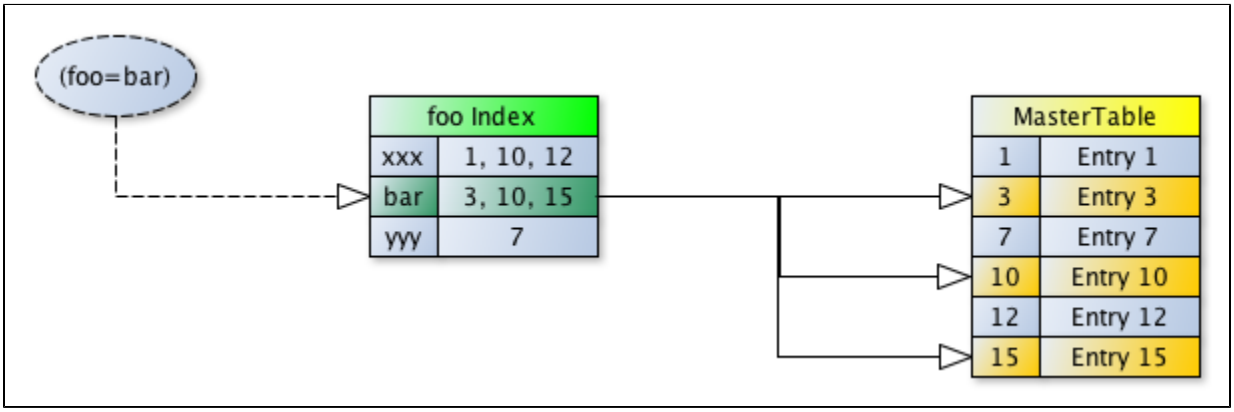
Index and IndexEntry

Index Abstraction

An Index is a bidirectional LUT (Look-Ups Table) which sorts keys (forward index) and values (reverse index). It can be built on any physical attribute or a computed characteristic of an entry.

Index Tuples contain a value for the attribute and the entry identifier which represents a pointer into the MasterTable. This way lookups into the master table for entries with a specific attribute value are fast. For example, when conducting a search for all entries where the **foo** attribute is equal to the String 'bar', the search engine looks first to see if an index is available for **foo**. If an Index on **foo** exists then it is used to quickly list the identifiers of all entries with attribute **foo** equal to 'bar'. This lookup occurs in almost constant time and the Index can be walked to only return those identifiers over the Tuples with key set to 'bar'. The following schema expose the relations between all those elements :



Without an Index on **foo**, the following sequence of events must occur on each entry in the master table:

1. the entry must be read from the MasterTable as a byte[]
2. the entry must then be deserialized
3. the **foo** attribute must be extracted
4. each value of the **foo** attribute must be compared to see if it equals 'bar'

As you can see this is an expensive proposition. With very large directories search may take a very long time or may return without the intended entries because time limits are reached. Indices are critical for increasing search performance.

Even if we have all the entries in the cache, this is still an O(n) operation. We just spare the disk access and the deserialization phase.

LDAP amortizes the expense of building indices at add, modify and delete times in return for faster search performance. LDAP servers for this reason are highly indexed and are better suited for read intensive applications.

Index Forward and Reverse tables

At the moment, an Index is implemented using 2 tables :

- The Forward table, which maps the key to a list of entry's ID in the master table
- The Reverse table, which maps the entry's ID to the keys

The forward table is used to retrieve the entry from the MasterTable, when we only have a key. The key can be associate with more than one entry, so we will get back a cursor on the associated entry's ID when we do a lookup.

The reverse index is used when we do a search on a combination of attributes : (&(cn=test)(sn=acme)). If we suppose that the selected primary index will be the **CN** index, then we will pull a list of entry's ID. Now, using the **SN** reverse index, we can avoid pulling each entry from the MasterTable, simply because we can check in the **SN** reverse table that the selected IDs are present or not.

This might be superfluous : it's enough to pull the **SN** entry's ID for the **acme** value, and do an intersection with the first set of entry's ID. The biggest advantage would be that we will speed up the modification operations if we were to remove those reverse tables.

Index Cursor's Get IndexEntry Objects

Index Cursors, unlike Tables that return Tuples, return IndexEntry objects instead on get() operations after positioning the Cursor. Indices always use a Long identifier into the master table for forward LUT keys and reverse LUT values. So instead of returning Cursors over two kinds of Tuples over these LUTs, their Cursors return IndexEntry objects which swap Tuple key/value pairs where appropriate.

For example, let's presume the **foo** index contains String values for the foo attributeType. The forward LUT will map Strings representing foo attribute values like 'bar' to Long identifiers pointing to entries in the MasterTable. The reverse LUT will map Long identifiers pointing to entries in the MasterTable to Strings representing values of the foo attribute in that entry. The Tuples of these LUT's look like so:

- Forward LUT => Tuple<String,Long>
- Reverse LUT => Tuple<Long,String>

Index Cursors walk these LUT's using Table Cursors that return Tuples. Regardless of whether the forward LUT or the reverse LUT is used, Tuples are transduced into IndexEntry objects which always return the Long identifier into the master table on calls to getId() and the attribute value String (or whatever - really a generic type - String is used here for the foo attribute example) on calls to getValue(). This way Index Cursors do not care if the underlying Tuple Cursor walks the forward LUT or the reverse LUT.

Index used in the server

We used two distinct kind of index in the server :

- system indexes
- user indexes

The **system** indexes are those we create at startup. An administrator can modify some of their characteristics, but basically, they will always exist. The list of existing System index is given in the following table :

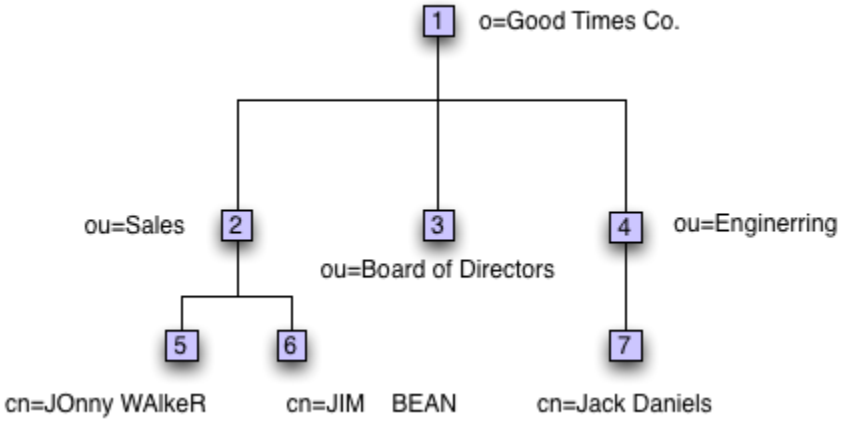
Index	Key	duplicates	Description
ObjectClass	String	Allowed	Associate an ObjectClass to a list of entries ID
Rdn	ParentIdAndRdn	Forbidden	Stores the relation between a child entry and its parent in the DIT. The key is a compound element built using the entry's RDN and the entry ID.
oneLevel	Long (ID)	Allowed	An index storing a relation between an entry and all its children
subLevel	Long (ID)	Allowed	An index storing a relation between an entry and all its children, recursively
presence	String	Allowed	An index storing a relation between an AttributeType name and the list of entry's ID having this AttributeType
alias	String	Forbidden	Stores the relation between an Alias and the targetted entry ID
oneAlias	Long (ID)	Allowed	Stores the list of aliases under an alias
subAlias	Long (ID)	Allowed	Stores the list of aliases under an alias with no limit in depth
entryUUID	String (UUID)	Forbidden	Stores the relation between an UUID and the associated entry ID
entryCSN	String (CSN)	Forbidden	Stores the relation between a CSN and the associated entry ID

Those index are exposed in full details in the Structure and Organization chapter.

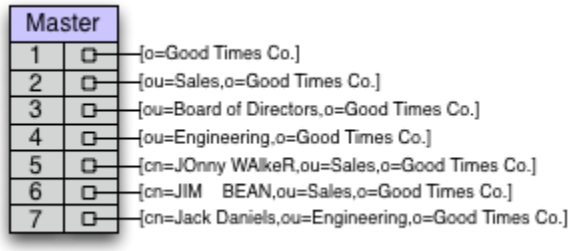
The Alias, oneAlias and subAlias indexes are probably useless.

Example Directory Information Tree (DIT)

We'll use this example DIT for the purpose of our discussions around indices.



Here we have an example tree where the nodes are labeled with their user provided relative distinguished names at the point of creation. The numbers in the nodes represent the order of creation and the unique identifiers of these entries representing the Tuple keys in the master table. We'll use this example and modified versions of it throughout this document. So let's take a look at how the MasterTable would look but presume the distinguished names to the right represent the full serialized entry as an array of bytes.

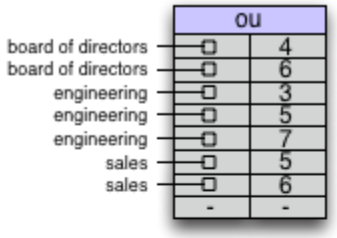


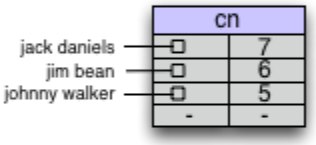
Index Normalization, and Matching

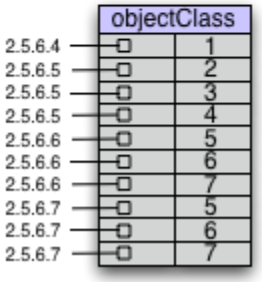
An Index can be built on any attribute. For example we can build indices on the ou, objectClass, and cn attributes in our example. Here's what the forward lookup tables of these indices would contain:

> ⚠ Remember an Index is bidirectional! This means we can lookup all the entry identifiers representing the set of entries with value 'bar' for the attribute **foo** using the forward LUT in the Index. We can also use the reverse LUT to lookup an id, to see all the values of attribute **foo** an entry may contain.

> ⚠ The previous note may be void in ADS 2.0 : we don't necessarily need a reverse LUT.







OID	Alias
2.5.6.0	top
2.5.6.4	organization
2.5.6.5	organizationalUnit
2.5.6.6	person

2.5.6.7	organizationalPerson

The index keys contain attribute values, and the values of the index contain entry identifiers which refer back into the master table. Notice the values of attributes in these indices have been transformed. This transformation is referred to as normalization or canonicalization. This means the value has been reduced to a canonical form from where variations like for example white space and case, or aliases have been removed. Other forms of normalization besides this textual normalization may exist. All variations of an attribute's value considered to be equal are reduce down to this common unambiguous form.

The transformation is governed by the **EqualityMatch** matchingRules associated with the attributeType. You'll notice for example the **ou** and **cn** attribute matchingRules result in transformation that remove whitespace variance while preserving tokenization: meaning we take out whitespace by reducing multiple whitespace characters into a single space. Also the case variance is also eliminated: all values are reduced to lowercase. Note this is possible because both **ou** and **cn** are case insensitive since they use a matchingRule which ignores case: their equality matchingRule is caseIgnoreMatch.

Attribute matchingRules may not all be this trivial. For example the attribute may have a complex binary syntax and there might be some intricate operation required to reduce the value into a canonical representation. At other times normalization may involve transforming ambiguous representations into a unique identifier that can never have variance. The **objectClass** Index is a good example of this. The objectClass attribute lists the objectClasses associated with an entry. Unfortunately LDAP allows alias names for objectClasses as well as other entities like attributeTypes (i.e. cn verses commonName). These aliases are human readable representations but they can have case variance as well and any alias can be used. Most schema entities in LDAP have an Object IDentifier or OID. OID's are by default canonical. As you can see this is the canonical representation used in the **objectClass** index. Incidentally, users are even allowed to use OID's for the value of the objectClass attribute if they wanted to.

Notice Index Tuple keys are all sorted and so are the Tuple values of the same key. Indices use matchingRules as well to order both keys and values based on matchingRules in both the forward and reverse LUTs.

If you look at the objectClass index you'll see that the OID (2.5.6.0) for the 'top' objectClass is not included in the index. This is because every entry will contain 'top' so there's no point to bloating the index including it as a value. If we need to walk all the entries in the server we can walk the MasterTable or one of the system indices like the RDN or presence indices described here.

MatchingRules, Comparators and Normalizers

LDAP defines 3 different kinds of matchingRules for attributeTypes: equality, ordering and substring. The equality matchingRule governs how attribute values are compared for equality. Ordering matchingRules control how attribute values are compared to see if one value is greater or less than another. The substring matching rule manages how embedded fields in larger values can be matched for patterns. Most LDAP servers have different kinds of indices you can create for each of these kinds of matching techniques. Sometimes there's more for example some servers have an approximate match index, which uses the soundex algorithm or a derivative to determine matches.

ApacheDS has only one type of index and makes some trade offs in space, time and simplicity while applying some common sense. ApacheDS functionally models a matchingRule as a Java Comparator coupled with a Normalizer. The Normalizer applies the transformation required to produce a canonical form. The Comparator performs checks to see if values are equal, greater or less than one another. ApacheDS Index objects use the equality matchingRule's Normalizer to create index Tuples and the ordering matchingRule's Comparator (only if defined) to determine insertion order into Index LUT BTrees. If there exists no ordering matchingRule for the attributeType then the Comparator for the equality matchingRule is used instead.

If a search uses an equality assertion with the index attribute lookups return the answers we need. If a search filter uses a substring assertion then depending on the pattern the substring matchingRules are used (only if defined for the attributeType) with a bit more processing using regular expressions to match normalized values on index walks. When > or < filter operators are used, and a ordering matchingRule is defined for the attributeType then that matchingRule's comparator is used to compare values on Index walks with Cursors. Also because of these choices ApacheDS indices are bi-directional and so ApacheDS pays a price on write operations to maintain both forward and reverse LUTs.

ApacheDS does not support approximate match since this kind of matching is language sensitive and just a pile of horse poop. For this reason approximate match does not work with internationalization. No server has yet to satisfy my search when I've tried to phonetically find names using approximate match.

So ApacheDS made some choices to stay simple. We applied a pattern of thinking where trade offs were made to keep the most common operations fastest while the least common operations may suffer slightly. Others like approximate match are essentially ignored. ApacheDS processes approximate match based filter terms as if they were equality assertions.

JDBM Base Index Implementation

The JdbmIndex uses two JdbmTables each backed by it's own JDBM BTree in the same file managed by a JDBM RecordManager for the file. These files end with the .db file extension and use the first alias name of the attributeType for the file name. The JdbmIndex leverages these JdbmTables for preforming forward reverse lookup operations to assert values as well as for generating Cursors over the Tuples of these Tables.

JdbmIndex instances require schema information to properly initialize their LUTs with the right Comparators. JdbmIndex uses operations that write Tuples into these two JdbmTables normalized values before inserting them. Attribute values inserted into these JdbmTables are normalized for speed and efficiency. If these values were not normalized, the Cursors walking these JdbmTables and lookup operations would need to normalize then compare. Search operations are heavily dependent on Index Cursors and lookup operations and normalization is an expensive process. Why repeat normalization and pay on each search operation when you can pay the price once on write operations. After all, this is LDAP, and LDAP is read optimized.

Normalization

Right now, every entry is normalized (so are its Attributes and values) when it's added into the server (same for any modification done later), so those Attributes are stored normalized in the index. Though we don't need to cache anything.