Write-intensive applications with LDAP
Write-intensive applications with LDAP

Cuong Bui

22nd September 2004

Under supervision of Ling Feng, Djoerd Hiemstra, Rick van Rein
Database group, Department of Computer Science, University of Twente

Abstract

LDAP (lightweight directory access protocol) is a protocol specification that can be used to access directories. OpenLDAP is an open source LDAP service implementation. By design LDAP is optimized for read (lookup) operations. The LDAP specification does allow write operations. Performance measurements have been conducted by [1] to measure the read performance of OpenLDAP. However, there is little known about the write-intensive performance of OpenLDAP. Applications using LDAP commonly have a read-intensive characteristic and research is concentrated at this aspect. This thesis will use a write-intensive application case provided by OpenFortress (workflow reseller system) to conduct measurements to ensure whether OpenLDAP can support write-intensive applications in general. During the performance measurements a performance problem has been discovered. The performance of OpenLDAP decreases significantly after a number of operations. Both read and write operations generate this problem. The least recently used (LRU) cache replacement policy has been identified as the possible cause of the problem. Several replacements policies have been studied and adaptive replacement cache (ARC) has been chosen to replace the LRU implementation in OpenLDAP because ARC is more efficient with smaller cache sizes and can withstand certain cache flushes. OpenLDAP with the ARC implementation shows a significant decrease of cache misses (especially with smaller cache sizes). Cache misses result in disk access and therefore it is preferred to minimize this number. LRU is the most common cache replacement policy used at the time of writing of this thesis, ARC has been shown to be a good substitute for LRU. ARC manages limited resources more efficiently than LRU. The performance degradation issue was eventually tracked down to a bug in the Berkeley database code used by OpenLDAP. With this problem having been fixed, there’s no objection (performance wise) to use OpenLDAP with write-intensive applications.
Contents

1 Introduction 5
  1.1 Problem description ....................................... 6
    1.1.1 The problem ........................................ 6
    1.1.2 Related work ......................................... 6
    1.1.3 Thesis contribution .................................. 6
  1.2 What is LDAP ............................................... 6
  1.3 What is OpenLDAP .......................................... 7
  1.4 Data stored in OpenLDAP ................................ 7
  1.5 OpenLDAP back-ends ...................................... 9

2 Choosing LDAP over a relational database 11
  2.1 Type of application ...................................... 11
  2.2 Lightweight ............................................... 12
  2.3 Access control ............................................ 12
  2.4 Authentication ........................................... 13
  2.5 Flexibility ................................................ 14
  2.6 Storage of data ........................................... 14
  2.7 Data safety ................................................. 15
  2.8 Transactions ............................................... 15
  2.9 Write ahead logging ...................................... 17
  2.10 Scalability and reliability .............................. 17
  2.11 Maintenance .............................................. 17
  2.12 Extending server functionality ......................... 18
  2.13 Documentation ........................................... 18
  2.14 Summary .................................................. 18

3 Approach to test OpenLDAP 20
  3.1 Test setup ................................................ 21
  3.2 Hardware and software used ............................... 21
    3.2.1 Server ............................................... 22
    3.2.2 Client ................................................ 22
  3.3 Test results ............................................... 23
    3.3.1 OpenLDAP configuration ............................... 23
    3.3.2 Evaluation of preliminary results .................. 25

4 Proposed solutions 27
  4.1 OpenLDAP query processing ............................... 27
  4.2 Add back-off mechanism .................................. 28
  4.3 Buffer management ....................................... 29
    4.3.1 Access patterns ...................................... 30
    4.3.2 LRU: Least Recently Used ............................ 30
    4.3.3 LRU-K/LRU-2 ....................................... 32
    4.3.4 2Q .................................................. 32
    4.3.5 MQ .................................................. 34
CONTENTS

4.3.6 ARC ................................................. 34
4.4 Choosing a cache replacement policy for OpenLDAP ............ 36
  4.4.1 Experience with substitution of LRU with ARC in PostgreSQL ..................................... 37
4.5 Substituting LRU with ARC in OpenLDAP ......................... 39
  4.5.1 LRU in OpenLDAP ........................................ 39
  4.5.2 Modify LRU structure to support ARC in OpenLDAP ....... 40
  4.5.3 Modify code to support ARC in OpenLDAP ................. 41
  4.5.4 Problems encountered during implementation .......... 42
4.6 Performance test ........................................... 43
  4.6.1 Test setup ............................................ 43
  4.6.2 Definition of recorded data ................................ 44
  4.6.3 Patterns ................................................... 45
  4.6.4 Pattern motivation ........................................ 46
4.7 Statistical analysis of the data .................................. 47
  4.7.1 Overview of the used theory .................................. 47
  4.7.2 Analysis of the data .................................... 48
4.8 Evaluation of results ......................................... 50
  4.8.1 Optimal concurrent connections .............................. 51
  4.8.2 Evaluation of pattern S1 ................................... 52
  4.8.3 Evaluation of pattern S2 ................................... 56
  4.8.4 Evaluation of pattern S3 ................................... 58
  4.8.5 Evaluation of pattern L1 ................................... 62
  4.8.6 Evaluation of pattern C1 ................................... 64
  4.8.7 Evaluation summary ....................................... 67
5 Evaluation of OpenLDAP documentation .......................... 68
  5.1 User documentation .......................................... 68
  5.2 Developer documentation ..................................... 69
  5.3 Summary ...................................................... 70
6 Conclusions en recommendations ................................. 71
1 Introduction

The Lightweight Directory Access Protocol (LDAP) is being used for a variety of applications. The nature of these applications tend to be read-intensive. However this is not a restriction of the LDAP specification [4]. Although LDAP often is optimized for read operations, it is capable of handling write operations as well.

OpenFortress, a technology provider specialized in applications of digital signatures, has implemented a workflow reseller system that stores its information in OpenLDAP [3]. Workflow systems often require more write operations than typical LDAP applications and are usually implemented on top of a Relational Database Management System (RDBMS). RDBMS are optimized for reading, writing, concurrency and they often have mechanism for distributing load, which makes them highly scalable. A RDBMS is often chosen for such system because of these factors. OpenFortress has chosen to deploy OpenLDAP because OpenLDAP is lightweight, can store complicated data structures, has fast data retrieval and replication.

This thesis will use a case provided by OpenFortress to ensure whether OpenLDAP is suitable for a write-intensive application such as a workflow reseller system.

Section one contains the problem description and a global introduction. In section two a motivation is given to choose OpenLDAP over a relational database followed by section three and four containing the test approach, test setup and the results. Section five will briefly explore the documentation of OpenLDAP. The conclusion is located in section six.
1.1 Problem description

1.1.1 The problem
Write-intensive applications have different requirements than read-intensive applications. It is reasonable to assume that write-intensive applications will access the disk for a longer period of time with exclusive locking than read-intensive applications. Applications, which are optimized for reading such as OpenLDAP are designed for fast read actions and write-intensive applications may fail to achieve a reasonable speed with that design. This thesis will explore the possibilities to deploy write-intensive applications with OpenLDAP which is by design optimized for read actions.

The main focus will be on performance measurements with read and write-intensive applications. Typical performance simulations are being created to represent the OpenFortress reseller system (a write-intensive application). Several other aspects such as maintenance and documentation will also be explored. These parts are also essential to deploy any kind (read or write-intensive) of applications.

1.1.2 Related work
Performance with read-intensive applications using OpenLDAP has been measured and the results are presented in [1]. Improvements in this area have been proposed and accepted in the development version of OpenLDAP. Examples of such an improvement is a proxy cache mechanism to cache queries. Caching queries yields a reduced client latency and a better scalability (shown in [18]). Deploying caches can reduce frequent disk access. The commonly used replacement policy is Least Recently Used (LRU, explained in subsection 4.3.2). This algorithm is relatively easy but it have some drawbacks. Solutions for these drawbacks have been presented in [13, 10, 14, 12, 11].

1.1.3 Thesis contribution
This thesis will explore practical implementation of write-intensive applications with OpenLDAP. OpenLDAP uses the cache system extensively and an improvement in this part of the system will improve the OpenLDAP performance. The OpenLDAP LRU implementation will be substituted with ARC and the results will be studied. Also this thesis will try to motivate why in some cases OpenLDAP is a better solution than a traditional relational database.

1.2 What is LDAP
The Directory Access Protocol (DAP) is being used to access X.500 directories. DAP is a complex protocol to use and to implement, therefore an easier protocol was specified with most of DAP functionality but with less of the complexity. LDAP is the name for this less complex protocol. At the time of writing of this
thesis, there have been three revisions [8, 9, 4] of LDAP. The LDAP definitions specifies how one can access a specialized database (a directory).

A directory can be compared by a phone book. One can lookup information. For example a phone book is being used to lookup phone numbers or addresses using a key (such as the last name). A phone book is printed once over a period of time, but lookups of phone numbers occur frequently. A common usage of a directory is storage of centralized user account information. Account information can consist of user name, password and other credentials. Read access (for example a login action of a user) to this directory occurs frequently compared to write access (for example addition of a new user). Generally more reading than writing is being done on a directory.

1.3 What is OpenLDAP

OpenLDAP [3] is an open source implementation of the LDAP specification and the source code freely distributed under the OpenLDAP public license which has similarities with the BSD style license. One can use this software freely and make modifications to it.

The OpenLDAP suite consists of:

- slapd - stand-alone LDAP server
- slurpd - stand-alone LDAP replication server
- libraries implementing the standardized Application Programmers Interface (API) and utilities for managing the OpenLDAP environment.

The slapd daemon is the actual LDAP server. This server consist a front-end that handles incoming connections and a back-end which manages the actual storage and retrieval of the data.

The slurpd daemon (a service program) is used to propagate updates from one slapd to another slapd daemon. Slapd must be configured to maintain the replication log, which will be used for replication. A slapd daemon can send updates to one or more slapd slaves. OpenLDAP uses a single master / multiple slaves model for replication. Also temporary inconsistency between replica are allowed, as long as they are synchronized eventually.

To complete the suite a set of utilities and libraries is also provided. Developers can use the libraries to create own software applications which can interact with OpenLDAP. The set of utilities can be used to perform maintenance tasks such as backup and restoring data. There are also command line utilities for adding entries, searching entries and modifying entries.

1.4 Data stored in OpenLDAP

Data stored in OpenLDAP is based on an entry-based information model. An entry is a collection of attributes uniquely globally identified by its Distinguished Name (DN). Attributes are typed and can have one or more values. Types are
defined as strings (sequence of characters) such as \textit{cn} for Common Name or \textit{ou} for Organizational Unit. Values are dependant of their types. For example an \textit{ou} can contain the string “Helpdesk” where as a \textit{jpegPhoto} can contain binary data.

The data organization is being represented as a hierarchical tree-like structure. The DN for R van Rein in figure 1 is “\textit{cn=R van Rein, ou=helpdesk, o=OpenFortress, C=NL }”. The DN for J doe is “\textit{cn=J Doe, ou=helpdesk, o=Company, st=California, C=US }”. Notice the inconsistency between a help desk person located in the Netherlands and one in the USA. The state is missing in the Dutch DN.

![Figure 1: Example data representation](image)

The geographical location is commonly located on top of the tree, followed by state, organization, organizational unit and common name in this example. The DN consists of the nodes in the tree. The DN is associated with a number, which will be used to access a certain record.
1.5 OpenLDAP back-ends

As stated before OpenLDAP can store its data in several back-ends. Each has its advantages and disadvantages. This thesis will only use the two commonly used back-ends. These are:

- Back-LDBM (using Berkeley DB version 3)
- Back-BDB (using Berkeley DB version 4)

LDBM is the default back-end for OpenLDAP 2.0.x and uses Berkeley DB version 3.x for storage and retrieval. BDB is the default back-end for OpenLDAP 2.1.x and uses Berkeley DB version 4.x for storage and retrieval of data. LDBM is a generic Application Programmers Interface (API) that can be plugged on top of several other storage engines such as GDBM, MDBM, Berkeley DB (version 3 or 4). Back-bdb uses the full BDB 4 API. Table 1 shows the main difference between back-bdb (bdb v4) and back-lbmb (bdb v3).

<table>
<thead>
<tr>
<th>Features</th>
<th>back-lbmb</th>
<th>back-bdb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record lock</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Write ahead logging</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Transaction support</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>On-line backup</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Two phase locking</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1: Difference between back-lbmb and back-bdb

Back-lbmb has no fine-grained mechanism to perform record locking. When locking is required, back-lbmb will lock the entire database. This behavior is not desirable for an application with high concurrency requirements. Back-bdb stores one record in a page. BDB supports storage of multiple records on a page but OpenLDAP has chosen to store only one record in a page. BDB v4 can only lock pages. By storing one record in a page at a time OpenLDAP 2.1 using back-bdb can support single record locking.

Write Ahead Logging (WAL) is a mechanism to recover the data to a consistent state from crashes or power outage. Modifications are first written to a log file before the actual modification is committed. If a disaster does occur, one can use the WAL log to recover to a consistent state of the data by replaying the log file. WAL feature is desirable for applications processing data, which may not be corrupted.

Transactions permit groups of operations (for example changes) to appear at once. RDBMS can update a set of records using transactions. Transactions are ACID compliant. ACID stands for atomicity, consistency, isolation, and durability. A set of operations in a transaction happens all at once or not at all. This is the atomicity property. When a transaction starts or ends it leaves the system in as consistent state, fulfilling the consistency property. Transactions
are also isolated, meaning no other transactions or operations can interfere with a transaction once it has started. The durability property requires the system to maintain its changes when a transaction completes, even if the system crashes. The back-bdb supports transactions. LDAP can use the transactional back-end to ensure data safety of the stored data.

The features offered by back-bdb are valuable to applications handling important data. Back-bdb can recover from crashes and guarantees consistent data. This is essential to administrative applications such as a reseller workflow system. These features are coming with a certain overhead. The WAL and transaction support requires more system resources (slow disk IO, disk space needed for log file storage).

Administrative systems may need to run 24 hours a day. They may never be shutdown. Back-ldbm cannot support on-line backup because it doesn’t support record locks. When a backup is being performed the backup application has to read the database files. Because slapd is running (it will access the database file, and thus it will lock the file for other applications), the backup application cannot read the database files and therefore cannot perform on-line/hot backups (An on-line/hot backup is a consistent backup performed while the system is operating). This behavior is undesirable. This problem does not exist with OpenLDAP with the back-bdb back-end.

Two phase locking is used in conjunction with transactions. A transaction is divided into two phases. During the first phase, the transaction is only allowed to acquire a lock. During the second phase the transaction is only allowed to release a lock. This implies once a transaction releases a lock it cannot acquire another lock.
2 Choosing LDAP over a relational database

The workflow reseller system can be classified as an administrative system where stored data is important and may not be lost. This administrative system has a high percentage of write operations compared to common use of LDAP systems. The common LDAP systems can be classified as read-intensive application (e.g. a phone book where one can lookup information relatively fast and where changes don’t appear frequently). An important question is why one would choose to implement such a system using LDAP instead of a relational database. The motivation for the choice has been based on several aspects. The aspects will be discussed for OpenLDAP and for a relational database. Before each aspect is discussed, specific order information shall be given. An order exists of two parts. Figure 2 is the decomposition of an order.

![Figure 2: Entity relation diagram of the order part](image)

The entity relation diagram (ERD) shows an order can have multiple order items. An order item is part of an order. An order can have fields like dates and order numbers. Order items can have an amount of purchased product and the description for the product itself. This is a highly simplified representation of the data. Orders can are to resellers and have a certain workflow status. This simplified representation will be used to motivate the choice between OpenLDAP and a relational database. The motivation consist of several aspects which are discussed in the following sections.

2.1 Type of application

Between reads and writes there are differences. Because of these differences an application with a high number of read actions compared to the number of write actions (classified as read-intensive) have different requirements than an application with a high number of write actions compared to the number of read actions (classified as write-intensive). OpenLDAP is optimized for read actions as explained in subsection 1.2. This is a common requirement of applications using OpenLDAP. This is not an explicit requirement for the OpenLDAP server, but OpenLDAP is optimized for read actions because it has to support these common applications. Applications with many concurrent write operations can cause problems with OpenLDAP. A relational database can be tuned for a certain kind of application (read-intensive, write-intensive or a combination of those). With regard to this aspect a relational database has an advantage (i.e. proven to work with read-intensive/write-intensive applications). The goal of this thesis is to enhance OpenLDAP in this respect.
2.2 Lightweight

OpenLDAP is a lightweight implementation of the LDAP specification (part of the DAP specification). OpenLDAP is smaller and less complex compared to a database (such as PostgreSQL [7]). A relational database does have more overhead. It has logic to process procedural languages, manage multiple types of indexes (such as full text index and clustered index), load balancing functionality (fail over, fail safe, replication), manage triggers, support foreign key enforcement and so on. This list of extra functionality, which is not in LDAP is not complete. This 'overhead' (it’s called overhead here, but it’s an important part of a relational database) is not all present in OpenLDAP. Having less overhead helps to keep the application relatively simple and requires less resources (in term of memory footprint and disk space). Transactions are a major cause of overhead and complexity for the purposes of the workflow reseller system.

2.3 Access control

The OpenLDAP hierarchical storage structure can allow an elegant representation of data. Figure 3 depicts a decomposition of storage of order information from several resellers.

 LDAP can store order information from a certain reseller in a sub-tree. Access to this sub-tree can be restricted to this reseller. Other resellers cannot
2.4 Authentication

A reseller can store multiple orders and an order has one or more order items. With a relational database two tables are required to represent this order structure. This first table is a table where global order information is stored (such as issue date and payment). The second table stores order items (what has been ordered for an order). These two tables are linked together. Using this construction it’s hard to enforce access at database level to certain orders for a particular reseller. Enforcement at the application level is possible but is not safe. Access cannot be regulated at row level in a relational database (such as PostgreSQL). Any reseller with access to the order table will be able to access all the rows and therefore he or she is able to access information from other resellers. A database view can be defined to overcome this problem at application level. The user can always log on to the database manually using his or her credentials and access/modify the records with SQL queries.

Another solution with a relational database is to store each order information for a certain reseller in a distinct database. Access can now be granted to certain reseller for a certain database. However this solution is not practical. If a reseller is added, an extra database has to be created. The number of databases grows instead of the number of records.

With stored procedures it is possible to solve this problem. One can use a stored procedure, which will return a data set (order items) for a certain reseller. An user table is also required where reseller credentials are stored. The stored procedure can use this user table to determine what kind of records needs to be returned for a user. These solutions will only work if the database has extensive user rights management. The stored procedure needs complete access to the order tables, but the user (reseller) who is calling/invoking the stored procedure may not have access to these tables. Not all databases support this and it is therefore no common solution for all databases.

2.4 Authentication

OpenLDAP offers two methods of authentication. The first option is the “simple” method. With this method a user can authenticate with a name and password. It is recommended to use this method in a controlled environment because the user name and password are sent plain over the network. Anonymous access is also possible. The second option is the “Simple Authentication and Security Layer” (SASL) method, which is a standardized option for all LDAP implementations [26]. SASL provides more authentication options than the “simple” method. SASL also offers plain login method such as the simple method. This method should again only be used in a controlled environment. It is possible to use this login method with a Transport Layer Security (TLS). Other authentication methods specified by SASL such as Kerberos v4, DIGEST-MD5 and Generic Security Services Application Programming Interface (GSSAPI, see [27] for more detailed information) specified by the SASL offer more flexibility and security. They should preferably be used. OpenLDAP offers more standardized authentication options than a relational database which is an advantage over a
OpenLDAP has advantages over a relational database with respect to the data structures. There are more pre-defined types in OpenLDAP (such as jpegPhoto) that are not present in relational databases. Having more pre-defined types help to prevent one from creating their own types. However, there are some relational databases (such as [7]) who can define their own data types. This behavior is however not standard and is different for each database, this will make exchange of data and schemes (with user defined types) difficult among databases.

The OpenLDAP scheme can also be more flexible than a relational database scheme. The tree structures are more easily dividable given user more flexibility. A user can for example host different sub-trees on different servers. Also sub-trees can be assigned to a certain person (such as a reseller).

The user rights management with OpenLDAP is also more flexible than the user rights management of a relational database. Access can be restricted at tree/record level whereas a relational database only has rights management for tables.

With OpenLDAP there is also more control on how certain data is stored. Blobs (binary large objects) can grouped (stored) together, this will make some tasks (maintenance) easier. Relational databases can have table spaces. A table space is a mechanism, where one can have more control on how a database will store the data. Indexes for example can be stored on fast drives and 'normal' data can be stored on slower drives. Indexes are more often accessed than 'normal' data and will benefit from the storage on the fast drives. Table spaces do give a relational database more flexibility on how data will be stored, but it is again not standard and not all databases support this feature.

2.6 Storage of data

Both OpenLDAP and relational databases are implemented on top of a file storage system. However a relational database can have its own storage manager with raw partitions (bypassing the Operating systems storage system to provide more custom tailored functionality). Figure 4 illustrates the level of storage of both OpenLDAP and a relational database. The storage manager is part of the RDMS, but it’s displayed as a distinct layer to illustrate the similarities with OpenLDAP.
OpenLDAP also uses a storage manager to manage its data. A possible storage manager is supplied by the Berkeley DB layer. The difference between OpenLDAP and a relational database is the extra functionality that is supplied by the relational database. Examples of this extra functionality are stored procedures and foreign keys constraints, which are not present in OpenLDAP. Having this extra functionality is an advantage but also makes the relational database system larger and more complex than OpenLDAP.

2.7 Data safety

OpenLDAP can use several back-ends to store the data. All Berkeley DB versions with version numbers less than 4.2.x are not ACID compliant. A non-ACID compliant back-end has several disadvantages as was described in subsection 1.5. Non ACID systems cannot guarantee the safeness of the data. A power failure or a crash might corrupt the data and there might be no way to recover from it (beside from using a backup). Order / workflow information is the kind of information that can change often and where a loss of a single record can have a financial consequence. For example if some reseller ordered 5000 items and the system crashes, the valuable information lost by the system causes a potentially large amount of money. The data can also be corrupted and all order information since the last backup could be lost. A system with ACID capabilities might have prevented this disaster. And if disaster happens (for example power failure over a long period of time) ACID compliant systems guarantee they can recover the data in a consistent state (This guarantee is more or less theoretical, there are situations where even an ACID compliant system may not recover to a consistent state. Such example situation is a physically damaged hard disk). LDAP and a relational database can also replicate data to a slave to ensure a higher degree of data safety.

2.8 Transactions

Transactions are, as discussed in subsection 1.5, ACID compliant. With transactions the system can enforce a successive execution of a sequence of operations. All operations are either successful or none of them are. Such a property is
required for example with the insertion of order information. Order information consist of one global order information block and one more items, which have been ordered for that order. If the system doesn’t support transactions a crash or power failure during the insertion of an order can result in loss of information as shown in Figure 5.

![Diagram of order insertion with failure](image)

**Figure 5: Example insert of an order with items**

First the global order is inserted followed by two orderline items. Before item z can be inserted or during the insertion of item z, a system failure occurs. Non-transactional systems may now have lost item z. The system only has the order information with item x and item y (and is not correct, item z is missing). If this system was transactional all of the insert operations are aborted and the data remains consistent. The order information is not stored in this case. If the system also uses WAL (explained in subsection 2.9) the system can replay the log and still insert the order with item x, y and z. With a non transactional system one has to manually remove the inserted order and order items. A relational database such as PostgreSQL is transactional and ACID compliant. OpenLDAP with BDB greater than 4.1.x can also be transactional and ACID compliant at object level. There still will be a potential loss of data as illustrated by figure 5. OpenFortress solves this problem by storing global and order items in one object.

The transaction support only guarantees safety per order object or order item object. The transactions used by BDB are small transactions (i.e. store (key, value) is one transaction). These transactions can be used to build a larger transaction monitor/manager to solve this potential problem (this is demonstrated by the MySQL database, which can use BDB as a transactional back-end).
2.9 Write ahead logging

WAL allows data recovery in case of a system failure. A history of operations is replayed to recover data. This is possible because WAL writes a log entry before it actually executes a certain operation. WAL can be used to implement the Durability property of ACID. Both OpenLDAP with BDB greater than 4.0.x and a relational database has WAL.

2.10 Scalability and reliability

OpenLDAP is lightweight but it has a mechanism to scale. For this purpose OpenLDAP has a replication mechanism. The content of one OpenLDAP server can be distributed (replicated) to multiple servers. A reseller can even run their own sub tree on their own server. The replication mechanism isn’t transparent as with a relational database where mechanism like fail over and fail safe are implemented. Using this mechanism one can deploy several database servers as one database server (database cluster). If one of the database servers fails, the other servers will take the tasks of the failed server and process them. If the loads to the system is high, a cluster of databases can divide workload to maintain acceptable performance. OpenLDAP doesn’t have this mechanism. It only has replication and backups. OpenLDAP does have a similar structure as the Domain Name System (DNS). Partial trees can be stored on different locations. One could use this to spread the load of a tree if the load gets too high. This particular workflow system will not require this functionality at this point. If the system will get large and heavily used scalability can be solved by spreading sub trees to multiple locations.

2.11 Maintenance

Server maintenance consists of several tasks. Only the backup procedure will be discussed in detail for both back-ldbm and back-bdb. In order to perform a backup of a running OpenLDAP directory the server has to be shutdown with back-ldbm. Only one process can access the database files at a time. The OpenLDAP server has to be shutdown to release the database files to the backup utilities. The database files can now be copied to another location. The preferred method to perform a backup is to dump the data in a so-called LDIF format file. This format is interchangeable. To restore a backup one can copy the backup database files to the right location. The preferred method is to use the LDIF file to restore the data.

With the BDB back-end the OpenLDAP server doesn’t have to be shutdown to perform the backup. The LDIF backup file can be created and restored with the server running. For systems with the requirement of running 24 hours a day, back-bdb is the most suitable solution when using OpenLDAP.

With a relational database backup and restore procedure can be effective. A database server can still operate if a user wants to create a backup. Such a feature is called a hot backup and is generally supported by databases (except
CHOOSING LDAP OVER A RELATIONAL DATABASE

MySQL with standard table type). Restoring data with databases can also be done while the system is still on-line.

2.12 Extending server functionality

A relational database provides a standard set of extra functionalities such as aggregations, date/time and conversion functions. Using these functionalities one can save significant amount of time. For report purposes the aggregation functions are very useful. The maximum, minimum and average can be easily calculated. For example to calculate the average of the orders the user has to query all orders and process them manually (or write a script/program to do so) with OpenLDAP. With a RDBMS a query is formulated and executed. The RDBMS will do all the work outlined by the query and the result will be returned.

Relational databases also offer functionality to write user-defined functions to extend the server with functionality. A user defined function is a function declared in a certain (often called procedural language) language to perform operations on the dataset. Relational databases work with datasets and therefore it’s practical to implement extra functionality with user defined functions. OpenLDAP implements the LDAP specification and this specification has no user defined function requirement. OpenLDAP is open source and the source code is freely available for download. One can modify this code to add extra functionality.

Events can be useful if the system needs to perform certain checks before an order is inserted. These checks can be implemented with triggers. A trigger is an event that can be executed in case a record gets deleted, modified or inserted. Implementing such functionality with OpenLDAP requires one modify the OpenLDAP source code or to manage these checks at (client) application level.

2.13 Documentation

The documentation of OpenLDAP consist of a few marginal manuals. For extensive understanding of the system one has to read the LDAP RFC [4, 8, 9]. There is also a frequently asked question list but it is too marginal and sometimes the information is incomplete. In contrast to a relational database (such as PostgreSQL, Oracle) there are many books and good extensive on-line documentation. Good documentation and on-line resources are important for fast and good deployment of a product. With regard to this aspect a relational database has a significant advantage over OpenLDAP.

2.14 Summary

OpenLDAP (with BDB greater than 4.1.x) and a relational database has sufficient functionality to guarantee safeness of the stored data. Both systems have tools to recover from system disasters. OpenLDAP has a slight advantage for
2.14 Summary

being lightweight but a relational database can use the extra functionality to provide more ease (provided aggregation functions for example) of use to the user.

There is very little known about the write performance of OpenLDAP. Research [1] has shown OpenLDAP is capable of handling read-intensive operations. The performance of write-intensive applications need further exploration.

The documentation can be an obstacle. There are very few design documents on OpenLDAP (the developers even suggest the design is in the source code). Good documentation is also required to do maintenance and other administrative tasks (configuration and tuning the server for instance).

OpenLDAP has a hierarchical structure that is elegant to implement the reseller system. Access can be regulated elegantly. OpenLDAP has an advantage over a relational database with respect to this point. Also OpenLDAP offers more standardized authentication options than a relational database.

There are some concerns (write performance and documentation) but OpenLDAP offers the same level of data safety as a relational database except for transactions over a sequence of operations. This will be no major problem for the reseller system as access will be mainly based on objects (each order is one data object in the reseller system). The safeness of data is the first property, which OpenLDAP can fulfill better than a RDBMS. Write performance needs additional research to ensure OpenLDAP will be able to handle such a write-intensive application like the OpenFortress reseller system. The next section will explore the write performance of OpenLDAP.
3 Approach to test OpenLDAP

Before this thesis work, there is very little is known about using OpenLDAP with write-intensive applications. A top down approach is selected to explore possible problems. The system is considered as a black box and tests are performed on this system. The black box approach is chosen because it is not known where potential problems may exists. Observations such as running time and system load will be recorded and analyzed.

To determine whether OpenLDAP is suitable for write-intensive applications (and to what extent) a program is written to simulate certain workload. Test are performed with different percentage of read, write and authentication actions. If problems do occur during this test, the cause for this problem can be located. If there are no problems encountered a workload has to be generated to simulate the expected workload of the reseller system. Running time and system load will be also recorded with these test to determine if the running time and system load are acceptable for the reseller workflow system. Acceptable running time is defined in the order of ten operations per seconds.
3.1 Test setup

This section describes how the benchmarks are performed and what kind of hardware and software has been used. Figure 6 presents a schematic overview of the test environment.

![Test setup diagram]

Figure 6: Test setup

There is one client communicating over TCP/IP with the OpenLDAP server. Each client thread has its own connection with the server. A client can have a certain number of threads running. The OpenLDAP server was compiled with several back-ends but only back-ldbm and back-bdb was used for these tests. Both client and server are connected to a 100 mbit/s switch. No other devices were connected to this switch.

3.2 Hardware and software used

This subsection describes the hard and software used in detail. First the detailed information about the server is given, followed by the detailed information on the client.
3.2.1 Server

Table 2 lists what kind of hardware and software was used for the OpenLDAP server. The OpenLDAP server is running on an Intel Pentium 4 system operating on the Gentoo Linux distribution.

<table>
<thead>
<tr>
<th>OpenLDAP server (hardware)</th>
<th>OpenLDAP server (software)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium 4 (northwood) 2.53 Ghz (533 fsb)</td>
<td>configured Linux 1.4rc2</td>
</tr>
<tr>
<td>512 MB DDR ram (cas 2)</td>
<td>Stock kernel 2.4.20 using ext3 file system</td>
</tr>
<tr>
<td>ultra dma enabled hard disk</td>
<td>g++ 3.2.2 used to compile software</td>
</tr>
<tr>
<td>100 Mbit/s network interface card</td>
<td>OpenLDAP 2.0.27 / 2.1.17 (compiled with 02 and i686 optimizations)</td>
</tr>
<tr>
<td></td>
<td>only cron, syslog and ssh daemon where running</td>
</tr>
</tbody>
</table>

Table 2: Software and hardware used by server

The OpenLDAP server 2.0.27 and 2.1.17 were both tested. OpenLDAP 2.0.27 was configured with back-ldbm. OpenLDAP 2.1.17 was both configured to run with back-ldbm and back-bdb. Communication with OpenLDAP was only allowed with LDAP protocol version 3 [4]. Linux (kernels 2.4.x. and 2.6.x) was chosen because (at the time of writing of this thesis) it has a good thread handling (OpenLDAP is a multi-threaded application).

3.2.2 Client

<table>
<thead>
<tr>
<th>Client(s) (hardware)</th>
<th>Client(s) (software)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium 4 (willamette) 1.8 Ghz</td>
<td>Gentoo linux 1.4rc1</td>
</tr>
<tr>
<td>512 MB DDR ram (cas 2.5)</td>
<td>Stock kernel 2.4.18 (with pre-emptive patch) using ReiserFS file system</td>
</tr>
<tr>
<td>ultra dma enabled hard disk</td>
<td>g++ 3.2.2 used to compile software</td>
</tr>
<tr>
<td>100 Mbit/s network interface card</td>
<td>only cron, syslog and ssh daemon where running</td>
</tr>
</tbody>
</table>

Table 3: Software and hardware used by client

Table 3 illustrates what kind of hardware and software was used to perform the test. The test client will be executed on one computer. It simulates N number of clients by creating N threads, which simulates parallel execution of clients. The tasks to be performed are defined in a central task queue. Client threads can access this queue. Because this central queue is a shared resource, access to this shared resource must be regulated (mutual exclusive access). The
task queue is generated before the clients start. The creation of the task queue can be influenced by four parameters. These parameters are:

- Total number of tasks
- Percentage of the total number of task that are read operations
- Percentage of the total number of task that are writing operations
- Percentage of the total number of task that are authentication operations

A simple algorithm is used to ensure the queue will reflect the percentage of each type of operation. The algorithm creates 10 tasks each time for each block. The 10 tasks are being distributed over the percentages of read, write and authentication. Figure 7 illustrates a task queue with 60% read (illustrated with the symbol R), 30% write (illustrated with the symbol W) and 10% authentication (illustrated with the symbol A) operations.

\[
\begin{array}{cccccccc}
R & R & R & R & W & W & W & A \\
\end{array}
\]

Block 1 Block ... Block ...

Figure 7: Example content of a task queue

Each block following block 1 is arranged the same way as block 1. The arguments and parameters can be different (i.e. update other record, select other record). After the queue has been created, a predefined number of threads will be created. A start signal is sent to the created threads to start the actual test. The threads will perform all the tasks defined in the task queue. Each thread retrieves a task from the tasks queue and tries to execute it. Upon completion of the task, a new task is acquired until there are no tasks left. Time used to complete the tasks will be recorded. Linux was also chosen for the client with the same motivation as the server (the test program is also a multi threaded application).

3.3 Test results

The test is performed with different parameters. This way, one can obtain an indication of the extent to which OpenLDAP is capable of handling write-intensive applications. This subsection will discuss the results of the preliminary test done with OpenLDAP.

3.3.1 OpenLDAP configuration

The OpenLDAP server uses default configuration supplied by the installation. The OpenFortress specific schemas were added to the configuration as well as
the access control list. OpenLDAP has a default cache size of 1000 entries. Logging (query log, access log and so on) is disabled during the tests. Logging will imply frequent disk access and it is not desirable to have this 'noise' during the test.
3.3 Test results

3.3.2 Evaluation of preliminary results

The performance of back-bdb will be less than back-ldbm because of the additional transaction overhead. Each operation needs to be logged (WAL) with the BDB back-end in order to be able to recover from a disaster. This logging is done with log files which are written to the disk. It’s this disk access that causes the slowdown.

The difference in performance between read-intensive simulation and write-intensive simulation should be relatively small. An explanation for this behavior can be found in the storage mechanism used by OpenLDAP and BDB. BDB uses a persistent on-disk cache. This cache resides on the hard disk and is filled upon access of certain elements. A read operation will always yield a write action to the on-disk cache (the requested element will be first read into the on-disk cache before it becomes available to OpenLDAP). A write action will first cause the elements to be read into the on-disk cache of BDB and later on that element will be modified and the on-disk cache will eventually be synchronized with the actual storage of the data (the *.bdb files). OpenLDAP itself also has a cache in memory. The on-disk BDB cache is required by bdb to ensure consistent and safe data storage (see [20] for more detailed information). BDB stores per-thread and per-process shared information in an environment region. Locks and mutexes are also stored in these regions as well as the on-disk cache.

Figure 8: Degrading performance after a number of operations
During the performance test a problem has been discovered after about 10 test runs. The OpenLDAP server seems to do nothing at a certain point. It doesn’t seem to use any computing power at all and new request to the system are significant slower than the previous ones. Additional requests are stalled even longer. The response time seems to decrease exponentially. Graph 8 illustrates this problem. The reseller system must run 24 hours a day and must not break down after a number of requests. Read operations or write operations or a combination of them all cause this behavior so it’s safe to assume the problem is not related to the operation type. It is triggered after a certain number of requests. The only way to repair this behavior is to increase the BDB cache-size parameter for the on-disk cache or to increase the capacity of the cache in OpenLDAP. Due to this solution it is assumed that the behavior is somehow cache related. An inefficient cache algorithm might be used or perhaps the cache is flushed all the time because the number of requested elements is in essence a large sequence, which exceeds the cache capacity limit. The preliminary benchmark program will not request an element which have been requested before and therefore renders the cache useless. The cause for the problem was eventually tracked down to bugs in BDB versions \( \leq 4.2.48 \).

The BDB subsystem can be configured to ensure ACID properties. OpenLDAP 2.1.x with back-bdb uses write ahead logging and transactions provided by the BDB subsystem to ensure data safety. Despite the ACID properties of the BDB subsystem, the system did crash and could not be recovered to a consistent state. This problem could however not be reproduced systematically and therefore it was impossible to determine the exact cause.
4 Proposed solutions

A performance degradation problem has been discovered during several test runs. The response time of a request would collapse after a number of requests. Increasing the cache size will hide/prevent the performance degradation. The cause for this problem is believed to be the ineffective cache replacement policy with a large sequence and large looping access patterns (explained in subsection 4.3.1) in combination with a polling resource claiming mechanism (explained in subsection 4.2). A possible improvement for resource claiming mechanism is presented in subsection 4.2 and a possible solution for the performance degradation is presented in 4.3. Subsection 4.1 will first discuss in detail how queries work in OpenLDAP and how they depend on the caching mechanism.

4.1 OpenLDAP query processing

There are basically four types of query (Add, Remove, Modify, Search) which can be sent to an LDAP server. The four types of queries interact similarly with OpenLDAP and its buffer management system. The Modify and Search operations can produce a cache hit. The Add operation can produce a cache hit if the element which will be added is already present in the cache. This is similar to the Remove operation where the element might reside in the cache.

A Search operation is processed as follows:

1. Distinguished Name (DN) to ID (a numerical ID) translation (cache interaction)
2. ID to entry number lookup (cache interaction)
3. Retrieve base element and candidates
4. Filter the candidates and return the results

Because BDB can only store pairs of (key,value) a DN has to be translated to an ID (ID will be used as key). This ID needs to be mapped back to an entry. The DN is used to lookup an ID entry in the cache. If the ID is not present a new ID is created and inserted into the cache. With the acquired ID number an entry number lookup is performed. This entry number can be used to retrieve the base element and candidates matching this entry number. A base entry could be the reseller entry and the candidates can be certain order numbers. The candidates are then filtered with a filter criterion (such as order number). The matching results are then returned.

The Modify operation works similar to the search. There is an additional step with the Modify operation. First the entry is retrieved (same mechanism as search operation) and after the retrieval the modifications are made and the results are stored in the cache (and disk).

The Search and Modify queries heavily depend on the caching mechanism. The Add and Remove operations works in a similar way.
There is another type of interaction that causes OpenLDAP to access the cache. The login procedure and credentials check also involves a cache access. The four steps of the Search operation also apply to this interaction. The next subsection will explore and test a possible solution for resource polling mechanism.

4.2 Add back-off mechanism

Most requests to back-bdb are done through a construction denoted in figure 9. The figure shows several threads trying to acquire a resource. If a thread fails to acquire the resource it will immediately try again.

![Figure 9: Multiple threads claiming one resource](image)

On a busy OpenLDAP server a resource is likely to be in use. OpenLDAP uses locks and mutex to control access to the Berkeley database (BDB) file. Only one writer is allowed at a time with BDB and such retry without wait is only a waste processing time. An exponential back-off mechanism can solve this problem. A wait counter is used by processes to back-off and to retry later. The wait counter is doubled each time a process fails to acquire a resource. A relatively small number of retries will result in a small delay. A large number of failed retries will result in a long period of delay. The idea behind the algorithm is that a relatively large number of failed resource acquire attempts indicates the systems is busy and it will be better to wait for the resource. Ether network also have a similar problem. There is only one channel where multiple clients can send data to. If a collision (e.g. channel resource is taken) occurs a binary exponential back off is used to resolve the conflict (as shown in [21]). The same algorithm will also be used to resolve the resource claiming conflict. There is a back-off mechanism in the development version of OpenLDAP and it has been back ported to test whether the problem still exists. The problem still existed after back porting this back-off mechanism. The cause of performance problem is not the resource claiming mechanism. The next subsection will discuss the second potential cause for the performance degradation problem.
4.3 Buffer management

A cache system consists of three parts. These parts are:

- A main (cache) memory
- An auxiliary memory
- A replacement policy

The main memory is the memory where the cache items will be stored. This memory is fast but expensive compared to the auxiliary memory. In a cache system there is a fixed (relative small compared to the auxiliary memory) amount of main memory and a large amount of auxiliary memory. Data from the auxiliary memory is first read into the main memory before it will be used. Accessing this data from the main memory is faster than accessing it from the auxiliary memory. Data, which has recently been read into the cache is expected to be used again in a short period of time (near future). This future access of data will be read from the fast main memory. Data elements are constantly added in this main memory until the cache capacity is reached. If the cache system is full (capacity has been reached), an element has to be elected to be replaced by a new element. Such a process is called a replacement policy. A replacement policy is an algorithm that determines what element will be swapped out in favor of a new element if the cache system has reached its capacity. There are many different replacement policies described in the literature. A short description of a few algorithms and their characteristics are given in the following subsections.

Several commonly used metrics will be to determine the effectiveness and the cost of the replacement policy. Metrics used in this thesis are:

- Cache hit rate ($H_r = \frac{\text{hits in the main memory}}{\text{total request to the cache}} \times 100\%$)
- Computational overhead (Number of list-iterations used)
- Space overhead (Additional amount of memory needed)

A replacement policy is effective if the hit rate $H_r$ high. A high $H_r$ indicates most data items were in the main memory when requested and a low $H_r$ indicates most items were not in the main memory when requested.

Computational overhead is defined by the number of times the algorithm has to iterate through a data set to perform a certain action. A lower and upper bound can be given that represent the best-case and worst-case scenario for that algorithm. The average computational overhead can be use as a characteristic in general for an algorithm. Computational overhead can be polynomial, constant, logarithmic and exponential (combinations are also possible). A constant computational overhead requires a constant time to iterate through a data set regardless of the dataset size. The cost increases logarithmically or exponentially with a larger dataset with logarithmic and exponential computation overhead. The space overhead cost is expressed by the amount of extra memory
needed by the replacement policy. It’s desirable to have a low space overhead because the algorithm doesn’t use a large amount of memory in that case. An ideal replacement policy will have a high $H_r$, low constant overhead and a low space overhead.

### 4.3.1 Access patterns

An access pattern is a sequence of defined actions that will be executed. This thesis will define an access pattern as a sequence of OpenLDAP operations. An operation can consist of the following actions:

- **read**: A read action is used to simulate a request of an order. The information will be fetched from OpenLDAP and returned to the requesting party.

- **write**: A write action is used to simulate an insertion and modifications of an order object. Insertion and modifications causes OpenLDAP to perform write operations.

- **authentication**: Authentication is used to simulate authentication/authorization. A reseller for example has to identified/authorized and these actions will occur regularly.

The structure of such a pattern can be classified. The classification of access patterns used in this thesis are shown in table 4.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small sequence</td>
<td>ordered list of orders. #operations in pattern &lt; cache size</td>
</tr>
<tr>
<td>Large sequence</td>
<td>ordered list of orders. #operations in pattern &gt; cache size</td>
</tr>
<tr>
<td>Small random</td>
<td>random list of orders. #operations in pattern &lt; cache size</td>
</tr>
<tr>
<td>Large random</td>
<td>random list of orders. #operations in pattern &gt; cache size</td>
</tr>
<tr>
<td>Small loop</td>
<td>repeating block of operations. #operations in pattern &lt; cache size</td>
</tr>
<tr>
<td>Large loop</td>
<td>repeating block of operations. #operations in pattern &gt; cache size</td>
</tr>
<tr>
<td>Changing pattern</td>
<td>A combination of two or more different patterns concatenated</td>
</tr>
</tbody>
</table>

Table 4: Classification of access patterns

Classification of access patterns can help to compare the different cache replacement policies with each other. If one replacement policy performs bad with a certain pattern, another replacement policy might be chosen to eliminate this bad performance.

### 4.3.2 LRU: Least Recently Used

The Least Recently Used (LRU) cache replacement strategy is a common cache replacement policy used in a variety of systems. The algorithm assumes recently
used pages will be used again in the near future. A double linked list is commonly used as data structure for LRU. Items on top of the list represent Most Recently Used (MRU) items. Items at the tail of the list are the items which will be evicted if the maximum capacity has been reached for a certain cache and a page fault (cache miss) has occurred.

Figure 10: LRU structure

A brief description will be given to illustrate how the algorithm operates. Figure 10 illustrates an example data structure commonly used for LRU. A cache directory entry is an entry with information describing the page it is referring to. Information such as page number and number of processes currently accessing this page is also stored here. The pointer to the actual cache page is also stored here. If some process tries to access a certain page, it will first try to find the desired information in the cache. If that page is located somewhere in the cache, it will be removed from its current position and inserted in the front (at the MRU position) of the cache. Items located at the MRU position represents recently accessed data. A page fault occurs if the requested information is not located somewhere in the cache. In this case a page needs to be read from disk and put on top of the cache (at the MRU position). If the cache has reached its maximum capacity, the entry at the LRU position (least recently used position) of the cache will be removed.

The LRU replacement policy only takes recency into account. Recently used pages are placed on top of the cache and will gradually move to the tail of the cache and are eventually swapped out if not referenced (accessed) again. Popular pages can potentially be accessed frequently over a relatively 'long' period of time. These pages are likely to be swapped out of the cache once they
have been read because the page has a 'long' period of time before it will be accessed again. This is one disadvantage of LRU.

Another problem for LRU is a large sequence pattern. A sequence pattern is a pattern with \( N \) successive different items which will be requested. Let \( C \) be the capacity (number of entries) of a LRU cache. It is clear to that if \( N \) is greater or equal to \( C \) all cache pages will travel from the MRU position to the LRU position and are eventually be swapped out. Such access patterns performed against a LRU cache will render the cache useless. No caching mechanism is better in this situation than a LRU cache. It’s evident that LRU will have a problem with looping patterns with a loop size is greater or equal to the cache capacity. Looping patterns can be considered as a number of large sequence patterns executed sequentially (access patterns are explained in subsection 4.3.1)

The changing access pattern can also be a problem for a replacement policy such as LRU. Changing access pattern is a combination of access patterns. The problem explained with LRU and large sequence patterns and large looping patterns also applies here if one of these two patterns is present in one of the combinations of the changing pattern.

4.3.3 LRU-K/LRU-2

The LRU-K algorithm paper[11] introduces an improved LRU algorithm, which also takes frequency information into account. LRU-K keeps track of the number \( K \) last references to a popular page. This information is statistically used to determine which page will be used frequently and therefore be given higher priority than less frequently used pages. LRU-2 \((K = 2)\) needs to maintain a priority queue and usage of a priority queue results in a logarithmic computational overhead. A priority queue is implemented as a heap structure (shown in [22]). Insertions and extractions (deletes) on heap structures require a logarithmic \((\log(C))\) computational overhead. As shown in [10] a \(\log(C)\) implementation complexity is a severe overhead with large cache sizes.

Eviction candidates are selected based on the values of the backward K-distance function. The cache entry with the highest backward K-distance value will be evicted. If two or more cache entries have the same largest value a subsidiary algorithm is required to determine the eviction candidate. The algorithm performs well for most traces (shown in [10]) and can adapt to changing patterns. Large sequence patterns and looping patterns will not flush out popular pages because of their backward K-distance value.

LRU-2 requires an offline parameter that captures the amount of time a page has only been seen once recently and should be kept in the cache. Different access patterns (workloads) require different parameter values for optimal algorithm-performance.

4.3.4 2Q

2Q [13] uses two queues to maintain its cache. The first queue is a First In First Out (FIFO) queue (named A1) and the second queue is a plain LRU queue
(named Am). The basic idea behind 2Q is to store hot pages in Am. Hot pages are popular pages, which will be accessed frequently in time. The data structure for 2Q is graphically given in figure 11.

![Figure 11: 2Q data structure](image)

A first reference to a page will always cause the page to be placed on top of the A1 buffer. During its resident a page can move from the A1 buffer to the Am buffer if that page is referenced for a second time. This page is considered as a hot (popular) page and will be kept in the main queue (Am). Pages that have not been referenced for a second time are considered cold pages and eventually will travel down to the tail and swapped out of the A1 queue. The Am queue is maintained as a normal LRU queue discussed in subsection 4.3.2. A page in the Am queue will only be swapped out if the capacity of Am has been reached and a hot page has been identified in the A1 queue.

The size of the A1 and Am queue needs to be selected carefully. A small A1 will prevent effective recognition of hot pages. Possible hot pages can be removed from the A1 buffer before they will be referenced again. A large A1 implies a small Am and therefore less capacity to store hot pages. These offline parameters need to be carefully tuned to get optimal algorithm-performance.

2Q has a constant time computational overhead, which is better than LRU-2. The performance [10] of 2Q and LRU-2 are very similar with carefully selected offline parameters.

2Q can resist a large sequence pattern and large looping pattern. Such patterns will only flush out the A1 queue leaving the Am queue intact. Let \( N \) be number of different references and \( C_{a1} \) be the capacity of A1 queue and \( C_{am} \) be the capacity of the Am queue. The only way Am could get polluted
is when \( N < C_{a1} \) is true. A sequence pattern with \( N < C_{a1} \) will cause very little pollution since this pattern is performed only once. However a small \( Am \) can cause hot pages to be swapped out. A constant repeat of this pattern is considered as a looping pattern and will eventually flush out all hot pages in \( Am \). This again demonstrates the importance of the two offline parameters.

4.3.5 MQ

Multi queue (MQ) replacement policy was designed for storage controllers (such as IDE controllers and RAID cards). MQ uses \( m \) (for example \( m = 8 \)) LRU queues. The queues are named \( Q_0, \ldots, Q_{m-1} \). A queue \( Q_i \) contains pages which have been seen between \( 2^i \) times and \( 2^{i+1} - 1 \) times recently. Table 5 illustrate how the queues are organized for \( m = 8 \).

<table>
<thead>
<tr>
<th>Queue</th>
<th>( Q_0 )</th>
<th>( Q_1 )</th>
<th>( Q_2 )</th>
<th>( Q_3 )</th>
<th>( Q_4 )</th>
<th>( Q_5 )</th>
<th>( Q_6 )</th>
<th>( Q_7 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>seen #times</td>
<td>1..1</td>
<td>2..3</td>
<td>4..7</td>
<td>8..15</td>
<td>16..31</td>
<td>32..63</td>
<td>65..127</td>
<td>128..255</td>
</tr>
</tbody>
</table>

Table 5: MQ with 8 LRU queues

\( Q_0 \) holds elements that have been seen only once recently (accessed once recently), \( Q_2 \) holds elements that have been seen at least two times and at most three times recently and so on. Cache pages can move from \( Q_0 \) to \( Q_7 \) when referred to frequently. A page is swapped out for a new page of \( Q_i \) when it reaches the LRU position of the \( Q_i \) and \( Q_i \) is at max capacity.

MQ has a constant time computational overhead but it is dependant on the number of queues. The lookup functions needs to look into all queues until it finds the requested page. If the number of queues is large, lookups can get expensive.

4.3.6 ARC

Adaptive Replacement Cache [10] has been proposed by researchers of IBM. ARC is a low computational, low space overhead and self tuning replacement policy. ARC uses four LRU queues as data structure. Two of these LRU queues have pointers to cache pages. The remaining two are ghost caches. A ghost cache is a cache directory structure without a valid pointer to an actual cache page. The data structure is shown in Figure 12.
4.3 Buffer management

![ARC data structure diagram](image)

Figure 12: ARC data structure

The four lists are named T1, T2, B1 and B2. T1 holds elements that have been referenced only once and T2 holds elements which have been referenced at least twice. B1 is called the bottom of T1 and holds directory entries from recently evicted pages from T1. B2 is the bottom of T2 and holds recently evicted entries from T2. B1 and B2 will be used to guide a learning variable called Target_T1. Intuitively a cache hit in B1 is an indication for a small T1 size and a cache hit in B2 is an indication for a small T2 size. Cache hits in B1 and B2 will adjust Target_T1 to reflect the target size of T1.

A first reference to a cache page will cause the page to be placed at the MRU position of T1. This page will move to the top of T2 if it is referenced again before it has been swapped out of T1. If a cache hit occurs in the B1 queue of the cache, the cache directory will be moved to the top of T2 and the actual data from disk will be re-fetched. Cache hits in T2 will cause the page to be removed from its current position and placed on top of T2. If the hit occurs in B2, the cache directory will be moved from B2 to the top of T2 and the actual data from disk will be read into the cache page.

As mentioned before, elements in B1 and B2 don’t have a valid pointer to a cache page. Elements resident in one of these queues will eventually be removed. The T1, T2, B1 and B2 can also dynamically grow or shrink. The target size for T1 determines the size of T1, T2, B1 and B2. The learning variable Target_T1 needs to be updated after each page fault and cache hit. Luckily this book keeping time can be considered as constant time overhead.

ARC performs well on most access patterns and is able to adapt to evolving (changing) access patterns. The algorithm is self tuning and has a low constant overhead.
4.4 Choosing a cache replacement policy for OpenLDAP

As explained in subsection 4.3.2, LRU has problems with certain access patterns. With certain access patterns it’s better to have no cache system because those access patterns will flush the cache constantly. Let $O_s$ be the symbol representing space overhead and $O_c$ the symbol representing computational overhead. Table 6 denotes the characteristics of the discussed replacement policies.

<table>
<thead>
<tr>
<th>Replacement policy</th>
<th>Overhead</th>
<th>Recency</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>low constant $O_c$, low $O_s$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>LRU-2</td>
<td>logarithmic $O_c$, low $O_s$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>2Q</td>
<td>low constant $O_c$, low $O_s$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MQ</td>
<td>constant $O_c$, low $O_s$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ARC</td>
<td>low constant $O_c$, low $O_s$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Replacement policy</th>
<th>Adapt to changing patterns</th>
<th>Require offline parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>LRU-2</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>2Q</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MQ</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ARC</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 6: Characteristics of replacement policies

LRU-2 has a logarithmic computational overhead which makes this policy less suitable for large caches. The offline parameter for LRU-2 is also hard to define for all access patterns. The 2Q replacement policy removes the logarithmic computational complexity but still requires an offline parameter. Several different values are optimal for different access patterns and therefore makes this replacement policy hard to function optimal for all discussed access patterns. MQ is not suitable because it also has offline parameters. It has the same weakness where different parameters are optimal for certain access patterns. ARC replacement policy is suitable to replace LRU in OpenLDAP because it doesn’t require an offline parameter. Instead the algorithm is self-tuning and can therefore tune itself to different workloads (access patterns). ARC can also withstand a large sequence access pattern. Such a pattern will only flush the T1 and elements in T2 will be preserved.

It has been shown in [10] that ARC performs almost equally or better than the other replacement policies. LRU-2 and 2Q were tested with the optimal parameter for each access pattern. Although the reseller systems has a stable (constant) access pattern, it is better to have a cache system, which functions well for more scenarios. Other write-intensive applications can exist with for example changing access patterns. The next subsection will describe the substitution of LRU with ARC in PostgreSQL. PostgreSQL also uses LRU as cache replacement policy and with certain access patterns (traces) LRU also show its
weaknesses causing performance problems. An improvement with ARC in PostgreSQL is a good indication that OpenLDAP will also profit since both systems are (cache-wise) similar.

### 4.4.1 Experience with substitution of LRU with ARC in PostgreSQL

PostgreSQL is a sophisticated open source Relational Database Management System (RDBMS) originally developed at the University of California at Berkeley. PostgreSQL has evolved into a full-featured database with support for views, stored procedures, triggers, transactions, multi-version concurrency control (MVCC) and more. MVCC is interesting to this discussion because under certain circumstances it can also cause the cache to be flushed. MVCC allows a process to take a snapshot of data currently in the database. This snapshot can then be processed without having to worry about other processes which might alter the current data. Changes from multiple snapshots are automatically merged by the system. While MVCC improves concurrency, it also has a disadvantage. Deleted data cannot be removed directly because these pages can be in use by another snapshot, instead they are marked as deleted (invalid). An RDBMS using MVCC thus requires a periodic background process to reclaim these 'marked for deletion' data pages on disk. During this reclaiming period processes may not access these data and therefore the data is locked. Reclaiming marked for deletion database pages is done in PostgreSQL by a command/program called vacuum. To make matters worse, a vacuum has to check all pages and will flush the carefully populated LRU cache. Another potential problem with vacuum is the disk IO that's associated with it. If one vacuums too frequently the system is flooded with disk IO operations (from the vacuum itself, but also from write ahead logging).

As stated before PostgreSQL uses a LRU cache. This cache is located in the shared memory and supplies buffers to back-ends. A back-end is a process which executes some SQL query. These buffers are organized as a circular buffer using LRU as replacement policy. During processing of queries, a back-end may need to read a whole table to do some data processing. Reading a whole table can be compared to executing a sequence access pattern. If the table has more entries than the cache can hold, such a "sequential scan" on the table will also flush all of the LRU cache contents. Figure 13 illustrates a schematic overview of a sequential scan. The sequential scan requires a read of all the data items before it can make a selection of the data.
The opposite of a sequential scan is an index scan, where a back-end can use an index to retrieve only certain parts of the data without having to access all the records of a certain table.

Figure 13: Sequential scan

Figure 14: Btree index scan
Figure 14 is a schematic overview of an index scan (btree index in this case). An index is used to determine the upper bound and lower bound of the data. Only required data is read using this method.

During the release cycle of PostgreSQL 7.4 a vacuum daemon was introduced. Using a low vacuum interval, massive disk I/O is generated. With this massive disk I/O the cache is also flushed. While executing vacuum often helps the optimizer with extra statistical data, it has a negative impact on the disk I/O performance. ARC can be used as solution for this problem since it can withstand a sequential scan.

On-line Transaction Processing (OLTP) is a class of transaction-oriented applications typically used for data entry or data retrieval in a number of industries such as banking, airlines, mail order, supermarkets, and manufacturers. Access patterns (often called traces) for these types of applications are used to benchmark databases. During performance tests of PostgreSQL 7.4 with several OLTP traces a few performance problems were encountered. Traces that had a similar effect as a large sequential scan were causing slowdowns. ARC can also be used as solution in this case.

After evaluating 2Q, LRU-2 and ARC, the last cache replacement policy was used to replace the LRU replacing policy inside PostgreSQL. The main reason to choose for ARC was the ability of ARC to withstand sequential scans and the ability to automatically tune the replacement policy. The traces that were causing problems before are now running smoothly [19]. The system is now responsive during these traces. Without ARC the PostgreSQL server was not responsive during the traces. The author of this thesis actively helped the PostgreSQL community to evaluate 2Q, LRU-2 and ARC. The ARC replacement was identified as the most suitable policy and is integrated for the next major release of PostgreSQL (8.0).

4.5 Substituting LRU with ARC in OpenLDAP

ARC in PostgreSQL has been shown to improve performance (especially against cache flush patterns). OpenLDAP is a similar system to PostgreSQL (illustrated in figure 4) and it is reasonable to assume a similar improvement in the cache system of OpenLDAP will also improve performance against a large sequence pattern. The next subsections will describe how LRU is used in OpenLDAP and how ARC will replace it.

4.5.1 LRU in OpenLDAP

OpenLDAP uses the LRU as cache replacement policy. Figure 15 illustrates the structure used to support LRU in OpenLDAP.

A global cache structure is present to access the cache. This global structure has a pointer to the head and the tail of the cache. OpenLDAP supports multiple back-ends, whereby each back-end can have different storage structure requirements. Abstract information entry is supplied to support the different storage requirements. In order to support multiple back-end, OpenLDAP uses...
an entry abstraction layer. Traveling through this layer one can obtain the private entry data. The private entry stores all the back-end specific information. These private entries are organized as a double linked list. All entries except the head and the tail will have a pointer to the next and the previous elements. The head element doesn’t have a pointer to the previous entry and may have a pointer to the next element. This pointer may be absent if there is only one element in the cache. In this case the tail also points to this element. Tail elements may have a previous element.

The LRU implementation in OpenLDAP has been tested for a period of releases and is considered stable. The locking mechanism in OpenLDAP is also considered stable. The strategy is to make a minimal number of adjustments in the code to replace LRU with ARC. Using this strategy one can benefit from the (proved) implementations.

4.5.2 Modify LRU structure to support ARC in OpenLDAP

Figure 16 illustrates the changes required in the current LRU implementation in order to replace the LRU replacement policy with ARC. The global cache structure, abstract entry and private entry info are still present. Preserving these structures will result in less modifications to the current OpenLDAP code. The double linked list used to connect private entries has been removed. This is LRU specific and therefore its removal is safe.

There are four list (T1, B1, T2, B2) added to the cache structure. T1 and T2 can hold actual cache pages whereas B1 and B2 are phantom caches. Phantom caches are caches without data page entries. Phantom caches only maintain the directory and directory entries (Not the actual cache page). T2 has the same structure as T1, the cache directory and its elements are not shown in figure 16 to keep the illustration simple. The same holds for B2 and B1.
4.5 Substituting LRU with ARC in OpenLDAP

An extra pointer is added between an abstract entry and a directory entry. Adding this extra pointer will allow fast access from an abstract entry to a directory entry. If this pointer is not present, an access to a directory entry from an abstract entry would result in a linear search through T1 and T2. This linear search time can be large if the caches size is large. A search through B1 and B2 is not required because these list cannot hold actual entries.

4.5.3 Modify code to support ARC in OpenLDAP

A cache replacement policy always has two cases. The first case is a cache hit where an entry is requested which is located in the cache and the second case is a page fault (cache miss) where an entry is requested and is not located in the cache. The LRU implementation for these two cases is relatively simple. A cache hit causes the element to be moved in front of the LRU list and a cache miss results (if necessary) in an eviction of a least recently used page and
insertion of the newly read information. The implementation of these two cases for ARC is harder. A cache hit can cause an element to switch from list (i.e. element can move from T1 to T2). It also may cause an element to be moved to the phantom cache.

ARC is implemented as two functions. The first function is a process cache hit function, which handles a cache hit. The second function has logic to handle a cache miss. These two functions will be inserted into the current OpenLDAP implementation. The LRU implementation will be removed. Inserting calls to cache hit and cache misses will cause a minimal adjustment to the code and therefore is less likely to introduce new bugs in the current implementation. Another advantage is that the locking mechanism can be reused. The disadvantage of this ARC implementation is the longer period of time required to hold an exclusive lock on the cache. If the exclusive lock has been claimed all other threads trying to acquire that mutex will block them self until the current holder of the exclusive lock releases this lock. Less concurrency is possible with this construction. A refinement of the global mutex into four mutexes for T1, T2, B1 and B2 can improve concurrency. Also moving the mutex acquire/release into the cache hit and cache miss function can improve concurrency.

Reimplementing data fetches from disk is also avoided. Manually fetching and decoding this information requires many function calls and error handling. Changing large chunks of code is error prone and therefore avoided. Instead the cache miss call will be inserted after a successful read action from disk. Using this implementation, results about cache hits and cache misses can be measured. These metrics will give a good indication of the cache performance.

4.5.4 Problems encountered during implementation

Several problems were encountered during implementation. The first problem is the lack of a good freely available debugger that can handle multi threaded applications. OpenLDAP can run in a single thread and can therefore be loaded into a debugger like DDD (a graphical front end for gdb). However problems that only occur with multiple threads cannot be simulated/debugged using this setup. Debugging multi threaded applications is possible but it is very hard due to locking and the fact that each of the running thread (or multiple threads) could be causing a problem. OpenLDAP has fine grained logging mechanism that can be used to output debugging information to syslog. Review of this log can also reveal complex problems but it is time intensive (the log for a full trace is very large, and takes a lot of time to be studied).

The BDB 4.1.x had several bugs in the locking subsystem. These bugs were causing unexplainable slowdowns and were confirmed by Sleepycat (the company that develops the BDB back-end). A solution for this problem was to increase the OpenLDAP cache size. The error was assumed to be located somewhere inside the cache handling routine (because the problem was 'solved' with a bigger cache size). This assumption was proven to be false since the ARC implementation also suffers from the same problem (although it required a longer period of runs to reproduce the flaw). The problem was finally tracked
down to several bugs in the BDB subsystem. These bugs have been fixed in the next major release of BDB (version greater than 4.2.52). All BDB 4.1.x versions still have the bugs and it is not recommended to use this version.

Lack of design and implementation documents also slows down development significantly. It is hard to read code with very limited code comments and its even harder to insert/modify code with limited documentation. Also the code is full with so called “#ifdefs” which makes to code even harder to read and to understand.

4.6 Performance test

4.6.1 Test setup

The test setup for the second series of performance tests are more or less the same as the initial setup. The software used on the server was upgraded due to bugs discovered in BDB 4.1.x. A newer version of BDB (4.2.52) was used for the second series of performance tests. Also OpenLDAP 2.1.25 was used instead of the older version. Version 2.1.25 had several important bdb back-end fixes (crash fixes).

The software for the clients also has been updated. The second series of tests uses predefined access patterns. These patterns will be generated with a pattern generator. The source codes of the pattern generator and benchmark program can be found at http://www.lokishop.nl/verslag/code/. The main benchmark program is also altered to read a pattern at startup instead of generating one pattern on the fly.

The compiler used to compile OpenLDAP and the benchmark client is upgraded to version g++ 3.3.2. Version 3.2.x sometimes generates faulty code for Pentium 4 class processors [15] with -O3 flag.

The test procedure have also been altered to rule out more external influences. The new procedure is as follow:

1. Stop the OpenLDAP server (if it’s still running).
2. Restore the bdb data. For this purpose a zip file was created with the content of a newly imported OpenLDAP database containing a test set of order items.
3. Restore BDB environments with `db_recover`. `db_recover` will reset the BDB environment to the defaults or to the values specified by the configuration file.
4. Flush the operating system cache by issuing a copy command of 100 MB from /dev/urandom to a file on the hard disk. This will cause the file system cache to be flushed.
5. Start the OpenLDAP server.

\footnote{O3 optimization was used to generate optimized code for the benchmark}
6 Perform benchmark program with a certain pattern

After the test has been completed the results are recorded in a spreadsheet. The whole procedure will be performed from step 1 to step 6 for the next pattern. All test are performed five times to compute the average value (i.e. the whole test procedure is repeated five times for one pattern). The next subsection will describe what data is recorded and how they are calculated.

4.6.2 Definition of recorded data

This subsection describes the definition of the recorded data. The definition is given first for OpenLDAP with LRU followed by OpenLDAP with ARC. Table 7 list the definitions used for measurements with the LRU replacement policy.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description for LRU replacement policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>hits</td>
<td>Number of access to an item which was located in the cache</td>
</tr>
<tr>
<td>misses</td>
<td>Number of access to an item which was not located in the cache</td>
</tr>
<tr>
<td>hit ratio</td>
<td>( \frac{\text{hits}}{\text{hits} + \text{misses}} \times 100% )</td>
</tr>
<tr>
<td>miss ratio</td>
<td>( \frac{\text{misses}}{\text{hits} + \text{misses}} \times 100% )</td>
</tr>
<tr>
<td>run time</td>
<td>The runtime in seconds is the time recorded for the completion of one pattern. The runtime is an average of five measurements.</td>
</tr>
</tbody>
</table>

Table 7: Definition of recorded data (LRU)

Table 8 list the definitions used for measurements with the ARC replacement policy. Only hits in the T1 and T2 lists are accounted as real cache hits. Hits in the tail or plain cache misses are accounted as cache misses.
4.6 Performance test

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description for ARC replacement policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>hits</td>
<td>Number of access to an item which was located in the cache. Items are only in the cache if they are located in the T1 or T2 list. Items in the B1 and B2 list are phantom caches and are therefore accounted as cache misses.</td>
</tr>
<tr>
<td>misses</td>
<td>Number of access to an item which was not located in the cache. Items are only in the cache if they are located in the T1 or T2 list. Items in the B1 and B2 list are phantom caches and are therefore accounted as cache misses.</td>
</tr>
<tr>
<td>hit ratio</td>
<td>( \frac{\text{hits}}{\text{hits} + \text{misses}} \times 100% )</td>
</tr>
<tr>
<td>miss ratio</td>
<td>( \frac{\text{misses}}{\text{hits} + \text{misses}} \times 100% )</td>
</tr>
<tr>
<td>run time</td>
<td>The runtime in seconds is the time recorded for the completion of one pattern. The runtime is an average of five measurements.</td>
</tr>
</tbody>
</table>

Table 8: Definition of recorded data (ARC)

The next subsection will describe the access patterns in detail.

4.6.3 Patterns

Several access patterns have been created to test the LRU and ARC replacement policies. The patterns will be discussed in this subsection.

S1  Sequence pattern with a loop. Entries with order number from 1 to 1000 will be accessed. The total number of tasks is 10000 and are structured as chunks of 1000 items. For example: [1..1000][1..1000] ...

S1-S Similar to pattern S1. 1000 order numbers will be randomly drawn from a collection of 50000 order numbers.

S2  Large sequence pattern. Entries with order number from 1 to 10000 will be accessed. The total number of tasks is 10000 and is structured as one chunk of 10000 items. For example: [1..10000]

S2-S Similar to pattern S2. 10000 order numbers will be randomly drawn from a collection of 50000 order numbers.

S3  Small sequence pattern with a loop. Entries with order number from 1 to 500 will be accessed. The total number of tasks is 10000 and is structured as chunks of 500 items. For example: [1..500][1..500] ...

S3-S Similar to pattern S3. 500 order numbers will be randomly drawn from a collection of 50000 order numbers.
L1 Loop access pattern. Entries with order number from 1 to 2000 will be accessed. The total number of tasks is 10000 and is structured as chunks of 2000 items. For example: [1..2000][1..2000] ...

L1-S Similar to pattern L1. 2000 order numbers will be randomly drawn from a collection of 50000 order numbers.

C1 Changing access pattern. This pattern is a combination of the patterns S3 and L1. The total number of tasks is 50000 and is structured in chunks of S3 and one L1 pattern. (e.g. [S3][S3][S3][L1][S3][S3][S3])

C1-S Similar to pattern C1, except patterns S3-S and L1-S was used instead of S3 and L1.

The tests will be performed with a cache with a capacity of 1000 entries, which is the default OpenLDAP cache size. 60% reads, 30% writes and 10% authentications (read-intensive setting) will be used for the first test-series. The second series will have 30% reads, 60% writes and 10% authentications (write-intensive setting). Using these different percentage one can compare performance of OpenLDAP with read-intensive applications and write-intensive applications.

4.6.4 Pattern motivation

In this subsection a motivation is given for each pattern. Every sequential pattern containing numbers from 1 to a predefined number is being paired with a shuffled version of that pattern. Shuffled version of a pattern can be recognized by the postfix “-S” behind the pattern name. The added value of the shuffled version is the ability to test whether some algorithm might perform perfecting.

S1 This pattern will be used to test a flush of the cache with a capacity less than 1000 entries. The chunks of 1000 entries will cause a denial of service (DoS) effect on caches with a capacity less than 1000 entries. Caches with capacity greater than 1000 entries will work very efficiently with this pattern as they can store all elements in the cache.

S2 The large sequence pattern will be used to cause a DoS effect on caches with a capacity of less than 10000 elements.

S3 The small sequence pattern will be used to determine the effectiveness of a replacement policy if there is enough space in the cache to store all the 500 elements.

L1 With this pattern one can determine the effectiveness of the replacement policy against a loop pattern.

C1 This pattern will be used to test the replacement policy against changing access patterns.
4.7 Statistical analysis of the data

This subsection will discuss basic statistical analysis. Readers familiar with basis statistics can skip this subsection.

4.7.1 Overview of the used theory

This subsection describes how the 95% reliability interval is calculated. This interval is used in the result graph to show (statistically) significant differences.

To perform statistical analysis, a number of formula’s [16, ?] will be used. The first formula is $\bar{X}$ (Mean value) and that formula represents the weighted average of a series of test runs. \( N \) is the number of experiments performed.

\[
\bar{X} = \frac{1}{N} \sum_{j=0}^{N-1} X_j
\]

The Mean value \( \bar{X} \) is a characteristic of a distribution. It’s the distributions central value. Width and variability around the mean value is characterized by the variance \( \text{Var}(x_0...x_{N-1}) \) and standard deviation \( \sigma(x_0...x_{N-1}) \). The variance estimates the mean squared deviation of \( x \) from it’s mean value. The standard deviation is the square root of the variance and is in the same quantity as the mean value.

\[
\text{Var}(x_0...x_{N-1}) = \frac{1}{N-1} \sum_{j=0}^{N-1} (X_j - M(X))^2
\]

\[
\sigma(x_0...x_{N-1}) = \sqrt{\text{Var}(x_0...x_{N-1})}
\]

A normal distribution (also known as Gaussian distribution) can be characterized by the formula \( f(x, \mu, \sigma) \). \( f(x, \mu, \sigma) \) is the formula for the height \( (y) \) of a normal curve for a given value of \( x, \mu \) and \( \sigma \).

\[
f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}
\]

The \( f(x) \) function with the parameters \( \mu \) and \( \sigma \) can be graphically represented has the histogram listed in figure.
A normal distribution is also known as $N(\mu, \sigma^2)$. For a known $\mu$ and $\sigma^2$ a 95% reliability interval is defined as $[\bar{X} - c \times \sigma / \sqrt{n}, \bar{X} + c \times \sigma / \sqrt{n}]$. The $c$ constant can be retrieved from a so-called $N(0, 1)$ [24] table. $n$ is the number of experiments, $\sigma$ is the standard deviation and $\bar{X}$ is the mean value. A 95% reliability interval indicates the outcome of an experiment will be located in the calculated interval. A so called $t$/student test can be performed to verify the 95% interval claim.

### 4.7.2 Analysis of the data

For all measurements it is assumed that the outcome of the experiments are (Gaussian) normal distributed. To verify the assumption, one test is performed 25 times and each time the runtime is recorded. Table 10 on page 72 shows the run times for these test. The average for the test run is:

$$\mu = \frac{1}{25} \sum_{j=0}^{25-1} X_j = 59.8$$

The variance for this test run is:
4.7 Statistical analysis of the data

\[ \sigma^2 = \frac{1}{25-1} \sum_{j=0}^{25-1} (X_j - M(X))^2 = 1.83 \]

And with the variance the standard deviation can be calculated:

\[ \sigma = \sqrt{1.83} = 1.35 \]

The histogram can be plotted for \( \mu = 59.8 \) and \( \sigma = 1.35 \) as shown in figure 18.

![Histogram with \( \mu = 59.8, \sigma = 1.35 \)]

Figure 18: Histogram with \( \mu = 0.27 \) and \( \sigma = 1.35 \)

A approximation curve can be drawn in the figure and this approximation will resemble the typical graph of a normal distribution. It is therefore reasonable to assume a normal distribution for the outcome of the experiments. With this assumption the reliability interval \( [\bar{X} - c \cdot \sigma / \sqrt{n}, \bar{X} + c \cdot \sigma / \sqrt{n}] \) can be computed. The \( c \) value is determined as 1.96 with the \( N(0, 1) \) lookup table. The final 95% reliability interval is \([59.8 - 1.96 \cdot 0.27, 59.8 + 1.96 \cdot 0.27] = [59.27, 60.33]\). It is now statistically determined that 95 out of 100 experiments will have a
result, which lies in this 95% reliability interval. The reliability interval actually consist of the mean value with plus/minus some $c = 1.96$ multiplied with the standard error of the mean ($\sigma_M$) for a normal distribution. Since all tests are performed with this method, it is reasonable to assume all experiments are normal distributed and therefore the 95% reliability interval for all experiments can be computed given $\mu, \sigma$ and $\overline{X}$ for an experiment.

4.8 Evaluation of results

This section will discuss the results obtained from the tests with the described patterns in the previous section. For each access pattern the test have been runned with a low, medium and large cache size. Low is defined as 250 entries, medium is defined as 1000 entries (the default size of OpenLDAP) and large is defined as 2500 entries. Each result graph will be plotted with the reliability interval. Statistically 95% of the measurements will be located somewhere in this interval. With this interval one can make statistically founded claims.

The notation for the x-axis in the generated graphs is [non-shuffled | shuffled cache size - cache replacement policy]. The x-axis for figure on page 52 (and all figures similar) can be interpreted as described in table 9.

<table>
<thead>
<tr>
<th>x-axis description</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-LRU</td>
<td>Pattern S1, Small cache size, LRU replacement policy</td>
</tr>
<tr>
<td>S-S-LRU</td>
<td>Pattern S1 shuffled, Small cache size, LRU replacement policy</td>
</tr>
<tr>
<td>S-ARC</td>
<td>Pattern S1, Small cache size, ARC replacement policy</td>
</tr>
<tr>
<td>S-S-ARC</td>
<td>Pattern S1 shuffled, Small cache size, ARC replacement policy</td>
</tr>
<tr>
<td>M-LRU</td>
<td>Pattern S1, Medium cache size, LRU replacement policy</td>
</tr>
<tr>
<td>S-M-LRU</td>
<td>Pattern S1 shuffled, Medium cache size, LRU replacement policy</td>
</tr>
<tr>
<td>M-ARC</td>
<td>Pattern S1, Medium cache size, ARC replacement policy</td>
</tr>
<tr>
<td>S-M-ARC</td>
<td>Pattern S1 shuffled, Medium cache size, ARC replacement policy</td>
</tr>
<tr>
<td>L-LRU</td>
<td>Pattern S1, Large cache size, LRU replacement policy</td>
</tr>
<tr>
<td>S-L-LRU</td>
<td>Pattern S1 shuffled, Large cache size, LRU replacement policy</td>
</tr>
<tr>
<td>L-ARC</td>
<td>Pattern S1, Large cache size, ARC replacement policy</td>
</tr>
<tr>
<td>S-L-ARC</td>
<td>Pattern S1 shuffled, Large cache size, ARC replacement policy</td>
</tr>
</tbody>
</table>

Table 9: Interpretation of x-axis for figure 20

For each pattern four graphs will be presented. Two graphs will show the hit ratio results with several cache sizes for a pattern. The last two graphs will display the performance (execution time numbers) with a 95% interval. With this 95% interval one can compare two measurements with a certainty of 95% (e.g. with 95% certainty measurement 1 is significant faster than measurement 2). The 95% is not explicitly given for the graphs with the hit ratios because the 95% interval is the same as the mean value ($\sigma = 0$, thus the interval equals $[\overline{X}, \overline{X}]$). Comparing runtimes and hit ratios between LRU and ARC in the subsections 4.8.x are based on the 95% interval.
4.8 Evaluation of results

4.8.1 Optimal concurrent connections

OpenLDAP utilizes a thread pool of 16 threads. Using S1 access pattern, the optimal number of concurrent connection with fastest execution time in our measurements is with 3-4 concurrent request as can be seen in figure on this page. This result can be explained by the performance of a single processor system. Only a few threads can be processed efficiently with a 'simple' processor. There are processors that are capable of handling more concurrent threads efficiently. Examples of these processors are multi-core processors (multiple cores on one chip, such as IBM Power5). Simple processor in this context means a processor, which can only process a few concurrent threads efficiently (such as an Intel Pentium 4 (no hyperthreading)). Thread handling is also dependent on the operating system that has been used. The Linux 2.4 kernel that has been used to measure the performance is known to have reasonable threading performance (for more information on 2.4/2.6 performance see [23]).

![Execution time with variable number of threads with pattern S1 (S1 tasks will be divided over all threads)](image)

Figure 19: Optimal concurrent connections

The disk IO probably becomes the bottleneck when more than four clients execute tasks. The OpenLDAP server needs to maintain transaction logging information and also retrieve/store the requested/mutated data. These actions will cause many request to the disk IO. The execution time is more or less stable
with more than 13-14 concurrent clients. At this point there is a constant queue of operations and the disk IO can’t deliver more data to the requesters. The disk IO latency bottleneck is also explained in [1].

4.8.2 Evaluation of pattern S1

Pattern S1 will request 1000 different elements in one chunk. There are 10 chunks in this S1 pattern. The hit ratio for the read-intensive setup is as expected. With a small cache size (e.g. 250 entries) LRU and ARC will need to swap more elements as it can only store 250 entries. Pattern S1 requests 1000 different records in a sequence. The graph in figure 20 also shows ARC has a slightly higher hit ratio than LRU with small cache size setting. With medium and large cache settings (1000 entries) the performance of LRU and ARC are similar. The medium/large settings allow both LRU and ARC to store most records in the cache. There’s also no significant difference between shuffled S1 pattern and non-shuffled S1 pattern.

![Graph showing hit ratio for LRU and ARC with different cache sizes and patterns.](image)

Figure 20: Read-intensive tests with S1 pattern, hit ratio per pattern

The results acquired with the write-intensive settings resembles the results of the read-intensive settings. The hit ratio with the small cache setting is significantly higher than with LRU. With medium/large settings ARC and LRU have no significant differences. Figure 21 also shows no significant difference between shuffled S1 and non-shuffled S1 pattern.
4.8 Evaluation of results

Figure 21: Write-intensive tests with S1 pattern, hit ratio per pattern

The execution time results for pattern S1 (read-intensive settings) with the multiple cache sizes and replacement policies are stable. Except for the small cache setting with LRU, the execution time is in between 50-55 seconds. The execution time for shuffled S1 pattern is noteworthy. It’s significantly faster than non-shuffled S1 although they have both a similar hit ratio. Apparently the shuffled S1 pattern causes less random disk IO. This might be related to the way BDB stores the internal values. The file-system can also pre-fetch near data items. Overall the shuffled versions of S1 have better runtime performance than non-shuffled versions. The graph in figure 22 also shows ARC and LRU are on par with each other except for LRU with small cache size setting. ARC with small cache size outperforms LRU significantly (also shown with hit ratio). ARC manages the cache with 250 entries more efficiently than LRU. With the LRU replacement policy, the cache will be flushed because of the 1000 different order items.
With the write-intensive settings the graph obtained is presented in figure 23. LRU with small cache size is slower than ARC with small cache size. With medium and large cache size the execution time is similar. LRU has a slight advantage over ARC with medium and large cache size. There is also a little
4.8 Evaluation of results

difference between the shuffled version and the non-shuffled version.

With pattern S1 ARC has a slight advantage over LRU with smaller cache sizes. The two replacement policies are on par with medium and large cache sizes.
4.8.3 Evaluation of pattern S2

Pattern S2 consists of one large sequence of 10000 (different) requests. There is also a shuffled version of S2. The S2 pattern will cause a flush on small, medium and large cache size setting. This is due to the size and the different numbers request in pattern S2 that is larger than the small, medium and large cache capacity.

Figure 24 illustrate the results with read-intensive settings. As expected the results are stable. The hit ratio is around 70%. An explanation for this 'high' hit ratio is the way a query is processed. Each read and write operation will use the access method described in subsection 4.1. Also there is always a 10% of authentication simulation. These simulations of an authentication also causes cache hits. On each request, an authentication action is performed, dn2id lookup is done, followed by an id2entry lookup. The last step is the actual retrieval of the element. Each step is being cached. LRU and ARC both perform equally (hit ratio-wise). There is virtually no difference between the shuffled version and non-shuffled version.

Using the write-intensive settings graph 25 is obtained. The results are as expected. The S2 pattern causes all caches to be flushed. The graph also shows a high hit ratio. This high hit ratio is also being caused by the way a query is processed. Performance of shuffled and non-shuffled version of S2 also don’t differ much from each other.

![Hit ratio chart](image)

Figure 24: Read-intensive tests with S2 pattern, hit ratio per pattern
4.8 Evaluation of results

Measuring execution time with the standard error yields figure 25. With the shuffled version of S2, a consistent faster execution time is achieved than the non-shuffled version. ARC and LRU both perform more or less the same except for the large cache size setting using the ARC replacement policy. Although the hit ratio are the same, the ARC run takes significantly longer to complete than the LRU version. A possible explanation for this behavior is the way ARC is implemented in OpenLDAP. Locks can be held over a longer period of time with ARC.

With the write-intensive settings graph 27 is obtained. The performance is similar to read-intensive settings. The shuffled version is faster on all runs than the non-shuffled version. ARC and LRU both performs at the same level. The only exception is the large cache size setting with ARC.

With pattern S2 the results are mostly similar. Only ARC with larger cache size perform significantly slower than its LRU counterpart. Though they do both have the same hit ratio. The same explanation about locking also applies here.
4.8.4 Evaluation of pattern S3

S3 pattern is similar to pattern S1, but it consist of chunks of 500 elements. It’s expected that the small cache size settings will perform worse than the medium
and large version. The small cache size setting can only hold 250 elements and therefore cannot accommodate the 500 element of pattern S1.

Figure 28 on the next page represents the results with the read-intensive settings in a graph. ARC and LRU have both similar results. The hit ratio for ARC with small cache size and shuffled pattern S2 is higher than the LRU equivalent. As expected the small cache size settings results in a lower hit ratio than the medium and large settings.

With the write-intensive settings there is virtually no difference between ARC and LRU. Additionally there is also no significant difference between the shuffled version of S3 and non-shuffled version of S3. Again with the small cache size settings, LRU and ARC both have a lower hit rate than the runs with the medium and large cache size settings. ARC with small cache size is also slightly faster than LRU with small cache size. The behavior has also been observed with the S1 pattern. With the medium and large cache size settings, both ARC and LRU achieve hit ratios close to 100%.
Figure 28: Read-intensive tests with S3 pattern, hit ratio per pattern

Figure 29: Write-intensive tests with S3 pattern, hit ratio per pattern

Using the read-intensive cache settings the execution times plotted in figure 30. As expected the cache settings with medium and large settings perform similarly. With small cache size both ARC and LRU require more time to complete the test runs. This is also reflected in the hit ratio of ARC/LRU with
small cache size. It’s noteworthy that both the ARC and LRU replacement policies perform better with the shuffled S3 pattern.

Figure 30: Read-intensive tests with S3 pattern, Execution time per pattern

Figure 31: Write-intensive tests with S3 pattern, Execution time per pattern

Using the write-intensive cache settings figure 31 is obtained. With the
write-intensive settings, LRU performs slightly better than ARC. LRU with small cache size performs slightly less than ARC with a small cache size. There is no significant difference between the shuffled version of the test runs and the non-shuffled version of the test runs.

With the S3 pattern LRU has a slight advantage over ARC. With medium and large cache size the results are as expected.

4.8.5 Evaluation of pattern L1

L1 pattern consist of chunks of 2000 request. In total there are 50000 requests in the L1 pattern. It’s expected that only the large cache setting will have a reasonable performance with hit ratios above 90%. Only the large cache setting has enough entries to accommodate the 2000 different request. The results plotted in figure 32 with read-intensive settings are as expected. ARC has a slight advantage over LRU achieving slightly higher hit ratios. There is no significant difference between the test runs with shuffled L1 and non-shuffled L1. With large cache size ARC and LRU both achieve hit ratios above 90%.

The results with the write-intensive settings are similar to those with the read-intensive settings. With medium cache size, ARC has a slight advantage over ARC. The test runs with the large cache size achieves a hit ratio above 90%. There is also no significant difference between the test runs with the shuffled and non-shuffled version of L1.

Figure 32: Read intensive tests with L1 pattern, hit ratio per pattern
4.8 Evaluation of results

The execution times with the read-intensive settings are represented in figure 34. LRU with small cache setting performs significantly slower than ARC with small cache size. It’s noteworthy that small cache size LRU and small cache size ARC perform similar with the shuffled version of the pattern. Overall the shuffled L1 pattern produces faster test runs than the non-shuffled L1 pattern. ARC achieves lower execution times than the LRU versions (non-shuffled).

With the write-intensive settings the results are closer to each other as shown in figure 35. With small cache size ARC slightly outperforms LRU with small cache size. With medium and large cache size the results are similar. The only exception is the test run with large cache size using the LRU replacement policy that outperforms ARC with the same settings. There is a slight difference between test runs with shuffled and non-shuffled versions of L1.

Overall ARC performs better with smaller cache size (execution time wise). LRU seems to perform better with large cache sizes.

Figure 33: write-intensive tests with L1 pattern, hit ratio per pattern
Figure 34: Read-intensive tests with L1 pattern, Execution time per pattern

Figure 35: Write-intensive tests with L1 pattern, Execution time per pattern

4.8.6 Evaluation of pattern C1

Pattern C1 consist of 50000 requests. The requests consist of a combination of S3 and L1 pattern. The expectation is that only the large cache settings will
handle this pattern effectively. Small and medium cache sizes will be flushed by the L1 pattern.

Figure 36: Read-intensive tests with C1 pattern, hit ratio per pattern

ARC and LRU with large cache size both achieve hit ratios near 100%. With 50000 request, the systems actually processes a lot more request due the way the OpenLDAP works. One LDAP query generates at least three requests, which are all cached. Shuffled and non-shuffled test runs are performance wise more or less the same.
Figure 37: Write-intensive tests with C1 pattern, hit ratio per pattern

The results with the write-intensive settings in figure 37 are similar to the read-intensive settings in figure 36.

Figure 38: Read-intensive tests with C1 pattern, Execution time per pattern

The measured execution times reveal more differences. ARC with small cache size outperforms LRU with small cache size (non-shuffled C1). The ex-
execution times with the shuffled pattern doesn’t show large differences. Medium and large settings perform more or less the same. The test runs with the non-shuffled version of C1 is consistently faster than the non-shuffled versions.

![Graph](image.png)

**Figure 39: Write-intensive tests with C1 pattern, Execution time per pattern**

The performance with the write-intensive settings is similar to the read-intensive settings. Again LRU with small size is significantly slower than it’s ARC counterpart. Overall ARC performs better than LRU with smaller cache sizes and performance with medium and large cache sizes are more or less the same.

### 4.8.7 Evaluation summary

Due to the software bug in BDB, several replacements algorithms have been studied. The flush of the cache with LRU was identified as a problem using certain access patterns. ARC was chosen and performs well (hit ratio wise). ARC is more complex and harder to implement correctly than LRU. But on the other hand ARC can cope with changing patterns, and it can withstand cache flushes. ARC is also preferable over LRU because it achieves significant lower amount of cache misses with traces S3, S3-S, C1 and C1-S than LRU (shown in table 11 on page 73). LRU achieves for traces L1, L1-S, S1 and S1-S a lower amount of cache misses than ARC. Cache misses are costly because the require a disk read action. ARC in PostgreSQL also shows encouraging performance improvements [19].

The measured runtime performance shows the potential of ARC with certain traces. With traces S1 (small cache setting, read and write intensive), S3 (small
cache setting, write intensive), L1 (all read intensive and some write intensive) and C1 (small cache setting, read intensive), this non-optimal implementation is already faster than the proven/tested LRU implementation. LRU also outperforms ARC on several traces which means ARC isn’t always better than LRU. ARC is more effectively with small cache sizes and cache flush patterns. These cache flush patterns do occur in practice (backup, restore and replicating for example all cause a cache flush).

5 Evaluation of OpenLDAP documentation

This section will describe the documentation resources available to both end users and application programmers who will use OpenLDAP. The documents will be classified as developer documentation or user documentation. Developer documentation and user documentation will be evaluated using requirements/methods proposed by[25].

5.1 User documentation

The user documentation consists of the following components:

- Administration guide
- Quick start guide
- FAQ (Frequently Asked Questions)

There are several documentation functionalities that will be explored in this subsection. The first documentation functionality is the introduction functionality. The purpose of this function is to introduce the user with the program. Methods such as guided tour (simulation), demos and introduction guide are proposed by [25] to support this function. OpenLDAP has no extensive guided tours where a user can to know the system. The Administration guide has some brief examples that can help to introduce the user to the system. More examples are required for a good introduction. The quick start guide can be regarded as a small installation guide. It does help to understand the installation procedure. The FAQ consists of a number of common questions with common answers. The FAQ contains a few introductory questions and answers about OpenLDAP and they do help to understand the system. The FAQ doesn’t contain examples on how to perform simple queries. In practice, a user needs to look at several places (FAQ, Admin guide, quick start guide) to be introduced with the system. It’s recommended to have a introduction guide where the relevant information are collected.

The second document functionality is to teach the user how he/she can use this program. Methods to support this functionality can be courses and example databases. An example database is a collection of examples that the user can use to do certain task. The Administration guide has some examples of common
The third documentation functionality is to support the use with the tasks he/she has to perform. Methods to support these functionalities are user guides, on-line help, helpdesks and wizards. The FAQ can be used to support the user. The FAQ information is minimal. There are no examples on how to perform the backup. The FAQ refers to slapcat(8), which is a man page, but it isn’t explained. There are no wizards (its all command-line interface) and no helpdesks. There is a mailing list that can be regarded as a semi helpdesk. This user list is highly crowded and it has a large volume of emails that can make it hard to look/ask for specific information. The Administration guide can be regarded as a user guide. It does have information on how tasks are performed but it is very minimal (for example there are no modify entry examples/descriptions).

The last documentation functionality is the reference functionality. Methods to support these functionalities are reference guides, on-line help and helpdesk. OpenLDAP has no reference guide and no helpdesk. The mailing list can be used to retrieve certain information, but the mailing-list is high volume and finding relevant information is hard. The manual pages is normally a resource for developers. They also contain global information and can be used as reference.

5.2 Developer documentation

The OpenLDAP developer documentation set consist of the following components:

- Manual pages
- Drafts
- RFC (Request For Comments)

The users of developer documentation are developers who need more detailed information. Details such as design and implementation choices are helpful to these users. The same documentation functionalities will be used to evaluate the developer documentation.

The first documentation functionality is the introduction functionality. There is no introduction documents for developers. New developers are expected to read the RFC documents. Methods such as guided tours, demos and introduction guide are not present.

The second documentation functionality is to teach developers how to use the system and how to write programs with it. Methods such as courses and example databases are absent with OpenLDAP. There are no programming examples and developers are again suppose to read the RFC documents. On the other hand OpenLDAP implements the standardized API and this API is documented in the RFC. A programming tutorial would greatly help new developers.

The supporting function of the documentation is more or less present. A developer’s mailing list exists where developers can discuss problems and related
materials. This list is strictly moderated and developers are often redirected to the users mailing-list. There are some drafts documents on implementation details of OpenLDAP. The developer guides consist of man pages and some drafts. The draft documents are document created by the OpenLDAP developers. These drafts contain implementation details. However these sets of drafts are not complete and therefore they cannot act as a complete developers guide.

The last documentation functionality is the reference functionality. The RFC documents, drafts and man pages can be used as reference documentation. The developers mailing list can also be used to lookup certain information. This list has less volume than the user mailing-list.

5.3 Summary

The user documentation supplied with OpenLDAP is useful but as shown in the previous subsection, its not complete and not as effective as it can be. The document set is sufficient but at certain points it misses information (how does one modify a entry, how to use slapcat), which can be an obstruction to people who wish to start with OpenLDAP.

The developers documentation are of less quality than the user’s documentation. There are no introductory documents and there are no real “getting to know the system” documents (programming-wise). The support and reference documents are more or less present but they are incomplete and poorly structured. There are too few examples on how to write programs with OpenLDAP. (e.g. important flags such as how to specify protocol version are absent from the man page)

Overall the user part of the documentation is at an acceptable level but needs to be organized properly (e.g. parts of an introduction guide are located at several sources). Maintenance of this information is not done frequently. The documentation for application programmers (or programmers who want to extend/modify) OpenLDAP is a severe problem.
6 Conclusions en recommendations

OpenLDAP can support write-intensive applications. The data store/ modification performance is acceptable (With all access patterns OpenLDAP achieves a performance of more than 10 operations per second). The performance can be improved by ARC because ARC prevents certain cache flushes and can adapt to different kind of workload. ARC is also more efficiently with smaller cache sizes. Due to bugs present in BDB, it is recommended to use BDB version (≥ 4.2.53). Other software projects based on BDB should also avoid using old versions of BDB. OpenLDAP also offers a certain degree of data integrity which is a good property to have with systems such as the reseller system.

The documentation of OpenLDAP needs a lot of work. It is minimal and often information is scattered over several sources. A systematic approach (such as [25]) is recommended to improve the documentation set. Software acceptance is also dependant on good documentation. A viewpoint that seems common in the software engineering is that the actual software writing practice is the core of the work, and that documentation is a side issue, a necessary evil. Documentation is part of a software project and must not be regarded as a side issue. Overall OpenLDAP has flexible data structures and offers sufficient performance to implement read intensive and write intensive applications. The documentation however is seriously lacking and improvements in this part of the package can positively influence the acceptance of OpenLDAP.
Appendix

Statistical data

<table>
<thead>
<tr>
<th>Run</th>
<th>Measured time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>3</td>
<td>59</td>
</tr>
<tr>
<td>4</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>59</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>61</td>
</tr>
<tr>
<td>9</td>
<td>59</td>
</tr>
<tr>
<td>10</td>
<td>62</td>
</tr>
<tr>
<td>11</td>
<td>62</td>
</tr>
<tr>
<td>12</td>
<td>59</td>
</tr>
<tr>
<td>13</td>
<td>60</td>
</tr>
<tr>
<td>14</td>
<td>58</td>
</tr>
<tr>
<td>15</td>
<td>61</td>
</tr>
<tr>
<td>16</td>
<td>61</td>
</tr>
<tr>
<td>17</td>
<td>60</td>
</tr>
<tr>
<td>18</td>
<td>60</td>
</tr>
<tr>
<td>19</td>
<td>61</td>
</tr>
<tr>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>21</td>
<td>60</td>
</tr>
<tr>
<td>22</td>
<td>61</td>
</tr>
<tr>
<td>23</td>
<td>58</td>
</tr>
<tr>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td>25</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 10: 25 measurements of pattern S1 with LRU replacement policy
Absolute cache misses per pattern

Table 11 lists the misses and total access to the cache in the format “misses / total access”. Only the cache misses with small cache size are listed. With medium and large cache size, ARC and LRU don’t differ much.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>LRU read int.</th>
<th>ARC Read int.</th>
<th>LRU write int.</th>
<th>ARC write int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>9002/29014</td>
<td>6765/31714</td>
<td>9002/29014</td>
<td>9878/34414</td>
</tr>
<tr>
<td>S1-S</td>
<td>9002/29014</td>
<td>6751/31714</td>
<td>9002/29014</td>
<td>9560/34414</td>
</tr>
<tr>
<td>S2</td>
<td>9002/29014</td>
<td>9010/29014</td>
<td>9002/29014</td>
<td>9008/29014</td>
</tr>
<tr>
<td>S2-S</td>
<td>9002/29014</td>
<td>9008/29014</td>
<td>9002/29014</td>
<td>9012/29014</td>
</tr>
<tr>
<td>S3</td>
<td>9002/29014</td>
<td>3908/31864</td>
<td>9002/29014</td>
<td>5736/34714</td>
</tr>
<tr>
<td>S3-S</td>
<td>9002/29014</td>
<td>3897/31864</td>
<td>9002/29014</td>
<td>5867/34714</td>
</tr>
<tr>
<td>L1</td>
<td>9002/29014</td>
<td>7966/31414</td>
<td>9002/29014</td>
<td>11242/33814</td>
</tr>
<tr>
<td>L1-S</td>
<td>9002/29014</td>
<td>8030/31414</td>
<td>9002/29014</td>
<td>11011/33814</td>
</tr>
<tr>
<td>C1</td>
<td>45002/145014</td>
<td>23010/159414</td>
<td>45002/145014</td>
<td>34392/173814</td>
</tr>
<tr>
<td>C1-S</td>
<td>45002/145014</td>
<td>22985/159414</td>
<td>45002/145014</td>
<td>34370/173814</td>
</tr>
</tbody>
</table>

Table 11: Absolute cache misses per pattern

Acknowledgement

I would like to express my gratitude to my supervisors Ling Feng, Djoerd Hiemstra and Rick van Rein for supplying me with invaluable hints and tips. This report would not have been possible without their help. I would also like to thank Daniel Knoppel, Binh Chu and Loan Chu for proofreading this thesis and helping me improve it.
References


REFERENCES


