Gavel: Software-Defined Network Control with Graph Databases

Osamah L. Barakat, David Koll, Xiaoming Fu

University of Goettingen, Germany
{obarakat,dkoll,xfu}@cs.uni-goettingen.de

Abstract—Writing network policies for Software Defined Networks (SDNs) is not a straightforward task. Abstractions play a major role in easing this task, but they are typically provided for a particular use case. As a consequence, emerging applications will require the development and implementation of new abstractions, and combining the policies of different abstractions becomes difficult. To offer a simpler framework, plain data representations of the network and its control infrastructure have been proposed recently to offer programmable ad-hoc abstractions to administrators. However, these representations still induce unnecessary complexity and are additionally inferior to use function-specific abstractions from a performance point-of-view. In this work, we propose Gavel, an SDN controller based on a graph database. By exploiting the native graph support of the database engine, Gavel significantly eases application and policy writing. Additionally, we show by experimental evaluation of several typical applications that Gavel can significantly increase the performance when compared with previous works.

Index Terms—Software-defined Networks; Controller; Database; Graph

I. INTRODUCTION

Software-defined Networking (SDN) promises to deliver an easy-to-manage network [1]. One key characteristic of SDN that helps to abide in that promise is that it separates the control plane from the data plane. This way, control plane applications and protocols can be developed regardless of hardware configurations and consequently without considering any specifications of the data plane devices.

Such control plane applications are typically designed to perform one particular task in the network (e.g., routing), and network administrators have usually implemented these applications at a low level of abstraction, which has hindered modular programming for SDNs [2]. To tackle this problem, higher level north-bound interface abstractions have been proposed recently. These abstractions usually propose an easy-to-write high level language to express application policies, combine these policies into a single network policy and then translate this policy to a lower level protocol (i.e., OpenFlow).


As the variety of proposed solutions shows however, each abstraction is usually targeted at a certain type of network policies. As a result, one abstraction is often not enough to implement a complete network policy, especially as network policy requirements and consequently different administrator needs continuously evolve. Thus, network administrators have to combine two or more abstractions to formulate and implement their own policies in many cases. The complexity of combining abstraction policies increases with the number of employed abstractions and can be a tedious task [9].

To address the need for a simpler way of expressing and combining network policies, plain data representations of the network and its control infrastructure have been proposed recently [9]. This suggestion simplifies different network north-bound interfaces by discarding any application-specific structure that might be outgrown by new demands and instead considers these interfaces as plain data representations. The benefit of these representations is a much simpler model of the network, on which policies can be composed regardless of any application-specific structures.

The state-of-the-art approach to plain data representation is to model the complete network as a relational database [9]. On these tables, ad-hoc views can be constructed and then queried by application developers. This model yields three main advantages. First, the entire network can be orchestrated using a well-known query language (SQL). Second, relational databases offer consistency guarantees among different network views. Finally, different applications (e.g., virtual networking) can be realized by exploiting database views.

However, building network control around a relational database comes with a price. As data in relational databases is distributed across different tables, and each data item is related to other data items in further tables, collecting aggregated views from the database is complex. Although inserting very specific information (e.g., a firewall entry) is fast, retrieving information that needs to be collected from many tables is costly (e.g., retrieving routes). At the same time, implementing a new application requires first a very good understanding of the database, and second manipulating this database scheme.

As a consequence even simple applications need to interact with a large number of database tables. For instance, to calculate a shortest path between two nodes in the network, one of several steps requires—for each switch on the path—
to query a table storing that switch’s port connectivity. As a result, processing application requests is slow. Additionally, writing network applications can still be overly complex. In particular, each application needs to create its own tables and link them properly to the existing database. This can be a tedious task for the application developer, especially as the number of applications running in the network increases.

In this paper we present Gavel, a network controller that overcomes these shortcomings by proposing a more natural way to implement a plain data representation that can be easily queried by network applications. Concretely, our contributions are as follows:

- We propose a new concept that respects the graph nature of computer networks and consequently employs a graph database instead of a relational database. The result is a much simpler network model that yields substantial reductions in programming complexity as well as significant gains in terms of performance over state-of-the-art solutions.

- We implement exemplary network applications of different types to show how administrators can develop previously complex applications easily. Finally, by deploying our prototype in an emulated testbed, we show that our approach indeed yields significant performance gains over state-of-the-art plain data representations.

The remainder of this paper is structured as follows. We first advocate the use of graph databases for plain data representations in Software-defined Networks in Section II. Then, we propose and implement both Gavel and exemplary applications in Sections III and IV, before evaluating the system in Section V. We discuss the benefits of our system and future work in Section VI before summarizing related work in Section VII. Finally, we present concluding remarks Section VIII.

II. FINDING A PLAIN DATA REPRESENTATION: THE CASE FOR GRAPH DATABASES

Our goal is to provide a plain data representation of the network, which then can be queried by networking applications of arbitrary type to update or retrieve information (such as, e.g., firewall rules or routing tables). Thus, we need to provide two main operations—(i) manipulating the data saved in the representation (i.e., inserting, updating, and deleting) and (ii) retrieving any information required by the applications.

A. Drawbacks of Relational Databases

State-of-the-art solutions currently employ a relational database, where these operations are provided by creating tables that can later be queried, aggregated, updated, and so on. However, the more complex applications become, and the more applications we deploy in our network, the more tables needed to be dealt with, and the more complex dependencies among tables become, which increases the programming effort needed from the network administrator.

For instance, as depicted in Figure 1, a representation based on a relational database (RDBMS) will need to keep track of different network entities (such as hosts, switches, or links) in different tables. A routing application that tries to find the shortest path between two hosts will need to query and combine information from all these different tables, and then insert the information in a new table that contains routing information. If another application (e.g., a firewall) is run in parallel, this application may have to modify the database schema (as, e.g., firewall rules can not be represented by the existing standard tables) or even existing applications (e.g., the routing application has to be modified to also respect firewall rules when querying tables).

At the same time, the increasing complexity also results in reduced performance. The more applications are deployed in the network, the more (and likely more complex) tables need to be queried to retrieve an answer. Similarly, when updating network information, this information has to be made consistent across all tables. As network control is time-critical, this processing overhead can become prohibitive.

B. Advantages of Graph Databases

To overcome this issue, we propose to employ a graph database as the main component of the control plane architecture (GDBMS in Figure 1). Graph databases, such as Neo4j [10] have so far mainly been considered as an alternative for handling the exponential growth and high complexity of social networks, as they offer significant advantages for modeling networks over traditional relational databases [11].

In this paper, we advocate to exploit these advantages when modeling Software-defined networks. Since networks are best represented by a graph in nature, graph databases are a much more natural fit for them. In general, we can model each device in the network as a node in the graph database, and then easily manipulate its attributes. These advantages are best shown by illustrating examples.
• Inserting a new switch into a relational database model requires to dissemble the request into basic data types and then to map each of the data types to an appropriate table. In particular, the information of a new switch, all its ports, and all the neighbors it is connected to has to be stored in the corresponding tables. Additionally, we need to modify those table entries that contain information on the neighbors of the new switch, their ports, and so on. In a graph database on the other hand, inserting a switch only requires to add a node and one edge to each of its neighbors to the graph structure.

• Similarly, requesting data from a relational database (e.g., finding a route between two hosts) will require first to collect the relevant nodes, their attributes and connectivity information from different tables. In a graph database on the other hand, such operations are natively supported. That is, we can simply query the graph structure for the shortest path between the two nodes.

• Finally, the orchestration of applications, in particular keeping network data consistent, is easier with graph databases. For instance, when implementing a firewall, we simply need to modify node labels or attributes in the graph to restrict access for certain hosts or interfaces, instead of manipulating a routing application and the database scheme.

While these are simple examples, we believe they generalize well to more complex scenarios. In fact, complexity in relational databases increases with increasing network size or more complex topologies, as table simplicity increases with that. Graph databases on the other hand have proven to scale extremely well in complex multi-million node networks [12].

III. GAVEL

In this section, we propose a new way of network control that makes use of the observations in the previous section. Gavel is an SDN controller which as its key component employs a graph database as the plain data representation that is used to control and manipulate the network state. To offer easy application deployment, Gavel uses a query language supported by the graph database.

In the following, we describe the way that Gavel handles and processes SDN networks. This includes the graph database employed, the unified language used to write network applications, and how it models the SDN network.

A. Network Model

One major advantage of employing a graph model is that we can represent the network in a much simpler way. As shown in Figure 2a, to model the entire network—including all nodes, links, and policies present in the network—Gavel constructs the network graph based on only two entities.

• Nodes represent either forwarding devices or hosts. They keep information that suits the object they represent (e.g., IP addresses for hosts and OpenFlow Data Path ID (dpid) for switches). We show an exemplary representation of two switch nodes in Figure 2b (left side).

• Edges represent either a physical connection between two nodes (i.e. a host connected to a switch) or a virtual connection (i.e. installed path between two hosts). In the first case, they keep track of the connected ports from each device, while in the second case they keep a list of all switches that constitute the installed path in order with their respective port numbers. Edges are undirected, i.e., they represent bidirectional (full-duplex) connections. We show an exemplary representation of a physical connection between two switches in Figure 2b (right side).

Compared to complex table structures, this simplicity in modeling the network helps to quickly retrieve any informa-
tion required by an application. Moreover, whenever needed, developers can extend node or edge objects to add additional information required by an application (e.g., the cost of an edge). Note that while this changes the database structure, it does not require to modify multiple tables to achieve consistency.

B. Selecting a Graph Database for Gavel

To realize our network model, we use the Neo4j graph database as the core component of Gavel. Neo4j has three major advantages over other graph databases [13], as it is distributed as Open Source software, supports native graph functions and implements Cypher as a fast query language that is similar to SQL in its function [14].

1) Native Functions: Neo4j natively offers important graph functions that can significantly enhance the controller performance. For instance, one of the most called routines in networking is that of finding a path between two devices. Relational databases cannot perform this task natively and instead need to exploit third-party software. For instance, PostgreSQL can use pgRouting [15]. However, this would require redesigning the database to be compatible with the third party’s library and retrieving the path still requires significant processing of tables. In contrast, Neo4j natively supports the calculation of a shortest path as it processes a graph structure directly.

2) Query Language: Besides processing native graph functions, Neo4j in Cypher offers its own declarative query language. Cypher is simple, easy to read and has a flat learning curve. In Figure 3c we show exemplary snippets of how two switches and a link between these switches can be created. Note that, after creation of the nodes, Neo4j can natively traverse paths in the graph via the connected_to property. In spite of its simplicity, Cypher is strong enough to manipulate a large-scale graph data store in an efficient way [16]. Another advantage of utilizing Cypher is its flexibility towards combination with other programming languages (e.g., Python or Java; we will later present an example for applications which employ both Cypher and Python).

IV. APPLICATIONS FOR GAVEL

As a proof of concept, we next implement a variety of applications on top of Gavel. In this section, we show that Gavel can i) indeed profit from native graph functions (by implementing a functional routing application), ii) handle different types of applications (by implementing a stateful access control firewall as a finite state machine application), and iii) offer functionality to combine Cypher with different programming languages (by implementing a loadbalancer in combination with Python).

Additionally, we show that these applications can be run in isolation of each other without the need to be modified (e.g., routing does not need to be modified to work with the firewall), and the changes needed in the database scheme are minimal.

A. Routing application

Once a forwarding device in the data plane receives a packet with an unknown destination, it asks for an instruction of where to forward the packet from the network controller. As discussed before, deciding the path for a packet can be efficiently achieved by exploiting Gavel’s native source shortest path algorithms.

In particular, Gavel will query the graph model and traverse edges via the connected_to attribute. The returned route contains information about all switches and their connected ports on the path from the respective switch towards the destination, which also means that it is not necessary to query the database again when composing the relevant OpenFlow messages.

Figure 3a shows an example of this procedure. Here, to calculate the route between the two hosts h1 and h2, Gavel starts first to find and locate the two hosts, querying for their IP addresses. Afterwards, using the find-route routine built in to Neo4j, it calculates and retrieves the shortest path between the two hosts and as a result also collects all relevant path

```python
MATCH (h1:Host{ip:srcip}), (h2:Host{ip:dstip})
MATCH p=shortestPath((h1)-[:Connected_to*]->(h2))
WITH h1, h2, ... as nodes(p), n as dpid,
    ports as r in rels(p)
RETURN pa.switches, pa.ports;
```

(a) Code to implement a routing application in Gavel

```python
MATCH (h1:Host{ip:srcip})-[r:Connected_to]->(s2:Switch)
REMOVE h1:Host SET h1:blockedHost
RETURN DISTINCT s2.dpid;
```

(b) Pseudo code for processing a routing request in Gavel [9]

```python
MATCH (h1:Host{ip:srcip})-[r:Connected_to]-(s2:Switch)
REMOVE h1:Host SET h1:blockedHost
RETURN DISTINCT s2.dpid;
```

(c) Code to implement host blocking in a Gavel firewall

```python
def installrouteWithLib(self, srcIP):
    self.initconnectWithDB()
    dstIP = self.getnextavailableserver()
    return getroute(self.session, srcIP, dstIP)
```

(d) Code to implement load-balancer functionality in Gavel (Python)
information (switches with ports that connect them ordered from source to destination). This information is stored in the database as a relationship between the two end hosts with a path_to label (cf. Figure 1).

This yields two advantages. First, while a relational database controller would need to re-compute the return route, Gavel is able to install reverse paths by simply reversing the returned switch list. This is expected to be happen especially in any TCP connection which starts by a three-way handshake. Second, further optimization could be applied so that all forwarding rules for hosts that are connected to the same switch, could be summarized into a single forward message using route summarization techniques. This yields in lower number of OpenFlow rules in each switch and consequently saves TCAM space. Theoretically a relational DB controller could implement this functionality by intensively finding all routes that are installed and share the same source and destination, Gavel can do so with much less overhead.

In general, when compared with a routing application of a state-of-the-art relational database controller, Gavel introduces much less overhead. Figure 3b shows the pseudo-code for calculating and storing a route in a relational database. Before actually calculating the path, the request needs to be inserted into several tables to trigger relevant routines, which will return the relevant information (note that this step requires processing several tables itself). After inserting the calculated route into yet another different table, the information required to install OpenFlow rules along the path needs to be queried for. This involves, for each switch on the path to query the database again.

B. Access Control Firewall

Next, to test Gavel’s ability to handle a finite state machine application, we implemented a simple stateful firewall application similar to [9].

Our firewall generally allows for access control in the network by managing the visibility of resources to applications. For instance, as described above, the routing application finds shortest paths between two host entities in the graph. Disabling a host can thus be achieved by simply changing the label of the node from host to blockedHost. In general (the same concept applies, for instance, to edges between switches or paths between hosts, too), by changing the type or attributes of a node or edge in the graph, we can alter its visibility to applications.

Importantly, this concept yields isolation of the firewall from other applications. While the firewall needs to alter the database scheme and the routing application itself in [9] (before finding a route, the routing application needs to check for blocked resources in a distinct table), in Gavel, resources are hidden by the firewall and simply not visible to the routing application. For instance, once the firewall blocks an edge in the graph by changing its type, the routing application will not find that edge anymore and will instead route a request via a different edge, if available, or return a no-route error if not.

We exemplary show the Cypher code that implements the functionality for blocking a host in Figure 3c. The firewall first needs to locate the host to be blocked by its IP address. Afterwords, it changes the label from “Host” to “blocked-Host”. Moreover, to enforce the policy on any previously installed path, the application continues to locate all switches connected directly to that host and then installs the respective drop OpenFlow rules (drop packets targeted at that host) in these switches. By that, no device will be able to reach the blocked host until a reverse (unblock) routine is called. Slightly more challenging is a scenario, in which we want to block a path that is not installed yet. Here, the firewall will first compute the relevant path, and then change its type to blocked.

Applying these concepts using relational DB based controllers would require to add a column to the hosts table to indicate each host status (i.e., blocked or not), and thus an alteration of the database schema.

C. Load-Balancer

Finally, we also implemented a simple load-balancer application to demonstrate the ability of combining Cypher with other languages (in this case Python), as shown in Figure 3d. Here, the load-balancer keeps track of and sequentially cycles through all hosts over which the load should be balanced (e.g., all web servers servicing the same content). The route from the requesting source to the target host is determined with the help of the underlying graph database. Again, implementing the load-balancer does not require any change in the database schema or any other application.

D. Summary

In summary, we have provided three examples of how different types of applications can be implemented in Gavel based on the Cypher query language. The key advantages of Gavel over state-of-the-art relational database solutions are that i) applications are simpler to implement, ii) do not need to alter concurrently running applications, and iii) only need to update the graph itself and not a multitude of tables. In the next section, we evaluate Gavel and these applications with regards to their performance.

V. Evaluation

In Section II, we have argued that employing a graph database instead of a relational database will have two major benefits. After showing that applications can be developed with ease in Gavel, we now focus on the second promise: improved performance. In the following, we show that Gavel is indeed able to outperform the state-of-the-art for plain data representations with relational databases.

A. Evaluation Setup

To evaluate its performance, we have implemented Gavel and deployed a prototype together with our exemplary applications on an emulated Mininet [17] testbed, where we evaluated our system with regards to its performance on
different topologies as shown in Table I. In particular, we checked for the performance of Gavel in the context of both data-center topologies (FatTree [18] with k=16 through FatTree with k=64) as well as in ISP topologies (Geant and Deutsche Telekom topologies [19]).

Table I: Topologies used to evaluate Gavel

<table>
<thead>
<tr>
<th>Topology</th>
<th>Switches</th>
<th>Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>FatTree 16</td>
<td>320</td>
<td>3072</td>
</tr>
<tr>
<td>FatTree 32</td>
<td>1280</td>
<td>24576</td>
</tr>
<tr>
<td>FatTree 64</td>
<td>5120</td>
<td>196608</td>
</tr>
<tr>
<td>Geant2012</td>
<td>40</td>
<td>61</td>
</tr>
<tr>
<td>Deutsche Telekom</td>
<td>39</td>
<td>101</td>
</tr>
</tbody>
</table>

B. Performance and Scalability

The main metric for evaluating Gavel is the latency that applications experience.

1) Routing: As discussed before, the routing application is one of the core applications of every network. It is thus important to investigate the performance of our solution and its ability to operate on different scales. Concretely, our solution should work well in networks with smaller topologies (e.g., provider networks), but also with increasing network size (e.g., datacenter networks).

In a database based network, the process of routing between a source and a destination can be decomposed into three steps:

1) **Path Computation (PC):** This step involves calculating and getting the shortest path between the two nodes.
2) **Path Writing (PW):** After the path has been computed, it needs to be written to the database.
3) **Port Extraction (PE):** Finally, to build the respective OpenFlow messages for all switches on the path, all port information for these switches needs to be extracted from the database.

In Figures 4a and 4b we show the routing performance of Gavel and compare it with that of Ravel [9], the state-of-the-art in relational database based controllers. Here, we list the measured delay for each component of the routing process as well as the total delay introduced.

Our key observations are:

- Due to its native fit for networks, Gavel’s graph database engine can compute the path (PC) almost instantly, and almost two orders of magnitude faster than Ravel.
- The cost of all three operations (PC, PW, PE) remains constant within increasing network size in Gavel, while Ravel significantly suffers from the increased network complexity. For instance, when comparing different scales of FatTree networks, Ravel performs approximately two orders of magnitude worse in a 64-ary FatTree than in a 16-ary FatTree. For Gavel, we do not see any difference in performance.
- We see that path writing is actually faster in relational databases for small topologies. We believe that this is caused by highly optimized operations available in these databases (e.g., PostegreSQL), while graph databases have not yet matured to that point.
- One key benefit of using a graph database is that it does not need to perform any port extraction (PE) at all, while this remains a costly step in relational database controllers like Ravel. The reason for this advantage is that the very fast PC step already yields the relevant information in Gavel.
- In a FatTree topology with k=16, Ravel is slightly faster for the complete routing operation. Note that this is not caused by the size of the network (as counter-examples see the performance for the much smaller ISP topologies in Figure 4b), but rather by the tree property of the FatTree topology, which results in comparatively short paths between most hosts, and thus in lesser database queries required in Ravel.
- In all other scenarios, Gavel significantly outperforms Ravel, in some cases by one order of magnitude or more.

2) Firewall: Next, we evaluate our firewall application and show the results for FatTree networks in comparison with Ravel in Figure 5. Here, comparison is not as straightforward as in the routing case. As described in Section IV, in Gavel blocking a host means simply changing its type in the graph structure. Then, any subsequent routing requests will not be able to route to that host. Different from this, in Ravel,
blocking a host means inserting a rule into the firewall table. In our experiments, this operation (comparable with path writing in the routing application) is fast in Ravel for smaller network sizes. In the firewall case however, after this rule has been inserted, in any subsequent routing request, Ravel needs to query this table again to verify the eligibility of the flow to be installed (which additionally requires the routing logic to be changed, as described in Section IV). Hence, in our comparison, we consider both of these operations and show the resulting sum of delay for blocking a host (BH), and the reverse operation of unblocking a host (UbH).

Our key observations are as follows:

- For k=16, Gavel is a bit slower than Ravel, due to the highly optimized write operations of PostegreSQL and short routing paths in this topology. However, both systems can perform firewall operations fast with a delay of ≤ 5 ms.

- However, as network size (and/or complexity) increases, the delay introduced by Ravel can become prohibitive (∼400 ms for 64-ary FatTree network), because every route request needs to be checked against the respective firewall table. Note that this adds significant overhead and results in a performance loss of more than one order of magnitude when compared to the isolated routing cost of Ravel for k=64. At the same time, the cost in Gavel remains constant, and these results are consistent with our routing results.

VI. DISCUSSION AND FUTURE WORK

In Section II we argued that a graph database based controller can yield two major benefits for plain data representations in the network over relational database based controllers—i) it can allow for easier application development and ii) it can offer improved performance. In this section, we briefly summarize how Gavel has achieved these goals.

A. Easier Application Development

In Section IV we have shown that Gavel supports different types of applications to be easily developed in few lines of code in the easy-to-learn Cypher programming language on top of its network model. We have demonstrated this ability exemplary with routing, access control (firewall) and load-balancing applications. We have shown that the programming complexity is significantly reduced over state-of-the-art solutions using relational databases as fewer and less complex queries are needed (especially with increasing application complexity) and that the orchestration complexity is reduced, because applications do not need to alter the logical flow of other applications, and all applications will automatically have access to the same network state.

B. Improved Performance

In Section V we have shown that Gavel outperforms the state-of-the-art plain data representation Ravel. In particular, across both routing and firewall applications, it is able to exploit native graph functions of the highly scalable Neo4j graph engine to offer constant and fast performance regardless of the network size, while relational databases suffer from increasing database scheme complexity. We expect this problem to exponentially grow with the number of applications that are running concurrently in a relational database based system.

C. Future Work

While Gavel is fully functional, it currently does not offer mechanisms to resolve conflicts between different applications. For instance, Gavel cannot detect currently whether a change to the graph model issued by a firewall would override a previously installed policy that has a higher priority. To resolve such conflicts, Gavel needs to facilitate orchestration. This orchestration could take place either before making decisions requested by applications or after it. Beforehand coordination focuses on resolving intents, which express applications or polices rules, then generating one policy that satisfies most of the applications objectives. In afterward coordination, the focus is on checking control consistency using priority or voting mechanisms. In general, orchestration can be realized either by using existing graph-based approaches (e.g., PGA [7]) or by implementing a specific orchestration layer for Gavel. We leave the design of an orchestration layer covering these cases for future work.

VII. RELATED WORK

Related efforts prior Gavel can be categorized in three different classes of proposals—north bound interface abstractions, use of graph databases in networks, and the use of databases in existing SDN Controllers.

A. Abstractions

Many abstractions have been proposed to ease network programming for specific types of applications. Pyretic [2], Frenetic [3], and NetCore [4] proposed functional abstractions to construct SDN control applications. FatTire[5] and

![Figure 5: Comparison of routing delay in combination with firewall routines for blocking hosts (BH) and unblocking hosts (UbH) in both Ravel and Gavel in k-ary FatTree networks with k=16,32,64](image-url)
Merlin[6] allowed for extended support for fault-tolerance and resource provisioning type policies, respectively. Finally, PGA[7] and Kinetic[8] proposed additional abstractions for graph handling and finite state machine support on top of Pyretic. As a plain data representation, Gavel on the other hand is able to support different types of policies.

In previous plain data representations, Ravel [9] is currently the state-of-the-art. As discussed throughout the paper, while Ravel relies on relational databases for representing the network, Gavel uses a graph database engine and thereby offers improved application programmability and controller performance.

B. Use of Graph Databases in Networks

Gavel is not the first system to exploit graph databases for networking applications. Researchers have indeed recognized the opportunities of graph databases in using graph database libraries before [20], [21], [22]. Gavel on the other hand builds the entire network model on a graph database and exploits the graph database engine to facilitate network control.

C. Databases in SDN Controllers

Finally, graph databases have been used in previous SDN network controllers (e.g., ONIX[23] and ONOS[24]). However, in these systems graph databases usually play a passive role and help with state distribution, distributed processing, concurrency, replication control or network state storage. Gavel is the first controller to employ an active graph database engine for the purpose of network control.

VIII. CONCLUSION

In this paper we have proposed Gavel as a new way of providing a plain data representation for a Software-defined Network. To provide this representation, Gavel is the first SDN controller that exploits the benefits of graph databases and employs these graph databases as the control engine for managing the network. After arguing why graph databases can provide a better representation of the network than relational databases and introducing Gavel, we have shown that our approach offers substantially easier application development with less complications than in previous works. In our evaluation, we have shown that Gavel further significantly outperforms the state-of-the-art of plain data representations, especially with increasing network size.

REFERENCES