Key-value stores
Redis

http://redis.io

The Key-value Abstraction

- (Business) Key → Value
- (twitter.com) tweet id → information about tweet
- (amazon.com) item number → information about it
- (kayak.com) Flight number → information about flight, e.g., availability
- (yourbank.com) Account number → information about it

The Key-value Abstraction (2)

- It's a dictionary data structure.
  - Insert, lookup, and delete by key
  - E.g., hash table, binary tree
- But distributed.
- Sound familiar?
  - Distributed Hash tables (DHT) in P2P systems

- It's not surprising that key-value stores reuse many techniques from DHTs.

Key Value Stores

- Key-Valued data model
  - Key is the unique identifier
  - Key is the granularity for consistent access
  - Value can be structured or unstructured
- Gained widespread popularity
  - In house: **Bigtable** (Google), **PNUTS** (Yahoo!), **Dynamo** (Amazon)
  - Open source: **Redis**, **HBase, Hypertable, Cassandra, Voldemort**
- Popular choice for the modern breed of web-applications

Important Design Goals

- **Scale out: designed for scale**
  - Commodity hardware
  - Low latency updates
  - Sustain high update/insert throughput
- **Elasticity – scale up and down with load**
- **High availability – downtime implies lost revenue**
  - Replication (with multi-mastering)
  - Geographic replication
  - Automated failure recovery

Lower Priorities

- **No Complex querying functionality**
  - No support for SQL
  - CRUD operations through database specific API
- **No support for joins**
  - Materialize simple join results in the relevant row
  - Give up normalization of data?
- **No support for transactions**
  - Most data stores support single row transactions
  - Tunable consistency and availability
- ***Avoid scalability bottlenecks at large scale***

System Interface

- Two basic operations:
  - Get(key):
  - Put(key, value)

Redis

- Redis is an open source, advanced **key-value data store**
- Often referred to as a **data structure server** since keys can contain strings, hashes, lists, sets and sorted sets
- The name Redis means Remote Dictionary Server
- Redis works with an **in-memory** dataset
- It is possible to **persist** dataset either by
  - dumping the dataset to disk every once in a while
  - or by appending each command to a log

Who is using Redis?

- Twitter
- GitHub
- Weibo
- Pinterest
- Snapchat
- Craigslist
- Digg
- StackOverflow
- Flickr

Configuration

- Configuration file: **`/redis/redis.conf`**
- It is possible to change a port (if you wish):

```
port 6379
```

- For development environment it is useful to change data persisting policy

```
save 900 1
save 300 10
save 60 10000
```

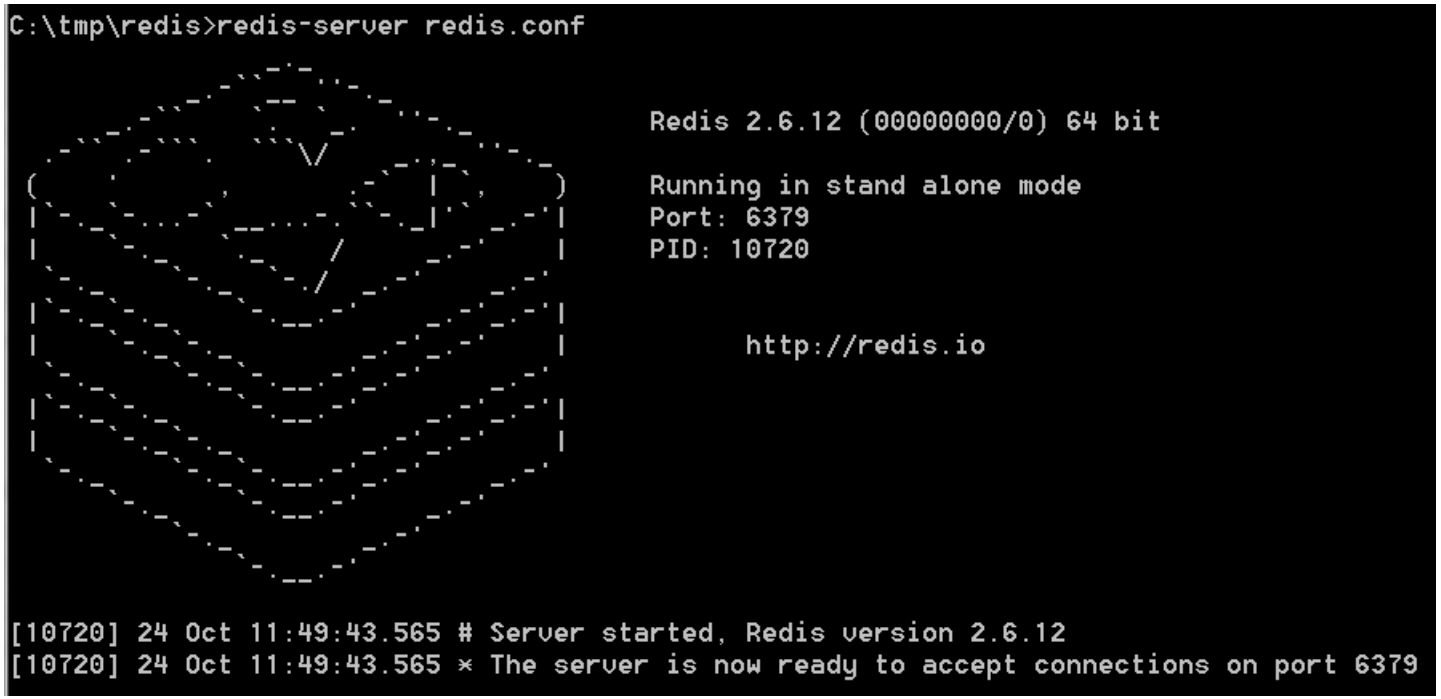
```
save 10 1
```

save after 10 sec if at least 1 key changed

Running Redis Server

- Run **`/redis/bin/redis-server.exe`** and specify configuration file to use

```
redis>redis-server redis.conf
```



Running Redis Client

- Run **`redis-cli.exe`**
- Now we can play with Redis a little bit

```
C:\tmp\redis>redis-cli
redis 127.0.0.1:6379> SET MyVar 10
OK
redis 127.0.0.1:6379> GET MyVar
"10"
redis 127.0.0.1:6379> INCR MyVar
(integer) 11
redis 127.0.0.1:6379> INCRBY MyVar 10
(integer) 21
```

Useful Commands

- Print all keys:

```
KEYS *
```

- Remove all keys from all databases

```
FLUSHALL
```

- Synchronously save the dataset to disk

```
SAVE
```

Redis keys

- Keys are binary safe - it is possible to use any binary sequence as a key
- The empty string is also a valid key
- Too long keys are not a good idea
- Too short keys are often also not a good idea ("`u:1000:pwd`" versus "`user:1000:password`")
- Nice idea is to use some kind of schema, like: "`object-type:id:field`"

Redis data types

Redis is often referred to as a **data structure server** since keys can contain:

- Strings
- Lists
- Sets
- Hashes
- Sorted Sets

Redis Strings

- Most basic kind of Redis value
- Binary safe - can contain any kind of data, for instance a JPEG image or a serialized Ruby object
- Max 512 Megabytes in length
- Can be used as atomic counters using commands in the INCR family
- Can be appended with the APPEND command

Redis Strings: Example

```
redis 127.0.0.1:6379> SET COUNTER 10
OK
redis 127.0.0.1:6379> INCRBY COUNTER 100
(integer) 110
redis 127.0.0.1:6379> DECR COUNTER
(integer) 109
redis 127.0.0.1:6379> APPEND COUNTER 01
(integer) 5
redis 127.0.0.1:6379> GET COUNTER
"10901"
redis 127.0.0.1:6379> INCR COUNTER
(integer) 10902
```

Transactions

- Redis's MULTI block atomic commands are a similar concept to transactions. Wrapping two operations like SET and INCR in a single block will complete either successfully or not at all.
- We begin the transaction with the MULTI command and execute it with EXEC (rollback with DISCARD).

```
redis 127.0.0.1:6379> MULTI
redis 127.0.0.1:6379> SET foo bar
redis 127.0.0.1:6379> INCR counter
redis 127.0.0.1:6379> EXEC
```

Redis Hashes

- Redis objects that can take any number of key-value pairs
- Map between string fields and string values
- Perfect data type to represent objects

```
HMSET user:1000 username gomez password P1pp0 age 34
HGETALL user:1000
HVALS user:1000
HKEYS user:1000
HSET user:1000 password 12345
HGETALL user:1000
HGET user:1000 username
```

Redis Lists

- Lists of ordered values (insertion order)
- Can act as queues or stacks (or just lists)
- Add elements to a Redis List pushing new elements on the head (on the left) or on the tail (on the right) of the list
- Max length: (2^32 - 1) elements
- Model a timeline in a social network, using LPUSH to add new elements, and using LRANGE in order to retrieve recent items
- Use LPUSH together with LTRIM to create a list that never exceeds a given number of elements

Redis Lists: Example

```
redis 127.0.0.1:6379> LPUSH myList a
(integer) 1
redis 127.0.0.1:6379> LPUSH myList b
(integer) 2
redis 127.0.0.1:6379> LPUSH myList c
(integer) 3
redis 127.0.0.1:6379> LLEN myList
(integer) 3
redis 127.0.0.1:6379> LRANGE myList 0 -1
1) "c"
2) "b"
3) "a"
redis 127.0.0.1:6379> RPUSH myList d e f
(integer) 6
redis 127.0.0.1:6379> LRANGE myList 0 -1
1) "c"
2) "b"
3) "a"
4) "d"
5) "e"
6) "f"
redis 127.0.0.1:6379> LTRIM myList 2 4
OK
redis 127.0.0.1:6379> LRANGE myList 0 -1
1) "a"
2) "d"
3) "e"
```

More list functions

- LREM removes from the list given value

  LREM myList 0 a
- LPOP removes from the left (head) of the list

  LPOP myList
- RPUSH/RPOP add/remove from the right of the list
- RPOPLPUSH pop a value from the tail of one list and push it to the head of another

  RPOPLPUSH myList yourList

Blocking lists

- Producer-consumer example.
- Open another redis client, one client (the consumer) just listens for new comments and pop them as they arrive.

  BRPOP comments 300
- The command will block until a value exists to pop. Timeout in seconds is set to five minutes.
- Now the producer should push a message to comments.

  LPUSH comments “ModernDB is a great class!"
- Switch back to the consumer console, two lines will be returned: the key and the popped value. The console will also output the length of time it spent blocking.

Sets

- Unordered collections with no duplicate values, supports unions and intersections

```
SADD myPref movies reading walking
```

- SMEMBERS retrieves the whole set.

```
SMEMBERS myPref
SADD yourPref running painting reading fishing
```

- To find the intersection use the SINTER command.

```
SINTER myPref yourPref
```

- Remove any matching values in one set from another:

```
SDIFF myPref yourPref
```

- Union is a set, any duplicates are dropped.

```
SUNION myPref yourPref
```

- That set of values can also be stored directly into a new set:

```
SUNIONSTORE hobbies myPref yourPref
```

Redis Sorted Sets

- Every member of a Sorted Set is associated with score, that is used in order to take the sorted set ordered, from the smallest to the greatest score

```
ZADD scoreboard 500 me 9 you 15 him
```

- To increment a score, we can either re-add it with the new score, which just updates the score but does not add a new value, or increment by some number, which will return the new value.

```
ZINCRBY scoreboard 1 you
```

Sorted Set Ranges

- To get scores from the sorted set:

```
ZRANGE scoreboard 0 1
```

- To get scores append WITHSCORES
- ZREVRANGE gets them in reverse
- ZRANGEBYSCORE allow to provide score range (inclusive by default)

```
ZRANGEBYSCORE scoreboard 100 500
ZRANGEBYSCORE scoreboard (100 500
ZRANGEBYSCORE scoreboard (100 inf
```

Sorted Set Unions

```
ZUNIONSTORE destination numkeys key [key ...]
    [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]
```

- destination is the key to store into
- numkeys is simply the number of keys you're about to join
- key is one or more keys to union
- weight [optional] is the number to multiply each score of the relative key by (if you have two keys, you can have two weights, and so on).
- aggregate is the optional rule, sum is default

For today

- Add a sorted set called scoreboard with the values 50 me, 30 you, 15 her, and 10 him
- Add the gamesWon sorted set for three players: you (3), me (4), and him (8)
- Multiply by 10 the number of wins as the points for each player and add them to the scoreboard (can be done in one or more commands)
- Retrieve the scoreboard in reverse order
- Submit your commands and outputs to ICON

Advanced usage and Distribution Redis

http://redis.io

Expiry

- EXPIRE command, an existing key, and a time to live (in seconds).
- Set ice to expire after 10 seconds
- SET ice "I'm melting..."
- EXPIRE ice 10

- EXISTS ice
- Wait 10 secs
- EXISTS ice

- Shortcut command:
- SETEX ice 10 "I'm melting...“
- Check time to live:
- TTL ice
- Make it persistent
- PERSIST ice

Database Namespaces

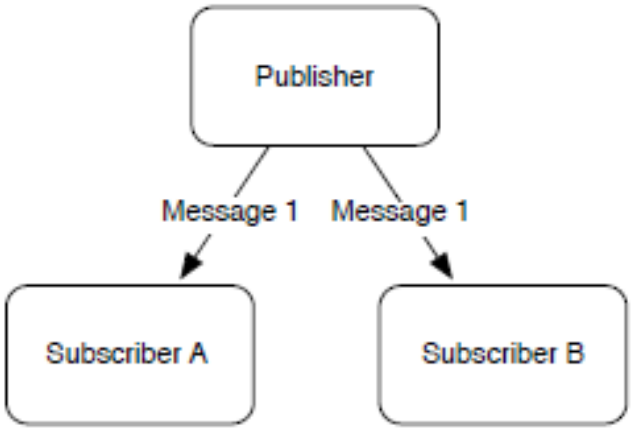
- In Redis a namespace is called a *database* and is keyed by number
- So far we've always interacted with the default database (namespace 0)
- SET greeting hello
- GET greeting

- SELECT 1
- GET greeting
- SET greeting hola

- SELECT 0
- GET greeting
- Since all databases are running in the same server instance, you can shuffle keys around using MOVE
- MOVE greeting 2
- SELECT 2
- GET greeting

Publish-subscribe

- Multiple subscribers



- Start two clients:
- SUBSCRIBE comments

- Start publisher:
- PUBLISH comment “Thanksgiving is next week!”

- UNSUBSCRIBE disconnects client – in redis-cli console press Ctrl+C to break the connection

Redis configuration

- Server info: INFO
- daemonize no – starts in the foreground
- port 6379
- loglevel verbose | notice | warning
- logfile stdout /*filename req if daemonize mode*/
- database 16

Redis configuration (cont.)

- save 300 1 (snapshotting, save every 5 mins if any keys change at all)
- appendonly yes (keeps record of all write commands)
  - appendfsync always | everysec | no

Security

- Redis is not natively built to be a fully secure server
- Plaintext password not really safe
- Use SSH security
- Redis allows you hide or suppress commands
  - Include rename-command in the conf file:
  - rename-command FLUSHALL c123456789
  - rename-command FLUSHALL “”

Master-slave replication

- First make a copy of the redis.conf file
  - cp redis.conf redis-s1.conf
  - Change port and slaveof
    - port 6380
    - Slaveof 127.0.0.1 6379
  - Start both servers
    - Redis-server redis-s1.conf
  - Add to the server
    - SADD meetings “Initial group meeting” “ECE potluck”
  - Query in the slave
    - SMEMBERS meetings

Redis cluster

- Many Redis clients provide an interface for building a simple ad hoc distributed Redis cluster.
- Unlike the master-slave setup, both of servers take the master (default) configuration.

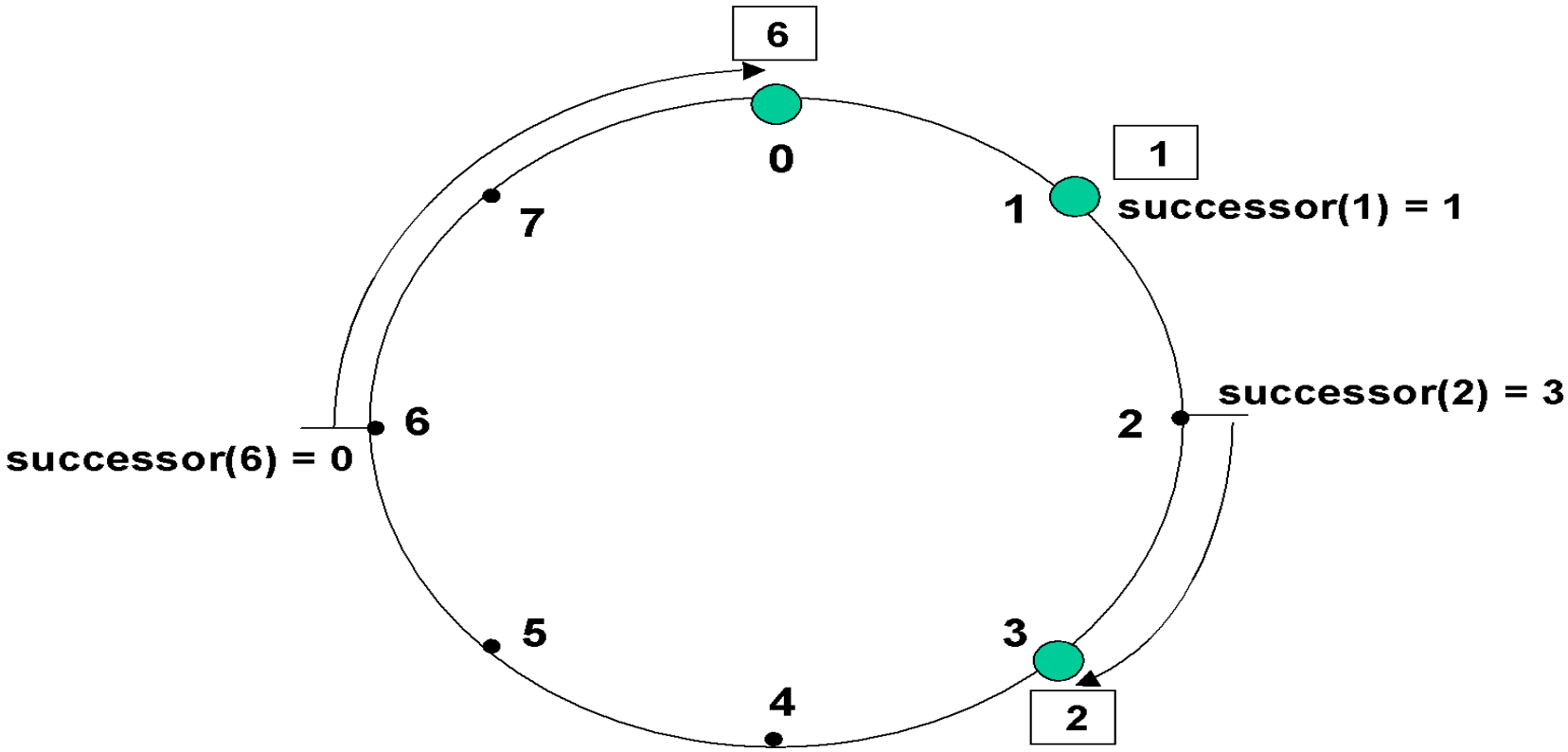
```
redis = Redis::Distributed.new([
"redis://localhost:6379/",
"redis://localhost:6380/" ])
```

- Consistent hashing is used to manage the cluster

Consistent Hashing System

- Given *k,* every node can locate n_k
- Hash every node's IP address
  - map these values on a circle
- Given a key *k*, hash k
  - k is assigned to closest node on circle, moving clockwise.

Consistent Hashing System



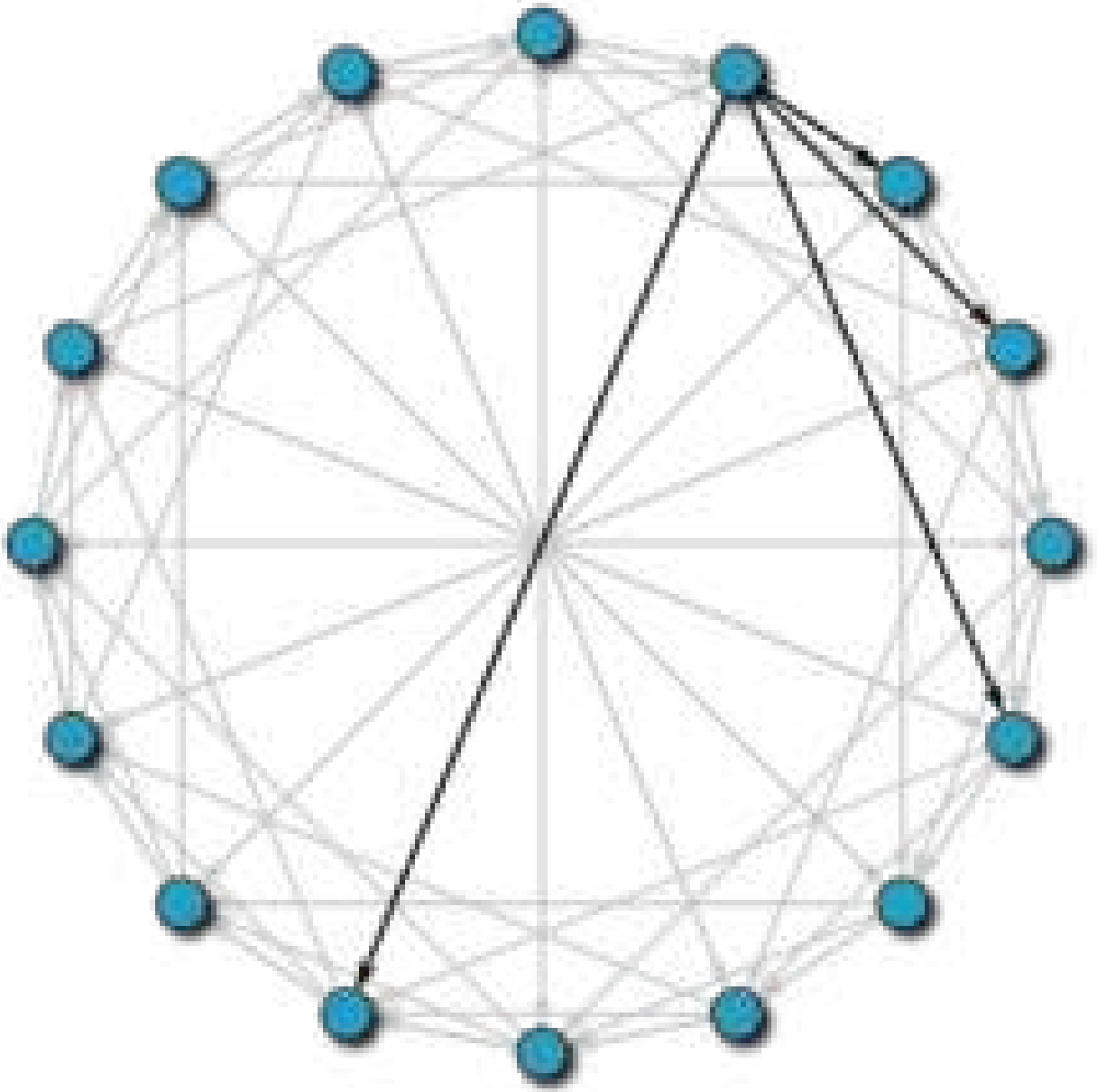
Consistent Hashing System

- Pros:
  - Load Balanced
  - Dynamic membership
    - when N^{th} node joins network, only O(1/N) keys are moved to rebalance
- Con:
  - Every node must know about every other node
  - O(N) memory, O(1) communication
    - Not scalable in number of nodes

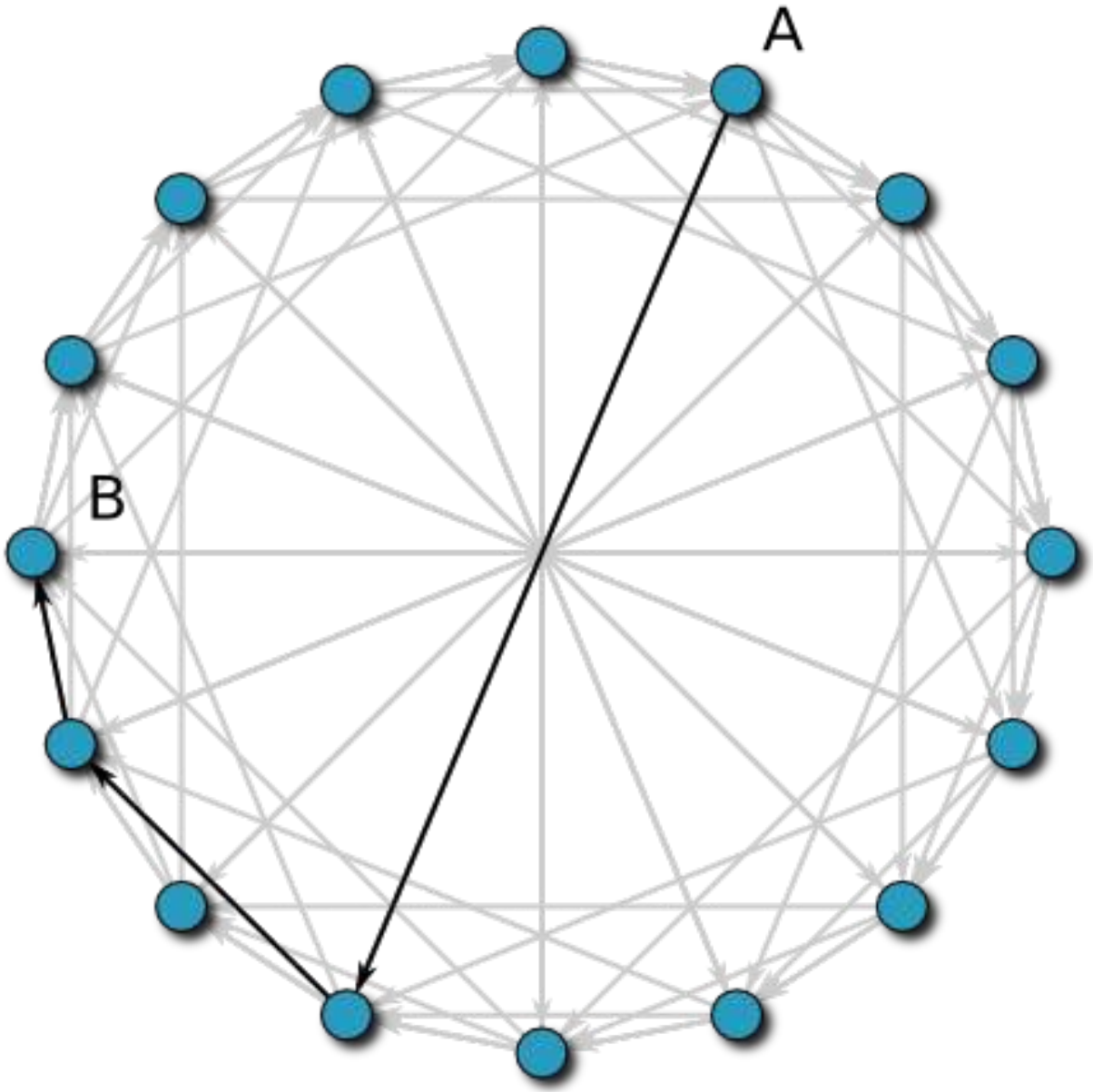
Scaling Consistent Hashing

- Approach 0:
  - Each node keeps track of only their successor
  - Resolution of hash function done through routing
  - O(1) memory
  - O(N) communication
- Approach 1:
  - Each node keeps track of O(log N) successors in a “finger table”
  - O(log N) memory
  - O(log N) communication

Finger Table Pointers



Routing with Finger Tables



Incremental Scalability

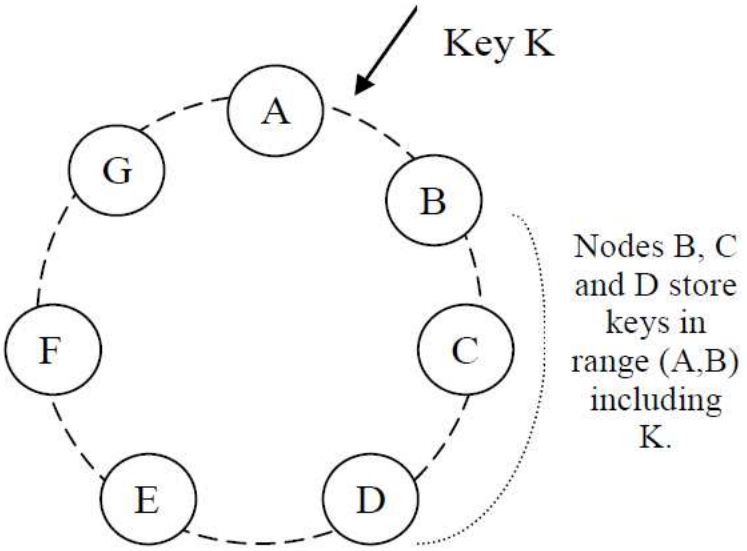
- Utilize “virtual nodes” along ring
  - Many virtual nodes per physical node
  - larger machines can hold more virtual nodes
  - Heterogeneous hardware is properly load balanced

Replication

Each data item is replicated at N hosts.

preference list: The list of nodes that is responsible for storing a particular key.

Some fine-tuning to account for virtual nodes



Preference Lists

- List of nodes responsible for storing a particular key.
- Due to failures, preference list contains more than N nodes.
- Due to virtual nodes, preference list skips positions to ensure distinct physical nodes.

Replication: Sloppy Quorum

- **Quorum System**: R + W > N, W > N/2
  - R, W, N are tunable

- Each node maintains a "preference list" of replicas for its own data
- Replicas are made on first N *healthy* nodes from preference list
  - require R nodes to respond for get()
  - require W nodes to respond for put()

Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- Challenge: an object may have distinct versions
- Solution: use vector clocks in order to capture causality between different versions of same object.

Vector Clock

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- If the all counters on the first object's clock are less-than-or-equal to all of the counters in the second clock, then the first is an ancestor of the second and can be forgotten.
- Application reconciles divergent versions and collapses into a single new version.

High Availability for Writes

- Clients write to first node they find
  - Vector clocks timestamp writes
  - Different versions of key's value live on different nodes

- Conflicts are resolved during reads
  - Like git: "automerge conflict" is handled by end application

Using Redis from your program

- Java Clients for Redis
  - **Lettuce** or **Jedis**
    - https://redislabs.com/lp/redis-java/
- Python Client for Redis
  - **Redis-py**
    - https://redislabs.com/lp/python-redis/

For today…

- Install your favorite programming language driver and connect to the Redis server.
- Insert and increment a value within a transaction.
- Submit the file with your code to ICON.

Bitmaps and Bloom Filters Redis

http://redis.io

Binary Vectors in Redis

- Redis support bit level operations
- Commands SETBIT and GETBIT let you manipulate bit locations in a bit sequence, starting at 0
- Commands BITOP, BITCOUNT, and BITPOS operate on groups of bits
  - SETBIT subscribers 0 1
  - SETBIT subscribers 98 1
  - BITCOUNT subscribers
  - SETBIT visitors 98 1
  - GETBIT subscribers 3
  - BITOP AND sub:visitors subscribers visitors
  - BITPOS sub:visitors 1

Membership queries

- Improve searches with data structures that check for the nonexistence of an item in a set.
- Can return false positives but guarantee no false negatives.
- Approximate set membership problem
- Trade-off between the space and the false positive probability .
- Generalize the hashing ideas.

Approximate set membership problem

- Suppose we have a set

  $S = \{s_1, s_2, \ldots, s_n\} \subseteq$ universe U
- Represent S in such a way we can quickly answer "Is x an element of S ?"
- To take as little space as possible ,we allow false positive (i.e. $x \notin S$, but we answer yes)
- If $x \in S$, we must answer yes .

Bloom filters

- Originally developed by Burton Howard Bloom in 1970 for spell-checking applications
- Consist of an arrays A[n] of n bits (space) , and k independent random hash functions

  $h_1, \ldots, h_k : U \dashrightarrow \{0,1,..,n-1\}$

1. Initially set the array to 0
2. $\forall\ s \in S$, $A[h_i(s)] = 1$ for $1 \leq i \leq k$

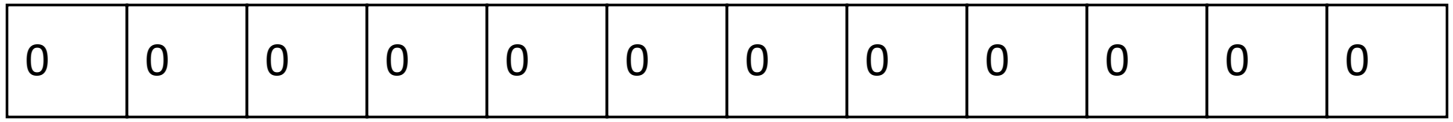
   (an entry can be set to 1 multiple times, only the first times has an effect)
3. To check if $x \in S$, we check whether all location $A[h_i(x)]$ for $1 \leq i \leq k$ are set to 1

If not, clearly $x \notin S$.

If all $A[h_i(x)]$ are set to 1, we assume $x \in S$

Bloom filter example

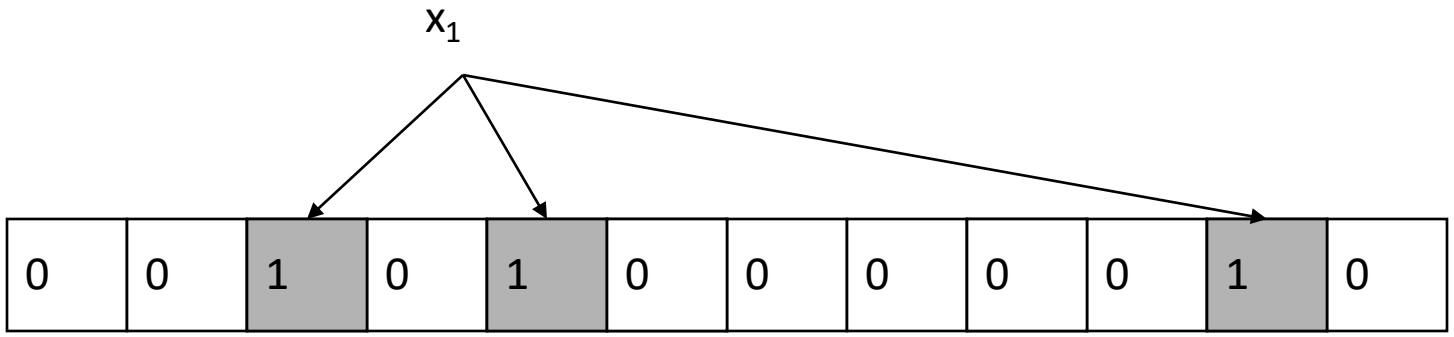
Consider k=3 independent hash functions
Bloom filter size n=12 bits
Possible number of elements m



Initially all positions are 0

Bloom filter example

Insert X_1

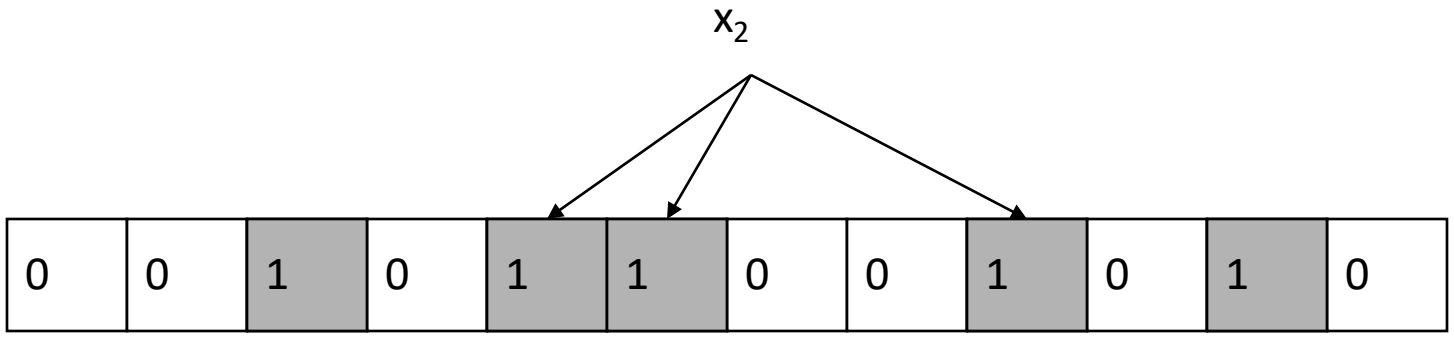


Each element of S is hashed k times

Each hash location set to 1

Bloom filter example

Insert X_2

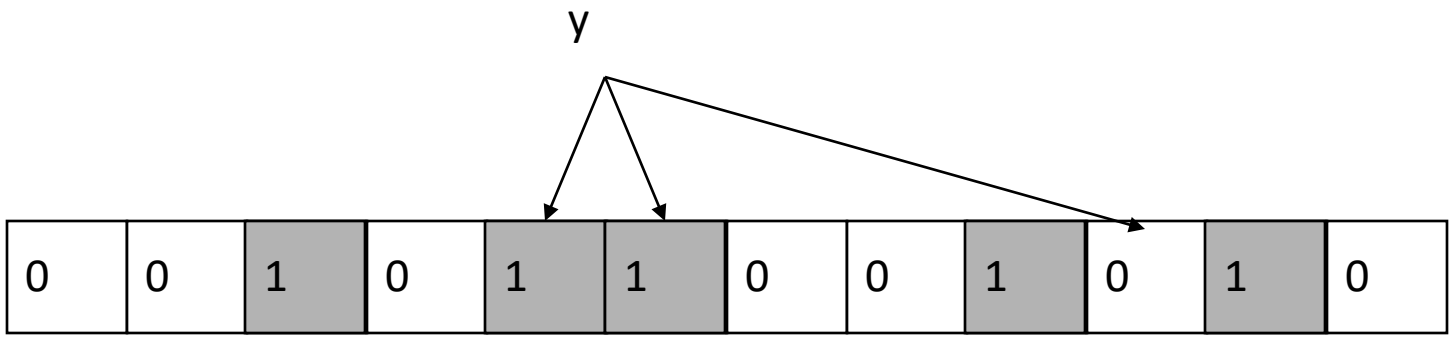


Each element of S is hashed k times

Each hash location set to 1

Bloom filter example

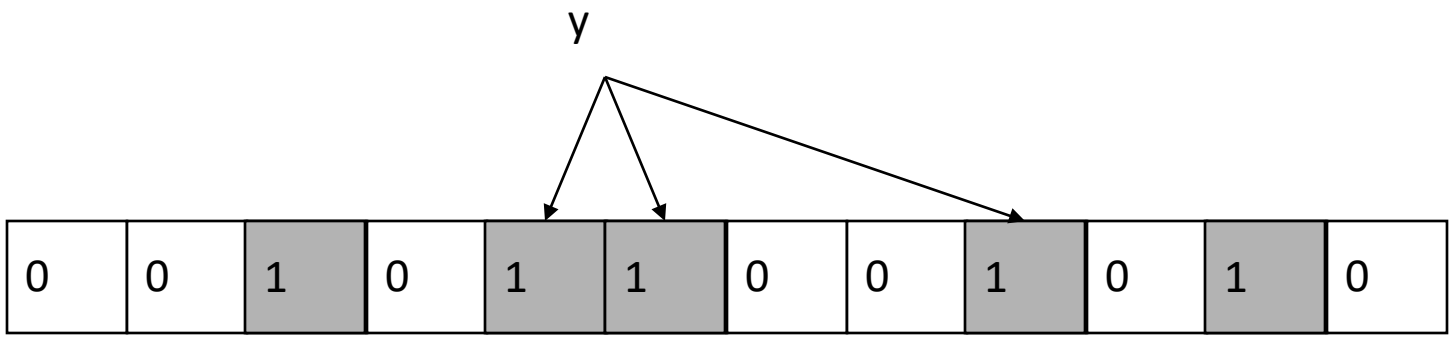
Query y



To check if y is in S, check the k hash location. If a 0 appears , y is not in S

Bloom filter example

Query y



If only 1s appear, conclude that y is in S

This may yield false positive

The probability of a false positive

- We assume the hash function are random.
- After all the elements of S are hashed into the bloom filters ,the probability that a specific bit is still 0 is

$$p = (1 - \frac{1}{n})^{km} \approx e^{-km/n}$$

To simplify the analysis ,we can assume a fraction p of the entries are still 0 after all the elements of S are hashed into bloom filters.

Probability of a false positive

- The probability of a false positive f is

$$f = (1-p)^k \approx (1-e^{-km/n})^k$$

- To find the optimal k to minimize f .

Minimize f iff minimize g=ln(f)

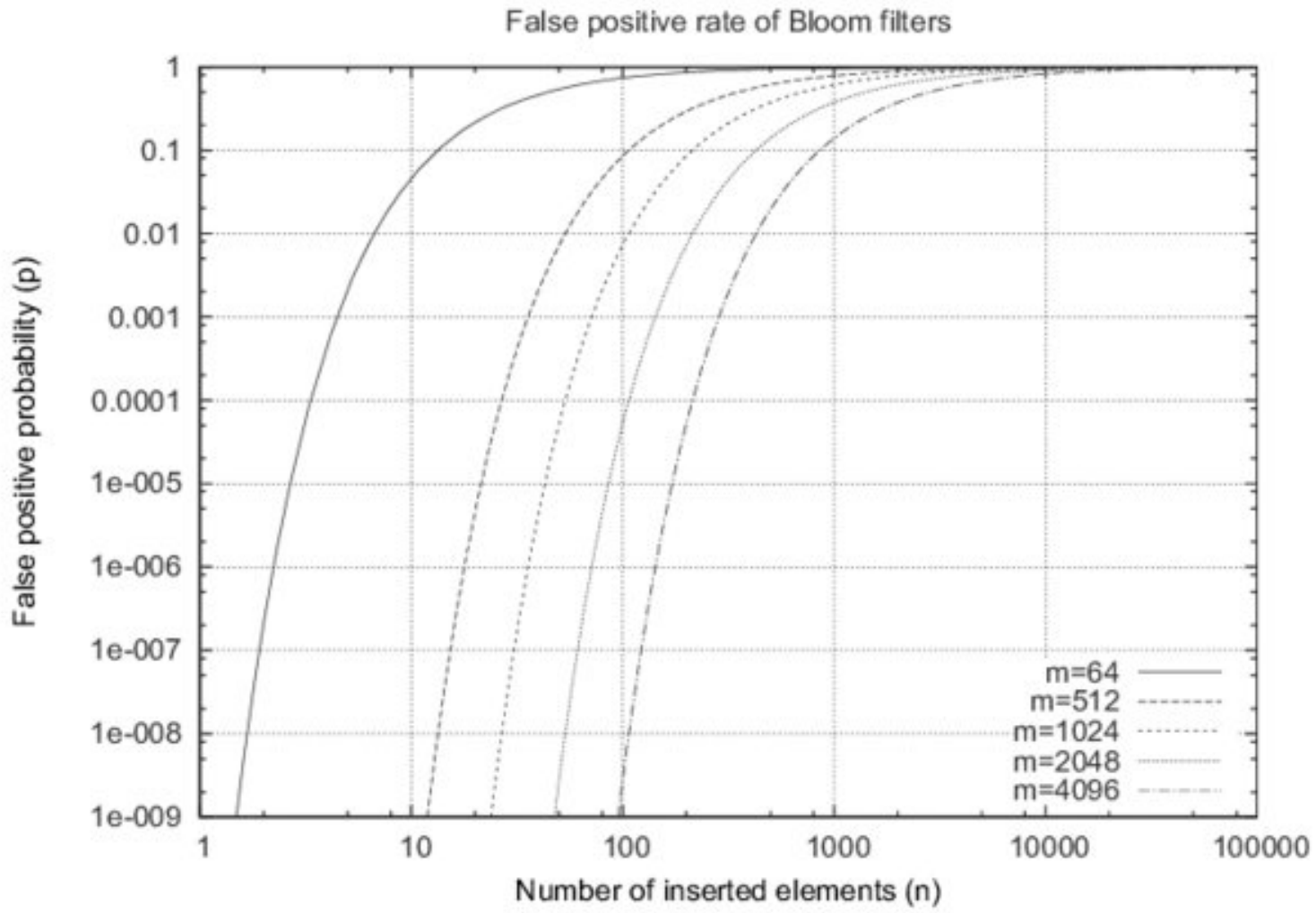
$$\frac{dg}{dk} = \ln(1-e^{-km/n}) + \frac{km}{n}\frac{e^{-km/n}}{1-e^{-km/n}}$$

$\Rightarrow$k=ln(2)*(n/m)

$\Rightarrow$f = $(1/2)^k$ = $(0.6185..)^{n/m}$

The false positive probability falls exponentially in n/m ,the number bits used per item !!

False positive rate for Bloom Filters



Bloom filter remarks

- A Bloom filters is like a hash table ,and simply uses one bit to keep track whether an item hashed to the location.
- If k=1 , it's equivalent to a hashing based fingerprint system.
- If n=cm for small constant c, such as c=8 ,then k=5 or 6 ,the false positive probability is just over 2% .
- It's interesting that when k is optimal

  k=ln(2)*(n/m) , then p= 1/2.

  An optimized Bloom filters looks like a random bit-string

Redis modules

- Check out redis modules for third-party libraries and enhancement you can use in your projects
- https://redis.io/modules

For today…

- Finalize the groups
- Decide on the dataset you'll use for your program
- One person from your team should submit the ICON survey to describe the project:
  - Title of the project and short description
  - Dataset you plan to use – provide link and/or stats
  - Databases you plan to use (at least two)
  - Queries/analysis you will perform (at least two)