Soufflé in the Cloud

Lyndon Henry

November 13, 2017

THE UNIVERSITY OF SYDNEY

Soufflé in the Cloud provides the automatic synthesis of cloud services from logic programs.

Contribution

- **Programs become totally independent cloud systems.**
- Programs transformed to execute as concurrently as possible.
- Fully automated, no difference in how programs are written.
- No other system can do logic program to cloud service synthesis.

Contribution

- Programs become totally independent cloud systems.
- **Programs transformed to execute as concurrently as possible.**
- Fully automated, no difference in how programs are written.
- No other system can do logic program to cloud service synthesis.

Contribution

- Programs become totally independent cloud systems.
- Programs transformed to execute as concurrently as possible.
- **Fully automated, no difference in how programs are written.**
- No other system can do logic program to cloud service synthesis.

Contribution

- Programs become totally independent cloud systems.
- Programs transformed to execute as concurrently as possible.
- Fully automated, no difference in how programs are written.
- **No other system can do logic program to cloud service synthesis.**

Cloud Services

Logic Programs

separation of concerns

cost effective

✓ massive scalability

robust & reliability

expertly optimized

Cloud Services

separation of concerns
cost effective
massive scalability
robust & reliability
expertly optimized

Logic Programs

language simplicity
available concurrency
speed & efficiency
mathematical basis
correctness & verifiability

Cloud Services

separation of concerns
cost effective
massive scalability
robust & reliability
expertly optimized

service complexity
limited optimization
network bottleneck
no standardisation
difficult to analyse

Logic Programs

language simplicity
available concurrency
speed & efficiency
mathematical basis
correctness & verifiability

Cloud Services

separation of concerns
cost effective
massive scalability
robust & reliability
expertly optimized

service complexity
limited optimization
network bottleneck
no standardisation
difficult to analyse

Logic Programs

lack of paradigms
self-hosted expenses
single-machine execution
system dependent behaviour
implementation specifics

language simplicity
available concurrency
speed & efficiency
mathematical basis
correctness & verifiability

Cloud Services		Logic Programs
separation of concerns	$\longrightarrow$	lack of paradigms
cost effective	$\longrightarrow$	self-hosted expenses
massive scalability	$\longrightarrow$	single-machine execution
robust & reliability	$\longrightarrow$	system dependent behaviour
expertly optimized	$\longrightarrow$	implementation specifics
service complexity	$\longleftarrow$	language simplicity
limited optimization	$\longleftarrow$	available concurrency
network bottleneck	$\longleftarrow$	speed & efficiency
no standardisation	$\longleftarrow$	mathematical basis
difficult to analyse	$\longleftarrow$	correctness & verifiability

Soufflé
(Logic Programs)

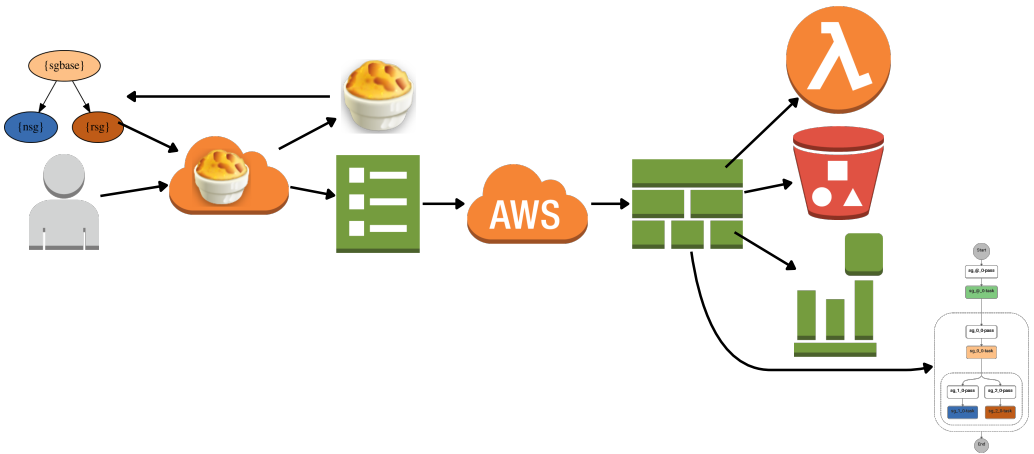
+

The Cloud
(Amazon Web Services)

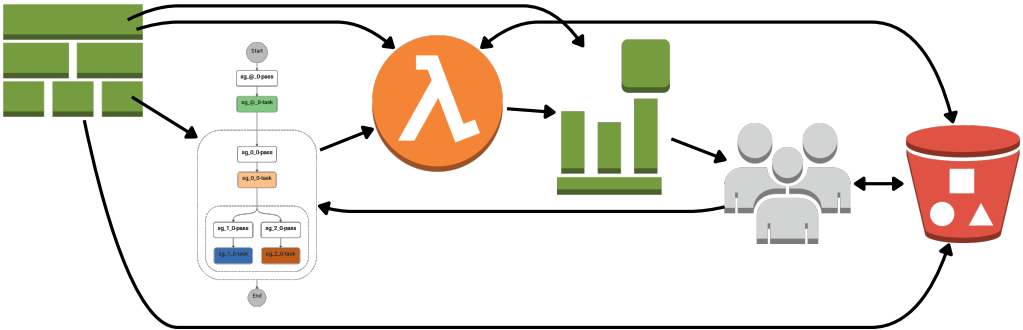
=

Soufflé in the Cloud

Creating a Service

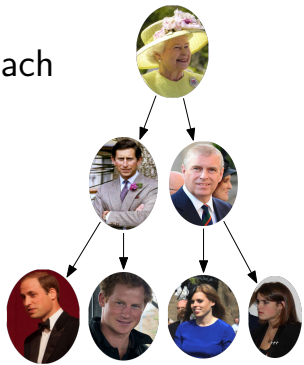


Using a Service



Same Generation Example

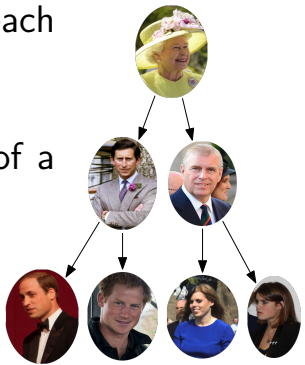
Concept Given a family tree, the people at each level are of the same generation.



Same Generation Example

Concept Given a family tree, the people at each level are of the same generation.

Problem Find all "same generation" vertices of a graph.

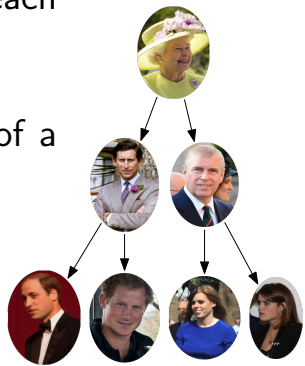


Same Generation Example

Concept Given a family tree, the people at each level are of the same generation.

Problem Find all "same generation" vertices of a graph.

Input Edges as a set of pairs.



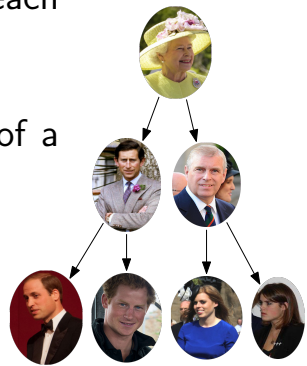
Same Generation Example

Concept Given a family tree, the people at each level are of the same generation.

Problem Find all "same generation" vertices of a graph.

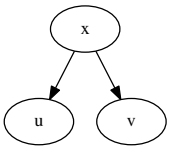
Input Edges as a set of pairs.

Output Pairs of same generation vertices.



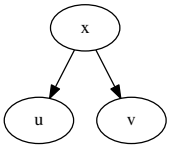
u and v are of the *same generation* if

- x is parent of u
- x is parent of v



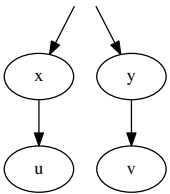
u and v are of the *same generation* if

- x is parent of u
- x is parent of v



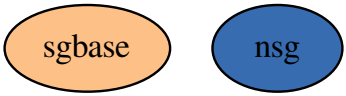
or

- x is parent of u
- y is parent of v
- x and y are of same generation

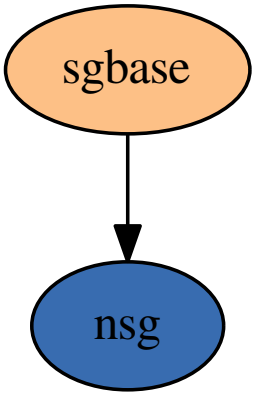


```
// Base Relation
.input sgbase
```

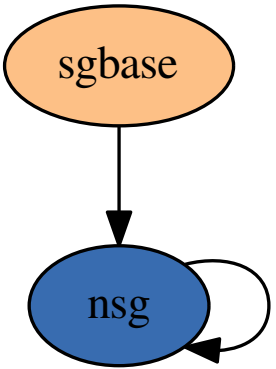
```
// Base Relation
.input sgbase
// Normal Same Generation
.output nsg
```



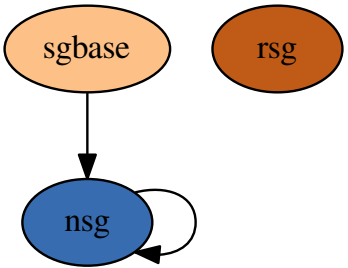
```
// Base Relation
.input sgbase
// Normal Same Generation
.output nsg
nsg(u, v) :- sgbase(x, u),
             sgbase(x, v).
```



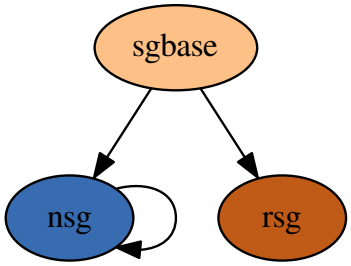
```
// Base Relation
.input sgbase
// Normal Same Generation
.output nsg
nsg(u, v) :- sgbase(x, u),
             sgbase(x, v).
nsg(u, v) :- sgbase(x, u),
             sgbase(y, v),
             nsg(x, y).
```



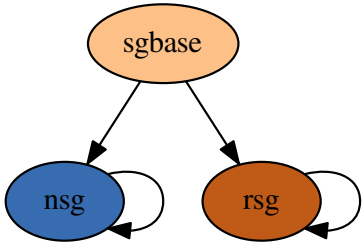
```
// Base Relation
.input sgbase
// Normal Same Generation
.output nsg
nsg(u, v) :- sgbase(x, u),
             sgbase(x, v).
nsg(u, v) :- sgbase(x, u),
             sgbase(y, v),
             nsg(x, y).
// Reverse Same Generation
.output rsg
```



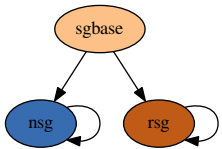
```
// Base Relation
.input sgbase
// Normal Same Generation
.output nsg
nsg(u, v) :- sgbase(x, u),
             sgbase(x, v).
nsg(u, v) :- sgbase(x, u),
             sgbase(y, v),
             nsg(x, y).
// Reverse Same Generation
.output rsg
rsg(u, v) :- sgbase(u, x),
             sgbase(v, x).
```



```
// Base Relation
.input sgbase
// Normal Same Generation
.output nsg
nsg(u, v) :- sgbase(x, u),
             sgbase(x, v).
nsg(u, v) :- sgbase(x, u),
             sgbase(y, v),
             nsg(x, y).
// Reverse Same Generation
.output rsg
rsg(u, v) :- sgbase(u, x),
             sgbase(v, x).
rsg(u, v) :- sgbase(u, x),
             sgbase(v, y),
             rsg(x, y),
```

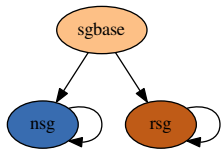


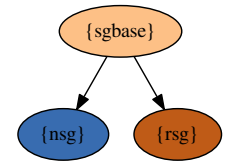
Precedence Graph



1. Obtain dependencies of program.

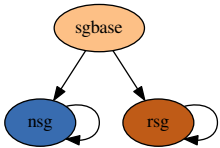
Precedence Graph $\longrightarrow$ Strongly-Connected Component Graph

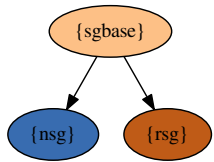


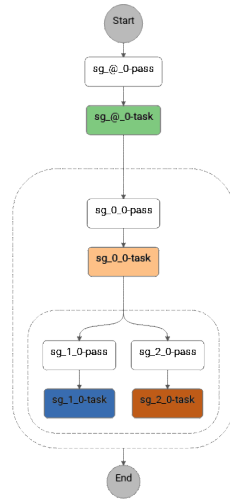


1. Obtain dependencies of program.
2. Contract cycles to sets.

Precedence Graph ⟶ Strongly-Connected Component Graph ⟶ AWS StateMachine







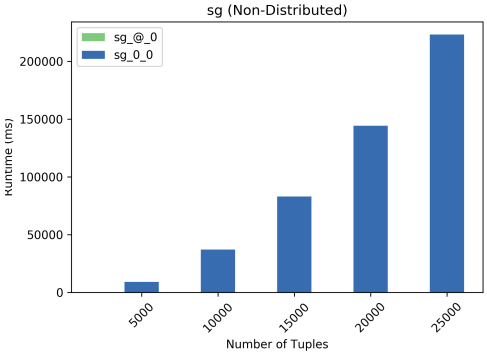
1. Obtain dependencies of program.
2. Contract cycles to sets.
3. Convert to execution template.

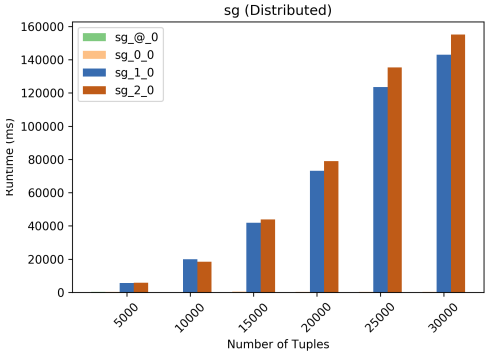
Subprograms as Actors

- ⋆ Subprogram become actors.
- ⋆ Actors communicate by message passing.
- ⋆ Models only actors, acts, and messages.

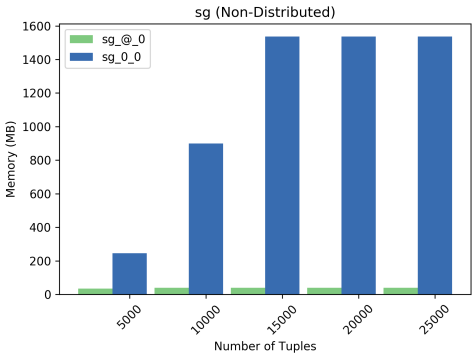
- No global locks.
- Localised coordination.
- Earliest possible execution.
- Minimal communication.
- Reduced overhead.
- Power by model simplicity.

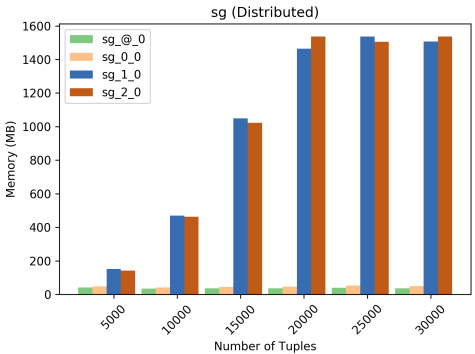
Runtime of Same Generation



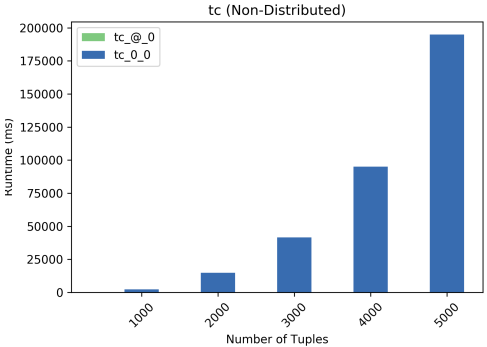


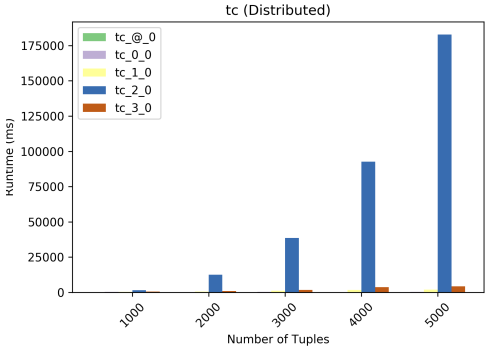
Memory of Same Generation



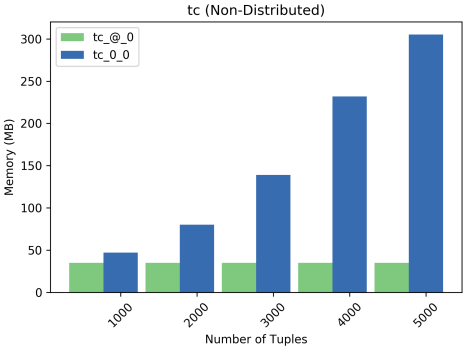


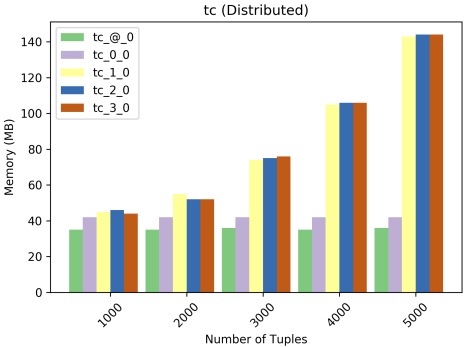
Runtime of Transitive Closure





Memory of Transitive Closure



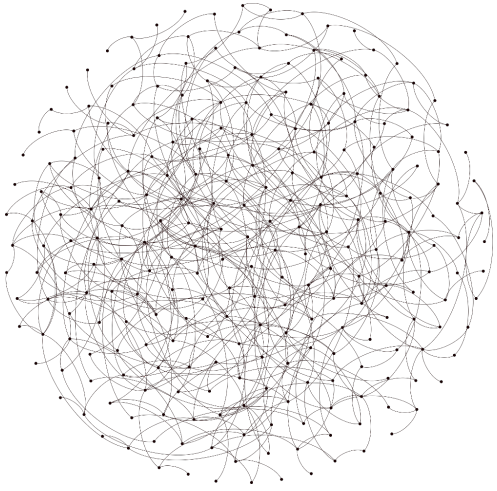


Experiments & Results

Same Generation	Exponential 2^{14}	Distributed	62.85s	1287MB	−32.22s	66%
		Non-Distributed	95.06s	1571MB	−284MB	82%
	Linear 25×10^3	Distributed	135.95s	1628MB	−87.13s	61%
		Non-Distributed	223.08s	1576MB	+52MB	103%
Transitive Closure	Exponential 2^{12}	Distributed	103.10s	192MB	−14.68s	88%
		Non-Distributed	117.78s	284MB	−92MB	68%
	Linear 5×10^3	Distributed	183.24s	222MB	−11.73s	94%
		Non-Distributed	194.97s	340MB	−118MB	65%

Summary of Findings

- Generally more performant in runtime & memory than Soufflé.
- Limited by resources available to AWS Lambda, Soufflé is not.
- Longest duration subprograms are greatest bottleneck.



https://github.com/souffle-lang/souffle

https://github.com/lyndonhenry/souffle-in-the-cloud