Integrate Indexing / Search

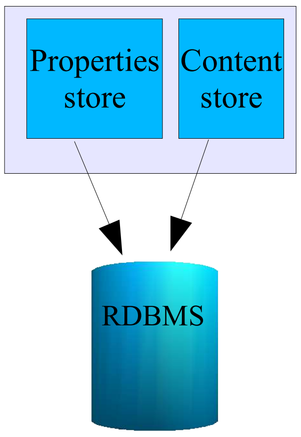
Indexer, Search engine and store are closely coupled. A store might have an indexer per se (RDBMS using indexed tables), or you may associate an indexer to a store.

Store:
stores content and properties on some kind of datastore (RDBMS, file system,...).

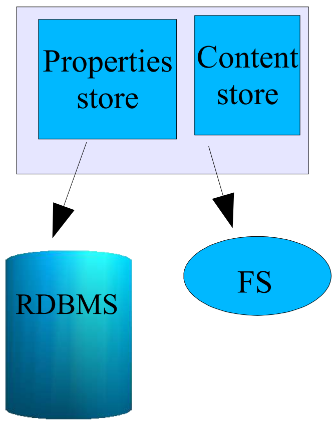
Indexer:
writes property and / or content index data to some kind of datastore to help the associated Search Engine to quickly find resources.

Search engine:
Gets the basicsearch request and generates calls / statements to the underlying indexer(s) and / or stores. If different stores / indexers are used, merging might be necessary.

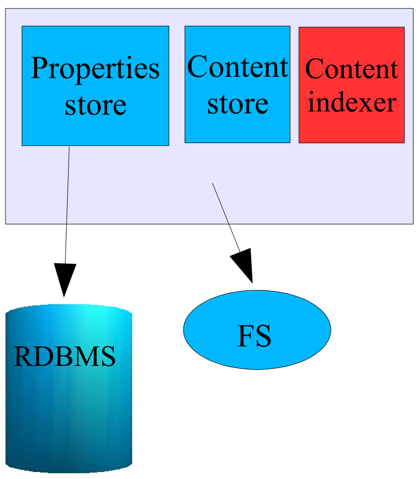
Lets regard three different scenarios:



Scenario I: Properties and content stored in RDBMS, one store implementation for properties and content is used. RDBMS provides indexing functionality and SEARCH capabilities.



Scenario II: Property data stored in RDBMS, (i.e. MySQL), content stored on filesystem, different stores for properties and content. RDBMS provides indexing and search, FS no indexing and no search.



Scenario III: Properties stored on RDBMS, content on FS using a content indexer (i.e. Lucene).

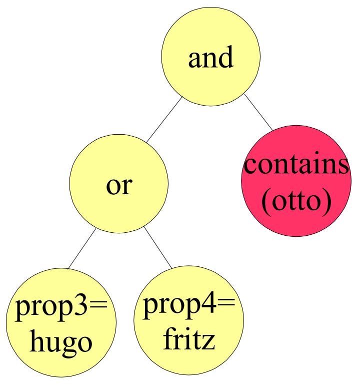
The Search Engine

Lets write Search Engines for those scenarios.

The <basicsearch> request comes as a JDOM tree. Lets look at the <where>, which is the most interesting part at the moment.

```
<where>
  <and>
    <or>
      <eq>
        <prop>
          <prop3/>
        </prop>
        <literal>hugo</literal>
      </eq>
      <eq>
        <prop>
          <prop4/>
        </prop>
        <literal>fritz</literal>
      </eq>
    </or>
    <contains>otto</contains>
  </and>
</where>
```

That somehow gives a semantic tree:



The SEARCH request is processed in two major steps, parseRequest() and executeRequest(). Within parseRequest() the Search Engine (SE) compiles the semantic tree into an executable tree. How this tree looks like, depends on the underlying store / indexer.

○	Semantic expression
▭	Executable expression

Scenario I:
SE walks down the tree to prop3="hugo" and compiles the fragment of the WHERE clause. Same with prop4="fritz". One step up, the OR expression with the two already compiled expressions can be compiled into a new fragment, *prop3="hugo" OR prop4="fritz"*.
As content is stored in RDBMS as well, and we assume, the underlying RDBMS has full text search, we can generate a fragment *content like („otto")*. At last the uppermost AND can be compiled, our executable tree now is shrinked to one executable node that has an execute method calling the RDBMS with SQL somehow looking like (using pseudo SQL, as real statement highly depends on the table structure):

```
Execute() {
  result=rdbms.execute (
    „... where ((prop3="hugo"
          or prop4="fritz")
         and content like („otto"))
    ...")
}
```

The search engine creates ONE SQL statement, returning all URIs matching the search criteria. As RDBMS maintains its own index, the search is fast.

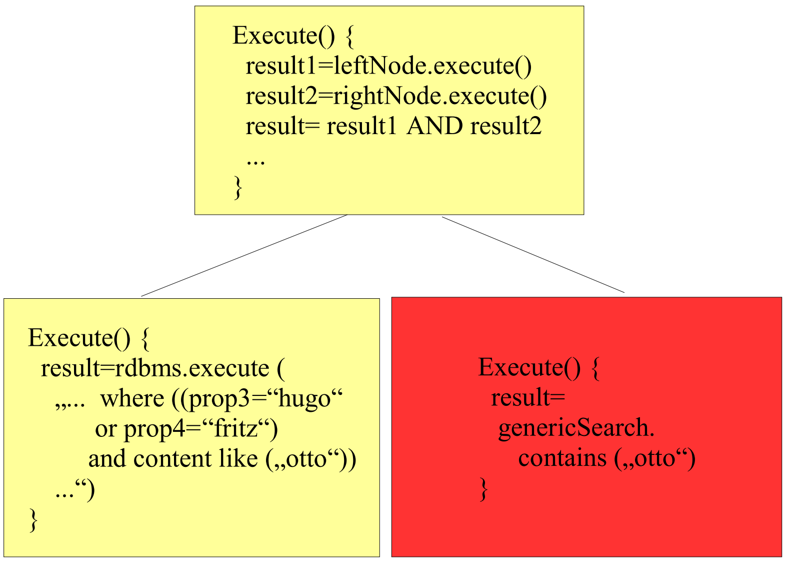
Scenario II:
The left part of the tree is compiled as in scenario I, giving an executable expression with a where fragment

prop3="hugo" OR prop4="fritz".

As the file content store does not implement the ExpressionFactory, the generic search implementation (implemented in the slide kernel) takes over this part.

resultset = genericSearch.contains („otto")

The AND expression now has a left part with an SQL expression to send to RDBMS, and right part with a method call to genericSearch.contains, it cannot be resolved further. Both expressions are executed, the resultsets are intercepted.

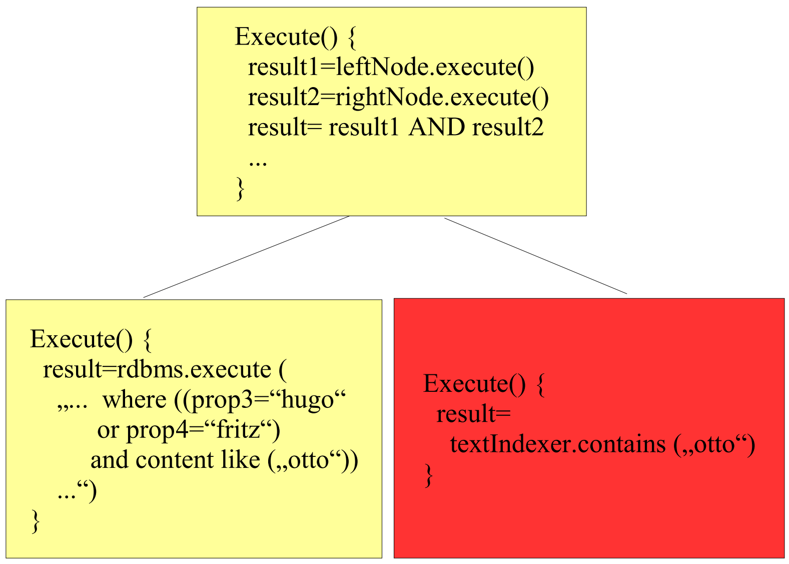


```
Execute() {
  result1=leftNode.execute()
  result2=rightNode.execute()
  result= result1 AND result2
  ...
}
```

```
Execute() {
  result=rdbms.execute (
    „... where ((prop3="hugo"
          or prop4="fritz")
         and content like („otto"))
    ...")
}
```

```
Execute() {
  result=
   genericSearch.
      contains („otto")
}
```

Scenario III:
The left part is similar to scenario II. The text indexer however implements the ExpressionFactory, so a textIndexer specific expression is generated.



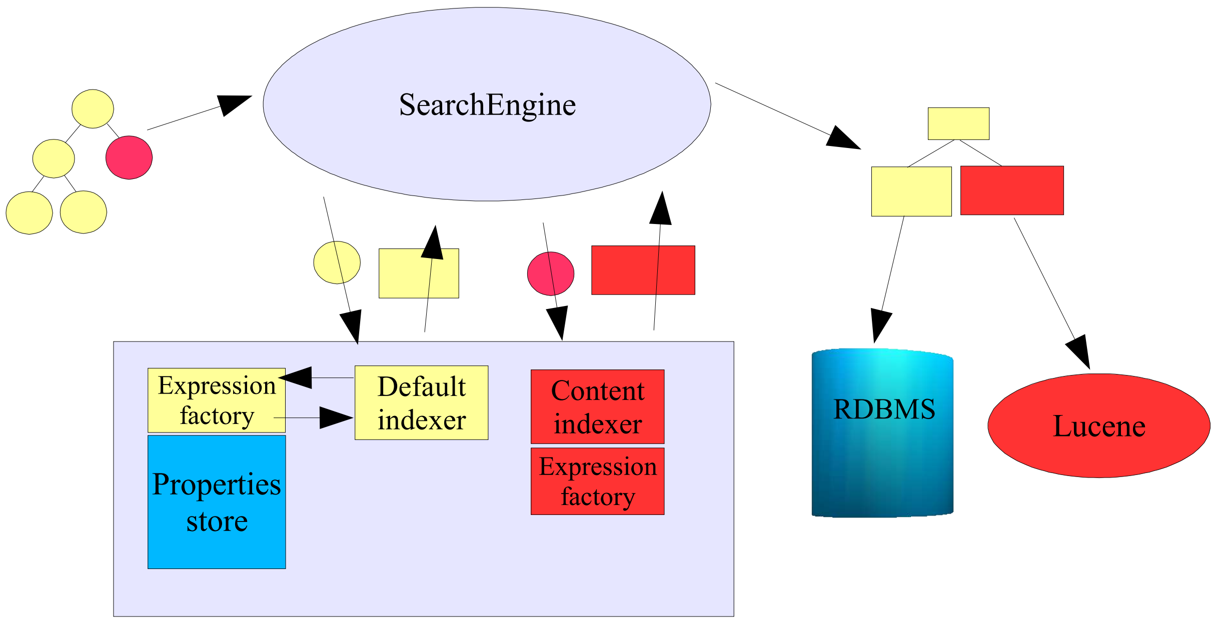
```
Execute() {
  result1=leftNode.execute()
  result2=rightNode.execute()
  result= result1 AND result2
  ...
}
```

```
Execute() {
  result=rdbms.execute (
    „... where ((prop3="hugo"
         or prop4="fritz")
        and content like („otto"))
    ...")
}
```

```
Execute() {
  result=
    textIndexer.contains („otto")
}
```

In executeRequest() the execute method of the root node is called, a result is delivered regardless of the underlying store.

But how the SE knows, which kind of executable expression to create? The Indexer knows! Lets look at the parseRequest() step (scenario III):



The store has two Indexers, descriptorIndexer and contentIndexer. In our scenario the descriptorIndexer is the defaultIndexer doing nothing, contentIndexer is using Lucene. An Indexer provides an ExpressionFactory. The defaultIndexer gets its ExpressionFactory from the properties store. To provide compatibility with existing store implementation, reflection is used to check, if the store implements ExpressionFactory.

The SE visits each operator in the <where> node, and calls the ExpressionFactory of the appropriate indexer, i.e. <gt> is processed by the ExpressionFactory of descriptorIndexer (here delegated to properties store), <contains> by the ExpressionFactory of contentIndexer.

Indexer

An Indexer inspects the content and / or properties of a document and stores information to help a SearchEngine to quickly find this document.

Two Indexers are defined within one store, one for properties, one for content. If no indexers are defined, a default implementation is loaded.

Each time, a resource is created, updated or deleted, the indexer must update it's indexes. This is implemented in the ParentStore (AbstractStore), so the concrete store implementation does not need to take care about. As the indexer is triggered by the store, no IndexHelper is necessary.

Indexer implements Service, so it may take part at the two phase commit. However, the implementation might fake the two phase commit and updates the indexes asyncronously for maximum of performance.

Indexer provides an ExpressionFactory.

Configuration

The indexers are defined in Domain.xml, in the <store> element. See following sample:

```
   <definition>
   <store name="jdbc" classname="org.apache.slide.store.BindingStore">
     <nodestore classname="org.apache.slide.store.impl.rdbms.JDBCStore">
       <parameter name="driver">com.mysql.jdbc.Driver</parameter>
       <parameter name="url">jdbc:mysql://localhost:3306/test?autoReconnect=true</parameter>
       <parameter name="user">myUser</parameter>
       <parameter name="password"/>
       <parameter name="adapter">org.apache.slide.store.impl.rdbms.MySqlRDBMSAdapter</parameter>
     </nodestore>
     <securitystore>
       <reference store="nodestore"/>
     </securitystore>
     <lockstore>
       <reference store="nodestore"/>
     </lockstore>
     <revisiondescriptorsstore>
       <reference store="nodestore"/>
     </revisiondescriptorsstore>
     <revisiondescriptorstore>
       <reference store="nodestore"/>
     </revisiondescriptorstore>
     <contentstore classname="org.apache.slide.store.txfile.TxFileContentStore">
       <parameter name="rootpath">store/content</parameter>
       <parameter name="workpath">work/content</parameter>
     </contentstore>
     <!-- no propertiesindexer specified, default is loaded. -->
     <contentindexer classname="org.apache.slide.store.LuceneIndexer">
      <!-- define any parameter needed to configer LuceneIndexer. -->
       <parameter name="myParm1">myVal1</parameter>
     </contentindexer>
     <parameter
name="basicQueryClass">org.apache.slide.search.basic.sample.BasicQuerySample</parameter>
   </store>
   <scope match="/" store="jdbc"/>
  </definition>
```

Outlook

To allow a certain degree of orthogonality, the indexer could be defined and configured outside the <store> definition and then referenced within the store. So you could have one indexer for several stores / scopes.

An indexer should be regarded as a framework, where you can plug in contenttype specific indexers. If an indexer can extract property like data (author from a word document, composer from a music file, ...), this data should be written to the revisiondescriptor.