Evaluation and Comparison of BIND, PDNS and Navitas as ENUM Server

Charles Shen and Henning Schulzrinne
Department of Computer Science
Columbia University
{charles, hgs}@cs.columbia.edu

Columbia University Computer Science
Technical Report CUCS-029-06

September 2006

Abstract

ENUM is a protocol standard developed by the Internet Engineering Task Force (IETF) for translating the E.164 phone numbers into Internet Universal Resource Identifiers (URIs). It plays an increasingly important role as the bridge between Internet and traditional telecommunications services. ENUM is based on the Domain Name System (DNS), but places unique performance requirements on DNS server. In particular, ENUM server needs to host a huge number of records, provide high query throughput for both existing and non-existing records in the server, maintain high query performance under update load, and answer queries within a tight latency budget. In this report, we evaluate and compare performance of serving ENUM queries by three servers, namely BIND, PDNS and Navitas. Our objective is to answer whether and how these servers can meet the unique performance requirements of ENUM. Test results show that the ENUM query response time on our platform has always been on the order of a few milliseconds or less, so this is likely not a concern. Throughput then becomes the key. The throughput of BIND degrades linearly as the record set size grows, so BIND is not suitable for ENUM. PDNS delivers higher performance than BIND in most cases, while the commercial Navitas server presents even better ENUM performance than PDNS. Under our 5M-record set test, Navitas server with its default configuration consumes one tenth to one sixth the memory of PDNS, achieves six times higher throughput for existing records and an order of two magnitudes higher throughput for non-existing records than the bottom line PDNS server without caching. The throughput of Navitas is also the highest among the tested servers when the database is being updated in the background. We investigated ways to improve PDNS performance. For example, doubling CPU processing power by putting PDNS and its backend database in two separate machines can increase PDNS throughput for existing records by 45% and that for non-existing records by 40%. Since PDNS is open source, we also instrumented the source code to obtain a detailed profile of contributions of various systems components to the overall latency. We found that when the server is within its normal load range, the main component of server processing latency is caused by backend database lookup operations. Excessive number of backend database lookups is the reason that makes PDNS throughput for non-existing records its key weakness. We studied using PDNS caching to reduce the number of database lookups. With a full packet cache and a modified cache maintenance mechanism, the PDNS throughput for existing records can be improved by 100%. This brings the value to one third of its Navitas counterpart. After enabling the PDNS negative query cache, we improved PDNS throughput for non-existing records to the level comparable to its throughput for existing records, but this result is still an order of magnitude lower than the corresponding value in Navitas. Further improvements of PDNS throughput for non-existing records will require optimization of related processing mechanism in its implementation.

Contents

1 Introduction

Driven by both the cost savings and the great flexibility to provide new value-added services, an increasing number of carriers, enterprises are adopting Voice over Internet Protocol (VoIP) to replace traditional Public Switched Telephone Network (PSTN) services. However, VoIP identifies entities using URIs, rather than the traditional phone numbers everyone has been used to. Bridging the telephone numbers with URIs may therefore further accelerate the proliferation of VoIP among non-technical users. ENUM [1] is a protocol standard developed by the IETF for this purpose. It translates the global PSTN phone numbers, also known as E.164 numbers [2], into URIs. With ENUM, an entity in the IP world can be located by a traditional phone number. For example, a user can place a VoIP call to his friend by using the friend's usual phone number. ENUM relies on DNS, which was originally designed to map domain names into IP addresses.

ENUM stores information about mappings between the telephone number and the corresponding URI in the form of a special DNS record type. Because of the volume and type of ENUM data, as well as the related service quality expectation, ENUM places unique requirements and challenges on the existing DNS infrastructure components. Below we summarize some of the most important performance requirements for an ENUM server:

Accommodation of a huge number of records. To see why this is necessary, we may consider the total number of telephone numbers in the US. The 2005 *Trends in Telephone Services* published by FCC [3] reports about 170-180 million wireline local telephone loops as of December 2003, plus 170 million mobile subscribers as of June 2004. Today, the total number of phone numbers in US alone might be well over 400 million. Each of these number may be associated with one or multiple ENUM records, thus requiring over 400 million records to be stored. To provide ENUM service with a relatively small set of servers, a single server may need to store many more records than most of the current DNS servers do.

High query throughput or successful queries per second (qps) For example, if we assume a customer population of 100 million, which is around one third of the US population, and further assume that each one of them makes on average two calls per busy hour, then we will have to sustain an ENUM query rate of $50,000$ qps.

Short lookup response time The ENUM lookup latency is one component of the total VoIP call setup latency. In order to provide VoIP at a quality comparable to the traditional PSTN service, the total VoIP call setup signaling must be no longer than the traditional Signaling System 7 (SS7) call setup signalling in PSTN. ENUM lookup may be comparable to a routing function as part of the PSTN switch processing for ISDN User Part (ISUP) messages. According to [4], the total switch response time budget for such messages has a 205-218 ms mean and 337-350 ms in the 95th percentile. That means the total ENUM lookup time including ENUM server processing time and round trip time should take only part of this budget.

High query throughput under server database update load ENUM service providers may from time to time need to add new records into the database, or update existing records in the database. The ENUM server should not have to be restarted and the query service should not be significantly affected during the database update. Although some DNS servers provide dynamic DNS functionality, most ENUM providers today are much more likely to employ some background bulk update mechanisms for database update due to security concerns and the operation model. So the challenge is to maintain high query performance under background update load.

High query performance for non-existing records This is a special consideration that is somewhat different from the requirements for traditional DNS servers. Given the huge telephone number space and the absence of a universal ENUM server, any initial deployment of ENUM by one or multiple organizations will likely contain only a tiny fraction of the whole ENUM record space. Therefore, the probability of searching for a non-existing record in a particular ENUM server may well exceed that for retrieving an existing record in the same ENUM server.

Our work aims to provide a comprehensive study on ENUM performance to determine optimal system design for commercial-grade ENUM architecture capable of meeting projected requirements of telecommunications carriers and service providers likely to be encountered in large scale VoIP deployments. This study also provides an independent third-party verification of ENUM server performance testing conducted by the FiberNet Telecom Group [5] in association with FiberNet's PHONOMENUM VoIP Peering platform.

In this document, we report on the development of an ENUM benchmarking tool called enumperf, based on the Nominum modified open source DNS performance measurement software queryperf [6]. Using this tool, we derived and analyzed a series of test results according to each of the ENUM requirements mentioned above. We tested BIND [7], PDNS [8] and Nominum Navitas [9] servers. BIND is the traditional, open source DNS server that still dominates today's DNS deployment. PDNS is also open source and is one of the most popular substitute for BIND. Navitas is a high-end commercial name server produced by Nominum Inc for ENUM applications.

Among the three servers, we chose PDNS for a more thorough analysis. PDNS is more modern compared with BIND. The open source nature of PDNS makes it possible for us to add detailed profiling to the server, so we can analyze the contribution of various system components to the overall performance. We also study the performance limitations caused by server processing capacity and database scaling. We investigated how to improve PDNS performance by using PDNS caching and optimizing its cache maintenance mechanism.

The other two servers, BIND and Navitas, are generally treated as black boxes. We compared them with PDNS in various aspects, including query performance, database scaling, server processor capacity, server memory usage, and query performance under update load. Then we match the results with the ENUM requirements.

This independent study corroborates the FiberNet test results and presents them here in a forum accessible to the community of service providers engaged in migrating voice service from the PSTN to IP.

The rest of the report is organized as follows: Section 2 provides background knowledge on DNS, ENUM, common DNS servers and introduces related work. Section 3 describes the experimental setup, including clients, servers and the benchmarking tool. Section 4 presents detailed PDNS performance. Section 5 and Section 6 discuss BIND and Navitas performance, respectively. Section 7 summarizes and compares the main results of the three name servers serving ENUM queries. Finally, Section 8 offers concluding remarks.

2 Background and Related Work

2.1 Domain Name System

The domain name system is a distributed database organized as a tree with a hierarchical structure. Each node in the tree serves as the root node of a subtree of the overall tree. A subtree represents a partition of the overall tree and corresponds to a domain, which is indexed by a domain name in the DNS database. A domain name is a sequence of text labels on the path from the node to the root of the overall DNS tree, with dots separating the labels.

The root node of the overall DNS tree is the root domain. It has a null label as its name and is written in text as a single dot (.). A domain name with a trailing dot indicates that the name is written relative to the root and is a Fully Qualified Domain Name (FQDN). Otherwise, the name may be interpreted as relative to some non-root domain.

In the DNS tree hierarchy immediately following root, there are a number of Top Level Domains (TLDs), further divided into generic TLDs (gTLDs), such as .com, .edu, .net, and country code TLDs (ccTLDs), such as .us, .de, .cn., .in, as well as a special-purpose TLD for Internet infrastructure, .arpa [10]. The next step down the DNS hierarchy is the second level domains, e.g., columbia.edu, newyorktimes.com.

Each host in the Internet usually has a FQDN that points to its information such as IP address or mail server. Hosts may also have domain name aliases, which are pointers from another domain name to the hosts' official or canonical domain names (CNAME).

In order to achieve scalable and decentralized management of the DNS database, a domain in the DNS domain space is divided into many subdomains. Each subdomain may be administered independently by organizations responsible for the domain. This creates pieces of autonomously administered DNS name space called zones. Zone data are stored in zone data files in the form of an extensible data structure called Resource Records (RRs). Each RR consists of name, type, class, TTL and resource data. The name is a FQDN to which this RR pertains; class is typically the Internet (IN); TTL specifies the time that this RR may be cached by a caching server, type identifies the format of the resource data. A zone data file starts with one Start of Authority (SOA) record and one or more Name Server (NS) records, followed by other records. The SOA record indicates that the authority for the data within this zone. The NS records list name servers for this zone. Example types of other records are A record for host addresses and CNAME record for the host's canonical name for an alias. The following shows part of a zone data file containing these types of records:

```
;
; SOA record
;
enum.example.com. IN SOA ns.enum.example.com. root.enum.example.com. 5 10800 1800 604800 1800;
;
; NS records
;
enum.example.com. IN NS ns1.enum.example.com.
enum.example.com. IN NS ns2.enum.example.com.
;
; A records
;
ns1.enum.example.com. IN A 11.1.1.1
ns2.enum.example.com. IN A 11.1.1.2
bigbang.enum.example.com. IN A 11.1.1.8
;
; CNAME records and aliases
;
bb.enum.example.com. IN CNAME bigbang.enum.example.com
```

The SOA record starts with the `enum.example.com` domain name, indicating that the server containing this file is authoritative for the `enum.example.com` zone. The `IN` stands for Internet. The first name after `SOA`, `ns.enum.example.com` is the primary name server for the zone `enum.example.com`. The `root.enum.example.com` in the SOA record specifies the email address of the contact person in charge of the zone, when the first "." is replaced by an "@". The five values following `root.enum.example.com` is used for various DNS server data maintenance tasks. Next are the two NS records that list the two name servers, namely, `ns1.enum.example.com` and `ns2.enum.example.com` for the zone. The first two A records following the NS records give the IP addresses of the two name servers as `11.1.1.1` and `11.1.1.2`. The third A record represents another host in the zone, `bigbang.enum.example.com`, whose IP address is `11.1.1.8`. The last record is a CNAME record. It says that the `bb.enum.example.com` is an aliase for `bibbang.enum.example.com`. When the name server receives a DNS lookup for `bb.enum.example.com`, it first looks for the canonic name of the queried domain, `bigbang.enum.example.com` and then return the IP address of the canonical name.

DNS name servers manage zone data files and answer queries from DNS clients. The name server that keeps track of all information about the domain names in a zone is authoritative for the zone. This name server itself is the authoritative name server for the zone. An authoritative name server also has information about authoritative name servers of the subdomains rooted at the domain it manages.

An authoritative name server that reads the zone data from a file on its host is called the primary master server, or simply primary. Other authoritative name servers get the zone data from a different authoritative name server, and are called slave or secondary servers. The process of a slave server getting zone data from another server (either a master server or yet another slave server) is called zone transfer. Setting up multiple authoritative name servers including master and slaves offers redundancy and increases both reliability and scalability of the DNS.

DNS clients, also called resolvers, access the name servers to find information about domain names. This process is called name resolution. A common form of resolvers are the stub resolvers provided by the operating system as library routines. These stub resolvers can only perform the most basic name resolution tasks, such as sending a query, interpreting responses and returning it to the calling program, and re-sending the query if no response is received. Such queries are usually called recursive queries because the name server receiving such a query will possibly need to repeat the query toward multiple other name servers recursively until an answer is found. The name servers that serve recursive queries are called recursive name servers. During the resolution of a recursive query, the recursive name server may send out non-recursive (iterative) queries to target name servers closer to the queried domain, or to the configured root name servers, if no lower level target name servers are found. The name server that receive the non-recursive query will simply give the best answer it already has to the recursive name server without performing additional queries. The best answer may be the exact answer to the query, or the names and addresses of the referral name servers it knows that is closest to the queried domain name. The recursive name server may then choose to follow the referrals to continue the resolution process. In case the referral contains more than one name server, the recursive name server decides which one to use based on some network metric such as Round Trip Time (RTT). At the end of each query, the recursive name server receives an answer with a response code indicating the status of the result, for example, "No error (noErr)" for a successful query, "format error" for uninterpretable queries, "server failure (ServFail)" for queries unable to be processed due to server problem, "name error (NXDomain)" for non-existing reference query domain, "not implemented" for unsupported kind of query, and "refused" for queries that are not allowed.

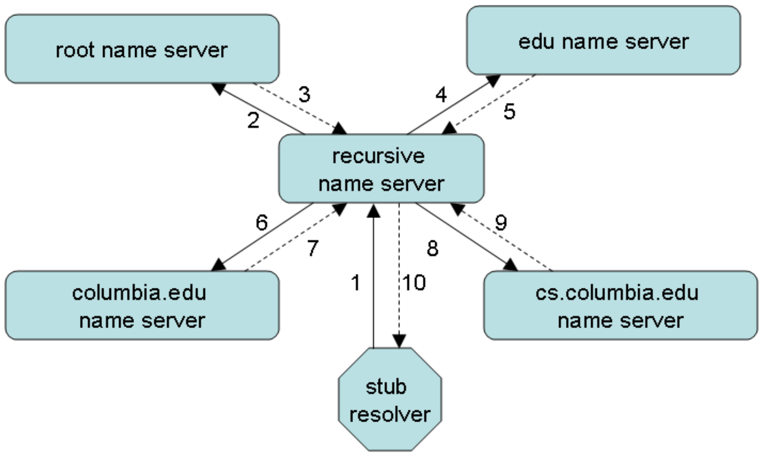


Figure 1: A recursive DNS query example

As an example, Figure 1 shows a recursive query process for the domain `foo.cs.columbia.edu`. The following steps are performed:

1. The stub resolver sends the query for the domain `foo.cs.columbia.edu` to a configured recursive name server;
2. the recursive name server queries one of the global root name servers that has been configured for the domain `foo.cs.columbia.edu`;
3. the global root name server refers the recursive name server to the top-level edu name server;
4. the recursive name server queries the edu name server for the domain `foo.cs.columbia.edu`;
5. the edu name server refers the recursive name server to the `columbia.edu` name server;
6. the recursive name server queries the `columbia.edu` name server for the domain `foo.cs.columbia.edu`;

7. the `columbia.edu` name server refers the recursive name server to the `cs.columbia.edu` name server;
8. the recursive name server queries the `cs.columbia.edu` name servers for the domain `foo.cs.columbia.edu`;
9. the `cs.columbia.edu` name server, which is authoritative for the domain `foo.cs.columbia.edu`, returns the answer to the recursive name server;
10. the recursive name server forwards the answer for the domain `foo.cs.columbia.edu` to the stub resolver.

Finally, the way name servers are deployed is determined by zone administrators. Usually, multiple name servers are used to serve a zone to improve service reliability and security. Since a name server that answers recursive queries can be separate from an authoritative server, a set of authoritative-only servers with recursion disabled can be deployed together with recursive-only name servers that are not authoritative for any zones. While the authoritative name servers provide authoritative name service to the Internet at large, the recursive-only name servers may only provide recursive service to local clients and do not need to be reachable from the Internet. These recursive-only name servers usually cache the results to speed up the lookup process and are also called caching servers. Caching can be applied to both positive and negative results. The length of time for which a record may be retained by a caching server is controlled by the TTL field associated with each resource record.

2.2 ENUM and DNS NAPTR Records

ENUM is a technology that maps E.164 telephone numbers used in PSTN network to URIs used in the Internet. By bridging two well-established conventions - E.164 number system and DNS, ENUM facilitates the convergence of Internet services with existing telecommunication network services, enabling much faster adoption of voice over IP, video over IP, fax over IP and many other services.

Since the translation between telephone numbers and domain names is analogous to the translation between IP addresses and domain names, it is natural to build ENUM on top of existing DNS technology. The original goal of ENUM is to establish a global, universally available telephone number directory in the domain name system. For this reason, a special second level DNS infrastructure domain called `e164.arpa` is allocated to ENUM as its root domain. Under this root domain, each country will manage registration for numbers falling in its own country code. There are three tiers of entities in ENUM. Tier 0 entities currently include the International Telecommunications Union (ITU) [11] and RIP NCC [12]. They are responsible for the root ENUM domain `e164.arpa`. Tier 1 entities are country-level ENUM registries. Tier 2 entities contain the registrars within each country code. After registering the numbers via the registrars, users can edit and modify the information associated with their telephone numbers. Currently only a few countries, such as Austria, have implemented ENUM for public use. Quite a few other countries are carrying out ENUM field trials, including the United States. Implementation of the global ENUM, however, has been slow due to major hurdles caused by organizational, legal, and political reasons. Therefore, carriers and Multi-Service Operators (MSOs) such as cable companies have started to develop their own ENUM. To differentiate these ENUM implementations from the global ENUM originally envisioned, the global ENUM is called public ENUM, while these more limited-scope ENUM is called infrastructure ENUM. Unlike public ENUM where the numbers are always rooted in the `e164.arpa` domain, numbers in infrastructure ENUM are mapped to different subtrees in DNS.

The management of ENUM records is the same as domain management in DNS. Authoritative ENUM root name servers for the `e164.arpa` domain contain referrals pointing to authoritative name servers which serve the next level of the ENUM domain divided by the country codes. For example, the nations using North American Numbering Plan, the United Kingdom, and China have country code `1`, `44`, `86`, so their respective root ENUM domains are `1.e164.arpa`, `4.4.e164.arpa`, and `6.8.e164.arpa`. The ENUM delegation below that is entirely the policy of the country that owns the country code. Large countries may choose to partition all numbers under the country code according to area code. Small countries may choose not to partition at all and leave all numbers under the country code as a single zone.

In the ENUM database, information that associates a telephone number with URIs is stored as DNS NAPTR RRs [1]. NAPTR RR is one type of DNS resource record that provides mapping from a domain to services. In order to retrieve the associated NAPTR RR for an E.164 telephone number, the telephone number is first converted to its domain name format. This is done by reversing the digits of the phone number, inserting dots between each, and appending the appropriate domain name at the end. In case of public ENUM, all converted domain names end in `e164.arpa`. In the following example, the first line contains the target domain name `4.3.2.1.9.3.9.2.1.2.1.e164.arpa.`, which is converted from +1 212 939 1234. The next three lines are the NAPTR records retreived for that domain name. The meaning of the various fields in the NAPTR record is explained as follows:

```
@ORIGIN 4.3.2.1.9.3.9.2.1.2.1.e164.arpa.
IN NAPTR 10 100 "u" "E2U+sip" "!^.*$!sip:info@enum.example.com!" .
IN NAPTR 10 101 "u" "E2U+h323" "!^.*$!h323:info@enum.example.com!" .
IN NAPTR 10 102 "u" "E2U+msg" "!^.*$!mailto:info@enum.example.com!" .
```

- The IN in the first column indicates the Internet type. The NAPTR in the second column indicates that the resource record is of type NAPTR.
- The number 10 in the third column is the order field, specifying the order in which the NAPTR records must be processed. If these values are the same across different rows, no order needs to be followed.
- The numbers 100, 101 and 102 in the fourth column is the preference field, indicating the order in which NAPTR records with equal order values should be processed.
- The `'u'` in the fifth column is a flag which means the record is terminal and the output is a URI.
- The sixth column contains service parameters. The service parameters starts with a `'E2U'`, followed by one or more ENUM services types. Examples of ENUM service types include SIP, H323, messaging (MSG) and presence (PRES).
- The string in the seventh column is a regular expression, whose grammar is specified in RFC 2915 [13]. It is a substitution expression that is applied to the query string to construct the next domain name to lookup. In the first NAPTR record of the above example, the `'!'` is the delimiter character. `'^.*$'` maps to the whole input domain name string, and the output will be `sip:info@enum.example.com` which is a URI used to contact the queried telephone number.
- The last column is the replacement field. It contains the next NAME to query for NAPTR, SRV, or address records depending on the value of the flag field. In the above example the replacement field is null.

The results of the above three NAPTR records tell the ENUM client that the domain `4.3.2.1.9.3.9.2. 1.2.1.e164.arpa.` is preferably contacted by SIP via the SIP URI `sip:info @enum.example.com`, secondly by H.323 via H.323 URI `h323:info@enum.example.com`, and thirdly by SMTP at `info@enum. example.com` for email.

In short, ENUM is basically a mapping between a domain name derived from a E.164 telephone number to one or more URIs. The mapping information is stored as NAPTR records in a DNS server. The original vision of a universal ENUM database, or public ENUM, still seems to be a long way to go due to legal and regulatory obstacles. Infrastructure ENUM is an ENUM implementation that uses essentially the same technology but bypasses the main hurdles of public ENUM. It is gaining a lot of momentum within carriers and MSOs.

2.3 BIND, PDNS and Navitas

ENUM is built on the DNS infrastructure. In this section we briefly review the main features of three of the existing DNS servers.

BIND [7] is one of the earliest implementation of the DNS protocol and provides an openly redistributable reference implementation of the DNS. It contains both authoritative server functionalities and recursive server functionality. It supports most major components including relatively new ones such as dynamic DNS update and DNS SECurity (DNSSEC) [14–18]. Today BIND is maintained by the Internet Software Consortium. BIND is by far the dominant name server software used on the Internet. It has been proven to be a robust and stable software for traditional DNS needs of most organizations. However, it has a number of limitations. First, since it stores all data in memory, the amount of memory becomes a bottleneck as the data size grows. Second, since records are stored as text zone files, both the initial loading of the records and the latency in querying the records increases significantly as the size of the data increases. Third, update of the server records requires reloading of the entire zone file. During the reloading, it cannot answer queries. Fourth, BIND supports dynamic DNS update but the quantity and types of data that can be updated are limited. These properties make BIND unsuitable for carrier grade operation where scalability, reliability, high performance, and high availability are the keys.

PDNS [8], open source since 2001, is a modern and advanced DNS software developed by the company with the same name. It is mainly an authoritative only server but also contains certain recursive functions. It implements most DNS resource record types including SRV, NAPTR, DNSKEY, full master and slave support. As of this writing, it does not support DNSSEC and dynamic DNS update yet. One of its major differences from BIND is that it can not only support traditional BIND zone files, but also interface with most common database backends such as MySQL, PostgreSQL, LDAP, DBMS, and Oracle. Dealing with databases with standardized interfaces makes the record much more manageable than editing text files, especially when the number of records is large. PDNS server is also capable of caching both packets and database queries. These properties, together with the more concise but efficient code written from scratch, make PDNS a good candidate to deliver higher and more scalable performance than BIND. As one of the most popular BIND substitutes, PDNS is serving top-level domains including `.mn`, `.mp` and `.tk`, all Wikipedia sites, Tucows, E164.org (ENUM), Siemens (ENUM trial), Register.com, and Ascio. It is used by 10% to 20% of all domains globally, and 50% of the domains under `.de` [19].

Navitas [9] is a commercial name server produced by Nominum Inc for ENUM applications. The company consists of the team that developed the open source BIND 9. The team noticed that BIND is a multifunctional server containing different aspects of protocol features, and it is not optimized for any specific component. Therefore, they developed from scratch several product implementation that are highly optimized for a specific function, such as the Navitas server for ENUM, ANS server for authoritative domain name service.

2.4 Related Work

Benchmarking of ENUM server has been reported recently by Nominum Inc. [20]. The Nominum work tested four DNS implementations: ANS, BIND, PDNS-BIND and DJBDNS. Their testbed contains one server and five client machines all running Enterprise Linux 3.0. Each client machine runs one queryperf process using a small fixed portion of the whole database as query space. Each test is run two times to allow any advance caching heat up. The work reported throughput and response time results under a fixed load with 200M, 50M and 10M record set for ANS, and 10M record set for BIND and DJBDNS. It also tested query performance in the presence of dynamic DNS update traffic.

We tested BIND, PDNS, ANS, and Navitas. Our configuration for PDNS is completely different from [20]. Nominum tested PDNS with its BIND backend, citing response time considerations. But PDNS-BIND scales badly. As a result, all their tests for PDNS failed. We found that response time is not a real concern on our platform and chose to evaluate PDNS with its more scalable MySQL backend. Both ANS and Navitas are products of Nominum. We do not observe notable performance difference in our tests for ANS and Navitas when using the same Nominum *vdb* database driver. In this report, we only present performance results for Navitas because the vendor specifies that “Though its roots come from the Authoritative Name Server

ANS, Nominum has focused development on Navitas specifically for the ENUM application environment. Nominum Inc. does not support the use of ANS in ENUM applications and restricts these completely to the domain of Navitas". Navitas supports more database driver types than ANS does.

Our test methodology also differs significantly from that of [20]. First, unlike the Nominum work which treats all servers as black boxes, we instrumented the PDNS server code to determine the contributing system components for a better understanding of the overall performance. Second, we randomized the query space in all tests instead of using a small fixed portion of the query space, and looked at how server transforms from its normal working condition to the saturation point instead of checking one operation point only. Third, we measured the performance also for non-existing records rather than just for existing records. Forth, we determined the limitation of CPU processing power on the servers, computed the memory utilization efficiency, tested query performance under background update load other than dynamic DNS, none of which are covered by [20].

Due to the many differences in the tests, it is hard to directly compare results of [20] to ours. For a couple of values that we may roughly compare after some extrapolation, including the throughput of BIND and ANS, we do find the results within the same order of magnitude.

3 Experimental Setup

3.1 Hardware Profile

Our experimental system consists of three Linux machines all running Red Hat Enterprise 3.4.4 with kernel version 2.6.9. Two of them have four 3 GHz Intel Xeon CPUs, 8 GB memory, and a 150 GB hard drive. The third one has two 3 GHz CPUs with 2 GB memory, and a 80 GB hard drive. The three machines were connected via 100 Mbps Ethernet connections. In most of our tests, the server is running on one of the 4-CPU machines. In the case where PDNS and its database backend are located on separate machines, both of the 4-CPU machines are used as server. The machine with two CPUs is used as the client load generator all the time. Depending on the required load, one of the 4-CPU machines that have spare CPU capacity is also used to generate client load.

3.2 ENUM Server

We evaluated the performance of BIND, PDNS and Navitas as ENUM servers. Our choice of these three software is not meant to deprecate any other existing name server implementations. Rather, we tried to pick a small group of typical name server implementations that represent a mix of factors, including large user population, traditional and modern flavor, open source and commercial product, multifunctional and dedicated architecture. To this end, the above three name servers seem to be a good combination. The version of the software we tested are BIND 9.2.4, PDNS 2.9.19 and Nominum Navitas 2.6.0.0.

Among the three tested servers, we further choose PDNS for a more thorough analysis because it has apparent scaling advantage over BIND. Unlike Navitas, PDNS is open source so we are able to treat it as a white box and instrument detailed profiling in the source code. PDNS supports different database backends, we have chosen the popular open source MySQL database backend in our tests. The MySQL version running in the server machine is 4.1.12.

Navitas is a dedicated name server for ENUM. Both BIND and PDNS can serve as authoritative and recursive name server. Since our goal is to test the behavior of authoritative name servers, we turned off the recursion function in BIND and PDNS in all cases. More details about individual server configurations will be supplied along with the description of the specific tests where applicable.

3.3 ENUM Clients and Enumperf

The ENUM load generator we used is the Nominum-modified open source queryperf-Nominum-2.1 [6]. Queryperf is widely used for DNS benchmarking and is also used in Nominum's ENUM benchmarking efforts. Queryperf is designed for measuring the performance of authoritative DNS servers. It can simulate multiple clients with a single process. Each client works in a closed loop fashion in that the next query is only generated after the response to the previous query has been processed. The default number of clients each queryperf process generates is 20. However, the load generated in this default configuration is way too much in most of our tests. In order to load the server from a relatively light level to a more stressful level, we set each queryperf process to simulate one client only. Then we gradually increase the load level by increasing the number of queryperf processes, which is equivalent to the number of clients in our case. There are a couple of caveats that must be noted in using a closed loop traffic generator like queryperf for performance evaluation. First, the actual query load generated with the same number of clients is usually not the same for different server configurations. Second, as the server approaches a saturation point, continuing to increase number of clients can result in the subsequent total load generated fluctuating around the server maximum capacity because the server working condition may not be completely stable at that point.

To better serve our tests, we also implemented an extension to queryperf. The existing queryperf takes input from either standard input or a text file containing queries. Our extended queryperf has the option to generate random queries within a range of ENUM query domains. For example, if the range of phone numbers is from +1-212-939-0000 to +1-212-939-9999, queryperf can generate uniformly distributed queries for domains from `0.0.0.0.9.3.9.2.1.2.1.e164.arpa.` to `9.9.9.9.9.3.9.2.1.2.1.e164.arpa.`

Based on our extended queryperf, we developed the enumperf test-suite. It consists of a set of perl and shell scripts running on the Linux platform and uses the Unix program rexec for executing processes on remote machines, modelled on the SIP server benchmarking tool called SIPStone [21]. The enumperf test-suite includes common configuration scripts for test settings, main execution scripts that define testing procedures for specific test groups, resource monitoring scripts, output parsing and processing scripts and a specific test description script for the particular test. Most of the tests we performed contain multiple steps with increasing number of queryperf processes running in each step. This information can be supplied in the test description script by specifying the number of queryperf processes in the initial step, the number of queryperf processes to be added in each step, the number of total steps to be run, and the time duration of each step. On our three-machine testbed, we deploy enumperf as shown in Fig. 2. The ENUM server is running on one of the 4-CPU machines denoted as server system. The other two machines run as client systems. One of the client systems runs the Bench Master script. Execution of the Bench Master script causes its host system to connect to appropriate client and server systems to start resource monitoring and create the desired number of ENUM clients at pre-defined intervals during each test step. When all test steps are finished, the Bench Master collects all results data from client and server systems and processes them to produce statistics. In the cases where PDNS and its database backend need to be on separate machines, we use both of the 4-CPU machines as server systems.

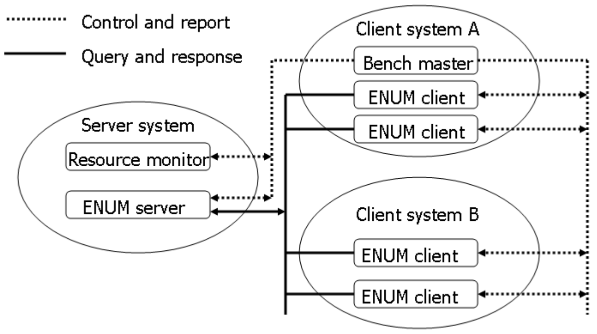


Figure 2: Enumperf testbed architecture

Set	Starting Record	Ending Record
500k	`0.0.0.0.0.0.1.6.4.6.1.e164.arpa`	`9.9.9.9.4.9.1.6.4.6.1.e164.arpa`
5M	`0.0.0.0.0.0.0.2.1.2.1.e164.arpa`	`9.9.9.9.9.4.9.2.1.2.1.e164.arpa`
20M	`0.0.0.0.0.0.0.0.1.3.1.e164.arpa`	`9.9.9.9.9.9.4.3.1.3.1.e164.arpa`

Table 1: Records in each record set

Set	Starting Zone	Ending Zone
500k	`0.1.6.4.6.1.e164.arpa`	`9.1.6.4.6.1.e164.arpa`
5M	`0.2.1.2.1.e164.arpa`	`9.2.1.2.1.e164.arpa`
50M	`0.1.3.1.e164.arpa`	`3.1.3.1.e164.arpa`

Table 2: Zones in each record set

3.4 Database Record Sets

We used three ENUM record sets of different scales: 500k, 5M and 20M. All records are generated using a perl script. The starting and ending records of each set are shown below in Table 1.

Records in the 500K and 5M record set are divided into ten zones each consisting of 1/10 of the total records. Records in the 20M record set is divided into four zones each consisting of 1/4 of the total records. The starting and ending zones for each record set are listed in Table 2.

Each zone file starts with an SOA record and two NS records for the zone, as well as two A records for the NS records shown below:

```
@ IN SOA ns.enum.example.com. root.enum.example.com. 5 10800 1800 604800 1800;
@ IN NS ns1.enum.example.com.
@ IN NS ns2.enum.example.com.
ns1 IN A 11.1.1.1
ns2 IN A 11.1.1.2
```

The rest of the records in the zone file are all NAPTR records. For example, the first zone file for the 5M record set contains 500k NAPTR records for domains from `0.0.0.0.0.0.0.2.1.2.1.e164.arpa` to `9.9.9.9.9.4.2.1.2.1.e164.arpa`. Each domain has exactly one NAPTR record in our testing zone files. The records look like the following:

```
0.0.0.0.0.0.0.2.1.2.1.e164.arpa. IN NAPTR 0 0 "u" "sip+E2U" "!^.*$!sip:0000000" \
host212-1.enum.example.com.
0.0.0.0.0.1.0.2.1.2.1.e164.arpa. IN NAPTR 0 0 "u" "sip+E2U" "!^.*$!sip:0100000" \
host212-2.enum.example.com.
0.0.0.0.0.2.0.2.1.2.1.e164.arpa. IN NAPTR 0 0 "u" "sip+E2U" "!^.*$!sip:0200000" \
host212-3.enum.example.com.
0.0.0.0.0.3.0.2.1.2.1.e164.arpa. IN NAPTR 0 0 "u" "sip+E2U" "!^.*$!sip:0300000" \
host212-4.enum.example.com.
0.0.0.0.0.4.0.2.1.2.1.e164.arpa. IN NAPTR 0 0 "u" "sip+E2U" "!^.*$!sip:0400000" \
host212-5.enum.example.com.
...
```

4 PDNS Performance

4.1 Basic PDNS Server Configurations

We chose the popular open source MySQL as PDNS database backend with InnoDB as the storage engine, as recommended in the PDNS manual. The optimization issues for MySQL InnoDB storage engine are discussed in the MySQL manual [22]. The two main parameters of MySQL InnoDB engine are *innodb_buffer_pool_size* and *innodb_thread_concurrency*. The *innodb_buffer_pool_size* is the total size of the memory buffer for InnoDB to cache data and indexes of its tables, thus minimizing disk I/O operations. Its default value is 8 MB. Since our system has a relatively large memory size of 8 GB, we set the *innodb_buffer_pool_size* to 4 GB. This value is large enough to hold both the data and index files in our experiments using the 5M and 500k record sets, but not for the 20M record set. The *innodb_thread_concurrency* value limits the number of operating system threads that are existing concurrently in InnoDB. MySQL manual suggested its value to be the sum of the number of processors and hard disk drives in the system. In our case this number is 5. In order to minimize the effect of initial cache heat-up of MySQL and file system cache in our tests, we do a complete database table scan before each of our tests where applicable.

PDNS server also allows the configuration of the number of PDNS query handling threads. The value is five in most of our tests, which is large enough for the server to take advantage of all available processor capacity. We tuned this parameter to other values only in the study involving increased processor capacity, as will be noted in Section 4.4.1.

4.2 Overall Query Performance

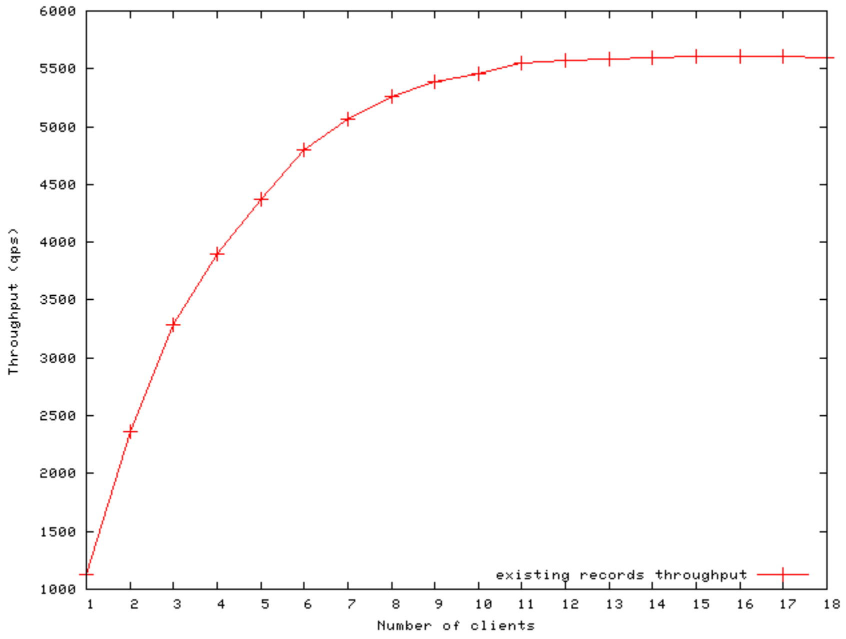
Our basic test is performed on the 5M-record set with PDNS and MySQL server collocated in the same machine. The performance is shown in Figure 3 and Figure 4.

Figure 3(a) and Figure 4(a) show PDNS query throughput and response time when querying records that exist in the database. That is, the query space is within the 5M-record set, from `0.0.0.0.0.0.0.2.1.2.1.e164.arpa` to `9.9.9.9.9.9.4. 2.1.2.1.e164.arpa`. As we can see, when the number of clients is below 3, the throughput increases quickly, from 1100 qps to 3300 qps, while the response time remains below 0.8 ms. After that the response time increases faster, and throughput increase rate slows down. The throughput saturates at around 5,500 qps at client number 12, with the response time taking a dramatic increase beyond that point.

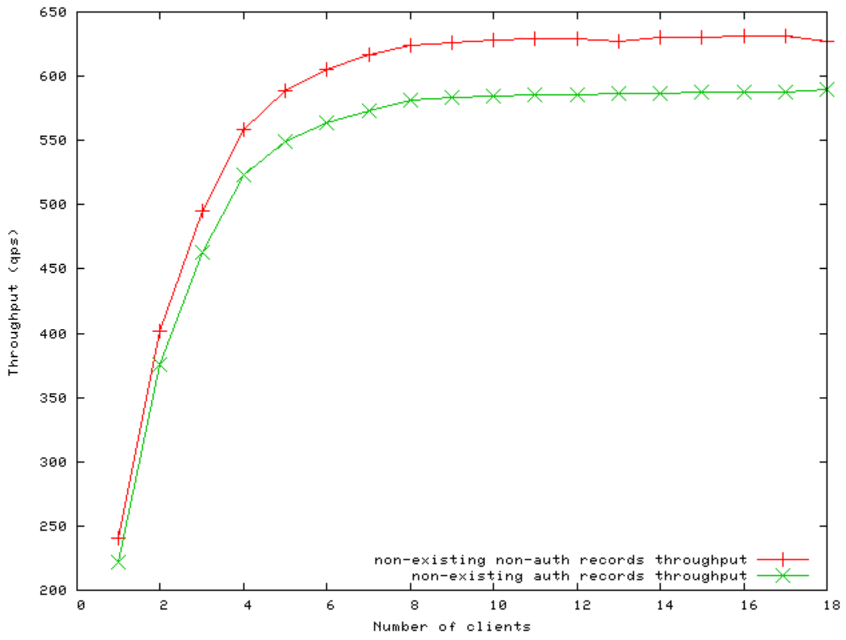
When querying domains that do not have matching records in the server database, there are two possibilities. First, the server does not have exact record match for the queried domain but contains the SOA records for a subdomain of the queried domain. Such queries are for non-existing authoritative records. Second, the server contains neither an exact record match, nor any subdomain SOA record match for the queried domain. Such queries are for non-existing non-authoritative records.

The throughput and response time for querying non-existing records are shown in Figure 3(b) and Figure 4(b). The query space for 5M non-existing authoritative records is from `0.0.0.0.0.0.0.5.2.1.2.1.e164.arpa` to `9.9.9.9.9.9.9.2.1.2.1.e164. arpa`. The query space used as 5M non-existing non-authoritative records is from `0.0.0.0.0.0.0.5.3.1. 2.1.e164.arpa` to `9.9.9.9.9.9.9.3.1.2.1.e164.arpa`. There are several main points we can see. First, querying non-existing non-authoritative records achieves a higher throughput and lower response time, as opposed to querying non-existing authoritative records. Second, the overall performance of querying non-existing records is significantly worse than that of querying existing records. At the saturation point, the throughput of the former is only 1/9 of the latter, the response time is 7.5 times longer. Third, the response time using one client is longer than that using two clients. We will analyze the causes and further quantify these facts in the next section when we examine components that make up the overall response time.

The query pattern in reality is more likely to be mixed between queries for both existing and non-existing records. We found that the response time to mixed queries is additive. From Figure 3 and Figure 4 we can obtain the individual cost of querying a specific types of records, namely, existing record, non-existing authoritative records, and non-existing non-authoritative records. Denote these values as t_1, t_2, and t_3. If we know the proportion of each type of queries as p_1, p_2 and p_3, we can obtain the estimated response time for the mixed query space by $t_1 * p_1 + t_2 * p2 + t_3 * p_3$. We tested two mixed query space cases. The first one contains queries from `0.0.0.0.0.0.0.0.2.1.2.1.e164.arpa` to `9.9.9.9.9.9.9.3.1.2.1.e164.arpa`. This 20M query range is composed of 5M existing records, 5M non-existing authoritative records and 10M non-existing non-authoritative records. The second one contains queries within the whole range from `0.0.0.0.0.0.0.0.0.0.0.0.e164.arpa` to `9.9.9.9.9.9.9.9.9.9.9.9.e164.arpa`. Figure 5 shows the measured throughput and response time of these two cases, as well as the estimated

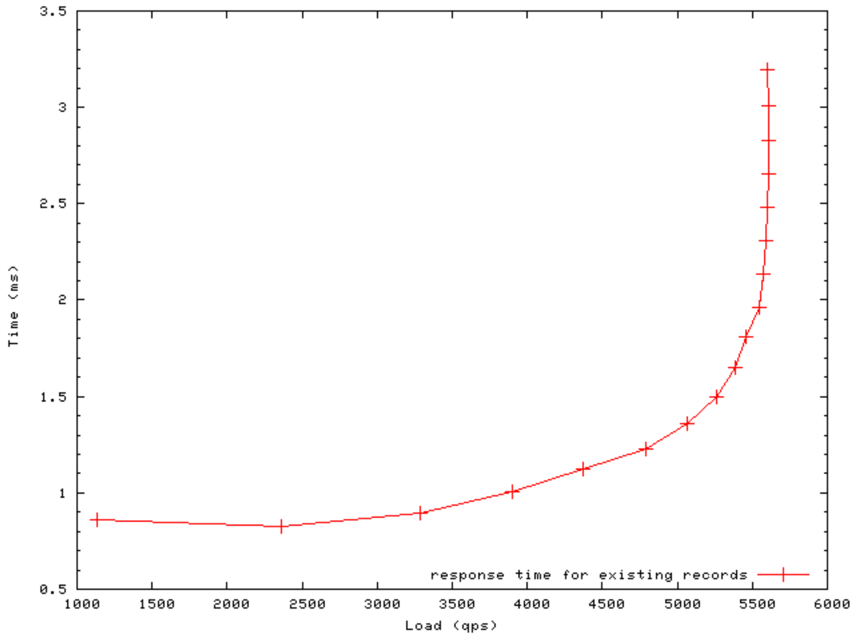


(a) Querying existing records

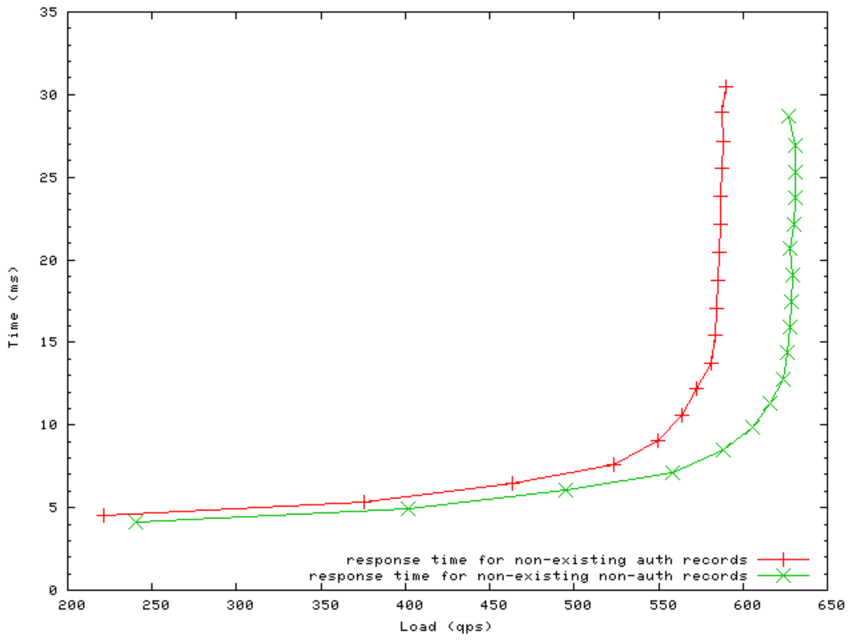


(b) Querying non-existing records

Figure 3: PDNS throughput for 5M-record set



(a) Querying existing records



(b) Querying non-existing records

Figure 4: PDNS response time for 5M-record set

response time. In both cases, the measured response time value and the estimated response time value match quite well. The maximum throughput in the first mixed query case, which has a 25% record hit rate, is only 14% the maximum throughput of the case with a 100% record hit rate shown previously in Figure 3(a). This indicates that throughput increase will grow faster when record hit rate is higher. In the second mixed query case the performance is essentially similar to the previous test for 5M non-existing records, because the number of non-existing records in the test far exceeds the number of existing records.

4.3 Analysis of Response Time Components

4.3.1 PDNS Query Processing Details

To analyze the details of the PDNS query response time, it helps to first have a look at the PDNS internal architecture. Figure 6 shows the major thread components of the PDNS server.

The PDNS main thread spawns a query listening thread to take in queries, a server control listening thread to accept server control commands (e.g., cache purge), and a number of query handling threads (the distributor) that interact with the backend database. There are two types of cache shown in Figure 6. The packet cache caches the entire query packet and its reply, while query cache caches the database query results and is a lower-level caching mechanism compared to the packet cache. The entries in the packet cache are indexed by the combination of the queried domain name, the queried resource record type, and a bit indicating whether DNS recursion is desired; the entries in the query cache are indexed by the queried domain name, the queried resource record type, and the DNS zone identifier. The packet cache and query cache share the same storage space. During a normal query, the query listening thread receives the query and checks the packet cache first. If a packet cache hit occurs, a response is sent back directly without submitting the query further to the distributor. Otherwise, the query will be handed to one of the query handling threads in the distributor. The query handling thread formulates a database query command for the backend database. Before this database query command actually makes its way to the database, the query cache is checked. If a cache hit occurs, the result will be retrieved from the query cache directly. If not, the query command goes to the backend database. Note that caching can be applied to both positive queries and negative queries.

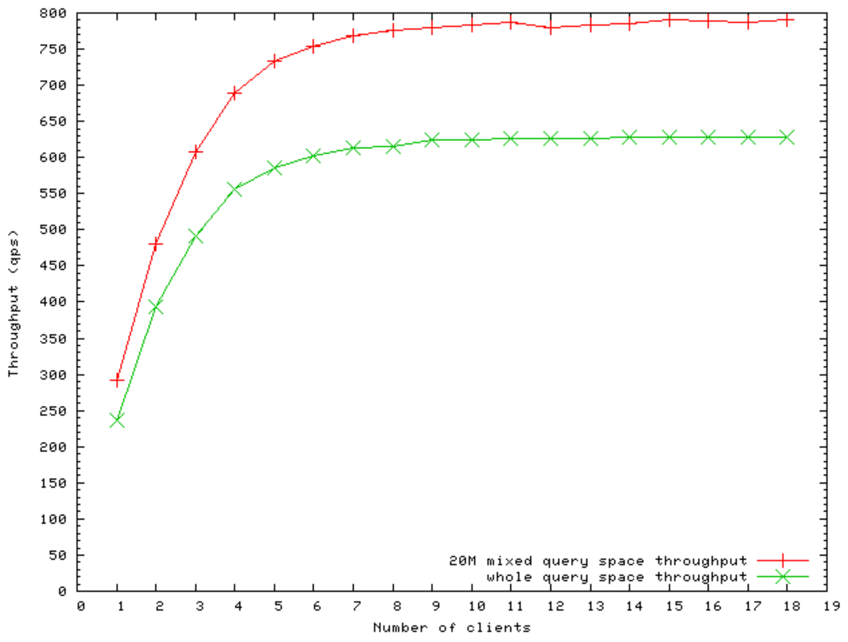
The PDNS query processing flowchart is shown in Figure 7. We can see that the only operation involving the packet cache is the first step after the query is just received. There are multiple lookup operations that involve the backend database, including lookup of the CNAME record, the actual type of record requested, domain and subdomain SOA records, NS for SOA record, and "any" type record. All these operations will pass the query cache check before the backend database is involved.

As seen from Figure 7, different query processing procedures apply to existing records, non-existing authoritative records, and non-existing, non-authoritative records. In the following we use a specific example to illustrate the procedures. We consider a first time ENUM lookup of a NAPTR record for the queried domain `0.0.9.0.2.0.9.7.1.3.1.e164.arpa`. This is one of the non-existing, authoritative domains for our 5M-record set.

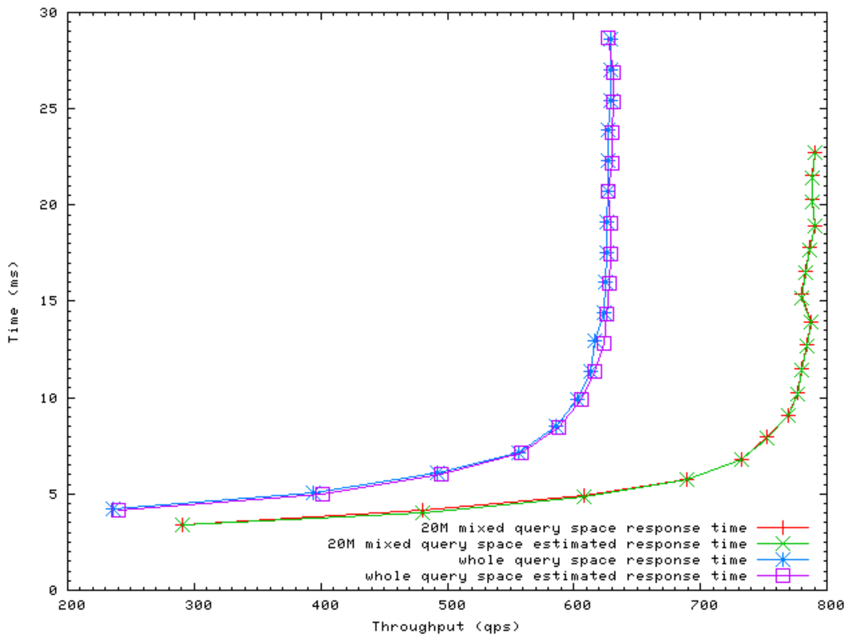
When the query is first received, it is checked against the packet cache. Since this is our first time to query this domain, no packet cache match is found. The query is submitted to the question queue and picked up by one of the query handling threads. There, it first goes through the lookup for canonical records (make canonic). This operation involves submitting an "any" query type for the queried domain. This query will first pass the query cache check, like any other queries submitted to the backend database. Since this is the first time lookup, no query cache match is found. The following statement is then passed to the MySQL database:

```
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE name='0.0.9.0.2.0.9.7.1.3.1.e164.arpa'
```

Once results are returned, the server checks to see if the results contain a CNAME record for the queried domain. In addition, the server checks to see if the returned results contain



(a) Throughput with mixed query space



(b) Response time with mixed query space

Figure 5: PDNS performance for 5M-record set with mixed query space

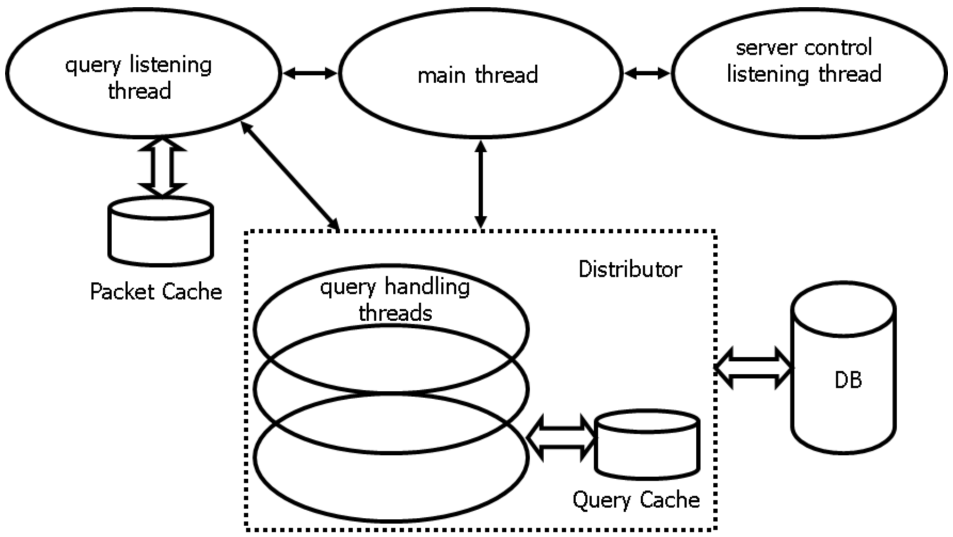


Figure 6: PDNS internal thread view

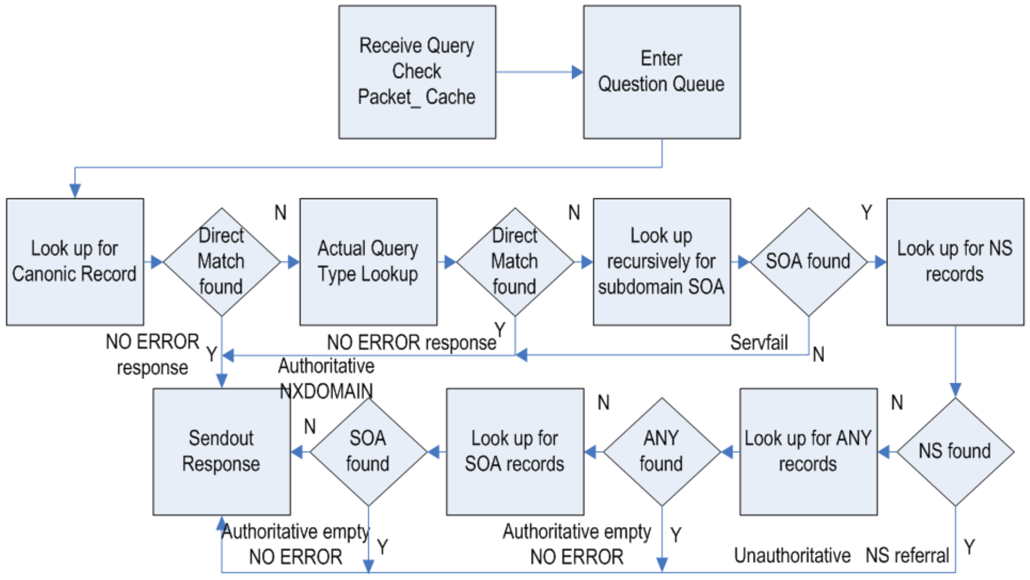


Figure 7: PDNS query processing flowchart

a record of the same type as what had been requested, which is NAPTR in our case. If yes, this indicates we have already got a direct match for our query. So remaining lookups will be bypassed and the program proceeds to send out the response. This is where a query for an existing record will end. In our example, no result is found from the "any" type query for the domain `0.0.9.0.2.0.9.7.1.3.1. e164.arpa`.

The server then proceeds to the next "actual type" lookup phase. This operation is similar to the previous one, which is another query of "any" type for the specified domain. As in the previous "make canonic" lookup, no result is returned for the specified queried domain. The difference is that if negative query cache is enabled, this "actual type" lookup phase could be faster than the previous "make canonic" lookup because of the cache entry created in query cache during the previous lookup.

Next the server will check subdomain authority for the queried domain recursively. For the specific domain `0.0.9.0.2.0.9.7.1.3.1.e164.arpa` in our example, the following queries will be made:

```
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='0.0.9.0.2.0.9.7.1.3.1.e164.arpa'
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='0.9.0.2.0.9.7.1.3.1.e164.arpa'
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='9.0.2.0.9.7.1.3.1.e164.arpa'
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='0.2.0.9.7.1.3.1.e164.arpa'
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='2.0.9.7.1.3.1.e164.arpa'
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='0.9.7.1.3.1.e164.arpa'
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='9.7.1.3.1.e164.arpa'
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='7.1.3.1.e164.arpa'
```

The check could continue to be made for `1.3.1.e164.arpa`, `3.1.e164.arpa`, `1.e164.arpa`, `e164.arpa`, `arpa`, ".", if the server is not authoritative for any subdomain already checked. So the total number of subdomain SOA queries can be up to 14. In our example, the server is indeed authoritative for the `7.1.3.1.e164.arpa` domain. So the subdomain SOA check stops when this subdomain name is reached.

The next step is to check for appropriate NS records. In our example the following lookups will be made:

```
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='NS' and name='0.9.0.2.0.9.7.1.3.1.e164.arpa' and domain_id=9
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='NS' and name='9.0.2.0.9.7.1.3.1.e164.arpa' and domain_id=9
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='NS' and name='0.2.0.9.7.1.3.1.e164.arpa' and domain_id=9
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='NS' and name='2.0.9.7.1.3.1.e164.arpa' and domain_id=9
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='NS' and name='0.9.7.1.3.1.e164.arpa' and domain_id=9
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='NS' and name='9.7.1.3.1.e164.arpa' and domain_id=9
```

The *domain_id* value is obtained during the previous successful lookup of the SOA record for subdomain `7.1.3.1.e164.arpa`. In our case, none of the above NS records exist in the server, the program proceeds to perform another "any" type query:

```
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE name='0.0.9.0.2.0.9.7.1.3.1.e164.arpa'
```

No results are returned from this operation and the program makes yet another attempt for the SOA record of the queried domain:

```
SELECT content,ttl,prio,type,domain_id,name FROM records
 WHERE type='SOA' and name='0.0.9.0.2.0.9.7.1.3.1.e164.arpa'
```

After this operation which again returns no results, the server constructs the and sends final response indicating that the server is authoritative for the queried domain but no such domain exists.

The above example shows almost the worst case lookup procedure. The server could jump to sending out a response in multiple locations in the middle of these procedures as shown in Figure 7.

4.3.2 Response Time Components

Based on Section 4.3.1, we summarize the PDNS server processing delay as containing the following main components:

1. time waiting in question queue;
2. "make canonic" database lookup;
3. "actual type" database lookup;
4. SOA-related database lookup (including all subsequent NS and more SOA lookups);
5. other server processing time (including all server processing other than those listed above).

Note that in the implementation, both the "make canonic" and "actual type" lookup operation submit an "any" type query to the database. So the database will return any matching records, regardless of the type. For example, if we query for an domain `4.3.2.1.9.3.9.2.1.2.1.e164.arpa.` and it exists in the database, the "make canonic" database lookup will return the actual NAPTR record since our server does not contain any CNAME records. This completes the database lookup. So querying existing records involves only three components, namely question queue waiting, "make canonic" database lookup, and other server processing time. On the other hand, if the queried domain does not exist in the database, neither the "make canonic" nor "actual type" database lookup will return any results. Thus the SOA-related database lookups will be invoked. Querying non-existing records will therefore involve all five delay components mentioned above.

Figure 8 illustrates the three server processing time components along with the total response time we previously showed in Figure 4(a) for querying existing records in the 5M-record set. The database lookup time item refers to the second latency component "make canonic" database lookup, which is the only database lookup operation for querying existing records. As we can see, before the server is saturated, the database lookup time is the dominant factor. It always accounts for over 60% of the total server processing time. After reaching the saturation point, its value keeps at around 0.85 ms. This is because the load PDNS can deliver to the MySQL database is relatively constant once the saturation point is reached. Another component, the question queue waiting time increases slowly while the server is in normal operation range. The increasing rate grows rapidly as the server approaches the saturation point, until it replaces the database lookup time as the main factor of the overall server processing time. Another delay component "other server processing time" remains relatively constant and small all the time, with an overall increasing trend.

There is also a small difference between the total response time and the overall server processing time. This difference can be explained by considering the definition of these two terms. The total response time is what is seen by the client between sending a query and finally receiving a response. The server processing time is from when the query is actually received by the server application until the system routine to send out the response is called. So the differences between the two are mainly the client and server side processing and CPU scheduling delays. Among these delays we found that the main component is the receiving buffer waiting time

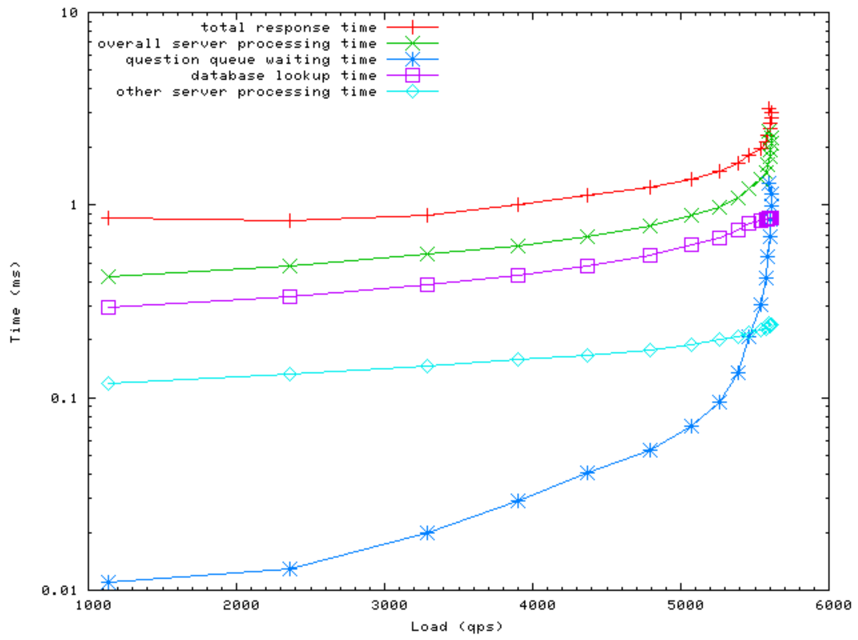


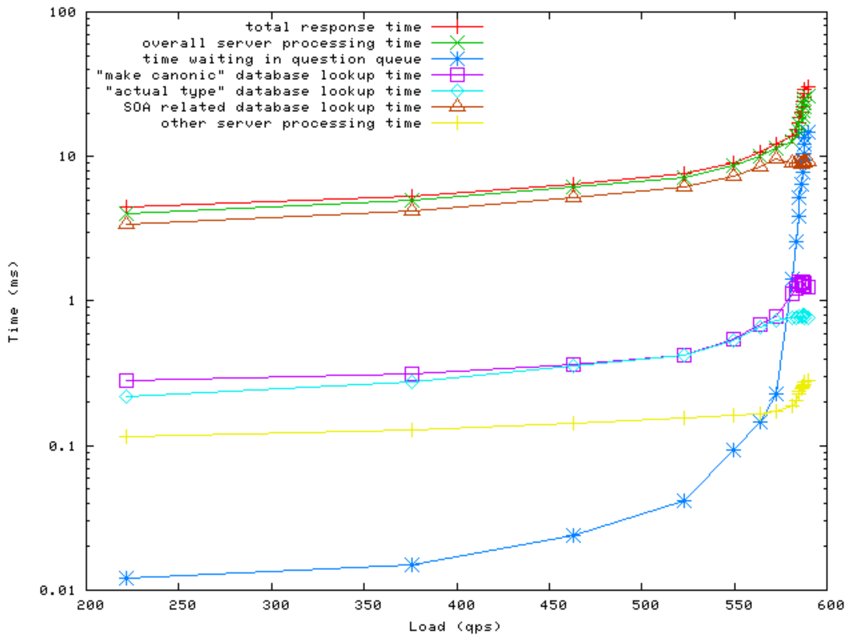
Figure 8: PDNS Response time components for querying 5M existing records

at the server, i.e., the packets are sitting in the receiving buffer waiting to be picked up by the server application. In certain cases, this value could be hundreds of milliseconds. In the following part of the document we will denote this difference of total response time and overall server processing time as receiving buffer waiting time. In Figure 8, the receiving buffer waiting time is between 0.3 ms to 0.6 ms before the server is saturated, and it does not increase much after that. We can also see that although the server processing time is clearly shorter in the one client run than that in the two clients run, the receiving buffer waiting time shows the opposite. This explains why the total response time for one client is slightly larger than that of two clients in Figure 4(a).

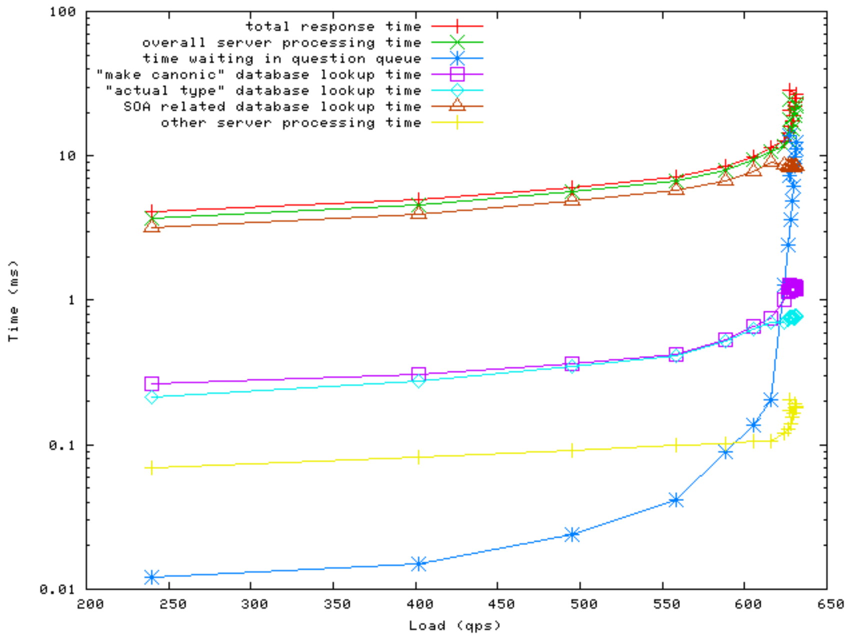
Figure 9 shows response time components for querying non-existing authoritative and non-existing non-authoritative records, corresponding to Figure 4(b). Comparing Figure 9 with Figure 8 we see that before the server reaches its maximum capacity, the first database lookup time common in querying existing and non-existing records is similar.

The reason that makes querying a non-existing record significantly longer than querying an existing record is due to the additional "actual type" database lookup and the extra SOA related database lookups. The additional "actual type" database lookup time is similar to the "make canonic" database lookup time, as expected when no cache is enabled. The SOA-related lookups cost over 12 times of the "make canonic" database lookup latency. This is explained by the PDNS query processing details we presented in Section 4.3.1 where we mentioned that querying a non-existing non-authoritative record may result in up to 14 extra SOA-related database lookups.

Processing differences for non-existing authoritative records and non-existing non-authoritative records can also be seen. According to Figure 7, the number of extra database lookups could be even higher in the case of non-existing authoritative records because of additional NS records and other SOA record lookups after the first round of subdomain SOA lookup. Comparing Figure 9(a) and Figure 9(b), we see that the SOA-related lookup time component for non-existing authoritative records is longer than that for non-existing non-authoritative records.



(a) Querying 5M non-existing authoritative records



(b) Querying 5M non-existing non-authoritative records

Figure 9: PDNS response time components for querying 5M non-existing records

4.4 Performance Limitations

4.4.1 Server Processor Capability

In this section, we study how server processor capability may limit the performance of PDNS. The results in Section 4.2 are based on a collocated PDNS and MySQL architecture, i.e., both PDNS and MySQL are located in the same machine. The CPU utilization of PDNS and MySQL for the tests in Section 4.2 is shown below in Figure 10.

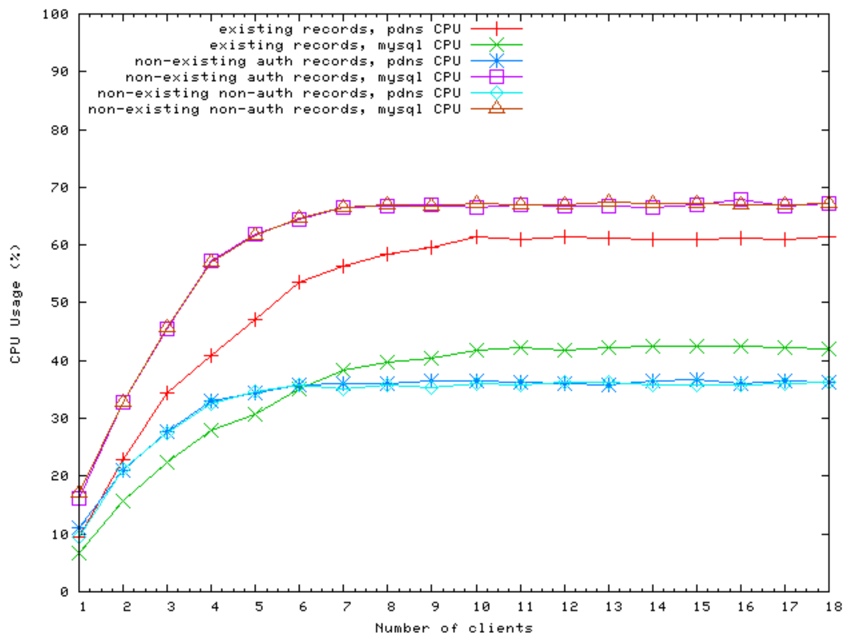


Figure 10: PDNS and MySQL CPU utilization for 5M-record set

We can see that in all tests, the performance is bounded by the CPU processing power. At the saturation point, the sum of PDNS and MySQL CPU usage approaches 100%. In the case of querying existing records, PDNS is more processing intensive and consumes 1.5 times the amount of CPU used by MySQL. In the case of querying non-existing records, the ratio is reversed, with MySQL consuming about twice the amount of CPU used by PDNS. This is because in the non-existing record case, the number of database lookups far exceeds the number of submitted queries.

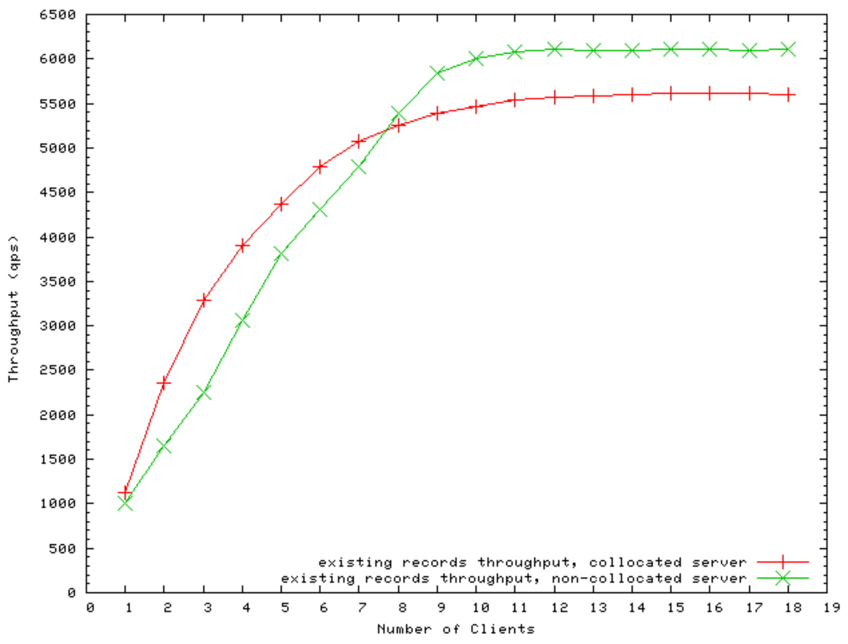
To measure performance with more server CPU processing power, we put the PDNS server and MySQL server in two separate machines with the same configuration, referred to as non-collocated architecture. Each machine has four 3 GHz CPUs and 8 GB memory.

Figure 11 shows the comparison of the resulting throughput and response time in the collocated and non-collocated PDNS architecture for querying existing records.

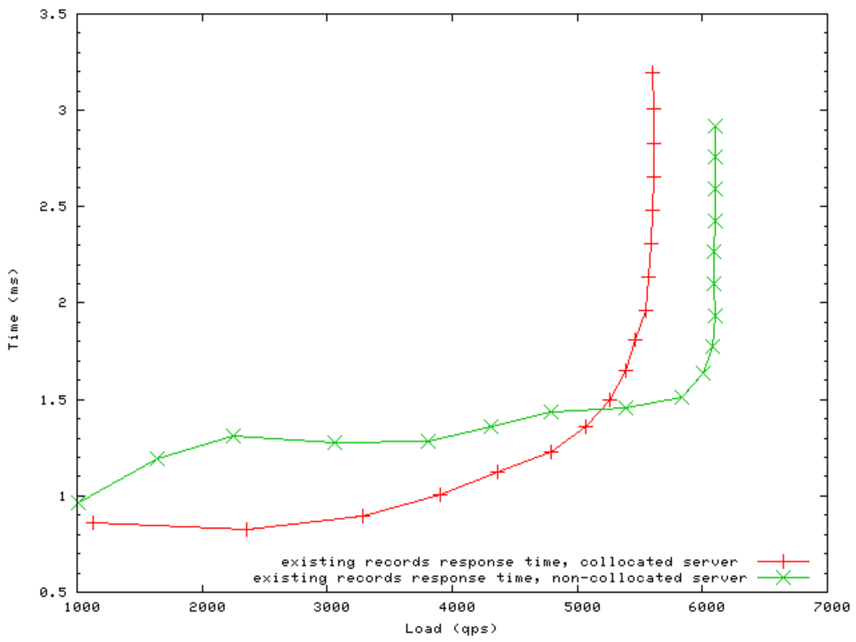
When querying existing records, the response time is a few hundred microseconds longer in the non-collocated PDNS architecture during most of the load range. Comparing the response time components in non-collocated architecture in Figure 13 with that of the collocated architecture in Figure 8, we see that although the receiving buffer waiting time is slightly smaller, the apparently longer database lookup time in the non-collocated case is the reason for the increase in total response time.

When querying non-existing records, Figure 12(b) shows that the response time in the non-collocated case could be about 7 ms longer in the normal load range. Similarly by comparing the response time components in Figure 14 with that in Figure 9, we can confirm that the cause is the extended database lookup time. As we mentioned before, querying non-existing records is far more MySQL intensive than querying existing records.

As far as throughput is concerned, the maximum throughput for querying existing records

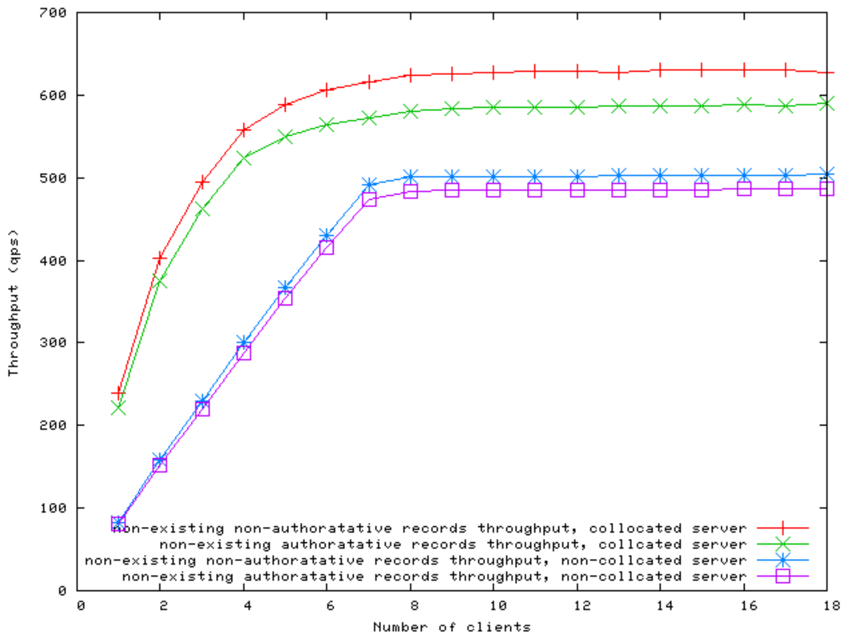


(a) Throughput

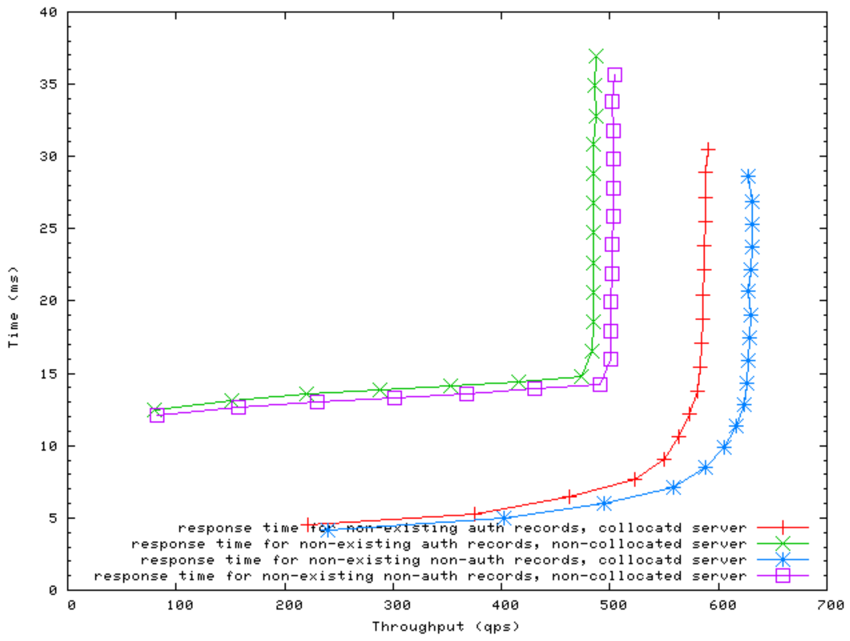


(b) Response time

Figure 11: Comparison of PDNS performance for querying existing records when PDNS and MySQL servers are collocated vs. non-collocated



(a) Throughput



(b) Response time

Figure 12: Comparison of PDNS performance for querying non-existing records when PDNS and MySQL are collocated vs. non-collocated

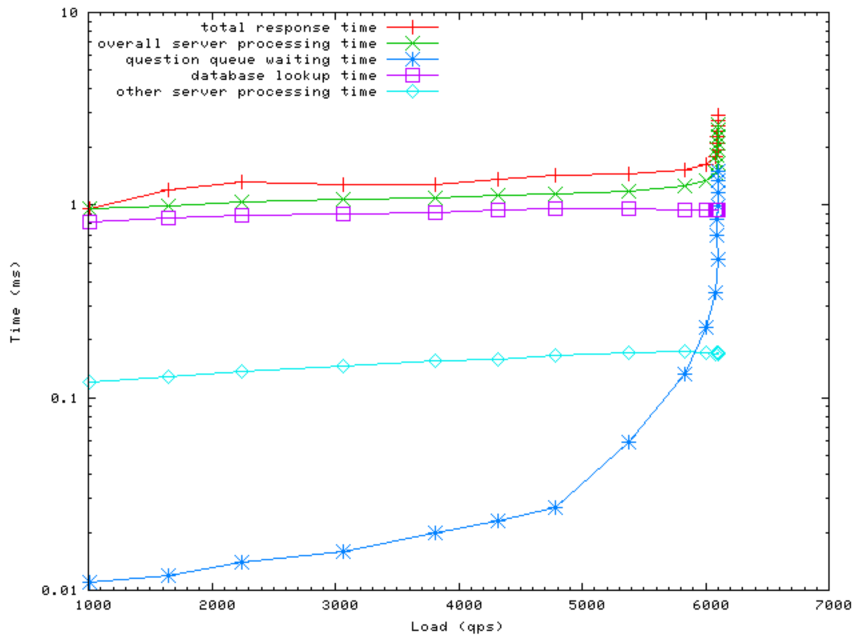
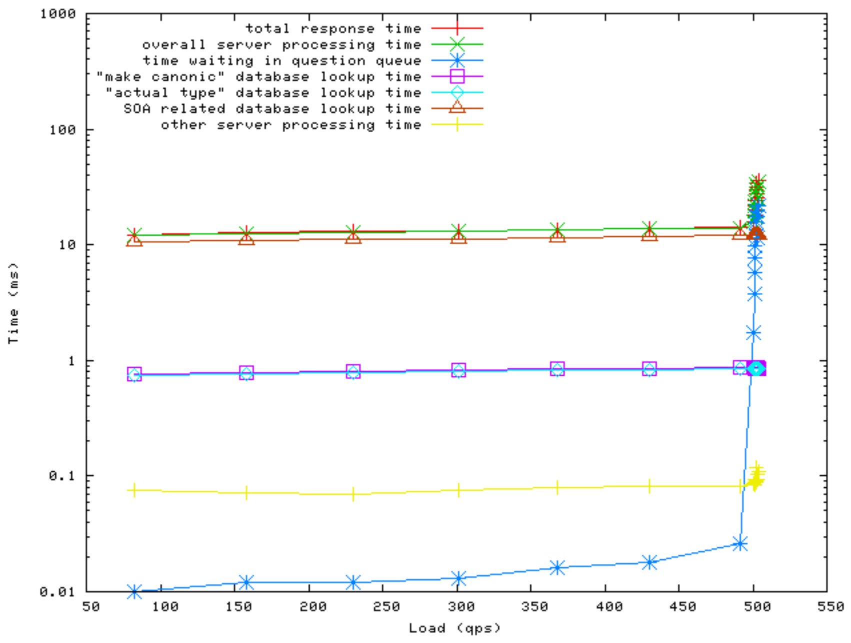


Figure 13: PDNS response time components for querying existing records when PDNS and MySQL are non-collocated

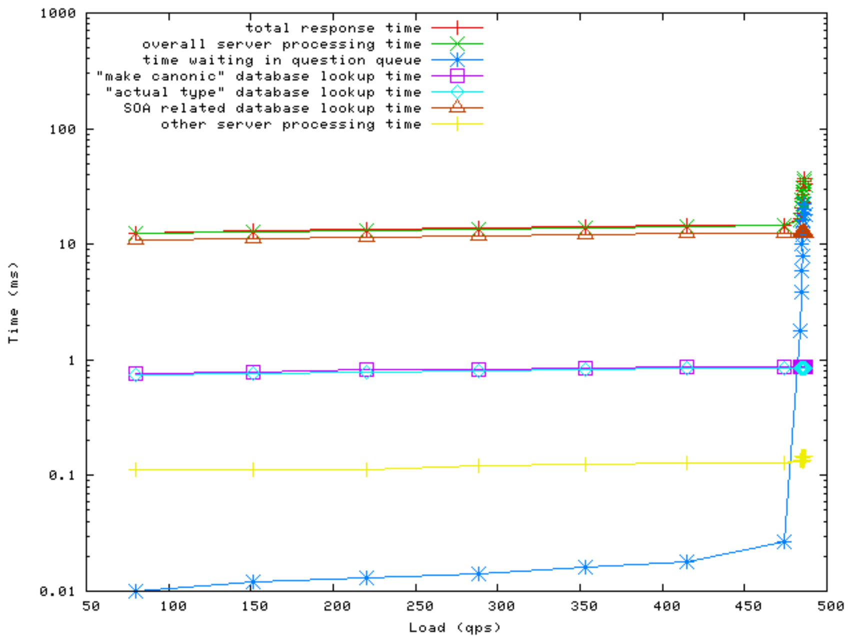
in the non-collocated case exceeds that of the collocated case by only about 500 qps, as seen in Figure 11(a). For querying non-existing records, Figure 12(a) shows that the maximum throughput in the non-collocated case is even smaller than in the collocated case, apparently caused by more MySQL lookup overhead. These results do not justify the use of non-collocated PDNS architecture with more CPU processing power. We therefore compared the CPU usage of the collocated and non-collocated cases in Figure 15. This figure shows that neither PDNS nor MySQL is able to take advantage of the increased processing power in the non-collocated case because of the concurrency overhead between them. Comparing this figure with Figure 10, we see that for existing records, the total amount of processing power used by both PDNS and MySQL is similar in the collocated and non-collocated case; for non-existing records, the total amount of CPU resrouces consumed is less in the non-collocated case than that in the collocated case. This is in line with the corresponding reduction in throughput in the non-collocated case. It is worth noting that despite the difference in total CPU usage, the ratio of MySQL CPU usage versus PDNS CPU usage is about the same in both collocated and non-collocated cases.

In order to take advantage of extra processing power in the non-collocated case, we repeated our tests with varying number of PDNS query handling threads. In addition to our default number of PDNS threads, which is five and equals to the number of MySQL threads, we measured the performance when the number of PDNS query handling threads T is set to 2, 8, and 16.

Figure 16 shows the throughput for querying existing records with varying number of PDNS query handling threads (shown as T in the figure). It can be seen that the maximum throughput increases notably with the increase of number of PDNS threads. At $t = 16$ the maximum throughput doubles the value of $t = 2$, or is 33% higher than the value of $T = 5$. The increase can be attributed to the fact that more queries can be handled concurrently at the server and therefore the server can make more use of the available resources. This can be confirmed by the CPU utilization of each server as shown in Figure 17. Note that the ratio 1.5 of PDNS vs. MySQL CPU usage for querying existing records is still valid in all cases tested. In the $T = 16$ case, Figure 17(c) shows that PDNS server is about to reach the full CPU processing power of the machine that hosts it. Therefore, the throughput at $T = 16$ approximates the true maximum throughput we can achieve with the non-collocated architecture. This value is about 45% more than the maximum throughput in the collocated architecture.



(a) Response time components for non-existing authoritative records



(b) Response time components for non-existing non-authoritative records

Figure 14: PDNS response time components for querying non-existing records when PDNS and MySQL are non-collocated

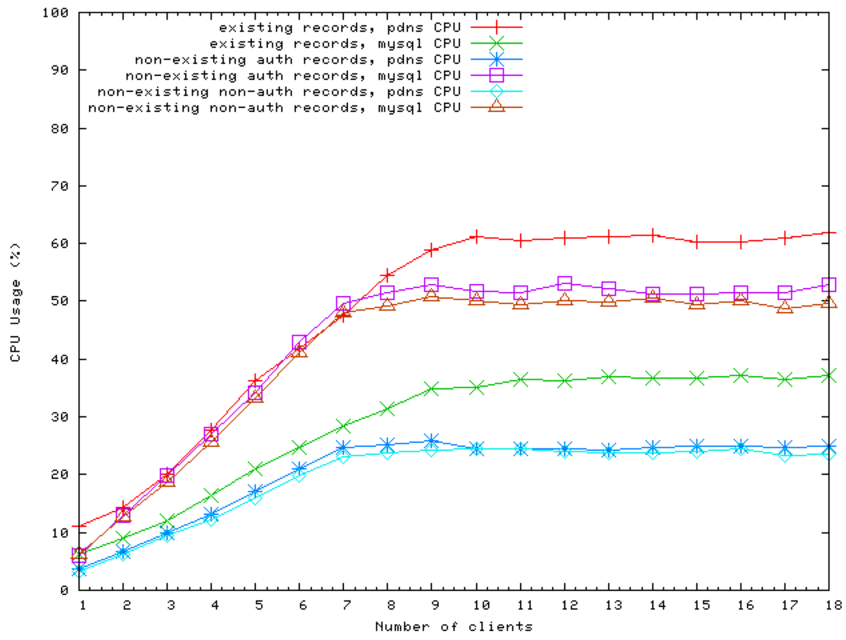


Figure 15: PDNS CPU and Memory utilization for querying non-existing records when PDNS and MySQL servers are non-collocated

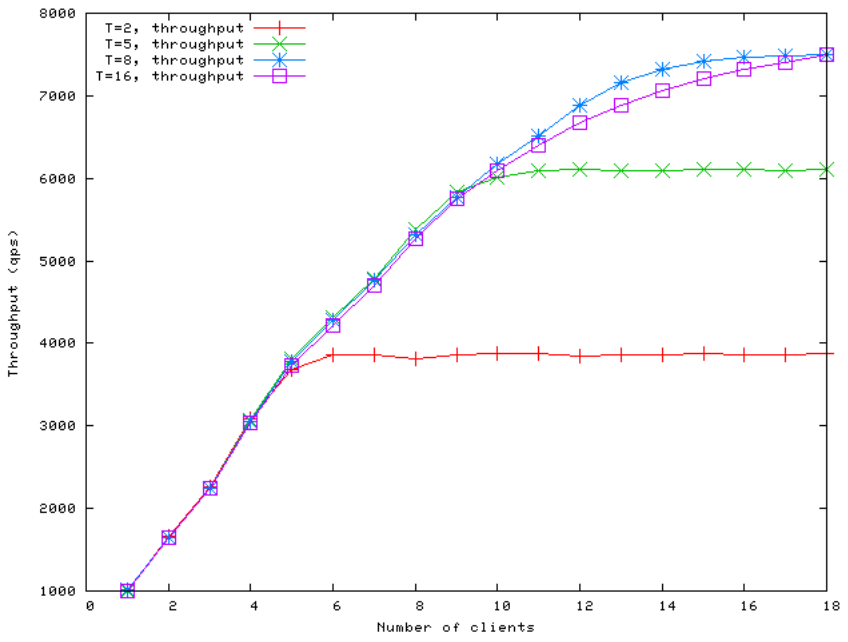
Figure 18 shows the total response time and server processing time with varying number of PDNS threads. It shows that response time remains relatively constant for the increasing throughput. This also confirms that the added throughput is due to better utilization of the processing power.

The performance for non-existing threads with varying number of PDNS threads is shown in Figure 19, Figure 20 and Figure 21. Here and also in the remaining part of this document, we will refer to non-existing records as non-existing non-authoritative records only, unless explicitly specified otherwise. (We have discussed the slight performance difference between non-existing authoritative and non-existing non-authoritative records and the reason for that difference. In reality the number of non-existing non-authoritative records is usually much larger than the number of non-existing authoritative records for any given record set.)

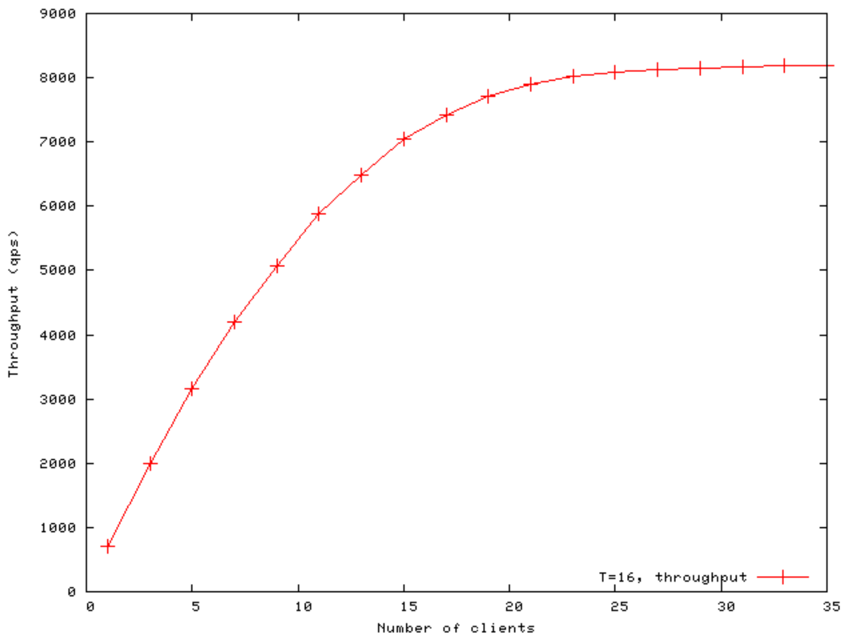
Similarly, the throughput increases notably with the increasing number of PDNS threads. The absolute throughput values are much smaller than those in the existing records case because of the processing overhead for non-existing records. The maximum throughput with $T = 16$ for querying non-existing records is three times that of $T = 2$ and is 80% more of the $T = 5$ case. The corresponding values for querying existing records are twice and 33%, respectively. So the relative gain of increasing number of PDNS threads for querying non-existing records is larger than that for querying existing records. On the other hand, the ratio 2 of PDNS CPU usage vs. MySQL CPU usage we obtained previously is also retained in all cases as seen from Figure 20. When $T = 16$, Figure 20(c) shows that MySQL consumed almost all 100% CPU of its host, while PDNS consumed a little less than 50% CPU of its host. So the $T = 16$ case approximates the true maximum performance under the non-collocated architecture. This value is about 40% more than that in the collocated case.

The total query response time for querying non-existing records shown in Figure 21 indicates that the latency remains similar in all cases as long as the server operates within its normal operation range. The same property held in the previous case of querying existing records.

In summary, CPU processing power plays an important role in limiting the PDNS performance. Comparing with a collocated PDNS server, using a proper PDNS thread number configuration and a non-collocated PDNS architecture can achieve a throughput improvement of 40% to 45% percent, with latency increasing by only a few milliseconds. It is important to

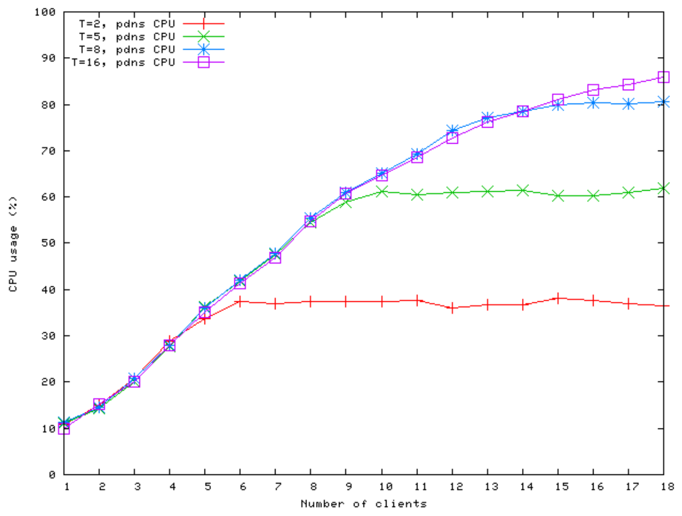


(a) Comparison of PDNS throughput when number of query handling threads is 2, 5, 8, 16

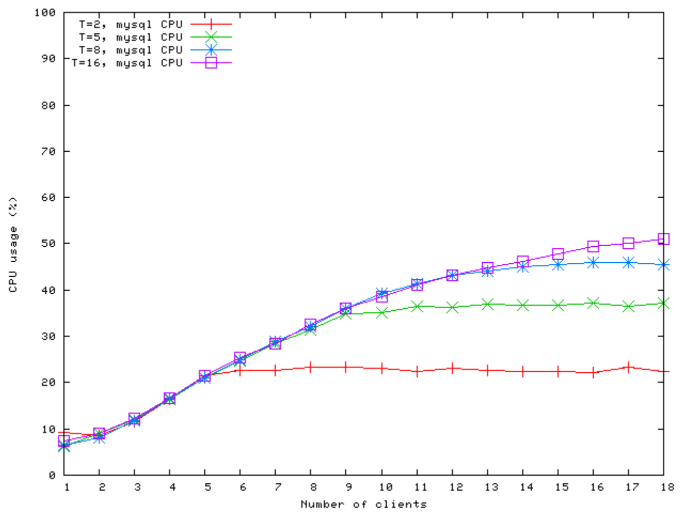


(b) Extended plot of PDNS throughput when number of query handling threads is 16

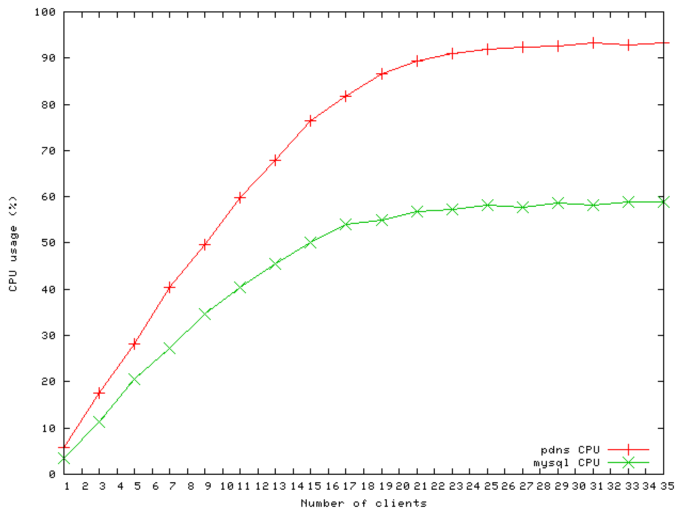
Figure 16: PDNS throughput for querying existing records as a function of a different number of query handling threads in non-collocated PDNS architecture



(a) Comparison of PDNS CPU utilization when number of query handling threads is 2, 5, 8, 16

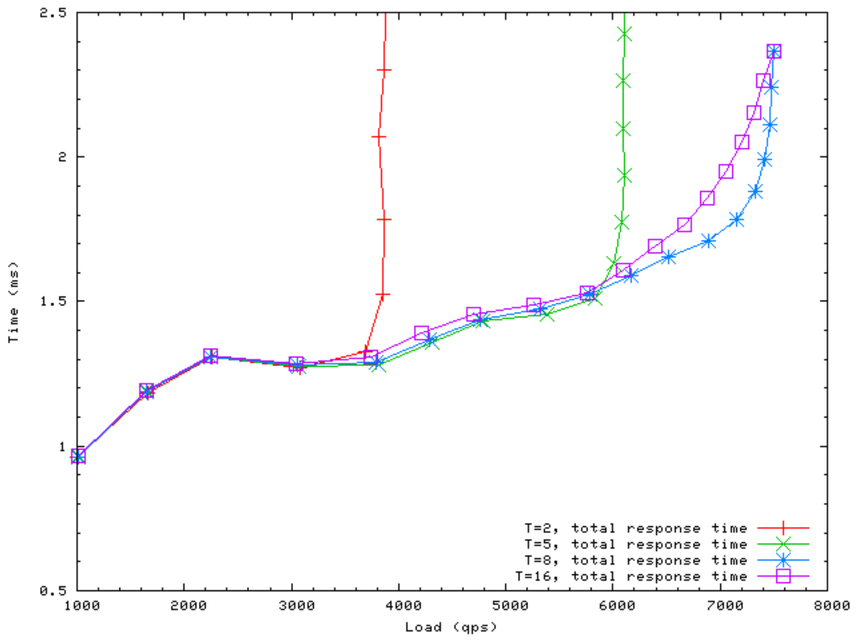


(b) Comparison of MySQL CPU utilization when number of query handling threads is 2, 5, 8, 16

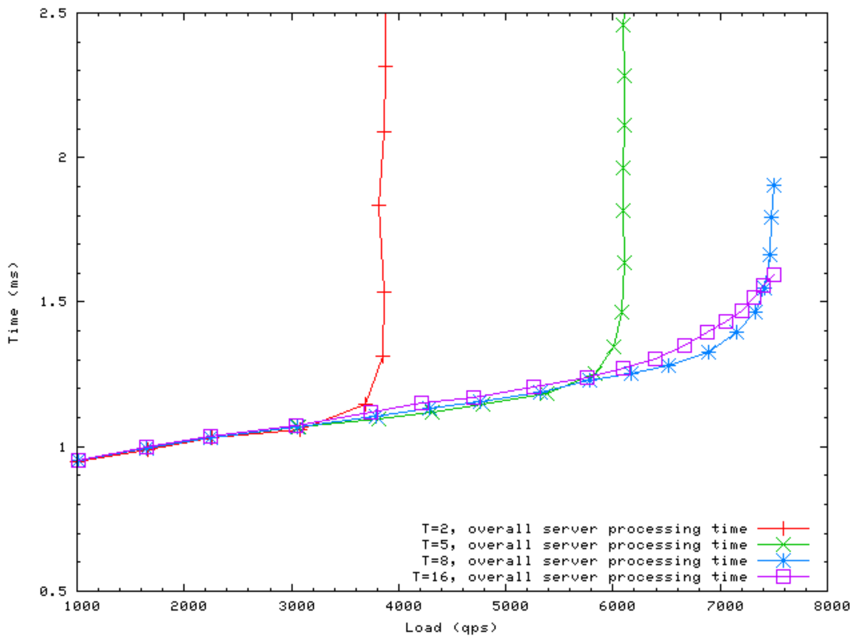


(c) Extended plot of PDNS and MySQL CPU utilization when number of query handling threads is 16

Figure 17: PDNS and MySQL CPU utilization for querying existing records as a function of a different number of query handling threads in non-collocated PDNS architecture

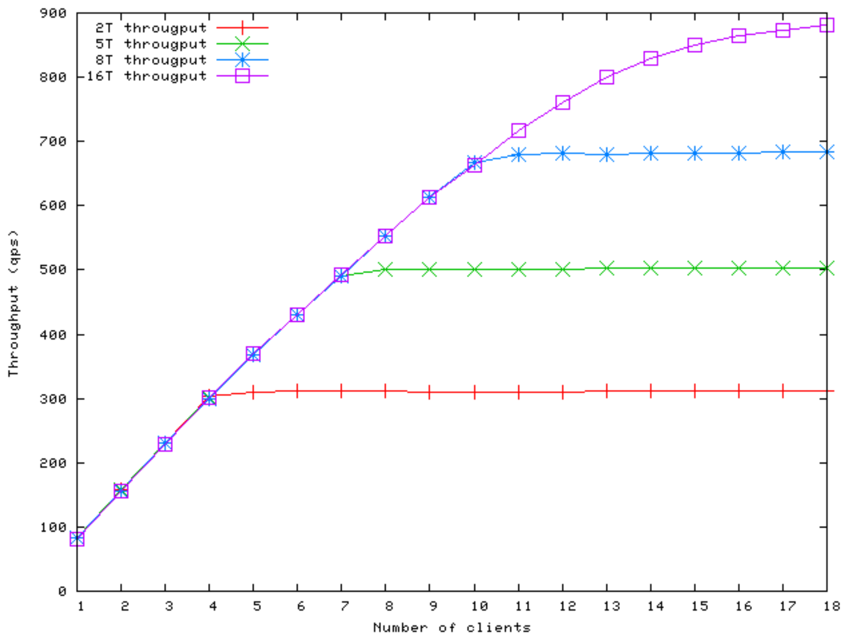


(a) Comparison of PDNS total response time when number of query handling threads is 2, 5, 8, 16

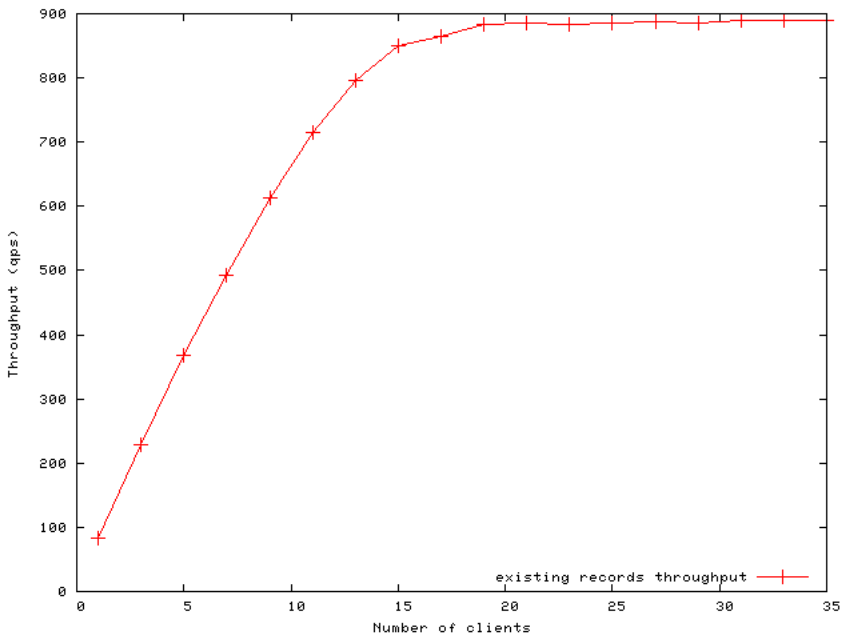


(b) Comparison of PDNS overall server processing time when number of query handling threads is 2, 5, 8, 16

Figure 18: PDNS response time for existing records as a function of different number of query handling threads in non-collocated PDNS architecture

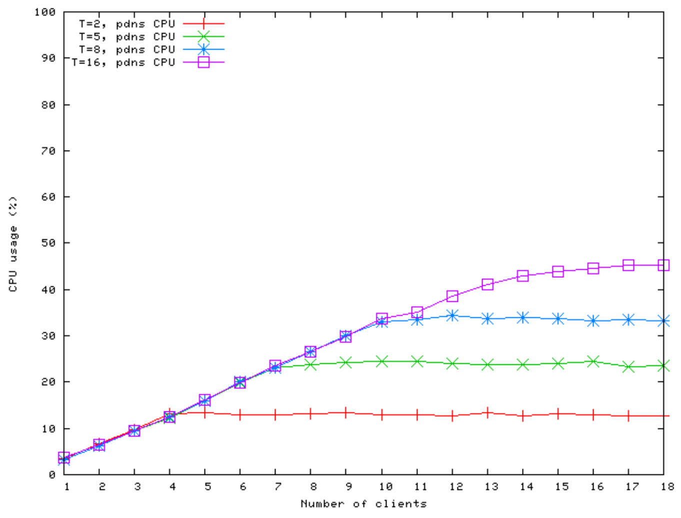


(a) Comparison of PDNS throughput when number of query handling threads is 2, 5, 8, 16

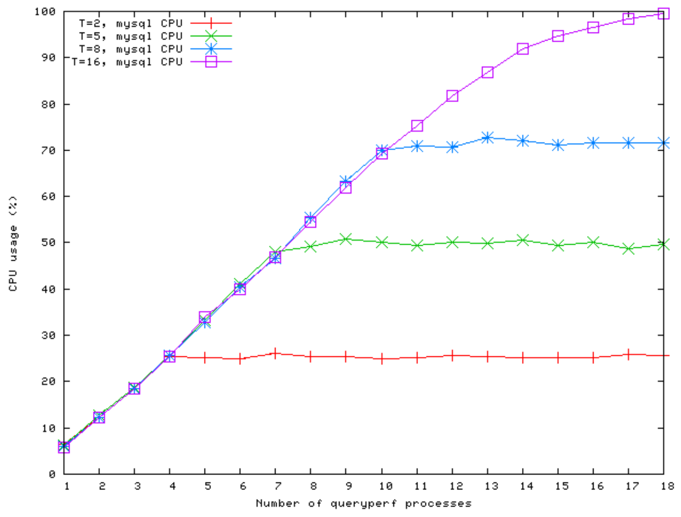


(b) Extended plot of PDNS throughput when number of query handling threads is 16

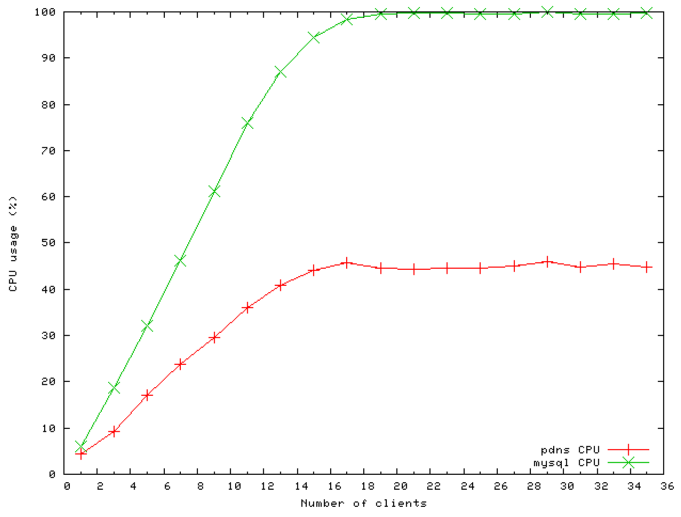
Figure 19: PDNS throughput for querying non-existing records as a function of a different number of query handling threads in non-collocated PDNS architecture



(a) Comparison of PDNS CPU utilization when number of query handling threads is 2, 5, 8, 16

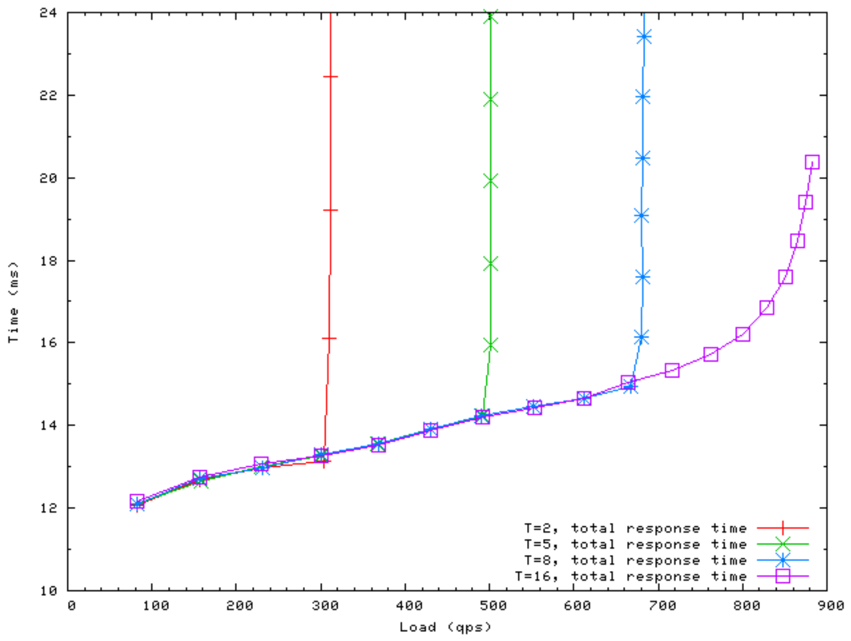


(b) Comparison of MySQL CPU utilization when number of query handling threads is 2, 5, 8, 16

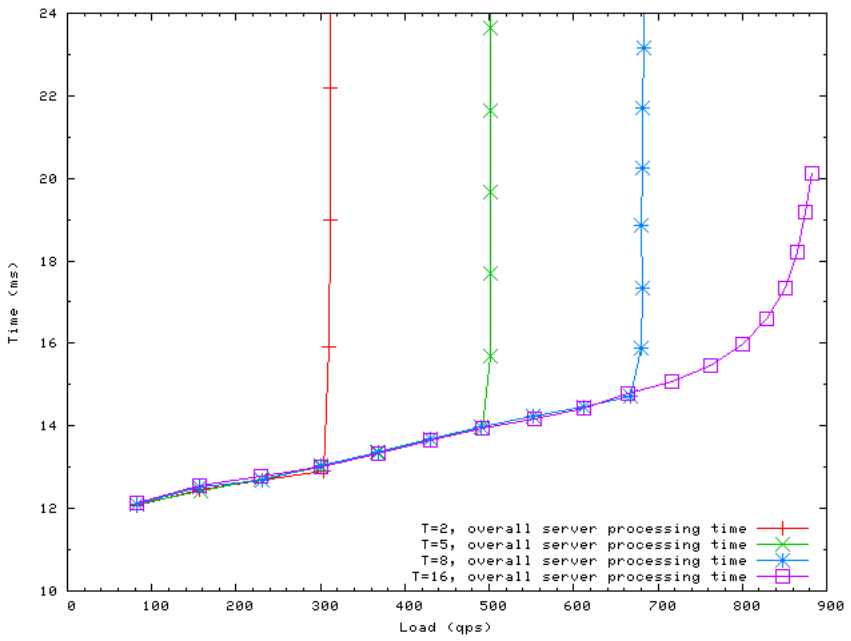


(c) Extended plot of PDNS and MySQL CPU utilization when number of query handling threads is 16

Figure 20: PDNS CPU utilization for querying non-existing records as a function of a different number of query handling threads in non-collocated PDNS architecture



(a) Comparison of PDNS total response time when number of query handling threads is 2, 5, 8, 16



(b) Comparison of PDNS overall server processing time when number of query handling threads is 2, 5, 8, 16

Figure 21: PDNS response time for querying non-existing records as a function of a different number of query handling threads in non-collocated PDNS architecture

note that the limiting factor is different for different query types. Querying for existing records is bounded by the CPU available to PDNS, which consumes about 1.5 times of the MySQL CPU; on the other hand, querying for non-existing records is bounded by the CPU available to MySQL, which takes about 2 times of PDNS CPU. Therefore, when deploying the non-collocated PDNS architecture, the expected query pattern should be taken into consideration in determining the optimal processing power for the systems that host PDNS and MySQL.

4.4.2 Database Scaling

In this section, we consider the scaling of the record set in the database. Our main record set used so far contains 5M records, which are all located in memory. We tested two additional record sets, the first one is a scaling down of 500k records, also in memory. The second one is a scaling up of 20M records. In this case the total size of MySQL data and index files is 7.4 GB which cannot fit into the 4 GB memory allocated to MySQL.

Figure 22 shows the PDNS throughput and response time for existing records with varying database sizes. When the records are in memory, as in the cases of 500k-record set and 5M-record set, the response time is almost the same and the maximum throughput differs only by 2%. The performance comparison for querying non-existing records with 500k and 5M record set is similar, as shown in Figure 23.

Figure 22 also shows, however, the performance degrades significantly when all the records cannot be fit into memory. The maximum throughput for the 20M-record set is reduced by over 80% compared with the other two, and response time increases by over 7 times.

To summarize, PDNS has a good scaling property as long as the database is in memory. There will be significant performance loss when the database is out of memory. Given that these days memory is relatively cheap, it may be worth it to use sufficiently large memory for higher PDNS performance.

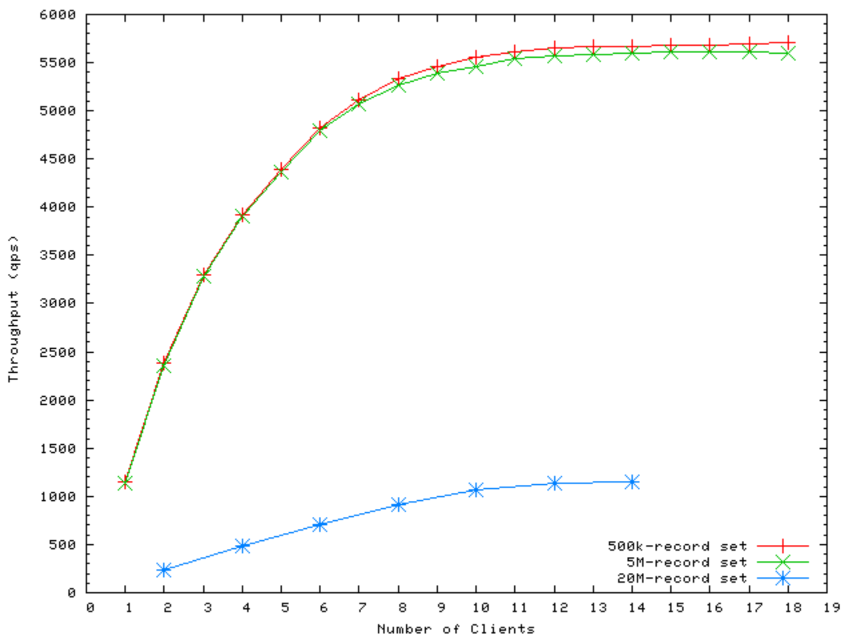
4.5 Improving Performance by PDNS Caching

The tests on PDNS in the previous sections assume the basic operation mode without any PDNS caching enabled. In this section, we look at the PDNS performance in caching mode. PDNS provides two levels of caching mechanisms, namely, packet cache and query cache. Packet cache is a higher level caching which caches the entire query and response packet directly. Query cache is sometimes also called record cache. It is a lower level cache that caches returned database lookup results. Both queries for existing records and non-existing records can be cached. The former is called positive query caching, and the latter is called negative query caching. A specific type of caching is enabled in PDNS by setting the corresponding TTL values to be positive. In our tests we set the TTL value to be longer than the experiment duration to obtain the performance of a full cache scenario.

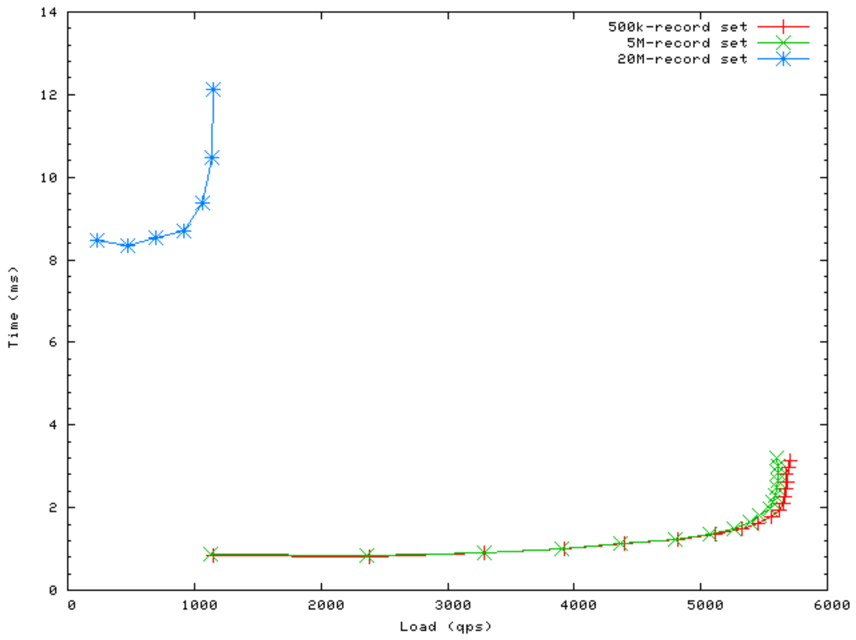
4.5.1 General Impact of Packet Cache and Query Cache

The PDNS caching tests are preceded by rounds of complete database scan to populate the PDNS cache. Due to the scanning traffic pattern and server thread contention, one round of complete scan does not guarantee that all records will be in the cache. We therefore repeated several rounds until the percentage of records in the cache is sufficiently high for our tests. In the packet-cache-only or query-cache-only test, 98% of the records are in the cache. In the test where both packet cache and query cache is enabled, 97% of the records are in the cache. Figure 24 shows the performance comparison of querying existing records with and without caching for the 5M-record set. Surprisingly, the results showed that the throughput in any of the PDNS cache enabled case is much worse than the original case without PDNS caching. The maximum throughput with both packet cache and query cache enabled is only half of the maximum throughput with only one type of PDNS cache enabled.

We repeated the same test for the smaller 500k-record set. The results are shown in Figure 25. The results in this figure seem more reasonable. The throughput with either type of cache enabled is higher than the case without caching. The packet cache case performs the best,

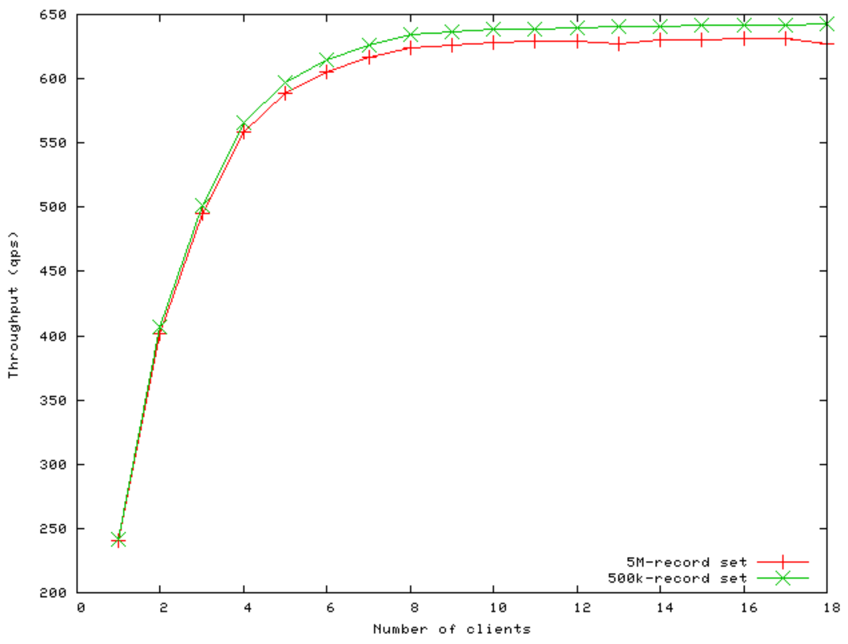


(a) Throughput

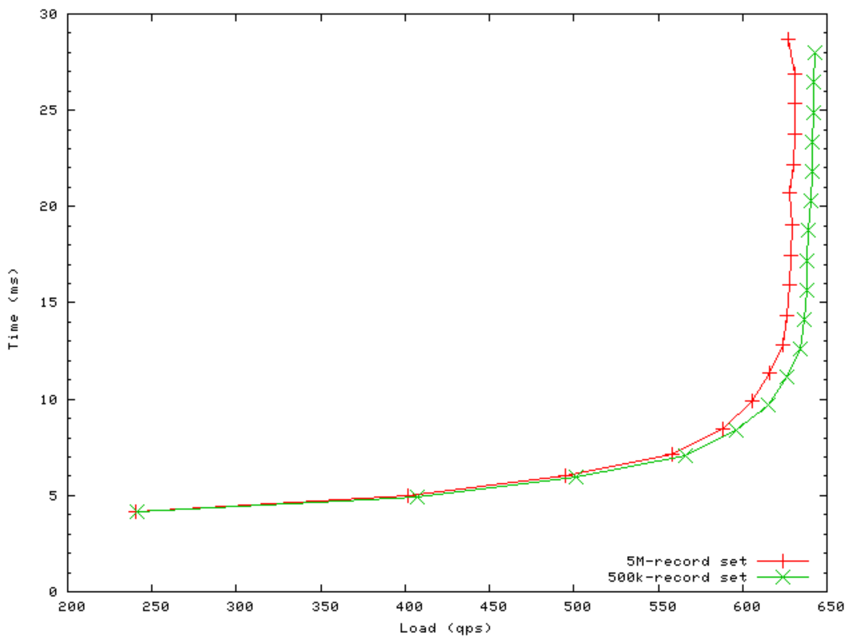


(b) Response time

Figure 22: Comparison of PDNS performance for querying existing records with different record set size



(a) Throughput



(b) Response time

Figure 23: Comparison of PDNS performance for querying non-existing records with different record set size

followed by the query cache case, which requires more processing. However, the fact that the performance with both caching enabled performed worse than with only one type of caching enabled is not intuitive. With both types of caching enabled, one would expect the packet hit the packet cache in most cases and occasionally hit the query cache if the packet cache happens to be busy. So the performance is expected to be at least no worse than the packet cache enabled case.

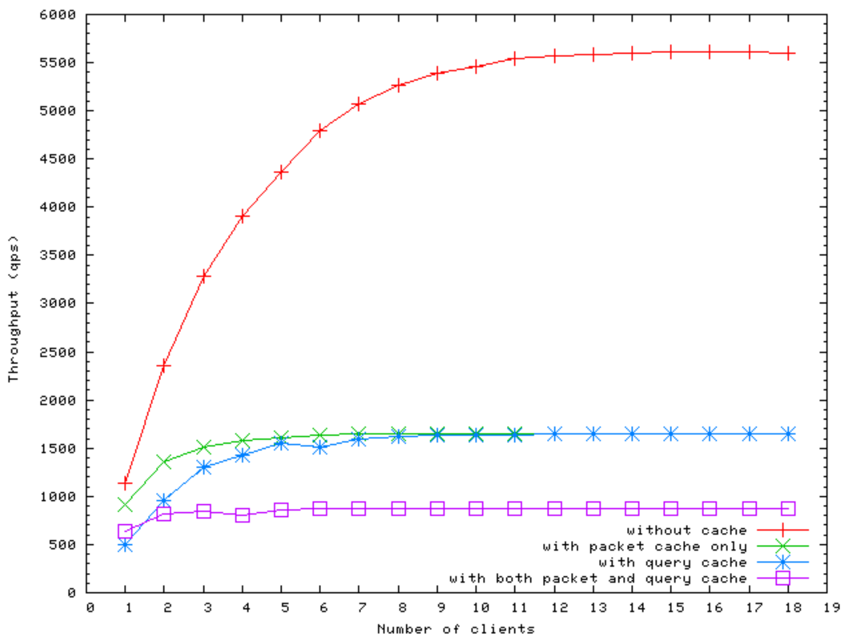
After examining the PDNS source code, we found that the abnormal results are linked to the cache size. Although packet cache and query cache have different keys and values, they share the same physical storage space. Therefore, the full cache size when both types of cache are enabled doubles the full cache size when only one type of cache is enabled. On the other hand, the full cache size in the test for 5M-record set is ten times that of the test for 500k-record set.

After profiling the server we further found that the problem is caused by a periodic cache maintenance procedure. In particular, for every 1,000 queries processed by the server, there will be an explicit cache cleanup operation which will go through the whole cache and remove those cache entries whose TTL value have expired. Since this is an *O(n)* operation, the cleanup time grows linearly when the cache size increases. This explains what we have seen above about the caching performance. At the level of 500k-record set, the impact of explicit cleanup operation is not severe enough to be noticed except when both packet cache and query cache are enabled. At the level of 5M-record set, the cost of this cleanup operation surpasses the server's normal operation in any of the three cache enabled cases, virtually eliminating the benefit of caching and making the performance worse than that without caching. This problem will also have significant effect on negative query caching containing SOA query results because there could be a very large number of negative SOA results when the query space is large. Given that nowadays many DNS server could set the TTL to fairly large values to reduce lookup latency, this cache cleanup problem should be fixed. One solution is to make such explicit cleanup run at the background only when the load is not a concern. Alternatively, an implementation may use a data structure that will place the most recently used entry at the head of the cache. When cache size exceeds the limit, entries can be removed from the bottom without the need to go through all cache entries.

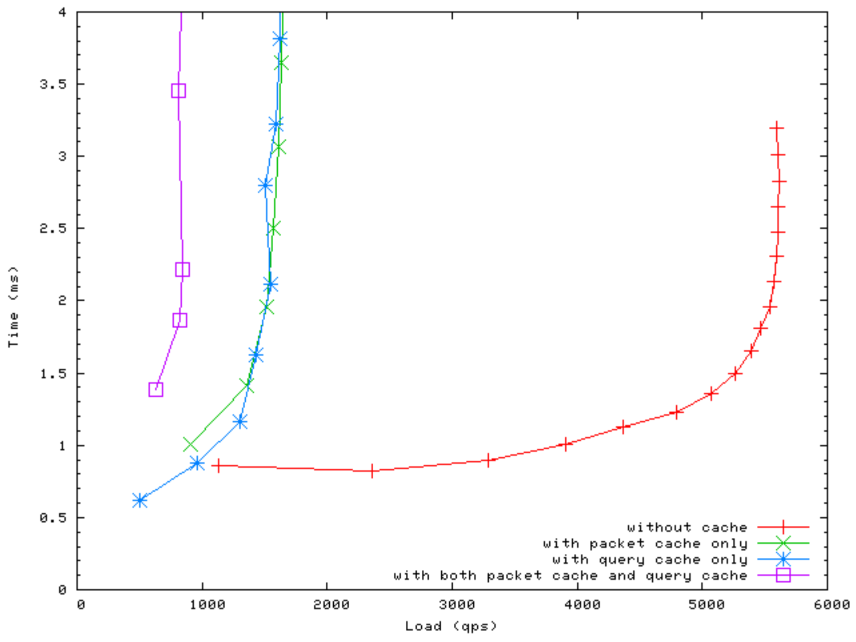
To see the difference in performance when this problem is fixed, we disabled the explicit cache cleanup (in our tests, the cache entries do not expire anyway) and repeated our tests again. The results are shown in Figure 26 for the 500k-record set and Figure 27 for the 5M-record set. We can see that throughput now achieves a more than 100% increase with packet cache enabled or over 50% increase with query cache enabled, compared to the case without PDNS caching. The response latency is reduced by several hundred microseconds with caching. Figure 26 also plotted the performance with both packet cache and query cache enabled. Although both caches contain the results to the queries, the packet cache will be hit first and therefore the performance is similar to the one with packet cache. Overall the performance with 500k-record set and 5M-record set appear similar, which is in line with the PDNS scaling property we have seen before.

4.5.2 Impact of Negative Query Cache on Querying Performance for Non-existing Records

We have seen that the PDNS performance of querying non-existing records is much worse than querying existing records, due to many additional SOA-related lookups. This situation can be alleviated by enabling the negative query cache. With that, queries for domains that share the same subdomains may find matches at the query cache instead of going to MySQL database during the SOA lookups. Since negative query cache also shares the same physical storage with rest of the PDNS cache, the explicit cache cleanup issue will still affect it. Figure 28 shows the performance for querying non-existing records with negative query cache enabled. It can be seen that explicit cache cleanup cuts more than half of the overall throughput. After fixing this issue, the throughput of querying non-existing records reaches 7,000 qps, an order of magnitude of improvement compared to the value shown in Figure 3(b) without negative query caching. In general, if only the subdomain SOA lookups are cached, the performance of querying non-

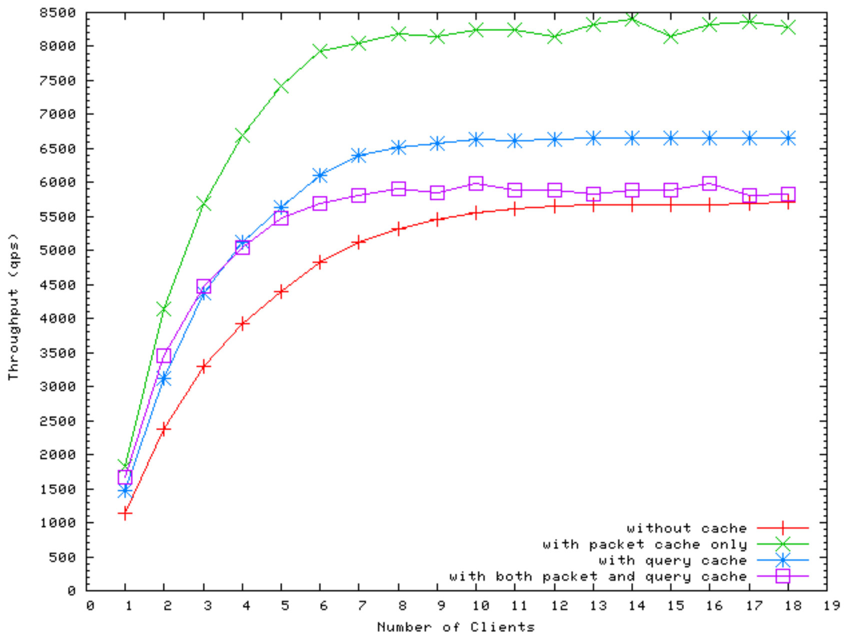


(a) Throughput

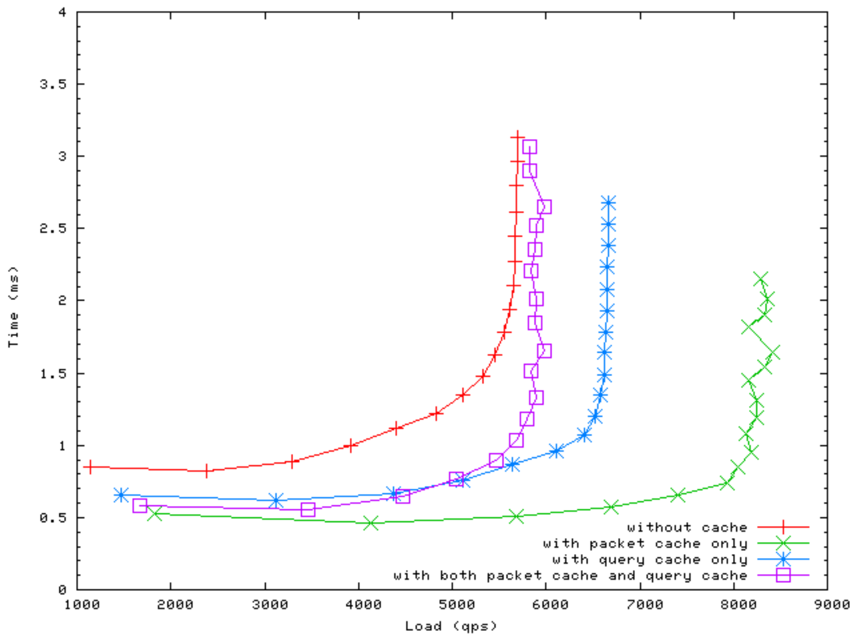


(b) Response time

Figure 24: Comparison of PDNS performance with and without caching for querying existing records with 5M-record set

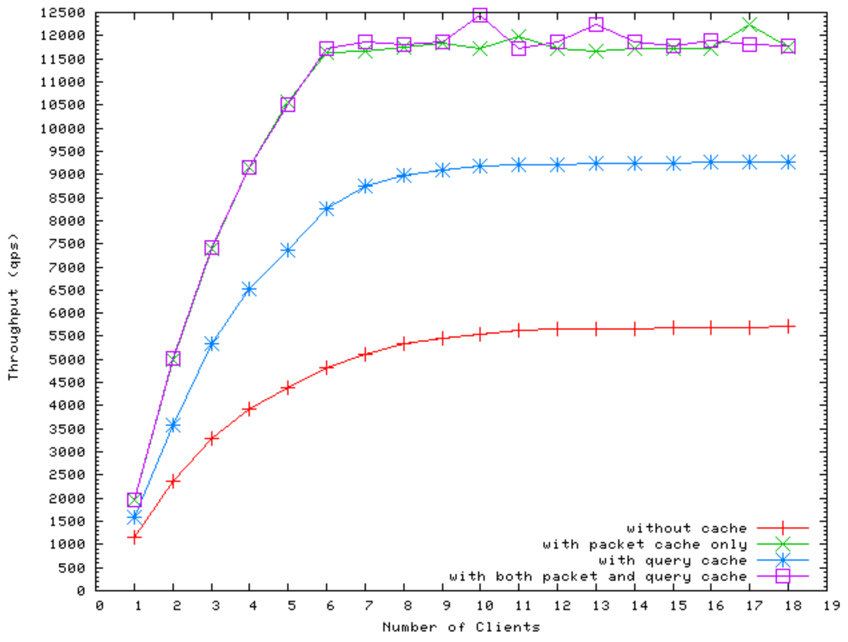


(a) Throughput

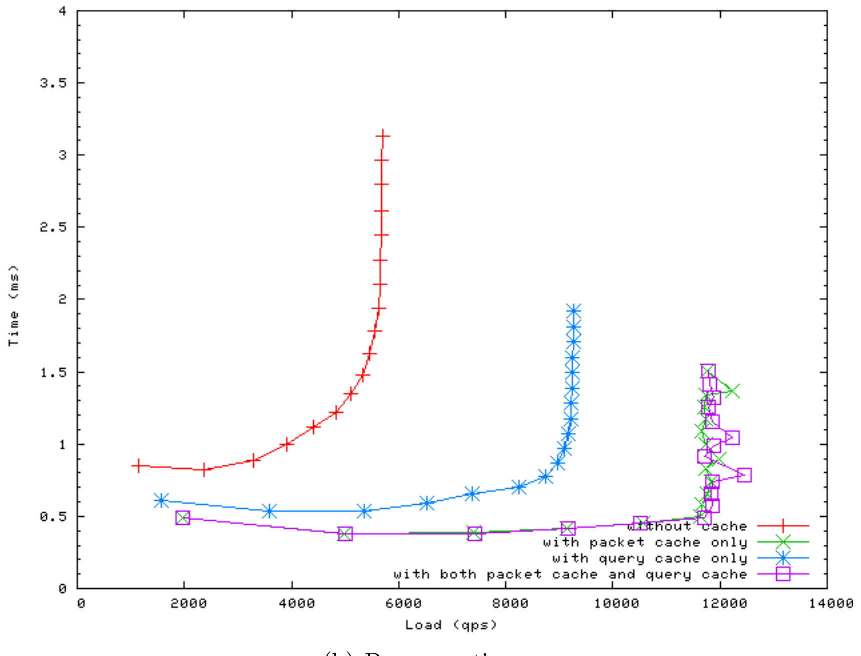


(b) Response time

Figure 25: Comparison of PDNS performance with and without caching for querying existing records with 500k-record set

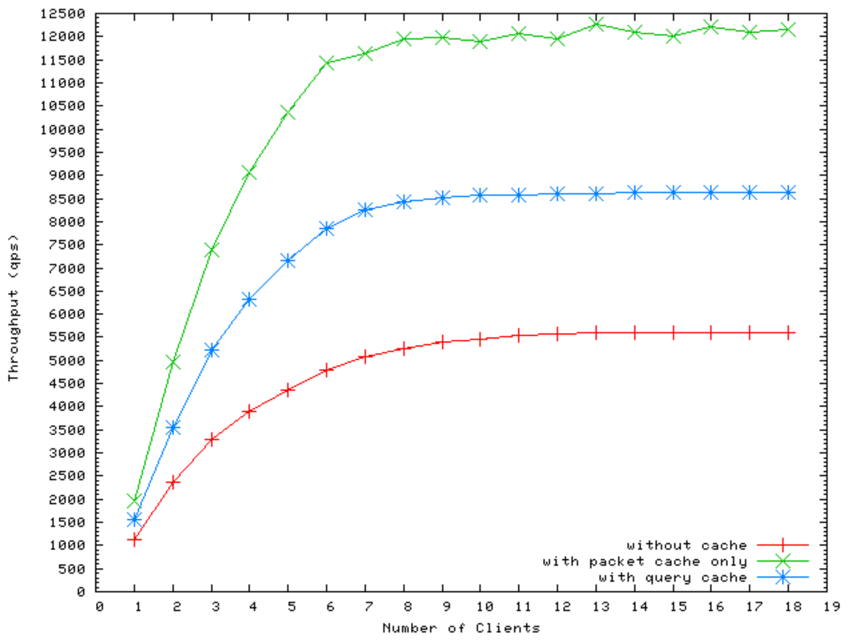


(a) Throughput

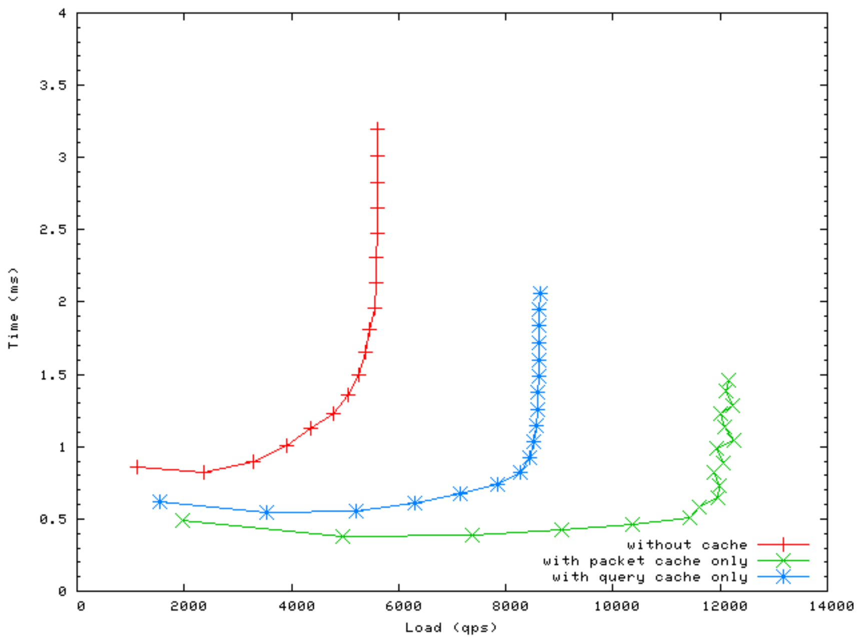


(b) Response time

Figure 26: Comparison of PDNS performance with and without caching for querying existing records with 500k-record set, after disabling explicit cache maintenance



(a) Throughput



(b) Response time

Figure 27: Comparison of PDNS performance with and without caching for querying existing records with 5M-record set, after disabling explicit cache maintenance

existing records should be lower than the throughput in querying existing records without any caching, because processing for non-existing records takes more steps than for existing records anyway as described in Section 4.3.1. This fact will cap the throughput for non-existing records to 5,500 qps according to Figure 3. However, in Figure 28 the maximum throughput exceeds this value. This is because once the negative query cache is enabled, it also caches the responses for negative queries themselves. Even in that case, the maximum throughput with negative query caching will not exceed the maximum throughput in querying existing records with the query cache enabled, which is 8,500 qps according to Figure 24(a).

It should be noted that the actual number of subdomain SOA entries that will be cached depends on the negative query space. Larger negative query space will require more subdomain SOA entries to be stored before the caching benefits can be shown.

4.6 Performance under Database Update Load

An important requirement for an ENUM server is to keep high query performance under database update load. In this section we look at the impact of database updates on the query performance and also take a snapshot of the database update performance itself.

The two major database update operations in a real ENUM deployment are adding new records and updating existing records. For example, when a new corporate customer is brought in, new zones of records need to be added; when existing customers change their telephone numbers, their entries in the database need to be updated. Due to security and operation related concerns, most ENUM providers are unlikely to ask their customers to use dynamic DNS for record update directly. Instead they tend to use some sort of background bulk update mechanism for their databases. With our PDNS configuration, the background load is applied to the MySQL database. To create background add load, we start a MySQL process which reads in a script containing SQL `insert` commands for records that do not already exist in the database. To create background update load, we start a MySQL process that reads in a script containing SQL `update` commands for the 5M records already in the database. We then repeat our query tests for the 5M-record set, this time with either the background add load or update load.

Figure 29 shows the comparison of PDNS performance with and without background load. In general there is clear, but moderate performance degradation under background load. The maximum throughput is reduced by roughly 20%, the response time may increase by a few hundred microseconds.

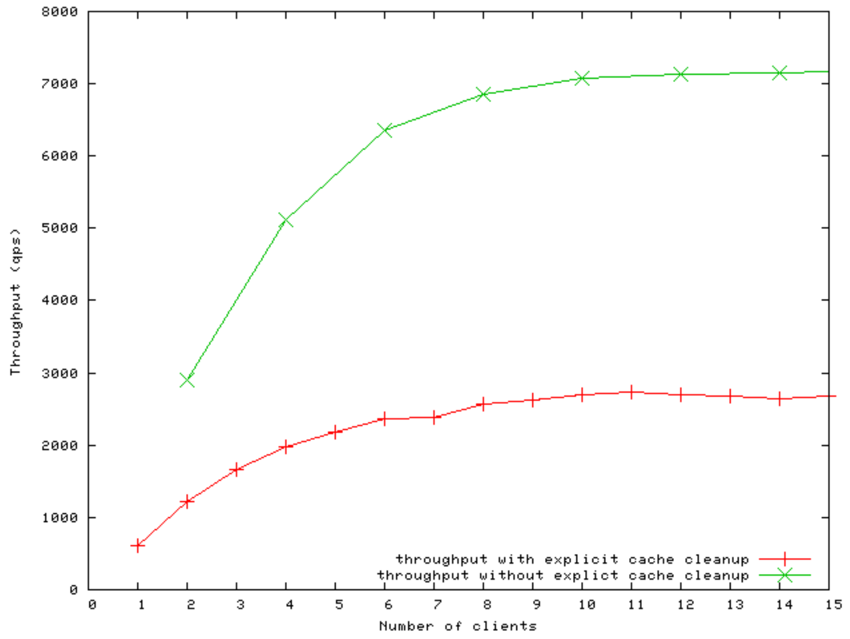
The performance under add and update load is largely similar, although Figure 29 shows that the performance under background update load seems to be better than the performance under background add load at low query load range, and the opposite seems true at higher query load range.

It should be further noted especially in the background update case the performance degradation is expected to be much more significant if the database engine is MyISAM instead of InnoDB. The former uses a table locking scheme while the latter uses row-level locking.

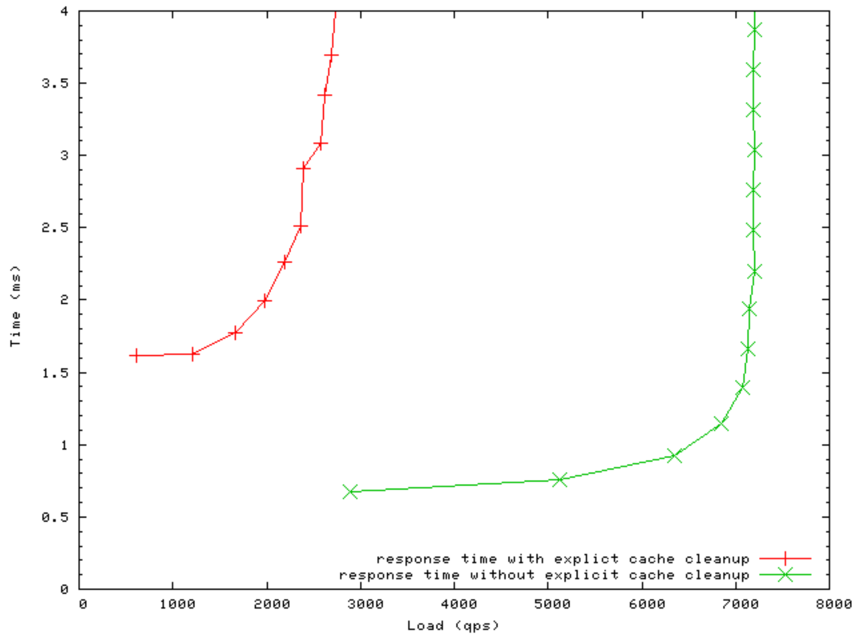
In addition to the above test, we did another test to evaluate the time it takes to update a given number of records during normal server operation. This test is performed by using three query clients to generate a load level within the server's normal capacity and initiating updates of 10,000 different records (that already exist in the database) every 30 s, 20 times. As a result, we found that when PDNS is at 3,000 qps, updating 10,000 records takes on average 2.7 s, i.e., an update rate of 3,700 records per second.

5 BIND Performance

This section presents the results obtained using BIND as the ENUM server. Figure 30 shows BIND performance for both existing and non-existing records with the 500k-record set and the 5M-record set. For existing records, BIND achieves a maximum throughput of 6,000 qps in the 500k-record set. Its throughput dropped linearly to less than 650 qps in the 5M-record set. The

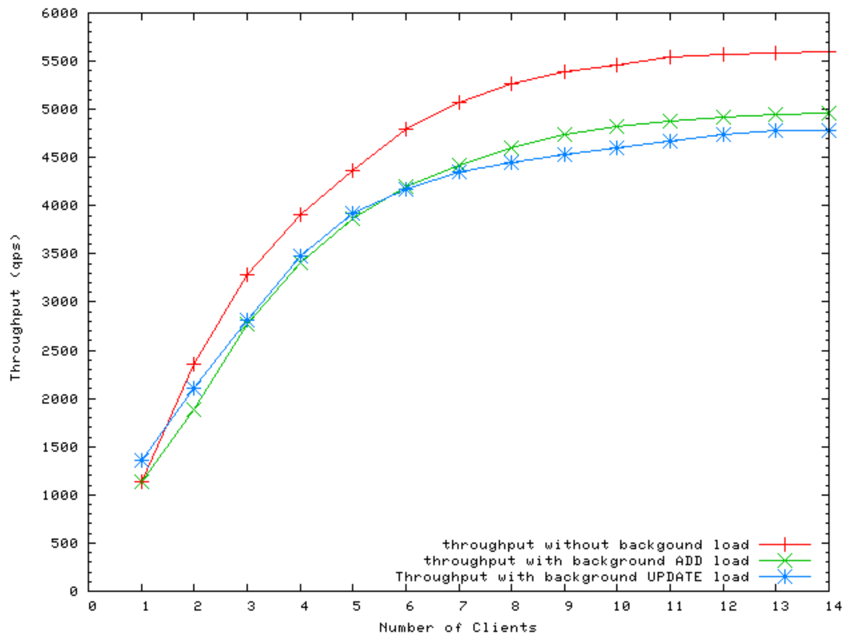


(a) Throughput

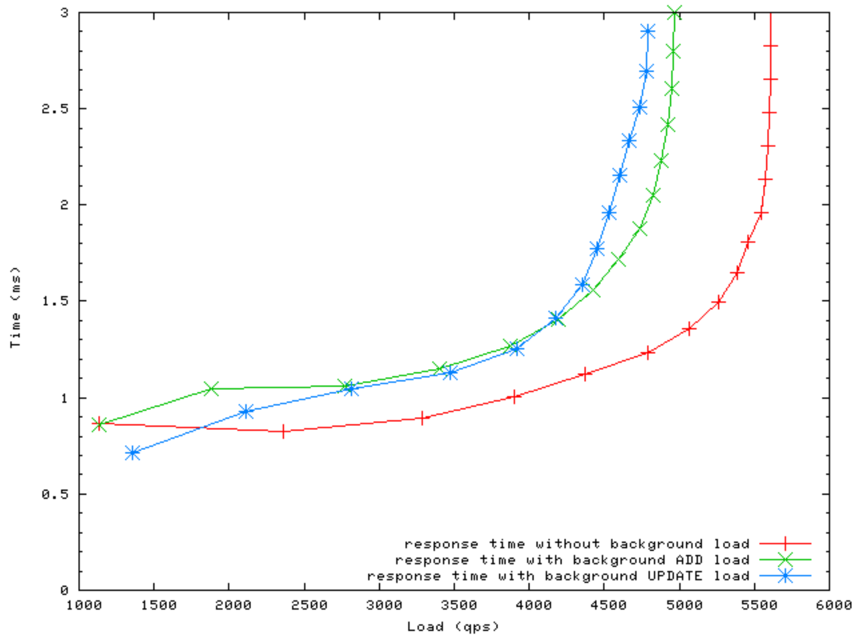


(b) Response time

Figure 28: Comparison of PDNS performance for querying non-existing records with negative query cache, explicit cache cleanup enabled vs. explicit cache cleanup disabled



(a) Throughput



(b) Response time

Figure 29: Comparison of PDNS performance with and without background load

response time in the 500-record set test is below 1 ms at most part of the normal operation range. This value increases up to three times in the test with 5M-record set.

On the other hand, BIND offers high performance for non-existing records. The performance is also independent of the database size. The maximum throughput of non-existing records for both 500k-record set and 5M-record set is over 11,000 qps. The response time is almost constant with a value less than 1 ms.

In summary, BIND has a very poor scaling property for existing records. The performance degrades linearly according to the record size. So the 20M-record set is not tested. But BIND's method to treat non-existing records scale well and offers high performance.

Updating records in BIND (other than DNS dynamic update) requires reloading the whole zone data file. The server is unable to answer queries during the loading, making BIND unsuitable for ENUM servers that need background database maintenance.

The CPU utilization of BIND in Figure 30(c) shows that BIND is CPU intensive. It consumes almost 100% of server CPU at its maximum performance in all cases.

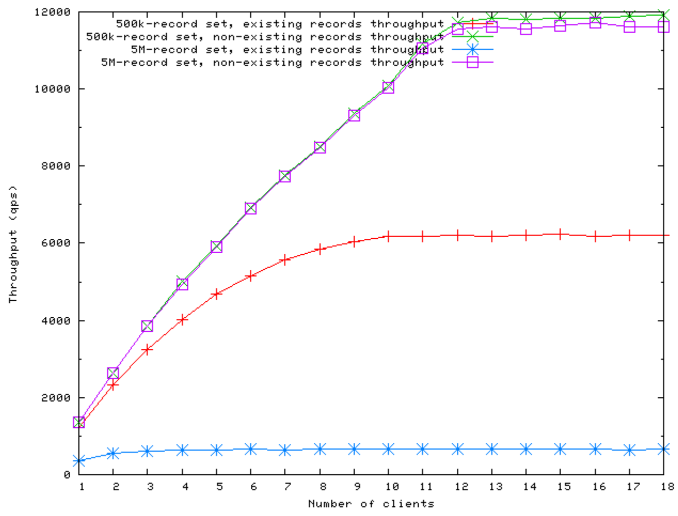
6 Navitas Performance

This section discusses the testing results obtained from Navitas as ENUM server. We tested Navitas with two types of database drivers, *vdb* and *enum_mnpz*. *vdb* is the driver for general DNS applications. *Enum_mnpz* is a driver that is optimized for ENUM applications. We used Navitas's default values for other settings, including a 100 MB cache size. The Navitas performance for both existing and non-existing records with 500k, 5M and 20M record sets is shown in Figure 31 and Figure 32. From the server statistics output, we found that at the maximum throughput, for the 500k record-set, all responses come from the Navitas cache; for the 5M record-set and 20M record-set, around 85-90% of the responses come from the Navitas cache. For existing records, Navitas achieves a maximum throughput of over 36,000 qps in all record sets. The response time is always below 0.6 ms during normal operation range. For non-existing records, Navitas achieves a maximum throughput of over 70,000 qps, almost double of the throughput for existing records. Response time for non-existing records is mostly less than 0.5 ms during normal operation range. Moreover, the Navitas CPU utilization at its maximum performance is less than 28%. In short, Navitas appears to deliver very high query performance with good scaling and resource usage property for both existing and non-existing records.

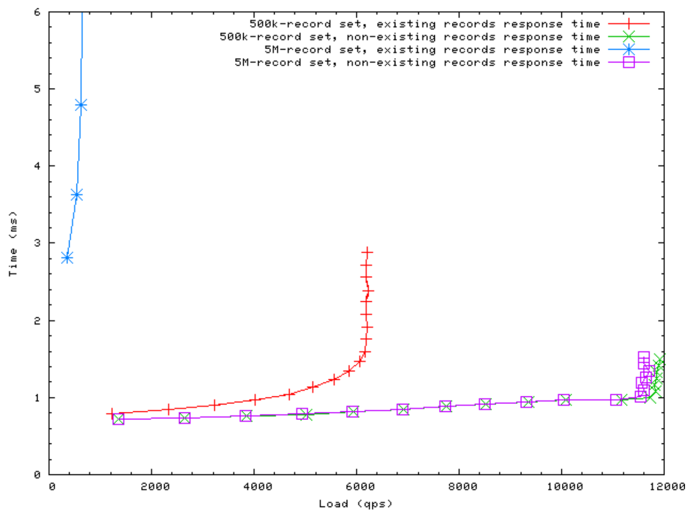
We also tested the Navitas performance under background load similar to the tests in Section 4.6. Here the background add load is generated by the `navitas_load` command [23] operating on a zone file that contains records not already in the database. The background update load is generated by using the `navitas_load -u` command [23] operating on the 5M records already existing in the database. The results using both *vdb* and *enum_mnpz* driver are shown in Figure 33 and Figure 34. Although the performance curve of *enum_mnpz* case shows more fluctuation than that of the *vdb* driver case, total maximum throughput in both cases are largely unchanged from the original value without background update load.

In another test, we evaluated the time to update a given number of records during normal server operation. Similar to the test in PDNS, we used three queryperf clients while carrying out updates of 10,000 different records every 30 s, 20 times. The results show that with *vdb* driver it takes on average two seconds to finish the 10,000 updates, corresponding to 5,000 updates per second at the querying throughput of 5,700 qps; with *enum_mnpz* driver the result is around 2500 updates per second at the querying throughput of 5,400 qps. The response time in both cases is seen to be less than 1.5 ms.

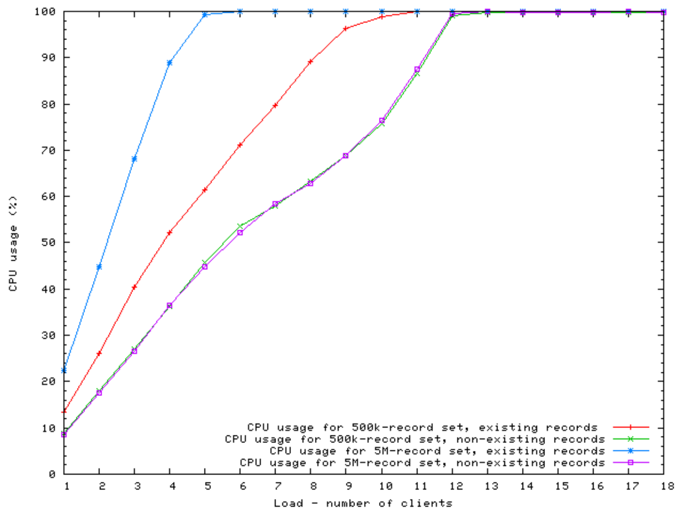
Comparing the *enum_mnpz* and *vdb* driver for Navitas, we found that the server querying throughput alone is similar in both cases. When record adding or updating loads are involved, the performance of *enum_mnpz* driver is not as good as that of the *vdb* driver. However, the *enum_mnpz* driver is found to be optimized for scalability. In fact, we observed a clear advantage in memory consumption when using the *enum_mnpz* driver for Navitas (see Section 7.6). As a result, we are able to load more records in the *enum_mnpz* driver case. In our test bed, we had problem loading more than 30M records to Navitas using the *vdb* driver. It is not clear whether this is related to the server software or the system itself. But using the *enum_mnpz*



(a) Throughput

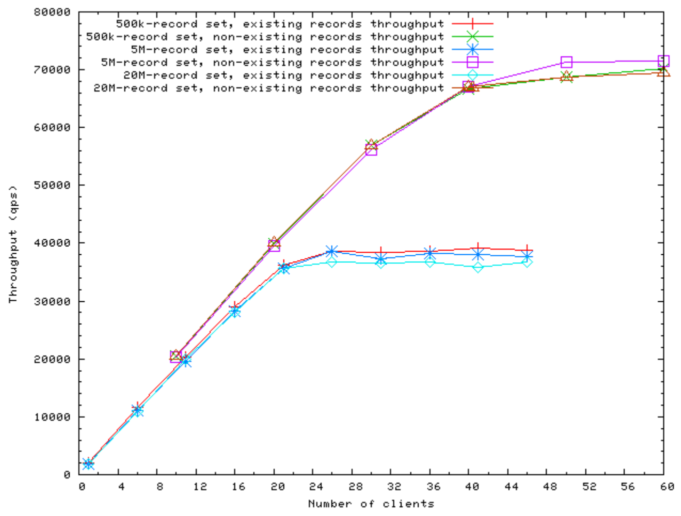


(b) Response time

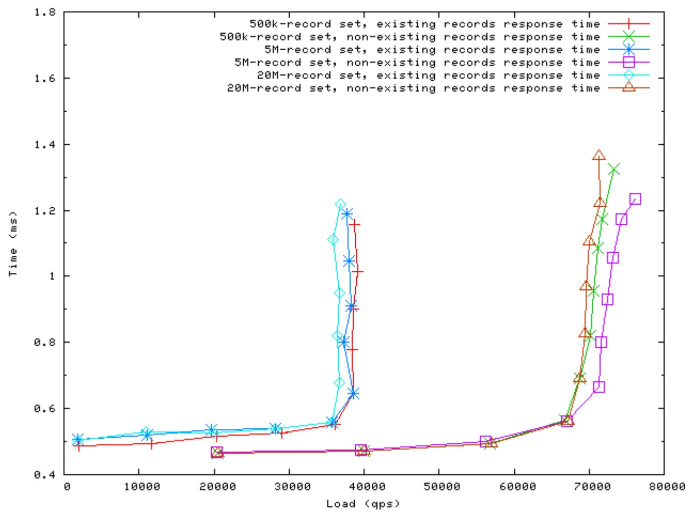


(c) CPU Usage

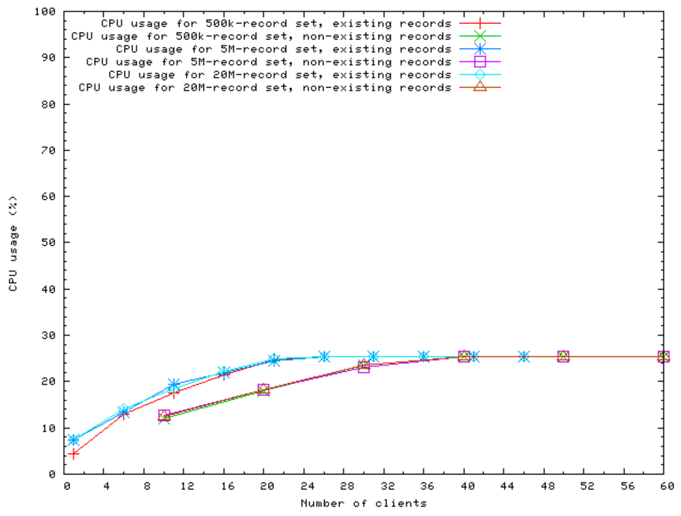
Figure 30: Comparison of BIND performance scaling



(a) Throughput

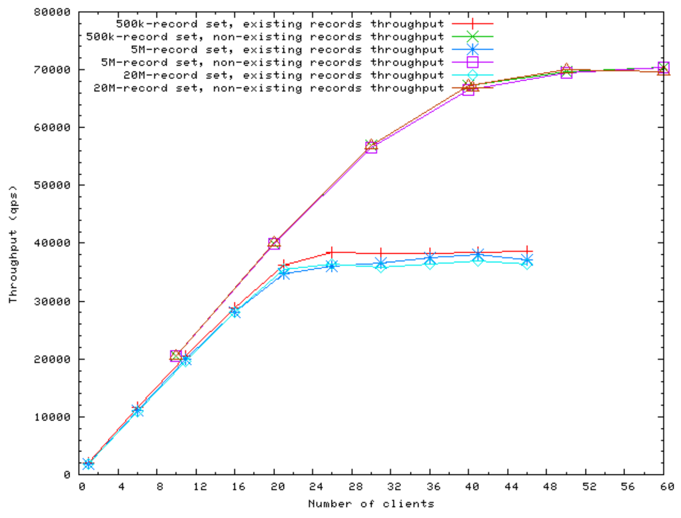


(b) Response time

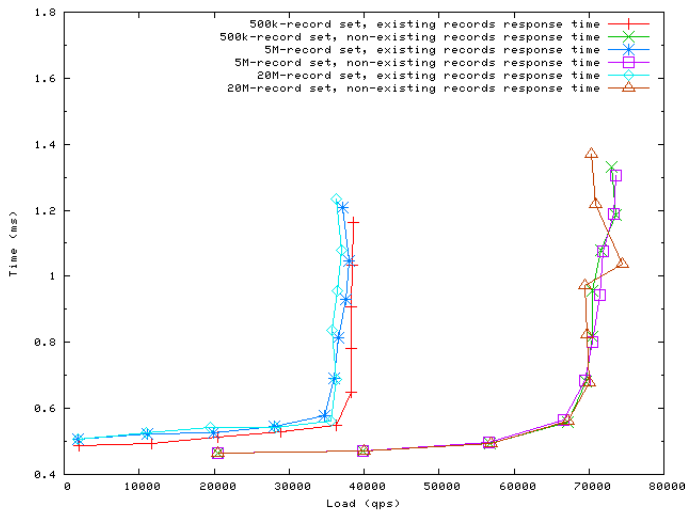


(c) CPU usage

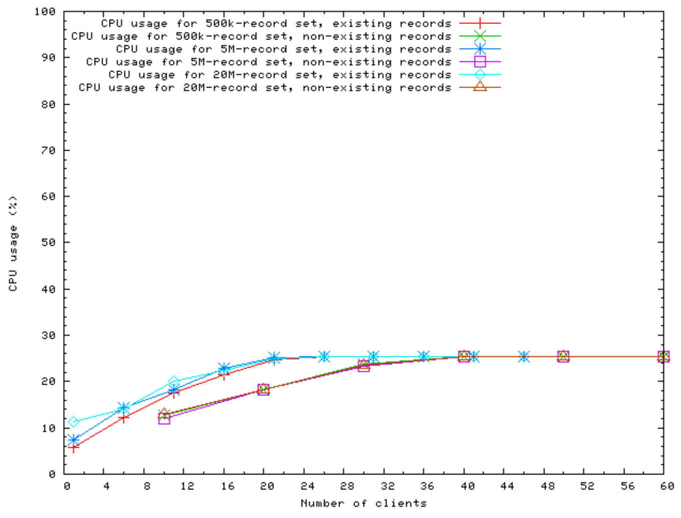
Figure 31: Comparison of Navitas performance scaling with *vdb* driver



(a) Throughput



(b) Response time



(c) CPU usage

Figure 32: Comparison of Navitas performance scaling with *enum_mnpz* driver

driver, we successfully loaded 50M records. We did not test loading more records because of system physical constraints, but it is expected that more records are loadable.

7 Comparison of PDNS, BIND and Navitas

In this section, we summarize and compare the major performance results for the three ENUM servers we have tested.

7.1 Performance for Querying Existing Records

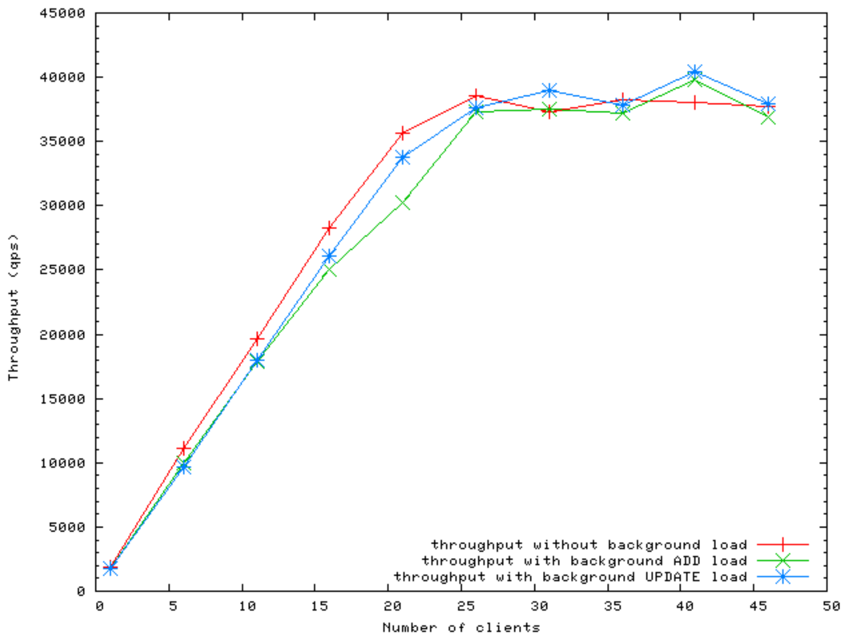
We show the comparison of throughput, response time and CPU usage for querying existing records using PDNS, BIND and Navitas in Figure 35. The tests use the 5M-record set. The values in this figure represent the best performance we obtained for the particular server under the given record set. For example, the PDNS performance is from the case where the packet cache is enabled and explicit cache cleanup disabled. It can be seen from Figure 35(a) that in terms of throughput, Navitas is clearly superior. Its maximum throughput of 39,000 qps is over three times that of PDNS and 50 times that of BIND. In terms of response time, there is a lack of data points in Figure 35(b) that makes it difficult to characterize at lower load range. However, it is still safe to argue from the figure that Navitas and PDNS offers a similar, relatively constant response time less than 0.5 ms at their normal load range. The BIND response time can be expected to be several times longer, but no longer than 2.8 ms. The CPU usage in Figure 35(c) shows that BIND and PDNS approach their maximum CPU utilization at a similar speed but the rate of increase of Navitas CPU utilization is much slower. Even though the BIND performance is the worst, it consumes virtually all 100% of the total CPU resources. Both Navitas and PDNS consume only around 25% of total CPU resources at the maximum throughput.

7.2 Performance for Querying Non-existing Records

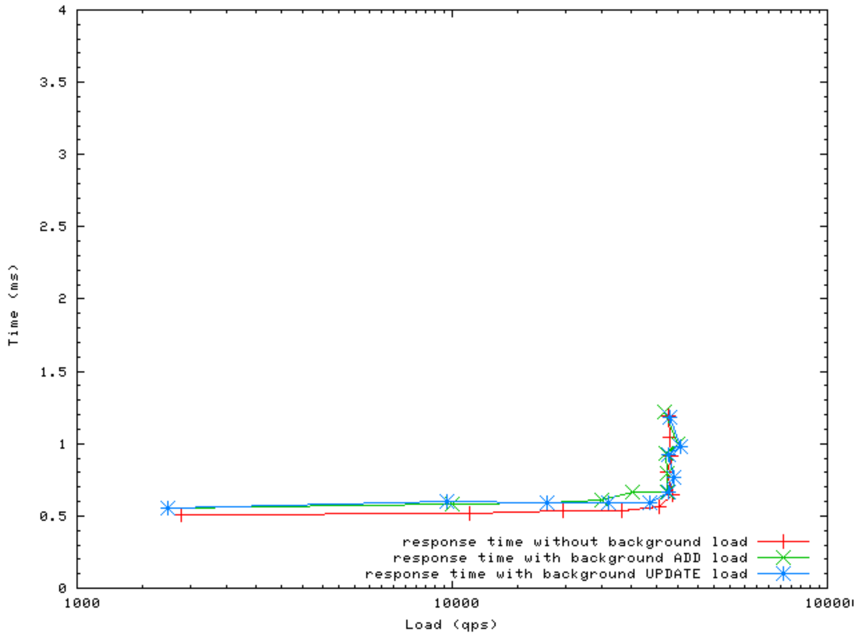
The comparison of throughput, response time and CPU usage for querying non-existing records using PDNS, BIND and Navitas is shown in Figure 36. The tests use the 5M-record set. The values in this figure represent the best performance we have obtained for the particular server with the given record set. For example, the PDNS performance used is from the case with negative query cache enabled and explicit cache cleanup disabled. It can be seen from Figure 36(a) that in terms of throughput, Navitas is still superior. Its maximum throughput of 72,000 qps is approximately six times that of BIND and 10 times of that of PDNS. Note that for querying non-existing records, BIND outperforms PDNS in throughput. In terms of response time, Figure 36(b) shows that Navitas has the shortest, with a value less than 0.5 ms at the normal server load range. The response time of PDNS and BIND are similar, both at a few hundred microseconds more than that of Navitas. As far as CPU usage is concerned, Figure 36(c) shows that BIND and PDNS approach their maximum CPU utilization at a similar speed but the increase rate of Navitas CPU utilization is much slower. The maximum CPU usage is almost 100% for BIND, 94% for PDNS and only 25% for Navitas.

7.3 Performance Limited by Server Processor Capability

Increasing CPU processing capability may improve performance if processing power is the bottleneck. We have seen that the basic PDNS throughput without caching can be increased by 40% to 45% when we double the server processor capability by putting PDNS and MySQL in two separate machines instead of in a single machine. CPU processing power may or may not be the bottleneck in other PDNS operation modes. For example, the best PDNS performance for existing records is obtained when all packets are in packet cache and no MySQL lookup is necessary. Figure 35(c) shows that there could still be over 70% spare CPU resrouces in this PDNS mode. On the other hand, the best PDNS performance for non-existing records requires negative cache being enabled, and MySQL lookup is also involved. From Figure 36(c) we can

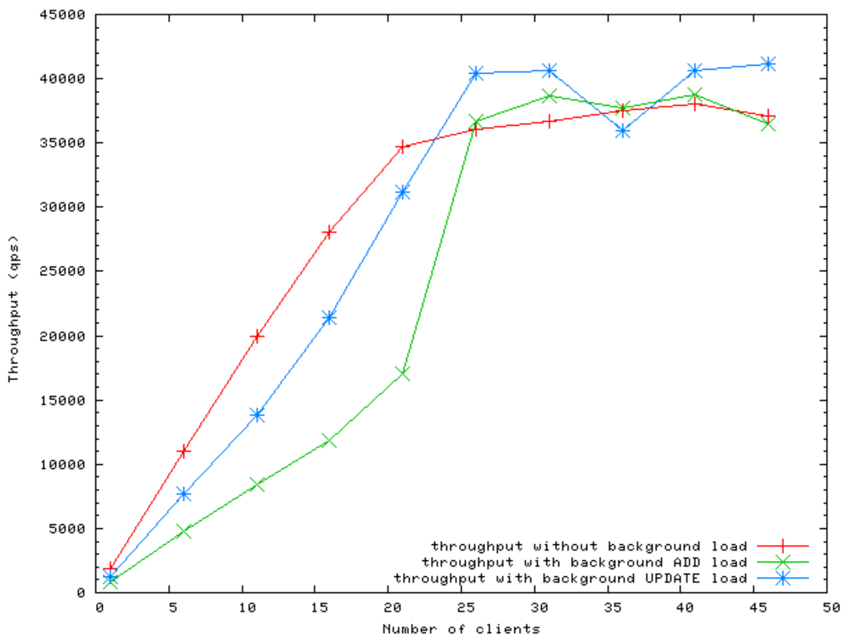


(a) Throughput

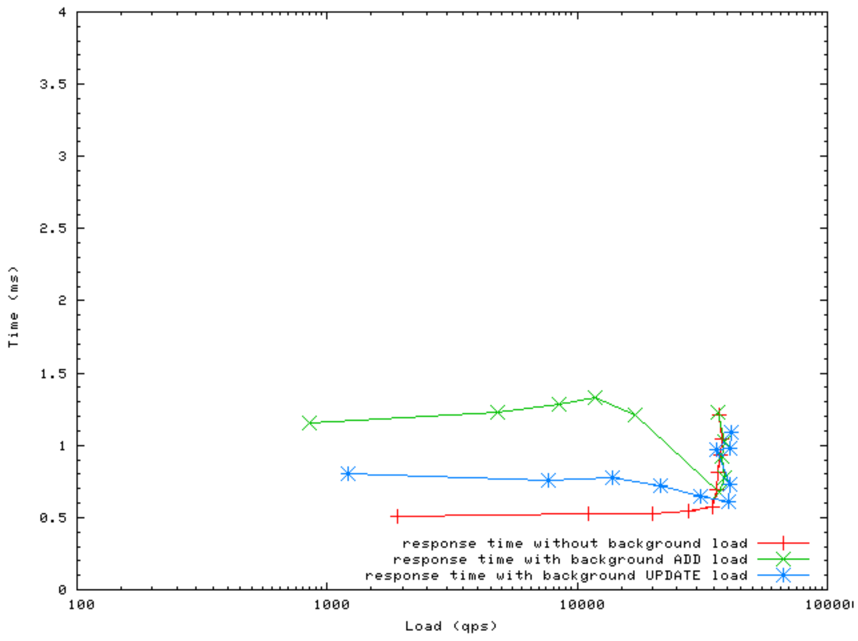


(b) Response time

Figure 33: Comparison of Navitas performance with and without background load using *vdb* driver

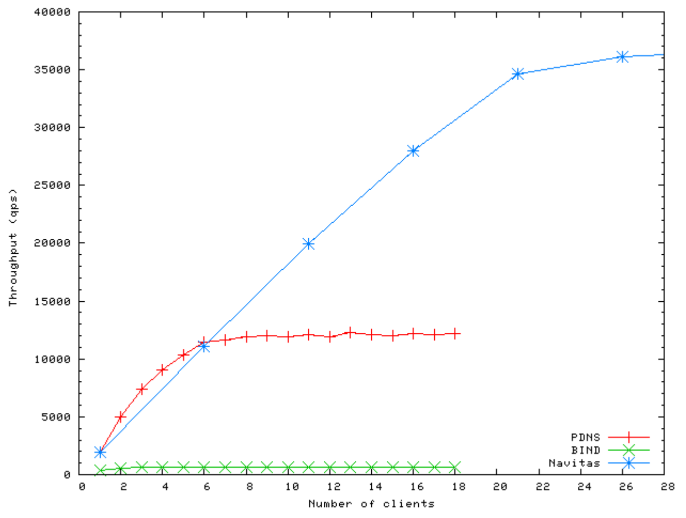


(a) Throughput

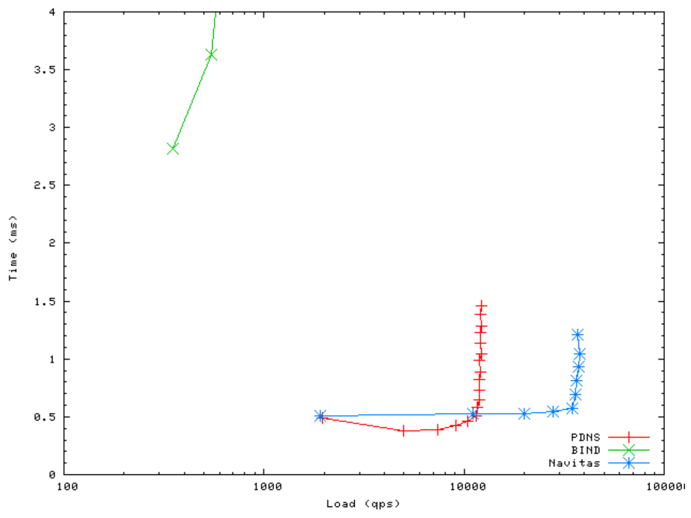


(b) Response time

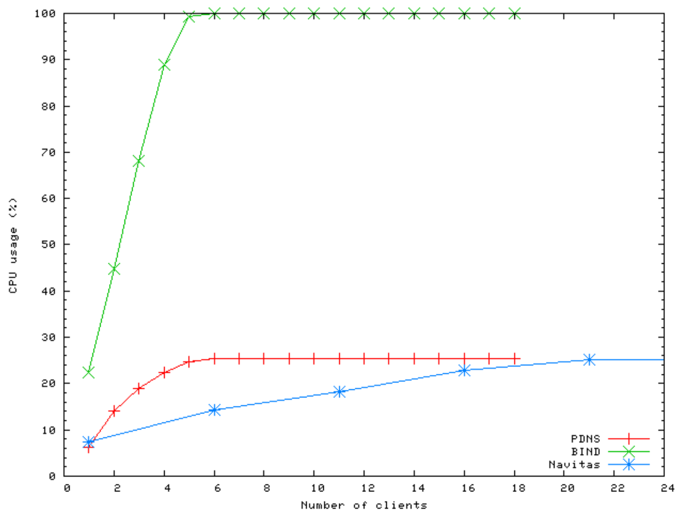
Figure 34: Comparison of Navitas performance with and without background load using *enum_mnpz* driver



(a) Throughput

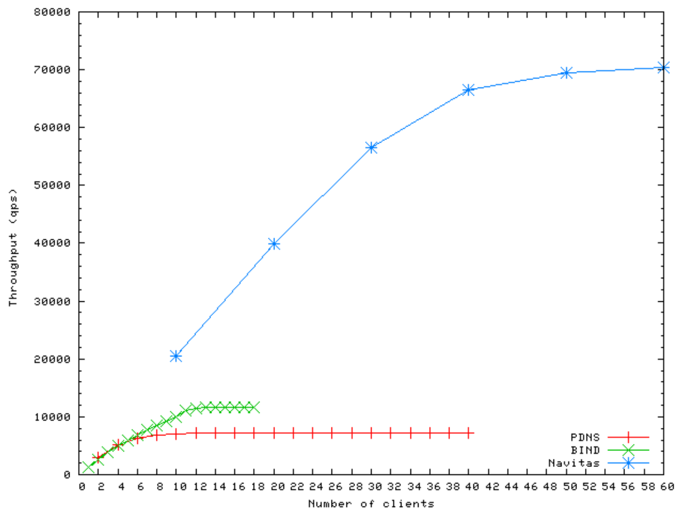


(b) Response time

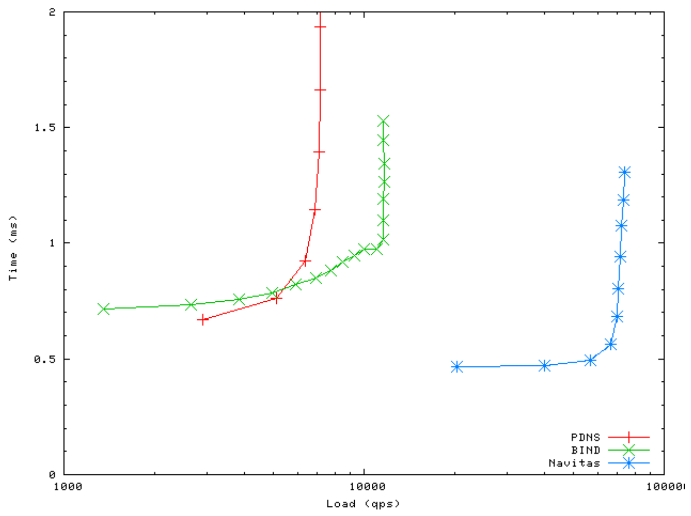


(c) CPU usage

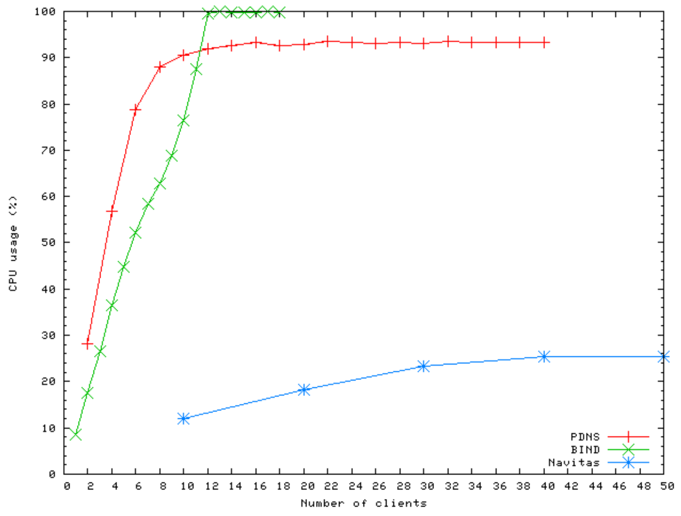
Figure 35: Performance comparison of PDNS, BIND and Navitas for querying existing records with 5M-record set



(a) Throughput



(b) Response time



(c) CPU usage

Figure 36: Performance comparison of PDNS, BIND and Navitas for querying non-existing records with 5M-record set

see that PDNS alone consumes over 90% of CPU resources. So CPU could be a bottleneck in this situation and increasing CPU processing capability might help.

Our tests also show that BIND always consumes all CPU resources available, while Navitas always consumes less than 30% of CPU resources. So increasing CPU capability is good for BIND but not necessarily useful for Navitas in our test.

7.4 Performance Limited by Database Scaling

The database scaling property of PDNS, BIND and Navitas are shown in Figure 22, Figure 30, Figure 31 and Figure 32. In summary, when records can fit into memory, Navitas offers the best performance and scaling property among the three servers. PDNS scales well and offers good performance especially for existing records. BIND scales poorly for existing records but offers better than PDNS performance with good scaling property for non-existing records. When the records cannot be fit into memory, BIND will fail to work, while PDNS performance degrades dramatically. Navitas should still work according to its manual, although we have not tested its performance difference. We will further discuss memory usage of each of the three servers in Section 7.6.

7.5 Performance under Update Load

From the query performance of PDNS and Navitas with background database update load shown in Figure 29, Figure 33 and Figure 34, we see that the maximum throughput of Navitas is largely unaffected. On the other hand, the maximum PDNS throughput appears to drop by about 20%. The response time variation under background update load for both servers is at sub-millisecond level.

The background record update rate under a normal query condition for Navitas and PDNS is summarized in Table 3. Navitas still outperforms PDNS overall. It should also be noted that query performance and update performance balance can be offset by running slaves to take queries and the master to take updates.

Parameter	Navitas (*vdb*)	Navitas (*enum_mnpz*)	PDNS
query throughput (qps)	5,700	5,400	3,700
update rate (records per second)	5,000	2,500	3,000

Table 3: Comparison of Navitas and PDNS background update rate under normal query load

7.6 Server Memory Usage

Table 4 lists the memory usage in different tests for the three server implementations we have tested.

Record set	Navitas (*enum_mnpz*)	Navitas (*vdb*)	PDNS MySQL	PDNS Cache	BIND
500k	16M+100M	55M+100M	88M+95M+360M	200M	100M
5M	160M+100M	550M+100M	880M+950M+360M	2G	1G
20M	1.2G+100M	2.3G+100M	over 3.6G+3.8G	8G	4G
mem/rec	32-60B	110B	366B	400B	200B

Table 4: Memory usage comparison of Navitas, MySQL, PDNS, and BIND

The memory usage of Navitas consists of two components. The first component is the basic Navitas memory usage after initially loading all the records. The value of this component when using the *vdb* driver is proportional to the database size, which is 55M, 550M and 2.27G for the 500k, 5M and 20M record sets. The basic memory usage of *enum_mnpz* driver depends on the actual zone structure. With the same zone structure we found its memory usage to be linear. In our tests, the 500k and 5M record sets have the same zone structure and they consume 16M and 160M memory space, respectively. The 20M record set has a different zone structure, it consumes 1.2G of memory space. The second Navitas memory component which is around 100 MB corresponds to the Navitas default cache size.

When PDNS works in the basic mode without using its own cache, memory is mostly consumed by MySQL. When PDNS works in caching mode, the number of MySQL lookups can be greatly reduced but may not be eliminated. For example, when there are queries for new non-existing records. Therefore, whether the MySQL data file and index file can be put in memory is important to either PDNS working mode.

The memory usage of MySQL consists of three components. The first one is the data file, the second one is the index file. The values of these two components are proportional to the database size. The data file size is 88M, 880M and 3.6G for the 500k, 5M and 20M databases. The index file size is 95M, 950M and 3.8G for the 500k, 5M and 20M databases. The third component is 360M for both the 500k and 5M database. In the 20M database records case, the total data file and index file size is 7.4 GB which is larger than our 4 GB maximum memory space allocated to MySQL, so the actual memory usage is 4 GB and the value of the third component is not obtained.

The memory usage of both PDNS and BIND increases linearly with the number of records in the database as shown in the table. The 20M database records case for PDNS and BIND were not tested, but their memory usage could be estimated as 8 GB and 4 GB, respectively.

To compare memory usage of the different ENUM servers, we computed the memory usage per record for each server in the last row of Table 4. The Navitas *enum_mnpz* driver is best optimized for scalability. It achieves 32B and 60B per record in the two zone structures we tested. The memory consumption of Navitas *vdb* driver is a few times more than that of the *enum_mnpz* driver. BIND consumes double of the Navitas *vdb* driver memory size, and MySQL and PDNS cache further double the memory size required for BIND.

8 Conclusion

In this report, we measured and evaluated ENUM performance using PDNS, BIND and Navitas. Now we use the results of our findings to answer whether important ENUM server performance requirements listed at the beginning of this document in Section 1 can be fulfilled.

Accommodation of huge number of records This requirement translates into the server's ability to use memory efficiently and the server's ability to work with out of memory records. When all records can be fit into memory we have seen that the average memory size per record is 32B or 60B in Navitas with *enum_mnpz* driver, 366B in MySQL, 400B in PDNS Cache, and 200B in BIND. With our allocated 4GB memory space, the maximum number of in-memory records these servers can accommodate will be approximately 125M or 67M in Navitas *enum_mnpz*, 11M in MySQL, 10M in PDNS Cache, and 20M in BIND. Therefore Navitas is clearly the best choice when the database size is large. For PDNS, if it operates in caching mode, MySQL and PDNS cache need to share the total memory allocation and the number of records hosted will have to be further reduced. In our tests, we used 5M records which allows PDNS to operate in either basic mode or the caching mode within the 4GB memory space allocation. Although BIND can host many more records than PDNS, it is practically unusable because of its poor throughput scaling property. We have shown that even at 5M-record set level, its query throughput for existing records is almost a magnitude lower than that of PDNS.

When the database size is too large to be accommodated in memory, BIND will not be able to load. PDNS has no problem if MySQL backend is used, but we will expect a significant performance drop. In our tests when the record set goes from 5M in memory to 20M partially out of memory, the basic PDNS throughput is reduced by over 80%, and response time increased by over seven times. We have not tested Navitas serving out of memory records, but it is supported according to its manual.

High query throughput This usually refers to queries for existing records. We first disqualify BIND because of its obviously poor performance under database scaling. In our testbed, when all records fit into memory, we obtained almost flat maximum query throughput of 39,000 qps for Navitas, 11,000 qps for PDNS caching mode. In order to support higher query rate requirement, we may run multiple slave server instances. As an example, let us consider the example mentioned in Section 1, i.e., 100M people population with a target query load of 50,000 qps. If each person has at least one ENUM entry, this leads to a total database size of 100M records. Supporting such a service using Navitas may require one Navitas master server and one slave server, each with 4G to 8G memory. Supporting such a service using PDNS caching mode may require one PDNS server with three slave servers. However, each of these PDNS server system will need 36.6G memory space to fully cache the 100M records. It is possible to alleviate the memory requirement for these PDNS server systems by reducing caching and further increasing the number of PDNS servers.

Short lookup response time We mentioned that the ENUM lookup latency budget can be compared with a routing function as part of the total PSTN switch processing time, which is on the order of the 200 ms to 350 ms. Throughout our tests we have seen that the ENUM lookup response time remains on the order of sub-milliseconds or a few milliseconds as long as the server is below its load capacity. Furthermore, given the normally several seconds of PSTN post dial delay, these delays under the ten-millisecond range are unlikely to matter. Therefore, this lookup response time requirement can be met relatively easily.

High query throughput under server database update load Our tests show that at the presence of background update load, the maximum query throughput in PDNS basic operation mode decreases by around 20%, Navitas maximum throughput does not appear to be affected. Furthermore, Navitas with *enum_mnpz* driver offers a background update rate of 2,500 records per second at 5,400 qps query load, while PDNS offers a background update rate of 3,000 records per second at 3,700 qps query load. In short, the performance requirement with update load is probably not a big concern for Navitas and PDNS. BIND does not meet this requirement because updating records requires reloading the zone data file and stops it from answering queries.

High query performance for non-existing records This requirement is not a concern for Navitas and BIND. Navitas offers a 72,000 qps throughput for non-existing records. BIND offers 12,000 qps for non-existing records. PDNS, however, possesses a major drawback in this requirement. Without any caching, the basic PDNS throughput for non-existing records is 1/100 of Navitas or 1/20 of BIND. By enabling PDNS negative query cache, this problem can be alleviated. But the resulting throughput, at around 7,000 qps, is still a magnitude lower than that of Navitas.

In conclusion, the poor scaling property and its way of loading and updating data disqualify BIND as a serious ENUM server. Both PDNS and Navitas can serve ENUM but Navitas outperforms PDNS significantly. We investigated ways to improve basic PDNS performance. By increasing processing power we gain 40% - 45% in PDNS throughput; by providing full caching with a modified caching maintenance mechanism, we double the PDNS throughput for existing records and increase PDNS throughput for non-existing records by almost 10 times. Even with these improvements, Navitas performs better in almost every aspect we have tested, especially in terms of throughput, where the gain can sometimes be an order of magnitude. This shows that an optimized implementation can have remarkable impact on specific server performance. There are at least three areas PDNS can be improved to optimize its performance in serving ENUM. The first is to use a more efficient cache maintenance algorithm so it will not dramatically degrade querying performance when the database is large. The second is to optimize the data structure used for ENUM records to reduce memory usage. The third is to change the handling of non-existing records. This has become its weakest point in satisfying ENUM requirements. One way is to eliminate some of the redundant operations starting from the lookup of SOA records in the query processing. But the problem with the existing approach is that all queries for non-existing records will first undergo the same processing for existing records, followed by additional steps. Therefore, the maximum throughput for non-existing records can never exceed that of existing records. This is not helpful if queries for non-existing records dominate the load, which is quite likely for initial ENUM deployments. It may be necessary to adopt a faster path in handling non-existing records to resolve this issue. As BIND and Navitas have shown, it is possible to make the throughput for non-existing records double of the throughput for existing records.

9 Acknowledgements

We wish to extend thanks to the FiberNet Telecom Group who porvided material support for this study including application and data servers and Navitas software as well as high-speed networks links between test locations. Special thanks to individual members of the FiberNet staff who provided logistic support and assisted in experiment set-up, especially: Douglas Kelly, Network Architect; Steve Cespedes, Senior Engineer; Frank Arundell, Director of Business Development, and Ernest Hoffmann, Vice President of Engineering. We would also like to thank Shawn Pouliotte and Brian Wellington of Nominum for helpful comments and clarifications.

References

[1] P. Faltstrom and M. Mealling, "The E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM)," RFC 3761 (Proposed Standard), Apr. 2004.

[2] International Telecommunications Union (ITU), "Recommendation E.164/I.331: The International Public Telecommunication Numbering Plan," May 1997.

[3] Federal Communications Commission, "Trends in Telphone Service," 2005.

[4] "Switch Processing Time Generic Requirements," Tech. Rep. GR-1364-CORE, Telcordia, June 1995.

[5] FiberNet Telecom Group, http://www.ftgx.com.

[6] Nominum modified queryperf, ftp://ftp.nominum.com/pub/nominum/queryperf-nominum-2.1.tar.gz.

[7] BIND, http://www.isc.org/index.pl?/sw/bind.

[8] Power DNS, http://www.powerdns.com.

[9] Nominum Navitas, http://www.nominum.com.

[10] Internet Assigned Numbers Authority (IANA), http://www.iana.org.

[11] International Telecommunications Union (ITU), http://www.itu.org.

[12] RIPENCC, http://www.ripe.net/info/ncc/index.html.

[13] M. Mealling and R. Daniel, "The Naming Authority Pointer (NAPTR) DNS Resource Record," RFC 2915 (Proposed Standard), Sept. 2000, Obsoleted by RFCs 3401, 3402, 3403, 3404.

[14] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, "Dynamic Updates in the Domain Name System (DNS UPDATE)," RFC 2136 (Proposed Standard), Apr. 1997, Updated by RFCs 3007, 4035, 4033, 4034.

[15] B. Wellington, "Secure Domain Name System (DNS) Dynamic Update," RFC 3007 (Proposed Standard), Nov. 2000, Updated by RFCs 4033, 4034, 4035.

[16] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," RFC 4033 (Proposed Standard), Mar. 2005.

[17] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "Resource Records for the DNS Security Extensions," RFC 4034 (Proposed Standard), Mar. 2005, Updated by RFC 4470.

[18] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "Protocol Modifications for the DNS Security Extensions," RFC 4035 (Proposed Standard), Mar. 2005, Updated by RFC 4470.

[19] B. Hubert, "DNS & PowerDNS, Convergence & ENUM+VoIP," http://ds9a.nl/pdns/pdns-presentation-ora.pdf.

[20] Nomimum, Inc., "ENUM Scalability and Performance Testing," http://www.nominum.com.

[21] Henning Schulzrinne, Sankaran Narayanan, Jonathan Lennox, and Michael Doyle, *SIP Stone*, http://www.sipstone.org.

[22] MySQL AB, *MySQL Reference Manual 4.1*, http://dev.mysql.com/doc/refman/4.1.

[23] Nominum Inc., *Nominum Navitas Administrator's Manual*, 2006.