Vertex-centric Parallel Computation of SQL Queries

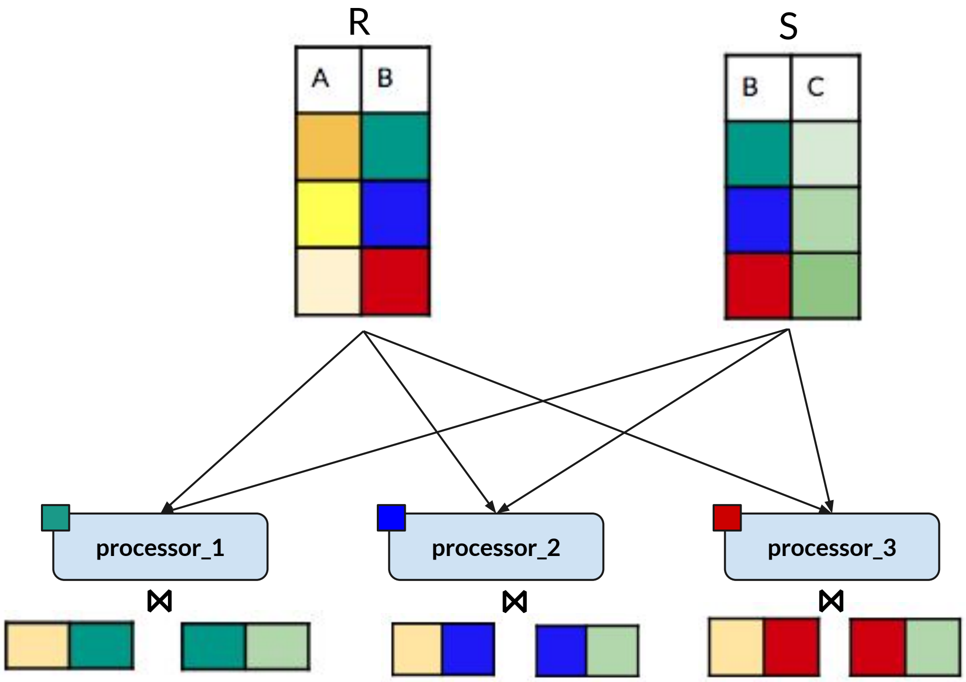
Ainur Smagulova
Factorized Databases Workshop
August 2022

Parallel Join Processing

- **Approach**
  1. Partition input on join attribute
  2. Each processor runs join independently
  3. Output is the union of each processor output

Issue: Need to reshuffle (re-hash) the input between individual join operations.

$Q(A,B,C) = R(A, B) \bowtie S(B, C)$



[DeWitt79], [Immerman87]

Parallel Join Processing: Approaches

Parallel Databases
(relational databases)

THE GAMMA DATABASE MACHINE PROJECT

TERADATA

VERTICA

Greenplum

Big Data Systems
(general purpose computation frameworks)

hadoop

HIVE

APACHE Spark

Google Dremel

Parallel Join Processing: Approaches

Parallel Databases
(relational databases)

THE GAMMA DATABASE MACHINE PROJECT

TERADATA

VERTICA

Greenplum

Graph Systems
(vertex-centric graph processing engines)

Pregel Google

APACHE GIRAPH

APACHE Spark GraphX

GraphLab

TigerGraph

PowerGraph

Big Data Systems
(general purpose computation frameworks)

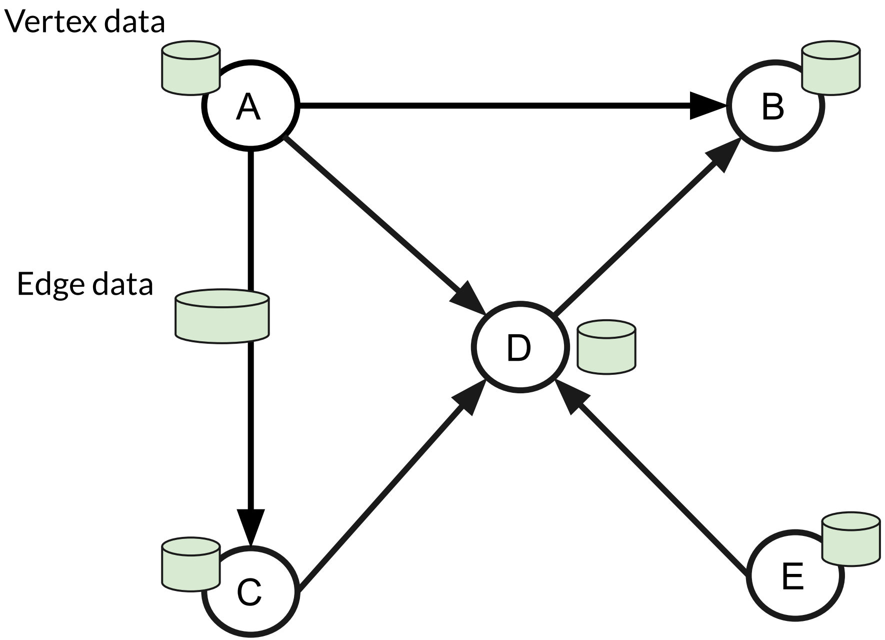
hadoop

HIVE

APACHE Spark

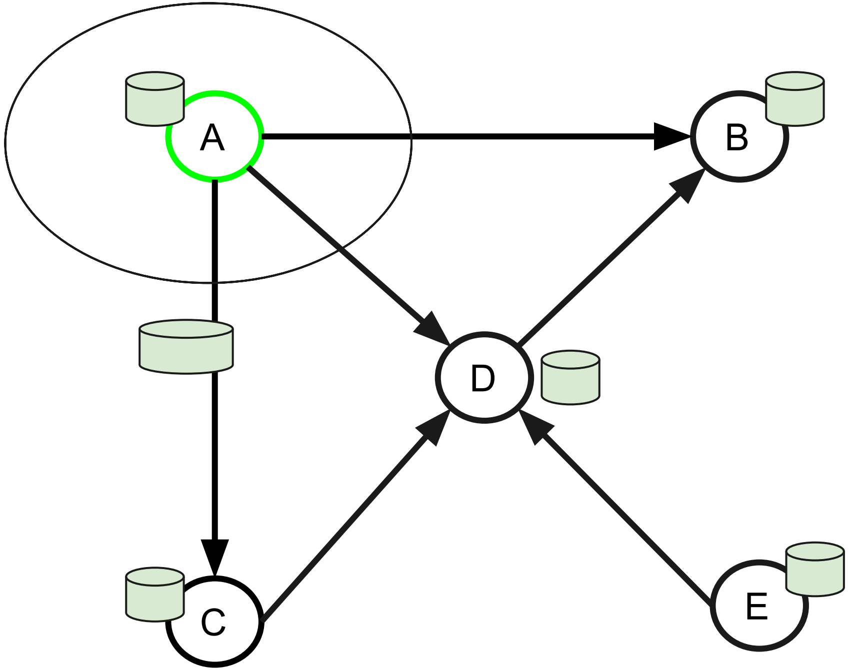
Google Dremel

Vertex-centric BSP Computational Model



- adaptation of Bulk Synchronous Parallel Model (BSP) [Valiant90] to graph data

Vertex-centric BSP Computational Model



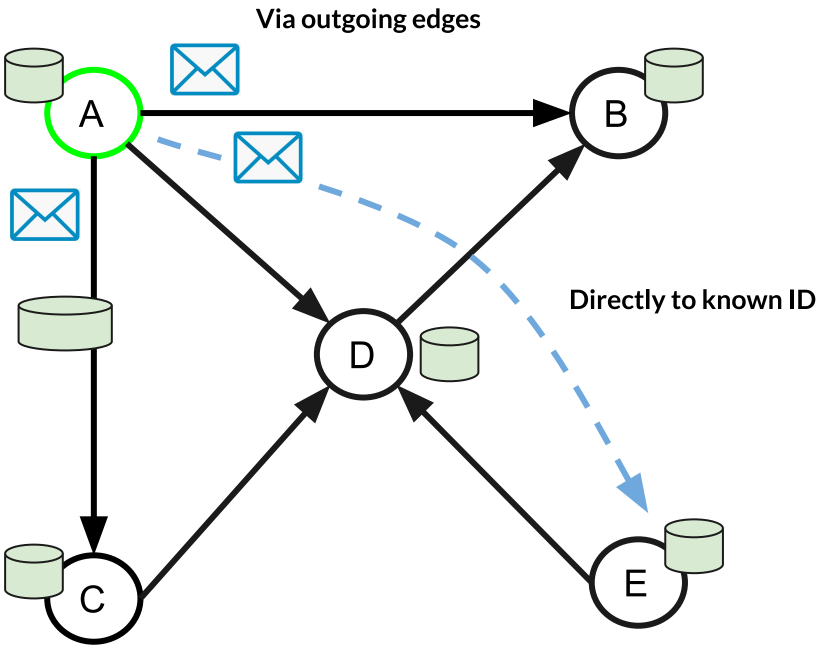
Computation consists of supersteps.

At each superstep each active vertex:

I - Local computation/vertex program

|| - Communication via message passing

Vertex-centric BSP Computational Model



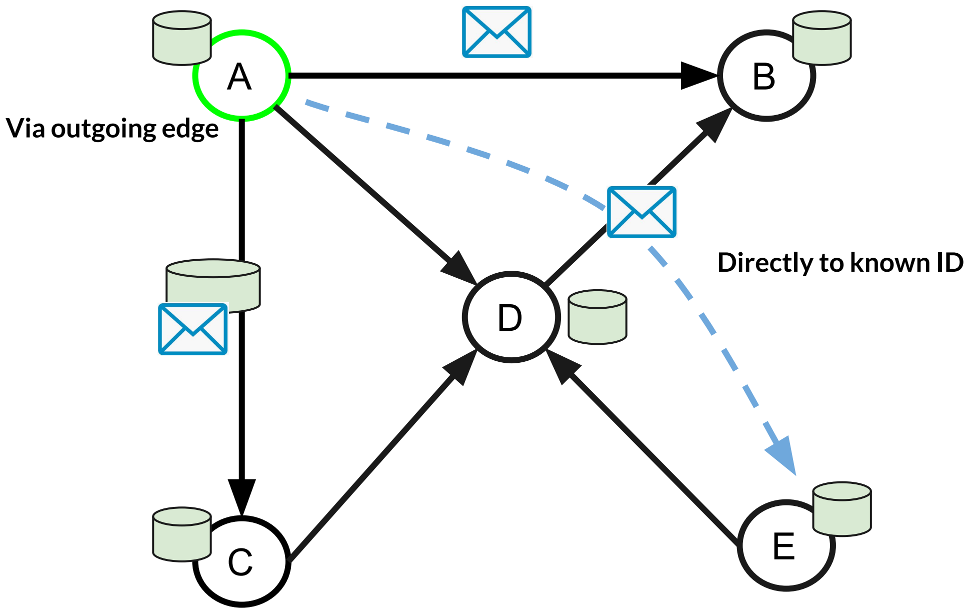
At each superstep each active vertex:

I - Local computation/vertex program

II - Communication via message passing

1. Via outgoing edges
2. Via direct known ID

Vertex-centric BSP Computational Model



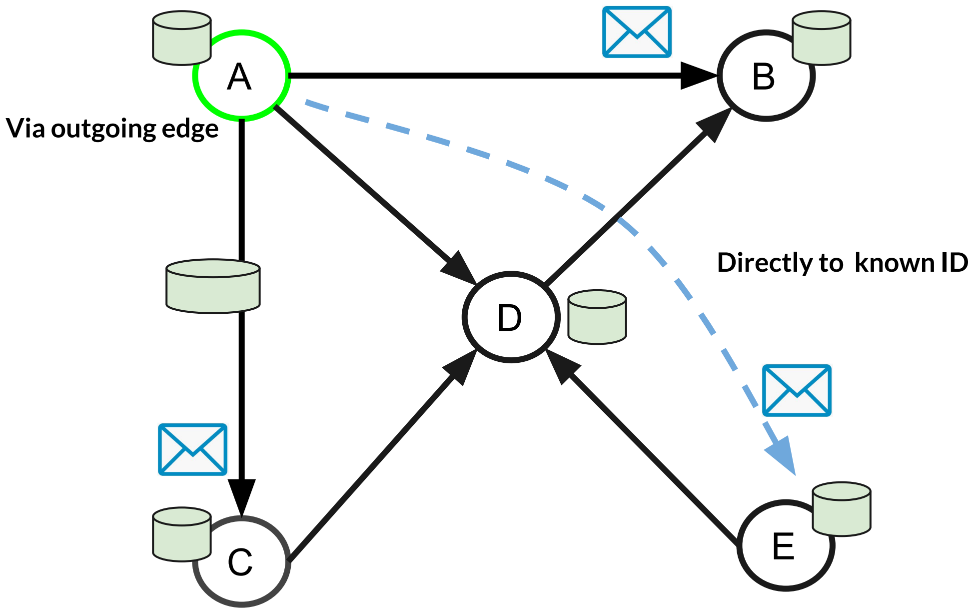
At each superstep each active vertex:

I - Local computation/vertex program

II - Communication via message passing

1. Via outgoing edges
2. Via direct known ID

Vertex-centric BSP Computational Model



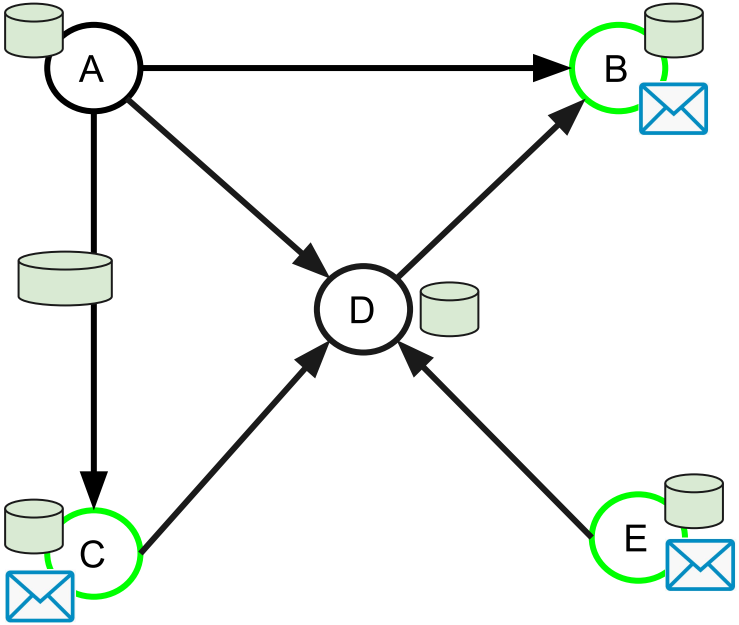
At each superstep each active vertex:

I - Local computation/vertex program

|| - Communication via message passing

1. Via outgoing edges
2. Via direct known ID

Vertex-centric BSP Computational Model



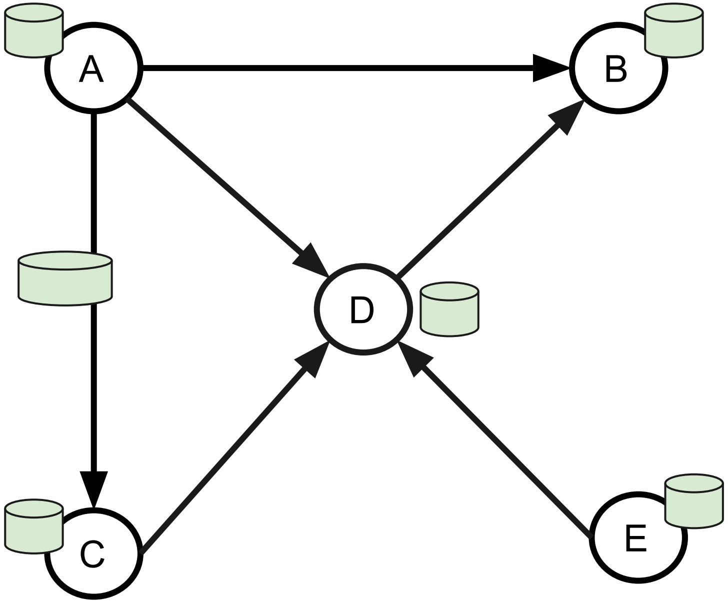
New superstep begins. Vertices receive messages sent during previous superstep.

Superstep:

I - Local computation/vertex program

|| - Communication via message passing

Vertex-centric BSP Computational Model



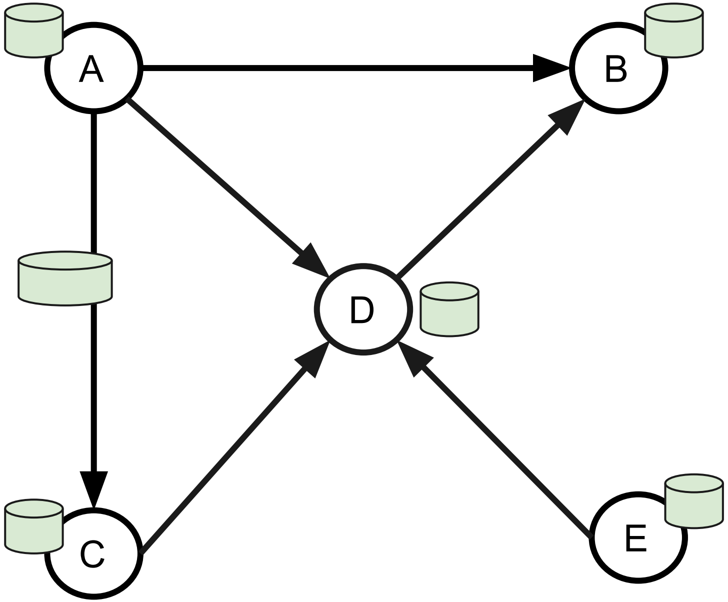
Computation terminates:

No messages in transit

No active vertices

The result is the union of outputs computed by vertices.

Complexity measures



- Total Communication Cost: O(#msg)
- Total Computation Cost: O(#msg)
- Number of rounds: O(|query|) = O(1)

We show that:

Vertex-centric parallelism is extremely well-suited to compute SQL queries with provable theoretical guarantees and good performance as validated by our experiments.

Our solution comprises:

(i) **Tuple-Attribute Graph (TAG) data model**

- a graph encoding of a relational db

(ii) **vertex-centric TAG-join algorithm**

- communication and computation complexities are competitive with the best-known parallel join algorithms
- avoids the relation reshuffling (rehashing or resorting) between individual join operations

Encoding relations as a graph

NATION

N_NATIONKEY	N_NAME
1	USA
2	France

CUSTOMER

C_CUSTKEY	C_NATIONKEY	C_NAME
10	1	Bob
2	2	Emma

ORDER

O_ORDERKEY	O_CUSTKEY	O_ORDERDATE
11	10	1998-05-01
2	2	1998-05-01

Encoding relations as a graph

NATION

N_NATIONKEY	N_NAME
1	USA
2	France

Tuple vertex: Each tuple (row) maps to a vertex

- Label of a tuple vertex corresponds to the name of the relation.

Encoding relations as a graph

NATION

N_NATIONKEY	N_NAME
1	USA
2	France

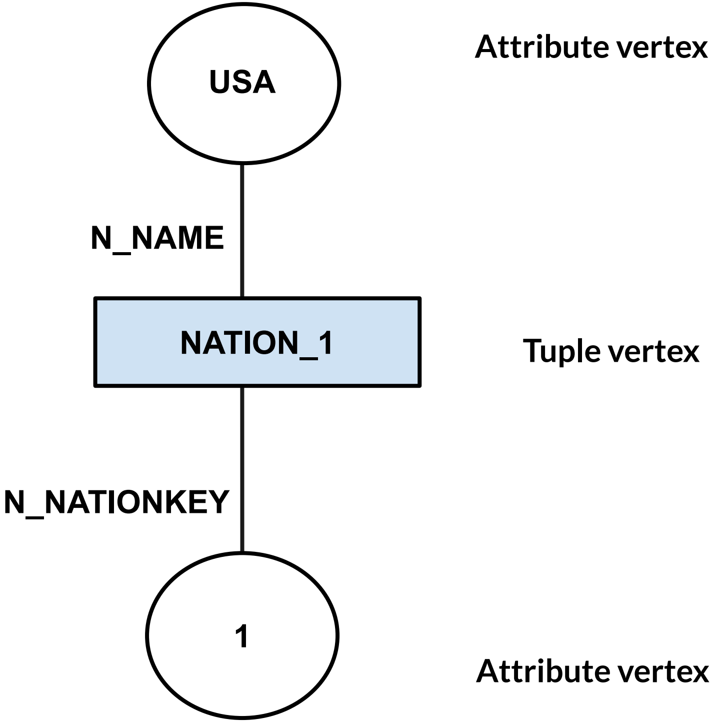
Attribute vertex: Each attribute value maps to a vertex

- Label of an attribute vertex matches the data type of the corresponding attribute.

Encoding relations as a graph

NATION

N_NATIONKEY	N_NAME
1	USA
2	France

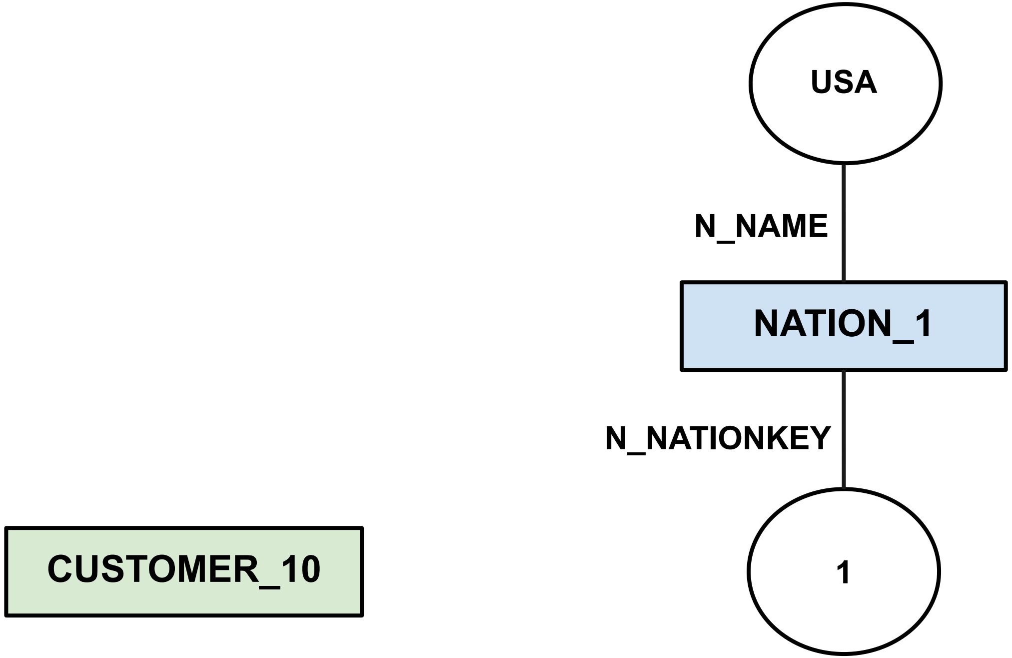


Edge: between tuple vertex and its attribute vertices

- Label of an edge matches the corresponding attribute name.

CUSTOMER

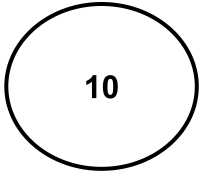
C_CUSTKEY	C_NATIONKEY	C_NAME
10	1	Bob
2	2	Emma

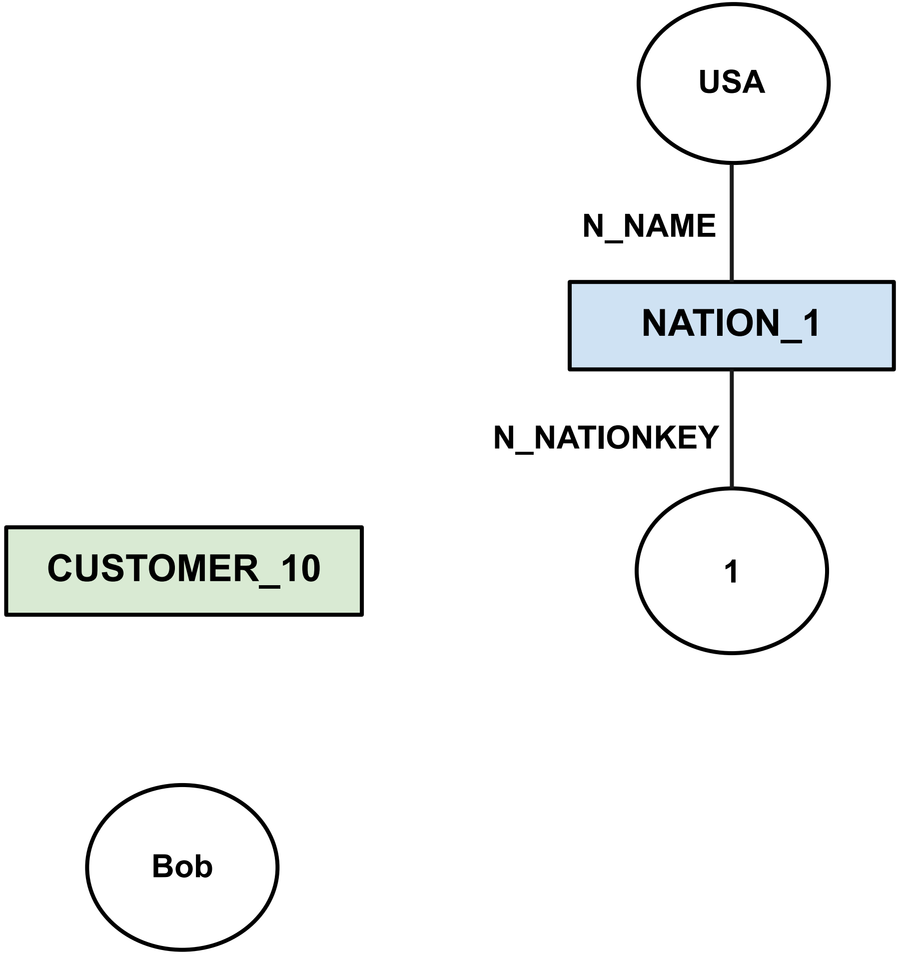


Map tuple to a tuple vertex “CUSTOMER_10

CUSTOMER

C_CUSTKEY	C_NATIONKEY	C_NAME
10	1	Bob
2	2	Emma

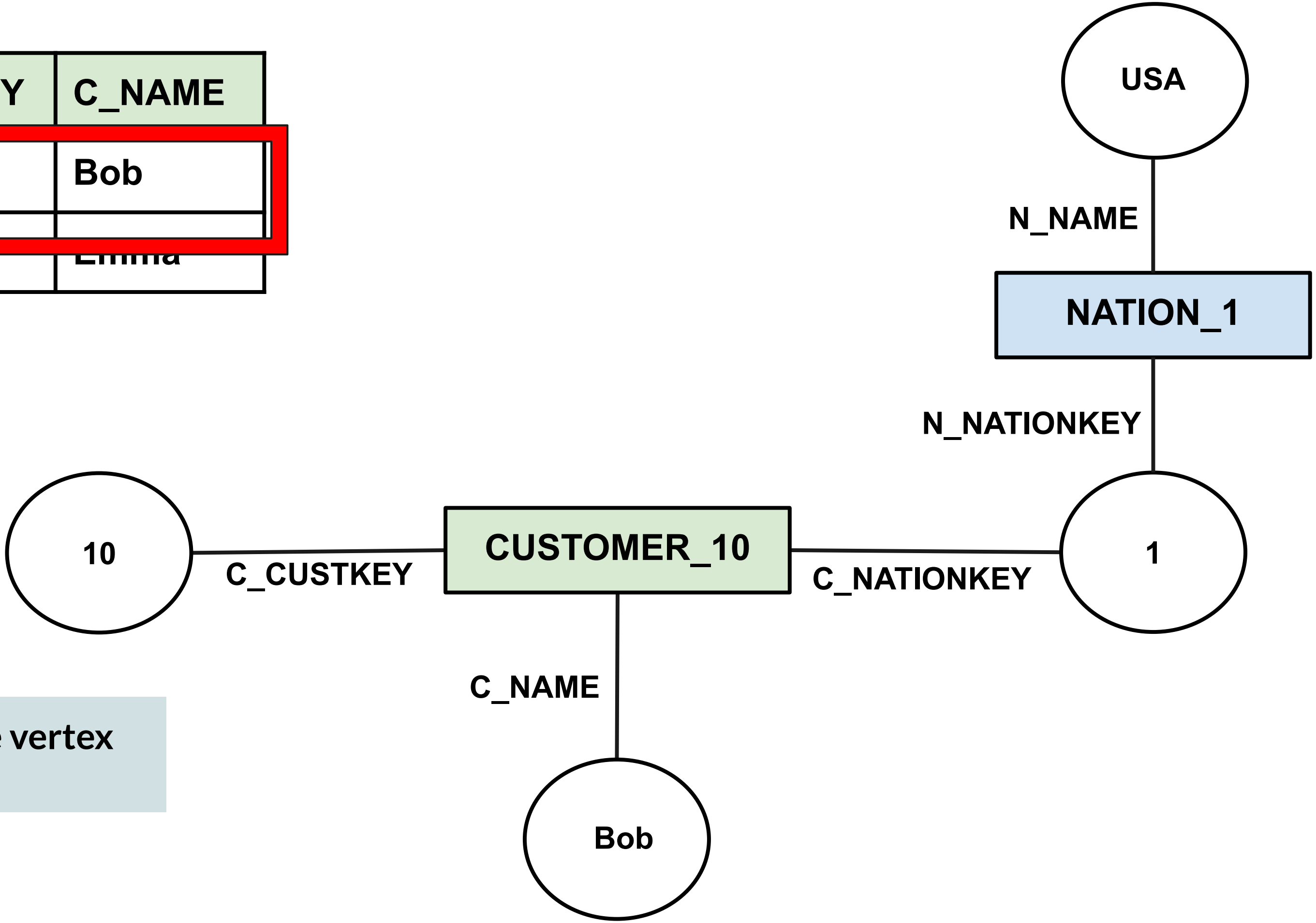




Map each attribute value to attribute vertices

CUSTOMER

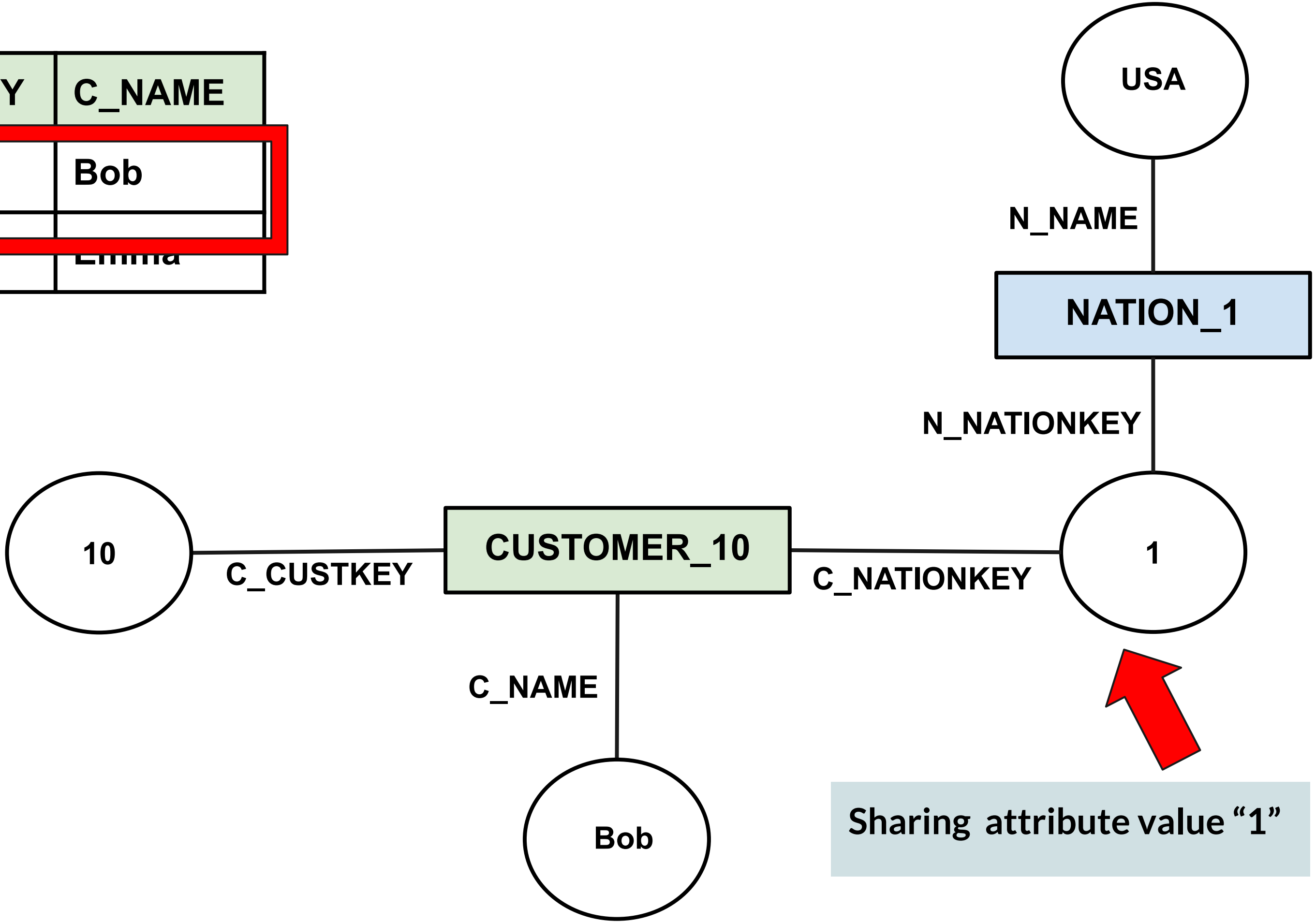
C_CUSTKEY	C_NATIONKEY	C_NAME
10	1	Bob
2	2	Emma



Add edges between tuple vertex and its attribute vertices

CUSTOMER

C_CUSTKEY	C_NATIONKEY	C_NAME
10	1	Bob
2	2	Emma



NATION

N_NATIONKEY	N_NAME
1	USA
2	France

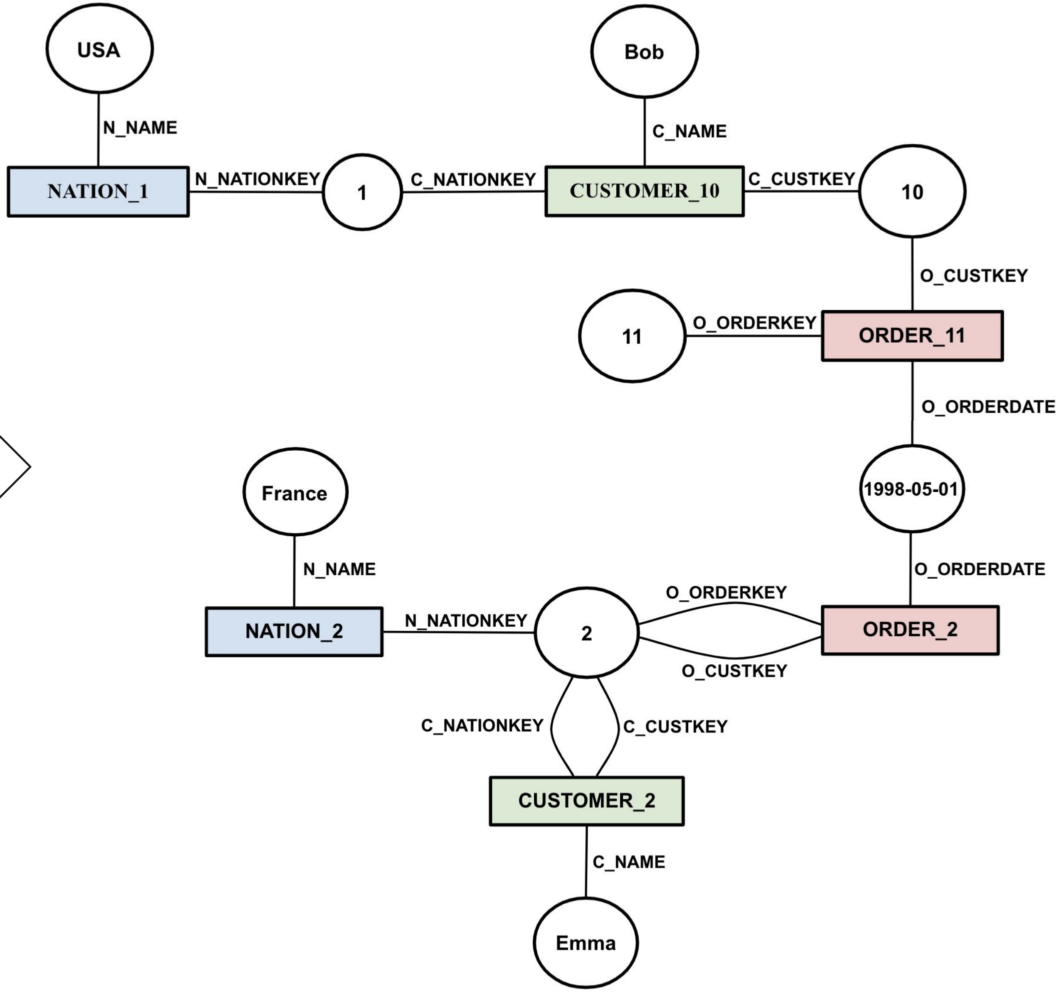
CUSTOMER

C_CUSTKEY	C_NATIONKEY	C_NAME
10	1	Bob
2	2	Emma

ORDER

O_ORDERKEY	O_CUSTKEY	O_ORDERDATE
11	10	1998-05-01
2	2	1998-05-01

Relational Data Instance



Tuple-Attribute Graph Instance

NATION

N_NATIONKEY	N_NAME
1	USA
2	France

CUSTOMER

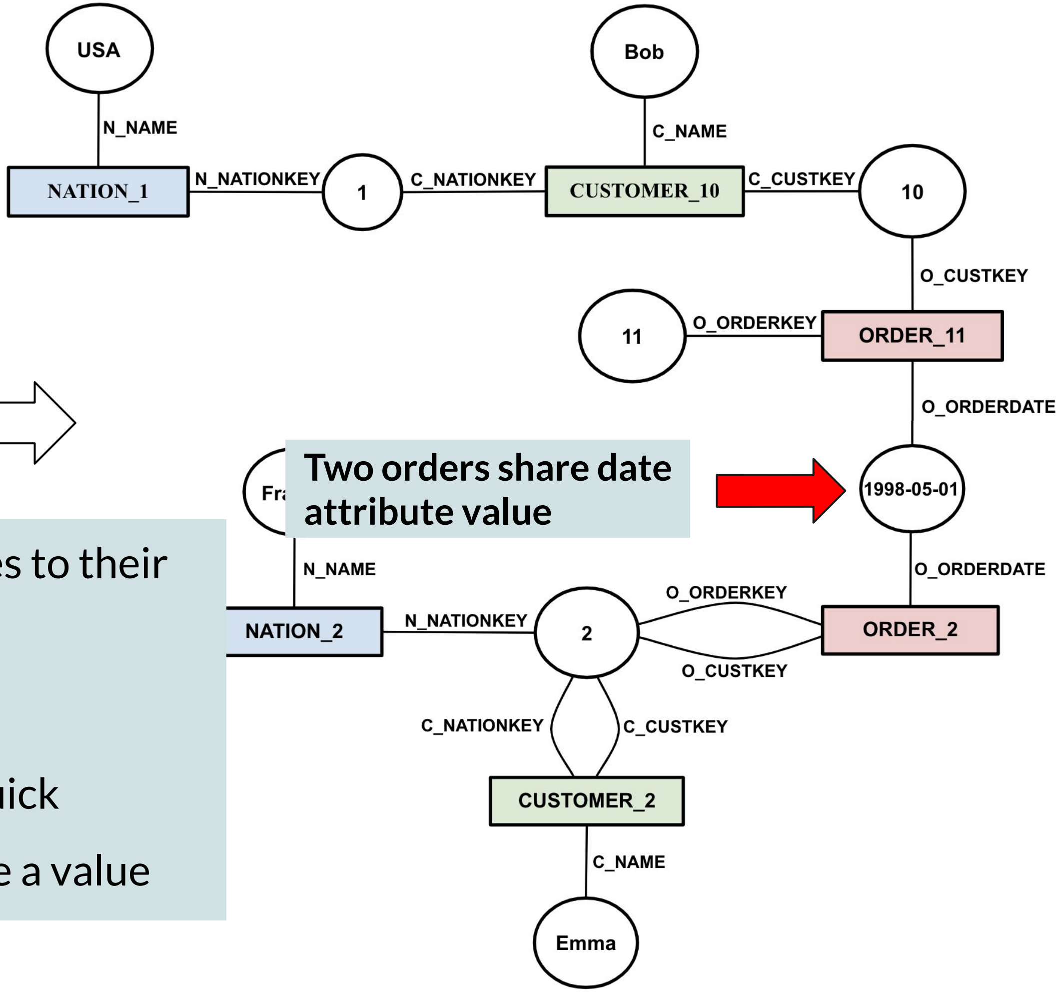
C_CUSTKEY	C_NATIONKEY	C_NAME
10	1	Bob
2	[illegible]	[illegible]

O_O[illegible]
11
2

Tuples are explicitly joined via edges to their join attribute values.

- **not limited to PK-FK joins**
- **Implicit indexing scheme** = quick navigation to tuples that share a value

Relational Data Instance

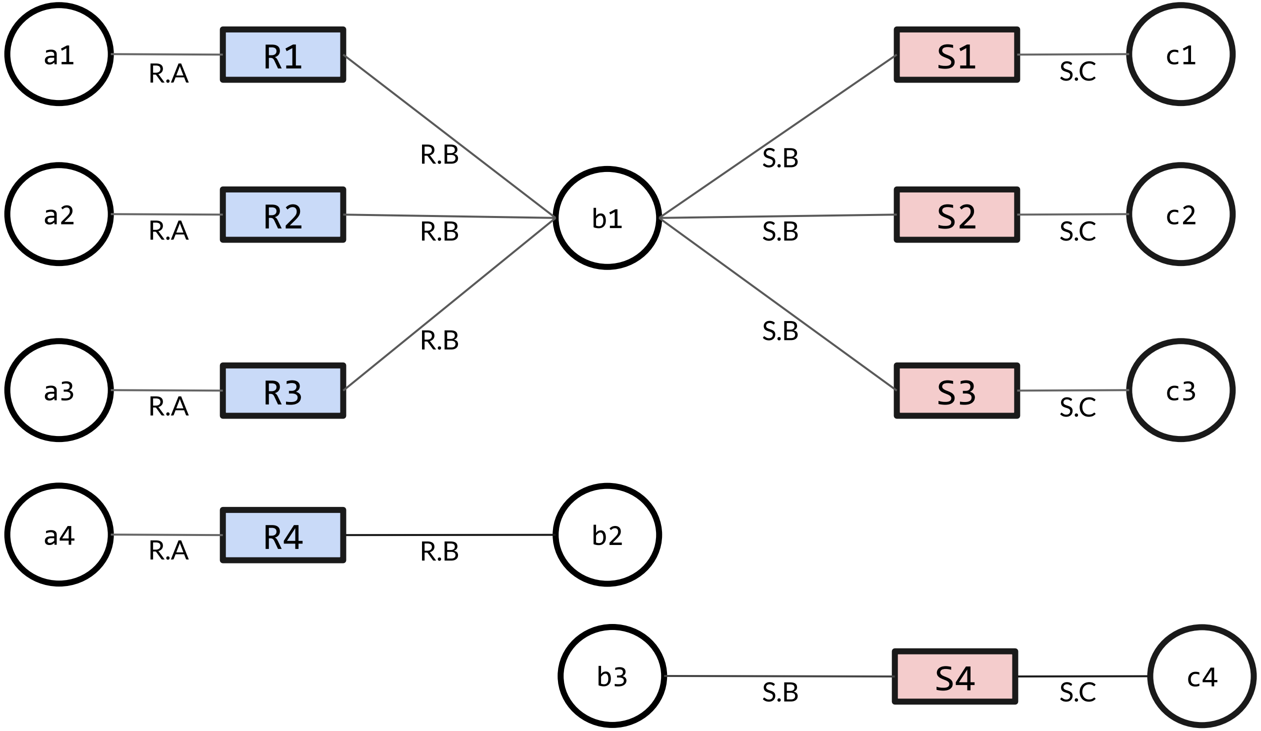


Two orders share date attribute value

Tuple-Attribute Graph Instance

Check join conditions in parallel: 2-way join example

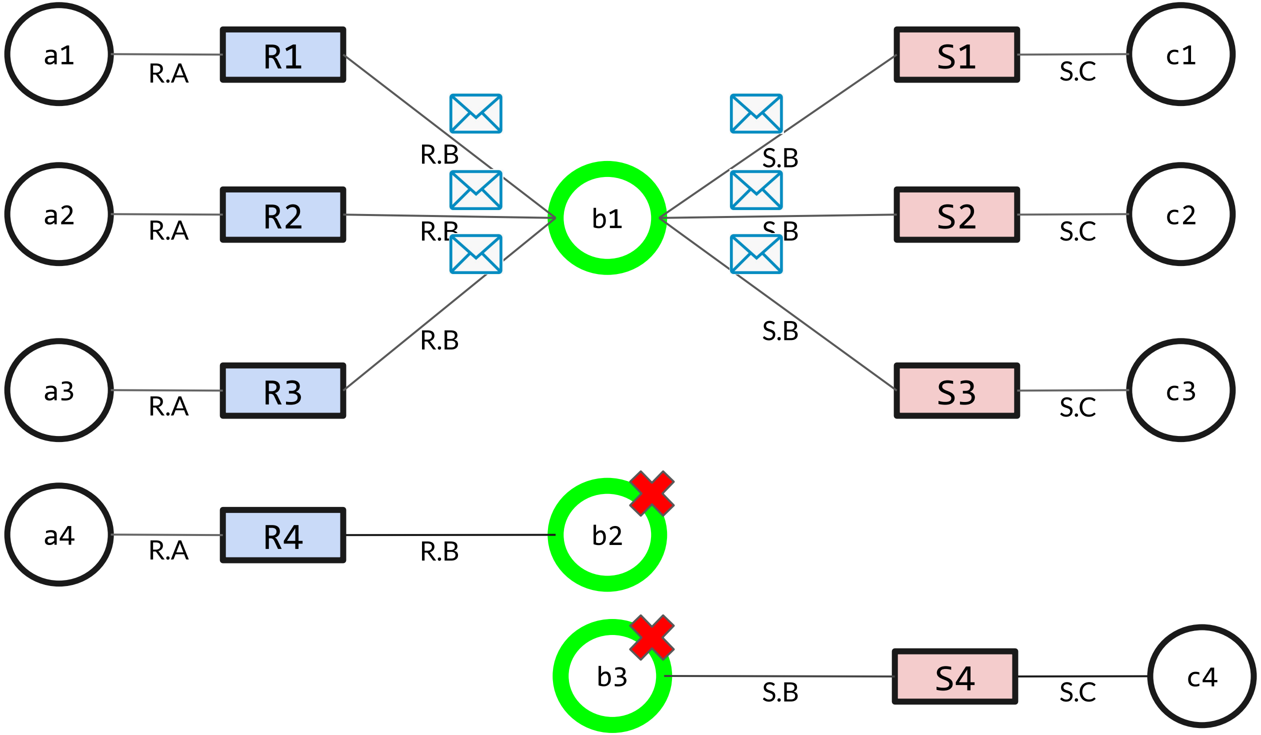
R(A, B) ⋈ S(B, C)



Check join conditions in parallel: 2-way join example

R(A, B) ⋈ S(B, C)

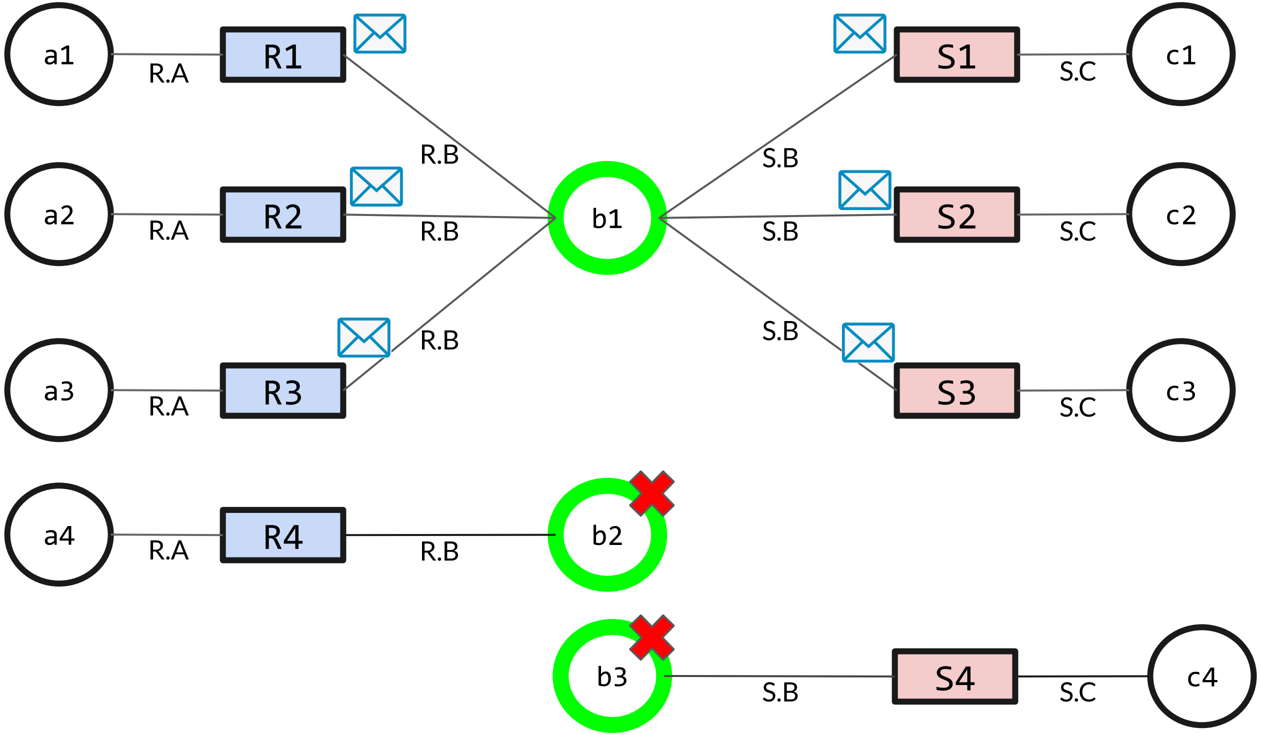
Superstep 1



Check join conditions in parallel: 2-way join example

R(A, B) ⋈ S(B, C)

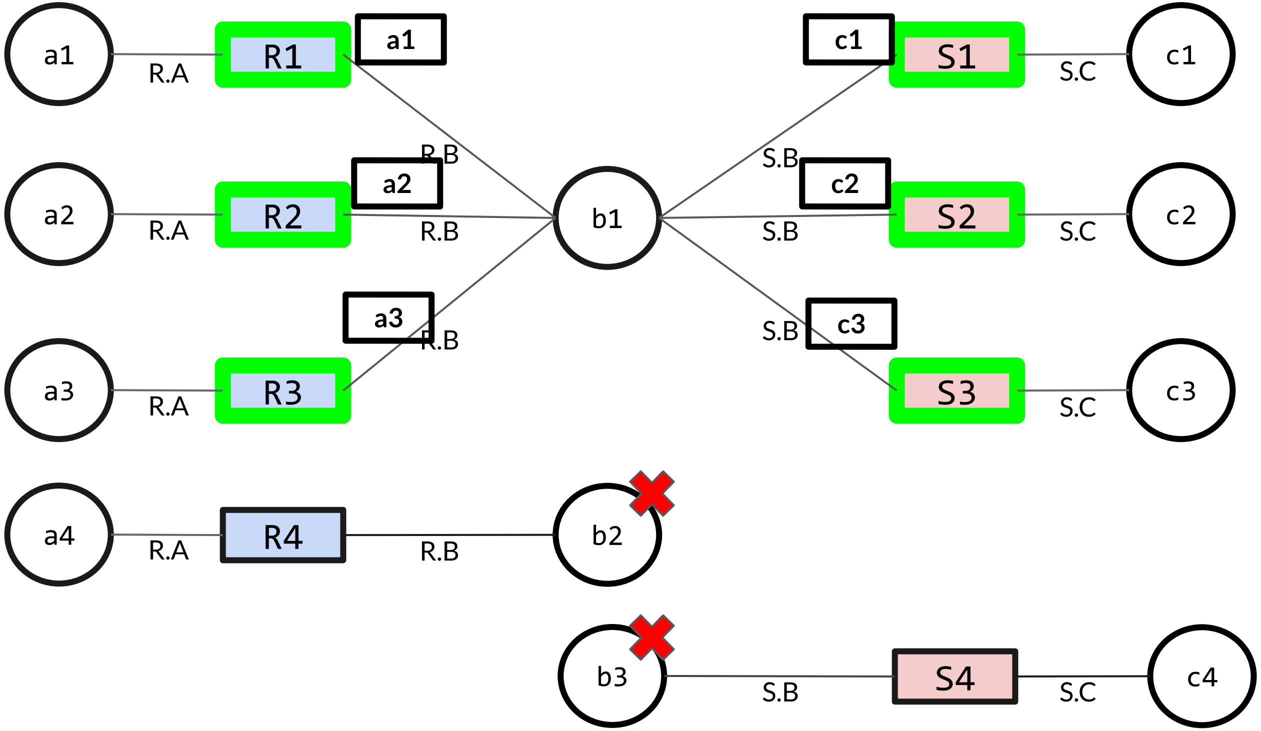
Superstep 1



Check join conditions in parallel: 2-way join example

R(A, B) ⋈ S(B, C)

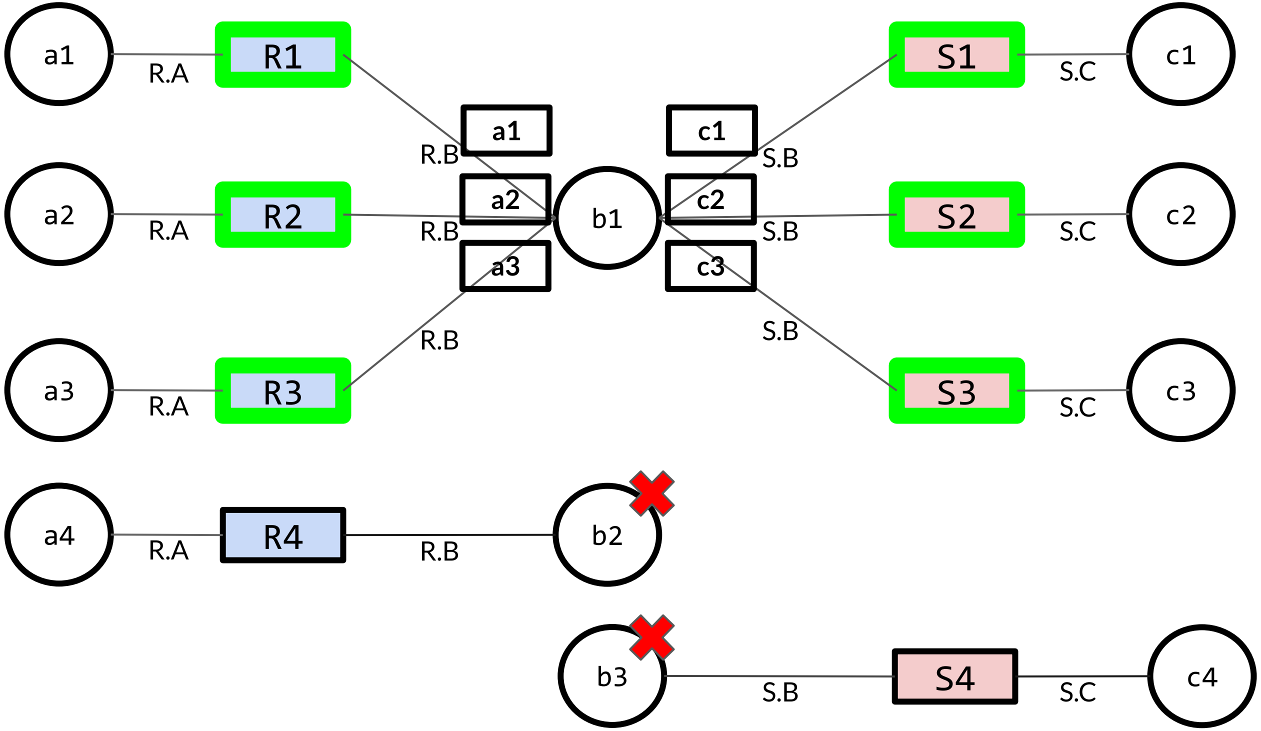
Superstep 2



Check join conditions in parallel: 2-way join example

R(A, B) ⋈ S(B, C)

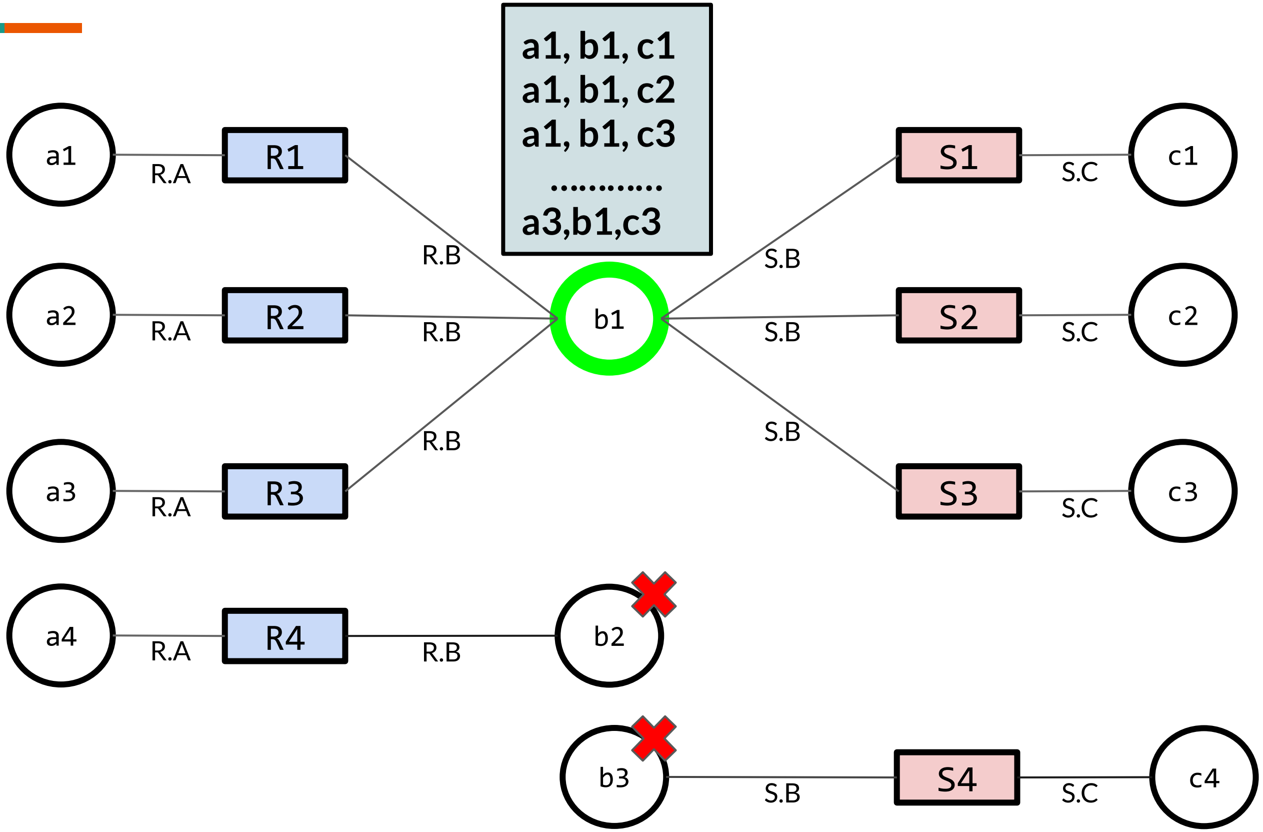
Superstep 2



Check join conditions in parallel: 2-way join example

$R(A, B) \bowtie S(B, C)$

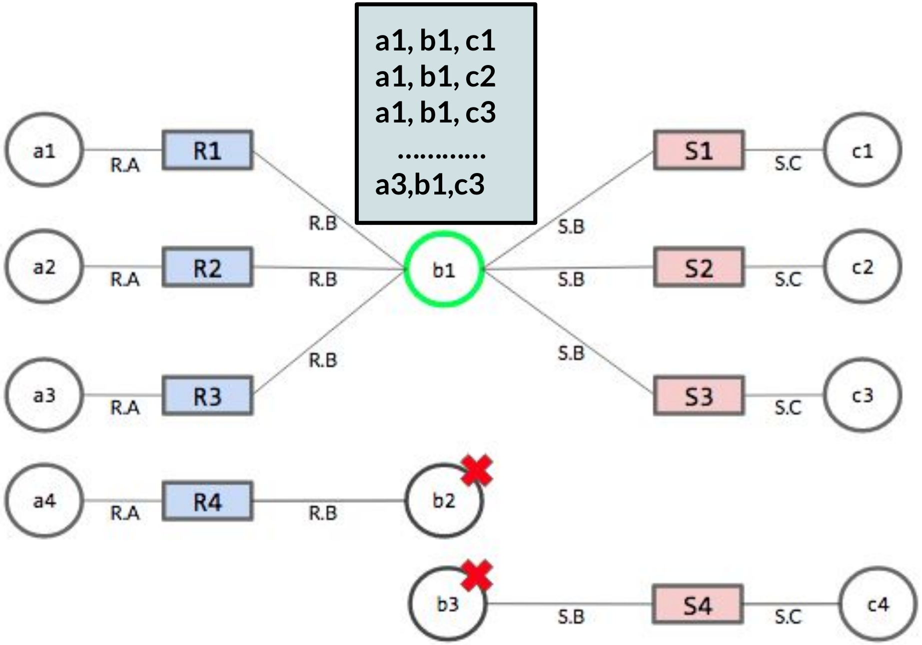
Superstep 3



2-way join example: cost analysis

At superstep 1 and 2

- Computation: O(IN)
- Communication: O(#msg) <= |R| + |S| = O(IN)
    - O(#msg) <= O(min(IN, OUT))



|R| - #tuples in relation R
|S| - #tuples in relation S
IN - #tuples in the input
OUT -#tuples in output

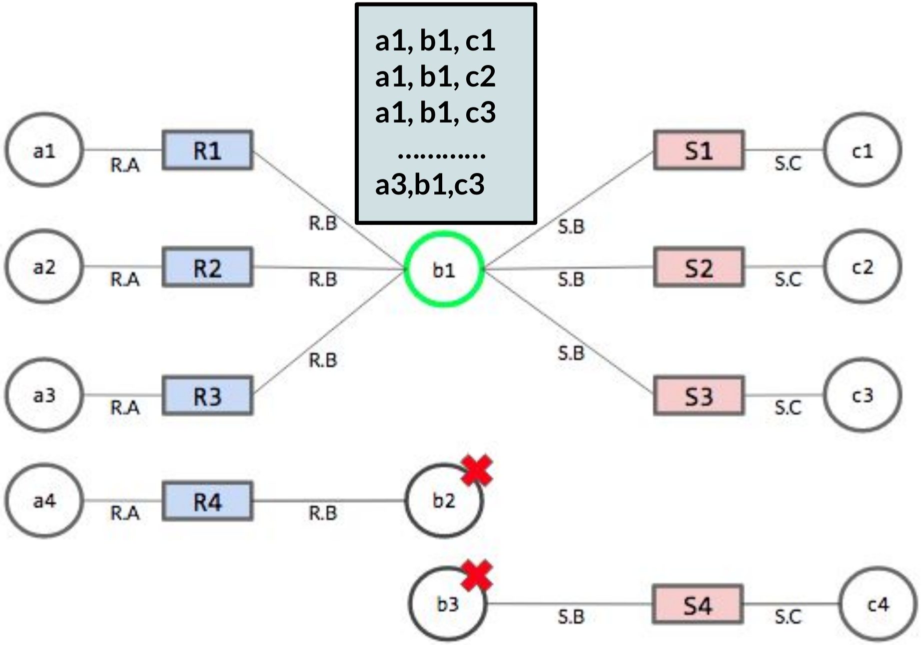
2-way join example: cost analysis

At superstep 1 and 2

- Computation: O(IN)
- Communication: O(#msg) <= |R| + |S| = O(IN)
  - O(#msg) <= O(min(IN, OUT))

At superstep 3: computing output

- Computation: O(#msg) = O(OUT)
- Communication: O(#msg) = O(OUT)



|R| - #tuples in relation R
|S| - #tuples in relation S
IN - #tuples in the input
OUT -#tuples in output

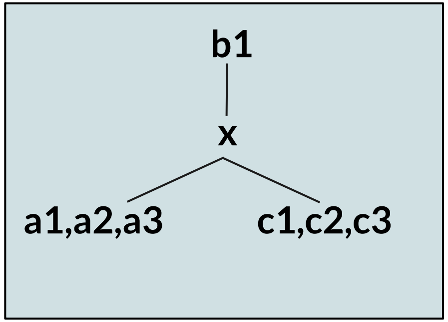
Compact Representation of the Output

Flat representation of join result

a1, b1, c1
a1, b1, c2
a1, b1, c3
....
a3, b1, c3

$OUT <= |R| * |S| = O(IN^2)$

Factorized representation of join result



$F_{OUT} <= |R| + |S| = O(IN)$

Factorizing join result [Otleanu’12,15,16]

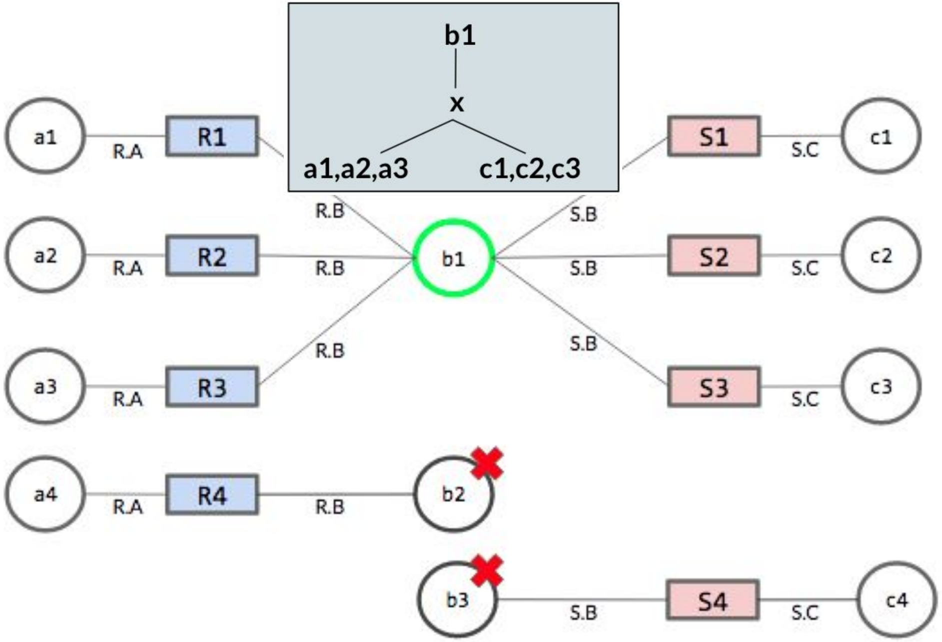
2-way join example: cost analysis with factorization

At each superstep 1 and 2

- Computation: O(IN)
- Communication: O(#msg) <= |R| + |S| = O(IN)
  - O(#msg) <= O(min(IN, OUT))

At superstep 3: computing output

- Communication: O(#msg**)** = |R| + |S|= O(IN)
- Computation: O(#msg) = |R| + |S| = O(IN)



|R| - #tuples in relation R
|S| - #tuples in relation S
IN - #tuples in the input
OUT -#tuples in output

2-way join: main result

Any 2-way join query can be computed by a vertex-centric algorithm with **O(IN + OUT)** communication* and computation cost.

- **A factorized representation** of a 2-way join can be computed with **O(IN)** cost.

*assuming output tuples are sent to one location

Vertex-centric Acyclic Join Algorithm

Input: TAG traversal plan (to guide the graph traversal)

Algorithm (two phases):

1. **Reduction***: mark the edges that connect tuple and attribute vertices that contribute to the join.

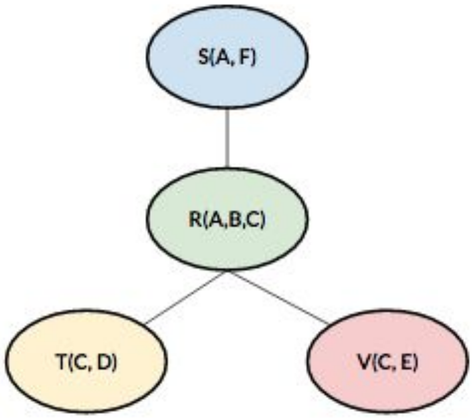
2. **Collection**: traverse the marked subgraph to collect the actual join result

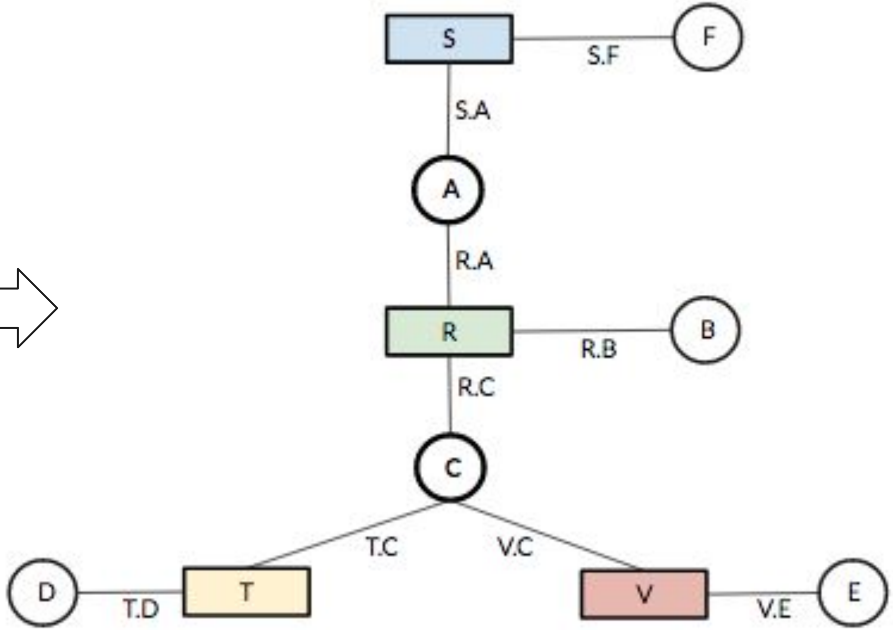
Output: union of vertex join results

*Semi-join reduction technique used in databases. [Bernstein81, Yannakakis81]

TAG plan construction

Q = S(**A**, F) ⋈ R(**A**, B, **C**) ⋈ T(**C**, D) ⋈ V(**C**, E)





Join tree

GHD = generalized hypertree decomposition

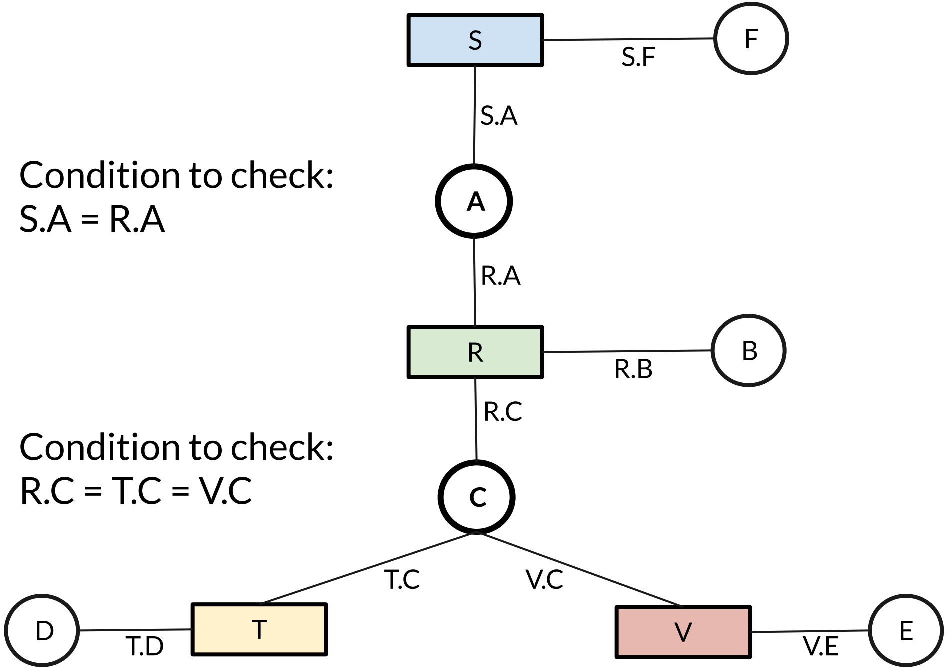
TAG traversal plan

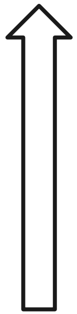
Traversal Plan

$Q = S(A, F) \bowtie R(A, B, C) \bowtie T(C, D) \bowtie V(C, E)$

Condition to check:
S.A = R.A

Condition to check:
R.C = T.C = V.C

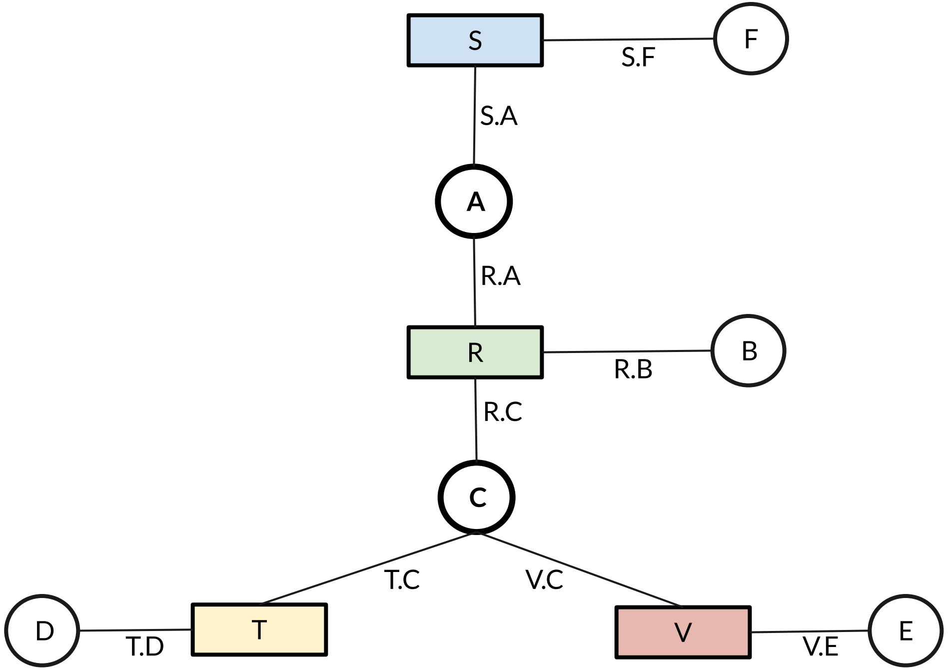


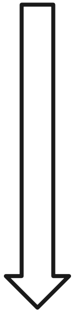


Reduction phase: bottom-up direction

Traversal Plan

$Q = S(A, F) \bowtie R(A, B, C) \bowtie T(C, D) \bowtie V(C, E)$

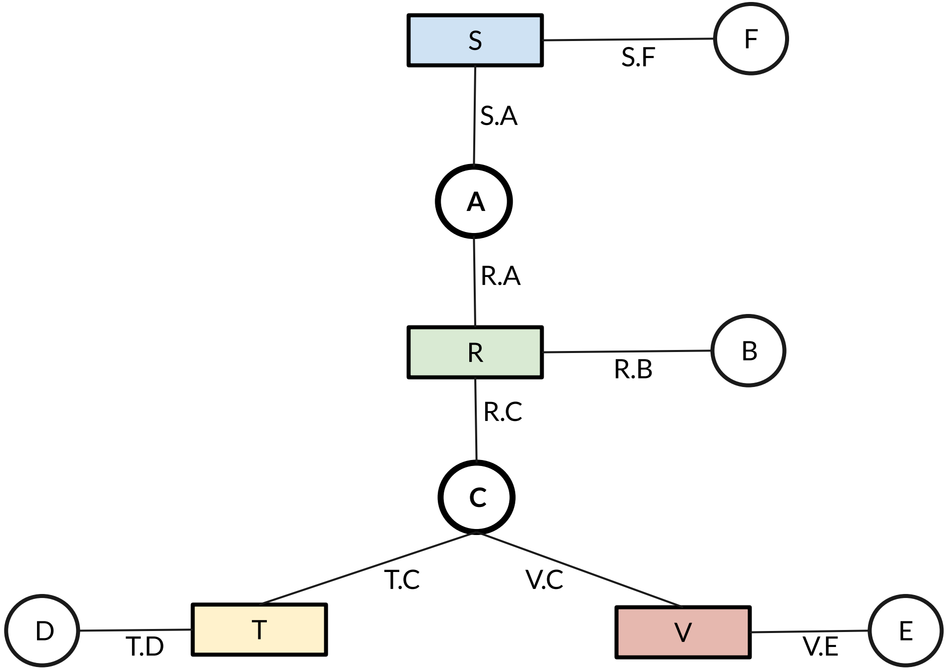


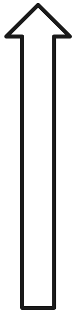


Reduction phase: top-down direction

Traversal Plan

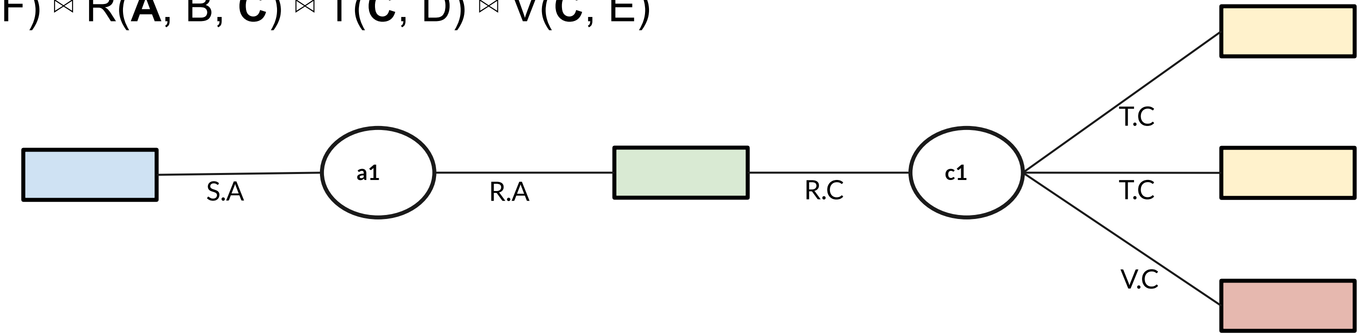
$Q = S(A, F) \bowtie R(A, B, C) \bowtie T(C, D) \bowtie V(C, E)$

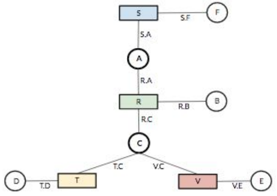




Collection phase: bottom-up direction

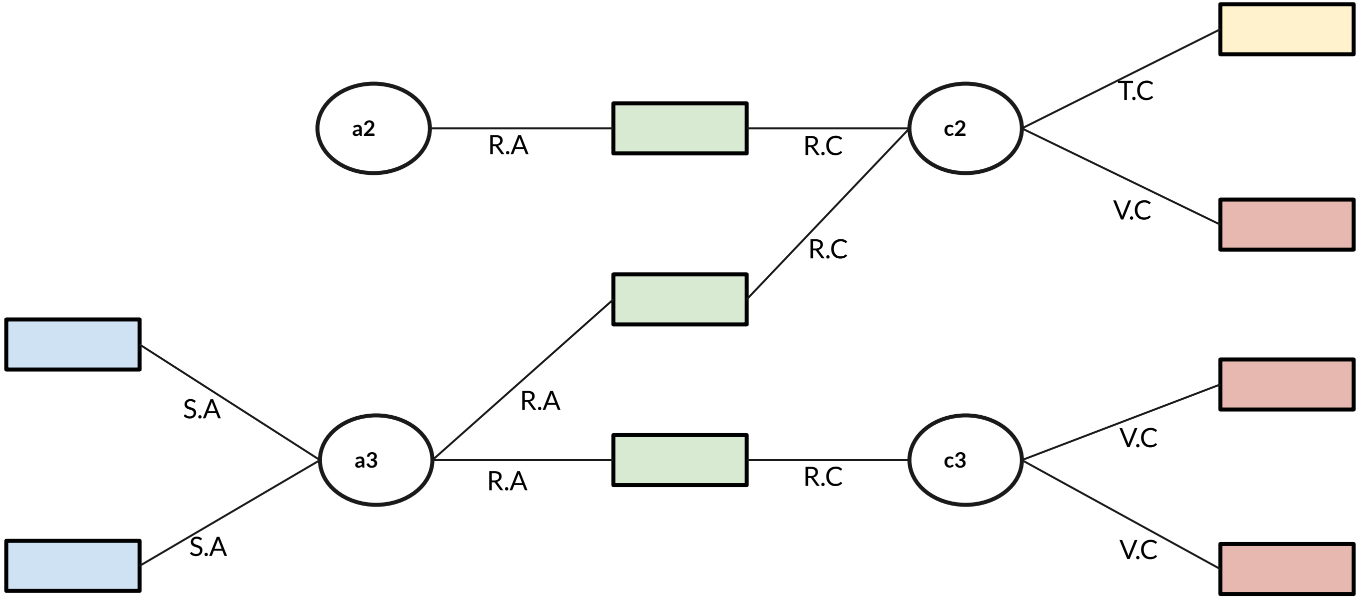
$Q = S(\mathbf{A}, F) \bowtie R(\mathbf{A}, B, \mathbf{C}) \bowtie T(\mathbf{C}, D) \bowtie V(\mathbf{C}, E)$



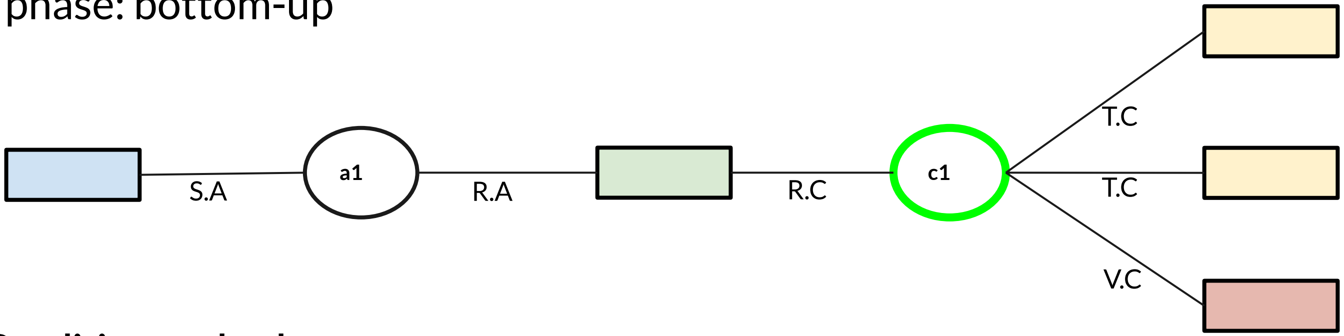


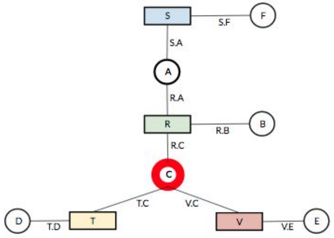
Traversal plan

TAG encoding of relations S, R, T and V

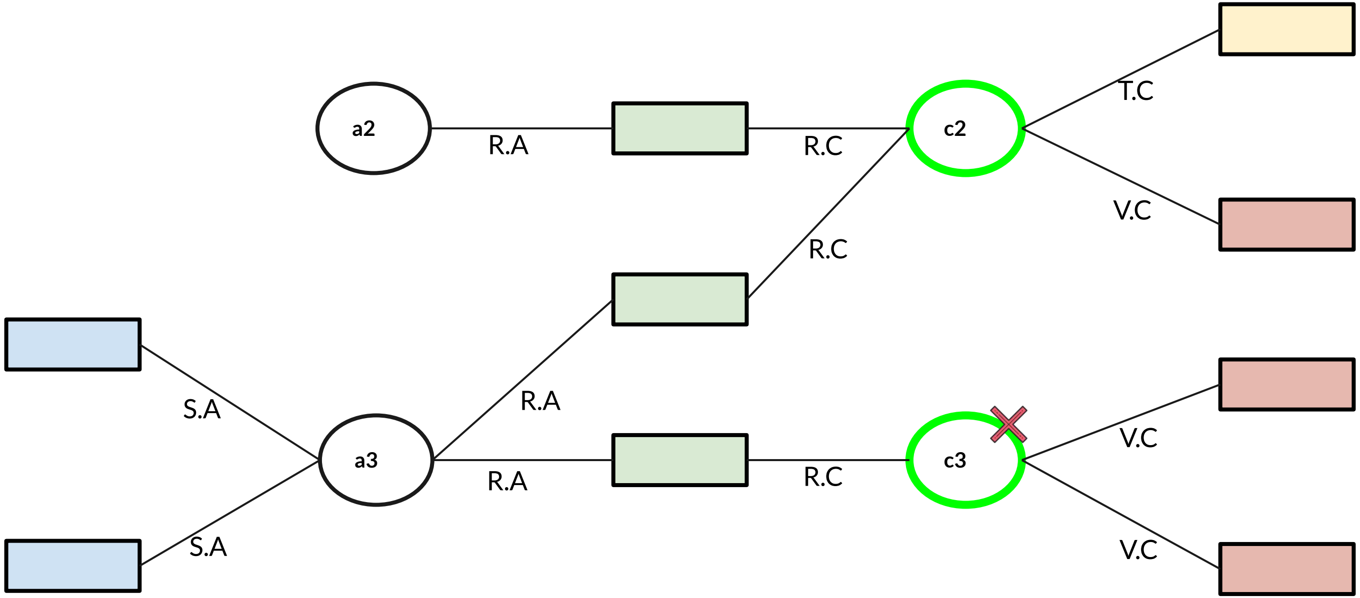


Reduction phase: bottom-up

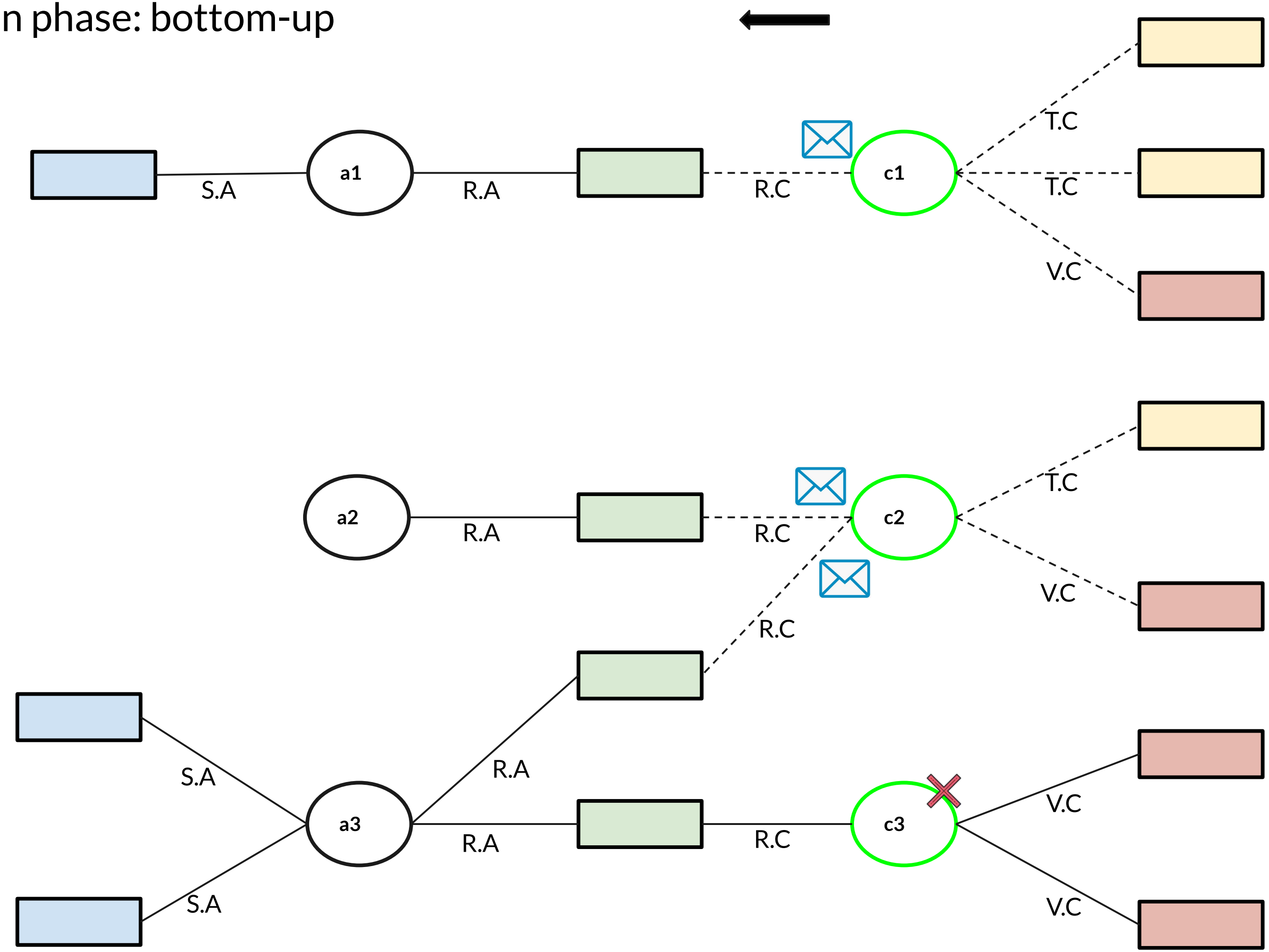


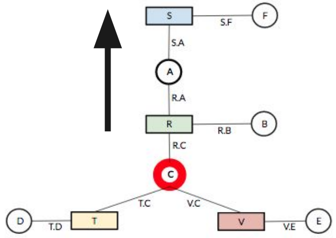


Condition to check:
R.C = T.C = V.C

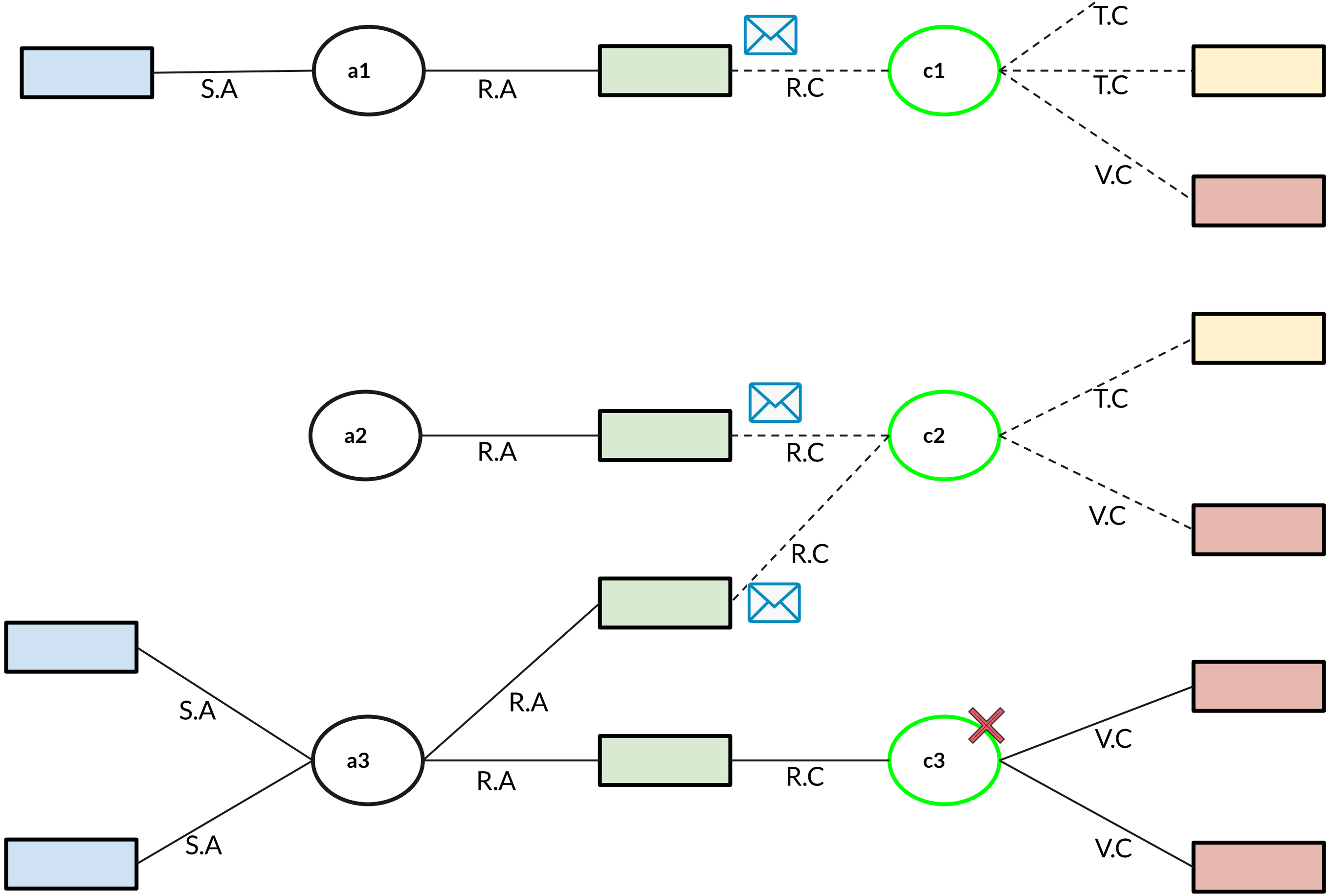


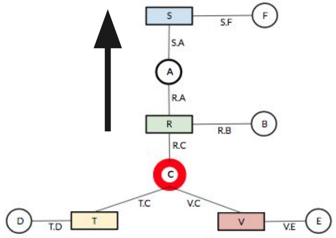
Reduction phase: bottom-up



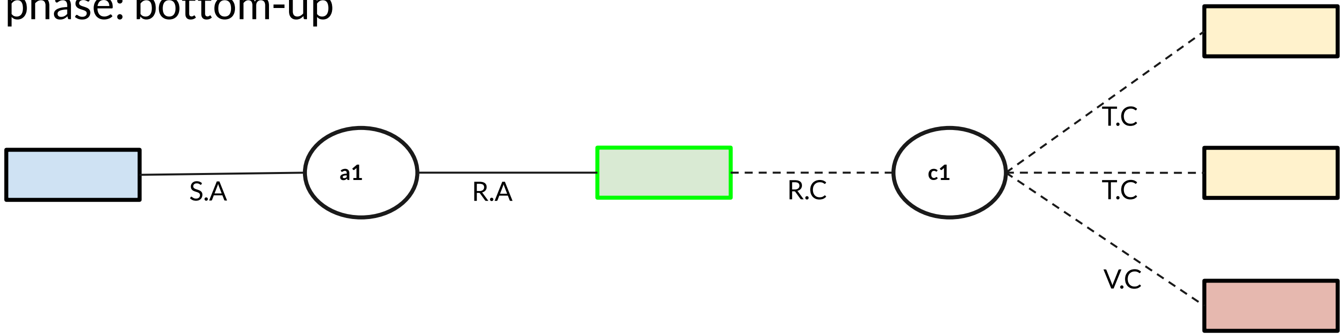


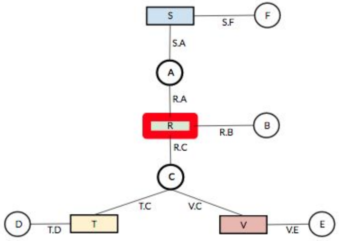
Reduction phase: bottom-up



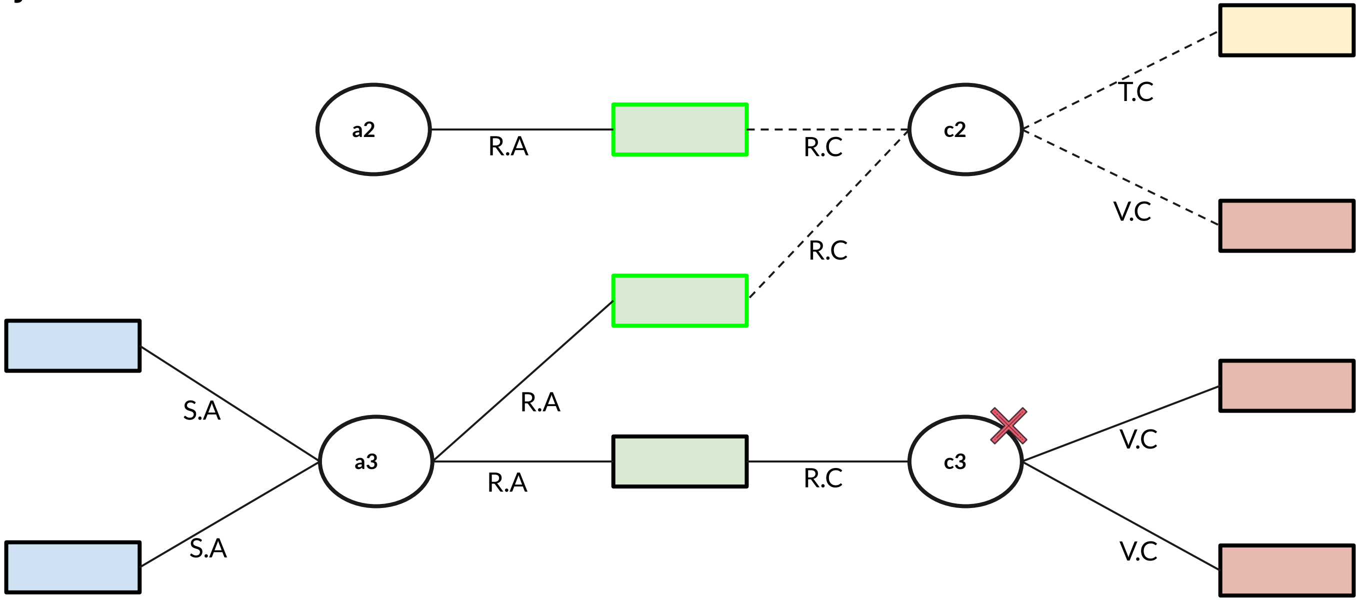


Reduction phase: bottom-up

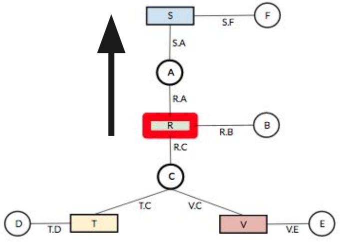


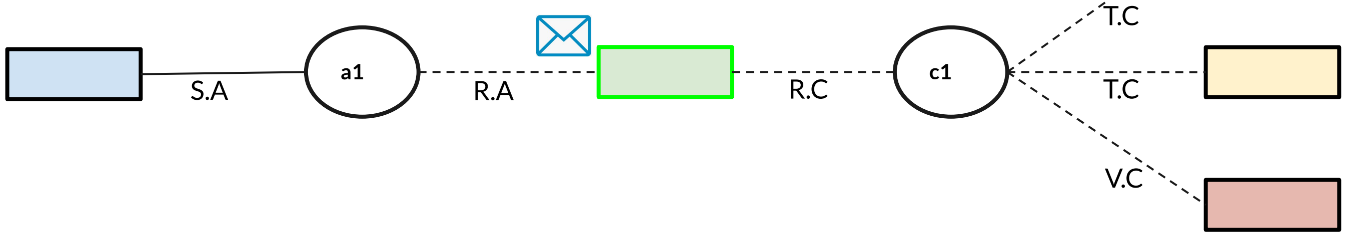


Propagate “wake up” message to the next join attribute vertex

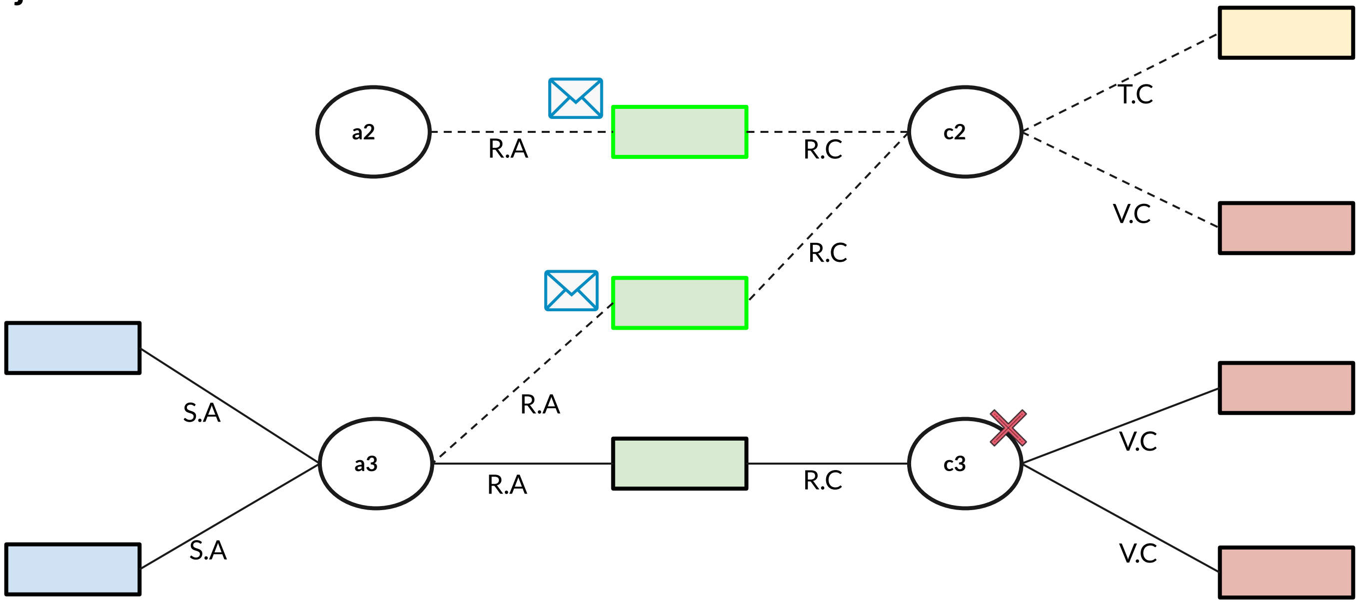


Reduction phase: bottom-up

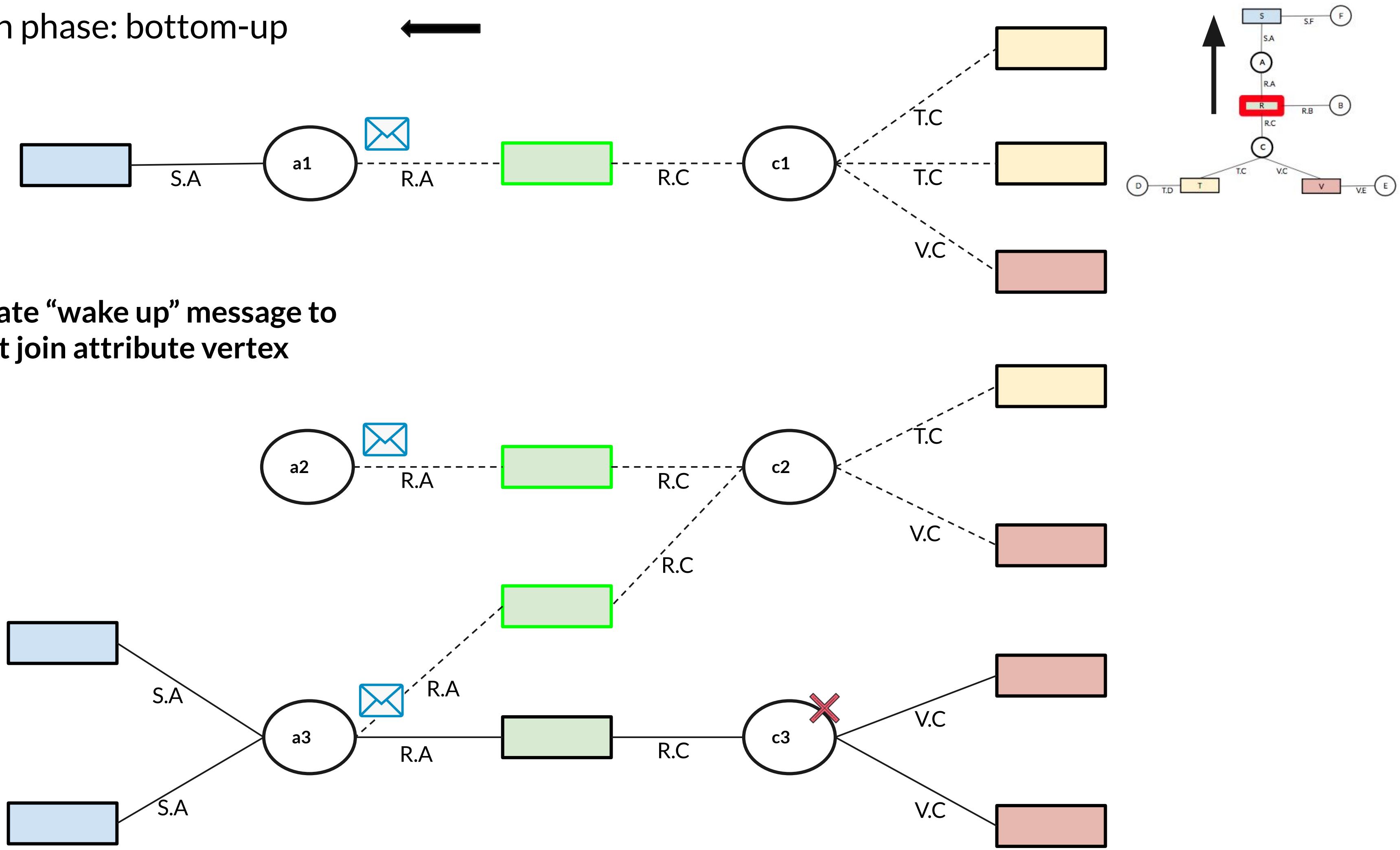




Propagate “wake up” message to the next join attribute vertex

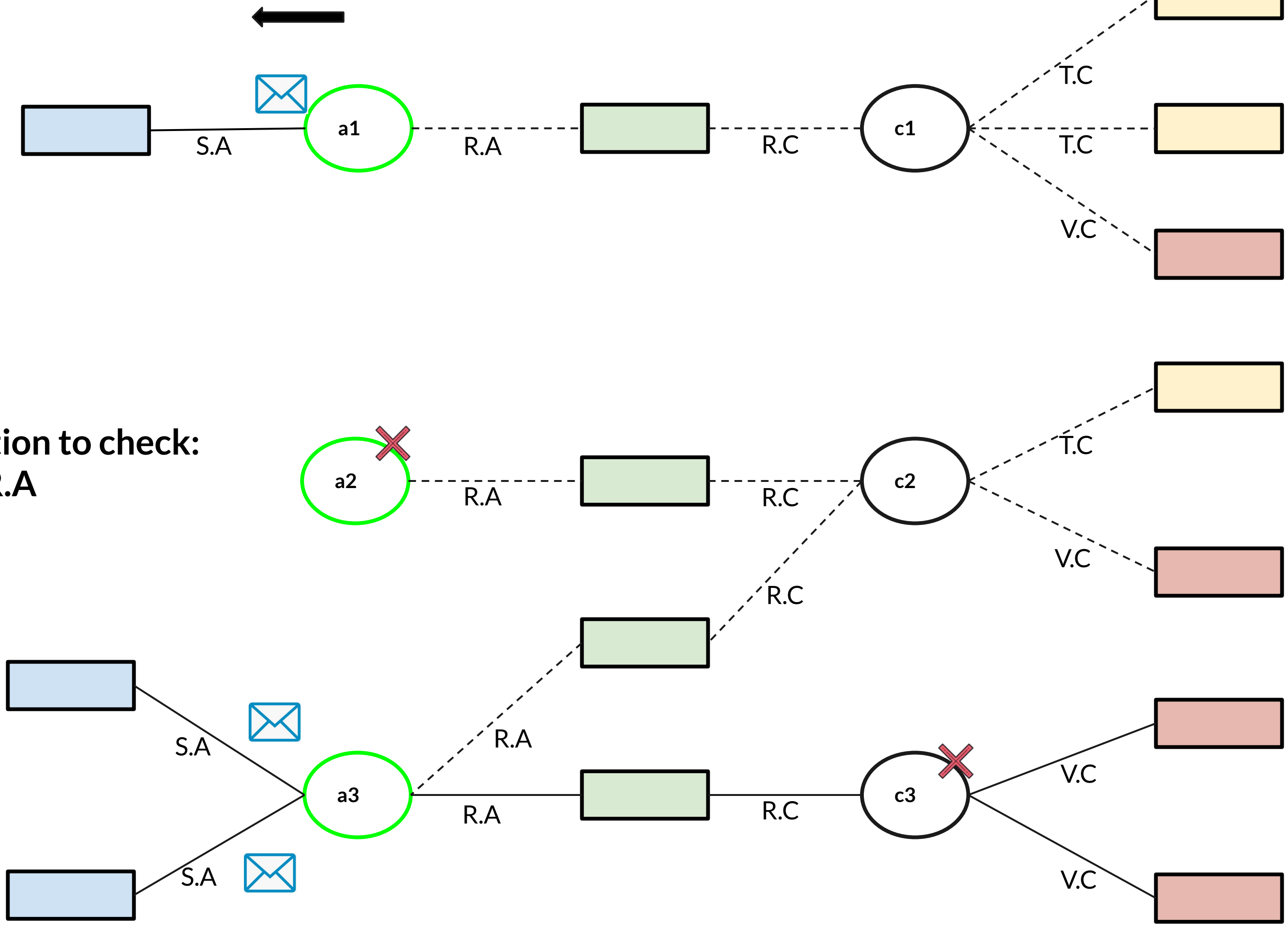


Reduction phase: bottom-up

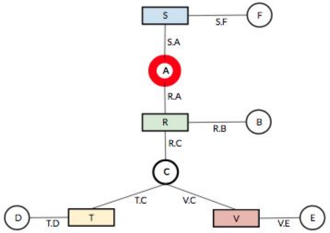


Propagate "wake up" message to the next join attribute vertex

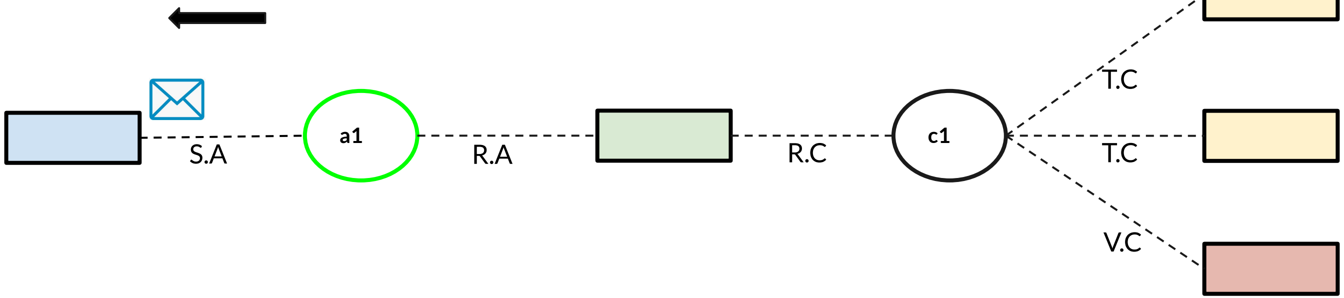
Reduction phase: bottom-up

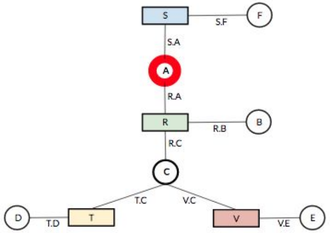


Condition to check:
S.A = R.A

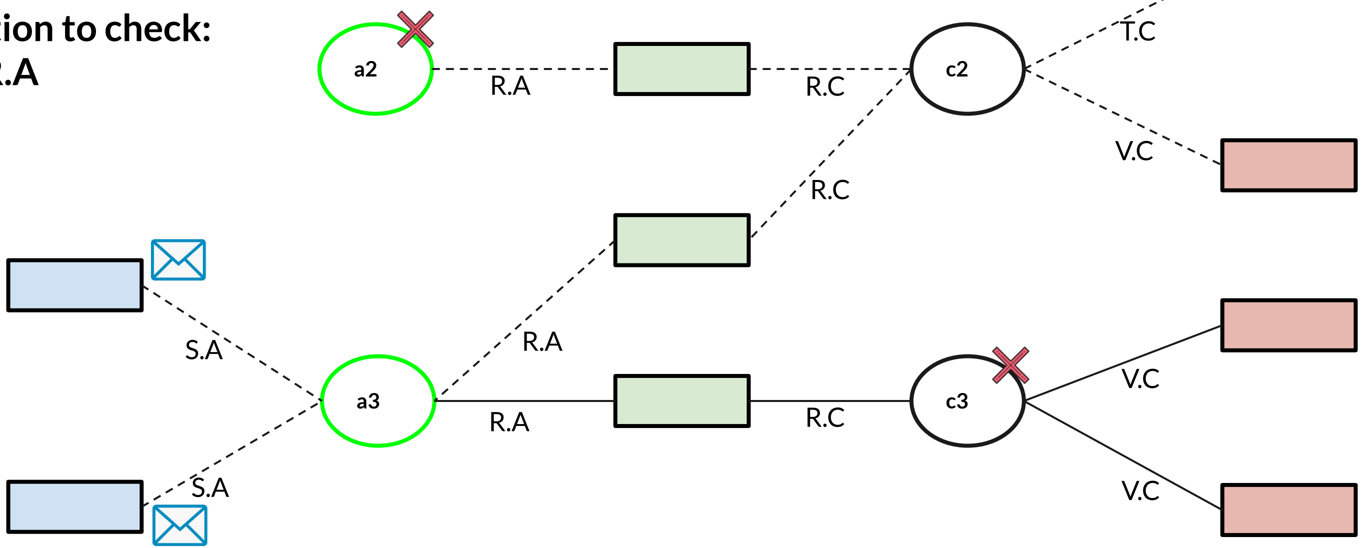


Reduction phase: bottom-up

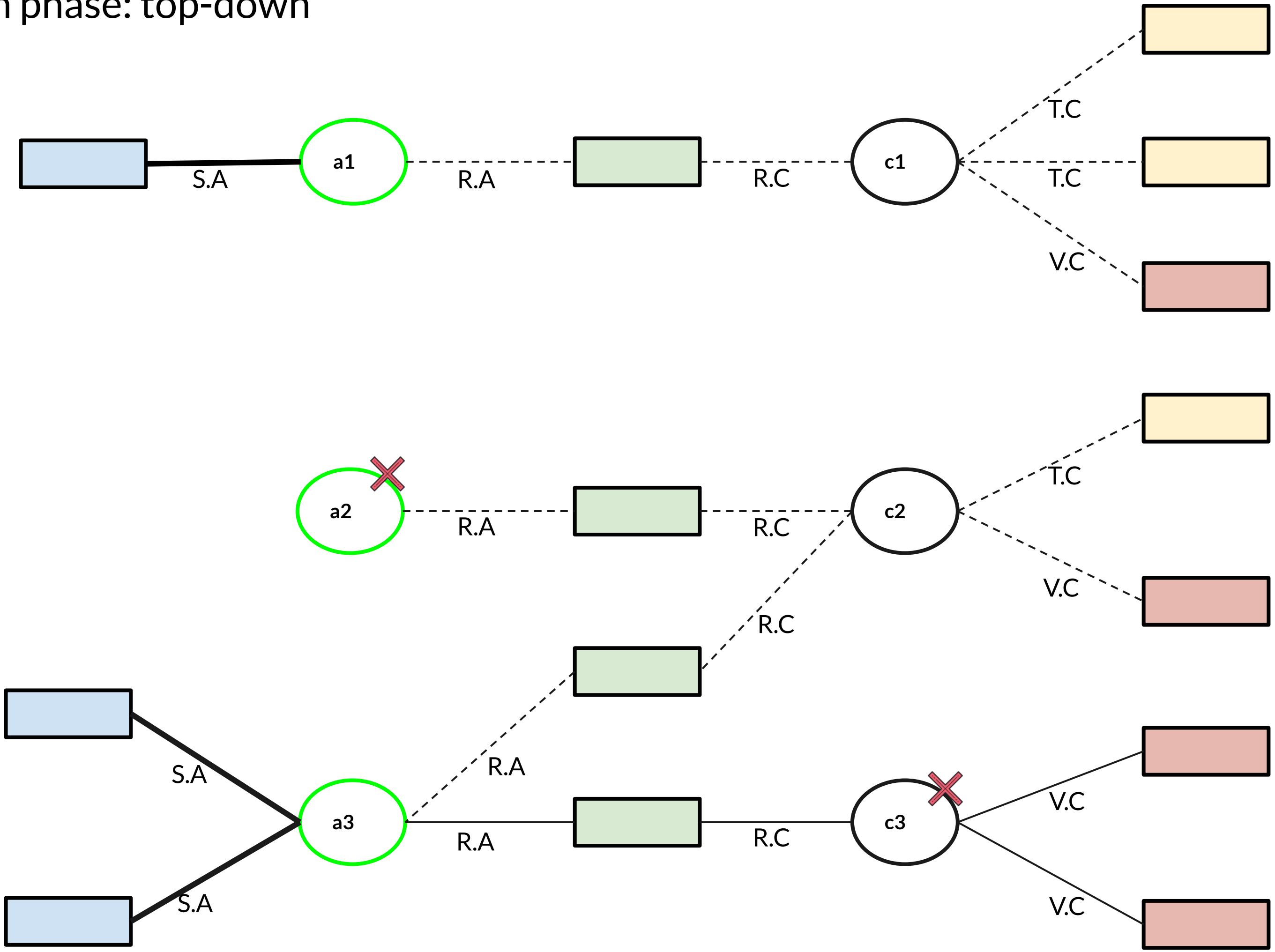


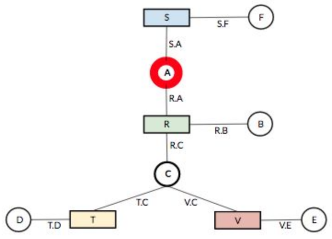


Condition to check:
S.A = R.A

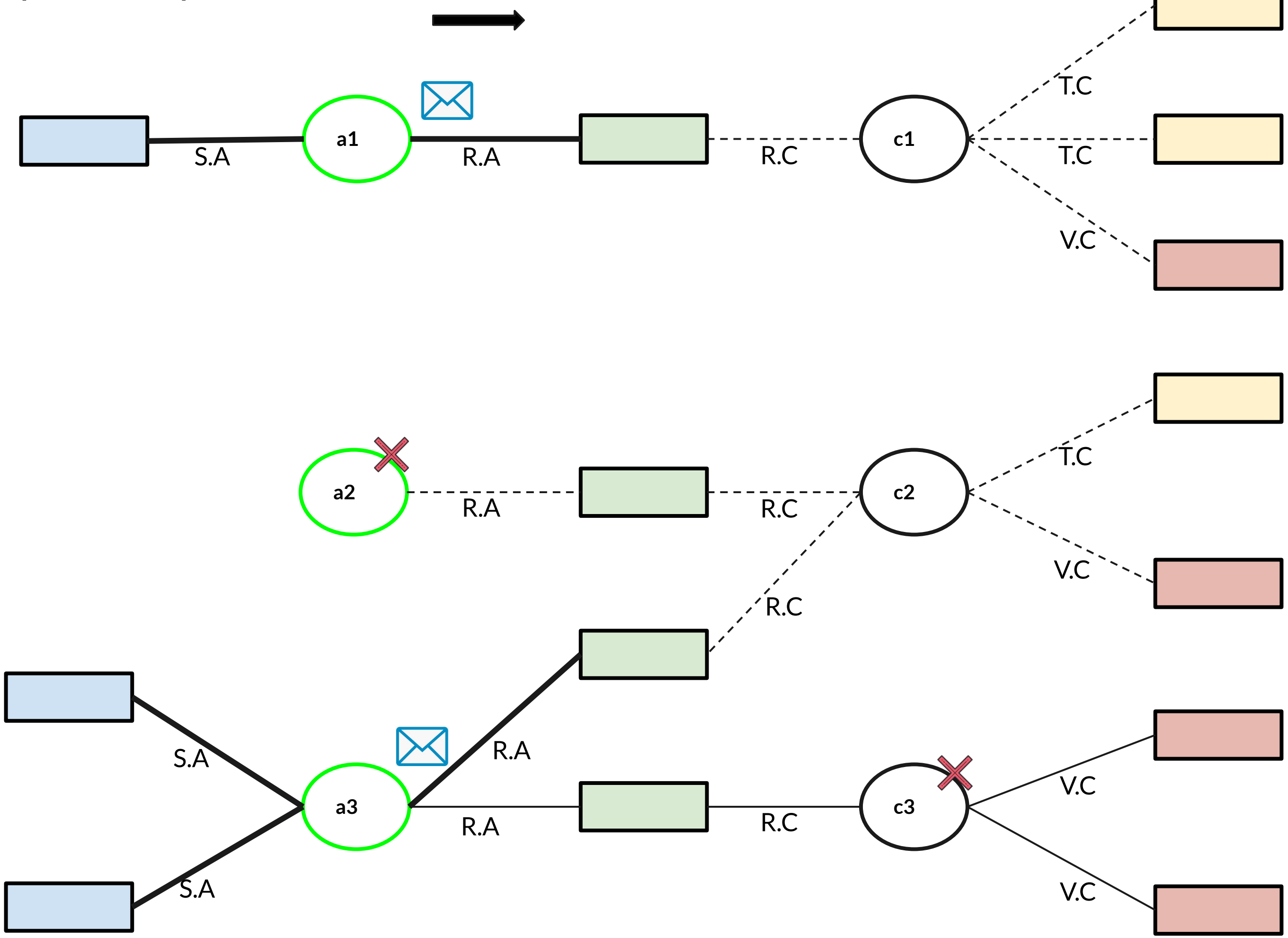


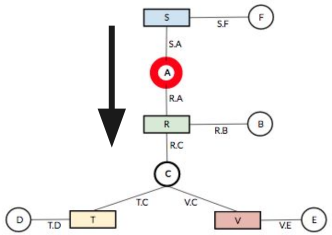
Reduction phase: top-down



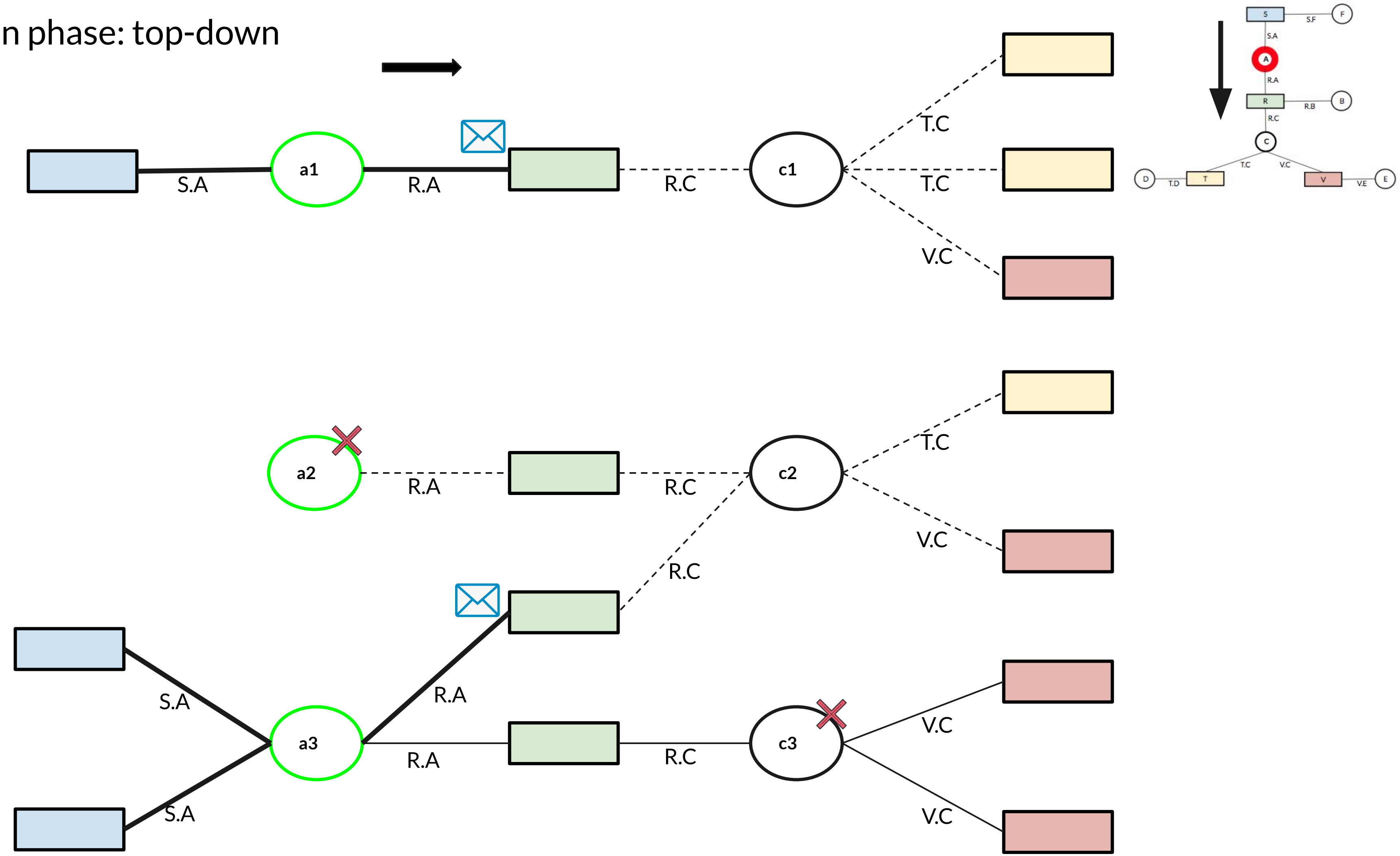


Reduction phase: top-down

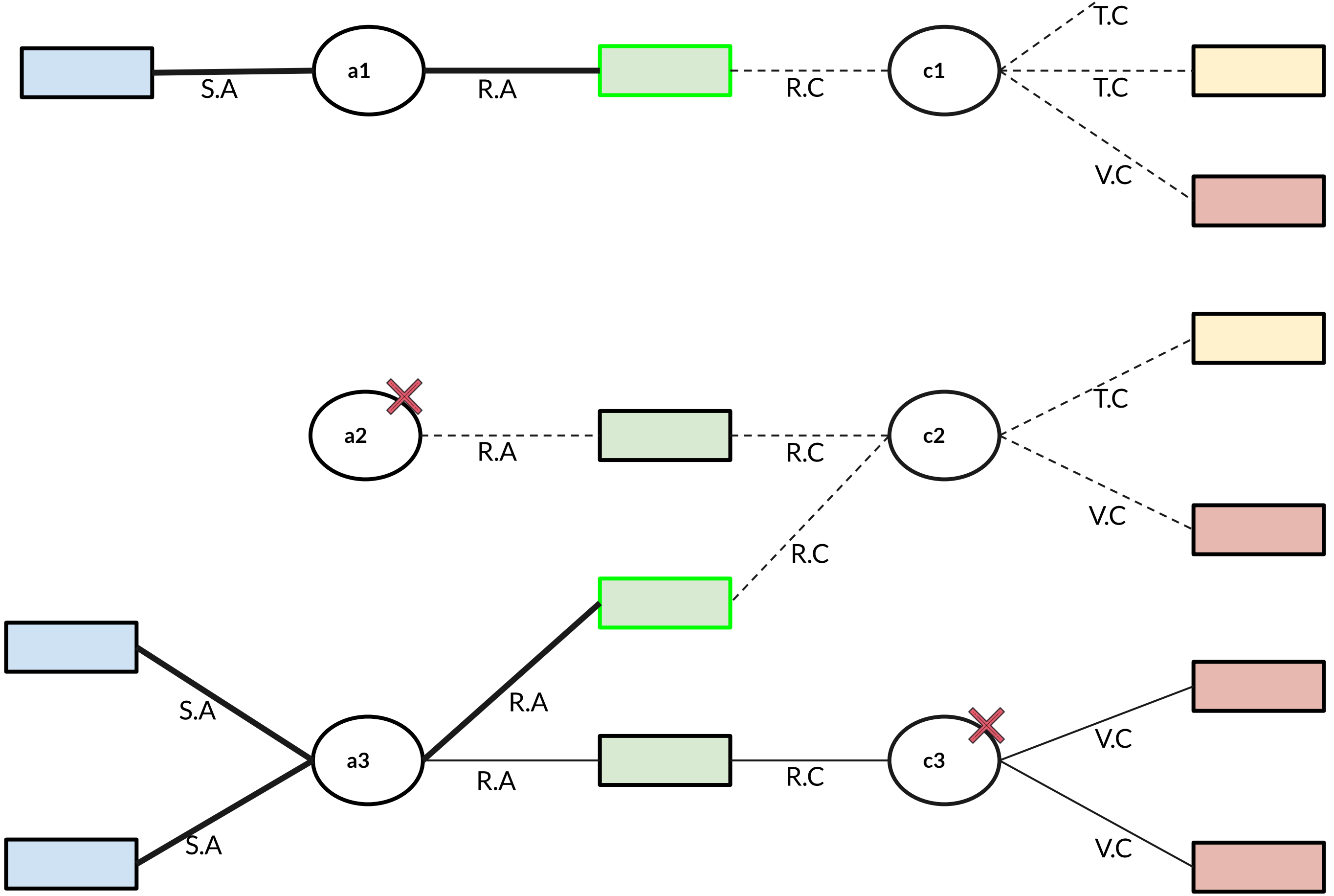


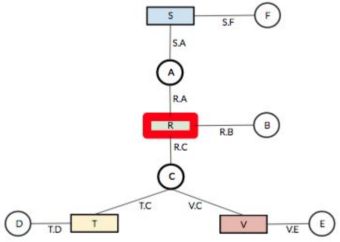


Reduction phase: top-down

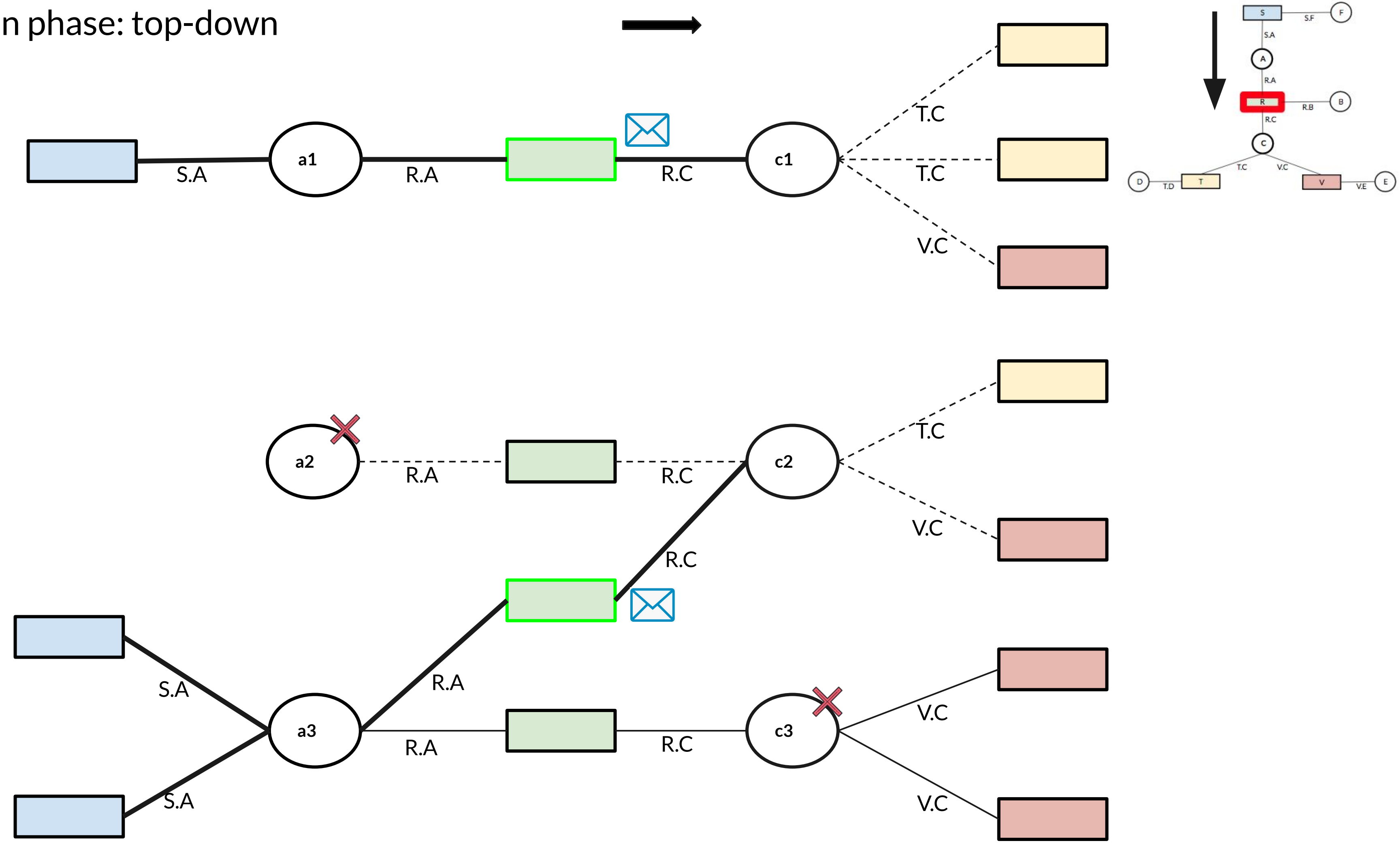


Reduction phase: top-down

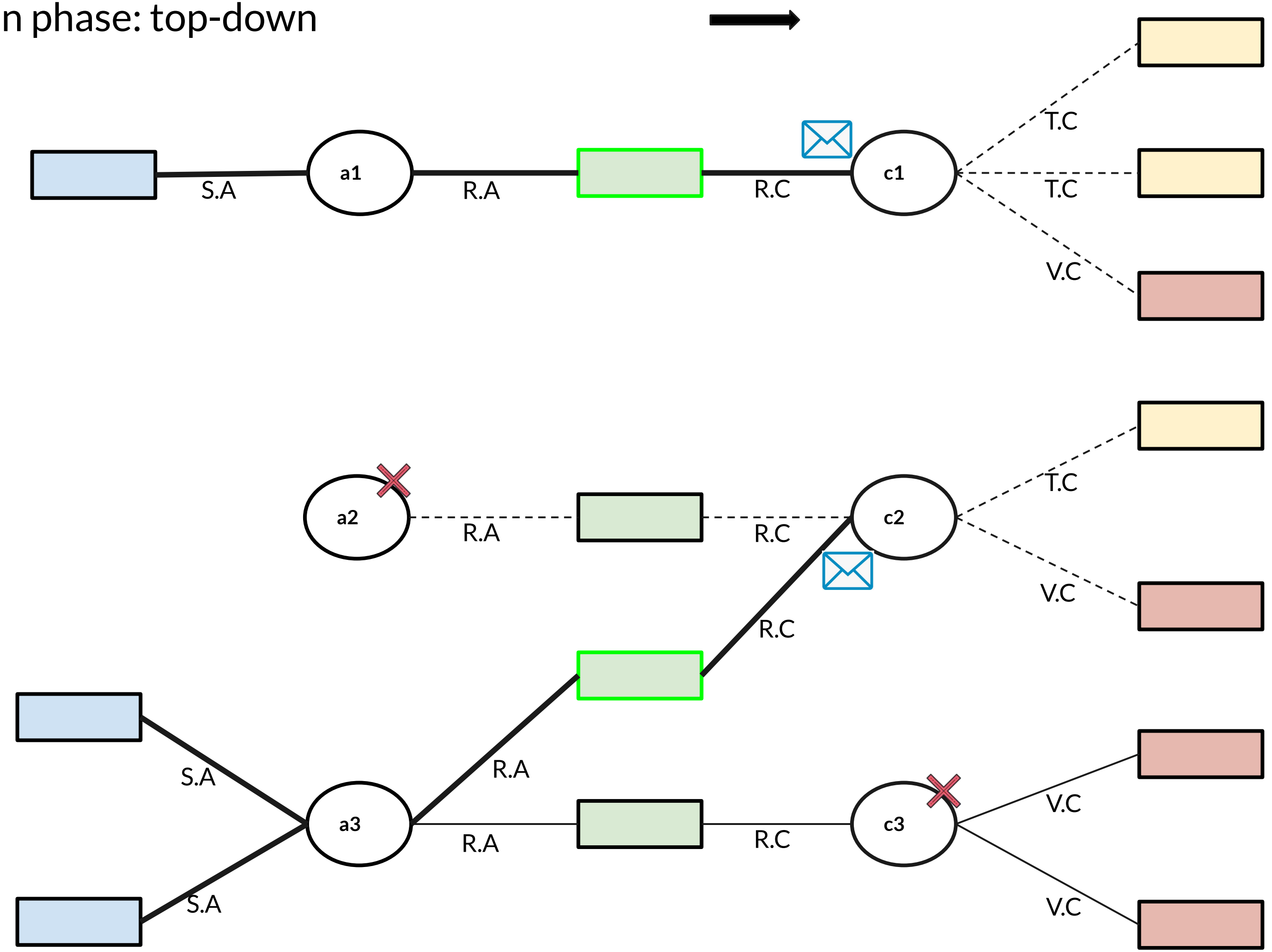


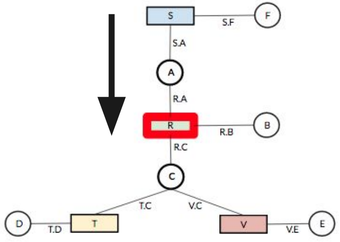


Reduction phase: top-down

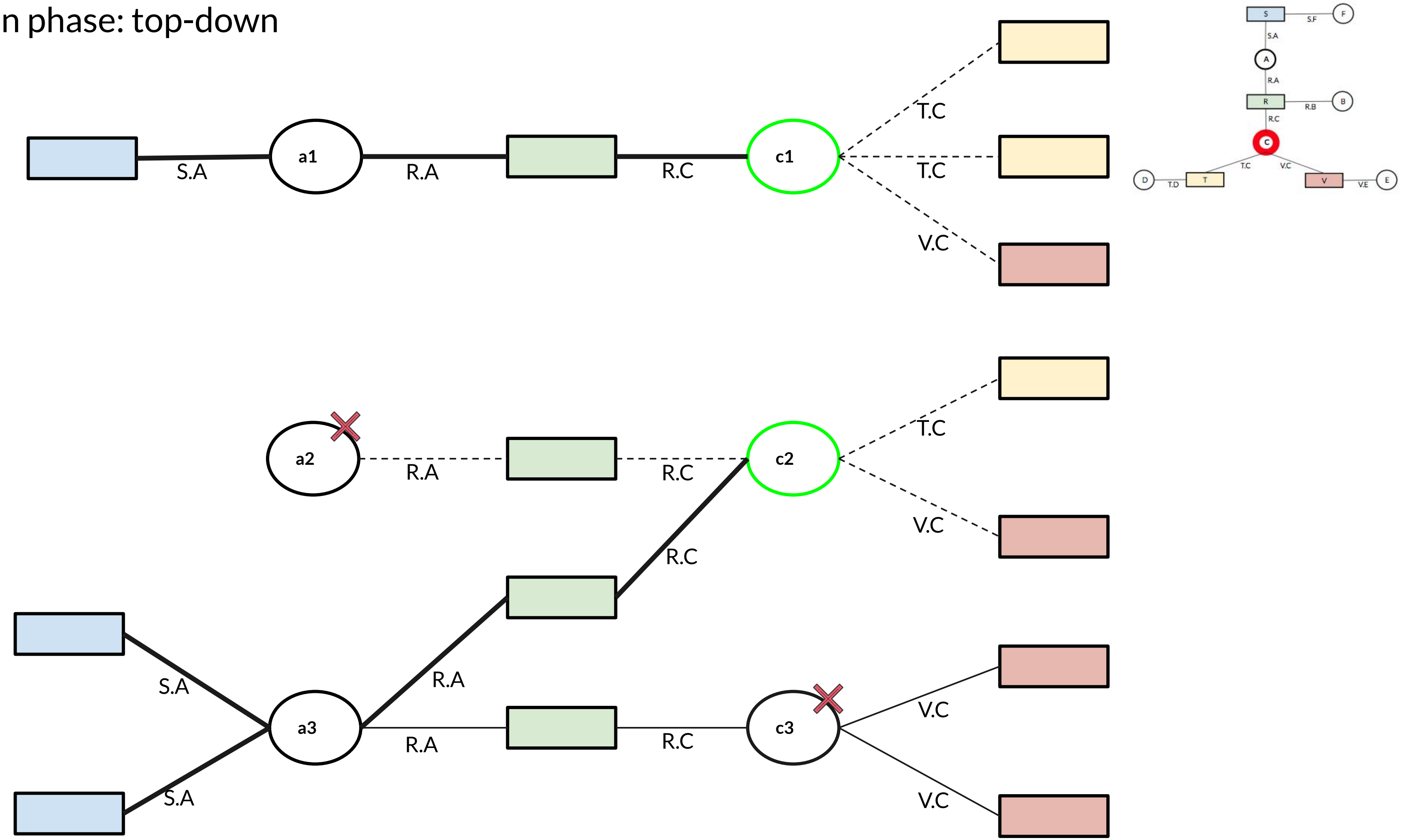


Reduction phase: top-down

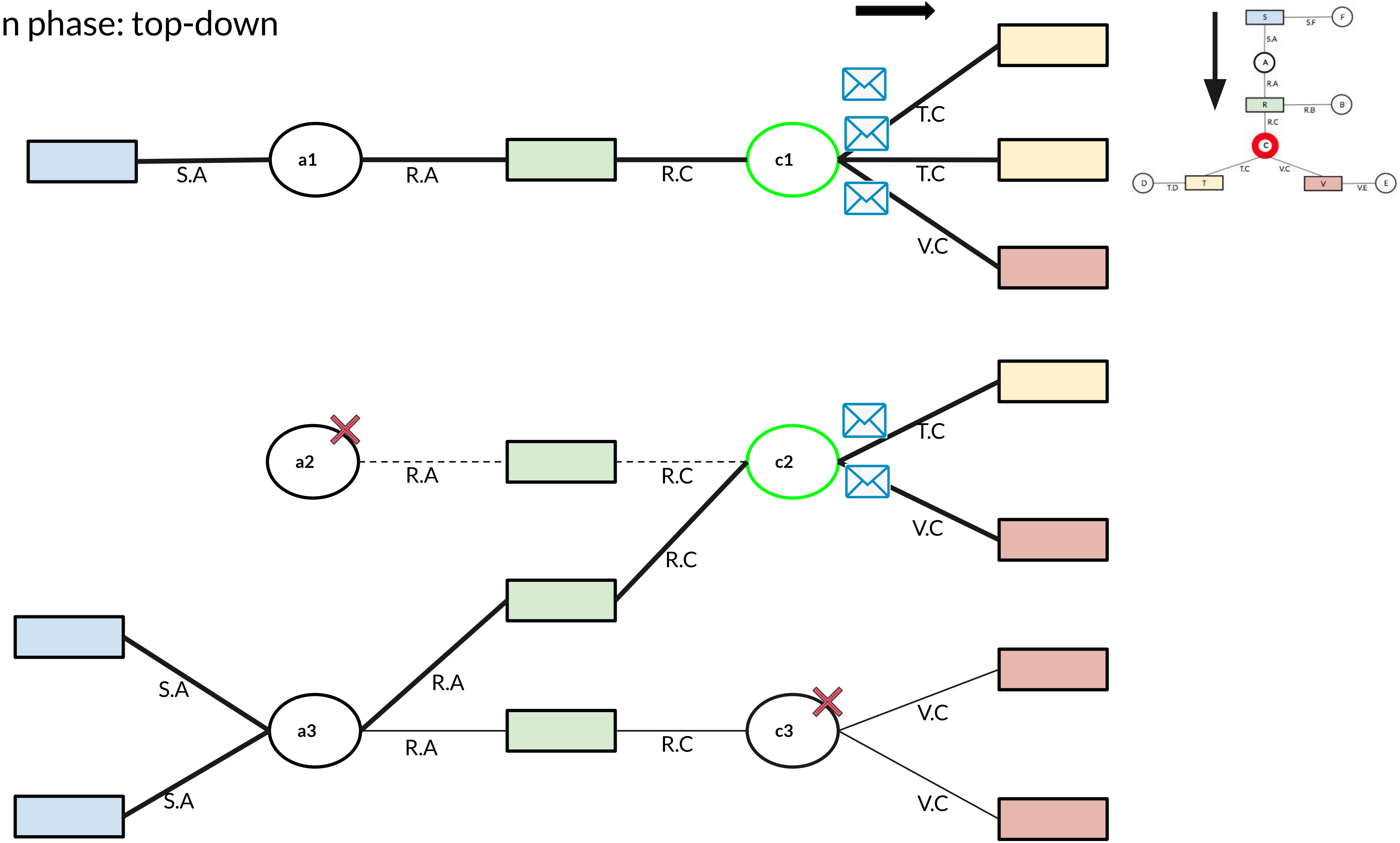




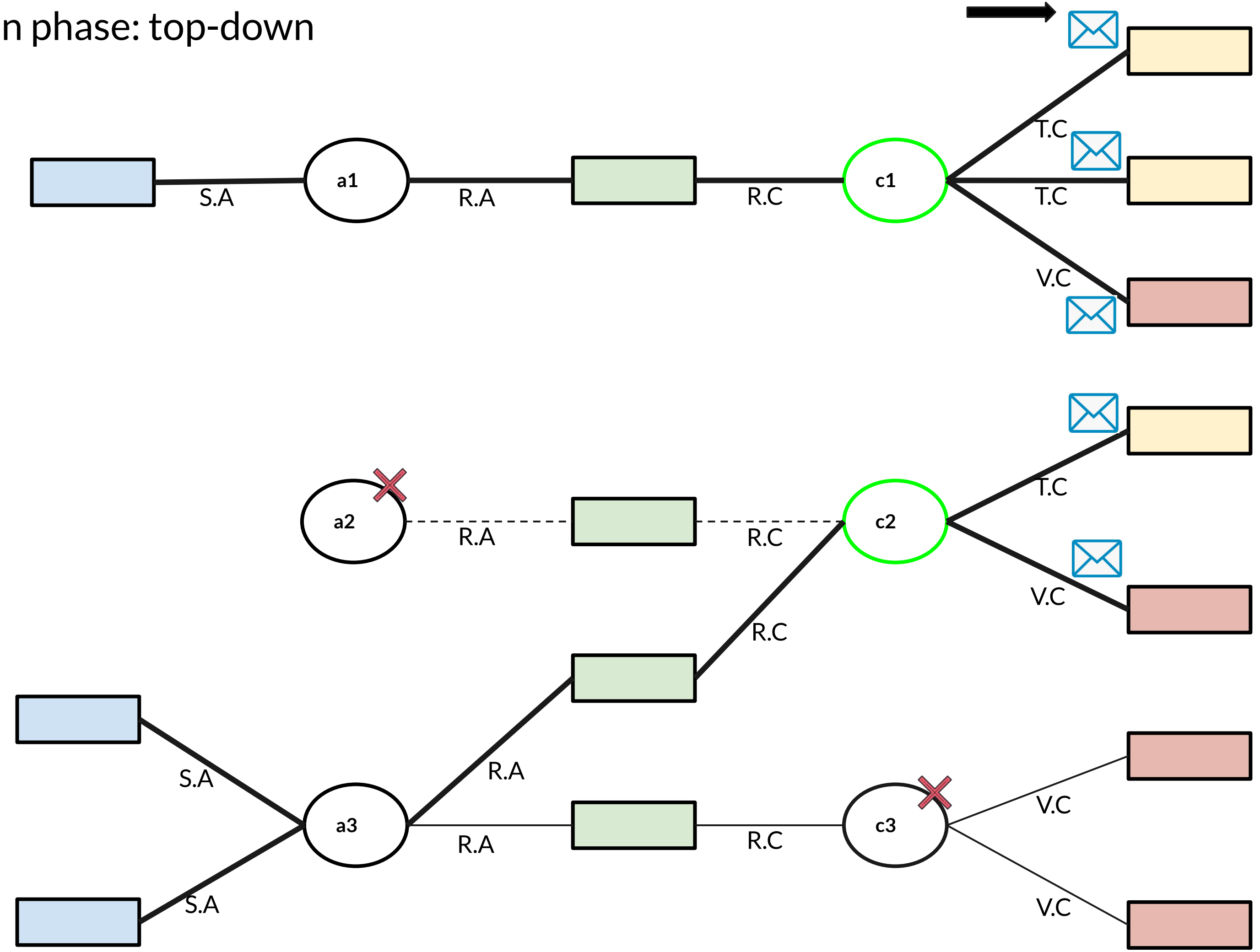
Reduction phase: top-down

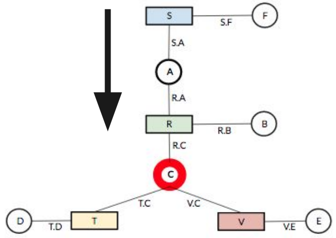


Reduction phase: top-down

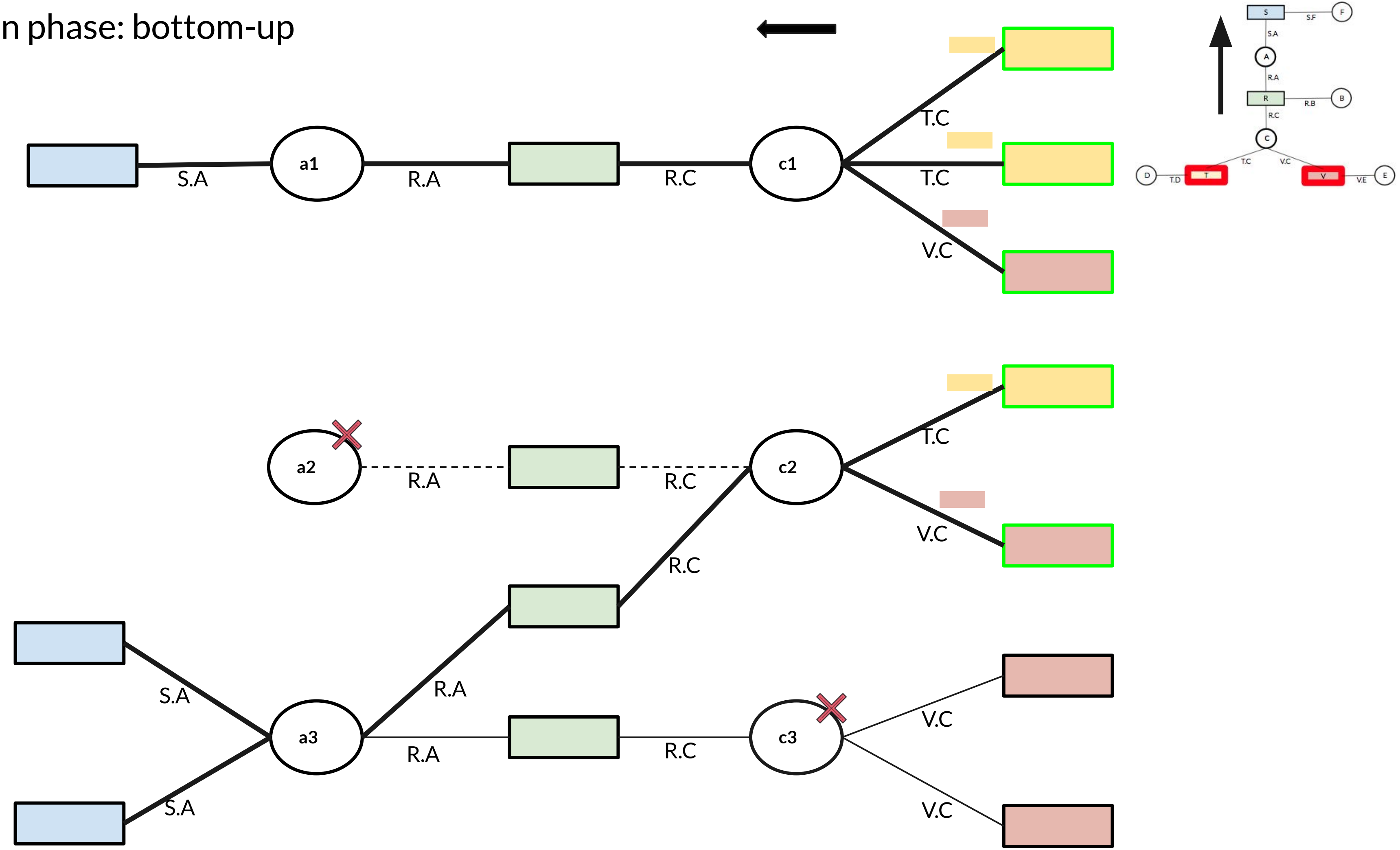


Reduction phase: top-down

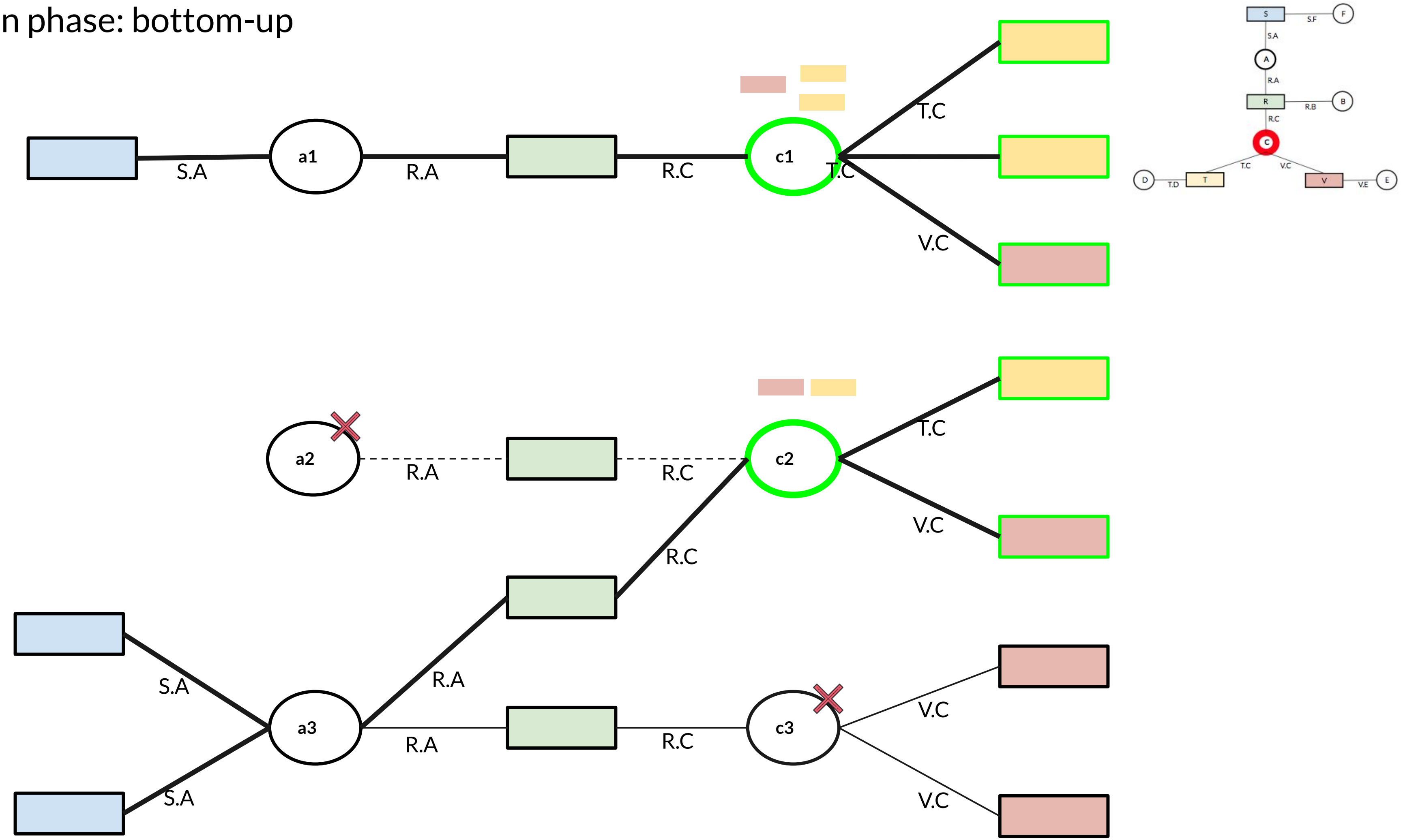




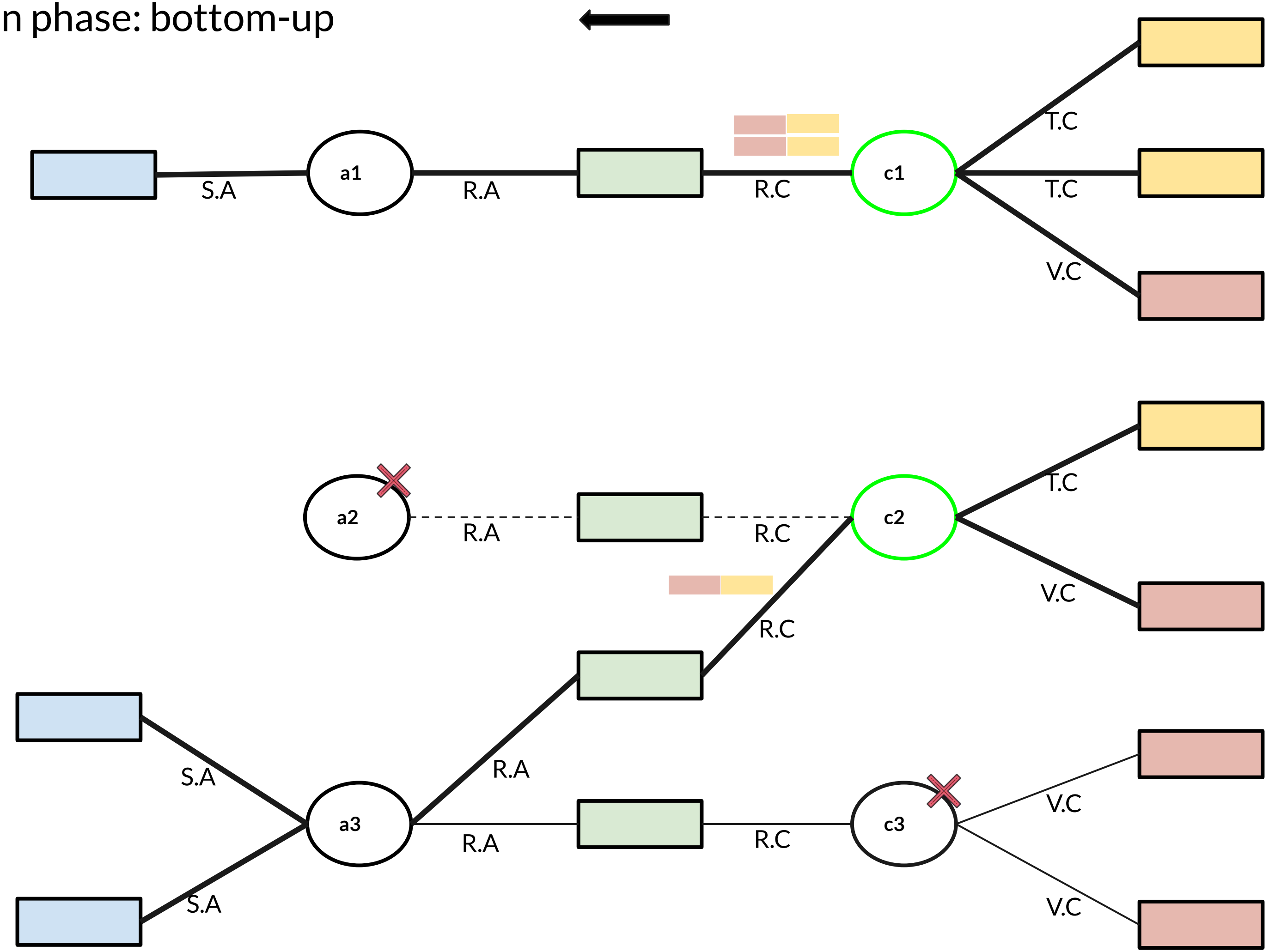
Collection phase: bottom-up

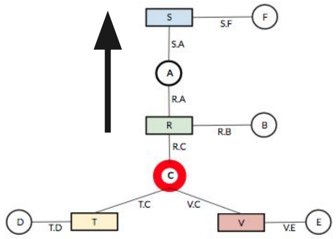


Collection phase: bottom-up

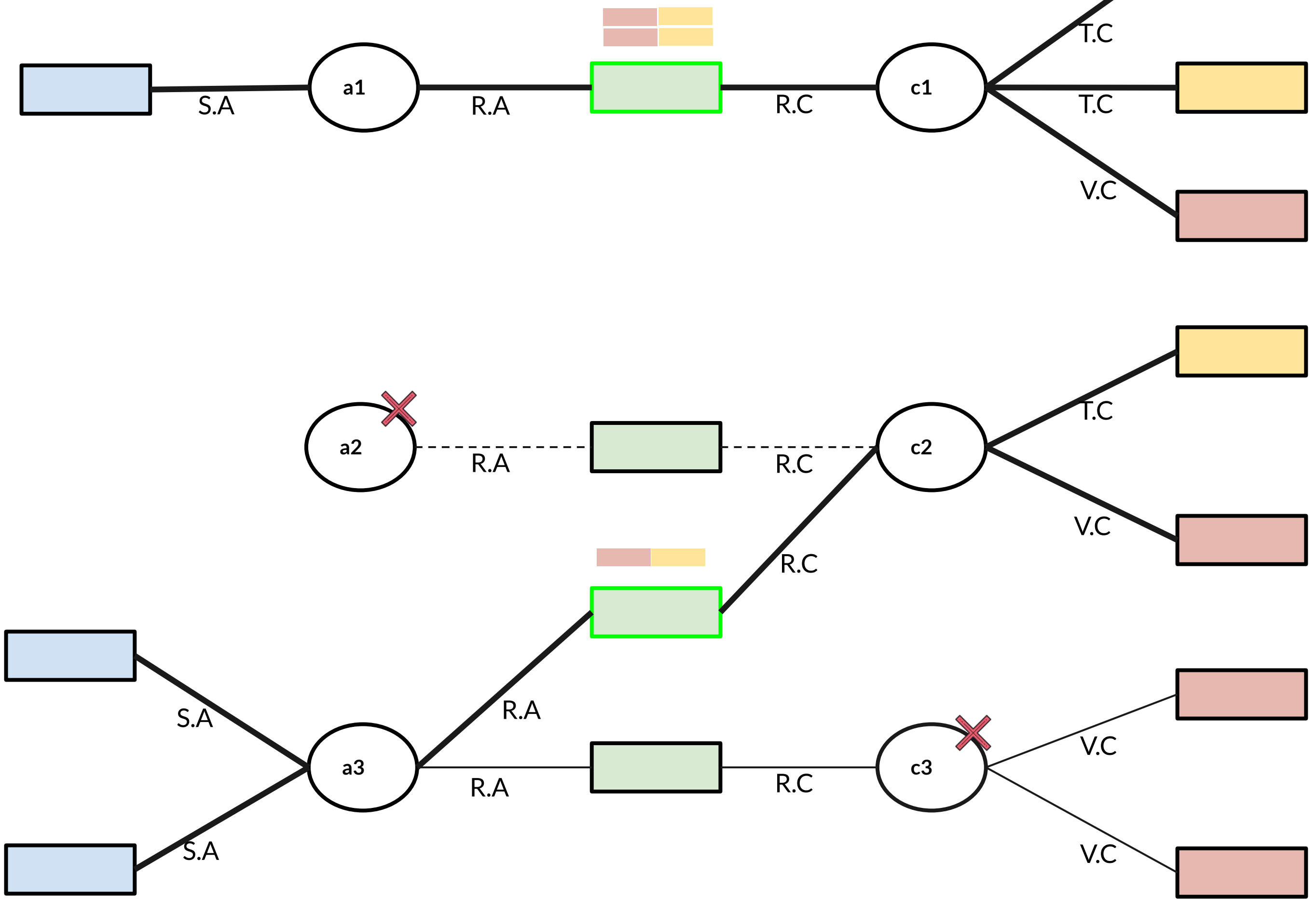


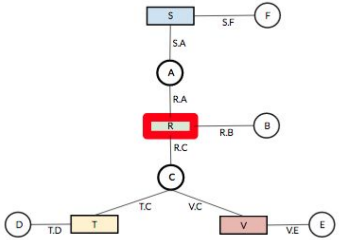
Collection phase: bottom-up



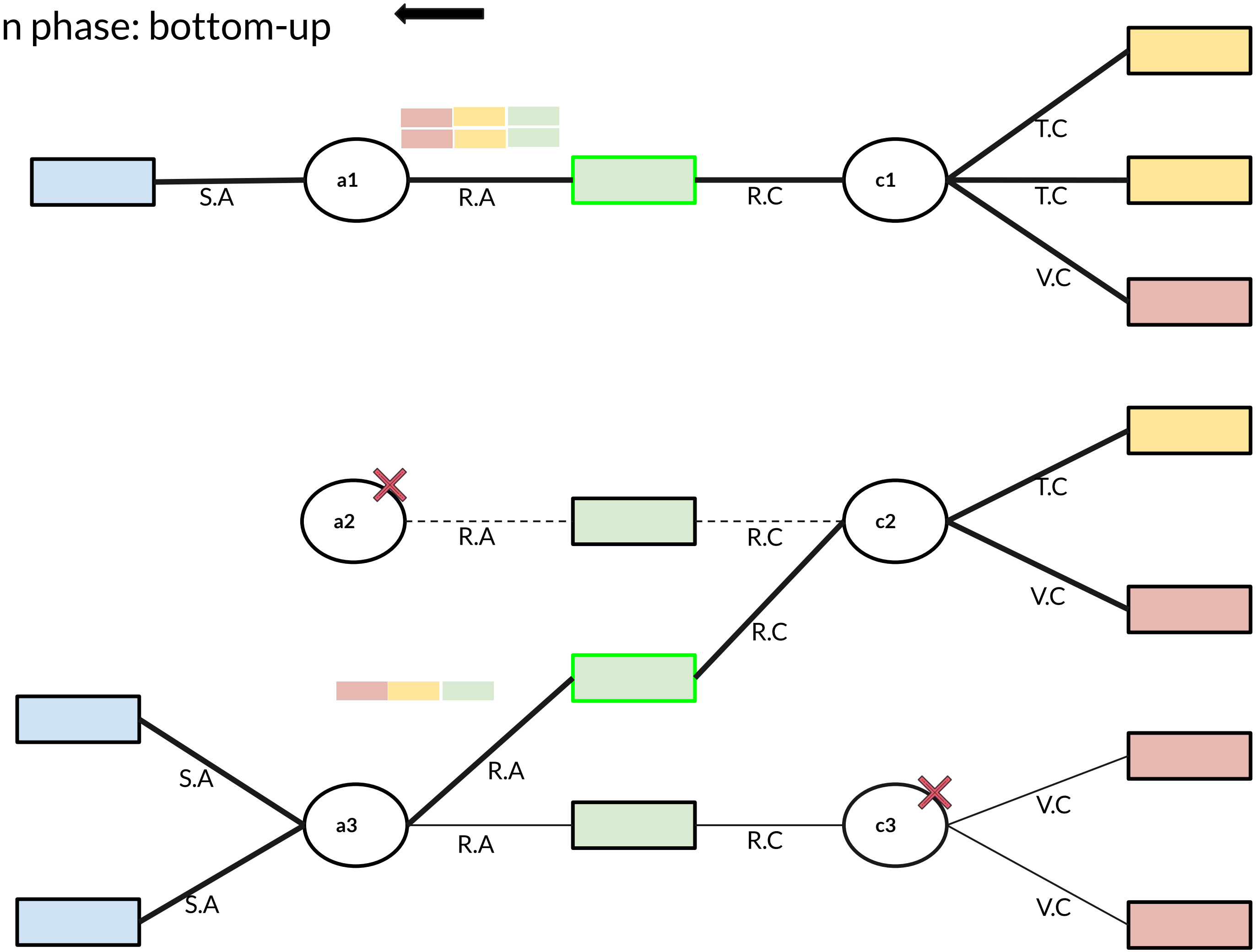


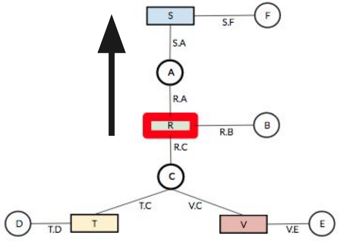
Collection phase: bottom-up



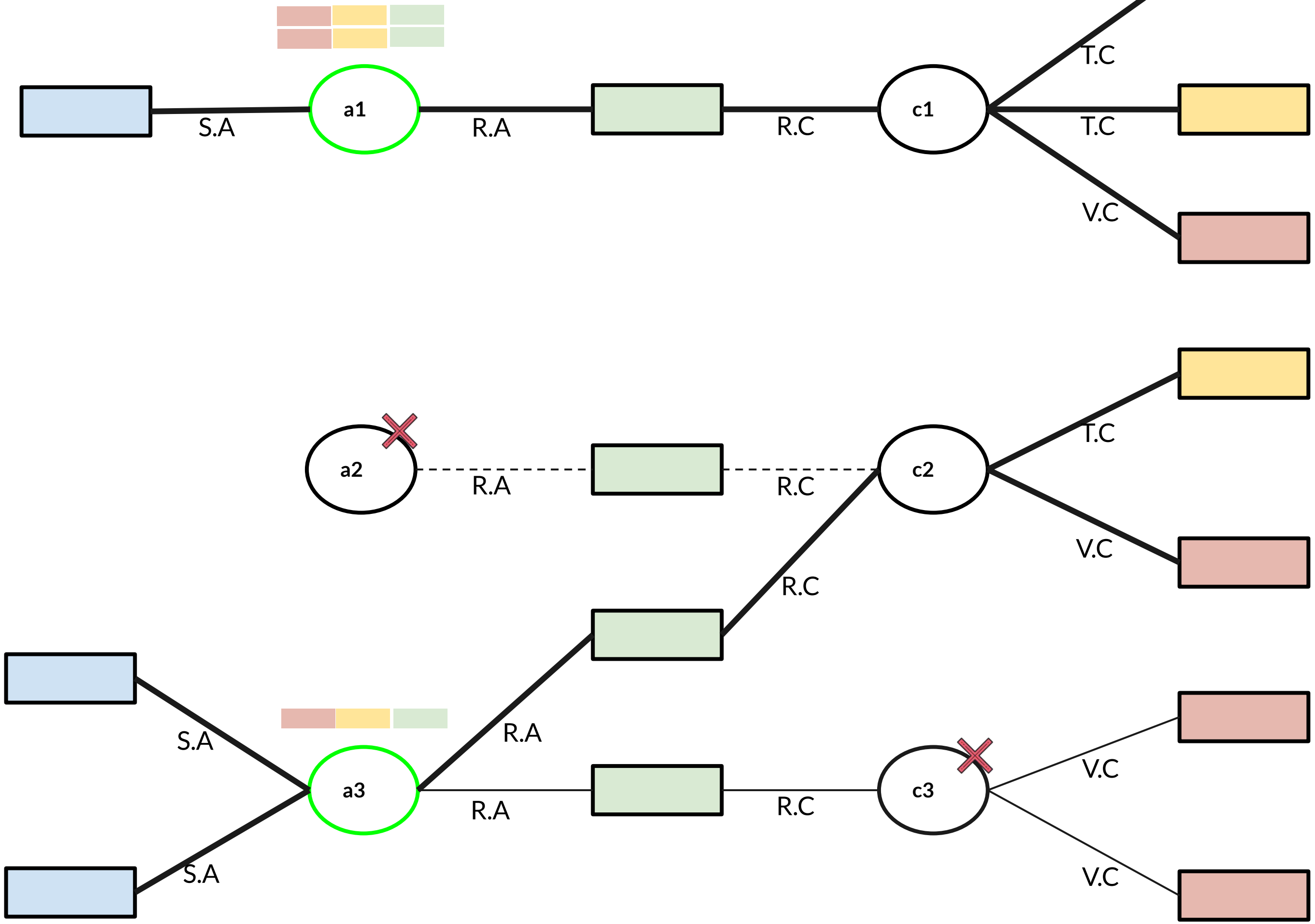


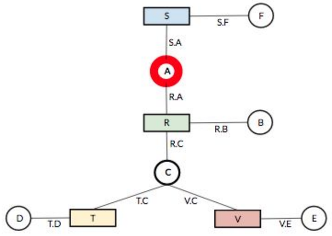
Collection phase: bottom-up



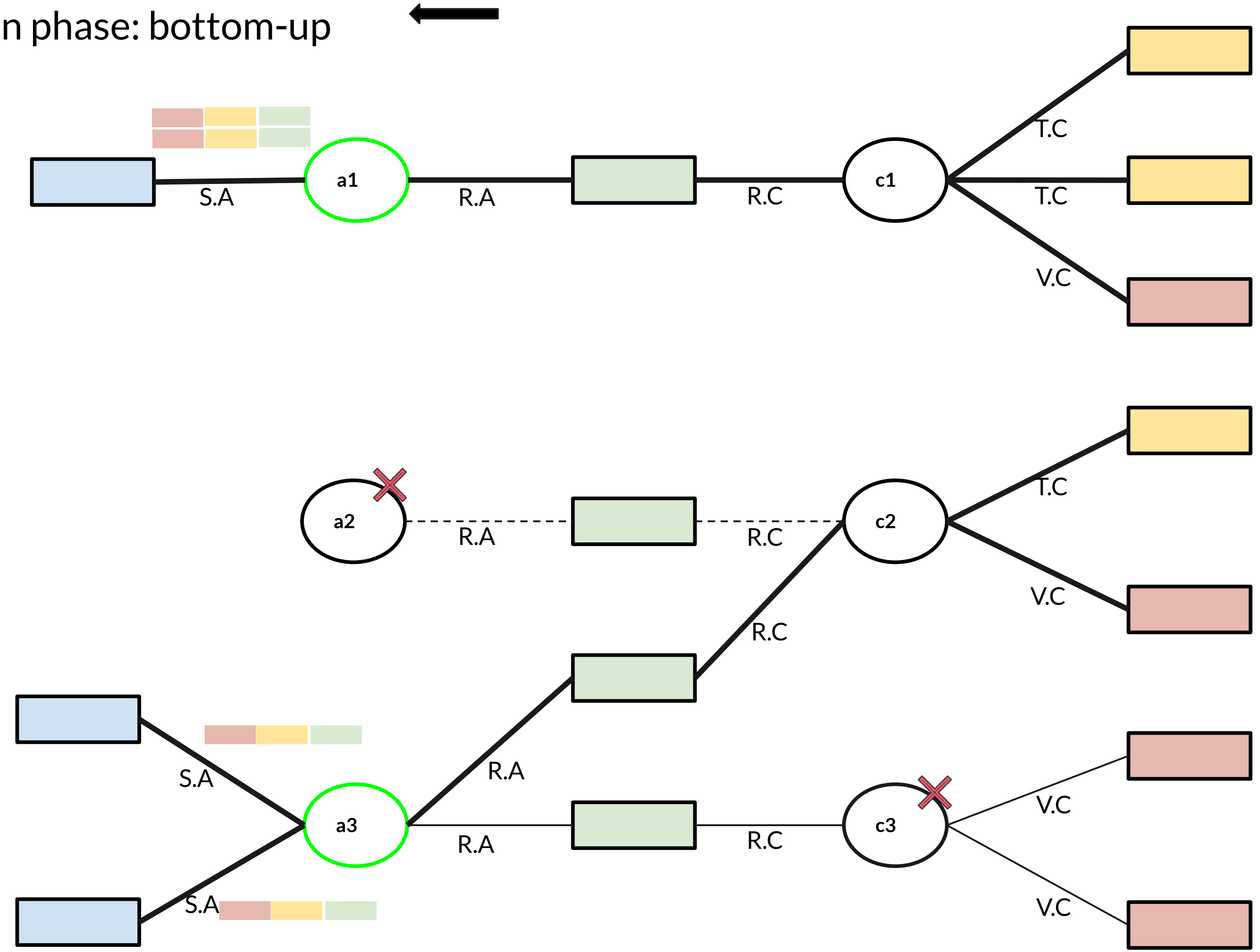


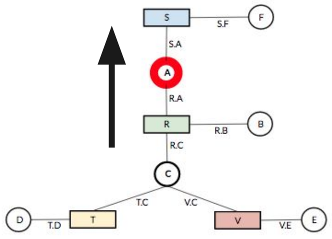
Collection phase: bottom-up



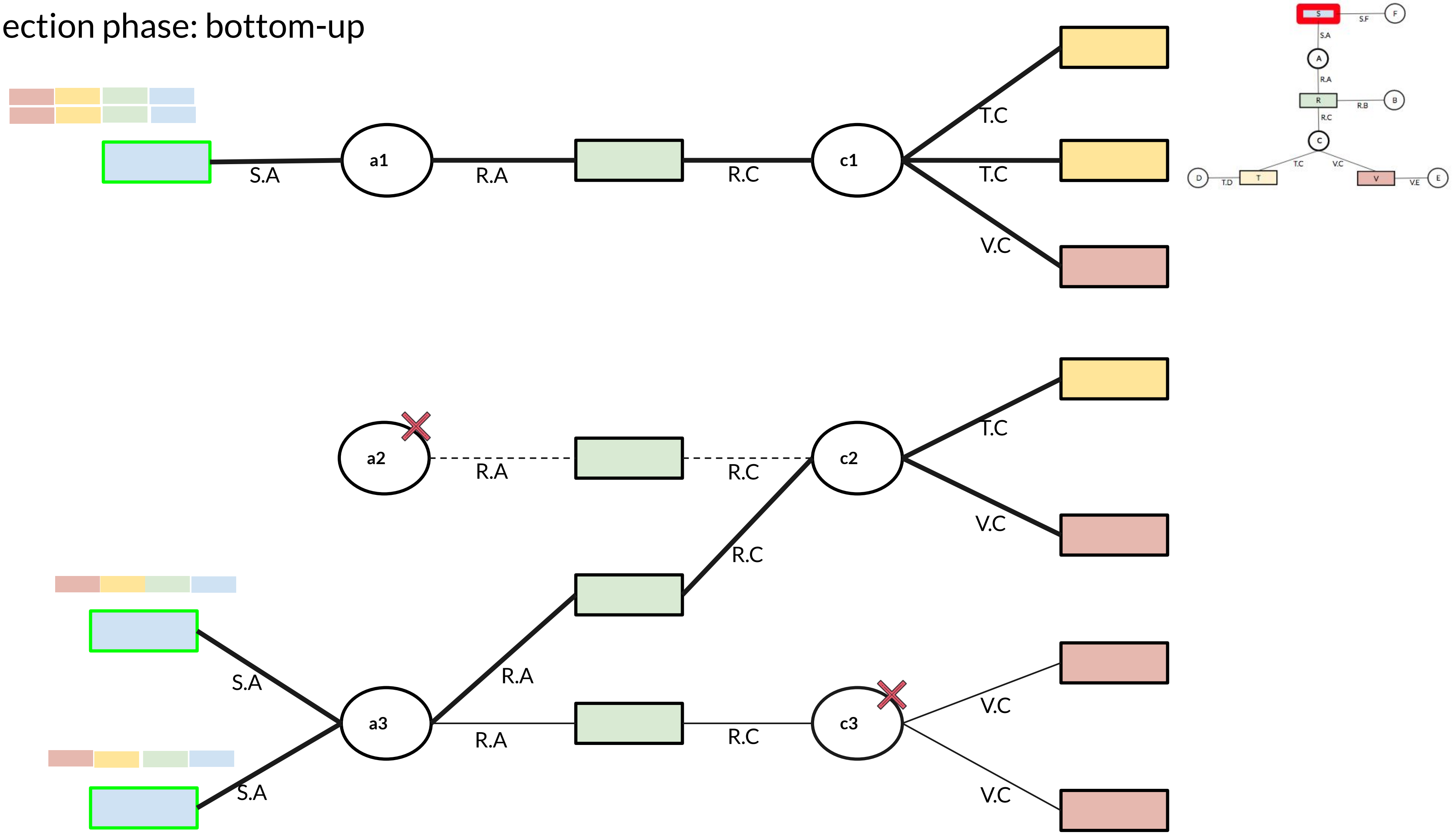


Collection phase: bottom-up

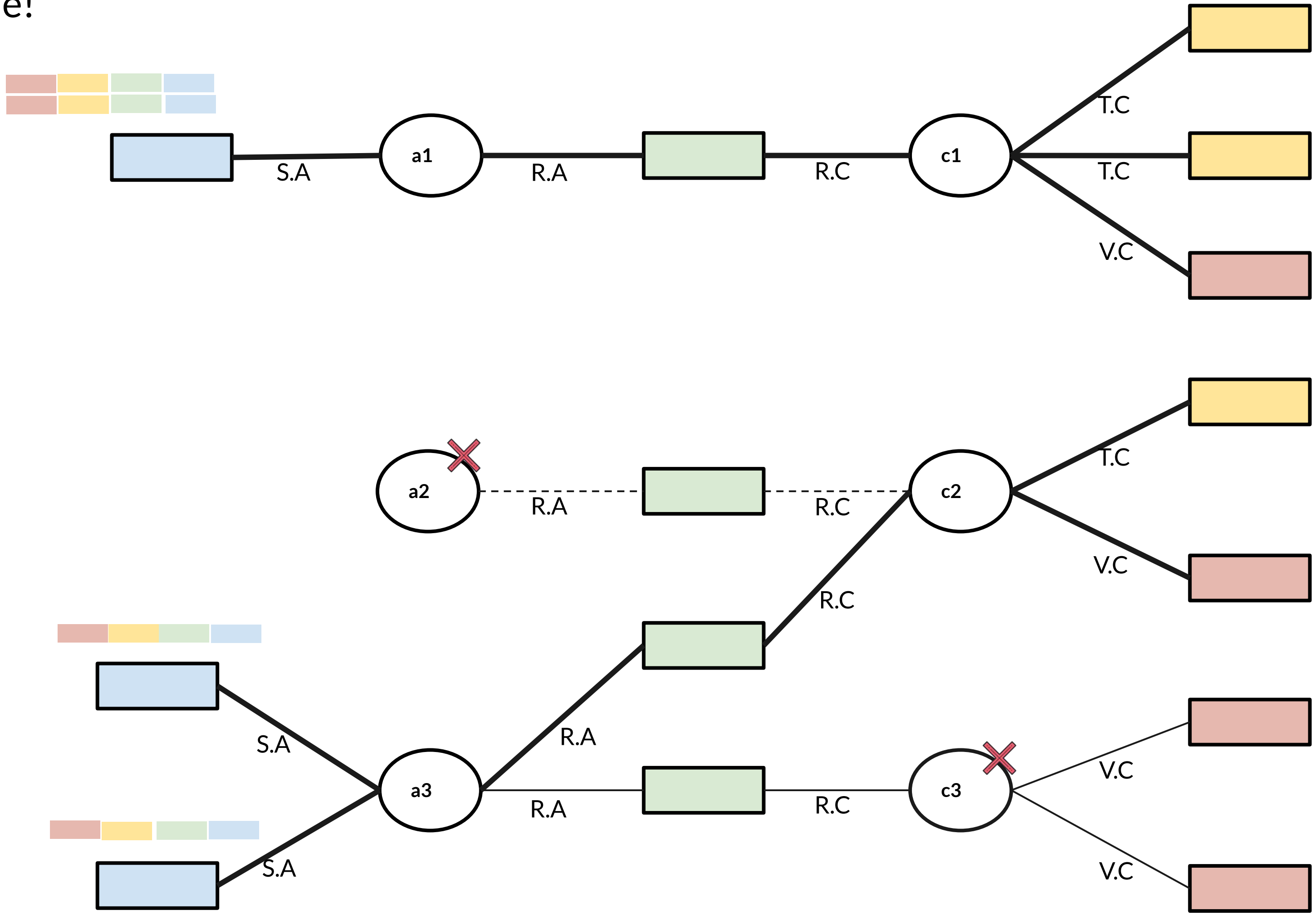




Collection phase: bottom-up



Done!



Acyclic Join Algorithm: Cost analysis

Total communication and computation: O(IN + OUT)

- **Reduction phase:** O(IN)
  - Sending messages along outgoing edges → #edges is linear in the size of the input
- **Collection phase:** O(OUT)
  - Only traverse vertices that are part of the output → total #messages is at most the number of tuples in the output

Total number of rounds = O(1):

- Only depends on the size of the query, i.e. number of relations to join
- Under assumption that query size is constant, then algorithm runs in O(1) rounds

Acyclic Multi-way Joins: Main Result

Any acyclic join query can be computed by a vertex-centric algorithm with **O(IN + OUT)** communication and computation cost.

*Vertex-centric analog of [Yannakakis81]

Acyclic Queries: Comparison to existing results

Distributed setting:

	Vertex-centric Join	**GYM** [Afrati17,Koutris18]	**Parallel Sort Join** [Hu'19]
Communication cost	O(IN +OUT) factorized : O (IN + F_{OUT})	O(IN +OUT)	$O(IN + \sqrt{IN \cdot OUT})$
Computation cost	O(IN +OUT) factorized : O (IN + F_{OUT})	(involves hashing cost)	(involves sorting cost)
Factorizing join result	yes	no	no
Partition/sort input at query runtime	no	yes	yes

Main theoretical results:

- **Acyclic queries** can be computed with optimal cost O(IN + OUT)
- **Cyclic queries:** $O(IN^{n/2})$
  - **Triangle queries** (simplest cycle) with worst-case optimal cost $O(IN^{3/2})$
- **Cartesian Product:** $O(IN^n)$

Main Theorem (TAG-join algorithm): An **arbitrary equi-join query**, given its tree decomposition with width *w*, can be computed in the vertex-centric BSP model with $\mathbf{O(IN^w+OUT)}$ communication and computation.

IN - input size , OUT - output size, n - number of relations in a query, w - width
Assuming query size is constant

Beyond Joins

- ***Selection***
- ***Projection***
- ***Grouping and Aggregation***
    - including over partition by, rollup and HAVING clause
- ***Subqueries***:
    - Scalar subqueries using >, <, = operators
    - Non-scalar (multi-row) using IN, EXISTS, NOT IN, NOT EXISTS
    - Correlated subqueries
    - Subqueries in FROM clause (inline view)
    - Subqueries defined using WITH clause
- ***Outer Joins*** (left, right, full)

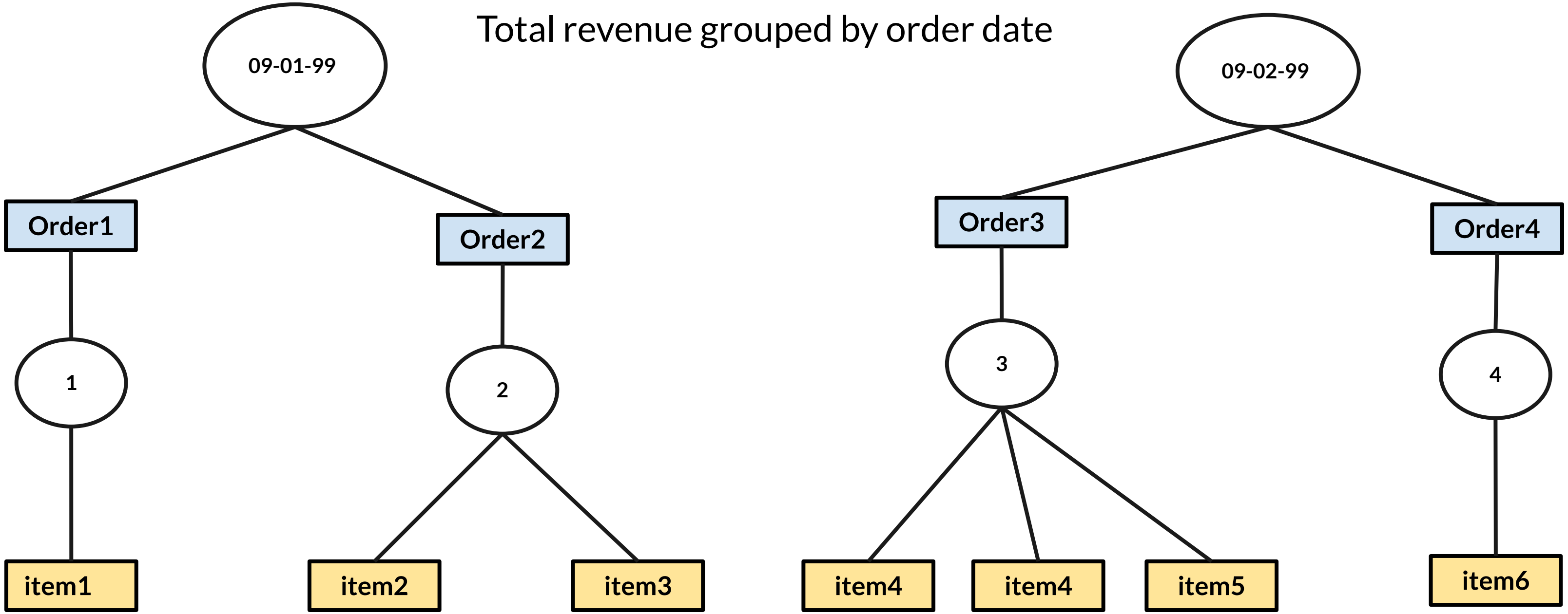
Beyond Joins: Grouping and Aggregation

To group tuples in the output on one or more attributes, and compute some aggregate (e.g count, avg, sum, max, min) value for each group.

- Compute aggregates as we traverse the graph bottom-up in the collection phase.

Beyond Joins: Grouping and Aggregation

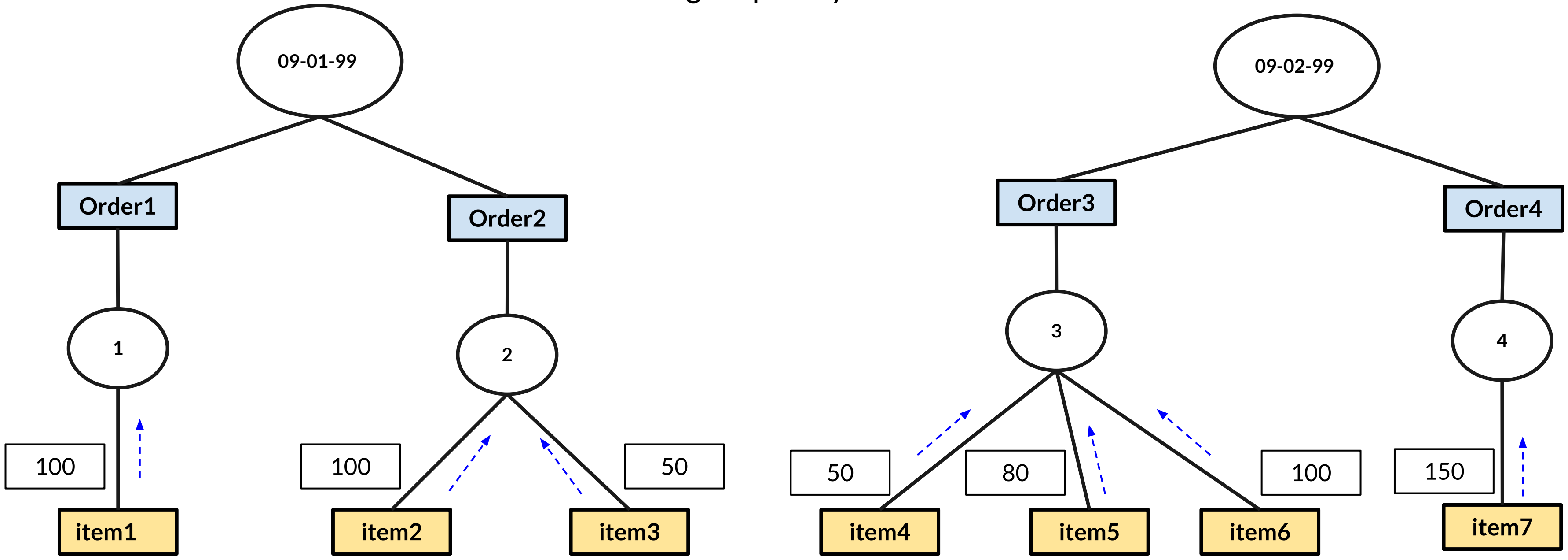
Total revenue grouped by order date



Build a traversal plan s.t. grouping attribute(s) is at the top of the traversal

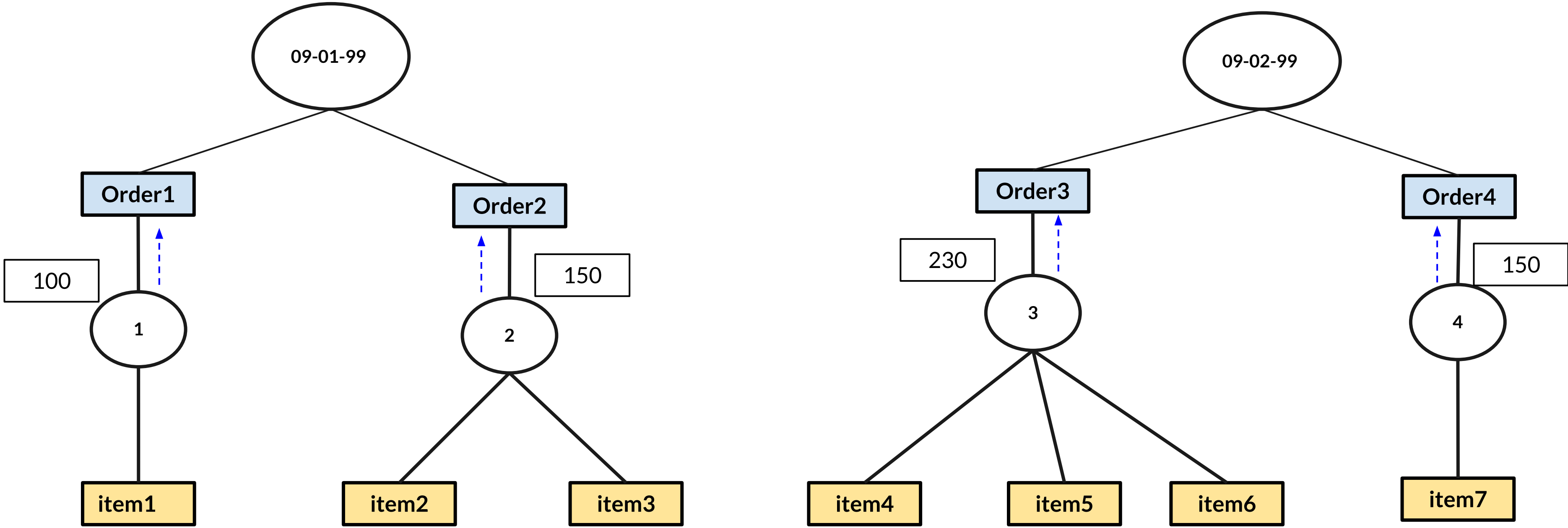
Beyond Joins: Grouping and Aggregation

Total revenue grouped by order date



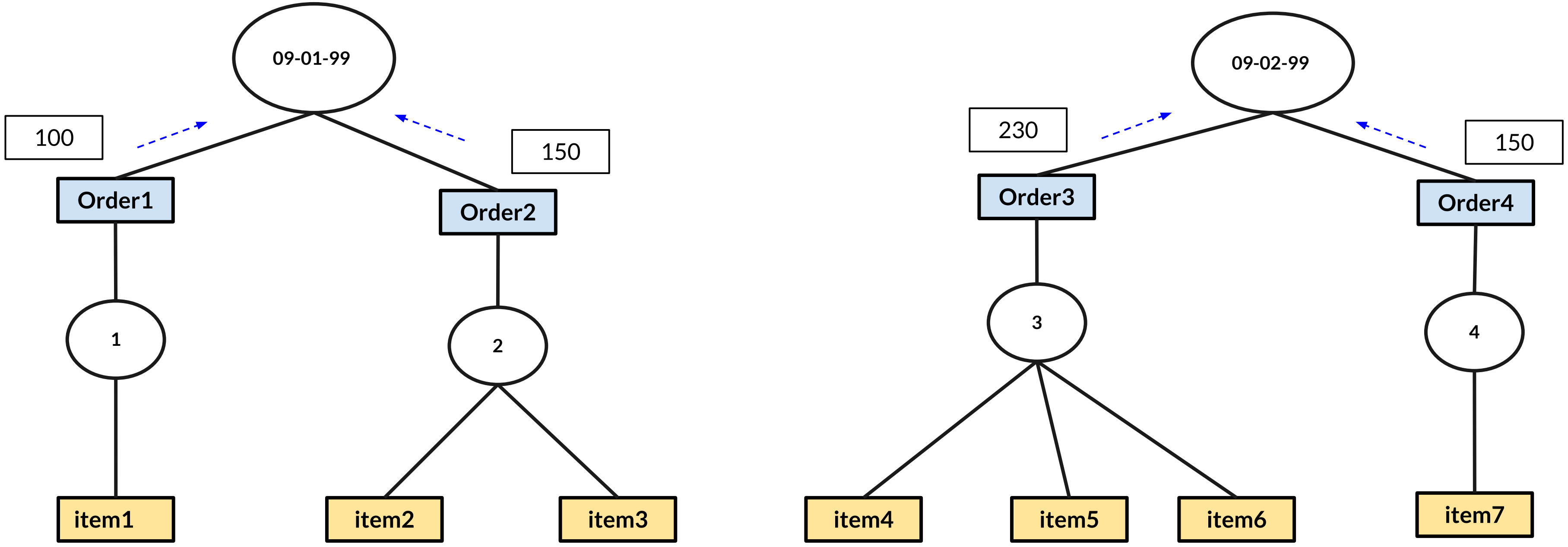
Beyond Joins: Grouping and Aggregation

Total revenue grouped by order date



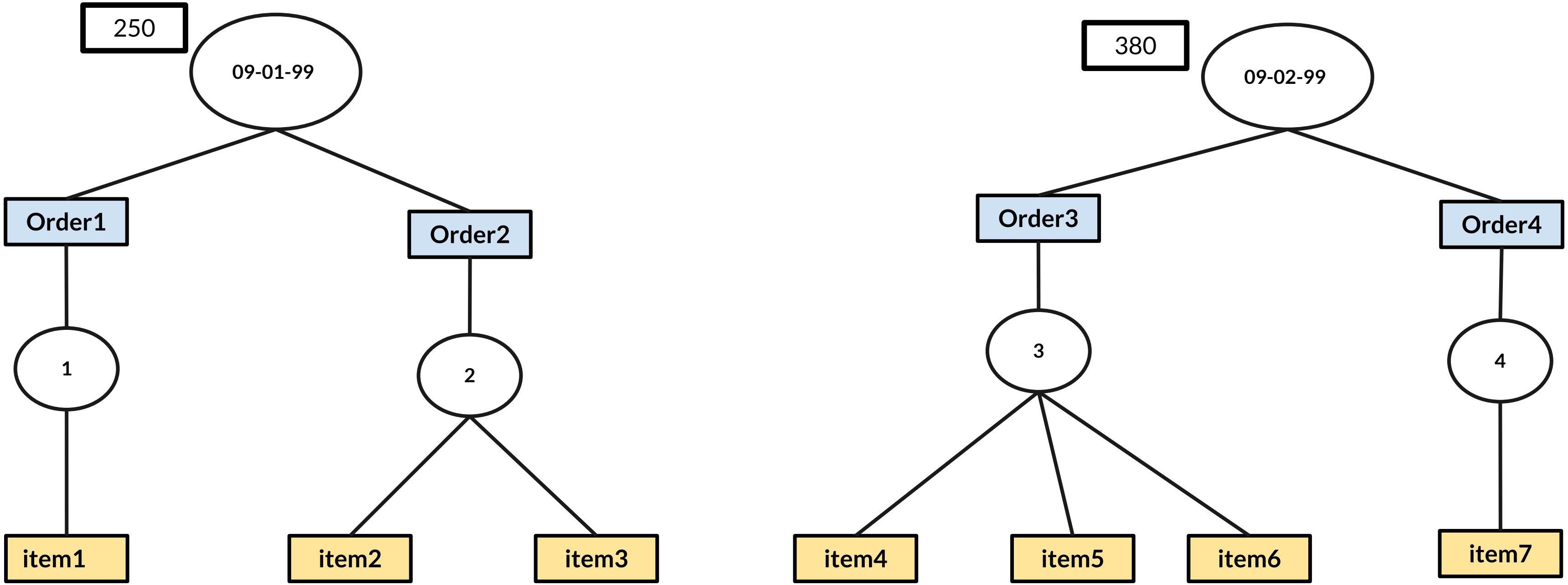
Beyond Joins: Grouping and Aggregation

Total revenue grouped by order date



Beyond Joins: Grouping and Aggregation

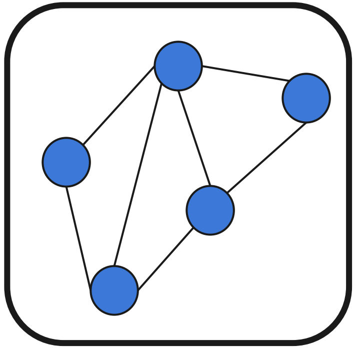
Total revenue grouped by order date



Local Aggregation - each group maps to a vertex

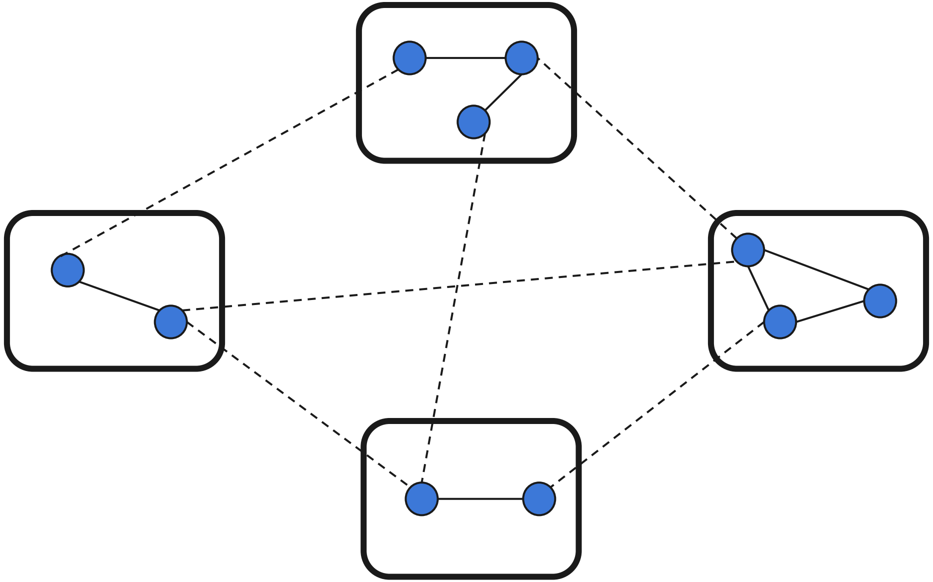
Experimental Evaluation: two settings

Intra-server parallelism



server

Distributed cluster parallelism



cluster

Single-server Experiments

Relational:

- **PostgreSQL** (psql)
- **RDBMS-X** (rdbmsX)
  - **In-memory Column store** (rdbmsX_im)
- **RDBMS-Y** (rdbmsY)
- **Spark/Spark SQL**

VS

Graph:

- **TigerGraph** (TAG_tg)
  - Native graph storage
  - High-level query language
  - Vertex-centric computation model

Hardware: 32 vCPU, 244 GB RAM

Dataset and Queries: TPC-H and TPC-DS benchmarks at SF-30, 50, 75

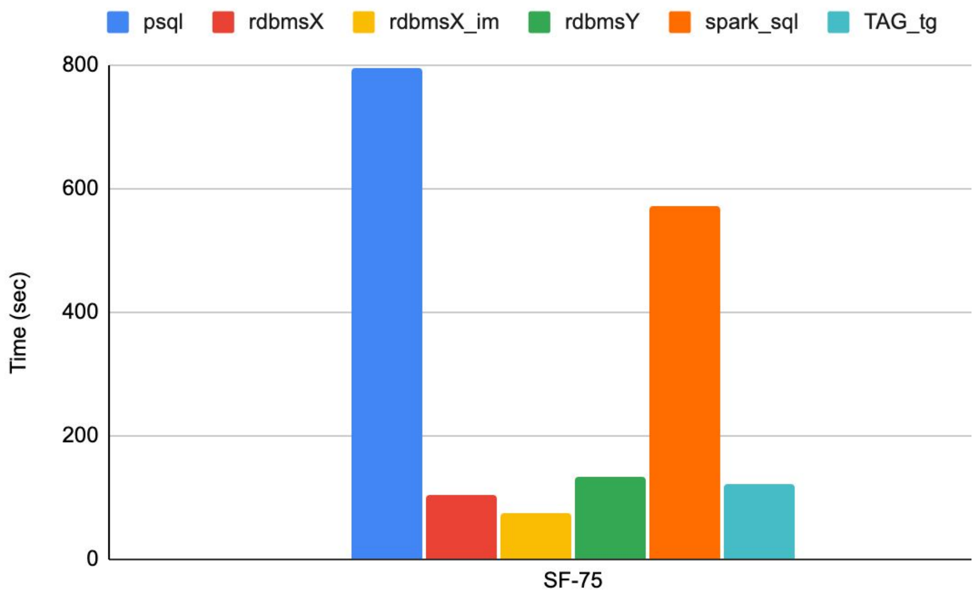
Methodology: measured warm cache runs

Single-server Experiments: TPC-H (22 queries)

In aggregate TAG-join on TigerGraph:

- **7x faster** than PostgreSQL
- **4.7x faster** than Spark SQL
- competitive with RDBMS-X and RDBMS-Y.

RDBMS-X column store outperforms by 1.6x

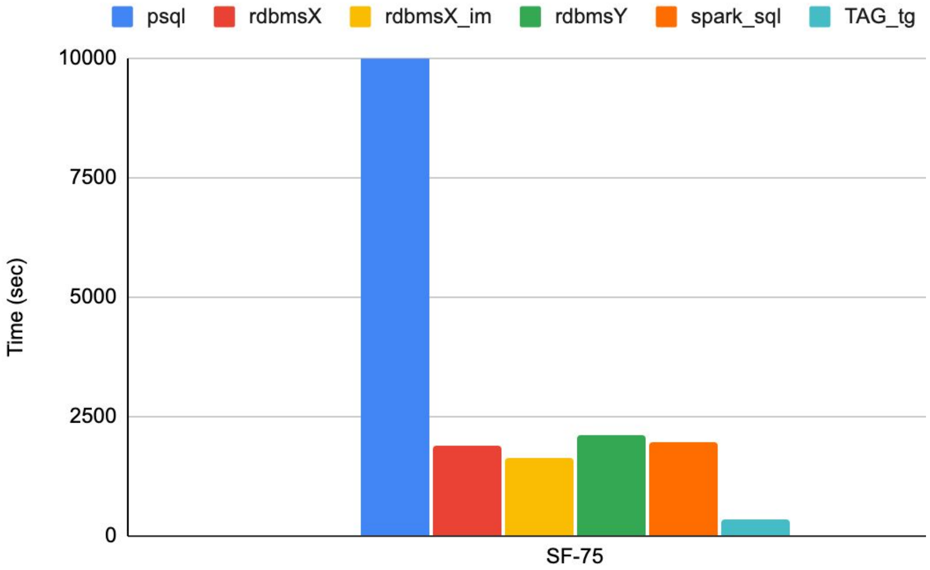


Aggregate runtimes (i.e. summed over all queries)

Single-server Experiments: TPC-DS (84 queries)

In aggregate TAG-join on TigerGraph:

- **28x faster** than PostgreSQL
- **6x** faster than RDBMS-Y
- **5x faster** than RDBMS-X
- **4.5x faster** than RDBMS-X column store
- **5.6x faster** than Spark SQL



Aggregate runtimes (i.e. summed over all queries)

Distributed Experiments

Relational:

- **Spark/Spark SQL 3.0.1**

VS

Graph:

- **TigerGraph 3.0** (TAG_tg)

Hardware: cluster of 6 machines, each with 16 vCPU, 64 GB RAM

Dataset and Queries: TPC-H and TPC-DS benchmarks at SF- 75

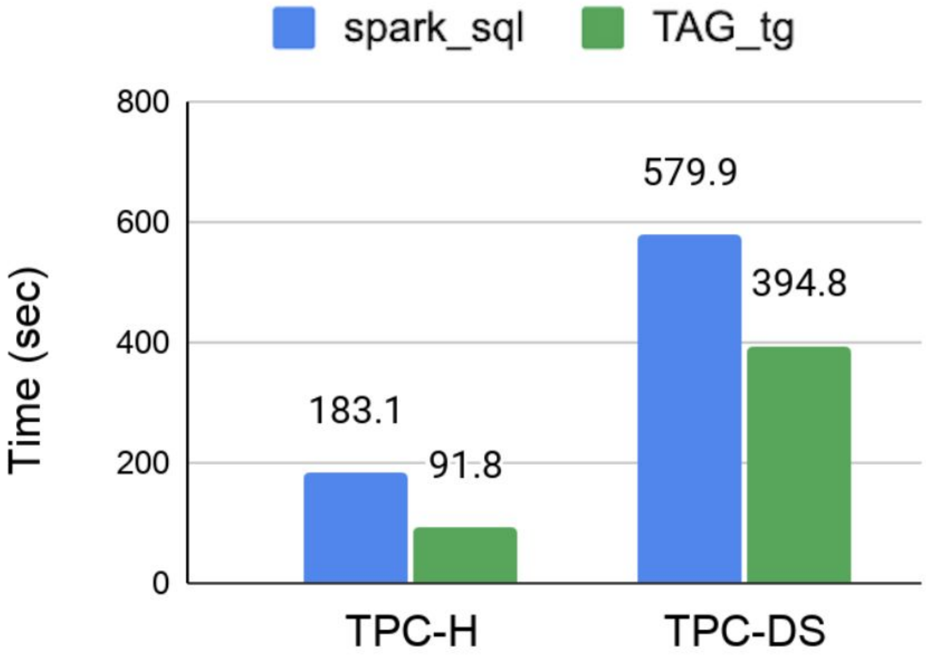
Distributed Experiments: Aggregate Runtimes

TPC-H queries:

- **TAG-join** is **2x faster** than Spark SQL.

TPC-DS queries:

- **TAG-join** is **1.5x faster** than Spark SQL



Aggregate runtimes (i.e. summed over all queries) at SF-75

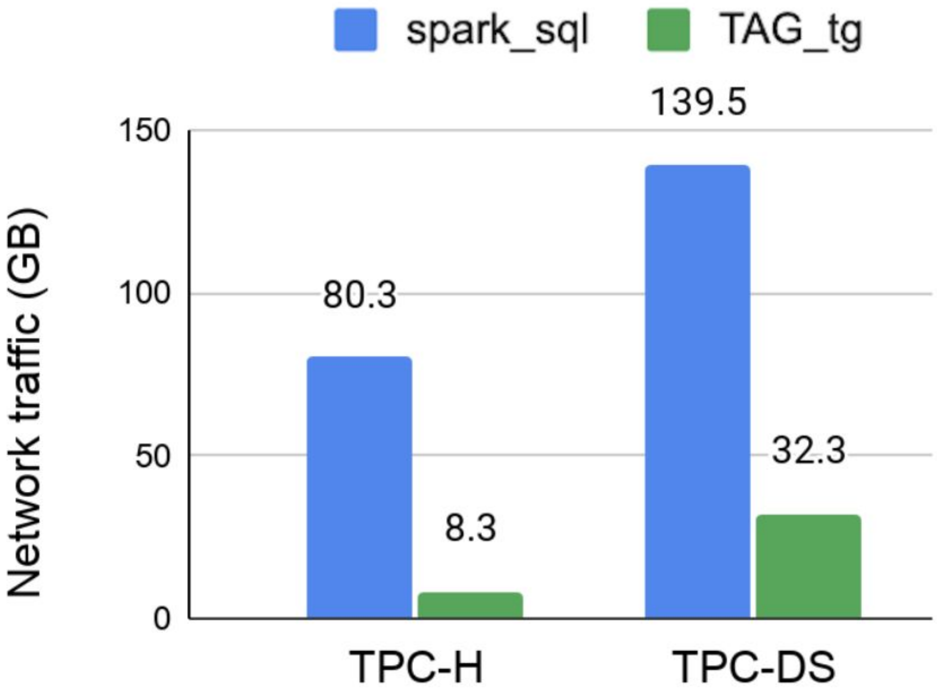
Distributed Experiments: Network Traffic

TPC-H queries:

- Spark SQL incurs **9x more** traffic

TPC-DS queries:

- Spark SQL incurs **4x more** traffic



Total incoming traffic at SF-75

We show that:

Vertex-centric parallelism is extremely well-suited to compute SQL queries with provable theoretical guarantees and good performance as validated by our experiments.

For details refer to:

“Vertex-centric Parallel Computation of SQL queries ”

Ainur Smagulova, Alin Deutsch, SIGMOD 2021

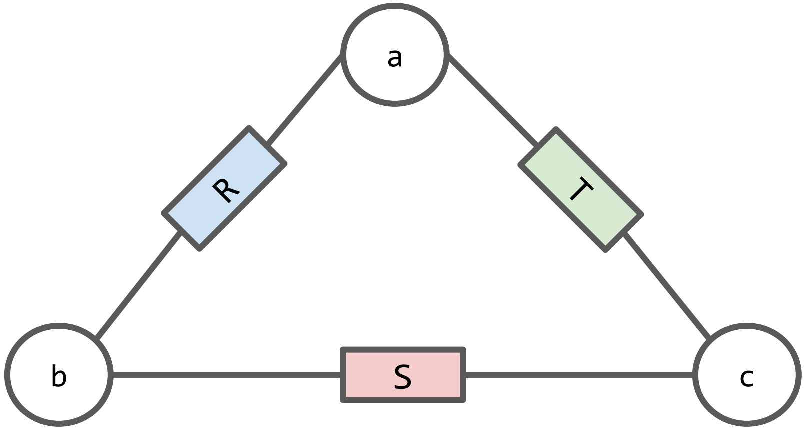
“Vertex-centric Parallel Computation of SQL queries (extended version) ” (ArXiv)

http://cseweb.ucsd.edu/~asmagulo/

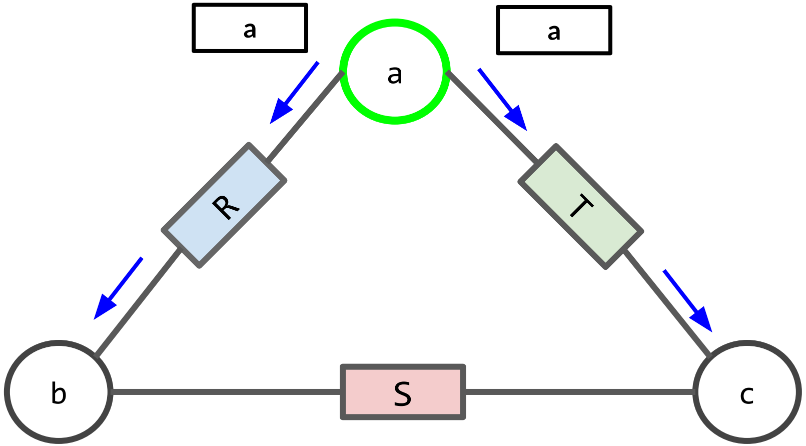
APPENDIX

Triangle Query Algorithm

$R(A,B) \bowtie S(B,C) \bowtie T(A,C)$



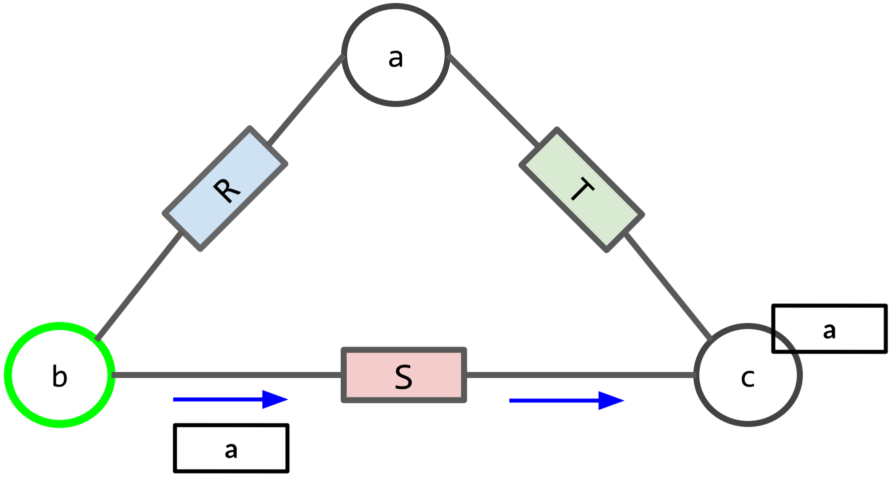
Triangle Query Algorithm (naive algorithm)



a sends its value in both directions

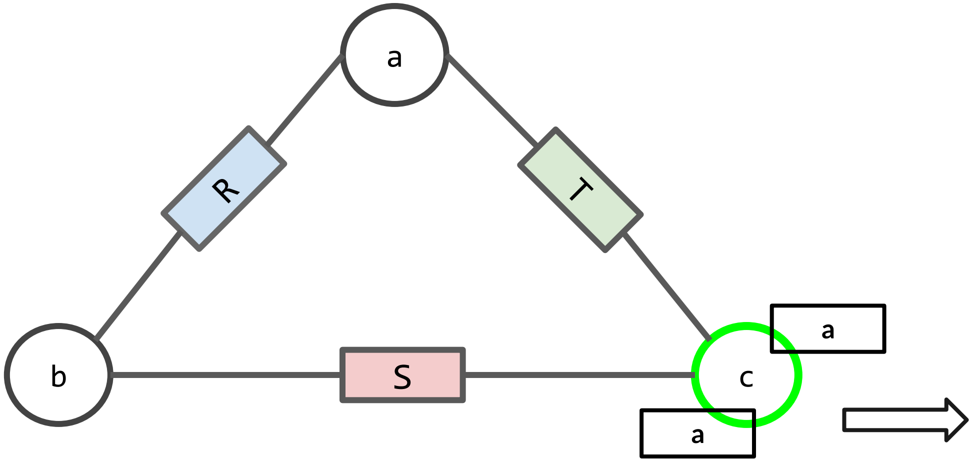
via path that leads to **c**

Triangle Query Algorithm



b sends the received message(s) further to c value

Triangle Query Algorithm

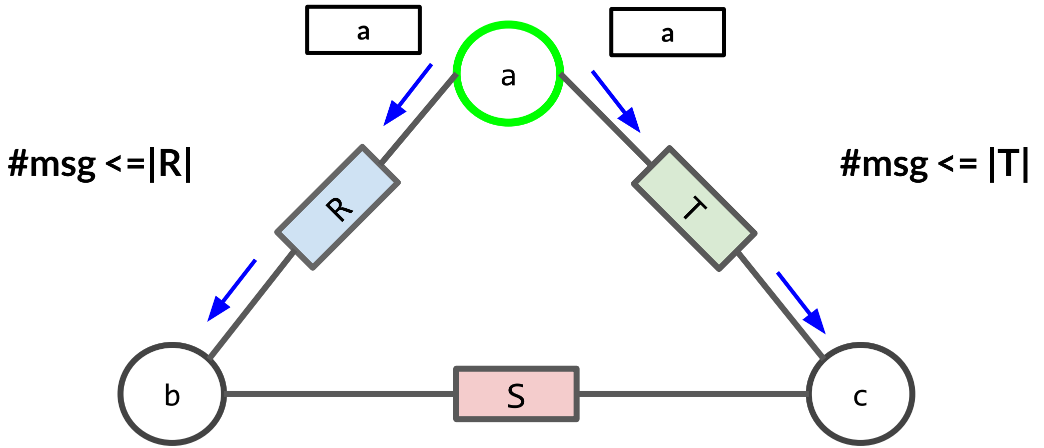


c intersects a values received from both sides.

a-values that survive the intersection are in the output

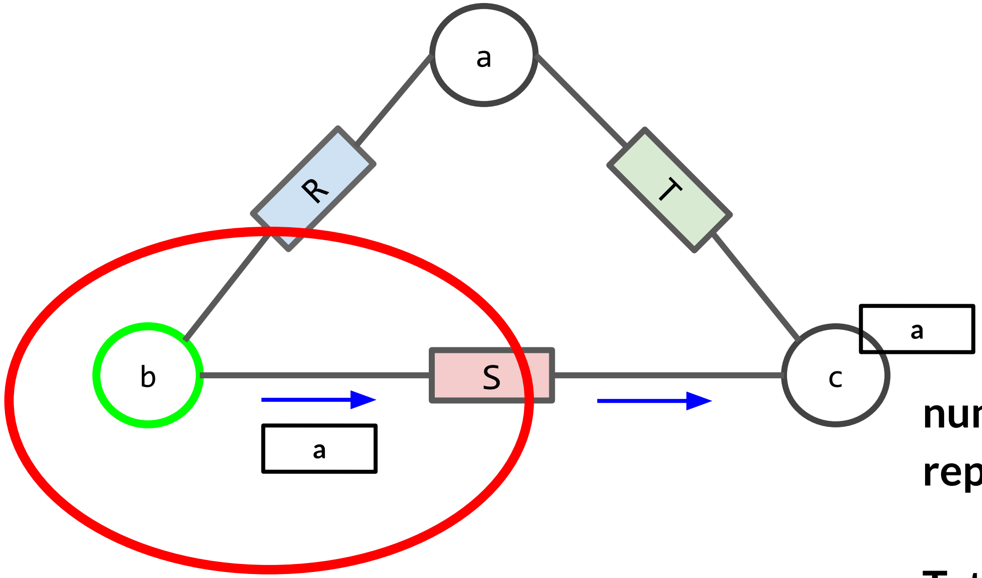
(a,b,c)

Triangle Query: communication cost analysis



O(|R|) + O(|T|)

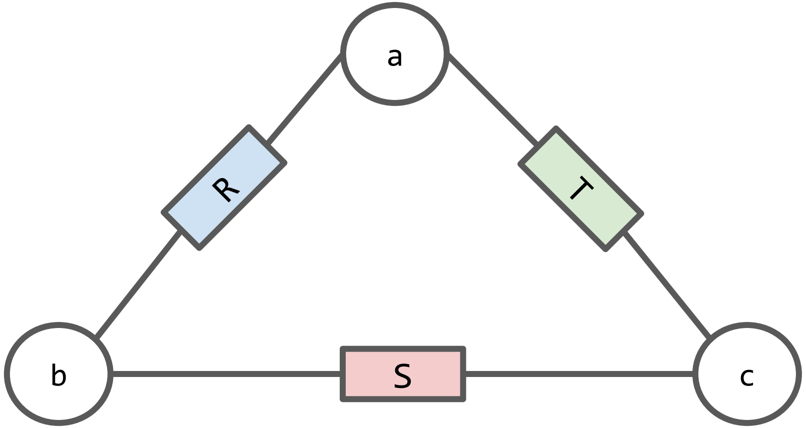
Triangle Query: communication cost analysis



number of a-values received <= |R |
replication of each a value <= |S|

Total #messages sent: O(|R | * |S|) = O(IN^2)
(worst-case instance)

Triangle Query Algorithm (WCO)



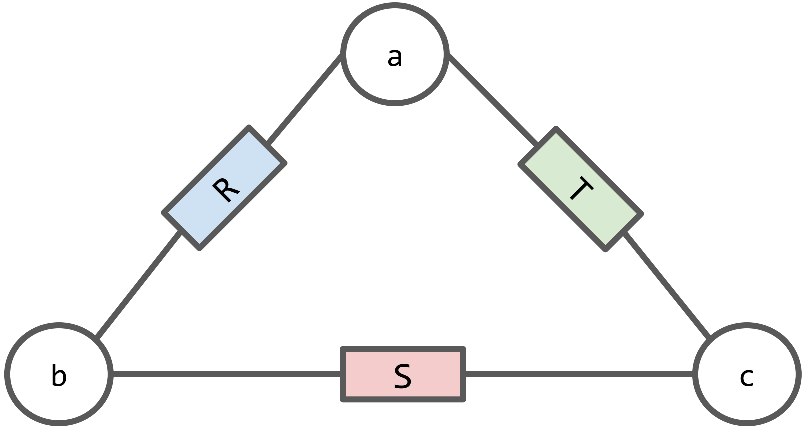
Vertex-centric approach:

O(AGM) communication cost

O(AGM) computation cost

AGM - worst-case bound on the output size

Triangle Query Algorithm



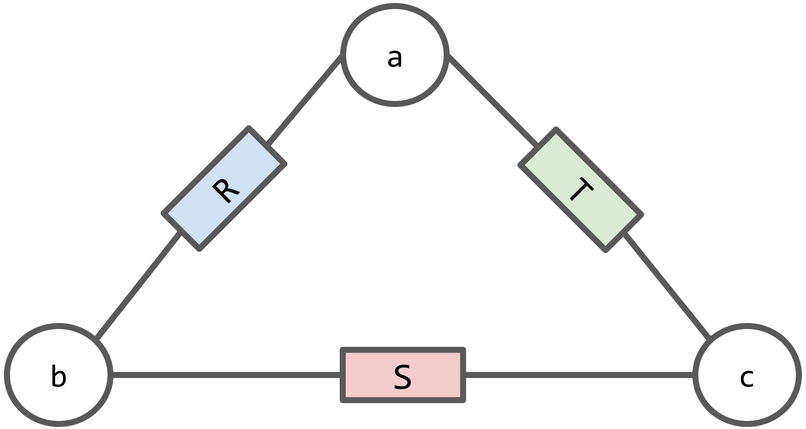
Vertex-centric approach:

$O(IN^{3/2})$ communication cost

$O(IN^{3/2})$ computation cost

Algorithm idea [Ngo12] : handle heavy (highly skewed) and light values separately - applied to graphs.

Triangle query algorithm



$R(A,B) \bowtie S(B,C) \bowtie T(A,C)$

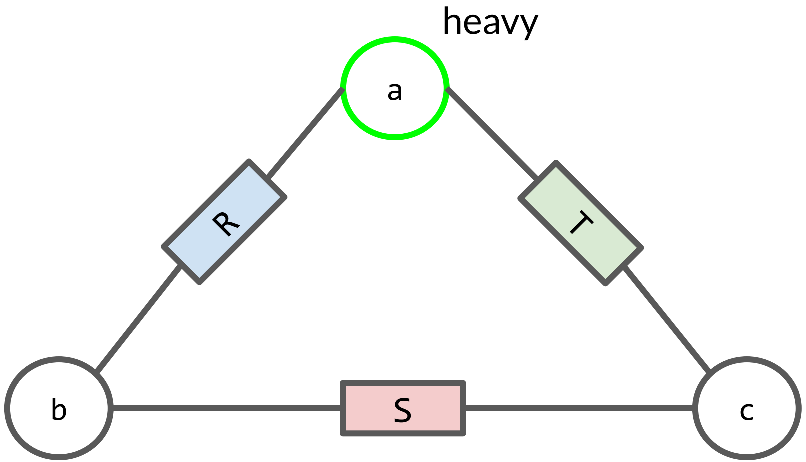
Split original query into two:

$[(R^{heavy} \bowtie S) \ltimes T)] \cup [(R^{light} \bowtie T) \ltimes S)$

a is heavy:
If $|R_{A=a}| > \theta$ then $(a,b) \rightarrow R^{heavy}$

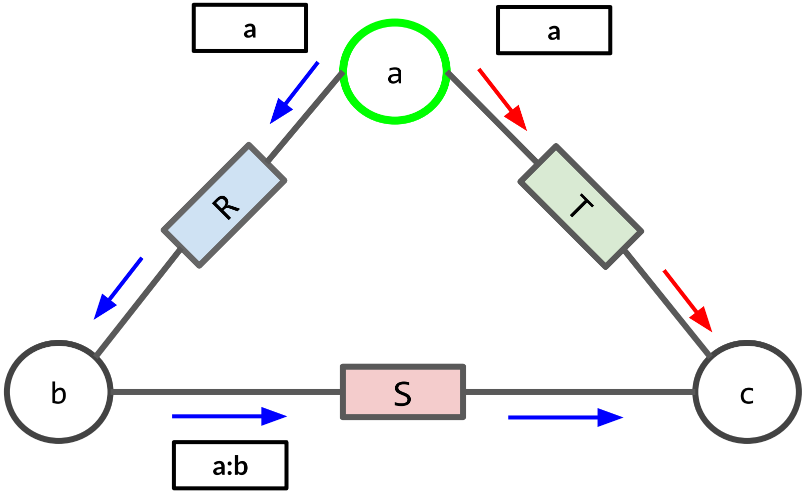
a is light:
If $|R_{A=a}| <= \theta$ then $(a,b) \rightarrow R^{light}$

Triangle query algorithm - Heavy



$(R^{heavy} \bowtie S) \ltimes T$

Triangle query algorithm - Heavy



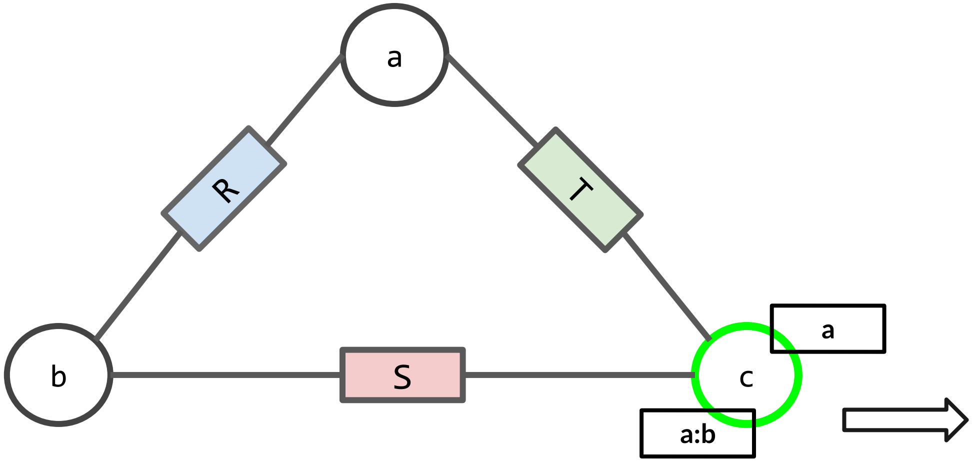
$(R^{heavy} \bowtie S) \ltimes T$

a sends its value in both directions

via path that leads to **c**

Triangle query algorithm - Heavy

$(R^{heavy} \bowtie S) \ltimes T$



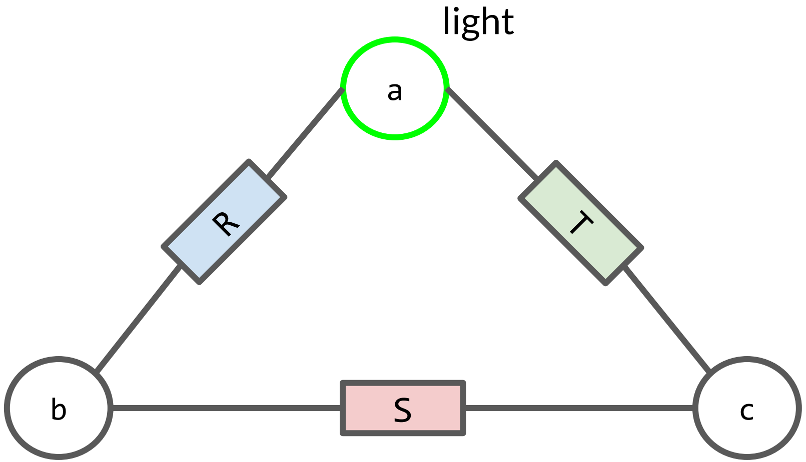
c intersects **a**-values received from both sides.

a-values that survive the intersection are in output

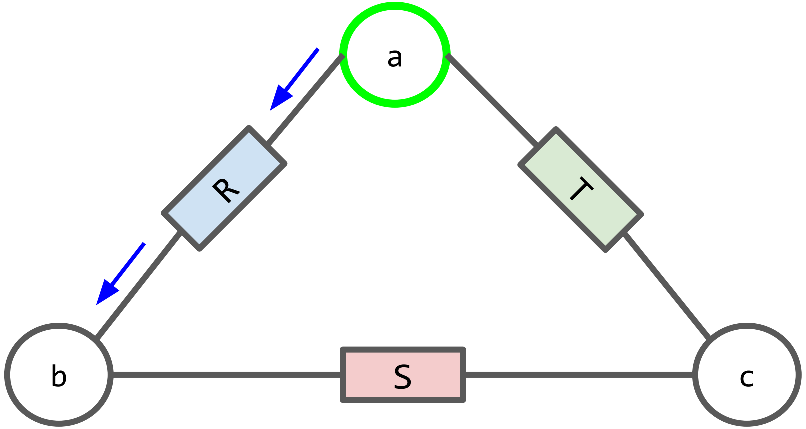
(a,b,c)

Triangle query algorithm - Light

$(R^{light} \bowtie T) \ltimes S$



Triangle query algorithm - Light

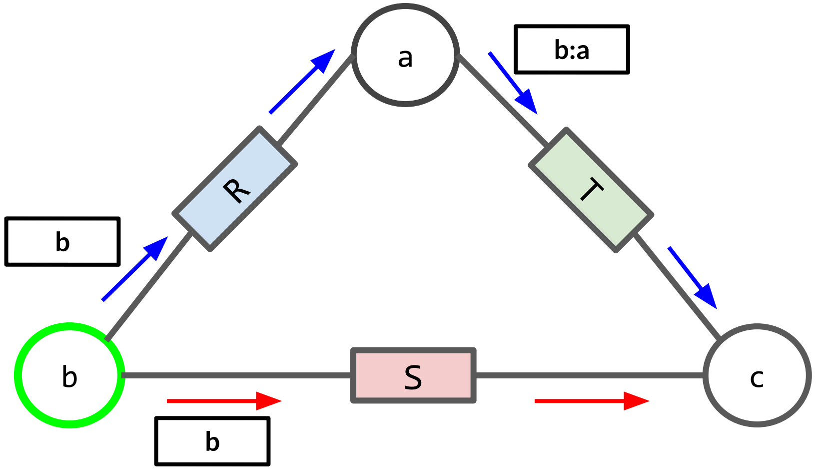


$(R^{light} \bowtie T) \ltimes S$

light **a**-values send “wake-up” messages to all the b-values that are connected to a

Triangle query algorithm - Light

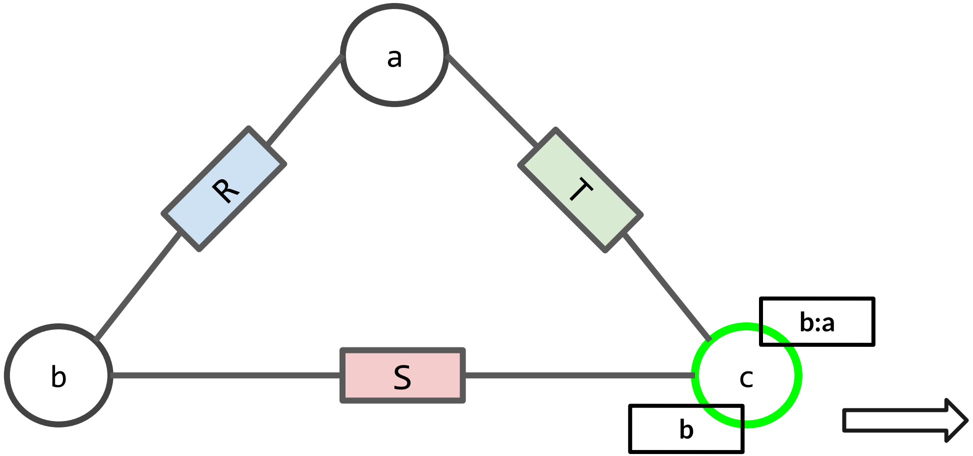
$(R^{light} \bowtie T) \ltimes S$



b sends its value in both directions

via path that leads to **c**

Triangle query algorithm - Light



$(R^{light} \bowtie T) \ltimes S$

c intersects **b** values received from both sides.

b-values that survive the intersection are in output

(a,b,c)

Triangle query: communication cost analysis

Heavy: $|R| + |T| + |R| / \theta * |S|$

Light: $|R| + |S| + \theta * |T|$

Setting $\theta = \sqrt{\frac{|R| \cdot |S|}{|T|}}$

Total Cost: O($\sqrt{|R| \cdot |S| \cdot |T|}$) [AGM bound]

Note: if $|R| = |S| = |T| = N$

total cost is $O(N^{3/2})$ [AGM bound]

$\theta = \sqrt{|N|}$