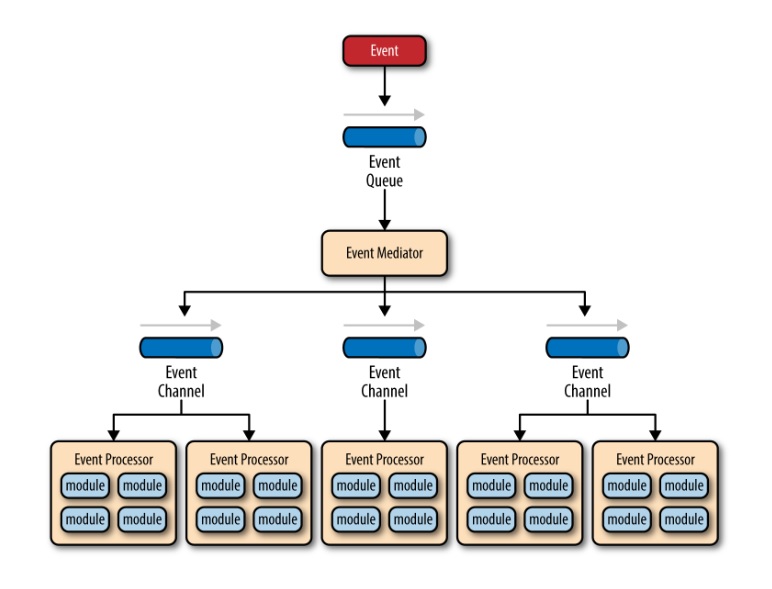
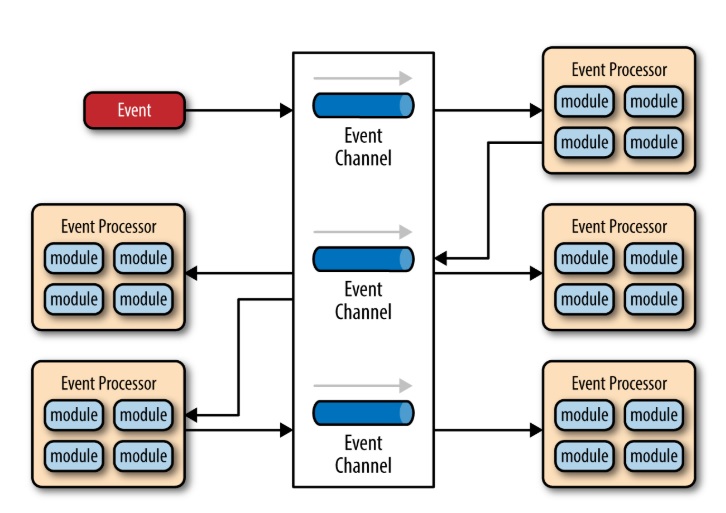
**Appunti stage**

**Event-Driven architecture**The event-driven architecture pattern is a popular distributed asynchronous architecture pattern used to produce highly scalable applications. It is also highly adaptable and can be used for small applications and as well as large, complex ones. The event-driven architecture is made up of highly decoupled, single-purpose event processing components that asynchronously receive and process events.   
The event-driven architecture pattern consists of two main topologies, the **mediator** and the **broker**. The mediator topology is commonly used when you need to orchestrate multiple steps within an event through a central mediator, whereas the broker topology is used when you want to chain events together without the use of a central mediator.

**Mediator Topology**The mediator topology is useful for events that have multiple steps and require some level of orchestration to process the event. For example, a single event to place a stock trade might require you to first validate the trade, then check the compliance of that stock trade against various compliance rules, assign the trade to a broker, calculate the commission, and finally place the trade with that broker. All of these steps would require some level of orchestration to determine the order of the steps and which ones can be done serially and in parallel.    
There are four main types of architecture components within the mediator topology: **event queues**, an **event mediator**, **event channels**, and **event processors**. The event flow starts with a client sending an event to an event queue, which is used to transport the event to the event mediator. The event mediator receives the initial event and orchestrates that event by sending additional asynchronous events to event channels to execute each step of the process. Event processors, which listen on the event channels, receive the event from the event mediator and execute specific business logic to process the event.  
There are two types of events within this pattern: an initial event and a processing event. The initial event is the original event received by the mediator, whereas the processing events are ones that are generated by the mediator and received by the event-processing components.  
The **event-mediator** component is responsible for orchestrating the steps contained within the initial event. For each step in the initial event, the event mediator sends out a specific processing event to an event channel, which is then received and processed by the event processor. It is important to note that the event mediator doesn’t actually perform the business logic necessary to process the initial event; rather, it knows of the steps required to process the initial event. **Event channels** are used by the event mediator to asynchronously pass specific processing events related to each step in the initial event to the event processors. The event channels can be either message queues or message topics, although message topics are most widely used with the mediator topology so that processing events can be processed by multiple event processors (each performing a different task based on the processing event received).  
The **event processor** components contain the application business logic necessary to process the processing event. Event processors are self-contained, independent, highly decoupled architecture components that perform a specific task in the application or system. While the granularity of the event-processor component can vary from fine-grained (e.g., calculate sales tax on an order) to coarse-grained (e.g., process an insurance claim), it is important to keep in mind that in general, each event-processor component should perform a single business task and not rely on other event processors to complete its specific task.

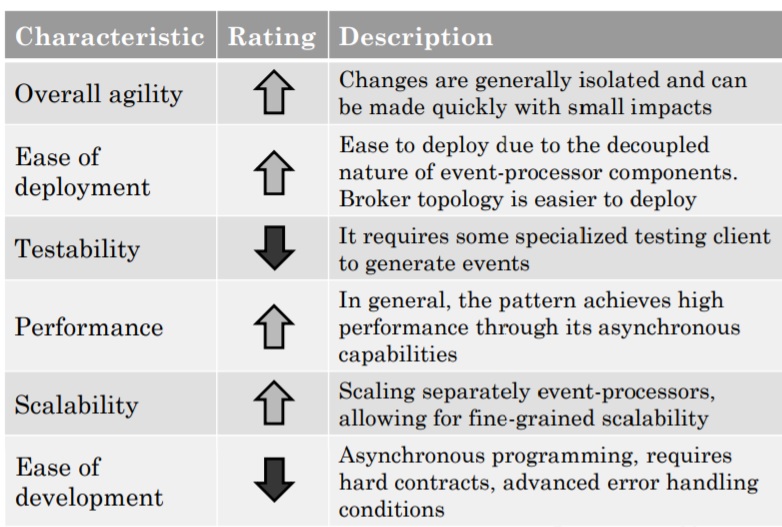
**Broker Topology**The broker topology differs from the mediator topology in that there is no central event mediator; rather, the message flow is distributed across the event processor components in a chain-like fashion through a lightweight message broker. This topology is useful when you have a relatively simple event processing flow.

  
 As you can see from the diagram, there is no central event-mediator component controlling and orchestrating the initial event; rather, each event-processor component is responsible for processing an event and publishing a new event indicating the action it just performed. For example, an event processor that balances a portfolio of stocks may receive an initial event called stock split. Based on that initial event, the event processor may do some portfolio rebalancing, and then publish a new event to the broker called rebalance portfolio, which would then be picked up by a different event processor.

**Conclusion**

One consideration to take into account when choosing this architecture pattern is the lack of atomic transactions for a single business process. Because event processor components are highly decoupled and distributed, it is very difficult to maintain a transactional unit of work across them. For this reason, when designing your application using this pattern, you must continuously think about which events can and can’t run independently and plan the granularity of your event processors accordingly (if you are using separate processors for something that should be an undivided transaction this is probably not the right pattern for your application).  
One of the most difficult aspects of the event-driven architecture pattern is the creation, maintenance, and governance of the event-processor component contracts. Each event usually has a specific contract associated with it.

**Pattern analysis**



* **Overall agility(High)**: is the ability to respond quickly to a constantly changing environment. Since event-processor components are single-purpose and completely decoupled from other event processor components, changes are generally isolated to one or a few event processors and can be made quickly without impacting other components.
* **Ease of deployment(High)**: this pattern is relatively easy to deploy due to the decoupled nature of the event-processor components. The broker topology tends to be easier to deploy than the mediator topology, primarily because the event mediator component is somewhat tightly coupled to the event processors: a change in an event processor component might also require a change in the event mediator, requiring both to be deployed for any given change.
* **Testability(Low)**: while individual unit testing is not overly difficult, it does require some sort of specialized testing client or testing tool to generate events. Testing is also complicated by the asynchronous nature of this pattern.
* **Performance(High):** the pattern achieves high performance through its asynchronous capabilities.
* **Scalability(High):** is naturally achieved in this pattern through highly independent and decoupled event processors.
* **Ease of development(Low):** development can be somewhat complicated due to the asynchronous nature of the pattern as well as contract creation and the need for more advanced error handling.

**Event collaboration**  
The well-known observer pattern is an example of Event Collaboration.  
[Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) results in very loose coupling which makes it particularly easy to add new components to a system without needing to modify existing components. The downside of [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) is that it's very hard to understand the collaboration. [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html) is much more implicit - which makes it much harder to debug when something unexpected happens.  
Events just communicate that something happened - with an event you let a system know that something has happened.  
(command dico al sistema remoto “questo è quello che dovresti fare”, event dico al sistema remoto “questo è cambiato” indicando il cambiamento.

**Introduzione**

“Esistono 4 pattern ad eventi, un sistema Event-Driven esiste se viene applicato almeno uno di questi pattern” cit di Martin Fowler.  
Possiamo usare più pattern allo stesso tempo per ottenere migliori performance senza rendere troppo complicato il nostro sistema?   
Cos’è un evento?

1. **Event notification**

This happens when a system sends event messages to notify other systems of a change in its domain (Observer pattern con push model?). A key element of event notification is that the source system doesn't really care much about the response. Event notification is nice because it implies a **low level of coupling.** The problem is that it can be hard to see such a flow as it's not explicit in any program text. Often the only way to figure out this flow is from monitoring a live system (this can make it hard to debug and modify such a flow). The danger is that it's very easy to make nicely decoupled systems with event notification, without realizing that you're losing sight of that larger-scale flow.

**N.B.** An event need not carry much data on it, often just some id information and a link back to the sender that can be queried for more information. The receiver knows something has changed, may get some minimal information on the nature of the change, but then issues a request back to the sender to decide what to do next.

**Vantaggi**: disaccoppiamento, flessibilità  
**Svantaggi**: difficile debug e capire il flusso degli eventi nei sistemi GUI

1. **Event-Carried State Transfer**

This pattern shows up when you want to update clients of a system in such a way that they don't need to contact the source system in order to do further work. A customer management system might fire off events whenever a customer changes their details (such as an address) with events that contain details of the data that changed. A recipient can then update it's own copy of customer data with the changes, so that it never needs to talk to the main customer system in order to do its work in the future.  
An obvious down-side of this pattern is that there's lots of data schlepped around and lots of copies. What we gain is greater resilience, since the recipient systems can function if the customer system is becomes unavailable. We reduce latency, as there's no remote call required to access customer information. We don't have to worry about load on the customer system to satisfy queries from all the consumer systems, but it does involve more complexity on the receiver since it has to sort out maintaining all the state, when it's usually easier just to call the sender for more information when needed.

**Vantaggi**: disaccoppiamento, migliori performance, minori richieste al supplier  
**Svantaggi**: lack of consistency, comunicazioni più onerose

1. **Event-Sourcing**

***“Event Sourcing ensures that all changes to application state are stored as a sequence of events.”*** Cit. Martin Fowler  
Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

The core idea of [event sourcing](https://martinfowler.com/eaaDev/EventSourcing.html) is that whenever we make a change to the state of a system, we record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future. The **event store** becomes the principal source of truth, and the system state is purely derived from it. Another common mistake is to assume that everyone using an event-sourced system should understand and access the event log to determine useful data. But knowledge of the event log can be limited. When working with an event log, it is often useful to build snapshots of the working copy so that you don't have to process all the events from scratch every time you need a working copy. Indeed, there is a duality here, we can look at the event log as either a list of changes, or as a list of states. We can derive one from the other. Version-control systems often mix snapshots and deltas in their event log in order to get the best performance (vedi Git). Event-sourcing has many interesting benefits, which easily come to mind when thinking of the value of version-control systems. The event log provides a strong audit capability (accounting transactions are an event source for account balances). We can recreate historic states by replaying the event log up to a point. We can explore alternative histories by injecting hypothetical events when replaying. Event sourcing does have its problems. Replaying events becomes problematic when results depend on interactions with outside systems. We have to figure out how to deal with changes in the schema of events over time.

The key to Event Sourcing is that we guarantee that all changes to the domain objects are initiated by the event objects. This leads to a number of facilities that can be built on top of the event log:

* **Complete Rebuild**: We can discard the application state completely and rebuild it by re-running the events from the event log on an empty application.
* **Temporal Query**: We can determine the application state at any point in time. Notionally we do this by starting with a blank state and rerunning the events up to a particular time or event. We can take this further by considering multiple time-lines (analogous to branching in a version control system).
* **Event Replay**: If we find a past event was incorrect, we can compute the consequences by reversing it and later events and then replaying the new event and later events. (Or indeed by throwing away the application state and replaying all events with the correct event in sequence.) The same technique can handle events received in the wrong sequence - a common problem with systems that communicate with asynchronous messaging.

**Structuring the Event Handler Logic**

There are a number of choices about where to put the logic for handling events. The primary choice is whether to put the logic in [Transaction Scripts](https://martinfowler.com/eaaCatalog/transactionScript.html) or [Domain Model](https://martinfowler.com/eaaCatalog/domainModel.html). As usual [Transaction Scripts](https://martinfowler.com/eaaCatalog/transactionScript.html) are better for simple logic and a [Domain Model](https://martinfowler.com/eaaCatalog/domainModel.html) is better when things get more complicated.  
A good way to think of this is that there are two responsibilities involved. *Processing domain logic* is the business logic that manipulates the application. *Processing selection logic* is the logic that chooses which chunk of processing domain logic should run depending on the incoming event. You can combine these together, essentially this is the [Transaction Script](https://martinfowler.com/eaaCatalog/transactionScript.html) approach, but you can also separate them. Next is whether to put the processing selection logic in the event object itself, or have a separate event processor object. The problem with the processor is that it necessarily runs different logic depending on the type of event. All things being equal you want the processing selection logic in the event itself, since that's the thing that varies with the type of event.

**Reversing Events**As well as events playing themselves forwards, it's also often useful for them to be able to reverse themselves. Reversal is the most straightforward when the event is cast in the form of a difference. An example of this would be "add $10 to Martin's account" as opposed to "set Martin's account to $110". In the former case I can reverse by just subtracting $10, but in the latter case I don't have enough information to recreate the past value of the account.   
If the input events don't follow the difference approach, then the event should ensure it stores everything needed for reversal during processing. You can do this by storing the previous values on any value that is changed, or by calculating and storing differences on the event.  
It's worth remembering that all the capabilities of reversing events can be done instead by reverting to a past snapshot and replaying the event stream. As a result reversal is never absolutely needed for functionality. However it may make a big difference to efficiency since you may often be in a position where reversing a few events is much more efficient than using forward play on a lot of events.  
**N.B.** Per questioni di consistenza dei dati è preferibile ritornare allo stato precedente ad un preciso evento e poi eseguire quelli successivi, piuttosto che usare operazioni inverse per tornare indietro.

**External Updates**

One of the tricky elements to Event Sourcing is how to deal with external systems that don't follow this approach (and most don't). You get problems when you are sending modifier messages to external systems and when you are receiving queries from other systems.

Many of the advantages of Event Sourcing stem from the ability to replay events at will, but if these events cause update messages to be sent to external systems, then things will go wrong because those external systems don't know the difference between real processing and replays.

An easy way to help deal with this is to ensure that your system calls external systems through gateways that can be configured to ensure that no messages go out unless you are processing an event 'for real'.

To handle this you'll need to wrap any external systems with a [Gateway](https://martinfowler.com/eaaCatalog/gateway.html). This in itself isn't too onerous since it's a thoroughly good idea in any case. The gateway has to be a bit more sophisticated so it can deal with any replay processing that the Event Sourcing system is doing. For rebuilds and temporal queries it's usually sufficient for the gateways to be able to be disabled during the replay processing. You want to do this in a way that's invisible to the domain logic.  
  
The problem with external queries is that the data that they return has an effect on the results on handling an event. If I ask for an exchange rate on December 5th and replay that event on December 20th, I will need the exchange rate on Dec 5 not the later one.

**Code Changes**We can think as three broad kinds of code changes here: new features, defect fixes, and temporal logic.

* **New features:** essentially add new capabilities to the system but don't invalidate things that happened before. These can be added pretty freely at any time. If you want to take advantage of the new features with old events you can just reprocess the events and the new results pop up.
* **Bug fixes:** occur when you look at past processing and realize it was incorrect. For internal stuff this is really easy to fix, all you need to do is make the fix and reprocess the events. Your application state is now fixed to what it should have been.
* **Temporal logic:** the third case is where the logic itself changes over time, a rule along the lines of "charge $10 before November 18 and $15 afterwords". This kind of stuff needs to actually go into the domain model itself. The domain model should be able to run events at any time with the correct rules for the event processing. You can do this with conditional logic, but this will get messy if you have much temporal logic. The better route is to hook strategy objects into a [Temporal Property](https://martinfowler.com/eaaDev/TemporalProperty.html).

**Optimizing performance**A replay becomes more and more complex as the number of events that need to be replayed increases. At first glance, the use of event sourcing seems to lead to read accesses becoming increasingly slow. You can take advantage of this situation by saving the currently calculated state as a so-called **snapshot**. The entire history does not always have to be played back all along the way. Usually it is sufficient to start from the last snapshot and only look at the events that have been saved since then. As a snapshot only supplements history, and does not replace it, the older events are still available if they are required for an evaluation.  
  
**N.B.** Al posto di salvare una lista ordinata di eventi salvo una istantanea dello stato in seguito ad ogni evento oppure in determinati istanti. In questo modo non sarà necessario ricostruire, evento dopo evento, lo stato del sistema, basterà quindi riferirsi ad un preciso istante.

Inoltre potrebbe essere una buona idea raggruppare gli eventi/snapshots in base al soggetto(es. per utenti).

**Vantaggi**: audit (revisioni guardando in modo retrospettivo gli eventi), debugging (riportando il sistema ad un determinato istante), storico degli stati/eventi, stati alternativi(non posso fare l’undo ma posso ricreare il sistema ad partire da un preciso istante), memory image(log di eventi al posto di usare database per persistenza dei dati)

**Svantaggi**: difficile comunicazione con servizi esterni, event schema (come salvare gli eventi)

1. **CQRS**

[Command Query Responsibility Segregation](https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf) (CQRS) is an architectural pattern which separates the responsibility for modifying data (Command) from reading them (Query). The formalization of this approach is generally attributed to [Greg Young](https://twitter.com/gregyoung). The use of two different models for writing and reading operations, in scope of CQRS, allows instead to design and optimize each model for its responsibilities. In addition to this, the use of distinct models also allows the selection of the most appropriate technologies. As soon as the reading and writing models are separated, the infrastructure could easily scale to best fit the needs. It often happens that the number of writings in a system is much lower than the readings. Obviously the two models must be synchronized to ensure that the read information are consistent with the written ones.

The justification for CQRS is that in complex domains, a single model to handle both reads and writes gets too complicated, and we can simplify by separating the models.

The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query.

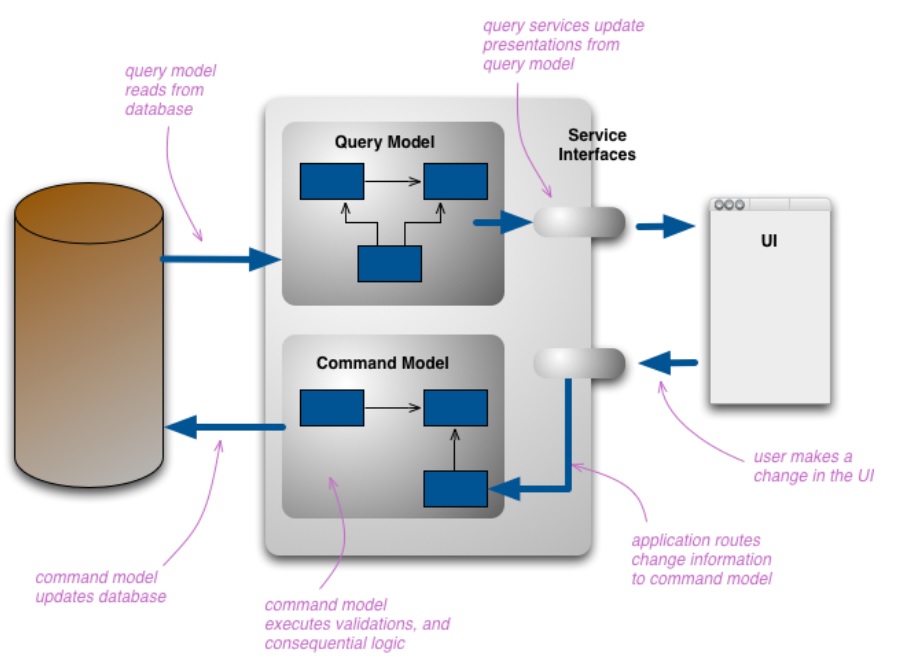
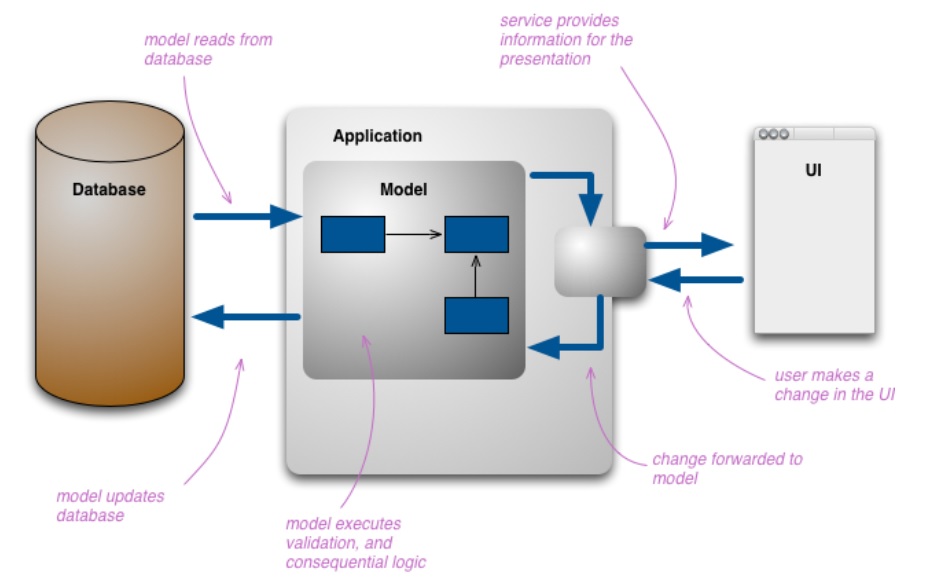
CQRS fits with some other architectural patterns:

* interact with CRUD, we can easily move to a task-based UI
* CQRS fits well with [event-based programming models](https://martinfowler.com/eaaDev/EventNarrative.html). It's common to see CQRS system split into separate services communicating with [Event Collaboration](https://martinfowler.com/eaaDev/EventCollaboration.html). This allows these services to easily take advantage of [Event Sourcing](https://martinfowler.com/eaaDev/EventSourcing.html).
* Having separate models raises questions about how hard to keep those models consistent, which raises the likelihood of using [eventual consistency](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html).
* CQRS is suited to complex domains, the kind that also benefit from [Domain-Driven Design](https://www.amazon.com/gp/product/0321125215?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0321125215)

In particular CQRS should only be used on specific portions of a systemand not the system as a whole.  
Firstly is that a few complex domains may be easier to tackle by using CQRS. The other main benefit is in handling high performance applications. CQRS allows you to separate the load from reads and writes allowing you to scale each independently. If your application sees a big disparity between reads and writes this is very handy. Even without that, you can apply different optimization strategies to the two sides. An example of this is using different database access techniques for read and update.

PRIMA

DOPO



**Vantaggi**: adatto per operazioni CRUD, separazione tra scrittura e lettura  
**Svantaggi**: può aggiungere complessità in sistemi semplici