2.1.2 A Solution

Surprisingly, there is a very simple hashing scheme (described in this single paragraph) which fulfills all of the rather stringent requirements set out above. Documents and servers are mapped to points on a circle using standard hash functions. Assume for intuition that these functions distribute the documents and servers randomly around the circle. Now, a document is assigned to the server whose point is the first encountered when the circle is traversed clockwise from the document's point. An example is shown in figure 2-4.

Remarkably, this simple construction has all of the desired properties. Following is an intuitive discussion of the construction and in section 2.2.3 a rigorous analysis is presented.

Intuitively, if documents and servers are distributed randomly around the circle then each server should be responsible for roughly the same number of documents. Of course, there is a possibility that the points could be distributed badly so that one server receives a disproportionate fraction of the documents. While we cannot completely prevent such a misfortune, we can improve the chance of a good result with a simple trick which is described later (section 2.2.3).

Suppose that a new server is added to the system. The only documents that are reassigned are those with points that are now nearest to the new server's point; the mapping is not completely reshuffeled. Documents are only moved to the newly introduced server, and are not moved between old servers. Therefore most of the previously cached copied of documents are still valid because the newly obtained mapping is "consistent" with the previous one. Figure 2-4 illustrates this point.

UPDATE

This construction also behaves remarkably well in the presence of multiple views. Intuitively, if a document is assigned to a server in some view, then that document's point is likely to be relatively close to the server's point[1]. So only documents with points close to the server point are likely to be assigned to the server. Since document points are distributed randomly around the circle only a few document points fall close

[1]Since if they were far apart, then another server's point would be likely to fall in between, preventing the document from being assigned to the original server.

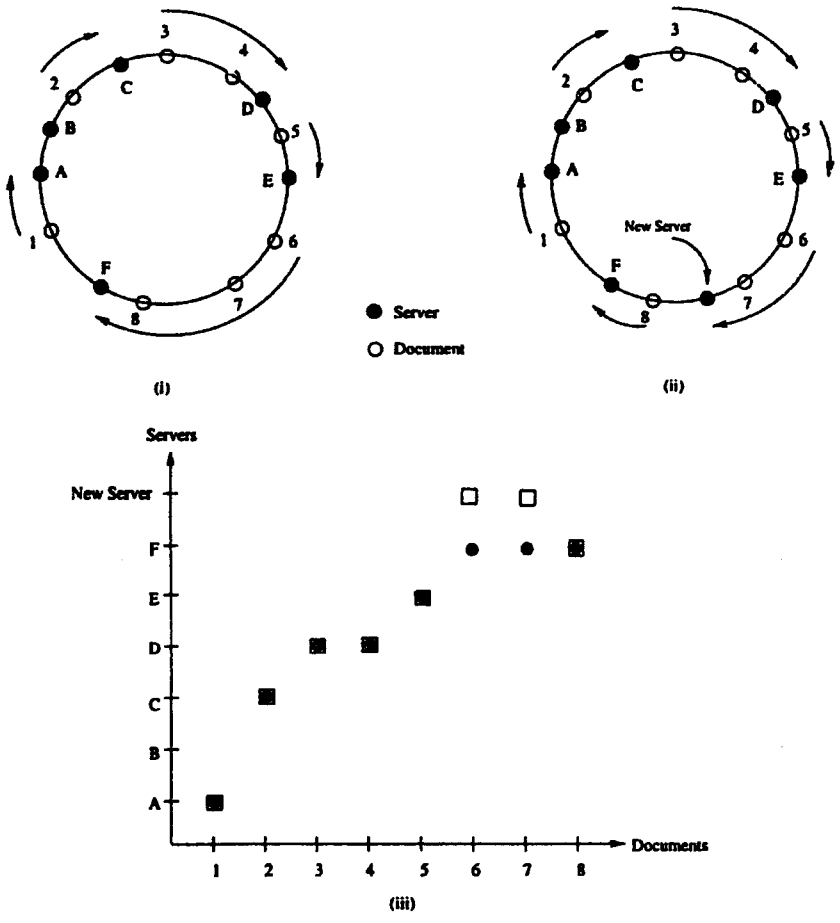

Figure 2-4: (i) Both documents and servers are mapped to points on a circle using standard hash functions. A documents is assigned to the closest server going clockwise around the circle. For example items 6, 7, and 8 are mapped to server F. Arrows show the mapping of documents to servers. (ii) When a new server is added the only documents that are reassigned are those those now closest to the new server going clockwise around the circle. In this case when we add the new server only items 6 and 7 move to the new server. Items do not move between previously existing servers. (iii) The mapping of documents to servers before and after the addition of a new server. Squares show the new mapping and circles are the previous mapping. Compare this with the results obtained from standard hashing in figure 2-3.

to a server's point. So even if there are a large number of views, only a relatively small number of documents are assigned to a server. The load of a server does not increase drastically with multiple views. In fact, we show in section 2.2.3 that the load increases only logarithmically in the number of views, while with standard hashing this dependence can be linear; clearly a substantial improvement.

Turning the above intuition inside out, if a server is responsible for a document then the server's point is likely to be relatively close to the document's point. Since server points are distributed randomly around the circle only a small number of servers points fall nearby any document point. So even if there are many views, only a few servers will have responsibility for any one document. The spread of a document does not increase dramatically with the number of views. As before, we show in section 2.2.3 that the spread increases only logarithmically in the number of views, while with standard hashing this dependence can again be linear.

Following is a summary of the important properties of the "circle hash function":

- Documents are distributed to servers "randomly".
- When a server is added, the only documents reassigned are those that are assigned to the new server. The newly obtained and old mappings are consistent with each other.
- The load of a server increases only logarithmically with the number of views in contrast to linearly with standard hashing.
- The spread of a document increases only logarithmically with the number of views in contrast to the linear dependence of standard hashing.

Clearly, using this new hashing technique in the previously described caching scheme solves the complications raised. The above "consistency properties" allow the caching scheme to grow gracefully with the growth of the network. Servers can be added to the network without disrupting the caching and multiple views of the servers can exist without degrading the performance of the system. Furthermore, the function is very simple and efficient to evaluate (see section 2.2.4).

2.2 Consistent Hash Functions

In this section we conceptualize the notion of a consistent hash function intuitively described in the previous section. Section 2.2.1 reviews the basics of hashing from a theoretical standpoint. Those familiar with the subject can skip this section. Consistent hash functions are defined in section 2.2.2. The circle construction of a consistent hash function is described in section 2.2.3, and the properties of this particular construction are derived and proved in section 2.2.3. Implementation issues are discussed in section 2.2.4.

In the following sections the motivating application of caching, is set aside as the central issue, and consistent hashing is discussed as a general hashing scheme which may have many other applications. Nonetheless, some intuitive discussions still derive from the simple caching scheme presented in the previous section.

2.2.1 Hash Families

We assume that the reader is familiar with the basic notion of a hash table data structure. In a hash table, a set of items I is mapped to a set of buckets B by a fixed *hash function.* In theoretical settings the intuitive notion of hashing is commonly modeled as follows: You are given a *family* of hash functions H whose elements are each functions that map the items I into the buckets B. The hashing is done by choosing a random function f from the family and using that function to map items to buckets.

Why do we use this seemingly more complicated model of hashing? Suppose that a malicious adversary[2] is aware of the hashing scheme being used and is attempting to devise an instance on which the scheme behaves poorly. Any scheme that uses a single fixed function to map items to buckets is vulnerable to an attack by an adversary since the input that causes the absolute worst case behavior of th· ·cheme can be determined.

[2]Throughout Throughout this paper we assume the existence of such adversaries. In fact, we claim that such adversaries are ubiquitous, and can be found anywhere you look.

This is best explained with an example. Suppose in the caching scheme presented in section 2.1, that an adversary has the power to permanently crash a number of servers. Assume in addition, that the mapping of documents (items) and servers (buckets) to the circle is known to the adversary. This adversary can concoct a plan to crash servers which would leave a remaining server heavily loaded. Specifically, the adversary can crash all servers in a portion of the circle, creating a gap with no servers at all. All the documents previously assigned to the crashed servers are now assigned to the server whose point is on the end of the gap. This server becomes overloaded with requests.

A standard method to foil such an adversary is adding randomization to the scheme. For example, if the points associated with servers and documents are chosen randomly and the adversary is not aware of these random choices, then the above scheme for overloading some servers does not work. The adversary does not know the placement of the points in the circle and therefore does not know how to create a gap.

In this paper, randomization is added to the scheme in a particular way. Choosing a function from a family of hash functions is done by taking one *uniformly and at random*. Thus, "using a hash family H" means that first a function $f \in H$ is selected uniformly and at random and then that function is used to map items to buckets.

Hash families are classified by various properties. Using a hash family involves randomization, so these properties are probabilistic. We say that a property *holds with probability* p for a hash family H if for a function chosen from the family uniformly and at random, the property holds with probability p. Following is an example.

Example 1 *(Linear Congruential Hashing) Let both the set of items I and the set of buckets B be $\{0, 1, 2, \ldots, p-1\}$ for a prime p. The family H is defined as all the functions of the form $f(x) = ax + b \pmod{p}$ for all $a, b \in \{0, 1, 2, \ldots, p-1\}$.*

To use this hash family, a and b are chosen at random (this is œ[Dœ[equivalent to choosing a function from the family at random). Then, item i is assigned to bucket $ai + b \pmod{p}$. The following lemma gives an example of a property of the hash

family that holds with a certain probability.

Lemma 2.2.1 *Let H be the hash family defined above, and let x and y be distinct items, then* $\Pr[f(x) = i \text{ and } f(y) = j] = 1/p^2$.

In other words, the probability that a randomly chosen function f from the family H maps the item x to i and the item y to j is the same as in the case where the two items are mapped to the buckets uniformly and independently.

Proof: Since the set $\{0, 1, 2, \ldots, p-1\}$ with operations $(mod\ p)$ is a field, there is a unique pair a and b such that $ax + b = i$ and $ay + b = j$. Thus the probability that $f(x) = i$ and $f(y) = j$ is equal to the probability of selecting the pair a, b while choosing a function from H. The probability of selecting any given pair a, b is $1/p^2$. ■

Lemma 2.2.1 shows that a randomly chosen function from H behaves like a completely random function with respect to pairs of items.[3] Why would we prefer to use the family H instead of choosing a completely random function? Notice that choosing, storing, and evaluating functions from H is remarkably simple and efficient. We pick a and b at random. These values can be stored using very little memory, and evaluating the function is simple. On the other hand, selecting a truly random function requires choosing and storing a random table of the value of the function on every single item.

Hash families similar to the one presented in the above example are important elements in the practical implementation of our consistent hash functions (section 2.2.4).

2.2.2 Definitions

Unlike traditional hash functions, consistent hash functions are intended to deal with situations in which the set of buckets (the range of the function) changes. Therefore,

[3] A hash family that looks random in respect to pairs of items is called *pairwise independent*. Similarly, families that look random with respect to k elements are k-way independent. Hash families that are k-way independent are important in section 2.2.4

$f_V(i)$ è il bucket a cui i è assegnato nella vista V.

we introduce the notion of a *ranged hash function* that permits the set of buckets in the range of the function to change.

Let the set of items be I and the set of buckets be B. A *view* is a subset of the buckets B. A *ranged hash function* is a function of the form $f : 2^B \times I \to B$. Such a function specifies an assignment of items to buckets for every possible view. That is, $f(V, i)$ is the bucket to which item i is assigned in view V. (We will use the notation $f_V(i)$ in place of $f(V, i)$ from now on.) Since items should only be assigned to available buckets (buckets that are in the current view), we require $f_V(I) \subseteq V$ for every view V.

A *ranged hash family* is a family of ranged hash functions.

In section 2.1 the following characteristics of ranged hash functions were discussed informally:

- **Balance:** Items are distributed to buckets "randomly" in every view.
- **Monotonicity:** When a bucket is added to a view, the only items reassigned are those that are assigned to the new bucket.
- **Load:** The *Load* of a bucket is the number of items assigned to a bucket over a set of views. Recall that ideally, the load should be small.
- **Spread:** The *Spread* of an item is the number of buckets an item is placed in over a set of views. Ideally, spread should be small.

The remainder of this section defines formally these intuitive properties. Throughout, we use the following notational conventions: H is a ranged hash family, f is a ranged hash function, V is a view, i is an item, and b is a bucket.

Def. **Balance:** A ranged hash family is *balanced* if, given any particular view V an item i, and a bucket $b \in V$, the probability that item i is mapped to bucket b in view V is $O(1/|V|)$.

The balance property is what is prized about standard hash functions: an item is equally likely to be put into any bucket. The balance property does not say anything

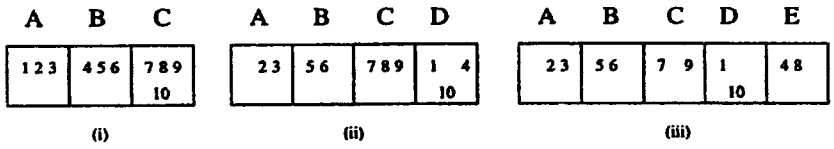

Figure 2-5: (i) The items $1, 2, \ldots, 10$ hashed into 3 buckets A, B, and C. (ii) A new, fourth bucket D is added. Since the hash function is monotone, the only items that are relocated are those that move into the new bucket D. (i.e. items 1, 4 and 10) (iii) Another new bucket E is added and some items move into E (items 4 and 8). Items do not move between old buckets.

about behavior over changing views, only that in each fixed view items are distributed with roughly equal probabilities.

Monotonicity: A ranged hash function f is *monotone* if for all views $V_1 \subseteq V_2 \subseteq B$, $f_{V_2}(i) \in V_1$ implies $f_{V_1}(i) = f_{V_2}(i)$. A ranged hash family is *monotone* if every ranged hash function in it is monotone.

The monotonicity property says that if items are initially assigned to a set of buckets V_1 and then some new buckets are added to form V_2, then an item may move from an old bucket to a new bucket, but not from one old bucket to another. This reflects one intuition about consistency: when the set of usable buckets changes, items should not be completely reshuffeled. Figure 2-5 gives an example of the monotonicity property.

The following lemma is a simple consequence of these two definitions and helps to clarify them. The lemma gives the the expected number of items that remain fixed when the range of the hash function changes.

Lemma 2.2.2 *Let H be a monotonic, balanced ranged hash family. Let V_1 and V_2 be views. The expected fraction of items i for which $f_{V_1}(i) = f_{V_2}(i)$ is $\Omega\left(\frac{|V_1 \cap V_2|}{|V_1 \cup V_2|}\right)$ where the probability is over the uniform selection of $f \in H$.*

Proof:

In this proof, we count the number of items i for which $f_{V_1}(i) \neq f_{V_2}(i)$. In other words, as the set of usable buckets changes from V_1 to V_2, we count items that "move" instead of the number of items that are "fixed".

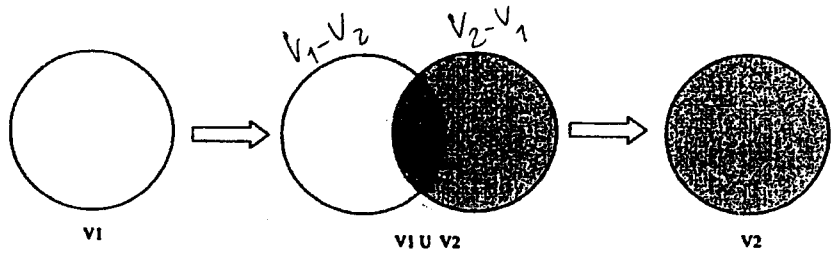

Figure 2-6: The view V_1 is transformed into the view V_2 in two steps. First V_1 is transformed into $V_1 \cup V_2$. In this step, monotonicity implies that items only move from V_1 into $V_2 - V_1$. Next, $V_1 \cup V_2$ is transformed into V_2. Here monotonicity implies that items only move from $V_1 - V_2$ into V_2. Balance tells us how many items are expected to move in each step.

We change the set of usable buckets from V_1 to V_2 in two steps. In the first step, we expand the set of buckets from V_1 to $V_1 \cup V_2$ (see figure 2-6). Balance implies that the expected fraction of items that move into $V_2 - V_1$ is $O\left(\frac{|V_2 - V_1|}{|V_1 \cup V_2|}\right)$. Monotonicity implies that no other items move.

In the second step we contract the set of buckets from $V_1 \cup V_2$ to V_2. Again, balance implies that the expected fraction of items that move into V_2 from $V_1 - V_2$ is $O\left(\frac{|V_1 - V_2|}{|V_1 \cup V_2|}\right)$ and monotonicity implies that no other items move.

In the first step, items were only moved into $V_2 - V_1$. In the second step, items were only moved out of $V_1 - V_2$. Since these sets are disjoint, no item moved in both steps. Therefore, the expected fraction of items which moved in either step is:

$$O\left(\frac{|V_2 - V_1|}{|V_1 \cup V_2|}\right) + O\left(\frac{|V_1 - V_2|}{|V_1 \cup V_2|}\right)$$

Therefore, the expected fraction of items that remain fixed is:

$$1 - O\left(\frac{|V_2 - V_1|}{|V_1 \cup V_2|} + \frac{|V_1 - V_2|}{|V_1 \cup V_2|}\right) = \Omega\left(\frac{|V_1 \cap V_2|}{|V_1 \cup V_2|}\right)$$

■

We continue to define properties of ranged hash families that capture additional aspects of the notion of "consistency".

Spread: Let $\mathcal{V} = \{V_1 \ldots V_k\}$ be a set of views. For a ranged hash function f, and a particular item i, the *spread* of the function on item i over the set of views $\mathcal{V}$ is the

quantity $|\{f_{V_1}(i), f_{V_2}(i), \ldots, f_{V_k}(i)\}|$. This quantity is denoted $spread_f(\mathcal{V}, i)$.

The spread of an item i over a set of views is the number of different buckets over all views that i is mapped to by f. Figure 2-7 illustrates spread.

In terms of the caching system described in section 2.1 spread is the number of different caches that are assigned responsibility for a document when there exist multiple views. If i is a document, then $\{f_{V_1}(i), f_{V_2}(i), \ldots, f_{V_k}(i)\}$ is the set of all the responsible caches. Recall that the central server supplies a copy of the document to each of these responsible caches, so low spread is vital if the scheme is to eliminate swamping.

Clearly, spread is very sensitive to the set of views $\mathcal{V}$. For example, if each view consisted of a single different bucket, then the spread of every item would be the total number of views! Even if views are restricted to contain at least a $1/t$ fraction of the buckets, then there exists a set of t views that force the spread of any item to be t. Simply take t *disjoint* views each containing a $1/t$ fraction of the buckets. In this case an item is assigned to a different bucket in every view. Hence, the spread of every item is t.

In general, we study spread under the assumption that each view contains at least a $1/t$ fraction of the buckets (t does not have to be a constant). As the above example shows, under this assumption spread cannot be less than t, but the question to be answered is: "How does the spread grow as a function of the number of views?" In some ranged hash families the spread grows linearly with the number of views. However, there exist families where the spread grows only logarithmically. One such family is described in section 2.2.3.

Load: Let $\mathcal{V} = \{V_1 \ldots V_k\}$ be a set of views. For a ranged hash function f, and a particular bucket b, the *load* of the function on bucket b over the set of views $\mathcal{V}$ is the quantity $\left|\bigcup_{V_j \in \mathcal{V}} f_{V_j}^{-1}(b)\right|$. This quantity is denoted $load_f(\mathcal{V}, b)$.

Note that $f_{V_j}^{-1}(b)$ is the set of items assigned to bucket b in view V_j. Thus, $load_v(\mathcal{V}, b)$ is the total number of items which in some view are assigned to the bucket b. Figure 2-8 illustrates the concept of load.

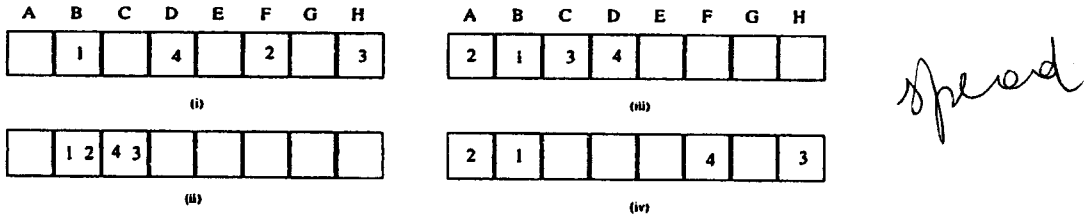

Figure 2-7: Four views of the buckets $\{A, B, C, D, E, F, G, H\}$ and the mapping of items $1, 2, 3, 4$ into the buckets for each view. Shaded buckets are not available in the given view. (i) The mapping for view $\{B, D, F, H\}$. (ii) The mapping for the view $\{B, C\}$. (iii) The mapping for the view $\{A, B, C, D\}$. (iv) The mapping for the view $\{A, B, F, H\}$. Item 1 is placed in bucket B in all the views. Thus, the spread of item 1 over these views is 1. Item 4 is placed in the buckets D,C, and F over the four views. Thus the spread of item 4 over these views is 3.

The load property measures the effect of multiple views on the number of items hashed to a given bucket. For example, in the caching scheme of section 2.1 a document i contributes to the load on a server b if there is as least one view such that i is assigned to b in that view ($f_V(i) = b$). Thus, load measures how many documents a server is responsible for in the presence of multiple views.

Load is related to spread, but they are not equivalent. For example imagine a hash function that maps all the items to a single bucket b. For any item and any set of views all containing b, the spread of i is one. On the other hand, the load of b is the total number of items. In this case the distribution of items is not balanced between the buckets.

As in the case of spread, load is generally studied under the assumption that each view contains at least a $1/t$ fraction of the buckets. (Again, t does not have to be a constant.)

We have defined a number of "consistency properties" of ranged hash families: Balance, Monotonicity, Spread, and Load. The name *consistent hash family* is used loosely to describe a ranged hash family that has good behavior with respect to these properties. When introducing a new hash family the performance relating to each of the consistency properties is explicitly stated.

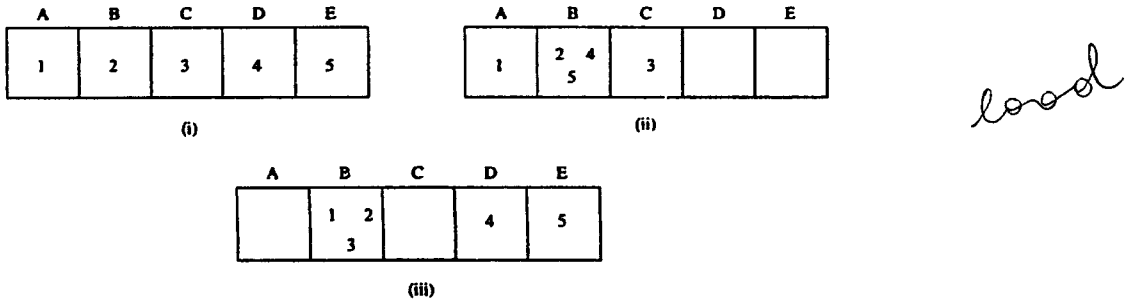

Figure 2-8: The distribution of items $1, 2, 3, 4, 5$ in buckets A, B, C, D, E for three different views. The load of bucket A over these views is one since only the item 1 is placed in it. The load of bucket B is five since all the items, in some view, are placed in it.

2.2.3 A Consistent Hash Family

This section describes the main example of a consistent hash family presented in this paper. The informal construction in section 2.1 is formalized as a ranged hash family in section 2.2.3. This formalization captures the intuition of the circle hash function. However, this simplified family requires manipulation of real numbers which is clearly a drawback. Nevertheless, in section 2.2.3 we state and prove the consistency parameters for this family. In section 2.2.4 we show how to modify the family to obtain one that is efficiently computable (does not rely on real numbers), and also retains the same consistency properties.

Construction

This section formalizes the intuition of the circle construction presented in section 2.1. In the basic construction, items and buckets are mapped to "random" points around a circle, and an item is mapped to the bucket whose point is encountered first when the circle is traversed clockwise from the item's point. Recall that there is a possibility that points could be distributed badly around the circle; one bucket may be responsible for a disproportionate section of the circle. Such an unfortunate instance is shown in figure 2-9. In order to decrease the likelihood of such an unlucky event we slightly modify the basic construction by assigning to a bucket m ($m \geq 1$) points around the circle instead of just one. As before, items are assigned to a single point

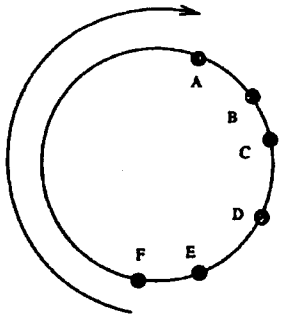

Figure 2-9: An unlucky random placement of bucket points around the unit circle. Bucket A is responsible for a disproportionately large section of the unit circle. Since items points are distributed randomly around the circle it is very likely that bucket A will have many more items assigned to it than other buckets do.

and are mapped to the bucket that has a point nearest to the item's point in the clockwise direction. Intuitively, when m is large there is less of a chance that buckets will be unequally distributed around th circle. We show that m need not be unmanageably large for there to be a very good chance of a good distribution of points. This modified version of the circle hash family is formally described below.

Let C be a circle with circumference one. We will call C the *unit circle.*[4] Let $r_I : I \mapsto C$ be a function that maps items to the unit circle. Let $r_B : B \times [m] \mapsto C$ ($[m] = \{1, 2, 3, \ldots, m\}$) be a function that maps multiple copies of buckets to the unit circle (Each pair (b, n) is considered a "copy" of the bucket b).

Each hash function in the "unit circle consistent hash family" is represented by a pair of functions: (r_B, r_I). Given such a pair, $f_V(i)$ is defined to be the bucket in the view V that has a point closest to $r_I(i)$ going clockwise around the circle. Thus, to compute the mapping of an item i in a view V we first map the bucket copies to the circle. Then, starting from the point $r_I(i)$, we sweep around the circle clockwise until we encounter a bucket point. The bucket associated with this point is the bucket that i is mapped to. Figure 2-10 depicts this situation for $m = 4$.

We say that a bucket point $r_B(b, n)$ is *responsible* for an arc in a view V if the

[4]Note that "unit circle" usually refers to a circle with radius equal to 1. We will always be referring to a circle with circumference equal to one.

bucket point is on the right end of the arc, there is no bucket point in the interior of the arc, and some other bucket point is on the left end of the arc. Any item whose point falls into this arc is assigned to the bucket b by the function. Figure 2-10 shows a mapping of bucket points to the unit circle, and the arcs that each bucket point is responsible for.

If we assume that the family is made up of all possible pairs of functions mapping buckets and items to the unit circle, then choosing a random function from the family is equivalent to choosing two completely random functions (r_B, r_I). Hence, we call this family UC_{random} (for "unit circle - random mappings").

Using the family UC_{random} requires the manipulation of real numbers. Clearly this is impractical, but in section 2.2.4 we show how to modify the construction so that real numbers are not used. For now, assume that we can compute with real numbers, that is, we can choose a function from the family and can evaluate functions in the family.

Analysis

In this section we state and prove the consistency parameters of the ranged hash family UC_{random}. These parameters hold probabilistically over the choice of function from the family. In particular, the bounds on the parameters will hold with probability at least $1 - 1/N$ where N is an arbitrarily chosen confidence factor. The confidence factor N appears in the bounds themselves so that if a high confidence factor is desired, then the bounds are degraded accordingly. This approach is a generalization of the common practice of expressing probabilities as a function of problem size. Throughout the thesis, logarithms are taken base e unless otherwise specified.

The following theorem states the consistency parameters of the family UC_{random}.

Theorem 2.2.3 *Let $\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$ be a set of views of the set of buckets B such that: $\left|\bigcup_{j=1}^{k} V_j\right| = T$ and for all $1 \leq j \leq k$, $|V_j| \geq T/t$. Let $N > 1$ be a confidence factor. If each bucket is replicated and mapped m times then:*

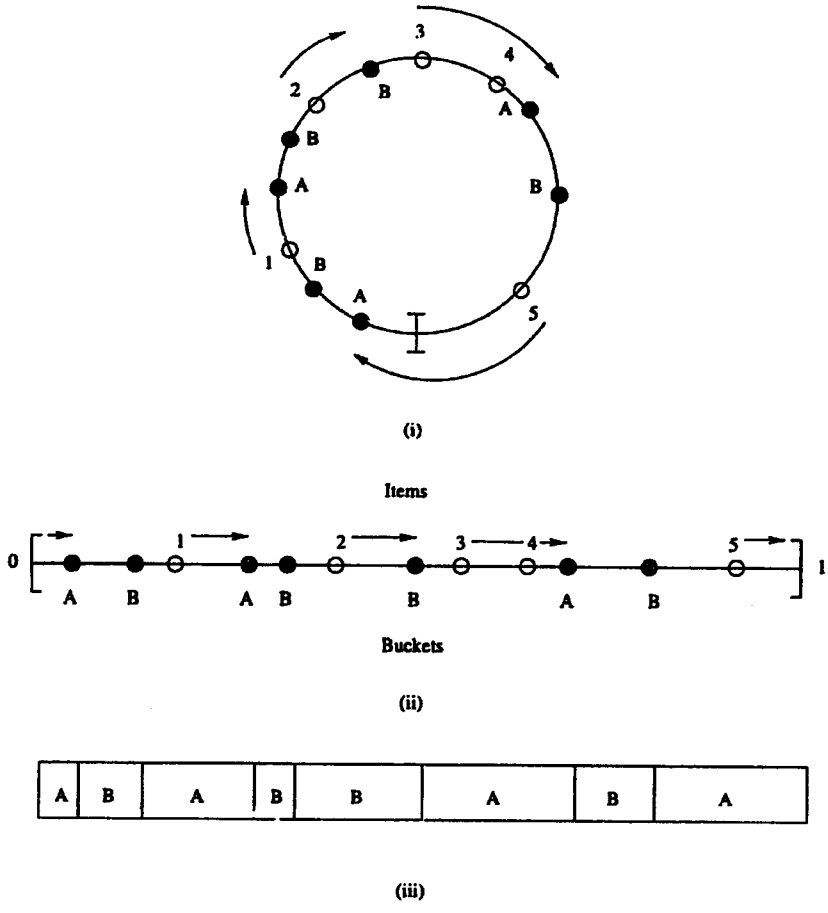

Figure 2-10: (i) A unit circle hash function with $m = 4$. Buckets A and B have 4 points associated with each of them. Items are mapped to the bucket closest to them going clockwise. Item 1 is closest to a point of bucket B and item 2 is closest to a point of bucket B. Item 5 is closest clockwise to a point of bucket A. (ii) The unit circle drawn as an interval with length one where we imagine that the endpoints of the interval are "glued together". (iii) The parts of the circle (viewed as an interval) that buckets A and B are responsible for. Bucket points are responsible for the arc directly to their left. Since there are multiple copies of each bucket, buckets are responsible for a set of arcs.