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

Christopher Leet
Yale University
c 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
Novel Contributions. This paper’s novel contributions are:

(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 can directly 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.

Benefits. The flow algebra provides several key benefits:

The flow algebra provides several key benefits:

50 FlowDB outperformed these tools by network debugging and cost map generation and found that Analysis [8] and Network Optimized Datalog [9] to perform extended two network verification engines, Header Space refactoring and network debugging in

50 management tasks, including cost map generation, dataplane fabric operated by a large internet company containing over 10K+ switches and 100 million+ flow table rules. Our evaluation shows 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 × more.

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.


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.
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: \( T : \text{dmac, vlan} \rightarrow \text{mod_vlan, out.} \) Suppose that T has the functional dependency \( \text{mod_vlan} \rightarrow \text{out} \) (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 \( T_0 : \text{dmac, vlan} \rightarrow \text{mod_vlan} \) and \( T_1 : \text{mod_vlan} \rightarrow \text{out} \) 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:

\[
\begin{align*}
\text{TB}' &= \sigma_{\text{fwd}(4)}\text{actions}(\text{TB}) \\
\text{TA}' &= \sigma_{\text{fwd}(2)}\text{actions}\land 9.0.0.0 \leq \text{dstip} \leq 9.0.0.3(\text{TA}) \\
\text{out} &\leftarrow \text{TA}' \land_{\text{TA}.\text{dstip}=\text{TB}'.\text{dstip}} \text{TB}'
\end{align*}
\]

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 string of sets 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 \( r_2 \) 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:

\[
(1, 9.0.0.x, \text{fwd}(2))
\]

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. 9.0.0.0.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 \( r_4 \) when it should yield the rules \((2, 9.0.0.0, \text{fwd}(1))\) to \((1, 9.0.0.3, \text{fwd}(1))\). Selection must split a rule if some values it represents satisfy its predicate but not others.

---

1. The network myNet.
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 \( op \). Collectively, these flow operators make up the flow algebra.

We formalize the preceding ideas by introducing the notion 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 \( op \) is relational algebra (RA) consistent to a relational algebra operator \( op \) iff for any flow table \( T \) (flow tables, \( T_1 \) and \( T_2 \)):

\[
[op(T)] = [op([T])]
\]

\[
([T_1 \ op \ T_2] = [T_1] \ op \ [T_2])
\]

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

Definition IV.2. The flow algebra operator \( op \) 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. If the relational algebra operator \( op \) is commutative (associative), the corresponding flow algebra operator \( op \) is commutative (associative).

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

\[
[T_1 \ op \ T_2] = [T_1] \ op \ [T_2] \quad \text{(RA-equiv.)}
\]

\[
[T_2 \ op \ T_1] \quad \text{(op commutes)}
\]

\[
[T_2 \ op \ T_1] \quad \text{(RA-equiv.)}
\]

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

\[
[T_1 \ op \ (T_2 \ op \ T_3)] = [T_1] \ op \ [T_2 \ op \ T_3] \quad \text{(RA-equiv.)}
\]

\[
=[T_1 \ op \ ([T_2] \ op \ [T_3])] \quad \text{(RA-equiv.)}
\]

\[
=([T_1] \ op \ [T_2]) \ op \ [T_3] \quad \text{(op associates)}
\]

\[
=[T_1 \ op \ T_2] \ op \ [T_3] \quad \text{(RA-equiv.)}
\]

\[
=([T_1 \ op \ T_2] \ op \ [T_3]) \quad \text{(RA-equiv.)}
\]

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, \( op \), 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.

\[
T_1 \sim T_2 := [T_1] = [T_2].
\]

Lemma IV.2. If flow tables \( T_1 \) and \( T'_1 \) and \( T_2 \) and \( T'_2 \) are equivalent, flow tables \( T_1 \ op \ T'2 \) and \( T'_1 \ op \ T'_2 \) are equivalent if \( op \) is RA-consistent.

\[
T_1 \sim T'_1 \land T_2 \sim T'_2 \Rightarrow T_1 \ op \ T_2 \sim T'_1 \ op \ T'_2.
\]
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 re-
solved 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. 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 x be a vector of values
in the domain of T’s match attributes. Let y be a vector of
values in the domain of T’s action attributes. For instance, if
T was TA, x might be (9.0.0.0) and y might be (2).

We say that T represents the tuple xy if xy ∈ T, denoted
x ∈ 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, s, is a a tuple (pri, m, a), where pri is the
flow rule’s priority, m is a vector of match attribute values
and a is a vector of action attribute values.

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

For example, rule s2 in TAS matches (9.0.0.0, 2).

Definition V.2. The match set of a tuple of values xy in a flow
table T, denoted T(xy) is the set of flow table rules whose
match fields m match on the tuple’s match fields x.

T(xy) = \{(pri, m, a) : m matches x ∧ (pri, m, a) ∈ T\}.

For example TAS((9.0.0.0, 2)) = \{s1, s2\}.

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(xy), be denoted max(T(xy)).

Definition V.3. A flow table T represents the tuple xy if the
highest priority rule in xy’s match set T(xy) represents xy.

xy ∈ T := T(xy) \neq \emptyset ∧ xy ∈ max(T(xy)).

For example, TAS does not represent (9.0.0.0, 2) since
s1, the highest priority rule in TAS(xy) does not represent it.

B. Efficient RA-Compatible Cartesian Product

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

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

Theorem V.2. An operator, $\times$, 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.

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

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

Lemma V.1. The rule $(p_A, m_A, a_A)$ is in $A(xy)$ and the rule $(p_B, m_B, a_B)$ is in $B(xy)$ if and only if the rule $(p_A + p_B, m_A m_B, a_A a_B)$ is in $(A \times B)(xyuv)$.

Proof. We write:
$(p_A, m_A, a_A) \in A(xy) \land (p_B, m_B, a_B) \in B(xy)$
$\iff x \text{ matches } m_A \land u \text{ matches } m_B$
$\iff xu \text{ matches } m_A m_B$
$\iff (p_A + p_B, m_A m_B, a_A a_B) \in (A \times B)(xyuv).$

Lemma V.2. The rule $(p_A, m_A, a_A) = \max(A(xy))$ and the rule $(p_B, m_B, a_B) = \max(B(xy))$ if and only if the rule $(p_A + p_B, m_A m_B, a_A a_B) = \max((A \times B)(xyuv))$.

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

Proof $(\Rightarrow)$. Suppose, again for the sake of contradiction, that $\max(A(xy)) = (p_A, m_A, a_A)$ and $\max(B(xyuv)) = (p_B, m_B, a_B)$ but $\max((A \times B)(xyuv)) \neq (p_A + p_B, m_A m_B, a_A a_B)$. Let $\max((A \times B)(xyuv)) = (p'_A + p_B, m_A m_B, a_A a_B)$.

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

An operator, \(\text{op}\), whose flow intersection of a predicate \(p\) and a predicate \(x\) outputs a flow table \(\text{op}(T,p)\) is RA-consistent if and only if:

\[
T \cup \exists p \alpha \iff T \cup \exists x \gamma
\]

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

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 state is thus the tuple \(A:= (ele, \text{port}, h_1, \ldots)\) where \(ele\) is a logical location, \(\text{port}\) is a physical location, and each \(h_i\) 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 \(ps_1:= (ele:TA.\text{port}:1,dstip:9.0.0.1)\).

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 \(ps:= (ele, \text{port}, h_1, \ldots)\), the history table has attributes \(ele(1), \text{port}(1), h_1(1), \ldots\) which store each history’s initial state, \(ele(2), \text{port}(2), h_1(2), \ldots\) 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 \(\text{myHistory}\) (Fig. 3) shows the first four packet states in \(\text{myNet}\)’s history table. Tuple \(s_2\) contains Fig. 5’s packet history.

If a history has fewer than \(k\) states, each value in its \(k\)th 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. 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 \((\text{TA}, \text{links}), (\text{TB}, \text{links}), (\text{links}, \text{TA})\), and \((\text{links}, \text{TB})\) 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 \((\text{TA}, 1.9.0.0.1)\) to the set of states \{\{\text{links,2.9.0.0.1}\}\}.

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 \((ele, \text{port}, h_1, \ldots)\), a normalized flow table has schema: \(T: ele(1), \text{port}(1), h_1(1), \ldots \rightarrow ele(2), \text{port}(2), h_1(2)\). 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...
in a normalized flow table, \( \text{links} \). 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 \((\text{links}, p, xxxx, \ldots, \text{ele}_k, q, \ldots)\) is added to \( \text{links} \) (where \( \text{ele}_k \) is the first logical element a packet entering port \( q \) encounters.)

Next, the rule \((\text{links}, p, xxxx, \ldots, \text{null}, \text{null}, \ldots)\) is added for each port \( p \) connected to a link leaving the network. For example, Fig. 8 (bottom) shows myNet’s \( \text{links} \) 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.

**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 \( \text{FA} \cup \text{FB} \).

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

**Joining the History and Network Tables.** The join:

\[
H \bowtie_{H.p_{\text{in}}=N.p_{\text{in}}} N.p_{\text{out}2}(N)
\]

computes the history table’s third step by joining each history \((p_{\text{in}}, p_{\text{out}})\) in the network table with each tuple \((p_{\text{in}}, p_{\text{out}})\) in the network table listing the packet that the network maps \( p_{\text{in}} \) to. Note that if a packet in state \( p_{\text{out}} \) is forwarded out of the network by the \( \text{links} \) relation, \( p_{\text{in}} \) is set to \( \text{null} \).

**Terminate Infinite Loops.** The selection:

\[
\sigma_{p_{\text{in}}=p_{\text{out}}}(H)
\]

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

**Check for incomplete histories.** The selection:

\[
\sigma_{p_{\text{in}}=\text{null}}(H)
\]

lists every history whose final step is not the \( \text{null} \) 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, \( p_{\text{s4}} \). Infinite loops which repeat at \( p_{\text{s3}} \) are terminated in step (4), etc. Since the network relation contains a tuple mapping the \( \text{null} \) state to the \( \text{null} \) state, each history terminating at state \( p_{\text{s4}} \) will have each subsequent state \( p_{\text{s4}+1}, p_{\text{s4}+2}, \ldots \) set to the \( \text{null} \) state.

**The last attribute.** Each history is scanned and its last non-\( \text{null} \) state’s number is added to it as the attribute \( \text{last} \).

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

FLOWSQL also provides a special indexed attribute, \( \text{step}(n) \), which references each attribute in the \( n \)th step of the history table. For example, \( \text{SELECT} \text{step}(3) \text{FROM myhistory and SELECT step}(3), \text{dstip}(3) \text{FROM myhistory are equivalent.} \)

The reachability task, “which packets are routed from port \( s \) to port \( d \)?”, can be expressed using indexed attributes:

\[
\text{SELECT step}(1) \text{FROM history WHERE port}(1) = s \text{ AND port}(\text{last}) = d
\]
**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 \( \{i, j, \ldots \} \) and a predicate whose indexed attributes use those variables as indices \( p(i, j, \ldots) \). An ANY clause evaluates to true if any binding of index variables to integers satisfies \( p \).

\[
\text{ANY}(i, j, \ldots) (p(i, j, \ldots)) := \exists i, j, \ldots : p(i, j, \ldots).
\]

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

\[
\text{ALL}(i, j, \ldots) (p(i, j, \ldots)) := \forall i, j, \ldots : p(i, j, \ldots).
\]

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 \( \text{TALLY}(i, j, \ldots) (p(i, j, \ldots)) \) returns the number of bindings to the indices \( i, j, \ldots \) that satisfy \( p \).

\[
\text{TALLY}(i, j, \ldots) (p(i, j, \ldots)) := \sum_{i=1}^{\infty} \sum_{j=1}^{\infty} p(i, j, \ldots)
\]

**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 management system (e.g., Robotron [1]) and traditional relations from external databases, converting each table into a simplified flow table ($\langle IV, \langle V \rangle$) 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 ($\langle VII \rangle$) 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 ($\langle V \rangle$). 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 ($\langle VI \rangle$).

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 more 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 Management.** 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 threelayer 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.

<table>
<thead>
<tr>
<th>Management Task</th>
<th>Time [s]</th>
<th>Management Task</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Map Generation</td>
<td></td>
<td>Dataplane Refactoring</td>
<td></td>
</tr>
<tr>
<td>ALTO</td>
<td>3010</td>
<td>2NF Decomposition</td>
<td>0.3</td>
</tr>
<tr>
<td>FlowDirector</td>
<td>3870</td>
<td>3NF Decomposition</td>
<td>0.3</td>
</tr>
<tr>
<td>Hop Count</td>
<td>1671</td>
<td>Dataplane Composition</td>
<td></td>
</tr>
<tr>
<td>ALTO Update</td>
<td>9.5</td>
<td>Sequential Composition</td>
<td>1.8</td>
</tr>
<tr>
<td>FlowDirector Update</td>
<td>10.7</td>
<td>Parallel Composition</td>
<td>1.4</td>
</tr>
<tr>
<td>Verification</td>
<td></td>
<td>Override Composition</td>
<td>1.4</td>
</tr>
<tr>
<td>Reachability</td>
<td>7.9</td>
<td>Network Policy Checking</td>
<td></td>
</tr>
<tr>
<td>Finite Loop Detection</td>
<td>40.8</td>
<td>Policy 1</td>
<td>2333</td>
</tr>
<tr>
<td>Infinite Loop Detection</td>
<td>52.3</td>
<td>Policy 2</td>
<td>3589</td>
</tr>
<tr>
<td>Blackhole Detection</td>
<td>27.3</td>
<td>Policy 3</td>
<td>3150</td>
</tr>
<tr>
<td>Waypoint Verification</td>
<td>45.8</td>
<td>Policy 1 Update</td>
<td>5.7</td>
</tr>
<tr>
<td>Multipath Consistency</td>
<td>69.2</td>
<td>Policy 2 Update</td>
<td>9.6</td>
</tr>
<tr>
<td>Equal Cost Multipath</td>
<td>48.2</td>
<td>Policy 3 Update</td>
<td>25.0</td>
</tr>
<tr>
<td>Isolation</td>
<td>1.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 12. Runtimes of selected network management 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.

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/NoD 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 ~ 55× (~ 100×). 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 ~ 6× (∼ 8×) 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.
Most Popular Databases in 2020: Here’s How They Stack...


