

나는 왜 루씬을 C++로 재작성 했는가?

Ultimate Lucene의 탄생

발표의 목적

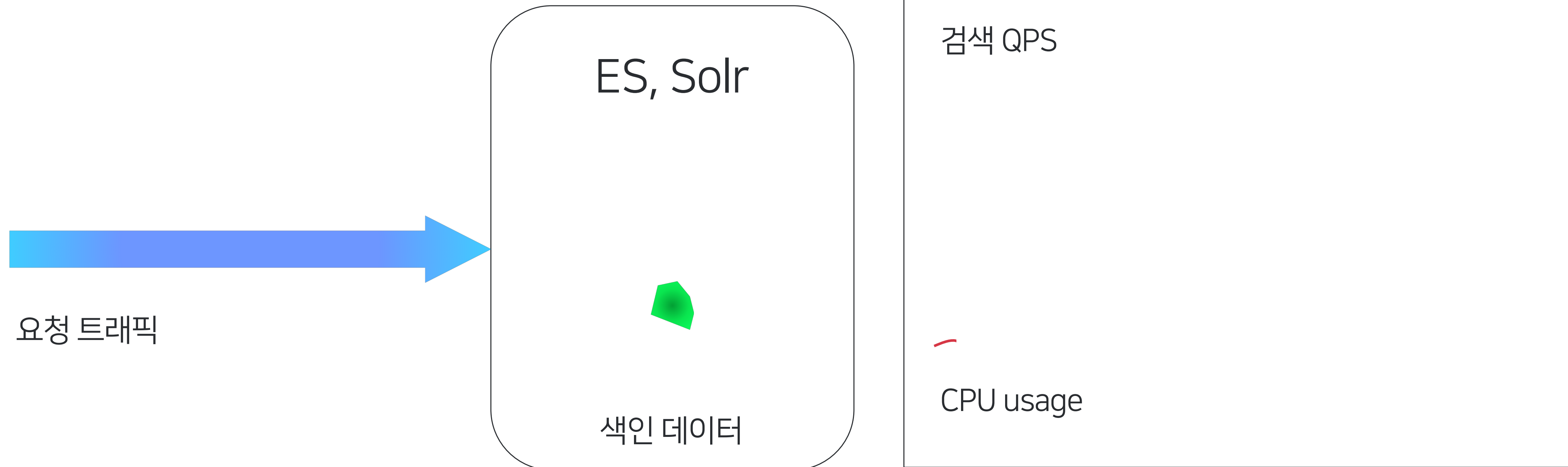
DEVIEW
2019

1. 왜 시작하였는가?
2. 어떻게 하였는가?
3. 얼마나 좋아졌는가?

1. ES, Solr에게 아쉬웠던 것들

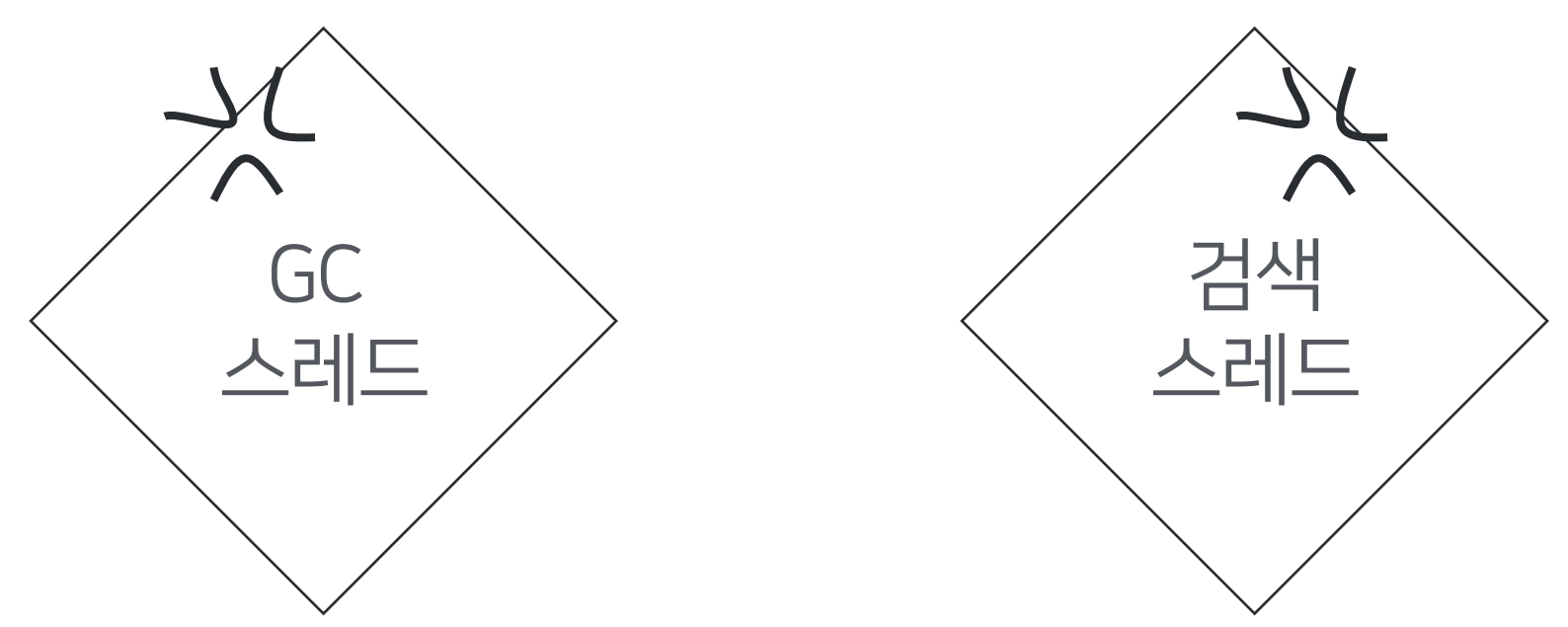
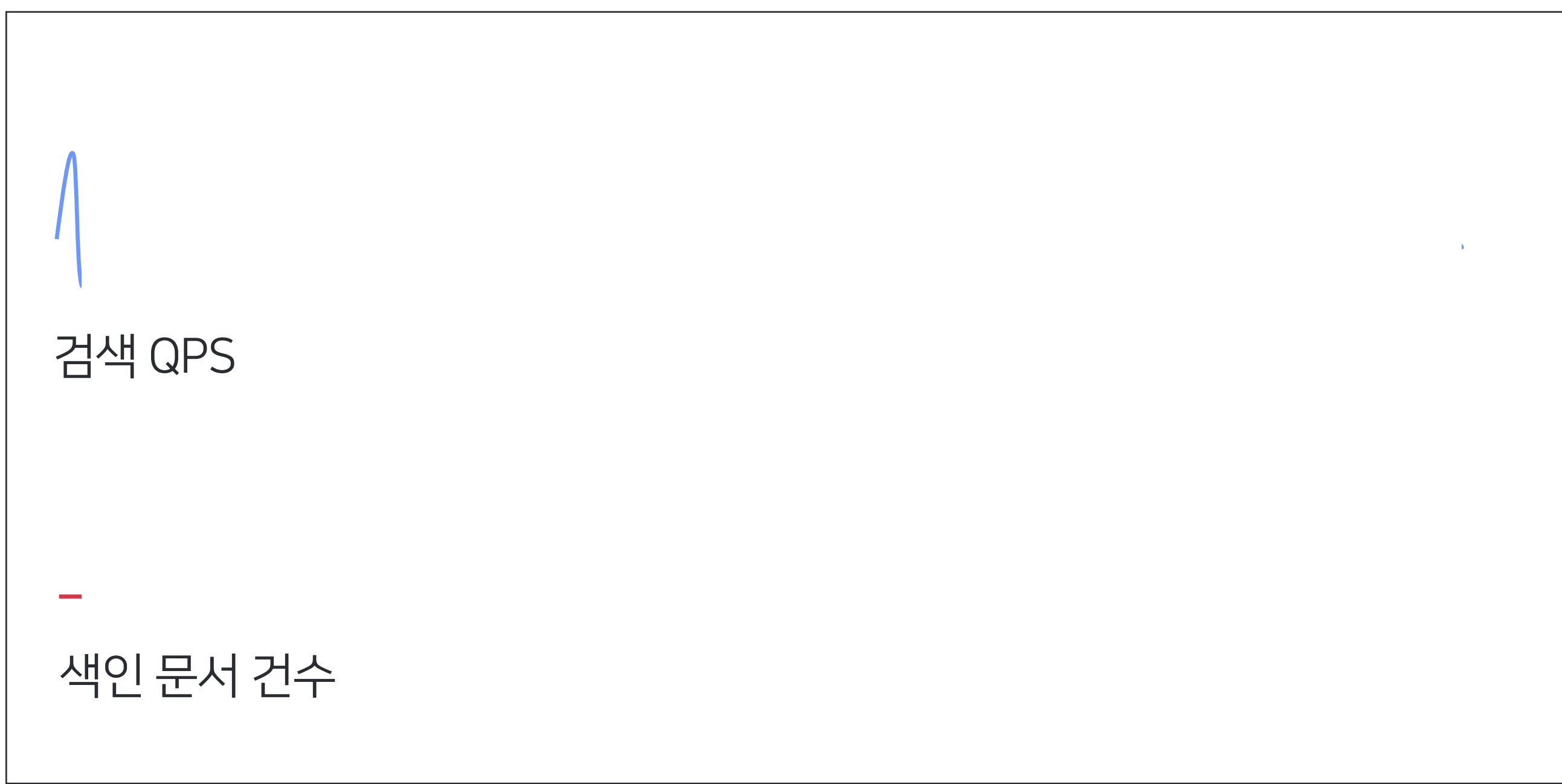
1.1 CPU 버닝

데이터, 트래픽이 시간에 따라 변화한다.
예상하지 못한 GC로 CPU 버닝



1.2 인덱싱과 검색의 분리

인덱싱으로 인해 검색 QPS가 영향을 받는다.



GC 스레드와 검색 스레드가 경쟁한다.

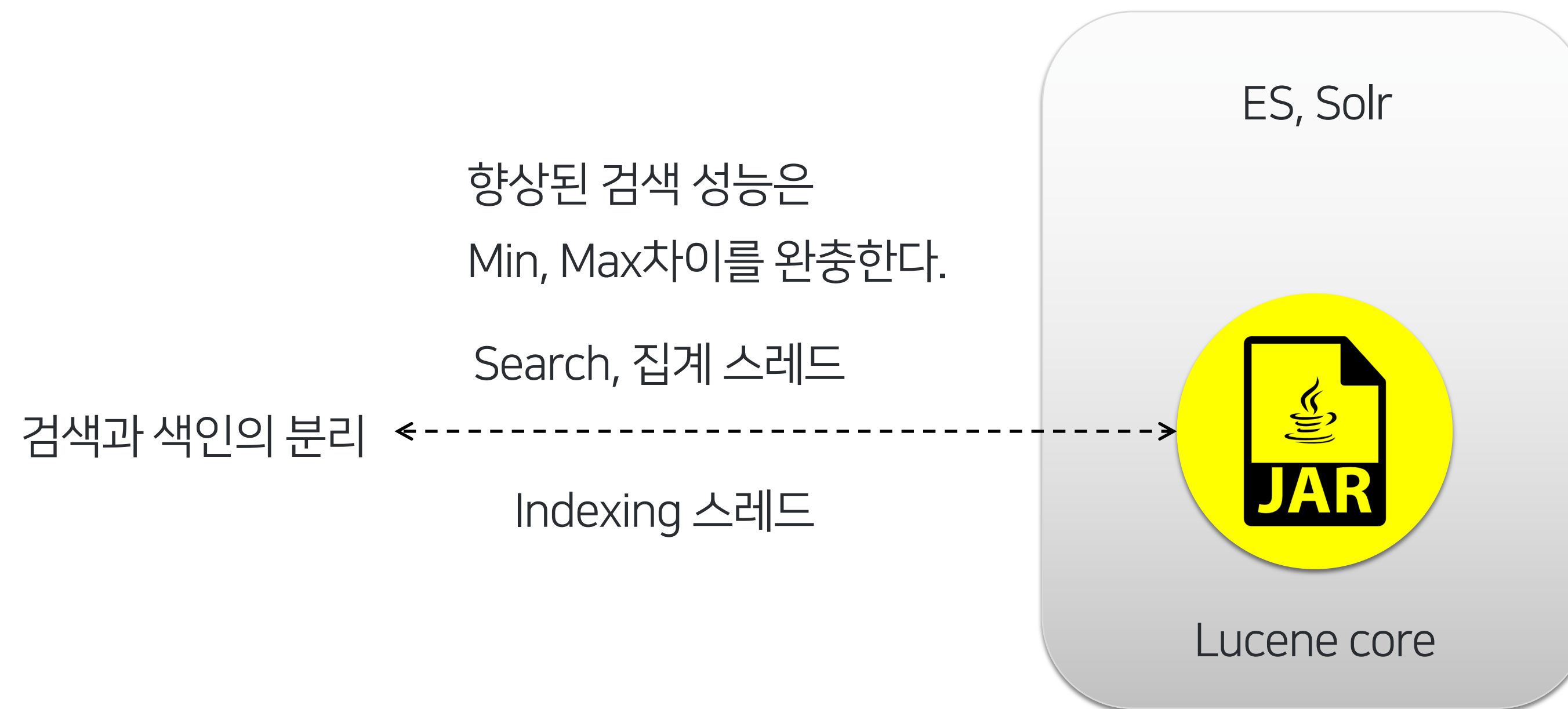
1.3 그 외

1. 단 하나의 Heavy한 집계쿼리로 클러스터가 죽을 수 있다
2. Restart의 늪
3. 한 머신, 2대의 Solr
4. 왜 이렇게 고통받아야 하는가?

2. 문제 정의, Idea

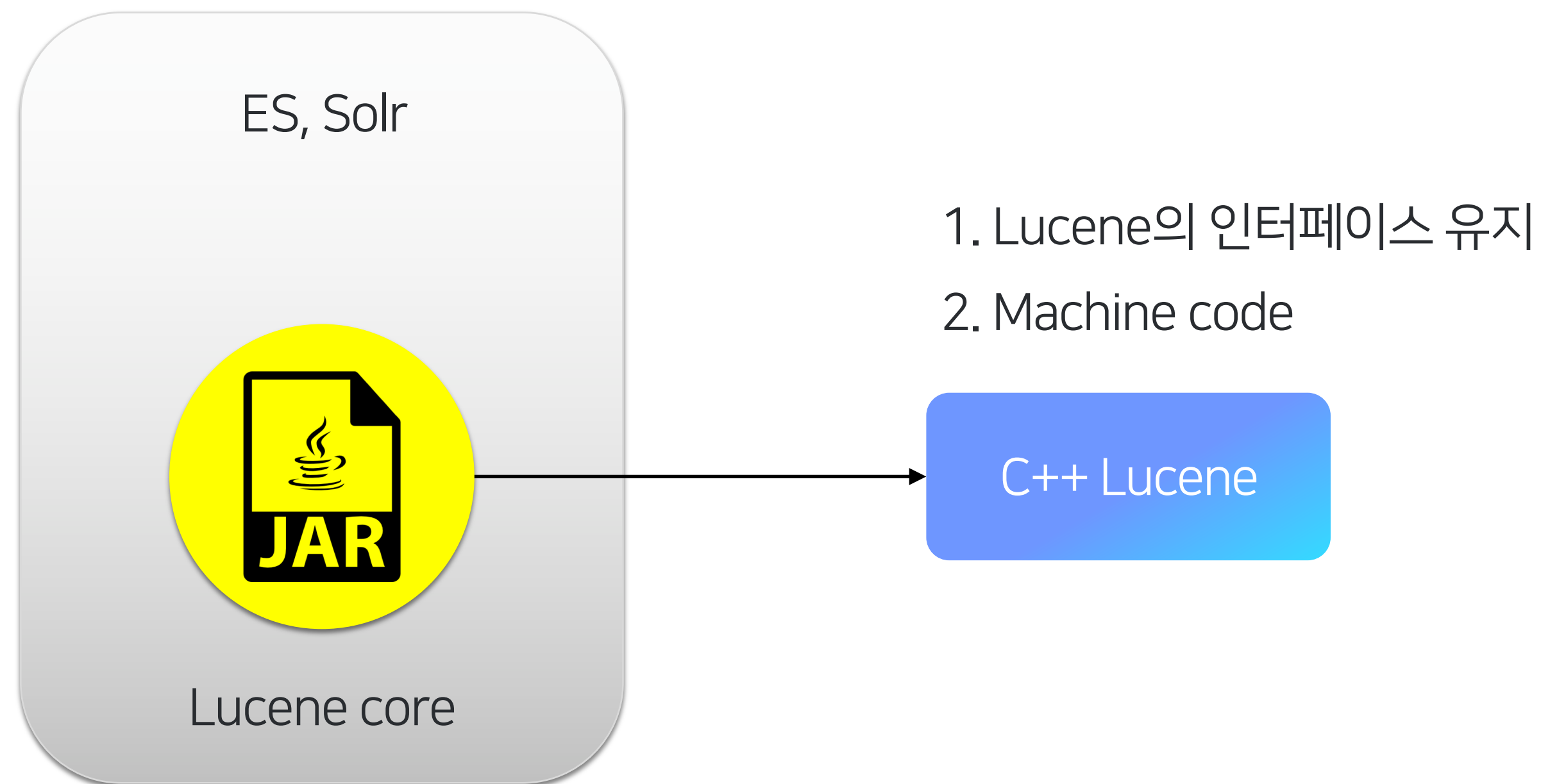
2.1 문제 정의

대량의 트래픽 동시 검색 + 색인이 힘들다. 안되는 것은 아니지만.
이상적으로는 **GC가 없고 검색 성능이 더 빠르다면**
더 이상 동료들이 고통 받는 일은 없을 것이다.



2.2 어쩌면 진부한 아이디어

Lucene을 C++로 더 좋게 만들면 본질적인 문제가 해결되지 않을까?



2.3 실제 반응들

20만 라인을? AI나 해라.
요새 코세라 강의 좋더라
연봉 2억 기사 못봤어?

2.3 실제 반응들

두용아, 이제는 클라우드 시대야
성능보단 확장성의 시대지

2.3 실제 반응들

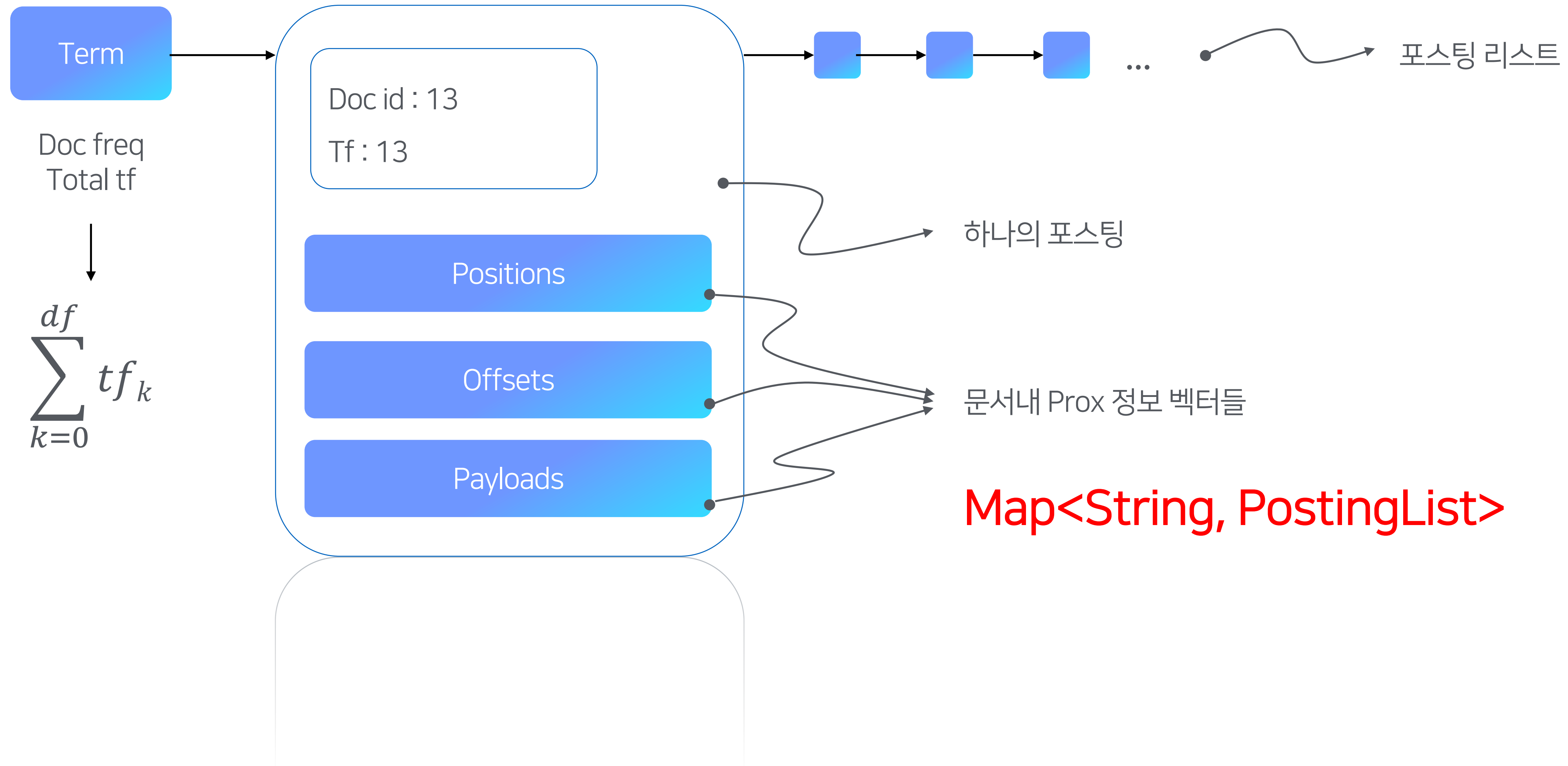
글씨 안될걸?
똑같은 역색인인데?
Java C++의 문제가 아니야.

2.4 Ultimate Lucene

1. Lucene의 방식은 따르되,
더 잘 할 수 있다면 굳이 고집하지 않는다.
2. Lucene의 Super set
3. Pure C++17
4. No GC overhead at all!
5. 평균 2배 성능 우위

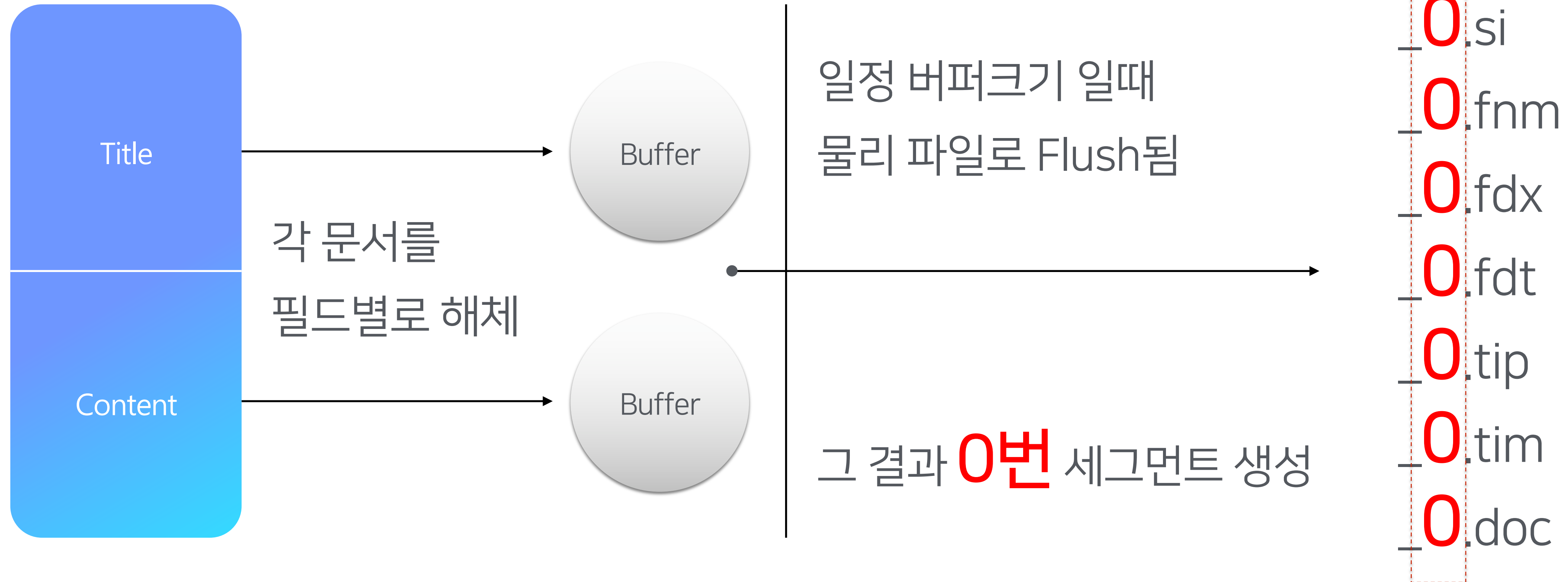
3. Lucene의 인덱싱

3.1 Lucene의 역색인

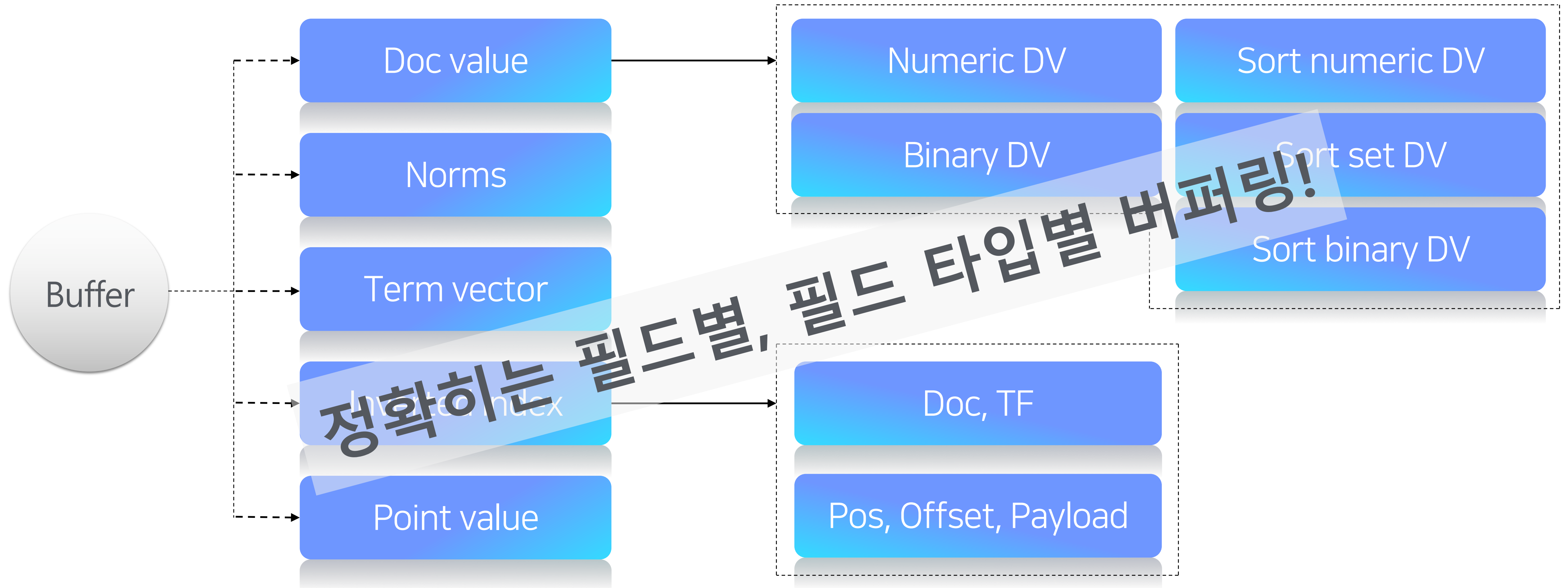


3.2 색인 플로우, 큰 그림

Doc id = 13



3.3 버퍼링, 인덱스 타입 관점에서



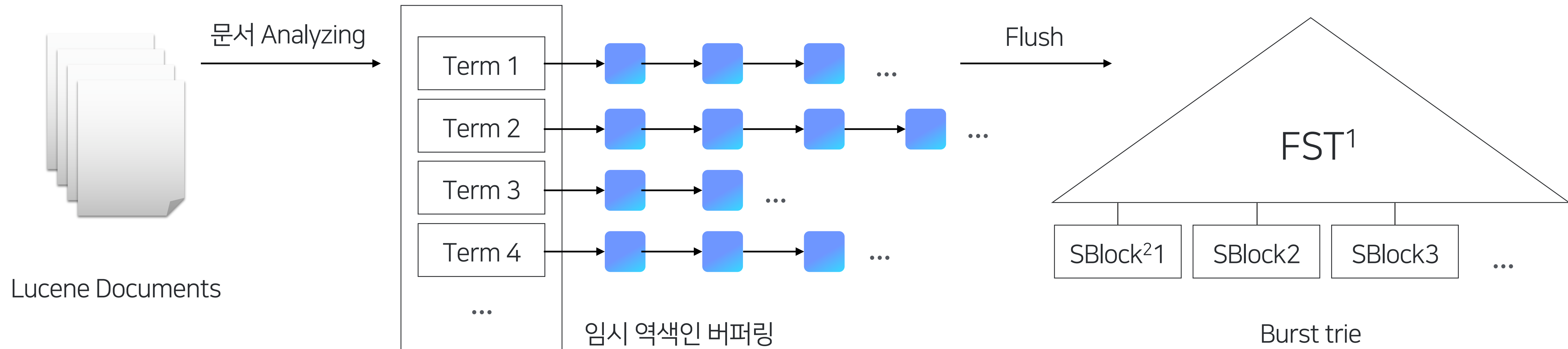
3.4 버퍼링 Lucene 소스 코드

<https://github.com/apache/lucene-solr/blob/releases/lucene-solr/7.6.0/lucene/core/src/java/org/apache/lucene/index/DefaultIndexingChain.java#L685>

XXWriter라는 이름과 달리 하는 일은 버퍼링입니다.

3.5 Lucene 인덱싱 플로우

문서들을 버퍼링 한 뒤 Flush 한다.



1. FST = Finite state transducer
2. Sblock = Suffix block

3.6 offset, pos, payload 3형제

ES, Solr는 전세계에서 널리 쓰이는 디팩토 검색엔진입니다.

하지만 그 깊은 내면에는 **Lucene**이 자리잡고 있습니다.

Lucene이 색인하는 방식이 곧 ES, Solr가 색인하는 방식이고
검색도 그렇습니다.

하지만 안타깝게도 **Lucene**이 어떻게 색인, 검색을 하는지에
대해서는 베일에 감춰져 있습니다.

물론 **Lucene** Java doc에는 여러 논문 링크들이 첨부되어
있지만

바쁜 현대 개발자들에게는 ...

Start offset : 127

End offset : 133

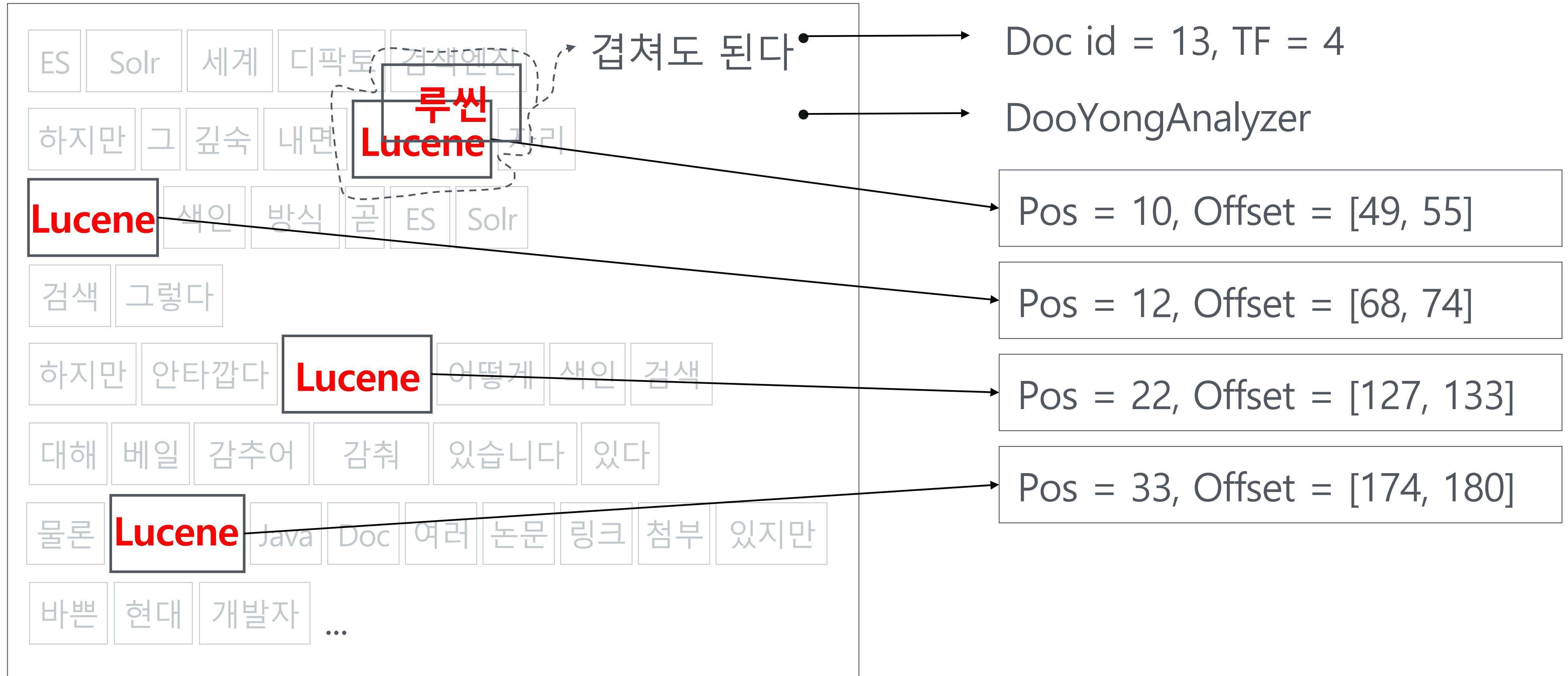
Doc id = 13

DooYongAnalyzer

- Docs
- Docs + TF
- Docs + TF + Pos
- Docs + TF + Pos + Offsets
- Docs + TF + Pos + Offsets + Payload

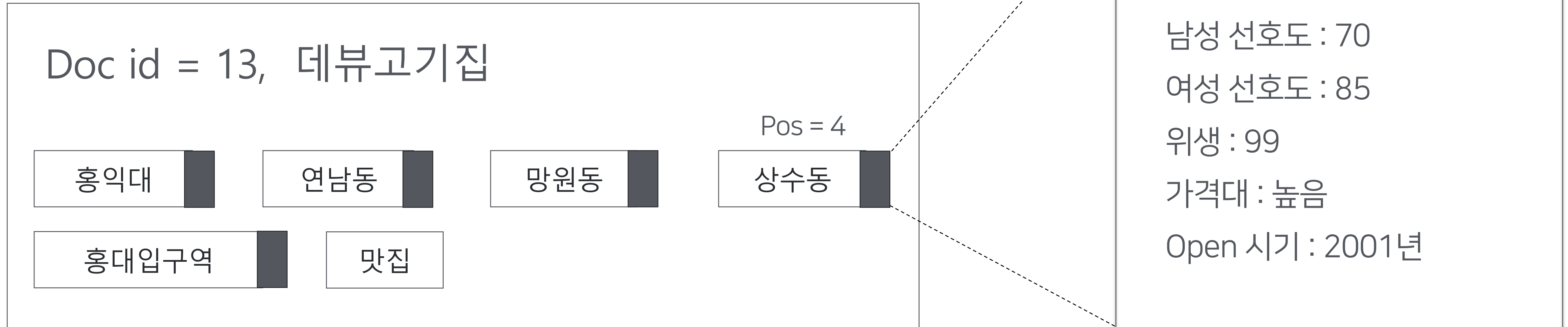
색인 옵션

3.7 offset, pos, payload 3형제

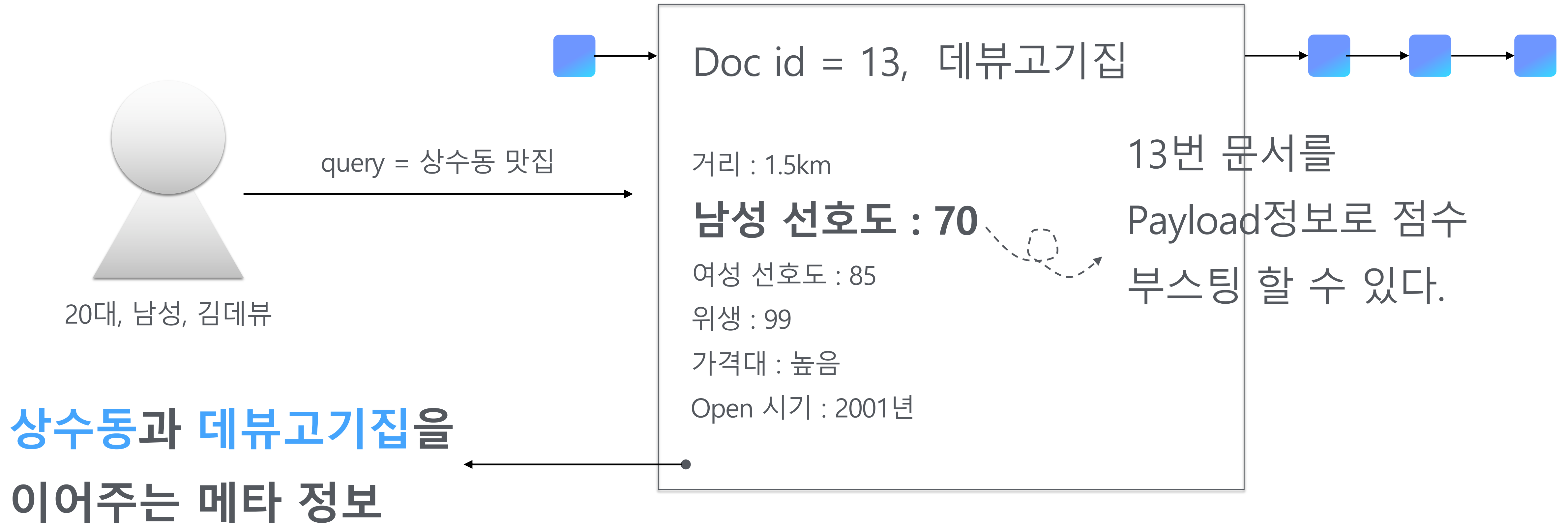


3.8 payload에 관해

개별 텀 포지션 마다 부여 할 수 있는 Byte[]
랭킹에서 유용하게 사용 될 수 있음

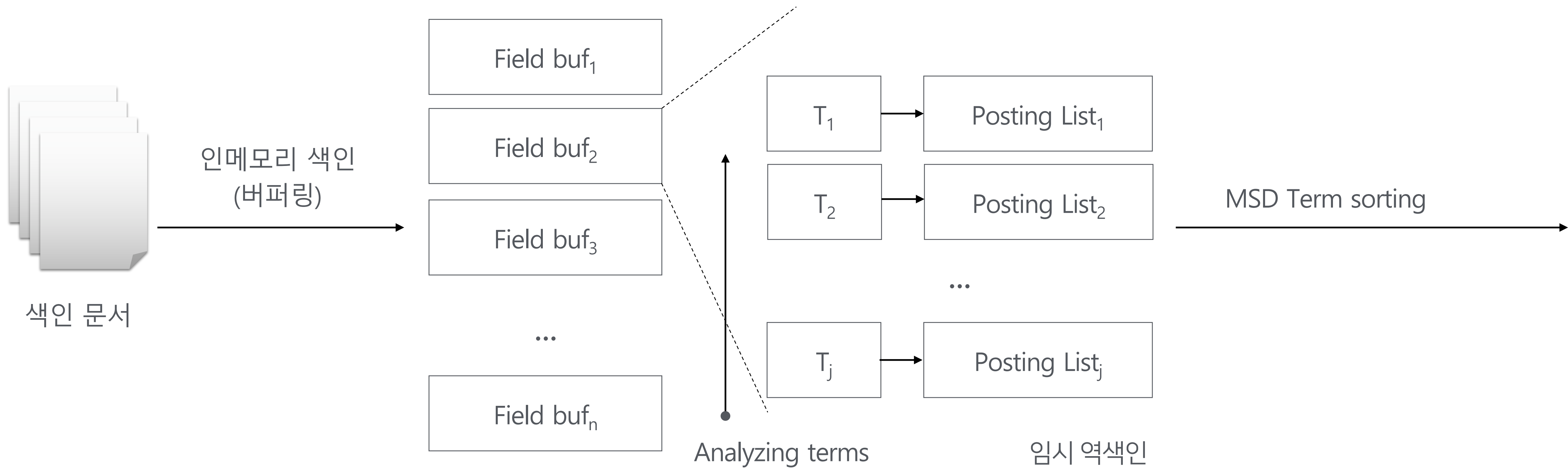


3.8 payload에 관해

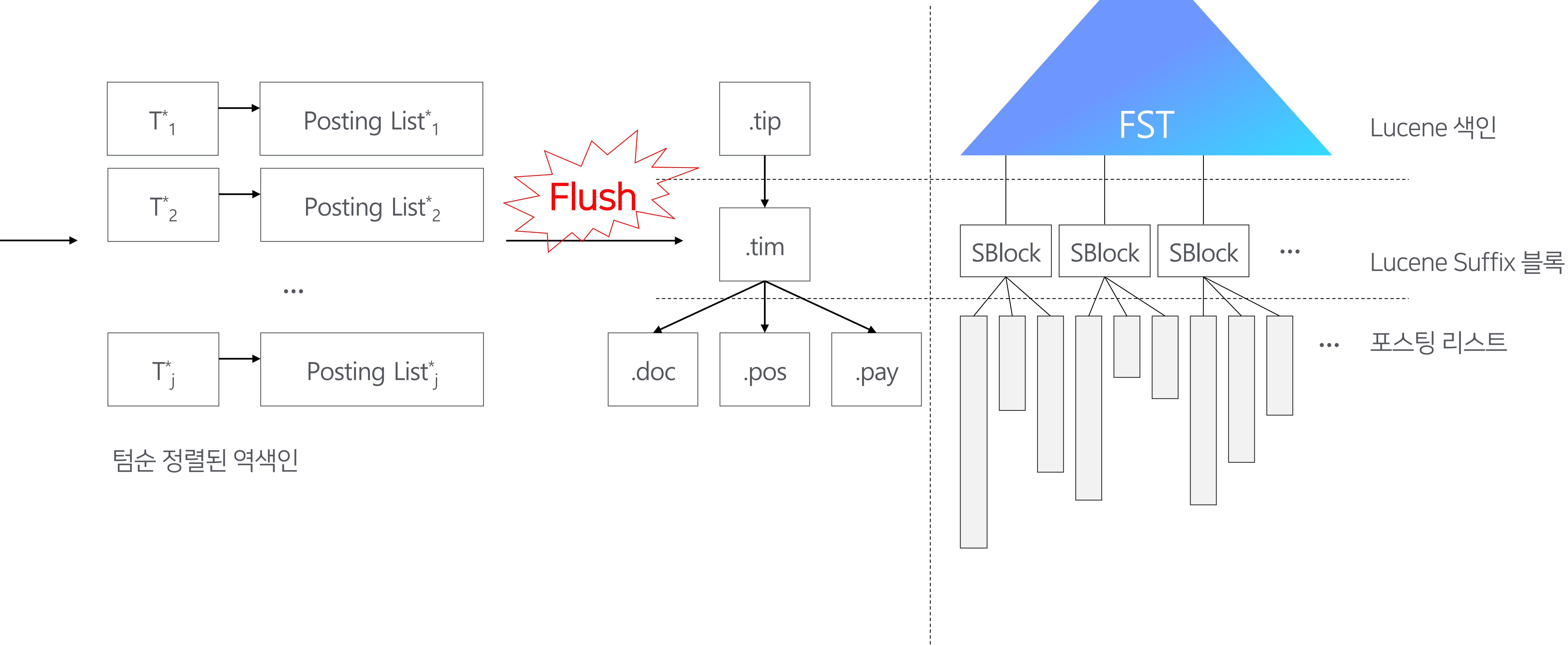


4. 역색인 버퍼링

4.1 전체 색인 플로우

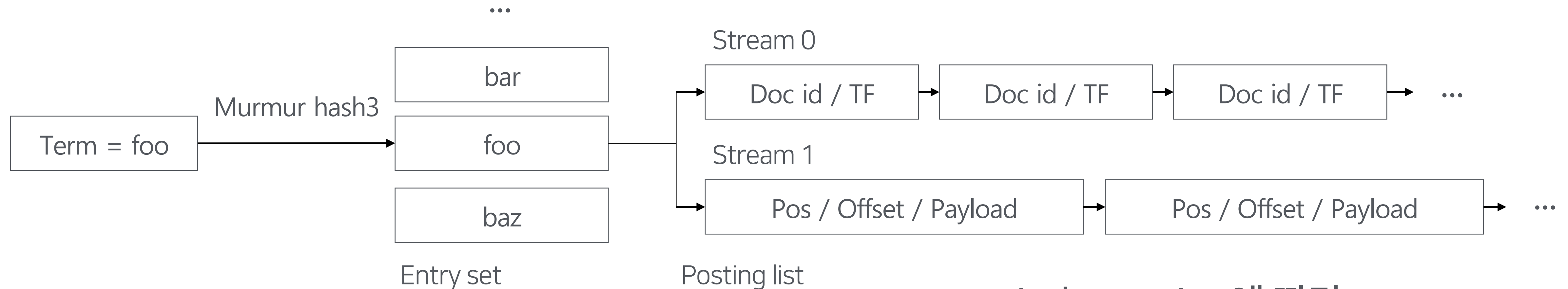


4.1 전체 색인 플로우



4.2 Lucene 역색인 버퍼링

Map<String, PostingList>



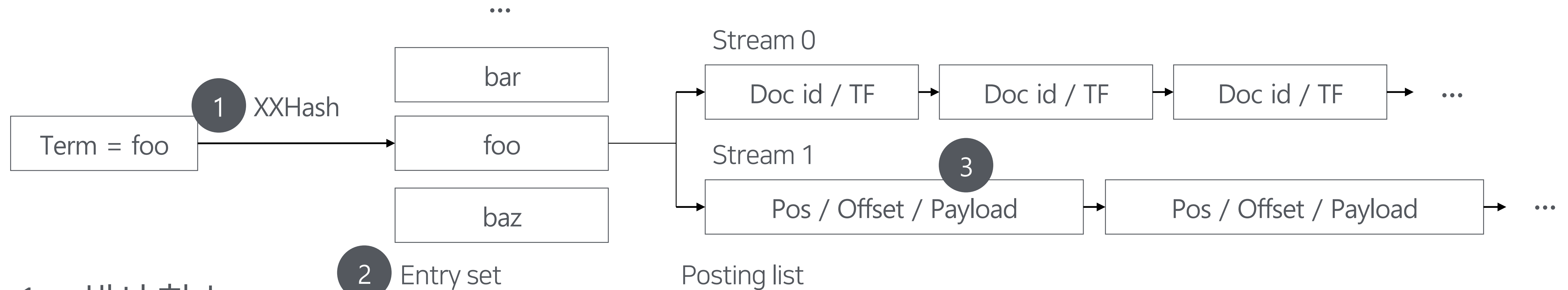
Index option에 따라
Stream의 개수가 결정된다.

4.3 Lucene 역색인 버퍼링

1. Find hash : <https://github.com/apache/lucene-solr/blob/releases/lucene-solr/7.6.0/lucene/core/src/java/org/apache/lucene/util/BytesRefHash.java#L257>
2. Murmur hash3 : <https://github.com/apache/lucene-solr/blob/releases/lucene-solr/7.6.0/lucene/core/src/java/org/apache/lucene/util/StringHelper.java#L149>

4.4 UTLC 역색인 버퍼링 개선

Map<String, PostingList>



1. 해시 함수
2. 해시 테이블 디자인
3. Payload 벌크 쓰기

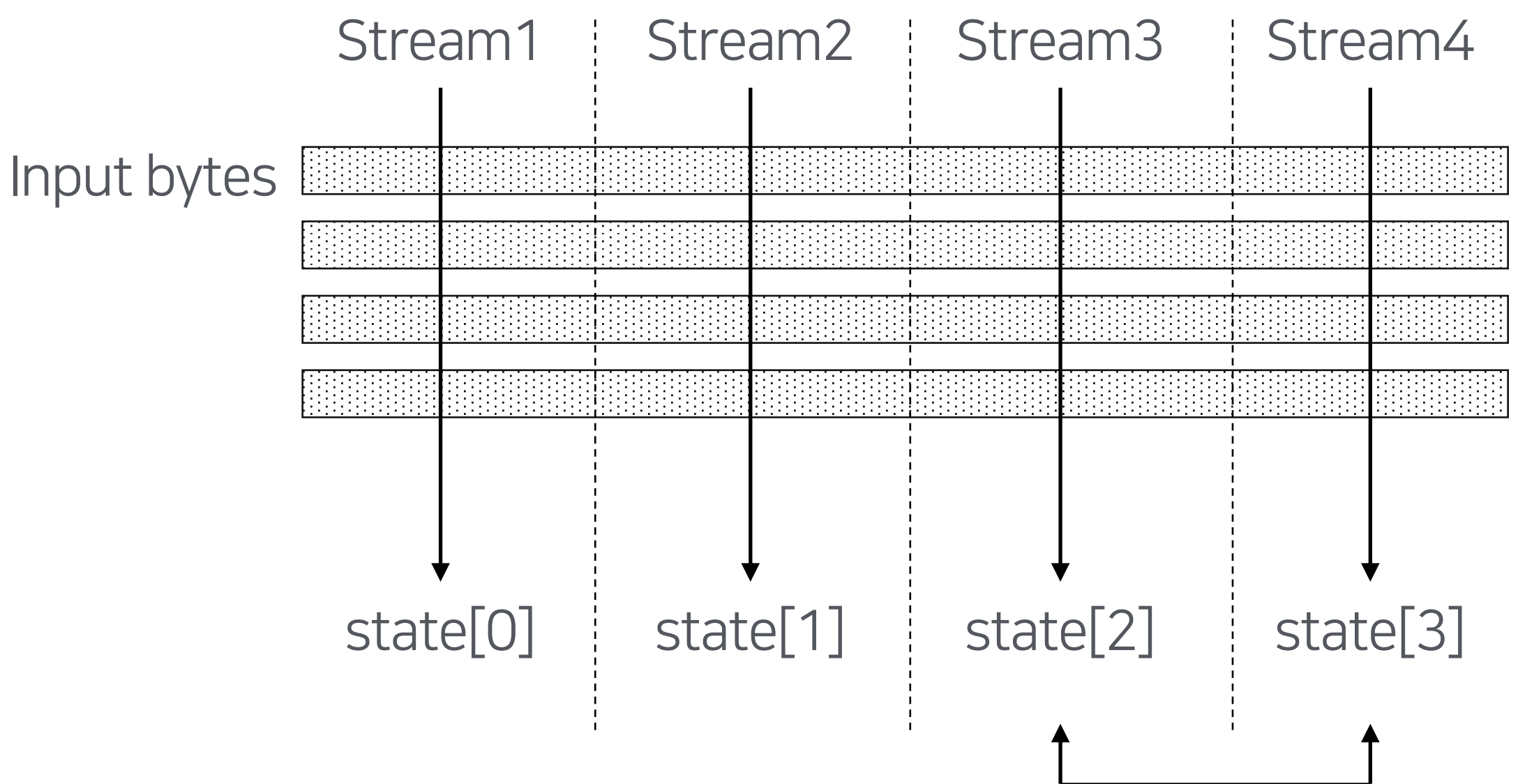
4.5 UTLC 해시 함수, XXHash

Benchmarks

The benchmark uses the **Open-Source SMHasher program** by Austin Appleby. The system uses a Core 2 Duo @3.0GHz.

Name	Speed	Quality	Author
xxHash	5.4 GB/s	10	Y.C.
MurmurHash 3a	2.7 GB/s	10	Austin Appleby
SBox	1.4 GB/s	10	Bret Mulvey

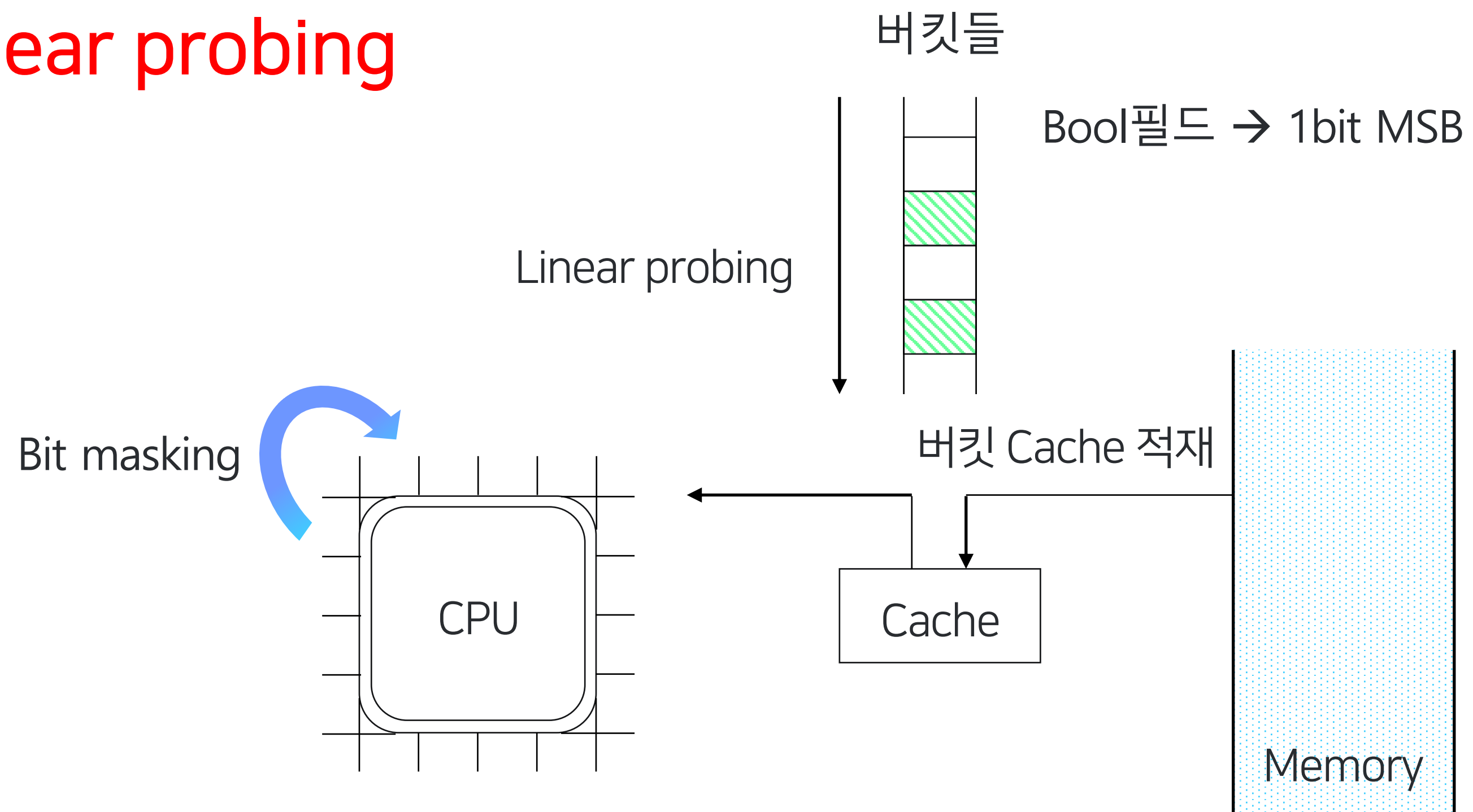
2배



Block들을 한꺼번에,
즉 **Instruction code parallelism.**

4.6 UTLC Hash table 디자인

1. Cache preloading + Linear probing
2. Byte pool 메모리 할당
3. % 대신 &, 9 cycle 절약!
4. 버킷 사이즈 최소화



4.7 UTLC, Linear probing

Index/Buffering.h



```
do {
    Entry& entry(table[pos]); 1 Cache preloading + Linear probing
    __builtin_prefetch(&entry, 0, 1);

    if (entry.hash & ItsEmpty) { 2 1 bit flag, ItsEmpty = (1UL << 63)
        // Entry was not found. Insert a new entry
        char* allocated_bytes(term_pool.Allocate(length)); 3 메모리는 Pool에서
        std::memcpy(allocated_bytes, bytes, length);
        const int32_t term_id(num_entries++);

        table[pos] = Entry(hash, allocated_bytes, length, term_id);
        index.emplace_back(pos);

        RehashIf();
    }
```

```
        return {term_id, true};
    } else if (hash == entry.hash &&
               length == entry.length &&
               std::memcmp(bytes, entry.bytes, length) == 0) {
        // Found one
        Due to the packed binary representation of entry,
        // we have to manually reassign it's value to a tmp variable
        const int32_t term_id(entry.term_id);
        return {term_id, false};
    } // End if

    pos = ((pos + 1) & table_mask); 4 % 보단 &
    } while (true); N % 2n = N & (2n - 1)
```


4.8 Lucene의 Payload 쓰기

```
public void writeBytes(int stream, byte[] b, int offset, int len) {  
    // TODO: optimize  
    final int end = offset + len;  
    for(int i=offset; i<end; i++)  
        writeByte(stream, b[i]);  
}
```

Payload와 같은 byte[]에 대해 바이트를 하나씩 옮기고 있다.
최적화 하는 게 TODO. 한번 해보자.

4.9 UTLC의 Payload 벌크 쓰기

```
while (left > 0) {
    if (upto == end_upto) {
        assert(target_buffer[upto] != 0);
        // End of slice; allocate a new one
        auto [new_upto, new_end] = AllocSlice(target_buffer, upto);

        upto = new_upto;
        end_upto = new_end;
        target_buffer = buffer;
        int_offsets[stream] = new_upto + byte_offset;
        int_offsets[stream + 1] = new_end + byte_offset;
        // Since we are allocated a new slice, delta has to be updated
        delta = 0;
    } // End if

    const int32_t copy_len(std::min(left, end_upto - upto));

    std::memcpy(target_buffer + upto, data, copy_len);

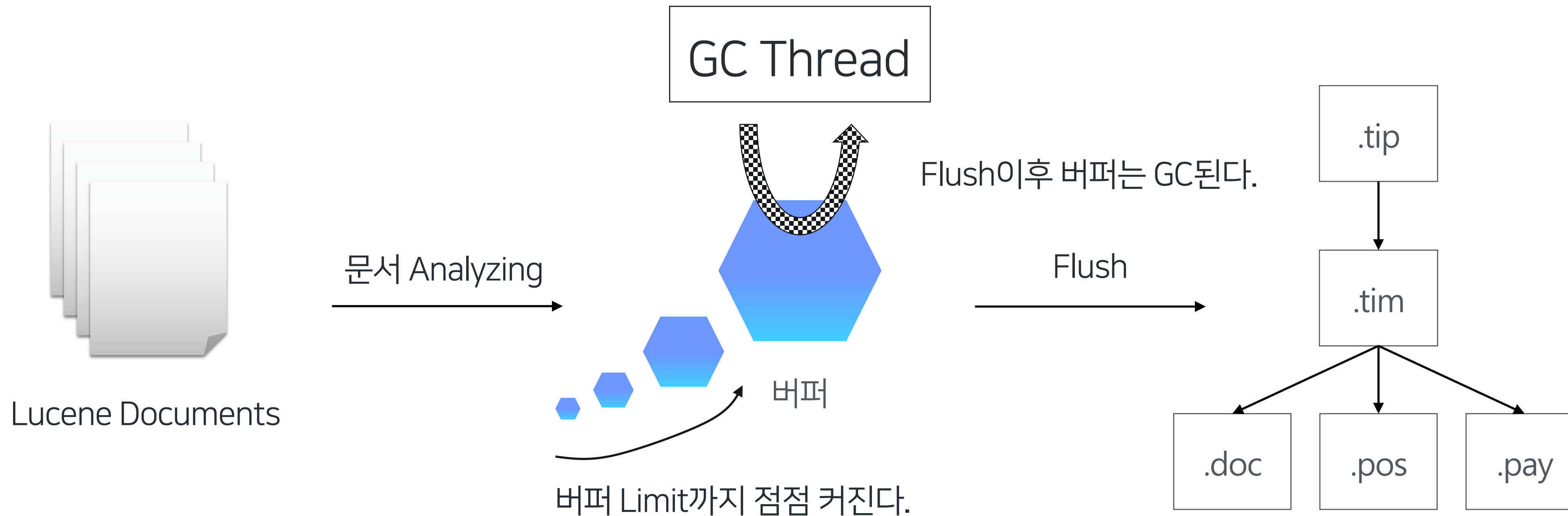
    left -= copy_len;
    upto += copy_len;
    data += copy_len;
    delta += copy_len;
} // End while
```

만약 공간이 부족하다면 Slice 추가

얼마나 써야 하는지 계산 후
`memcpy`로 벌크 복사

4.10 Lucene 버퍼는 1회용

Flush 이후 버퍼는 재사용 하지 않는다.



https://github.com/apache/lucene-solr/blob/branch_7_6/lucene/core/src/java/org/apache/lucene/index/DocumentsWriterFlushControl.java#L228

https://github.com/apache/lucene-solr/blob/branch_7_6/lucene/core/src/java/org/apache/lucene/index/DocumentsWriterPerThreadPool.java#L164

4.11 Lucene 버퍼는 1회용

Flush 이후 버퍼는 재사용 하지 않는다.

```
synchronized void doAfterFlush/DocumentsWriterPerThread dwpt) {  
    assert flushingWriters.containsKey(dwpt);  
    try {  
        Long bytes = flushingWriters.remove(dwpt);  
        flushBytes -= bytes.longValue();  
        perThreadPool.recycle(dwpt);  
        assert assertMemory();  
    } finally {  
        try {  
            updateStallState();  
        } finally {  
            notifyAll();  
        }  
    }  
}
```

```
void recycle/DocumentsWriterPerThread dwpt) {  
    // don't recycle DWPT by default  
}
```

https://github.com/apache/lucene-solr/blob/branch_7_6/lucene/core/src/java/org/apache/lucene/index/DocumentsWriterFlushControl.java#L228

https://github.com/apache/lucene-solr/blob/branch_7_6/lucene/core/src/java/org/apache/lucene/index/DocumentsWriterPerThreadPool.java#L164

4.12 UTLC 버퍼 재사용

Flush 이후 할당 했던 버퍼를 Free list에 모아 재사용 한다.

```
void Recycle(std::unique_ptr<DocumentsWriterPerThread>&& dwpt) {  
    dwpt->MarkRecycled();  
  
    std::lock_guard<std::mutex> sync_this(this_lock);  
    dwpt_free_list.emplace_back(std::move(dwpt));  
}
```

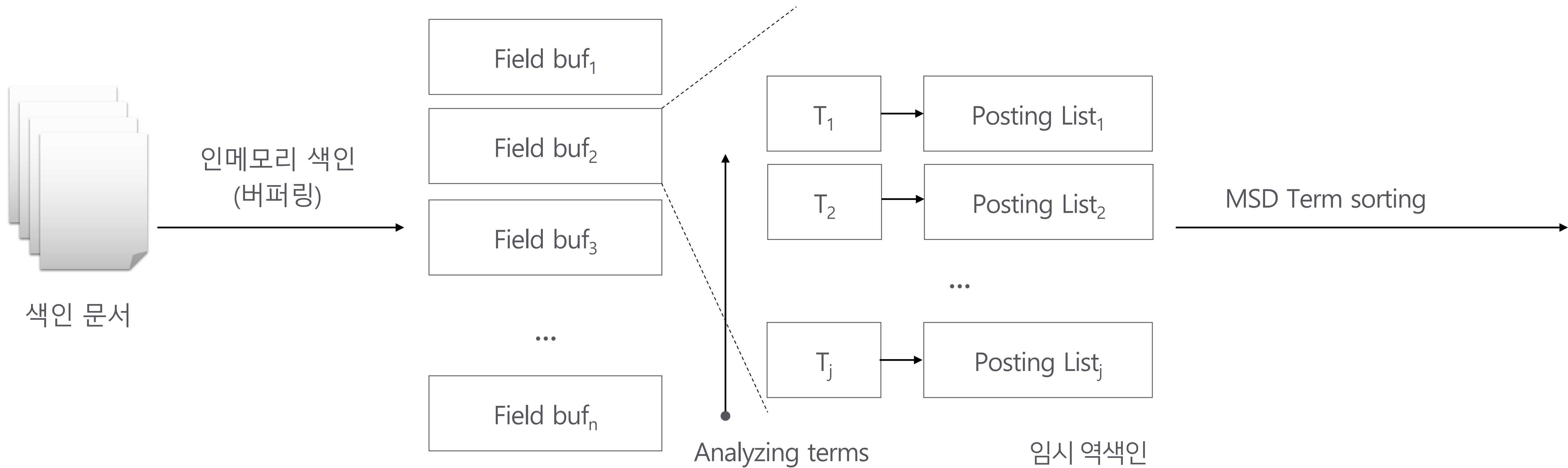
Free list에 추가한다.

```
if (not target->IsInitialized() && not dwpt_free_list.empty()) {  
    target->dwpt = std::move(dwpt_free_list.back());  
    dwpt_free_list.pop_back();  
} // End if
```

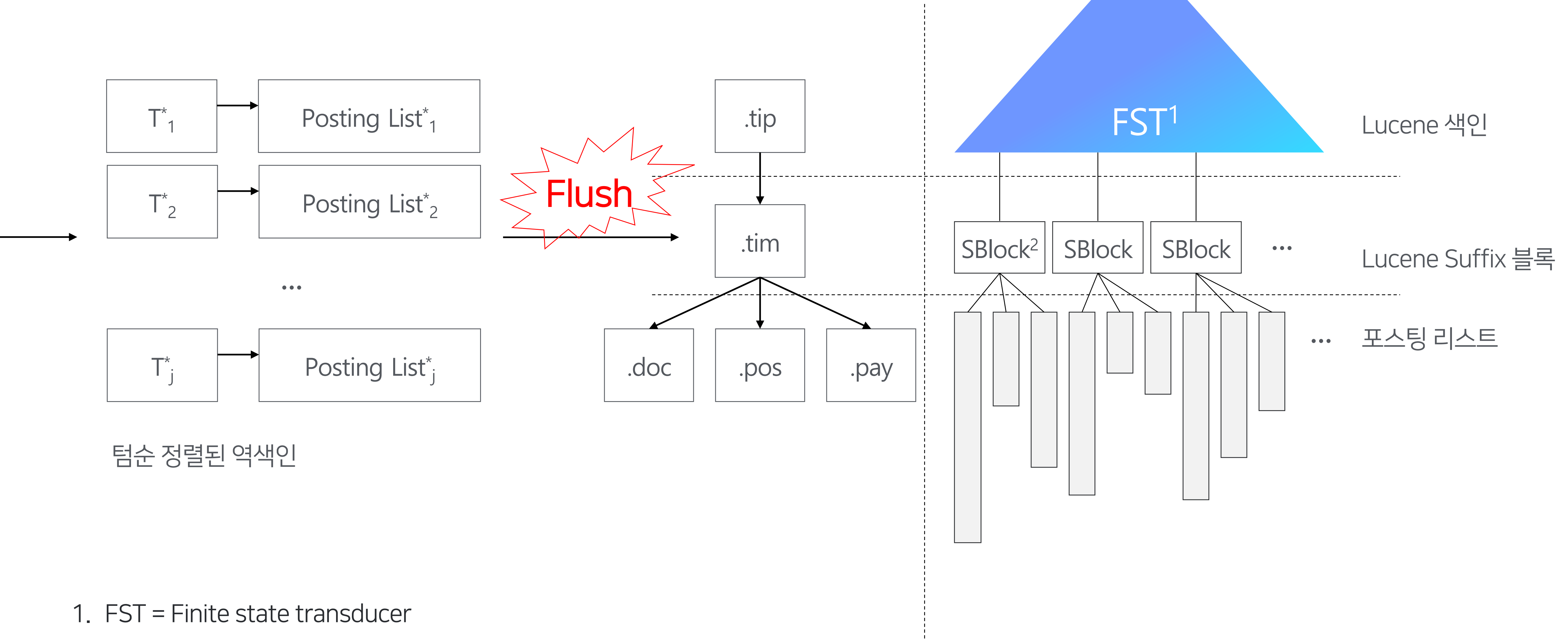
Free list에서 가져온다.

5. Flush, Lucene의 진짜 색인

5.1 전체 색인 플로우

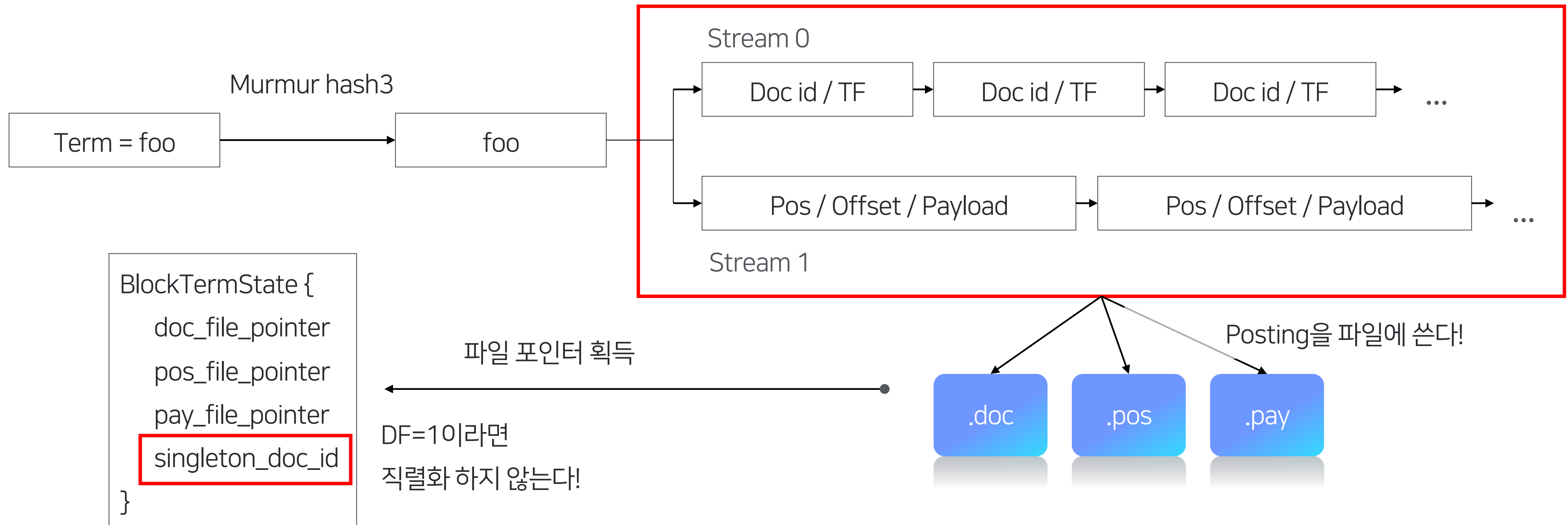


5.1 전체 색인 플로우



1. FST = Finite state transducer
2. Sblock = Suffix block

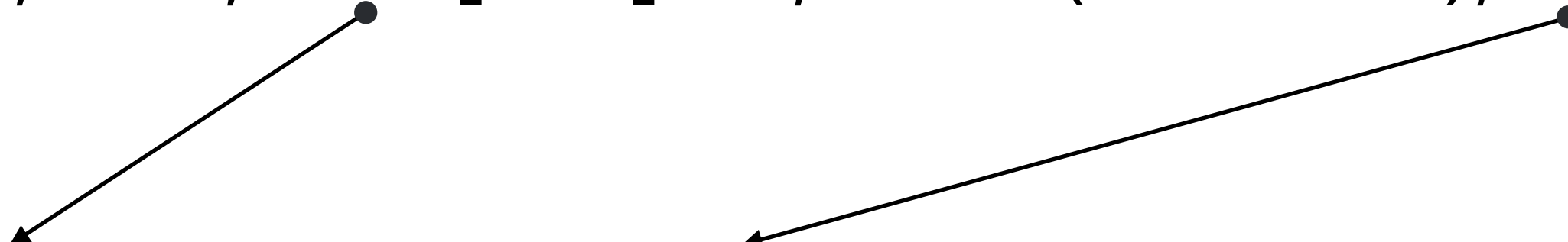
5.2 포스팅 리스트 Serialization



5.3 Delta 인코딩

전 값과의 **차이**만 저장한다.

$[13, 120, 190] \rightarrow [13, 107=(120-13), 70=(190-120)]$



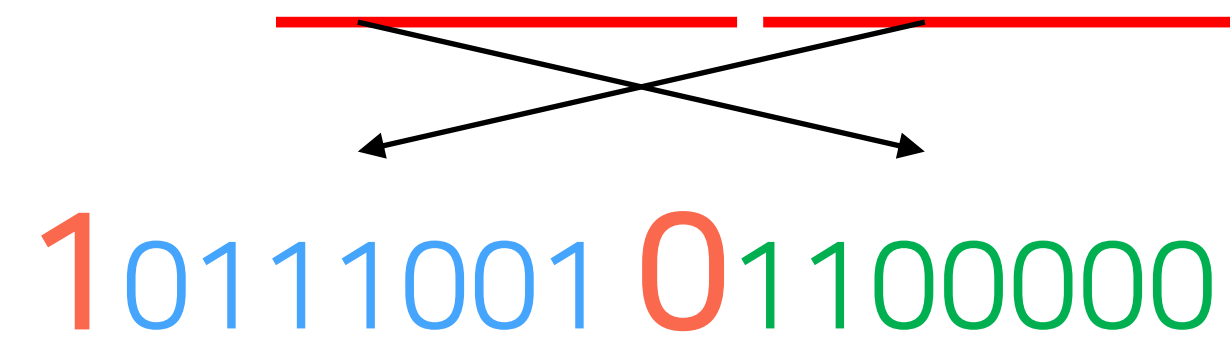
190 (2 바이트) 대신 70 (1 바이트) 만 저장하면 된다.

1 바이트를 절약!

5.4 VInt 인코딩

필요한 **최소 바이트만** 사용한다.

Ex: 12,345 = 00 11000000 0111001



10111001 01100000

[0, 2⁸) : 0xxxxxxx

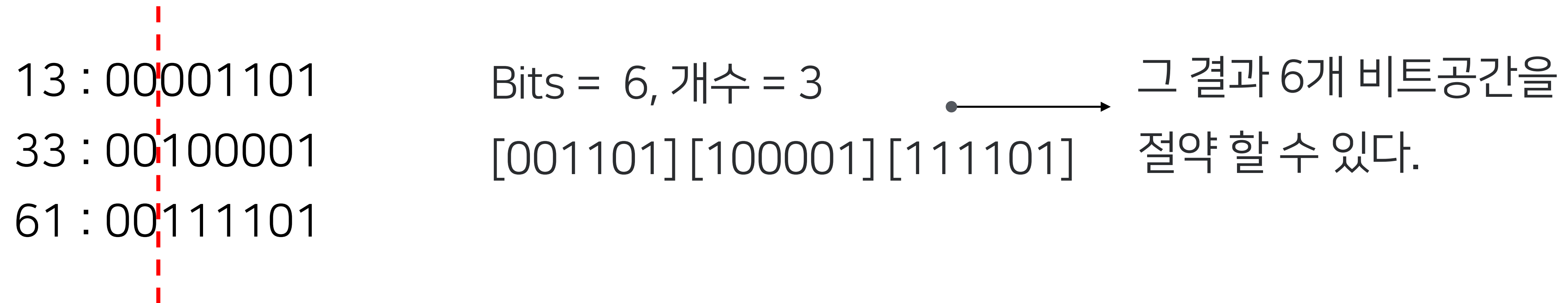
[2⁸, 2¹⁶) : 1xxxxxxx 0xxxxxxx

[2¹⁶, 2²⁴) : 1xxxxxxx 1xxxxxxx 0xxxxxxx

[2²⁴, 2³²) : 1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx

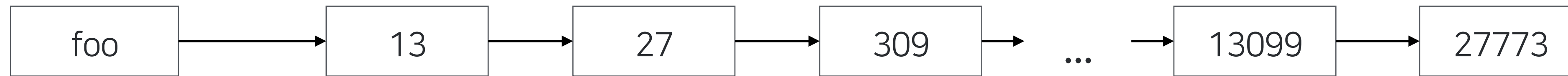
5.5 Bit packing

필요한 **최소 비트만** 사용한다.



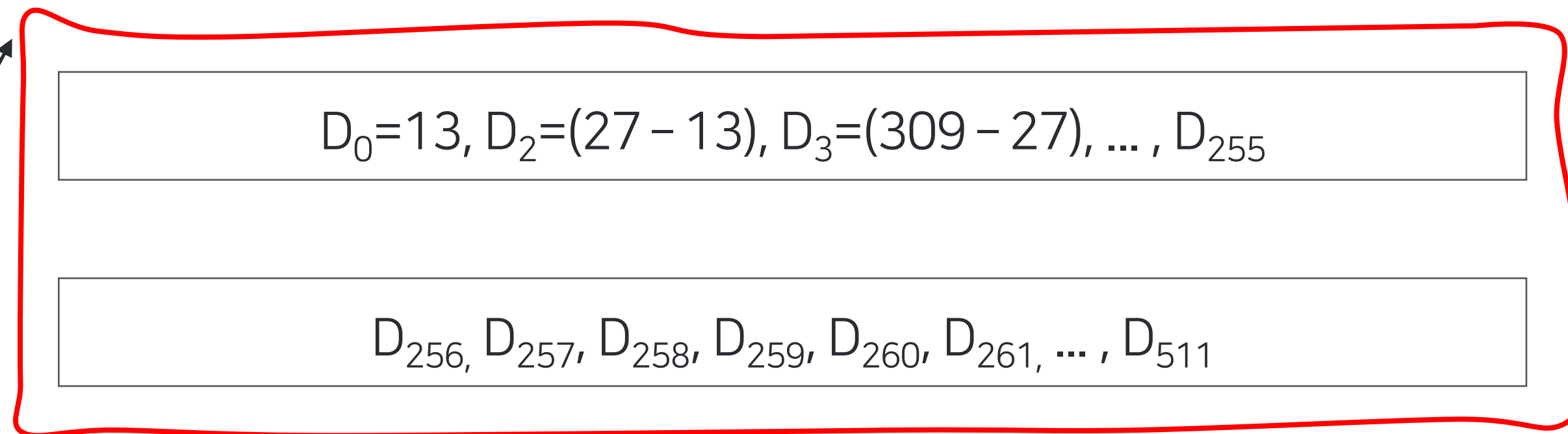
2개의 '00' 비트는 버려도 된다.

5.6 Lucene 포스팅 리스트 인코딩



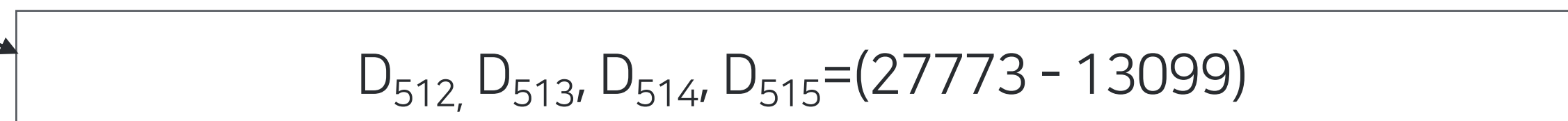
블록 인코딩

256 Ints를 모아,
델타 인코딩 + 비트 패킹



● → 256 Ints, Bit packing

나머지 Ints는
델타 인코딩 + VInt



● → Var Int encoding

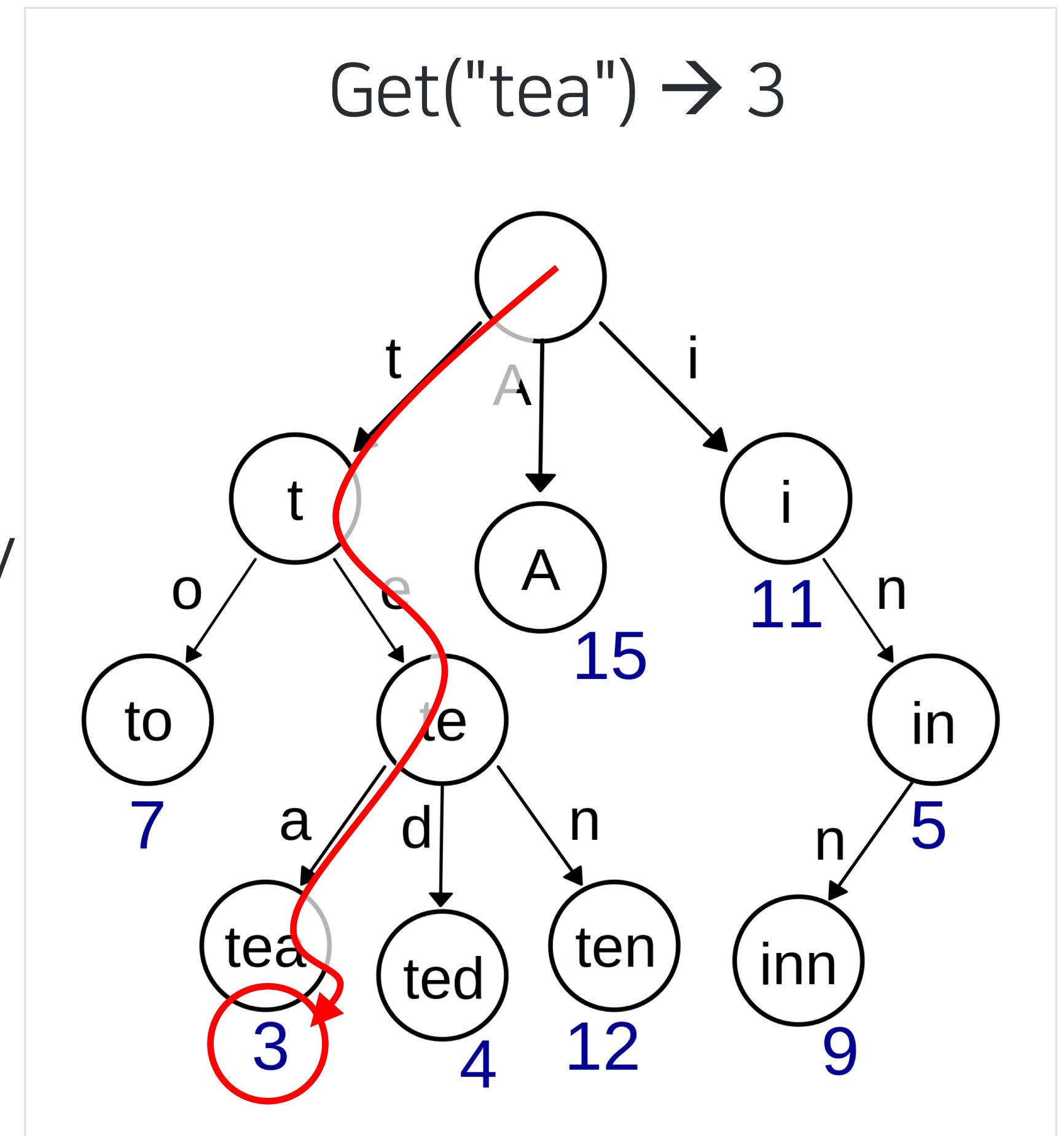
Payload는 그대로 저장된다

포스팅 리스트 먼저 그리고 색인 구축

5.7 Trie

1. Map<Key, Value>로 볼 수 있다.
2. DFA로 볼 수 있다 → FST

"A trie can be seen as a tree-shaped **deterministic finite automaton**. Each finite language is generated by a trie automaton, and each trie can be compressed into a **deterministic acyclic finite state automaton**."



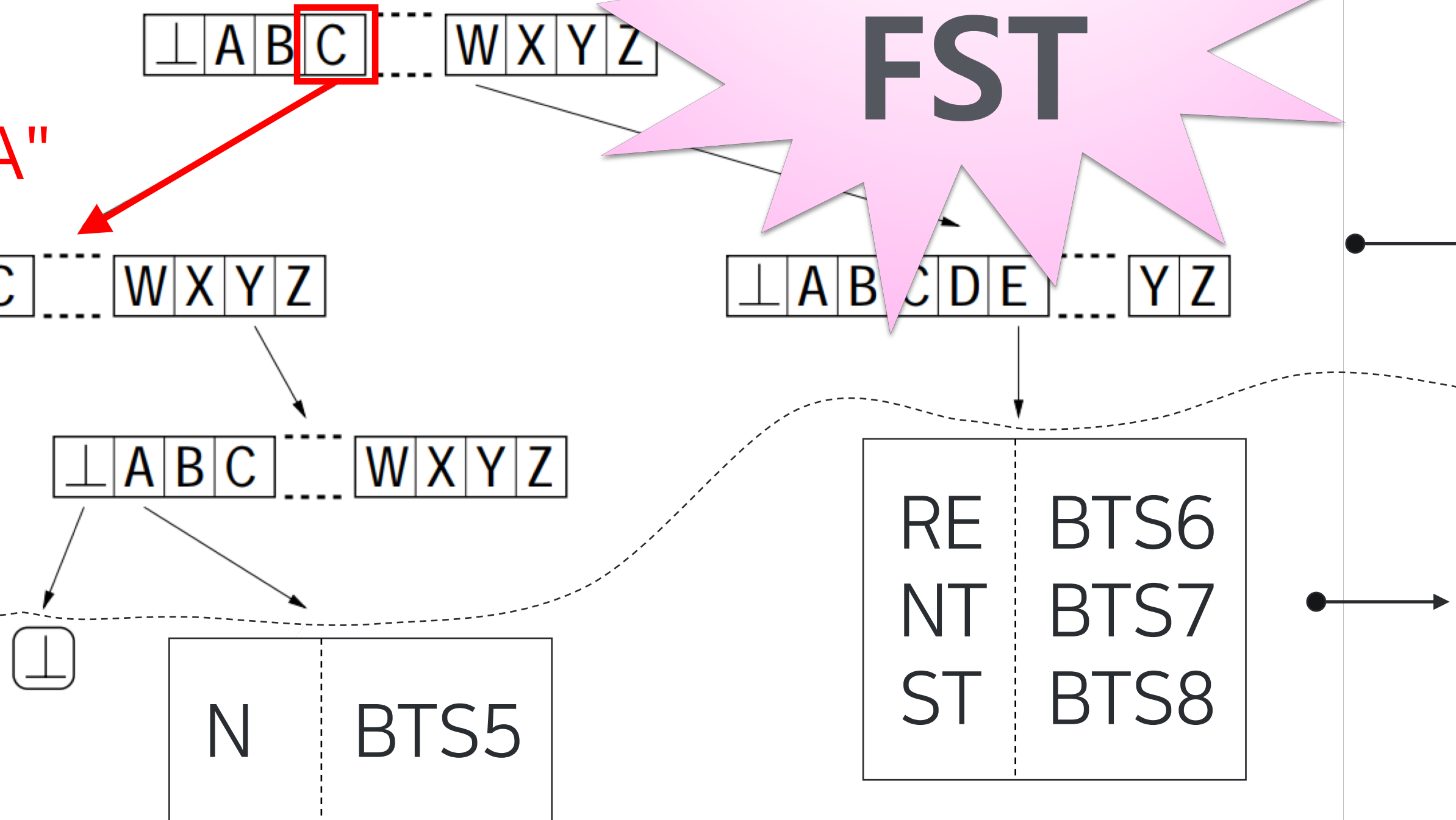
5.8 FST, Burst trie

공통 Prefix + Suffix 블록

Prefix "CA"

FST

CA	R	BTS1
CA	ME	BTS2
CA	T	BTS3
CA	VE	BTS4



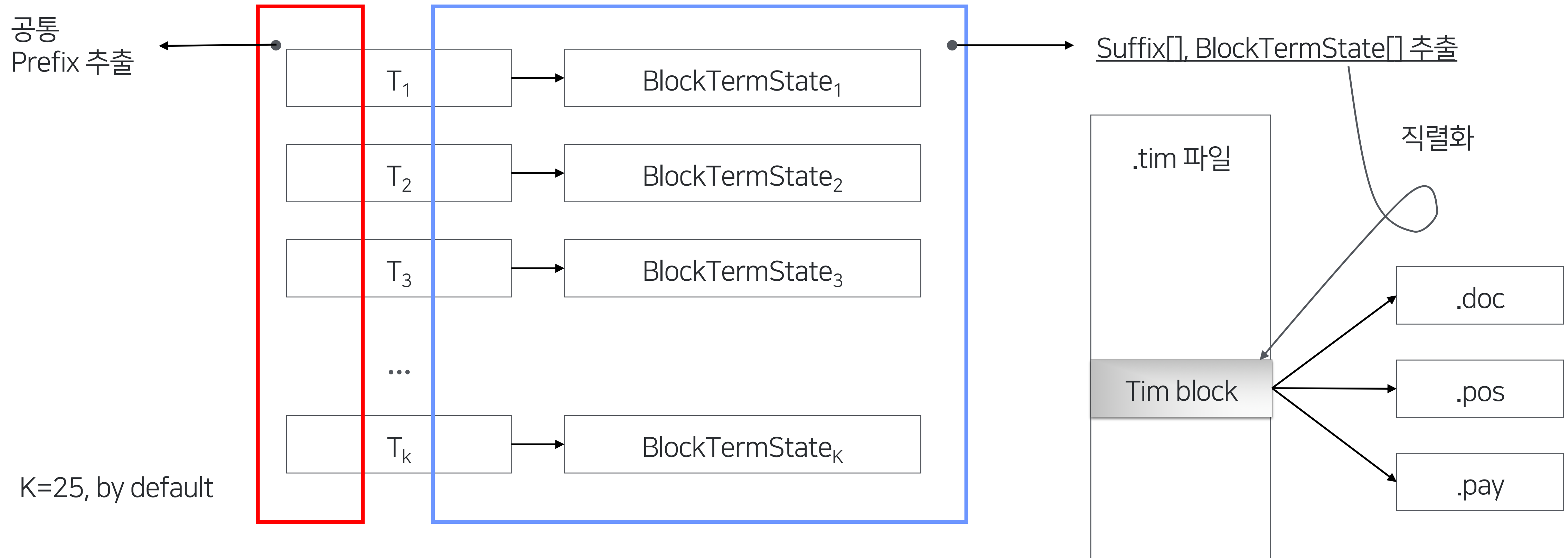
.tip

.tim

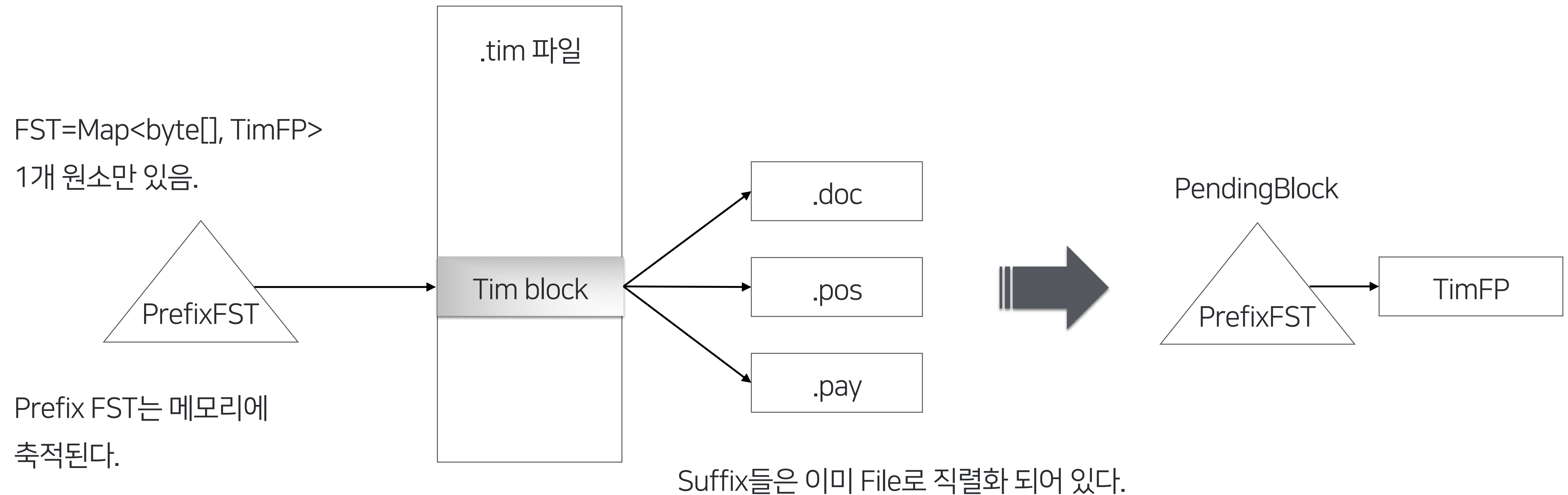
Figure 2: Burst trie after insertion of a record with key "western".

BTS = BlockTermState

5.9 Burst trie : suffix는 뭉쳐야 제맛

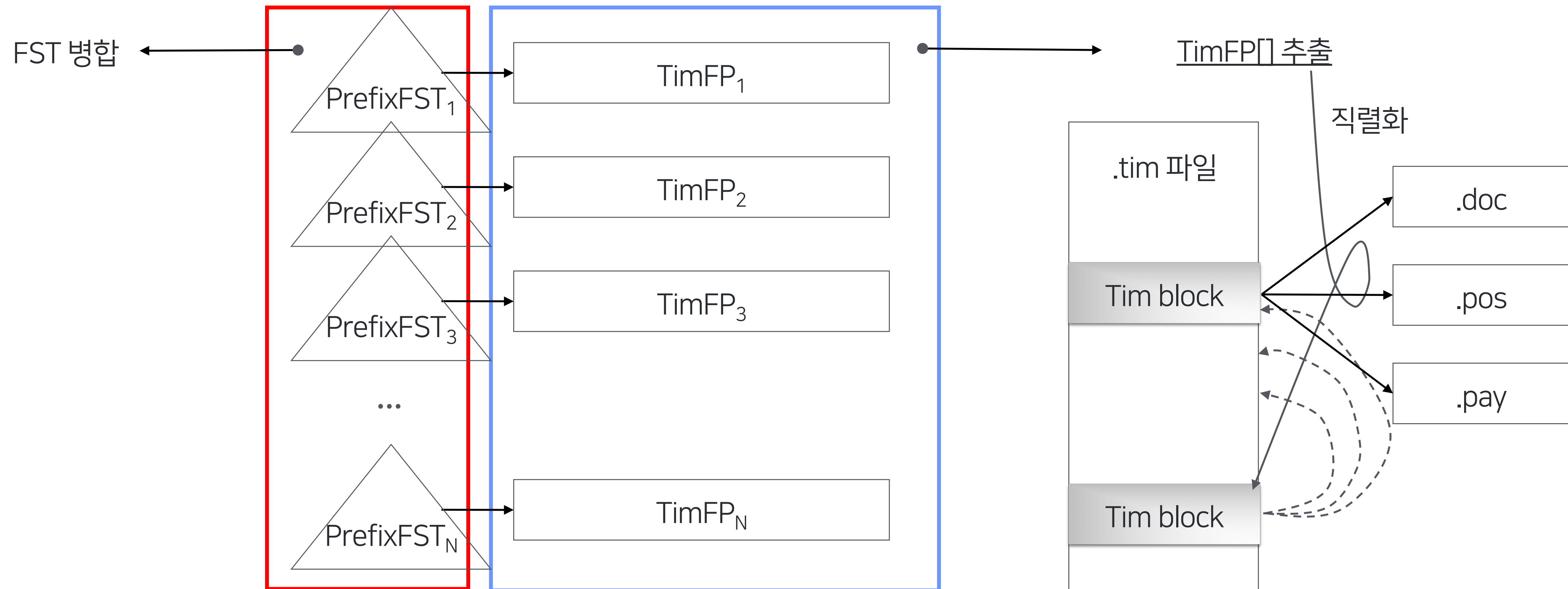


5.10 Burst trie : suffix는 뭉쳐야 제맛



TimFP = Tim block file pointer

5.11 Burst trie : suffix는 뭉쳐야 제맛



5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

T_1
BTS

5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

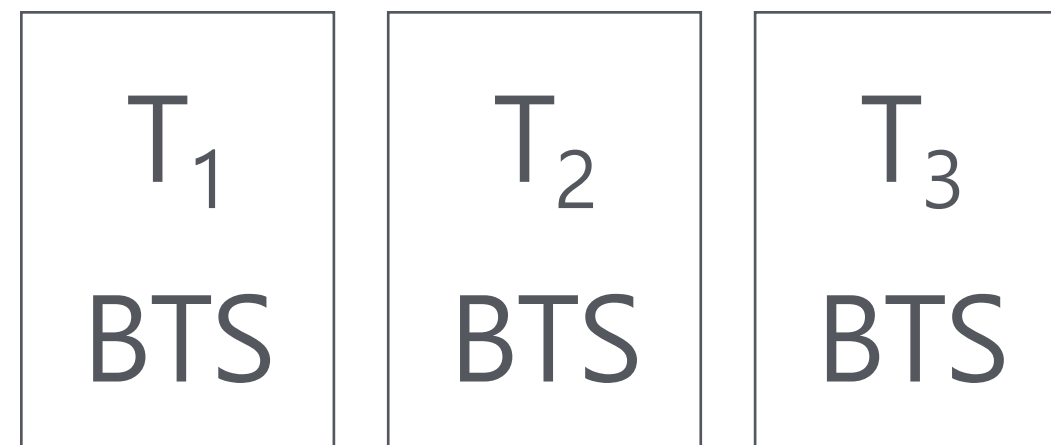


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer



5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

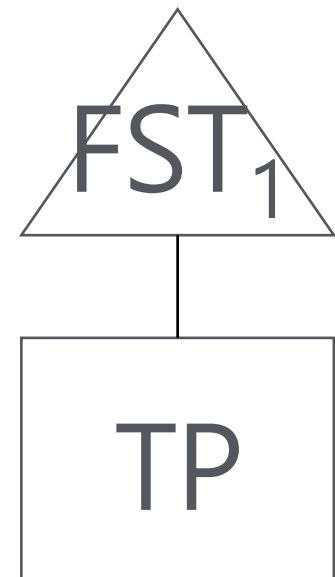


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

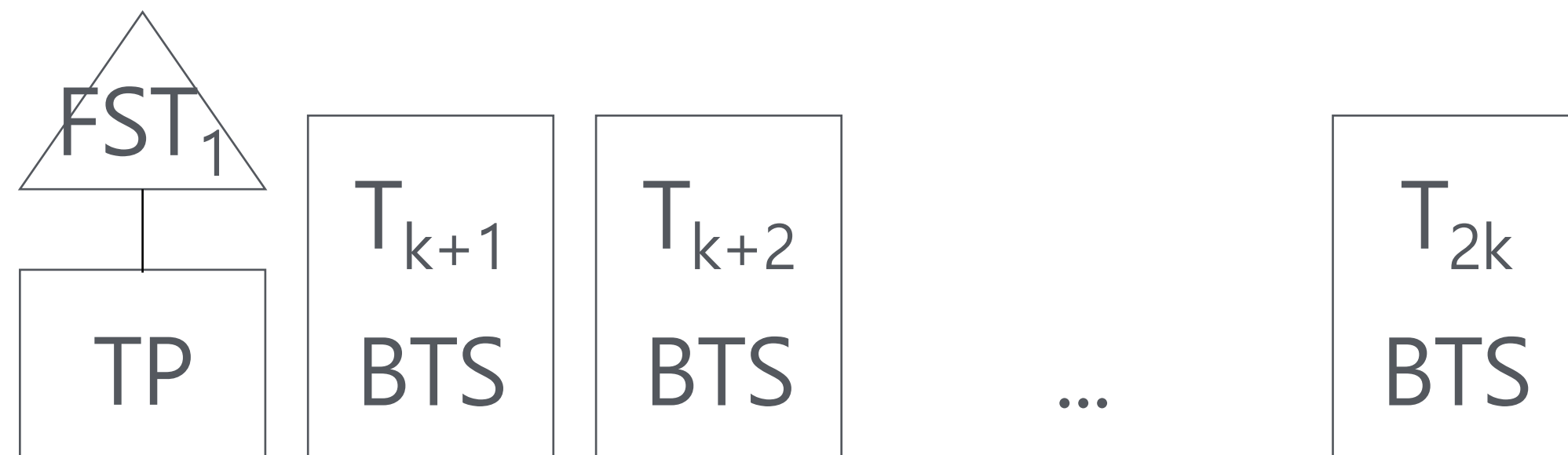


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

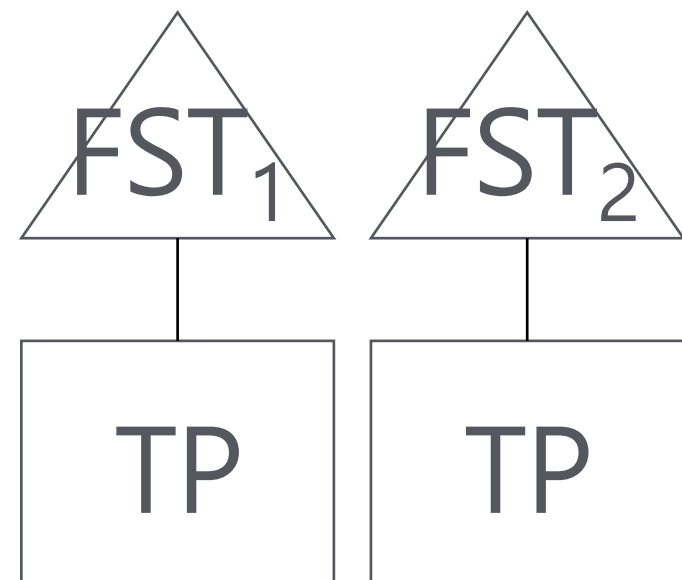


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

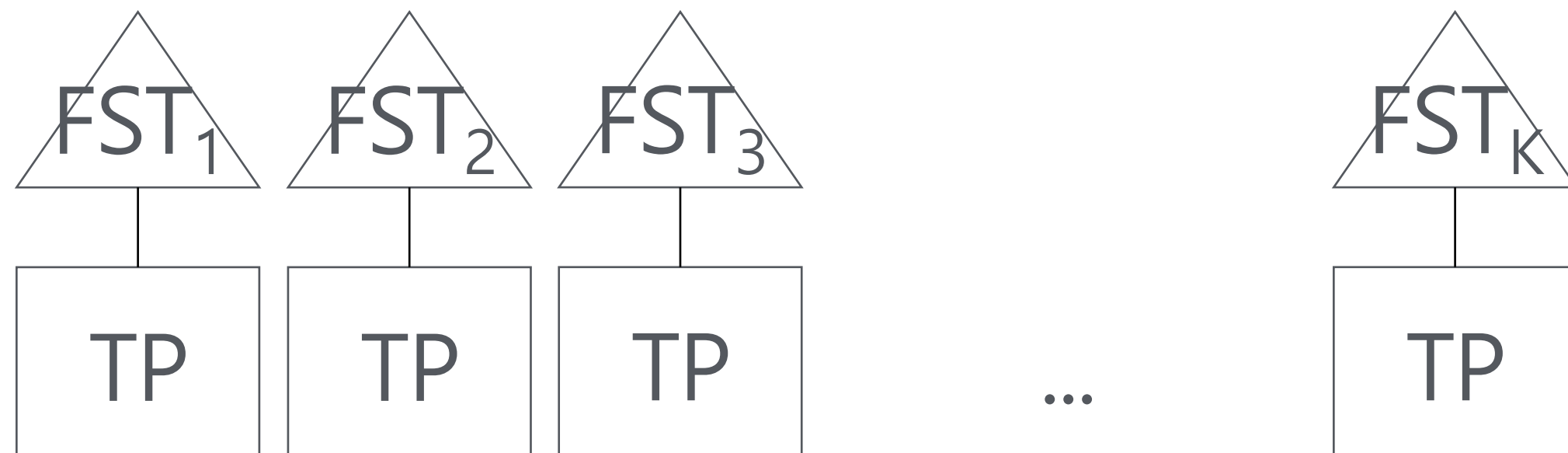


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

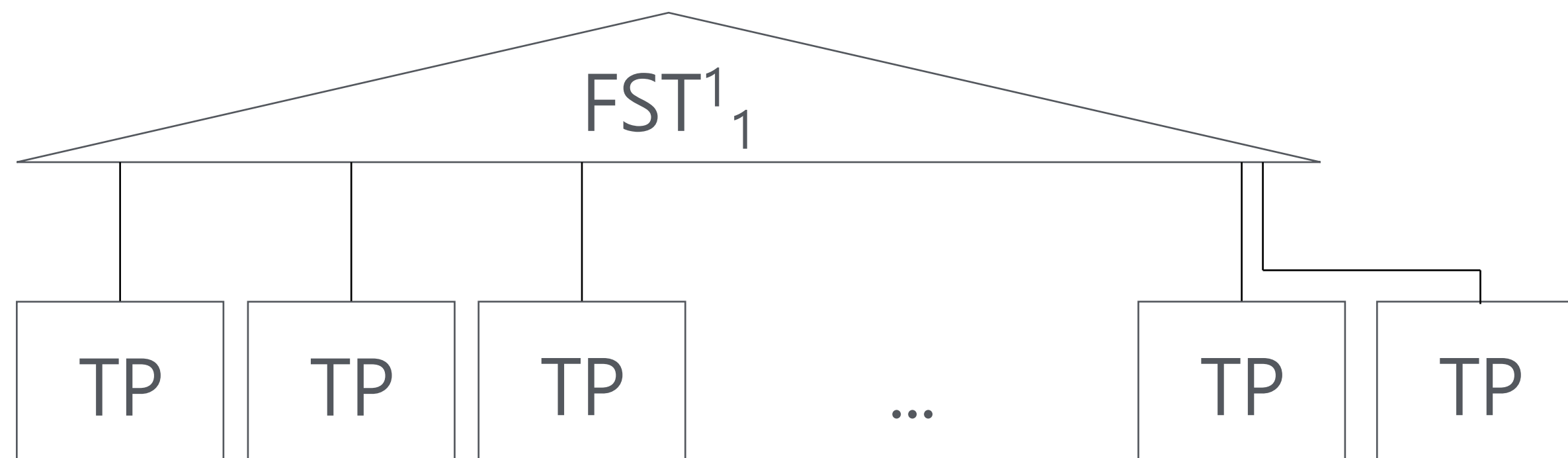


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

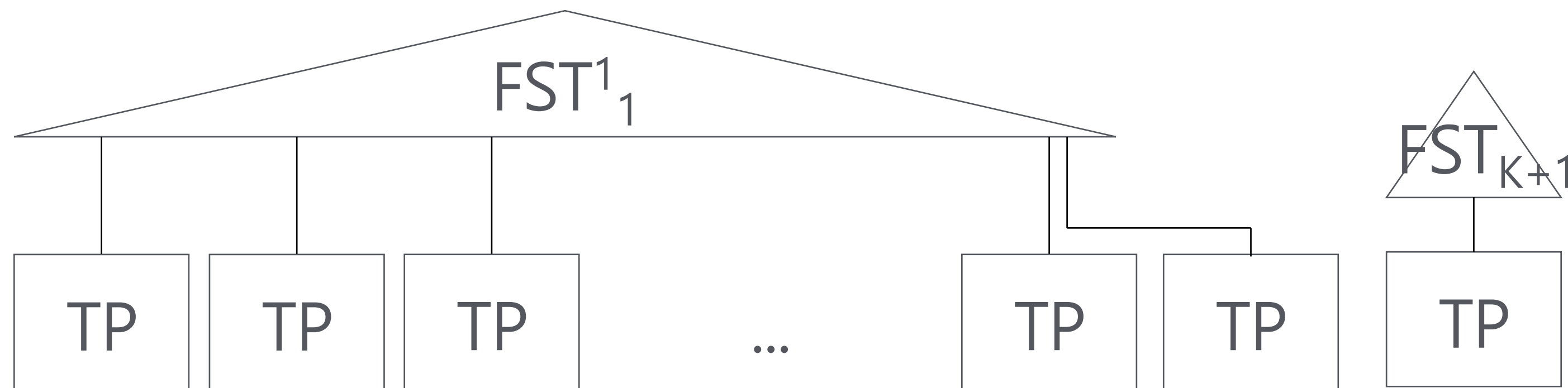


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

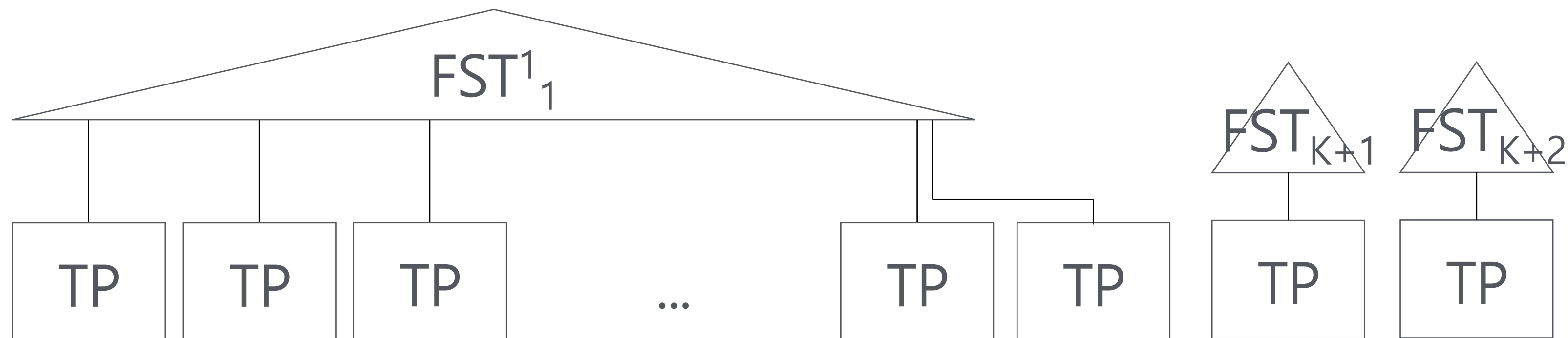


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

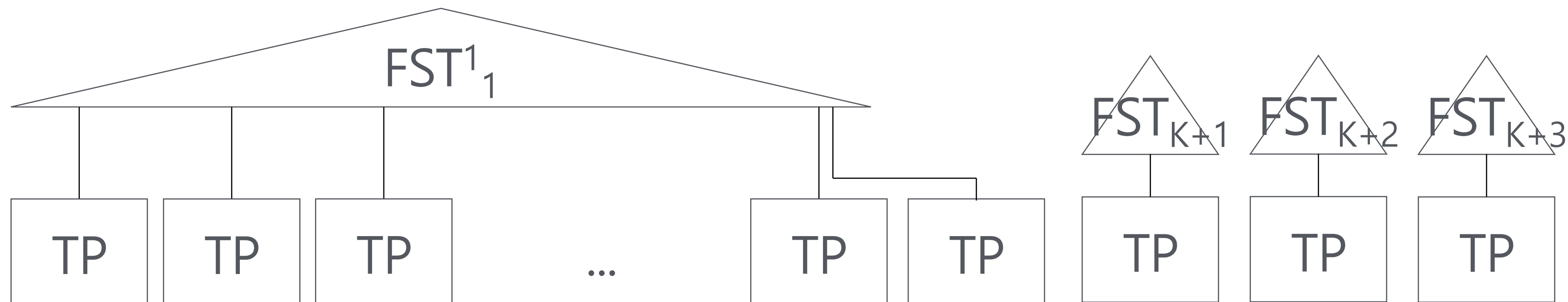


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

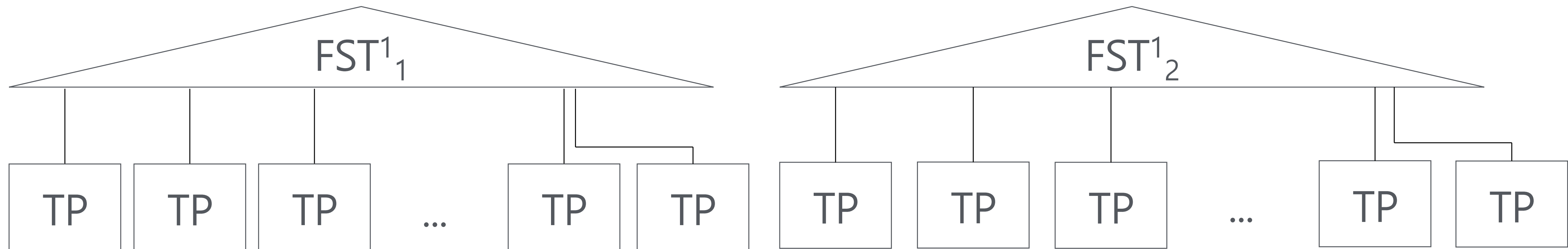


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer

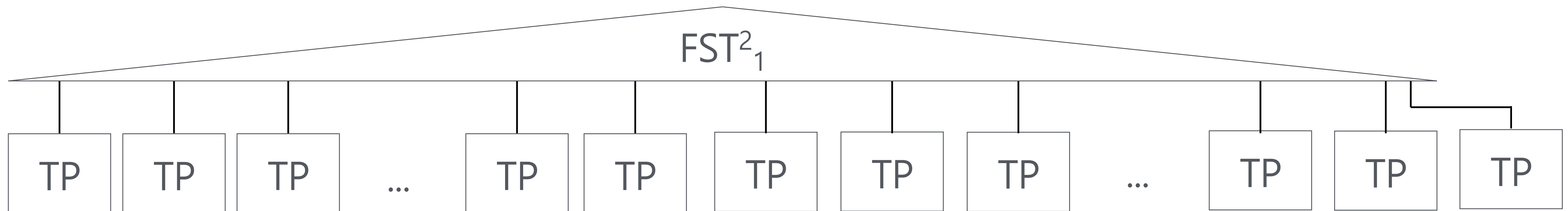


5.12 Burst trie : suffix는 뭉쳐야 제맛

T = Term

BTS = BlockTermState

TP = Tim File pointer



5.13 FST, Burst trie

공통 Prefix + Suffix 블록

Prefix "CA"

FST

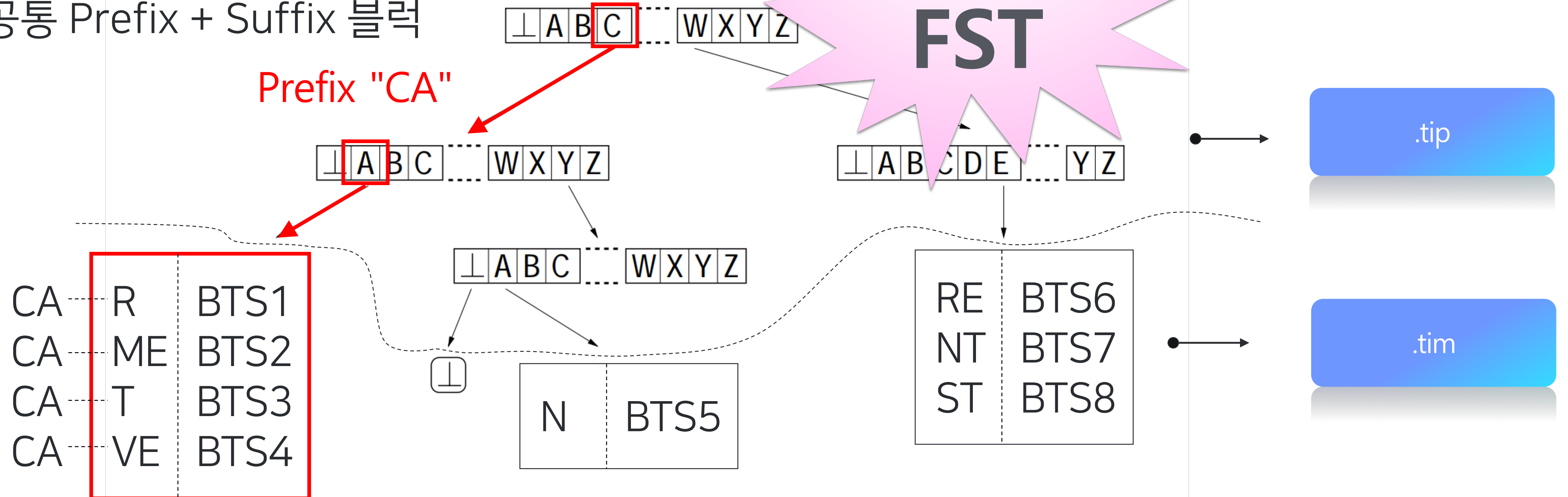
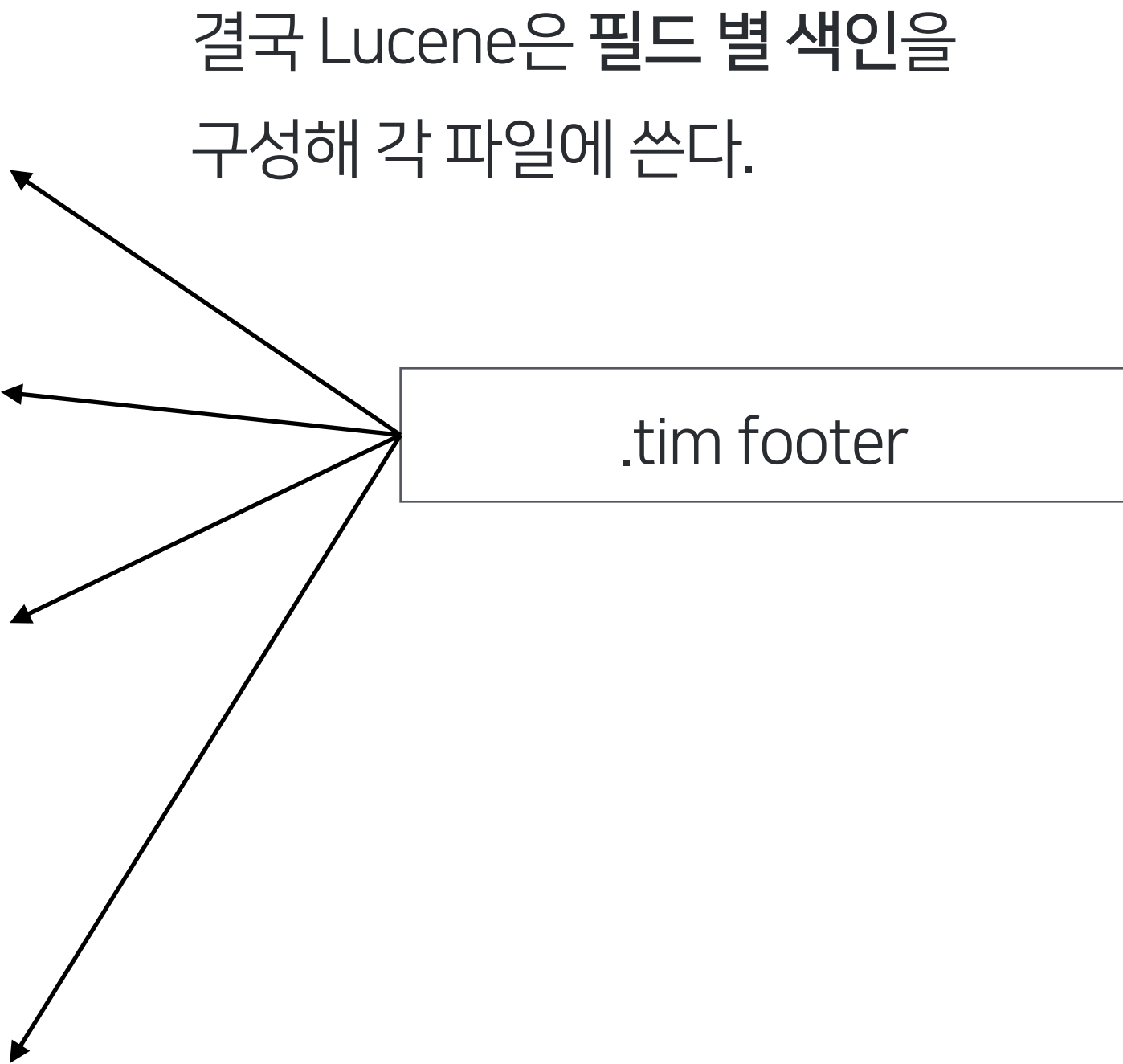


Figure 2: Burst trie after insertion of a record with key "western".

BTS = BlockTermState

5.14 필드별 저장 방식



5.15 FST : prefix는 겹쳐야 제맛

1. FST는 Trie 역할, 색인.
2. 공통 Prefix들을 저장
3. Field 쓰기 종료 까지 Memory에 존재한다.

$$\text{Ex) } FST^1_1 = \bigcup_{i=0}^k FST_i$$

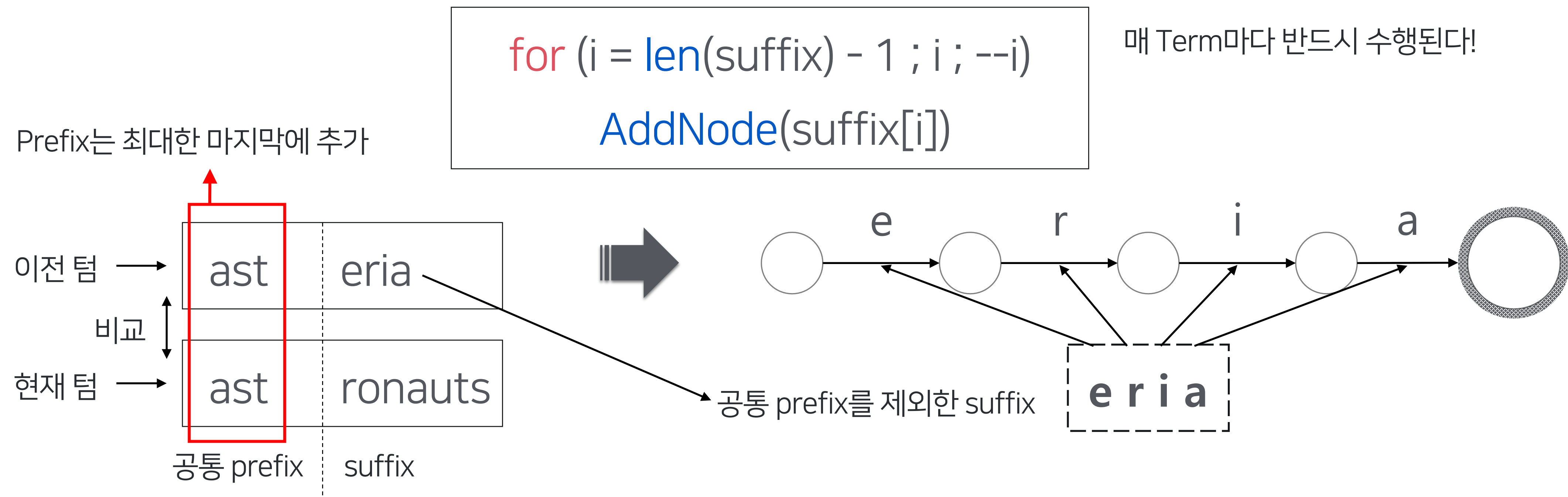
$$FST^2_1 = \bigcup_{i=0}^k FST^1_i$$

...

$FST^n_1 \leftarrow$ 최종 FST로써 .tip파일에 저장됨.

5.16 FST 빌딩 알고리즘

전 Term과 비교해 Suffix를 Graph에 추가한다.



5.17 FST Hot loop 최적화

```
final ByteSequenceOutputs outputs = ByteSequenceOutputs.getSingleton();
final Builder<BytesRef> indexBuilder = new Builder<>(FST.INPUT_TYPE.BYTE1,
                                                    0, 0, true, false, Integer.MAX_VALUE,
                                                    outputs, true, 15);
```

→ 파라미터들이
컴파일 타입에 결정된다.

```
final byte[] bytes = new byte[(int) scratchBytes.getFilePointer()];
assert bytes.length > 0;
scratchBytes.writeTo(bytes, 0);
```

↓ 컨트롤 변수 생성자 전달

↓ `minSuffixCountN` 은 0으로 결정되어 있다!

```
public Builder(FST.INPUT_TYPE inputType, int minSuffixCount1, int minSuffixCount2, boolean doShareSuffix,
              boolean doShareNonSingletonNodes, int shareMaxTailLength, Outputs<T> outputs,
              boolean allowArrayArcs, int bytesPageBits) {
```

5.18 FST Hot loop 최적화

```
for (int idx=lastInput.length(); idx >= downTo; idx--) {
  if (node.inputCount < minSuffixCount1)
    ...
  else if (idx > prefixLenPlus1)
    if (parent.inputCount < minsSuffixCount2 || (minSuffixCount2 == 1 ...))
      ...
    if (node.inputCount < minSuffixCount2 || (minSuffixCount2 == 1 && ...))
      ...
    if (doPrune) ...
  else {
    if (minSuffixCount2 != 0) ...
    if (doCompile) ...
  }
}
```

7개의 Branch들은 대부분 **컴파일에 결정**된다.

`minSuffixCountN == 0` 이므로 **평가할 필요가 없다.**

5.19 Templated FST

```
// Byte size = 1, min suffix count1 = 0, min suffix count2 = 0,  
// do share suffix = true, do share non singleton nodes = false,  
// share max tail length = MAX int32, allow array arcs = true  
FstBuilder<BytesRef,  
    FstInputType::Byte1,  
    0, 0, true, false, std::numeric_limits<int32_t>::max(), true>  
    index_builder(std::make_unique<ByteSequenceOutputs>(), 15);
```



컴파일 타임에 코드가 최적화 되어 생성된다.

5.19 Templated FST

```
492: for (int32_t idx = last_input.Length() ; idx >= down_to ; --idx) {
    ...
503: UnCompiledNode* parent(frontier[idx - 1]);
    ...
559: const bool is_final(node->is_final || node->arcs.empty());
    ...
561: if (do_compile) {
562:     CompiledNode* compiled_node(
563:         CompileNode(node, 1 + last_input.Length() - idx, false));
    ...
```

FstBuilder.h

Branch 코드들이 삭제 되었다!

CompileNode의 인라이닝

```
<_ZN6lucene4core4util10FstBuilderINS1_3ReflcFreezeTailEi>:
...
mov -0x8(%rax,%rbx,1),%rbp    # FstBuilder.h:503
movzbl 0x44(%r14),%r13d    # FstBuilder.h:559
mov 0x18(%r14),%rdx
mov 0x10(%r14),%rax
test %r13b,%r13b
jne 89 <_ZN6lucene4core4util10FstBuilderINS1_3ReflcFreezeTailEi>
cmp %rax,%rdx    # FstBuilder.h:559
sete %r13b
mov 0xa8(%r15),%rsi    # FstBuilder.h:453
...
```

FstBuilder.s

5.20 FST 빌드, 읽기, Lucene VS UTLC

DEVIEW
2019

Writing time

1415 ms

854 ms

0 ms

Lucene

UTLC

1.65배

Reading time

230 ms

125 ms

0 ms

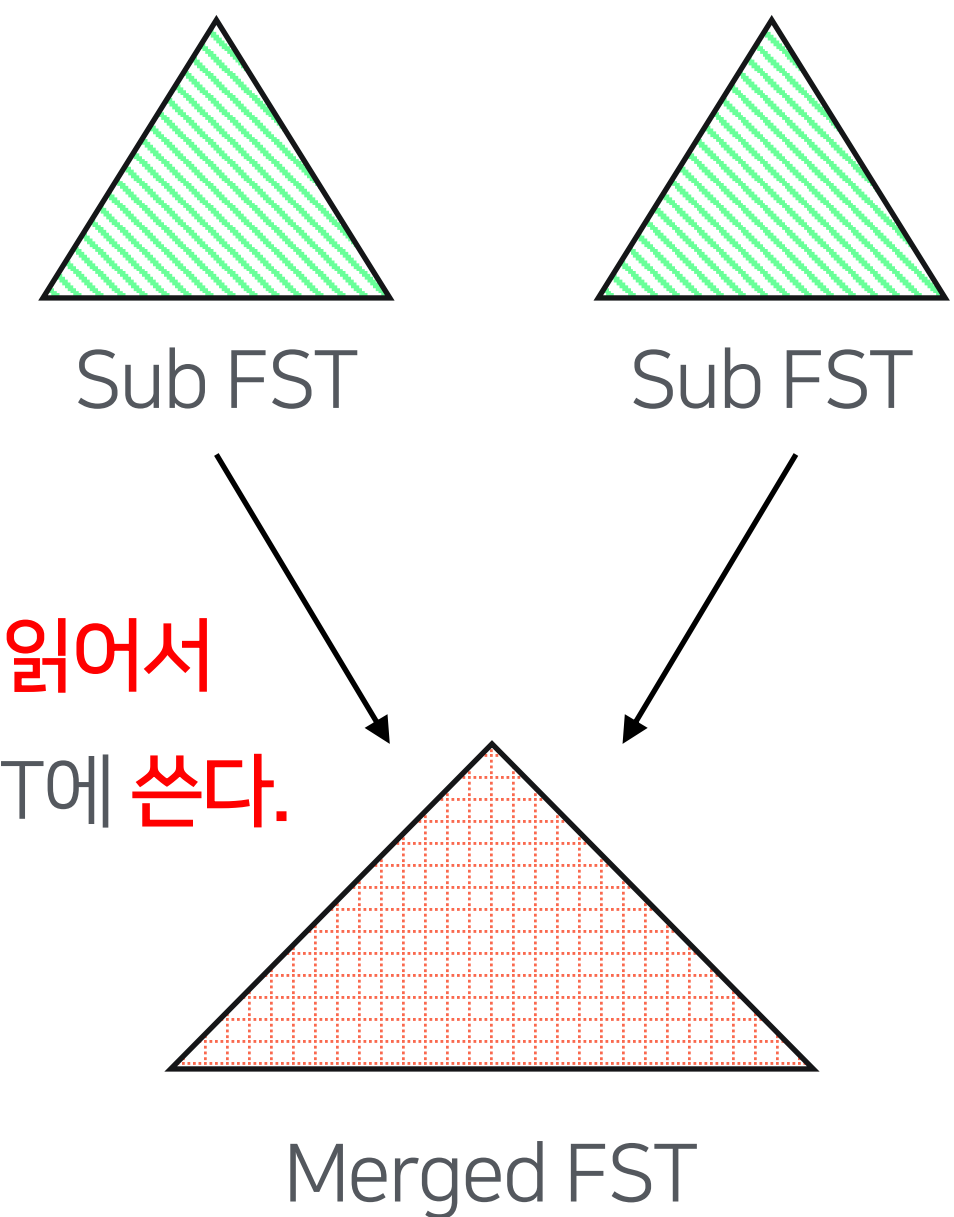
Lucene

UTLC

1.84배

FST 병합 =

Sub FST를 **읽어서**
Merged FST에 **쓴다.**



5.21 FST 빌드, Lucene VS UTLC

1. Intel® Core™ i7-8700 CPU @ 3.20GHz, 12cores, hyperthreaded, 48G Mem
2. 싱글 스레드
3. g++8, 03 컴파일 / Jdk11.0.2, 5G, 코드 컴파일 적용
4. 약 1,50만개 영단어
5. <https://www.keithv.com/software/wlist>

5.22 인덱싱 성능, Lucene VS UTLC

DEVVIEW
2019

검색, 병합 없는 순수 색인 성능 비교

Indexed docs

700,000

600,000

500,000

400,000

300,000

200,000

100,000

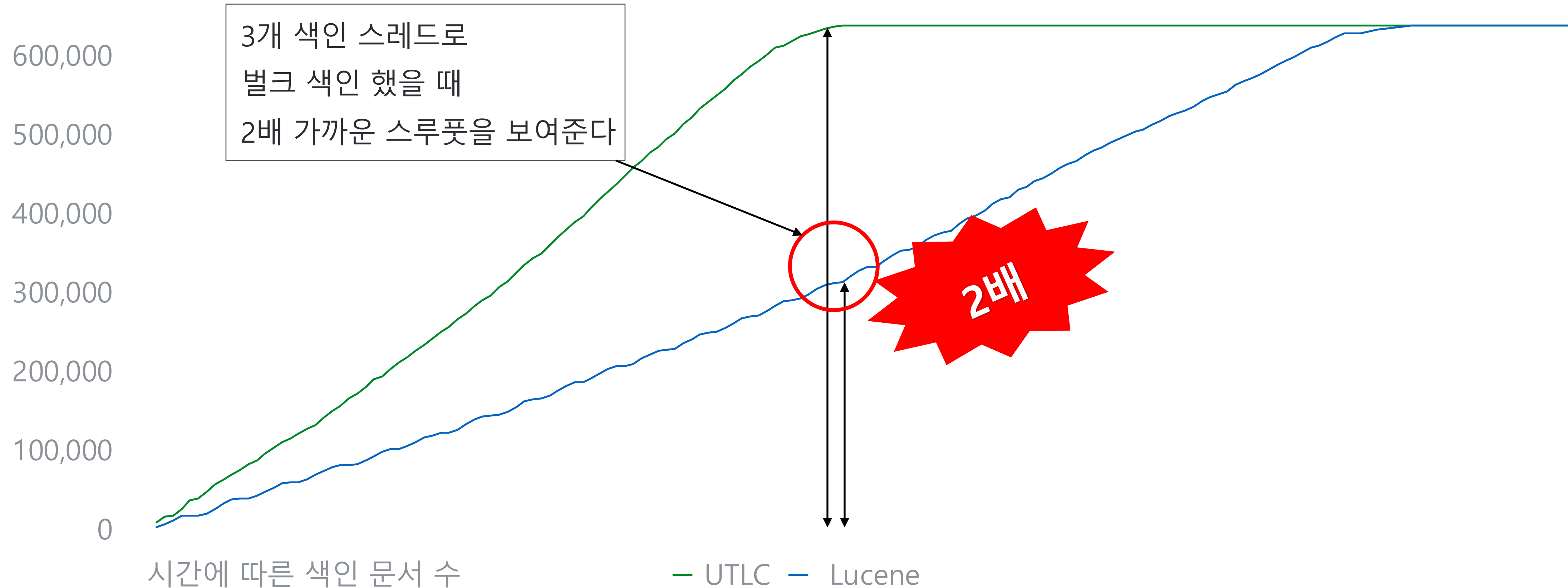
0

3개 색인 스레드로
벌크 색인 했을 때
2배 가까운 스루풋을 보여준다

2배

시간에 따른 색인 문서 수

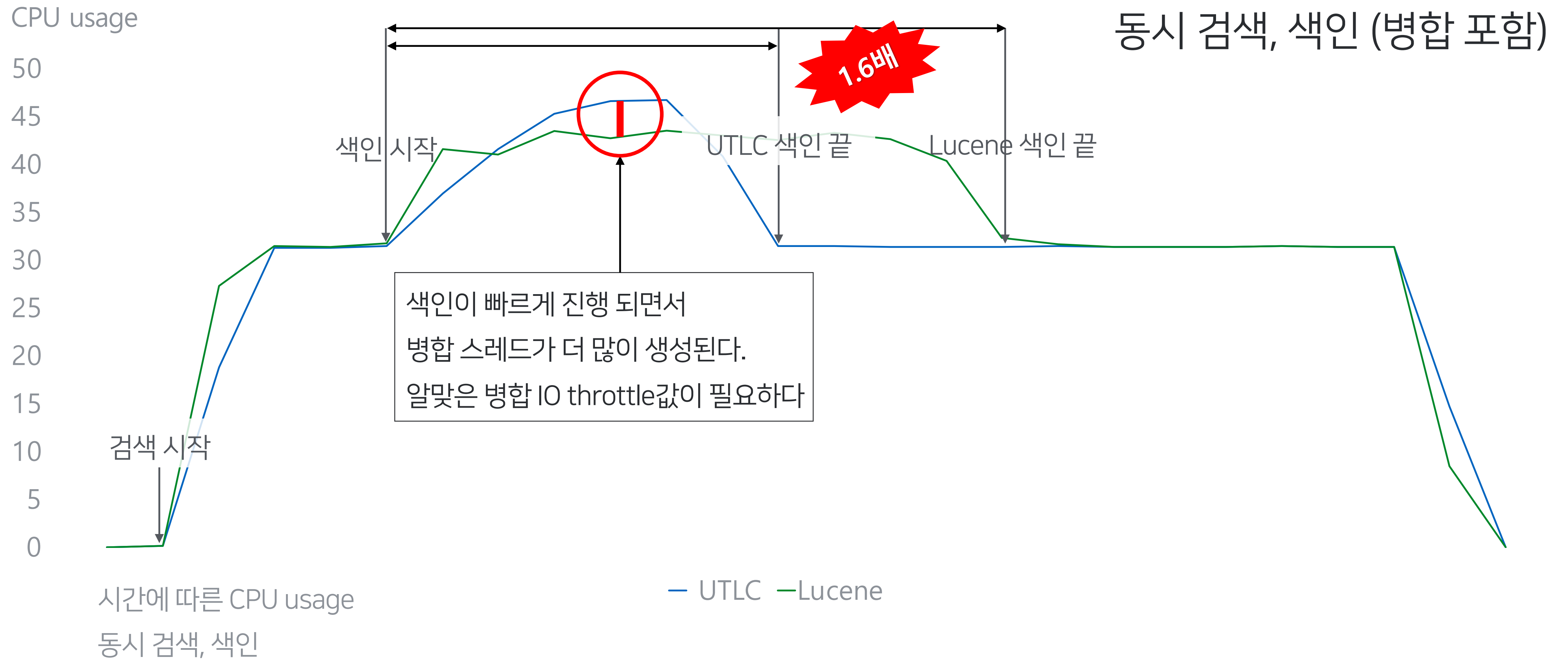
— UTLC — Lucene



5.23 인덱싱 성능, Lucene VS UTLC

1. Intel® Xeon® CPU E5-2530 @ 2.40GHz, 32 cores, hyper-threaded, 125G Mem
2. 약 65만건 문서 색인
3. 색인 스레드 3개
4. 한 문서 평균 크기 7,300 byte
5. 필드 60개
6. WhiteSpaceAnalyzer (Term analyzing 시간 최소화, 미리 전처리)
7. g++8, 03 컴파일 / Jdk11.0.2, 32G, 코드 컴파일 적용

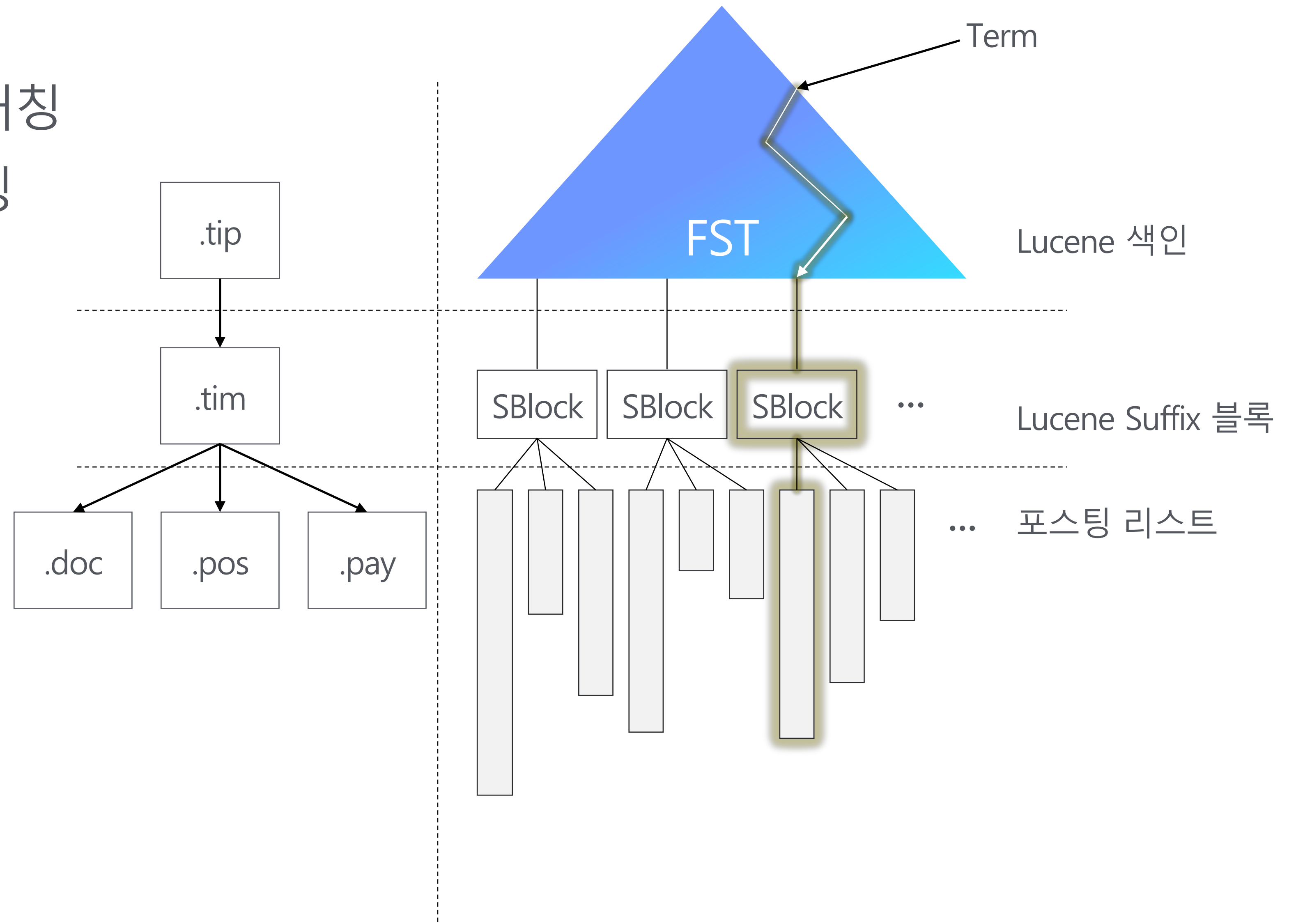
5.24 인덱싱 성능, Lucene VS UTLC



6. Lucene의 검색

6.1 Lucene 검색 플로우

1. 먼저 FST에서 Prefix를 매칭
2. Sblock에서 Suffix를 매칭
3. 포스팅 리스트 순회

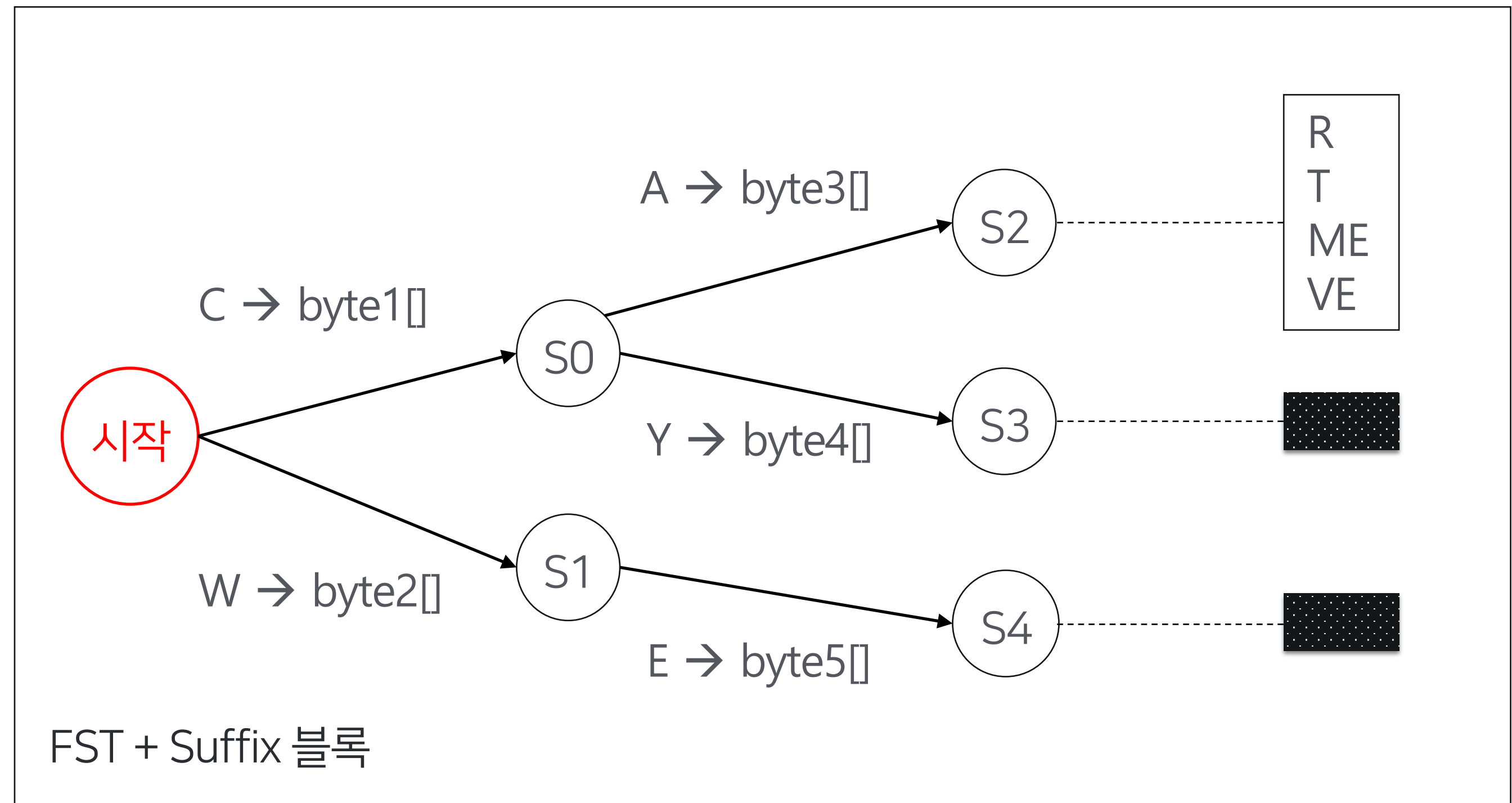


6.2 다시 FST

DEVIEW
2019

Search path에서 Output 조합한다

Key : CAVE
Matched : None
Value : None

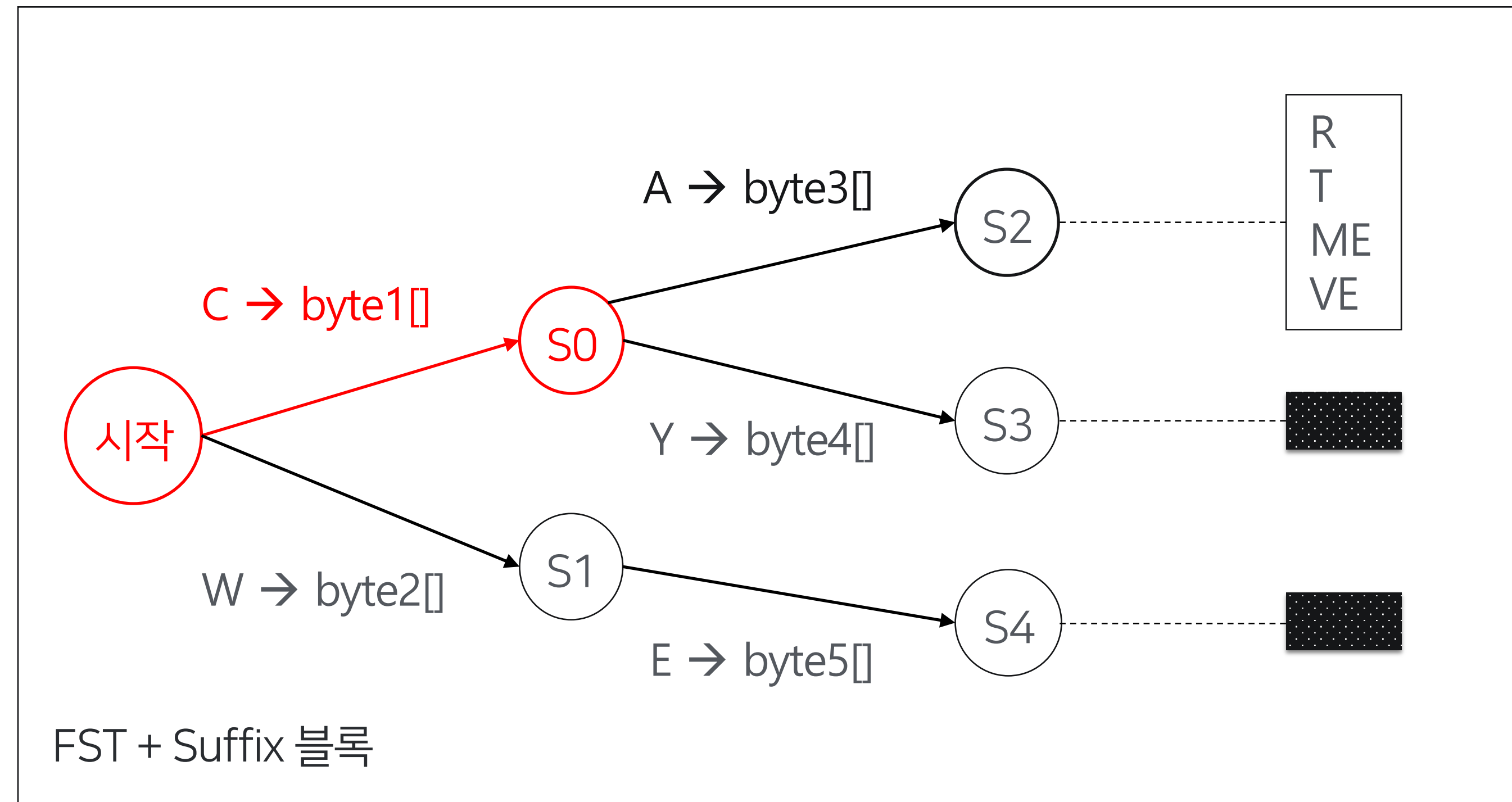


6.2 다시 FST

DEVIEW
2019

Search path에서 Output 조합한다

Key : CAVE
Matched : ^C
Value : byte1[]



6.2 다시 FST

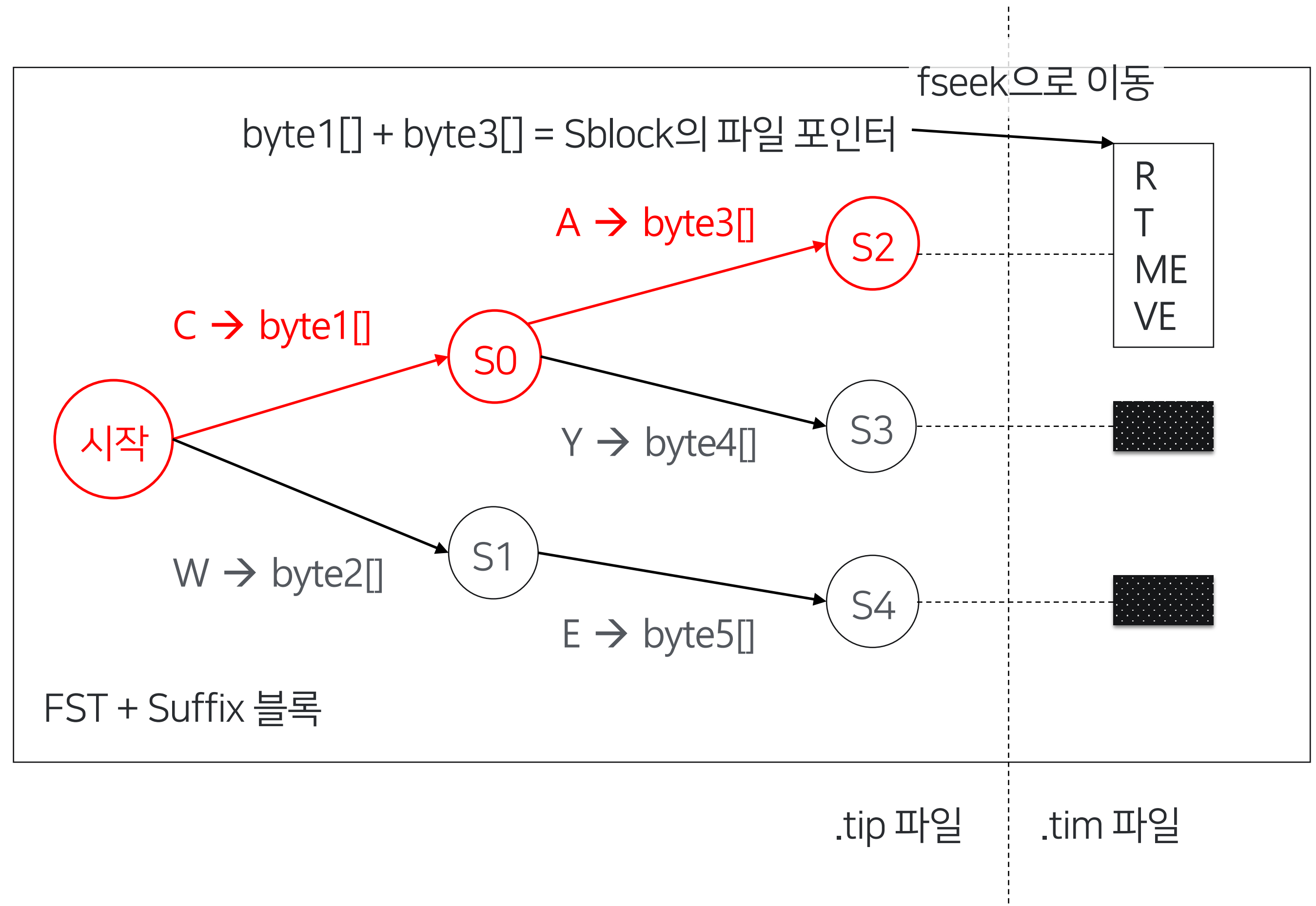
DEVVIEW
2019

Search path에서 Output 조합한다

Key : CAVE

Matched : ^CA

Value : byte1[] + byte3[]



6.2 다시 FST

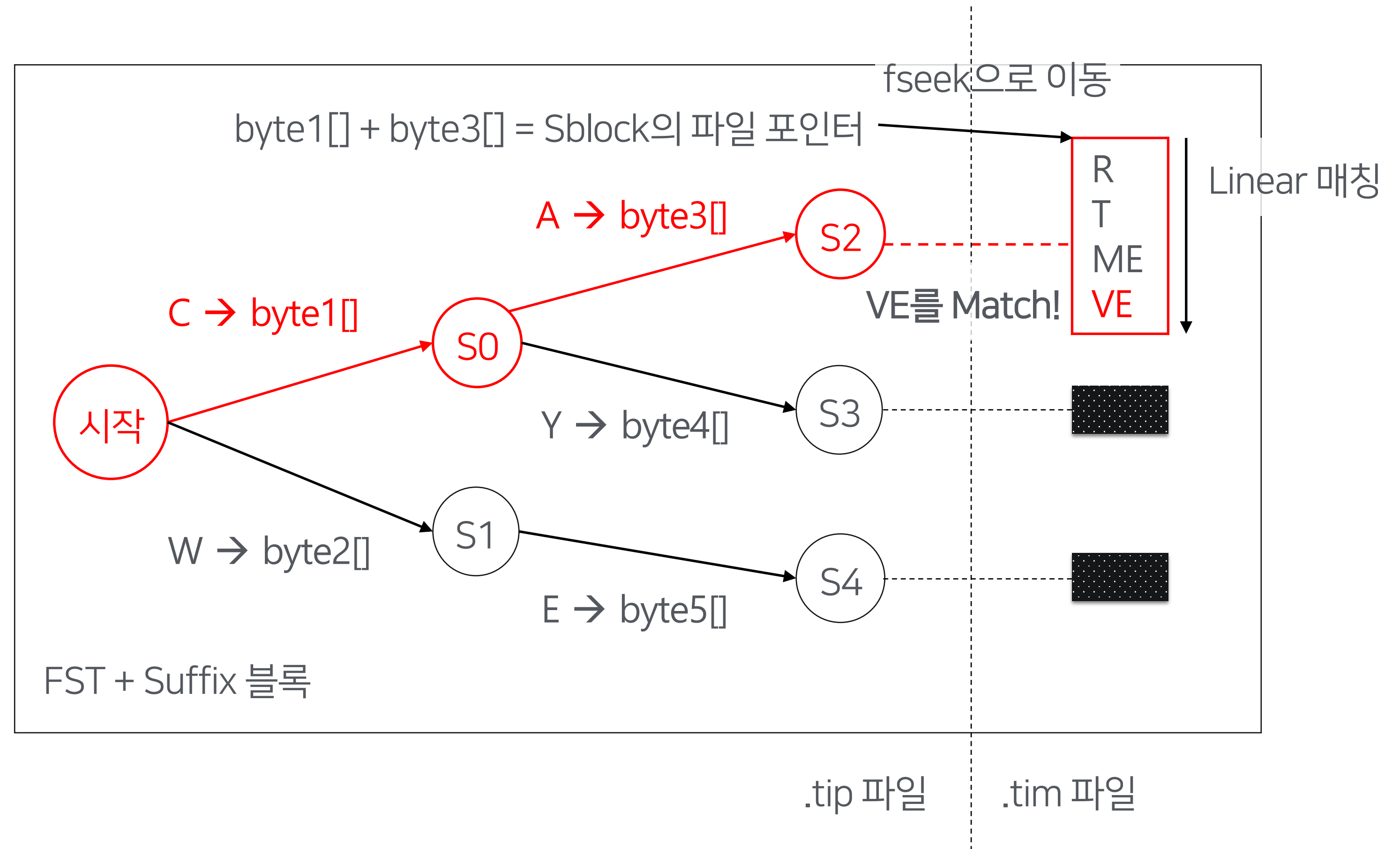
DEVIEW
2019

Search path에서 Output 조합한다

Key : cave

Matched : ^cave\$

Value : byte1[] + byte3[]



6.3 FST Output aggregation

```
// Aggregate output as we go:  
assert arc.output != null;  
if (arc.output != BlockTreeTermsReader.NO_OUTPUT) {  
    output = BlockTreeTermsReader.FST_OUTPUTS.add(output, arc.output);  
}
```

<https://github.com/apache/lucene-solr/blob/releases/lucene-solr/7.6.0/lucene/core/src/java/org/apache/lucene/codecs/blocktree/SegmentTermsEnum.java#L786>

6.4 빈번한 malloc은 당신의 목을 쥘다네

FST는 검색 Path에서 Output을 조합한다.

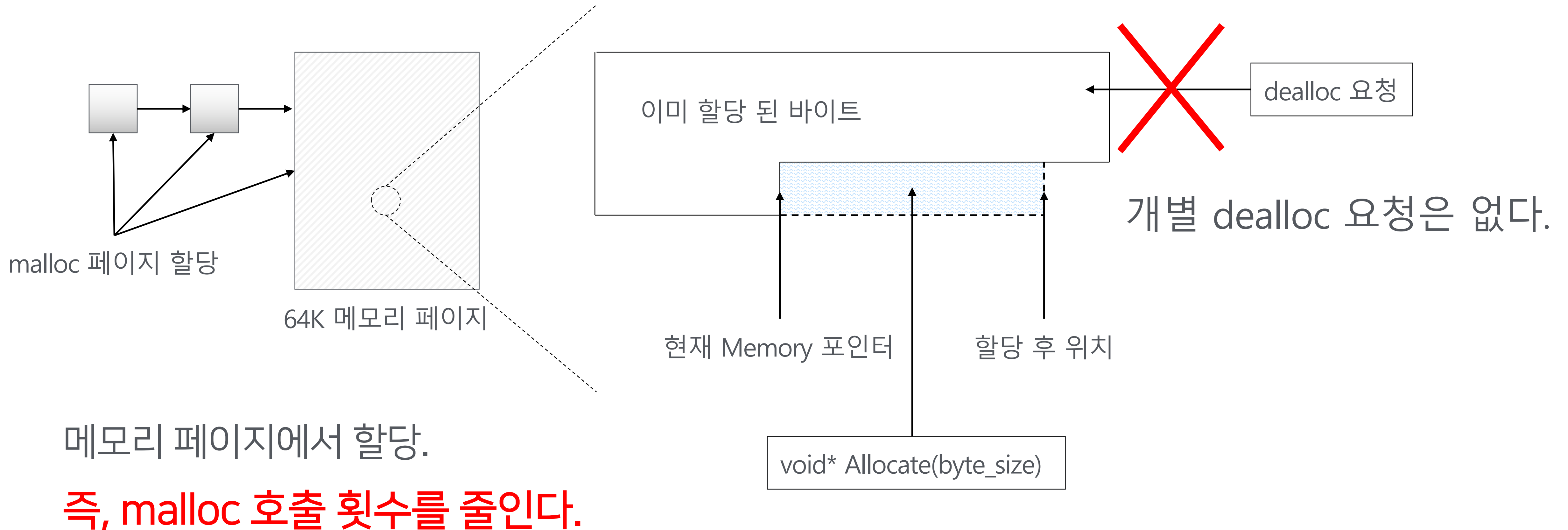
→ 가변 길이 → 빈번한 버퍼 더블링 → 빈번한 malloc, dealloc

! 자바보다도 훨씬 느리다! (최대 1.5배 차이)
• 특히 멀티 스레드에서.

```
BytesRef result = new BytesRef(prefix.length + output.length);  
System.arraycopy(prefix.bytes, prefix.offset, result.bytes, 0, prefix.length);  
System.arraycopy(output.bytes, output.offset, result.bytes, prefix.length, output.length);  
result.length = prefix.length + output.length;
```

Java를 C++로 옮긴다고 해서
무조건 빨라지지 않는다.

6.5 Linear allocator



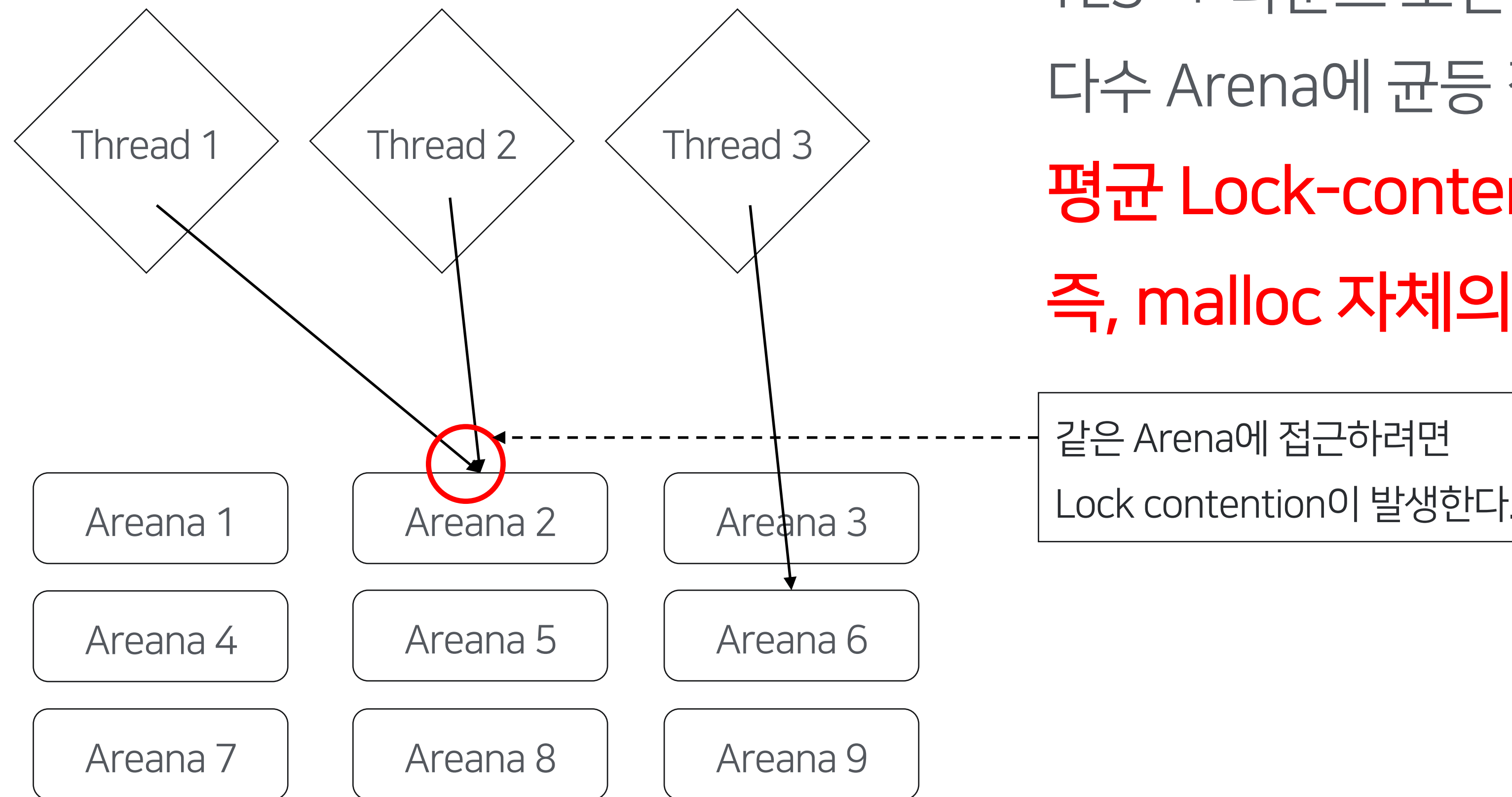
6.6 jemalloc

TLS¹ + 라운드 로빈 방식으로

다수 Arena에 균등 접근하도록 하여

평균 Lock-contention 횟수를 최소화 한다.

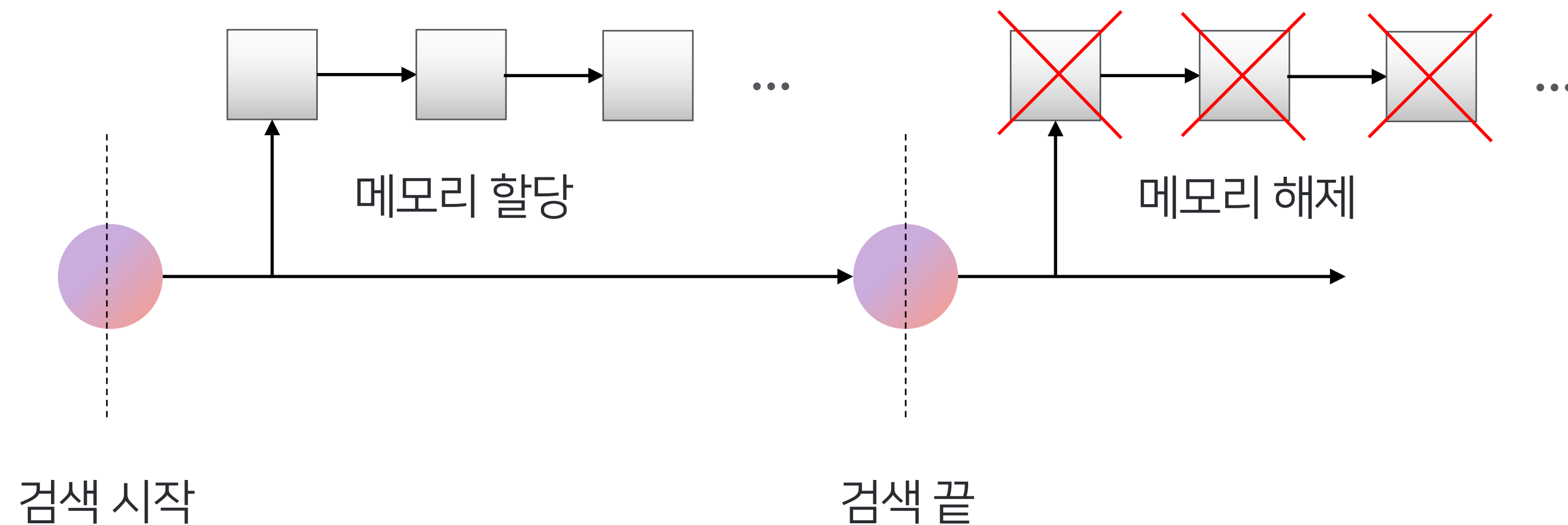
즉, malloc 자체의 성능을 높인다.



1. TLS = Thread local storage

<https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf>

6.7 Linear allocator, NGINX



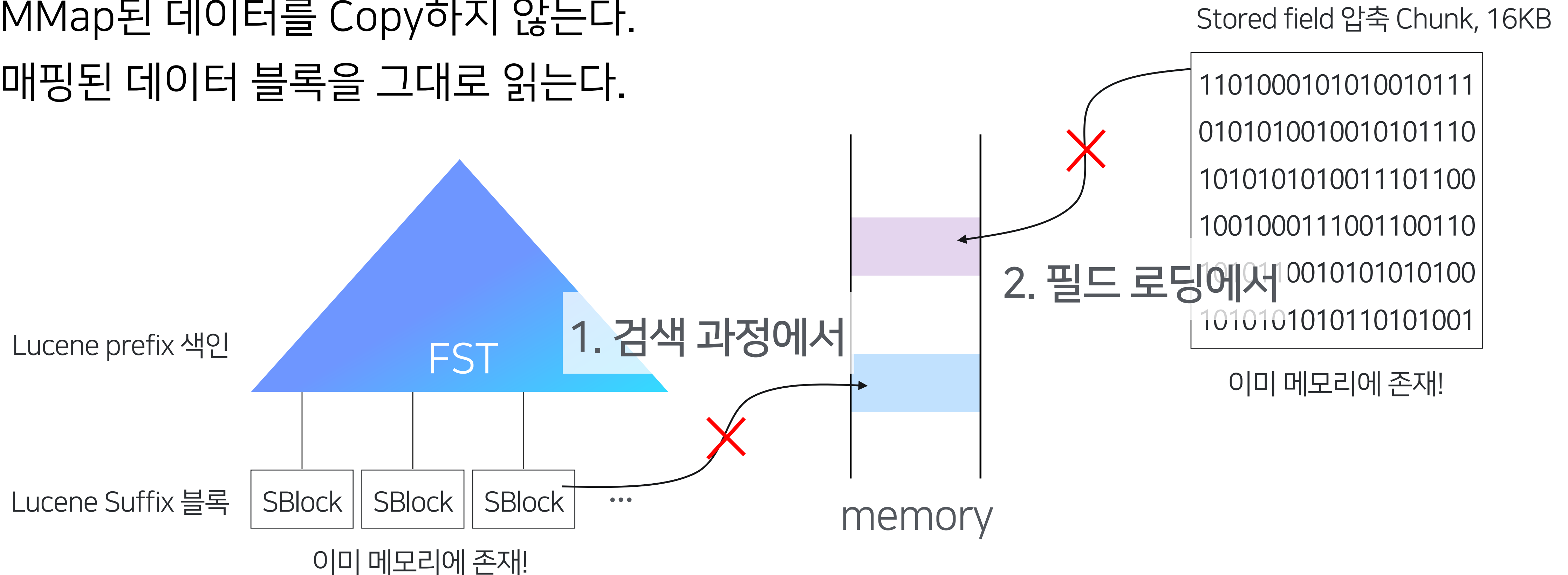
검색 과정의 모든 할당은
페이지에서 이루어지고
검색 종료 후 페이지들이 해제된다.

NGINX palloc 소스 사용. Many thanks to NGINX!

https://github.com/nginx/nginx/blob/master/src/core/nginx_palloc.c

6.8 Address forwarding

MMap된 데이터를 Copy하지 않는다.
매핑된 데이터 블록을 그대로 읽는다.

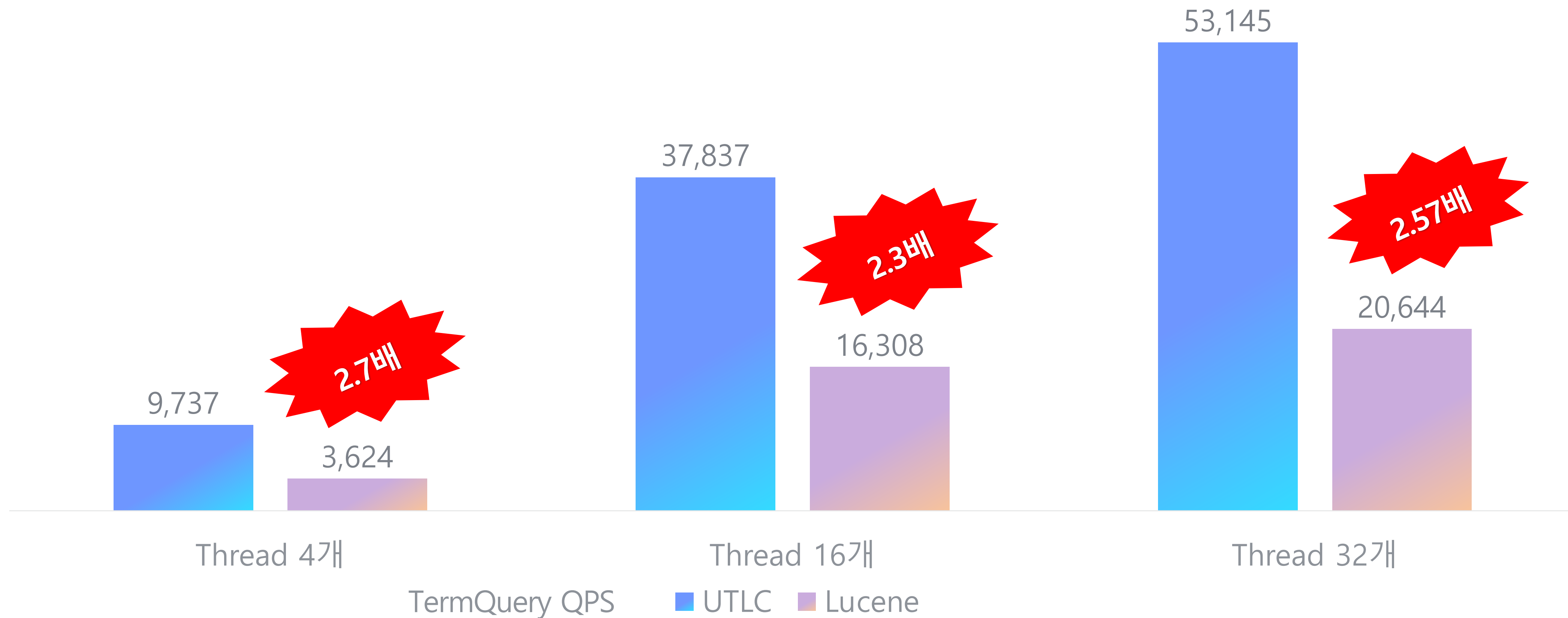


6.9 Address forwarding

- <https://github.com/apache/lucene-solr/blob/releases/lucene-solr/7.6.0/lucene/core/src/java/org/apache/lucene/codecs/blocktree/SegmentTermsEnumFrame.java#L173>
- <https://github.com/apache/lucene-solr/blob/releases/lucene-solr/7.6.0/lucene/core/src/java/org/apache/lucene/codecs/compressing/CompressingStoredFieldsReader.java#L210>

6.10 TermQuery 성능 평가

DEVIEW
2019

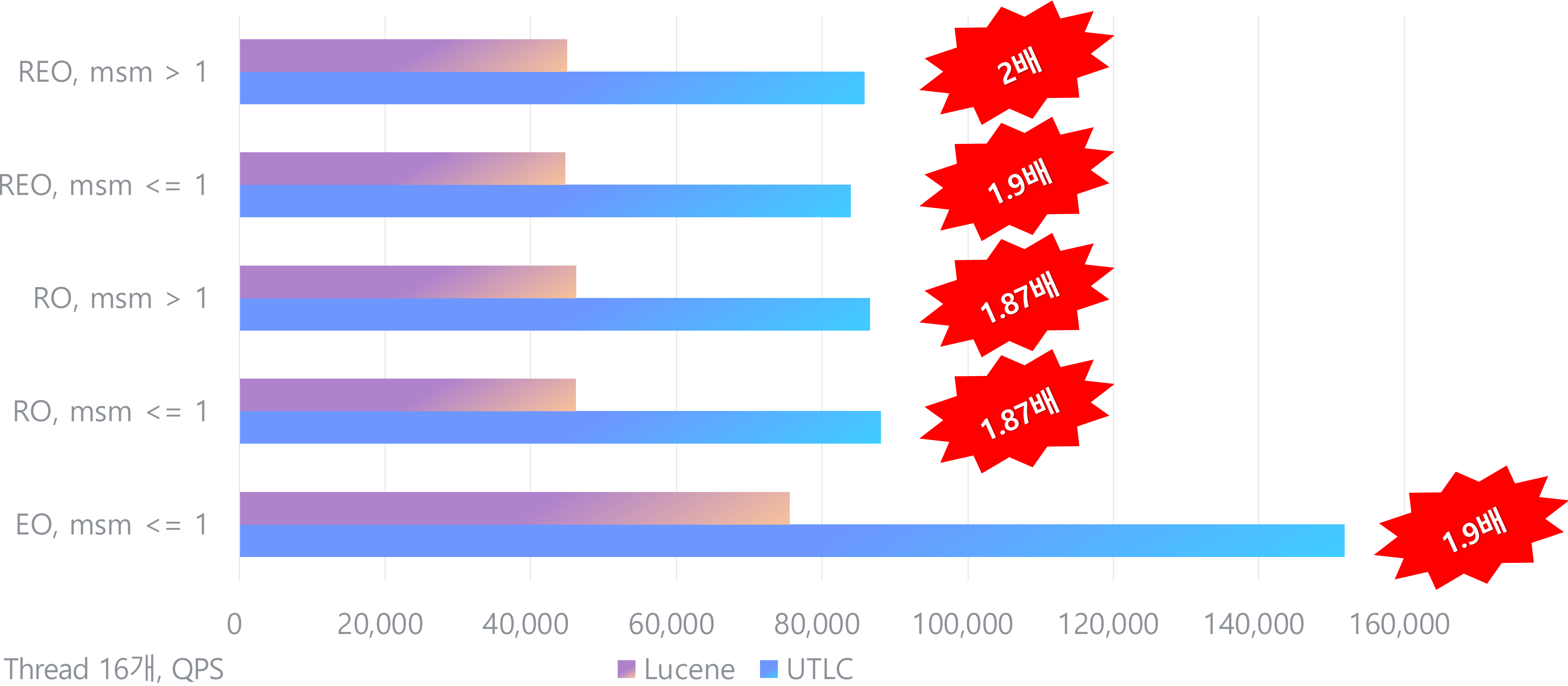


6.11 BooleanQuery 성능 평가

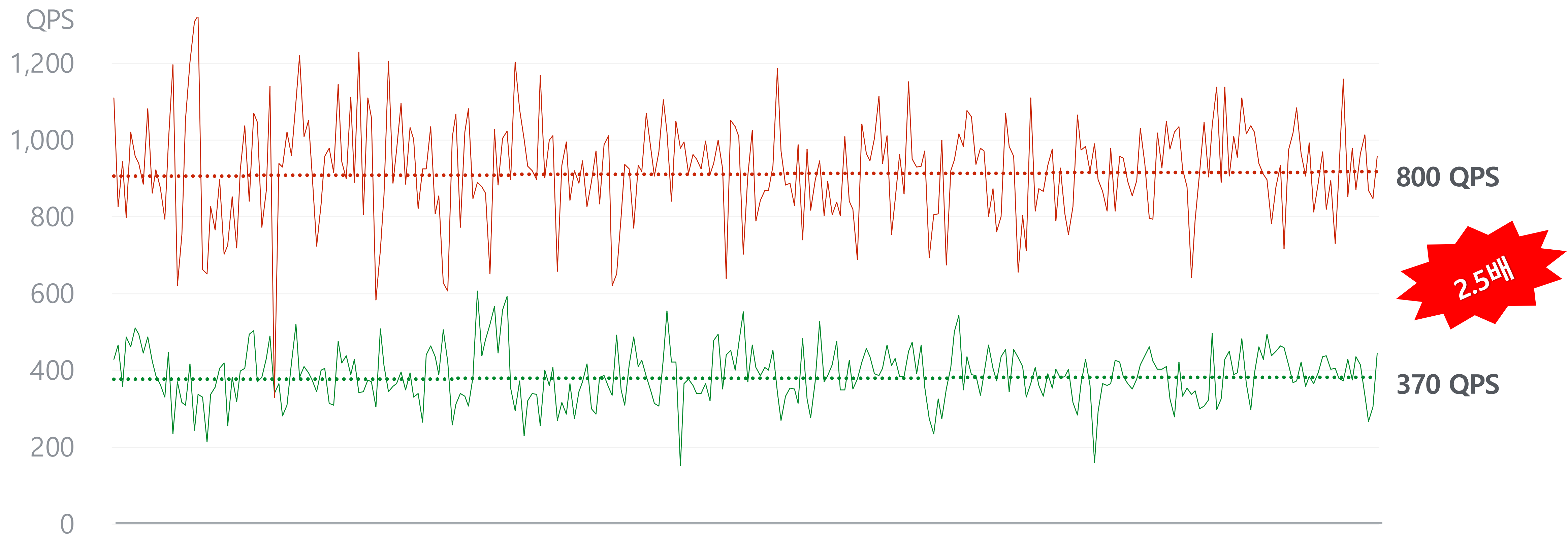
DEVVIEW
2019



6.11 BooleanQuery 성능 평가



6.12 리얼 서비스 시뮬레이션

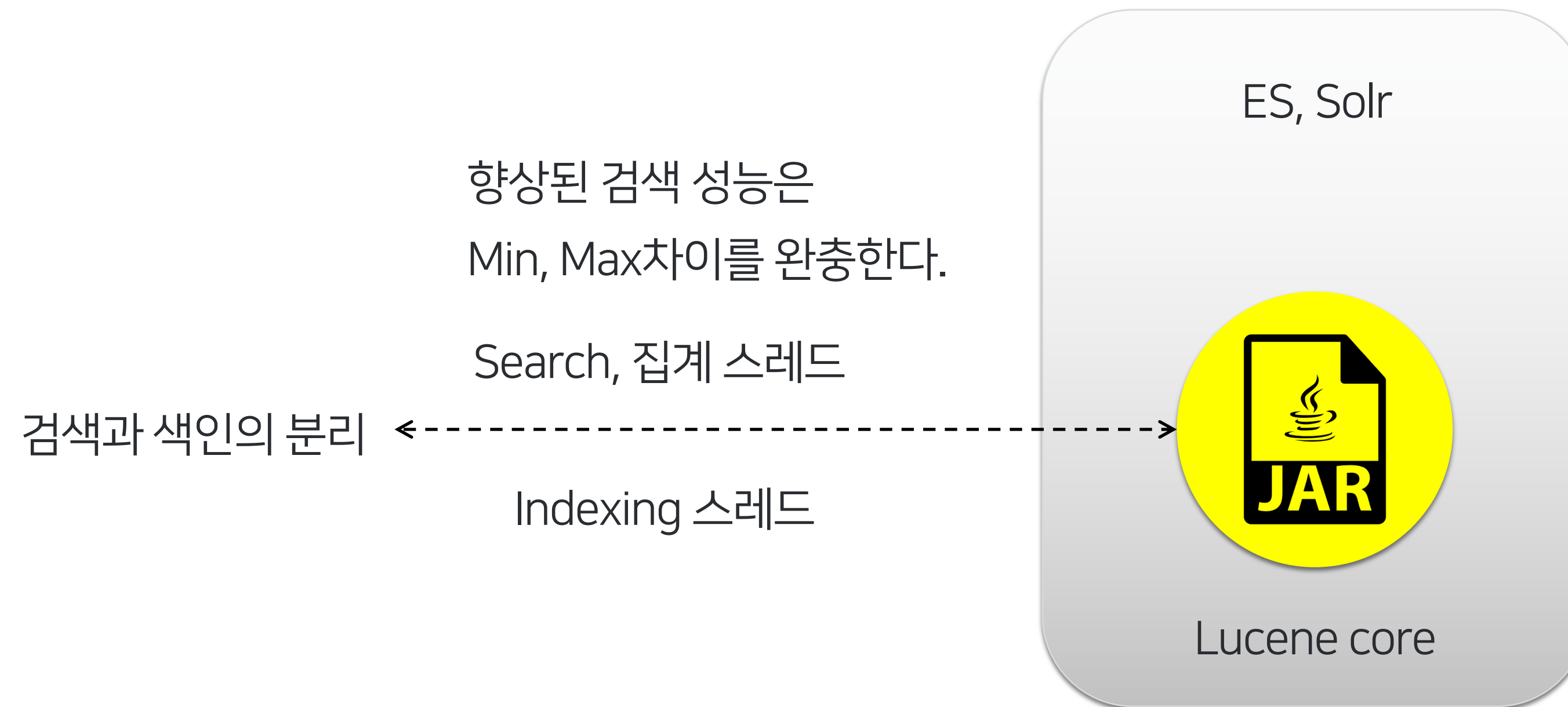


Thread 10개, 시간에 따른 QPS 변화 — Lucene — Ultimate Lucene ...지수 (Lucene) ...지수 (Ultimate Lucene)

Conj, msm > 1

6.13 문제 정의를 되돌아 보자

대량의 트래픽 동시 검색 + 색인이 힘들다. 안되는 것은 아니지만.
이상적으로는 **GC가 없고 검색 성능이 더 빠르다면**
더 이상 동료들이 고통 받는 일은 없을 것이다.



6.14 UTLC는 문제를 해결했는가?

- 2배 검색 성능
- 검색과 색인의 분리
- GC 없음
- 집계 엔진은?
- **어떻게 하면 검색엔진까지 연결 시킬 수 있을까?**



엔진은 일단 완성. Next move?

22번에서
당신을 기다려요!



지금까지, 그리고 이 후

Q & A

Thank You