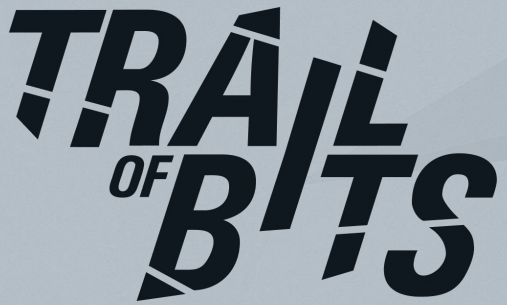


Dr. Lojekyll

The Mr. Hyde of Datalog Engines

Peter Goodman
Sonya Schriner
Eric Kilmer

Agenda

- **What is Datalog?**
- **Why create another Datalog engine?**
- **What does Dr. Lojekyll code look like?**
- **What design decisions did we make?**
- **How does Dr. Lojekyll work?**
- **Mini disassembler demonstration**

What is Datalog?

Relational, rules-based logical inference

```
tc(From, To) : tc(From, Hop), tc(Hop, To).

tc(From, To) : edge(From, To).
```

- **Canonical “transitive closure” example, although doubly recursive**
- **The transitive closure `tc` of a `From` node is every `To` that is reachable through zero or more `Hop` nodes**

Relational, rules-based logical inference

```
tc(From, To) : tc(From, Hop), tc(Hop, To).
tc(From, To) : edge(From, To).
```

- **Program composed of one or more clauses, each ending with a period**
- **Each clause is an inference rule, telling us how to produce new facts from existing facts**

Relational, rules-based logical inference

```
tc(From, To) : tc(From, Hop), tc(Hop, To).
tc(From, To) : edge(From, To).
```

- **Each clause contains a head and a body, separated by a colon**

Relational, rules-based logical inference

```
tc(From, To) : tc(From, Hop), tc(Hop, To).
tc(From, To) : edge(From, To).
```

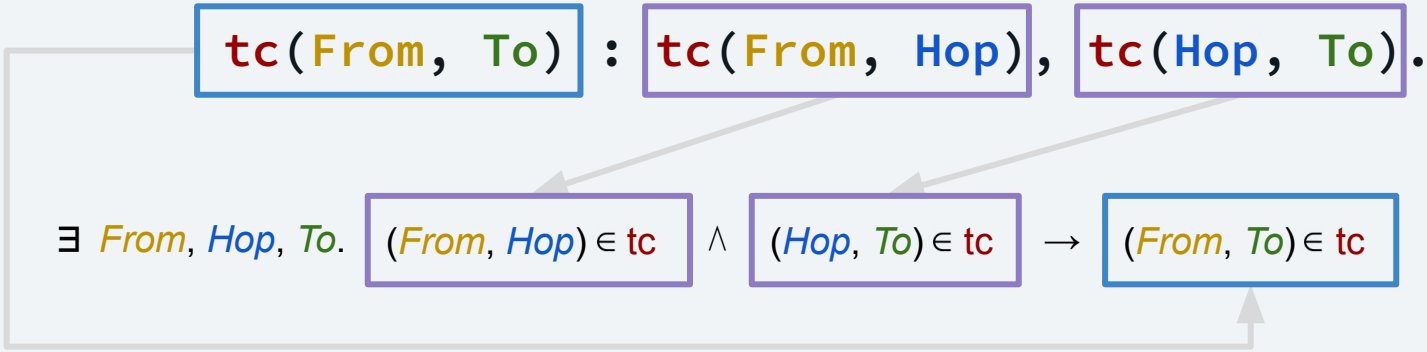
- **Clause bodies contain a list of predicates, separated by commas**
- **Interpreted as a list of conjuncts, i.e. , (comma) is logical AND**

Relational, rules-based logical inference

```
tc(From, To) : tc(From, Hop), tc(Hop, To).
tc(From, To) : edge(From, To).
```

- **Variables introduce constraints between predicates**
- **Sideways information passing style**
  - Top-down interpretation: First use of a variable binds it to a value
  - Bottom-up interpretation: Uses by more than one predicate induce a relational JOIN

Relational, rules-based logical inference



- **If the clause body can be satisfied, then the clause head is true**
- **Natural bottom-up semantics**

Why create another Datalog engine?

Prior experience with distributed systems…

- **DARPA Cyber Grand Challenge (CGC)**
  - Goal: Automatically find and fix exploitable bugs in a number of Linux-like binaries
  - Outcome: Cyberdyne
    - Microservices coordinating over a shared message bus
    - Used Redis as a distributed database and message passing system
- **Two key issues faced: change notification and orchestration**
  - Individual microservices implemented local caches
    - Symptom: Got out of date with what was in Redis
    - Problem: No way for services to be notified of changes to data of interest
  - Services re-implemented pipeline blocking logic
    - Symptom: Do *C* when *A* and *B* have been received or become available
    - Problem: Data and logic are often related, but not co-located

… led to idea of Datalog-based orchestration

- **DARPA Assured Micro-Patching (AMP)**
  - Problem: Legacy software is largely good (correct by process), but suffers some bugs
    - Need to minimally patch bugs in legacy binaries
    - Re-compilation might not be possible, or might not pass testing & evaluation process
  - Goal: Improve productivity of existing operators, i.e. **there is a human in the loop**
  - Requirements:
    - Decompile machine code to source code
    - Integrate source patch from newer version of software into old binary
- **Datalog to orchestrate a distributed system?**
  - Synchronization on messages can be expressed as a data dependency
  - Orchestration decisions should be made made off of most consistent view of the database
    - Data and logic now co-located

Ideally, we want Datalog servers

- **Want to support human-in-the-loop systems**
- **Most Datalog engines are batch systems, i.e. take their input all at once**
  - bddbddb, Soufflé
- **Some can support incrementality, but few support differential updates**
  - Push Method
- **Evolving knowledge base: what was once true might not always be true**
  - Should be able to retract data, data proven true should be able to be unproven
  - differential-datalog, IncA
- **Datalog servers are a bit like database servers**
  - Knowledge base is streamed to the server over time, not all-at-once
  - Responds to queries using materialized views, results may change over time
  - Changes to relations of interest are published to concerned parties
    - Published messages can be used to orchestrate distributed systems

Datalog engines should be...

- **Incremental**
  - A user should be able to introduce new axioms via arbitrary system interaction
  - When the inputs change, the results are “minimally” recomputed
  - Easiest method of implementing streaming input support is via incremental updates
- **Differential**
  - Users should be able to “undo” past things, i.e. remove axioms
  - Introduction of new, or removal of old axioms can trigger removal of prior inferences
    - E.g. negation (test for absence) previously satisfied, new data falsifies negation
- **Sympathetic**
  - Enable the programmer to communicate extralogical information to improve codegen
  - Enable the programmer to break the normal rules (e.g. stratified negation) because they have specialized domain knowledge that the compiler does not
  - Integrate into existing codebases (e.g. using “foreign” target code types in Datalog)

What does Dr. Lojekyll code look like?

Top-level syntax elements

- **Module**
  - A given Dr. Lojekyll file
- **Import**
  - Of another module
- **Inlined target-specific code**
  - Prologue
  - Epilogue
- **Types**
  - Built-in types
  - Foreign types
  - Enumeration types
- **Declarations**
  - Exports
  - Locals
  - Queries
  - Messages
  - Functors
  - Named constants
- **Clauses**
  - Head
  - Body

Example program: transitive closure

```
#message edge(u32 From, u32 To).

#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).

tc(From, To) : tc(From, Hop)
             , tc(Hop, To).
```


Example program: transitive closure

```
#message edge(u32 From, u32 To).

#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).

tc(From, To) : tc(From, Hop)
             , tc(Hop, To).
```

edge **relation declared as a message and used in a clause body means that it is received by the datalog engine**

Example program: transitive closure

```
#message edge(u32 From, u32 To).

#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).

tc(From, To) : tc(From, Hop)
             , tc(Hop, To).
```

`tc` **relation declared as a query means that it acts as an externally visible materialized view, accessible via a function call or RPC-like interface**

Example program: transitive closure

```
#message edge(u32 From, u32 To).

#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).

tc(From, To) : tc(From, Hop)
             , tc(Hop, To).
```

`free`-attributed parameters are the outputs of the query (e.g. columns in a `SELECT` clause)

`bound`-attributed parameters are the inputs to the query (e.g. columns in a `WHERE` clause)

What design decisions did we make?

Mechanical sympathy: Functor ranges

- **Functors are functions defined in the target code (C++ or Python) that can be called by Datalog code**
  - Functors parameters have binding attributes (`bound` for inputs, `free` for outputs)
  - Combination of all `free`-attributed parameters form a single tuple-structured output
  - Want to know how many outputs tuples will a functor produce
  - Want to support functors that act as accept/reject predicates, i.e. no `free` parameters
- `@range(…)` **pragma helps codegen output better code for your functors**
  - `@range(?)`: Zero-or-one outputs
  - `@range(.)`: Exactly one output
  - `@range(*)`: Zero-or-more outputs
  - `@range(+)`: One-or-more outputs

Mechanical sympathy: Referential transparency

- **Ideally, want to operate on values in the problem domain**
  - Target language types related to problem domain can be bound to Dr. Lojekyll types with `#foreign` type declarations, e.g. `#foreign YOLO ```c++ std::shared_ptr<YOLO>```.`
  - Problem: Is comparison for equality preserved after making copies of foreign-typed values?
- **Codegen conservatively assumes that copying a foreign-typed value does not produce an identical value**
  - Solution: Ensure referential transparency via interning
  - Problem: Interning is expensive, as it requires a hash set-based deduplication
- `@transparent` **pragma on a foreign type tells codegen that it does not need to intern values in order to maintain referential transparency**

Mechanical sympathy: Clause body ordering (1)

- **This clause doesn't execute in the order that you might expect**

```
function(ObjId, Arch, OS, EA)
    : json_spec(Arch, OS, Spec)
    , json_read_object_key(Spec, "functions", FuncList)
    , json_read_list_entry(FuncList, _, FuncInfo)
    , json_read_object_key(FuncInfo, "address", IntAddress)
    , json_read_address(IntAddress, EA)
    , instruction(ObjId, InstArch, EA)
    , object_file(ObjId, _ObjArch, OS, _ObjLoadEA, _ObjPath)
    , valid_arch_transition(Arch, InstArch).
```

Mechanical sympathy: Clause body ordering (2)

- **This clause uses the `@barrier` pragma to execute in the expected order**

```
function(ObjId, Arch, OS, EA)
    : json_spec(Arch, OS, Spec)
    , json_read_object_key(Spec, "functions", FuncList)
    , json_read_list_entry(FuncList, _, FuncInfo)
    , json_read_object_key(FuncInfo, "address", IntAddress)
    , json_read_address(IntAddress, EA)
    , @barrier
    , instruction(ObjId, InstArch, EA)
    , object_file(ObjId, _ObjArch, OS, _ObjLoadEA, _ObjPath)
    , valid_arch_transition(Arch, InstArch).
```

Footgun holster: Proper variable usage

- **Datalog requires clause head variables to be “range restricted”**
  - Every variable in a clause head must be used at least once in a clause body
  - Ensures that a value is bound to each parameter of a clause head
- **Variables in clause bodies do not need to be range restricted**
  - Typos (e.g. Hop vs Hopp) can lead to weaker conditions being tested
- **Dr. Lojekyll requires that every named variable be used at least twice**
  - Or prefix names with _ to mark them as “semantically labelled anonymous variables”

```
tc.dr:4:25 error: Named variable 'Hop' is only used once; you should use either '_' or prefix the name with an '_' to explicitly mark it as anonymous
       4 | tc(From, To) : tc(From, Hop), tc(Hopp, To).

tc.dr:4:34 error: Named variable 'Hopp' is only used once; you should use either '_' or prefix the name with an '_' to explicitly mark it as anonymous
       4 | tc(From, To) : tc(From, Hop), tc(Hopp, To).
```

Footgun holster: Blessing evil cross-products

- **Easy to accidentally write code with a cross-product**
  - Cross-products have very bad performance characteristics, producing MxN tuples
  - Often the result of an accidental typo, or fault in logic
  - When desired, a cross product must be “blessed” with the `@product` pragma

```
tc.dr:4:1 error: This clause requires a cross-product, but has not been annotated with a '@product' pragma (placed between the clause head and colon)
      4 | tc(From, To) : tc(From, Hop), tc(Hopp, To).

tc.dr:4:40 note: This variable contributes to view 1 of the 2 views that need to be combined into a cross product
      4 | tc(From, To) : tc(From, Hop), tc(Hopp, To).

tc.dr:4:34 note: This variable contributes to view 1 of the 2 views that need to be combined into a cross product
      4 | tc(From, To) : tc(From, Hop), tc(Hopp, To).

tc.dr:4:25 note: This variable contributes to view 2 of the 2 views that need to be combined into a cross product
      4 | tc(From, To) : tc(From, Hop), tc(Hopp, To).

tc.dr:4:19 note: This variable contributes to view 2 of the 2 views that need to be combined into a cross product
      4 | tc(From, To) : tc(From, Hop), tc(Hopp, To).
```

How does Dr. Lojekyll work?

Dr. Lojekyll compiles Datalog into C++/Python

- **Datalog → Data-flow IR → Control-flow IR → Target code**
  - **Data-flow IR** is a graph-based IR, which models a relational data-flow graph (DFG)
    - Nodes in the DFG are called “views,” and values in the nodes are called “columns”
  - **Control-flow IR** is a tree-based IR, representing a procedural scheduling of the data flow IR
    - Operates on tables, indices, vectors, and variables
    - Tables map tuples to state (present, absent, unknown)
    - Contains both bottom-up and top down code
      - Bottom-up: Insertion and removal
      - Top-down: Discover alternate proofs for opportunistically removed tuples
  - **Target code**, which can be Python or C++, is generated
    - Functors can be inlined across the Datalog/C++ boundary
    - Code can be embedded via `#prologue` and `#epilogue` statements in Datalog source
    - Foreign types (e.g. `std::shared_ptr<T>`) can be bound to Datalog type names using `#foreign` type declarations

Segue: Why target Python code?

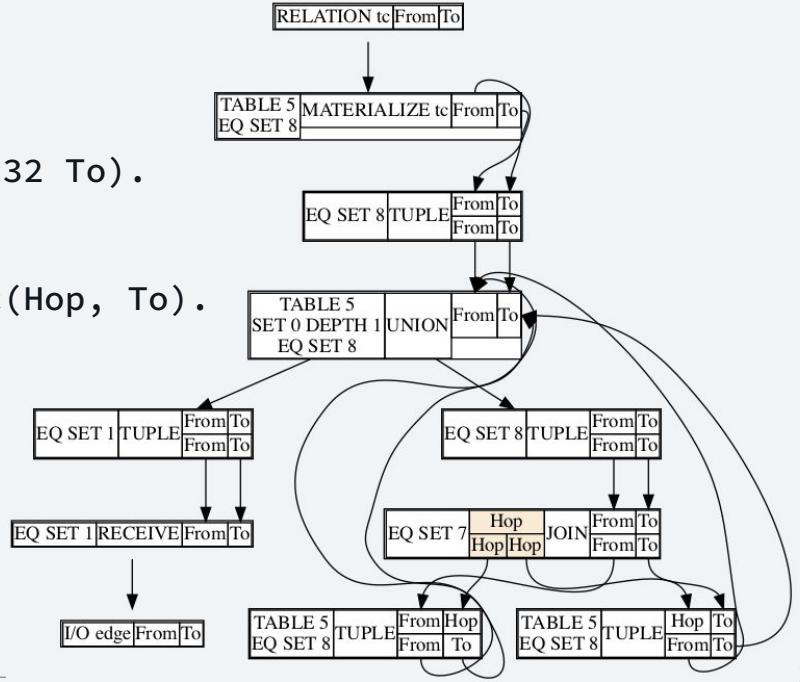
- **Ended up being an easy target for control-flow IR**
  - Tables and indices are dictionaries, vectors are lists, etc.
- **Used `#epilogue` statements to make Datalog self-testing**
  - Prologue statements for importing dependencies
  - Epilogue statements for defining a `main` function and running a unit test
  - Statically type-checked with mypy
  - Output Python code, when executed, would perform a unit test
    - Helped find bugs throughout compiler
- **Python code generation motivated early applications**
  - Disassembly of code, which helped us figure out what types of rules worked well
  - Differential parser of Solidity code: take your evidence where you can get it

Back to the compiler: Data-flow IR views/nodes

- **Many types of views (nearly an exhaustive list)**
  - **RECEIVE**: Receive data from a message stream (selection in relational algebra), associated with a `#message` declaration used in a clause body
  - **TUPLE**: Pass through data, introduce constants (selection/projection in relational algebra)
  - **JOIN**: Equi-join two or more views on one or more "pivots"
  - **PRODUCT**: Cross-product of two or more input views
  - **UNION**: Merge together multiple input views (set union)
  - **AND-NOT/AND-NEVER**: Negation (set difference)
  - **MATERIALIZE/PUBLISH**: Insert into a materialized view backing a `#query` declaration, or publish a message associated with a `#message` declaration used as a clause head
  - **MAP**: Apply a functor to some inputs, producing additional outputs
  - **PREDICATE**: Filter data via a functor call that produces no outputs
  - **COMPARE**: Filter data via a binary inequality operator (<, >, ≠)
  - **AGGREGATE**: Apply an aggregating functor to summarize inputs into outputs
  - **KVINDEX**: Models a key/value store, where a merge functor joins old values with new ones

Example data-flow IR for transitive closure

```
#message edge(u32 From, u32 To).
#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).
tc(From, To) : tc(From, Hop), tc(Hop, To).
```



Key features of the data-flow IR

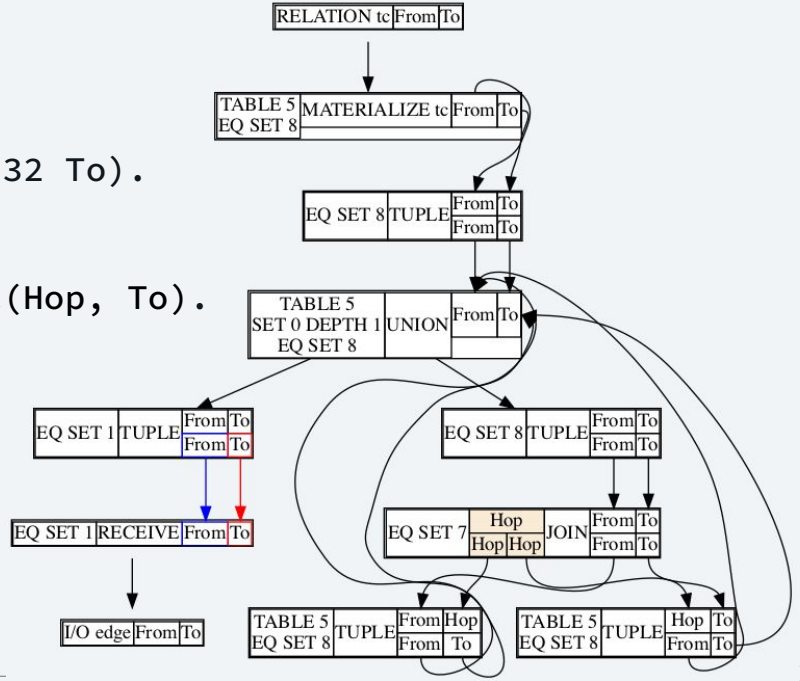
- **Directed graph**
  - Does not prescribe a concrete execution order
  - Edges in graph represent data sources; purple edges admit differential updates
- **Multiple views can share the same backing storage**
  - Nodes belong to storage equivalence classes "`EQ SET <n>`"
  - Two nodes in the same storage equivalence class have their data stored in the same place
- **Some views require storage, while others do not**
  - Views in the same storage equivalence class always share the same table (if any)
  - **UNION** nodes that belong to data flow cycles require storage
  - Predecessors of **JOIN** and **PRODUCT** nodes require storage
  - Target of **AND-NOT**/**AND-NEVER** nodes always require a table
  - Predecessor of **MATERIALIZE** nodes always require a table
- **Cyclic UNIONs belong to fixpoint sets**
  - Represented as "`SET <n> DEPTH <k>`" information
  - Equivalence class of cooperating **UNION**s into fixpoint sets "`SET <n>`"
  - Fixpoint sets are partially ordered by logical depth "`DEPTH <k>`"
    - Induces a happens-before relation on fixpoint sets

Purpose of data-flow IR is optimization

- **Optimizations are similar in spirit to “magic method” or SLDMagic**
  - Operate at the algebraic level, rather than as syntactic transformations
- **Constant propagation**
  - Constants are propagated “upward” through the data flow graph
  - Constants reaching into a pivot of a **JOIN** can be pushed up, around, and down to other predecessors of the **JOIN**
- **Predicate pushdown**
  - Can sink successors to become predecessors, thus “specializing” predecessors
- **Key challenge: Maintaining control-dependencies**
  - Control-dependencies are implicit, not explicit
    - Biggest form of control-dependency: upstream receipt of a message
  - Optimization can lead us to drop dependencies on predecessor nodes
    - Dependency on predecessor implies certain conditions must be satisfied, so we need to maintain those dependencies via other mechanisms (`CONDition` variable nodes)

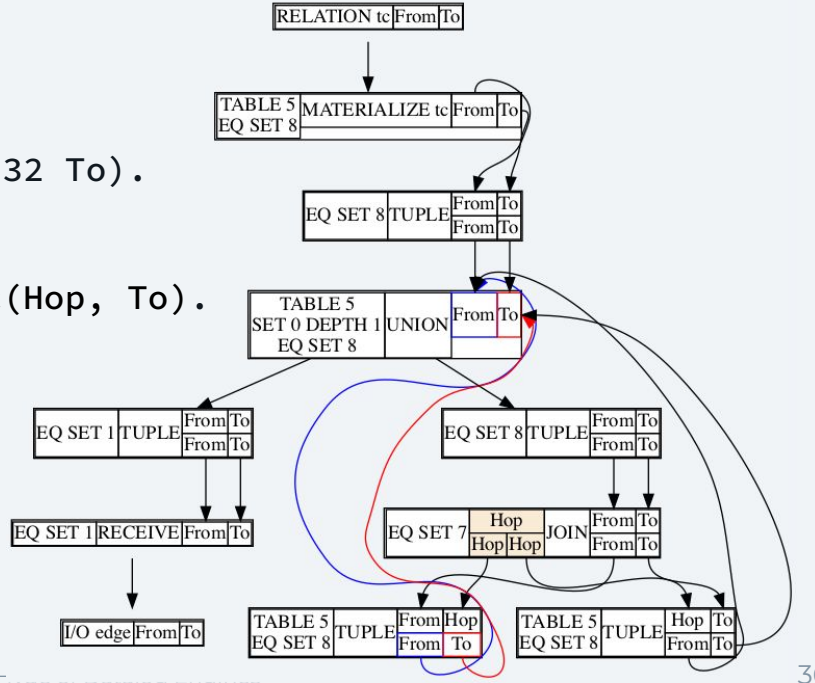
How Datalog code relates to the data-flow IR

```
#message edge(u32 From, u32 To).
#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).
tc(From, To) : tc(From, Hop), tc(Hop, To).
```



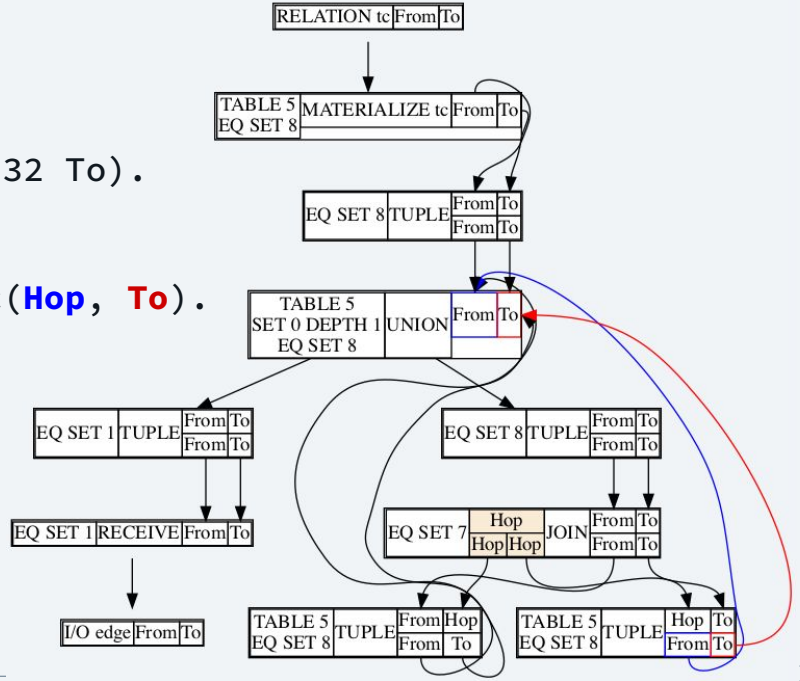
How Datalog code relates to the data-flow IR

```
#message edge(u32 From, u32 To).
#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).
tc(From, To) : tc(From, Hop), tc(Hop, To).
```



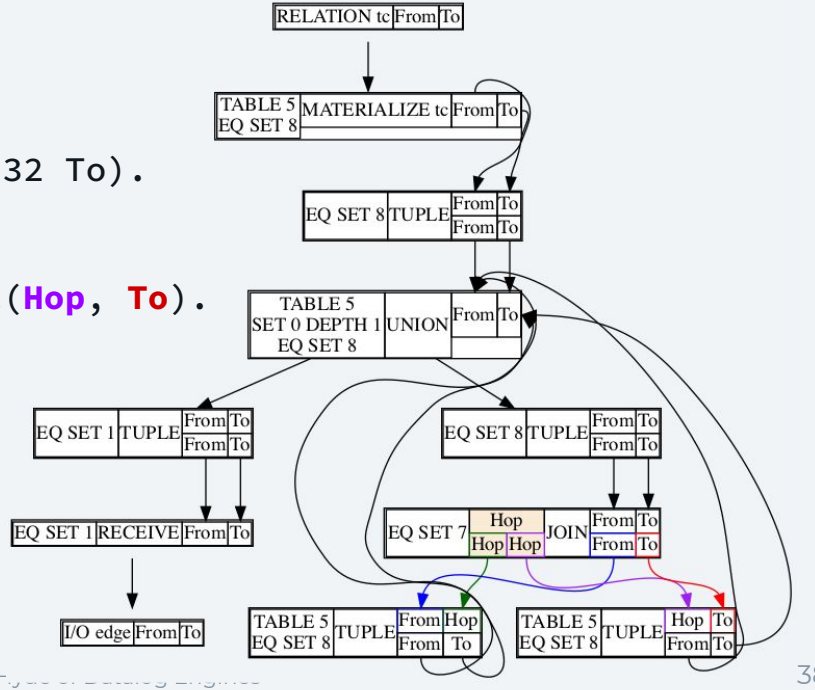
How Datalog code relates to the data-flow IR

```
#message edge(u32 From, u32 To).
#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).
tc(From, To) : tc(From, Hop), tc(Hop, To).
```



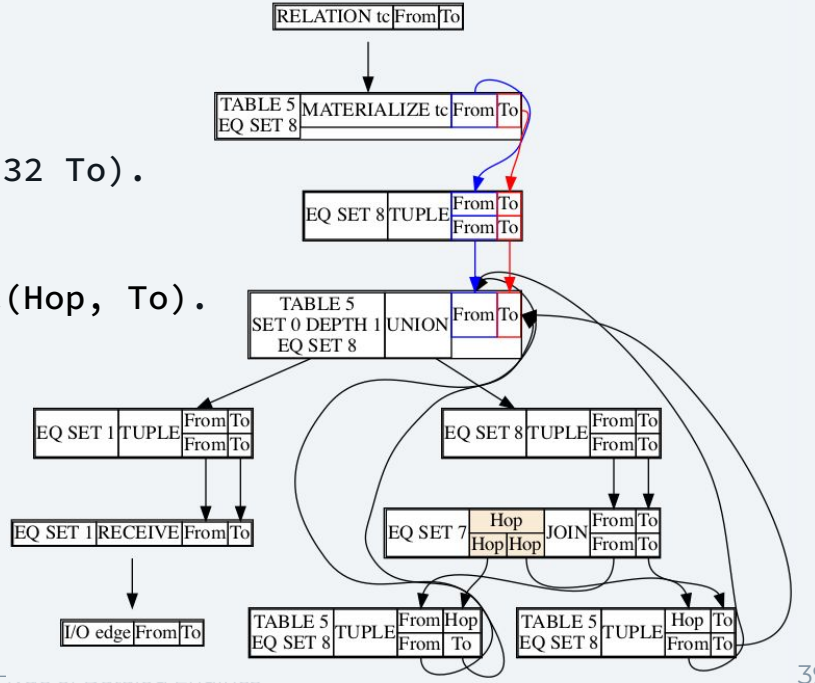
How Datalog code relates to the data-flow IR

```
#message edge(u32 From, u32 To).
#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).
tc(From, To) : tc(From, Hop), tc(Hop, To).
```



How Datalog code relates to the data-flow IR

```
#message edge(u32 From, u32 To).
#query tc(bound u32 From, free u32 To).

tc(From, To) : edge(From, To).
tc(From, To) : tc(From, Hop), tc(Hop, To).
```

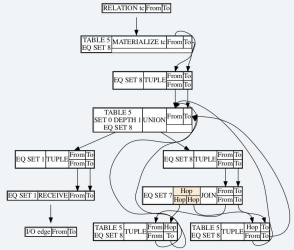


Regaining control-flow from data-flow

- **Several ways to make data-flow IR executable**
  - **Tuple-at-a-time execution** with push method
    - Problem: recursion depth leads to stack overflow
    - Problem: removal of tuples can't just be even deeper recursion, otherwise your continuation might introduce an inconsistency
  - **Unstructured vectorized execution**, where a node's inbox is a vector of tuples
    - Code for nodes fill inboxes of successor nodes; loop until all inboxes are empty
    - Problem: Lots of copying; Mitigation: columnar compression
    - Problem: Copying through vectors prohibits copy-propagation across nodes
      - Due to design limitation: nodes can't take values from arbitrary predecessors
  - **Structured vectorized execution**, where there are only vectors for pipeline blockers
    - Procedural, with control-flow structures for joins, products, and inductive loops
    - Optimizable IR, e.g. copy propagation, loop fusion, hoisting, etc.
    - Procedures for top-down alternative proof finders

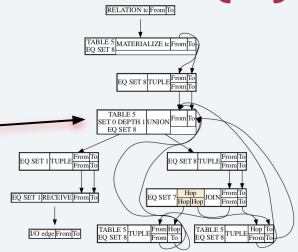
Excerpt of transitive closure control-flow IR (1)

```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $induction_in:16<u32,u32>
    vector-swap $induction_in:16<u32,u32>, $induction_swap:17<u32,u32>
    vector-loop {@From:21, @To:22} over $induction_swap:17<u32,u32>
       …
        vector-append {@To:22} into $pivots:26<u32>
        vector-append {@From:21} into $pivots:26<u32>
    join-tables
      vector-loop {@Hop:28} over $pivots:26<u32>
      select ... using %index:29[_,u32] where %col:7 = @Hop:28
      select ... using %index:30[u32,_] where %col:6 = @Hop:28
        change-tuple {@From:33, @To:34} ... from absent to present
          vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
    vector-clear $pivots:26<u32>
```



Excerpt of transitive closure control-flow IR (2)

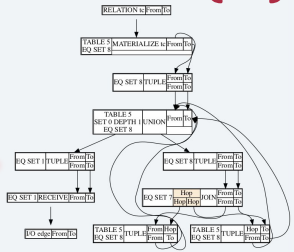
```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $induction_in:16<u32,u32>
    vector-swap $induction_in:16<u32,u32>, $induction_swap:17<u32,u32>
    vector-loop {@From:21, @To:22} over $induction_swap:17<u32,u32>
      …
        vector-append {@To:22} into $pivots:26<u32>
        vector-append {@From:21} into $pivots:26<u32>
    join-tables
      vector-loop {@Hop:28} over $pivots:26<u32>
      select ... using %index:29[_,u32] where %col:7 = @Hop:28
      select ... using %index:30[u32,_] where %col:6 = @Hop:28
        change-tuple {@From:33, @To:34} ... from absent to present
          vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
    vector-clear $pivots:26<u32>
```



Inductive loop that keeps looping so long as the vector, `induction_in:16`, contains tuples

Excerpt of transitive closure control-flow IR (3)

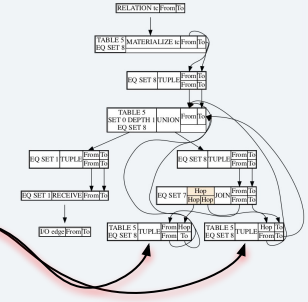
```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $induction_in:16<u32,u32>
    vector-swap $induction_in:16<u32,u32>, $induction_swap:17<u32,u32>
    vector-loop {@From:21, @To:22} over $induction_swap:17<u32,u32>
      …
        vector-append {@To:22} into $pivots:26<u32>
        vector-append {@From:21} into $pivots:26<u32>
    join-tables
      vector-loop {@Hop:28} over $pivots:26<u32>
      select ... using %index:29[_,u32] where %col:7 = @Hop:28
      select ... using %index:30[u32,_] where %col:6 = @Hop:28
        change-tuple {@From:33, @To:34} ... from absent to present
          vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
    vector-clear $pivots:26<u32>
```



Initially, the `induction_in:16` vector contains the inputs from the `edge` message

Excerpt of transitive closure control-flow IR (4)

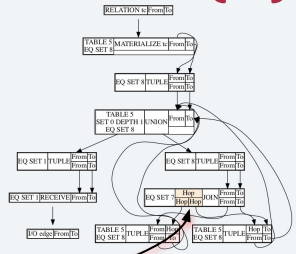
```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $induction_in:16<u32,u32>
    vector-swap $induction_in:16<u32,u32>, $induction_swap:17<u32,u32>
    vector-loop {@From:21, @To:22} over $induction_swap:17<u32,u32>
      …
        vector-append {@To:22} into $pivots:26<u32>
        vector-append {@From:21} into $pivots:26<u32>
      join-tables
        vector-loop {@Hop:28} over $pivots:26<u32>
        select ... using %index:29[_,u32] where %col:7 = @H
        select ... using %index:30[u32,_] where %col:6 = @H
          change-tuple {@From:33, @To:34} ... from absent to present
            vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
      vector-clear $pivots:26<u32>
```



The induction vector is swapped with a local vector, and we loop over tuples in the local vector

Excerpt of transitive closure control-flow IR (5)

```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $induction_in:16<u32,u32>
    vector-swap $induction_in:16<u32,u32>, $induction_swap:17<u32,u32>
    vector-loop {@From:21, @To:22} over $induction_swap:17<u32,u32>
      …
        vector-append {@To:22} into $pivots:26<u32>
        vector-append {@From:21} into $pivots:26<u32>
    join-tables
      vector-loop {@Hop:28} over $pivots:26<u32>
      select ... using %index:29[_,u32] where %col:7 = @
      select ... using %index:30[u32,_] where %col:6 = @
        change-tuple {@From:33, @To:34} ... from absent to present
          vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
    vector-clear $pivots:26<u32>
```

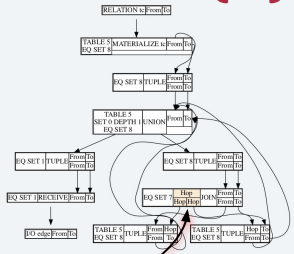


We conditionally add the `To` and the `From` values to a “pivot vector,” which is used to implement an equi-join

Excerpt of transitive closure control-flow IR (6)

```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $indu
    vector-swap $induction_in
    vector-loop {@From:21, @T
      …
        vector-append {@To:22} into $pivots:26<u32>
        vector-append {@From:21} into $pivots:26<u32>
    join-tables
      vector-loop {@Hop:28} over $pivots:26<u32>
      select ... using %index:29[_,u32] where %col:7 = @Hop:28
      select ... using %index:30[u32,_] where %col:6 = @Hop:28
        change-tuple {@From:33, @To:34} ... from absent to present
          vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
    vector-clear $pivots:26<u32>
```

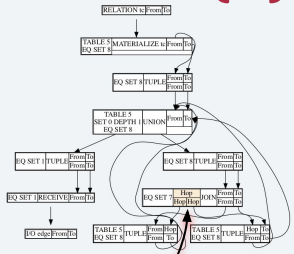
The join “blocks” on the pivot vector, i.e. we execute the join when the pivots are filled up by all possible predecessors



Excerpt of transitive closure control-flow IR (7)

```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $ind
    vector-swap $induction_i
    vector-loop {@From:21, @
       …
        vector-append {@To:22} into $pivots:26<u32>
        vector-append {@From:21} into $pivots:26<u32>
    join-tables
      vector-loop {@Hop:28} over $pivots:26<u32>
      select ... using %index:29[_,u32] where %col:7 = @Hop:28
      select ... using %index:30[u32,_] where %col:6 = @Hop:28
        change-tuple {@From:33, @To:34} ... from absent to present
          vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
    vector-clear $pivots:26<u32>
```

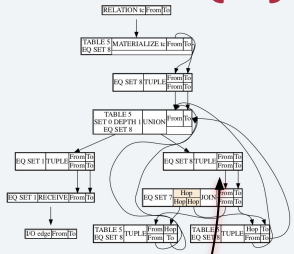
Joins map the columns from the pivot vector into indexed columns



Excerpt of transitive closure control-flow IR (8)

```
vector-define $induction_swap:17<u32,u32>
vector-define $pivots:26<u32>
induction
  fixpoint-loop testing $induction_in:16<u32,u32>
    vector-swap $induction_in:16<u32,u32>, $induction_swap:17<u32,u32>
    vector-loop {@From:21, @To:22} over $induction_swap:17<u32,u32>
      ...
        vec
        vec                                                    >
    join-ta
      vecto
      select ... using %index:29[_,u32] where %col:7 = @Hop:28
      select ... using %index:30[u32,_] where %col:6 = @Hop:28
        change-tuple {@From:33, @To:34} ... from absent to present
          vector-append {@From:33, @To:34} into $induction_in:16<u32,u32>
    vector-clear $pivots:26<u32>
```

Add new transitions to the induction vector, enabling more fixed point iterations



How does Dr. Lojekyll work?

Look ma, C++

```
bool flow_41_(::hyde::rt::Vector<StorageT, uint32_t, uint32_t> vec_16) {
  ::hyde::rt::Vector<StorageT, uint32_t, uint32_t> vec_17(storage, 17u);
  ::hyde::rt::Vector<StorageT, uint32_t, uint32_t> vec_18(storage, 18u);
  ::hyde::rt::Vector<StorageT, uint32_t> vec_26(storage, 26u);
  var_10 += 1;
  for (auto changed_15 = true; changed_15; changed_15 = !!(vec_16.Size())) {
    vec_17.Clear();
    vec_16.SortAndUnique();
    vec_16.Swap(vec_17);
    for (auto [var_21, var_22] : vec_17) {
      vec_18.Add(var_21, var_22);
      vec_26.Add(var_22);
      vec_26.Add(var_21);
    }
    vec_26.SortAndUnique();
    for (auto [var_28] : vec_26) {
      ::hyde::rt::Scan<StorageT, ::hyde::rt::IndexTag<29>> scan_27_0(storage, table_5, var_28);
      ::hyde::rt::Scan<StorageT, ::hyde::rt::IndexTag<30>> scan_27_1(storage, table_5, var_28);
      for (auto [var_33, var_31] : scan_27_0) {
        for (auto [var_32, var_34] : scan_27_1) {
          if (std::make_tuple(var_28, var_28) == std::make_tuple(var_31, var_32)) {
            if (table_5.TryChangeTupleFromAbsentToPresent(var_33, var_34)) {
              vec_16.Add(var_33, var_34);
            }
          }
        }
      }
    }
    vec_26.Clear();
  }
```

Mini disassembler demonstration

Mini disassembler demonstration

- **Semi-realistic rules that infer function entrypoint addresses based off of control-flow transitions between instructions**
- **Highlights incremental and differential updates**

```
#enum EdgeType u8.

#constant EdgeType FALL_THROUGH 0.
#constant EdgeType CALL 1.

#message instruction(u64 EA).
#message raw_transfer(u64 FromEA, u64 ToEA,
                      EdgeType TransferKind).
```

```
#query function(bound u64 ToEA)

    ; The target of a function call is an instruction.
    : raw_transfer(FromEA, ToEA, CALL)
    , instruction(FromEA)
    , instruction(ToEA)

    ; Any instruction without a predecessor is a function.
    : !raw_transfer(_, ToEA, _)
    , instruction(ToEA).

#query function_instructions(bound u64 FuncEA, free u64 InstEA)

    ; The first instruction of a function is a function instruction.
    : function(FuncEA)
    , FuncEA = InstEA

    ; The fall-through of one function instruction is also a function
    ; instruction, assuming it's not a function head.
    : function_instructions(FuncEA, PredEA)
    , raw_transfer(PredEA, InstEA, FALL_THROUGH)
    , !function(InstEA).
```

TRAIL OF BITS