Pp. 121-132 of the *Proceedings of LISA ’02: Sixteenth Systems Administration Conference* (Berkeley, CA: USENIX Association, 2002).

A New Architecture for Managing Enterprise Log Data

Adam Sah – Addamark Technologies, Inc.

ABSTRACT

Server systems invariably write detailed activity logs whose value is widespread, whether measuring marketing campaigns, detecting operational trends or catching fraud or intrusion. Unfortunately, production volumes overwhelm the capacity and manageability of traditional data management systems, such as relational databases. Just loading 1,000,000 records is a big deal today, to say nothing of the billions of records often seen in high-end network security, network operations and web applications. Since the magnitude of the problem is scaling with increases in CPU and networking speeds, it doesn’t help to wait for faster systems to catch up.

This paper discusses the issues involving large-scale log management, and describes a new type of data management platform called a Log Management System, which is specifically designed to cost effectively compress, manage and analyze log records in their original, unsummarized form. To quote Tom Lehrer, “I have a modest example here” – in this case commercial software that can store and process logs in parallel across a cluster of Linux-based PCs using a combination of SQL and perl. The paper concludes with some lessons we learned in building the system.

What Is a Log and Why There Is a Problem

Logs are append-only, timestamped records representing some event that occurred in some computer or network device. Once upon a time, logs were used by programmers and system administrators to figure out “what’s going on” inside systems, and weren’t of much value to business people. That’s all changed with the rise of internet-based communication, online shopping, online exchanges, and legal requirements to archive traffic and to protect privacy (a.k.a. avoid getting hacked). Unfortunately, tools to manage log data haven’t kept up with the rise in traffic, and people have reverted to building custom tools. This paper describes a general-purpose solution.

As a motivating example, one company we’ll call ABC Corp. was using a content delivery network (CDN) to “accelerate” (cache) the results of image requests from their image repository, which stored over 1,000,000 images. Unfortunately, CDNs are expensive and actually slow down the delivery performance for images that aren’t frequently accessed. In ABC’s application, the access patterns to the images were tied to promotions and other unpredictable criteria. To optimize their use of the CDN, they implemented a log management system (LMS) to capture traffic to the image repository and dynamically choose whether to use the CDN based on the frequency of access. In addition to accelerating their content, the system saved ABC $10,000 per month in network bandwidth costs.

Broadly, companies like KeyNote, NetRatings (AC Nielsen), DoubleClick, VeriSign, Google and Inktomi provide various hosted internet services, and need to report on their usage (for marketing), performance (for engineering and 24x7 operations) and conformance to service level agreements (SLAs, also for operations). Network security applications are drowning in log data, coming from system logs, routers, firewalls and intrusion detection systems (IDSs).

There are many reasons that traditional data management solutions cannot effectively manage log data, but the first one that users typically experience is in the sheer volume of log data. For example, here are some online applications and the volumes they generate:

- Loudcloud: over seven GB per day of security-related syslogs
- iPIX: over 20 GB per day for hosting photos on eBay.
- topica: over 60 GB per day logging email traffic.
- TerraLycos: 75 GB per day of weblogs from 12 major web portals.
- shockwave.com: over 24 GB per day of logs about people watching online films and playing online games.
- DoubleClick: over 200 GB/day of records about people seeing online ads.

This paper describes a Log Management System (LMS) which allows network admins to get their arms around their logs without breaking their backs. The author envisions never writing another one-off custom log analyzer, like he had to do for Inktomi (hotbot), bamboo.com (virtual tours) and Internet Pictures (eBay Picture Services).

Previous Solutions and Unresolved Problems

Until recently, most companies discarded operational logs, storing only logs of their financial transactions. Unfortunately, many companies no longer have this option: you can't figure out why online shoppers are abandoning their shopping carts before completing their purchases unless you look at the page views that **didn't** lead to sales.

One solution people attempt is to sample the data, then run reports against the samples. Similarly, people sometimes run aggregating summaries first, then report against the summaries. Both sampling and summaries suffer from the following issues. First, you have to plan everything in advance – you can't decide later what queries you want, since you've discarded the original data. Buggy sampling/summarization code results in corrupted results forever, another flavor of the "changed your mind" problem. Secondly, sampling is dangerous: if you don't sample across the correct dimension, you get the wrong answer, which can lead to bad business decisions. Lastly, a sample can't tell you whether a something didn't occur. For example, samples and summaries are not useful for security applications or when logs are stored for regulatory reasons.

Another solution is to build an LMS using off-the-shelf components, such as relational databases. Unfortunately, you still have to deal with parsing problems, sequence/session analysis and providing tools for non-experts to use, so the LMS author isn't stuck writing every query. All of these solutions also need to scale up, i.e., they need to parallelize and support paging to disk when running low on RAM. For example, parallel sequence analysis is notoriously tricky. Relational databases solve some of these scaling issues, to a point. Unfortunately, even the fastest databases can't load records as fast as enterprise applications generate them, much less provide the headroom to reload data in case something goes wrong in a load. When it comes time to run queries, they depend on "indexes" (e.g., B-trees) which accelerate some queries and not others, resulting in "cliffs" where performance suddenly degrades for no apparent reason. For example, a regular expression search in a database cannot take advantage of an index. Finally, databases are outrageously expensive, both in hardware, software and people to customize and tune them.

What Does It Mean to Solve the Problem?

Logs are generated, parsed then indexed and compressed – this then allows them to be queried and stored, respectively. As all sysadmins know, management tasks are critical, including reorganizing logs (e.g., for performance) and retiring them when no longer useful. See Figure 1 for a picture.

It is worth noting that it is usually impractical to keep logs "at the edge of the network," i.e., where they were generated. First, enterprises often require centralized reports, which becomes difficult when logs are separated by slow, unreliable networks and firewalls – or when the log-generating machines lack the storage or CPU power to effectively answer complex queries. Finally, managing widely distributed systems can be a nightmare, due to heterogeneity of hardware, operating systems, tools, access, etc.

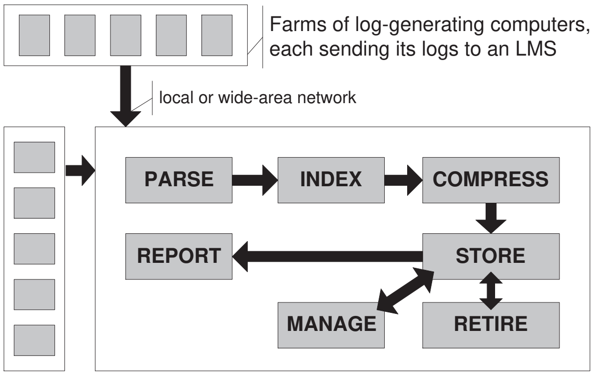

Figure 1: Workflow of log production and analysis.

It is also worth noting that scalability affects everything you do with logs: not only are excellent compressing and indexing basic requirements, but also parallel execution. Intuitively, if you have 5,000 devices generating logs, you probably need more than one collecting the results. Practically speaking, the mainframe-class system capable of keeping up with a large application's traffic costs a ridiculous amount of money.

The Vision

My vision is for a single piece of software to replace the five-minute perl hacks with solid infrastructure for handling log data. In doing so, it is key to create a community which shares scripts to parse various log formats, create various reports, etc. Ideally, there would be a dedicated group of software engineers with the time and talent to invest in features like parallel data management tools, concurrency control so you can load and query data at the same time and connectors to front-end tools like MRTG and CrystalReports.

So we built one. It's in use at places like topica, where they track over one billion emails a month. Running the LMS, five PCs running RedHat 7.1 are able to load more than 20,000 records per second (rps) of weblogs (200-600 bytes/record, depending on the site), then query them at rates of over 250,000 rps. We've handled qmail logs, apache and IIS weblogs, syslogs of various kinds, tuxedo logs and numerous custom logs. Yes, the LMS is a commercial package – it cost us several million dollars to build it.

Design Decisions for a Scalable LMS

Architecturally, the Addamark LMS looks like a webserver, only it listens for requests to a reserved URI (/cgi-app/xmlrpc/execute). If the request contains XML, the server parses the request (including data, e.g., for loading) and returns results, errors and/or progress indicators. Behind the scenes, when you connect to a server, it parses up your request, and farms it out across the cluster. Each host then parses its piece, matches table and column names against directories and files in its local filesystem (or its NFS-mounted partitions), and processes its chunk of the request.

There is a single config file listing the members of each cluster (cluster.xml) and a single config file describing the local config options for the given host (athttpd.conf). The local config file, for example, describes the paths to the data, port to listen on, etc. The LMS starts up using an /etc/init.d script. Finally, like apache, you can have multiple LMS installations per machine, and as long as they have separate paths, they can run concurrently. In fact, we even conspired to make the lockfiles compatible and the data file (backward) compatible, so two installations can share the same datastore directories, thereby allowing "rolling upgrades," which is critical for 24x7 operations, and also critical for avoiding the nightmare of reloading terabytes of data that were loaded over the course of months or years. The diagram in Figure 2 shows what a datastore file-tree might look like. Figure 3 depicts an architecture diagram for the LMS software. As you can see, we tried to avoid reinventing the wheel – even the parallel SQL engine started out as Postgres. As you can see, the network protocol is XML over HTTP, which makes it quite easy to build new clients, including test harnesses.

Loading

Requirements. An LMS should handle any type of logs, not just "standard" ones. Partly, this is

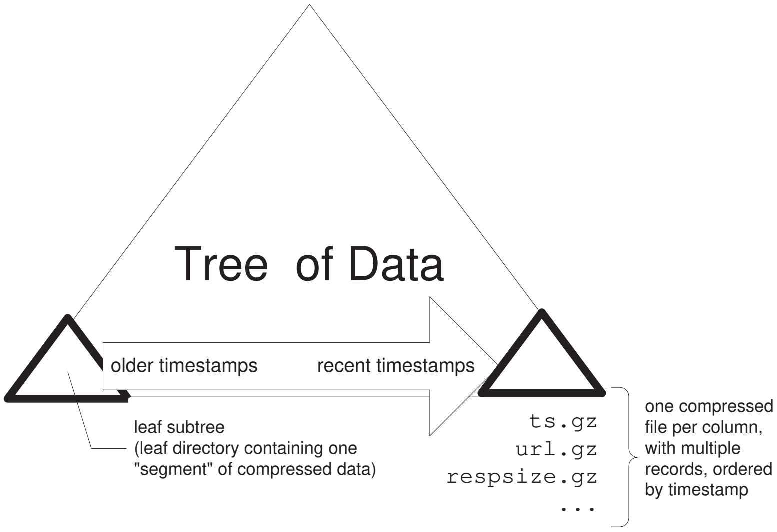

Figure 2: Sample database file tree.

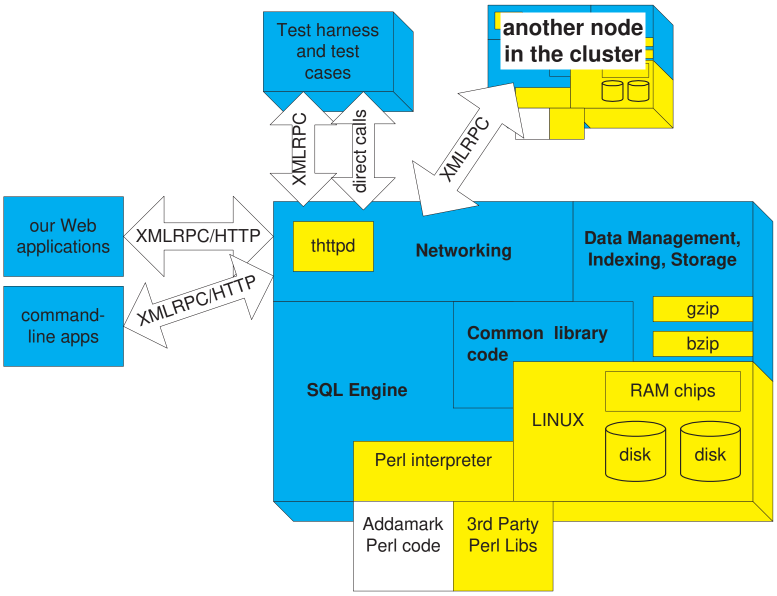

Figure 3: LMS software architecture diagram.

because the apps which most need an LMS are exactly the type who are likely to have large volumes of logs and to customize their log formats to save time or space – or to add special fields they find useful. In our experience, logs tend to exhibit these parsing issues:

- *Quoting and escapification*. Since logs are usually text data separated by some character, you need some way to handle the case when the separator character is present in a given field.
- *Binary data and internationalization*. These days, every data management tool needs to consider these issues.
- *ID fields and reverse DNS*. Logs will often contain fields whose meaning can only be discerned by looking the value up in some other database. An extreme example are IP address fields, where you might want to query on the DNS name they represent. This reverse DNS operation can be very expensive, especially when IP addresses fail to resolve.
- *Variant and XML records, name=value pairs*. Logs with variant records have different "formats" on each line, usually determined by some field in first N fields. This is typically found in custom application logs, rather than logs from commercial devices. However, XML log records are becoming more popular. Sometimes, a field will contain name=value pairs, e.g., the GET method arguments in a weblog's URL field.
- *Third party algorithms*. It is sometimes the case that you need (or want) to reuse some third party code to help parse a log record. For example, if one of the fields is encrypted, you almost certainly want to use a third party library for decrypting it.
- *Rejected record handling*. It is an unfortunate reality that log data almost always contains some number of bogus records that fail to parse. It is therefore helpful to have good support for handling rejected records when debugging parsing scripts. Likewise, in cases when "every byte counts" (e.g., legal disputes), you will want to ensure that rejected records aren't lost.
- *Excluding and double-loading columns*. Sometimes, users will want to discard a column (heresy!), e.g., to save space. Assuming you have excellent data compression, this will be less common than the case when a user will want to "double-load" a column for faster query performance, e.g., load both an IP address as well as its DNS name.

Design Decisions. For performance, we parse logs in parallel across the cluster, using a regular

```
# some example records  (for compatibility, we also support hash-comments)
# 199.166.228.8 - - [29/Jan/2002:23:44:37 -0800] "GET / HTTP/1.0" 200 7121
#               "check_http/1.32.2.6 (netsaint-plugins 1.2.9-4)" 0
# 212.35.97.195 - - [29/Jan/2002:23:45:06 -0800] "GET
#               /images/lms_overview_page1.gif HTTP/1.1" 200 17252
#               http://paulboutin.weblogger.com/2002/01/28" "Mozilla/4.0
#               (compatible; MSIE 6.0; Windows NT 5.0; Q312461)" 1
# 62.243.230.170 - - [19/Feb/2002:17:29:43 -0800] "POST /cgi-bin/form.pl
                HTTP/1.1" 302 5 "http://addamark.com/product/requestform.html"
#               "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)" 1
# vvvvvv  this is the regexp used to parse up the records  vvvvvvvvv
^... ... ... \[...\] "... ... ..." ... ... "..." "..." ...$
ClientIP:VARCHAR,unused1:VARCHAR,unused2:VARCHAR,tsStr:VARCHAR,
  Method:VARCHAR, Url:VARCHAR, HttpVers:VARCHAR, RespCode:INT32,
  RespSize:INT32, Referrer:VARCHAR, UserAgent:VARCHAR, RespTime:VARCHAR
--
-- ^^^^^^^^^ these are the assigned "parse field" names and datatypes ^^^^^^
--
--              this is the SQL statement used to transform the parse fields
-- vvvvvvvvv    (from the "stdin" table) into the final table records    vvvvvv
--
SELECT  _strptime( tsStr, "%d/%b/%Y:%H:%M:%S %Z") as ts,
        ClientIP,
        _rev_dns(ClientIP) as ClientDNS, -- perform a reverse DNS lookup
        Method,
        Url,
        HttpVers,
        RespCode,
        RespSize,
        Referrer,
        UserAgent,
        _int32(RespTime) as RespTime,  -- can also parse strings as numbers here,
FROM stdin;
```

Display 1: Example PTL script for loading an NCSA weblog.

expression designed to match single-line records. To handle multi-line records, we pre-process the data before loading, to force records onto one (virtual) line. To provide the flexibility needed, we provide a declarative language based on SQL. Roughly, a load "statement" is a SELECT from a table whose columns are the parse fields, and whose output is used to load the data. To transform a field (e.g., using builtin or third-party functions) simply place a SQL expression in the SELECT statement (the SQL "targets," as they're called); to exclude a column, just don't mention it in the SELECT statement; to double-load a column, mention it twice. For an example, see below ("Load Script Language").

In addition, we've extended our SQL to support functions written in Perl, which you can submit with any SQL statement, either at load- or query-time. In this way, you can write custom parsers and use third party libraries (e.g., perl5 modules) to parse the data. Using the new Inline Perl module (http://inline.perl.org/), you can even dynamically load code written in other languages, including C, C++, Java, Ruby, Python, etc. In practice, our users have used the Perl interface in ways we never expected. As an example, one user implemented functions to parse User-Agent tags and look for worms. In this way, he could exclude traffic that wasn't related to real users, including worms like NIMDA and robot-agents like the google crawler.

Both the regular expression match, SQL statement and any embedded Perl code are all run in parallel across the cluster. In practice, we've seen near-linear scaleups because parsing is CPU-intensive once you include all of the "business rules" of real world parsing.

The Addamark parse-transform-and-load "language" (PTL) uses a perl5 regular expression to perform the basic parse, while reusing the SQL and perl engines to perform the transformation. Display 1 shows an example PTL script for loading an NCSA weblog.

Again, it is important to note that the entire PTL script is executed in parallel across the cluster. Thus, even if you embed complex Perl functions or a multitude of complex regular expressions, you'll still be able to parse tens of thousands of records per second. For example, one customer has a PTL script which calls a home-brewed parse_useragent function on every record as it comes in, rather than doing this analysis on every query – although this improves query performance, the real value is in having the table pre-populated with the various browser attributes up-front, which makes query-writing easier.

Putting it together, Figure 4 shows the architecture diagram showing how the LMS loads data; each box represents a thread of control and set of vertically-aligned boxes represents one host. In this example, the cluster is of size three. Typically, a loading client sends the log data to one of the hosts in the cluster, which we call the "master." Any host can play "master" for any load request; the job of the master is to break up the datastream into records, and to farm those records out to machines in the cluster.

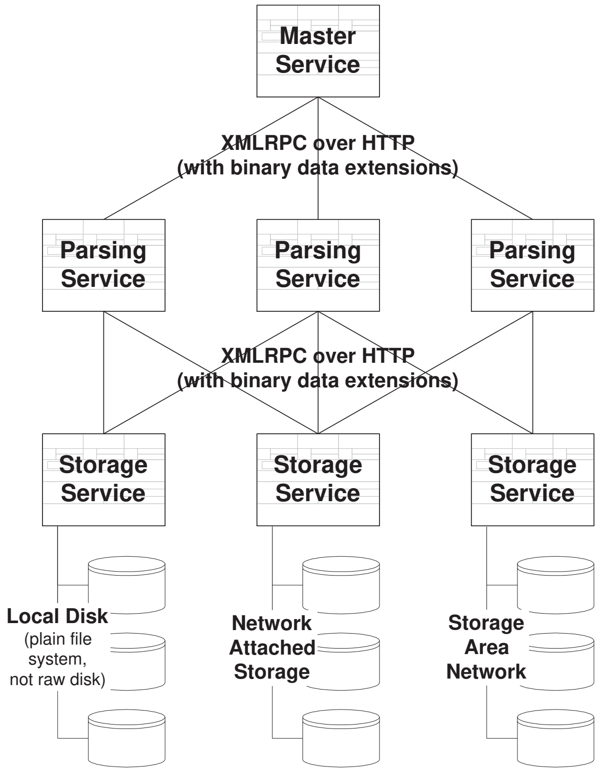

Figure 4: How LMS loads data.

Parsing and storage then happen in parallel, and finally, the records are merged into the existing sets of records on the given storage medium. As suggested by the diagram, the Addamark LMS can store its log data on either local disk, network attached storage (NAS, e.g., NFS), or on a storage area network (SAN, e.g., FiberChannel).

It's not shown in the diagram, but since the client-server and server-server are identical, it's straightforward to have client tools load directly into LMS datastore nodes, bypassing the need for a master (and the scalability bottleneck it creates), at the cost of greater configuration complexity.

Storage and Data Management

Requirements. An LMS should automatically handle all indexing, compression, storage, layout and so on – ideally in such a way that queries are then fast to run.

- **Indexing, compression, storage and data management**. Ideally, compression should be as good as GZIP, since managed storage is expensive. Compression ratios should be reasonably stable, as should indexing quality – basic queries should return in the same amount of time regardless of the log data being loaded.

- **data administration tools for giant tables**. Although the requirements are rather specialized, an LMS shares many requirements with databases – you need to control concurrent access, support incremental backup, restore and replication, deal with corrupted datastores, retire data and manage the evolution of the data definition (e.g., add/rename/remove columns).
- **Timestamp support**. Every log record has a timestamp. Unfortunately, it's rarely the case that log data is all in one format (i.e., need functions for parsing them), in GMT (i.e., need flexible timezone support) or synchronized across multiple sources (i.e., need clock sync routines). Some log data contains sub-second resolution, so it's important to handle this.
- **System admin**. If you can make the LMS "not a server," then it's a big win because system admin gets easier because it doesn't have to stay up 24x7. Unfortunately, this is probably not realistic because it leaves too much of the work for users.
  The next best design is to make the LMS into a set of CGI scripts, hostable inside a webserver, thereby reusing the 24x7 and monitoring support found in the webserver. Example issues include the ability to list/kill/pause/slow LMS operations such as loads or queries, install/uninstall/configure/reconfigure the LMS.
  It is a terrible idea to use threads or custom servers for the LMS, because you then need new tools to manage it and you'll have separate security issues, etc.
- **Security**. Access control, authentication and security are paramount issues in any data management system.

Design Decisions

- **Indexing, compression, storage and data management**. We chose gzip and bzip to perform compression for us, first parsing the data to get the best possible compression ratio. We also employ several tricks that leverage our knowledge of particular types of logs, for example encoding timestamps as delta-offsets from one another, rather than as distinct values. The net result is a compression that almost always beats gzip by a wide margin, sometimes as much as 2x better.
  We chose to store the log data as sets of plain files in the filesystem, one per column. Specifically, the files are laid out as a hierarchical set of directories, broken out by time. For concurrency control, we use lockfiles stored on local disk (e.g., /var/...) because locking over NFS can be flaky, and we figured that some users may want to store their log data on network disks.
  We're careful about touching files, allowing administrators to perform incremental replication, backup and restore using tools like rsync(1) and find(1). For more information, see "Data Administration" below.
- **Clustering and fault tolerance**. Because the use of multiple computers inherently increases the chances that one system will fail, we included automatic failover. It works by mirroring every record across two hosts in the cluster – each host has a "sibling" for the data it stores.
  When a computer fails, the hosts that are trying to contact it automatically failover to the sibling, e.g., for running queries. This causes a 50% performance degradation during failures, but 100% performance availability in their absence (unlike RAID).
  Since perfectly even distribution is not required, when a host fails, loads simply route around both a host and its sibling. Since our clusters typically start at five hosts, this leaves plenty of horsepower even during failures.
- **Timestamp support**. We support TIMESTAMP as a native datatype, represented as a 64-bit integer value of microseconds since the epoch – Jan 1, 1970. The SQL engine has the C library functions strptime() and strftime() built in for parsing and printing TIMESTAMPs, respectively.
  Timezone support is offered in all timestamp-related functions, and the SQL engine supports changing its default timezone using the clause "WITH TIMEZONE ..." To synchronize clocks, load requests from the log-generating devices can include their local clocktime, and the engine will automatically compute the difference and apply it to the log data.
  Internally, we store all data in GMT, which is simpler, but which requires users to set the timezone when printing timestamps using "WITH TIMEZONE" or in each formatting call.
- **System admin**. We chose to represent LMS operations as sets of Linux processes, allowing users to use linux tools (e.g., ps(1)) on them. These processes are launched from an off-the-shelf webserver (currently thttpd). In addition to being able to control jobs with per-machine utilities, such as nice(1), we also offer XML-RPC calls to control sets-of-processes across the cluster. We included scripts to perform cluster-wide install/uninstall/reconfigure.
- **Concurrency control**. The Addamark LMS provides a timerange-based concurrency control scheme that enables concurrent updates and queries. For example, retiring data does not block queries or new data loads. Also, two data loads will interleave in such a way as to block neither one, a critical requirement because loads can take a long time, causing timeouts in upstream processes. Unlike generalized database transactions, loads do not perform

reads in between updates, so this interleaving doesn't cause integrity loss.

- **Low-level tools**. Not that this would ever happen, but in practice files can become corrupted through software bugs, disk drive problems, etc. One customer told us a horror story about losing a person-week trying to recover data from a corrupted Microsoft SQL Server database.
  To reduce this pain, the LMS includes low-level tools to manage the data (e.g., read the files) that does not depend on the LMS being up, and which are resilient to corruption. Likewise, the Addamark LMS includes low-level tools for managing files that are replicated across a cluster, with cluster-wide diff (''cldiff''), synchronization (''clsync''), run-a-command (''clssh''), and so on.
  These tools are read the same cluster.xml config file, so membership changes affect both the LMS server and the lower-level utilities. However, for obvious reasons, the utilities can override the membership list.
- **Retirement and data evolution**. When the definition of a log changes, e.g., new columns, you need to be able change the definition in the LMS. So-called ''schema evolution'' can be handled with standard SQL statements such as ALTER TABLE ADD/DROP/RENAME COLUMN. Retiring data works using DELETE FROM, which allows you to define WHERE and DURING clauses to control what data gets deleted.
  By the time you read this, we'll have implemented a policy manager which provides a nice front-end to handle the most common cases. Also, since the goal of retirement is to save disk space, and since summaries are a tiny fraction of the size of the original data, we're adding facilities (e.g., INSERT INTO SELECT FROM) to retire-to-a-summary and utilities to implement the most common policies.
  In a clustered system, retiring to offline media can either be done per-system, or unified. The former takes advantage of the file-based storage, while the latter reuses the query mechanism, which already unifies data from across the cluster.
- **Security**. At one level, LMS security is simple: we support SSL access to the cluster. You can also use IP blocking, firewalls and/or VPNs to restrict access to selected clients. In reality, LMS authentication and access control is a complex topic, easily filling a paper all by itself. Simultaneously, this is an area of active development for us, so any information would be obsolete. Look for future reports on the subject.

Querying and Reporting

Requirements. From a high-enough level, querying an LMS is a lot like querying a database. In practice, the workload looks quite different, and the LMS should be optimized accordingly. First, for some applications, it is important to be able to quickly retrieve the original records – whitespace and all – for example, for legal use. More commonly, users want to get summaries and histograms, such as traffic per unit-time. More sophisticated queries include lookups (e.g., resolve ID fields into live data sources, rather than during loading) and sequencing/sessionizing queries (e.g., recreate web user sessions, or match activity to a given router as an ''attack''). In practice, real world use quickly demands custom filters (SQL WHERE clauses), custom counters (SQL aggregates, such as a new type of ''SUM'') and data sources outside the LMS (virtual/computed tables).

Reporting is the ''higher level'' functionality around querying, including metadata queries (''what data is available?''), query caching, presentation/formatting (e.g., Microsoft Excel, HTML, XML, etc.) and connectivity (e.g., ODBC, JDBC, DBI/DBD, etc.).

Parallel queries use a similar scheme to loading, but in reverse. Specifically, the SQL DURING and WHERE clauses get executed as part of the filtering service, then the results routed across the cluster to the ''compute'' services such that every group (i.e., GROUP BY expression) lands on the same host. To support parallel GROUP BY and SLICE BY, the groups are distributed randomly across the cluster. HAVING, which filters groups, is also implemented at the compute layer. Finally, ORDER BY and TOP-n are implemented at the compute layer, and merged together at the master to form the final result. The above description is the general case for simple aggregation queries – fancier cases like JOINs, subqueries, UNIONs, etc. are possible as well, but beyond the scope of this paper, as are the numerous optimizations that we've implemented.

Design Decisions. For querying, we chose to offer a simplified flavor of SQL, make sure it runs in parallel, then use the Perl extension mechanism to handle the custom needs of log applications. To handle sequences/sessions, we extended GROUP BY with SLICE BY, which ''slices'' a group into multiple groups based on a user-defined predicate. To handle sessions, this predicate can be stateful, e.g., 10 minutes since we've seen activity for a given user. The design of SLICE BY allows the LMS to ''sessionize'' traffic after it's been loaded – allowing you to change the business definition of a ''session'' after the fact – and it allows the LMS to sessionize traffic in parallel, a critical requirement (see Figure 5).

We offer ''system'' tables which contain lists of tables, columns, etc. For caching, formatting and connectivity, we provide a set of client-side tools and connectors. In addition, we opted to use XMLRPC-over-HTTP as our network protocol. This means that you can submit queries to the LMS using curl, lynx or even a homebrew perl script – without some fancy code library. In practice, partner companies have gotten new clients to work in under an hour, using nothing but examples.

We chose an extended flavor of SQL as the basis for querying the LMS. Display 2 shows the SQL to return the first 100 log records after midnight, Feb 1.

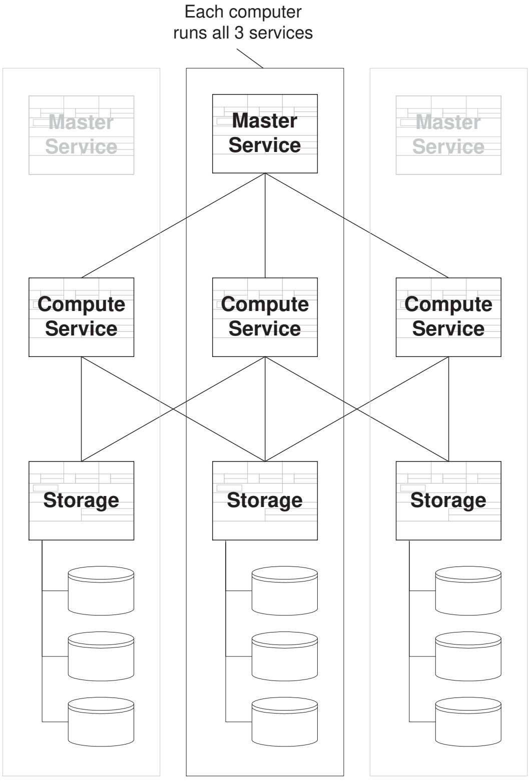

Figure 5: Database querying architecture.

The WITH clause sends various parameters to the SQL engine; these can be overridden on the command-line. In this case, we're telling the engine to produce results in California time, rather than its internal time (GMT).

The SELECT clause tells the engine what columns should appear in the results, and from which table to get them. In this case, we want all of the fields that appear in a (parsed) syslog.

The DURING clause is an Addamark extension which tells the engine which timerange you're interested in querying, so you don't accidentally query the whole table. In the rare case when you want to query everything, you can specify "DURING ALL".

To execute this query, you'd run something like:

```
atquery lms.myco.com:8072 myquery.sql
```

atquery(1) is our command-line utility for sending your SQL statement to the server, capturing the response (data, errors and/or progress indicators) and pretty-printing it to the screen, file, etc. In this example, "lms.myco.com" would be one of the systems in an LMS cluster. You can even map the LMS hosts into a "virtual IP" behind a load-balancer, which then provides additional fault tolerance.

Here's a more interesting example, retrieving a histogram of website traffic by day for the previous month.

The "WITH $foo" clauses define expression-macros, which work like C preprocessor macros. We've found macros to be lifesavers in practice, especially for clauses like DURING. Even better, the client tools support "include" which includes other files' worth of macros. This way, you can put the WITH TIMEZONE in a central file, then have every query affected by it. Finally, the tools also support overriding the WITH definitions from the command-line, allowing you to specify the $start and $end from the command-line, even though they were also given defaults in the query file. Unlike PL/SQL and other "stored procedure" languages, Addamark SQL uses Perl, i.e., an industry-standard language (Java and C++ coming soon), and the perl code automatically runs in parallel across the cluster of PCs. In practice, the CPUs on modern PCs can execute Perl code amazingly fast, resulting in terrific performance, even for complex algorithms containing numerous regular expressions.

```
-- dash-dash starts a SQL comment, much like hash (#) in the shell
WITH TIMEZONE 'US/Pacific'
SELECT TOP 100 _timef("%c", ts), hostname, progname, processID, message
FROM syslog
DURING time('Aug 08 04:00:00 2001'),time('max')
```

Display 2: SQL to return first 100 log records after midnight, Feb 1.

```
WITH TIMEZONE 'US/Pacific'
WITH $end   AS _now()
WITH $start AS _timeadd($end, -1, "month")

SELECT _timef("%m/%d/%Y", ts) as 'date'
   , COUNT(*)               as 'hits'
FROM example_websrv
GROUP BY 1   -- rollup the hits by date (result column #1)
ORDER BY 1   -- then sort the results by date (result column #2)
DURING $start, $end
```

Display 3: Retrieve a histogram of daily web traffic for previous month.

_now(), _timeadd() and _timef() are builtin Addamark functions. We chose to prefix our builtins with underscores to reserve the namespace for other uses. _now() returns the timestamp when the query was submitted to the LMS; _timeadd() is a builtin function which knows how to add timestamps correctly, including accounting for the timezone. _timef() is a function for formatting timestamps as ASCII strings, a direct mapping of the C function strftime(3).

If you also want to return the aggregate bandwidth per day, simply add a third result target – SUM(respsize)/1024.0/1024.0 AS "MB sent."

Lastly, to demonstrate the power of embedded Perl, Display 4 shows hot to compute the top 25 most popular "pages" in the website. Only, let's normalize the webpages, so that URLs like / and /index.htm don't show up separately.

Lessons Learned

Here are some of the things we learned implementing the LMS:

- **PC clustering changes everything**. Modern networks are very fast and very cheap, and the CPUs on modern PCs are also very fast, so much that it almost always makes sense to trade intra-cluster bandwidth and CPU performance for other resources, such as RAM capacity or disk I/O. For $70,000 in hardware, you can put together a system with over 100 GHz of CPU, with more switch bandwidth than the CPUs can saturate, and with 72 terabytes in storage, including a mirror copy.
- **Timestamps are a pain**. It is easy to underestimate the hassles in dealing with timestamps. As a simple example, the default routines for parsing timezones didn't recognize "PDT" (pacific daylight time) even though it's produced by the date(1) utility. If you don't solve issues like this, users get annoyed with not being able to cut and paste. Another example is the lack of a "%Z" in strptime() so you can capture the timezone from logs which contain per-record timezones. At the user-level, you'll find logs that are missing critical time fields, such as syslogs that don't include the year and weblogs lacking the timezone.
- **Buffer the logs**. Originally, we thought that users would want a library for collecting logs – but between syslog, weblogs, etc. users have plenty of logs already, they just need a management system for them! Typically, they "roll" the logs every T time, creating compressed files on the log-generating device. So-called "real-

```
WITH TIMEZONE 'US/Pacific'
WITH $end   AS _now()
WITH $start AS _timeadd($end, -1, "month")

-- this defines a new perl function, which can be called from SQL
WITH normalize_url AS 'perl5' FUNCTION <<EOF
 sub normalize_url {
 my($url) = @_;

 # in this site, index.html pages are the same as trailing-slash pages
 $url =~ s@/index.s?html?$@/@;

 # other rules go here...

 #
 # uncomment this to send debug messages back to the client tool
 # i.e., they're collecting from each of the nodes in the cluster,
 # unified and streamed back over the client-connection as out-of-
 # band messages.
 #
 # addamark::dbgPrint("hello, world");

 return $url;
 }
EOF

SELECT TOP 25 -- returns the first 25 records, assuming there's an
              -- ORDER BY to sort them.
       _perl("normalize_url", url) as url
     , COUNT(*)                    as 'hits'
     , SUM(respsize)/1024.0/1024.0 as 'MB sent'
  FROM example_websrv
 WHERE respcode < 300    -- ignore HTTP redirects and errors
GROUP BY 1
ORDER BY 2 DESC  -- this time, sort by the most-popular-first
DURING $start, $end
```

Display 4: Compute top 25 most popular "pages" in the website.

time analytics'' turned out to be a red herring: 99% of applications can live with 5-minute response times because people are involved in the chain, and they can't react faster than this. Five minutes is plenty to roll a log, compress it and send it to a centralized LMS. Wide-area collection remains a challenge especially across firewalls – but this problem doesn't seem to have a silver bullet.

- **Tag the data**. The combination of data compression and columnar storage makes ''tags'' essentially free in most cases (i.e., few unique values). Tagging can be used to provide all sorts of services, and solve all sorts of problems (for example, see ''guarantees'' below).
- **Guarantees**. In theory, end-to-end atomicity (''once and only once'') across an enterprise requires ''two phase commit.'' In practice, vendor heterogeneity and the complexity of automated recovery make this impractical. Instead, we rely on store-and-forward (aka buffering) to ensure against data loss. The downside is that duplication becomes possible. Fortunately, the same data-tagging system we use to allow users to track data back to its source also allows the LMS to detect and undo duplicate loads.
- **Packaging matters**. Packaging turned out to be surprisingly important. Our decision to ''make the LMS look like apache'' was a big win, because it was instantly familiar to users and because the config files were easy to explain. Likewise, replicating all configuration across the cluster made sense to people, while making the LMS resilient to individual machine failures. The biggest win of looking like a webserver, though, was the choice of HTTP as the network protocol, including a complete embedded webserver and a copy of the docs. This meant that our network protocol can be proxied, encrypted, tunneled, etc. – all without special support.

The Future

The existence of a scalable LMS has changed things, but much work remains. First, the combination of fast loading, aggressive data compression and PC disks has all but made log storage ''free.'' Early users would worry about running out of disk – until they did the math, and realized that even small clusters of PCs could store **Years** of data. Five PCs alone could store a month of traffic logs from all of Yahoo! This brings us to the second lesson: although you can store years of data online, and access to any (short) timerange is quick, if you want to analyze the whole thing, it's going to be slow. Therefore, you want to scale the LMS – more CPUs, RAM, etc. – according to the ''working set size'' rather than disk capacity. Thus, a balanced system would have a tiny disk drive. But extra disk capacity is cheap, so in practice users buy far more than they need. In other words, storage capacity just became free.

Demands from users have suggested our future directions. First, as users build up larger and larger recordsets, they are asking us to provide more and more facilities for managing and reorganizing this data over time. For example, as you grow a cluster, you'll want to buy the latest, fastest hardware, rather than the same model as when you started. Thus, we've recently added a way for the LMS to automatically detect performance differences between machines in the cluster, and load balance between them. Only, unlike with webservers, load balancing parallel SQL requests is quite complex, and is beyond the scope of this paper.

Second, users have started ''faking out'' the LMS by replicating the files by hand among multiple LMS clusters. While this works to some extent, we can imagine many features that would facilitate distributed, poly-clusters, with (partially) replicated data. Again, this is beyond the scope of this paper.

Thanks

This paper started as a talk at BayLISA April 18, 2002 – thanks to Heather Stern and Strata Rose Chalup for making that happen, and to Marcus Ranum and Rob Kolstad from the mother ship who helped get this paper reviewed and published. Eric Karlson, Nathan Watson, Cimarron Taylor are amazing engineers – they wrote, tested and shipped the LMS v1.0 in a year and a half, when it usually takes 10-15 engineers, and to Christina Noren, who actually made it work in production. Andy Mutz, Steve Levin, Rich Gaushell and Katy Ly (iPIX) contributed many design ideas. Thanks to the folks at topica, TerraLycos, AtomShockwave and other customers who waded through early versions. Michael Stonebraker taught me most of what I know about data management systems, and the NOW and Inktomi teams taught me about streaming, pipelining and clustering. We got big help from Mark Soloway, Dave Sharnoff, Sanford Barr, Brent Chapman, Arthur Bohren, Jeff Loomans and Dave Berger. Lastly, thanks to the angel investors who helped pay the bills and keep our spirits up through the dark days of 2001.

Software Availability

The Addamark LMS is a commercial software package available today, with introductory pricing starting around $75,000 for a complete package. Addamark also offers professional services and support. For more information, please see our website at http://www.addamark.com/.

Author Biography

Adam Sah is co-founder and CTO of Addamark Technologies, which makes software to manage enterprise log data, a source of recurring nightmares for him since 1995. Before Addamark, Adam held various management, 24x7 ops and development roles at iPIX (exclusive provider of eBay photohosting, market leader in virtual tours of real estate), Cohera (distributed database systems, acquired by PeopleSoft) Inktomi (search engines, proxy caches) and Ovid (medical research databases, now a division of Kluwer). Before joining Inktomi as its first employee, Adam was a PhD student at UC Berkeley, where he specialized in distributed databases and programming languages, and invented a way to compile TCL as part of his MS thesis, which Sun added to the language core starting in v8.0. Reach him electronically at asah@addamark.com .

References

Below are various papers ([LHMWY82], [G94], [HD90]) representing some of the key parallel and distributed database technologies that are used for storing data and processing queries. Also listed are some practitioner reports on using log data that we bumped into along the way ([HDM00], [TH99] and [GB98], [M00] and [M99]). I'm sure there are many I'm neglecting to mention.

[HDM00] A. Hume, S. Daniels, A. MacLellan. "Gecko: Tracking a Very Large Billing System," *Proc. 2000 USENIX Annual Techn. Conf.*, 2000.

[TH99] T. Dunigan, G. Hinkel. "Intrusion Detection and Intrusion Prevention on a Large Network," *Proc. First Workshop on Intrusion Detection and Network Monitoring*, 1999.

[GB98] L. Girardin and D. Brodbeck. "A Visual Approach for Monitoring Logs." *Proc. of the 12th Large Installation Systems Administration (LISA) Conf.*, 1998.

[HD90] H. Hsiao and D. J. DeWitt. "Chained Declustering: A New Availability Strategy for Multiprocssor Database Machines," *Proc. of Sixth Intl. Data Eng. Conf.*, 1990.

[LHMWY82] Lindsay, B. G., Haas, L. M., Mohan, C., Wilms, P. F., and Yost, R. A. Computation and Communication in R*: A Distributed Database Manager," *ACM Trans. Comp. Sys.,* 2(1), Feb. 1984.

[G94] Goetz Graefe. "Volcano: An Extensible and Parallel Query Evaluation System," *IEEE Trans. on Knowledge and Data Eng.*, 6(1), Feb, 1994.

[M00] M. Morton. "Logging and Critical Logs Files: The Decision to Effectively and Proactively Manage System Logging and Log Files," http://rr.sans.org/securitybasics/logging.php .

[M99] J. Mohr. "Managing Your Log Files," *Linux Magazine*, Nov. 1999, http://www.linux-mag.com/1999-11/guru_04.html .

[G02] google search result for 'linux vm "oom killer" ', http://www.google.com/search?hl=en&q=linux+vm+%22oom+killer%22