Google Linux servers hit with $5m patent infringement verdict (The Register)

Google Linux servers hit with $5m patent infringement verdict (The Register)

Posted Apr 21, 2011 21:31 UTC (Thu) by **orcmid** (guest, #74478)
Parent article: Google Linux servers hit with $5m patent infringement verdict (The Register)

The summary of what the patent covers is not very good here. The Nemes patents (all filed in the last century) all involve hash-indexed records becoming obsolete in some way but not being deleted at once for some reason. Instead, the records are removed opportunistically when the linked list of records having the same hash is traversed for some other reason (including the explicit deletion of one of them).

As far as I can tell, that is the novelty that is not dealt with in any of the extensive, existing work on hash-based searching of records.

I have no idea what the use is in the Linux kernel that would make it difficult to work around this, but I can think of two factors:

1. Allowing records to expire or to be marked as expired for later garbage collection could be a way to avoid locking overhead and issues.

2. It might be difficult to know and/or to keep track of the linked list that the record (the significant data, not necessarily in the linked-list entry itself) is actually on.

In any case, I suspect it will be important to work around the patent and to do so quickly.

---

to post comments

About Work-Around Difficulty

Posted Apr 21, 2011 22:12 UTC (Thu) by **orcmid** (guest, #74478) [Link] (1 responses)

Two observations around what may pinch for a work-around:

1. If the expiration of a record is connected with the execution of a critical section in the kernel, it can be of paramount importance that the terminating use of the record be accomplished as quickly as possible, running in a real-memory task. No page faults, no screwing around, do the real work and scram, as when servicing any kind of real-time interrupt.

The work-around in such a case is non-trivial and it may have footprint and other consequences.

2. These actions were brought in 2006. There have been at least 5 years to figure out how to avoid the essential claims of the patent for which infringement has been found. What's up with that? I would assume the kernel team (and Linus) know what they need to do already.

About Work-Around Difficulty

Posted Apr 22, 2011 3:22 UTC (Fri) by **orcmid** (guest, #74478) [Link]

The 2006 date is incorrect. The action was brought in 2009. I have no idea where I got 2006.

Google Linux servers hit with $5m patent infringement verdict (The Register)

Posted Apr 21, 2011 23:04 UTC (Thu) by **flewellyn** (subscriber, #5047) [Link] (9 responses)

If that's the novelty, it's not very novel. Opportunistic, incremental garbage collection is very old. So is hash indexing. So, the idea of opportunistically GCing "expired" hash-indexed records is pretty obvious.

If we had a more competent patent office, that would not have passed the "obviousness" test.

Opportunistic Garbage Collection Obviousness

Posted Apr 21, 2011 23:30 UTC (Thu) by **orcmid** (guest, #74478) [Link] (8 responses)

The claim is not about opportunistic garbage collection. It is about deleting no-longer-needed records from a linked list of same-hash records while visiting that list for some other purpose (finding, inserting, or deleting something).

It is a rather specific circumstance.

The first patent was applied for in 1989 and the last one (specific to the linked-list case) in 1997.

The patent applications list such texts as Knuth "Art of Computer Programming," vol.3 (1973). I only have the second edition (1998) and there is only limited mention of keeping expired ("deleted") items on their list and I could find nothing about incorporating their clean-up in other operations carried out via the hashed-index, apart from possible reuse for later insertions.

One thing that is not clear from my cursory inspection of the patent and Knuth has to do with whether or not the list threads through the records or the linked list is actually of pointers to the relevant records. This can matter in what is going on and when what portions are reclaimed.

Here's a garbage-collection type of situation where this kind of delayed deletion might matter. Consider a pure Lisp type memory of CONS cells. One can do any variety of garbage collection techniques with regard to how those cells depend on each other (including cyclically in the case of LISP), but you can only garbage-collect cells that are unreachable. If some cells are reachable from a hashed-table (say of symbolic names for some purpose), a cell is only removable when it is not reachable that way either. It usually takes some other means to actually release the connection from the hashed-table, and that may depend on it being known this is the only way something is left being reachable.

OK, I sort of hate that contrived example, but hope it is good enough to point out that the problem is different than opportunistic garbage collection as an abstracted-from-context known technique.

Defered Work Obviousness

Posted Apr 22, 2011 0:44 UTC (Fri) by **Pc5Y9sbv** (guest, #41328) [Link] (1 responses)

I think a number of us would think that the mere existence of incremental garbage collection strategies in the 80s and 90s, not to mention prior algorithmic work like heaps, ought to be prior art to kill such a patent. The

important point is being able to loosen representation invariants and defer some maintenance work; this method has been applied in many systems to either increase the size of work (bigger batches) or to allow incremental work units (less jitter). (And sometimes both occur based on adaptive thresholds, e.g. batch small requests and break up bulk requests.)

The fact that overly specialized corner cases of generally know technique can be enumerated and patented is part of the problem with the current patent system. It is obvious to one skilled in the art, and it amounts to a land rush to try to lay claim before everyone else gets there. It is not innovation that needs protecting. It is disgusting.

Defered Work Obviousness

Posted Apr 22, 2011 1:43 UTC (Fri) by **flewellyn** (subscriber, #5047) [Link]

Yeah, this is what I was going for. It's just a specific application of very well-known techniques.

Opportunistic Garbage Collection Obviousness

Posted Apr 22, 2011 1:28 UTC (Fri) by **jd** (guest, #26381) [Link] (5 responses)

Back wne I was at Uni, in the 80s, we looked at database theory. Standard technique for indexed sequential databases where you really don't want to fragment the file: when you "delete" a record, you don't remove it, you merely set a flag. When a new record is created, if a flagged record is of equal or slightly greater size you overwrite it.

Every so often, you vaccuum the database to eliminate gaps and remove any flagged records still left.

This gives you a situation where a record is not deleted immediately but is deleted when performing a different operation (an insert or a vaccuum).

This was considered a fairly basic, crude system that was a holdover from sequential media like magnetic tape.

This system would infringe the patent, despite the fact that I seriously doubt anyone has used magnetic tape for live database work within the lifetime of anyone involved in writing said patent or enforcing it in court.

The part that makes me concerned, though, is that this caswe is giving the impression to other companies that patent trolling is easy money and that actual software development is high-risk. That's an extremely dangerous mindset. When you start believing that producing is bad and privateering is good (privateering is piracy in the cutlass-and-walk-the-plank sense that has been legalized by a government), what do you imagine happens?

Opportunistic Garbage Collection Obviousness

Posted Apr 22, 2011 2:15 UTC (Fri) by **orcmid** (guest, #74478) [Link] (4 responses)

That is not an infringement of the patent unless your index uses hashed-key linked lists and, instead of an off-line cleanup, you did some bounded cleanup of only those deleted ones with the same hash when that particular list was being revisited for some useful purpose.

The essential claims in the patent are very specific about the situation in which they arise.

With regard to other observations about what is obvious or not, nothing we say here will impact the validity of the patent.

Google attempted to convince a jury that the patent was invalid and failed. I don't know what their argument was and whether they had any prior art to offer.

Google also failed to convince the jury that the patent was not being infringed, valid or not. They would have had to parade out the code and experts to explain it to accomplish that. Don't know how they failed at that either.

Now, we know that there appear to be some jurisdictions in Texas that are very friendly to patent-infringement suits. (This is where Microsoft ran up against i4i, for example.) But even so, it is not clear, based on the experience of others, that there is much room in the appeal process to over-rule the jury's findings of fact.

Whatever Google and other defendants end up paying, the only thing that will keep Linux as unencumbered FOSS is if the alleged infringement is cured by working around the essential claims of the patent so there is no infringement. If the patent were to be over-turned, put the code back if it is appropriate. Otherwise, just move on, I say.

I am far more concerned about Linux staying unfettered than I am about Google's pocketbook.

Opportunistic Garbage Collection Obviousness

Posted Apr 22, 2011 3:27 UTC (Fri) by **jd** (guest, #26381) [Link] (3 responses)

Indexed sequential: You hash then you sequentially search. Ok, so clearly we are indeed talking about items that hash to the same value.

Opportunistic overwriting: This is online, so we're clearly talking about bounded cleanup of items that hash to the same value.

Please, please read what you are replying to BEFORE replying to it.

Opportunistic Garbage Collection Obviousness

Posted Apr 22, 2011 5:48 UTC (Fri) by **orcmid** (guest, #74478) [Link] (2 responses)

That's not the definition of indexed sequential that I ever heard of (my experience going back to the original ISAM).

The "sequential" in indexed sequential means that the file can be accessed in record order (sequenced by key) or by direct access by key. In general, it matters that the keys are ordered. Typically, an index is used to map keys to the external storage blocks where the corresponding record is to be found. It is common for the index to be organized as some for of wide tree and how the index is cached matters.

The technique described in the patent is strictly about hashed indexing by keys and there is no information or assumption about being able to somehow navigate the hashed records in some key-based sequence nor is it presumed that such be possible.

I see that there is a variation of indexed-sequential in which the index is built using a hash technique (see http://en.wikipedia.org/wiki/Indexed_Sequential_Access_Me...). I have no idea what is hashed for that to work and how that helps with how the index itself is managed. But since you built one, perhaps you can enlighten us where hashing was done as part of working down to the location of a record have a particular exact key?

In any case, the Sequential in Indexed Sequential is not the sequential in the sense that the linked list of list items for records having the same hash code is considered sequential. Indeed, there is generally no requirement that the positions of those records in the list be sequenced in some way, although Knuth points out the advantage of doing so, since the detection that the desired record is not present can happen more efficiently.

There is also considerable importance, to ISAM, that the index for random access be loadable into memory in some manner or otherwise cached, since the records themselves were often on external storage and even insertions might be rather costly unless buckets of records retained slack for some reasonable number of insertions, with an overflow mechanism (not related to what we mean by hash overflow) for any excess.

And of course, in common usage, the only way to clean up an ISAM file was to reorganize the whole thing with some sort of utility, not unlike disk defragmenting.

Opportunistic Garbage Collection Obviousness

Posted Apr 22, 2011 6:10 UTC (Fri) by **orcmid** (guest, #74478) [Link] (1 responses)

I did go look at Berkeley DB because it does use a hashing scheme. What is revealed there is that hashing does not support indexed-sequential operation, but the B+Tree access method does. (This gets determined when you build the DB.)

The hashing method is extended linear hashing, where the hash function is adjusted dynamically to allow growing of the hash table. Linear hashing is mentioned on pp.548-549 (just before the exercises of section 6.4) of Don Knuth's "Art of Computer Programming," vol.3 Sorting and Searching, second edition. There is not enough information there to know how collisions on the same hash are dealt with.

Opportunistic Garbage Collection Obviousness

Posted Apr 22, 2011 18:32 UTC (Fri) by **Wol** (subscriber, #4433) [Link]

In linear hashing (typically used in Pick-based systems) it's normal to have a bucket that can hold multiple records. Records with the same hash are chained in a bucket, and should the bucket overflow, the records are often chained on the end of the file. So let's say your "modulo" is 20 - buckets 0-19 - the first bucket to overflow chains into 20, then next into 21, etc.

This then causes a bit of grief when the modulo increments, because you have to shove bucket 20 out of the way to create a new bucket 20 as non-overflow.

At least one implementation (Prime Information) used a special file/directory type, so each SEGSAM file contained subfiles, one per bucket, and they could grow as big as required.

Cheers,
Wol

Google Linux servers hit with $5m patent infringement verdict (The Register)

Posted Apr 22, 2011 9:35 UTC (Fri) by **oldtomas** (guest, #72579) [Link] (2 responses)

"The Nemes patents (all filed in the last century) all involve hash-indexed records becoming obsolete in some way but not being deleted at once for some reason [...]"

"As far as I can tell, that is the novelty that is not dealt with in any of the extensive, existing work on hash-based searching of records"

Still not *that* surprising. I haven't my Knuth around, but I wouldn't be surprised to see that covered in one of the exercises. Especially when not doing linked lists but secondary hashing: then it's really problematic to "disappear" something from the collission chain (aka hash bucket).

Heck, I'm sure I've seen reordering of the chains on search (on the theory that you are going to search some entries more often than others, so bringing those to the front amortizes itself). I even think I've seen that in Knuth's classic.

How far is that patent nonsense going to take us?

Google Linux servers hit with $5m patent infringement verdict (The Register)

Posted Apr 22, 2011 17:14 UTC (Fri) by **orcmid** (guest, #74478) [Link] (1 responses)

Knuth addresses the case of a delete operation by key not actually deleting the record on the linked-list but marking it deleted, with the record storage and list slot available for reclaiming at a later time, possibly by a new insertion having the same hash.

Knuth also raises (in my 2d edition copy) a case where there might be pointers to records from other than the hash-table index and some sort of reference counting might be needed to avoid deleting or reusing the actual record until the record actually becomes unreachable. This is the closest I have found that is a kind of "expiration." I don't know if that was in the 1973 edition cited in the patent itself. I obsserve that this is a situation that can arise in operating-system work, of course, and then race-condition avoidance, locking, and such come into the picture as well.

None of the other discussion of this kind of hash-table indexing, including in the exercises, touch any further on that nor the very specific claim of the patent.

The patent deals with removals where the expiration of a record is not by an effort to delete it by key but by virtue of some other condition that arises independent of use of the hash-table index. As I said, what the patent applies to is methods of removal that discover such expired records on the linked-list while it is being traversed on behalf of some other operation involving a record with the same hash code. The patent is quite specific about that, with some additional claims around details for limiting the number of such deletions done at one time.

An earlier comment mentioned a network application where some connections or network probes might time out. Another case might be indexes to the active records for running processes, or shared libraries and their stickiness, or something like that. I don't see this method being essential to any of those, but if it has been so used in Linux (or in some other way), an alternative is called for.

Places to make a hash of it

Posted Apr 22, 2011 18:11 UTC (Fri) by **orcmid** (guest, #74478) [Link]

As long as we're speculating on what sort of things might be handled with a hash-table mechanism that is covered by the patent we're concerned about, I should add the case of UUIDs. UIIDs (GUIDs to some of us) are quite heavily used these days and hashed-table lookups for them are not uncommon.

> Whether and how that might figure in Linux use of the patented method is not anything I have a clue about. And without knowing what the actual cases are, it is difficult to speculate further about what would be a suitable work-around.
>
> I presume that the Linux kernel developers are doing what they can to figure that out.

Google Linux servers hit with $5m patent infringement verdict (The Register)

Posted Apr 23, 2011 13:38 UTC (Sat) by **Ad** (guest, #74506) [Link] (1 responses)

Most of the cache associated network (DNS cache, HTTP server and may be browser) has expiry time associated with each entry. The key, value map in cache can also use hashing.

Also network programs using asynchronous IO typically has only one thread. Tasks (including GC) can be done only when processing events. No separate GC thread, no separate GC pass.

So if a cache implements a mechanism where, while inserting entry, iterates and also cleans up expired entries, does that count as violation of this patent?

Just curious.

Google Linux servers hit with $5m patent infringement verdict (The Register)

Posted Apr 23, 2011 18:01 UTC (Sat) by **orcmid** (guest, #74478) [Link]

I cannot respond to whether or not some speculative implementation would be an infringement.

Note, however, that the essential claims of '120 all involve the access to a linked list of same-hash records as part of the by-key record access mechanism and cleaning out at least some of the expired records on that list while doing so.

You should consult the text of U.S. Patent 5,893,120 to satisfy yourself one way or the other. It is found at http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&...

Also, you will need to look at the TIFF images for the code, flowcharts, and especially for the modifications made to the claims as part of a reexamination amendment that was issued on 2011-04-12. (The reexamination determination is on the last page. It modifies 4 of the 8 claims and adds 4 more.)

Linear Hashing Work-Around Potential

Posted Apr 23, 2011 19:39 UTC (Sat) by **orcmid** (guest, #74478) [Link] (15 responses)

As part of digging into materials on hashed-index techniques, I wondered if there was anything about Linear Hashing that would provide an alternative in which a non-infringing method of deleting automatically-expired records could be incorporated.

I did find such a place where the deletion of automatically-expired records could be handled opportunistically as part of a different operation that has a same-hash linked list be operated on, but not while searching for the presence or absence of a target record by its key (or the equivalent ways of speaking of "record search means" in the '120 patent).

THE DOWN SIDE

1. It appears that the novel feature of linear hashing (the way that the number of hash buckets is dynamically expanded and contracted at very low cost and in a smooth manner) as well as extended uses of linear hashing may already be subject to patent protection. I cannot conduct a serious search, but finding one example of a patent with regard to distributed linear hashing set off my early-warning detector. Linear Hashing developer Witold Litwin and colleagues, including Marie-Anne Niemat and Donovan Schneider may be involved.

2. The use of Linear Hashing is very appealing because of how it allows for the dynamic management of load-factor (the average number of list items that are examined on a keyed search) and the way the array of buckets can grow smoothly (without requiring dynamic memory-reallocations of array storage or anything heavy-duty like that, especially in a kernel). Although the 1970 Litwin paper is a great source, the later distillation of linear hashing as an in-memory technique has been superbly described by Per-Åke (Paul) Larson ["Dynamic Hash Tables", Communications of ACM 31, 4 (April 1988), 446-457].

3. The problem of switching to Linear Hashing, apart from the red flag that (1) raises, is that any simple code that accomplishes opportunistic deletions within the "record search means" has to be removed, the hash function is a little trickier (but very clever), and the way arrays of buckets are organized and managed is going to be completely replaced, with inclusion of the special procedures that continually adjust the number of buckets, the list for each hash value, and the number of hash-values delivered by the hashing function at any time.

MIXED BLESSING

4. An interesting facet of in-memory linear hashing is that it appears that the splitting and combination procedures by which buckets are added and removed can be handled concurrently with the "record search means" by which individual records are being searched for, inserted, accessed, or deleted by key. Keeping in mind consideration (1) of course.

5. It is trivial to add the deletion of automatically-expired records to the situations when a list is visited for the purpose of rehashing as part of the mechanism of (4), assuming that hasn't already been covered in the claims of a patent applicable to linear hashing.

6. It is not so trivial that the presence of automatically-expired records and their deletion (5) interferes with the management of load factor, and one can imagine pathological cases where there is chatter in the expansion-contraction of the hash-bucket array. Implementations require heuristics for avoiding that. [There are already pathological cases, this just adds to them and requires more-careful heuristics to avoid chatter without over-engineering the situation.]

BOTTOM LINE

Having dug around, I am now fascinated by Linear Hashing and its prospects for use in many situations, including the in-memory and the distributed database cases.

Yet I hesitate to use it myself considering that it is likely that there are any manner of patents that apply to it and its extensions and that it may be prohibitive to determine what such patents might be.

Linear Hashing Work-Around Potential

Posted Apr 24, 2011 8:25 UTC (Sun) by **Wol** (subscriber, #4433) [Link] (2 responses)

If you're worried about patents, you shouldn't be. At least, no more than any other technology. You've seen the mention of the papers from 1970 - that's forty years ago!

I first started working with a commercial linear-hashed product (Prime INFORMATION) in 1985 - that was rev5.3, and I don't know how long linear hashing had been in it. It's been very widely used since then - the

entire Pick/MultiValue ecosystem is based on it!

But if you want to know more about it, and its commercial use, investigate Pick. A good web-site is www.u2ug.org, that leads to the multi-value ring, and you'll find masses of good information. Maybe I'll see you on the u2ug mailing lists? ...

Cheers,
Wol

Linear Hashing Work-Around Potential

Posted Apr 24, 2011 18:08 UTC (Sun) by **orcmid** (guest, #74478) [Link] (1 responses)

It is true that we may be talking about patents that may have expired already. And even '120 will have expired no later than 2018 (probably sooner).

However, '120 does apply to linear hashing today if that linear hashing implementation deals with automatically-expiring records and it removes some or all of those on a list of same-hash records while carrying out an operation of the "record search means utiliizing a search key to access a linked list of records having the same hash address."

Considering how fruitful linear hashing has been seen to be, especially for distributed databases, I would still want to make sure I understood about applicable patents before distributing code using the technique (and especially if automatically-expired records are handled opportunistically as part of the split and condense process).

Linear Hashing is used in Berkeley DB in both the free and commercial versions. I actually have no need for a commercial one myself and would use an open-source one if I distributed an application that needed a DB.

I am surprised that Pick handles oveflow buckets the way you summarized it. I see no reason to ever have collisions with the expanding buckets of the linear hash directory (but I am looking only at the in-memory linked-list form at the moment).

Linear Hashing Work-Around Potential

Posted Apr 24, 2011 22:10 UTC (Sun) by **Wol** (subscriber, #4433) [Link]

Actually, Pick wouldn't run foul of the patent (aiui) because it doesn't store time-stamps with records, and so couldn't automatically expire records.

I want to re-implement a pick-style datastore, actually, and storing time stamps and expiring data was something I planned to do - I thought it was blatantly obvious! However, I would have done it as part of writing a record, because that's an obvious point at which to compact a bucket.

The original Pick model didn't include disk storage, it viewed it as a permanent virtual memory, and everything was based on hashed files, including RAM! This goes back to about 1967 - Pick is pretty much the same age as Unix :-)

Going forward to Prime INFORMATION, that had 2k buckets which can easily hold several records, depending on how big they are (stuff that was large than - by default - 1600 bytes got pushed to a secondary store). So, every time I add a record to a bucket, it's possible - even probable - that I need to compact the bucket to make room inside the 2k. (Deleting records, I would probably just flag them - not worth compacting them at that point.)

As for your surprise at the way of handling overflow buckets, you're thinking in terms of memory. And you badly misunderstand the hashing technique if you think expanding buckets will avoid collisions! :-) In a Prime Technical Paper I've got, they give the example of "number of buckets equals number of records", record id is sequential unique, BUT they add the records from 0 to 16 in random order. So at the end, with 17 records in seventeen buckets, there are no collisions. But if let's say the first two records to be added are both odd, 1 and 3 for example, they will both hash to bucket 1, and bucket 0 will be empty.

Linear hashing doesn't guarantee your file won't be lumpy. But the maths I've seen says that - absent a pathological situation - given a known key you need to access on average less than 1.05 buckets to find your record (or, more importantly, to know that it doesn't exist!) If every bucket access is a disk read, then that's damn fast! I don't know of any non-hashing technique that can come close. That's why, for pretty much ALL database sizes, a Pick system will leave pretty much any other database in the dust for speed.

My favourite war story is the company that ported their database from UniVerse (a Pick derivative) to Oracle. The Oracle consultants, after spending six months tweaking a particularly nasty SQL query, proudly announced to management that their new system was now ten percent faster than the system they were replacing. Unfortunately, they did it within earshot of the guy maintaining the old system, who scornfully responded "you're proud that your twin Xeon 800 is only 10% faster than my Pentium 90?" !!!

And that speed discrepancy is typical. I've seen loads of comments from people who can't believe how fast Pick systems are. We ran 32 users on a MIPS 3000 (equivalent to a 386) with 16Mb of RAM. System response was pretty good, although I don't think we hammered the system the way some people did. And watching the hard disk light, that machine was thrashing like mad ... !!! (Management didn't want to pay for a ram upgrade - until a ram chip failed and I wangled an upgrade with the support people :-)

Cheers,
Wol

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 24, 2011 18:40 UTC (Sun) by **orcmid** (guest, #74478) [Link] (11 responses)

In the note above, I mentioned "load-factor (the average number of list items that are examined on a keyed search)."

That's not what the load-factor is. The load factor is computed as the average length of the list in all current buckets, where a list is of length 0 for an empty bucket.

The load factor is an *estimate* of the maximum number of list items that are examined, on average, for a single keyed search under the assumption that searching is uniformly random by hash-value. (The estimated average number of list items consulted under that assumption is half the load factor.) That, of course, is the appeal of hash-table lookups. But as we must be reminded, when that is not the access pattern, the experienced average moves toward the worst case.

A pathological worst-case is easy to demonstrate. Suppose, through some bug or some peculiarity of the data being operated with, all keys have the hash code 0. That is, everything goes into bucket 0. As the load factor goes up, the linear hashing scheme will start splitting buckets and then rehashing their lists between the original bucket and its addition at the end of the used hash table. But bucket 0 still has everthing. So the actual search time gets longer and longer since it is a linear search of all of the records, the load factor keeps going down, and the hash-table expands for no useful purpose whatsoever. There are also cases that move the list from bucket to bucket while still having worst-case search behavior.

Although one would presumably not use hashing where this case had any likelihood at all, any kind of clumping of records and in the access pattern will cause performance deviations, that persist long enough to be important.

This is why it is valuable to test hash systems by throwing many distributions of data at them and carrying out other kinds of stressful simulations. It is also good to collect statistics in a running implementation so that degrading situations can be identified.

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 24, 2011 22:18 UTC (Sun) by **Wol** (subscriber, #4433) [Link] (10 responses)

Whoops - that is NOT the definition of load factor that I am used to!

Bearing in mind I'm describing a disk file, to me "load factor" is the average %age used of each primary bucket.

Let's say I've got 50K of data spread across 40 buckets of 2k each. That's a load factor of 50/80, or 60%. And then that's modified to exclude any oversize records that aren't stored in primary buckets :-) A typical Pick file will grow when load-factor hits 80%, and shrink when it goes below 50%. In practice, even wasting disk space equal to 100% of your data, it seems to be more compact than MS SQL Server, from my personal experience :-)

So "load factor" is actually quite a complex beast, depending on what you're doing :-)

Cheers,
Wol

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 25, 2011 18:01 UTC (Mon) by **orcmid** (guest, #74478) [Link] (9 responses)

Thanks for this and the previous expansion on the linear hashing implementations you're experienced with.

The load factor definition I use has to do with the average number of keys that have the same hash in the set of possible hashes (actually, same buckets).

I should clarify what I mean by same-hash. Typically, the adjustable bucket function that works with linear hashing is something like

bf(key,level) = hash(key) mod (N*2^level)

where N is the minimum (starting) number of buckets and level is initially 0.

Once splitting starts, there are at any time at most two of these functions being used.

Let M (not less than N-1) be the highest number of the buckets currently in use and let let N*2^level be the least integer greater than M.

Then the rule for finding the correct bucket is,

bucket(key,level) = if bf(key,level) > M then bf(key,level-1) else bf(key,level)

Note that for the pathological case, I had hash(key) be what was constant. It doesn't matter what constant, but 0 is easiest to visualize.

In the simplest linked-list in-memory case, a buck holds one record directly (in the head entry of the linked list) and anything else is linked from that. An empty bucket has some sort of special NULL entry so it can be

distinguished for a bucket holding onto one record. Note that if N is a power of 2, the bucket(key, level) algorithm becomes very simple since masking can be used instead of division but there needs to be more attention to the quality of hash(key). I prefer that case because masking can be used to dramatically speed up the entire process of getting to a specific bucket through whatever the bucket director structure is.

If we are talking about disk spill, and on-disk buckets, the overflow case is different and different load factor values are more important. For example, one would like to know the average number of disk blocks taken for buckets corresponding to the same hash, on the assumption that one wants to minimize the number of buckets with overflows to the point where the extra hit has a negligible impact on performance. I imagine for tuning purposes, the load factor as I described it also matters because it gives a clue as to how much empty space is carried in the first/primary disk block of a bucket, since having too many disk blocks that are nearly empty has its own performance consequences. It looks like these load factors (being estimates) are derivable from each other assuming primary and overflow blocks are all of the same size.

We've wandered a distance from workarounds for '120, which I think are more-easily handled without going to linear hashing if it is not already being used in the Linux kernel code that infringes an essential claim. Thanks for the exploration though.

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 25, 2011 19:48 UTC (Mon) by **Wol** (subscriber, #4433) [Link] (8 responses)

In UniVerse (the current implementation with which I am familiar, Prime having gone bust) you can adjust the size of the bucket. In INFORMATION it was stuck at 2K (which gave you a maximum key size of 1600 bytes because it needed to fit in one bucket).

In practice, strange as it may seem, I don't think records spilt out of the primary block that much - they couldn't have if my comment about needing only 1.05 reads average to find a record is correct ... :-)

At worst, even if all records were oversized, you'd need the 1.05 reads to locate it, and then one more read to actually get it because the primary bucket would tell you where it was.

btw, you said that "if N is a power of 2, masking can be used". I think you've got a bit confused, but you've also missed a trick. You can always use masking. First of all, I wouldn't use N at all in the hash function. It would only be used in split/merge. I had some trouble getting my head round your use of N, but it does appear to work :-)

However, what I'd do is

Given M buckets, calculate P such that 2^(P-1) <= M < 2^P

mask = 2^P -1

If hash mod mask < M then that's our bucket, else hash mod rightshift(mask) gives us our bucket.

Then

if loadfactor > 80% then split
if loadfactor < 50% AND M > N then merge

Cheers,
Wol

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 25, 2011 19:52 UTC (Mon) by **Wol** (subscriber, #4433) [Link] (2 responses)

Actually, that explains something I didn't understand before ... in both UniData and UniVerse, N is called MINIMUM.MODULUS. Apparently in UniData (with which I am unfamiliar), changing this triggers a complete file rebuild, and I never understood why.

Your version of linear hashing explains why ... :-)

Cheers,
Wol

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 28, 2011 17:48 UTC (Thu) by **orcmid** (guest, #74478) [Link] (1 responses)

If N, the MINIMUM.MODULUS is a power of 2, it is possible to change it without consequence. It means that level numbers might seem to change, but there is a way out of that. Instead of using N as the minimum, just set some Mmin as the minimum number of buckets. Let N be any power of 2 not greater than Mmin. It might as well be 1. Now you can adjust Mmin all you want, although making it greater than the current M forces splitting. Reducing it simply allows the splitting algorithm to shrink the database if that is what is wanted. (There are probably some games with LFmin and LFmax to coax things along.)

Now, if you decide to adjust Mmin for some practical reason, do so! In the creation of an initial hash-table, one wouldn't do it by splitting of arrivals but by creating Mmin empty buckets, and the current level being completed would be whatever it needs to be to allocate into those M = MMin buckets properly.

I think the bigger problem is when hash(key) needs to change because it doesn't provide enough good uniformly random bits for the level that has been reached or there are not enough bits in it to increase the level at all.

Changing hash(key) arbitrarily breaks the linearity scheme by which the hash-table grows and shrinks linearly at the end and splitting depends on rehashing of one bucket dividing things between the current bucket and the new bucket at the end.

However, you can safely change hash(key) to a new hash'(key) that still produces the same low order bits but provides more or better high-order bits, ones that aren't selected by the current biggest mask so there are no buckets that are selected by them in the current state of the system.

The above arrangements only if N hasn't been wired into other aspects of the implementation. It is important to make the design of the expandable hash-indexed directory to the buckets and any clustering of buckets be flexible enough to handle variation in N and, better, fluctuation in Mmin. It should be possible to adjust and tune that scheme independently. The scheme needs to be robust enough so that the base never has to be rebuilt by rehashing everything, since that is terrible for high-availability distributed cases where we are talking about records and buckets/overflows on disk.

(If there is a problem at the level of the current mask, one can adjust LFmax and LFmin to force the system to shrink to a lower level if the performance hit while waiting to get to the desired lower completed level is tolerable. The techniques for distributed operation of linear hashed bases might be relevant if it is necessary to do anything more drastic than that in order to institute an entirely different hash(key) in an alternate base that allows the original base to be switched off at an appropriate time.)

Changing Hash Function without wholesale rehashing/rebuilding

Posted Apr 29, 2011 1:17 UTC (Fri) by **orcmid** (guest, #74478) [Link]

I thought about the need to retire hash(key) in favor of an incompatible hash'(key) some more (in the shower naturally), and I realized that even if hash'(key) doesn't produce the same low order bits, it is easier to migrate than wholesale rebuilding.

Here is a brief sketch with lots of important details left out.

I assume that N=1 and we use Mmin, M, LFmin and LFmax as I have come to be using them in this series of comments. This is all for simplicity.

When it is necessary to migrate the current hash-index base to another hash-indexed base, do it this way.

1. Arrange, for the to-be-retired base, for a level to be exactly completed, either by shrinking or splitting. So M is at some 2^level - 1.

2. Set up the new (initially-empty) base, base'. If this is a linear hashing base too, then that's cool because it can be set to some Mmin' initial number of (empty) buckets and we will let it grow naturally. The technique we use will allow base' to grow gradually as the old base shrinks.

3. Stop inserting new records in the old base. The procedure will be such that new records not found in either base will be added to base' only.

4. Adjust the hashing bucket(key,level) algorithm in the old base so that whenever hash(key) > M, we don't use the bf(key, level-1), instead we go to base' and look for it there. If it is not there, we insert it there. Likewise, if we do use bf(key, level-1) and we don't find the record, we go to the new base and insert it there. (It shouldn't be there already.)

5. When we are ready to decrease M (probably fixing LFmin to some large value and keeping LFmax always larger than that and what the current LF is, to encourage that, e.g., LFmax > LFmin > LF), instead of merging the same-hash values in bucket [M] back to the split buddy of [M], we instead insert those records into base' and then reduce M by 1. We will never look for those records in the old base any longer.

6. Now, at some point we will want to stop searching first in the old base but start searching in the new base. If the record is not in the new base, we should search the old base for it (hashing with the old hash of course). If it is in the old base, we are done looking although it *might* be worth removing it from the old base and inserting it in the new base. I don't have any intuition on when that trade-off is important. The easy way is simply to end the query. If the query in the old base doesn't find the record though, then the old base sends the record back to the new base to be inserted.

7. The bottom line is that the new base never returns a record/query that comes to it from the old base, but when it is accepting queries directly, it will forward to the old base if it can't find the record in its base.

8. This continues until, at some point, the old base is compacted enough that it is simply time to move everything to the new base. And it is simple enough to have that be when M = Mmin = 0 and old-base bucket [0] is all that's left (the mask has gone to 0, in effect).

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 26, 2011 2:43 UTC (Tue) by **orcmid** (guest, #74478) [Link] (4 responses)

I used the formulation given in the two now-classical papers on Linear Hashing.

I was having the modulus always be a power of 2. Then the bit mask to compute it for a given bf(key,level) is 2^(level+Nbits)-1 where N = 2 ^ Nbits is the minimum number of buckets (which are then repeatedly doubled as new levels are completed). So the bucket algorithm can be optimized to use the two masks (one of which is double the other + 1) precomputed and it is easy to adjust them as level is expanded or contracted. If hash(key) is also cached in the linked-list entries for the same bucket, rehashing becomes trivial. (This could matter in-memory more than if disk accesses are involved, but there are the usual time-space trade-offs to ponder, including whatever the cost of computing hash(key) is.)

For the linked-list case in memory, I used a bucket size of one, meaning exactly one linked-list cell (the head) per bucket.

However, how buckets are organized into blocks/clusters of contiguous storage (or disk) and how overflows are kept near or in those blocks is also something interesting to explore, since that also matters when dealing with either disk blocks or virtual-memory allocations. Since the splitting algorithm runs sequentially through the buckets, I speculate that having clusters with location affinity can be important.

I haven't looked at the time-space trade-offs enough to have formulated a simulation and determined the sensitivity to different bucket directory and block/cluster organizations. Too many opportunities, not enough time ...

I think you and I are looking at the same sort of considerations, we are just cutting up the storage organization in different but more-or-less equivalent ways.

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 26, 2011 2:55 UTC (Tue) by **orcmid** (guest, #74478) [Link]

For some reason, an edit I tried to make didn't get into the just-immediate post:

Although caching hash(key) might be useful to avoid its recomputation when splitting a bucket, it is not needed at all when removing a bucket, since we know absolutely what bucket the removed-bucket's records go into. It is the bucket number (actually, M) masked by the smaller mask at the time of the merge-back and before M is decreased.

Whether or not hash(key) is cached, it is a nice value to keep as the record itself in a fast simulation looking for where splitting doesn't catch up fast enough and what the headroom needs to be on load factor, etc. OK, that's more than enough speculation from me on one day.

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 26, 2011 13:11 UTC (Tue) by **Wol** (subscriber, #4433) [Link] (2 responses)

Trying to get my head round what you're saying ... but I think you're using "minimum buckets" completely differently to my MINIMUM.MODULUS. It looks to me like you're using Nbits to keep level always at 0 or 1.

MINIMUM.MODULUS cannot change on the fly. As is shown by my comment about UniData :-) The recommended use for it is where a file can shrink, grow, or be cleared regularly, so its size is very variable, but it also has a "normal" size. By declaring a minimum size equal to the typical size, you avoid a lot of unnecessary split/merge activity. The other use is where you are loading a file with a lot of data - again by setting the minimum modulus to whatever it will be once the file is loaded, you're saving a load of activity while the file is built.

Mind you, for my first implementation, to calculate the bucket I used the following pseudo-code

```
unsigned int mask = -1 // yes I know :-)
bucket = hash;
while (bucket gt current-modulo)
mask = rightshift( mask)
bucket = and( bucket, mask)
repeat
```

Assuming "int" is a short, that means I've got a valid bucket with less than 16 iterations. And the only operations I'm using are and, shift, and compare. All cheap compared with multiply, divide, and mod, I think.

Even cheaper if rather than calculating mask each time, I cache the upper value :-) Unless you're on a Z80, it's quicker to get the lower mask by right shifting the upper, rather than going the other way by doubling and incrementing. (I gather the Z80, due to a bug in the silicon, didn't have a leftshift function. Then somebody tested the instruction that should be leftshift and discovered it set the new rightmost bit to 1. So what should have been leftshift became "left shift and increment", at first informally then Zilog actually documented it because everyone was using it :-)

Cheers,
Wol

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 27, 2011 18:59 UTC (Wed) by **orcmid** (guest, #74478) [Link]

No, I never change the value of N which you remark is also known as MINIMUM.MODULUS. And level grows as necessary to keep the load factor in bounds. Every time the load factor gets too large, a bucket is split. That is, a new bucket M+1 shows up and it is a split of its "mirror" bucket farther back in the preceding buckets. Every time a level is filled, splitting starts over from bucket 0 moving forward through all of them to the end of the just-completed level. Each completed level has double the number of buckets that the previous level did.

It is also possible for the number of used buckets to decrease when the load factor goes too small, and I allowed for that as well.

I did some interesting back-of-the-envelope analysis and there is an amusing situation. At the time a level is completed, the expected occupancy of bucket [0] is the load factor. But the expected occupancy of the last bucket to be split in completing the level is double the load factor at that point in time. That's because it is still double the size of the already-split buckets, under the uniform random assumption. (It is still a (level)-1 bucket and splitting other buckets has no effect on its share of the load factor until it too is split.) Splitting then has there be load-factor-expected content in the shrunk bucket and its split partner, which is the instantaneous expectancy for all of the buckets in the just completed level.

This saw-tooth effect may have to be taken into account when wanting to avoid bad congestion, especially if we are talking about the likelihood of some sort of disk or memory block overflow into another block at the 2*LFmax expectancy.

Of course, even for uniform randomness one needs to account for variance as well. I haven't looked at that and I have no idea how important this observation is under practical conditions.

PS: By masking I mean computing the equivalent of hash(key) mod (N*2**level) by simply extracting the low-order bits of hash(key) by a mask of level+log2(N) bits where N is a power of 2. It is trivial to keep a pair of current mask values, one for level, one for level-1 and implementing bucket(key,level) very quickly, adjusting them as level changes. And if we need the bucket number to be a multiple of a power of 2 (for pointer manipulation), we can handle that by adjusting the masks too. It's all about getting through the directory of buckets to the bucket slot as quickly as possible.

Linear Hashing Work-Around Potential - LoadFactor clarification

Posted Apr 27, 2011 19:18 UTC (Wed) by **orcmid** (guest, #74478) [Link]

Oh. NO, Nbits is just the number of bits in N, for N a power of 2, and which is constant for the hash table, regardless of level. (I may be off by one here). I think it is log2(N)+1.

Level moves up and down as needed. However, at any point, we only have bf(key, level) and bf(key, level-1) in use.

The initial conditions are level = 1, M = N-1, and buckets 0 to N-1 are ready and empty. So the LF is initially 0 but we never shrink below N-1 anyhow. With linear hashing, we always grow the buckets linearly at the end, so the first bucket to split is bucket [0] and it splits between itself and new bucket M+1. (Then M is then advanced by 1).

At the time of the first split, presumably LF > LFmax has happened and let us assume for simplicity that it is over just a hair and we can assume LF = LFmax for convenience.

Then bucket [0] is expected to have LF records in it and after the split it will have kept an expected LF/2 of them with new bucket [N] getting the other expected LF/2. By the time the level is complete (so there are 2N buckets), each of them will again have an expected LF records in them (which is how the saw-tooth fits in). Then we start splitting again from bucket [0], M = 2N-1, and we work on building out the next level, which will end up with a total of 4N buckets if it is completed. Rinse. Repeat.