A Highly Available, Hash-Based, Autonomous Data Structure for Time-Bound, Brute Force Calculation in the Cloud

Silvia Grampone
Universidad Católica del Uruguay
Montevideo, Uruguay
silviagrampone@gmail.com

Witold Litwin
Université Paris, Dauphine
Paris, France
witold.litwin@dauphine.fr

Thomas Schwarz, S.J.
Universidad Católica del Uruguay
Montevideo, Uruguay
TSchwarz@ucu.edu.uy

***Abstract*— The apparently unlimited resources of modern clouds bring distributed computing to the masses. We propose a new data structure for the organization of embarrassingly parallel calculations based on the cloud paradigm of easily creating and destroying virtual machines rented at low cost.**

The data structure has nodes materialize, scan, and discard virtual records and aggregate the scan results within a time limit provided by the client and is an instance of a scalable distributed virtual data structure. It is based on extensible hashing. We show how to deal with variances in the node capacities and how to make the structure resilient to failures.

***Index Terms*—Knapsack Problem; Integer Optimization; Cloud Computing; Scalable Distributed Virtual Data Structure**

I. Introduction

Cloud computing has brought distributed computing to the masses. The availability of cheap computing cycles for rent allows for calculations that before were only possible on specialized hardware or would tie up a computer for a long time. Now, these types of computations can be embedded into data analysis or even into database queries [27] if we can guarantee that they terminate in a user-defined time limit.

Even if the computational task is "embarrassingly parallel" and can be easily distributed over a large number of cores, the developer needs to write code that allocates the resources, distributes the work over a large number of cores, and deals with node failures. We propose here to simplify the development process with a structure that autonomically allocates and uses resources so that a certain programming task finishes within a user-defined limit. We concentrate on solving a very general class of optimization problems, namely zero-one-integer programming problems that are of great importance in Operations Research. Many optimization problems, including the knapsack problem and the traveling salesman problem can be expressed in this form.

Our structure is motivated by Scalable Distributed Data Structures (SDDS). SDDS store records in buckets distributed autonomously over the nodes in a distributed system. Records consist of a record identifier and a content field. Besides the key-based operations of insertion, retrieval, update, and deletion of records, they also allow a scan operation. The scan operation passes through all records (even those relocated after the beginning of the scan) and is used to select records or aggregate records for example for a top-k query or the calculation of a sum, average, or standard deviation.

Scalable, distributed, virtual data structures use virtual records instead of user-generated ones [16]. A typical use of the data structure is the optimization of an objective function over a finite, but large set. Each virtual record encodes an element of the set over which we optimize. The data structure distributes the virtual records over a number of nodes and then starts the scan operation. This operation materializes the virtual records at a node to obtain a description of a set element, evaluates the objective function, and maintains a list of top k best local results. In a final phase, the local results are agglomerated to a global result, that is then returned to the user. The data structure is responsible to extend the number of nodes to limit the scan time to a user defined limit (such as ten minutes) and to allow the calculation to be done even in the presence of node failures. Another use of scalable, distributed, virtual data structure is the inversion of a hash function. (We used this previously for a key-backup scheme [15]). Formally, this is still an optimization problem where the records describe possible arguments to the hash function and the objective function is one if the hash equals the value to be inverted and zero if not.

The structure we present here, called Scalable Virtual Distributed Hashing (SVDH) is based on Extendible Hashing [12] and the distributed version of linear hashing, LH$*$ [24] [21]. Extensible Hashing divides records in buckets and splits buckets in about even halves in order to grow the file. This form of splitting makes it especially attractive for solving certain zero-one-integer optimization problems. This is because a virtual record consists of an assignment of the binary values 0 or 1 to a set of variables and those assigned to a specific node are the ones where a subset of the variables have fixed values. The local calculation can thus simplify the evaluation by substituting the values for these variables.

Nodes are virtual machines that need to coexist with other virtual machines on the same physical nodes and compete for

resources such as memory bandwidth. Even if all physical machines are identical, each virtual machines has a different capacity for computation. SVDH allocates autonomously sufficient nodes in order to guarantee that the calculations terminate within a time limit set by the user, for example, 10 minutes.

Cloud computing environments are harsh and all applications need to be able to withstand loss of one, a few, or many computing nodes. For instance, Birman asserts that it is acceptable for a data center administrators to shut down a significant proportion of machines in order to recover from unstable situations [3]. Applications simply need to be able to recover from deliberate or accidental node unavailabilities. SVDH places nodes in small buddy groups that observe each other's progress, safeguard the others' progress, and can restart the saved calculation at another node in case of failure. With an added safety margin, SVDH can then guarantee that even in the presence of failed nodes, the computations terminate in time with a chosen high probability.

The rest of the article is organized as follows: We give the definition of SVDH in Section 2. Section 3 evaluates the performance of SVDH, especially node utilization. Section 4 evaluates its resilience to failure – due to random churn and to related outage. We discuss related work in Section 5 and then conclude.

II. A HASH-BASED SCALABLE DISTRIBUTED VIRTUAL DATA STRUCTURE

Scalable Virtual Distributed Hashing (SVDH) adapts extendible hashing [12] and scalable distributed data structures [24], [21] to the distribution of work in a discrete optimization by brute force. It uses hash partitioning to assign efficiently points in the search space to nodes (virtual machines) in a cloud data center. Each search point is treated as a virtual record and the assignment is based on hashing.

A. Example: Zero-one integer optimization

We use Zero-One Integer Optimization (ZOIO) as the target application. Many problems in Operations Research and Computer Science ranging from airline crew scheduling to register allocation by a compiler can be expressed as integer optimization. Even the traveling salesman problem is such a problem [29]. ZOIO consists in maximizing or minimizing an objective function over a number of variables that can only take values 0 and 1 and that are subject to a set of restraint inequalities:

$$\begin{aligned} f_{\text{opt}}(\mathbf{x}) &\rightarrow \max \\ f_{c_1}(\mathbf{x}) \leq d_1, &\ \ldots, \ f_{c_m}(\mathbf{x}) \leq d_n \\ \mathbf{x} &\in \{0,1\}^n \end{aligned}$$

A classical example of ZOIO is the zero-one knapsack problem where we are required to select a subset of items from $\{X_1, x_2, \ldots, X_n\}$. Item x_i has weight w_i and value v_i. We are looking for the subset whose combined weight does not surpass a weight limit W and has maximal combined value. If we let the variable x_i take the value 1 if the item X_i is included in our selection and 0 otherwise, then the problem becomes

$$\begin{aligned} w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n &\leq W \\ v_1 \cdot x_1 + v_2 \cdot x_2 + \ldots + v_n \cdot x_n &\rightarrow \max \\ x_1, x_2, \ldots x_n &\in \{0,1\} \end{aligned}$$

This knapsack problem is NP-complete even though the single constraint and the objective function are linear. It can be solved in pseudo-polynomial time by dynamic programming. A database applications would be a query that lets a department head select gifts for all her department members of a total value of less than 100 $ out of a gift catalogue.

As another example, we use a crew assignment problem. Assume that a company has two flights 1 and 2 from Montevideo to Buenos Aires every day. The company has three pilots 1, 2, and 3 and three copilots 4, 5, 6. We let x_{ij} be a *decision variable* that is set to one if (co-)pilot i is assigned to flight j and zero otherwise. The condition that there is at least one pilot on flight 1 is expressed as the constraint

$$x_{11} + x_{21} + x_{31} \geq 1$$

and the condition that there are at least two pilots or a pilot and a copilot on flight 1 is expressed as the constraint

$$x_{11} + x_{21} + x_{31} + x_{41} + x_{51} + x_{61} \geq 2.$$

To require at least one of pilots 1 and 2 to be both assigned to flight 1 we can use a constraint that uses integer multiplication to express a boolean and, namely

$$x_{11} \cdot x_{21} > 0.$$

If we want to avoid that both pilots 1 and 2 are assigned to flight 1, we can instead use

$$x_{11} + x_{22} \leq 1.$$

In this manner, we can easily express any number of constraints to the crew assignment problem. We finally need an objective function that could measure the costs of a certain assignment and that we want to minimize.

While it is fairly easy to express many combinatorial problems as a zero-one integer optimization problem, efficient methods for solving them are only known for certain subproblems. Even if all constraints and the objective function are linear (and the problem is a *zero-one integer linear programming problem*), cutting plane methods and branch-and-bound problems are reasonably efficient, but still exponential. It is known that integer linear programming is NP-complete.

The capability to solve general zero-one integer optimization problems introduces additional capabilities to distributed databases such as the capability to solve knapsack problems [27].

B. Scalable Distributed Virtual Data Structures

Scalable distributed virtual data structures are structures that provide a complete scan of all records stored in them in the manner of a scalable distributed data structure [16]. However, instead of storing explicit records stored in the nodes of a distributed system (e.g. a cloud data center), they scan virtual records. SDDS principles are used to assign virtual records to nodes. The records therefore consist of an identifier and a value field. A node generates iteratively all virtual records assigned to it from the record identifiers, evaluates the records, and aggregates the evaluation values to a local result (such as a list of the top k values). At the end of the calculation, the data structure aggregates the various local results into a global result and returns the result to the user.

In the case of zero-one integer optimization, an identifier l characterizes a virtual record that assigns the i^{th} bit of l to the binary variable x_i. The record identifiers of a ZOIO with 100 variables are therefore the range $\{0, 2^{100}-1\}$. The evaluation consists of plugging the values of the binary variables into the constraints and checking whether the constraints are fulfilled. If they are, then the objective function is evaluated. The node maintains the best or the top-k values of the objective function. The global aggregation process at the end of the local calculations has the structure aggregate the local results into a global result that is returned to the user.

C. SVDH Generation

SVDH is a hash-partitioned scalable, distributed, virtual data structure distributing work over a number of nodes. Nodes are identified by integers and are created by splitting already existing nodes starting with Node 0. As in extendible hashing [12], a node is split if it is overloaded. The split then distributes the work equally over the old and the new node.

To set up a calculation, the user defines a maximum calculation time R and creates a *virtual file scheme* S_F at a *coordinator* or *master* C. By default, C becomes Node 0. In general, S_F contains the program for the scan and aggregation phases of the calculation. In our application, S_F contains the structure of the zero-one integer optimization problem. The user also sets a maximum scan time R.

Each node performs a sample of the work to be done, to determine its *throughput*, the number of records that can be processed during a time unit. The formula $B = RT$ then determines the capacity, the number of records that can be scanned during the user-set scan time limit. If currently there are L records assigned to the node, its load is $\alpha = L/B$. The node splits when $\alpha > 1$. In a split operation, a new node is allocated and half the records assigned to the current node move to the new node. The load of the splitting node is halved.

Each node identifier is an integer $i \geq 0$. As for LH$*$, we define a *level* for each node by the following recursive procedure [16]: Node 0 starts out with level 0. Whenever a node splits, its level is increased and its descendent gets also the new level. When we split Node i when it has level $l(i)$, the new node has identifier $2^{l(i)} + i$. Node i gets all records with identifier x where $x \equiv i \bmod (2^{l(i)})$.

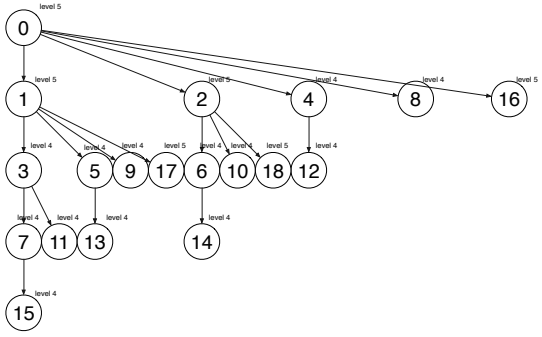

Fig. 1: A small Hash Tree

The nodes naturally organize themselves in a tree, the *Hash Tree* (Figure 1). When a node i splits, it becomes the parent of the new node $2^{l(i)} + i$. In the figure, we show a very small hash tree, where we assumed that the nodes have about the same capacities. Node 0 split five times, incrementing its level to 5 and acquiring nodes $0+2^0 = 1$, $0+2^1 = 2$, $0+2^2 = 4$, $0+2^3 = 8$ and $0+2^4 = 16$ as descendants. Node 1 started out with level one and split four times, gaining nodes $1+2^1 = 3$, $1+2^2 = 5$, $1+2^3 = 9$, and $1+2^4 = 17$ as descendants. The resulting tree is in general not balanced and Node 0 tends to have the highest number of descendants. The number of descendants and the graph distance to Node 0 always sum up to the level of a node. The load of a node with level l is always 2^{-l} of the total load. If each node has exactly the same capacity, then all nodes have the same level and their number is a power of two. If they are about equal, then in general their levels are two values l_0 and $l_0 + 1$.

In the global accumulation phase, nodes pass their local results to their parent in the tree, until the results reach Node 0, which then sends the agglomerated result to the user. In more detail, nodes without children send their local results to the parent, once they have terminated the scan of their assigned records. A node with children does the same, once it has obtained the results from all its children (and finished its scan). When a node has received results from all its children and its own results are available, it creates an agglomerated result, that is then passed up to its parent or to the user in the case of Node 0.

In the case of zero-one integer optimization, the materialized virtual records are truth assignments to the Boolean variables and those assigned to Node i have $l(i)$ of these variables assigned to constants. In the situation depicted in Figure 1, Node 3 has all virtual records with $x_0 = 1$, $x_1 = 1$, $x_2 = 0$, and $x_3 = 0$, since it has level 4. By plugging in these values, we obtain a problem with $l(i)$ less variables.

D. Using buddy groups for reliability

Cloud data centers are harsh environments as for example shutting down nodes is seen as a legitimate means for solving instability problems [3]. We use the idea of process groups [2] to provide failure tolerance by grouping nodes into buddy groups. We protect against individual nodes or a proportion of all nodes becoming unresponsive (for example because the virtual machine was deleted). Failure tolerance is simplified

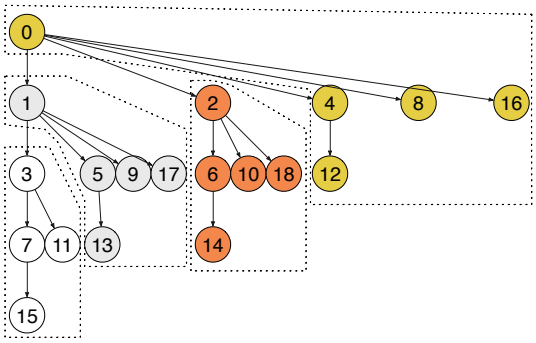

Fig. 2: Buddy group formation

by the at-least-once nature of brute force searches. A virtual record can be scanned several times without altering the result (idempotency). Failure tolerance is also simplified because an optimization result is just the record identifier of the optimizing record or possible a top-k set of such results. If two nodes run in parallel on the same task we do not incur additional costs beyond the rental costs of the additional node.

Buddy groups of nodes run membership and consensus protocols such as Paxos or Raft [4], [18], [31] in order to ensure that no member node of their group can become unavailable without being replaced. These protocols have been the subject of intense studies and well working versions for smaller group sizes are available. Unfortunately, these protocols do not scale well with membership size. To limit the effects of a node unavailability, the group leader periodically scans all other nodes and has them send a progress report to all other nodes in the group. This constitutes a *work window*. The progress report consists of the part of the search space already scanned and the partial results of the search. In the case of a node failure, a replacement node thus does not begin afresh but starts at the last check point of the failed node.

A simple procedure uses the node allocation process. Every n^{th} node receives leadership status and the following $2n-1^{\text{th}}$ nodes are assigned to the two previous leaders. A leader of a group of n nodes or more splits from the previous group. The results are groups of n and a last group with size between n and $2n-1$.

An alternative procedure uses the structure of the hash trees and forms buddy groups with between four and seven members. The procedure forms groups walking up the tree. Each group has a leader for the purpose of group construction. The groups formed have between 2^l and $2^{l+1}-1$ nodes. Anticipating our reliability results from Section IV, a reasonable value would be $l=2$ and the groups would have between four and seven elements. Initially, we form protogroups consisting of all those nodes that have no children. Each of these protogroups has only this node as a child. For the small tree given in Figure 1, these protogroups are $\{15\}$, $\{11\}$, $\{13\}$, $\{9\}$, $\{17\}$, $\{10\}$, $\{18\}$, $\{12\}$, $\{8\}$ and $\{16\}$. For the following step, each leader of a group with less than four members sends a "join" message to its parent. A leader of a group with more than four members sends a "go-ahead" message to its parent. A node which is a parent waits to receive messages from all its children. If a child sends a "join" message, it makes itself the temporary leader of a protogroup that contains the union of the groups of all children that send a "join" message. If a node that is a parent receives only "go-ahead" messages, it makes itself the leader of a protogroup with only itself as member. In our example, Node 0 will receive join messages from Nodes 8 and 16, but cannot proceed until it has received messages from Nodes 1, 2, and 4. Node 7 forms a group $\{7,15\}$, Node 5 a group $\{5,13\}$, Node 6 a group $\{6,14\}$, and Node 4 a group $\{4,12\}$. Once Node 7 has formed this group, it sends a "join" message to Node 3. Node 3 now has received messages from its children and forms a group $\{3,7,11,15\}$. It then sends a "go-ahead" message to Node 1. Node 1 receives join messages from Nodes 5, 9, and 17. This leads to the formation of group $\{1,5,9,13,17\}$ and lets Node 1 send a "go-ahead" message to Node 0. Similarly, Node 2 will become the leader of a group $\{2,6,10,14,18\}$. Node 0 will receive a "join" from Nodes 4, 8 and 16 and forms a group $\{0,4,8,16,12\}$. Figure 2 gives this example. The procedure is guaranteed to place every node in a group with four to seven members with the exception of the final protogroup containing Node 0. If this protogroup does not have enough members, it unifies with the group of an arbitrarily chosen child of Node 0. The union has either seven or less members or it has eight or more. In the first case, we are finished, in the second case, we only need to divide the last group. This procedure is guaranteed to work if there are at least four nodes in the tree. If there are N nodes, it will send $N-1$ "join" or "go-ahead" messages. Since each node receives or sends at most a constant number of messages, the procedure is scalable.

E. Message passing and phases of calculation

The lifetime of an SVDH structure naturally falls into various phases. The generation phase starts with the client creating the coordinator node through the cloud mechanism and handing it the virtual file scheme that defines the calculation. Nodes that have sufficient capacity and who have received a message from all their children that they also have finished the generation phase, report this to their parent. When the coordinator has received this message from all of its children, the coordinator starts the next phase that creates the buddy groups. At this moment, the system has become failure tolerant. Before, a practical implementation has nodes regenerate children with whom they cannot communicate, but at the small risk of having nodes allocated that are descendent of a failed node. Presumably, the interface to the cloud service has a way to detect the existence of such rogue nodes after a while. Nodes that do not receive acknowledgment of their communication to their ancestor will also eventually deallocate themselves automatically. The calculation phase proper starts at each node once it has joined a buddy group. The coordinator plays no special rôle in it until the calculation comes to an end.

The tree structure is used for the aggregation of results. Each leaf node that terminates its calculation sends its local optimization results to its parent. A parent sends a similar message to its parent if all its children have send their results,

the node itself has finished its calculation, and aggregated the various local results.

This tree structure can also be useful for variants that try to exploit the algebraic structure of the computation problem. For example, if we are dealing with 0-1 integer optimization, then we can have nodes accumulate their (temporary) local optima after the first work window and have the coordinator return the (temporary) global optimum to all other nodes. The nodes then can determine whether their local optimization problem can possibly beat the already seen temporary optimum. If this is not the case, then the node can terminate its calculation since it cannot possibly contribute. This is a variant of the well-known "branch-and-bound" method of optimization.

F. Capacity changes

Nodes are virtual machines collocated with other machines and suffer potentially from the effects of collocation by cache contention, network contention, etc. In this paper, we assume that the capacity of nodes varies (within bounds established for example by quality of service guarantees) among nodes and during the lifetime of a computation. However, we assume that the latter are rare events, triggered by moving the virtual machine to a new node or adding or removing other virtual machines at the same host.

During the calculation, a monitor module keeps track of the speed of the calculation. If this speed varies from the initial estimate, the monitor triggers a local restructuring of the data structure. If the capacity is no longer sufficient to finish the calculation in the user-defined limit, then the node splits. If the node has excess capacity and has descendants, then the node contacts the latest descendant (which is the one with the highest identifier) and sees whether it does not have the capacity to deal with its load as well. If this is the case, then the latest descendent leaves its buddy group and shuts down. If the node has excess capacity and no descendent, then it contacts its parent to see whether this parent has the capacity to deal with its own and the node's load, in which case the node leaves the system and the parent assumes the complete load. In summary, the detection of excess capacity triggers an attempt to reverse the last split.

Below, we show that *merging nodes* – the step of undoing the last split if possible – is important. If not, the overall loss in utilization is noticeable, especially when the variance of node capacities is high. With it, a wave of changes in the node capacities leads to a readjustment with statistically same node utilization as before.

G. Fast initialization

If SVDH uses a total of N nodes, the critical path in the timeline will run through $\lceil \log_2(N) \rceil$ node creations. While cloud applications generate new nodes fast, they still take some time. The time to set up a virtual machine appears to be 2-3 seconds based on experiments for the popular Vagrant VM [14]. If the coordinator has decided that its load factor is N, it can directly request the creation of $2^{\lfloor \log_2(N) \rfloor - 2}$ and assign them to the upper levels of the Hash tree. This creates SDVH faster than previously explained.

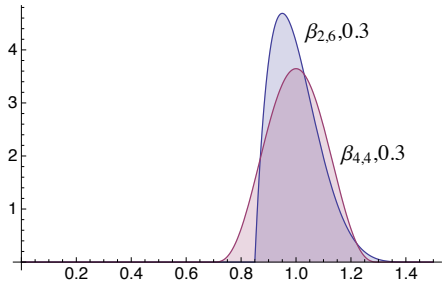

Fig. 3: PDF of $C_{2,6,0.3}$ and $C_{4,4,0.3}$ used as random variables representing actual node capacity. The former has support in the interval $[0.85, 1.45]$ and the latter in $[0.7, 1.3]$, but both have mean 1.

III. Performance Evaluation

To evaluate the utilization of the cloud resources using SVLH, we simulated the assignment of nodes for a small system (using Python 3) standardizing the average node capacity to 1 and distributing a total load of L between 500 and 2200 over the system. If all nodes have exact capacity one, then the system is deterministic and the system will use exactly $2^{\lceil \log_2(L) \rceil}$ nodes, yielding an average utilization that varies between 1, if L is an exact power of two and $0.5 + \varepsilon/L$ if L is an integer power of two plus ε. The average value for the utilization is 0.75. As we introduce variation into the node capacities, the resulting sawtooth graphs get spread out, but this pattern is still quite visible for distributions with smaller variance, as Figures 4 and 5 show.

As we do not have measurements on the capacity of virtual machines in actual data centers, and as we also desire to develop a data structure for general systems, we choose to model uncertainty by a family of distributions that are unimodal (the probability density function has a single hump) and have support in a limited interval. The latter because a virtual machine has a maximum performance and because quality of service guarantees will not allow its performance to fall below a certain level without migrating it to a less loaded computer. An example for such a distribution is the truncated normal distribution. Here, we use the beta distribution since a beta distribution with a fixed support is determined by two positive shape parameters α and β that yield a wide variety of distributions. Our experiments suggest that only the general shape of the distribution determines the behavior of the system.

A beta distribution defines a stochastic variable X between 0 and 1. It has mean $\mu(\alpha,\beta) = \alpha/(\alpha+\beta)$, variance $\sigma^2 = \alpha\beta(\alpha+\beta)^{-2}(1+\alpha+\beta)^{-1}$, and skewness $2(\beta-\alpha)(\alpha+\beta+1)^{1/2}(\alpha+\beta+2)^{-1}(\alpha\beta)^{-1/2}$. Since we model distributions with mean 1, we choose a stochastic variable $X_{\alpha,\beta}$ with beta-distribution with shape parameters α and β and consider the stochastic variables

$$C_{\alpha,\beta,s} = 1 - 2s\mu(\alpha,\beta) + 2sX_{\alpha,\beta}$$

which is a translated beta-distribution with mean 1. The parameter s is the "spread". We give an example in Figure 3,

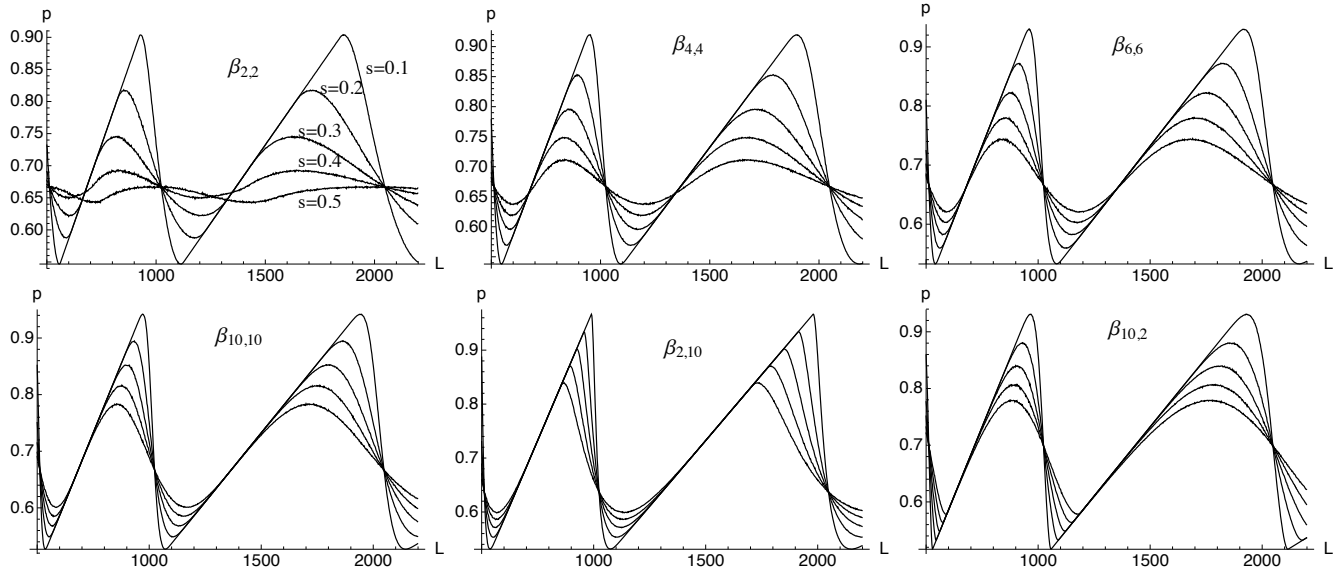

Fig. 4: Utilization in dependence of number of nodes. The nodes have capacity distributed by various beta distributions with average 1 and various spreads.

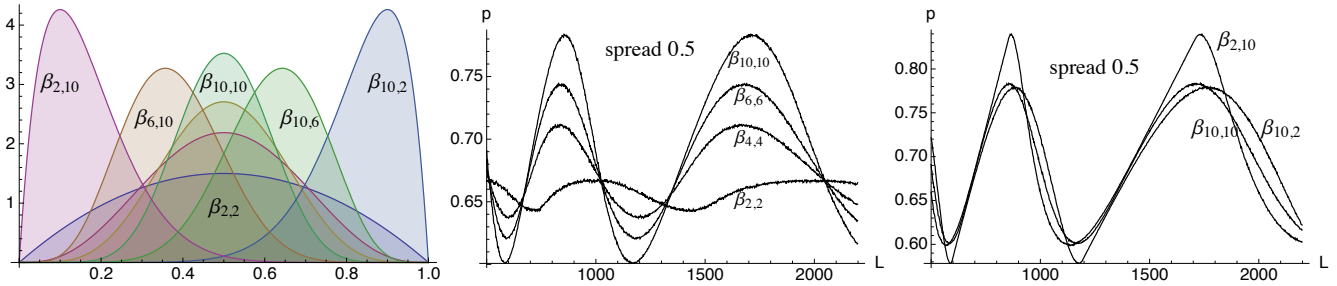

Fig. 5: PDF of the various beta-distributions used (left) and comparison of utilization for symmetric and for asymmetric beta distributions with spread 0.5

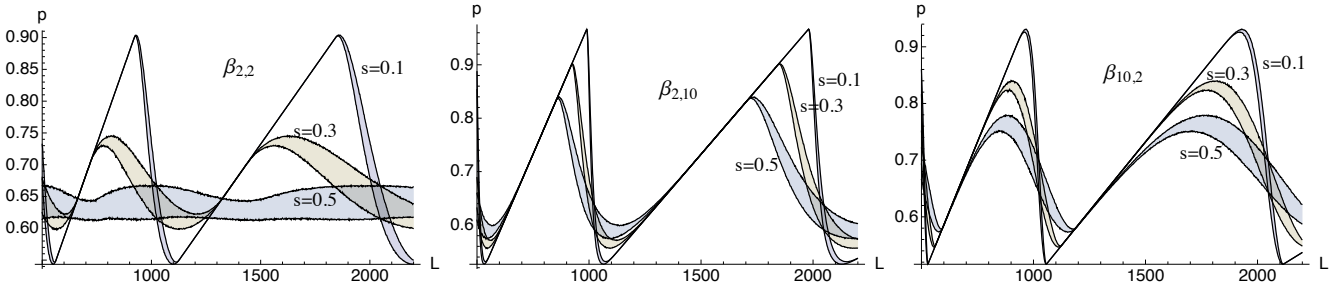

Fig. 6: Node utilization before and after 50% of the nodes change capacity after the initial assignment and before node merges for the various beta distributions and spreads s indicated. The before value and after values are connected through a filling.

where we give the probability density function of $C_{2,6,0.3}$ and $C_{4,4,0.3}$. The mean of the beta distribution with shape parameters 2 and 6 is $1/4$. To obtain a random variable with mean 1 and a support interval (where the PDF is positive) of size 2×0.3, we need it to start at $1-0.6\times 0.25=0.85$ and end at 1.45. Similarly, $C_{4,4,0.3}$ has support $[0.7,1.3]$. Both random variables have mean 1. The spread s is always half the size of the support interval.

We used these random variables to simulate the utilization of nodes in a system with initial load $L\in[500,2200]$ by generating node capacities using these translated beta distribution. Figure 4 gives the results. Each graph shows the results using one beta-distribution and for spreads 0.1, 0.2, 0.3, 0.4, and 0.5. The distribution with lowest variance, i.e. with spread 0.1 follows most closely the sawtooth curve where the distribution with highest variance, i.e. with spread 0.5 exhibits the least dependence on the initial load. The average utilization over loads from 1024 to 2048 given in Table I is lower with higher variance.

In Figure 5 we first show the PDF of the beta distributions use, and then a comparison between the results obtained with symmetric distributions and asymmetric distributions using

Parameters	0.1	0.2	0.3	0.4	0.5
baseline					
2,2	0.728	0.707	0.689	0.673	0.658
4,4	0.733	0.718	0.703	0.690	0.678
6,6	0.736	0.723	0.711	0.699	0.688
10,10	0.739	0.729	0.719	0.709	0.700
2,10	0.740	0.731	0.722	0.714	0.705
6,10	0.738	0.727	0.717	0.707	0.697
10,2	0.739	0.729	0.720	0.711	0.702
10,6	0.738	0.727	0.716	0.706	0.696
with 10% change					
2,2	0.726	0.704	0.683	0.665	0.649
4,4	0.732	0.715	0.699	0.684	0.671
6,6	0.735	0.721	0.707	0.695	0.683
10,10	0.738	0.727	0.716	0.705	0.696
2,10	0.740	0.730	0.720	0.711	0.702
6,10	0.737	0.725	0.714	0.703	0.692
10,2	0.738	0.727	0.717	0.707	0.697
10,6	0.737	0.725	0.713	0.702	0.691
with 50% change					
2,2	0.718	0.689	0.662	0.638	0.615
4,4	0.726	0.704	0.683	0.663	0.645
6,6	0.730	0.711	0.693	0.677	0.661
10,10	0.734	0.719	0.705	0.691	0.678
2,10	0.737	0.724	0.711	0.699	0.687
6,10	0.734	0.718	0.703	0.688	0.674
10,2	0.734	0.720	0.705	0.692	0.679
10,6	0.733	0.717	0.701	0.686	0.672

TABLE I: Average utilization rates without changes and with changes before merges.

spread 0.5. The y-axis has a different range, but we can clearly see that with high variance, the maxima and minima of node utilizations are slightly shifted. On the right, we see that the distribution with strong positive skew (with parameters $\alpha = 2$ and $\beta = 10$) approaches most the behavior of the deterministic case.

We then investigated the effects of load changes in the middle of the computation when node merges were not used. We assume that 50%, 20%, and 10% of the nodes change their capacity. A node splits if it can no longer meet the deadline because its load now exceeds its capacity. As we can see from the numbers in Table I, the average utilization diminishes. Figure 6 gives some of our results graphically.

If monitoring the progress at a node discovers that its capacity has increased, we try to merge this node with its youngest child, which is the one with highest node identifier. If the load of both nodes can be handled at the parent, then we free the child. With this operation, we obtain utilization numbers after a wave of capacity changes that are statistically indistinguishable from the utilization numbers before the wave of capacity changes. We do not give before and after graphs because the corresponding curves are indistinguishable. We used quantitative statistics to calculate 95% confidence intervals to ascertain this, but we present here only a qualitative statistics in Figure 7. There, we plot the node utilization before and after a wave of random capacity changes and see that about half the utilization values improve and about half deteriorate after the change. The utilization oscillates $\pm 0.5\%$ around 100% without any discernible pattern safe for differences in the amplitude. This makes the case for the assertion that our structure deals well with a wave of capacity changes.

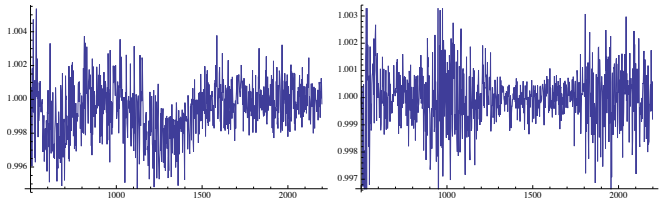

Fig. 7: Relation of node utilization after and before a capacity change affecting 50% of all nodes using a beta distribution with shape parameters 2 and 2 (left) and 2 and 10 (right).

IV. Reliability Evaluation

Failure tolerance is based on the existence of buddy groups. Periodically, all members of a buddy group distribute an aggregation of the scan results so far and the range of the records scanned to all other members of the group. We call the time between these distributions a *working window*. If a buddy group decides that a node is no longer responsive (and has presumably failed or become unreachable), they create a replacement node. The replacement node obtains the latest aggregation result and the range of already scanned records from the surviving members of the buddy group and starts scanning at the end of the scan range.

In the aggregation phase, nodes that are not sending their accumulated results are also probed and possibly restarted. This happens if a complete buddy group has become unreachable. Since in this case no interim results are available, the calculation at these nodes has to start fresh. This happens only rarely and the system can use more nodes for these calculations in order to terminate earlier.

A. Cloud Failure Models

An accurate, general failure model for nodes in a cloud data center does not exist. Vishwanath and Nagappan studied hardware failures in a datacenter and found that disk drives were the dominant failure cause [36]. They also found that a server who suffered one disk failure is more likely to suffer another one. Temperature can be a major reason for this observation and can even have a stronger effect [33]. For the time length that we are studying (compute jobs that take tens of minutes or maybe a couple of hours), the resulting effect would still be slight. Nagappan, Peeler, and Voulk also observed temporal peaks of unavailabilities at the application level, namely for the virtual computing lab) [30]. Rabkin and Katz monitored six months of a Hadoop ecosystem and identified misconfigurations and bugs as the major causes of failures [32]. Two incidents show that cloud failures can be drastic and lead to complete outages. Amazon's EC2 suffered an such an outage in 2011 through a human error causing a misconfiguration and an automatic misdiagnosis resulting in massive recovery effort for servers that had not failed [1]. Google suffered a massive service interruption to Gmail in 2009 when routine maintenance resulted in a larger than expected change in the traffic load at some routers, who became overloaded and unresponsive. [7].

While hardware failure apparently follows a Weibull distribution, our short time frames (at the most a few hours) allow

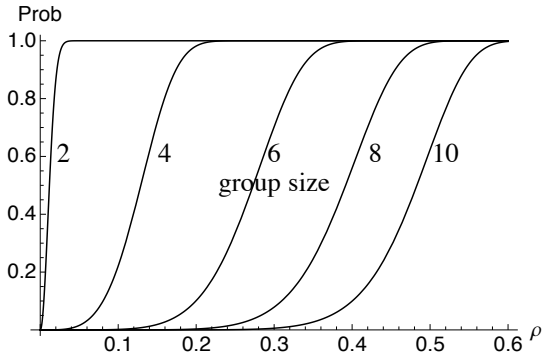

Fig. 8: Probability of loss of at least one complete group with 10000 nodes depending on individual failure rate ρ.

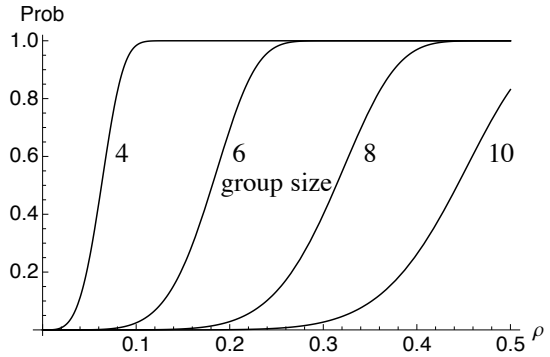

Fig. 9: Probability of loss of at least one group with 10000 nodes at a given failure rate ρ during a work and check window.

us to assume constant failure rates. At least some software failures such as misconfigurations appear to lead to large or even catastrophic outages. In the absence of better information for modeling, we test our reliability scheme against two different scenarios. The first is a scenario where a large proportion of nodes suddenly becomes unavailable. The second scenario assumes independent failures at a given failure rate and models failures as a Poisson process. The truth is some combination of both scenarios and a scheme that is reasonably resistant to both should do well for the general failure pattern. Finally, a massive service disruption at the scale that were suffered by EC2 and Gmail is not controllable.

B. Consequences of related failure

A related failure will occur at a portion of ρ of all nodes. Besides the usual physical reasons (such as failure of central network equipment), this failure model occurs if the data center administrator decides that the whole center is in an unstable state and arbitrarily shuts down a number of virtual machines. The probability that a group of k buddies has at least one surviving member is $1-\rho^k$ and the probability that all g groups have at least one survivor is the g-th power of that. The opposite is the probability that during such a wave of failures at least one group has lost all intermediate results. We obtain the characteristic s-curves that show the heavy dependance on the group size (Figure 8), but reasonable resistance. In these calculations, we assume that the buddy group size is constant, whereas in reality a buddy group can be almost twice as big. The bigger buddy groups however are even more resilient.

C. Consequences of unrelated failure

We assume that nodes fail individually at a rate ρ. We recall that N nodes are organized into N/k groups of k nodes. If during a work window of time t_w, no node has failed, the group passes onto the next window. There is total of n_w working windows. However, if at the end of a work window reports are missing, then a group membership protocol is triggered with possible replacement of nodes deemed to have failed. This takes time t_c. If during this time, none of the k involved nodes fails, then the next work window starts. However, if there was a failure, then the process is repeated. As in fact calculations proceed in parallel to running a group membership protocol, this model systematically overestimates the effects of failure on completion time.

We first calculate the probability that a group completely fails. We distinguish the following probabilities for the calculation that follows. Let p_{tw} be the probability that all k nodes fail during a working window. Similarly, p_{cw} denotes the probability that all k nodes fail during a check window, p_w the probability that at least one, but not all k nodes fail during a working window and p_c the probability that at least one, but not all k nodes fail during a check window. If there are no node failures during the working window, then the group survives. If all nodes fail during the working window, then we have failure. If some but not all nodes have failed, then we enter the check phase, consisting of one or possibly more check windows, since the check phase might need to be restarted. We have one additional check window with probability p_c, two additional check windows with probability p_c^2, three with probability p_c^3 etc. This gives for the probability of failure during one working window and the check phase

$$P_1 = p_{tw} + p_w(p_{tc} + p_c p_{tc} + p_c^2 p_{tc} + \ldots) = p_{tw} + \frac{p_w p_{tc}}{1-p_c}$$

The probability that no group fails during the calculation is

$$P = 1 - (1-P_1)^{n_w N/k}$$

As we can see from the example in Figure 9, the probability for loss of a complete group depend heavily on the group size (as should be expected). The figure of course includes failure rates that are unrealistically low.

We now calculate the expected delay for a single group. To have a delay at all, we need to have a partial failure (at least one node, but not all) during the work window. The number of repetitions then depends on whether there was a partial failure during the time for checking and reestablishing membership. Being a bit pessimistic, we assume that the number of nodes here has not changed. Mathematically, we obtain for the average number of times that we have to reenter checking

$$p_w(1 + 2p_c + 3p_c^2 + 4p_c^3 + \ldots) = \frac{p_w}{(1-p_c)^2}.$$

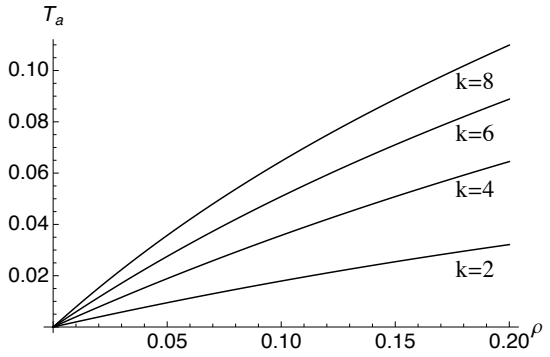

Fig. 10: Expected added time for group sizes $k = 2,4,6,8$ for a work window time of 1, a check membership time 0.1 and an independent node failure rate of ρ.

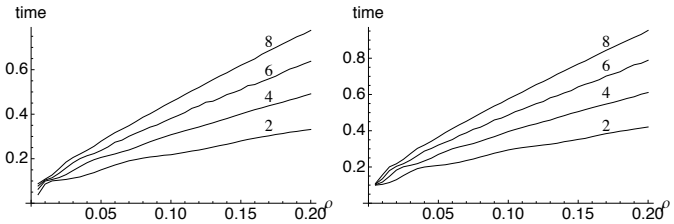

Fig. 11: Simulated *maximum* added time for group sizes $k = 2,4,6,8$ for a computation with ten work windows, each with a work window time of 1, a check membership time 0.1 and an independent node failure rate of ρ for an SDVH structure with $10,000$ (left) and $100,000$ (right) nodes.

For realistic failure rates and group sizes, the average additional time per work window is quite small as the example in Figure 10 demonstrates. Of course, we need to multiply this number by the number of work windows. However, these are average numbers and in a concrete calculation, the additional time is given by the worst additional time for potentially tens of thousands of groups. Obtaining an analytic solution for the expected worst time seems to be beyond our current mathematical abilities and we turned to simulation instead.

The results are shown in Figure 11 where we simulated the behavior of an SDVH structure with 10,000 and 100,000 nodes and various group sizes k for a calculation consisting of ten work windows. Figure 11 gives the average *maximum* increase in any of the $10000/k$ or $100000/k$ groups and shows these maxima to be about three times larger than the average values obtained by multiplying the analytical results displayed in Figure 10 by the number of windows. In the considered range for the node failure rate ρ, these values amount to less than an additional work window. This allows us to conclude that we still can guarantee task completion within the limits provided by the user by taking one additional in ten work windows into account.

V. Related Work

Scalable Distributed Data Structures (SDDS) have come into their own in recent years, though not under that name. Google's BigTable [6] is an SDDS based on range-partitioning. Just as RP$*$ [25] more than 10 years earlier, it allows efficient range queries. The same is true for MS Azur and MongoDB. Amazon's EC2 uses a distributed hash table called Dynamo [11]. VMWare's Gemfire provides its own hash scheme, etc. APIs for the popular MapReduce framework [10], in its Google or Hadoop version, have been created for some of these offerings.

Linear Hashing was used in the nineties to define a hash-based SDDS called LH$*$ [24]. Failure tolerance has been a concern from the beginning and lead to the definition of various schemes [20], [22], [23], [26]. The current best practice is LH$*_{\mathrm{RS}}$, which offers scalable high availability [21], [28], so that the resilience to failure increases with the number of nodes involved. While our proposal has high failure tolerance, this tolerance is not scalable, in part because it is not needed. While an SDDS assumes that servers are free, an SDVS has to pay rent and we are interested in limiting the number of nodes.

The idea of scalable distributed virtual data structures developed from using the difficulty of computation as a flexible protection mechanism in the context of a key recovery mechanism. The scheme used secret sharing in order to break a key into shares and store one of them at an escrow agency. To prevent the escrow agency from storing the key explicitly, the escrow agency is only given a hash of the key and a range of potential values. To recover the share, we systematically hash all values in this range until we have found the inverse of the hash and hence the key share [15], [19]. Scalable distributed virtual data structures as a method to allocate resources to the solution of some integer optimization problems were first presented in 2014 [16]. Here, we study the impact of capacity changes and failure tolerance.

The idea of grouping processes in a distributed system into groups that monitor each other is fundamental and has become an accepted tool for reliability in distributed systems, despite a basic, negative result [5]. While the use of replica and the topic of replica placement is important for P2P system, there seems to be very little literature on how to form small groups of peers. Slicing in P2P systems comes the nearest, but in general creates much larger groups [13]. The absence of literature might be because the use of replica only already solve the problem for P2P systems.

Since our proposal is not using the cloud in a traditional way, previous work on cloud failure tolerance does not apply directly. For instance, Dai and colleagues note that grid users care about services that they are using instead of the resources and this is even more the case for cloud services. They therefore develop a holistic model for calculating the probability that a cloud service can successfully complete [8], [9]. Similar models are given by Silva et al. and by Thanakornworakij [34], [35]. Jhawar and Piuri advocate fault tolerance as a service, provided possibly by the cloud provider [17].

VI. Conclusions

We have presented SDVH, a failure tolerant, self-organizing data structure for brute force calculations in the cloud and have shown it through analysis and simulation to be a feasible

way to solve optimization problems through brute force. Our method extends the range of solvable problem instances by the number of virtual machines that one can afford to rent. For a zero-one integer optimization problem, this increases the number of variables from 10 (1024 nodes) to 20 (1048576 nodes). The data structure uses a key feature of cloud computing, the capability to join and remove nodes on demand. It appears that SDVH is ready for implementation and use.

Future work will concentrate on building an actual implementation, though the experience with the LH$*$ family suggest that this is a multi-year effort. We are currently working on a distributed, robust protocol for the formation of buddy group with few messages. On the analytical side, native support for branch-and-bound methods need to be evaluated. We are planning on using the linear hash tree in order to gather periodically the globally best seen results and distribute them to all nodes. These can then evaluate their local problem set to see whether they have a chance to beat the already seen best results. The efficiency of this method will be very problem dependent.

REFERENCES

[1] C. Babcock, "When Amazon's cloud turned on itself," *Information Week*, vol. 31, June 2011.

[2] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, vol. 36, no. 12, pp. 37–53, 1993.

[3] ——, *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. Springer, 2012.

[4] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 398–407.

[5] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 322–330.

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

[7] T. Claburn, "Gmail outage a big deal, says google," *Information Week*, February 2009.

[8] Y.-S. Dai, M. Xie, and K.-L. Poh, "Reliability analysis of grid computing systems," in *Dependable Computing, 2002. Proceedings. 2002 Pacific Rim International Symposium on*. IEEE, 2002, pp. 97–104.

[9] Y.-S. Dai, B. Yang, J. Dongarra, and G. Zhang, "Cloud service reliability: Modeling and analysis," in *15th IEEE Pacific Rim International Symposium on Dependable Computing*, 2009.

[10] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] 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 *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[12] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashinga fast access method for dynamic files," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.

[13] A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, and M. Rayna, "Distributed slicing in dynamic systems," in *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*. IEEE, 2007, pp. 66–66.

[14] W. Hajaji, "Scalable partitioning of a distributed calculation over a cloud," Master's thesis, Université Paris, Dauphine, France, 2013.

[15] S. Jajodia, W. Litwin, and T. Schwarz, "Recoverable encryption through a noised secret over a large cloud," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems IX*. Springer, 2013, pp. 42–64.

[16] ——, "Scalable distributed virtual data structures," in *Big Data Science and Computing, Proceedings., Second ASE International Conference on*, 2014.

[17] R. Jhawar and V. Piuri, "Fault tolerance management in IaaS clouds," in *Satellite Telecommunications (ESTEL), 2012 IEEE First AESS European Conference on*. IEEE, 2012, pp. 1–6.

[18] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[19] W. Litwin, S. Jajodia, and T. Schwarz, "Privacy of data outsourced to a cloud for selected readers through client-side encryption," in *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*. ACM, 2011, pp. 171–176.

[20] W. Litwin, J. Menon, T. Risch, and T. J. Schwarz, "Design issues for scalable availability LH* schemes with record grouping." in *Workshop on Distributed Data and Structures*, 1999, pp. 38–55.

[21] W. Litwin, R. Moussa, and T. Schwarz, "LH*RS — a highly-available scalable distributed data structure," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 3, pp. 769–811, 2005.

[22] W. Litwin and M.-A. Neimat, "High-availability LH* schemes with mirroring," in *Cooperative Information Systems, 1996. Proceedings., First IFCIS International Conference on*. IEEE, 1996, pp. 196–205.

[23] W. Litwin, M.-A. Neimat, G. Lev, S. Ndiaye, and T. Seck, "LH*S: a high-availability and high-security scalable distributed data structure," in *Research Issues in Data Engineering, 1997. Proceedings. Seventh International Workshop on*. IEEE, 1997, pp. 141–150.

[24] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH* a scalable, distributed data structure," *ACM Transactions on Database Systems (TODS)*, vol. 21, no. 4, pp. 480–525, 1996.

[25] W. Litwin, M.-A. Neimat, and D. Schneider, "RP*: A family of order preserving scalable distributed data structures," in *Very Large Databases*, vol. 94, 1994, pp. 12–15.

[26] W. Litwin and T. Risch, "LH* g: A high-availability scalable distributed data structure by record grouping," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 4, pp. 923–927, 2002.

[27] W. Litwin and T. Schwarz, "Top k knapsack joins and closure," *Keynote address, Bases de Données Avancées*, 2010. [Online]. Available: "www.lamsade.dauphine.fr/ litwin/Top k Knapsack Join and Closure8.pdf"

[28] W. Litwin, H. Yakouben, and T. Schwarz, "LH* RS P2P: a scalable distributed data structure for p2p environment," in *Proceedings of the 8th international conference on New technologies in distributed systems*. ACM, 2008, p. 1.

[29] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer programming formulation of traveling salesman problems," *J. ACM*, vol. 7, no. 4, pp. 326–329, Oct. 1960.

[30] M. Nagappan, A. Peeler, and M. Vouk, "Modeling cloud failure data: a case study of the virtual computing lab," in *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*. ACM, 2011, pp. 8–14.

[31] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings Usenix Annual Technical Conference (ATC)*, 2014.

[32] A. Rabkin and R. Katz, "How hadoop clusters break," *IEEE Software*, vol. 30, 2013.

[33] S. Sankar, M. Shaw, K. Vaid, and S. Gurumurthi, "Datacenter scale evaluation of the impact of temperature on hard disk drive failures," *ACM Transactions on Storage (TOS)*, vol. 9, no. 2, p. 6, 2013.

[34] B. Silva, P. Maciel, E. Tavares, and A. Zimmermann, "Dependability models for designing disaster tolerant cloud computing systems," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–6.

[35] T. Thanakornworakij, R. F. Nassar, C. Leangsuksun, and M. Păun, "A reliability model for cloud computing for high performance computing applications," in *Euro-Par 2012: Parallel Processing Workshops*. Springer, 2013, pp. 474–483.

[36] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 193–204.