Redis Base

\*\*2. Redis Data Type

\*String

- 문자열, 숫자, Serialized Object(Json String) 등을 저장 할 수 있다.

- 명령어

1. Set lecture inflearn-redis : lecture 라는 키로 ‘inflearn-redis’라는 string 데이터 값을 저장

3. MSET price 100 language ko : MSET은 멀티Set의 약자로서 다수의 스트링값을 한번에 저장

4. MGET lecture price language : MGet은 다수의 스트링 값을 한번에 반환

- redis는 별도의 integer타입 없이 스트링으로 숫자를 저장, 다반 스트링 값으로 숫자를 저장할경우 별도의 더하기 빼기 연산이 가능하다.

1. INCR price : INCR 명령어는 increse의 약자로서 숫자형 스트링 값을 1 올릴때 사용

2. INCRBY price 10 : INCRBY 명령어는 숫자형 스트링 값에 특정 값을 더할 때 사용

-Redis에는 SET inflearn-redis ‘{“price : 100, “language”: “ko”}’과 같이 json string을 직접 저장이 가능하다. Json 오브젝트 형태의 key-valye 데이터를 스트링으로 저장하기 위해 사용할 때 해당 데이터를 직접 json으로 바꿔서 사용해야 한다.

-Redis에서는 일반적으로 키를 만들 때 다음과 같이 :(콜론)을 이용하여 의미를 부여한다.

SET inflearn-redis:ko:price 200 [인프런-레디스 강좌의 한국어 강의 가격이 200이다]

이러한 규칙은 반드시 지키는 것은 아니지만 redis 에서 자주 사용되는 네이밍 컨벤션이기 때문에 가급적 지켜주는게 좋다.

\*lists

- list타입은 String을 Linked list로 저장함. 특히 redis의 lists는 각 노드가, 이전노드와 다음 노드를 가르키는 double linked lists임.

- 양끝에서 데이터를 추가 및 삭제하는 함수가 최적화 되어있음.(push/pop이 o(1)로 최적화)

- Queue(FIFO)나 Stack(FILO)를 쉽게 구현할 수 있다.

1. LPUSH queue job1 job2 job3, RPOP queue

2. LPUSH stack job1 job2 job3, LPOP stack

3. LPUSH queue job1 job2 job3, LRANGE queue -2 -1 [여기서 LRANGE는 다수의 아이템 조회하는데 인덱스를 사용함], LTRIM queue 0 0

3.1 LRANGE 명령어의 경우 오른->왼쪽으로 카운팅하면 0..1..2처럼 증가하고 왼쪽에서 ->오른쪽으로 카운팅하면 -3..-2..-1..처럼 감소한다.

3.2 아이템을 조회후 필요한 작업을 마쳤다면 LTRIM 명령어를 통해 나머지 부분만 남길 수 있음. (예제 에서는 가장 마지막에 추가된 아이템은 0번만 남김)

\*Sets

- Sets은 중복없이 유니크한 string을 저장하는 정렬되지 않은 집합임. (순서 보장 x)

- Sets은 집합에 대한 연산이 구현되어 있음 ( intersection : 교집합, union : 합집합 , difference : 차집합)

- SADD명령어를 통해 Set에 아이템 추가가 가능하다.

[sadd 명령어를 통해 user1이 좋아하는 과일을 셋에 추가하기] -> SADD user:1:fruits apple banana orange orange, 이렇게 오렌지를 중복으로 추가하도 set의 중복제거 특성에 의하여 오렌지는 한번만 반영됨,

-SMEMBERS user:1:fruits 명령어는 set의 모든 멤버를 출력한다.

-SCARD user:1:fruits는 set의 카디널리티를 출력(카디널리티는 고유한 아이템의 개수임)

-SISMEMBER user:1:fruits banana : SISMEMBER를 이용하면 특정 아이템이 셋에 포함되었는지 확인할 수 있다.

-SINTER user:1:fruits user:2:fruits : SINTER는 user1과 user2의 공통 좋아하는 과일 출력.

-SDIFF user1:fruits user2:fruits : SDIFF는 USER1은 좋아하지만 user2가 좋아하지 않는 과일 출력

-SUNION user1:fruits user2:fruits : SUNION은 유저1과 유저2중에 한명이라도 좋아하는 과일 출력v

\*Hash

- Hash는 field-value구조의 데이터 타입으로서 딕셔너리나 맵과 유사한 개념임.

- 다양한 속성의 갖는 객체의 데이터를 저장할 때 유용함.

- HSET lecture name inflearn-redis price 100 language ko : HSET 명령어를 통해 Hash를 생성할 수 있는데 다수의 필드와 value를 한번에 저장 가능

- HGET lecture name : 다수의 필드와 value중 하나의 필드를 조회

- HMGET lecture price language invalid : 다수의 필드를 조회함.

- HINCRBY picture price 10 : 숫자형으로 저장된 필드의 스트링 값을 특정 값만큼 더함. (HINCRBY는 특히 숫자형 스트링에만 사용가능하고 다른 스트링 형태는 에러가 나옴.)

\* Sorted Sets

- Sotrted Sets는 중복없이 유니크한 스트링 value를 저장하는 set과 비슷하지만 score라는 추가적인 필드를 가짐으로써 score를 통해 데이터를 미치 정렬하는 타입임.( Set 기능 + score 속성 저장)

- redis 만의 독특한 데이터 타입으로 내부적으로 Skip list와 Hash table로 구현됨.

- 값이 추가될 때마다 score값에 의하여 정렬이 진행됨.

- score가 동일하면 lexicographically사전 편찬순으로 정렬함

- sorted set은 줄여서 ZSets로 부른다.

- ZADD points 10 TeamA 10 TeamB 60 Tream C, A->B->C

- ZRANGE points 0 -1 : 처음부터 마지막까지 조회

- ZRange points 0 -1 REV WITHSCORES : REV는 역순으로 반환하고 WITHSCORES는 스코어와 함께 반환함.

-ZRANK points TeamA : ZRANK는 해당 아이템의 랭킹을 반환함. (이 랭킹은 0부터 시작하는 인덱스 값과 동일함)

\*Steam

-Steam은 Append-only log에 Consumer Groups과 같은 기능을 더한 자료구조이다.

-Append-only log : 삭제, 수정 없이 항상 새로운 이벤트를 맨 뒤에 기록함

-Consumer Groups : 이벤트가 많을 때 여러 컨슈머가 나눠서 처리하는 부하분산

-Kafka와 같은 이벤트 스트리밍 플랫폼과 유사한 부분들이 있다.

-kafka는 대규묘 데이터 스트리밍 처리를 위한 분산 이벤트 스트리밍 플랫폼임.

-레디스 스트림에는 몇가지 추가적인 기능이 있따.

1. 스트림에 추가되는 이벤트, 메시지는 고유 ID를 갖는데 이는 스트림에 추가되는 시간과 순서를 기준으로 자동 할당됨.

2. 고유 ID는 해당 Entry를 읽는데 상수 시간(O(1)) 복잡도를 가짐.

3. 분산 시스템의 다수의 컨슈머에서 안전하게 메시지를 소비할수 있도록 컨슈머 그룹 기능이 포함됨.

4. 리스트를 통해 구현한 Message Broker의 경우 다수의 컨슈머에서 메시지를 컨슈밍 할 때 동일한 메시지를 여러 번 처리하는 문제가 발생할 수도 있는데 Redis Steam의 컨슈머 그룹을 이용하면 동일한 메시지를 중복처리 하는 문제를 쉽게 해결이 가능

- XADD events \* action like user\_id 1 product\_id 1, XADD events\*action like user\_id 2 product\_id 1 : xadd명령어를 통해 스트림의 entry를 추가하게 되고 key 이후 ‘\*’ 옵션을 주면 자동으로 유니한 읽기가 할당 됨. 이후에 해쉬와 같은 field-value 형태로 메시지 구성이 가능하다.

- XRANGE events -+ : Xrange 명령어를 이용하면 리스트와 유사하게 다수의 메시지를 조회 처리가 가능하다.

- XDEL events ID 처리가 완료된 메시지는 XDEL명령어로 스트림에서 제거 가능

\*Geospatial

-Geospatial 는 좌표를 저장하고, 검색 가능한 데이터 타입

- 거리계산, 범위 탐색등을 지원함.

-GEOADD seoul:station 126.923917 37.556944 hong-dae 127.027583 37.497928 gang-nam -> GEOADD를 통해 강남역과 홍대역을 저장

-GEODIST seoul:station hong-dae gang-nam KM : GETDIST 두 역의 거리를 구함. (옵션을 통해 출력 단위 조절도 가능)

\*Bitmaps

- 레디스의 Bitmap은 별도의 실제 데이터 타입은 아니지만, String에 Binary operation 쉽게 사용할 수 있게 만든 인터페이스이다.

- 최대 2^32(42억개)의 binary 데이터 표현

- bitmap은 적은 메모리를 사용하여 binary 상태값을 저장하는데 많이 사용함.

- SETBIT user:log-in:23-01-01 123 1 : SETBIT를 통해 2023년 1월 1일의 유저의 로그인 여부를 저장함. 만약 유저가 연속적인 ID를 가지고 있다고 할 때 하나의 유저를 하나의 오프셋으로 표현 가능하다.

Ex)

123유저와 456유저가 1일에 로그인 했고, 123번 유저가 1월 2일에도 로그인함. 이때 비트 카운트를 이용하면 1일에 로그인한 유저 수를 출력할 수 있다.

SETBIT user:log-in:23-01-01 123 1

SETBIT user:log-in:23-01-01 456 1

BITCOUNT user:log-in:23-01-01

-BITOP AND result user:log-in:23-01-01 user:log-in:23-01-02

BITOP명렁어에 AND를 이용하면 1일과 2일에 로그인한 유저 모두 확인 가능 결과는 result라는 비트맵을 생성 후 저장함.

- GETBIT result 123 : GETBIT를 이용하여 result의 123 오프셋 데이터를 출력

\*HyperLogLog

-HyperLogLog은 집합의 카디널리티를 추정할 수 있는 확률형 자료구조이다.

-확률형 자료구조란 결과값이 실제와 일정 부분 오차가 존재할 수 있다는 것임.

-HyperLogLog에서는 카디널리티 계산시 ‘추정’이라는 표현을 사용함.

-정확성을 일부 포기하지만 저장공간을 매우 효율적으로 사용함(평균 에러율 0.81%)

-따라서 정확히 알필요 없이 근사치만 알아도 되는 경우에 HyperLogLogd를 사용한다.

-원리

1. 멤버의 값을 해싱하여 Bucket이라는 단위로 분류해서 해쉬값에 맞게 표시함.

2. 동일한 아이템이 추가하면 동일한 해쉬를 받기 때문에 일정한 카디널리티 계산 가능

3. 비록 HyperLogLog는 확률적인 계산식이 다르기 때문에 최종 결과값은 실제와 다를 수 있음.

4. 예를 해쉬 충돌이 발생하게 됬을 때 HyperLogLog는 정확하지 않은 카디널리티를 반환함.

5. Set을 이용해서도 카디널리티를 계산할 수 있는데 중복 검사를 위해 값을 메모리에 모두 저장함. 즉 아이템 개수에 비례하여 저장공간이 증가하지만 HyperLogLog를 이용하면 매우 적인 메모리로 카디널리티 계산이 가능하다.

6. 하지만 HyperLogLog는 실제 값을 저장하지 않기 때문에 모든 아이템을 다시 출력 해야할 땐 활용하지 못한다.

- PFADD fruts apple orange grape kiwi

PFADD라는 명령어로 HyperLogLog의 멤버 추가를 한다.

-PFCOUNT fruits

PFCOUNT라는 명령어로 fruits HyperLogLog의 카디널리티를 출력할 수 있다.

-비교 (bash 또는 cli 환경에서 입력하자 , redis-cli에서 입력하는거 아님)

for (( i=1; i<=1000; i++)); do redis-cli SADD k1 $i; done, MEMORY USAGE k1(이건 redis), SCARDl1

for (( i=1; i<=1000; i++)); do redis-cli PFADD k1 $i; done, MEMORY USAGE k2, PFCOUNT k2

\*BloomFilter

- BloomFilter 는 HyperLogLog처럼 확률형 자료구조의 일종임.

- BloomFilter 는 아이템이 특정 집합 안에 포함되어 있는지 확인하는 확률형 자료구조임. (멤버쉽 테스트 기능 구현시 자주 사용 됨)

- 정확성을 일부 포기하지만 저장공간을 효율적으로 사용 가능

- False positive : BloomFilter 는 값이 집합에 포함되어 있지 않는다는 사실은 정확히 확인 할 수 있지만, 포함되지 않는 값이 존재한다고 잘못 말하는 false positive를 반환하는경우가 있다

- BloomFilter는 실제 값을 저장 하지 않아 SET에 비해 매우 적은 메모리를 사용한다.

- 원리

1. 값을 해싱하여 여러 개의 해쉬 키를 만들어 낸다.

2. 여러 개의 키에 해당하는 위치를 블룸필터에 표시하게 된다. 이후에 어떤 아이템이 존재하는지 여부를 확인할 때는 다시 해쉬키를 생성하여 해당 위치를 확인한다.

3. 만약 애플과 오렌지를 BloomFilter에 먼저 추가한다음 그레이프라는 값이 BloomFilter에 이미 추가되었는지를 확인할 때 그레이프의 경우 왼쪽 하나의 해시키는 표시가 되어 있지만 다른 오른쪽 해시키는 표시되어 있지 않기 때문에 해당 값이 집합에 포함되지 않는다는 것은 확인할 수 있다.

4. 하지만 다른 해시키의 조합으로 키위에 해당하는 해당키가 이미 표시되었을 경우 False positive로 집합에 포함되어 있지 않지만 집합에 키위가 포함되어 있다고 잘못된 반환을 한다.

5. 따라서 BloomFilter를 사용할땐 반드시 false positive에 대한 문제를 인지하고 있자.

-BF.MADD fruits apple orange [하나 넣는건 BF.ADD임]

BF.MADD 명령어를 통해 집합 에 아이템 추가

-BF.EXISTS fruits apple

BF.EXISTS 명령어를 통해 집합에 아이템이 포함되었는지 여부 확인.