Claims Of Corporate And Legal Misconduct

Addendum I

Walter Tuvell

August 28, 2011

Document History

Rev.	Date	Author	Remarks
1.0	August 28, 2011	Walter Tuvell	First draft

Final

Related Documents

Author	Date	Title
Walter Tuvell	August 18, 2011 (version 1.0)	*Claims Of Corporate And Legal Misconduct,* in two Parts: *Part I (Acts Of Fritz Knabe); Part II (Acts of Dan Feldman, HR, Legal)* — Referenced as *"original (two-Part) Complaint"*
Alan Beaulieu	2005 (1st ed.) 2009 (2nd ed.)	*Learning SQL,* O'Reilly

Table of Contents

28 Executive Summary — Addendum I

This document is Addendum I to my original two-Part Complaint (see Related Documents).[111] It documents additions and corrections to the original Complaint, and logically forms an extension of the original Complaint itself. This addendum "incorporates the original Complaint by reference", including its terminology, Related Documents, numbering scheme, etc.

Hence, hereinafter, the unqualified term "Complaint" includes the original two Parts, plus now this Addendum I, unless otherwise specified. Additional addenda may be published in future as circumstances warrant.

28.1 List Of Particulars

- On July 5, Dan "offered" to "help me with my communication style", by "vetting" my published materials for "inappropriate material". I allege this was a hoax, for the sole purpose of harassment/abusive-workplace/IIED, with the ultimate purpose (which was actually realized) of fabricating false "cause" for dismissal (namely, the whole "lazy" scandal).[112]
- After I duly/properly submitted my original Complaint, IBM officials refused to respond "promptly" to the submission (not even an acknowledgment of receipt for a full week) — and further delay was proposed, based on my STD status. These delays: (i) constituted hostile work environment, because they violated IBM Law promises of "prompt" action to employee problems; (ii) constituted hostile workplace, because of their basis in knowing the *reason* for my STD status; (iii) violated IBM Law, which guarantees prompt action irrespective of STD status; (iv) constituted illegal (ADA) disability discrimination (because of STD status).
- By means of its "BCG Program", IBM has established "IBM Law" as a contractual "condition of employment". By all the anti-"IBM Law" actions conducted to date (conducted by its agents, named throughout my Complaint), IBM is in breach of that contract, and even guilty of unconscionable contract.

29 Typos, Etc.

In this Section, we list various typographical errors (and/or other typographical "infelicities") discovered in the original Complaint since its publication. None are of substantive consequence; some are merely cosmetic.[113]

The abbreviations "top/mid/bot" mean "roughly top/middle/bottom of page"; and "fn." stands for "footnote".

111· The original (two-Part) Complaint document is stable (permanently "frozen") at its 1.0 version (dated August 18, 2011) — it will not be changed or updated. Instead, any "additions and/or corrections" to the original Complaint will be published as Addenda, of which the present document is the first installment. This "append-only" practice comports with legal custom (though legal "addenda" are called "amendments").

112· Though, I averted actual dismissal by vigorous prosecution of this Complaint, together with STD leave.

113· Of course, we limit our error-correction to only the body of the Complaint, not the Appendices.

- Part I, p. 6 mid: “summarization sentence” should read “ summarization sentences”.
- Part I, p. 9, fn. 14: “leading to Dan” should read “leading Dan”.
- Part II, p. 7, bot: strike “blacking-out” (that’s not quite synonymous with syncope).
- Part II, p. 10, fn. 63: “performancs” should read “performance”.
- Part II, p. 11, fn. 65: “[nzVtCapture.sh]),” should read “(nzVtCapture.sh, Appendix O).”.
- Part II, p. 14, bot: “Appendix R or S” should read “Appendix S or T”.
- Part II, p. 14, bot: “so I send her a copy” should read “so I sent her a copy”.

15

- Part II, p. 25, bot: “didn’t see Blktrace help out” should read “didn’t see how Blktrace would help out”.
- Part II, p. 18, bot: The phrase “that’s not what the thrust of my case was about at all” should be italicized.
- Part II, p. 19, top: In “states no reasons”, “reasons” should be italicized.
- Part II, p. 24, fn. 91: “way a manager” should be “why a manager”.
- Part II, p. 26, fn. 97: “Language” would read better as “Circumlocution”.
- Part II, p. 30 mid: “Appendix AD” should read “Appendix DD”.

30 “Lazy” In *Learning SQL*

On Saturday, August 20, I just happened to be leafing through a book of mine, *Learning SQL* (first edition, see Related Documents),[114] when I noticed it used the word “lazy” in substantively the *very same way* I did (Appendix Z), i.e.: “If you’re lazy, you can …”. Appendix GG.a. Nothing surprising about that, of course — such wording has never before raised eyebrows in the whole history of the world previous to Dan’s “lazy” scandal.

But then, out of due diligence, I wondered: Hmm, my edition is from 2005, that’s a “long time” ago in this industry, I wonder if the word “lazy” might have become “disturbing” in the technical community in the interim? So I looked up the second edition (2009) of the same book. The *identical* passage occurs in both editions. Appendix GG.b. If that “lazy” passage were so “bad”, you’d think somebody would have complained to the author/publisher, and the wording would have been changed/apologized — wouldn’t you?

Oh yes, did I mention that the IBM/Netezza NPS (Network Performance Server), which I/Dan/Fritz(/John, etc) all work on, is a “SQL server” (that is, communicates/interacts with the world using the SQL language)? So, the book *Learning SQL* is directly applicable to this Complaint.

Also, did I mention that *Learning SQL* is a well-known, widespread, standard/elementary introduction the SQL language (not some obscure academic tome that nobody knows about)?

114· I’d already read it previously, but it’s a life-long (“bookworm”) habit of mine to idly leaf through books previously read, just to sort-of “reawaken/solidify/question the knowledge by repetition/revisitation”.

And that the author, Alan Beaulieu, lives/works in the Greater Boston area — so his use of "lazy" isn't some kind of locality-based idiom, offensive in the Greater Boston area (where Netezza's Marlboro office is located), but not elsewhere.

see Add. II, p. 5, bot

As if there were ever any question.[115]

31 Blktrace Finale

A major focal point of Dan's attack on me throughout has been my work on the Blktrace work item I inherited from Sujatha. Part II, Section 13.

See Appendix U for the status of the published Blktrace wiki page as of June 27. After I submitted my original Complaint on August 18, I was finally able to spend some time "finishing" the Blktrace work.[116] The final product is, of course, excellent work, as all my work has always been. Appendix HH.[117,118]

This totally "puts the lie" to any "(faux) concerns" Dan pretended to have about my technical work.

32 Dan: Vetting

(The material in this Section was intended to appear in the original Complaint, Part II, but was omitted due to time deadline/pressure.[119])

On July 5, at Dan's "Three Behavior Issues" reconciliation meeting (Section 17), Dan "offered to help me with my communication issues", by "volunteering" to "vet" my written materials (principally emails) before I published them. Of course I didn't know it at the time, but it wasn't Dan alone who came up with this "vetting" idea — Diane Adams and a lawyer were certainly involved, by Dan's admission ("nor have I ever", Section 20.1).

115· It would be trivial to exhibit innumerable similar usages of "lazy", if necessary, by scanning through enough books/magazines, doing Google searches, etc. But that's not really necessary, is it? Can we just stop pretending now? "Lazy" (and "lazy"-like) phrases are universal, idiomatic figures-of-speech — not trenchant social commentary. If there were ever a "smoking gun" for abusive/hostile/IIED work-environment, the "lazy" scandal is obvously it. Put another way: The "Lazy" scandal is exactly the sort of thing that all reasonable people uniformly react to instinctively by thinking/saying, in shocked disbelief: "That's outrageous!" In fact, this reaction is precisely the informal test for the tort of IIED — also known as the "tort of outrage".

116· Of course, almost any project can potentially be extended *ad infinitum*. I'm talking here about "finishing Blktrace to the extent the project had been proposed to date, absent chasing down TBD's, possible future extensions, etc.; and anyway Dan ordered me to stop working on it (Appendix II)." Total effort for the delivered Blktrace work was ~10 days charged to IBM/Netezza (the rest was freely contributed by me, from my spare time, during evenings, weekends and STD leave).

117· Note that some of the longer tables in Appendix HH are truncated by the printing/reproduction process, but the missing material is only of technical interest . Enough of the wiki page is properly reproduced to convey the "substantive gist" of the work. (In any case, a complete wiki page is available, in "MHT" format, and of course the original exists in the Netezza "Confluence" wiki server.)

118· A simple comparison quickly reveals that my Blktrace work meets, and generally greatly exceeds, the standard set by all other work on the Netezza wiki. Such a comparison (as of August 25) is possible during discovery, by the history mechanism of the "Confluence" wiki technology used at Netezza.

119· I was in too much fear/rush at the time to hurry-up and file the original Complaint before I got (illicitly/illegally) fired.

At the time, I didn't take his suggestion of vetting "seriously" (in the true sense of "serious", as opposed to "Dan's fabricated faux-serious"), because after all I'd never had any problems along these lines in the past. Nevertheless, just to be sure I was "playing Dan's 'game', according to his rules", I did try (in all seriousness) to search my memory to make sure of my innocence. The only potential example I came up with was an interaction I'd had with Brian Maly, but Dan said that was not a problem. Appendix Y.

Later, of course, the illicit "vetting" thing became very "faux-serious" indeed, because it provided Dan the "cover" he wanted/needed for his whole "lazy" scandal. Appendix Z 07/11/2011 07:33 AM.

Due to the feverish speed with which Dan "rushed to judgment" (trying to fire me over the "lazy" scandal), and the fact that I'd been on leave for over 3 weeks (July 7–31), there was only one other example of the "vetting" tactic that can be exhibited. Appendix JJ.

I also alluded to "vetting" at Appendix KK 08/04/2011 03:54 PM.

Obviously, I allege the whole "vetting" scheme was a fraudulent hoax, not intended for its stated purpose of "helping me with my communication style", but for the sole purpose of harassment/IIED, with the ultimate goal of fabricating a false "cause" for dismissal — as was actually accomplished with the "lazy" scandal.

33 Dan's Public Embarrassment

Since Dan's demotion of me on June 10 (but never before that date), Dan has made a lot of noise about "oversight of my work", "providing me guidance", etc. E.g., Section 13; Appendix R; Appendix II.

That is content-free blather (connived with HR, according to Dan's "nor have I ever", Section 20.1) — concocted as a transparent smokescreen for continued harassment, abusive workplace, seeking-false-reasons-for-dismissal, IIED, etc. Plain proof has already been given in this Complaint, especially involving two where I've solved problems Dan himself had tried but failed at.[120] But in case there is any doubt left, one final example will be presented now.

instances

Appendix KK. As a result of Dan's demotion of me, Sujatha had inherited my work on Wahoo, and had run into some issues that seemed to indicate "trunk performance regression" (i.e., that the NPS code base was becoming infiltrated with code that exhibited unacceptable performance degradation). Nearly two months after taking over my Wahoo work, Sujatha approached Dan about this issue.[121] Dan double-checked ("reviewed her work-product") and confirmed her conclusion.

So Dan took the extraordinary measure of "going loudly public" with his/Sujatha's finding, in an impressively large high-level email conversation — no doubt in a prideful attempt to enhance his/Sujatha's value in the eyes of the company, and not incidentally to drive home to everyone how great a job was being done in Wahoo-land now that he'd removed me from it.

But that scheme backfired. As the email discussion developed, it turned out that the problem was due, not to NPS regression at all, but to Sujatha's running her tests incorrectly. The

120· PerfScore: Section 2.4, footnote #23, Appendix R 06/17/2011 09:27 AM. nzVtCapture.sh: Section 11, footnote #65, Appendix O.
121· Sujatha shouldn't be criticized here — she was no doubt following Dan's orders on how to proceed.

whole "crisis" was a tempest-in-a-teapot. By unwarrantedly "crying wolf", Dan had given himself a big black eye in public.

The irony is that it could have been easily avoided. The right thing for Dan/Sujatha to have done was to ask me to take a look (I would have spotted Sujatha's mistake immediately). Dan's desire to "rub my nose in it" blew up in his face.

So much for Dan's "oversight and guidance" prowess.

34 Proof Of Employment

On Saturday, August 20, I received an email from HR, informing me that my "move"/relocation from the Cambridge office to the Marlboro office had been formally completed. Appendix LL. I had already been reporting to Marlboro since June 13 (Section 6.1), but this formality (employee database updating) had been long-delayed by Dan's inaction.

What was most interesting about that email was not the information about the move itself, but that it confirmed my employment status (as of August 19). This is interesting for the purposes/"standing" of my Complaint, because it shows me as an "employee-in-good-standing" (i.e., I hadn't been fired yet, so my Complaint was properly filed).[122]

35 Filing Of Original Complaint

On Thursday, August 18, I formally filed my original Complaint to the IBM Corporate Open Door process, as well as the Confidentially Speaking process, by email (pursuant to C&A Sections 2.5 and 4.3, BCG p. 8, etc.). Appendix MM.

36 Russell Mandel: Justice Delayed[123]

For a full week following the submission of my original Complaint, I received no response whatsoever from anyone — not even an acknowledgment of receipt-of-filing. Yet, IBM had been fully aware that my submission of the Complaint was immanent (Appendix Q: 08/02/2011 03:07 PM, 08/02/2011 04:18 PM), and was even aware of the urgency of prompt action (Appendix AA.b, especially 8/04/2011 07:13 PM).

This delay was: (i) contrary to the provisions of "IBM Law" (BCG + AYJ + C&A), which consistently guarantees "prompt"[124] ("~couple of days") response to employee problems. Hence it constituted another act of hostile-workplace.

Therefore I followed-up by email on August 25. Appendix NN.

122· Admittedly, IBM hasn't made a "lack-of-standing" argument to date; but it doesn't hurt to plan ahead. As of the date of this Addendum I, my IBM/Netezza network credentials still work (though I've noticed some irregularities, e.g., some service disruptions when I try CC'ing some emails to my personal [non-IBM] email account).

123· "Justice delayed is justice denied." — Traditional legal maxim. Attribution in this exact form uncertain, but probably originating in the *Magna Carta*, 1215–97 (various versions), clause 40 (underline emphasis added): "We will not sell, nor will we deny or delay, right or justice" (*Nulli vendemus, nulli negabimus aut differemus rectum aut justiciam*).

124· BCG, pp. 7, 8: "IBM will promptly review your report …"

Finally, Russell Mandel responded later that day (August 25), after his "returning from vacation".[125] In his response, Russell stated he wouldn't "discuss [my] concerns directly with [me]" until I returned from STD leave.

This further attempt at delay/postponement of investigation/justice[126] constituted: (ii) continued/ongoing illicit hostile-workplace, because he knew[127] the very *reason* I was on STD leave was due completely to the very abuse/IIED inflicted upon me by this case.

Furthermore, Russell's attempted delay was: (iii) contrary to the stated terms of IBM Law, which everywhere emphasizes prompt dealings with employee concerns — explicitly including employees "on leave".[128] In other words, by his misinterpretation of the "on leave" clause, this was now the *second time*[129] Russell exposed himself as *corrupt and/or incompetent* — because the "on leave" clause was a term of IBM Law that he, as "C&A SME", *knew or should have known*.

Finally, Russell's attempted delay constituted: (iv) discrimination on the basis of (mental health) disability. Hence it was contrary to the Americans with Disabilities Act (ADA) — thus illegal.

37 IBM Law: Contractual Nature; Non-Optionality

It is extremely important to point out that "IBM Law" constitutes a *binding, enforceable contract* — bilaterally (that is, it binds not only the employee, but also IBM).[130] That's because IBM Law (the cornerstone of which is the BCG, but the BCG "incorporates by reference" many other documents, see BCG p. 31) constitutes a *condition of employment* at IBM. Specifically, *all* IBM employees in general (and I myself in particular; Section 20.2) are *required (non-optionally)* to review and formally *certify* ("pledge allegiance") to the BCG — yearly. This is called the "BCG Program". Appendix OO.

see BCG, p. 6, on Compliance

In this (contractual) light, it is very significant to note that my filing-of-[cC]omplaint (both small-"c"-complaint and big-"C"-Complaint) was *non-optional*. I was actually *required (non-optionally)* to file a report — and likewise IBM is *required (non-optionally)* to investigate my report, and to *affirmatively (non-optionally) "not tolerate"* retaliation — by the very strict, unambiguous and "non-optional" wording[131] of the BCG itself (p. 8, emphases added):

add space char

- "If you know of, or have good reason to suspect, an unlawful or unethical situation or believe you are a victim of prohibited workplace conduct, *immediately report* the matter through any of IBM's Communications Channels …"

See also Add. III, App. RR, p. 10, for Corporate Trust and Compliance restatement of this.

125· I don't know "on whose behalf" Russell responded: I'd submitted my Complaint to the Corporate Open Door and Confidentially Speaking processes, not to his C&A process.
126· For the purposes of C&A investigation, "vacation-time" is not a qualifying condition. It's not mentioned as such in IBM Law, and it's certainly not the sort of thing any competent professional operation (as IBM's C&A program claims to be) would fail to plan back-up for.
127· I had explained my STD situation in the Complaint, Section 26.
128· The "IBM Law" documents I have available to me don't seem to specify whether or not the term "leave" applies to "STD". However, that usage is "usual and customary"; therefore I am justified in adopting that usage in the context of IBM Law (by the principle of *contra proferentem*).
129· The first time was documented in Section 22.
130· And the principle of *contra proferentem* surely applies to the BCG in fullest force.
131· The wording is so clear, *contra proferentem* need not even be invoked here.

see Add. III, p. 7, top and Add. IV, p. 5, bot; IBM used terminology "medical leave" to cover my situation

- "… these programs allow you to submit your concerns *online*, by email, regular mail, fax or phone."
- "IBM *will promptly review* your report of unlawful or unethical conduct, and *will not tolerate* threats or acts of *retaliation* against you *for making that report*."

My Complaint is replete with alleged (and proven) violations of various terms of the "BCG Contract" (especially, but not limited to, clauses involving "unlawful or unethical conduct", "prohibited workplace conduct" and "threats or acts of retaliation … for making that report"), by various named, officially appointed agents/employees of IBM. Hence, all such alleged violations constitute allegations against IBM of *fraudulent breach of contract* as well (because, IBM agents knew IBM would not enforce certain clauses — the "will-not-tolerate-retaliation" clause in particular).

Let's make this even clearer. The "non-optional-report-filing" clause of IBM Law has the following consequence. If any authorized IBM agent were to simultaneously (i) be aware of the contractual nature (i.e., BCG certification) of IBM Law (including its non-optional clause requiring me to file the report I filed), yet (ii) was actively engaged in retaliation (against said filing of report), then IBM would stand vulnerable to accusation of knowingly perpetrating a fraudulent *unconscionable contract* upon me (and therefore upon all employees). Diane Adams is clearly one such a person (as we know from Dan's "nor have I ever" admission, Section 20.1), and it is equally clear there are many other such people in this saga in various capacities (for example, Russell Mandel, by way of his self-described "SME" expertise, yet "STD/leave-delaying" tactics). And I do hereby make such an accusation (of unconscionable contract) against IBM.

APPENDICES — Addendum I

GG *Learning SQL*, "Lazy"

▶*Example colloquial use of the phrase "… if you're lazy, you can …", in a technical context — without irreparable damage to the reading audience.*◀

GG.a First Edition (2005)

```
| 2004-12-20 |                   0 |
| 2004-12-21 |                   0 |
| 2004-12-22 |                   0 |
| 2004-12-23 |                   0 |
| 2004-12-24 |                   0 |
| 2004-12-25 |                   0 |
| 2004-12-26 |                   0 |
| 2004-12-27 |                   0 |
| 2004-12-28 |                   1 |
| 2004-12-29 |                   0 |
| 2004-12-30 |                   0 |
| 2004-12-31 |                   0 |
+------------+---------------------+
366 rows in set (0.03 sec)
```

This is one of the more interesting queries thus far in the book, in that it includes cross joins, outer joins, a date function, grouping, set operations (`union all`), and an aggregate function (`count( )`). It is also not the most elegant solution to the given problem, but it should serve as an example of how, with a little creativity and a firm grasp of the language, you can make even a seldom-used feature like cross joins a potent tool in your SQL toolkit.

Natural Joins

If you are lazy (and aren't we all), you can choose a join type that allows you to name the tables to be joined but lets the database server determine what the join conditions need to be. Known as the *natural join*, this join type relies on identical column names across multiple tables to infer the proper join conditions. For example, the `account` table includes a column named `cust_id`, which is the foreign key to the `customer` table, whose primary key is also named `cust_id`. Thus, you can write a query that uses a `natural join` to join the two tables:

```
mysql> SELECT a.account_id, a.cust_id, c.cust_type_cd, c.fed_id
    -> FROM account a NATURAL JOIN customer c;
+------------+---------+--------------+-------------+
| account_id | cust_id | cust_type_cd | fed_id      |
+------------+---------+--------------+-------------+
|          1 |       1 | I            | 111-11-1111 |
|          2 |       1 | I            | 111-11-1111 |
|          3 |       1 | I            | 111-11-1111 |
|          4 |       2 | I            | 222-22-2222 |
|          5 |       2 | I            | 222-22-2222 |
|          6 |       3 | I            | 333-33-3333 |
|          7 |       3 | I            | 333-33-3333 |
|          8 |       4 | I            | 444-44-4444 |
|          9 |       4 | I            | 444-44-4444 |
|         10 |       4 | I            | 444-44-4444 |
|         11 |       5 | I            | 555-55-5555 |
|         12 |       6 | I            | 666-66-6666 |
|         13 |       6 | I            | 666-66-6666 |
```

192 | Chapter 10: Joins Revisited

GG.b Second Edition (2009)

```
| 2008-01-02 |               0 |
| 2008-01-03 |               0 |
| 2008-01-04 |               0 |
| 2008-01-05 |              21 |
| 2008-01-06 |               0 |
| 2008-01-07 |               0 |
| 2008-01-08 |               0 |
| 2008-01-09 |               0 |
| 2008-01-10 |               0 |
| 2008-01-11 |               0 |
| 2008-01-12 |               0 |
| 2008-01-13 |               0 |
| 2008-01-14 |               0 |
| 2008-01-15 |               0 |
...
| 2008-12-31 |               0 |
+------------+-----------------+
366 rows in set (0.03 sec)
```

This is one of the more interesting queries thus far in the book, in that it includes cross joins, outer joins, a date function, grouping, set operations (`union all`), and an aggregate function (`count()`). It is also not the most elegant solution to the given problem, but it should serve as an example of how, with a little creativity and a firm grasp on the language, you can make even a seldom-used feature like cross joins a potent tool in your SQL toolkit.

Natural Joins

If you are lazy (and aren't we all), you can choose a join type that allows you to name the tables to be joined but lets the database server determine what the join conditions need to be. Known as the *natural join*, this join type relies on identical column names across multiple tables to infer the proper join conditions. For example, the `account` table includes a column named `cust_id`, which is the foreign key to the `customer` table, whose primary key is also named `cust_id`. Thus, you can write a query that uses `natural join` to join the two tables:

```
mysql> SELECT a.account_id, a.cust_id, c.cust_type_cd, c.fed_id
    -> FROM account a NATURAL JOIN customer c;
+------------+---------+--------------+-------------+
| account_id | cust_id | cust_type_cd | fed_id      |
+------------+---------+--------------+-------------+
|          1 |       1 | I            | 111-11-1111 |
|          2 |       1 | I            | 111-11-1111 |
|          3 |       1 | I            | 111-11-1111 |
|          4 |       2 | I            | 222-22-2222 |
|          5 |       2 | I            | 222-22-2222 |
|          6 |       3 | I            | 333-33-3333 |
|          7 |       3 | I            | 333-33-3333 |
|          8 |       4 | I            | 444-44-4444 |
|          9 |       4 | I            | 444-44-4444 |
|         10 |       4 | I            | 444-44-4444 |
```

HH Final Blktrace Wiki Page

NETEZZA Blktrace, Blkparse, Btt

Added by Walter Tuvell, last edited by Walter Tuvell on Aug 25, 2011

Executive Summary and Conclusions

This study concerns the so-called "Blktrace Suite" of tools (often referred to simply as "Blktrace"), which are used for examining low-level Linux storage block-I/O (i.e., reads/writes to block storage devices, such as rotating electro-mechanical hard-disk drives (HDD) and to non-rotating solid-state drives (SSD), typically in support of filesystem operations). The tools in the Blktrace suite are:

- Blktrace (blktrace program) – Capture info about NPS storage operations at the Linux block-I/O layer, and log to files (in binary format).
- Blkparse (blkparse program) – Parse the Blktrace output logs (above) into binary/ASCII format, for further analysis.
- Btt (btt program, "blktrace tool [or timeline]") – Produce human-readable reports, by analyzing the blkparse output files (above).
- Other – Such as the blkrawverify program, and documentation.

The goals of our investigation are: i) to create a "push-button (turnkey) framework" for running these tools; ii) to determine the extent to which these tools could be helpful for us in improving the performance of NPS.

Concerning goal i) : That's what most of this wiki page is all about (following this section).

Concerning goal ii): Our basic conclusion is that the blktrace suite, despite being a very nice tool, is not exceedingly useful for us, at least not for the way NPS engineering is currently done.

Namely, the place these tools are really very useful is for designers of Linux block-I/O schedulers (a.k.a. disk schedulers). But Netezza does not design its own block-I/O scheduler. So Netezza would have to get into the block-I/O scheduling business for the blktrace suite to be really useful. This could be the case if, for example, Netezza were to redesign NPS to sit directly on top of the block-I/O layer, instead of the higher filesystem layer, or a "near-filesystem" layer as NPS actually does (some database systems do control I/O all the way down to the block level, but NPS/PostgreSQL does not).

It is to be noted that NPS does support its own I/O scheduler, but it's at the higher layer of data-I/O (reads/writes to database tables), as opposed to the lower block-I/O layer. Thus the NPS data-I/O scheduler already finishes its work before its reads/writes ever get to the Linux block-I/O scheduler.

The standard block-I/O schedulers currently supported by Linux are: NOOP, AS (Anticipatory Scheduler), DEADLINE, CFQ (Complete Fair Queueing); FIFO is also available, but that's non-standard, i.e., not pre-configured in typical Linux distributions. When the TwinFin system was first designed/implemented, performance tests were run for these block-I/O schedulers, and it was determined that the **DEADLINE** scheduler was "by far" the best for our purposes, for both throughput and latency (source: Andy Galasso). This conclusion is consistent with the "received wisdom", that the DEADLINE scheduler is best for database servers (as opposed to, say, CFQ being best for general-purpose workstations). So NPS currently uses the Linux DEADLINE block-I/O scheduler (at least for disk-based systems; presumably SSD-based systems such as Wahoo will use NOOP, which is considered better for that technology). However, it is not a goal of this study to go further into a discussion of block-I/O scheduling, so we have no more to say about that subject here.

<<TBD – Repeat Andy's experiments, to verify that DEADLINE is still best with recent Linux releases, esp. compared to CFQ. See video at http://mirror.linux.org.au/pub/linux.conf.au/2007/video/talks/123.ogg – there have been some improvements to CFQ for "server-type" workloads.>>

Disclaimer

Notwithstanding our overall conclusion that the Blktrace Suite doesn't seem to be overly useful as a prospective performance tool for us, we immediately note it does indeed come in handy from time to time among Netezza engineers for debugging/diagnosing certain low-level issues at the Linux block-I/O layer when they occur. Examples include:

- Diagnosing write ordering issues in MD raid, for the minor compare tool.
- Diagnosing write performance of xfs vs nzmpfs.
- Diagnosing device-mapper multipath performance.

Along these lines, an idea that has been kicked around is to implement an NPS virtual table (or more than one), similar to _vt_disk_log, that would capture the blktrace output and make it accessible within NPS/SQL. If/when that gets done, it would provide a more convenient way to use the Blktrace Suite than that described here, assuming that corresponding SQL scripts were developed to replace/augment the reports available with blkparse/btt.

Attachments

The attachments to this wiki page contain everything you need to get you started using the blktrace suite:

- blktrace-8e8bb83.tar.gz
- blktrace.pdf
- btt.pdf
- perfbin.tgz, containing: nzBladeIpAddrs.sh, nzBlk*.sh, nzBtt*.sh, CreateTblBlkparse.sql, readWriteStream.py
- 2 example tarballs: 30 minutes for the single-stream test Atomics+TPC-DS@100b, and 30 minutes for the multi-stream test Muqry@100b. These are "simple" examples, enough to get you started. They are "pre-processed" tarballs, that is, just Blktrace binary output, before Blkparse/Btt (or the attached scripts) have been applied to them. They are each approx. 0.5 GB in size, hence too big to place on this wiki, so you have to pick up copies via NFS:

```
sudo mkdir -p /mount/snapsa
sudo mount -t nfs perfsnap1:/SHARE1 /mount/snapsa  # perfsnap1 = 172.16.102.14
ls -l /mount/snapsa/walt/Blktrace
# Look at: blktrace=TF6:6.2.0i=Atomics,TPC-DS@100b=2011-08-22.tgz
#     and: blktrace=TF6:6.2.0i=Muqry@100b=2011-08-22.tgz
```

- Another example ("pre-processed") tarball, named similarly to those above, but with the string "Muqry@1500g" embedded. This example is is more realistic than the simple example above: multi-stream, on a larger 15TB DB. It represents ~860 sec. (~14.5 minutes) of Muqry@1500g, starting 20 minutes into its run ("warm-up time"). We use this example for in-depth "extended example" study, below.
- Numerous images, stored as attachments

WARNING: It's possible to break the blktrace tools if your dataset is too big. For example, I tried blktrace'ing Muqry@1500b for 30 minutes, but that broke blktrace (with a "blkrawverify failed" error). Even if you can get blktrace to succeed, very large blktrace captures can break other tools, such as spreadsheets/R (giving invalid graphics, for example). I even broke "tar" once in the Muqry@1500g example, but that didn't seem related to the mere size of the dataset (tar failed with a very strange "file changed while scanning" error, even though the file was write-protected at the time).

Introduction To The Blktrace Suite

The Blktrace suite is an open-source tool, written in C, and distributed in source-code form. Version 8e8bb83 was used for this study (it is attached to this wiki page, as is everything else discussed here).

I compiled the Blktrace suite (both software and documentation) on my development workstation (which is a laptop running Fedora 14), and also on an NPS host (TwinFin 6, 6.0 vintage). The suite was run on my laptop, on the NPS host, and on the NPS blades (the version compiled on the host also ran on the NPS blades, without separate compilation).

There were no major problems involved in compiling/running the Blktrace tools, just minor problems: presence and location of async-I/O artifacts, libaio.h and libaio.so, which the experienced developer will have no problem overcoming. For example, libaio.h didn't exist on the NPS host, so I copied it from my laptop to /usr/include on the host; and the libaio.so library does exist on NPS systems, but with the wrong name, which can be fixed by supplying an appropriate symbolic link (ln -s libaio.so.1 libaio.so, in both /usr/lib and /usr/lib64). I run the Blktrace suite from its own directory, rather than installing it on the host; see below for why (namely, it only needs to run on the blades, not the host, and that can be done via the Blktrace directory and the /magic NPS mounts). Also, I had to install LaTeX for the docs. [Some earlier attempts to get the Blktrace suite working on NPS ran into more serious difficulties, but those seem to have been overcome now. In particular, the blade's Linux kernel needs to be configured to support Blktrace, and that wasn't the case previously, but it was the case for me because Blktrace kernel support is now a standard part of the major Linux distributions.]

The docs, blktrace.pdf (for Blktrace and Blkparse) and btt.pdf (for Btt), attached here to, are the definitive sources of information about the Blktrace suite. Some additional information is also available on the web, but there's not much that isn't already covered in the official docs. We leave it to the reader to consult these, rather than discuss the Blktrace suite in detail here.

Scriptification Of The Blktrace Suite

In order to "program the Blktrace Suite" at a slightly higher level than the raw blktrace/blkparse/btt programs themselves (in particular, to supply preferred options/arguments), I wrapped the Blktrace/Blkparse/Btt tools in some helper shell-scripts (they are attached, see perfbin.tgz). The ones involving the blktrace program run on the NPS host (blades only, no need to run it on host), while the ones involving

the blkparse/btt programs ran on my laptop (using the logfiles produced by blktrace, transferred to my laptop).

The shell-scripts I wrote were more-or-less simple/quick-and-dirty one-off's, not intended to be general purpose or "supported" scripts (so don't expect great beauty here). Nevertheless, the experienced developer should have little trouble modifying them for their own environment, should they desire to use them (in particular, replacing the hard-coded directory name "/nz/wall"). One gotcha I learned, for example, is that NPS blades only support Bourne shell (/bin/sh), not Bash (/bin/bash), so I had to write shell code that ran on the blades in that ancient shell dialect (for example, back-ticks instead of "$()", "1>/dev/null 2>&1" instead of "&>/dev/null", etc).

In order to use the blktrace scripts in the distributed NPS environment, I used SSH of course, so I had to set up password-free ssh to the blades (see System Setup & Maintenance, which also show how to set up password-free sudo, which I needed to do too). The scripts themselves took care of everything else (such as setting up access to the Blktrace tools, as well as the script tools themselves, via the NPS "/magic" NFS mounts: symlinks, /magic -> /nz/export/tools on host, (/magic -> /var/opt/nz/host/tools on blades).

Probably the biggest problem I had writing the scripts was NFS propagation (between the NPS "/magic" directories). For that reason, the scripts contain some sleep's, sync's, and repeated tries that would otherwise be unnecessary.

Running blktrace (On NPS)

Here are the steps to generate the blktrace logfiles ("blktrace capture" phase):

Step 1: Compile Blktrace Suite Programs (On NPS Host And On Workstation)

On the host, I did this in a directory called /nz/wall/blktrace-8e8bb83. You should also put PATH=/nz/wall/blktrace-8e8bb83:$PATH.

Step 2: Install Shell-Scripts On Host

I did this in a directory called /nz/wall/perlbin. You should also put PATH=/nz/wall/perlbin:$PATH.

Step 3: Run nzBlkTracesStart.sh On Host

The nzBlkTracesStart.sh script copies blktrace-8e8bb83 and perlbin to /magic, and it also creates directories named blktrace-A.B.C.D in /magic, where each blade will deposit its Blktrace logs. Here A.B.C.D denotes the IP address of each blade (I prefer this to blade names, but either could be used, via "nzhw show -detail" -- see nzBladeIpAddrs.sh). For example, on one TwinFin 6 I used, the blade IP addrs were: 10.0.4.151, 10.0.5.127, 10.0.8.198, 10.0.11.2, 10.0.11.88, 10.0.11.204.

The "plural" script nzBlkTracesStart.sh then invokes the "singular" subscripts nzBlkTraceStart.sh on each blade. Those subscripts discover all the "candidate" block devices on the blade, check/mount the debugfs pseudo-filesystem (required for running the blktrace program), and start the blktrace program itself, monitoring all the candidate devices. As will be seen, the actual NPS data block devices are only a small subset of the totality of candidate block devices on an NPS blade, so Blktrace ends up monitoring too much. The extraneous logfiles collected by Blktrace are discovered and ignored by examination of the logfiles by the scripts at a later post-processing stage (I didn't know of an easier way to narrow down the actual devices of interest).

Step 4: Run Your Workload

Once nzBlkTracesStart.sh returns, you can start the workload you want to monitor (or alternatively, you can start/stop Blktracing multiple times while your workload it running, see examples below). For example, I initially did this twice, both times using the PerfBar runBar.py program, once running Atomic+TPC-DS and once running Mucpy (with 1 loop), both on the TPC-DS 100b database. Each of these took approx 0.5 hr to run, which was judged to be sufficient for the initial purposes of this study (since the study on its face isn't very useful for us anyway, as discussed above).

Incidentally, I found that the performance of NPS under Blktracing is approx. 10% slower than a non-blktraced system, but that's of no concern for this kind of study.

Step 5: Run nzBlkTracesStop.sh

When the workload you want to monitor finishes, you stop Blktracing by running the "plural" script nzBlkTracesStop.sh. This calls "killall -TERM blktrace" on the blades (which is the recommended procedure for stopping blktrace). Note that nzBlkTracesStop.sh is written somewhat carefully, because it must allow for blktrace shutdown (flushing its buffers), and for NFS propagation delay (as discussed above).

When nzBlkTracesStop.sh returns, the Blktrace logfiles are found in the /magic/blktraceBlade=A.B.C.D directories. There can be a lot of them (almost 5,000 in my case).

Examples Of The Preceding 3 Steps ("Blktrace Capture" Phase)

Here's one typical example of how to invoke the preceding 3 steps (#3-5) just described (assuming you've set up your environment as recommended):

```
cd /nz/PerfBar5.1/scripts  # in preparation for running PerfBar
nzBlkTracesStart.sh  # start blktrace capture
./runBar.py --test=atomics,tpcds --tpcds=db=tpcds100b --force --debug -r n  # run PerfBar
nzBlkTracesStop.sh  # stop blktrace capture
```

Here's another example, with a little more flexibility thrown in for variety:

```
# In one ssh/terminal session to NPS:
cd /nz/PerfBar5.1/scripts  # in preparation for running PerfBar
./runBar.py --test=misery --tpcds=db=tpcds10000b --force --debug -r n &>runBar.out  # kill this (ctrl-

# In another ssh/terminal session to NPS:
sleep $((20*60))  # let the above misery "warm up" awhile before starting blktrace
nzBlkTracesStart.sh  # start blktrace capture
sleep $((10*60))  # collect blktrace data for 10 min while misery is running
nzBlkTracesStop.sh & # stop blktrace capture
# Move the captured data out of the way, via the "transfer raw data off of NPS" step (#6, below),
# then repeat the above 10 min blktrace capture a few more times, at various places in misery.
```

Step 6: Transfer Raw Data Off Of NPS

On the NPS host, create a tarball, giving it a good identifying name, something like this (some tools might have trouble coping with long filenames, and/or names with unusual characters, in which case you can temporarily rename or use a symlink having a short/plain name):

```
cd /migic
tar czf blktrace-TF0:6.2.01-Atomics,TPC-DB@100b-2011-08-22.tgz blktraceBlade=*
```

Then, scp this tarball to your workstation/laptop. You may also want to transfer any other useful information you care about, such as the nohup.out and/or dists/summary files from PerfBar (if you used PerfBar), though I didn't bother doing this in my case. the size of the tarball(s) were in the 0.5 GB range.

Step 7: Clean Up

At this point, you're finished with the NPS system, so you can clean it up and use it for other purposes if you wish.

Running Blkparse/Btt (On Workstation)

To post-process the Blktrace logfiles ("Blktrace post-processing" phase), do the following:

Step 8: Run nzBlkTracePostProcess.sh

On your workstation, untar your tarball, and invoke the nzBlkTracePostProcess.sh script (from the untar directory, containing the blktraceBlade=A,B,C,D log directories). This script calls several subscripts:

- nzBlkRawVerify.sh – Runs blkrawverify, from the blktrace suite.
- nzBlkParse.sh – Runs blkparse, from the blktrace suite.
- nzBtt.sh – Runs btt, from the blktrace suite.
- nzBttFindInteresting.sh – A home-grown tool.

The running of nzBlkTracePostProcessing.sh writes verbose progress messages to stdout (most of which comes from the underlying programs blkparse and btt of the Blktrace suite), the most interesting of which is a list of "interesting" files at the end. The list of interesting files is also saved to a file called interesting.txt.

Step 9: Examine interesting.txt

By definition, the "interesting" files are those associated with the NPS data block I/O devices (i.e., the "candidate" devices mentioned above that aren't of interest for NPS purposes are filtered out by nzBtFindInteresting.sh as uninteresting). In the case of the TwinFin 6, for example, there were 46 "interesting" devices (because the TwinFin6 has 48 data disks, 2 of which are reserved as spares and so were unused in my testing).

The list of files associated with such an "interesting" device is given in interesting.txt, in "ls -l" format. A subset of this information (just the name and size, not all the "ls -l" information) for a typical device is shown here (this is from the 10.0.0.84 blade of the Mugry@15000b example tarball, mentioned above). Here, the "*.blktrace.<N>" files are the original raw blktrace binary logfiles for the CPUs of the blade (numbered 0..7, where "CPU" means "logical processor", which includes cores and/or hyperthreads if present). The "*.blkparse.[bin|txt]" files are the blkparse output files, in both binary and ASCII formats (including per-program statistics, see "-s" option to blkparse). The "*.btt *" files are of course the btt output files. The "*.blkparse.csv" file is our own invention, see below.

```
68478592 sdh.blkparse.bin
80942212 sdh.blkparse.csv
 1447071 sdh.blkparse.txt.summary
 8516882 sdh.blktrace.0
 9878730 sdh.blktrace.1
 9383886 sdh.blktrace.2
 8147576 sdh.blktrace.3
 8132146 sdh.blktrace.4
 7494792 sdh.blktrace.5
 9865304 sdh.blktrace.6
 7208172 sdh.blktrace.7
   13796 sdh.btt.txt.avg
 4485137 sdh.btt.txt.dat
    7232 sdh.btt.txt_chist.dat
       0 sdh.btt.txt.msg
    7242 sdh.btt.txt_qhist.dat
     616 sdh.btt.txt.sys
   16794 8,112_d2c_plat.dat
    6631 8,112_iops_fp.dat
 5952304 8,112_live.dat
   11895 8,112_mbps_fp.dat
 7719253 8,112_pit.dat
   16794 8,112_q2c_plat.dat
   16755 8,112_q2d_plat.dat
 6796804 aqd_8,112_aqd.dat
 4484944 bno_8,112_c.dat
 4179672 bno_8,112_r.dat
  384072 bno_8,112_w.dat
 4731678 d2c_8,112_d2c.dat
 4731663 q2c_8,112_q2c.dat
 4731670 q2d_8,112_q2d.dat
 4716126 seeks_8,112_c2c_c.dat
 3951392 seeks_8,112_c2c_r.dat
  284736 seeks_8,112_c2d_w.dat
 7615008 seeks_8,112_c2q_c.dat
 7340864 seeks_8,112_c2q_r.dat
  276144 seeks_8,112_c2q_w.dat
   19404 sps_8,112.dat
     185 unplug_8,112.dat
```

Analysis

Now you can do the headfun stuff:

Step 10: Examine/Analyze Interesting Files, And Generate Reports (Numerics/Graphics)

Note there are some options to the Blktrace suite programs (esp. the btt program) that the user can tweak. As always, you should read the documentation (blktrace.pdf, btt.pdf) for details, but we've already added many/most of them to the scripts provided with this wiki page (which makes the scripts rather long-running, producing a lot of files that take up a lot of space). Such options can be modified in the scripts simply by editing the source code of the scripts themselves (search the source code for the calls to the blktrace suite programs, and change the options as you wish).

The choice of options affects the set of interesting files generated. The contents of the interesting files is described in the documentation (blktrace.pdf, btt.pdf). (The documentation is quite good, though it seems not to have quite kept up with the software, for example, the btt.pdf doc speaks of "Q2I", which now seems to be decomposed into "Q2G" and "G2I", and some of the options to btt don't seem to function as advertised.) We explore them in the Extended Example, below. Some of them, for example, are suitable for importing into spreadsheets and/or databases, for further study and/or graphing purposes.

Finally, we note that by using the raw blkparse data (i.e., not post-processed by btt), you can generate ever more detailed statistics as desired. This deeper level of post-processing may be necessary if the information you seek isn't available from the standard/canned blkparse/btt output files. We do some of that in the Extended Example, below.

Especially, for the deepest investigations (Exploratory Data Analysis, EDA), you'll probably want to "slice and dice" your dataset in various ways of particular interest for you (for example, to pare down your dataset to regions of interest to you, because very large dataset cannot be handled by the blktrace suite, or many other tools). That can be done in many ways (such as shell or python scripts hacking away at blkparse.txt [there's an example in nzBlkParseFilter.sh/py, which you should probably ignore]), but the best way is to import blkparse.txt into a SQL DBMS (such as NPS itself, though that's overkill). Then, you can further post-process your (possibly pared-down) dataset using, say, a spreadsheet-like application (such as MS Excel, OpenOffice.org Calc, Gnumeric, or R), thereby generating appropriate numerics and graphics. Again, we do some of that in the Extended Example, below.

But first, note that you can't quite "merely import" your blkparse.txt into a SQL DB: you first need to define a "blkparse schema", because the blkparse.txt file is "plain ASCII" (i.e., not SQL-data-typed). And, once you've defined a blkparse schema, the blkparse.txt file should be in CSV format (not just "plain ASCII"), for ease of importing into the SQL DB (the nzBlkTracePostProcess.sh/nzBlkParse.sh tools do this). An appropriate blkparse schema is defined in the file, CreateTblBlkparse.scl (included in the attached perfbin.tgz tarfile).

Here's how I did it, on my Fedora workstation/laptop, once PostgreSQL was installed (for simplicity, I just used PostgreSQL under my own identity, not as a service under the usual postgres identity):

```
$- export PGDATA=~/PostgreSqlData.d PGLOG=~/PostgreSqlLog.txt PGUSER=$USER PGDATABASE=blktrace  # PG
$- createdb $PGDATA
$- pg_ctl start  # also: status, restart, stop
$- psql postgres  # "postgres" = name of basic pre-installed DB (if $PGDATABASE=blktrace DB already
postgres=# create database blktrace;
postgres=# \c blktrace
postgres=# \i CreateTblBlkparse.sql
postgres=# copy blkparse from '/path/to/your/xyz.blkparse.csv' delimiters '|' csv;  -- import your x
postgres=# select count(*) from blkparse;  -- check this looks right to you (wc -l xyz.blkparse.csv)
```

Incidentally, once you have your blkparse.csv data in the blkparse SQL table, you can use SQL on it to recreate most (if not all) the subsidiary files that the btt program generates – plus a lot more. We leave that as a proverbial "exercise for the reader", though we do some of it in the Extended Example below.

An Extended Example

To pique the reader's interest in the blktrace suite, we will now conduct an in-depth analysis of the sdh device, from the 10.0.9.84 blade of the MLqry@15000b tarball, mentioned above.

We emphasize again: To really understand all this, you must first read (or at least follow along in) the blktrace/blkparse/btt documentation (attached PDFs, blktrace.pdf, btt.pdf).

The "Summary" File: sdh.blkparse.txt.summary

This file summarizes I/O information at multiple levels: per-process, per-CPU, and per-device. Here, for example, is the per-device roll-up:

```
Total (sdh):
 Reads Queued:     220,482,    20,368MiB  Writes Queued:       8,506,   344,892KiB
 Read Dispatches:  123,481,    20,368MiB  Write Dispatches:    8,273,   344,892KiB
 Reads Requeued:         0               Writes Requeued:         0
 Reads Completed:  123,481,    20,368MiB  Writes Completed:   14,197,   547,884KiB
 Read Merges:      105,916,    12,557MiB  Write Merges:          297,    20,016KiB
 PC Reads Queued:        0,        0KiB  PC Writes Queued:        0,        0KiB
 PC Read Disp.:         14,        0KiB  PC Write Disp.:          0,        0KiB
 PC Reads Req.:          0               PC Writes Req.:          0
 PC Reads Cmpl.:        14               PC Writes Cmpl.:    14,197
 IO unplugs:        58,883               Timer unplugs:      53,774

Throughput (R/W): 34,149KiB/s / 884KiB/s
Events (sdh): 1,327,458 entries
Skips: 0 forward (0 -   0.0%)
```

The "Averages" File: sdh.btt.txt.avg

We begin by examining (the bottom part of) the sdh.btt.txt.avg file. (The accompanying file sdh.btt.txt.xvg is a not-very-useful watered-down version of this one, less information, but "easier to parse", i.e., in a more CSV-like format.)

It shows, for example, that 237,968 requests/jobs were processed by the I/O scheduler during the period of this data capture (the "N" column of "Q2C" = queueing-of-job to completion-of-job, i.e., overall lifetime of an I/O job), and that the "average" (i.e., mean) queue-to-completion elapsed time for an I/O job was approx. 1.4 millisec (0.014028259 sec.).

As another example, it shows that the number of "Q2Q" "incoming seeks" issued to the I/O scheduler was 237,968 (that is, the same as the total number of I/O jobs); but the number of "D2D" "outgoing seeks" issued to the device was less than half that number, only 131,754 (due to merging of adjacent blocks). Correspondingly, the mean "distance" (absolute difference between device sector numbers) was 80,995,417.7 in the Q2Q case, but 107,945,313.0 in the D2D case (again, due to merging).

The "Active Requests At Q Information" table shows that on average, only 2.5 requests are in the block I/O scheduler's queue for this device whenever a new request comes in to it.

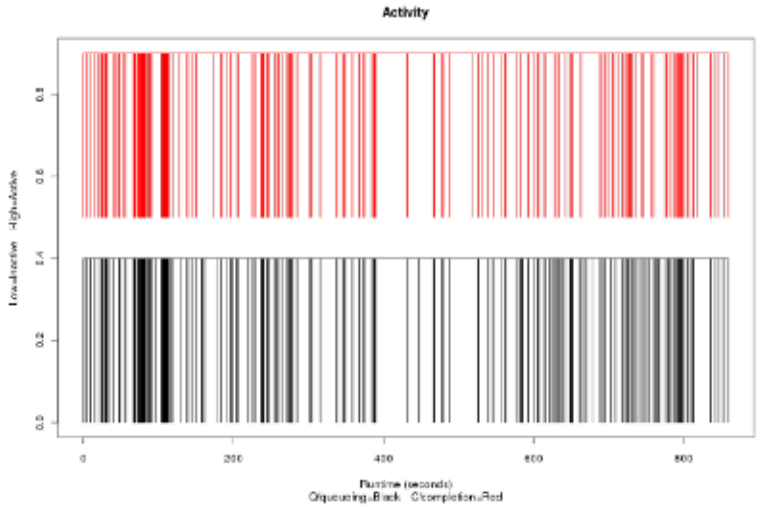
The "I/O Active Period Information" table shows that this device was "active" 50.9% of the time, where "active" is defined to mean: "Q=queueing or C=completion activities/events happened within 0.1 sec (a tunable parameter) of one another".

```
==================== All Devices ====================

            ALL           MIN           AVG           MAX           N
--------------- ------------- ------------- ------------- -----------

Q2Q               0.000000027   0.001613338   1.348638054      237068
Q2G               0.000000297   0.000001095   0.032318554      131759
G2I               0.000000279   0.000136048   0.106924407      137651
Q2M               0.000000584   0.000000859   0.002861800      105213
I2D               0.000001107   0.003785098   0.130117535      131756
M2D               0.000000651   0.005119644   0.130106508      105213
D2C               0.000060003   0.000506528   0.436833142      237069
Q2C               0.000612017   0.014826258   0.436872039      237069

==================== Device Overhead ====================

       DEV |       Q2G       G2I       Q2M       I2D       D2C
---------- | --------- --------- --------- --------- ---------
 (  8,112) |   0.0078%   0.5643%   0.0027%  14.9426%  68.1946%
---------- | --------- --------- --------- --------- ---------
   Overall |   0.0078%   0.5643%   0.0027%  14.9426%  68.1946%

==================== Device Merge Information ====================

       DEV |       #Q       #D   Ratio |   BLKmin   BLKavg   BLKmax    Total
---------- | -------- -------- ------- | -------- -------- -------- --------
 (  8,112) |   237069   131754     1.8 |        1      491     1024 194428488

==================== Device Q2Q Seek Information ====================

       DEV |          NSEEKS            MEAN          MEDIAN | MODE
---------- | --------------- --------------- --------------- | ---------------
 (  8,112) |          237068      68905617.7               0 | 0(153389)
---------- | --------------- --------------- --------------- | ---------------
   Overall |          NSEEKS            MEAN          MEDIAN | MODE
   Average |          237068      68905617.7               0 | 0(153389)

==================== Device D2D Seek Information ====================

       DEV |          NSEEKS            MEAN          MEDIAN | MODE
---------- | --------------- --------------- --------------- | ---------------
 (  8,112) |          131754     187045513.0               0 | 0(79033)
---------- | --------------- --------------- --------------- | ---------------
   Overall |          NSEEKS            MEAN          MEDIAN | MODE
   Average |          131754     187045513.0               0 | 0(79033)

==================== Plug Information ====================

       DEV |    # Plugs # Timer Us  | % Time Q Plugged
---------- | ---------- ---------- | ----------------
 (  8,112) |       4028(      53774) |  52.489272414%

       DEV |    IOs/Unp   IOs/Unp(to)
---------- | ----------   ----------
 (  8,112) |        2.3          2.1
---------- | ----------   ----------
   Overall |    IOs/Unp   IOs/Unp(to)
   Average |        2.3          2.1

==================== Active Requests At Q Information ====================

       DEV |  Avg Reqs @ Q
---------- | -------------
 (  8,112) |           2.5

==================== I/O Active Period Information ====================

       DEV |     # Live      Avg. Act     Avg. !Act % Live
---------- | ---------- -------------- -------------- ------
```

The "Activity" File: sdh.btt.txt.dat

This file contains detailed "Q/queueing and C/completion activity information" (in the sense defined above, where "activity" still refers to 0.1 sec.). As discussed in btt.pdf, this information is in a format that is directly graphable by the Grace graphing tool. If you don't use Grace (and who does? [most people use Excel, I use R]), you'll need to can do some text editing work to put the file into a format acceptable to your tool (for example, create 2 files, one for Q activity and one for C activity, then put each into CSV format for importation to a spreadsheet-like tool).

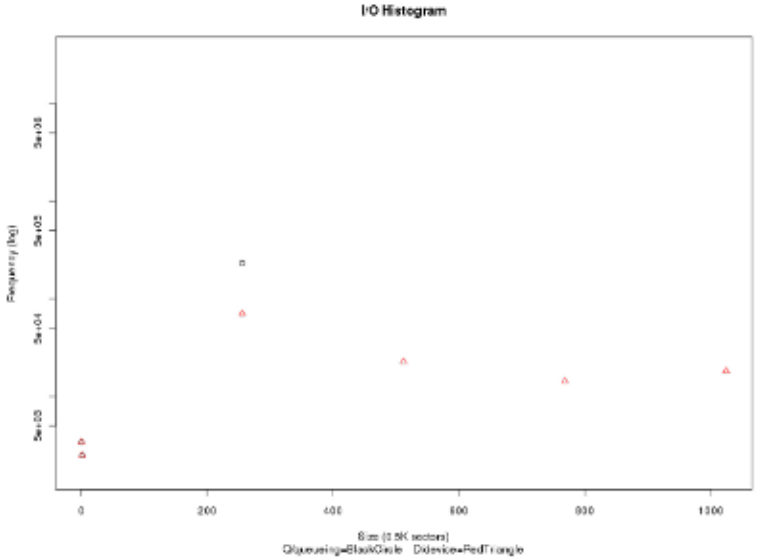
Here is the graph for our example. It shows that this device is nearly constantly active, at least insofar as the resolution of this graph can distinguish. (But if you "zoom in" to a higher degree of magnification/resolution, you can see more detail, of course [we don't do that with this table, though we do it with some other tables, below].)



The "Histogram" Files: sdh.btt.txt_[d|q]hist.dat

These 2 files give information about the sizes of I/O for "incoming" ("Q/queueing") and for "outgoing" ("D/device") requests, in a format suitable for producing histogram graphics.

In the btt.pdf document, the Q and D histograms are graphed separately, but for our (NPS) purposes it's better to graph both on a single graphic. The resulting histogram is shown here:



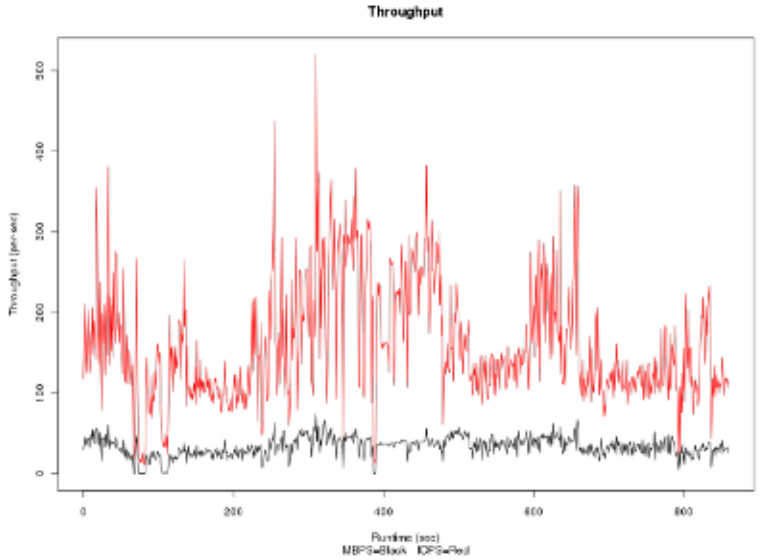
As you can see, in the case of NPS, this histogram graphic is not very informative. That's because NPS deals with table-level data in units of "extents" of size 3MB, by issuing 24 consecutive "page" I/Os to the Linux block I/O scheduler, each "page" being of size 128KB (= 128 1K OS-blocks = 256 0.5K disk-sectors). That's why, in the histogram graphic above, "all" the Q'BlackCircles are of size 256. Because of merging by the block I/O scheduler, these "all" become D'RedTriangles of size a multiple of 256. I.e., 256/512/768/1024. (It's unclear why these multiples max out at 1024 – undoubtedly it's a property of the block-I/O subsystem, but why?) The reason "all" is in quotes here is that some non-NPS activity is present, resulting in a few very small I/Os, and these are seen at the lower-left of the histogram graphic, with overlaid circles and triangles. So actually, in the NPS case, a numeric table is actually clearer than the histogram graphic. Here is that information (extracted from the sdh.bit.txt, [d|q]hist.dat files, by omitting I/Os with frequency 0):

Size	Freq	
	Q	D
1	3484	3484
2	2492	2492
256	232877	78451
512	0	22778
768	0	14342
1024	0	18255

The "Throughput" Files: <maj>,<min>_[io|mb]ps_fp.dat

As we know (from the "averages" file, and the listings in interesting.txt), the major/minor numbers corresponding to our sdh device are 8/112. The 2 files 8,112_[io|mb]_fp.dat provide (equivalent) information about the per-second throughput rate of our device, measured in 2 (equivalent) ways: (i) IOPS (number of I/O operations per sec); (ii) MBPS (number of MB per sec).

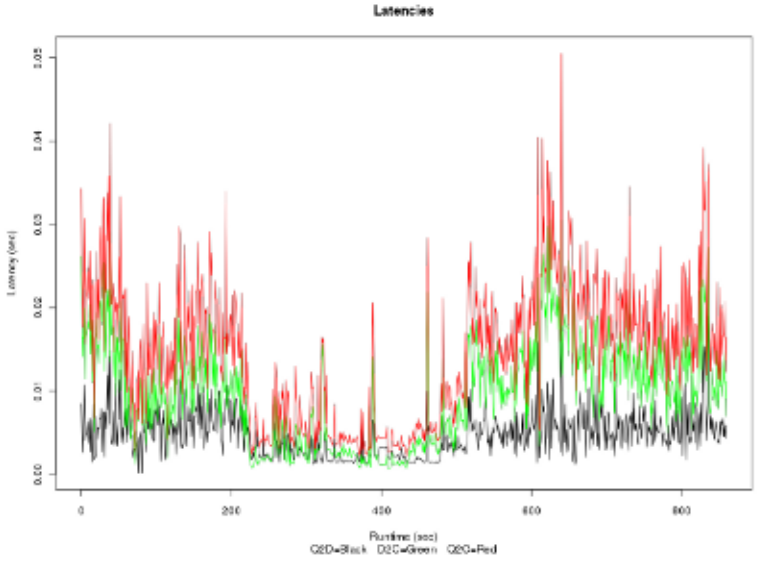
Again, these files are formatted for Grace graphics (but we use R). And as with the histogram files/graphics above, we join them in a single graph (as opposed to the btt.pdf document, which graphs them separately):



The "Latency" Files: <maj>,<min>_[q2d|d2c|q2c]_plat.dat

The 3 files 8,112_[q2d|d2c|q2c]_plat.dat provide information about per-second latency for our device, measured in 3 different ways: (i) q2d = queue-to-device = time I/O requests spend in the I/O-scheduler's queue; (ii) d2c = device-to-completion = time I/O requests spend in device controller/hardware; (iii) q2c = queue-to-completion = overall time of whole I/O requests (= q2d + d2c, approximately).

Once again, we superimpose these 3 graphs on a single graphic (the btt.pdf document does not show example graphs of this):



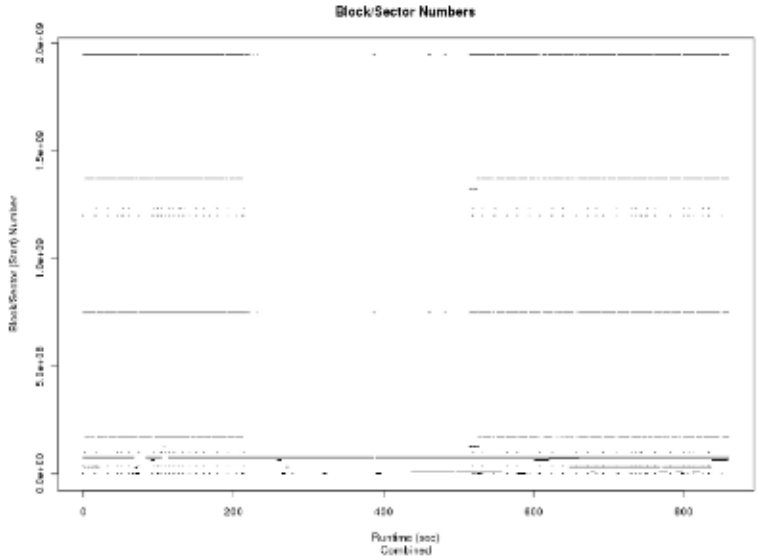
The "Block/Sector Number" files: bno_<maj>,<min>_[r|w|c].dat

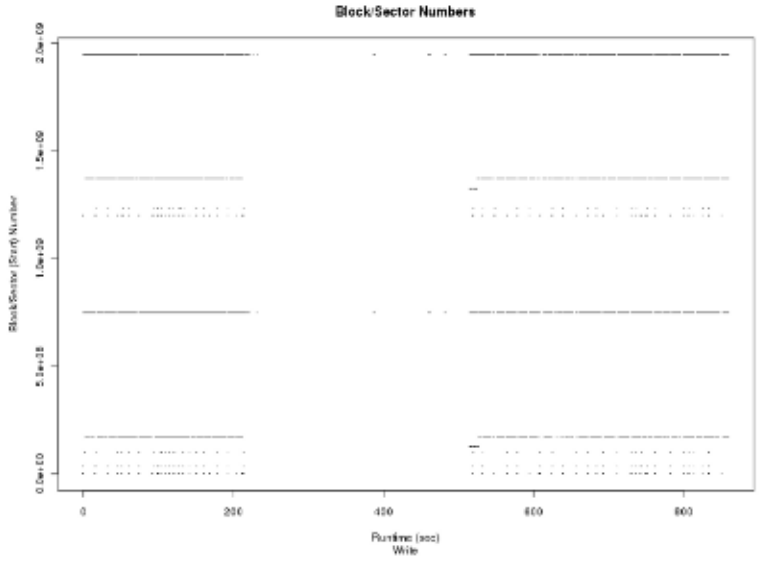
First, a matter of terminology in this area (which has been inconsistent since disk technology was introduced in the 1950's). The blktrace/blkparse/btt tools use the word "block", by which they mean the basic unit of data that can be addressed or I/O'd from/to the storage device. However, in the terminology of storage-device-land, the term for this unit is "sector" (it is a subdivision of a "track", which is a subdivision of a "cylinder"). For the disk technology currently used in NPS, the size of a sector is 512 (0.5K) bytes. Linux also uses the term "sector" in the same way, and uses "block" to mean the unit of filesystem allocation (which is always a multiple sector size, and is 1K bytes for most filesystem types). [In Windows, "cluster" is used to mean some combination of "block" in this sense and "pie-wedge-shaped chunk of platter" (apparently).] This is why we sometimes use the terminology "block/sector" (by which we mean "block in the blktrace sense, i.e., sector in the storage sense"). Another term you'll see is "logical block address (LBA)", which we're just calling "block/sector number" here.

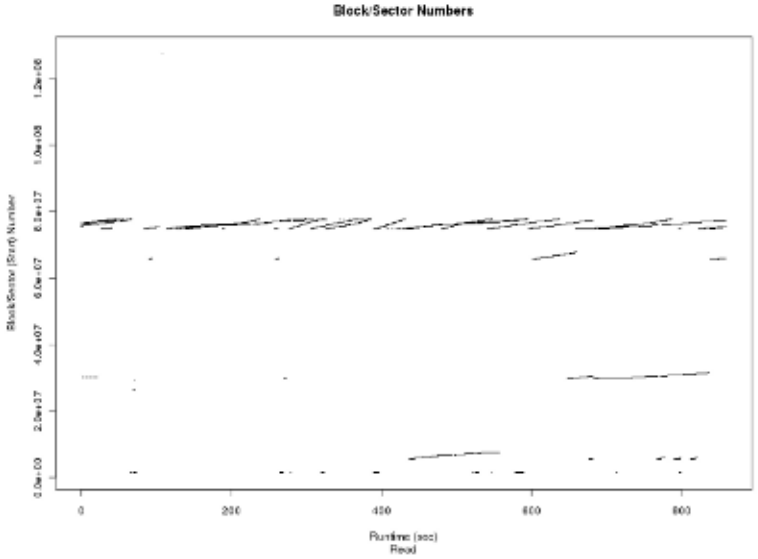
The 3 files bno_8,112_[r|w|c].dat provide the start/end block/sector numbers of all I/Os, where r/w/c mean read/write/combined. Since these files record every individual I/O, they represent a lot of data, and graphs can be hard to decipher (unless you zoom in).

The following are the 3 graphs for our device, at a rather coarse level of graphing graularity – we may/will refer to the first 2 of the 3 as "50,000-foot views". We also speak in terms of the following "abuse of language": "Every dot is a datapoint". By this, we really mean: "Every dot on the graph is supposed to represent a datapoint, however some datapoints may be coalesced into single/overlapping dots due to too-coarse granularity of the graphics medium/device."

Note that the scale for the "read" case below (the 3rd of the 3 graphs) is different from the first two (and so we refer to it as a "25,000-foot view"). That's because the reads happen in a narrower (horizontal) band of sector numbers, but we want to keep all graphs the same size (nominally 9-by-6 inches) for this wiki page. Graphically, this has the effect of "magnifying" (a.k.a. "zooming-in on") that "read" data (all of which occurs in the lowest band of the graph). We're going to be exploring the "read" case more closely, below, but at this point we simply note how the zooming-in already brings out some additional interesting detail for the "read" case.


Block/Sector Numbers
Block/Sector (Start) Number
0
200
400
600
800
Runtime (sec)
Combined


Block/Sector Numbers
Block/Sector (Start) Number
0
200
400
600
800
Runtime (sec)
Write



As far as "large-scale structure" goes, we immediately notice there are 4 distinctly visible "horizontal bands" (looking specifically at the "combined" graph, the first of the 3 above). What do those represent? These represent the disk partitions, with: (i) the lowest band (smallest block/sector numbers, outermost disk cylinders) denoting the disk's NPS primary partition; (ii) the next-lowest band denoting the nztmp partition <<TBD>>; (iii) <<TBD>>; (iv) the highest band (largest block/sector numbers, innermost disk cylinders) denoting the disk's NPS mirror secondary partition. <<TBD – THIS NEEDS CONFIRMATION AND BETTER EXPLANATION>>

There's another "large-scale structure" that jumps out at us, namely, the "vertical bands". Specifically, there's a huge vertical (nearly-blank-zone region in the middle in the middle third of (first 2 of the) graphs above, but the 2 outer-third regions aren't so nearly-blank. What does that mean? What it means (comparing the first 2 graphs with the 3rd one) is that there are no writes (only reads) happening in the middle-third, but writes are happening in the outer 2 thirds. This is a function of the workload (mostly), not of the OS (we're not seeing some arbitrary Linux daemon doing funny things here, or anything like that) – because, recall, the device we're studying is dedicated to NPS storage, so any activity happening on it is due to the workload. (Or possibly to some sort of non-workload-related NPS bookkeeping chores? <<TBD>>)

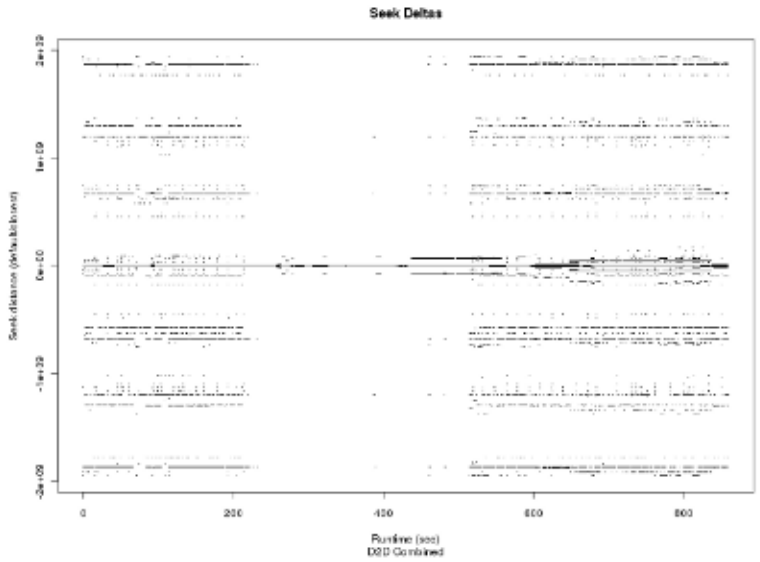
Here's what the (approximate) time-bounds are for that big no-write blank-zone (it's a time-range of ~300 sec = ~5 min):

```
blktrace=# select max(elapsedtime) from blkparse
blktrace-# where (1.0e+09<sectstrt and sectstrt<1.5e+09) and elapsedtime<400;
213.113867321
blktrace=# select min(elapsedtime) from blkparse
blktrace-# where (1.0e+09<sectstrt and sectstrt<1.5e+09) and elapsedtime>400;
515.582688371
```

The "Seek Deltas" Files: seeks_<maj>,<min>_[q2q|d2d]_[r|w|c].dat

These 6 files provide detailed information about "seek deltas" (= difference/distance between sector numbers for successive operations), for: (i) q2q (I/O-scheduler queue seeks) and d2d (actual device seeks); (ii) r/w/c (read/write/combined).

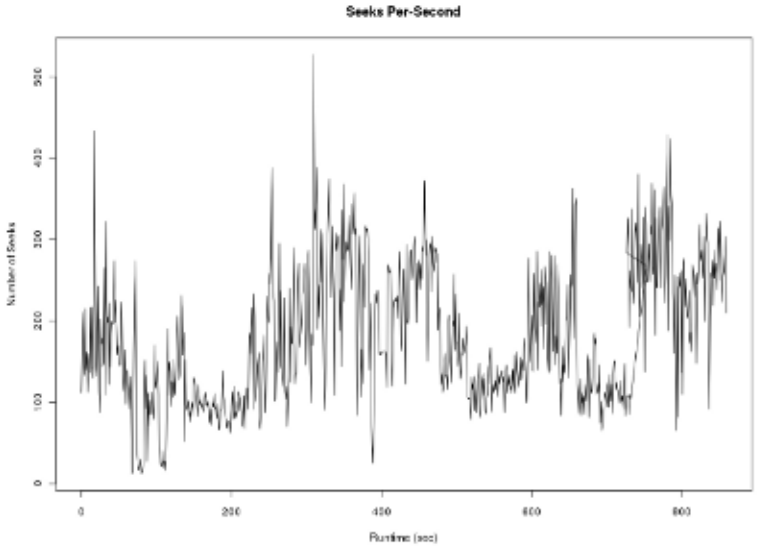
Here's the "d2d combined" graph for our device (every dot is a datapoint):



The "Seeks Per Second" File: sps_<maj>,<min>.dat

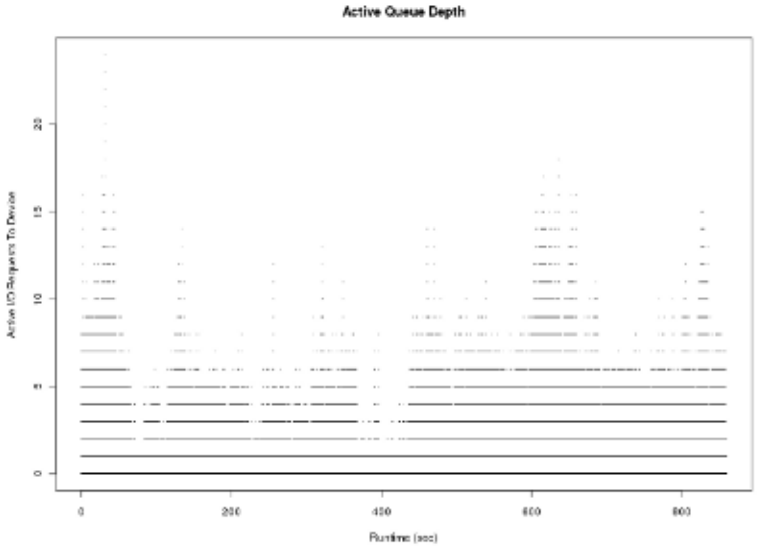
The file sps_8,112.dat provides seeks-per-second information about our device. Here's the graph.

<<TBD – OPEN QUESTION>> Does this measure "seeks" at the Q/queueing level or the D/device level? The documentation (btt.pdf) isn't clear at all about this, though I think it's D/device (which is what it "should" be). Furthermore, there seems to be something broken around the 750 sec. mark (time seems to move backward).



The "Active Queue Depth" File: aqd_<maj>,<min>_aqd.dat

The file aqd_8,112_aqd.dat provides the "active queue depth" of the device, i.e., the number of requests issued to the device that have not yet completed. Here's the graph for our device (every dot is a datapoint):



Deeper Investigation (Exploratory Data Analysis): SQL

The work we've done above in this Extended Example nearly (though not quite, see btt.pdf for a bit more) exhausts what we can learn from a straightforward (first-order) examination of the blkparse/btt output files. One of the main problems with what we've learned is that some of the graphics produced above are too coarse-grained ("50,000 foot level"). Some of the graphs are OK, namely those with "relatively few" datapoints, or at least whose datapoints have "relatively few" numbers of distinct x/y-coordinates. The problem comes with graphs having so many datapoints that they must be plotted using "very small dots" to represent the datapoints (otherwise the graphed datapoints become too clustered, and the graph becomes too cluttered to interpret properly). Such as, the Block/Sector Number graphs above (the "combined" one, for example, has "wc -l bno_8.1.12_c.dat" = 131,754 datapoints – too many for nominally 9-by-6 inch graphs).

So to go further, we need some additional tools. Our tool of choice is SQL, together with data graphics (via R), to "zoom-in" on areas of particular interest. Recall that we already imported our sdh.blkparse.txt file into PostgreSQL, above. To begin with, observe that our blkparse table looks like the following (from the CreateTblBlkparse.sql file, in perfbin.tgz):

```
CREATE TABLE blkparse (
    Maj         SMALLINT      NOT NULL,  -- device major number
    Min         SMALLINT      NOT NULL,  -- device minor number
    CPU         SMALLINT      NOT NULL,  -- CPU ID
    SeqNo       INT           NOT NULL,  -- sequence number (per CPU ID)
    ElapsedTime NUMERIC(16,9) NOT NULL,  -- elapsed time, sec.microsec, starts at 0.0 (a.k.a. "runti
    PID         SMALLINT      NOT NULL,  -- process ID
    Act         VARCHAR(2)    NOT NULL,  -- trace "actions" (a.k.a. events), see blktrace.pdf
    RWBS        VARCHAR(3)    NOT NULL,  -- R/W/B/S, see blktrace.pdf
    SectStrt    BIGINT        NOT NULL,  -- I/O sector start number
    SectCnt     SMALLINT      NOT NULL,  -- I/O sector count
    ErrNo       SMALLINT      NOT NULL,  -- error number (0 means success)
    Command     VARCHAR(32)   NOT NULL,  -- "command", often "(null)"
    InfoPlus    VARCHAR(64)   NOT NULL   -- other, see btt.pdf (not well documented)
);
```

Not surprisingly, the ElapsedTime field is a primary key to our Blkparse table:

```
blktrace=# select count(elapsedtime) from blkparse;
1327449
blktrace=# select count(distinct elapsedtime) from blkparse;
1327449
```

Using SQL one can re-compute many/most/all of the statistics that are present in the Btt output files (mentioned above as an "exercise for the reader"), plus lots of other things. As a simple example: what are the sizes (sectcnt, number of sectors) of the reads/writes that NPS issues?:

```
blktrace=# select distinct sectcnt from blkparse where act='D' and rwbs='R';
     0
   256
   512
   768
  1024
blktrace-# select distinct sectcnt from blkparse where act='D' and rwbs='W';
     1
     2
   256
   512
   768
  1024
```

Deeper Investigation (Exploratory Data Analysis): Graphics (Esp. Zooming-In)

We can now conduct some graphical exploration of our data ("read" case of block/sector numbers), aided by a modicum of SQL. In particular, we can visually enhance our data by "zooming-in" ("getting closer than the 50,000-foot level").

We've already generated the "regular" (coarse-grain) graph of "read" block/sector numbers from the btt output file bno_8,112_r.dat, see above. Looking at that graph again, which we already noted is at a different scale from the other ("combined", "write") block/sector number graphs, we see that it "should" be at even a different scale that it is. Because, there's a huge blank region at the top of the graph that "should" be eliminated: there "seems to be" only a single datapoint with value >8.0e+07 in that region (it occurs around the 100 sec. mark). We can test that conjecture (noting that "sectcnt" is the y-axis of our graph):

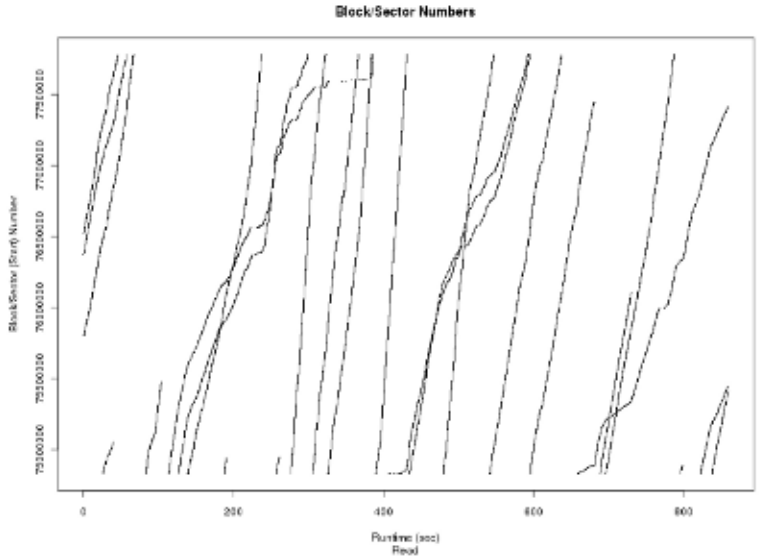
```
$- psql
blktrace=# select * from blkparse
blktrace-# where rwbs='R' and sectstrt>8.0e+07;  -- 27 datapoints
 maj | min | cpu | seqno |  elapsedtime  | pid | act | rwbs | sectstrt  | sectcnt | errno | command
-----+-----+-----+-------+---------------+-----+-----+------+-----------+---------+-------+---------
   8 | 112 |   0 | 17515 | 107.892314893 |  27 | A   | R    | 127656002 |     25G |     0 | (null)
   8 | 112 |   0 | 17516 | 107.892314416 |  27 | Q   | R    | 127656002 |     25G |     0 | (null)
   8 | 112 |   0 | 17517 | 107.892316935 |  27 | G   | R    | 127656002 |     25G |     0 | (null)
   8 | 112 |   0 | 17518 | 107.892319137 |  27 | I   | R    | 127656002 |     25G |     0 | (null)
   8 | 112 |   0 | 17520 | 107.892328634 |  27 | A   | R    | 127656258 |     25G |     0 | (null)
   8 | 112 |   0 | 17521 | 107.892328775 |  27 | Q   | R    | 127656258 |     25G |     0 | (null)
   8 | 112 |   0 | 17522 | 107.892329093 |  27 | M   | R    | 127656258 |     25G |     0 | (null)
   8 | 112 |   0 | 17523 | 107.892337965 |  27 | A   | R    | 127656514 |     25G |     0 | (null)
   8 | 112 |   0 | 17524 | 107.892338100 |  27 | Q   | R    | 127656514 |     25G |     0 | (null)
   8 | 112 |   0 | 17525 | 107.892338894 |  27 | M   | R    | 127656514 |     25G |     0 | (null)
   8 | 112 |   0 | 17526 | 107.892347231 |  27 | A   | R    | 127656770 |     25G |     0 | (null)
   8 | 112 |   0 | 17527 | 107.892347372 |  27 | Q   | R    | 127656770 |     25G |     0 | (null)
   8 | 112 |   0 | 17528 | 107.892347954 |  27 | M   | R    | 127656770 |     25G |     0 | (null)
   8 | 112 |   0 | 17529 | 107.892356820 |  27 | A   | R    | 127657026 |     25G |     0 | (null)
   8 | 112 |   0 | 17530 | 107.892356965 |  27 | Q   | R    | 127657026 |     25G |     0 | (null)
   8 | 112 |   0 | 17531 | 107.892357729 |  27 | G   | R    | 127657026 |     25G |     0 | (null)
   8 | 112 |   0 | 17532 | 107.892358528 |  27 | I   | R    | 127657026 |     25G |     0 | (null)
   8 | 112 |   0 | 17533 | 107.892366008 |  27 | A   | R    | 127657282 |     25G |     0 | (null)
   8 | 112 |   0 | 17534 | 107.892367043 |  27 | Q   | R    | 127657282 |     25G |     0 | (null)
   8 | 112 |   0 | 17535 | 107.892367087 |  27 | M   | R    | 127657282 |     25G |     0 | (null)
   8 | 112 |   0 | 17536 | 107.892375803 |  27 | A   | R    | 127657538 |     25G |     0 | (null)
   8 | 112 |   0 | 17537 | 107.892375150 |  27 | Q   | R    | 127657538 |     25G |     0 | (null)
   8 | 112 |   0 | 17538 | 107.892375690 |  27 | M   | R    | 127657538 |     25G |     0 | (null)
   8 | 112 |   0 | 17541 | 107.899835430 | 121 | D   | R    | 127656002 |    1024 |     0 | (null)
   8 | 112 |   0 | 17542 | 107.899844827 | 121 | D   | R    | 127657026 |     768 |     0 | (null)
   8 | 112 |   3 | 19745 | 107.923300630 |   0 | C   | R    | 127656002 |    1024 |     0 | swapper
   8 | 112 |   7 | 17699 | 107.938876710 |   0 | C   | R    | 127657026 |     768 |     0 | swapper
(27 rows)
```

Sure enough, there's only a tiny amount of read activity at high-sector-numbers (127,656,002-127,657,538), all occurring in a burst within an 0.05 sec. time span (107.89-107.94), most of it from pid::27 on cpu::0, and only 4 of them are act::D/C. (Note it's enough to just check "rwbs='R'", as you can verify by executing the SQL statement "select distinct rwbs from blkparse;", which shows that the only possibilities are R/WWB/WBSN, where N is an undocumented value meaning "N/A", not applicable. [BTW, if you're interested, you can also check that "select distinct act from blkparse;" yields A/C/D/F/G/I/M/P/Q/U/UT.]) But there are 27 such datapoints, not just the single one that shows up on our graph. This just goes to illustrate the limitations of coarse graphics (all the high-sector-number read datapoints are glommed into just 1 single graph dot [but there are only 2 such points, not 27, because the bno_8,112_c.dat file only includes act='D' datapoints. I.e., datapoints that signify operations actually issued to the device]).

Still looking at our "regular" graph, we see that the majority of "read" activity happens in the region sectstrt>7.0e+07. There are certainly a lot of reads outside the region 7.0e+07..8.0e+07, but let's ignore that for the moment (and below, we're going to zoom-in on a time region (x-axis) that misses those low-sector-numbers anyway). This leads us to look at the datapoints generated by the following SQL query (the first 3 lines are called "CSV preamble" – they configure PostgreSQL to generate a CSV file, /tmp/foo1, with delimiter "|", suitable for graphing; note that the "\o" operation create/opens/truncates the file):

```
blktrace=# -- You want: "\t" state to be "Showing only tuples," and "\a" state to be "Output format
blktrace=# \t on
blktrace=# \a
blktrace=# \o /tmp/foo1
blktrace=# select elapsedtime,sectstrt from blkparse
blktrace-# where rwbs='R' and act='D'
blktrace-# and (7.0e+07<sectstrt and sectstrt<8.0e+07)
blktrace-# order by elapsedtime;  -- 102,442 datapoints
```

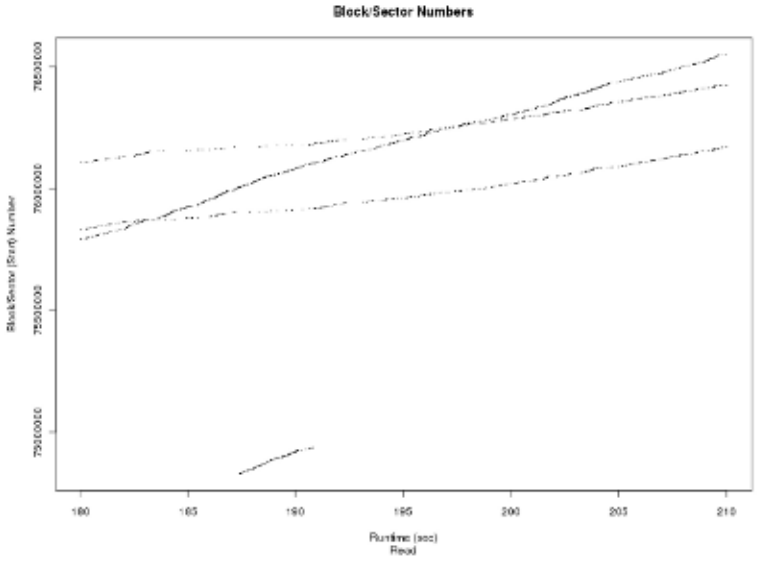
You can then plot /tmp/foo1, to get the following graph:



Now, this is starting to look really interesting! Notice how nicely the graph depicts the Linux block I/O scheduler's "elevator" algorithm (which is the **DEADLINE** scheduler here). The interpretation of the individual "strands" of the graph is "multiple SQL streams/sessions". Also, note that "steeper strands" correspond to "faster reading". And, "steeper strands" generally correspond to "fewer strands", because that corresponds to less resource contention (be warned that we're only looking at reads here, and there might be writes happening "behind our back", i.e., outside the viewport of this graph).

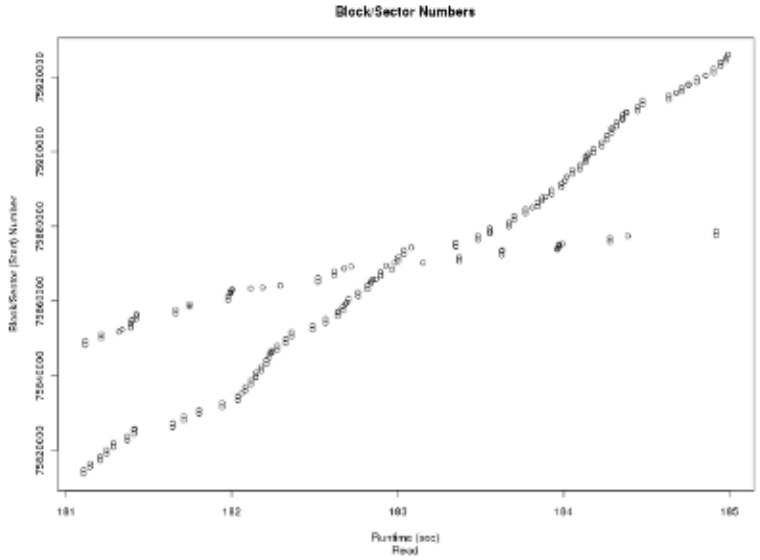
But this is still a pretty high-level view (we'll call it "10,000 feet"). Let's zoom-in yet closer on the very interesting-looking region 30-sec 180..210 (noting this captures all the read activity during this time range, in particular avoiding the low-sector-number reads as mentioned above):

```
blktrace=# -- CSV preamble as above, but for the file /tmp/foo2
blktrace-# select elapsedtime,sectstrt from blkparse
blktrace-# where rwbs='R' and act='D'
blktrace-# and (7.8e+07<=sectstrt and sectstrt<=8.8e+07)
blktrace-# and (180<=elapsedtime and elapsedtime<=210)
blktrace-# order by elapsedtime;  -- 2150 datapoints
```



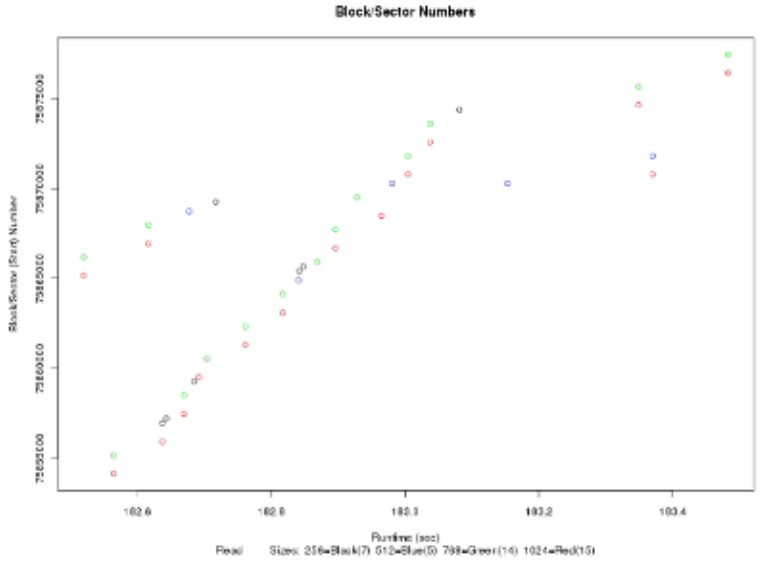
What about the "crossings" here? Do we get "collisions". Ie., do they actually touch some of the same blocks/sectors, for example? Well, to find out, let's zoom-in even closer, to a 4-sec region around the crossing on the left-hand-side of the graph (using circles now to mark the datapoints, instead of dots, which makes sense since there are now just a few datapoints):

```
blktrace=# -- CSV preamble as above, but for the file /tmp/foo3
blktrace=# select elapsedtime,sectstrt from blkparse
blktrace-# where rwbs='R' and act='D'
blktrace-# and (75600000<=sectstrt and sectstrt<=76000000)
blktrace-# and (101<=elapsedtime and elapsedtime<=105)
blktrace-# order by elapsedtime;  -- 197 datapoints
```



This comes close to answering our question about collisions, but there's still a bit of uncertainty. So, we zoom-in even closer, to a 1-sec region around the 183 sec mark (using colored circles now to denote the size, sectors, of the datapoints; the number of datapoints at each size is also denoted on the graph):

```
blktrace=# -- CSV preamble as above, but for the file /tmp/foo4
blktrace=# select elapsedtime,sectstrt from blkparse
blktrace-# where rwbs='R' and act='D'
blktrace-# and (75800000<=sectstrt and sectstrt<=76000000)
blktrace-# and (182.5<=elapsedtime and elapsedtime<=183.5)
blktrace-# order by elapsedtime;  -- 41 datapoints
```



Here, finally, we're fine-grained enough to answer our question about collisions. Namely, you can manually/visually count 41 distinct datapoints in this graph, and there are 41 datapoints in the dataset being graphed, according to SQL:

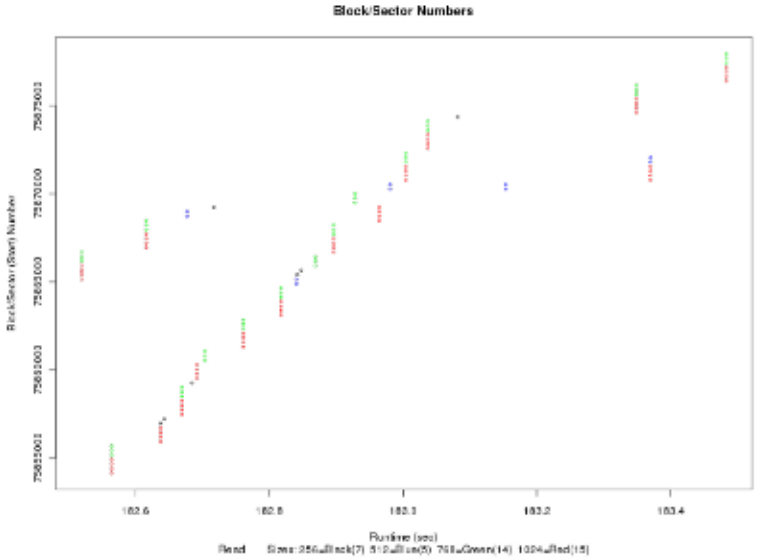
```
blktrace=# select count(*) from blkparse
blktrace-# where rwbs='R' and act='D'
blktrace-# and (75500000<=sectstrt and sectstrt<=76000000)
blktrace-# and (182.5<=elapsedtime and elapsedtime<=183.5);
41
blktrace=# select distinct count(*) from blkparse
blktrace-# where rwbs='R' and act='D'
blktrace-# and (75500000<=sectstrt and sectstrt<=76000000)
blktrace-# and (182.5<=elapsedtime and elapsedtime<=183.5);
41
blktrace=# select * from blkparse
blktrace-# where rwbs='R' and act='D'
blktrace-# and (75500000<=sectstrt and sectstrt<=76000000)
blktrace-# and (182.5<=elapsedtime and elapsedtime<=183.5)
blktrace-# order by elapsedtime;  -- 41 datapoints
 maj | min | cpu | seqno |  elapsedtime  |  pid  | act | rwbs | sectstrt | sectcnt | errno |  comma
-----+-----+-----+-------+---------------+-------+-----+------+----------+---------+-------+--------
   8 | 112 |   1 | 26067 | 182.519509971 |     0 | D   | R    | 75851154 |    1024 |     0 | swapper
   8 | 112 |   1 | 26068 | 182.519514870 |     0 | D   | R    | 75851678 |     768 |     0 | swapper
   8 | 112 |   2 | 28478 | 182.564337615 |   123 | D   | R    | 75854146 |    1024 |     0 | kblockc
   8 | 112 |   2 | 28479 | 182.564353656 |   123 | D   | R    | 75855170 |     768 |     0 | kblockc
   8 | 112 |   5 | 24626 | 182.618822385 |    32 | D   | R    | 75856946 |    1024 |     0 | (null)
   8 | 112 |   5 | 24627 | 182.618829860 |    32 | D   | R    | 75857970 |     768 |     0 | (null)
   8 | 112 |   4 | 25610 | 182.637477462 |   125 | D   | R    | 75851938 |    1024 |     0 | (null)
   8 | 112 |   4 | 25611 | 182.637486484 |   125 | D   | R    | 75852962 |     256 |     0 | (null)
   8 | 112 |   1 | 26078 | 182.643868622 |   122 | D   | R    | 75857218 |     256 |     0 | (null)
   8 | 112 |   7 | 27915 | 182.669471480 | 22898 | D   | R    | 75857474 |    1024 |     0 | xpu10
   8 | 112 |   7 | 27916 | 182.669478222 | 22898 | D   | R    | 75858498 |     768 |     0 | xpu10
   8 | 112 |   6 | 28027 | 182.677308399 |   127 | D   | R    | 75858738 |     512 |     0 | (null)
   8 | 112 |   6 | 27332 | 182.684883475 |   121 | D   | R    | 75859266 |     256 |     0 | (null)
   8 | 112 |   1 | 26120 | 182.692116020 |   122 | D   | R    | 75859522 |    1024 |     0 | (null)
   8 | 112 |   1 | 26134 | 182.704877787 |   122 | D   | R    | 75856546 |     768 |     0 | (null)
   8 | 112 |   7 | 27902 | 182.717353880 |  5383 | D   | R    | 75859250 |     256 |     0 | md7_rn5
   8 | 112 |   0 | 27336 | 182.761192610 | 22898 | D   | R    | 75851314 |    1024 |     0 | (null)
   8 | 112 |   6 | 27337 | 182.761208870 | 22898 | D   | R    | 75852338 |     768 |     0 | (null)
   8 | 112 |   1 | 26166 | 182.817485855 |   122 | D   | R    | 75853106 |    1024 |     0 | (null)
   8 | 112 |   1 | 26167 | 182.817506358 |   122 | D   | R    | 75854130 |     768 |     0 | (null)
   8 | 112 |   1 | 26170 | 182.840857363 |   122 | D   | R    | 75854898 |     512 |     0 | (null)
   8 | 112 |   4 | 25653 | 182.842156282 | 22335 | D   | R    | 75855410 |     256 |     0 | (null)
   8 | 112 |   1 | 26200 | 182.848077761 |   122 | D   | R    | 75855666 |     256 |     0 | (null)
   8 | 112 |   2 | 28502 | 182.869246201 |   123 | D   | R    | 75851822 |     768 |     0 | kblockc
   8 | 112 |   2 | 28530 | 182.898320804 |   123 | D   | R    | 75856690 |    1024 |     0 | kblockc
   8 | 112 |   2 | 28531 | 182.898339478 |   123 | D   | R    | 75857714 |     768 |     0 | kblockc
   8 | 112 |   2 | 28569 | 182.923527140 |   123 | D   | R    | 75859506 |     768 |     0 | kblockc
   8 | 112 |   5 | 30410 | 182.964578710 |   124 | D   | R    | 75858482 |    1024 |     0 | (null)
   8 | 112 |   4 | 25665 | 182.988829640 |   125 | D   | R    | 75870274 |     512 |     0 | (null)
   8 | 112 |   2 | 28590 | 183.004848378 |   123 | D   | R    | 75870786 |    1024 |     0 | kblockc
   8 | 112 |   2 | 28600 | 183.004853534 |   123 | D   | R    | 75871810 |     768 |     0 | kblockc
   8 | 112 |   6 | 28589 | 183.057927709 |   127 | D   | R    | 75872578 |    1024 |     0 | (null)
   8 | 112 |   6 | 28596 | 183.087886230 |   127 | D   | R    | 75873602 |     768 |     0 | (null)
   8 | 112 |   6 | 28599 | 183.088685740 |   127 | D   | R    | 75874370 |     256 |     0 | (null)
   8 | 112 |   5 | 24652 | 183.153884315 |   126 | D   | R    | 75870274 |     512 |     0 | kblockc
   8 | 112 |   5 | 24697 | 183.240248577 |   126 | D   | R    | 75874626 |    1024 |     0 | kblockc
   8 | 112 |   5 | 24698 | 183.240245001 |   126 | D   | R    | 75875650 |     768 |     0 | kblockc
   8 | 112 |   4 | 25717 | 183.378241986 |   125 | D   | R    | 75870786 |    1024 |     0 | (null)
   8 | 112 |   4 | 25718 | 183.378318660 |   125 | D   | R    | 75871810 |     512 |     0 | (null)
   8 | 112 |   2 | 28637 | 183.483238183 |   123 | D   | R    | 75876418 |    1024 |     0 | kblockc
   8 | 112 |   2 | 28638 | 183.483238294 |   123 | D   | R    | 75877442 |     768 |     0 | kblockc
(41 rows)
```

So this "almost" proves there are no collisions (at this crossing-point).

Why is this only an "almost proof", and not really a full proof? Because all we've really shown is that there is no collisions amongst the starting sectors (sectstrt), but since I/Os consist of a number of sectors (256/512/768/1024), there may be some collisions amongst those other (non-starting) sectors. To complete our proof, we can reason in a number of ways. Ordinarily, you'd probably want to attack a problem like this via numerics, but we'll do it via graphics, as one last example of the power of graphics in EDA (Exploratory Data Analysis).

Here's the explanation of the reasoning/technique: Consider the above dataset, consisting of 41 datapoints. We know the sectors of each datapoint is a multiple (1, 2, 3, or 4) of 256. Now let's modify that dataset into a new (equivalent but slightly different) dataset, by replicating

each datapoint into multiple new datapoints (1, 2, 3, or 4), each of size exactly 256, adjusting the sectstr accordingly, but keeping the same elapsedtime (because we don't have a good way to guess what the new elapsedtimes "should be"). (For a small dataset like this, this can simply be done via your favorite text editor, starting from the above SQL dataset (and, if your favorite text editor happens to be Emacs, you can even automate the updating of the sectstr values, via Emacs's built-in calculator, "calc").) This results in a new ("SQL-lookalike") dataset, consisting of a total of 1*7 + 2*5 + 3*14 + 4*15 = 119 datapoints. This new dataset can now be graphed to get a new (equivalent but visually different), which is still small enough to deal with graphically/visually, in particular we can check (manually/visually) whether there are the required 119 distinct graphed circles we're looking for. Here's that resulting graph (where we've also made the colored circles a bit smaller, to avoid the mistaken/bogus perception of overlapping datapoints (after all, the "real" dot on the graph is at the center of the circle, not around the rim)) – there are indeed 119 distinct circles:



This completes the "proof" (but see below) that there are no collisions (at this crossing-point). [Full disclosure: Well, OK, all it really "shows" (but, again, see below) is that there are no collisions among the starting sectors of the 256-sector sized "pages", and we're relying on an acceptance of the correctness of the NPS architecture/implementation (not to mention the Blktrace Suite tools themselves, and PostgreSQL, and R, and . . . – see Ken Thompson, Reflections on Trusting Trust, Comm. ACM, Vol. 27, No. 8, August 1984, available at http://cm.bell-labs.com/who/ken/trust.html). For the truly sceptical, who may be concerned that there may be some internal collisions amongst the sectors in those "pages", probably the only convincing proof would need to involve numerics, not just graphics. But then again, for the truly sceptical, only numerics would ever suffice for any proof, not graphics, and so this whole discussion of graphics-based EDA would be moot. There we leave this debate, to be revisited another day.]

Incidentally, it would be surprising if there were ever any collisions at crossing-points in an all-read workload, because that would imply that the Linux block-caching layer wasn't doing its job (the whole idea of block-caching is to avoid excessive/colliding disk reads over short time intervals, such as at crossing-points).

BUT IT TURNS OUT THAT DOES HAPPEN! As will be seen (below), there are indeed some "collisions" – they're just occur differently from the way we were just now trying to detect. (i.e., our "proof" above "proved the wrong thing".) Which just goes to highlight another principle of EDA: check, then double-check again, preferably from multiple points-of-view. If there's anything that "keeps nagging at you", you probably don't understand the full story yet. [For example, in the study we just did concerning collisions, we really should have known a priori that there wouldn't be an collisions amongst datapoints, because the elapsedtime field already disambiguates datapoint, but we wanted to go through this exercise anyway, as an "object lesson". What we really should have been looking for is not "collisions in graphing points", but "collisions of block/sector numbers" – that's what we'll do below.]

Deeper Investigation (Exploratory Data Analysis): Numerics

As an illustration of the kind of numerics we might want to run on the blktrace/blkparse data for a device, let's consider the following question:

To what extent is our device doing "streaming-reads" (a.k.a. "read-streaming", a.k.a. "chained-reads", a.k.a. "read-chaining")? That is, how often does our device do "sequential/linked reads", by which we mean a read operation (rwbs='R') immediately followed by a second read, where the last sector of the first read is "followed-immediately" by the first sector of the second read, without an intervening seek or write operation (noting that both seeks and writes result in a streaming discontinuity [writes do so because the disk head must change state, which takes an appreciable time delta, during which the disk keeps spinning])? (And similarly for read-linked chains of length > 2.) What does it mean for one sector to "follow-immediately" (a.k.a. "be adjacent to") another sector? It means that after the device reads the preceding sector, it can read the following/succeeding sector "immediately", in the sense of incurring the smallest time-delay supported by the device (nominally called "zero" time-delay, by abuse-of-language). What does the "follow-immediately" concept translate to in terms of something we can measure with blktrace? It means that the sector number of the succeeding block is 1 greater than the sector number of the preceding block (because for the disk technology currently used by NPS, the "interleaving ratio" is 1:1).

Why is this question interesting? Well, take a look at the two preceding graphs (the 41-datapoint graph, or the 119-datapoint graph, which are equivalent for our purposes here: that is, numerically, not visually), and consider the two leftmost datapoints (a red/green pair). They seem to be "very close together" (visually), both horizontally and vertically (indeed they almost appear to be aligned vertically). Similarly for the next pair of datapoints on the graphs. And the next pair. And the next pair (though this last is a red/black pair). The pattern ends there (strictly speaking, though it's easy to find other such examples of "very close" pairs scattered throughout the graph). Further, if you look at the distances between datapoints, it looks like whenever the vertical distance between datapoints is bigger (which we call "longer seek distance", because the block/sector numbers are further apart), then the horizontal distance is bigger too (which we call "longer seek time", for obvious reasons). Here, "seek" means "successive device operations of the same type (both reads or both writes) of sectors that are not adjacent". Our interest is piqued. We conjecture that the closest datapoint pairs represent "streaming reads", in the sense just defined.

How can we prove that? Considering the first pair (leftmost) datapoints again, we observe they're represented in the SQL dataset above by the first two (topmost) rows. If we take the first datapoint's starting block/sector number, sectstrt, and add to that the size/length of the read, sectcnt, we get 75865154 + 1024 = 75866178 -- which exactly equals the sectstrt of the second datapoint. I.e., (it appears that) this pair is indeed a read-streaming pair. Similarly for the next pair of datapoints; and the next; and the next.

As above, that "almost" amounts to a proof, but not quite. The reason it's not a full proof is that the SQL dataset above captured only the datapoints with rwbs='R', which leaves us with some uncertainty as to whether there might be some intervening writes (which would indicate a discontinuity in the read-streaming we're looking for). Moreover, we know there aren't any intervening reads/seeks, because the SQL dataset above is read-complete (that is, without any read-gaps), and we can see just by inspecting the dataset that there aren't any reads between the first pair of datapoints -- at least in the range 75,500,000 <= sectstrt <= 76,000,000. But this view-window (rwbs='R', 75,500,000 <= sectstrt <= 76,000,000) isn't the whole truth, at least from the device's point-of-view (which is the only point-of-view that really counts here). So to close this gap in our proof, what we have to do is examine all the data in the time- and sector-range of interest, not just the "rwbs='R' and 75,500,000 <= sectstrt <= 76,000,000" view-window. Here it is:

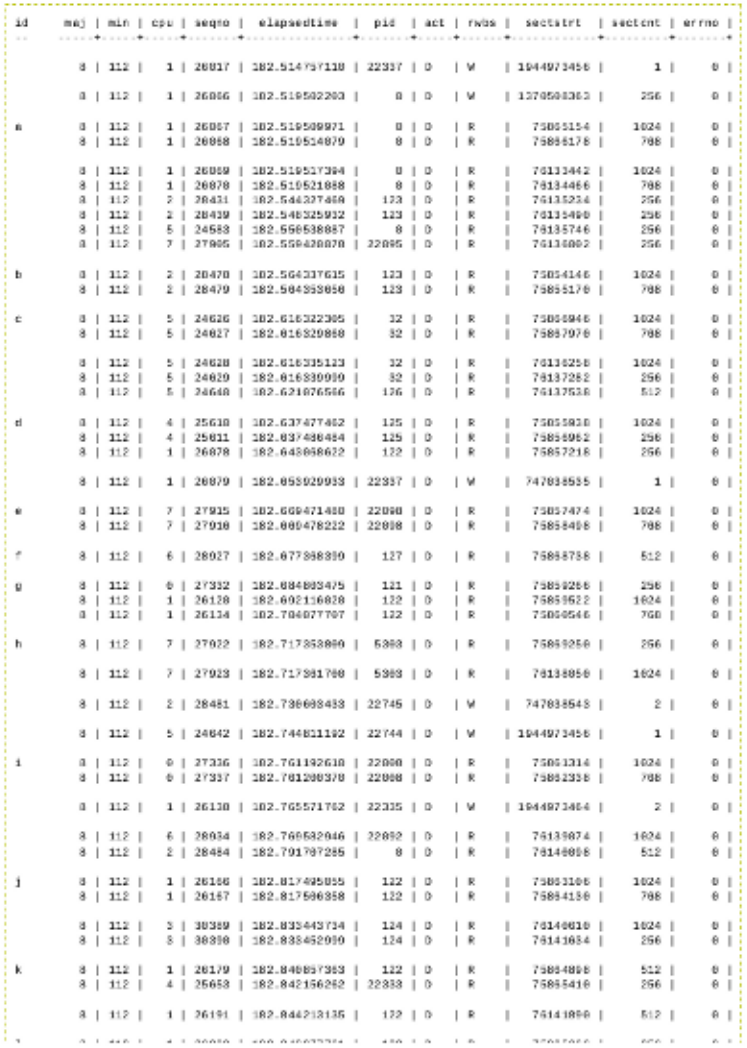
```
blktrace=# select * from blkparse
blktrace-# where act='D'
blktrace-# and (102.5<=elapsedtime and elapsedtime<=103.5)
blktrace-# order by elapsedtime;  -- 86 datapoints
 maj | min | cpu | seqno |  elapsedtime  |  pid  | act | rwbs |  sectstrt  | sectcnt | errno |  comm
-----+-----+-----+-------+---------------+-------+-----+------+------------+---------+-------+--------
   8 | 112 |   1 | 26617 | 102.514757110 | 22337 | D   | W    | 1944975496 |       1 |     0 | (null)
   8 | 112 |   1 | 26666 | 102.519502203 |     0 | D   | W    | 1370588363 |     256 |     0 | swapp
   8 | 112 |   1 | 26667 | 102.519509071 |     0 | D   | R    |   75885154 |    1024 |     0 | swapp
   8 | 112 |   1 | 26668 | 102.519514879 |     0 | D   | R    |   75866178 |     768 |     0 | swapp
   8 | 112 |   1 | 26669 | 102.519517394 |     0 | D   | R    |   76133442 |    1024 |     0 | swapp
   8 | 112 |   1 | 26670 | 102.519521888 |     0 | D   | R    |   76134466 |     768 |     0 | swapp
   8 | 112 |   2 | 28431 | 102.544227469 |   123 | D   | R    |   76135234 |     256 |     0 | kblo
   8 | 112 |   2 | 28439 | 102.548925932 |   123 | D   | R    |   76135490 |     256 |     0 | kblo
   8 | 112 |   5 | 24583 | 102.559638887 |     0 | D   | R    |   76135746 |     256 |     0 | swapp
   8 | 112 |   7 | 27905 | 102.559820078 | 22895 | D   | R    |   76136002 |     256 |     0 | (null
   8 | 112 |   2 | 28478 | 102.564337815 |   123 | D   | R    |   76854146 |    1024 |     0 | kblo
   8 | 112 |   2 | 28479 | 102.564353656 |   123 | D   | R    |   76855170 |     768 |     0 | kblo
   8 | 112 |   5 | 24626 | 102.618822385 |    32 | D   | R    |   76866946 |    1024 |     0 | (null
   8 | 112 |   5 | 24627 | 102.618829868 |    32 | D   | R    |   76867970 |     768 |     0 | (null
   8 | 112 |   5 | 24628 | 102.618835123 |    32 | D   | R    |   76136258 |    1024 |     0 | (null
   8 | 112 |   5 | 24629 | 102.618838999 |    32 | D   | R    |   76137282 |     256 |     0 | (null
   8 | 112 |   5 | 24648 | 102.621876566 |   126 | D   | R    |   76137538 |     512 |     0 | kblo
   8 | 112 |   4 | 25610 | 102.637477462 |   125 | D   | R    |   76855938 |    1024 |     0 | (null
   8 | 112 |   4 | 25611 | 102.637486484 |   125 | D   | R    |   76866962 |     256 |     0 | (null
   8 | 112 |   1 | 26078 | 102.643868622 |   122 | D   | R    |   76857218 |     256 |     0 | (null
   8 | 112 |   1 | 26079 | 102.653929933 | 22337 | D   | W    |  747038535 |       1 |     0 | (null
   8 | 112 |   7 | 27915 | 102.669471400 | 22898 | D   | R    |   76857474 |    1024 |     0 | spu16
   8 | 112 |   7 | 27916 | 102.669478222 | 22898 | D   | R    |   76858498 |     768 |     0 | spu16
   8 | 112 |   6 | 28027 | 102.677368300 |   127 | D   | R    |   76868738 |     512 |     0 | (null
   8 | 112 |   0 | 27332 | 102.684803475 |   121 | D   | R    |   76859266 |     256 |     0 | (null
   8 | 112 |   1 | 26126 | 102.692116820 |   122 | D   | R    |   76859522 |    1024 |     0 | (null
   8 | 112 |   1 | 26134 | 102.704877787 |   122 | D   | R    |   76866546 |     768 |     0 | (null
   8 | 112 |   7 | 27922 | 102.717353809 |  5303 | D   | R    |   76869250 |     256 |     0 | md7_r
   8 | 112 |   7 | 27923 | 102.717361780 |  5303 | D   | R    |   76138658 |    1024 |     0 | md7_r
   8 | 112 |   2 | 28481 | 102.736683433 | 22745 | D   | W    |  747038543 |       3 |     0 | (null
   8 | 112 |   5 | 24642 | 102.744811102 | 22744 | D   | W    | 1944975456 |       1 |     0 | (null
   8 | 112 |   0 | 27336 | 102.761182610 | 22898 | D   | R    |   76861214 |    1024 |     0 | (null
   8 | 112 |   0 | 27337 | 102.761280970 | 22898 | D   | R    |   76862238 |     768 |     0 | (null
   8 | 112 |   1 | 26138 | 102.765571702 | 22335 | D   | W    | 1944975464 |       2 |     0 | (null
   8 | 112 |   6 | 28034 | 102.769503946 | 22892 | D   | R    |   76139074 |    1024 |     0 | spu16
   8 | 112 |   2 | 28484 | 102.791787285 |     0 | D   | R    |   76140098 |     512 |     0 | swapp
   8 | 112 |   1 | 26166 | 102.817405855 |   122 | D   | R    |   75863186 |    1024 |     0 | (null
   8 | 112 |   1 | 26167 | 102.817506350 |   122 | D   | R    |   75864130 |     768 |     0 | (null
   8 | 112 |   3 | 30380 | 102.833443734 |   124 | D   | R    |   76140610 |    1024 |     0 | (null
   8 | 112 |   3 | 30390 | 102.833453000 |   124 | D   | R    |   76141634 |     256 |     0 | (null
   8 | 112 |   1 | 26170 | 102.840857308 |   122 | D   | R    |   75884898 |     512 |     0 | (null
   8 | 112 |   4 | 25653 | 102.842156262 | 22333 | D   | R    |   75865410 |     256 |     0 | (null
   8 | 112 |   1 | 26191 | 102.844213135 |   122 | D   | R    |   76141890 |     512 |     0 | (null
   8 | 112 |   1 | 26200 | 102.848877781 |   122 | D   | R    |   75885666 |     256 |     0 | (null
   8 | 112 |   2 | 28503 | 102.869246201 |   123 | D   | R    |   75865922 |     768 |     0 | kblo
   8 | 112 |   2 | 28530 | 102.890328004 |   123 | D   | R    |   75886690 |    1024 |     0 | kblo
   8 | 112 |   2 | 28531 | 102.894889478 |   123 | D   | R    |   75867714 |     768 |     0 | kblo
   8 | 112 |   2 | 28569 | 102.923227140 |   123 | D   | R    |   75869506 |     768 |     0 | kblo
   8 | 112 |   2 | 28570 | 102.928833944 |   123 | D   | R    |   76142482 |    1024 |     0 | kblo
   8 | 112 |   2 | 28571 | 102.928837708 |   123 | D   | R    |   76143426 |     768 |     0 | kblo
   8 | 112 |   5 | 30410 | 102.964578710 |   124 | D   | R    |   75886482 |    1024 |     0 | (null
   8 | 112 |   6 | 28960 | 102.973325873 |   127 | D   | R    |   76144194 |    1024 |     0 | (null
   8 | 112 |   6 | 28961 | 102.973332157 |   127 | D   | R    |   76145218 |     768 |     0 | (null
   8 | 112 |   4 | 25605 | 102.988820040 |   125 | D   | R    |   75870274 |     512 |     0 | (null
   8 | 112 |   2 | 28599 | 103.004348373 |   123 | D   | R    |   75870786 |    1024 |     0 | kblo
   8 | 112 |   2 | 28600 | 103.004355534 |   123 | D   | R    |   75871810 |     768 |     0 | kblo
   8 | 112 |   4 | 25606 | 103.012820380 |   125 | D   | R    |   76145506 |    1024 |     0 | (null
   8 | 112 |   4 | 25601 | 103.012836696 |   125 | D   | R    |   76147010 |     512 |     0 | (null
   8 | 112 |   6 | 28989 | 103.037327709 |   127 | D   | R    |   75872578 |    1024 |     0 | (null
   8 | 112 |   6 | 28990 | 103.037336230 |   127 | D   | R    |   75873602 |     768 |     0 | (null
   8 | 112 |   0 | 27371 | 103.044897981 |   121 | D   | R    |   76147522 |     768 |     0 | (null
   8 | 112 |   6 | 28998 | 103.061685740 |   127 | D   | R    |   75874370 |     256 |     0 | (null
   8 | 112 |   3 | 30439 | 103.084577579 |   124 | D   | R    |   76148290 |    1024 |     0 | (null
```

There are now 86 datapoints in this dataset: 41 of them are exactly the rwbs='R' rows of the preceding SQL dataset; and the remaining 45 are new datapoints not represented in the 41-datapoint graph above: 14 of them are rwbs='W' rows, and the other 31 are rwbs='R' rows outside of the "75,500,000 <= sectstr <= 76,000,000" range (you can easily see where these additional rwbs='R' rows come from by looking at the 30-sec, 2150-datapoint graph, above).

Now, painstakingly/laboriously examining/comparing the preceding 2 SQL datasets, what we'll now do is take our preceding 41-datapoint graph, and mark it up to highlight the read-streams we're looking for. Here's how we'll do it:

- In the 41-datapoint dataset above, find the consecutive rows that "appear" to be read-streams -- that is, chains of datapoints such that a row's sectstr+sectcnt equals the next row's sectstr. Let's call these our "purported" read-chains.
- But then, look at the corresponding 86-datapoint dataset, and check to see whether or not there's really an intervening seek or write in each of those purported read-chains. If there is, then the purported read-stream we found in the preceding bullet-item is deceiving. In those deceiving cases, break up the 86-datapoint dataset into the pieces that really are true read-streams (not just "purported" ones).

This process (the preceding 2 bullet-items), properly annotated (as explained below the following dataset), yields this "annotated" (SQL-like) dataset:

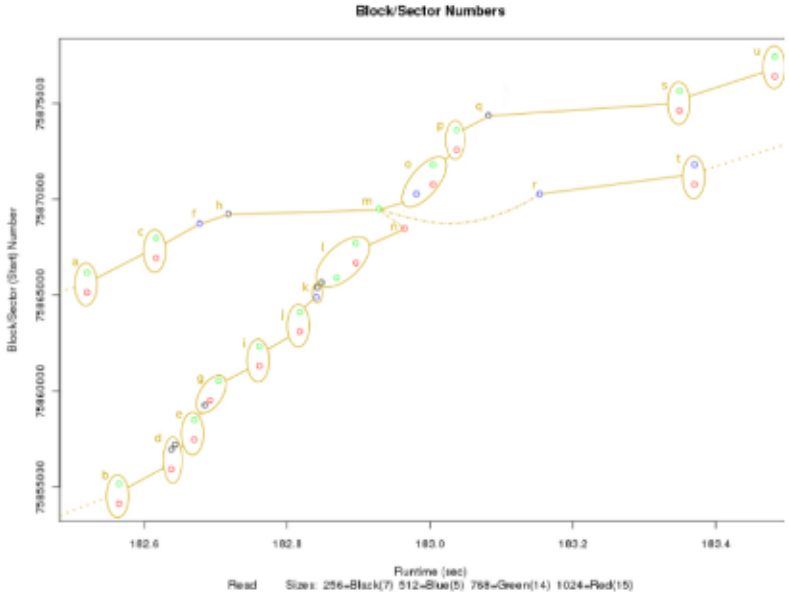


id	maj	min	cpu	seqno	elapsedtime	pid	act	rwbs	sectstrt	sectcnt	errno
	8	112	1	26017	182.514797110	22937	D	W	1944973456	1	0
	8	112	1	26066	182.519502203	0	D	W	1370508363	256	0
a	8	112	1	26067	182.519509971	0	D	R	75855154	1024	0
	8	112	1	26068	182.519514079	0	D	R	75856178	768	0
	8	112	1	26069	182.519517304	0	D	R	76133442	1024	0
	8	112	1	26070	182.519521088	0	D	R	76134466	768	0
	8	112	2	28431	182.544207480	123	D	R	76135234	256	0
	8	112	2	28439	182.548325932	123	D	R	76135490	256	0
	8	112	5	24583	182.550838887	0	D	R	76135746	256	0
	8	112	7	27905	182.559420870	22095	D	R	76136002	256	0
b	8	112	2	28470	182.564337615	123	D	R	75854146	1024	0
	8	112	2	28479	182.564853858	123	D	R	75855170	768	0
c	8	112	5	24626	182.616322305	32	D	R	75856946	1024	0
	8	112	5	24627	182.616329868	32	D	R	75857970	768	0
	8	112	5	24628	182.616335123	32	D	R	76136258	1024	0
	8	112	5	24629	182.616339919	32	D	R	76137282	256	0
	8	112	5	24648	182.621076566	136	D	R	76137538	512	0
d	8	112	4	25610	182.637477402	125	D	R	75855010	1024	0
	8	112	4	25611	182.637480484	125	D	R	75856962	256	0
	8	112	1	26078	182.643048622	122	D	R	75857218	256	0
	8	112	1	26079	182.655929933	22937	D	W	747038535	1	0
e	8	112	7	27915	182.669471480	22098	D	R	75857474	1024	0
	8	112	7	27918	182.669478222	22098	D	R	75858498	768	0
f	8	112	6	28027	182.677888830	127	D	R	75858738	512	0
g	8	112	0	27332	182.684803475	121	D	R	75859250	256	0
	8	112	1	26128	182.692116038	122	D	R	75859522	1024	0
	8	112	1	26134	182.704077707	122	D	R	75860546	768	0
h	8	112	7	27922	182.717353880	5388	D	R	75859250	256	0
	8	112	7	27923	182.717981780	5388	D	R	76138050	1024	0
	8	112	2	28481	182.736008438	22745	D	W	747038543	2	0
	8	112	5	24642	182.744811102	22744	D	W	1944973456	1	0
i	8	112	0	27336	182.761192610	22098	D	R	75861314	1024	0
	8	112	0	27337	182.761200370	22098	D	R	75862338	768	0
	8	112	1	26138	182.765571702	22335	D	W	1944973464	2	0
	8	112	6	28034	182.766582046	22092	D	R	76139074	1024	0
	8	112	2	28484	182.791797285	0	D	R	76140098	512	0
j	8	112	1	26166	182.817495055	122	D	R	75863106	1024	0
	8	112	1	26167	182.817500358	122	D	R	75864130	768	0
	8	112	5	30389	182.833443734	124	D	R	76140610	1024	0
	8	112	5	30390	182.833452090	124	D	R	76141634	256	0
k	8	112	1	26179	182.840897363	122	D	R	75864898	512	0
	8	112	4	25653	182.842156262	22333	D	R	75865410	256	0
	8	112	1	26191	182.844218135	122	D	R	76141890	512	0

Specifically, the above "annotated" (SQL-like) dataset is constructed/annotated as follows:

- Start with the 86-datapoint dataset, above.
- Hand-add the blank lines (in Emacs), in such a way that each block of rows (between the blank lines) is a "stream-block", in the sense that "this.sectnlt + this.sectcnt = next.sectstrt", and each block consists of all-reads or all-writes. That is: each blank line represents a discontinuity/"seek" between read/write-streams/chains.
- Hand-add the "id" column on the left in Emacs, and tag/ID (using the letters 'a'..'u') each of the (read-)stream-blocks that represents a (read-)stream which lies within the parameters of our 41-datapoint dataset above (i.e., rwbs='R' and 75,500,000 <= sectstrt <= 76,000,000).

Now, we can create an "annotated" graph from the above "annotated" dataset, as follows (see explanation below the graph):



Here's how the above "annotated" graph is constructed:

- Start from our 41-datapoint graph, above.
- Draw an ellipse enclosing each of the (true-)read-streams, according to the tags/IDs in the above annotated dataset, and label each ellipse with its corresponding tag/ID. (Don't bother drawing the ellipse around the "solitaries"/singletons.) Notice that there are some read-streams out-of-bounds of our 41-datapoint graph (i.e., beyond the limits 75,500,000 <= sectstrt <= 76,000,000), so those can't be ellipsified here. Similarly, there are some write-streams that can't be ellipsified here.
- Using solid-lines, connect those ellipses which combine to form "logical" read-streams – i.e., such that "thisEllipse.lastDatapoint.sectstrt + thisEllipse.lastDatapoint.sectcnt = nextEllipse.firstDatapoint.sectstrt". So really, the solid-lines indicate a connection between the "last" (upper-right) datapoint in one ellipse to the "first" (lower-left) datapoint in the next ellipse.
- Note the ellipses 'k' and 'l' are logically connected with such a solid-line, but it has length 0 (the ellipses are tangent). If you look at

the annotated 86-datapoint dataset above, you'll see that 'k' and 'l' really are distinct read-streams, because there's an intervening write between them ("at the point of tangency", as to speak).

- Also, add "evocative" dotted-lines at the left and right edges of the graph, for beautification/ambience purposes.
- Using a dot-dash-line, connect the datapoints 'n' and 'm'. (Explanation below.)
- Using a dot-dash-Bezier-curve, connect the datapoints 'm' and 'r'. (Explanation below.)

Here's the explanation about what this graph is telling us (particularly, about the exact details of this crossing-point):

- Each ellipse represents a "physical (read-)stream"; that is, the datapoints inside the ellipse represent consecutive block/sector numbers (in "positive-slope order", i.e., lower-left to upper-right), and these sectors are actually read sequentially from the device.
- Each solid-line (between ellipses and/or solitaries) represents a "logical (read-)stream"; that is, the dots along the combined ellipse/solid-line path represent consecutive block/sector numbers, but the solid-line indicates a gap/discontinuity in the (read-)stream – i.e., the reads of the 2 sectors/blocks connected by the solid-line are not read sequentially from the device, rather there's an intervening disk-head operation (seek, or read or write at another location).
- At the point 'n', the "next logical block/sector" "should be" the point 'm'. But 'm' has already been read, as a consequence of the .../l/h/m/... logical-stream. So there's no need to read 'm' as part of the .../k/l/n logical-stream. That's why we denote the n/m link with a dot-dash-line – it indicates caching!
- The point 'r' "looks like" it appears out of thin air. But that's not quite true. The thing to notice is that it "almost logically" follows the point 'm', because 75869506 (= 'm') + 788 = 75870274 (= 'r').
- In particular, it "looks like" (see below) the 2 "SQL sessions" are graphically represented by the 2 block/sector-number-streams: .../a/c/f/h/m/r/... and .../b/d/e/g/i/j/k/l/n[/m]/o/p/q/s/u/... Along these lines too, observe that the different amounts/scales of device activity along these 2 streams/sessions is consistent with the different slopes of the ellipse/solid-line paths (faster I/O throughput = steeper slope).

BTW, the reason we said "looks like" just above is that we don't "really know for sure". Because, we're seeing things here at the device layer, after the Linux block-I/O scheduler has gotten into the act – not at the application/NPS layer, which is where we'd have to look to "really know for sure" (because we'd have to know the actual content of these blocks/sectors). The most intriguing datapoint for this question is of course 'm' – which stream/session does it belong to? We don't "really know for sure" (it just looks "more likely" to me that 'm' belongs to the shorter stream/session). All we do "really know for sure" is that the block-I/O scheduler figured out 'm' needed to be scheduled "around this time" for peak I/O efficiency (to make both streams/sessions happy).

But it's here that we uncover the biggest puzzle/surprise found during this study: There ARE some blocks/sectors in common between these two streams (i.e., data shared between the 2 SQL sessions)! AND, that data is read twice! In other words: WE ARE SEEING SOME "COLLISIONS", DESPITE EXPECTATIONS (why isn't Linux caching working as we expect?)! For a check of the above annotated 81-datapoint dataset shows that the 3 datapoints in 'o' are the same as the 3 datapoints in 'r'+'t' (except that the sectcnt differs for one of those 3 datapoints). <<TBD – What's happening here? A bug in Linux block caching, or in the Linux block-I/O scheduler? A race condition (say, multiple CPUs with multiple block-I/O scheduler queues)? Does the Linux layer make an assumption that such "collisions" will be taken care of at the device controller layer (queue and/or caching)? Contact Linux block-I/O scheduler maintainer, and blktrace author, Jens Axboe, about this?>>

Be that as it may, in any case this annotated graph confirms that the 4 "very close" (read-)pairs discussed above do indeed represent the read-streaming behavior we're looking for (they're in the ellipses 'a', 'b', 'c', 'd' in our annotated 41-datapoint graph). Consequently, the gaps between those read-streaming pairs (that is, between the first pair and the second pair, and the second pair and third pair, etc.) represent "discontinuities" – whose exact nature can be determined by examining the annotated 86-datapoint dataset above.

So far, so good. But so far, this analysis is still more graphical than numerical in nature. To make it more numerical, we have to do some programming. We'll use a combination of SQL and Python here, by personal preference, though of course there are many other ways to "skin this cat", so you should feel free to choose any way you're comfortable with.

For this exercise we're interested exactly in all the I/O requests that are actually issued to the device – not in any other activities/events such as the completion of those requests, or I/O-block scheduler queue-management activities, or any of the other 'act'activities (you should review those activities, as documented in the blktrace.pdf documentation, to convince yourself of this statement). This motivates us to embark upon our numerical journey by first generating into a file the list of all act='D' datapoints:

```
blktrace-A -- CSV preamble as above, for /tmp/foo
blktrace=# select elapsedtime,rwbs,sectstrt,sectcnt from blkparse
blktrace-# where act='D'
blktrace-# order by elapsedtime;  -- 131,768 datapoints
```

Now, it "just so happens" (well, it's a fact/rule/law, as you can deduce by studying the definitions of the 'act' and 'rwbs' fields in blktrace.pdf) that this list consists entirely of reads (act='R') and writes (act='W'):

```
blktrace=# select count(*) from blkparse
blktrace-# where act='D';
131768
blktrace=# select count(*) from blkparse
blktrace-# where act='D' and rwbs='R';
123495
blktrace=# select count(*) from blkparse
blktrace-# where act='D' and rwbs<>'R';
8273
blktrace=# select count(*) from blkparse
blktrace-# where act='D' and rwbs='W';
8273
```

What that means is that we now know our Python programming job will be "easy" (or, at least, that certain potential complexities have been averted).

But before diving straight into Python, perhaps we should take a moment to do a "sanity-check" on our dataset (the one generated above, with 131,768 datapoints), that is, try to convince ourselves that our dataset "makes sense". (This is generally a good policy/practice for the programming craftsman at every stage of development, because it's easier to debug baby-steps than giant-leaps.) In the case at hand, the simplest check we can do is visually inspect our dataset (in a text editor, which for me is Emacs). When we do that, we're fortunate to spot an anomaly right near the top. Here are the first 30 lines of our file (/tmp/foo):

```
0.000180887|W|1944973456|1
0.020600783|W|2829122|256
0.020625501|W|1200086001|256
0.022740744|R|75790874|1024
0.022758281|R|75800896|256
0.022765687|R|75801154|512
0.022776133|R|76377154|768
0.024330418|R|76377922|1024
0.024330672|R|76527426|1024
0.024345297|R|76528450|768
0.024370819|W|2831170|256
0.187680655|W|747988535|1
0.121660202|W|1944973456|1
0.148817999|W|35651856|256
0.169588848|W|1944955271|1
0.169511755|R|0|0
0.191685093|R|75891856|1024
0.191673739|R|75892608|768
0.191678737|R|76378046|1024
0.191683866|R|76378870|768
0.191688317|R|76529218|1024
0.191693697|R|76530242|768
0.191697520|W|35654976|256
0.295975910|W|1944955272|1
0.296081419|R|0|0
0.368890267|W|747988535|1
0.387857841|W|1024417794|256
0.388185247|R|75893458|1024
0.388196573|R|75894482|768
```

The anomaly lies, of course, in the lines "...|R|0|0". Why are some reads (or writes, for that matter) doing "no work" (i.e., touching 0 sectors) being issued to the device? And, why are those no-op operations being done at sector 0? Recall that sector 0 is the MBR (Master Boot Record), which isn't a normal data sector in the operating system sense (it doesn't belong to any disk partition). Maybe this has something to do with keeping the device "happy" for some reason, such as the device cabinet being bumped and "parking" the head for a moment at sector 0 (probably not – sector 0 is at the outer edge of the disk, but the parking/landing zone is towards the center of the disk; and anyway, NPS machines are supposed to be sitting on stable machine-room floors, where no impact should be happening [I haven't noticed any earthquakes around here lately, have you?]). Anyway, what does the device do with such a request? Does it honor the request by seeking to the specified sector and doing nothing? Does it simply ignore the request (in which case we could delete those lines from our dataset)? Or does the request have some special meaning for the device (such as entering a suspend/unloaded position, sort-of the opposite of parking – which at least makes some kind of sense, since the unload ramps are at the outside edge of the disk)?

Let's examine this anomalous dataset to see if can find a clue.

```
blktrace=# select * from blkparse
blktrace-# where act='D' and sectcnt=8
blktrace-# order by elapsedtime;
 maj | min | cpu | seqno |   elapsedtime   |  pid  | act | rwbs | sectstrt | sectcnt | errno | comma
-----+-----+-----+-------+-----------------+-------+-----+------+----------+---------+-------+------
   8 | 112 |   2 |    96 |   0.108511755 |     0 | D   | R    |        0 |       8 |     0 | swappe
   8 | 112 |   2 |   188 |   0.298981419 | 18957 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   0 | 16322 | 128.143268014 | 22111 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   0 | 16336 | 128.281709496 | 22111 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   2 | 16668 | 256.130531704 | 23558 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   2 | 16674 | 256.197415804 | 23558 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   1 | 78205 | 384.182715016 | 25449 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   1 | 78327 | 384.318394957 | 25449 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   1 | 128826 | 512.179271995 | 27984 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   1 | 128838 | 512.236380818 | 27984 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   7 | 165206 | 640.258018688 | 28325 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   0 | 129874 | 640.429244268 | 28161 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   5 | 129450 | 768.283773755 | 29769 | D   | R    |        0 |       8 |     0 | (null)
   8 | 112 |   2 | 164304 | 768.385831988 | 29768 | D   | R    |        0 |       8 |     0 | (null)
(14 rows)
```

Aha, there's our clue staring us right in the face: the anomalous behavior is happening very regularly (periodically), every ~128 seconds, in pairs, always with the same per-pair data in the info field, though the pairs are not always from the same CPU or PID. Could this be some sort of system thing, such as S.M.A.R.T.? Furthermore, the anomalous behavior "just happens" to start at elapsedtime=~0. That's very suspicious; could it be triggered by blktrace itself for some reason? And besides, who measures time in binary?

You're probably wondering what would happen if you ran the preceding SQL query, but with "sectstrt=0" instead of "sectcnt=0". The answer is that you'd get the same dataset of 14 datapoints. On the other hand, examining the datapoints with act<>'D' and sectstrt=0 (or sectcnt=0) doesn't tell us anything (both of which return 206,459), because blkparse uses "sectstrt=0,sectcnt=0" as a default/placeholder, for activities that don't actually go to the device, such as block-I/O scheduling activities dealing with queue maintenance.

At this point, the trail starts running cold. I did a Google search for "read 0 bytes sector 0", but didn't find any good leads in the first few pages of results, so I decided to give up. A major factor in "giving up" is that there are only 14 of the anomalous datapoints in any case, and inspecting all of them (in /tmp/foo in Emacs) reveals that only 4 stream-read discontinuity counts in our proposed Python program would be affected by the anomaly (they're the ones at positions 6/7/9/11, starting at 1). This is hardly enough to affect our conclusions significantly in any case, but since it'll be easy enough to run the test twice, once with the anomalies included and once with them excluded (via a switch in the Python script), I decided that's what I'd do.

<<TBD – WHO KNOWS WHAT READS OF sectstrt=0/cnt=0 REALLY MEAN? SEND EMAIL TO blktrace/blkparse/btt maintainers.>>

With the information we now have at hand, it's time to write our Python analysis script. It's included in the perfbin.tgz tarball, under the name readWriteStream.py. (It contains a flag, PURGE_R_0_0, near the top which determines whether or not the "...[R0]0" lines are included or excluded; the default is to exclude them (because that's what btt seems to do). The user can change this flag by changing its value inline in the script.)

As usual with a programming task, we begin by "sanity-checking" the program on a small dataset. For that purpose, we've got one ready-made: the 88-datapoint dataset we exhaustively studied above (by annotating it, and studying its annotated graph). We proceed as follows:

```
blktrace=# -- CSV preamble as above, for /tmp/foo
blktrace=# select elapsedtime,rwbs,sectstrt,sectcnt from blkparse
blktrace-# where actn='D'
blktrace-# and (182.5<=elapsedtime and elapsedtime<=183.5)
blktrace-# order by elapsedtime;  -- 86 datapoints

$- cat /tmp/foo  # our 86-datapoint dataset, slimmed down to the 4 columns needed by readWriteStream
182.514757110|W|1944973456|1
182.519582283|W|1370588363|256
182.519589971|R|75865454|1024
182.519514079|R|75866178|768
182.519517304|R|78138442|1024
182.519521008|R|78134466|768
182.544327409|R|78135234|256
182.548825382|R|78135490|256
182.550530807|R|78135746|256
182.559420978|R|78136002|256
182.564237615|R|75854146|1024
182.564353856|R|75855170|768
182.616322385|R|75866946|1024
182.616329360|R|75867970|768
182.618385123|R|78136258|1024
182.616339909|R|78137282|256
182.621076368|R|78137538|512
182.65747702|R|75855938|1024
182.637486484|R|75856962|256
182.643068622|R|75857218|256
182.653029931|W|747018535|1
182.669471480|R|75852474|1024
182.669478222|R|75868498|768
182.677268399|R|75868738|512
182.684883475|R|75850286|256
182.692116020|R|75869522|1024
182.704077707|R|75868546|768
182.717853380|R|75801250|256
182.717361700|R|78138050|1024
182.738603483|W|747018543|2
182.744011102|W|1944973456|1
182.761192610|R|75801314|1024
182.761200370|R|75862338|768
182.765571762|W|1944973464|2
182.780582346|R|78139074|1024
182.781767285|R|78140098|512
182.817405055|R|75861186|1024
182.817508358|R|75804130|768
182.833443734|R|78140610|1024
182.833452300|R|78141634|256
182.840857363|R|75864898|512
182.842156282|R|75805410|256
182.844213185|R|78141890|512
182.849977761|R|75865666|256
182.869246281|R|75805922|768
182.869329864|R|75866690|1024
182.896229473|R|75867714|768
182.928527148|R|75809506|768
182.928833344|R|78142402|1024
182.928887703|R|78143426|768
182.964579710|R|75868482|1024
182.973525973|R|78144194|1024
182.973882157|R|78145218|768
182.980029640|R|75870274|512
183.004348373|R|75870786|1024
183.004853584|R|75871810|768
183.012829389|R|78145986|1024
183.012898896|R|78147010|512
183.037827709|R|75872578|1024
183.057539239|R|75873602|768
183.040007981|R|78147522|768
183.061885740|R|75874370|256
```

Here's the explanation for readWriteStream.py's 5 blocks of output:

The first 3 blocks of output report frequency information for the (sizes of) "reads" and "writes", that is, all operations (both rwbs='R' and rwbs='W', of any size) actually issued to our device. For this example, it shows that 72 reads were issued, of which none were of size 0 (i.e., sectors=0), 16 were of size 256, etc. The percentage figures are based of the total number of 72 reads.

The 4th block deals with reads. The first line, readsUnStreamed, shows how many (and what percentage) of reads are "unstreamed" – a.k.a., "length-1 read-streams"; that is, "solitaries"/singletons, which are not "adjacent" to any other read (in the sense defined above). The remainder of the first line is just like the "length>=2" lines, which we now discuss. The rest of the read lines deal with streaming-reads ("length>=2") – which is what we're interested in. The second line, "readStreamLen 2", show how many reads occur in read-streams of length (exactly) 2. Namely, in this example, there are 16 such length-2-read-chains, accounting for a total of 2*16 = 32 read operations. The percentage figure is simply the percentage of all reads that occur in length-2 read-streams, that is, 32/72 = 44.444%. On the right of the percentage figure, the pairs of numbers (separated by "->") record the elapsedtimes (which uniquely identify the datapoints themselves, of course) corresponding to the initial and final reads in these (length-2) read-streams (earliest 10 such, at most); this information is given so that you can look up the read-streams visually (on any graph that contains both endpoints) (you can also inspect the dataset file, /tmp/iso, for these endpoints if in Emacs if you wish). The remaining lines ("length>=3") are for the other "readStreamLen N" lines, N >= 3. (Lines for length-N read-streams which occur with frequency 0 are omitted.) The final line is simply a roll-up report for the sum-totals for streamed-reads: there are a total of 24 read-streams of length > 1 ("non-solitaries"), comprising a total of 63 reads. Naturally, the sum of readsUnStreamed=9 and allReadsStreamed=63 is allReads=72.

The 5th block deals with writes. It's just like the 4th block, mutatis mutandis.

Now that we know what kind of output readWriteStream.py gives us, we can validate that readWriteStream.py is indeed telling us the truth, simply by comparing the above readWriteStream.py output with the preceding annotated 86-datapoint dataset. We leave that as an (easy) exercise for the reader.

OK, so this convinces us that the readWriteStream.py numerics are indeed telling us the truth.

To summarize: All this brings us face-to-face with the Great Duality of Exploratory Data Analysis: the "graphics" (e.g., the preceding annotated 86-datapoint graph) are what our intuitive "right brain" believes, but it's the "numerics" (e.g., the output of readWriteStream.py) that tell the "left-brain" truth. So which should we believe? Both: The graphics and numerics "play off" of one another, each suggesting further exploration with the other, both telling "different aspects of the same story". If they're incompatible with one another, you don't yet understand the story they're trying to tell you. (Or, "the cross-tie 'tension of yin/yang'", if you're philosophically inclined.)

So now let's return to our quest: running readStream.py on our full dataset:

```
blktrace=# -- CSV preamble as above, for /tmp/foo
blktrace=# select elapsedtime,rwbs,sectstrt,sectcnt from blkparse
blktrace-# where actn='D'
blktrace-# order by elapsedtime;  -- 131,788 datapoints

$- readWriteStream.py /tmp/foo
readSiz          0 :        0     0.000%
readSiz        256 :    68365    55.365%
readSiz        512 :    22568    18.276%
readSiz        768 :    14307    11.586%
readSiz       1024 :    18249    14.775%
allReads           :   123481

writeSiz         1 :     3404    41.148%
writeSiz         2 :     2402    38.122%
writeSiz       256 :     2126    25.698%
writeSiz       512 :      218     2.636%
writeSiz       768 :       35     0.423%
writeSiz      1024 :        8     0.097%
allWrites          :     8273

allReadsWrites     :   131754

readsUnStreamed  1 :    26378  -->    26378   21.362%    0.731493244->0.731493244 0.731493244->0.7
readStreamLen    2 :    15084  -->    30008   24.302%    0.822778133->0.824386418 0.824389872->0.8
readStreamLen    3 :     3848  -->    11328    9.172%    0.822748744->0.827763537 0.336183934->0.3
readStreamLen    4 :     1685  -->     6740    5.458%    1.345523488->3.355550945 7.171141395->7.1
readStreamLen    5 :      982  -->     4910    3.976%    2.073085135->2.083059740 2.287412548->2.2
readStreamLen    6 :      854  -->     5024    3.788%    1.276583243->1.293826452 11.747847881->11
readStreamLen    7 :      453  -->     3171    2.568%    1.40119731->1.444576147 10.544769486->10.
readStreamLen    8 :      317  -->     2536    2.054%    2.092372989->2.111804353 14.977361832->15.
readStreamLen    9 :      209  -->     1881    1.523%    0.818721041->0.848467502 7.178832293->7.1
readStreamLen   10 :      195  -->     1950    1.579%    17.767502915->17.792077616 40.041382191->4
readStreamLen   11 :      153  -->     1683    1.363%    36.184069857->36.229503721 50.795120472->5
readStreamLen   12 :       98  -->     1176    0.875%    34.072084826->34.114899583 42.503577381->4
readStreamLen   13 :       71  -->      923    0.747%    17.527765428->17.56467722 38.522736116->3
readStreamLen   14 :       46  -->      644    0.522%    17.6128755->17.652876840 44.784832066->44
readStreamLen   15 :       30  -->      450    0.364%    50.464316746->50.549092831 55.440329540->5
readStreamLen   16 :       21  -->      336    0.272%    46.128327884->46.252328541 62.840976875->6
readStreamLen   17 :       20  -->      340    0.275%    31.384383702->31.444575654 40.473876450->4
readStreamLen   18 :       12  -->      216    0.175%    46.080388686->46.376466636 57.085076697->5
readStreamLen   19 :       19  -->      361    0.292%    63.852089611->63.710991861 88.847825758->8
readStreamLen   20 :       10  -->      200    0.162%    97.276882772->97.412594438 100.240876520->1
readStreamLen   21 :       16  -->      336    0.272%    85.499019418->85.705076116 98.102229466->9
readStreamLen   22 :       13  -->      286    0.232%    84.149325484->84.251862847 125.588326521->1
readStreamLen   23 :       12  -->      276    0.224%    96.744827713->96.877335847 98.357389954->9
readStreamLen   24 :        9  -->      216    0.175%    57.744575976->57.844041139 62.856483688->6
readStreamLen   25 :        4  -->      100    0.081%    84.457585124->84.590193734 225.401075077->2
readStreamLen   26 :       10  -->      260    0.211%    98.486212934->98.584480849 115.929584281->1
readStreamLen   27 :        6  -->      162    0.131%    70.268274488->70.376588248 88.503085448->8
readStreamLen   28 :        7  -->      196    0.159%    61.557329447->61.768708582 100.663155674->1
readStreamLen   29 :        9  -->      261    0.211%    58.889331867->58.993528467 85.381576587->8
readStreamLen   30 :        2  -->       60    0.049%    116.48283778->116.68109885 204.554828850->2
readStreamLen   31 :        7  -->      217    0.176%    65.864907042->66.068239214 70.605577781->7
readStreamLen   32 :       10  -->      320    0.259%    81.380078725->81.412189286 82.831382784->8
readStreamLen   33 :        6  -->      198    0.160%    85.83220116->85.184326587 97.746307212->9
readStreamLen   34 :        2  -->       68    0.055%    149.718255782->149.888853396 347.94882087
readStreamLen   35 :        8  -->      280    0.227%    46.376437663->46.507826487 61.017575592->6
readStreamLen   36 :        6  -->      216    0.175%    95.112930732->95.302861266 121.452877087->
readStreamLen   37 :        2  -->       74    0.060%    117.484828815->117.576579278 202.78588802
readStreamLen   38 :        7  -->      266    0.215%    65.555004481->65.700859008 99.241207031->9
readStreamLen   39 :        6  -->      234    0.190%    70.513351991->70.647352924 124.809077229->
readStreamLen   40 :        3  -->      120    0.097%    114.2445760->114.437588086 128.444708096->1
readStreamLen   41 :        8  -->      328    0.266%    50.587724836->50.768944819 102.192880076->
readStreamLen   42 :        2  -->       84    0.068%    90.765390731->91.037325977 197.088575995->
readStreamLen   43 :        5  -->      215    0.174%    225.197825758->225.360575112 247.581305648
readStreamLen   44 :        7  -->      308    0.249%    54.271328885->54.555121449 87.815695419->8
readStreamLen   45 :        3  -->      135    0.109%    58.162561917->58.364585237 344.009688984->
readStreamLen   46 :        2  -->       92    0.075%    296.269030081->296.446749095 414.197326079
```

Note that the "summary" values here for the number of read-operations and write-operations sent to the device (allReads=123,481, allWrites=8,273) are consistent with those in the summary file (sdh blkparse.txt.summary) discussed above: (123,481 , 8,273). (Recall that we're excluding the 14 values for "...|R|0|0" here (we could have included them if wanted to, by setting PURGE_R_0_0=True in readWriteStream.py as noted above, in which case we'd now have allReads=123,495).) Note also that the number of length-1 read-chains (26,378) plus the number of length>1 read-chains (97,103) equals 123,481 – giving us an additional consistency ("sanity") check. Similarly for the writes (8,168 + 105 = 8,273.)

The sum of the percentages for the read-streams of lengths 1..10 is 21.362 + 24.302 + ... + 1.579 = 75.572%. So, whatever else is happening, we're justified in saying that "most read-chains are short (<= 10)", at least in this sample.

As mentioned above, in NPS all reads are done on the basis of "3MB extents", which consist of "24 consecutive pages", each page being 128 KB = 256 blocks/sectors – so every NPS read results in 24 256-block/sector read operations sent to the device. That is, "6,144 blocks/sectors-in-chunks", which we write as "24x256" – though with merging at the Linux block-I/O scheduler layer, this could become "12x512", "8x768", "8x1024", or a combination thereof as issued to the device). It has sometimes been asserted that "NPS engineers generally believe this extent/page-based design 'almost always' results in read-streams at the device of size 24 (or more)." However, we now see that this is not the case in the mugy sample we're studying here: the sums of the percentages for the read-streams of lengths 1..23 is 21.362 + 24.302 + ... + 0.224 = 81.347%, in our mugy sample. So, in this sample, the percentage of reads that occur in read-streams of length >= 24 is 18.653%. However, this last value is artificially high, because the mugy sample we're studying does not exhibit very many parallel/simultaneous SQL sessions/streams. In a "busier" NPS system ("more SQL sessions/streams", with any combination of reads and/or writes), there will be "fewer long read-streams" – as we're now going to find out as we continue with our analysis.

One of the first things that grabs our attention about the above readWriteStream.py output is the occurrence of "very-long read-streams", whereas our study of the 41-datapoint graph above led us to believe long read-streams would be rare. How can we reconcile this? Well, here's where the Great Duality comes in handy again: we can't quite understand the numerics, so we turn to the graphics for guidance. Reconsider that big vertical blank-zone in the middle third of our "50,000-foot graph" again. There are no writes in that zone, but there reads. What kind of reads? Single-stand, or at least few-stand, reads, that's what kind. We can see that by looking at our 25,000-foot and 10,000-foot graphs. This means there is little/no competition for the disk head in that blank-zone. So the Linux block-I/O scheduler has a field-day doing optimal I/O scheduling for read-streaming in that zone, resulting in very-long read-streams.

In order to get a better handle on the overall question of "how important are long-read-chains?", let's now take a closer look at the "very-long read-chains" from the list above, namely those of length readStream.en >= 50. There are 1 + 3 + 2 + 3 + ... + 1 = 116 of these very-long read-chains, out of a total number of 26,378 + 24,206 = 50,584 read-chains – a vanishingly small percentage of 116/50,584 = 0.229321 5%.

If we expand out our list of 116 very-long read-chains (by "counting-with-multiplicity", i.e., expanding the length-51 read-chains from 1 lines to 3 lines, etc), and also compute the amount of time each of them takes (by using the starting/ending-elapsedtime's reported by readWriteStream.py), we get the following list. (I did this simply using a macro in Emacs/Calc.)

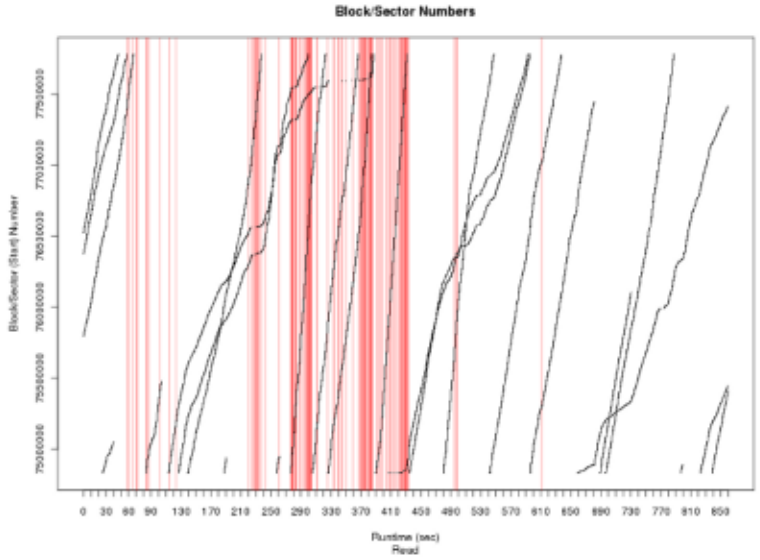
```
 50 : 340.392326032 -> 340.67632504        0.283999000
 51 : 220.70932711  -> 221.015777321       0.240450211
 51 : 423.852577250 -> 424.048575046       0.195998500
 51 : 431.557075985 -> 431.801075056       0.243999071
 53 : 208.543831474 -> 208.744575814       0.20074414
 53 : 480.760077621 -> 480.93602502        0.160747189
 54 :  84.785722521 ->  84.98457408        0.198852159
 54 :  86.768826416 ->  86.968104597       0.199278181
 54 : 303.057333503 -> 303.277327739       0.219994140
 55 :  58.481282201 ->  58.765325564       0.284043363
 55 : 495.016325085 -> 495.301531779       0.285206094
 56 : 101.212575483 -> 101.411842829       0.199267340
 56 : 230.752825811 -> 231.001325061       0.24840025
 56 : 282.636592452 -> 282.880325715       0.243733263
 57 : 207.0576766   -> 207.287326068       0.230250308
 57 : 330.832827680 -> 339.216025536       0.383997837
 57 : 428.347826082 -> 428.581328399       0.233500317
 58 :  60.288019461 ->  60.685090467       0.397071006
 59 : 497.137675087 -> 497.30832553        0.171250443
 61 :  83.316079652 ->  83.798600744       0.482611002
 62 : 276.93250091  -> 277.284075900       0.351406990
 62 : 208.612327213 -> 208.873325605       0.260998382
 62 : 343.480330357 -> 343.828325956       0.347995599
 63 : 236.489325250 -> 236.777329323       0.288003865
 63 : 242.445236000 -> 242.729326181       0.284090082
 63 : 350.351325337 -> 350.868325131       0.515999794
 65 : 232.380577608 -> 232.833325740       0.45274808
 65 : 279.064079216 -> 279.413196145       0.349116929
 65 : 295.528575513 -> 295.823272041       0.294697128
 65 : 610.66280442  -> 611.101575880       0.438681460
 66 :  66.158327745 ->  66.38032520        0.221997545
 67 :  71.001677027 ->  71.200838872       0.199761645
 67 : 430.465584148 -> 430.701187965       0.235603817
 68 : 123.59220977  -> 123.933939172       0.341729402
 68 : 284.380326037 -> 284.729011887       0.35068325
 68 : 383.800076793 -> 384.028076510       0.227997726
 69 : 480.943620521 -> 481.26457577        0.320740240
 71 : 376.488076117 -> 376.736076307       0.24000007
 72 : 283.907825201 -> 284.130325764       0.228500503
 72 : 373.948500792 -> 374.217600855       0.269010063
 73 : 277.509324802 -> 277.853377405       0.264052503
 73 : 289.908326971 -> 290.148080802       0.239753831
 74 : 311.144076208 -> 311.424076001       0.280000603
 76 : 209.12702407  -> 209.4311665         0.30334163
 76 : 333.53232509  -> 334.012325854       0.479999004
 82 : 311.576404438 -> 312.076326124       0.499921686
 82 : 301.861340000 -> 302.240327122       0.387980510
 84 : 302.367825610 -> 302.561447613       0.293621995
 85 : 225.977576718 -> 226.737091482       0.759514744
 87 : 207.840676720 -> 208.152001036       0.303314307
 88 : 225.552826701 -> 225.961575525       0.408740744
 88 : 277.380577338 -> 277.580075849       0.279498511
 90 : 200.066085062 -> 200.400013245       0.352928103
 91 : 302.995037256 -> 303.336075740       0.340230493
 92 : 230.311826542 -> 230.717325482       0.40549804
 97 : 376.005576046 -> 376.380076061       0.374500615
 98 : 372.690088813 -> 373.549500109       0.840787350
 99 : 415.145340195 -> 415.797325703       0.651985500
100 : 278.088001575 -> 278.663880057       0.575806482
103 : 260.578221180 -> 261.190077068       0.617350850
104 :  71.348468545 ->  71.700585204       0.360116749
105 : 359.636825292 -> 360.260825427       0.624000135
107 : 303.093676085 -> 303.460824063       0.367748008
110 : 231.008826704 -> 231.513325512       0.504480800
115 : 301.285074901 -> 301.822911377       0.417830470
117 : 382.524640308 -> 382.004076025       0.370236527
118 : 300.481331374 -> 300.929769001       0.448377627
118 : 383.400076294 -> 383.776076956       0.376000662
118 : 426.104576403 -> 426.823025042       0.719240639
```

We'd like to get some overall idea about where these very-long-read-chains are distributed(in-time) over our workload. One idea for doing that is to rearrange the preceding list according to the starting-elapsedtime of the very-long read-chains (this is easily done by putting the above list in a file, and doing "sort -g -b -k3" on it):

```
 55 :  58.481282201 ->  58.765325564      0.284043363
 68 :  60.298819401 ->  60.695800467      0.397871080
 66 :  66.156327745 ->  66.38032520       0.221997545
 67 :  71.081077027 ->  71.280838672      0.199761645
104 :  71.348468545 ->  71.708585204      0.360116740
107 :  71.760579525 ->  72.412576711      0.651997186
 61 :  83.316079652 ->  83.798090744      0.482011092
 54 :  84.785722521 ->  84.98457460       0.198852159
 54 :  88.768826410 ->  88.968104697      0.199278181
 56 : 101.217575483 -> 101.411842820      0.199267346
170 : 114.958536829 -> 115.661016967      0.702480138
 68 : 123.50228077  -> 123.958930172      0.341720402
 51 : 220.76932711  -> 221.015777321      0.246450211
 88 : 225.552826781 -> 225.961575525      0.408748744
 85 : 225.977576718 -> 226.737891462      0.759514744
410 : 226.78457958  -> 228.847025903      2.063246123
180 : 229.404326284 -> 230.180327575      0.776001291
 92 : 230.311826542 -> 230.717325402      0.40549884
 56 : 230.752825811 -> 231.001325001      0.24849925
110 : 231.008826704 -> 231.613325512      0.604408808
 65 : 232.380577666 -> 232.833325746      0.45274080
202 : 234.371825851 -> 236.467825427      2.096000576
 63 : 236.489325258 -> 236.777320123      0.288003865
 63 : 242.445236099 -> 242.729326301      0.284090082
183 : 260.578221180 -> 261.190077068      0.617850850
 62 : 276.93258081  -> 277.284075980      0.351486080
 88 : 277.300577338 -> 277.580075849      0.279498511
 73 : 277.509324802 -> 277.053377405      0.264052503
100 : 278.088081575 -> 278.663880057      0.575880482
 65 : 278.064076216 -> 279.413196145      0.340116920
143 : 279.432077372 -> 280.271026830      0.838749466
398 : 280.296576819 -> 282.660326044      2.363749225
 56 : 282.636592462 -> 282.880325715      0.243733763
235 : 282.904578612 -> 283.684325047      0.779746435
 72 : 283.907825201 -> 284.136325704      0.228500503
 68 : 284.369326637 -> 284.729011087      0.35960375
 76 : 280.12762487  -> 280.4811666       0.30334163
 73 : 288.908326871 -> 290.140080082      0.229753831
 53 : 293.543831474 -> 293.744575014      0.20074414
 65 : 295.528575518 -> 295.828272641      0.294607128
 57 : 297.0570766   -> 297.287326960      0.230250360
 87 : 297.840676729 -> 298.152081038      0.303814387
 62 : 298.612327218 -> 298.873326506      0.260908382
 90 : 299.056085052 -> 299.409013245      0.352920193
118 : 300.481331374 -> 300.929769001      0.448377627
115 : 301.205074001 -> 301.622911377      0.417836476
154 : 301.688579031 -> 302.238380587      0.549800578
 84 : 302.267825610 -> 302.561447613      0.293621985
128 : 302.612826585 -> 303.052825718      0.439999181
107 : 303.093076885 -> 303.460824063      0.367748008
128 : 303.505002020 -> 303.953042075      0.447959046
144 : 304.012825587 -> 305.100588016      1.095762429
 74 : 311.144076208 -> 311.424076801      0.280000683
 82 : 311.576404430 -> 312.076326124      0.499921606
120 : 325.736076829 -> 326.251305015      0.515228788
 76 : 333.53232589  -> 334.012326854     0.479000964
135 : 334.620327133 -> 335.472326275      0.851999142
 57 : 338.837627690 -> 339.216825536      0.383907837
 50 : 340.392326032 -> 340.67632504       0.283999608
 62 : 343.480330357 -> 343.828325056      0.347995599
140 : 345.445756045 -> 346.758257405      1.31350145
 63 : 350.353325317 -> 350.869325131      0.515999794
105 : 358.636825292 -> 360.260825427      0.624000135
160 : 367.168500367 -> 368.913520692      1.744950325
210 : 368.920827836 -> 370.814749394      1.893921558
424 : 370.466328312 -> 371.428326533      1.371907221
101 : 371.436325738 -> 372.084325591      0.647999853
119 : 372.082325085 -> 372.480026177      0.396701092
 98 : 372.699800813 -> 373.549596169      0.849707356
```

Right away, we can learn something from this list. Namely, "almost all" (all but 13 out of 116) very-long-read-chains occur during that big no-write blank-zone (213.113067321->515.582808871 sec). Yep, that makes sense (another good sanity check): in that no-write blank-zone, we can look at our "35,000-foot" graph, and see that not only are there no writes (so, the Linux block-I/O scheduler only has to worry about building read-chains), but also there are very few "strands" (i.e., parallel/simultaneous SQL sessions), making the block-IO scheduler's life even easier.

So this tells us "something". But it's not "enough". I.e., it starts to give us some idea of the very-long read-chain distribution (namely, "mostly in the no-write blank-zone"), but we want more. What we want is a "visualization" (the Great Duality again). We'll do that by starting from our "10,000-foot" graph, then adding vertical lines (116 of them) at (the starting-times of) our very-long read-chains (above):



So now by looking at this graphic, in light of everything else we've learned along the way, we can now answer the question (about read-streaming) we set out in quest of:

NPS experiences long read-streams (and especially "very-long" read-streams) only where only where there are "few strands (mostly, 1)". I.e., few parallel/simultaneous SQL sessions. In our mugly sample, that's especially the case in the no-write blank-zone in the middle third of our sample -- though it's not "read vs. write" that's important, it's only "few strands".

But, that case ("few strands") is, after all, rather uninteresting -- because "few strands" is a vanishingly small use-case scenario (and besides which, in that case we're pre-assured of getting the best possible performance anyway).

As nice as it has been (and it IS nice to see things at a deeper-than-normal level) for us to be able to visually/quantitatively validate things we "suspected" all along, we shouldn't let that cloud our "conclusion" set out at the beginning of this wiki page: Overall, the Blktrace Suite hasn't really told us anything really new/useful for NPS. For, the same conclusion ("few strands, i.e., little parallelism, implies long read/write-streams/chains") holds for any system, NPS or not. For, the key to what's happening at the device level lies with the Linux block-I/O scheduler -- not at higher layers. That is, after all, the reason the Linux block-I/O scheduler layer exists.

For Further Study

If NPS did want to get into the business of block-I/O scheduling (and it might very well want to consider that), some research into recent literature would be necessary, of course. For example, there's this recent paper from HP Labs: http://www.hpl.hp.com/techreports

(2009/HPL-2009-344.pdf)

Here's a curious question: Why do all the "strands" on our "10,000-foot" graph only have positive slope? It would seem that the Linux block-IO scheduler only schedules the head to do reading in the "positive" direction (from lower block/sector numbers to higher ones). Why should that be? Hint: This is mentioned in the HP Labs paper just cited, where the uni-directional algorithm is called Circular SCAN, but see also http://www.ece.cmu.edu/~ganger/papers/sigmetrics94.pdf.

<<TBD – WHAT'S NEXT?>>

Labels

perf research

II Email Chain: Stop Working (August 26)

- From: Daniel Feldman
  To: Walter Tuvell
  Date: 08/26/2011 10:28 AM
  Subject: *IBM Confidential: STD

  I see by your frequent changes to the wiki page that you are working on the disk streaming behavior problem while you are out on Short Term Disability (STD). While I understand your desire to continue to make progress on this assignment, you should stop working now since you are on STD. It is more important that you focus on getting well and returning to work fully able to participate as a member of the Netezza Performance Architecture team. Also, it is inappropriate for me to give you assignments or to interact with you to properly manage the execution of assignments while you are out on STD. As I'm not able to interact with you as I would were you here, it is impossible for me to supervise your work and provide guidance that will ensure that this assignment (or any other) is being performed in the way that is best for IBM.

  Please let me know that you have received this email and please include in your reply one or more phone numbers that I can reliably use to get in touch with you.

- From: Walter Tuvell
  To: Daniel Feldman
  Date: 08/26/2011 10:33 AM
  Subject: *IBM Confidential: Re: STD

  I will not do any more work on this, or on anything else.

  I am not available by phone. Anything that needs to be said to me can be done via Notes.

JJ Email Chain: Email Vetting (August 4)

- From: Walter Tuvell
  To: Daniel Feldman
  Date: 08/04/2011 10:58 AM
  Subject: Vet, please

  To: Jay Wentworth
  Subject: NPS disk partitions?

  Hi Jay, I have a question for you. (Or, if you know of a better person to ask, a reference pointer.)

I'm doing a wiki page on the "Blktrace Suite" of tools (search Confluence wiki for "blktrace", you want the page entitled "Blktrace, Blkparse, Btt").

The PNG attached to this note is one of the graphics on the wiki. On the y-axis, it shows the "block" (actually, 512-byte disk sectors) numbers for one NPS device, during ~14.5 min of a multi-stream (muqry) test run. The graphs is noticeably banded into 4 horizontal bands, which I believe are disk partitions. Can you tell me what those bands are for? And, what are their typical sizes (this is a TF6 I used for this test)?

It's my (shaky) understanding that the disk's outer cylinders correspond to low sector-numbers (i.e., the bottom band), and it's used for the NPS primary partition. Is that right? And the inner cylinders (top band) are for mirror secondary, right? And one of the other bands is for nztmp (is that's it's correct designation), but which one? And what about the last band (I think I may have heard about it in some video presentation, but I'm not sure)?

Also, do you have any idea what that huge vertical blank-ish region in the middle third of the graph (it means there are nearly no writes happening during that time range) is doing?

Thanks!

▶*Attachment (Rplot.bno_c.png) omitted (irrelevant to this Complaint).*◀

- From: Daniel Feldman
  To: Walter Tuvell
  Date: 08/04/2011 01:34 PM
  Subject: Re: Vet, please

  This is fine. If it were me, I'd include a link to the relevant wiki page rather than requiring him to search for it. The principle is: if you're asking someone to do something for you, you should make it as easy as possible for them to do it. Up to you, though.

- From: Walter Tuvell
  To: Jay Wentworth
  Date: 08/04/2011 01:44 PM
  Subject: NPS Disk partitions?

  Hi Jay, I have a question for you. (Or, if you know of a better person to ask, a reference pointer.)

  I'm doing a wiki page on the "Blktrace Suite" of tools. (Search Confluence wiki for "blktrace", you want the page entitled "Blktrace, Blkparse, Btt". Alternatively, if

you're connected to the Netezza network in such a way that hyperlinks from Notes work for you, you can click http://wiki2.netezza.com:8080/display/Perf/Blktrace%2C+Blkparse%2C+Btt if you wish.)

The PNG attached to this note is one of the graphics on the wiki. On the y-axis, it shows the "block" (actually, 512-byte disk sector) numbers for one NPS device, during ~14.5 min of a multi-stream (muqry) test run. The graph is noticeably banded into 4 horizontal bands, which I believe are disk partitions. Can you tell me what those bands are for? And, what are their typical sizes (this is a TF6 I used for this test)?

It's my (shaky) understanding that the disk's outer cylinders correspond to low sector-numbers (i.e., the bottom band), and it's used for the NPS primary partition. Is that right? And the inner cylinders (top band) are for mirror secondary, right? And one of the other bands is for nztmp (is that it's correct designation), but which one? And what about the last band (I think I may have heard about it in some video presentation, but I'm not sure)?

Also, do you have any idea what that huge vertical blank-ish region in the middle third of the graph (it means there are nearly no writes happening during that time range) is doing?

Thanks!

▶*Attachment (Rplot.bno_c.png) omitted (irrelevant to this Complaint).*◀

KK Email Chain: Dan's Public Embarrassment

- From: Daniel Feldman
  To: Steve McAfee, Gordon Booman, Jay Wentworth, Fritz Knabe
  Cc: netezza-perf-arch, John Metzger
  Date: 08/04/2011 11:43 AM
  Subject: Trunk performance regressions
  Folks,

  Pursuant to an ongoing effort to calibrate the performance of the Wahoo prototype, Sujatha Mizar has been running a number of performance tests on that platform and on a Skimmer. It became clear early last week that we needed to try to discriminate among the possible sources for unexpected (slow) performance seen during some of these tests. So, she ran the Atomics and TPC-DS subsets of the perfbar suite against a release 6 build (/nfs/production/builds/rel-6.0/110505-17071rel-6.0/) and int-trunk (a turbo build off of int-trunk stream with a time basis of Jul 26, 2011 10:46:57 AM). The database is a 100GB TPC-DS database.

  The attached spreadsheet shows that we have significant regressions in substantial numbers of tests from both suites. Overall, the Atomics suite shows about a 19% slow down while the TPC-DS suite shows about a 52% slowdown.

The spreadsheet is divided into two sections, one for Atomics and one for TPC-DS. In each section, the test cases are itemized along with the elapsed time (ET) for each of the two builds tested and the ratio of the int-trunk build's elapsed time to the release 6.0 build's elapsed time. The ratio is identified as the ETR. The last cell in each of the two ETR columns has the geometric mean of the ratios; this is where the 19% and 52% numbers come from.

Sujatha made the following notes on these results:

1) The Rollback test no longer works on the trunk. If anyone from dev would like to take a look at this please let me know and I can point you to the failure. It is easily reproducible by doing "nzsql -d tpcds100 -f /nz/Bar5.1/sql/atomics/rollback.sql" . S1-8 is currently pointing to the kit I used when I observed this.

2) The following Atomics queries seem to have regressed considerably: THREE_WAY_JOIN , NESTED_LOOPS , MERGE_JOIN ,LEFT_JOIN ,SUB_SELECT ,JOIN_MULTI_HASH ,RIGHT_JOIN and JOIN_BROADCAST

3) TPC-DS queries seem to have regressed more than the Atomics queries. The TPC-DS queries of interest here are QUERY072, QUERY072V, QUERY091 ,QUERY013 ,QUERY082 ,QUERY017 and QUERY084. There are a whole bunch of queries that have ETRs higher than 1 but these are at least 5x times slower.

I will ask Sujatha to open one or more defects corresponding to these findings. Would you like one for the whole kit and kaboodle? One for each suite? One for each test case? Someting else?

▶*Attachment (spreadsheet) omitted (irrelevant to this Complaint).*◀

■ From: Gordon Booman
To: Daniel Feldman
Cc: Fritz Knabe, Jay Wentworth, John Metzger, netezza-perf-arch, Steve McAfee
Date: 08/04/2011 12:03 PM
Subject: Re: Trunk performance regressions

As part of the int-genesis testing, we've been looking at some long standing performance issues in int-trunk and int-genesis. In particular, Larry noticed a 30% degradation a while ago (SWS-68205). Have Larry and Sujatha compared results? Is this different, in addition, or...?

■ From: Larry Lutz
To: Gordon Booman
Cc: Daniel Feldman, Fritz Knabe, Jay Wentworth, John Metzger, netezza-perf-arch, Steve McAfee
Date: 08/04/2011 12:40 PM
Subject: Re: Trunk performance regressions

I have opened three main bugs for the regressions against int-trunk/int-genesis. These three issue seemed categorically different, which is why I logged them as three issues. I didn't bother looking at the rest of the regressions (mainly atomics, tpc-ds & tpc-h) because most (if not all) would likely be traced back to the scan speed bug. It would be a significant effort that seems better left till after the scan bug is fixed.

For my case with the TF12 (M3 &HS22) the scan speed bug is obvious. I think Ottavio tried to repro on a P50 of old vintage, but couldn't repro there. Sujatha is on a P50 too I believe, so she may or may not be experiencing the same issue. I would suggest debugging the scan speeds on that system to verify whether or not it has the same issue.

- From: Daniel Feldman
  To: Larry Lutz
  Cc: Gordon Booman, Fritz Knabe, Jay Wentworth, John Metzger, netezza-perf-arch, Steve McAfee
  Date: 08/04/2011 01:13 PM
  Subject: Re: Trunk performance regressions

  This testing was done on a small database (100GB) on a Skimmer.

  I think it is important to get them all logged; if it turns out that the scan speed issue, once resolved, results in may bugs being closed, that seems ok to me. My question in the original mail is more about how to log them than whether or not to. Perhaps someone on the core engineering team can weigh in on this?

  Also, the combination of the problems Larry already identified and the (perhaps the same, perhaps not) problems that Sujatha is finding, when combined with the difficulty of diagnosing the problem suggests that we're missing something in the regular build bag-of-tricks. It seems that we should have some kind of regularly executed (say, nightly) mechanism that will flag (at least some) performance problems. I imagine this has been considered in the past and I'm curious why it doesn't seem to be in place. Can anyone educate me on this?

- From: Gordon Booman
  To: Daniel Feldman
  Cc: Larry Lutz, Fritz Knabe, Jay Wentworth, John Metzger, netezza-perf-arch, Steve McAfee
  Date: 08/04/2011 01:38 PM
  Subject: Re: Trunk performance regressions

  You can log bugs by symptom, or by cause. As a developer, I prefer cause. Which is why I am suggesting some more investigation to narrow it down. There are good reasons for symptom, but it leads to way too many tickets.

I think the regression test you describe is Amal's raison d'etre...No?

- From: Daniel Feldman
  To: Goordon Booman
  Cc: Larry Lutz, Fritz Knabe, Jay Wentworth, John Metzger, netezza-perf-arch, Steve McAfee
  Date: 08/04/2011 02:36 PM
  Subject: Re: Trunk performance regressions

  Adding Amal as his name has come up a couple of time.

  Well, I think the formal regression testing associated with a release, change in hardware, etc, is the original motivation for Amal's group. I think there's a slightly different question about automating some basic perf sanity checking at or near check-in time.

  The symptom v. cause question is interesting. I'm ok with the idea of spending a little more time on the first level of diagnosis.

- From: Larry Lutz
  To: Gordon Booman
  Date: 08/04/2011 03:39 PM
  Cc: Daniel Feldman, Fritz Knabe, Jay Wentworth, John Metzger, netezza-perf-arch, Steve McAfee
  Subject: Re: Trunk performance regressions

  I looked at the SUT with Sujatha. The test framework is not PerfBar, it is something else. In PerfBar we check database statistics prior to running and re-compute as necessary. On this system the stats are stale. Worse yet, as I understand it, JIT stats are turned off, so the planner would be full dependent on existing statistics. I don't know if it makes sense to log bugs with JIT stats off as this is not a deliverable configuration. And since stats are invalid, I think all tests would need to be re-run from scratch to get valid measurements. As it is, I would expect many plans would differ.

  The scan speed on this system is fine as Sujatha verified.

- From: Walter Tuvell
  To: Daniel Feldman
  Date: 08/04/2011 03:54 PM
  Subject: Fw: Trunk performance regressions

  /* This just to you, I have no intention of getting involved in the public discussion, so this is not a request for vetting. */

It sounds like what Sujatha may have done is run WaltBar on Skimmer, but using Wahoo settings (instead of Skimmer settings) for NPS. If so, then she's done something I explicitly warned her to be aware of, and avoid doing, and her results would of course be bogus. (I never did such a thing, but I was aware of the potential when I briefed her on how to do Wahoo testing, so I was very careful to emphasize it [the difference of Wahoo settings and Skimmer settings].)

(This in no way involves an "ad hominem" attack on anybody, just impartial observations of a technical and procedural nature.)

- From: Daniel Feldman
  To: Walter Tuvell
  Date: 08/04/2011 04:03 PM
  Subject: Re: Fw: Trunk performance regressions

  OK.

- From: Daniel Feldman
  To: netezza-perf-arch, Fritz Knabe
  Date: 08/05/2011 09:58 AM
  Subject: The controversial perfbar runs

  Folks,

  Here is my understanding of what was run that prompted my email yesterday asking about opening performance defects:

  Three different builds were run on the Skimmer platform. They were:

  production 6.0
  int-trunk as of 7/26
  dev_wahoo (-Dnowahoo (or whatever the actual switch name is))

  All were run with the following parameters/configuration/process:

  Jit stats were off
  2phase planner was off
  genstats was not executed - but, all were performed on newly instantiated 100GB non ballooned TPC-DS databases using the same initialization and load processes

  These appear to me (unless I've forgotten to ask about something) to be identical in all regards except the build, which is exactly what we wanted to control for.

  The differences in performance were documented yesterday.

It was asserted that not running jitstats and 2pp is non-standard/non-production/not-supported - something like that. I know that customers do choose to turn off both the 2pp and jit stats and that they do so causes consternation but not a withdrawal of support. Maybe I misunderstood the point, if so, a clarification is welcome.

Not running genstats is likely to result in sub-optimal (if not actually pessimal) plans.

An investigation of two specific TPC-DS queries (I don't know which ones) showed that the prod 6.0 and the int-trunk builds generated different plans. The plans generated by the dev-wahoo build for those two queries weren't examined.

The scan speed issue that is plaguing int-genesis was not found in this test.

It seems to me, naively perhaps, that the two builds that produced different plans should have produced the same plans. Can someone explain to me why this wouldn't be the case and why we would not consider this a regression between prod 6.0 and int-trunk?

- From: Larry Lutz
  To: Daniel Feldman
  Cc: Fritz Knabe, netezza-perf-arch
  Date: 08/05/2011 10:33 PM
  Subject: Re: The controversial perfbar runs

  Here is the issue. The planner needs stats to make decisions, otherwise it is flying blind. Who knows what dumb decisions it will make. The planner normally gets stats from JIT. However, if that is off, then it must go to the stored stats. If those are not there, then it has nothing to work from. If a customer has JIT off (as was pre-JIT) and they call in to complain about query performance, the first thing we tell them to do is generate updated statistics and see if the issue still exists. In this case we are not doing that. Also note that to get the full stats needed, we have to start an nzsql session, turn off JIT stats, then run gen stats. If we do not turn off JIT before running gen stats, then we will only get express stats that still rely on JIT for the big tables. The database creation process does not gen full stats for any database; it normally only creates the default express stats which rely on JIT.

  Regarding regression, I really don't know how the system should behave in this scenario as it is not a config we have ever tested (at least not intentionally). Perhaps Jay or Babu could provide input as to whether or not it makes sense to file a bug. If they are going to write the tickets off as invalid (and therefore never look at the them), then I don't see the point.

LL Email: Notice Of Relocation (August 20)

■ From: hrprofil
To: Walter Tuvell
Date: 08/20/2011 11:02 AM
Subject: (A) HR Employee Record Change Confirmation for: Tuvell

.....IBM CONFIDENTIAL.....

HR EMPLOYEE RECORD CHANGE CONFIRMATION

----
* Warning: Please DO NOT reply to the HRPROFIL userid, it is only a service *
* machine. Many personal updates can be made using About You via *
* w3.ibm.com/hr.....THANK YOU *

HR Data as of: 08/19/11 (* indicates HR Employee record change)

DEPT: Z1-CMOA (Netezza)

ADDRESS INFORMATION
Last Name. . . . . : Tuvell
First Name . . . . : Walter
Middle Name. . . . :
Initials . . . . . : W
Serial . . . . . : 0G3821
Address. . . . . : 836 Main Street
Address (2nd line) :
City and State . . : Reading, MA
Zip Code . . . . . : 01867
Country. . . . . : UNITED STATES
Phone Number (Type): 781-944-3621 (REGULAR)

TAX / LOCATION INFORMATION
Tax Jurisdiction Code (TJC): MAA
* IBM Work Location (WKL). . : HC5
Address. . . . . .: 26 FOREST ST
Address (2nd line):
City and State . : MARLBOROUGH, MA
Zip Code. . . . . : 01752
* Actual Work Location (AWL) : HC5
Address. . . . . .: 26 FOREST ST
Address (2nd line):
City and State . : MARLBOROUGH, MA
Zip Code. . . . . : 01752
Work Place Indicator (WPI) : TRADITIONAL OFFICE

EMPLOYEE STATUS
Current Status . . . . . : REGULAR FULL-TIME EMPLOYEE
ACTIVE
Supplemental Type . . . . :

PERSONAL INFORMATION
Service Reference Date . . : 01/01/2011
Hire Effective Date . . . : 01/01/2011
Date of Birth. . . . . : xx/xx/xxxx (Go to About You to view this data)
Social Security Number . . : xxx-xx-xxxx (Go to About You to view this data)

EMERGENCY CONTACT INFORMATION
Emergency Contact (1) . . : LINDA KING
Relationship . . . . . : WIFE
Phone Number (Type). . . . : 781-944-3617 (REGULAR)
Emergency Contact (2) . . : SUSAN TUVELL
Relationship . . . . . : DAUGHTER
Phone Number (Type). . . . : 781-944-3617 (REGULAR)

ETHNICITY INFORMATION
Hispanic/Latino. . . . . : N

RACE INFORMATION
White. . . . . : Y
Black. . . . . : N
Asian. . . . . : N
Amer Indian/Alaska Native. : N
Native Hawaiian/Pacific Isl: N

VETERAN MILITARY DISCHARGE DATE - LAST UPDATED : 12/08/2010

Veteran Military Discharge Date . . . . . :

CURRENT VETERAN INFORMATION - LAST UPDATED : 12/08/2010

Armed Forces Service Medal Veteran . . . . . : N
Recently Separated Veteran . . . . . : N
Disabled Veteran . . . . . : N
Other Covered Veteran . . . . . : N
Newly Separated Veteran . . . . . : N
Veteran of the Vietnam Era . . . . . : N
Special Disabled Veteran . . . . . : N
Veteran Not Included under existing Category definitions : N

Please direct any questions to the Employee Services Center (1-800-796-9876)
Access About You at http://w3.ibm.com/hr/aboutyou/

.....IBM CONFIDENTIAL.....

MM Email: Original Complaint Filing (August 18)

- From: Walter Tuvell
  To: Sam Palmisano, Randy MacDonald, Steve Mills, Robert Weber, Lynea St. Pier
  Cc: Arvind Krishna, Prat Mogue, David Flaxman, Russell Mandel
  Date: 08/18/2011 01:06 PM
  Subject: Corporate Open Door filing

  To Sam and selected members of executive staff, and Lynea St. Pier:

  It pains me greatly to take this action, but:

  Pursuant to the "Corporate Open Door" clause of the IBM Concerns and Appeals Program (section 2.4 of document number USHR102, dated May 19, 2008), as well as the Confidentiality Speaking clause, I hereby formally submit the attached (two-part) Complaint for your consideration and action.

  Confidentially

  Please read it carefully, and take it seriously, as I do. When you read it, you will see why there is no more appropriate action I can be taking, regretfully. (You may have seen a preliminary draft I sent two weeks ago; this is the final version.)

  I pledge to cooperate with, and I am available to communicate/meet with, anyone, anywhere, anytime. Initially though, I can be reached by Lotus Notes only, because I prefer to have a written record at this time (I'm sure you understand).

  I look forward to your response.

  Thank you.

  ▶*Attachments (omitted here): IbmComplaint-I.pdf, IbmComplaint-II.pdf (versions 1.0).*◀

NN Email Chain: Delayed Investigation (August 25)

- From: Walter Tuvell
  To: Sam Palmisano, Randy MacDonald, Robert Weber, Steve Mills, Lynea St Pier
  ▶*Also forwarded separately to Carolyn Austin, assistant to Lynea St. Pier.*◀
  Date: 08/25/2011 08:06 AM
  Subject: Re: Corporate Open Door filing

  Gentlemen and Lady -

  It was a week ago today that I formally filed my Complaint with Corporate Open Door, and with Confidentially Speaking. I know you received it (because Lotus

Notes is IBM's official/trusted communications medium, and Notes notified me of no email failures).

Yet to date, I have not heard back from anyone. I have not even received acknowledgement of receipt of my filing.

This is contrary to what the programs themselves promise. In the Concerns and Appeals employee handbook, COD promises initial contact "normally within two business days", and assignment of a case worker within that timeframe. And while CS doesn't appear to promise a specified timeframe for initial contact (it only speaks of "generally within 20 days" for completion of the process), it does claim to be "a tangible example of our values in practice", hence "prompt initial contact" is a reasonable expectation. A week is not "prompt".

If I have submitted my Complaint improperly in any way, please advise me what I should do, and I will promptly make the appropriate correction.

Absent such an error on my part, it is reasonable for me to expect some sort of contact from both COD and CS by the end of this week. If I do not receive such contact, I will be justified in interpreting IBM's silence to mean that IBM, at all levels, implicitly supports what happened to me as "OK" (i.e., my Complaint has "no merit"). At that point I will therefore have no alternative but to seek relief elsewhere (as Randall Mandel already told me I am free to do at anytime anyway).

- Walt Tuvell

PS. As mentioned previously, I am available only by Lotus Notes at this time (other communications arrangements can be made via Notes).

■ From: Russell Mandel
To: Walter Tuvell
Date: 08/25/2011 03:43 PM
Subject: Re: Corporate Open Door filing

I will begin investigating your issue(s) now that I have returned from vacation. I do not plan on discussing your concerns directly with you until you return from Short Term Disability, so you may concentrate on your health improving.

■ From: Walter Tuvell
To: Russell Mandel
Cc: Sam Palmisano, Randy MacDonald, Robert Weber, Steve Mills, Lynea St Pier
Date: 08/25/2011 05:19 PM
Subject: Re: Corporate Open Door filing

Russell -

This is ABSOLUTELY UNACCEPTABLE. The very REASON I'm on STD leave, and will continue to remain so, is due DIRECTLY AND SOLELY to the psychological abuse (IIED) that is being heaped upon me by Dan Feldman, and yourself, and everybody else who has touched this case to date. And you know it. The ONLY way for me to recover sufficiently to return to work from STD is to settle this case. Properly and correctly.

The ONLY reason for you/IBM to delay at this point is that IBM must be hoping I'll stumble into some idiotic trap (such as the "Lazy" scandal) so you'll have false reason to fire me, thereby avoiding the necessity for IBM to actually deal forthrightly/honestly/ethically with my case. You cannot point to any policy that prevents IBM from working with me on my case NOW. Nothing in "IBM Law" (as expressed in the "employee handbook": BCG, AYJ, C&A) says anything to the effect of "STD disqualification". (To do so would run afoul of the ADA.) To the contrary, IBM Law guarantees me certain rights, in enforceable writing, prominent among which is "prompt" dealing with wrongdoing. I hereby INSIST upon being afforded that "promptness" right IMMEDIATELY.

Furthermore, as is well-known for all investigative proceedings of this sort ("Rules of Procedure"), the Complaint I have filed MUST BE CONSIDERED FACTUALLY TRUE at this pleading stage. Therefore, the people you need to contact now are those I have accused of wrongdoing, NOT ME. Justice demands that they must now be given the opportunity to formally respond to my charges, IN WRITING (no more "secret, behind-doors whispering"). Their responses must then be forwarded to me for my response. The people you need to get "affidavits" from at this point are: Fritz Knabe, Dan Feldman, John Metzger, Diane Adams, Lisa Due, Russell Mandel, Arvind Krishna (because of his comment that he "doesn't care about IBM Law"). If you don't know what I'm talking about, just check with your nearest lawyer.

To repeat what's in my Complaint: You are not a "competent authority" to hear my case, for reasons set forth in my Complaint (specifically, you are a named party to the wrongdoing -- in fact you STILL haven't responded to my properly-filed "third-party complaint"). Therefore, it is IMPERATIVE that you/IBM turn this investigation over to a "trusted higher authority" -- an INDEPENDENT BODY -- for further prosecution. If that independent body isn't named by IBM's Board of Directors (or better, a committee-to-name-the-independent-investigative-body should be appointed by the BoD), there will be obvious and insurmountable CONFLICT-OF-INTEREST (bias) questions to be overcome, and any conclusions made that are contrary to my interests would be too-obviously-suspect.

I DEMAND that substantive investigation into my case begin the first thing next week (9:00 AM EDT, Monday, August 29). What must happen at that time is that the above-named defendants must be forwarded my Complaint, for formal response. Due to reasons of continuing all-too-obvious mishandling of my case, I EXPECT to be UPDATED DAILY about the progress of this case. By Lotus Notes email, so it's on-the-record. I consider anything less than meeting these these demands (which are reasonable, and even required by IBM Law) to be CONTINUED CAPRICIOUS ABUSIVE-OF-POWER AND INTENTIONAL-IIED/HOSTILE-WORK-PLACE.

If anything I've written here is unclear, contact me IMMEDIATELY, and I will "promptly" clarify to the extent required to satisfy any "reasonable-person" standard.

OO BCG Program