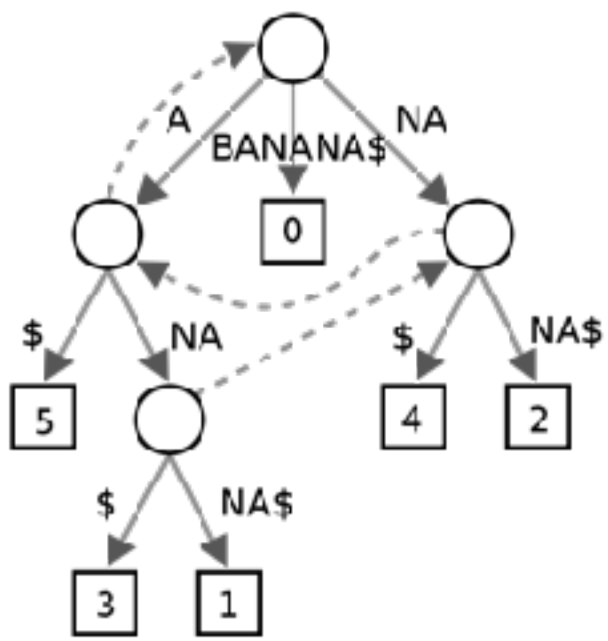
Burrows-Wheeler Transform & FM Index

Ben Langmead

Johns Hopkins
Whiting School of Engineering

Department of Computer Science

Indexing with suffixes



Suffix Tree

6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

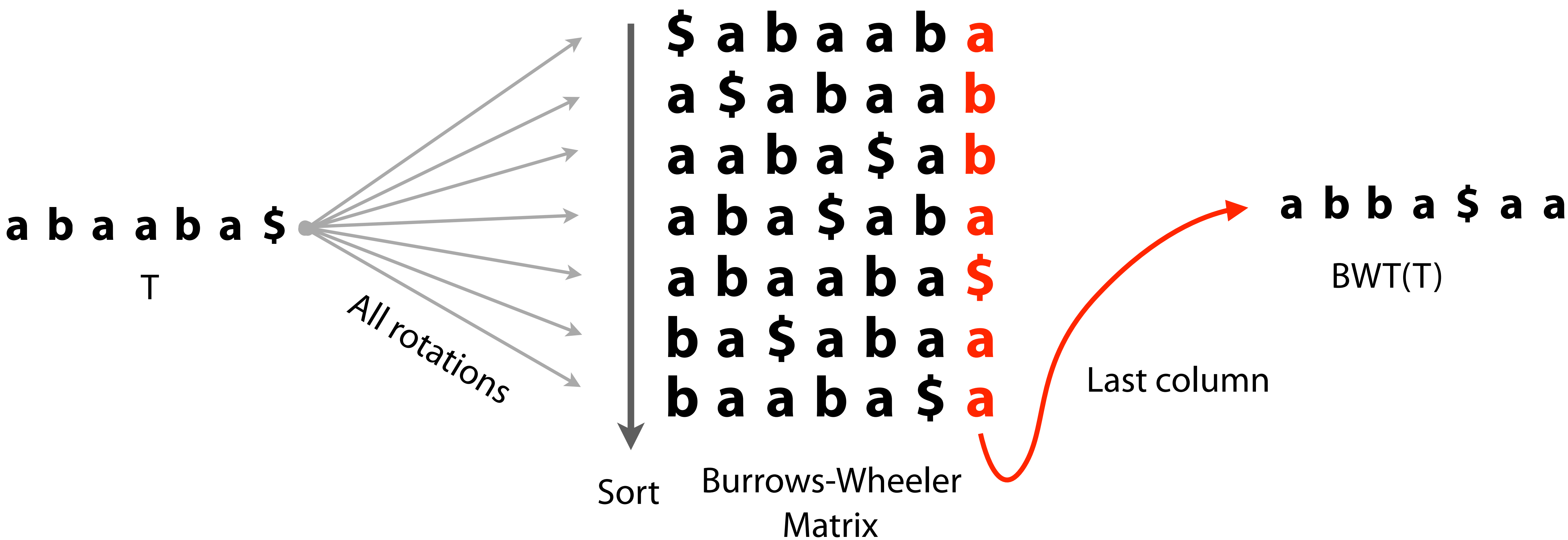
Suffix Array

$ B A N A N **A**
A $ B A N A **N**
A N A $ B A **N**
A N A N A $ **B**
B A N A N A **$**
N A $ B A N **A**
N A N A $ B **A**

FM Index

Burrows-Wheeler Transform

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



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

All rotations

a b a a b a $

b a a b a $ a

a a b a $ a b

a b a $ a b a

b a $ a b a a

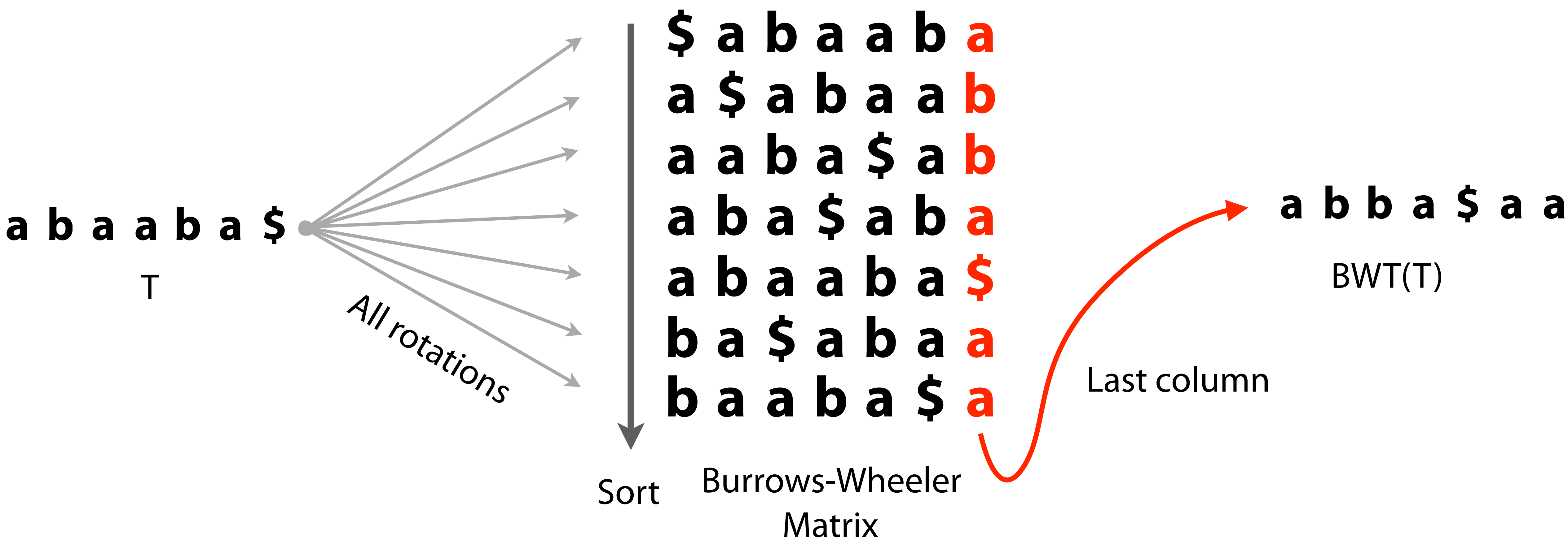
a $ a b a a b

$ a b a a b a

(then they repeat)

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 range(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_'
```

http://j.mp/CG_BWT

Burrows-Wheeler Transform

a b a a b a $

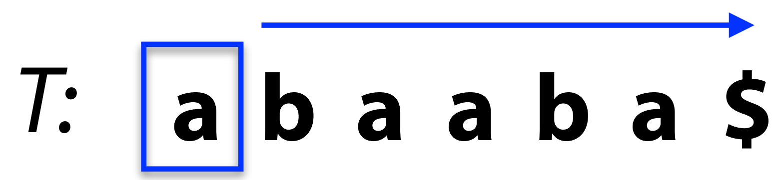
T

$ a b a a b **a**	→	**a**	$ a b a a b
a $ a b a a **b**		**b**	a $ a b a a
a a b a $ a **b**		**b**	a a b a $ a
a b a $ a b **a**		**a**	a b a $ a b
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**		**a**	b a a b a $
		BWT	Right contexts

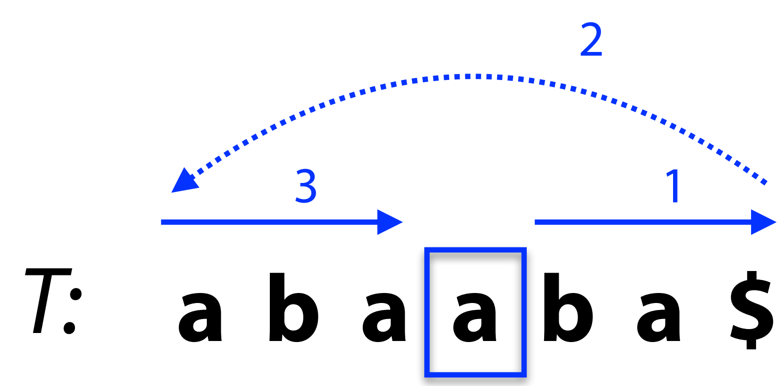
BWT(T) orders T's characters according to alphabetical order of their right contexts in T

Right context

The right context of a position in *T* consists of everything that comes after it with "wrap around"

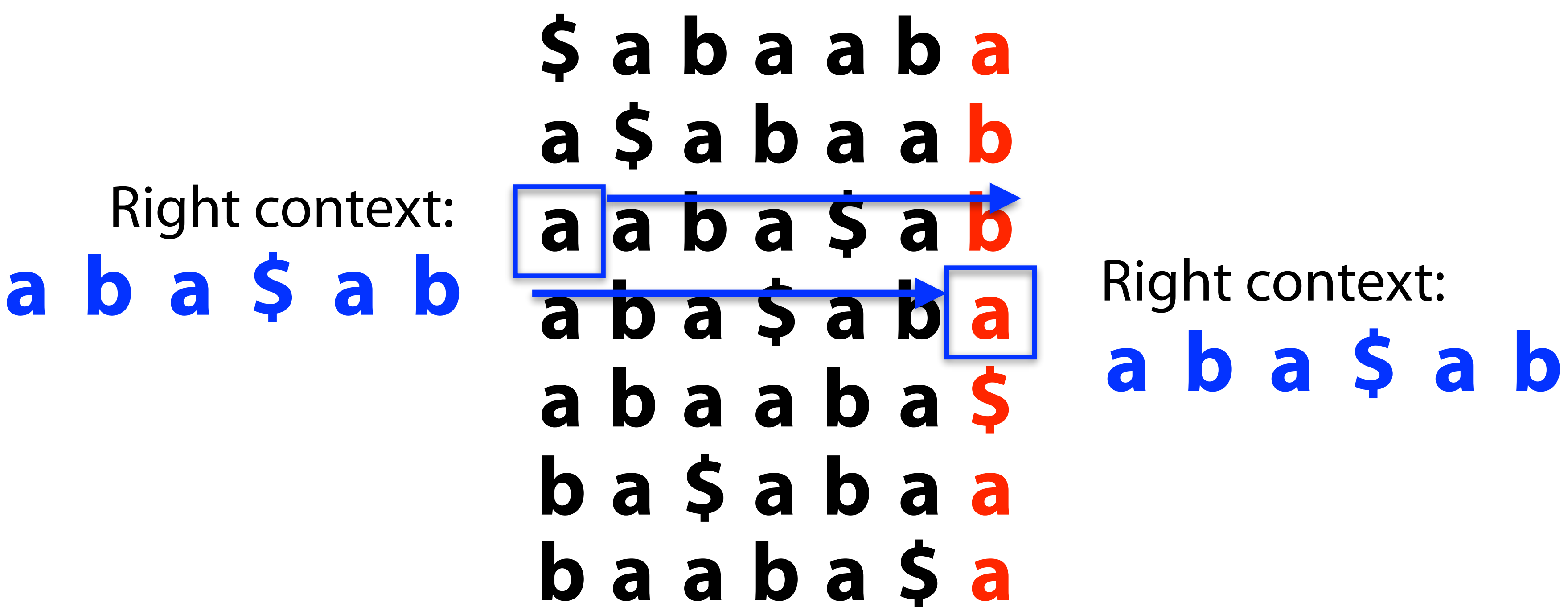


Right context: **b a a b a $**



Right context: **b a $ a b a**

Burrows-Wheeler Transform



Burrows-Wheeler Transform

Sorted by *right-context*

Gives “structure” to BWT(T), making it more compressible

final char (L)	sorted rotations
a	n to decompress. It achieves compression
o	n to perform only comparisons to a depth
o	n transformation} This section describes
o	n transformation} We use the example and
o	n treats the right-hand side as the most
a	n tree for each 16 kbyte input block, enc
a	n tree in the output stream, then encodes
i	n turn, set $L[i]$ to be the
i	n turn, set $R[i]$ to the
o	n unusual data. Like the algorithm of Man
a	n use a single set of probabilities table
e	n using the positions of the suffixes in
i	n value at a given point in the vector $R
e	n we present modifications that improve t
e	n when the block size is quite large. Ho
i	n which codes that have not been seen in
i	n with ch appear in the {\em same order
i	n with ch. In our exam
o	n with Huffman or arithmetic coding. Bri
o	n with figures given by Bell~\cite{bell}.

Figure 1: Example of sorted rotations. Twenty consecutive rotations from the sorted list of rotations of a version of this paper are shown, together with the final character of each rotation.

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

BWM is related to the suffix array

BWM(T)	SA(T)	Suffix
$ 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 $

BWM(T)

SA(T)

Same order 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

```
def suffixArray(s):
    """ Given T return suffix array SA(T).  We use Python's sorted
        function here for simplicity, but we can do better. """
    satups = sorted([(s[i:], i) for i in xrange(0, len(s))])
    # Extract and return just the offsets
    return map(lambda x: x[1], satups)
```

Make suffix array

```
def bwtViaSa(t):
    """ Given T, returns BWT(T) by way of the suffix array. """
    bw = []
    for si in suffixArray(t):
        if si == 0: bw.append('$')
        else: bw.append(t[si-1])
    return ''.join(bw) # return string-ized version of list bw
```

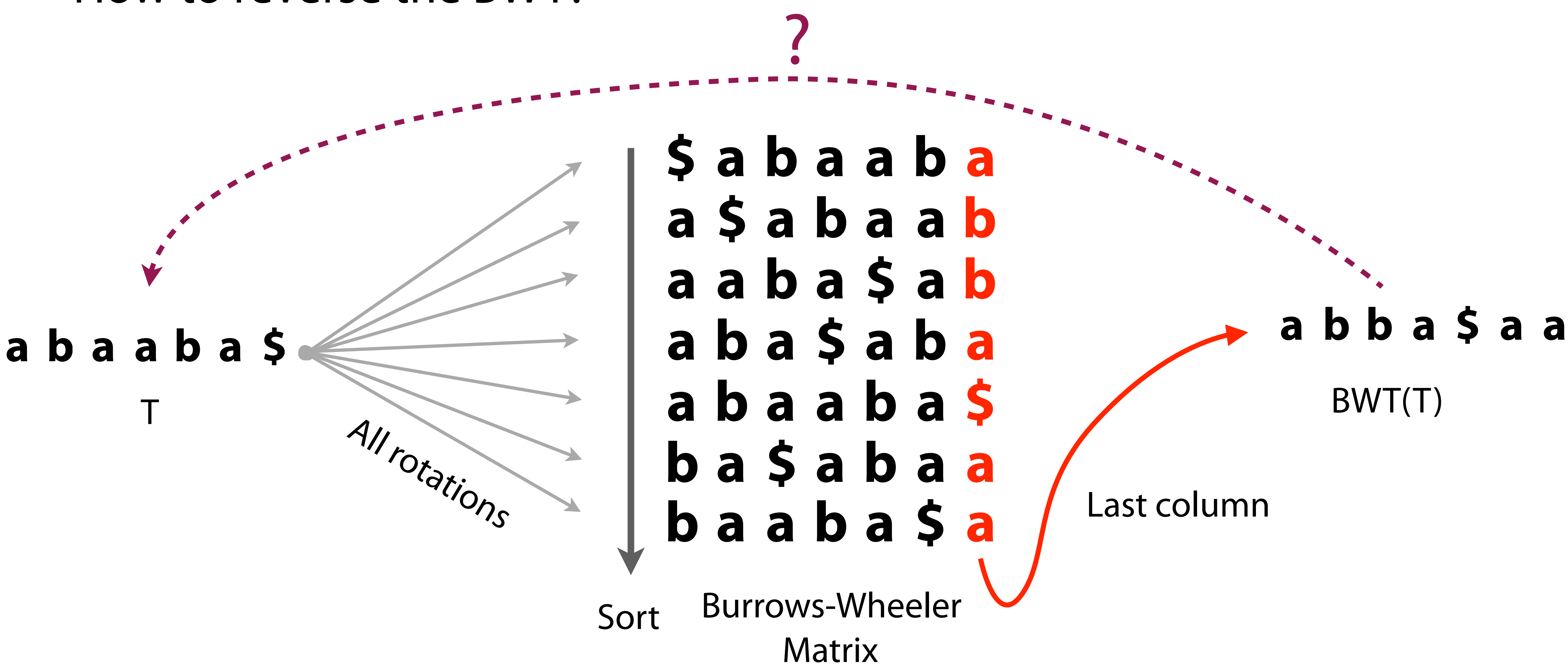
Take characters just to the left of the sorted suffixes

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

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

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

Python example: http://bit.ly/CG_BWT_SimpleBuild

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*.

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

Ranks aren't explicitly stored; they are just for illustration

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

Burrows-Wheeler Transform

BWM with T-ranking:

F						*L*
$\$$	a_0	b_0	a_1	a_2	b_1	$\mathbf{a_3}$
$\mathbf{a_3}$	$\$$	a_0	b_0	a_1	a_2	b_1
$\mathbf{a_1}$	a_2	b_1	a_3	$\$$	a_0	b_0
$\mathbf{a_2}$	b_1	a_3	$\$$	a_0	b_0	$\mathbf{a_1}$
$\mathbf{a_0}$	b_0	a_1	a_2	b_1	a_3	$\$$
b_1	a_3	$\$$	a_0	b_0	a_1	$\mathbf{a_2}$
b_0	a_1	a_2	b_1	a_3	$\$$	$\mathbf{a_0}$

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: $\mathbf{a_3}, \mathbf{a_1}, \mathbf{a_2}, \mathbf{a_0}$

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	$\mathbf{b}_1$
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

Same with **b**s: $\mathbf{b}_1$, $\mathbf{b}_0$

Burrows-Wheeler Transform: LF Mapping

BWM with T-ranking:

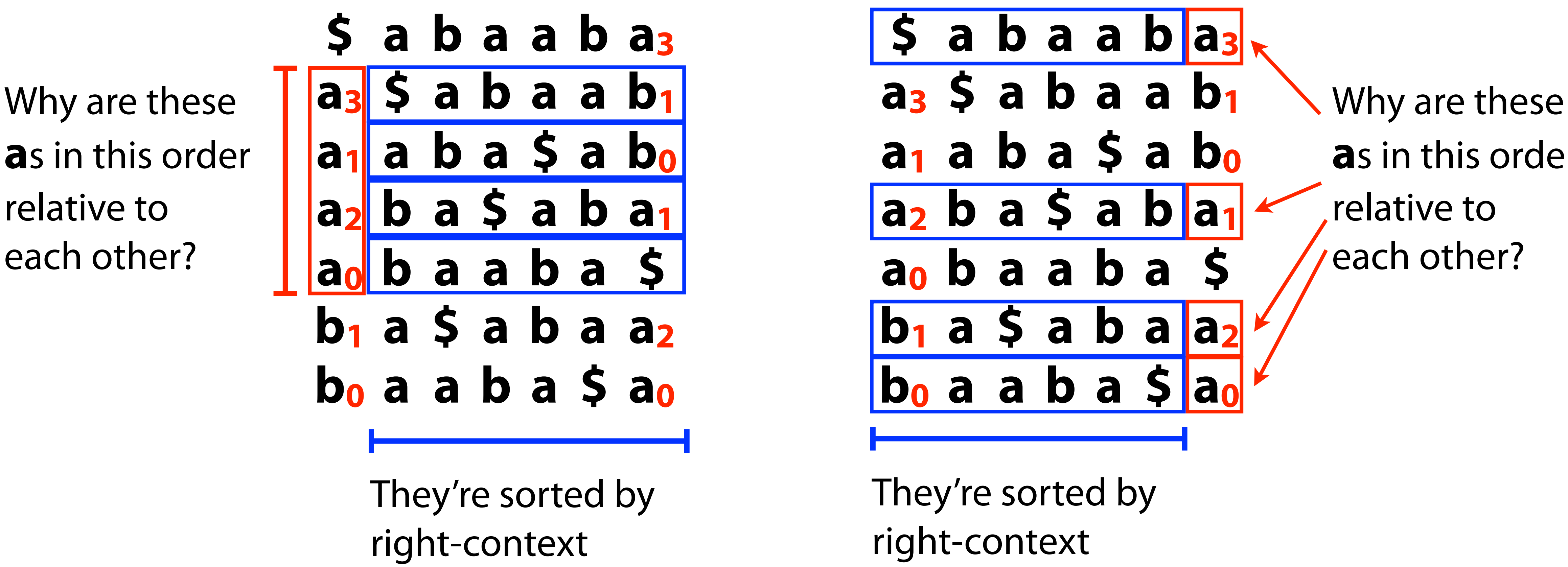
F						*L*
$	a_0	b_0	a_1	a_2	b_1	a_3
a_3	$	a_0	b_0	a_1	a_2	b_1
a_1	a_2	b_1	a_0	$	a_0	b_0
a_2	b_1	a_3	$	a_0	b_0	a_1
a_0	b_0	a_1	a_2	b_1	a_3	$
b_1	a_3	$	a_0	b_0	a_1	a_2
b_0	a_1	a_2	b_1	a_3	$	a_0

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* (i.e. have same rank)

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

Burrows-Wheeler Transform: LF Mapping

Why does the LF Mapping hold?



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:

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

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_0	$\$$	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

Ascending rank

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

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)

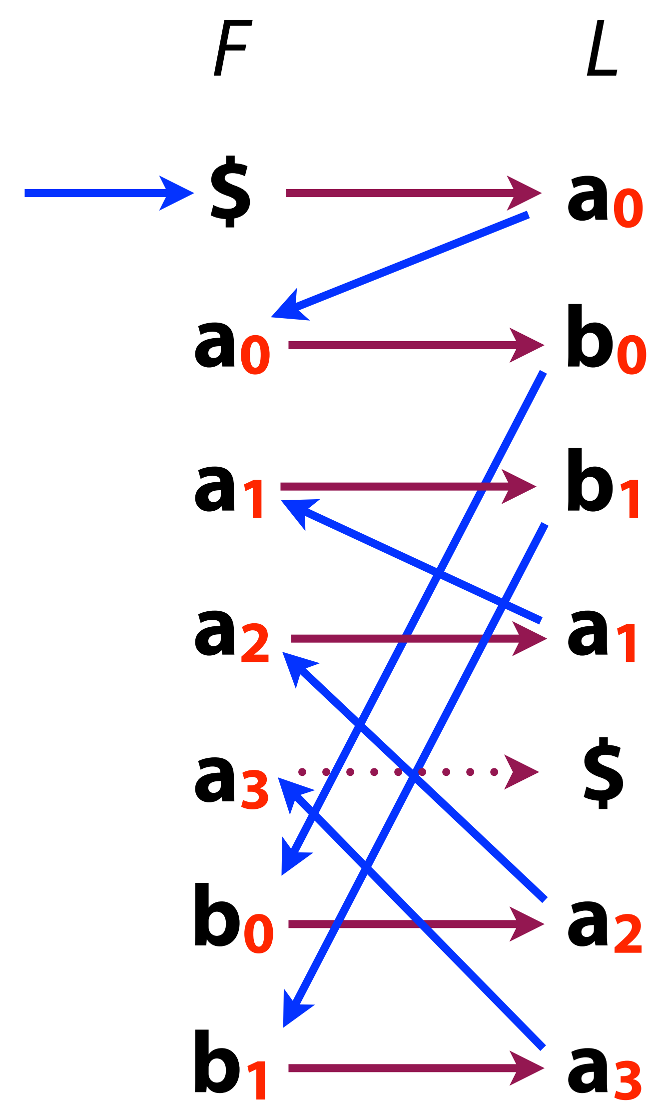
Skip first 100 rows starting with **G** (100 rows)

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

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



Start in first row. *F* must have **$**.

L contains character just prior to **$**: a_0

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

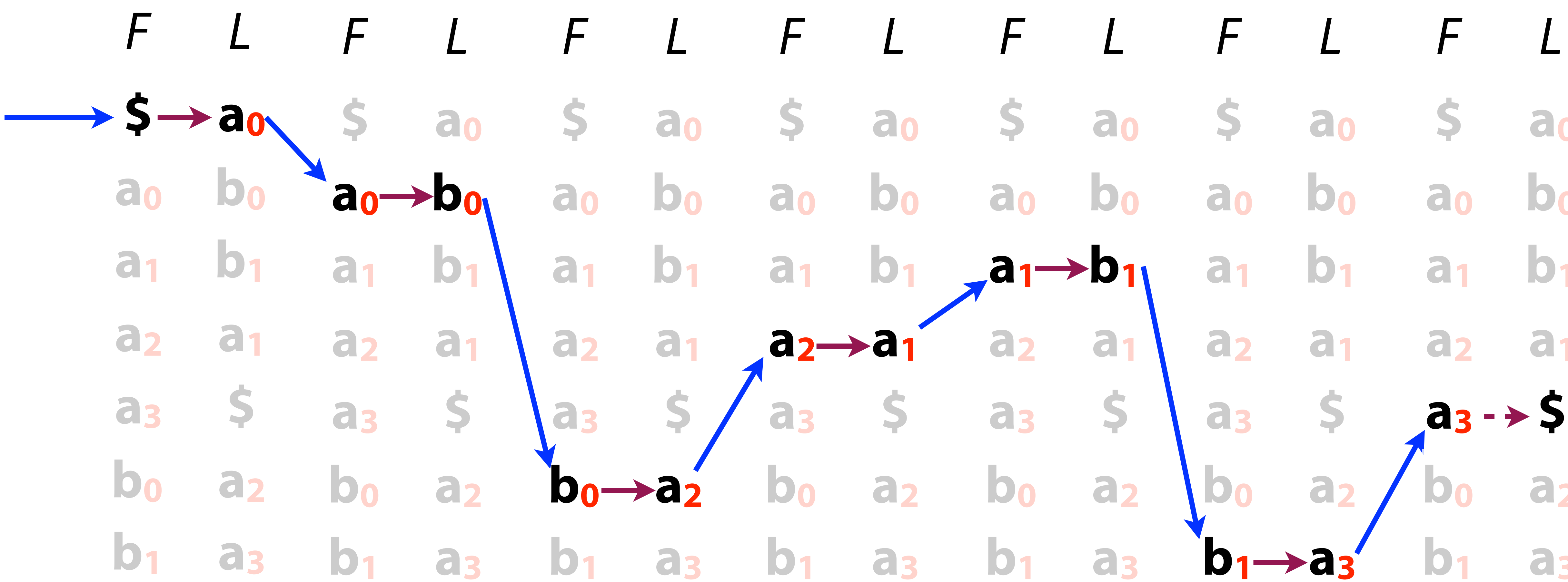
Repeat for b_1, get a_3

Repeat for a_3, get **$** (done)

In reverse order, we saw = a_3 b_1 a_1 a_2 b_0 a_0 **$** = *T*

Burrows-Wheeler Transform: reversing

Another way to visualize:



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

Burrows-Wheeler Transform: reversing

http://bit.ly/CG_BWT_reverse

```
def rankBwt(bw):
    ''' Given BWT string bw, return parallel list of B-ranks.  Also
        returns tots: map from character to # times it appears. '''
    tots = dict()
    ranks = []
    for c in bw:
        if c not in tots: tots[c] = 0
        ranks.append(tots[c])
        tots[c] += 1
    return ranks, tots
```

L

a_0
b_0
b_1
a_1
$
a_2
a_3

{ a: 4, b: 2, $: 1}

Like when we did it by eye, the code depends on *knowing the ranks* of all the characters in L

But **ranks** is big! We'll fix this later

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 auxiliary data structures*

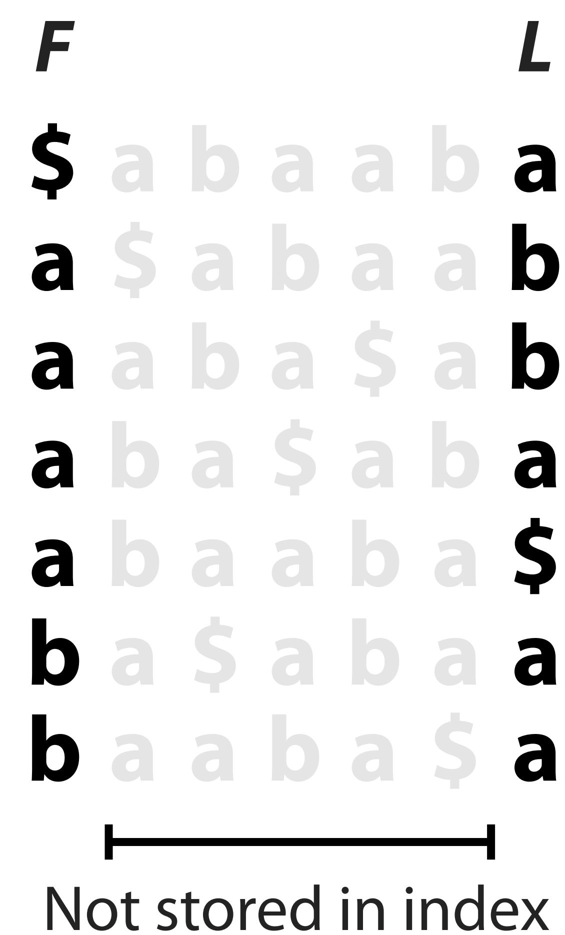
Core of index is ***F*** and ***L*** from BWM:

L is the same size as *T*

F can be represented as array of |Σ| integers

L is compressible (but even uncompressed, it's small compared to suffix array)

We're discarding *T*



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

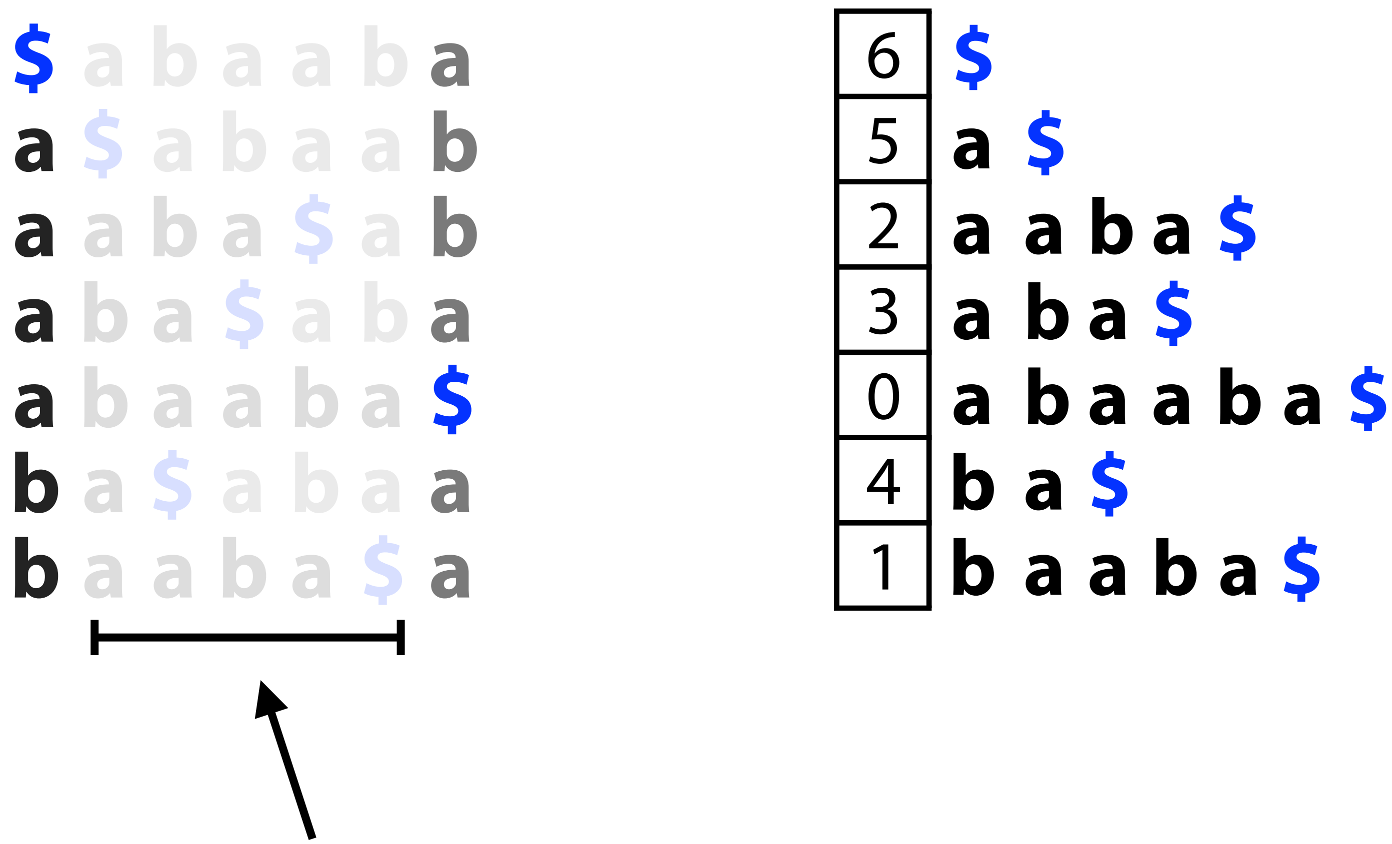
FM Index: querying

How to query?

$	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

FM Index: querying

Can we query like the suffix array?



We don't have these columns, and we don't have T.
Binary search not possible.

FM Index: querying

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

Start with shortest suffix, then match successively longer suffixes

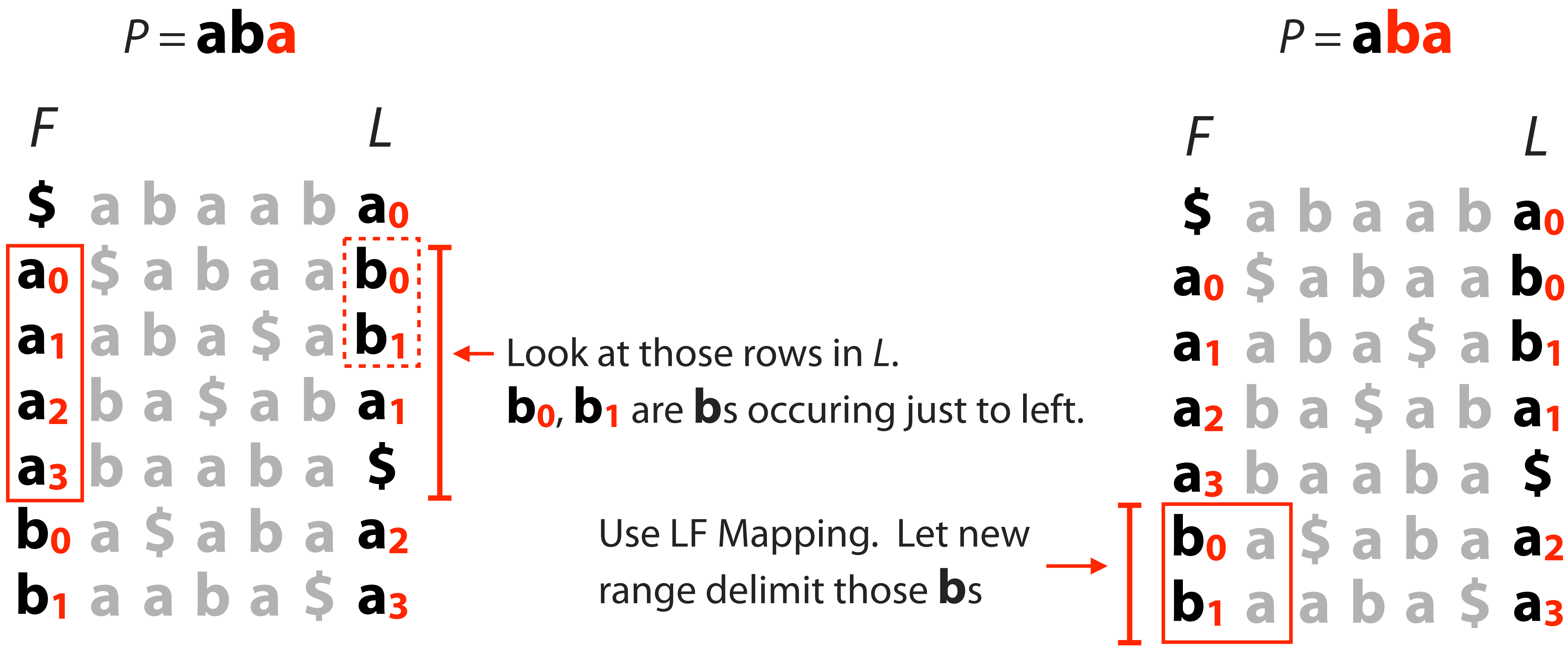
P = **aba**

Easy to find all the rows beginning with **a**

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

FM Index: querying

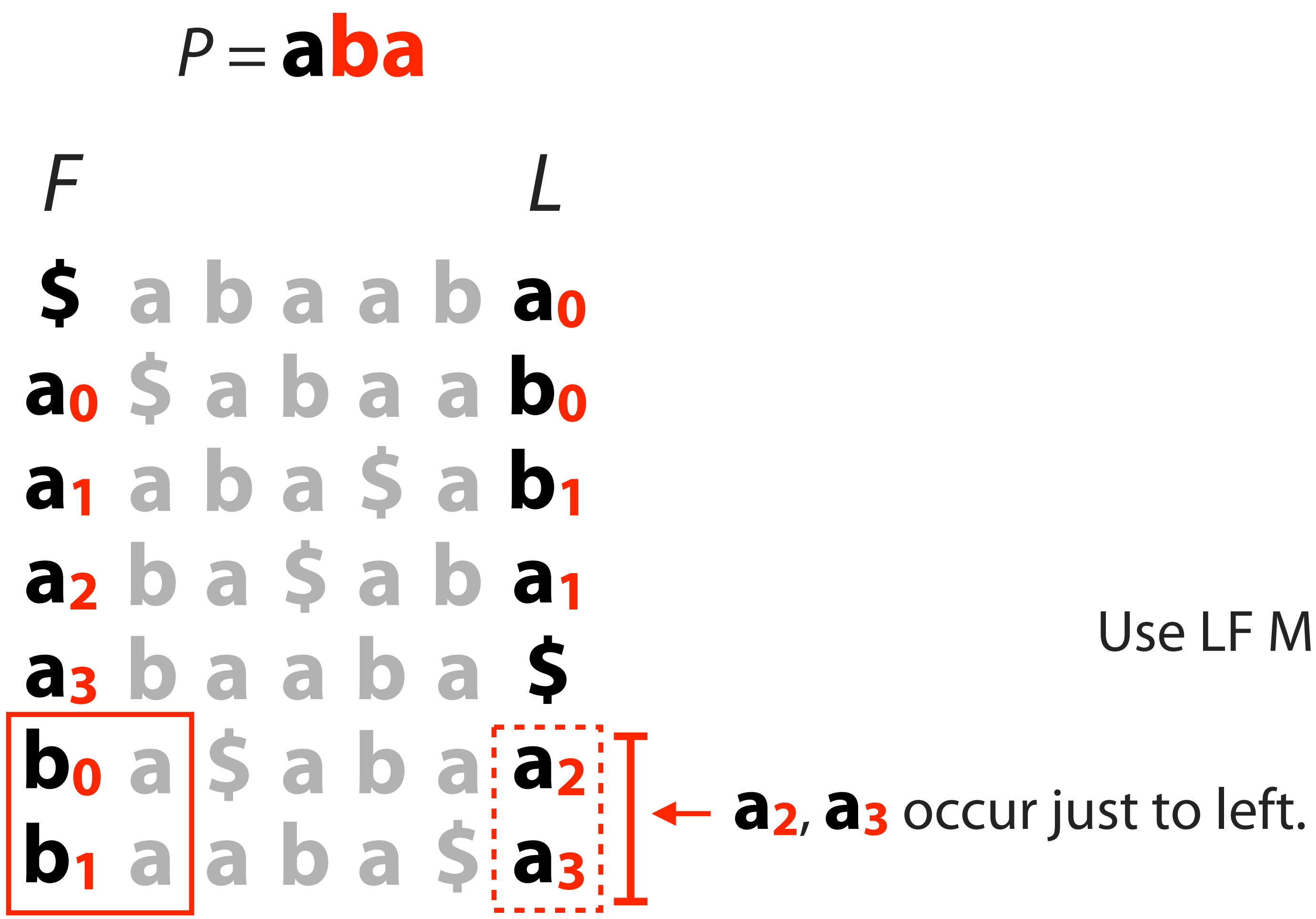
We have rows beginning with **a**, now we want rows beginning with **ba**

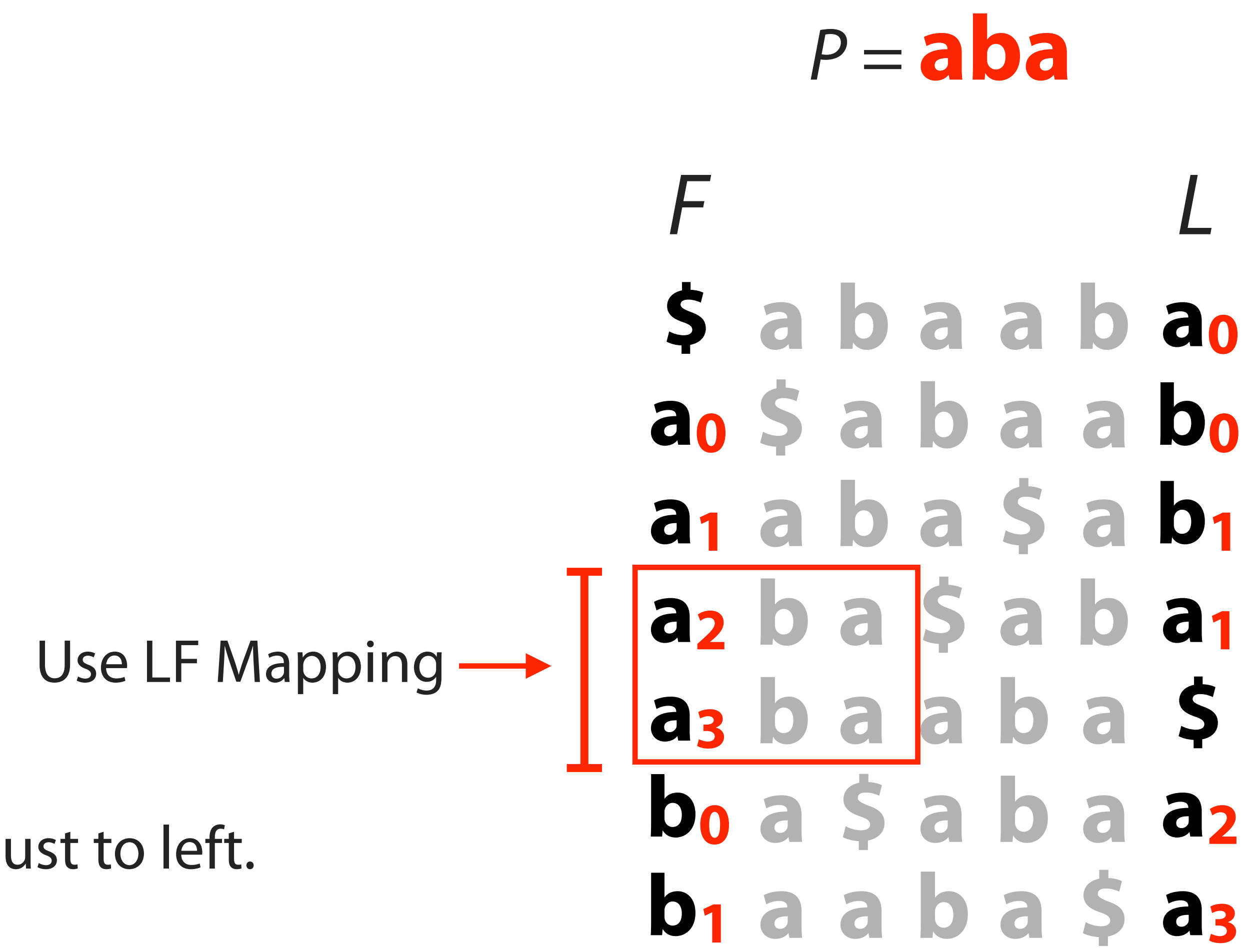


Now we have the rows with prefix **ba**

FM Index: querying

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



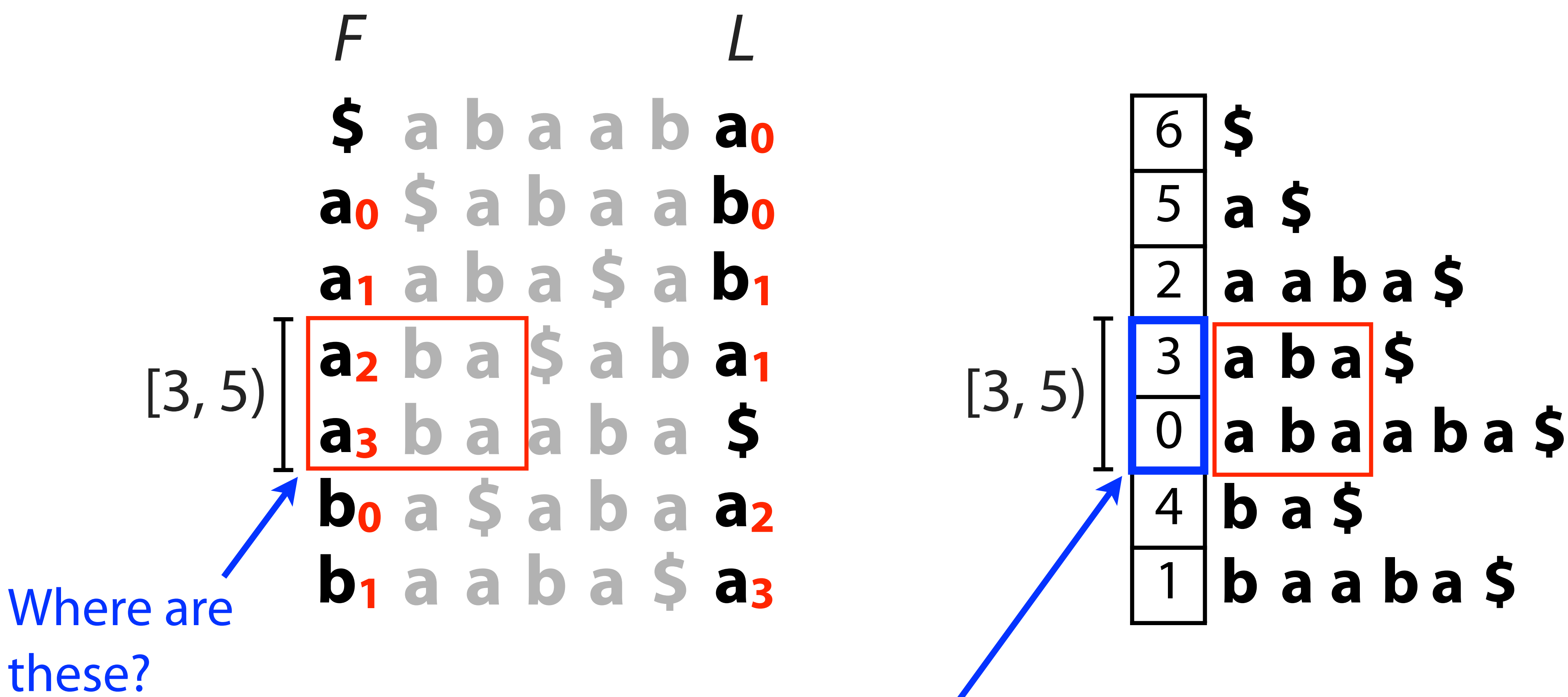


Now we have the rows with prefix **aba**

FM Index: querying

P = **aba**

Got the same range, [3, 5), we would have got from 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 eventually fail to find 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

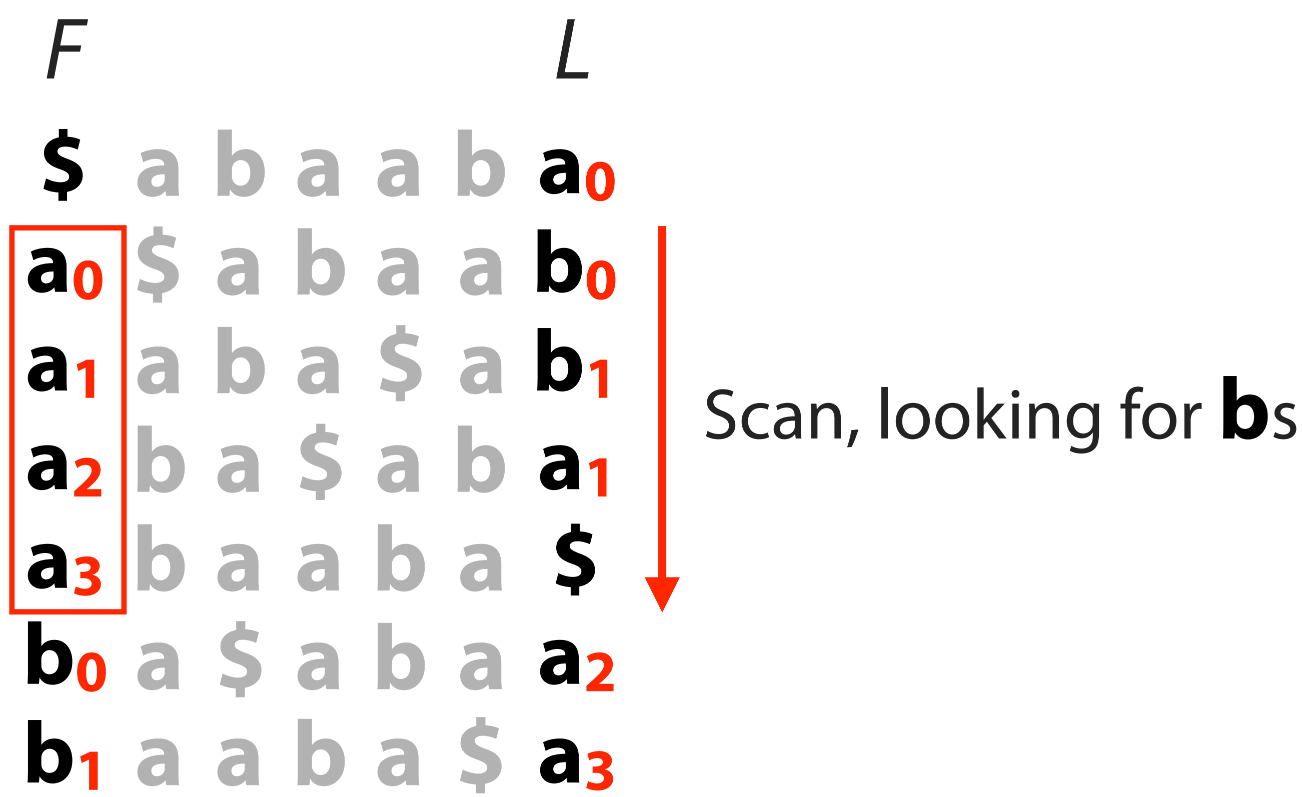
Rows with **ba** prefix

← No **b**s!

FM Index: querying

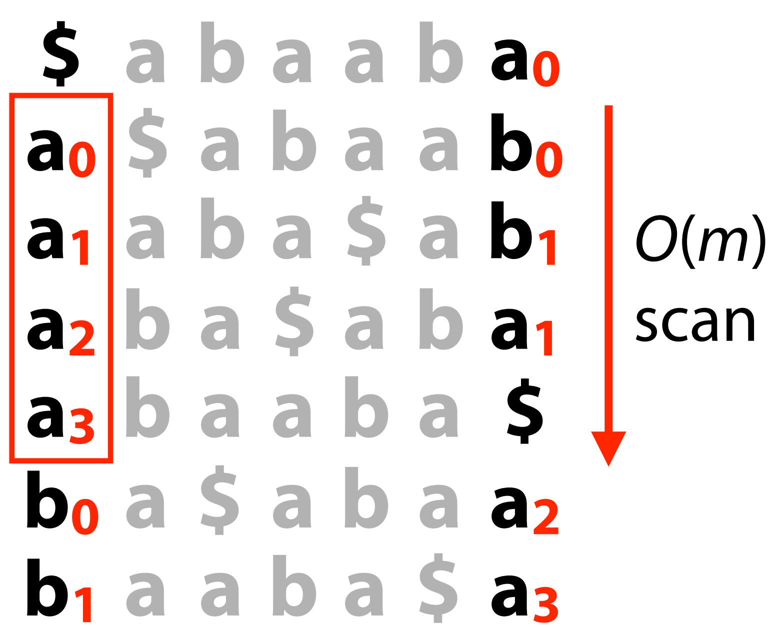
If we *scan* characters in the last column, that can be slow, $O(m)$

$P =$ **aba**



FM Index: lingering issues

(1) Scanning for preceding character is slow

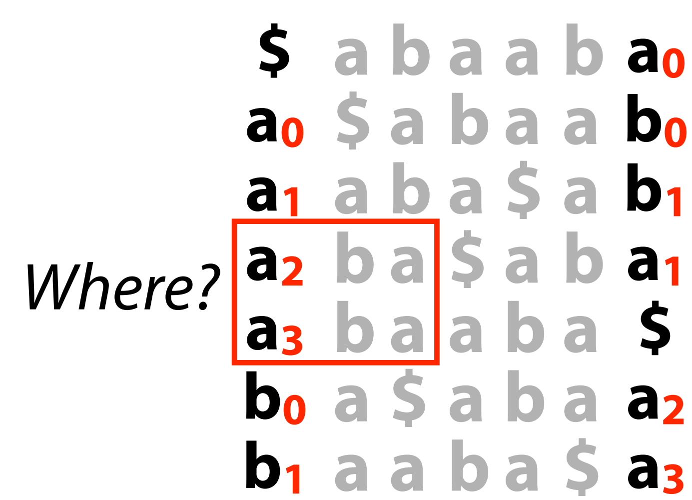


(2) Storing ranks takes too much space

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*:



FM Index: fast rank calculations

Is there an fast 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

FM Index: fast rank calculations

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

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

FM Index: fast rank calculations

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

Tally

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

FM Index: fast rank calculations

F	*L*	*Tally* **a**	*Tally* **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

0 **b**s up to & including this row (row 1)

2 **b**s up to & including this row (row 5)

So $\mathbf{b}_0$ and $\mathbf{b}_1$ must be in there! (rows 2–5 of *L*)

FM Index: fast rank calculations

F	*L*	*Tally* **a**	*Tally* **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

2 **a**'s up to & including this row

4 **a**'s up to & including this row

So $\mathbf{a}_2$ and $\mathbf{a}_3$ must be in there!

O(1) time; 2 lookups
regardless of range size

FM Index: fast rank calculations

Tally

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

m

$|\Sigma|$

Tally is $m \times |\Sigma|$ integers

Too big!

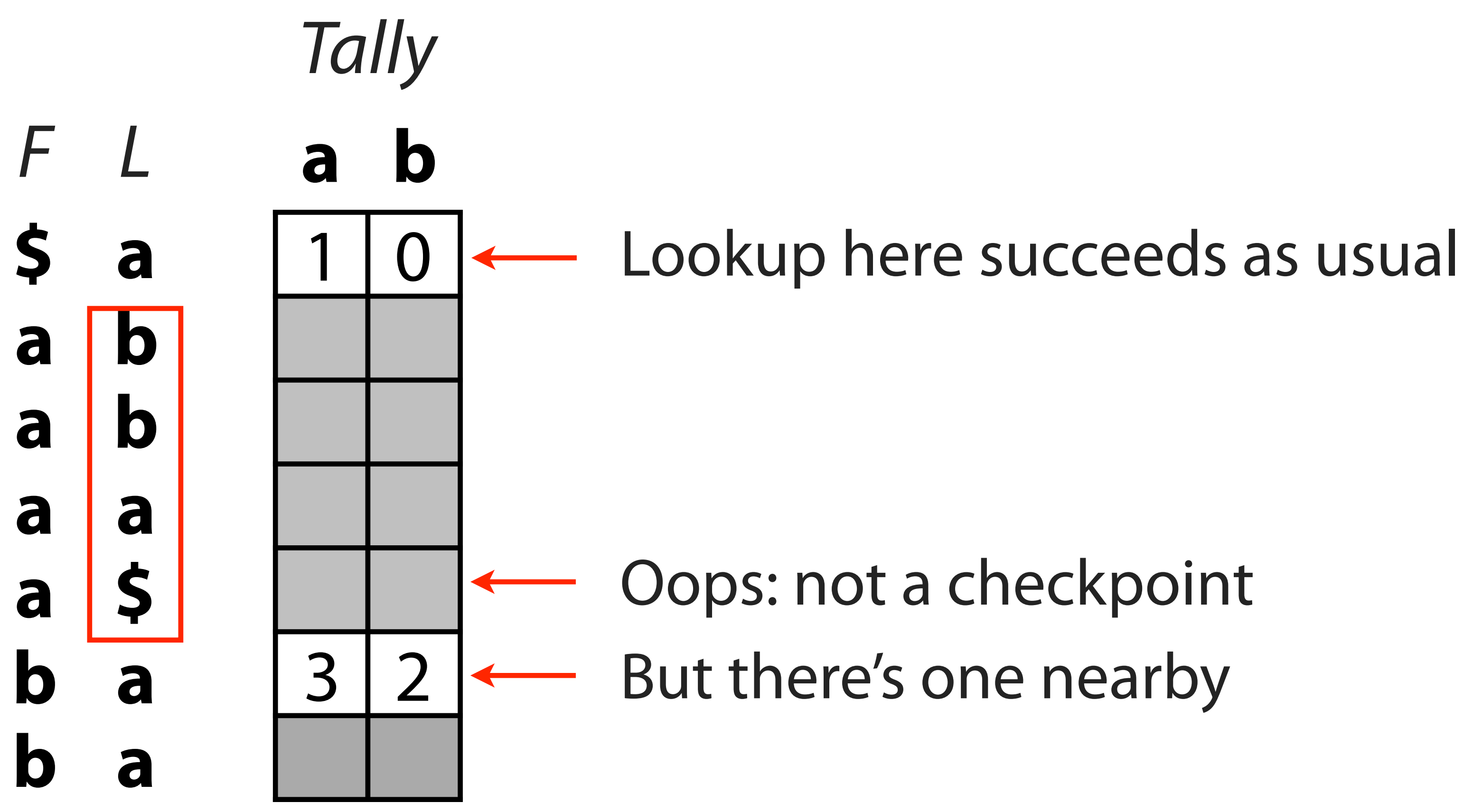
FM Index: fast rank calculations

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

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

FM Index: fast rank calculations

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



To resolve a lookup for a non-checkpoint row, walk to nearest checkpoint. Use tally at that checkpoint, *adjusted for characters we saw along the way*.

FM Index: fast rank calculations

What goes here?

$482 + 2 = 484$

Checkpoint above

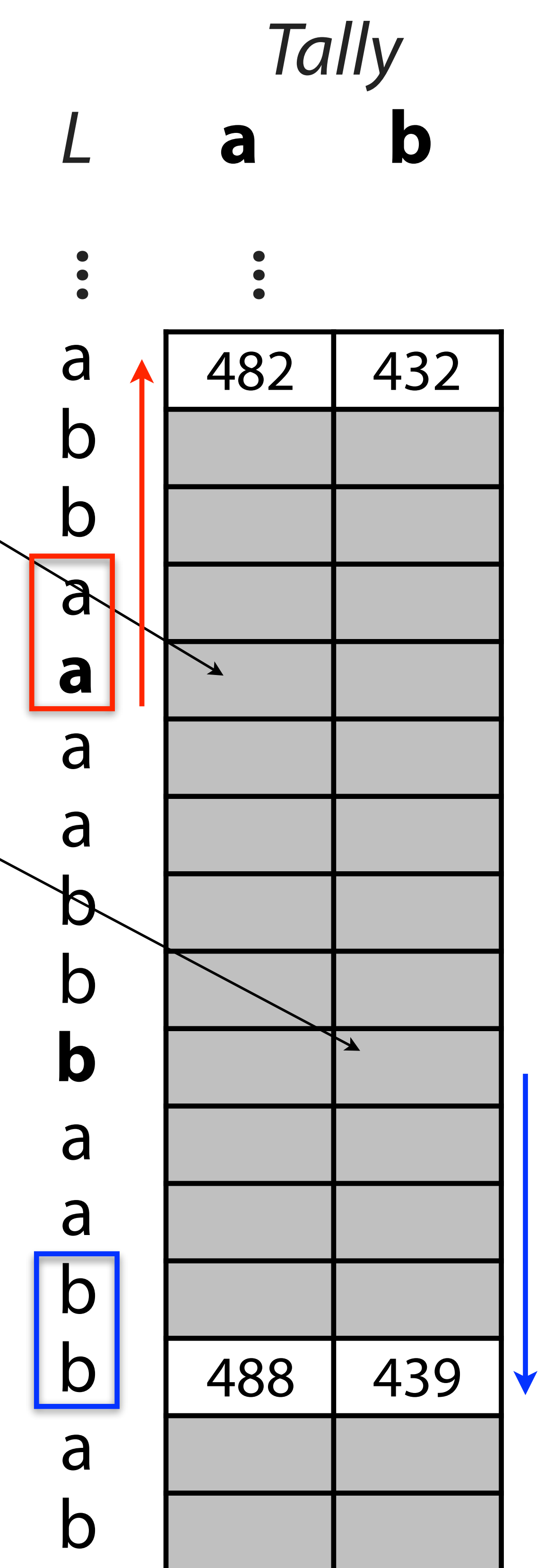
as along the way

What's goes here?

$439 - 2 = 437$

Checkpoint below

bs along the way

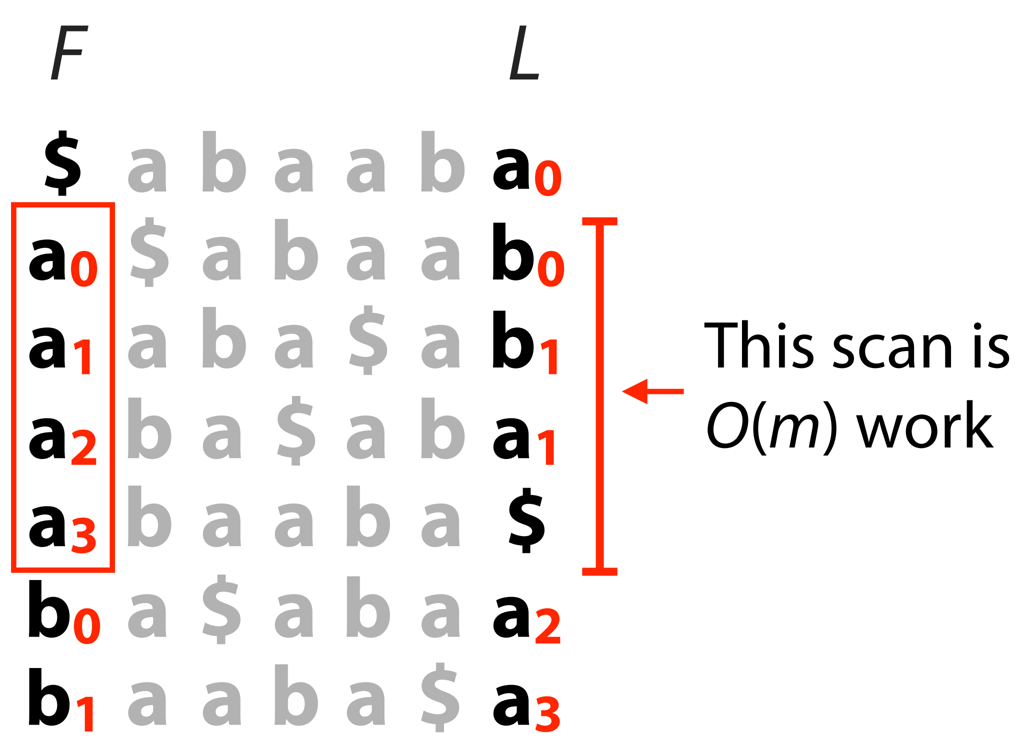


If checkpoints are $O(1)$ distance apart, lookups are $O(1)$

FM Index: a few problems

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

(1)



$O(1)$ with checkpoints

(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
```

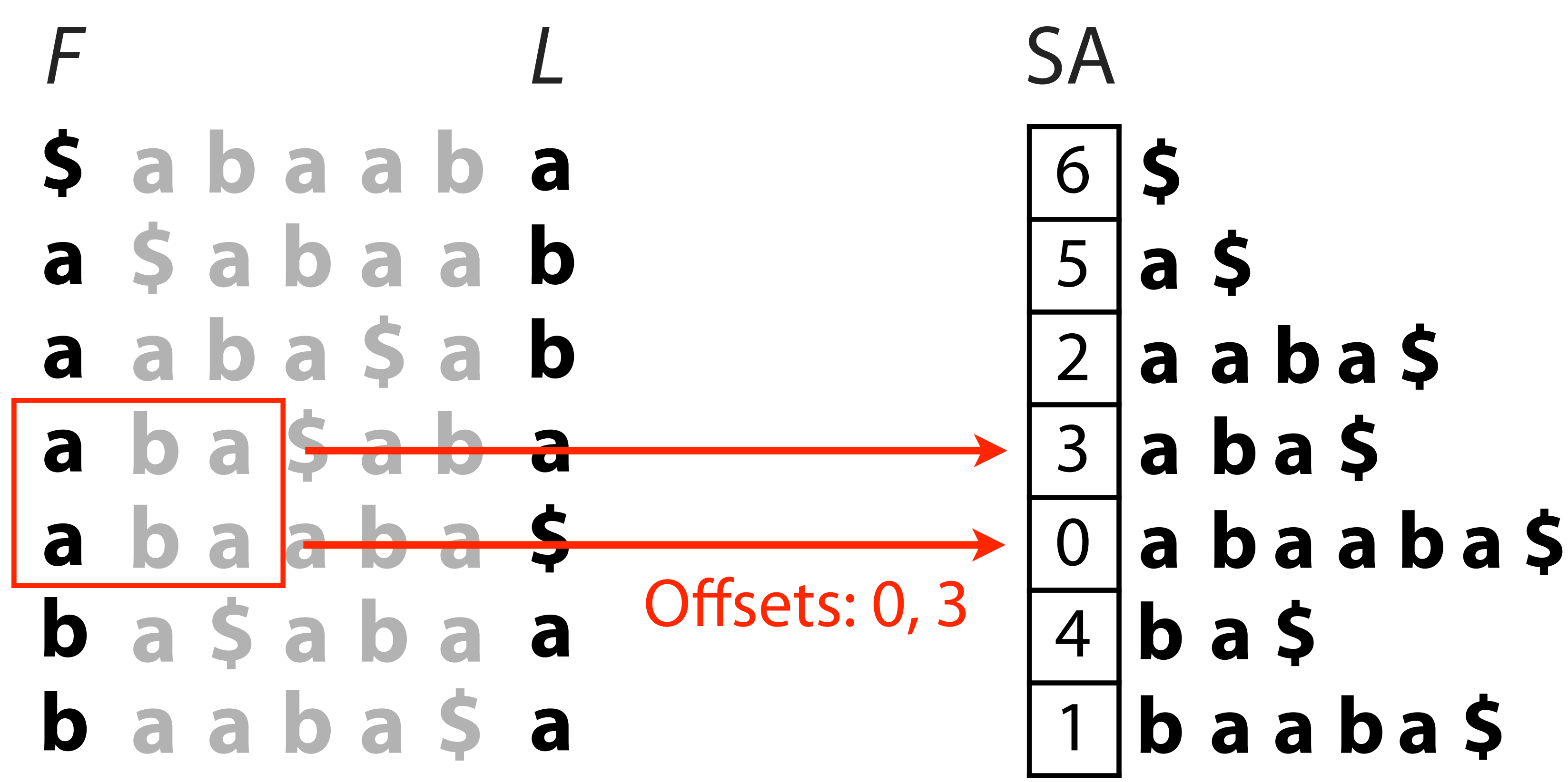
Still $O(m)$ space to store checkpoints, but *we control the constant*

FM Index: a few problems

Not yet solved:

(3) Where are these occurrences in *T*?

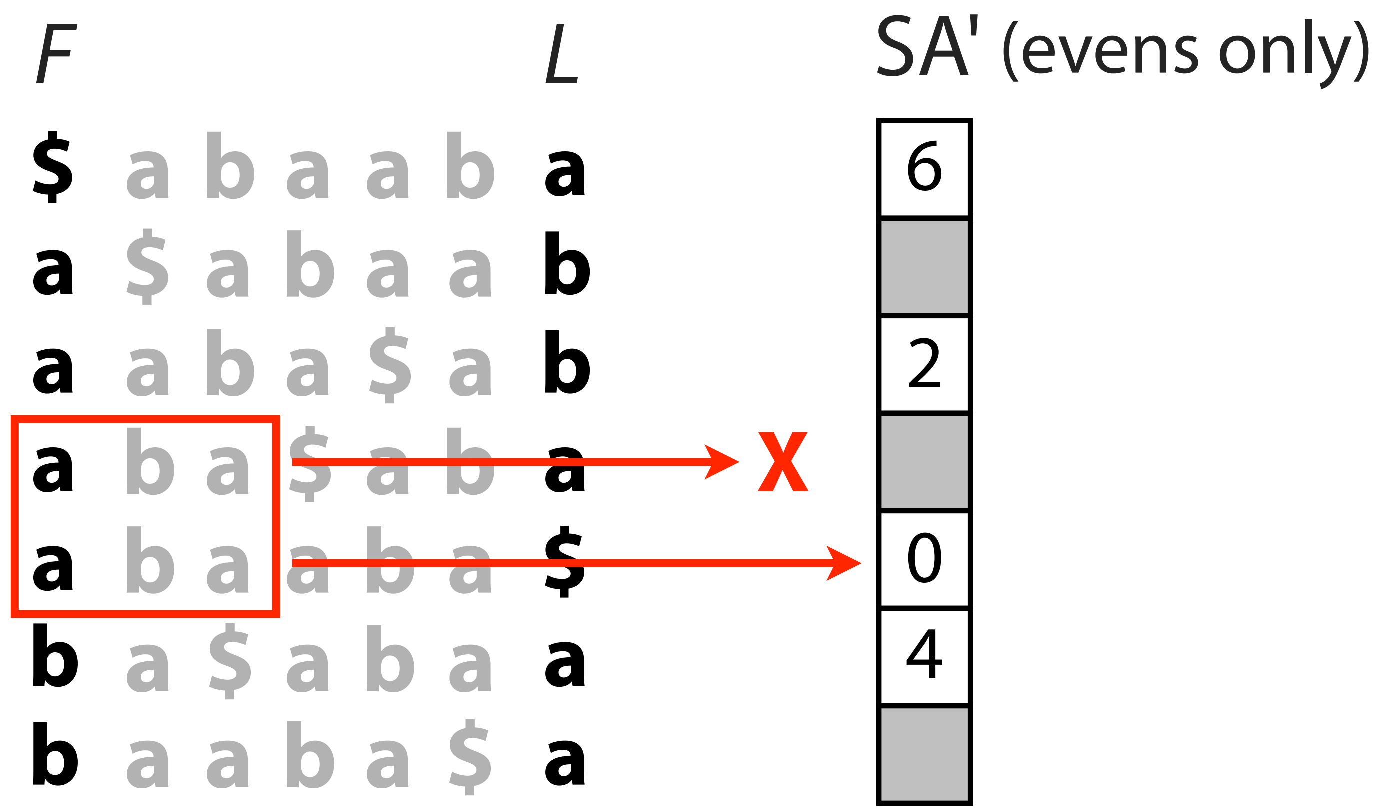
If we had suffix array, we could look up offsets...



...but we don't; we are trying to avoid storing *m* integers

FM Index: resolving offsets

Idea: store some suffix array elements, but not all



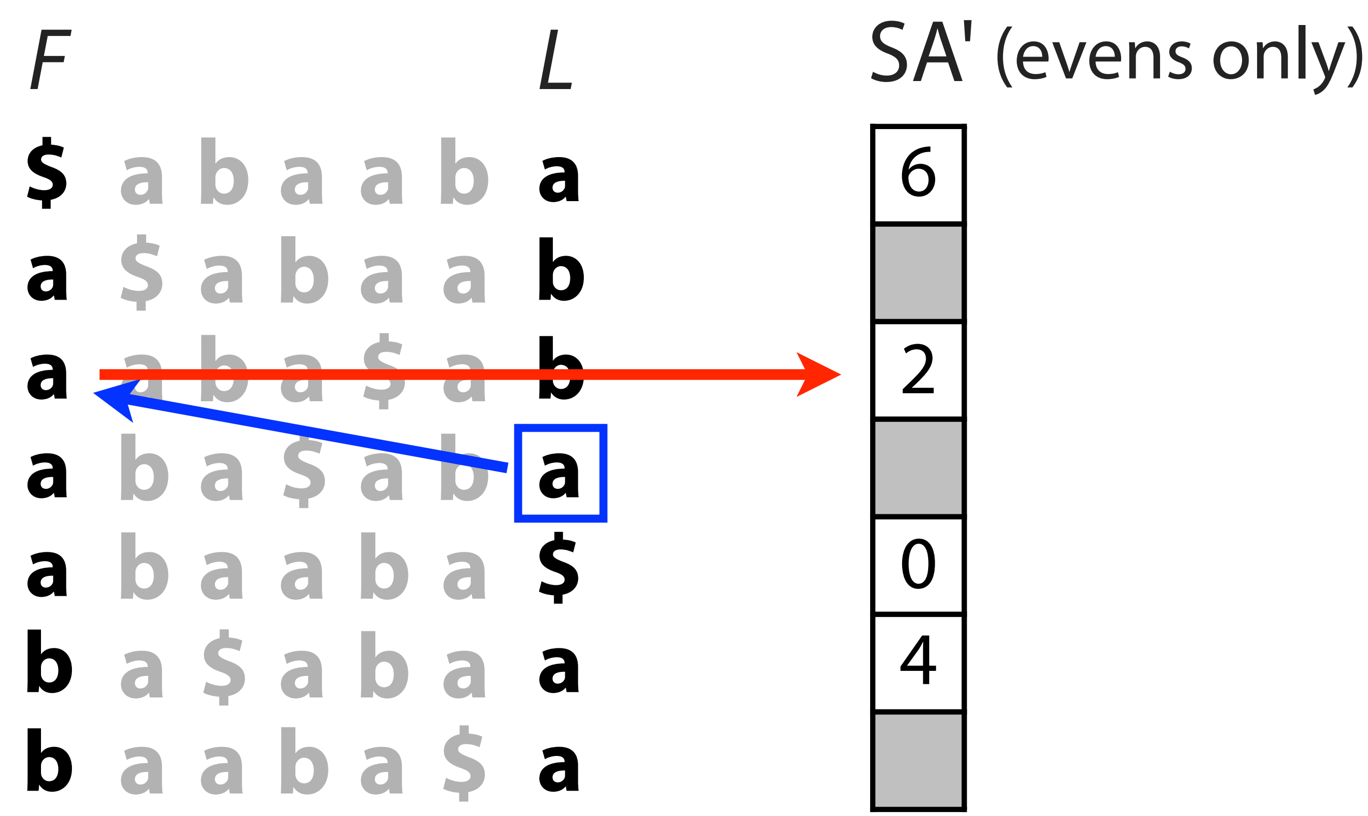
Lookup for row 4 succeeds

Lookup for row 3 fails - SA entry was discarded

FM Index: resolving offsets

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

... "**a**" at the beginning of row 2



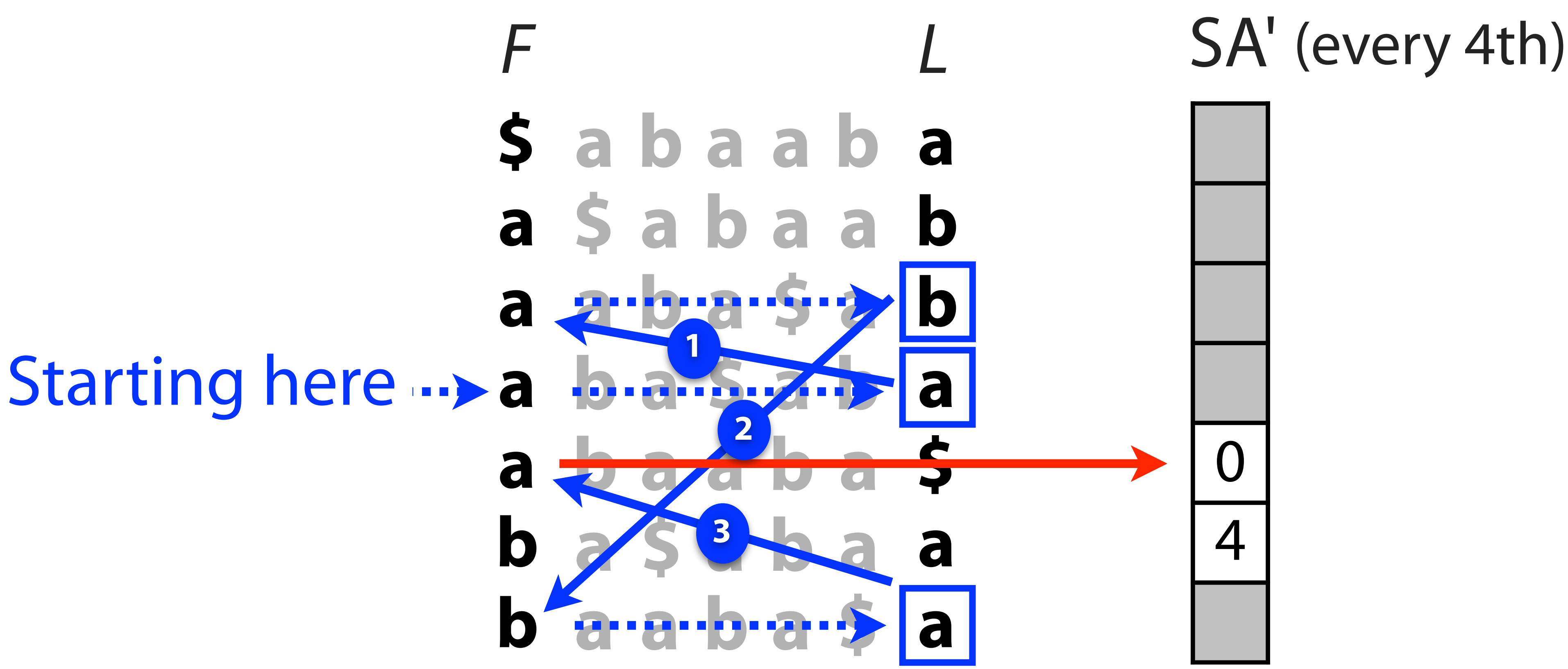
Row 2 of suffix array = 2

Missing value in row 3 = 2 (row 2's SA val) + 1 (# steps to row 2) = **3**

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

FM Index: resolving offsets

Many LF-mapping steps may be required to get to a sampled row:



Missing value = 0 (SA elt at destination) + 3 (# steps to destination) = **3**

FM Index: problems solved

Solved!

At the expense of adding some SA values ($O(m)$ integers) to index

Call this the "SA sample"

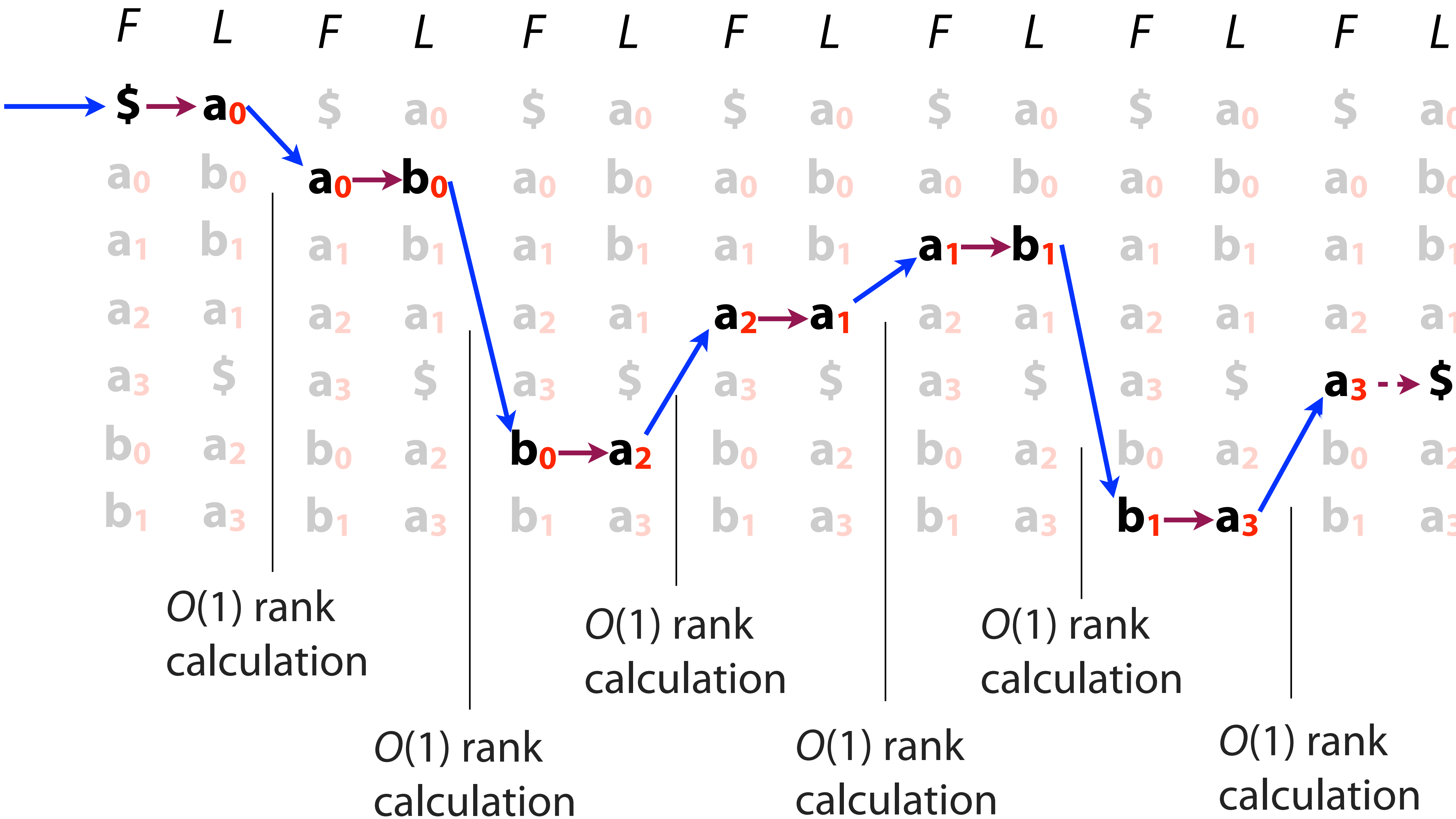
(3) Need a way to find where these occurrences are in *T*:

$	a	b	a	a	b	**a_0**
a_0	$	a	b	a	a	**b_0**
a_1	a	b	a	$	a	**b_1**
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**

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

FM Index

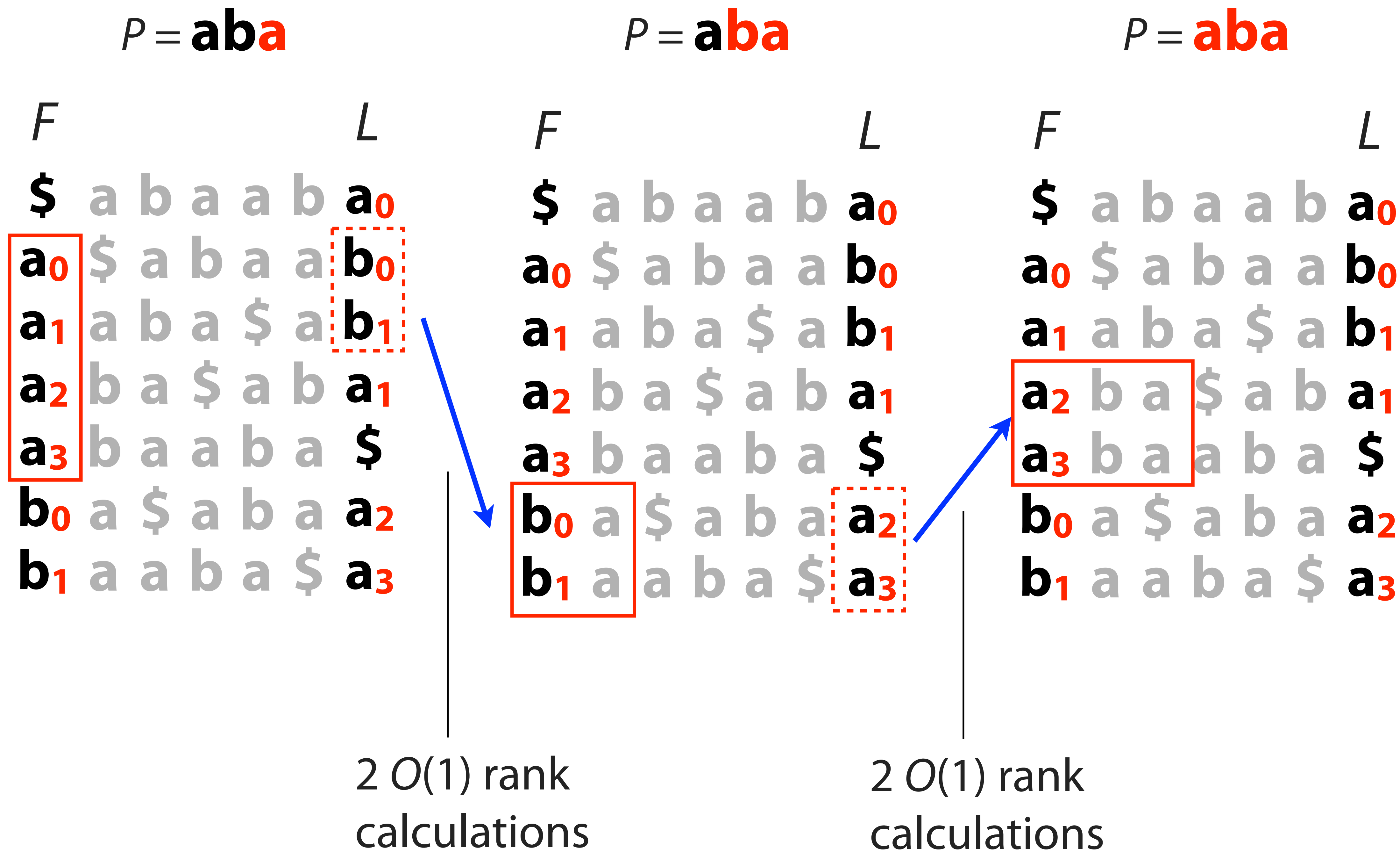
$|T| = m$



Reversing BWT(T) in FM Index is $O(m)$ time

FM Index

$|T| = m, |P| = n$

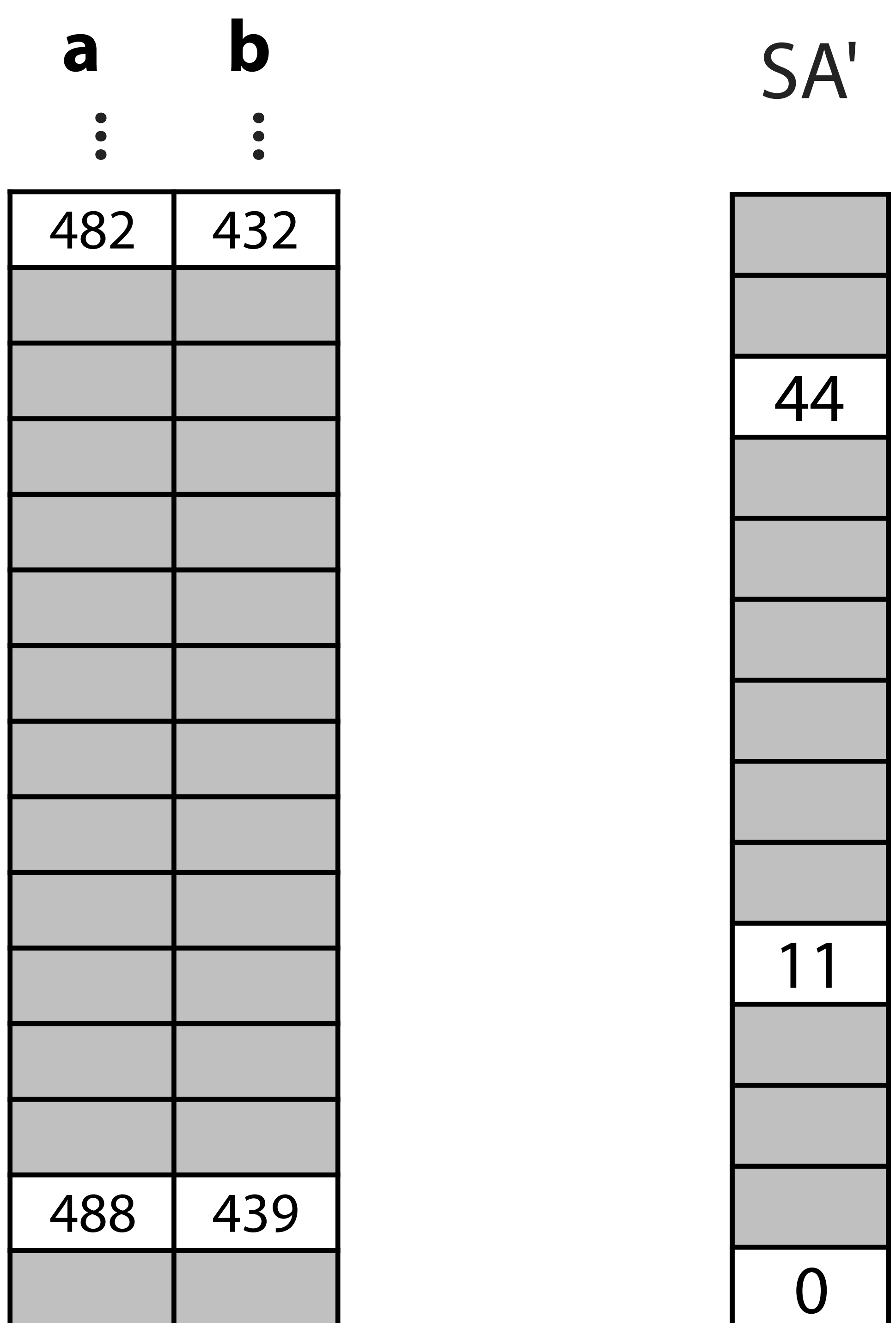


Determining of P occurs in T in FM Index is $O(n)$ time

FM Index

Let ***a*** = fraction of rows we keep

Let ***b*** = fraction of SA elements we keep



FM Index consists of these, plus *L* and *F* columns

Note: suffix tree/array didn't have parameters like ***a*** and ***b***

FM Index

Components of FM Index:

First column (*F*):	~ $\lvert \Sigma \rvert$ integers
Last column (*L*):	*m* characters
SA sample:	$m \cdot a$ integers, *a* is fraction of SA elements kept
Checkpoints:	$m \cdot \lvert \Sigma \rvert \cdot b$ integers, *b* is fraction of tallies kept

For DNA alphabet (2 bits / nt), *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 / 32 = ~ 400 MB
Checkpoints:	3 billion * 4 alphabet chars * 4 bytes / *128* = ~ 400 MB

Total ≈ 1.5 GB

~0.5 bytes per input char

(blue indicates what we can adjust by changing *a* & *b*)

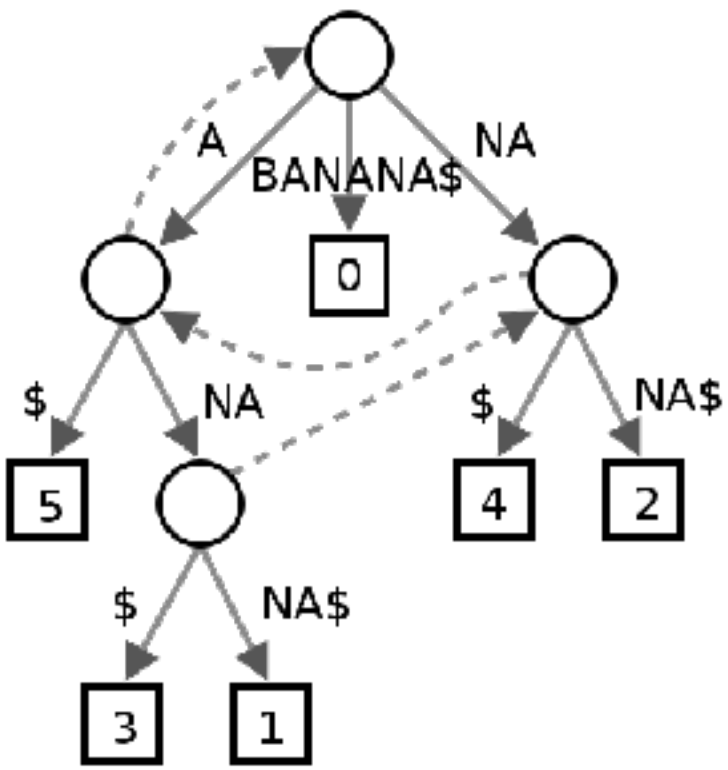
FM Index: small memory footprint

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

FM Index described here is simplified version of what's described in paper

Also discussed in paper: compressing BWT(T) for further savings (and selectively decompression portions of it at query time)

FM Index: small memory footprint



6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

$ BANAN **A**
A $BANA **N**
A NA$BA **N**
A NANA$ **B**
B ANANA **$**
N A$BAN **A**
N ANA$B **A**

Suffix tree	Suffix array	FM Index
≥ 45 GB	≥ 12 GB	**~ 1.5 GB**

Suffix index bounds

	Suffix tree	**Suffix array**	**FM Index**
Time: Does P occur?	$O(n)$	$O(n \log m)$	$O(n)$
Time: Count k occurrences of P	$O(n + k)$	$O(n \log m)$	$O(n)$
Time: Report k locations of P	$O(n + k)$	$O(n \log m + k)$	$O(n + k)$
Space	$O(m)$	$O(m)$	$O(m)$
Needs T?	*yes*	*yes*	*no*
Bytes per input character	>15	~4	~0.5

$m = |T|, n = |P|, k = \#$ occurrences of P in T