Cal Poly **CPE/CSC 369:** Distributed Computations Alexander Dekhtyar

Maps and Key-Value Stores

Theory

A lot of distributed computations you see in this class take place on objects often referred to as Maps or collections of Key-Value pairs or Key-Value stores.

Maps. In our conversations, a map is a partial finite function between two domains. That is:

Let $\mathcal{K} = \{\int_{\infty}, \ldots, \int_{\setminus}, \ldots\}$ be a set of objects called *keys*[1]. Let $\mathcal{V}$ be another set of objects (possibly infinite, possibly uncountable).

Let $K = \{k_1, \ldots, k_N\} \subseteq \mathcal{K}$ be a finite set of keys.

A map is any function $M : K \longrightarrow \mathcal{V}$.

Dictionary. Another name for a map defined as above that has been traditionally used in programming languages is dictionary.

We use the terms map and dictionary as synonyms.

Key-Value pairs. Given a map M, consider some key $k \in K$. Let $v = M(k)$. The pair $\langle k, v\rangle$ is known as a key-value pair in M.

Key-Value stores. Another way of looking at maps is to think of them as *sets of key-value pairs.* Indeed, we can describe a map M both as a function:

$$M : K \longrightarrow \mathcal{V}$$

as well as a set:

$$M = \{\langle k, v\rangle | k \in K, v \in \mathcal{V}, \sqsubseteq = \mathcal{M}(\|)\}.$$

[1] In this definition, this set is made countable. This is not a strict requirement, but under most circumstances it suffices.

or

$$M = \{\langle k, M(k)\rangle | k \in K\}$$

These two views of a map (as a function or as a set) are equivalent.

When viewed as a set of key-value pairs, a map is often referred to as a Key-Value Store.

Key-Value Store as Abstract Data Type

Maps/dictionaries are often implemented as an Abstract Data Type. The Map ADT comes with the following set of operations:

Operation	Parameters	Result	Action
`put`	`key, value`	none	add the `<key,value>` pair to the map
`get`	`key`	`value`	retrieve the `value` given a `key`
`exists`	`key`	`True/False`	return `True` if map contains a key
`size`	none	*integer*	return the number of key-value pairs in map
`remove`	`key`	none	remove the key-value pair with given `key` from map
`update`	`key, value`	none	replace the existing key-value pair for given `key` with the new `<key, value>` pair
`clear`	none	none	remove all key-value pairs from map

Note: The minimally viable Map ADT really just needs to implement put and get operations. Truly mutable maps will also require remove operation. All other operations are there for convenience.

Why Key-Value Stores are Important

Key-Value Store is an appropriate abstraction for dealing with a large number of use cases for distributed computing. Some of these cases are outlined below.

The "Facebook" example. This use case involves storing a large collection of records in a way where each record needs to be retrieved very fast given a unique key associated with the record (user id in the actual Facebook example). User Ids form the set of keys, and the user records (represented, for example, as a byte array) are values. The Key-Value store can be formed out of `<UserId, UserRecord>` pairs. `Put` operations can be used to add new records, `get` operations — to retrieve new records. Updates can be performed using an `update` update operation or aremove operation followed by a `put` operation.

Building simple indexes. A simple index is essentially a Key-Value store. The indexing attribute becomes the key. If key values are unique, the "values" in the key-value pairings are the specific objects being indexed. If they are not unique, than the values are collections of the objects being indexed (remember - the set $\mathcal{V}$ can consist of arbitrary objects, including of collection objects).

Inverted Indexes. A dictionary storing for each value, the list of objects in which this value occurs is often called an *inverted index*. Again, such a structure is essentially a map from a finite set of keys to collections of objects (or object references).

Key-Value Store Implementations

Many programming languages have Key-Value stores as implementations of the Map ADT.

Python. Python implements maps as `dictionary` objects.

Java. Java has a representation of the map ADT: the `Map <K,V>` interface. Its implementations are `HashMap`, `TreeMap` and `SortedMap`. The `Map` interface essentially implements the entire set of map operations, plus adds a few more operations for convenient manipulation of data.

JSON. A single JSON object can be easily viewed as a dictionary mapping the attribute/field names to their values.

Efficient Implementation of Key-Value Stores/Maps

There are three essential strategies to efficiently storing and retrieving a large collection of key-value pairs. All three are essentially represented in Java implementations of the `Map` interface.

1. **Hashing**. The keys are hashed and the values (or pointers to their actual locations) are stored in a hash table. This is what Java's `HashMap` does.

   Put operation hashes the key, finds the bucket, places the value/value pointer into the bucket.

   Get operation hashes the key, finds the bucket, searches the bucket for the key and its value.

2. **Sorting.** The keys are sorted, so the values are all stored in sort-order of the keys. This is implemented in Java's `SortedMap` class.

   Put operation finds where the key needs to go in the sort order, inserts the key-value pair there. Get operation navigates to the key and retrieves the value.

3. **B-trees.** B-trees and their equivalents (e.g., Red-Black trees) can be used to store the keys in a balanced sorted way. Java's `TreeMap` implements a Red-Black tree - based storage of key-value pairs.

   Put operation navigates the tree and inserts a new key-value pair in the correct location.

   Get operation navigates the tree and retrieves the value.