Item 334
git_comments:

1. "t1 t2 t1 t3 t2 t3" ----------- -------- --------

git_commits:

1. **summary:** LUCENE-2878: Another test bug
   message: LUCENE-2878: Another test bug git-svn-id: https://svn.apache.org/repos/asf/lucene/dev/branches/LUCENE-2878@1408674 13f79535-47bb-0310-9956-ffa450edef68

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Allow Scorer to expose positions and payloads aka. nuke spans
   description: Currently we have two somewhat separate types of queries, the one which can make use of positions (mainly spans) and payloads (spans). Yet Span*Query doesn't really do scoring comparable to what other queries do and at the end of the day they are duplicating lot of code all over lucene. Span*Queries are also limited to other Span*Query instances such that you can not use a TermQuery or a BooleanQuery with SpanNear or anthing like that. Beside of the Span*Query limitation other queries lacking a quiet interesting feature since they can not score based on term proximity since scores doesn't expose any positional information. All those problems bugged me for a while now so I stared working on that using the bulkpostings API. I would have done that first cut on trunk but TermScorer is working on BlockReader that do not expose positions while the one in this branch does. I started adding a new Positions class which users can pull from a scorer, to prevent unnecessary positions enums I added ScorerContext#needsPositions and eventually Scorere#needsPayloads to create the corresponding enum on demand. Yet, currently only TermQuery / TermScorer implements this API and other simply return null instead. To show that the API really works and our BulkPostings work fine too with positions I cut over TermSpanQuery to use a TermScorer under the hood and nuked TermSpans entirely. A nice sideeffect of this was that the Position BulkReading implementation got some exercise which now :) work all with positions while Payloads for bulkreading are kind of experimental in the patch and those only work with Standard codec. So all spans now work on top of TermScorer (I truly hate spans since today) including the ones that need Payloads (StandardCodec ONLY)!! I didn't bother to implement the other codecs yet since I want to get feedback on the API and on this first cut before I go one with it. I will upload the corresponding patch in a minute. I also had to cut over SpanQuery.getSpans(IR) to SpanQuery.getSpans(AtomicReaderContext) which I should probably do on trunk first but after that pain today I need a break first :). The patch passes all core tests (org.apache.lucene.search.highlight.HighlighterTest still fails but I didn't look into the MemoryIndex BulkPostings API yet)

jira_issues_comments:

1. here is the patch
2. I took a quick glance at the issue (not a proper review though: its too easy to get lost in the spans!) Here are my thoughts/questions: * I like the idea of adding the positions iterator separate, this is one thing i don't like about the Spans API. * I don't like the MultiSpansWrapper... it only exists to support PayloadSpanUtil and tests? I don't like PayloadSpanUtil either, i think its a huge trap and it should move to src/test or be removed. * any ideas of the performance comparison of this patch as-is (SpanScorerWrapper over the Scorer/bulk enum) versus the existing Spans which doesn't use the bulk API? Seems like if we could get the API how we like on this issue, and fix all codecs bulkpositions implementations to work with payloads, that this would be a nice step we could make...
3. bq. I don't like PayloadSpanUtil either, i think its a huge trap and it should move to src/test or be removed. It's not that bad of a trap :) I has warnings all over it - some people use it out there. I think it's more useful than trapful - if not just as demo code. It is src/test quality at best though though. I never would have committed it to core - I expected to put it in contrib at best. But someone else committed it at the time, so I just did a 'meh' and put warnings about experimental on it. I forgot to take it out of the patch on the issue or something. I never even used it or tested it much myself - I just wanted to put in some example/test code for this use case - I know some users on the list have used it for some form of simple highlighting, where the coords to highlight are in the payloads. I wouldn't trust it these days though :) It will be found less if its in src/tests, but that is probably a fine place for it - it would need some love/support above what it has to really make sense in any module/contrib. I've never really meant to maintain it myself - and I've never really felt it belonged in core (just never had the itch to squash it either).
4. bq. It's not that bad of a trap I has warnings all over it - some people use it out there. I think it's more useful than trapful - if not just as demo code. In this case maybe contrib? And if we were careful/re-organized tests a bit, MultiSpansWrapper could be pkg-private.
5. A first impression, no time to read the whole patch yet. I like Positions being abstract, instead of the current Spans interface. The name Positions is understated though. It has a begin and an end that can be iterating/iterated in place, so it is actually "a" spans. I wish I had a better name. I hope that the addition to TermScorer does not slow it down. The patch has some duplicates (see Scorer for example), but that is easy to fix. It also contains a 2005 copyright for a new file. Time goes by. More later...
6. bq. In this case maybe contrib? I dunno - I think I would be for it if I was willing to work on it and maintain it. But honestly, I am not. Some users have figured out how to use it for something, but I never really used it for anything. So my interest is low, and list of other stuff I'd love to tackle high. I'm still pro removing some stuff from contrib, so I don't want to be responsible for more cruft going in. Perhaps anyone using it should just copy out the code to their project ... thats really the spirit it was written in anyway - here is sample code to get you started - showing what you can do with spans and payloads.
7. First of all thanks for the reviews so far... seems like the overall feedback is positive though. I should have made my point more clear that the most of this patch is a PoC more than anything else. The changes to spans are rather to proof correctness and give the bulk api a little exercise. bq. I don't like the MultiSpansWrapper... it only exists to support PayloadSpanUtil and tests? I don't like PayloadSpanUtil either, i think its a huge trap and it should move to src/test or be removed. The reason for MultSpansWrapper is pretty simple. MultiBulkEnum is not entirely implemented yet so I had to move the Spans to per-segment using AtomicReaderContext. What I needed was an easy way to

cut over all the tests and that seemed to be straight forward. Its really just a testing class and should go away over time. bq. any ideas of the performance comparison of this patch as-is (SpanScorerWrapper over the Scorer/bulk enum) versus the existing Spans which doesn't use the bulk API? not yet, once I get back to this I will run a benchmark bq. A first impression, no time to read the whole patch yet. my plans are to cut over spans to AtomicREaderContext on trunk so this patch will be way way smaller and easier to read maybe you should just skip all the span stuff for now. those are just ARC based changes bq. PayloadSpanUtil <dream> Spans as they exist today should go away anyway so this is not much of a deal for now.</dream> bq. The name Positions is understated though. It has a begin and an end that can be iterating/iterated in place, so it is actually "a" spans. I agree we might need to go away from this start end thing and have a wrapper on top of Position(s) bq. I hope that the addition to TermScorer does not slow it down. Those positions == null checks are well predictable and should be optimzed away - if hotspot plays tricks on us we can still specialize a scorer....

8. Attaching my current state. I committed LUCENE-2882 today which contained all the cut over to SpanQuery#getSpans(AtomicReaderContext) that made this patch much smaller and easier to read now. I also fixed MemoryIndex such that now all tests pass with -Dtests.codec=Standard Not much progress other than that - more tomorrow....
9. I haven't looked at the patch, but one of the biggest issues with Spans is the duality within the spans themselves. The whole point of spans is that you care about position information. However, in order to get both the search results and the positions, you have to, effectively, execute the query twice, once to get the results and once to get the positions. A Collector like interface, IMO, would be ideal because it would allow applications to leverage position information as the queries are being scored and hits being collected. In other words, if we are rethinking how we handle position based queries, let's get it right this time and make it so it is actually useful for people who need the functionality. As for PayloadSpanUtil, I think that was primarily put in to help w/ highlighting at the time, but if it has outlived it's usefulness, than dump it. If we are consolidating all queries to support positions and payloads, then it shouldn't be needed, right?
10. {quote} I haven't looked at the patch, but one of the biggest issues with Spans is the duality within the spans themselves. The whole point of spans is that you care about position information. However, in order to get both the search results and the positions, you have to, effectively, execute the query twice, once to get the results and once to get the positions. A Collector like interface, IMO, would be ideal because it would allow applications to leverage position information as the queries are being scored and hits being collected. In other words, if we are rethinking how we handle position based queries, let's get it right this time and make it so it is actually useful for people who need the functionality. {quote} Grant I completely agree! Any help here very much welcome. I am so busy fixing all the BulkEnums and spinnoffs from this issue but I hope I have a first sketch of how I think this should work by the end of the week! bq. As for PayloadSpanUtil, I think that was primarily put in to help w/ highlighting at the time, but if it has outlived it's usefulness, than dump it. If we are consolidating all queries to support positions and payloads, then it shouldn't be needed, right? Yeah!
11. Attaching my current state - still rough & work in progress though.... In this patch I added a PositionsIntervalIterator returned from Scorer#positions() and implemented some usecases like filtering Near / Within (unordered) with Term & BooleanScorer2. BooleanScorer2 if in conjunction mode now has a PositionIntervalIter implementation that returns the minimal interval of its unordered query terms and can easily be filtered within a range of pos (range, first) or within a relative positions (near) simply by wrapping it in PositionFilterQuery. An example for this is in TestBooleanQuery. The PosIntervalIterator decouples positional operation nicely from Scoring / matching so a Boolean AND query gets the normal query score but can be restricted further based on positions. Even adding positional scoring can simple be plugged on top of it. Further, with a specialized Collector implementation - positions could be fetched only if the score is within the top N to prevent positions matching for all documents. One big problem is BooleanScorer which does the bucket based scoring - I will ignore that one for now. I need to run some benchmarks to see if that does any good though but haven't had time to do so. If somebody has time and take a look at this patch - feedback would be very much appreciated.
12. just attaching my current state
13. This patch looks awesome (and, enormous)! Finally we are making progress merging Span* into their corresponding non-positional queries :) I like how you added payloads to the BulkPostings API, and how someone is finally testing the bulk positions code. So now I can run any Query, and ask it to enumerate its positions (PositionInterval iterator), but not paying any price if I don't want positions. And it's finally single source... caller must say up-front (when pulling the scorer) if it will want positions (and, separately, also payloads -- great). It's great that you can just emulate spans on the new api with SpanScorerWrapper/MockSpanQuery, and use PositionFilterQuery to filter positions from a query, eg to turn a BooleanQuery into whatever SpanQuery is needed -- very nice! How does/should scoring work? EG do the SpanQueries score according to the details of which position intervals match? The part I'm wondering about is what API we should use for communicating positions of the sub scorers in a BooleanQuery to consumers like position filters (for matching) or eg Highlighter (which really should be a core functionality that works w/ any query). Multiplying out ("denormalizing") all combinations (into a flat stream of PositionIntervals) is going to be too costly in general, I think? Maybe, instead of the denormalized stream, we could present a UnionPositionsIntervalIterator, which has multiple subs, where each sub is its own PositionIntervalIterator? This way eg a NEAR query could filter these subs in parallel (like a merge sort) looking for a match, and (I think) then presenting its own union iterator to whoever consumes it? Ie it'd only let through those positions of each sub that satisfied the NEAR constraint.
14. {quote} And it's finally single source... caller must say up-front (when pulling the scorer) if it will want positions (and, separately, also payloads – great). {quote} Does it make sense that we could just want AttributeSources as we go here?
15. {quote} How does/should scoring work? EG do the SpanQueries score according to the details of which position intervals match? {quote} I didn't pay any attention to scoring yet. IMO scoring should be left to the query which is using the positions so a higher level query like a NearQuery could just put a custom scorer on top of a boolean conjunction and apply its own proximity based score. This should be done after we have the infrastructure to do it. I think that opens up some nice scoring improvements. I am not sure if we should add proximity scoring to existing queries, I rather aim towards making it easy to customize. {quote} The part I'm wondering about is what API we should use for communicating positions of the sub scorers in a BooleanQuery to consumers like position filters (for matching) or eg Highlighter (which really should be a core functionality that works w/ any query). Multiplying out ("denormalizing") all combinations (into a flat stream of PositionIntervals) is going to be too costly in general, I think? {quote} I thought about that for a while and I think we should enrich the PosIntervalIterator API to enable the caller to pull the actual subintervals instead of an Interval from the next method. Something like this: {code} public abstract class PositionIntervalIterator implements Serializable{ public abstract PositionInterval next() throws IOException; /** *Returns all sub interval for the next accepted interval. **/ public abstract PositionIntervalIterator nextSubIntervals() throws IOException; public abstract PositionIntervalIterator[] subs(boolean inOrder); {code} so that if you are interested in the plain positions for eventually each term like highlighting you can pull them per match occurence. That way you have positional matching and you can iterate the subs. {quote} Maybe, instead of the denormalized stream, we could present a UnionPositionsIntervalIterator, which has multiple subs, where each sub is its own PositionIntervalIterator? This way eg a NEAR query could filter these subs in parallel (like a merge sort) looking for a match, and (I think) then presenting its own union iterator to whoever consumes it? Ie it'd only let through those positions of each sub that satisfied the NEAR constraint. {quote} I don't get that entirely ;) bq. Does it make sense that we could just want AttributeSources as we go here? you mean like we are not extending Scorer but add an AttributeSource to it? I think this is really a core API and should be supported directly
16. What should I do for it?
17. it doesn't seem that this issue is worth staying so tight coupled to the bulkpostings branch. I originally did this on bulk postings since it had support for positions in termscorer (new bulk API) where on trunk we don't have positions there at all. Yet, I think bulk postings should be

sorted out separately and we should rather move this over to trunk. On trunk we can get rid of all the low level bulk API hacks in the patch. the only thing that is missing here is a TermScorer that can score based on positions / payloads. I think since we have ScoreContext and how this works here in the patch we can simply implement a TermScorer that works on DocsEnumAndPositions and swap it in once positions are requested. I think I can move this over to trunk soon.

18. here is a patch that applies to trunk. I added a simple maybe slowish PositionTermScorer that is used when pos are required. This is really work in progress but I am uploading it just in case somebody is interested.
19. some more cleanups, all tests pass now on trunk
20. I've been fiddling with highlighter performance, and this looks a great step towards being able to do hl in an integrated way that doesn't require a lot of post-hoc recalculation etc. I worked up a hackish highlighter that uses it as a POC, partly just as a way of understanding what you've done, but this could eventually become usable. Here are a few comments: I found it convenient to add: {{boolean Collector.needsPositions() and needsPayloads()}} and modified {{IndexSearcher.search(AtomicReaderContext[] leaves, Weight weight, Filter filter, Collector collector)}} to set up the ScorerContext accordingly And then I am accessing the scorer.positions() from Collector.collect(), which I think is a very natural use of this API? At least it was intuitive for me, and I am pretty new to all this. I think that when it comes to traversing the tree of PositionsIntervalIterators, the API you propose above might have some issues. What would the status of the returned iterators be? Would they have to be copies of some sort in order to preserve the state of the iteration (so scoring isn't impacted by some other consumer of position intervals)? The iterators that are currently in flight shouldn't be advanced by the caller usually (ever?), or else the state of the dependent iterator (the parent) won't be updated correctly, I think? I wonder if (1) you could add {{PositionInterval PositionIntervalIterator.current()}} and (2) return from subs() and nextSubIntervals() some unmodifiable wrappers - maybe a superclass of PII that would only provide current() and subs(), but not allow advancing the iterator. I hope you'll be able to pick it up again soon, Simon!
21. Hey Mike, great to see interest here! :) bq. boolean Collector.needsPositions() and needsPayloads() +1 that makes lots of sense. Let me give you some insight of the current patches state. This whole thing is still a prototype and needs lots of cleanups all over the place. I moved it to trunk lately since I don't want to wait for bulkpostings to move forward. I think there are lots of perf impacts with its current state but eventually I think it will be much better, more powerful and cleaner than spans after all. bq. And then I am accessing the scorer.positions() from Collector.collect(), which I think is a very natural use of this API? At least it was intuitive for me, and I am pretty new to all this. this is one way of doing it for sure. The other way would be to wrap the top level scorer and do your work in there with a PositionScoringQueryWrapper or something like that which would set up the ScorerContext for you. The main question is what you want to do with positions. For matching based on positions you have to use some scorer I guess since you need to check every document if it is within your position constraints, something like near(a AND b). If you want to boost based on your positions I think you need to do a 2 phase collection, Phase 1 simply running the query collecting n + X results and Phase 2 re-ranking the results from Phase 1 by pulling the positions. bq. I think that when it comes to traversing the tree of PositionsIntervalIterators, the API you propose above might have some issues I agree this is very flaky right now and I only tried to mimic the spans behavior here to show that this is as powerful as spans for now. But eventually we need a better API for this, so its good you are jumping in with a usecase! bq. What would the status of the returned iterators be? currently if you pull an iterator you are depending on the state of your scorer. Let me give you an example on TermScorer, if you are on document X you can iterate the positions for this document if you exhaust them or not once the scorer is advanced your PositionInterator points to the documents position you advanced to. The same is true for all other Scorers that expose positions. Yet, some problems arise here with BooleanScorer (in contrast to BooleanScorer2) since it reads documents in blocks which makes it very hard (nearly impossible) to get efficient positions for this scorer (its used for OR queries only with NOT clauses < 32). So PositionsInterators are never preserve positions for a document you pulled the interval for. You can basically pull the iterator only once and keep it until you scorer is exhausted. Bottom line here is that you are depending on the DocsAndPositionsEnum your TermScorer is using. Once this is advanced your positions are advanced too. We could think of a separate Enum here that advances independently, hmm that could actually work too, lets keep that in mind. bq. (so scoring isn't impacted by some other consumer of position intervals) there should be only one consumer really. Which usecase have you in mind where multiple consumers are using the iterator? bq. PositionInterval PositionIntervalIterator.current() what is the returned PI here again? In the TermScorer case that is trivial but what would a BooleanSocorer return here? bq. (2) return from subs() and nextSubIntervals() some unmodifiable wrappers - maybe a superclass of PII that would only provide current() and subs(), but not allow advancing the iterator. I think that could make sense but let me explain the reason why this is there right now. So currently a socrer has a defined PositionIterator which could be a problem later. for instance I want to have the minimal positions interval (ordered) of all boolean clauses for query X but for query Y I want the same interval unorderd (out of order) I need to replace the logic in the scorer somehow. So to make that more flexible I exposed all subs here so you can run your own alg. I would love to see better solutions since I only hacked this up in a couple of days though. Currently this patch provides an AND (ordered & un-ordered) and a BLOCK PositionIterator based on this paper http://vigna.dsi.unimi.it/ftp/papers/EfficientAlgorithmsMinimalIntervalSemantics while the OR implementation is still missing so if you want to jump on that issue and help there is tons of space for improvements. Eventually I think we can leave spans as they are right now and concentrate on the API / functionality, making things fast under the hood can be done later but getting things right to be flexible is the most important part here. Mike, would you be willing to upload a patch for your hacked collector etc to see what you have done? bq. I hope you'll be able to pick it up again soon, Simon! I would love to ASAP, currently I have so much DocValues stuff todo so this might take a while until I get back to this.
22. bq. there should be only one consumer really. Which usecase have you in mind where multiple consumers are using the iterator? I guess I am coming at this from the perspective of a Highlighter; the Highlighter wants to iterate over the top-level Scorer, finding each of its matching positions, and then for each of those, it wants to iterate over all the individual terms' positions. Possibly some clever HL of the future will be interested in intermediate-level nodes in the tree as well, like highlighting a near-span, or coalescing phrases. The problem I see is that with the current API the only way to retrieve the lower-level positions is to advance their iterators, but if that is done directly (without the knowledge of the enclosing scorer and its iterator), the scoring will be messed up. I guess that's what I meant by multiple consumers - of course you are right, there should be only one "writer" consumer that can advance the iteration. My idea is that there could be many readers, though. In any case, I think it is typical for an iterator that you can read the current position as many times as you want, rather than "read once" and expect the caller to cache the value? bq. what is the returned PI here again? In the TermScorer case that is trivial but what would a BooleanScorer return here? It has its own PI right? I think it is the minimum interval containing some terms that satisfy the boolean conditions. bq. I think that could make sense but let me explain the reason why this is there right now. So currently a socrer has a defined PositionIterator which could be a problem later. for instance I want to have the minimal positions interval (ordered) of all boolean clauses for query X but for query Y I want the same interval unorderd (out of order) I need to replace the logic in the scorer somehow. So to make that more flexible I exposed all subs here so you can run your own alg. I would love to see better solutions since I only hacked this up in a couple of days though. Hmm I haven't yet looked at how BooleanScorer2 and BooleanScorer works, but I understand there is some additional complexity there. Perhaps if the only distinction is order/unordered there might be a special case for that when you create the Scorer, rather than exposing internals to the caller? But I don't know - would have to understand this better. Maybe there are other cases where that could be needed. bq. Mike, would you be willing to upload a patch for your hacked collector etc to see what you have done? The PosHiglighter is a bit messy - filled with debugging and testing code and so on, and it's also slow because of the need to match positions->offsets in kind of a gross way.. Robert M had an idea for storing this mapping in the index which would

improve things there, but I haven't done that. In any case, I'll be happy to share the patch when I get back home and can clean it up a bit. Maybe if I have a chance I will look into implementing OR-queries - I stumbled on that limitation right away!

23. Attaching a patch with a simple highlighter using the positions() iterators. Includes a PositionTreeIterator for pulling out leaf positions. The names are getting a bit long: I almost wrote PositionIntervalIteratorTree? Maybe a PositionIntervalIterator could just be a Positions? The variables are all called positions...
24. updated patch provides positions() for Boolean OR (disjunction) queries, but only with n should match = 1. I refactored the ConjunctionPositionIterator and its Queue in order to do this without a lot of cut-and-paste.
25. Updated PosHighlighter patch that actually works :) PosCollector now caches positions when collecting docs. This way highlighting can be decoupled, and doesn't have to be performed on docs that will eventually drop off the top n list. This seems to work ok, but it would be nice to come up with a way to make this usable with existing collectors.
26. bq. So PositionsInterators are never preserve positions for a document you pulled the interval for. You can basically pull the iterator only once and keep it until you scorer is exhausted. Bottom line here is that you are depending on the DocsAndPositionsEnum your TermScorer is using. Once this is advanced your positions are advanced too. We could think of a separate Enum here that advances independently, hmm that could actually work too, lets keep that in mind. So after working with this a bit more (and reading the paper), I see now that it's really not necessary to cache positions in the iterators. So never mind all that! In the end, for some uses like highlighting I think somebody needs to cache positions (I put it in a ScorePosDoc created by the PosCollector), but I agree that doesn't belong in the "lower level" iterator. bq. Eventually I think we can leave spans as they are right now and concentrate on the API / functionality, making things fast under the hood can be done later but getting things right to be flexible is the most important part here. As I'm learning more, I am beginning to see this is going to require sweeping updates. Basically everywhere we currently create a DocsEnum, we might now want to create a DocsAndPositionsEnum, and then the options (needs positions/payloads) have to be threaded through all the surrounding APIs. I wonder if it wouldn't make sense to encapsulate those options (needsPositions/needsPayloads) in some kind of EnumConfig object. Just in case, down the line, there is some other information that gets stored in the index, and wants to be made available during scoring, then the required change would be much less painful to implement. I'm thinking for example (Robert M's idea), that it might be nice to have a positions->offsets map in the index (this would be better for highlighting than term vectors). Maybe this would just be part of payload, but maybe not? And it seems possible there could be other things like that we don't know about yet?
27. Mike, its so awesome that you help here. I will be back on wednesday and post comments / suggestions then. simon
28. Yeah that would be great - I hope I'm not being too irritating -- enjoy your vacation! I worked on using the standard Highlighter with a TokenStream built on a PosIterator, and the results are promising in terms of performance, but I think I am still grappling with the right way to retrieve positions.
29. hey Mike, I applied all your patches and walked through, this looks great. I mean this entire thing is far from committable but I think we should take this further and open a branch for it. I want to commit both your latest patch and the highlighter prototype and work from there. {quote}So after working with this a bit more (and reading the paper), I see now that it's really not necessary to cache positions in the iterators. So never mind all that! In the end, for some uses like highlighting I think somebody needs to cache positions (I put it in a ScorePosDoc created by the PosCollector), but I agree that doesn't belong in the "lower level" iterator.{quote} after looking into your patch I think I understand now what is needed to enable low level stuff like highlighting. what is missing here is a positions collector interface that you can pass in and that collects positions on the lowest levels like for pharses or simple terms. The PositionIterator itself (btw. i think we should call it Positions or something along those lines - try to not introduce spans in the name :)) should accept this collector and simply call back each low level position if needed. For highlighting I think we should also go a two stage approach. First stage does the matching (with or without positions) and second stage takes the first stages resutls and does the highlighting. that way we don't slow down the query and the second one can even choose a different rewrite method (for MTQ this is needed as we don't have positions on filters) {quote} As I'm learning more, I am beginning to see this is going to require sweeping updates. Basically everywhere we currently create a DocsEnum, we might now want to create a DocsAndPositionsEnum, and then the options (needs positions/payloads) have to be threaded through all the surrounding APIs. I wonder if it wouldn't make sense to encapsulate those options (needsPositions/needsPayloads) in some kind of EnumConfig object. Just in case, down the line, there is some other information that gets stored in the index, and wants to be made available during scoring, then the required change would be much less painful to implement. {quote} what do you mean by sweeping updates? For the enum config I think we only have 2 or 3 places where we need to make the decision. 1. TermScorer 2. PhraseScorer (maybe 2. goes away anyway) so this is not needed for now I think? {quote} I'm thinking for example (Robert M's idea), that it might be nice to have a positions->offsets map in the index (this would be better for highlighting than term vectors). Maybe this would just be part of payload, but maybe not? And it seems possible there could be other things like that we don't know about yet? {quote} yeah this would be awesome... next step :)
30. {quote} For highlighting I think we should also go a two stage approach. First stage does the matching (with or without positions) and second stage takes the first stages resutls and does the highlighting. that way we don't slow down the query and the second one can even choose a different rewrite method (for MTQ this is needed as we don't have positions on filters) {quote} I think this would be a good approach, its the same algorithm really that you generally want for positional scoring: score all the docs the 'fast' way then reorder only the top-N (e.g. first two pages of results), which will require using the positioniterator and doing some calculation that you typically add to the score. So if we can generalize this in a way where you can do this in your collector, I think it would be reusable for this as well.
31. bq. what do you mean by sweeping updates? I meant adding positions to filters would be a sweeping update. But it sounds as if the idea of rewriting differently is a better approach (certainly much less change). bq. For highlighting I think we should also go a two stage approach. I think I agree. The only possible trade-off that goes the other way is in the case where you have the positions available already during initial search/scoring, and there is not too much turnover in the TopDocs priority queue during hit collection. Then a Highlighter might save some time by not re-scoring and re-iterating the positions if it accumulated them up front (even for docs that were eventually dropped off the queue). I think it should be possible to test out both approaches given the right API here though? The callback idea sounds appealing, but I still think we should also consider enabling the top-down approach: especially if this is going to run in two passes, why not let the highlighter drive the iteration? Keep in mind that positions consumers (like highlighters) may possibly be interested in more than just the lowest-level positions (they may want to see phrases, eg, and near-clauses - trying to avoid the s-word). Another consideration is ordering. I think (?) that positions are retrieved from the index in document order. This could be a natural order for many cases, but score order will also be useful. I'm not sure whose responsibility the sorting should be. Highlighters will want to be able to optimize their work (esp for very large documents) by terminating after considering only the first N matches, where the ordering could either be score or document-order. I'm glad you will create a branch - this patch is getting a bit unwieldy. I think the PosHighlighter code should probably (?) end up as test code only - I guess we'll see. It seems like we could get further faster using the existing Highlighter, with a positions-based TokenStream; I'll post a patch once the branch is in place.
32. {quote} I think I agree. The only possible trade-off that goes the other way is in the case where you have the positions available already during initial search/scoring, and there is not too much turnover in the TopDocs priority queue during hit collection. Then a Highlighter might save some time by not re-scoring and re-iterating the positions if it accumulated them up front (even for docs that were eventually dropped off the queue). I think it should be possible to test out both approaches given the right API here though? {quote} Yes, I think we should go and provide both possibilities here. {quote} The callback idea sounds appealing, but I still think we should also consider

enabling the top-down approach: especially if this is going to run in two passes, why not let the highlighter drive the iteration? Keep in mind that positions consumers (like highlighters) may possibly be interested in more than just the lowest-level positions (they may want to see phrases, eg, and near-clauses - trying to avoid the s-word). {quote} I am not sure if I understand this correctly. I think the collector should be some kind of a visitor that walks down the query/scorer tree and each scorer can ask if it should pass the current positions to the collector something like this: {code} class PositionCollector { public boolean register(Scorer scorer) { if(interestedInScorere(scorere)) { // store infor about the scorer return true; } return false; } /* * Called by a registered scorer for each position change */ public void nexPosition(Scorer scorer) { // collect positions for the current scorer } } {code} that way the iteration process is still driven by the top-level consumer but if you need information about intermediate positions you can collect them. {quote} Another consideration is ordering. I think that positions are retrieved from the index in document order. This could be a natural order for many cases, but score order will also be useful. I'm not sure whose responsibility the sorting should be. Highlighters will want to be able to optimize their work (esp for very large documents) by terminating after considering only the first N matches, where the ordering could either be score or document-order. {quote} so the order here depends on the first collector I figure. the usual case it that you do your search and retrieve the top N documents (those are also the top N you want to highlight right?) then you pass in your top N and do the highlighting collection based on those top N. In that collection you are not interested all matches but only in the top N from the previous collection. The simplest yet maybe not the best way to do this is using a simple filter that is build from the top N docs. I will go ahead and create the branch now

33. bq. I am not sure if I understand this correctly. I think the collector should be some kind of a visitor that walks down the query/scorer tree and each scorer can ask if it should pass the current positions to the collector something like this: Yes that sounds right Re: ordering; I was concerned about the order in which the positions are iterated within each document, not so much the order in which the documents are returned. I think this is an issue for the highlighter mostly, which can "score" position-ranges in the document so as to return the "best" snippet. This kind of score may be built up from tfidf scores for each term, proximity, length of the position-ranges and so on.
34. mike I created a branch here: https://svn.apache.org/repos/asf/lucene/dev/branches/positions
35. Not sure what to call these patch files now? This is relative to the positions branch... * added PosTokenStream: wraps a PositionIntervalIterator and PositionOffsetMapper * extracted PositionOffsetMapper into standalone class * changed PosHighlighterTest to use Highlighter; may abandon PosHighlighter? * Added support for minNrShouldMatch in DisjunctionSumScorer.positions() and ConjunctionPositionIterator; plus a test in PosHighlighterTest * ConstantScoreQuery.positions() now calls its wrapped Scorer's positions() * PosHighlighterTest tests highlighting WildcardQuery by rewriting MTQs
36. To make further progress, I think we need to resolve the position API. The testMultipleDocumentsOr test case illustrates the problem with the approach I was trying: walking the PositionIterator tree when collecting documents. Something like the PositionCollector API could work, but I think we still need to solve the problem Mike M alluded to back at the beginning: bq. ... a NEAR query could filter these subs in parallel (like a merge sort) looking for a match, and (I think) then presenting its own union iterator to whoever consumes it? Ie it'd only let through those positions of each sub that satisfied the NEAR constraint. We want to highlight positions that explain why the document matches the query. Not all terms that match the term queries will count - some of them should be "filtered out" by near-conditions; ie in a PhraseQuery, matching terms not in the phrase should not be highlighted. I think if I just register a callback with the sub-scorers (scoring terms), I would see *all* the terms, right? Also, I wonder if callers could use the existing Scorer.ScorerVisitor to walk the Scorer tree and register interest in positions with a scorer?
37. I made some progress with a Collector-based positions API; my tests (and existing tests) are passing now. I am getting all the right positions back for highlighting from a variety of queries, so I think the basic idea is workable. But there are a bunch of things I don't like still (and more queries/scorers to instrument); needs more work, for sure. The basic idea is to extend Collector so it can accept positions. Collectors simply return true from Collector.needsPositions() and implement collectPositions(Scorer, PositionIntervalIterator). The way of hooking this up is the easiest possible for now; I added Scorer.setPositionCollector(Collector); Collectors would call it in Collector.setScorer(). This approach forced one change that probably isn't good for performance, so we might want to revisit this, and "hook up" the Scorer.positionCollector during construction, say with the register() idea you mentioned. The problem currently is that in DisjunctionSumScorer, there is some initialization that happens (advancing all the iterators) during the constructor, before setScorer() is ever called. I've changed it to happen lazily, but this introduces an extra if(initialized) check into every call to advance() and nextDoc(). Another problem with this callback approach (from the performance perspective) is that you end up paying the cost to check whether there is a positionCollector registered in a fairly tight inner loop (while matching/scoring), I think, which you'd rather not do if there is no interest in positions. It would probably be better to introduce some kind of Scorer wrapper that calls collectPositions() and only gets created if needed. Another thought I had was to push the callback into PositionIntervalIterator, but somehow someone has to know to actually iterate over those iterators; if it's the Scorer, you have the same problem as now, but the Collector can't do it (in collect(), say) because it is often too late then; the Scorer will have advanced its internal state (and thus its positions() iterator) to the next document. The core problem solved here is how to report positions that are not consumed during scoring, and also those that are, but not to report positions that don't contribute to the query match (terms outside their phrase). In order to do this, different Scorers do different things: the Boolean family of scorers (and wrapping scorers like ConstantScorer) simply pass any positionCollector down to their child Scorers, since they effectively want to report all positions found. PositionTermScorer reports its positions, naturally. The interesting case is PositionFilterScorer, which filters its child Scorers. I added PositionIntervalIterator.getTermPositions() to enable this; this walks the tree of position iterators and returns a snapshot of their current state (as another iterator) so the consumer can retrieve all the term positions as filtered by intermediate iterators without advancing them. There are a few (11) failing tests with this branch+patch (ran lucene tests only), but they seem unrelated (TestFlushByRamOrCountsPolicy has 5, eg) I am ignoring? Simon - thanks for setting up the positions branch - I would really appreciate it if you have the time to review this because I think there is headway, but I'm also sure there is lots of room for improvement.
38. {quote} We want to highlight positions that explain why the document matches the query. Not all terms that match the term queries will count - some of them should be "filtered out" by near-conditions; ie in a PhraseQuery, matching terms not in the phrase should not be highlighted. I think if I just register a callback with the sub-scorers (scoring terms), I would see all the terms, right? {quote} this is why I think we should add a dedicated collector API (ie. not part of Collector maybe an interface?). the current api gives you a "view" for each match meaning that once you advance the iterator you get the positions for the "current" positional match. I think the caller should also drive the collection of intermediate positions / intervals. The big challenge here is to collect the positions you are interested in efficiently. I agree that the if(foo==null) is a problem as long as foo is not final so maybe we should try to make them final and make the pos collector part of the scorer setup (just a thought), we could do that using a ScorerContext for instance. {quote} make further progress, I think we need to resolve the position API. The testMultipleDocumentsOr test case illustrates the problem with the approach I was trying: walking the PositionIterator tree when collecting documents. Something like the PositionCollector API could work, but I think we still need to solve the problem Mike M alluded to back at the beginning: {quote} Agreed we should work on the API. I looked at your patch and some changes appear to be not necessary IMO. Like the problems in testMultipleDocumentsOr are not actually a problem if we sketch this out properly. As I said above if the collector is part of the initialization we can simply pass them to the leaves or intermediate scorers and collect safely even if scorers are advanced. Since during Documents collection the view should be stable, right? So bottom line here is that we need an api that is capable of collecting fine grained parts of the scorer tree. The only way I see doing this is 1. have a subscribe / register method and 2. do this subscription during scorer creation. Once we have this we can implement very simple collect methods that

only collect positions for the current match like in a near query, while the current matching document is collected all contributing TermScorers have their positioninterval ready for collection. The collect method can then be called from the consumer instead of in the loop this way we only get the positions we need since we know the document we are collecting. bq. The core problem solved here is how to report positions that are not consumed during scoring, and also those that are, this can be solved by my comment above? {quote} The interesting case is PositionFilterScorer, which filters its child Scorers. I added PositionIntervalIterator.getTermPositions() to enable this; this walks the tree of position iterators and returns a snapshot of their current state (as another iterator) so the consumer can retrieve all the term positions as filtered by intermediate iterators without advancing them. {quote} this would work the same way ey? We register during setup, something like {code}void PositinoCollector#registerScorer(Scorer){code} then we can decide that if we need that scorer or rather its positions for collection or not. The entire iteration should only be driven by the top-level consumer, if you advance the iterator on an intermediate iterator you might break some higher level algs. like conjunction / disjunction though. So lets drive this further, lets say we have all collectors that we are interested in, when should we collect positions? I think the top level consumer should 1. advance the positions 2. call collect on the scorers we are interested. While I talk about this I start realizing that it might even be easier that this if we walk the PositionInterator tree rather than the scorer tree and collect the positin iterators from there. This is already possible with the subs() call right? What we essentially need is a method that returns the current interval for each of the iterators. It still might be needed to have a collect method on the iterator so that something like Conjunctions can call collect on the subs if needed? Oh man this is all kind of tricky ey :) bq. There are a few (11) failing tests with this branch+patch (ran lucene tests only), but they seem unrelated (TestFlushByRamOrCountsPolicy has 5, eg) I am ignoring? I don't see anything failing... can you attach a file with the failures?

39. bq. if(foo==null) is a problem as long as foo is not final so maybe we should try to make them final and make the pos collector part of the scorer setup (just a thought), we could do that using a ScorerContext for instance. Yes, agreed. I just wanted to implement something simple first. I think we can fix the setup problem separately from the actual collection/reporting of intervals. We can eventually undo the changes to DisjunctionSumScorer, and get rid of those extra if()s. Also, as you say the other tests can be made final if we do them during setup. bq. While I talk about this I start realizing that it might even be easier that this if we walk the PositionInterator tree rather than the scorer tree and collect the positin iterators from there. Did you look at SubTermPositionIterator (I think that's what I called it) and getTermPositions() yet? They are supposed to be providing pretty much just that capability. bq. Oh man this is all kind of tricky ey I tore my hair out all weekend and lost sleep! But I think it actually is where we want now, aside from the registration piece.
40. OK I think I brushed by some of your comments, Simon, in my hasty response, sorry. Here's a little more thought, I hope: bq. So bottom line here is that we need an api that is capable of collecting fine grained parts of the scorer tree. The only way I see doing this is 1. have a subscribe / register method and 2. do this subscription during scorer creation. Once we have this we can implement very simple collect methods that only collect positions for the current match like in a near query, while the current matching document is collected all contributing TermScorers have their positioninterval ready for collection. The collect method can then be called from the consumer instead of in the loop this way we only get the positions we need since we know the document we are collecting. I *think* it's necessary to have both a callback from within the scoring loop, and a mechanism for iterating over the current state of the iterator. For boolean queries, the positions will never be iterated in the scoring loop (all you care about is the frequencies, positions are ignored), so some new process: either the position collector (highlighter, say), or a loop in the scorer that knows positions are being consumed (needsPositions==true) has to cause the iteration to be performed. But for position-aware queries (like phrases), the scorer *will* iterate over positions, and in order to score properly, I think the Scorer has to drive the iteration? I tried a few different approaches at this before deciding to just push the iteration into the Scorer, but none of them really worked properly. Let's say, for example that a document is collected. Then the position consumer comes in to find out what positions were matched - it may already too late, because during scoring, some of the positions may have been consumed (eg to score phrases)? It's possible I may be suffering from some delusion, though :) But if I'm right, then it means there has to be some sort of callback mechanism in place *during scoring*, or else we have to resign ourselves to scoring first, and then re-setting and iterating positions in a second pass. I actually think that if we follow through with the registration-during-construction idea, we can have the tests done in an efficient way during scoring (with final boolean properties of the scorers), and it can be OK to have them in the scoring loop.
41. {quote} But if I'm right, then it means there has to be some sort of callback mechanism in place during scoring, or else we have to resign ourselves to scoring first, and then re-setting and iterating positions in a second pass. {quote} But I think this is what I think we want? If there are 10 million documents that match a query, but our priority queue size is 20 (1 page), we only want to do the expensive highlighting on those 20 documents. Its the same for the positional scoring, its too expensive to look at positions for all documents, so you re-order maybe the top 100 or so. Or maybe I'm totally confused by the comments!
42. bq. But I think this is what I think we want? If there are 10 million documents that match a query, but our priority queue size is 20 (1 page), we only want to do the expensive highlighting on those 20 documents. Yes - the comments may be getting lost in the weeds a bit here; sorry. I've been assuming you'd search once to collect documents and then search again with the same query plus a constraint to limited by gathered docids, with an indication that positions are required - this pushes you towards some sort of collector-style callback API. Maybe life would be simpler if instead you could just say getPositionIterator(docid, query). But that would force you actually into a third pass (I think), if you wanted positional scoring too, wouldn't it?
43. {quote} But that would force you actually into a third pass (I think), if you wanted positional scoring too, wouldn't it? {quote} I think thats ok? because the two things are different: * in general i think you want to rerank more than just page 1 with scoring, e.g. maybe 100 or even 1000 documents versus the 20 that highlighting needs. * for scoring, we need to adjust our PQ, resulting in a (possibly) different set of page 1 documents for the highlighting process, so if we are doing both algorithms, we still don't yet know what to highlight anyway. * if we assume we are going to add offsets (optionally) to our postings lists in parallel to the positions, thats another difference: scoring doesnt care about offsets, but highlighting needs them.
44. FWIW, I do think there are use cases where one wants positions over all hits (or most such that you might as well do all), so if it doesn't cause problems for the main use case, it would be nice to support it. In fact, in these scenarios, you usually care less about the PQ and more about the positions.
45. {quote} FWIW, I do think there are use cases where one wants positions over all hits (or most such that you might as well do all), so if it doesn't cause problems for the main use case, it would be nice to support it. In fact, in these scenarios, you usually care less about the PQ and more about the positions. {quote} I don't think this issue should try to solve that problem: if you are doing that, it sounds like you are using the wrong Query!
46. bq. I don't think this issue should try to solve that problem: if you are doing that, it sounds like you are using the wrong Query! It's basically a boolean match on any arbitrary Query where you care about the positions. Pretty common in e-discovery and other areas. You have a query that tells you all the matches and you want to operate over the positions. Right now, it's a pain as you have to execute the query twice. Once to get the scores and once to get the positions/spans. If you have a callback mechanism, one can do both at once.
47. I don't understand the exact use case... it still sounds like the wrong query? What "operating" over the positions do you need to do?
48. In the cases where I've both done this and seen it done, you often have an arbitrary query that matches X docs. You then want to know where exactly the matches occur and then you often want to do something in a window around those matches. Right now, w/ Spans, you have to run the query once to get the scores and then run a second time to get the windows. The times I've seen it, the result is most often given to some downstream process that does deeper analysis of the window, so in these cases X can be quite large (1000's if not more). In

those cases, some people care about the score, some do not. For instance, if one is analyzing all the words around the name of a company, you search term would be the company name and you want to iterate over all the positions where it matched, looking for other words near it (perhaps sentiment words or other things)

49. {quote} In those cases, some people care about the score, some do not. For instance, if one is analyzing all the words around the name of a company, you search term would be the company name and you want to iterate over all the positions where it matched, looking for other words near it {quote} Grant, I'm not sure this sounds like an inverted index is even the best data structure for what you describe. I just don't want us to confuse the issue with the nuking of spans/speeding up highlighting/enabling positional scoring use cases which are core to search.
50. bq. I'm not sure this sounds like an inverted index is even the best data structure for what you describe The key is you usually have a fairly complex Query to begin with, so I do think it is legitimate and it is the right data structure. It is always driven by the search results. I've seen this use case multiple times, where multiple is more than 10, so I am pretty convinced it is beyond just me. I think if you are taking away the ability to create windows around a match (if you read my early comments on this issue I brought it up from the beginning), that is a pretty big loss. I don't think the two things are mutually exclusive. As long as I have a way to get at the positions for all matches, I don't care that it. A "collector" type callback interface or a way for one to iterate all positions for a given match should be sufficient. That being said, if Mike's comments about a collector like API are how it is implemented, I think it should work. In reality, I think one would just need a way to, for whatever number of results, be told about positions as they happen. Naturally, the default should be to only do this after the top X are retrieved, when X is small, but I could see implementing it in the scoring loop on certain occasions (and I'm not saying Lucene need have first order support for that). As long as you don't preclude me from doing that, it should be fine. I'll try to find time to review the patch in more depth in the coming day or so.
51. {quote} The key is you usually have a fairly complex Query to begin with, so I do think it is legitimate and it is the right data structure. {quote} Really, just because its complicated? Accessing other terms 'around the position' seems like accessing the document in a non-inverted way. {quote} I've seen this use case multiple times, where multiple is more than 10, so I am pretty convinced it is beyond just me. {quote} Really? If this is so common, why do the spans get so little attention? if the queries are so complex, how is this even possible now given that spans have so many problems, even basic ones (e.g. discarding boosts) If performance here is so important towards looking at these 'windows around a match' (which is gonna be slow as shit via term vectors), why don't I see codecs that e.g. deduplicate terms and store pointers to the term windows around themselves in payloads, and things like that for this use case? I don't think we need to lock ourselves into a particular solution (such as per-position callback API) for something that sounds like its really slow already.
52. bq. Really, just because its complicated? Accessing other terms 'around the position' seems like accessing the document in a non-inverted way. Isn't that what highlighting does? This is just highlighting on a much bigger set of documents. I don't see why we should prevent users from doing it just b/c you don't see the use case. bq. Really? If this is so common, why do the spans get so little attention? if the queries are so complex, how is this even possible now given that spans have so many problems, even basic ones (e.g. discarding boosts) Isn't that the point of this whole patch? To bring "spans" into the fold and treat as first class citizens? I didn't say it happened all the time. I just said it happened enough that I think it warrants being covered before one "nukes spans". bq. If performance here is so important towards looking at these 'windows around a match' (which is gonna be slow as shit via term vectors), why don't I see codecs that e.g. deduplicate terms and store pointers to the term windows around themselves in payloads, and things like that for this use case? Um, b/c it's open source and not everything gets implemented the minute you think of it? bq. I don't think we need to lock ourselves into a particular solution (such as per-position callback API) for something that sounds like its really slow already. Never said we did.
53. {quote} Isn't that what highlighting does? This is just highlighting on a much bigger set of documents. I don't see why we should prevent users from doing it just b/c you don't see the use case. {quote} well it is different: I'm not saying we should prevent users from doing it, but we shouldn't slow down normal use cases either: I think its fine for this to be a 2-pass operation, because any performance differences from it being 2-pass across many documents are going to be completely dwarfed by the term vector access!
54. Yeah, I agree. I don't want to block the primary use case, I'm just really hoping we can have a solution for the second one that elegantly falls out of the primary one and doesn't require a two pass solution. You are correct on the Term Vec access, but for large enough sets, the second search isn't trivial, even if it is dwarfed. Although, I think it may be possible to at least access them in document order.
55. I hope you all will review the patch and see what you think. My gut at the moment tells me we can have it both ways with a bit more tinkering. I think that as it stands now, if you ask for positions you get them in more or less the most efficient way we know how. At the moment there is some performance hit when you don't want positions, but I think we can deal with that. Simon had the idea we could rely on the JIT compiler to optimize away the test we have if we set it up as a final false boolean (totally do-able if we set up the state during Scorer construction), which would be great and convenient. I'm no compiler expert, so not sure how reliable that is - is it? But we could also totally separate the two cases (say with a wrapping Scorer? - no need for compiler tricks) while still allowing us to retrieve positions while querying, collecting docs, and scoring.
56. bq.Yeah, I agree. I don't want to block the primary use case, I'm just really hoping we can have a solution for the second one that elegantly falls out of the primary one and doesn't require a two pass solution. You are correct on the Term Vec access, but for large enough sets, the second search isn't trivial, even if it is dwarfed. Although, I think it may be possible to at least access them in document order. Grant, as far as I understand your concerns I think they are addressed already. if you want to do span like (what spans does today) you can already do that. You can simply advance the iterator during search and get the matches / position. or do I misunderstand what you are saying...
57. Cool. I think as positions become first class citizens and as this stuff gets faster, we're going to see more and more use of positional information in apps, so it will likely become more common.
58. Mike & all other interested users :) I think I got around all the pre scorer creation collector setup etc. by detaching Scorer from Positions (and its iteration + collection) entirely. on the lowest level TermScorer now uses two enums, one for scoring (no positions) and one for the position iterator if needed. This change required some more upstream changes since the consumer now has to advance the positions to the next doc the scorer points to. Yet, this gives us some more freedom how and when to consume the positions. A wrapping scorer can still consume the positions or we can simply leave this to the collector. I think this gets reasonably closer to what we need while still pretty rough. Mike what do you think, does that help with highlighting? i also added two types of collectors, one collects only leaves (term positions) and the other collects the intermediate (composite) intervals. I call them by default without null checking, the default is simply an empty collector so hopefully the compiler will no-op this.
59. Looks good, Simon! So - working with two enums; one for basic scoring w/o positions, and one for gathering positions allows additional flexibility and cleaner separation between the position-aware code and the scorers, and makes it more straightforward to implement the desired API. We can now set up a PositionCollector (it's good to allow as separate from Collector) that collects both term positions and (separately) composite position intervals (like phrases, intervals containing conjoined terms, etc). Some will be reported naturally if a position-aware scorer consumes them; the collector can iterate through the remainder by calling collect() => actually I might suggest renaming PositionIntervalIterator.collect() to distribute(), to distinguish it from its counterpart, PositionCollector.collect(). Do you have any concern about the two iterators getting out of sync? I noticed the nocommit, I guess that's what you meant? What's the scope for mischief - should we be thinking about making it impossible for the user of the API get themselves in trouble? Say, for example, I call advanceTo(randomDocID) - I could cause my PositionFilterQuery to get out of whack, maybe? I am going to clean up the PosHighlighter tests a bit, get rid of dead code, etc., possibly add some tests for the composite interval stuff, and do a little benchmarking.

60. bq. actually I might suggest renaming PositionIntervalIterator.collect() to distribute(), to distinguish it from its counterpart, PositionCollector.collect(). how about gatherPositions() ? bq. Do you have any concern about the two iterators getting out of sync? I noticed the nocommit, I guess that's what you meant? Actually I am not super concerned about this. its all up to the API consumer. The nocommit is just a reminder that we need to fix this method (PII#doc()) to return the actual document the DocsAndPositionsEnum points to or rather the iterator points to right now. I think we should start sketching out the API and write some javadoc to make clear how things work. Beside working on highlighting I think we should also cut over remaining queries to positions and copy some of the span tests to positions (dedicated issue for this would be helpful this gets a little big). bq. should we be thinking about making it impossible for the user of the API get themselves in trouble? Say, for example, I call advanceTo(randomDocID) - I could cause my PositionFilterQuery to get out of whack, maybe? phew, I think we can work around this but we need to make sure we don't loose flexibility. Maybe we need to rethink how PositionFitlerQuery works. Lets leave that for later :) For spans I think we should move them to the new queries module and eventually out of core (we should have a new issue for this no?). For the position iterating stuff I think we can mainly concentrate on getting positions work and leave payloads for later. Further I think we should also open a ticket for highlighting as well as for positional scoring where we can add the 2 stage collector stuff etc. I will create a "positions branch" version so we can flag issues correctly. bq. I am going to clean up the PosHighlighter tests a bit, get rid of dead code, etc., possibly add some tests for the composite interval stuff, and do a little benchmarking. awesome, if you clean up the patch make sure we have the right headers in all new files and add @lucene.experimental to the classes. I want to commit our stage soonish (once you cleaned it up) and continue with fine grained issues. I am glad that you spend so much time this man! Making positions first class citizens is very important and it will pave the way to get rid of spans eventually.
61. bq. how about gatherPositions() ? Seems OK; anything but collect! bq. I want to commit our stage soonish (once you cleaned it up) and continue with fine grained issues. Good - yes it would be helpful to split out some issues now: finish up API, more queries, positional scoring and highlighting? Do you have a plan for PhraseQuery? It looks scary to me! API note: I wonder if it still makes sense to use Collector.needsPositions() as the trigger for requesting positions - if Collectors are not really what end up doing the gathering of positions? I ran some quick benchmarks, and the results are promising - highlighting with positions is more than 10x faster than regular highlighting and slightly (10-15%?) faster than fast vector highlighter. While doing this I found a bug in DisjunctionPositionIterator.advanceTo() - it could return a docId that none of its subqueries matched, so you'd eventually get a NPE. Fix is in the patch I'm uploading. Oh yes - also added SingleMatchScorer.positions() bq. I am glad that you spend so much time this man! Making positions first class citizens is very important and it will pave the way to get rid of spans eventually. This is exciting, I think! Glad you are able to work on it again. I will probably slow down a bit since I am traveling for a few days, but I'll be back next week.
62. bq. Good - yes it would be helpful to split out some issues now: finish up API, more queries, positional scoring and highlighting? Do you have a plan for PhraseQuery? It looks scary to me! I committed the latest patch with some more cleanups, headers, test etc. I also started working on the PhraseQuery, exact case works already but I need some more time and brain cycles for the sloppy part. (it is scary) I am going to open a new issue for this now. bq. ran some quick benchmarks, and the results are promising - highlighting with positions is more than 10x faster than regular highlighting and slightly (10-15%?) f awesome I can't wait to do some more benchmarks though. bq. This is exciting, I think! Glad you are able to work on it again. I will probably slow down a bit since I am traveling for a few days, but I'll be back next week. same here, I was traveling this week and next week again so lets see how much progress we can make here. :) looks good so far
63. mike, I created subtasks (listed below the attached files) for this issue since this gets kind of huge. I also made you a JIRA contributor so you can assign issues to yourself. Please don't hesitate to open further issues / subtasks as we proceed.
64. hey folks, due to heavy modifications on trunk I had almost no choice but creating a new branch and manually move over the changes via selective diffs. the branch is now here: https://svn.apache.org/repos/asf/lucene/dev/branches/LUCENE-2878 the current state of the branch is: it compiles :) lots of nocommits / todos and several tests failing due to not implemented stuff on new specialized boolean scorers. Happy coding everybody!
65. Patch changing the Scorer#positions() signature to Scorer#positions(needsPayloads, needsOffsets), and implementing the payload passing functionality. All Span payload tests now pass.
66. Alan this is awesome. I fixed some compile errors in solr and modules land and test-core passes! I will go ahead and commit this to the branch. I think next is either fixing all missing queries (PhraseScorer and friends) or exposing offsets. Feel free to create a subtask for the offsets though. My first step here would be to put the offsets next to PositionInterval#begin/end as offsetBegin/End. This is more of a coding task than anything else in the beginning since this info needs to be transported up the PositionIntervalIterator "tree" during execution. on the lowest level (TermPositions) you can simply assign it via DocsAndPositionsEnum#start/endOffset() since that returns -1 if offsets are not indexed. thanks & good job!
67. Alan this is awesome. I fixed some compile errors in solr and modules land and test-core passes! I will go ahead and commit this to the branch. I think next is either fixing all missing queries (PhraseScorer and friends) or exposing offsets. Feel free to create a subtask for the offsets though. My first step here would be to put the offsets next to PositionInterval#begin/end as offsetBegin/End. This is more of a coding task than anything else in the beginning since this info needs to be transported up the PositionIntervalIterator "tree" during execution. on the lowest level (TermPositions) you can simply assign it via DocsAndPositionsEnum#start/endOffset() since that returns -1 if offsets are not indexed. thanks & good job!
68. I'll start on the offsets - some relatively mindless coding is probably about where I'm at today, and the brief look I had at ExactPhraseScorer scared me a bit.
69. its fantastic how you guys have brought this back from the dead!
70. Patch against the branch head, adding offsets to PositionInterval. Includes a couple of test cases showing that it works for basic TermQueries.
71. The patch also includes an @Ignored test case for BooleanQueries, as this didn't behave in the way I expected it to. At the moment, ConjunctionPositionIterator returns PositionIntervals that span all the parent query's subclauses. So searching for 'porridge' and 'nine' returns an Interval that starts at 'porridge' and ends at 'nine'. I would have expected this instead to return two separate intervals - if we want phrase-type intervals, then we can combine the individual intervals with a Filter of some kind. But I may just be misunderstanding how this is supposed to work.
72. hey alan, great job.. your are getting up to speed. I fixed that testcase (the boolean one) since in the conjunction case you have to consume the conjunction positions/offsets ie. the intervals given by the term matches. I also fixed the license header in that file and brought the highlighter prototype test back. I will commit this to the branch now. wow man this makes me happy! Good job.
73. I messed up the last patch - here is the actual patch.
74. oh btw. All tests on the branch pass now :)
75. I think my next step is to have a go at implementing ReqOptSumScorer and RelExclScorer, so that all the BooleanQuery cases work. Testing it via the PosHighlighter seems to be the way to go as well. This might take a little longer, in that it will require me to actually think about what I'm doing...
76. bq. This might take a little longer, in that it will require me to actually think about what I'm doing... no worries, good job so far. Did the updated patch made sense to you? I think you had a good warmup phase now we can go somewhat deeper!

77. New patch, implementing positions() for ReqExclScorer and ReqOptSumScorer, with a couple of basic tests. These just return Conj/Disj PositionIterators, ignoring the excluded Scorers. It works in the simple cases that I've got here, but they may need to be made more complex when we take proximity searches into account.
78. bq. New patch, implementing positions() for ReqExclScorer and ReqOptSumScorer, with a couple of basic tests. looks good! I will commit it and we can iterate further. Good to see those additional tests! Proximity searches are a different story and I will leave that for later. We can even add that once this is in trunk. In general we need to add a ton of testcases and straight out the api at some point but lets get all queries supporting that stuff first.
79. I've spent a bit of time on ExactPhraseScorer this weekend, and I think I'm going to need some pointers on how to proceed. BlockPositionIterator expects all component terms in its target phrase to have their own subscorers, but ExactPhraseScorer uses a different algorithm that doesn't use subscorers at all. Are we happy with the positions() algorithm being completely separate from the algorithm used by the initial search? Or should I be looking at refactoring PhraseQuery to create subscorers and pass them down to the various Scorer implementations?
80. Ah, never mind, I'm an idiot. I extend BlockPositionIterator to take an array of TermPositions. Patch will follow later today.
81. Patch implementing positions() for ExactPhraseScorer. I've had to make some changes to PhraseQuery#scorer() to get this to work, and MultiPhraseQuery is now failing some tests, but it's a start.
82. bq. I've had to make some changes to PhraseQuery#scorer() to get this to work, and MultiPhraseQuery is now failing some tests, but it's a start. that's fine changes are necessary to make this work. I updated your patch with some quick fixes to give you some more insight how I'd do it. the core tests pass now but I am not sure if that is really the way to go or if we need to smooth some edges. I didn't have too much time so I hacked it together to get you going. We can iterate on that a little during the week. thanks for the patch man!
83. I committed the latest patch. thanks alan
84. Patch adding positions() support to SloppyPhraseScorer. Some tests fail here, as MultiPhraseQuery doesn't create a TermDocsEnumFactory in certain circumstances yet. I'll get working on that next. The meaty bit of the patch is a refactoring of SloppyPhraseScorer#phraseFreq() to use an iterator when calculating phrase frequencies. We can then reuse this logic when finding the phrase positions. I think we can probably simplify the PostingsAndFreq and TermDocsEnumFactory constructors as well now - for example, we don't need the TermState in TDEF because we want to wind back the DocsAndPositionsIterators to their initial positions. I think. (I'm still getting my head round some of these internal APIs, can you tell?)
85. This fixes the MultiPhraseQuery tests. Simplifies the TermDocsEnumFactory interface considerably, and implements a new version that can return a UnionDocsAndPositionsEnum. MPQ still doesn't support positions() completely, because UnionDocsAndPE doesn't return offsets yet. That'll be in the next patch!
86. Updated patch implementing startOffset and endOffset on UnionDocsAndPositionsEnum. MultiPhraseQuery can now return its positions properly.
87. hey alan, I won't be able to look at this this week but will do early next week! good stuff on a brief look!
88. Hi Simon, I'm going to be away for the rest of the month, but will hopefully be able to work more on this in a couple of weeks. Let me know if there's more I can do.
89. Patch incorporating my previous uncommitted patches, but catching up with changes in trunk.
90. positions() is now implemented on all the various different types of query, I think, with the exception of the BlockJoin queries. Next step is to try and reimplement the various SpanQuery tests using the position filter queries in their place.
91. hey alan, I merged the branch up with trunk and applied you patch (latest-1). Your changes to PhraseQuery are tricky. The SloppyPhraseScorer now uses the same DocsAndPosition enums as the PosIterator in there. That is unfortunately not how it should be. If you pull a PosIterator from a scorer there should not be any side-effect on the score or the iterator if you advance one or the other. Currently I see a failure in PosHighlighterTest: {noformat} [junit4:junit4] Suite: org.apache.lucene.search.poshighlight.PosHighlighterTest [junit4:junit4] FAILURE 0.22s J0 | PosHighlighterTest.testSloppyPhraseQuery [junit4:junit4] > Throwable #1: java.lang.AssertionError: nextPosition() was called too many times (more than freq() times) posPendingCount=-1 [junit4:junit4] > at __randomizedtesting.SeedInfo.seed([C966081DA1EFC306:32134412A4F88738]:0) [junit4:junit4] > at org.apache.lucene.codecs.lucene40.Lucene40PostingsReader$SegmentFullPositionsEnum.nextPosition(Lucene40PostingsReader.java:1127) [junit4:junit4] > at org.apache.lucene.search.PhrasePositions.nextPosition(PhrasePositions.java:76) [junit4:junit4] > at org.apache.lucene.search.PhrasePositions.firstPosition(PhrasePositions.java:65) [junit4:junit4] > at org.apache.lucene.search.SloppyPhraseScorer.initSimple(SloppyPhraseScorer.java:230) [junit4:junit4] > at org.apache.lucene.search.SloppyPhraseScorer.initPhrasePositions(SloppyPhraseScorer.java:218) [junit4:junit4] > at org.apache.lucene.search.SloppyPhraseScorer.access$800(SloppyPhraseScorer.java:28) [junit4:junit4] > at org.apache.lucene.search.SloppyPhraseScorer$SloppyPhrasePositionIntervalIterator.advanceTo(SloppyPhraseScorer.java:533) [junit4:junit4] > at org.apache.lucene.search.poshighlight.PosCollector.collect(PosCollector.java:53) [junit4:junit4] > at org.apache.lucene.search.Scorer.score(Scorer.java:62) [junit4:junit4] > at org.apache.lucene.search.IndexSearcher.search(IndexSearcher.java:574) [junit4:junit4] > at org.apache.lucene.search.IndexSearcher.search(IndexSearcher.java:287) [junit4:junit4] > at org.apache.lucene.search.poshighlight.PosHighlighterTest.doSearch(PosHighlighterTest.java:161) [junit4:junit4] > at org.apache.lucene.search.poshighlight.PosHighlighterTest.doSearch(PosHighlighterTest.java:147) [junit4:junit4] > at org.apache.lucene.search.poshighlight.PosHighlighterTest.testSloppyPhraseQuery(PosHighlighterTest.java:378) ... [junit4:junit4] 2> NOTE: reproduce with: ant test -Dtestcase=PosHighlighterTest -Dtests.method=testSloppyPhraseQuery -Dtests.seed=C966081DA1EFC306 -Dtests.slow=true -Dtests.locale=sr -Dtests.timezone=Africa/Harare -Dtests.file.encoding=UTF-8 [junit4:junit4] 2> [junit4:junit4] > (@AfterClass output) [junit4:junit4] 2> NOTE: test params are: codec=Lucene40: {}, sim=RandomSimilarityProvider(queryNorm=true,coord=false): {f=DFR I(ne)B3(800.0)}, locale=sr, timezone=Africa/Harare [junit4:junit4] 2> NOTE: Linux 2.6.38-15-generic amd64/Sun Microsystems Inc. 1.6.0_26 (64-bit)/cpus=12,threads=1,free=327745216,total=379322368 [junit4:junit4] 2> NOTE: All tests run in this JVM: [PosHighlighterTest] [junit4:junit4] 2> {noformat} this makes the entire sloppy case very tricky though. Even further I don't think sloppy phrase is really correct and if I recall correctly there some issue with it that haven't been resolved for years now. I am not sure how we should proceed with that one. I will need to think about that further. bq. Next step is to try and reimplement the various SpanQuery tests using the position filter queries in their place. please go ahead!
92. Patch with a couple of new tests that exercise the SpanNearQuery-like functions.
93. Alan! I am so glad you are still sticking around! thanks for your patch, I already committed it together with some additions I added today. I saw your comment in the test {noformat} //TODO: Subinterval slops - should this work with a slop of 6 rather than 11? {noformat} I fixed this today since this bugged me for a long time. I basically use the same function that sloppyphrase uses to figure out the matchDistance of the current interval. The test now passes with slop = 6. I also fixed all the tests in TestSimplePositions that did this weird slop manipulation. I also added a new operator based on the Brouwerian difference (here is the crazy paper if you are interested: http://vigna.dsi.unimi.it/ftp/papers/EfficientAlgorithmsMinimalIntervalSemantics) SloppyPhraseScorer now works with a new positioniterator for the single term case ie. not created through MultiPhraseQuery and all tests pass. I still need to find a good way to fix

the multi-term case. What I think is a good plan for the next iteration is to create more tests. What I did with TestSimplePositions is that I copied TestSpans and modified the tests to use PositionIterators and not spans. If you are keen go ahead and grab some of those tests and copy them to the positions package and port them over. I will soon refactor some classnames since IMO PostionIntervalIterator and PositionInterval is 1. too long and 2. not true anymore. we also have offsets in there so for now I will just call them IntervalIterator. Since those are all svn moves I will commit them directly. looking forward to your next patch!

94. FYI - refactoring is done & committed. Alan, I might not be very responsive in the next 2 weeks so happy coding! :)
95. New patch, does a few things: - adds some Javadocs. Not many, though! This is mainly me trying to understand how things fit together here. - pulls the SnapshotPositionCollector into its own class, and extends OrderedConjunctionIntervalIterator to use it. Also adds a new test illustrating this. - cleans up Interval and IntervalIterator a bit. I'll commit this shortly.
96. ALAN! you have no idea how happy I am that you picking this up again. I put a lot of work into this already and I really think we are close already. Only MultiTermSloppyPhrase doesn't work at this point and I honestly think we can just mark this as unsupported (what a crazy scorer) anyway. We really need to clean this stuff up and you basically did the first step towards this. +1 to commit! :)
97. Heh, it was a long two weeks :-) As another step towards making the API prettier, I'd like to rename the queries: - OrderedConjunctionQuery => OrderedNearQuery - BrouwerianQuery => NonOverlappingQuery And maybe add an UnorderedNearQuery that just wraps a BooleanQuery and a WithinIntervalFilter. These names are probably a bit more intuitive to people unversed in IR theory...
98. +1 to the renaming. I still think we should document the actual used algorithm (ie. for BrouweianQuery) with references to the paper though. Please go ahead and add this. I will need to bring this branch up-to-date, will do once you committed these changes.
99. OK, done, added some more javadocs as well. Next cleanup is to make the distinction between iterators and filters a bit more explicit, I think. We've got some iterators that also act as filters, and some which are distinct. I think they should all be separate classes - filters are a public API that clients can use to create queries, whereas Iterators are an implementation detail.
100. Alan, I merged up with trunk and fixed some small bugs. +1 to all the cleanups
101. This patch removes the abstract BooleanIntervalIterator, as it doesn't seem to gain us anything. Other than writing javadocs, we need to replace PayloadTermQuery and PayloadNearQuery, I think. I'll work on that next.
102. alan +1 to the patch BooleanIntervalIterator is a relict. I will go ahead and commit it. bq. Other than writing javadocs, we need to replace PayloadTermQuery and PayloadNearQuery, I think. I'll work on that next. Honestly, fuck it! PayloadTermQuery and PayloadNearQuery are so exotic I'd leave it out and move it into a sep. issue and maybe add them once we are on trunk. We can still just convert them to pos iters eventually. For now that is not important. we should focus on getting this on trunk.
103. OK! I think we're nearly there...
104. alan FYI - I committed some refactorings (renamed Scorer#positions to Scorere#intervals) etc. so you should update. I also committed your lattest patch
105. I've committed a whole bunch more javadocs, and a package.html. There's still a big nocommit in SloppyPhraseScorer, but other than that we're looking good. We could probably do with more test coverage, but then that's never not the case, so...
106. alan, I just committed some more javadocs including more content for the package.html (review would be appreciated) I also fixed all nocommits in the latest commit. The nocommit in SloppyPhraseScorer I removed last week or so while I cheated here a bit. I current throw an UnsupportedOE if there are multiple terms per position right now since I think its not crucial for us to have this for now. I really want it since its the entire point of this feature but moving towards trunk is really what we want so other people get into it too. Being on trunk is very helpful. Regarding tests - I agree we should have more tests especially with bigger documents. I might add a couple of random tests next week but feel free to jump on it. Next step would also be to run ant precommit on top level and see where it barfs. Other than that I really need other committers to look over the API but if nobody does we just gonna put up a patch and tell them we gonna reintegrate in X days :) simon
107. I committed a few more javadoc fixes. Ant precommit passes when run from the top level. Let's get this in trunk!
108. bq. I committed a few more javadoc fixes. Ant precommit passes when run from the top level. Let's get this in trunk! good stuff! lets give other folks the chance to jump on it / comment on what we have and then move forward! BTW. I'd rename the package o.a.l.s.positions to o.a.l.s.intervals what do you think?
109. > I'd rename the package o.a.l.s.positions to o.a.l.s.intervals what do you think? +1
110. I renamed the package and fixed the package html.
111. here is a diff against trunk for better reviewing
112. I'm still trying to catch up here (net/net this looks awesome!), but here's some minor stuff I noticed: Instead of PostingFeatures.isProximityFeature, can we just use X.compareTo(PostingsFeatures.POSITIONS) >= 0? (We do this for IndexOptions). Should we move PostingFeatures to its own source instead of hiding it in Weight.java? Can we put back single imports (not wildcard, eg "import org.apache.lucene.index.*")? PostingFeatures is very similar to FieldInfo.IndexOptions (except the latter does not cover payloads) ... would be nice if we could somehow combine them ...
113. I'm confused on how one uses IntervalIterator along with the Scorer it "belongs" to. Say I want to visit all Intervals for a given TermQuery ... do I first get the TermScorer and then call .intervals, up front? And then call TermScorer.nextDoc(), but then how to iterate over all intervals for that one document? EG, is the caller supposed to call IntervalIterator.scorerAdvanced for each next'd doc? Or ... am I supposed to call .intervals() after each .nextDoc() (but that looks rather costly/wasteful since it's a newly alloc'd TermIntervalIterator each time). I'm also confused why TermIntervalIterator.collect only collects one interval (I think?). Shouldn't it collect all intervals for the current doc?
114. thanks mike for taking a look at this. It still has it's edges so every review is very valuable. {quote}Instead of PostingFeatures.isProximityFeature, can we just use X.compareTo(PostingsFeatures.POSITIONS) >= 0? (We do this for IndexOptions). {quote} sure, I was actually thinking about this for a while though. After a day of playing with different ways of doing it I really asked myself why we have 2 different docs enums and why not just one and one set of features / flags this would make a lot of things easier. Different discussion / progress over perfection.. {quote}Should we move PostingFeatures to its own source instead of hiding it in Weight.java?{quote} alone the same lines, sure lets move it out. {quote}Can we put back single imports (not wildcard, eg "import org.apache.lucene.index.*")?{quote} yeah I saw that when I created the diff I will go over it and bring it back. {quote}PostingFeatures is very similar to FieldInfo.IndexOptions (except the latter does not cover payloads) ... would be nice if we could somehow combine them ... {quote} I agree it would be nice to unify all of this. Lets open another issue - we have a good set of usecases now. {quote} I'm confused on how one uses IntervalIterator along with the Scorer it "belongs" to. Say I want to visit all Intervals for a given TermQuery ... do I first get the TermScorer and then call .intervals, up front? And then call TermScorer.nextDoc(), but then how to iterate over all intervals for that one document? EG, is the caller supposed to call IntervalIterator.scorerAdvanced for each next'd doc?{quote} so my major goal here was to make this totally detached, optional and lazy ie no additional code in scorer except of IntervalIterator creation on demand. once you have a scorer you can call intervals() and get an iterator. This instance can and should be reused while docs are collected / scored / matched on a given reader. For each doc I need to iterate over intervals I call scorerAdvanced and update the internal structures this prevents any additional work if it is not really needed ie. on a complex query/scorer tree. Once the iterator is setup (scorerAdvanced is called) you can just call next() on it in a loop --> while ((interval = iter.next) != null) and get all the intervals. makes sense? {quote}Or ... am I supposed to call .intervals() after each .nextDoc() (but that looks rather costly/wasteful since it's a newly alloc'd TermIntervalIterator each time).

{quote} no that is not what you should do. I think the scorer#interval method javadoc make this clear, no? if not I should make it clear! {quote} I'm also confused why TermIntervalIterator.collect only collects one interval (I think?). Shouldn't it collect all intervals for the current doc?{quote} the collect method is special. It's an interface that allows to collect the "current" interval or all "current" intervals that contributed to a higher level interval. For each next call you should call collect if you need all the subtrees intervals or the leaves. one usecase where we do this right now is highlighing. you can highlight based on phrases ie. if you collect on a BQ or you can do individual terms ie. collect leaves. makes sense?

115. This is what I had in mind to remove PostingFeatures.isProximityFeature (it's only used in one place...).
116. {quote} bq. I'm confused on how one uses IntervalIterator along with the Scorer it "belongs" to. Say I want to visit all Intervals for a given TermQuery ... do I first get the TermScorer and then call .intervals, up front? And then call TermScorer.nextDoc(), but then how to iterate over all intervals for that one document? EG, is the caller supposed to call IntervalIterator.scorerAdvanced for each next'd doc? so my major goal here was to make this totally detached, optional and lazy ie no additional code in scorer except of IntervalIterator creation on demand. once you have a scorer you can call intervals() and get an iterator. This instance can and should be reused while docs are collected / scored / matched on a given reader. For each doc I need to iterate over intervals I call scorerAdvanced and update the internal structures this prevents any additional work if it is not really needed ie. on a complex query/scorer tree. Once the iterator is setup (scorerAdvanced is called) you can just call next() on it in a loop --> while ((interval = iter.next) != null) and get all the intervals. makes sense? {quote} OK so it sounds like I pull one IntervalIterator up front (and use it for the whole time), and it's my job to call .scorerAdvanced(docID) every time I either .nextDoc or .advance the original Scorer? Ie this "resets" my IntervalIterator onto the current doc's intervals. bq. I think the scorer#interval method javadoc make this clear, no? if not I should make it clear! I was still confused :) I'll take a stab at improving it ... also we should add @experimental... {quote} bq. I'm also confused why TermIntervalIterator.collect only collects one interval (I think?). Shouldn't it collect all intervals for the current doc? the collect method is special. It's an interface that allows to collect the "current" interval or all "current" intervals that contributed to a higher level interval. For each next call you should call collect if you need all the subtrees intervals or the leaves. one usecase where we do this right now is highlighing. you can highlight based on phrases ie. if you collect on a BQ or you can do individual terms ie. collect leaves. makes sense? {quote} Ahhh .... so it visits all intervals in the query tree leading up to the current match interval (of the top query) that you've iterated to? OK. Maybe we can find a better name ... can't think of one now :)
117. thanks mike for the commits! much apprecitated! {quote} OK so it sounds like I pull one IntervalIterator up front (and use it for the whole time), and it's my job to call .scorerAdvanced(docID) every time I either .nextDoc or .advance the original Scorer? Ie this "resets" my IntervalIterator onto the current doc's intervals. {quote} exactly! {quote} Ahhh .... so it visits all intervals in the query tree leading up to the current match interval (of the top query) that you've iterated to? OK. Maybe we can find a better name ... can't think of one now {quote} a better name for "collect"?
118. bq. a better name for "collect"? Yeah, to somehow reflect that it's visiting/collecting/recursing on the full interval tree ... but nothing comes to mind ... When I first saw it / read the docs I thought this was analogous to Scorer.score(Collector), ie that it would "bulk collect" all intervals from the iterator.
119. I think the patch for review is incomplete? e.g. I see PostingsFeatures was added to the Scorer api but its not in the patch.
120. bq. I think the patch for review is incomplete? e.g. I see PostingsFeatures was added its a inner class of Weight in that patch but we might move it out!
121. {quote} Instead of PostingFeatures.isProximityFeature, can we just use X.compareTo(PostingsFeatures.POSITIONS) >= 0? (We do this for IndexOptions). {quote} I think this is a confusing part of the current patch. For example: {noformat} // Check if we can return a BooleanScorer - if (!scoreDocsInOrder && topScorer && required.size() == 0) { + if (!scoreDocsInOrder && flags == PostingFeatures.DOCS_AND_FREQS && topScorer && required.size() == 0) { {noformat} I don't think we should be doing these == comparisons. What if someone sends DOCS_ONLY? (which really ConstantScoreQuery i think should pass down to its subs and so on, so they can skip freq blocks, but thats another thing to tackle).
122. There seems to be a lot of unrelated formatting changes in important classes like TermWeight.java etc Can we factor these out and do any formatting changes separately?
123. I don't like the general style of things like Collector.postingsFeatures() From the naming, you cant tell this is a "getter". In general I think methods like this should be getPostingsFeatures() ?
124. It's sort of disturbing that if you iterate over intervals for a PhraseQuery we pull two DocsAndPositionsEnums per term in the phrase ... But then ... this would "typically" be used to find the locations to hilite, right? Ie not for the "main" query? Because if you wanted to do this for the main query you should really use one of the oal.search.intervals.* queries instead, and those pull only a single D&PEnum per Term I think?
125. It seems like the new oal.search.interval queries are meant to replace spans? So ... should we remove spans? Or is there functionality missing in intervals?
126. {quote} But then ... this would "typically" be used to find the locations to hilite, right? Ie not for the "main" query? Because if you wanted to do this for the main query you should really use one of the oal.search.intervals.* queries instead, and those pull only a single D&PEnum per Term I think? {quote} correct. {quote} It seems like the new oal.search.interval queries are meant to replace spans? So ... should we remove spans? Or is there functionality missing in intervals? {quote} eventually yes. Currently they don't score based on positions they only filter. My plan was to bring this on trunk including spans. Once on trunk move spans to a module and cut over the functionality query by query. We are currently missing payload support which I think we should add once we are on trunk. makes sense?
127. To me Interval.java looks a lot like a span. I think it would be good to resolve this before landing on trunk. If we cant score based on positions, it seems to me the api is not fully baked? e.g. i think it would be better to score based on positions and run benchmarks and so on first. here we also get a lot more index option flags. I dont like how many of these we have: * indexoptions * flags on docsenum * flags on docsandpositionsenum * now here, flags on scorer/collector/etc I am working on some ideas to clean some of this up in trunk separate from this branch. i think this makes the apis really confusing.
128. bq. Interval.java looks a lot like a span. I think it would be good to resolve this before landing on trunk. what exactly did you expect? I mean its basically the same thing but reusing the name sucks. what do you wanna resolve here? regarding your comments could you put your ideas up here in a somewhat more compact form than 5 1-line comments? This would be great especially what kind of ideas you have and you resovle / work on on trunk so we can maybe be more productive here. thanks
129. Where do we stand on this now? It sounds as though we need to get more implemented before everyone's happy with it being merged in. I can make a start at cutting TestSpansAdvanced and TestSpansAdvanced2 over to intervals tests this week (at a glance they're the only tests we have for Span scoring at the moment), although I guess things are going to go on hold a bit for the ApacheCon.
130. +1 to add scoring! go ahead this would be great. will you be at apache con?
131. > will you be at apache con? Not this year :-(Will try and make one next year, though!
132. Here's a first attempt at duplicating the Span scoring in IntervalFilterQuery. It needs more tests, and the Explanation needs to be modified, but it's something :-) One thing I'm uncertain of is the algorithm being used for scoring here. It treats all matching 'spans' (or 'intervals' now, I suppose) in a document as equivalent for the purposes of scoring, weighting only by match distance, but this seems to throw away certain bits of possibly useful info - for example, if we're filtering over a BooleanQuery with a number of SHOULD clauses, then an

interval that contains all of them is going to score the same as an interval that contains only two, but with the same overall match distance. This seems wrong... Or maybe I'm just misunderstanding how this gets put together.

133. I've started to use this branch in an (experimental!) system I'm developing for a client. The good news is that performance is generally much better than the existing system that uses SpanQueries - faster query time and smaller memory footprint, and also nicer GC behaviour (I can't give exact numbers, but suffice to say that where the previous system regularly ran out of memory, this one hasn't yet)! There are definitely some rough edges, though, which I'll try and smooth out and add as patches. 1) There isn't a replacement for SpanNotQueries - the BrouwerianIterator comes close, but doesn't quite cover all the use cases. In this instance, I need to have the equivalent of a 'not within' operator - match intervals that do not fall within a given another interval. I've written a new iterator, which I've called an 'InverseBrouwerianIntervalIterator' for want of a better name, but it definitely could do with some more eyes on it... 2) The API is not very nice when it comes to subclassing Iterators. For example, I have 'anchor' terms at the start and end of documents, which allow users to query for terms within a certain distance from them. These shouldn't be highlighted, so I created an AnchorTermQuery which returned a different type of IntervalIterator that didn't do anything in its collect() method. To do this, I had to create an AnchorTermWeight, an AnchorTermScorer and an AnchorTermIntervalIterator, all of which were more or less copy-pastes of the equivalent Term* classes; it would be nice to make this easier... 3) MultiTermQueries don't return iterators unless you set their rewrite policies to something other than CONSTANT_SCORE_REWRITE. 4) I found a bug in the iterators() method of DisjunctionSumScorer - if all subscorers are PositionFilterScorers, then you can get NPEs if the subscorers have matches that don't pass the filters. I'll add a test case shortly 5) I had to run this without my scoring patch (this case doesn't actually use scoring, so it doesn't matter that much), because MultiTermQueries can blow up in scoring if they get rewritten into blank queries; I guess this wasn't a problem with Span* queries, but I haven't had a chance to work out how to get round it. Will add another test case for this as well. All in all, though, these are looking much better than the equivalent SpanQueries. Position filters on boolean queries in particular work much better - the semantics of SpanQueries are completely wrong for this, and involved generating very heavy queries for pretty simple cases. Nice work!
134. hey alan, bq. I've started to use this branch in an (experimental!) system I'm developing for a client. very good news! cool stuff - can you provide more infos what you are doing there? Do you highlight too? regarding your latest patch - commit it! bq. 1) There isn't a replacement for SpanNotQueries - the BrouwerianIterator comes close, but doesn't quite cover all the use cases. I can you provide a testcase what it doesn't cover? you can go ahead and commit it even if you don't have a fix. bq. 2) The API is not very nice when it comes to subclassing Iterators. For example, I have 'anchor' terms at the start and end of documents, which allow I am not sure I understand this. if you have marker terms how do they differ from ordinary terms can't you just do a nearOrdered("X", "_ENDMARKER_") query? I don't see where you need to subclass here. can you elaborate? bq. 4) I found a bug in the iterators() method of DisjunctionSumScorer great, can you submit the testcase? bq. 3) MultiTermQueries don't return iterators unless you set their rewrite policies to something other than CONSTANT_SCORE_REWRITE. yeah the problem here is that we use a filter instead of a scorer, you should see an exception right? I think it would make sense to have a MTQ rewrite a query on a ConstantScoreQuery instead of a filter - we can't get a interval iter from a filter :/ I think overall we should move out of this issue and create separate issues for all you cases. Also for the things robert mentioned like exploring "Scorere extends DocsAndPosEnum"
135. Hi Simon, I'll open separate sub-tasks for the issues. The system I'm building is basically an equivalent of the elasticsearch percolator - we register a bunch of queries, and then run them all against individual documents passing through the system. We then emit the exact positions which have matched, which is a type of highlighting, I guess. The point of the anchor terms is that we don't want to highlight them - if you're searching for a term within five positions of the start of a document, you don't want the first term of the document highlighted as well.
136. Just committed a massive test refactoring, which should show where the problems are in DisjunctionIntervalScorer and MultiTermQuery. Lots of the tests fail now, as the previous ones weren't necessarily picking up false positives (UnorderedNearQuery is particularly bad for this).
137. I want to get this moving again - will get the branch up to date tomorrow and then iterate from there.
138. YEAH!!!
139. Since the last patch went up I've fixed a bunch of bugs (BrouwerianQuery works properly now, as do various nested IntervalQuery subtypes that were throwing NPEs), as well as adding Span-type scoring and fleshing out the explain() methods. The only Span functionality that's missing I think is payload queries. If we want to have *all* the span functionality in here before it can land on trunk I can work on that next. It would also be good to do some proper benchmarking. Do we already have something that can compare sets of queries?
140. So at the moment, IntervalFilterScorer doesn't consume all the intervals on a given document when advancing, it just checks if the document has any matching intervals at all. Which is great for speed, but bad for scoring - you want to iterate through the intervals on a document to get the within-doc frequency, which can then be passed to the docscorer. You also need to iterate through everything to deal with payloads. Is it worth specialising here? Have two query types (or maybe just a flag on the query), so you can optimize for query speed or for scoring. SpanScorer always iterates over all spans, by comparison.
141. bq. The only Span functionality that's missing I think is payload queries. If we want to have all the span functionality in here before it can land on trunk I can work on that next. I really think we can skip that for now. bq. It would also be good to do some proper benchmarking. Do we already have something that can compare sets of queries? We do have LuceneUtil but its not like straight forward. I will take a look what we can do here. bq. Is it worth specialising here? Have two query types (or maybe just a flag on the query), so you can optimize for query speed or for scoring. SpanScorer always iterates over all spans, by comparison. I think we should specialize the Scorere here. Visiting the least amount of intervals possible is maybe worth it. So from my perspective what we should try exploring is making the scorer a DocsAndPosEnum in the branch and see if we can remove the Interval API mostly in favor of the DocsAndPos API. The only problem I have with this is really that if a given scorer consumes intervals from a subscorer it needs to buffer all those if it's parent needs all of them too. Not sure if it is worth it at this point. Ideally I would want to have DocsAndPosEnum to be folded into DocsEnum first too.
142. I'm going to try applying the patch from LUCENE-4524 here and see if that helps. Next step would be to add startPosition() and endPosition() to DocsEnum, and try re-implementing the filter queries using methods directly on child scorers, rather than pulling a separate interval iterator.
143. Alan, I don't think you can cut over to DocsEnum or DocsAndPositionsEnum. DocsAndPosEnum has a significant problem that doesn't allow efficient PosIterator impl underneath it. It defines that DocsAndPosEnum#nextPosition should be called at max DocsEnum#freq() times which is fine on a low level but bogus for lazy pos iterators since we don't know ahead of time how many intervals we might have. I think we first need to fix this problem before we can go and do this refactoring, makes sense? PhraseQuery does only know his freq currently because it's greedy and pulls all intervals at once.
144. Hm, OK. So can we change the nextPosition() API to return -1 once the positions have been exhausted, rather than becoming undefined? So a consumer would look something like: {{monospaced}} int pos; while ((pos = dp.nextPosition()) != -1) { // do stuff here } {{monospaced}} Implementations that need to know the frequency call freq(), others can iterate lazily.
145. Alan: its a nice idea... we should seriously consider something this (-1 or NO_MORE_POSITIONS or whatever) if it would allow this stuff to just work over the existing D&P api. I dont know what the cost would be to existing impls (e.g. Lucene41PostingsReader would need some code changes), but hopefully small or nil. And of course having an API like this would be well worth any small performance hit.

146. bq. Hm, OK. So can we change the nextPosition() API to return -1 once the positions have been exhausted, rather than becoming undefined? So a consumer would look something like: yeah I was saying the same thing yesterday when I talked about this to rob. This would make stuff more consistent too. I will open an issue
147. I think this should be explored in the branch versus a separate issue E.g. we shouldnt impose this on postings implementations unless it sorta works with the whole design here. I'd also really recommend NO_MORE_POSITIONS not -1. -1 currently means "invalid" (e.g. you should not have called nextPosition). Its not like any Scorer would need to check for this, because if you try to do prox operations on a field that omits position information, the user should be getting an exception up-front from the Weight.
148. I've been chipping away at this for a bit. Here's a summary of what I've done: - Applied LUCENE-4524, and also added startPosition() and endPosition() to DocsEnum - Changed the postings readers to return NO_MORE_POSITIONS once nextPosition() has been called freq() times - Extended the ConjunctionScorer and DisjunctionScorer implementations to return positions - Added an abstract PositionFilteredScorer with reset(int doc) and doNextPosition() methods - Added a bunch of concrete implementations (ExactPhraseQuery, NotWithinQuery, OrderedNearQuery, UnorderedNearQuery, RangeFilterQuery) with tests - these are all in the posfilter package I still need to implement SloppyPhraseQuery and MultiPhraseQuery, but I actually think these won't be too difficult with this API. Plus there are a bunch of nocommits regarding freq() calculations, and this doesn't work at all with BooleanScorer - we'll probably need a way to tell the scorer that we do or don't want position information. [~simonw] and I talked about this on IRC the other day, about resolving collisions in ExactPhraseQuery, but I think that problem may go away doing things this way. I may have misunderstood though - if so, could you add a test to TestExactPhraseQuery showing what I'm missing, Simon?
149. Commit 1535436 from [~romseygeek] in branch 'dev/branches/LUCENE-2878' [https://svn.apache.org/r1535436] LUCENE-2878: Merge from trunk
150. Now we are talking.... Sent from my iPhone
151. Ooh, hello. So the LUCENE-2878 branch is a bit of a mess, in that it has two semi-working versions of this code: Simon's initial IntervalIterator API, in the o.a.l.search.intervals package, and my DocsEnum.nextPosition() API in o.a.l.search.positions. Simon's code is much more complete, and I've been using a separately maintained version of that in production code for various clients, which you can see at https://github.com/flaxsearch/lucene-solr-intervals. I think the nextPosition() API is nicer, but the IntervalIterator API has the advantage of actually working. The github repository has some other stuff on it too, around making the intervals code work across different fields. The API that I've come up with there is not very nice, though. It would be ace to get this moving again!
152. A patch against trunk of the latest changes I've been working with, using Simon's original IntervalIterator API. This also adds a getField() method to Query, which can be null if the query doesn't have a single associated field (eg for booleans that span across multiple fields).
153. This patch replaces Query.getField() with Query.getFields(), which has the benefit of not actively lying about the state of the query. It also removes the O(n^2) getField() implementation on BooleanQuery, which turned out to not be such a good idea.
154. Talk of a 5.0 release has got me working on this again. Here's my latest patch against trunk. * consolidates DocsEnum and DocsAndPositionsEnum * all Scorers now implement nextPosition(), startPosition(), endPosition(), startOffset() and endOffset(), including ExactPhraseScorer and SloppyPhraseScorer * Collectors have a postingFeatures() method to indicate the features they require from scorers (frequencies, positions, offsets, payloads, etc) * adds a number of new queries in oal.search.posfilter that can use nextPosition() There are a few test failures still, which I'm chasing down. Still to do: * work out a decent way of ensuring that position filter queries don't get run inadvertently on subqueries from separate fields * clean up the docs() and docsAndPositions() API in PostingsReader * javadocs, etc * payload queries * nuke spans! This has moved on a long way from the existing branches, to the point that it's probably worth deleting them and opening a new one. I've gone with adding nextPosition() directly to Scorer, which I think ends up with a cleaner API than the IntervalIterators that Simon and I worked on last year. It's been three years, let's get this done.
155. This would be a huge selling point for Lucene5.x NUKE SPANS!
156. Yeah, tell me about it. I haven't looked at the details yet but I think scorers exposing positions would enable a new highlighter to be built that could accurately know where a match occurred. Hopefully there is control to expose positions for queries that are normally position-less (e.g. wildcard queries).
157. The good news is that I have somebody to sponsor me to actually do this work in the next month or so. I have a heap of other things I need to do first though :-(
158. Commit 1638800 from [~romseygeek] [https://svn.apache.org/r1638800] Branch for LUCENE-2878
159. Any reason why the branch was not created in the branches folder alongside the other lucene«jira issue» branches? Where it is now it is not replicated on git.apache.org, nor on GitHub...
160. Because I'm an idiot, mainly. Will move the branch, thanks...
161. Attached: one massive patch. All tests are passing, including the SloppyPhraseQuery highlighter ones which have had me sweating blood over the past couple of days. It would be good to get a Jenkins instance running against the lucene2878 branch to see if I've missed any cases. The API changes are small but they touch a lot of classes. I appreciate that this will be difficult to review, just because of the size of the patch. Should I open a review board for it?
162. Can you explain the core api changes such as DocsAndPositionsEnum being folded into DocsEnum? It looks inconsistent, with some impls returning -1 for the "positions" methods and some throwing UOE. And why do we still have an Interval class, if D&PEnum has what we need? I am also unsure of Collector.postingsFeatures, why do we need this? It seems enough to pass flags to Weight.scorer()
163. Hm, it is inconsistent at the moment, as I started out returning -1 and then started throwing UOE to find places that were returning the wrong thing. Probably what it ought to do is return NO_MORE_POSITIONS if it doesn't support positions at all (and -1 for offsets, as that's expected everywhere). I'll update the patch. Interval is there as a concrete POJO that can be stored, cached, passed around, etc, after the scorer has moved on. Maybe it should be an interface instead, and DocsEnum should implement it? Collector.postingsFeatures() is there for things like highlighting. Normally a query wouldn't require offsets, for example, but if you're using the PostingsCollector for highlighting, then you want to retrieve offsets when pulling scorers.
164. I don't think the method shoudl be on collector, this is slow and unrelated to collection. For highlighting, i think it should actually invoke Weight.scorer(..., PARAM, ...) and advance() scorers to precise docids of interest. It isn't doing collection.
165. bq. I don't think the method shoudl be on collector, this is slow and unrelated to collection. OK, I'll try ripping it out and see what stops working
166. {quote} Interval is there as a concrete POJO that can be stored, cached, passed around, etc, after the scorer has moved on. Maybe it should be an interface instead, and DocsEnum should implement it? {quote} How often is this POJO needed? I see what you are saying, but we have a fairly advanced collection of scorers today, and none need that with Scorer :) If this "snapshot of docsenum" is only needed for say one or two scorers, maybe it can be explicitly and (package-)privately used by them, whereas the rest just use docsenum?
167. Commit 1643349 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643349] LUCENE-2878: return NO_MORE_POSITIONS rather than throwing UOE
168. Currently this NO_MORE_POSITIONS = Integer.MAX_VALUE? Unlike docids, we don't prevent a user from having a position with this value: we only look for overflow (negative) in IndexWriter. I think -1 works better here?
169. Interval (outside the posfilter package) and Collector.postingsFeatures() are basically only used by the PositionsCollector, which I'm using for highlighting and testing. The highlighter can just be removed for now, and worked on in a separate issue - maybe as an extension of

PostingsHighlighter? I'll rework the tests as well.
170. In general, I think anything we can do to split it up is good :)
171. The disadvantage of using -1 to indicate NO_MORE_POSITIONS is that having an exhausted iterator sort 'naturally' after an unexhausted one simplifies a lot of the interval spanning logic in things like the PositionQueue or in OrderedNearQuery.
172. I dont really see it as a choice, the current value is not a reserved value.
173. bq. I dont really see it as a choice, the current value is not a reserved value. We could chose to make it a reserved value since we're going into a new major version. Does anyone even have positions of MAX_VALUE?
174. Its more than that, indexwriter has to reject such positions and tests updated, old postings formats need to have a check and do something, etc. Because some people like large position increment gap values between field instances, someone can easily blow up positions to large values without actually having that many terms.
175. bq. Its more than that, indexwriter has to reject such positions and tests updated, That part seems easy. bq. old postings formats need to have a check and do something, etc. That part could be more of a pain... not sure. But maybe not necessary if it's never happened? (see below) bq. Because some people like large position increment gap values between field instances, someone can easily blow up positions to large values without actually having that many terms. Solr has defaulted to 100 forever... and even if a user changed to a *huge* value, it would be super unlikely to hit MAX_INT exactly (esp since they would get an exception if they went over... they are already in super dangerous territory). I think the only people that would have MAX_INT position are those that are using positions to encode something other than real positions (i.e. using positions like payloads) and have values going all the way up to MAX_INT, or are using MAX_INT as a special value. It could very well be that no such user exists.
176. It does not matter to me really what solr defaults to, we cant be lenient here. Changing these constants requires work.
177. bq. It does not matter to me really what solr defaults to I was just trying to determine the likelihood of *anyone* actually having pos=MAX_INT in their index. bq. Changing these constants requires work. Sure - I was only pointing it out as an option (5.0 would be the right time), not strongly advocating for it.
178. Commit 1643787 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643787] LUCENE-2878: Make Interval package-private, remove PositionsCollector
179. Commit 1643788 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643788] LUCENE-2878: Remove positions highlighter (for now); MemoryPostingsFormat fix
180. Commit 1643791 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643791] LUCENE-2878: Remove postingsFeatures() from Collector
181. Commit 1643819 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643819] LUCENE-2878: Fix compilation in TestGrouping; don't update extremes in PositionQueue with exhausted iterator
182. Commit 1643835 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643835] LUCENE-2878: PostingsHighlighter uses Integer.MAX_VALUE to indicate empty docsenum
183. Commit 1643843 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643843] LUCENE-2878: Get Solr compiling
184. Commit 1643846 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1643846] LUCENE-2878: Set NO_MORE_POSITIONS to be -1
185. re NO_MORE_POSITIONS being -1, turned out it only required about 4 lines of code to change, so that was easy enough. Updated patch: * Collector.postingsFeatures() is gone * PositionsHighlighter is gone * Interval is now package-private, and I might go further and make it a protected subclass of PositionIntervalFilter * NO_MORE_POSITIONS = -1 There are still a few nocommits, which I'll work through next.
186. Commit 1644050 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1644050] LUCENE-2878: UnorderedNearQuery scoring
187. The remaining nocommits are: * implement nextPosition() etc on TermAutomatonScorer ** this should be fairly simple, and I'm working on it now * implement nextPosition() etc on SpanScorer ** this is not so simple, as Spans.next() will move to the next document if it's exhausted the positions on the current doc, a completely different API. I'm tempted to just not implement these, seeing as they already don't work with 'normal' queries. A followup JIRA will remove Span queries entirely from trunk and deprecate them in 5.0 anyway. * should PositionFilteredScorer enforce that all its subqueries should be on the same field? ** again, I'm tempted not to. 1) this means adding getField() to all queries, which is a far more invasive API change, and 2) we already have FieldMaskingSpanQuery to get around this limitation in SpanQueries, which suggests that it's not necessarily a good thing anyway. On the other hand, most of the time you want phrases or proximity queries to be on the same field. * how to deal with Payloads on intervals. ** At the moment we just return the payload at the starting position. This is probably not what we want. Spans has a different API, returning a Collection<byte[]> from getPayload rather than a BytesRef, which may be closer to what should happen. I think this can be changed from a nocommit to a TODO, however, and dealt with in a separate issue. I also want to have a look at the MultiTermQuery rewrite API, as at the moment it defaults to CONSTANT_SCORE_FILTER_REWRITE, which of course doesn't support positions. It would be nice to have some way of telling the query whether positions are needed or not at rewrite time. But again, that's for another issue. So I think this is close. I'll get TermAutomatonScorer sorted today.
188. Another thing to think about is the interaction of freq() and nextPosition() calls. At the moment the API contract is that you can call both, and indeed this was necessary to ensure that you didn't call nextPosition() too many times. However, now that you can just call nextPosition() until you get NO_MORE_POSITIONS this isn't needed, and seeing as many Scorers actually need to consume all their positions to return an accurate freq(), I'd like to change this to say that you *can't* call nextPosition() after you've called freq(). We can then have a default Scorer implementation of freq() that calls nextPosition() repeatedly to count instances, with TermScorer etc having their own specialized overrides.
189. {quote} We can then have a default Scorer implementation of freq() that calls nextPosition() repeatedly to count instances, with TermScorer etc having their own specialized overrides. {quote} Can we not do the slow default implementation and just keep it abstract?
190. How about keeping it abstract, but having a concrete protected slowFreq() method?
191. I would rather us not add slow methods at all to our scoring api. if these position iterators do not support freq(), then they do not support freq().
192. So this doesn't quite work, because things like CheckIndex want to validate frequencies and positions, and in fact the vast majority of DocsEnum implementations have no problem supporting this. I'm now leaning towards making all these 'interval' Scorers just return 1 from freq(), and instead enforcing the rule that you can't call score() and nextPosition(), moving the limitation up to Scorer itself.
193. Why would checkindex be calling this on an interval scorer? I dont understand the problem. I dont think we should add complex rules to Scorer, just throw UOE here from the interval ones.
194. Commit 1645525 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1645525] LUCENE-2878: Scoring on positionfilterqueries
195. Commit 1645528 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1645528] LUCENE-2878: last nocommits

196. Commit 1645535 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1645535] LUCENE-2878: precommit cleanups
197. Commit 1645925 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1645925] LUCENE-2878: merge conflicts
198. nocommits either resolved or (mostly!) changed to TODOs. I think this is ready.
199. Hi Alan, Why are there now additional branches in nextPosition() in posting readers? Can we avoid these? If it is to support calling nextPosition() after it has already been exhausted, I am unsure we should be doing this. nextDoc() does not support this, for example. Can we improve this logic in reuse code? Reuse code is notorious for sneaky bugs, and I don't like the comparison done here. Why can't we just check if the bit is set? {code} if ((flags & DocsEnum.FLAG_POSITIONS) >= DocsEnum.FLAG_POSITIONS) {code} This check also is confusing to me in the first place. Why do we have both docsEnum and docsAndPositionsEnum methods, both returning DocsEnum, with the former checking for a FLAG_POSITIONS and calling the other one in that case. I think there should just be one method instead if we want both to share the same api, thats ok. TermScorer introduces outdated dead code that is unused. What are the Span scoring classes still doing here? Do we have tests for any of this new code? I see no added tests files. What about benchmarks? We should try to compare against trunk, since there are a lot of change here. In general, I can try to make a patch for smaller things, to make things faster. Iterating this way may take a long time.
200. I ran the benchmark: {noformat} Task QPS trunk StdDev QPS patch StdDev Pct diff HighSloppyPhrase 13.40 (10.4%) 10.31 (5.5%) -23.1% (-35% - -7%) HighPhrase 17.98 (5.6%) 13.92 (3.1%) -22.6% (-29% - -14%) MedPhrase 257.86 (7.1%) 213.38 (3.6%) -17.2% (-26% - -7%) LowPhrase 35.68 (1.8%) 33.32 (1.7%) -6.6% (-9% - -3%) MedSloppyPhrase 15.79 (4.2%) 14.92 (3.6%) -5.5% (-12% - 2%) LowSloppyPhrase 118.09 (2.4%) 112.14 (2.0%) -5.0% (-9% - 0%) HighTerm 138.18 (10.2%) 136.72 (6.7%) -1.1% (-16% - 17%) MedTerm 202.67 (9.6%) 200.94 (6.3%) -0.9% (-15% - 16%) HighSpanNear 144.67 (4.3%) 144.35 (4.3%) -0.2% (-8% - 8%) MedSpanNear 143.52 (3.9%) 143.30 (4.0%) -0.2% (-7% - 8%) Respell 85.33 (1.8%) 85.32 (2.6%) -0.0% (-4% - 4%) LowTerm 1052.81 (8.5%) 1053.59 (5.3%) 0.1% (-12% - 15%) LowSpanNear 27.81 (2.9%) 27.83 (2.9%) 0.1% (-5% - 6%) Prefix3 232.97 (4.6%) 233.55 (4.5%) 0.3% (-8% - 9%) AndHighHigh 90.67 (1.7%) 91.01 (1.1%) 0.4% (-2% - 3%) Fuzzy1 102.98 (2.1%) 103.38 (3.5%) 0.4% (-5% - 6%) AndHighLow 1121.50 (4.8%) 1126.02 (3.9%) 0.4% (-7% - 9%) AndHighMed 127.28 (2.0%) 127.88 (1.1%) 0.5% (-2% - 3%) Fuzzy2 68.39 (2.1%) 68.77 (3.1%) 0.5% (-4% - 5%) Wildcard 48.08 (2.4%) 48.43 (4.2%) 0.7% (-5% - 7%) IntNRQ 9.69 (5.8%) 9.79 (7.2%) 1.1% (-11% - 15%) OrNotHighLow 67.55 (8.1%) 68.88 (7.8%) 2.0% (-12% - 19%) OrNotHighMed 61.00 (8.3%) 62.38 (8.0%) 2.3% (-12% - 20%) OrNotHighHigh 35.44 (9.5%) 36.50 (9.5%) 3.0% (-14% - 24%) OrHighNotHigh 25.97 (9.6%) 26.80 (9.7%) 3.2% (-14% - 24%) OrHighNotMed 82.14 (10.1%) 84.84 (10.2%) 3.3% (-15% - 26%) OrHighHigh 29.25 (10.3%) 30.27 (10.5%) 3.5% (-15% - 27%) OrHighNotLow 104.15 (10.3%) 107.82 (10.5%) 3.5% (-15% - 27%) OrHighMed 65.67 (10.4%) 68.01 (10.7%) 3.6% (-15% - 27%) OrHighLow 63.61 (10.6%) 65.91 (10.7%) 3.6% (-16% - 27%) {noformat} We should look into the regressions for phrases. But first I need to work on LUCENE-6117, it is killing me :)
201. Hi Rob, Thanks for helping out here! The additional branches are there to allow for returning NO_MORE_POSITIONS once the positions are exhausted, but maybe I should move that logic up to TermScorer instead and put the assertions back in for the PostingsReaders. Merging TermsEnum.docs() and TermsEnum.docsAndPositions() might be tricky because their API is different - docs() never returns null, while docsAndPositions() will return null if the relevant postings data isn't indexed. Although having said that, I'm probably already breaking that contract by redirecting from one to the other with the flags check. I'll fix TermScorer. I haven't nuked Spans yet, mainly because I think we should probably keep them (as deprecated) in 5.0, and remove them only in trunk. It would also make the patch bigger :-) I changed existing test files rather than adding any new ones, apart from the tests exercising the PositionFilterQueries. Maybe a way to reduce the size of the patch would be to remove the PositionFilterQueries from this issue and create a new one for them? Then this one is just about changing the DocsEnum/TermsEnum API.
202. Commit 1646271 from [~romseygeek] in branch 'dev/branches/lucene2878' [https://svn.apache.org/r1646271] LUCENE-2878: Remove dead code from TermScorer
203. To make this more digestible, is it worth breaking it out into LUCENE-4524 for the DocsEnum/DocsAndPositionsEnum changes and then keeping this one for the changes to the Weight API and the new query classes?
204. Please have a look at LUCENE-6308, I think that is a good intermediate step for this.
205. Alan, could you please comment on the status of this journey? I know you've made incremental steps but it's not clear if Spans will eventually get nuked or not.
206. It's a bit disturbing quite how long it takes to scroll to the bottom of this JIRA now... Spans have received a lot of attention over the past year or so (many thanks to [~paul.elschot@xs4all.nl]!), so they certainly aren't going to be removed. It's also become clear to me that there are many Scorers that it doesn't make sense to expose positions on. My rough plan of action from here is: * LUCENE-6845: make Spans *be* a Scorer * Move the basic SpanQueries out of the a.o.l.search.spans package and into a.o.l.search ** some of these queries can then move into the queries module, eg SpanFieldMaskingQuery, but I think it's worth having basic near/orderednear/nonoverlapping functionality in core * Turn TermQuery and PhraseQuery (and possibly others, eg MultiTermQuery) into SpanQueries ** this should be simple after LUCENE-6845, because they can still expose their efficient no-positions scorers, and additionally expose a Spans view Span scoring is still a bit odd (eg with a SpanOrQuery you can end up scoring terms that don't actually match in the current document), but that can be dealt with separately. We should probably close this as Won't Fix, but it's been open for so long it feels a bit wrong to do that :-) Maybe I'll wait until TermQuery and PhraseQuery can expose positions, and resolve as a duplicate instead.
207. This ticket has changed direction wildly a few times, but always had two basic requirements: * allow positional queries with a nicer API and a more concrete theoretic footing than Spans * make it possible to build accurate highlighters We now have IntervalQuery (LUCENE-8196) that does the first, and Matches (LUCENE-8229) that does the second. There's still work to be done on both (I don't think we're quite ready to nuke Spans, and phrase/span/interval queries don't report their matches yet), but I think we can close this issue out now?
208. +1
209. monumental effort thanks everybody involed