OpenLDAP Development

Back-hdb – Hierarchical Backend

Howard Chu March 21, 2003

hyc@symas.com

Motivation

- Support ModDN/Subtree Rename
- Avoid the Quadratic growth in DN2ID
- Enhance update performance
- Purist – it's a hierarchical namespace, after all!

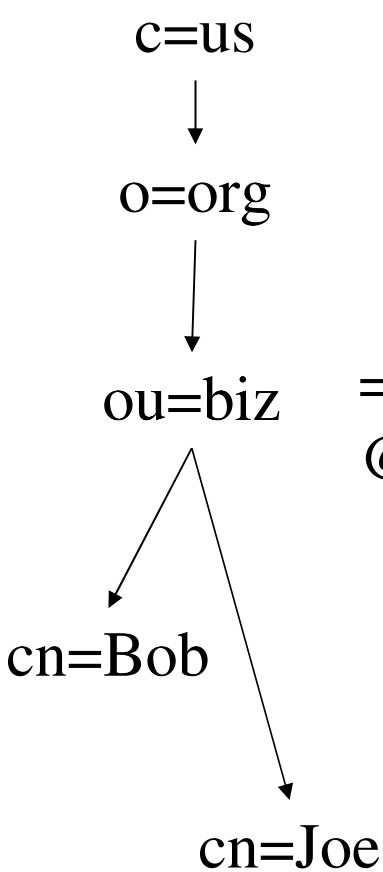
Review of back-bdb DN2ID

- Uses same DN2ID design as back-ldbm
- Can locate any DN with only 1 DB call – good for fast searching
- Maintains explicit list of onelevel and subtree IDs per entry – also good for fast searching

Back-bdb DN2ID drawbacks

- Slow for updates, some kinds of updates are impossible/impractical (e.g. subtree rename)
  - Costs a minimum of 2 DB updates plus 1 per level of DN depth to Add/Delete entries
  - Stores excessive redundant DN information
  - Paradoxically, the deeper/more organized the tree, the worse it performs
- Historically, people speak of LDAP as “good for reads, bad for writes” – this is largely due to the DN2ID design, not the LDAP/X.500 specs

Back-bdb DN2ID example



=c=us:1; %c=us:2; @c=us: 1,2,3,4,5

=o=org,c=us:2; %o=org,c=us:3;
@o=org,c=us: 2,3,4,5

=ou=biz,o=org,c=us:3; %ou=biz,o=org,c=us: 4,5;
@ou=biz,o=org,c=us: 3,4,5

=cn=bob,ou=biz,o=org,c=us: 4;
@cn=bob,ou=biz,o=org,c=us: 4

=cn=joe,ou=biz,o=org,c=us: 5;
@cn=joe,ou=biz,o=org,c=us: 5

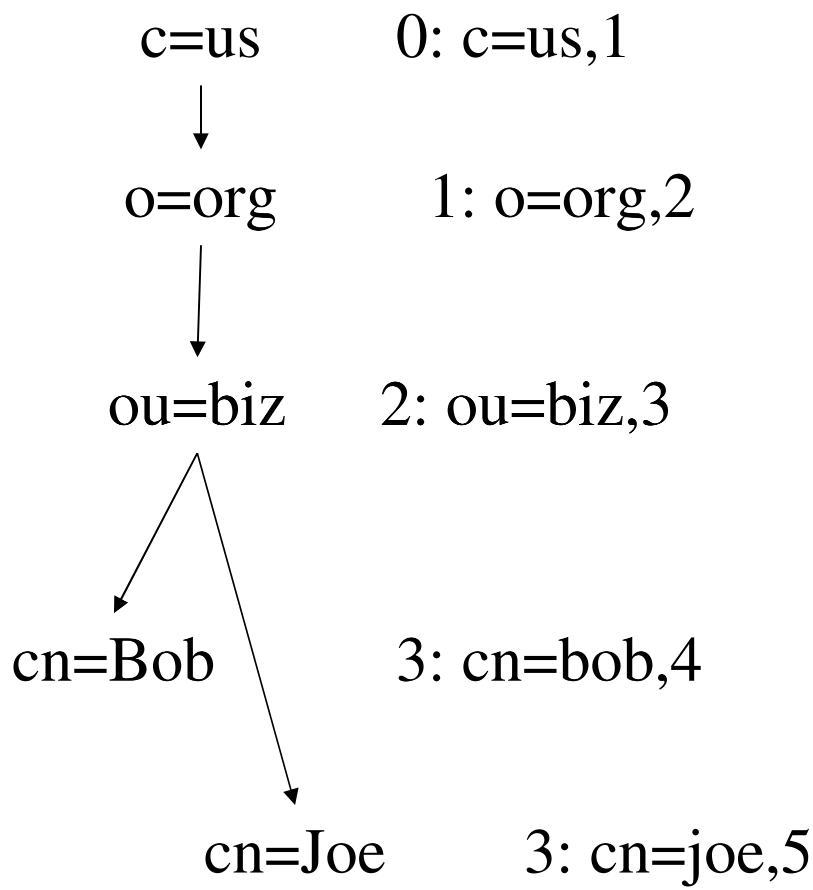
Back-bdb DN2ID example (2)

- 5 nodes in DIT
- 13 keys in database
- 23 data items in these 13 keys
- Keys are long, consume more DB pages
- It only gets worse from here…

Back-hdb Principles

- Only store RDNs and parent references
  - Eliminates redundant DN storage
  - Allows subtree rename
  - Performs Add/Delete/ModRDN with only 1 DB update – O(constant) instead of O(n) efficiency.
- Sacrifices search performance?
  - Requires Depth(DN) DB searches to locate a base DN
  - No subtree IDLs, requires recursive DB searches

Back-hdb DN2parent example



Back-hdb DN2parent (2)

- 5 nodes in DIT
- 4 keys in database
- 5 data items in these 4 keys
- Keys are short, DB remains small

Back-hdb “Sacrifices?”

- Back-bdb DN2ID search is always O(log(N)).
  - Efficiency is guaranteed by Btree balancing
  - But each compare is over a full DN – expensive
  - N is large, due to subtree/onelevel IDLs
- Back-hdb best case is O(log(N)).
  - Efficiency not guaranteed, poor DIT layout will have negative effects
  - But each compare is only over an RDN – very cheap
  - N is small, no redundant IDLs cluttering things up

Back-hdb "Sacrifices?" (2)

- Back-bdb DN2ID subtree IDL speeds subtree searching?
  - Only for small trees. Beyond the fixed IDL size, the subtree IDL does more harm than good, bringing in false candidates
- Back-hdb subtree recursion is expensive?
  - Never brings in false candidates – makes search evaluation more efficient

Test Results

- Back-hdb & back-bdb search performance tests out to nearly identical, with a 2% advantage to back-hdb. Entry caching has leveled the field here, but back-hdb's smaller footprint still gives it an edge.
- Back-hdb Add/Delete performance relative to back-bdb depends on database size; even on small DBs 10% gains are noticeable. Overshadowed by attribute indexing.

Conclusions

- Back-hdb is faster for both reads and writes, even without entry caching.
- Minus attribute indexing, back-hdb performs DIT updates in constant time – disproving the myth that LDAP must be slow for writes.
- This is the most viable approach for really large scaling.