Blocked Bloom Filters with Choices

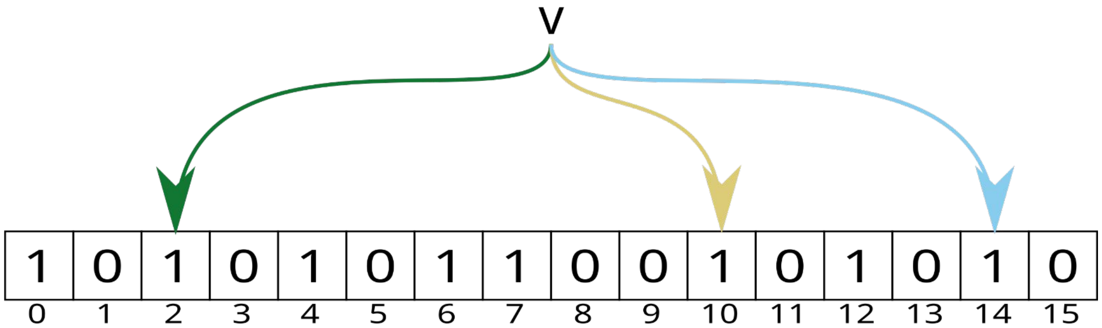
Jens Zentgraf, Johanna Elena Schmitz and Sven Rahmann
Algorithmic Bioinformatics, Saarland University

Bloom Filter

- Probabilistic set membership data structure
- A set K of $n=|K|$ elements
- k hash functions
  - false positive rate of 2^{-k}
- Bit array of size m bits
  - $m = nk / \ln(2)$

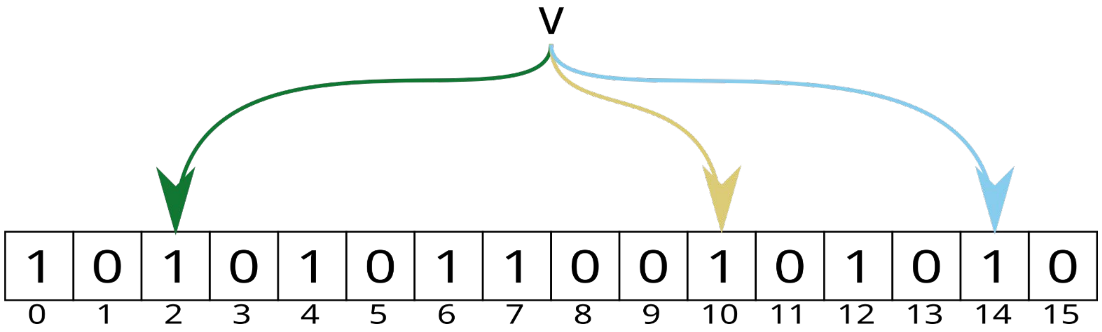
Bloom Filter

- Probabilistic set membership data structure
- A set K of n=|K| elements
- k hash functions
  - ➢ false positive rate of 2^{-k}
- Bit array of size m bits
  - ➢ $m = nk / \ln(2)$



Bloom Filter

- Probabilistic set membership data structure
- A set K of $n=|K|$ elements
- k hash functions
  - ➢ false positive rate of 2^{-k}
- Bit array of size m bits
  - ➢ $m = nk / \ln(2)$
- Insert:
  - ➢ Compute k positions
  - ➢ Set all positions to 1
- Lookup:
  - ➢ Compute k positions
  - ➢ All positions 1 ➡ contained



Bloom Filter

Advantages:

- Simple
- Adjustable FPR
  (number of hash functions)
- Online insertion

Disadvantages:

- High overhead (≈1.44)
- Slow
  - *k* cache misses

Bloom Filter

Advantages:

- Simple
- Adjustable FPR
  (number of hash functions)
- Online insertion

Disadvantages:

- High overhead (≈1.44)
- **Slow**
  - *k* cache misses

Blocked Bloom Filter

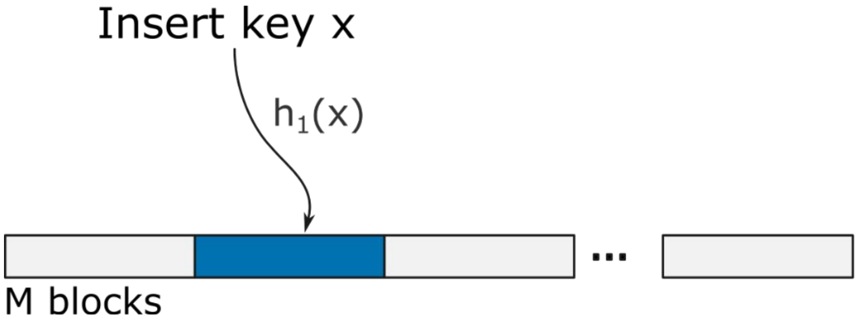
- Split the bit array into *B* blocks of size *M*
- *M* is typically a cache line (512 bits)
- 1 additional hash function to pick a block

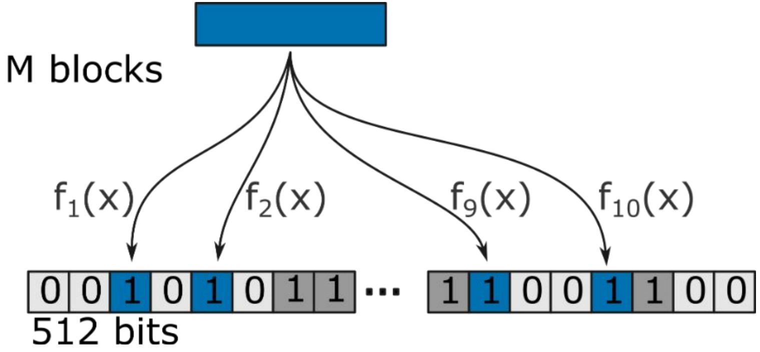
- Insert:
  - Compute 1 block
  - Compute *k* positions inside a block
  - Set all positions to 1

Blocked Bloom Filter

- Split the bit array into *B* blocks of size *M*
- *M* is typically a cache line (512 bits)
- 1 additional hash function to pick a block

- Insert:
  - ➢ Compute 1 block
  - ➢ Compute *k* positions inside a block
  - ➢ Set all positions to 1





Blocked Bloom Filter

Advantages:

- One cache miss
- Faster than the normal Bloom filter

Disadvantages:

- One additional hash function to select a block
- Blocks are not filled evenly.
  - Some blocks are more filled, some are less
  - Higher FPR
- Increase size to counter increased FPR

Blocked Bloom Filter

Advantages:

- One cache miss
- Faster than the normal Bloom filter

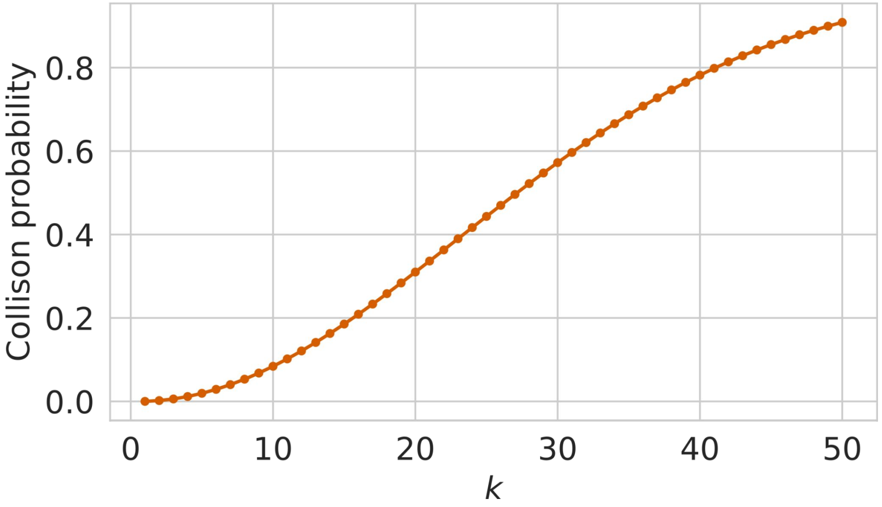
- **Goal**:
  - Reduce FPR and keep filter size
  - Reduce overhead and keep FPR

Disadvantages:

- One additional hash function to select a block
- Blocks are not filled evenly.
  - Some blocks are more filled, some are less
  - Higher FPR
- Increase size to counter increased FPR

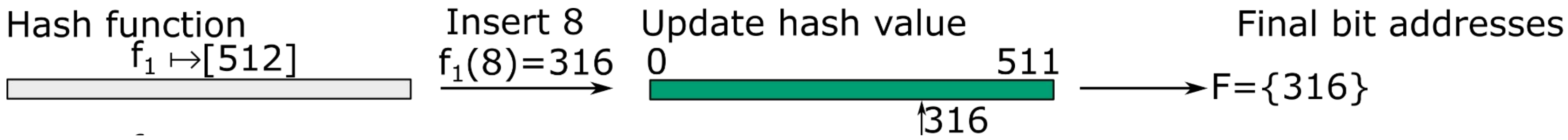
Random vs. Distinct hash positions

- One or multiple hash functions can point to the same bit positions
- We only get $k' \leq k$ different positions
- Reduces the FPR to $2^{-k'}$



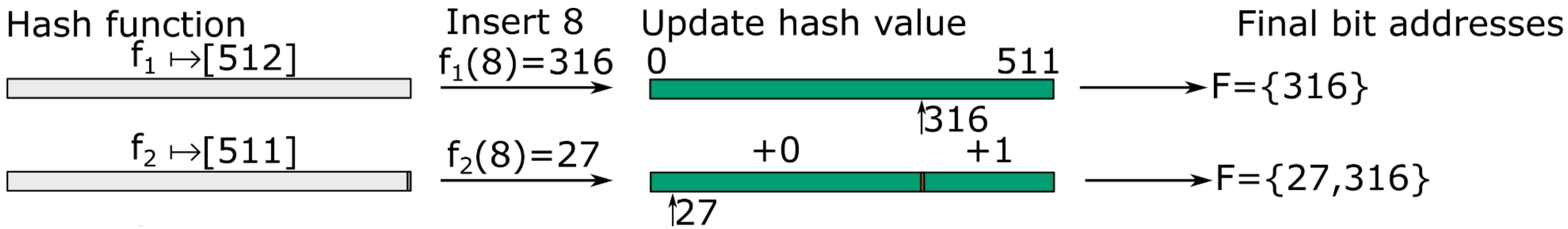
Random vs. Distinct hash positions

- One or multiple hash functions can point to the same bit positions
- We only get $k' \leq k$ different positions
- Reduces the FPR to $2^{-k'}$



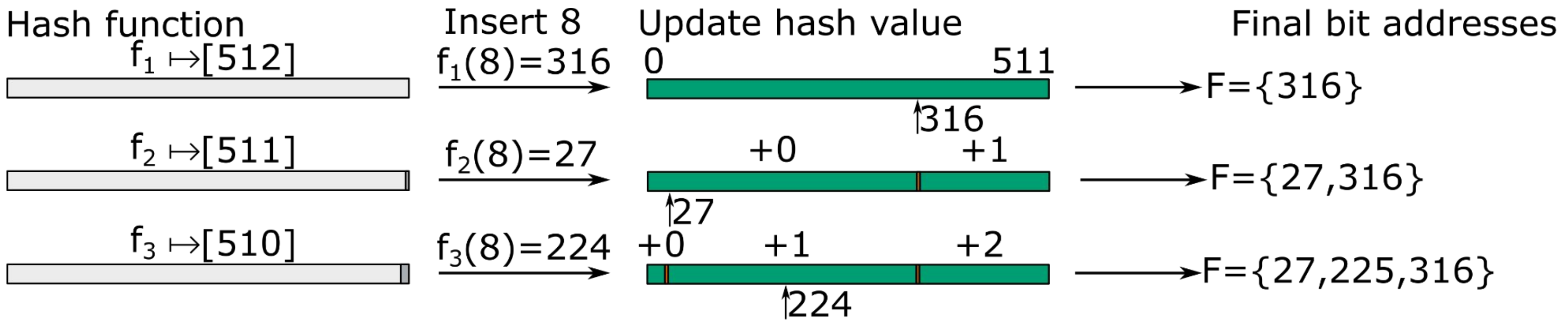
Random vs. Distinct hash positions

- One or multiple hash functions can point to the same bit positions
- We only get $k' \leq k$ different positions
- Reduces the FPR to $2^{-k'}$



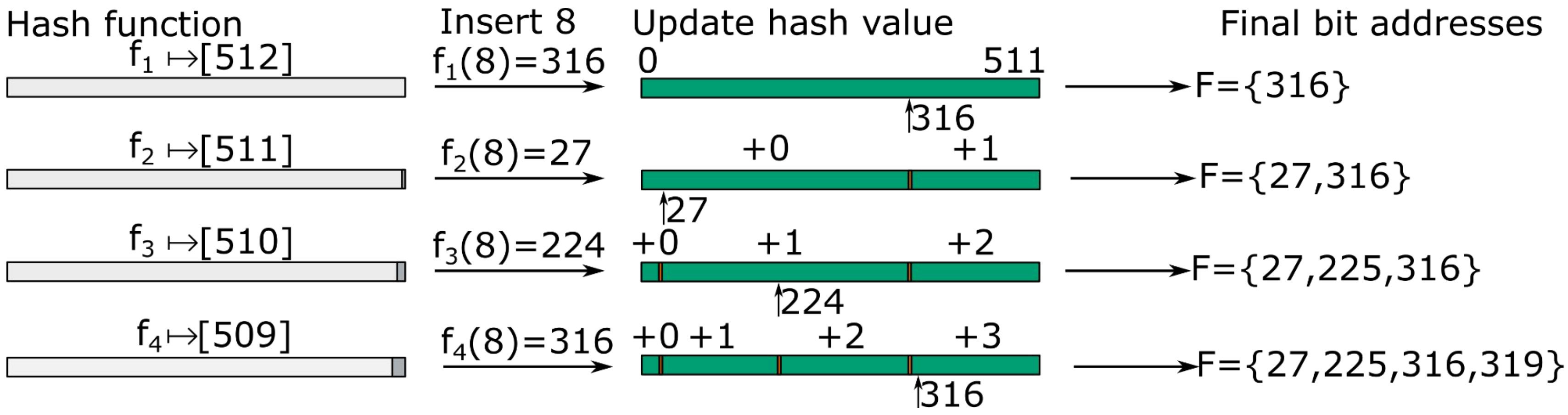
Random vs. Distinct hash positions

- One or multiple hash functions can point to the same bit positions
- We only get $k' \leq k$ different positions
- Reduces the FPR to $2^{-k'}$



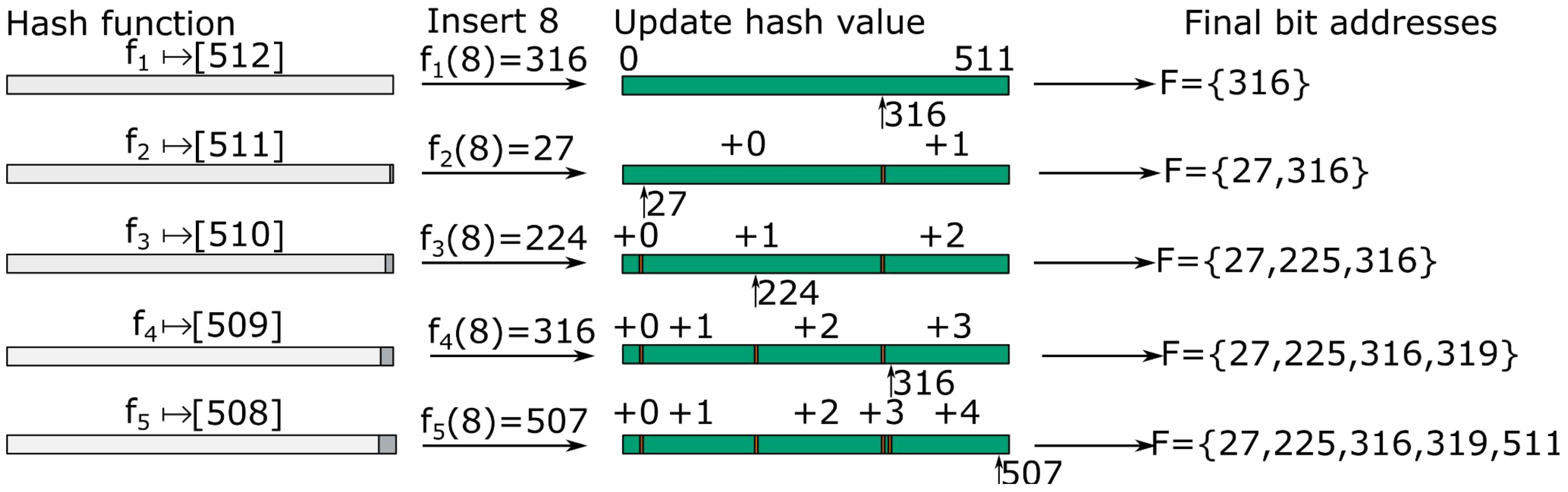
Random vs. Distinct hash positions

- One or multiple hash functions can point to the same bit positions
- We only get $k' \leq k$ different positions
- Reduces the FPR to $2^{-k'}$



Random vs. Distinct hash positions

- One or multiple hash functions can point to the same bit positions
- We only get $k' \leq k$ different positions
- Reduces the FPR to $2^{-k'}$



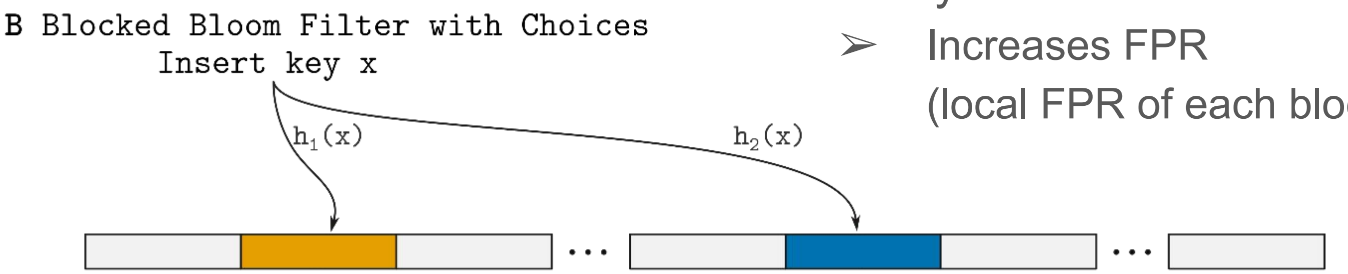
Blocked Bloom filters with choices

- Instead of computing one block, we can choose one block out of *c* possible blocks.

- Keep local FPR low in each block
  - ➢ Pick the block with the lower FPR
- Always check *c* blocks.
  - ➢ Increases FPR (local FPR of each block)

Blocked Bloom filters with choices

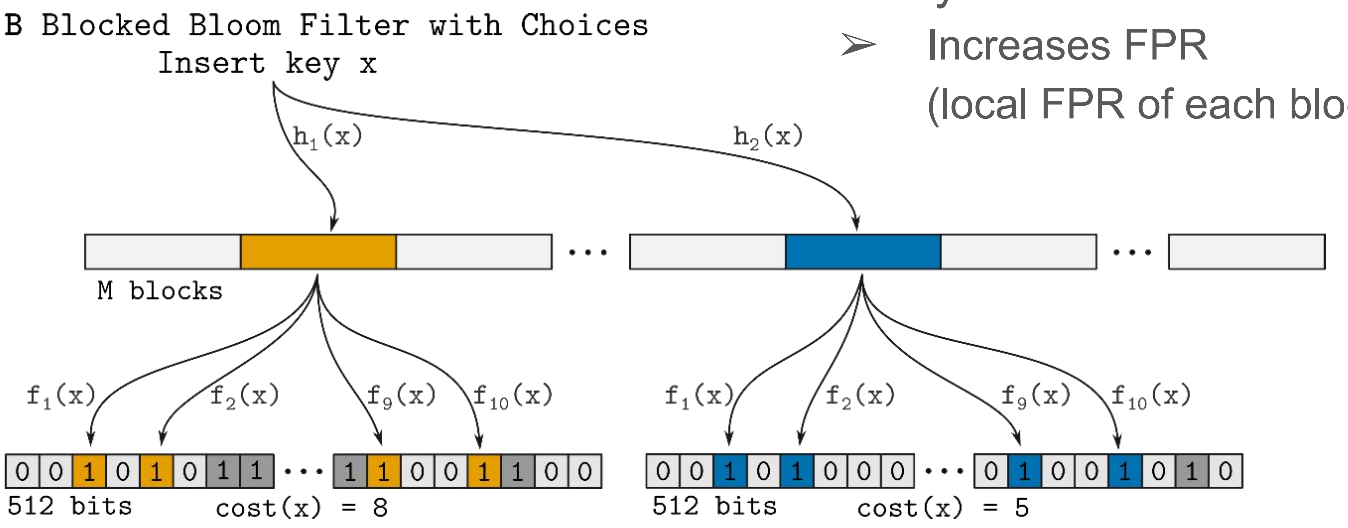
- Instead of computing one block, we can choose one block out of *c* possible blocks.



- Keep local FPR low in each block
  - ➢ Pick the block with the lower FPR
- Always check *c* blocks.
  - ➢ Increases FPR (local FPR of each block)

Blocked Bloom filters with choices

- Instead of computing one block, we can choose one block out of *c* possible blocks.

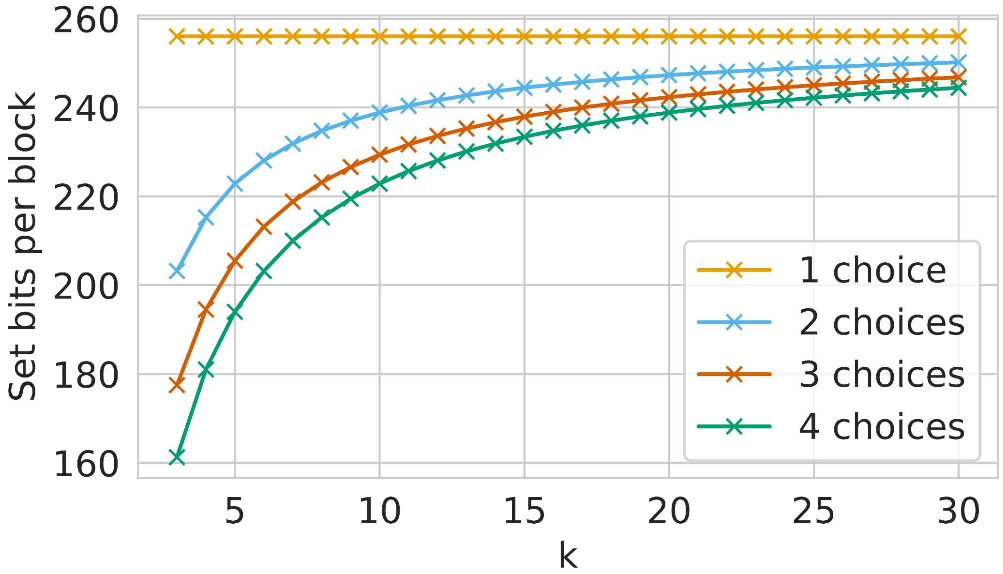


- Keep local FPR low in each block
  - ➢ Pick the block with the lower FPR
- Always check *c* blocks.
  - ➢ Increases FPR (local FPR of each block)

Blocked Bloom filters with choices

- Instead of computing one block, we can choose one block out of *c*.

- Keep local FPR low in each block
  - Pick the block with the lower FPR
- Always check *c* blocks.
  - Increases FPR (local FPR of each block)



Cost functions

The cost functions are based on two parameters:

- *j* number of set bits after insertion
- *a* number of new set bits after insertion

Goal:

- Reduce local FPR in blocks
- Reuse bits if possible

Cost functions

The cost functions are based on two parameters:

- j number of set bits after insertion
- a number of new set bits after insertion

Goal:

- Reduce local FPR in blocks
- Reuse bits if possible

- $k = 10$
- $2^{-k} = 2^{-10} \approx 0.0009765625$

k=10	random		distinct	
choices	set bits (j)	new bits (a)	set bits (j)	new bits (a)
2	0,001634	0,008066	0,001587	0,008201
3	0,001957	0,034652	0,001893	0,035655

Cost functions

The cost functions are based on two parameters:

- *j* number of set bits after insertion
- *a* number of new set bits after insertion

Goal:

- Reduce local FPR in blocks
- Reuse bits if possible

Three different cost functions:

- Mixed cost function
- Lookahead cost function
- Exponential cost function

Mixed Cost function

- Keep number of set bits in a block low
- Reuse bits if possible

Mixed Cost function

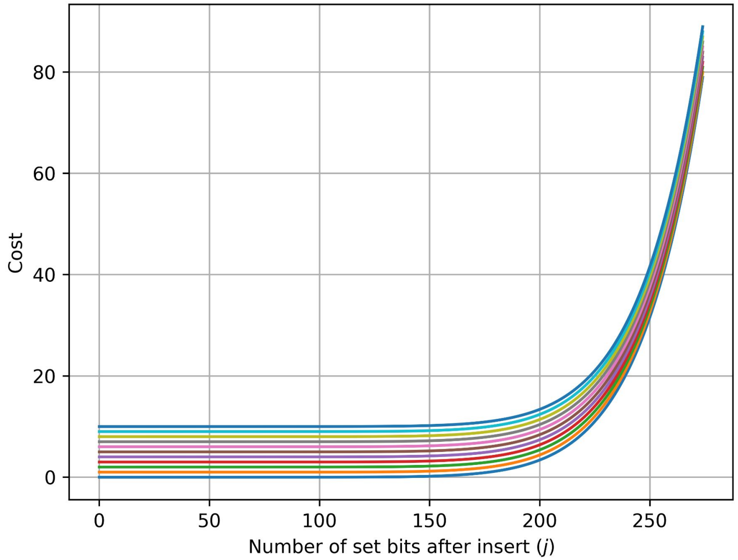
- Keep number of set bits in a block low
- Reuse bits if possible

- $\text{cost}^{\text{MIX}}_{\sigma}(j, a) := \sigma k \cdot (j/256)^{k} + a$

Mixed Cost function

- Keep number of set bits in a block low
- Reuse bits if possible

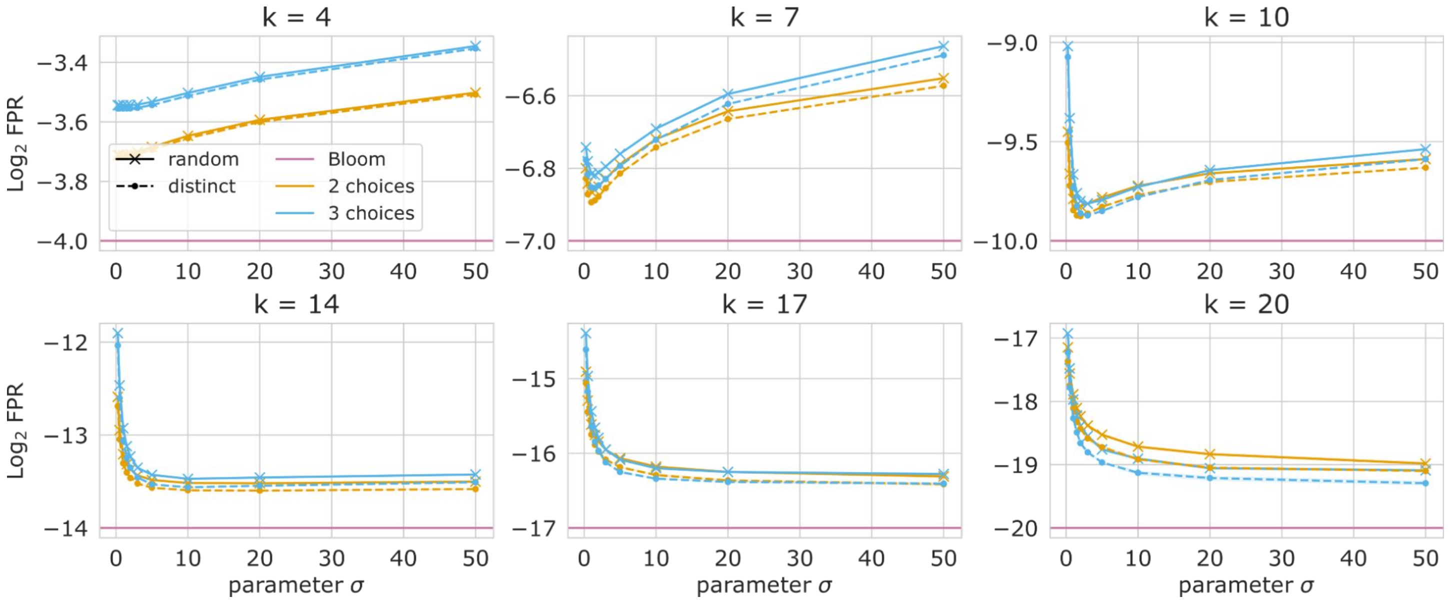
- $cost^{MIX}_{\sigma}(j, a) := \sigma k \cdot (j/256)^{k} + a$



Mixed Cost function

- Keep number of set bits in a block low
- Reuse bits if possible

- $\text{cost}^{\text{MIX}}_{\sigma}(j, a) := \sigma k \cdot (j/256)^k + a$



Lookahead Cost function

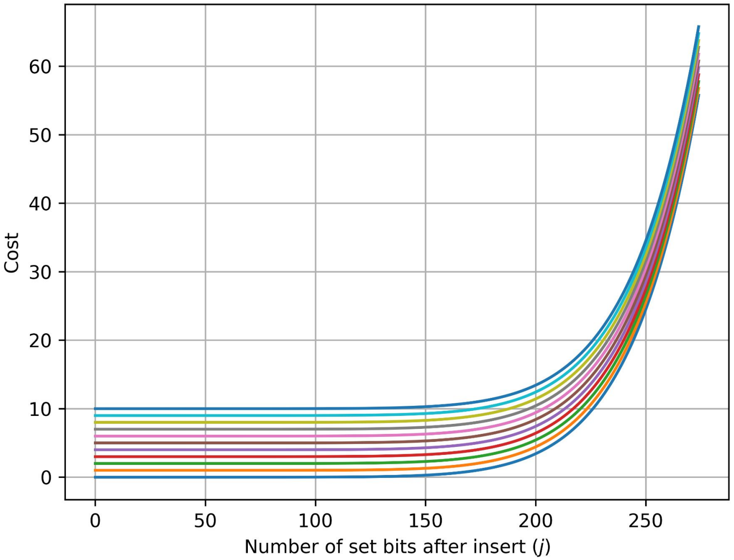
- Still a lot of overfull blocks
- Penalize already less full blocks stronger

$$\mathrm{cost}^{LA}_{\mu}(j, a) := \mathrm{cost}^{MIX}_{1}(j + \mu k, a)$$
$$= k \cdot ((j + \mu k)/256)^{k} + a$$

Lookahead Cost function

- Still a lot of overfull blocks
- Penalize already less full blocks stronger

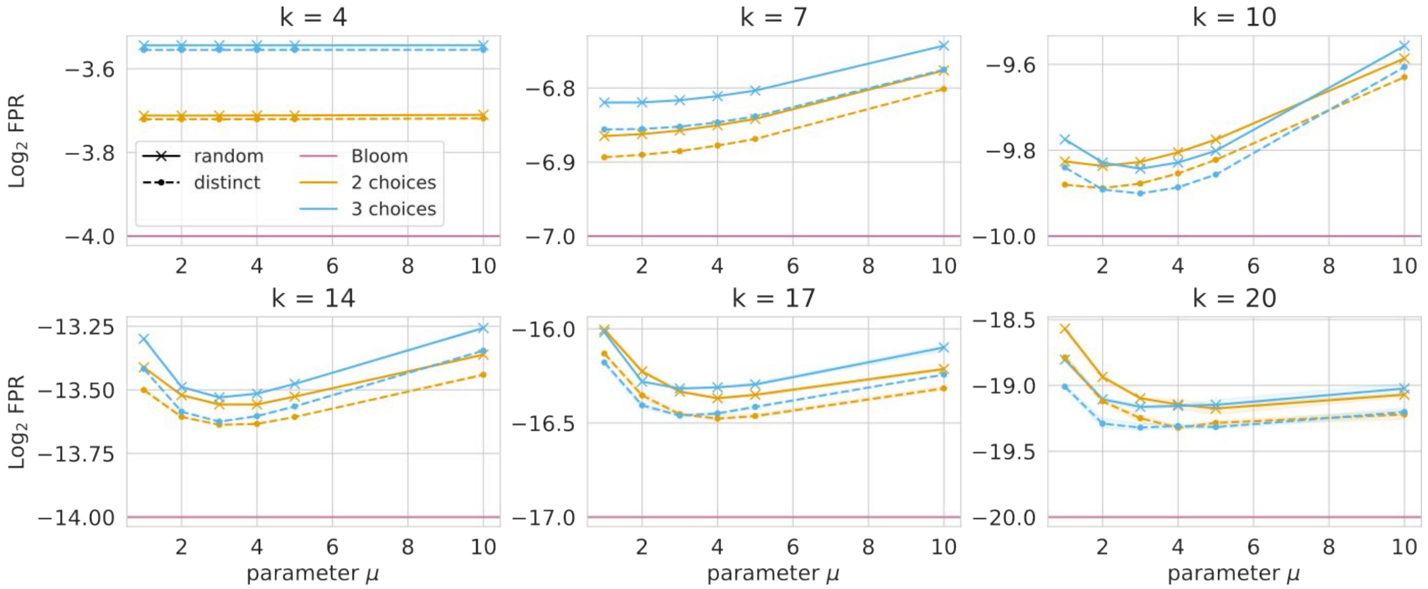
$$\mathrm{cost}^{LA}_{\mu}(j, a) := \mathrm{cost}^{MIX}_{1}(j + \mu k, a) = k \cdot ((j + \mu k)/256)^k + a$$



Lookahead Cost function

- Still a lot of overfull blocks
- Penalize already less full blocks stronger

$$\text{cost}^{LA}_{\mu}(j, a) := \text{cost}^{MIX}_{1}(j + \mu k, a) = k \cdot ((j + \mu k)/256)^k + a$$



Exponential Cost function

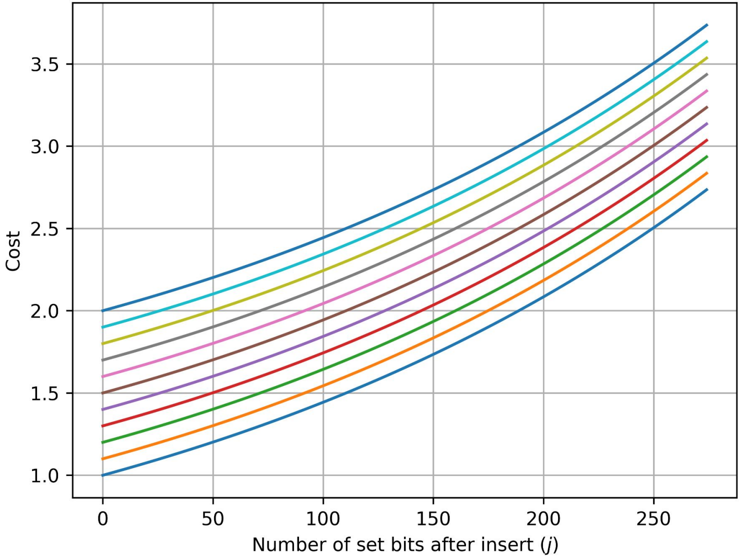
- Reduce the risk of overfilling a bucket further

$$\text{cost}^{\text{EXP}}_{\beta}(j, a) := \beta^{(j/128)} + a/k$$

Exponential Cost function

- Reduce the risk of overfilling a bucket further

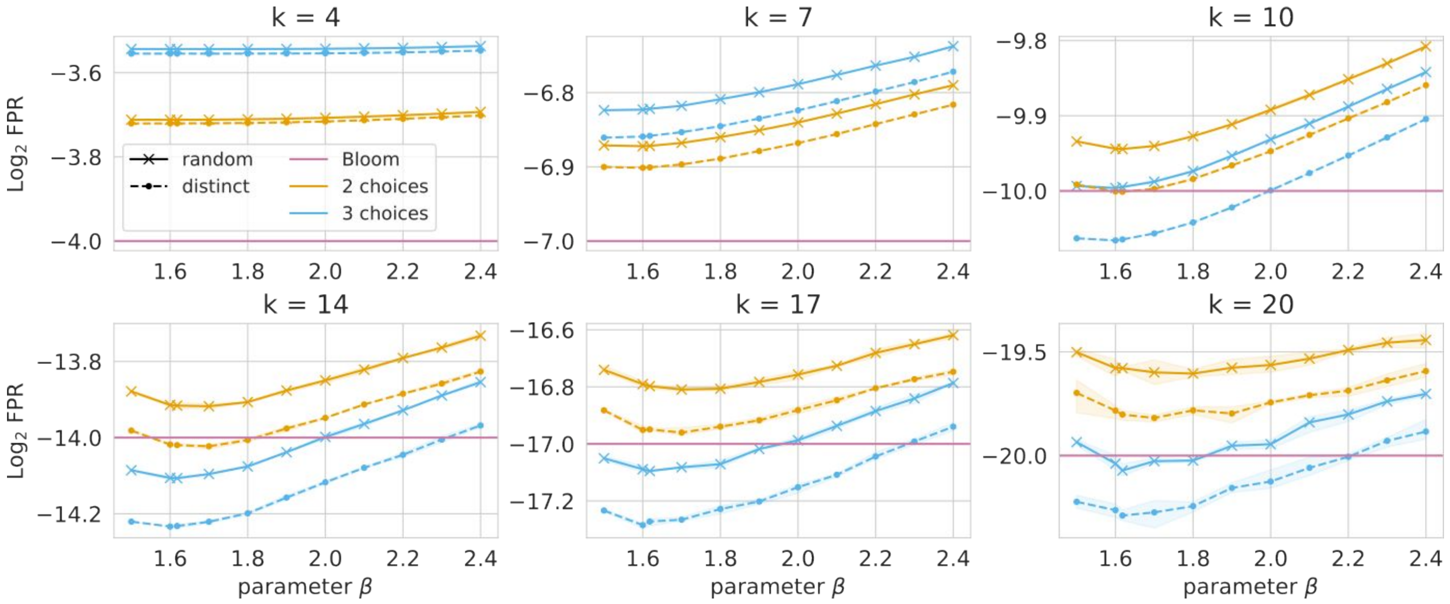
$$\text{cost}^{\text{EXP}}_{\beta}(j, a) := \beta^{(j/128)} + a/k$$



Exponential Cost function

- Reduce the risk of overfilling a bucket further

$$\text{cost}^{\text{EXP}}_{\beta}(j, a) := \beta^{(j/128)} + a/k$$



Linear Cost function

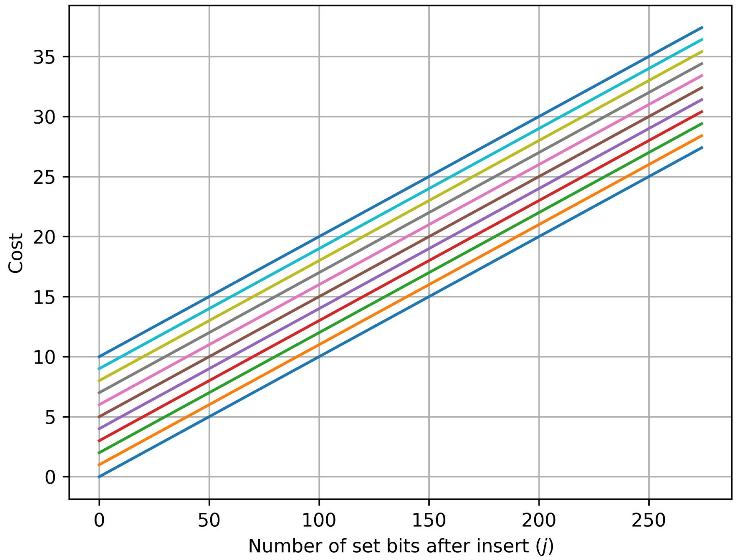
- Perhaps a linear function works best?

$$\text{cost}^{LINEAR}{}_{m}(j, a) := mj + a$$

Linear Cost function

- Perhaps a linear function works best?

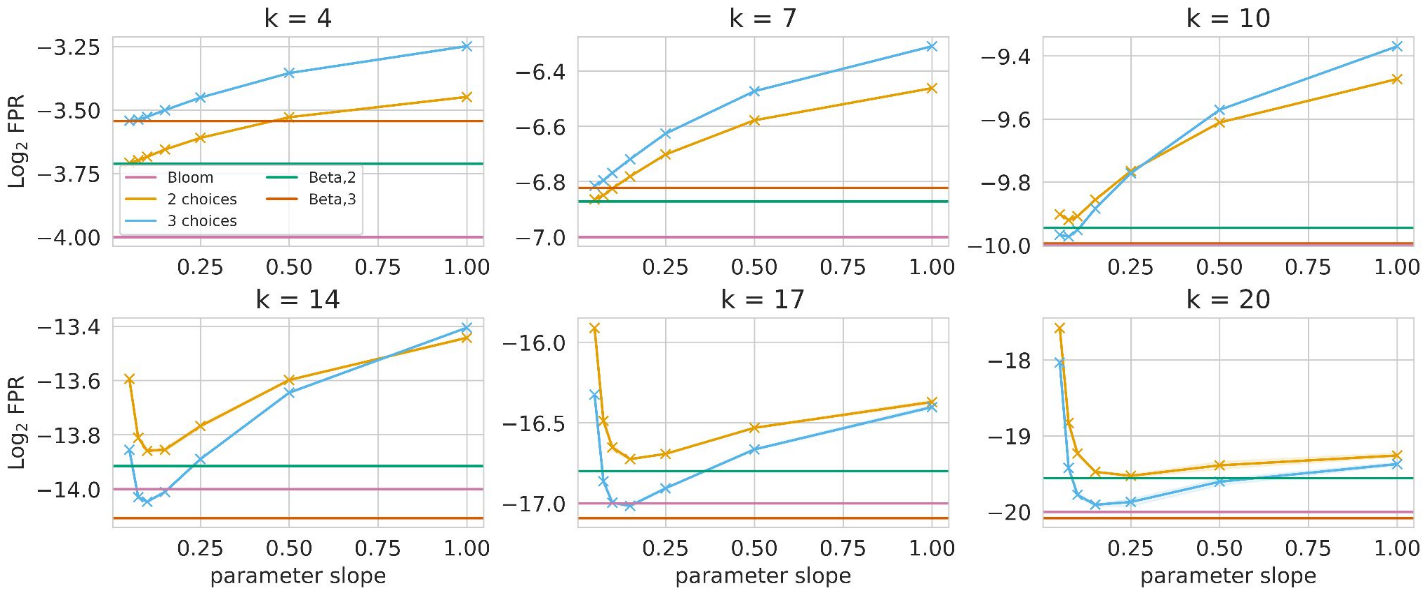
$$\text{cost}^{LINEAR}_{m}(j, a) := mj + a$$



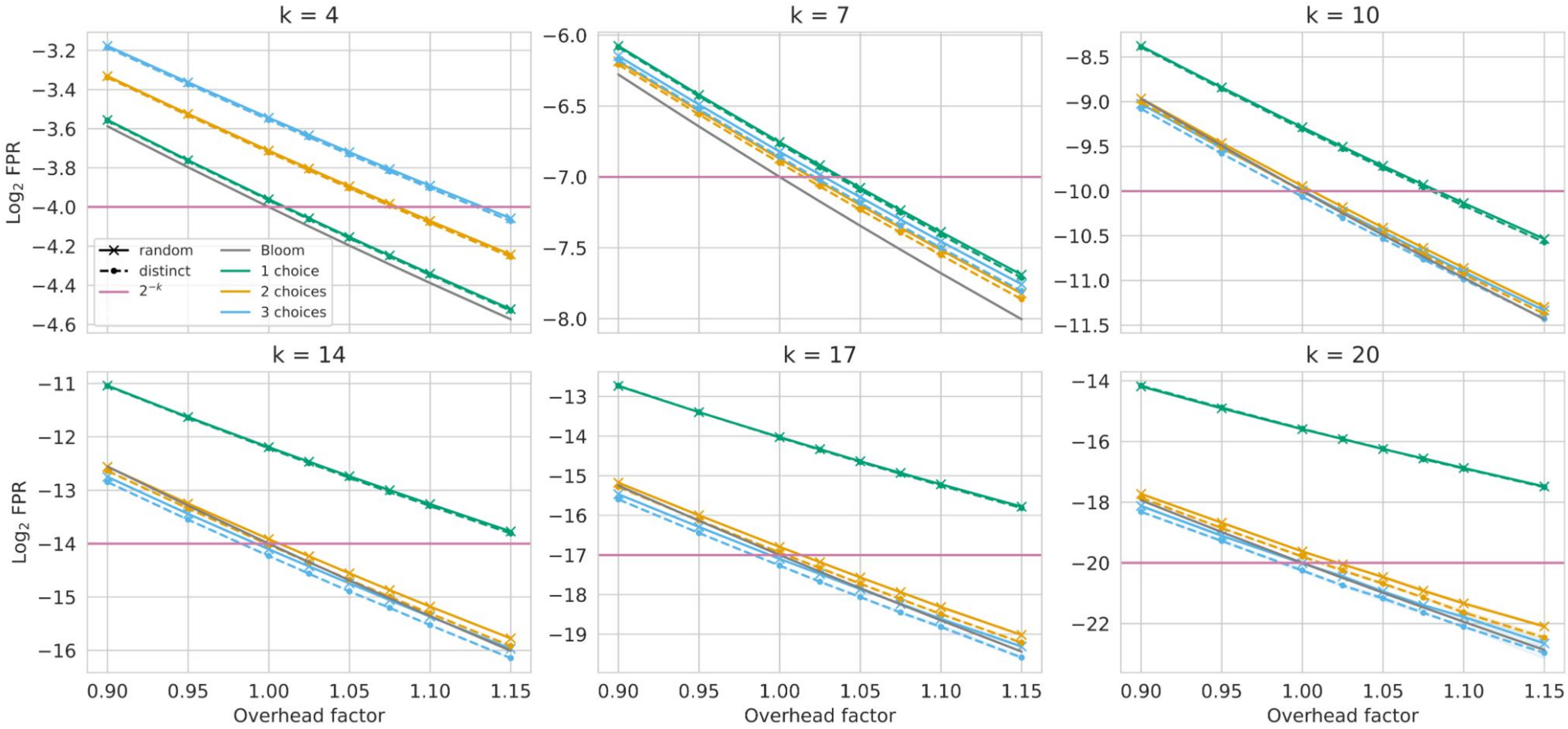
Linear Cost function

- Perhaps a linear function works best?

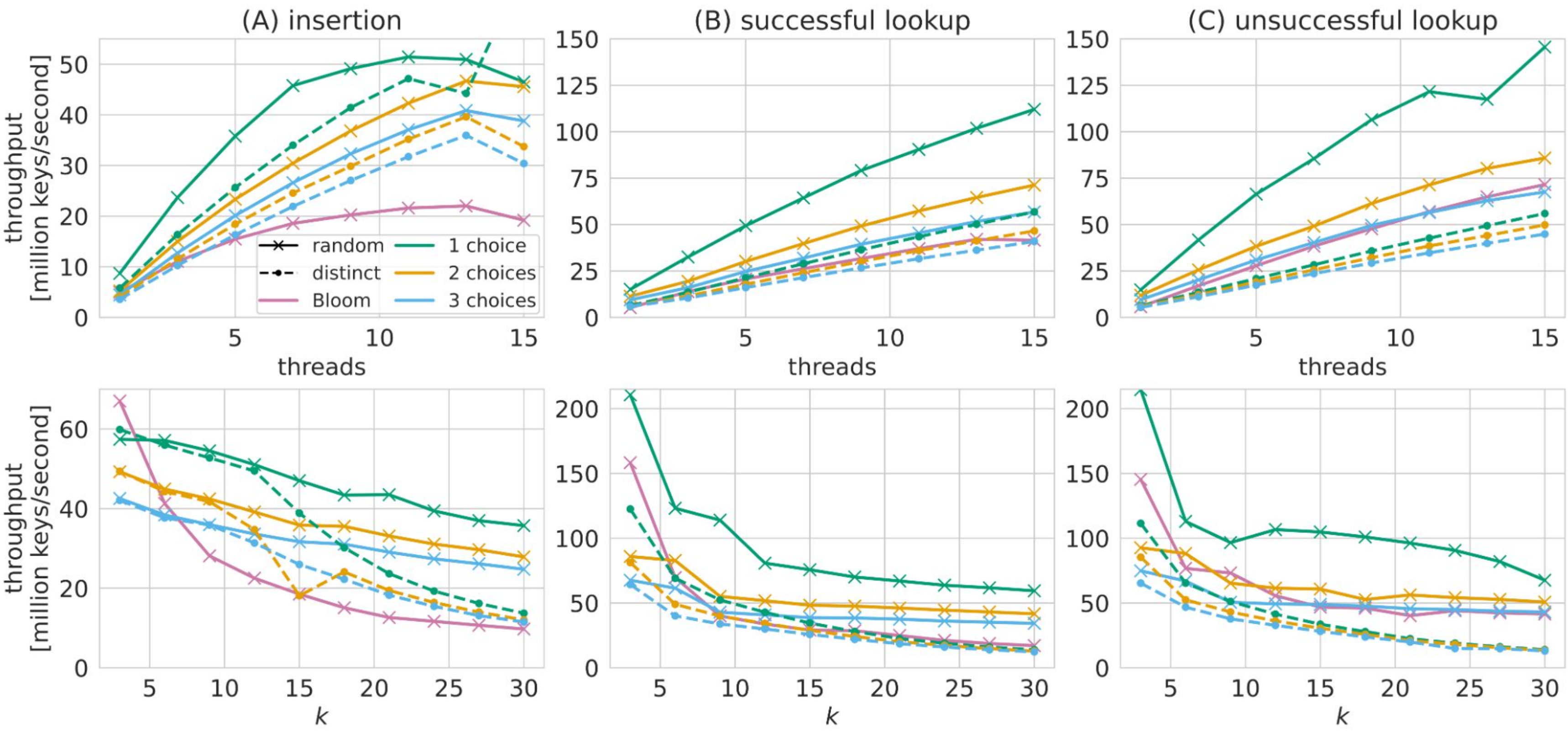
$$cost^{LINEAR}_m(j, a) := mj + a$$



Overhead (Exp. Cost function)



Running times (Exp. Cost function)



Summary

Blocked Bloom filters with choices:

- Same space overhead as normal Bloom filters
- Better FPR than Blocked Bloom filters.
- Better FPR than normal Bloom filters using exponential cost function.