Efficient Bulk Insertion into a Distributed Ordered Table

Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, Erik Vee,
Ramana Yerneni and Raghu Ramakrishnan

Yahoo! Research
{silberst,cooperb,utkarsh,erikvee,yerneni,ramakris}@yahoo-inc.com

ABSTRACT

We study the problem of bulk-inserting records into tables in a system that horizontally range-partitions data over a large cluster of shared-nothing machines. Each table partition contains a contiguous portion of the table's key range, and must accept all records inserted into that range. Examples of such systems include BigTable [8] at Google, and PNUTS [15] at Yahoo! During bulk inserts into an existing table, if most of the inserted records end up going into a small number of data partitions, the obtained throughput may be very poor due to ineffective use of cluster parallelism. We propose a novel approach in which a *planning phase* is invoked before the actual insertions. By creating new partitions and intelligently distributing partitions across machines, the planning phase ensures that the insertion load will be well-balanced. Since there is a tradeoff between the cost of moving partitions and the resulting throughput gain, the planning phase must minimize the sum of partition movement time and insertion time. We show that this problem is a variation of NP-hard bin-packing, reduce it to a problem of packing vectors, and then give a solution with provable approximation guarantees. We evaluate our approach on a prototype system deployed on a cluster of 50 machines, and show that it yields significant improvements over more naïve techniques.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*parallel databases*

General Terms

Performance

Keywords

bulk loading, ordered tables, distributed and parallel databases

1. INTRODUCTION

The primary recipe for scalability in many data management systems is to horizontally partition data over a massive cluster of shared-nothing machines. The PNUTS system [15, 10] that we are building at Yahoo! will employ horizontal partitioning in order to handle the enormous scale of data collected, processed, and served by Yahoo!'s internet properties. Systems that scale similarly by horizontal partitioning include Amazon's Dynamo [12], and Google's BigTable [8]. There are two horizontal partitioning schemes commonly employed: *hash* partitioning and *range* partitioning. We study the problem of bulk inserting records into tables that are range partitioned over a shared-nothing cluster. While a large amount of work has been done on bulk insertion, these efforts have primarily focused on single node environments, such as warehousing (see Section 7).

Bulk insertion is a common operation. For example, a system serving shopping listings might receive an aggregated, daily feed of new items from partner vendors, which must then be bulk inserted into the live, serving (operational) table. Similarly, consider a system that hosts and serves web logs for analysis. There may be an hourly process that collects logs from all web servers worldwide, and bulk inserts them into the main database. Due to the frequency and computational demands of bulk insertion operations, it is important that the system handle them efficiently, and that regular workload processing is minimally affected while bulk inserts are in progress.

A naïve approach to bulk inserts is to handle them as a collection of regular inserts. However, in our system, data is *range partitioned* over a cluster of machines, i.e., each partition corresponds to a range of primary-key values and holds all records whose primary keys lie in that range. Bulk inserts can therefore be very unevenly distributed among the existing data partitions. For example, in a system that serves shopping listings, the listings may be range partitioned by timestamp. Thus, when the feed of new items arrives each day, all inserts go to the last partition (which holds the most recent timestamp range).

This problem of uneven distribution of insert load is alleviated if data is *hash partitioned* instead of range partitioned, i.e., each record's primary key is hashed to determine the partition to which that record goes. In contrast to range partitioning, however, hash partitioning does not allow efficient ordered access to data. For example, we might need to do a range scan over the listings in order to expire all those more than 7 days old. In the remainder of this paper, we only consider systems that perform range partitioning.

Thus, in our system with range partitioning, if bulk inserts are done naïvely, skewed inserts might severely overload the machines that host affected partitions. The throughput would hence be limited to that of a few machines, and the far greater aggregate throughput of the entire cluster would not be realized. Moreover, normal workload processing on these overloaded machines would take a severe performance hit while the bulk insert is in progress.

We propose a novel approach to performing bulk insertion operations with the goal of minimizing the total time taken by such operations. The key idea behind our approach is to carry out a *planning phase* before executing any of the inserts that constitute the bulk operation. The planning phase gathers statistics over the data to be bulk inserted, and uses those statistics to prepare the system to gracefully handle the forthcoming insert workload. This preparation consists of:

- **Splitting**: Existing partitions that would become too large after the bulk insert must be split into smaller partitions. The smaller partitions should be such that they are able to handle the forthcoming inserts without becoming too large.

- **Balancing**: Partitions may need to be moved among machines such that the forthcoming insertion load will be well-balanced. Essentially, some data partitions should be moved away from machines that will see a large fraction of the inserts to those that will see a smaller fraction.

Because moving a partition between machines involves copying of data, it can be fairly expensive. Hence there is a tradeoff between the cost of moving partitions among machines on the one hand, and the resulting throughput gain due to a well-balanced insert workload on the other. To arrive at a good point in this tradeoff in a cost-based manner, the planning phase must solve an optimization problem. We formulate this optimization problem and show that it is NP-hard. We also propose an approximate algorithm for solving it. A component of this algorithm is the novel problem of bin-packing vectors, for which our algorithm provides provable approximation guarantees.

Developing a complete solution to bulk insertion also has several system-level challenges: The (external) platform the client uses to store the data prior to bulk insertion may not provide high throughput access, thus limiting the eventual throughput of the bulk insert. It may also not provide a sampling interface to allow our planning phase to gather statistics. Lastly, if the client-issued ordering of the records to be inserted is not randomized, then irrespective of the planning performed by us, the inserts may only touch a few partitions at a time, resulting in poor throughput.

To address these challenges, we optionally allow the client to *stage* data into a temporary (unordered) table managed by our system. During the staging phase, data is transferred from the client machines to machines owned by our system; we piggyback statistics gathering and randomization of the data order (to ensure a well-balanced insert load) to this step.

The rest of the paper is organized as follows:

- We give a high-level overview of PNUTS, our massively distributed data serving platform, and make the case for bulk insertion (Section 2). Although we describe PNUTS for concreteness, our techniques are general enough to be applied in any system that employs range partitioning of data over shared-nothing machines.

- We then introduce the bulk insertion framework, and give an overview of our three-phase approach to bulk insertion: staging of data, planning for the bulk insertion, and the actual insertion operations (Section 3).

- In Section 4, we describe the planning phase, formulate the optimization problem that must be solved, prove its hardness, and propose a method for solving it that is near optimal in practice.

- We also propose a number of additional bulk operations beyond insertion to which our techniques can be applied (Section 5).

- We report experimental results of evaluating our technique on a prototype system deployed over 50 machines (Section 6).

- Finally, in Section 7, we survey related work.

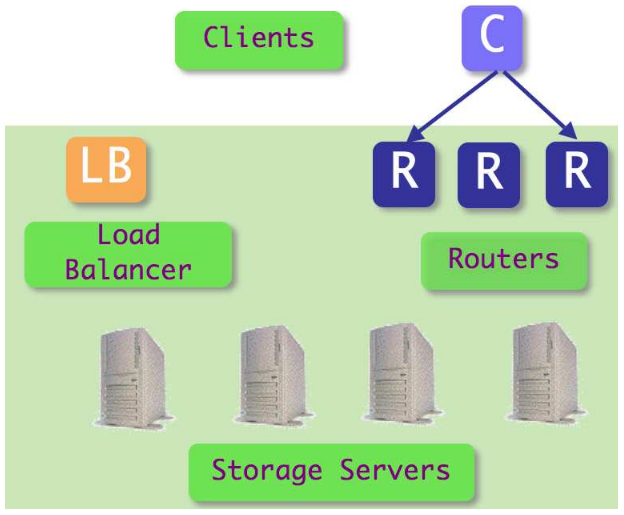

Figure 1: Simplified Architecture Diagram of a PNUTS Region

2. SYSTEM OVERVIEW

We are building the PNUTS [15, 10] system to serve as the backend to Yahoo!'s live web properties. PNUTS is designed to provide low-latency, high-throughput access to very large tables. Typical applications may have tables with anywhere from tens of millions to billions of rows, with records varying in size from a few hundred bytes to tens or hundreds of kilobytes. Most of our applications are internet applications and the workload consists of a majority of reads and writes against single records, and occasional range queries over the data.

A highly simplified architecture diagram of a PNUTS *region* (only the components relevant to this paper) is shown in Figure 1. In a region, data tables are partitioned over multiple physical machines (*storage servers* in the figure), so that many requests may be processed in parallel. All machines in a region are interconnected over a LAN. PNUTS

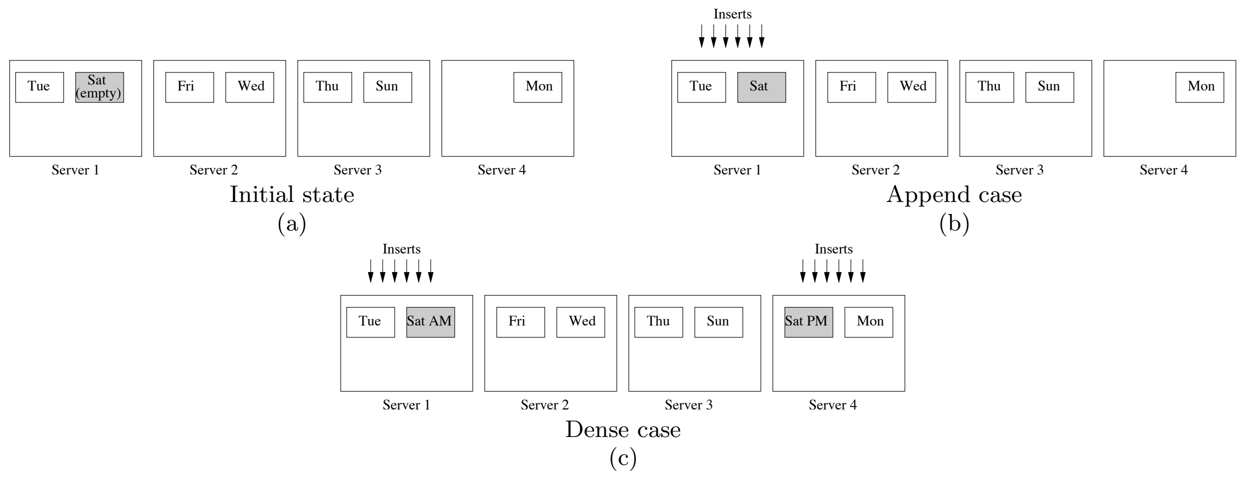

Figure 2: Bulk insert example. Partitions are labeled with the time range of the contained data.

targets scales of several thousand storage servers. Using such a large number of machines is the only way to provide the aggregate RAM, CPU and disk seek capacity to keep up with large-scale internet workloads. Not shown in the figure, data tables are also asynchronously replicated to multiple, geographically distant regions both for disaster recovery and to provide low latency access for users all over the world. Different regions are interconnected over a WAN.

A typical storage server will have hundreds of partitions, which allows us to do fine-grained load balancing by moving individual partitions from overloaded servers to underloaded servers. The *load balancer* is a separate server that maintains the current assignment of partitions to storage servers. This assignment is flexible: any partition can be assigned to any storage server, making it easy to load balance by moving partitions between servers. Load balancing is important during normal operation to alleviate hotspots, and during bulk insert to allow us to spread incoming inserts evenly over many machines. Also, if a server fails, we can reassign its partitions by spreading them over multiple live servers (after recovering them from a replica).

The assignment of partitions to storage servers is cached by multiple *routers* (application-level software servers). *Clients* do not connect directly to storage servers, but instead connect to a router, which looks up the requested key or keys in the partition assignment and routes the request to the appropriate storage server. In the case of a range query which spans multiple partitions, the router sends the request to multiple storage servers, collects the results, and returns them to the user. This level of indirection isolates clients from changes in the underlying partition boundaries or server assignments.

Our data tables are sorted, and are *range-partitioned*. Different partitions contain disjoint subranges of the table's primary key. Partitions can have flexible sizes, unlike disk pages in a traditional database, and different partitions can have different sizes. However, we typically limit the maximum size of a partition, to limit the cost of moving and recovering individual partitions. Since internet applications typically read and write single records, they are frequently built on hash-partitioned database systems [12], especially since hashing provides good load-balancing properties. However, many of our applications occasionally require range queries in addition to point lookups, e.g., a user on a listings-management website might ask for all listings from 7 days ago until now. Range partitioning a table on the primary key allows us to answer such range queries efficiently[1], while hash partitioning does not. Of course, range partitioning only provides a single sort order over the data, and for other sort orders we must construct a secondary index. We discuss secondary indexes and how they are affected by bulk insertions in Section 5.

2.1 Bulk Insert Challenges

In a range partitioned database system, we must be careful to avoid overloading individual storage servers when a particular portion of the range is hot. Consider for example a shopping site that aggregates listings of items for sale from multiple sources. Figure 2a shows an example of keeping 7 days worth of data in range partitions. In this simple example, each day has a partition. Each day, we would receive new listings from each external source. The first feed on a given day (say, Saturday) would provide listings appended to the end of the range. The result is that the partition corresponding to the end of the range (shaded in the figure) is the target of all of the inserts, as shown in Figure 2b. As this partition becomes overfull, it must split, and if the load balancer operates correctly, split partitions must be moved to other storage servers. The result is that both the storage server and the load balancer become overloaded trying to handle the effect of the inserts, even as other storage servers remain underloaded. If normal database operations continue to proceed during the bulk insert, those operations that touch the overloaded server will experience long latencies. The bulk load itself will also take a long time.

Subsequent feeds on the same day may also cause overloading. Even if the load balancer is able to spread the split partitions over multiple servers, the "Saturday" range is still concentrated on a few partitions on a few servers. As shown in Figure 2c, more inserts into the "Saturday" range will overload those servers, cause more splits, and require more load balancing.

This example serves to illustrate several aspects of the

[1]If the queried range is large, answering it might still involve going to multiple storage servers. However, each subrange corresponding to a partition still achieves locality.

bulk insert problem. First, a simple solution of using the normal record-at-a-time insertion process will be inefficient in many common cases, overloading servers and failing to take advantage of system parallelism. Second, during bulk insertion, it is inevitable that we must split and move data partitions. The key to our solution is to proactively split and move partitions to avoid overloading servers during the actual insertion of data.

We also observe that there are several different scenarios of bulk insert:

- *Initialize*: We bulk load an empty table with data for the first time.
- *Append*: We insert data at the end of the range of an already populated table.
- *Sparse*: We insert a small amount of data into a range already populated with data. We call a bulk insert "sparse" if the data is small enough to fit into existing partitions with little or no splitting.
- *Dense*: We insert a large amount of data into a range already populated with data. We call the bulk insert "dense" if the insertion of the data would cause many splits of partitions.

The most challenging situation is the *dense* case: we must pre-split partitions to support efficient parallel inserts, but we must trade off this efficiency with the cost of transferring existing data between servers. The *sparse* case is easier, since it does not require splits and thus does not require existing data to be moved. The *initialize* and *append* cases are similar in that the range being inserted does not have existing data; thus, while splits are required, existing data does not need to be moved. Though *sparse*, *initialize* and *append* are special cases of bulk insertion, our bulk insert utility handles all four cases uniformly, generating the most efficient solution for each.

3. OVERVIEW OF OUR APPROACH

This section presents the overview of our approach for invoking bulk insertion, consisting of:

1. Staging the input data
2. Planning
3. Inserting records

3.1 Staging Input Data

The data to be bulk inserted can come from a variety of sources, including an external database, a periodic feed, a file system, or even another table in our own system. In many of these cases, the data will not be available in a form factor that supports fast bulk loading. In particular, we want the incoming data to be partitioned across multiple machines, so we can load our storage servers in parallel, and we want to be able to sample the incoming data and choose its ordering. To support these capabilities, we provide a set of *staging servers* which temporarily store the data. Incoming data is transferred to these staging servers, sampled, and then bulk inserted. Note that staging is simpler than bulk loading, since we do not manage multiple partitions per server or keep the staged data in a totally sorted order. If the source of bulk loaded data supports the correct form factor, we can avoid the staging step; in particular, bulk loading one table from another table in our system can bypass staging.

3.2 Planning Phase

Our approach to dealing with the challenges of Section 2.1 is to execute a planning step to determine how to minimize the total time for bulk insertion, including the time to move existing data and the time to insert new data. The result of the planning phase is a schedule of partition splits and partition moves (from one storage server to another) that take the system from its current state to one in which underfull partitions are spread across storage servers. Pre-splitting existing partitions into underfull partitions and moving those underfull partitions between servers makes room on every storage server for the incoming inserts, so that all storage servers receive equal insert load and parallelism is maximized. Recall that moving partitions is inevitable during bulk insertion; by moving the partitions before the actual insertion (e.g., when they are still underfull) we save time compared to the one-at-a-time insertion approach.

Spending more time moving underfull partitions between storage servers may better balance the resulting insertion load; however, this extra move time must be balanced with the savings in insertion time. Thus, the planning step involves an optimization problem. In Section 4, we analyze the optimization problem formally and show that it is NP-hard. We then develop an approximation algorithm that is close to optimal in practice.

We now explain the mechanics of the planning process (deferring the approximation algorithm itself to the next section). The planning is carried out at a single *bulk load server*, which interacts with the rest of the system to gather necessary information, execute the split and move plan, and trigger the actual bulk insertion. In our implementation, the bulk load server runs on the same machine as the load balancer (recall Figure 1). The planner is given the following inputs:

- The current set of partition boundaries
- The current assignment of partitions to servers
- A set of samples of record keys from both the existing and new data
- The maximum number of records allowed per partition

The samples enable us to estimate which existing partitions will likely have more than the maximum number of records after the bulk insertion, and thus require pre-splitting. Samples also allow us to choose split points for those partitions (e.g., new partition boundaries), and to compute the expected number of inserts per storage server once our split and move plan is executed.

The planner uses these inputs to produce a two stage plan: 1) all of the partition splits, and 2) all of the partition moves. Although the splits and moves can be executed in parallel, we plan them separately. We choose partition splits so that the partitions, after insertion, are all of roughly equal size. Equally sized partitions simplify load balancing and recovery during normal operation. The number of samples of new and existing data necessary to choose a final partitioning where partitions are expected to be equally sized is given by a result from Seshadri and Naughton [24].

Given the set of partitions that would exist after the splits execute, the move planning process determines which partitions to move between storage servers. These moves can be carried out in parallel, although the bandwidth capacity of each storage server places a limit on how many concurrent partition moves a storage server can participate in. Thus, more moves in the plan result in a longer move time.

Once a plan is computed, the bulk load server contacts each storage server and initiates the necessary splits and moves.

3.3 Insertion

As the bulk load server receives success notification for its split and move commands, it updates the partition assignment table; splits lead to new partitions, while moves modify the machine to which a partition maps. Once all splits and moves complete, the bulk load server triggers bulk inserts by sending the current partition assignment table to each staging machine. A staging server uses the table to determine where to transmit each of its records. To maximize throughput, each staging machine bundles multiple records to be inserted into single calls to storage servers.

3.4 Concurrent Workloads

Our algorithm assumes that the assignment of partitions is fixed and does not need to change while the bulk insertion is in progress. In practice, however, due to changes in load patterns, some balancing of partitions may be necessary during the bulk insertion. Since the load balancer runs on the same machine as the bulk load server, it can use information about the current bulk insertion to produce a balancing plan that takes the ongoing operation into account and is least disruptive. The techniques for doing so are left as future work.

Our algorithm optimizes a single bulk insertion at a time. Online planning and scheduling of multiple bulk operations while previous ones are still running is a topic of future work.

4. PLANNING AND OPTIMIZATION

This section discusses the planning phase, which dictates and executes a series of partition splits and moves for the purpose of minimizing time spent both planning and inserting. The planning input consists of a set of machines, M, and a set of partitions, P. The ith machine is denoted m_i and the jth partition is denoted p_j. π denotes the maximum number of records per partition.

4.1 Partition Splits

Partition splitting, an operation that is local to a machine, is not a major contributor to bulk load running time. We briefly discuss it here and show how it frames the problem of minimizing bulk load time. Recall we gather sample keys for both existing and new records. For a given pre-existing partition p_e with r_e records, we check the number of new record samples falling within p_e's boundaries and estimate the number r_n of new records falling within it. If $r_e+r_n \leq \pi$, p_e is left alone. Conversely, if $r_e + r_n > \pi$, p_e splits. The number of resulting partitions must be at least $\frac{r_e+r_n}{\pi}$. Given p_e key boundaries l_e and h_e, we consult the subset of samples in that range, and divide the subset into $\frac{r_e+r_n}{\pi}$ evenly-sized segments; the new set of boundaries is $l_e, b_{e_1}, b_{e_2}, \ldots, b_{e_i}, h_e$. This partitioning is executed as a series of splits on p_e: $\mathsf{split}((l_e, h_e), b_{e_1}), \mathsf{split}((b_{e_1}, h_e), b_{e_2}), \ldots, \mathsf{split}((b_{e_{i-1}}, h_e), b_{e_i})$.

Sampling frequency impacts this approach's effectiveness. In general, if frequency is too low, evenly spacing boundaries on samples will not result in actual even boundaries. In practice, we find we can set sampling frequency around 1% with no ill effects. This is because machines each hold many partitions and can absorb skew in partition size. That is, a machine with both unexpectedly large and small partitions will have taken on the same total number of records as a machine with no skewed partitions.

Once all splits execute, each partition p_j can be described as a quadruple: $< e_j, n_j, h_j, d_j >$, where e_j and n_j are the estimated numbers of existing records and new (to-be-inserted) records on p_j, respectively, h_j is the machine on which p_j currently resides by default, and d_j is the currently assigned destination machine to which p_j will move prior to insertions; $h_j = d_j$ initially. As moves are planned, the destinations are updated, so that the set of quadruples at all times encodes the entire plan of splits and moves.

4.2 Bulk Insert Minimization

Maximum partition size π dictates the set of splits that occur and then serve as input to the record move/insert (RMI) problem. The goal is to minimize the time from when bulk insert is initiated to when all new records are stored in ordered partitions. The most time-intensive operations in this process are those that transmit large amounts of data across machines, namely partition moves and record inserts. RMI minimizes time spent doing these. Note that we may not always want the bulk insert to proceed as quickly as possible, in order to not overburden the system to the point it cannot easily handle normal operations. The key to fast bulk insert is maximizing parallelism across machines. Therefore, even if we choose to limit the pace at which insertion proceeds, the optimizations described here will still minimize insert time under the limitation, as well as equally easing the burden on each machine.

4.2.1 Problem Definition

Each machine m_i pays move cost o_i and insert cost s_i, where o_i is the number of records moved to and from m_i within partitions during the move process, and s_i is the number of records inserted on m_i during the insert stage. For each m_i, we define two relevant subsets of P: O_i, the set of partitions contributing to m_i's move cost, and S_i, the set of partitions contributing to m_i's insert cost.

$$\begin{aligned} O_i &= \{p_j \in P \,|\, h_j \neq d_j \wedge (h_j = m_i \vee d_j = m_i)\} \\ S_i &= \{p_j \in P \,|\, d_j = m_i\} \end{aligned}$$

We then define m_i's costs as: $o_i = \sum_{p_j \in O_i} e_j$ and $s_i = \sum_{p_j \in S_i} n_j$.

The planning and insertion phases are cleanly divided; that is, all partition moves complete before insertions begin. To minimize total bulk insert time, we minimize the sum of time spent moving and time spent inserting. These times are determined by the machine taking the most time in each phase. Suppose $o_{max} = max\{o_i | m_i \in M\}$ and $s_{max} = max\{s_i | m_i \in M\}$. Our optimization goal is to minimize $(o_{max} + s_{max})$.

The minimization goal encodes that we should balance insert cost across machines, while respecting the tradeoff that shifting partitions introduces move cost. Our flexibility in this balance comes from our ability to move partitions between machines. Intuitively, we want to move partitions from those machines expecting a large proportion of the inserts to those machines expecting a small proportion.

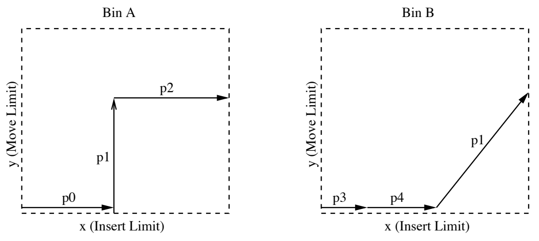

Figure 3: Two machine "bins" with partition "vectors."

We next define the record move/insert problem. To a first approximation, intuitively, we solve this problem by moving partitions from machines expecting many inserts to machines expecting few. Delving deeper, we introduce the vector packing problem, whose solution dictates how a set of partitions are re-assigned. We fit this into the larger context of off-loading and on-loading partitions.

4.3 Record Move/Insert Minimization Problem

The record move/insert (RMI) minimization problem is a variation of the two dimensional dual bin packing problem, which contains a fixed number of bins that must all have the same size. The problem is to find the smallest possible bin size that allows all items to be packed. In our setting, each m_i is a bin, and must be large enough in both the move and insert dimensions to accommodate the maximum move and insert costs across all units. Each p_j is a vector (x_j, y_j) defined by the corresponding quadruple. x_j simply corresponds to n_j. If $h_j = d_j$, no partition move is executed, and $y_j = 0$. Otherwise, $y_j = e_j$ and, additionally, another vector $(0, e_j)$ is added to h_j to account for the cost of offloading p_j from it. Clearly, y_j's behavior is cumbersome; we show how to overcome this in the remainder of the section. Finally, the assigning of vectors to bins does not follow standard bin packing. The first packed vector starts in the lower left corner of the bin. Each subsequent vector's start point is placed incident to the preceding vector's endpoint. A bin is full once its last vector touches the upper right corner. This pattern is shown in Figure 3. Observe vectors $p0$ and $p2$ are native to Bin A, while $p3$ and $p4$ are native to Bin B. None of these vectors contribute move cost to their respective bins. $p1$, native to A, in contrast, has moved to B. Therefore it accrues move cost on A for offloading (but no insert cost) and accrues both move and insert costs on B. Had $p1$ not been moved from A, the added insert cost on A would have caused its sequence of vectors to overrun A's insert limit. Vector packing, along with move behavior, necessitates a novel algorithm to find minimum bin size.

It is straightforward to show that the RMI problem is NP-hard, by reducing to the special case where moves are free. All partitions are placed in a pool, unassigned to machines. We are then left with the standard one dimension dual bin packing problem, with the goal to minimize maximum insertion cost over each machine. This simpler version of bin packing is known to be NP-hard.

4.4 The Vector Packing Problem

We now introduce the novel *vector packing problem*, which dictates how partitions are assigned to underfull machines and which lies at the core of our problem (following that, we show how to offload partitions from overfull machines). We will give an efficient algorithm for the vector packing problem, providing a near-optimal solution.

Let U denote a set of bins $B_1, .., B_{|U|}$. Each bin is identified with a vector, $B_i = (u_i, v_i)$ with positive entries. Let P denote a set of vectors $p_1, p_2, ..., p_{|P|}$, where each $p_i = (x_i, y_i)$ with nonnegative entries. The vector packing problem asks, given U and P, to find a one-to-one assignment of vectors to bins such that $\sum_{p \in A_j} p \leq B_j$ for all j, where A_j is the set of vectors assigned to bin B_j, and we take $\leq$ to mean less than or equal in each dimension. Notice that although this is related to the multiple-choice knapsack problem, the bin-packing problem, and the general assignment problem, the vector-packing problem is distinct from all of these, and the methods we develop to solve the problem are novel.

We first present our algorithm. For convenience, let $(u, v) = \sum_{B \in U} B$ and let $(x, y) = \sum_{p \in P} p$. We add two dummy vectors, $p_0 = (u - x, 0)$ and $p_{|P|+1} = (0, v - y)$. Think of these dummy vectors as representing the amount of slack we have in each dimension. We assume the vectors in P and the bins in U are sorted in increasing order of their slope, y_i/x_i (and v_i/u_i). We denote this slope by $\Delta(p)$ (and $\Delta(B)$).

Since $\Delta(p_0) = 0$ and $\Delta(p_{|P|+1}) = \infty$, they will always be the first and last items, respectively, in the sorted list. When the algorithm below says to assign a dummy vector, we take it as a null operation; intuitively, we are assigning that extra slack to a bin (hence, it will have leftover space in that dimension).

Algorithm 1 The vector-packing ONLOAD algorithm

```
1: Sort bins by slope $B_1, B_2, ...$
2: Sort vectors by slope $p_0, p_1, ...$
3: while $|P| > 0$ do
4:    Let $B$ be the bin with minimum slope.
5:    Set $X = 0$ and $Y = 0$
6:    Find $\ell$ s.t. $\Delta(p_\ell) \leq \Delta(B) \leq \Delta(p_{\ell+1})$
7:    Set $r = \ell + 1$
8:    while $(X, Y) < B$ do
9:       if $(Y + y_\ell + y_r)/(X + x_\ell + x_r) \geq \Delta(B)$ then
10:         Update $X = X + x_\ell$, $Y = Y + y_\ell$, and $\ell = \ell - 1$.
11:      else
12:         Update $X = X + x_r$, $Y = Y + y_r$, and $r = r + 1$.
13:      end if
14:   end while
15:   If we last updated $\ell$, then update $r = r + 1$. Otherwise,
      update $\ell = \ell - 1$.
16:   for all $p_c$ s.t. $l < c < r$ do
17:      Remove $p_c$ from $P$ and assign it to $B$
18:   end for
19:   Remove $B$ from $U$.
20: end while
```

The analysis of ONLOAD requires a concept we refer to as the *envelope* formed by the partition vectors in P, together with the dummy vectors. In Figure 4, we have drawn the vectors $p_1, p_2, ...$ end to end, forming a polygonal arc, which we think of as the lower edge of the envelope. This is a continuous, piecewise linear function (ignoring the last-appearing vectors, which may have infinite slope; we ignore this technicality for ease of exposition), which we denote by $\text{ENVEL}_P(x)$. We have also drawn the upper edge of the envelope. This is formed by placing the vectors $p_{|P|+1}, p_{|P|}, ...$ end to end, essentially giving the upper bound of where the

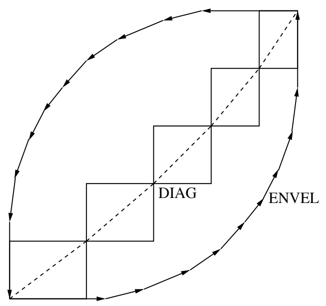

Figure 4: Envelope Envel$_P$(x) surrounding bin diagonals Diag$_U$(x).

vectors can reach. Although it is helpful to think of this upper edge, it will be unnecessary in any of our proofs.

We have also drawn the bins' vectors in the figure. The diagonals of these bins trace out a continuous, piecewise linear function that we refer to as $\textsc{Diag}_U(x)$. Note that $\textsc{Diag}_U(x)$ and $\textsc{Envel}_P(x)$ meet at their endpoints. This is ensured by the choice of the dummy vectors.

It is straightforward to see that there is a solution to the vector packing problem only if $\textsc{Diag}_U(x) \geq \textsc{Envel}_P(x)$ for all x, since the lower envelope edge necessarily lies at or below any viable solution (and the viable solution must pass through the corners of $\textsc{Diag}_U(x)$, when we add dummy vectors to represent the slack in each bin). Although the converse is not necessarily true, it is the case that if $\textsc{Diag}_U(x) \geq \textsc{Envel}_P(x)$ for all x, there is a *fractional* solution. More importantly, we can show that in this case, the ONLOAD algorithm runs correctly.

THEOREM 1. *Let U and P be as described above. If $\textsc{Diag}_U(x) \geq \textsc{Envel}_P(x)$ for all x, then the* ONLOAD *algorithm finds an assignment of vectors to bins such that for all bins B, the sum of the vectors assigned to B is at most B plus $2p_{max}$, where p_{max} is the vector whose x-coordinate (resp., y-coordinate) is the maximum x-coordinate (resp., y) over all $p \in P$. The running time of* ONLOAD *is $O(|P| \log |P|)$. If $\textsc{Diag}_U(x) < \textsc{Envel}_P(x)$ for some x, there is no solution.*

Before proving this, we first state a key lemma.

LEMMA 1. *Let B be the bin with the smallest slope. Suppose there are two consecutive vectors, p, p' such that $\Delta(B) = \Delta(p + p')$ and $p + p' \leq B$. Let U' denote the set of bins with B replaced by $B - p - p'$, and let P' be P with p and p' removed.*

Then if $\textsc{Diag}_U(x) \geq \textsc{Envel}_P(x)$ for all x, then $\textsc{Diag}_{U'}(x) \geq \textsc{Envel}_{P'}(x)$ for all x.

PROOF. First, note that because bins are sorted by slope, for all $u < v$, $\textsc{Diag}_U(u + x) - \textsc{Diag}_U(u) \leq \textsc{Diag}_U(v + x) - \textsc{Diag}_U(v)$. Setting $u = 0$,$v = x_1$, and $x = x_2$, we get $\textsc{Diag}_U(0+x_2) - \textsc{Diag}_U(0) \leq \textsc{Diag}_U(x_1 + x_2) - \textsc{Diag}_U(x_1)$. That is,

$$\forall x_1, x_2 \geq 0,\ \textsc{Diag}_U(x_1) + \textsc{Diag}_U(x_2) \leq \textsc{Diag}_U(x_1 + x_2) \quad (1)$$

Since $p + p'$ lies on the diagonal for bin B,

$$\textsc{Diag}_U(p_x + p'_x) = p_y + p'_y \quad (2)$$

Let b denote the x coordinate where p first appears in the envelope. When we remove p, p' from the envelope, all points with x coordinate less than b remain unchanged, while points to the right of b are shifted by $-p - p'$. That is,

$$\forall x \leq b\ \textsc{Envel}_{P'}(x) = \textsc{Envel}_P(x) \quad (3)$$
$$\forall x > b\ \textsc{Envel}_{P'}(x)$$
$$= \textsc{Envel}_P(x + p_x + p'_x) - p_y - p'_y \quad (4)$$

When we replace B by $B-p-p'$, all points along the diagonal are shifted by $-p - p'$. That is,

$$\forall x\ \textsc{Diag}_{U'}(x) = \textsc{Diag}_U(x + p_x + p'_x) - p_y - p'_y$$

Finally, because all bin corners are initially contained within the envelope,

$$\forall x\ \textsc{Diag}_U(x) \geq \textsc{Envel}_P(x) \quad (5)$$

To prove the new set of bins are contained in the new envelope, it suffices to prove $\forall x\ \textsc{Diag}_{U'}(x) \geq \textsc{Envel}_{P'}(x)$. First consider the case where $x > b$:

$$\begin{aligned} \textsc{Diag}_{U'}(x) &= \textsc{Diag}_U(x + p_x + p'_x) - p_y - p'_y \\ &\geq \textsc{Envel}_P(x + p_x + p'_x) - p_y - p'_y (\mathit{from}\ 5) \\ &= \textsc{Envel}_{P'}(x) (\mathit{from}\ 4) \end{aligned}$$

Finally consider the case where $x \leq b$:

$$\begin{aligned} \textsc{Diag}_{U'}(x) &= \textsc{Diag}_U(x + p_x + p'_x) - p_y - p'_y \\ &\geq \textsc{Diag}_U(x) + \textsc{Diag}_U(p_x + p'_x) - p_y - p'_y \\ &\quad (\mathit{from}\ 1) \\ &= \textsc{Diag}_U(x) + 0 (\mathit{from}\ 2) \\ &\geq \textsc{Envel}_P(x) (\mathit{from}\ 5) \\ &= \textsc{Envel}_{P'}(x) (\mathit{from}\ 3) \end{aligned}$$

□

We now proceed with the proof of Theorem 1.

PROOF SKETCH OF THEOREM 1. In order to understand the key property of the algorithm, we imagine rewriting lines 9 through 13 of the algorithm. Suppose for a moment that we were allowed to add fractional portions of vectors. Throughout the algorithm, we maintain the invariant that $\Delta(p_\ell) \leq \Delta(B) \leq \Delta(p_r)$. Without loss of generality, assume $\Delta(p_\ell + p_r) \geq \Delta(B)$. Then there is an $\alpha \geq 0$ such that $\Delta(p_\ell + \alpha p_r) = \Delta(m_{min})$. Break p_r into two pieces, one of size αp_r and the other of size $(1 - \alpha)p_r$. Notice that p_ℓ and αp_r now satisfy the requirements of Lemma 1. So assign both p_ℓ and αp_r to bin B. Since the envelope property still holds, we can continue this way until we overrun both the x and y limits of the bin— since we always move along the diagonal, both limits will be overrun in the same step. (Note that overrunning on a particular bin can only help the other bins, so this causes no harm; it is only when we are underfull in a particular dimension that other bins may have to account for the extra slack.)

Of course, we cannot actually break vectors into pieces. It is not hard to see, however, that the procedure described above only has one fractional vector at any given time; if we break p_r into two pieces, then one will be assigned immediately to bin B. The other piece will either be assigned in the next step (in which case p_ℓ is broken into pieces), or it will be broken into two more pieces in the next step. In either case, we still have just one fractional piece.

So, when we completely fill a bin, we simply assign the remaining fractional vector to the bin as well. The amount that we overrun is at most $2p_{max}$; suppose we are barely inside the bin. One step later, we assign two more vectors (as well as the fractional piece), and move outside.

To finish the proof, we show that the ONLOAD algorithm has running time of $O(|P|\log|P|)$: We sort the $|P|$ partition vectors and the $|U|$ machines, taking time $O(|P|\log|P| + |U|\log|U|) = O(|P|\log|P|)$. We store the partition vectors in a linked list. To execute line 6, we simply move forward through the list until we find the appropriate ℓ. Since we process the bins in increasing order of their slopes, this find step always moves forward through the list, hence taking $O(|P|)$ time overall. All of the other steps are clearly $O(1)$ time, since a linked list supports fast deletions. □

4.5 The RMI Algorithm

Having solved the vector packing problem, we now convert the RMI problem to a vector packing one. We then run ONLOAD to find a good solution.

We set a per-machine insert limit of ι. We then call OFFLOAD, an algorithm that guarantees no machine has an insert cost larger than ι (plus π). The OFFLOAD algorithm temporarily and symbolically assigns some partitions to a *common pool*, denoted CP. We then seek to assign those offloaded partitions from CP to machines (which we think of as bins). This problem is simply vector packing, and we run ONLOAD to find a good assignment. Note that before we can run ONLOAD, we first need to set a per-machine move limit, μ, to set both bin dimensions. We discuss how ι and μ are set in Section 4.5.3.

4.5.1 *The* OFFLOAD *algorithm*

The OFFLOAD algorithm takes a per-machine insert limit, ι, as input. Every machine with $s_i > \iota$ is considered overfull, and must offload some partitions to CP. Note that moving a partition vector $p_j = (x_j, y_j)$ to CP from machine m_i dictates that both m_i and the machine to which p_j is eventually assigned pay a move cost of y_j. The pseudocode for OFFLOAD follows.

Algorithm 2 The OFFLOAD algorithm

```
1:  for all $m_i$ do
2:    if $s_i > \iota$ then
3:      Sort $s_i$'s partitions in increasing $y_j/x_j$ order
4:      while $s_i \geq \iota$ do
5:        Remove min partition $p_{min}$ from machine $m_i$.
6:        Put $p_{min}$ in the common pool, CP
7:        Update $s_i = s_i - x_{min}$ and $o_i = o_i + y_{min}$
8:      end while
9:      Reassign $p_{min}$ to $m_i$
10:   end if
11: end for
```

LEMMA 2. *For each overfull m_i, the* OFFLOAD *algorithm reduces the total insert cost, s_i, to at most $\iota + \pi$.*

PROOF. The OFFLOAD algorithm stops when $s_i < \iota$ and adds back the last partition removed. At worst, this last partition contains π new records to be inserted. □

Note the partitions in CP have the smallest possible total move cost, given that each machine has insert cost at most ι. In fact, the partition vectors we choose for CP are good in an even stronger sense: our choices form the lowest possible envelope. That is, for all offloaded sets CP' we could have chosen in the offload phase, $\text{ENVEL}_{\mathsf{CP}'}(x) \geq \text{ENVEL}_{\mathsf{CP}}(x)$ for all x. Since there is a vector packing only if the envelope lies below the bin diagonals, this is optimal.

If the set of partitions over all overfull bins is P, the total running time for OFFLOAD is $O(|P|\log|P|)$. Each overfull bin now takes on at most $\iota + \pi$ records.

4.5.2 *Solving with* ONLOAD

After OFFLOAD has completed, there are a number of partitions assigned to CP. An insert limit, ι, has already been set. Once an upper move limit, μ, is set, we can calculate how much space is left in each underfull machine, for each dimension. We identify each underfull machine m_i with a bin whose vector is $(\iota - \sum_{p_j \in P_i} n_j, \mu)$. (Recall n_p is the insert cost of partition p, and let P_i be the set of partitions residing on m_i at the start of the algorithm.) The set U consists of the underfull machines.

Thus, once μ is chosen, assigning partitions from CP to machines can be done using the ONLOAD algorithm.

4.5.3 *Setting insert and move limits*

We now describe how to set insert and move limits (i.e., ι and μ, respectively) and how they interact with OFFLOAD and ONLOAD. First, for a given bin size B (i.e. choice of ι and μ), if an onloading solution exists, we find one such that each bin has cost less than or equal to $B + 2p_{max}$. ONLOAD, which from Theorem 1 assigns bins as many as two partitions over capacity, determines this result.

The remaining challenge is in choosing ι and μ. Searching all combinations exhaustively is not a viable option. Instead, we only explicitly adjust ι. Once ι is chosen, the set CP (as computed by OFFLOAD) is determined, and $\text{ENVEL}_{\mathsf{CP}}(x)$ is fixed (ignoring the vertical dummy vector). Note, however, that as we vary μ, the bin vectors will vary as well, hence $\text{DIAG}_U(x)$ is implicitly a function of μ; write it as $\text{DIAG}_U(x; \mu)$.

We know a solution to the vector packing problem exists only if $\text{DIAG}_U(x; \mu) \geq \text{ENVEL}_{\mathsf{CP}}(x)$ for all x; moreover, ONLOAD finds a packing if the inequality is satisfied. μ should ideally be set to the smallest value possible such that the inequality holds. Denote the maximum of this μ and the move cost incurred during OFFLOAD by $\text{MINMOVE}(\iota)$. Note that once the machines and partition vectors are sorted by slope, finding $\text{MINMOVE}(\iota)$ can be done in linear time.

The smallest possible value of ι is $\frac{1}{|M|}\sum_i s_i$, which occurs if inserts are perfectly balanced across all machines. Also notice that if $\iota = \sum_i s_i$, all machines trivially meet the insert limit. These serve as lower and upper bounds on ι settings worth checking. We binary search through this space of potential ι values, looking for the smallest $\iota + \text{MINMOVE}(\iota)$ value. We output this minimum, along with the assignment that attained it.

Testing ι values this way means we rely on OFFLOAD to fill CP, which in turn determines μ. Note that once we have chosen the set of partitions to move, the ONLOAD algorithm finds a neary optimal rearrangement. OFFLOAD, however, is itself a heuristic, and may not necessarily find the best set of partitions to move (i.e. the best CP). It is possible, by shuffling partitions between underfull machines, to find a better CP that leads to a lower μ. Nevertheless, it is intuitively an effective heuristic that performs well in practice. By removing those partitions with lowest y_j/x_j (move to insert ratio) we get close to the smallest number of moves from each machine possible.

The overall running time of the optimization is dominated by sorting, namely of the partitions in each overfull machine, the partitions in CP, and the underfull bins. We need only do each sort once, however; as ι changes, none of these sort orders change. If the set of all partitions in the system is P, running time is $O(|P| \log |P|)$.

5. INTERACTION WITH INDEXING AND REPLICATION

5.1 Secondary Indexing

Alongside a primary table sorted by primary key, we may want to maintain one or more secondary index tables sorted by other attributes (with primary key appended to maintain unique keys). In one-at-a-time insertion, for each record added to the primary table, a corresponding record must be generated for each secondary table. Generation requires parsing the original record payload, extracting the index attributes, and writing new records. For bulk insertion we simply include this same functionality in the bulk load server. In the planning stage, when we gather samples from the staging servers and storage servers, we parse them for their secondary attributes, generating sets of samples for each secondary index. These samples, as usual, dictate partition splits that serve as input to the move/insert problem. From this point, we pool all partitions together, without care to which tables they correspond. We then utilize our algorithm to plan moves, in order to minimize total move and insert time.

5.2 Bulk update and delete

It is possible to extend our techniques to deal with bulk updates and deletes. Bulk updates can be viewed as a simplification of bulk insert where the amount of data in partitions is the same both before and after the bulk operation. In this case, it is necessary to move (but not split) partitions to load balance updates across servers. We can still use our RMI algorithm, substituting *update cost* for *insert cost*. Bulk delete proceeds similarly to bulk update. The number of deletes per partition is bounded by partition size. We once again omit partition splitting and run RMI with *delete cost* replacing *insert cost*; because deletes need only list the record key, delete cost is lower than insert.

5.3 Replication

A common approach to making distributed databases robust to failure, both in terms of recovery and minimizing unavailability, is to replicate tables across regions. When a bulk insert is initiated in one region, the inserts must be applied to all regions. In PNUTS, the set of partitions constituting a table are identical in each region. The partition splits are therefore identical as well. Partitions may, however, be differently distributed among machines in each region. Using the same set of samples, we must run a separate instance of the RMI algorithm for each region and produce a partition assignment map for each.

Once splits and moves complete, a simple approach for insertion is to do them in one region, and rely on the system's normal replication mechanism to propagate the new data to other regions. However, this will likely overload the replication mechanism, and will impact normal clients of the system. A better option is to replicate the staging in all the regions by transferring the incoming data to stager units in each region, and bulk load each region separately. This requires using a protocol such as GridFTP [1] for effectively transferring large datasets between regions to stager servers.

This discussion still leaves open a number of cross-region issues, such as assuring that each new record is either successfully inserted in each region or else fails. We leave such consistency issues as future work.

6. EVALUATION

In this section, we provide experimental validation of our proposed approach for bulk insertions. We have prototyped, deployed, and experimented with our algorithm on a cluster consisting of up to 50 storage servers, 10 stagers, and 5 routers. Each machine in our cluster had two Intel 2.13GHz processors, and 4GB of RAM. Each of the storage servers runs a local Berkeley Database for storing the partitions assigned to it.

In this section, and in our charts, we refer to our algorithm for bulk insertions as `Planned Bulk`. We compare `Planned Bulk` to the following more naïve approaches:

- One-at-a-time, Random Order (`OAT-Random`): In this approach, the data to be bulk inserted is in random key order. Records are simply inserted one-at-a-time into the system by multiple clients simultaneously, relying on the load balancer to balance load. For a fair comparison, in this approach we always use the same number of clients as the number of stagers used by `Planned Bulk`.

- One-at-a-time, Sorted Order (`OAT-Sorted`): This approach is the same as `OAT-Random` except that the data to be bulk inserted is sorted by key. This approach is meant to capture the relatively common case where the data to be bulk inserted is received in sorted order, and is inserted as is.

Our experiments broadly demonstrate that for a wide variety of data distributions and workload patterns:

- `Planned Bulk` performs bulk inserts up to 5-6 times faster than `OAT-Random`, and 10 or more times faster than `OAT-Sorted`.

- `Planned Bulk` scales extremely well as the number of storage servers and the data size increase. `OAT-Random` is somewhat scalable, while `OAT-Sorted` does not scale at all.

- The impact on normal workload processing during bulk insertion is acceptable.

6.1 Experimental Procedure and Parameters

We have experimented with both synthetic and real data and workloads.

Our synthetic data generation works as follows: The key range is first divided into equal subranges. Keys are then chosen from these subranges, where the probability of choosing from each subrange draws from a Zipfian distribution with parameter Z. If $Z = 0$, this process results in a uniform key distribution. As Z becomes higher, keys get increasingly clustered in various parts of the key range. Since it has been observed that web workloads tend to follow a

Parameter	Description
n_i (5 million)	Number of initial records
n_b (1 million)	Number of records bulk inserted
N_S (10)	Number of storage servers
f (1%)	Sampling fraction for `Planned Bulk`
Z (1)	The Zipfian parameter for the key distribution

Table 1: Experiment Parameters

Zipfian distribution [7], our data follows $Z = 1$ by default, although we experiment with other values of Z. The size of the record is fixed at 1KB, and the key length is fixed at 16 bytes.

In all our experiments, we initially loaded a table with a data set having uniformly distributed keys. We then bulk inserted another batch of data into the table using one of the various approaches being compared, and measured the total completion time. The parameters that we varied in our experiments are as shown in Table 1, with default values shown in parentheses. In all our experiments, the system was configured to split partitions when the partition size reached 10MB. The number of routers was fixed at 5.

6.2 Experiments on Synthetic Data

In this section, we report our experiments on synthetic data where we varied the parameters listed in Table 1, and observed their effect on system performance.

6.2.1 Varying the Bulk Load Size

In this experiment we varied n_b, the number of records to be bulk inserted, from 1 million to 5 million. We used the default settings for the rest of the parameters. The results are shown in Figure 5. For all the approaches, the total time for the bulk insertion increased linearly with the number of records to be inserted. But the time for `Planned Bulk` increased much more slowly as compared to other approaches. In other words, the *insert throughput*, i.e., the number of inserts per unit time, obtained by `Planned Bulk` was much higher than other approaches.

From Figure 5, for 5 million new records, the throughput obtained by `Planned Bulk` was 4374 inserts per second, as compared to 1625 inserts per second for `OAT-Random`, and 694 inserts per second for `OAT-Sorted`. Note that the throughput obtained by `OAT-Random` is low in spite of the inserts being in random order, and hence the load being evenly distributed. This is because in `OAT-Random`, all the storage servers spend a fraction of their time moving partitions around, as directed by the load balancer. The same problem does not arise in `Planned Bulk` because in the planning phase, the partitions are pre-split and moved while they are still small. Finally, the throughput obtained by `OAT-Sorted` is only about 1/7th that of `Planned Bulk`. Keeping in mind that there are 10 storage servers, this result matches our intuition: `OAT-Sorted` hits only a small number of partitions at a time, and thus gets close only to single-storage-server throughput.

Since `OAT-Sorted` has obvious, severe bottlenecks, and its performance is always dominated by that of `OAT-Random`, in many of the rest of our experiments we leave out the `OAT-Sorted` numbers, and only compare our approach against `OAT-Random`.

6.2.2 Bulk Insertion into an Empty Table

In this experiment, we set $n_i = 0$, i.e., we started with an empty table, and again varied n_b from 1 million to 5 million. The results are shown in Figure 6. `Planned Bulk` remains significantly faster than `OAT-Random` at all values of n_b. Comparing with the previous experiment we can see that the performance of `Planned Bulk` is basically unaffected, thereby validating that our approach works equally well irrespective of the size of the original table. The performance of `OAT-Random`, on the other hand, is better in this experiment than in the previous one. This is because when we start with an empty table, the partitions that have to be moved for load balancing are of much smaller size, thereby reducing the move overhead incurred by `OAT-Random`. The same improvement does not apply to `Planned Bulk` because it is already optimized to pre-plan and move only small, often empty, partitions.

6.2.3 Varying the Distribution of Bulk Inserted Data

In this experiment, we varied the distribution of the data to be bulk inserted. We tried three different distributions—a uniform distribution, a Zipfian distribution with Z=1, and one in which the keys for the data to be bulk loaded were at the end of the existing key range. The last one is referred to as the *append* case, and can be quite common in practice as motivated in Section 1.

Figure 7 shows the performance of bulk inserts for these distributions for all three approaches. `Planned Bulk` again performs consistently better than the one at a time approaches, and its performance is essentially independent of data distribution. `OAT-Random` is also almost unaffected by the data distribution due to the automatic load balancing that is carried out[2]. However the performance of `OAT-Sorted` degrades heavily for the append case: insertion in sorted order for the append case leads to consistent load on the last partition, thereby obtaining only single-storage-server throughput.

6.2.4 Varying the sampling fraction

Recall that the planning phase of `Planned Bulk` uses sampling over both the new and the existing data to gather statistics. Figure 8 (the x-axis is on log scale) shows the effect on the total time for bulk insertion by `Planned Bulk` as the sampling fraction is varied. Parameter n_i was set to 0, while n_b was 5 million. Most of the benefit of our approach can be obtained by a sampling rate of as low as 1%. As the sampling rate is increased beyond 1%, there is very little improvement in performance. In fact at very high sampling rates such as 8%, performance begins to degrade as the time spent on sampling the data begins to become a noticeable overhead.

6.2.5 Effect on Normal Workload Processing

During our experiments, we also set up a client that carried out regular reads and writes on randomly chosen keys in the existing table. The client performed an equal number of reads and writes. Figure 9 shows the effect on average per-operation latency observed by the client during differ-

[2]The slightly better performance of `OAT-Random` for Zipfian data than for uniform data may be because more insertions go to the same partition in quick succession, thereby allowing the Berkeley DB at the storage server to get a higher benefit out of group commit.

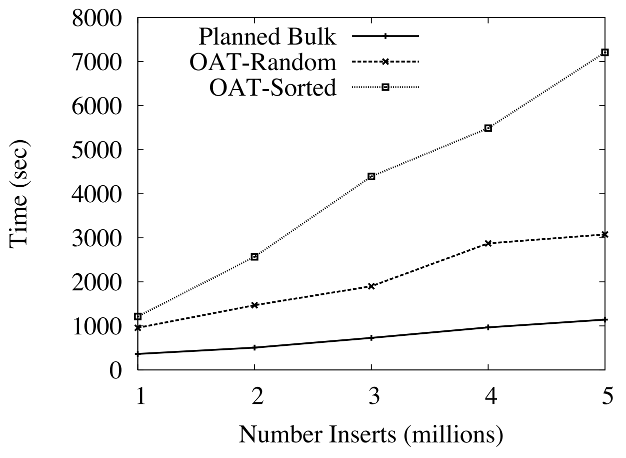

Figure 5: Effect of number of inserted records.

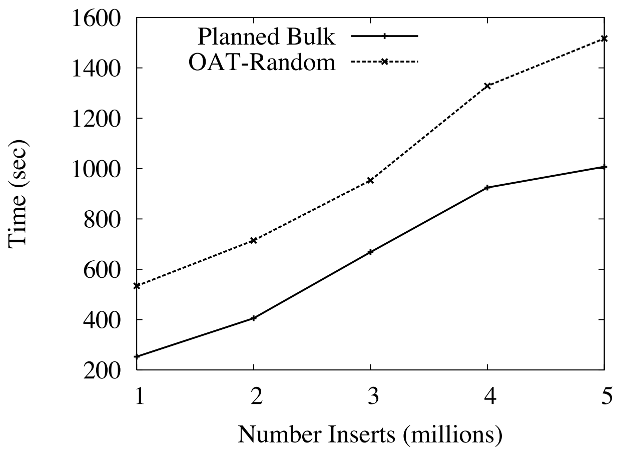

Figure 6: Inserting into empty table.

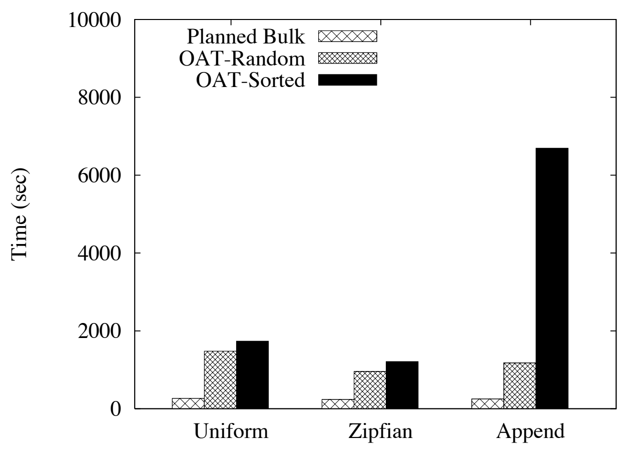

Figure 7: Varying the distribution of inserted data.

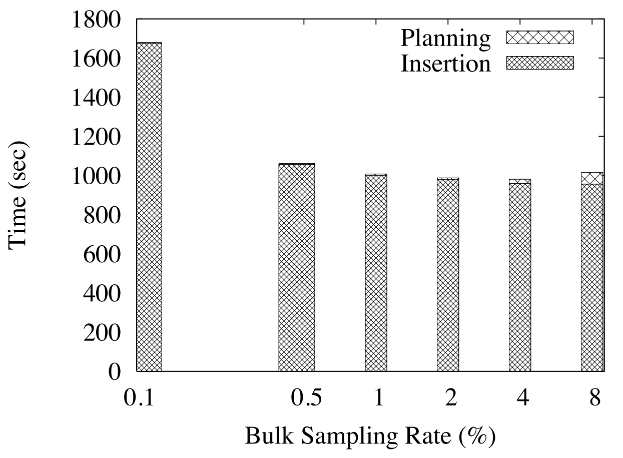

Figure 8: Effect of sampling rate.

ent phases of the bulk insertion (`PB` refers to `Planned Bulk`). The effect on client latency under our approach is no worse than inserting one-at-a-time: In all approaches, while bulk inserts are in progress, the client latency roughly triples as compared to the latency when the system is idle. It can be argued that since bulk inserts finish faster under our approach (Section 6.2), the "overall" impact on the client is less compared to other approaches that see the same degradation but for a longer time.

Figure 9 also shows that during the planning phase of `Planned Bulk` (when pre-planned splits and moves are in progress), the client latency is only about 1.5 times the idle latency, thereby validating that the splits and moves are chosen intelligently by our algorithm to minimize the drain on system resources.

6.3 Experiments on Real Data

We obtained a set of Yahoo! Autos car sale listings with about 500,000 records. We first removed all the `Porsche` records (roughly 0.3% of the records), and all the `Chevrolet` records (roughly 15.2% of the records) from the data set. We then loaded the remaining records into our system. We carried out two independent experiments: one in which the `Porsche` records were inserted, and another in which the `Chevrolet` records were inserted. The results using different approaches are shown in Figure 10.

In these experiments too, `Planned Bulk` was much more efficient than `OAT-Random` or `OAT-Sorted`. The performance advantage was especially high for the case when Chevrolets were inserted: the Chevrolet records actually happened to be concentrated on a narrow key range, thereby resulting in *dense inserts* (Section 2.1). Hence `OAT-Random` and `OAT-Sorted` only targeted a few partitions at a time, while `Planned Bulk` was able to obtain parallelism by doing the appropriate splits and moves.

6.4 Scaling up the System

In this experiment, we evaluated how well our algorithm scales as the number of storage servers, and the amount of data handled, increases. We scaled the number of storage servers from 10 up to 50. At the same time, other parameters were proportionally scaled up: the number of clients/stagers from 2 to 10, the number of initial records in the table from 5 million to 25 million, and the number of records to be bulk inserted from 1 million to 5 million. We again used synthetic data for this experiment. Figure 11 shows the bulk insert

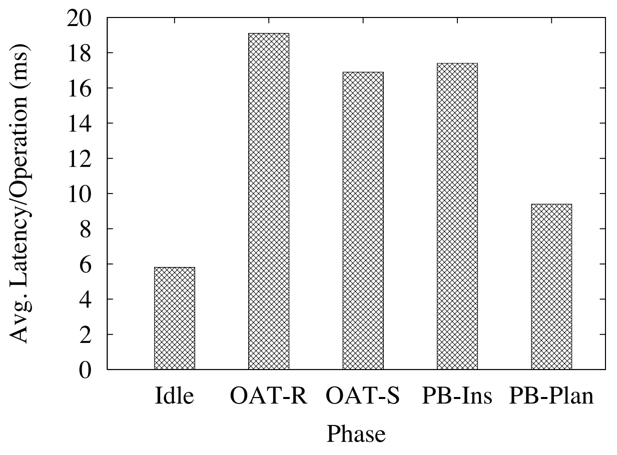

Figure 9: Normal operation latency during different phases.

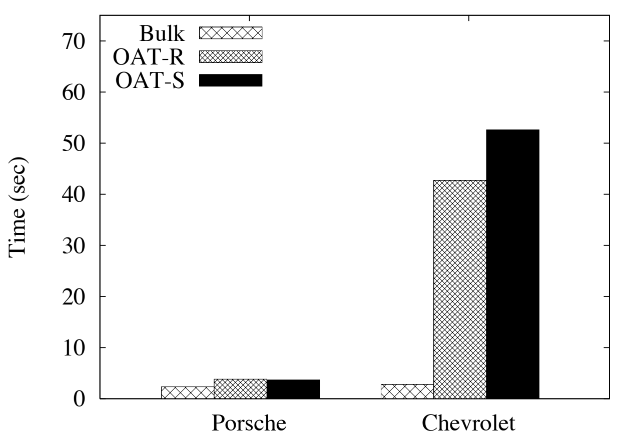

Figure 10: Insertion times for Internet Auto Sales Data.

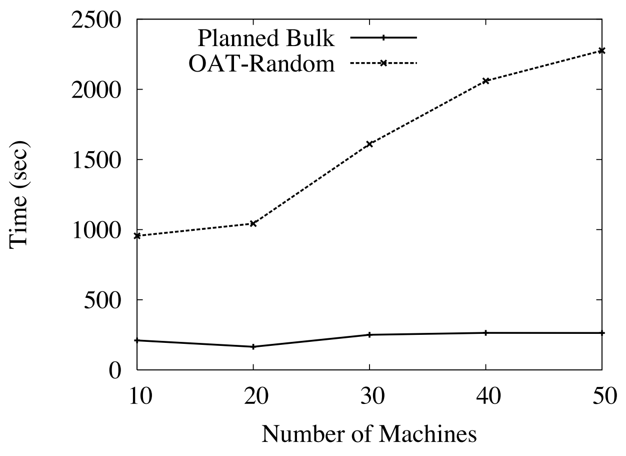

Figure 11: System Scale-up

performance as the system scales up. `Planned Bulk` scales almost perfectly as the system size is increased: the bulk load time remains roughly constant, thereby showing that our algorithm is able to extract maximum parallelism from the cluster. The performance of `OAT-Random`, however, degrades as the system size is increased. This is because the size of the data is simultaneously increased, and hence the move cost paid by `OAT-Random` increases. `OAT-Sorted` (not shown) did not scale well at all, and always got close to single-server throughput.

7. RELATED WORK

Recently, many groups have recognized the need for massively scalable distributed databases, especially for Internet applications: *BigTable* at Google [8], *Dynamo* at Amazon [12], and at Microsoft [6]. Our own project, PNUTS [15, 10], which motivates this work, falls in this category as well. Such systems are also useful in enterprise and scientific domains [27, 5]. Bulk insertion is a generally useful feature for each of these systems, and is non-trivial to implement, especially for ordered tables. To our knowledge, none of the above projects have previously published details of bulk-load techniques.

Single-Node Database Bulk Insertion: There is a large body of work on centralized settings where data is bulk inserted, e.g., data warehouses being updated by bulk inserts. As in our system, one-at-a-time insert is too slow in this scenario. The main technique used in such centralized settings is to carefully group records by key range and batch insert them into the database. The *log-structured merge-tree* [22] maintains an auxiliary index of inserts, and carries out the bulk insertion through a rolling merge: the leaves of the auxiliary and the primary indices are scanned sequentially and merged. Jagadish *et al.* [17] build multiple different-sized auxiliary indexes, with smaller indexes rolling into larger ones. Other approaches for B-tree bulk insertions that handle heavy insert rates are surveyed by Graefe [14].

All these techniques focus on a single-disk setting, and hinge on fast sequential scans of B-tree blocks. Hence they try to group together operations that are for the same block. In our setting, however, if we view each horizontal partition as a block, these blocks reside on different machines, and hence on different disks. Our goal is quite the *opposite*: we want to do operations to multiple blocks simultaneously to harness the parallelism of our cluster, thereby motivating an entirely new approach. The above techniques still remain useful to us since we can reuse them locally at each storage server as bulk inserts arrive from stagers.

Other work on bulk insertions only caters to specific types of data, such as object-oriented [28] or multidimensional databases [25]. The SQLServer documentation [20] suggests best practices for bulk loading time-ordered data that includes the approach of maintaining a portion of disk for each day or week, such that all inserts hit a single portion, again the exact opposite of what we want.

Building indexes: A classic, stylized bulk insertion problem is that of building a secondary index from an existing primary table. Mohan and Narang [21] provide techniques for ensuring that the index is consistent without having to suspend updates to the primary table. We discuss loading indexes in Section 5, but it is not the primary focus of this paper.

Shared-nothing cluster databases: Our system is an example of a shared-nothing cluster database, although its scale (thousands of nodes) is an order of magnitude or more larger than traditional cluster databases. Our bulk loading approach draws inspiration from the bulk sorting techniques of DeWitt *et al.* [13]. The challenge in parallel sorting is to evenly divide records among machines to ensure load balancing, much as we divide partitions among machines to balance insertions. As in our approach, parallel sorting uses sampling to determine a good distribution of key ranges to machines, and Seshadri and Naughton [24] provide analysis showing the number of samples needed to ensure roughly even partitions. While these techniques can be used for bulk insertions into empty tables, their applicability is limited when the table has pre-existing data.

A more recent well-known shared-nothing system is *MapReduce* [11]. The *map* stage can be viewed as a bulk insertion and partitioning of records. This is again analogous to bulk insertion into an empty table, and does not extend to non-empty tables. Moreover, in map-reduce, records are partitioned through hashing by default, or else by a user-provided partitioning function. Also, under map-reduce, bulk operations constitute the entire workload, unlike in our system, where normal queries are also being served. We are not aware of any work for shared-nothing systems that focuses on bulk insertion into an already populated database.

Shared-Disk Cluster Databases: Prominent examples of shared-disk distributed databases include Oracle RAC [3] and IBM's HACMP [2]. The shared "disk" in these systems is typically multiple disks, often in a storage area network. To utilize the bandwidth of multiple disks in parallel, one needs to ensure that the overlap between the set of data blocks accessed by different processing nodes is small to avoid expensive cache coherency protocols [4]. We are not aware of any published work on how bulk insert operations over ordered tables are handled in shared-disk architectures.

Vector-packing: To the best of our knowledge, we are the first to define the vector-packing problem. It is, however, related to several variants of the knapsack problem. The *multiple knapsack problem* [16, 19, 9], can be phrased as follows in our terminology: We are given a set of bins, each with an insert limit, and a set of partitions, each with an insert cost and a move cost. Our goal is to find a subset of partitions, assigned to bins so that the total insert cost does not exceed the insert limit, and so that the *total* move cost is as small as possible. Although this is similar to our vector-packing problem, it clearly does not solve it, since we are concerned with minimizing the maximum move cost over each bin, taken separately. In addition, we must assign every partition, rather than taking a subset.

Another related problem is the *multi-dimensional knapsack problem* [23, 18, 26]. Again using our terminology, this can be thought of as follows: We are given a single bin with an insert limit and a move limit. We are also given a set of items, each with an insert cost and a move cost. Our goal is to find the largest set of partitions possible assigned to the bin so that neither the total insert cost nor the total move cost exceeds their respective limits. This does not, however, solve the vector-packing problem, since we have multiple bins.

Finally, one fundamental difference in our approach is that we give an algorithm with a fixed, additive error. More traditional approaches either produce algorithms with relative ϵ error, generally taking time polynomial in ϵ^{-1}, or give heuristics to solve the problem. Our algorithm is both simple and has good guarantees. Due to the relative sizes of the bins' capacities, as compared to the partitions' insert and move costs, these guarantees are incredibly strong.

8. CONCLUSIONS

We have presented a technique for bulk insertion in large scale distributed databases, such as our own PNUTS. In particular, we have focused on distributed sorted tables. Bulk inserts are useful both to initially populate a table, and to keep the table populated with new data. A straightforward approach of inserting records one-at-a-time can overload individual storage servers, while failing to exploit the system's parallelism. This overloading not only increases the time for bulk insertion, but can negatively impact normal users of the system. Our approach is to stage the data if necessary, plan a set of partition splits and moves that allow us to maximize the parallelism of inserts, and then insert the data. The move planning process is NP-hard, and we have presented and formally characterized an approximation algorithm for the problem. Experimental results using both synthetic and real data demonstrate that our approach can provide significant performance improvements.

9. REFERENCES

[1] GridFTP. http://www.globus.org/grid_software/-data/gridftp.php.

[2] Hacmp for system p servers. http://www-03.ibm.com/systems/p/software/hacmp/index.html.

[3] Oracle real application clusters 11g. http://www.oracle.com/technology/products/-database/clustering/index.html.

[4] Scalability and performance with oracle 11g database. http://www.oracle.com/technology/deploy/-performance/pdf/db_perfscale_11gr1_twp.pdf, 2007.

[5] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proc. SOSP*, October 2007.

[6] P. Bernstein, N. Dani, B. Khessib, R. Manne, and D. Shutt. Data management issues in supporting large-scale web services. *IEEE Data Engineering Bulletin*, December 2006.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. INFOCOM*, 1999.

[8] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[9] C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2003.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. Technical report, Yahoo! Research, 2008.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, December 2004.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, October 2007.

[13] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. Conference on Parallel and Distributed Information Systems*, pages 280–291, 1991.

[14] G. Graefe. B-tree indexes for high update rates. *SIGMOD Record*, 35(1):39–44, March 2006.

[15] C. S. Group. Community systems research at yahoo! *SIGMOD Record*, 36(3):47–54, September 2007.

[16] S. Hung and J. Fisk. An algorithm for 0-1 multiple knapsack problems. In *Naval Res. Logist. Quart.*, pages 571–579, 1978.

[17] H. Jagadish, P. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *Proc. Very Large Databases*, August 1997.

[18] R. Loulou and E. Michaelides. New greedy-like heuristics for the multidimensional 0-1 knapsack problem. *Operations Research*, 27:1101–1114, 1979.

[19] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[20] S. Mishra. Loading bulk data into a partitioned table. http://www.microsoft.com/technet/prodtechnol/sql/-bestpractice/-loading_bulk_data_partitioned_table.mspx, September 2006.

[21] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proc. SIGMOD*, June 1992.

[22] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-sructured merge-tree. In *Acta Informatica*, volume 33, pages 351–385, 1996.

[23] S. Senju and Y. Toyoda. An approach to linear programming with 0-1 variables. *Management Science*, 15(4):B196–B207, 1968.

[24] S. Seshadri and J. Naughton. Sampling issues in parallel database systems. In *Proc. Extending Database Technology*, March 1992.

[25] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. Very Large Databases*, August 1997.

[26] M. Vasquez and J. Hao. A hybrid approach for the 01 multidimensional knapsack problem. In *IJCAI*, pages 328–333, 2001.

[27] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. OSDI*, 2006.

[28] J. Wiener and J. Naughton. Oodb bulk loading revisited: The partitioned-list approach. In *Proc. Very Large Databases*, September 1995.