Flow Algebra: Towards an Efficient, Unifying Framework for Network Management Tasks

Paper #1707 (1570668823)

Christopher Leet
Yale University
christopher.leet@aya.yale.edu

Robert Soulé
Yale University
robert.soule@yale.edu

Yang Richard Yang
Yale University
yry@cs.yale.edu

Ying Zhang
Facebook
zhangying@fb.com

***Abstract*—A modern network needs to conduct a diverse set of tasks, and the existing approaches focus on developing specific tools for specific tasks, resulting in increasing complexity and lacking reusability. In this paper, we propose Flow Algebra as a unifying, easy-to-use framework to accomplish a large set of network management tasks. Based on the observation that relational databases based on relational algebra are well understood and widely used as a unifying framework for data management, we develop flow algebra based on relational algebra. On the other hand, flow tables, which are the fundamental data specifying the state of a network, cannot be stored in traditional relations, because of fundamental features such as wildcard and priorities. We define flow algebra based on novel, generalized relational operations that use equivalency to achieve efficient, unifying data store, query, and manipulation of both flow tables and traditional relations. We realize flow algebra with FlowDB and demonstrate its ease of use on diverse tasks. We further demonstrate that generality and ease-of-use do not need to come with a performance penalty. For example, for the well-studied network verification task, our system outperforms two state-of-the-art network verification engines, NoD and HSA, in their targeted domain, by 55x.**

***Index Terms*—SDN, Network Management**

I. Introduction

Modern network management involves a diverse range of tasks such as policy management (*e.g.,* to ensure network performance meets the operator's SLAs), routing cost computation (*e.g.,* during traffic optimization), switch data plane optimization (*e.g.,* to improve switch performance), debugging (*e.g.,* to diagnose an underperforming service) and dataplane composition (*e.g.,* when managing multiple virtual networks).

Despite these use cases' diversity each primarily (and often solely) involves manipulating tables of data: flow tables stored in the network dataplane, tables of network configuration data stored by configuration management tools (*e.g.,* FBNet [1]), and network policy, network traffic, geospatial, etc. data stored by the network operator in traditional databases.

Given this common thread, it might seem natural to accomplish each network management task by placing the preceding tables in a database and then analyzing and manipulating these tables using SQL queries. Unfortunately, flow tables are not traditional relations and contain features such as actions, priority and wildcard expressions which make it difficult to manipulate them with the relational algebra or to combine them with data in traditional relations.

In response to this challenge, we introduce the *flow algebra*, an extension of the relational algebra to flow tables. The flow algebra provides a novel formal algebraic framework for the composition and decomposition of flow tables permitting operators, for the first time ever, to accomplish a wide range of network management tasks using a single, unified framework. Each flow algebra expression is closed over flow tables, allowing the flow algebra to perform flow table manipulation tasks such as database refactoring and composition. The flow algebra, views a traditional relation as a type of flow table, allowing it to query relations from traditional databases during tasks such as policy management or network debugging.

While the flow algebra alone provides a complete framework for many network management tasks, some tasks, such as network debugging, often require the generation of one or more packet histories [2]: the sequence of packet states (a packet's physical location, its header fields, and potentially its logical location within a switch datapath and any metadata currently associated with it) a packet passes through as it travels from one network endpoint to another. Generating a history involves recursively combining datapath flow tables. Rather than burdening the programmer with building network histories using a generic, hard-to-optimize recursion operator, we introduce a novel *virtual* table called *the history table*, which conceptually contains every packet history in a network. The history table is treated by a normal table by the flow algebra but is materialized lazily for performance.

We implement the flow algebra in a novel system termed FlowDB, a novel relational database which can store, query and manipulate both flow tables and traditional relations. FlowDB is queriable using an augmented dialect of SQL, *FlowSQL*, which contains additional keywords that allow the programmer to search the history table for complex patterns. It employs two key optimizations taken from prior work. First, each table taken from the data plane is indexed using atomic predicates [3]–[5] to improve join performance. Second, a FlowDB query can be incrementally updated if its underlying tables change using a similar mechanism to CoVisor [6].

FlowDB is evaluated on two real, large scale networks: Stanford's backbone network [7] and a datacenter switching

fabric operated by a large internet company containing over 10K+ switches and 100 million+ flow table rules. Our evaluations show that FlowDB can perform a wide range of network management tasks, including cost map generation, dataplane refactoring and network debugging in 50 - 70 seconds. We extended two network verification engines, Header Space Analysis [8] and Network Optimized Datalog [9] to perform network debugging and cost map generation and found that FlowDB outperformed these tools by 50 - 300×.

Benefits. The flow algebra provides several key benefits:

(1) *A Formal Algebraic Framework.* The flow algebra provides a formal algebraic framework for flow table manipulation. Just as the relational algebra allows programmers to reason about SQL queries, the flow algebra allows operators to reason about flow table operations.
(2) *Flow Table Closure.* The output of any flow algebra expression is a flow table, allowing operators to refactor and optimize the data plane using the flow algebra.
(3) *Expressiveness.* A broad range of network management tasks (*e.g.,* routing cost computation, network debugging) are naturally expressable in flow algebra.
(4) *Inheriting Relation Algebra's Properties.* The flow algebra inherits the relational algebra's algebraic properties, allowing FlowDB to leverage existing query planner optimizers to optimize query execution.
(5) *Data Compatibility.* Many internet companies (*e.g.,* Google [10], Facebook [1]) store configuration/policy data in relations which flow algebra directly can act on.
(6) *Ease of use.* The overwhelming popularity of SQL databases among developers attests to its ease of use. Stack Overflow [11] found that the most used DBs were MySQL, PostgreSQL, Microsoft SQL server and SQLite, used by 54.0%, 34.3%, 32.8% and 31.6% of respondents.

Novel Contributions. This paper's novel contributions are:

(1) *Flow Algebra. (§IV, §V)* The flow algebra is a formal algebraic framework created by extending the relational algebra to flow tables. Whilst its natural join and union operators were anticipated in [6], its theta join, selection, projection and set difference operators are novel, as is its development of their algebraic properties.
(2) *History Table.* (§VI) The Flow Algebra is augmented with the history table, a virtual table which allows programmers to directly query network histories.
(3) *FlowDB and FlowSQL. (§VII, §VIII)* FlowDB is the first ever SQL-based database which can store, query and manipulate both flow tables and traditional relations. It is programmable with FlowSQL, a dialect of SQL augmented with keywords to search the history table. Together, they allow common network management tasks to be succinctly expressed.
(4) *Evaluation. (§IX)* We evaluate FlowDB, and find that it can execute a wide range of network management tasks on real networks at reasonable speeds. We show that FlowDB outperforms two state of the art network verification engines by 50× or more.

II. Related Work

Network Optimized Datalog. FlowDB's closest comparison is Network Optimized Datalog (NoD) [9]. NoD is a verification engine which places a network's flow tables in a Datalog database and then performs verification with Datalog queries. Datalog is a superset of the relational algebra, and while NoD's authors do not do so one could imagine writing Datalog queries to perform a broad range of tasks.

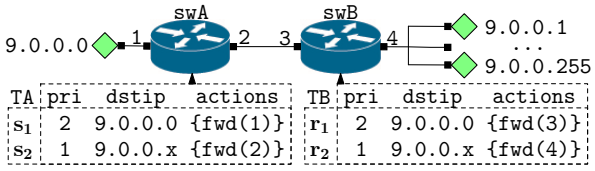
NoD, however, does not emit flow tables and cannot be used for datapath manipulation. Further, whilst NoD introduces new table manipulation operators, it does not provide a formal algebraic framework. Practically, many network operators, (e.g., Google, Facebook [1]), store policy and network data in MySQL, not Datalog, making FlowDB less disruptive to their software ecosystems. Moreover, despite Datalog's venerability [12] and expressiveness, empirically developers prefer to build systems on SQL-based databases [11].

Flow Table Manipulation Tools. CoVisor [6], a compositional hypervisor for software define networks, defines operators for flow table composition in serial (parallel) which anticipate Flow Algebra's natural join (union) operator. CoVisor, however, does not place these operators in a theoretical framework, establish their algebraic properties, or allow decomposition.

Computing Network Properties. A wide range of tools compute network properties. In particular, network verification is a well researched field. A wide range of tools compute network properties. In particular, network verification is a well researched field. Plankton [13], Minesweeper [14], Batfish [15], ERA [16] and ARC [17] perform network verification by analyzing the control plane. HSA [8], Veriflow [18] and APKeep [3]–[5] do the same by analyze the data plane. Vera [19], p4v [20] and P4-assert [21] target P4 programs, whilst Buzz [22], SymNet [23], NetSMC [24] and [25] target stateful networks. Beyond verification, tools such as P4P [26], ALTO [27], and FlowDirector [28] measure (and publish) the cost of routing packets between two network end points.

None of these tools, however, emits flow tables, and thus none of them can manipulate the data plane. Moreover, these tools cannot easily perform analysis spanning network and non-network data. The results of a database query could be injected into many of these tools, but this lacks the ease of use of working in a single framework. Finally, these tools are often non-trivial to generalize out of their targeted domain. One can conceive of an extension to Plankton, for example, that measures routing costs, but the average developer would likely find implementing it difficult.

Other Database-Based Network Systems. Ravel [29], NDLog [30] and SeNDLog [31] provide a declarative query interface for describing and instantiating networks. SONATA [32], Gigascope [33] and NetQRE [34] analyze streams of traffic data. None of these systems, however, analyzes flow tables, and thus all are orthogonal to FlowDB.



TA	pri	dstip	actions
s_1	2	9.0.0.0	{fwd(1)}
s_2	1	9.0.0.x	{fwd(2)}

TB	pri	dstip	actions
r_1	2	9.0.0.0	{fwd(3)}
r_2	1	9.0.0.x	{fwd(4)}

Fig. 1. The network myNet.

III. Problem Formulation

Many common network management tasks fall into three general categories: (i) computing network properties (e.g. cost map generation, network verification), (ii) manipulating the dataplane (e.g. dataplane refactoring, dataplane composition) and (iii) computation spanning network and non-network data (e.g. network debugging, policy management.) A concrete example from each category is given below.

Cost Map Generation. Flow Director [28] (among other systems [26], [27]) allows ISPs to provide Content Delivery Networks (CDNs) with *"cost maps"*, listing the cost of routing traffic between end points. To generate a cost map, the ISP computes each route a packet takes through the network by combining the network's routing tables with its topology (often also stored in a database). A distance metric (e.g., the number of long haul links traversed) is then computed for each route.

Datapath Refactoring. Consider an flow table that reads a packet's destination MAC address and VLAN tag, modifies its VLAN tag, and assigns it an output port: `T : dmac, vlan → mod_vlan, out`. Suppose that `T` has the functional dependency `mod_vlan → out` (i.e., `out` can be determined from `mod_vlan`). Németh et al. [35] suggest that `T` should be refactored into two smaller, redundancy free tables T_0 : `dmac, vlan → mod_vlan` and T_1 : `mod_vlan → out` to minimize datapath churn and improve switch performance.

Network Debugging. Suppose a data center operator is informed that a service hosted in her data center is failing intermittently. To diagnose the problem, she might:

(1) Query a database to determine which servers the service is running on and which IP addresses it is using.
(2) Compute the routes the service's traffic is forwarded along (from the network's flow tables and topology.)
(3) Check the logs of each device on each route for abnormalities (e.g. packet errors, CPU spikes)

Despite the diversity of these tasks, each one only involves manipulating tables of data. As a result, each one is simple to implement (as our evaluations show) if all relevant data is placed in a single relational database. Unfortunately, flow tables are not traditional relations and contain features which are difficult for the relational algebra to manipulate.

Consider the network, `myNet` (Fig. 1.). This network contains two switches `swA` and `swB`, programmed with the tables `TA` and `TB`. Switch `swA` is connected to the host at `9.0.0.0`, while switch `swB` is connected to the 254 hosts `9.0.0.1` to `9.0.0.255`. Suppose `myNet`'s operator asks:

Which packets can travel from the host on `swA` to the hosts `9.0.0.0` to `9.0.0.3` on `swB`?

Intuitively, the operator can answer this question by: (i) checking which packets with `dstip` between `9.0.0.0` and `9.0.0.3` are forwarded by `TA` out port `2`, and then (ii) checking which of these packets are forwarded by `TB` out port `4`. If `TA` and `TB` were traditional relations, the operator could accomplish this task by applying a selection to each relation and then joining the results as follows:

$$\texttt{TB}' \leftarrow \sigma_{\texttt{fwd(4)} \in \texttt{actions}}(\texttt{TB}).$$
$$\texttt{TA}' \leftarrow \sigma_{\texttt{fwd(2)} \in \texttt{actions} \wedge \texttt{9.0.0.0} \leq \texttt{dstip} \leq \texttt{9.0.0.3}}(\texttt{TA}).$$
$$\texttt{out} \leftarrow \texttt{TA}' \bowtie_{\texttt{TA}'.\texttt{dstip}=\texttt{TB}'.\texttt{dstip}} \texttt{TB}'.$$

Unfortunately, the relational algebra cannot be applied to flow tables like `TA` and `TB` because of their actions, priority and wildcard expressions.

Problem (i): Actions. Constructing `TB'` (line 1) requires extending selection to handle actions. Naïvely, one could view an action set as a set of strings to be searched for a desired string (here `fwd(4)`). This approach works for `TB`, but fails when an action's parameter takes a variable or expression instead of a constant. Naïve string matching as above would reject a rule with the action `fwd(defaultPort)` (found, e.g., in [36]), even when `defaultPort` = 4!

As a further complication, an action's parameter may be non-constant over the range of packets its rule matches on. For example, a data center switch might balance traffic between two ports using the action `fwd(dstip%2 + 3)`. If rule $\mathbf{r_2}$ used this action, should it be included in `TB'`? This action evaluates to `fwd(4)` for packets with `dstip` = `9.0.0.1` but `fwd(3)` for packets with `dstip` = `9.0.0.2`.

Problem (ii): Priority. Constructing `TB'` also requires selection to handle priority. Naïvely performing selection by selecting each rule in `TB` whose action set contains `fwd(4)` yields a flow table containing the single rule:

```
(1, 9.0.0.x, fwd(2))
```

This result incorrectly implies that `TB` forwards packets with `dstip` = `9.0.0.0` out port `2`. Unlike a tuple in a relation, a rule in a flow table's semantic meaning can be affected by other rules. Selection can not ignore a rule that doesn't satisfy its predicate if that rule overrides a rule that does.

Problem (iii): Wildcard Expressions. Constructing `TA'` (line 2) further requires selection to handle wildcard expressions. Promisingly, existing databases do implement certain wildcard expressions. PostgreSQL's INET datatype, for instance, can represent an IP address with a wildcard suffix (e.g. `9.0.0.x`). Unfortunately, FlowDB cannot exploit these implementations because they treat wildcard expressions as atomic values. PostgreSQL, for instance, compares two INET values on the smallest address each represents. Executing line 2 in PostgreSQL therefore yields rule $\mathbf{r_2}$ when it should yield the rules $(2, \texttt{9.0.0.0}, \texttt{fwd(1)})$ to $(1, \texttt{9.0.0.3}, \texttt{fwd(1)})$. Selection must split a rule if some values it represents satisfy its predicate but not others.

[TA]	dstip	port
r_1	9.0.0.0	1
r_2	9.0.0.1	2
...	...	...
r_{256}	9.0.0.255	2

TAS	pri	dstip	port
r_1	2	9.0.0.0	1
r_2	1	9.0.0.x	2

Fig. 2. TA's canonical relation, [TA], (left), TA simplifed, TAS (right).

IV. Overview

A. Applying the Relational Algebra to Flow Tables

Despite these problems, the expressiveness of relational algebra, its rich array of optimizations, its familiarity to programmers and its potential as a common query language for both flow tables and regular database relations make it a potentially attractive platform for network analytics.

FlowDB observes that a flow table, T, can be viewed as a map from a set of input header fields to (i) a set of output ports (if T contains forwarding actions), (ii) some other flow table (if T contains jump actions) and (iii) a set of output header fields (if T contains write actions). This map can be represented in a database relation. We term this relation T's canonical relation and denote it [T]. TA's canonical relation, [TA], is shown in Fig. 2. (left).

A flow table can be converted into its canonical relation by: (i) expanding each rule with a wildcard expression match field into a set of single valued rules that collectively match each packet the original rule matched, (ii) evaluating each expression a rule's action contains on these match field values[1] and then (iii) removing any overridden rule.

One could therefore apply the relational algebra to a flow table by converting it into its canonical relation and then placing this relation in a database. Unfortunately, since a wildcard expression with k wildcards over a n-letter alphabet represents n^k atomic values, expanding wildcard expressions can result in an exponential increase in the size of a flow table. Even though TA has 2 rules, [TA] has 255 tuples: one for each IP addresses between 9.0.0.0 and 9.0.0.255.

B. The Flow Algebra

To avoid generating and manipulating such large quantities of data, we take a second, related approach. As before, we conceptually view a flow table, T, as a compressed representation of a canonical relation, $[T]$. However, when we want (for example) to apply the unary relational algebra operator op to T, instead of decompressing T and then applying op to $[T]$, we define a new version of op which can act directly on T's compressed (flow table) representation. This operator maps T to a new flow table, T', whose canonical relation $[T']$ is the relation $op([T])$. We term this operator the *flow table equivalent of op* or the *flow op*, and denote it $\underline{op}$. Collectively, these flow operators make up the *flow algebra*.

We formalize the preceding ideas by introducing the notation of *relational algebra (RA) equivalence*.

[1]adding new match fields as necessary to fix the expression's operands

Definition IV.1. *An unary (binary) flow algebra operator $\underline{op}$ is* relational algebra (RA) consistent *to a relational algebra operator op iff for any flow table T (flow tables, T_1 and T_2):*

$$[\underline{op}(T)] = op([T])$$
$$([T_1 \ \underline{op} \ T_2] = [T_1] \ op \ [T_2])$$

With RA-equivalence, we can formally define each flow algebra operator.

Definition IV.2. The flow algebra operator $\underline{op}$ *is defined as an operator which is RA-consistent to the relational algebra operator op and closed over flow tables.*

From definition IV.2, it follows that the flow algebra inherits the relational algebra's algebraic properties. For instance:

Lemma IV.1. *Iff the relational algebra operator op is commutative (associative), the corresponding flow algebra operator $\underline{op}$ is commutative (associative).*

Proof (Commutativity). Let T_1 and T_2 be arbitrary flow tables.

$$\begin{aligned} [T_1 \ \underline{op} \ T_2] &= [T_1] \ op \ [T_2] && \text{(RA-equiv.)} \\ &= [T_2] \ op \ [T_1] && \text{(op commutes)} \\ &= [T_2 \ \underline{op} \ T_1] && \text{(RA-equiv.)} \end{aligned}$$

Proof (Associativity). Let T_3 be an arbitrary flow table.

$$\begin{aligned} [T_1 \ \underline{op} \ (T_2 \ \underline{op} \ T_3)] &= [T_1] \ op \ [T_2 \ \underline{op} \ T_3] && \text{(RA-equiv.)} \\ &= [T_1] \ op \ ([T_2] \ op \ [T_3]) && \text{(RA-equiv.)} \\ &= ([T_1] \ op \ [T_2]) \ op \ [T_3] && \text{(op associates)} \\ &= [T_1 \ \underline{op} \ T_2] \ op \ [T_3] && \text{(RA-equiv.)} \\ &= [(T_1 \ \underline{op} \ T_2) \ \underline{op} \ T_3]. && \text{(RA-equiv.)} \end{aligned}$$

Similar theorems exist for the inheritance of distributivity, cascade, and idempotence. These inherited properties allow Flow Algebra expressions to be rewritten using the same equivalence rules as relational algebra expressions, allowing an existing relational algebra query planners (with a modified cost function) to optimize Flow Algebra queries.

C. Implementing the Flow Algebra

A valid implementation of flow algebra operator, $\underline{op}$, is a function which satisfies Definition IV.2. There are thus multiple valid implementations of each operator. Our objective is to find an implementation with acceptable space/time complexity. We implement our flow algebra by exploiting this idea.

Definition IV.3. *Two flow tables, T_1 and T_2, are equivalent, $T_1 \sim T_2$, if they have the same canonical representation.*

$$T1 \sim T2 := [T1] = [T2].$$

Lemma IV.2. *If flow tables T_1 and T_1' and T_2 and T_2' are equivalent, flow tables $T_1 \ \underline{op} \ T2$ and $T_1' \ \underline{op} \ T_2$' are equivalent if $\underline{op}$ is RA-consistent.*

$$T_1 \sim T_1' \wedge T_2 \ \sim T_2' \Rightarrow T_1 \ \underline{op} \ T_2 \sim T_1' \ \underline{op} \ T_2'.$$

We therefore instantiate the flow algebra by: (1) replacing each flow table with a simpler, equivalent table and then (2) instantiating each operator on the simplified tables.

We simplify a flow table by reducing each of its actions to a (1) jump to a second flow table, (2) forwarding action to a set of ports or (3) constant valued write to a header/metadata field. This is done by evaluating any expressions a rule's action contains on the values in its match field (adding new match fields if required, and splitting the rule if an expression is resolved ambiguously.) In theory, modern network programming languages such as P4 permit a rule's action to contain complex logic which cannot be easily resolved. Such actions were absent in the major internet provider's network we evaluated on. We view extending the flow algebra to flow tables with very complicated actions as a direction for future work.

After simplification, a flow table can be represented as a traditional relation by adding an attribute for each priority and header field the table writes. If the table contains jump or forwarding actions, the attributes `jump` and `out` are also added. We term these attributes *"action attributes"* and the original match fields *"match attributes"*. If a rule does not write to an action attribute `attr`, `attr` is also added as a match attribute and the rule takes the special value `=attr`, indicating a nop. Fig. 2. (right) shows `TA` simplified, `TAS`.

V. The Flow Algebra

A. A Formal Flow Table Model

We instantiate RA-consistent flow algebra operators for: Cartesian product, selection, projection, union, set difference, natural join, theta join and group by.[2] In this section, we present flow Cartesian product and flow selection: the other operators can be found in the expanded technical report [37].

Before we can instantiate our operators, we need a formal model of a simplified flow table T. Let $\mathbf{x}$ be a vector of values in the domain of T's match attributes. Let $\mathbf{y}$ be a vector of values in the domain of T's action attributes. For instance, if T was `TA`, $\mathbf{x}$ might be `(9.0.0.2)` and $\mathbf{y}$ might be `(2)`.

We say that T represents the tuple $\mathbf{xy}$ if $\mathbf{xy} \in [T]$, denoted $\mathbf{x} \Subset T$. We now define how T achieves this representation.

A simplified flow table T is a set of flow table rules. A flow table rule, $\mathbf{s}$, is a a tuple $(pri, \mathbf{m}, \mathbf{a})$, where pri is the flow rule's priority, $\mathbf{m}$ is a vector of match attribute values and $\mathbf{a}$ is a vector of action attribute values.

Definition V.1. *A flow table rule $\boldsymbol{s} := (pri, \boldsymbol{m}, \boldsymbol{a})$ represents the tuple $\boldsymbol{xy}$, denoted $\boldsymbol{xy} \Subset \boldsymbol{s}$ iff each field in $\boldsymbol{m}$ matches the corresponding value in $\boldsymbol{x}$ and likewise for $\boldsymbol{y}$ and $\boldsymbol{a}$.*

For example, rule $\mathbf{s_2}$ in `TAS` matches `(9.0.0.0, 2)`.

Definition V.2. *The match set of a tuple of values $\boldsymbol{xy}$ in a flow table T, denoted $T(\boldsymbol{xy})$ is the set of flow table rules whose match fields m match on the tuple's match fields $\boldsymbol{x}$.*

$$T(\boldsymbol{xy}) = \{(pri, \boldsymbol{m}, \boldsymbol{a}) : \boldsymbol{m} \ matches \ \boldsymbol{x} \wedge (pri, m, a) \in T\}.$$

[2]Each operator except for projection, set difference and group by, in certain cases, runs in $O(n^2)$ time, where n is the number of rules in the largest operand table.

For example `TAS((9.0.0.0, 2))` = $\{\mathbf{s_1}, \mathbf{s_2}\}$.

Observe that, since a flow table may not have two equal priority rules that match on the same packet, each rule in a match set must have a different priority, Let the highest priority rule in a match set, $T(\mathbf{xy})$, be denoted $max(T(\mathbf{xy}))$.

Definition V.3. *A flow table T represents the tuple $\boldsymbol{xy}$ if the highest priority rule in $\boldsymbol{xy}$'s match set $T(\boldsymbol{xy})$ represents $\boldsymbol{xy}$.*

$$\boldsymbol{xy} \Subset T := T(\boldsymbol{xy}) \neq \emptyset \wedge \boldsymbol{xy} \Subset max(T(\boldsymbol{xy})).$$

For example, `TAS` does not represent `(9.0.0.0, 2)` since $\mathbf{s_1}$, the highest priority rule in `TAS`$(\mathbf{xy})$ does not represent it.

B. Efficient RA-Consistent Cartesian Product

Theorem V.1. *An operator, $\underline{op}$, which takes two flow tables, A and B, and outputs a flow table, $A \ \underline{op} \ B$, is RA-consistent with Cartesian product iff:*

$$\boldsymbol{xy} \Subset A \wedge \boldsymbol{uv} \Subset B \Leftrightarrow \boldsymbol{xuyv} \Subset A \ \underline{op} \ B.$$

Proof. Theorem V.1, V.4 and V.5 follow from Definition IV.1

Theorem V.2. *An operation, $\underline{\times}$, which takes each pair of rules in two flow tables, A and B, and outputs a rule with the sum of their priorities, the union of their match fields and union of their action fields is RA-consistent with Cartesian Product.*

$$A \underline{\times} B := \{(p_A + p_B, \boldsymbol{m_A m_B}, \boldsymbol{a_A a_B}) : (p_A, \boldsymbol{m_A}, \boldsymbol{a_A}) \in A \wedge (p_B, \boldsymbol{m_B}, \boldsymbol{a_B}) \in B\}.$$

Our proof of Theorem V.2 uses two key Lemmas.

Lemma V.1. *The rule $(p_A, \boldsymbol{m_A}, \boldsymbol{a_A})$ is in $A(\boldsymbol{xy})$ and the rule $(p_B, \boldsymbol{m_B}, \boldsymbol{a_B})$ is in $B(\boldsymbol{uv})$ iff the rule $(p_A + p_B, \boldsymbol{m_A m_B}, \boldsymbol{a_A a_B})$ is in $(A \underline{\times} B)(\boldsymbol{xuyv})$.*

Proof. We write:

$$\begin{aligned}
&(p_A, \mathbf{m_A}, \mathbf{a_A}) \in A(\mathbf{xy}) \wedge (p_B, \mathbf{m_B}, \mathbf{a_B}) \in B(\mathbf{uv}) \\
&\quad \Leftrightarrow \mathbf{x} \ matches \ \mathbf{m_A} \wedge \mathbf{u} \ matches \ \mathbf{m_B} \\
&\quad \Leftrightarrow \mathbf{xu} \ matches \ \mathbf{m_A m_B} \\
&\quad \Leftrightarrow (p_A + p_B, \mathbf{m_A m_B}, \mathbf{a_A a_B}) \in (A \underline{\times} B)(\mathbf{xuyv}).
\end{aligned}$$

Lemma V.2. *The rule $(p_A, \boldsymbol{m_A}, \boldsymbol{a_A}) = max(A(\boldsymbol{xy}))$ and the rule $(p_B, \boldsymbol{m_B}, \boldsymbol{a_B}) = max(B(\boldsymbol{uv}))$ iff the rule $(p_A + p_B, \boldsymbol{m_A m_B}, \boldsymbol{a_A a_B}) = max((A \underline{\times} B)(\boldsymbol{xuyv}))$.*

Proof ($\Leftarrow$). Suppose, for the sake of contradiction, that $max((A \underline{\times} B)(\mathbf{xuyv})) = (p_A + p_B, \mathbf{m_A m_B}, \mathbf{a_A a_B})$ but that $max(A(\mathbf{xy})) \neq (p_A, \mathbf{m_A}, \mathbf{a_A})$. Let $max(A(\mathbf{xy})) = (p'_A, \mathbf{m'_A}, \mathbf{a'_A})$. By Lemma V.1, $(p'_A + p_B, \mathbf{m'_A m_B}, \mathbf{a'_A a_B})$ is in $(A \underline{\times} B)(\mathbf{xuyv})$ and, since $p'_A > p_A$, this rule has higher priority than the original rule which was $max(A(\mathbf{xy}))$. $\Rightarrow\Leftarrow$. A symmetric argument exists for $max(B(\mathbf{uv}))$.

Proof ($\Rightarrow$). Suppose, again for the sake of contradiction, that $max(A(\mathbf{xy})) = (p_A, \mathbf{m_A}, \mathbf{a_A})$ and $max(B(\mathbf{uv})) = (p_B, \mathbf{m_B}, \mathbf{a_B})$ but $max((A \underline{\times} B)(\mathbf{xuyv})) \neq (p_A + p_B, \mathbf{m_A m_B}, \mathbf{a_A a_B})$. Let $max((A \underline{\times} B)(\mathbf{xuyv})) = (p'_A + p'_B, \mathbf{m'_A m'_B}, \mathbf{a'_A a'_B})$.

By Lemma V.1, $(p'_A, \mathbf{m'_A}, \mathbf{a'_A}) \in A(\mathbf{xy})$. Thus $p'_A \leq p_A$. By Observation V.2, if $p'_A = p_A$ then $(p'_A, \mathbf{m'_A}, \mathbf{a'_A}) = (p_A, \mathbf{m_A}, \mathbf{a_A})$. A similar argument exists for p'_B and p_B. Thus if:

- $p'_A < p_A$ and $p'_B \leq p_B$, $p'_A + p'_B < p_A + p_B$. $\Rightarrow\Leftarrow$.
- $p'_A \leq p_A$ and $p'_B < p_B$, $p'_A + p'_B < p_A + p_B$. $\Rightarrow\Leftarrow$.
- $p'_A = p_A$ and $p'_B = p_B$, then $(p'_A + p'_B, \mathbf{m'_A m'_B}, \mathbf{a'_A a'_B}) = (p_A + p_B, \mathbf{m_A m_B}, \mathbf{a_A a_B}) = max((A \underline{\times} B)(\mathbf{xuyv}))$. $\Rightarrow\Leftarrow$.

Theorem V.1 follows directly from Lemma V.2.

C. Efficient RA-Consistent Selection.

Theorem V.3. *An operator, $\underline{op}$, that takes a flow table T and a predicate p and outputs a flow table $\underline{op}(T, p)$ is RA-consistent iff:*

$$\boldsymbol{x} \in A \wedge p\boldsymbol{x} \Leftrightarrow \boldsymbol{x} \in \underline{op}(T, p).$$

Our efficient RA-consistent flow selection operator is built on top of our efficient RA-consistent intersection operator:

Theorem V.4. *An operator, $\underline{op}$, which takes two flow tables, A and B, and outputs a flow table, $A \ \underline{op}\ B$, is RA-consistent with set intersection iff:*

$$\boldsymbol{xy} \in A \wedge \boldsymbol{xy} \in B \Leftrightarrow \boldsymbol{xy} \in A \ \underline{op}\ B.$$

Let $\underline{\wedge}$ be an RA-consistent set intersection operator. Further, let T_p be a flow table such that $\mathbf{x} \in T_p \Leftrightarrow p(\mathbf{x})$. Then:

Theorem V.5. *An operator, $\underline{\sigma}_p(T)$, which takes a flow table T and a predicate p, synthesizes the table T_p, and then takes the flow intersection of T and T_p is RA-consistent with selection.*

$$\underline{\sigma}_p(T) := T \underline{\wedge} T_P$$

Proof. We write:

$$\begin{aligned} \mathbf{x} \in T \underline{\wedge} T_P &\Leftrightarrow \mathbf{x} \in T \wedge \mathbf{x} \in T_p && \text{(Theorem V.4)} \\ &\Leftrightarrow \mathbf{x} \in T \wedge p(\mathbf{x}) && \text{(by the construction of } T_p\text{.)} \end{aligned}$$

T_p can be constructed for arbitrary boolean expression p whose terms are conditionals by: (1) placing p in disjunctive normal form, (2) building a flow table to represent each conditional, (3) joining each conjunct's tables by set intersection and then (4) joining each disjunct's tables by union.

VI. The History Table

Many common network management tasks require the generation of one or more packet histories (see §IV). Rather than requiring the programmer to manually synthesize packet histories, FlowDB provides a virtual table termed the history table, which conceptually lists every history in the network.

The history table is built on the concept of packet state. A packet's state consists of four components:

- *Physical Location.* The last port the packet traversed.
- *Logical Location.* The processing element the packet is about to enter. This can be a flow table or the network links.
- *Header fields.* The packet's header field values.
- *Metadata fields.* A packet in a switch is often assigned metadata fields to store intermediate processing results. These fields are stripped on exiting a switch.

A packet's state is thus the tuple $ps := (\texttt{ele}, \texttt{port}, \texttt{h}_\texttt{i}, \ldots)$ where `ele` is a logical location, `port` is a physical location, and each $\texttt{h}_\texttt{i}$ is a header or metadata field. If a metadata field is not written, it is set to `null`. *E.g.,* a packet in `myNet` with `dstip=9.0.0.1` entering port `swA.1` (Fig. 5) has state $ps_1 := (\texttt{ele} : \texttt{TA}, \texttt{port} : 1, \texttt{dstip} : \texttt{9.0.0.1})$.

History. A history is a sequence of packet states $ps_1\ ps_2\ \ldots\ ps_n$ describing the sequence of states a packet with initial state ps_1 is routed through. If a network has multiple histories starting with ps_1, ps_1 is multicast by the network. If a network has no histories starting with ps_1, ps_1 is an invalid initial state.

History Table. The history table lists the network's histories. If a network has packet state $ps := (\texttt{ele}, \texttt{port}, \texttt{h}_\texttt{i}, \ldots)$, the history table has attributes `ele(1)`, `port(1)`, $\texttt{h}_\texttt{i}$`(1)`, ... which store each history's initial state, `ele(2)`, `port(2)`, $\texttt{h}_\texttt{i}$`(2)`, ... which store each history's second state, etc. The history table conceptually unbounded number of attributes makes it differ from a standard flow table. The history table's match attributes are its initial state. The table `myHistory` (Fig. 3.) shows the first four packet states in `myNet`'s history table. Tuple $\mathbf{s_2}$ contains Fig. 5's packet history.

If a history has fewer than k states, each value in its kth state and onwards is `null`. The history relation also lists each history's last non-`null` state's number in the attribute `last`.

Materializing the History Table. The history table is materialized using the Flow Algebra. Since a programmer is rarely interested in the entire history relation, materialization is postponed to query execution when it is combined the user's query to minimize computation. Before describing materialization, however, we must describe the network view it acts on.

Logical Network. A network can be viewed as a directed graph of logical elements. Each element is a dataplane flow table or the network links. Each edge (ele_i, ele_j) indicates that ele_i can transmit a packet to ele_j for further processing. For example, `myNet`'s logical network (Fig. 6) contains three elements, `TA`, `TB` and `links`. Its four edges $(\texttt{TA}, \texttt{links})$, $(\texttt{TB}, \texttt{links})$, $(\texttt{links}, \texttt{TA})$, and $(\texttt{links}, \texttt{TB})$ indicate that `TA` and `TB` can transmit to `links` and vice versa.

A logical network's *ingress element* is an element that a packet entering the network can arrive at first. In `myNet`, `TA` and `TB` are logical elements. In a logical network, each element's flow table's map is viewed as a multifunction mapping a packet state to a set of packet states. For example, `TA` maps the packet state $(\texttt{TA}, 1, \texttt{9.0.0.1})$ to the set of states $\{(\texttt{links}, 2, \texttt{9.0.0.1})\}$.

Flow Table Normalization. A flow table is normalized if it explicitly represents a map from one packet state to another. If an network has state $(\texttt{ele}, \texttt{port}, \texttt{h}_\texttt{i}, \ldots)$, a normalized flow table has schema: `T` : `ele(1)`, `port(1)`, $\texttt{h}_\texttt{i}$ `(1)`, ... $\rightarrow$ `ele(2)`, `port(2)`, $\texttt{h}_\texttt{i}$`(2)`. To materialize the history table, each flow table in the logical network must be normalized.

Datapath Flow Table Normalization. A flow table `T` is not normalized if some packet state variable is missing from either its match attributes or action attributes. To normalize `T`, any missing match/action attribute is added to it and that attributes fields populated as shown in Fig. 7. Figure 8 (top) shows the relation `FA` after normalization.

Representing the network's links. History table materialization also requires the network's links topology to be represented

	last	ele(1)	port(1)	dstip(1)	ele(2)	port(2)	dstip(2)	ele(3)	port(3)	dstip(3)	ele(4)	port(4)	dstip(4)	Ordering
s_1	2	FA	x	9.0.0.0	links	1	=dstip(1)	null	null	null	null	null	null	$s_1 > s_2$
s_2	4	FA	x	9.0.0.x	links	2	=dstip(1)	FB	3	=dstip(1)	links	4	=dstip(1)	$s_3 > s_4$
s_3	4	FB	x	9.0.0.0	links	3	=dstip(1)	FA	2	=dstip(1)	links	1	=dstip(1)	
s_4	2	FB	x	9.0.0.x	links	4	=dstip(1)	null	null	null	null	null	null	

Fig. 3. The history table myHistory, summarizing myNet's behavior.

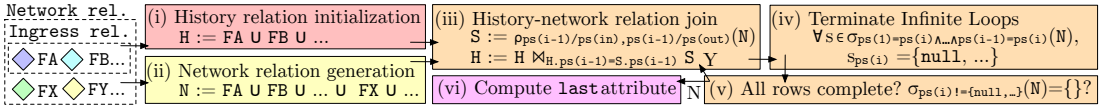


Fig. 4. The history table materialization workflow. The expressions under steps (iii)-(v) generate each history's ith step.

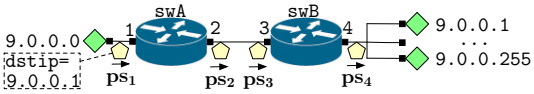


Fig. 5. A packet travelling through myNet.

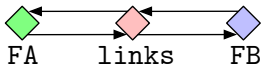


Fig. 6. The graph of myNet's logical network.

in a normalized flow table, links. This flow table can be constructed as follows. First, for each pair of ports p and q in the network connected by a link, the rule (links, p, xxxx, ..., ele_k, q, $=h_i(1)$, ...) is added to links (where ele_k is the first logical element a packet entering port q encounters.)

Next, the rule (links, p, xxxx, ..., null, null, ...) is added for each port p connected to a link leaving the network. For example, Fig. 8 (bottom) shows myNet's links relation.

History Table Materialization. Once the logical network's flow relations have been normalized, they can be transformed into the history relation following the workflow shown in Fig. 4.

(1) History Table Initialization. The history table, H, is initialized as a list of each history's first two steps by taking the union of each ingress element. For example, myHistory is initialized to FA ∪ FB.

(2) Network Table Initialization. The network table, N, contains a tuple (ps_{in}, ps_{out}) for each packet state, ps_{in}, that can enter a network element and each packet state, ps_{out}, that this element maps ps_{in} to. It is initialized by taking the union of each element's relation and then adding an all null tuple. For example, myNet's network relation is FA ∪ FB ∪ links ∪ {(null, null, null, ...)}.

(3) Joining the History and Network Tables . The join:

$$H \bowtie_{H.ps_2 = N.ps_2} \rho_{ps_2/ps_{in}, ps_3/ps_{out}}(N)$$

computes the history table's third step by joining each history (ps_1, ps_2) in the history table with each tuple (ps_2, ps_3) in the network table listing the packet state that the network maps ps_2 to. Note that if a packet in state ps_2 is forwarded out of the network by the links relation, ps_3 is set to null.

(4) Terminate Infinite Loops. The selection:

$$\sigma_{ps_1 = ps_2}(H)$$

finds each history with an infinite loop whose second cycle starts at ps_2 (loops whose second cycle starts at ps_3 will be discovered in a subsequent step.) Each such history has its value for ps_3 set to the null state.

(5) Check for incomplete histories. The selection:

$$\sigma_{ps_3 != \texttt{null}}(H)$$

ele(1)	port(1)	$h_1(1)$	ele(2)	port(2)	$h_2(2)$
F	x	x	links	=port(1)	$=h_1(1)$

Fig. 7. The values a flow table T's missing match/action attribute's fields are populated with during normalization.

FTA	ele(1)	port(1)	dstip(1)	ele(2)	port(2)	dstip(2)
s_1	FTA	x	9.0.0.0	links	1	=dstip(1)
s_2	FTA	x	9.0.0.x	links	2	=dstip(1)

links	ele(1)	port(1)	dstip(1)	ele(2)	port(2)	dstip(2)
s_1	links	1	x.x.x.x	null	null	null
s_2	links	2	x.x.x.x	FTB	3	=dstip(1)
s_3	links	3	x.x.x.x	FTA	2	=dstip(1)
s_4	links	4	x.x.x.x	null	null	null

Fig. 8. The normalized flow relations FTA and links.

lists every history whose final step is not the null tuple. These are the histories that have yet to exit the network, i.e., the incomplete histories. If there is an incomplete history, construction returns to step (3), which computes each history's fourth step, ps_4. Infinite loops which repeat at ps_3 are terminated in step (4), etc. Since the network relation contains a tuple mapping the null state to the null state, each history terminating at state ps_i will have each subsequent state ps_{i+1}, ps_{i+2}, etc. set to the null state.

(6) The last attribute. Each history is scanned and its last non-null state's number is added to it as the attribute last.

VII. FlowSQL

FlowSQL is a dialect of SQL built on top of the Flow Algebra. FlowSQL provides FlowDB with a declarative, familiar programming interface by implementing the subset of SQL listed in Fig. 9. FlowSQL augments this subset with indexed attributes and the ANY, ALL and TALLY clauses to allow the programmer to search the history table for histories matching complex patterns and compute these histories' properties.

Indexed Attributes. An indexed attribute, attr(n), references the packet state variable attr belonging to the nth step of the history table. For example, the index attribute dstip(2) references the dstip of the history table's second step. An index's value can be an integer or the special keyword, last, which references the last non-nil step in a history. In myHistory, for example, port(last) references port(2) in $\mathbf{r_1}$ but port(3) in $\mathbf{r_2}$.

FlowSQL also provides a special indexed attribute, step(n), which references each attribute in the nth step of the history table. For example,SELECT step(3) FROM myHistory and SELECT port(3), dstip(3) FROM myHistory are equivalent.

The reachability task, *"which packets are routed from port s to port d?"*, can be expressed using indexed attributes:

```
SELECT step(1) FROM history
WHERE port(1) = s AND port(last) = d
```

Keyword Category	Keywords
SELECT command	SELECT, FROM, WHERE, HAVING, GROUP BY
Set operation	EXCEPT, INTERSECT, UNION
Conditional	NOT, AND, OR, ANY, IN, IS NULL, EXISTS
Aggregate	AVG, COUNT, SUM
Misc.	CREATE TABLE, AS, VALUE

Fig. 9. FlowSQL Keywords.

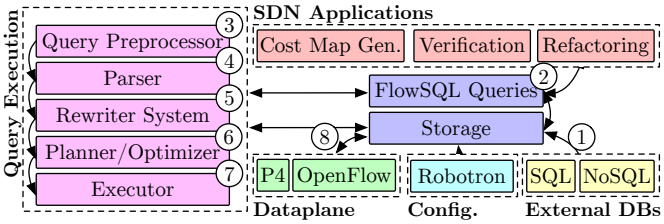


Fig. 10. FlowDB system diagram.

ANY/ALL Clauses. A programmer may want to identify whether the history table contains a history whose steps match a particular pattern. For example, the waypoint verification task, *"does any packet traveling from port* `s` *to port* `d` *not pass through port* `w`*?"*, requires identifying each history where `port(1) = s`, `port(last) = d`, and no `port(i)` is equal to `w`. Such queries are expressed using `ANY`/`ALL` clauses.

An `ANY`/`ALL` clause takes a set of index variables {`i`, `j`, ...} and a predicate whose indexed attributes use those variables as indices `p(i, j...)`. An `ANY` clause evaluates to true if any binding of index variables to integers satisfies `p`.

$$\texttt{ANY(i, j, ...)(p(i, j, ...))} := \exists\, i \in \mathbb{N}, \exists\, j \in \mathbb{N}, \ldots : \texttt{p(i, j, ...)}.$$

An `ALL` clause evaluates to true if each binding of index variables to integers satisfies `p`.

$$\texttt{ALL(i, j, ...)(p(i, j, ...))} := \forall\, i \in \mathbb{N}, \forall\, j \in \mathbb{N}, \ldots : \texttt{p(i, j, ...)}.$$

For example, the waypointing task above can be expressed:

```
SELECT step(1) FROM history
WHERE port(1) = s AND port(last) = d
AND ALL(i)(port(i) != w)
```

TALLY Clauses. A programmer may also want to know how many times a particular pattern occurs in a history. For example, verifying that no packet routed from port `s` to port `d` makes more than `k` hops requires counting the number of times each history passes through the network links. Such queries are expressed using `TALLY` clauses. A `TALLY` clause `TALLY(i, j, ...)(p(i, j...))` returns the number of bindings to the indices `i`, `j`,... that satisfy `p`.

$$\texttt{TALLY(i, j...)(p(i, j...))} := \sum_{i=1}^{\infty}\sum_{j=1}^{\infty} \texttt{p(i, j...)}$$

VIII. Implementation

The flow algebra's implementation, termed FlowDB, is shown in Fig. 10. FlowDB was prototyped as an extension to the PostgreSQL database. Upon initialization, FlowDB is populated (1) by extracting routing tables from each switch's dataplane (e.g., Openflow [38]/P4 [39] tables), network topology/configuration information from the network managment system (e.g. Robotron [1]) and traditional relations from external databases, converting each table into a simplified flow table (§IV, §V) and storing it. An atomic predicate [3], [4] based index is created for each flow table to accelerate joins.

Network applications can view, analyze and manipulate these tables using FlowSQL (§VII) queries (2). Upon receiving a FlowSQL query, FlowDB's query preprocessor (3) scans it and, where necessary, converts its WHERE and HAVING clauses into joins (§V). The rewritten query is converted into a parse tree by the PostgreSQL parser (4), which is provided with the additional keywords FlowSQL introduces. The rewrite system (5) identifies any part of the query tree referencing a view and alters it to reference the tables in the view's definition. Care is taken when building the history table (§VI).

The planner/optimizer (6) then takes the query tree and generates a query plan. The planner/optimizer is augmented with the definitions of FlowDB's flow table operators and functions which estimate their execution cost. Finally, the executor (7) executes this query plan. The results may be written back (8) to a switch's dataplane.

IX. Evaluation

FlowDB's evaluation answers the following three questions:

(1) Can FlowDB run a range of network management tasks?
(2) Do these tasks run reasonably quickly on real networks?
(3) How does FlowDB's performance compare to other tools?

We find that: (1) FlowDB can run a wide range of network management tasks, (2) that these tasks run in a reasonable time frame on two large real-world networks, and (3) FlowDB outperforms two state-of-the-art verification engines, Header Space Analysis [8] and NoD [9] on network debugging and verification tasks by mode than $55\times$ and $100\times$.

Benchmark Tasks. FlowDB was run on five classes of tasks:

(1) *Cost Map Generation.* Generating ALTO [27], Flow Director [28] and simple hop count cost maps.
(2) *Verification.* Reachability, finite/infinite loop detection, black hole detection, waypoint verification, multipath consistency, equal cost multipath, isolation.
(3) *Dataplane refactoring.* Decomposing a flow table into 2NF and 3NF forms [35].
(4) *Dataplane Composition.* Composing two tables sequentially, in parallel and with one table overriding the other [6].
(5) *Network Policy Managment.* Certifying that a network maintains policies practiced by a real large internet company.

Benchmark Networks. FlowDB is evaluated on a publicly available snapshot of Stanford's campus network [7] (Fig. 11. (left)) and a parametrizable model of a large internet company's data center switching fabric (Fig. 11. (right)). Stanford's campus network contains 757,000+ forwarding entries. The switching fabric is built on a real three layer topology taken from a Facebook data center and contains 10K+ switches, 100K+ links, and 150 million+ forwarding entries.

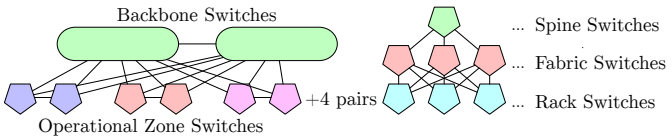


Fig. 11. (left) Stanford's network topology. (right) A segment of a large internet company's switching fabric topology.

Management Task	Time (s)	Management Task	Time (s)
Cost Map Generation		**Dataplane Refactoring**	
ALTO	3010	2NF Decomposition	0.3
FlowDirector	3870	3NF Decomposition	0.3
Hop Count	1671	**Dataplane Composition**	
ALTO Update	9.5	Sequential Composition	1.8
FlowDirector Update	10.7	Parallel Composition	1.4
Verification		Override Composition	1.4
Reachability	7.9	**Network Policy Checking**	
Finite Loop Detection	40.8	Policy 1	2333
Infinite Loop Detection	52.3	Policy 2	3589
Blackhole Detection	27.3	Policy 3	3150
Waypoint Verification	45.8	Policy 1 Update	5.7
Multipath Consistency	69.2	Policy 2 Update	9.6
Equal Cost Multipath	48.2	Policy 3 Update	25.6
Isolation	1.2		

Fig. 12. Runtimes of selected network managment tasks on a datacenter switching fabric with FlowDB.

Benchmark Verification Engines. FlowDB is benchmarked against two publicly available state-of-the-art verification engines: HSA's HasselC implementation [7] and NoD [9].

Hardware. All tests were run on an Intel Core i7, 2.20GHz CPU running Ubuntu 18.04 LTS with with 8GB of RAM.

A. Running a Range of Network Management Tasks

Fig. 12 shows the wide range of network management tasks FlowSQL can express. These tasks involve dataplane analysis, manipulation, and query both dataplane and non-dataplane data. Fig. 13 shows the maximum number of lines of code required to express a task from each category of tasks (e.g., each of the verification tasks in Fig. 12 took ≤ 7 lines of code to express.) While code length weakly correlates with readability, it shows that FlowSQL expresses tasks succinctly.

B. Network Management Task Runtimes

Fig. 12 shows the median time FlowDB took to compute each benchmark task on the Facebook data center switch fabric over 10 runs. Unsurprisingly, each task's runtime is determined by the number of joins it requires. Each dataplane refactoring (composition) task, which required no (one large) join, completed just in 0.3s (1.4s to 1.8s). Reachability, which finds a history between a pair of source and destination ports, required ~ 8 joins and took 7.9s, whilst queries like finite loop detection, which generate every history starting at a given port, required ~ 50 joins and took 40 to 70s. Runtime does not linearly increase with join number, since more joins provide greater opportunity to optimize join ordering and reuse results.

Cost map generation and network-wide policy checking compute a substantial fraction of the network's histories and took 30 to 60min. Fortunately, however, once a cost map is generated (network policy is certified), updating the policy (certificate) against network churn is rapid (9.5-25.6s) since network churn only affects a small number of rules. Each verification query (other than isolation) referenced the history table. Since the history table is lazily generated, however, only a small part of it is built for each task, decreasing runtime.

Management Category	Line #	Managment Category	Line #
Cost Map Generation	12	Dataplane Composition	2
Verification	7	Network Policy Checking	19
Dataplane Refactoring	2		

Fig. 13. The maximum number of FlowSQL lines to express a task from each category of benchmark tasks.

System	Time (s)	System	Time (s)
HSA (Hassel-C)	190.1	FlowDB (no index)	18.3
NoD (Batfish)	421.3	FlowDB (no opt.)	145.4
FlowDB	3.7		

Fig. 14. Benchmarking FlowDB against state-of-the-art verification tools.

C. Benchmarking FlowDB

Fig. 14 lists the median time that FlowDB, HSA and NoD require to certify a real network policy (policy 1 in [37]) on Stanford's network over 20 trials. (Note that since HSA/HoD cannot query non-network tables, a Python scaffolding was set up to transform the policy certification task into a series of HSA/NoD verification tasks. The runtime of this Python scaffolding is not included in Fig. 14.) FlowDB outperformed HSA (NoD) by $\sim 55\times$ ($\sim 100\times$). We hypothesized that FlowDB's performance increase stemmed from its atomic predicate based flow table index, which allowed it to join network tables when computing packet histories more efficiently than HSA/NoD, and its use of the PostgreSQL optimizer, which allowed it to follow a more efficient join ordering than HSA/NoD.

To test this hypothesis, we ran FlowDB on this task two more times, once without using our efficient index-based join and once without our index-based join and enforcing a strict left-to-right join ordering. Fig. 14 shows that indexing (efficient join ordering) provided a $\sim 6\times$ ($\sim 8\times$) speed up. Without these two optimizations, FlowDB performed similarly to HSA/NoD (the remaining performance improvement is perhaps attributable to PostgreSQL's efficient engineering.)

X. Conclusion

Flow Algebra provides a generic and rigorous formal framework for a great diversity of network management tasks. By inheriting the relational algebra's algebraic properties, the flow algebra allows network managers to leverage the significant progress database researchers have made in query optimization, allowing FlowDB to scale to very large networks. Moreover, by implementing cutting edge domain specific optimizations such as atomic predicate "under the hood", FlowDB allows operators to benefit from these optimizations without having to understand their complexities.

Acknowledgement

We would like to acknowledge Facebook for funding and supporting this research, and Qiao Xiang, Shenshen Chen and Kai Gao for their advice, support and kindness.

REFERENCES

[1] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng, "Robotron: Top-down network management at facebook scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," 2014.

[3] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, 2015.

[4] ——, "Scalable verification of networks with packet transformers using atomic predicates," *IEEE/ACM Transactions on Networking*, 2017.

[5] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "Apkeep: Realtime verification for real networks," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/zhang-peng

[6] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *NSDI15*. Oakland, CA: USENIX Association, May 2015.

[7] P. Kazemian, *Hassel C*, Accessed August 14, 2020. [Online]. Available: https://bitbucket.org/peymank/hassel-public/wiki/Home

[8] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *NSDI*, 2013.

[9] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[10] S. Babeni, *Most Popular Databases in 2020: Here's How They Stack Up*, January 24, 2020 (Accessed August 14, 2020). [Online]. Available: https://ormuco.com/blog/most-popular-databases

[11] S. Overflow, *Developer Survey Results 2019*, 2019 (Accessed June 12, 2020). [Online]. Available: https://insights.stackoverflow.com/survey/2019

[12] D. Maier, K. T. Tekle, M. Kifer, and D. Warren, *Datalog: concepts, history, and outlook*. ACM Books, 09 2018.

[13] S. Prabhu, K.-Y. Chou, A. Kheradmand, P. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," 2019.

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017.

[15] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015.

[16] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, 2016.

[17] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016.

[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013.

[19] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018.

[20] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018.

[21] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in p4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018.

[22] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016.

[23] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[24] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "Netsmc: A custom symbolic model checker for stateful network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020.

[25] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Feb. 2020.

[26] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. G. Liu, and A. Silberschatz, "P4p:provider portal for applications," *Acm Sigcomm Aug*, 2008.

[27] R. Alimi, Y. Yang, and R. Penno, "Application-layer traffic optimization (ALTO) protocol," *IETF RFC*, 2014.

[28] E. Pujol, I. Poese, J. Zerwas, G. Smaragdakis, and A. Feldmann, "Steering hyper-giants' traffic at scale," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019.

[29] A. Wang, X. Mei, J. Croft, M. Caesar, and B. Godfrey, "Ravel: A database-defined network," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: Association for Computing Machinery, 2016.

[30] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: Language, execution and optimization," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2006.

[31] M. Abadi and B. T. Loo, "Towards a declarative language and system for secure networking," in *Proceedings of the 3rd USENIX International Workshop on Networking Meets Databases*, ser. NETB'07. USA: USENIX Association, 2007.

[32] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018.

[33] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, 2003.

[34] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative network monitoring with netqre," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017.

[35] F. Németh, M. Chiesa, and G. Rétvári, "Normal forms for match-action programs," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019.

[36] *switch.p4*, 2016. [Online]. Available: https://github.com/p4lang/switch

[37] *Flow Algebra Technical Report*. [Online]. Available: https://drive.google.com/drive/folders/15XS3ZEVZY5GM6Usbav0mFVJfPiOvc8Re?usp=sharing

[38] *OpenFlow Spec.*, 2014. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.4.1.pdf

[39] *P4-16 Language Specification*, 2016. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html