Oak, the Architecture of the new Repository

Michael Dürig, Adobe Research Switzerland

APACHECON EUROPE

CORINTHIA HOTEL
BUDAPEST, HUNGARY
NOVEMBER 17-21, 2014

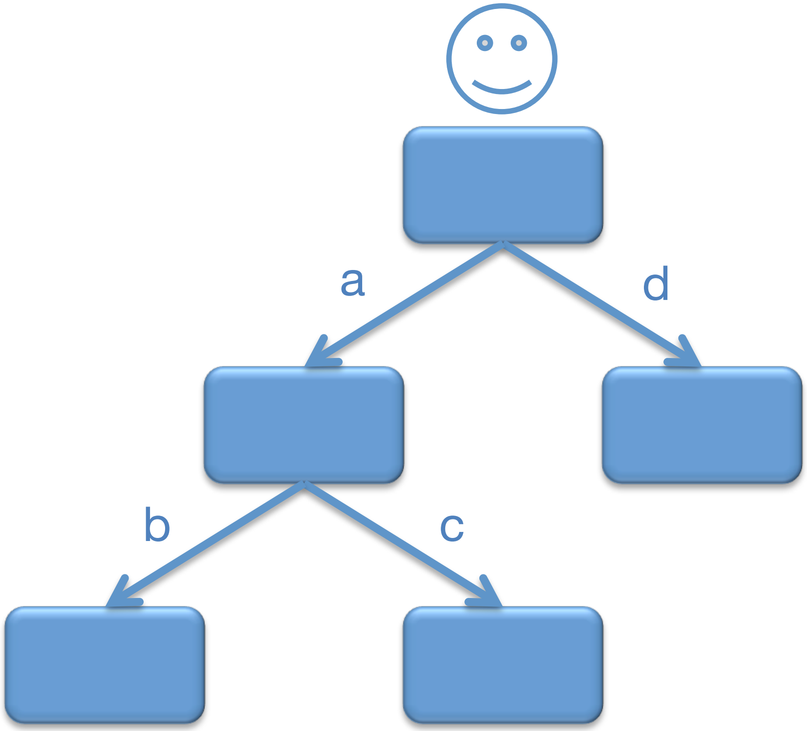
Design goals

- Scalable
- Big repositories
- Clustering
- Customisable, flexible
- OSGi friendly

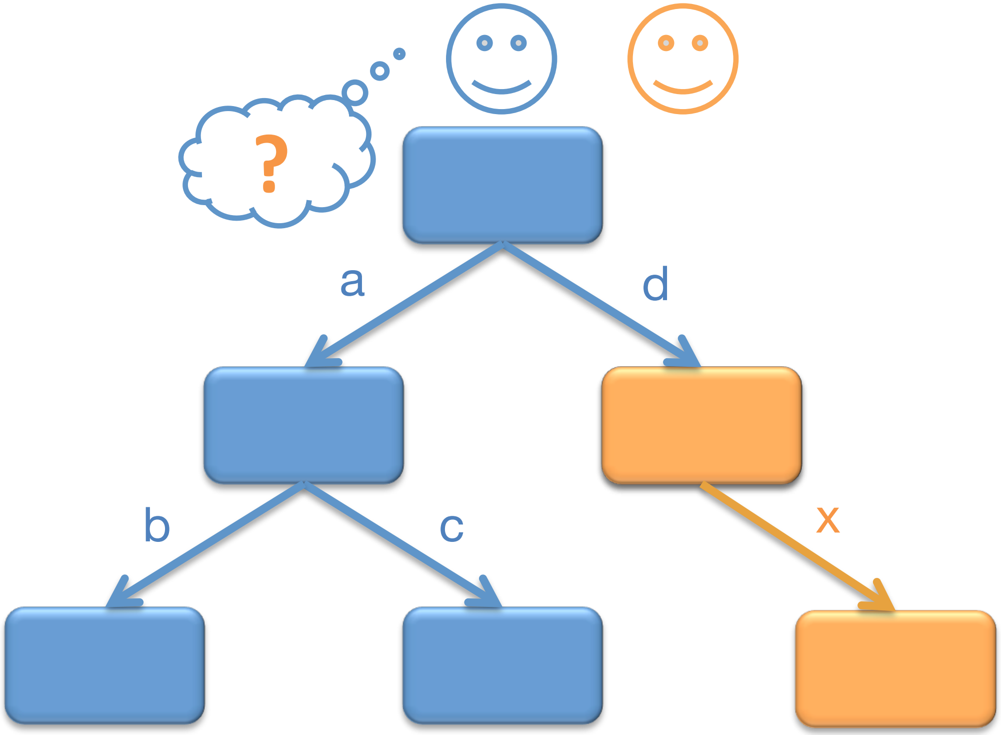
Outline

- CRUD
- Changes
- Search

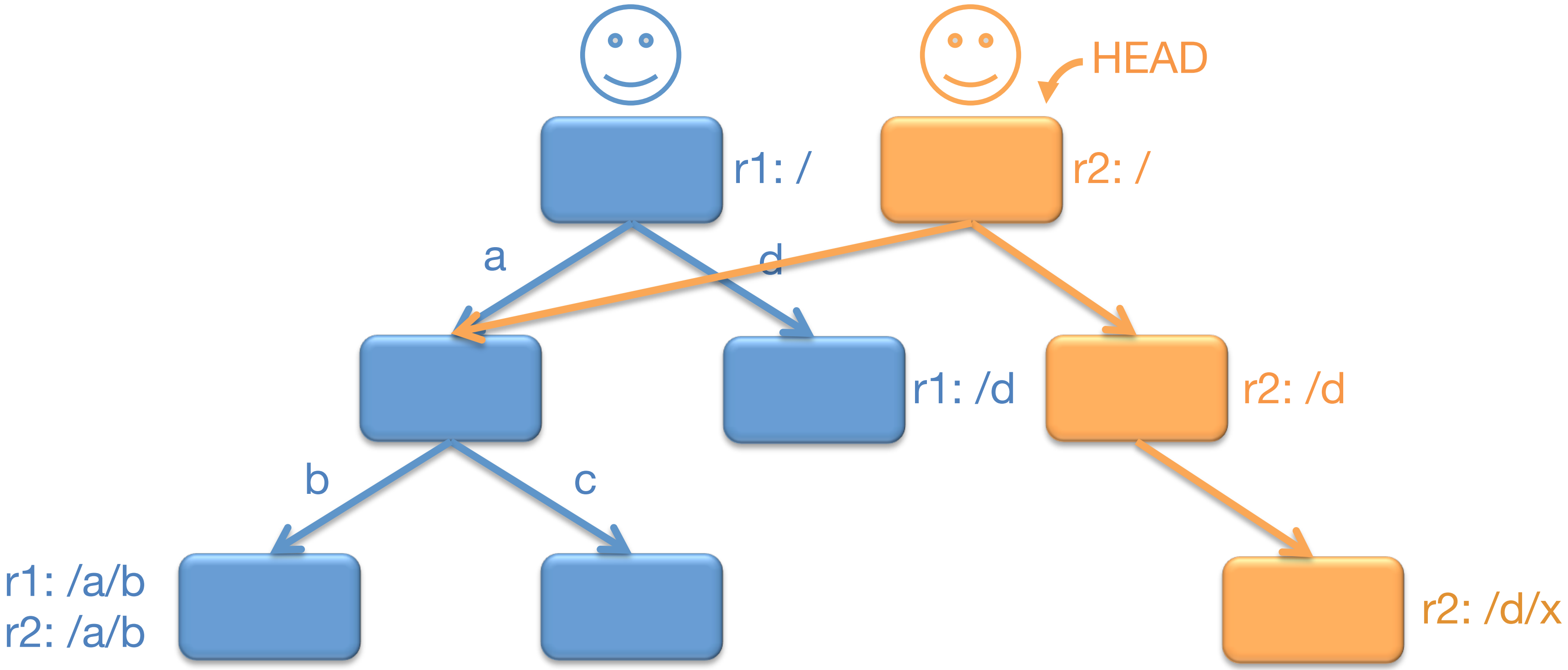
Tree model



Updating

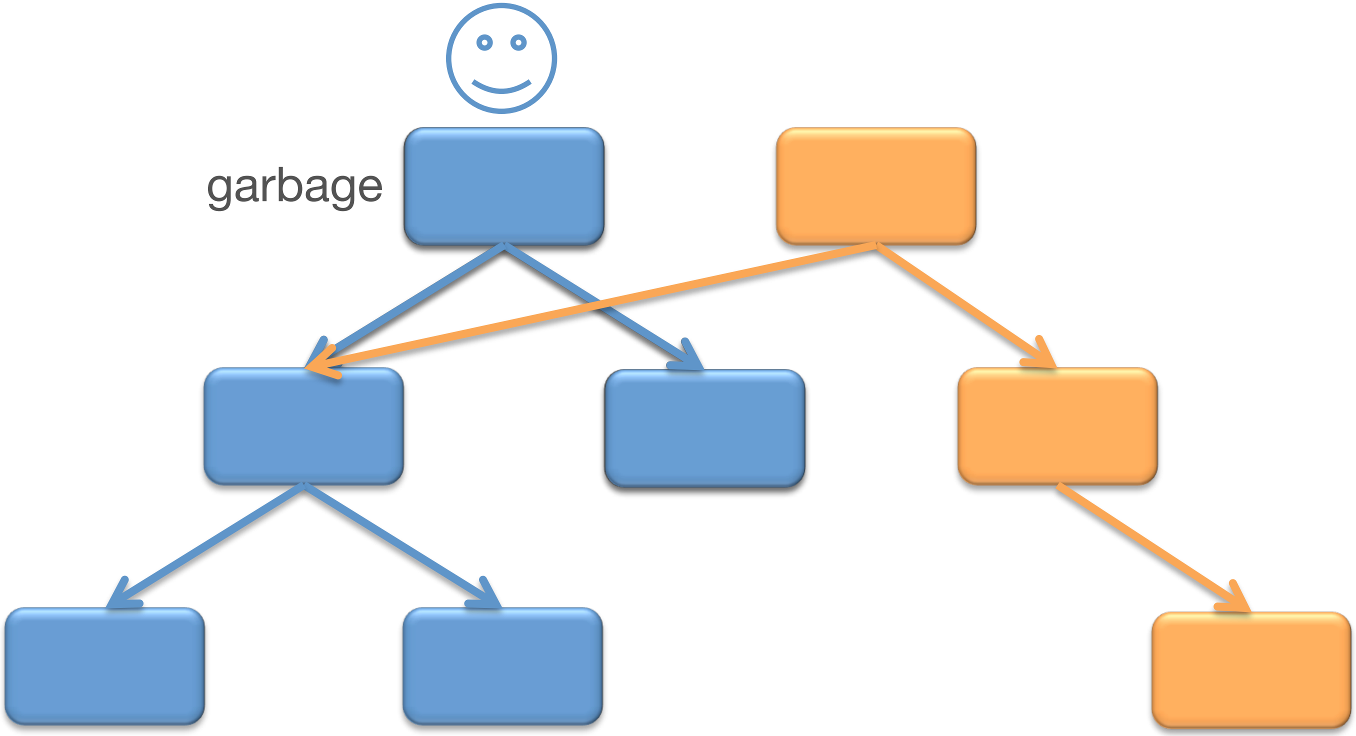


MVCC

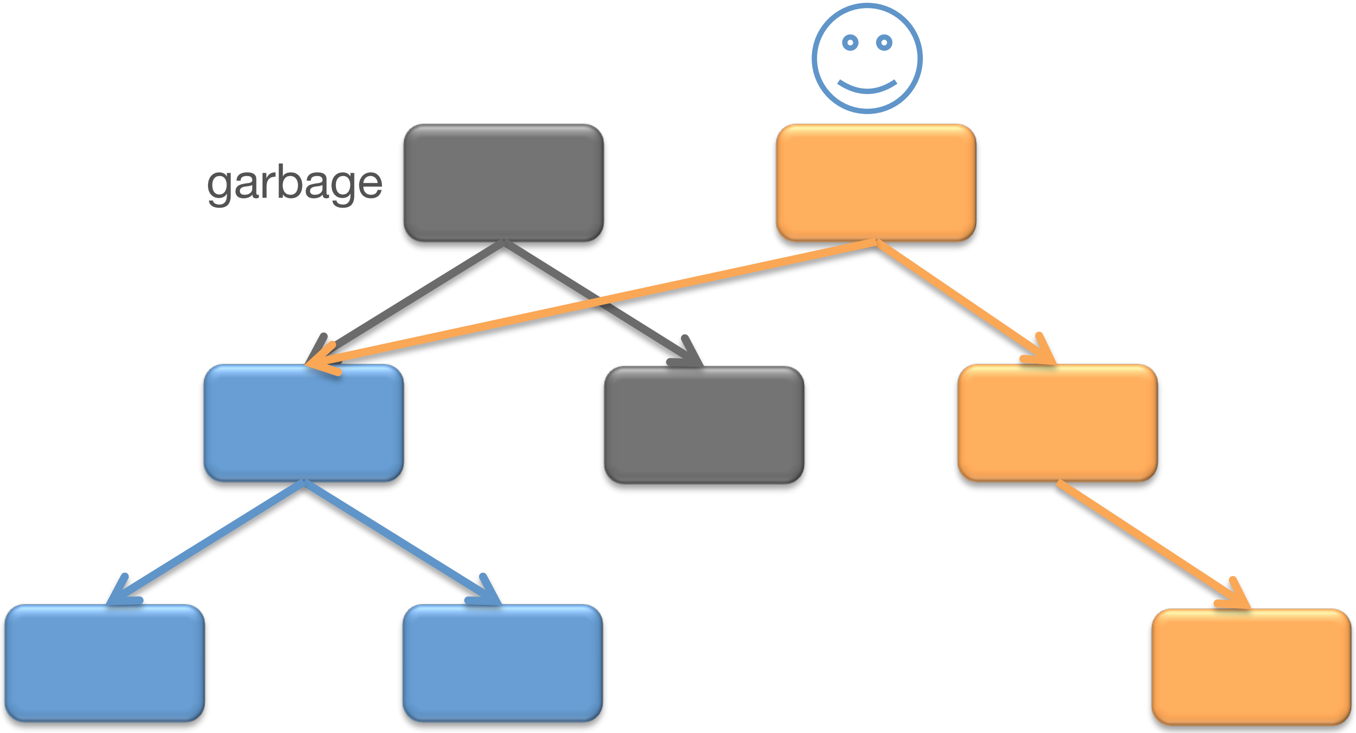


Refresh and Garbage Collection

Refresh

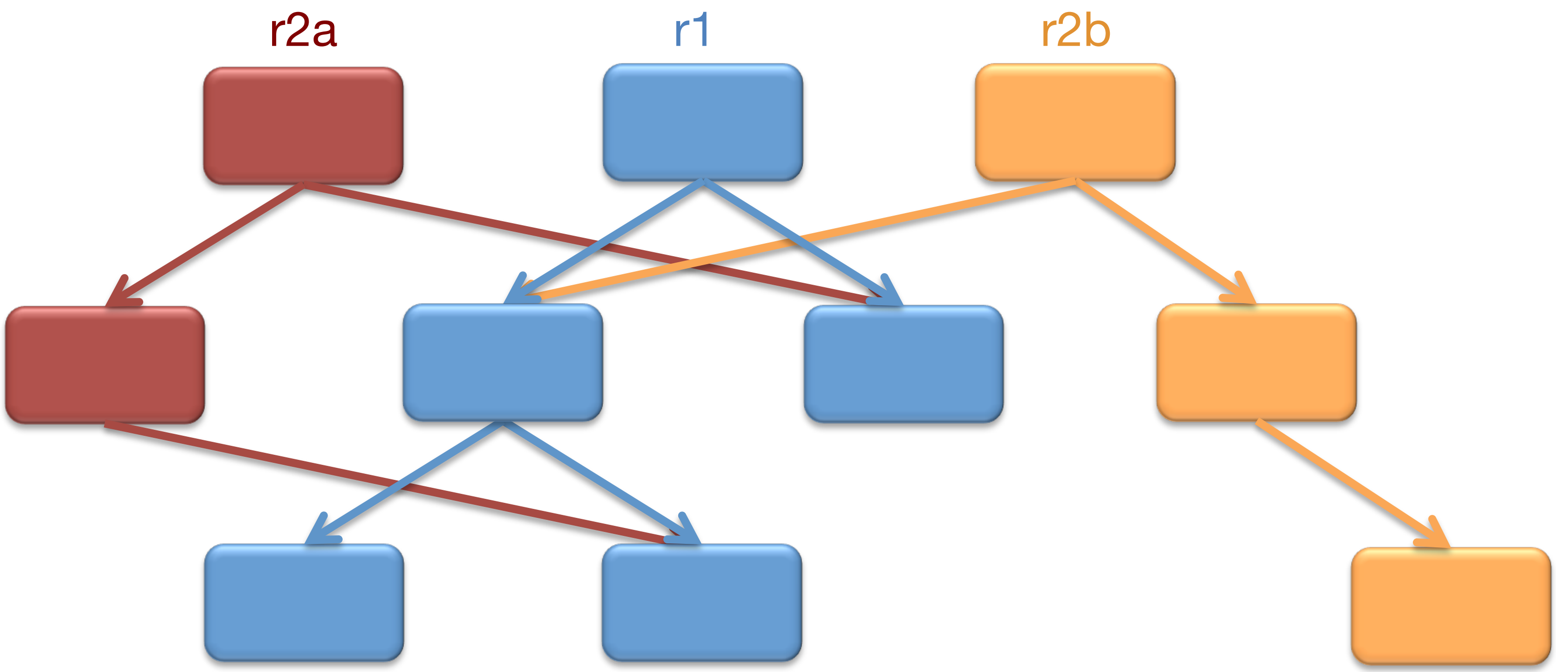


Garbage collection

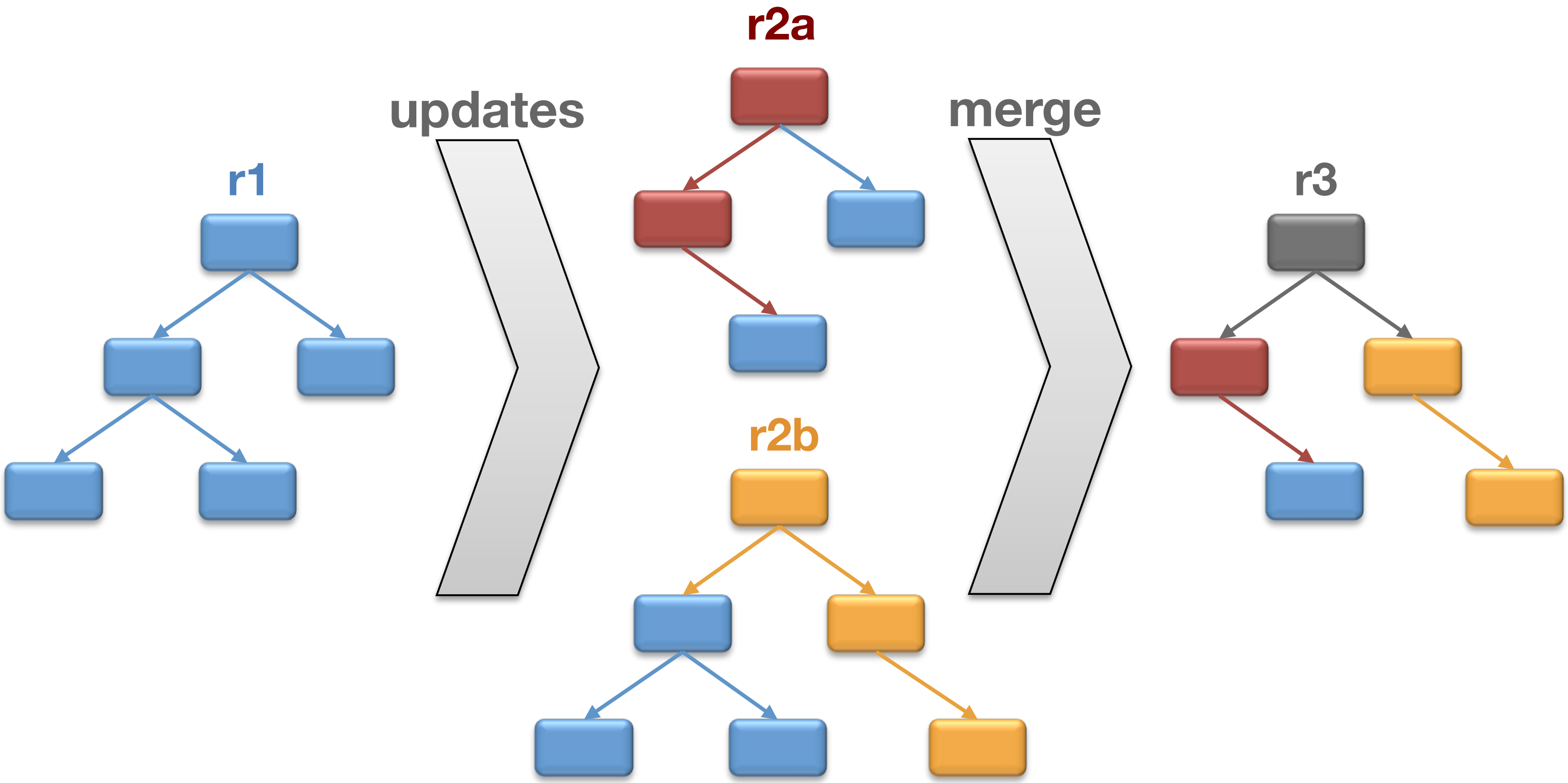


Concurrency and Conflicts

Concurrent updates



Merging



Conflict handing: serialisation

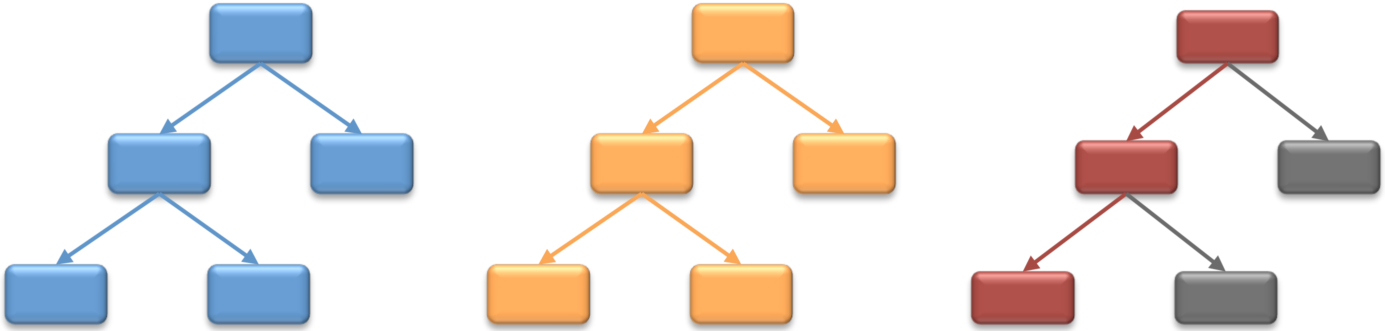
- Fully serialised
  - – Fail, no concurrent update
- Partially serialised
  - – Concurrent conflict free updates

Conflict handling strategies: merging

- Partial merge
  - Conflict markers, deferred resolution
- Full merge
  - Need to choose victim

Replicas and Shards

Replica and caches

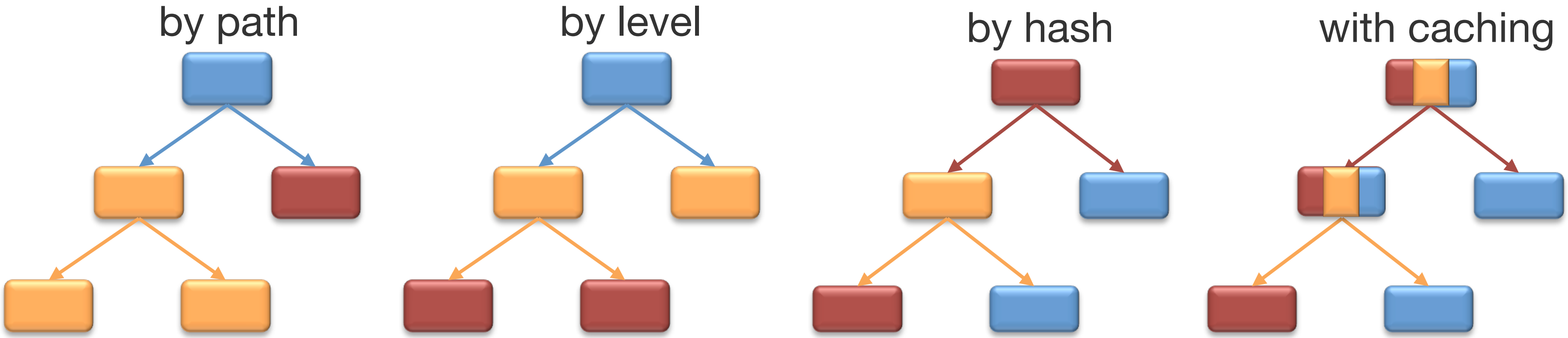


master copy

full replica

cache

Sharding strategies



by path

by level

by hash

with caching

Implementations

MicroKernel / NodeStore

- Tree / Revision model implementation

Responsible for
Clustering
Sharding
Caching
Conflict handling

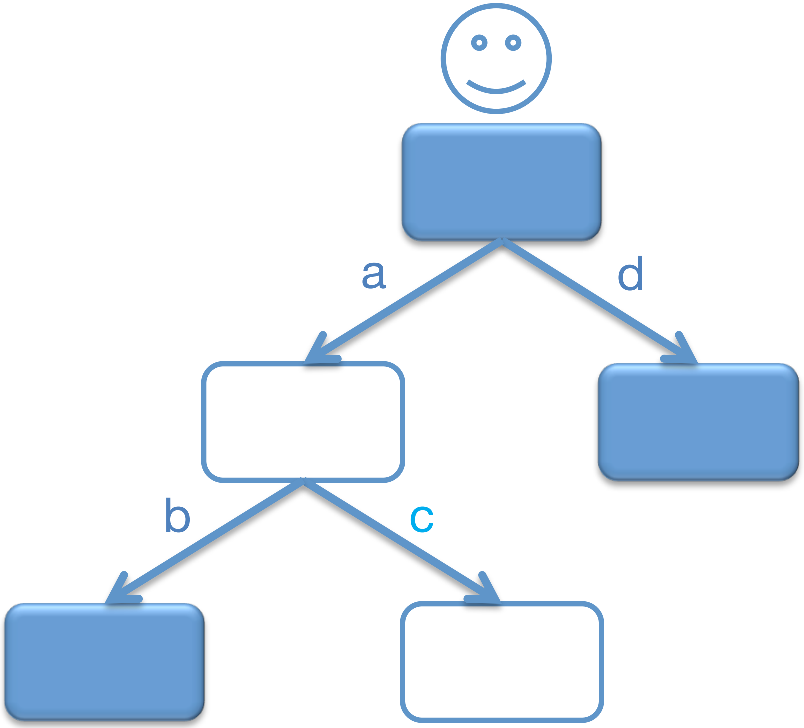
Not responsible for
Validation
Access control
Search
Versioning

Current implementations

	DocumentMK	TarMK (SegmentMK)
Persistence	MongoDB, JDBC	Local FS
Conflict handling	Partial serialisation	Full serialisation
Clustering	MongoDB clustering	Simple failover
Sharding	MongoDB sharding	N/A
Node Performance	Moderate	High
Key use cases	Large deployments (>1TB), concurrent writes	Small/medium deployments, mostly read

Access Control

Accessible paths



∃xistentialism

- All paths traversable
  - Node may not exist
  - Decorator on NodeStore

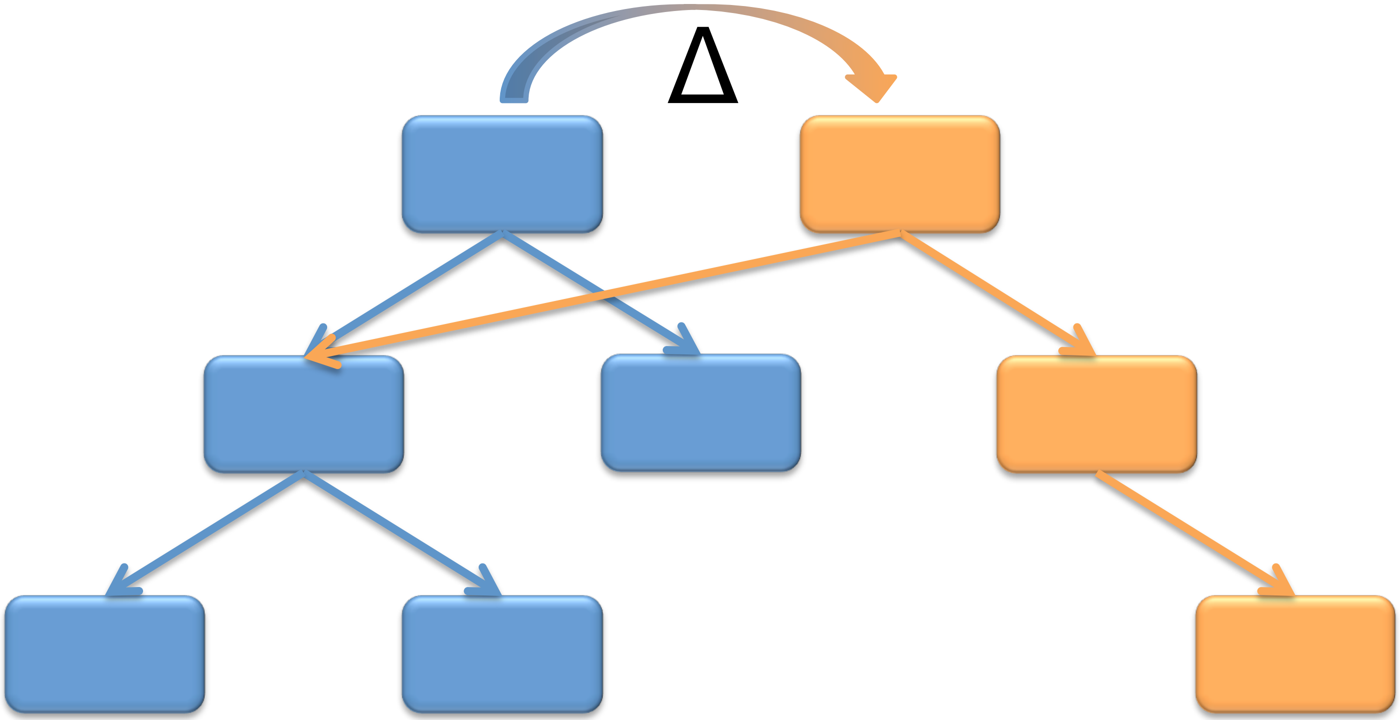
```
root.getChildNode("a").exists();      ⇒ false  ✗

root.getChildNode("a")
    .getChildNode("b").exists();      ⇒ true   ✓
```

Comparing Revisions

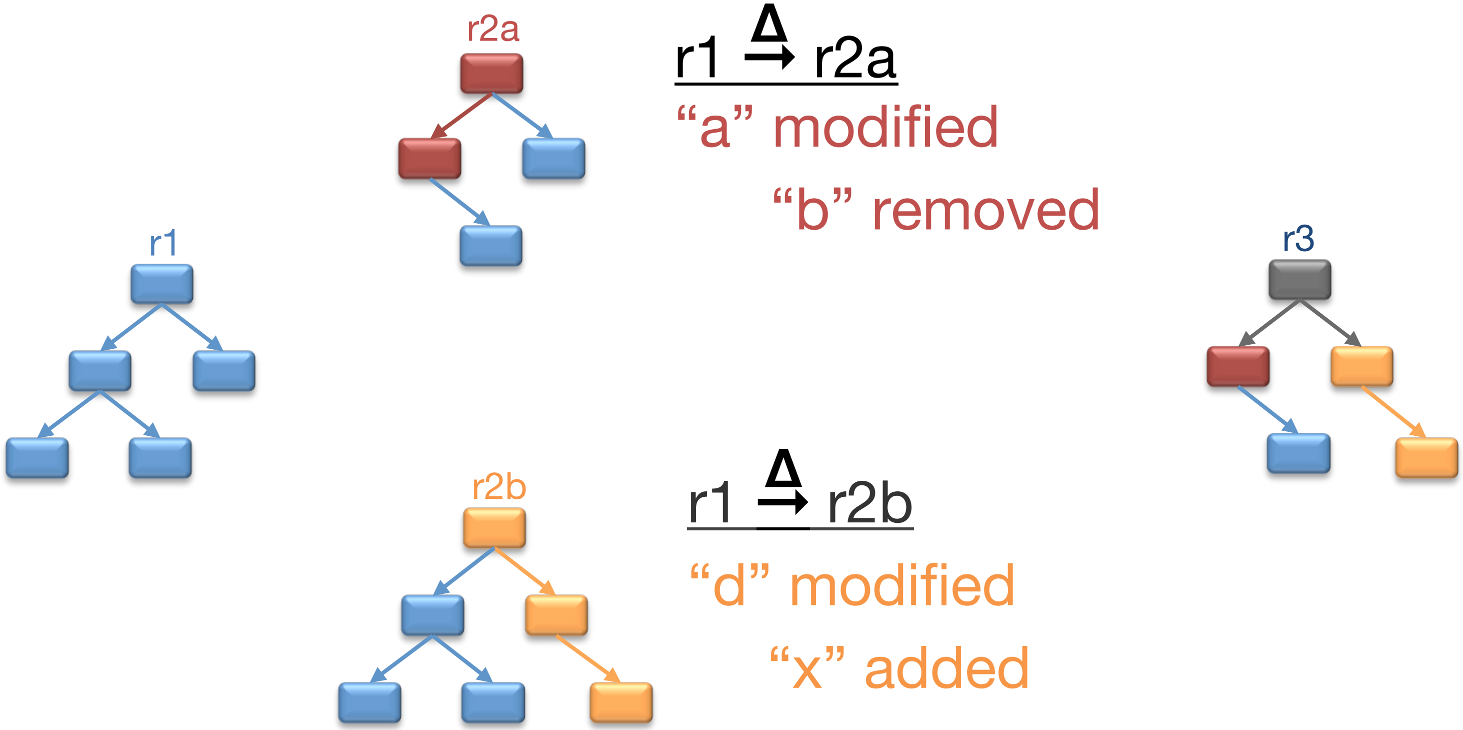
Content diff

- What changed between trees
- Cornerstone for
  - Validation
  - Indexing
  - Observation
  - …

What changed?



Example: merging



$r1 \xrightarrow{\Delta} r2a$

“a” modified

“b” removed

$r1 \xrightarrow{\Delta} r2b$

“d” modified

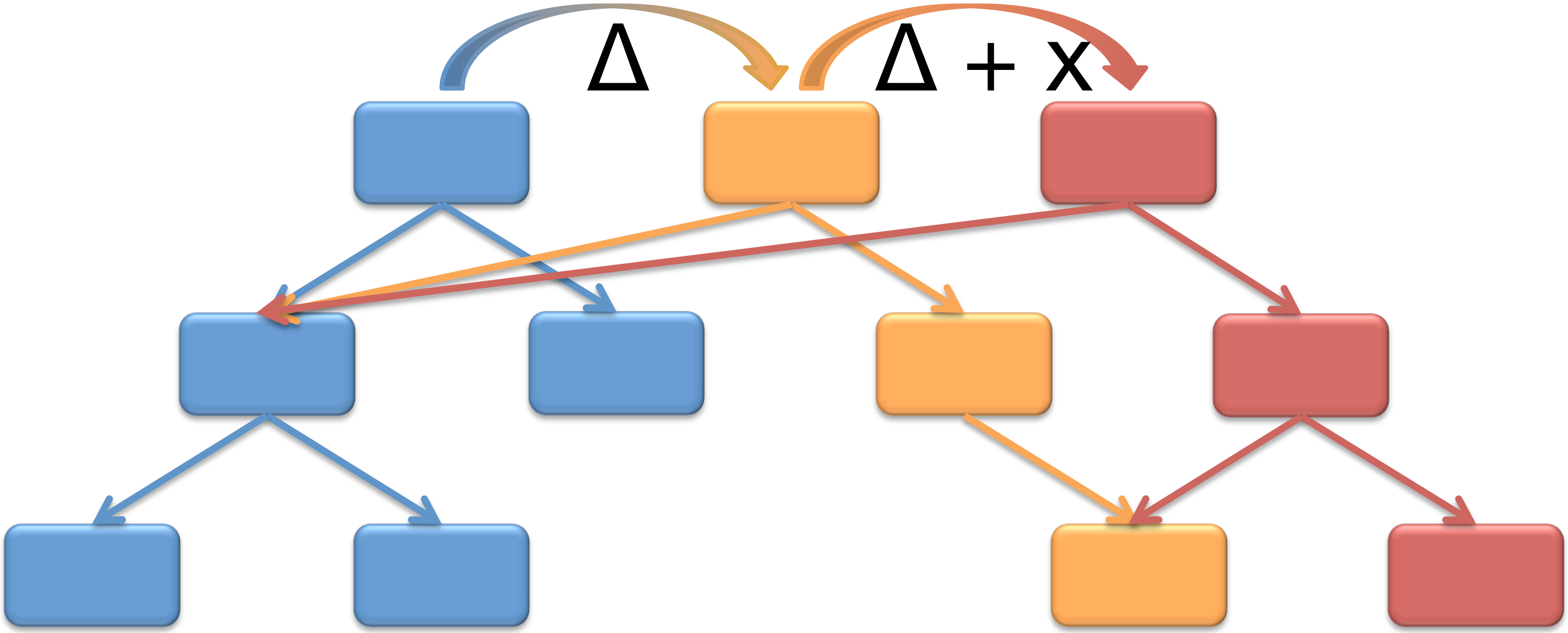
“x” added

Commit Hooks

Commit hooks

- Key plugin mechanism
  - Higher level functionality
    - Validation (node type, access control, …)
    - Trigger (auto create, defaults, …)
    - Updates (index, …)

Editing a commit



Commit hooks

- Based on content diff
  - – pass a commit
  - – fail a commit
  - – edit a commit

- Applied in sequence

Type of hooks

	CommitHook	Editor	Validator
Content diff	Optional	Always	Always
Can modify	Yes	Yes	No
Programming model	Simple	Callbacks	Callbacks
Performance impact	High	Medium	Low

Observers

Observers

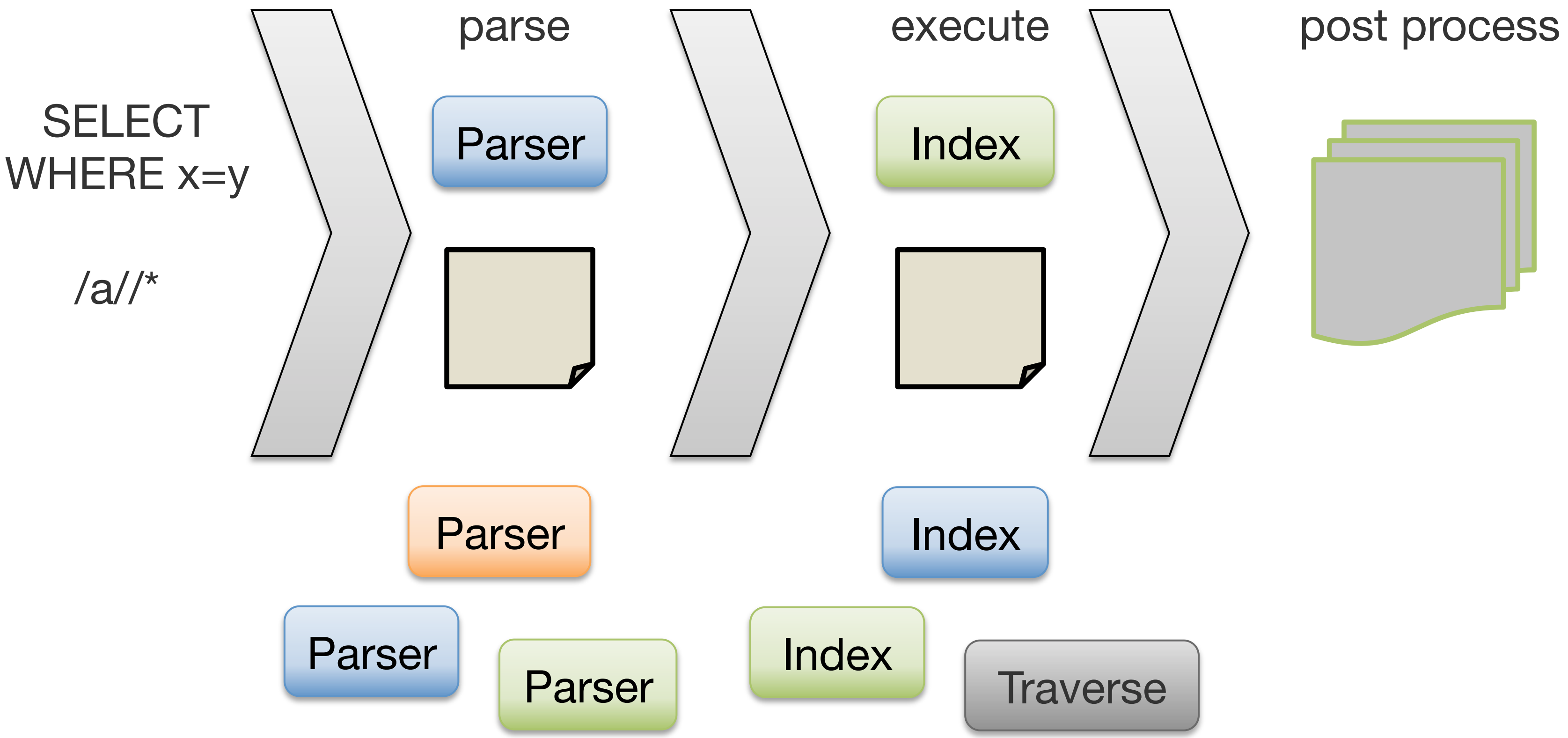
- Observe changes
  - After commit
  - Often does a content diff
  - Asynchronous
  - Optionally synchronous
    - Local cluster node only

Examples

- JCR observation
- External index update
- Cache invalidation
- Logging

Search

Query Engine

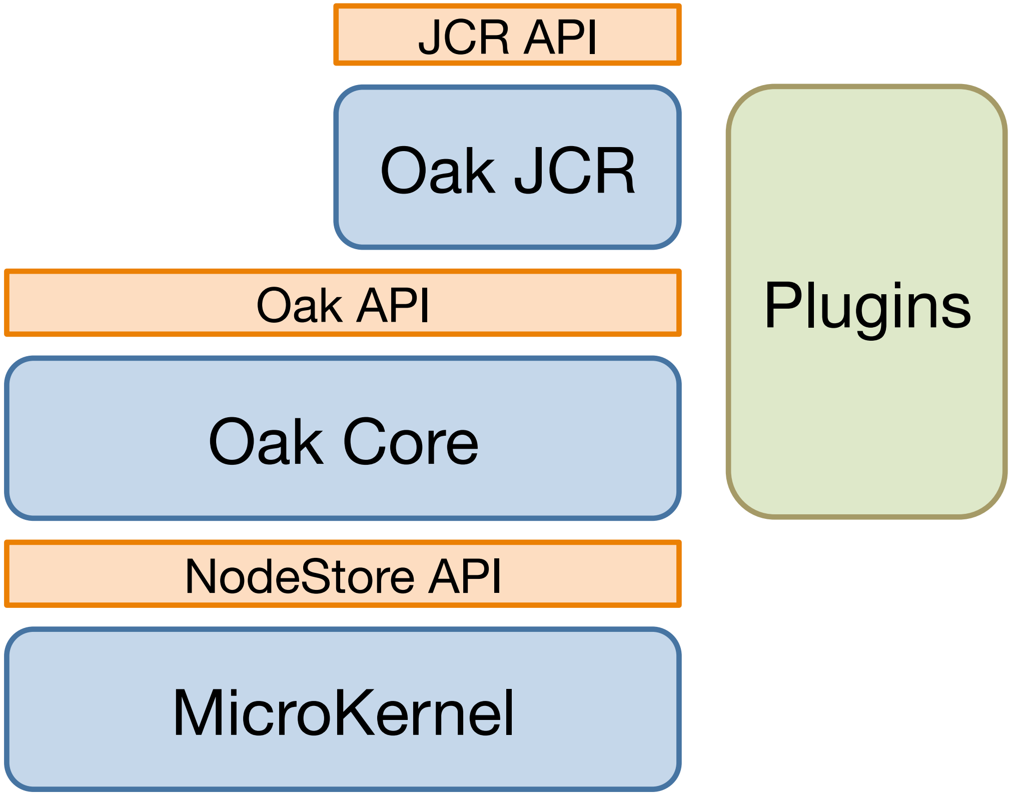


Index Implementations

- Property (ordered)
- Reference
- Lucene
  - In-content or file system
- Solr
  - Embedded or external

Big Picture

Big picture


JCR API
Oak JCR
Oak API
Oak Core
NodeStore API
MicroKernel
Plugins

Resources

http://jackrabbit.apache.org/oak/

Appendix

Resources

http://jackrabbit.apache.org/oak/

http://jackrabbit.apache.org/oak/docs/

https://svn.apache.org/repos/asf/jackrabbit/oak/trunk/

Session Notes

Slide 1

This presentation is mainly about Oak's architecture and design. Understanding these concepts gives crucial insight in how to make the most out of Oak and to why Oak might behave differently than Jackrabbit 2 in some cases.

Slide 2

Jackrabbit Oak started early 2012 with some initial ideas dating back as far as 2008. It became necessary as many parts of Jackrabbit 2 outgrew their original design. Most of Jackrabbit 2’s features date back to the 90-ies and are not well suited for today's requirements. Oak was designed to overcome those challenges and to serve as the foundation of modern web content management systems.

Key design goals:

- scalable writes. The web is not read only any more.
- large amounts of data. There is much more as a few web pages nowadays.
- Built in clustering. Instead of built on top
- Customisable
- OSGi friendly

Since Oak doesn't need to be the JCR reference implementation, we gained some additional design space by not having to implement all of the optional features (like e.g. same name siblings and support for multiple work spaces).

Slide 3

- CRUD: this presentation first covers the underlying persistence model: the tree model and basic create, read, update and delete operations.

- Changes: being able to track changes between different revisions of a tree turns out to be crucial for building higher level functionality.

- Search: while nothing much changed on the outside, search is completely different in Oak wrt. Jackrabbit 2.

Slide 4

Let’s consider a simple hierarchy of nodes. Each node (except the root) has a single parent and any number of child nodes. The parent-child relationships are named, i.e. each child has a unique name within its parent. This makes it possible to uniquely identify any node using its path: a user can access all content by path starting from the root node.

This is a key different to Jackrabbit 2 where each node was assigned an unique id to look it up from the persistence store. In Oak nodes are always addressed its path from the root. In this sense Oak stores (sub) trees while Jackrabbit 2 stores key value pairs. In Oak one traverses down from the root following a path while in Jackrabbit 2 traversal was from a node to its parent up to the root.

- Tree persistence vs. key/value persistence
- Path vs. UID as primary identifier
- Traversing down vs. traversing up

Slide 5

Let’s consider what happens when another user updates parts of the tree. For example adds a new node at /d/x. Such in place changes might confuse other users whose tree suddenly change.

This is how Jackrabbit 2 works, each update is immediately made visible to all users. Unfortunately, beyond the potential for confusion, this design turns out to be a major concurrency bottleneck, as the synchronisation overhead of keeping everyone aware of all changes as they happen becomes very high. The existing Jackrabbit architecture was heavily optimized for mostly-read use cases, with only occasional and rarely concurrent content updates. Unfortunately that optimisation no longer works too well with increasingly interactive web sites and other content applications where all users are potential content editors.

More generally the way such state transitions are handled has a major impact on how efficiently a system can scale up to handle lots of concurrent updates. Many noSQL systems use the concept of eventual consistency which leaves the rate (and often order) at which new updates become visible to users undefined. This solves the concurrency issue, but can lead to even more confusion as it might not be possible to clearly define the exact state of the repository.

The hierarchical structure of Oak allows us to solve both of these issues by borrowing an idea from version control systems like Git or Subversion.

Slide 6

Instead of overwriting the existing content, a new revision of content is created, with new copies of the parent nodes all the way to the root if needed. This allows all users to keep accessing their revision of content regardless of what changes are being made elsewhere. All users are under the impression they operate on a private copy of the whole repository (MVCC)

To make this work, each revision is assigned a unique revision identifier, and all paths and content accessed are evaluated in the context of a specific revision. For example the original version of the /d node would be found by following that path within revision r1, and the new version of the node by following the path in revision r2. The unchanged node /a/c would be reachable and identical through both revision r1 and r2.

Additionally the repository keeps track of the HEAD revision that records what the latest state of the repository is. A new user that has not already accessed the repository would start with the HEAD revision.

The ordered list of the current HEAD revision together with all former HEAD revisions form a journal, which represent a linear sequence of changes the repository went through until eventually reaching the current HEAD revision.

Slide 8

Let’s consider what happens when a user on an old revision wants to go forward the HEAD revision. Unlike with classic Jackrabbit, where this would always happen automatically, in Oak this state transition is handled explicitly as a refresh operation.

Refresh in Oak is explicit where it was implicit in Jackrabbit 2. For backward compatibility Oak provides some "auto refresh" logic. See the Oak documentation for further details.

Slide 9

After the refresh, the older versions of specific nodes may no longer be accessible by any clients, and will thus become garbage, which the system will eventually collect. This is a key difference to a version control system, and allows Oak repositories to work even with workloads like thousands of updates per second without running out of space.

Slide 11

What about concurrent updates? Here we have two competing revisions, r2a and r2b. Both modify the same base revision r1. The r2a revision removes the node at /a/b, and r2b is the revision we saw earlier, which add a new node at /d/x.

Slide 12

We start with the base revision r1. Then the two new revisions are created concurrently. Finally, the system will automatically merge the results, which will create the new revision r3 containing the merged changes from r2a and r2b. The merge could happen in different cluster nodes, different data centres or even different local disconnected copies (e.g. like git clone).

Slide 13

What happens when merging is not possible because two concurrent revisions conflict? There are several strategies for handling this case:

- Full serialisation: don't allow concurrent updates and fail. This heavily impacts the write rate in the face of many concurrent writers.

- Partial serialisation: instead of locking on the whole tree, lock on the root of the modified sub tree. Allows concurrent updates of changes to separate sub trees.

Both strategies are currently implemented by Oak depending on the persistence back-end in use.

Slide 14

Instead of pure serialisation a more sophisticated approach would be to semantically merge conflicting changes. Generally this needs domain knowledge and cannot be fully implemented in the persistence layer.

- Partial merging: persist conflict markers along with the conflicting changes for deferred resolution by e.g. and administrator.

- Full merging: this inevitable leads to data loss as one of the changes have to be preferred over the other. This is unless we can establish a total order over all revisions, which is usually not the case.

Oak currently does not implement these strategies, though they could be plugged in.

Slide 16

Replicas are straight forward: each replica only needs to follow the primary. Thanks to the immutable append only model of persistence, there is no need for invalidation. Replicas can do their individual garbage collections cycles.

As a variant a replica can also serve as cache by only following frequently accessed parts of the tree and ignoring updates to other parts.

Slide 17

Sharding by path is the most straight forward variant. However as sub tree sizes tend to be not evenly distributed shards end up to greatly vary in size.

Sharding by level turn out to be even more problematic as there will be more content the deeper the tree while the root only has a single node. Although there are more revisions on the root (as every change creates a new root node), garbage collection will keep that number within reasonable limits.

Sharding by content hash is what the DocumentMK does and which turned out to be most effective. A drawback of this approach is the loss of locality: the nodes of a path tend to be spread across various shards forcing navigational access to access multiple shards. An idea for addressing this is to allow each shard to cache a node's parent nodes. Since the lower levels contains the most content caching the parents is cheap while at the same time it restores locality.

Slide 19

Unfortunately there is a bit of a naming confusion in Oak as the terms MicroKernel and NodeStore might both refer to APIs and to architectural components and are used somewhat interchangeably.

One way to think of it is that the MicroKernel is an implementation of the tree model. While the NodeStore is a Java API for accessing the nodes of that tree model.

A MicroKernel implementation is responsible for all the immutable update mechanics of the content tree along with conflict handling, garbage collection, clustering and sharding. It doesn't have any higher level functionality like validation, complex data types, access control, search or versioning. All the latter a implemented on to of the NodeStore API.

Slide 20

Oak comes with basically two MicroKernel implementations: the DocumentMK (formerly MongoMK) and the TarMK (aka SegmentMK). The DocumentMK started out as being MongoDB backed but is now more flexible regarding the choice of the back-end. A JDBC back-end is currently being worked on.

- The DocumentMK leverages the clustering and sharding capabilities of the underlying back-ends and implements partial serialisation. Due to the extra network layer involved, it has moderate single node performance, but scales with additional cluster nodes.
- The TarMK uses local tar files for storage. It is not currently not cluster aware and only offers a simply fail over mechanism but offers maximal single node performance.

The two MicroKernel implementations cover different uses cases: where DocumentMK is preliminary for large deployments involving concurrent writes, the TarMK is better suited to small to medium deployments, which are mostly read. We expect the gap between the two implementations to decrease in the future as there are ideas to make the TarMK more cluster ready while the DocumentMK still has room for improved performance. Finally other implementations (e.g. Hadoop based) are quite possible with the JDBC back-end for DocumentMK most likely being the first one.

Slide 22

Access control is probably the most trick feature of JCR to implement. However it is also a very prominent feature and might as well be considered "the killer feature" for quite some content centric applications as implementing fine grained access control on top of applications is difficult and error prone.

Access control in JCR allows a node to be accessible while it's parent isn't. This is contrary to the tree model where all access is by path traversing down from the root of the tree. We solved this in Oak by introducing the concept of "existence" for a node. Nodes that are not accessible are still traversable but do not exist. That is, the result of asking for a non accessible child node is indistinguishable from a "really" non existing child: both do not exist. With this all syntactically correct paths eventually resolve to a node, which however might not exist.

In this example all paths are traversable, however as only /, /d, and /a/b are accessible only those nodes exist. The nodes at /a and /a/c do not exist.

Slide 23

This approach leads to a clean architecture where API clients needn't care about null values or exceptions. Just traverse the path starting from root and check for existence at the end.

The existence concept is implemented as a decorator of the tree model on top of the MicroKernel by a thin wrapper of the relevant parts of the NodeStore API.

Slide 25

The content diff is key to most of Oak's higher level functionality. It allows to just process the parts of the content tree that changed (during or after a commit) instead of going through the whole repository. It is used e.g. for:

- Validation (node types, referential integrity, item names)
- Observation: apparently commit boundaries are get lost when the content diff is done across more than a single revision. This is an important limitation wrt. Jackrabbit 2 as some commit related information as the user or the time stamp might not always be available in Oak.
- Indexing

Slide 26

Finding out what changed between revisions is crucial for much of Oak's higher level functionality. Changes are extracted by comparing two trees. This is similar to a diff in a version control system but applied to content trees.

A content diff runs from the root of two trees to its leaves. A node is considered changed when it has added/removed/changed properties or child nodes. Each changed node recursively runs a content diff on each of its changed child nodes.

Comparing trees in this way is generic as it works on any (sub) trees, not only from the root. It is however heavily optimise for the case where an earlier revision is compared against a later revision as this is the case most often encountered.

In the depicted case the sub-tree at /a doesn't need deep comparison as it is shared between both revisions. The diff process can stop right here.

Slide 27

Merging concurrent changes relies on content diffs: first the diff between r1 and r2a is applied to r1 followed by the diff between r1 and r2b, which will finally result in the new revision r3.

Slide 29

Commit hooks and their variants provide the key plugin mechanism of Oak. Much of Oak's functionality is in one way or another implemented in this way:

- Validation of node types, access control, protected or invisible content
- Trigger for auto created or default values
- Updates for indexes or caches

Slide 30

Commit hooks rely on content diffs and allow for validation or editing of a commit before actually persisting it.

Slide 31

A commit hook can either

- pass a commit on to be persisted unchanged
- fail a commit so it wont get persisted
- edit a commit so a changed tree will be persisted

Commit hooks are applied in sequence and tend to be expensive as most likely each of them does a separate content diff.

Slide 32

Editors and Validators are commit hooks in disguise. They do a single content diff calling back to the respective implementations. This makes them considerably less expensive the raw commit hooks. However due to the call back based programming model they are more difficult to use.

Slide 34

Observers are related to commit hooks. In fact both share the same signature as they get a before and an after state and can run a content diff on those processing the differences. However observers are run after the fact. That is, after the content has been persisted. Observers report what has happened while commit hooks report what might happen.

Observers might or might not see each separate revision. After big cluster merges they might receive bigger chunks spanning over multiple individual commits. There might be no way to get to those individual commits as they might have already been garbage collected on the other cluster node. The important part is that each observer will see a monotonically increasing sequence of revisions.

Slide 35

As JCR observation in Oak is based on observers and those might not get to see all commits, commit boundaries are usually lost. That is information attached to individual commits like the user or a time stamp are only available on a best effort basis in Oak.

External index updates are e.g. used for integration Solr.

Slide 37

Although search supports the same languages as Jackrabbit (SQL and XPath), the underlying implementations greatly differ.

Oak has pluggable parsers and indexes. For each query a suitable parser is first searched and - if available - used to parse the query into an intermediate representation. Then all registered indexes are asked to provide a cost estimate for executing the query. Finally the query is executed choosing the cheapest index. If there is no suitable index for a query the synthetic "traverse" index will be used. This index, while very slow, allows each query to eventually succeed.

Oak's approach to query execution is closer to the RDBMS world than Jackrabbit 2 as it allows for creating indexes to speed up queries.

Slide 38

Indexes are specified in content via special index definition nodes.

- Property index for properties of a given name. Can optionally by ordered.
- Reference index for looking up referees of referenceable nodes
- Lucene full text index for full text search. Stored in content replicates it automatically across cluster nodes and make it easy to back up along with the content.
- Solr can be run embedded but usually runs externally

Slide 40

- The MicroKernel implements the tree model
- The NodeStore API exposes the tree model as immutable trees
- Oak core implements mutable trees on top of the NodeStore API.
- Plugins contribute much of the Oak core functionality like access control, validation, etc. through commit hooks and observers.
- The Oak API exposes mutable trees, whose behaviour is shaped through the respective plugins configured.
- JCR bindings contribute the right shaping for JCR by provisioning Oak with the right set of plugins.

Other bindings besides JCR are possible. E.g. a HTTP binding as an alternative to the current WebDav implementation. Also new application could use the Oak API directly and only reuse the plugins as necessary for their needs.