Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declarative Routing

Seminar in Distributed Computing 08
with papers chosen by Prof. T. Roscoe

Presented by David Gerhard

Overview

- Motivation
- P2
- NDLog
- Conclusion
- Questions...?

Motivation

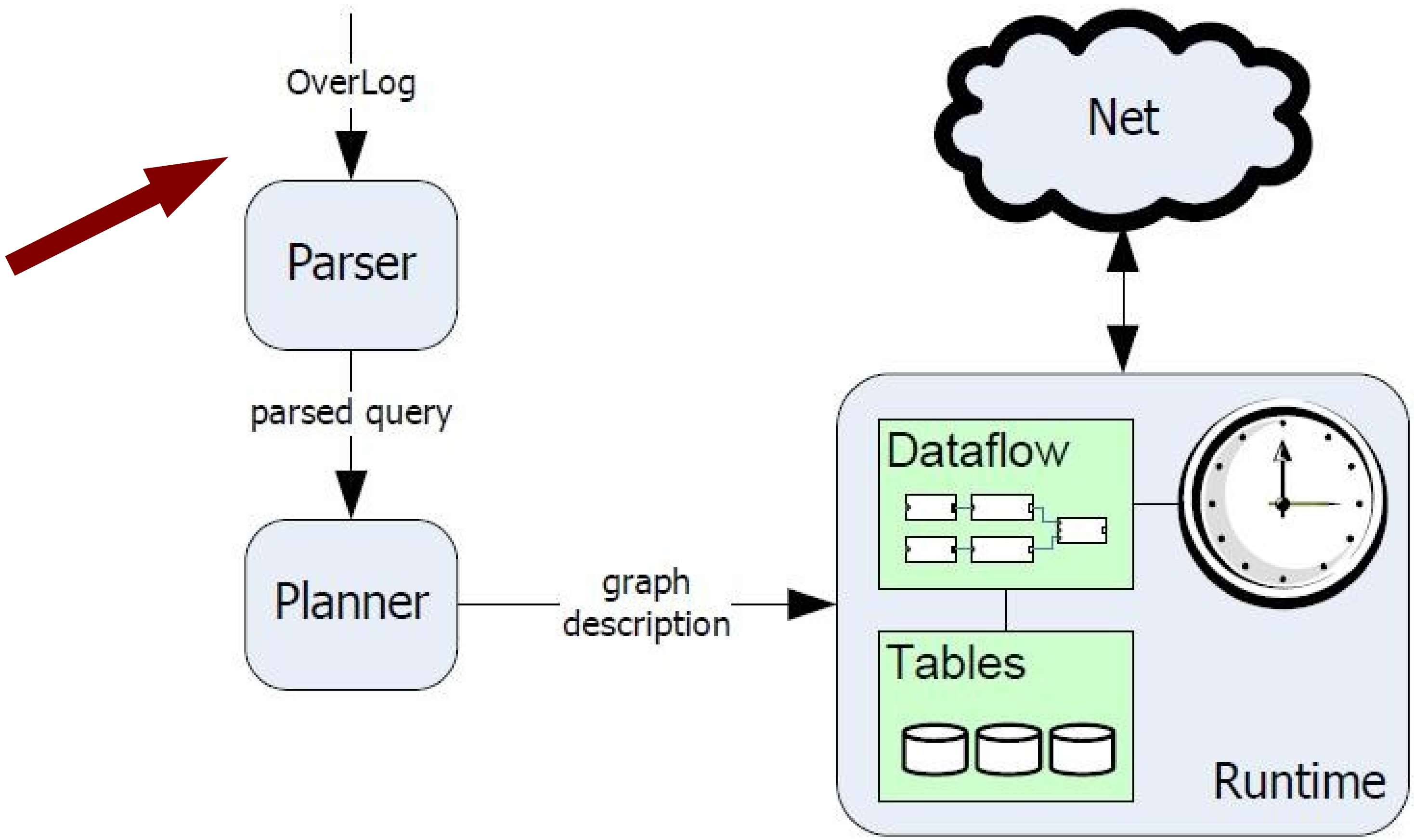
- Overlay networks are widely used today (p2p,...)
- Difficult to create and implement
- Not really extensible, not really reusable
- Declarative approach promises flexibility and compactness
- Declarative language enables static program checks for correctness and security
- Declarative networking is part of larger effort to revisit the current Internet Architecture

P2

- P2 is a system for the construction, maintenance and sharing of overlay networks, using:
  - Declarative language
  - Dataflow architecture
  - Soft-state tables, streams of tuples
  - Implemented in C++ using UDP

- Does resource discovery and network monitoring

Structure of a P2 Node



OverLog

- Based on Datalog(subset of Prolog) query language
- Specification of physical distribution (e.g. where tuples are generated, stored, sent)
- Direct translation into dataflow graphs

OverLog - Example

- [<ruleID> <head> :- <body>]

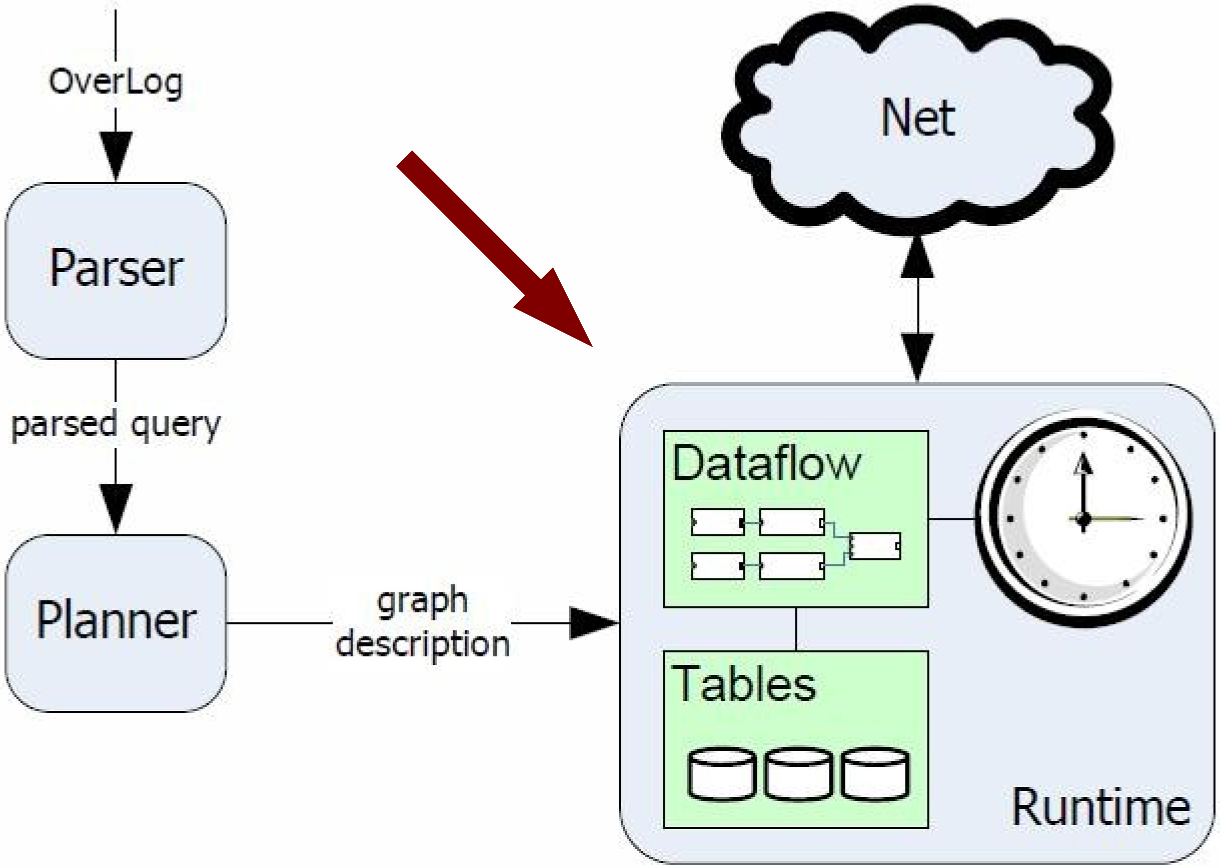
- P2 pong@X(X, Y, E, T) :- ping@Y(Y, X, E, T).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

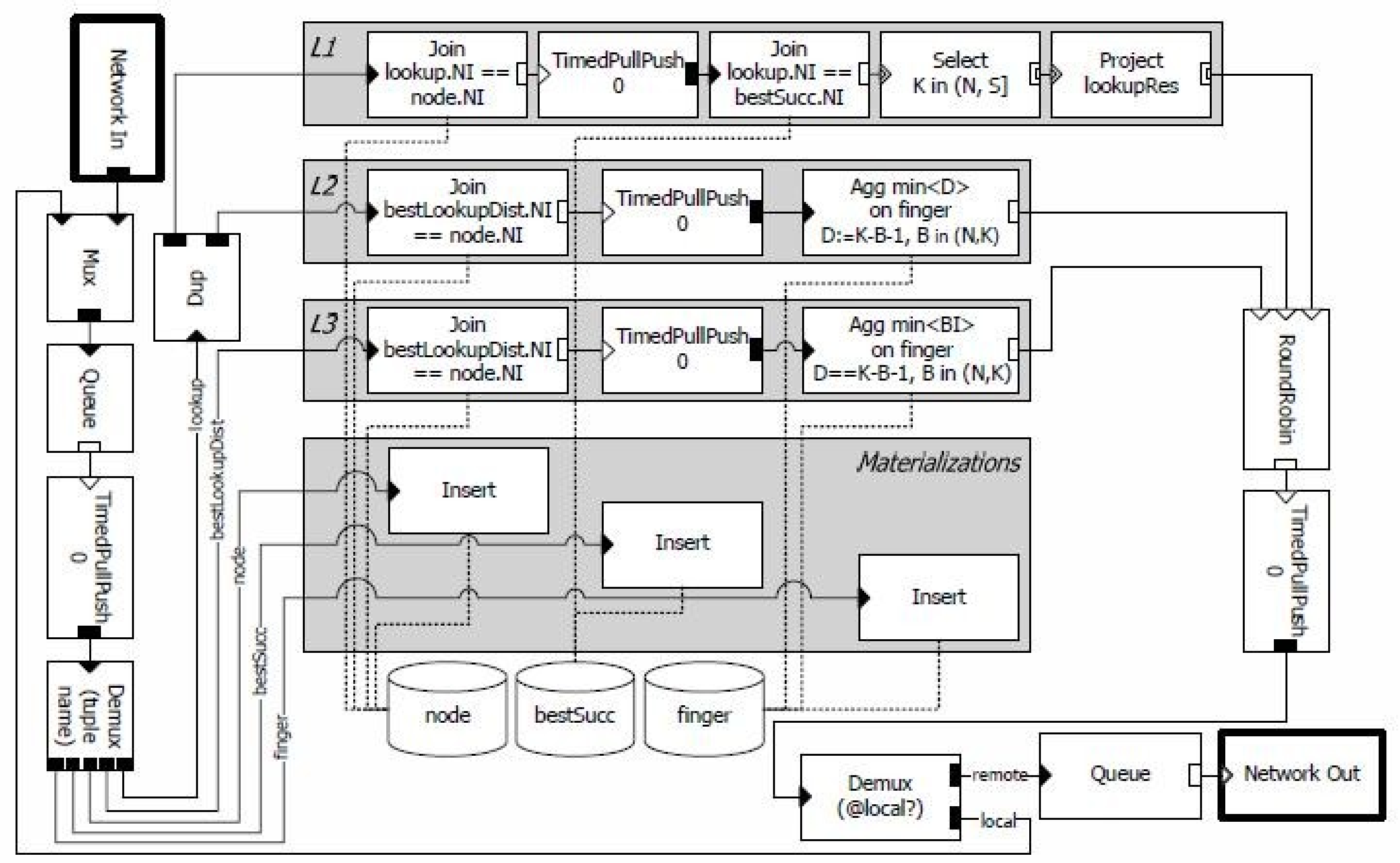
OverLog – Ping Example

```
materialize(member, 120, infinity, keys(2)).

P0 pingEvent@X(X, Y, E, max<R>) :- periodic@X(X, E, 2),
member@X(X, Y, _, _, _), R := f_rand().

P1 ping@Y(Y, X, E, T) :- pingEvent@X(X, Y, E, _), T := f_now@X().

P2 pong@X(X, Y, E, T) :- ping@Y(Y, X, E, T).

P3 latency@X(X, Y, T) :- pong@X(X, Y, E, T1), T := f_now@X() - T1.
```

Structure of a P2 Node



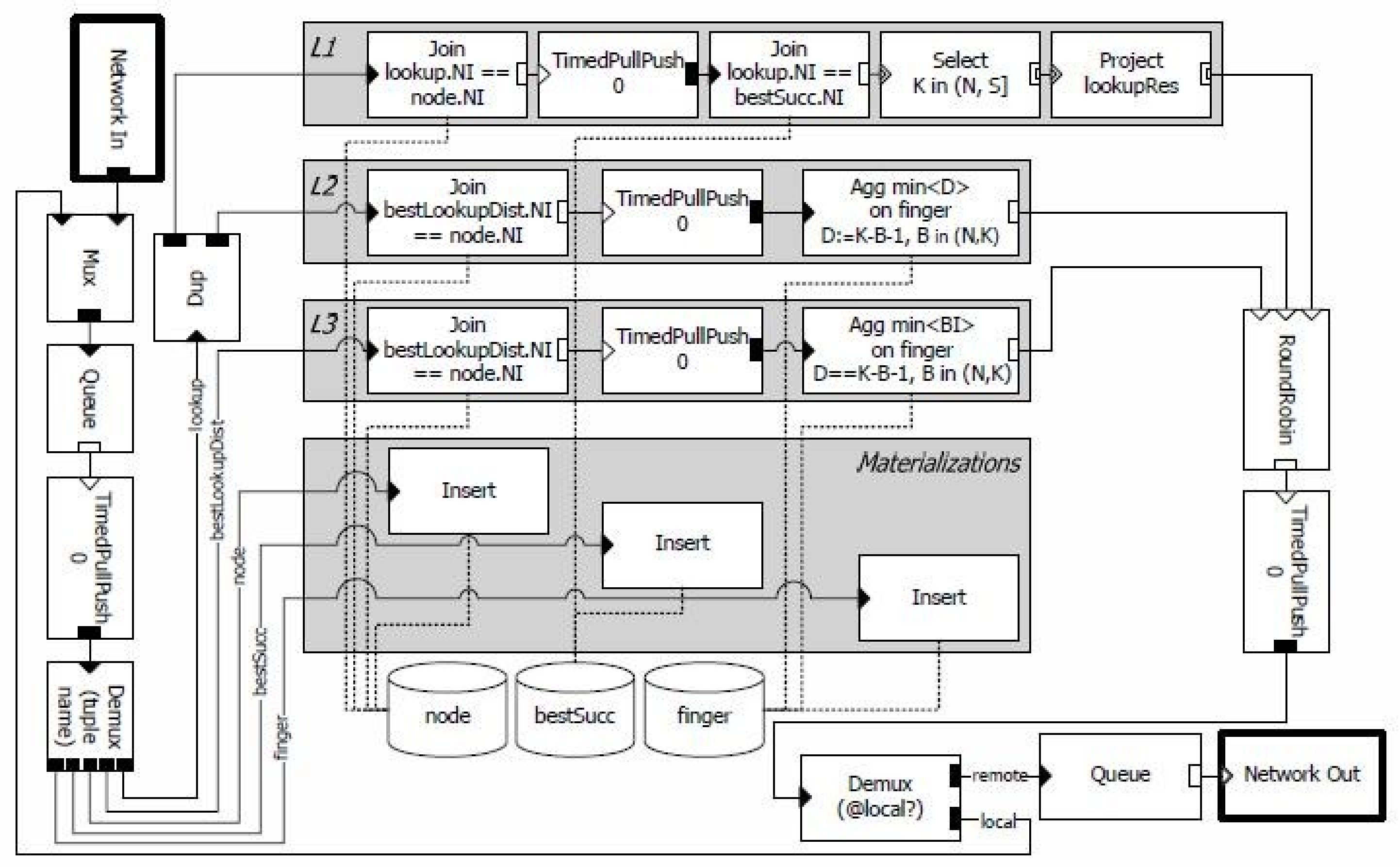
Dataflow


Network In
L1
Join
lookup.NI ==
node.NI
TimedPullPush
0
Join
lookup.NI ==
bestSucc.NI
Select
K in (N, S]
Project
lookupRes
L2
Join
bestLookupDist.NI
== node.NI
TimedPullPush
0
Agg min<D>
on finger
D:=K-B-1, B in (N,K)
L3
Join
bestLookupDist.NI
== node.NI
TimedPullPush
0
Agg min<BI>
on finger
D==K-B-1, B in (N,K)
Mux
Dup
Queue
TimedPullPush
0
Demux
(tuple
name)
lookup
bestLookupDist
node
bestSucc
finger
Materializations
Insert
Insert
Insert
node
bestSucc
finger
RoundRobin
TimedPullPush
0
Demux
(@local?)
remote
local
Queue
Network Out

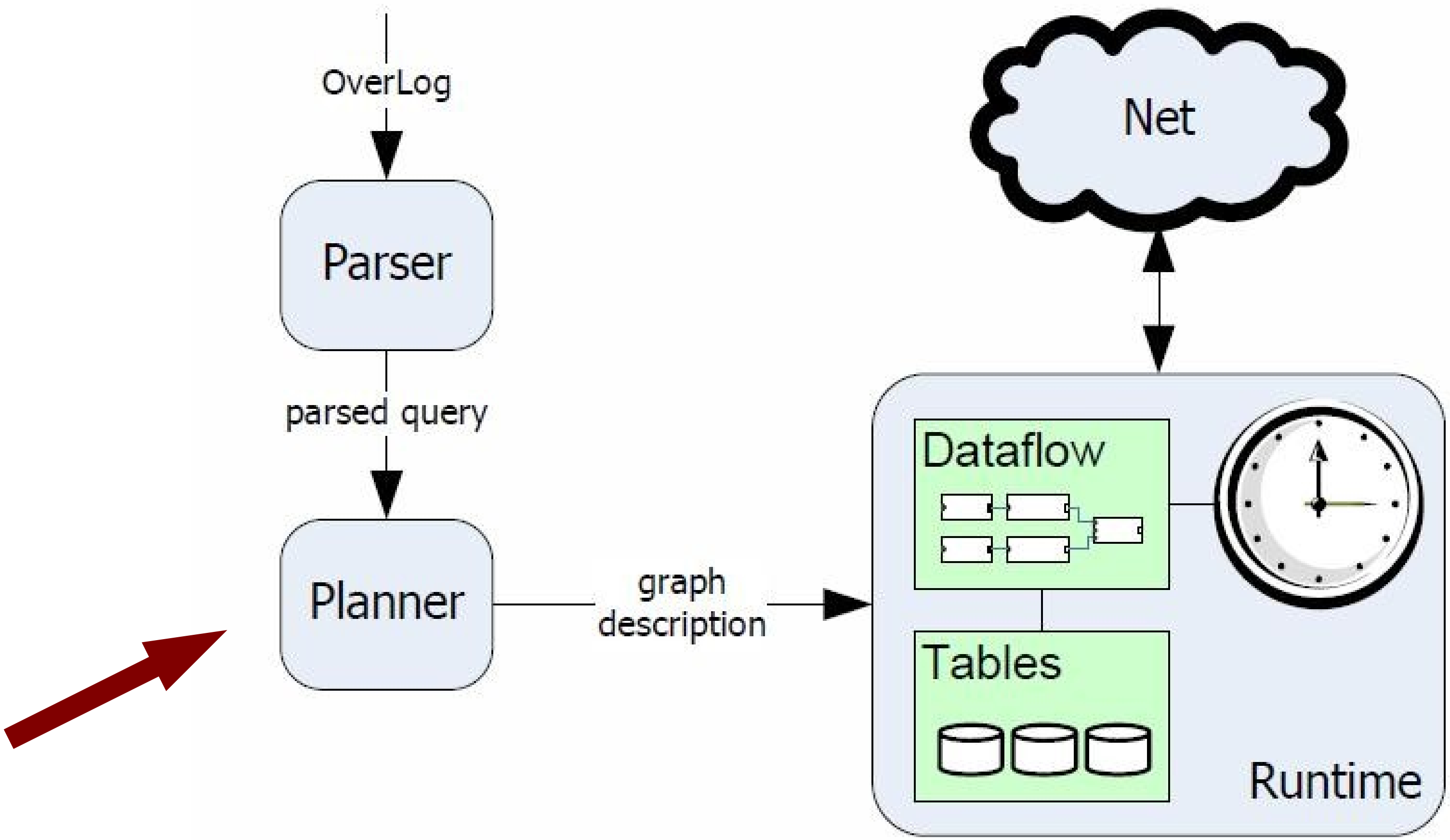
Dataflow

- Consists of nodes(elements)
  - Selection, projection, join, group-by, aggregation
- Forms a directed dataflow graph
- Edges carries well structured tuples
- Arbitrary number of input/output ports per element
- Handles “network”
  - Responsible for Sockets
  - Packet scheduling
  - Congestion control
  - Reliable transmission
  - Data serialization

Dataflow


Network In
L1
Join lookup.NI == node.NI
TimedPullPush 0
Join lookup.NI == bestSucc.NI
Select K in (N, S]
Project lookupRes
L2
Join bestLookupDist.NI == node.NI
TimedPullPush 0
Agg min<D> on finger D:=K-B-1, B in (N,K)
L3
Join bestLookupDist.NI == node.NI
TimedPullPush 0
Agg min<BI> on finger D==K-B-1, B in (N,K)
Materializations
Insert
Insert
Insert
node
bestSucc
finger
Mux
Dup
Queue
TimedPullPush 0
Demux (tuple name)
lookup
bestLookupDist
node
bestSucc
finger
RoundRobin
TimedPullPush 0
Demux (@local?)
remote
local
Queue
Network Out

Structure of a P2 Node



Planer

- Input: parsed OverLog
- Output: dataflow graph

- Adds network stack
- Uses “built in” elements (e.g. periodic, f_now), which are directly mapped to dataflow elements

Evaluation - Setting

- Using a P2 implementation of Chord DHT
  - Configured to use low bandwidth
  - Aiming at high consistency and low latency
- Tested on the Emulab testbed(100 machines)
- 10 transit domains (100Mbps)
- 100 stubs (10Mpbs)
- RTT transit-transit 50ms
- RTT stub-stub same domain 2ms

Evaluation – Results Static Test

- 500-node static network, 96% lookups complete in <=6s
- About the same as the published numbers of MIT Chord

Evaluation – Results “Handling Churn”

- Churn = continuous process of node arrival&departure
- Low Churn(session time >=64min)
  - P2 Chord does well
  - 97% consistent lookups
  - Most of which under 4s
- High Churn(session time <= 16min)
  - P2 Chord does not well
  - 42% consistent lookups
  - 84% with high latency
- MIT Chord
  - 99.9% consistent lookups, session time 47min
  - High Churn mean lookup latency of less than 5s

Conclusion I

- Feasibility study
- Approach looks promising, but needs further work
  - Further tests with other overlay networks
  - Security
- Planner does not handle some constructs of OverLog
  - Multi-node rule bodies
  - Negation
- Combination of declarative language and dataflow graphs powerful, alternative: automaton
- Declarative language enables static program checks for correctness and security

Conclusion II

- OverLog is very concise (Chord in 47 rules)
- OverLog is “difficult”
  - Not easy to read (Prolog is hard to read), but can be directly compiled and executed by P2 nodes
  - Non-trivial learning curve
  - No if-then-else
  - No order of evaluation, everything is tested “in parallel”
    - Could profit from multiprocessor environments
- OverLog Chord implementation not declarative enough
- Replace OverLog?

NDLog - Introduction

- Extends P2
- New declarative language NDLog
  - Explicit control over data placement and movement
- Buffered/pipelined semi-naïve evaluation
- Concurrent updates of the network while running
- Query optimization
- Assumes not fully connected network graph, but assumes bidirectional links

NDLog

- Introduces new datatype address
  - Address variables/constants name start with “@”
- First field in all predicates is the location address of the tuple (**bold** for clarity)
- *Link relation* are stored, representing the connectivity information of the queried network
- *Link literal* is a link relation in the body of a rule
  - #link(@src,@dst,...)

NDLog II

- Rules with the same location specifier in each predicate, including Head, are called *local rules*
- Link-restricted rule
  - exactly one link literal
  - all other literals are located either at the Src or Dst of the link literal
- Every rule in NDLog is either a local rule or a link-restricted rule

NDLog - Example

- [<ruleID> <head> :- <body>]

- OverLog
  - P2 pong@X(X, Y, E, T) :- ping@Y(Y, X, E, T).

- NDLog
  - **SP1:** path(**@S**,@D,@D,P,C) :- #link (**@S**,@D,C), P = *f_concatPath*(link(**@S**,@D,C), nil).

NDLog - Example

SP1: path(**@S**,@D,@D,P,C) :- #link (**@S**,@D,C),

P = *f concatPath*(link(**@S**,@D,C), nil).

SP2: path(**@S**,@D,@Z,P,C) :- #link (**@S**,@Z,C1),
path(**@Z**,@D,@Z2,P2,C2), C = C1 + C2,

P = *f concatPath*(link(**@S**,@Z,C1),P2).

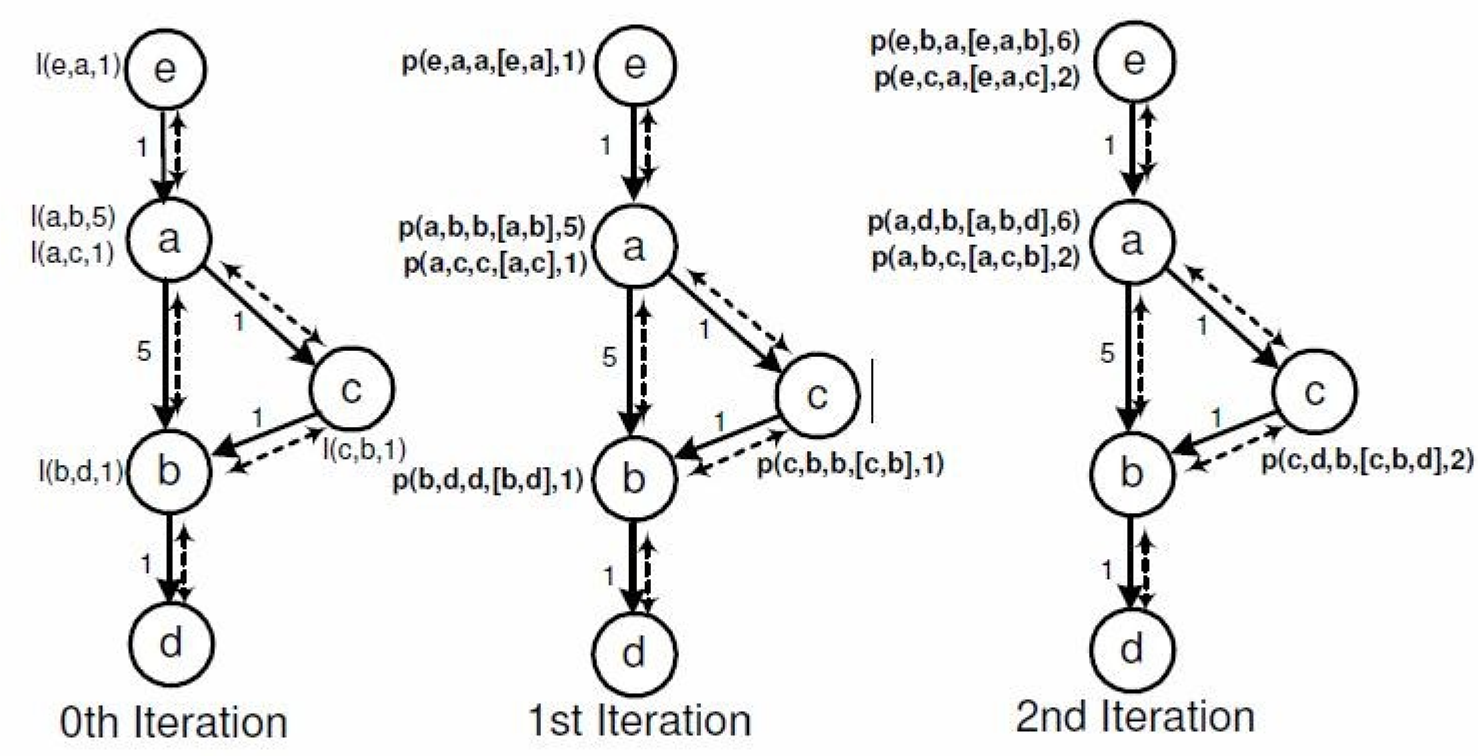
SP3: spCost(**@S**,@D,min<C>) :- path(**@S**,@D,@Z,P,C).

SP4: shortestPath(**@S**,@D,P,C) :- spCost(**@S**,@D,C),

path(**@S**,@D,@Z,P,C).

Query: shortestPath(**@S**,@D,P,C).

Example


l(e,a,1)
e
1
l(a,b,5)
l(a,c,1)
a
5
1
c
1
l(c,b,1)
l(b,d,1)
b
1
d
0th Iteration
p(e,a,a,[e,a],1)
p(a,b,b,[a,b],5)
p(a,c,c,[a,c],1)
p(c,b,b,[c,b],1)
p(b,d,d,[b,d],1)
1st Iteration
p(e,b,a,[e,a,b],6)
p(e,c,a,[e,a,c],2)
p(a,d,b,[a,b,d],6)
p(a,b,c,[a,c,b],2)
p(c,d,b,[c,b,d],2)
2nd Iteration

Centralized Plan Generation

- Semi-naïve fixpoint evaluation
  - Any new tuples generated for the 1st time are used as input for the next iteration
  - Repeated till a fixpoint is achieved (no new tuples generated)

- Does not work efficiently in Distributed Systems
  - Next iteration on a node can only start when all other nodes have finished the iteration step and all new tuples have been distributed (Barrier)

Distributed Plan Generation

- Iterations are local at every node
- Non-local rules are rewritten that the body is computable at one node
- Buffered semi-naïve
  - Buffers all incoming tuples during a iteration
  - Handled in a future iteration
- Pipelined semi-naïve
  - At arrival every tuple is used to compute new tuples
  - Join operator matches each tuple only with older tuples (timestamp)
  - Enables optimization on a per tuple basis

Semantics in Dynamic Network

- State of the network is constantly changing
- Queries should reflect the most current state of the network

- Continuous Update Model
  - Updates occur very frequently, faster than the fixpoint is reached
  - Query results never fully reflect the state of the network

- Bursty Update Model
  - Updates occur in bursts
  - Between bursts no updates
  - Allows the system to reach a fixpoint

Centralized Semantics

- Insertion
  - Handled by pipelined semi-naïve evaluation
- Deletion
  - Deletion of a base tuple leads to the deletion of any tuples derived from it
- Updates
  - A deletion followed by an insertion

- Works as well in Distributed Systems, as long as
  - There are only FIFO links or
  - All tuples are maintained as soft-state

Query Optimizations

- Traditional Datalog optimizations
  - Aggregate Selections
  - Magic Sets and Predicate Reordering

- Multi-Query Optimizations
  - Query-Result Caching
  - Opportunistic Message Sharing

Experiments

- Using modified P2, running 4 different shortest-path queries
  - Running on a similar emulab testbed

- Results
  - Aggregate Selection reduces communication overhead, periodic even more (by up to 29%)
  - Magic sets and predicate reordering reduce communication overhead when only a limited number of paths are queried
  - Multi-query sharing techniques demonstrate potential to reduce overhead when multiple queries are running concurrent
  - On a network with bursty updates, incremental query evaluation can recompute paths at a fraction of the original costs

Conclusion

- NDLog has a clearer semantic than OverLog
- Relaxations overcome problems in asynchronous distributed settings
- Link restriction allows many optimizations
- Still no negation
- Usability?

Questions?