

(what's new in)

Redis 2.2

October 27th 2010
Pieter Noordhuis

Who am I?

- Live in Groningen, NL
- Redis contributor since March
- Backed by VMware



The screenshot shows a web browser window displaying a GitHub commit page. The browser's address bar shows the URL: `github.com/antirez/redis/commit/69d95c3e1cb1e6e306261957fba361cc45290a6d`. The commit message is "initial implementation for augmented zsets and the zrank command". The author is "pietern" (author) and the date is "March 03, 2010". The commit hash is "69d95c3e1cb1e6e30626". The parent commit hash is "cd5a96eedac0e3029966". The commit shows 2 changed files: `redis.c` (93 additions and 7 deletions) and `test-redis.tcl` (12 additions). The `redis.c` file content is partially visible, showing a typedef for `_redisSortOperation` and `zskiplistNode`.

git Commit 69d95c3e1cb1e6e306261957fba361cc45290a6d

github.com/antirez/redis/commit/69d95c3e1cb1e6e306261957fba361cc45290a6d

initial implementation for augmented zsets and the zrank command

commit 69d95c3e1cb1e6e30626
tree 9fed508f4493398a56b7
parent cd5a96eedac0e3029966

pietern (author)
March 03, 2010

Showing 2 changed files with 98 additions and 7 deletions.

redis.c 93

test-redis.tcl 12

redis.c View file @ 69d95c3

```
... .. @@ -456,6 +456,7 @@ typedef struct _redisSortOperation {  
456 456 typedef struct zskiplistNode {
```

So, what's new?

- Memory efficiency
- Throughput improvements
- Improved `EXPIRE` semantics

Memory efficiency

(lists)

```
typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
} listNode;
```

Memory efficiency

(lists)

```
typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
} listNode;
```



```
typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
} listNode;
```



```
typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
} listNode;
```

Memory efficiency

(lists)

LPUSH

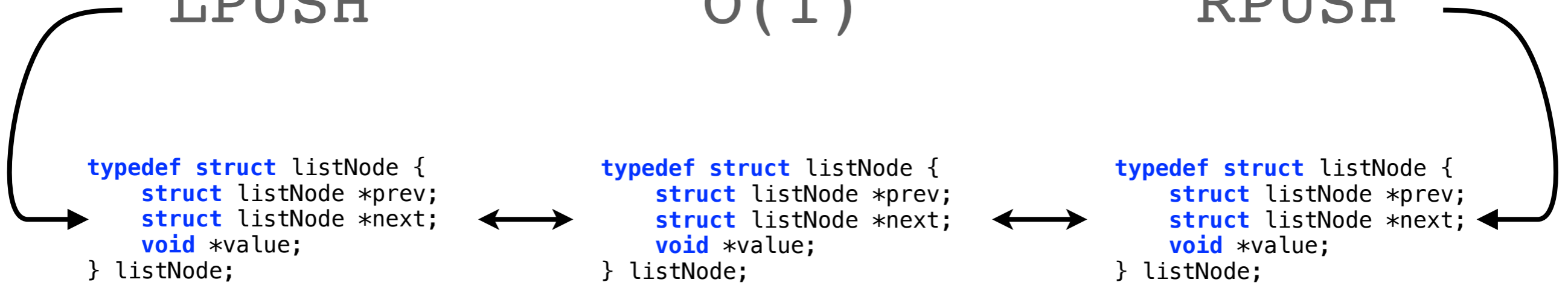
$O(1)$

RPUSH

```
typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
} listNode;
```

```
typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
} listNode;
```

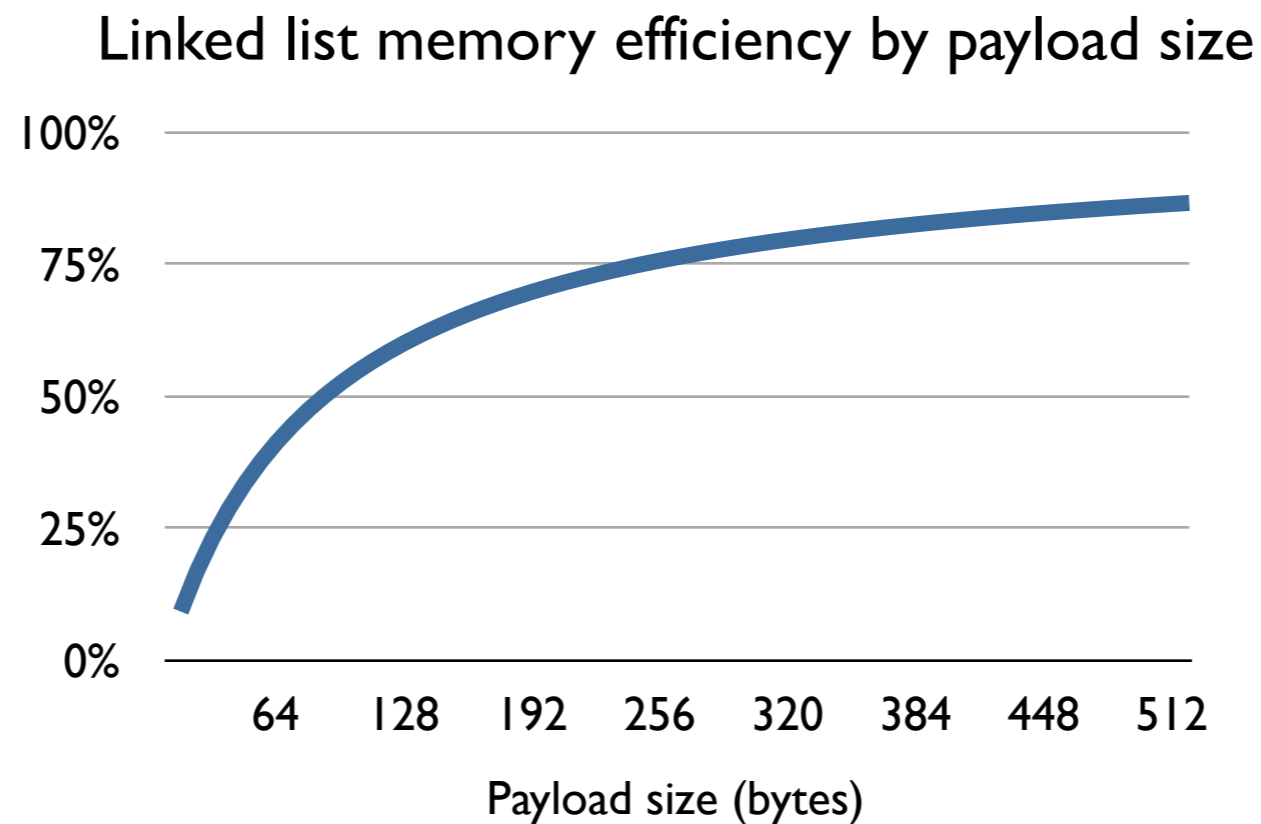
```
typedef struct listNode {  
    struct listNode *prev;  
    struct listNode *next;  
    void *value;  
} listNode;
```



Memory efficiency

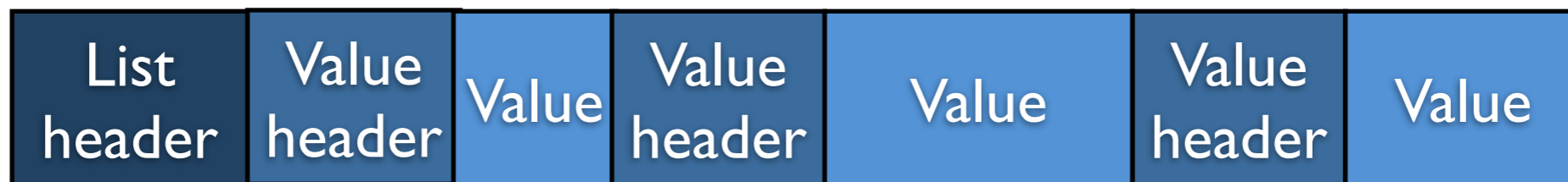
(lists)

- $O(1)$ is cool, but at what cost?
- Pointer overhead is constant per element



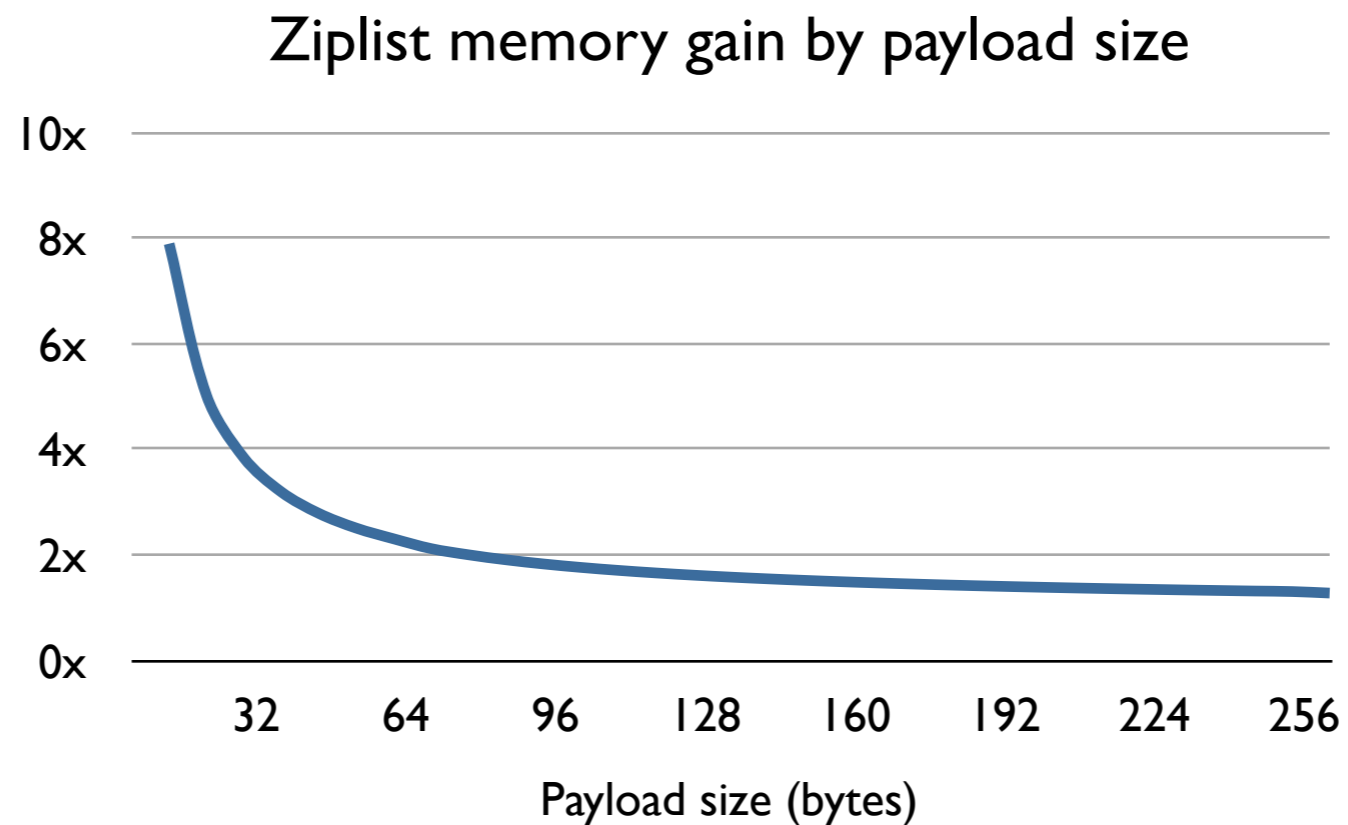
The ziplist

- Save memory by using a little more CPU
- Pack list in a single block of memory
- Value header holds encoding / value length
- `O(memory size)` LPUSH / LPOP



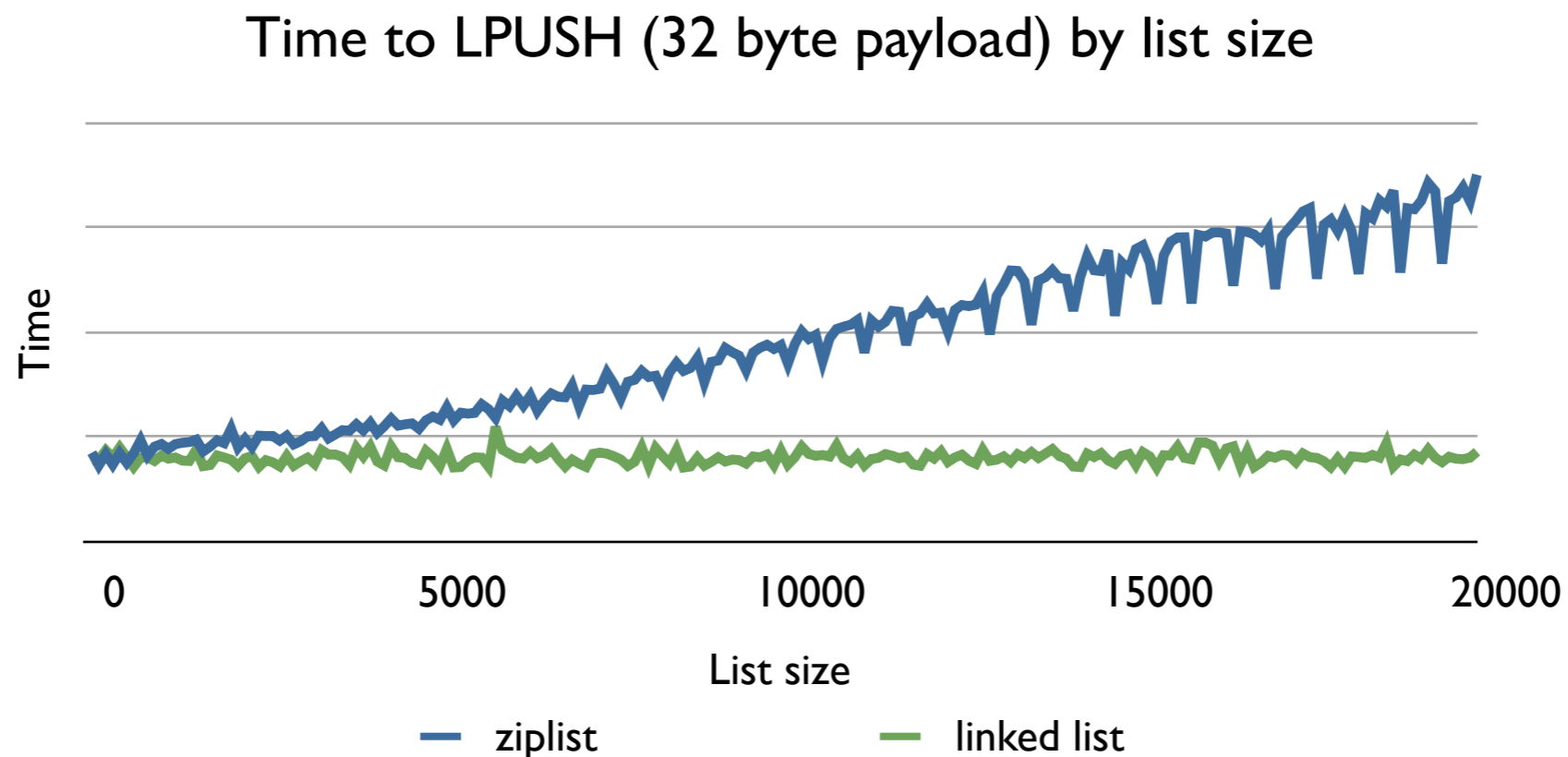
The ziplist

- Linked list memory efficiency improves for larger payload. What is the *gain* of ziplists?



The ziplist

- Gain of $\sim 4x$ for 32 byte payload
- What is the *throughput* impact?



The ziplist

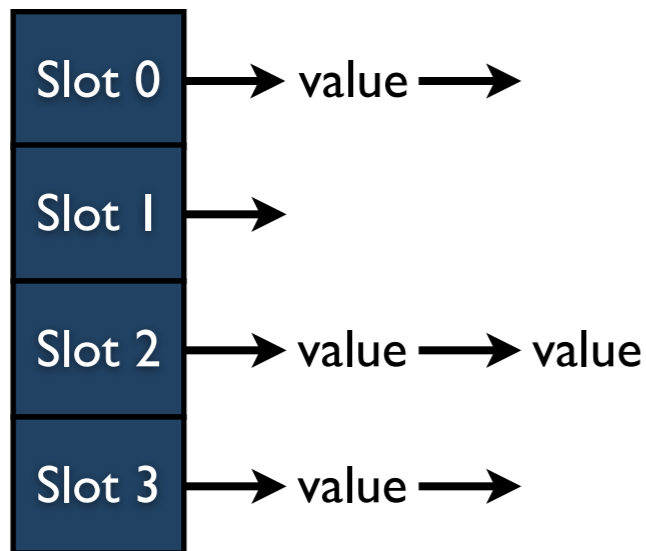
- Good fit for *small payload, limited size*
- Redis uses the hybrid approach

`list-max-ziplist-entries` (default: 1024)

`list-max-ziplist-value` (default: 32)

Memory efficiency

(sets)



- Backed by a hash table
- $O(1)$ access
- Commonly holds integers (think user IDs)

```
typedef struct dictEntry {  
    void *key;  
    void *val;  
    struct dictEntry *next;  
} dictEntry;
```

Memory efficiency

(sets)

- What is the cost of having the hash table?
- Only consider 8-byte integers
- *Remember:* lots of pointer-filled structs to guarantee $O(1)$ lookup

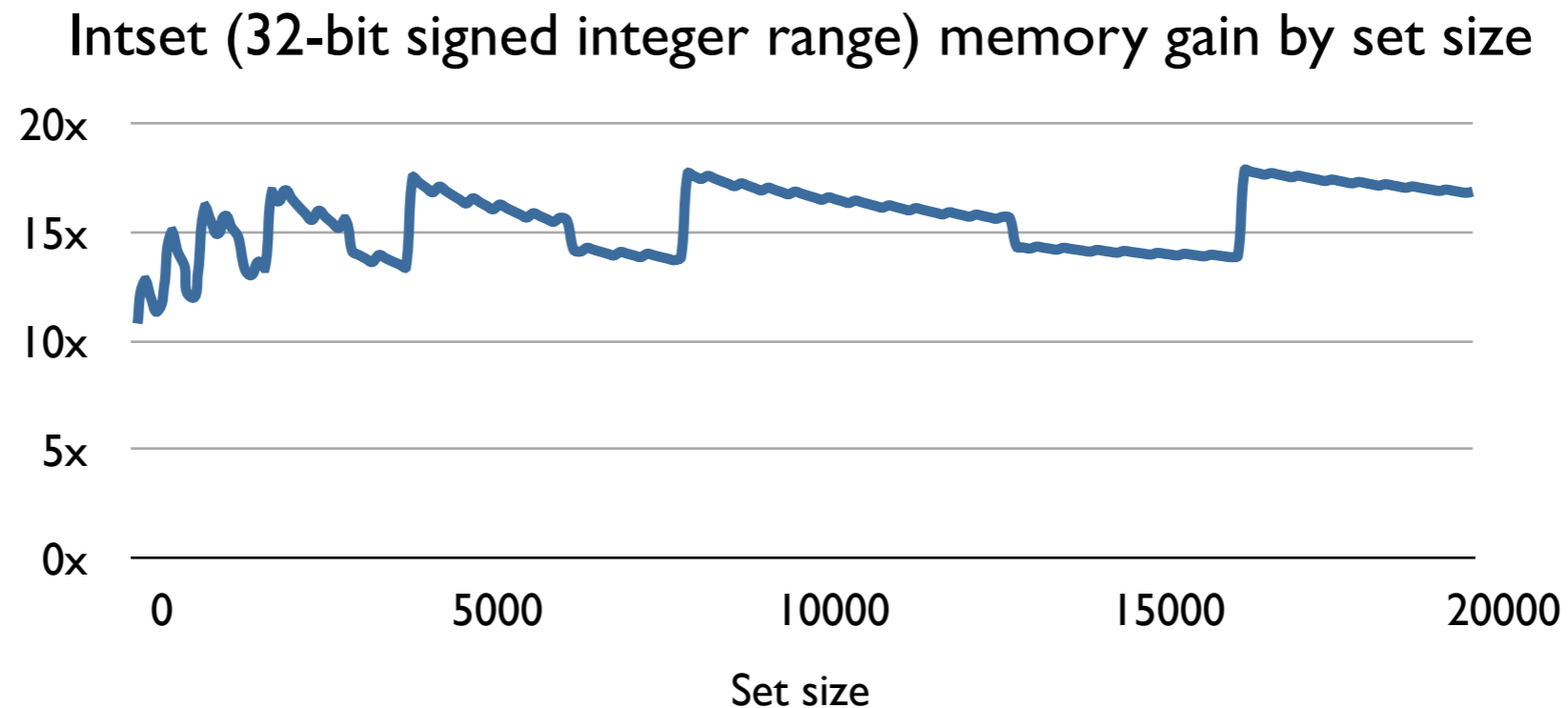
The intset

- Same idea as ziplist, *but ordered*
- Fixed width values allow binary search
- $O(\log N + \text{memory size})$ SADD / SREM
- $O(\log N)$ SISMEMBER



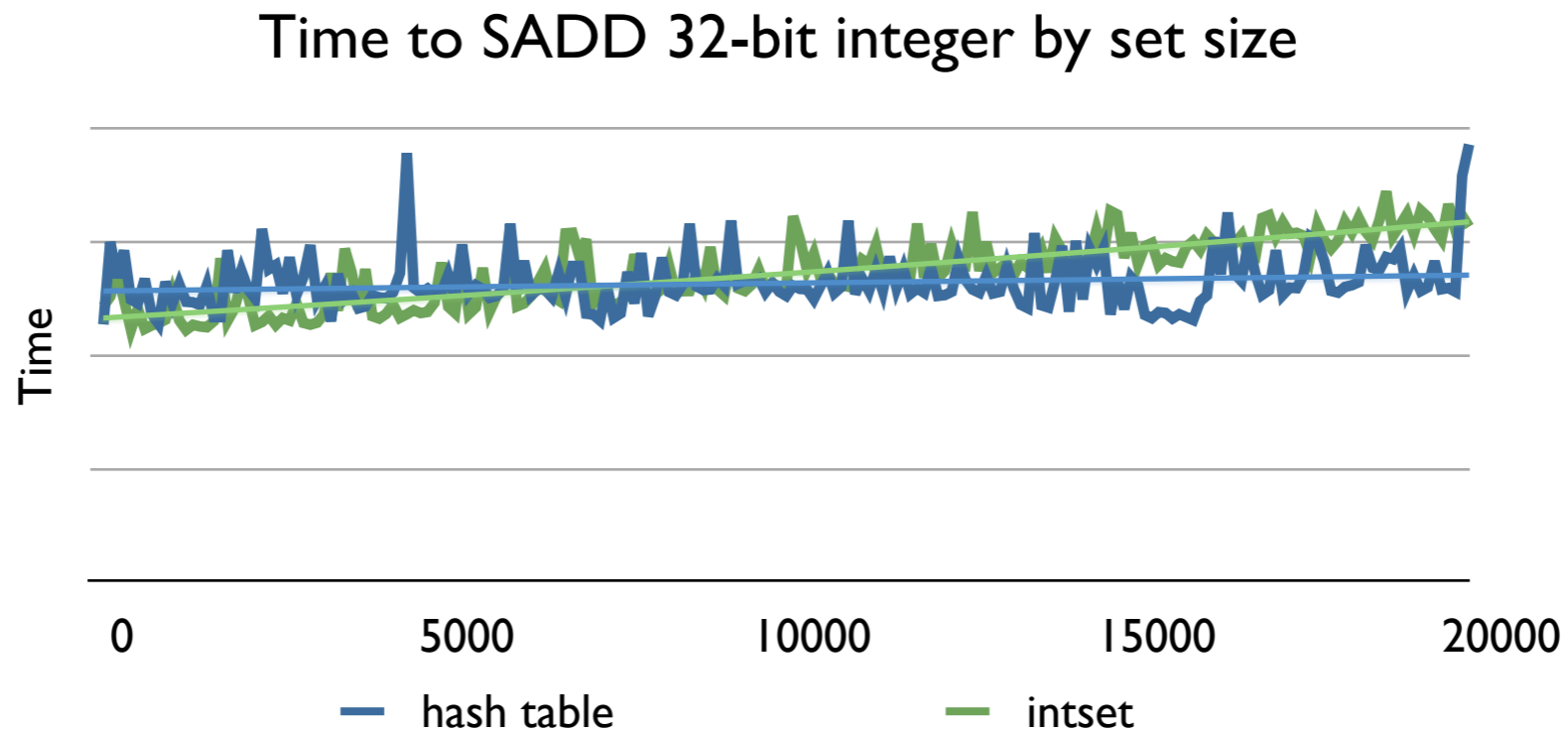
The intset

- What is the *gain* of using an intset instead of a hash table (*for this integer range*)?



The intset

- Gain of ~10-15x
- What is the *throughput* impact?



The intset

- Good fit for *size up to 20-50K*
- As with ziplists, hybrid approach

`set-max-intset-entries` (*default: 4096*)

Memory efficiency

(misc.)

- ↓ General keyspace overhead (*VM enabled*)
- ↓ Sorted set metadata (~20%)

Throughput

1. AE_READABLE
2. **read**(fd, buf);
3. **while**(request = **parseRequest**(buf))
 process(request);

1. AE_WRITABLE
2. **while**(response = **buildResponse**())
 write(fd, response);

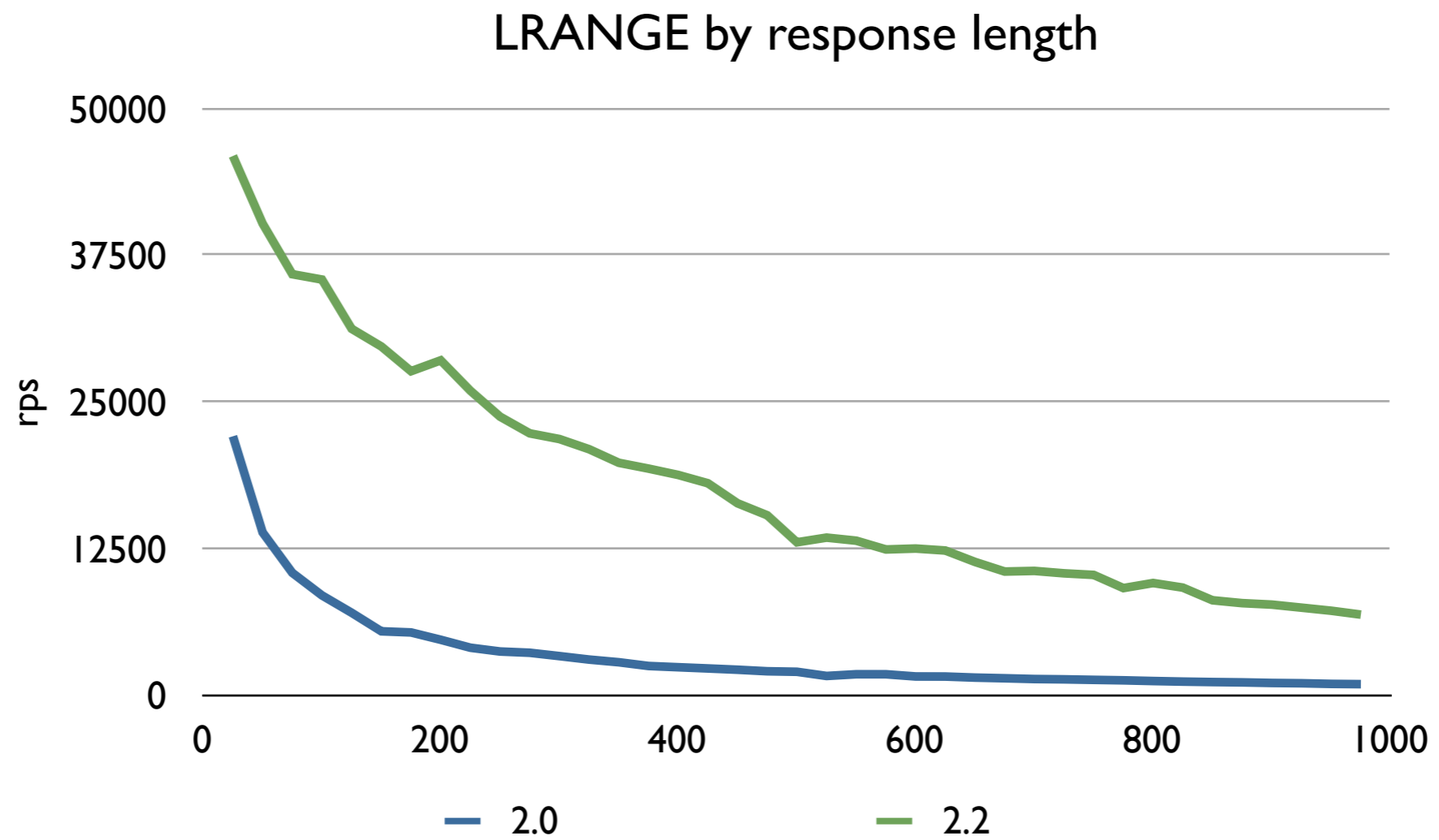
Throughput

(response)

- Glue responses into large chunks
- Fixed buffer per connection (7500 bytes)
- **+ |** for response with many bulk items

Throughput

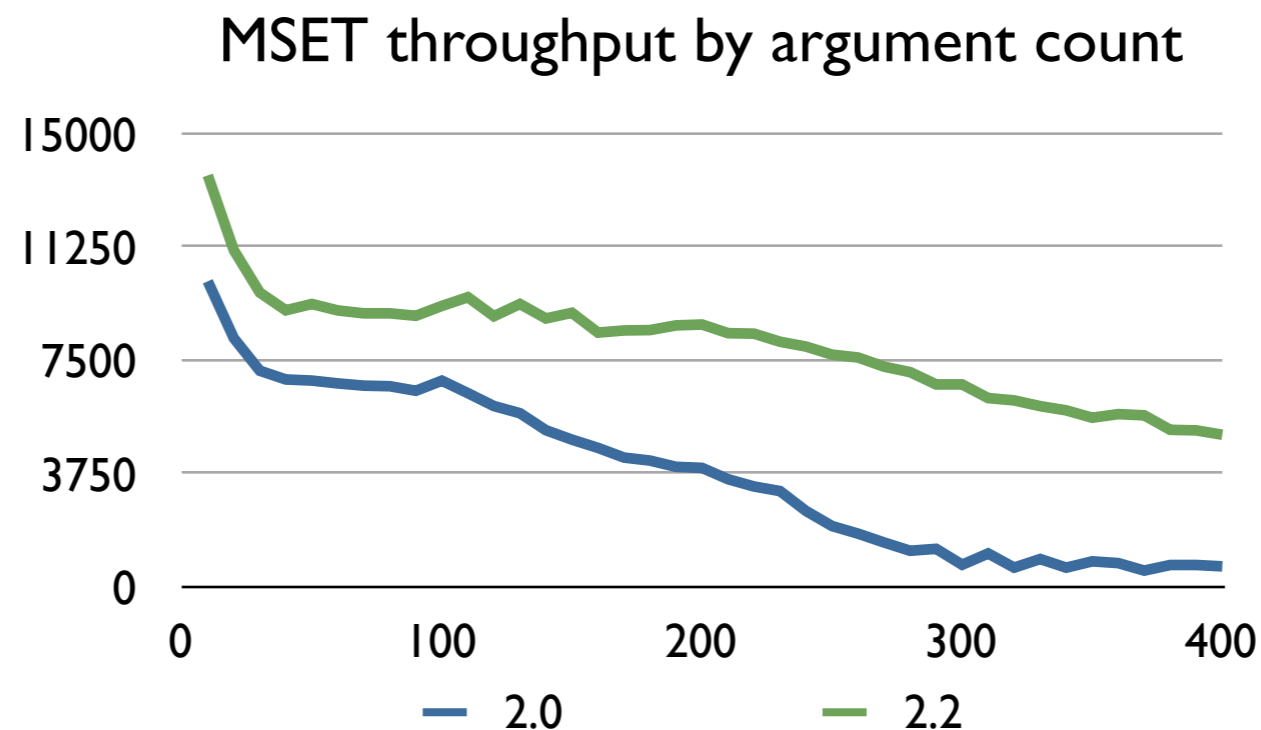
(*response*)



Throughput

(request)

- General overhaul of processing code
- Less complex & faster for multi bulk req.



EXPIRE

(new behavior)

- Volatile keys (keys with `EXPIRE` set) are:
- `<= 2.0`: deleted on write
- `>= 2.2`: not touched

EXPIRE

(new behavior)

```
redis> SADD online:<timestamp> 15
(integer) 1
redis> EXPIRE online:<timestamp> 600
(integer) 1
redis> SADD online:<timestamp> 23
(integer) 1
redis> SADD online:<timestamp> 27
(integer) 1
redis> SMEMBERS online:<timestamp>
1. "15"
2. "23"
3. "27"
```


max-memory

(purge policies)

- When `max-memory` is hit, purge:
 - volatile key by random, TTL, LRU
 - any key by random, LRU (*memcached*)

Other things in 2.2

- Unix Sockets (tav)
- `LINSERT`, `LPUSHX`, `RPUSHX` (Robey Pointer)
- See “`git diff 2.0.0`” for things I’m forgetting...

Work in progress

- hiredis: easy to use C-client that ships with a decoupled protocol parser
- Memory fragmentation (tcmalloc, slabs, ...)

Questions?