Document Counting in Compressed Space

Jouni Sirén, Wellcome Trust Sanger Institute

with

Travis Gagie, Aleksi Hartikainen, Juha Kärkkäinen, and Simon J. Puglisi,
University of Helsinki

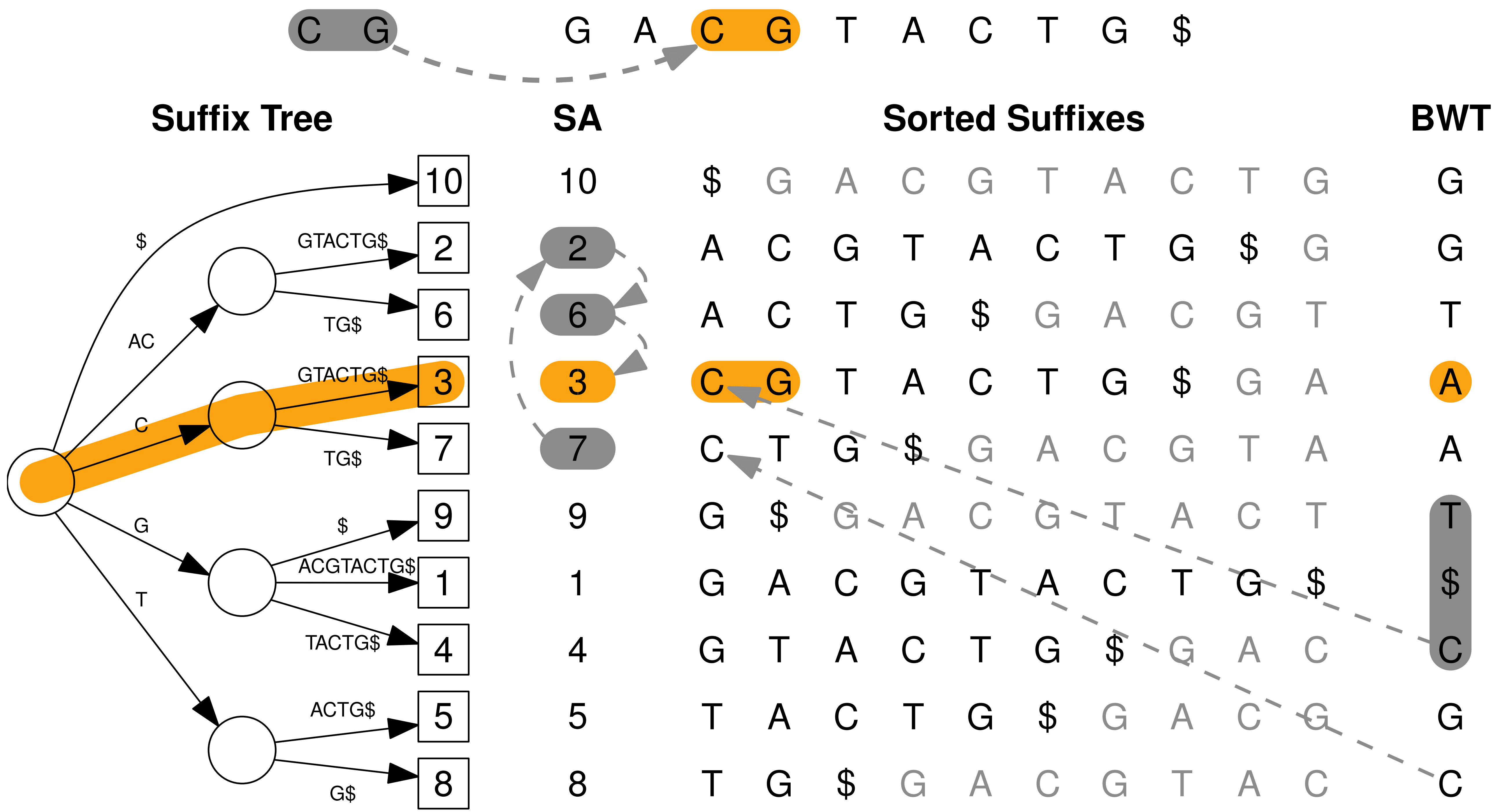
Gonzalo Navarro, University of Chile

Document counting

- We have a collection of documents (strings, texts, sequences).
- We want to count the number of documents a pattern (a string) occurs in.
- Pattern occurrences are substrings of a document.
- We are interested in time/space trade-offs for data structures that augment existing text indexes.

Background

Query	Result	Description
find(P)	[sp, ep] v	Lexicographic range of suffixes starting with pattern P, or suffix tree node corresponding to P.
locate(P) locate(sp, ep) locate(v)	SA[sp, ep]	Starting positions of the occ = ep + 1 – sp occurrences of pattern P in the document collection.
count(P) count(sp, ep) count(v)	docc	The number of documents where the pattern occurs at least once.
list(P) list(sp, ep) list(v)	{ DA[i] | sp ≤ i ≤ ep }	The identifiers of the documents where the pattern occurs at least once.



C G
G A C G T A C T G $
Suffix Tree
SA
Sorted Suffixes
BWT
$
AC
C
G
T
GTACTG$
TG$
GTACTG$
TG$
$
ACGTACTG$
TACTG$
ACTG$
G$
10
2
6
3
7
9
1
4
5
8
10 $ G A C G T A C T G G
2 A C G T A C T G $ G G
6 A C T G $ G A C G T T
3 C G T A C T G $ G A A
7 C T G $ G A C G T A A
9 G $ G A C G T A C T T
1 G A C G T A C T G $ $
4 G T A C T G $ G A C C
5 T A C T G $ G A C G G
8 T G $ G A C G T A C C

Compressed text indexes

- The FM-index (Ferragina and Manzini, 2005) and the compressed suffix array (Grossi and Vitter, 2005) are based on the Burrows-Wheeler transform. Their size is close to a compressed representation of the BWT.

- They solve find() essentially in $O(|P|)$ time (0.1 to 1 μs/character) by using backward searching.

- By sampling one out of s suffix array cells, they also solve locate() in $O(s \cdot occ)$ time (typically 1 to 10 μs/occurrence) with $O(s \log n)$ bits of extra space.

Document listing

- Document array stores the document identifier for each suffix. $DA[i] = j$, if character $T[SA[i]]$ is in document j. The array takes $n \log d$ bits.
- Brute-D solves list() by sorting DA[sp, ep] and reporting all unique document identifiers.
- Muthukrishnan (2002) and Sadakane (2007) proposed algorithms for finding the first occurrences of each document identifier in DA[sp, ep]. The algorithms are usually not competitive in practice (Navarro et al., 2014).

Precomputed document listing

- PDL (Gagie et al., 2013) stores the answers for list() queries for a subset of suffix tree nodes. The answers are compressed with a grammar-based compressor.

- The answer for a list() query is computed as the union of a small number of stored answers (for long ranges) or by using locate() for (short ranges).

- PDL is usually as fast as Brute-D, but it may use much less space.

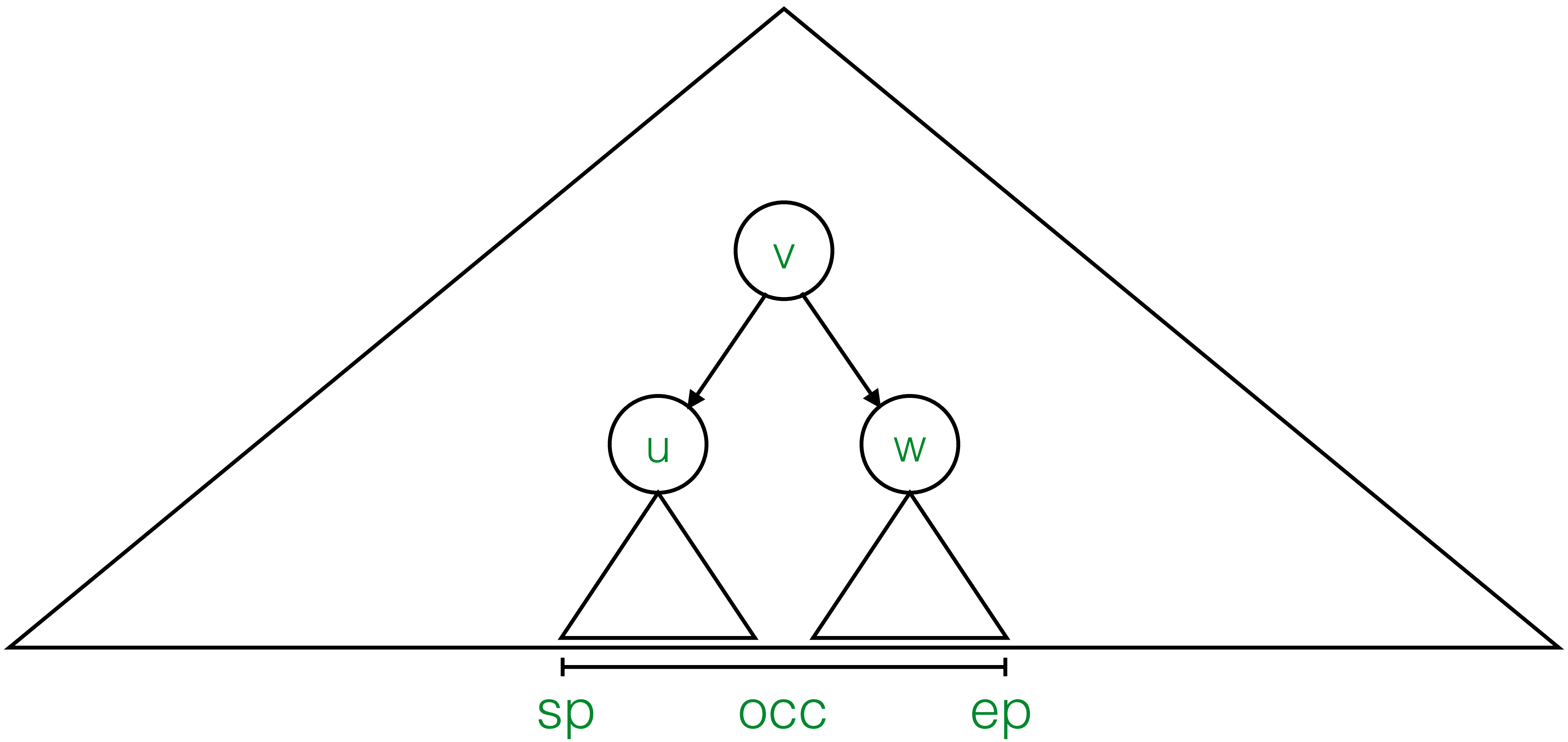
Our index

- We already have a CSA/FMI augmented with DA or PDL for the other queries, so we can use them for count() for free.

- Any specialized counting structure must be faster than Brute-L and PDL to justify the additional space usage.

Sadakane's Method (2007)

Binary suffix tree



Redundant suffixes: $h(v) = |\, list(u) \cap list(w)\, |$

$count(v) = count(u) + count(w) - h(v) = \ldots = occ - \sum_{v'} h(v')$

How to find the subtree of v from sp and ep?

- We form array $H[1, n - 1]$ by traversing the binary suffix tree in inorder and listing $h(v)$ for each internal node v. This simplifies the counting queries to $count(sp, ep) = occ - \sum H[sp, ep - 1]$.

- Array H can be encoded in unary as a bitvector of length $2n - d - 1$. With a select structure, we can solve count() in $O(1)$ time and $2n + o(n)$ bits as $count(sp, ep) = 2occ - 1 - (select(ep) - select(sp))$.

- There are several ways to compress the bitvector to use even less space.

Compression

1. Reordering

- We never use count(v) for nodes that do not exist in the original suffix tree.
- Let V be the set of binary tree nodes created from original node v. The bitvector is easier to compress, if we set $h(v) = \sum_{u \in V} h(u)$, and $h(u) = 0$ for the remaining $u \in V$.
- We will always do the reordering, as it has no significant drawbacks.

2. Run-length encoding

- If a pattern occurs in multiple documents, but only once in each document, the corresponding subtree has no redundant suffixes, and the bitvector is compressible with run-length encoding.

- This happens, if the collection contains random sequences or multiple revisions of base documents.

- There are $\Theta(n^2 / d)$ pairs of substrings of a fixed length from the same document. Intra-document collisions become unlikely at substring probability $\Theta(\sqrt{d} / n)$, when the expected occ is $\Theta(\sqrt{d})$. Hence the expected number of runs is $\Theta(n / \sqrt{d})$.

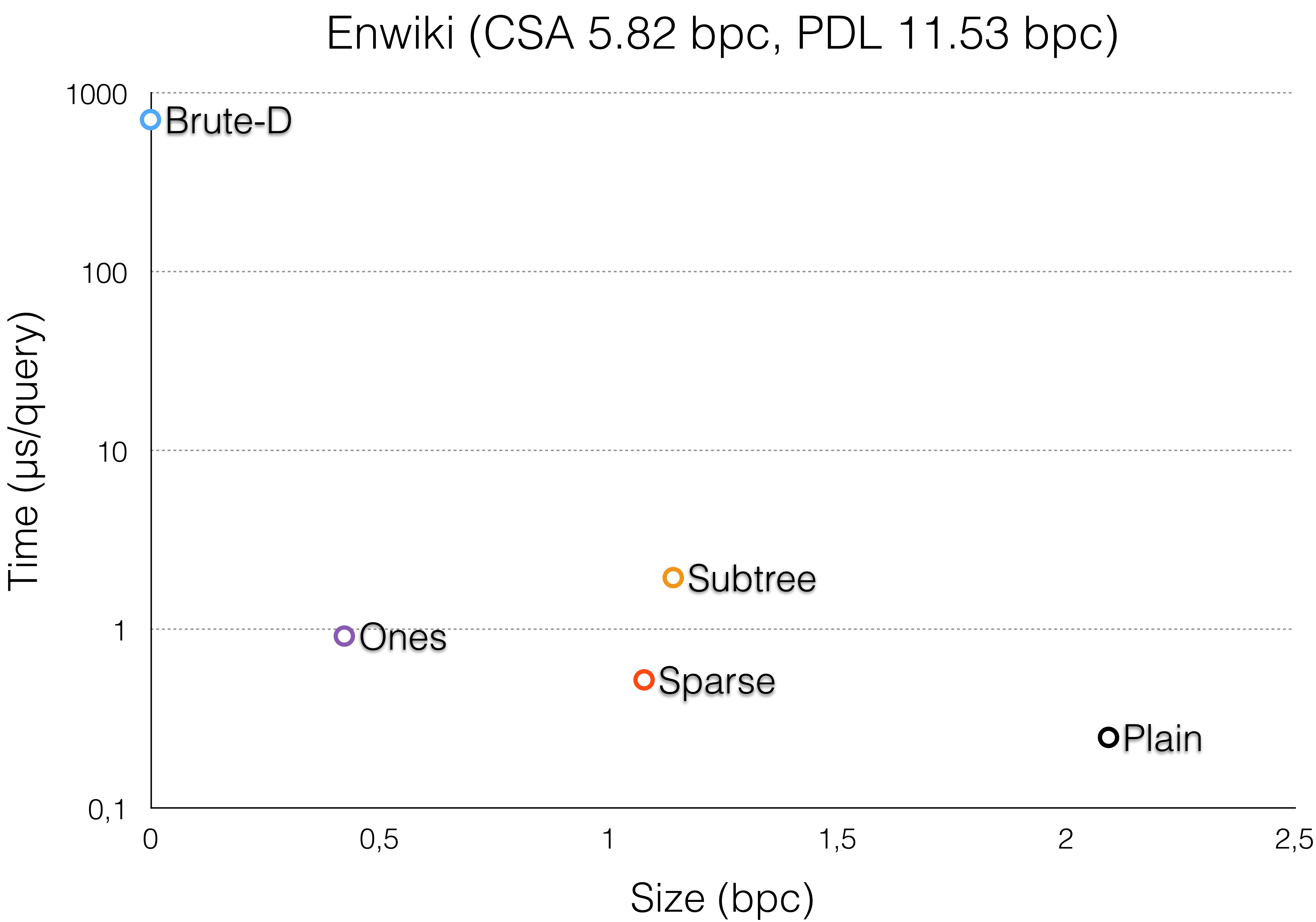
3. Filtering

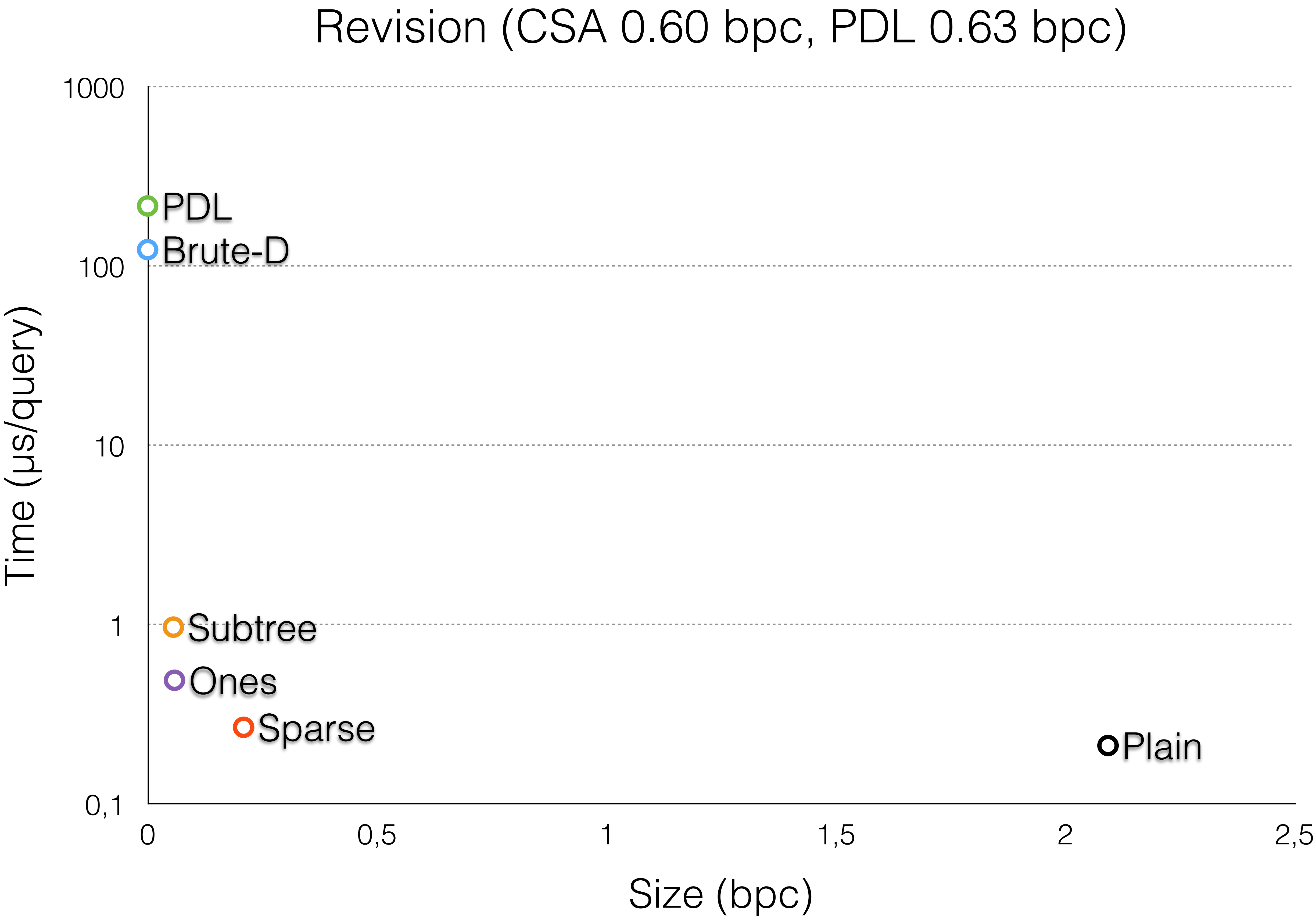
- If we concatenate all revisions of a base document into a single document, the suffix tree may have large subtrees with docc = 1.
- We can filter these subtrees out by collapsing them into leaves and handling them separately.
- Filters can also be based on the properties of the bitvector. An 1-filter handles nodes with $h(v) = 1$ separately, while a sparse filter does the same for nodes with $h(v) > 0$.

Experiments

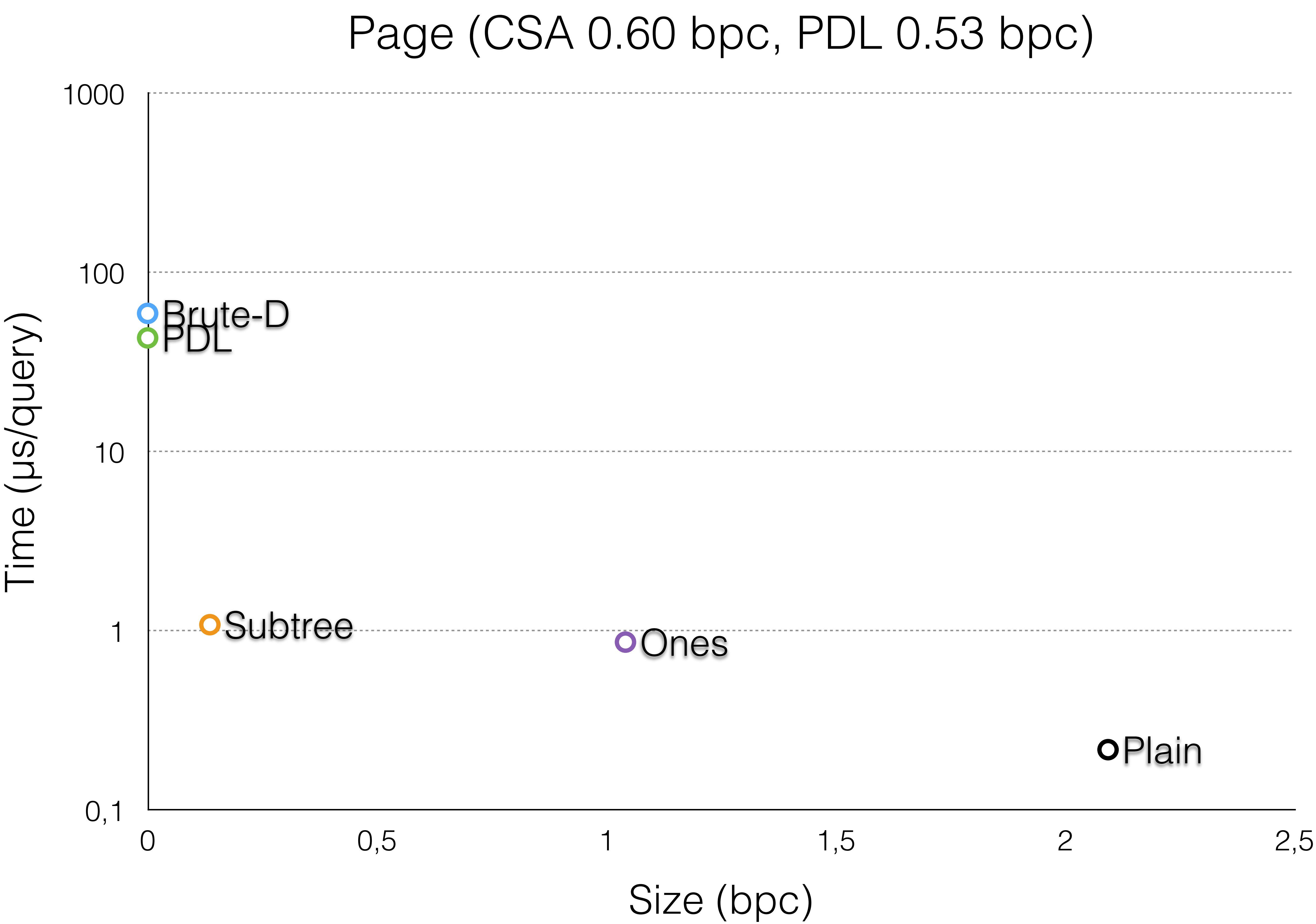
Structures

- Brute-D and PDL use the existing document listing structures.
- Plain is the original Sadakane's bitvector.
- Subtree uses run-length encoding for the subtree filter and Sadakane's bitvector.
- Sparse uses sparse bitvectors for both the sparse filter and the stored $h(v)$ values.
- Ones uses run-length encoding for Sadakane's bitvector and a sparse bitvector for the 1-filter.

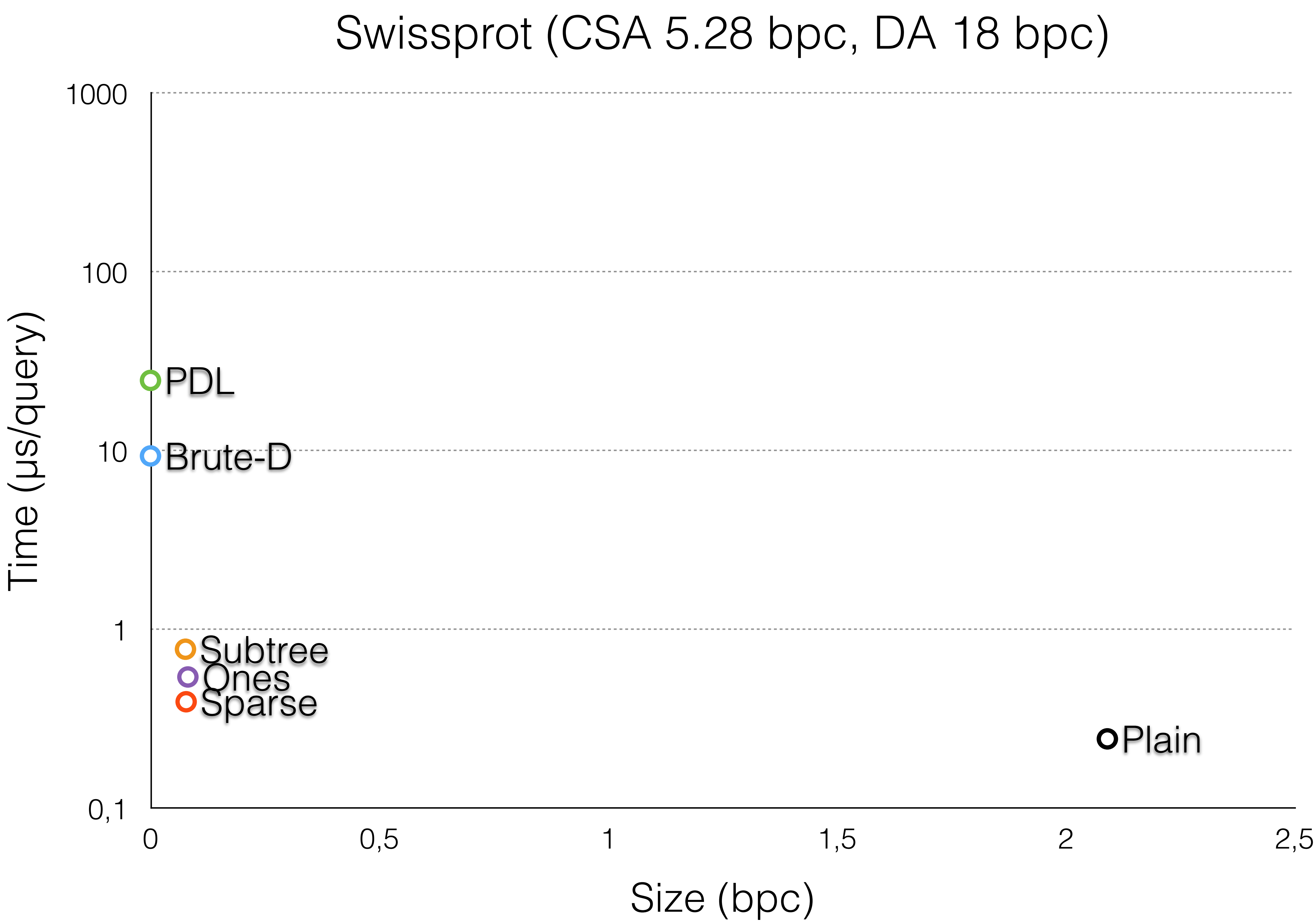

Enwiki (CSA 5.82 bpc, PDL 11.53 bpc)
Time (µs/query)
1000
100
10
1
0,1
0
0,5
1
1,5
2
2,5
Size (bpc)
Brute-D
Subtree
Ones
Sparse
Plain

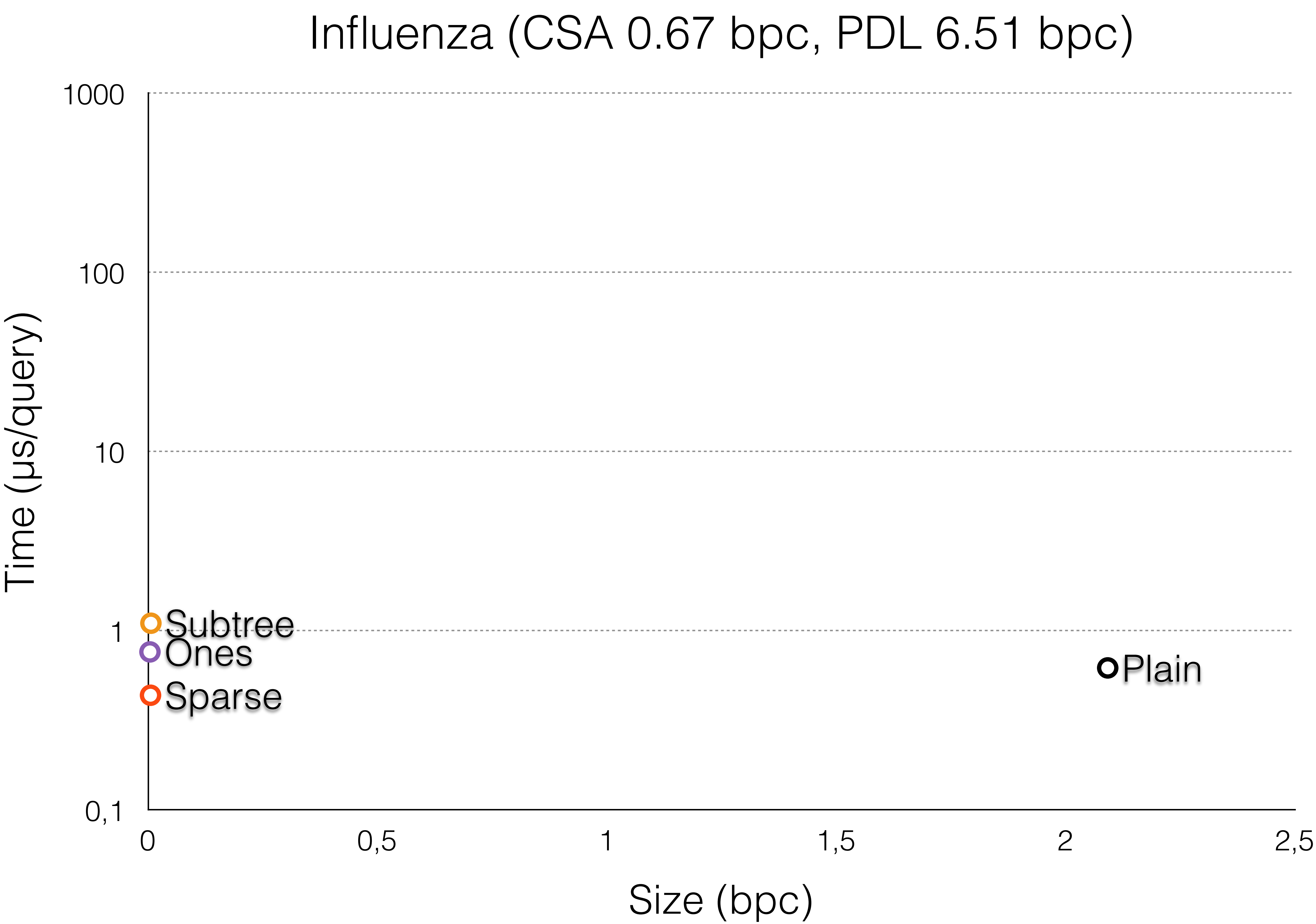


Revision (CSA 0.60 bpc, PDL 0.63 bpc)
Time (µs/query)
1000
100
10
1
0,1
PDL
Brute-D
Subtree
Ones
Sparse
Plain
0
0,5
1
1,5
2
2,5
Size (bpc)



Page (CSA 0.60 bpc, PDL 0.53 bpc)
1000
100
10
1
0,1
Time (μs/query)
Brute-D
PDL
Subtree
Ones
Plain
0
0,5
1
1,5
2
2,5
Size (bpc)


Swissprot (CSA 5.28 bpc, DA 18 bpc)
Time (µs/query)
1000
100
10
1
0,1
PDL
Brute-D
Subtree
Ones
Sparse
Plain
0
0,5
1
1,5
2
2,5
Size (bpc)



Influenza (CSA 0.67 bpc, PDL 6.51 bpc)
Time (μs/query)
1000
100
10
1
0,1
0
0,5
1
1,5
2
2,5
Size (bpc)
Subtree
Ones
Sparse
Plain

Conclusions

- Sadakane’s document counting structure can be compressed with run-length encoding and filters.
- “Typical” document collections, where a pattern usually occurs multiple times in multiple documents, seem to be the worst case.
- Construction algorithms are the current bottleneck. While BWT-based indexes work with hundreds of gigabytes, Sadakane’s bitvector is hard to build beyond a few gigabytes.