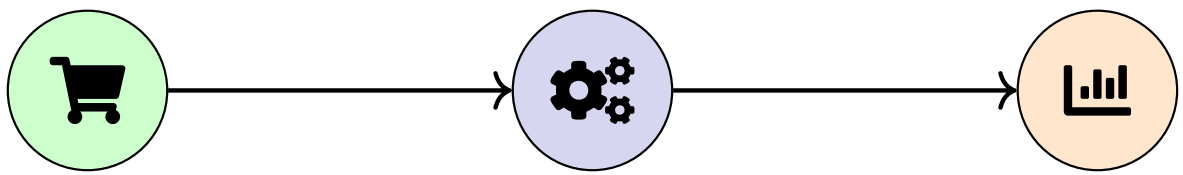
What if the DBMS only provides **limited control** over the physical level?

Denormalization may be the **only option** for:

- Acceptable performance
- Simplified reads

⚠ Inconsistencies become possible, and updates become slower.

Best suited when data is **read often, updated rarely**: OLAP rather than OLTP.



OLTP: the source stays normalized

Extract, Transform, Load

OLAP: a denormalized copy for reports

Aggregates

Denormalize the Table

Even 1NF? What if we store Alex with all his orders as **one record**?

Customers

CustomerID	Name	City
C01	Alex	Paris

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O04	C01	2025-11-04

Order_Products

OrderID	ProductID	Quantity
O01	P01	1
O01	P02	2
O04	P03	1

CustomerID	Name	City	Orders
C01	Alex	Paris	`[(O01, 2025-11-01, [(P01, 1), (P02, 2)]), (O04, 2025-11-04, [(P03, 1)])]`

Denormalize the Table

Even 1NF? What if we store Alex with all his orders as **one record**?

Customers

CustomerID	Name	City
C01	Alex	Paris

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O04	C01	2025-11-04

Order_Products

OrderID	ProductID	Quantity
O01	P01	1
O01	P02	2
O04	P03	1

CustomerID	Name	City	Orders
C01	Alex	Paris	`[(O01, 2025-11-01, [(P01, 1), (P02, 2)]), (O04, 2025-11-04, [(P03, 1)])]`

⚠ A list of orders, each with a list of lines: a **repeating group**, so the record is not in 1NF.

The Aggregate[2]

Definition (Aggregate – Bugiotti et al.)

An aggregate is a group of related data that has **its own key**, is stored as **a single value** that may contain lists and nested records, and is **the unit** in which the application reads and writes data.

Alex and all his orders form one aggregate, with key C01.

```
{"customer_id": "C01", "name": "Alex",
 "city": "Paris",
 "orders": [
 {"order_id": "O01", "date": "2025-11-01",
  "lines": [{"product_id": "P01", "quantity": 1},
            {"product_id": "P02", "quantity": 2}]},
 {"order_id": "O04", "date": "2025-11-04",
  "lines": [{"product_id": "P03", "quantity": 1}]}]}
```

Like a row
One key: C01.

Unlike a row
It nests his orders.

[2] Francesca Bugiotti et al. "Database Design for NoSQL Systems". In: **Conceptual Modeling – ER 2014**. Vol. 8824. Lecture Notes in Computer Science. Springer, 2014, pp. 223–231. DOI: 10.1007/978-3-319-12206-9_18.

Denormalization in Aggregates

One aggregate per customer: which operations get faster, which get slower?

Alex's aggregate

```
{"customer_id": "C01", ...,
 "orders": [
  {"order_id": "O01", ...,
   "lines": [
    {"product_id": "P01",
     "product":  "Coffee Machine",
     "quantity": 1}, ...]},
  ...]}
```

Mia's aggregate

```
{"customer_id": "C03", ...,
 "orders": [
  {"order_id": "O03", ...,
   "lines": [
    {"product_id": "P01",
     "product":  "Coffee Machine",
     "quantity": 1}]}]}
```

copied from **Products**

Denormalization in Aggregates

One aggregate per customer: which operations get faster, which get slower?

Alex's aggregate

```
{"customer_id": "C01", ...,
 "orders": [
  {"order_id": "O01", ...,
   "lines": [
    {"product_id": "P01",
     "product":  "Coffee Machine",
     "quantity": 1}, ...]},
  ...]}
```

Alex's orders

✓ one read

Mia's aggregate

```
{"customer_id": "C03", ...,
 "orders": [
  {"order_id": "O03", ...,
   "lines": [
    {"product_id": "P01",
     "product":  "Coffee Machine",
     "quantity": 1}]}]}
```

copied from **Products**

Denormalization in Aggregates

One aggregate per customer: which operations get faster, which get slower?

Alex's aggregate

```
{"customer_id": "C01", ...,
 "orders": [
  {"order_id": "O01", ...,
   "lines": [
    {"product_id": "P01",
     "product":  "Coffee Machine",
     "quantity": 1}, ...]},
  ...]}
```

Mia's aggregate

```
{"customer_id": "C03", ...,
 "orders": [
  {"order_id": "O03", ...,
   "lines": [
    {"product_id": "P01",
     "product":  "Coffee Machine",
     "quantity": 1}]}]}
```

copied from **Products**

Alex's orders

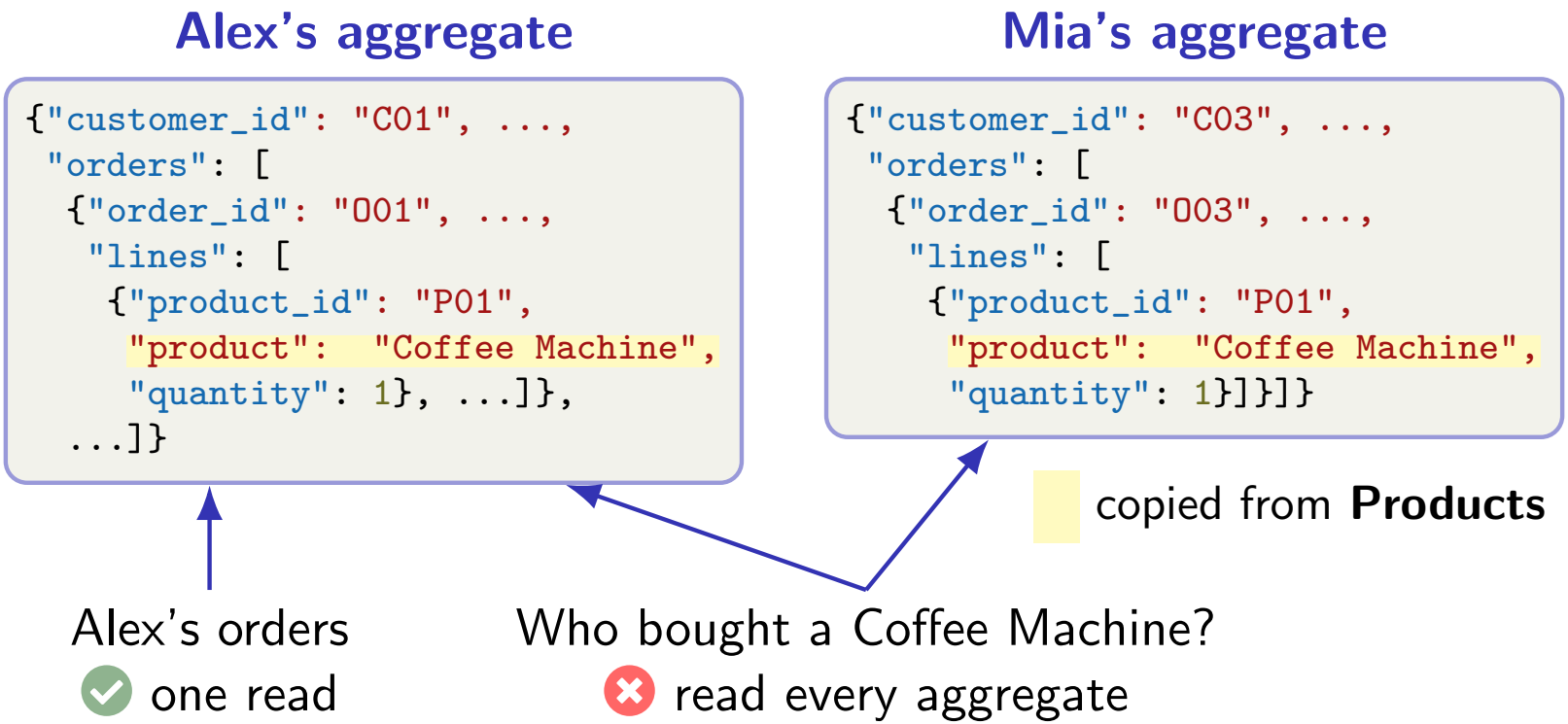
✓ one read

Who bought a Coffee Machine?

✗ read every aggregate

Denormalization in Aggregates

One aggregate per customer: which operations get faster, which get slower?



⚠ Renaming the Coffee Machine rewrites both aggregates: an **update anomaly**.

Distribution

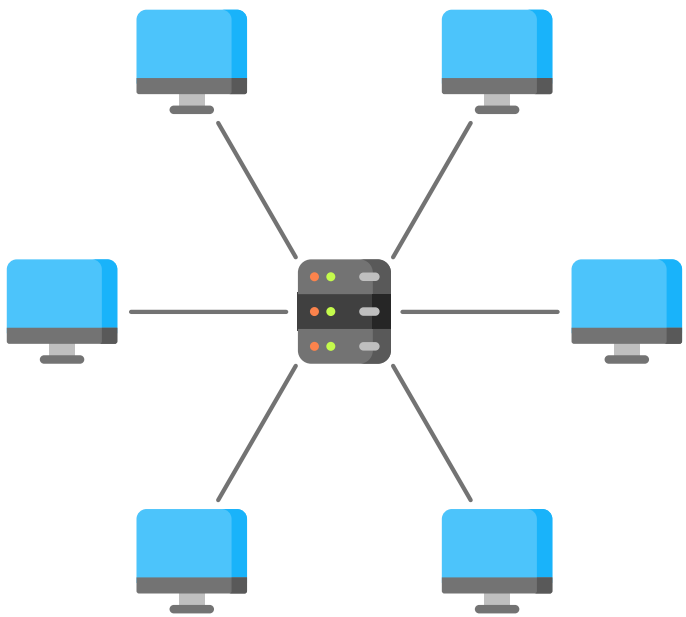
How Do We Scale?

- Volume: Massive data growth
- Velocity: High-speed data generation
- Variety: Diverse data types

How Do We Scale?

- Volume: Massive data growth
- Velocity: High-speed data generation
- Variety: Diverse data types

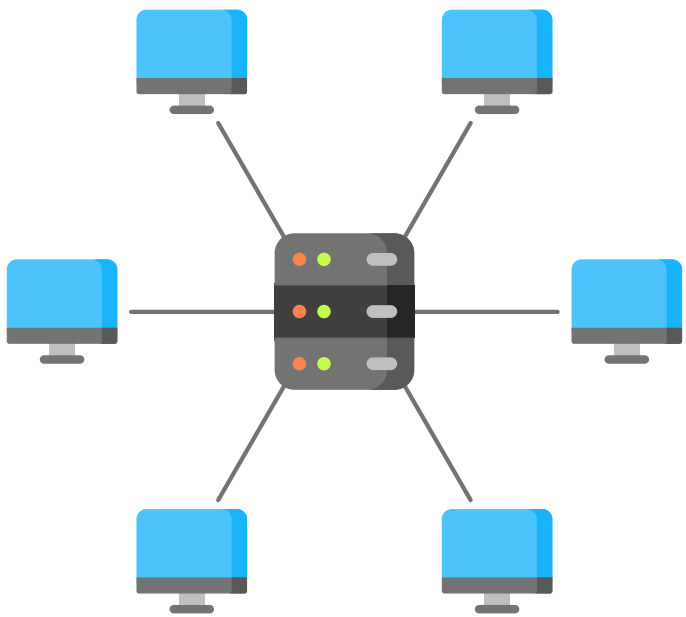
Scaling up



How Do We Scale?

- Volume: Massive data growth
- Velocity: High-speed data generation
- Variety: Diverse data types

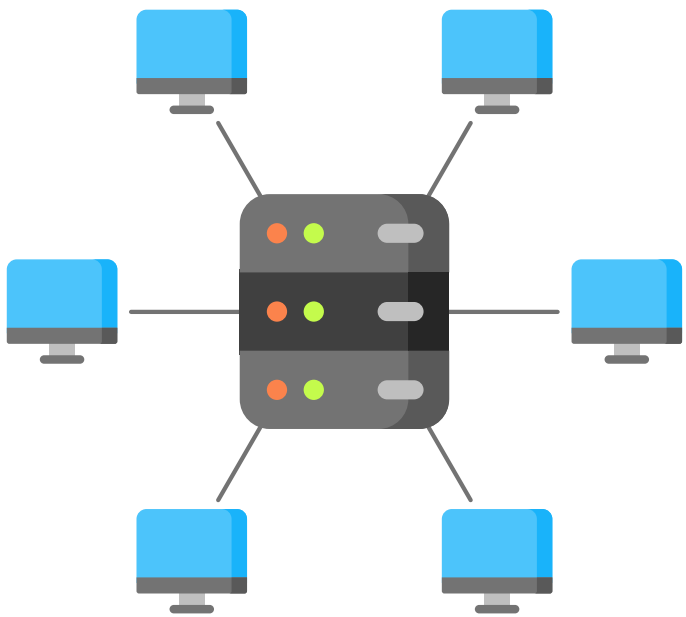
Scaling up



How Do We Scale?

- Volume: Massive data growth
- Velocity: High-speed data generation
- Variety: Diverse data types

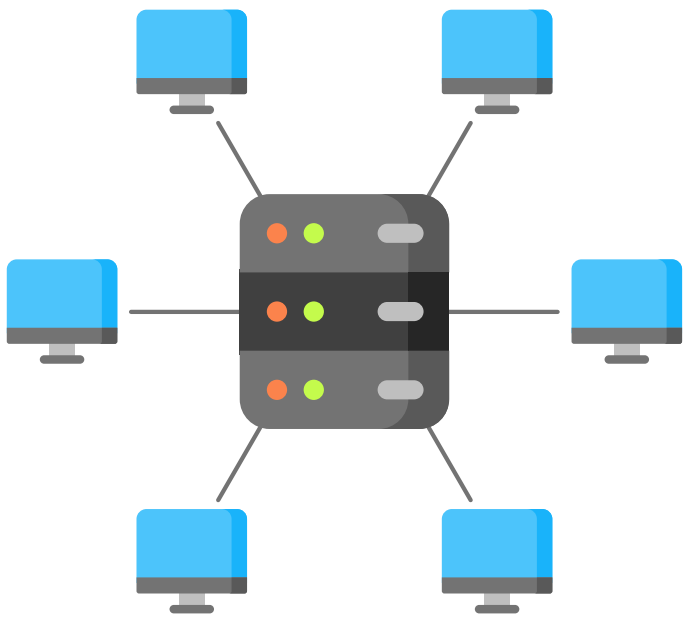
Scaling up



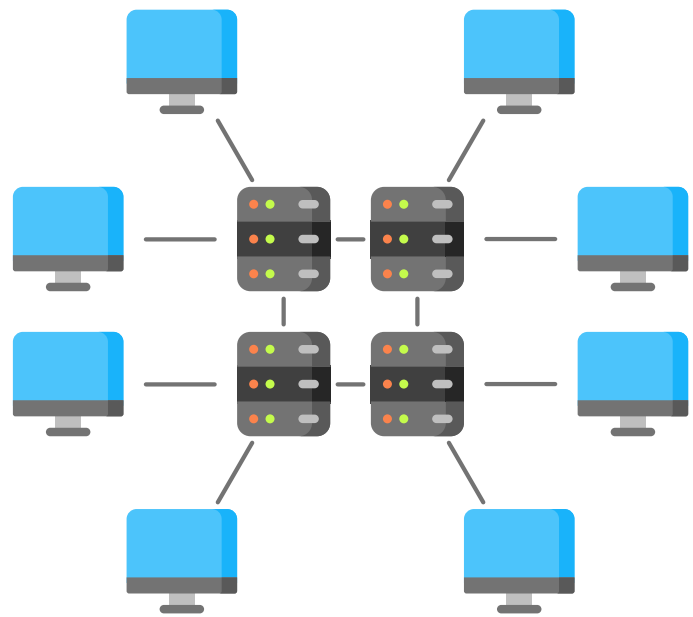
How Do We Scale?

- Volume: Massive data growth
- Velocity: High-speed data generation
- Variety: Diverse data types

Scaling up



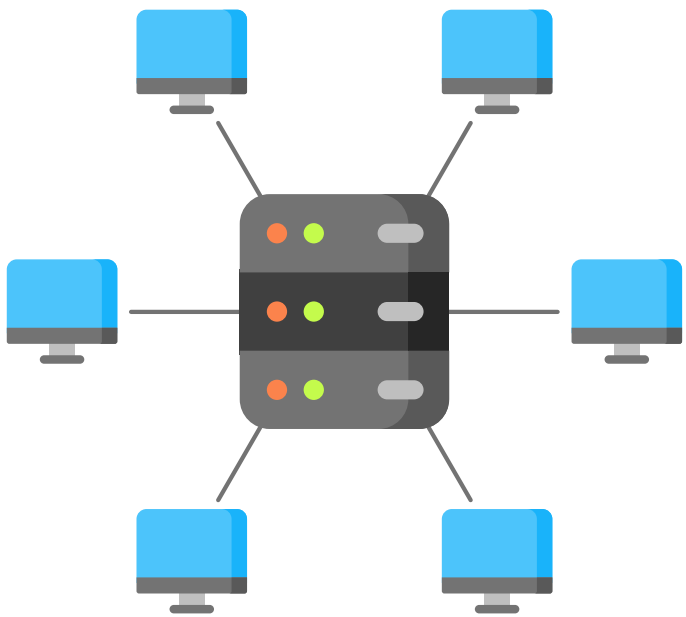
Scaling out



How Do We Scale?

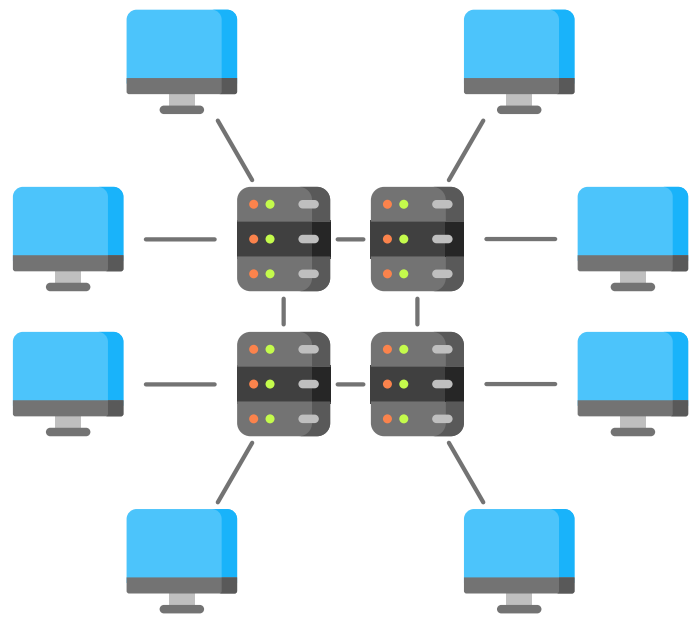
- Volume: Massive data growth
- Velocity: High-speed data generation
- Variety: Diverse data types

Scaling up



Centralized

Scaling out



Distributed

How Do We Split a Database?

Customers

CustomerID	**Name**
C01	Alex
C02	Sam
C03	Mia

Orders

OrderID	**CustomerID**	**Date**
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04

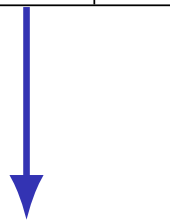
How Do We Split a Database?

Customers

CustomerID	Name
C01	Alex
C02	Sam
C03	Mia

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04





Node 0

Customers

Node 1

Orders

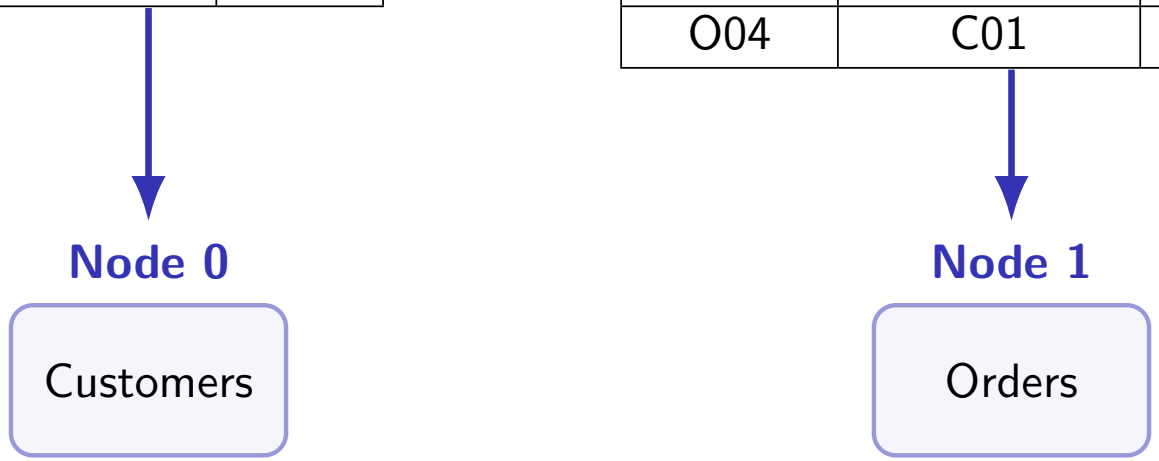
How Do We Split a Database?

Customers

CustomerID	Name
C01	Alex
C02	Sam
C03	Mia

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04



⚠ One table per node does not solve the issue.

- Orders alone still outgrows its node.
- "Get all orders for Alex" needs a `JOIN` across nodes.

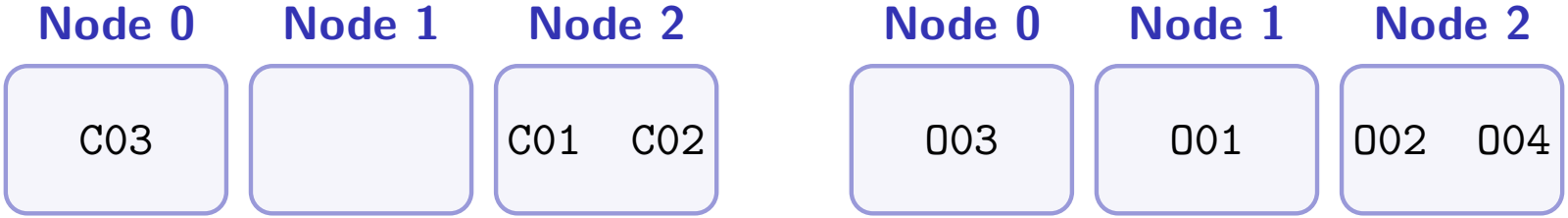
How Do We Split a Database?

Customers

CustomerID	Name
C01	Alex
C02	Sam
C03	Mia

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04



Definition (Sharding)

Sharding, or **horizontal partitioning**, splits a table by rows and stores each part, a **shard**, on its own node.

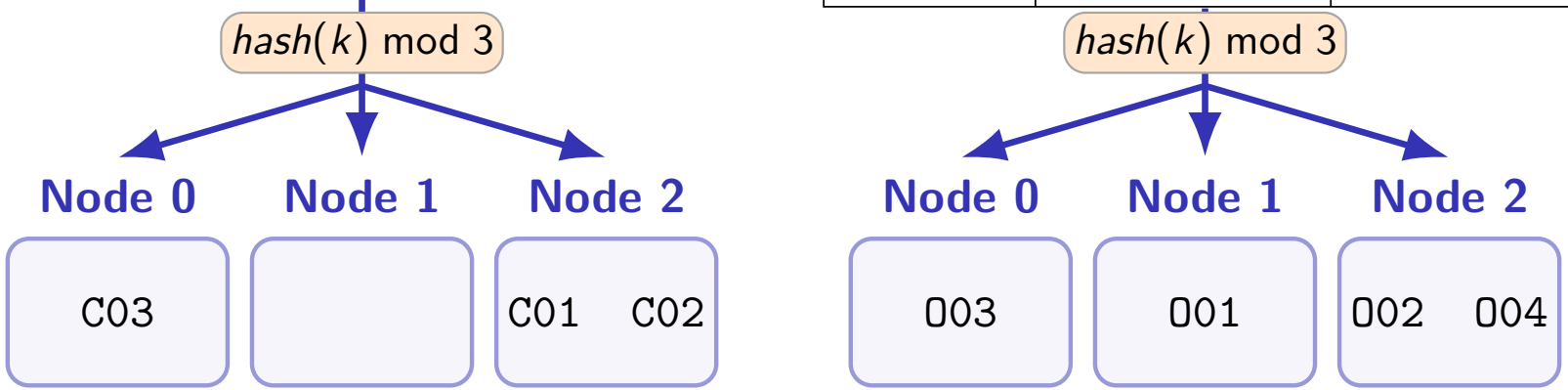
How Do We Split a Database?

Customers

CustomerID	Name
C01	Alex
C02	Sam
C03	Mia

Orders

OrderID	CustomerID	Date
O01	C01	2025-11-01
O02	C02	2025-11-02
O03	C03	2025-11-03
O04	C01	2025-11-04



Definition (Sharding)

Sharding, or **horizontal partitioning**, splits a table by rows and stores each part, a **shard**, on its own node.

Hash partitioning: with N nodes, the row with key k goes to node $hash(k) \bmod N$.

Does Related Data Stay Together?

Each row is placed by its own key: a customer by its CustomerID, an order by its OrderID.

	Node 0	Node 1	Node 2
Customers	C03		C01 C02
Orders	O03	O01	O04 O02

Does Related Data Stay Together?

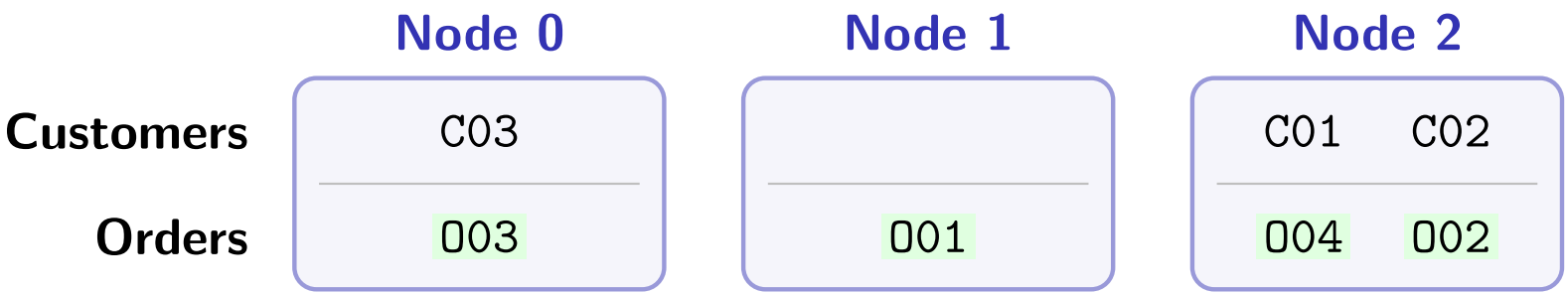
Each row is placed by its own key: a customer by its CustomerID, an order by its OrderID.

	Node 0	Node 1	Node 2
Customers	C03		C01 C02
Orders	O03	O01	O04 O02

- Reading Alex's orders asks nodes 1 and 2.
- Each new order goes wherever its own key sends it.

Does Related Data Stay Together?

Each row is placed by its own key: a customer by its CustomerID, an order by its OrderID.



- Reading Alex's orders asks nodes 1 and 2.
- Each new order goes wherever its own key sends it.
- Reading the orders from 1 to 4 Nov asks all three nodes.

⚠ As with a hash index, a range of keys or dates asks every node.

Does Related Data Stay Together?

Each row is placed by its own key: a customer by its CustomerID, an order by its OrderID.

	Node 0	Node 1	Node 2
Customers	C03		C01 C02
Orders	O03	O01	O04 O02

- Reading Alex's orders asks nodes 1 and 2.
- Each new order goes wherever its own key sends it.
- Reading the orders from 1 to 4 Nov asks all three nodes.

⚠ As with a hash index, a range of keys or dates asks every node.

Hashing rows gives no control over what sits together.

Aggregates for Distribution

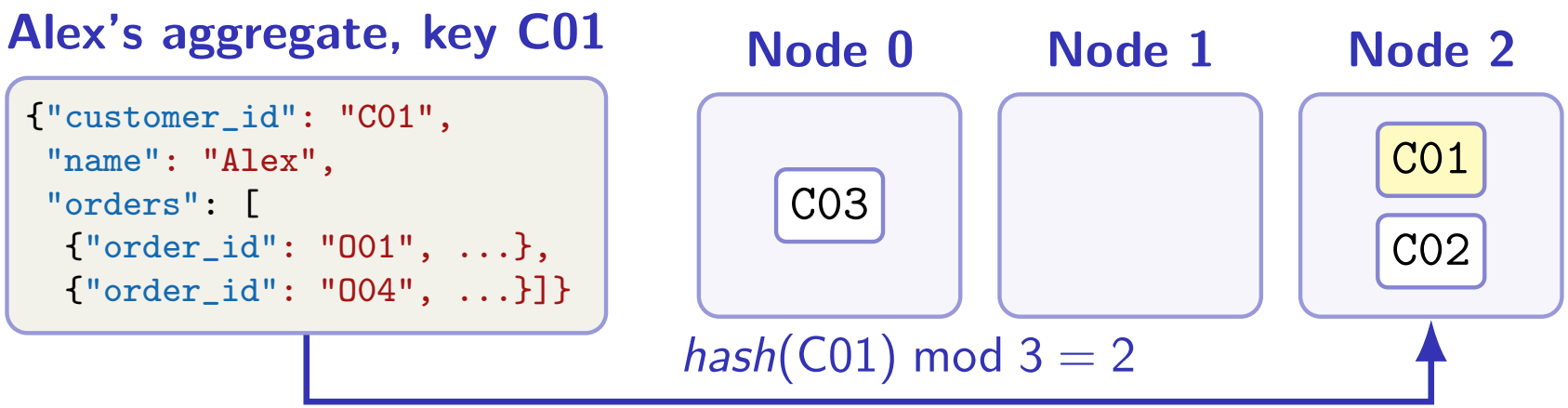
What if we hash bundles of related data?

Alex's aggregate, key C01

```
{"customer_id": "C01",
 "name": "Alex",
 "orders": [
  {"order_id": "O01", ...},
  {"order_id": "O04", ...}]}
```

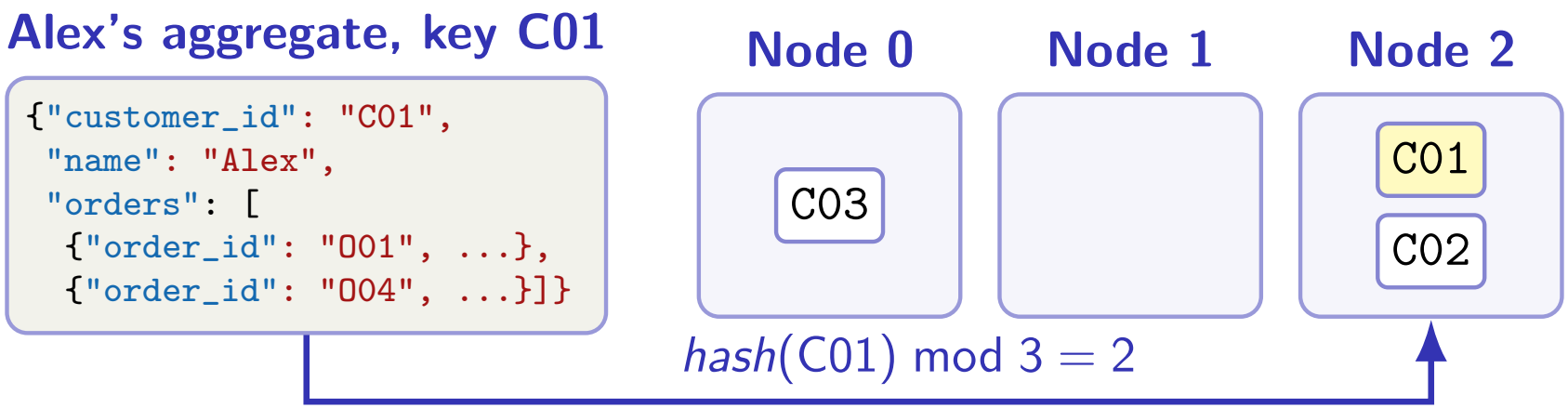
Aggregates for Distribution

What if we hash bundles of related data?



Aggregates for Distribution

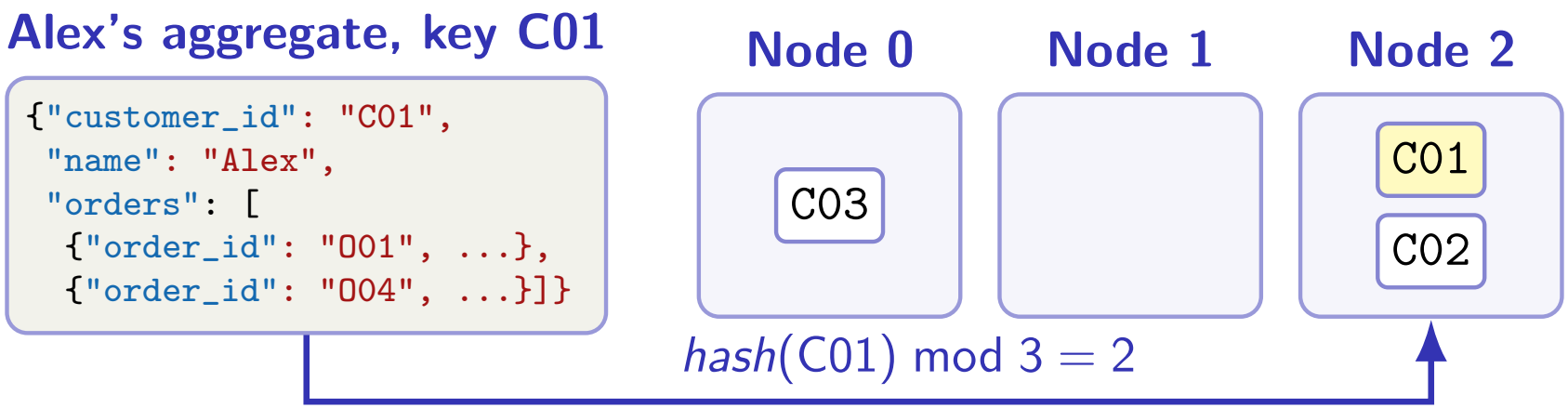
What if we hash bundles of related data?



Reading Alex's orders is one read on one node.

Aggregates for Distribution

What if we hash bundles of related data?



Reading Alex's orders is one read on one node.

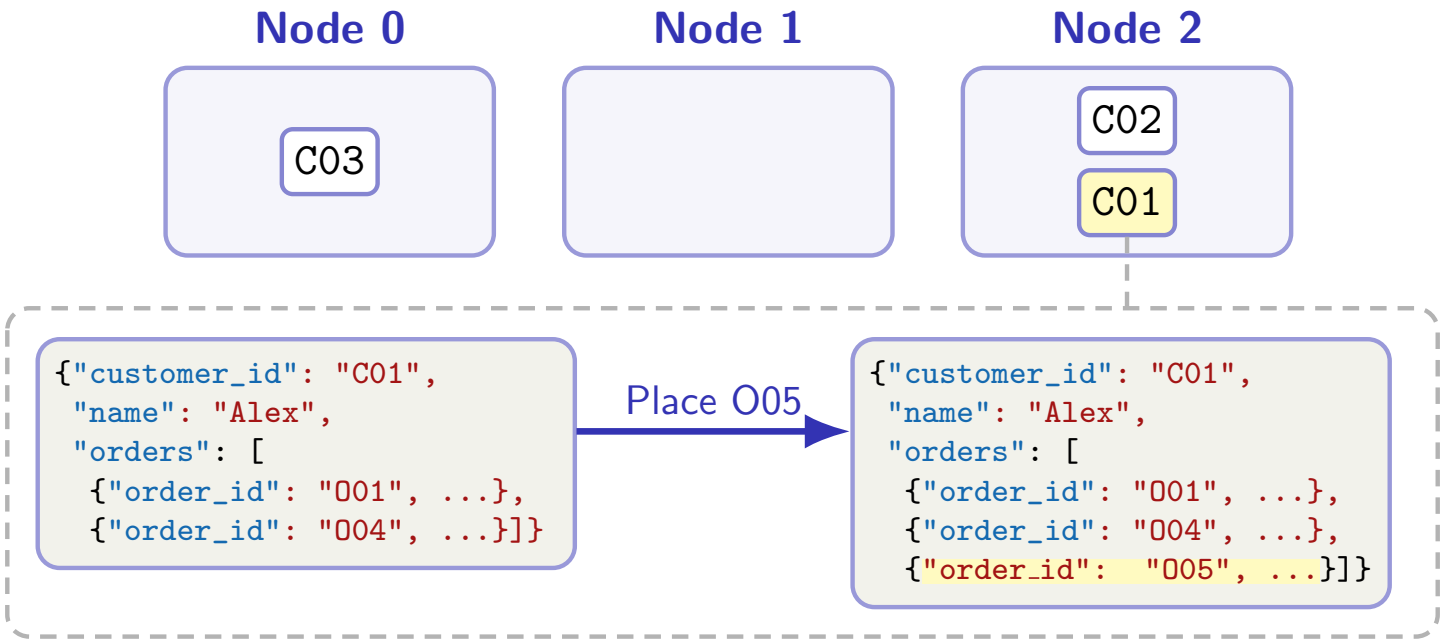
Definition (Aggregate)

An aggregate is also the unit of distribution: each aggregate is stored whole on one node.

Atomic Operations on Aggregates

Operations on a single aggregate are **atomic**:

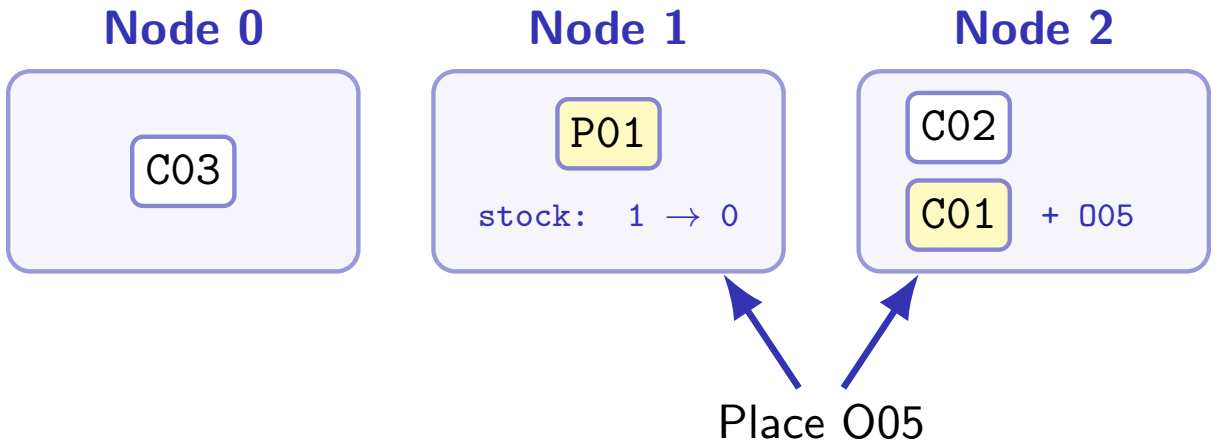
- The entire operation is completed as a single unit.
- No partial updates, even after a crash.



An aggregate is never left half-updated, simplifying error handling and recovery.

Cross-Aggregate Operations

O05 also lowers the Coffee Machine's stock: one operation, two aggregates.

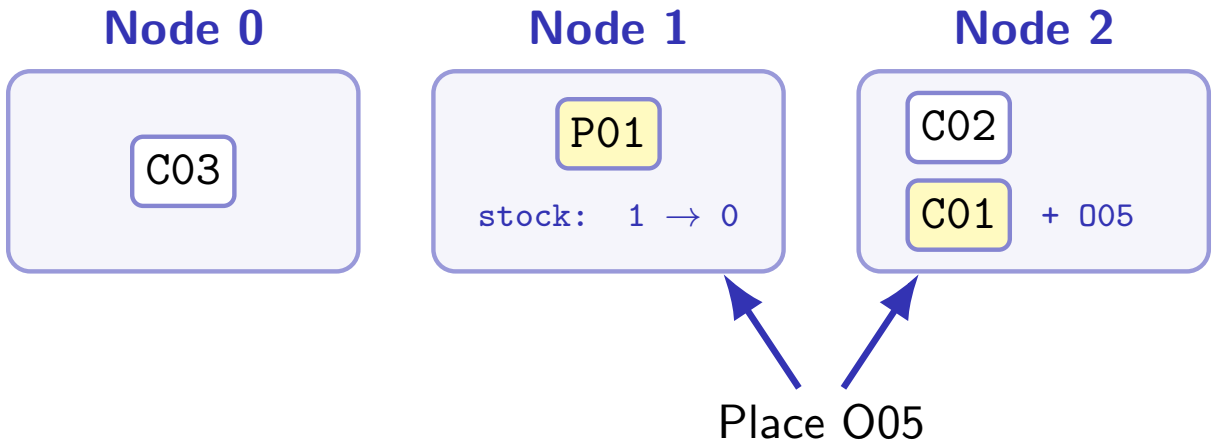


⚠ Across aggregates, no atomicity to count on

- **Crash between the writes:** O05 recorded, stock not lowered.
- **Two writers:** two customers buy the last Coffee Machine.

Cross-Aggregate Operations

O05 also lowers the Coffee Machine's stock: one operation, two aggregates.



⚠ Across aggregates, no atomicity to count on

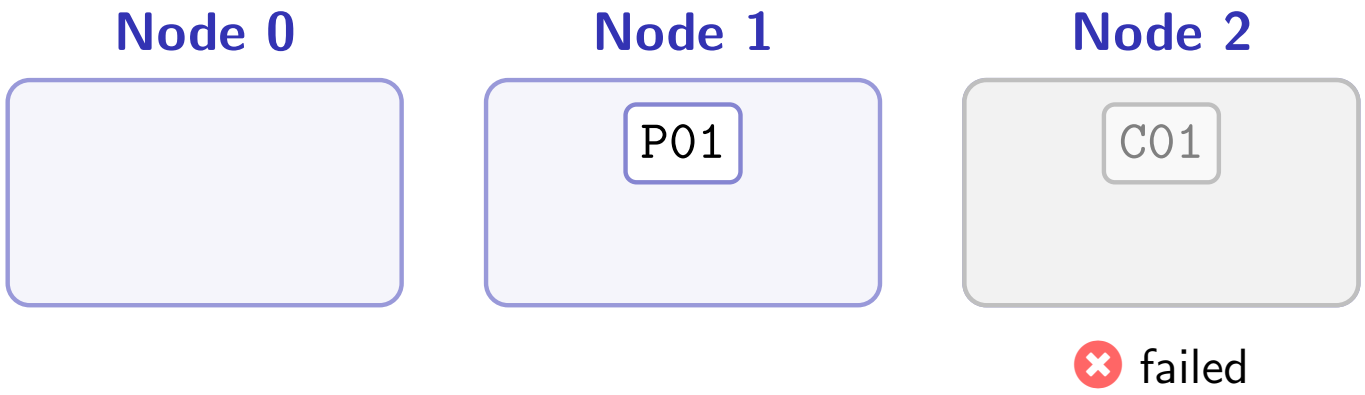
- **Crash between the writes:** O05 recorded, stock not lowered.
- **Two writers:** two customers buy the last Coffee Machine.

Too small, and one operation spans several aggregates.
Too large, and unrelated writes wait.

Replication and the CAP Theorem

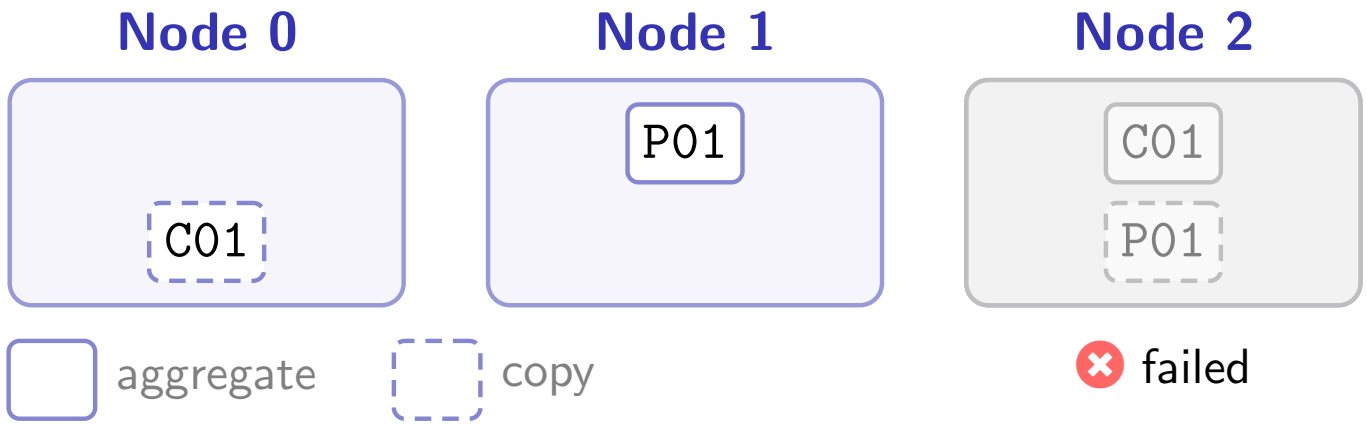
Replication

What if a node **fails**? Its aggregates can no longer be read or written.



Replication

What if a node **fails**? Its aggregates can no longer be read or written.

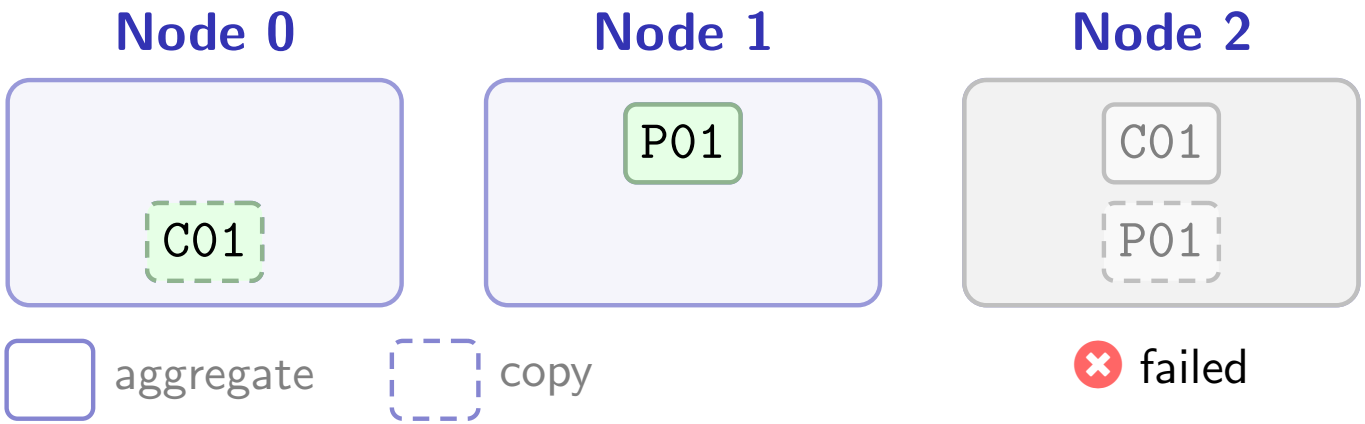


Definition (Replication)

Replication stores each aggregate on **several nodes**, as copies.

Replication

What if a node **fails**? Its aggregates can no longer be read or written.



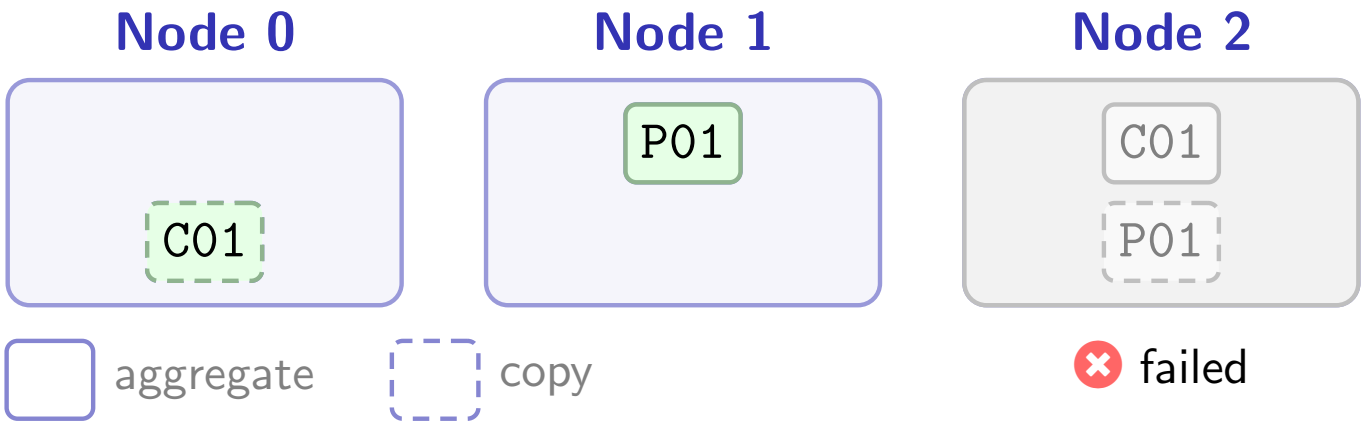
Definition (Replication)

Replication stores each aggregate on **several nodes**, as copies.

Each aggregate on two of the three nodes survives one failure, and reads can go to either copy.

Replication

What if a node **fails**? Its aggregates can no longer be read or written.



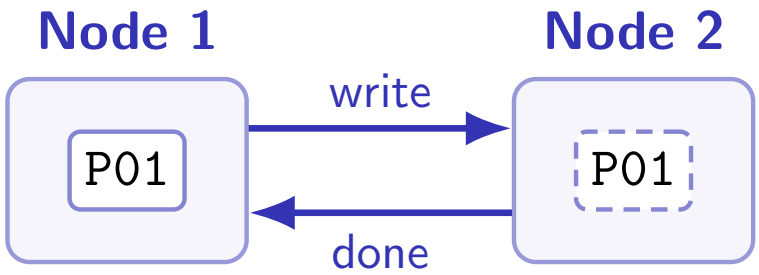
Definition (Replication)

Replication stores each aggregate on **several nodes**, as copies.

Each aggregate on two of the three nodes survives one failure, and reads can go to either copy.

The **cost**: every write must reach both copies.

What If a Message Is Lost?

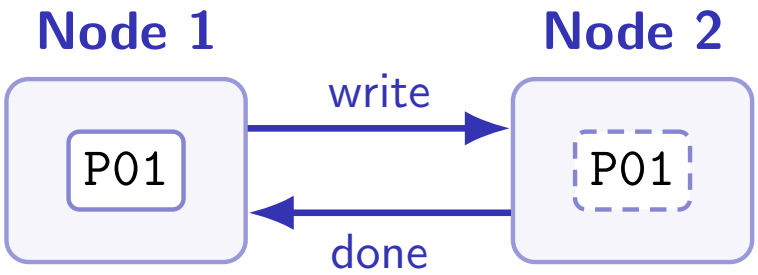


The write is answered

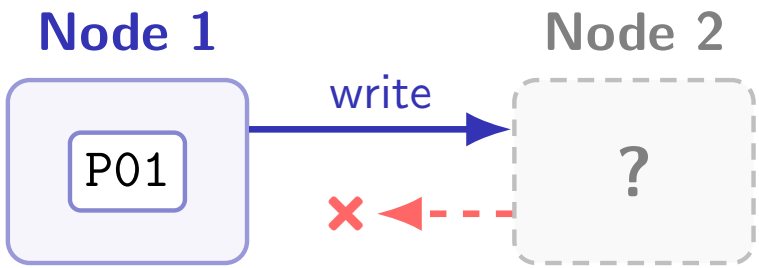
How does a write reach the other copies?

- As messages over the network, from node to node.

What If a Message Is Lost?



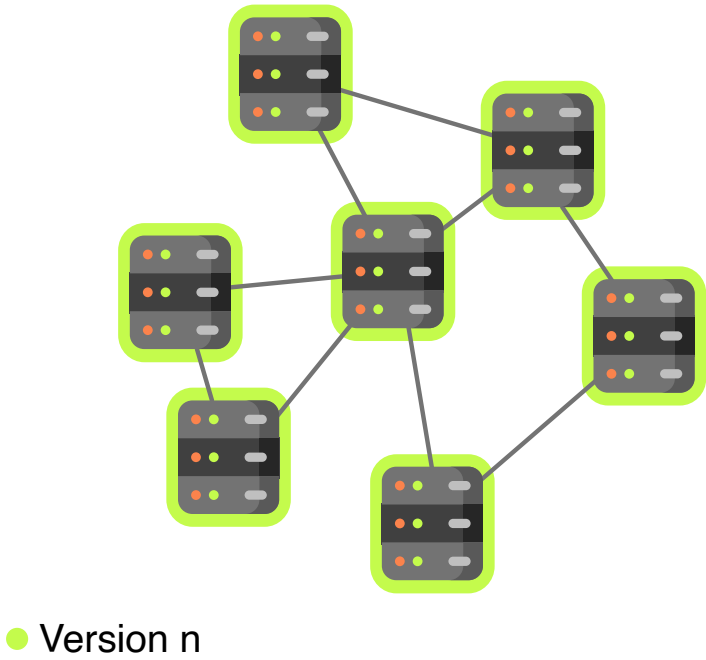
The write is answered



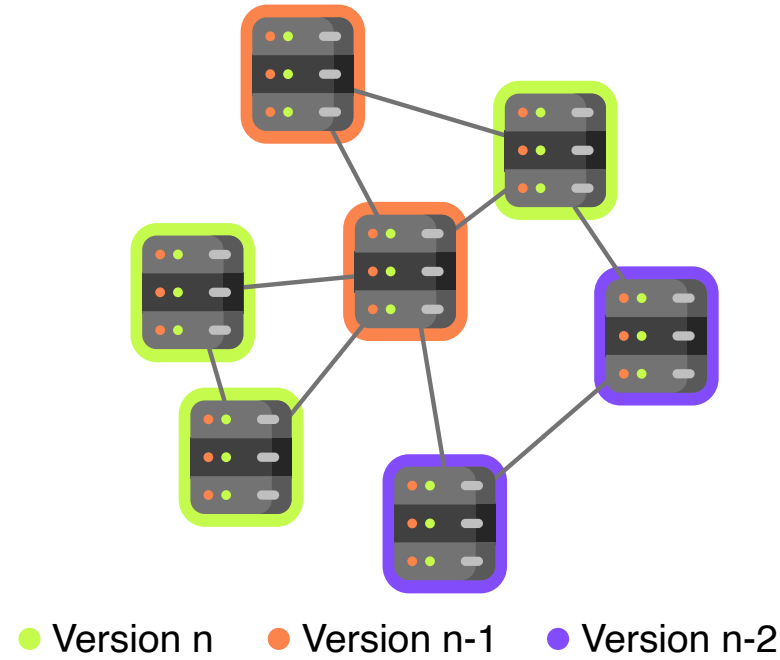
No answer

How does a write reach the other copies?

- As messages over the network, from node to node.
- A lost message gets no answer: the sender cannot tell whether the other node has failed, is cut off, or is just slow.



Consistent

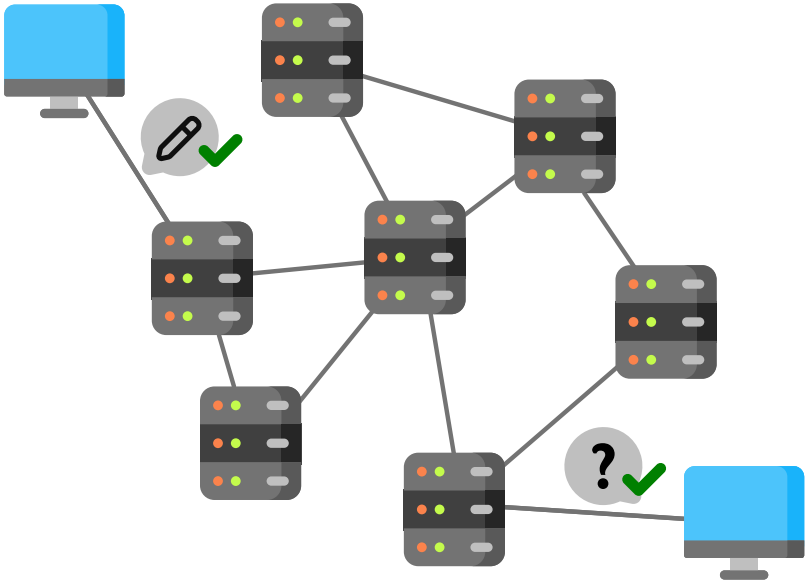


Inconsistent

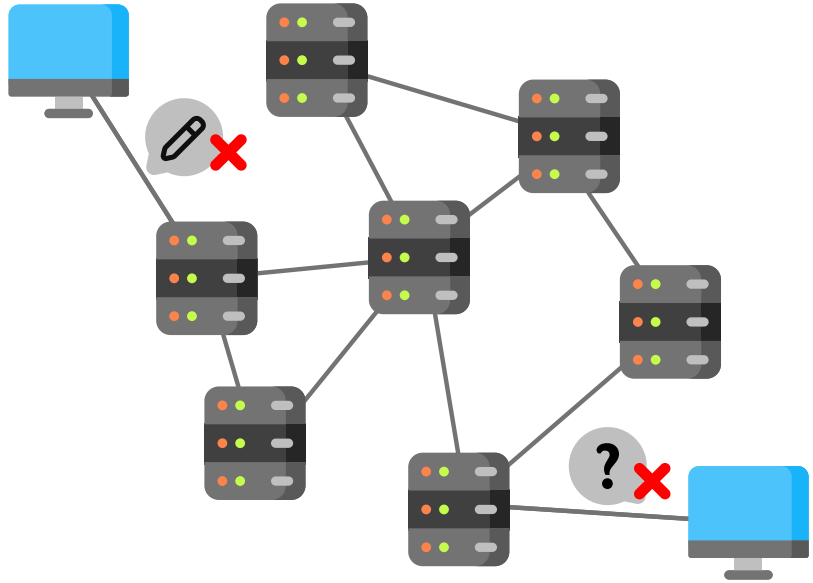
Consistency (C):

- Every read receives the most recent write, or an error.
- Reads behave as if there were a single copy.

⚠ Not ACID's consistency, nor Data Quality's.



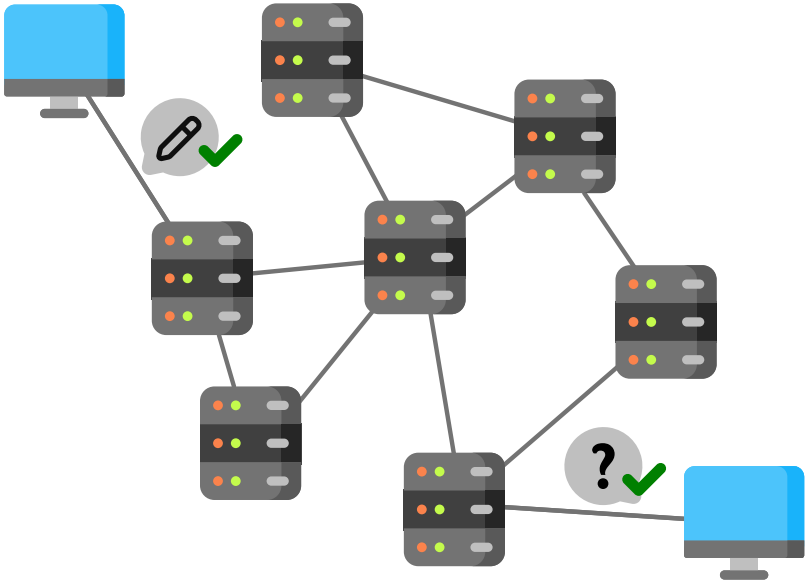
Available



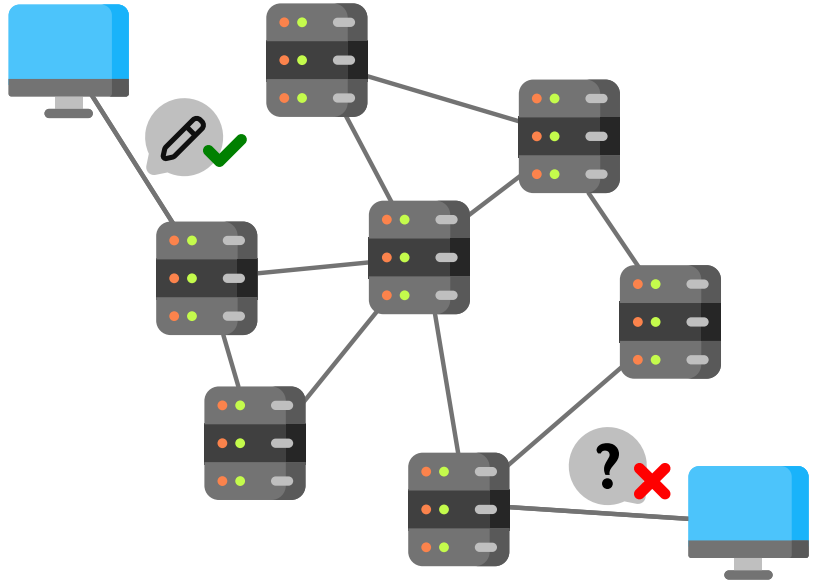
Unavailable

Availability (A):

- Every request received by a non-failing node in the system must result in a response



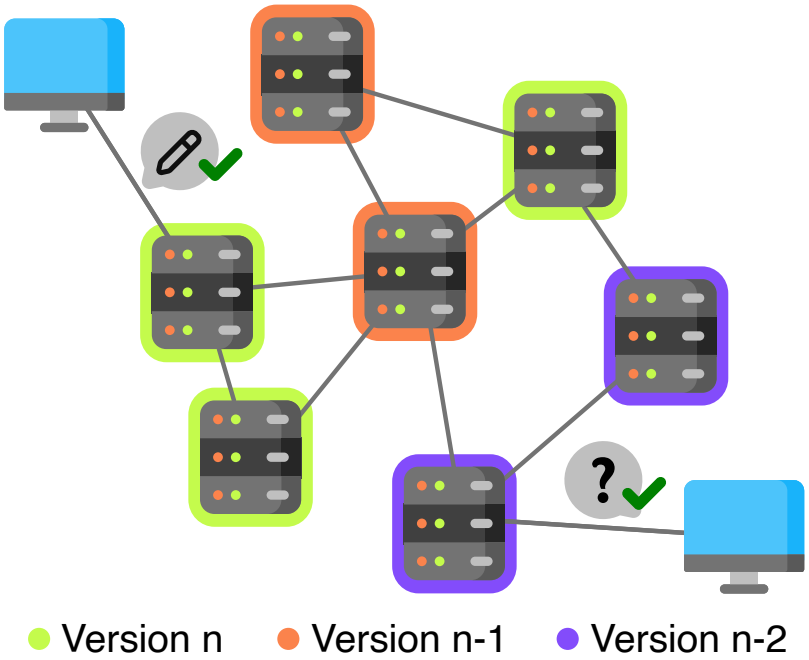
Available



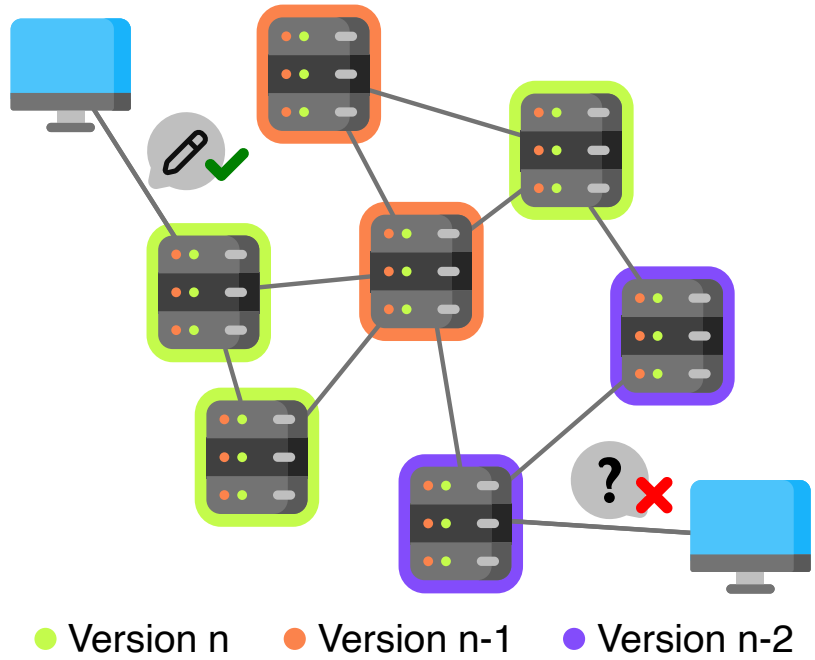
Unavailable

Availability (A):

- Every request received by a non-failing node in the system must result in a response



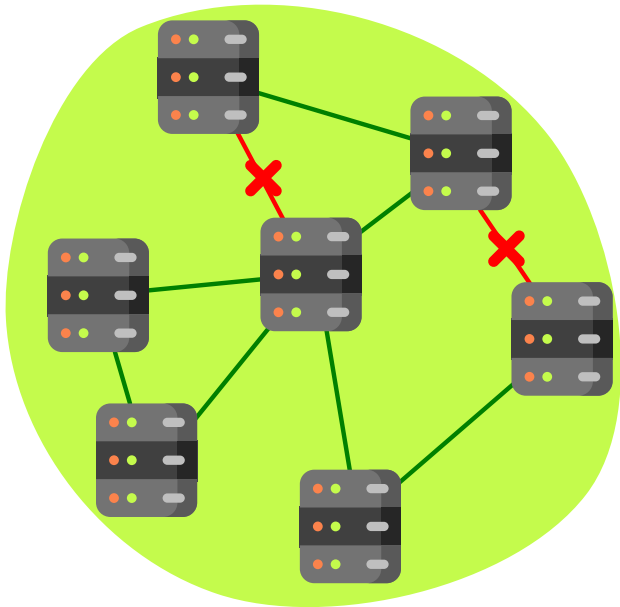
Available



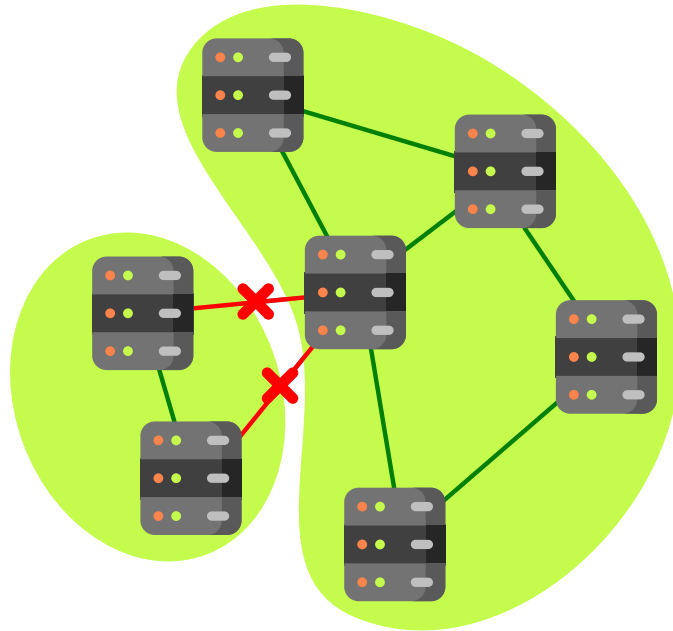
Unavailable

Availability (A):

- Every request received by a non-failing node in the system must result in a response, even if it is not with the latest data.



No partition



Partition

Partition Tolerance (P):

- The system continues to function despite network partitions: groups of nodes that cannot exchange messages.

Theorem (CAP)

During a network partition, a distributed system cannot be both

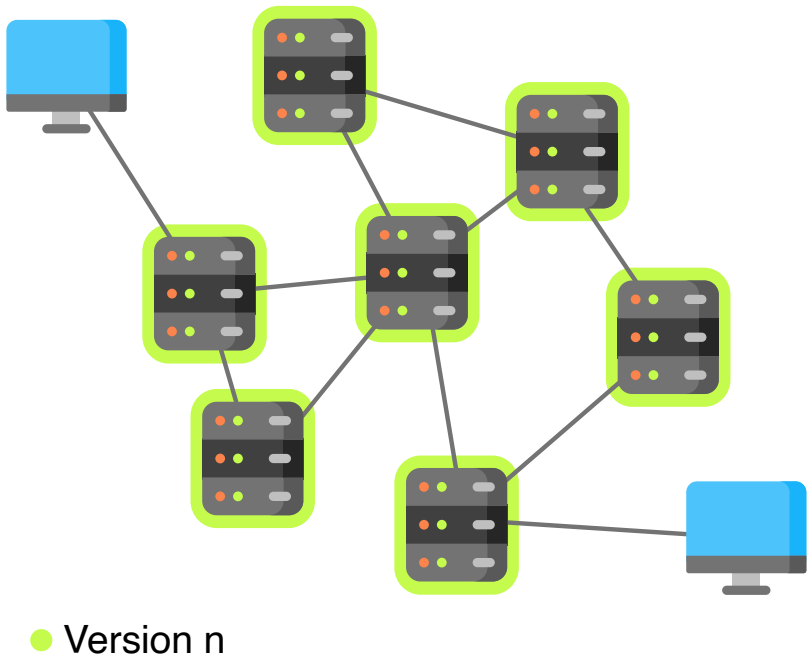
- **C***onsistent and*
- **A***vailable.*



Conjectured by Eric Brewer[3]

[3] Armando Fox and Eric Brewer. "Harvest, Yield, and Scalable Tolerant Systems". In: **Proceedings of the Seventh Workshop on Hot Topics in Operating Systems**. Mar. 1999, pp. 174–178

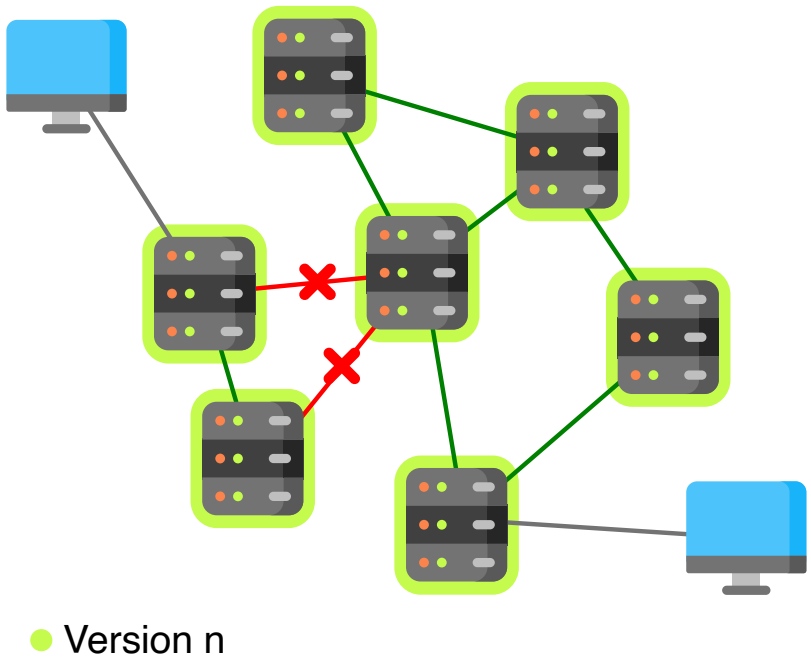
Sketch of the proof from Gilbert and Lynch[4]



1. By contradiction, consider a system S that fulfills the three properties.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

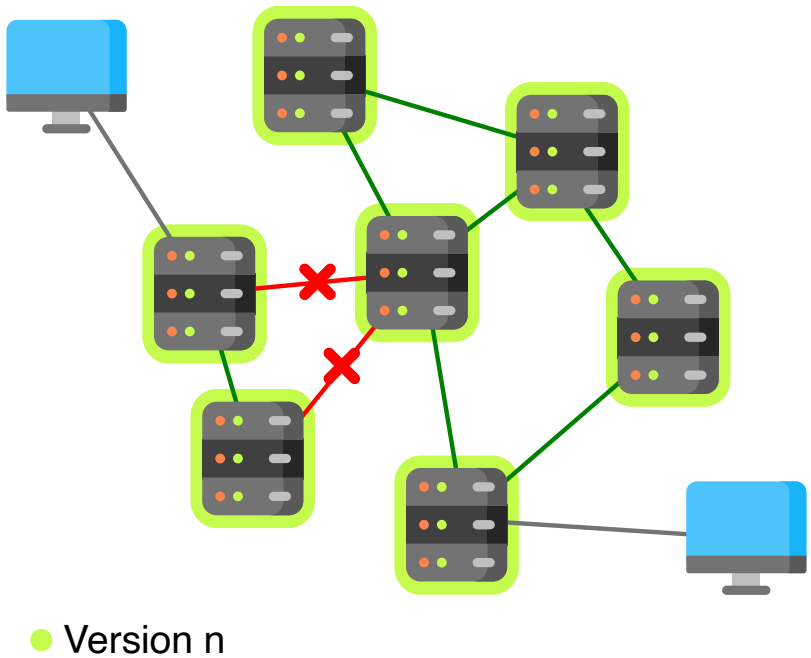
Sketch of the proof from Gilbert and Lynch[4]



2. Partition the system into two subsystems S_1 and S_2.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

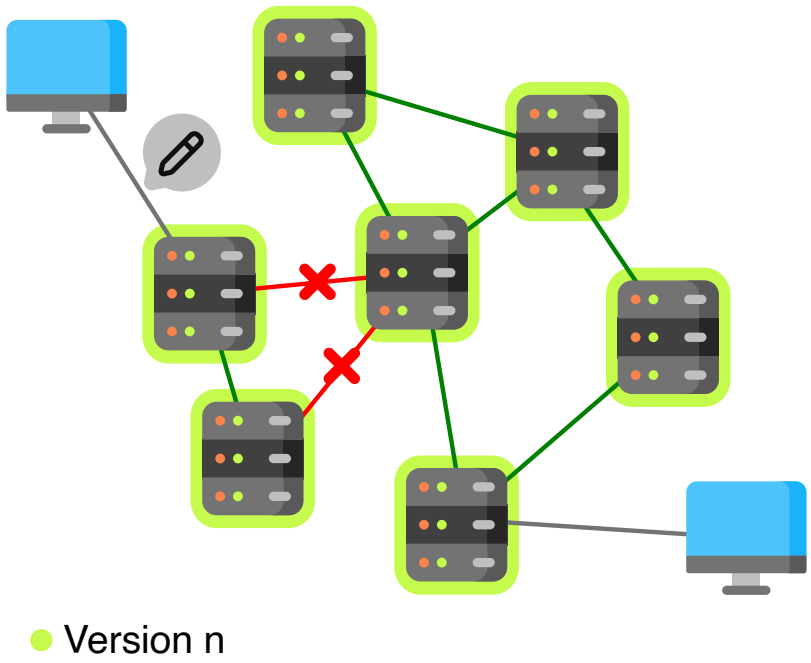
Sketch of the proof from Gilbert and Lynch[4]



3. By **partition tolerance**, the system continues to function.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

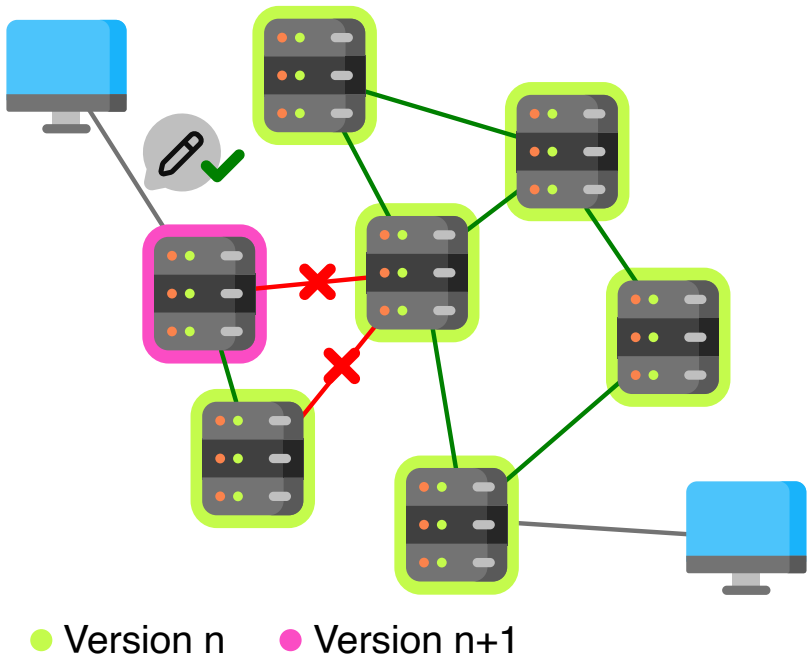
Sketch of the proof from Gilbert and Lynch[4]



4. Client c_1 requests to write new data to a node in S_1.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

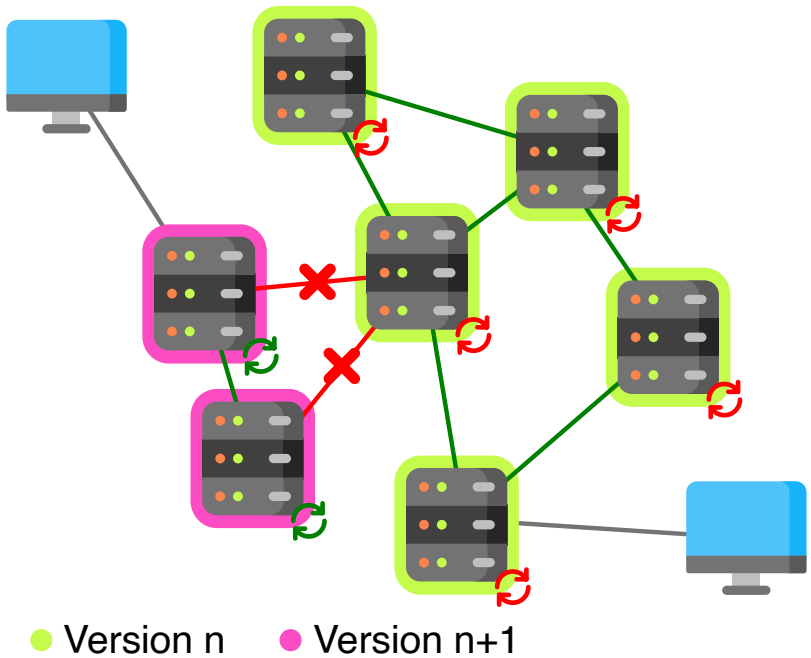
Sketch of the proof from Gilbert and Lynch[4]



5. By **availability**, the system handles the request and writes the new data.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

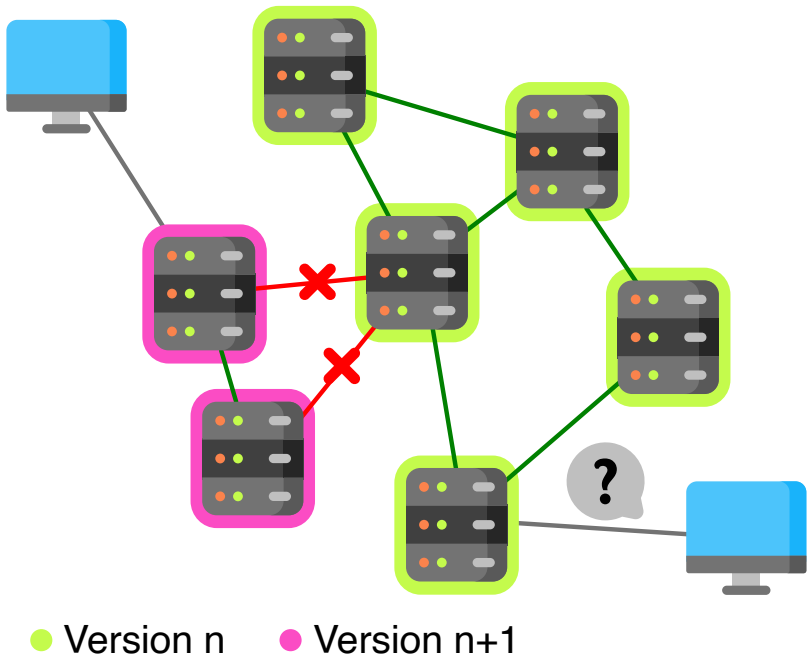
Sketch of the proof from Gilbert and Lynch[4]



6. Since the network is partitioned, the data cannot be replicated to the nodes in S_2.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

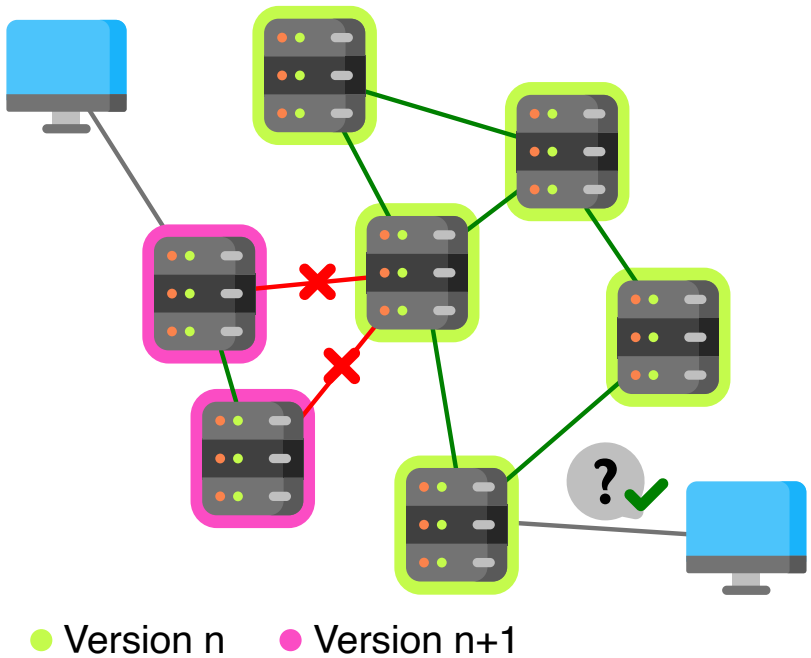
Sketch of the proof from Gilbert and Lynch[4]



7. Client c_2 requests to read the data from a node in S_2.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

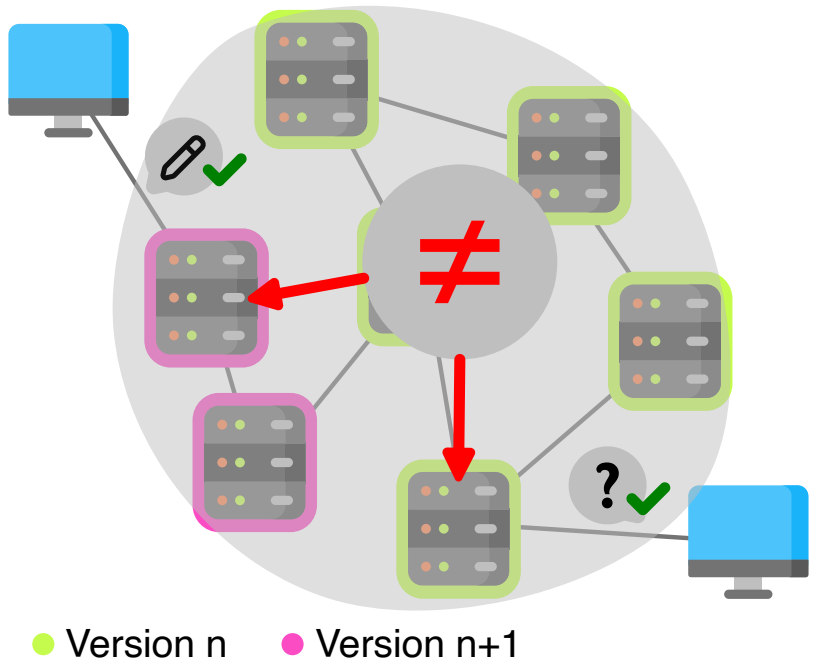
Sketch of the proof from Gilbert and Lynch[4]



8. By **availability**, the system handles the request and outputs the old data.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

Sketch of the proof from Gilbert and Lynch[4]

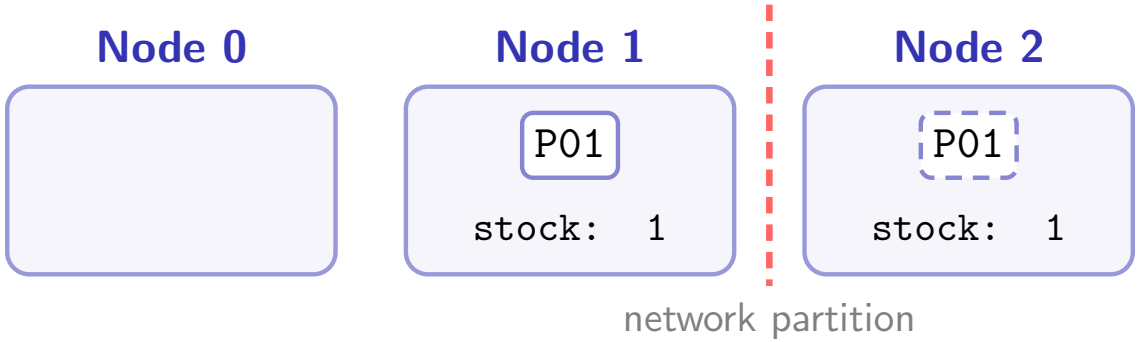


9. The second request yields an **inconsistent** answer.

[4] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: **SIGACT News** 33.2 (2002), pp. 51–59.

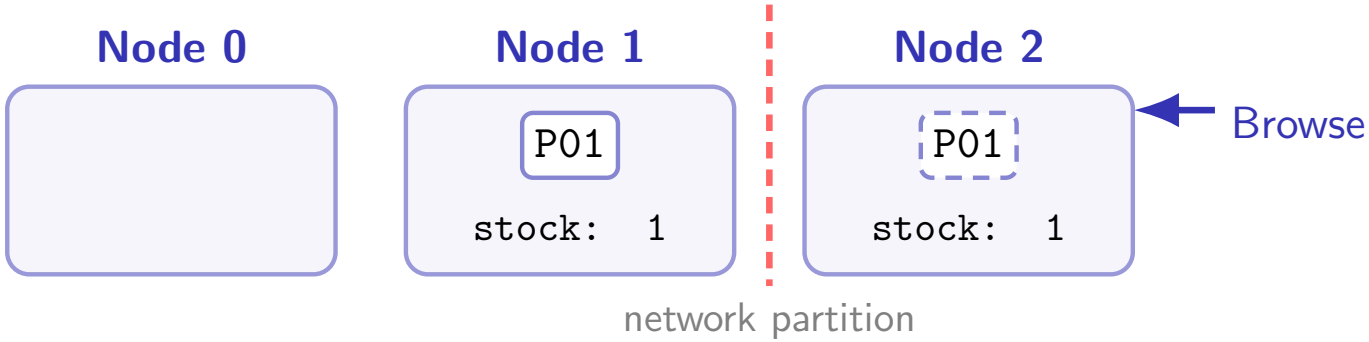
C or A, Operation by Operation

Must the whole system pick C or A once and for all?



C or A, Operation by Operation

Must the whole system pick C or A once and for all?

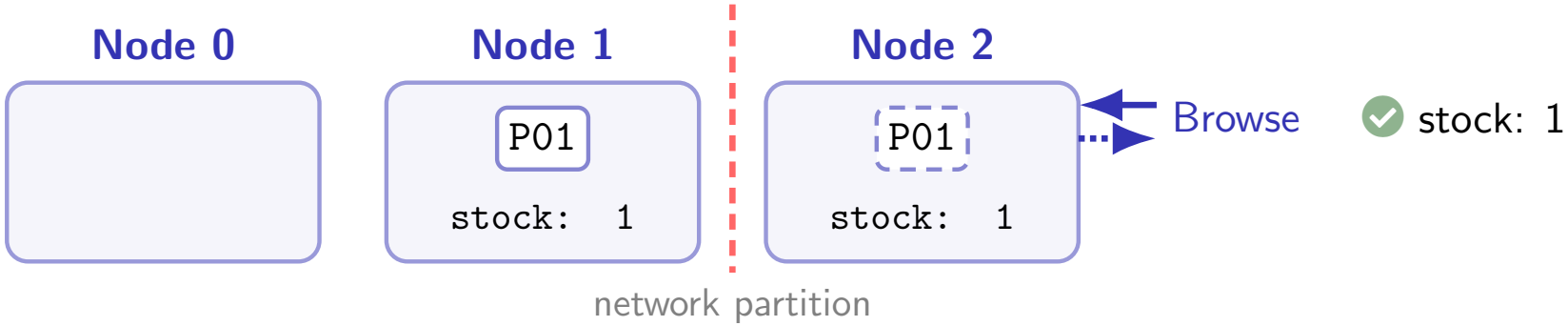


Poll

Browse the catalog: C or A?

C or A, Operation by Operation

Must the whole system pick C or A once and for all?

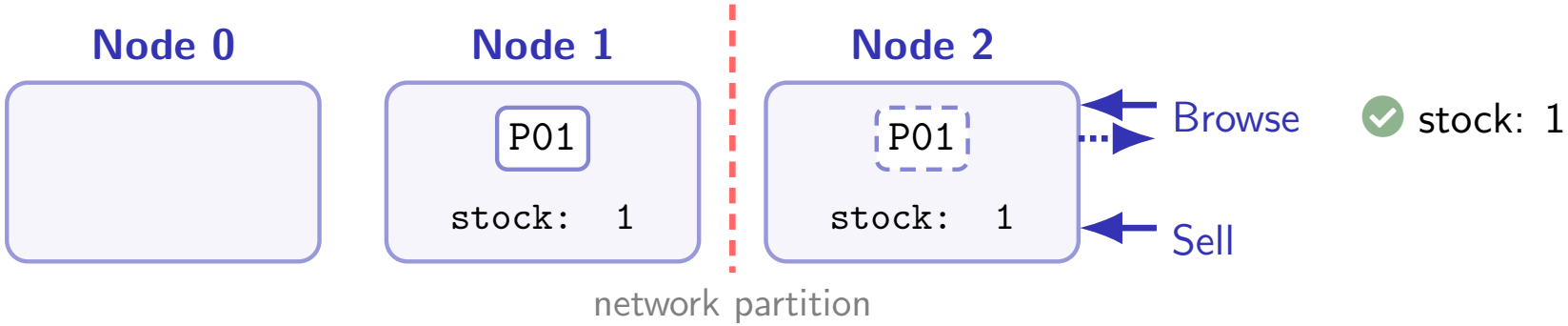


Poll

Browse the catalog: C or A?
Availability: node 2 answers from its copy, even if the stock is out of date.

C or A, Operation by Operation

Must the whole system pick C or A once and for all?



Poll

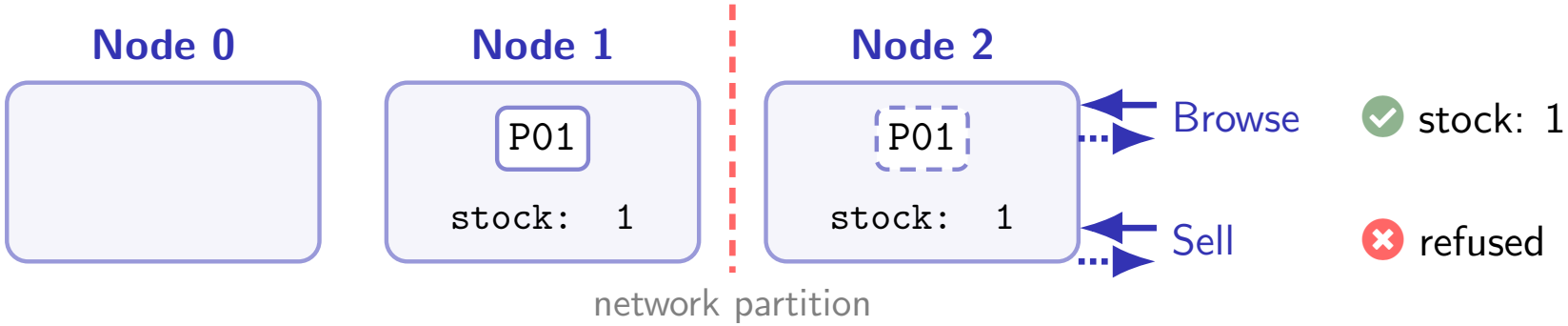
Browse the catalog: C or A?
Availability: node 2 answers from its copy, even if the stock is out of date.

Poll

Sell the last one: C or A?

C or A, Operation by Operation

Must the whole system pick C or A once and for all?



Poll

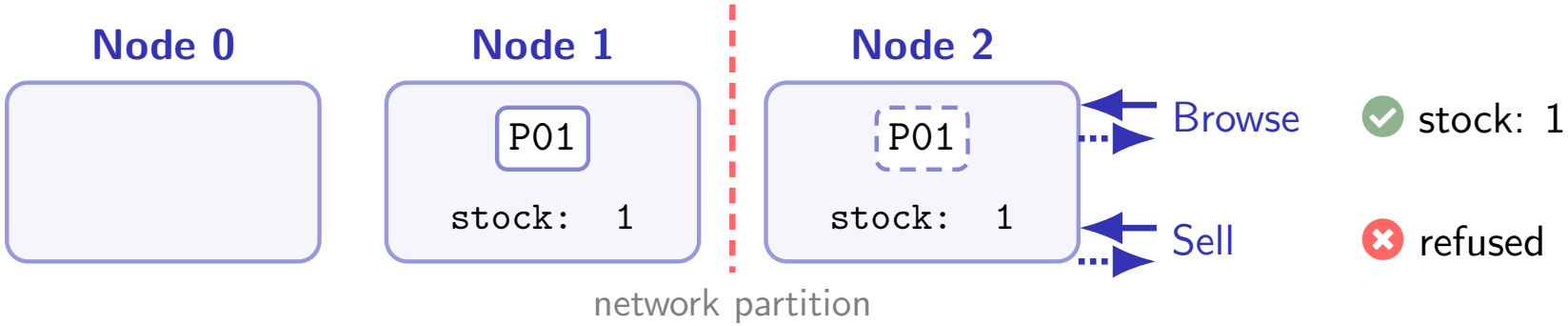
Browse the catalog: C or A?
Availability: node 2 answers from its copy, even if the stock is out of date.

Poll

Sell the last one: C or A?
Consistency: node 2 refuses. Otherwise each copy might sell it.

C or A, Operation by Operation

Must the whole system pick C or A once and for all?



Poll

Browse the catalog: C or A?
Availability: node 2 answers from its copy, even if the stock is out of date.

Poll

Sell the last one: C or A?
Consistency: node 2 refuses. Otherwise each copy might sell it.

The choice comes up only during a partition, and it is made operation by operation.

Conclusion

Aggregate-Based Databases

Denormalization

- Copies of a fact for faster reads
- Every copy to update on a write

Distribution

- One node per aggregate, by hashing its key
- Related data stays together

Aggregate Update

- All or nothing inside one aggregate
- One operation, one node

Cross-Aggregate

- Two nodes: no atomicity
- Boundaries: not too small, not too large

Replication

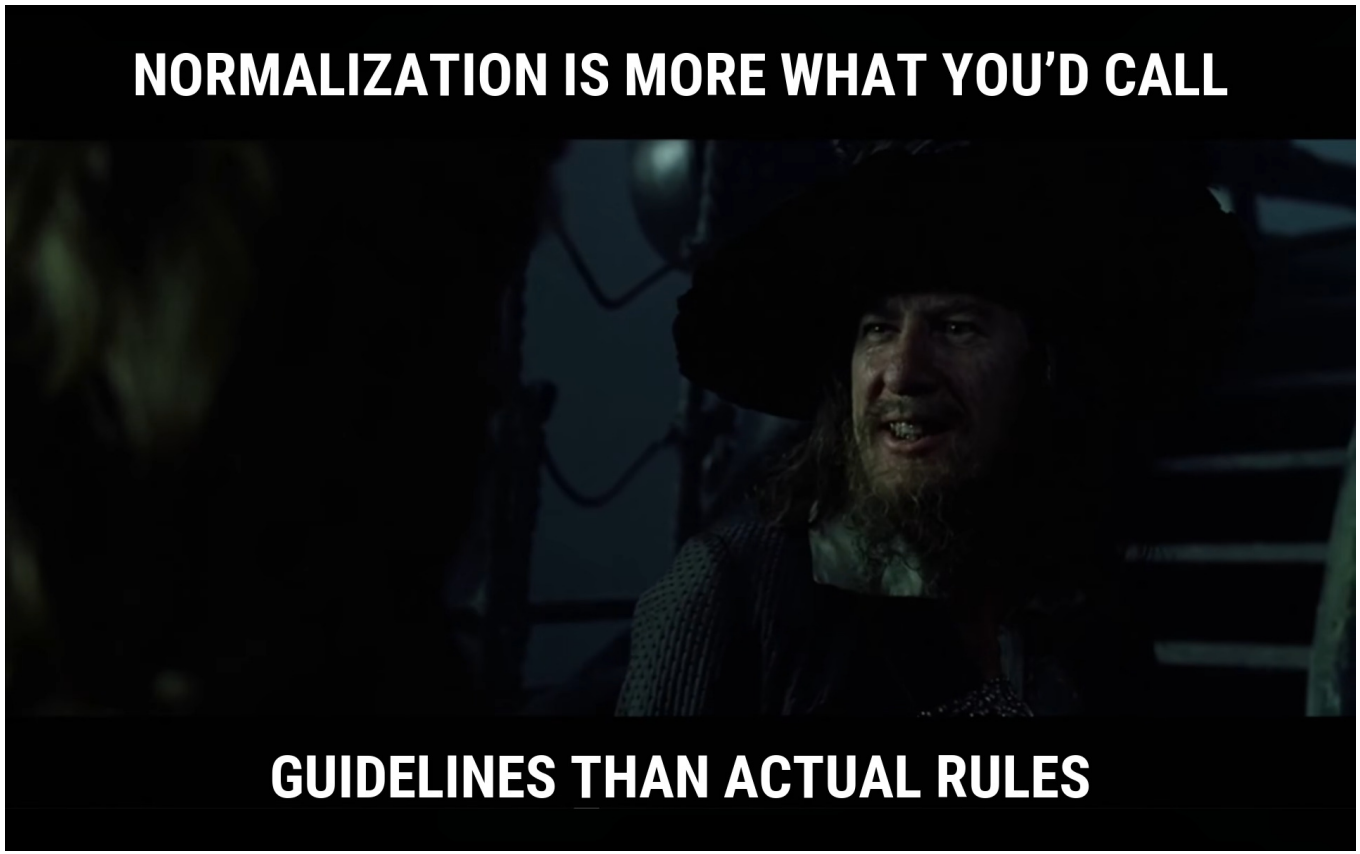
- Each aggregate on two nodes
- Survives a failure: every write reaches both copies

Design

- Aggregates follow the operations
- C or A, operation by operation

Takeaways: Denormalization and Aggregates

1. **Denormalization trades writes for reads:** faster reads, but the anomalies come back on writes.
2. **The physical level comes first:** indexes, join algorithms and views speed up a query, schema unchanged.
3. **An aggregate breaks 1NF:** related data as one record with one key, shaped like the application's object.
4. **Hash the aggregate:** it sits on one node, and an operation on it is atomic.
5. **Replication survives a failure:** every write must reach every copy.
6. **CAP:** during a partition, choose C or A, operation by operation.



Breaking the rules on purpose.