

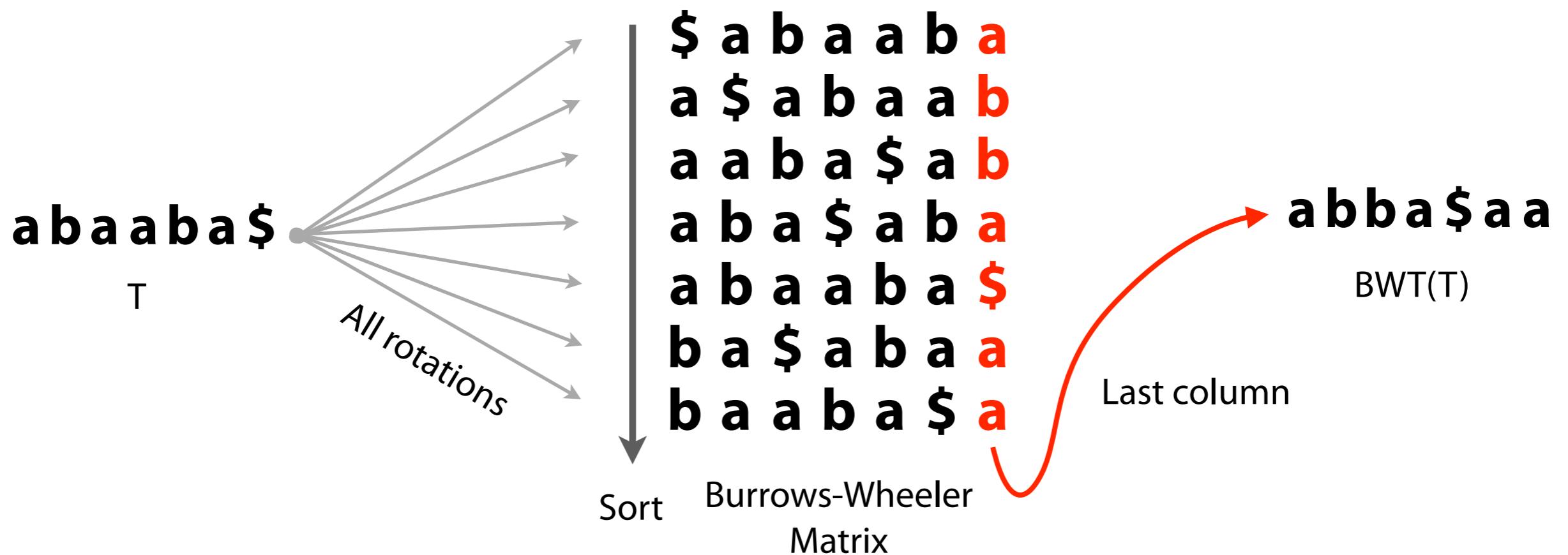
# The BWT and FM-index

**NOTE: This lecture is  
being recorded**

Slides marked with \* adopted from Ben Langmead

# Burrows-Wheeler Transform

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



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

Make list of all rotations  
Sort them  
Take last column

```
>>> bwtViaBwm("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnooooattTmmmrccccrooo__ooo'  
  
>>> bwtViaBwm("It_was_the_best_of_times_it_was_the_worst_of_times$")
's$esttssfftteww_hhmmboottt_ii__woeeaaressIi_____'  
  
>>> bwtViaBwm('in_the_jingle_jangle_morning_Ill_come_following_you$')
'u_gleeeengj_mlhl_nnnnt$nwj__lggiolo_iiafcmylo_o_'
```

# Burrows-Wheeler Transform

BWM bears a resemblance to 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( $T$ )

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

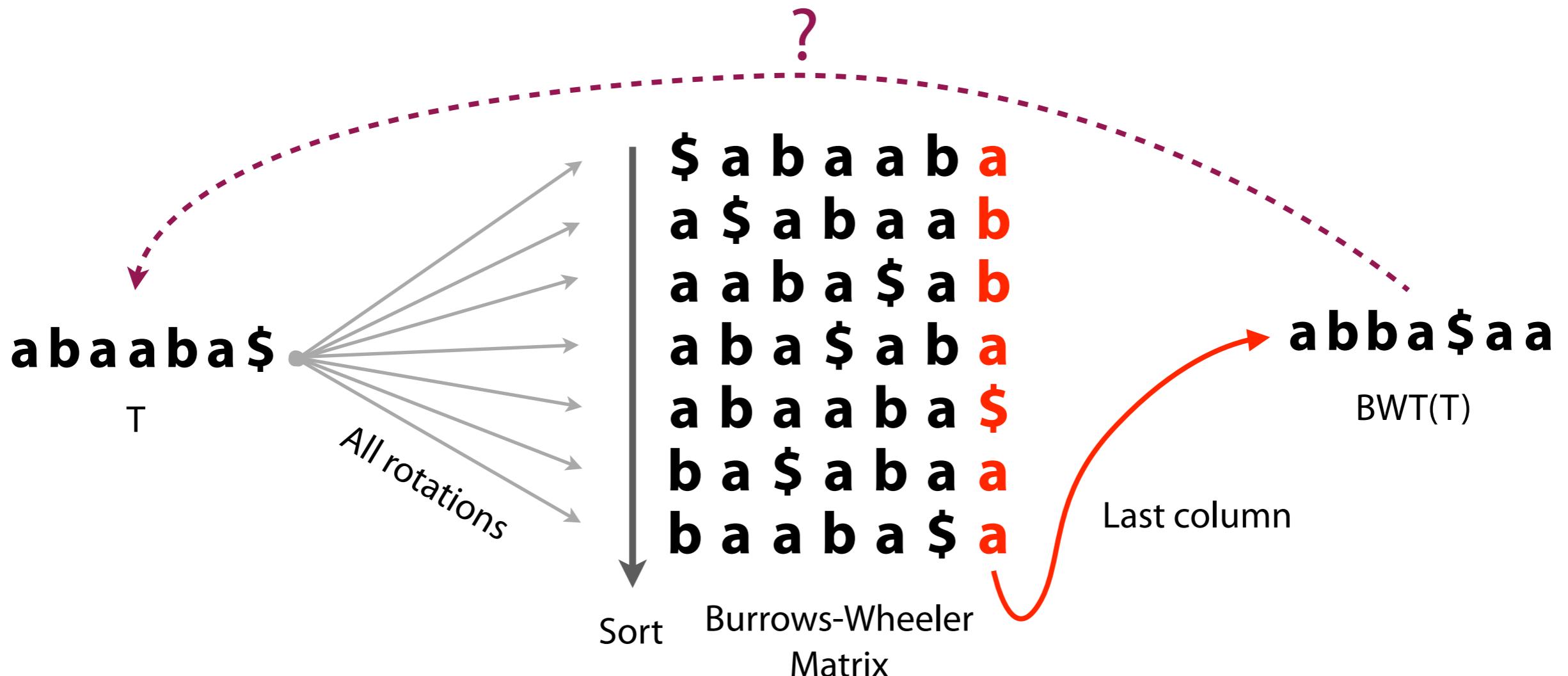
BWM( $T$ )

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?



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*.

**a<sub>0</sub> b<sub>0</sub> a<sub>1</sub> a<sub>2</sub> b<sub>1</sub> a<sub>3</sub> \$**

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

	<i>F</i>	<i>L</i>
BWM with T-ranking:	\$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> <b>a<sub>3</sub></b>	
	<b>a<sub>3</sub></b> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub>	
	<b>a<sub>1</sub></b> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub>	
	<b>a<sub>2</sub></b> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> <b>a<sub>1</sub></b>	
	<b>a<sub>0</sub></b> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$	
	b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> <b>a<sub>2</sub></b>	
	b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ <b>a<sub>0</sub></b>	

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

And look at just the **a**s

**a**s occur in the same order in *F* and *L*. As we look down columns, in both cases we see: **a<sub>3</sub>, a<sub>1</sub>, a<sub>2</sub>, a<sub>0</sub>**

# Burrows-Wheeler Transform

*F*

BWM with T-ranking:

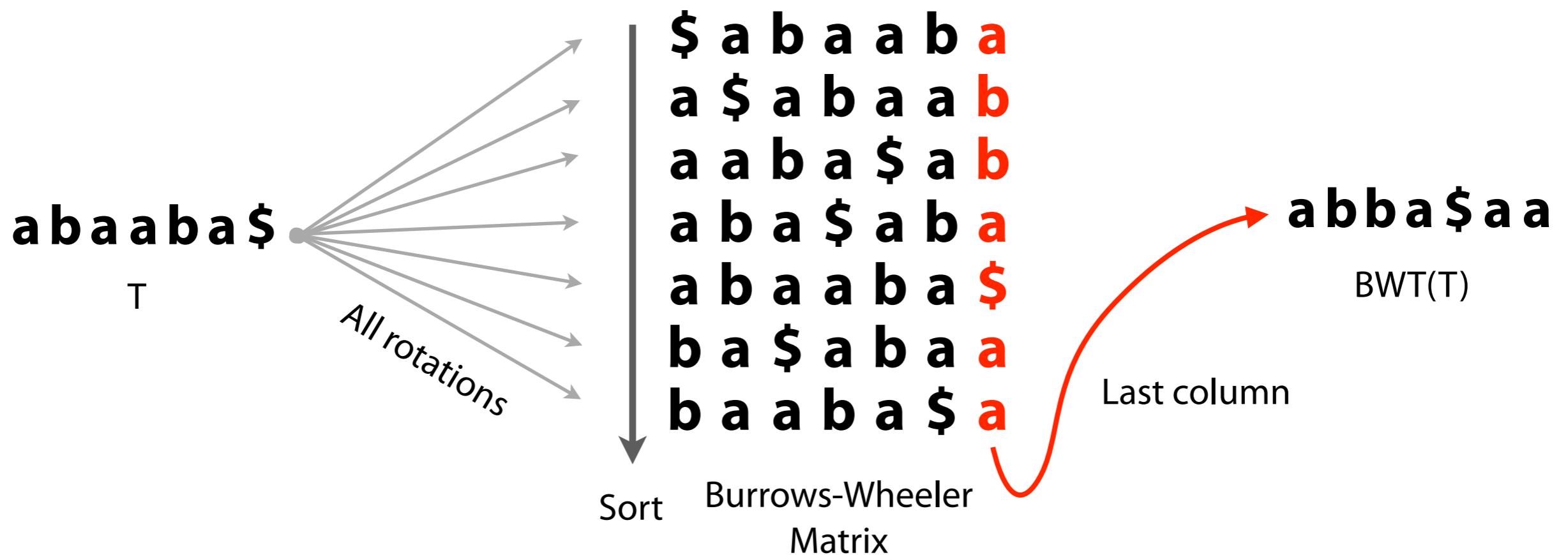
\$ a<sub>0</sub> b<sub>0</sub> a<sub>1</sub> a<sub>2</sub> b<sub>1</sub> a<sub>3</sub>  
a<sub>3</sub> \$ a<sub>0</sub> b<sub>0</sub> a<sub>1</sub> a<sub>2</sub> **b<sub>1</sub>**  
a<sub>1</sub> a<sub>2</sub> b<sub>1</sub> a<sub>3</sub> \$ a<sub>0</sub> **b<sub>0</sub>**  
a<sub>2</sub> b<sub>1</sub> a<sub>3</sub> \$ a<sub>0</sub> b<sub>0</sub> a<sub>1</sub>  
a<sub>0</sub> b<sub>0</sub> a<sub>1</sub> a<sub>2</sub> b<sub>1</sub> a<sub>3</sub> \$  
**b<sub>1</sub>** a<sub>3</sub> \$ a<sub>0</sub> b<sub>0</sub> a<sub>1</sub> a<sub>2</sub>  
**b<sub>0</sub>** a<sub>1</sub> a<sub>2</sub> b<sub>1</sub> a<sub>3</sub> \$ a<sub>0</sub>

*L*

Same with **b**s: **b<sub>1</sub>, b<sub>0</sub>**

# Burrows-Wheeler Transform

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



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

	<i>F</i>	<i>L</i>
BWM with T-ranking:	\$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub>	
	a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub>	
	a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub>	
	a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub>	
	a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$	
	b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub>	
	b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub>	

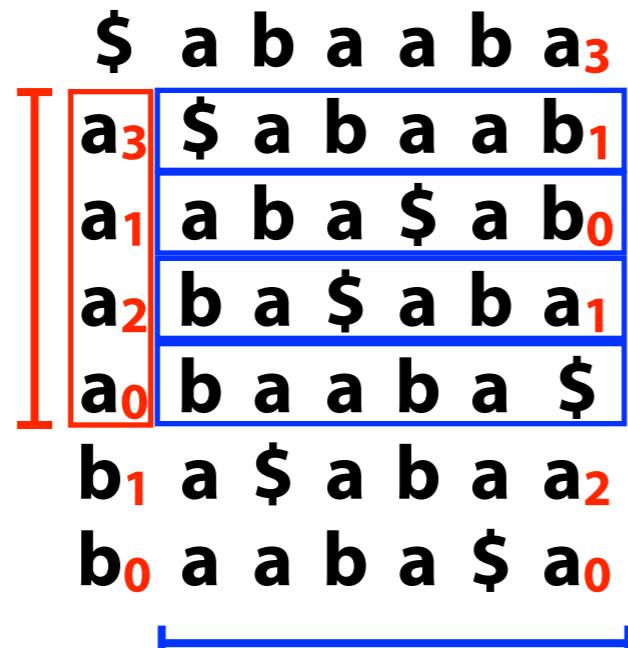
LF Mapping: The  $i^{\text{th}}$  occurrence of a character  $c$  in  $L$  and the  $i^{\text{th}}$  occurrence of  $c$  in  $F$  correspond to the *same* occurrence in  $T$

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

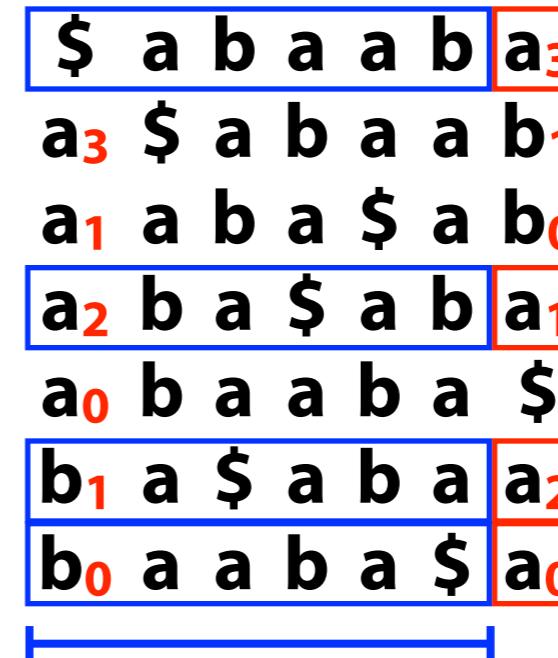
# Burrows-Wheeler Transform: LF Mapping

Why does the LF Mapping hold?

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



They're sorted by  
right-context



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

They're sorted by  
right-context

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

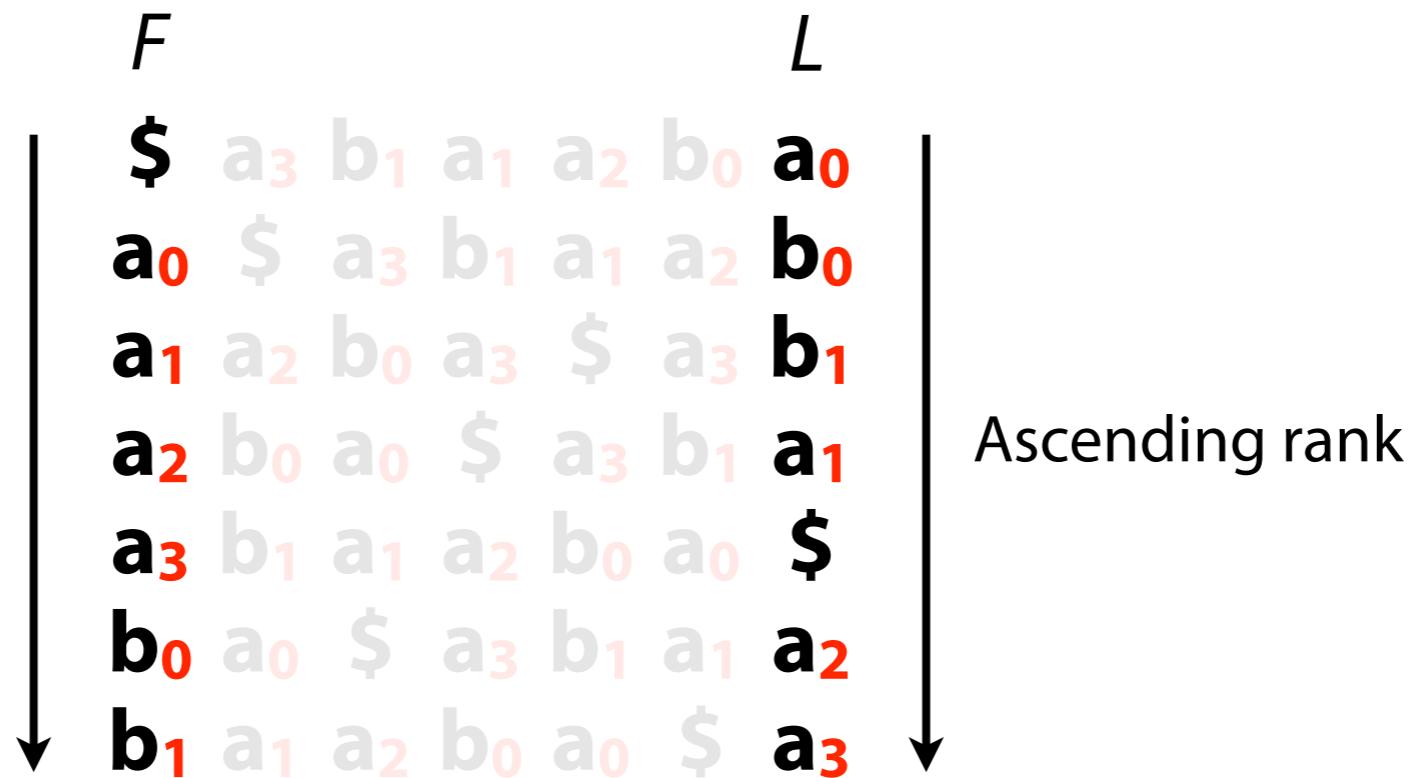
BWM with T-ranking:

<i>F</i>	<i>L</i>
\$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub>	
a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub>	
a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub>	
a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub>	
a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$	
b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub> b <sub>0</sub> a <sub>1</sub> a <sub>2</sub>	
b <sub>0</sub> a <sub>1</sub> a <sub>2</sub> b <sub>1</sub> a <sub>3</sub> \$ a <sub>0</sub>	

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:



$F$  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

<i>F</i>	<i>L</i>	
\$	a <sub>0</sub>	
a <sub>0</sub>	b <sub>0</sub>	
a <sub>1</sub>	b <sub>1</sub> ←	Which BWM row <i>begins</i> with b <sub>1</sub> ?
a <sub>2</sub>	a <sub>1</sub>	Skip row starting with \$ (1 row)
a <sub>3</sub>	\$	Skip rows starting with a (4 rows)
b <sub>0</sub>	a <sub>2</sub>	Skip row starting with b <sub>0</sub> (1 row)
row 6 →	b <sub>1</sub>	Answer: row 6
	a <sub>3</sub>	

# 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 **G100**? (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  $\$$ :  $a_0$

$a_0$ : LF Mapping says this is same occurrence of **a** as first **a** in  $F$ . Jump to row *beginning* with  $a_0$ .  $L$  contains character just **prior** to  $a_0$ :  $b_0$ .

Repeat for  $b_0$ , get  $a_2$

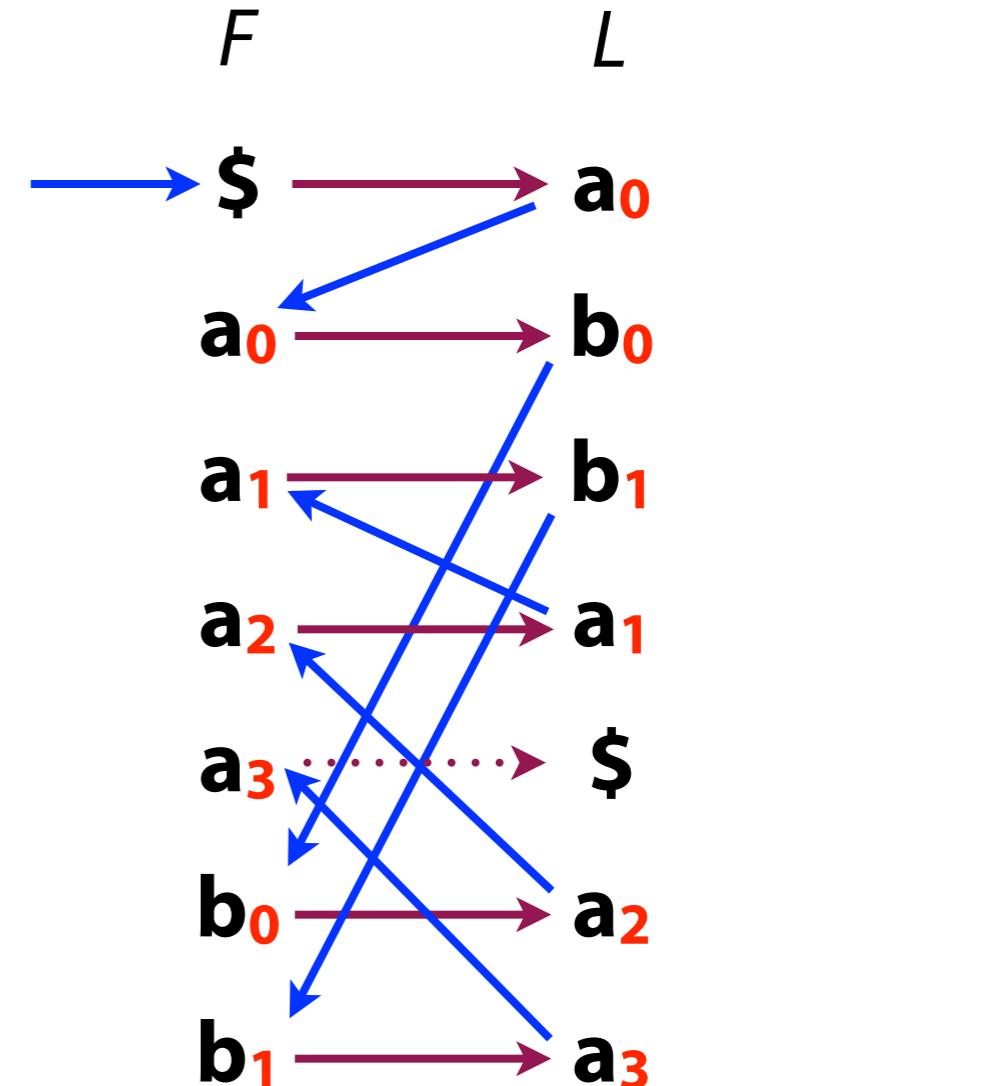
Repeat for  $a_2$ , get  $a_1$

Repeat for  $a_1$ , get  $b_1$

Repeat for  $b_1$ , get  $a_3$

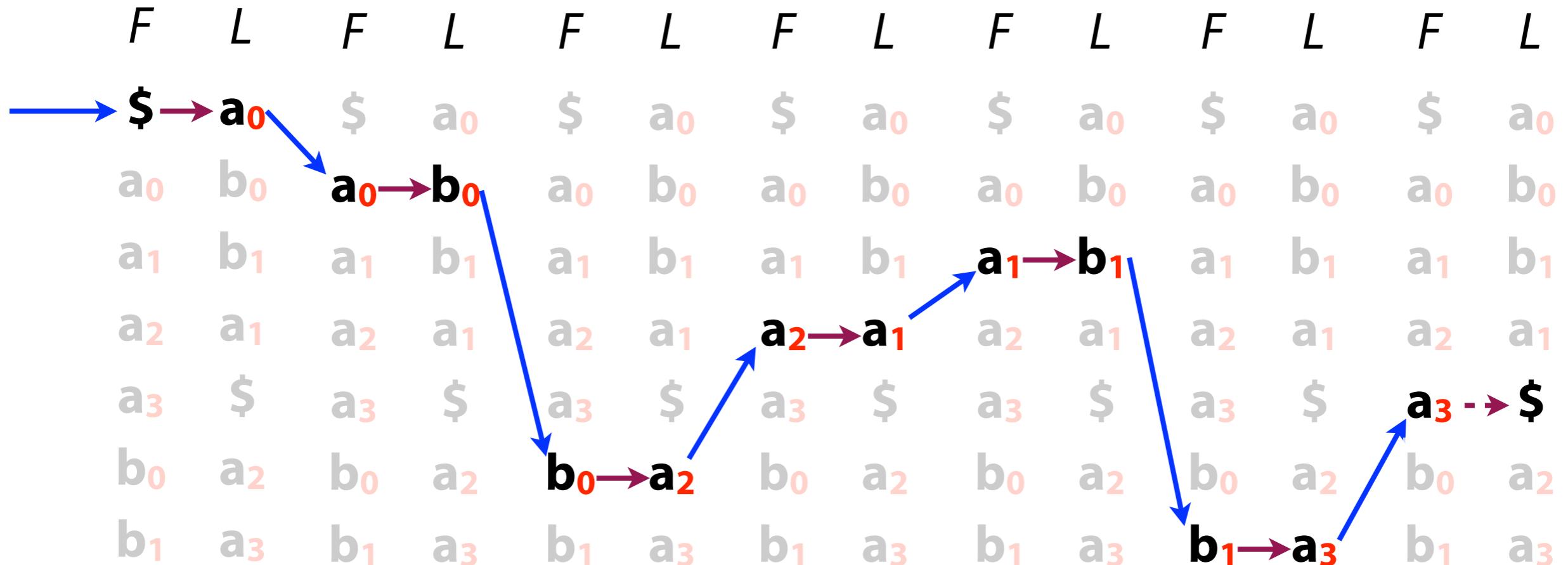
Repeat for  $a_3$ , get  $\$$ , done

Reverse of chars we visited =  $a_3 b_1 a_1 a_2 b_0 a_0 \$ = T$



# Burrows-Wheeler Transform: reversing

Another way to visualize reversing BWT( $T$ ):



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

# Burrows-Wheeler Transform: reversing

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

>>> reverseBwt("s$esttssfftteww_hhmmbootttt_i_ _woeeaaressIi_____")
'It_was_the_best_of_times_it_was_the_worst_of_times$'

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

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!

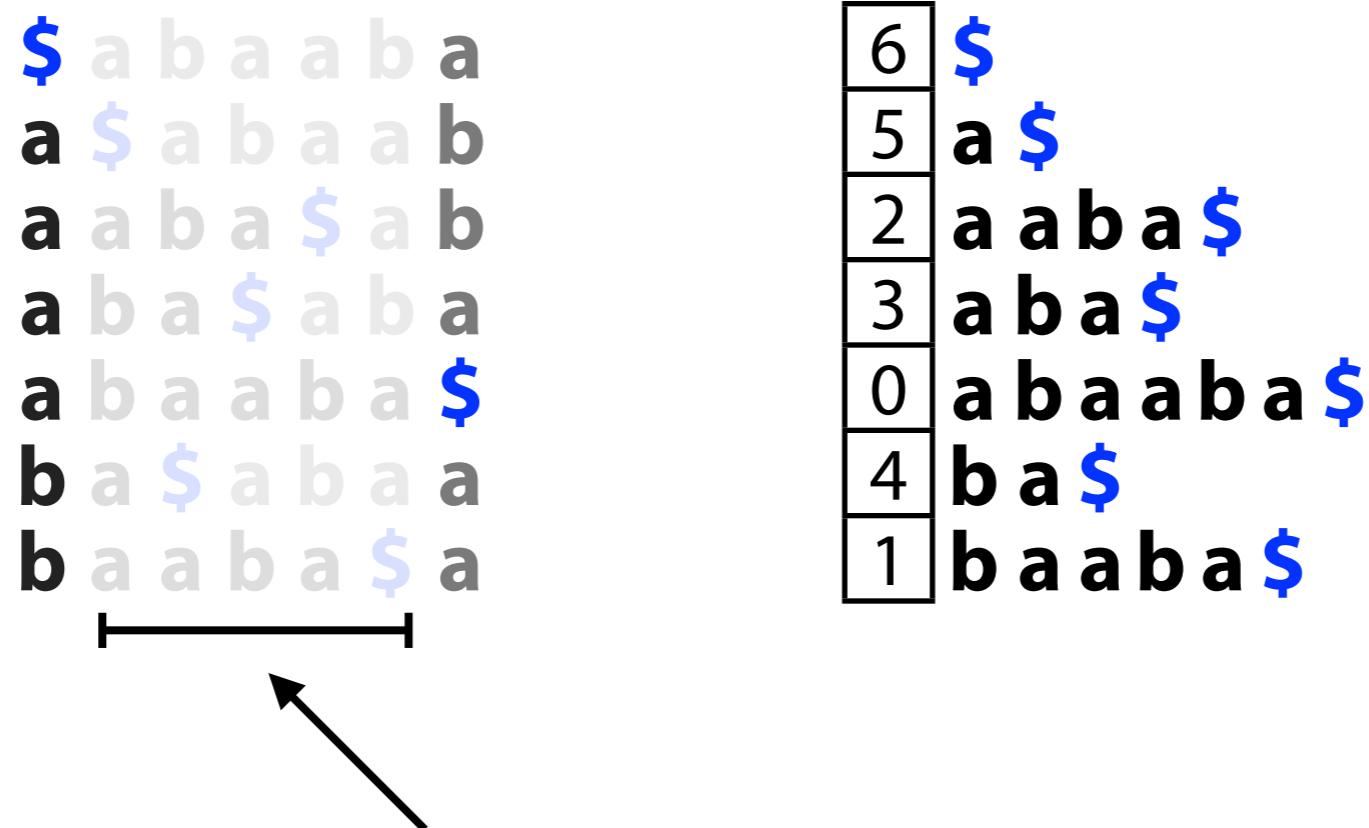
$F$	$L$
\$	a b a a b a
a	\$ a b a a b
a	a b a \$ a b
a	b a \$ a b a
a	a b a a b a \$
b	a \$ a b a a
b	b a a b a \$ a

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



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  $P$

$$P = \mathbf{aba}$$

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

$F$	$L$
\$	a b a a b a <sub>3</sub>
a <sub>0</sub>	\$ a b a a b <sub>1</sub>
a <sub>1</sub>	a b a \$ a b <sub>0</sub>
a <sub>2</sub>	b a \$ a b a <sub>1</sub>
a <sub>3</sub>	b a a b a \$
b <sub>0</sub>	a \$ a b a a <sub>2</sub>
b <sub>1</sub>	a a b a \$ a <sub>0</sub>

# FM Index: querying

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

$$P = \mathbf{aba}$$

$F$	$L$
\$	a b a a b <b>a<sub>0</sub></b>
<b>a<sub>0</sub></b>	\$ a b a a <b>b<sub>0</sub></b>
<b>a<sub>1</sub></b>	a b a \$ a <b>b<sub>1</sub></b>
<b>a<sub>2</sub></b>	b a \$ a b <b>a<sub>1</sub></b>
<b>a<sub>3</sub></b>	b a a b a \$
<b>b<sub>0</sub></b>	a \$ a b a <b>a<sub>2</sub></b>
<b>b<sub>1</sub></b>	a a b a \$ <b>a<sub>3</sub></b>

Look at those rows in  $L$ .

**b<sub>0</sub>, b<sub>1</sub>** are **b**s occurring just to left.

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

$$P = \mathbf{aba}$$

$F$	$L$
\$	a b a a b <b>a<sub>0</sub></b>
<b>a<sub>0</sub></b>	\$ a b a a <b>b<sub>0</sub></b>
<b>a<sub>1</sub></b>	a b a \$ a <b>b<sub>1</sub></b>
<b>a<sub>2</sub></b>	b a \$ a b <b>a<sub>1</sub></b>
<b>a<sub>3</sub></b>	b a a b a \$
<b>b<sub>0</sub></b>	<b>b<sub>0</sub></b> a \$ a b a <b>a<sub>2</sub></b>
<b>b<sub>1</sub></b>	<b>b<sub>1</sub></b> a a b a \$ <b>a<sub>3</sub></b>

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

# FM Index: querying

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

$$P = \mathbf{aba}$$

$F$	$L$
\$	a b a a b <b>a<sub>0</sub></b>
<b>a<sub>0</sub></b>	\$ a b a a <b>b<sub>0</sub></b>
<b>a<sub>1</sub></b>	a b a \$ a <b>b<sub>1</sub></b>
<b>a<sub>2</sub></b>	b a \$ a b <b>a<sub>1</sub></b>
<b>a<sub>3</sub></b>	b a a b a \$
<b>b<sub>0</sub></b>	a \$ a b a <b>a<sub>2</sub></b>
<b>b<sub>1</sub></b>	a a b a \$ <b>a<sub>3</sub></b>

$$P = \mathbf{aba}$$

$F$	$L$
\$	a b a a b <b>a<sub>0</sub></b>
<b>a<sub>0</sub></b>	\$ a b a a <b>b<sub>0</sub></b>
<b>a<sub>1</sub></b>	a b a \$ a <b>b<sub>1</sub></b>
<b>a<sub>2</sub></b>	b a \$ a b <b>a<sub>1</sub></b>
<b>a<sub>3</sub></b>	b a a b a \$
<b>b<sub>0</sub></b>	a \$ a b a <b>a<sub>2</sub></b>
<b>b<sub>1</sub></b>	a a b a \$ <b>a<sub>3</sub></b>

Use LF Mapping



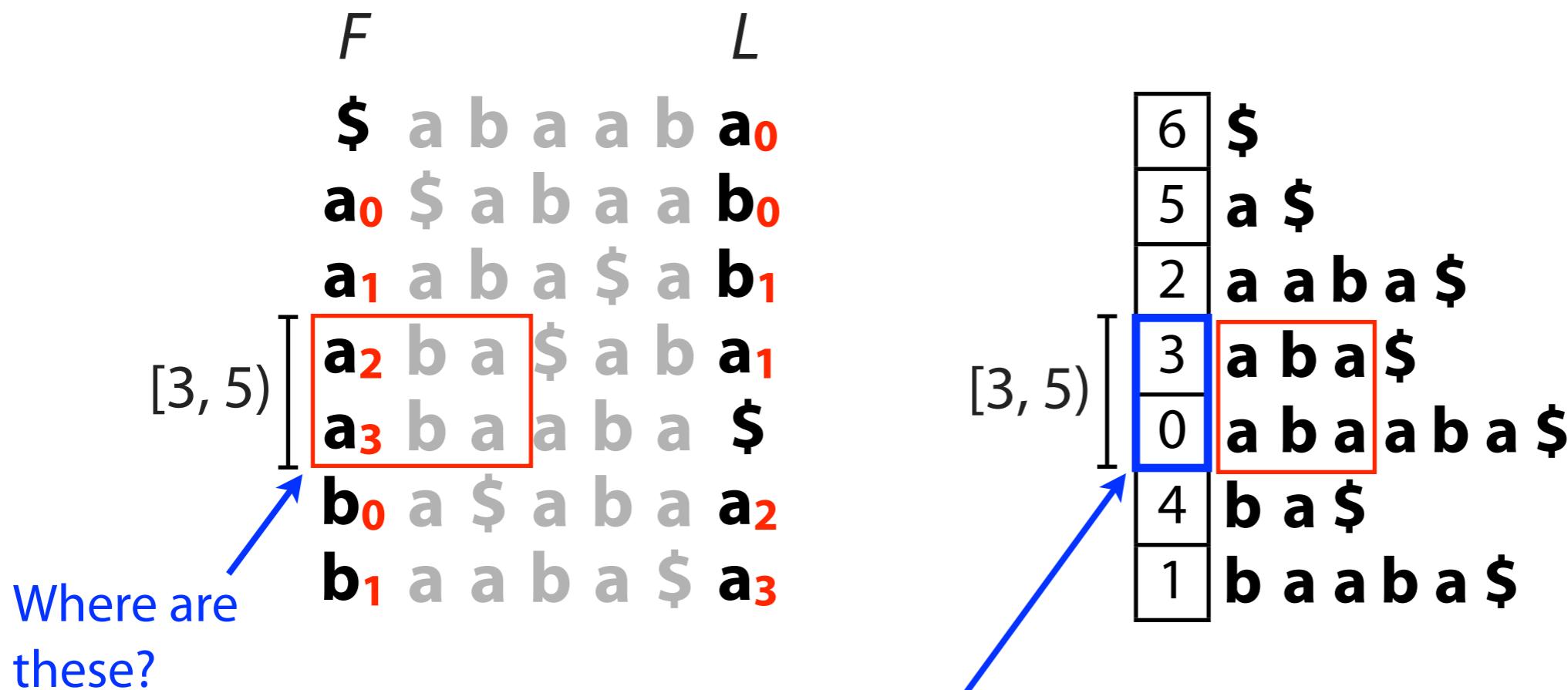
$\leftarrow a_2, a_3$  occur just to left.

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

# FM Index: querying

$P = \text{aba}$

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



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  $L$ :

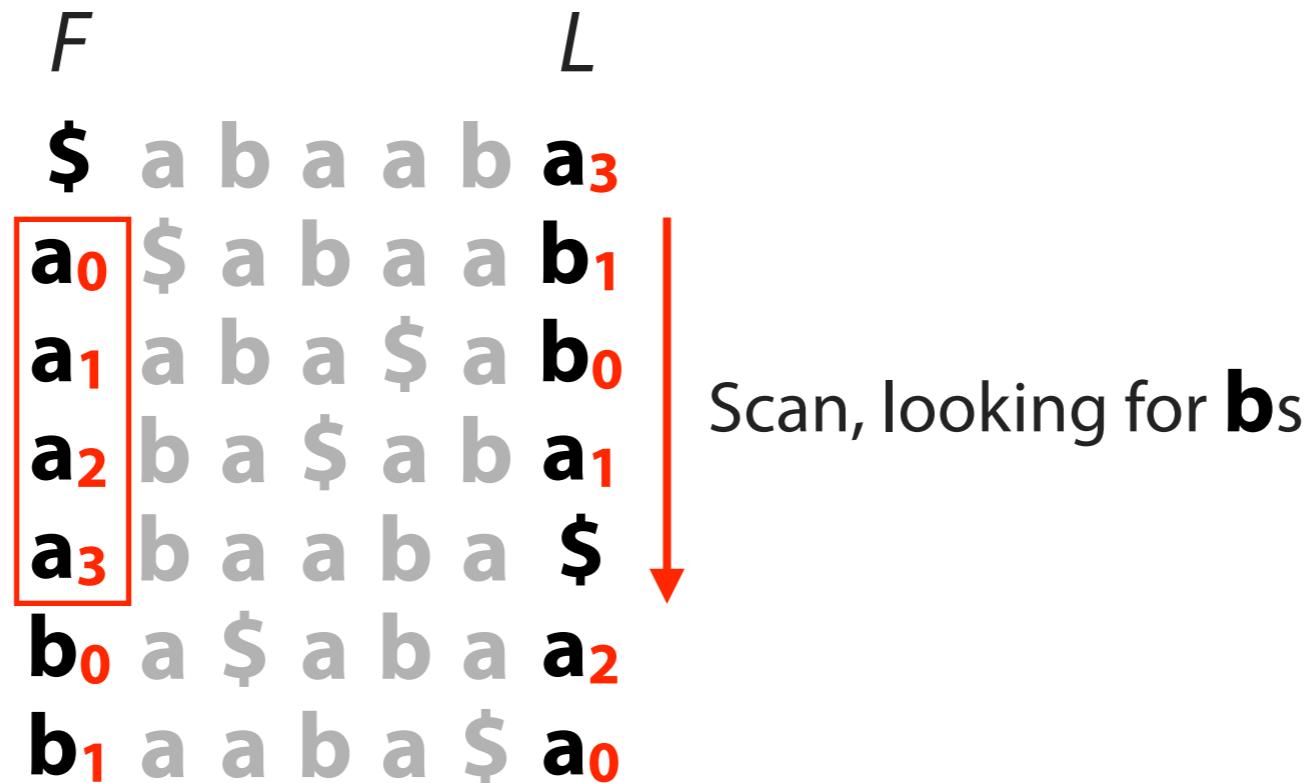
$$P = \mathbf{bba}$$

$F$	$L$
\$ a b a a b	a <sub>0</sub>
a <sub>0</sub> \$ a b a a	b <sub>0</sub>
a <sub>1</sub> a b a \$ a	b <sub>1</sub>
a <sub>2</sub> b a \$ a b	a <sub>1</sub>
a <sub>3</sub> b a a b a	\$
Rows with <b>ba</b> prefix	I [b <sub>0</sub> a \$ a b a a <sub>2</sub> ] ← No bs!
	[b <sub>1</sub> a a b a \$ a <sub>3</sub> ]

# FM Index: querying

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

$$P = \mathbf{aba}$$



# FM Index: lingering issues

- (1) Scanning for preceding character is slow

\$ a b a a b a<sub>0</sub>  
a<sub>0</sub> \$ a b a a b<sub>0</sub>  
a<sub>1</sub> a b a \$ a b<sub>1</sub>  
a<sub>2</sub> b a \$ a b a<sub>1</sub>  
a<sub>3</sub> b a a b a \$  
b<sub>0</sub> a \$ a b a a<sub>2</sub>  
b<sub>1</sub> a a b a \$ a<sub>3</sub>

m integers

O(m) scan

m

- (2) Storing ranks takes too much space

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<sub>0</sub>  
a<sub>0</sub> \$ a b a a b<sub>0</sub>  
a<sub>1</sub> a b a \$ a b<sub>1</sub>  
Where? a<sub>2</sub> b a \$ a b a<sub>1</sub>  
a<sub>3</sub> b a a b a \$  
b<sub>0</sub> a \$ a b a a<sub>2</sub>  
b<sub>1</sub> a a b a \$ a<sub>3</sub>

# 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 <b>a<sub>0</sub></b>
<b>a<sub>0</sub></b>	\$ a b a a <b>b<sub>0</sub></b>
<b>a<sub>1</sub></b>	a b a \$ a <b>b<sub>1</sub></b>
<b>a<sub>2</sub></b>	b a \$ a b <b>a<sub>1</sub></b>
<b>a<sub>3</sub></b>	b a a b a <b>\$</b>
<b>b<sub>0</sub></b>	a \$ a b a <b>a<sub>2</sub></b>
<b>b<sub>1</sub></b>	a a b a \$ <b>a<sub>3</sub></b>

$\text{Occ}(c, k) = \# \text{ of } c \text{ in the first } k \text{ characters of } \text{BWT}(S)$ , aka the LF mapping.

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

$F$	$L$	<b>a</b>	<b>b</b>
\$	a	1	0
<b>a</b>	<b>b</b>	1	1
<b>a</b>	<b>b</b>	1	2
<b>a</b>	a	2	2
<b>a</b>	\$	2	2
<b>b</b>	a	3	2
<b>b</b>	a	4	2

We infer **b<sub>0</sub>** and **b<sub>1</sub>** 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 5<sup>th</sup> row.  
Call pre-calculated rows *checkpoints*.

*Tally*

$F$	$L$	<b>a</b>	<b>b</b>
\$	a	1	0
a	b		
a	b		
a	a		
a	\$		
b	a	3	2
b	a		

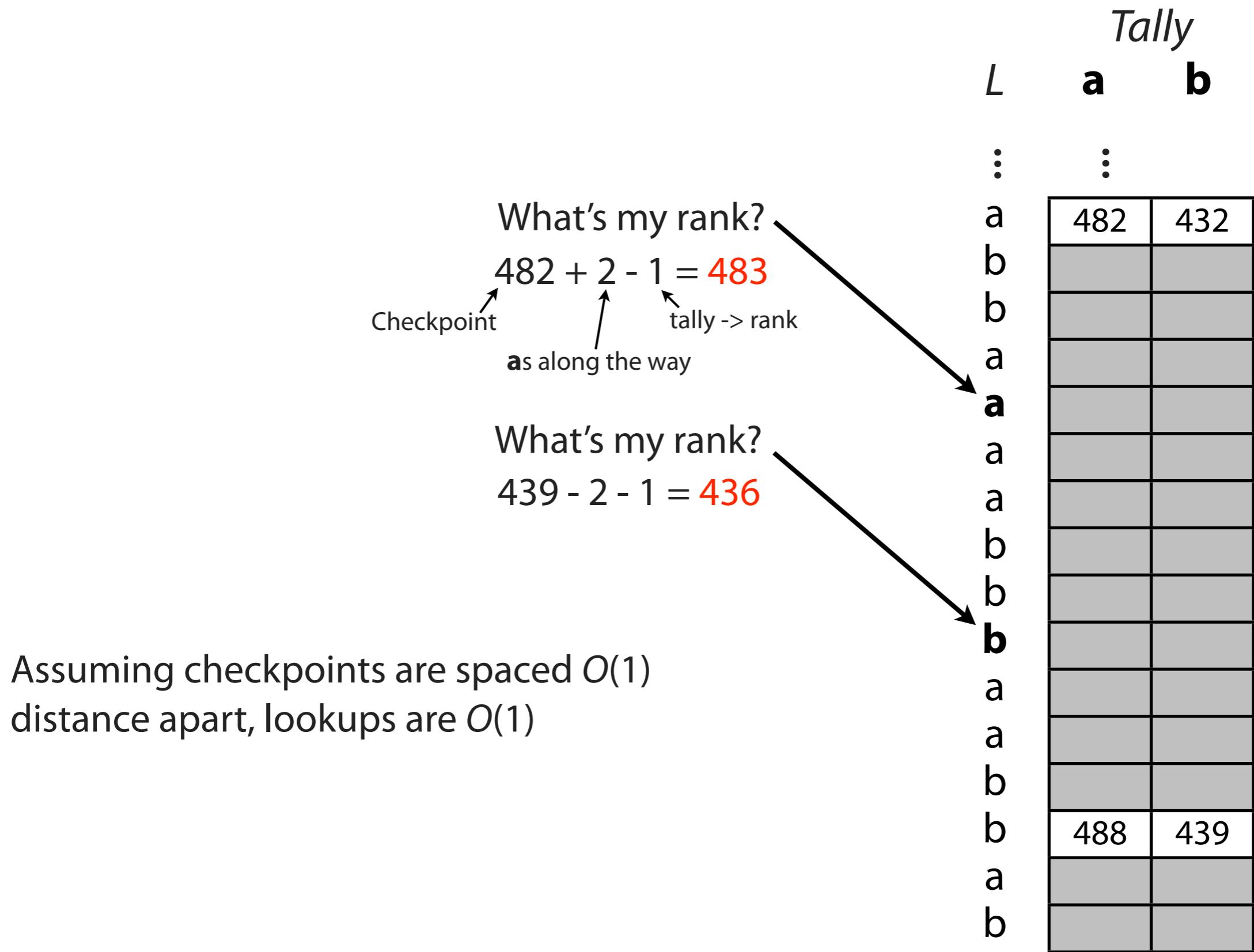
Lookup here succeeds as usual

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



# FM Index: fast rank calculations

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

		<i>Tally</i>	
<i>F</i>	<i>L</i>	<b>a</b>	<b>b</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

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*

**rank(a,3) = 2**

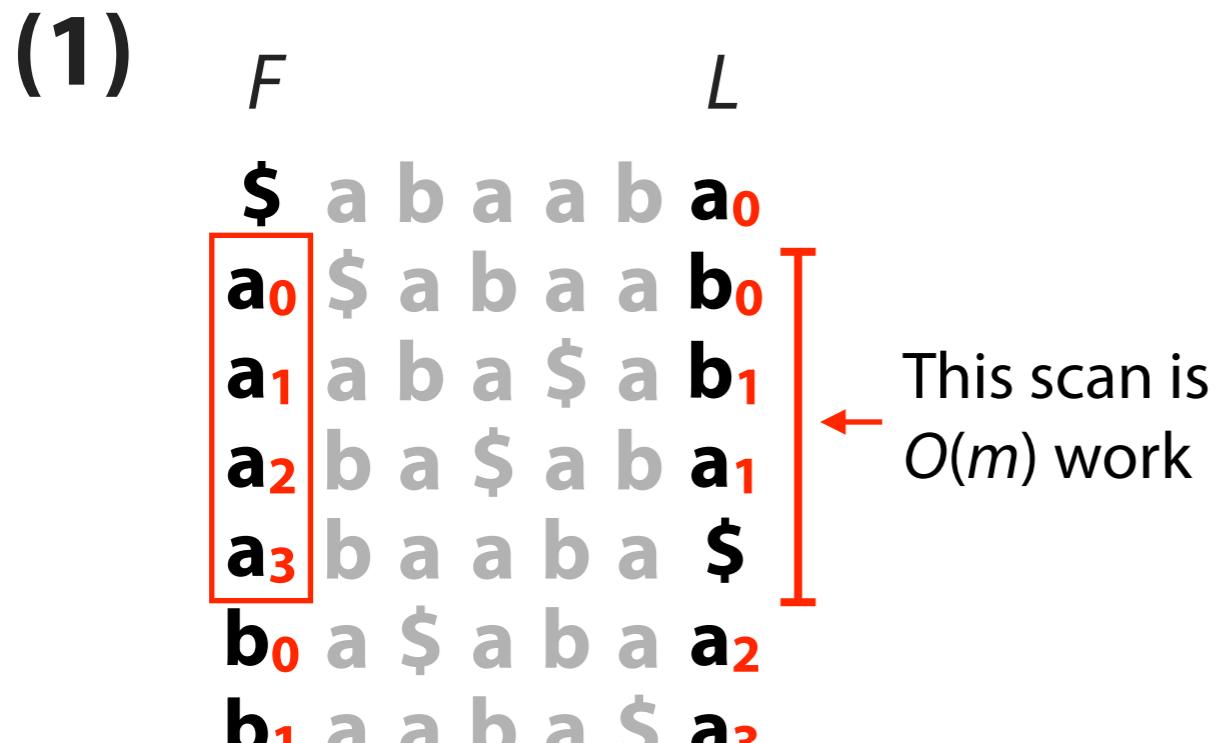
**rank(a,5) = 3**

**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  $\Sigma$  bit-vectors, each of  $o(m)$  bits.

# FM Index: a few problems

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



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

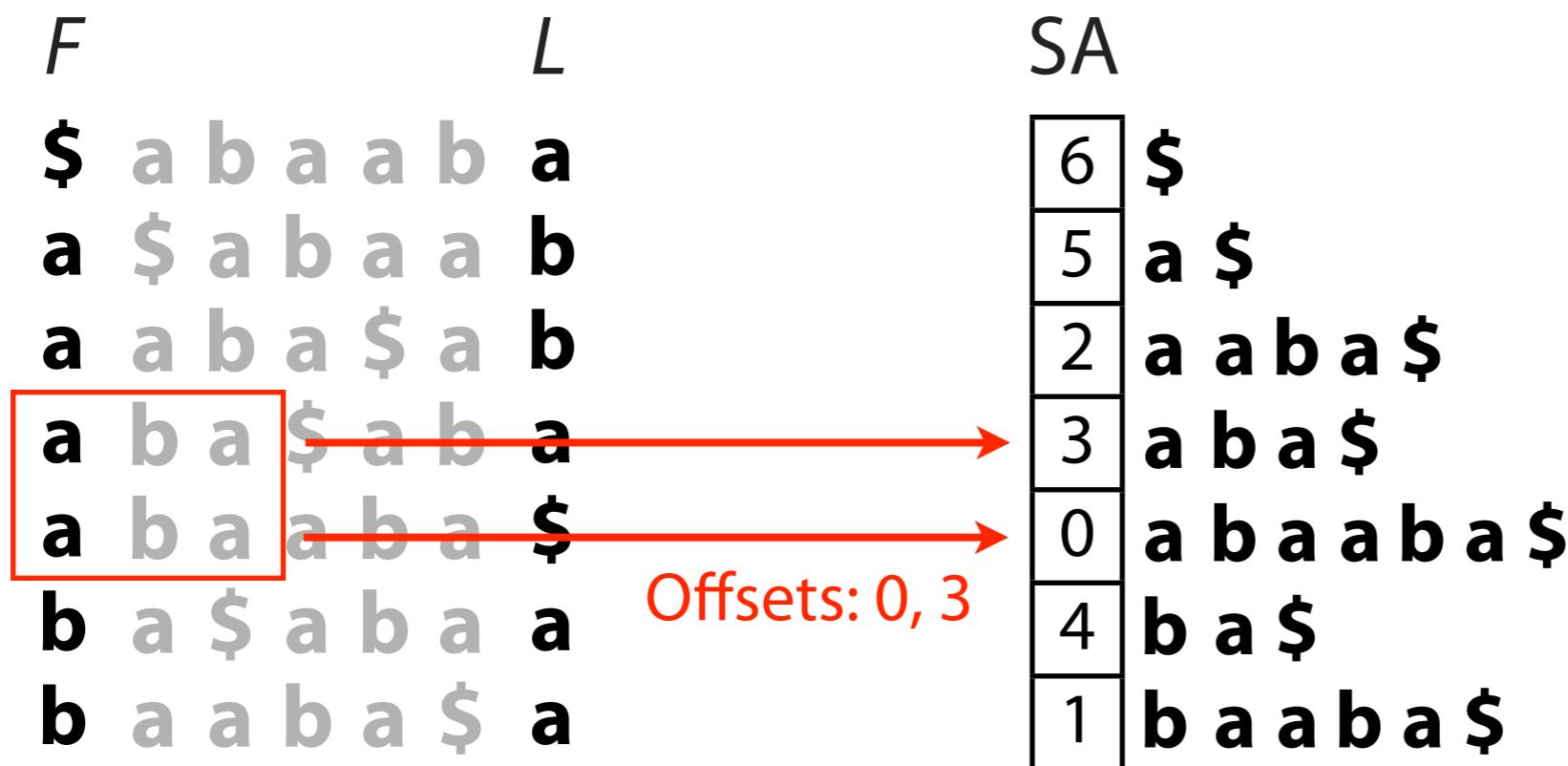
# FM Index: a few problems

Not yet solved:

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

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

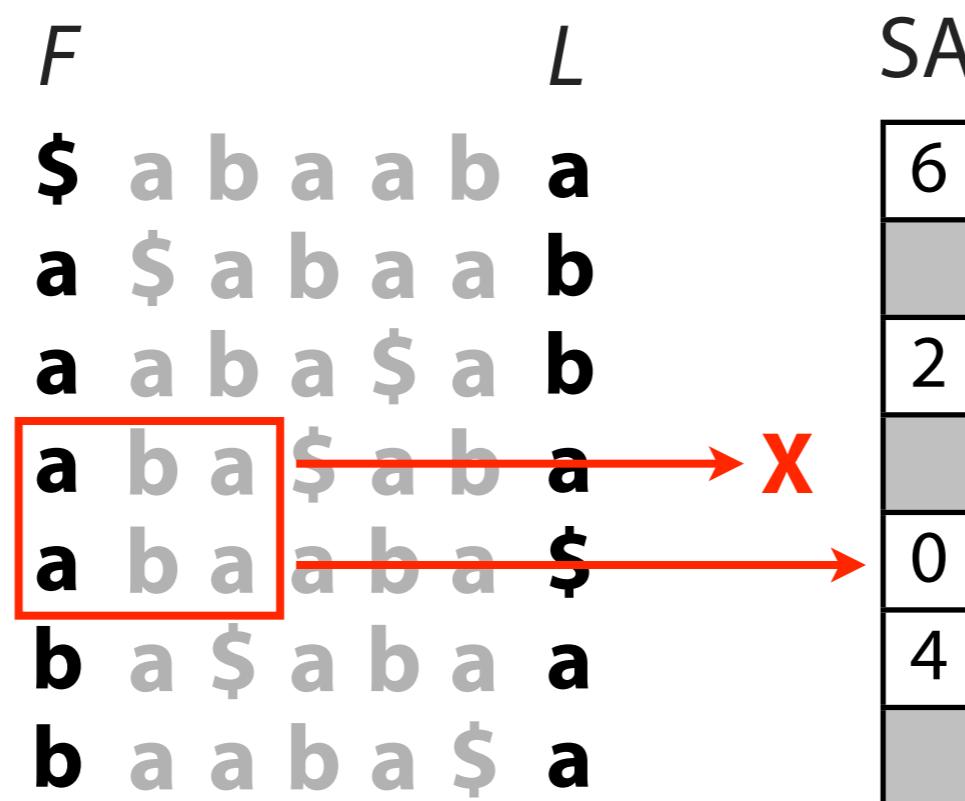
\$	a b a a b	<b>a<sub>0</sub></b>
a <sub>0</sub>	\$ a b a a	<b>b<sub>0</sub></b>
a <sub>1</sub>	a b a \$ a	<b>b<sub>1</sub></b>
a <sub>2</sub>	b a \$ a b	<b>a<sub>1</sub></b>
a <sub>3</sub>	b a a b a	\$
b <sub>0</sub>	a \$ a b a	<b>a<sub>2</sub></b>
b <sub>1</sub>	a a b a \$	<b>a<sub>3</sub></b>



But SA requires  $m$  integers

# FM Index: resolving offsets

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

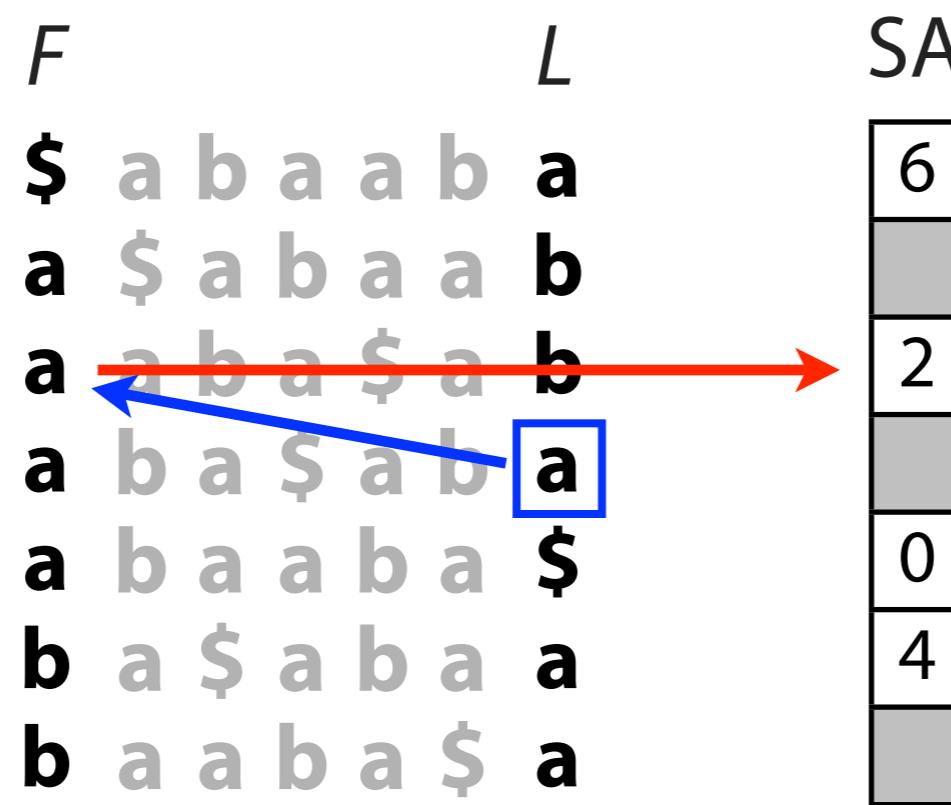


Lookup for row 4 succeeds - we kept that entry of SA

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 beginning of row 2



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	<b>a<sub>0</sub></b>
<b>a<sub>0</sub></b>	\$	a	b	a	a	<b>b<sub>0</sub></b>
<b>a<sub>1</sub></b>	a	b	a	\$	a	<b>b<sub>1</sub></b>
<b>a<sub>2</sub></b>	b	a	\$	a	b	<b>a<sub>1</sub></b>
<b>a<sub>3</sub></b>	b	a	a	b	a	\$
<b>b<sub>0</sub></b>	a	\$	a	b	a	<b>a<sub>2</sub></b>
<b>b<sub>1</sub></b>	a	a	b	a	\$	<b>a<sub>3</sub></b>

**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 ( $F$ ):  $\sim |\Sigma|$  integers
- Last column ( $L$ ):  $m$  characters
- SA sample:  $m \cdot a$  integers, where  $a$  is fraction of rows kept
- Checkpoints:  $m \times |\Sigma| \cdot b$  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 =  $\sim 400$  MB
  - Checkpoints: 3 billion \* 4 bytes/char / 128 =  $\sim 100$  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.

# 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  $k \geq 0$  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  $O(p + \text{occ} \log^2 u)$  time on a RAM. The space occupancy is bounded by  $5H_k(T) + O\left(\frac{\log \log u}{\log u}\right)$  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  $O(p + \text{occ} \log^\epsilon u)$  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  $k \geq 0$ .* ■

# Bowtie2 : Building a gap-aware aligner off of Bowtie

Brief Communication | Published: 04 March 2012

## Fast gapped-read alignment with Bowtie 2

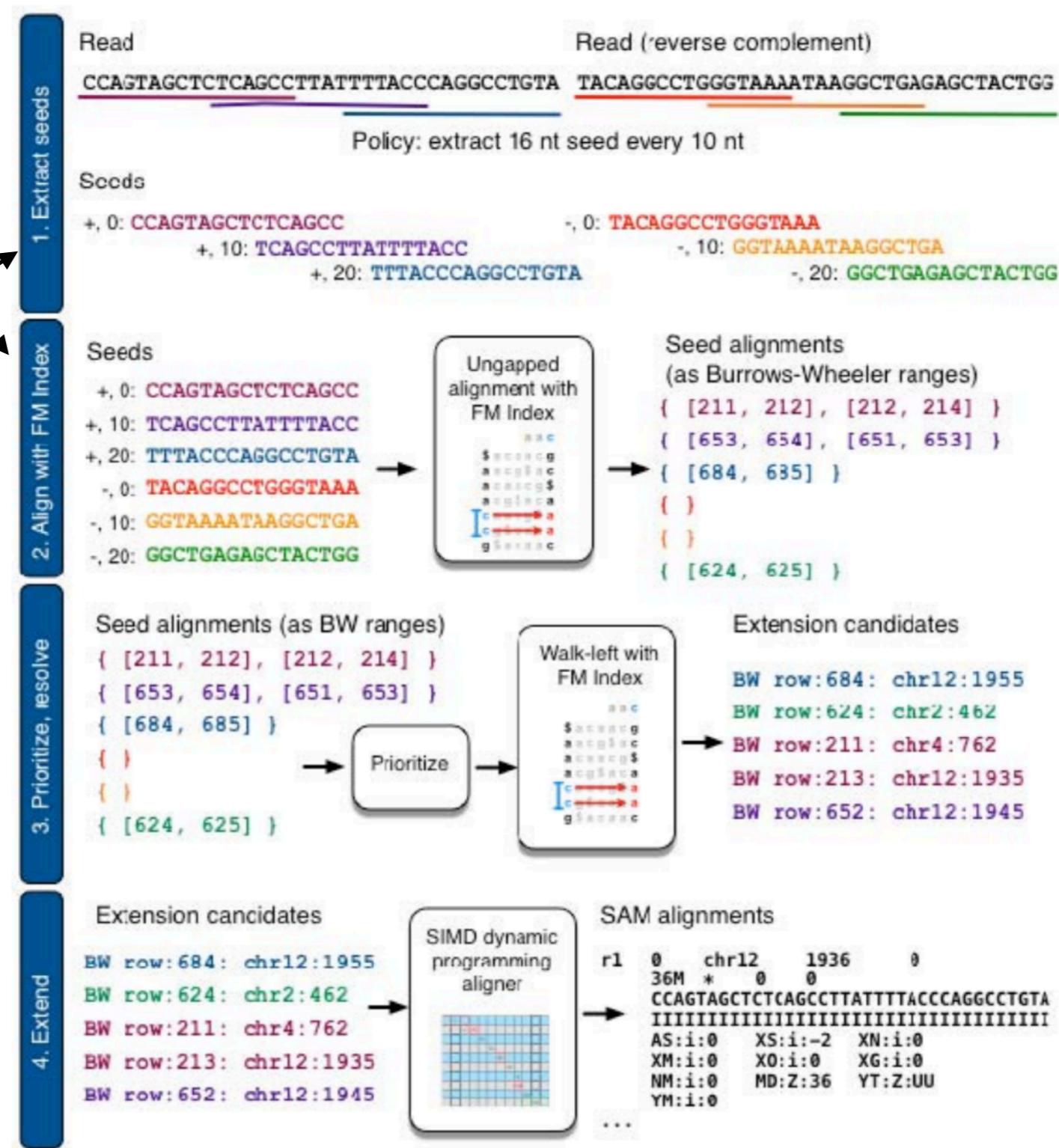
Ben Langmead  & Steven L Salzberg

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

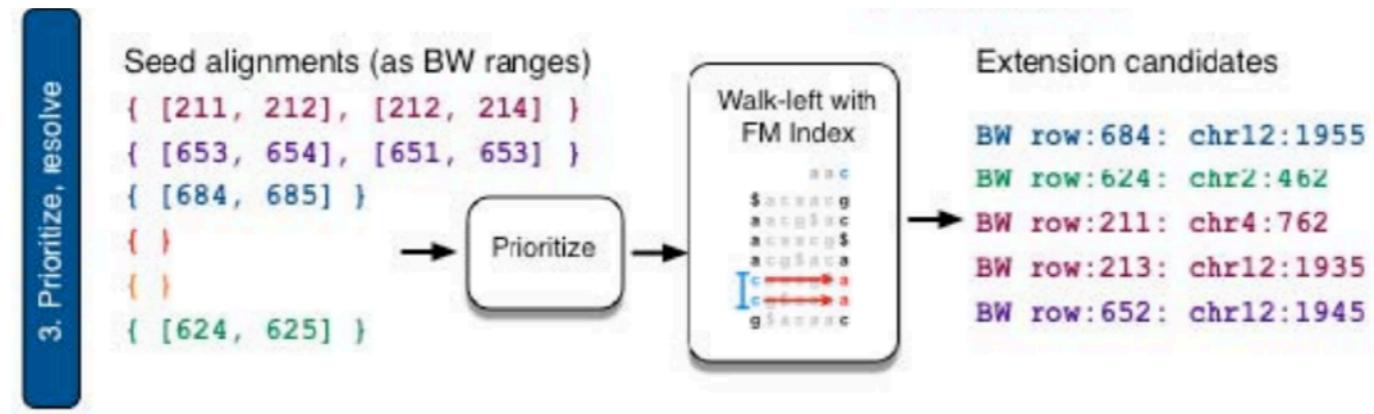
# Bowtie2 proceeds in 4 phases

Supplementary Figure 1

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



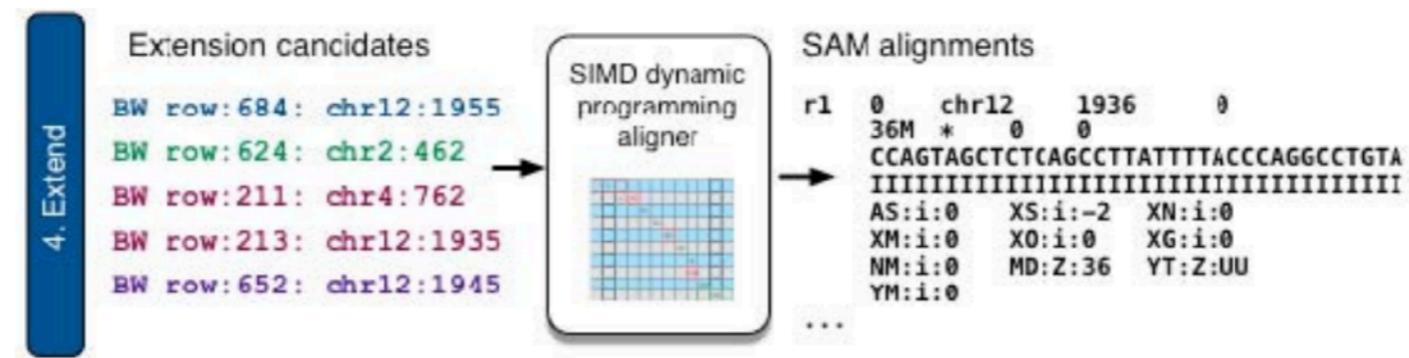
# Phase 3 (prioritizing seeds)



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)



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.

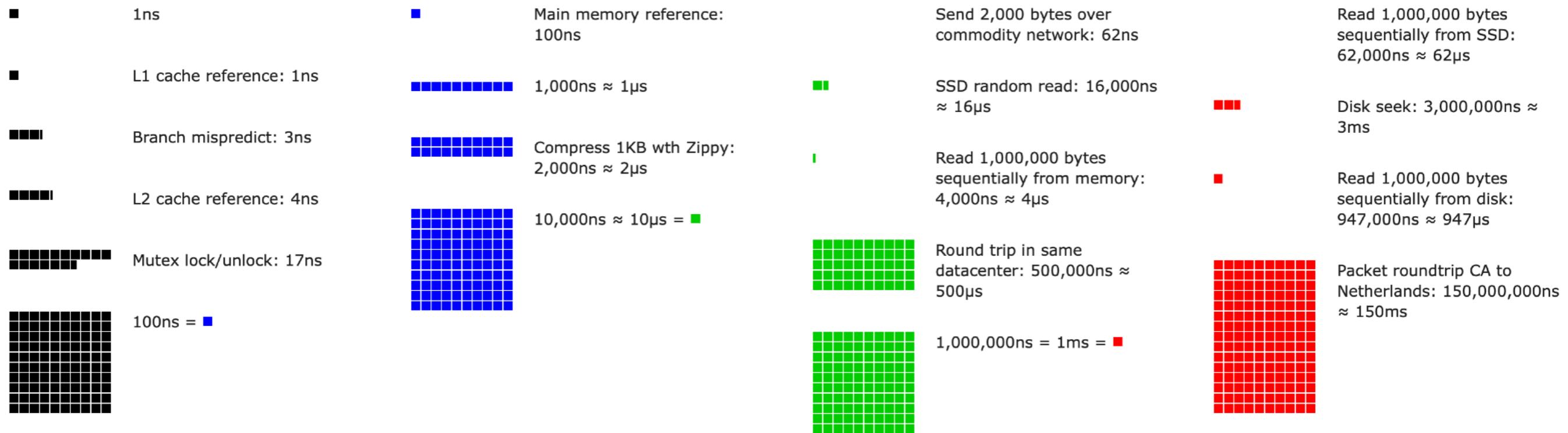
# Hierarchical FM-index

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.

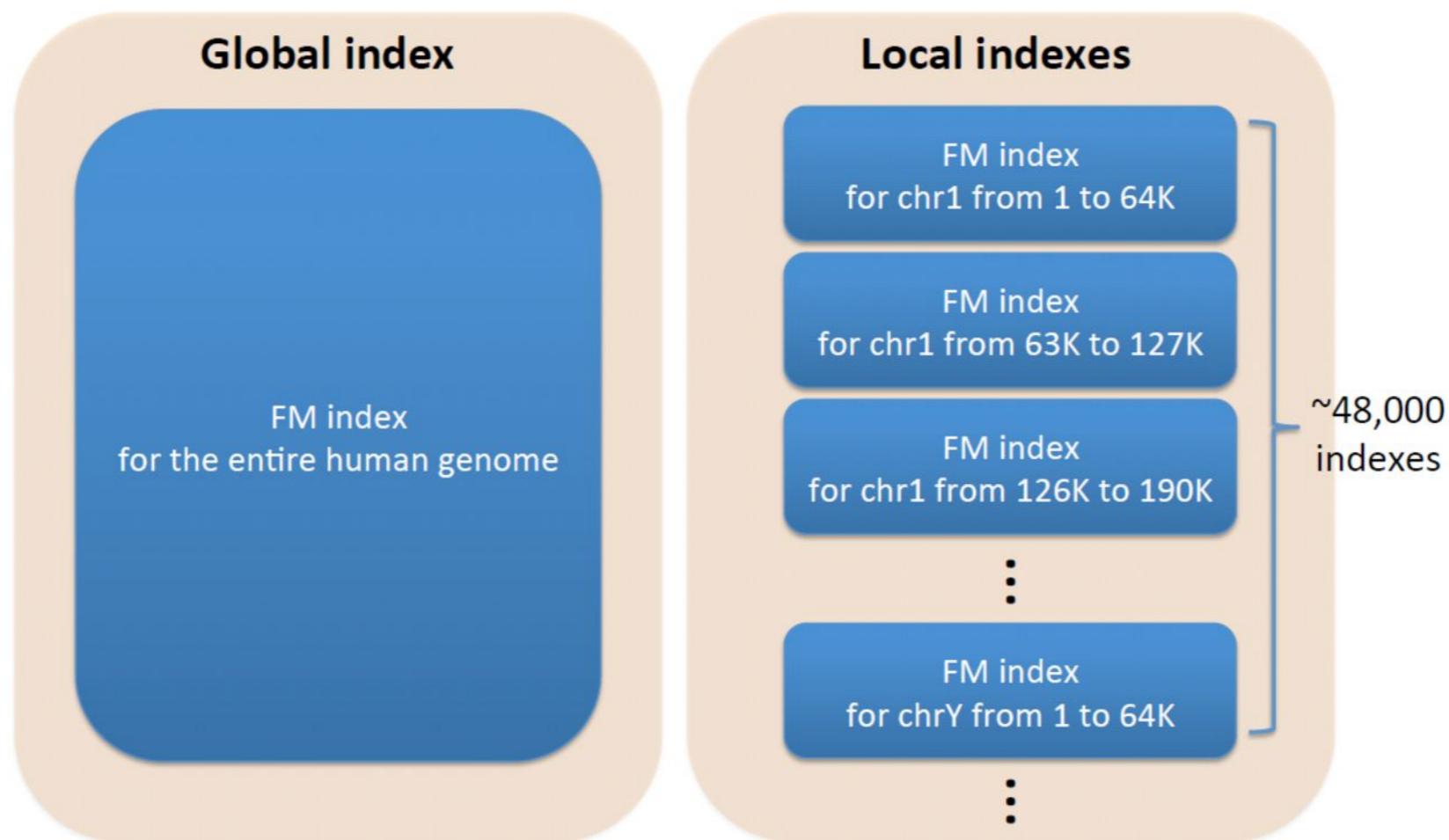
# Cache is \$\$



# Search in the Hierarchical FM-index

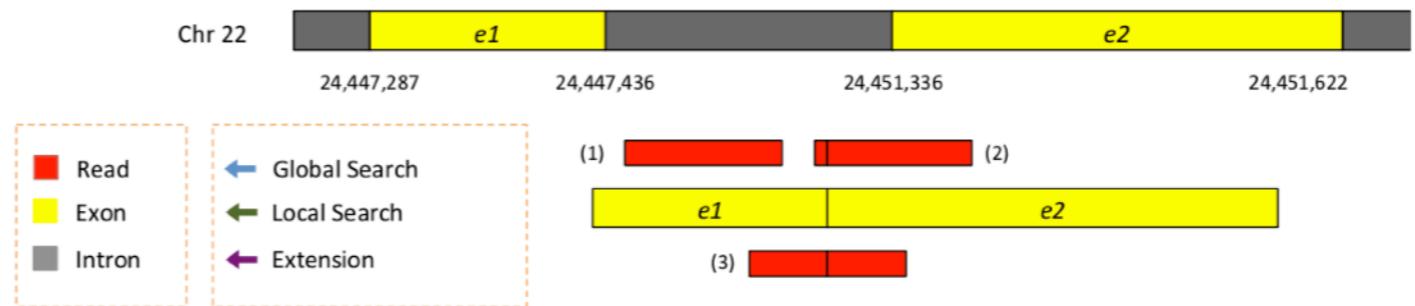
**Idea:** Start search in the global FM-index, where it is *slow*. Once a sufficiently-long suffix has been found to restrict the pattern to one or a few local indices, continue extending the pattern in the local index, where it is *fast*.

## Hierarchical Indexing

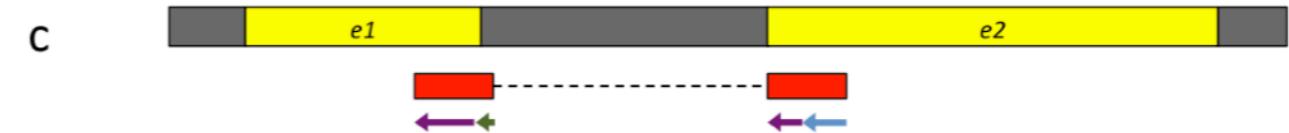
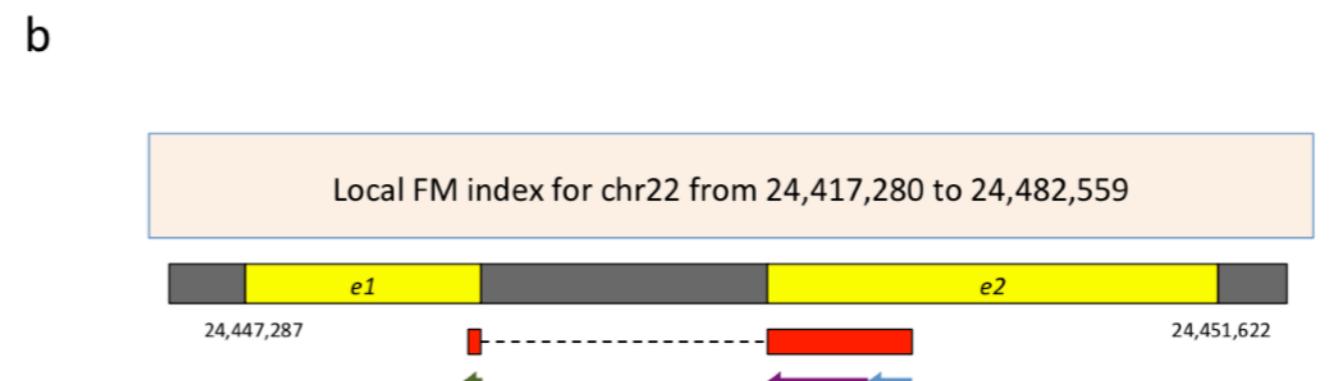
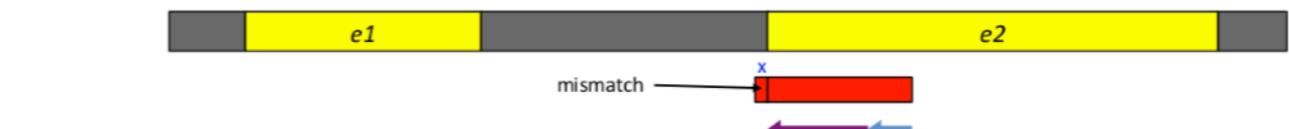


# Using a hierarchical FM-index for search

**Global search of short suffix gives 1 or more potential positions**

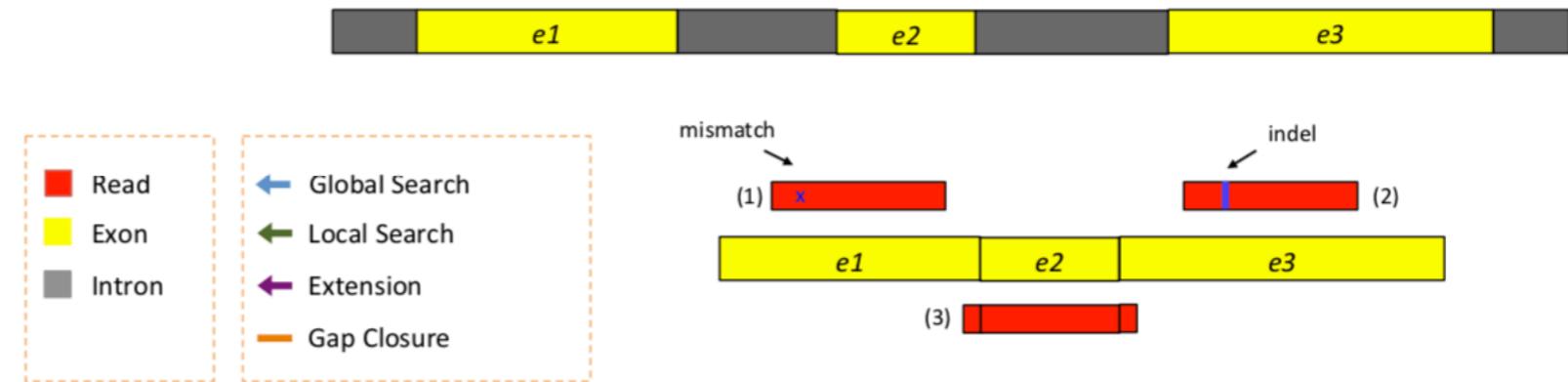


**Local index is used to “extend” the matches within this region, as well as to perform local search for upstream exons with the same local index.**



# Mapping in the presence of substitutions & indels

**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**

