Software-Defined Networks as Databases

ABSTRACT

Software-Defined Networking seeks to make networks more flexible, with designs centering around programmability utilizing operating system (OS) and programming language (PL) abstractions. Although these SDNs have decoupled network programming from the physical infrastructure, they are still too low-level and inflexible from the perspective of network designers interested in application- and policy-level goals. We believe one of the key long-term challenges for SDN research is to develop abstractions that effectively navigate the tradeoff space of human convenience managing distributed data, and performance in a shared infrastructure.

We attack this abstraction challenge by championing a shift from OS/PL to database (DB)-oriented techniques. Our "Database-Based Networking" (*SDB*) approach utilizes the DB principle of *data independence* to allow dynamic creation, modification, and use of high-level (e.g. policy-level) abstractions or "views" of the distributed network on-demand, and exploits *transaction processing* to efficiently schedule user programs in a shared environment while preserving ACID. While this is an ambitious long-term goal, a prototype of several core features demonstrates promising performance, showing *SDB*-induced per-rule update latency of 14ms, for the most expensive DB operation on a datacenter network of 10k nodes (4.3k links). We also discuss the opportunities and challenges of *SDB*.

1. INTRODUCTION

In the 1980s, operating system (OS) and programming language (PL) techniques were proven to fall short for mediating between increasingly complex online data management needs and multiple users of little computer specialty. Likewise, network operations and infrastructure development today remain manual and difficult, despite the recent efforts to separate logical abstractions from physical infrastructure via OS and PL means [16, 15, 51]. For example, SDN introduces network OSes [37, 25] that maintain what is effectively a network-wide data structure, e.g., Network Information Base (NIB) [37], which is manipulated at a logically centralized controller through programming APIs [42, 24]. However, the network data-plane is still distributed and shared, enforcing the operator to explicitly deal with data redundancy, isolation, consistency, and recovery by composing complete programs out of wrapping APIs, a task that is notoriously challenging and tedious.

Similar to the transition of online commercial data management in the 1980s, this paper champions a shift from OS/PL techniques in networking to database (DB) techniques [2, 50, 45], adapting the two DB pillars—data independence and transaction processing—to the domain of networking. Data independence refers to abstractions that simplify human interaction by hiding data storage and representation, while transaction processing achieves concurrency and recovery control without severely hurting performance. In networking, *data-independent networking* offers a mechanism to create high-level abstractions of the distributed network data-plane, while *transactional networking* creates an illusion of isolated user operations in a shared infrastructure. By porting data independence and transaction processing into networking, we propose a **D**atabase **B**ased **N**etworking (*SDB*) design.

More specifically, *data-independent networking* offers a programmable user-level abstraction exposing a human interface that simplifies user logic, together with a language that creates the abstraction on demand, and a mechanism that pushes abstract operation back into the network device. The key is that, rather than seeking one kind of predefined abstraction that fits every existing user application[1] (and potentially future ones) [24, 42], we turn the network into a distributed relational database, where the base tables are switches' FIBs, and where we use SQL language to create abstractions called *(network) views* on demand – permitting high-level application-specific perspectives. For example, end-to-end reachability policy is nothing but a view – the abstract network-wide data-structure selected from the distributed FIBs and restructured in a form, e.g., a relation named `e2e_policy` of three attributes `{flow_id,source,dest}`, reducing high-level policy manipulation to simple DB operations. Similarly, routing policy is a view of `{flow_id, path_vector}`. To enable network management directly on views, *SDB* utilizes DB view mechanisms, namely view maintenance and update. View maintenance keeps abstractions updated as the network changes (i.e., FIB changes), enabling ad-hoc network verification by running queries over views against the latest network state. For example, to verify reachable switches for a flow with id `3` via a node `W` (waypoint), operator issues the following query:

```
SELECT e.dest
FROM   e2e_policy e NATURAL JOIN routing_policy r
WHERE  e.flow_id = 3 AND ('W' in r.path_vector);
```

which returns all the reachable nodes in real time. Conversely, view update compiles an abstract view operation into a collection of network FIBs operations, synthesizing the actual FIB implementation based on higher-level policy constraints. For example, to set up a new path for flow `4` between node `A` and `B`, the operator simply inserts this constraint into the policy view:

```
INSERT INTO e2e_policy VALUES (1, 'A', 'B');
```

[1]We use user program to refer to any control program, e.g., network-wide controller program by an administrator, or application programs by end users with limited network access.

which is translated by *SDB* into a set of FIB inserts.

Next, *SDB*'s *transactional networking* provides sequential and recoverable behavior of concurrent user operations, with *ACID* semantics: sequentiality ensures user operations proceed *atomically* and are *isolated* from one another, always leaving the network in a *consistent* state; and recoverability assures operation failure does not pollute network state, leaving effects of committed operation *durable*. Unlike conventional database systems (DBSes) where the operations, transaction, and their connections are obvious [8, 6, 26, 7], the interpretation in networking is obscure, as observed in early works [13, 40, 46]. The inherent dilemma is that transaction, the logical unit that preserves ACID, usually is meaningful only for network-wide actions; whereas the enabling mechanism usually is only efficiently enforceable for switch-level operation. *Transactional networking* solves this by leveraging the view abstraction introduced in *data-independent networking*: transactions, like other high-level operations, are specified over views; whereas the enforcing mechanisms are built on base tables. For example, if a transaction over a particular view is processed via two-phase locking, according to that view (the SQL query), *SDB* translates locks over the abstract view into a set of locks that can be performed locally at the relevant distributed base tables.

In sum, by leveraging decades of DB research, *SDB* introduces the following features:

- **Customizable and ad-hoc abstractions** that allow creation and change of abstractions on-demand. (§ 3, 4)
- **Realtime verification and synthesis** that provide online support for network state verification and ad-hoc requirement implementation. (§ 3, 4)
- **Transaction processing** that hides concurrency and recovery control. (§ 3)

2. EXAMPLES

This section discusses *SDB* features by examples.

Enterprise outsourcing: Enterprise network today demands new functionalities beyond the traditional end to end connectivity such as security and privacy. Enterprises are also moving their networks to cloud, leasing virtual networks from (multiple) remote datacenters rather than hosing a local infrastructure, which further complicate the issue. In response, enterprise outsourcing is emerging, which separates who own the network and who manage it by outsourcing the management task to a third party expert [48, 35]. Existing proposals, e.g., one-big-switch [35, 32] and network virtualization, expose enterprise network through unified primitive APIs. A network abstraction with easy, flexible, and full control is still missing. Can the enterprise expose only network aggregates, e.g., average bandwidth, or even just a range, without revealing the sensitive details? Can the enterprise adopt application-specific abstractions, and change abstractions as business goes?

To this end, *SDB* abstractions is ***customizable***, easily created and destroyed by the user: *SDB* abstraction does not enforce commitment to pre-defined freezing abstraction, granting users full control over what and how an outside party sees his network. By using SQL as abstraction definition and manipulation language, *SDB* also requires lesser computer specialty, lowering the bar for adoption.

Datacenter provisioning: In a datacenter that provides elastic cloud computing service [3, 10, 44], leases virtual network that is either pre-configured or customizable, the administrator is facing two problems: network provisioning, e.g., improving utilization by migrating VMs while respecting resource limits and customer SLAs; and service implementation, e.g., compiling customer's logical network configuration into the actual datacenter infrastructure. Virtualization techniques [19, 22, 17] automate many primitive operations that ease datacenter tasks. However, existing primitives are too primitive to achieve automatic online datacenter management. Can the administrator inspect the network state, checking high-level network-wide properties? Can the operator implement a customer's "non-standard" request without composing a complete program out of primitive wrapping APIs?

To this end, *SDB* performs ***realtime network verification and synthesis*** directly on user-defined abstractions. By the bi-directional data synchronizer between abstraction and the data-plane, administrator analyzes network state (e.g., check SLA conformance) by issuing a query over *SDB* abstraction, as simple as a SQL select statement. Conversely, arbitrary customer operations on logical network is automatically populated into to the datacenter without the administrator's supervision.

Distributed firewalls: Firewall offers a direct abstraction for specifying and enforcing network policy beyond standard routing. However, conventional firewall functionality relies on topology constraints, e.g., assuming that devices on the protected side are trusted. The placement of the firewall also introduces a choke-point to network performance such as throughput. Distributed firewalls is proposed [30, 5] to mitigate these shortcomings, implemented by a high-level language that specifies centralized firewall policy, and a mechanism that distributes the policy into end nodes. Such implementations, no matter how carefully designed to assure functional correctness, leaves behind concurrency and recovery control. While one may leverage existing consistent network update solutions for concurrency control, which, nevertheless, incurs overhead on network device or is too slow for online use. Recovery control that correctly roll back the network in case of failure is also missing.

To free users from the challenging concurrency and recovery control, *SDB* supports ***transaction processing***, where a centralized firewall operation is processed as a single logic transaction that is distributed over the network, by adapting transaction processing from distributed DB research, such as two-phase locking to achieve ACID semantics.

Integrable networking: Previously we demonstrate *SDB* features that separate high-level centralized user operation and low-level distributed implementation in a unified manner in a vertical architecture.This example shows that *SDB* also enables network integration horizontally. Consider merging two existing enterprise sites when two company department merges, or a network of independently administrated networks in the case of SDX (Software-defined Internet Exchange) [43], or the inter-operation of multiple controllers in a SDN-powered virtualized networks. To achieve a consistent inter-site network behavior, resolve potential conflicts, while minimizing the modification imposed at, maximizing the autonomy demanded by individual participant, what is the desirable "collaboration interface"?

These problems are challenging but not new, seen by DB community in 1990s when data integration research emerged to connect autonomous DBSes. We believe that, by lending itself to the rich ***data integration*** research such as federated DB [49], semi-structured DB [41, 12], *SDB* brings hope to a natural solution. Intuitively, the inter-site/controller network is managed through a super-abstraction, constructed from the per-site abstractions, in a way no different from the per-site ones. The super-abstractions form a public interface, where the per-site abstractions are for private management, hiding participant's internal data and isolating each another. Features of customizability, real-time support, and transaction processing also extend to the super-abstraction.

3. SYSTEM OVERVIEW

SDB's design is view-centric, application-specific data (e.g., policy) is nothing but a piece of selected and restructured view, managed through DB operations. This section outlines the two enabling components: *data-independent networking* that interfaces users with customer abstractions and *transactional networking* that processes operations.

data-independent networking offers a two-level data abstraction, internal base tables for network FIBs, and separate external views for simplifying user operation. *data-independent networking* is centered around the external views, which offers application-specific network-wide data-structures that are customizability, virtual, and updatable.

Views are customizable. While base (stored) tables are unified distributed over the network to achieve reasonable performance, as we will discuss more in *transactional networking*. Views are the external interface exposed to users, who can dynamically create and change views by SQL queries that select relevant information from base tables. Views also restructure the selected data to to simplify operations. A particularly useful class of views are policy views: the view schema structures the "network-wide data", specifying attributes of the derived data item. E.g., the schema of `end_to_end policy` view (Details in Table 2b) is `{flow, ingress, egress}`, structuring the view into three attributes. A view record represents a policy instance, e.g.,`(1,1,4)` directs that flow `1` entering the network by `1` is to exit the network via `4`. Unlike the network-view in network-OS or programming APIs, in *SDB*, users have complete control over views, can create or destroy views as needed, to simplify different applications at their will.

Views are virtual. Views are derived from the base tables, where the "derivation relation" is the SQL query that generates the view. E.g., a specific routing decision in the `end_to_end policy` view is derived from the forwarding rules stored in the base tables by a recursive query that computes end-to-end reachability. Since the output of a SQL query is a table by itself, views are used as the base tables, though only the view definition is stored. From performance perspective, a view consumes zero resource until it is referred i.e. queried by other programs. When a view is queried, the stored SQL query is simply re-computed. This treatment also keeps the view contents fresh, always reflecting the latest network configuration. This process is called view maintenance. Decades of DB query optimization have made view maintenance very fast, enabling real-time network verification by simply querying the views.

Views are updatable to enable network management directly through views. For example, an administrator re-selecting a path for all flows entering node `1` to exit via `3` (Figure 1), only need to "update" the `end_to_end policy view` (§ 4) records, like the following:

```
UPDATE e2e_policy SET egress = 3 WHERE ingress = 1;
```

SDB view update facility populates this update into the per_-switch `configuration` base table, like the following:

```
UPDATE configuration SET next = 2 WHERE switch = 1;
UPDATE configuration SET next = 3 WHERE switch = 2;
```

In general, view update which populates arbitrary modification to views into to the base is a harder one: since views are derived from the base, and in principle contains only partial information of the base, a unique base table update that implements the view update does not always exist. As a result, commercial DBS only supports updates for a limited class of views when an unambiguous one-one mapping between base and view exists. In *SDB*, however, we implement view update for a larger family of views (e.g., reachability) via triggers (DBS's equivalent for call-back functions). We will revisit view update in § 4.

The data considered in *SDB* is the forwarding plane consisting of all the switch configurations, i.e. FIB. These "distributed" rules naturally form *SDB base (stored) tables* and become *SDB*'s internal representation of the network forwarding plane. Just like a distributed network FIB is primarily designed for traffic processing, the base tables are designed for transaction processing and are not exposed to *SDB*'s end users. Externally, *SDB* exposes a separate programmable abstraction called *(network) views*. The customizable view is simply a SQL query which takes base tables as input, and outputs a new virtual table. A view typically contains network-wide data, e.g., network data selected from nodes belonging a path or a spanning tree, whichever relevant to a particular user's task. The view data is also restructured to simplify user logic. In the following, we use an example network (Figure 1) to illustrate the details.

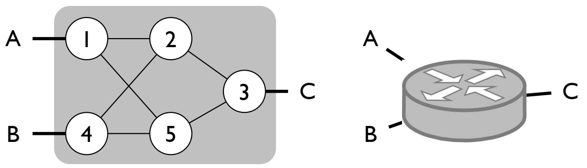

Figure 1: Example network and its one big switch abstraction

As base tables represent a network's forwarding state, its role is to hide hardware heterogeneity, enable transaction processing, and ease the creation of external abstractions. Hence, they hold all the network (configuration) data in a unified form that is easily accessible. A natural choice is a relational model [29] that consists of three tables: `topology` table that models network as a pool of resource capacity, `configuration` table that is the union of all FIBs, and an optional `constraint` table for network constraints (e.g., SLAs). E.g., Table 1 shows example table instances. All tables are populated when the network is set up (assuming the FIBs configured pro-actively), and incrementally updated afterwards as network topology and/or configuration changes.

(a) topology

node	node	avail_bw	used_bw
1	2	3	2
1	5	5	0
		...	
4	5	5	0
4	2	5	0
		...	

(b) per_switch configuration

switch	flow	next	bw
1	1	2	1
1	2	2	1
	...		
2	1	3	1
	...		
3	1	C	1
	...		

Table 1: Example base tables

The user-defined virtual views are external abstractions interfacing with users that are derived from the base tables as SQL queries and serve as a logical perspective for a user's task. Compared to the logical contexts introduced in earlier works [16, 17], the strength of *SDB* views is that it does not confine users to preselected frozen abstractions, instead *SDB* views can be created and changed by users on demand by simply issuing or modifying a SQL query, which selects from the distributed base tables the relevant information and restructures them into a network-wide data-structure that fits the user's task.

(a) routing policy

flow	path_vector
1	(1,2,3)
2	(1,2,4)
3	(4,5,3)
	...

(b) end_to_end policy

flow	ingress	egress
1	1	3
2	1	4
3	4	3
	...	

(c) one_big_switch

flow	next
1	C
2	B
3	C
...	

Table 2: Example views.

For example, a network-wide routing policy is nothing but the abstract data as a view (Table 2a), derived from the per-switch `configuration` table, by the following query:

```
CREATE VIEW routing_policy AS (
  SELECT DISTINCT c.flow_id, fp.path_vector
  FROM configuration c
       NATURAL JOIN flow_policy_fun(c.flow_id) fp
  ORDER BY c.flow_id );
```

In line 2, the `select` statement selects two attributes to from the view schema: attribute `flow_id` from `configuration` and `path_vector` from `flow_policy_fun` which is a recursive function that computes routing path for `flow_id`. Similarly, we can derive a one-big-switch configuration view from `configuration` table and the `obs_mapping` table which keeps tracks of the constituting physical nodes `p_node`.

```
CREATE VIEW obs_configuration AS (
  SELECT flow_id, t.next_id
  FROM   configuration t INNER JOIN obs_mapping ob
  ON     t.switch_id = ob.p_node );
```

Since a view is virtual, only its definition (the query) is stored. Codd's relational model ensures that a SQL query outputs nothing but tables (relations), allowing users to use views exactly as a table. A direct usage is to create views on top of views. The next example shows how to create an end-to-end policy (Table 2b) view from `routing_policy` view:

```
CREATE VIEW e2e_policy AS (
 SELECT flow_id,
       path_vector[1] AS ingress,
       path_vector[array_length(path_vector,1)] AS egress
 FROM  routing_policy
 ORDER BY flow_id ) ;
```

4. VIEW INTERFACE

The design goal of *SDB* view interface is to combine the strength of the following:

- An interface that loads and transforms the dataplane states. The challenge is to identify a small set of views that are also expressive enough for common networking tasks. (§ 4.1)
- Composition. (§ 4.3.2)
- Built-in services of real-time verification and synthesis. (§ 5)
- Performance. The challenge is that a naive implementation of relational queries does not scales well for the networking setting of SDN, where most interesting abstractions, by nature, will call for path-related computation that is recursive, and hence expensive in the naive implementation. (§ 6)

To achieve all the above, *SDB* features a library of view primitives. ... create abstractions on the fly that can be categorized in two groups: (1) the per-flow views ... We also call this type "local views", because ... (2) the network-wide views ... we call this type "global views". (1) enables real-time verification and synthesis; (2) provides the interface for integrating network-wide service such as traffic engineering. This section presents (1,2) in details. Services and performance are discussed in § 5 and § 6.

4.1 Abstraction hierarchy

4.2 Base tables

Before introducing the derived views, let us first discuss in depth the base tables – the flat universe of networking data that are actually stored in the database. The base tables serve two roles: (1) a flat universe of tables to which all network configuration states and state changes can be easily loaded; (2) the bases for building layers of view abstractions where the reverse view update shall be (relatively) easy.

```
CREATE OR REPLACE FUNCTION reachability_perflow(f integer)
RETURNS TABLE (flow_id int, source int, target int, hops bigint) AS
$$
BEGIN
DROP TABLE IF EXISTS tmpone;
CREATE TABLE tmpone AS (
SELECT * FROM configuration c WHERE c.flow_id = f
) ;

RETURN query
        WITH ingress_egress AS (
SELECT DISTINCT f1.switch_id as source, f2.next_id as target
               FROM tmpone f1, tmpone f2
       WHERE f1.switch_id != f2.next_id AND
      f1.switch_id NOT IN (SELECT DISTINCT next_id FROM tmpone) AND
              f2.next_id NOT IN (SELECT DISTINCT switch_id FROM tmpone)
                ORDER by source, target),
     reach_can AS(
                SELECT i.source, i.target,
              (SELECT count(*)
                        FROM pgr_dijkstra('SELECT 1 as id,
                 switch_id as source,
   next_id as target,
   1.0::float8 as cost FROM tmpone',
     i.source, i.target,TRUE, FALSE)) as hops
        FROM ingress_egress i)
SELECT f as flow_id, r.source, r.target, r.hops FROM reach_can r where r.hops != 0;
END
$$ LANGUAGE plpgsql;
```

4.3 Virtual views

4.3.1 View primitives

Per-flow forwarding graph and reachability views

Lies in the heart of any pair-wise reachability views, be it reachability for a plain network, for an one-big-switch network, or for an arbitrary virtual network, is a query like the following:

4.3.2 View composition

From user perspective, a network view is a derived table that offers the same tabular interface as ordinary (i.e. materialized table that is actually stored) tables. This allows to a stacking of views ... ... We use "composition" to loosely refer to the derivation of views from existing views. ...

Consider one big switch, its per-flow forwarding graph view can be built on top of per-flow forwarding graph view and one big switch topology view, as follows:

```
CREATE OR REPLACE VIEW obs_1_fg_36093_3 AS (
       select 1 as id,
              switch_id as source,
      next_id as target,
      1.0::float8 as cost
       FROM obs_1_topo, fg_36093
       WHERE switch_id = source AND next_id = target
       ORDER BY source, target
);
```

Note that, `fg_36093` is used as a filter for selecting `obs_1_topo` records that ... Symmetrically, one could also "reverse" the filtering and use `fg_36093` as a filter instead, as follows:

```
CREATE OR REPLACE VIEW obs_1_fg_36093_4 AS (
       select 1 as id, source, target,
      1.0::float8 as cost
       FROM obs_1_topo, fg_36093
       WHERE switch_id = source AND next_id = target
       ORDER BY source, target
);
```

While the above two view composition, namely SQL join, is very intuitive: its body of "view join" directly translates. It is no longer updatable. ... As an alternative equivalent view is by ... where one table is used as an ... parameter ...

```
CREATE OR REPLACE VIEW obs_1_fg_36093_2 AS (
       select 1 as id,
              switch_id as source,
      next_id as target,
      1.0::float8 as cost
       FROM configuration NATURAL JOIN topology
       WHERE flow_id = 36093 AND subnet_id = 1
       ORDER BY source, target
);
```

To make advantage of view updates, *SDB* adopts this approach. Unfortunately, views are not allowed to be paramterized in SQL standard. To achieve the affect of parameterized views, *SDB* uses Python wrapper as walk-round, as follows:

5. VERIFICATION AND SYNTHESIS

5.1 Verification and synthesis as data synchronization

more here ... (this subsection) HotNet texts, will rework

A network is in constant change. A virtual view is useful only if its records are fresh – reflecting the latest network instantly. For example, when per-switch rules change, a query on the high-level policy view `e2e_routing` (Table 2b) shall automatically returns the updated `e2e` reachability. Conversely, to enable network manipulation via views, the base tables need to be updated to reflect operations on the views. For example, to set a new route `(1,5,4)` for `flow 1` in the example network (Figure 1 (left)), a user simply insert a new record denoting the path into the `routing_policy` view with `flow` attribute set to `1`. *SDB* is responsible of pushing this abstract view insert into the relevant base `configuration` inserts.

Generally, view maintenance that keeps virtual views fresh and view update that synthesizes the base table changes, jointly form a bi-directional data synchronizer between the base and the view. While modern DBSes implement view maintenance very efficiently, view update is supported for restricted cases [34, 11]. This is no surprise, as view update is the harder one: a view contains only partial information of the base, it is not always possible to locate a unique base table update [4]. To enable network operations on views, *SDB* takes advantage of existing view maintenance implementation and extends the support of view updates to network view updates.

Real-time view maintenance enables network verification. *View maintenance* is well supported in modern database systems (DBS), which *SDB* adopts straightforwardly. Specifically, when a view generated by SQL query program q, is queried by a SQL program p, view maintenance translates p on q into queries $p \circ q$ on the base tables. Interestingly, view maintenance offers exactly what is needed in real-time network verification: by specifying the property of interests as p over q, view maintenance performs check of $p \circ q$ on network states on the fly.

Real-time view update enables network synthesis. Given the ambiguity and non-existence in view update, we first characterize the correctness criteria in networking. We identify updates that keep a view's independent and complementary counter-parts constant. Two views are *independent* if the update on one does not affect that on the other. Two views are *complementary*, if they contain enough information to recover the base tables. A view updates that keeps the independent views constant eliminates accident changes made to other existing views; An update that keep a view's *complementary* constant is a stronger requirement that does not pollute any possible views (existing and future ones).

SDB assumes user views are independent, and only performs updates that keep independent views constant. In the current prototype, view update is implemented by hand coded triggers, the call-back functions that are automatically fired to update the bases when the associated view update is issued. We evaluate this manual implementation (details in § 6) to measure the DB induced delay. Ultimately, *SDB* aims for a generic view update algorithm that synthesizes for any user-defined views. (We have sketched a novel algorithm, omitted due to space.) We leave the implementation of the generic algorithm for future work.

5.2 Verification example

5.3 Synthesis example

```
select * from vn_reachability where flow_id = 77899 ;
 flow_id | ingress | egress
---------+---------+--------
   77899 |     486 |     19
   ...
```

This entry corresponds to a record in configuration as follows:

```
SELECT flow_id, pv FROM configuration_pv WHERE flow_id = 77899;
---------+---------------------------------
 flow_id |               pv
   77899 | 486,498,462,463,456,472,109,19
```

To update the virtual network policy that revoke transient service for flow `77899` between ingress `486` and egress `19`, the user could direcly modify the `vn_reachability` table by deleting the corresponding record as follows:

```
DELETE FROM vn_reachability WHERE
        flow_id = 27079 AND ingress = 486 AND egress = 19;
```

This deletion results in the deletion of three switch-level configurations as follows:

```
[[462, 463], [486, 498], [498, 462]]
```

The reason that only three entries at switches `462, 486, 498` are removed is that the rest of the ... are

```
SELECT flow_id, pv FROM configuration_pv WHERE flow_id = 77899;
 flow_id |                pv
---------+---------------------------------
   77899 | 483,463,456,472,109,19
   77899 | 486,498,462,463,456,472,109,19
   ...
```

Similarly, on user request for adding new or updating exiting virtual network end to end policy, *SDB* synthesizes the relevant switch-level configuration modification. Add new policy:

```
INSERT INTO vn_reachability VALUES (55716, 557, 483);
```

Update policy:

```
UPDATE vn_reachability SET egress = 230
        WHERE flow_id = 97940 AND ingress = 497;
```

Finally, it is worth noting that, a policy update request may not always be valid. For example, a deletion of policy `(42692, 497, 375)` in ...

```
SELECT source, target, pv FROM configuration_pv WHERE flow_id = 42692;
 source | target |                 pv
--------+--------+---------------------------------
    486 |    375 | 486,498,462,463,456,472,108,375
    497 |    230 | 497,462,463,456,230
    497 |    375 | 497,462,463,456,472,108,375
    ...
```

While *SDB*, which utilizes the default view updates mechanism implemented by postgres, will simply remove policy `(42692, 497, 375)` from `vn_reachability` view, and leaves the per-switch configuration unchanged.

6. EVALUATION

We develop a prototype featuring view interface and verification/synthesis services. Our prototype serves two purposes: First, we use it to study the feasibility of managing SDN data-plane with database. In particular, we gathered performance overhead introduced by database on various ISP topologies (up to) with configurations initialized with real routeview feeds (2 million). ... ; Second, with the prototype, we explored two fundamental tradeoff: expressiveness (of views) versus operation performance, and automation (of updates) versus performance. We conclude that ...

The prototype is implemented in PostgreSQL [47], an advanced database that is popular for both academia and commercial use, on ...

6.1 Feasibility study

6.2 Performance vs. Expressivenss

6.3 Performance vs. Automation

7. DISCUSSION AND FUTURE WORK

7.1 Network updates and synthesis

Forwarding plane updates via query optimization

Network synthesis via database provenance

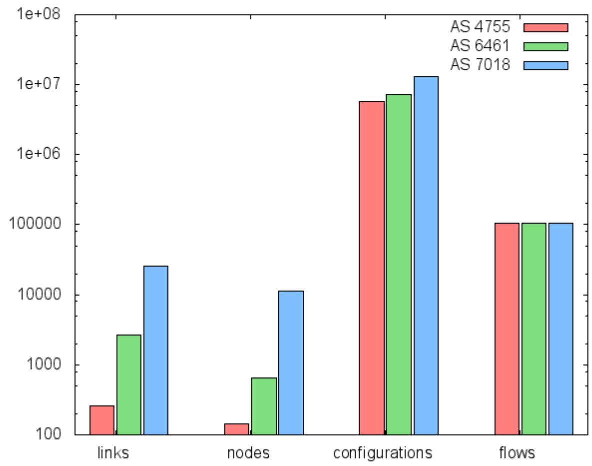

Figure 2: Configuration size.

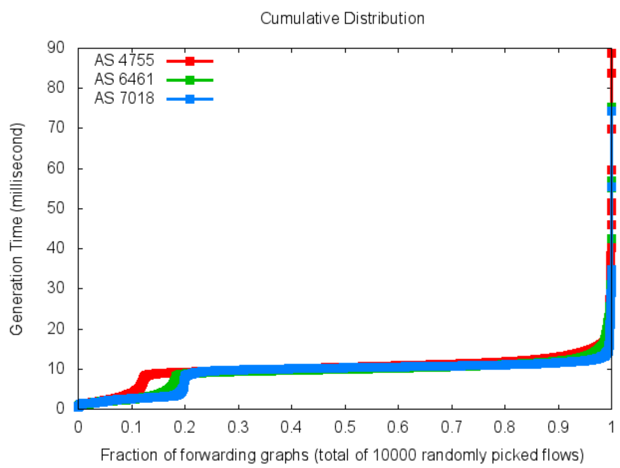

Figure 3: Forwarding graph generation.

7.2 Distributed forwarding plane

Transactional networking offers an efficient execution ab-

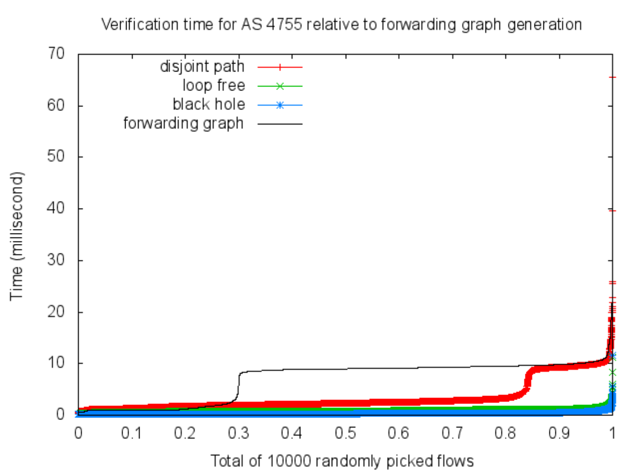

Figure 4: Verification overhead.

straction of user programs in a shared distributed network, freeing the user from the challenging concurrency and recovery control problem [23, 6, 26, 7, 52, 9].

Transaction preserves ACID properties. In *SDB*, a transaction is a logical unit of operations that are atomically and isolated from one another, preserving network consistency and preventing failure from polluting effects of already committed transaction. The operations in a transaction are partially-ordered, defined in the user program. An operation is either a read or write: a write maps to network (re)configuration in the form of an insert or delete[2] of records in a base table or view; a read, on the other hand, maps to packet processing, since packet processing is the effect of "read" policy data. An example transaction is the collection of flow events interleaved with network reconfigurations issued by a user program. By DB concurrency and recovery control, *SDB* executes transactions concurrency while retaining the ACID semantics.

Transactions on views. Like users interact with *SDB* via views, they also conceive transactions on views. Specifically, we write $(T, v, \overline{op})$ for a transaction T with operation set $\overline{op} = op_1, \cdots op_n$ on a view v. Users tell *SDB* of T by wrapping his program like the following:

```
Start; program (op1;...opn;) Commit;
```

The key to efficient transaction processing for parallel programs is a scheduler that coordinates data access of operations while preserving ACID. It has been shown that the scheduling problem decompose to two sub-problems: resolving conflicting read-write operations, and that for write-write operations [8]. Read-write conflict occurs between a configuration update (write) and processing of in-fly traffic (read) that will be affected by the update; write-write conflict occurs when the updated data items overlap. A standard scheduler that prevents both conflicts is two-phase locking [8] where a transaction $(T, v, \overline{op})$ becomes $(T, v, (lock(v), \overline{op}, unlock(v)))$, that is:

```
Start; lock(v); program (op1;...opn;) unlock(v); Commit;
```

Transactions on base tables at switches. Transactions on views are inefficient: a view is by nature application-specific, typically network-wide, involving multiple distributed nodes (e.g., that forms a path). Transactions over views require synchronization among the participant nodes, making `lock(v)` and `unlock(v)` complex tasks that lack performance. Hence rather than adding concurrency and recovery enforcement to views, *SDB* implemented them at base tables, where the locks can be performed locally at individual node. To enable this, *SDB* translates a transaction $(T, v_{routing}, (lock(v_{routing}), op, unlock(v_{routing})))$ on a path defined by a routing policy view $v_{routing}$ to a set of base table transactions $(T, b_i, (lock_{b_i}, op_{b_i}, unlock_{b_i}))$ where b_i, the base table derived from the view $(v_{routing})$. Operations on b_i proceed independent of each other. When multiple T_i is executed, two-phase locking over views is

[2] An update is a deletion followed by an insertion

achieved by switch-level locking that enforce a consensus of partial ordering among conflicting operations.

8. CHALLENGES

Data plane is a distributed system where datas are partitioned and replicated. The network-wide forwarding states are partitioned across nodes

Data plane is a distributed system with realtime demands

9. RELATED WORK

Network verification and synthesis, despite recent efforts [36, 33, 39, 53], have not matured into a wanted industry of standard reusable tools with a general-purpose interface. General-purpose verification tools, despite powerful tools like SMT solvers, are not directly applicable to network size; Domain-specific heuristics are fast but require extra effort and are not reusable. Formal synthesis is even harder and slower: existing tools does not scale well [54]. By utilizing DB's general query engine that has been optimized for two decades as the main reasoning engine, *SDB* brings new hope to a networking verifier that strikes a balance between general-purpose support and realtime performance. We would throughly study the power and limits of *SDB*'s support for realtime verification and synthesis.

Views, authorization, and locking are three inherently connected concepts in database [18]: views expose data, authorization prescribes access to data, and locking implements authorization. Thus, the view-centric *SDB* lends itself to a rich body of locking based authorization mechanism that is particularly straightforward and flexible: one may associate lock with different operations e.g., write or/and read, and data of different granularity e.g., entire table, one record, one column, or arbitrary region defined by a condition. As such, we envision that *SDB* may hold the solution to the increasingly complicated network security and privacy requirement (e.g., isolation) where users could conveniently project the network data into views, with authorization granted via locks.

Network OS and SDN programming APIs introduce a global network abstraction of the distributed forwarding plane along with programming APIs that offers reusable primitives [37, 25, 42, 24]. Similar to pre-database era, where online commercial data management used file system and general-purpose programming language, we view OS/PL powered SDN as a preliminary stage of *SDB* [20, 14], which is restrictive and requires considerable expertise. *SDB* seeks a more flexible and accessible user interface that is customizable and managed through simple databases operators that are processed as transactions.

Declarative networking [38, 21] draws on the natural connection of recursive Datalog (declarative deductive database language) and network properties such as reachability, presents a Datalog based platform for specifying distributed networking services. Later work on declarative network management [28] extends Datalog to policy management for enterprise networks. Compared to these efforts, *SDB* applies DB techniques in a broader sense, covering the two main DB pillars: data independence and transaction processing. On the other hand, declarative networking sheds light on many issues *SDB* is facing, ranging from the details in connecting DB to a real network [28, 38], to the data-independence principle discussed in [27][3].

10. CONCLUSION

This paper champions a shift from OS/PL to DB-oriented techniques towards more flexible and manageable networks. Our *SDB* approach features customizable abstractions, realtime verification and synthesis, and transaction processing by applying DB principles of *data independence* and *transaction processing* into networking domain. While this is an ambitious long-term goal, a prototype of several core features demonstrates promising performance.

11. REFERENCES

[1] *Traffic Engineering with Forward Fault Correction*, ACM Association for Computing Machinery.

[2] ABITEBOUL, S., HULL, R., AND VIANU, V., Eds. *Foundations of Databases: The Logical Level*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] AMAZON ELASTIC COMPUTE CLOUD (AMAZON EC2). `http://aws.amazon.com/ec2/`.

[4] BANCILHON, F., AND SPYRATOS, N. Update semantics of relational views. *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 557–575.

[5] BELLOVIN., S. M. Distributed firewalls. In *November 1999 issue of; login*.

[6] BERNSTEIN, P. A., AND GOODMAN, N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*.

[7] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv.*.

[8] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[9] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[10] BOBROFF, N., KOCHUT, A., AND BEATY, K. Dynamic placement of virtual machines for managing sla violations.

[11] 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.

[12] BUNEMAN, P. Semistructured data. *PODS* (1997).

[13] CANINI, M., KUZNETSOV, P., LEVIN, D., AND SCHMID, S. Software transactional networking: Concurrent and consistent policy composition. In *Proceedings of HotSDN* (New York, NY, USA, 2013).

[3] [27] deals solely with physical data-independence, whereas logical data-independence also plays a key role in *SDB*

[14] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec. 1985), 471–523.

[15] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., GUDE, N., MCKEOWN, N., AND SHENKER, S. Rethinking enterprise network control. *IEEE/ACM Trans. Netw. 17*, 4 (Aug. 2009), 1270–1283.

[16] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., AND SHENKER, S. Ethane: taking control of the enterprise. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 1–12.

[17] CASADO, M., KOPONEN, T., RAMANATHAN, R., AND SHENKER, S. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow* (New York, NY, USA, 2010), PRESTO '10, ACM, pp. 8:1–8:6.

[18] CHAMBERLIN, D. D., GRAY, J. N., AND TRAIGER, I. L. Views, Authorization, and Locking in a Relational Data Base System. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition* (New York, NY, USA, 1975), AFIPS '75, ACM, pp. 425–430.

[19] CHOWDHURY, N. M. M. K., AND BOUTABA, R. Network virtualization: State of the art and research challenges. *Comm. Mag.*.

[20] DATE, C. J., AND HOPEWELL, P. File definition and logical data independence. In *Proceedings of the 1971 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (New York, NY, USA, 1971), SIGFIDET '71, ACM, pp. 117–138.

[21] DECLARATIVE NETWORKING. `http://p2.berkeley.intel-research.net/`.

[22] DRUTSKOY, D., KELLER, E., AND REXFORD, J. Scalable network virtualization in software-defined networks. *IEEE Internet Computing, 2013*.

[23] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM* (1976).

[24] 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.

[25] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev. 38*, 3 (July 2008), 105–110.

[26] HAERDER, T., AND REUTER, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv.* (1983).

[27] HELLERSTEIN, J. M. Toward network data independence. *SIGMOD Rec. 32*, 3 (Sept. 2003), 34–40.

[28] HINRICHS, T. L., GUDE, N. S., CASADO, M., MITCHELL, J. C., AND SHENKER, S. Practical declarative network management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking* (New York, NY, USA, 2009), WREN '09, ACM, pp. 1–10.

[29] HULL, R. Relative information capacity of simple relational database schemata. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems* (1984), p. 97109.

[30] IOANNIDIS, S., KEROMYTIS, A. D., BELLOVIN, S. M., AND SMITH, J. M. Implementing a distributed firewall. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2000), CCS '00, ACM, pp. 190–199.

[31] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 3–14.

[32] KANG, N., LIU, Z., REXFORD, J., AND WALKER, D. Optimizing the "one big switch" abstraction in software-defined networks.

[33] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 99–112.

[34] KELLER, A. M. *Updating Relational Databases Through Views*. PhD thesis, 1995.

[35] KELLER, E., AND REXFORD, J. The "Platform as a service" model for networking. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking* (Berkeley, CA, USA, 2010), INM/WREN'10, USENIX Association, pp. 4–4.

[36] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 15–28.

[37] 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* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[38] 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.

[39] LOPES, N., BJORNER, N., GODEFROID, P., AND VARGHESE, G. Network verification in the light of program verification.

[40] MAHAJAN, R., AND WATTENHOFER, R. On consistent updates in software defined networks (extended version). Tech. Rep. MSR-TR-2013-99, November 2013.

[41] MCHUGH, J., ABITEBOUL, S., GOLDMAN, R., QUASS, D., AND WIDOM, J. Lore: A database management system for semistructured data. *SIGMOD Record 26* (1997), 54–66.

[42] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-Defined Networks. Proc. Networked Systems Design and Implementation, April 2013.

[43] NICK FEAMSTER, JENNIFER REXFORD, S. S. D. L. R. C. J. B. Sdx: A software dened internet exchange. ONS'13.

[44] NIRANJAN MYSORE, R., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. Portland: A scalable fault-tolerant layer 2 data center network fabric.

[45] PAPADIMITRIOU, C. H. Database metatheory: Asking the big queries. *SIGACT News 26*, 3 (Sept. 1995), 13–30.

[46] PEREŠÍNI, P., KUZNIAR, M., VASIĆ, N., CANINI, M., AND KOSTIŪ, D. Of.cpp: Consistent packet processing for openflow. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (New York, NY, USA, 2013), HotSDN '13, ACM, pp. 97–102.
[47] POSTGRESQL. `http://www.postgresql.org`.
[48] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making middleboxes someone else's problem: Network processing as a cloud service.
[49] SHETH, A. P., AND LARSON, J. A. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv. 22*, 3 (Sept. 1990), 183–236.
[50] SILBERSCHATZ, A., KORTH, H., AND SUDARSHAN, S. *Database Systems Concepts*, 5 ed. McGraw-Hill, Inc., New York, NY, USA, 2006.
[51] THE FUTURE OF NETWORKING, AND THE PAST OF PROTOCOLS. `http://www.opennetsummit.org/archives/apr12/site/talks/shenker-tue.pdf`.
[52] TRAIGER, I. L., GRAY, J., GALTIERI, C. A., AND LINDSAY, B. G. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*.
[53] WANG, A., BASU, P., LOO, B. T., AND SOKOLSKY, O. Declarative Network Verification. In *PADL* (2009).
[54] WANG, A., MOARREF, S., LOO, B. T., TOPCU, U., AND SCEDROV, A. Automated synthesis of reactive controllers for software-defined networks.