

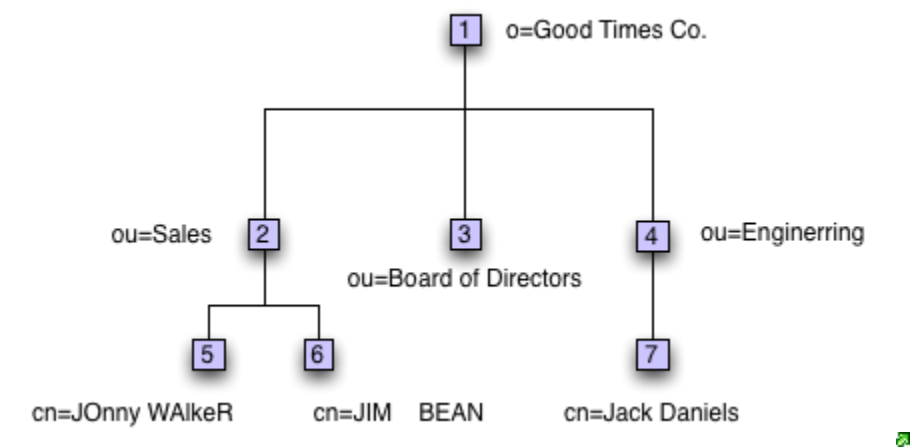
# Structure and Organization

## Introduction

We already covered some of the basic structural aspects in a 2 column Table based partition design in the other sections. It should be clear how indices store Long pointers into the MasterTable to access entries with specific attribute values. Indices prevent the need for full MasterTable scans which would entail a massive volume of expensive lookup, deserialization, normalization and comparison operations.

This document will cover higher level details such as the default system indices used for common operations and how these indices along with optional user indices work together to efficiently search Table based partitions using LDAP filters. Namespace and hierarchy maintenance aspects are also covered.

## System Indices



ParentIdAndRdn	
[o=Good Times Co., 0]	1
[ou=Sales, 1]	2
[ou=Board of Directors, 1]	3
[ou=Engineering, 1]	4
[cn=JOnny WAlkeR, 2]	5
[cn=JIM BEAN, 2]	6
[cn=Jack Daniels, 4]	7

	existence	onelevel	sublevel
[2.5.4.3]	5	1 2	2 5
[2.5.4.3]	6	1 3	2 6
[2.5.4.3]	7	1 4	4 7
[2.5.4.11]	2	2 5	- -
[2.5.4.11]	3	2 6	- -
[2.5.4.11]	4	4 7	- -
-	-	- -	- -

Legend of Object Identifiers

OID	Aliases
2.5.4.3	cn,commonName
2.5.4.11	ou,organizationalUnitName

System indices are required indices. Table based partitions need these indices to properly manage entries in the LDAP namespace, and the entry hierarchy. These indices are the minimum indices required to properly conduct search operations on the part of the directory information base stored in the partition. Just for easy referral we show the example tree here again for index content examples.

## ParentIdAndRdn (called Rdn) Index

This index associates the RDN of each entry with its entry ID in the aster table. It also stores the hierarchical structure of the whole DIT. In order to do that, we keep a track to the parent's ID in the key.

The key is then a composition of the entry's RDN and its parent's ID.

We have a specific case : a context entry (which has a naming context in the rootDSE) is using the full DN and a special marker as the parent ID (note that as we don't know the type of the ID, we will have a specific value depending on the ID type : 0 if the ID is a long, a special UUID if the ID is an UUID, etc)

We don't necessarily store only a RDN in this data structure : when we are dealing with a naming context, which may have a DN with more than one RDN, we treat this DN as if it was a unique RDN.

For instance, if 'dc=example, dc=org' is a naming context, then its key will be <'dc=example, dc=org', 0>.

## One Level (onelevel) Index

The one level index maps parent entry identifiers from the master to the identifiers of their children. This index is used to facilitate various name space operations like renaming, and moving branches which impacts children. It is also used by the search engine to conduct one level search requests.

## Sub Level (sublevel) Index

This index maps ancestor entry identifiers from the master to the identifiers of all their descendants including immediate children. It does not map the descendants of the context entry (at the root) of the partition. This is just unnecessary since all other entries in the partition satisfy this condition. If this is something we desire to enumerate we can get a reverse Cursor on the ndn index and advance past the first entry, to start enumerating all the descendant identifiers of the context entry. Note that this index contains the <id,id> tuple in order to include the added entry itself to the SUBTREE scope search.

This index plays a critical role in subtree search. It allows the search optimizer to detect a count and annotate the modified search filter for subtree scope nodes. It also allows a Cursor to enumerate the set of descendants associated with an entry. Without this index, the search engine must resort to expensive DN based operations for subtree scope constraint checking.



### Other Untapped Advantages

Besides helping with search, the forward LUT of the sublevel index can be used to turn recursive methods for rename and move operations into iterative operations.

Note that the reverse LUT of this index maps entries to all their ancestors minus the context entry. This can be used to quickly walk the lineage of an entry in the tree. This may be useful for certain operations.



Digram needs to be updated to reflect the changes done in StoreUtils.java

## OneLevel/SubLevel index removal

We have created those two indexes back in a time we were using UpDN and NormDn table, instead of RDN table. Now that we are using the RDN index, we don't need those two index.

Let's see with an exemple how we can get rid of those two indexes.

### OneLevel index removal

We don't need the OneLevel index at all, as we can use the Rdn index to get the very same result : the list of all the children for a specific entry. The Rdn index can be browsed to get all the ID for an entry like <parentID, >. **For instance, if we are looking for all the entries under the \*ou=Sales entry**, we will first get this entry ID (ie, 2), and fetch for all the elements where the key is <2, \*>. That will give us back {cn=JONny WAlkeR, cn=JIM BEAN}.

At the end of the day, it's just a matter of being able to process the <parentID, \*> special element.

### SubLevel removal

We can also get rid of this index, and use the ParentIdAndRdn index instead. The algorithm is a bit more complex that the one we use for the OneLevel index, as we will have to recurse on all the entries having at least one child.

In order to ease this computation, we will add two values in the ParentIdAndRdn data structure :

- the nbChildren count
- the nbDescendants count.

The first one counts the number of direct children on each entry (0 means the entry does not have children)  
The second one counts the number of its descendant (its children, the children of its children, ...)

Those two values are updated whenever the DiT is modified (addition, removal or move).

## Existence/Presence Index

The existence index maps the OID of only indexed attributeTypes, to the identifiers in the master for entries containing one or more values for the attribute. All attributeTypes are not tracked. When an index is created on ou it will contain IndexEntry objects for 'ou' in this existence index. Note the OID for attributeTypes is the normalized form for attribute type identifiers.



Remember for our example DiT's partition we indexed objectClass, ou, and cn.

The existence index is used to conduct searches using the existence operator =\*. For example a search with the following filter, (cn=\*), returns entries 5, 6, and 7. The search engine acquires a Cursor and positions it just before the key '2.5.4.11' which is the unambiguous representation of 'cn' or 'commonName'. It then walks (advances one step at a time) the IndexEntry objects for this key, and stops the walk as soon as this key's values are exhausted. At each advance, the entry identifier is used to lookup the entry in the master table and return it.

Notice the existence index does not add the OID for the objectClass attribute. This is specifically avoided since all entries contain the objectClass attribute. There is no reason for this index to bloat by including it.



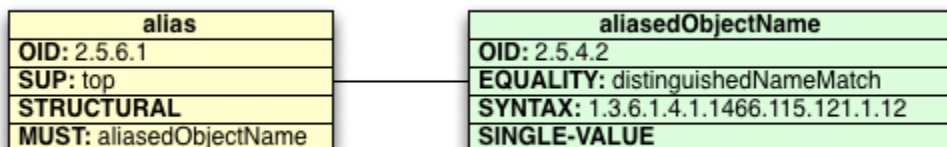
I 'think' the objectClass attribute is not considered for existence indices however this may not be the case. We need to determine this and make sure we do **NOT** add objectClass IndexEntries to the existence index.

## Alias Indices

Unknown macro: {warn}

Those index might be removed.

## Alias Schema Elements



Aliases are entries of the alias objectClass which reference other entries in the DiT. Aliases are not allowed to have subordinate (child) entries. To the left you'll see the schema ER diagram for the alias objectClass and it's mandatory aliasedObjectName attributeType.

## Alias Impact on Search

Alias entries cause cycles in a DiT partition which would otherwise be a perfect tree with the context entry at the root. LDAP search operations are conducted using one of four alias dereferencing modes:

1. ignore aliases while searching, returning alias entries as part of the results
2. dereference aliases while trying to find the search base
3. dereference aliases only while searching (search propagates and continues at the entry referenced)
4. dereference aliases while trying to find the search base, and while searching

Aliases complicate the mechanics of conducting search operations. Searches, without alias dereferencing enabled, contain a clear candidate range for which filter evaluation is applied. Without dereferencing the candidate range is based on the search scope parameter and the search base. Alias dereferencing expands the candidate range into areas of the DiT that would not have been normally eligible for filter evaluation. Candidate determination with dereferencing must factor in the presence of aliases.

Three specialized system indices are used to manage the 3 modes above which allow for some kind of alias dereferencing. These system indices are called the alias, oneAlias, and subAlias indices. We describe what each index contains and how it is used in the subsections below.

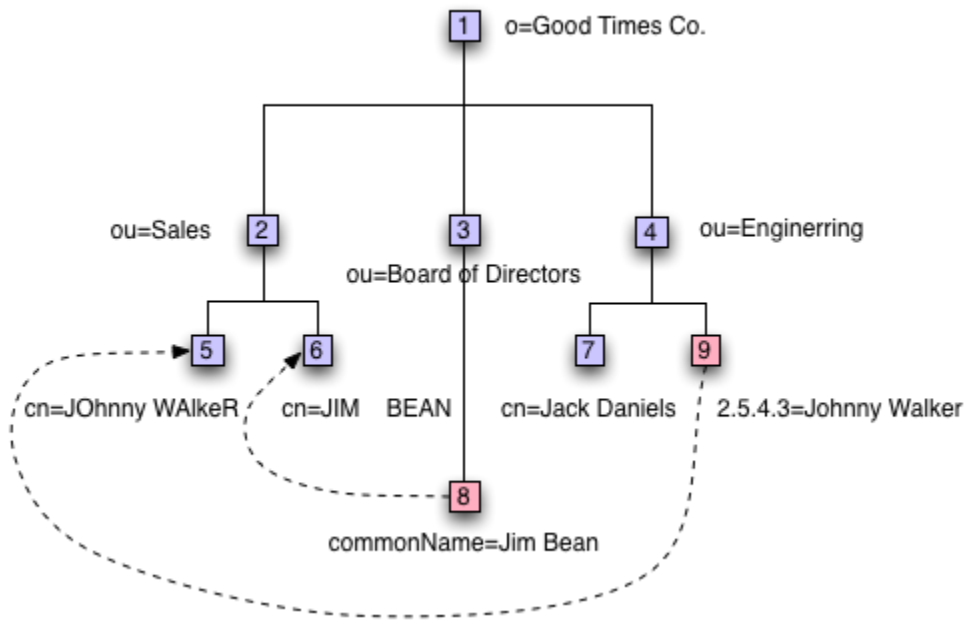
## Example DiT Reloaded (with Aliases)

But first we should add some alias entries to our example DiT to show what the composition of these alias indices would be. On the right is our example DiT with some new alias entries in red. Two new alias entries have been added:

Id	Alias DN	Ref. Id	Referenced Entry DN
8	6	commonName=Jim Bean,ou=Board of Directors,o=Good Times Co.	cn=JIM BEAN,ou=Sales,o=Good Times Co.

9	5	2.5.4.3=Johnny Walker,ou=Engineering,o=Good Times Co.	cn=Johnny WAlkeR,ou=Sales,o=Good Times Co.
---	---	---	--

Follow the link on the image to the right or click [here](#) to view the LDIF for these two newly added alias entries. These LDIF entries have the extensibleObject objectClass set for a value of their objectClass attribute. This marker is a cue to allow these entries to contain values for any defined attribute in the server's schema. It's what allows us to use a commonName attribute for the relative distinguished names of these entries. Also we mix things up here to make a point regarding attribute name variance: the attribute identifier for commonName can be 'cn', 'commonName', or even the OID for this attributeType which is 2.5.4.3.



⚠ Digram needs to be updated to reflect the changes done in StoreUtils.java

## Alias Index (alias)

The 'alias' index is just like a user index on the aliasedObjectName attribute. The normalized distinguished name of the aliased object (the value of aliasedObjectName attribute) is stored in Tuple keys of the forward LUT. The Long identifier of the alias entry is stored in the Tuple value. The alias index is not essential, however it does improve the performance of some operations as an optimization.

To the right, you'll see the composition of the forward LUT for this index. It's straight forward: value of aliasedObjectName normalized maps to the id of the alias entry containing this value.

alias	
[cn=jim bean,ou=sales,o=good times co.]	6
[cn=johnny walker,ou=sales,o=good times co.]	5
-	-
-	-

## One Level Alias Index (oneAlias)

The one level alias index assists in candidacy determination when one level search is enabled with dereferencing while searching. This index contains Long identifiers for both the key and value fields of it's Tuples. For the forward LUT, the Tuple keys represent non-leaf entries with alias child entries **NOT** pointing to their siblings. Child alias entry identifiers are stored in the value of the corresponding Tuple. Child aliases pointing to entries that are siblings do not alter the candidate range. Only those child aliases that point to non-sibling entries expand the candidate range by bringing the referred to entries into scope.

To the right, you'll see the composition of the forward LUT for this index. The first Tuple maps id 3 to id 8. This is because the alias entry 8, expands the scope of one level search when conducted with search base set to entry 3. Entry 8 brings it's referenced entry, entry with id 6, into the scope of the search. Notice the alias entry, the entry with id 8, does not point to a sibling so it is included. The same reasoning holds for the second Tuple in this index's forward LUT.

oneAlias	
3	8
4	9
-	-
-	-
-	-
-	-
-	-

⚠ Digram needs to be updated with the correct key-value ids

### Sub Level Alias Index (subAlias)

The subtree level alias index assists in candidacy determination when subtree level search is enabled with dereferencing while searching. This index also contains Long identifiers for both the key and value fields of it's Tuples. For the forward LUT, the Tuple keys represent non-leaf entries with alias descendants referencing entries that are **NOT** descendants of the key. Descendant alias entry identifiers are stored in the value of the corresponding Tuple. Descendant aliases pointing to entries that are descendants of the key do not alter the candidate range. Only those descendant aliases that point to non-descendant entries expand the candidate range by bringing the subtree of nodes rooted at the referred to entries into scope. Another way to think about this index is that it tracks all the ancestors of alias entries.

To the right, you'll see the composition of the forward LUT for this index. The first Tuple maps id 3 to id 8. This is because the alias entry 8, expands the scope of subtree level search when conducted with search base set to entry 3. Entry 8 brings it's referenced entry, entry with id 6, into the scope of the search. Notice the alias entry, the entry with id 8, does not point to a descendant of the search base, entry with id 3, so it is included. The same reasoning holds for the second Tuple in this index's forward LUT.

Notice though we do not have Tuples for (1, 8) and (1,9). Although alias entries 8 and 9 are descendants of entry 1, they do not expand the search scope with subtree search operations using entry 1 as the base. All these aliases and the entries they point to were reachable as candidates before creating alias entries 8 and 9. Only alias entries pointing out of the subtree of the base tow entries that are not search base descendants are tracked by this index.

subAlias	
3	8
4	9
-	-
-	-
-	-
-	-
-	-
-	-
-	-
-	-
-	-
-	-
-	-
-	-

⚠ Digram needs to be updated with the correct key-value ids

### Handling Dereferencing with Object Level Scope

Object level search requests are trivially handled. Only two of the dereferencing modes matter to us in this case.

Mode	Considered	Reason
neverDerefAliases	no	this is a normal search and dereferencing does not matter
derefFindingBase	yes	if the base is an alias we must dereference it before filter evaluation
derefSearching	no	with object level search we're not searching
derefAlways	yes	same as derefFindingBase

So if alias dereferencing is in effect always or when finding the base a check must be performed to see if the search base entry is an alias. If it is an alias, the referenced entry is resolved, and the search base is set to the referenced entry. In this simple case with object level scope, alias dereferencing does not expand the single candidate. Instead alias dereferencing switches the base to the referenced entry.

### Handling Dereferencing with One Level Scope

One level search requests apply the same technique of switching the base when base dereferencing is enabled. Actually all search scopes switch the search base when base dereferencing is enabled, and the original search base is in fact an alias.

The search engine uses this index to find aliases which expand the range of candidates to entries outside the scope. After the base entry is resolved, the search engine, uses the identifier of the base to lookup all alias children. The referred to entries are then included in search filter evaluation and returned if accepted by the filter. This way the candidate range expands to include referenced entries in the `aliasedObjectName` attribute.



When alias dereferencing while searching is enabled, `ScopeNodes`, in the filter AST, should not be used as candidate enumerators. In fact the search engine's optimizer should annotate the scope node with a maximum count value when dereferencing while searching is enabled. I do not know if this is the case today.

For our example, let's do a search where the search base is entry with id 4. The scope is one level and `derefSearching` is enabled. The search engine positions a Cursor over the children of entry 4 using the onelevel hierarchy index. It joins the results of this Cursor with the results from another Cursor on the oneAlias index over forward LUT key 4. This includes entries 6, 7 and 9 as candidates. Entry 9 however is filtered out of the result set when `derefSearching` is enabled. It is not returned since it is an alias.

## Handling Dereferencing with Subtree Level Scope

Dereferencing while searching with subtree scope is rather painful. After dereferencing the base, if needed and `derefFindingBase` is enabled, the search engine looks up all alias descendants in the `subAlias` index. Alias descendants are then collected, and put into a special `ScopeAssertion` which uses this list and search scope parameters to determine if an entry is an eligible candidate for filter evaluation.

The semantics of search with `derefSearching` enabled presumes search propagation into dereferenced subtrees. This means a propagated search may encounter new alias entries which it must dereference as well.

You'll see in more advanced sections of this guide that assertion and candidate enumerating Cursors are used by the search engine. One assertion is the scope assertion. For search without alias dereferencing while searching, this scope assertion just makes sure the candidate entries are in scope by checking if they are subordinate to (with on level search) or descendants of the search base (with subtree level search).

When `derefSearching` is enabled, the search engine modifies how this scope assertion determines if a candidate is within scope. The algorithm is simple and it uses the `subAlias` index with subtree scope search:

1. Use the provided search base parameter to find all alias entries that are descendants of the base and point to entries outside the scope to entries that are not descendants of the search base. To do so we get a Cursor on the forward LUT of the `subAlias` index over Tuples with the key of the search base.
2. For each value of the Tuples returned, repeat the process recursively by presuming the alias entry is now the search base. Stop recursing when no scope expanding alias descendants are found. Do this while collecting the complete set of aliases found.
3. Use the set of aliases and the search base in the scope assertion: the assertion returns true if any candidate entry is a descendant of any of these aliases or the search base. Otherwise it returns false.

By modifying the behavior of the scope assertion to factor in scope expanding aliases in the scope constraint check we effectively appear to propagate search through aliases.



### Optimizations

Sometimes aliases will be return in this set of aliases to consider in scope assertions, which are descendants of other aliases already in this set. With subtree searching the search will automatically hit these descendant aliases so there's no need to have them in the set. Hence we can prune this set, or perform checks before insertion to prevent including redundant aliases in this set.

We will cover more of this while looking closer at the search engine in different sections of this documentation.

## Alias Limitations

ApacheDS places some limitations on Aliases. The limitations below are imposed either for the sake of simplicity in conducting search or due to structural limitations.

1. Alias chaining where one alias refers to another alias is not allowed.
2. Aliases can only reference entries within the same Partition. This limitation is not by choice and due to the inability to share access to indices across partitions which may change in the future.

## objectClass Index

For the time being, the `objectClass` index is considered a system index. This is purposefully done since `objectClass` values are critical for server operation and used heavily. Without setting this index performance would be terrible.

## User Indices

The partition can be asked to maintain an index on any attribute defined in the servers schema. These indices contain Tuples in their forward LUT, mapping a value of the attribute, to the id of the entry with that attribute value. This way after consulting the index, entries with specific values for this attribute can be quickly looked up by identifier in the master table.

## Other Useful Indices

The function of several optional ApacheDS features may be greatly enhanced by the presence of indices on certain attributes. One example is the revision attribute which has not yet been defined, but it will. This attribute will be used by the change log facility to track the last altering revision on an entry. When this system and its related snapshotting capabilities are enabled, having an index on this attribute will be very valuable.

Other similar examples exist. You'll find for different usecases, and for different kinds of canned search distributes, indices on some attributes will greatly improve search performance. More about this is covered on the search engine section of this guide.