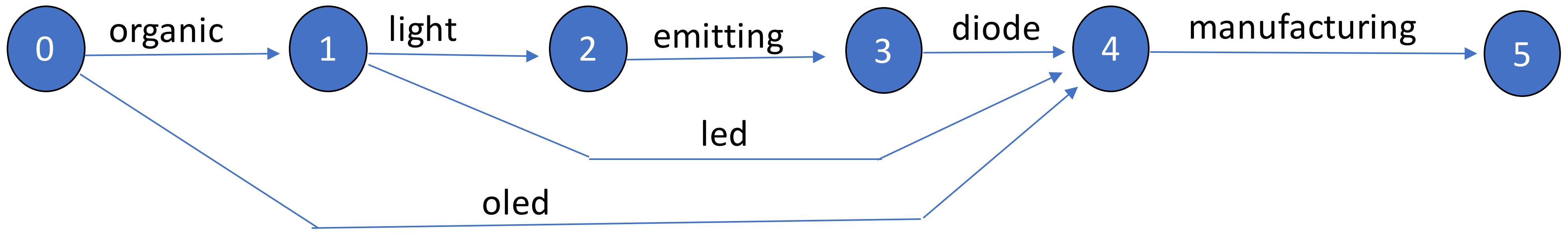
# Complete, precise graph-based phrase query

Michael Gibney
University of Pennsylvania Libraries
michael@michaelgibney.net

# TokenStream structure

- Lucene's TokenStream API:
    - Initially assumed linear stream of tokens
    - Since addition of PositionLengthAttribute (3.6.0), represents tokens as branching graphs
- Uses of TokenStream API:
    - Index-time: determine tokens/token structure serialized to the index
    - Query-time: determine terms and structure of proximity-based queries

Example: "oled manufacturing"



## Latent potential of PositionLengthAttribute

Braided River: Waimakariri River, Canterbury, NZ

I, Gobeirne [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)]

With PositionLengthAttribute, TokenStreams may split and join back together, without losing information about token adjacency in different branches.

To fully leverage the structure of the TokenStream API would call for three interrelated changes:

1. Store position length in the index
2. Augment Postings API to expose position length to index readers
3. Update query implementation(s) to leverage indexed position length (as exposed through the Postings API)

## LUCENE-7398: nominally about "nested" Span queries

(in the absence of indexed position length, nested Span queries are the primary – perhaps only – way of generating/encountering variable-length subclauses)

But the underlying problem is more general: phrase search over variable-length subclauses.

Accordingly, this presentation will discuss an implementation over SpanNearQuery; but the general approach is relevant and applicable to any phrase query implementation, including the new IntervalSource API.

# Abstraction of the Spans API contract with respect to position order

For simplicity, and to emphasize the generality of the problem, we will not explicitly consider "nested" Span queries, disjunctions, etc.

The solution will be discussed strictly with respect to the formal constraints of the Spans API, namely:

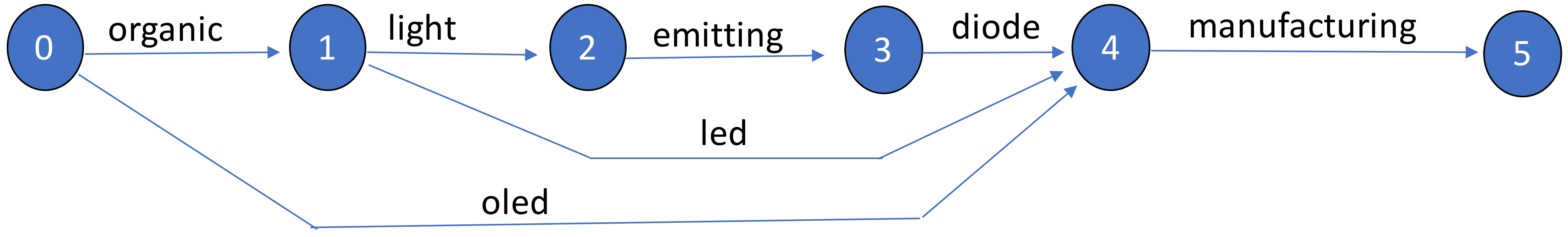
1. Spans positions are advanced forward-only, by calls to nextStartPosition();
2. within a given doc, positions are ordered "by increasing start position and finally by increasing end position"

Notably, we make no other assumptions; endPosition may be arbitrarily larger than startPosition (and as a consequence, endPosition may decrease for subsequent startPositions), positions with identical startPosition and/or endPosition may be repeated arbitrarily many times, etc.

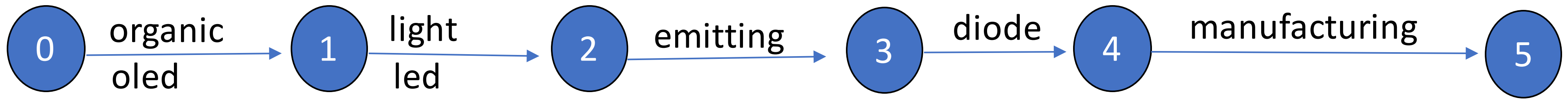
# Index-time graph token adjacency is not preserved

"OLED manufacturing"

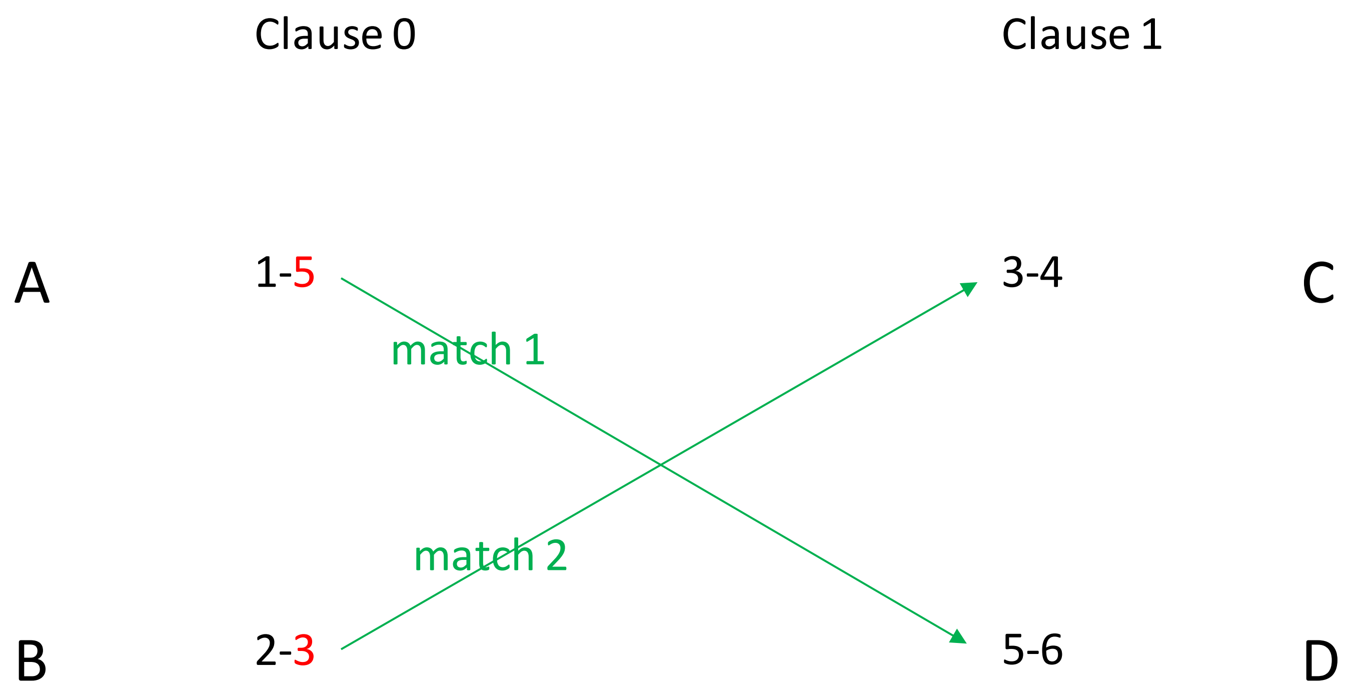
As tokenized at index-time:



As recorded in index:



A simple case that would break for lazy matching:



## Status quo: LUCENE-7398

In practice, use of SpanNearQuery currently implies (witting or unwitting) acceptance of:

1. The assumption that the index-time token stream should be linear, minimally augmented, and with incrementally increasing position
2. The fact that lazy iteration over subclauses will miss some valid matches, generate some spurious matches, and score unpredictably.

Fallback to (Multi)PhraseQuery (which enumerates all possible paths through a given query) fixes the second of these issues, but not the first (and doesn't scale well, introducing the potential for exponential query expansion)

# Foundation: a backtracking-capable Spans wrapper

We need a generic wrapper around Spans that allows us to support backtracking efficiently, without buffering any more positions than necessary.

Backtracking Spans: rather than advancing by calling nextStartPosition(), we advance by calling:

```
public int nextMatch(int hardMinStart, int softMinStart, int startCeiling, int minEnd);
public int reset(int hardMinStart, int softMinStart);
```

- hardMinStart: forever discard positions with start < this parameter
- softMinStart: skip (for purpose of returning nextMatch, but do not discard) positions with start < this parameter
- startCeiling: disregard (for purpose of returning nextMatch, but do not discard) positions with start >= this parameter
- minEnd: when non-negative, defines a minimum threshold for span endPositions. Spans with endPosition < this value should be discarded forever

## Properties of backtracking-capable Spans wrapper

- Position queue (sorted according to the order of positions from the backing Spans)
- Linked, for efficient iteration and node removal
- Array-backed (circular buffer), for efficient binary seek to particular startPositions during backtracking
- Dynamically resizable backing array, to support arbitrarily large position buffer

# Building matches over subclauses

Each subclause has its own queue of positions

Nodes in those queues are linked laterally across subclauses to "build" matches without duplicating information about positions

Nodes store information about a given position, and prev/next nodes for the given subclause, but also "reachable" (within slop contstraints) nodes in prev/next subclauses, and reachable nodes in the last subclause. N.b.: Node references must be stable!

The resulting "word lattice" is built by using "nextMatch()" to drive a depth-first search to discover (and build/cache) edges in the dynamic graph represented by valid Nodes at a point in time.

Phrase "paths" already explored are cached, enabling "match tree" traversal to be shortcircuited (and downstream postfix "subtrees" grafted onto new upstream match "prefixes".

# 2-dimensional queue traversal, building word lattice

$$\begin{tikzcd}
A \arrow[r] \arrow[dr] & A' \arrow[r] \arrow[dr] & A'' \\
B \arrow[r] \arrow[dr] & B' \arrow[r] \arrow[dr] & B'' \\
C \arrow[r] \arrow[dr] \arrow[ddr] \arrow[dddr] & C' \arrow[dddr] & C'' \\
D \arrow[r] \arrow[dr] \arrow[ddr] & D' \arrow[ddr] & D'' \\
E \arrow[r] \arrow[dr] & E' \arrow[dr] & E'' \\
F & F' \arrow[r] & F''
\end{tikzcd}$$

# Managing GC for heavily-linked data structure

links/edges between Nodes require *many* simple linking nodes (similar to those required to “link” a linked list).

- Lots of small, transient objects
- Lots of GC (potentially)

Solution: pool queue Nodes and linking nodes.
- Separate pool for each subclause (pool grows to suit needs of particular subclause)
- Nodes are released/reused when they are no longer referenced in the match "word lattice"
- Linking nodes are returned to the pool upon release/reuse of their currently-associated Node.

Results, on a moderate-sized production index:
- Consistent performance
- 2x to 10x better performance than with object pooling disabled (the highly variable response time is in keeping with the intermittent nature of long GC-related pauses).

## Support for indexed position length

This query implementation finally gives a reason to index (as opposed to ignore) position length

For purposes of development, testing, and initial deployment, position length has been recorded in Payloads in the index.

This encoding is accomplished by a "PositionLengthOrderTokenFilterFactory", which orders tokens to conform to the ordering specified by the Spans API (startPosition, secondary sort on endPosition).

I would love to evaluate the performance impact of position length recorded natively in the index and exposed via postings API … LUCENE-4312?

# startPosition lookahead

Thorough matching requires storing all relevant positions, to support backtracking

But to know whether all relevant positions have been seen, must always iterate *past* relevant positions, to the first *irrelevant* position … so we'd *always* be buffering *all* relevant positions.

Solution: at term level (with benefits cascading up to higher-level conjunction Spans), support lookahead startPosition *without actually advancing Spans position*.

Implemented with Payloads for now; but might theoretically be integrated directly in codecs, pre-loading and buffering exactly 1 startPosition

This approach works nicely, but was a little tricky to integrate with the "word lattice" 2-dimensional queue approach to building matches – integration was accomplished by having each Node start as a "provisional" unbuffered wrapper around the backing Spans.

## PositionLength edge cases

Optimizations for the relatively common case of a SpanNearQuery composed mainly (or exclusively) of simple TermSpans subclauses

Now that we're indexing position length and using it at query time, there are shortcircuits and optimizations that we can make if we know, for a given term in a given document:

1. Whether endPosition ever decreases across subsequent positions (often "no")
2. The maximum positionLength (often "1")

Currently implemented as 4 bytes pre-allocated in the Payload of the first instance of each term in each doc. And updated on doc flush.

## Common words: the last hurdle

Now that SpanNearQuery respects position length, and position length is indexed, we have a reason to use SpanNearQuery for *all* queries – not just ones that are recognizable at as graph queries

Common words proved problematic though: once we drop the assumption of positionLength==1, we *always* have to start our matches from the first subclause, losing the ability* to lead the search with less common/costly terms. Standard "common word" solutions both wreak havoc with phrase search:

1. CommonGramsFilter
2. StopwordFilter

Current solution: leave main field indexed as-is, build separate field of "CommonGrams"-style shingles to perform pre-filtering of conjunction Spans within pre-determined maximum slop.

For our use cases, this approach has resulted in worst-case SpanNearQuery performance nearly identical to extant worst-case PhraseQuery performance, and we are now running full graph queries in production for every user query.

* "losing the ability" -- or, the logic and prerequisites for doing so would become considerably more complicated

# Match modes

Different types of matching are appropriate to different situations. The thoroughness of matching under the modified SpanNearQuery supports nuanced match modes, including:

1. **Greedy**: Once a valid match is found for a given startPosition, greedy match mode shortcircuits without attempting to find other valid paths for that startPosition.

2. **PerEndPosition**: This mode will continue exploring possible match paths until it is determined that all valid endPositions for a given startPosition have been discovered.

3. **PerPosition**: This mode will continue exploring possible match paths until it is determined that all valid positions for all subclauses have been reported as a match.

## Major Benefits

Introduces robust support for:

- index-time multi-term synonyms
- Index-time WordDelimiterGraphFilter
- Index-time multi-token orthographic variants (e.g., CJK, etc.)
- Other tokenization schemes that could create complex token graph structures (e.g., NGrams, Shingles, etc.)

More intuitive/predictable behavior of queries:

- More thorough, predictable, nuanced scoring
- Improved behavior of other queries (perhaps ComplexPhraseQParser)

# Potential non-text use cases?

There are non-text use cases that could be well-represented as "text": an ordered stream of discrete, possibly overlapping, elements, each having a start "position" and an end "position".

Time-series data would provide promising candidates:

- Travel scheduling
- Complex event processing

But also perhaps other domains -- especially domains that prioritize precision.

## Thank you!

Testing and feedback are welcome!

Links:
https://issues.apache.org/jira/browse/LUCENE-7398
https://michaelgibney.net/lucene/graph/
https://github.com/magibney/lucene-solr/tree/LUCENE-7398/master

Contact:
michael@michaelgibney.net