Session 11

Fall 2018

Topics

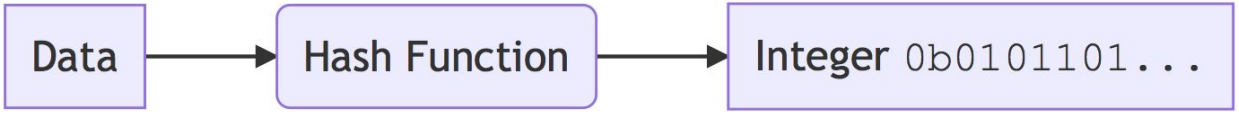
Hashing review and applications

Hashing python objects

Memoization with lru_cache

Q&A

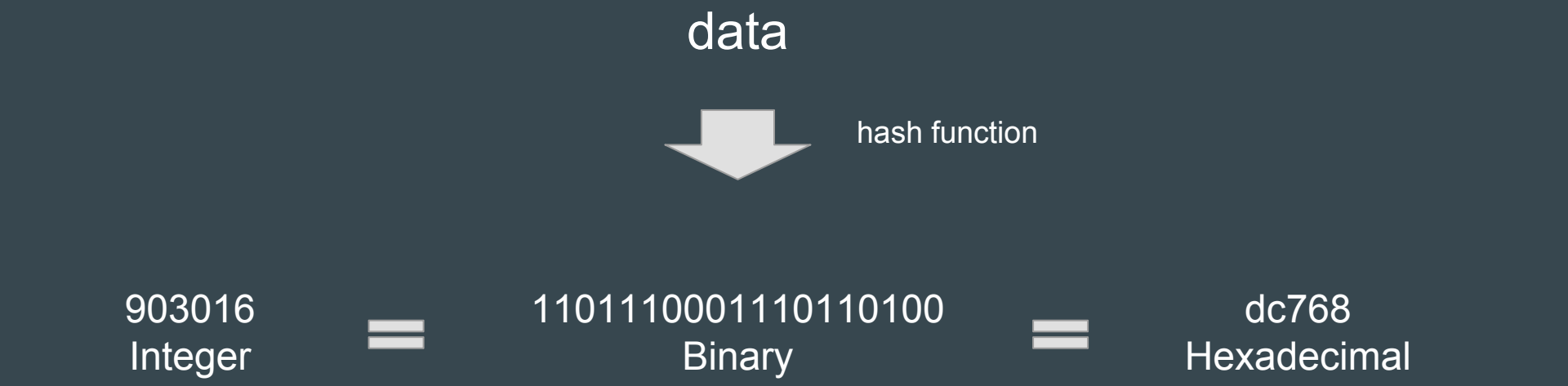
Review: hashing



A hash function, like `sha256`, maps an arbitrary string of data into a fixed-width integer, eg 256 bits

The integer appears ***uniform*** and is deterministic, which makes it very useful for ***anonymity***, without a DB tracking name -> id

Review: hashing



Modulus of hash integer output

The % (modulo) operator yields the remainder from the division of the first argument by the second.

What can we use this for?

```
903016 % 5
```

```
1
```

```
903016 % 10
```

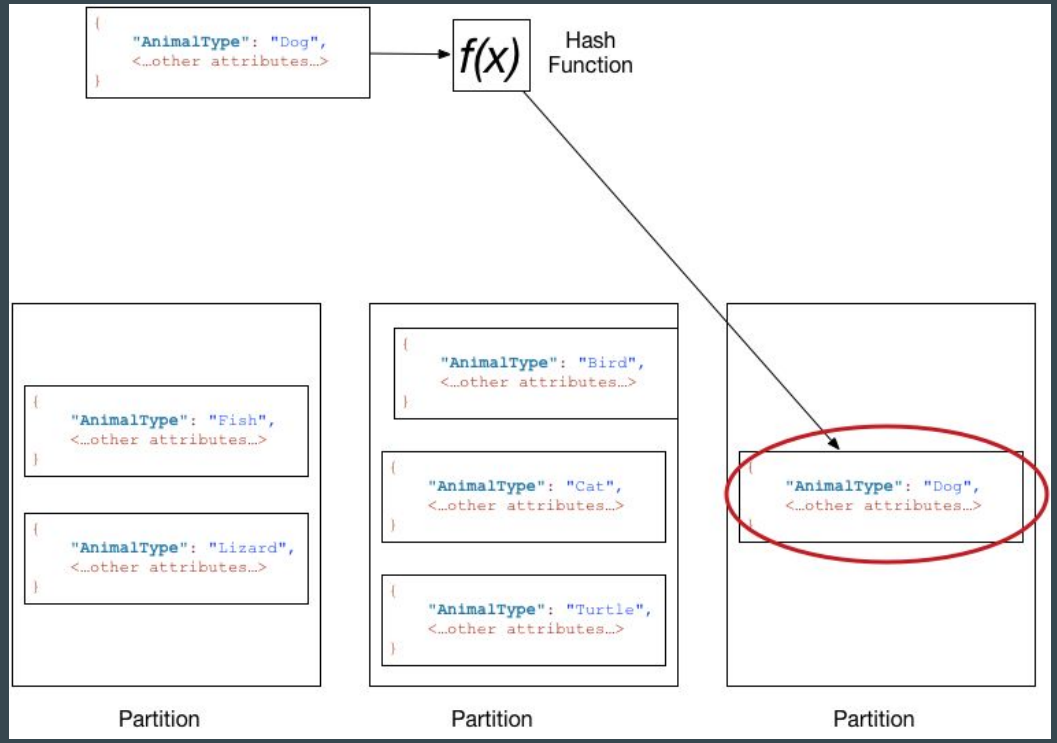
```
6
```

Partitioning example: AWS Dynamodb

Data Distribution: Partition Key

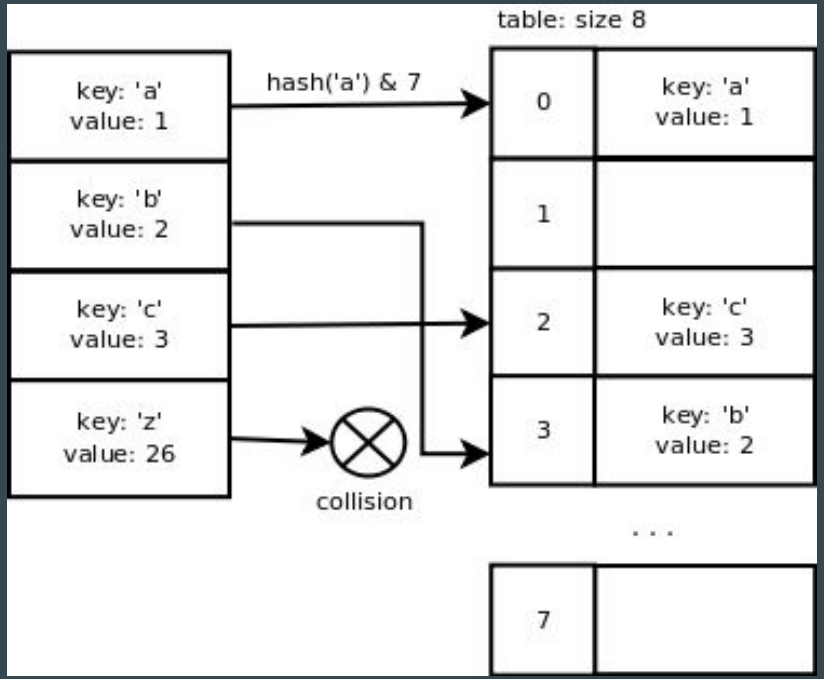
If your table has a simple primary key (partition key only), DynamoDB stores and retrieves each item based on its partition key value.

To write an item to the table, DynamoDB uses the value of the partition key as input to an internal hash function. The output value from the hash function determines the partition in which the item will be stored.



https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html

Python dictionary implementation: keys are hashed to get indices to the underlying storage array



https://www.laurentluce.com/posts/python-dictionary-implementation/

Hashable python objects

“**hashable**” (https://docs.python.org/3/glossary.html)

An object is hashable if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python’s immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their id().

Review: mutability

Mutability vs State

Mutable

An object/system which ***can change*** (even if it never does)

Immutable objects can be more performant

Stateful

An object/system which ***has a state***, which may be mutable or not

State can impact execution, even if the state is immutable

Stateless implies immutable!

Why the design choice to make mutable built-in objects non-hashable?

Hashing python objects

```
hash('a')
```

```
6736348275231354548
```

```
hash([1,2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-9ce67481a686> in <module>()
----> 1 hash([1,2])

TypeError: unhashable type: 'list'
```

```
hash((1,2))
```

```
3713081631934410656
```

What happens if we hash a custom object / class instance?

Implementing __hash__ method on an object

object.**__hash__**(*self*)

Called by built-in function `hash()` and for operations on members of hashed collections including `set`, `frozenset`, and `dict`. `__hash__()` should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a tuple and hashing the tuple. Example:

```
def __hash__(self):
    return hash((self.name, self.nick, self.color))
```

To make an object unhashable: __hash__ = None

https://docs.python.org/3/reference/datamodel.html#object.__hash__

Memoization/caching with functools.lru_cache

@functools.lru_cache(maxsize=128, typed=False)

Decorator to wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls. It can save time when an expensive or I/O bound function is periodically called with the same arguments.

Since a dictionary is used to cache results, the positional and keyword arguments to the function must be hashable.

Distinct argument patterns may be considered to be distinct calls with separate cache entries. For example, f(a=1, b=2) and f(b=2, a=1) differ in their keyword argument order and may have two separate cache entries.

Examples with lru_cache

```
from time import sleep
from functools import lru_cache

@lru_cache()
def double(a):
    sleep(1)
    return a+1
```

Performance gain from caching / Use .cache_info() to get stats on caching activity

```
%%time
double(1)
print(double.cache_info())
```

```
CacheInfo(hits=0, misses=1, maxsize=128, currsize=1)
CPU times: user 833 µs, sys: 1.18 ms, total: 2.01 ms
Wall time: 1 s
```

```
%%time
double(1)
print(double.cache_info())
```

```
CacheInfo(hits=1, misses=1, maxsize=128, currsize=1)
CPU times: user 69 µs, sys: 24 µs, total: 93 µs
Wall time: 86.1 µs
```

Use __wrapped__ to access the wrapped function

```
%%time
double.__wrapped__(1)
```

```
CPU times: user 613 µs, sys: 1.14 ms, total: 1.75 ms
Wall time: 1 s
2
```

```
%%time
double.__wrapped__(1)
```

```
CPU times: user 658 µs, sys: 970 µs, total: 1.63 ms
Wall time: 1.01 s
2
```

Recursive fibonacci: cached vs uncached

```
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

```
%%time
fibonacci(30)
```

```
CPU times: user 307 ms, sys: 2.3 ms, total: 309 ms
Wall time: 307 ms
```

```
@lru_cache()
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

```
%%time
fibonacci(30)
print(fibonacci.cache_info())
```

```
CacheInfo(hits=28, misses=31, maxsize=128, currsize=31)
CPU times: user 196 µs, sys: 288 µs, total: 484 µs
Wall time: 2.35 ms
```

Using lru_cache on a class

```python
from functools import lru_cache

@lru_cache()
class MyClass:
    def __init__(self, value):
        self.value = value

    def inc(self):
        self.value+=1
```

```python
x = MyClass(2)
# view memory address
print(x)
# increment instance attribute
x.inc()
print(x.value)
```

```
<__main__.MyClass object at 0x105d2e0f0>
3
```

```python
# compare the memory address to the previous instantiation call
y = MyClass(2)
print(y)
# we're working with the same instance - the value is already incremented
print(y.value)
```

```
<__main__.MyClass object at 0x105d2e0f0>
3
```

Using lru_cache on a class

```
# compare the memory address to the previous instantiation call
y = MyClass(2)
print(y)
# we're working with the same instance - the value is already incremented
print(y.value)
```

```
<__main__.MyClass object at 0x105d2e0f0>
3
```

```
# compare to wrapped class with no caching - instantiate a new class instance and increment
a = MyClass.__wrapped__(2)
print(a)
a.inc()
print(a.value)
```

```
<__main__.MyClass object at 0x105d2e2e8>
3
```

```
# instantiate again - we're working with a different instance
b = MyClass.__wrapped__(2)
print(b)
print(b.value)
```

```
<__main__.MyClass object at 0x105d2e2b0>
2
```

Using lru_cache on an instance method

What will happen here?

```
from functools import lru_cache
from time import sleep

class MyClass:
    def __init__(self, value):
        self.value = value

    def inc(self):
        self.value+=1

    @lru_cache()
    def get_value(self):
        sleep(1)
        return self.value

    __hash__ = None
```

Q&A