Scalable Distributed Virtual Data Structures

Sushil Jajodia [1], Witold Litwin [2], Thomas Schwarz, SJ [3]

[1] George Mason University, Fairfax, Virginia, USA
[2] Université Paris Dauphine, Paris, France
[3] Universidad Católica del Uruguay, Montevideo, Rep. Oriental del Uruguay

ABSTRACT

Big data stored in scalable, distributed data structures is now popular. We extend the idea to *big, virtual data*. Big, virtual data is not stored, but materialized a record at a time in the nodes used by a scalable, distributed, *virtual* data structure spanning thousands of nodes. The necessary cloud infrastructure is now available for general use. The records are used by some big computation that scans every records and retains (or aggregates) only a few based on criteria provided by the client. The client sets a limit to the time the scan takes at each node, for example 10 minutes.

We define here two scalable distributed virtual data structures called VH* and VR*. They use, respectively, hash and range partitioning. While scan speed can differ between nodes, these select the smallest number of nodes necessary to perform the scan in the allotted time R. We show the usefulness of our structures by applying them to the problem of recovering an encryption key and to the classic knapsack problem.

I INTRODUCTION

The concept of *Big Data*, also called *Massive Data Sets* [12], commonly refers to very large collections of stored data. While this concept has become popular in recent years, it was already important in the eighties, although not under this monicker. In the early nineties, it was conjectured that the most promising infrastructure to store and process Big Data is a set of mass-produced computers connected by a fast, but standard network using Scalable Distributed Data Structures (SDDS). Such large-scaled infrastructures now are a reality as P2P systems, grids, or clouds. The first SDDS was LH* [1,7,9], which creates arbitrarily large, hash-partitioned files consisting of records that are key-value pairs. The next SDDS was RP* [8], based on range partitioning and capable of processing range queries efficiently. Many other SDDS have since appeared; some are in world-wide use. BigTable, used by Google for its search engine, is a range-partitioned SDDS, as are Azur Table provided by MS Azur and MongoDB. Amazon's EC2 uses a distributed hash table called Dynamo. VMWare's Gemfire provides its own hash scheme, etc. APIs for the popular MapReduce framework, in its Google or Hadoop version, have been created for some of these offerings [2,3].

As many data structures, an SDDS stores records in buckets. Records are key-value pairs and are assigned to buckets based on the key. Buckets split incrementally to accommodate the growth of an SDDS file whenever a bucket overflows its storage capacity. Every SDDS offers efficient access to records for key-based operations; these are the read, write, update, and delete operations. LH$*$ even offers constant access times regardless of the total number of records. SDDS also support efficient scans over the value field. A scan explores every record, usually in parallel among buckets. During the scan, each node selects (typically a few) records or produces an aggregate (such as a count or a sum) according to some client-defined criterion. The local results are then (usually) subjected to a (perhaps recursive) aggregation among the buckets.

Below, we use the SDDS design principles to define *Scalable Distributed Virtual (Data) Structure* (SDVS). Similar to an SDDS, a SDVS deals efficiently with a very large number of records, but these records are *virtual records*. As such, they are not stored, but materialized individually at the nodes. The scan operation in an SDVS visits every materialized record and either retains if for further processing or discards it before materializing the next record. Like an SDDS scan, the process typically retains very few records.

Virtual records have the same key-value structure as SDDS records. Similar to an SDDS with explicitly stored records, an SDVS deals efficiently with *Big*

Virtual Data that consists of a large number of *virtual* records. Virtual data however are not stored, but are materialized individually, one record at a time, according to a scheme provided by the client. As an example, we consider the problem of maximizing an unruly function over a large, finite set, subject to a number of constraints. We assume that we have an efficient enumeration of the set that we can therefore write as $S = \{s_i | i \in \{0, 1, \ldots, M-1\}\}$. A virtual record then has the form (i, s_i) and the record is materialized by calculating s_i from i. An SDVS does not support key-bases operations, but provides a scan procedure. The scan materializes every record and either retains and aggregates or discards the record. In our example, the scan evaluates whether the value field s_i fulfills the constraints, and if this is the case, evaluates the function on the field. It retains the record if it is the highest value seen at the bucket. Often, a calculation with SDVS will also need to aggregate the local results. In our example, the local results are aggregated by selecting the global maximum of the local maxima obtained at each bucket.

A client needs to provide code for the materialization of records, the scan procedure, and the aggregation procedure. These should typically be simple tasks. The SDVS provides the organization of the calculation.

An SDVS distributes the records over an arbitrarily large number N of cloud nodes. For big virtual data, N should be at least in the thousands. Like in an SDDS, the key determines the location of a record. While an SDDS limits the number of records stored at a bucket, an SVDS limits the time to scan all records at a bucket to a value R. A typical value for R would be 10 minutes.

The client of a computation using SDVS (the SDVS client) requests an initial number $N_c \geq 1$ of nodes. An SDVS determines whether the distribution of the virtual records over that many buckets would lead to scan times less than R. If not, then the data structure (not a central coordinator) would spread the records over a (possible many times) larger set N of buckets. The distribution of records tries to minimize N to save on the costs of renting nodes. It adjusts to differences in capacity among the nodes. That nodes have different scanning speed is a typical phenomenon. A node in the cloud is typically a Virtual Machine (VM). The scanning throughput depends on many factors. Besides differences in the hardware and software configuration, several VM are typically collocated at the same physical machine and compete for resources. Quite possibly, only the node itself can measure the actual throughput. This type of problem was already observed for MapReduce [14].

In what follows we illustrate the feasibility of our paradigm by defining two different SDVS. Taking a cue from the history of SDDS, we propose first VH$*$, generating hash partitioned, scalable, distributed, virtual files and secondly VR$*$, using scalable range partitioning. In both structures, the keys of the virtual records are integers within some large range $[0, M]$, where for example M might be equal to 2^{40}. We discuss each scheme and analyze its performance.

We show in particular that both schemes use on the average about 75% of its capacity for the scanning process. The time needed to request and initialize all N nodes is $O(\log_2(N))$. If N only grows by a small factor over N_c, say less than a dozen, then the *compact* VR$*$ can use almost 100% of the scan capacity at each nodes at the expense however of $O(N)$ time for the growth to N nodes. The time to set up a VM appears to be 2-3 seconds based on experiments. This was observed experimentally for the popular Vagrant VM [4, 13].

To show their usefulness, we apply our cloud-based solutions to two classical problems. First, we show how to invert a hash, which is used to implement recovery of an encryption key [5, 6]. The second is the generic knapsack problem, a famously hard problem in Operations Research [11]. Finally, we argue that SDVS are a promising cloud-based solution for other well-known hard problems. Amongst these we mention those that rely on exploring all permutations or combinations of a finite set such as the famous traveling salesman problem.

II SCALABLE DISTRIBUTED VIRTUAL HASHING

Scalable Distributed Virtual Hash (VH*) is a hash-partitioned SDVS. VH* nodes are numbered consecutively, starting with 0. This unique number assigned to a node is its *logical address*. In addition, each node has a physical address such as its TCP/IP address. The scheme works on the basis of the following parameters which are common, in fact, to any SDVS scheme we describe below.
(a) Node capacity B. This is the number of virtual records that a node can scan within time R. Obviously, B depends on the file, the processing speed of the node, on the actual load, e.g. on how many virtual machines actually share a physical machine, *etc.*

There are also various ways to estimate B. The simplest one is to execute the scan for a sample of the records located at a node. This defines the node's throughput T [6]. Obviously we have $B = RT$.
(b) Node load L This is the total number of virtual records mapped to a node.
(c) Node load factor α, defined by $\alpha = L/B$.

This terminology also applies to schemes with stored records. In what follows, we assume that B is constant for the scan time at the node. The above parameters depend on the node. We use subscripts to indicate this dependencies. A node n terminates its scan in time only if $\alpha_n \leq 1$. Otherwise, the node is *overloaded*. For smaller values of α_n the node is *reasonably loaded* and becomes underloaded for $\alpha_n < 1/2$. An overloaded node *splits* dividing its load between itself and a new node as we will explain below. The load at the splitting node is then halved. Each split operation appends a new node to the file. New nodes also follow the splitting rule. The resulting *creation* phase stops when $\alpha_n \leq 1$ for every node n in the file.

In more detail, VH* grows as follows:
1. The client sends a *virtual file scheme* S_F where F is a file, to a dedicated cloud node, called the *coordinator* (or *master*) and denoted by C in what follows. By default, C is Node 0. Every file scheme S_F contains the maximally allowed time R and the key range M. These parameters determine the *map* phase as we have just seen. We recall that the file F consists of M virtual records with keys numbered from 0 to $M - 1$. There are no duplicate keys. The rules for the materialization of value fields and the subsequent selection of relevant records are also specified in S_F. These determine the *scan* phase of the operation. Finally, the *termination* phase is also specified, determining the return of the result to the client. This includes perhaps the final inter-node aggregation (reduction). Details depend on the application.

2. Node 0 determines B based on S_F. It calculates $\alpha_0 = M/B$. If $\alpha_0 \leq 1$ then node 0 scans all M records. This is unlikely to happen, and normally, $\alpha_0 >> 1$. If $\alpha_0 > 1$, then Node 0 chooses additional $N_0 - 1$ nodes, sends S_F to all nodes in $\mathcal{N}$, sends a level $j = 0$ to all nodes in $\mathcal{N}$ and labels the nodes contiguously as 1, …, $N_0 - 1$.

The intention of this procedure is to evenly distribute the scan over $N = N_0$ nodes (including the initial node) in a single round of messages. By setting $N_0 = \alpha_0$, the coordinator (Node 0) guesses that all N_0 nodes have the same capacity as Node 0 and will now have load factor $\alpha = 1$. This is only a safe bet if the nodes are homogeneous and have the same load. This is the case when we are working with a homogeneous cloud with private virtual machines. By setting $N_0 = 2$, Node 0 chooses to not assume anything. In this strategy, the number of rounds could be markedly increased, but we avoid a low guess that will underutilize many of the allocated and paid for nodes.

3. Each node i in $\{0, \ldots, N_0\}$ determines its load factor α_i. If $\alpha_i \leq 1$, Node i starts scanning. Node i generates the records assigned to it by the LH addressing principles [9]. In general, this means that the records have keys K with $i = K \bmod 2^j N_0$. We will see nodes with $j \geq 1$ shortly and recall that the nodes in the set $\mathcal{N}$ all have $j = 0$.

4. If Node i has a load factor $\alpha_i > 1$, then this node splits. It requests from the coordinator (or from some virtual agent serving as cloud administrator) the allocation of a new node. It gives this node number $i + 2^j$, assigns itself and the new node level $j = j + 1$. If finally sends this information and S_F to the new node.

5. All nodes, whether the original Node 0, the nodes in $\mathcal{N}$ or subsequentially generated nodes loop over steps 3-4.

At the end of this possibly recursive loop, every node enters the *scan* phase. For every node, this phase usually requires the scan of every record mapped to the node. The scan therefore generates successively all keys in its assigned key space, materializes the record according to the specifications in S_F and calculates the resulting key-value pair to decide whether to select the record for the final result. If a record is not needed, it is immediately discarded.

VH* behaves in some aspects different than LH*, from which it inherits its addressing scheme. Whereas LH* creates buckets in a certain order and the set of bucket addresses is always contiguous, this is not the case for the node address space in VH*. For example, if the initial step creates $N_0 = 1000$ nodes, it is possible that only Node 1 splits, leaving us with an address space of $\{0, 1, \ldots, 999, 1001\}$. If the same Node 1 splits again, we would add a node 2001. In LH*, buckets are loaded at about 70% and the same number should apply to VH*. We call the splitting node the *parent* and the nodes it creates its *children*.

6. Once a node has finished scanning records, it enters the *termination* phase with its proper protocol.

After the termination phase, the node always deallocates itself.

The simplest protocol for the termination phase is a *direct* protocol that requires every node to deliver its results, basically the selected records, to the client or to a coordinator. This protocol performs well if one expects only a small number of results. Otherwise, it can create a message storm to the client or coordinator. As an alternative, we offer a protocol with inter-node aggregation. Because the node address space is not contiguous, our best option is for children to report to their parents. The total number of children is the difference between the initial level at node creation and the current level. This allows a node to determine whether it has heart from all of its children. Therefore, when a node finishes scanning its records and has no children, it sends the result to its parent. Otherwise, it waits for all of its children to send it their results. If it has heart from all its children, it aggregates the results. Finally, all nodes in the original set $\mathcal{N}$ aggregate by sending their results to the client.

7. It may happen that the scan searches for exactly k, $k \geq 1$ records in the record space. In this case, we can change the termination protocol by requiring that each node that has found a record reports this directly to the coordinator. We then want the coordinator be able to terminate the calculation at all nodes. Here, information flows contrary to the information flow in the *termination* phase from coordinator to nodes. Doing so is simple. If the coordinator does not have all node addresses – because the coordinator was not responsible for requesting the generation of all nodes – then we can send from parent to children.

8. It is possible that nodes fail or become otherwise unavailable. For the direct termination protocol, the coordinator might not know the addresses of unavailable nodes. In this case, we need to expand the protocol to detect unavailable nodes. In the alternative termination protocol, the parents know their children and can ascertain whether they are unavailable if they are overdue in delivering their results. Once the coordinator knows the fact that nodes are unavailable and their addresses and levels, it can recreate them and bring the calculation to a successfull end. We leave the discussion and evaluation of recovery schemes to future work

Using a simple invariant argument, we can show that the VH* splitting rules define an exact partitioning of the key space. In other words, every virtual key hashes to exactly one node and is scanned therefore only once. This is certainly true when the client has started the scheme and the coordinator is responsible for the complete key space. It remains true after the coordinator has created N_0 nodes and partitioned the key space among them. Finally, each further split partitions the assigned key space among the splitting and the new node.

EXAMPLE 1: NOISED SHARED SECRET FOR ENCRYPTION KEY RECOVERY

This procedure was designed to always recover a (secret-sharing) share of a client's encryption key from a remote backup in a given time limit [5,6]. The recovery must use a large cloud to perform this task in the allotted time. We show how VH$*$ organizes the work in a large cloud.

More in depth, the backup scheme uses a key escrow agency. The client uses secret splitting to generate at least two shares. One of the shares is given to an entity as is, whereas the other is *noised* at a different escrow agency. Noising first calculates the hash of the key using a cryptographically secure hash. The noised share is stored as the hash together with a search space. The escrow agency only knows that the noised share is in the search space and that it can verify whether an element of the search space is the key share by calculating its hash and comparing it with the stored hash. Even if it were to use the information it has available, it would only recover one share and not the key. Given the costs of finding the hash, there is no incentive to do so unless the client requests and reimburses it. The cloud resources needed to find the key in the search are made too large to deter illegal attempts but small enough to still allow finding the noised key share and thereby allow key recovery.

We can formulate the recovery problem abstractly. The client defines an integer p, the *prefix*, of length, say, 216 bits. It also defines a large number M, say $M = 2^{40}$. The client then defines an integer s such that s is the concatenation of p and a secret suffix c, $c \in \{0, 1, \ldots, M-1\}$ i.e. $s = p.c$. In our example, the length of s is 256 bits and c is expressed as a bit string with 40 bits. Using some good one-way hash function h such as as a member of the SHA family, the client defines a *hint* $H = h(s)$. Given the size of M, it is very unlikely to have two different values $c_1, c_2 \in \{0, 1, \ldots, M-1\}$ with the same hint $h(p.c_1) = h(p.c_2)$.

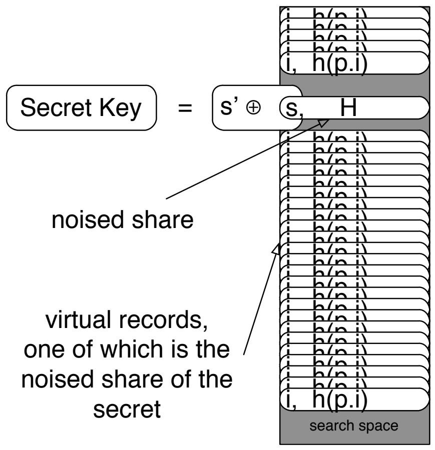
Secret Key
=
s' ⊕
s, H
i, h(p.i)
noised share
virtual records, one of which is the noised share of the secret
search space

Figure 1: The cryptographic key is secret shared and one share is noised. The noised share is in a large collection of records only one of which will result in a hash value equal to the hint H.

It also sets a reasonable time limit R, say $R = 60$ min. The problem is to retrieve s given H, M, and R. Here we assume that given the size of M, it is very unlikely to have two values for $s = p.c$ with the same hash value H.

The VH* solution is a virtual file mapped over a large number of nodes. The virtual records have keys i in $\{0, 1, \ldots, M-1\}$ and values $h(p.i)$, the hash value of the concatenation of the prefix and the key i. A result is a record with value H.

The *scanning* phase of VH* looks for every possible value of i. The records created are ephemeral and exists only for the purpose of evaluating the hash. As each key i is equally likely to be the solution, it is possible that all records have to be created. There should be only one result, so the coordinator attempts to stop all further calculation as outlined in Step 7 above. On average, the solution is found after half the maximum time, so that the costs are on average only half the maximum costs.

If the coordinator knows that the cloud is homogeneous and that every node will be a private VM (not sharing with other VMs), then the coordinator can determine exactly how many nodes are necessary in order to terminate the scan in time R. For example, if the load factor at the coordinator, Node 0 is $\alpha_0 = 1000$, the coordinator might simply choose to spread the virtual file over 1000 nodes 0 (itself), 1, …, 999. This is the smallest possible set of nodes. Node 0 will then scan (i.e. attempt to match records with key 0, 1000, 2000, … and Node 1 will scan the records with key 1, 1001, 2001, ….

If on the other hand the cloud is heterogeneous or there is interference with other virtual machines collocated at nodes, then the coordinator does best in limiting guessing. For instance, with the same initial load factor of $\alpha_0 = 1000$, the coordinator chooses an initial set of size 2. In the following, the coordinator will split the bucket at Node 0 ten times, generating the nodes 1, 2, 4, … 1024. Assume that Node 1 has double the capacity of Node 0 or in our notation $B_1 = 2B_0$. Since it is tasked with working the same number of records as Node 0 after the first split, its load is $\alpha_1 = 8$. Correspondingly, when Node 1 is created, it splits eight times, creating Nodes 3, 7, … 259, 513. Every other node will split according to its initial load factor, depending on its respective capacity B and the initial load it receives upon creation. The coordinator will not know neither the existence nor the address of all created nodes.

Suppose now that the search for the noised share succeeds at a node. The coordinator enters the *termination* phase upon notification. In the case of the homogeneous cloud, the coordinator simply sends the message to every node. Otherwise, it uses the parent-child relations to terminate the calculation at the other nodes by sending first to its children which then recursively send to their children. On average, the node with the searched-for record will find the record in half the time, so that this procedure usually saves almost half the maximum rental costs for the nodes. A preliminary performance analysis shows that these numerical values are practical in a cloud with thousands or tens of thousands of nodes [6]. Several cloud providers offer homogeneous clouds with private virtual machines.

Let us finally mention some extensions to key backup with noised share. First, as in classical secret splitting schemes, we might have more than two secret shares, of which we can noise all but one. In this case, we have that many hints and the value field in the record will be compared against all of them. If one record has one match, it becomes a result and is sent to the coordinator. Finally, the owner of the original, cryptographic key can distribute additional secrets, called *discounts* that limits the size of the search space if they are made known to the search [5]. This naturally means that the lesser work is distributed over fewer nodes.

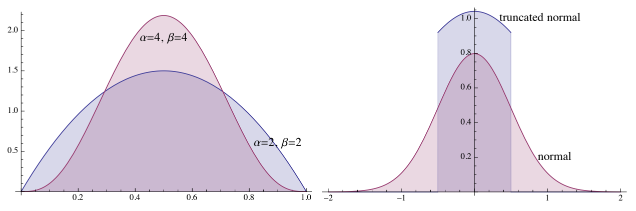

Figure 2: PDF for the two beta distributions (top) and the truncated normal distribution and normal distribution (bottom).

EXAMPLE TWO: KNAPSACK PROBLEM

The client application considers a number Q of objects, up to about 40 or 50. Each object has a *cost* c and a utility u. The client looks for the top-k sets (k in the few dozens at most) of objects that maximize the aggregated utilities of the objects in the set while keeping the total costs below a cost limit C. Additionally, the client wants to obtain the limit in time less than R. Without this time limit, the knapsack problem is a famously complex problem that arises in this or a more general form in many different fields. It is NP-complete [11] and also attractive to database designers [10].

To apply VH* to solve a knapsack problem of medium size with time limit, the client first numbers the objects 0, 1, ..., $Q-1$. Then the client creates a virtual key space of $\{0, 1, \ldots M = 2^Q\}$. Each element of the key space encodes a subset of $\{0, 1, \ldots Q-1\}$ in the usual way. Bit ν of the key is set if and only if ν is an element of the subset. If we denote by i_ν the value of bit ν in the binary representation of i, then the value field for the virtual record with key i is

$$\sum_{\nu=0}^{Q-1} i_\nu u_\nu, \sum_{\nu=0}^{Q-1} i_\nu c_\nu$$

In other words, the value field consists of the utility and the costs of the set encoded by the key. A match attempt involves the calculation of the field and the decision whether it is a feasable solution and is in the top K of the records currently seen by the node. At the end of the scanning phase, each node has generated a local top-k list.

In the termination phase, the local top k-lists are aggregated into a single top-k list. Following the termination protocol is S_F, this is done by the client or by the cloud, with the advantages and disadvantages already outlined. In the cloud, we can aggregate by letting each child send its top-k list to the parent. When the parent has received all these lists,

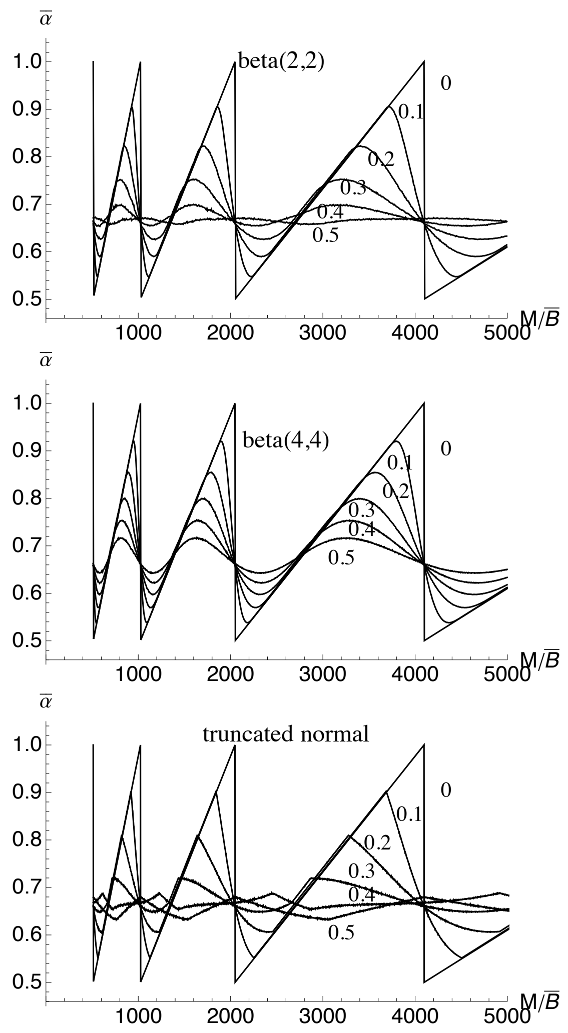

Figure 3: Values for average load factor $\bar{\alpha}$ dependent on normalized total load $M/\bar{B}$ for three different distributions. The labels give a measure of heterogeneity as explained in the text.

it creates an aggregated top-k list and sends it to its parent. When Node 0 has received its lists and aggregates them, it sends the result as the final result to the client.

Our procedure gives the *exact* solution to the knapsack problem with a guaranteed termination time using a cloud, which is a first to the best of our knowledge. Preliminary performance analysis shows that values of $Q \approx 50$ and R in the minutes are practical if we can use a cloud with thousands of nodes. Such clouds are offered by several providers.

An heuristic strategy for solving a knapsack problem that is just too large for the available or affordable resources is to restrict ourselves to a random sampling of records. In the VH* solution, we determine the total feasible number of nodes and use the maximum load for any node. Each node uses a random number generator to materialize only as many virtual records as it has capacity, that is, exactly B records, each with a key $K = i + \text{random}(0, M-1)2^j N_0$. The resulting *random scan* yields a heuristic, sub-optimal solution that remains to be analyzed.

Our algorithm is a straight-forward brute force method. It offers a feasible solution where known single machine OR algorithms fail. The method transfers immediately to problems of integer linear programming. There is much room for further investigation here.

VH* PERFORMANCE

The primary criteria for evaluation are (1) the response time and (2) the costs of renting nodes in the cloud. We can give average and worst case numbers for both. The worst response time for both of our examples is R (with an adjustment for the times to initialize and shut down the system), but with the key recovery problem, we will obtain the key on average in half the time, whereas for the knapsack problem the average value is very close to R. These values depend on the homogeneity of the cloud and whether the coordinator succeeds into taking heterogeneity into account.

At a new node, the total time consists of the allocation time, the start time to begin the scan, the time for the scan, and the termination time. Scan time is clearly bound by R. Allocation involves communication with the data center administrator and organizational overhead such as the administrator's updating allocation tables. This time usually requires less than a second [4] and is therefore negligible even when there are many round of allocation. The start time involves sending a message to the node and the node starting the scan. Since no data is shipped (other than the file scheme S_F) this should take at worst milli-seconds. The time to start up a node is that of waking up a virtual machine. Experiments with Vagrant VM have shown this time to be on average 2.7 seconds [4].

While individual nodes might surpass R only negligibly, this is not true if we take the cumulative effects of allocations by rounds into account. Fortunately, for VH$*$, there will be about $\log N$ rounds. The largest clouds for rent may have 10^7 nodes, so that the number of round is currently limited to 17 and might be as small as 1.

The provider determines the cost structure of using the cloud. There is usually some fee for node setup and possibly costs for storage and network use. Our applications have more than enough with the minimal options offered. There is also a charge per time unit. In the case of VH$*$, the use time is basically the scan time. The worst case costs are given by $c_1 N + c_2 RN$ where c_1 are the fixed costs per node and c_2 the costs per time unit per node. The value of N depends on the exact strategy employed by the coordinator.

A cloud can be homogeneous (all nodes have the same capacity B) in which case every node has the same capacity B. The coordinator then chooses a static strategy by choosing N_c to be the minimum number of nodes without overloading. In this case, all nodes are set up in parallel in a single round and the maximum response time is essentially equal to R. If α_0 is the load factor when the coordinator tests its capacity, then it will allocate $N = N_c = \alpha_0 = \lceil M/B_0 \rceil$ nodes and the maximum cloud cost is $c_1\alpha_0 + c_2\alpha_0 R$. The average cost in this case depends on the application. While the knapsack problem causes them to be equal to the maximum cost, the key recovery problem uses on average only half of R before it finds the sought value. The average cost is then $c_1\alpha_0 + (1/2)c_2\alpha_0 R$.

The general situation is however clearly that of a heterogeneous cloud. The general choice of the coordinator is $N_c = 1$. The VH$*$ structure then only grows by splits. The split behavior will depend on the load factor $M/\bar{B}$ and on the random node capacities in the heterogeneous cloud. If the cloud is in fact homogeneous unbeknown to the coordinator, then the node capacity B_n at each node n is identical to the average node capacity $\bar{B}$. In each round, each node n makes then identical split decisions and splits will continue until $\alpha_n \leq 1$ at every node n. Therefore, VH$*$ faced with a total load of M and an initial load factor $\alpha_0 = M/\bar{B}$ will allocate $N = 2^{\lceil \log_2(\alpha_0) \rceil}$ nodes, which is the initial load rounded up to the next integer power of two. Each node will have a load factor $\bar{\alpha} = \alpha_0 2^{-\lceil \log_2(\alpha_0) \rceil}$ which varies between 0.5 and 1.0. The maximum scan time is equal to $\bar{\alpha}R$. The cloud costs are $c_1 N + c_2\bar{\alpha}NR$ if we have to scan all records (the knapsack problem) and $c_1 N + 0.5 c_2 \bar{\alpha} NR$ for the key-recovery problem.

We also simulated the behavior of VH$*$ for various degrees of heterogeneity. We assumed that the capacity of the node follows a certain distribution. In the absence of statistical measurements of cloud node capacities, we assumed that very large and very small capacities are impossible. Observing very small node capacities is a violation of quality of service guarantees, and the virtual machine would be migrated to another setting. Very large capacities are impossible since the performance of a virtual machine cannot be better than that of the host machine. Therefore, we can only consider distributions that yield capacity values in an interval. Since we have no argument for or against positive or negative skewness, we picked symmetric distributions in an interval $(1-\delta)\bar{B}$ and $(1+\delta)\bar{B}$ and chose $\delta \in \{0, 1/10, 2/10, 3/10, 4/10, 5/10\}$. For $\delta = 0$, this models a homogeneous cloud. If $\delta = 5/10$, then load capacities lie between 50% and 150% of the average node capacity.

For the distributions, we picked a beta distribution with shape parameters $\alpha = 2$, $\beta = 2$, a beta distribution with shape parameters $\alpha = 4$, $\beta = 4$, and a truncated normal distribution in the interval $(1-\delta)\bar{B}$ and $(1+\delta)\bar{B}$ where the standard deviation of the normal distribution was $\delta\bar{B}$. The truncated normal distribution is generated by sampling from the normal distribution, but rejecting any value outside of the interval. We depict the probability density functions of our distributions in Figure 2.

Figure 3 shows the average load factor $\bar{\alpha}$ in dependence on the *normalized* total load $M/\bar{B}$. This value is the initial load factor α_0 in the homogeneous case. In the homogeneous case, the average load factor shows a sawtooth behavior with maxima of 1 when $M/\bar{B}$ is an integer power of two. For low heterogeneity, the oscillation is less pronounced around the powers of two. With increasing heterogeneity the dependence on the normalized total load decreases. We also see that $\bar{\alpha}$ averaged over the normalized load between 512 and 4096 decreases slightly with increasing heterogeneity. As we saw, it is exactly 0.75 for the homogeneous case, and we measured 0.728 for $\delta = 0.1$, 0.709 for $\delta = 0.2$, 0.692 for $\delta = 0.3$, 0.678 for $\delta = 0.4$, and 0.666 for $\delta = 0.5$ in the case of the beta distribution with parameters $\alpha = 2, \beta = 2$. We attribute this behavior to two effects. If a node has a lower than average capacity it is more likely to split in the last round increasing the number of nodes. If the new node has larger than usual capacity, then the additional capacity is likely to remain unused. Increased variation in the node capacities then has only negative consequences and the average load factor is likely to decrease with variation.

In the heterogeneous case, the total number of nodes is $N = M/(\bar{\alpha}\bar{B})$ and the average scan time is $\bar{\alpha}R$. The cloud costs are $c_1N + c_2\bar{\alpha}NR$ for the knapsack problem and $c_1N + 0.5c_2\bar{\alpha}RN$ in the key-recovery case.

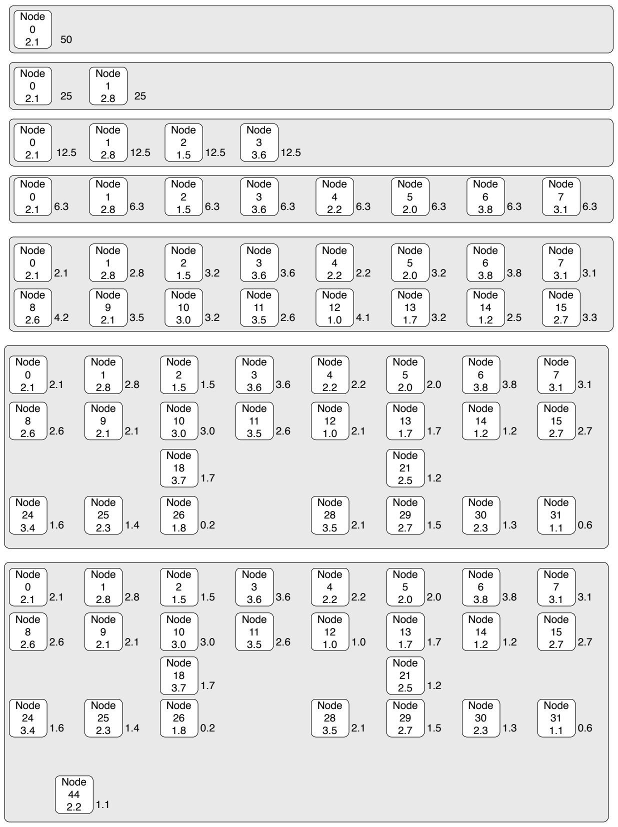
Figure 4: Example of the creation of a file using scalable partitioning with limited load balancing.

III SCALABLE DISTRIBUTED VIRTUAL RANGE PARTITIONING

Scalable distributed virtual range partitioning or VR$*$ SDVS partitions the virtual key space $[0, M)$ into N successive *ranges*

$$[0, M_1) \cup [M_1, M_2) \cup \ldots \cup [M_{N-1}, M)$$

Each range is mapped to a single and different node such that $[M_l, M_{l+1})$ is mapped to Node l. The VR* mapping of keys by ranges allows to easily tune the load distribution over a heterogeneous cloud with respect to the hahs-based VH* mapping. This potentially reduces the extent of the file and hence the costs of the cloud. The extent may be the minimal extent possible, i.e., the file is a *compact* file by analogy with a well-known B-tree terminology. In this case, the load factor is $\alpha = 1$ at each node. However, it can penalize the start time, and hence the response time.

For the range j located on Node j, M_{j-1} is the *minimal* key while $M_j - 1$ is the *maximum* key. The coordinator starts by creating the file at one node, namely itself, with a range of $[0, M)$. Next, as for VH*, the coordinator generates an initial mapping using N_0 nodes, where each node j carries the range of keys $[M_{j-1}, M_j)$ each of the same length of M/N_0. The coordinator chooses $N_0 = \lceil \alpha_0 \rceil$ based on its initial load factor. In an homogeneous cloud, the load factor at each node is therefore ≤ 1 and optimal. This yields a compact file.

In a heterogeneous cloud, we split every node with load factor $\alpha > 1$. The basic scheme divides the current range into two equal halves. However, we can also base the decision on the relative capacities of the nodes [6]. In previous work on recoverable encryption, we have the splitting node obtain the capacity of itself and of the new node. This implies that the new node has to determine its capacity first, which implies scanning a certain number of records. Only then do we decide on how to split. This appears to be a waste of time, but we notice that the records used to assess the capacity are records that have to be scanned anyway. Thus, postponing the decision how to split does not mean that we loose time processing records. However, if the second node also needs to split, then we have waited for bringing in the latest node. This means that these schemes do not immediately use all necessary nodes. If there are many rounds of splitting, then the nodes that enter the scheme latest either have to do less work or might not finish their work by the deadline. All in all, an optimized protocol needs to divide the load among the necessary number of nodes in a single step as much as possible.

The scheme proposed in our previous work (scalable partitioning with limited load balancing) [6] splits evenly until the current load factor is ≤ 3. If a node has reached this bound through splits, we switch to a different strategy. The splitting node assigns to itself all the load that it can handle and gives to the new node the remaining load. There are two reasons for this strategy: First, there is the mathematical version (the 0-1 law) of Murphy's law: If there are many instances – in our case thousands of nodes – then bad things will happen with very high probability if they can happen at all – in our case, one node will exhaust almost all its allotted time to finish. Therefore, there is no harm done by having a node use the maximum allotted time. Second, by assuming the maximum load, we minimize the load at the new node. Therefore, the total number of splits starting with the new node is less likely. We also note that the new node does not have to communicate its capacity with the splitting node. The reason for not using this strategy from the beginning is that we want to reach close to the necessary number of nodes as soon as possible. Thus, if the node load is very high, the current load needs to be distributed among many nodes and we do so more efficiently by splitting evenly. The number of descendants will then be about equal for both nodes.

We give a detailed example in Figure 4. The boxes contain the name of the node, which is its number according to LH* addressing and encodes the parent-child relationship. With each node, we have the capacity and outside the box, the current load. We generated the capacities using a beta-distribution with parameters $\alpha = 4$ and $\beta = 2$ and mean 8/3, but then rounded to one fractional digit. In the first panel, the initial load of 50 is assigned to Node 0 with a capacity of 2.1. This node makes a balanced split, requesting Node 1 and giving it half of its load. Both nodes split again in a balanced way, Node 0 gives half of its load to Node 2 and Node 1 to Node 3. All four nodes are still overloaded and make balanced splits (fourth panel). The load at all nodes is (rounded up) 6.3. All nodes but Node 2 and Node 5 have capacity at least one third of the load and split in an unbalanced way. For instance, Node 0 requests Node 8 to take care of a load of 4.2 while it only assigns itself a load equal to its capacity, namely 2.1. Nodes 2 and 5 however make a balanced split. As a result, in the fifth panel, various nodes are already capable of dealing with their load. Also, all nodes that need to split have now at least capacity one third of their assigned load. After the splits, we are almost done. We can see in the sixth panel that there is only one node, namely Node 12, that still needs to split, giving rise to Node 44 $(= 12+32)$. We have thus distributed the load over a total of 26 nodes. Slightly more than half, namely 15 nodes are working at full capacity. We notice that our procedure made all decisions with local information, that of the capacity and that of the assigned load. Of this, the assigned load is given and the capacity needs to be calculated only once. A drawback to splitting is that the final node to be generated, Node 44 in the example, was put into service in the sixth generation, meaning that excluding its own capacity calculation, there were five times that a previous node generated the capacity. If we can estimate the average capacity of nodes reasonably well, we can avoid this by having Node 0 use a conservative estimate to assign loads to a larger first generation of nodes.

A final enhancement uses the idea of balancing from

B-trees. If a node has a load factor just a bit over 1, it contacts its two neighbors, namely the nodes that have the range just below and just above its own range to see whether these neigbors have now a load factor below 1. If this is the case, then we can hand over part of the current node's range to the neighbor.

We have shown by simulation that these enhancements can be effective, but that their effects depend heavily on assumptions about the heterogeneity of the nodes [6]. For an exact assessment, we need to experiment with actual cloud systems. As has become clear from our discussion, the final scheme should guess the number of nodes necessary, allocate them, verify that the capacity is sufficient and accept that sometimes local nodes need to redistribute the work. We leave the exact design of such a protocol and its evaluation through experiments and simulations to future work.

IV CONCLUSIONS

Scalable distributed hash and range partitioning are currently the basic structures to store very large data sets (big data) in the cloud. VH$*$ and VR$*$ apply their design principles to very large *virtual* data sets in the cloud. They form possibly minimal clouds for a given time limit for a complete scan. They work with homogeneous and, more importantly because more frequent, heterogeneous clouds. These schemes constitute to the best of our knowledge the first cloud-based and - more importantly - fixed execution time solutions to two important applications, namely key recovery and knapsack problems.

Further work should implement these schemes at a large scale and add details to the performance analysis. For instance, the behavior of actual node capacity in commercial cloud solutions needs to be measured so that performance modeling can be placed on a firmer foundation.

The two schemes can be used for other difficult problems in Operations Research such as 0-1 integer linear programming or the traveling salesman problem. The problems amenable to these solutions are search problems where a large, search space $S = \{s_i | i = 1, \dots M\}$ has to be scanned in fixed time.

References

[1] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, P. Senellart et al.: Web data management, Cambridge University Press, 2012.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. "Bigtable: A distributed storage system for structured data." ACM Transactions on Computer Systems (TOCS) 26, no. 2 (2008):4.

[3] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51, no. 1 (2008): 107-113.

[4] W. Hajaji. "Scalable partitioning of a distributed calculation over a cloud". Master's Thesis, Université Paris, Dauphine, 2013 (in French).

[5] S. Jajodia, W. Litwin, and T. Schwarz. "Key Recovery using Noised Secret Sharing with Discounts over Large Clouds". Proceedings ASE / IEEE Intl. Conf. on Big Data, Washington D.C., 2013.

[6] S. Jajodia, W. Litwin, T. Schwarz. "Recoverable Encryption through a Noised Secret over a Large Cloud". Transactions on Large-Scale Data and Knowledge-Centered Systems 9, Springer, LNCS 7980, pages 42-64, 2013

[7] W. Litwin, M.-A. Neimat, and D. A. Schneider. "LH$*$: Linear Hashing for distributed files". Proceedings of ACM SIGMOD international conference on management of data, 1993.

[8] W. Litwin, M.-A. Neimat, and D. Schneider. "RP*: A family of order preserving scalable distributed data structures." In VLDB, vol. 94, pp. 12-15. 1994.

[9] W. Litwin, M.-A. Neimat, and D. Schneider. "LH$*$ - a scalable, distributed data structure." ACM Trans. on Database Systems (TODS) 21, no. 4 (1996): 480-525.

[10] W. Litwin and T. Schwarz. "Top k Knapsack Joins and Closure." Keynote address, Bases de Données Avancées 2010, Toulouse, Fr, 19-22 Oct. 2010 www.irit.fr/BDA2010/cours/LitwinBDA10.pdf

[11] S. Martello and P. Toth. Knapsack problems: algorithms and computer implementations. John Wiley and Sons, Inc., 1990.

[12] A. Rajaraman, J. Leskovec, and J. Ullman. Mining of massive datasets. Cambridge University Press, 2012.

[13] Vagrant. www.vagrantup.com

[14] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. "Improving MapReduce Performance in Heterogeneous Environments." In Proc. of Symp. on Operating Systems Design and Implementation (OSDI), 2008.