A Distributed Ordered Dictionary with $(1+\epsilon)$ Load Balance

Brian Babcock **Prasanna Ganesan**
Stanford University
{babcock,prasannag}@cs.stanford.edu

Abstract

We consider how to maintain a distributed dictionary over a set of nodes such that each node stores all the keys in one contiguous range of the (ordered) key domain. Such range-partitioned dictionaries are commonly used in parallel databases as they enable efficient range queries. As keys are inserted and removed from the dictionary, the partitioning needs to be adjusted in order to ensure storage balance across nodes. We develop an online algorithm that ensures that the asymptotic ratio of storage load between any pair of nodes is at most $(1+\epsilon)$, for any constant $\epsilon > 0$, while ensuring that the amortized cost per key insertion or deletion, measured as the number of keys that are migrated across nodes, is constant. Our algorithm can be extended to work for peer-to-peer systems where nodes themselves may join and leave the distributed dictionary.

1 Introduction

Consider a distributed dictionary over a set of n nodes, $N_1, N_2, ...N_n$, with keys drawn from an ordered domain. A common strategy for distributing the dictionary across nodes is to use range partitioning: node N_i stores all the keys in a range $[R_{i-1}, R_i)$, with $R_0 < R_1 < R_2 \ldots < R_n$. Such range partitioning is extremely popular in parallel database systems [1]; the cost of executing a range query in such systems is typically proportional to the number of nodes that process the query, thus making range partitioning an attractive strategy for minimizing query costs.

It is important to ensure that keys are equally distributed across nodes, even as keys are inserted and deleted. Such a requirement entails dynamic modifications to the range partitioning, with a concomitant movement of keys across nodes to establish load balance. We may define a node's *load* as the number of keys stored at that node, and characterize load balance by the *imbalance ratio* σ, defined to be the ratio of the largest and smallest loads. In this paper, we present an online algorithm to achieve an asymptotic imbalance ratio arbitrarily close to one, while ensuring that the amortized cost of achieving this load balance, measured as the number of keys migrated between nodes, is constant per key insertion and deletion. Our algorithm improves on prior work [2], which achieved an imbalance ratio equal to the cube of the golden ratio.

Our algorithm can be extended to work for distributed ordered dictionaries constructed on peer-to-peer systems [2, 3], where nodes may join and leave the system dynamically. Details are omitted from this extended abstract.

2 Problem Statement

In the dynamic range partitioning problem, we maintain a mapping f from a dynamic set of elements $X = x_1, \ldots, x_m$, drawn from a totally ordered domain, to the set of integer tags $[1, n]$. We also maintain a permutation π of the tags, such that $\pi(f(x_i)) < \pi(f(x_j))$ whenever $x_i < x_j$. Thus f partitions the m elements into n contiguous ranges.

Define the load $L(u)$ of a partition u to be the number of elements assigned to partition u, i.e. $L(u) = |\{x_i \in X : f(x_i) = u\}|$. We would like the partitioning to be well-balanced, in the sense that the imbalance ratio $\sigma = \frac{\max_i L(i)}{\min_j L(j)}$ is not too large. Maintaining a well-balanced partitioning as elements are inserted and deleted from the set X may require relabeling some elements. Our goal is to minimize the number of

relabeling operations required, subject to the constraint that the imbalance ratio cannot exceed a threshold $(1+\epsilon)$ representing the maximum tolerable level of imbalance.

In this paper, we present a range partitioning scheme, parameterized by ϵ, with the following properties: (1) the imbalance ratio σ is always less than $(1+\epsilon)$ whenever $|X| > c$, for some constant c, and (2) only a constant amortized number of relabelings are performed for each element insertion and deletion. These properties hold for any constant $\epsilon > 0$.

3 Algorithm

Our algorithm makes use of two basic relabeling primitives, `SHIFT` and `REORDER`. The `SHIFT` primitive operates on two adjacent partitions u and v, with $\pi(v) = \pi(u) + 1$. In a `SHIFT`, either the largest element from partition u is reassigned to partition v, or else the smallest element from partition v is reassigned to partition u. The `REORDER` primitive modifies the permutation π by swapping an empty partition u with either its successor or its predecessor. The cost (i.e. number of relabeled elements) of a `SHIFT` is 1, while the cost of `REORDER` is 0. However, before `REORDER` can be applied, a partition must be emptied out using a series of `SHIFT`s.

The following example demonstrates why the `REORDER` operation is useful. Suppose that X initially consists of 1000 elements with keys $1, 2, \ldots, 1000$, and let $n = 100$. Suppose that initially the load is perfectly balanced, with keys $(10(i-1), 10i]$ assigned to partition i. Now imagine that the elements with keys $1 \ldots 5$ and $11 \ldots 15$ are deleted and new elements with keys $1001 \ldots 1010$ are inserted. This creates a load imbalance, with partitions 1 and 2 containing only 5 elements while partition 100 contains 20 elements. To rebalance the load using only the `SHIFT` operation would incur a relabeling cost of 985. Consider the following alternative procedure: `SHIFT` the 5 remaining elements in partition 1 to partition 2; `REORDER` the partitions so that partition 1 follows partition 100, i.e., $\pi = (2, 3, \ldots, 100, 1)$; then `SHIFT` the 10 largest elements from partition 100 to partition 1. This restores load balance while relabeling only 15 elements.

We define the *level* of a partition u as $I(u) = \lfloor \log_\delta L(u) \rfloor$, where $\delta = (1+\epsilon)^{1/3}$. (Let $I(u) = 0$ when $L(u) = 0$). Also, define the *neighborhood* of a partition u as $N(u) = \{v \in [1, n] : |\pi(v) - \pi(u)| \leq z\}$, where $z = \lceil \frac{2}{\delta - 1} + 1 \rceil$. In other words, the neighborhood of u consists of u and its first z predecessors and successors according to π. Similarly, define the *extended neighborhood* of u as $E(u) = \{v \in [1, n] : |\pi(v) - \pi(u)| \leq 2z\}$.

At all times, our algorithm maintains the following two invariants:

- **Invariant 1:** $\max_i I(i) - \min_j I(j) \leq 2$
- **Invariant 2:** $\forall u, \forall v \in N(u), |I(u) - I(v)| \leq 1$.

Invariant 1 states that the partitions fall into at most three levels. This is sufficent to guarantee the desired imbalance ratio of $(1+\epsilon)$. Invariant 2 states that the levels of two partitions in the same neighborhood can differ by at most one. We will say that a partition u is at *maximum capacity* if adding one more element to u would violate one of the invariants, and similarly that u is at *minimum capacity* if removing one element from u would violate an invariant.

Our algorithm performs relabeling only when an insertion or deletion changes the level of a partition, causing one of the invariants to be violated. Let u denote the partition whose level changed. Suppose for now that the level of u increased due to an insertion, violating an invariant. (Deletion is similar and will be discussed later.) There are two cases: (1) $\exists u' \in E(u)$ such that u' was not at maximum capacity before the insertion, and (2) $\forall u' \in E(u)$, u' was at maximum capacity.

Case 1. In this case, we reduce $L(u)$ by 1 and increase $L(u')$ by 1 in order to restore the invariants. This can be done be a series of at most $2z$ `SHIFT` operations; e.g. if $\pi(u) < \pi(u')$ then we shift one element from u to its successor, then one element from $succ(u)$ to $succ(succ(u))$, and so on until u' is reached.

Case 2. In this case, we cannot restore the invariants using a small number of local `SHIFT`s, so instead we

use the following procedure:

1. Select the most lightly-loaded partition v. It must be the case that $I(v) = I(u) - 3$. (Otherwise, either u or one of the partitions in its extended neighborhood would not have been at maximum capacity.) We will define $r = \delta^{I(v)}$, implying that $r \leq L(v) < \delta r$.
2. Using a series of `SHIFT`s, empty out partition v, dividing its load equally among the other nodes in $N(v)$. This increases the load for each partition in $N(v) \setminus \{v\}$ by at most $\left\lceil \frac{\delta r}{z} \right\rceil$ and at least $\lfloor \frac{r}{2z} \rfloor$, since $r \leq L(v) < \delta r$ and $z \leq |N(v) \setminus \{v\}| \leq 2z$. By Invariant 2, no partition in $N(v)$ had load greater than $\delta^2 r$ prior to this step. Thus, after this step, $\forall w \in N(v), w \neq v, r + \lfloor \frac{r}{2z} \rfloor \leq L(w) \leq \delta^2 r + \left\lceil \frac{\delta r}{z} \right\rceil$. The cost of this step is bounded by $\delta r z$, since load of at most δr is transferred over distance of at most z.
3. The previous step may have caused a violation of Invariant 2 by increasing the level of some partition(s) in $N(v)$. While $\exists w \in N(v), \exists y \in N(w), y \neq v$ with $I(w) = I(y) + 2$, transfer load from w to y using a series of `SHIFT`s. The amount of load so transferred, for each partition w, is no more than the amount gained by w in step 2, and the distance over which it is transferred is at most z, so the total cost of this step is also bounded by $\delta r z$. (Note that y's load increase cannot cause further invariant violations.)
4. Using a series of `REORDER` operations, move v next to u so that $\pi(v) = \pi(u) + 1$.
5. Use `SHIFT` operations to evenly redistribute the load across the partitions $N(u) \bigcup \{v\}$. After this step, $\forall w \in N(u) \bigcup \{v\}, \left\lfloor \frac{\delta r z + \delta^2 r}{z+2} \right\rfloor \leq L(w) \leq \left\lceil \frac{\delta^3 r(2z+1)+1}{2z+2} \right\rceil$. The total cost of this step is bounded by $\delta^3 r z$.

The choice of $z = \lceil \frac{2}{\delta - 1} + 1 \rceil$ guarantees that after rebalancing, for sufficiently large r, $r + \Theta(r) < L(w) < \delta^3 r - \Theta(r)$ for all partitions $w \in N(u) \bigcup N(v)$.

Deletions are handled in a symmetric manner; when the level of partition u decreases due to a deletion, causing an invariant violation, then either load is transferred to u from some partition in its extended neighborhood that is not at minimum capacity, or else u is emptied, reordered to be adjacent to the most heavily-loaded partition v, and refilled by evenly redistributing the load among $N(v) \bigcup u$.

The following theorem summarizes the performance of our algorithm:

Theorem 3.1 *For any given $\epsilon > 0$, the previously described rebalancing algorithm, which has constant amortized relabeling cost per insertion or deletion, maintains an imbalance ratio $\sigma < 1 + \epsilon$, assuming that $|X| > c$, for some constant c.*

Proof Sketch: The fact that $\sigma < 1 + \epsilon$ follows directly from Invariant 1.

The constant amortized cost can be demonstrated as follows: each Case 1 rebalancing operation incurs cost at most $2z = O(1)$. Each Case 2 rebalancing operation incurs cost at most $2\delta r z + \delta^3 r z$. However, after Case 2 rebalancing, all the partitions in $N(u)$ and $N(v)$ are $\Omega(r)$ far from maximum / minimum capacity, while the load of all other partitions is unchanged. (The exception is that some partitions in $E(v) \setminus N(v)$ may be modified in Step 3; however, the load of such partitions can only be made further from capacity, never closer.) Thus the next Case 2 rebalancing operation initiated by an insertion or deletion involving a partition $w \in N(u) \bigcup N(v)$ cannot occur until $\Omega(r)$ insertions or deletions have occurred in $N(w)$, or the three system load levels change, implying that the amortized cost of Case 2 rebalancing is $O(\delta z + \delta^3 z) = O(1)$.

References

[1] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database processing. *Communications of the ACM*, 36(6), 1992.

[2] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to P2P systems. In *Proc. VLDB*, 2004.

[3] D. R. Karger and M. Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. In *Proc. SPAA*, 2004.