Ravel: A Database-Defined Network

Anduo Wang[†][*] Xueyuan Mei[¶] Jason Croft[¶] Matthew Caesar[¶] Brighten Godfrey[¶]
Temple University[†] *University of Illinois at Urbana-Champaign* [¶]

ABSTRACT

SDN's logically centralized control provides an insertion point for programming the network. While it is generally agreed that higher-level abstractions are needed to make that programming easy, there is little consensus on what are the "right" abstractions. Indeed, as SDN moves beyond its initial specialized deployments to broader use cases, it is likely that network control applications will require diverse abstractions that evolve over time.

To this end, we champion a perspective that SDN control fundamentally revolves around data representation. We discard any application-specific structure that might be outgrown by new demands. Instead, we adopt a plain data representation of the entire network — network topology, forwarding, and control applications — and seek a universal data language that allows application programmers to transform the primitive representation into any high-level representations presented to applications or network operators. Driven by this insight, we present a system, Ravel, that implements an entire SDN network control infrastructure within a standard SQL database. In Ravel, network abstractions take the form of user-defined *SQL views* expressed by SQL queries that can be added on the fly. A key challenge in realizing this approach is to *orchestrate* multiple simultaneous abstractions that collectively affect the same underlying data. To achieve this, Ravel enhances the database with novel data integration mechanisms that merge the multiple views into a coherent forwarding behavior. Moreover, Ravel is exposed to applications through the one simple, familiar and highly interoperable SQL interface. While this is an ambitious long-term goal, our prototype built on the PostgreSQL database exhibits promising performance even for large scale networks.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Centralized networks; C.2.3 [**Network Operations**]: Network management

[*]The work was mostly done while at University of Illinois at Urbana-Champaign.

SOSR'16, March 14–15, 2016, Santa Clara, CA.
© 2016 ACM. ISBN 978-1-4503-4211-7/16/03 ...$15.00.
DOI: http://dx.doi.org/10.1145/2890955.2890970.

Keywords

Software-Defined Networks; SQL Database; Views; Programming Abstraction

1. INTRODUCTION

Software-defined networks employ APIs to control switch data planes via a logically centralized controller. This provides a platform so that it is possible to write software that can control network-wide behavior. To make that programming *easy*, it is generally agreed that higher-level abstractions are needed, but there is little consensus on what are the "right" abstractions. Frenetic [8], Pyretic [28], and NetCore [22] introduce functional constructs that enable modular creation of SDN control. Flowlog [23] extends this functional abstraction to the data-plane by adding SQL-like rule constructs. FatTire [29] and Merlin [30] add additional language-level support for fault-tolerance and resource provisioning. On the other hand, to address stateful middleboxes and service chains, Kinetic [14] and PGA [27] raise the abstraction level by stacking a new layer of abstractions in the form of state-automata and graph expressions.

Indeed, as the network evolves over time, the need for abstractions will likely outgrow the abstractions of the present, thus requiring continual "upgrade" that extends or raises the level of existing abstraction. Each "upgrade" incurs the tremendous effort of careful design of a new abstraction and the engineering of the supporting runtime. With the deployment of drastically different abstractions, the network will be jointly driven by multiple controls created with disparate abstractions. Orchestration of these controls across many abstractions is thus needed. However, current approaches only offer a fragmented solution: higher-level coordination often depends on and is restricted to the use of certain abstractions [28, 27, 14], while abstraction-agnostic support depends on common structures (OpenFlow rules, network state variables) that are usually low-level [21, 32, 12].

To this end, we champion a perspective that SDN control fundamentally revolves around data representation. We discard any application-specific structure that might be outgrown by new demands. Instead, we adopt a plain data representation of the entire network — network topology, forwarding, and control applications — and seek a universal data language that allows application programmers to transform the primitive representation into any high-level representations presented to applications or network operators. Driven by this insight, we present a system, Ravel, that implements an entire SDN network control infrastructure within a standard SQL database. We take the entire SDN control system under the hood of a standard SQL database, and rely on SQL for data manipulation, the database runtime for data mediation, and propose a novel protocol that refines the database runtime to enforce only orchestrated execution. As a realization of this approach, Ravel offers the following attractive advantages:

Ad-hoc programmable abstractions via database views. The database *view* construct enables new abstractions to be constructed ad-hoc and enables them to build on each other. For example, from the "base tables" defining the network topology and OpenFlow forwarding rules, one can construct a view representing a virtual network that spans a subset of the topology; and one can further derive a load balancer's view of host availability within the virtual network. These SQL views form the structure of the abstractions; furthermore, integrity constraints over the views express high-level policy invariants and objectives. These views and constraints can be expressed via SQL statements and can even be constructed ad-hoc, i.e., dynamically in the running controller.

Orchestration across abstractions via view mechanisms. Once we have a mechanism to construct ad-hoc abstractions, we need to coordinate across these multiple views of a single network. View maintenance [9] and update [2, 4, 7, 5, 13] mechanisms are the "runtime" for view abstractions, constructed through normal SQL operations (queries and updates). First, *view maintenance* continuously refreshes the view abstractions of a dynamic network. Second, to translate updates in a derived view back down to a lower-level view, Ravel allows users to define a view update policy that governs how updates on the higher-level abstraction are realized on the underlying data-plane via *triggers*, which can incorporate custom heuristics at runtime to optimize applications.

Orchestration across applications via a data mediation protocol. In Ravel, an orchestration protocol mediates multiple applications whose database modifications affect each other. The protocol assumes a simple conflict resolution strategy — an ordering of view constraints where lower-ranked constraints yield to the higher-ranked. Given a view update request as input, the protocol produces as output an orchestrated update set that respects all applications constraints (subject to conflict resolution). The orchestrated set may append to the request additional updates for completion (e.g., by invoking a routing application when an access control application attempts to remove an unsafe path) or reject the update if a cohesive update is not possible.

Network control via SQL. The entire Ravel system above is exposed to application programmers and network engineers via standard SQL interfaces and database view, constraint, and trigger mechanisms. We believe this is likely to be valuable both technically and as a way to encourage more rapid uptake of SDN. SQL databases have proved over decades to be an effective platform for high-level manipulation of data, and are broadly familiar (compared to domain-specific languages, for example). Moreover, network architects today need to combine heterogeneous data sources — network forwarding rules, data flow and QoS metrics, host intrusion alerts, and so on — to produce a cohesive view of the network and investigate problems; this will be eased by the interoperability of SQL databases.

We note that certain past SDN controllers have employed databases as specific limited modules, including state distribution, distributed processing, concurrency, and replication control. For example, Onix [15] delegates to distributed databases for concurrency and replication of low-level network state in a common predefined data schema. The database silently accepts and executes transactions submitted by external control applications. Pre-SDN era declarative networking [17, 16, 19], on the other hand, uses a distributed query engine for fast processing of customized routing protocols, where the database executes routing queries submitted by end-hosts. In contrast, Ravel makes the database an active participant that uses views to incorporate multiple high-level and low-level abstractions of the network which are orchestrated online. In other words, in Ravel, the database *is* the controller.

While this is an ambitious long-term goal, our Ravel prototype built on the PostgreSQL database [26] exhibits promising performance even for large scale networks. On various fat-tree (up to 5,120 switches / 196,608 links) and ISP (up to 5,939 nodes / 16,520 links) topologies, we microbenchmark Ravel delay, uncovering the source of database overhead, showing that the most relevant database operation (triggers) scales well to large networks. Orchestration of various applications scales to large network and policy size. Ravel also integrates a classic view optimization algorithm that accelerates view access by one to two orders of magnitude with a small maintenance overhead.

2. OVERVIEW

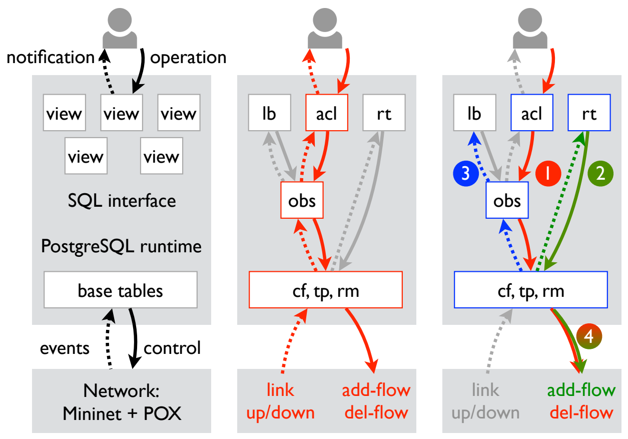

Figure 1: Ravel overview

Figure 1 shows the main components of Ravel: the users, the Ravel runtime, and the network. The network (bottom) is the services and resources being controlled. The users include network operators and a Ravel administrator that interact with the network via the view interface (by observing database notification and inputting view updates). The Ravel administrator can modify the behavior of Ravel, such as enriching the control abstractions by adding new views or changing the orchestration behavior (database execution model) by adding new protocols. These interactions all take the form of normal SQL statements: queries, updates, and triggers (rules).

The main body of Ravel is the runtime sitting in the middle. It is the powering engine that enables multiple controls to jointly drive the network. The runtime consists of two components. The database component interfaces with the users and is the brain that controls the variety of abstractions. The network component interfaces with the network, to bridge database events (table updates) and external network events (control messages and network changes).

The database runtime operates on a set of tables — the predefined base tables (§ 3) and a hierarchy of user-defined control application views. The base tables talk to the network runtime via SQL triggers, while the control views interface users with SQL queries and updates. The database runtime offer two services: vertical orchestration that "executes" individual application control, and horizontal orchestration that "coordinates" the executions.

Vertical orchestration (Figure 1, middle) is implemented by a view mechanism. View maintenance automatically refreshes application views, while view updates translate application updates to network controls. Modern databases are capable of automatic view update on simple cases. For complicated scenarios that involve ambiguity (e.g., a reachability view update corresponds to multiple viable path assignments in a lower base table), the operator can direct the database via a trigger before deployment, or alter the trigger behavior at run time.

Horizontal orchestration is implemented by a mediation protocol. The protocol assumes a priority ordering among the control applications. Upon an initial control request that attempts to modify the base data, the protocol consults all higher ranked applications that are affected, allowing them to check the proposed update and make modifications according to its local logic. The resulting updates are combined and populated back to the lower ranked applications. Finally, the orchestrated updates are grouped into a transaction committed to the base tables.

For example, as shown in Figure 1 (right), assume applications `lb,acl,rt` each maintain a property (in the usual sense) that is totally (pre-) ordered (left to right). Upon an initial control request that attempts to modify data (❶), the protocol consults all higher ranked applications (❷) that are affected, allowing them to check the proposed update and make modifications according to its local logic. The resulting updates (❷, ❸) are combined and populated back (❸) to lower ranked applications. Finally, the orchestrated updates are grouped into a transaction (❶ ❷ ❸) committed to the network (❹).

3. ABSTRACTIONS

Shared network state: the base tables

Ravel takes the entire network under the hood of a standard relational database, where the network states and control configurations are organized in two levels — the lower level is the tables storing the shared network state and the higher level is the virtual views pertaining to each application. We also include the network tables in the base. Ravel pre-defines the network base — a set of stored database tables — based on our extensive study of state-of-the-art control applications. The base table schema design is as follows:

```
tp(sid,nid)         # topology
rm(fid,sid,nid,vol) # end-to-end reachability (matrix)
cf(fid,sid,nid)     # configuration (forwarding table)
```

`tp` is the topology table that stores link pairs `(sid,nid)`. `rm` is the end-to-end reachability matrix that specifies for each flow (`fid`) the reachability (from `sid` to `nid`) and bandwidth (`vol`) information. `cf` is the flow table that stores for each flow (`fid`) the flow entries (the next hop `nid` for the switch `sid`). All the attributes here are symbolic identifiers with integer type for performance. For each attribute, Ravel keeps an auxiliary table that maps to real-world identifiers (e.g., an IP prefix for `sid` and `nid` in `cf`).

Ravel base tables provide fast network access and updates while hiding the packet processing details. They talk directly to the underlying network via OpenFlow. Initially, the base is loaded with the network configurations (e.g., topology, flow tables). As the network changes, the Ravel runtime updates the base (e.g., switches, links) accordingly. Similarly, as the base entries for flow rules change due to network control produced by the application, the runtime sends the corresponding control messages to the network.

While the network base design can be expensive, we argue that compared to the "fluid" control abstraction, it is relatively stable and retains the schema. Thus, we view the base schema design a one-time effort.

Application-specific policy: the SQL views

While the shared base talks continuously to the network, a separate higher level permits abstraction pertaining to individual applications. By running SQL queries on the base, SQL views derive from the base the data relevant to individual applications and restructure that data in a form that best suits the application's logic. SQL views allow a non-expert to add abstractions on demand, and the resulting views are immediately usable — referenced and updated via the database system runtime (§ 4). This frees users from committing to a fixed "right model that fits all", thus making Ravel abstractions ad-hoc extensible as requirements evolve.

For example, a load balancer abstraction is a view `lb` defined by the following SQL query.

```
CREATE VIEW lb AS(
       SELECT nid AS sid,
              count(*) AS load
       FROM rm
       WHERE nid IN server_list)
```

SQL views are also "composable", like function composition via procedure call, a composite view is built by a query that references other views. In a SQL query, views are indistinguishable from normal tables. Imagine a tenant network managed via view `tenant_policy` as follows:

```
CREATE VIEW tenant_policy AS (
   SELECT * FROM rm
   WHERE host1 IN (SELECT * FROM tenant_hosts)
         AND host2 IN (SELECT * FROM tenant_hosts));
```

`tenant_policy` monitors all traffic pertaining to the tenant slice. To manage load in this slice, we built a composite view `tlb` from `tenant_policy`, just like `lb` is built from the base `rm`:

```
CREATE VIEW tlb AS (
  SELECT sid,
         (SELECT count(*) FROM tenant_policy
         WHERE host2 = sid) AS load
  FROM tlb_tb);
```

Control loop: Monitoring and repairing policy violations with rules/triggers. SDN application dynamics typically follow a "control-loop": the application monitors (reads) the network state against its constraints, performs some computation, and modifies (writes) the network state according to its policy. For example, a load balancer checks traffic on servers of concern, and re-balances traffic when overloading occurs (e.g., the load on a server exceeds a threshold `t`).

Ravel allows a natural translation of application policy as integrity constraints over view data and the "control-loop" that maintains the policy as SQL rule processing, in the form:

```
  ON event DO action WHERE condition
```

For the load balancer, its invariant is a simple constraint `load < threshold (t)` over `lb`. Thus, we monitor the violations by a query on `lb` against the constraint:

```
CREATE VIEW lb_violation AS (
  SELECT * FROM lb WHERE load >= t);
```

To repair the violations, we simply empty the `lb_violation` view by a `DELETE` operation on `lb_violation`. We introduce the rule `lb_repair` to translate the repair onto the actual operations on `lb`, which simply sets the load to the default threshold:

```
CREATE RULE lb_repair AS
   ON DELETE TO lb_violation
   DO ALSO (
      UPDATE lb SET load = t WHERE sid = OLD.sid);
```

Another example action is to move two units of flow away from a particular server with id `k`:

```
UPDATE lb
  SET load = (SELECT load FROM lb WHERE sid = k) - 2
  WHERE sid = server_id;
```

4. ORCHESTRATION

4.1 Vertical orchestration via view mechanisms

Vertical orchestration is about synchronizing the views and the base by leveraging two view mechanisms. View mechanisms power view-based network control on individual applications: view maintenance populates base dynamics to the views, and view update translates view updates to the base (implementation). View maintenance is well-established, with a family of fast incremental algorithms developed over a decade [34, 10, 9]. Ravel implements classic view maintenance algorithms [10, 9] that automatically optimize application views for faster access (more in § 5, § 6). An interesting usage of view maintenance is that an application can ask interesting questions (e.g., a summary or aggregate) about it by submitting a SQL query on its view. For example, to find overloaded servers (exceeding some threshold `t`), the application simply writes:

```
 SELECT sid FROM lb WHERE load > t;
```

Conversely, view update pushes the changes on the view (by the application) back to the base (network). View update is a difficult and open research problem [13, 2, 4] because an update can correspond to multiple translations on the underlying base. To disambiguate, Ravel relies on user input for an update policy. For example, consider an update on `lb` that re-balances traffic to randomly chosen lightly selected servers. Specified by the following actions, it reduces the load on a server from `OLD.load` to `NEW.load` by picking a server with lowest load and redirecting the `OLD.load-NEW.load` oldest flows in `rm` to that server.

```
  UPDATE rm
    SET nid =
      (SELECT sid FROM lb
       WHERE load = (SELECT min (load)
                     FROM lb LIMIT (OLD.load - NEW.load))
       LIMIT 1)
    WHERE fid IN
       (SELECT fid FROM rm WHERE nid = NEW.sid
        LIMIT (OLD.load - NEW.load));
```

In general, users can program an update policy with rules of the form:

```
CREATE RULE view2table AS
  ON UPDATE TO view
  DO INSTEAD UPDATE TABLE --- actions ---
```

Here, `action` can be an arbitrary sequence of SQL statements, or a procedure call to external heuristics. This has the benefit of making Ravel abstractions open. Users can dynamically control the implementation of its high-level policy — how the view update maps to the underlying network — by simply creating or modifying the associated rules.

Similar to views derived directly from the base tables, nested views can be updated to manage the underlying network states. The update policy only needs to specify how changes on the view are mapped to its immediate source. For example, `tlb` only needs an update policy on `tenant_policy`, from which it is derived, and `tenant_policy` will handle the mapping down to the network base tables.

```
CREATE RULE tlb2tenant_policy AS
  ON UPDATE TO tlb
    DO INSTEAD
    UPDATE tenant_policy ...;
```

4.2 Horizontal orchestration via mediation protocol

Eventually, all control application operations are translated to updates on the network base data. To integrate the individual controls into a consistent base, it is sufficient to control their updates on the shared base. To this end, Ravel enhances the database runtime with a mediation protocol that instructs the shared data access.

The protocol offers participating applications three primitive operations. (1) *Propose*: an application attempts a tuple insertion, deletion, or update to its view. (2) *Receive*: an application observes updates (side-effects) reflected on its view due to network updates initiated by other participants. (3) *Reconcile*: an application checks the received view updates against its policy and performs either an accept (no violation), overwrite (to resolve a conflict), or reject (no conflict resolution exists). A conflict occurs when an update made by one application violates the constraints of another. Ravel adopts a simple conflict resolution policy based on priorities. Applications are required to provide a global ranking among all the constraints (one application can have multiple constraints) — higher-ranked constraints can overwrite updates from those lower-ranked when a conflict occurs.

Starting from a consistent network state where all application policies (invariants) are satisfied, the protocol takes an update proposal and a globally agreed priority list, and produces a set of orchestrated updates as follows. First, Ravel computes the effects of the proposal on other applications. A view update affects another view if the corresponding update on the shared data item causes modification to that view. Next, the affected applications sequentially reconcile their received update against their constraints in increased ranks. Finally, all the reconciled updates are merged and form the orchestrated output set. The orchestrated output is applied atomically as one transaction in the database, which transforms the network from its current state to the next.

Implementing the protocol is straightforward with PostgreSQL triggers [24, 33, 6, 33]: to enforce the priorities and properly invoke the overwrites, we only need to enforce the global ordering according to the priorities. Specifically, for each participant `Q`, assume some applications `P,Q` and associated priorities that satisfy `p<q<r`. We supplement `Q` with an extra priority table `q` and three rules `p2q, q2r, q_constraint. p2q` invokes updates at `Q` once `P` is done (`status = 'off'`), followed by `q_constraint` that performs the local reconciliation, and `q2r` (similar to `p2q`) that invokes the next application `R`.

```
CREATE TABLE q (counts integer, status text);

CREATE RULE p2q AS
  ON UPDATE TO p
  WHERE (NEW.status = 'off')
    DO ALSO INSERT INTO q values (NEW.counts, 'on');

CREATE RULE q_constraint AS
  ON INSERT TO q
  WHERE NEW.status = 'on'
  DO ALSO -- local reconciliation of application q;
            UPDATE q SET status = 'off';

CREATE RULE q2r AS ...
```

Correctness

Ravel orchestration handles two scenarios. In one, applications are called to collaborate for a common task (e.g., when a firewall drops an unsafe path, the routing component shall be invoked to remove the corresponding path). In the other, independent applications are prevented from in-adversely affecting each other (e.g., traffic engineering and device maintenance). Combined, the goal is to avoid partial and conflicting updates that lead a network into an inconsistent state. Formally, a network state is consistent if it is compliant with all the application policies. That is, consistent

Ravel component	Lines (#)	PostgreSQL features
network base	120	SQL
routing (rt)	230	SQL, rule, trigger
load balancer (lb)	30	SQL, rule
access control (acl)	20	SQL, rule
tenant (rt, lb, acl)	130	SQL, rule, trigger
runtime	200	rule, trigger
Mininet integration	260(SQL) 102(Python)	SQL, PL/Python

Table 1: Summary of Ravel components

network states correspond to applications with invariant-preserving views. A set of network updates potentially contributed by multiple applications is correctly orchestrated if the set transforms a consistent (i.e., correct) network state into only another consistent states.

PROPOSITION 1. *Ravel orchestration preserves network consistency.*

PROOF. (Sketch) The data mediation protocol translates each application update into a set of updates (contributed by all relevant applications) which, combined, satisfy all application constraints (subject to conflict resolution), thus preserving network consistency. □

5. IMPLEMENTATION

We built a prototype of Ravel using PostgreSQL [26] and Mininet [20]. PostgreSQL is an advanced, open source database popular for both academic and commercial purpose. It is highly customizable with many useful features (e.g., pgRouting [25]) for networking.

The prototype is implemented by 1000+ lines of SQL and 100+ lines of Python code, consisting of four components: the network base, the control applications (routing, load balancer, access control, and tenants), the runtime, and gluing code for integration with Mininet. Table 1 summarizes the implementation size and the PostgreSQL features used.

During the development of the prototype, we learned two optimization tactics that reflect the time-space tradeoff in a database-centered system. First, Ravel optimizes the base table schemas to avoid recursive path computation. By adding an additional reachability requirement table `rm` and triggers that recursively reconfigure the corresponding paths on the per-switch configuration table, higher-level views can control routing paths via non-recursive query and update over `rm`. Second, Ravel optimizes the performance of the application view by integrating a classic view maintenance algorithm [10], which avoids wasteful re-computation of the views from scratch as the network changes.

6. EVALUATION

To study the feasibility of Ravel, we run three sets of experiments on large fat-tree and ISP network topologies. We first examine the delay imposed by Ravel and microbenchmark database operations for route insertion and deletion. Next, we look at the scalability in orchestrating applications. Finally, we examine the overhead and performance improvement from view optimizations. The results are promising: even on a fat-tree topology with 196608 links and 5120 switches, and the largest ISP topology (Rocketfuel dataset) with 5939 switches and 16520 links, the delay for rule/trigger and key-value query overhead remains in the single-digit millisecond range. All experiments were run on a 64-bit VM with a single CPU and 7168M base memory.

fat-tree			ISP		
k	switches	links	AS#	nodes	links
16	320	3072	4755	142	258
32	1280	24576	3356	1772	13640
64	5120	196608	7018	25382	11292
			2914	5939	16520

Figure 2: Topology setup: (left) fat-tree with k pods, (right) four Rocketfuel ISP topologies with varying size.

Network setup

Figure 2 shows the three fat-tree topologies and four ISP topologies (taken from the Rocketfuel [31] dataset) we use. Among them, AS `2914` is pre-populated with a synthetic routing configuration from Route View [1] BGP feeds.

Profiling delay

We profile and analyze the various primitive database operations that constitute Ravel delay: `pgr_routing` (`rt`) that computes a shortest path between two nodes, `lookup ports` (`lk`) that fetches the outgoing port, `write to table (wt)` that installs the per-switch forwarding rules, and `trigger and rules` (`tr`) that optimize application operations and coordinate their interactions. We find the delay is dominated by the routing component (`rt`) and database writes (`wt`), which are dependent on the network topology. The additional database overhead of `lk,tr` remains low even for large network size, demonstrating good scalability. The details of the constituted delays are shown in Figure 3.

As the network size grows for fat-tree and ISP topologies, the delay caused by `tr` and `wt` operations increase linearly. Even for `fattree k=64`, both operations only double to 5ms. While the path-computation component `rt` grows and quickly dominates Ravel delay, it can be easily replaced by a faster implementation with better heuristics or precomputing k-shortest path. In contrast to the network size, the growth of the policy size has a smaller effect on the operations. This is because Ravel handles each `flow_id` independent of the rest of the policy space. Therefore, the scalability study below examines only the network size.

Scaling orchestration

We measure, on increasing network sizes, Ravel's delay in orchestrating applications including load balancer (`lb`), access control (`acl`), tenant network (`t`), and routing application (`rt`). `acl` removes (i.e., prohibits) traffic requirements from a blacklist. `lb` re-balances traffic requirements among a list of provisioned servers. `rt` is the component that actually realizes the traffic requirement with per-switch rules. Tenant network `t` is a 10-node full-mesh network that controls the traffic slice pertaining to its topology.

We examine vertical and horizontal orchestration scenarios. We use `x@t` to denote vertical orchestration of an application `x` controlling a tenant network `t`. For horizontal orchestration, we use `x+y+...` to denote applications `x,y,...` collectively controlling a network. For example, `lb+acl+rt` denotes the scenario that upon a newly inserted route, load balancer is engaged first, followed by access control, and finally by the routing application to set up the actual path on the underlying network state. The CDF in Figure 4 shows the overall orchestration delay in milliseconds.

We find that Ravel adds a small delay for orchestration, around 1ms for most scenarios. Delay is dominated by `rt` because of its semantics. `rt` must compute the path and reconfigure the switches. In contrast, `acl` imposes a negligible delay (<1ms) since it only needs to read from its blacklist, i.e., a fast key-value lookup. `lb` sits between these two applications and handles the extra path computation to direct traffic to a less loaded server. In particular,

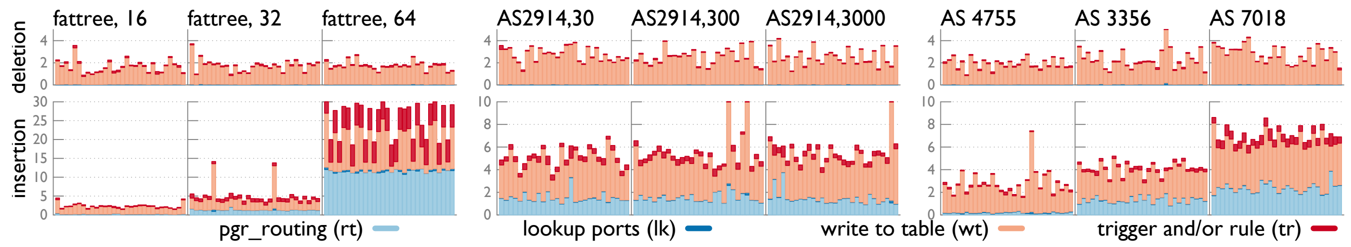

Figure 3: Sources of Ravel delay (ms) for route insertion and deletion.

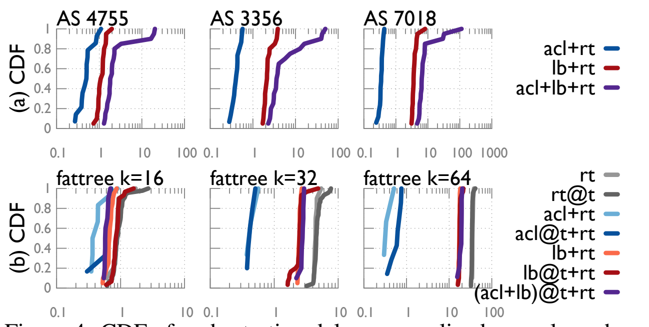

Figure 4: CDF of orchestration delay: normalized per-rule orchestration delay (ms) on various network sizes.

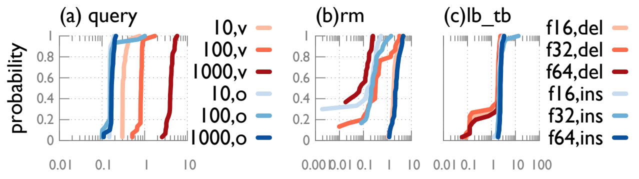

Figure 5: (a) CDF of querying (ms) on a view and its materialized equivalent. (b,c) CDF of maintenance delay (ms).

`lb+acl+rt` is bound by `rt`, and `x@t` is almost identical to that of `x`.

Optimizing application views

Ravel optimizes application views by translating them into equivalent materialized tables that offer faster access with small overhead. Figure 5 (a) compares the performance (query delay) on a load balancer view (`v`) and its materialized version (`o`) for three policy sizes (`10,100,1000`). Queries on optimized views (blue shade) are an order of magnitude faster (.1ms vs 1-2ms). As policy size grows (from 10 to 1000), the performance gain is more obvious. Figure 5 (b,c) shows the overhead of view maintenance, measured on three fat-tree topologies (`k=16,32,64`) and two scenarios: updates (deletion and insertion) to `lb_tb` and `rm`. In all cases, view maintenance incurs small delay (single-digit ms) that scales well to large network size.

7. RELATED WORK

Declarative networking. In the pre-SDN era, declarative networking [17, 16, 19] — a combined effort of deductive database (recursive datalog) and distributed system (distributed query optimization) research — uses a distributed recursive query engine as an extensible and efficient routing infrastructure. This allows rapid implementation and deployment of new distributed protocols, making it an alternative design point that strikes a balance among its peers like overlay [18] and active networks [11]. Ravel differs in every aspect. We build on relational database research, making novel use of SQL views and contributing new data mediation techniques, with target usage — mediating applications with higher-level user support in a centralized setting — better described in network OS and SDN programming APIs.

Database usage in network controllers. The use of database and the notion of network-wide views are not unfamiliar. Advanced distributed controllers such as Onix [15] and ONOS [3] provide consistent network-wide views over distributed network elements and multiple controller instance. Unlike Ravel, these systems use the database as a mere transactional repository to "outsource" state management for distributed and replicated network states, and treat the database as a passive recipient that only executes queries and transactions. Furthermore, the network-wide views are often predefined by the system (e.g., Onix's NIB APIs with fixed schemas for all control applications), making little use of user-centered database views. In Ravel, the database *is* the reactive controller with user-centered database views: control applications and the dynamic orchestrations are moved into the database itself, while SQL offers a native means to create and adjust application-specific ad-hoc views.

8. CONCLUSION

We present a novel SDN design, Ravel, based on a standard SQL database. With the simple and familiar SQL query, constraints, and triggers, non-experts can rapidly launch, modify, and switch between abstractions that best fit their needs. The database runtime, enhanced with view mechanisms and a data mediating protocol, allows multiple disparate applications — collaborative or competitive — to collectively drive the network in a user-defined meaningful way. A prototype built on the PostgreSQL database exhibits promising performance even for large scale networks.

9. ACKNOWLEDGMENTS

This work was supported by an NSA Science of Security grant, and NSF grant CNS 1513906.

10. REFERENCES

[1] Route views project. `http://www.routeviews.org`.
[2] Bancilhon, F., and Spyratos, N. Update semantics of relational views. *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 557–575.
[3] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., and Parulkar, G. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2014), HotSDN '14, ACM, pp. 1–6.
[4] Bohannon, A., Pierce, B. C., and Vaughan, J. A. Relational lenses: A language for updatable views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2006), PODS '06, ACM, pp. 338–347.

[5] DAYAL, U., AND BERNSTEIN, P. A. On the updatability of relational views. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4* (1978), VLDB '78, VLDB Endowment, pp. 368–377.

[6] DAYAL, U., HANSON, E. N., AND WIDOM, J. Active database systems. In *Modern Database Systems* (1994), ACM Press, pp. 434–456.

[7] FAGIN, R., ULLMAN, J. D., AND VARDI, M. Y. On the semantics of updates in databases. In *Proceedings of the 2Nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (New York, NY, USA, 1983), PODS '83, ACM, pp. 352–365.

[8] FOSTER, N., GUHA, A., REITBLATT, M., STORY, A., FREEDMAN, M. J., KATTA, N. P., MONSANTO, C., REICH, J., REXFORD, J., SCHLESINGER, C., WALKER, D., AND HARRISON, R. Languages for software-defined networks. *IEEE Communications Magazine 51*, 2 (2013), 128–134.

[9] GUPTA, A., AND MUMICK, I. S. Materialized views. MIT Press, Cambridge, MA, USA, 1999, ch. Maintenance of Materialized Views: Problems, Techniques, and Applications, pp. 145–157.

[10] GUPTA, A., MUMICK, I. S., AND SUBRAHMANIAN, V. S. Maintaining Views Incrementally. In *SIGMOD* (1993).

[11] HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. Plan: A programming language for active networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1998), pp. 86–93.

[12] JIN, X., GOSSELS, J., REXFORD, J., AND WALKER, D. Covisor: A compositional hypervisor for software-defined networks. In *NSDI* (2015).

[13] KELLER, A. M. Updating relational databases through views, 1995.

[14] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., AND CLARK, R. Kinetic: Verifiable dynamic network control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 59–72.

[15] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., AND SHENKER, S. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), OSDI'10.

[16] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative networking: Language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), SIGMOD '06, ACM, pp. 97–108.

[17] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative networking. In *Communications of the ACM* (2009).

[18] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing Declarative Overlays. In *SOSP* (2005).

[19] LOO, B. T., HELLERSTEIN, J. M., STOICA, I., AND RAMAKRISHNAN, R. Declarative routing: Extensible routing with declarative queries. *SIGCOMM Comput. Commun. Rev. 35*, 4 (Aug. 2005), 289–300.

[20] MININET. `http://mininet.org/`.

[21] MOGUL, J. C., AUYOUNG, A., BANERJEE, S., POPA, L., LEE, J., MUDIGONDA, J., SHARMA, P., AND TURNER, Y. Corybantic: Towards the modular composition of sdn control programs. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2013), HotNets-XII, ACM, pp. 1:1–1:7.

[22] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A compiler and run-time system for network programming languages. *SIGPLAN Not. 47*, 1 (Jan. 2012), 217–230.

[23] NELSON, T., FERGUSON, A. D., SCHEER, M. J. G., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (2014), pp. 519–531.

[24] PATON, N. W., AND DÍAZ, O. Active database systems. *ACM Comput. Surv. 31*, 1 (Mar. 1999), 63–103.

[25] PGROUTING PROJECT. `http://pgrouting.org/`.

[26] POSTGRESQL. `http://www.postgresql.org`.

[27] PRAKASH, C., LEE, J., TURNER, Y., KANG, J.-M., AKELLA, A., BANERJEE, S., CLARK, C., MA, Y., SHARMA, P., AND ZHANG, Y. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 29–42.

[28] REICH, J., MONSANTO, C., FOSTER, N., REXFORD, J., AND WALKER, D. Modular sdn programming with pyretic.

[29] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2013), HotSDN '13, ACM, pp. 109–114.

[30] SOULÉ, R., BASU, S., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Managing the network with merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2013), HotNets-XII, ACM, pp. 24:1–24:7.

[31] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. In *ACM SIGCOMM* (2002).

[32] SUN, P., MAHAJAN, R., REXFORD, J., YUAN, L., ZHANG, M., AND AREFIN, A. A network-state management service. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 563–574.

[33] WIDOM, J., AND CERI, S. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[34] ZHUGE, Y., GARCIA-MOLINA, H., HAMMER, J., AND WIDOM, J. View maintenance in a warehousing environment. *ACM SIGMOD Record 24*, 2 (1995), 316–327.