Memory-Efficient

Searchable Symmetric Encryption

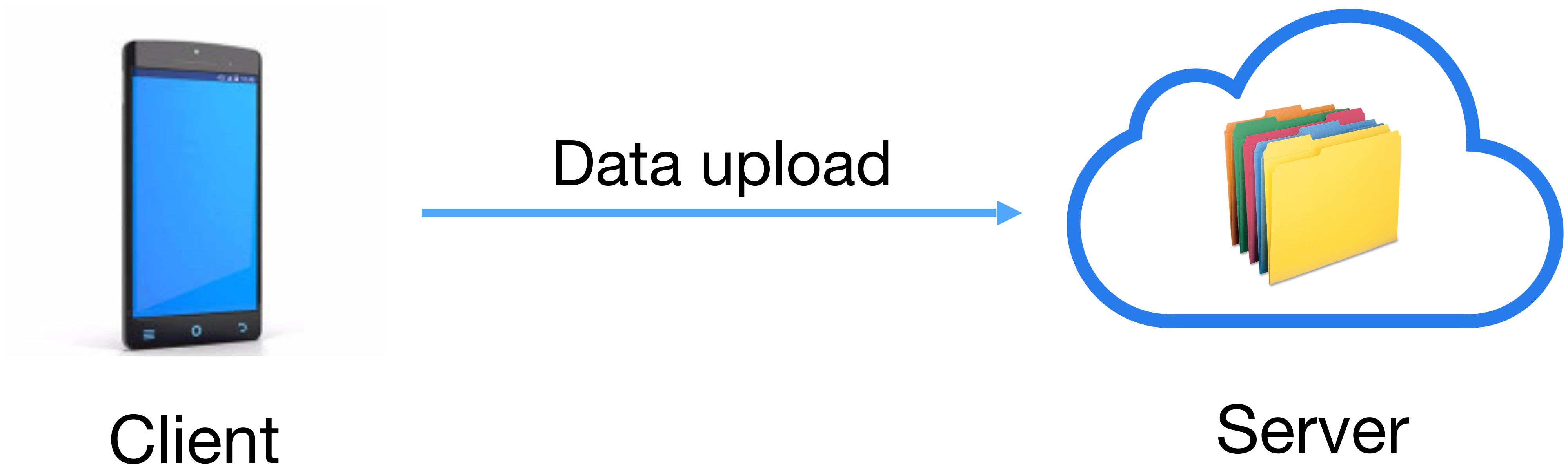
Angèle Bossuat, Raphael Bost, Pierre-Alain Fouque,
Brice Minaud, Michael Reichle

ia.cr/2021/716

Roadmap

- Searchable Encryption: Introduction
- Problem statement: Page-efficient encryption.
- A solution: Tethys.

Outsourcing storage

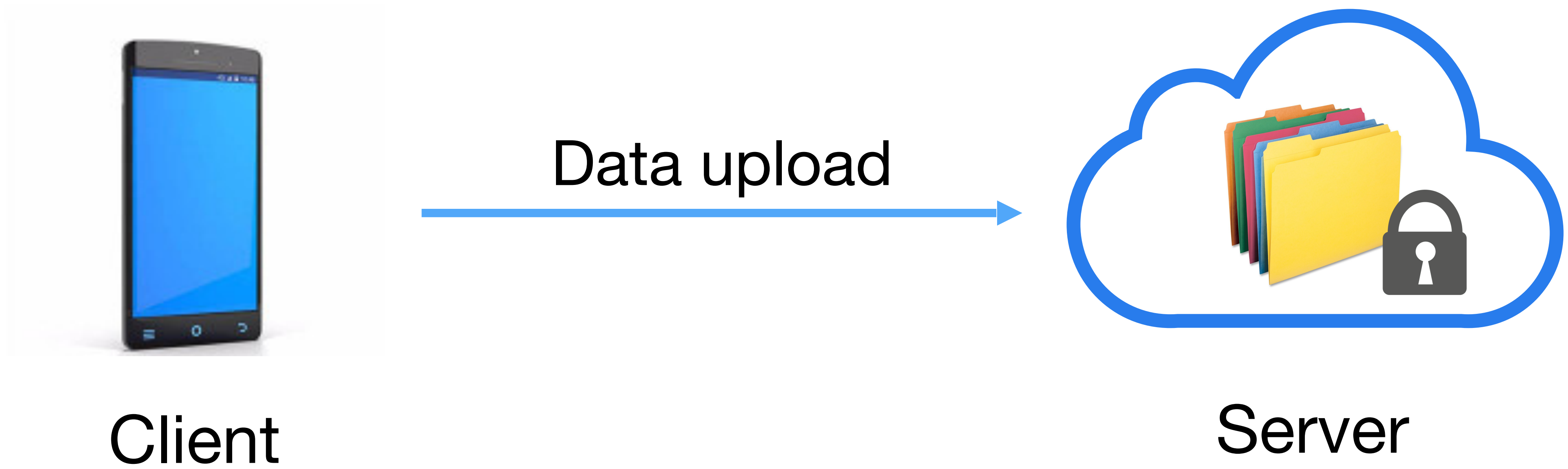


Scenario: Client outsources storage of sensitive data to Server.

Examples:

- Company/hospital outsourcing client/patient info.
- Private email service.

Outsourcing storage



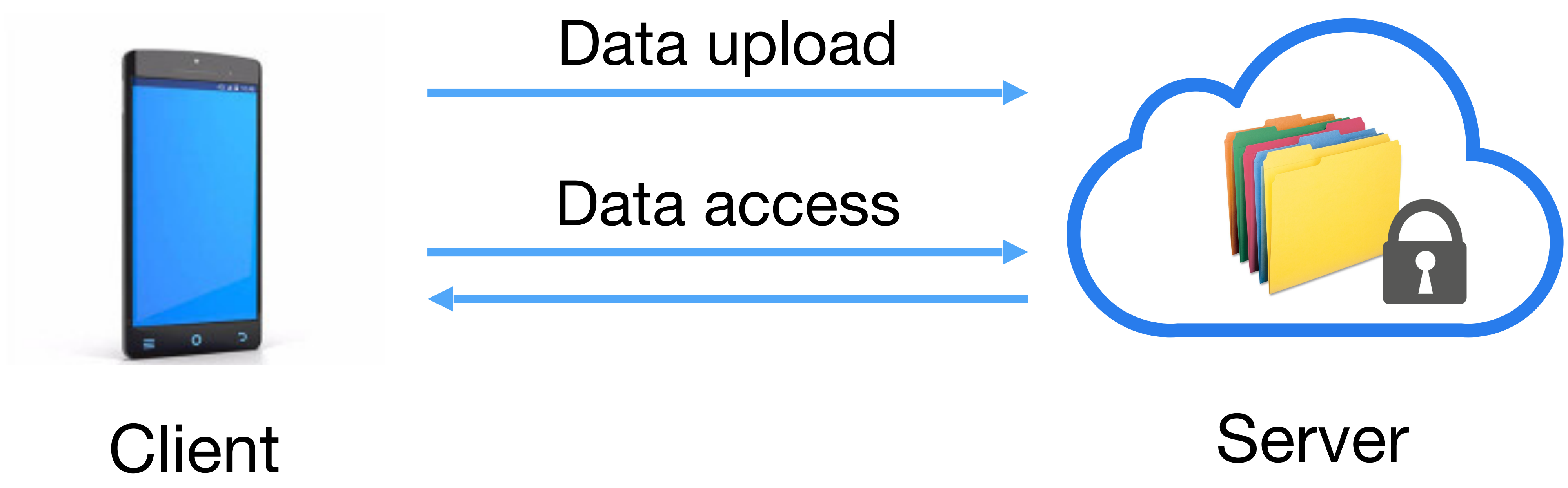
Scenario: Client outsources storage of sensitive data to Server.

Examples:

- Company/hospital outsourcing client/patient info.
- Private email service.

Sensitive data → encryption is needed.

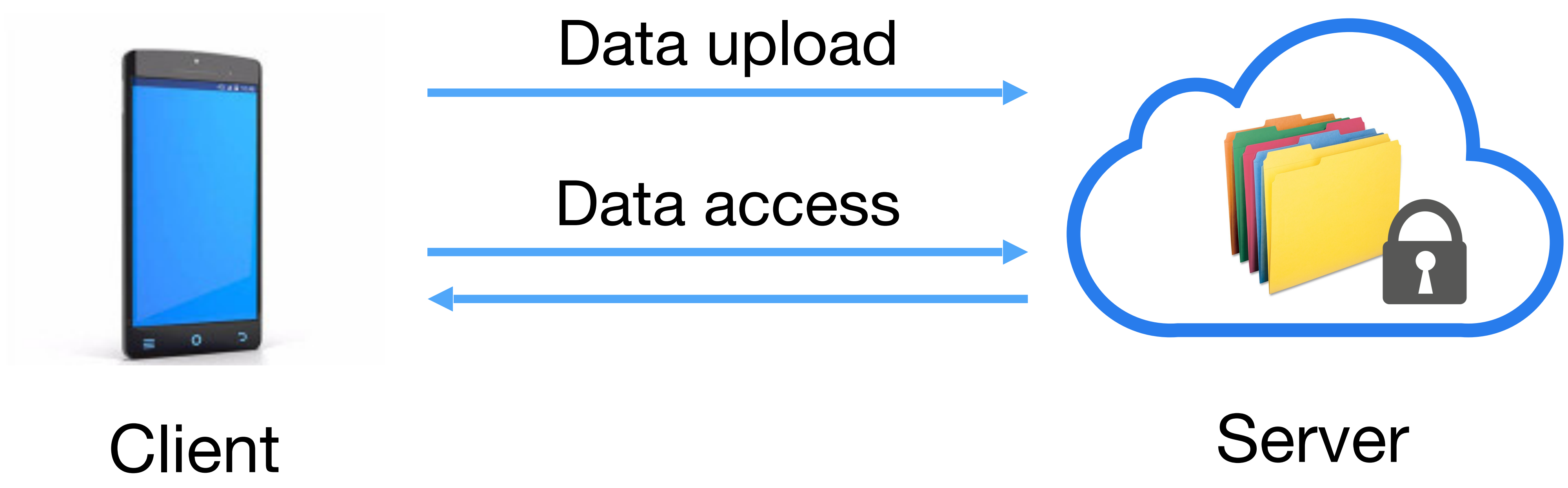
Searchable Encryption



Searchable Encryption (SE):

- Client stores encrypted database on server.
- Client can perform search queries.
- Privacy of data and queries is desired.

Searchable Encryption



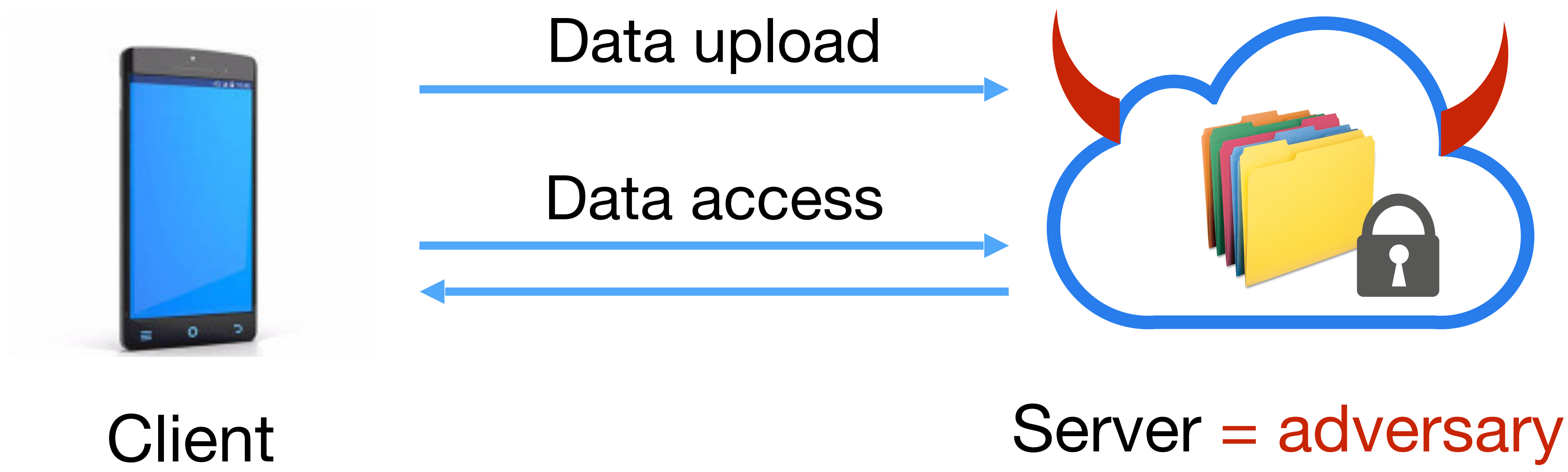
Searchable Encryption (SE):

- Client stores encrypted database on server.
- Client can perform search queries.
- Privacy of data and queries is desired.

Static SE: search queries.

Dynamic SE: search + update queries.

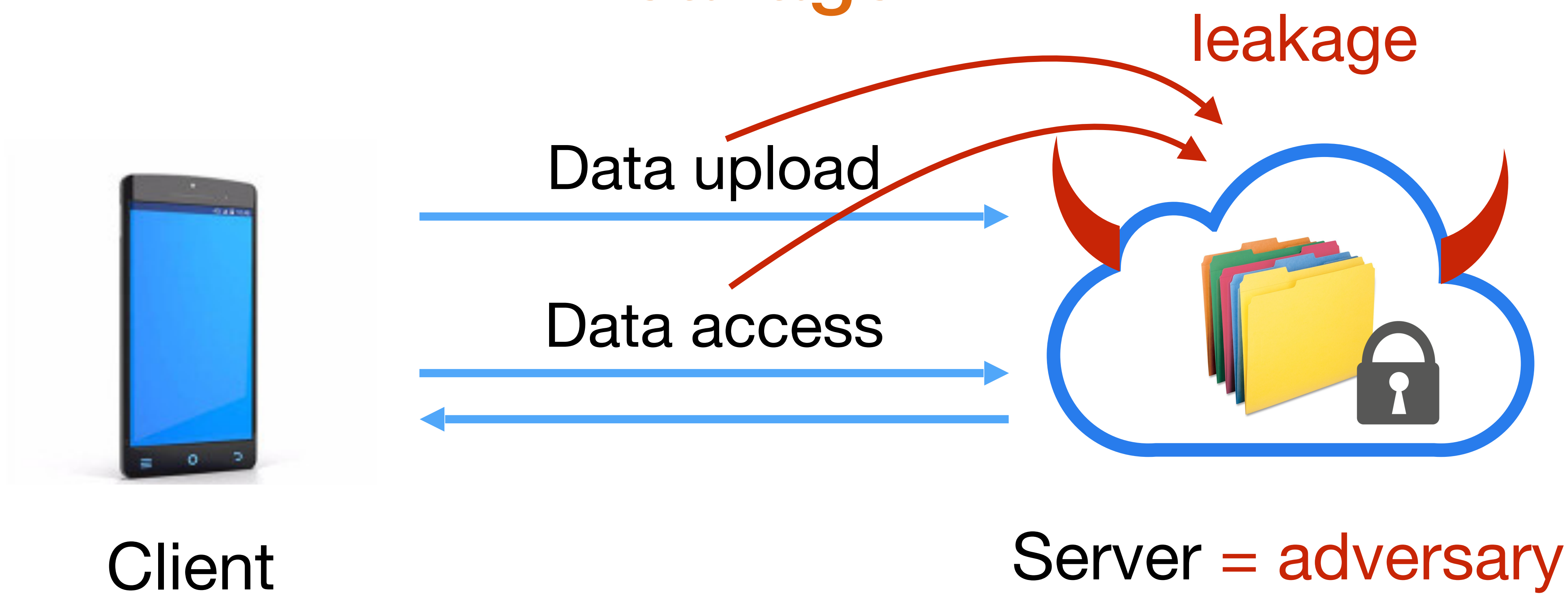
Searchable Encryption



Adversary: **honest-but-curious** server.

Security goal: **privacy** of data and queries.

Leakage

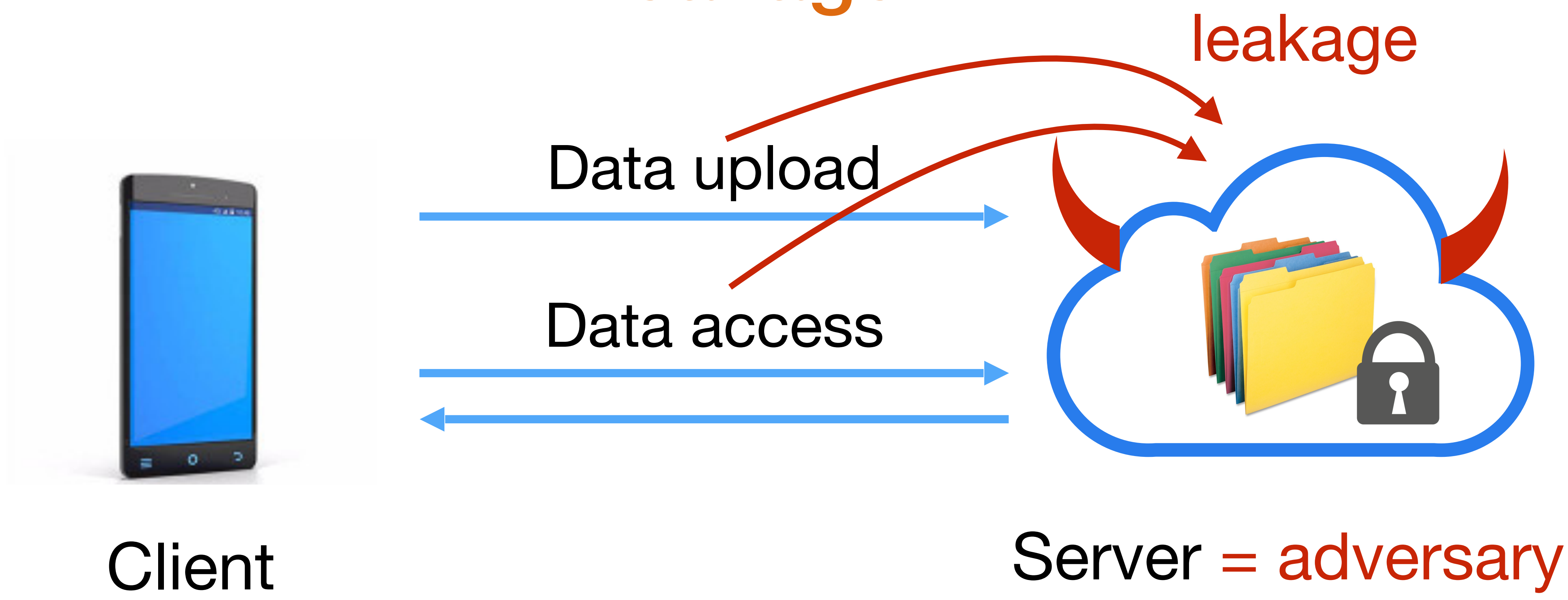


Generic solutions (FHE) are infeasible at scale
→ for efficiency reasons, some **leakage** is allowed.

Example:

- **Setup** leaks: total number of elements in database.
- **Search** leaks: repetition of queries + IDs of documents matching each query.

Leakage



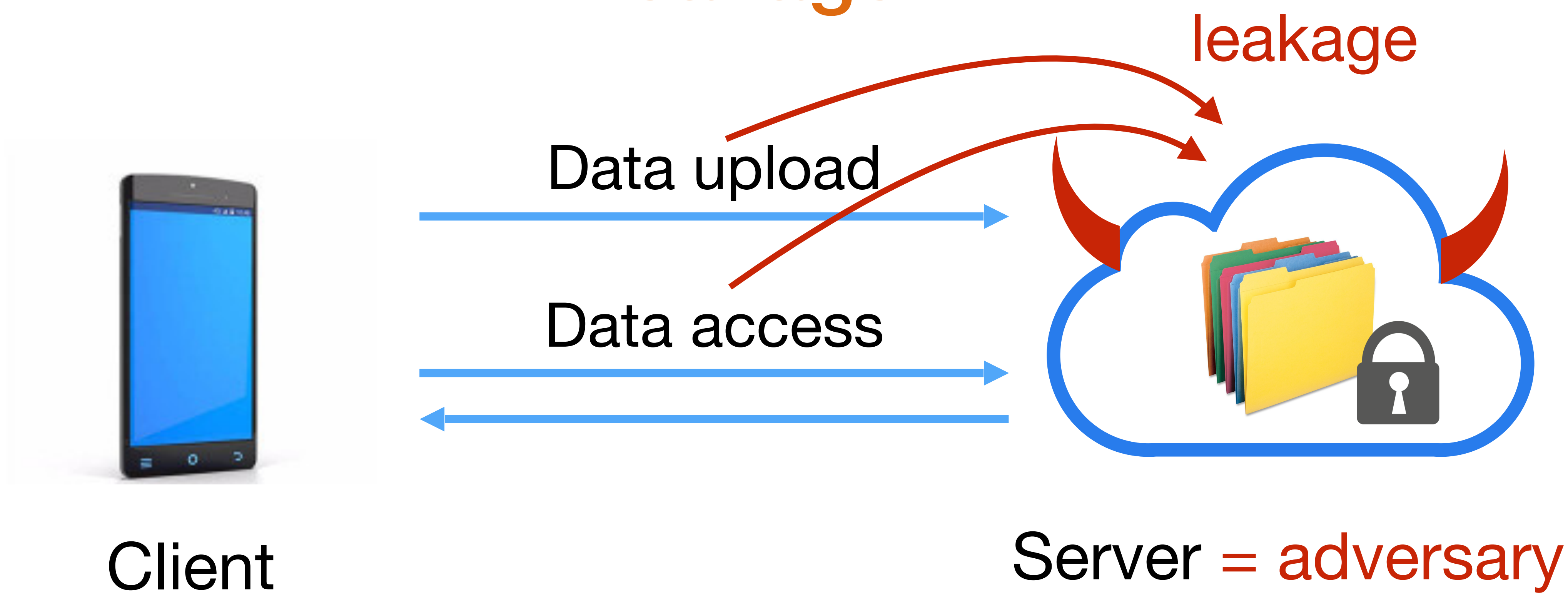
Generic solutions (FHE) are infeasible at scale
→ for efficiency reasons, some **leakage** is allowed.

Example:

- **Setup** leaks: total number of elements in database.
- **Search** leaks: repetition of queries + IDs of documents matching each query.

Security model: Server learns **nothing** except leakage.

Leakage



Generic solutions (FHE) are infeasible at scale
→ for efficiency reasons, some **leakage** is allowed.

Example:

- **Setup** leaks: total number of elements in database.
- **Search** leaks: repetition of queries + IDs of documents matching each query.

Security model: Server learns **nothing** except leakage.

No leakage about **unqueried** keywords.

State of the Art

- **No perfect solution.**

  Every solution is a trade-off between functionality and security.

- **Large amount of literature.**

  [AKSX04], [BCLO09], [PKV+14], [BLR+15], [NKW15], [KKNO16], [LW16], [FVY+17], [SDY+17], [DP17], [HLK18], [PVC18], [MPC+18]…

- **A few “complete” solutions:**

  Mylar (for web apps)

  CryptDB (handles most of SQL)

  ➔ Cipherbase (Microsoft), Encrypted BigQuery (Google), …

State of the Art

- **No perfect solution.**

  Every solution is a trade-off between functionality and security.

- **Large amount of literature.**

  [AKSX04], [BCLO09], [PKV+14], [BLR+15], [NKW15], [KKNO16], [LW16], [FVY+17], [SDY+17], [DP17], [HLK18], [PVC18], [MPC+18]…

- **A few “complete” solutions:**

  Mylar (for web apps)

  CryptDB (handles most of SQL)

  ⚠️ Very controversial security

  ➔ Cipherbase (Microsoft), Encrypted BigQuery (Google), …

State of the Art

- **No perfect solution.**

  Every solution is a trade-off between functionality and security.

- **Large amount of literature.**

  [AKSX04], [BCLO09], [PKV+14], [BLR+15], [NKW15], [KKNO16], [LW16], [FVY+17], [SDY+17], [DP17], [HLK18], [PVC18], [MPC+18]…

- **A few “complete” solutions:**

  Mylar (for web apps)

  CryptDB (handles most of SQL)

  ⚠️ Very controversial security

  ➔ Cipherbase (Microsoft), Encrypted BigQuery (Google), …

Today: single-keyword SSE.

Single-keyword SSE

id_1 id_2 id_3 ...

Reverse index:

"*car*" $\mapsto$ id_1, id_3

"*duck*" $\mapsto$ id_2, id_3, id_6, ...

...

Single-keyword SSE

id_1 id_2 id_3 …

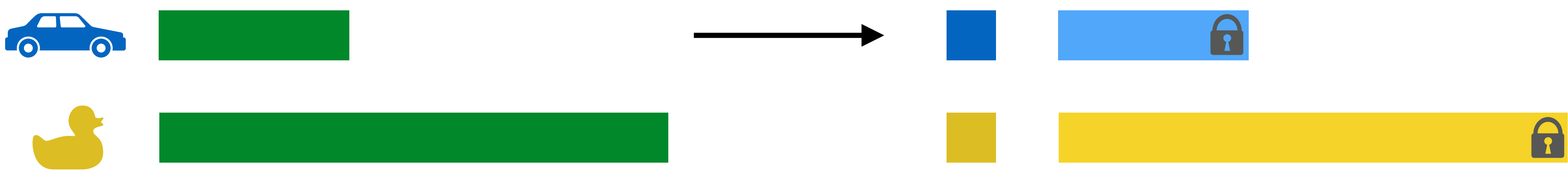
Reverse index:

…

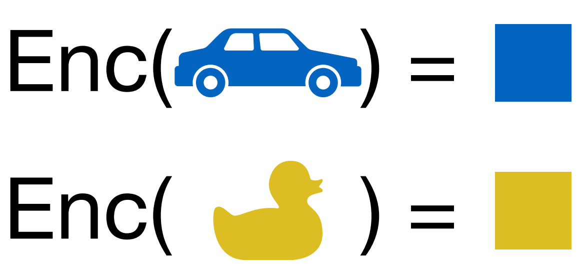
Single-keyword SSE: Setup

Reverse index:

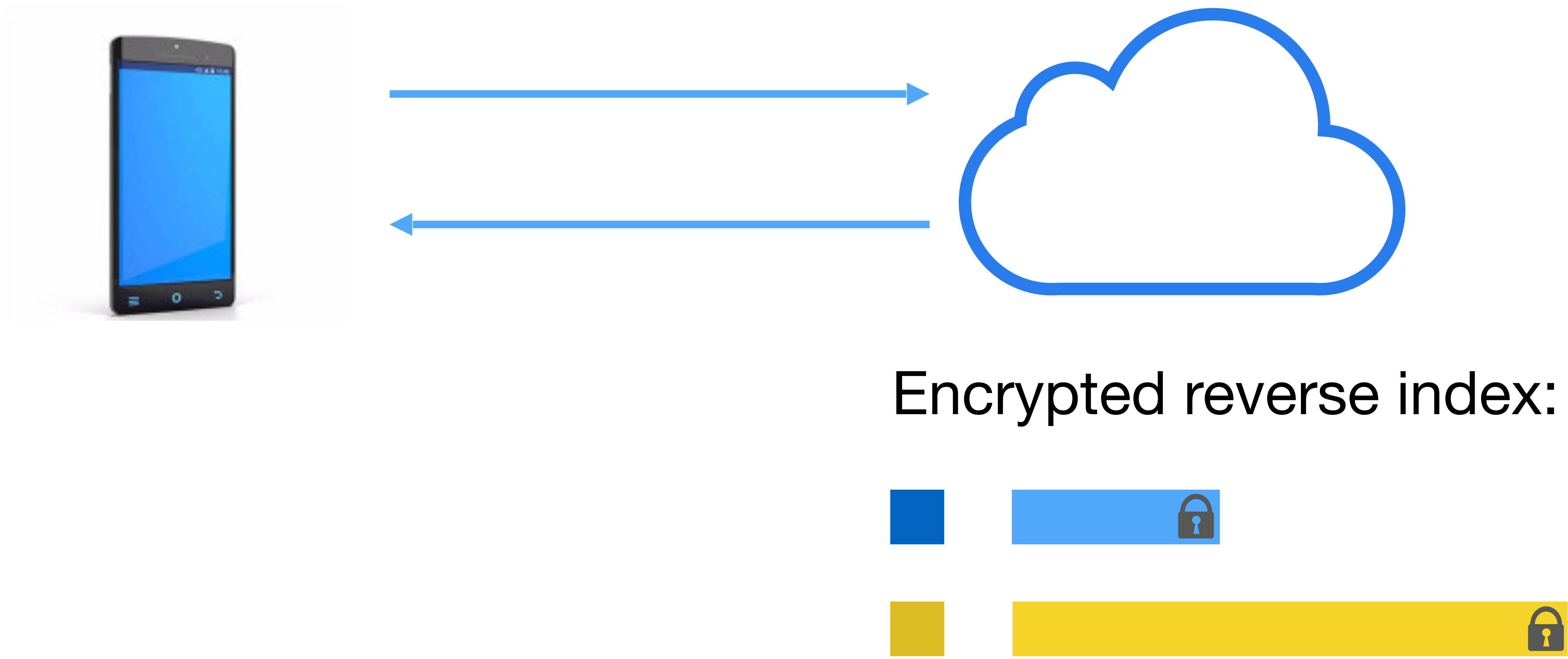
Encrypted reverse index:



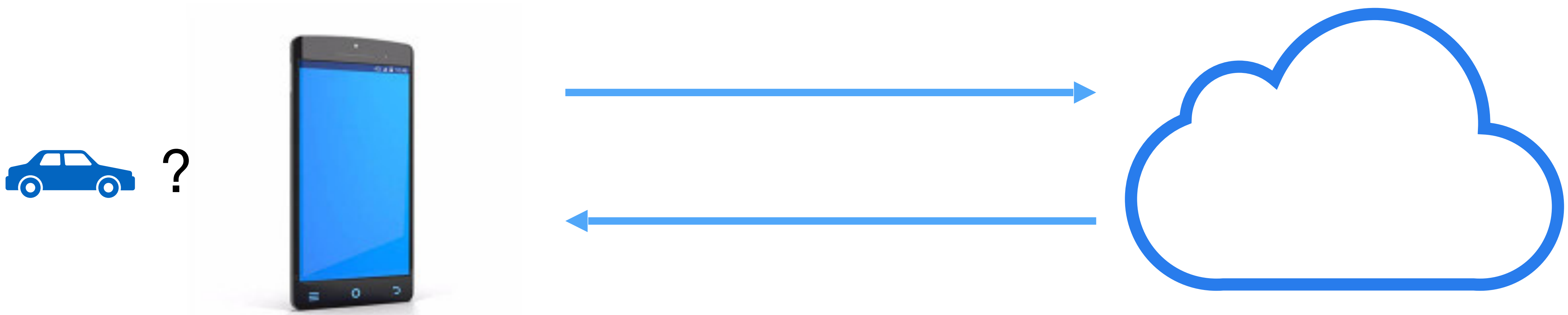
Legend:



Single-keyword SSE: Search

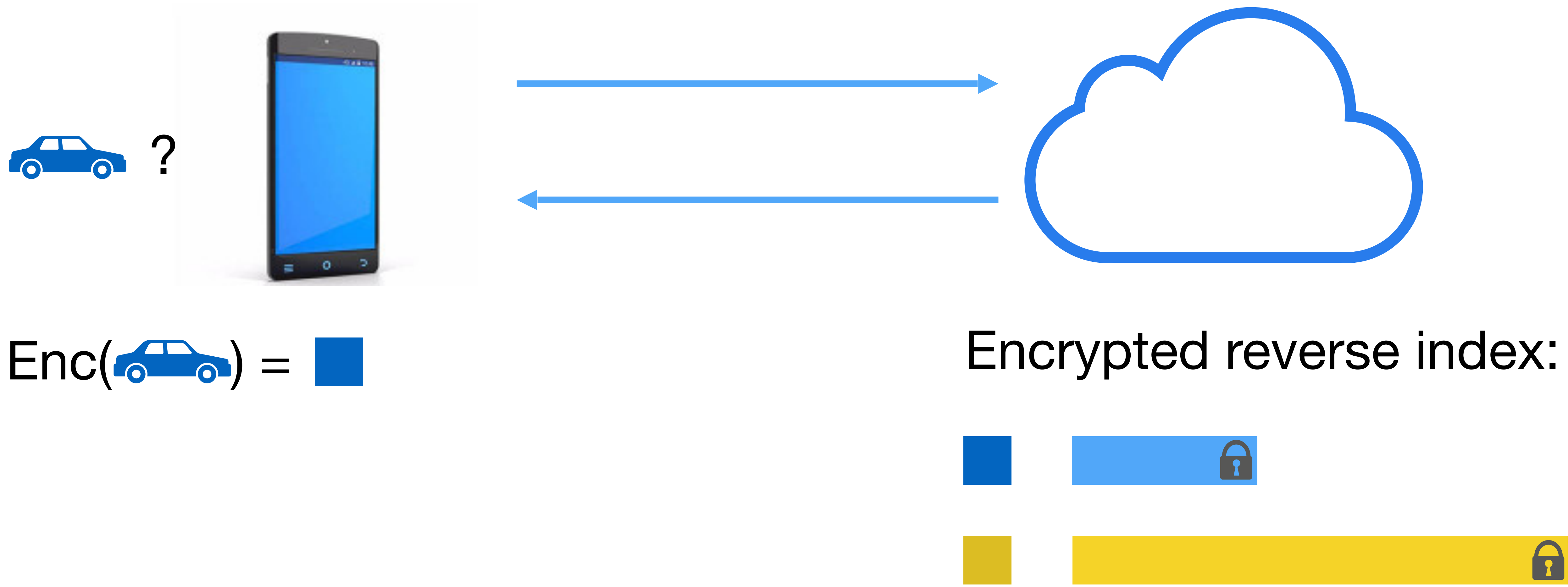


Single-keyword SSE: Search

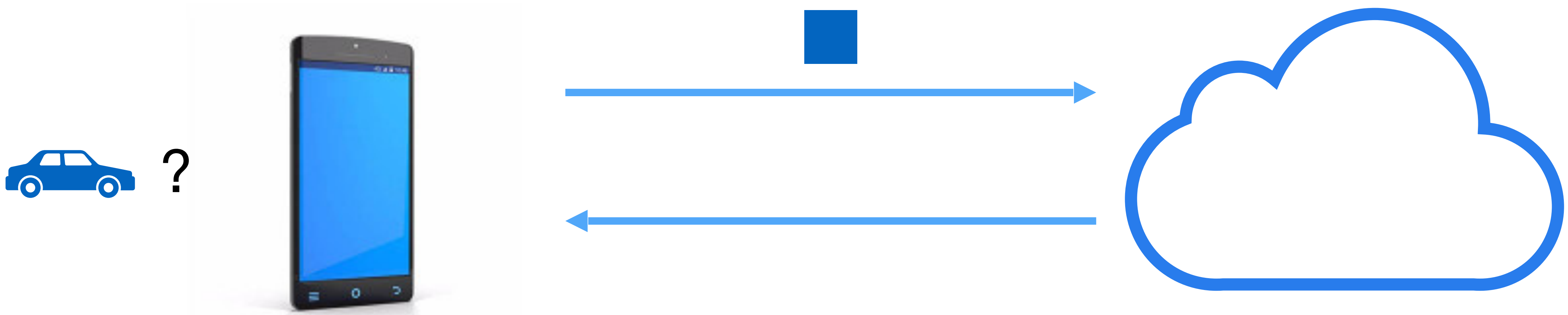


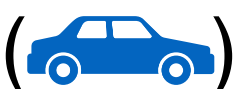
Encrypted reverse index:

Single-keyword SSE: Search



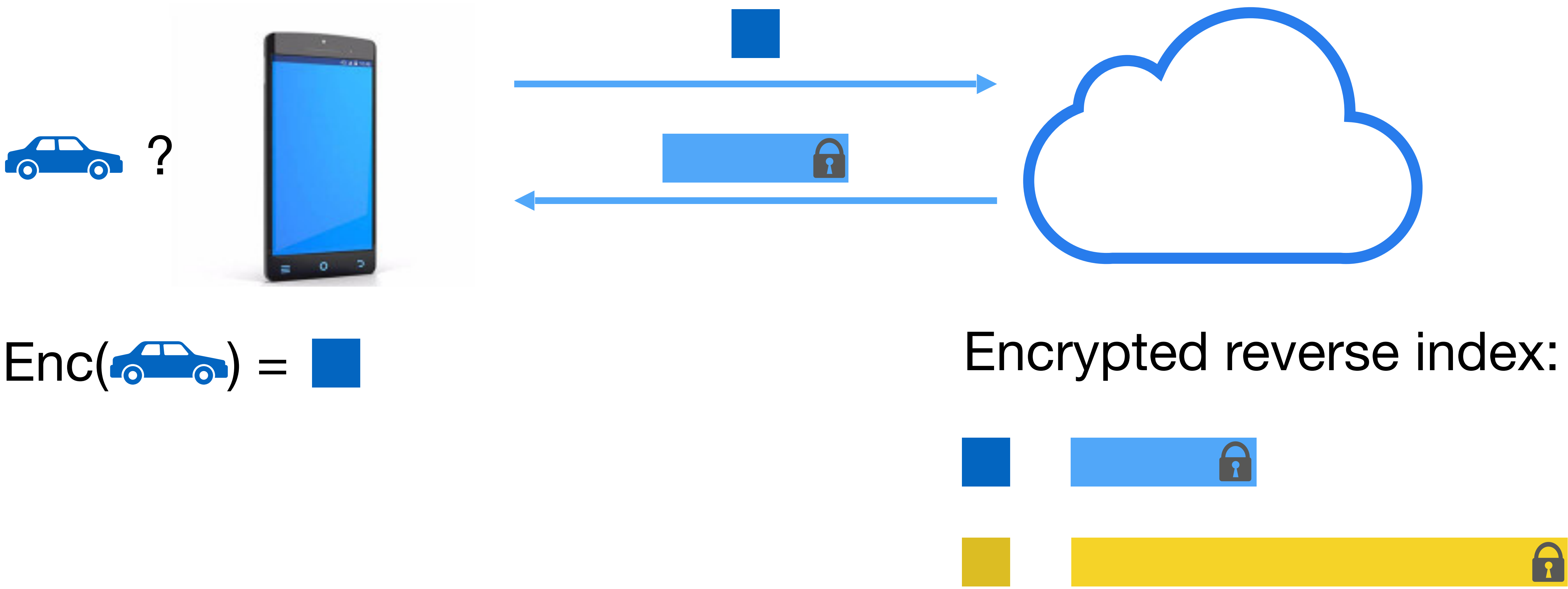
Single-keyword SSE: Search



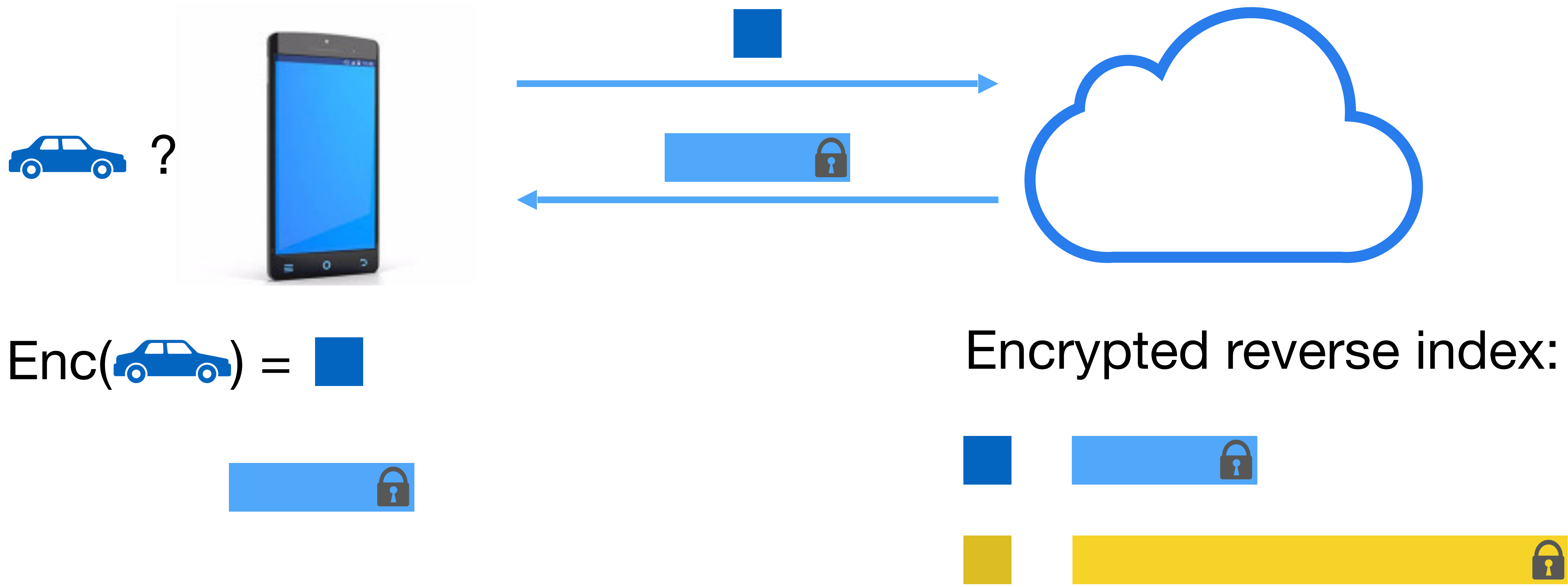
Enc() = 

Encrypted reverse index:

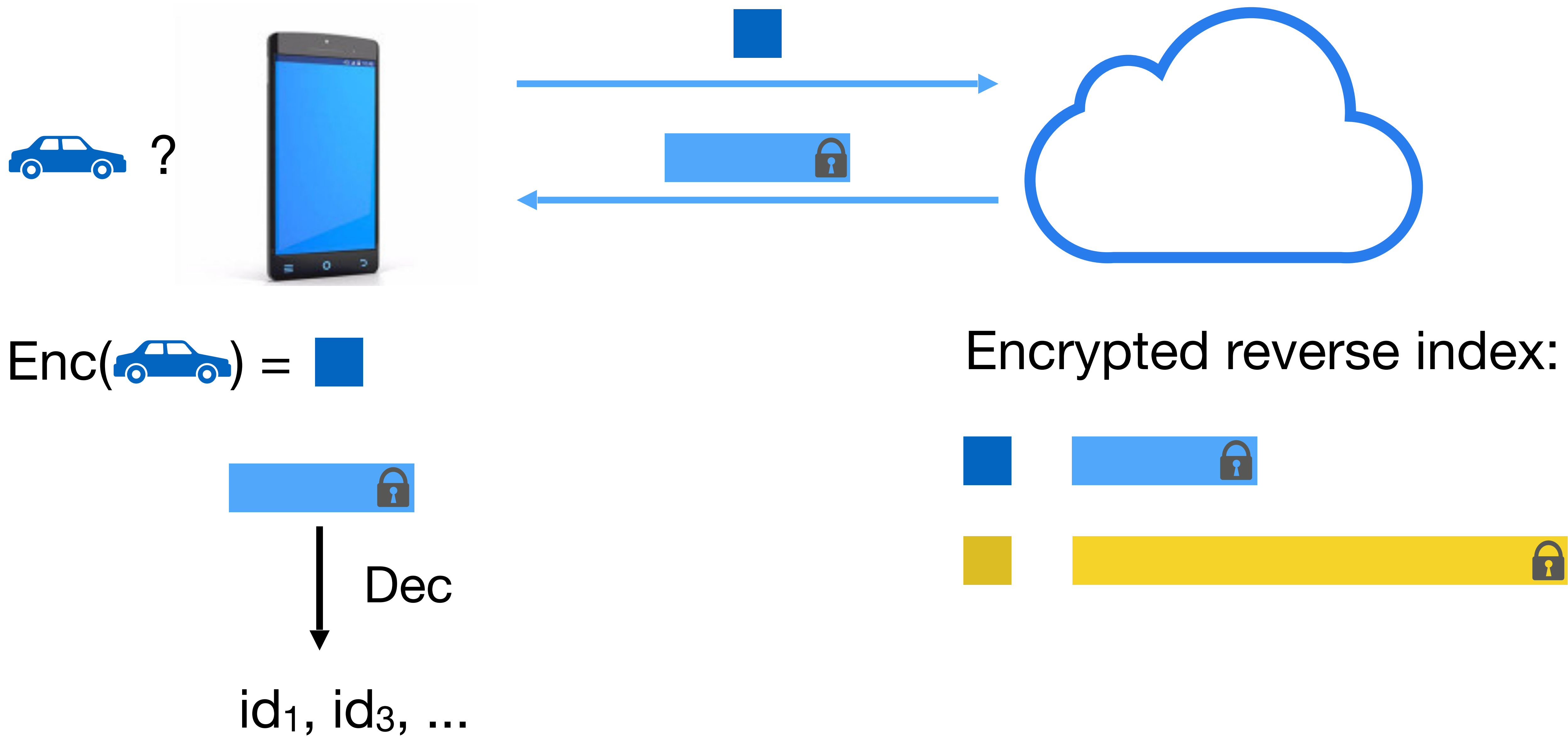
Single-keyword SSE: Search



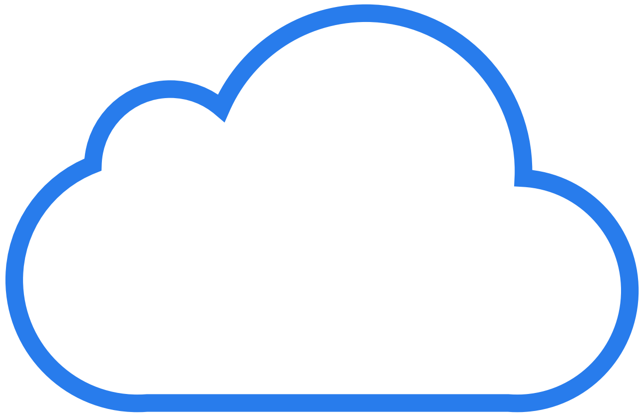
Single-keyword SSE: Search

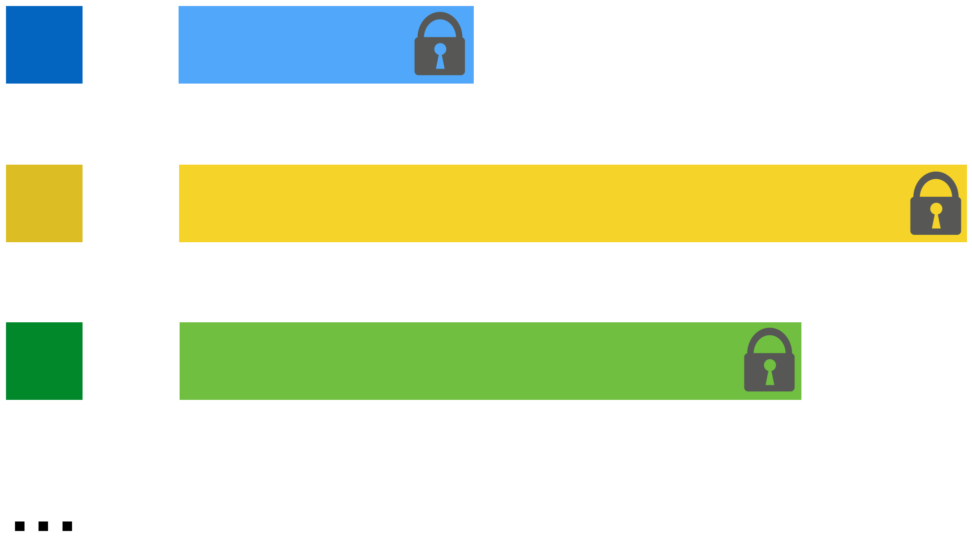


Single-keyword SSE: Search

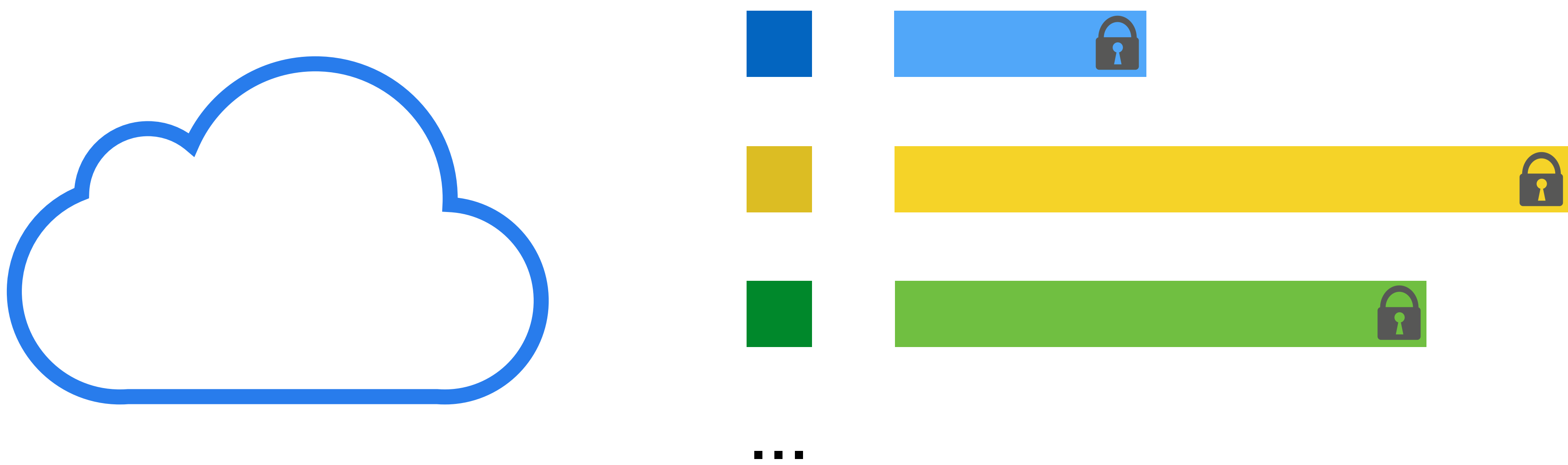


List storage



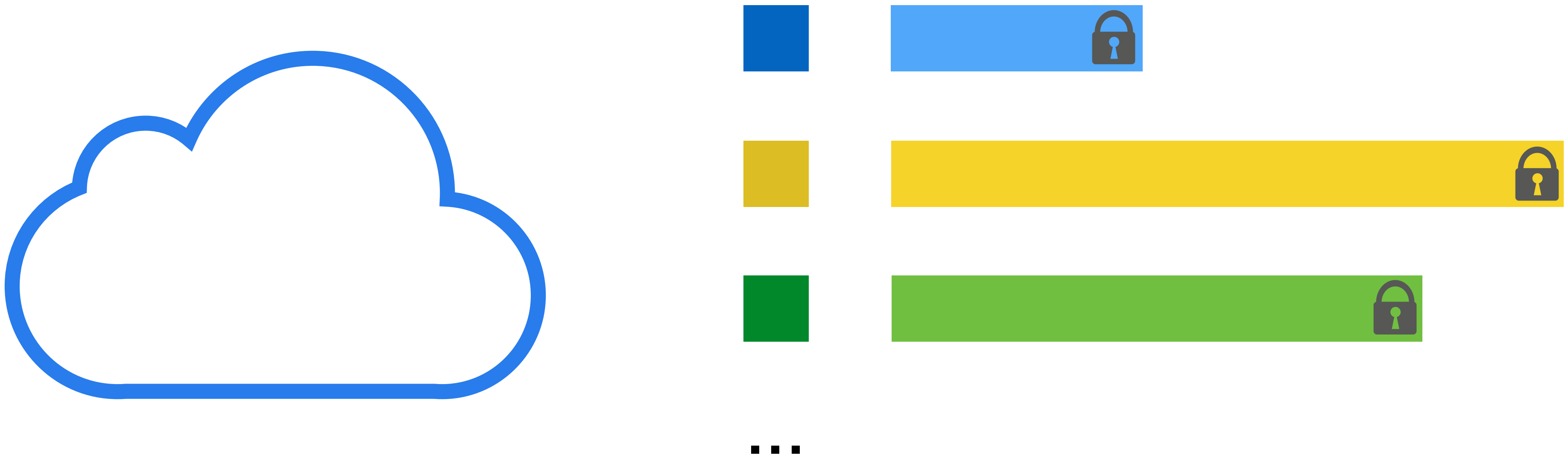


List storage



Naive solutions for list storage:

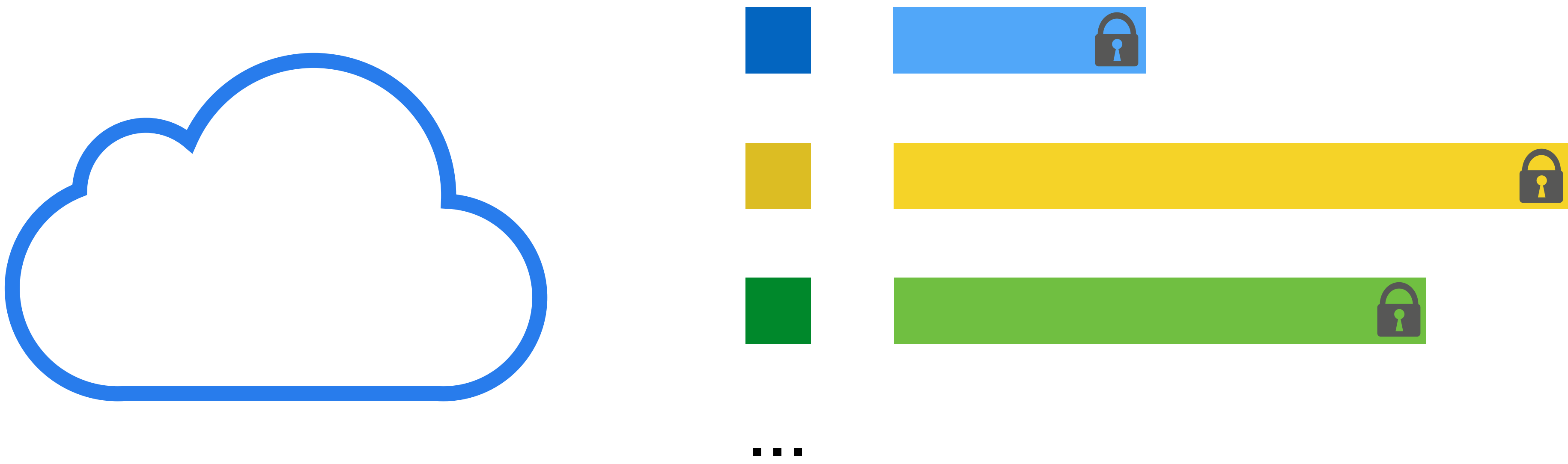
List storage



Naive solutions for list storage:

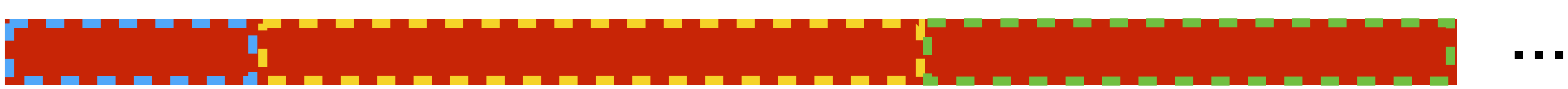
Leaks lengths of all keywords.

List storage

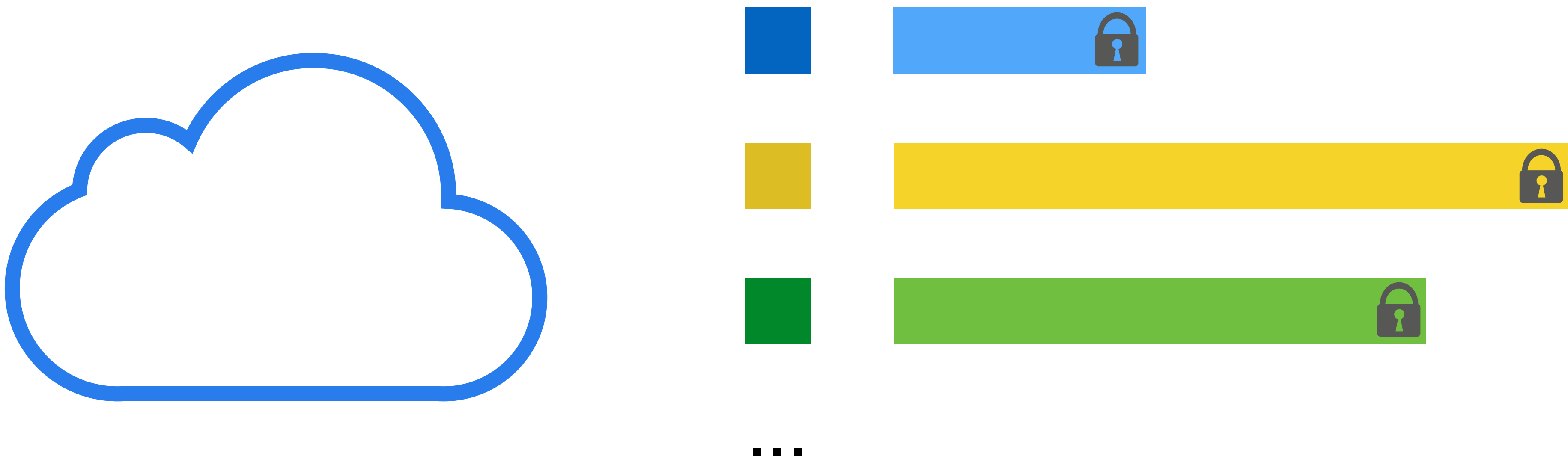


Naive solutions for list storage:

Leaks lengths of all keywords.

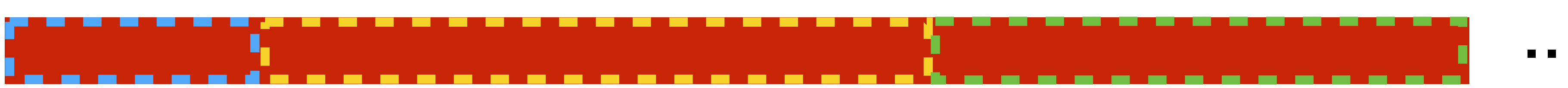


List storage



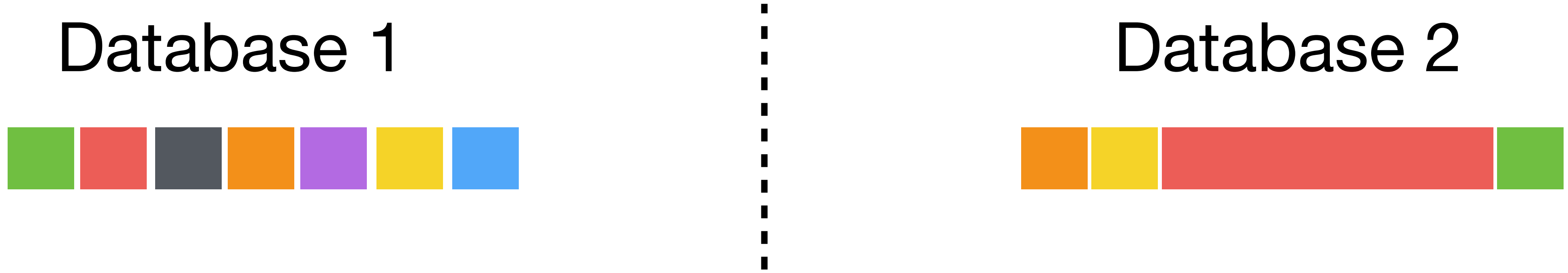
Naive solutions for list storage:

Leaks lengths of all keywords.

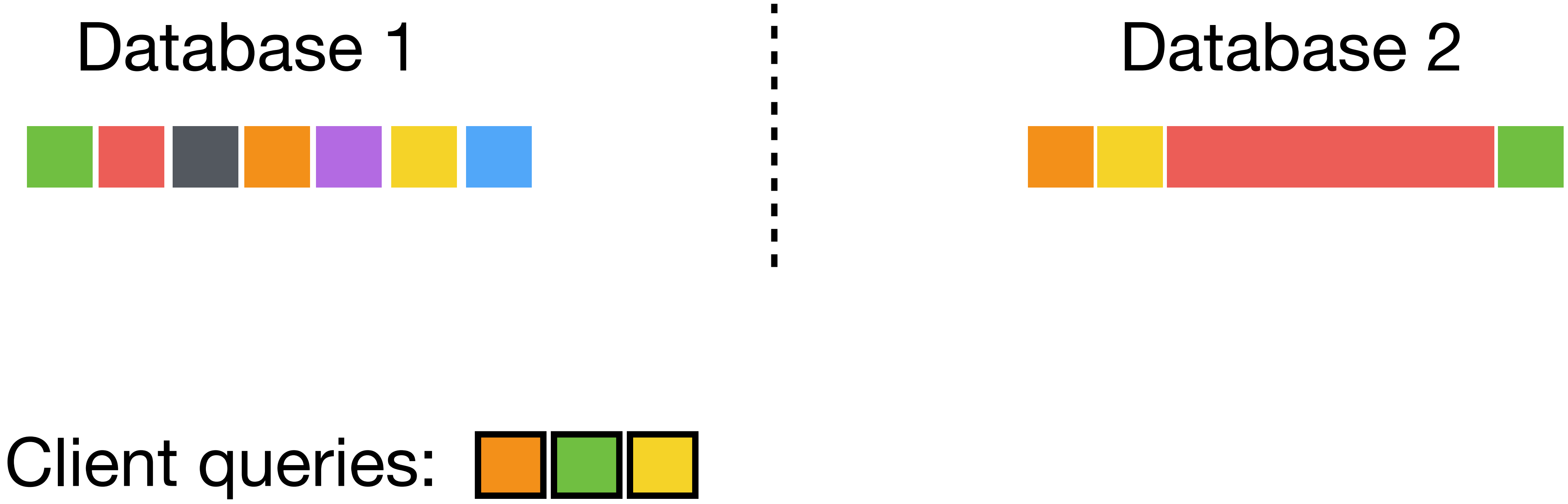


Position of one list depends on lengths of other lists.

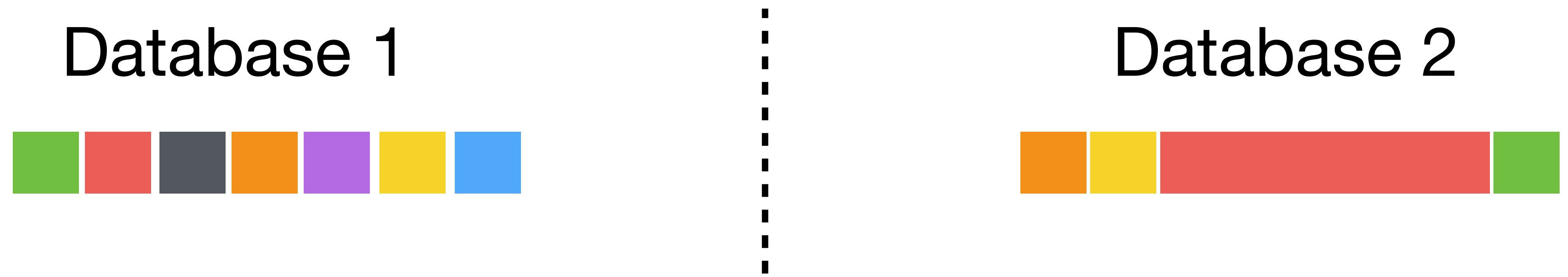
Naive storage



Naive storage

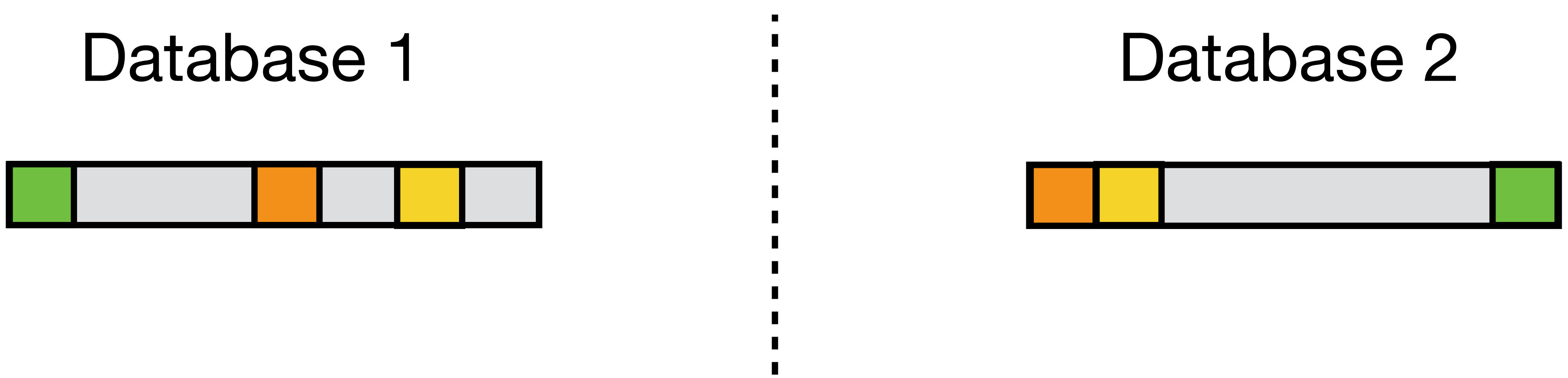


Naive storage

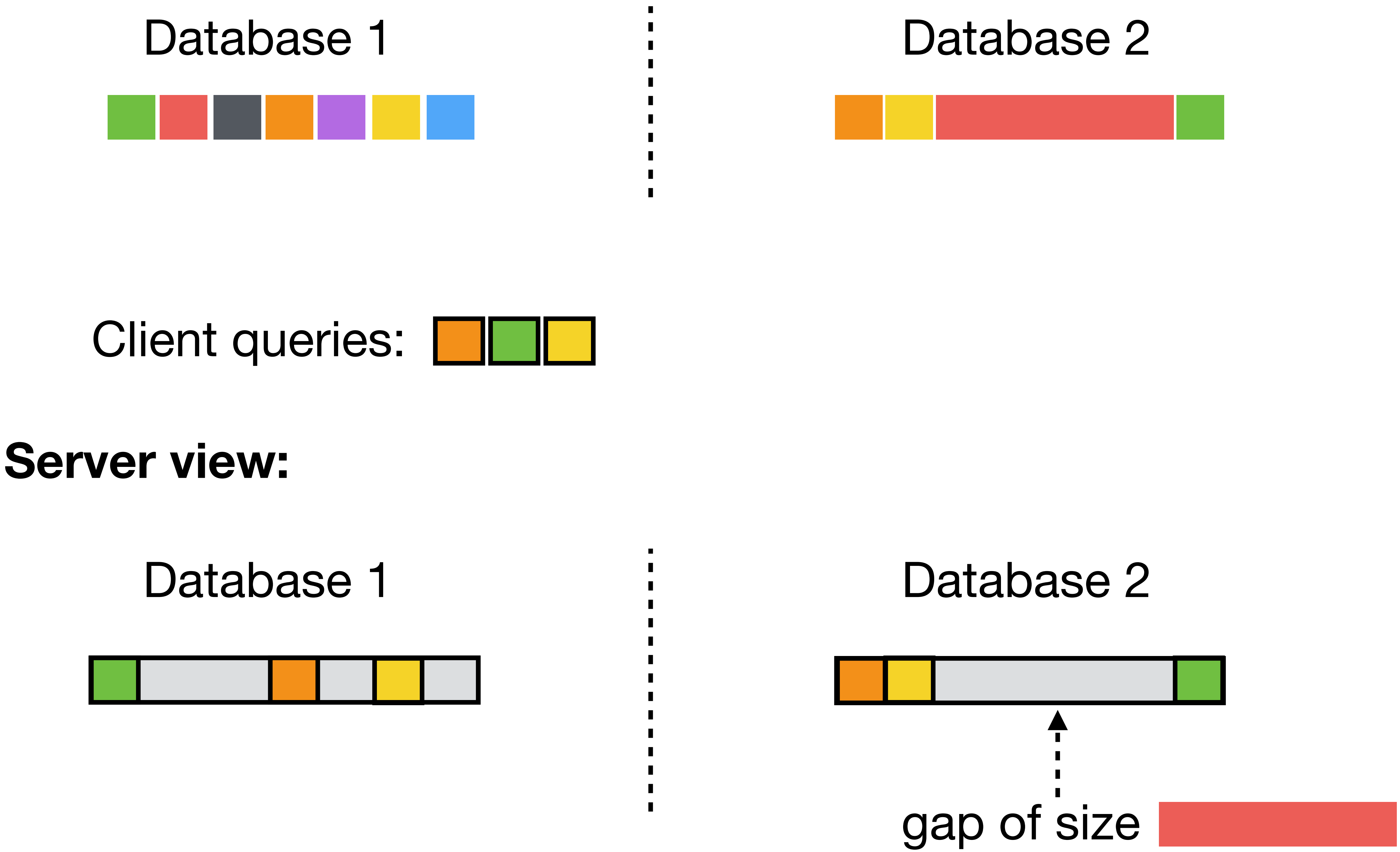


Client queries: 

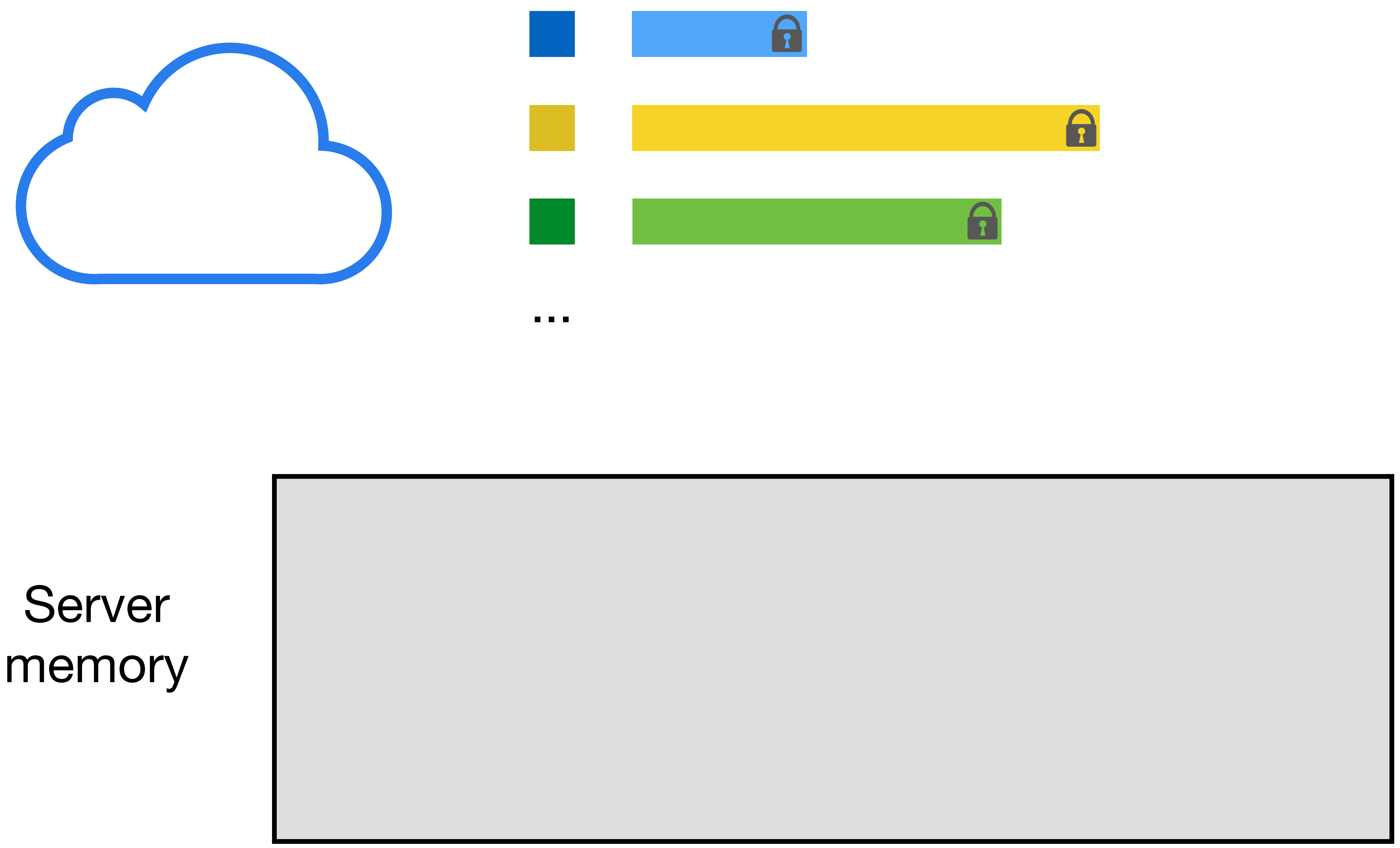
Server view:



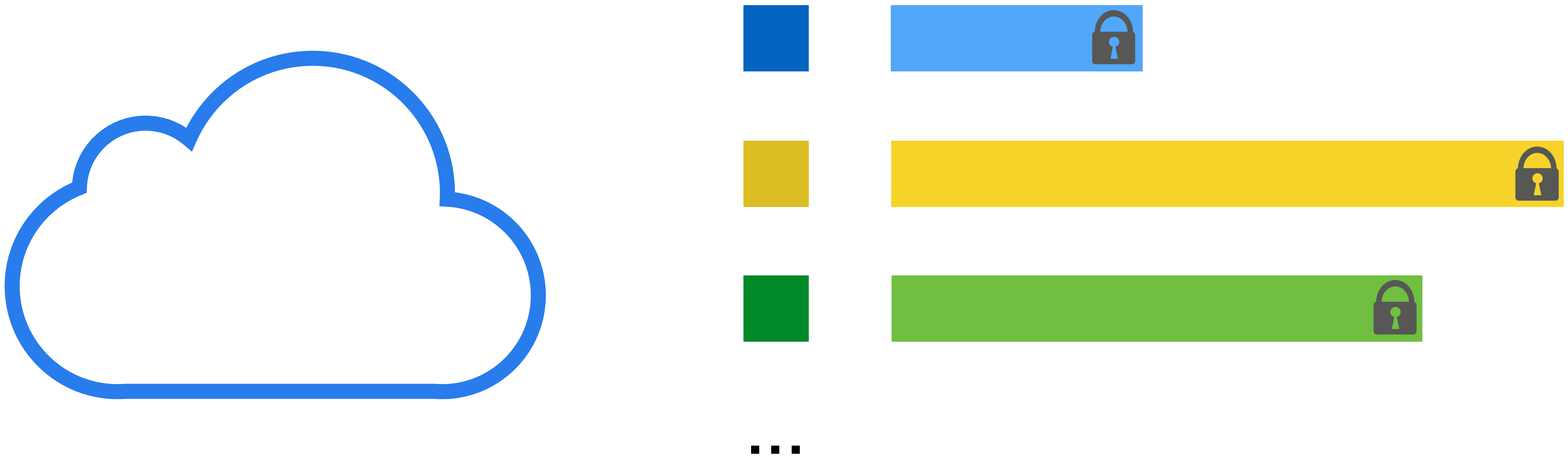
Naive storage



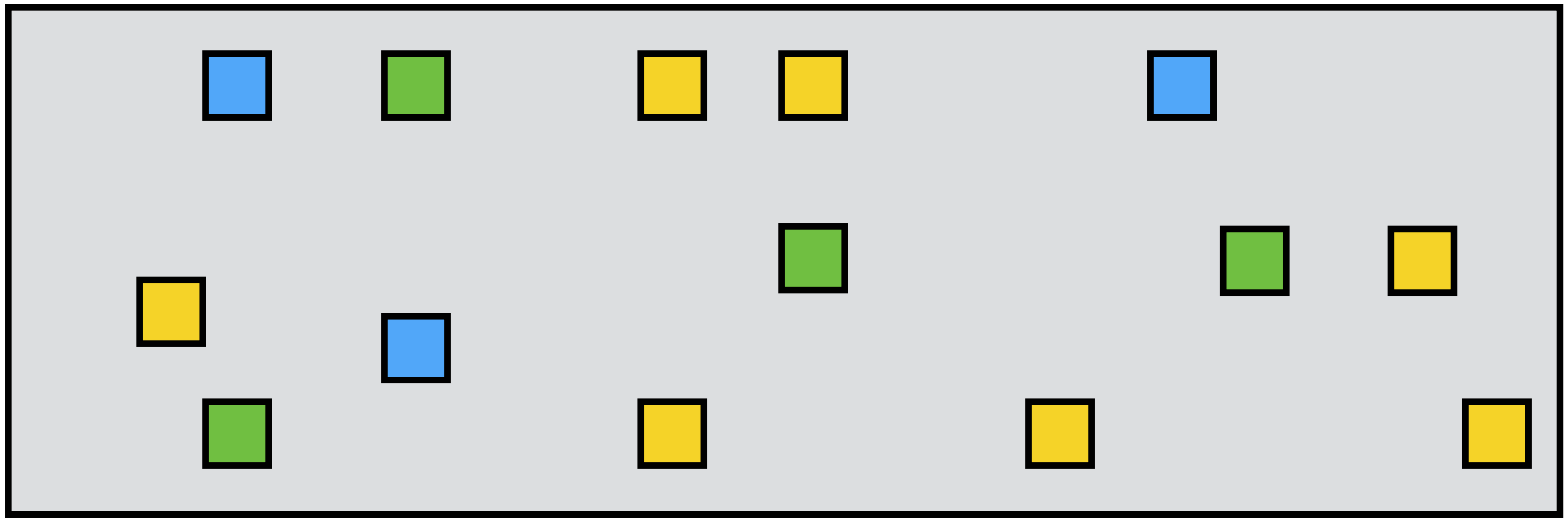
Secure list storage



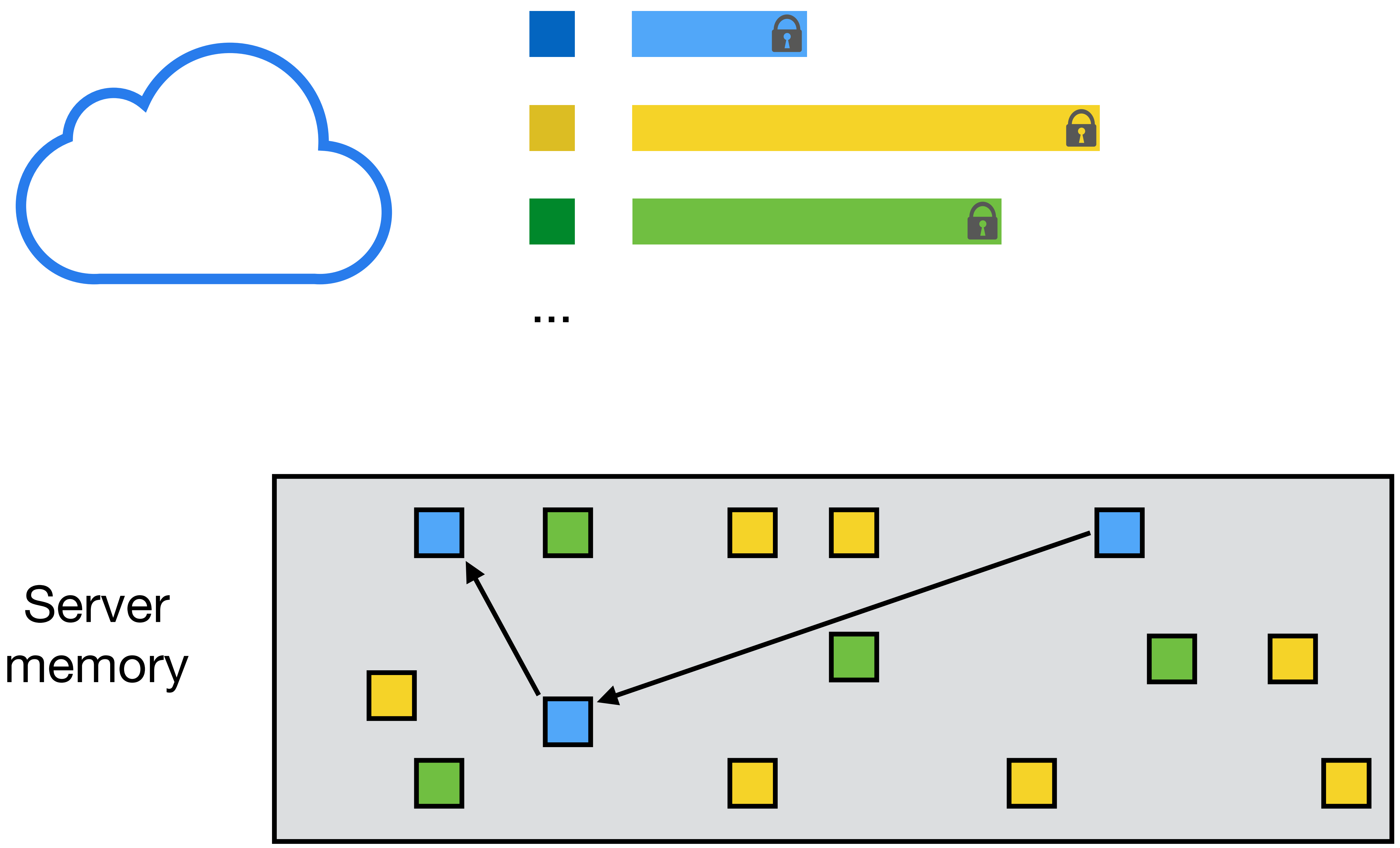
Secure list storage



Server
memory

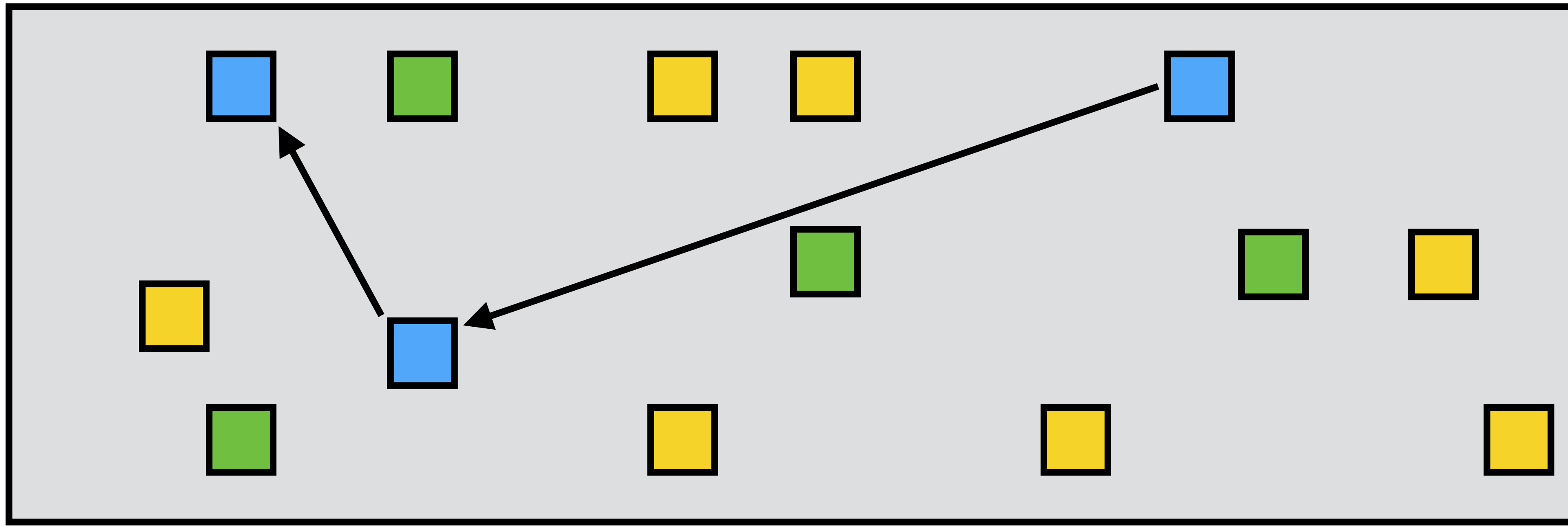


Secure list storage



Secure list storage

Server memory

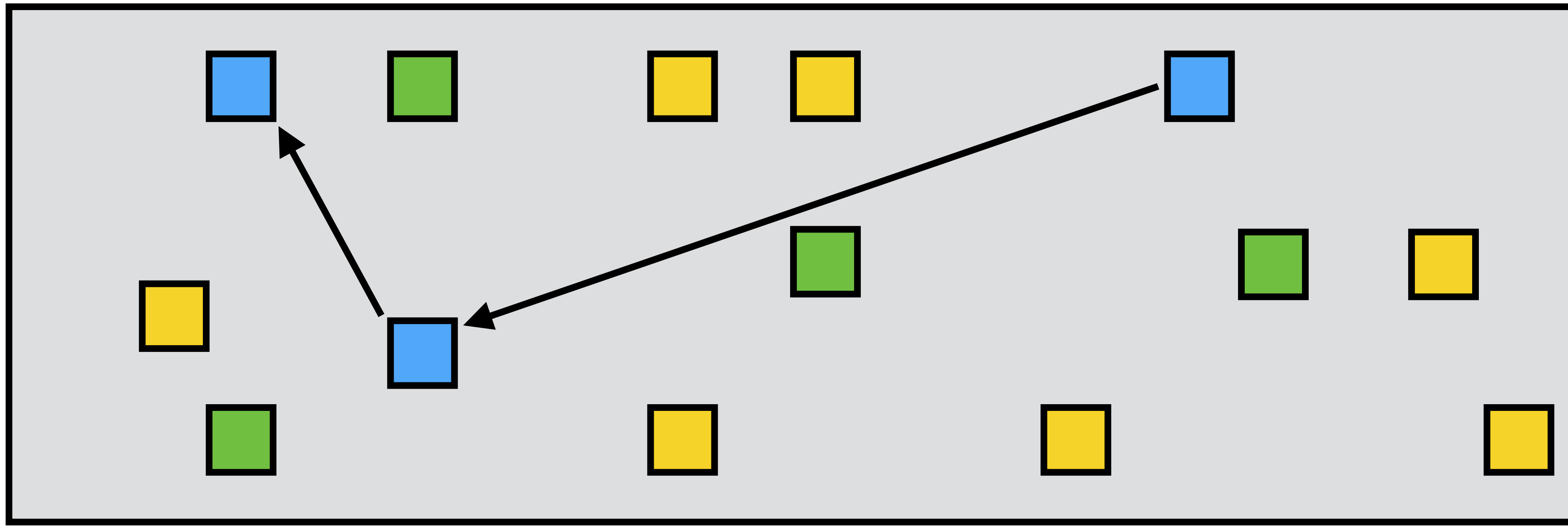


Security: OK.

List of length ℓ = ℓ unif. random memory accesses

Secure list storage

Server memory



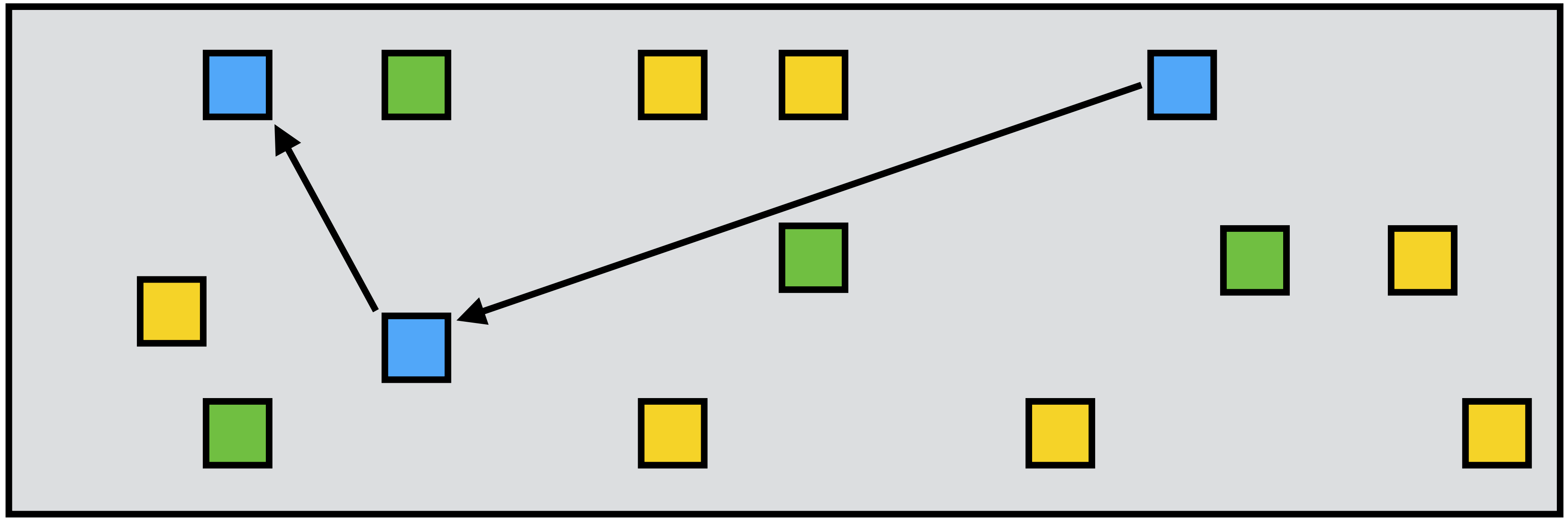
Security: OK.

List of length ℓ = ℓ unif. random memory accesses

Efficiency: Terrible.

Secure list storage

Server
memory



Security: OK.

List of length ℓ = ℓ unif. random memory accesses

Efficiency: Terrible.

Worst-case cost for Hard Disk Drives: reading contiguous memory much cheaper than random locations.

Formalizing the problem

Cash & Tessaro EC ’14

Formalizing the problem

Cash & Tessaro EC '14

Locality: #discontinuous memory accesses to answer a query.

Formalizing the problem

Cash & Tessaro EC ’14

Locality: #discontinuous memory accesses to answer a query.

Read efficiency: #memory words accessed to answer a query / #memory words of plaintext answer.

Formalizing the problem

Cash & Tessaro EC '14

Locality: #discontinuous memory accesses to answer a query.

Read efficiency: #memory words accessed to answer a query /
#memory words of plaintext answer.

Storage efficiency: #memory words to store encrypted DB /
#memory words of plaintext DB.

Formalizing the problem

Cash & Tessaro EC '14

Locality: #discontinuous memory accesses to answer a query.

Read efficiency: #memory words accessed to answer a query / #memory words of plaintext answer.

Storage efficiency: #memory words to store encrypted DB / #memory words of plaintext DB.

Theorem (Cash & Tessaro EC'14):

Secure SSE cannot have O(1) in all 3 measures.

Building local SSE

Asharov et al. STOC '16 N = size of DB

Scheme	Locality	Storage eff.	Read eff.
"One-choice"	O(1)	O(1)	$\tilde{O}(\log N)$
"Two-choice"	O(1)	O(1)	$\tilde{O}(\log \log N)$*
"Pad-and-split"	O(1)	O(log N)	O(1)

*under condition: longest list size $\leq N^{1-1/\log \log N}$

Demertzis et al. Crypto '18

Scheme	Locality	Storage eff.	Read eff.
Untitled	O(1)	O(1)	$O(\log^{2/3+\varepsilon} N)$

...

Page efficiency

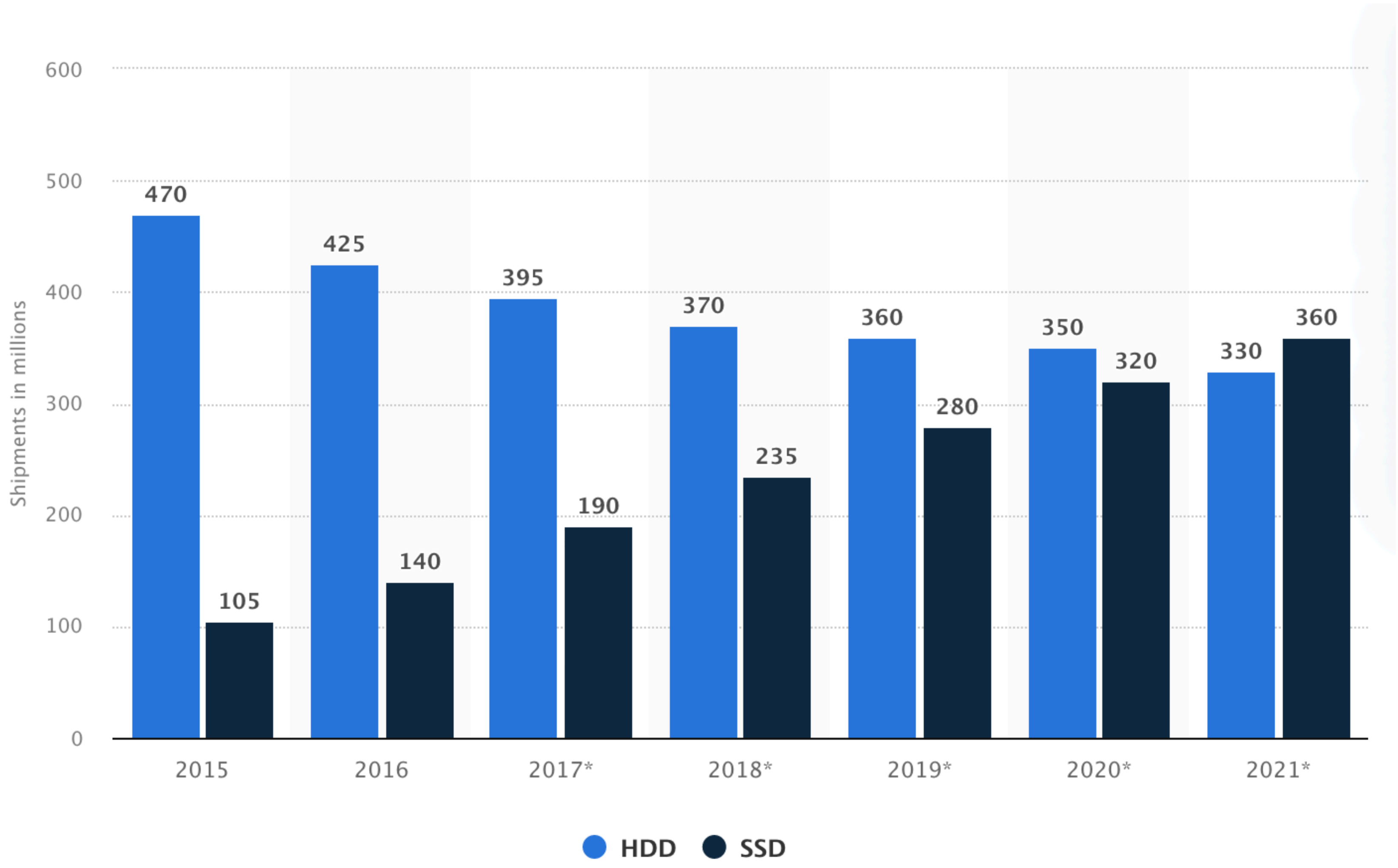
HDD vs SSD

Two most prevalent media for storage:

Hard Disk Drive
(HDD)

Solid State Drive
(SSD, “Flash”)

HDD vs SSD



SSDs Outsell HDDs in Unit Sales 3:2: 99 Million Vs. 64 Million in Q1

By Anton Shilov May 21, 2021

But HDDs maintain exabytes lead: 288.3EB vs 61.5EB.

https://www.statista.com/statistics/285474/hdds-and-ssds-in-pcs-global-shipments-2012-2017/
https://www.tomshardware.com/news/ssd-market-shares-q1-2021-trendfocus

Performance criterions

Performance criterions

HDD: locality (+ read efficiency).

Performance criterions

HDD: locality (+ read efficiency).

SSD: locality does not matter...

Performance criterions

HDD: locality (+ read efficiency).

SSD: locality does not matter...

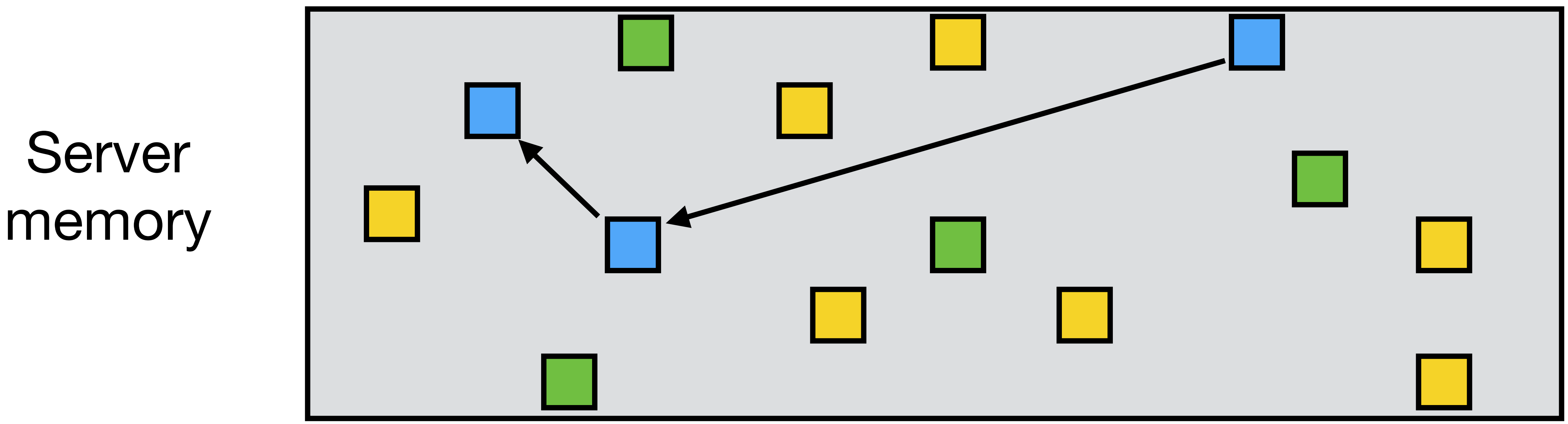
What matters: number of memory pages read.

Performance criterions

HDD: locality (+ read efficiency).

SSD: locality does not matter...

What matters: number of memory pages read.



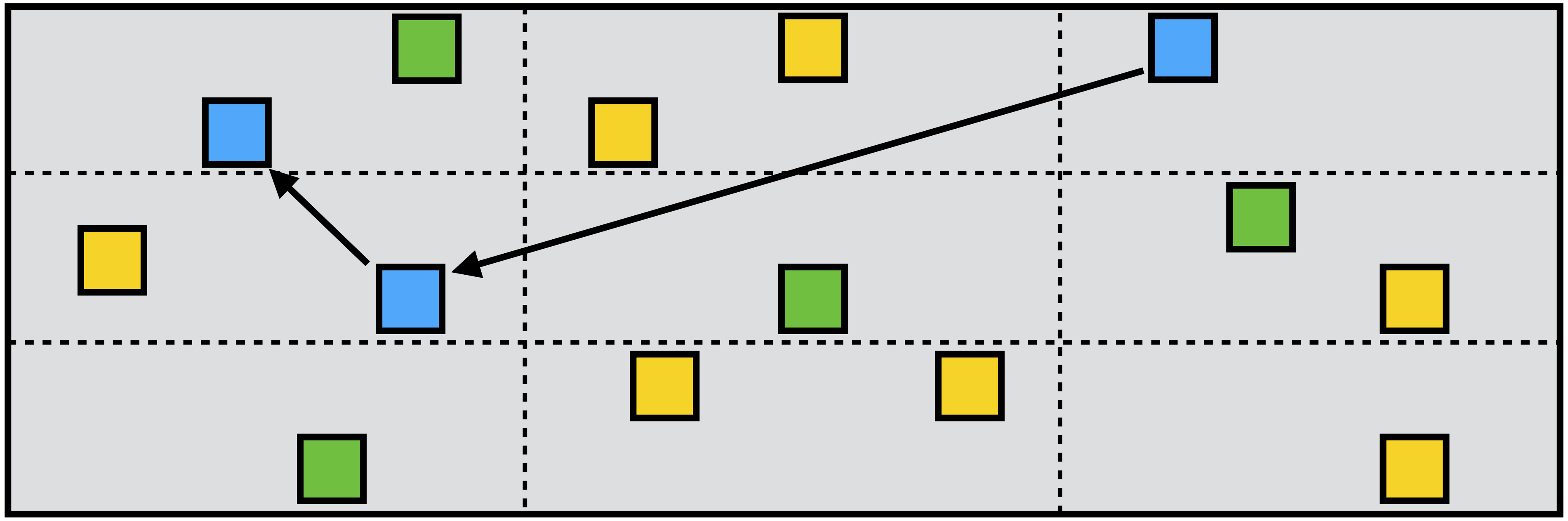
Performance criterions

HDD: locality (+ read efficiency).

SSD: locality does not matter...

What matters: number of memory pages read.

Server memory

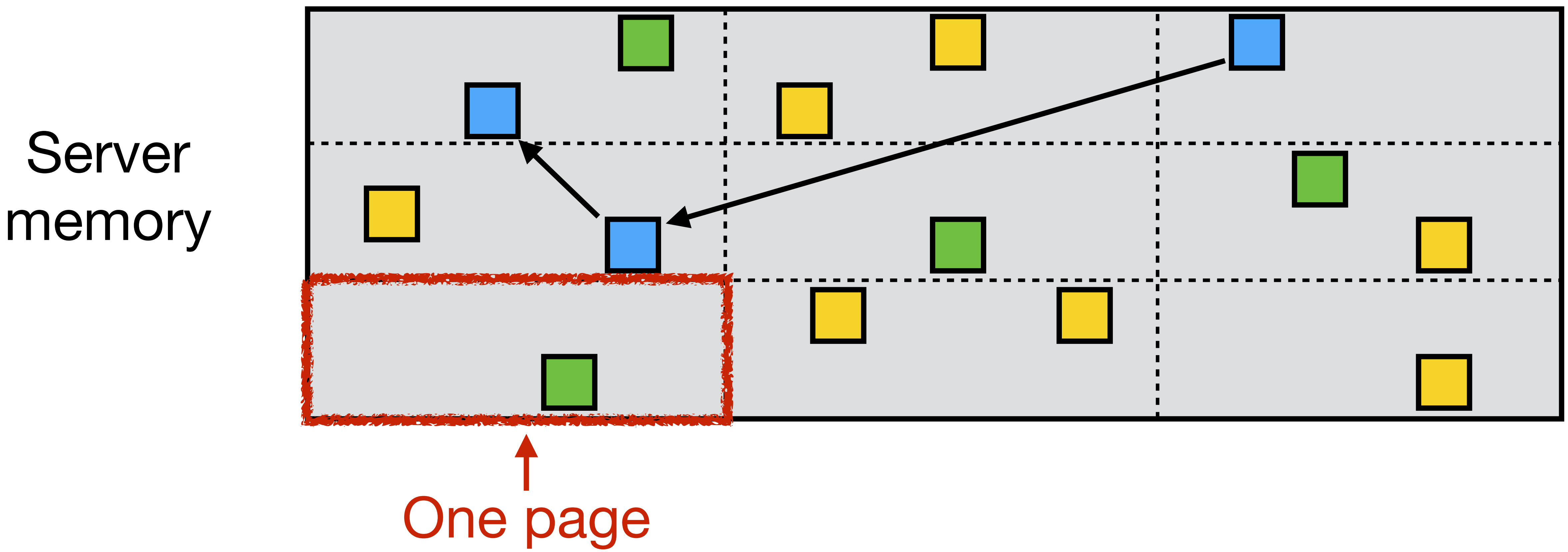


Performance criterions

HDD: locality (+ read efficiency).

SSD: locality does not matter...

What matters: number of memory pages read.



Page efficiency

HDD: Locality + Read efficiency + Storage efficiency.

SSD:

Page efficiency: #memory pages accessed to answer a query / #memory pages of plaintext answer.

Storage efficiency: #memory words to store encrypted DB / #memory words of plaintext DB.

Page efficiency

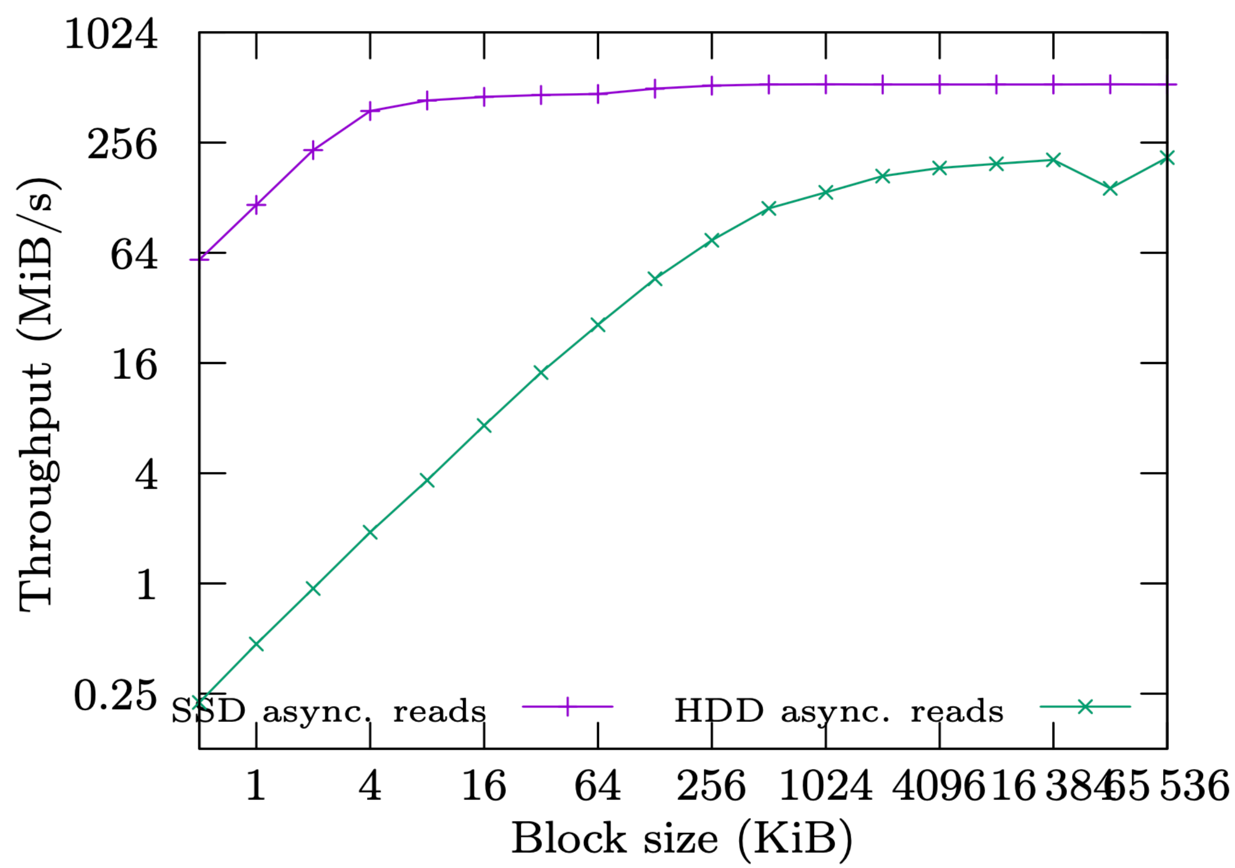
HDD: Locality + Read efficiency + Storage efficiency.

SSD:

Page efficiency: #memory pages accessed to answer a query / #memory pages of plaintext answer.

Storage efficiency: #memory words to store encrypted DB / #memory words of plaintext DB.

Throughput of asynchronous reads, function of the block size



Page efficiency

HDD: Locality + Read efficiency + Storage efficiency.

Theorem (Cash & Tessaro EC '14):

Secure SSE cannot have O(1) in all 3 measures.

Page efficiency

HDD: Locality + Read efficiency + Storage efficiency.

Theorem (Cash & Tessaro EC '14):

Secure SSE cannot have O(1) in all 3 measures.

SSD: Page efficiency + Storage efficiency.

Can we get O(1) in both measures?

Page efficiency

HDD: **Locality** **+** **Read efficiency** **+** **Storage efficiency.**

> Theorem (Cash & Tessaro EC '14):
>
> *Secure* SSE cannot have O(1) in all 3 measures.

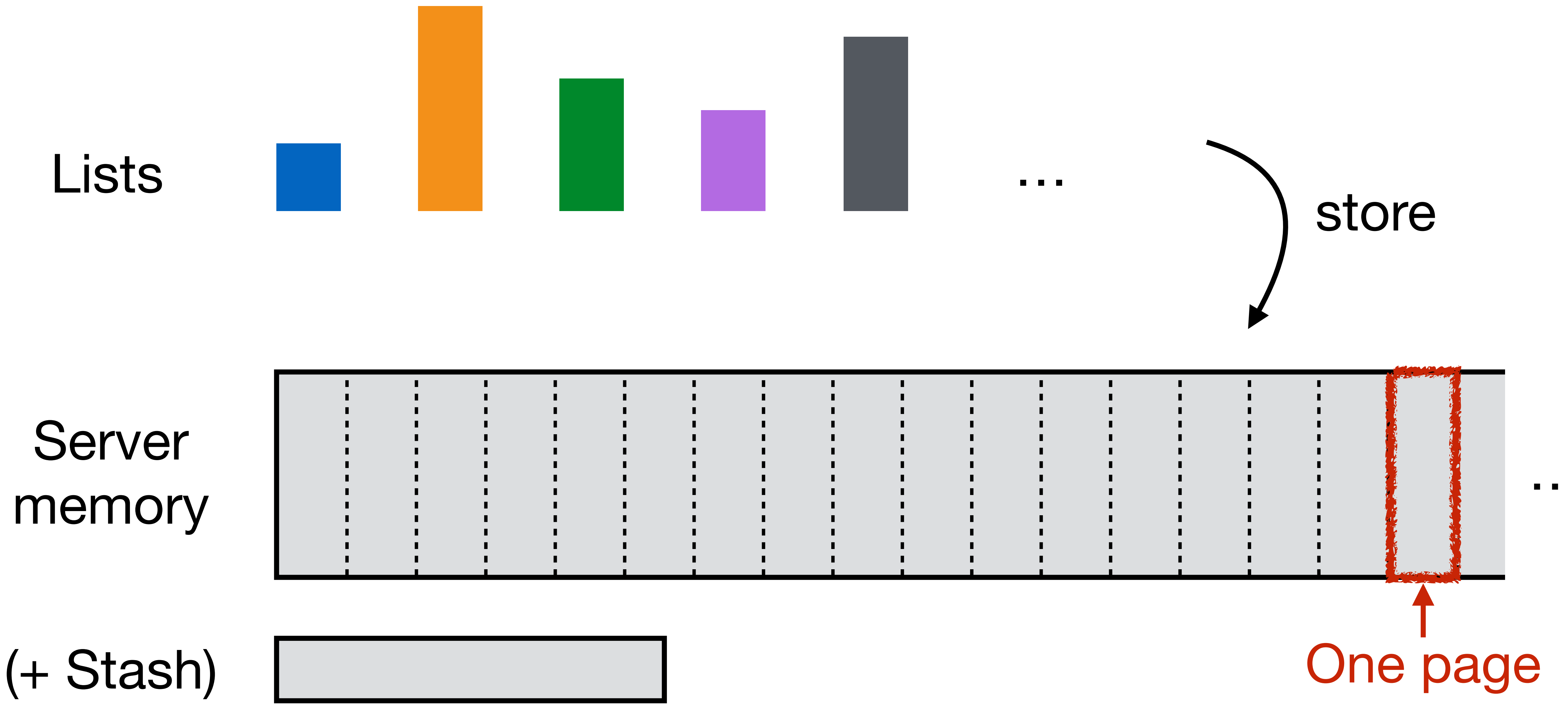
SSD: **Page efficiency** **+** **Storage efficiency.**

Can we get O(1) in both measures?

(Yes.)

Page-efficient allocation

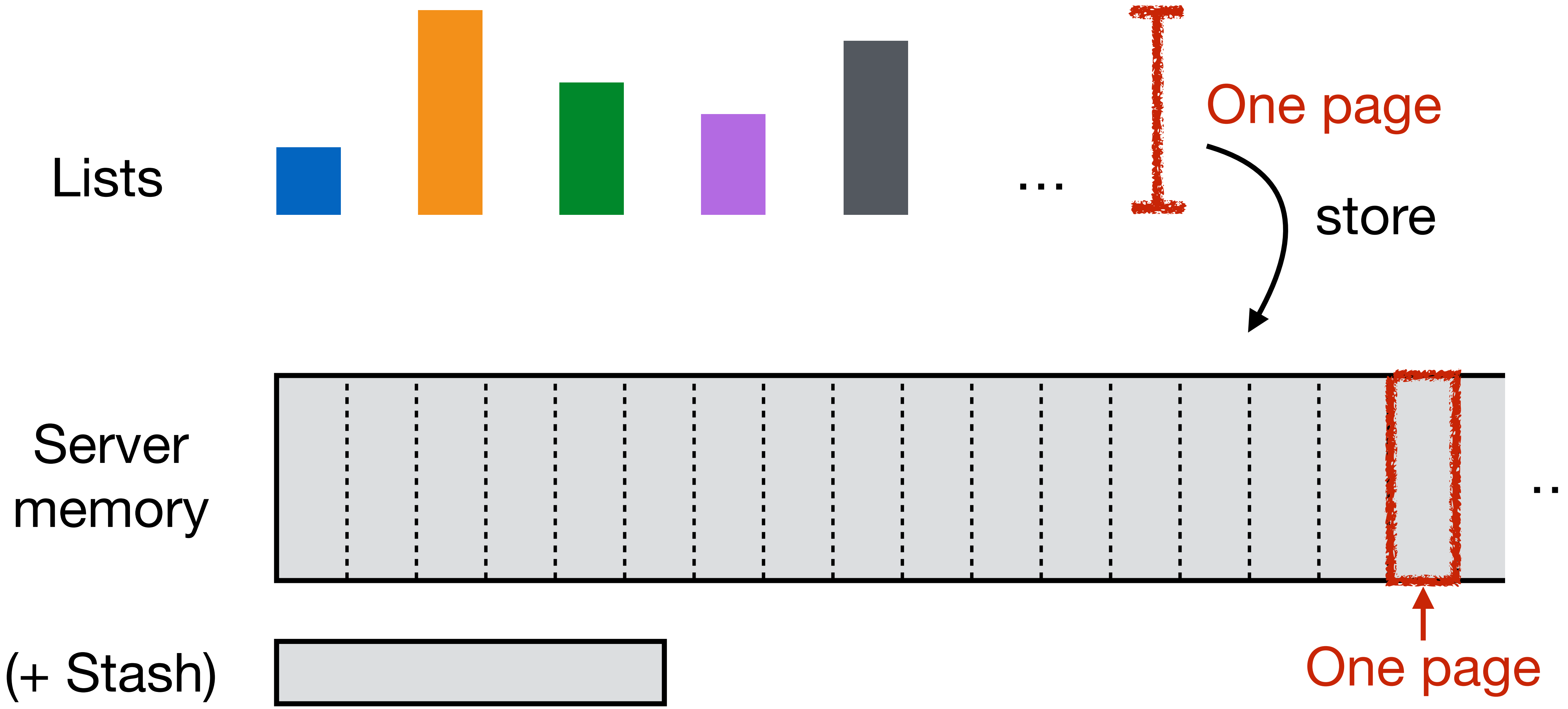
Problem recap



Problem recap

WLOG all lists are of size at most one page:

Problem recap



WLOG all lists are of size at most one page:

Problem recap

Goal:

Page efficiency: #pages accessed to get one list = O(1).

Storage efficiency: #pages to store encrypted DB = O(n).

n = Σ list sizes / p

Data-Independent Packing

Goal:

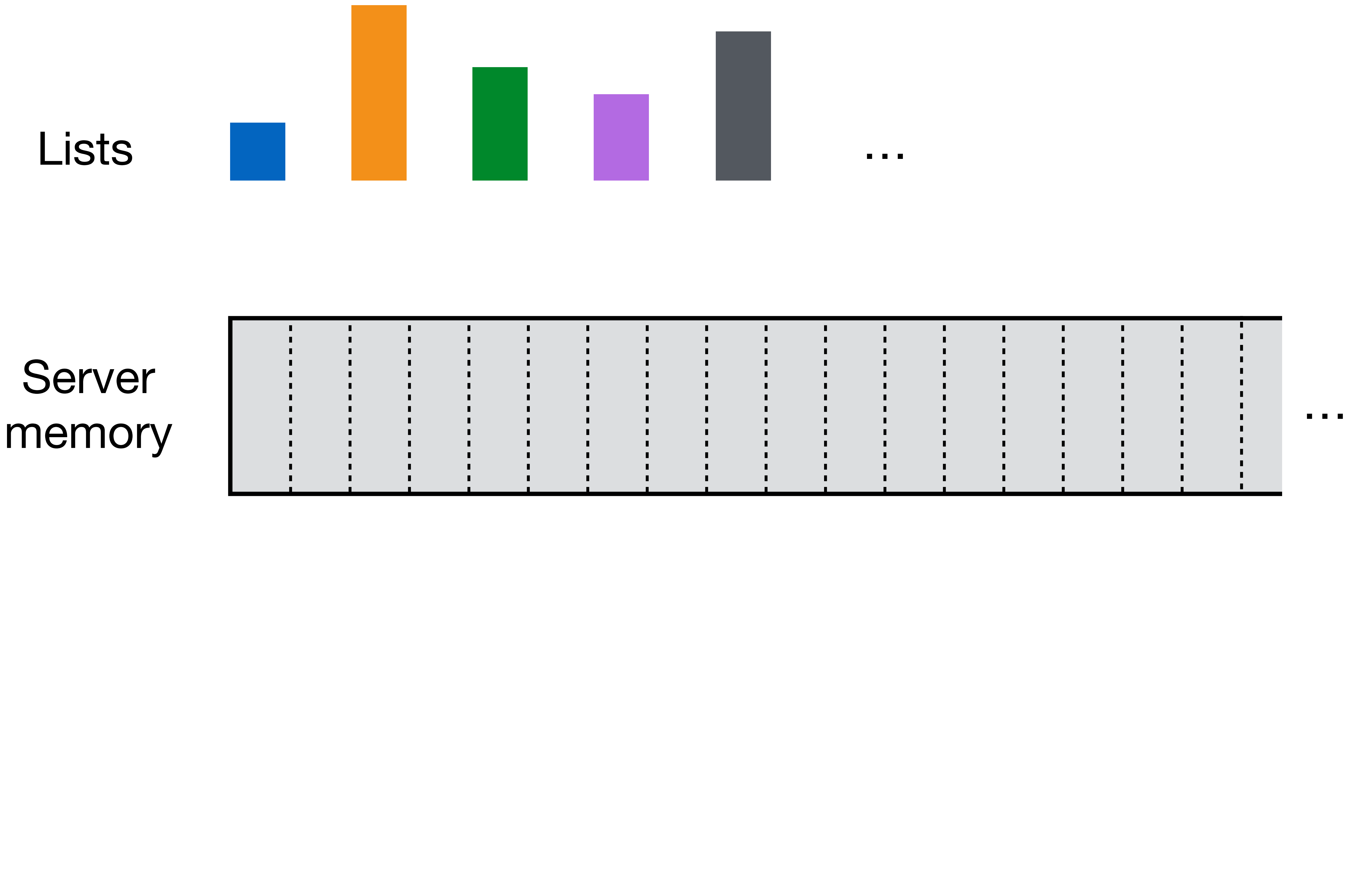
Page efficiency: #pages accessed to get one list = O(1).

Storage efficiency: #pages to store encrypted DB = O(n).

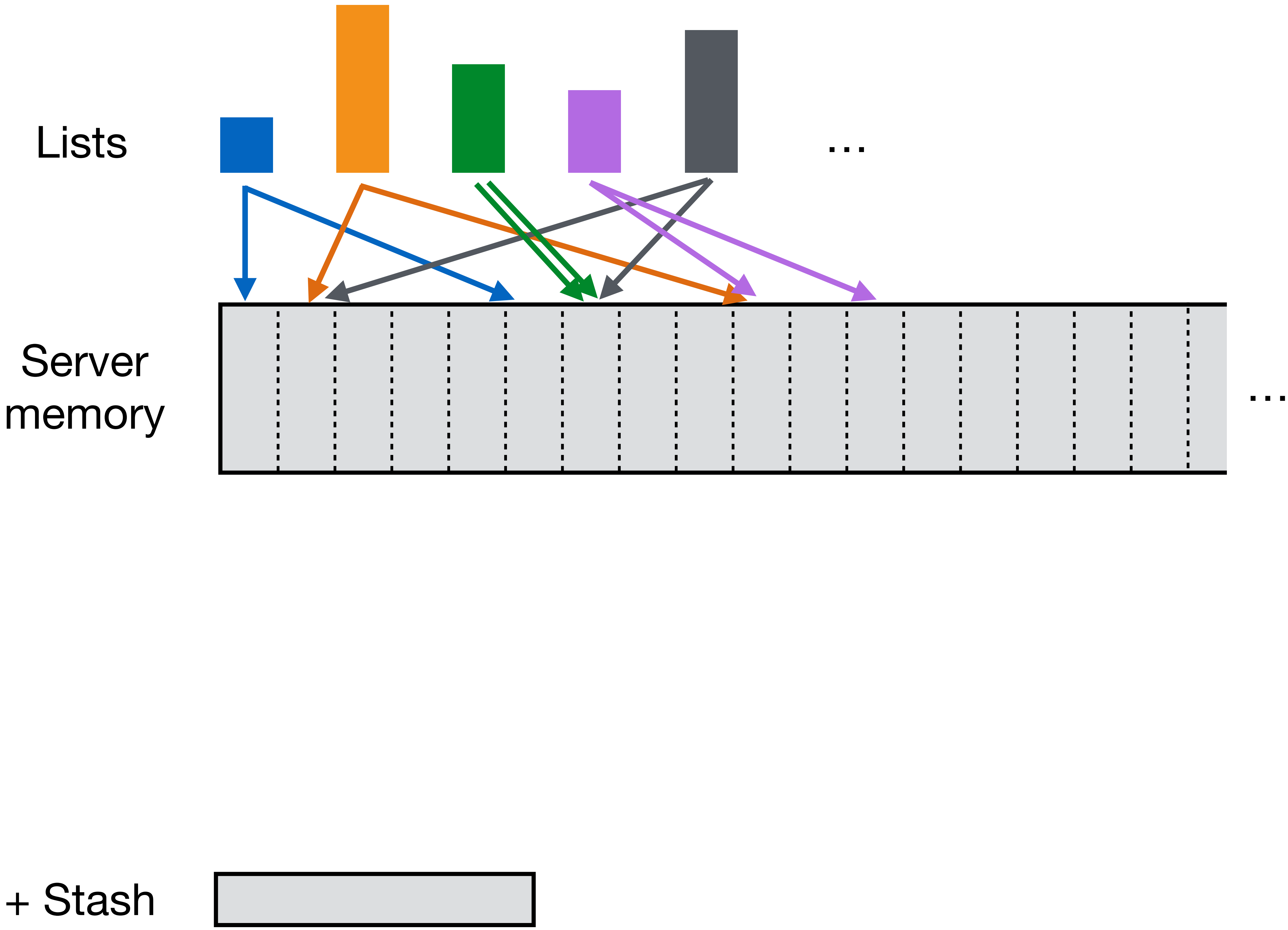
Security: pages accessed to get list ID = f_n(ID).

Does not depend on rest of DB.

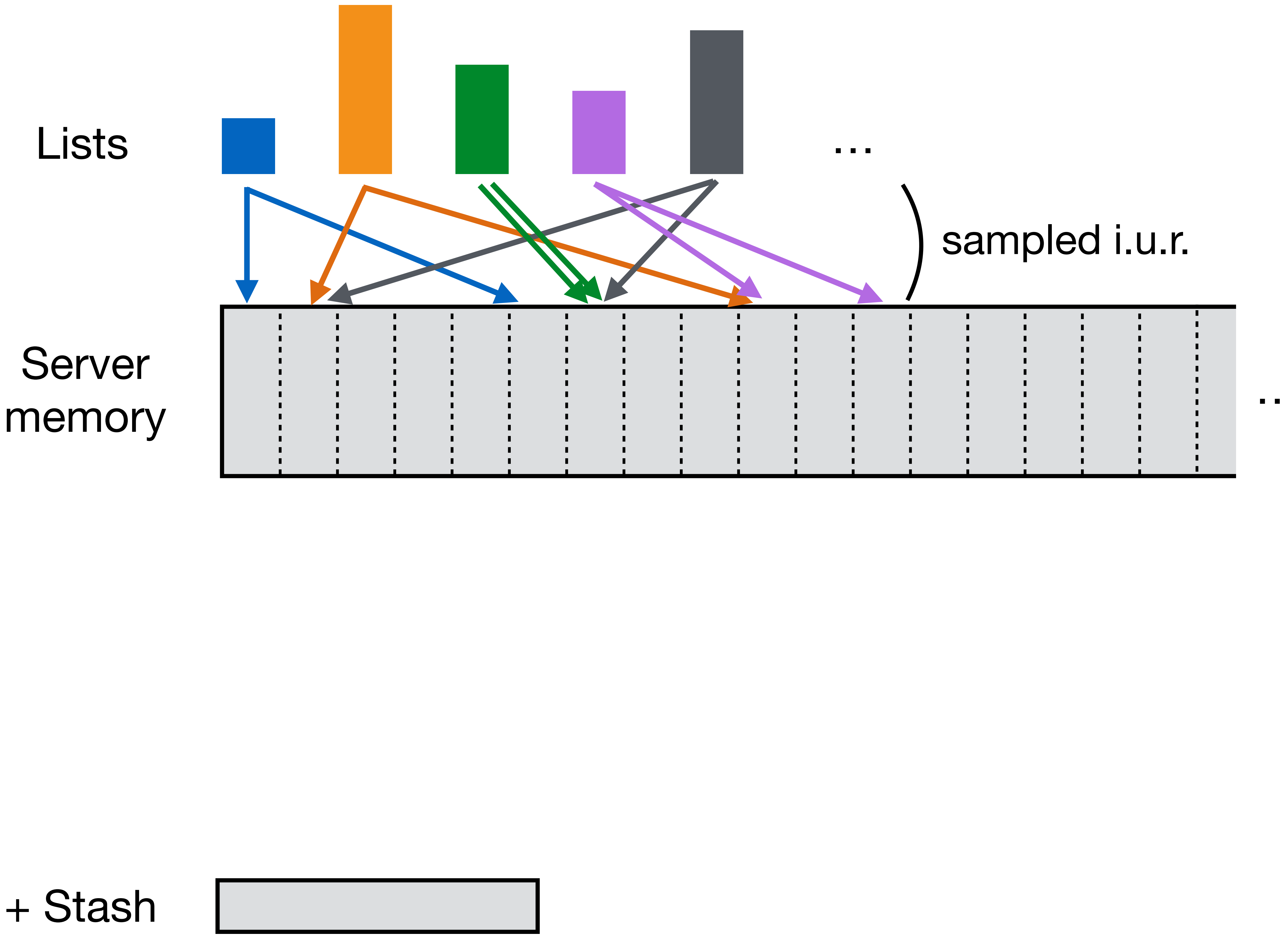
Tethys allocation



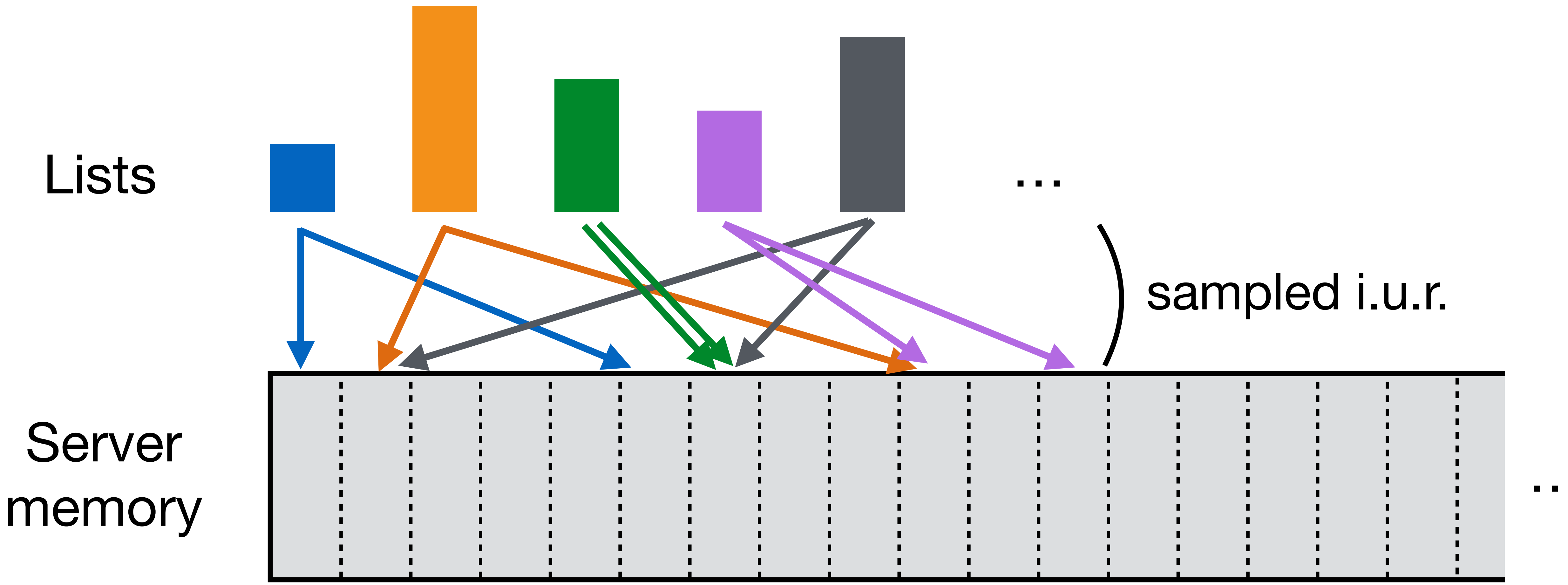
Tethys allocation



Tethys allocation

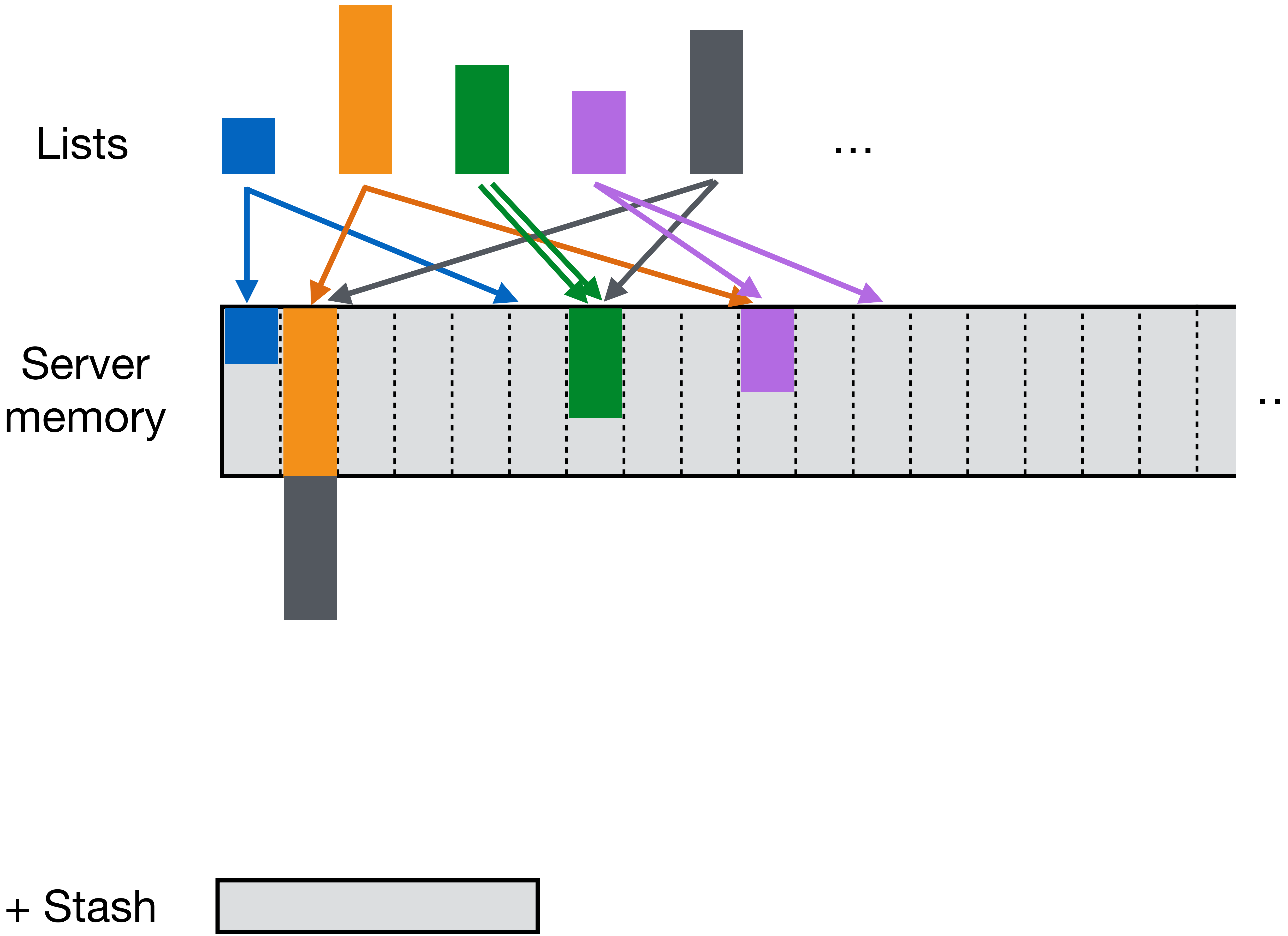


Tethys allocation

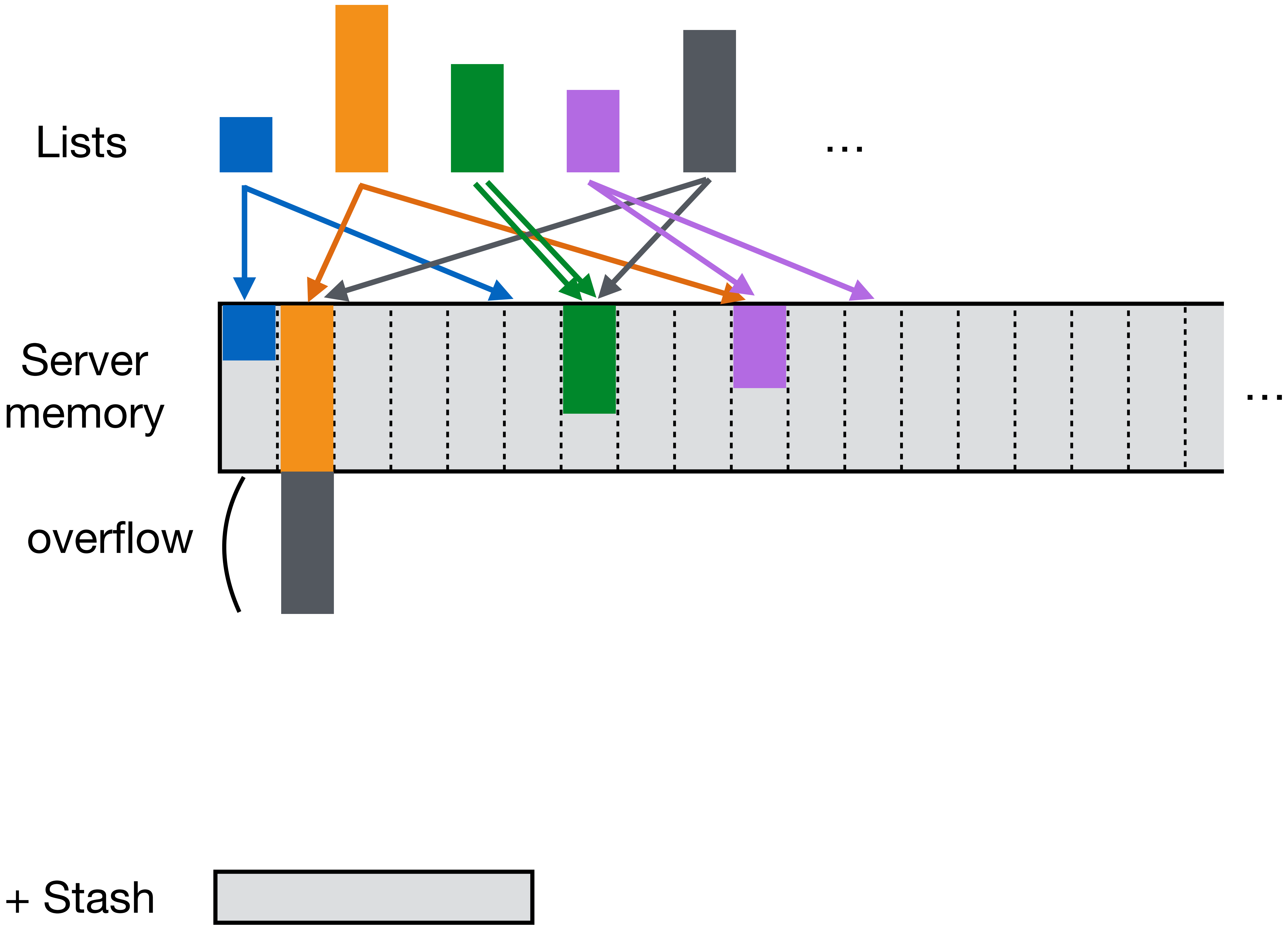


Invariant:
Every list ⊆ its two associated buckets + the stash.

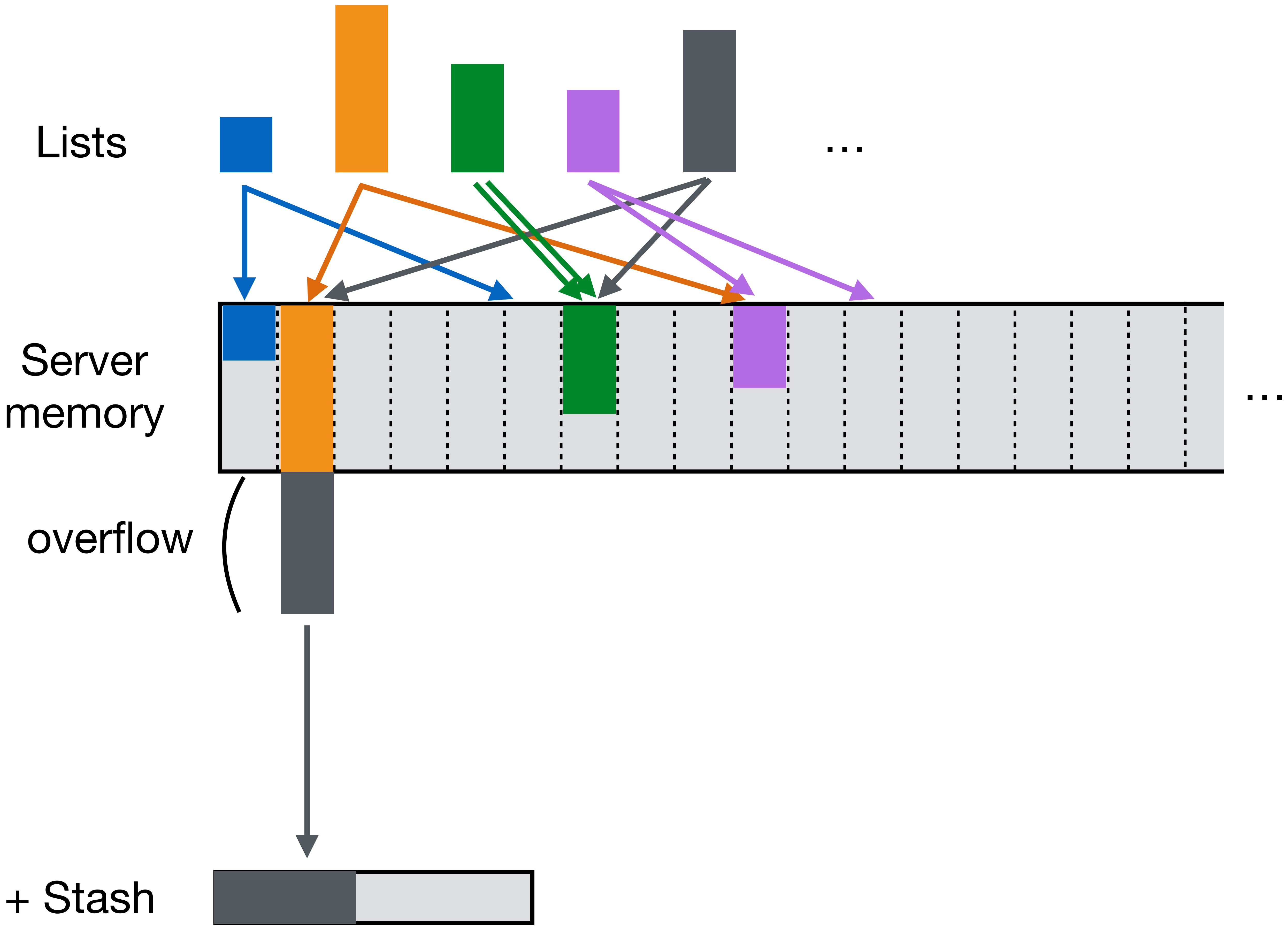
Tethys allocation



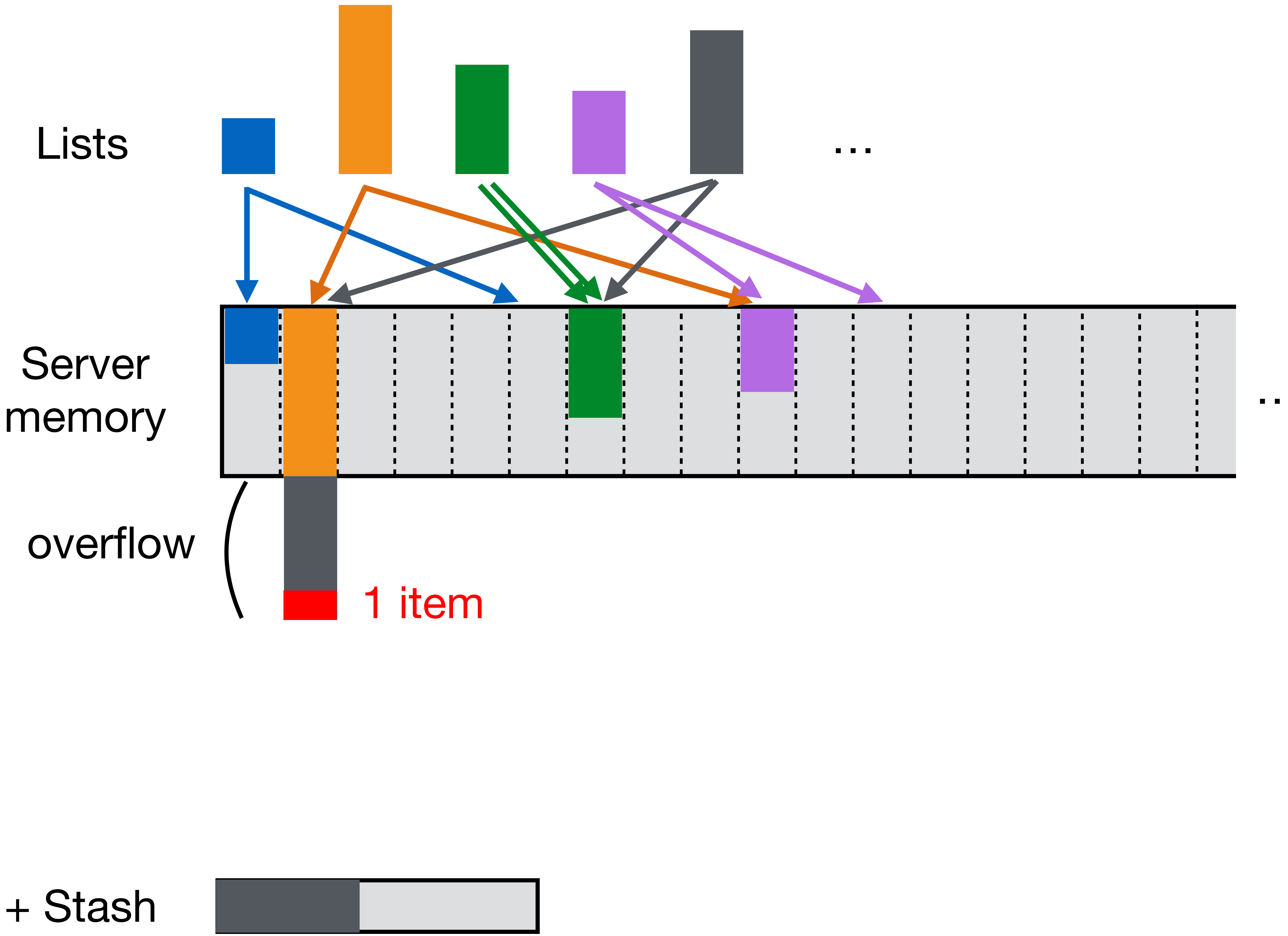
Tethys allocation



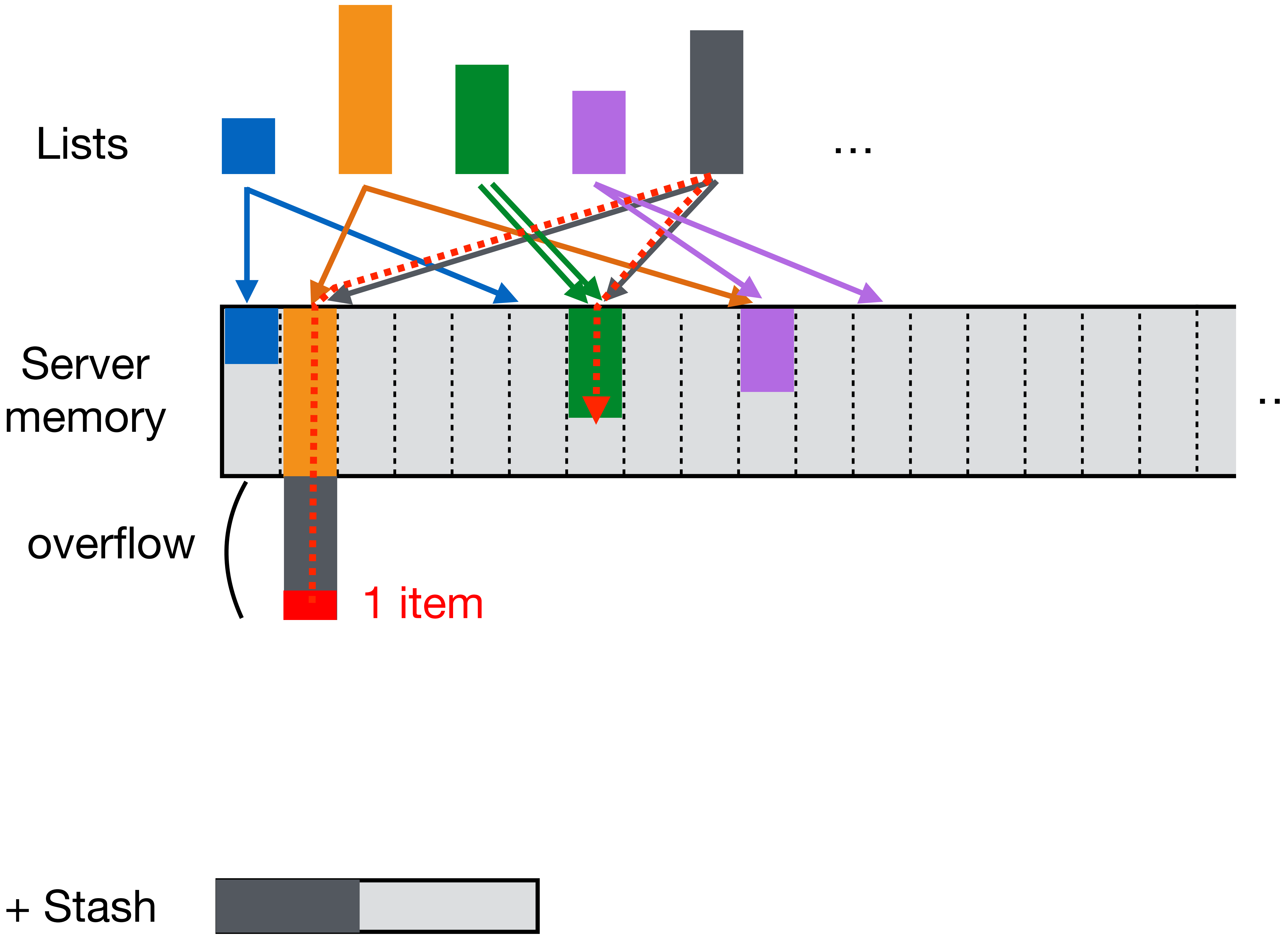
Tethys allocation



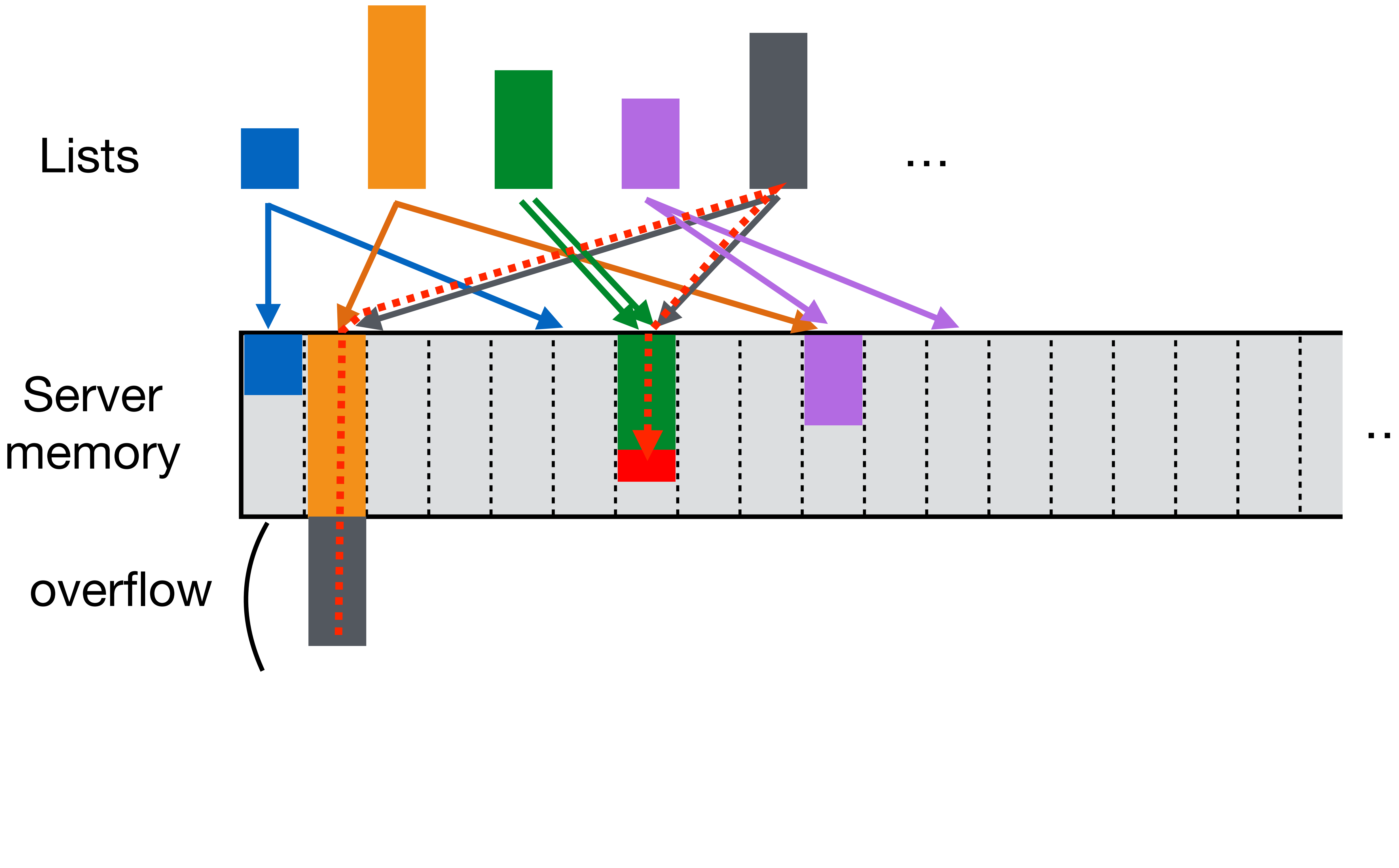
Tethys



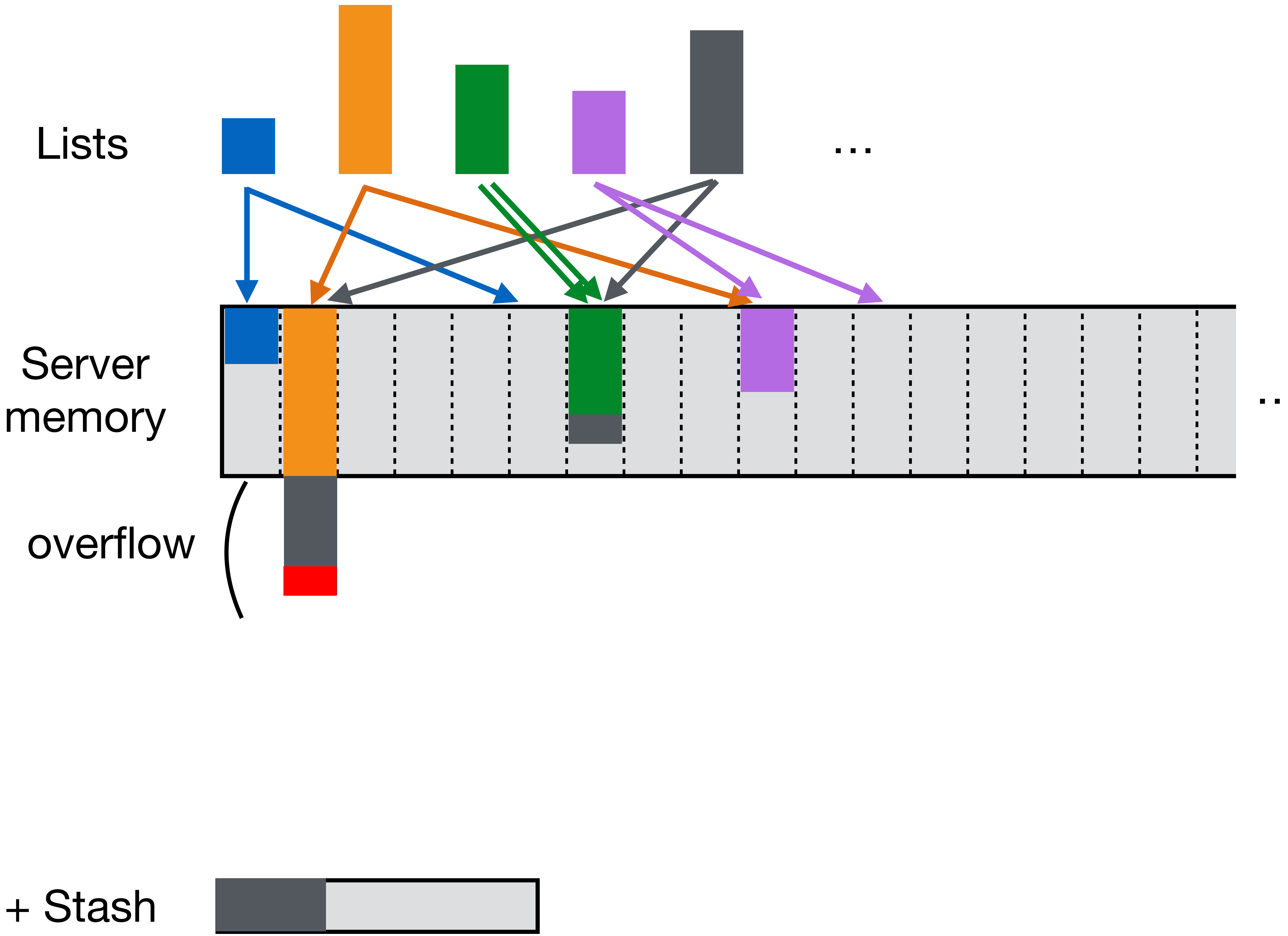
Tethys



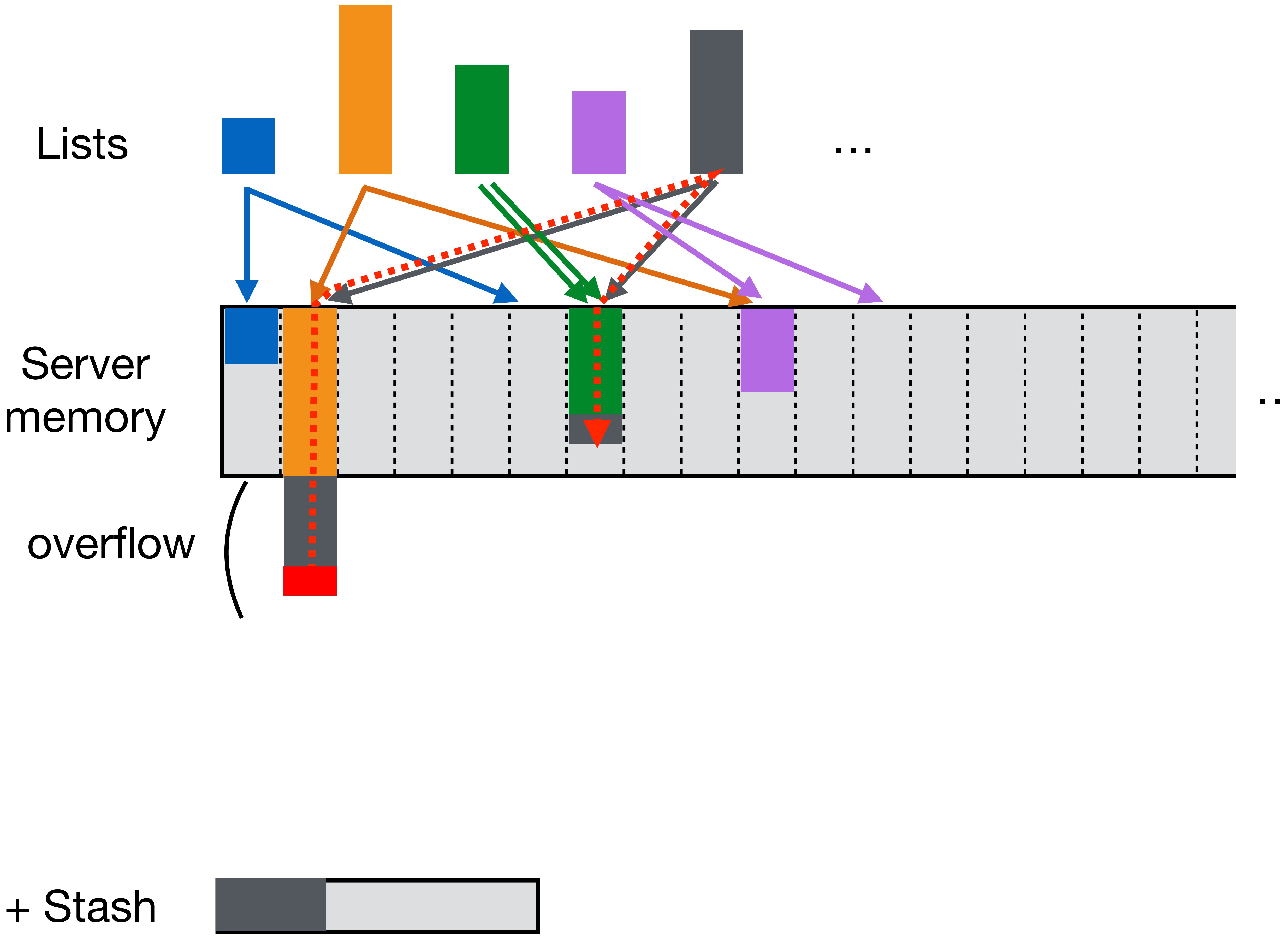
Tethys



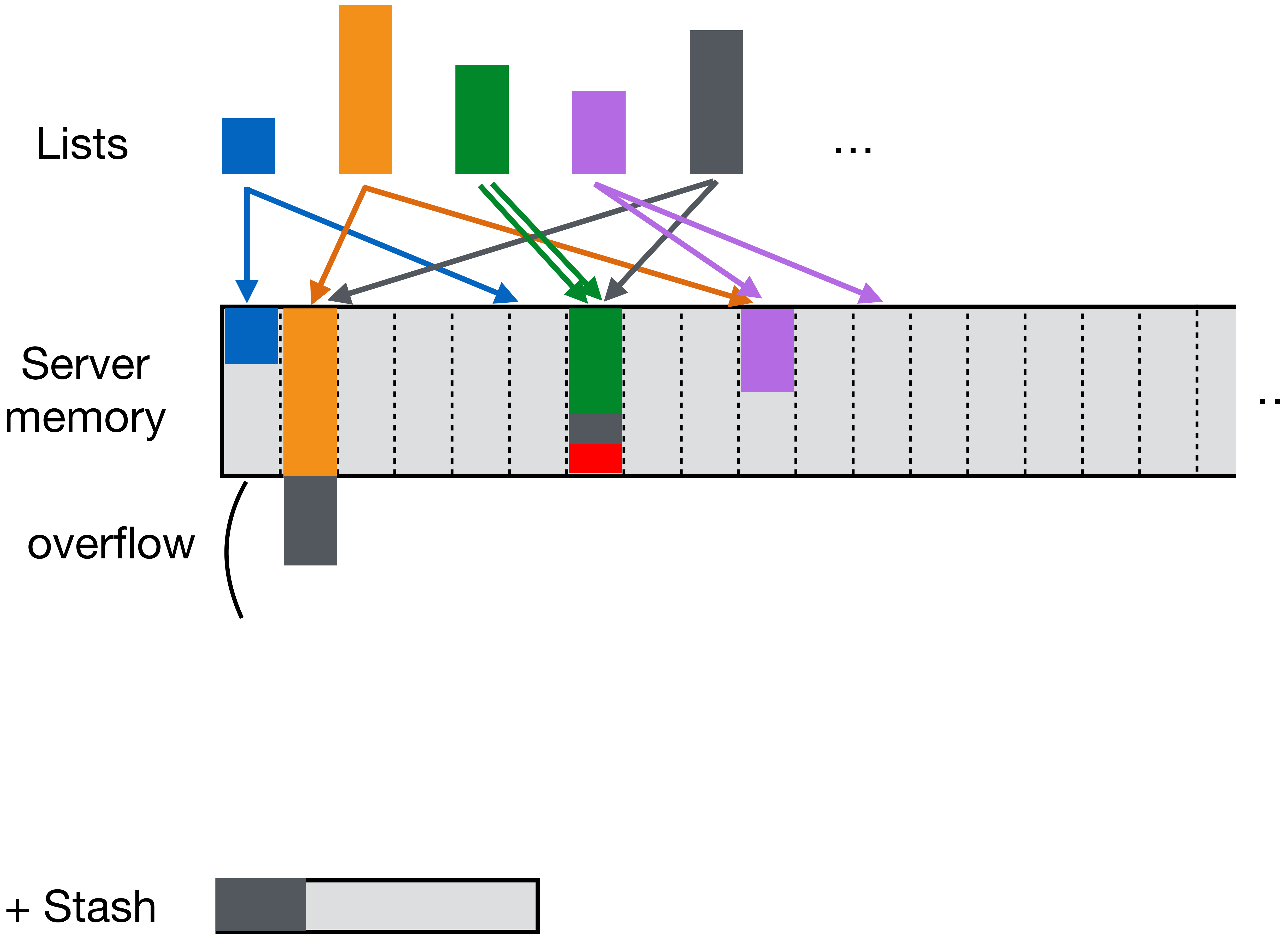
Tethys



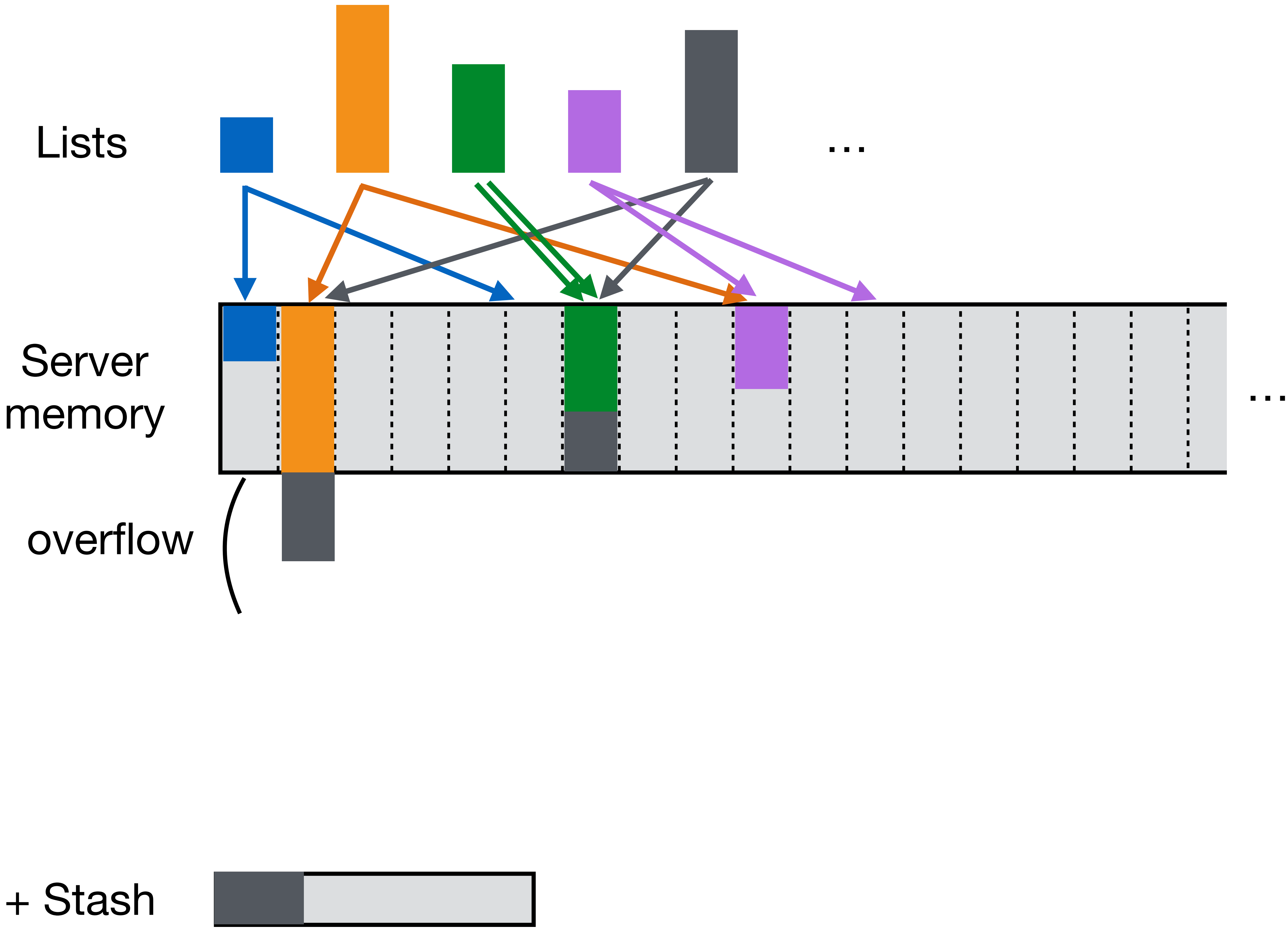
Tethys



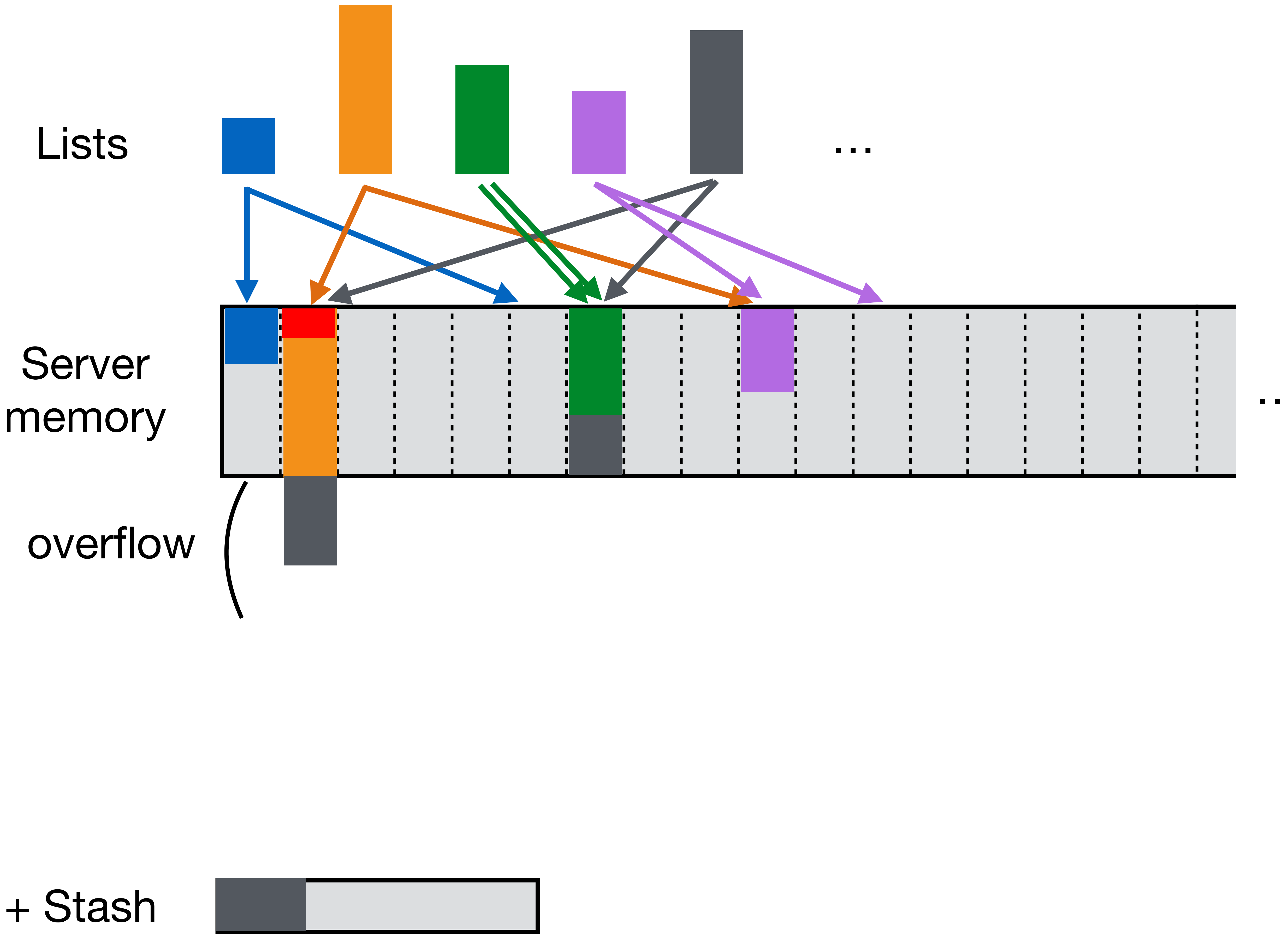
Tethys



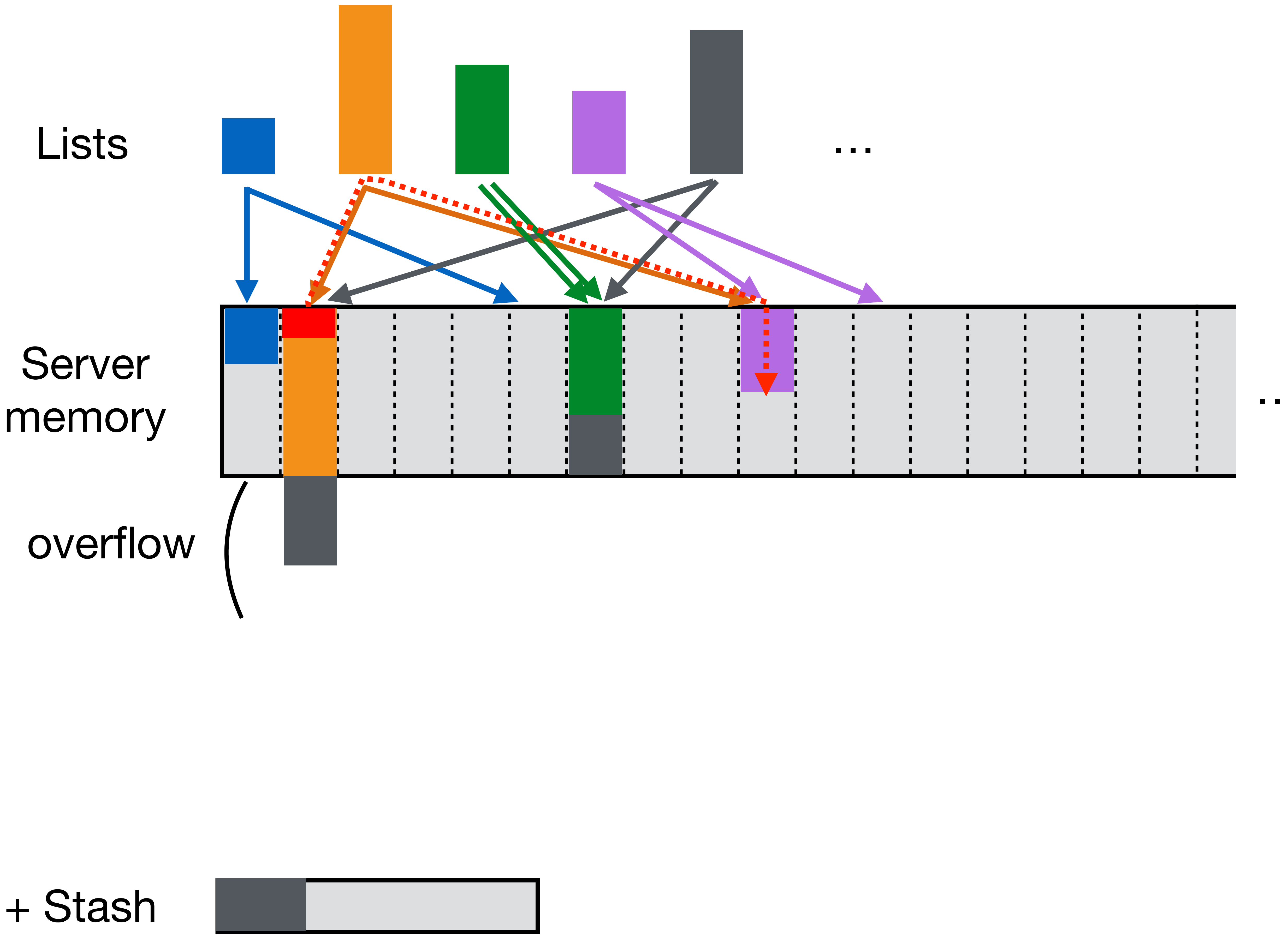
Tethys



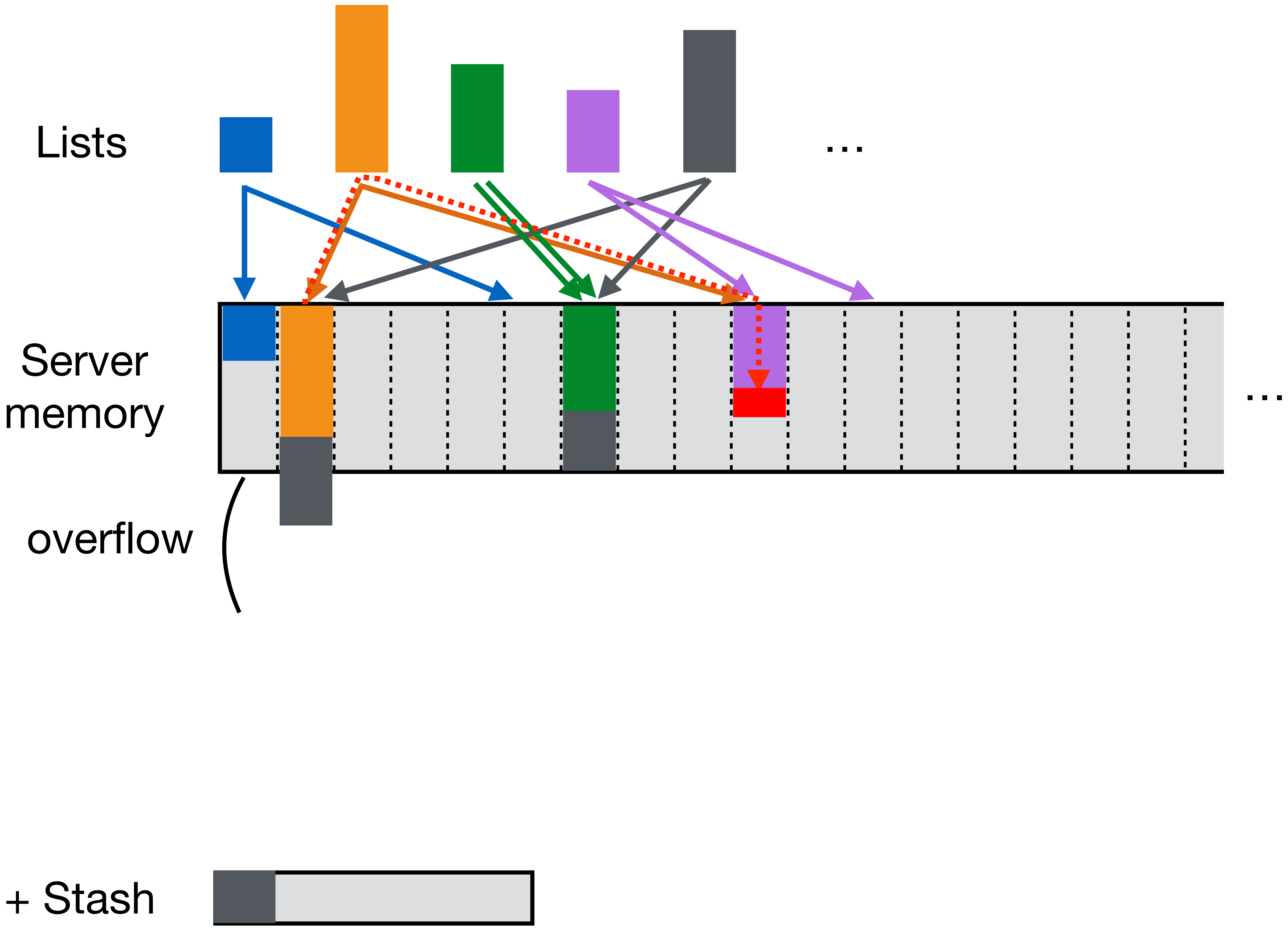
Tethys



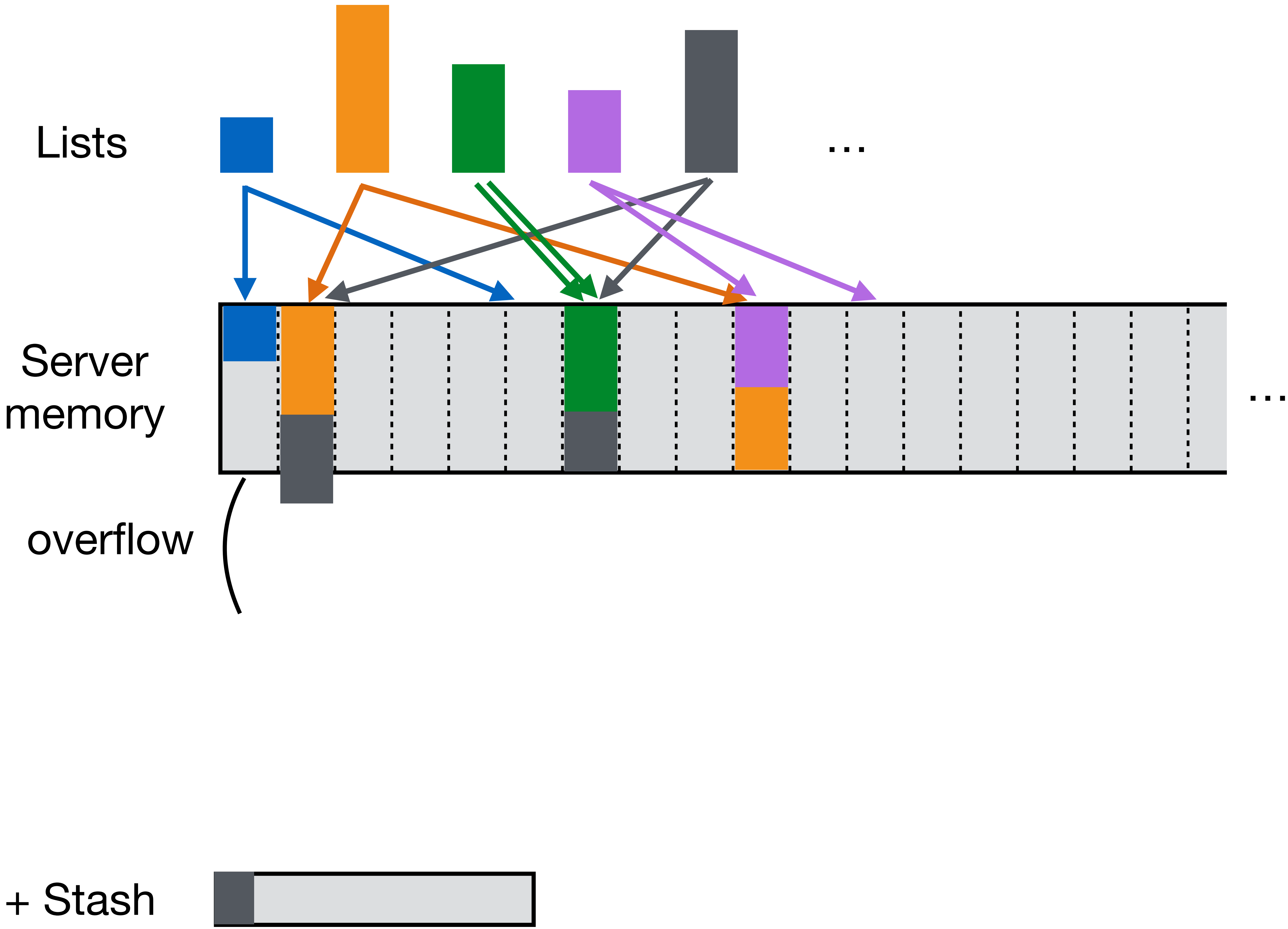
Tethys



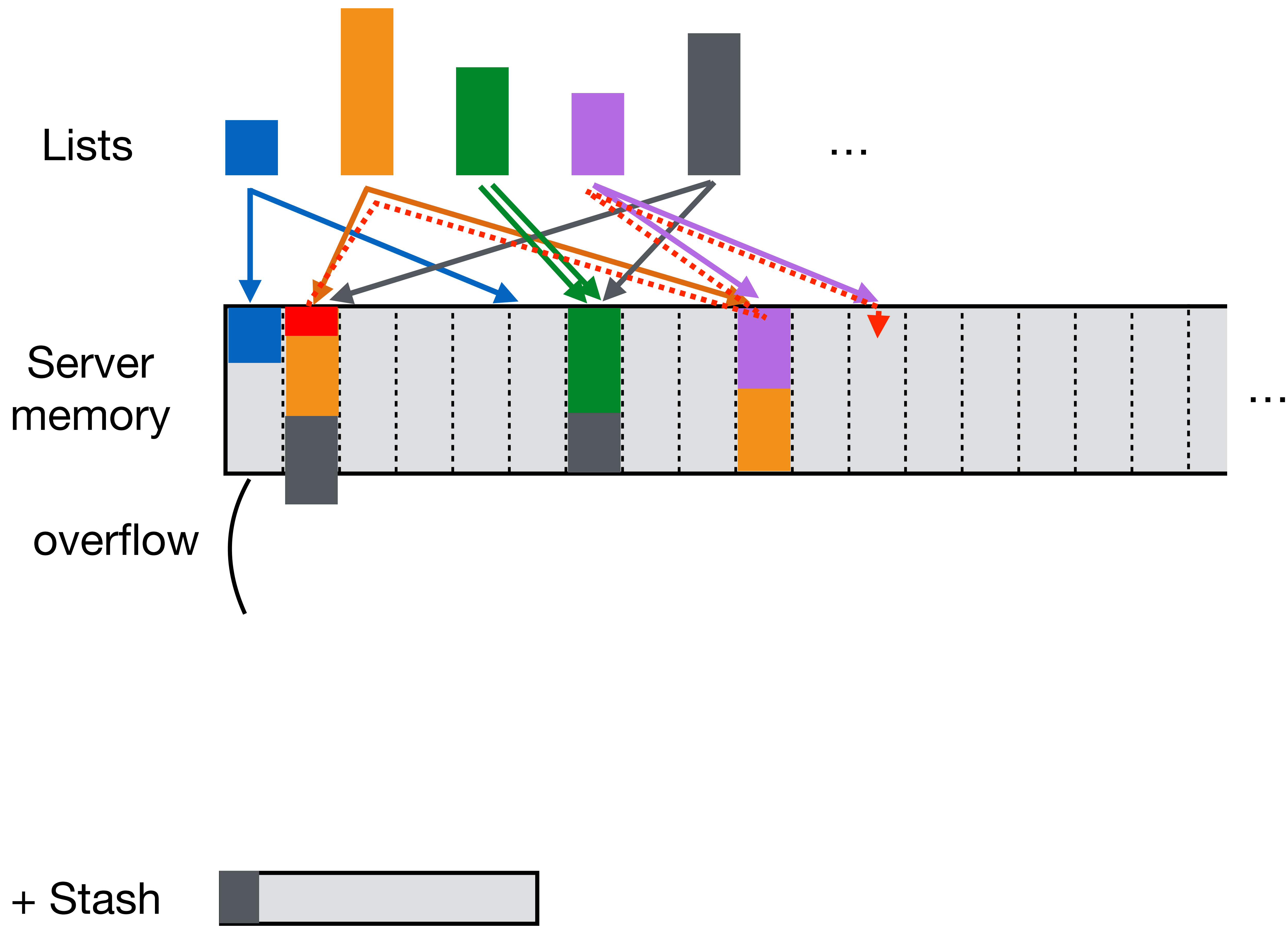
Tethys



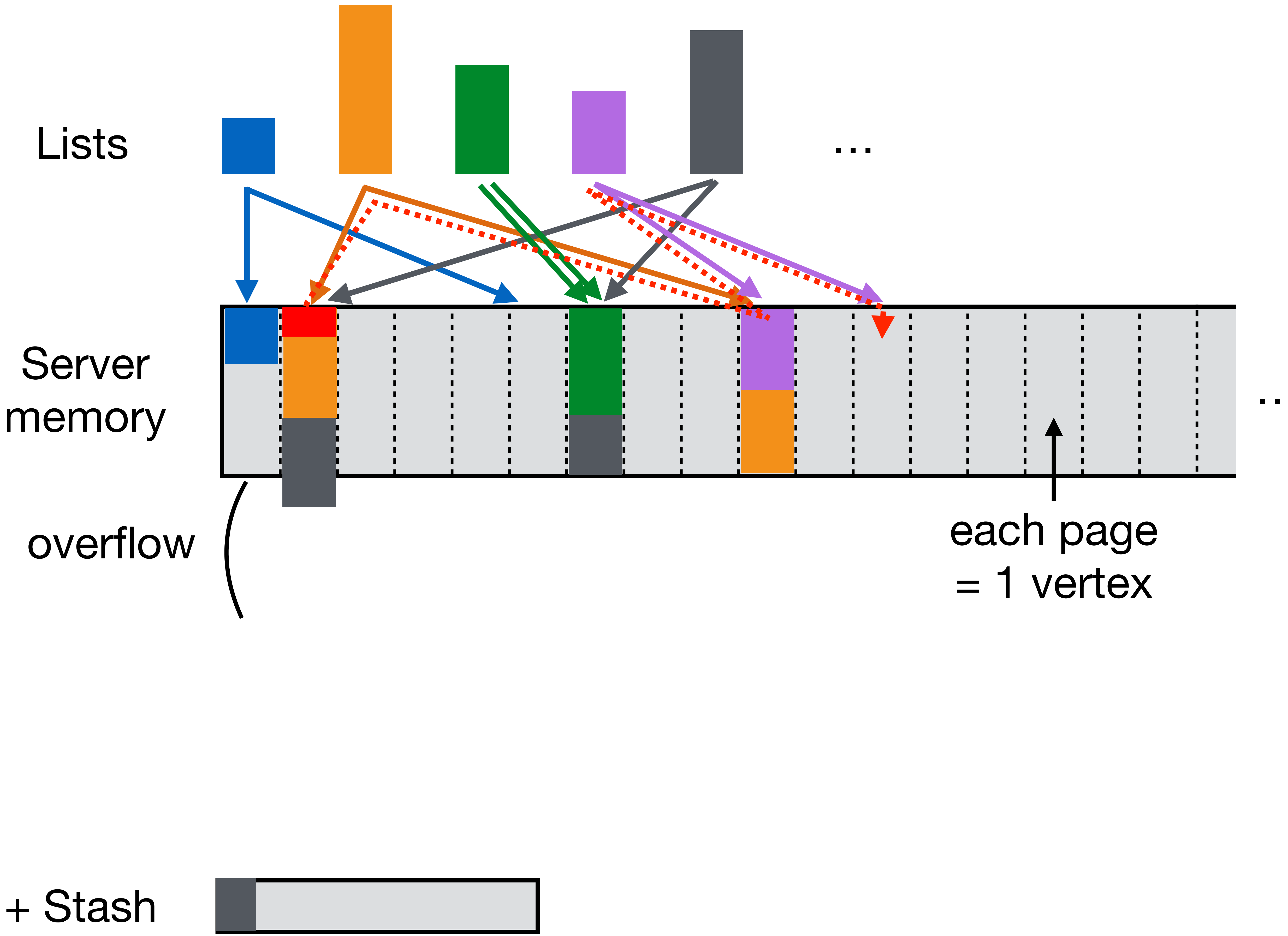
Tethys



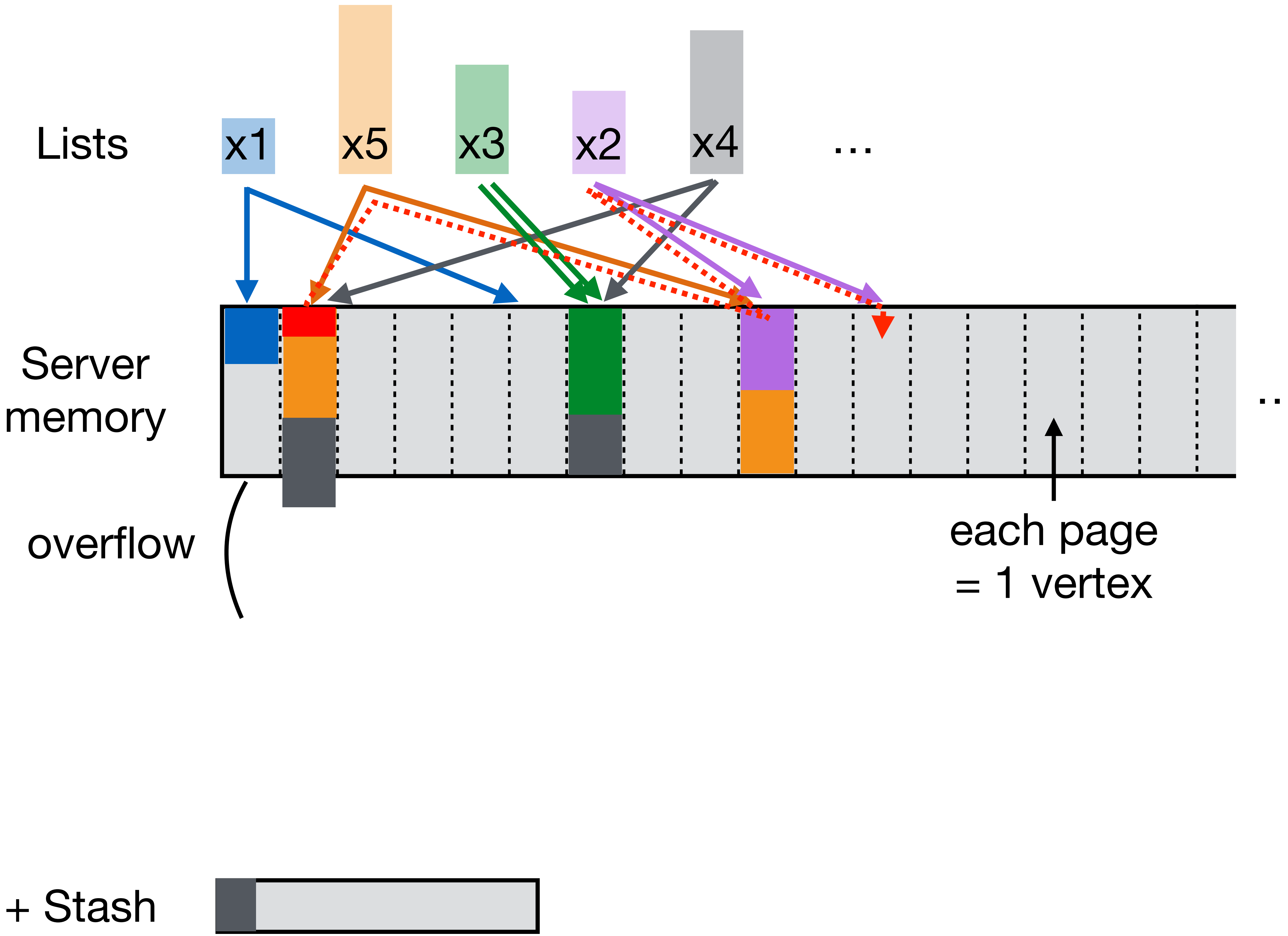
Tethys



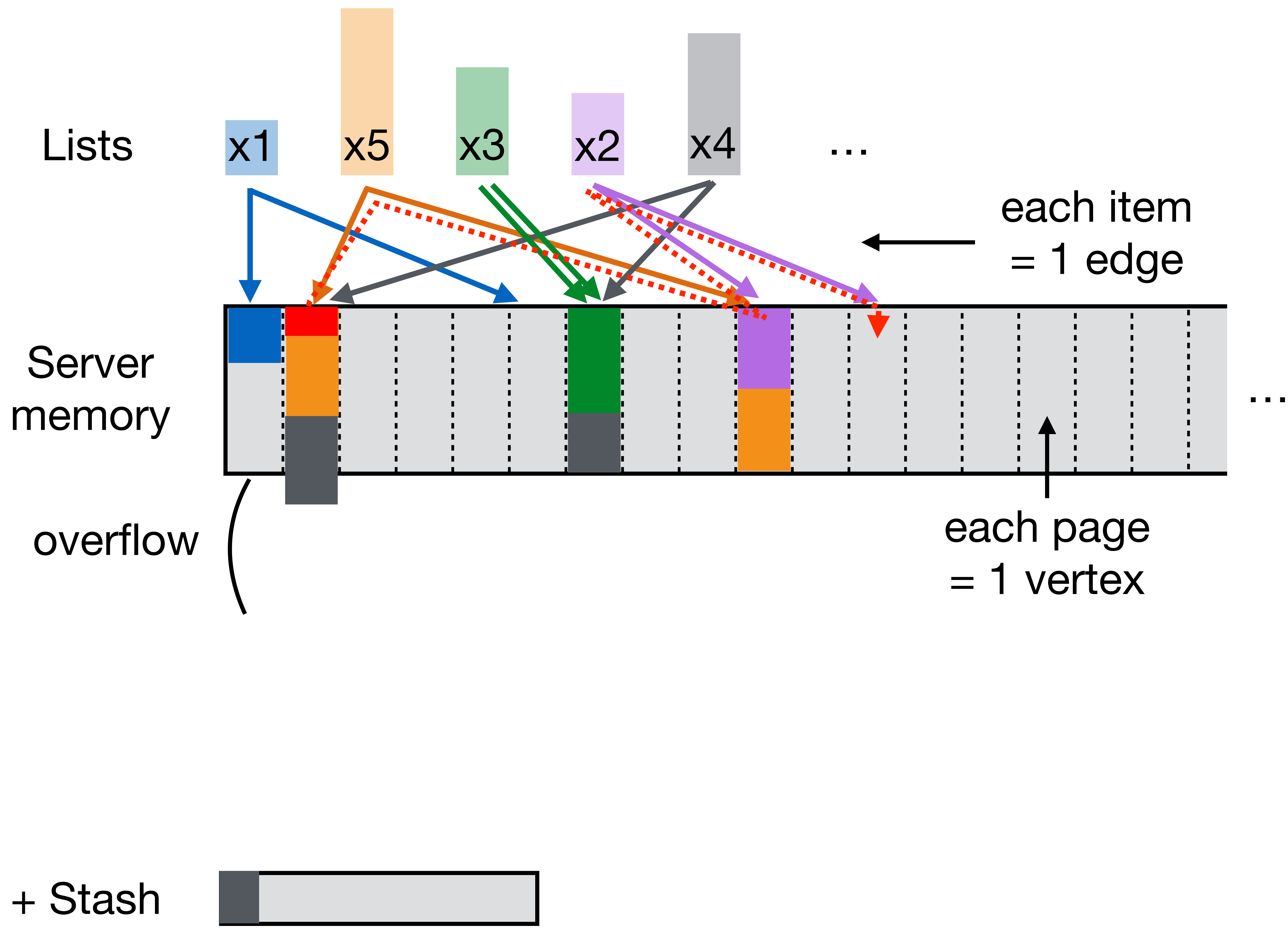
Tethys



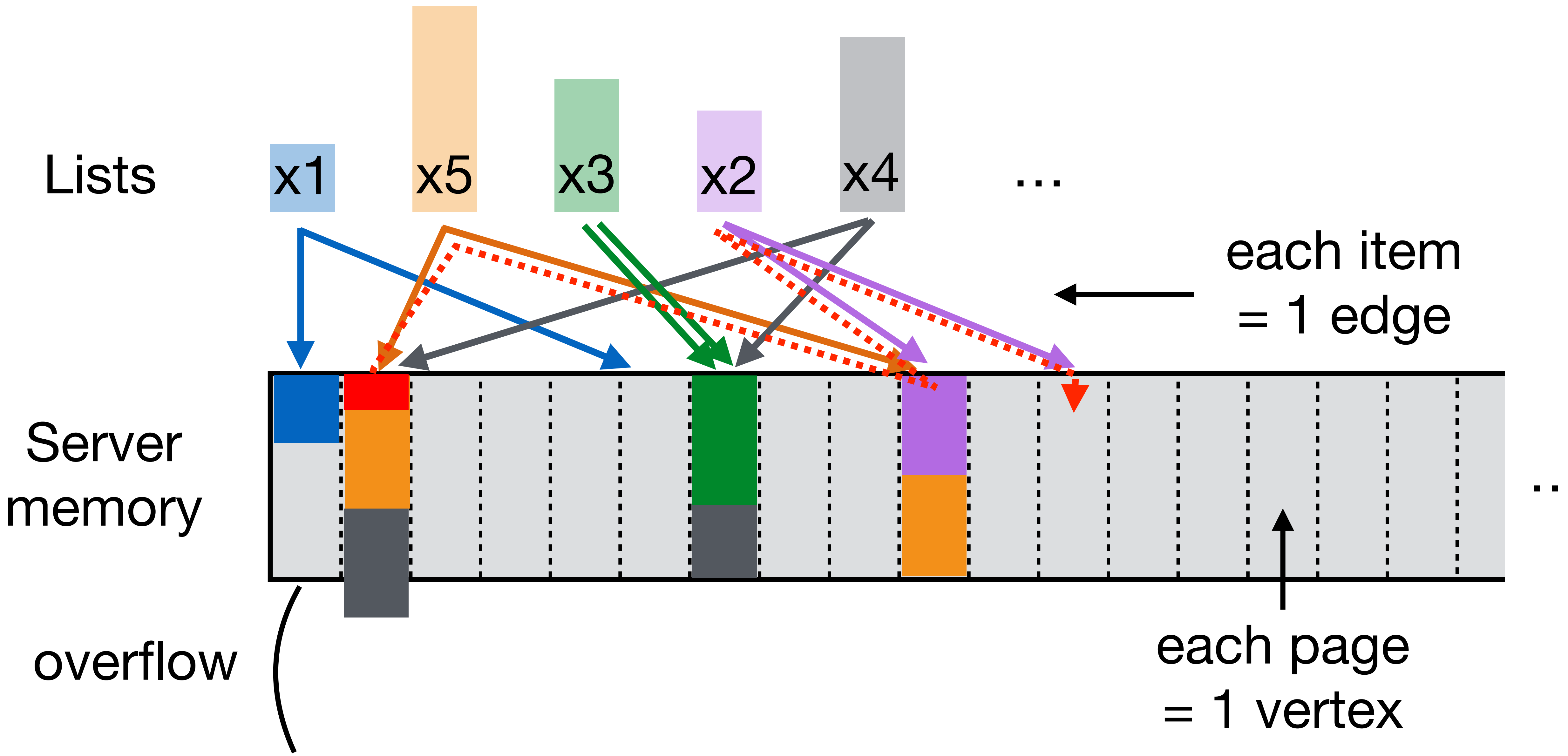
Tethys


Lists
x1
x5
x3
x2
x4
...
Server memory
...
overflow
each page = 1 vertex
+ Stash

Tethys

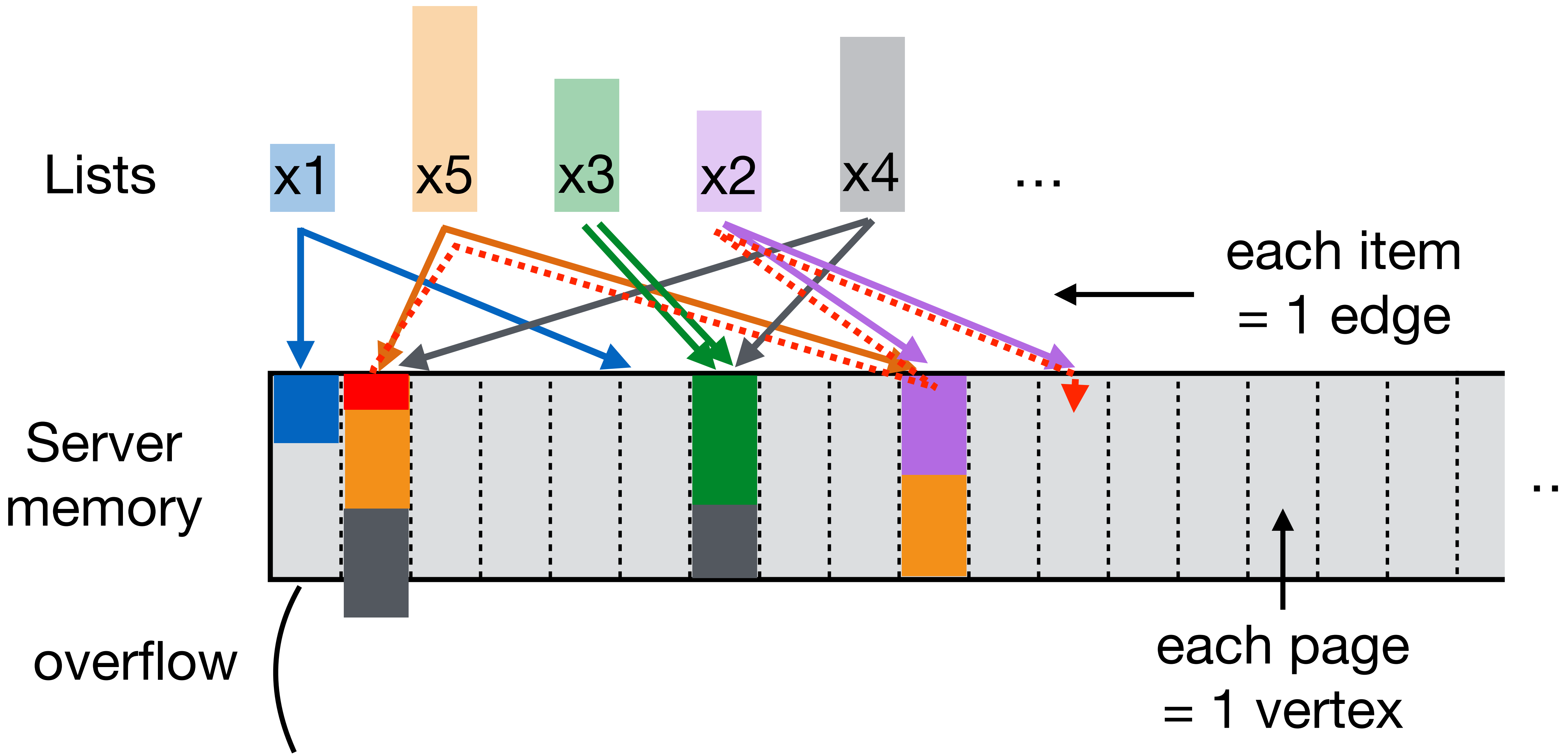

Lists
x1
x5
x3
x2
x4
...
each item
= 1 edge
Server
memory
...
overflow
each page
= 1 vertex
+ Stash

Tethys



Goal of algorithm: find maximal set of disjoint paths from overfull pages to underfull pages.

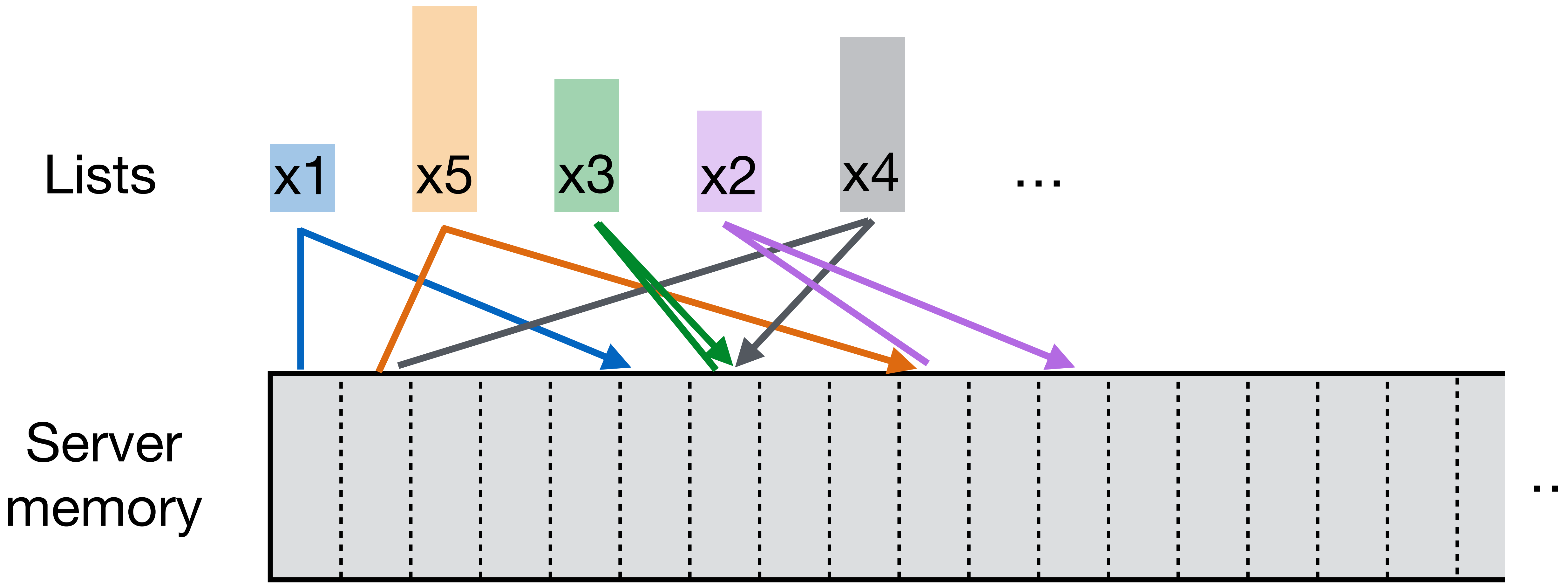
Tethys



Goal of algorithm: find maximal set of disjoint paths from overfull pages to underfull pages.

This is exactly a maximum flow algorithm.

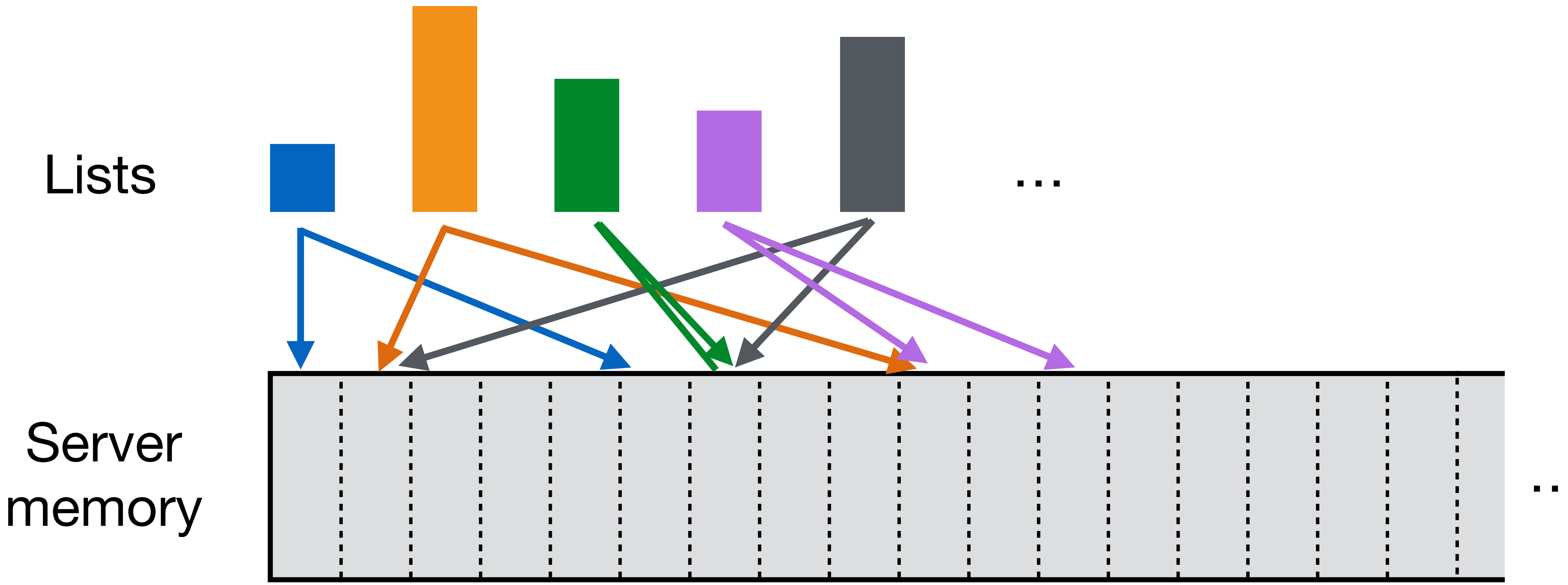
Tethys summary



Tethys allocation

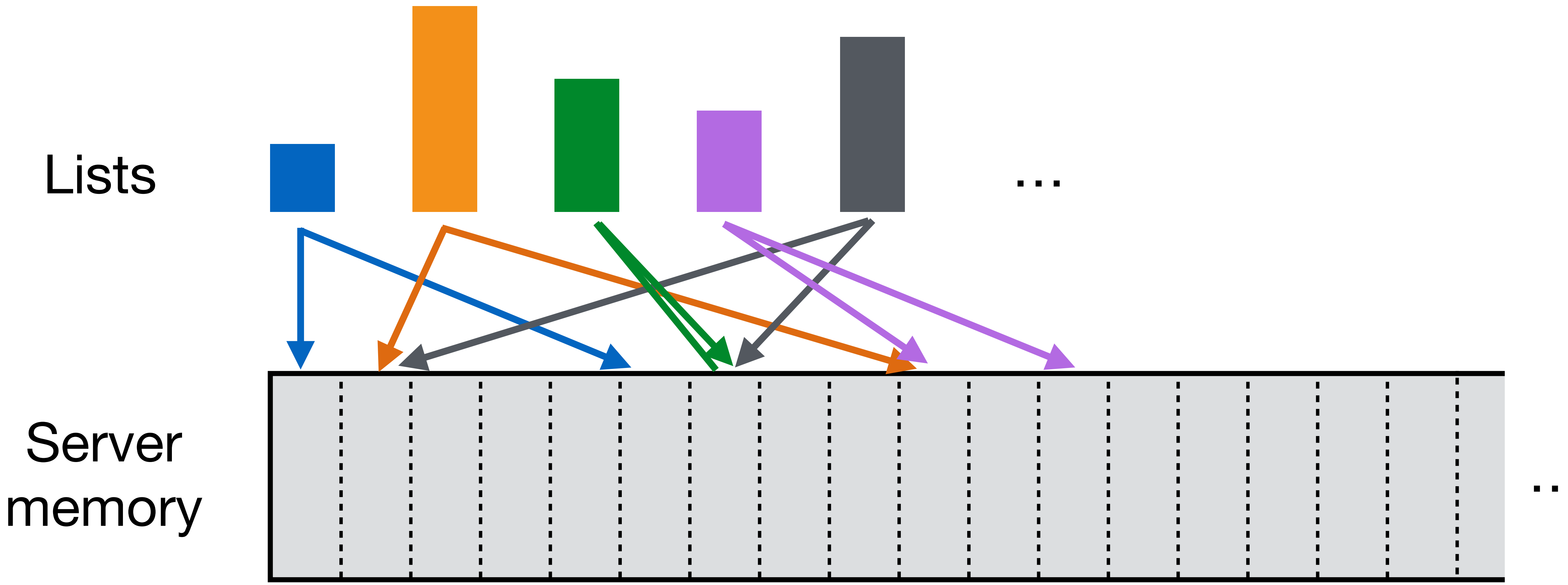
1. Assign 2 unif. random pages to each list.
2. Put each list in one of the two pages (arbitrarily).
3. Compute max flow over graph to find set of paths.
4. Move items along paths.
5. Items that still overflow, if any, go to the stash.

Optimization problem



Objective: minimize stash size
under **constraint:** all items are assigned somewhere, no page overflows.

Optimization problem



Objective: minimize stash size
under **constraint:** all items are assigned somewhere, no page overflows.

Tethys is **optimal** wrt this optimization problem.

i.e. outputs minimal stash size regardless of starting graph.

What size is the stash?

<u>Parameter choice</u>

Let n = **Σ** list sizes/p = #*pages* in DB.

Let s = #*pages* in stash.

Pick m = (2 + ε)n pages for server memory, for any cst ε>0.

<u>Theorem</u>

For **any** set of lists (s.t. n = **Σ** list sizes/p):

$$\text{Prob[min stash size} > \text{s]} = O(n^{-s/2})$$

What size is the stash?

Parameter choice

Let n = Σ list sizes/p = #*pages* in DB.

Let s = #*pages* in stash.

Pick m = (2 + ε)n pages for server memory, for any cst ε>0.

Theorem

For **any** set of lists (s.t. n = Σ list sizes/p):

$$\text{Prob}[\text{min stash size} > s] = O(n^{-s/2})$$

→ stash size $\omega(\log \lambda)/\log n \Rightarrow$ prob of failure is negligible.

Stash is stored on the client side.
Does not grow with the size of the database.

Experiments: a few pages suffice.

Remarks about the proof

Let L be a multiset of list sizes s.t. $\mathbf{\Sigma}$ L = np.

Let M be the multiset $\{p,p,p,\ldots\}$ s.t. $\mathbf{\Sigma}$ M = np.

Central statement (simplified):

Prob[minStash(L) > s] ≤ Prob[minStash(M) > s]

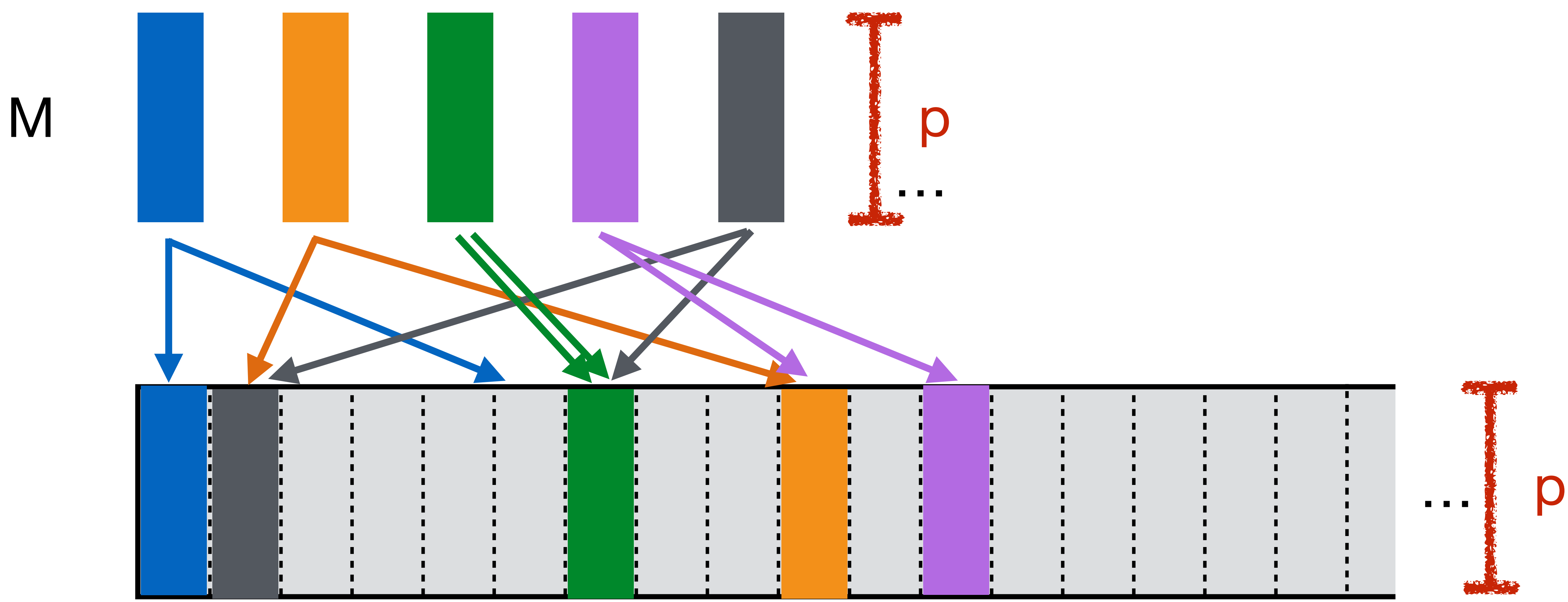
Remarks about the proof

Let L be a multiset of list sizes s.t. Σ L = np.

Let M be the multiset $\{p,p,p,\ldots\}$ s.t. Σ M = np.

Central statement (simplified):

Prob[minStash(L) > s] ≤ Prob[minStash(M) > s]



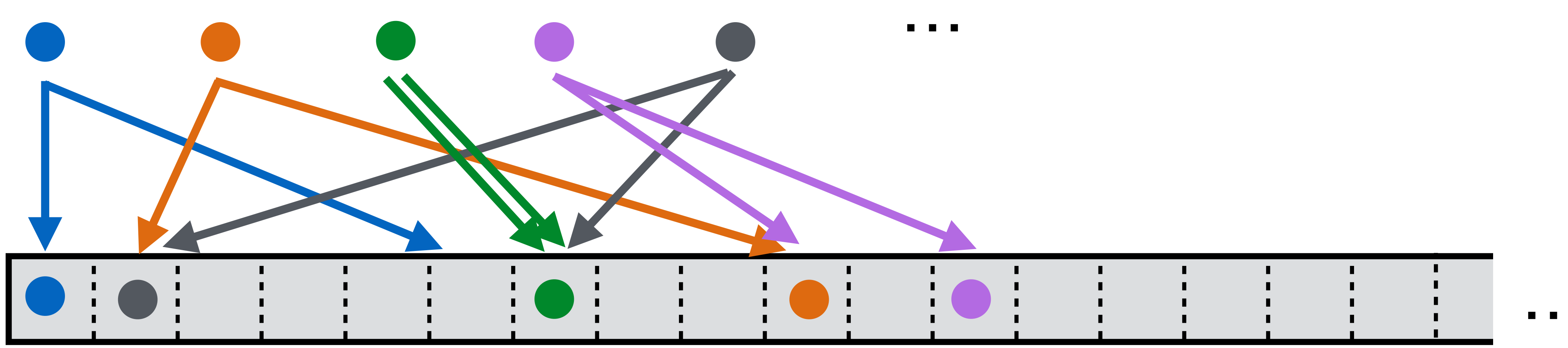
Remarks about the proof

Let L be a multiset of list sizes s.t. **Σ** L = *np*.

Let M be the multiset {*p*,*p*,*p*,...} s.t. **Σ** M = *np*.

Central statement (simplified):

Prob[minStash(L) > s] ≤ Prob[minStash(M) > s]



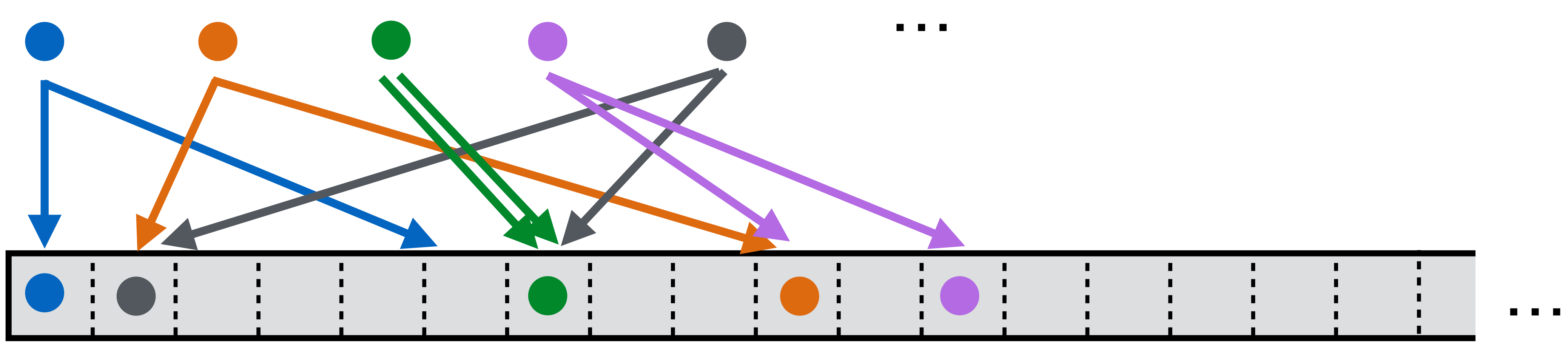
Remarks about the proof

Let L be a multiset of list sizes s.t. Σ L = np.

Let M be the multiset {p,p,p,...} s.t. Σ M = np.

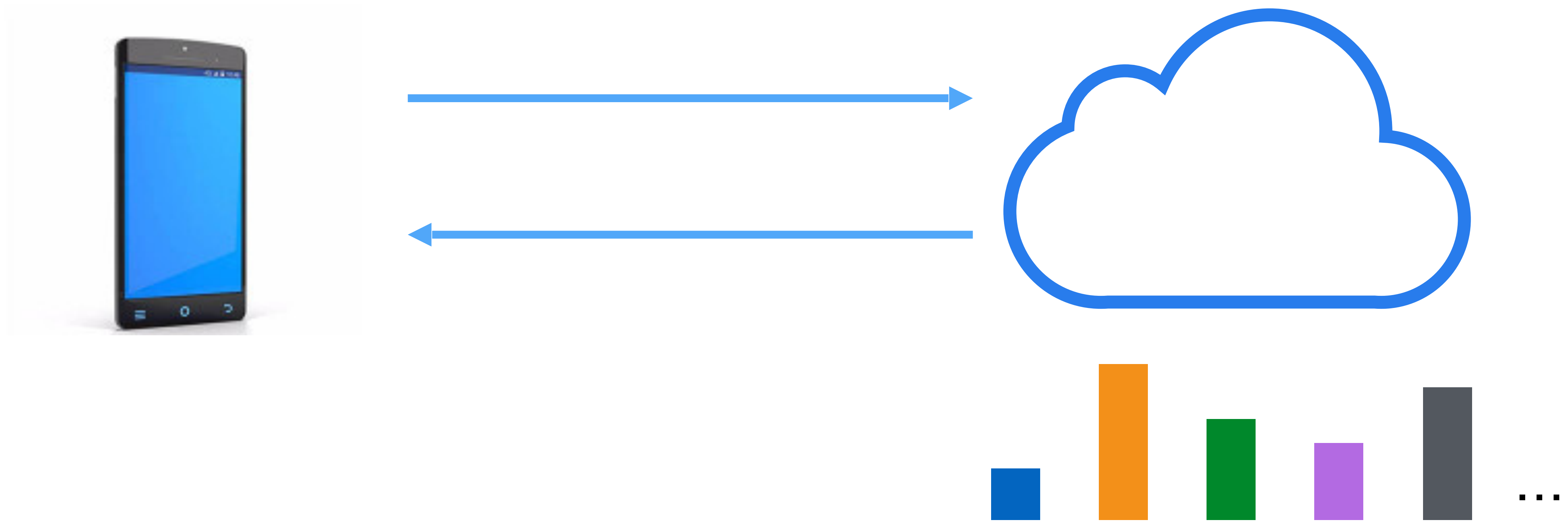
Central statement (simplified):

Prob[minStash(L) > s] ≤ Prob[minStash(M) > s]

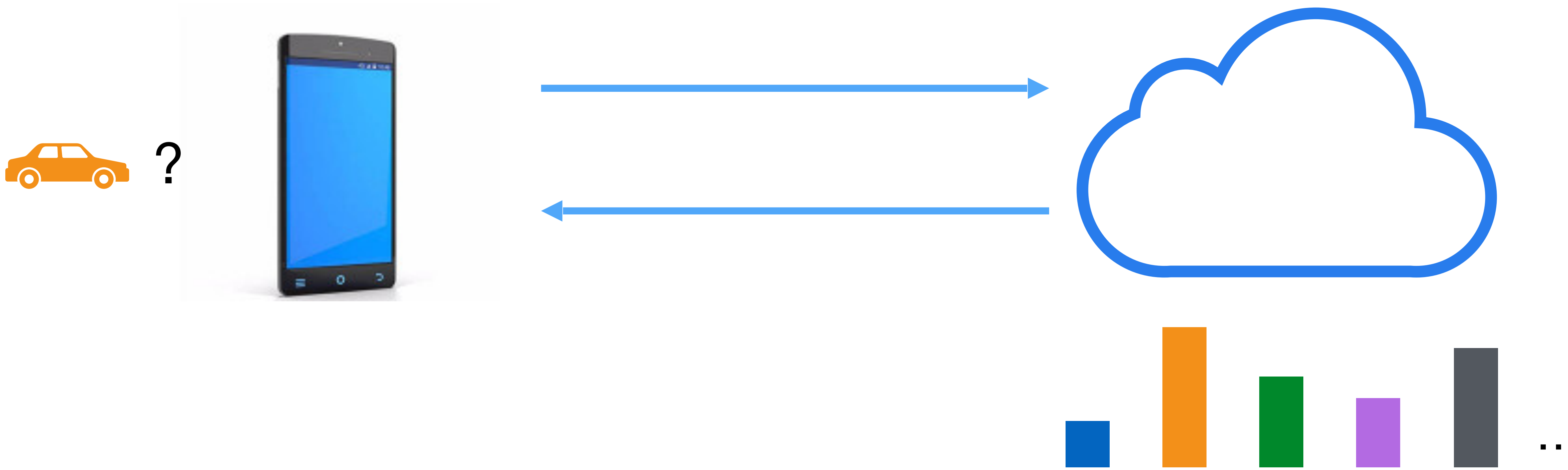


This is cuckoo hashing!

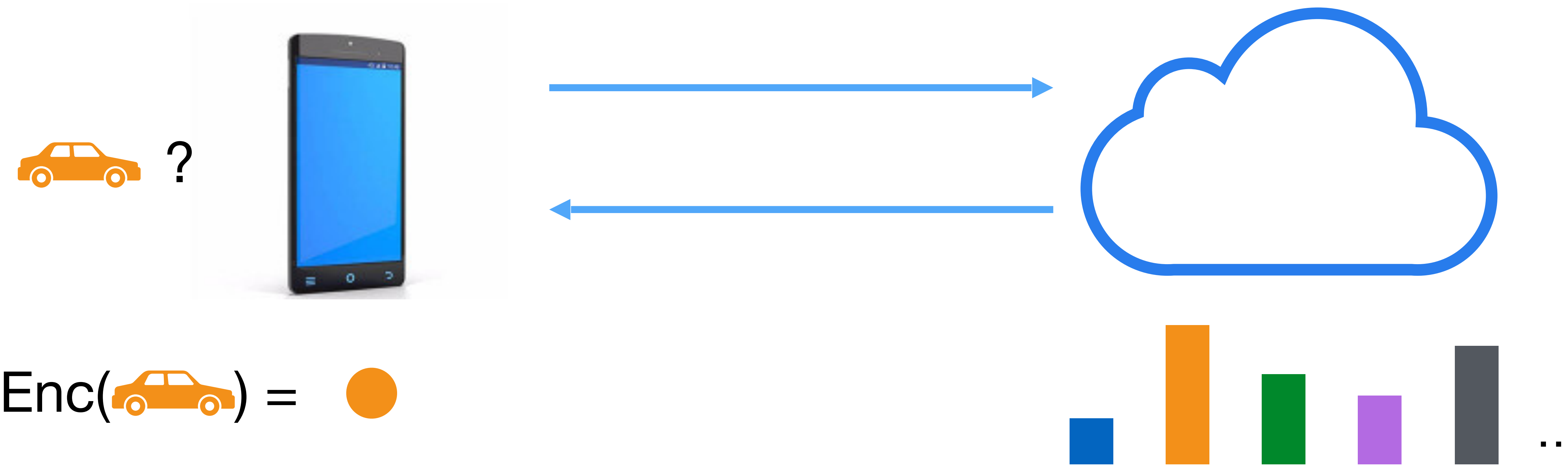
Tethys SSE



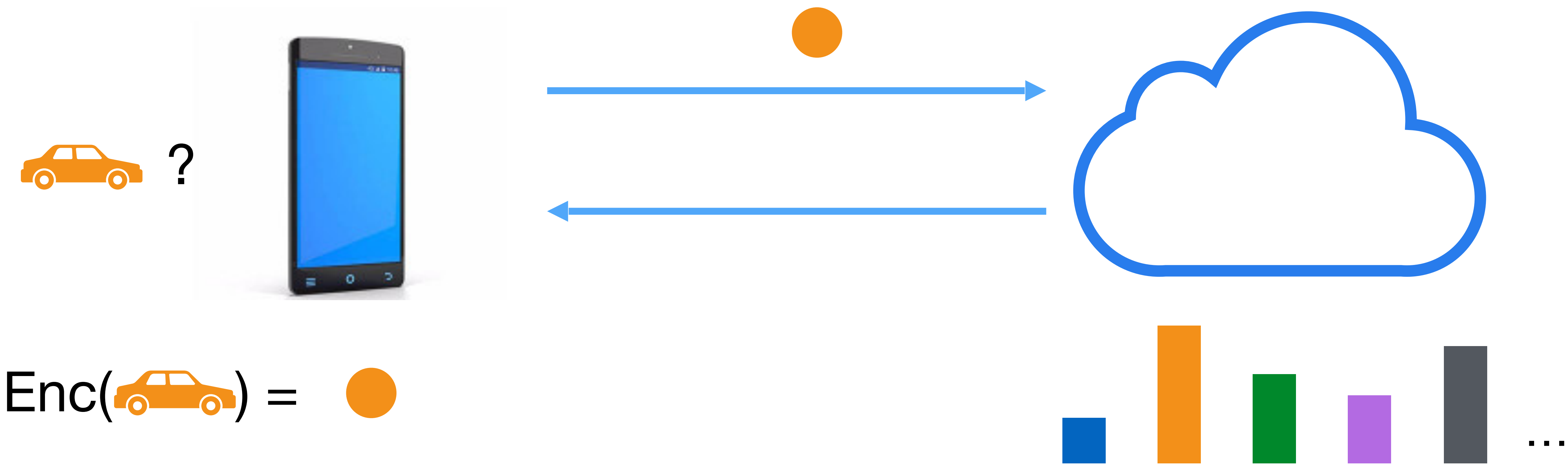
Tethys SSE



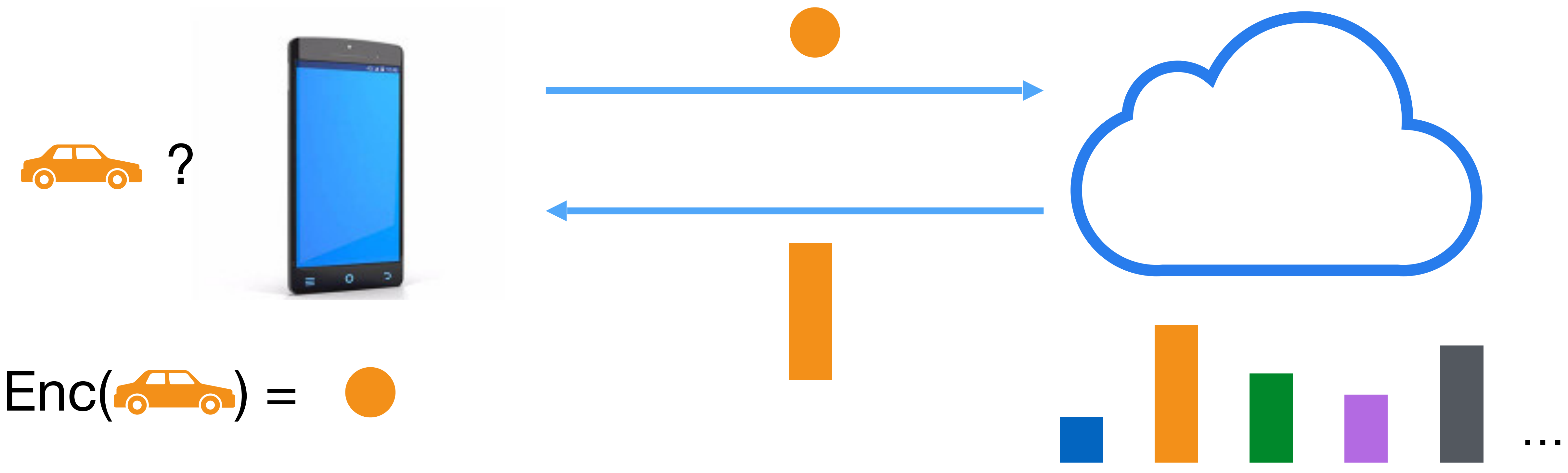
Tethys SSE



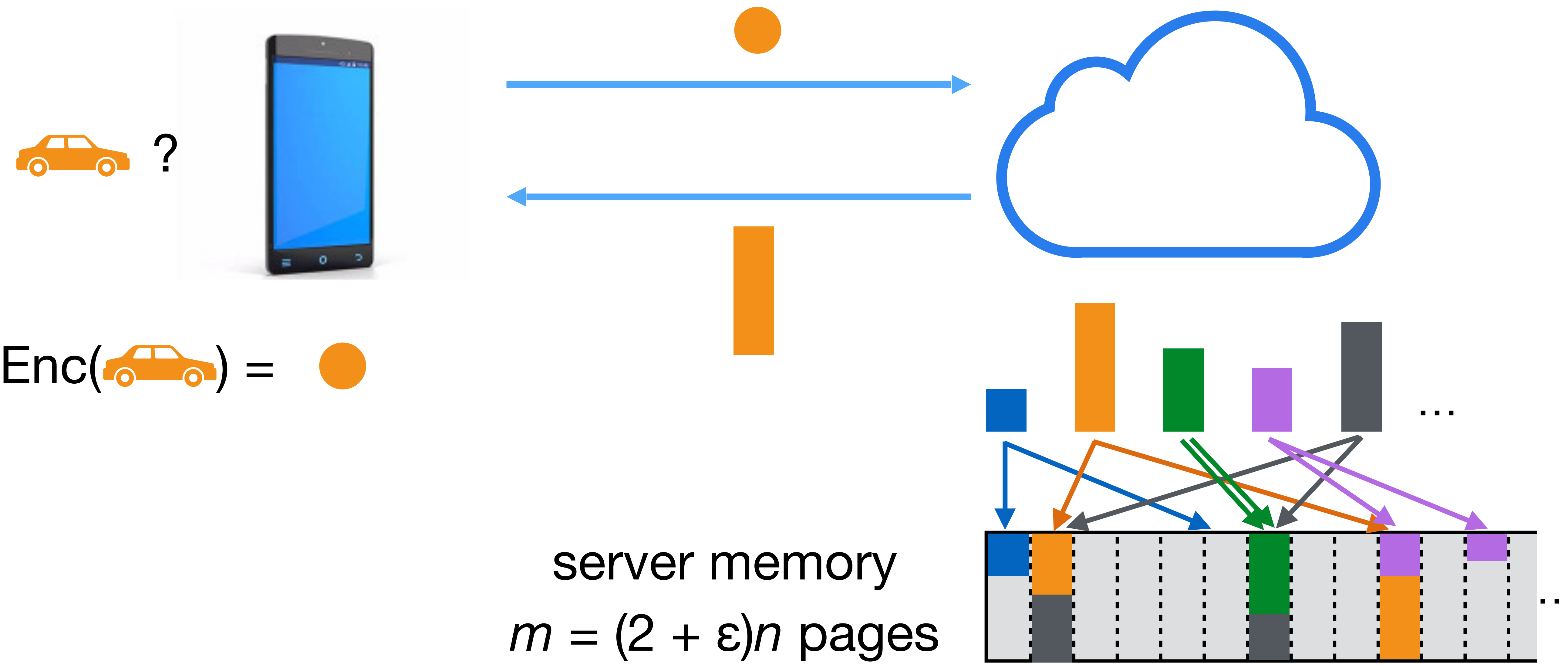
Tethys SSE



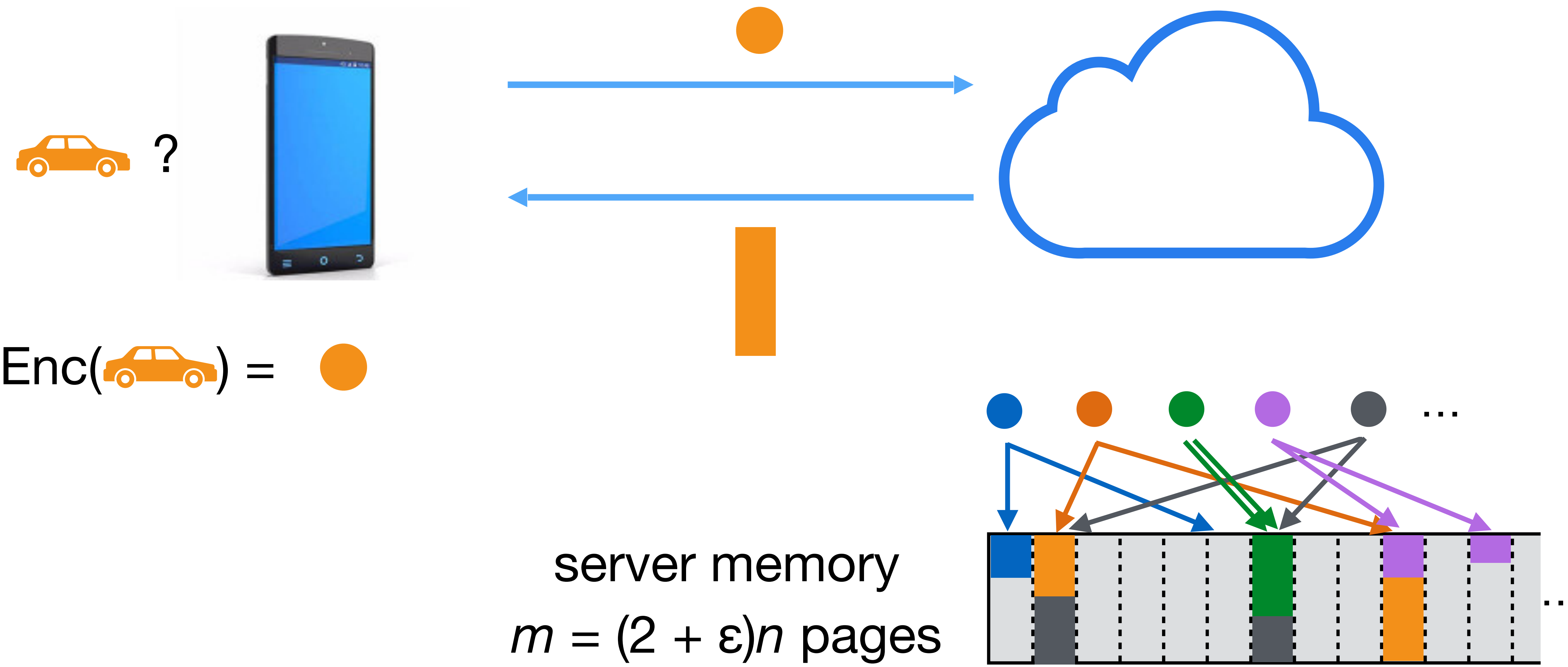
Tethys SSE



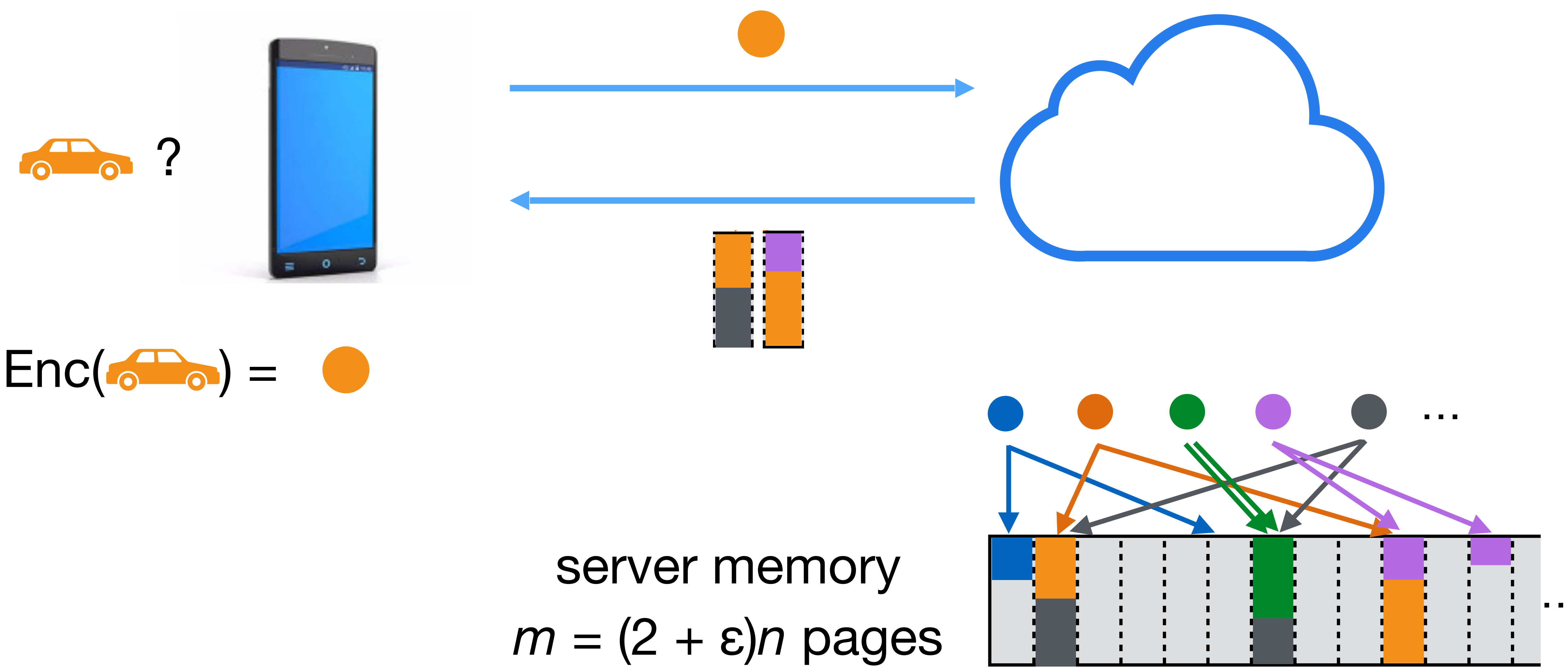
Tethys SSE



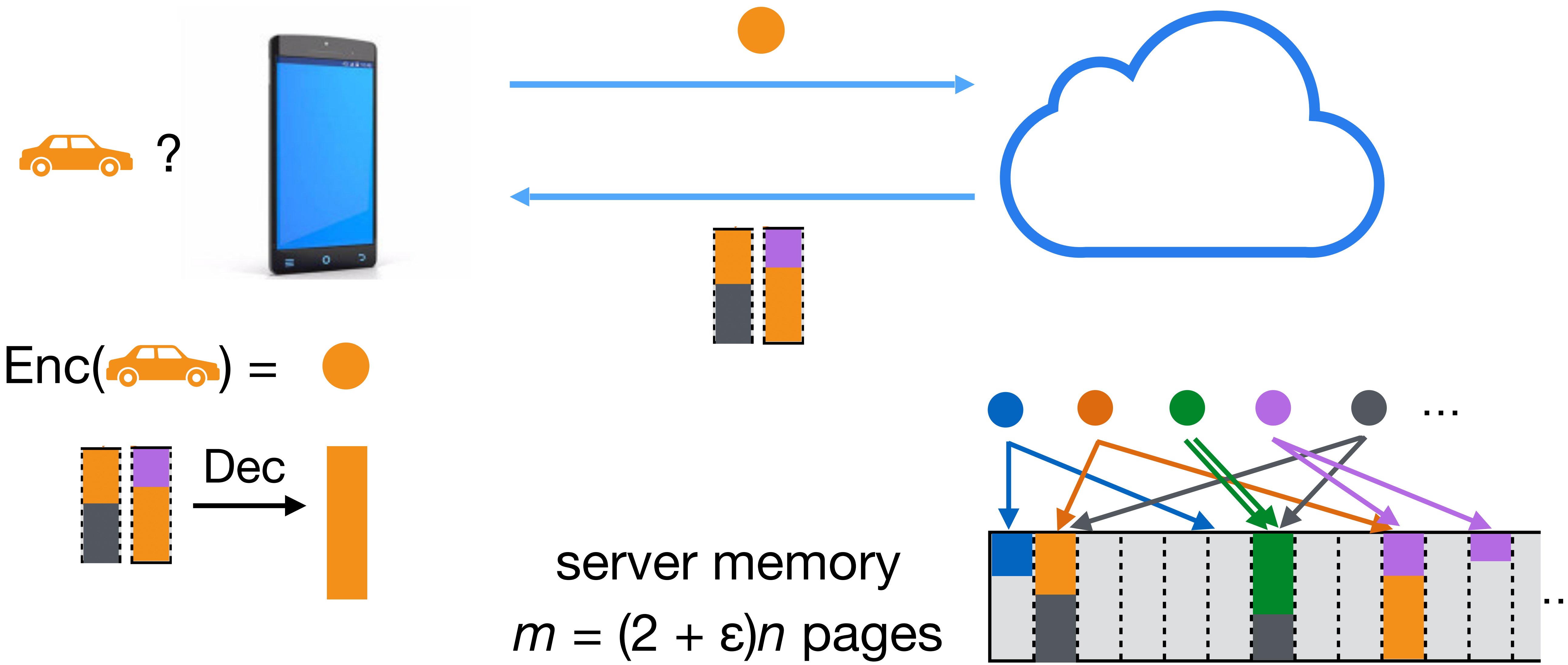
Tethys SSE



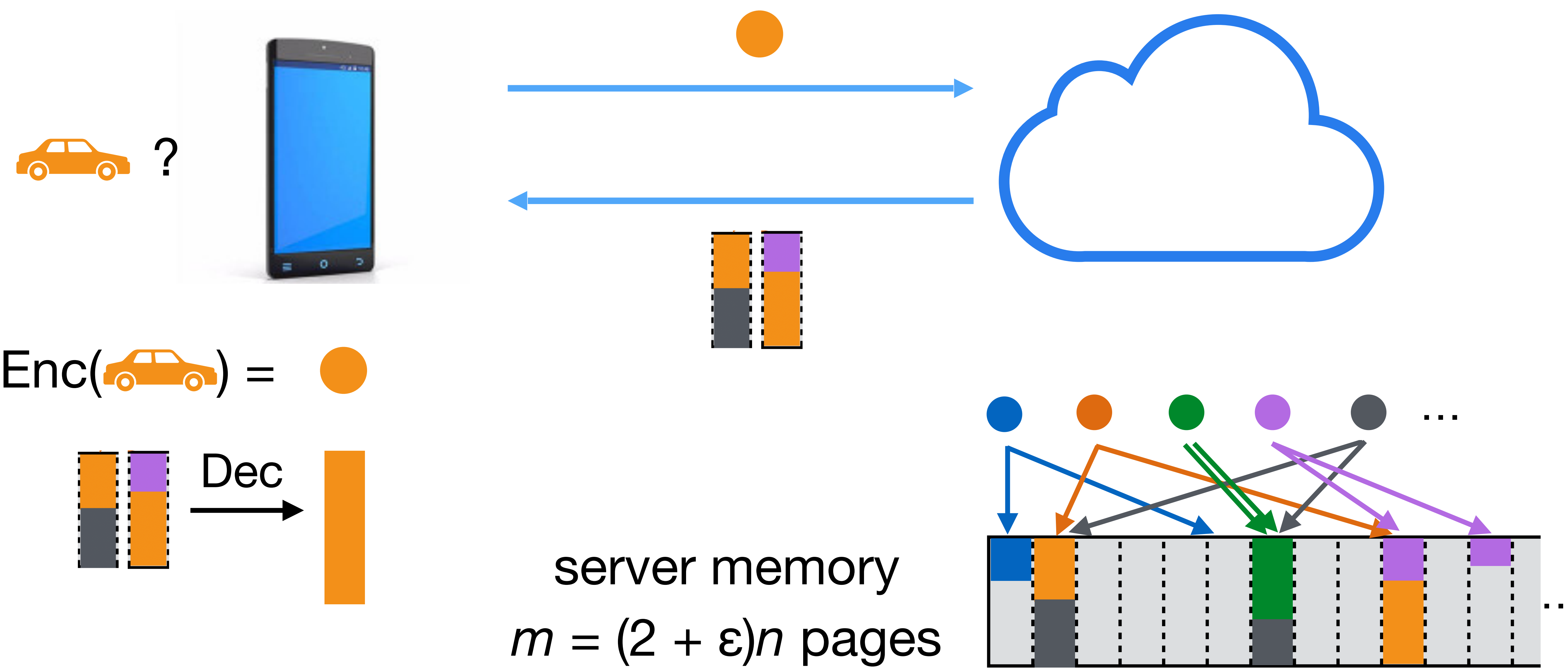
Tethys SSE



Tethys SSE



Tethys SSE



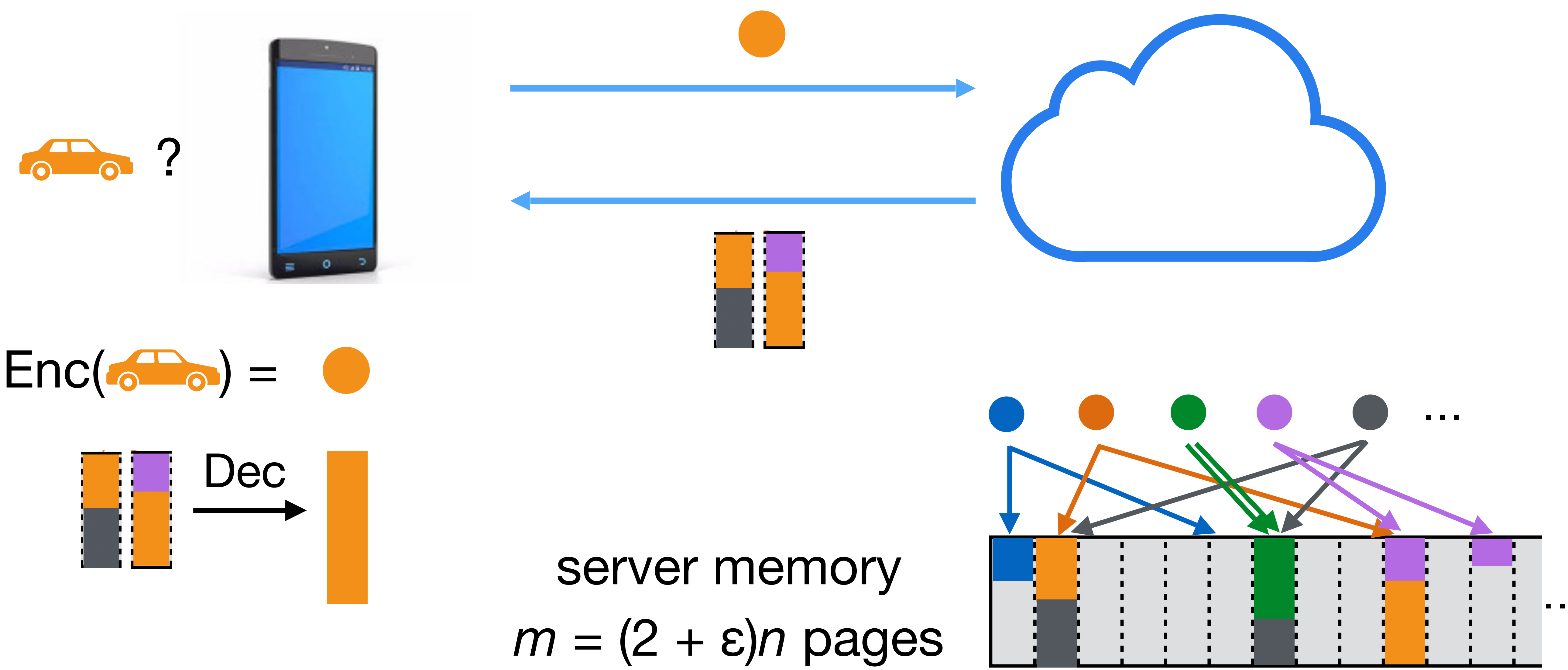
Performance:

Page efficiency: #pages accessed to get one list

Storage efficiency: #pages to store encrypted DB / n

Client storage: $\omega(\log \lambda)/\log n$ pages.

Tethys SSE



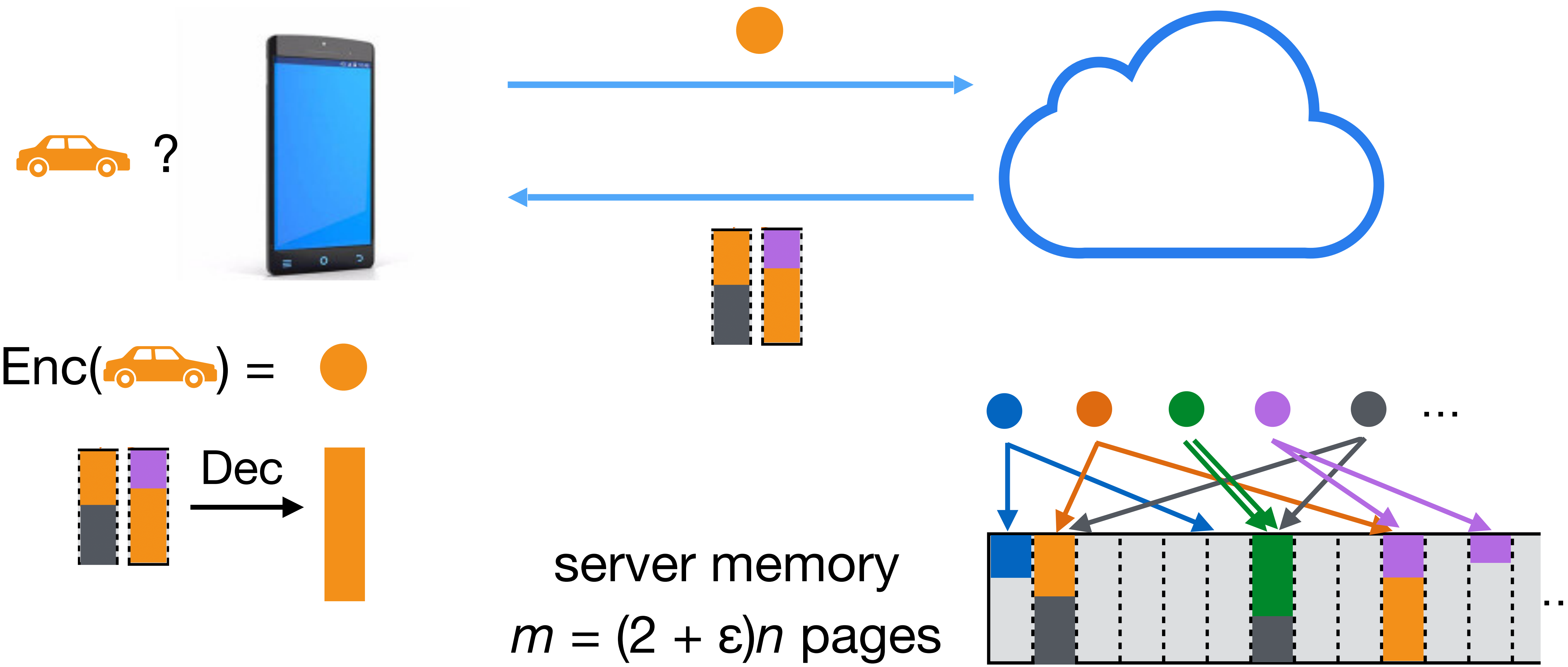
Performance:

Page efficiency: #pages accessed to get one list = 2 (+1)

Storage efficiency: #pages to store encrypted DB / n

Client storage: $\omega(\log \lambda)/\log n$ pages.

Tethys SSE



Performance:

Page efficiency: #pages accessed to get one list = 2 (+1)

Storage efficiency: #pages to store encrypted DB / n = 2+ε

Client storage: $\omega(\log \lambda)/\log n$ pages.

Performance evaluation

Theory

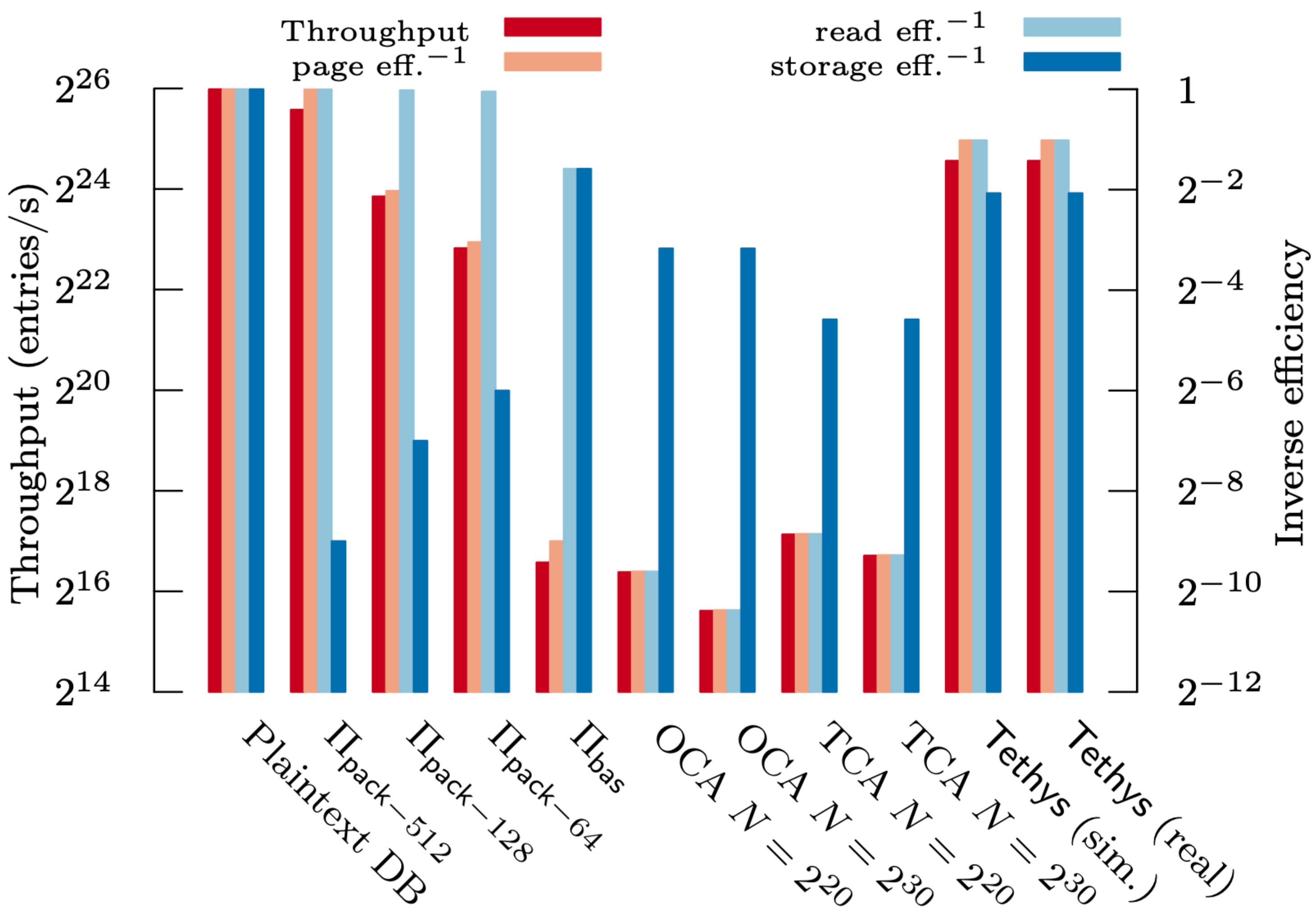
Schemes	Client st.	Page eff.	Storage eff.
Π_{bas}	$\mathcal{O}(1)$	$\mathcal{O}(p)$	$\mathcal{O}(1)$
$\Pi_{\text{pack}}, \Pi_{\text{2lev}}$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$
1-Choice	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log N)$	$\mathcal{O}(1)$
2-Choice	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log \log N)$	$\mathcal{O}(1)$
Tethys	$\mathcal{O}(p \log \lambda)$	3	$3 + \varepsilon$
Pluto	$\mathcal{O}(p \log \lambda)$	3	$3 + \varepsilon$
Nilus$_t$	$\mathcal{O}(p \log \lambda)$	$2t + 1$	$1 + (2/e)^{t-1}$

Performance evaluation

Theory

Schemes	Client st.	Page eff.	Storage eff.
Π_{bas}	$\mathcal{O}(1)$	$\mathcal{O}(p)$	$\mathcal{O}(1)$
$\Pi_{\text{pack}}, \Pi_{\text{2lev}}$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$
1-Choice	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log N)$	$\mathcal{O}(1)$
2-Choice	$\mathcal{O}(1)$	$\widetilde{\mathcal{O}}(\log \log N)$	$\mathcal{O}(1)$
Tethys	$\mathcal{O}(p \log \lambda)$	3	$3 + \varepsilon$
Pluto	$\mathcal{O}(p \log \lambda)$	3	$3 + \varepsilon$
Nilus$_t$	$\mathcal{O}(p \log \lambda)$	$2t + 1$	$1 + (2/e)^{t-1}$

Implementation



Open-source optimized implementation: https://github.com/OpenSSE/

THANKS

Questions?