Gecko: tracking a very large billing system

Andrew Hume

AT&T Labs - Research
andrew@research.att.com

Scott Daniels

Electronic Data Systems Corporation
scott.daniels@ti.com

Angus MacLellan

AT&T Labs - Research
amaclellan@ems.att.com

Abstract

There is a growing need for very large databases which are not practical to implement with conventional relational database technology. These databases are characterised by huge size and frequent large updates; they do not require traditional database transactions, instead the atomicity of bulk updates can be guaranteed outside of the database. Given the I/O and CPU resources available on modern computer systems, it is possible to build these huge databases using simple flat files and simply scanning all the data when doing queries. This paper describes Gecko, a system for tracking the state of every call in a very large billing system, which uses sorted flat files to implement a database of about 60G records occupying 2.6TB.

This paper describes Gecko's architecture, both data and process, and how we handle interfacing with the existing legacy MVS systems. We focus on the performance issues, particularly with regard to job management, I/O management and data distribution, and on the tools we built. We finish with the important lessons we learned along the way, some tools we developed that would be useful in dealing with legacy systems, a benchmark comparing some alternative system architectures, and an assessment of the scalability of the system.

1. Introduction

Like most large companies, AT&T is under growing pressure to take advantage the data it collects while conducting its business. Attempts to do this with call detail (and in the near future, IP usage) are hampered by the technical challenge of dealing with very large volumes of data. Conventional databases are not able to handle such volumes, particularly when updates are frequent, mostly because of the overhead of performing these updates as transactions [Kor86]. (In the following, many of the names that follow, such as RAMP, are acronyms. Most of the time, the expanded version of the acronym is both obscure and unilluminating; we therefore will treat them simply as names. On the other hand, Gecko is not an acronym; it's simply a type of lizard.)

Three existing examples show a range of solutions. The bill history database, used by customer care to access the last few months of bills for residential customers billed by RAMP, uses conventional database technology and massive parallelism (many thousands of instances of IMS databases) and handles about 25% of AT&T's daily call detail volume. The SCAMP project, part of a fraud detection system, uses the Daytona database[Gre99] to maintain 63 days of full volume call detail (250-300M calls/day).

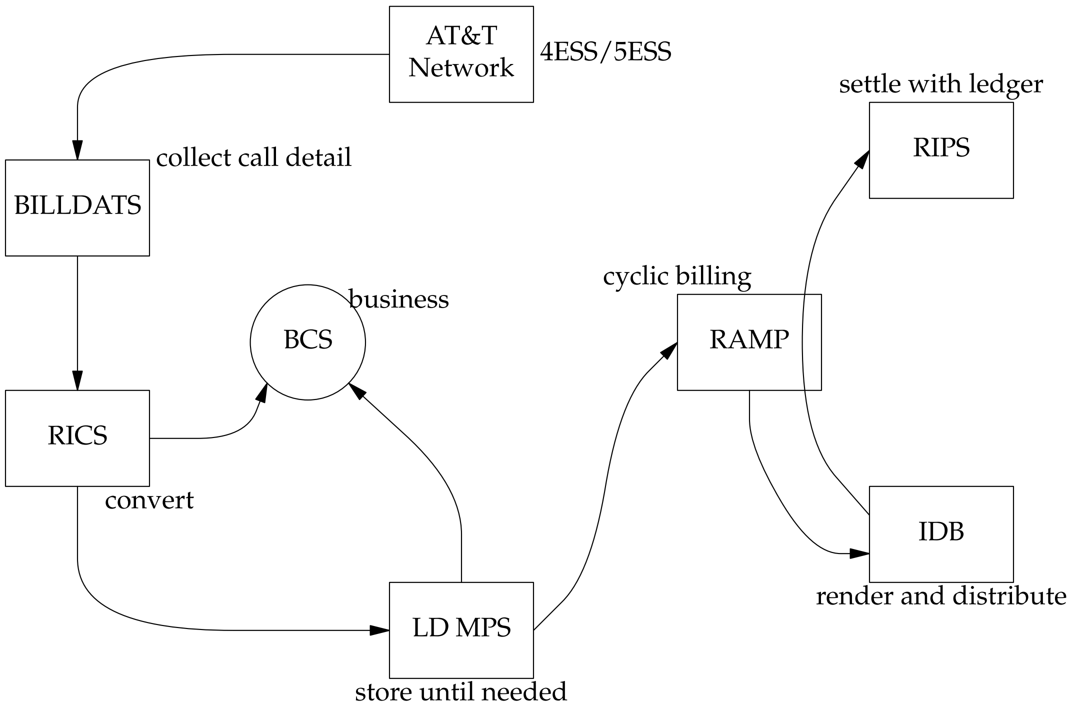

Figure 1: High level billing data flow

Each call is represented by a modest (28 fields) fixed sized record. Finally, Greyhound uses a flat file scheme to store call detail and customer account information which is then used to build various aggregate data, such as marketing and segmentation models, for marketing.

This problem, to track every message as it went through the billing process, all the way from recording through to settlement, was first raised with us in late 1995. This is an extremely hard problem; not only are the volumes huge (about four times the daily call volume), but there is no simple way to correlate the various records (from different systems) for a message. After a team at Research, including two interns from Consumer Billing, built a successful prototype in 1996, the decision was made to build a production version. A team of six people (within Consumer Billing) started in March 1997 and the system went live in December 1997.

2. The problem

The business problem

The flow of records through AT&T's billing systems, from recording to settlement, is fairly complicated. Even for simple residential billed calls, the records flow through seven major systems (see figure 1) and are processed by a few hundred different processing steps. There is considerable churn both at the process level, and at the architectural level. In the face of this complexity and change, how do we know that every call is billed exactly once? This is the business question that Gecko answers.

Gecko attacks this question in a novel way: it tracks the progress of (or records corresponding to) each call throughout the billing process by tapping the dataflows between systems and within systems. Although this seems an obvious solution, the volumes involved have hitherto made this sort of scheme infeasible.

The technical problem

The problem is threefold: we need to convert the various dataflow taps into a canonical fixed-length form (*tags*), we need to match tags for a call together into *tagsets* and maintain them in a datastore (or database), and we need to generate various reports from the datastore. A tagset consists of one or more tags sharing the same key (24 bytes including fields like originating number and timestamp). Each day, we add tags generated from new tap files, age off certain tagsets (generally these are tagsets that have been completely processed), and generate various reports about the tagsets in the datastore. The relevant quantitative information is

- there are now about 3100 tap files per day, totaling around 240GB, producing about 1.2B tags.
- each tag is 96 bytes; a tagset is 80+24*n* bytes (where the tagset has *n* tags).
- the datastore will typically contain about 60B tags in 13B tagsets.
- the target for producing reports is about 11 hours; the target for the entire cycle is about 15 hours (allowing some time for user ad hoc queries), with a maximum of 20 hours (allowing some time for system maintenance).

Tagsets which have exited the billing process, for example, calls that have been billed, are eventually aged out of the datastore. Typically, we keep tagsets for 30 days after they exit (in order to facilitate analysis).

Finally, there needs to be a mechanism for examining the tagsets contributing to any particular numeric entry in the reports; for example, if we report that 20,387 tagsets are delayed and are currently believed to be inside RICS, the users need to be able examine those tagsets.

3. The current architecture

The overall architecture had two main drivers: minimising the impact on the existing internal computing and networking infrastructure, and the inability of conventional database technology to handle our problem.

Our internal network support was apprehensive about Gecko because of its prodigious data transmission requirements. It was felt that adding an extra 200GB per day to the existing load, most of it long haul, was not feasible; it is close to 15% of the total network traffic. The data does compress well but it takes CPU resources to do the compression, and because our internal computing charges are based largely on CPU usage, Gecko would end up paying an awful lot. (Actually, the project could not survive the MVS cost of data compression and would have been cancelled.) There is a loophole where small systems are billed at a flat monthly rate. Thus, we have a satellite/central server design where uncompressed data is sent from the tapped systems over a LAN to a local Gecko system (satellite) which compresses the tap data and then transmits it to the central server.

The design we implemented to solve the database problem does not use conventional database technology; as described in [Hum99], we experimented with an Oracle-based implementation, but it was unsatisfactory. The best solution only stored the last state for a call, and not all the states, and even then, the daily update cycle took 16 hours. Backup at that time was horrendous, although better solutions exist now. Finally, the database scheme depended intimately on the desired reports; if the reports changed significantly, you would likely have to redo the whole database design.

Instead, we used sorted flat files and relied on the speed and I/O capacity of modern high-end Unix systems, such as large SGI and Sun systems.

The following description reflects our current implementation; in some cases, as described in section 6, this was rather different than our original design.

3.1 *High level system design:* Gecko is constructed from three systems, as shown in figure 2. Two systems, `dtella` (in Alpharetta) and `tokay` (in Kansas City), are simple buffer systems; they receive files from local legacy systems, compress them, and then transmit them to the central system `goldeye`. These "tap files" are received into the *loading dock*, which does integrity and completeness checks, makes archival copies on tape, and then creates tags which are put into the *tag cache*. Finally, each file in the tag cache is split up into a subfile in each of the filesystems making up the datastore. The processing for a file is scheduled when it arrives, which can be 24 hours/day.

Once a day, currently at 00:30, we perform an update cycle which involves taking all the tag files that have been split and adding them to the datastore.

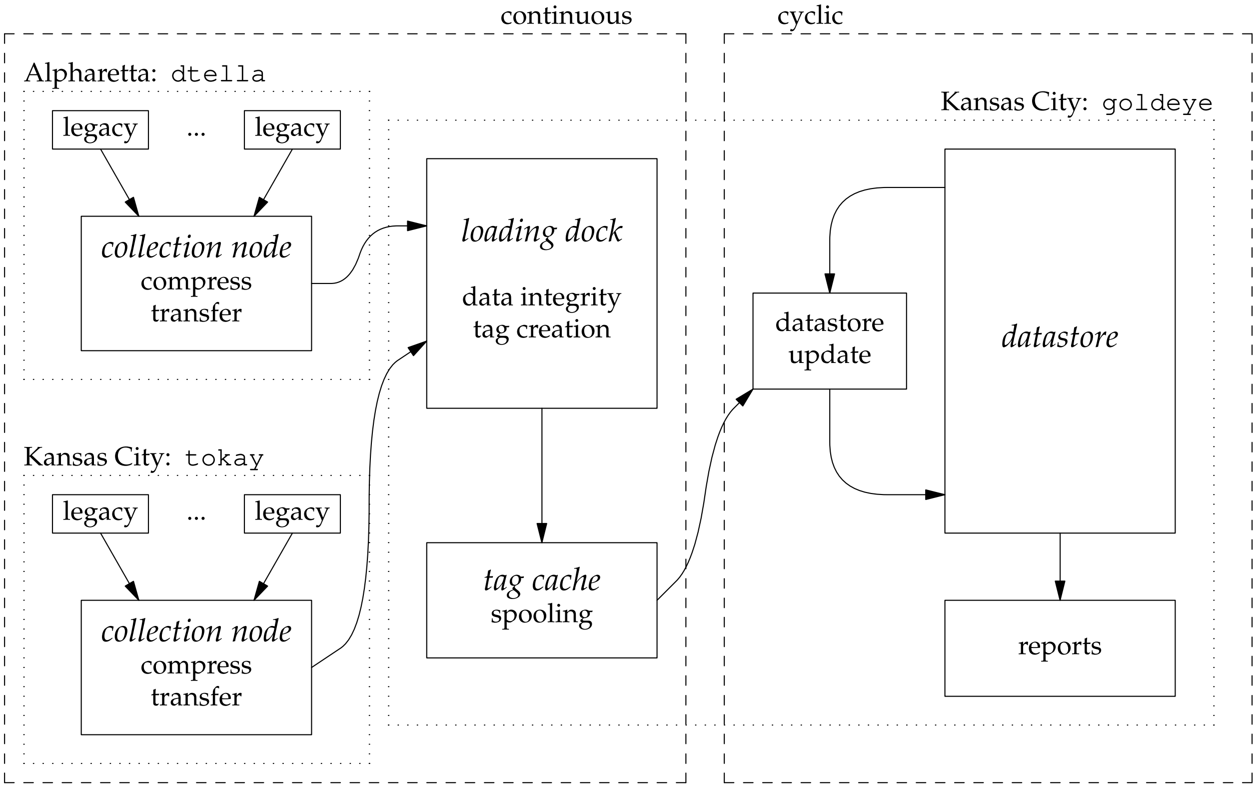

Figure 2: High level logical and system design

After all the datastore has been updated, we generate the reports.

3.2 *Data design:* The system supporting the datastore is a Sun E10000, with 32 processors and 6GB of memory, running Solaris 2.6. The datastore disk storage is provided by 16 A3000 (formerly RSM2000) RAID cabinets, which provides about 3.6TB of RAID-5 disk storage. For backup purposes, we have a StorageTek 9310 Powderhorn tape silo with 8 Redwood tape drives. Each drive can read and write at about 10MB/s. The silo has a capacity of 6000 50GB tapes.

The datastore is organised as 93 filesystems, each with 52 directories; each directory contains a *partition* of the datastore (the various magic numbers here are explained in section 3.5). Tagsets are allocated to one of these 4836 partitions according to a hash function based on the originating telephone number. Currently, the datastore is about 2.6TB. Because of Solaris's inability to sustain large amounts of sequential file I/O through the buffer cache, all the datastore filesystems are mounted as direct I/O; that is, all file I/O on those filesystems bypasses the buffer cache.

This 'feature' turned out to be a blessing in disguise because it helped us discover an unexpected and deep design paradigm: designing for a scalable cluster of systems networked together is isomorphic to designing for a single system with a scalable number of filesystems. Just as with a cluster of systems, where you try to do nearly all the work locally on each system and minimise the inter-system communications, you arrange that processing of data on each filesystem is independent of processing on any other filesystem. The goal, of course, is to make the design scalable and have predictable performance. In this case, using the system-wide buffer cache would be an unnecessary bottleneck. This isomorphism is so pervasive that when we evaluate design changes, we think of filesystems as having bandwidths and copying files from one filesystem to another is exactly the same as *ftp*ing files over a network.

This isomorphism seems related to the duality discussed in [Lau79], a duality between systems made of a smaller number of larger processes communicating by modest numbers of messages and systems comprising large numbers of small processes communicating via shared memory. By replacing 'shared memory' by 'intrasystem file I/O' and 'message passing' with 'networked file I/O', Lauer and Needham's arguments about the fundamental equivalence of the two approaches seem fresh and persuasive.

It also helps with extracting maximal performance out of each filesystem, which depends, and has always depended, on minimising the amount of disk head movement. We did this by two design principles: accessing data sequentially rather than random access, and carefully controlling the amount of simultaneous access to the filesystem. The implementation also needs care; the two most important techniques are strict adherence to a good set of I/O block sizes (we use 256KB chosen to match the stripe width of the RAID-5 configuration of the underlying storage array), and using multiple buffers so that there is always an I/O request outstanding (we use an internally developed buffering scheme, based on POSIX threads, that currently uses 3 readers or writers).

Controlling the amount of simultaneous access to each filesystem was easy because we use a single tool, *woomera*, to control the over 10,000 jobs needed to update the datastore. The relevant part of *woomera,* which is described in more detail in section 5.4, is that jobs can be marked as using various resources and it is easy to specify limits on how many jobs sharing a resource may run simultaneously. By marking all the jobs that access filesystem `gb` with a resource `LVOLgb`, we can set the limit for the resource `LVOLgb` to one and thus ensure that at most one job will access that filesystem at any given time.

3.3 *Report architecture:* Gecko is required to generate three different reports. Two of the reports describe calls that are still being processed, and the other describes the eventual disposition of calls. This latter report requires a summary of all calls ever processed. The current reporting architecture is a combination of two things. One, the history file, is a summary of all tagsets that have been deleted from the datastore. This summary is a fairly general breakdown that can support a fairly wide range of reports related to the existing reports. Thus, most changes to the reports do not require modifying the history; just its interpretation and tabulation. The history file is currently a few hundred MB in size and grows slowly over time. The second are summaries, in an intermediate report format (IRF), of tagsets still in the datastore. These latter summaries are never stored for any length of time; they are simply intermediate results for the daily update/report cycle.

More details on the report process are given below, but eventually, reports are generated. After the reports are generated, they are shipped to a web site for access by our customers.

3.4 *Process architecture:* The current processing architecture is fairly straightforward. With the exception of the first step, which occurs throughout the day, the remaining steps occur as part of the daily update cycle.

The first step is to distribute the incoming tags out to the datastore. For every source file, files with the same name are created in each filesystem (and not partition) and filled with tags that hash to any partition on that filesystem.

The next step examines all the filesystems, determines which input files will be included in this cycle, and then generates all the jobs to be executed for this update cycle. The 10,000+ jobs are then given to *woomera* for execution.

Within each filesystem, the incoming tags are sorted together in 1GB parcels. The resulting files are merged and then split out into each of the 52 partitions. The end result is the *add file*, a sorted file of tags to be added to each partition's data file. (The 1GB size is an administrative convenience; from this we experimentally tuned the various sorting parameters, most noticeably the amount of memory used.)

The next process, *pu*, updates the partition data file by merging in the new tags, deleting appropriate tagsets, and generating an IRF output for the new partition data file. The deleted tagsets are put into the *delete file*. We then generate an IRF output for the delete file. Because the underlying filesystem is unbuffered, all tag-related I/O goes through a *n*-buffering scheme (we currently use triple buffers). We generate a summary description of the delete file.

The next step, performed after all the partitions on a specific filesystem have been processed, rolls up the two different report summaries in all those partitions into two equivalent files for the whole filesystem.

The next step generates the reports for that cycle. First, we combine the summary for the deleted tagsets with the old history file and generate a new history file. Second, we combine that and the 93 filesystem summaries and generate a single set of reports.

Finally, we backup the datastore in two passes. The first pass stores all the add files and delete files. The second pass stores a rotating sixth of the 4836 datastore partitions. (This is exactly analogous to incremental and full backups.)

3.5 *Numerology:* The system layout described above contains several seemingly arbitrary numbers; this section explains their derivation.

The number of filesystems (93) is a consequence of our RAID hardware. We had 16 cabinets, each with 2 UltraSCSI channels and arranged as 7 (35GB) LUNs. We wanted all our LUNs to have good but predictable performance so we used one LUN per filesystem and 3 LUNs per SCSI bus, giving us 96 possible filesystems. Three of these were needed for other purposes, leaving us 93 for the datastore. The 7th LUN in each cabinet was used for storage, such as the tag cache, not used during the daily update cycle.

The other numbers derived from a single parameter, namely how long it takes to process the average database partition. Although of little overall consequence, this affects how much work is at risk in a system crash and how long it takes to recover or reprocess a partition. We set this at 3 minutes, which implies an average partition size of 5-600MB. For a database of 2.6TB, this is about 5000 partitions. Here's where it gets weird: midway through Gecko's production life, we had to move the hardware from Mesa (Arizona) to Kansas City. It was infeasible to suspend the data feeds during the move, so we sent some RAID cabinets ahead to hold the data, which meant we had to run with only 78 filesystems for a few weeks, but after the move we would be back to 93. Changing the number of partitions is extremely expensive (about 4 days clock time), but if the number of partitions is constant, then redistributing them amongst different numbers of filesystems (essentially renaming them), is relatively cheap (5-6 hours clock time for file copies). The least common multiple of 78 and 93 is 2418 and so we chose 4836 partitions, which meant 52 partitions per filesystem.

We chose 1GB for the parcel size as a compromise between two factors: it is more efficient to sort larger files, but larger files require much more temporary space. This latter restriction was critical; peak disk usage for sorting 1GB is 3GB which was 10% of our filesystem. Our average disk utilisation was about 85%, and thus we couldn't afford larger files.

4. Current performance

We can characterise Gecko's performance by two measures. The first is how long it takes to achieve the report and cycle end gates. The second is how fast we can scan the datastore performing an ad hoc search/extract.

The report gate is reached when the datastore has been completely updated and we've generated the final reports. Over the last 12 cycles, the report gate ranged between 6.1 and 9.9 wall clock hours, with an average time of 7.6 hours. The cycle end gate is reached after the updated datastore has been backed up and any other housekeeping chores have been completed. Over the last 12 cycles, the cycle end gate ranged between 11.1 and 15.1 wall clock hours, with an average time of 11.5 hours. Both these averages comfortably beat the original requirements.

There are a few ways of measuring how fast we can scan the datastore. The first is *tagstat*, which is a C program gathering various statistics about the datastore. The second and third are different queries to a SQL-like selection engine (*comb*). The second is a null query `1`, which is always true. The third is a simple query which selects tagsets with a specific originating number and biller: `(onum == 7325551212) && (bsid == 34)`. The speeds and total run times are

query	run time	speed
tagstat	71 min	606MB/s
null	110 min	392MB/s
simple	170 min	255MB/s

Note that we have not yet tuned *comb* to increase its performance.

5. Tools

The implementation of Gecko relies heavily on a modest number of tools in the implementation of its processing and the management of that processing. Nearly all of these have application beyond Gecko and so we describe them here. Most of the code is written in C and *ksh*; the remainder is in *awk*.

5.1 *Reliable file transmission:* By internal fiat, we used CONNECT:Direct for file transmission; it is essentially a baroque embellishment of *ftp*. No matter what file transfer mechanism we used, we wanted to avoid manual intervention in the case of file transfer errors. The scheme used is a very simple, very reliable one:

a) the sender registers a file to go.

b) the xmit daemon transmits the file, and a small control file containing the name, length and checksum, and logs the file as sent.

c) on the receiving system, the rcv daemon waits for control files and upon receipt, verifies the length and checksum. If they match, the file is distributed (in whatever way is convenient) and is logged as a received file.

d) after the xmit daemon has been idle for some time (typically 30 minutes), it sends a control file listing the last several thousand files transmitted.

e) when the rcv daemon receives the control file list from d), it compares that list against its own and sends retransmit requests for any that appear to have been transmitted but were not received.

This has proved very resilient against all sorts of failures, including system crashes and cases where due to operator error, the file is safely received but inadvertently removed or corrupted (in this latter case, we can simply rerequest the file!).

5.2 *Parser generation:* One of the greatest challenges for Gecko is being able to parse the various data tap feeds. Roughly 60% of our input are AMA records (standard telecommunications formats). There are about 230 AMA formats; these formats rarely change, but a few new formats are added each year. The rest are described by *copybooks*, which are the COBOL equivalent of a C `struct` definition combined with the *printf* format used to print it out. We have a reference library of tens of thousands of copybooks, but the data we need to parse seems to only involve a few tens of copybooks. The copybooks change fairly frequently.

In both cases, parsing has the same structure: we convert the raw record into an internal C structure, and then populate a tag from that C structure. The latter function has to be handcoded as it involves semantic analysis of the raw record fields. The former function is completely automated and depends only on either an online database of AMA record formats or the copybooks themselves. For example, we have a copybook compiler (a rather complicated *awk* script) that takes a copybook as input and produces C definitions for the equivalent `structs`, a procedure that takes an EBCDIC byte stream and populates the C equivalent (in ASCII where appropriate), and a structured pretty-printing function that lets you look at the EBCDIC byte stream in a useful way. Most copybook changes involve adding new members or rearranging members; this is handled transparently by just dropping in the new copybook. Over 80% of the C code in Gecko is generated in this fashion.

5.3 *Job management:* Gecko performs many thousands of jobs per day. We needed a tool that could support conditional execution (stalling some jobs when other jobs failed), sequential execution (stalling some jobs until other jobs have run), parallel execution (execute as many jobs in parallel as we can), and management of these jobs. The resulting tool, a job dispatch daemon, was called *woomera* (an Aboriginal term for an implement to enhance spear throwing), and *wreq* (which submits requests to *woomera*). The essential aspects of *woomera* are:

- the two main concepts are *jobs* and *resources*.
- jobs consist of a name, priority,

optional *after* clauses (or more accurately, prerequisite conditions), an optional list of resources and a ksh command.

- resources are capitalised names and have one major property: an upper limit.
- jobs become *runnable* when their prerequisites complete. A job prerequisite means waiting for that job to complete successfully (that is, with a zero exit status). A resource prerequisite means waiting for all jobs using that resource to finish successfully.
- when a job is runnable, it is executed subject to resource limits (and certain other limits, such as total number of jobs and actual machine load). If a resource `Rsrc` has a limit of 3, then at most 3 jobs using the resource `Rsrc` can run simultaneously.
- there are various administrative functions such as deleting jobs, dumping the internal state, and forcing a job to be runnable.

The ubiquitous use of *woomera* has been of enormous benefit. It provides a uniform environment for the execution of Gecko jobs, logging job executions, and a very flexible mechanism for controlling job execution. For example, during the daily datastore update, we need to halt parsing activities. This is simply done by making all the parse jobs use a specific resource, say `PRS_LIMIT`, and setting it's limit to zero. The jobs themselves are unaware of these activities.

5.4 *Execution management:* Initially, we controlled the job flow by manually setting various resources within *woomera*. After a while, this became mechanical in nature so we automated it as a ksh script called *bludge*. Every few minutes, *bludge* analyses the system activity and determines what state the machine is in, and sets various limits accordingly. For example, when the tag cache becomes uncomfortably full, *bludge* sets the limit for `PRS_LIMIT` to zero so that no more tags will be produced and avoid the situation where tags would be thrown away. (Even though its a cache, recreating the data would likely involve accessing magnetic tape which we would really like to avoid!)

Bludge calculates the system state from scratch each times it runs, rather than "knowing" what ought to be happening or remembering what was happening last time. Although this is less efficient, it is far more robust and is resilient against ad hoc changes in the system environment or workload.

5.5 *Tape store/restore:* Gecko has relatively simple needs for tape operations. There are three classes of files we backup to tape:

- raw tap files, backed up 6 times a day, retention is forever, about 40GB/day.
- full datastore backups (the actual data files), once per cycle, retention is 3 copies, about 400GB/day.
- incremental datastore backups (the add and delete files), once per cycle, retention is 3-6 months, about 135GB/day.

In each case, we have an exact list of the absolute filenames to backup. Recovery is infrequent, but it also uses a list of absolute filenames.

By internal fiat we were forced to use a specific product, Alexandria. We've been assured that someone is happy with Alexandria, but we are not. We've had to build and maintain our own database of files we've backed up in order to get plausible performance out of Alexandria. Overall, our simple store/restore (a list of files) operations require over 2000 lines of ksh scripts and about 1GB of databases as wrappers around the basic Alexandria commands.

In retrospect, the tape subsystem, consisting of the StorageTek Powderhorn silo with Redwood drives and Alexandria software, was surprisingly unreliable. Of the 600 tapes we wrote successfully, about 20 physically failed upon reading (tape snapping or creasing). Alternatively, about 15% of file recoveries failed for software reasons and would eventually succeed after prodding Alexandria (and sometimes, the tape silo) in various ways.

5.6 *Gre:* The Gecko scripts make extensive use of *grep*, and in particular, *fgrep* for searching for many fixed strings in a file. Solaris's *fgrep* has an unacceptably low limit on the number of strings (we routinely search for 5-6000 strings, and sometimes 20000 or so). The XPG4 version has much higher limits, but runs unacceptably slowly with large lists. We finally switched to *gre*, developed by Andrew Hume in 1986. For our larger lists, it runs about 200 times faster, cutting run times from 45 minutes down to 15 seconds or so. While we have not measured it, we would expect the GNU *grep* to perform about as well as *gre*. Both tools use variants of the

Commentz-Walter algorithm [Com79], which is best described in [Aho90] (the original paper has a number of errors). Commentz-Walter is effectively the Aho-Corasick algorithm used in the original *fgrep* program combined with the Boyer-Moore algorithm.

6. What we learned along the way

6.1 *Decouple what from how:* The natural way to perform the daily update cycle is to have some program take some description of the work and figure out what to do and then do it, much like the Unix tool *make*. We deliberately rejected this scheme in favour of a three part scheme: one program figures out what has to be done (*dsum*), and then gives it to another to schedule and execute (*woomera*), while another program monitors things and tweaks various *woomera* controls (*bludge*). Although superficially more complicated, each component is much simpler to build and maintain and allows reuse by other parts of Gecko. More importantly, it allows real-time adjustments (e.g. pause all work momentarily) as well as structural constraints (e.g. keep the system load below 60 or no more than 2 jobs running on filesystem `gb`).

The decision to execute everything through *woomera* and manage this by *bludge* has worked out extremely well. We get logging, flexible control, and almost complete independence of the mechanics of job execution and the management of that execution. We can't imagine any real nontrivial production system not using similar schemes. In addition, this has allowed us to experiment with quite sophisticated I/O management schemes; for example, without affecting any other aspect of the daily update cycle, we played with:

- minimising head contention by allowing only one job per filesystem (by adding a per filesystem resource)
- managing RAID controller load by restricting the number of jobs using filesystems associated with specific RAID controllers (by adding a per controller resource)
- managing SCSI bus load (by adding a per SCSI bus resource)

We typically conducted these experiments during production runs, manually adjusting limits and measuring changes in processing rates and various metrics reported by the standard Unix system performance tools *iostat*, *vmstat*, and *mpstat*.

6.2 *Cycle management:* As described above, *bludge* manages the overall system environment by tweaking various resource limits within *woomera*. For example, when we are in the CPU-intensive part of the update cycle, *bludge* sets the limit for `PRS_LIMIT` to zero in order to prevent tap file parsing during the update cycle.

More importantly for the cycle, *bludge* ensures that processing on all filesystems finishes at about the same time, thus minimising the overall cycle length (there is significant variation in the processing time required for each filesystem, and by simple round-robin scheduling, the cycle would take as long as the longest filesystem). Recall that the datastore update jobs have a resource indicating the filesystem containing the partition, say `LVOLgb`. Typically, we run about 50 update jobs simultaneously. So if *bludge* notices that filesystem `gb` is 70% done and filesystem `bf` is only 45% done, it will likely set `LVOLgb` to zero and `LVOLbf` to 1 or 2 until filesystem `bf` catches up.

6.3 *Recovery:* For the first few months of production, we averaged a system crash every 2-3 days. This caused us to quickly develop and test effective techniques for restarting our update cycle. The two key concepts were careful logging of program start and end, and arranging that programs like *pu* were transactions that either completed cleanly, or could be rerun safely (regardless if they had either failed or just hadn't finished).

6.4 *Centralising tag I/O:* All tag I/O flows through one module. While this seems an obvious thing to do, it has meant this module is the most difficult piece of code in the entire project, and for performance reasons, the most sensitive to code and/or operating system changes. The most visible benefits have been: performance improvements (such as when we changed from normal synchronous I/O to asynchronous multibuffered I/O) are immediately available to all tools processing tags or tagsets, application ubiquity (files can be transparently interpreted as files of tags or tagsets regardless of what was in the original file), and functional enhancements (such as when we supported internal tagset compression) are immediately available to all tools processing tags or tagsets.

6.5 *Weakness of system hardware/software:* While most people would agree that Gecko is pushing the limits of what systems can deliver, we were surprised by how many system hardware and software problems impacted our production system. Most were a surprise to us, so we'll list a few as a warning to others:

- we originally had fewer, larger filesystems made by striping together 3 36GB LUNs. We expected to get faster throughput, but instead ran into controller throughput bottlenecks and baffling (to both Sun and us) performance results as we varied the stripe width.
- trying to force several hundred MB/s of sequential I/O through the page cache never really worked; it either ran slowly or crashed the system. Apparently, the case of sequentially reading through terabytes of disk was never thought of by the designers of the virtual memory/page cache code. (To be fair, large sequential I/O also seems to confuse system configurers and RAID vendors, who all believe more cache memory will solve this problem.) Tuning various page cache parameters helped a little, but in the end, we just gave up and made the filesystems unbuffered and put double-buffering into our application. (Of course, that didn't help the backup software or any other programs that run on those filesystems, but c'est la vie.)
- we ran into unexplained bottlenecks in the throughput performance of pipes.
- we ran into annoying filesystem bugs (such as reading through a directory not returning all the files in that directory) and features (such as the internal filename lookup cache has a hard coded name length limit; unfortunately all our source filenames, about 60-70 characters long, are longer than that limit!).
- it is fairly easy to make the Solaris virtual memory system go unstable when you have less physical swap space than physical memory. While this is an easy thing to avoid, it took several months before we found someone at Sun who knew this.

6.6 *Trust but verify:* A significant aspect of our implementation, and one we didn't anticipate, involves performing integrity checks whenever we can. This extends from checking that when we sort several hundred files together, the size of the output equals the sum of the sizes of the input files, to whenever we process tagset files, we verify the format and data consistency. (And recently, in order to track down a bug in our RAID systems where a bad sector is recorded every 30-40TB, we have been checksumming every 256KB block of tag data we write and verifying the checksum after we close the file!) Although this is tedious and modestly expensive, it has been necessary given the number of bugs in the underlying software and hardware.

6.7 *Sorting:* the initial sorting takes about 25% of the report gate time budget. The original scheme split the source tag files directly into each partition, and then sorted the files within each partition as part of the *pu* process step. This ran into a filename lookup bottleneck. Not only did it require 52 times as many filename lookups (once per partition rather than once per filesystem), these lookups were not cached as the filenames were too long. The current scheme is much better, but we thought of a superior scheme, derived from an idea suggested by Ze-Wei Chen, but have not yet implemented it yet. Here, we would split the original source tag files into several buckets (based on ranges of the sorting key) in each filesystem. After we sort each bucket, we can simply split the result out to the partitions appending to the add file. This eliminates the final merge pass and avoids the pipe performance bottleneck.

6.8 *Distributed design:* The distributed layout of the datastore has worked out very well. It allows a high degree of parallel processing without imposing a great load on the operating system.

Although we have not yet made use of it, it also allows processing distributed across distinct systems as well as filesystems. If we had implemented Gecko on a central server and a number of smaller servers (rather than one big SMP), then the only significant traffic between servers would be the background splitting of tags out to the smaller server throughout the day and copying the rolled up report summaries back to the central server. This latter amounts to only a couple of GB, which is easily handled by modern LANs.

6.9 *Move transactions outside the database:* Because we only update the datastore as part of a scheduled process, we can assure the atomicity of that update operationally, rather than rely on mechanisms within the datastore itself. This had several advantages, including simpler datastore code, more efficient updates, and a simple way of labelling the state of the datastore (namely, the name of the update cycle performing the updates). This label was embedded in all the add and delete files, figured prominently in all the reports, and allowed complete unambiguity and reproducibility of both datastore and reports.

6.10 *Processing MVS feeds:* Although the most obvious problem in dealing with MVS feeds has been a surprisingly large number of file header/trailer sequencing schemes, the worst problem has been a simple one: the absence of a unambiguous date and time stamp. Some feeds only have a date, and not a time of day. But even those that do have a time of day neglect to indicate a timezone. It is quite hard, therefore, to nail down in absolute terms what time the file was generated. (We guess based on the processing center.)

6.11 *Tools, not objects:* Contrary to popular trends, our approach was very much tool based, rather than object based. It seems that this is a performance issue; if you really need a process to go fast, you make a tool to implement that process and tune the heck out of it (you don't start with objects and methods and so on).

6.12 *Focus:* We did one thing that really helped our design, which unfortunately might not be applicable to most developers: we had a clear vision of what our design could do well, and we rejected suggestions that did not suit the design. This sounds worse than it is; we are able to produce all the reports that our customers have required. However, as in all architectures, the customer will ask for things that seriously compromise the basic system design, and we denied those rather than warp or bloat the design.

7. Performance comparisons

The Gecko system is quite portable, requiring a regular ISO C environment augmented by sockets and POSIX threads. This allows to port the system to different systems and do true benchmarking. This section will describe the results and price/performance for the original Sun system and an SGI system. We had intended to include a Compaq 4 CPU server but were stymied by inadequacies in PC environments easily available to us. (None of the POSIX environments for Windows, such as UWIN, support threads, and we have not yet been able to bludgeon the GCC/Linux environment and their so-called extensions into submission.) Given the poor compatibility of the various thread libraries, the only practical solution is to remove threads and depend on file system buffering to work.

We can calculate the price/performance rating for a system by combining three factors:

1) CPU speed (how fast can a single process do a specific amount of work)

2) system efficiency (how fast can the system execute a set of processes)

3) price

The overall rating is simply the product of these three numbers. Given the vagaries of computer pricing, this section will omit the pricing factor.

In more detail, we benchmarked the update cycle. (The other significant activity we do is parsing of tap files, and this is heavily CPU-bound and covered by 1) above.) The production task is to run 4836 *pu* jobs; our benchmark ran a smaller number that depended on the particular system capacity. It is infeasible to carry around terabytes of data for benchmarking. Our solution was to replicate a small group of filesystems to whatever size we need; we tracked total filesystem processing times and found `ja`† was consistently around the 25th percentile, `ea` around the 50th percentile, and `nd` around the 75th percentile (we named our 93 filesystems as `[a-o][a-f]` and `p[a-c]`). To follow the real data sizes, we replicated groups of 5: `ja ea ea ea nd`.

We measured the CPU speed by averaging the user time needed to process all unique data files in the benchmark.

Task	Resource	Cur. Value	Comments
a1	networking	250GB/day	raw data feeds into the satellites
a2	CPU	760Ks/day	compress raw data feeds
a3	networking	40GB/day	compressed data feeds from satellites to *goldeye*
a4	CPU	265Ks/day	tag creation (from data feeds) into tag cache
a5	interFS	100GB/day	tags from tag cache to datastore filesystems
b1	CPU&FS	61H	sorting new tags
b2	CPU	166H	updating the datastore
b3	interFS	2GB/day	intermediate report files from datastore filesystems to *goldeye*
b4	CPU	0.5H	final report generation
b5	TAPE&FS	4H	datastore backup

Table 1: Performance model

Efficiency is simply the ratio of total CPU time to real (clock) time. The final factors are:

Factor	Sun E10k	SGI O2K	SGI/Sun
CPU	63.2s	49.0s	1.28
Efficiency	0.183	0.357	1.95

In this case, multiplying these factors together gives about a 2.5 advantage to SGI. Of course, performance is but one factor in choosing a vendor; in our case, Sun was chosen for other reasons. In addition, your application will behave differently. But do keep in mind the issue of system efficiency; the size of this factor was a surprise to us.

8. Scalability

We are often asked whether Gecko is scalable; the answer is "Of course!" The real question is somewhat different: given a specified workload, do we have a way to predict the expected processing time and the necessary resources? Yes, we do.

The model, shown in Table 1, concentrates on the two main aspects of any batch-oriented system; moving bytes, and processing bytes. In our discussion, we'll refer to two architectures: one is the SMP scheme that we described above, and the second is a (smaller) central server connected to a cluster of small machines with the datastore distributed amongs the small machines. In the following, we will denote the amount of incoming tags for a day by i; the size of the datastore by d, and the total amount of tape throughput is t.

The first section is the work that runs asynchronously from the daily update cycle. Resources consumed by tasks a1, a2 and a3 are linear in the size of the raw data feeds. We can increase the resources for a1 and a2 at a linear rate by simply adding more satellites. The networking for a3 will scale almost linearly until we hit fairly large limits imposed by hardware limits on the number of network cards; this is unlikely to be a problem as modern large servers can support several 100BaseT or GigaBit connections capable of supporting 250+GB/hour. For the cluster architecture, we could implement the loading dock as a separate system and start replicating that. The CPU needed for a4 will scale linearly. Task a5 on *goldeye* (SMP) takes about 2 hours; perversely, it would probably run faster over a network to a cluster. It should scale linearly until we run into either backplane limits on total I/O movement or operating system bottlenecks (the two ones we've seen most are virtual memory related and file system related locking).

The second set of tasks (b?) make up the daily cycle. The tasks b1, b2, b3 and b4 can run in parallel limited only by the number of processors and independent filesystems available. In particular, tasks b1 and b2 are composed of 4836 subtasks, all of which can be run in parallel. Task b1 involves sorting and thus takes $O(i \log i)$; b2 and b3 are both $O(d)$. Task b4 takes time linear in sum of the number of days represented by data in the datastore and the history file. Task b5 will take $O(\frac{i+d}{t})$.

Our Sun implementation would probably process up to 2-3 times the daily input that we do now. Switching to a faster SMP, such as an SGI, could push that to 4-6 times. For real growth, though, you would want to go with a cluster implementation. Carefully constructed, this would make nearly every aspect of the workload scale linearly by adding more systems. The only task that doesn't is the sorting step b1, and even that could be mitigated by presorting files as they arrive on each filesystem and thus make the part of b1 necessary for the update cycle be a linear performance merge pass.

9. Conclusion

By any measure, Gecko aims at solving a very large problem. Indeed, originally it was thought that the problem was not solvable at all (and during our darkest days, we almost believed this as well). But the fact remains that we have a system in production today that handles the volumes and meets its deadlines. Furthermore, the project initially went live only 8 months after starting with all design and development done by a team of 6 people.

And even after a year, the volumes are still stunning. On an average day, we process about 240GB of legacy data, add about 1B tags to a 13B tagset datastore stored on 2.6TB of disk, and backup about 900GB of data to tape. And on our peak day (recovering from our system's move from Mesa to Kansas City), those numbers have stretched to 5B tags added and 1.2TB backed up to tape.

The datastore design, a myriad of sorted flat files, has proved to be a good one, even though it isn't a conventional database. It works, it comfortably beats its processing deadlines, and has proved flexible enough to cope with several redesigns.

References

[Aho90]Alfred V. Aho, *Algorithms for Finding Patterns in Strings*, Handbook of Theoretical Computer Science, Elsevier, 1990. pp 278-282.

[Com79]Beate Commentz-Walter, *A string matching algorithm fast on the average*, Proceedings of the 6th Internat. Coll. on Automata, Languages and Programming, Springer, Berlin, 1979. pp 118-132.

[Gre99]R. Greer, *Daytona and the Fourth-Generation Language Cymbal*, ACM SIGMOD Conference, June 1999.

[Hum99]A. Hume and A. MacLellan, *Project Gecko: pushing the envelope*, NordU'99 Proceedings, 1999.

[Kor86]H. F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1986.

[Lau79]Hugh C. Lauer and Roger M. Needham, "On the Duality of Operating System Structures", *Operating Systems Review*, 13(2), 3-19 (1979).

Acknowledgements

This work was a team effort; the other Gecko developers are Ray Bogle, Chuck Francis, Jon Hunt, Pam Martin, and Connie Smith. There have been many others within the Consumer Billing and Research organisations with AT&T who have helped; in fact, too many to list here. We have had invaluable help from our vendors, but Martin Canoy, Jim Mauro and Richard McDougall from Sun were outstanding.

The comments of the reviewers and our shepherd greatly improved this paper; the remaining errors are those of the authors. We thank Rob Kolstad for suggesting the Lauer and Needham paper.

This is an experience paper, and as such, contains various statements about certain products and their behaviour. Such products evolve over time, and any specific observation we made may well be invalid by the time you read this paper. Caveat emptor.