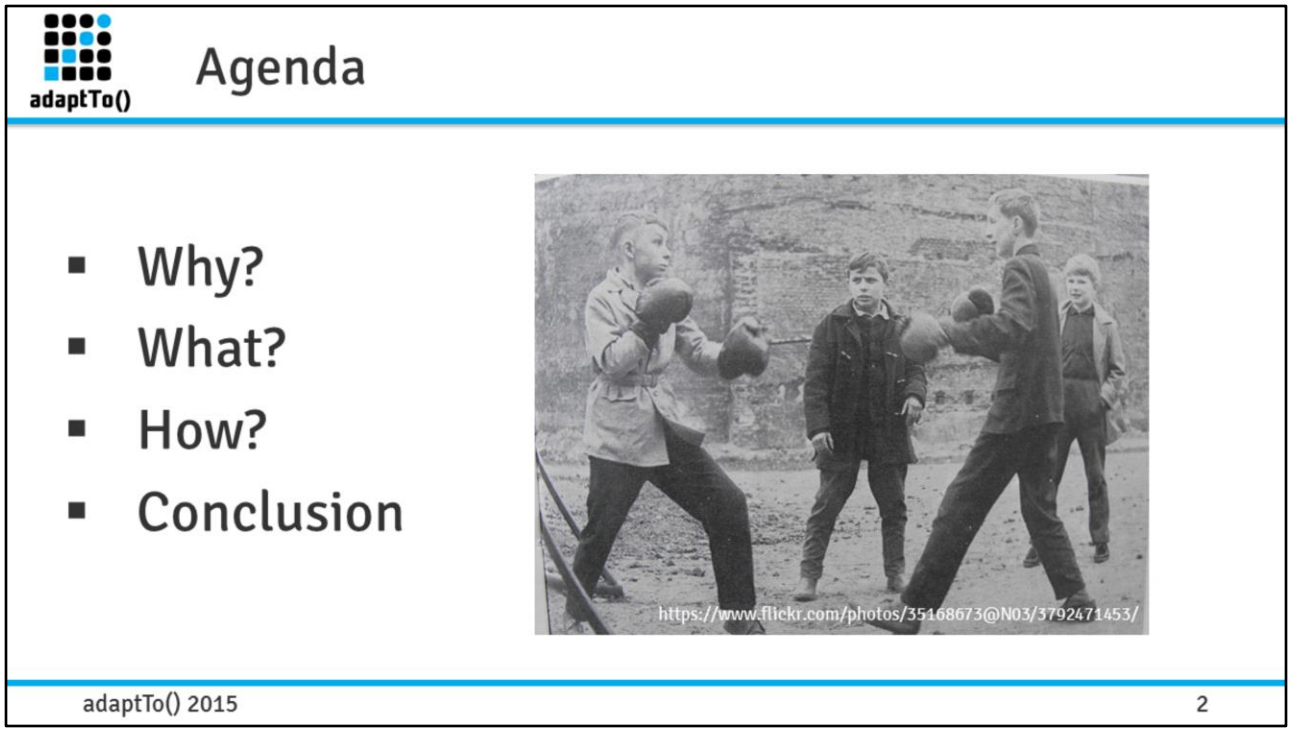
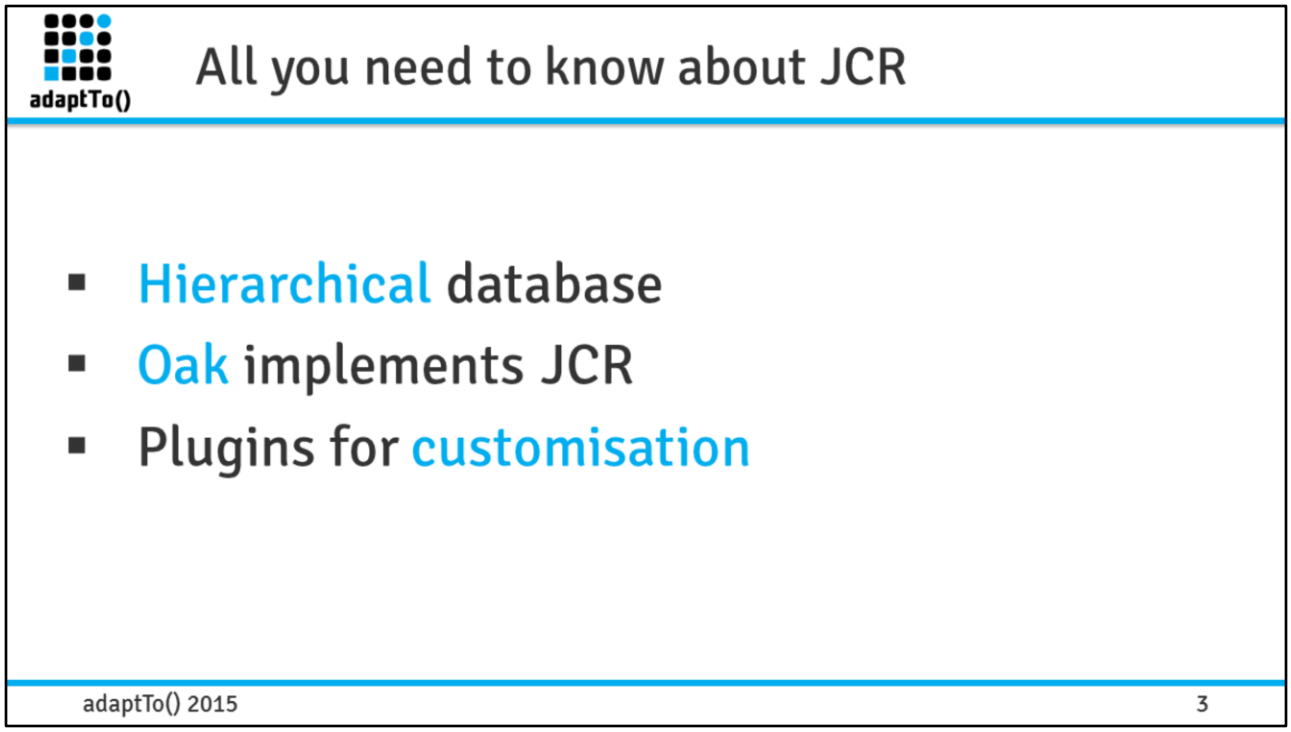
Apache Jackrabbit Oak offers better horizontal scalability and concurrency than its predecessor, Apache Jackrabbit 2. The downside of which is increased chances of conflicts between concurrent updates.

In this session I demonstrate how to deal with such conflicts by taking advantage of Oak's underlying consistency model. I will show how to build functionality like counting, voting, rating, negotiating, bidding, etc. common to collaborative applications. Such functionality traditionally requires some form of global

consensus (e.g. locking, atomic commit protocols, ...). I will show how with Oak it is often possible to avoid conflicts all together by choosing the right content model. For cases where this is not possible I will discuss the mechanisms that Oak provides to deal with conflicts while they occur and after the fact.



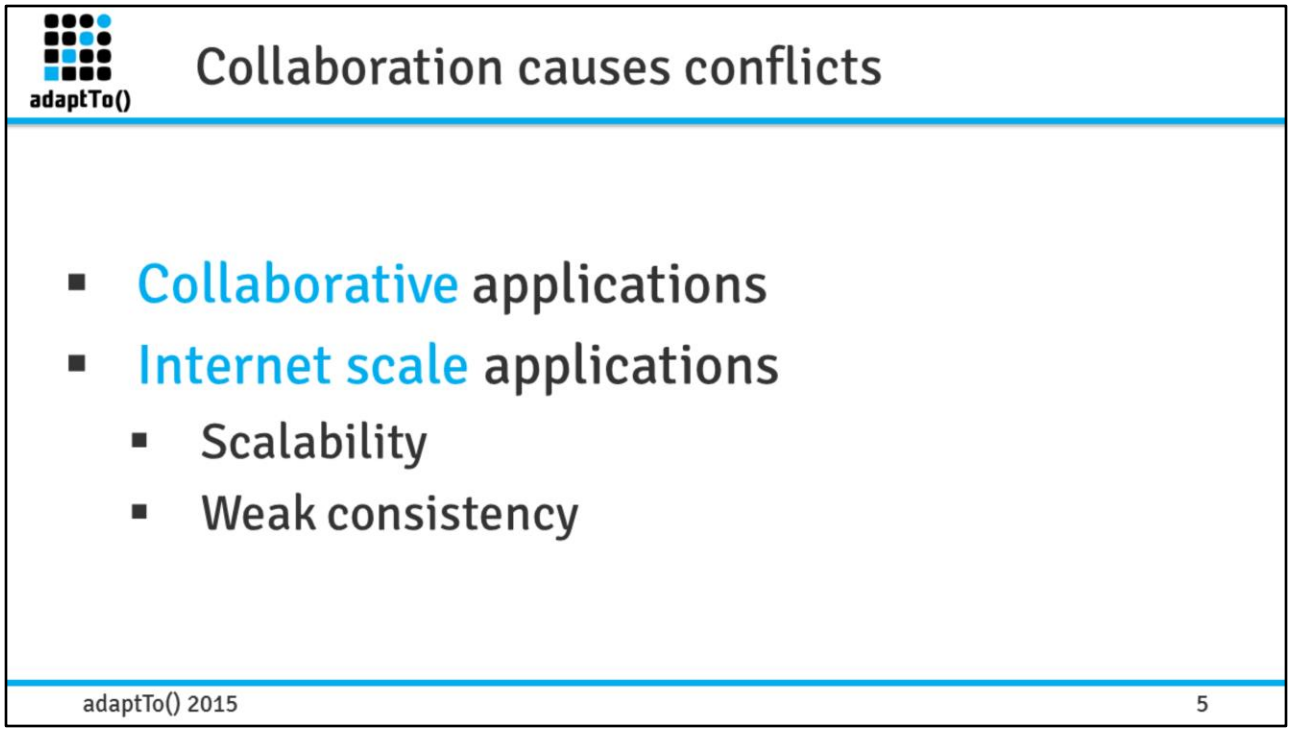
- Why should I care about conflicts? Why can't this be taken care of by the backend?
- What is a conflict? Specific to the application domain and also to back-end. Definition for Oak.
- How can we handle conflicts? What does Oak offer? Examples.



All you need to know about JCR and Oak (for this session):

- Oak is an implementation of the JCR (Java content repository) standard
- A JCR is a hierarchical database. Its content tree is made up of nodes and properties. Properties are the leaves of the tree and carry values.
- Oak heavily relies on commit hook plugins for providing its functionality. A commit hook can pass, edit or fail a commit.
- Custom commit hooks can provide further functionality.

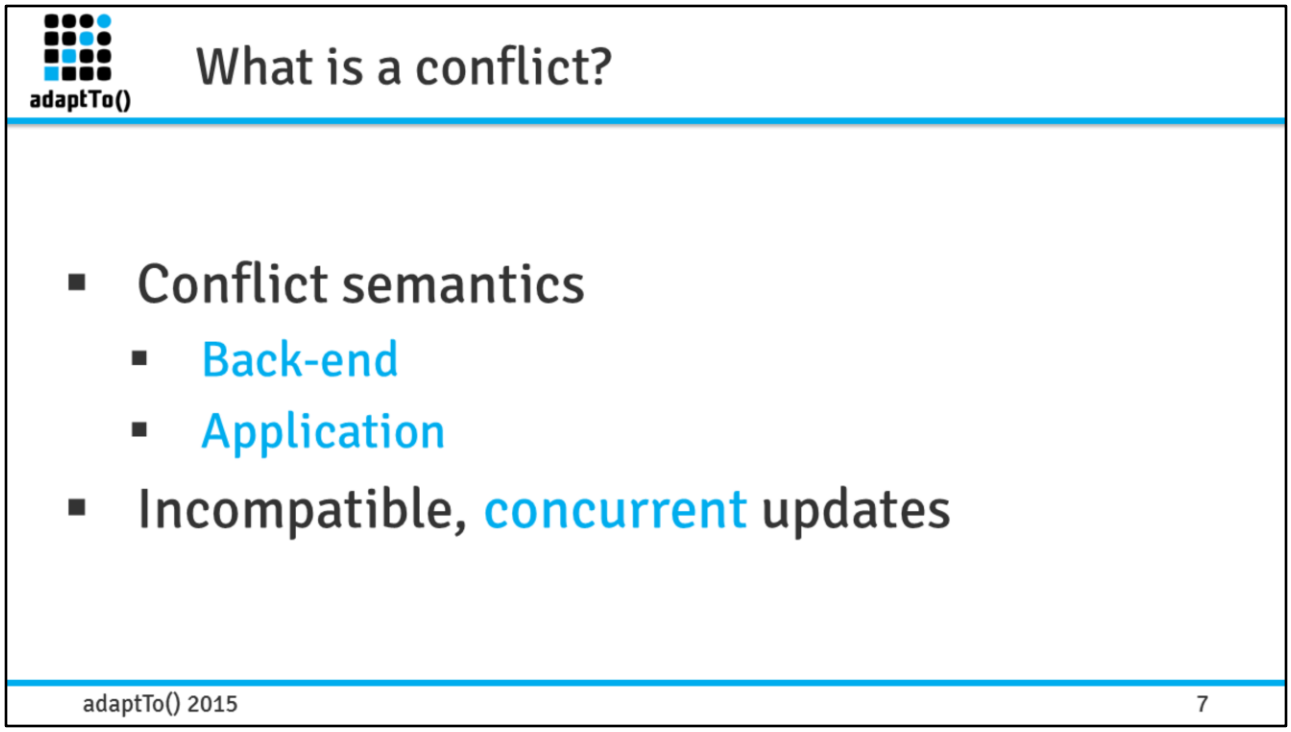
Why?



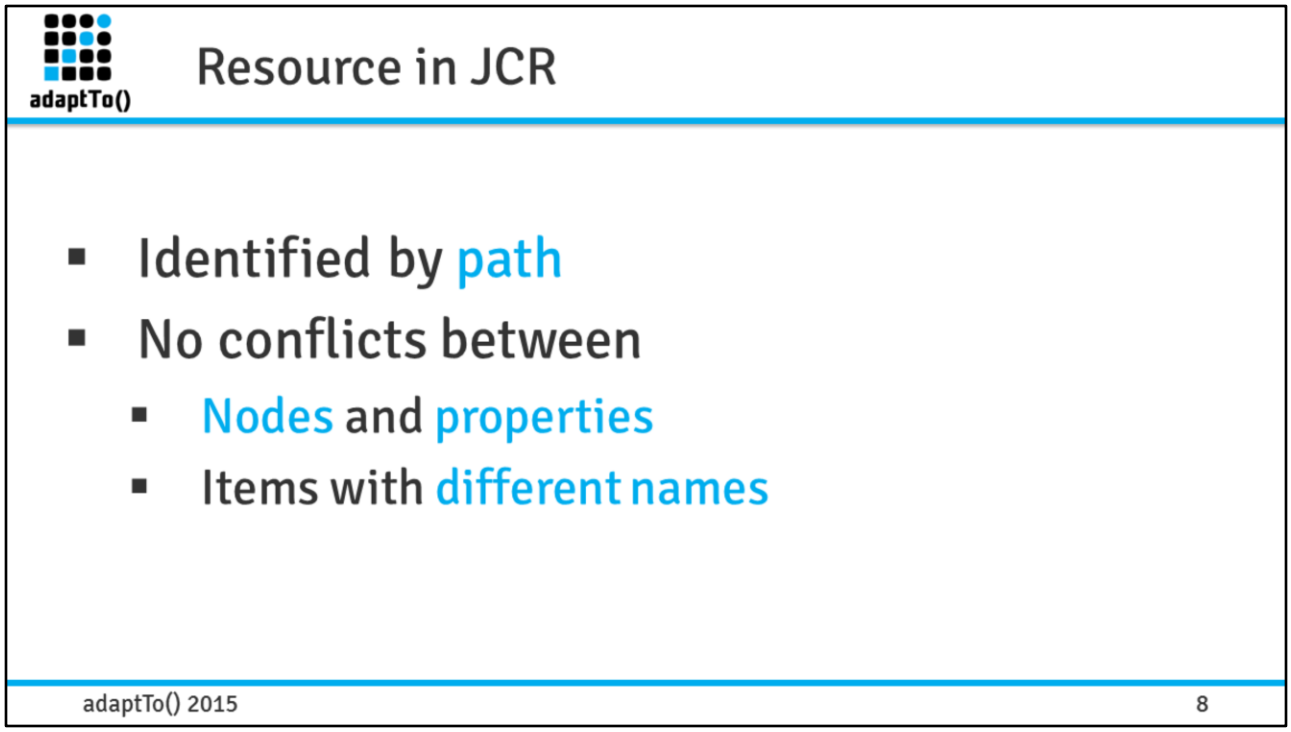
Conflicts appear in the context of collaboration. While this is evident for some applications like collaboratively editing the same spread sheet, it isn't for others. For example updating the like counts on forum posts is a not so evident form of collaboration. The necessity to rely on a weaker consistency model for such large scale applications force us to cope with conflicts where we traditionally would have relied on the backend.

Weaker consistency: giving up on ACID, aka NoSQL, trade consistency for throughput

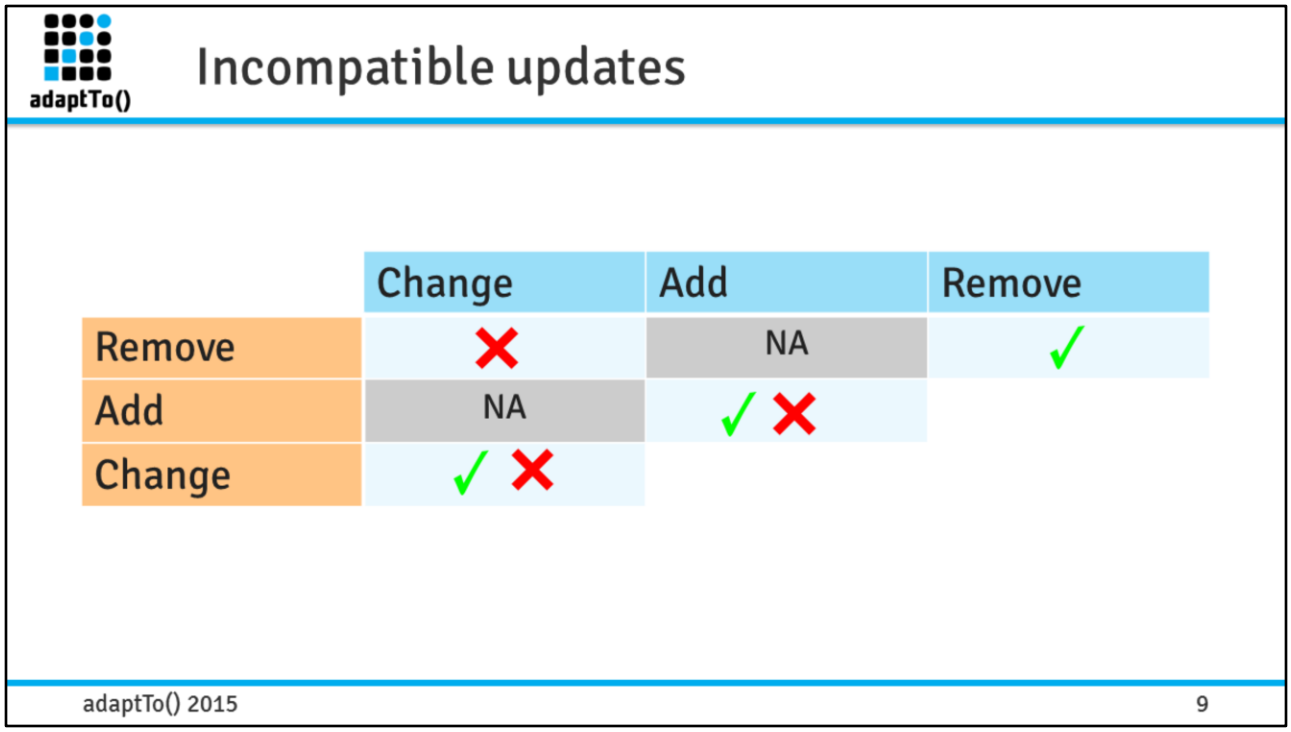
What?



A conflict is caused by multiple parties updating the same shared resource concurrently in incompatible ways. The exact meaning of incompatible is domain specific. However, there is usually a common least dominator specific to the back-end.
When building an application it is important to know the conflict semantics of the back-end and the conflict semantics of the application domain. This allows the application to be built in a way to avoid conflicts where possible and to choose the cheapest method for resolving them otherwise.



Updates on nodes never conflict with updates on properties. Also updates on items with different names never conflict. All conflict are either between nodes of the same name or between properties of the same name.



Updates of items of the same name and type conflict when (marked with a red cross)

- changing an item that has been concurrently removed
- adding an item that has been concurrently added with a different value
- changing an item that has been concurrently changed to a different value

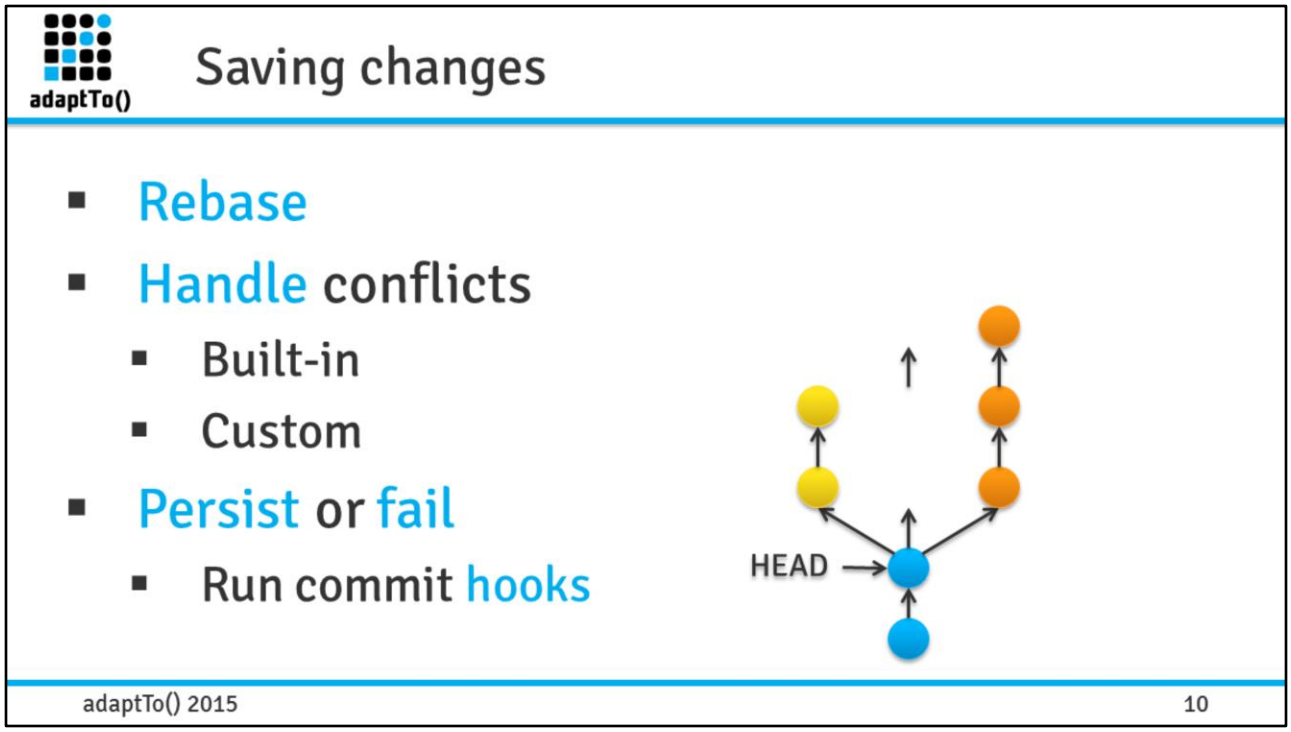
Such updates do not conflict when (marked with a green check)

- removing an item that has been concurrently removed

- adding an item that has been concurrently added with the same value
- changing an item that has been concurrently changed to the same value

The other combination are not applicable as they cannot occur: an item cannot be concurrently removed or changed before it has been added.

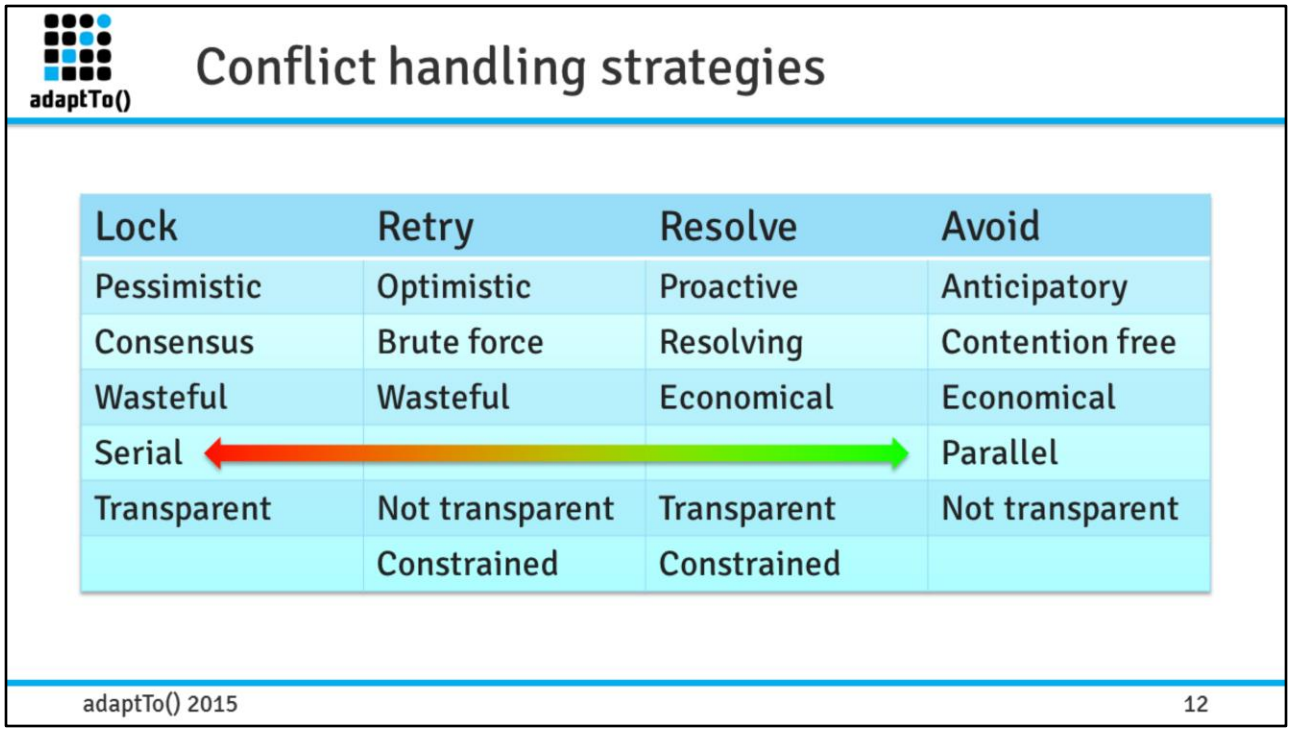
Oak's `ConflictHandler` interface has one method for each of these conflicts. Implementations are responsible to resolve these conflict according to the applications need.



When saving changes Oak rebases them on top of the latest trunk resolving conflicts along the way. On success the rebased changes are persisted. Otherwise saving fails with an exception.
Rebasing is done by calculating the difference between the current state of the sessions against its base state and applying them on top of the latest trunk. This is the point where actual conflicts are detected and resolved according to the mechanism shown on the previous slide. Additionally this the point where Oak allows injection of custom conflict

handlers.

How?

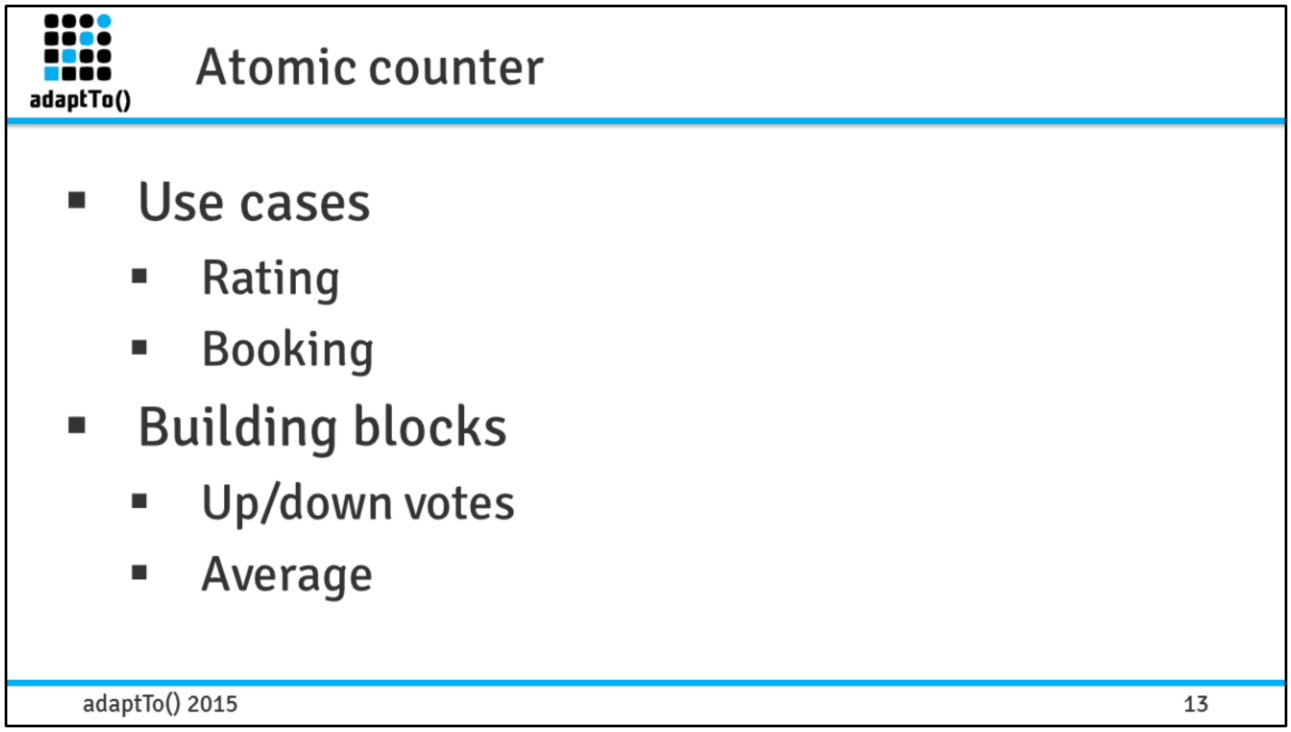


Strategies for handling conflicts in order of decreasing cost:

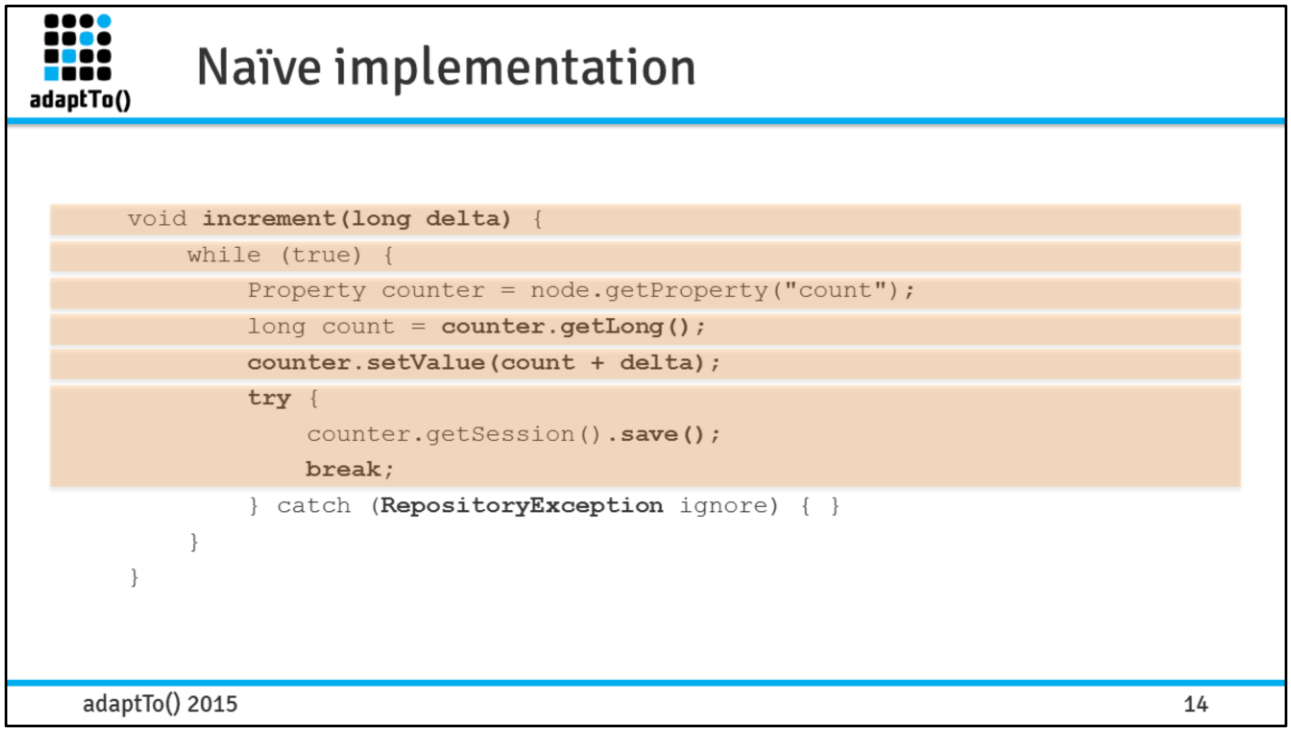
- Lock
  - Well known, pessimistic approach effectively serialising access to the back-end thus leaving most CPU cores idle.
  - Requires upfront global consensus, which can be expensive to acquire.
  - Transparent to the application.
- Retry
  - Optimistic brute force approach relying on conflicts being rare.

    - Conflicts are detected after the fact and commits are retried by the application, which wastes CPU cycles.
    - Not transparent to the application due to the retry logic necessary.
    - Constrained by the conflict semantics of the back-end being a subset of those of the application. Otherwise some conflicts will not be detected.
- Resolve
    - Proactive approach relying on conflicts being rare.
    - Conflicts are detected after the fact and resolved as part of the commit process through custom commit hooks.
    - Transparent to the application.
    - Constrained by the conflict semantics of the back-end being a subset of those of the application. Otherwise some conflicts will not be detected.
- Avoid
    - Leverage conflict semantics of the application to avoid conflicts.
    - Fully parallel, no point of contention.
    - Not transparent to the application as it

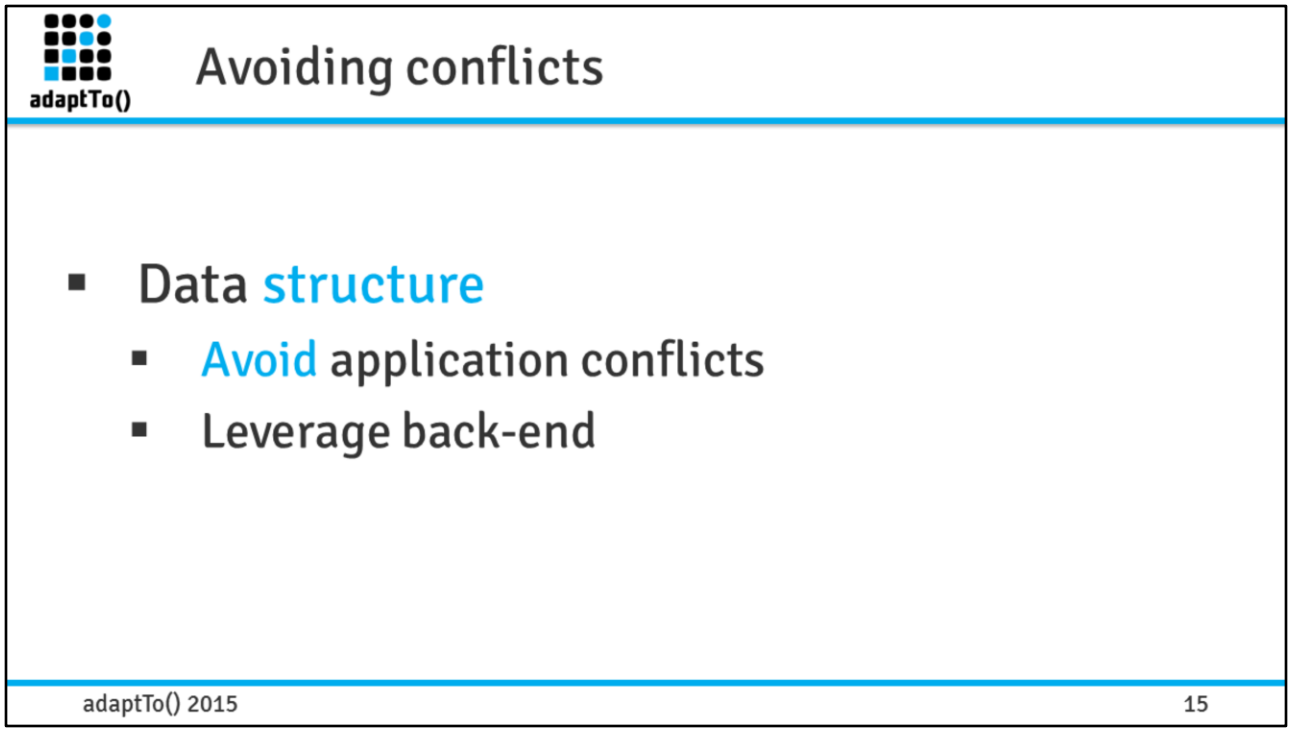
needs a conflict aware data model



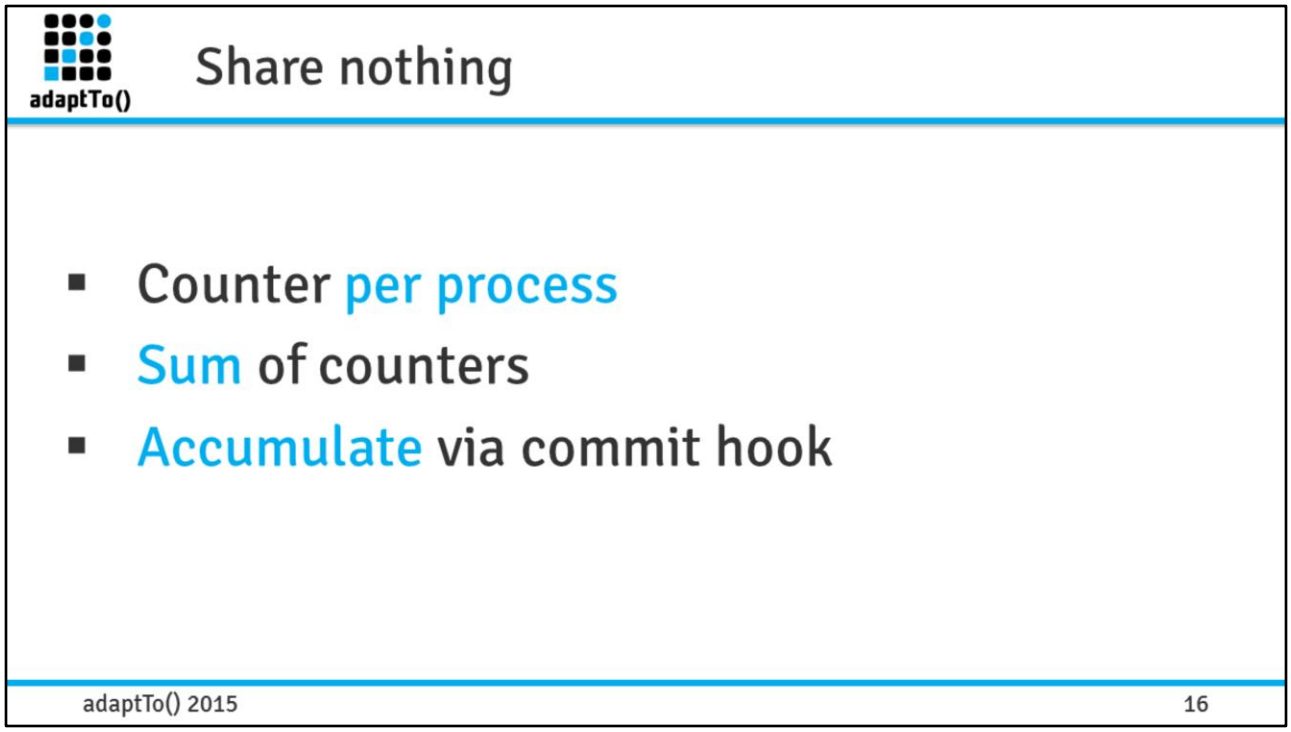
Atomic counters are useful to implement e.g. rating functionality for blog comments. More complex behaviour (e.g. averages) can be implemented by combining multiple counters. A naïve counter implementation on Oak is prone to data races though as in some cases no conflict is detected on back-end.



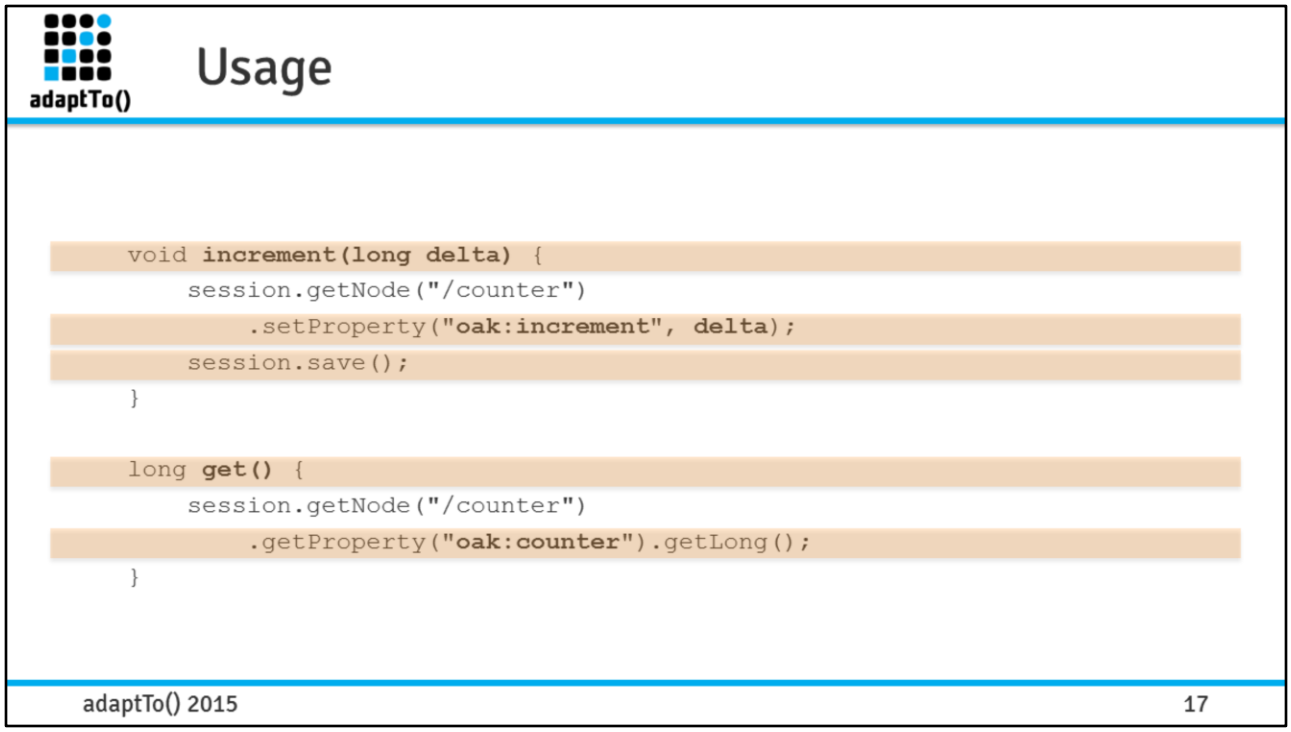
This naïve implementation of a counter retries to update its value until it succeeds. Apart from the brute force approach, the implementation has a data race: two concurrent increments by the same value will result in the counter only being updated once.



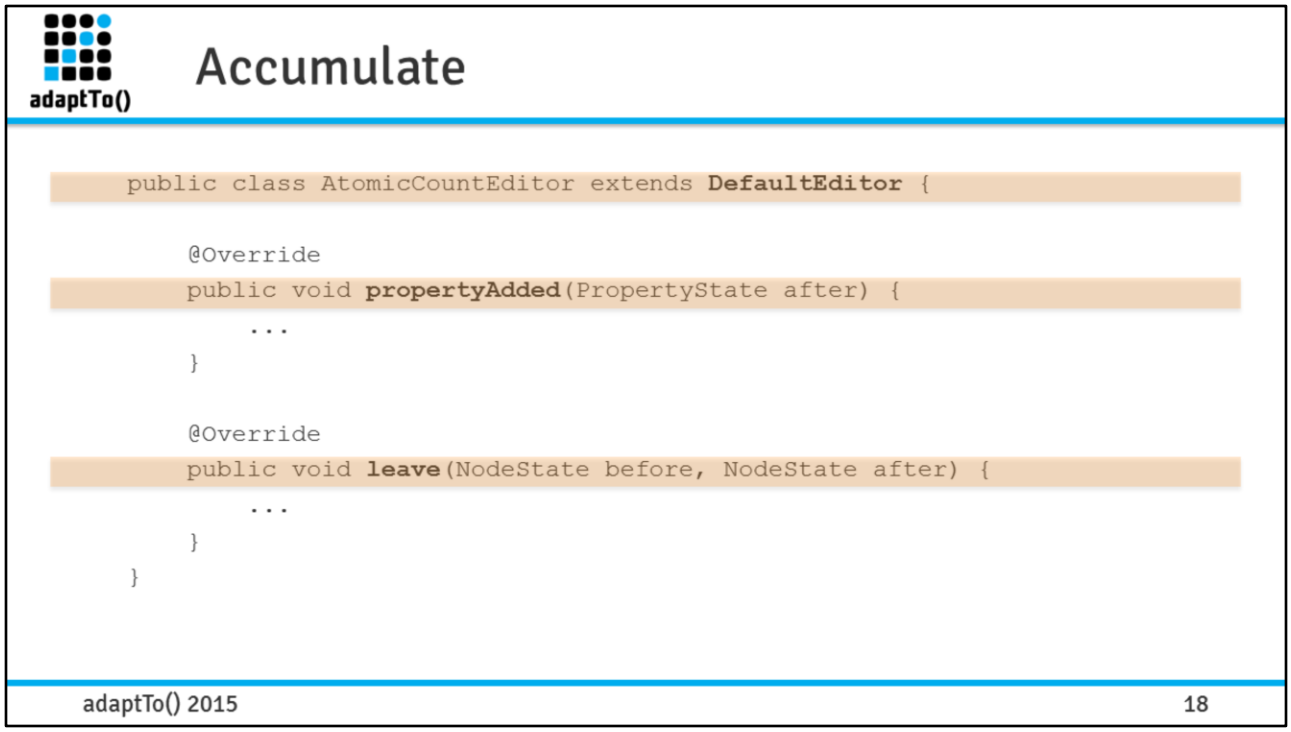
Leverage conflict semantics of the back-end to resolve application specific conflicts. This requires applications to choose their data model accordingly.



A better approach uses a private counter per involved process, avoiding conflicts altogether. The sum of all private counters is the total sum of the counter. In Oak we can take this further and use a custom commit hook that will take care of accumulating the individual counters into one global value. Also as Oak implements has strong session isolation we don't need to worry about name clashes of the involved counters.
Accumulation allows for easy indexing/sorting/querying of counters values, which would otherwise not be possible.



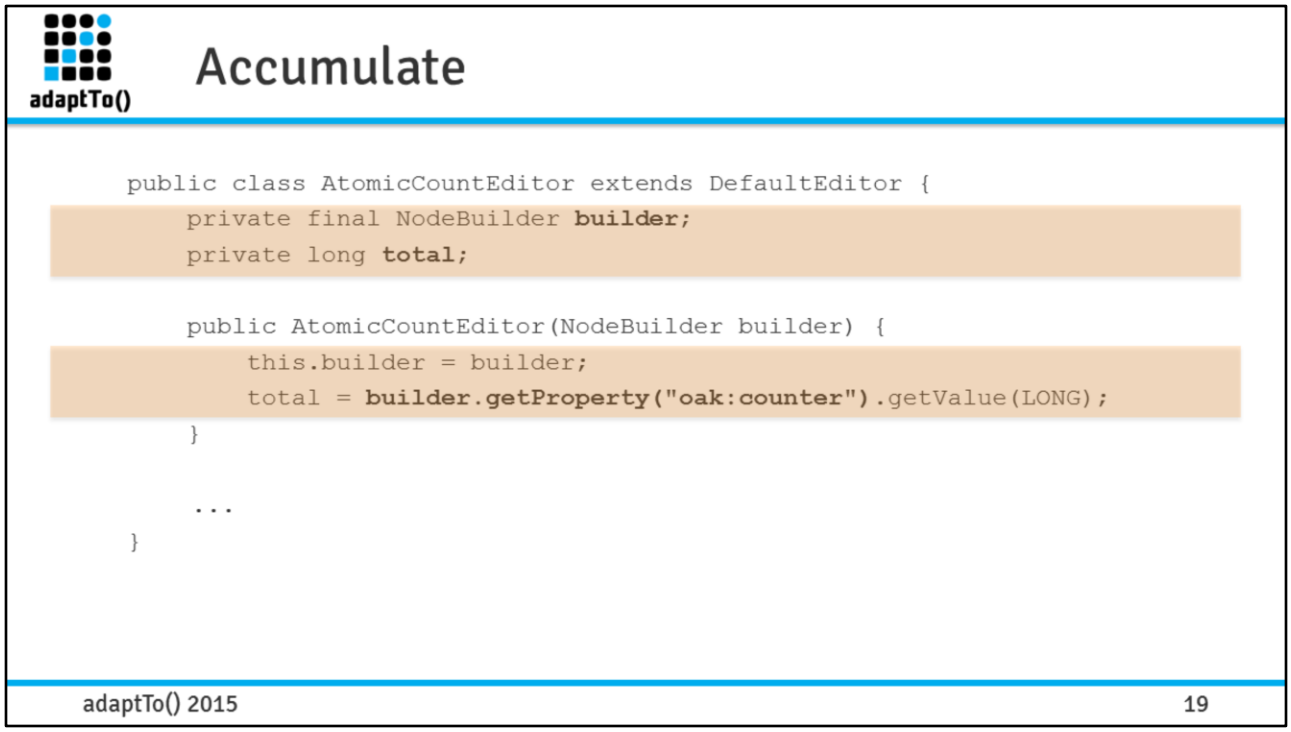
The retry logic becomes unnecessary when each process has its own counter: instead of counting itself we set the desired increment and let the accumulation logic in the commit hook deal with updating the counter.



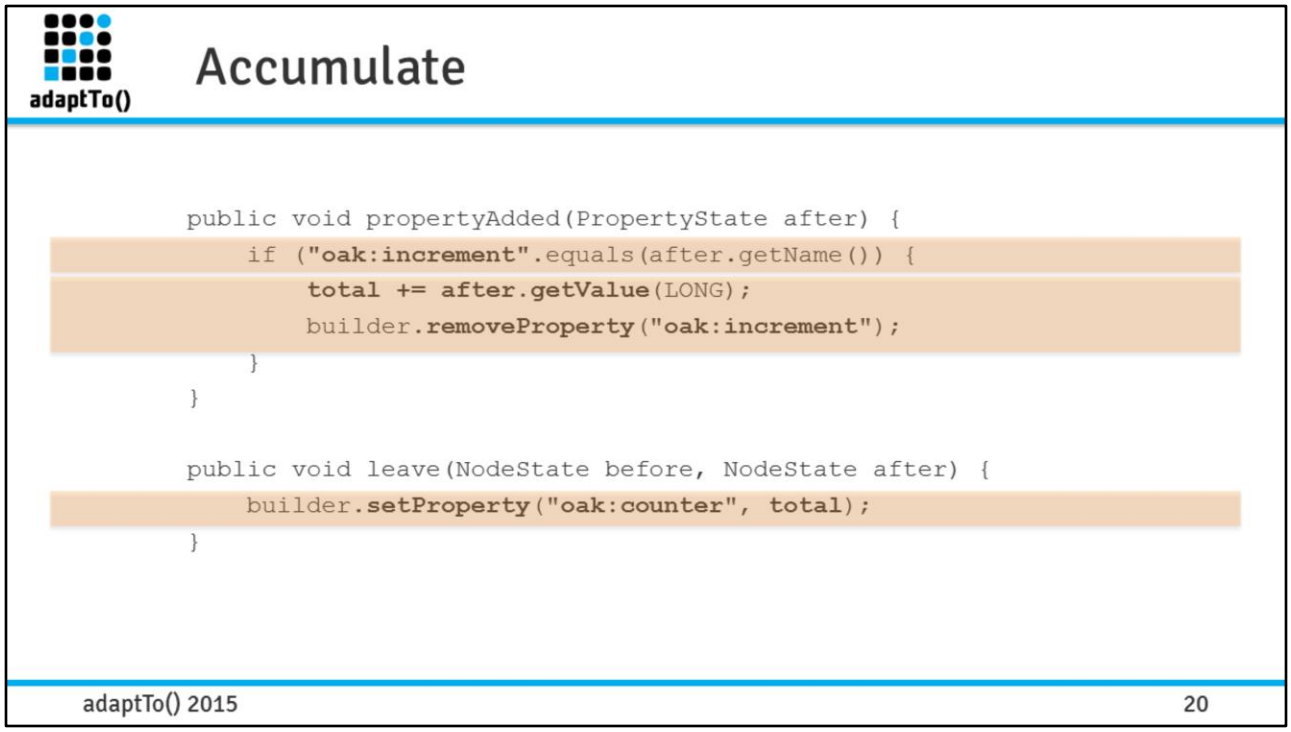
To accumulate the individual counter values into on consistent counter value we need to install a custom commit hook into Oak. A commit hook is called once changes are committed and can arbitrarily modify what is being committed.

In our case we need to detect additions of properties named `oak:increment` and add their values to the value of the `oak:counter` property. Subsequently we can discard the `oak:increment` property as it is not needed anymore and doesn't need to be persisted. We implement the commit hook by extending from

`DefaultEditor`. We need to override the `propertyAdded` method, which is called whenever the commit contains a newly added property and the `leave` method, which is called once this commit hook is done.



On initialisation we retrieve the current counter value from the `oak:counter` property. In addition the constructor receives a `NodeBuilder` instance to record out changes to the commit.

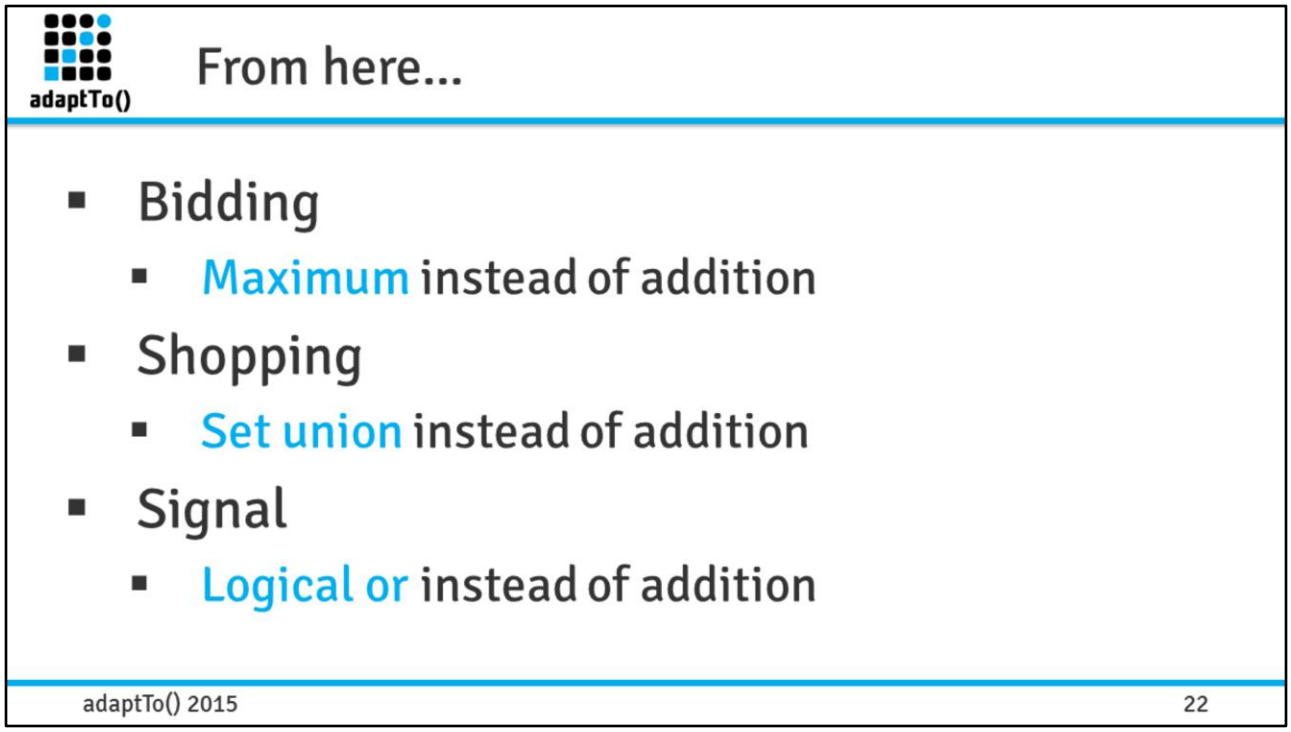


The `propertyAdded` method updates the total value by adding this counters value and subsequently discarding the transient `oak:increment` property. When done the `oak:counter` value will be assigned back to the respective property in the `leave` method.

adaptTo()

Demo

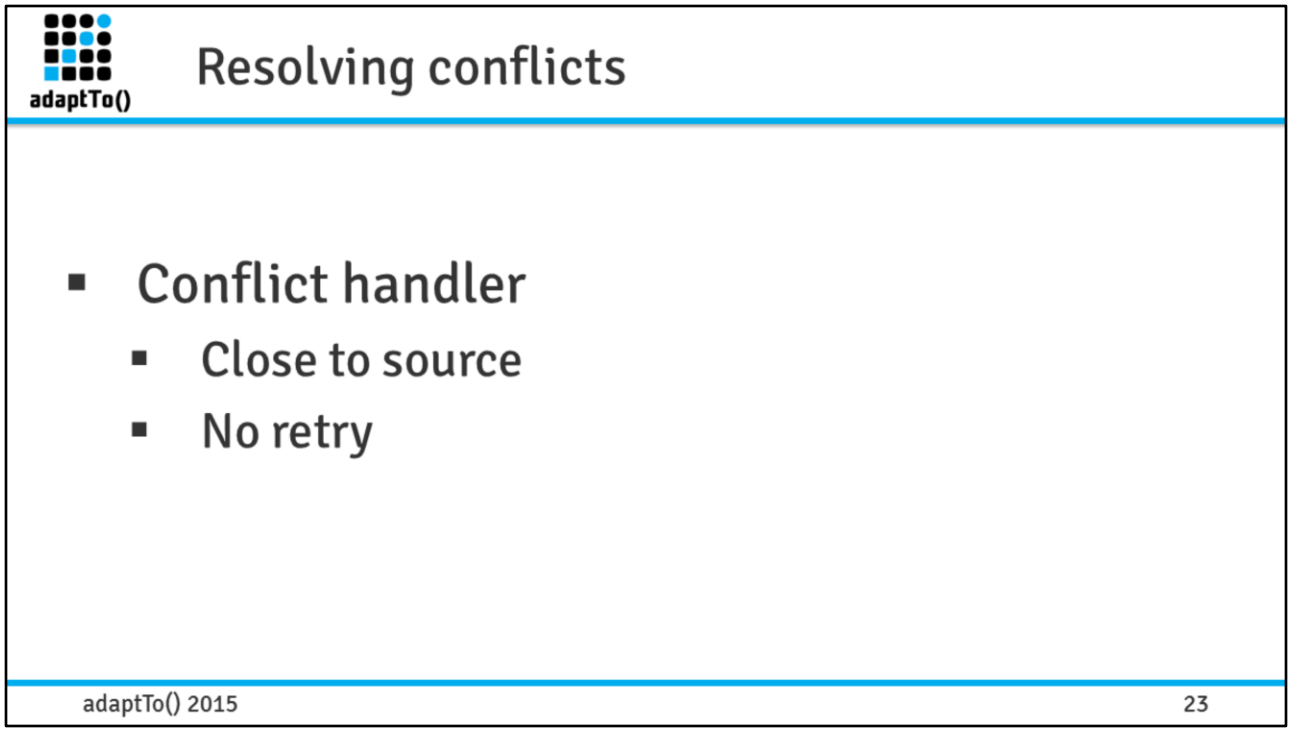
https://www.flickr.com/photos/ruiwen/3260095534/



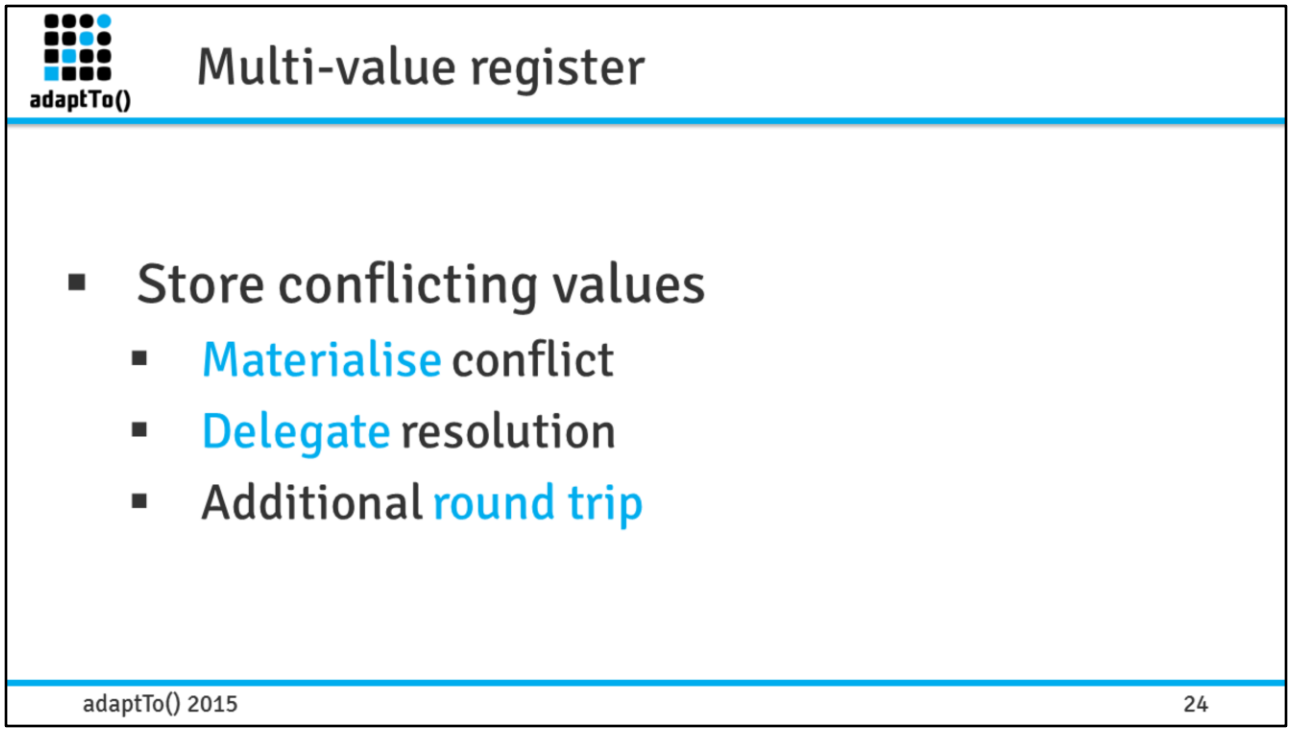
The concept is easily adapted to other uses cases. For bidding in an auction we would replace addition with maximum in the accumulation step. This ensures eventually awarding the highest bidder. An e-commerce application would use set union instead of addition to add items to a trolley. For signalling a certain condition with a shared boolean flag, we would use logical or instead of addition.

Most generally this concept applies whenever the elements under consideration together with the accumulation function form a semilattice.

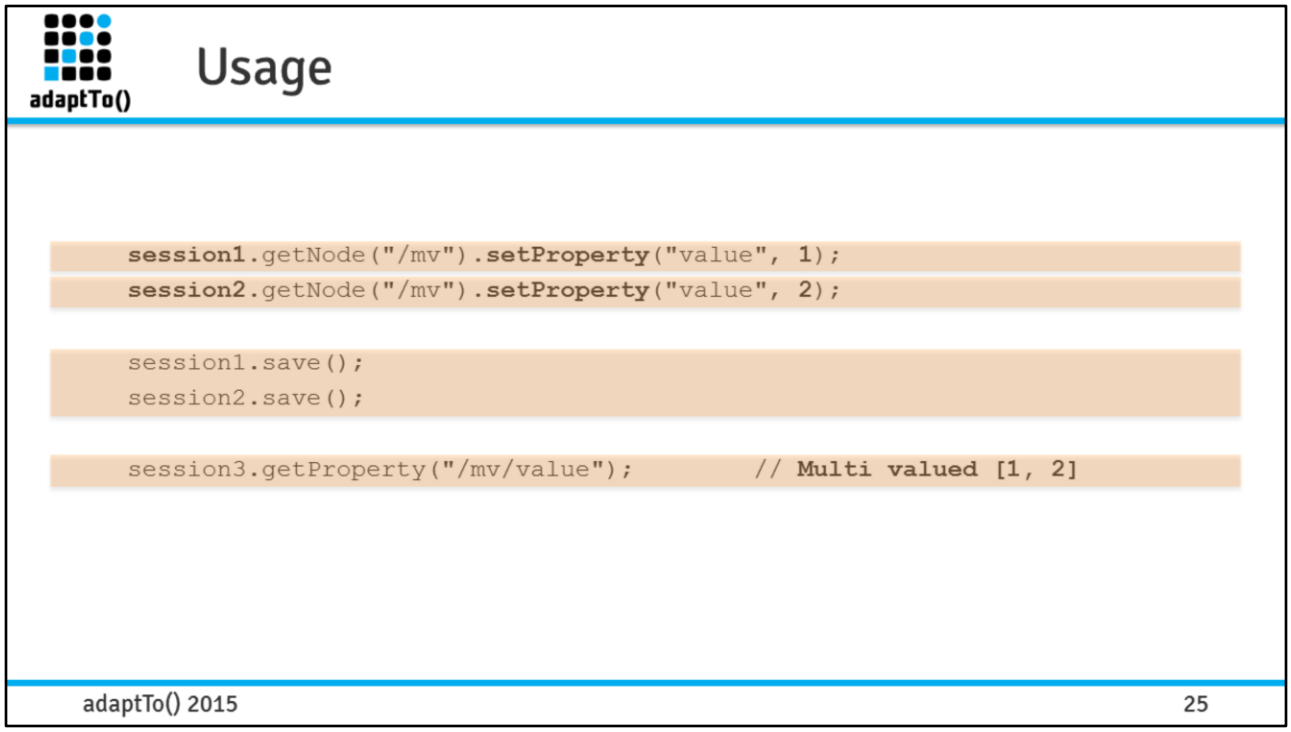
Put simply: for any two elements we know how to accumulate them.



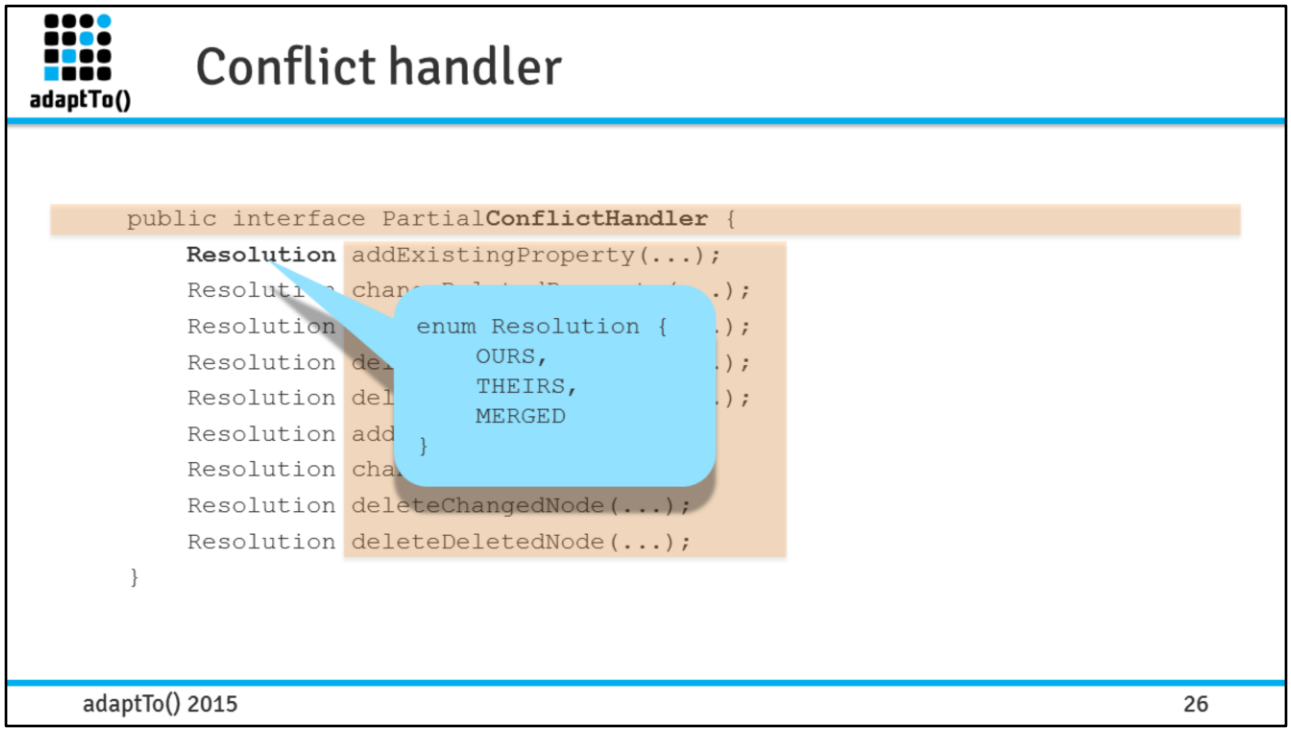
Resolve conflicts as they occur instead of retrying a failed commit. Requires injection of a custom conflict handler. Oak supports this through implementations of the `ConflictHandler` and related interfaces.



A multi-value register is much like a register for a single value unless that in the case of a conflict it will store all conflicting values. Multi-value registers provide a simple way for applications to detect a conflict and resolve it after the fact. Such a conflict resolution scheme adds additional round trips in the presence of conflicts though.



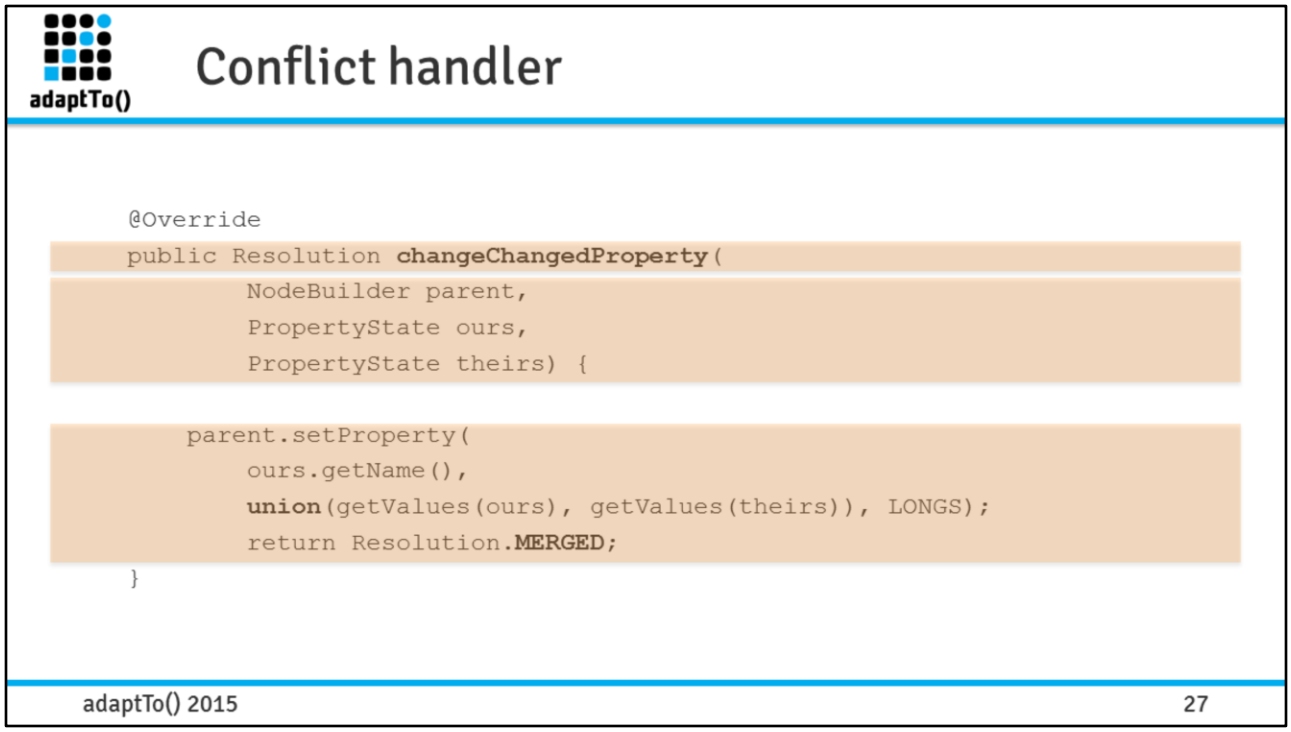
Updating the same property from two different sessions concurrently with incompatibly values should not fail (as it usually does). Rather should it create the property as a multi-valued storing both conflicting values.
Later updates can overwrite that value again.



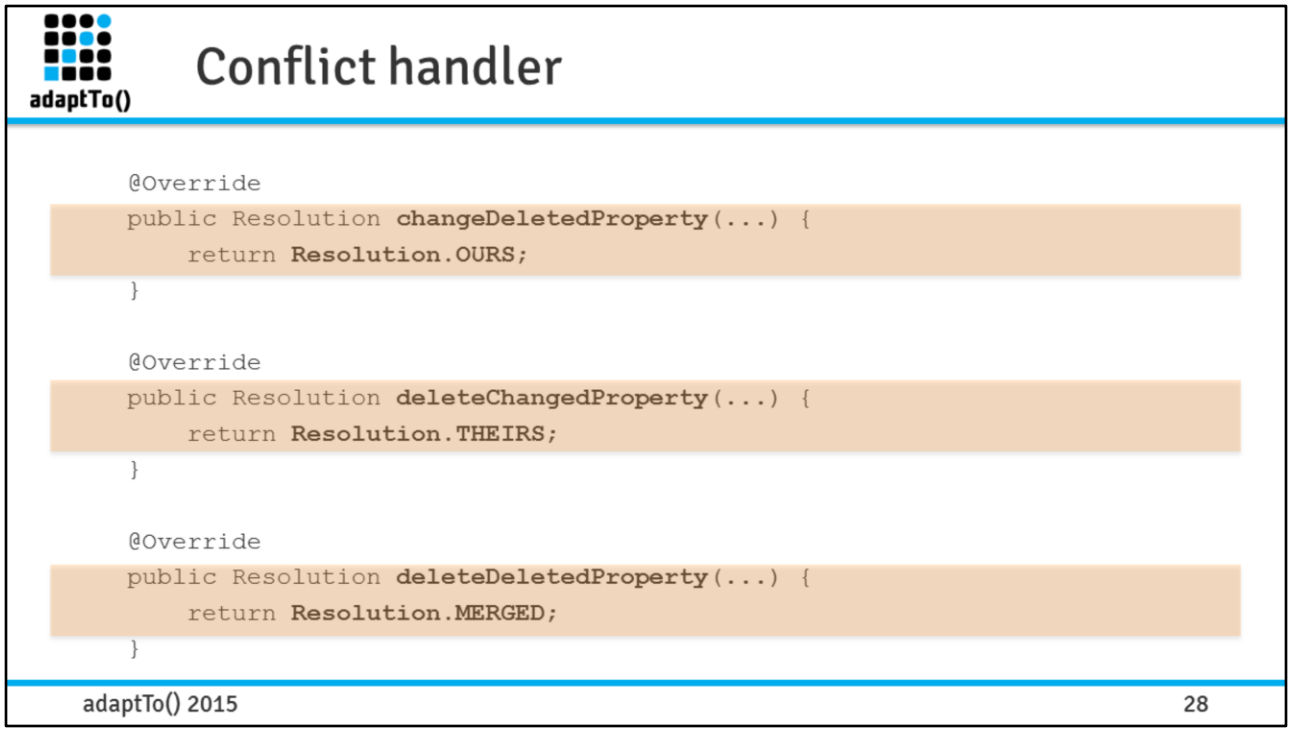
Custom conflict handlers can be injected into Oak to resolve conflicts from concurrent updates. A conflict handler needs to implement a method for every type of conflict that can occur. The arguments to those methods provide access to all values of the conflicting parties. It has to come up with a resolution by either choosing one of the values (`OURS` or `THEIRS`) or by implementing some custom merge algorithm (`MERGED`).

In addition a partial conflict handler can also choose not to cope with a conflict and result `null` instead. This means another partial

conflict handler further up the chain can take care of the conflict. Oak composes all partial conflict handlers into a (total) conflict handler. This is done by chaining the partial conflict handlers together and adding a default handler at the end of the chain, which will just cause a commit to fail if it detects an unresolved conflict.



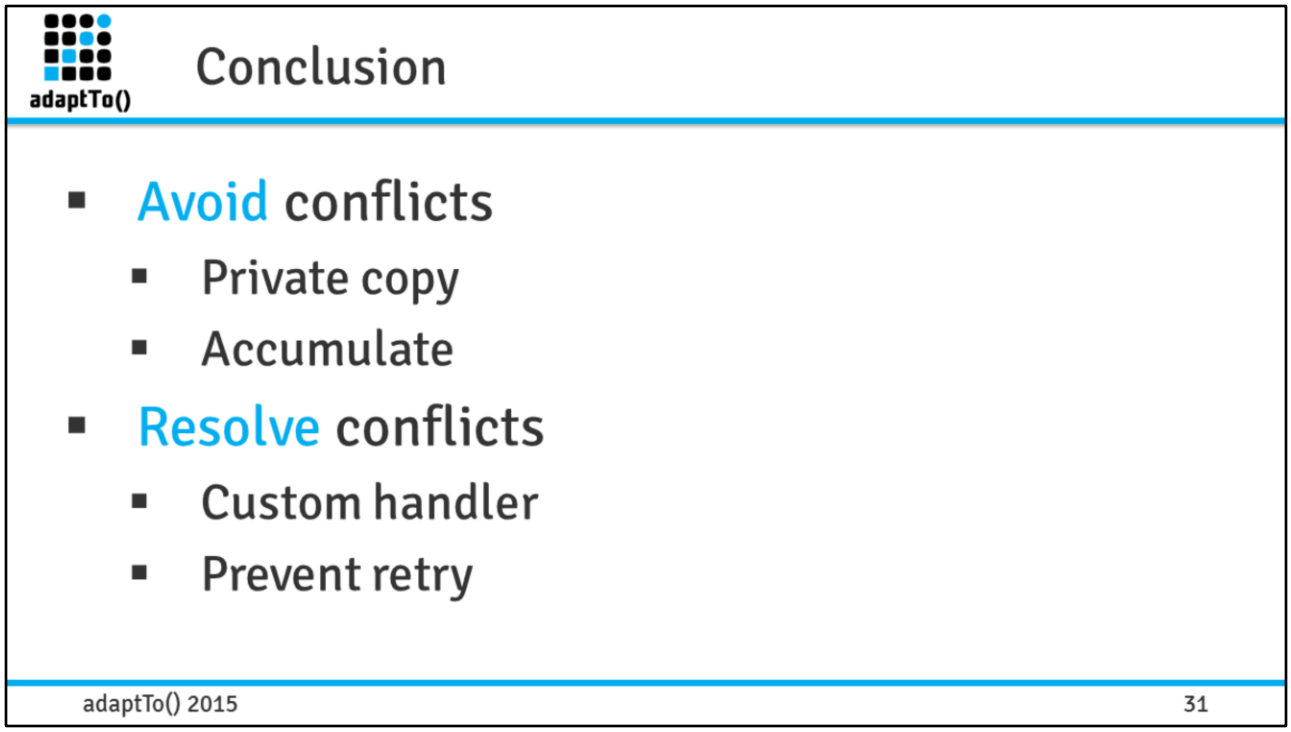
The implementations of `addExistingProperty` and `changeChangedProperty` read the values of both parties, merge them into one multi-valued property and use that value to resolve the conflict.



The remaining cases can simply resolve to `OURS`, `THEIRS` and `MERGED`, respectively.

Demo

Conclusion



Using carefully crafted data structures it is often possible to avoid conflicts altogether. The general pattern here is to provide a private copy to every process involved and accumulate the values later on. Oak supports accumulation through custom commit hooks. This way of avoiding conflict is loosely based on “Convergent and Commutative Replicated Data Types” [1]. The reason for the relative simplicity of our implementations is Oak’s rather strong consistency model (Oak is sequentially consistency, which is only slightly weaker than linearizable). The general

approach discussed in [1] is based on a much weaker “eventual consistent” storage model.

Alternatively Oak also provides hooks to resolve conflicts as they occur. This allows conflicts to be handled as close as possible to their source preventing the need to retry failed commits.

[1] http://hal.upmc.fr/file/index/docid/555588/filename/techreport.pdf

Thank you

- https://github.com/mduerig/oak-crdt