Redis

dict / sds structures
GET / SET command flow

2018. 02. 02
김도영

SW STAR LAB
Software Technology Advanced Research

과제명: IoT 환경을 위한 고성능 플래시 메모리 스토리지 기반 인메모리 분산 DBMS 연구개발
과제번호: 2017-0-00477

과학기술정보통신부
Ministry of Science and ICT

연세대학교
YONSEI UNIVERSITY

정보통신기술진흥센터
Institute for Information & communications Technology Promotion

Contents

- *dict* analysis
  - dict structures
  - Add / Remove key
  - Expand / Rehash (Incremental)
- *sds* analysis
  - SDS?
  - Strengths / Weaknesses
  - Structure
  - Useful functions
- GET / SET command flow

Dict analysis

Implemented in *dict.h / dict.c*

- Redis의 K/V Hash table 구조체가 구현되어 있음.
  - Redis data storage
  - Redis HM types

- Dynamic hashing을 사용함.
  - 2개의 hash table을 사용하여 구현.
  - Collision
    - linear chaining
  - Rehash
    - Expandable (Similar!)
    - Incremental

Dict structures

- dictEntry
  - hash table에 저장되는 entry
  - key, value, next pointer를 가지고 있음
- dictht
  - dictEntry를 저장하는 hash table
  - size, used 정보도 저장함
- dict
  - hash table을 관리하는 구조체
  - 2개의 dictht를 가지고 있음
    - rehash

Dict structures

- dictType
  - dict의 polymorphism을 구현함
  - dict에서 사용하는 function들을 override

- dictIterator
  - dict(hash table) iteration를 담당 (SCAN)
  - safe / unsafe

dictEntry

Data Type	Name	Description
void*	key	키
(union) void* uint64_t int64_t double	val u64 s64 d	값
dictEntry*	next	(Chaining) Hash 값 충돌 시 다음 entry pointer

dictht

Data Type	Name	Description
dictEntry**	table	(Bucket) Entry pointer array
unsigned long	size	Bucket size
unsigned long	sizemask	Bucket Size에 맞는 hash key를 bitwise 추출하기 위한 mask
unsigned long	used	Used entry size

dict

Data Type	Name	Description
dictType*	type	Dict type (pre-defined)
void*	privdata	-
unsigned long	sizemask	Size에 맞는 hash key를 bitwise추출하기 위한 mask
long	rehashidx	Rehash 해야할 bucket start index Rehash중이 아니면, -1로 set되어있다
unsigned long	iterators	현재 작업중인 ‘safe’ iterator 개수

dictType

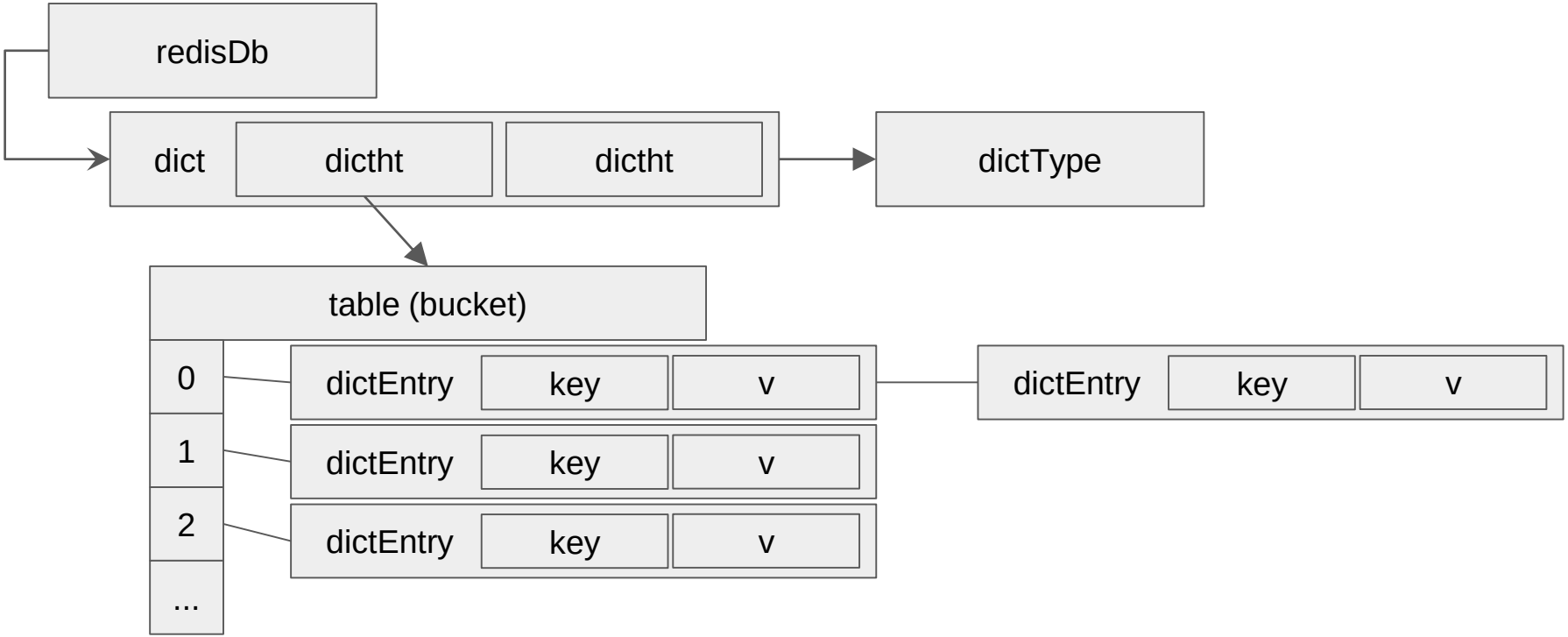
Data Type	Name	Description
uint64_t	*hashFunction	Hash function
void*	*keyDup	Key duplication
void*	*valDup	Value duplication
int	*keyCompare	Key compare function
void	*keyDestructor	Key destructor (zfree)
void	*valDestructor	Value destructor (zfree)

dictType

- dbDictType
- setDictType
- hashDictType
- clusterNodesDictType
- ...etc

- Hash function generator는 “siphash”을 사용함
  - https://en.wikipedia.org/wiki/SipHash

Defined in server.c


redisDb
dict
dictht
dictht
dictType
table (bucket)
0
1
2
...
dictEntry
key
v
dictEntry
key
v
dictEntry
key
v
dictEntry
key
v

Add / Delete key

Add

- dictAdd()에 의해 수행됨
  - dictAddRaw / dictSetVal 로 나누어서 수행함

- dictAddRaw()
  - dict에 key에 해당하는 dictEntry를 생성시키는 function
  - key에 해당하는 bucket index를 구한 다음, bucket chain에 entry를 추가시킴
  - entry만 추가하고 value는 set하지 않음
  - key가 이미 dict 내부에 존재하면, existing을 populate시켜주고 return

- dictSetVal()
  - dictType에 맞는 value duplicate function을 호출함.
    - 거의 default를 사용함. *(entry->v.val = val)*

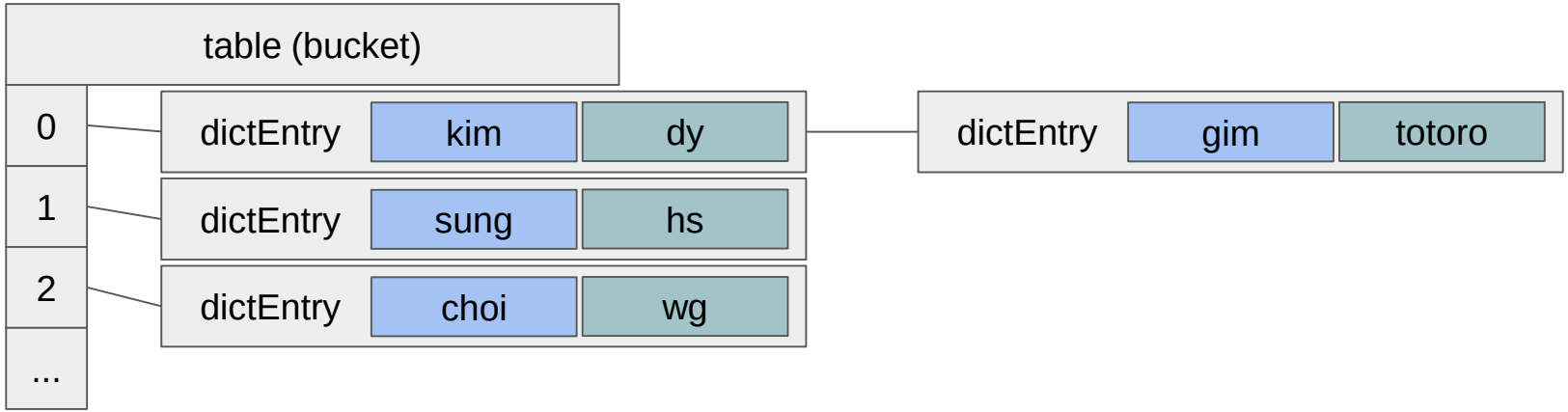
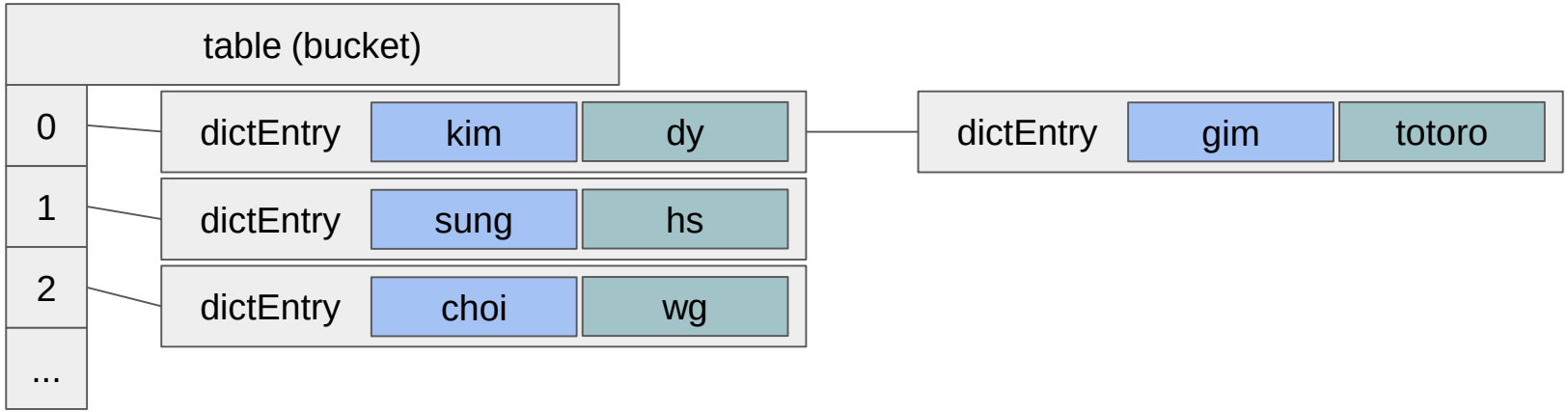


table (bucket)
0
1
2
...
dictEntry
kim
dy
dictEntry
gim
totoro
dictEntry
sung
hs
dictEntry
choi
wg

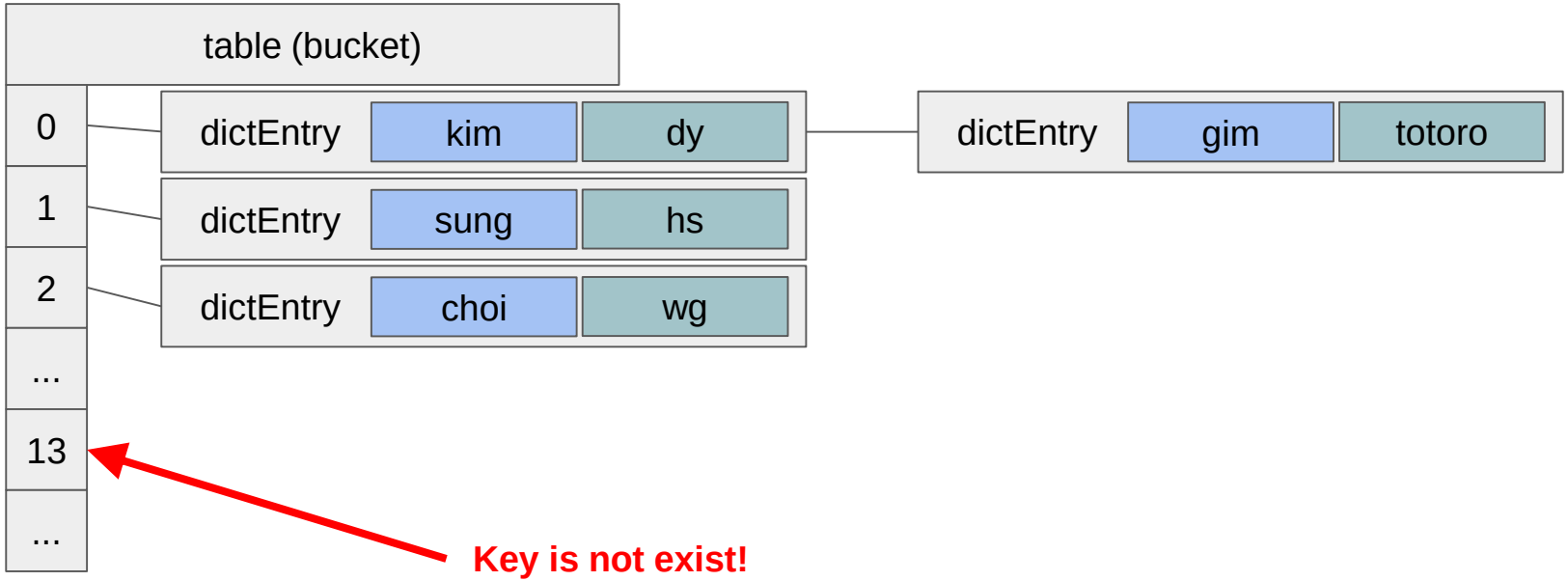
dictAdd(d, “lee”, “jh”)

hash(lee) & sizemask = 13

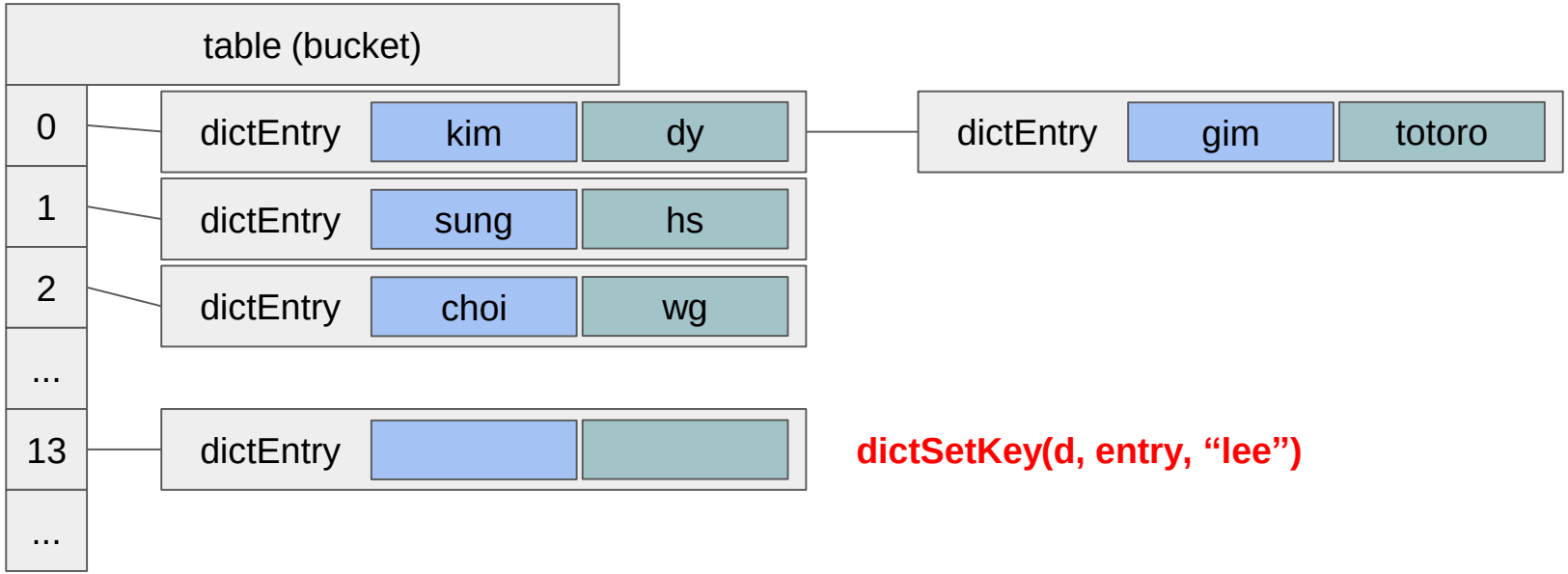


dictAdd(d, “lee”, “jh”)

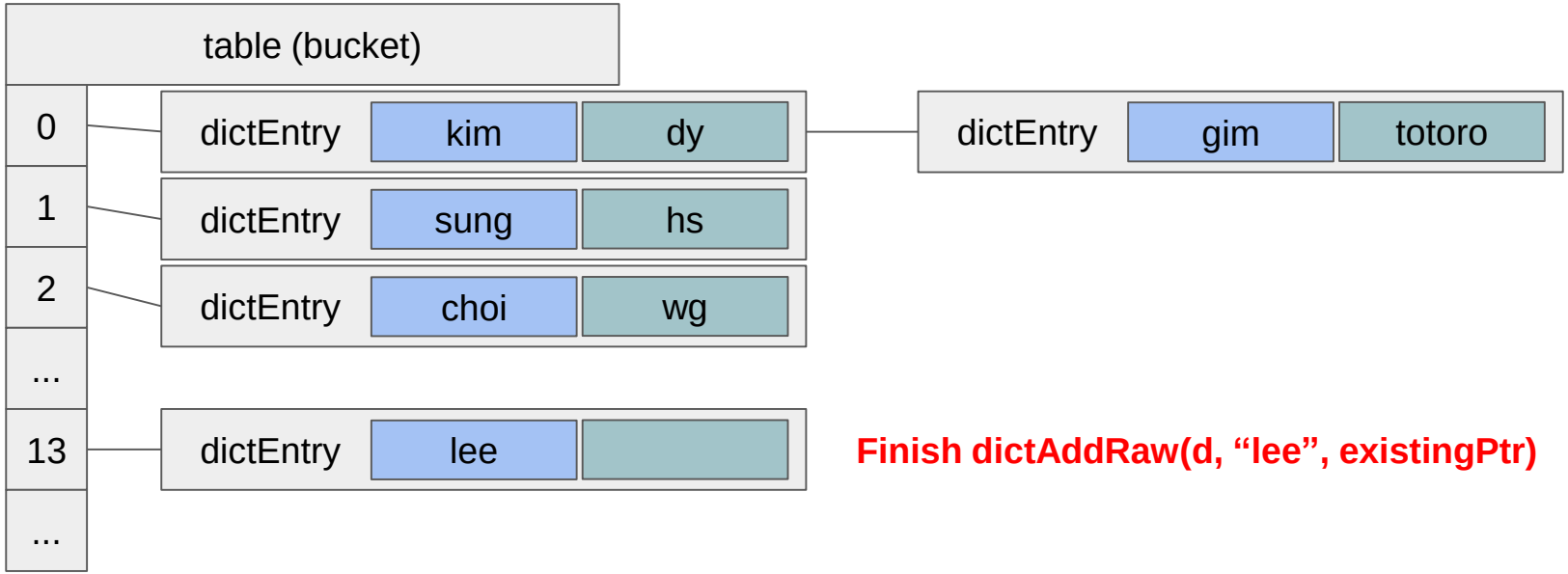
hash(lee) & sizemask = 13



dictAdd(d, “lee”, “jh”)

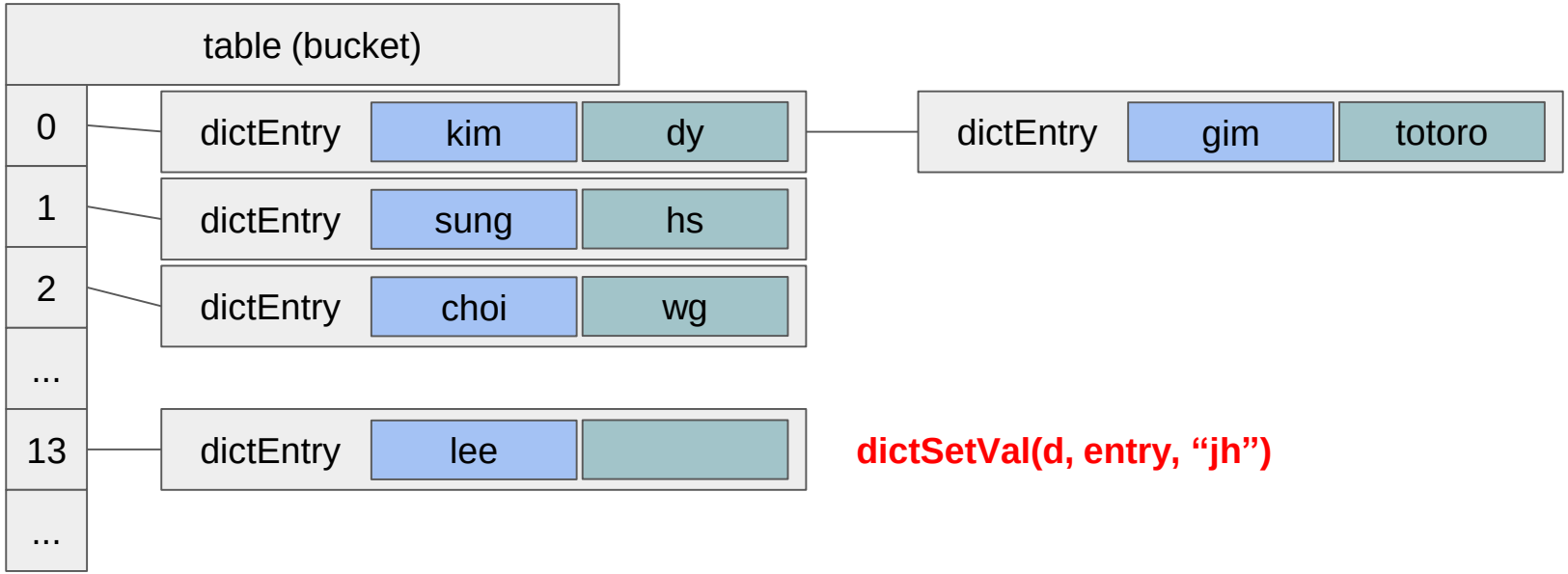


dictAdd(d, “lee”, “jh”)

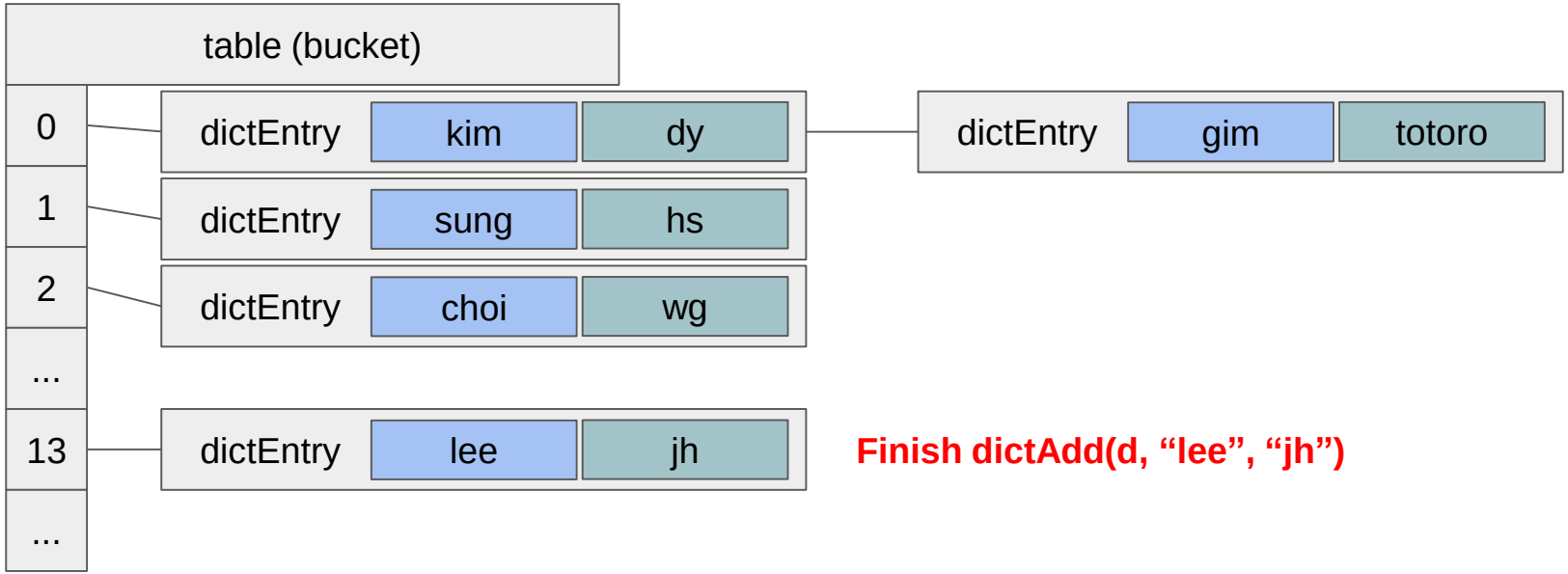


Finish dictAddRaw(d, “lee”, existingPtr)

dictAdd(d, “lee”, “jh”)



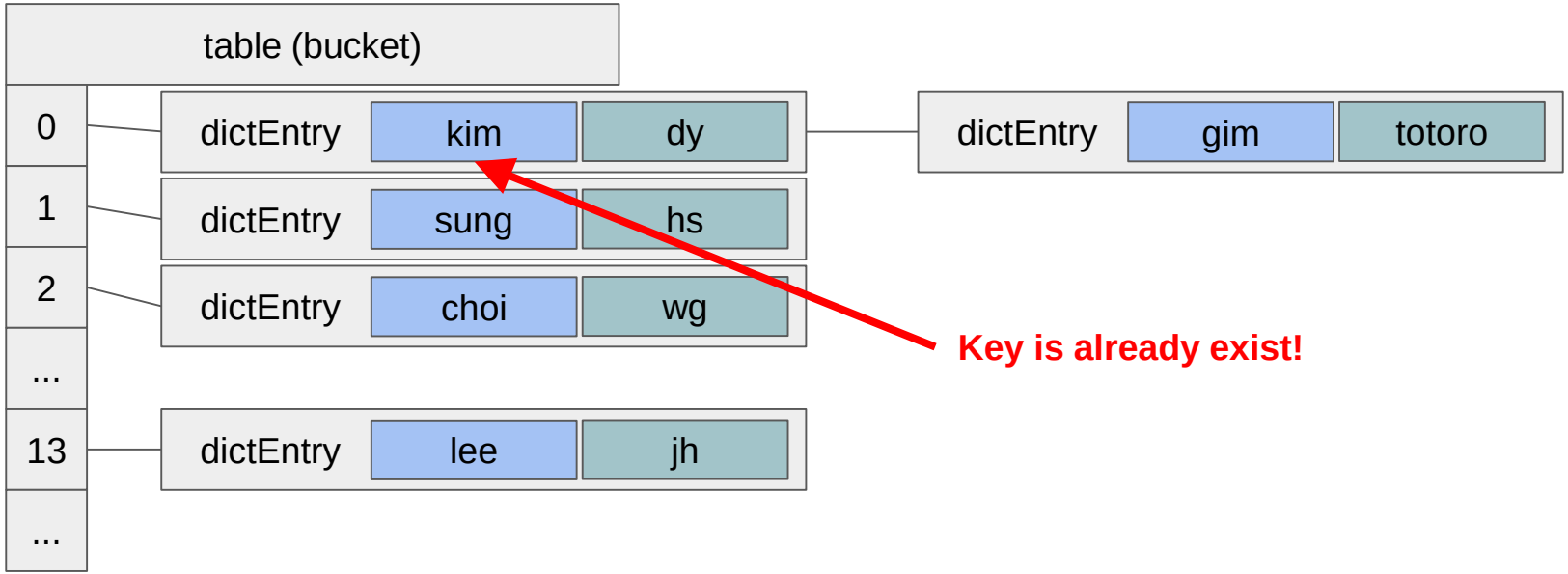
dictAdd(d, “lee”, “jh”)



dictAdd(d, “kim”, “jk”)

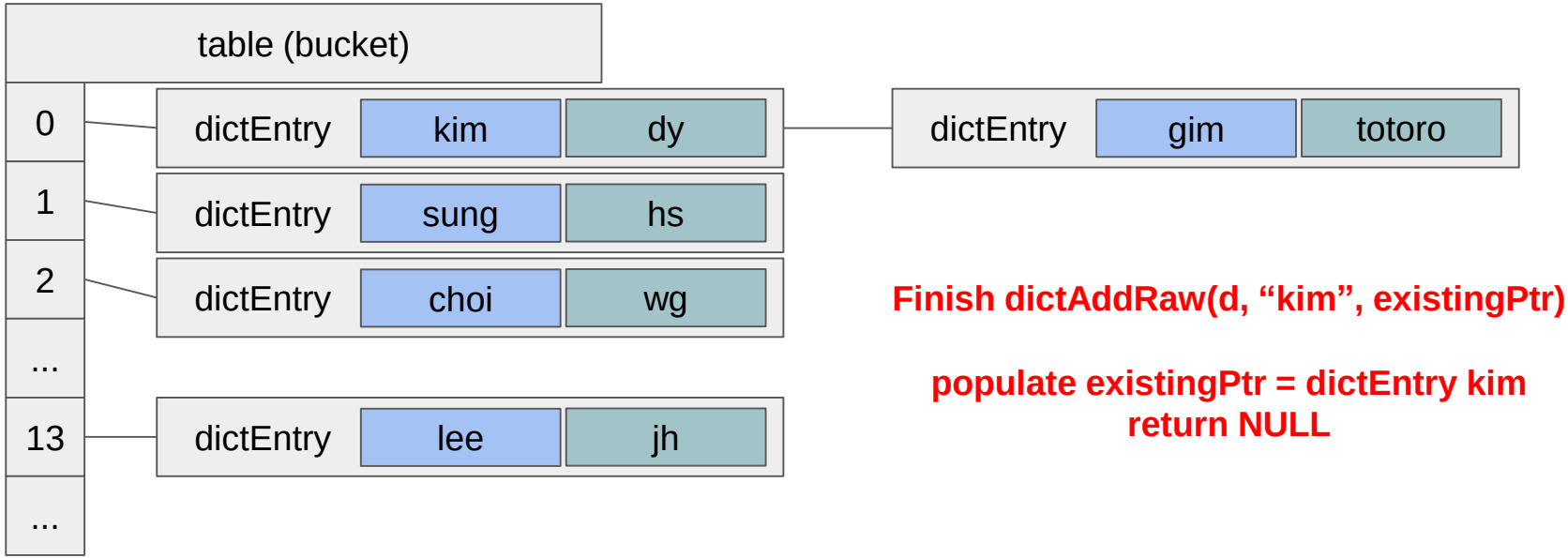
Animation

hash(kim) & sizemask = 0



dictAdd(d, “kim”, “jk”)

hash(kim) & sizemask = 0



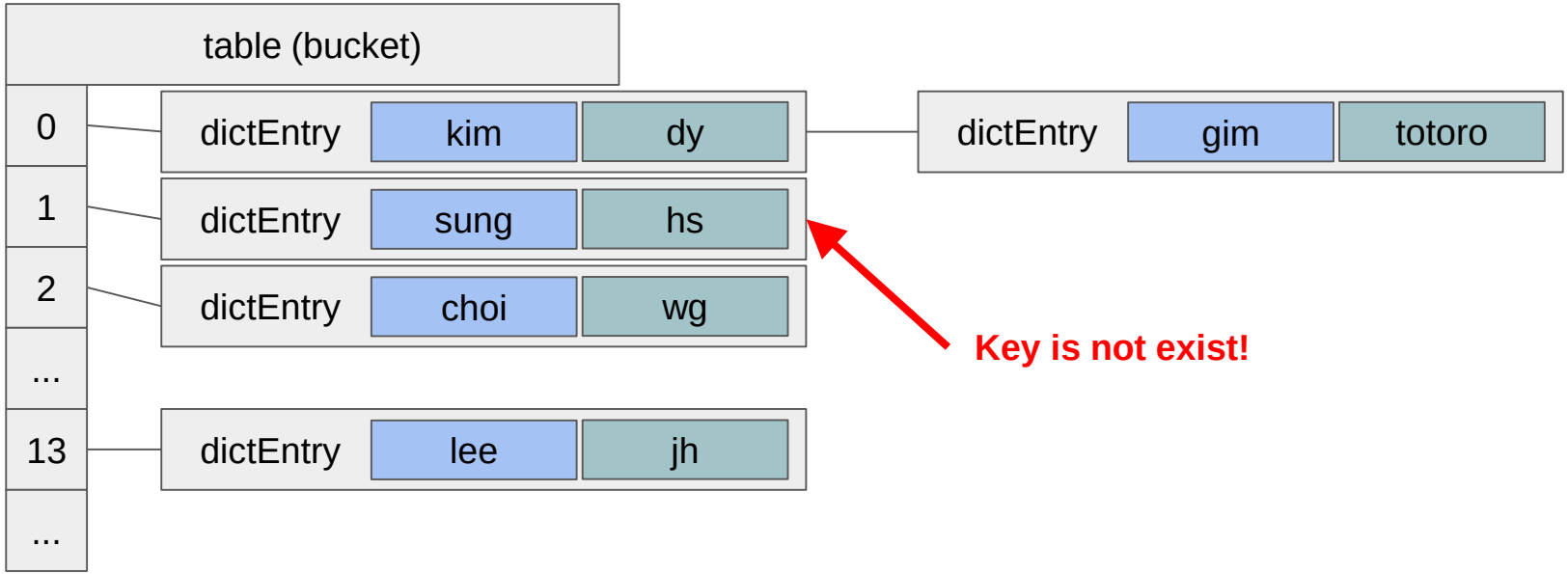
Finish dictAddRaw(d, “kim”, existingPtr)

populate existingPtr = dictEntry kim
return NULL

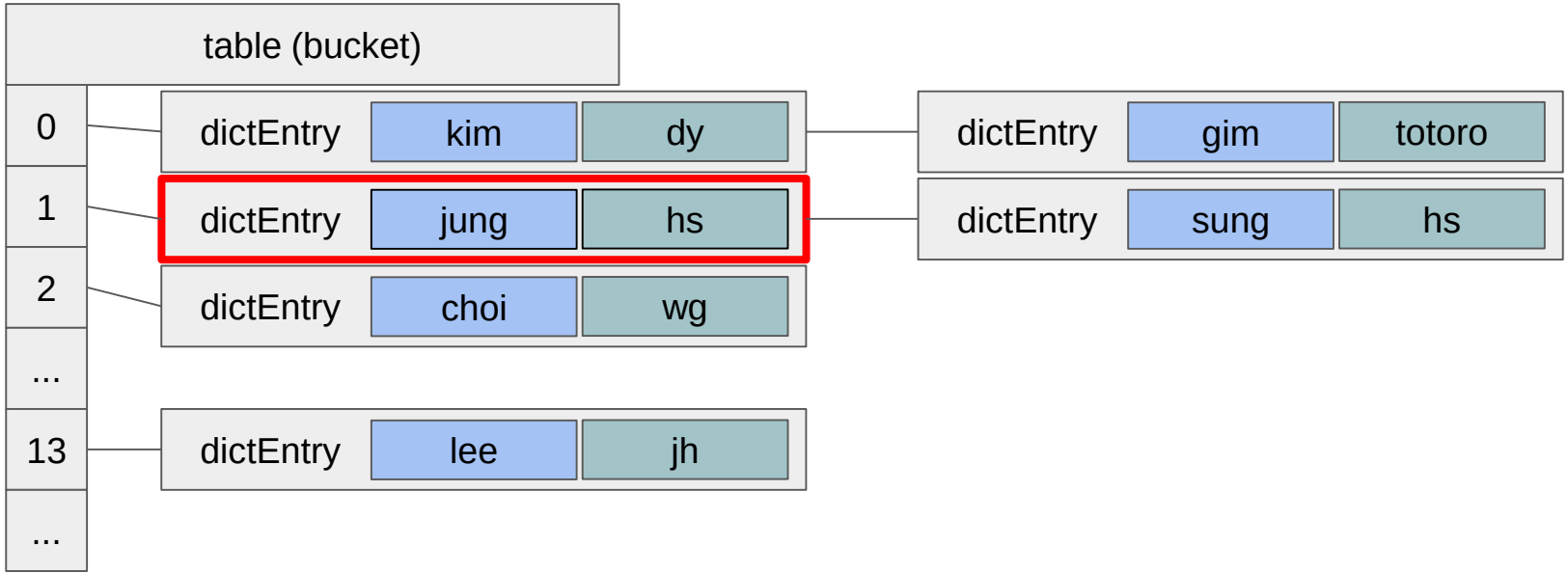
dictAdd(d, “jung”, “hs”)

Animation

hash(jung) & sizemask = 1



dictAdd(d, “jung”, “hs”)



Delete

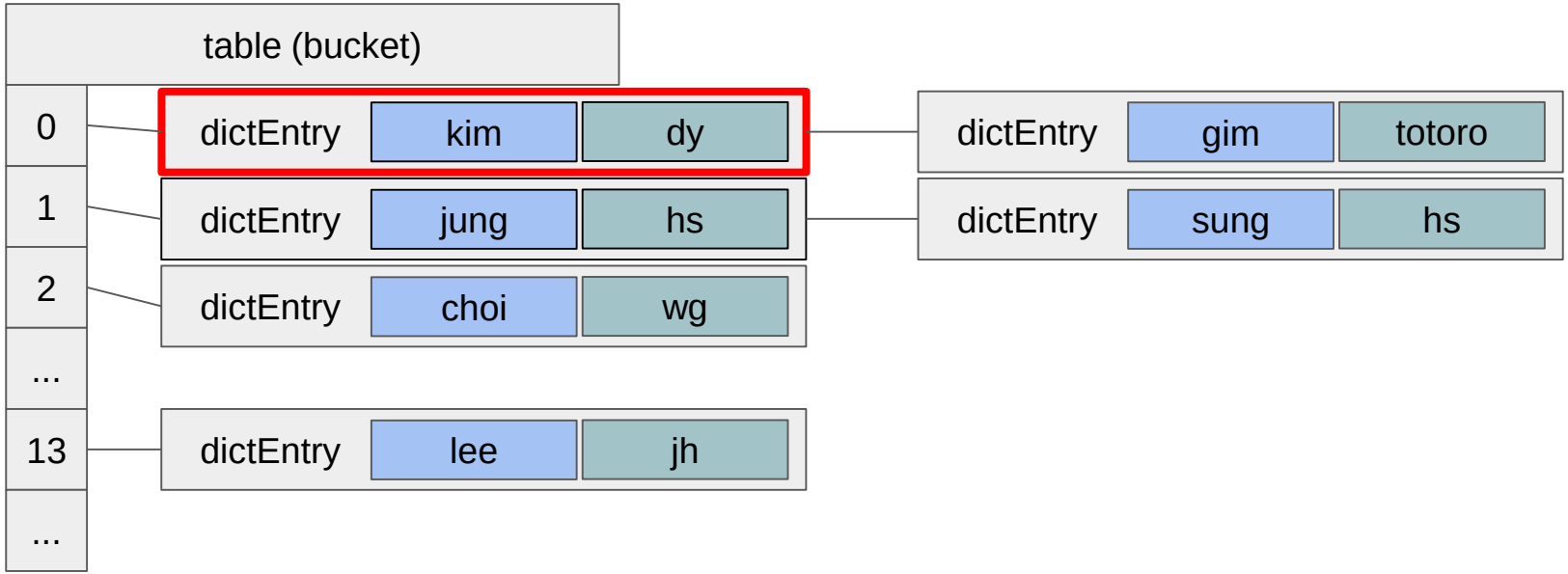
- dictDelete()에 의해 수행됨

- dictUnlink()
  - dictEntry를 메모리에서 날리지 않고, bucket table에서만 지움
  - 이점
    - 다음의 상황에서 lookup을 2번하는걸 방지할 수 있다.

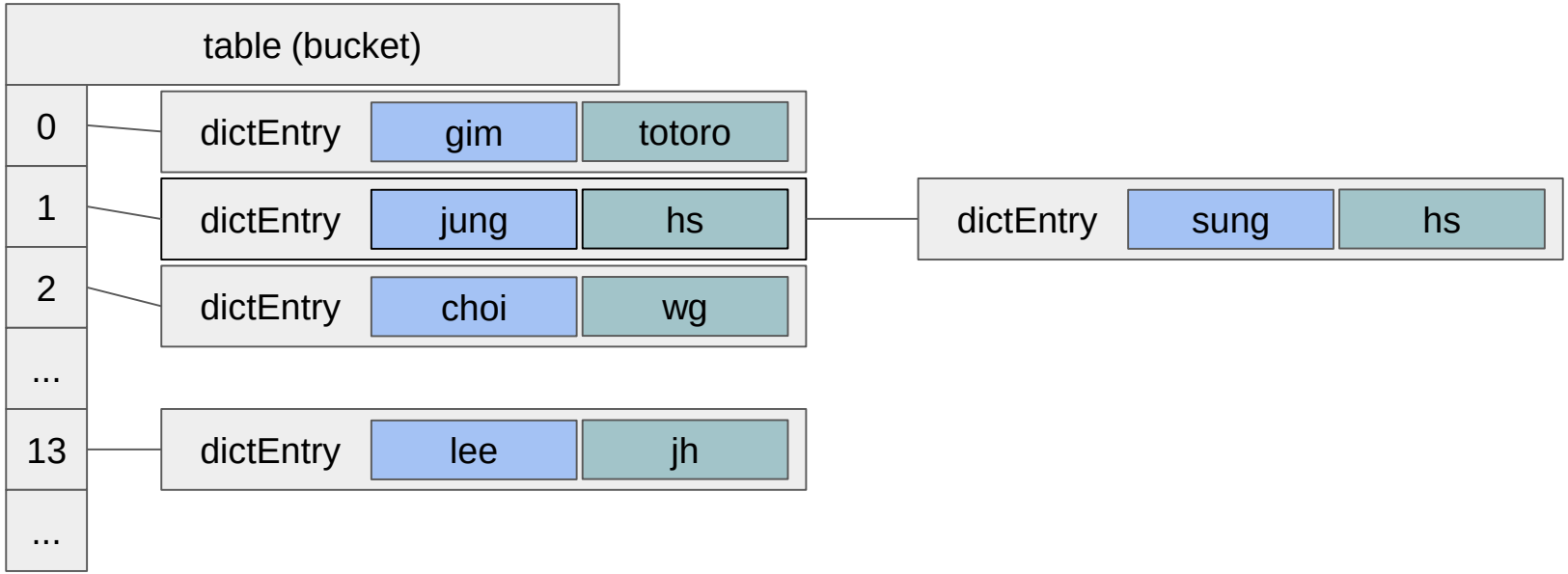
```
entry = dictFind(...);
// Do something with entry
dictDelete(dictionary,entry);

entry = dictUnlink(dictionary,entry);
// Do something with entry
dictFreeUnlinkedEntry(entry); // <- This does not need to lookup again.
```

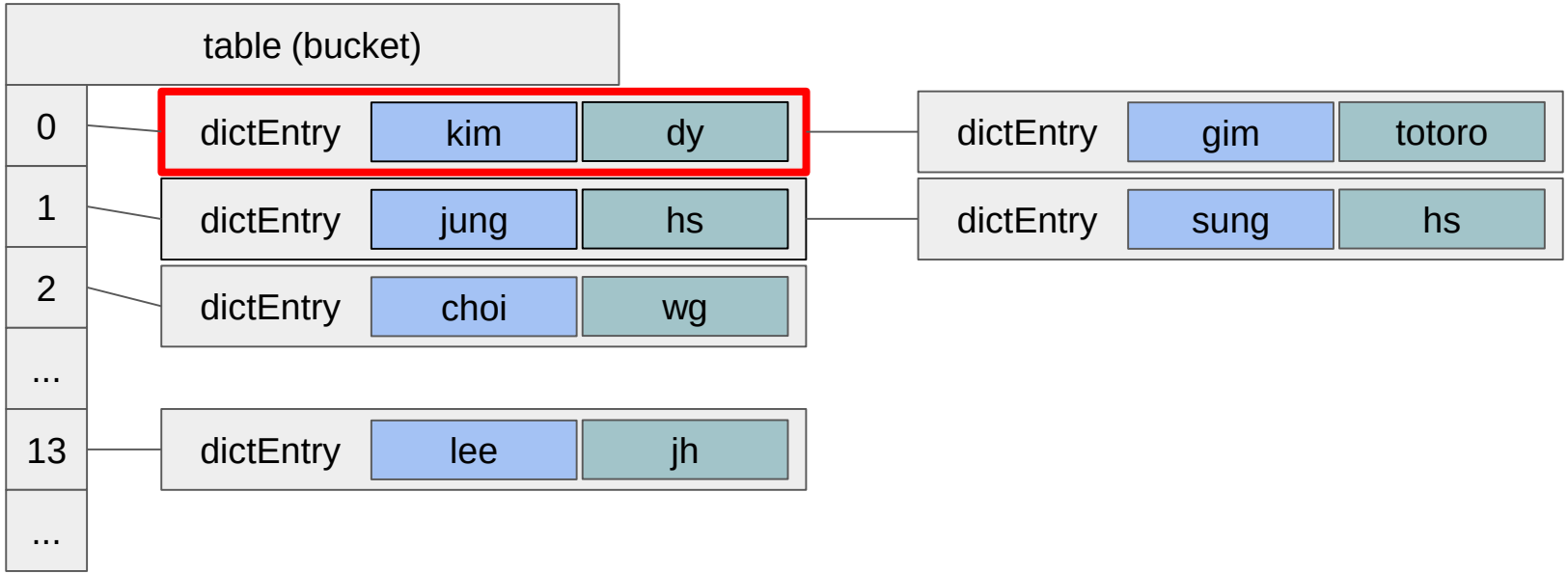
dictDelete(d, “kim”)



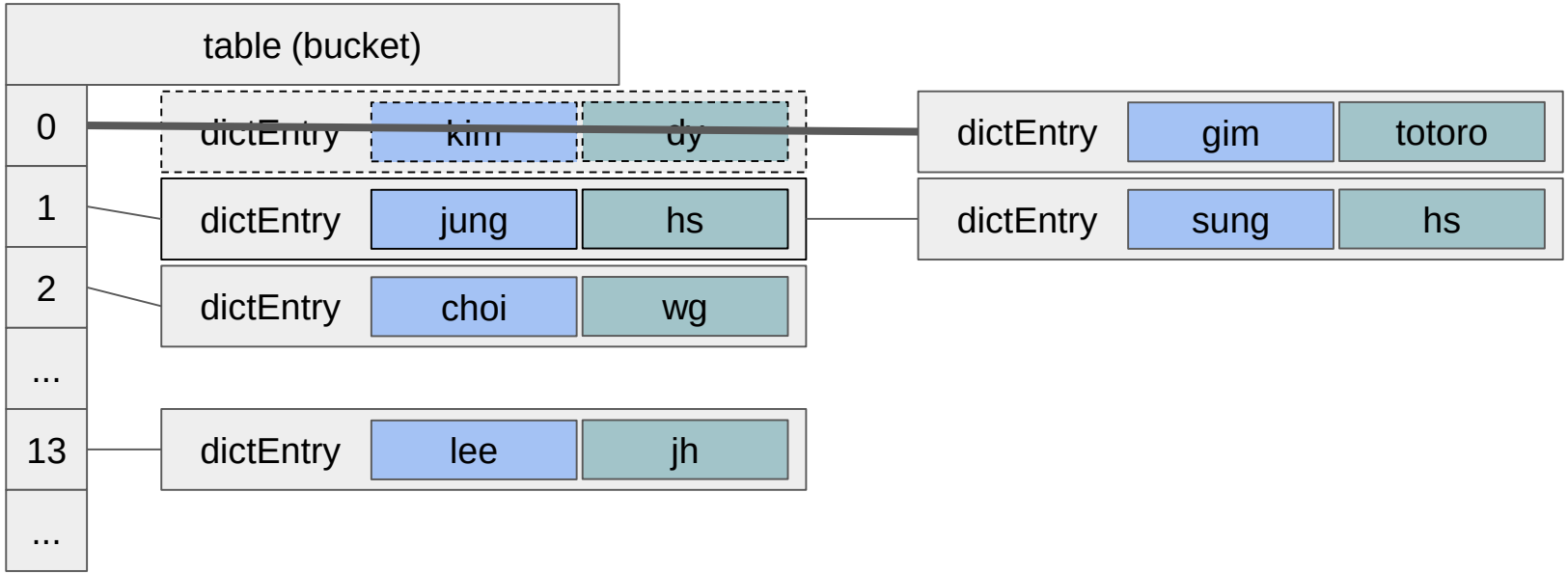
dictDelete(d, “kim”)



dictUnlink(d, “kim”)

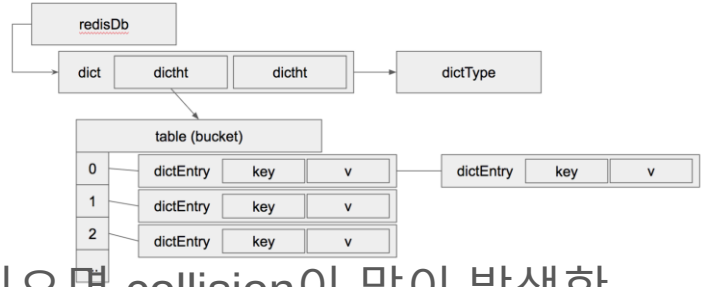


dictUnlink(d, “kim”)



Expand / Rehash

Expand / Rehash

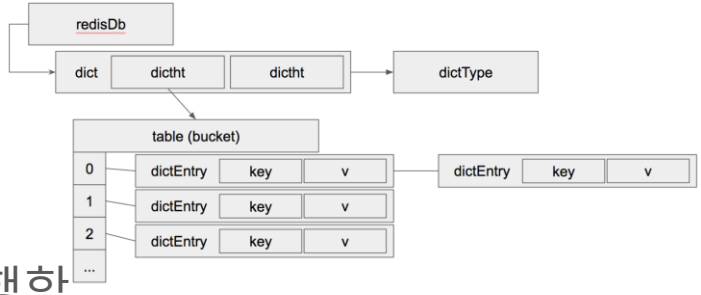


- Key를 Add할 때, hash bucket 사이즈가 정해져 있으면 collision이 많이 발생함
  - Linear Chaining이 많이 생기게 됨
  - Find / Add / Delete load ↑

- Bucket table보다 사용한 entry 개수가 더 많아지면 table을 expand
  - Find key operation할 때마다 check
  - ```
    if (d->ht[0].used >= d->ht[0].size &&
        (dict_can_resize ||
         d->ht[0].used/d->ht[0].size > dict_force_resize_ratio))
    {
        return dictExpand(d, d->ht[0].used*2);
    }
    ```

- Bucket table size는 2^n으로 늘어남
- Rehash를 trigger할 수 있게 준비함. *(rehashidx = 0)*
    ```

Expand / Rehash



- Rehash는 redis server에서 주기적(1ms)으로 실행함
  - ht[1]이 Expand 되었을때만 rehash가 가능함
  - 한꺼번에 rehash하지 않고 조금 조금씩 rehash
  - incremental rehash

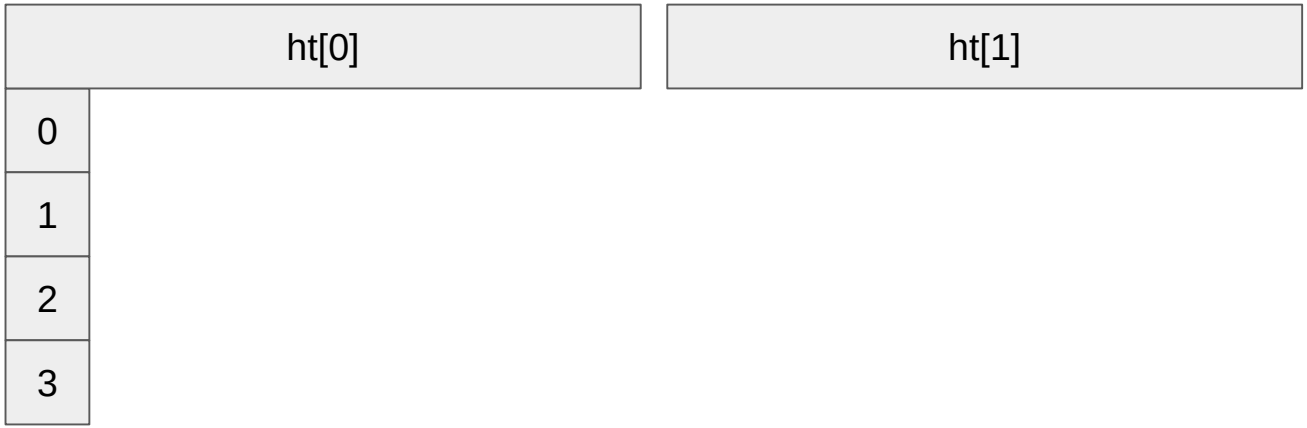
- (Rehash method) dict는 2개의 Bucket table을 가지고 있음
  - Rehash가 일어나지 않는 이상, *ht[0]*에서 entry를 find
  - Expand는 *ht[1]*의 size를 2배로 늘림
  - Rehash는 Expand된 이후에 *ht[0]*의 entry address를 *ht[1]*로 다시 hash function을 수행하여 hash함.

Incremental rehash

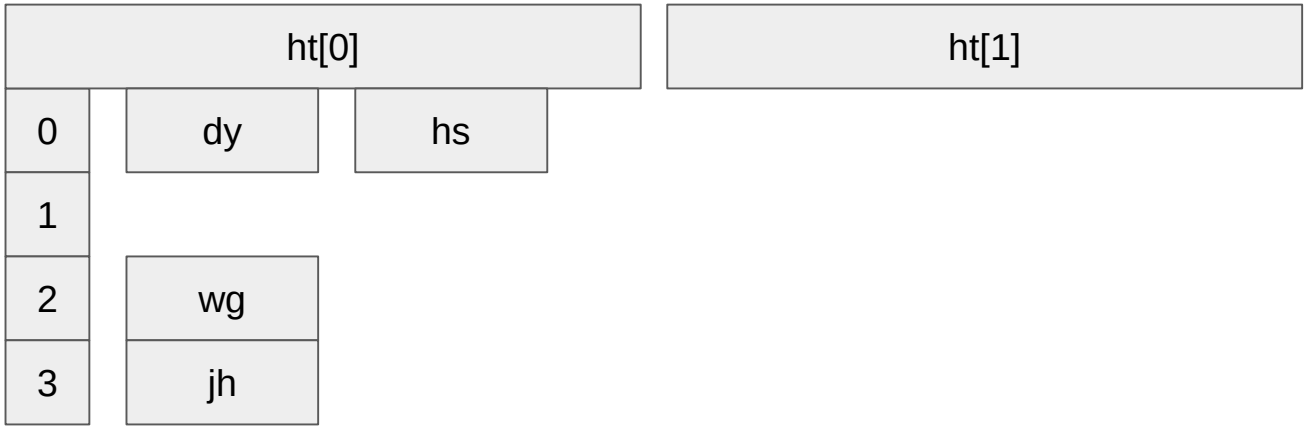
- Rehash 작업을 한꺼번에 진행하지 않고, 일부만 진행한 뒤 잠시 멈춰놓는 방법
  - ht[0]에 남는 entry 없이 Hash 작업이 모두 완료되면 return 0
  - ht[0]에 남은 entry가 있으면 return 1

- Rehash중인 dict는 ht[0]와 ht[1] 모두 사용한다.
  - **Find:** ht[0], ht[1] 모두 조회
  - **Add:** ht[1]에 작성

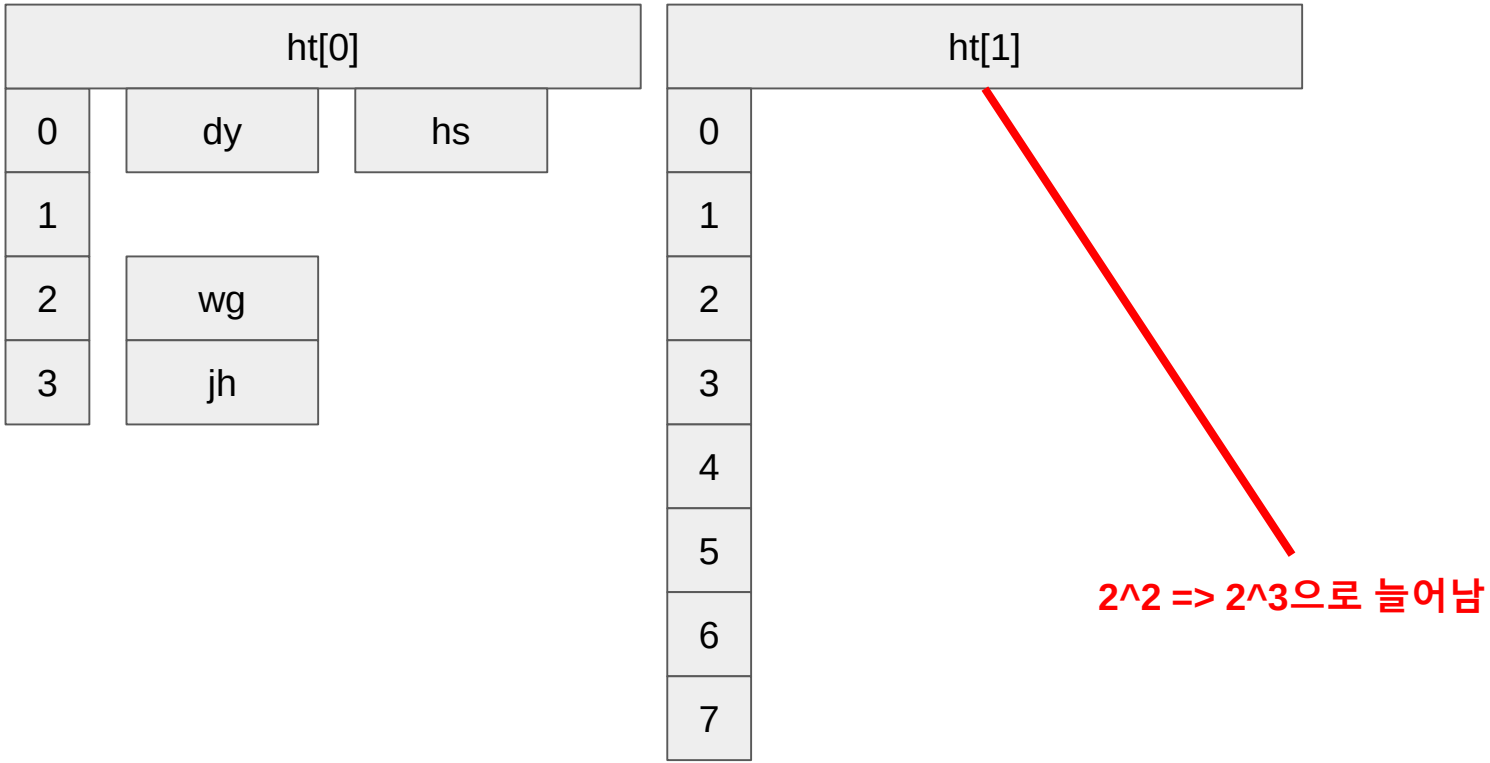
- Rehash가 완료되면 ht[1]을 ht[0]로 옮긴다.
  - Rehash중이지 않을때는 ht[0]만 사용하도록 함. 코드 통일성.
  - address coping이기 때문에 memory load가 거의 없음.

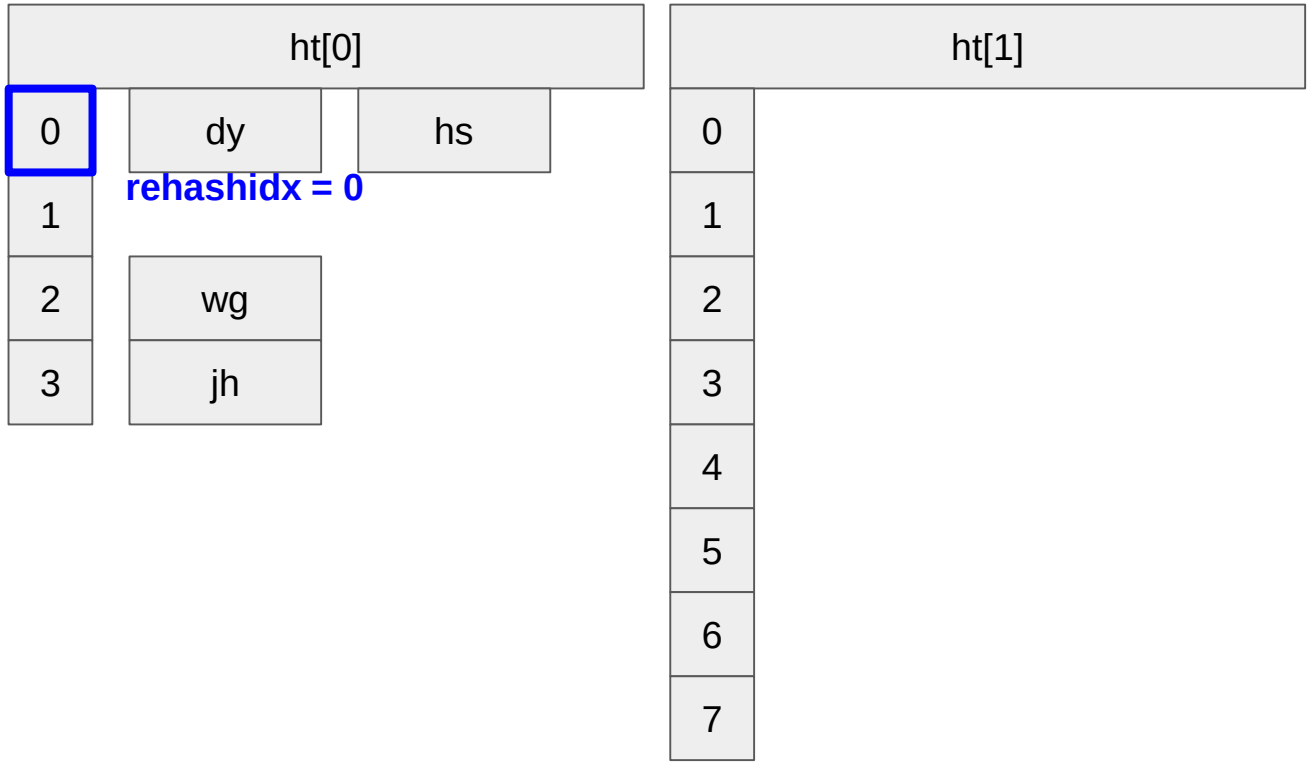


DICT_HT_INITIAL_SIZE = 4

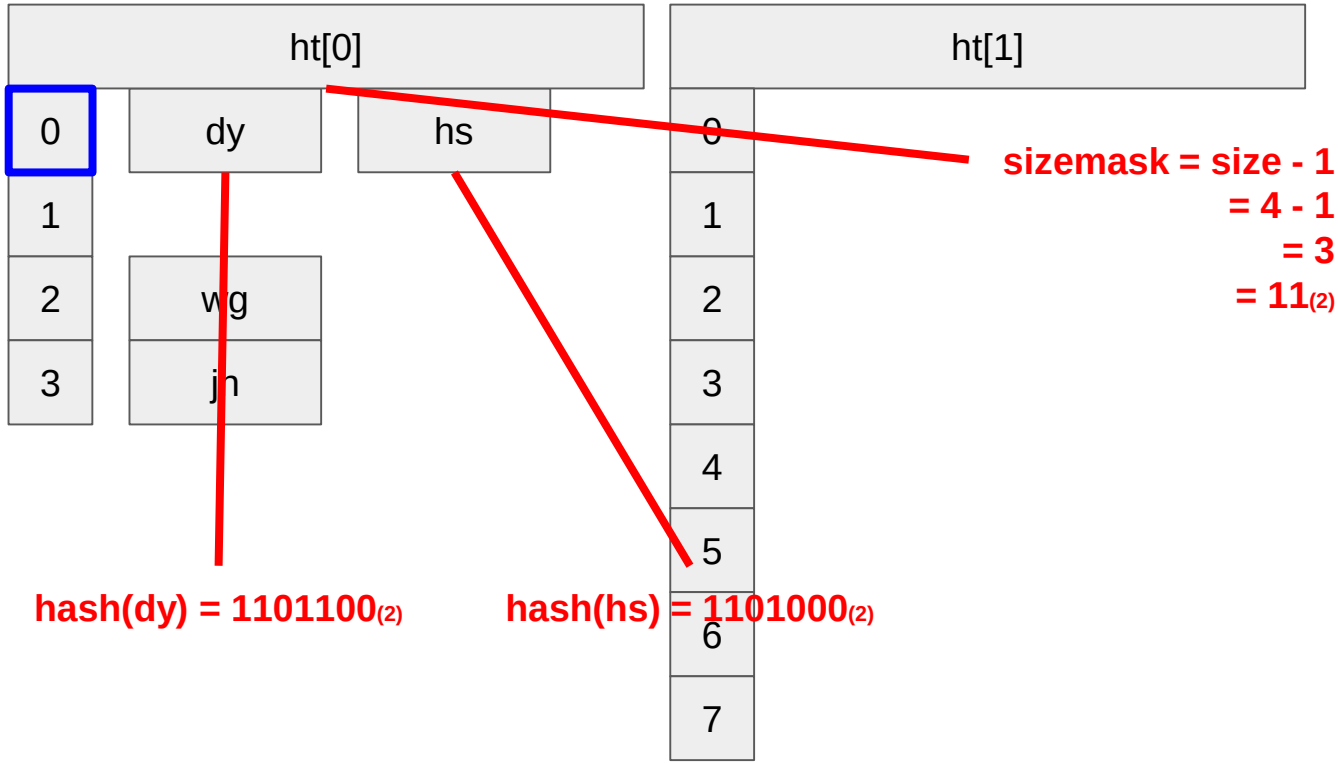


Add entries...

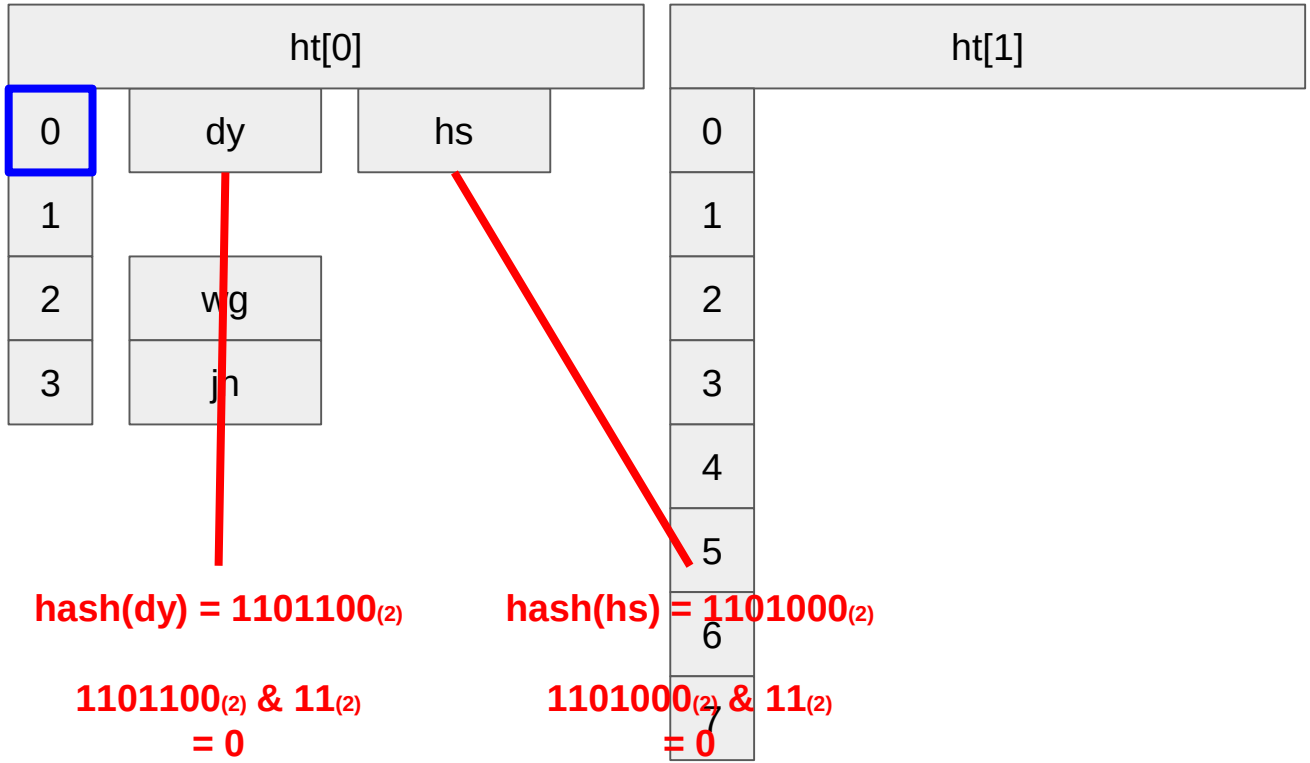

ht[0]
0
dy
hs
1
2
wg
3
jh
ht[1]
0
1
2
3
4
5
6
7
Expand triggered
2^2 => 2^3으로 늘어남



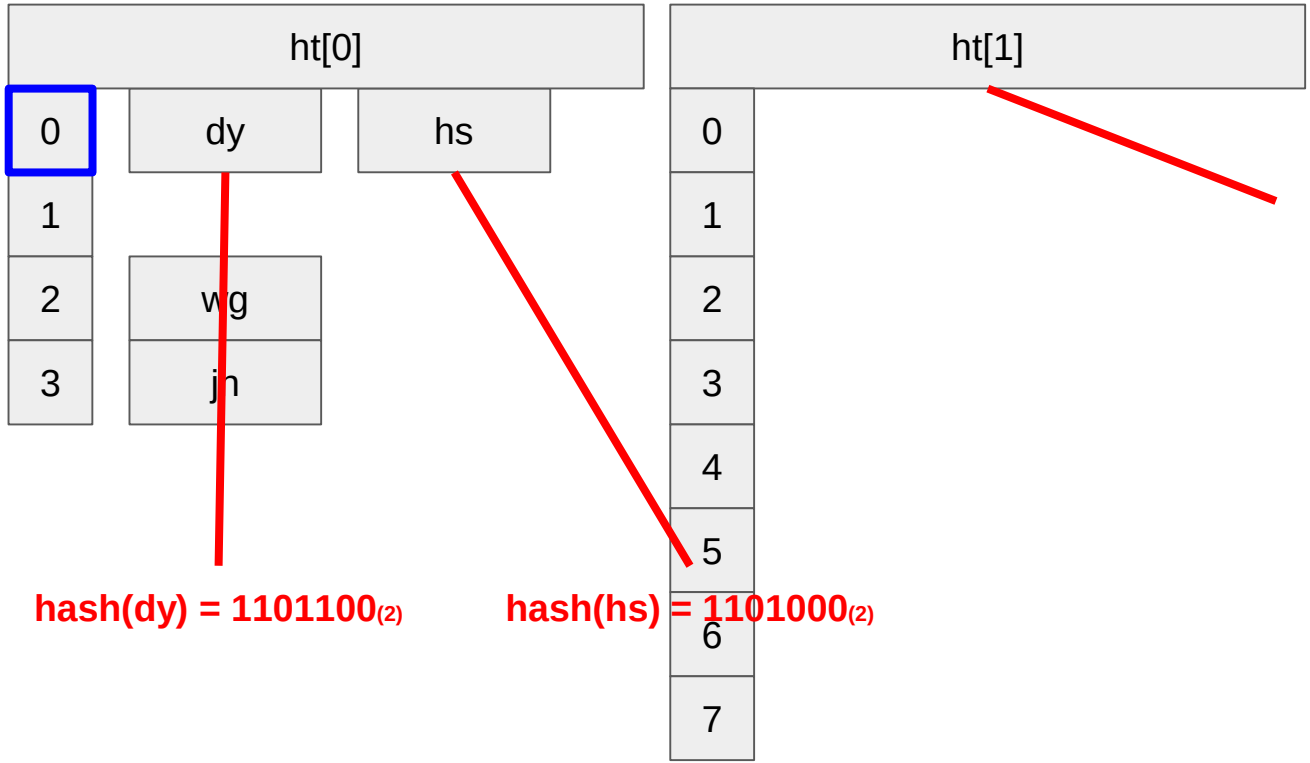
Incremental Rehashing…



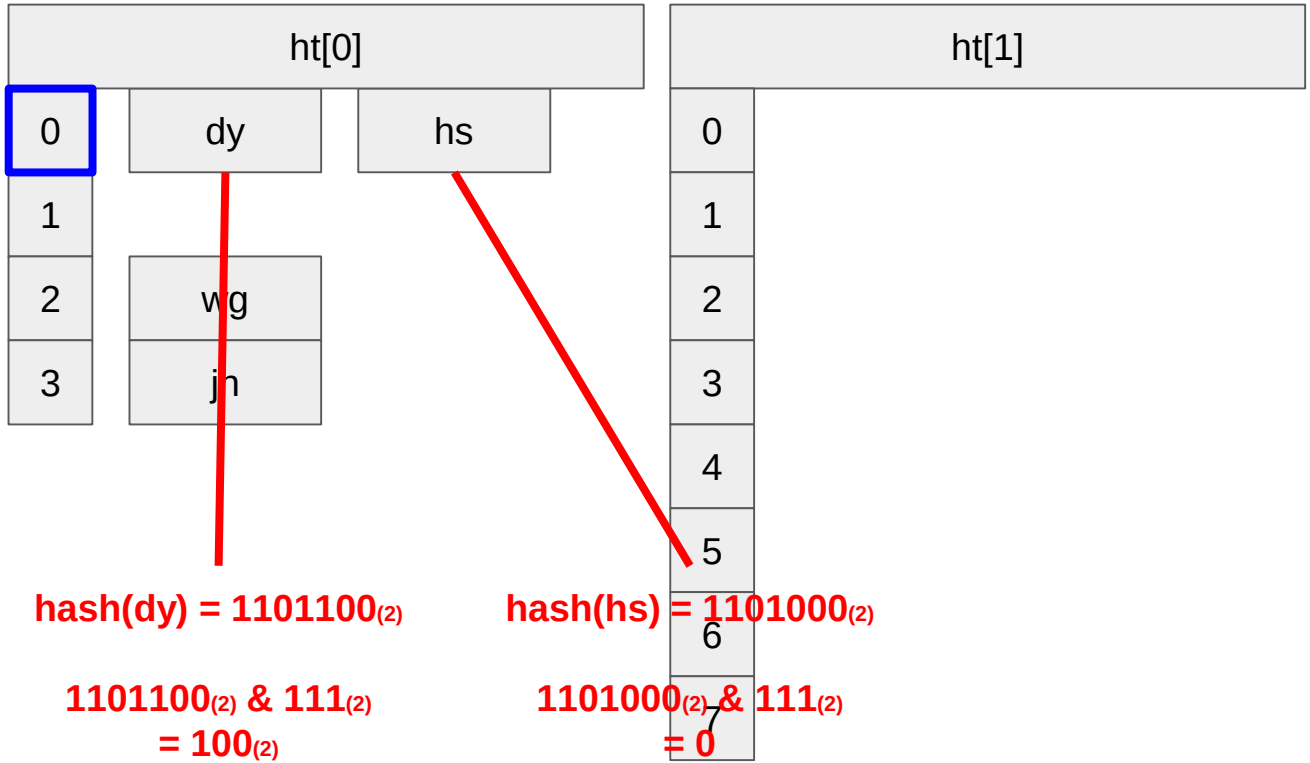
Incremental Rehashing…



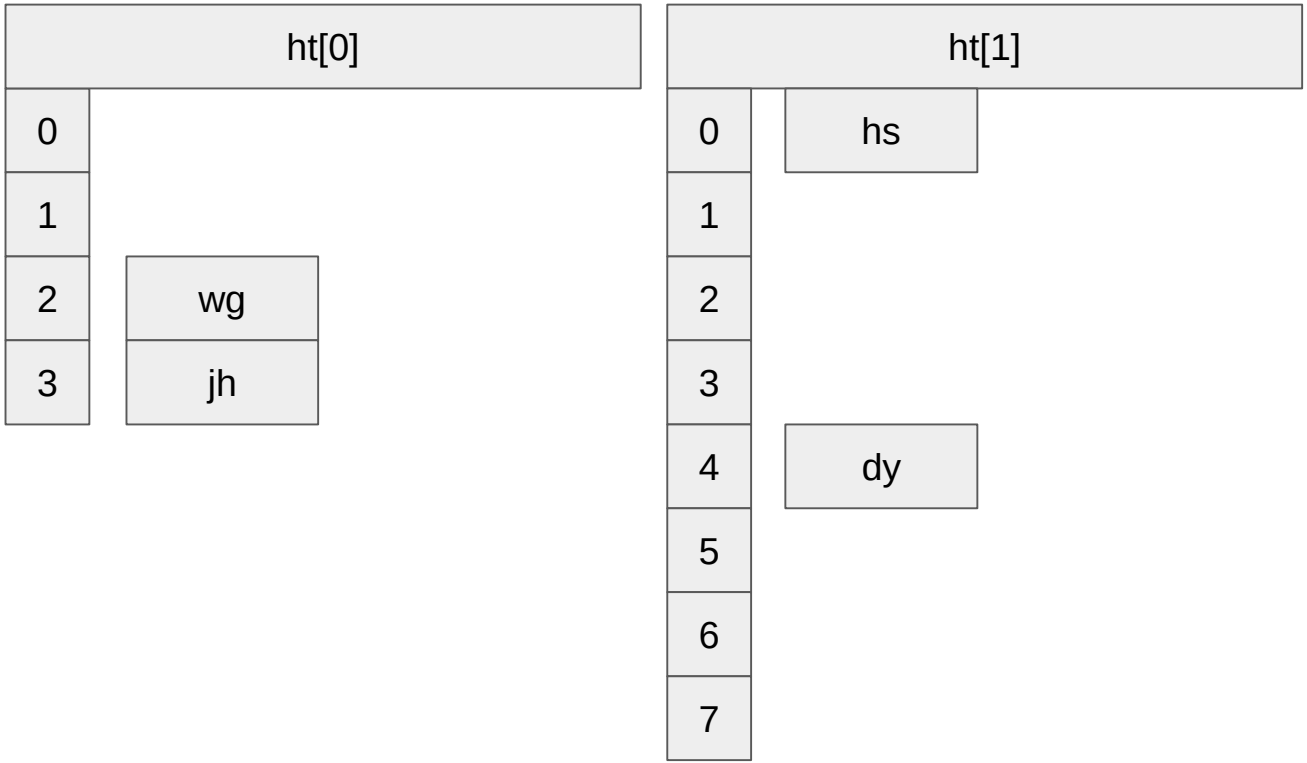
Incremental Rehashing…



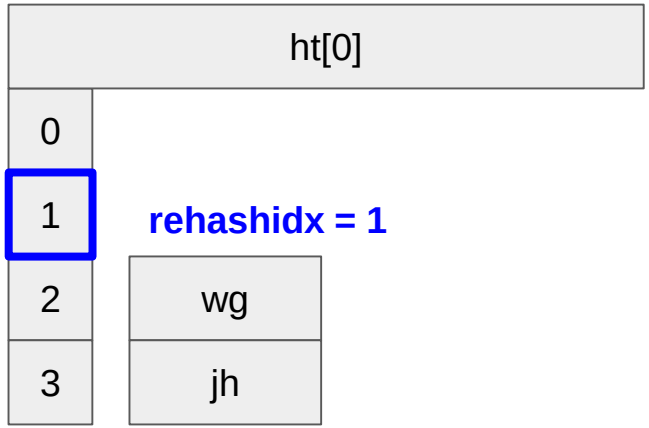
Incremental Rehashing…

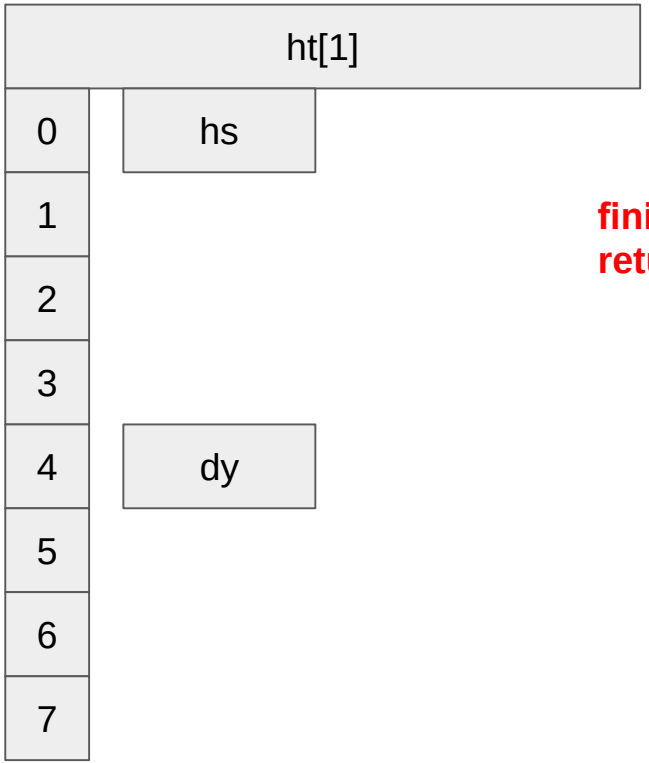


Incremental Rehashing…



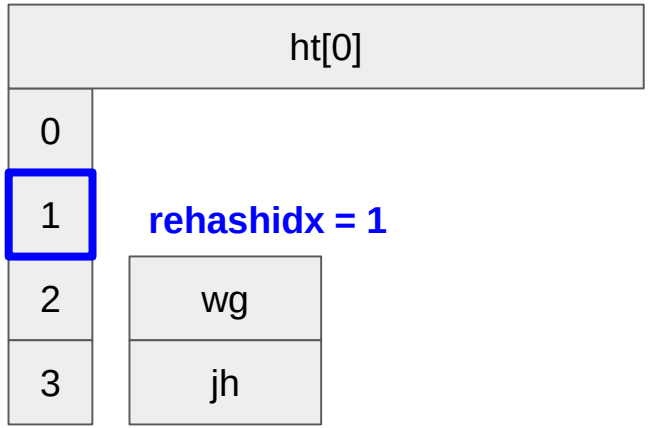
Incremental Rehashing…

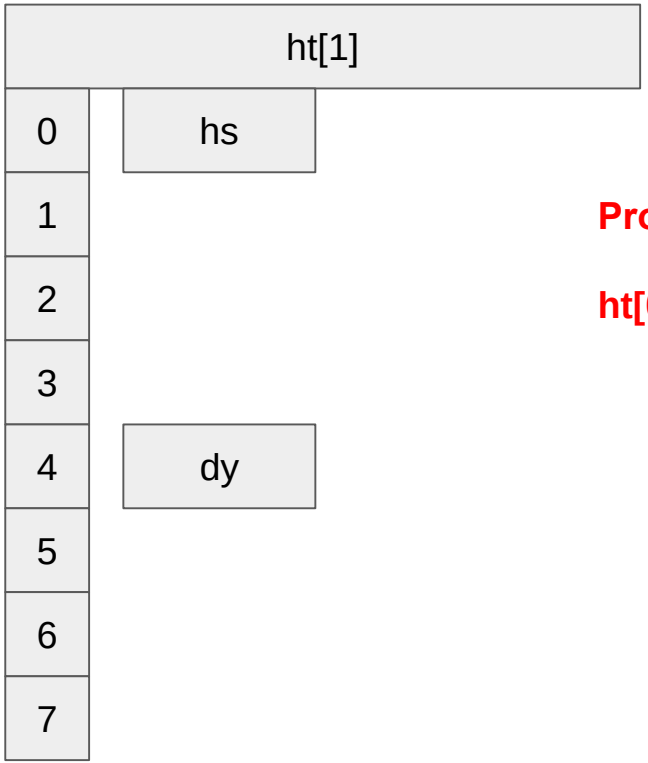




Incremental Rehashing…

finish rehash temporary…
return 1...

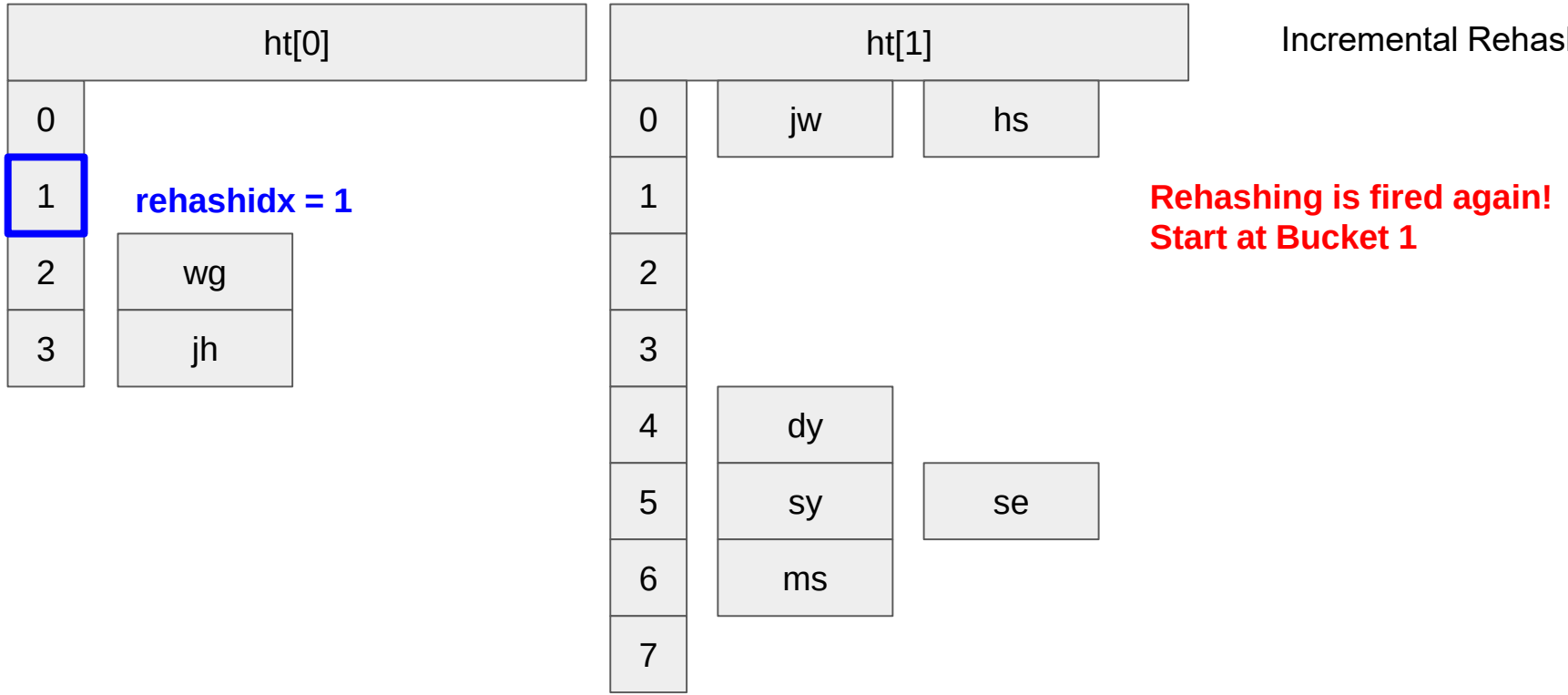


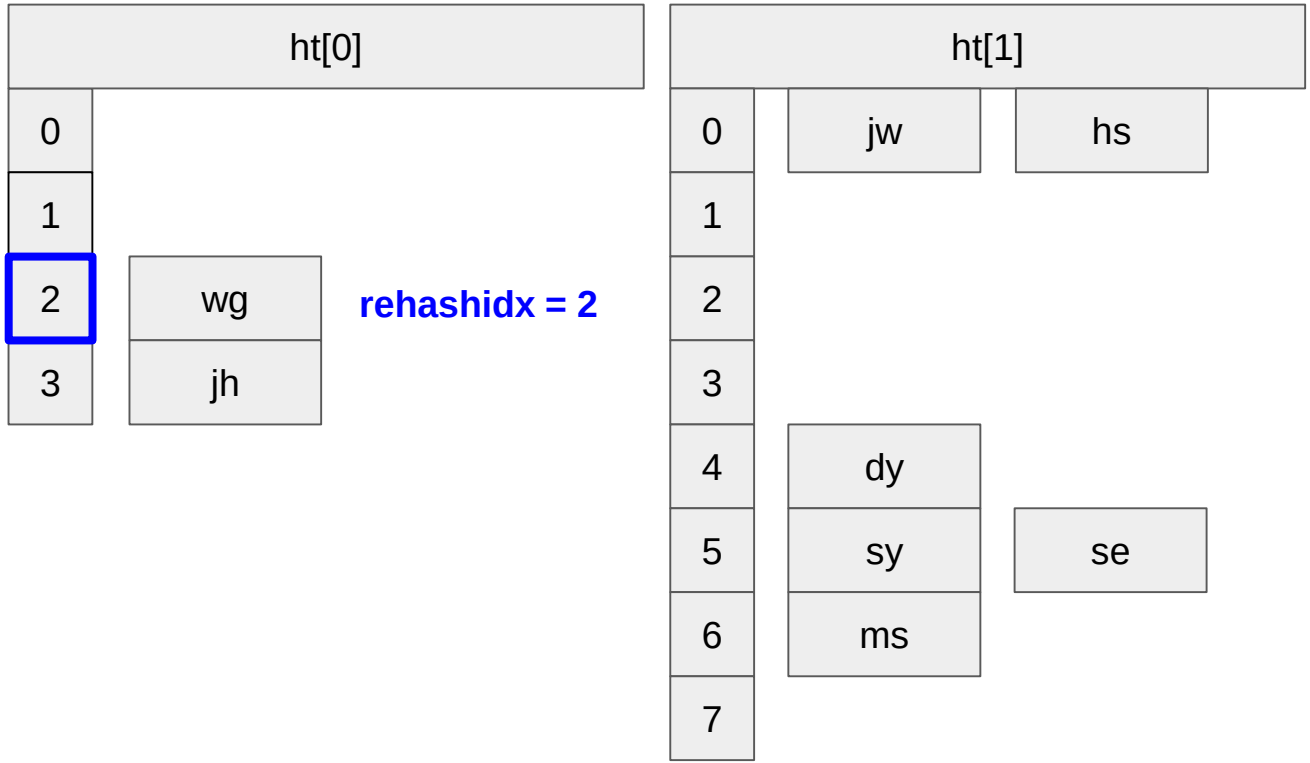


Incremental Rehashing…

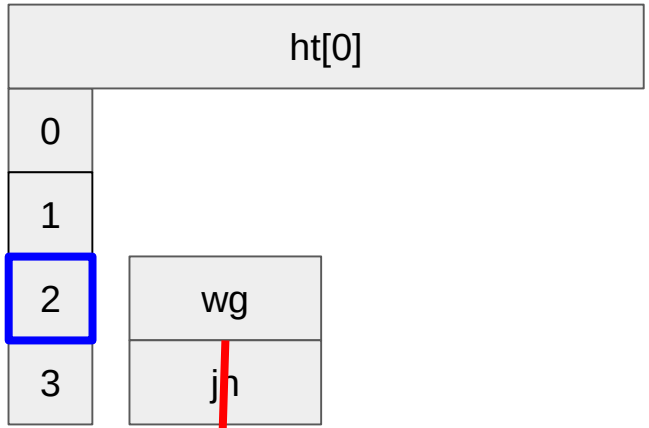
Process other redis operations…

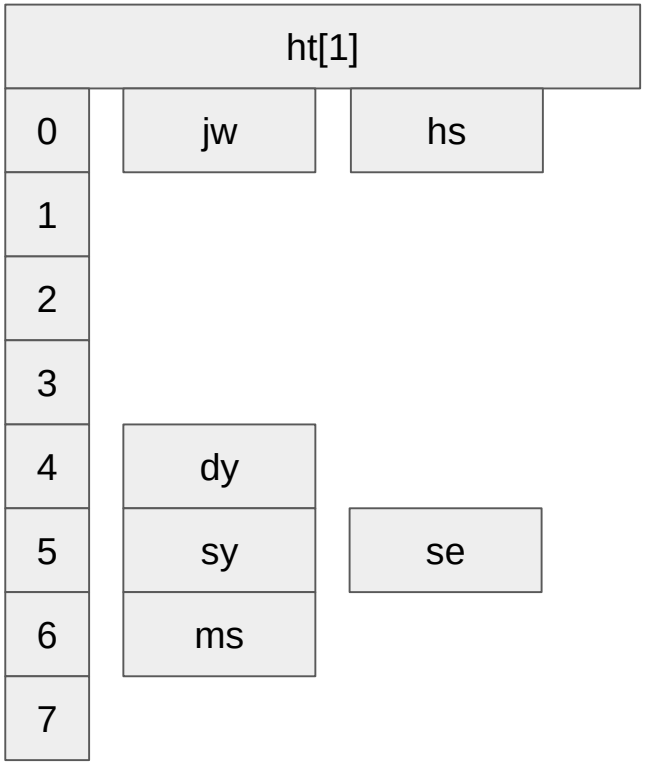
ht[0], ht[1]이 모두 사용됨.


ht[0]
0
1
2
3
wg
jh
rehashidx = 1
ht[1]
0
1
2
3
4
5
6
7
jw
hs
dy
sy
ms
se
Incremental Rehashing…
Rehashing is fired again!
Start at Bucket 1



Incremental Rehashing…



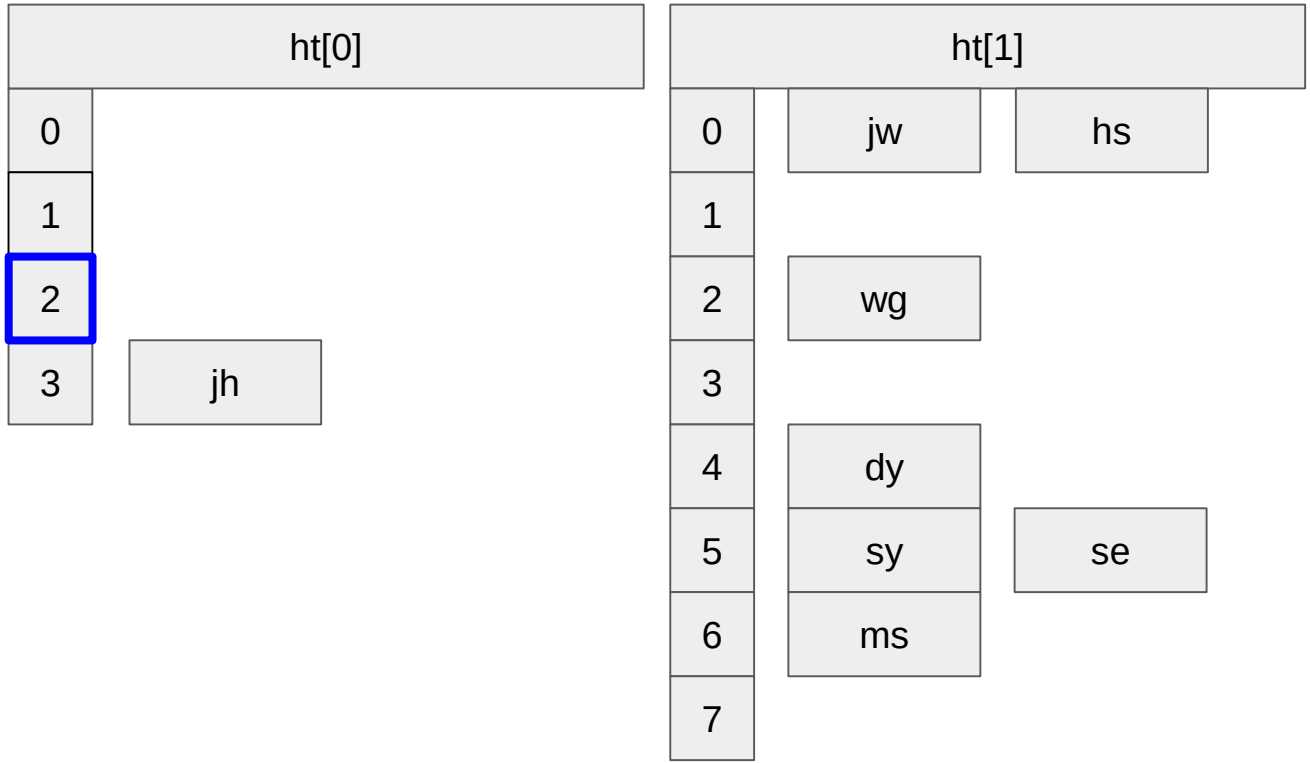


Incremental Rehashing…

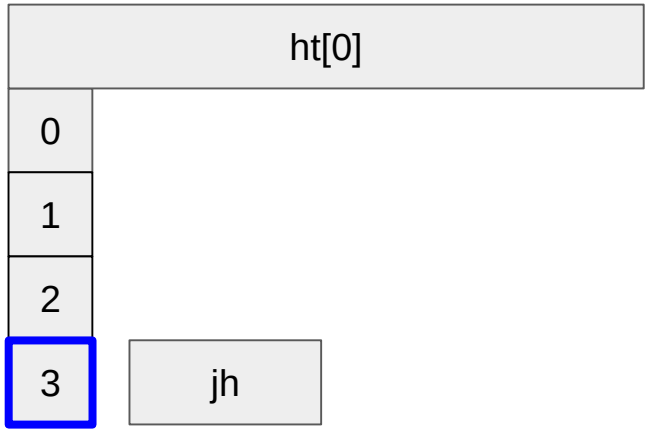
hash(wg) = $1101010_{(2)}$

$1101010_{(2)}$ & $11_{(2)}$ = 2

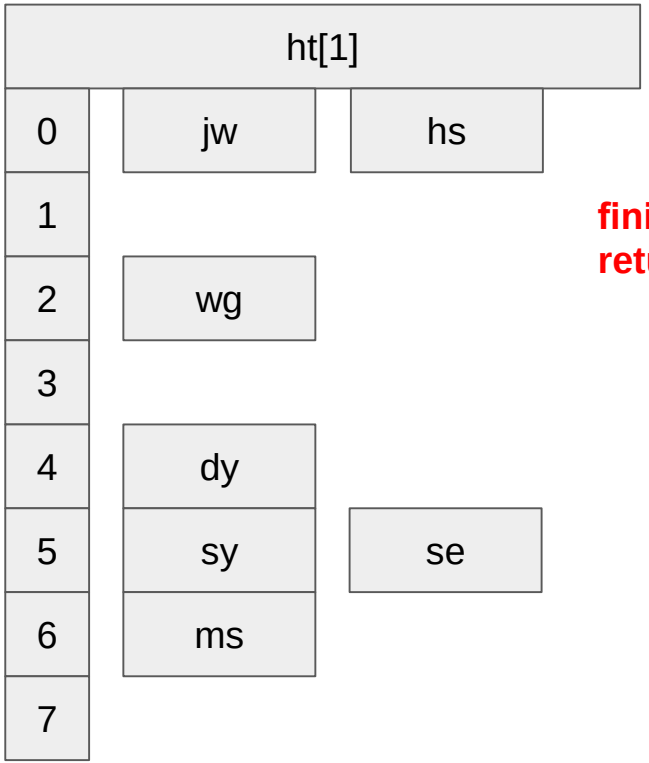
$1101010_{(2)}$ & $111_{(2)}$ = 2



Incremental Rehashing…

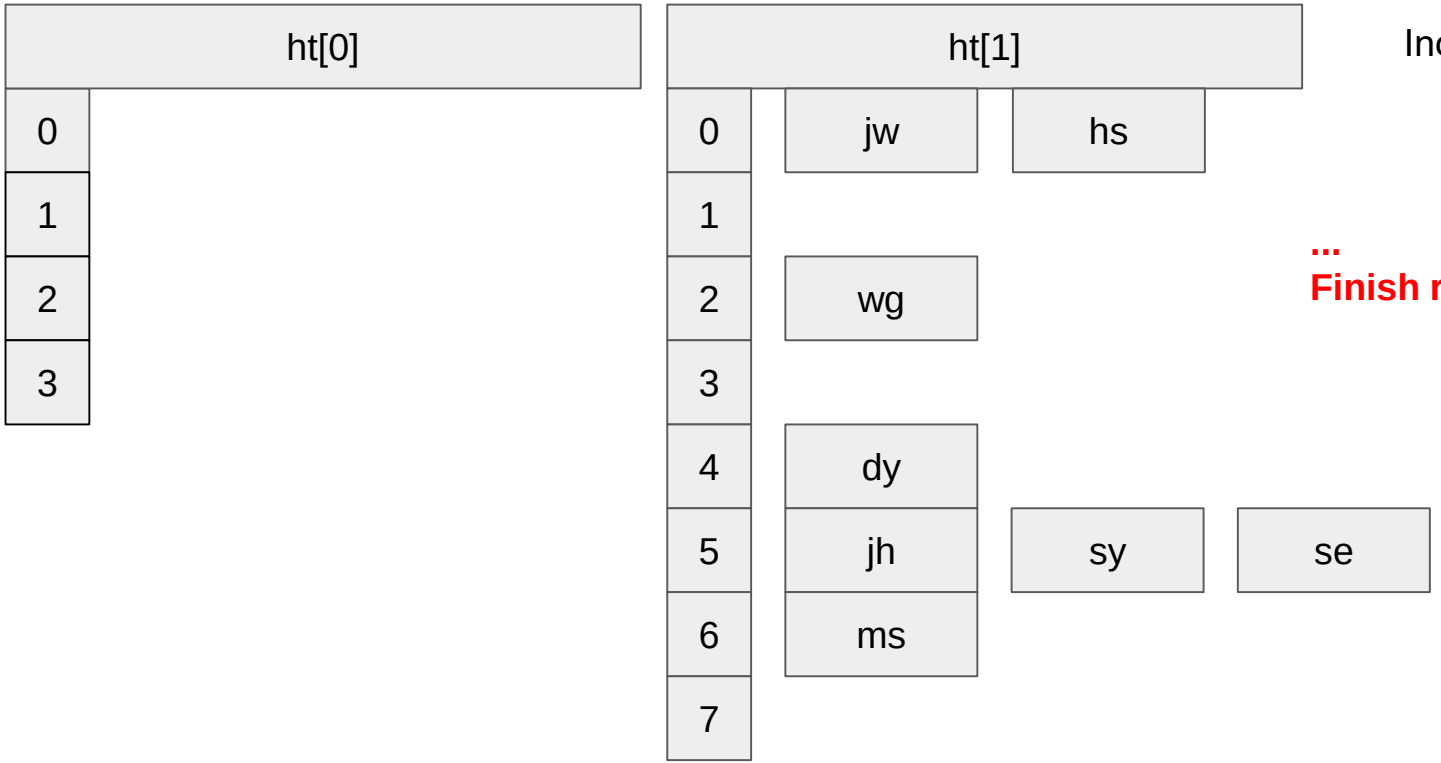


rehashidx = 3



Incremental Rehashing…

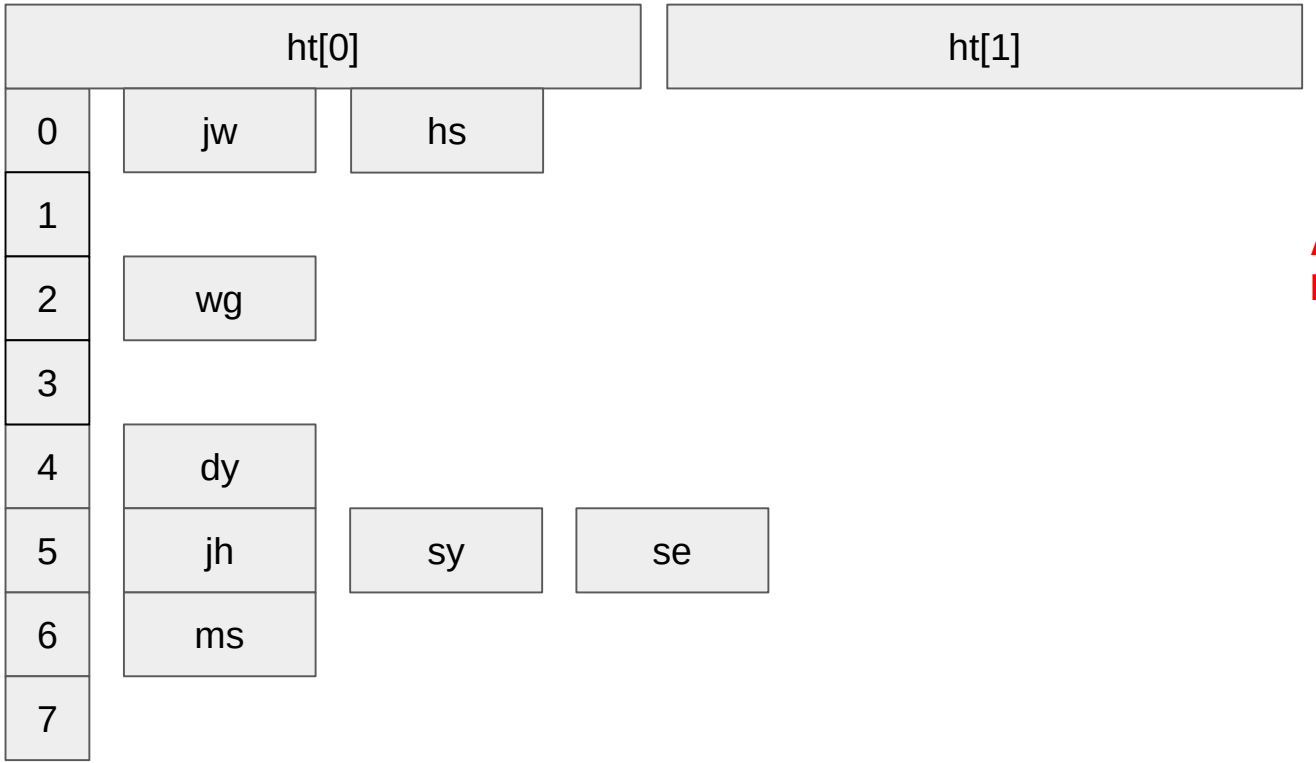
finish rehash temporary…
return 1...



Incremental Rehashing…

...

Finish rehash last one



Incremental Rehashing…

rehashidx = -1

At last, return 0
Rehashing is finished

SDS

abcdefghijklmnopqrstu	\0

Representable with C string

abcdef\0sdfl\0asdfasc\0sdf	\0

Not representable with C string

abcdefghijklmnopqrstu	\0

abcdef\0sdfl\0asdfasc\0sdf	\0

Calculate length of string
⇒ strlen()
⇒ O(n)

SDS?

- **S**imple **D**ynamic **S**trings
- 특징
  - Simpler to use (just like C string)
  - Binary safe
    - string 공간 전에 string 정보를 나타내는 header를 저장해서 사용
    - 모든 binary를 표현할 수 있음 (중간에 “\0”이 있어도 출력 가능)
  - Computationally more efficient
  - Compatible with normal C string functions (not perfectly yet)
  - Heap allocation

SDS?

- Why use this?
  - Natural C string library만으로는 high level 단계의 string operation을 적용하기가 까다로움 (귀찮)
  - high level string operation
  - high performance with no penalty
  - compatibility

SDS?

- 장점
  - 다른 custom string library과는 다르게, sds 타입이 C string type과 compatible함.
    - 그래서 C string library에다가 sds를 그대로 사용할 수 있음.
    - `printf("%s\n", sds_string);` `printf("%s\n", string->buf);`
  - String buffer에 접근하여 사용하는것이 아니기 때문에, string에 접근할 때 별도의 reference variable이 필요가 없어짐. (additional memory address buffer가 필요 없음)
    - *string->buf* 로 사용하는 경우는 string에 접근할 때마다 implicit하게 address buffer를 생성한다고 함.
  - 사용하는 Address가 시스템적으로 항상 동일하기에 Cache에도 용이함.

SDS?

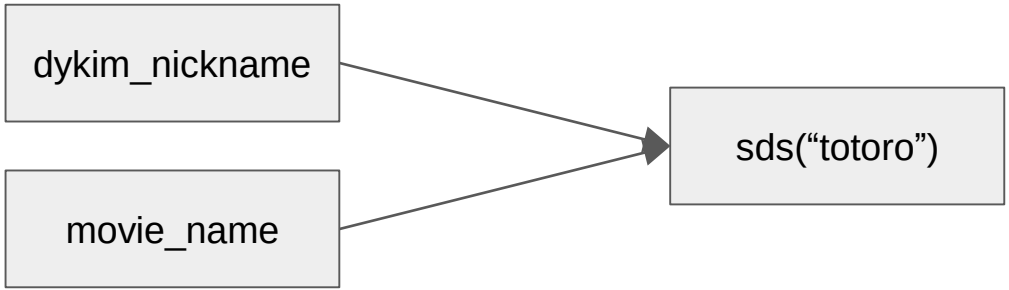
- 단점
  - String의 spec(concat, size, …)을 변경하려면 항상 함수를 통하여 변경해야함.

    ```
    s = sdscat(s,"Some more data");
    ```

    function 내부에 side effect가 있는지 판단하기 어려움으로 항상 유의해야함.
    버그가 생길 수 있음.

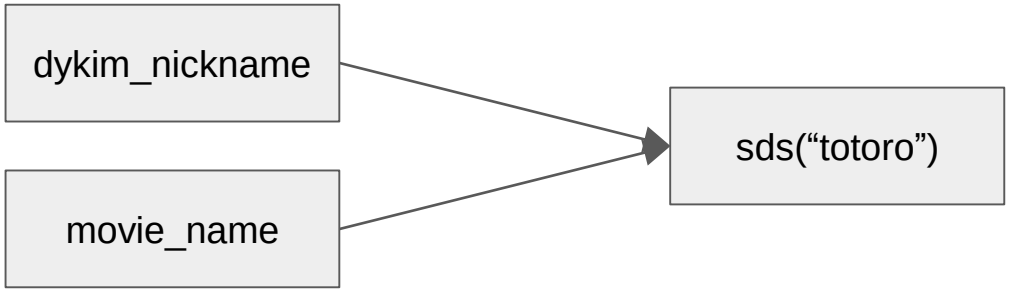
  - SDS string을 여러 place에서 share하여 사용하면, 해당 string을 변경할 때마다 여러 place에서 사용하고 있는 reference들을 모두 변경해야하는 경우가 생길 수 있음.
    → 사용할 때 reference count를 관리하는 structure로 sds sharing을 관리해주어야 함

단점 1 – side effect



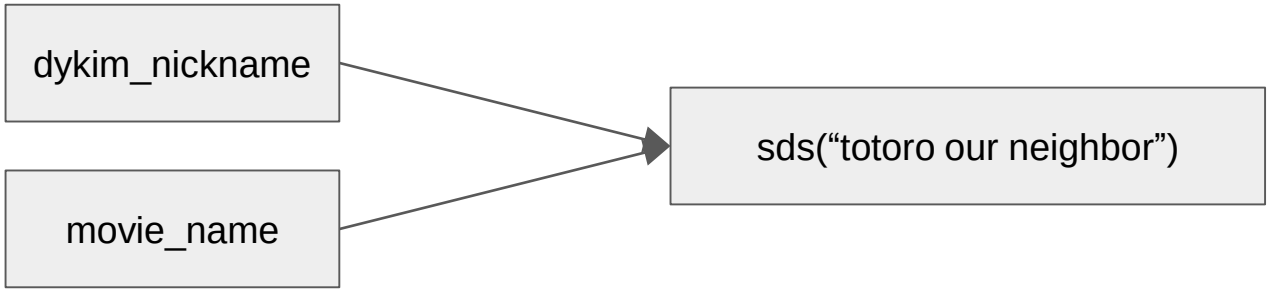
단점 1 – side effect

movie_name = sdscat(totoro, “ our neighbor”)



단점 1 – side effect

movie_name = sdscat(totoro, “ our neighbor”)

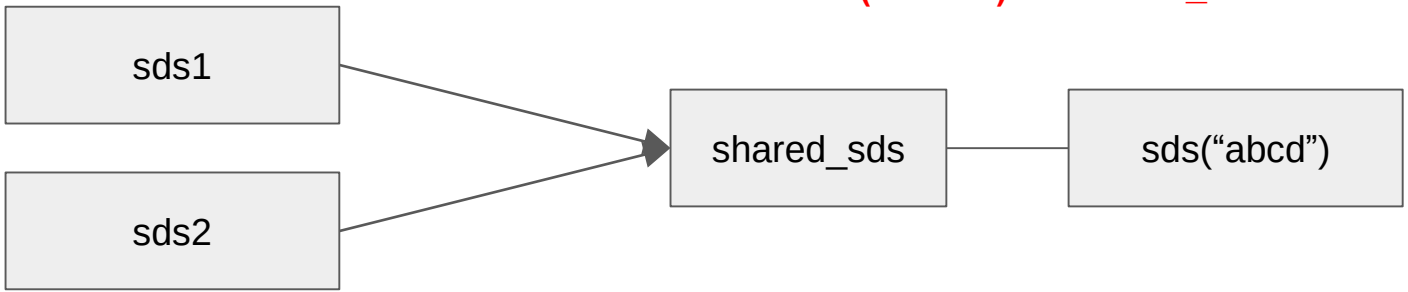


Side effect!

단점 2 – reference problem

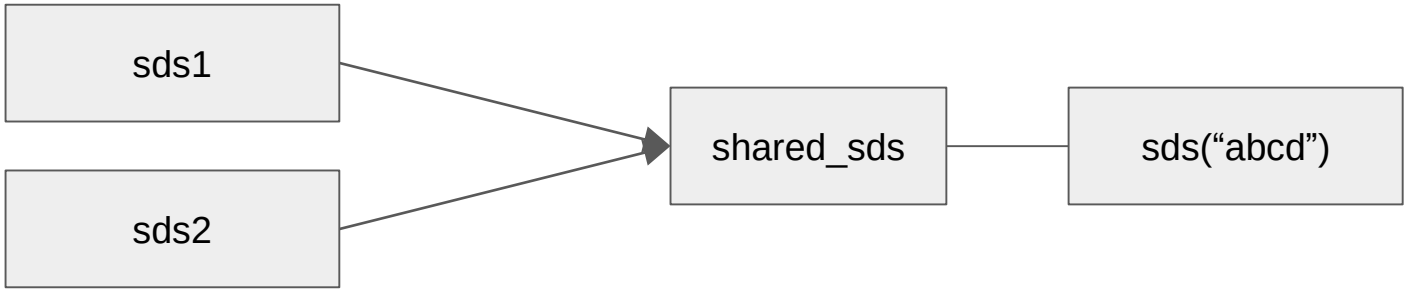
sds1 = shared_sds
sds2 = shared_sds

sds(“abcd”).reference_count = 3



단점 2 – reference problem

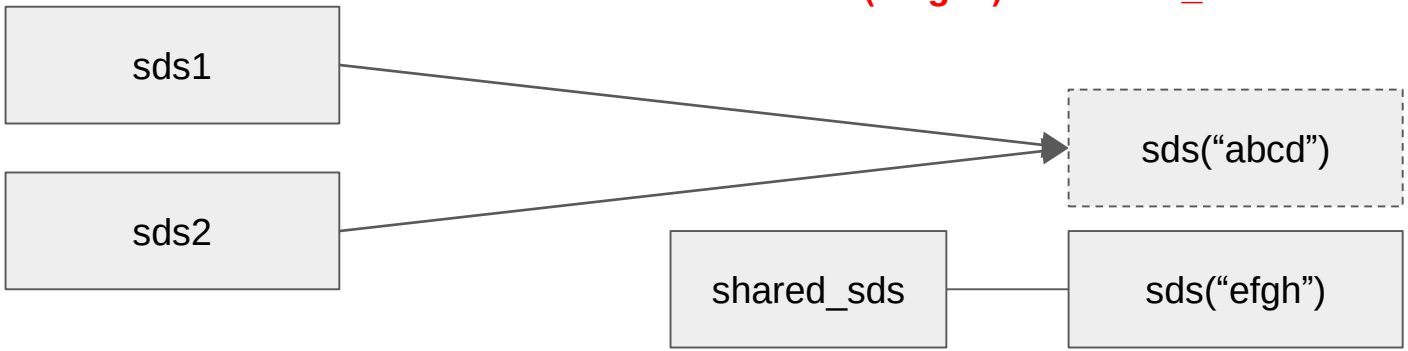
shared_sds = sdsnew(“efgh”)



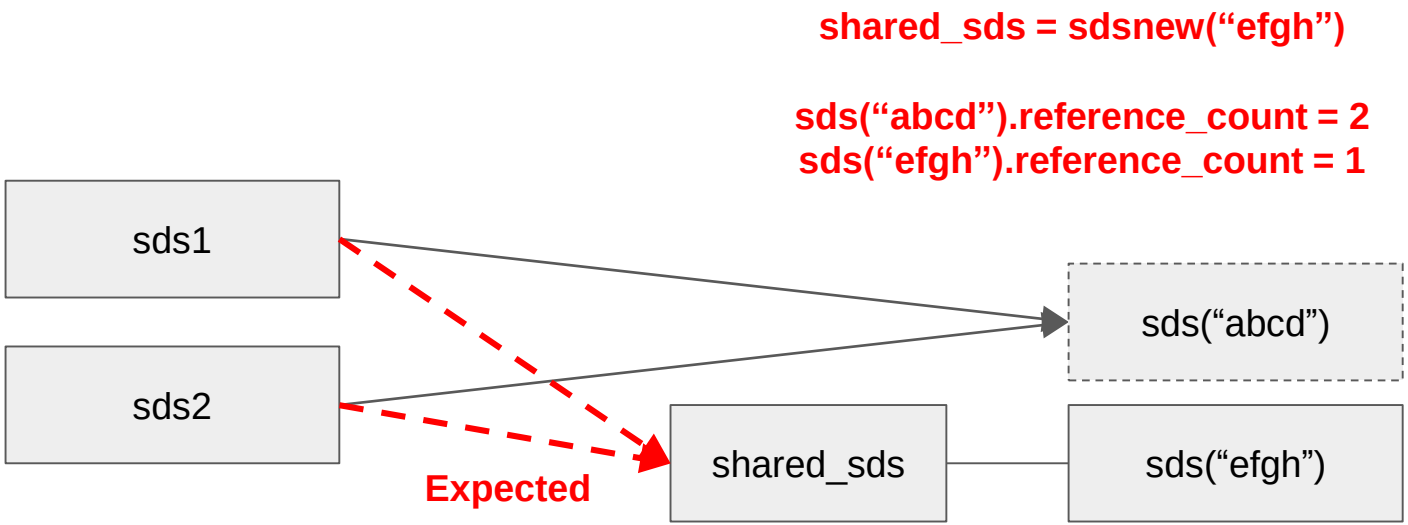
단점 2 – reference problem

shared_sds = sdsnew(“efgh”)

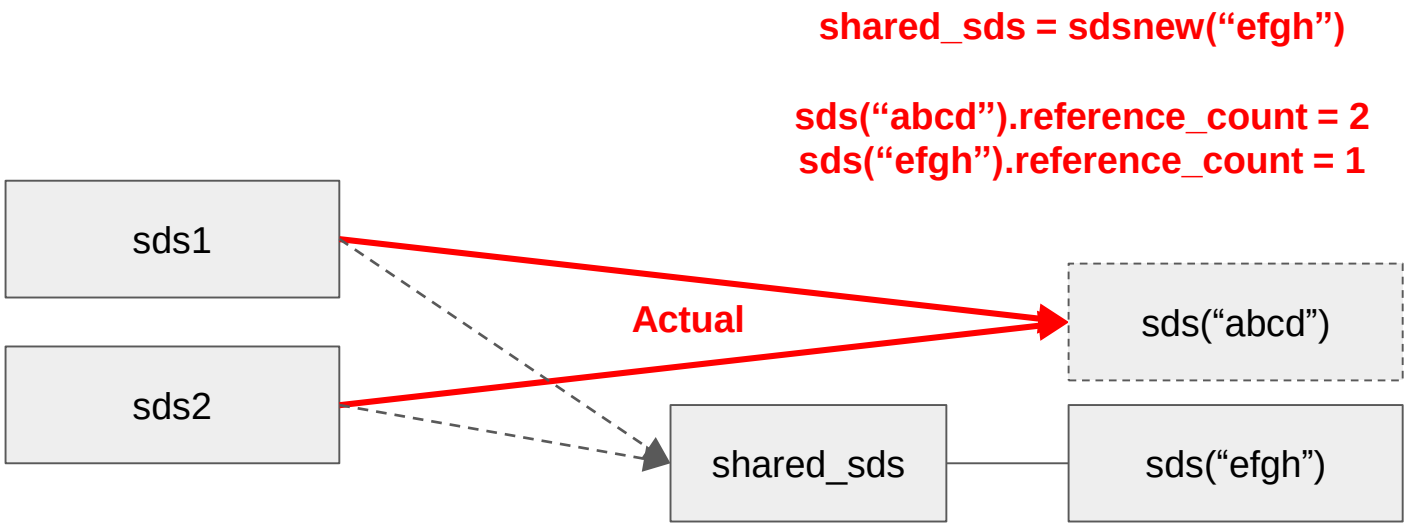
sds(“abcd”).reference_count = 2
sds(“efgh”).reference_count = 1



단점 2 – reference problem

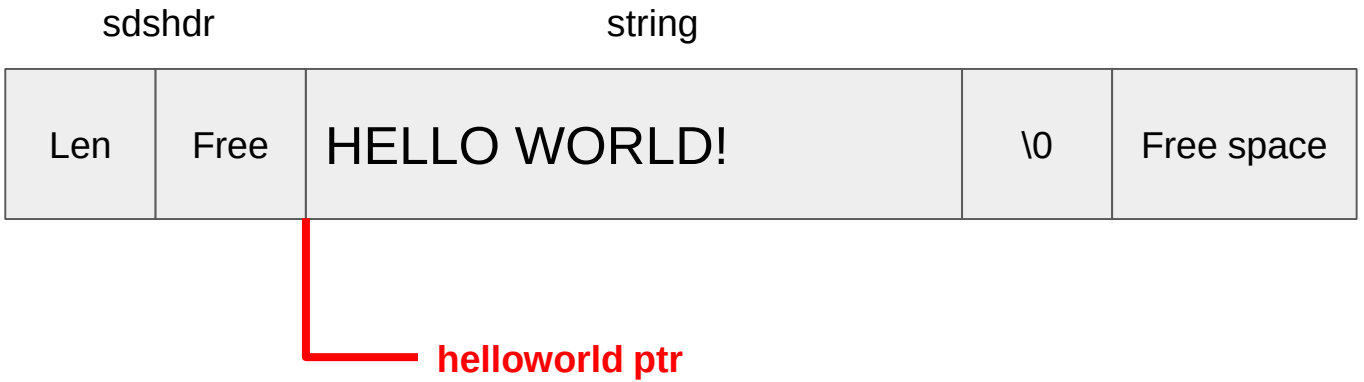


단점 2 – reference problem



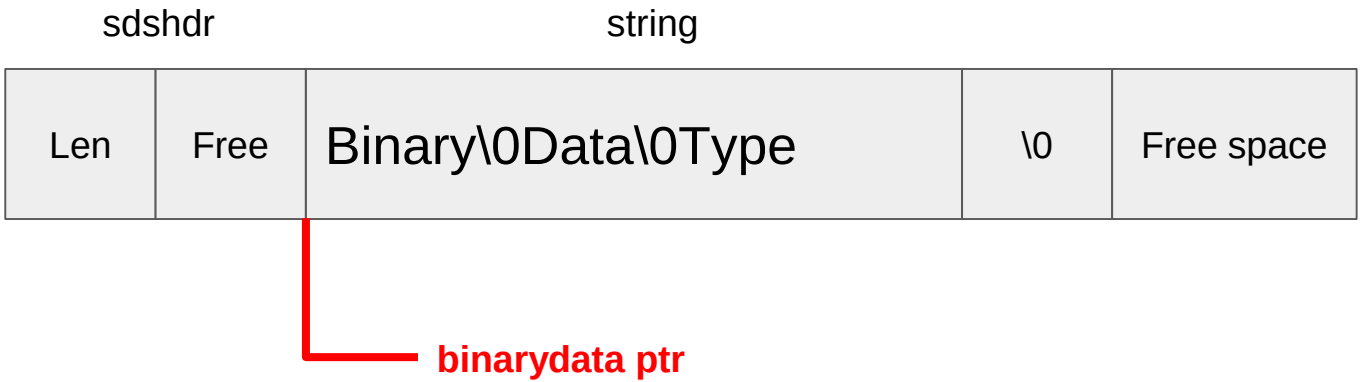
Structure of SDS

sds helloworld = sdsnew(“HELLO WORLD!”)
(char *)



Structure of SDS

sds binarydata = sdsnew(binaryData)
(char *)



Structure of SDS

Name	Description
sdshdr	string의 metadata를 저장함. string 저장공간 이전에 저장
len	string length
free (alloc)	sds에 남은 free space size
string	string
free space	string mutation에 대비한 free space (size up에 대비)

Power of SDS structure

- String length를 구하는데 O(1)
  - metadata가 무조건 string 앞에 위치하기 때문에 address 계산 한번으로 바로 조회할 수 있음.
- C string compatible
  - 표면적으로는 string 부분의 pointer를 바라보고 있기 때문에, C string library를 자유로이 사용할 수 있음.
  - 다만 내부에 “\0”이 있으면 거기까지만 data로 인식해버리는 단점이 있음.
- Pre-allocated space (free space)
  - sdscat이 많이 발생하는 경우를 대비
  - 이게 과하면 오히려 loss이므로 redis는 1MB 이하로 제한을 두고 있음.
- 최적화 되어있는 sds functions
  - 필요할때만 sds reallocate. 최대한 남은 sds space를 활용하도록 구현되어 있음

Useful sds functions

- sds를 create / mutate / free를 할때는 sds function들을 사용해야함
- Functions

```
sds sdsnewlen(const void *init, size_t initlen);
sds sdsnew(const char *init);
sds sdsempty(void);
sds sdsdup(const sds s);
void sdsfree(sds s);
sds sdsgrowzero(sds s, size_t len);
sds sdscatlen(sds s, const void *t, size_t len);
sds sdscat(sds s, const char *t);
sds sdscatsds(sds s, const sds t);
sds sdscpylen(sds s, const char *t, size_t len);
sds sdscpy(sds s, const char *t);
```

Useful sds functions

- Examples

```
sds favorite_movie = sdsnew("totoro");
sds dykim_nickname = sdsdup(favorite_movie);
sds semititle = sdsnew("Our neighborhood ");
// Beware of side effects!!
// `sds fulltitle = sdscatsds(semititle, favorite_movie);`
// instead of this...
sds fulltitle = sdsempty();
fulltitle = sdscatsds(fulltitle, semititle);
fulltitle = sdscatsds(fulltitle, favorite_movie);
```

GET / SET Command Flow

Command

- Redis를 수정할 수 있는 command
- redisServer에서 dict table로 관리되고 있음

```
struct redisServer {
    /* General */
    pid_t pid;                  /* Main process pid. */
    char *configfile;           /* Absolute config file path, or NULL */
    char *executable;           /* Absolute executable file path. */
    char **exec_argv;           /* Executable argv vector (copy). */
    int hz;                     /* serverCron() calls frequency in hertz */
    redisDb *db;
    dict *commands;             /* Command table */
    dict *orig_commands;        /* Command table before command renaming. */
```

Command

- server.c에 정의해놓은 redisCommandTable array를 redisServer에 populate

```
struct redisCommand redisCommandTable[] = {
    {"module",moduleCommand,-2,"as",0,NULL,0,0,0,0,0},
    {"get",getCommand,2,"rF",0,NULL,1,1,1,0,0},
    {"set",setCommand,-3,"wm",0,NULL,1,1,1,0,0},
    {"setnx",setnxCommand,3,"wmF",0,NULL,1,1,1,0,0},
    {"setex",setexCommand,4,"wm",0,NULL,1,1,1,0,0},
    {"psetex",psetexCommand,4,"wm",0,NULL,1,1,1,0,0},
    {"append",appendCommand,3,"wm",0,NULL,1,1,1,0,0},
    {"strlen",strlenCommand,2,"rF",0,NULL,1,1,1,0,0},
    {"del",delCommand,-2,"w",0,NULL,1,-1,1,0,0},
    {"unlink",unlinkCommand,-2,"wF",0,NULL,1,-1,1,0,0},
    {"exists",existsCommand,-2,"rF",0,NULL,1,-1,1,0,0},
    {"setbit",setbitCommand,4,"wm",0,NULL,1,1,1,0,0},
    {"getbit",getbitCommand,3,"rF",0,NULL,1,1,1,0,0},
    {"bitfield",bitfieldCommand,-2,"wm",0,NULL,1,1,1,0,0},
    {"setrange",setrangeCommand,4,"wm",0,NULL,1,1,1,0,0},
    {"getrange",getrangeCommand,4,"r",0,NULL,1,1,1,0,0},
    {"substr",getrangeCommand,4,"r",0,NULL,1,1,1,0,0},
    {"incr",incrCommand,2,"wmF",0,NULL,1,1,1,0,0},
    {"decr",decrCommand,2,"wmF",0,NULL,1,1,1,0,0},
    {"mget",mgetCommand,-2,"rF",0,NULL,1,-1,1,0,0},
    {"rpush",rpushCommand,-3,"wmF",0,NULL,1,1,1,0,0},
    {"lpush",lpushCommand,-3,"wmF",0,NULL,1,1,1,0,0},
    {"rpushx",rpushxCommand,-3,"wmF",0,NULL,1,1,1,0,0},
    {"lpushx",lpushxCommand,-3,"wmF",0,NULL,1,1,1,0,0},
    {"linsert",linsertCommand,5,"wm",0,NULL,1,1,1,0,0},
```

```
/* Populates the Redis Command Table starting from the hard coded list
 * we have on top of redis.c file. */
void populateCommandTable(void) {
    int j;
    int numcommands = sizeof(redisCommandTable)/sizeof(struct redisCommand);

    for (j = 0; j < numcommands; j++) {
        struct redisCommand *c = redisCommandTable+j;
        char *f = c->sflags;
        int retval1, retval2;
```

Command

- networking.c에서 client가 server의 command를 실행시킬 수 있도록 함

```
void processInputBuffer(client *c) {
    server.current_client = c;
    /* Keep processing while there is something in the input buffer */
    while(sdslen(c->querybuf)) {
```

```
/* Only reset the client when the command was executed. */
if (processCommand(c) == C_OK) {
```

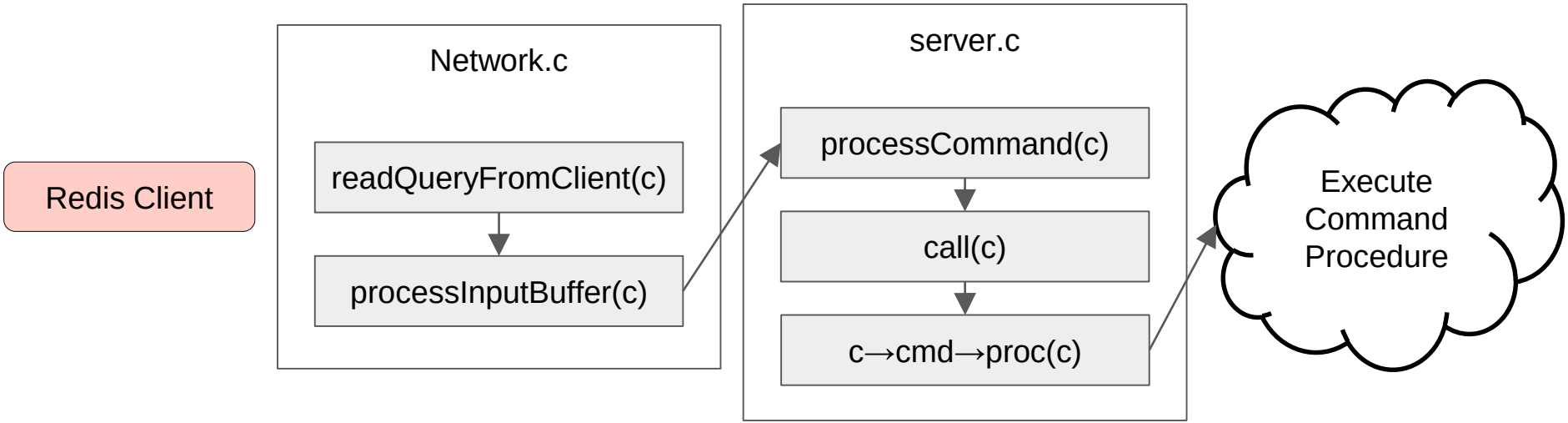
Command

- server.c의 processCommand에서는 client가 요청한 command를 실행한다.

```
int processCommand(client *c) {
    /* The QUIT command is handled separately. Normal command procs will
     * go through checking for replication and QUIT will cause trouble
     * when FORCE_REPLICATION is enabled and would be implemented in
     * a regular command proc. */
    if (!strcasecmp(c->argv[0]->ptr,"quit")) {
        addReply(c,shared.ok);
        c->flags |= CLIENT_CLOSE_AFTER_REPLY;
        return C_ERR;
    }

    /* Now lookup the command and check ASAP about trivial error conditions
     * such as wrong arity, bad command name and so forth. */
    c->cmd = c->lastcmd = lookupCommand(c->argv[0]->ptr);
```

```
    /* Exec the command */
    if (c->flags & CLIENT_MULTI &&
        c->cmd->proc != execCommand && c->cmd->proc != discardCommand &&
        c->cmd->proc != multiCommand && c->cmd->proc != watchCommand)
    {
        queueMultiCommand(c);
        addReply(c,shared.queued);
    } else {
        call(c,CMD_CALL_FULL);
        c->woff = server.master_repl_offset;
        if (listLength(server.ready_keys))
            handleClientsBlockedOnKeys();
    }
    return C_OK;
```


Redis Client
Network.c
readQueryFromClient(c)
processInputBuffer(c)
server.c
processCommand(c)
call(c)
c→cmd→proc(c)
Execute
Command
Procedure

GET / SET Command Procedure

- *t_string.c*에 선언되어있음.

```c
void getCommand(client *c) {
    getGenericCommand(c);
}
```

```c
/* SET key value [NX] [XX] [EX <seconds>] [PX <milliseconds>] */
void setCommand(client *c) {
    // Some Encoding, flag logics...
    int j;
    robj *expire = NULL;
    ...
    setGenericCommand(c,flags,c->argv[1],c->argv[2],expire,unit,NULL,NULL);
}
```

- get/setGenericCommand(c)에서 command 작업이 수행됨.

getGenericCommand()

- *db.c*의 lookupKeyReadOrReply()을 이용하여 client가 사용하고 있는 db를 조회함.

```
int getGenericCommand(client *c) {
    robj *o;

    if ((o = lookupKeyReadOrReply(c,c->argv[1],shared.nullbulk)) == NULL)
        return C_OK;

    if (o->type != OBJ_STRING) {
        addReply(c,shared.wrongtypeerr);
        return C_ERR;
    } else {
        addReplyBulk(c,o);
        return C_OK;
    }
}
```

setGenericCommand()

- *db.c*의 lookupKeyWrite()을 이용하여 client가 사용하고 있는 db에 key를 조회하고, setKey()를 이용하여 key를 write함

```
void setGenericCommand(client *c, int flags, robj *key, robj *val, robj *expire, 
    long long milliseconds = 0; /* initialized to avoid any harmness warning */

    if (expire) {
        if (getLongLongFromObjectOrReply(c, expire, &milliseconds, NULL) != C_OK)
            return;
        if (milliseconds <= 0) {
            addReplyErrorFormat(c,"invalid expire time in %s",c->cmd->name);
            return;
        }
        if (unit == UNIT_SECONDS) milliseconds *= 1000;
    }

    if ((flags & OBJ_SET_NX && lookupKeyWrite(c->db,key) != NULL) ||
        (flags & OBJ_SET_XX && lookupKeyWrite(c->db,key) == NULL))
    {
        addReply(c, abort_reply ? abort_reply : shared.nullbulk);
        return;
    }
    setKey(c->db,key,val);
    server.dirty++;
    if (expire) setExpire(c,c->db,key,mstime()+milliseconds);
    notifyKeyspaceEvent(NOTIFY_STRING,"set",key,c->db->id);
    if (expire) notifyKeyspaceEvent(NOTIFY_GENERIC,
        "expire",key,c->db->id);
    addReply(c, ok_reply ? ok_reply : shared.ok);
}
```

lookupKey()

- db의 dict structure를 dictFind를 이용하여 찾아냄

```
robj *lookupKey(redisDb *db, robj *key, int flags) {
    dictEntry *de = dictFind(db->dict,key->ptr);
    if (de) {
        robj *val = dictGetVal(de);
```

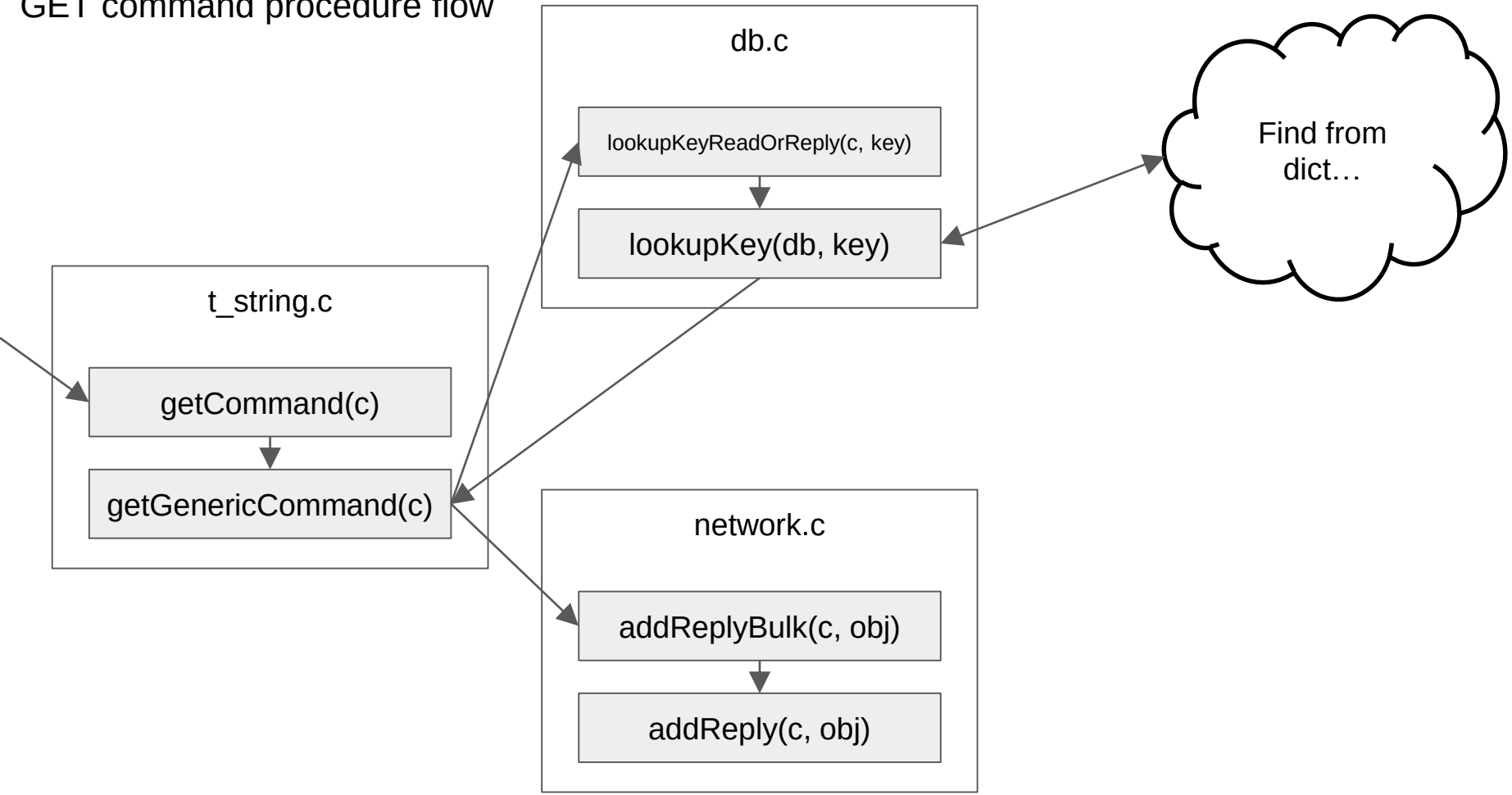
setKey()

- lookupKey()로 key가 있는지 조회한 뒤, add를 할지 overwrite를 할지 정함

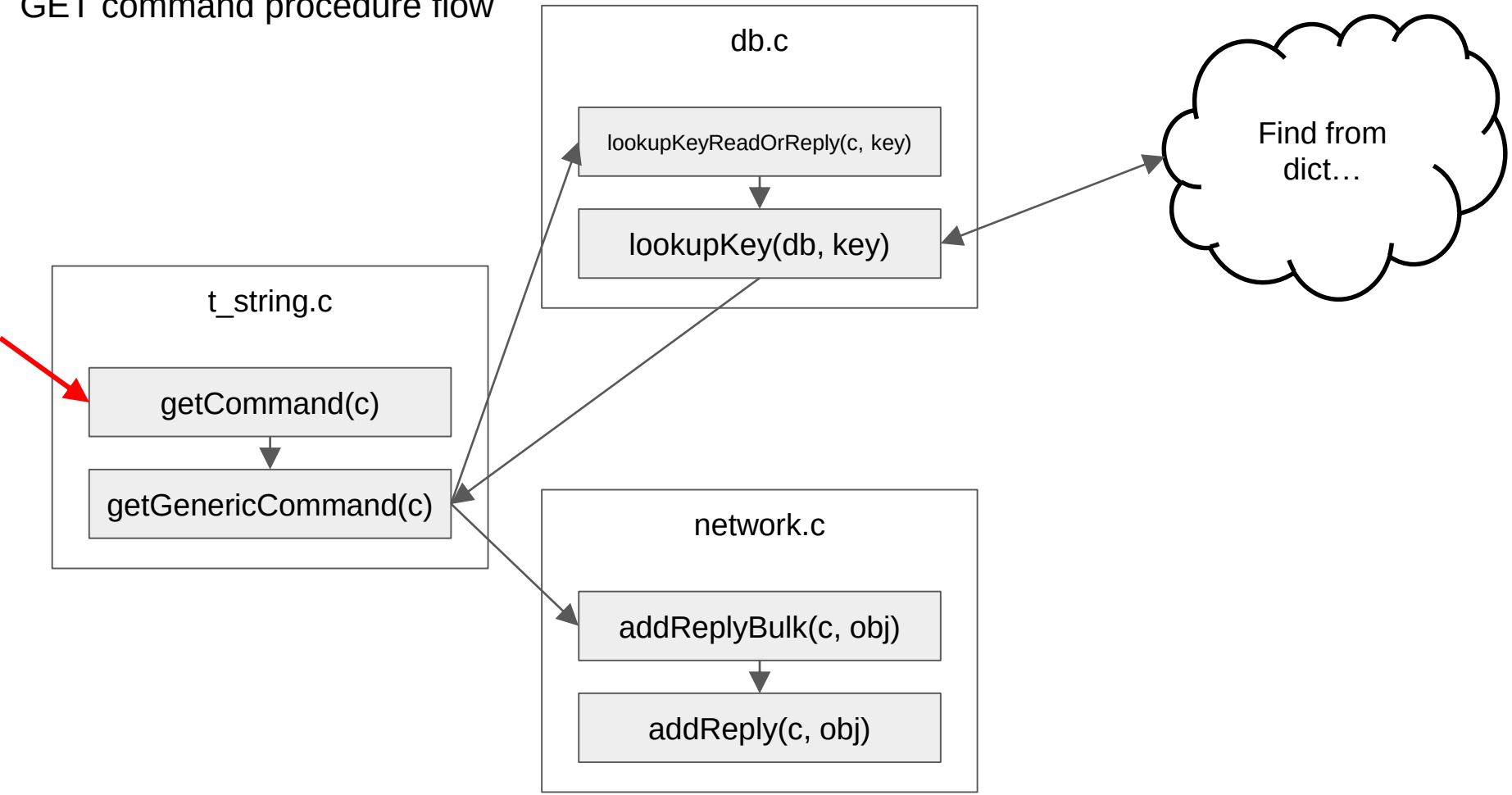
```
void setKey(redisDb *db, robj *key, robj *val) {
    if (lookupKeyWrite(db,key) == NULL) {
        dbAdd(db,key,val);
    } else {
        dbOverwrite(db,key,val);
    }
    incrRefCount(val);
    removeExpire(db,key);
    signalModifiedKey(db,key);
}
```

- dbAdd()는 dictAdd()를 사용함.
- dbReplace()는 dictReplace()를 사용함.

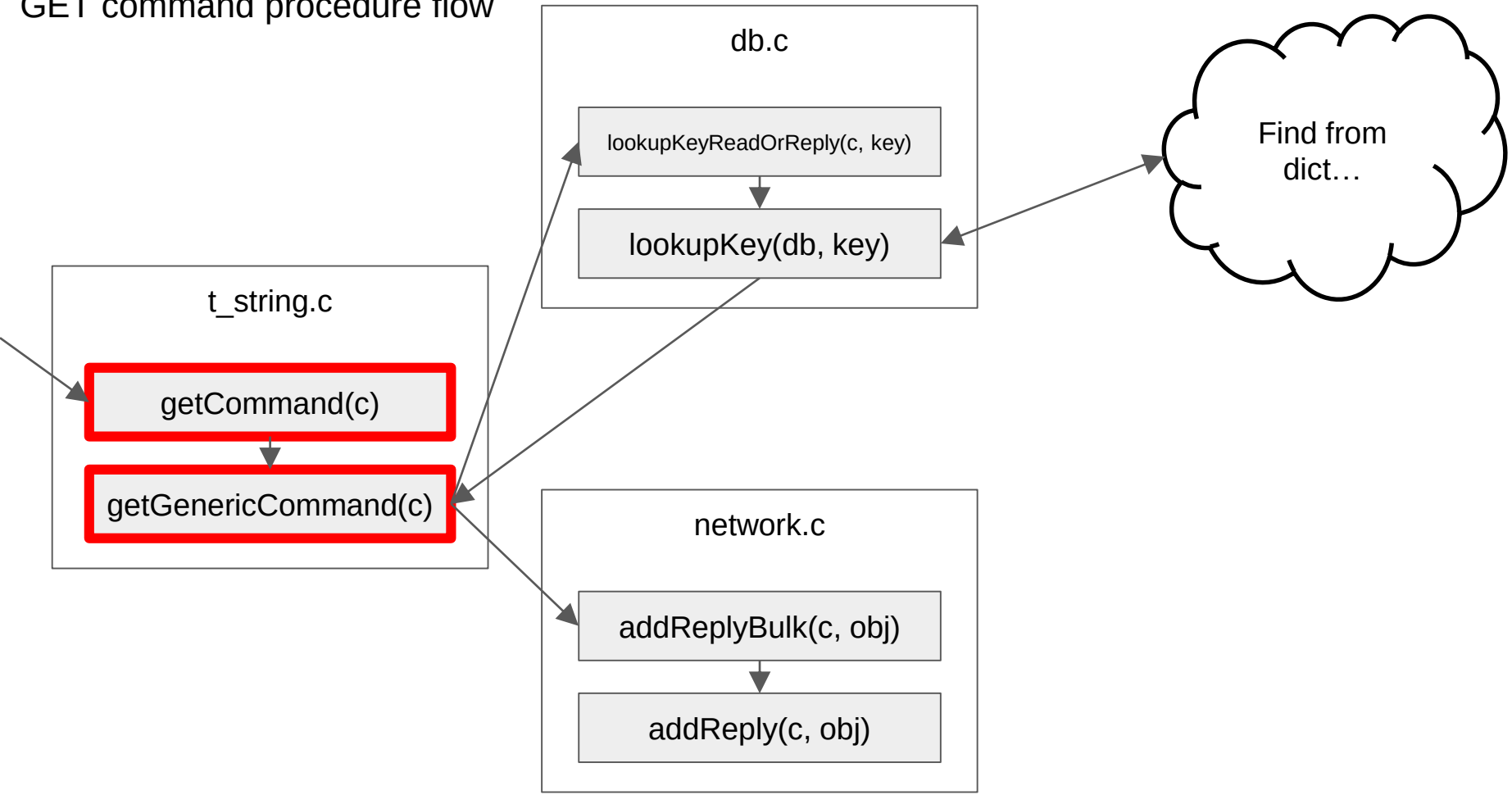
GET command procedure flow



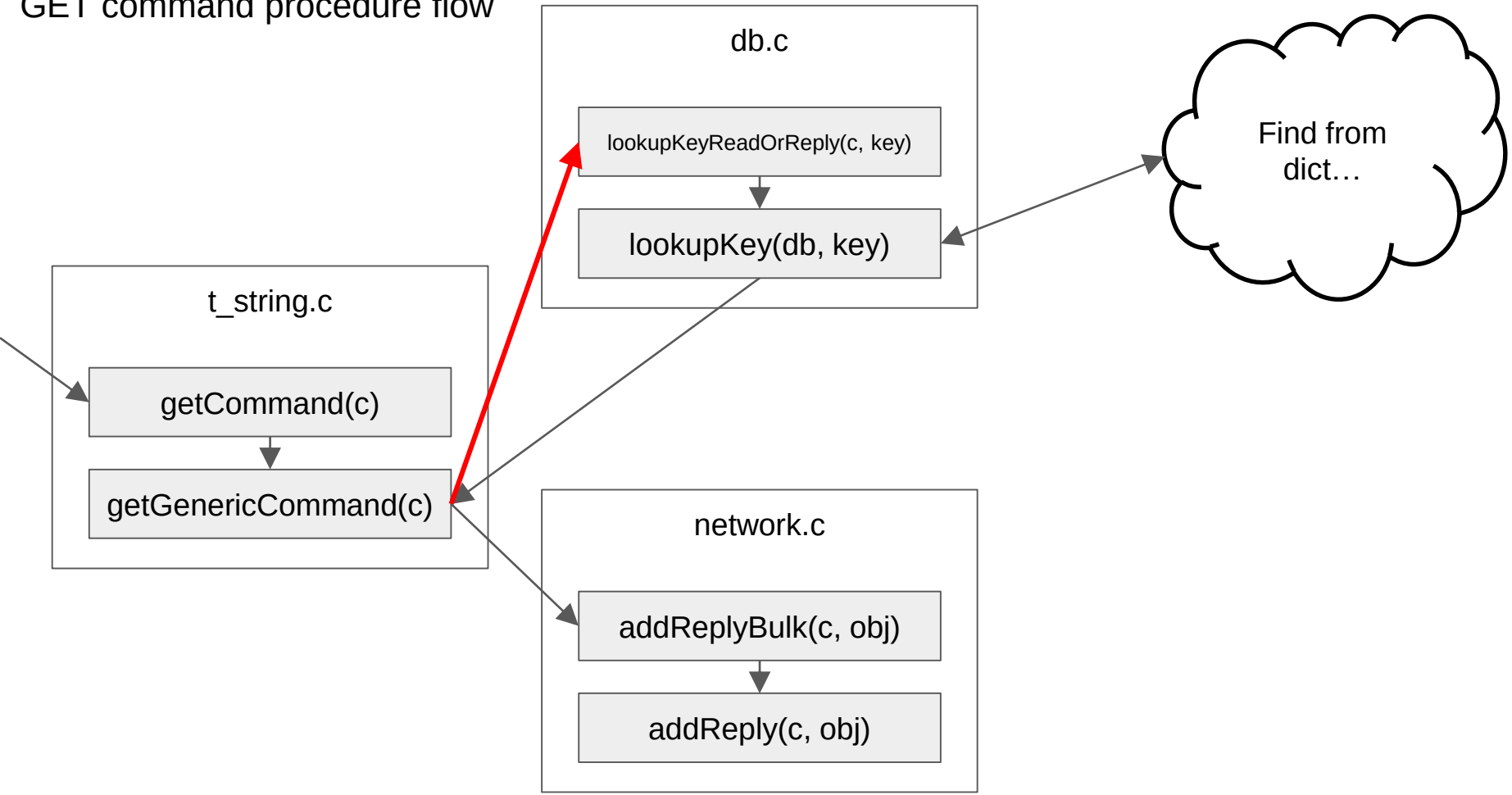
GET command procedure flow



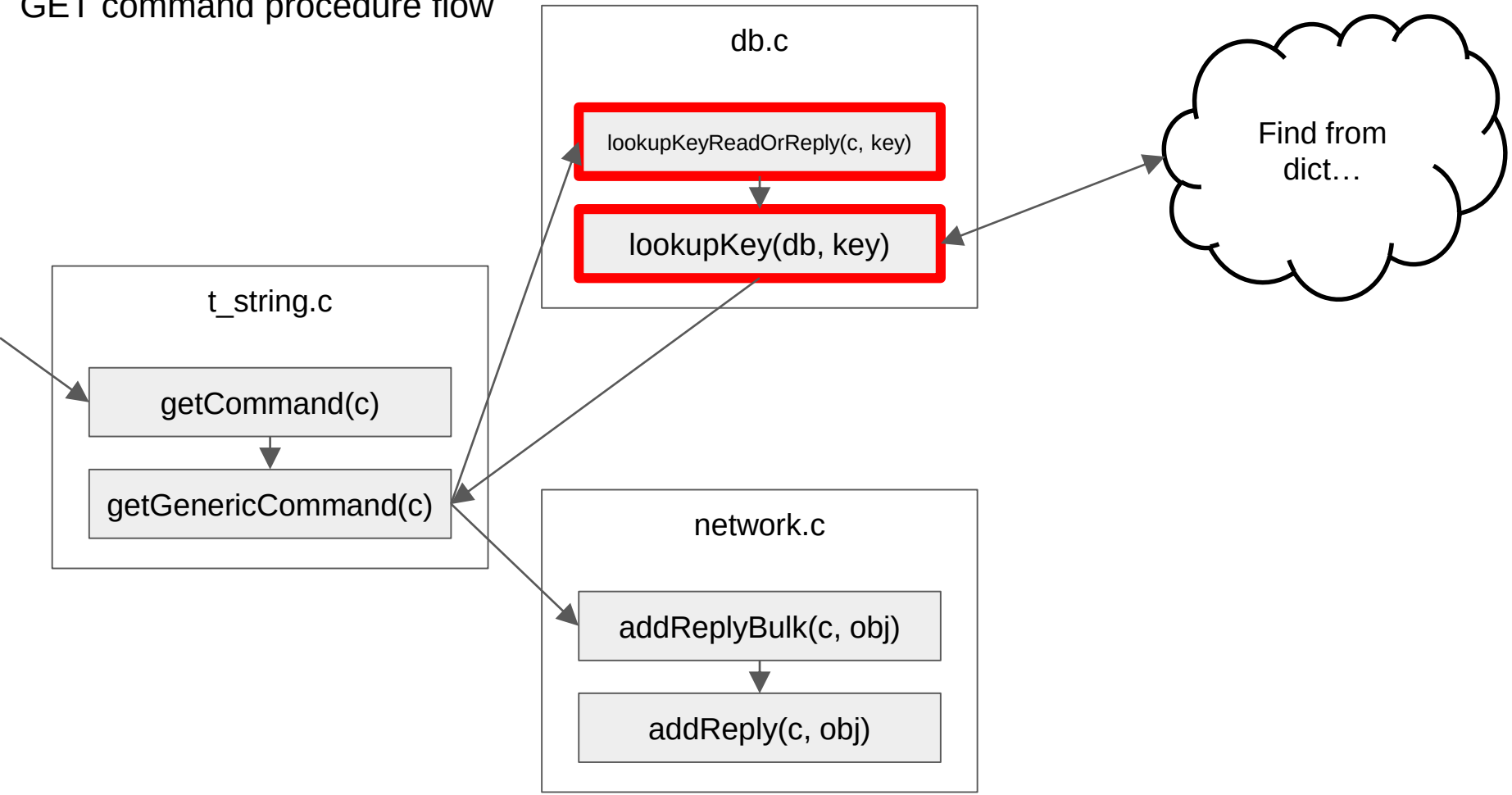
GET command procedure flow



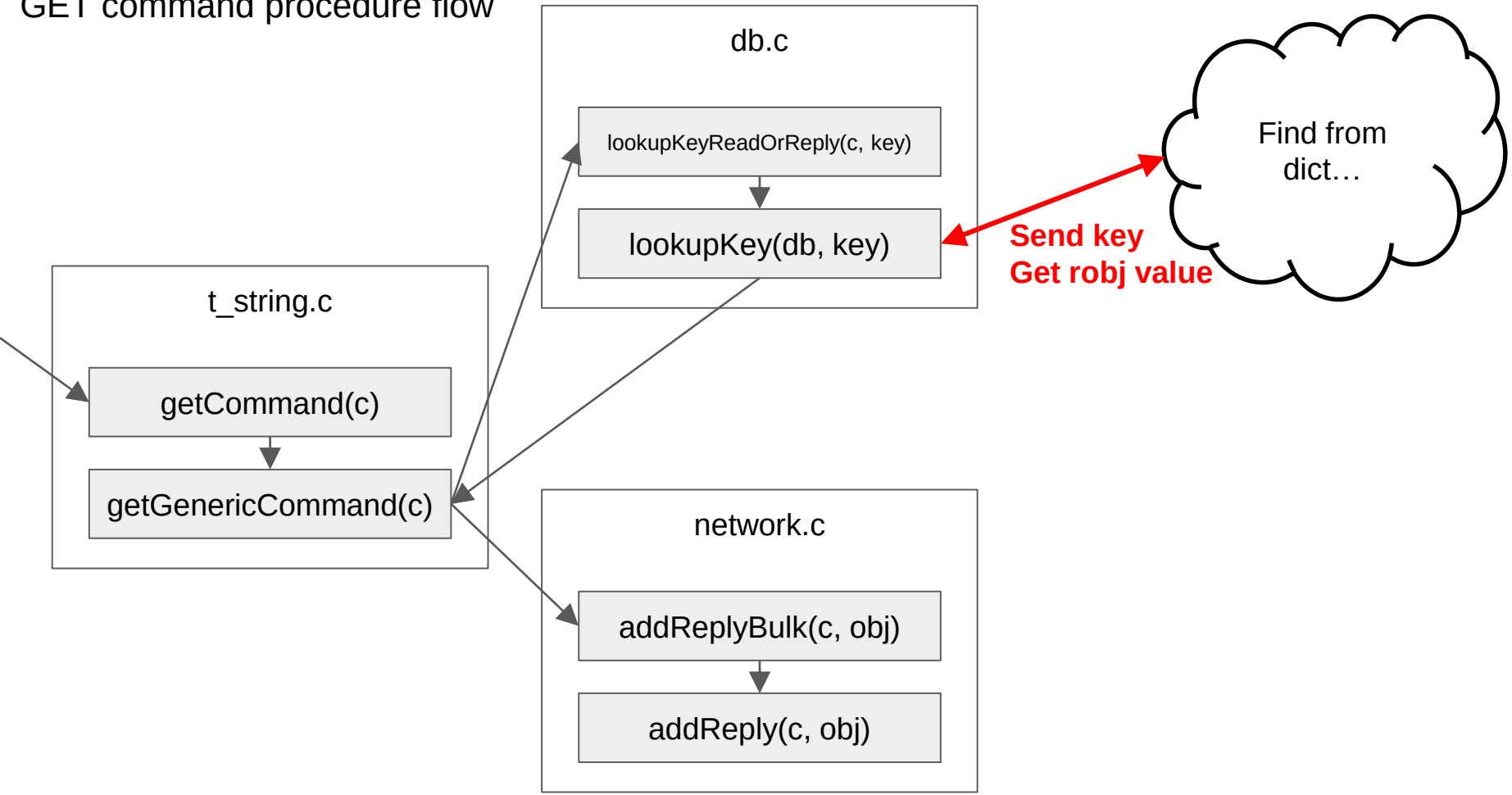
GET command procedure flow



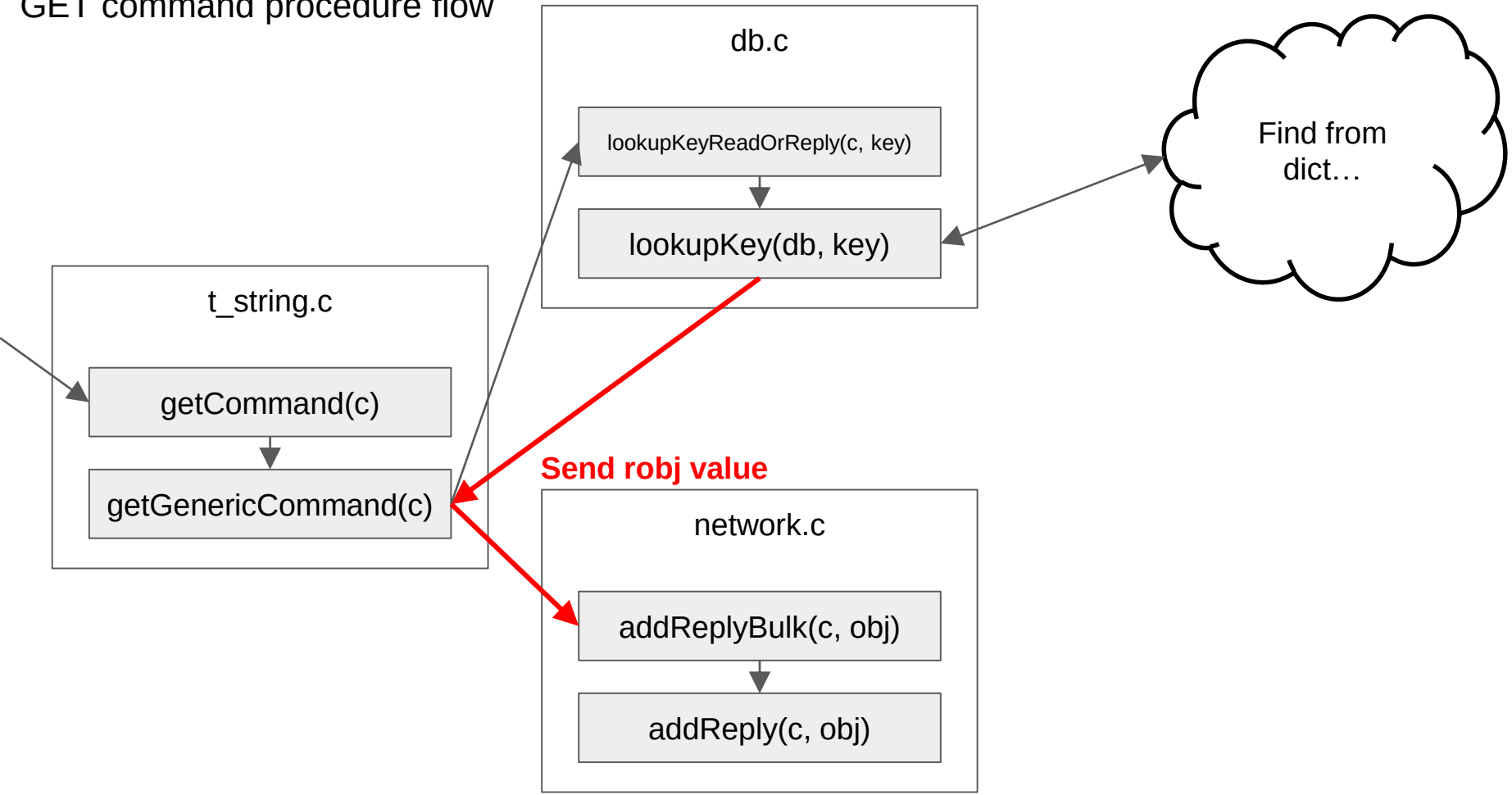
GET command procedure flow



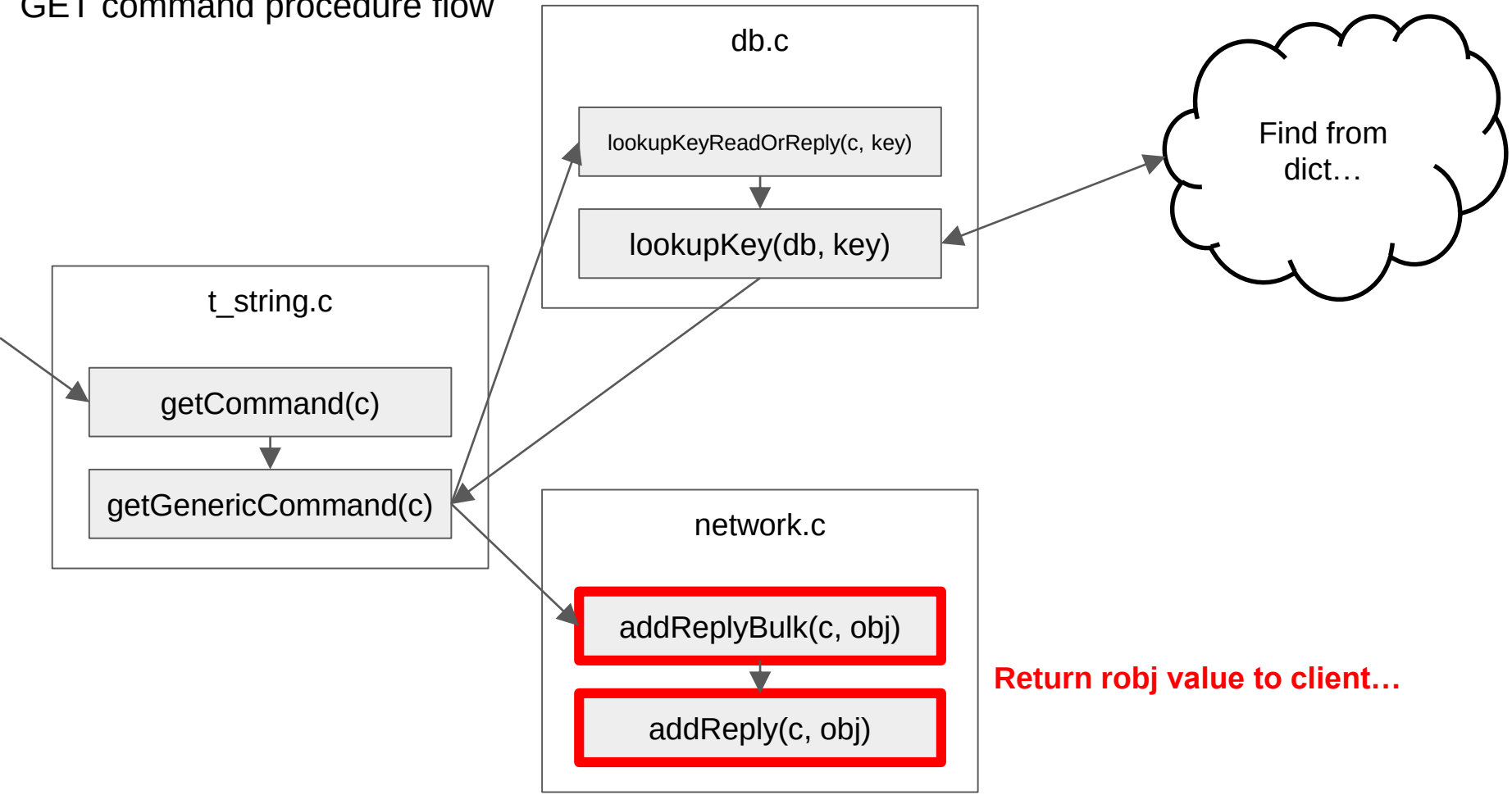
GET command procedure flow



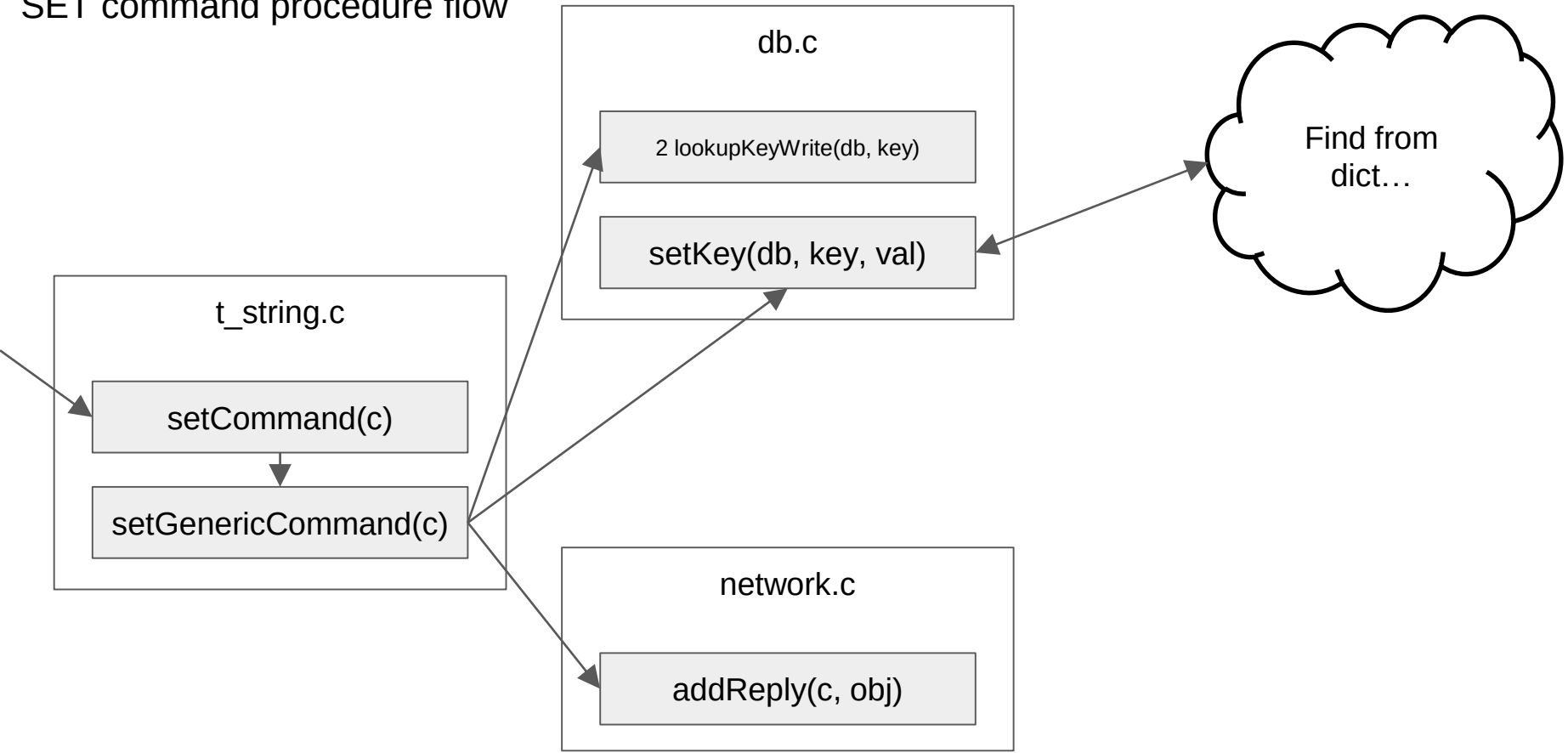
GET command procedure flow



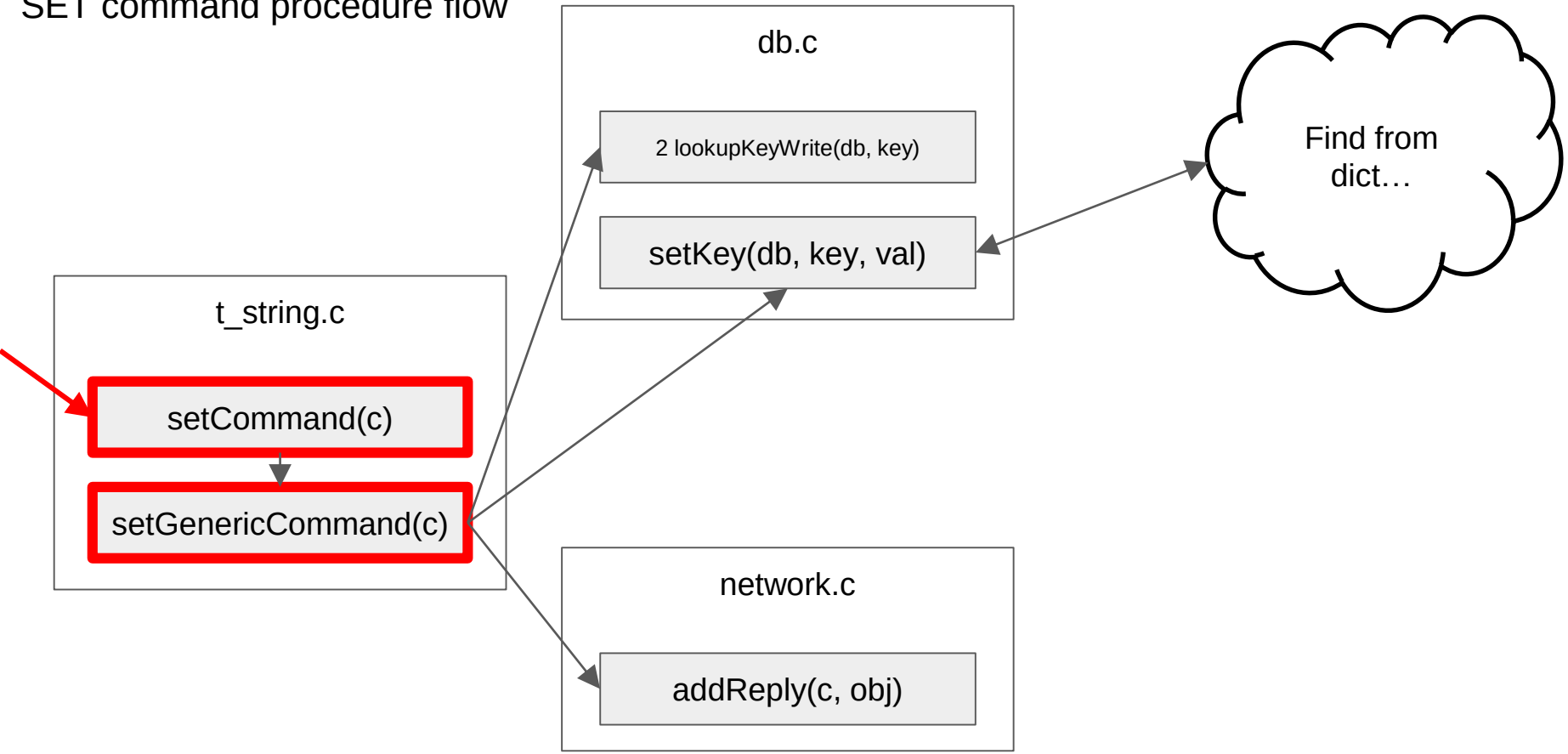
GET command procedure flow



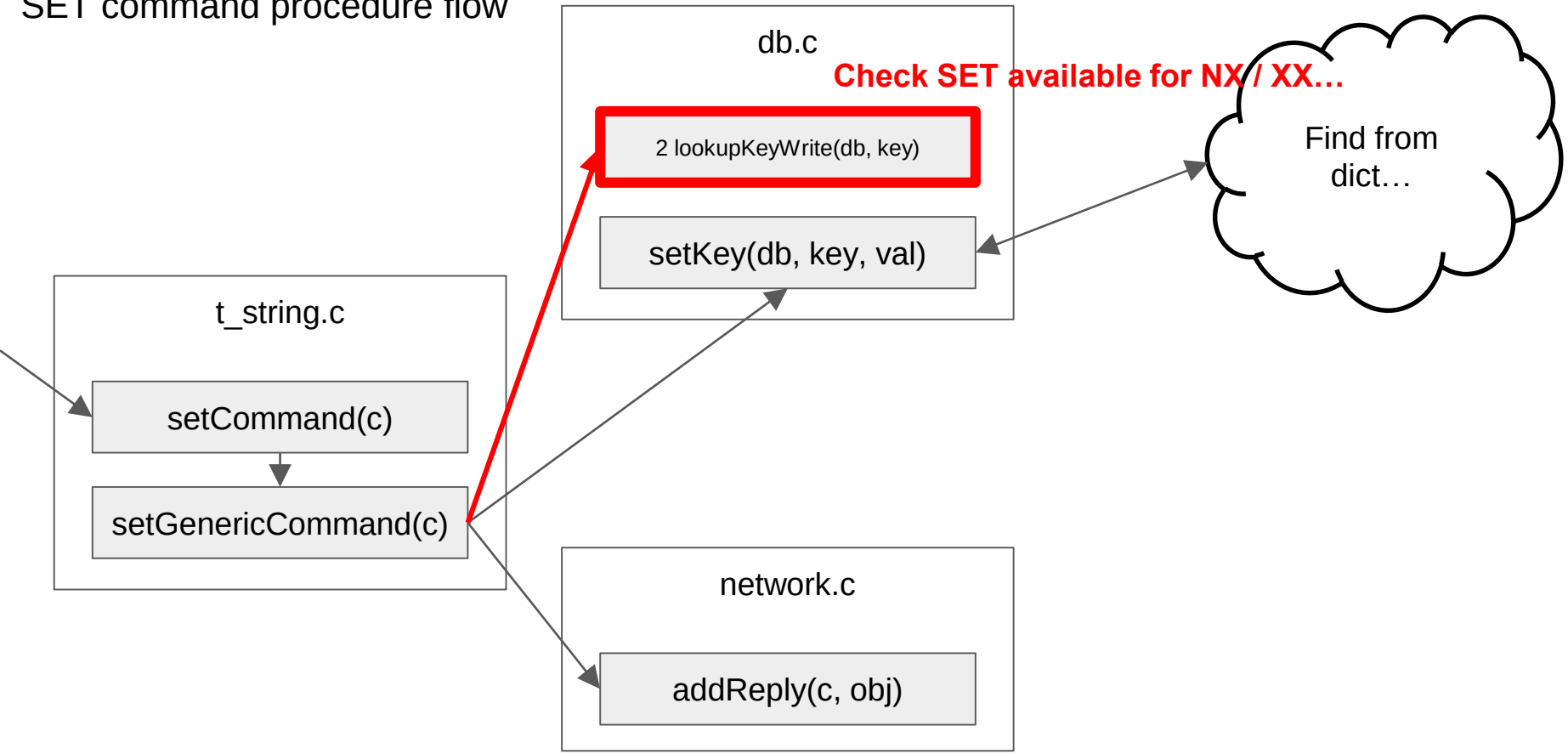
SET command procedure flow



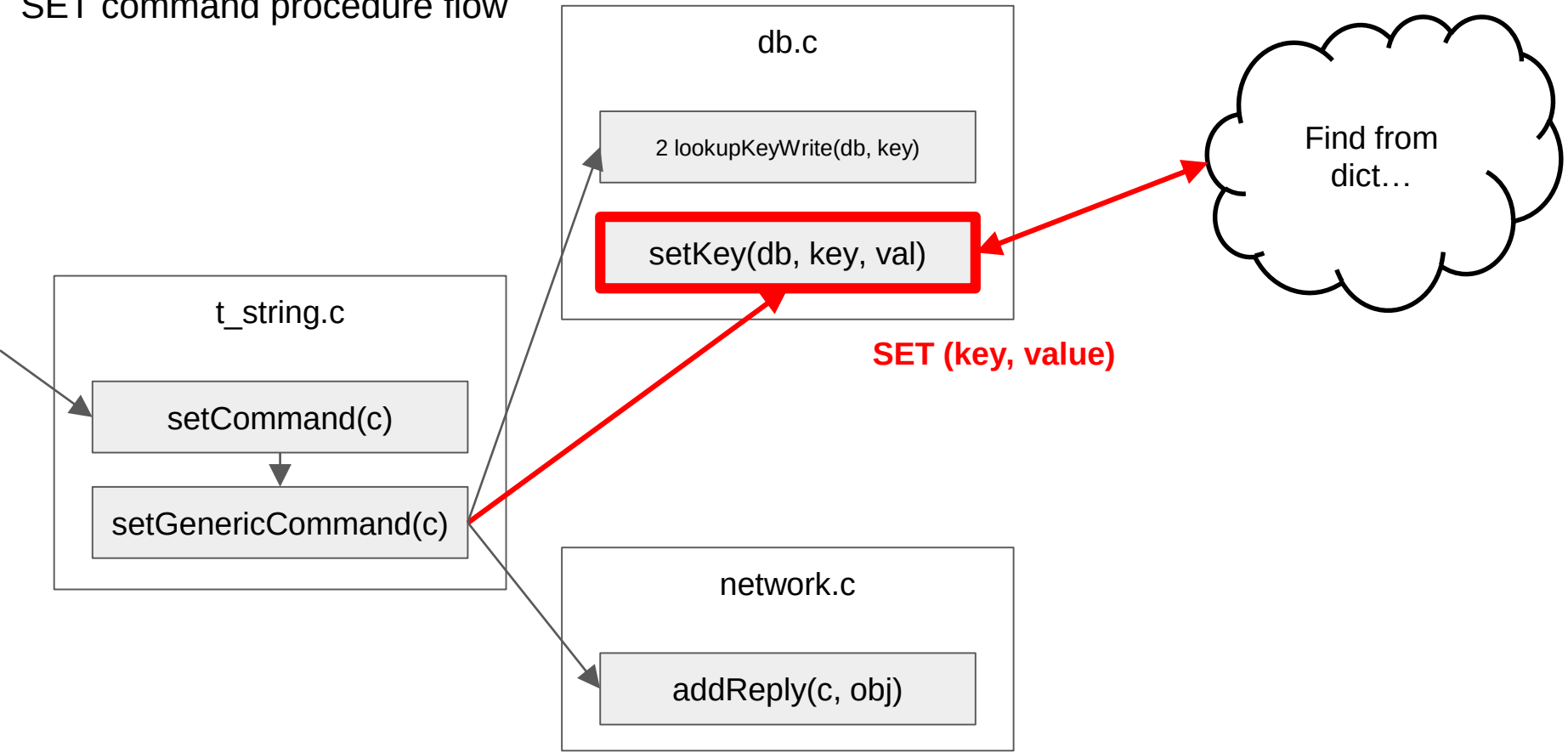
SET command procedure flow



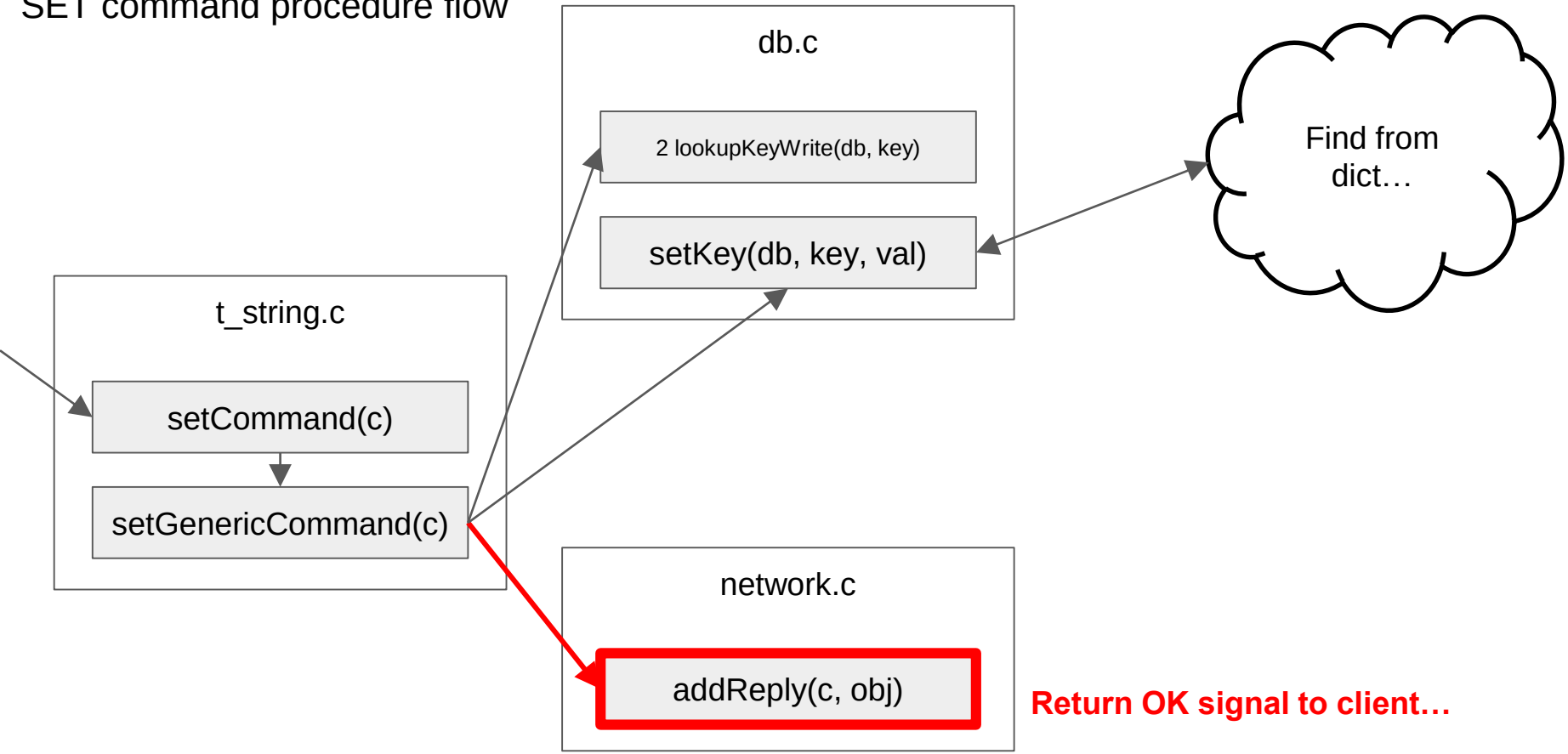
SET command procedure flow



SET command procedure flow



SET command procedure flow



Q & A