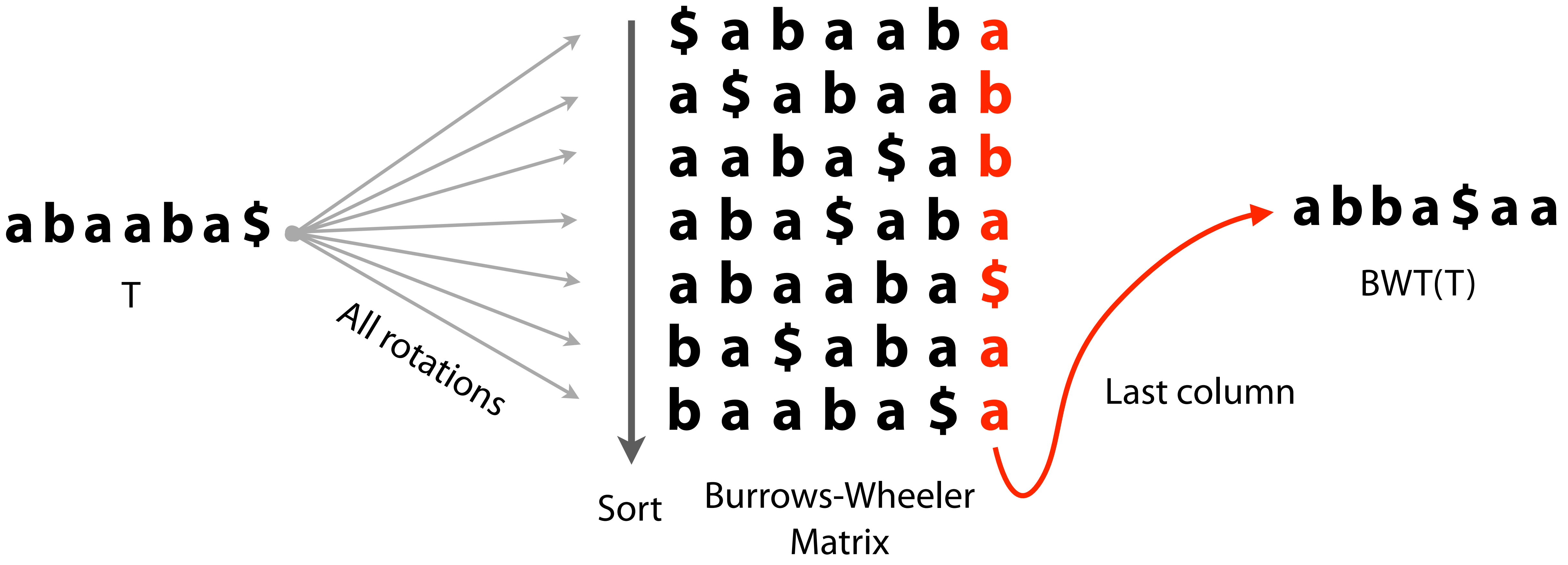
The BWT and FM-index

UNIVERSITY OF MARYLAND

Slides marked with * adopted from Ben Langmead

Burrows-Wheeler Transform

Reversible permutation of the characters of a string, used originally for compression



abaaba$

T

All rotations

Sort

$abaaba

a$abaab

aaba$ab

aba$aba

abaaba$

ba$abaa

baaba$a

Burrows-Wheeler Matrix

Last column

abba$aa

BWT(T)

How is it useful for compression?

How is it reversible?

How is it an index?

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm. *Digital Equipment Corporation, Palo Alto, CA* 1994, Technical Report 124; 1994

Burrows-Wheeler Transform

```
def rotations(t):
    """ Return list of rotations of input string t """
    tt = t * 2
    return [ tt[i:i+len(t)] for i in xrange(0, len(t)) ]
```

Make list of all rotations

```
def bwm(t):
    """ Return lexicographically sorted list of t’s rotations """
    return sorted(rotations(t))
```

Sort them

```
def bwtViaBwm(t):
    """ Given T, returns BWT(T) by way of the BWM """
    return ''.join(map(lambda x: x[-1], bwm(t)))
```

Take last column

```
>>> bwtViaBwm("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnoooaattTmmmrrrrrrooo__ooo'

>>> bwtViaBwm("It_was_the_best_of_times_it_was_the_worst_of_times$")
's$esttssfftteww_hhmmbootttt_ii__woeeaaressIi_______'

>>> bwtViaBwm('in_the_jingle_jangle_morning_Ill_come_following_you$')
'u_gleeeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmylo_oo_'
```

Burrows-Wheeler Transform

BWM bears a resemblance to the suffix array

$abaaba
a$abaab
aaba$ab
aba$aba
abaaba$
ba$abaa
baaba$a

BWM(T)

6	$
5	a$
2	aaba$
3	aba$
0	abaaba$
4	ba$
1	baaba$

SA(T)

Sort order is the same whether rows are rotations or suffixes

Burrows-Wheeler Transform

In fact, this gives us a new definition / way to construct BWT(T):

$$BWT[i] = \begin{cases} T[SA[i]-1] & \text{if } SA[i] > 0 \\ \$ & \text{if } SA[i] = 0 \end{cases}$$

“BWT = characters just to the left of the suffixes in the suffix array”

```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

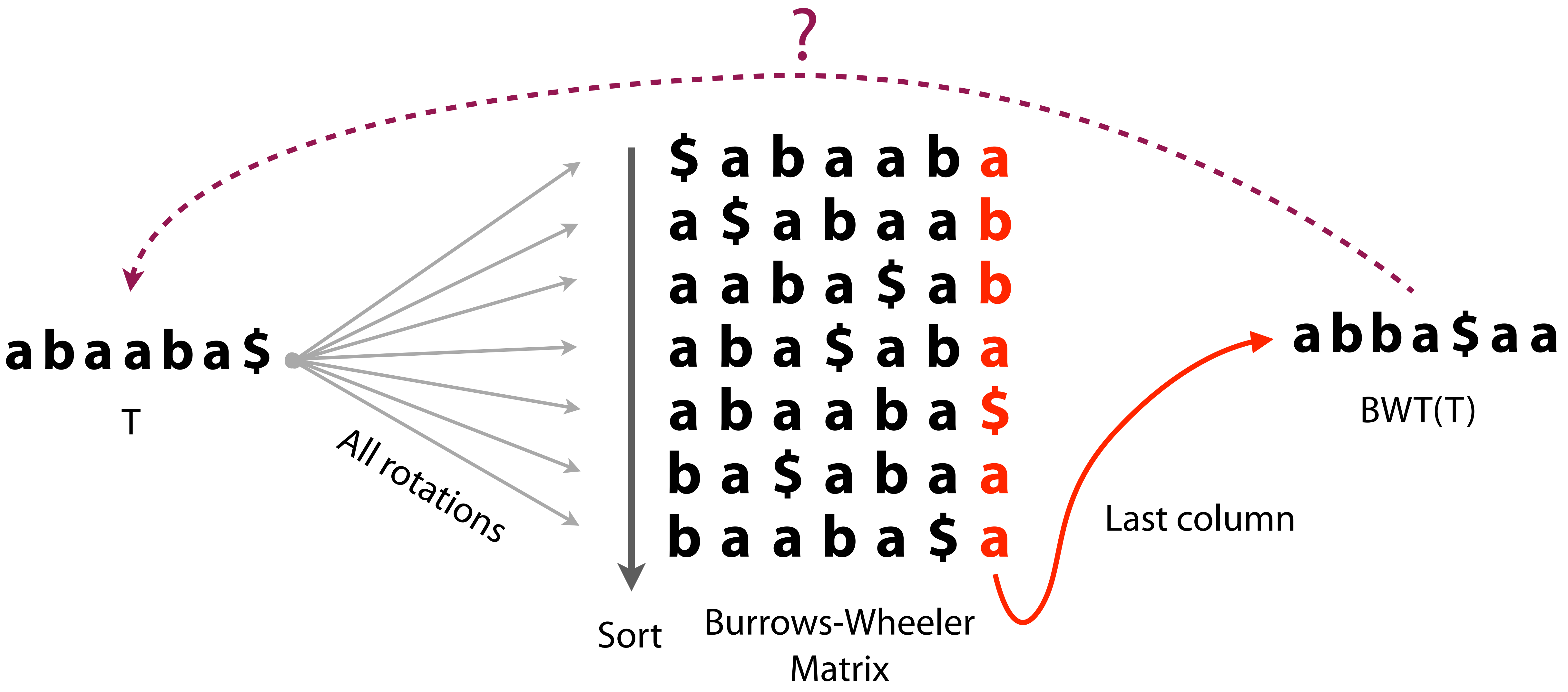
BWM(T)

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

SA(T)

Burrows-Wheeler Transform

How to reverse the BWT?



```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

Burrows-Wheeler Matrix

BWM has a key property called the *LF Mapping*...

Burrows-Wheeler Transform: T-ranking

Give each character in *T* a rank, equal to # times the character occurred previously in *T*. Call this the *T-ranking*.

$$\mathbf{a}_0\ \mathbf{b}_0\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{b}_1\ \mathbf{a}_3\ \$$$

Now let's re-write the BWM including ranks...

Note: we *do not* actually write this information in the text / BWM, we Are simply including it here to help us track "which" occurrences of each character in the BWM correspond to the occurrences in the text.

Burrows-Wheeler Transform

BWM with T-ranking:

F						*L*
\$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	\$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_3	\$	a_0	b_0
a_2	b_1	a_3	\$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	\$
b_1	a_3	\$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	\$	a_0

Look at first and last columns, called *F* and *L*

And look at just the **a**s

as occur in the same order in *F* and *L*. As we look down columns, in both cases we see: a_3, a_1, a_2, a_0

Burrows-Wheeler Transform

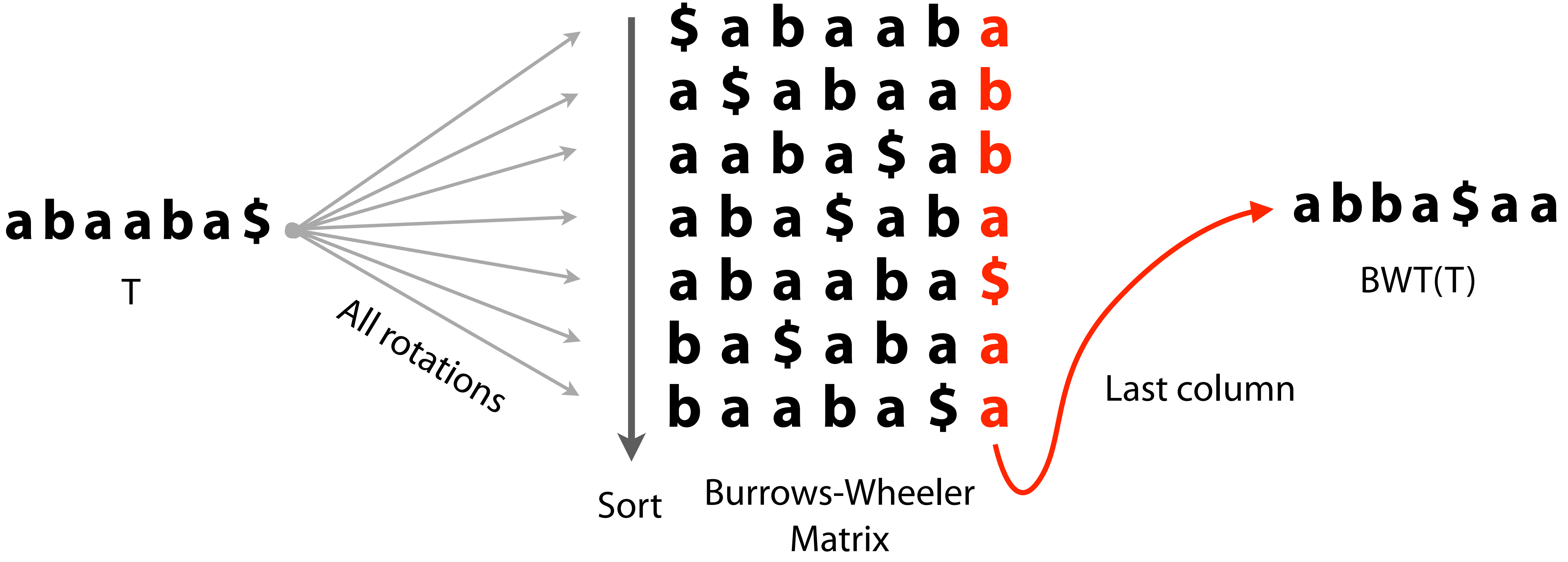
BWM with T-ranking:

$$\begin{array}{ccccccc}
F & & & & & & L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & \mathbf{b_1} \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & \mathbf{b_0} \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
\mathbf{b_1} & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
\mathbf{b_0} & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}$$

Same with **b**s: $\mathbf{b_1}$, $\mathbf{b_0}$

Burrows-Wheeler Transform

Reversible permutation of the characters of a string, used originally for compression



a b a a b a $

T

All rotations

Sort

```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

Burrows-Wheeler Matrix

Last column

a b b a $ a a

BWT(T)

How is it useful for compression? How is it reversible? How is it an index?

Burrows M, Wheeler DJ: A block sorting lossless data compression algorithm.
Digital Equipment Corporation, Palo Alto, CA 1994, Technical Report 124; 1994

Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

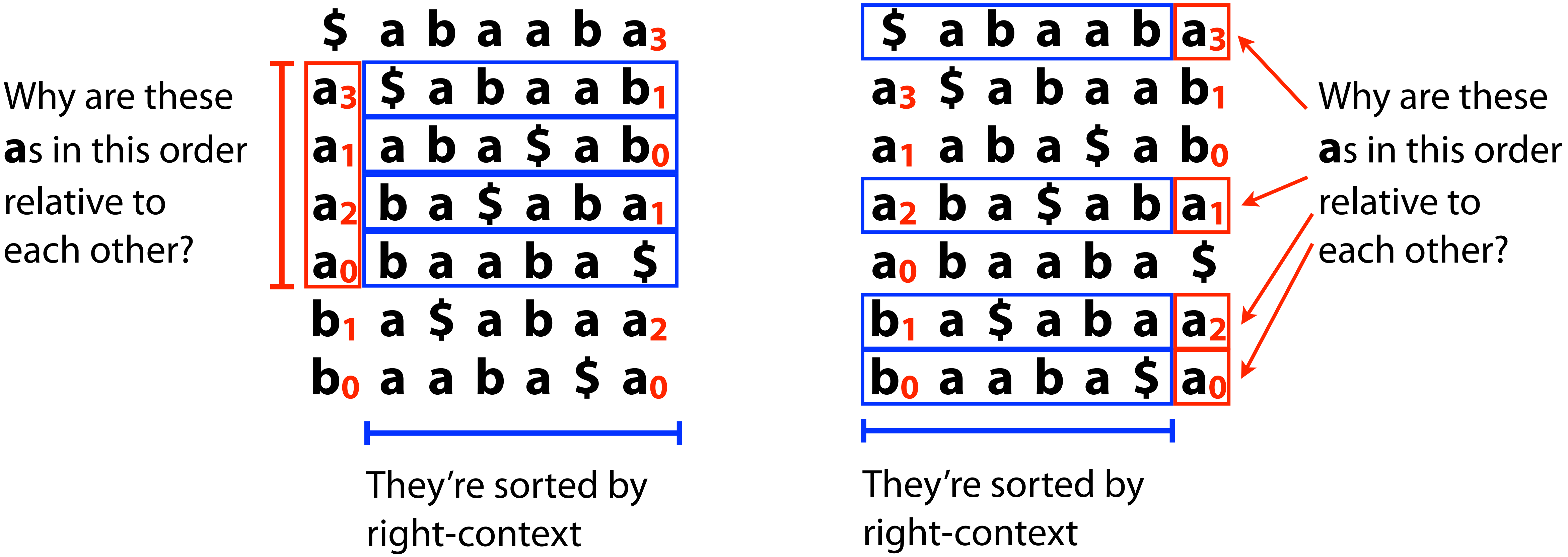
$$\begin{array}{ccccccc}
F & & & & & & L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & b_0 \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
b_1 & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}$$

LF Mapping: The i^{th} occurrence of a character c in L and the i^{th} occurrence of c in F correspond to the *same* occurrence in T

However we rank occurrences of c, ranks appear in the same order in F and L

Burrows-Wheeler Transform: LF Mapping

Why does the LF Mapping hold?



Why are these **a**s in this order relative to each other?

$\$\ a\ b\ a\ a\ b\ a_3$

$a_3\ \$\ a\ b\ a\ a\ b_1$

$a_1\ a\ b\ a\ \$\ a\ b_0$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_0\ b\ a\ a\ b\ a\ \$$

$b_1\ a\ \$\ a\ b\ a\ a_2$

$b_0\ a\ a\ b\ a\ \$\ a_0$

They're sorted by right-context

$\$\ a\ b\ a\ a\ b\ a_3$

$a_3\ \$\ a\ b\ a\ a\ b_1$

$a_1\ a\ b\ a\ \$\ a\ b_0$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_0\ b\ a\ a\ b\ a\ \$$

$b_1\ a\ \$\ a\ b\ a\ a_2$

$b_0\ a\ a\ b\ a\ \$\ a_0$

They're sorted by right-context

Why are these **a**s in this order relative to each other?

Occurrences of *c* in *F* are sorted by right-context. Same for *L*!

Whatever ranking we give to characters in *T*, rank orders in *F* and *L* will match

Burrows-Wheeler Transform: LF Mapping

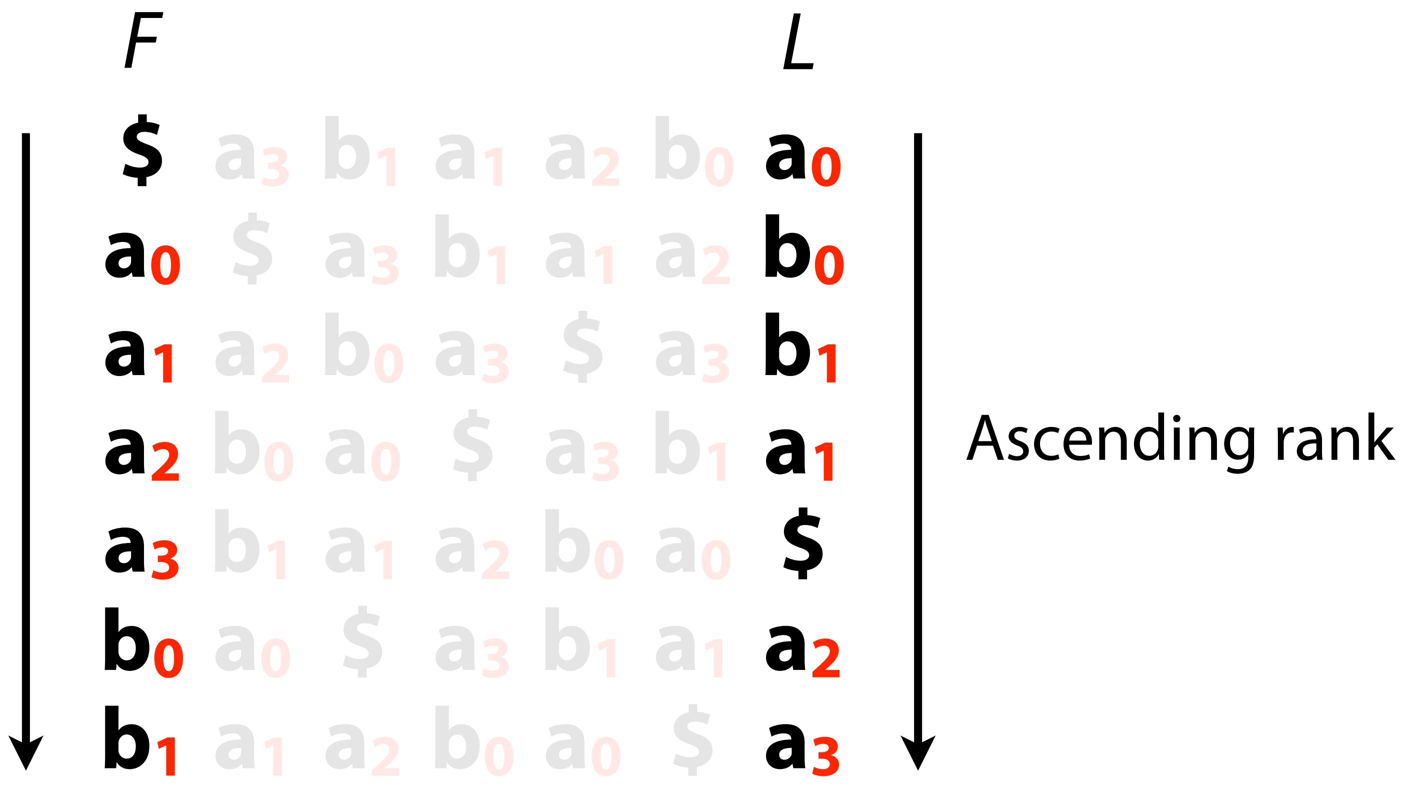
BWM with T-ranking:

$$\begin{array}{ccccccc}
F & & & & & & L \\
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & b_0 \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
b_1 & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}$$

We'd like a different ranking so that for a given character, ranks are in ascending order as we look down the F / L columns...

Burrows-Wheeler Transform: LF Mapping

BWM with B-ranking:



F						L
$\$$	a_3	b_1	a_1	a_2	b_0	a_0
a_0	$\$$	a_3	b_1	a_1	a_2	b_0
a_1	a_2	b_0	a_3	$\$$	a_3	b_1
a_2	b_0	a_0	$\$$	a_3	b_1	a_1
a_3	b_1	a_1	a_2	b_0	a_0	$\$$
b_0	a_0	$\$$	a_3	b_1	a_1	a_2
b_1	a_1	a_2	b_0	a_0	$\$$	a_3

F now has very simple structure: a **$\$$**, a block of **a**s *with ascending ranks*, a block of **b**s *with ascending ranks*

Burrows-Wheeler Transform

	F	*L*	
	$\mathbf{\$}$	$\mathbf{a}_0$	
	$\mathbf{a}_0$	$\mathbf{b}_0$	
	$\mathbf{a}_1$	$\mathbf{b}_1$	← Which BWM row *begins* with $\mathbf{b}_1$?
	$\mathbf{a}_2$	$\mathbf{a}_1$	
	$\mathbf{a}_3$	$\mathbf{\$}$	
	$\mathbf{b}_0$	$\mathbf{a}_2$	
row 6 →	$\mathbf{b}_1$	$\mathbf{a}_3$	

Skip row starting with **$** (1 row)

Skip rows starting with **a** (4 rows)

Skip row starting with $\mathbf{b}_0$ (1 row)

Answer: row 6

Burrows-Wheeler Transform

Say *T* has 300 **A**s, 400 **C**s, 250 **G**s and 700 **T**s and **$** < **A** < **C** < **G** < **T**

Which BWM row (0-based) begins with $\mathbf{G}_{100}$? (Ranks are B-ranks.)

Skip row starting with **$** (1 row)

Skip rows starting with **A** (300 rows)

Skip rows starting with **C** (400 rows)

Skip first 100 rows starting with **G** (100 rows)

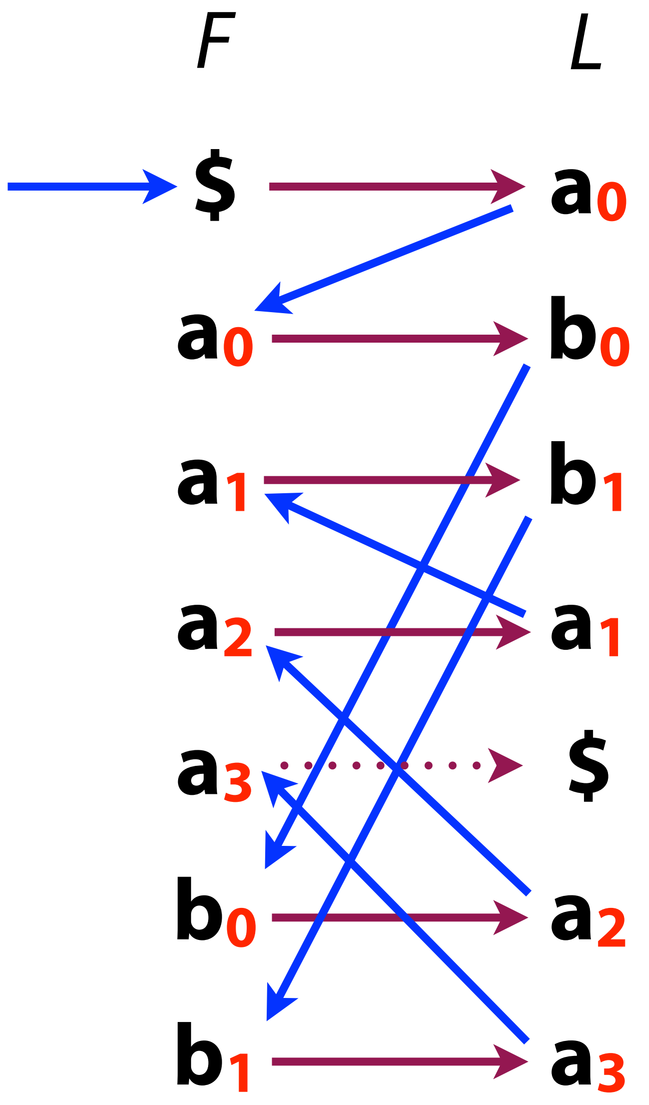
Answer: row 1 + 300 + 400 + 100 = **row 801**

Burrows-Wheeler Transform: reversing

Reverse BWT(T) starting at right-hand-side of *T* and moving left

Start in first row. *F* must have **$**. *L* contains character just prior to **$**: $\mathbf{a}_0$

$\mathbf{a}_0$: LF Mapping says this is same occurrence of **a** as first **a** in *F*. Jump to row *beginning* with $\mathbf{a}_0$. *L* contains character just prior to $\mathbf{a}_0$: $\mathbf{b}_0$.



Repeat for $\mathbf{b}_0$, get $\mathbf{a}_2$

Repeat for $\mathbf{a}_2$, get $\mathbf{a}_1$

Repeat for $\mathbf{a}_1$, get $\mathbf{b}_1$

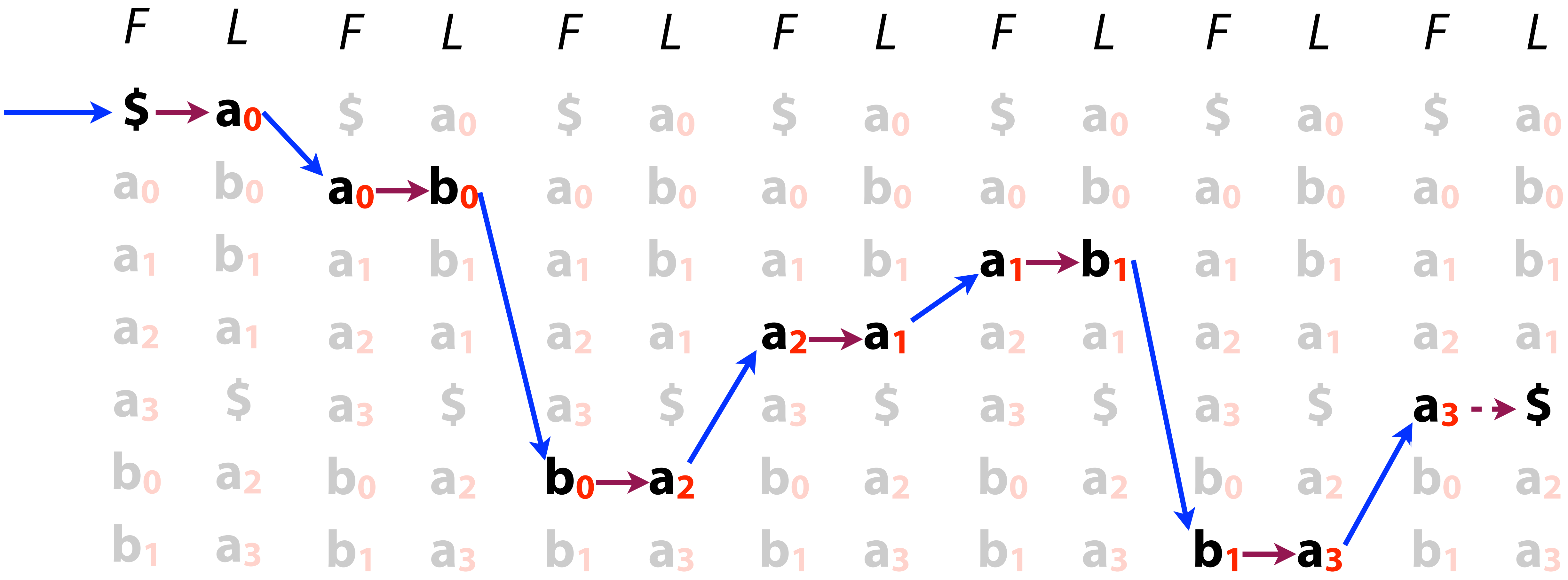
Repeat for $\mathbf{b}_1$, get $\mathbf{a}_3$

Repeat for $\mathbf{a}_3$, get **$**, done

Reverse of chars we visited = $\mathbf{a}_3\ \mathbf{b}_1\ \mathbf{a}_1\ \mathbf{a}_2\ \mathbf{b}_0\ \mathbf{a}_0$ **$** = *T*

Burrows-Wheeler Transform: reversing

Another way to visualize reversing BWT(T):



F	*L*	*F*	*L*	*F*	*L*	*F*	*L*	*F*	*L*	*F*	*L*	*F*	*L*
\$	a_0	\$	a_0	\$	a_0	\$	a_0	\$	a_0	\$	a_0	\$	a_0
a_0	b_0	a_0	b_0	a_0	b_0	a_0	b_0	a_0	b_0	a_0	b_0	a_0	b_0
a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1	a_1	b_1
a_2	a_1	a_2	a_1	a_2	a_1	a_2	a_1	a_2	a_1	a_2	a_1	a_2	a_1
a_3	\$	a_3	\$	a_3	\$	a_3	\$	a_3	\$	a_3	\$	a_3	\$
b_0	a_2	b_0	a_2	b_0	a_2	b_0	a_2	b_0	a_2	b_0	a_2	b_0	a_2
b_1	a_3	b_1	a_3	b_1	a_3	b_1	a_3	b_1	a_3	b_1	a_3	b_1	a_3

T: $a_3\ b_1\ a_1\ a_2\ b_0\ a_0\ \$$

Burrows-Wheeler Transform: reversing

```
>>> reverseBwt("w$wwdd__nnoooaattTmmmrrrrrrooo__ooo")
'Tomorrow_and_tomorrow_and_tomorrow$'

>>> reverseBwt("s$esttssfftteww_hhmmboottt_ii__woeeaaressIi_______")
'It_was_the_best_of_times_it_was_the_worst_of_times$'

>>> reverseBwt("u_gleeeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmylo_oo_")
'in_the_jingle_jangle_morning_Ill_come_following_you$'
```

ranks list is *m* integers long! We'll fix later. →

```
def reverseBwt(bw):
    ''' Make T from BWT(T) '''
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0 # start in first row
    t = '$' # start with rightmost character
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t # prepend to answer
        # jump to row that starts with c of same rank
        rowi = first[c][0] + ranks[rowi]
    return t
```

Burrows-Wheeler Transform

We’ve seen how BWT is useful for compression:

- Sorts characters by right-context, making a more compressible string

And how it’s reversible:

- Repeated applications of LF Mapping, recreating *T* from right to left

How is it used as an index?

FM Index

FM Index: an index combining the BWT *with a few small auxilliary data structures*

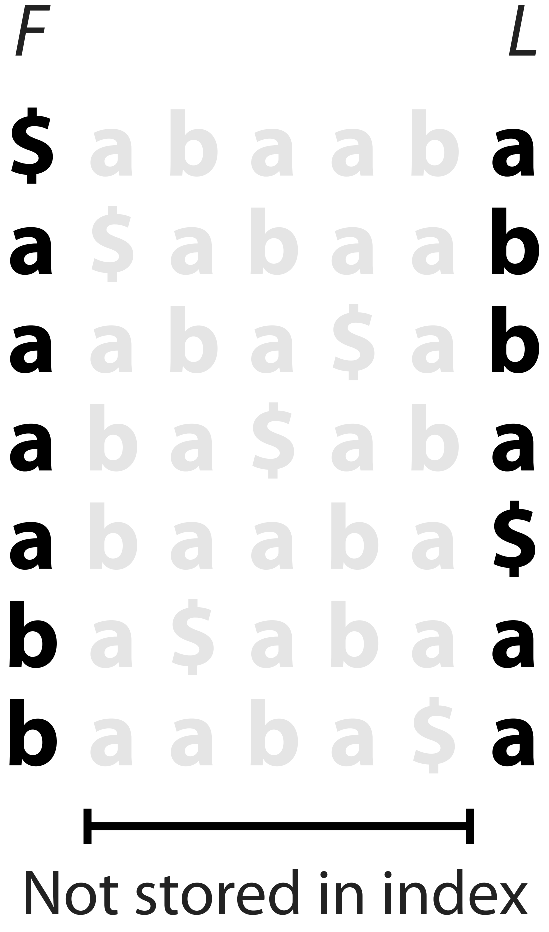
"FM" supposedly stands for "Full-text Minute-space."
(But inventors are named Ferragina and Manzini)

Core of index consists of *F* and *L* from BWM:

F can be represented very simply
(1 integer per alphabet character)

And *L* is compressible

Potentially very space-economical!



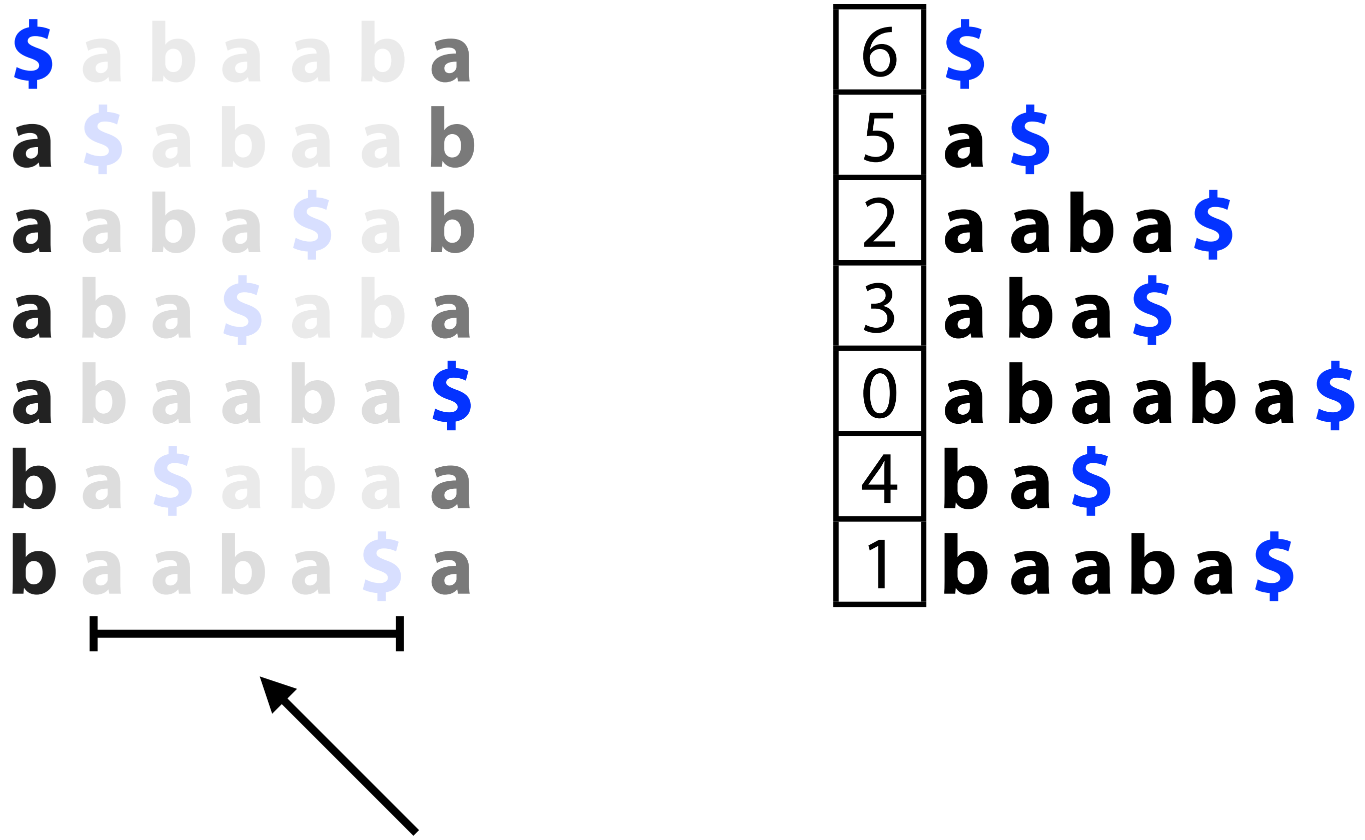
F						*L*
$	a	b	a	a	b	a
a	$	a	b	a	a	b
a	a	b	a	$	a	b
a	b	a	$	a	b	a
a	b	a	a	b	a	$
b	a	$	a	b	a	a
b	a	a	b	a	$	a

Not stored in index

Paolo Ferragina, and Giovanni Manzini. "Opportunistic data structures with applications." *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000.

FM Index: querying

Though BWM is related to suffix array, we can't query it the same way



```
$ a b a a b a
a $ a b a a b
a a b a $ a b
a b a $ a b a
a b a a b a $
b a $ a b a a
b a a b a $ a
```

6	$
5	a $
2	a a b a $
3	a b a $
0	a b a a b a $
4	b a $
1	b a a b a $

We don't have these columns; binary search isn't possible

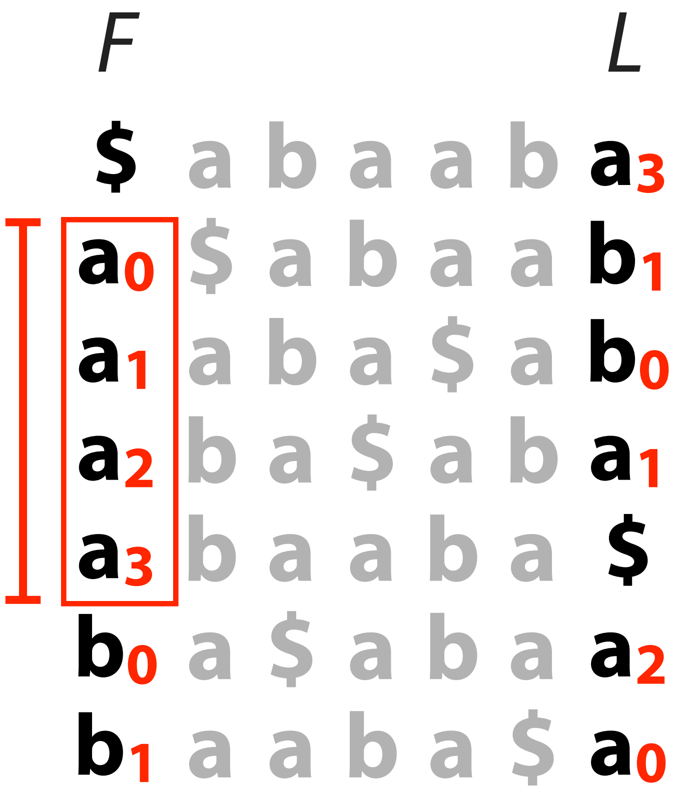
FM Index: querying

Look for range of rows of BWM(T) with *P* as prefix

Do this for *P*'s shortest suffix, then extend to successively longer suffixes until range becomes empty or we've exhausted *P*

P = **aba**

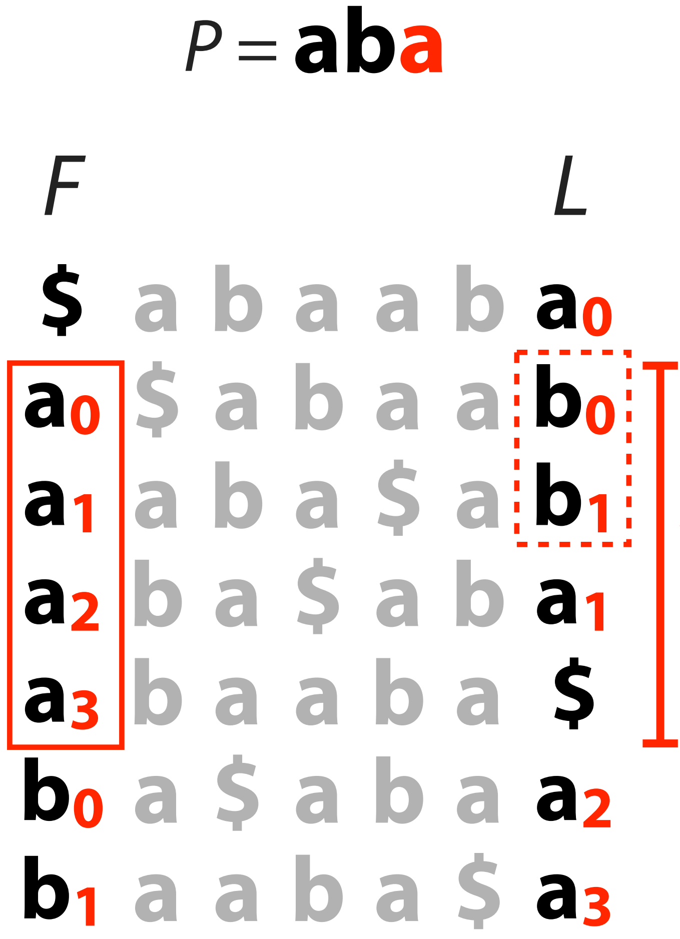
Easy to find all the rows beginning with **a**, thanks to *F*'s simple structure



F						*L*
$	a	b	a	a	b	a_3
a_0	$	a	b	a	a	b_1
a_1	a	b	a	$	a	b_0
a_2	b	a	$	a	b	a_1
a_3	b	a	a	b	a	**$**
b_0	a	$	a	b	a	a_2
b_1	a	a	b	a	$	a_0

FM Index: querying

We have rows beginning with **a**, now we seek rows beginning with **ba**



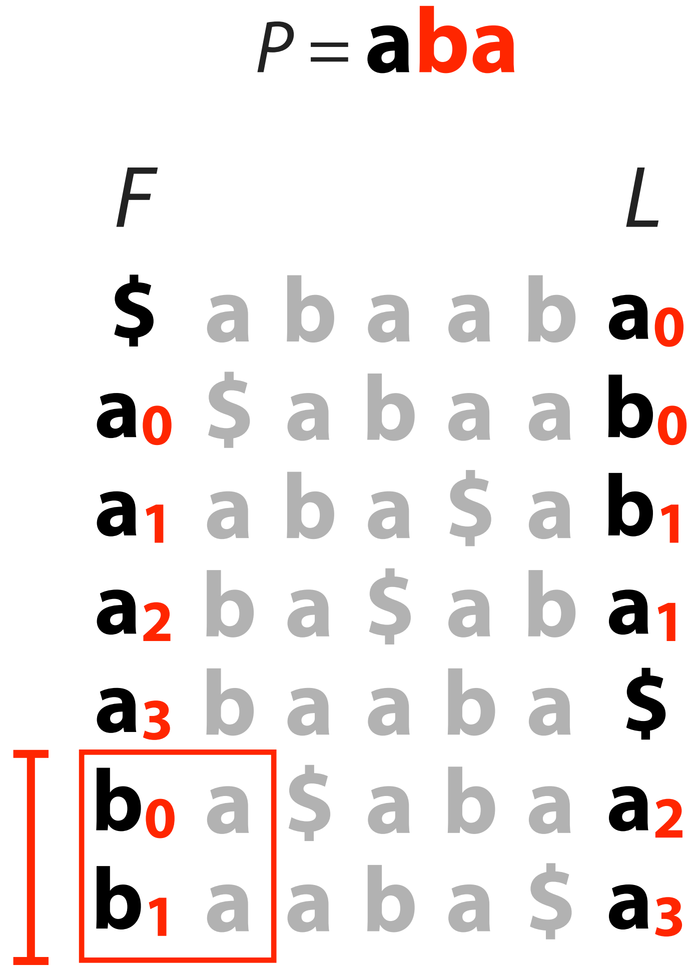
P = **aba**

F						*L*
$\$$	a	b	a	a	b	a_0
a_0	$\$$	a	b	a	a	b_0
a_1	a	b	a	$\$$	a	b_1
a_2	b	a	$\$$	a	b	a_1
a_3	b	a	a	b	a	$\$$
b_0	a	$\$$	a	b	a	a_2
b_1	a	a	b	a	$\$$	a_3

Look at those rows in *L*.

b_0, **b_1** are **b**s occuring just to left.

Use LF Mapping. Let new range delimit those **b**s



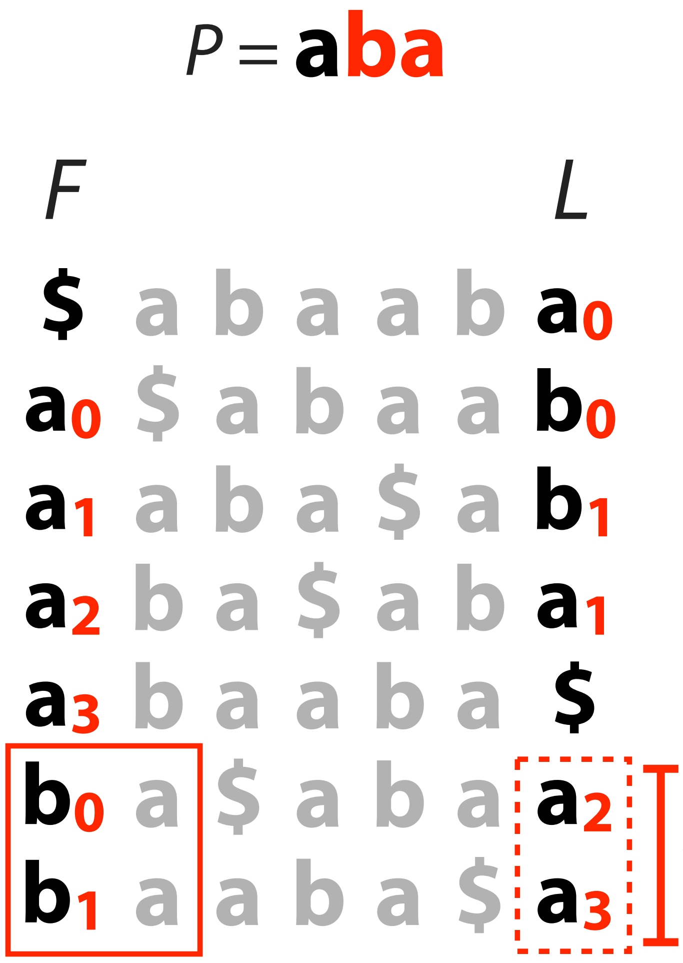
P = **aba**

F						*L*
$\$$	a	b	a	a	b	a_0
a_0	$\$$	a	b	a	a	b_0
a_1	a	b	a	$\$$	a	b_1
a_2	b	a	$\$$	a	b	a_1
a_3	b	a	a	b	a	$\$$
b_0	a	$\$$	a	b	a	a_2
b_1	a	a	b	a	$\$$	a_3

Now we have the rows with prefix **ba**

FM Index: querying

We have rows beginning with **ba**, now we seek rows beginning with **aba**

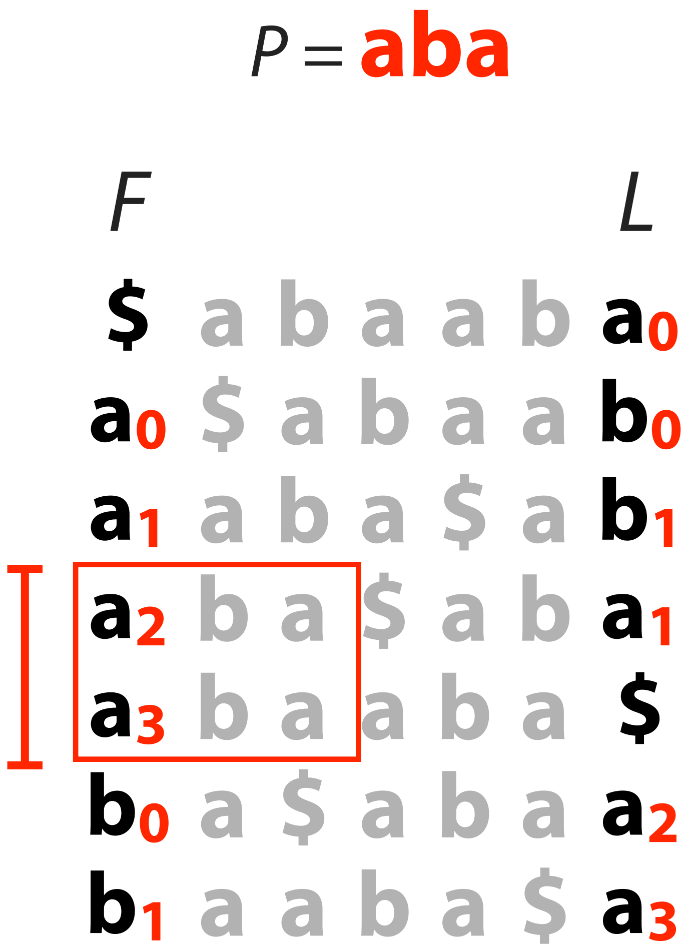


$P =$ **aba**

F						*L*
$	a	b	a	a	b	a_0
a_0	$	a	b	a	a	b_0
a_1	a	b	a	$	a	b_1
a_2	b	a	$	a	b	a_1
a_3	b	a	a	b	a	$
b_0	a	$	a	b	a	a_2
b_1	a	a	b	a	$	a_3

← a_2, a_3 occur just to left.

Use LF Mapping →



$P =$ **aba**

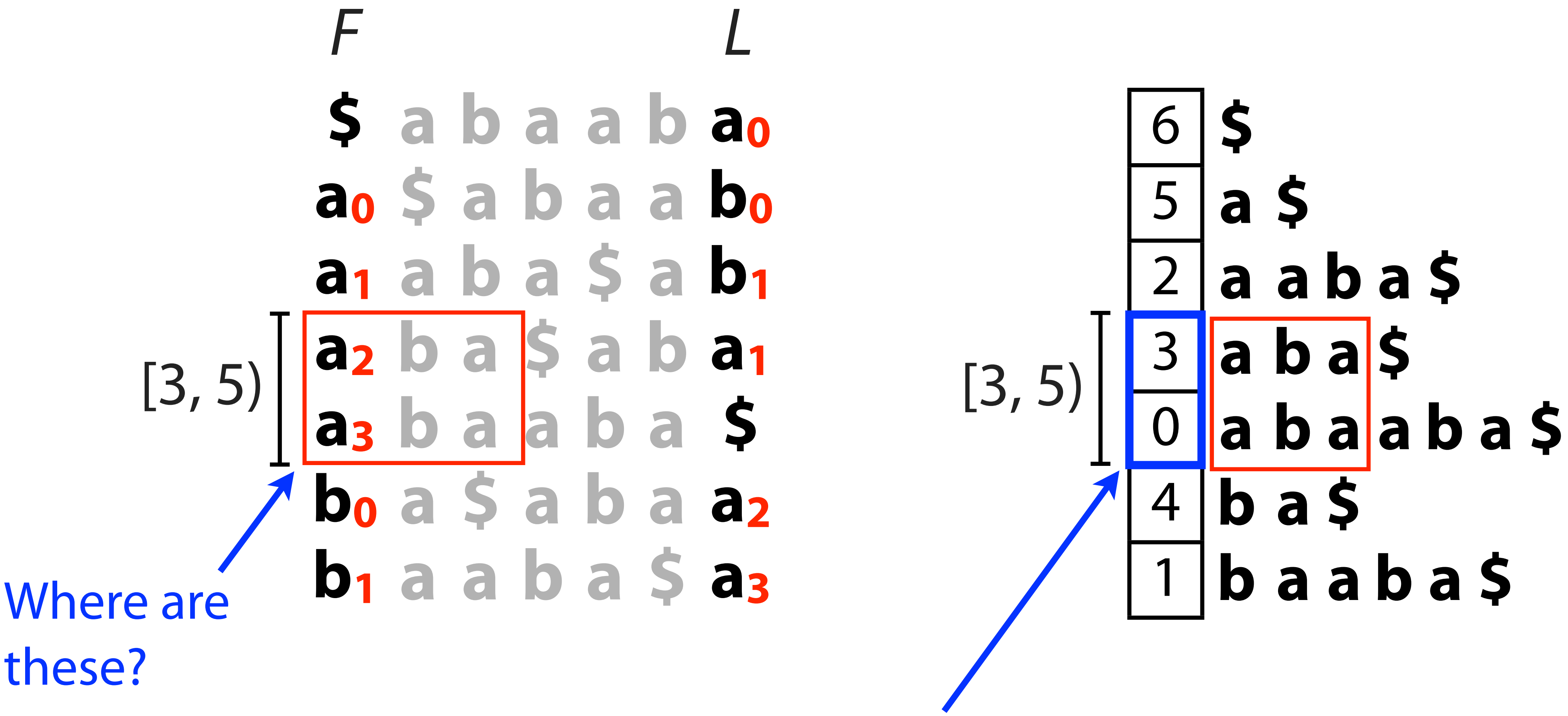
F						*L*
$	a	b	a	a	b	a_0
a_0	$	a	b	a	a	b_0
a_1	a	b	a	$	a	b_1
a_2	b	a	$	a	b	a_1
a_3	b	a	a	b	a	$
b_0	a	$	a	b	a	a_2
b_1	a	a	b	a	$	a_3

Now we have the rows with prefix **aba**

FM Index: querying

P = **aba**

Now we have the same range, [3, 5), we would have got from querying suffix array



F						*L*
$	a	b	a	a	b	**a_0**
a_0	$	a	b	a	a	**b_0**
a_1	a	b	a	$	a	**b_1**
a_2	b	a	$	a	b	**a_1**
a_3	b	a	a	b	a	**$**
b_0	a	$	a	b	a	**a_2**
b_1	a	a	b	a	$	**a_3**

6	**$**
5	**a $**
2	**a a b a $**
3	**a b a $**
0	**a b a a b a $**
4	**b a $**
1	**b a a b a $**

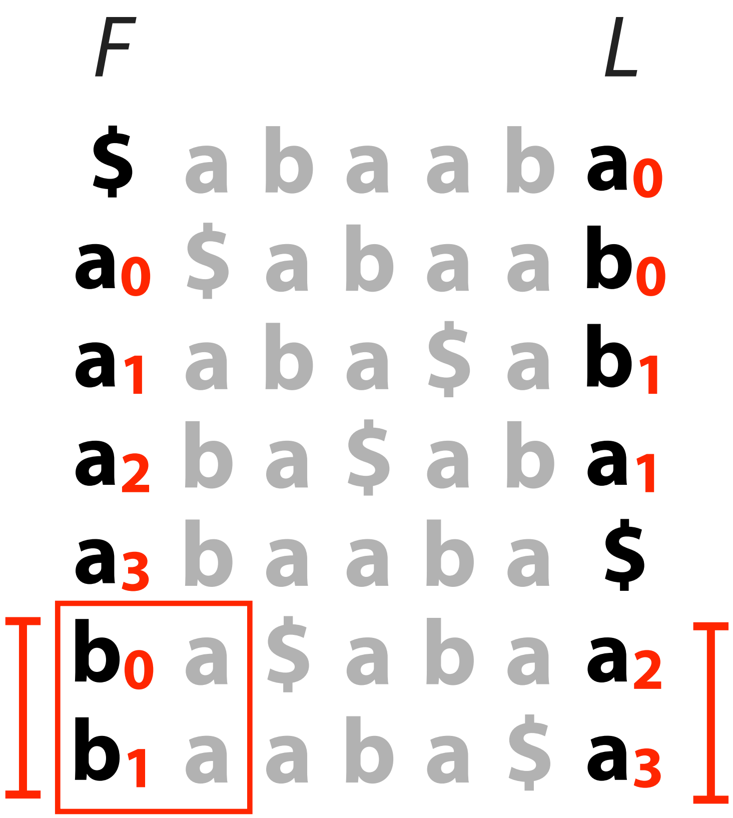
Where are these?

Unlike suffix array, we don’t immediately know *where* the matches are in T...

FM Index: querying

When *P* does not occur in *T*, we will eventually fail to find the next character in *L*:

P = **bba**



F						*L*
\$	a	b	a	a	b	a_0
a_0	\$	a	b	a	a	b_0
a_1	a	b	a	\$	a	b_1
a_2	b	a	\$	a	b	a_1
a_3	b	a	a	b	a	\$
b_0	a	\$	a	b	a	a_2
b_1	a	a	b	a	\$	a_3

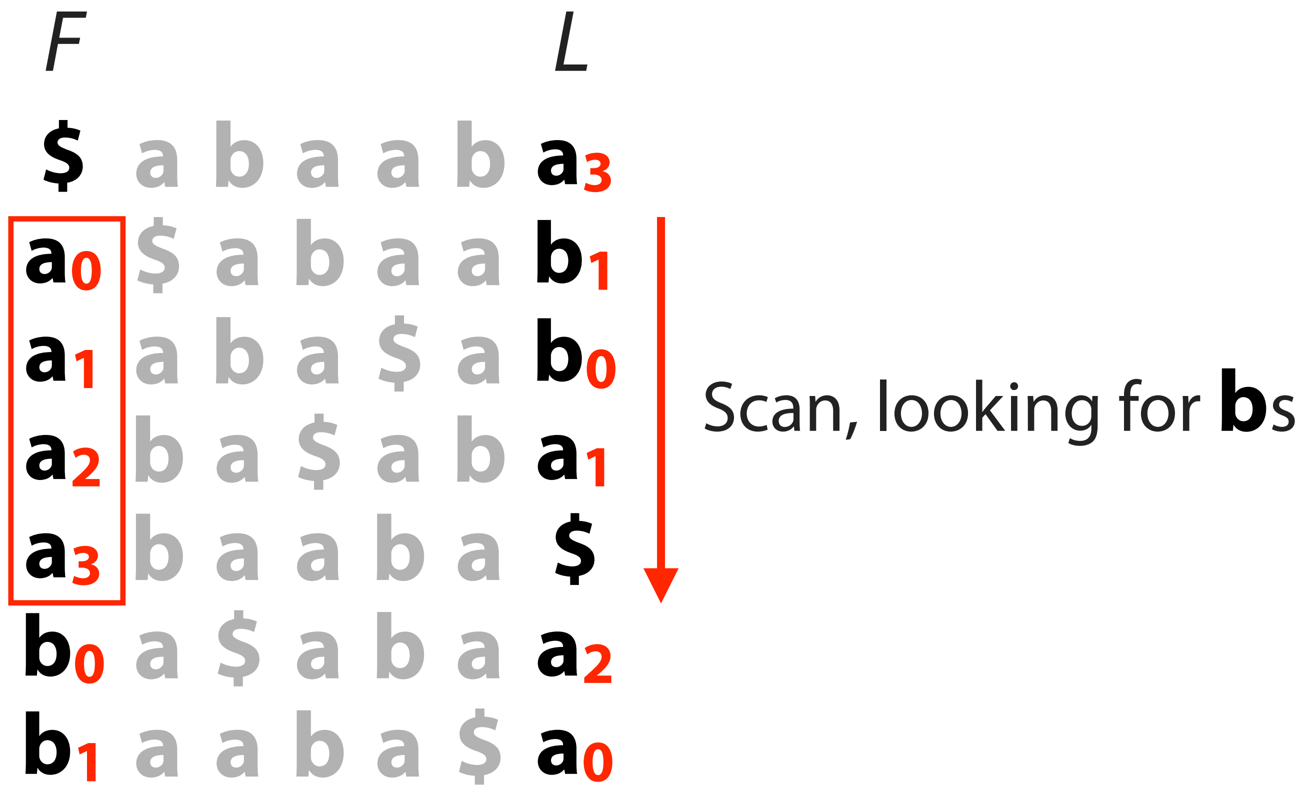
Rows with **ba** prefix

← No **b**s!

FM Index: querying

If we *scan* characters in the last column, that can be very slow, $O(m)$

$P =$ **ab**a

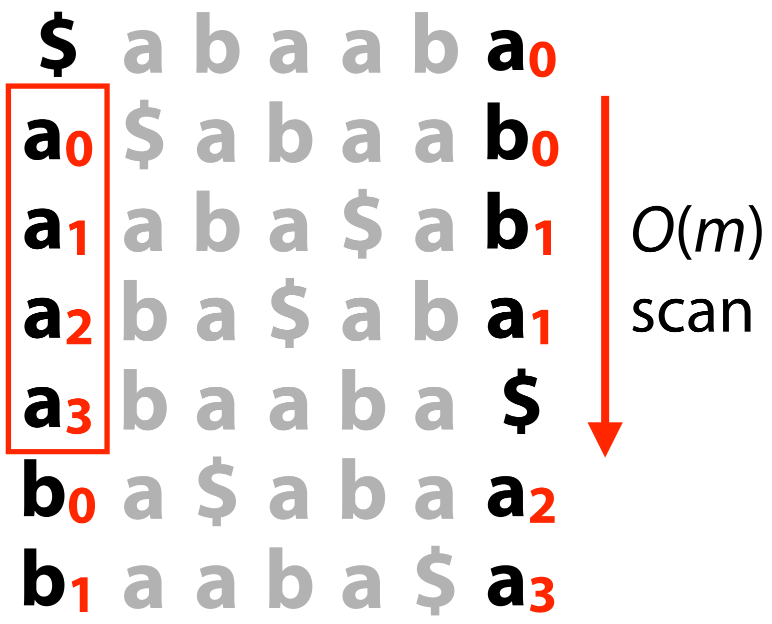


F						*L*
$	a	b	a	a	b	$\mathbf{a}_3$
$\mathbf{a}_0$	$	a	b	a	a	$\mathbf{b}_1$
$\mathbf{a}_1$	a	b	a	$	a	$\mathbf{b}_0$
$\mathbf{a}_2$	b	a	$	a	b	$\mathbf{a}_1$
$\mathbf{a}_3$	b	a	a	b	a	**$**
$\mathbf{b}_0$	a	$	a	b	a	$\mathbf{a}_2$
$\mathbf{b}_1$	a	a	b	a	$	$\mathbf{a}_0$

Scan, looking for **b**s

FM Index: lingering issues

(1) Scanning for preceding character is slow



$\$ \;\; a \; b \; a \; a \; b \;\; a_0$

$a_0 \;\; \$ \; a \; b \; a \; a \;\; b_0$

$a_1 \;\; a \; b \; a \; \$ \; a \;\; b_1$

$a_2 \;\; b \; a \; \$ \; a \; b \;\; a_1$

$a_3 \;\; b \; a \; a \; b \; a \;\; \$$

$b_0 \;\; a \; \$ \; a \; b \; a \;\; a_2$

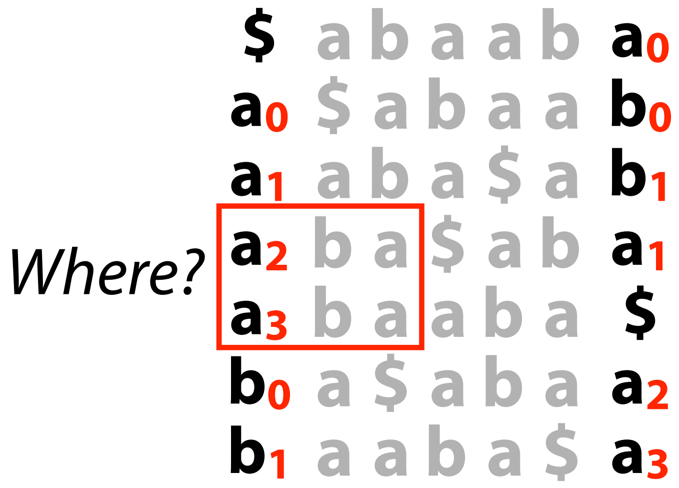
$b_1 \;\; a \; a \; b \; a \; \$ \;\; a_3$

(2) Storing ranks takes too much space

m integers

```
def reverseBwt(bw):
    """ Make T from BWT(T) """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```

(3) Need way to find where matches occur in *T*:



$\$ \;\; a \; b \; a \; a \; b \;\; a_0$

$a_0 \;\; \$ \; a \; b \; a \; a \;\; b_0$

$a_1 \;\; a \; b \; a \; \$ \; a \;\; b_1$

$a_2 \;\; b \; a \; \$ \; a \; b \;\; a_1$

$a_3 \;\; b \; a \; a \; b \; a \;\; \$$

$b_0 \;\; a \; \$ \; a \; b \; a \;\; a_2$

$b_1 \;\; a \; a \; b \; a \; \$ \;\; a_3$

FM Index: fast rank calculations

Is there an O(1) way to determine which **b**s precede the **a**s in our range?

F						*L*
$	a	b	a	a	b	**a_0**
a_0	$	a	b	a	a	**b_0**
a_1	a	b	a	$	a	**b_1**
a_2	b	a	$	a	b	**a_1**
a_3	b	a	a	b	a	**$**
b_0	a	$	a	b	a	**a_2**
b_1	a	a	b	a	$	**a_3**

Occ(*c*, *k*) = # of of c in the first *k* characters of BWT(S), aka the LF mapping.

Tally — also referred to as **Occ**(c, k)

Idea: pre-calculate # **a**s, **b**s in *L* up to every row:

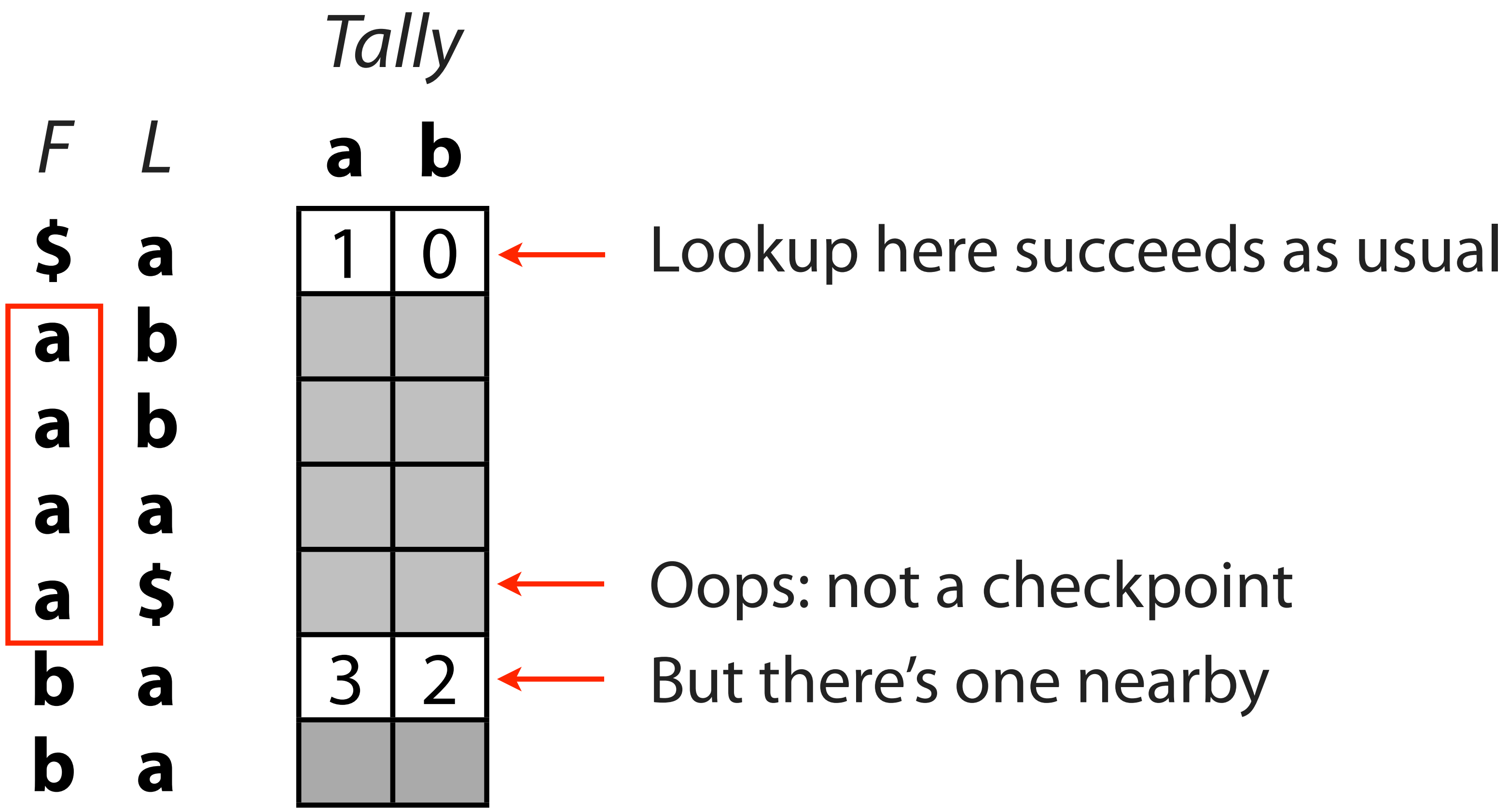
F	*L*	**a**	**b**
$	**a**	1	0
a	**b**	1	1
a	**b**	1	2
a	**a**	2	2
a	**$**	2	2
b	**a**	3	2
b	**a**	4	2

We infer **b_0** and **b_1** appear in *L* in this range

O(1) time, but requires $m \times |\Sigma|$ integers

FM Index: fast rank calculations

Another idea: pre-calculate # **a**s, **b**s in *L* up to *some* rows, e.g. every 5th row. Call pre-calculated rows *checkpoints*.



Tally

F	*L*	**a**	**b**
$	**a**	1	0
a	**b**		
a	**b**		
a	**a**		
a	**$**		
b	**a**	3	2
b	**a**		

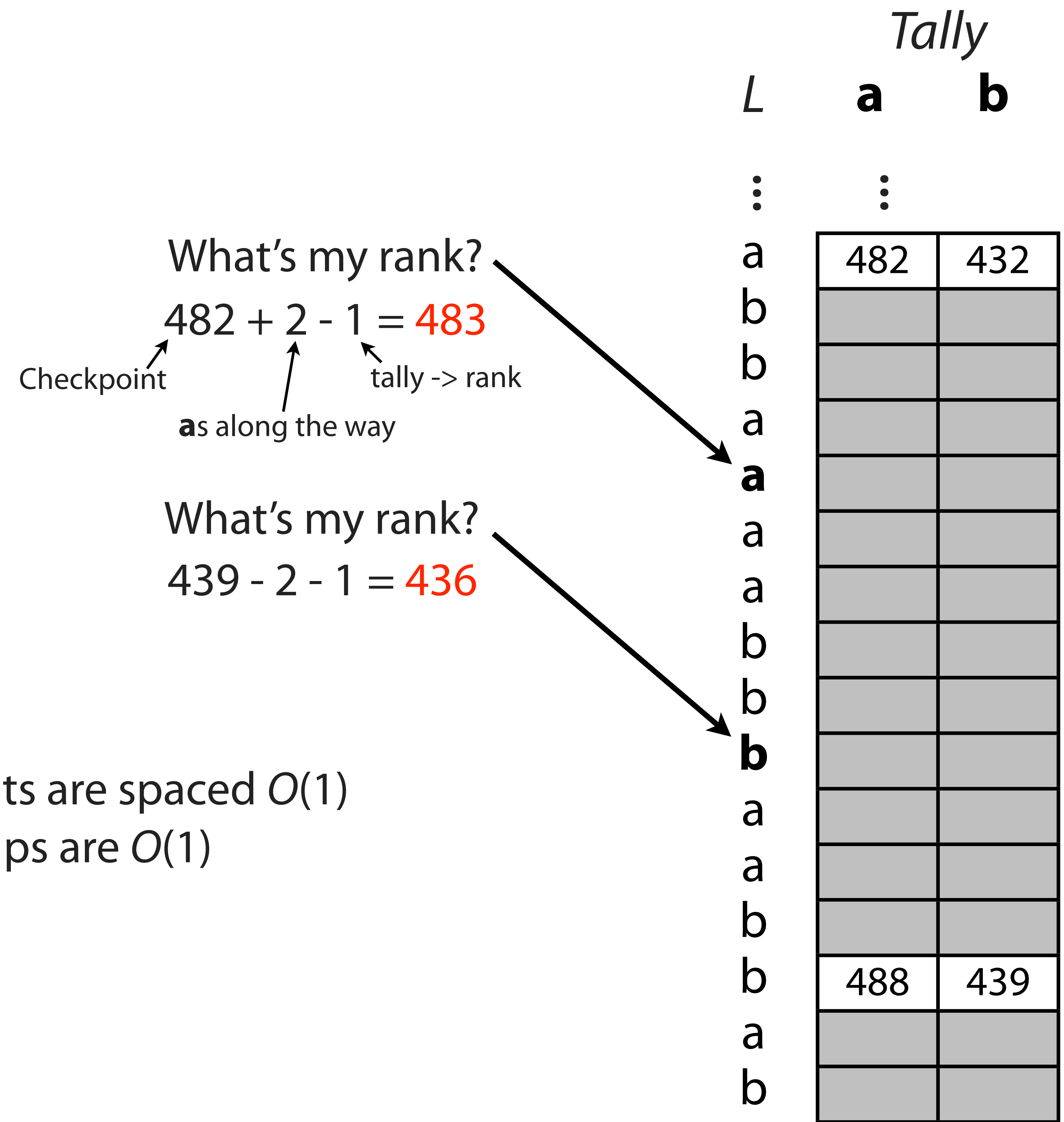
Lookup here succeeds as usual

Oops: not a checkpoint

But there's one nearby

To resolve a lookup for character *c* in non-checkpoint row, scan along *L* until we get to nearest checkpoint. Use tally at the checkpoint, *adjusted for # of cs we saw along the way*.

FM Index: fast rank calculations



What’s my rank?

482 + 2 - 1 = 483

What’s my rank?

439 - 2 - 1 = 436

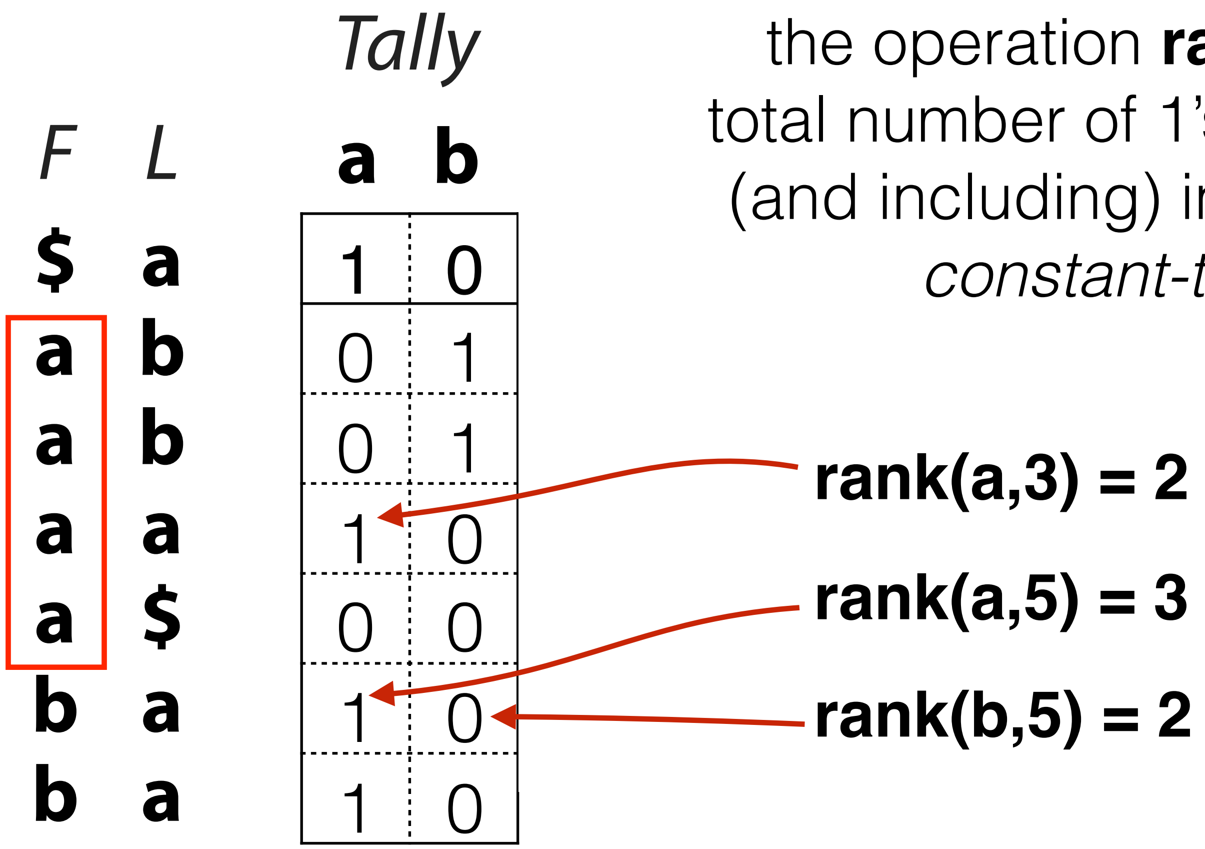
	Tally	
L	**a**	**b**
⋮	⋮	
a	482	432
b		
b		
a		
a		
a		
a		
b		
b		
b		
a		
a		
b		
b	488	439
a		
b		

Assuming checkpoints are spaced *O*(1) distance apart, lookups are *O*(1)

FM Index: fast rank calculations

This can also be accomplished using **bit-vector rank** operations. We store one bit-vector for each character of Σ, placing a 1 where this character occurs and a 0 everywhere else:

the operation **rank(x, i)** returns the total number of 1's in a bit-vector up to (and including) index i. **rank(x,i)** is a *constant-time operation*



		Tally	
F	*L*	**a**	**b**
$	**a**	1	0
a	**b**	0	1
a	**b**	0	1
a	**a**	1	0
a	**$**	0	0
b	**a**	1	0
b	**a**	1	0

rank(a,3) = 2

rank(a,5) = 3

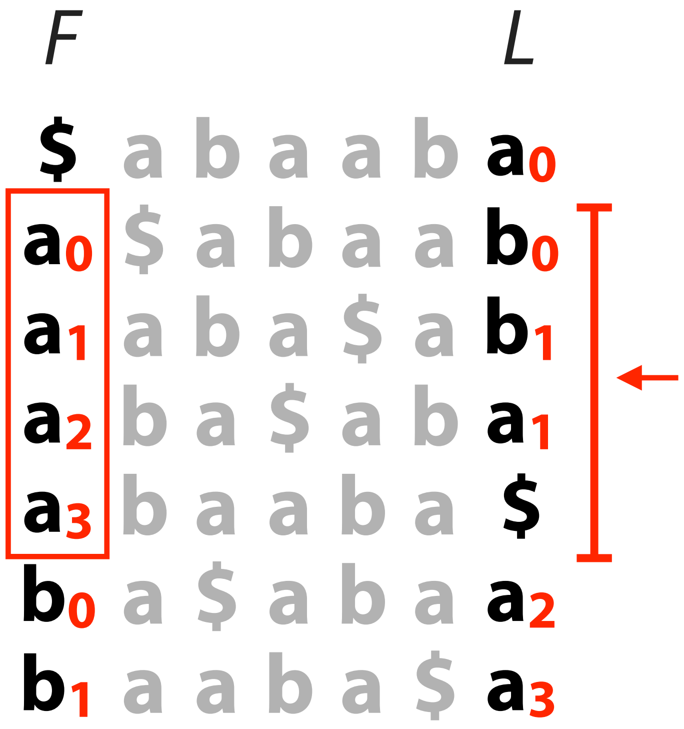
rank(b,5) = 2

To resolve the rank for a given character **c** at a given index **i**, we simply issue a **rank(c,i)** query. This is a practically-fast constant-time operation, but we need to keep around Σ bit-vectors, each of o(m) bits.

FM Index: a few problems

Solved! At the expense of adding checkpoints ($O(m)$ integers) to index.

(1)



$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$a_2\ b\ a\ \$\ a\ b\ a_1$

$a_3\ b\ a\ a\ b\ a\ \$$

$b_0\ a\ \$\ a\ b\ a\ a_2$

$b_1\ a\ a\ b\ a\ \$\ a_3$

This scan is $O(m)$ work

With checkpoints it's $O(1)$

(2) Ranking takes too much space

m integers

```
def reverseBwt(bw):
    """ Make T from BWT(T) """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```

With checkpoints, we greatly reduce # integers needed for ranks

But it's still O(*m*) space - there's literature on how to improve this space bound

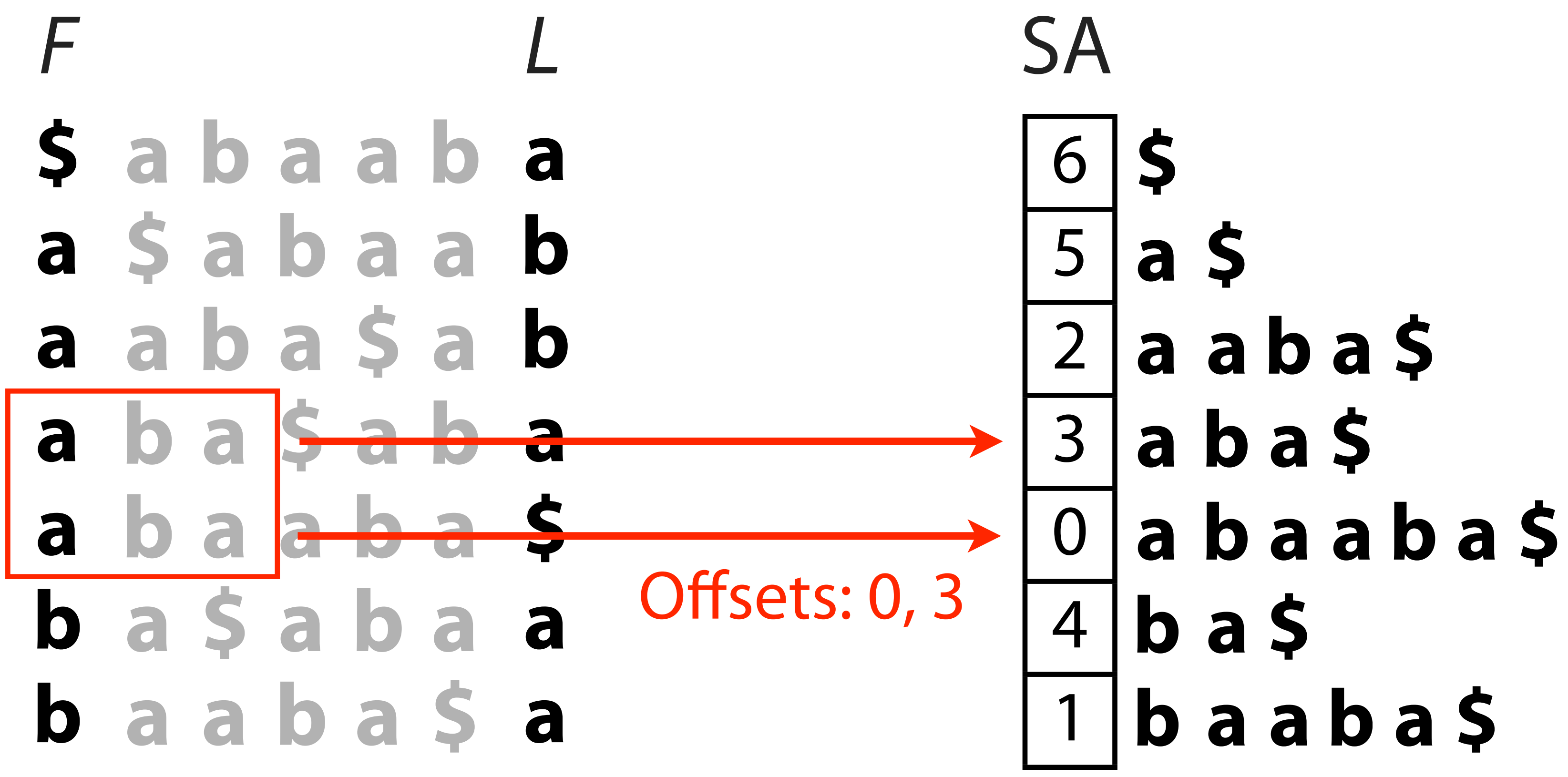
FM Index: a few problems

Not yet solved:

(3) Need a way to find where these occurrences are in *T*:

$\$ \; a\,b\,a\,a\,b \; a_0$
$a_0 \; \$\,a\,b\,a\,a \; b_0$
$a_1 \; a\,b\,a\,\$\,a \; b_1$
$a_2 \; b\,a\,\$\,a\,b \; a_1$
$a_3 \; b\,a\,a\,b\,a \; \$$
$b_0 \; a\,\$\,a\,b\,a \; a_2$
$b_1 \; a\,a\,b\,a\,\$ \; a_3$

If suffix array were part of index, we could simply look up the offsets



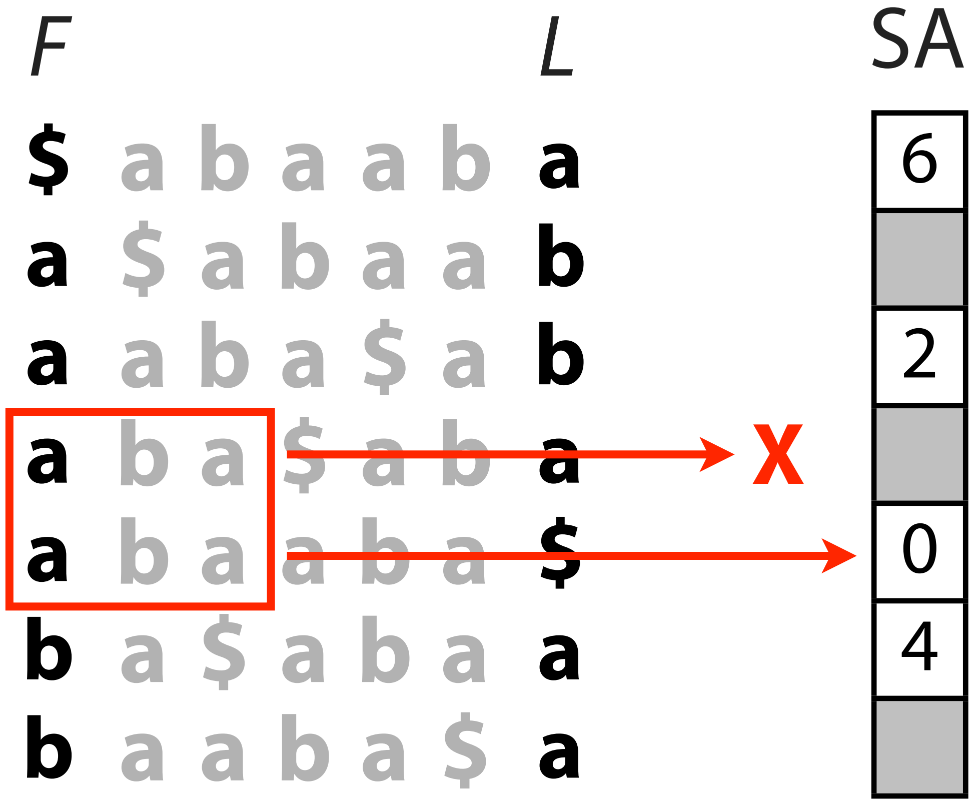
F		*L*	SA	
$	a b a a b	a	6	$
a	$ a b a a	b	5	a $
a	a b a $ a	b	2	a a b a $
a	b a $ a b	a	3	a b a $
a	b a a b a	$	0	a b a a b a $
b	a $ a b a	a	4	b a $
b	a a b a $	a	1	b a a b a $

Offsets: 0, 3

But SA requires *m* integers

FM Index: resolving offsets

Idea: store some, but not all, entries of the suffix array



F						*L*	SA
$	a	b	a	a	b	a	6
a	$	a	b	a	a	b	
a	a	b	a	$	a	b	2
a	b	a	$	a	b	a	
a	b	a	a	b	a	$	0
b	a	$	a	b	a	a	4
b	a	a	b	a	$	a	

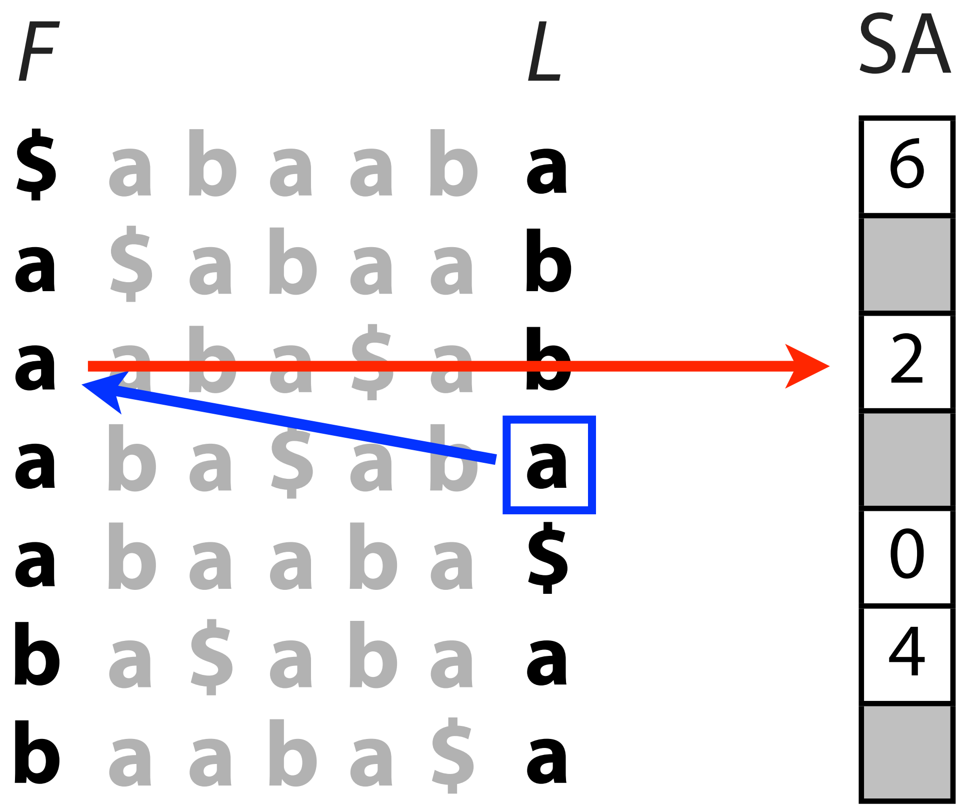
Lookup for row 4 succeeds - we kept that entry of SA

Lookup for row 3 fails - we discarded that entry of SA

FM Index: resolving offsets

But LF Mapping tells us that the **a** at the end of row 3 corresponds to...

...the **a** at the begining of row 2



F						*L*	SA
$	a	b	a	a	b	**a**	6
a	$	a	b	a	a	**b**	
a	a	b	a	$	a	**b**	2
a	b	a	$	a	b	**a**	
a	b	a	a	b	a	**$**	0
b	a	$	a	b	a	**a**	4
b	a	a	b	a	$	**a**	

And row 2 has a suffix array value = 2

So row 3 has suffix array value = 3 = 2 (row 2's SA val) + 1 (# steps to row 2)

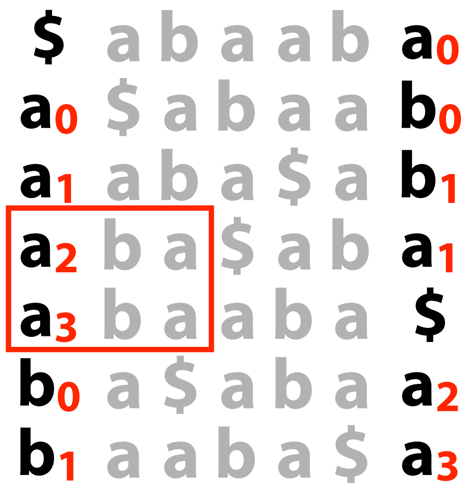
If saved SA values are O(1) positions apart in *T*, resolving offset is O(1) time

FM Index: problems solved

Solved! At the expense of adding some SA values (*O*(*m*) integers) to index

Call this the “SA sample”

(3) Need a way to find where these occurrences are in *T*:



$\$\ a\ b\ a\ a\ b\ a_0$

$a_0\ \$\ a\ b\ a\ a\ b_0$

$a_1\ a\ b\ a\ \$\ a\ b_1$

$\boxed{a_2\ b\ a}\ \$\ a\ b\ a_1$

$\boxed{a_3\ b\ a}\ a\ b\ a\ \$$

$b_0\ a\ \$\ a\ b\ a\ a_2$

$b_1\ a\ a\ b\ a\ \$\ a_3$

With SA sample we can do this in *O*(1) time per occurrence

FM Index: small memory footprint

Components of the FM Index:

First column (*F*):	$\sim \lvert\Sigma\rvert$ integers
Last column (*L*):	m characters
SA sample:	$m \cdot a$ integers, where a is fraction of rows kept
Checkpoints:	$m \times \lvert\Sigma\rvert \cdot b$ integers, where b is fraction of rows checkpointed

Example: DNA alphabet (2 bits per nucleotide), T = human genome, $a = 1/32$, $b = 1/128$

First column (*F*):	16 bytes
Last column (*L*):	2 bits * 3 billion chars = 750 MB
SA sample:	3 billion chars * 4 bytes/char / 32 = ~ 400 MB
Checkpoints:	*3 billion* * 4 bytes/char / *128* = ~ 100 MB

Total < 1.5 GB

Computing BWT in O(n) time

- Easy O(n^2 log n)-time algorithm to compute the BWT (create and sort the BWT matrix explicitly).

- Several direct O(n)-time algorithms for BWT.
  These are space efficient. (Bowtie e.g. uses [1])

- Also can use suffix arrays or trees:

  Compute the suffix array, use correspondence between suffix array and BWT to output the BWT.

  O(n)-time and O(n)-space, but the constants are large.

[1] Kärkkäinen, Juha. "Fast BWT in small space by blockwise suffix sorting." *Theoretical Computer Science* 387.3 (2007): 249-257.

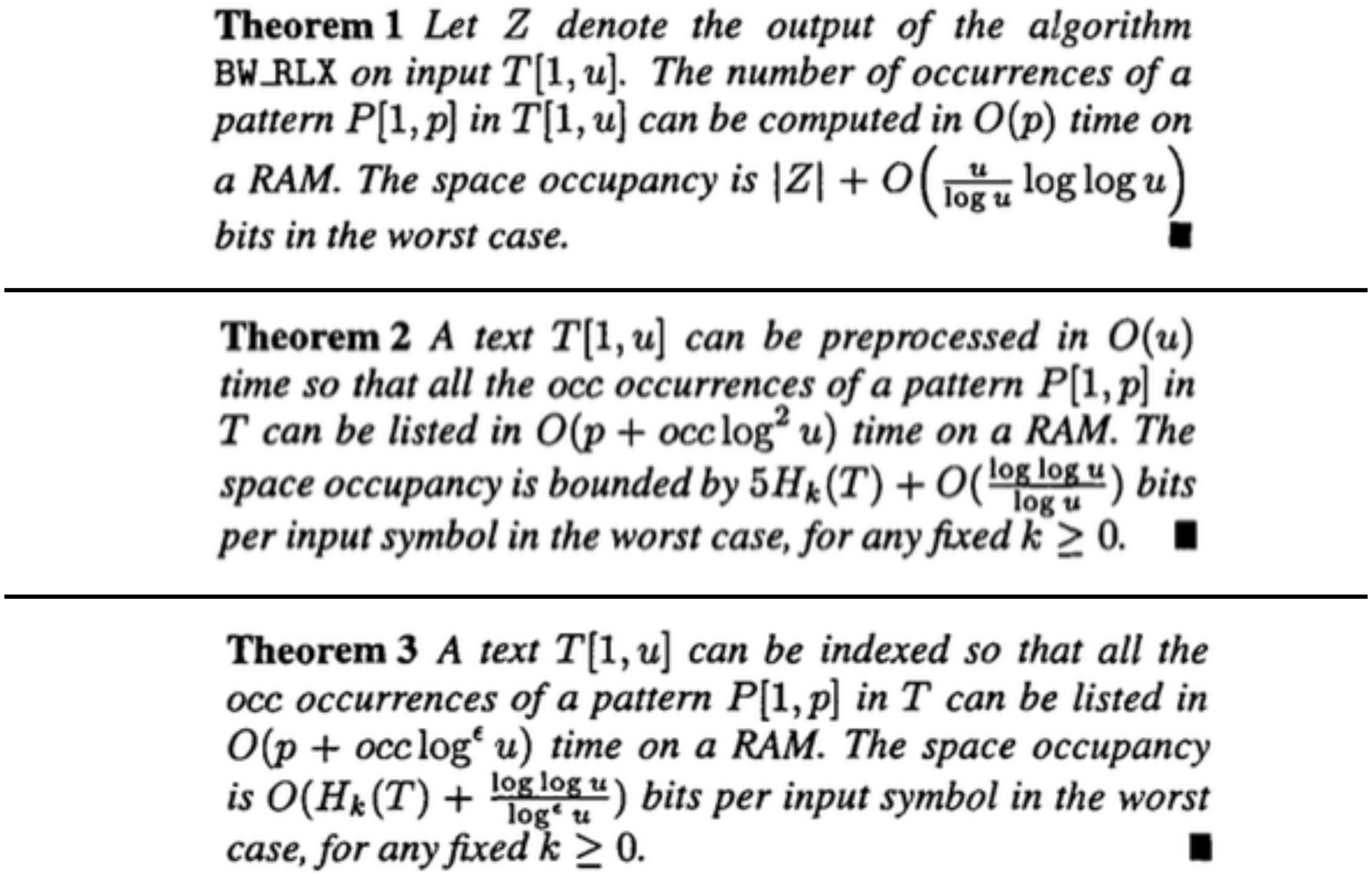
*slide courtesy of Carl Kingsford

Actual FM-Index Built on Compressed String

Ferragina, Paolo, and Giovanni Manzini. "Opportunistic data structures with applications." Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. IEEE, 2000.

Data structure has “space occupancy that is a *function of the entropy* of the underlying data set”

Stores text T[1,u] in O(H$_k$(T)) + o(1) bits for k ≥ 0 where H$_k$(T) is the kith order empirical entropy of the text — **sub-linear** for a compressible string

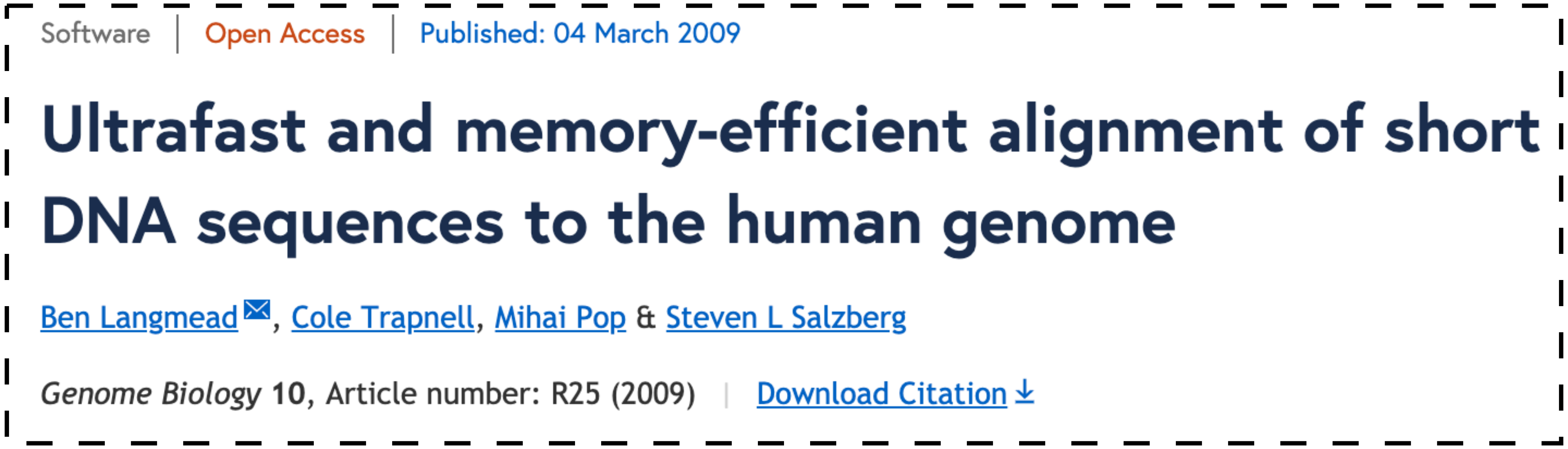


Theorem 1 *Let Z denote the output of the algorithm* BW_RLX *on input $T[1,u]$. The number of occurrences of a pattern $P[1,p]$ in $T[1,u]$ can be computed in $O(p)$ time on a RAM. The space occupancy is $|Z| + O\left(\frac{u}{\log u} \log\log u\right)$ bits in the worst case.* ■

Theorem 2 *A text $T[1,u]$ can be preprocessed in $O(u)$ time so that all the occ occurrences of a pattern $P[1,p]$ in T can be listed in $O(p + occ \log^2 u)$ time on a RAM. The space occupancy is bounded by $5H_k(T) + O(\frac{\log\log u}{\log u})$ bits per input symbol in the worst case, for any fixed $k \geq 0$.* ■

Theorem 3 *A text $T[1,u]$ can be indexed so that all the occ occurrences of a pattern $P[1,p]$ in T can be listed in $O(p + occ \log^{\epsilon} u)$ time on a RAM. The space occupancy is $O(H_k(T) + \frac{\log\log u}{\log^{\epsilon} u})$ bits per input symbol in the worst case, for any fixed $k \geq 0$.* ■

Using the FM-index in read alignment



Software | Open Access | Published: 04 March 2009

Ultrafast and memory-efficient alignment of short DNA sequences to the human genome

Ben Langmead ✉, Cole Trapnell, Mihai Pop & Steven L Salzberg

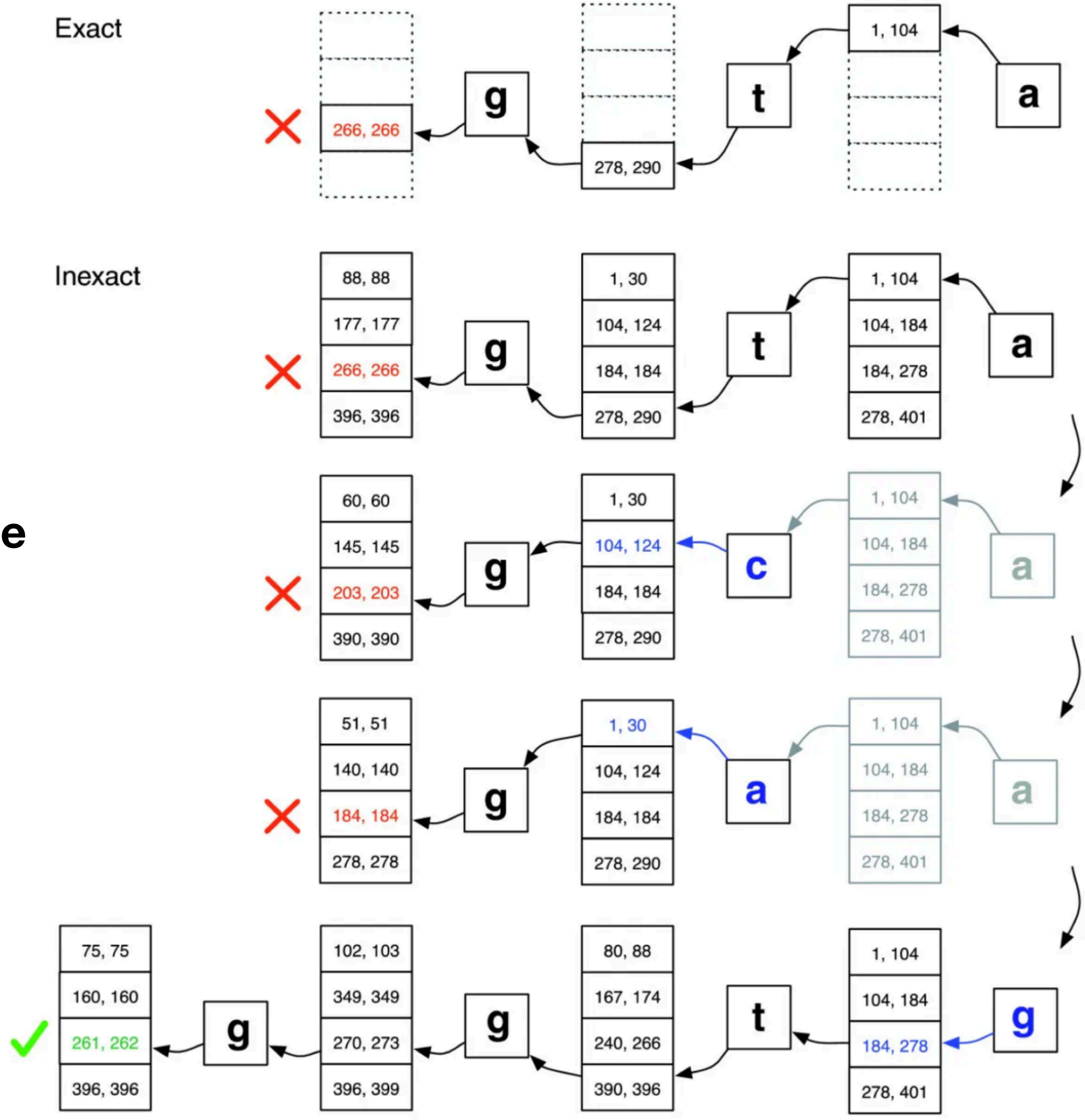
Genome Biology **10**, Article number: R25 (2009) | Download Citation ⤓

Tolerating mismatches via backtracking

Search for : GGTA

It doesn’t exist, but GGTG does.

Numbers in the boxes denote BWT intervals of search.



Finding alignments via seed & extend

Bowtie makes use of **2** FM-indices, a “forward” and “mirror” index. The forward index is over the reference and the mirror index is over the reverse (**not reverse-complement**) of the reference. This allows searching queries from left-to-right or right-to-left.

Use basic seed-and-extend paradigm

Seed is some prefix or suffix of the read of user defined length

Seed contains some maximum user-defined # of mismatches

Seeding strategy (seeds with or without mismatches)

Seed is considered (by default) the first 28bp of the read

Seed is allowed to contain up to 2 (by default) mismatches

After the seed, the subsequent portion of the read is "aligned" (allowing an arbitrary number of mismatches, but no gaps)

When up to 2 mismatches are allowed, the seed matching falls into one of 4 cases:

1. There are no mismatches in the seed
1. There are no mismatches in first 1/2 of seed, and 1 or 2 mismatches in

Seeding strategy (seeds with or without mismatches)

When up to 2 mismatches are allowed, the seed matching falls into one of 4 cases:

1. There are no mismatches in the seed

2. There are no mismatches in the first 1/2 of the seed, and 1 or 2 mismatches in the second half.

3. There are no mismatches in the second 1/2 of the seed, and 1 or 2 mismatches in the first half.

4. There is 1 mismatch in the first 1/2 and one in the second 1/2.

First case is trivial, here’s how Bowtie handles 2-4.

Seeding strategy (seeds with or without mismatches)

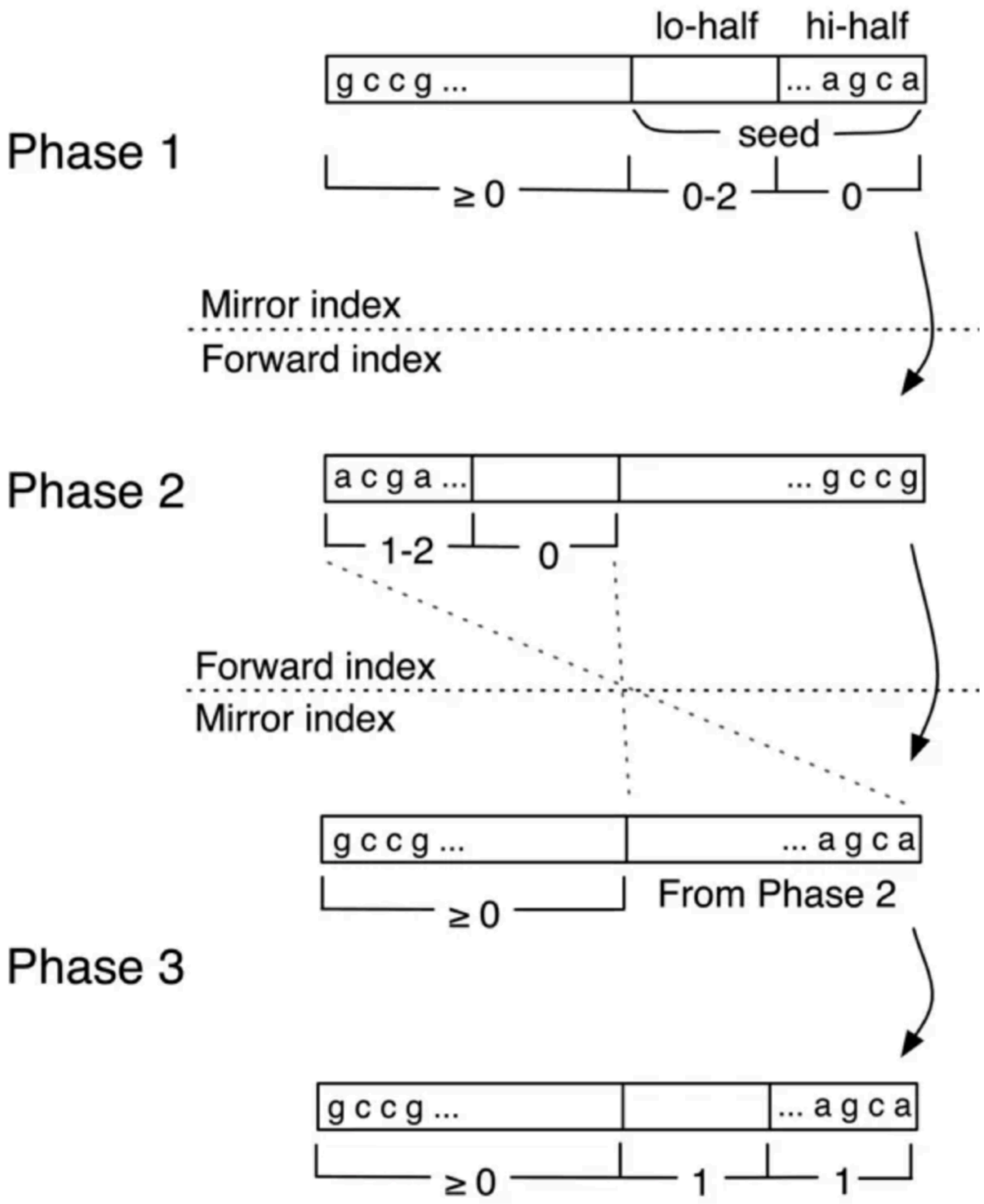
Handles case (2)

Hi-half will have exact match

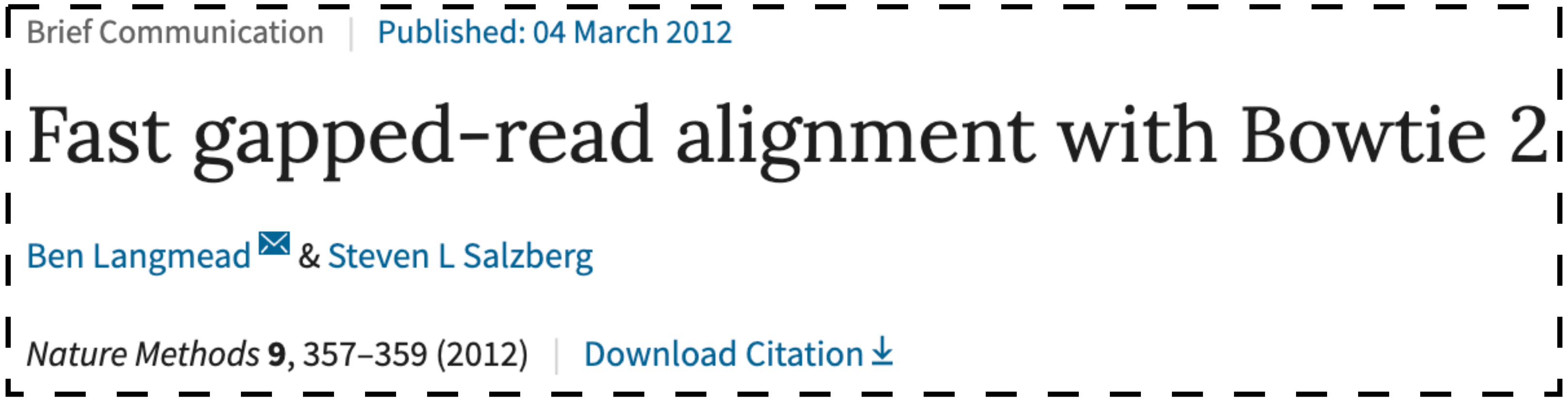
Handles case (3)

lo-half will have exact match

Handles case (4)



Bowtie2 : Building a gap-aware aligner off of Bowtie



Brief Communication | Published: 04 March 2012

Fast gapped-read alignment with Bowtie 2

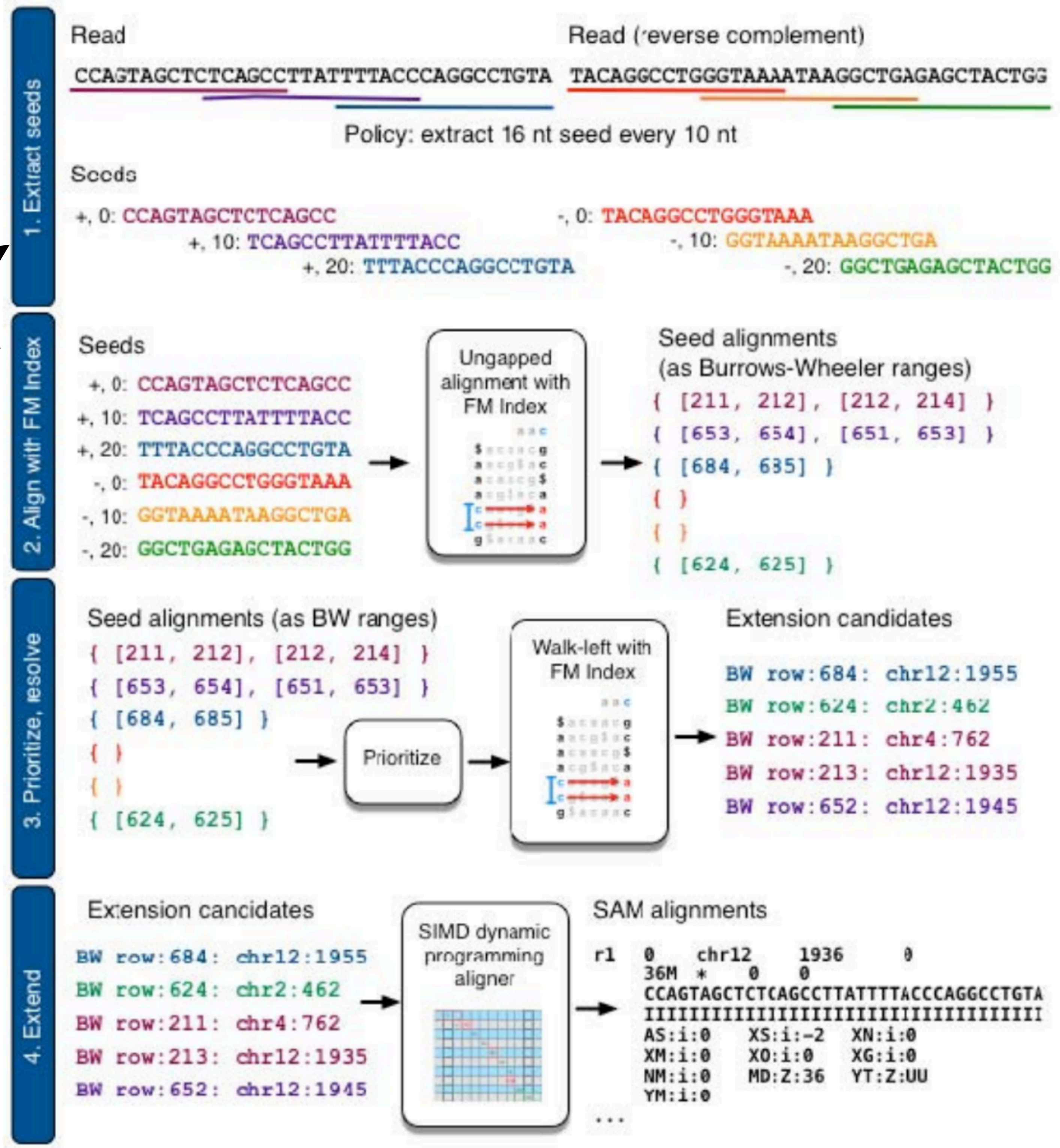
Ben Langmead ✉ & Steven L Salzberg

Nature Methods **9**, 357–359 (2012) | Download Citation ⤓

Bowtie2 proceeds in 4 phases

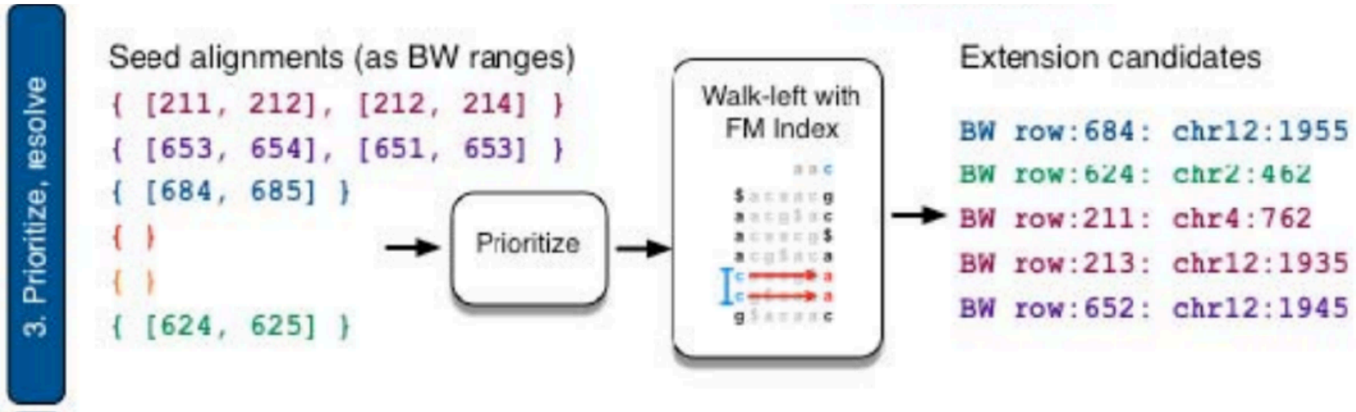
First 2 phases essentially align multiple seeds per-read using Bowtie1 (ungapped) alignment.

Supplementary Figure 1



```
r1   0    chr12    1936    0
     36M  *    0    0
     CCAGTAGCTCTCAGCCTTATTTTACCCAGGCCTGTA
     IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
     AS:i:0    XS:i:-2   XN:i:0
     XM:i:0    XO:i:0    XG:i:0
     NM:i:0    MD:Z:36   YT:Z:UU
     YM:i:0
...
```

Phase 3 (prioritizing seeds)



3. Prioritize, resolve

Seed alignments (as BW ranges)

```
{ [211, 212], [212, 214] }
{ [653, 654], [651, 653] }
{ [684, 685] }
{ }
{ }
{ [624, 625] }
```

Prioritize

Walk-left with FM Index

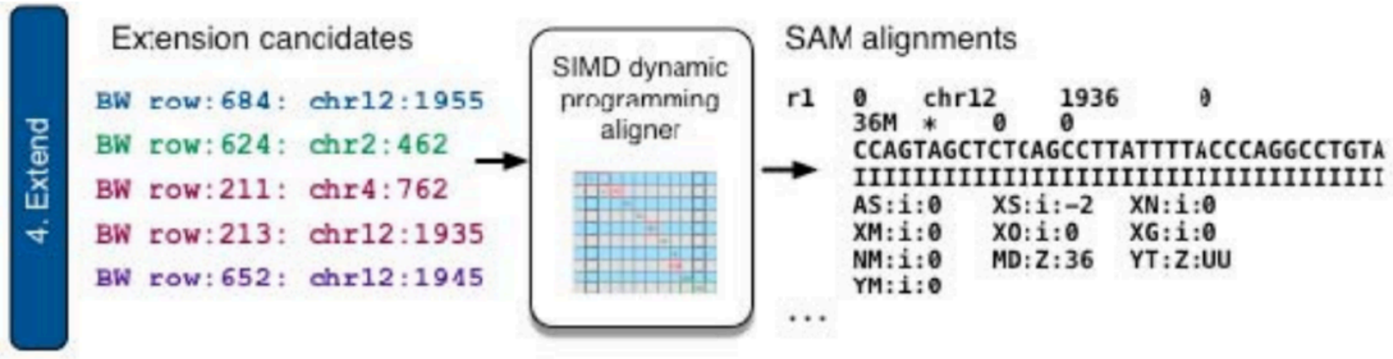
Extension candidates

```
BW row:684: chr12:1955
BW row:624: chr2:462
BW row:211: chr4:762
BW row:213: chr12:1935
BW row:652: chr12:1945
```

Each seed (individual BWT row) is “scored” based on the width of its range. A seed, x, occurring in a range of width r is assigned a weight of w(x) = $1/r^2$.

Then, the seeds are selected at random, according to these weights, and an alignment extension is attempted around each seed.

Phase 4 (aligning around seeds)



```
BW row:684: chr12:1955
BW row:624: chr2:462
BW row:211: chr4:762
BW row:213: chr12:1935
BW row:652: chr12:1945
```

```
r1   0    chr12    1936     0
     36M  *     0    0
     CCAGTAGCTCTCAGCCTTATTTTACCCAGGCCTGTA
     IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
     AS:i:0    XS:i:-2   XN:i:0
     XM:i:0    XO:i:0    XG:i:0
     NM:i:0    MD:Z:36   YT:Z:UU
     YM:i:0
...
```

There are many important enhancements to the "basic" DP, which are used in BT2 and other aligners. Some relevant ones are:

Alignments are computed in "bands" around the diagonal to avoid filling out irrelevant parts of the alignment matrix.

Wide instruction set operations are used to fill in multiple cells simultaneously.

Complex scoring functions are used that enable e.g. incorporating quality values.

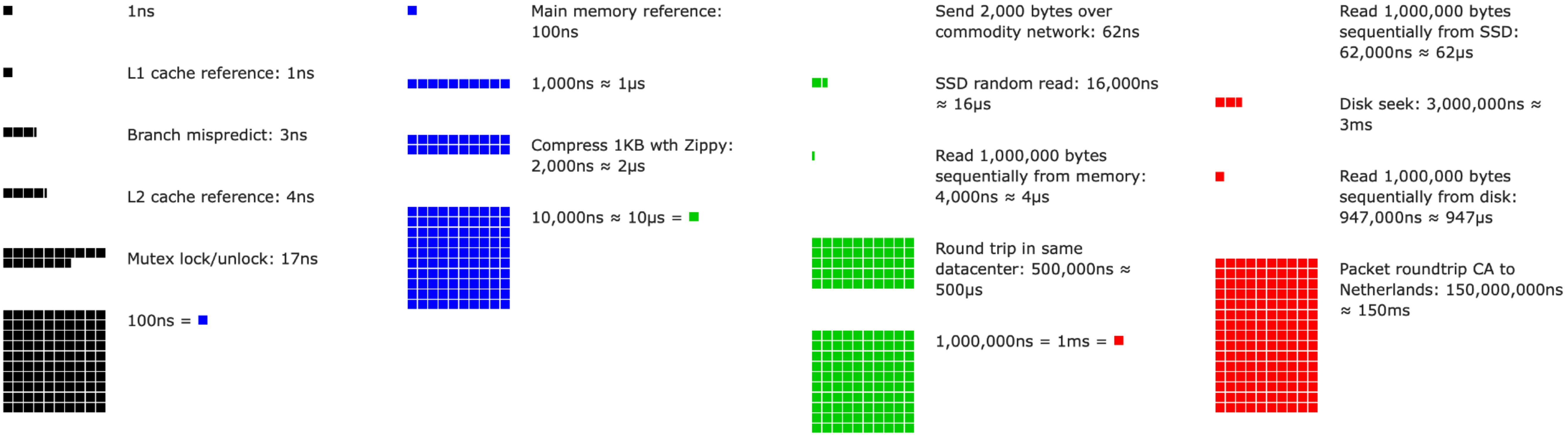
Hierarchical FM-index

Introduced by Kim, Langmead & Salzberg ('15)

Observation: Despite its asymptotic efficiency, search in the FM-index can be slow, in part, because the patterns of memory access are very incoherent (think about the search procedure).

Idea: Instead of a single *global* FM-index, build a global FM-index and a series of *local* FM-indices, where each local index is small enough to fit in CPU cache … recall the cache speed advantage.

Cache is $$



1ns

L1 cache reference: 1ns

Branch mispredict: 3ns

L2 cache reference: 4ns

Mutex lock/unlock: 17ns

100ns = 🟦

Main memory reference: 100ns

1,000ns ≈ 1µs

Compress 1KB wth Zippy: 2,000ns ≈ 2µs

10,000ns ≈ 10µs = 🟩

Send 2,000 bytes over commodity network: 62ns

SSD random read: 16,000ns ≈ 16µs

Read 1,000,000 bytes sequentially from memory: 4,000ns ≈ 4µs

Round trip in same datacenter: 500,000ns ≈ 500µs

1,000,000ns = 1ms = 🟥

Read 1,000,000 bytes sequentially from SSD: 62,000ns ≈ 62µs

Disk seek: 3,000,000ns ≈ 3ms

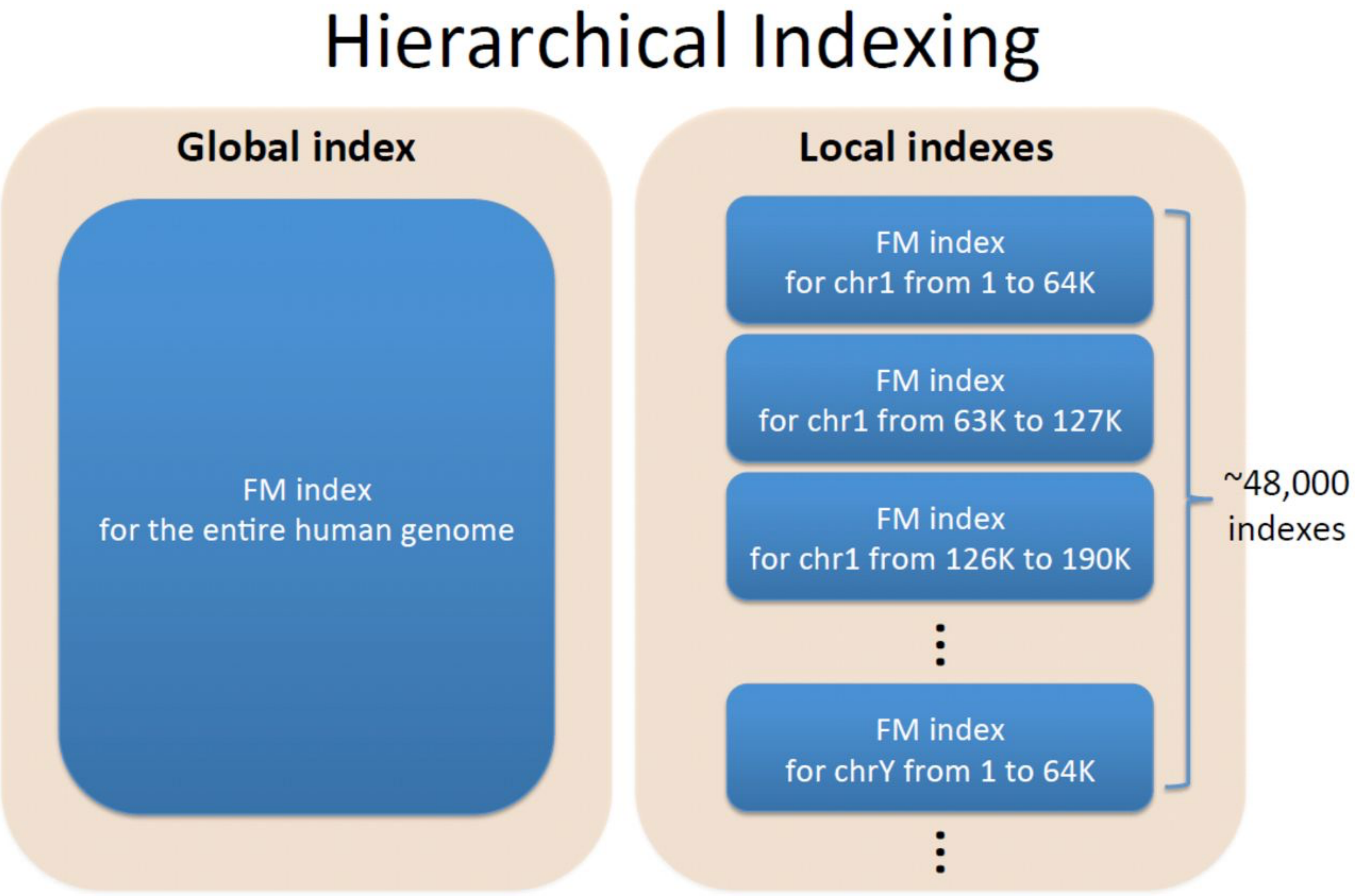
Read 1,000,000 bytes sequentially from disk: 947,000ns ≈ 947µs

Packet roundtrip CA to Netherlands: 150,000,000ns ≈ 150ms

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Search in the Hierarchical FM-index

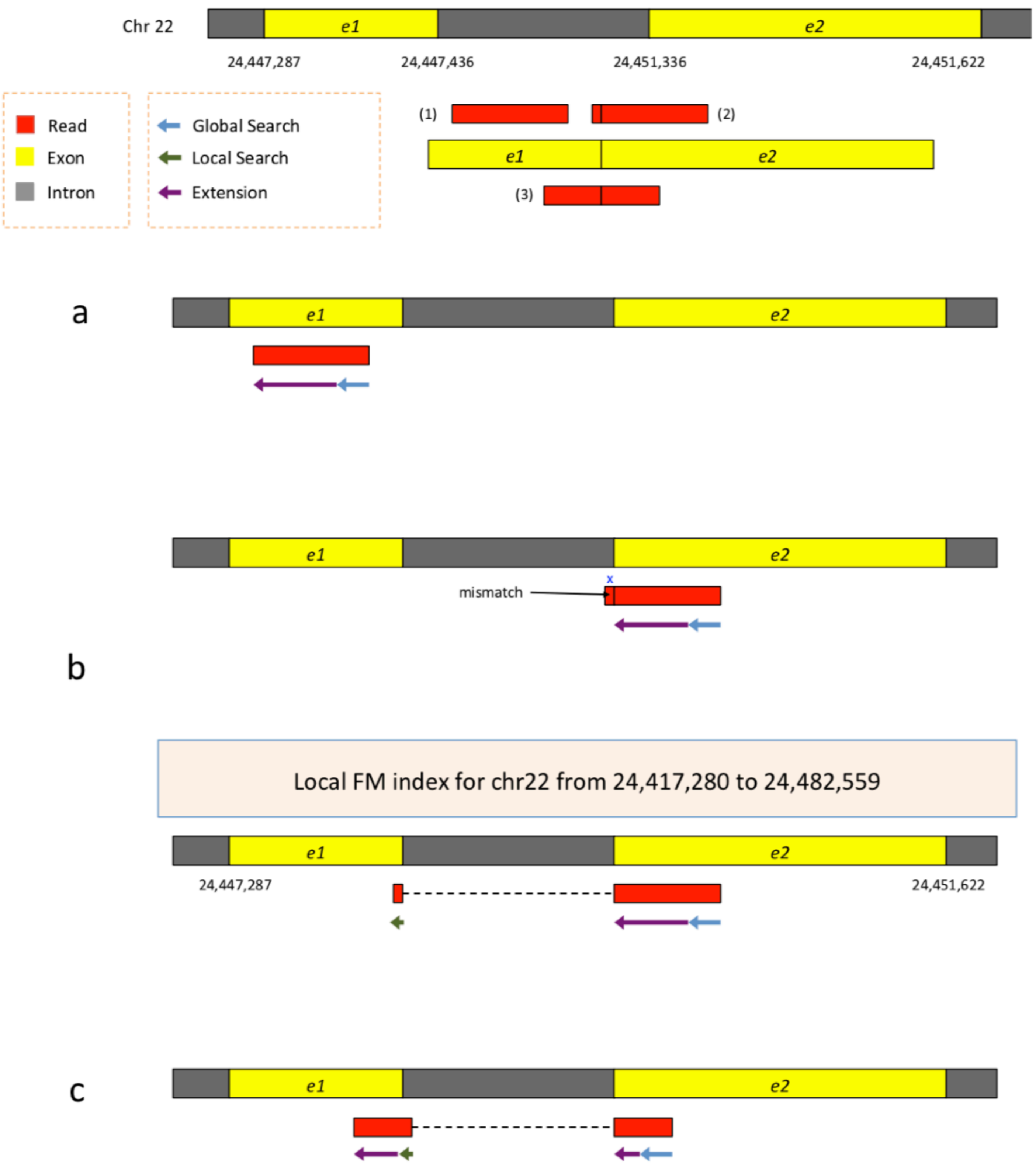
Idea: Start search in the global FM-index, where it is *slow*. Once a sufficiently-long suffix has been found to restrict the pattern to one or a few local indices, continue extending the pattern in the local index, where it is *fast.*



Using a hierarchical FM-index for search

Global search of short suffix gives 1 or more potential positions

Local index is used to "extend" the matches within this region, as well as to perform local search for upstream exons with the same local index.

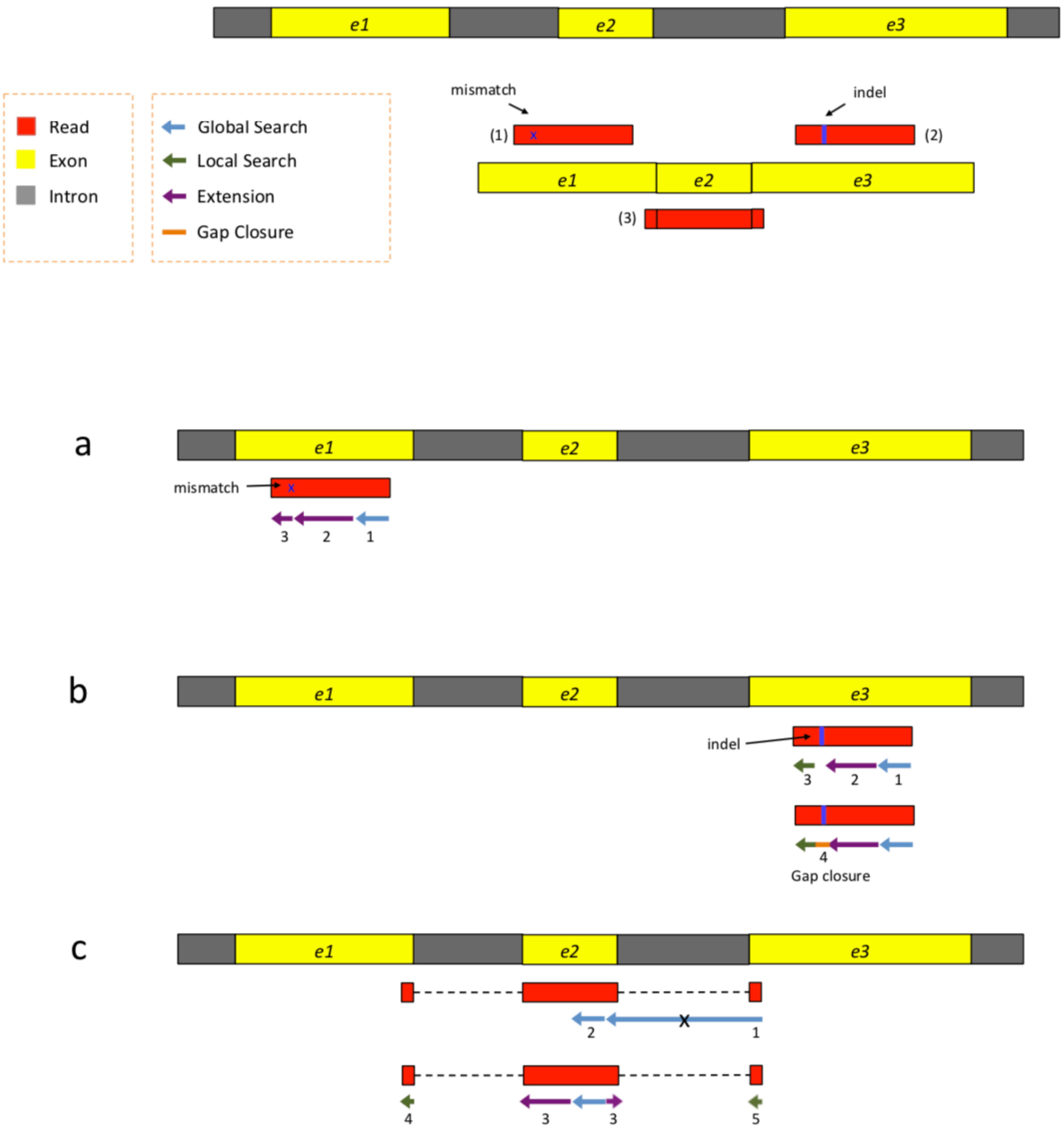


Mapping in the presence of substitutions & indwells

Extension has a specialized case for single nucleotide substitutions (keep extending and look for subsequent matches)

If more than 1 base mismatches, do search again in the local index

Multiple local alignments stitched together with “gap closure” procedure



e1
e2
e3
Read
Exon
Intron
Global Search
Local Search
Extension
Gap Closure
mismatch
indel
(1)
(2)
(3)
a
mismatch
3
2
1
b
indel
3
2
1
4
Gap closure
c
2
1
4
3
3
5