Applying Graph Databases to Cloud Management: An Exploration

Vijayaraghavan Soundararajan and Shishir Kakaraddi
VMware, Inc.
Palo Alto, CA, USA
{ravi,skakaraddi}@vmware.com

***Abstract*— Graph databases have become increasingly popular for a variety of uses ranging from modeling online code repositories to tracking software engineering dependencies. These areas use graph databases because many of their problems can be expressed in terms of graph traversals. Recent work has applied graph databases to virtualization management, noting that many IT questions can also be expressed as graph traversals. In this paper, we study another area in which graphs are valuable: reporting and auditing in cloud infrastructure. We first examine cloud infrastructure and map its data model to a graph. Building upon this model, we recast a number of reporting queries in terms of graph traversals. We then modify the model both for performance and for accommodating additional use cases related to cloud computing, including migration from private to hybrid clouds. Our results show that while a graph backend makes it straightforward to formulate certain kinds of queries, a naïve mapping of graphs to a graph database can result in poor performance . Utilizing knowledge of the problem domain and restructuring the graph can provide dramatic gains in performance and make a graph database feasible for such queries.**

Keywords- Graph databases, Virtual Machine management, cloud computing, datacenter management tools

I. INTRODUCTION

One of the most common tasks of IT administrators is monitoring and auditing the systems that they manage. Given the scale of today's datacenters and cloud environments, simply keeping track of incoming and outgoing assets can be extremely challenging. Moreover, when virtualization is utilized, the resources are even more fluid, as virtual machines (VMs) can move seamlessly across hosts and across networks.

In this work, we explore one aspect of this monitoring question: tracking the connections between entities. For example, to determine if an administrator has a disaster recovery strategy, she might ensure that each host is connected to multiple networks, and that each network is connected to a different switch. To see which racks in a datacenter are most heavily-utilized, she might examine the number of VMs that are currently running on hosts within that rack. Each of these questions involves determining the link between different assets.

Previous researchers [15] have explored this area and speculated on the use of graph databases as a means for storing and retrieving the connections between entities. Graph databases have been shown to be useful across a wide variety of problem domains due to their extensibility: accommodating new types of entities or assets is as simple as adding a new object type or specifying a new type of relationship. Moreover, for queries that are simple traversals (for example, finding hosts connected to a given VM), the syntax of graph databases is relatively straightforward. This ease of use and extensibility is particularly important in IT environments, since the number and type of assets can change fairly frequently as new devices (e.g., mobile phones, storage arrays, network switches) enter an infrastructure.

In this paper, we extend upon this previous work on applying graph databases to virtualized infrastructure. We do a more in-depth analysis of the data model of a virtualized infrastructure and examine how to map it to a graph. We present a number of use cases to validate this model and demonstrate the flexibility and ease of use of graph databases for this purpose. With simple modifications to our initial mapping, we can create more complex functionality that can aid the administrator of cloud infrastructure.

The structure of this paper is as follows. In section II, we give a brief description of graph databases and virtualization management. In section III, we describe how we map virtualization infrastructure to a graph. In section IV, we show a number of examples that exploit this graph structure. In section V, we show some preliminary performance data for some of these examples. We present related work in section VI, and we conclude in section VII.

II. GRAPH DATABASES AND CLOUD MANAGEMENT

In this section we give a brief overview of graph databases and cloud management.

A. Graph Databases

A graph database [16] is designed for the storage and retrieval of graphs. The API for a graph database allows creating, removing, and modifying nodes and relationships. The syntax for queries is amenable to graph traversals, like finding all nodes that satisfy a criterion within a certain number of hops from a parent node. For example, using the Neo4j open source graph database and its Cypher query language [9], the syntax for finding all nodes of type B within 3 hops of a given parent node A is as follows:

```
1. START a=node:indexName(name="A")
2. MATCH (a)-[*1..3]-(b)
3. WHERE b.node_type = "B"
4. RETURN distinct b;
```

The power of a graph database is in line 2 above, which finds connections of three hops or less between nodes a and b. In comparison, to express such a query using a standard SQL database can be difficult depending on the schema and

978-1-4799-3766-0/14 $31.00 © 2014 IEEE
DOI 10.1109/IC2E.2014.47

table layout, and may involve hundreds of JOIN statements to exhaustively search for anything of 3 hops or less.

Like other NoSQL databases [11][12], a graph database has no fixed schema. Adding attributes to nodes requires key-value pair manipulation. For example, in Neo4j, adding a property requires the following:

```
1. START a=node:indexName(name="nodeName")
2. SET a.property_name = property_val;
```

Adding a new relationship in a graph database is equally straightforward:

```
1. START a=node:indexName(name="Node A"),
     b=node:indexName(name="Node B")
2. CREATE (a)-[r:RELATION_TYPE]→(b)
3. RETURN r
```

Graph databases have seen the most success in applications with a natural graph structure. For example, to apply a graph to a software development tracking tool [14], the nodes can be developers or files, and the relationships may reflect authorship or may reflect dependencies between files. More recently, graph databases have found application in bioinformatics for interactions between proteins [4]. It is important to note that the advantage of a graph database is not necessarily performance or functionality, since traditional relational databases can be used instead and have been highly optimized over the years. Rather, the primary advantage of a graph database is the close fit between the problem statement (navigating and modifying a graph) and the data storage model (a graph). This close fit can lower the barrier to entry for IT administrators that may be unfamiliar with all of the nuances of a traditional SQL database.

B. Cloud management

An increasing number of cloud infrastructures are built on top of virtualization and contain a large number of assets that need to be tracked. For example, a cloud computing environment based on virtualization consists of physical hosts running hypervisors, which, in turn, run VMs. The hypervisors are hosted in racks with power distribution elements and are also connected to storage devices and network switches. Cloud management software allows an administrator to probe the status of these components from a central dashboard and to perform certain types of system-management actions like live-migration of VMs [1][8]. There are a wide range of cloud management solutions, including OpenStack [10], VMware vSphere [22] and vCloud Director [19], and Eucalyptus [3].

While cloud management is one task of the IT administrator, another sometimes overlooked task is auditing and reporting. For example, to charge customers for usage, administrators may need to collect resource usage information across an infrastructure. To justify the need to buy more storage, an administrator may need to show that storage free space is running low. To ensure that an environment is appropriately configured in the case of an outage, an administrator may need to make sure that each host has multiple power supplies, connections to multiple storage arrays, and paths to multiple switches. Finally, an administrator may also need to check for compliance with certain best practices, for example, a policy in which each host should run no more than 30 VMs for best performance. Many tools exist for monitoring [18], but each environment is different, and it is difficult to fulfill the needs of every administrator. Administrators often write simple shell scripts for such auditing tasks because existing tools either do not gather the necessary information or do not provide a means to automate the retrieval and storage of such data. In this paper, our goal is to provide an intuitive and extensible means for reporting-style queries. We base our queries on our experiences with administrators that use VMware vSphere [22] for their cloud environments

III. MAPPING VIRTUAL INFRASTRUCTURE TO A GRAPH

In this section, we present a simple mapping of a vSphere virtual infrastructure to a graph backend. Though we assume vSphere, the general principles can apply to any virtual hierarchy. In vSphere, resources are organized according to function, whether it be compute, storage, or networking. For example, VMs run on physical hosts running hypervisors, and these hosts are typically organized into *clusters* to allow load balancing [17]. Clusters are in turn organized into *datacenters*, which often map to physical datacenter locations. The resulting chain of connections is as follows:

1. Datacenter → Cluster → Host → VM.

With this chain of connections, any VM can be traced to its datacenter. The datacenter-to-VM path is a strict tree, since at any one time, a cluster can reside in only one datacenter, a host can reside in only one cluster, and a VM can reside on only one host.

We next consider connections due to storage. A host is connected to logical storage elements called datastores. Datastores contain the persistent state of the VM in one or more *VMDK* files. A single VM can consist of multiple VMDKs, and a VMDK can reside on exactly one datastore. A reasonable set of connections for storage therefore includes VMs, hosts, VMDKs, and datastores, as follows:

1. VM → VMDK
2. Datastore → VMDK
3. Host → Datastore

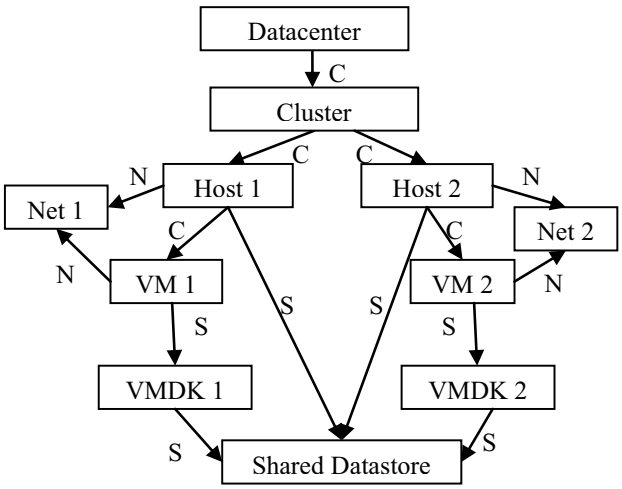

Figure 1: Virtual Hierarchy as a Graph. Relationships are labeled as compute [C], storage[S], or network [N] links. By combining the compute, storage, and networking trees into a single graph, we can eliminate some redundant paths. For example, the Host 1 → Shared Datastore link can be derived from the path Host 1 → VM 1 → VMDK 1 → Shared Datastore.

Unlike for compute, because a host can belong to multiple datastores, and datastores are shared by multiple hosts, this particular set of connections does not form a strict tree.

The final connections we consider are for networks. A host consists of multiple networks (typically one per physical device), and VMs belong to one or more of those networks (typically one per virtual device). We can characterize such connections as follows:

1. Network → Host
2. Network → VM

As in the case of storage, for networking, because hosts and VMs can belong to multiple networks, and networks comprise multiple hosts and VMs, the relationship is not a strict tree.

We can now take all of the connections described above and combine them into a graph. Consider a setup with one datacenter consisting of one cluster of two hosts, and assume each host has one VM and one network connection, with a single shared datastore across the hosts. We come up with a sample graph as shown in Figure 1. The labels "C", "N", and "S" refer to compute, network, or storage relationships.

As Figure 1 illustrates, combining compute, storage, and networking relationships into a single graph results in redundant connections. For example, there are multiple paths to a given network or datastore. To reduce the number of relationships, we can eliminate certain redundant paths. For example, the connection between Host 1 and the Shared Datastore can be derived by traversing the link from Host 1 to VM 1 to VMDK 1 to Shared Datastore. Alternatively, we can incorporate all entities into a single graph yet use different relationship types, as labeled in the figure. For example, the path from a datacenter to its VMs can be of type "compute", while the path from a VMDK to its datastore or from a host to its datastore might be of type "storage". There is a space/performance tradeoff when reducing the number of links: the overall storage required for the graph my decrease, but query latencies may increase because multiple hops are now required where a single hop would have previously sufficed. For our initial study, we preserve all connections shown in Figure 1.In addition. we use a single relationship type, rather using C, N, and S.

Armed with this simple abstraction of a virtual infrastructure, we next consider a number of use cases for this graph structure.

IV. USE CASES

In this section, we take the mapping described in the previous section and apply it to a number of practical use cases in cloud administration. For our prototype, we use the Neo4j graph database, the Cypher query language, and the py2neo [13] python bindings to Neo4j.

A. Risk Analysis

Our first example is related to risk analysis. In particular, we wish to find out which VMs would be affected if a certain datastore and network combination experienced an outage. This example appears in previous work [15], but we include it in order to show a straightforward application of our data model.

In our default graph, we have links from VMs to their VMDKs and from VMDKs to their datastores. We also have links from VMs to networks. We therefore need to find any VMs that are two hops from a given datastore and one hop from a given network. The resulting Cypher query is as follows:

```
1. START a=node:indexName(node_type="VM")
2. MATCH (a)-[*2]-(b), (a)-[*1]-(c) WHERE b.node_type
   = "datastore"  and b.moid = 10 and c.node_type =
   "network" and c.moid = 20
3. RETURN distinct a;
```

In line 1, we choose all nodes of type "VM" and call them 'a'. In this case, the type attribute is called node_type, and for faster retrieval, we create an index [node:indexName] based on this node_type value ("VM"). In line 2, we find all one-hop paths from 'a' to 'b', where b is a datastore with a given 'managed object identifier' (moid = 10). In the same line, we find all one-hop paths from 'a' to 'c', where c is a network with a given moid (20). This problem clearly illustrates the value of using a graph database: the semantic gap between problem statement and the query that solves the problem is reasonably small.

B. Simple reporting

We next give a slightly more complicated example that is representative of a common IT reporting use case. For charging purposes or for justifying the purchase of new storage arrays, an administrator might wish to know which datastores are storing the most VMs. In our graph database, we note that datastores are two hops from VMs. Thus, we traverse all datastores and find all VMs that are two hops

away. We then group the datastore and the number of VMs and return them in sorted order: In Cypher, the query is as follows:

```
1. START a=node:indexName(node_type="Datastore")
2. MATCH (b)-[*2]-(a)
3. WHERE b.node_type = "VM"
4. WITH a, count(b) AS vm_count
5. ORDER BY VM_count DESC
6. RETURN a.moid, vm_count;
```

To provide another example of auditing, we first leverage the usefulness of connections as data. Consider a linked clone [5]. A linked clone is created when one VM is cloned from another VM, but instead of allocating a new VMDK, a redo log is used to store changes. When analyzing connections, we therefore realize that any VMDK with multiple incoming links is a linked clone: we do not need a special 'linked clone' property, but simply need to understand the connections in our graph. In terms of auditing, we might wish to know how commonly this feature is used within a cluster. To answer a question like this within our graph, we simply look at each VMDK, find out which ones are connected to multiple VMs, and then trace the path upward until we locate the cluster containing the VMs.

```
1. START a = node:indexName(node_type="VMDK")
2. MATCH (b)-[*1]-(a) WHERE b.node_type = "VM"
3. WITH a, count(b) as vm_count WHERE vm_count > 1
4. MATCH (d) -[*4]-(a) WHERE d.node_type = "Cluster"
5. RETURN distinct d;
```

C. *Inventory comparison*

We next consider a more complicated example. Consider an IT administrator that must compare two virtualization hierarchies to ensure that they are equivalent. There a variety of reasons that this might be necessary:

1. An administrator has existing hierarchy in place and has just upgraded the management software to the latest version. The administrator would like to verify that the upgrade worked correctly and did not change the environment.
2. An administrator is migrating from a private cloud to a hybrid or public cloud, and would like to make sure that the migrated hierarchy is the same as the original hierarchy on the source.
3. An administrator has a standard configuration for provisioning virtualization infrastructure, and would like to know if a recently-deployed environment fits the template properly.

The general case of comparing two graphs is NP. However, we exploit the structure of our graph to transform it into a collection of trees and thereby improve the runtime by orders of magnitude. This requires making a few modifications to the default graph structure. First, we must create multiple copies of any shared leaf nodes (for example, if a VMDK is used by multiple VMs in a linked clone configuration). We must also remove any redundant links that we may have added in order to reduce the number of hops. For example, if a host is directly connected to a datastore, and then host can also be connected to the datastore by going to the VM, going to the VMDK, and then following the path to the datastore, we must remove the direct host-to-datastore connection. Finally, we must create different relationship types for compute, storage, and networking relationships. Once this transformation is complete, we no longer have redundant paths to any node, and therefore, we have three sets of trees. We perform a comparison on each, and then accumulate the result. The result is an algorithm that completes in $O(n^2)$.

In summary, we separate the comparison into multiple phases:

1. Retrieve the sub-graphs (compute, storage, or network trees) from the graph database.
2. Transform the sub-graphs into trees.
3. Label each node.
4. Compare the nodes at each level.

Steps 1 through 3 can be completed in O(n). Step 4 requires a pairwise comparison and is bounded at $O(n^2)$. The overall algorithm flow is as follows. After retrieving the sub-graphs in step 1 and transforming them into trees in step 2, we start at a given leaf node and traverse the path to its root. We label each node according to its constituent children. A sample label for a datacenter might be 2CL-4H-16VM, indicating that the datacenter has 2 clusters, that those clusters have a total of 4 hosts, and that those hosts have a total of 16 VMs. For the constituent clusters, the label might be 2H-8VM, indicating that each cluster has 2 hosts and 8 VMs. For the hosts, if one host has 2 VMs and the other has 6 VMs, then the labels would be 2VM and 6VM.

Once the labeling phase is complete, we perform step 4, in which we compare the labels for the source inventory and the destination inventory. We compare the nodes at each level of the source inventory with the nodes from the same level of the destination inventory. We determine if each node in the source inventory has a corresponding label in the destination inventory at the same level. If there is such a match, then we move to the next level up in the respective trees. Once we have reached the root of the tree without any unpaired nodes, we know that the trees are isomorphic. We then complete the comparison with a different resource tree (i.e., compute, storage or networking). This example illustrates the value of combining the graph structure with an understanding of the graph domain in order to make the problem tractable.

D. *Application migration from private tohybrid cloud*

For our final use case, consider again an administrator that is migrating from a private cloud to a hybrid cloud. It is important to move applications as a group between the private cloud and the hybrid cloud. For example, if an application consists of three tiers, and only two of the three tiers are moved to a hybrid cloud, then the network latency between these two tiers and the remaining tier may be high enough to cause performance issues.

Using a graph database can help with this issue. The first step is to identify groups of related VMs. This can be done either manually through explicit tagging by the administrator, or in an automated way by observing traffic and determining with VMs are communicating with each

other. Once these connections are determined, such VMs are explicitly connected using an "application" relationship. When an administrators wishes to move VMs from a private cloud to a hybrid cloud, the administrator can search the graph for all VMs that have an "application" relationship with other VMs. The search can also include grabbing relevant properties from related entities: for example, in addition to grabbing the relevant VMs, an administrator might want to capture the CPU type of the host on which the VMs are running, or capture the bandwidth characteristics of the storage elements on which the VMs are located. With this collection of information, relevant dependencies can be captured and then compared against the target cloud to make sure they can be satisfied.

V. Performance Results

This paper is an exploration of using a graph database for graph-like questions in cloud management. One tradeoff we face is how many relationships to create. For example, our default path from cluster to VMDK is three hops: cluster to a host to a VM to a VMDK. Instead we might choose to create a special cluster-to-VMDK relationship, trading off storage space for number of hops.

In this section, we do a simple examination of the performance of our graph. For these results, we use a graph consisting of approximately 60,000 nodes and 300,000 relationships. This setup represents a graph of a virtual infrastructure consisting of 32 clusters each with 32 hosts. Each host has 15 VMs, for a total of 1024 hosts, and 15,000 VMs, the current single-vSphere limit. Finally, the setup has 2048 datastores spread across the hosts and VMs. We consider the path from a cluster to the VMDK files of its VMs. We perform each query multiple times: the first query caches the nodes in memory, and the remaining queries leverage these cached nodes for faster computation.

The default path from a cluster to a VMDK is cluster to host to VM to VMDK, or three hops. For an out-of-the-box, untuned configuration of Neo4j on a 4x2.4 GHz CPU, 4 GB RAM system, a query that searches for a specific VMDK from a given cluster takes nearly 80s to complete, even when the entire graph is cached in memory. For this size graph, the memory footprint is negligible, approximately 11MB for the entire graph. However, the branching factor from cluster to host and from to VM is so high that the CPU becomes saturated. For shorter traversals, the level of branching is reduced, resulting in dramatically improved performance. For example, the latency of the single hop traversal from cluster to host is only 3ms when the graph is cached (50ms when it is not cached), and the CPU usage is negligible. When adding a second hop to the traversal, the latency increases nearly two orders of magnitude, to approximately 580ms, showing the impact of branching. The first query again caches the graph and takes nearly 600ms, and the CPU is saturated during this traversal. When we perform it a second time with the cached graph, the CPU drops from 100% to 20%, but the traversal latency only drops by a small amount from 580ms to 560ms. Going to three hops, as shown above, introduces a latency increase of another two orders of magnitude to 80s each time we run the query due to CPU saturation, even though the entire graph is cached in memory after the first query.

The second query we test is a two hop query: finding the number of VMs per datastore. This query is nearly 400s with an out of-the-box configuration, worse than the cluster query though it has fewer hops. In this case, the large number of datastores leads to a significant branching factor, as in the cluster-to-VMDK navigation.

To improve the performance of the previous queries, we must reduce the number of levels traversed, which in turn reduces the impact of high branching factors. One way reduce to the number of levels traversed is to introduce a 'skip link' (i.e., a link directly from the cluster to VMDK or datastore to VM). Adding such a link, we see a dramatic improvement in performance, with latency dropping to 3ms in each case. Clearly, choosing a poor representation for the graph can result in unacceptable latencies for reporting queries.

While Neo4j utilizes caching to reduce query latency, for large graphs (larger than in this example), the cache may still be insufficient. Moreover, as we have seen above, caching may not be as important as simply limiting the number of levels of branching and the degree of branching.

VI. Related Work

Graph databases have seen increasing popularity due in part to Facebook [6] and Google+ [7]. These sites emphasize the need for a scalable technique for storing connections. In these sites, the nodes are primarily humans. More recent work has focused on using the graph analogy in a wide variety of other contexts [14]. For example, source-control software might have humans and files as nodes, and may label the relationship as dependencies between files or use the relationship to indicate ownership of the file by the connected human. A movie recommendation site may place humans and movies as nodes, with the links as recommendations. Graph databases have also seen emerging use in bioinformatics, where they have been used to analyze the interactions between proteins [4]. Ultimately, graph database systems leverage the connection and the data as critical pieces of information, rather than just the data itself.

Previous work on simplifying virtualization management using graph databases [15] utilizes the metaphor of a social network and creates two types of links: membership and following. Nodes that serve as aggregation points rather than sources of messages are treated as groups, and the incoming links are 'member' links, while nodes that are capable of sending status messages in the infrastructure (like hosts and VMs) are connected to each other via 'follow' links. The idea is that if a host follows a VM, then when a VM encounters issues, the host is immediately apprised and can potentially invoke scripts to remedy the situation. With a small set of relationship types, the queries are quite simple, though the chain of connections may be long. In this work, we de-emphasize the social aspect of the connections, and instead focus on richer connection types (compute, storage, and network) that map better to our use

cases, namely auditing and reporting. For auditing and reporting, the flexibility and extensibility of the query engine is potentially more valuable than the raw speed, since such queries do not need to complete in real time. Moreover, in this work, we analyze the tradeoffs in determining a reasonable graph mapping and study how changes in that mapping can help facilitate more use cases in cloud management.

VII. Conclusions and Future Work

In this paper, we explore the use of graph databases to solve problems in cloud management. We specifically target reporting-style queries, which can often be couched in terms of graph traversals. Our goal is to see if questions that can be stated as graph traversals are best answered by graph databases. We provide some initial performance results, but we do not solely consider performance in our study. Rather, we focus on the flexibility, the extensibility, and the ease of use of the graph database paradigm for graph traversals.

Our results are mixed. The language of a graph database is very powerful and expressive, and many auditing questions can be formulated fairly intuitively using a graph database. However, doing the initial mapping of a cloud inventory to a graph database to enable this is nontrivial. A straightforward mapping introduces long chains between source and destination nodes, exposing known scalability problems of graph databases. Introducing additional links to shorten these chains is desirable, but must be done selectively to prevent an explosion of storage requirements. Utilizing an understanding of cloud inventories and doing subtle transformations on the data model (for example, converting a generic graph into sets of trees) can reduce the algorithmic complexity of queries, but requires a fairly deep understanding of cloud infrastructure.

Our most immediate future work involves a more in-depth performance characterization. We have not performed any significant tuning of our Neo4j implementation or our queries. Another area of future work is in profiling queries and trying to establish 'skip links' on-the-fly: direct connections between nodes like clusters and VMs rather than requiring two hops to connect them. To some extent, this is analogous to creating good indices in SQL databases or updating the statistics on a SQL query engine so that the execution plan is optimized. In order to create such skip lists, we intend to profile more auditing queries to useful connections to create. We will also expand our use cases to include more virtualization entity types. For example, we have considered hosts and clusters, but we have not examined resource pools. Another example is network switches: using the link-layer discovery protocol (LLDP [2]), we can identify the connection between a host and its physical switch, allowing us to create the complete path from a virtual device of a VM all the way to its host. This sort of information is invaluable as network virtualization [21] becomes more prevalent, creating an even looser connection between the virtual and physical in a cloud. Finally, as with other types of so-called multi-domain graphs [14] like Wikipedia [23], it is interesting to speculate on redefining the node types even more. For example, in cloud environments, events like network outages can form chains that indicate causality, for example, host outages or VM outages. If we put events and virtualization entities on the same graph, we move beyond a simple static connection between hosts and VMs, allowing us to potentially help with *post mortem* analysis in case of failures by connecting events and the originating entities.

References

[1] C. Clark, et al, "Live Migration of Virtual Machines", Proceedings of Network Systems Design and Implementation, Boston, MA, May 2005, pp. 273-286.

[2] Cisco Link Layer Discovery Protocol. http://www.cisco.com/en/US/docs/wireless/asr_901/Configuration/Guide/lldp.html

[3] Eucalyptus. Eucalyptus Cloud Software. http://www.eucalyptus.com/eucalyptus-cloud

[4] C. Have and L. Jensen, "Are graph databases ready for bioinformatics", Bioinformatics, Vol. 29, No. 24. (15 December 2013), pp. 3107-3108, doi:10.1093/bioinformatics/btt549.

[5] C. Hogan. "Linked Clones, Part 1 – Fast Provisioning in vCloud Director 1.5," http://blogs.vmware.com/vsphere/2011/11/linked-clones-part-1-fast-provisioning-in-vcloud-director-15.html

[6] A. Lakshman and P. Malik. "Cassandra: a decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44. April 2010, pp. 35-40.

[7] G. Malewicz, et al. "Pregel, a system for large-scale graph processing," Proceedings of the 28th ACM Symposium on Principles of Distributed Computing. Calgary, Alberta, Canada, August 2009, p. 6.

[8] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast Transparent Migration for Virtual Machines," Proceedings of USENIX 2005, Anaheim, CA, April 2005, pp. 391-394.

[9] Neo4j. http://www.neo4j.org

[10] OpenStack. OpenStack Cloud Software. http://www.openstack.org

[11] E. Plugge, T. Hawkins, and P. Membrey, "The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing," Apress, Berkeley, CA. 2010.

[12] J. Pokorny, "NoSQL databases: a step to database scalability in a web environment," Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, Ho Chi Minh City, Vietnam, December 2011, pp. 278-283.

[13] Py2neo. http://py2neo.org.

[14] M. Rodriguez, "Supporting the Emerging Graph Landscape," http://markorodriguez.com/

[15] V. Soundararajan and L. Spracklen, "Simplifying Virtualization Management with Graph Databases," VMware Technical Journal, vol. 2. June 2013, pp. 46-53

[16] J. Webber, et al. Graph Databases. Sebastopol: O'Reilly Media; 2013.

[17] VMware Distributed Resource Scheduling (DRS). http://www.vmware.com/products/datacenter-virtualization/vsphere/drs-dpm.html

[18] VMware vCenter Operations Manager. http://www.vmware.com/products/vcenter-operations-manager/

[19] VMware vCloud Director. http://www.vmware.com/products/vcloud-director/overview.html

[20] VMware vSphere API Reference Documentation. https://www.vmware.com/support/pubs/sdk_pubs.html

[21] VMware NSX. http://www.vmware.com/products/nsx/

[22] VMware vSphere. http://www.vmware.com/products/datacenter-virtualization/vsphere/overview.html.

[23] Wikipedia. http://en.wikipedia.org.