Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case Optimal Joins (WCOJs)**
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   2.2 System Integration Approaches
      i) **Index-based WCOJs (Graphflow & EmptyHeaded)**
      ii) **Hash-based WCOJs (Umbra)**
   - **Optimization approaches (cost-based DP, GHD, rule-based)**

3) **Factorized Query Processing**

Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case Optimal Joins (WCOJs)**
   Handling Intermediate Size Growth for Cyclic Joins
   2.1. Foundations
   2.2 System Integration Approaches
      i) **Index-based WCOJs (Graphflow & EmptyHeaded)**
      ii) **Hash-based WCOJs (Umbra)**
   - **Optimization approaches (cost-based DP, GHD, rule-based)**

3) **Factorized Query Processing**

Index-based Worst-case Optimal Joins

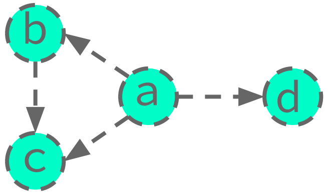
Index-based Worst-case Optimal Joins

Requires pre-sorted indexes which in graph terms map to adjacency list indexes. Graphflow

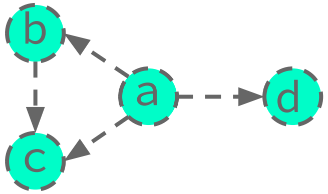
Relational Systems such as EmptyHeaded, LogicBlox, and relationalAI use sorted trie indexes.

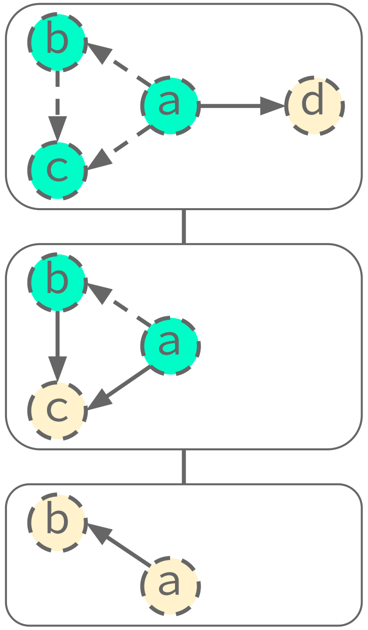
Query Vertex Orderings (QVOs)

Query Vertex Orderings (QVOs)



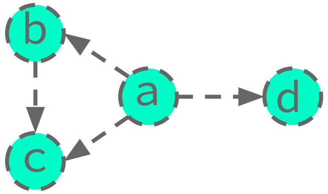
Query Vertex Orderings (QVOs)

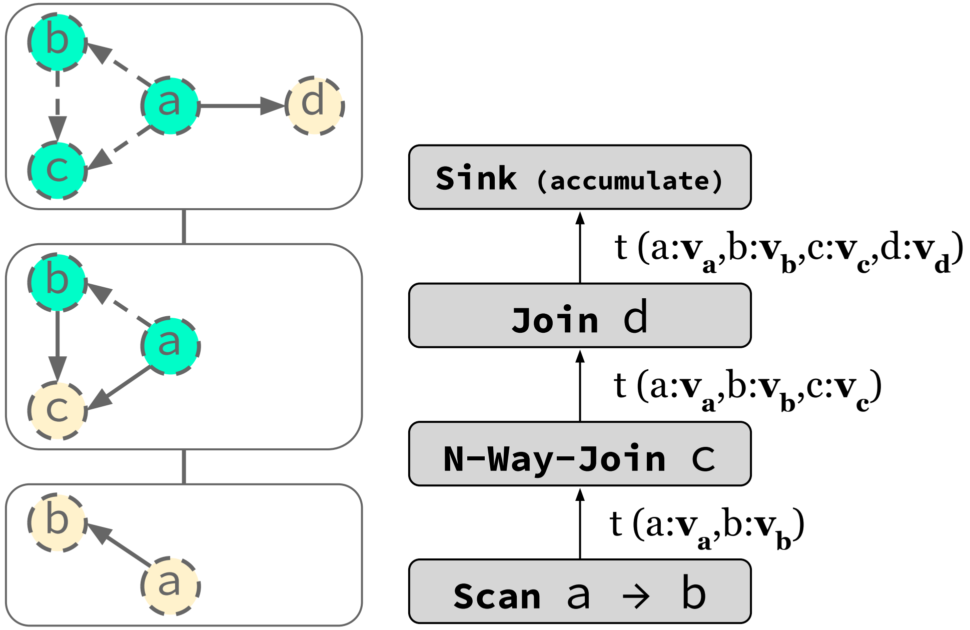




Plan$_1$[a,b,c,d]

Query Vertex Orderings (QVOs)





Sink (accumulate)

$t\,(a{:}\mathbf{v}_a, b{:}\mathbf{v}_b, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\,(a{:}\mathbf{v}_a, b{:}\mathbf{v}_b, c{:}\mathbf{v}_c)$

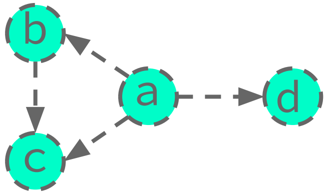
N-Way-Join c

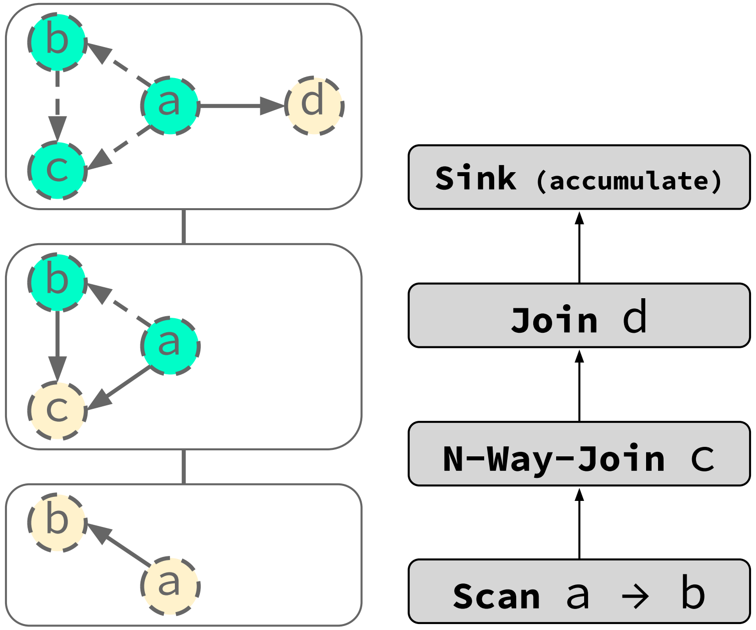
$t\,(a{:}\mathbf{v}_a, b{:}\mathbf{v}_b)$

Scan a → b

$\mathrm{Plan}_1[a,b,c,d]$

Query Vertex Orderings (QVOs)





Sink (accumulate)

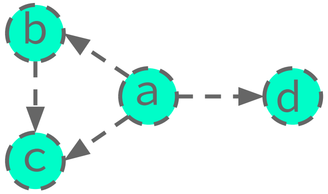
Join d

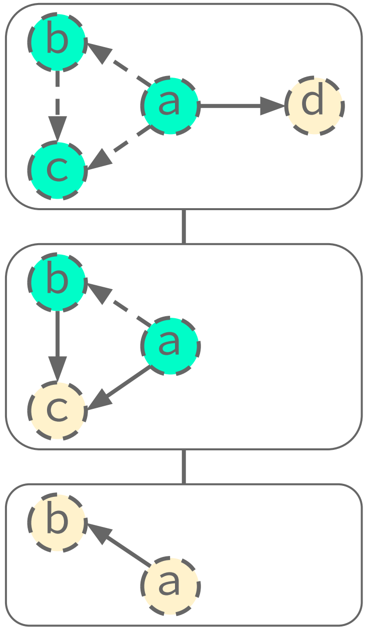
N-Way-Join c

Scan a → b

Plan_1[a,b,c,d]

Execution Simulation

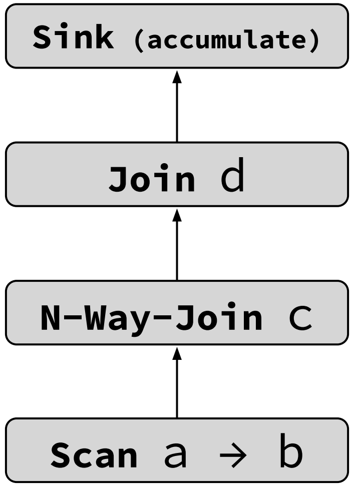




Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
-	-	-	-



Sink (accumulate)

Join d

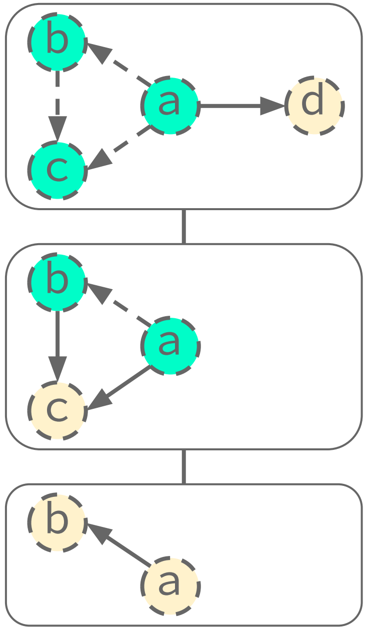
N-Way-Join c

Scan a → b

output relation

a	b	c	d

Execution Simulation



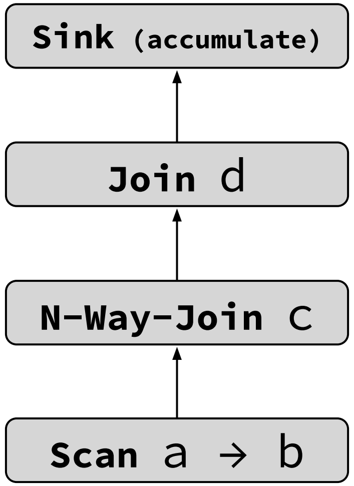
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
-	-	-	-

output relation

a	b	c	d



Sink (accumulate)

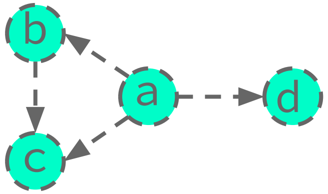
Join d

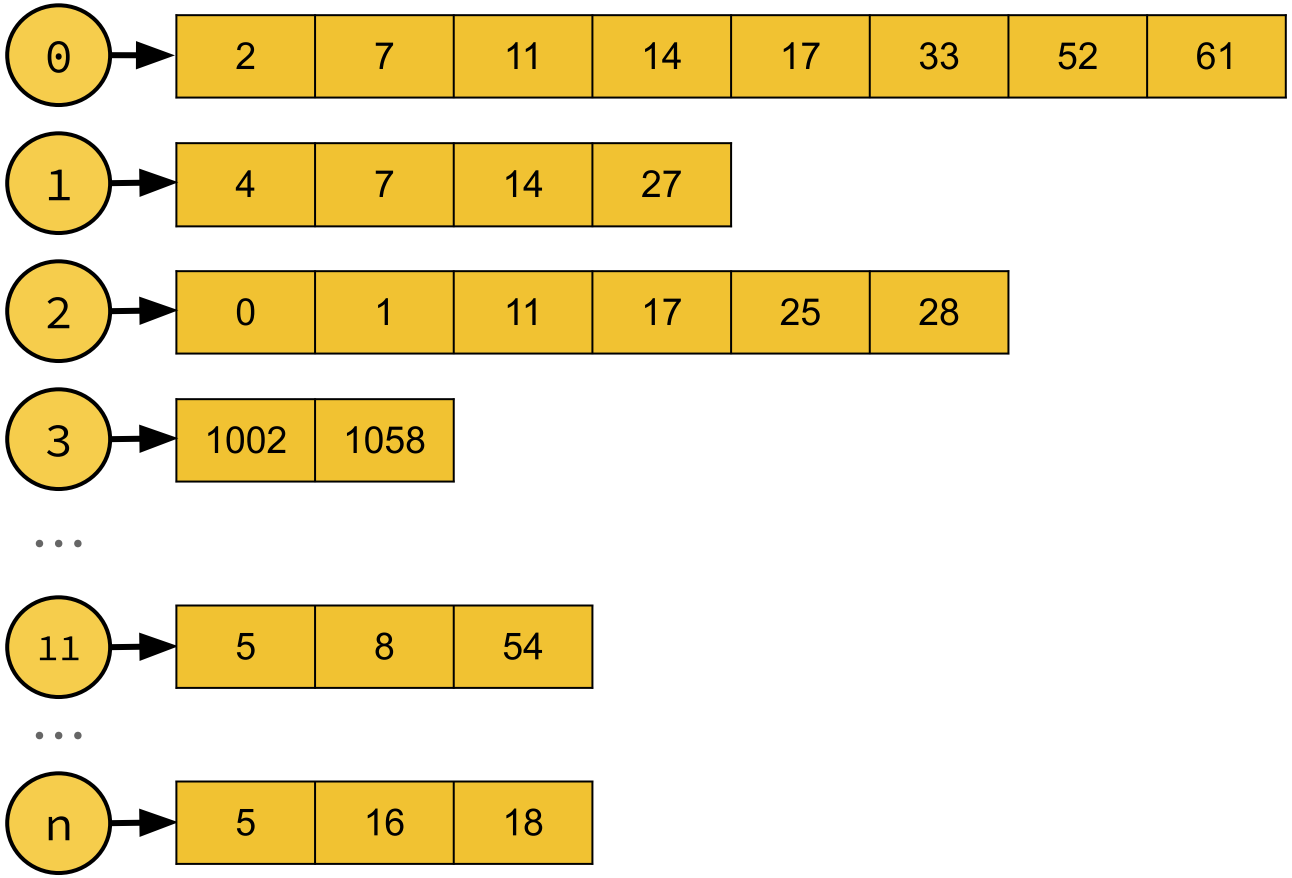
N-Way-Join c

Scan a → b

Edges Relation

FROM	TO





0 → 2 7 11 14 17 33 52 61

1 → 4 7 14 27

2 → 0 1 11 17 25 28

3 → 1002 1058

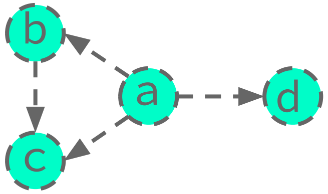
...

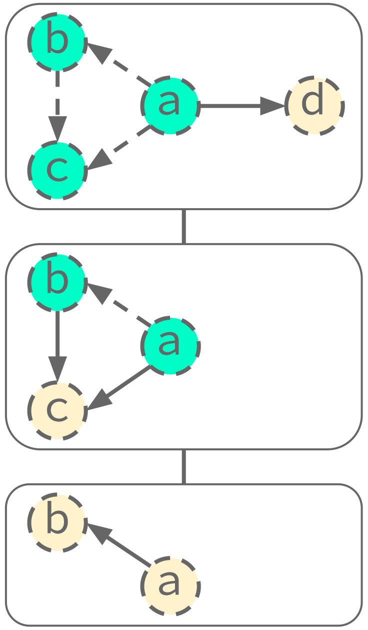
11 → 5 8 54

...

n → 5 16 18

Execution Simulation





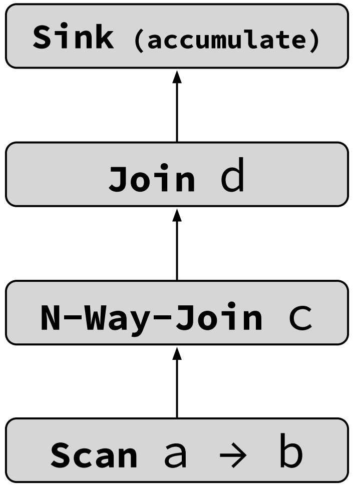
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
-	-	-	-

output relation

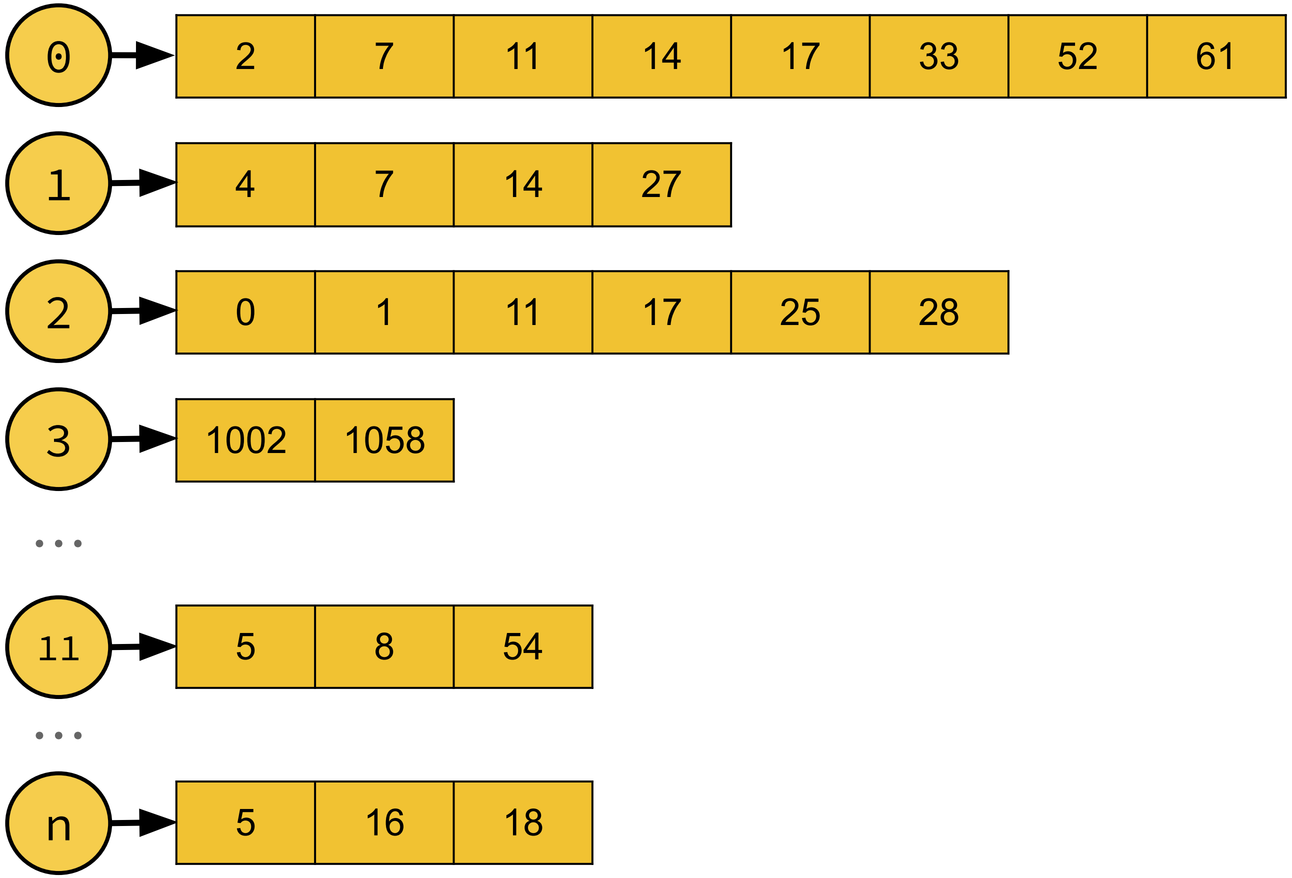
a	b	c	d


Sink (accumulate)

Join d

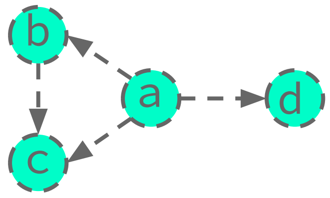
N-Way-Join c

Scan a → b



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
...								
11	5	8	54					
...								
n	5	16	18					

Execution Simulation

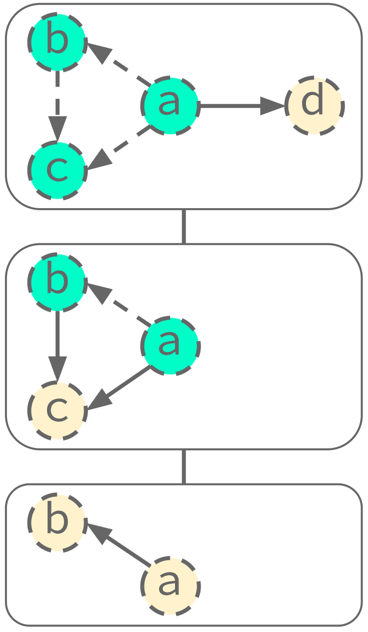


intermediate

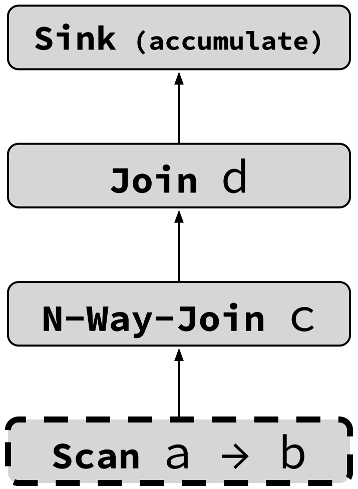
a	b	c	d
-	-	-	-

output relation

a	b	c	d



Plan$_1$[a,b,c,d]



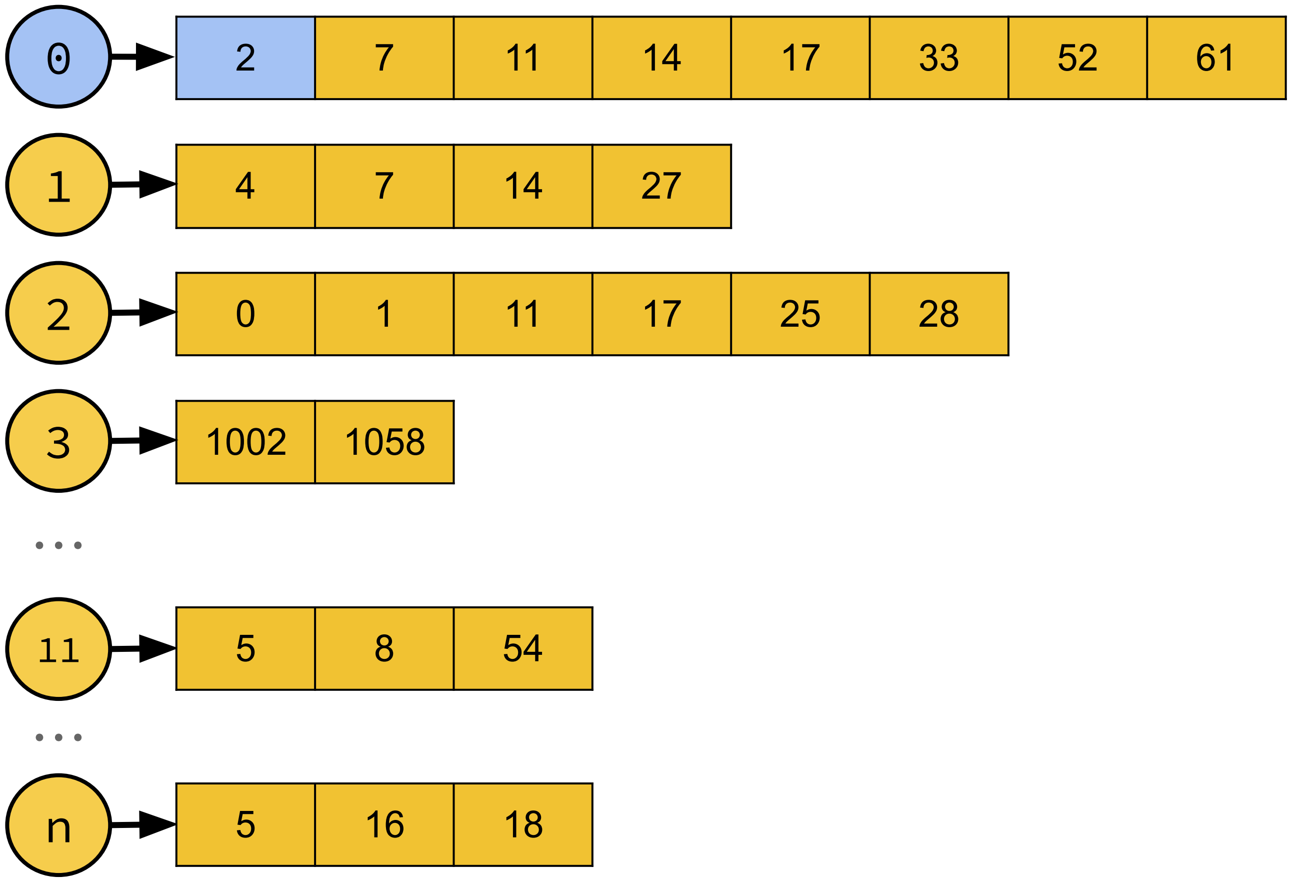
Sink (accumulate)

Join d

N-Way-Join c

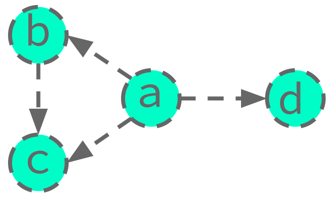
Scan a → b

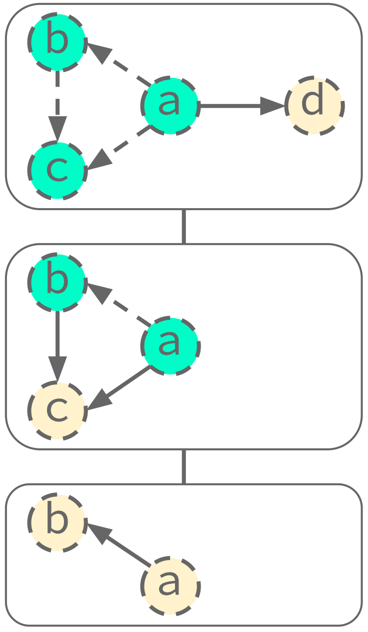
Fw(0)={2,7,11,…}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





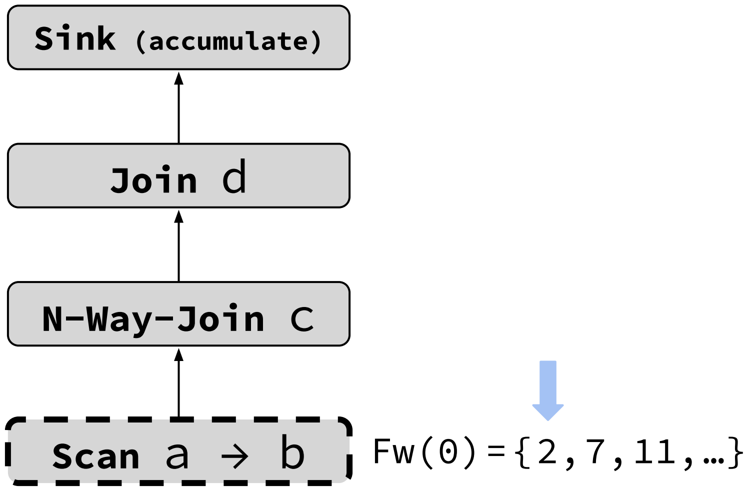
Plan$_1$[a,b,c,d]

intermediate

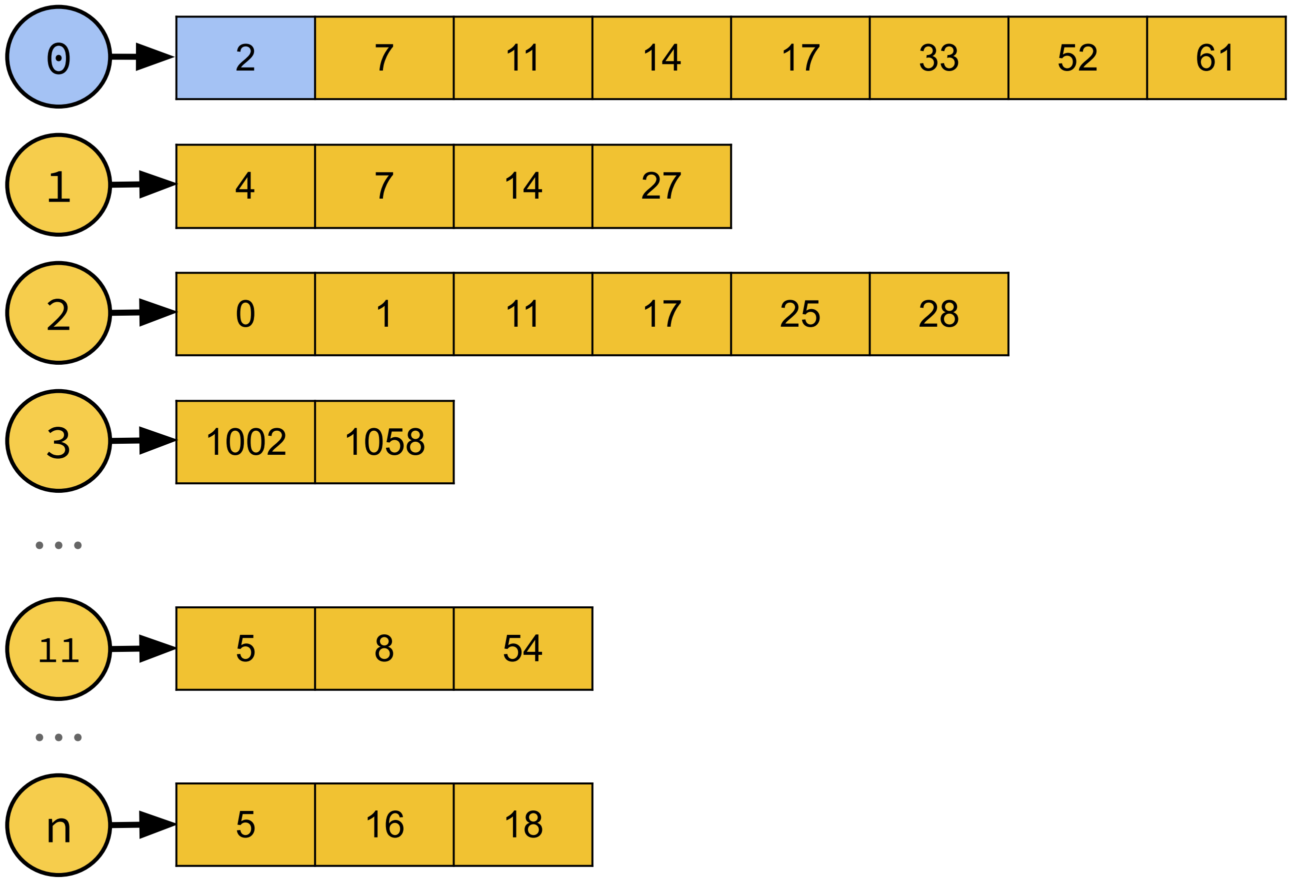
a	b	c	d
0	2	-	-

output relation

a	b	c	d

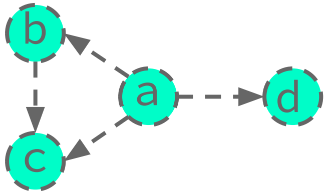


Fw(0) = {2, 7, 11, …}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation

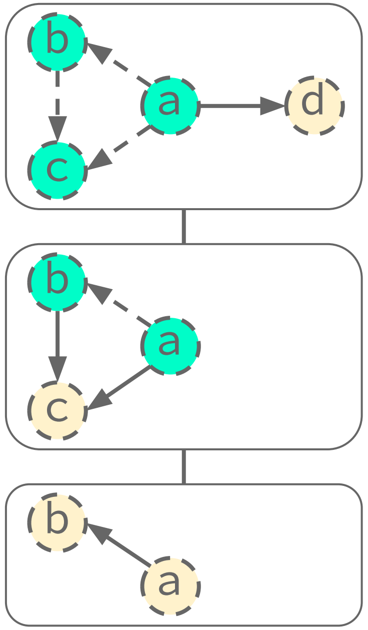


intermediate

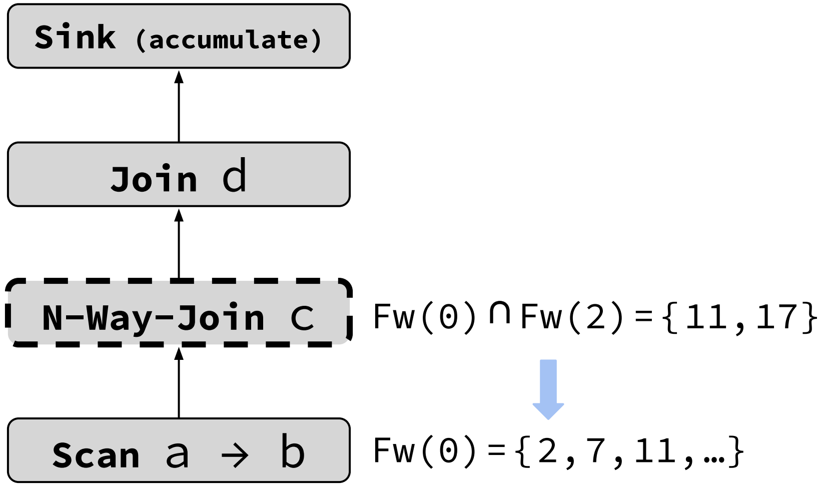
a	b	c	d
0	2	-	-

output relation

a	b	c	d



$Plan_1[a,b,c,d]$



Sink (accumulate)

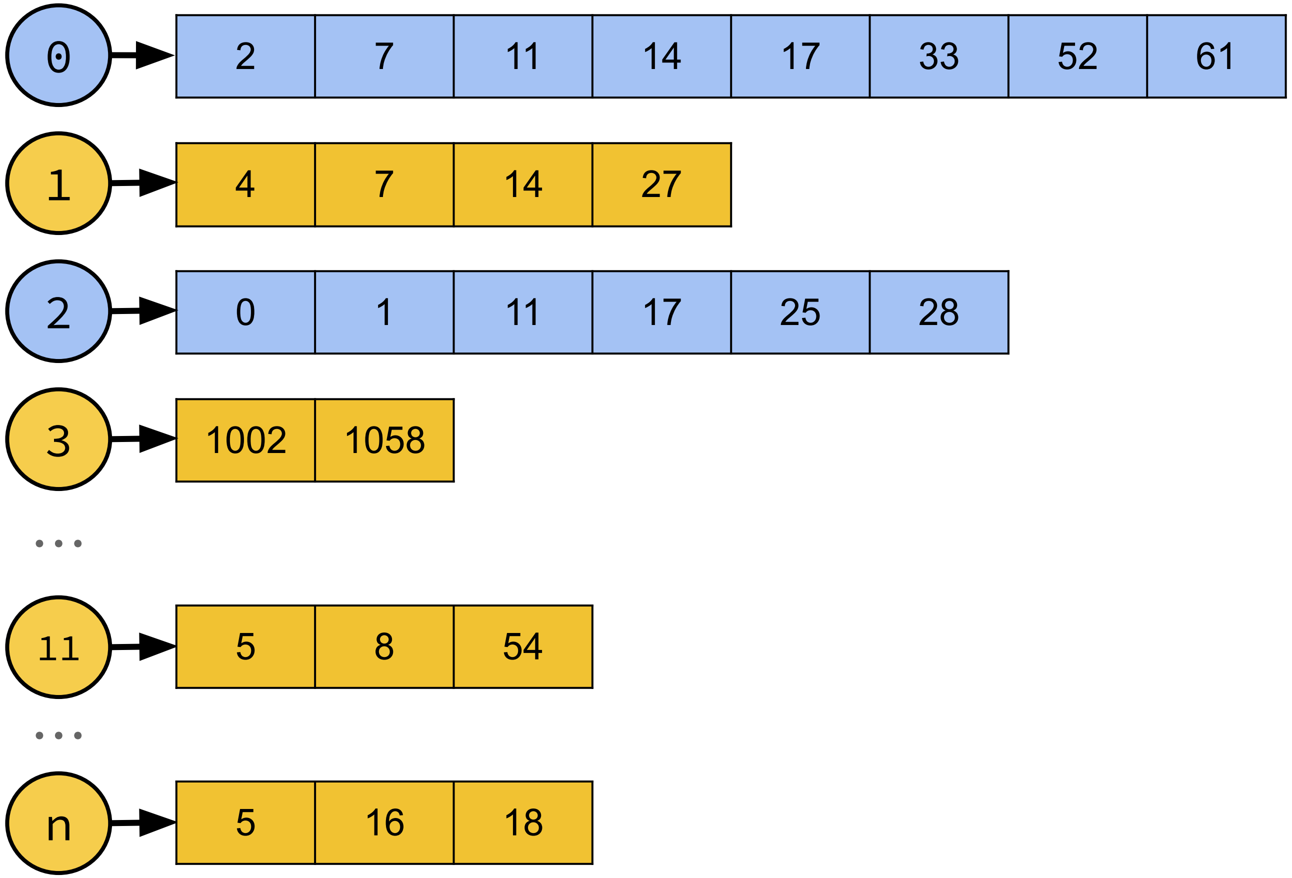
Join d

N-Way-Join c

Fw(0) ∩ Fw(2) = {11, 17}

Scan a → b

Fw(0) = {2, 7, 11, …}



0 → 2, 7, 11, 14, 17, 33, 52, 61

1 → 4, 7, 14, 27

2 → 0, 1, 11, 17, 25, 28

3 → 1002, 1058

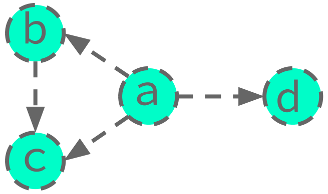
…

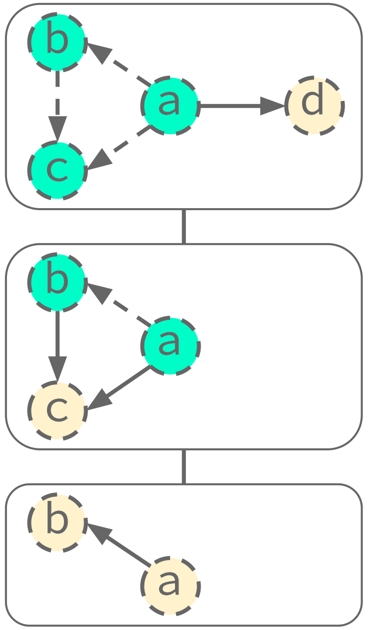
11 → 5, 8, 54

…

n → 5, 16, 18

Execution Simulation





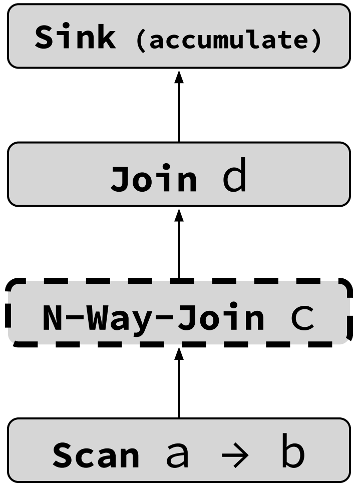
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	-

output relation

a	b	c	d



Sink (accumulate)

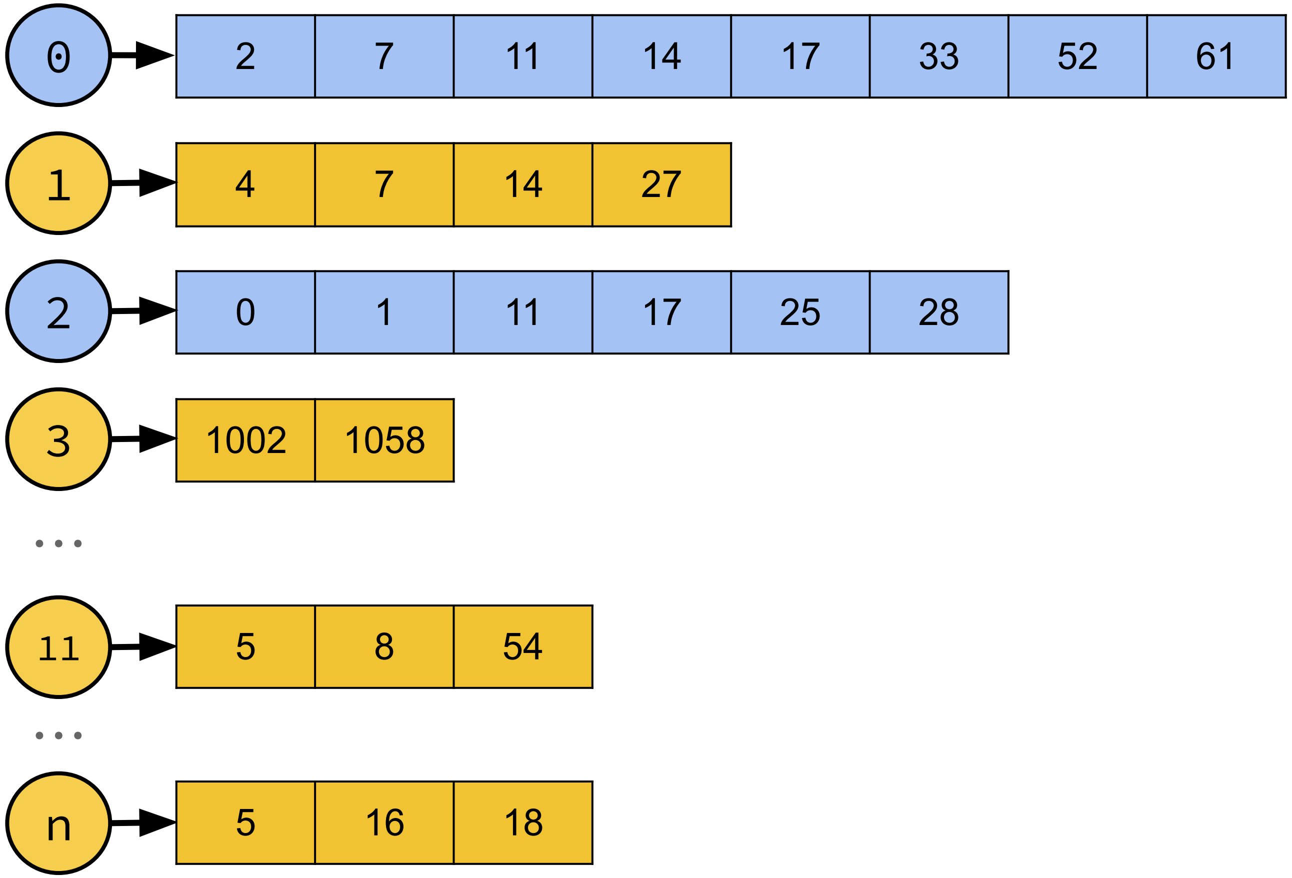
Join d

N-Way-Join c

Scan a → b

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, …}



0 → 2, 7, 11, 14, 17, 33, 52, 61

1 → 4, 7, 14, 27

2 → 0, 1, 11, 17, 25, 28

3 → 1002, 1058

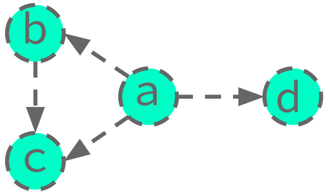
...

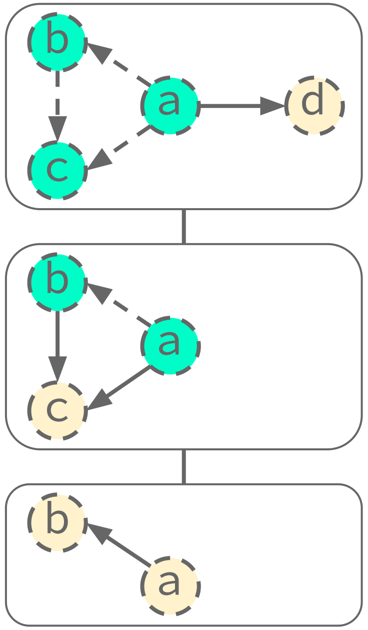
11 → 5, 8, 54

...

n → 5, 16, 18

Execution Simulation





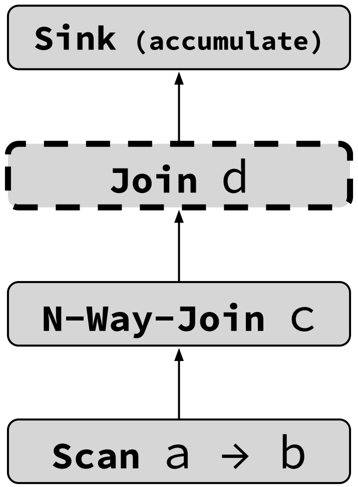
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	-

output relation

a	b	c	d



Sink (accumulate)

Join d

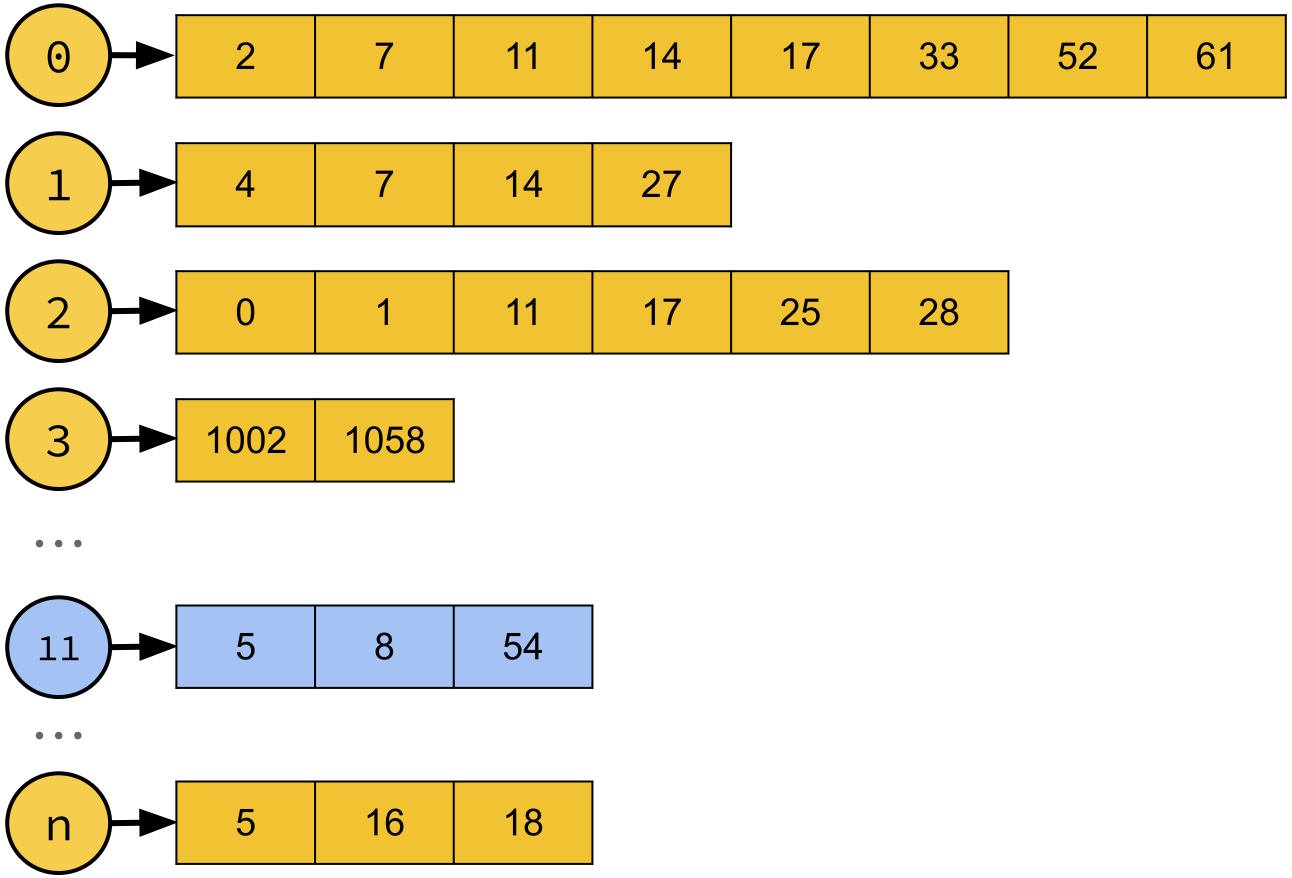
N-Way-Join c

Scan a → b

Fw(11)={5,8,54}

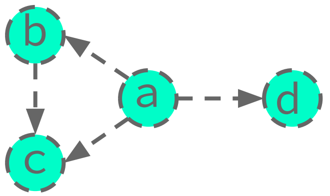
Fw(0)∩Fw(2)={11,17}

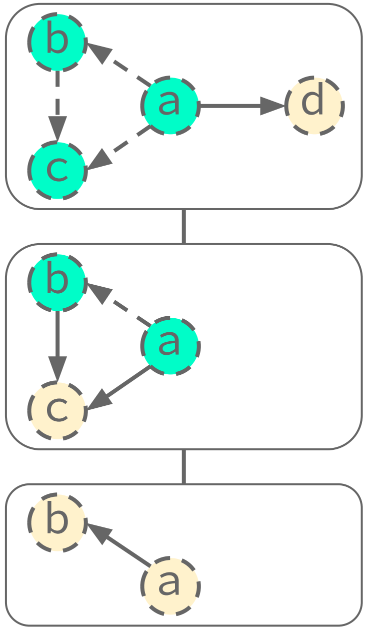
Fw(0)={2,7,11,…}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





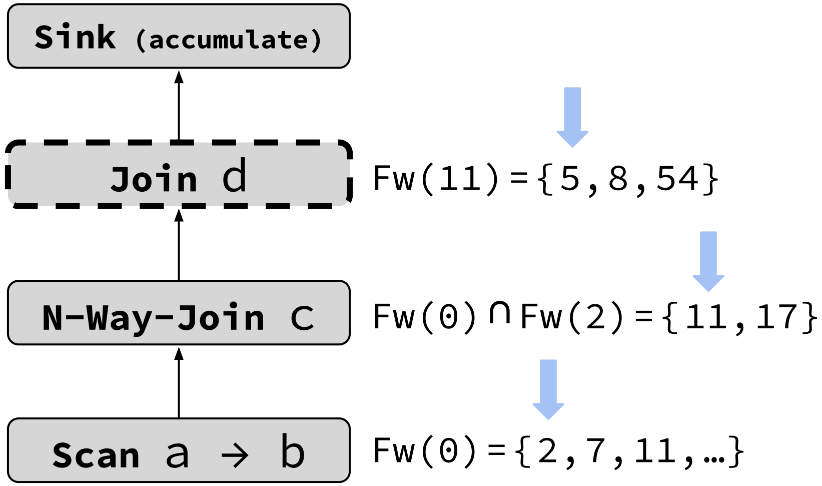
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	5

output relation

a	b	c	d

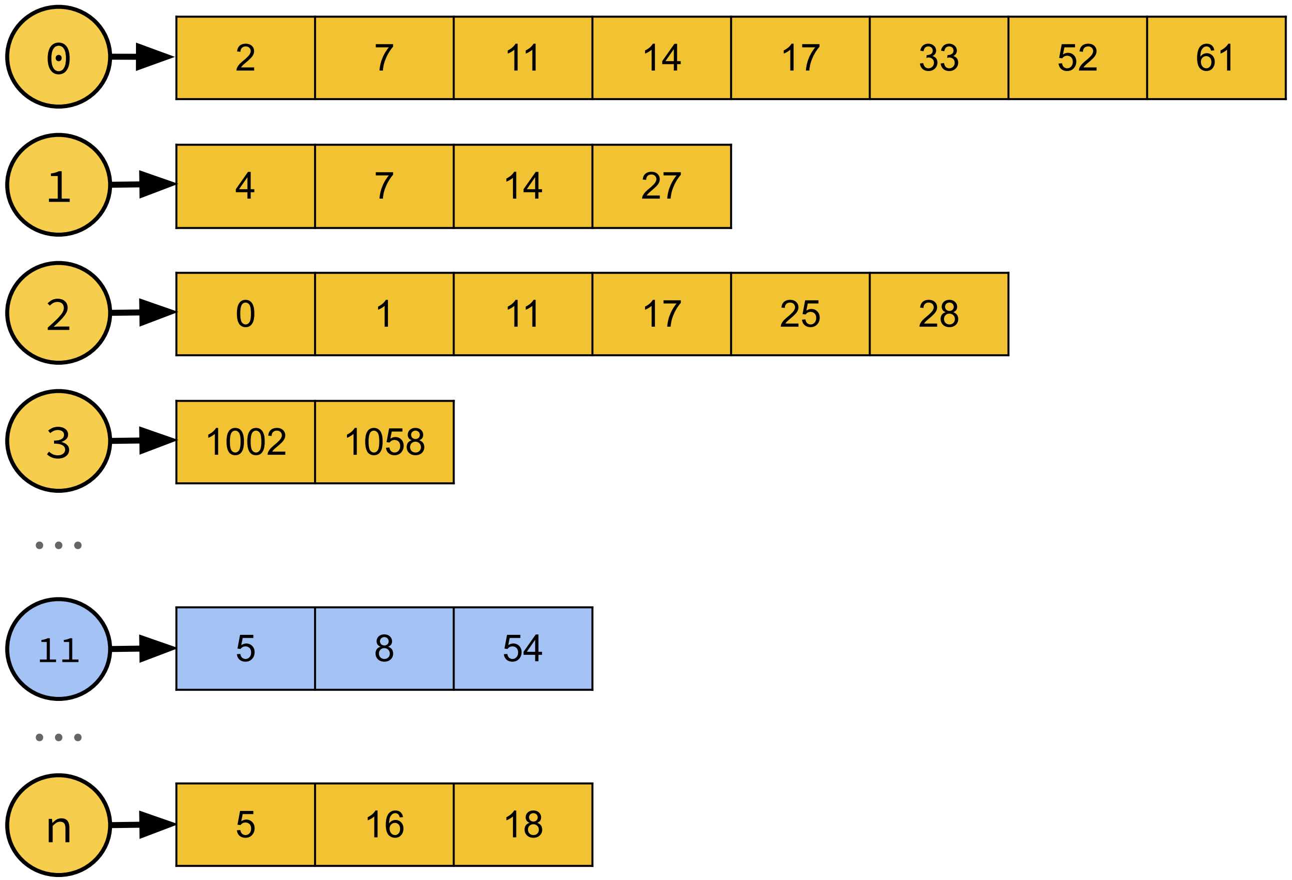


Sink (accumulate)

Join d — Fw(11) = {5, 8, 54}

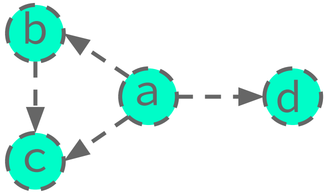
N-Way-Join c — Fw(0) ∩ Fw(2) = {11, 17}

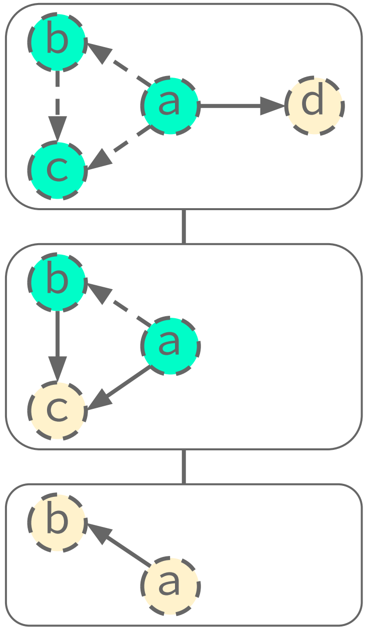
Scan a → b — Fw(0) = {2, 7, 11, …}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





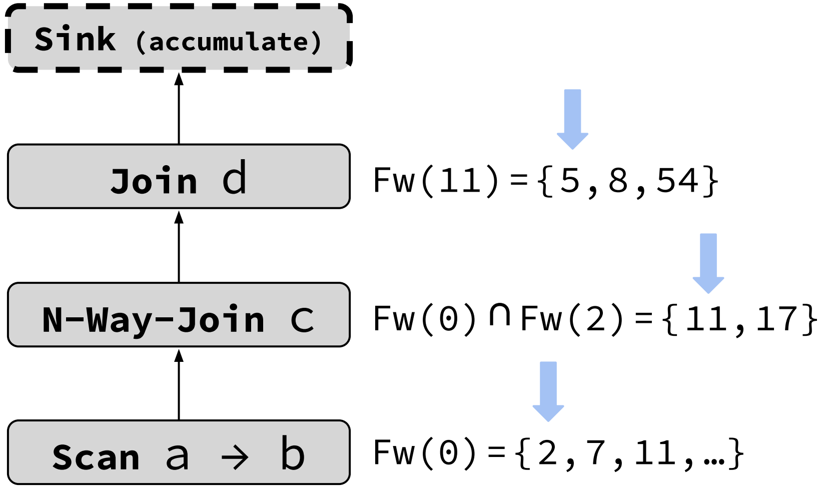
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	5

output relation

a	b	c	d



Sink (accumulate)

Join d

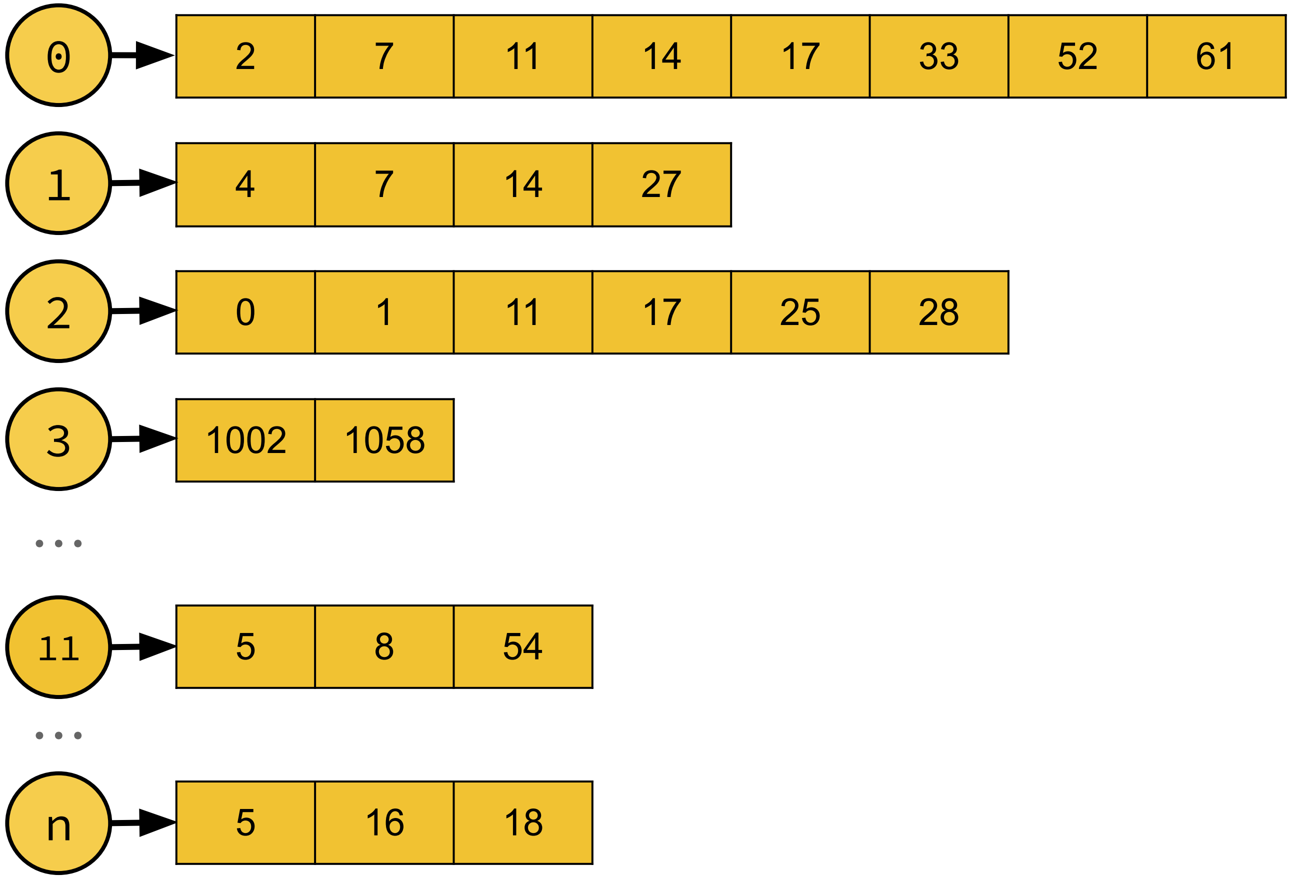
Fw(11)={5,8,54}

N-Way-Join c

Fw(0)∩Fw(2)={11,17}

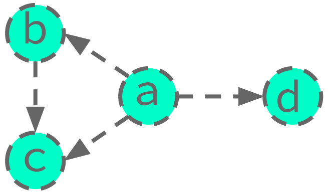
Scan a → b

Fw(0)={2,7,11,…}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
...								
11	5	8	54					
...								
n	5	16	18					

Execution Simulation

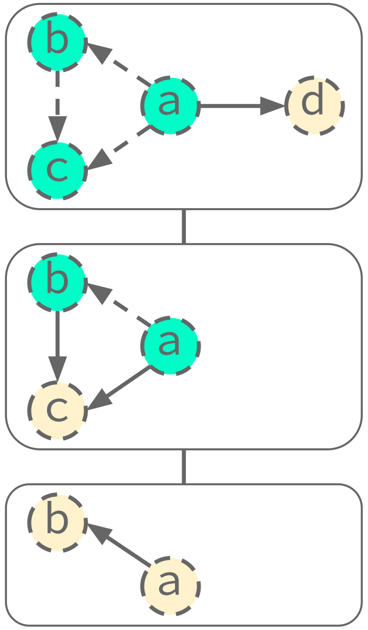


intermediate

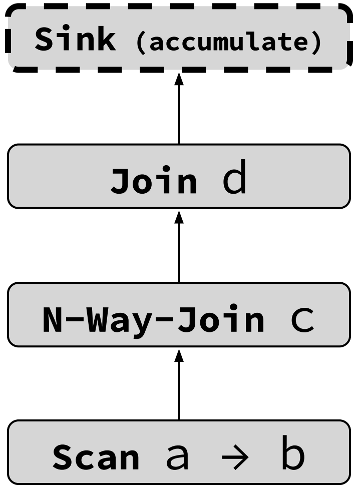
a	b	c	d
0	2	11	5

output relation

a	b	c	d
0	2	11	5



Plan$_1$[a,b,c,d]



Sink (accumulate)

Join d

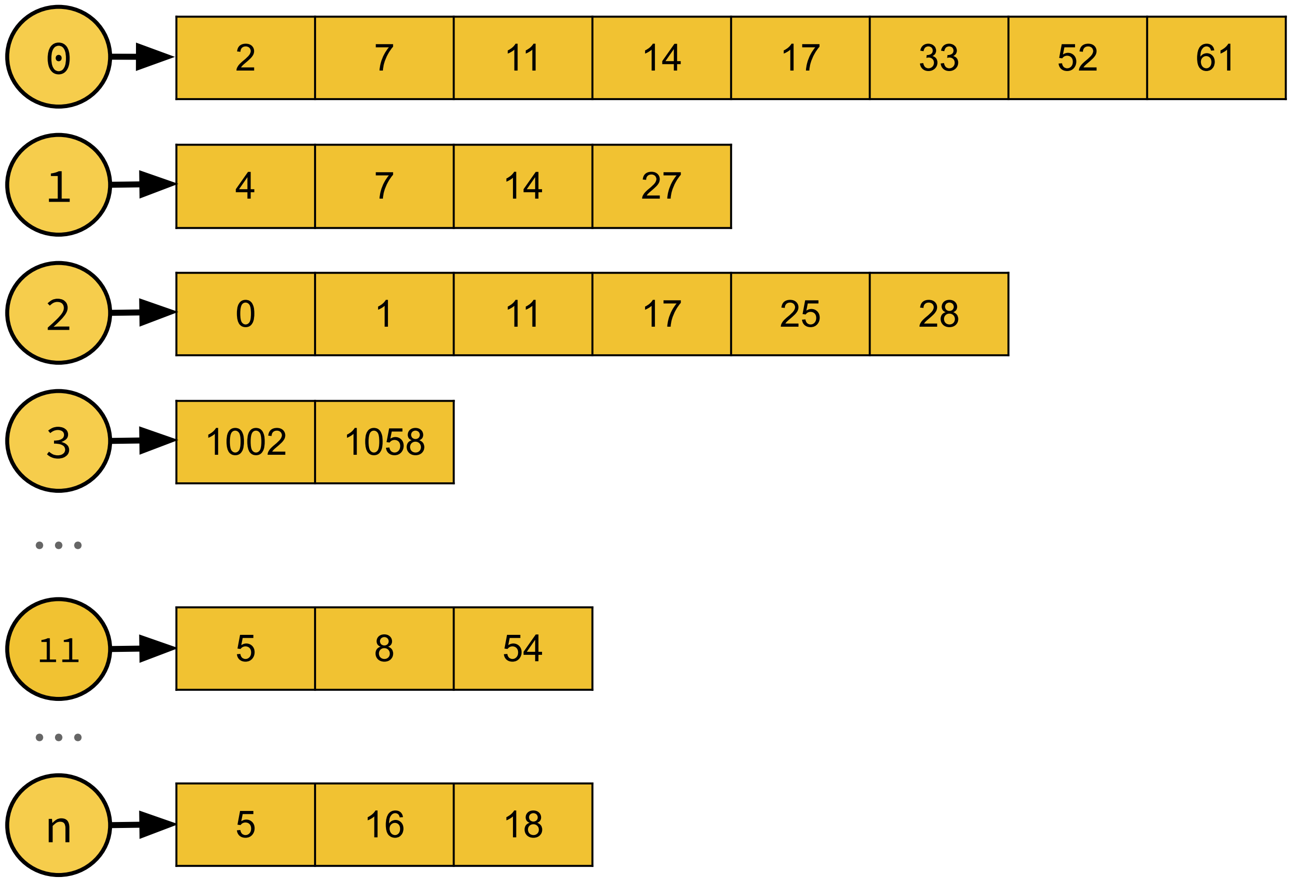
N-Way-Join c

Scan a → b

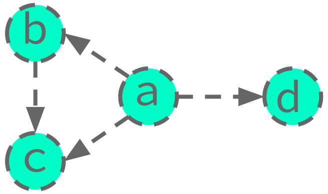
Fw(11) = {5, 8, 54}

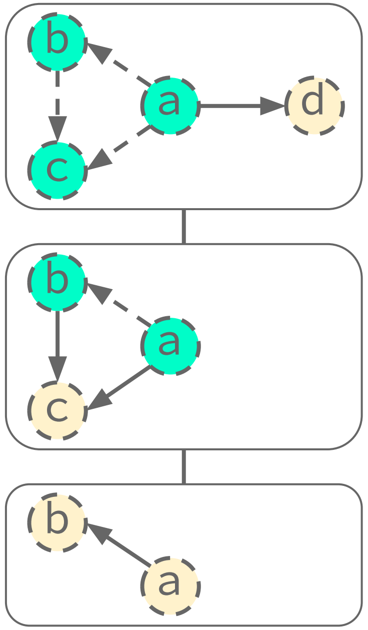
Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, …}



Execution Simulation





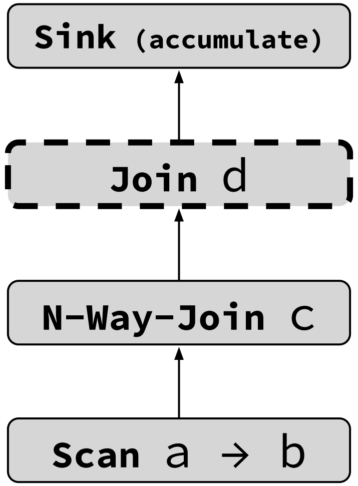
Plan_1[a,b,c,d]

intermediate

a	b	c	d
0	2	11	5

output relation

a	b	c	d
0	2	11	5



Sink (accumulate)

Join d

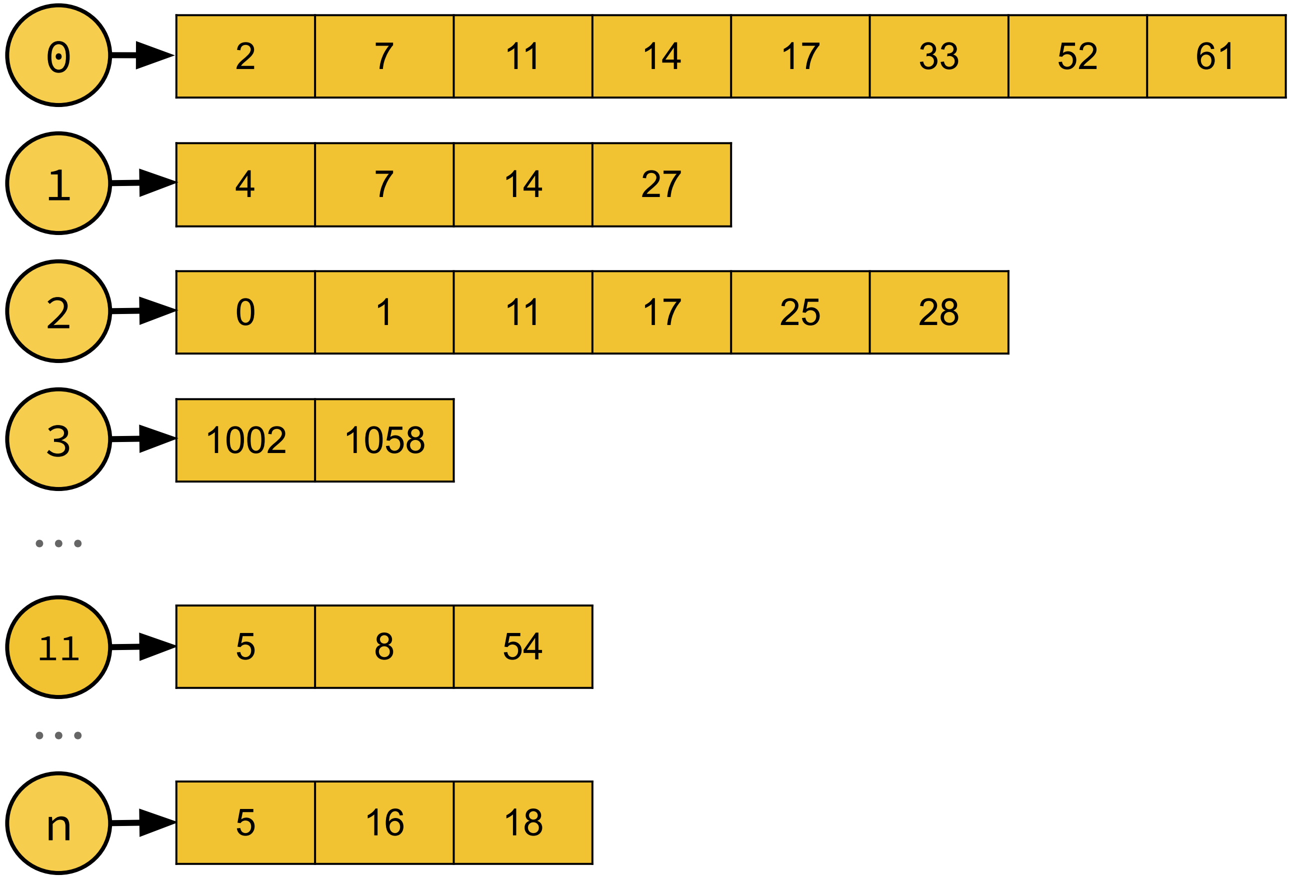
N-Way-Join c

Scan a → b

Fw(11) = {5, 8, 54}

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, …}



0 → 2, 7, 11, 14, 17, 33, 52, 61

1 → 4, 7, 14, 27

2 → 0, 1, 11, 17, 25, 28

3 → 1002, 1058

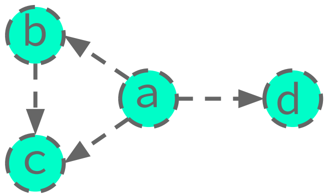
…

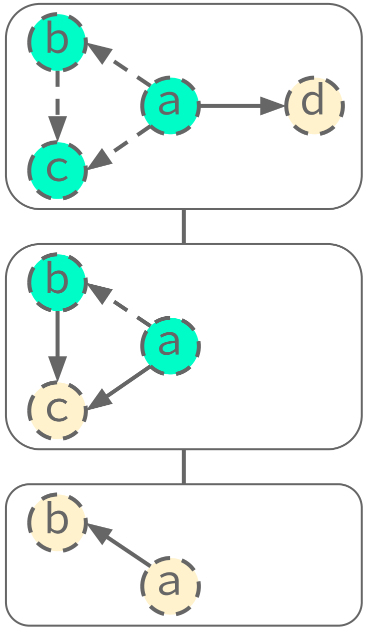
11 → 5, 8, 54

…

n → 5, 16, 18

Execution Simulation





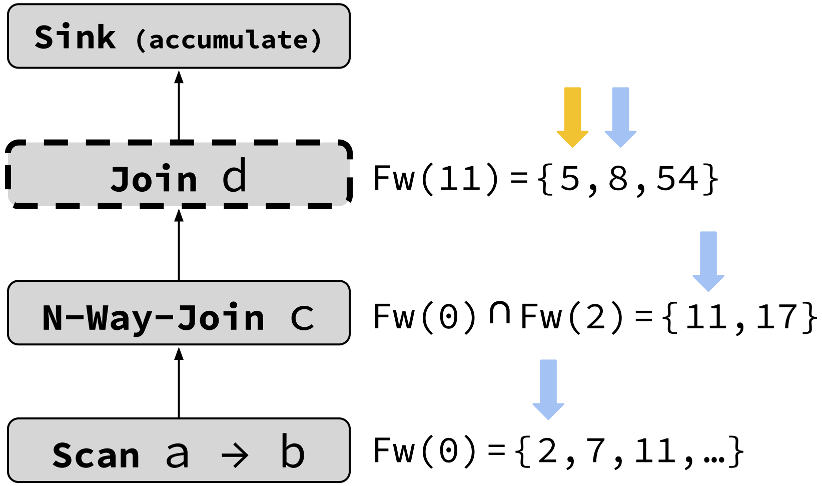
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	5

output relation

a	b	c	d
0	2	11	5



Sink (accumulate)

Join d

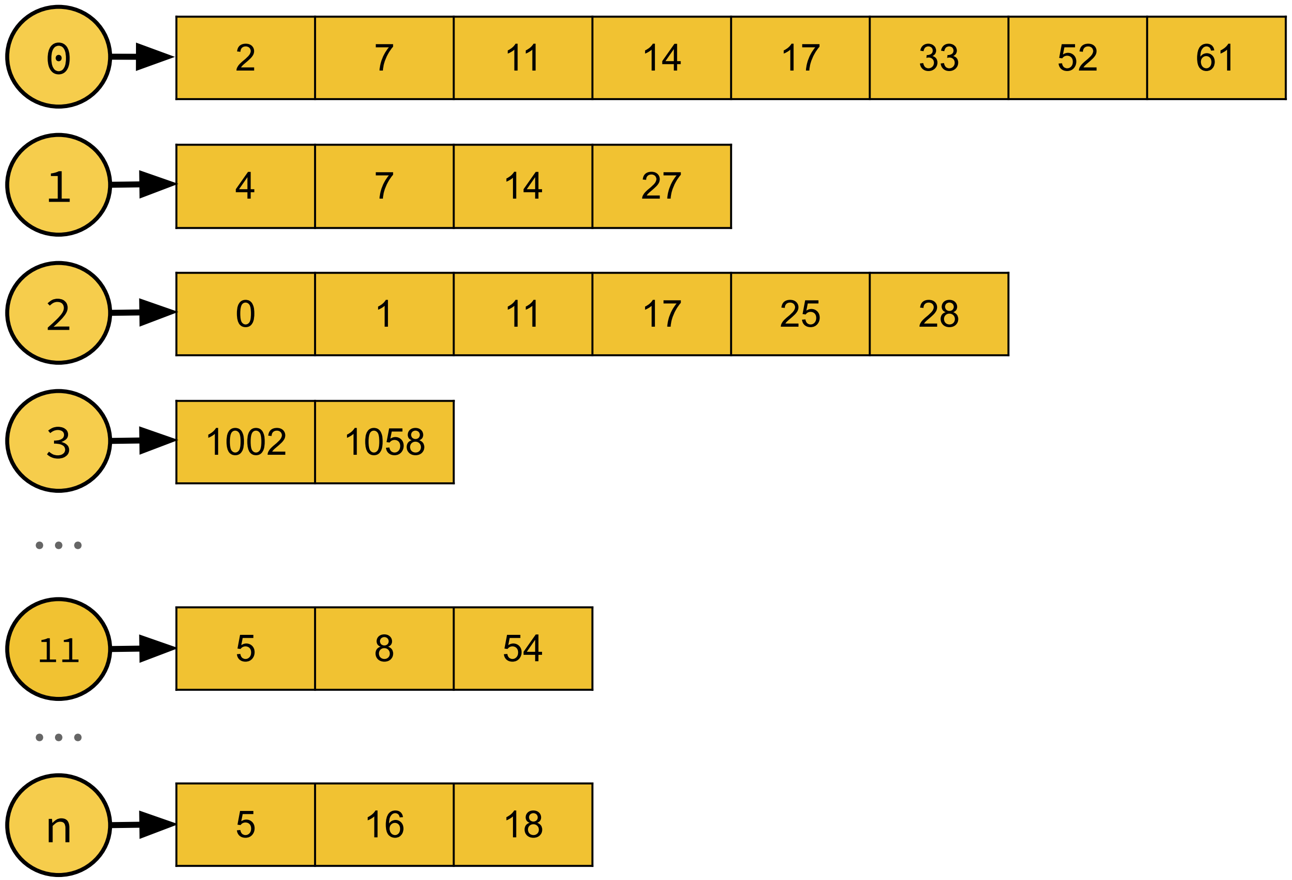
N-Way-Join c

Scan a → b

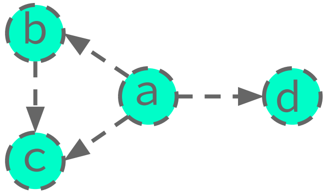
Fw(11)={5,8,54}

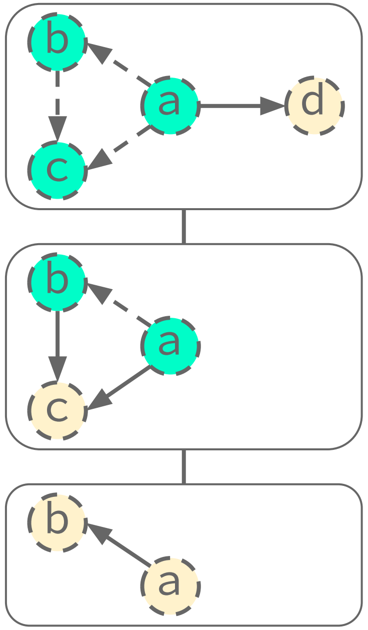
Fw(0)∩Fw(2)={11,17}

Fw(0)={2,7,11,…}



Execution Simulation





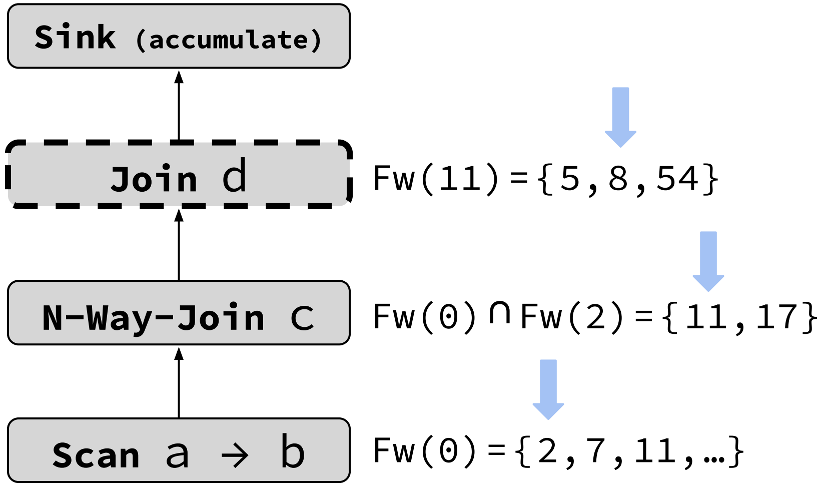
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	8

output relation

a	b	c	d
0	2	11	5

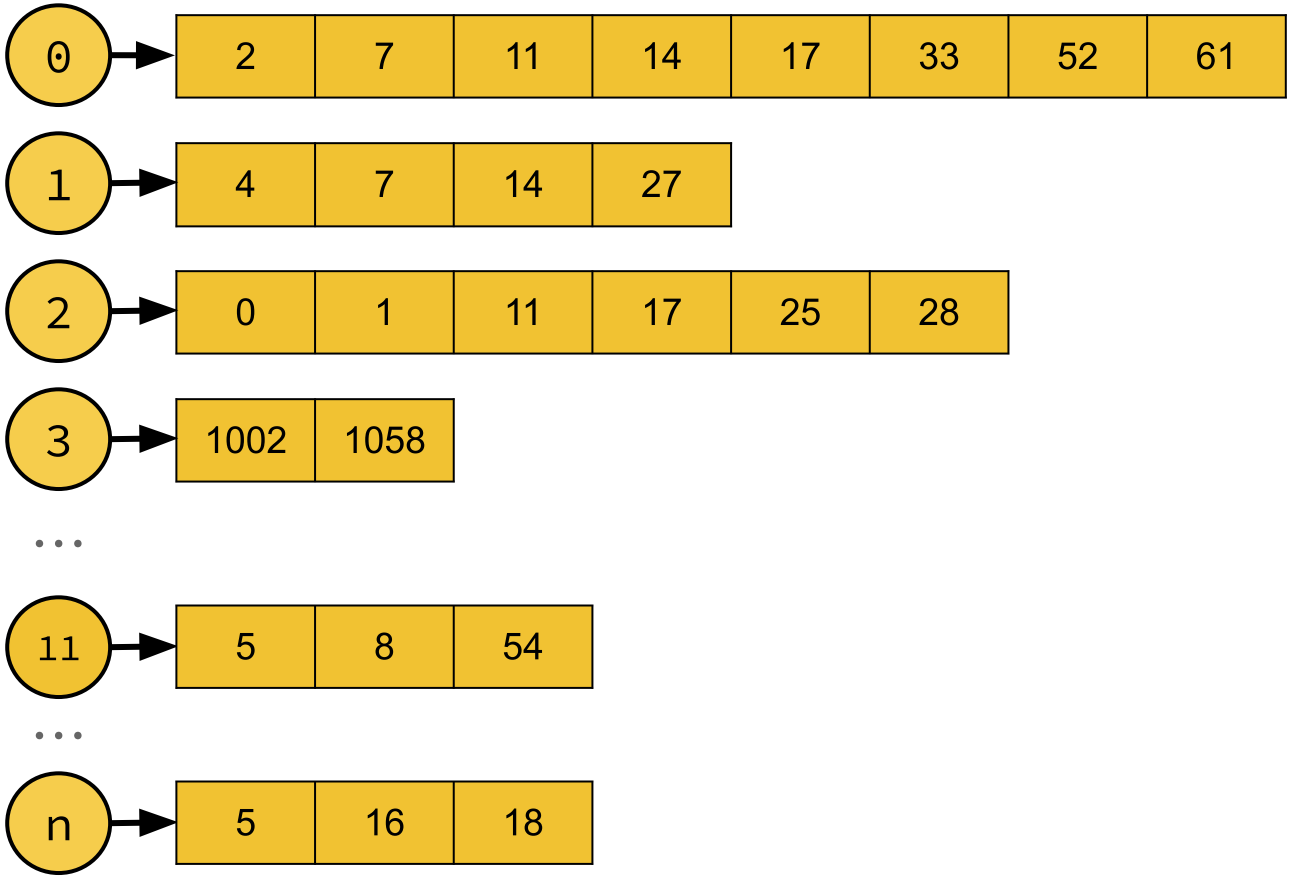


Sink (accumulate)

Join d Fw(11) = {5, 8, 54}

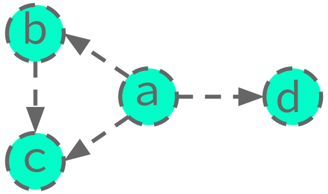
N-Way-Join c Fw(0) ∩ Fw(2) = {11, 17}

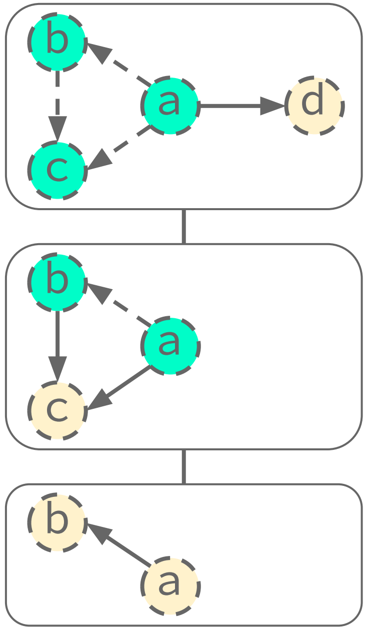
Scan a → b Fw(0) = {2, 7, 11, …}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
...								
11	5	8	54					
...								
n	5	16	18					

Execution Simulation





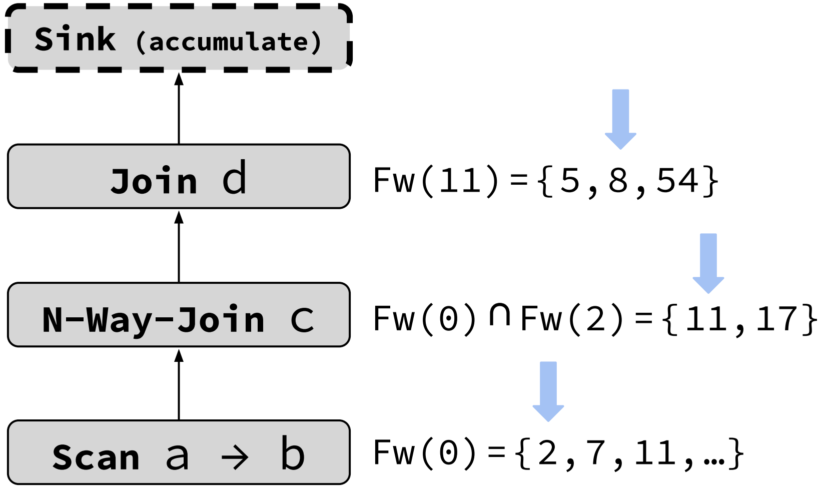
$\text{Plan}_1[a,b,c,d]$

intermediate

a	b	c	d
0	2	11	8

output relation

a	b	c	d
0	2	11	5



Sink (accumulate)

Join d

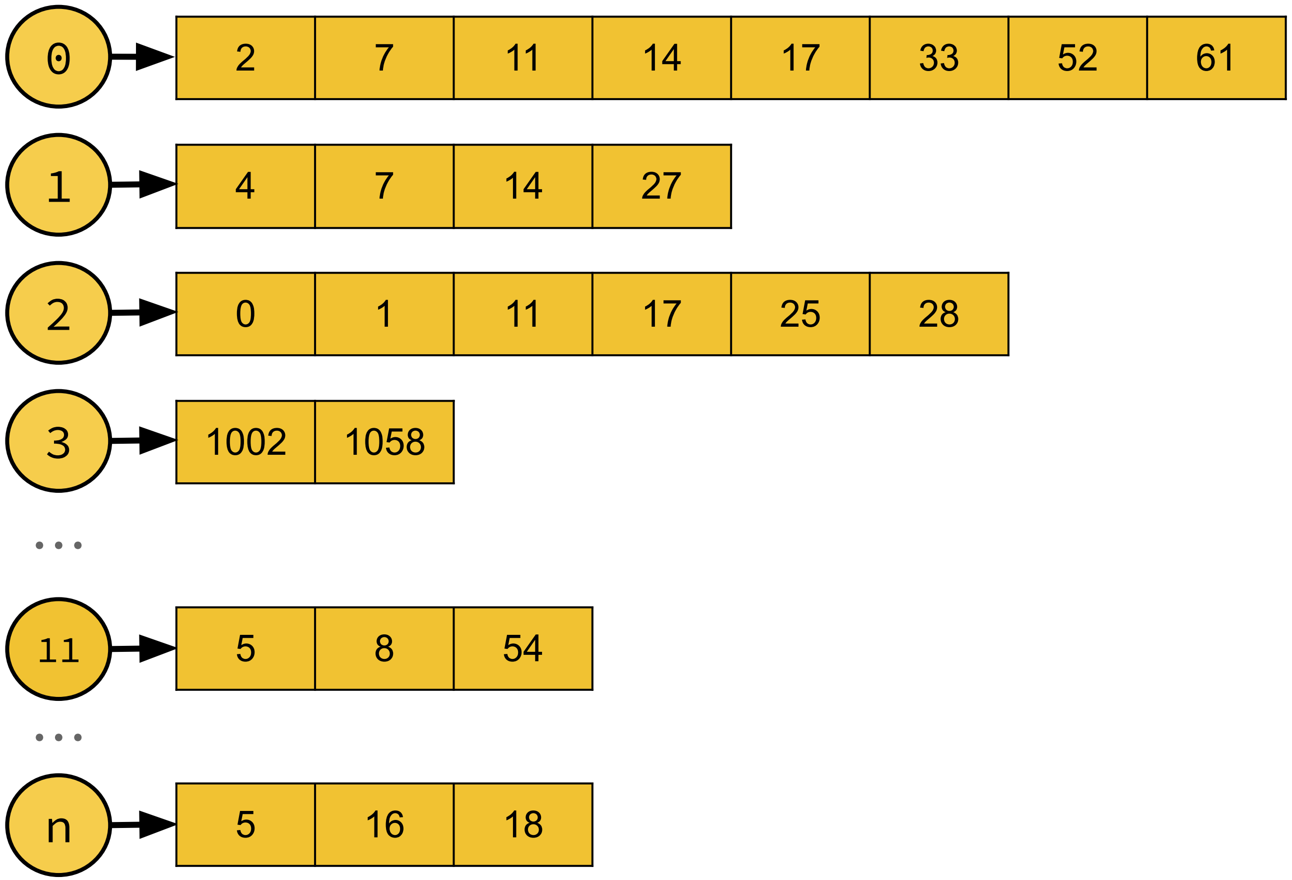
Fw(11) = {5, 8, 54}

N-Way-Join c

Fw(0) ∩ Fw(2) = {11, 17}

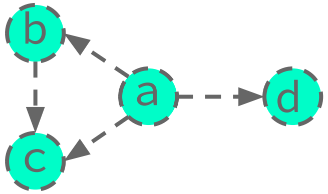
Scan a → b

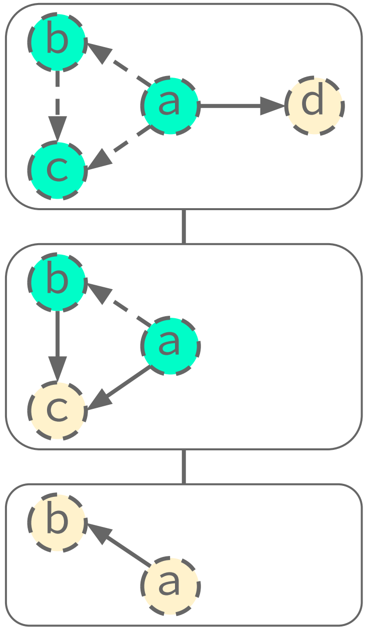
Fw(0) = {2, 7, 11, …}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





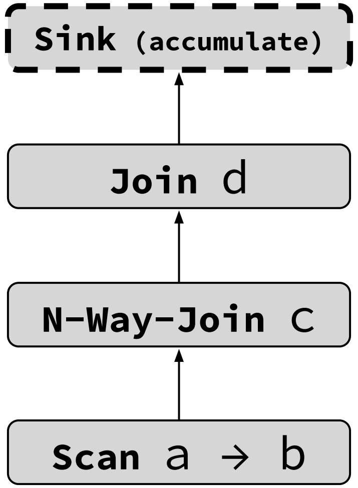
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	8

output relation

a	b	c	d
0	2	11	5
0	2	11	8



Sink (accumulate)

Join d

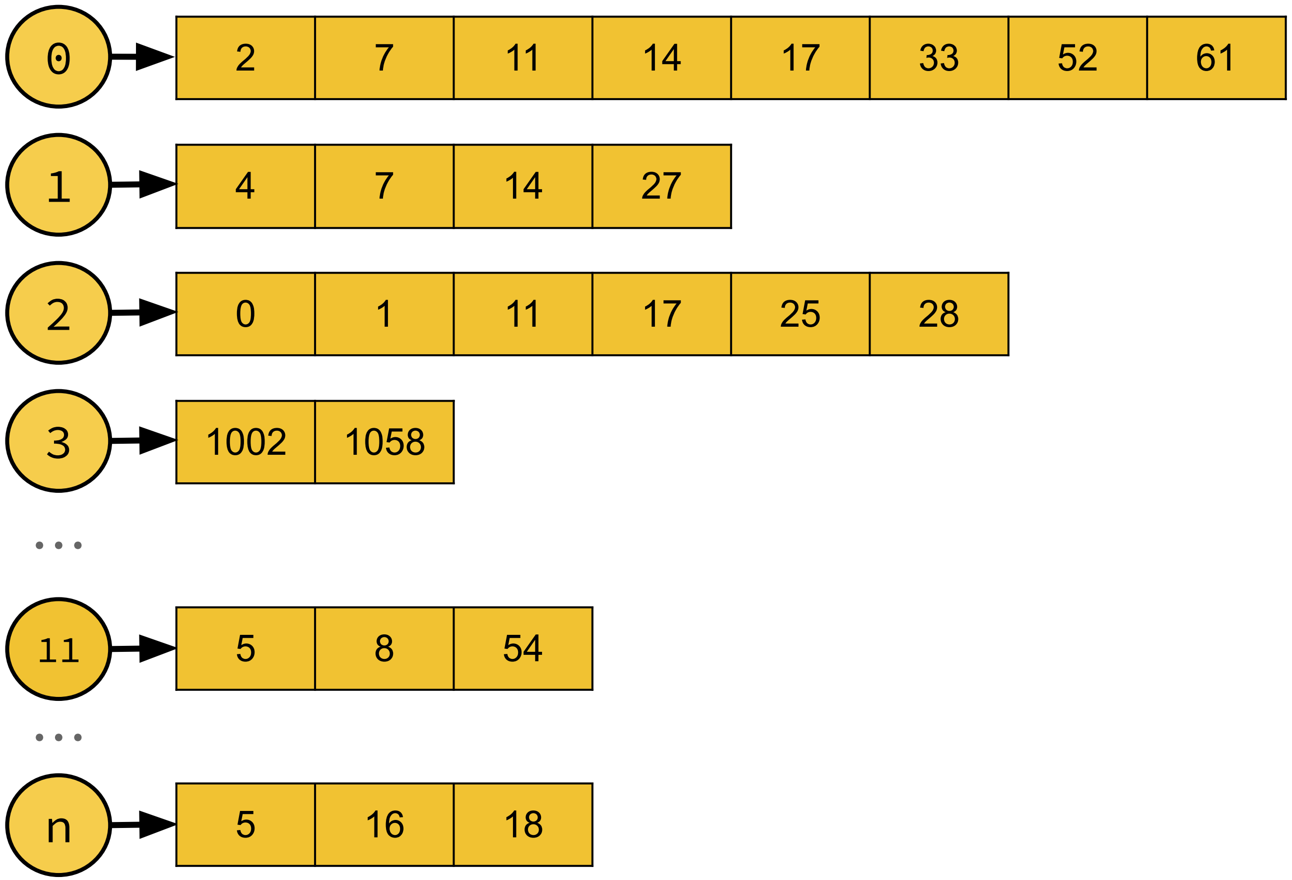
N-Way-Join c

Scan a → b

Fw(11) = {5, 8, 54}

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, …}



0 → 2, 7, 11, 14, 17, 33, 52, 61

1 → 4, 7, 14, 27

2 → 0, 1, 11, 17, 25, 28

3 → 1002, 1058

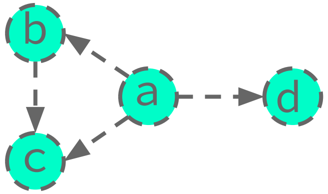
...

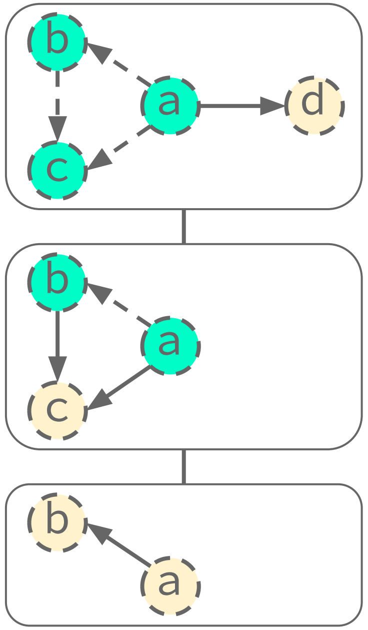
11 → 5, 8, 54

...

n → 5, 16, 18

Execution Simulation





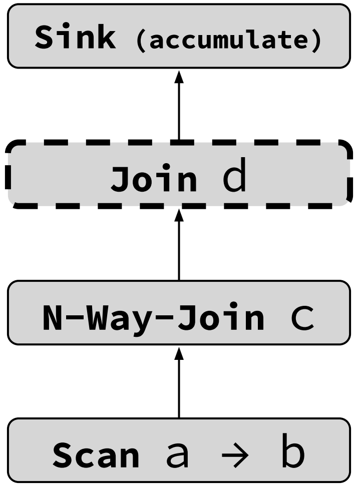
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	8

output relation

a	b	c	d
0	2	11	5
0	2	11	8



Sink (accumulate)

Join d

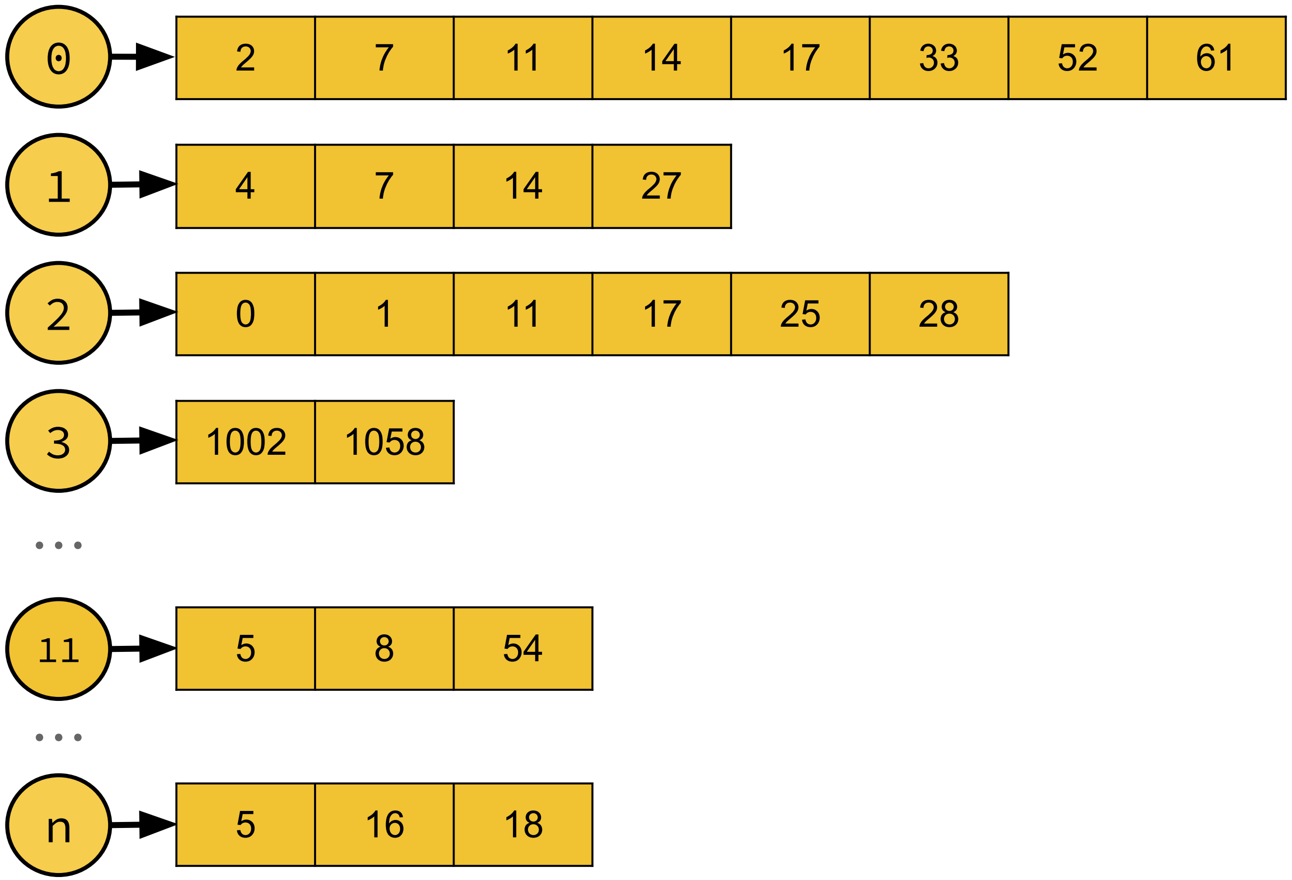
N-Way-Join c

Scan a → b

Fw(11)={5,8,54}

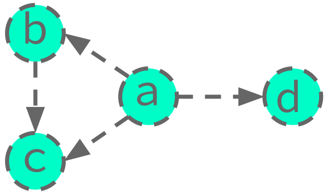
Fw(0) ∩ Fw(2)={11,17}

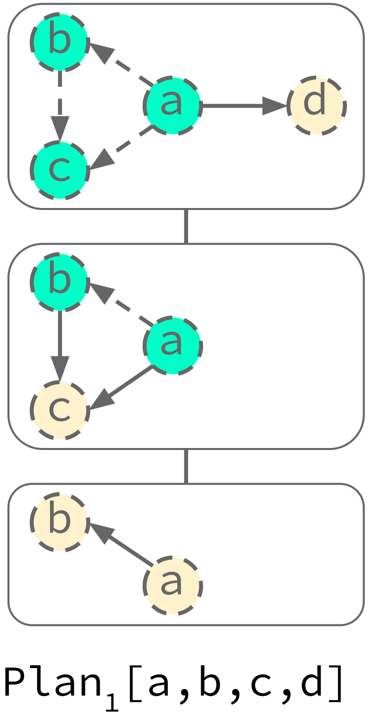
Fw(0)={2,7,11,…}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





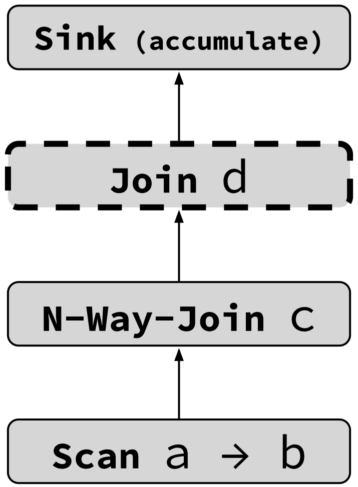
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	8

output relation

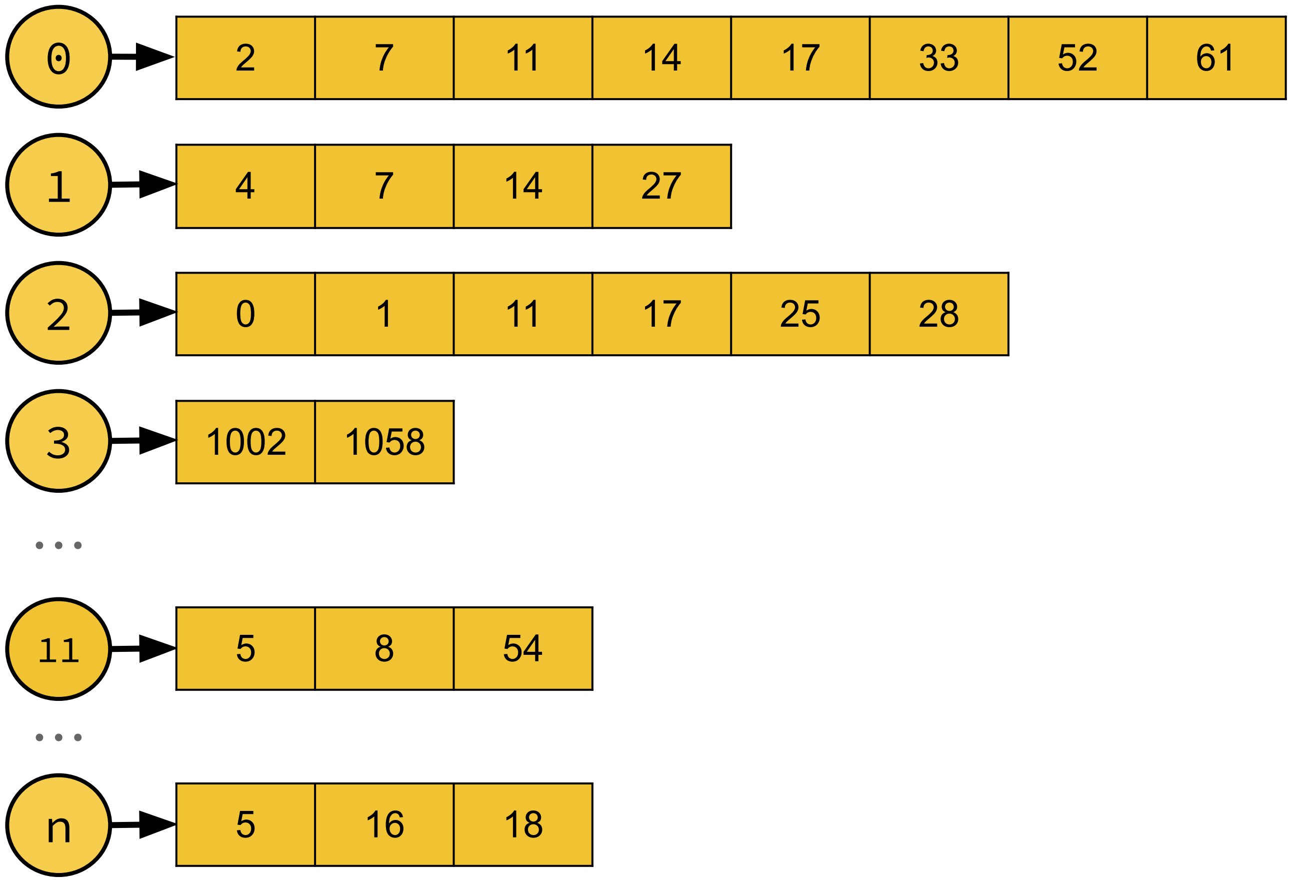
a	b	c	d
0	2	11	5
0	2	11	8



Fw(11)={5,8,54}

Fw(0)∩Fw(2)={11,17}

Fw(0)={2,7,11,…}



0 → 2, 7, 11, 14, 17, 33, 52, 61

1 → 4, 7, 14, 27

2 → 0, 1, 11, 17, 25, 28

3 → 1002, 1058

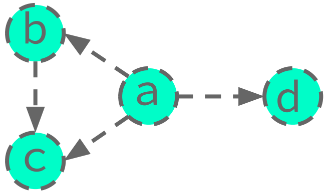
…

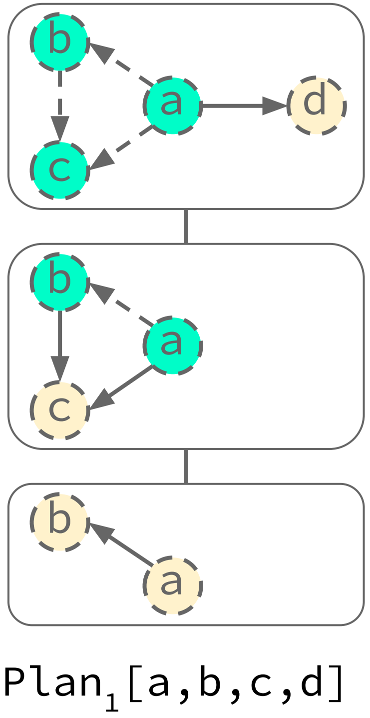
11 → 5, 8, 54

…

n → 5, 16, 18

Execution Simulation





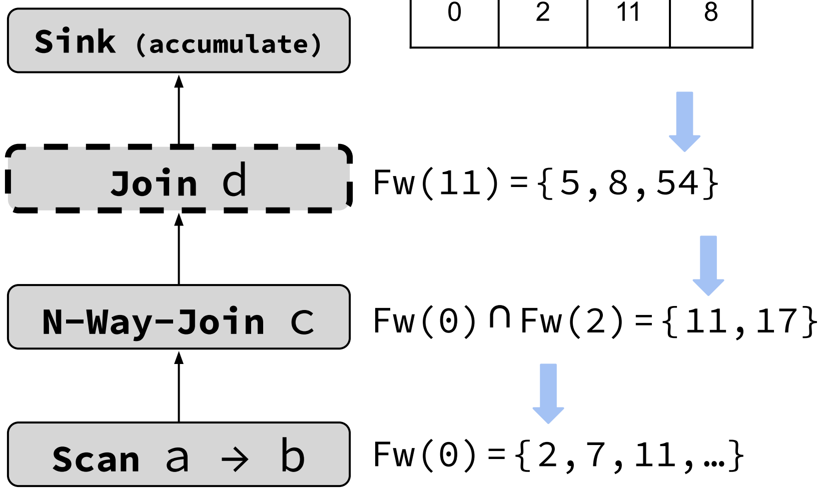
$Plan_1[a,b,c,d]$

intermediate

a	b	c	d
0	2	11	54

output relation

a	b	c	d
0	2	11	5
0	2	11	8



Sink (accumulate)

Join d

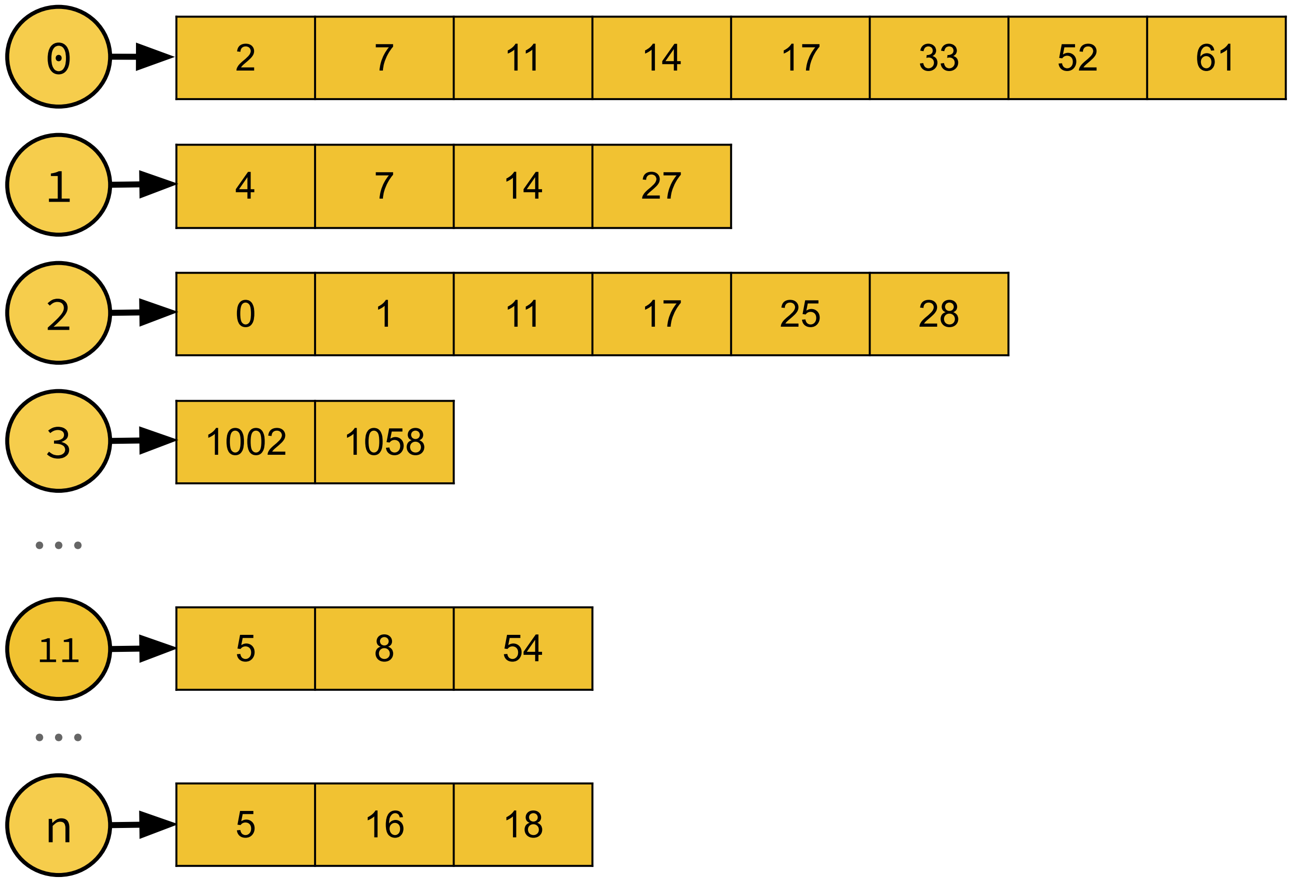
N-Way-Join c

Scan a → b

Fw(11) = {5, 8, 54}

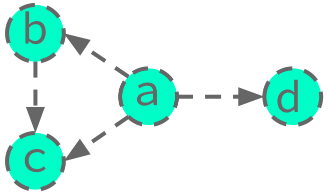
Fw(0) ∩ Fw(2) = {11, 17}

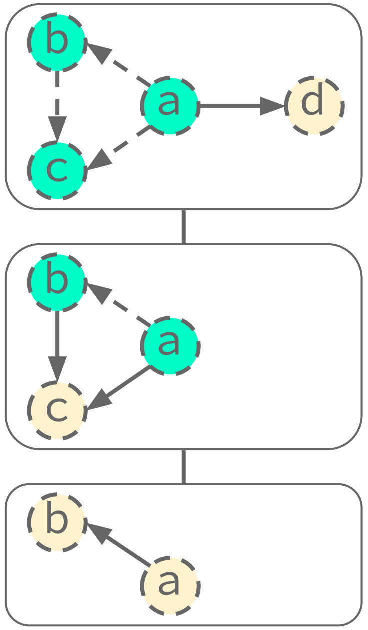
Fw(0) = {2, 7, 11, …}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





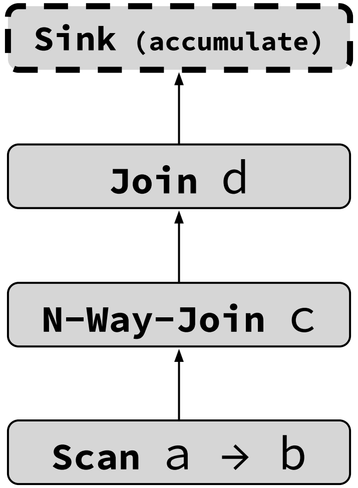
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	54

output relation

a	b	c	d
0	2	11	5
0	2	11	8
0	2	11	54



Sink (accumulate)

Join d

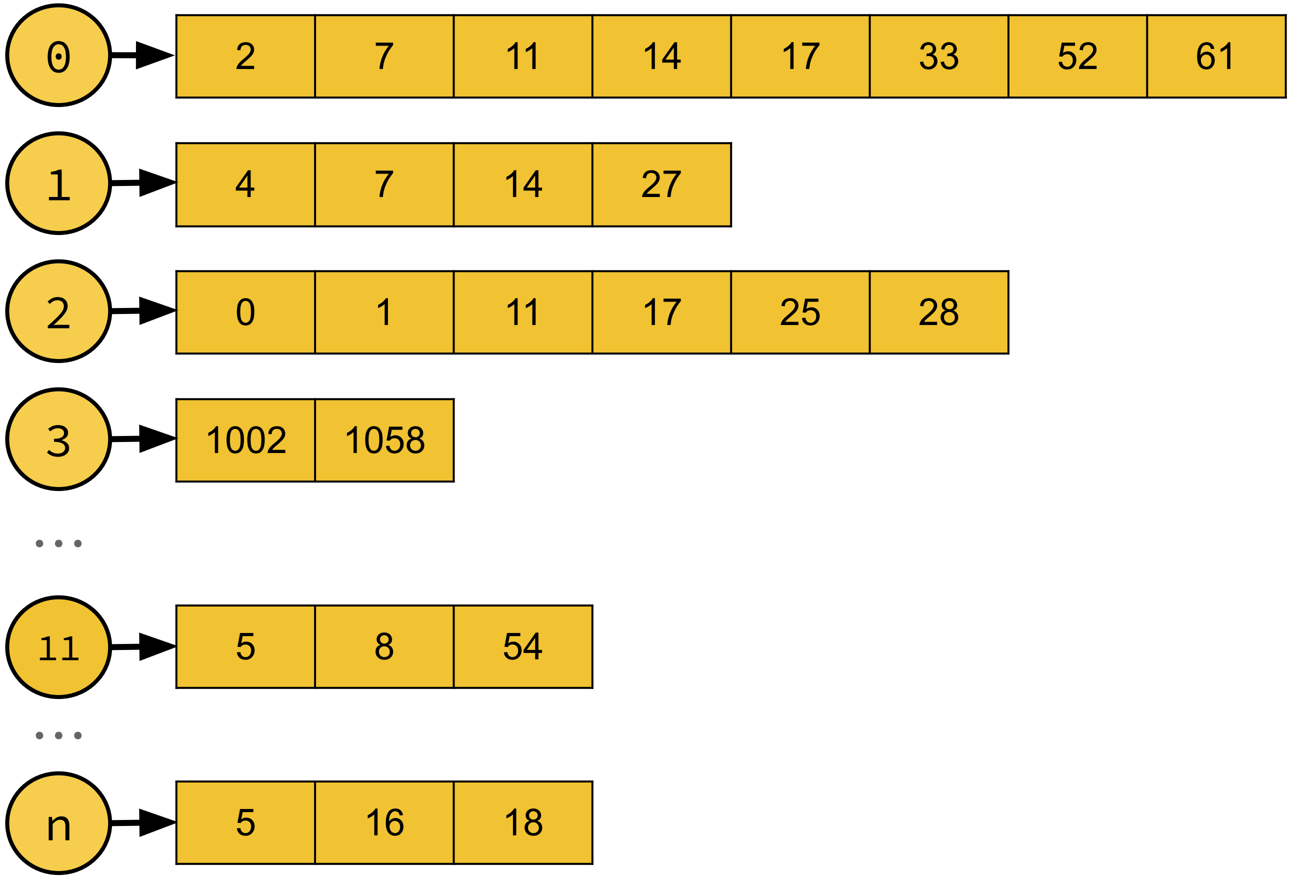
N-Way-Join c

Scan a → b

Fw(11) = {5, 8, 54}

Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, …}



0 → 2 | 7 | 11 | 14 | 17 | 33 | 52 | 61

1 → 4 | 7 | 14 | 27

2 → 0 | 1 | 11 | 17 | 25 | 28

3 → 1002 | 1058

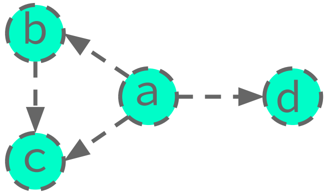
…

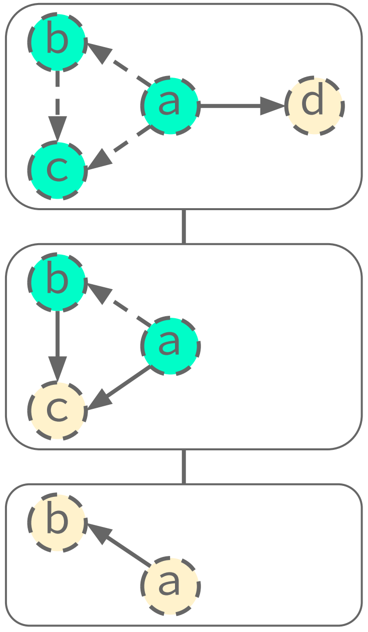
11 → 5 | 8 | 54

…

n → 5 | 16 | 18

Execution Simulation





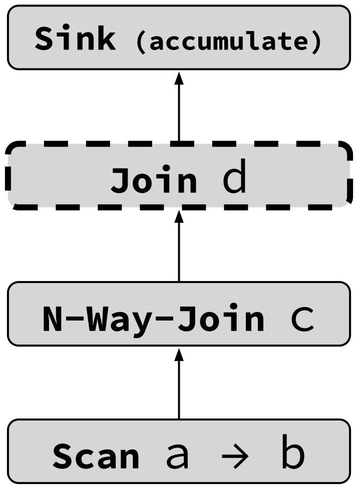
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	54

output relation

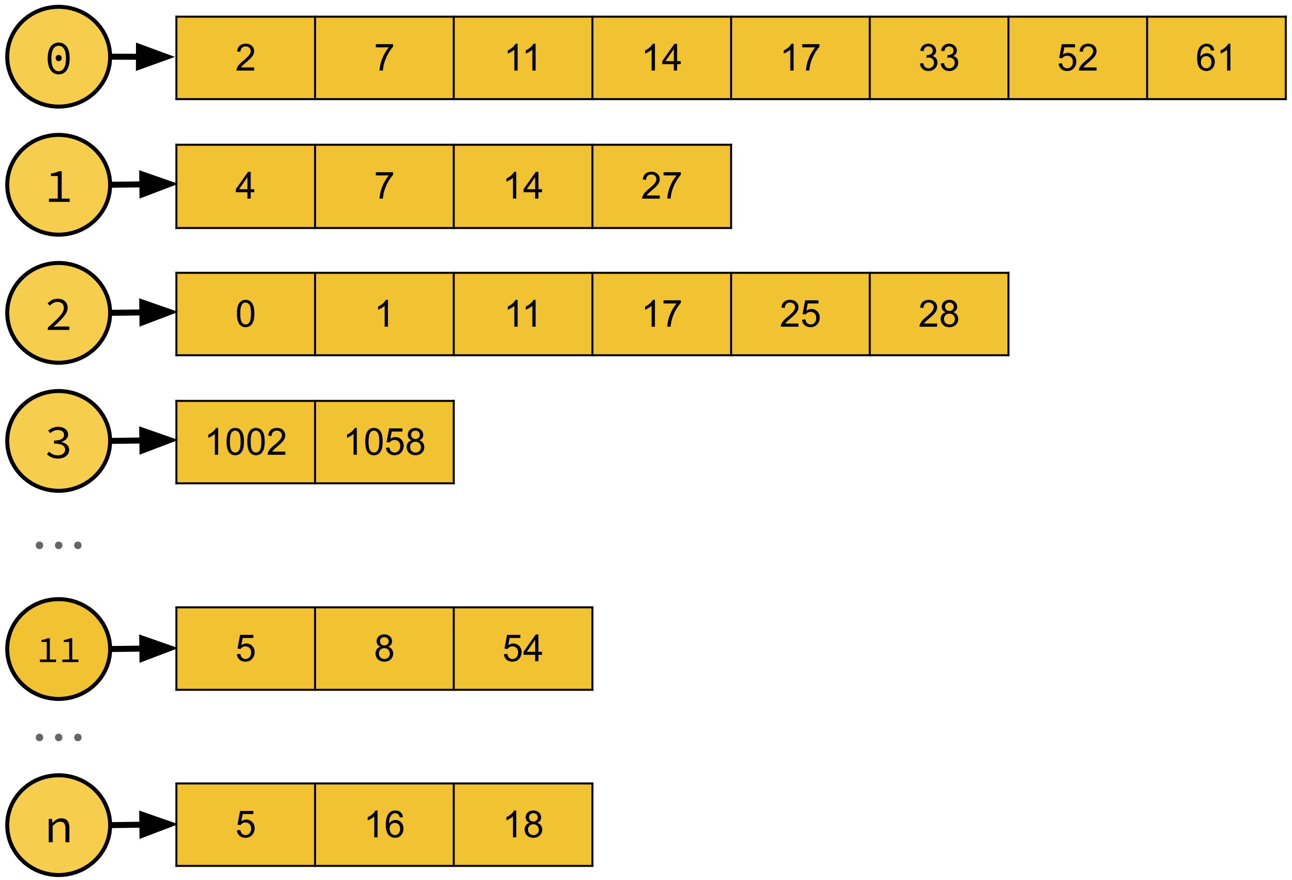
a	b	c	d
0	2	11	5
0	2	11	8
0	2	11	54



Fw(11)={5,8,54}

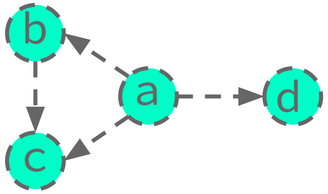
Fw(0)∩Fw(2)={11,17}

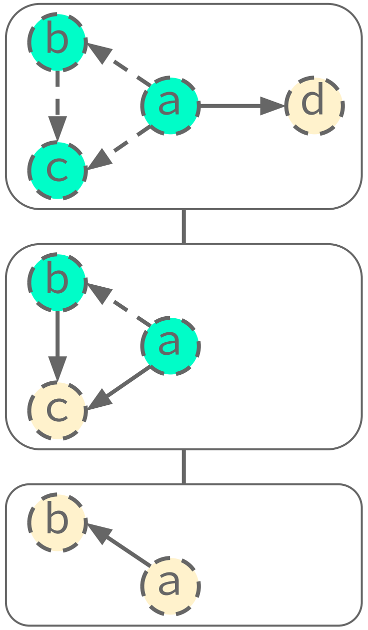
Fw(0)={2,7,11,…}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





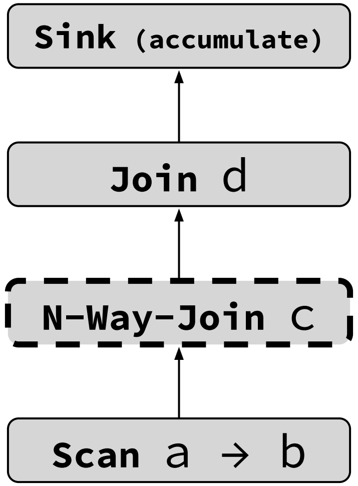
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	54

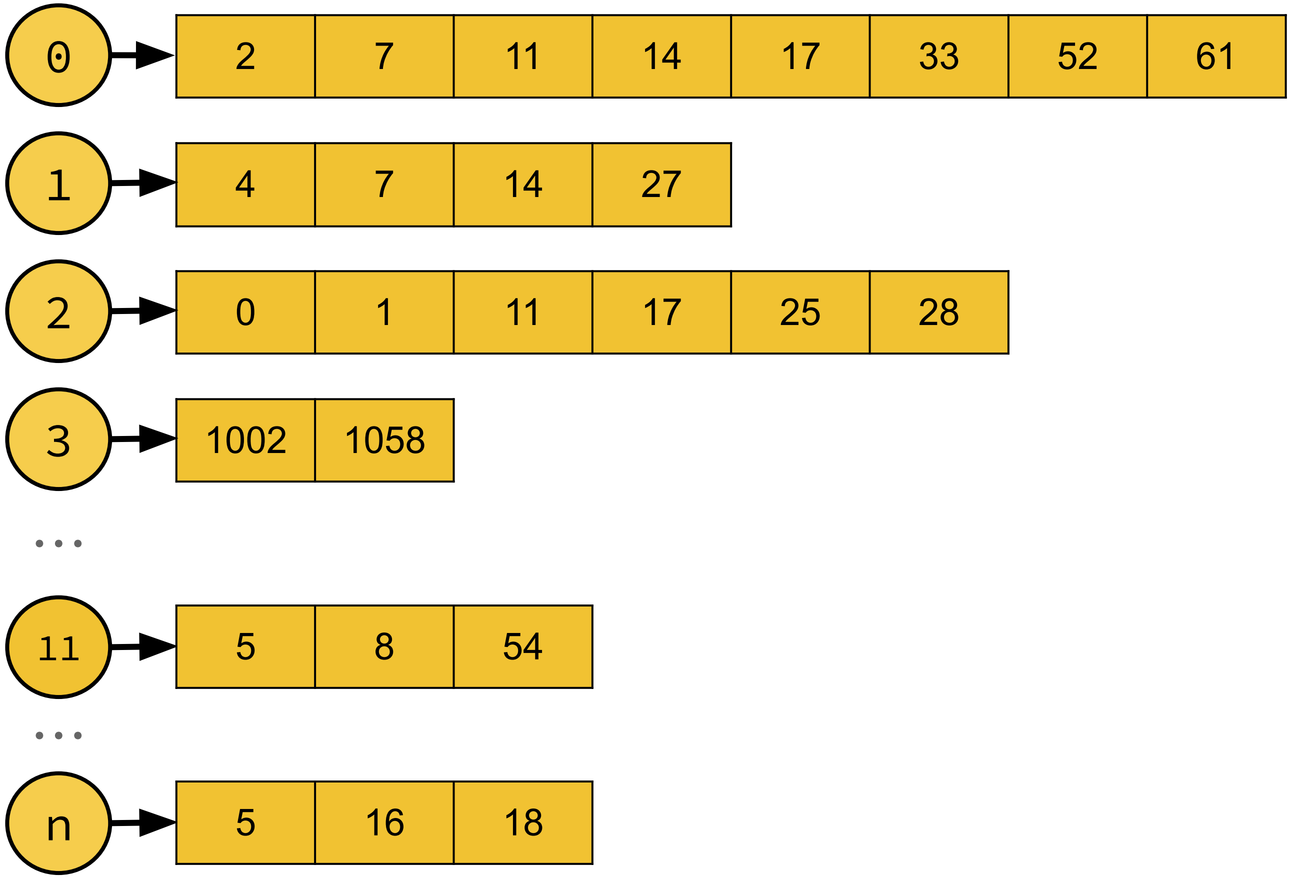
output relation

a	b	c	d
0	2	11	5
0	2	11	8
0	2	11	54

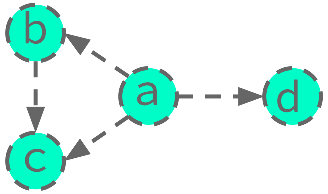


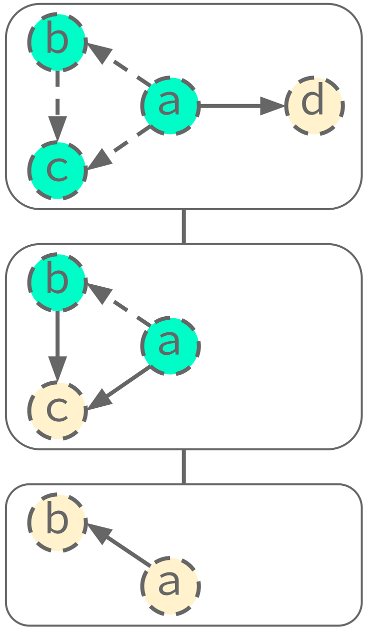
Fw(0) ∩ Fw(2) = {11, 17}

Fw(0) = {2, 7, 11, …}



Execution Simulation





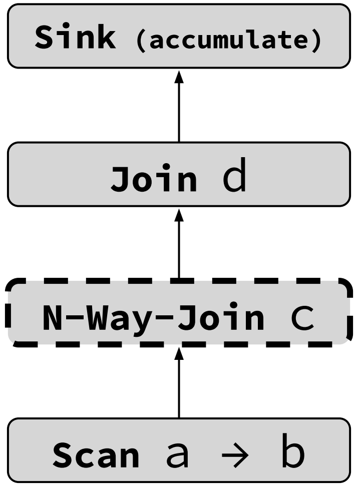
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	11	54

output relation

a	b	c	d
0	2	11	5
0	2	11	8
0	2	11	54



Sink (accumulate)

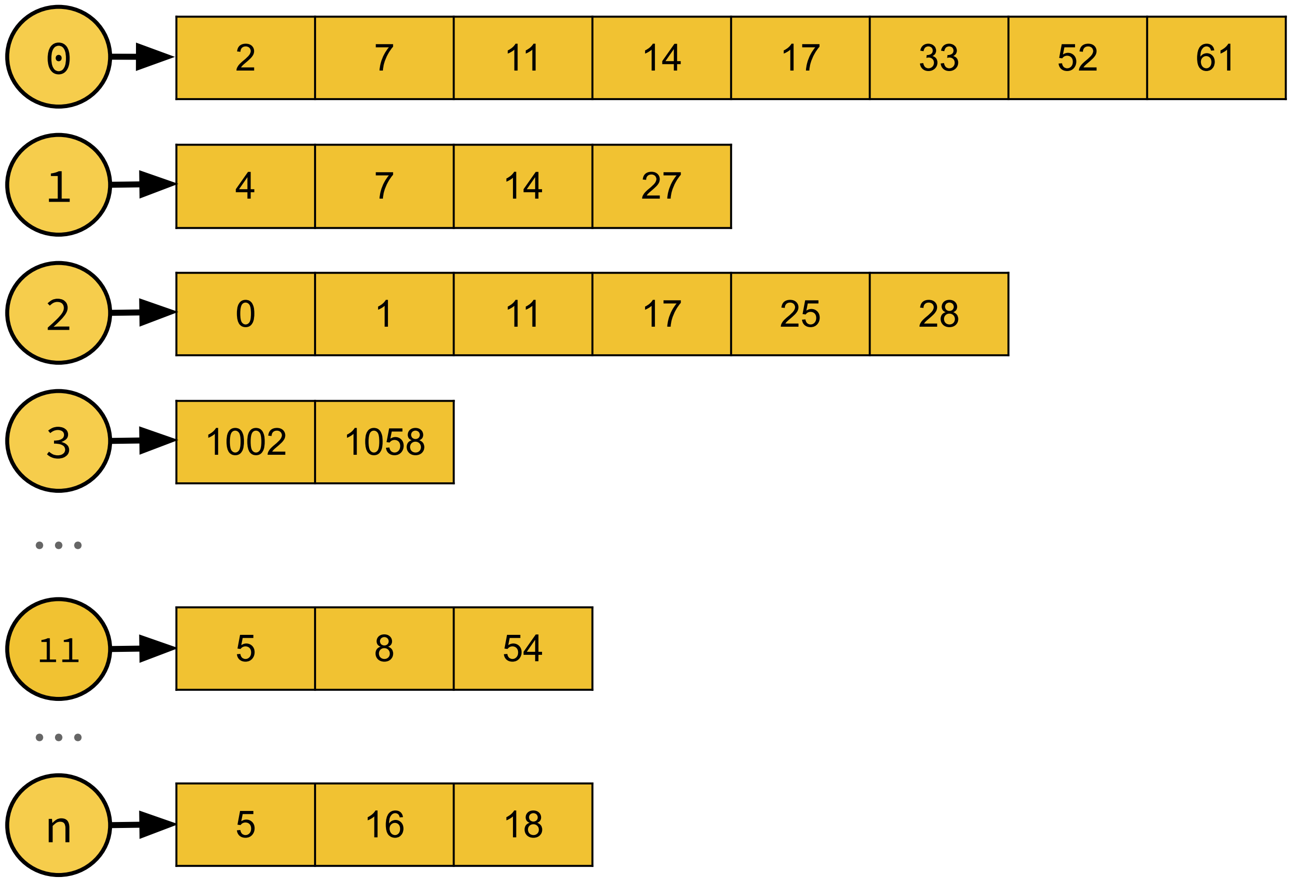
Join d

N-Way-Join c

Scan a → b

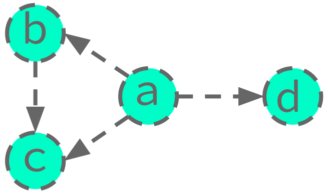
Fw(0) ∩ Fw(2) = {11, 17}

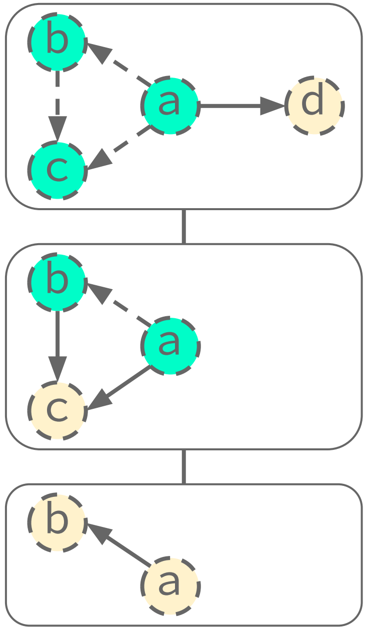
Fw(0) = {2, 7, 11, …}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Execution Simulation





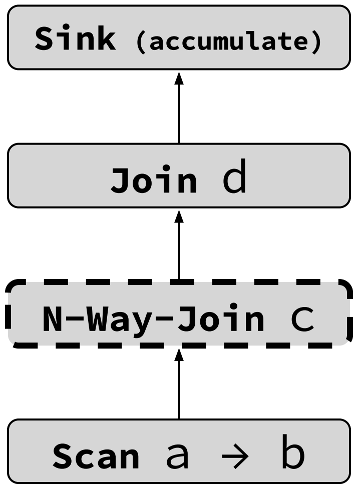
Plan$_1$[a,b,c,d]

intermediate

a	b	c	d
0	2	17	54

output relation

a	b	c	d
0	2	11	5
0	2	11	8
0	2	11	54



Sink (accumulate)

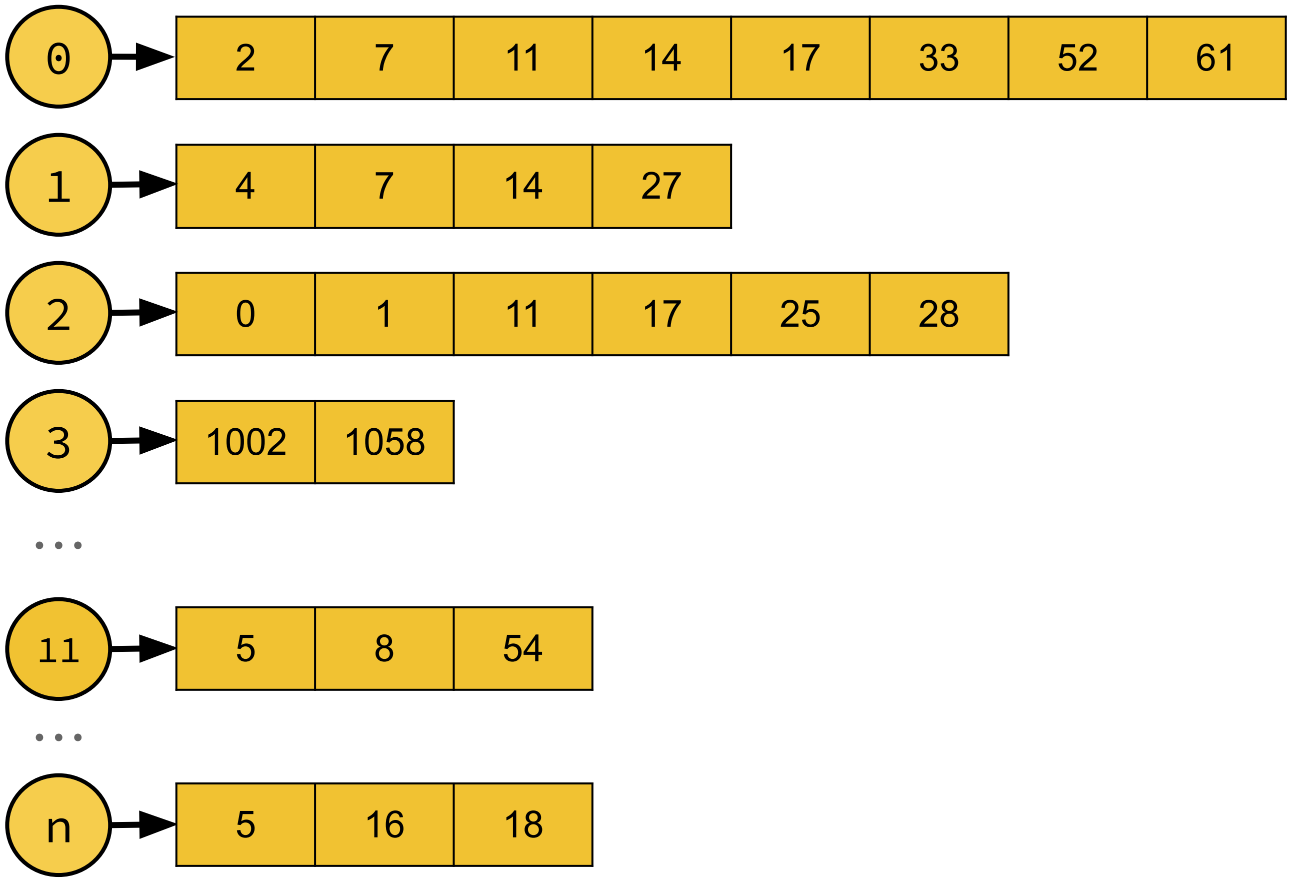
Join d

N-Way-Join c

Scan a → b

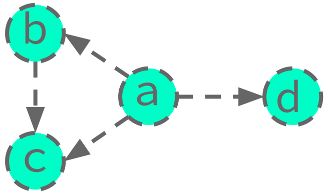
Fw(0) ∩ Fw(2) = {11, 17}

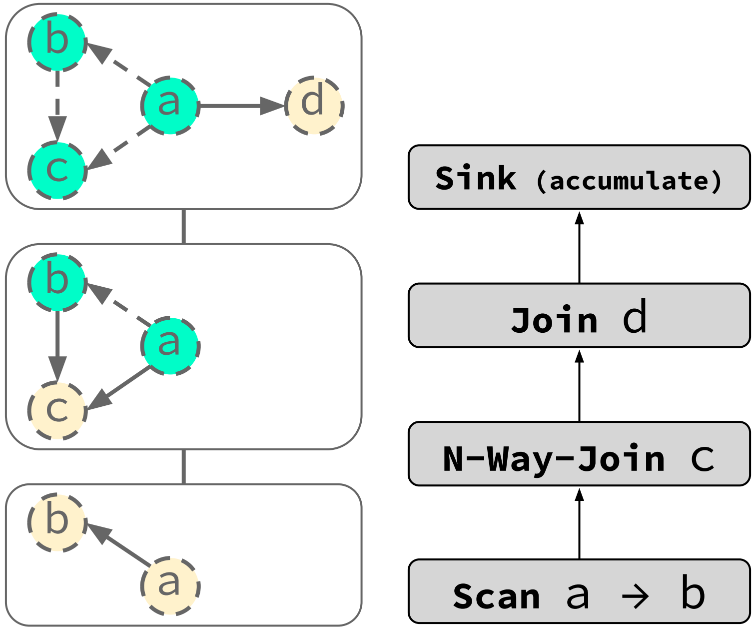
Fw(0) = {2, 7, 11, …}



0	2	7	11	14	17	33	52	61
1	4	7	14	27				
2	0	1	11	17	25	28		
3	1002	1058						
…								
11	5	8	54					
…								
n	5	16	18					

Multiple possible Query Vertex Orderings (QVOs)





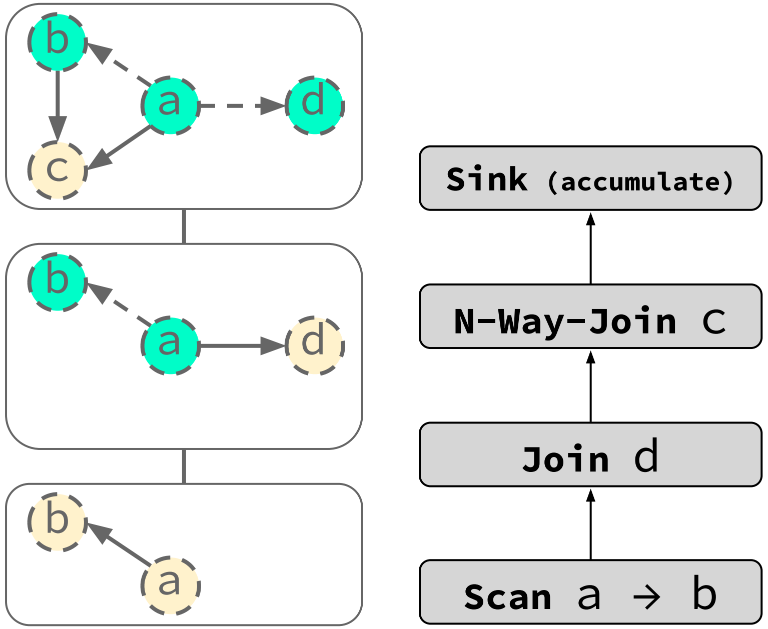
Sink (accumulate)

Join d

N-Way-Join c

Scan a → b

Plan$_1$[a,b,c,d]



Sink (accumulate)

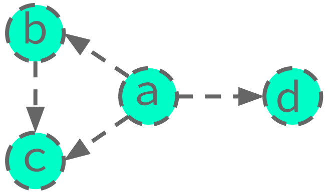
N-Way-Join c

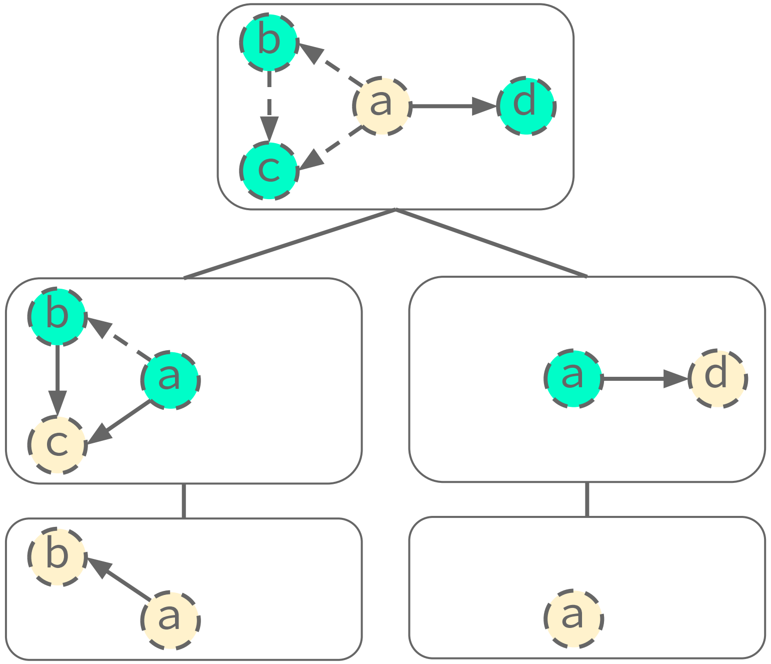
Join d

Scan a → b

Plan$_2$[a,b,d,c]

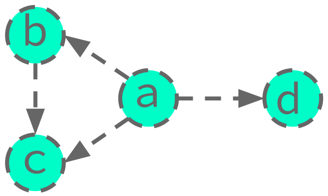
Decomposition using Binary Joins

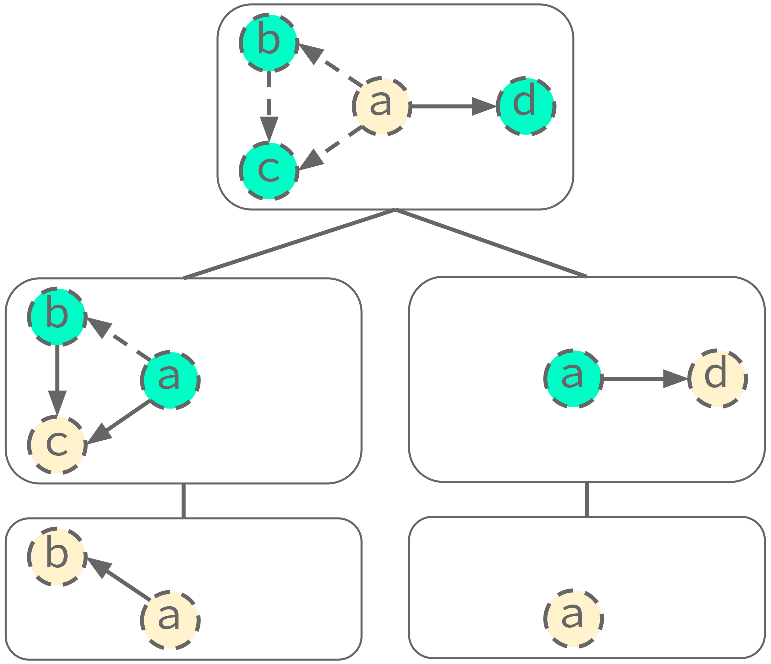




Plan$_1$[a,b,c]   Plan$_2$[a,b,d]

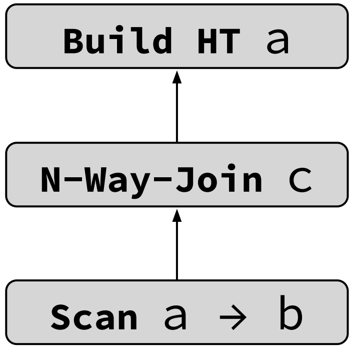
Decomposition using Binary Joins





Plan$_1$[a,b,c]

Plan$_2$[a,b,d]

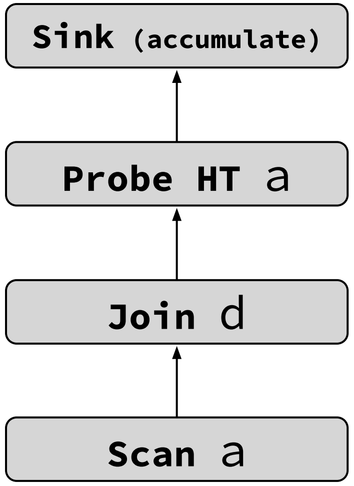


Build HT a

N-Way-Join c

Scan a → b

pipeline 0



Sink (accumulate)

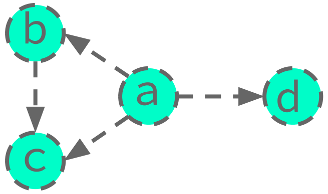
Probe HT a

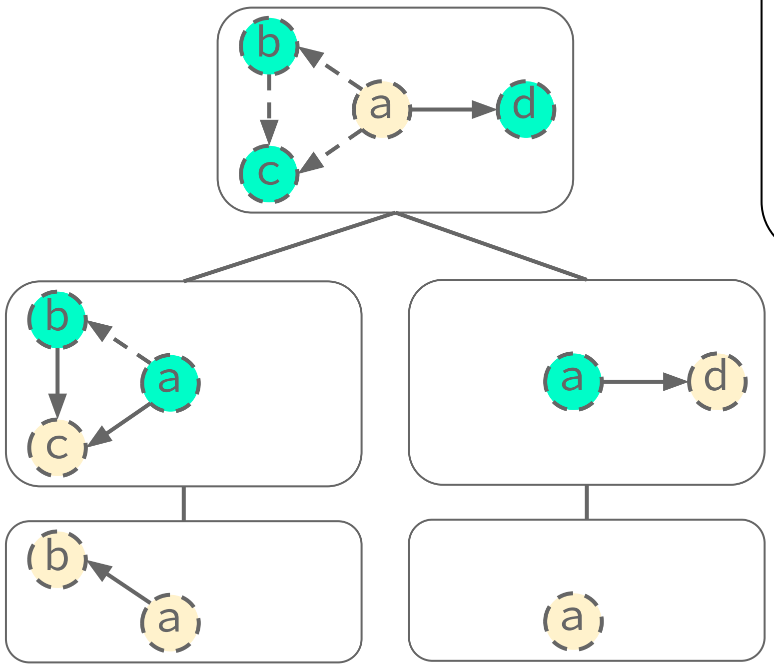
Join d

Scan a

pipeline 1

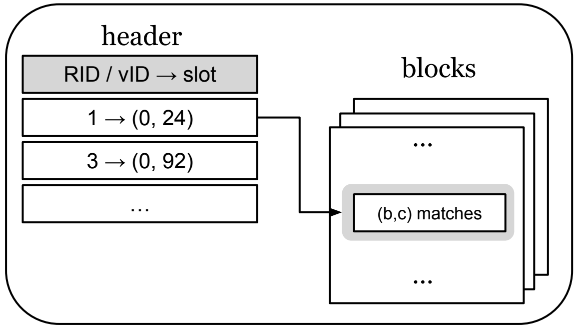
Decomposition using Binary Joins

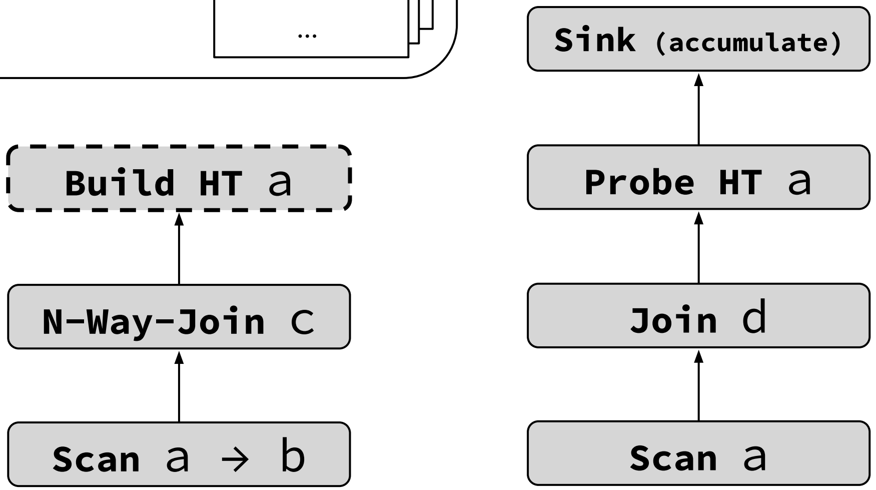




Plan$_1$[a,b,c]   Plan$_2$[a,b,d]

HT on a





pipeline 0   pipeline 1

Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case Optimal Joins (WCOJs)**

   Handling Intermediate Size Growth for Cyclic Joins

   2.1. Foundations

   2.2 System Integration Approaches

   i) **Index-based WCOJs (Graphflow & EmptyHeaded)**

   ii) **Hash-based WCOJs (Umbra)**

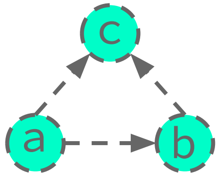
   - **Optimization approaches (cost-based DP, GHD, rule-based)**

3) **Factorized Query Processing**

How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have 2 different effects on runtime.

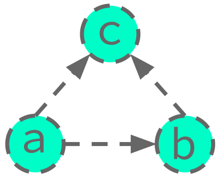
1. Number of intermediate results.
2. Direction of adjacency lists intersected.

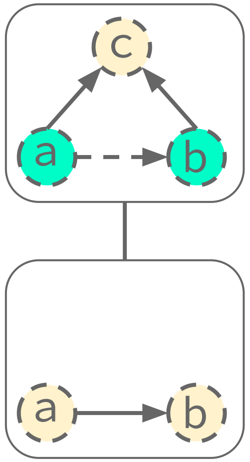


How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have 2 different effects on runtime.

1. Number of intermediate results.
2. Direction of adjacency lists intersected.





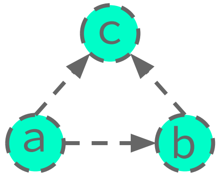
Plan$_1$[a,b,c]

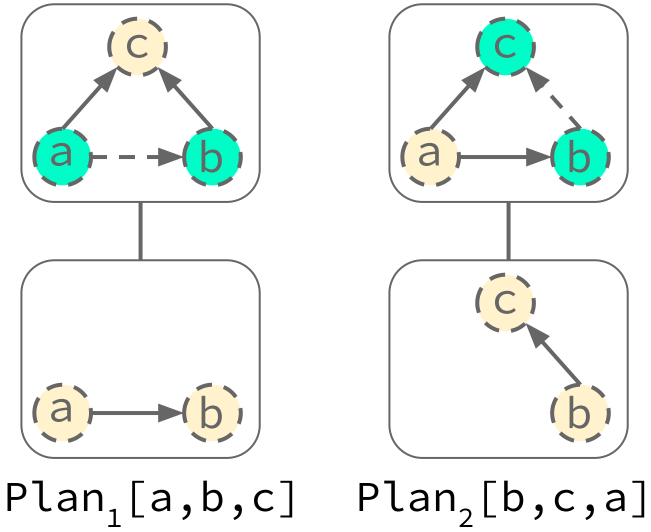
How to Pick Good Query Vertex Ordering?

Query Vertex Orderings have 2 different effects on runtime.

(1) Number of intermediate results.

(2) Direction of adjacency lists intersected.



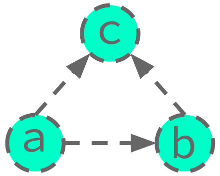


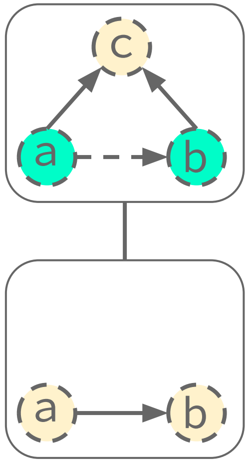
Plan$_1$[a,b,c] Plan$_2$[b,c,a]

How to Pick Good Query Vertex Ordering?

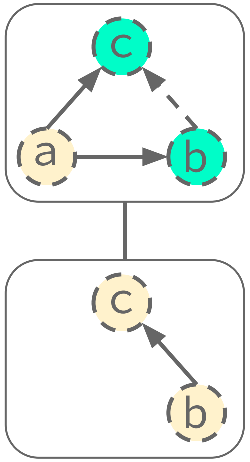
Query Vertex Orderings have 2 different effects on runtime.

(1) Number of intermediate results.
(2) Direction of adjacency lists intersected.

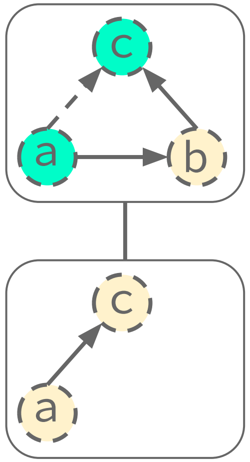




Plan$_1$[a,b,c]



Plan$_2$[b,c,a]

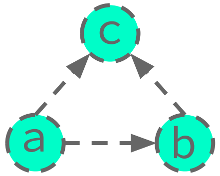


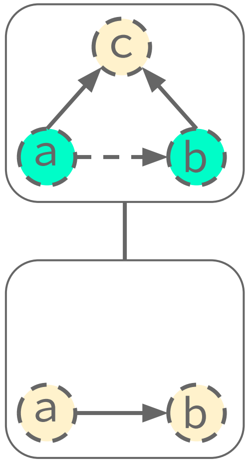
Plan$_3$[a,c,b]

How to Pick Good Query Vertex Ordering?

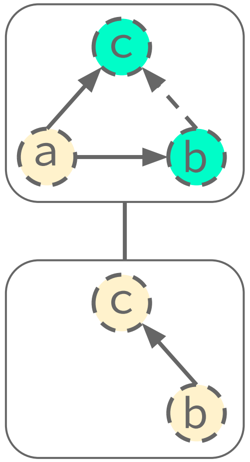
Query Vertex Orderings have 2 different effects on runtime.

(1) Number of intermediate results.
(2) Direction of adjacency lists intersected.

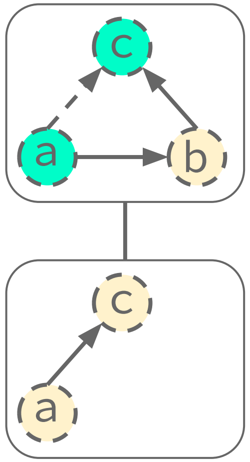




Plan$_1$[a,b,c]



Plan$_2$[b,c,a]

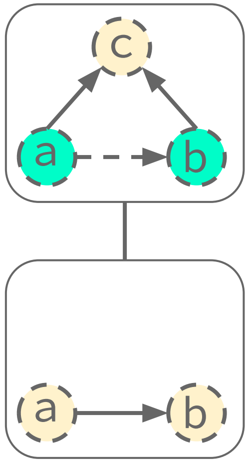


Plan$_3$[a,c,b]

Dataset	Plan	∑ Adj List sizes	Runtime (secs)
Web-BerkStan (V=685K, E=7.6M)	Plan$_1$	0.5B	2.6
	Plan$_2$	55B **(113.8x)**	15.2 **(5.8x)**
	Plan$_3$	55B **(114.0x)**	31.6 **(12.2x)**

Cost Metric: I-Cost

Cost of a worst-case optimal plan is total **intersection-cost** of all operators.



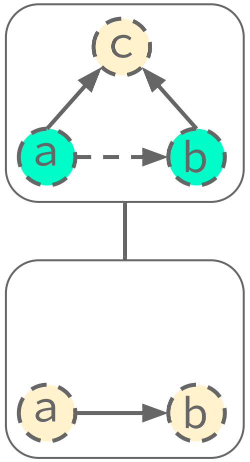
Plan [a,b,c]

Cost Metric: I-Cost

Cost of a worst-case optimal plan is total **intersection-cost** of all operators.

`***I-cost:*** size of intersected adj lists throughout execution.`

$$\sum_{Q_{k-1} \in Q_2 \ldots Q_{m-1}} \sum_{t \in Q_{k-1}} \sum_{\substack{(i,dir) \in A_{k-1} \\ \text{s.t. (i, dir) is accessed}}} |t[i].dir|$$



Plan [a,b,c]

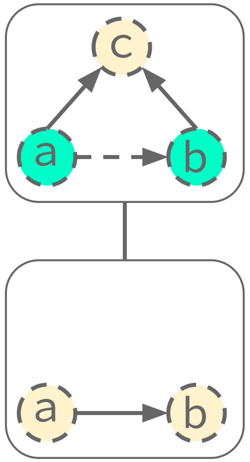
Cost Metric: I-Cost

Cost of a worst-case optimal plan is total **intersection-cost** of all operators.

I-cost: size of intersected adj lists throughout execution.

$$\underbrace{\sum_{Q_{k-1} \in Q_2 \ldots Q_{m-1}} \sum_{t \in Q_{k-1}}}_{\text{1) number of intermediate results.}} \sum_{\substack{(i,dir) \in A_{k-1} \\ \text{s.t. (i, dir) is accessed}}} |t[i].dir|$$

1) number of intermediate results.

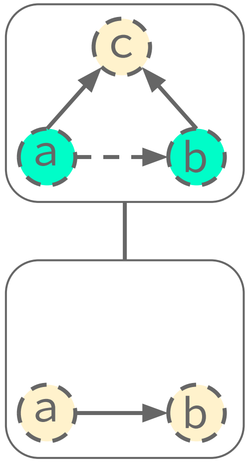


Plan [a,b,c]

Cost Metric: I-Cost

Cost of a worst-case optimal plan is total **intersection-cost** of all operators.

I-cost: size of intersected adj lists throughout execution.



Plan [a,b,c]

$$\underbrace{\sum_{Q_{k-1}\in Q_2\dots Q_{m-1}} \; \sum_{t\in Q_{k-1}}}_{\text{1) number of intermediate results.}} \; \underbrace{\sum_{\substack{(i,dir)\in A_{k-1}\\ \text{s.t. (i, dir) is accessed}}} |t[i].dir|}_{\text{2) size of adj. lists}}$$

Cost Metric: I-Cost

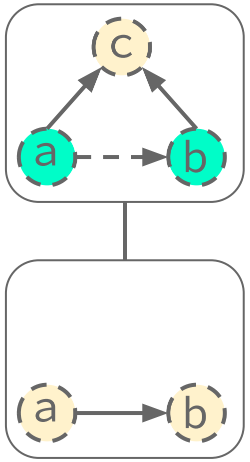
Cost of a worst-case optimal plan is total **intersection-cost** of all operators.

`*I-cost:*` size of intersected adj lists throughout execution.

$$\underbrace{\sum_{Q_{k-1} \in Q_2 \ldots Q_{m-1}} \sum_{t \in Q_{k-1}}}_{\text{1) number of intermediate results.}} \underbrace{\sum_{\substack{(i,dir) \in A_{k-1} \\ \text{s.t. (i, dir) is accessed}}} |t[i].dir|}_{\text{2) size of adj. lists}}$$

1) number of intermediate results.

2) size of adj. lists



Plan [a,b,c]

I-Cost captures both effects & # of intermediate results and size of adj lists is estimated using a subgraph catalogue.
Summary-based Cardinality estimation using sampling.

Graphflow Hybrid Plan

Graphflow

Dynamic Programming
Cost-Based Optimizer

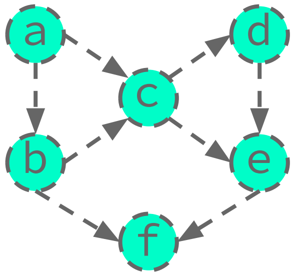
Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

Difference with classic optimizer: At each level, considers multiway joins to a query vertices.

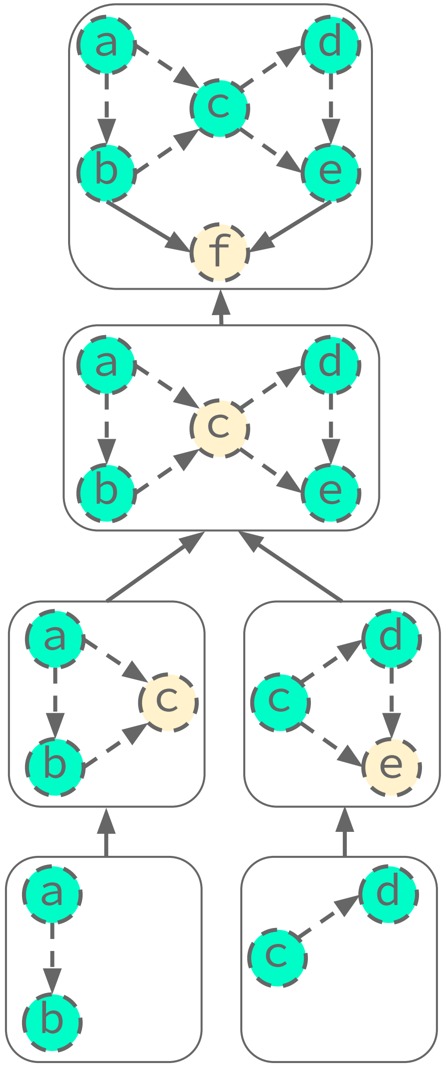
Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer



Example Graphflow Hybrid Plan

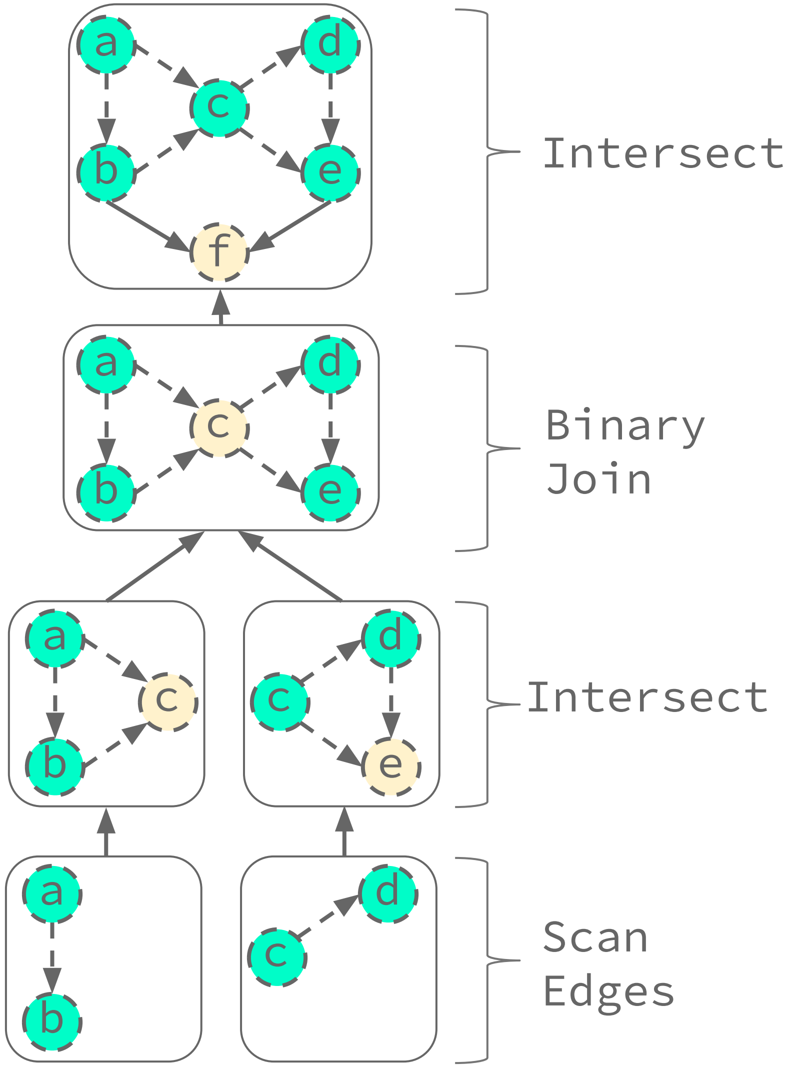
Dynamic Programming
Cost-Based Optimizer



Example Graphflow Hybrid Plan

Graphflow

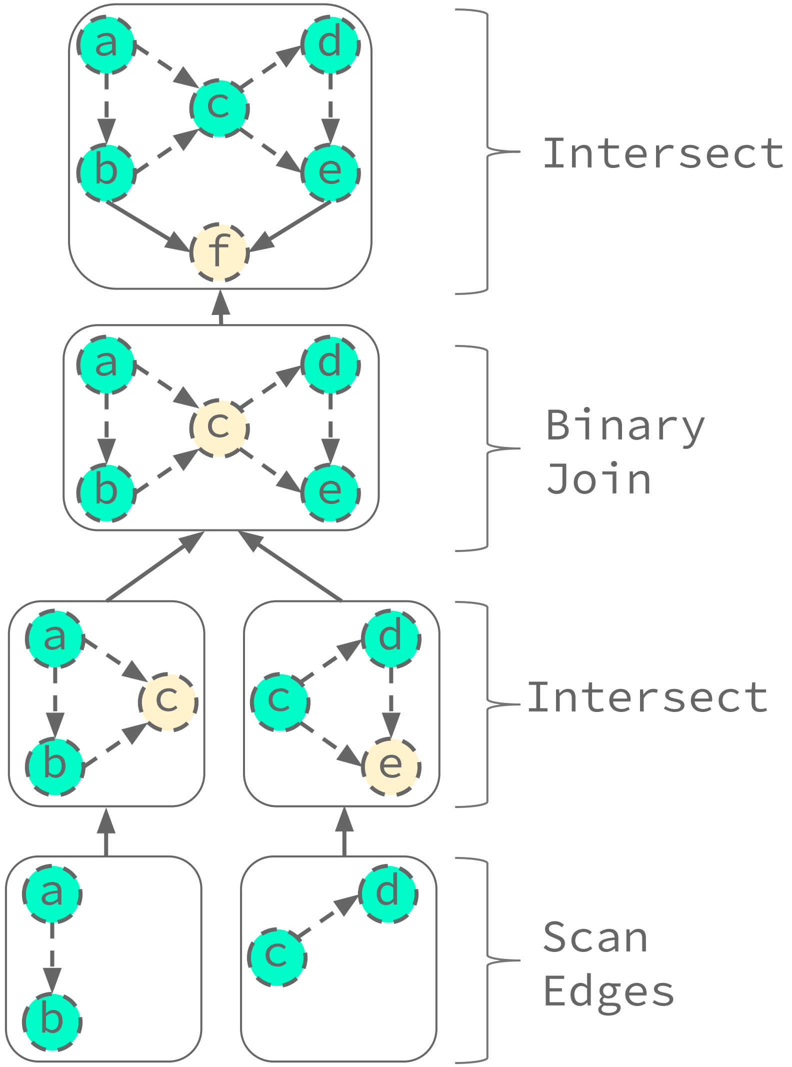
Dynamic Programming Cost-Based Optimizer

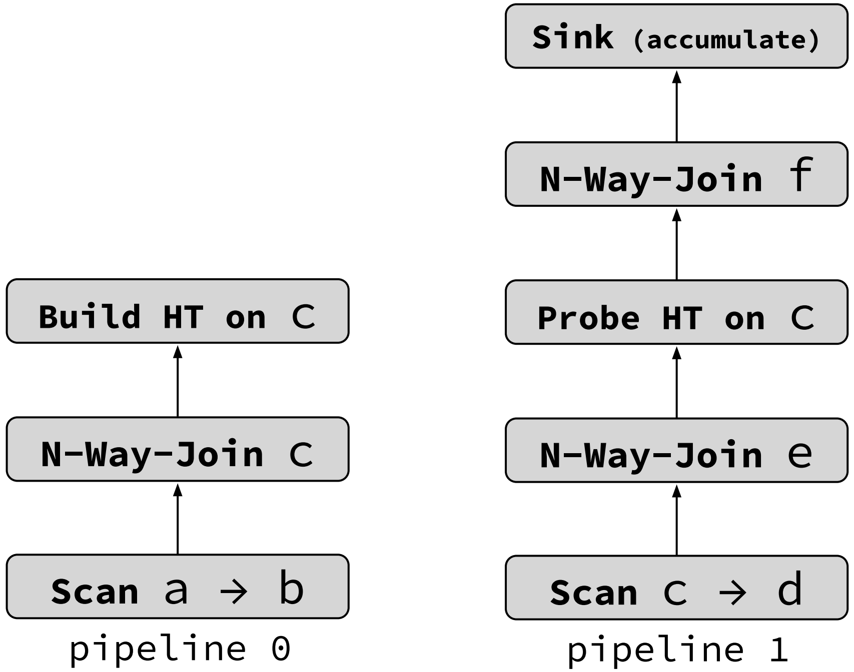


UNIVERSITY OF WATERLOO | DSg Data Systems Group

Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer





Sink (accumulate)

N-Way-Join f

Probe HT on c

N-Way-Join e

Scan c → d

pipeline 1

Build HT on c

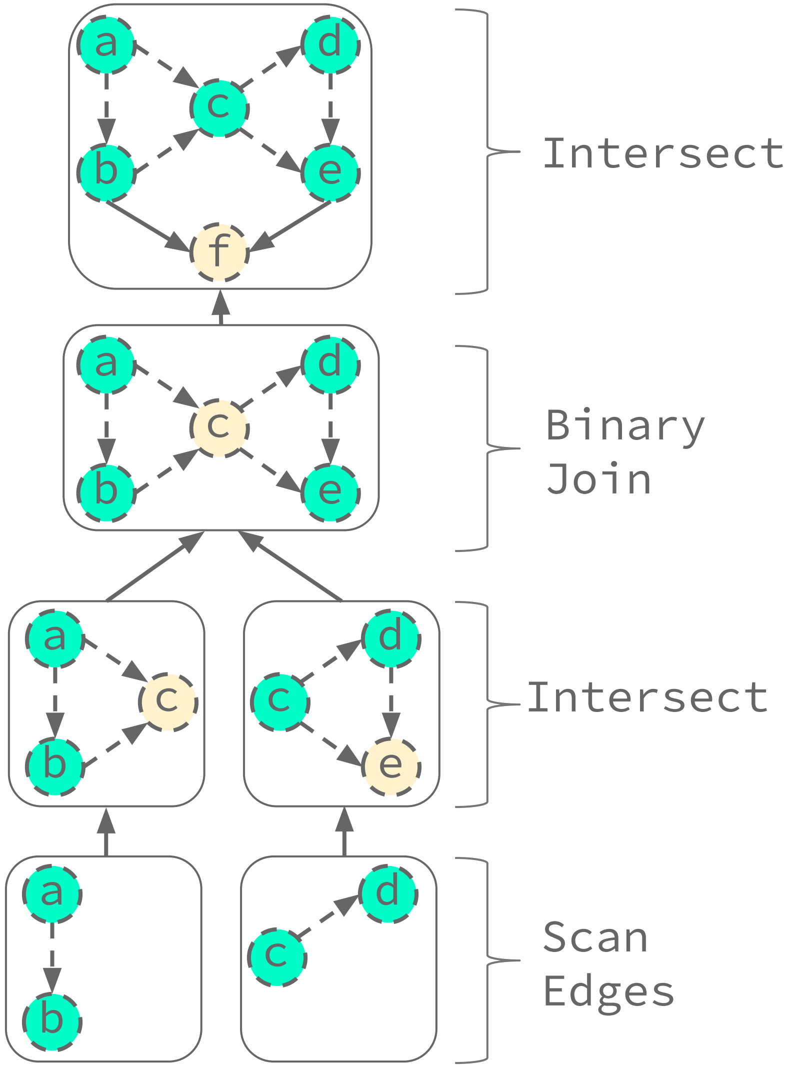
N-Way-Join c

Scan a → b

pipeline 0

Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer

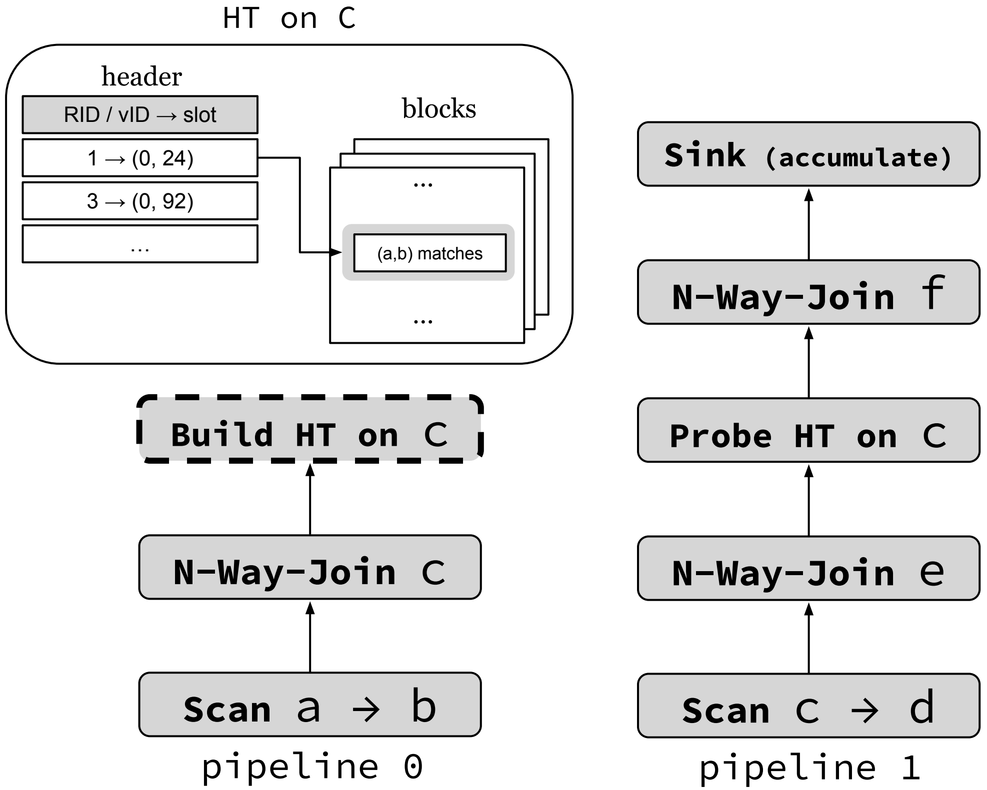


Intersect

Binary Join

Intersect

Scan Edges


HT on C

header

RID / vID → slot
1 → (0, 24)
3 → (0, 92)
…

blocks

…

(a,b) matches

…

Sink (accumulate)

N-Way-Join f

Probe HT on C

N-Way-Join e

Scan c → d

pipeline 1

Build HT on C

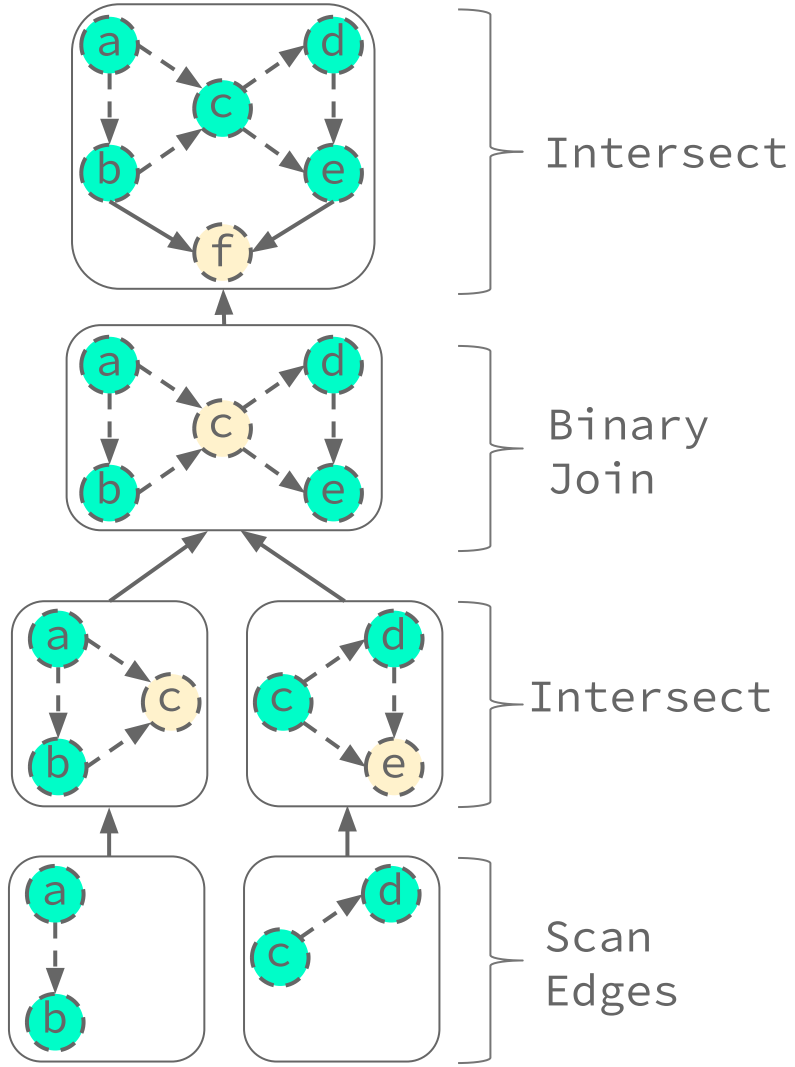
N-Way-Join c

Scan a → b

pipeline 0

Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

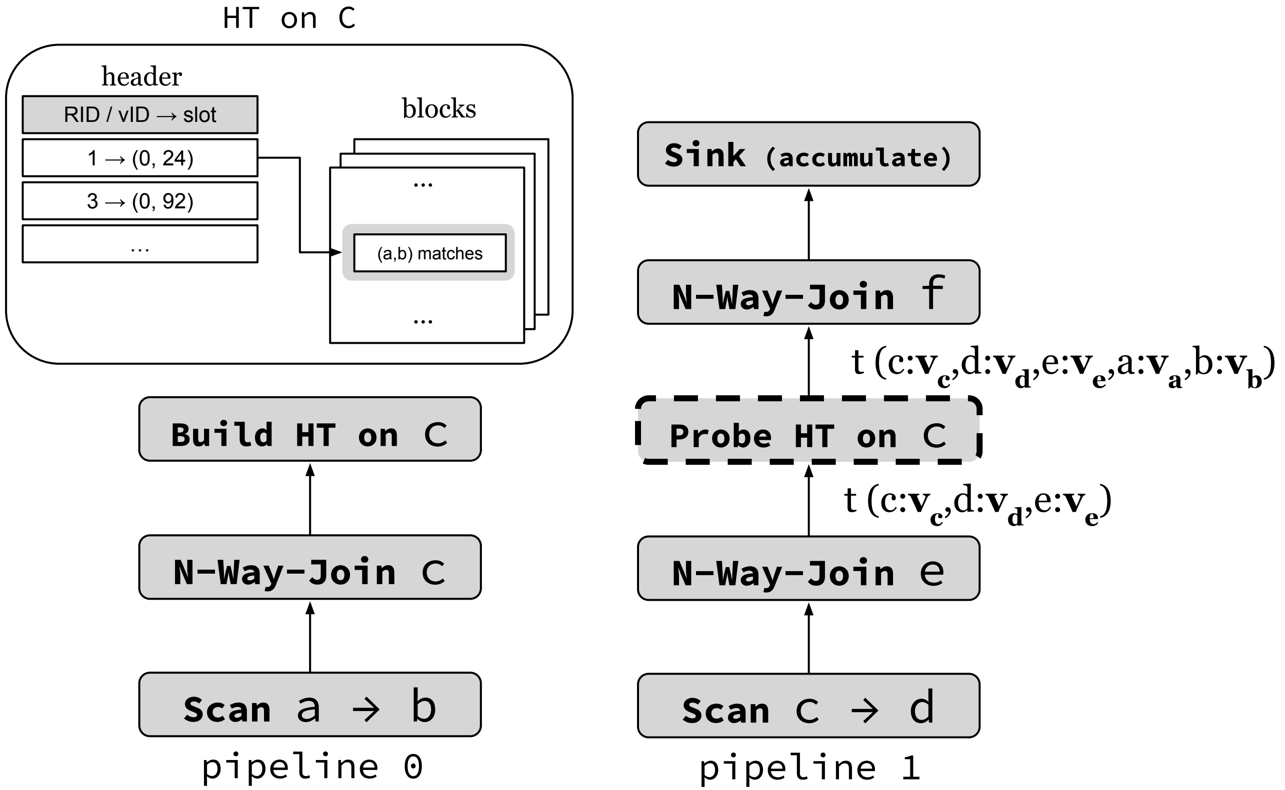


Intersect

Binary Join

Intersect

Scan Edges



HT on C

header

RID / vID → slot
1 → (0, 24)
3 → (0, 92)
…

blocks

…

(a,b) matches

…

Build HT on C

N-Way-Join c

Scan a → b

pipeline 0

Sink (accumulate)

N-Way-Join f

$t\,(c{:}\mathbf{v}_c, d{:}\mathbf{v}_d, e{:}\mathbf{v}_e, a{:}\mathbf{v}_a, b{:}\mathbf{v}_b)$

Probe HT on C

$t\,(c{:}\mathbf{v}_c, d{:}\mathbf{v}_d, e{:}\mathbf{v}_e)$

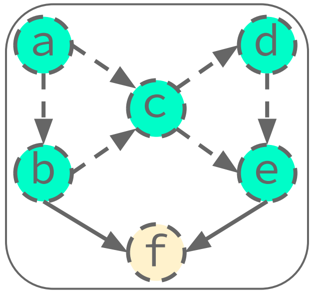
N-Way-Join e

Scan c → d

pipeline 1

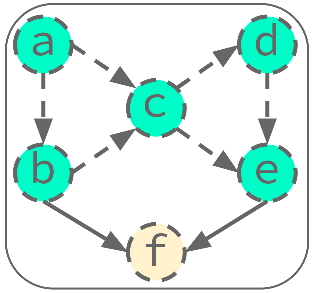
Example EmptyHeaded Plan

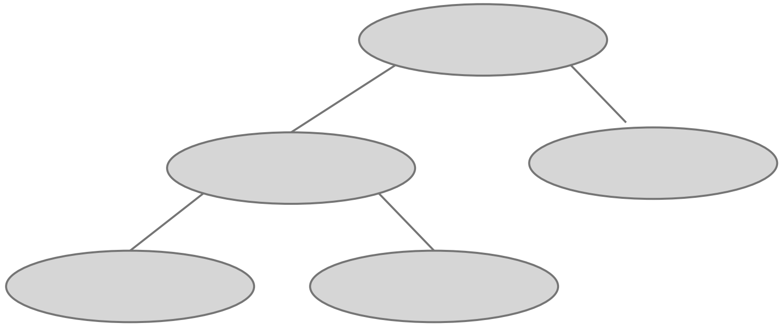
Generalized Hypertree Decomposition
Cost-Based Optimizer



Example EmptyHeaded Plan

Generalized Hypertree Decomposition
Cost-Based Optimizer

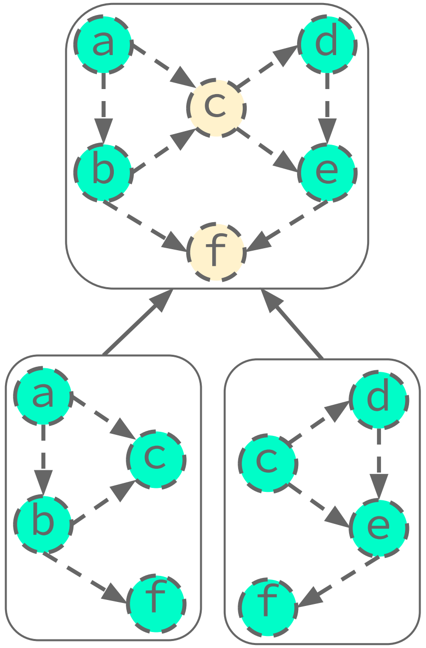




A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.

Example EmptyHeaded Plan

Generalized Hypertree Decomposition
Cost-Based Optimizer

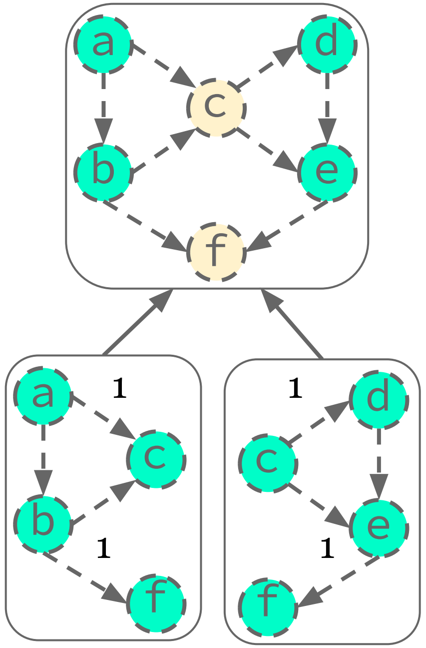


A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.

Example EmptyHeaded Plan

EMPTYHEADED

Generalized Hypertree Decomposition
Cost-Based Optimizer

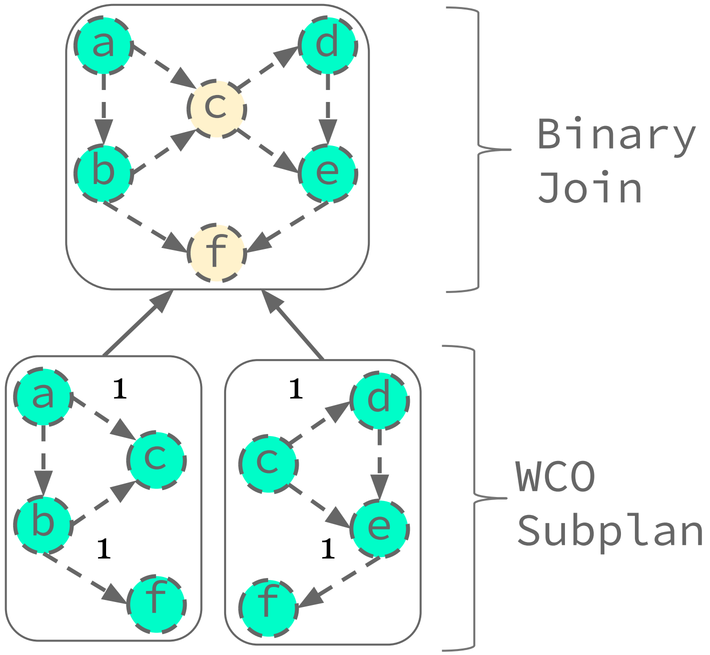


A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.

EmptyHeaded picks GHD with minimum fractional hypertree width which is the maximum AGM bound of any of the leaves.

Example EmptyHeaded Plan

Generalized Hypertree Decomposition
Cost-Based Optimizer

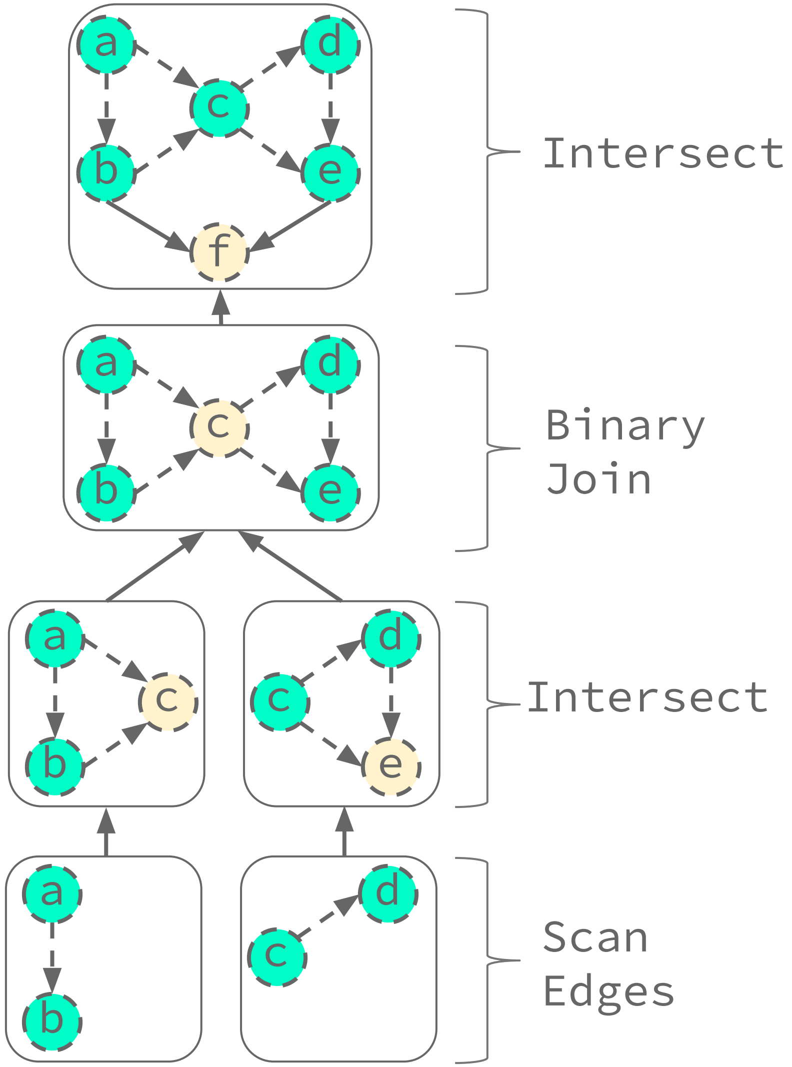


A GHD is similar to a syntax tree of a relational expression. Nodes represent a join and projection operation and edges indicate data dependencies.

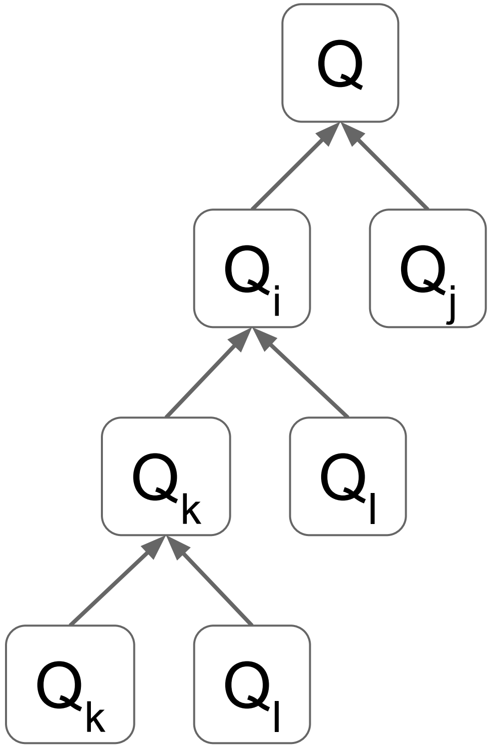
EmptyHeaded picks GHD with minimum fractional hypertree width which is the maximum AGM bound of any of the leaves.

Example Graphflow Hybrid Plan

Dynamic Programming
Cost-Based Optimizer

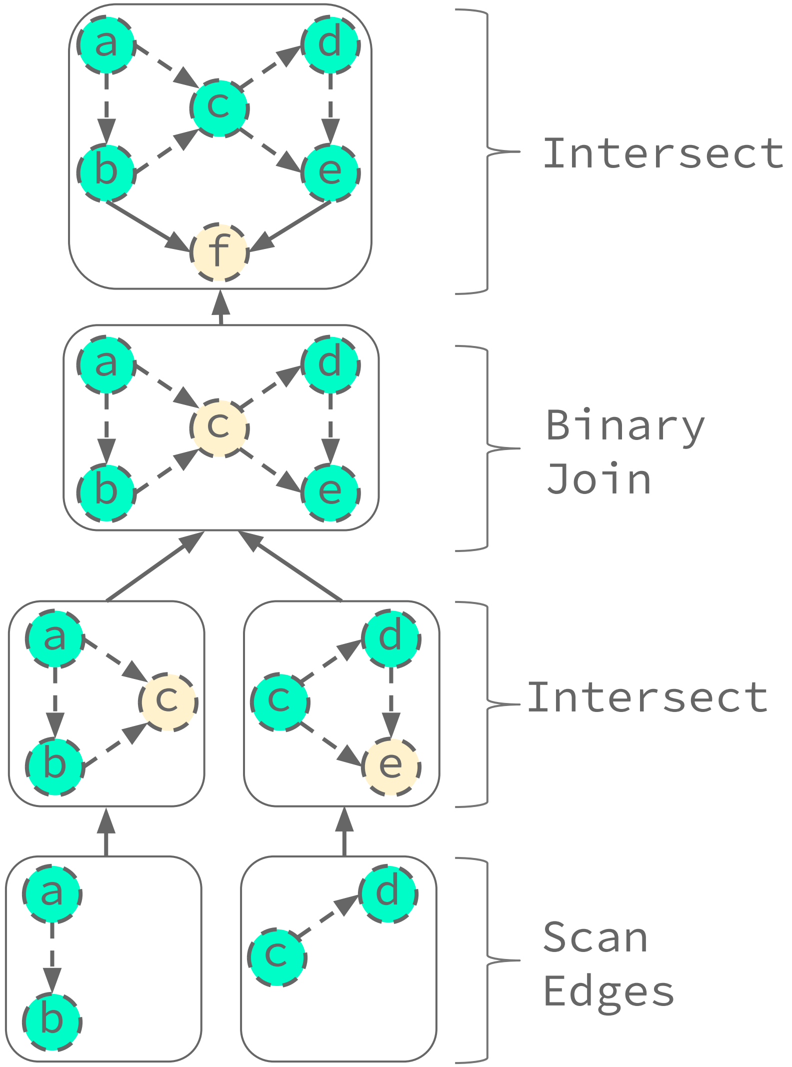


Generalized Hypertree Decomposition
Cost-Based Optimizer



Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer



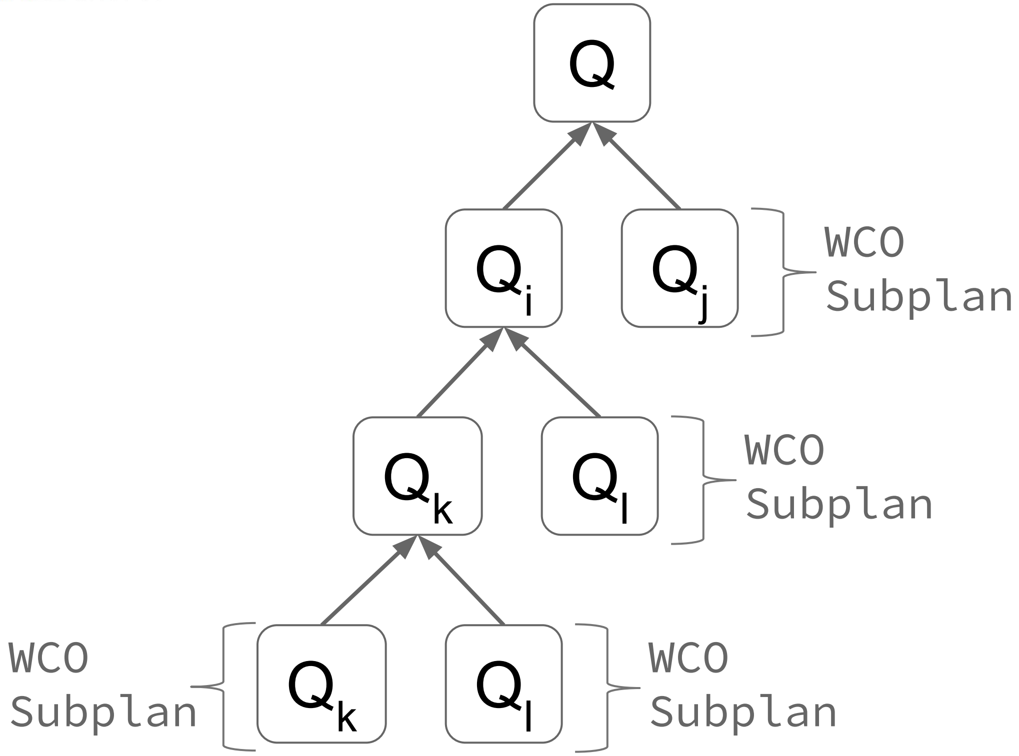
Intersect

Binary Join

Intersect

Scan Edges

Generalized Hypertree Decomposition Cost-Based Optimizer



WCO Subplan

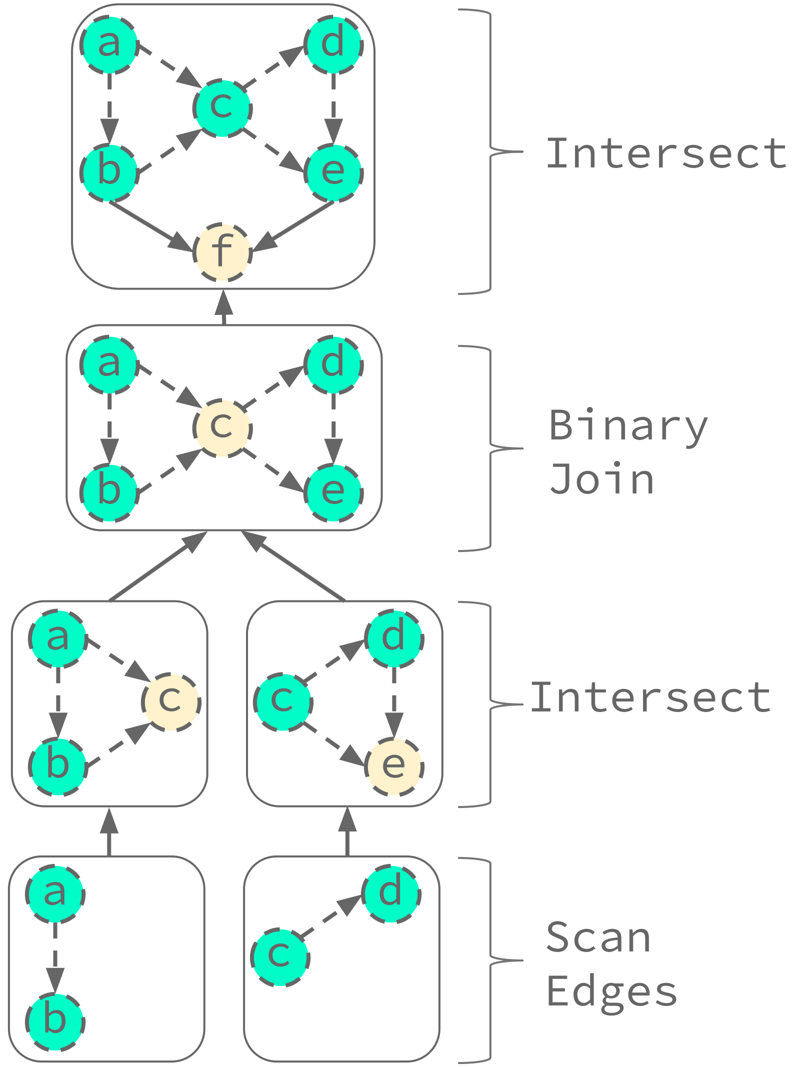
WCO Subplan

WCO Subplan

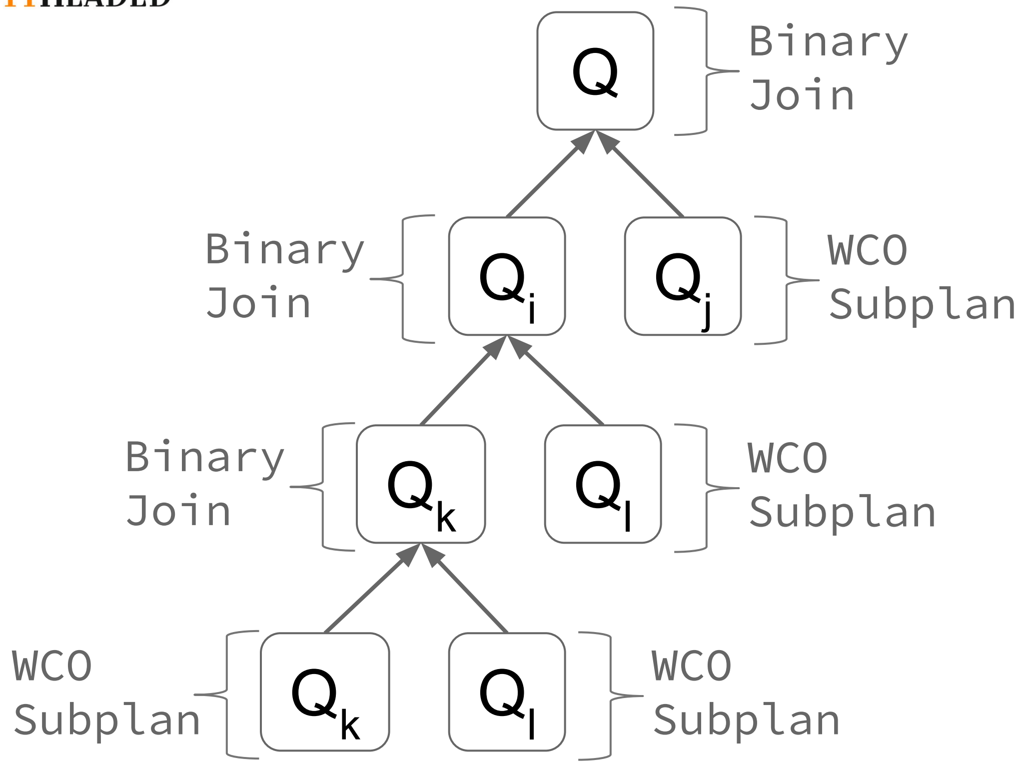
WCO Subplan

Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer

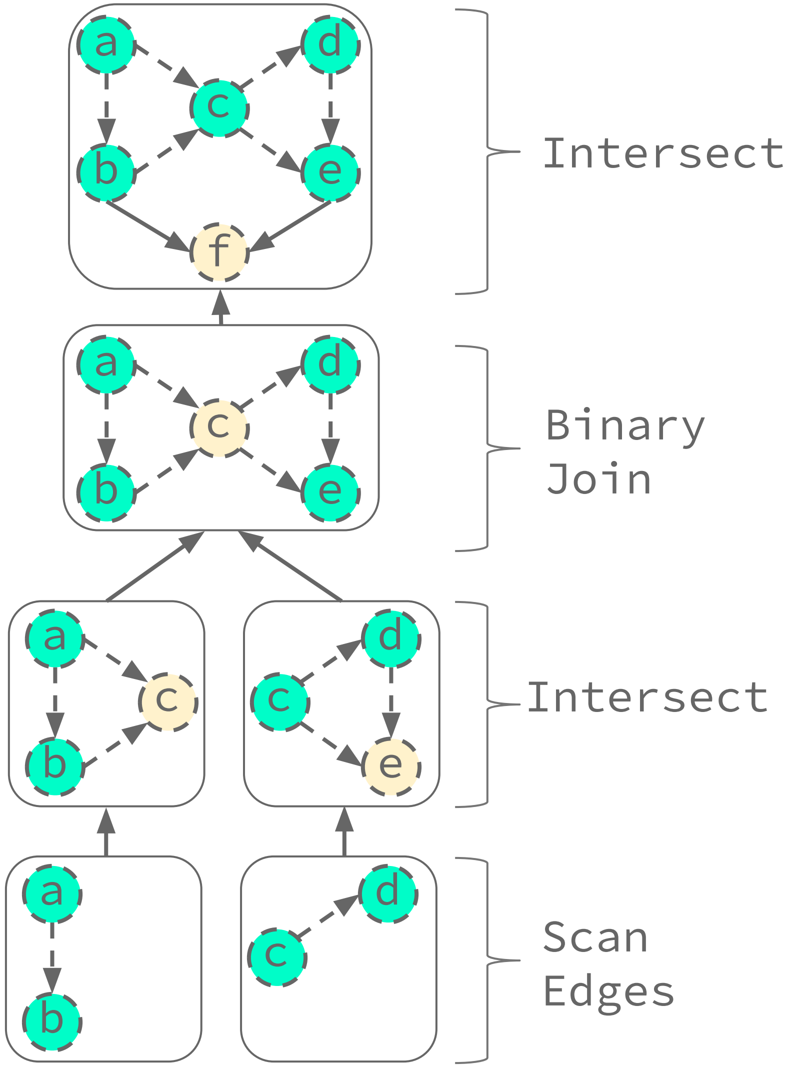


Generalized Hypertree Decomposition Cost-Based Optimizer

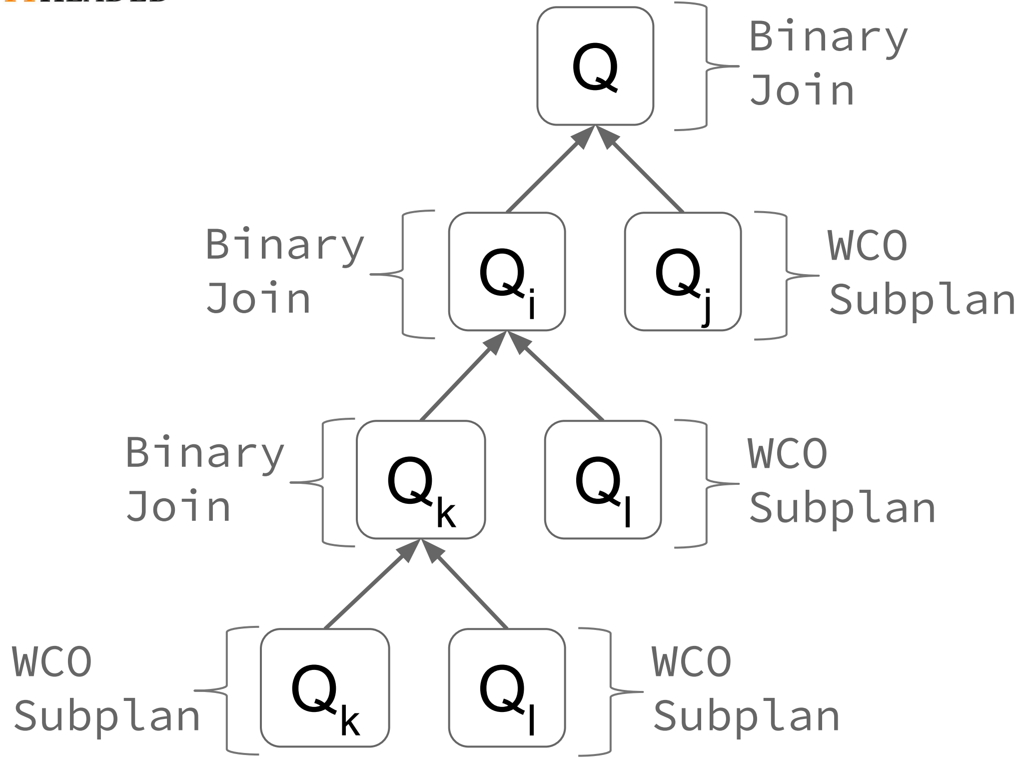


Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer



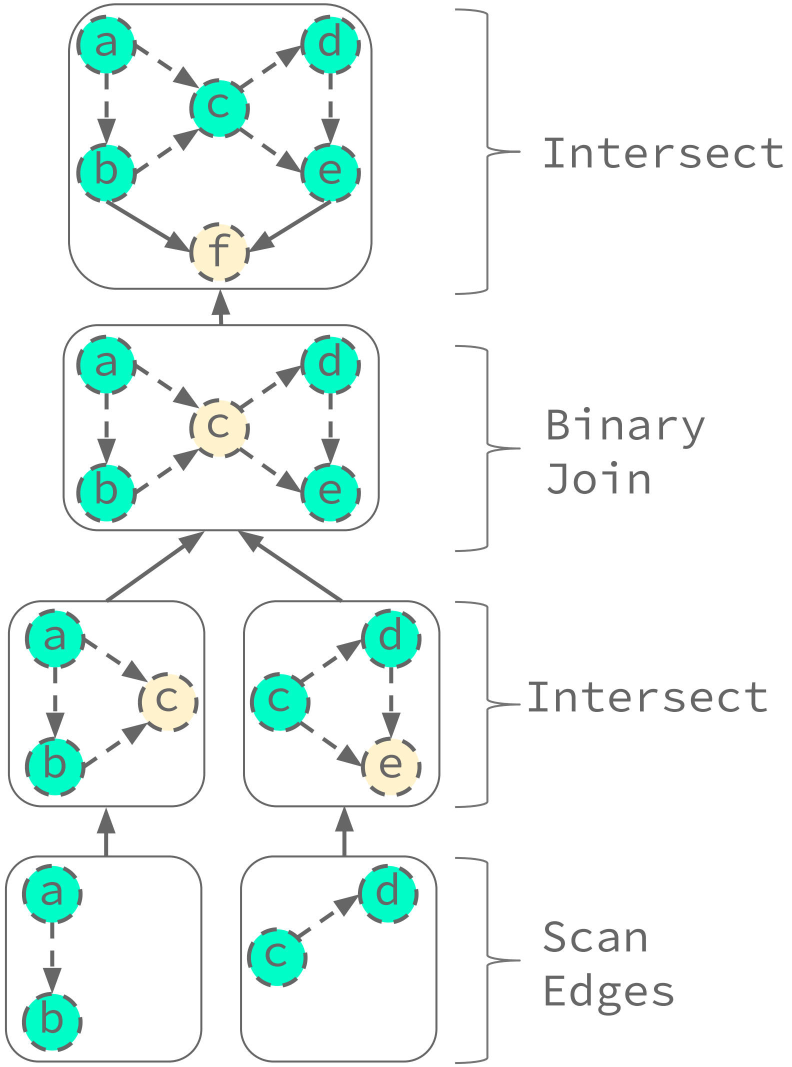
Generalized Hypertree Decomposition Cost-Based Optimizer



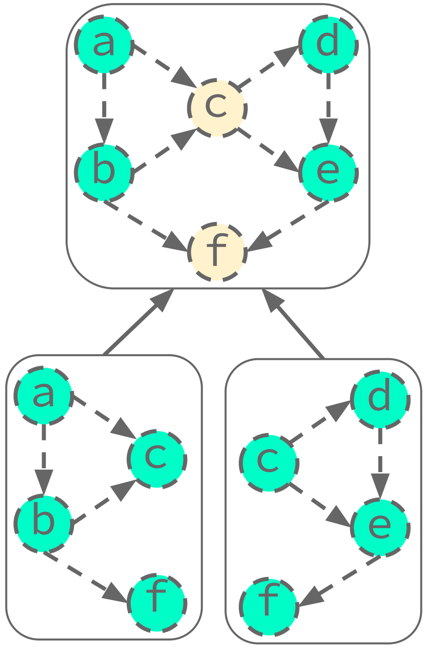
Generates only WCO subplans followed by multiple binary joins.

Example Graphflow Hybrid Plan

Dynamic Programming Cost-Based Optimizer



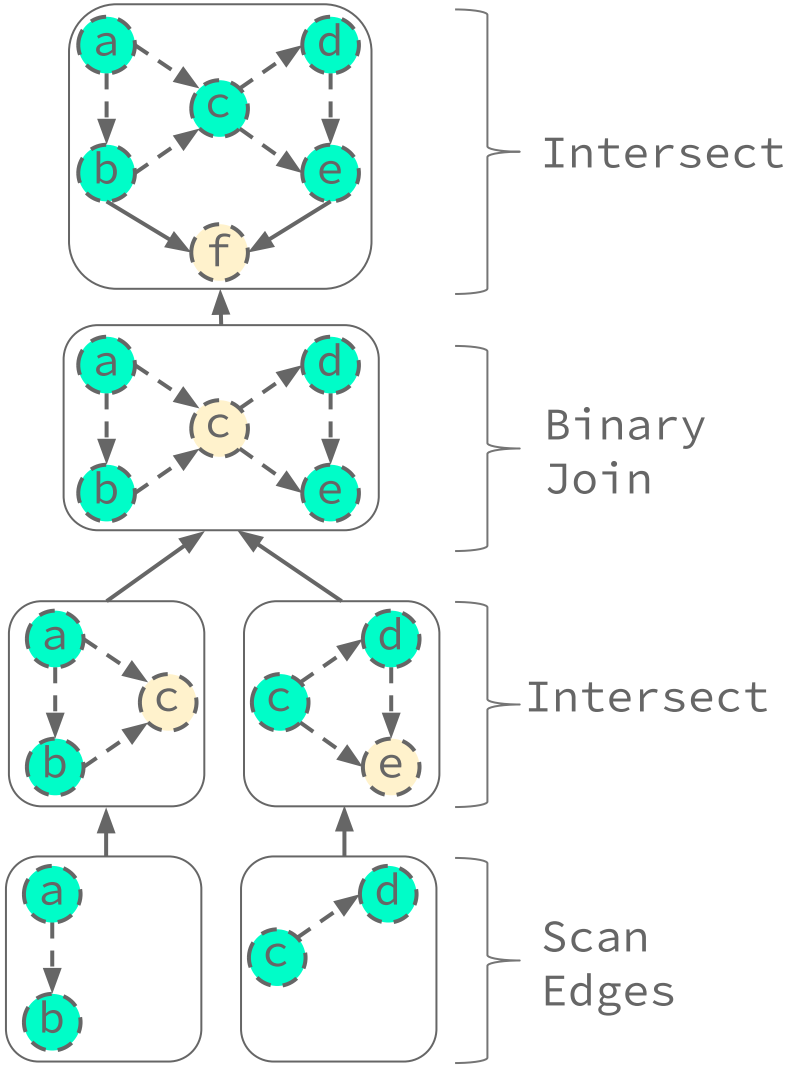
Generalized Hypertree Decomposition Cost-Based Optimizer



Example Graphflow Hybrid Plan

Graphflow

Dynamic Programming
Cost-Based Optimizer



Intersect

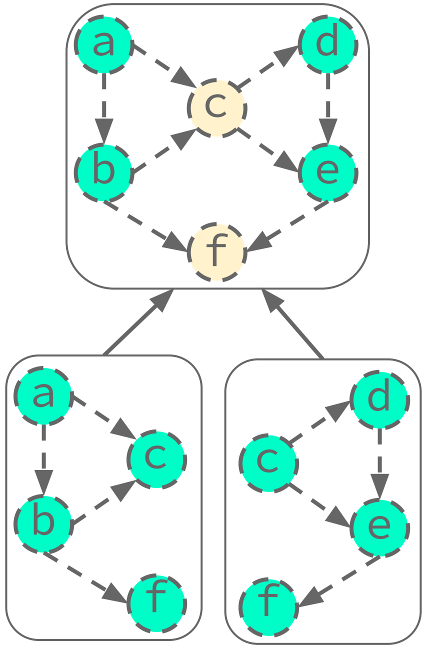
Binary
Join

Intersect

Scan
Edges

EMPTYHEADED

Generalized Hypertree Decomposition
Cost-Based Optimizer



Graphflow	24.7 secs
EmptyHeaded (EH)	> 30 mins
EH in Graphflow	5.8 mins (**14x**)

Amazon (V=403K, E=3.4M)

Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case Optimal Joins (WCOJs)**

   Handling Intermediate Size Growth for Cyclic Joins

   2.1. Foundations

   2.2 System Integration Approaches

   i) **Index-based WCOJs (Graphflow & EmptyHeaded)**

   ii) **Hash-based WCOJs (Umbra)**

   - **Optimization approaches (cost-based DP, GHD, rule-based)**

3) **Factorized Query Processing**

Index-based Worst-case Optimal Joins

Generic Join uses pre-sorted indexes which in graph terms map to adjacency list indexes.

Relational Systems such as EmptyHeaded, LogicBlox, and relationalAI use sorted trie indexes.

Index-based Worst-case Optimal Joins

Graphflow

Generic Join uses pre-sorted indexes which in graph terms map to adjacency list indexes.

Relational Systems such as EmptyHeaded, LogicBlox, and relationalAI use sorted trie indexes.

High-cost associated with keeping indexes up-to-date for a general database system!

- EmptyHeaded keeps all possible Tries with different sorts on the attributes → Not practical.
- The adjacency list indexes of Graphflow are specialized for subgraph query workloads.

Index-based Worst-case Optimal Joins

Graphflow

Generic Join uses pre-sorted indexes which in graph terms map to adjacency list indexes.

Relational Systems such as EmptyHeaded, LogicBlox, and relationalAI use sorted trie indexes.

High-cost associated with keeping indexes up-to-date for a general database system!

- EmptyHeaded keeps all possible Tries with different sorts on the attributes → Not practical.
- The adjacency list indexes of Graphflow are specialized for subgraph query workloads.
    - What is needed?

      A generic WCOJ algorithm for a changing workload e.g., HTAP systems that does not rely on these pre-computed indexes that analytical systems can afford building and updating.

Hash-based Worst-case Optimal Joins

Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Hash-based Worst-case Optimal Joins

UMBRA

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Relation **R1**: a | c ⋈ Relation **R2**: b | c ⋈ Relation **R3**: a | b | c

Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Relation **R1**			Relation **R2**			Relation **R3**		
a	**c**	⋈	**b**	**c**	⋈	**a**	**b**	**c**

Attribute ordering for evaluation: `[a, b, c]`

Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Relation **R1**

a	c

⋈

Relation **R2**

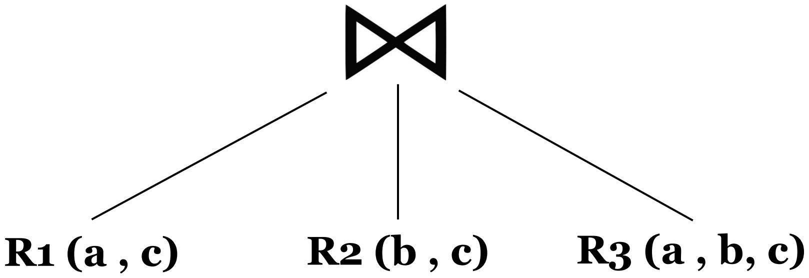
b	c

⋈

Relation **R3**

a	b	c

Attribute ordering for evaluation: `[a, b, c]`

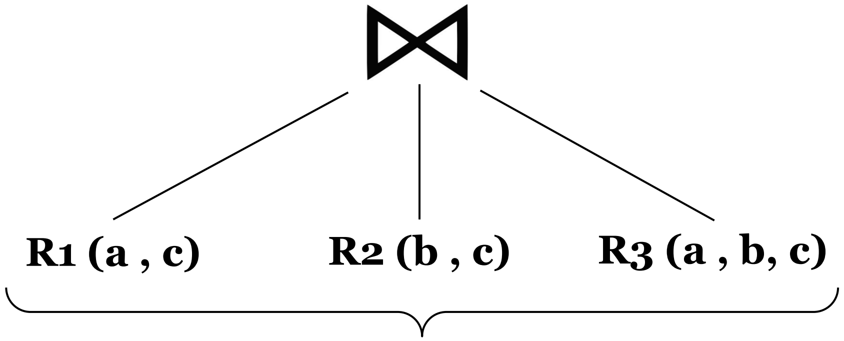


Hash-based Worst-case Optimal Joins

Building Hash Tries on-the-fly that can be used within the worst-case optimal join algorithm.

Relation **R1**			Relation **R2**			Relation **R3**		
a	c	⋈	b	c	⋈	a	b	c

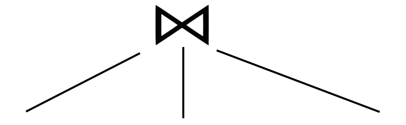
Attribute ordering for evaluation: `[a, b, c]`



R1 (a , c) **R2 (b , c)** **R3 (a , b, c)**

Build Hash Tries for R1, R2, and R3 following
attribute ordering [a, b, c] in Trie-levels

Hash-based Worst-case Optimal Joins



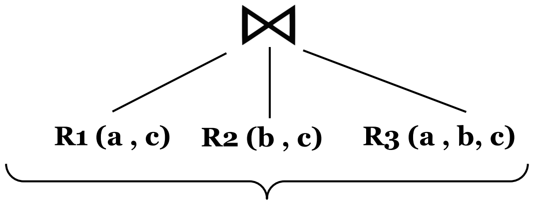
⋈

R1 (a , c) **R2 (b , c)** **R3 (a , b, c)**

Build Hash Tries for R1, R2, and R3 following
attribute ordering [a, b, c] in Trie-levels

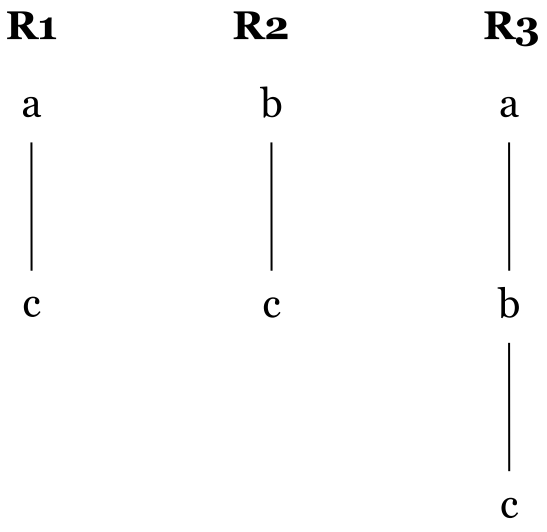
Attribute ordering: `[a, b, c]`

Hash-based Worst-case Optimal Joins

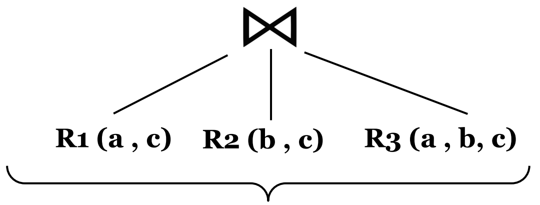


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [`a`, `b`, `c`]

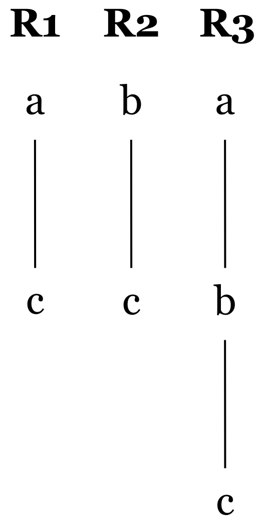


Build Hash Tries



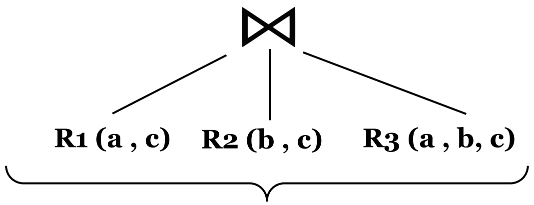
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]



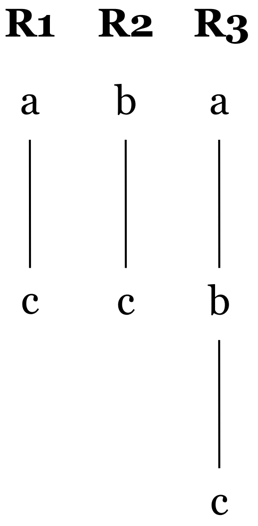
Build Hash Tries

A multiway join of K relations → Building K Hash Tries.

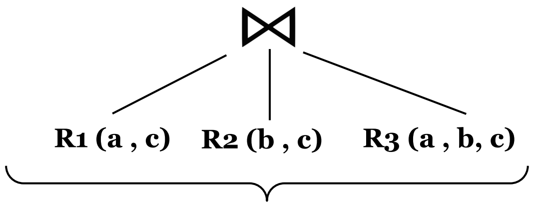


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [`a`, `b`, `c`]



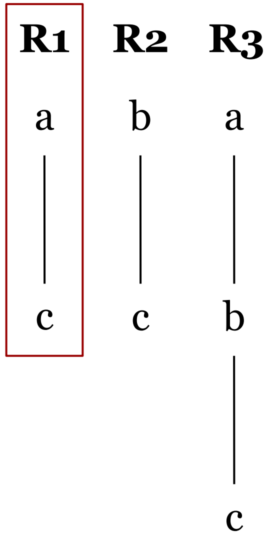
Build Hash Tries



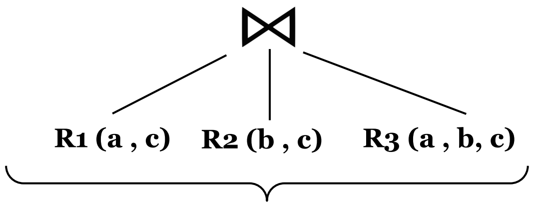
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

R1 (a, c)



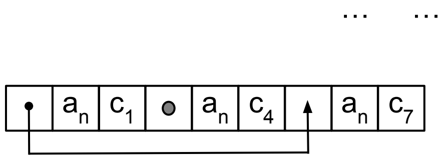
Build Hash Tries

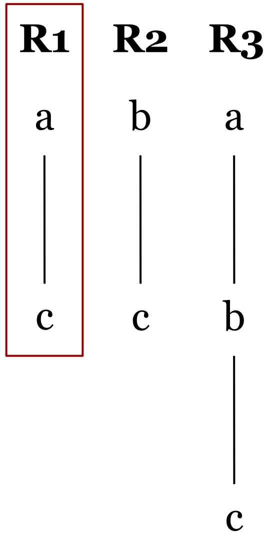


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

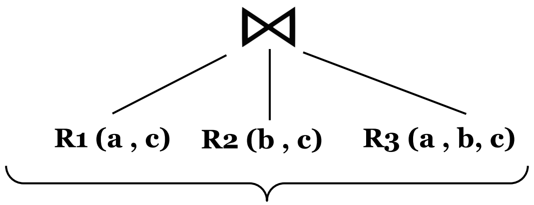
Attribute ordering: [a, b, c]

R1 (a, c)





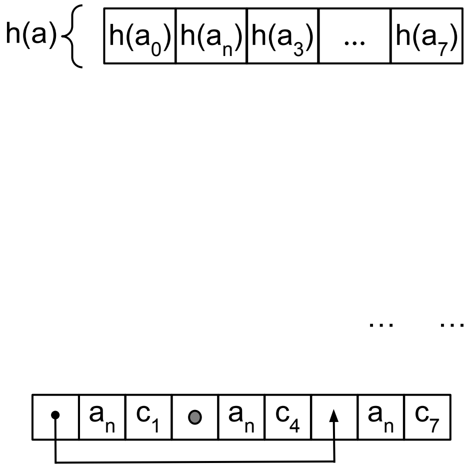
Build Hash Tries

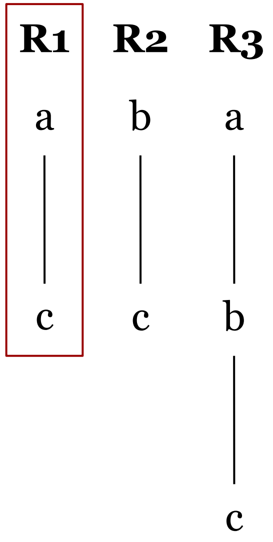


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

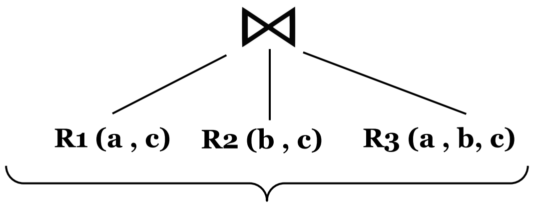
Attribute ordering: [a, b, c]

R1 (a, c)





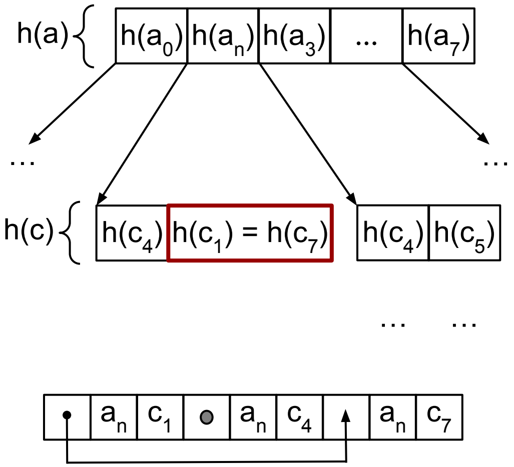
Build Hash Tries

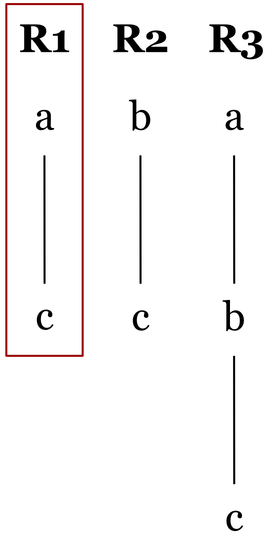


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

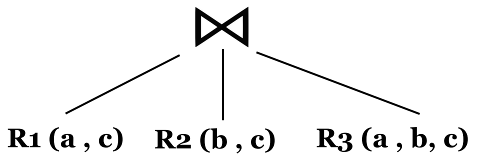
Attribute ordering: [a, b, c]

R1 (a, c)





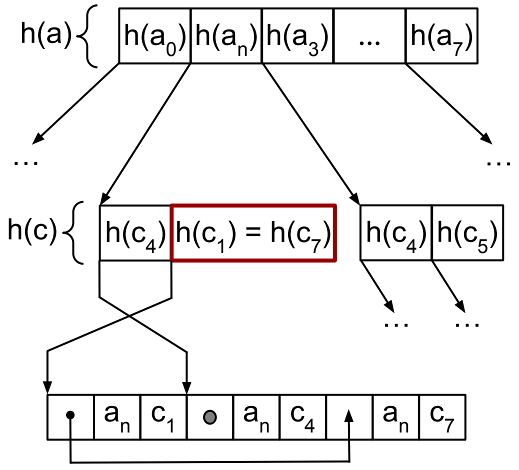
Build Hash Tries

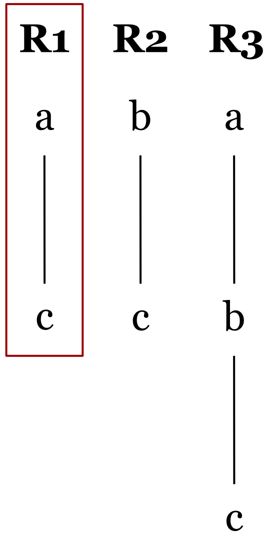


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

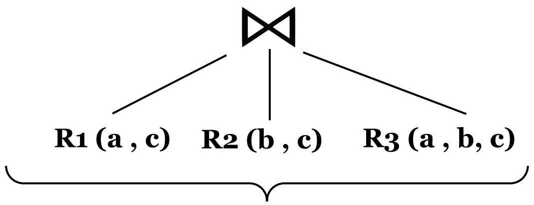
Attribute ordering: `[a, b, c]`

R1 (a, c)





Build Hash Tries

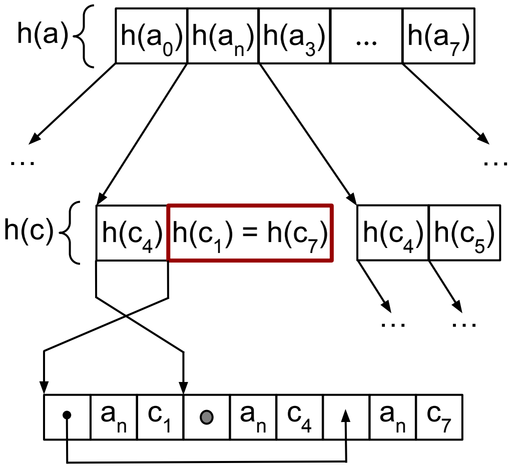


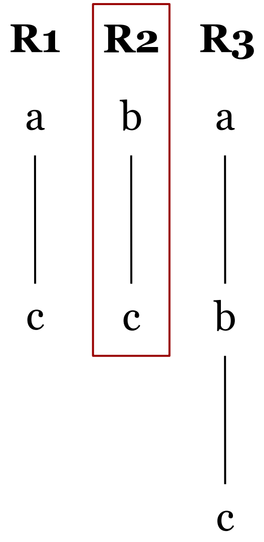
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

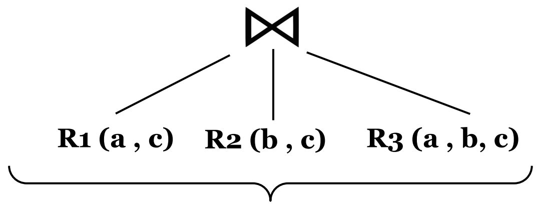
R1 (a, c)

R2 (b, c)





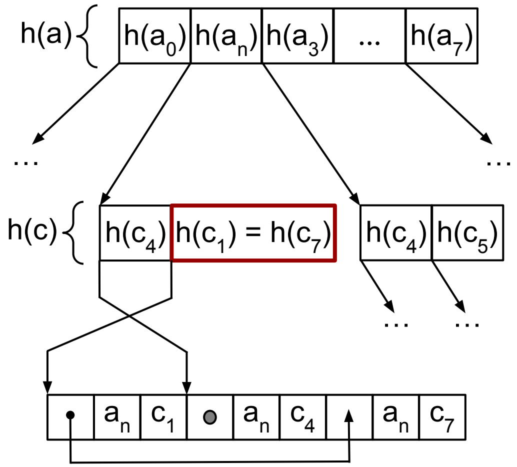
Build Hash Tries



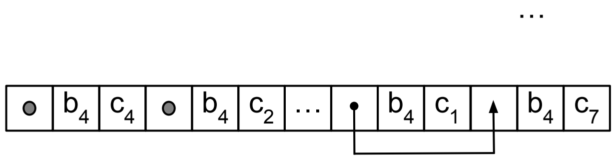
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

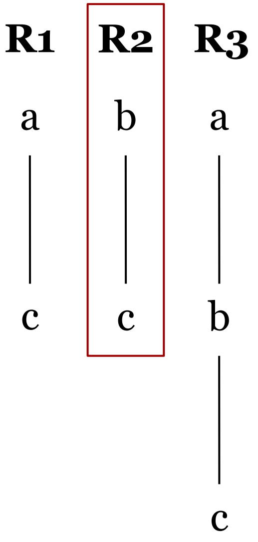
Attribute ordering: [a, b, c]

R1 (a, c)



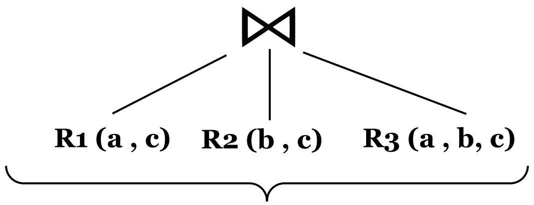
R2 (b, c)





Build Hash Tries

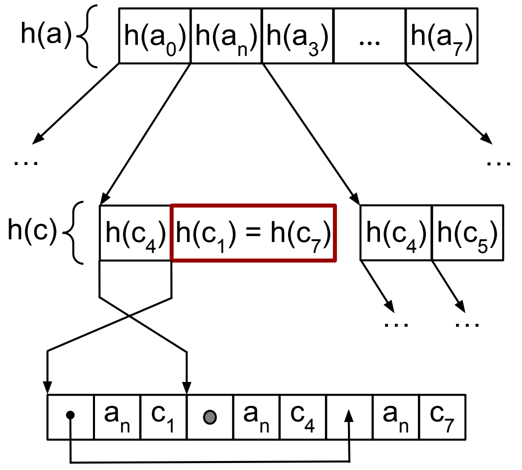
UMBRA



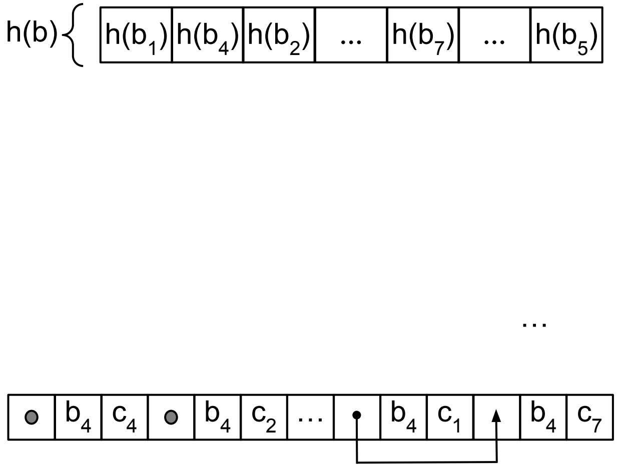
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

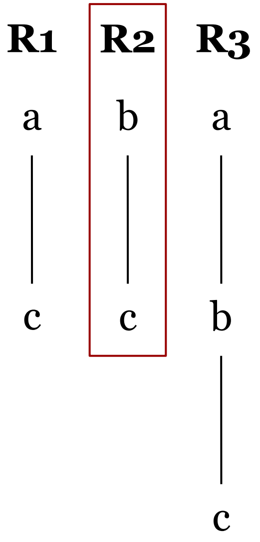
Attribute ordering: [a , b , c]

R1 (a, c)

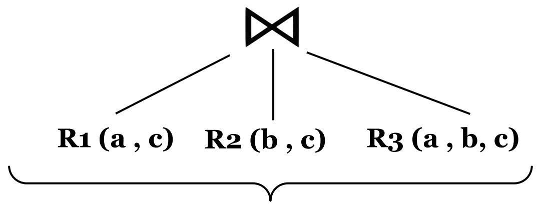


R2 (b, c)





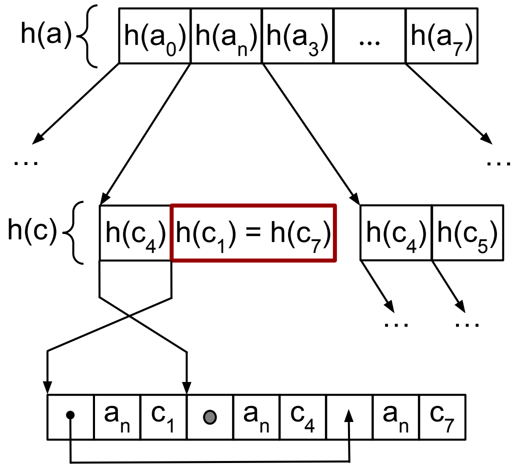
Build Hash Tries



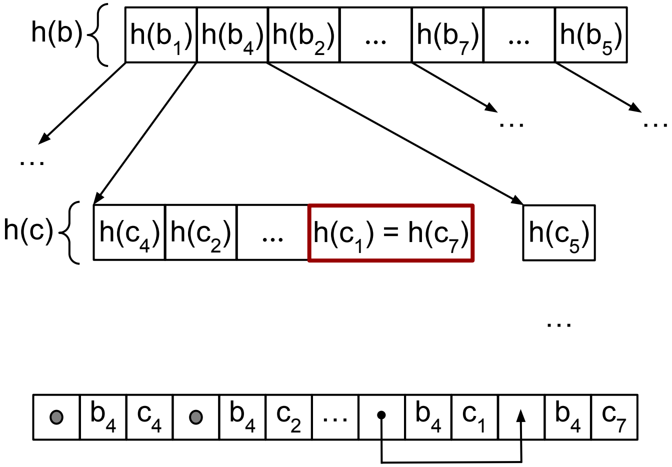
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

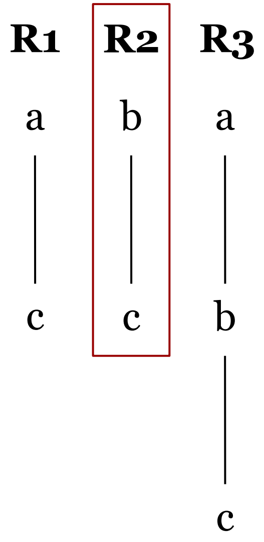
Attribute ordering: [a, b, c]

R1 (a, c)

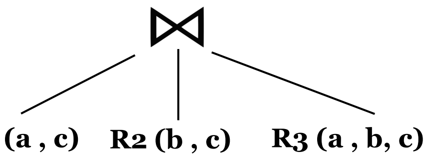


R2 (b, c)





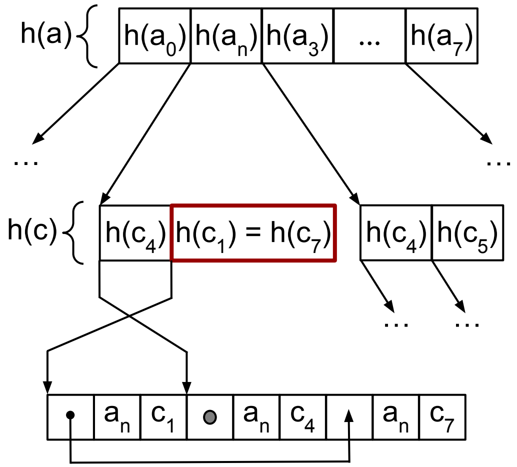
Build Hash Tries



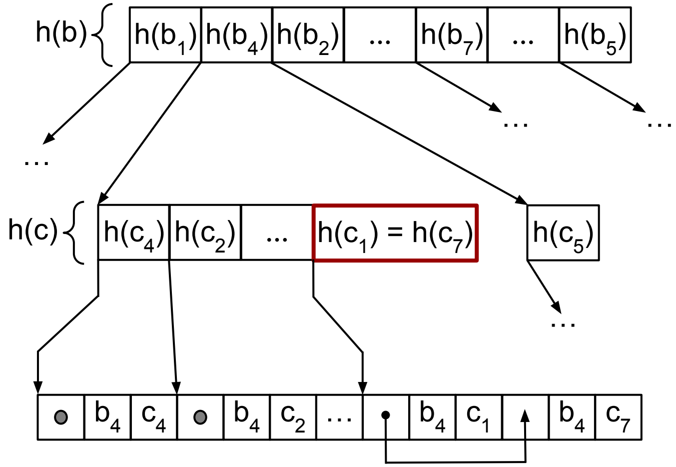
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

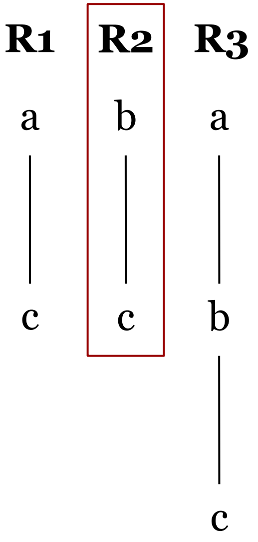
Attribute ordering: [a, b, c]

R1 (a, c)

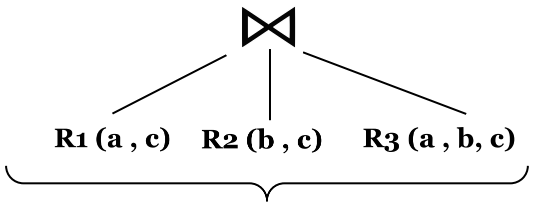


R2 (b, c)





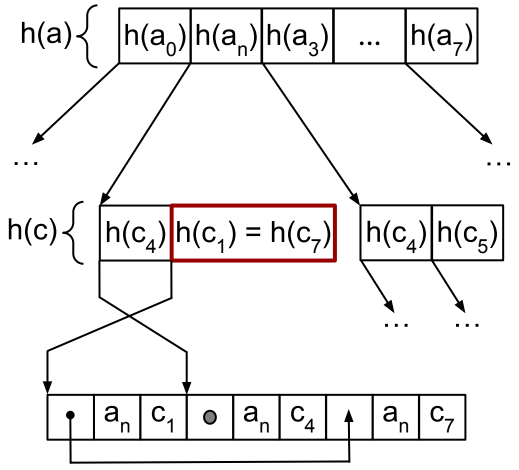
Build Hash Tries



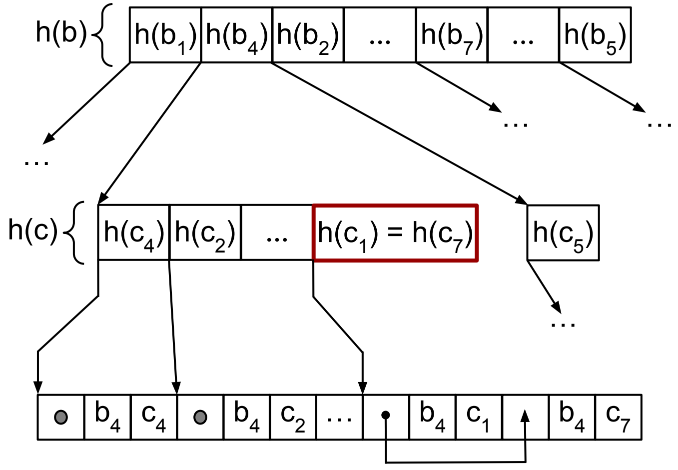
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

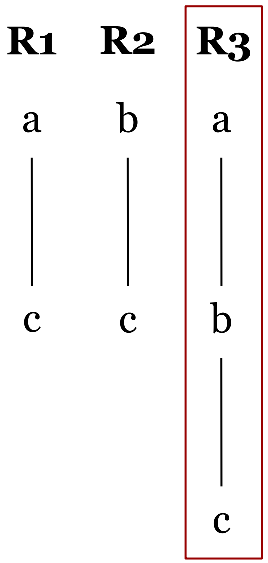
R1 (a, c)



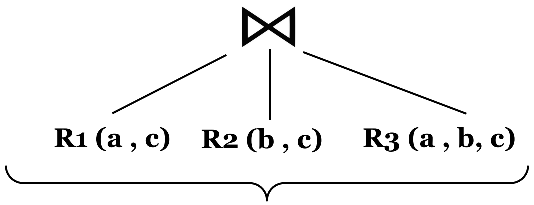
R2 (b, c)



R3 (a, b, c)



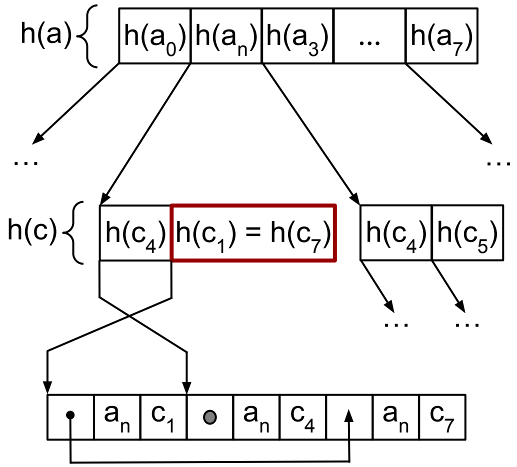
Build Hash Tries



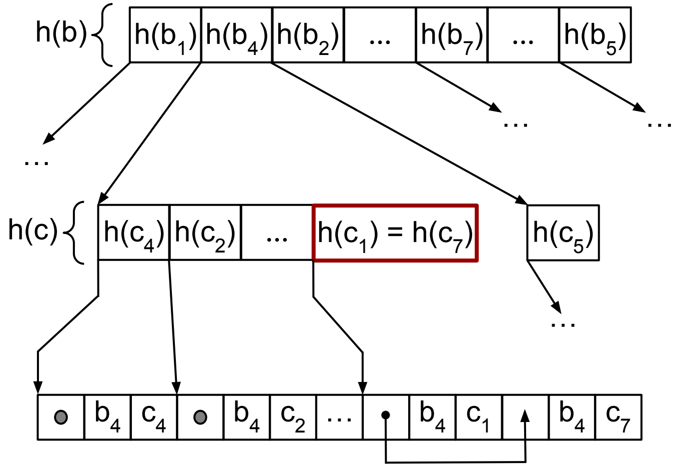
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

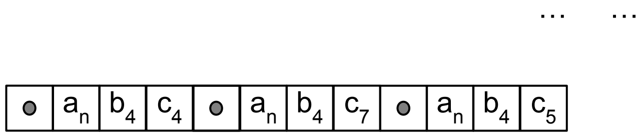
R1 (a, c)

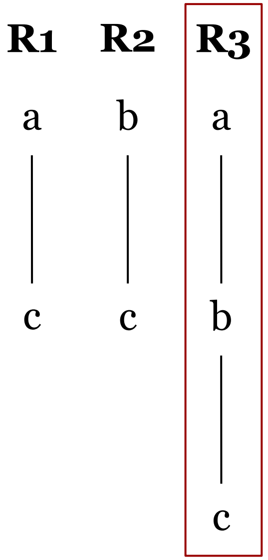


R2 (b, c)



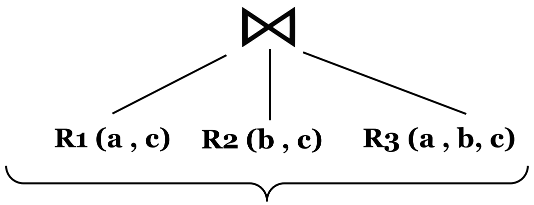
R3 (a, b, c)





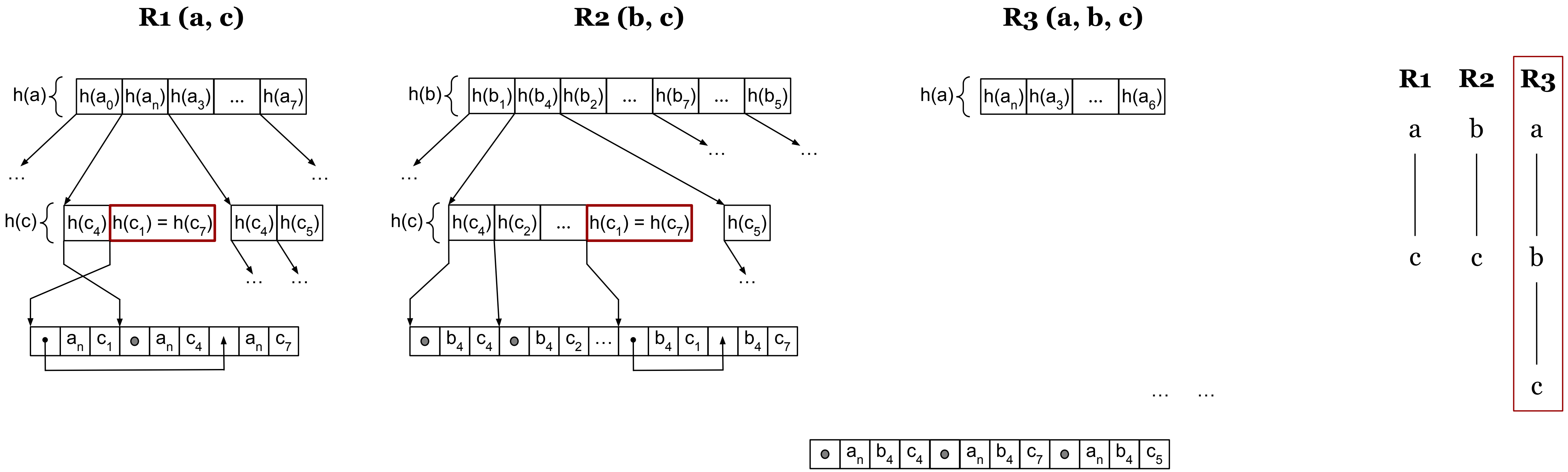
Build Hash Tries

UMBRA



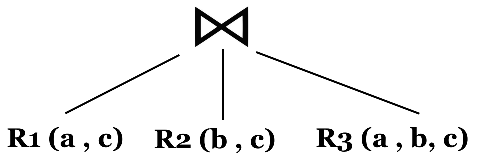
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]



UNIVERSITY OF WATERLOO | DSG Data Systems Group

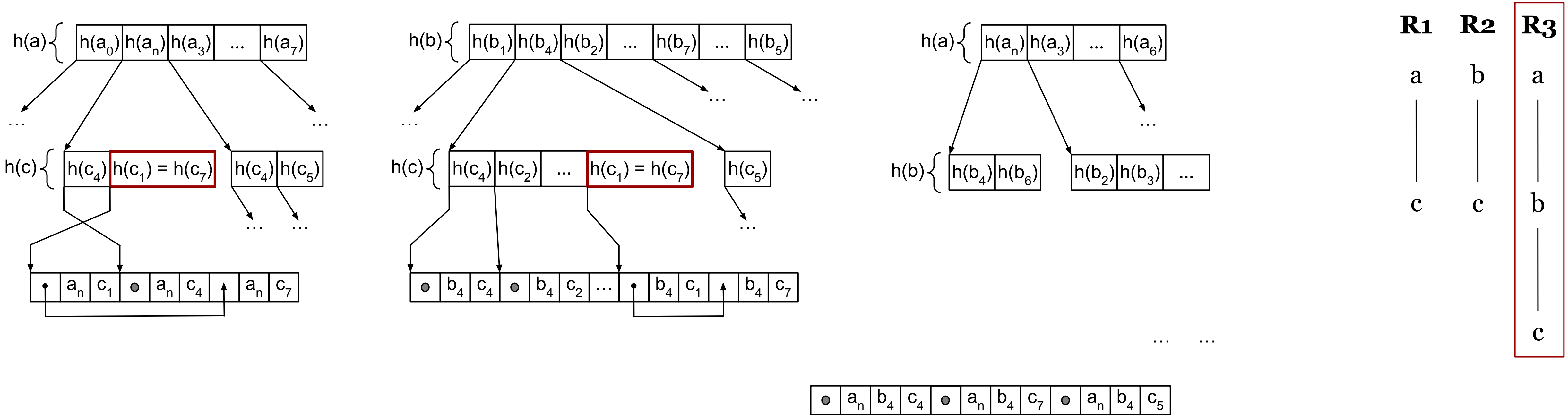
Build Hash Tries



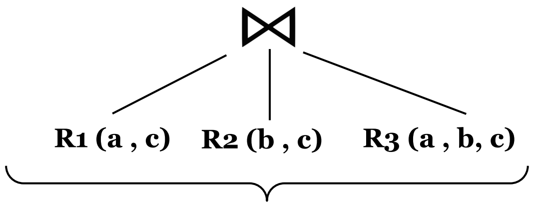
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

R1 (a, c) **R2 (b, c)** **R3 (a, b, c)**



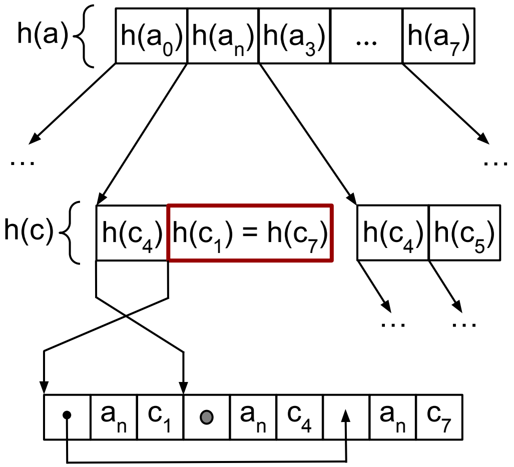
Build Hash Tries



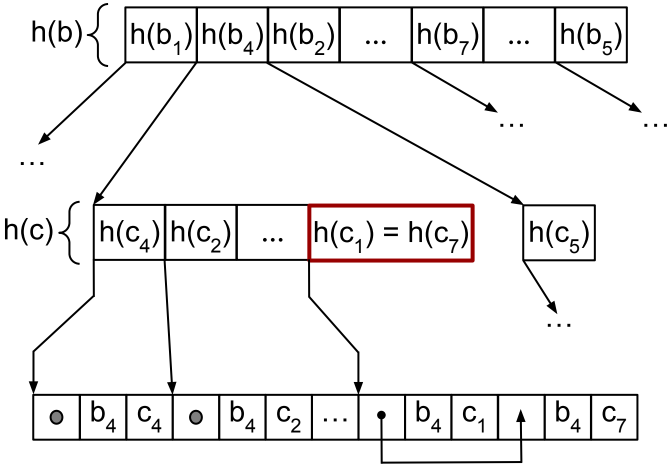
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]

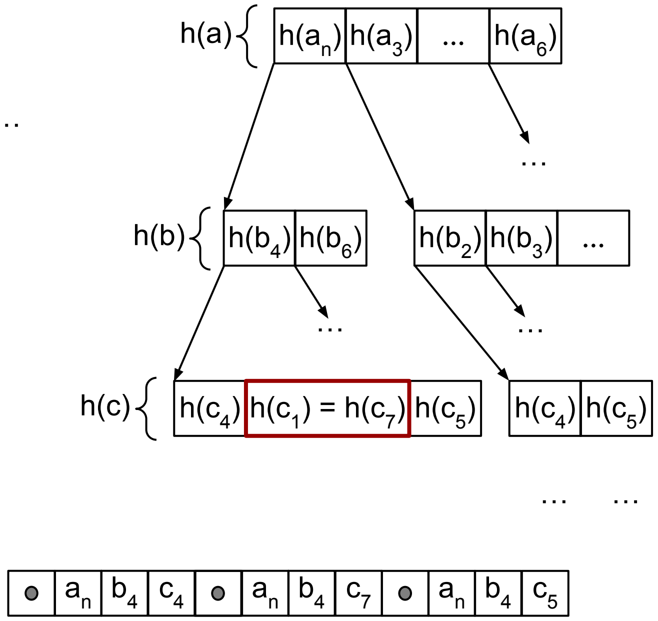
R1 (a, c)

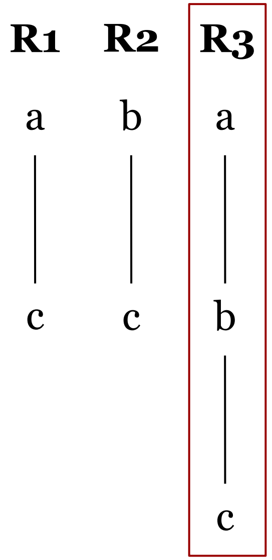


R2 (b, c)

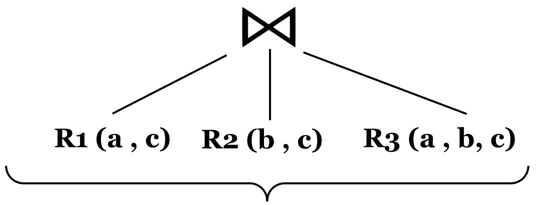


R3 (a, b, c)



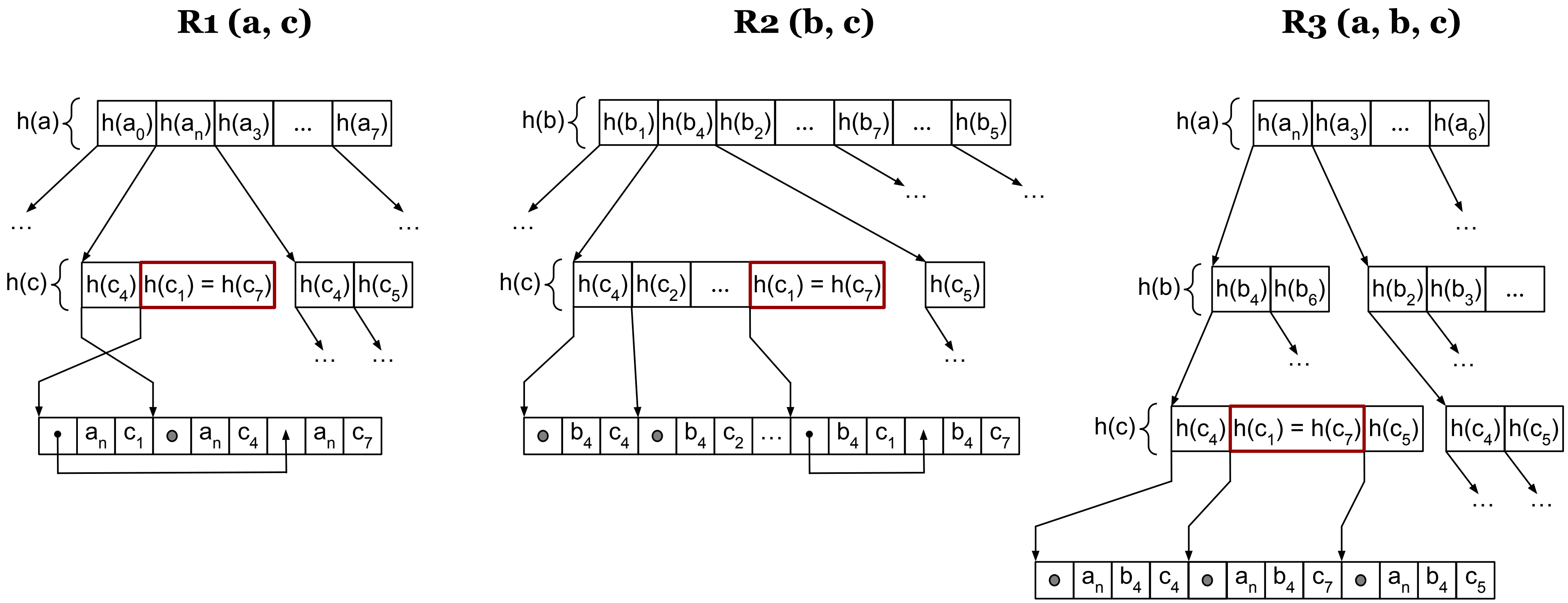


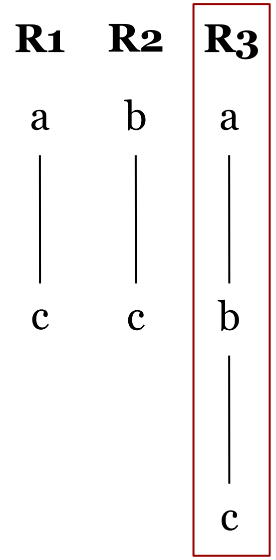
Build Hash Tries



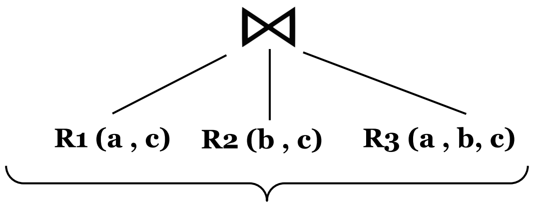
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]





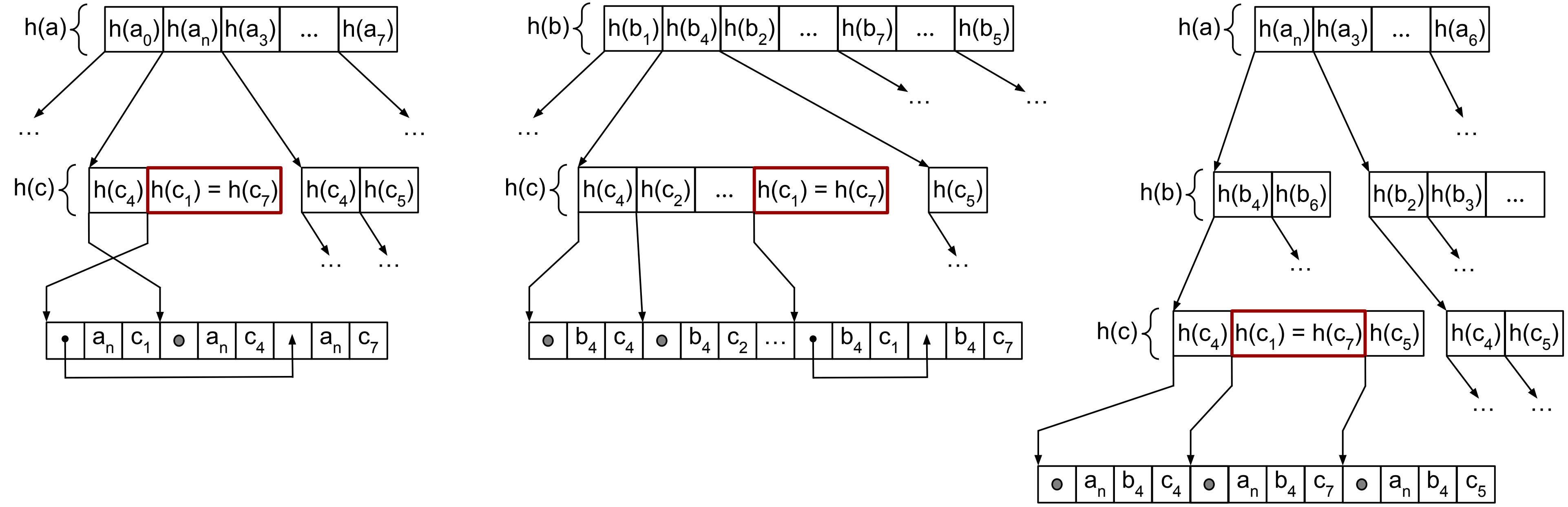
Probe - Enumeration Simulation

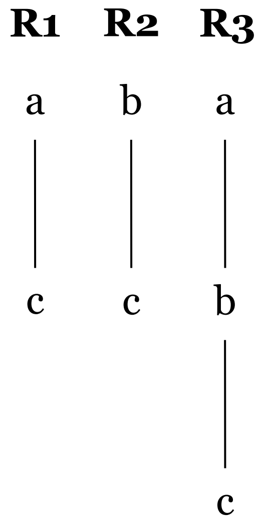


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

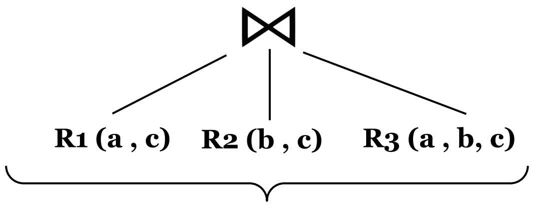
Attribute ordering: [a , b , c]

R1 (a, c) **R2 (b, c)** **R3 (a, b, c)**



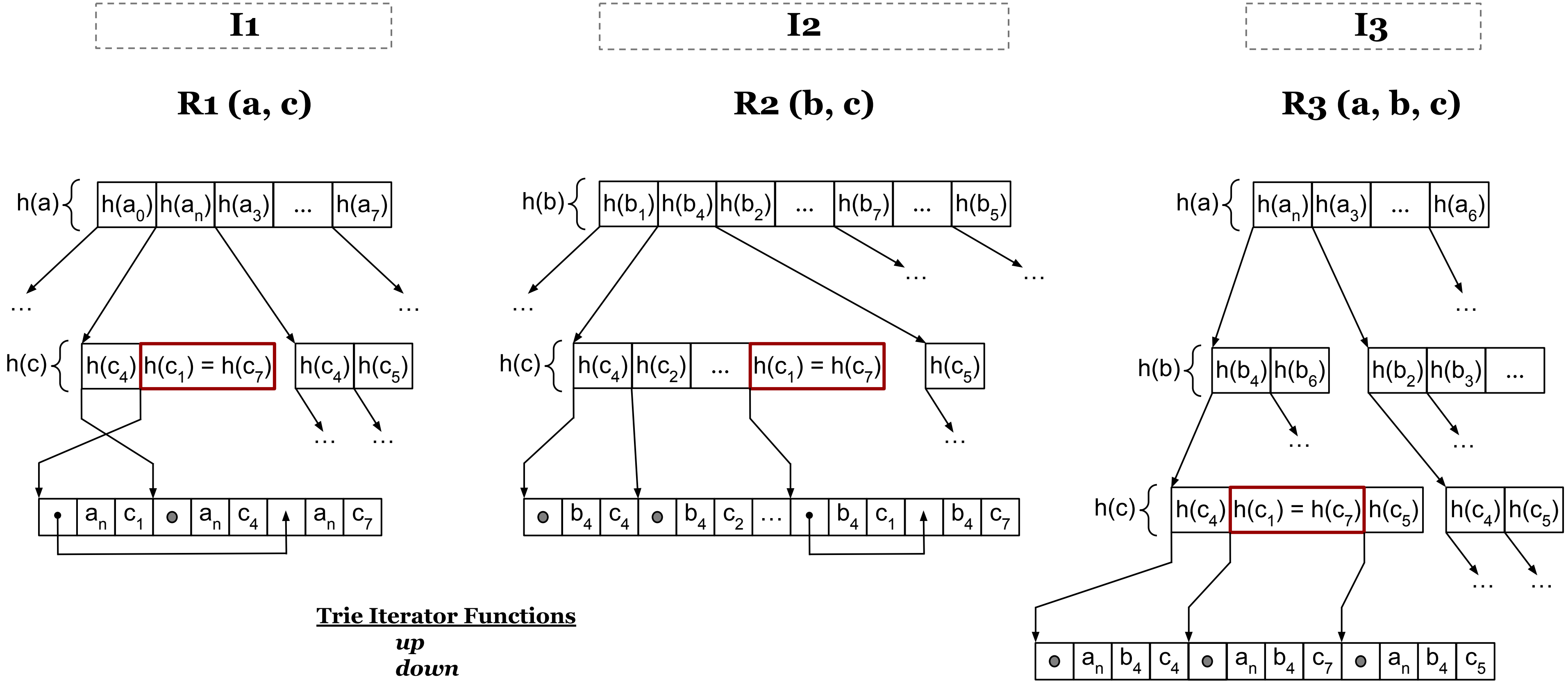


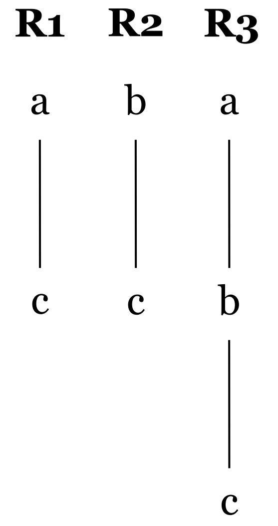
Probe - Enumeration Simulation



Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]



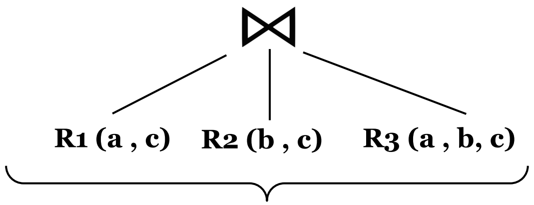


Trie Iterator Functions

up
down
next
lookup

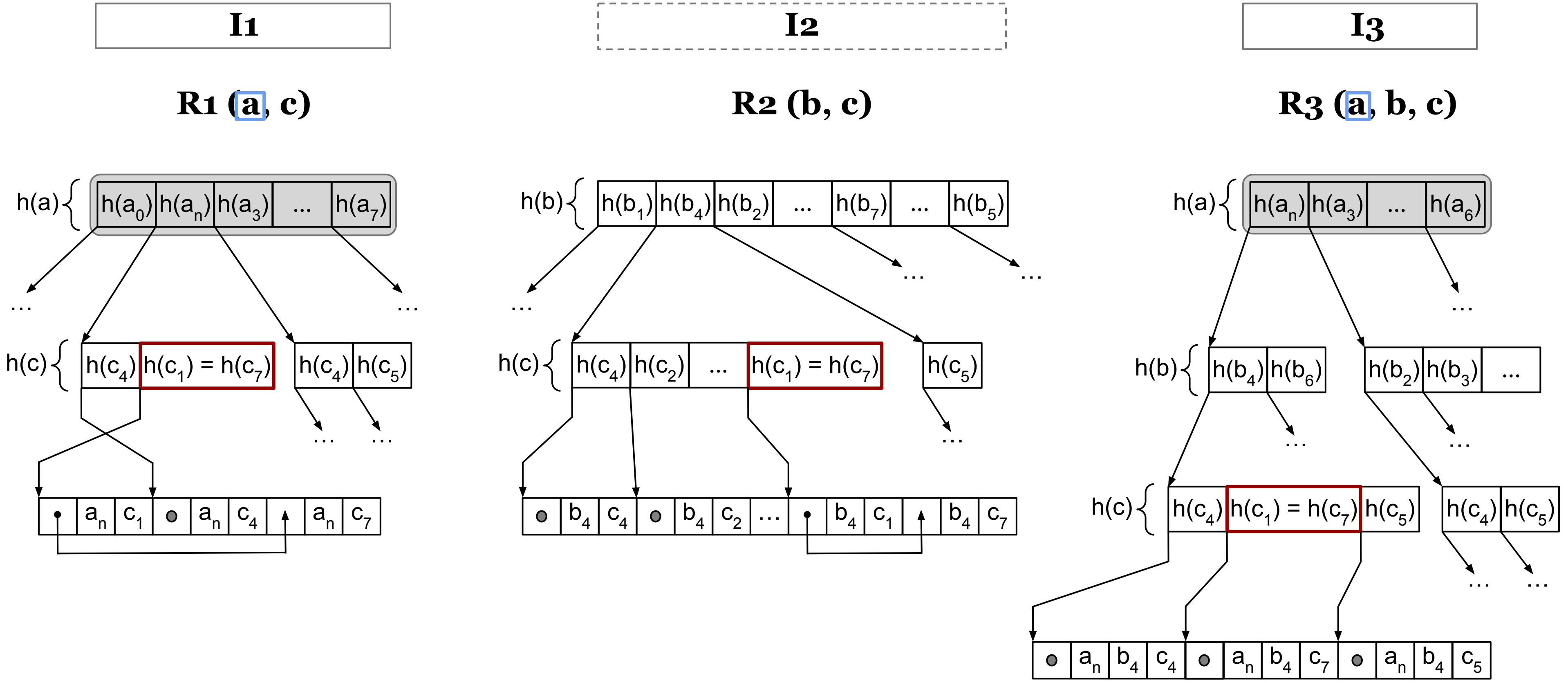
hash
size
tuples

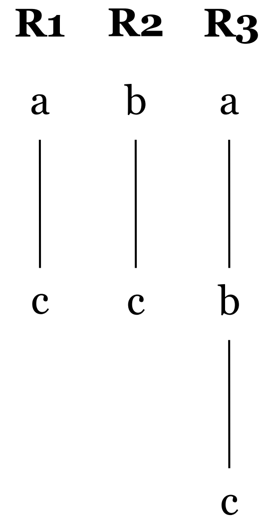
Probe - Enumeration Simulation



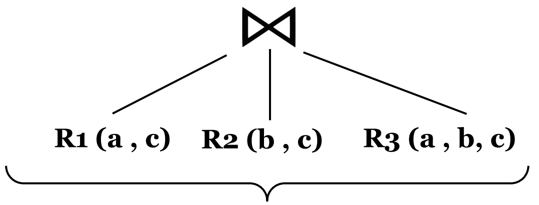
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]



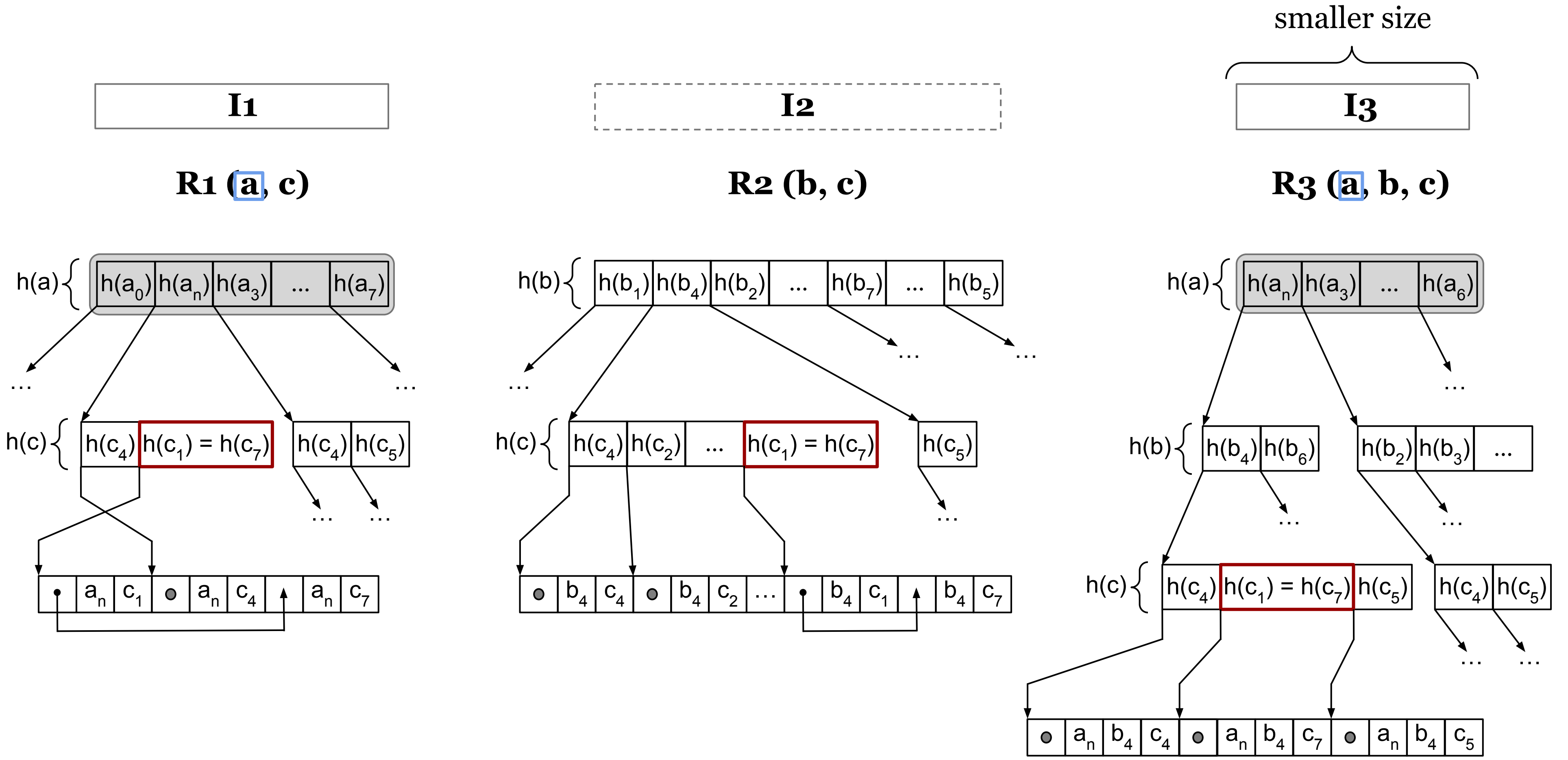


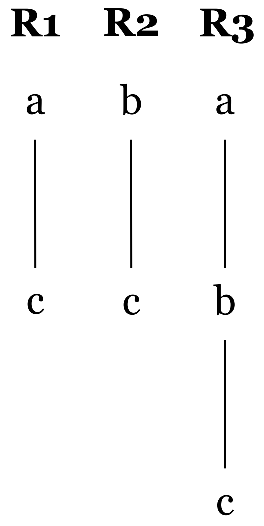
Probe - Enumeration Simulation



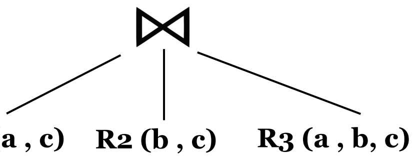
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]



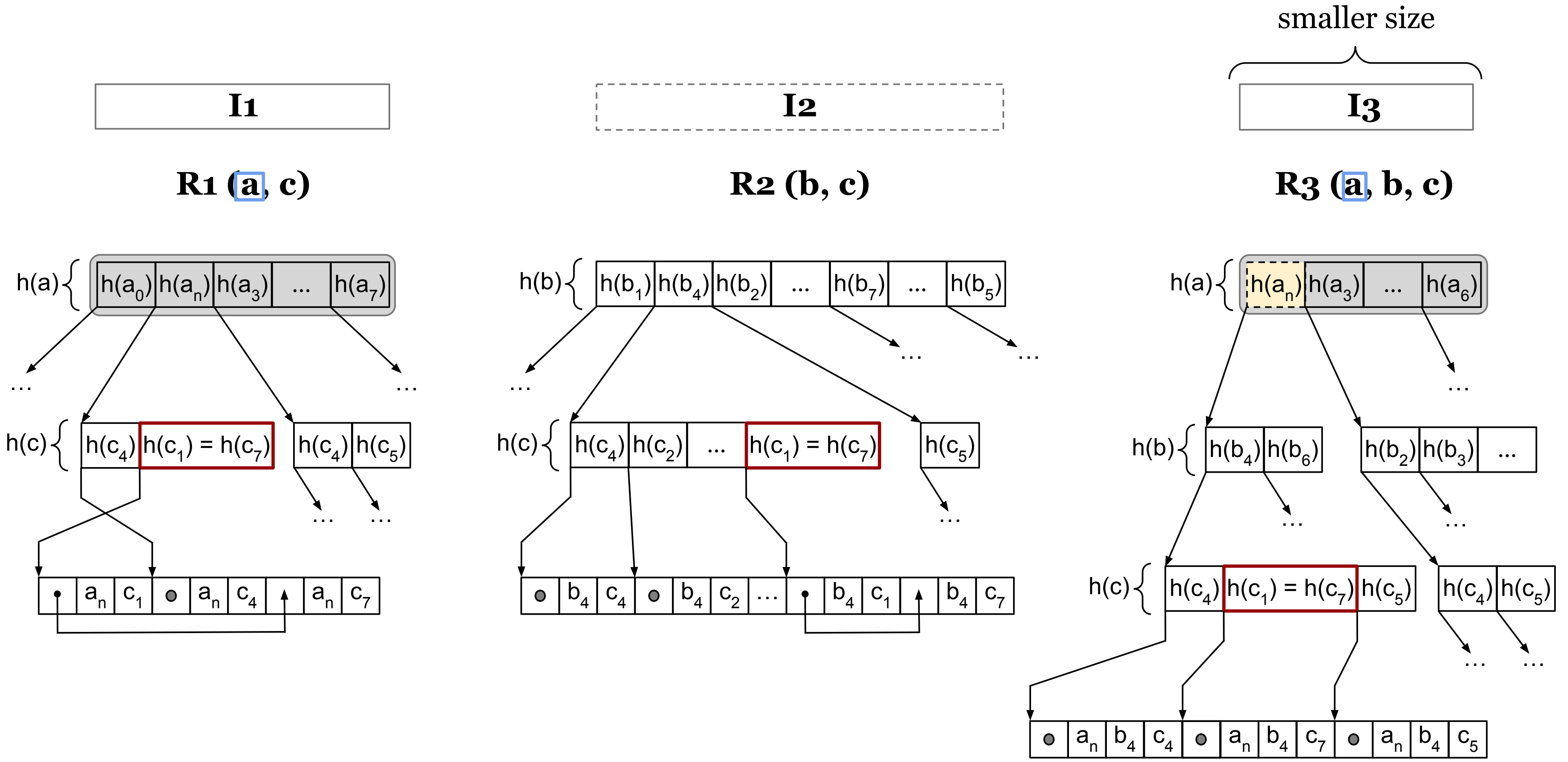


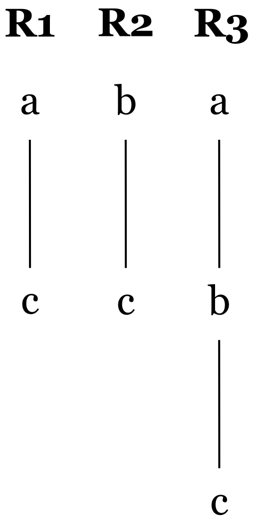
Probe - Enumeration Simulation



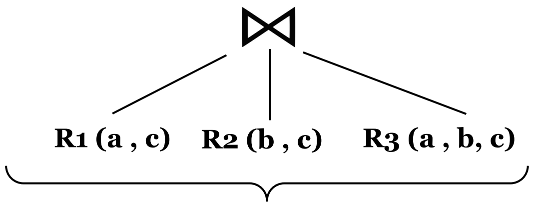
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]



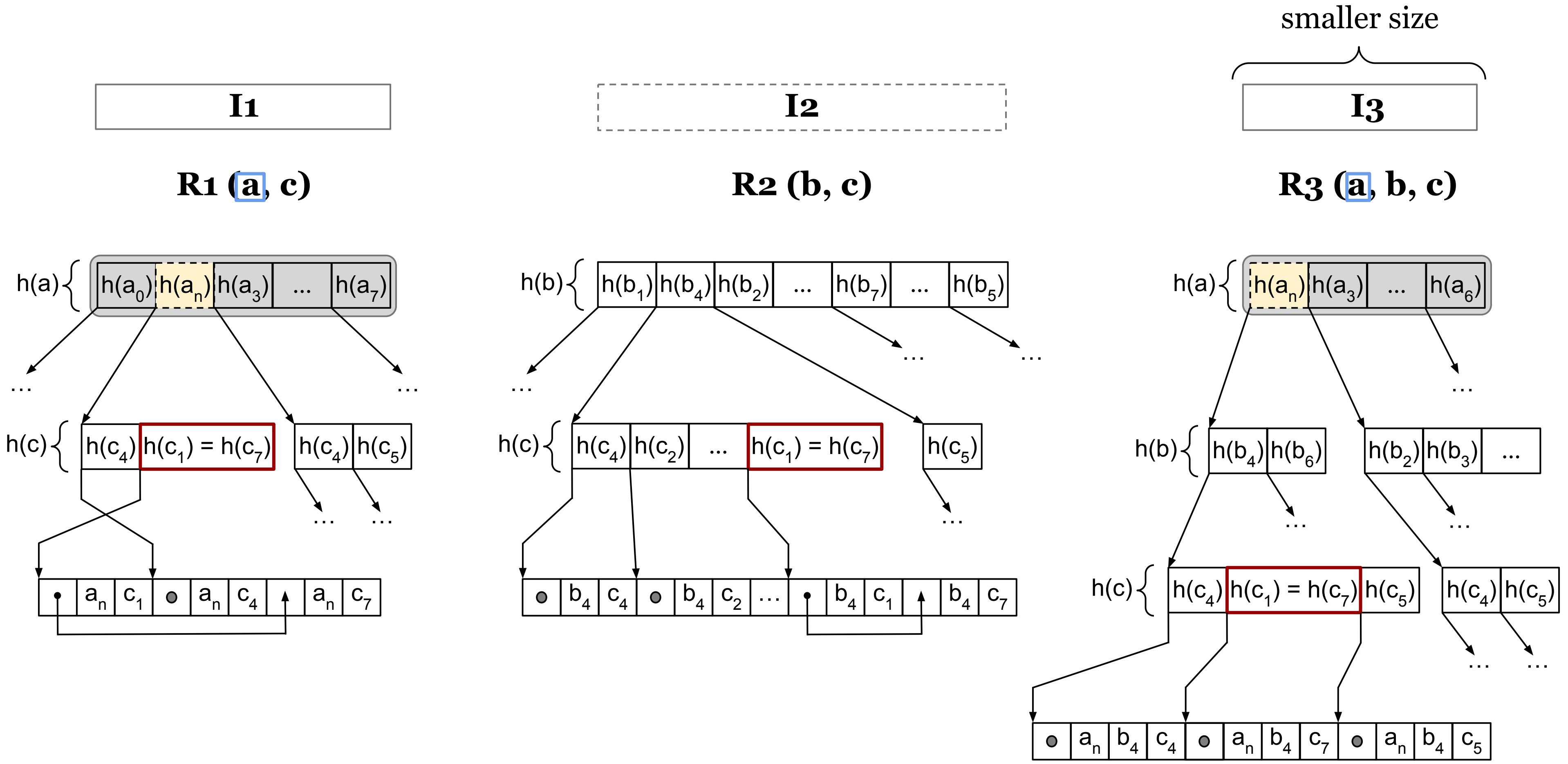


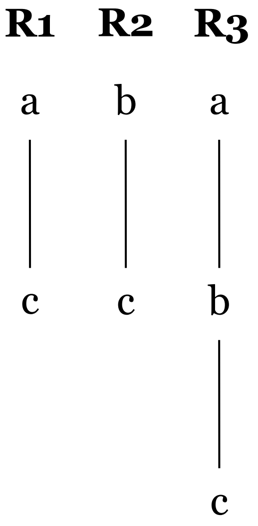
Probe - Enumeration Simulation



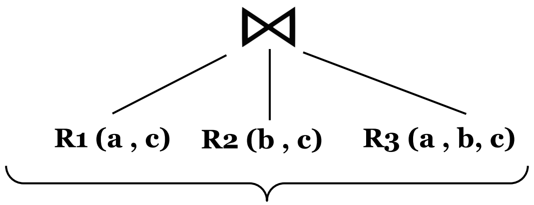
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a, b, c]



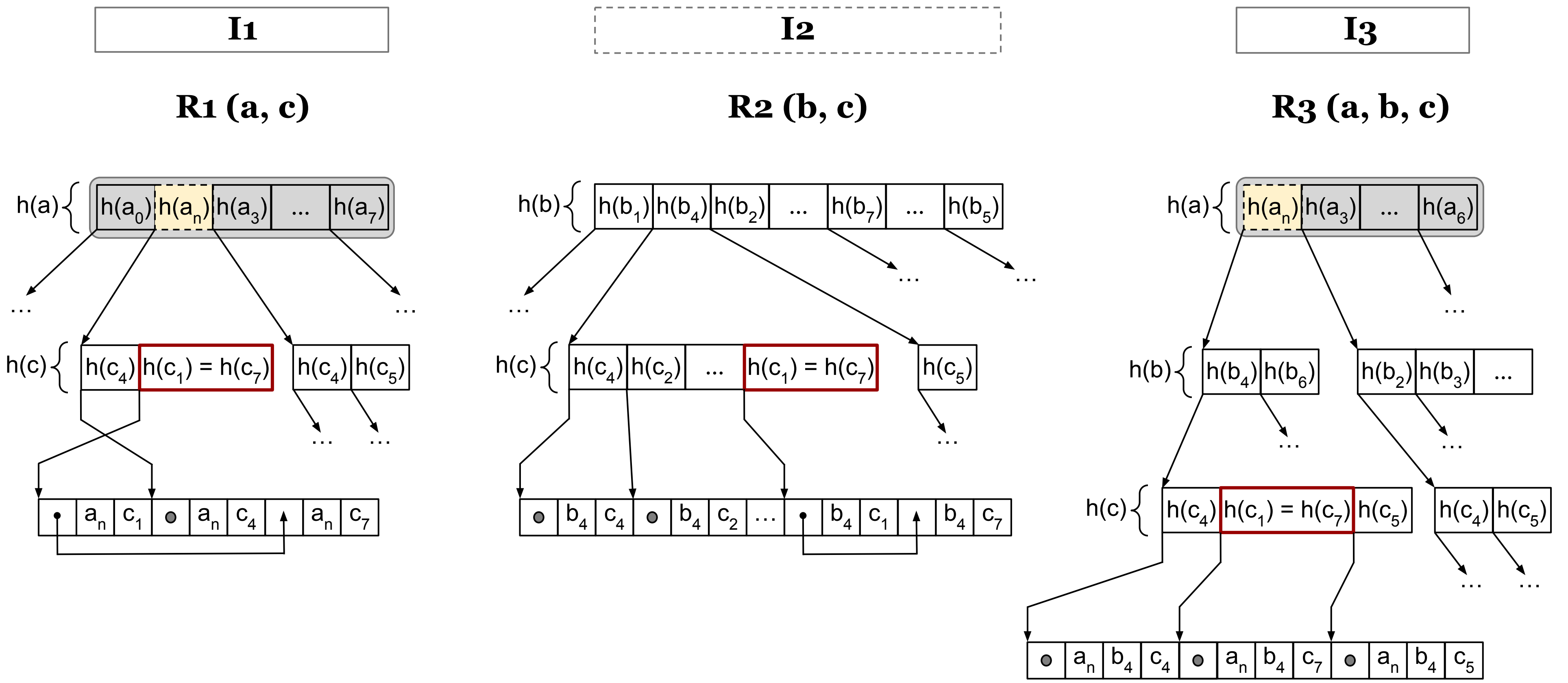


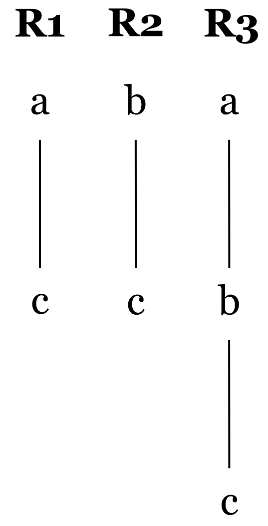
Probe - Enumeration Simulation



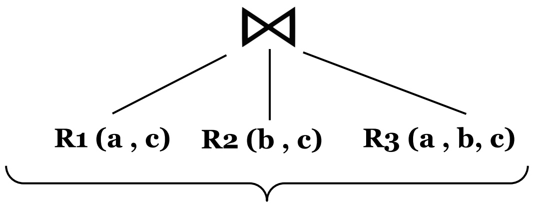
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]



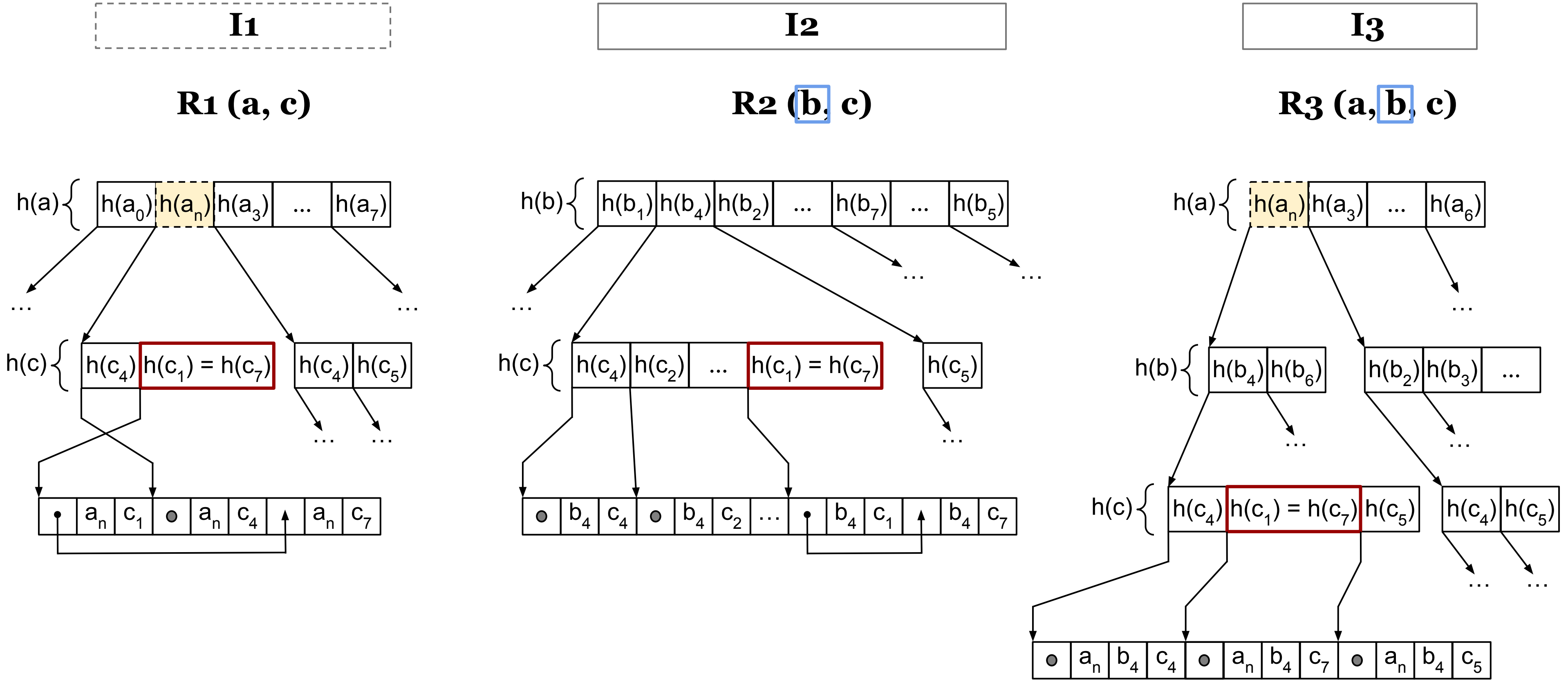


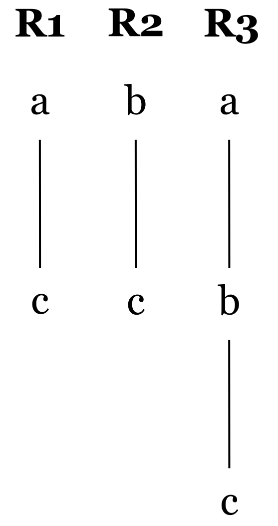
Probe - Enumeration Simulation



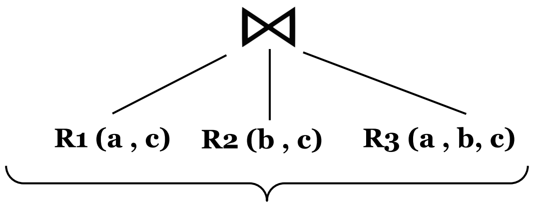
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b, c]



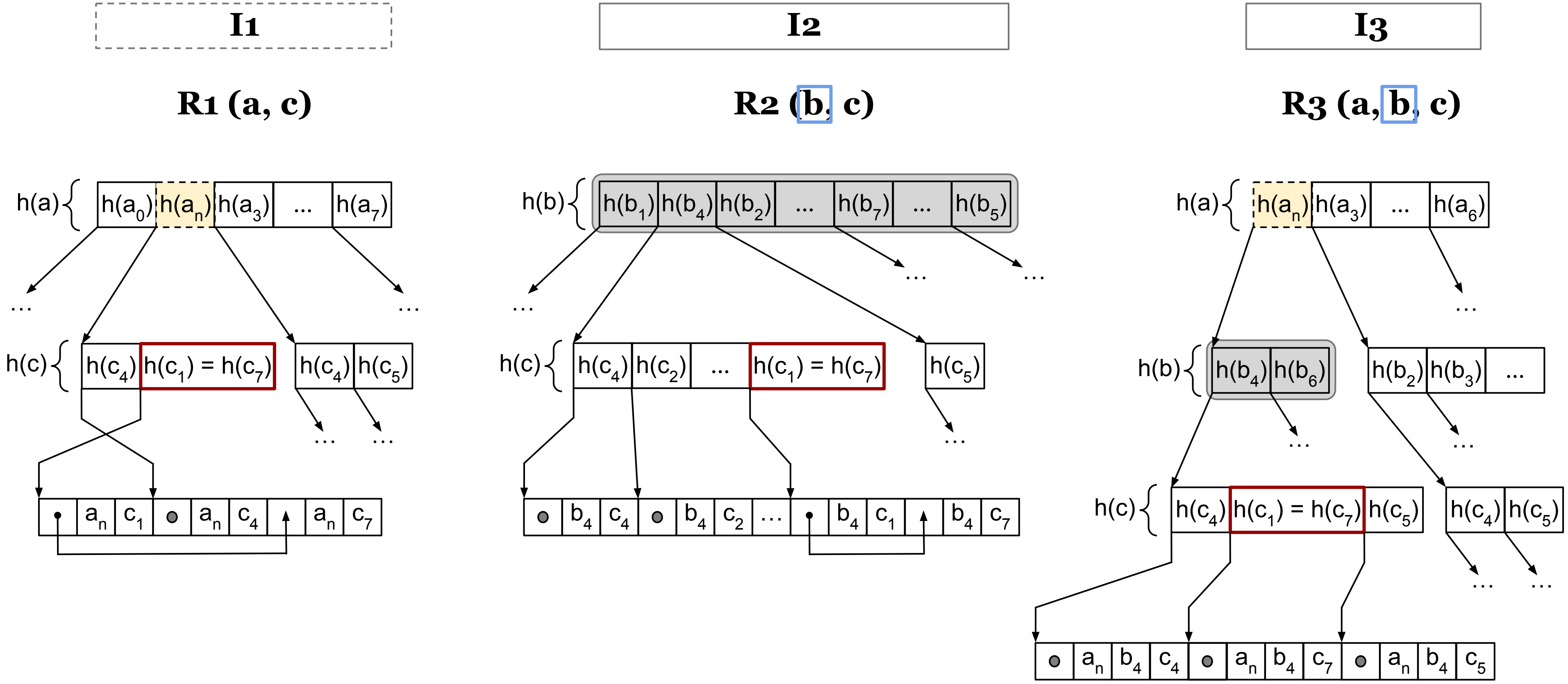


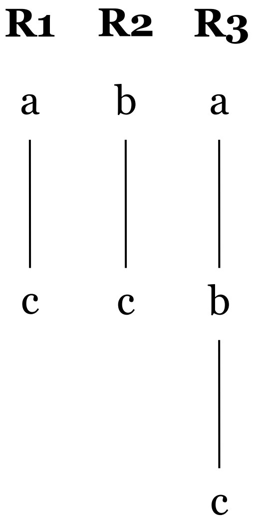
Probe - Enumeration Simulation



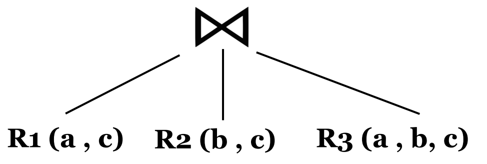
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b, c]



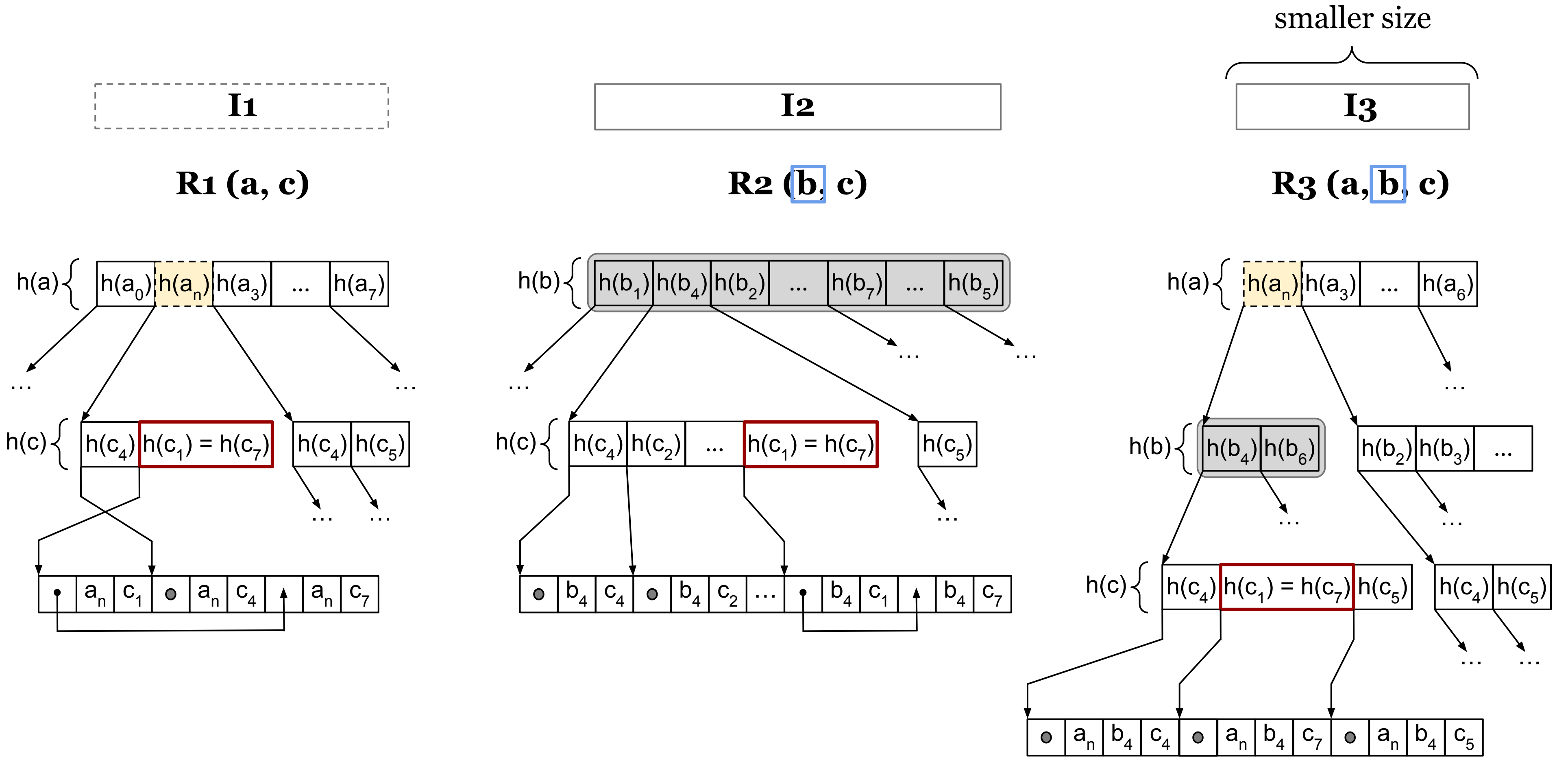


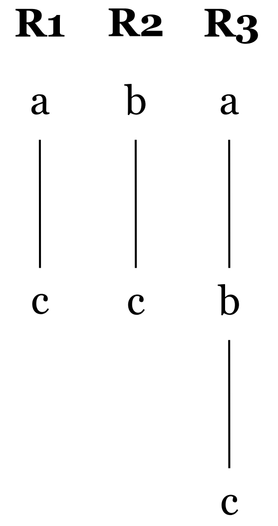
Probe - Enumeration Simulation



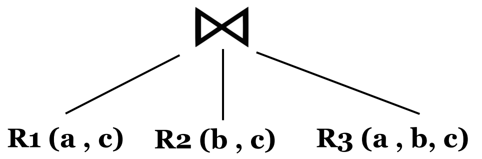
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b, c]



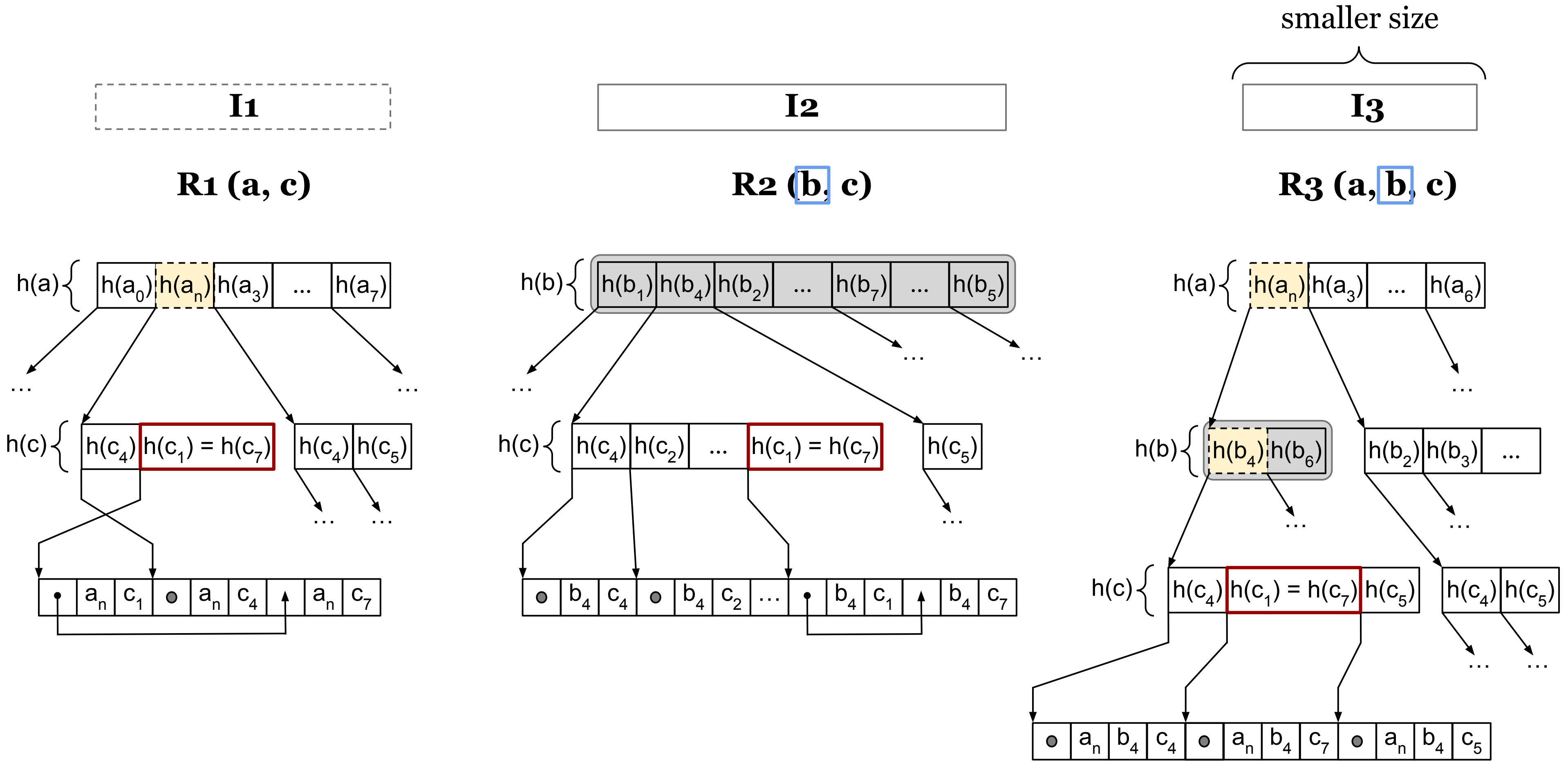


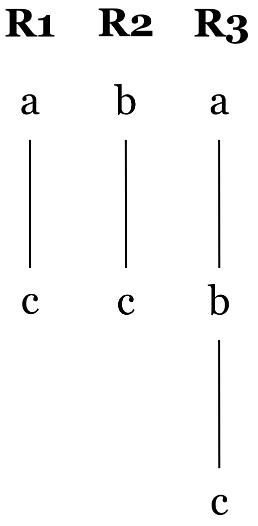
Probe - Enumeration Simulation



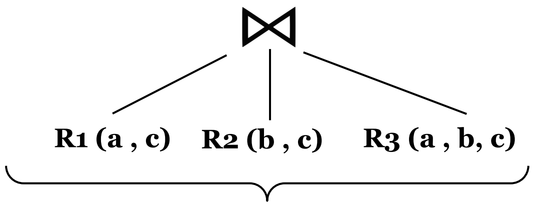
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b, c]



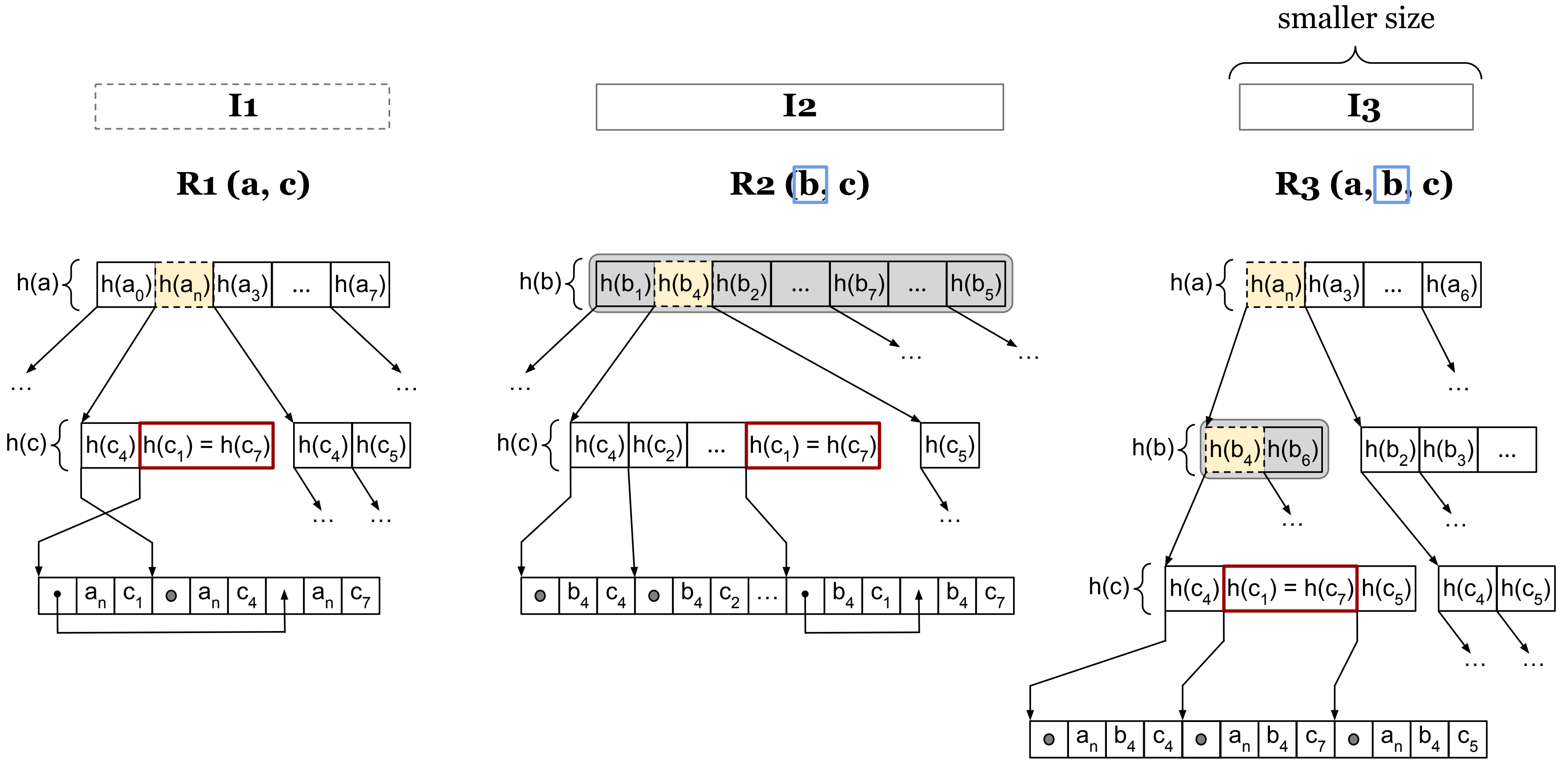


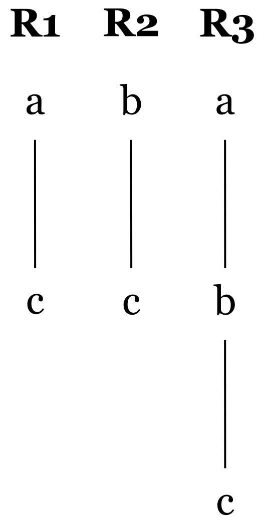
Probe - Enumeration Simulation



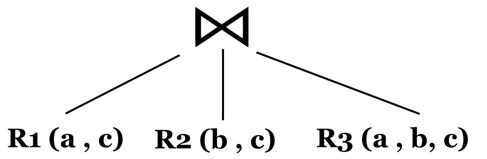
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b, c]





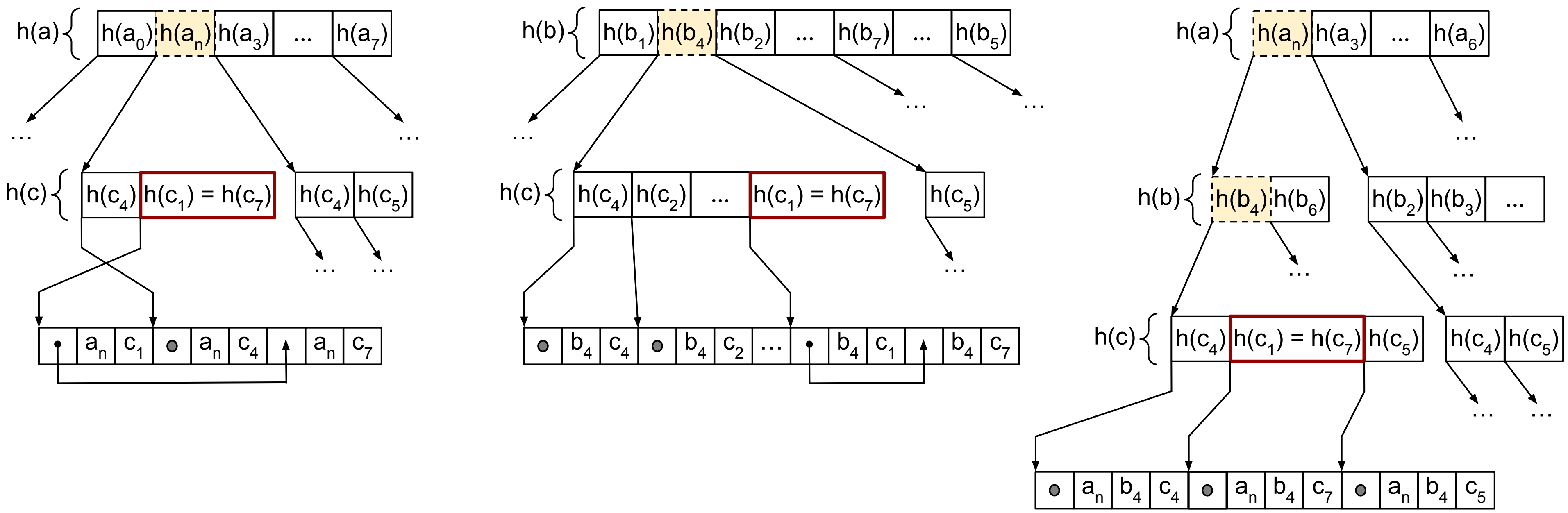
Probe - Enumeration Simulation

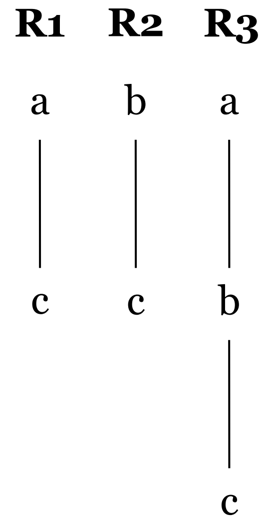


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

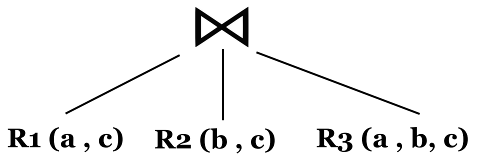
Attribute ordering: [a , b , c]

I1	I2	I3
R1 (a, c)	R2 (b, c)	R3 (a, b, c)



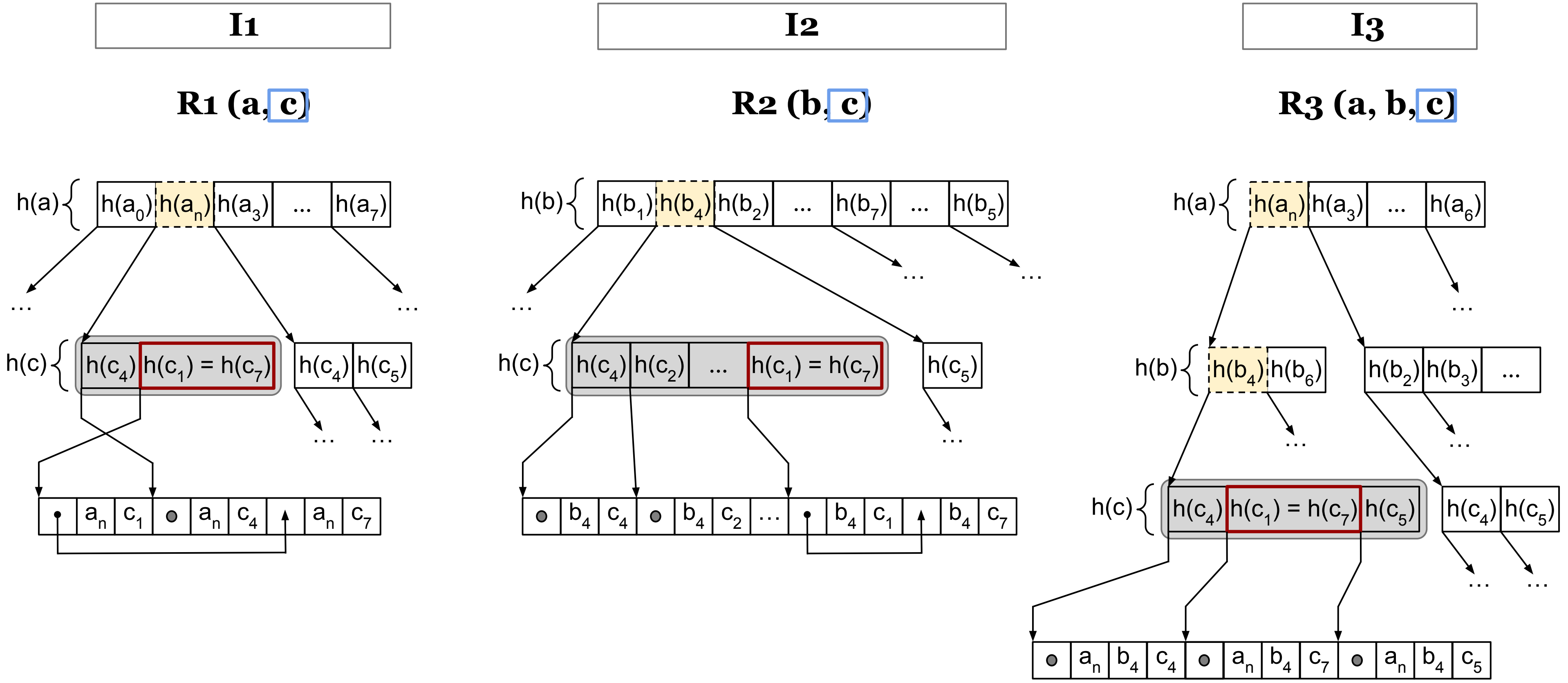


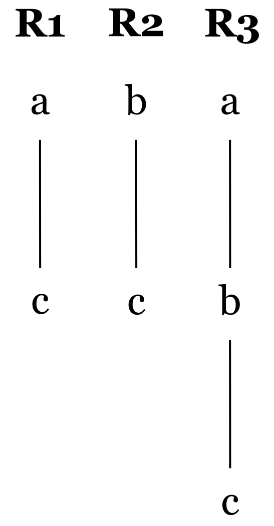
Probe - Enumeration Simulation



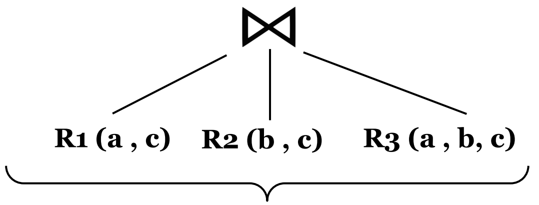
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]



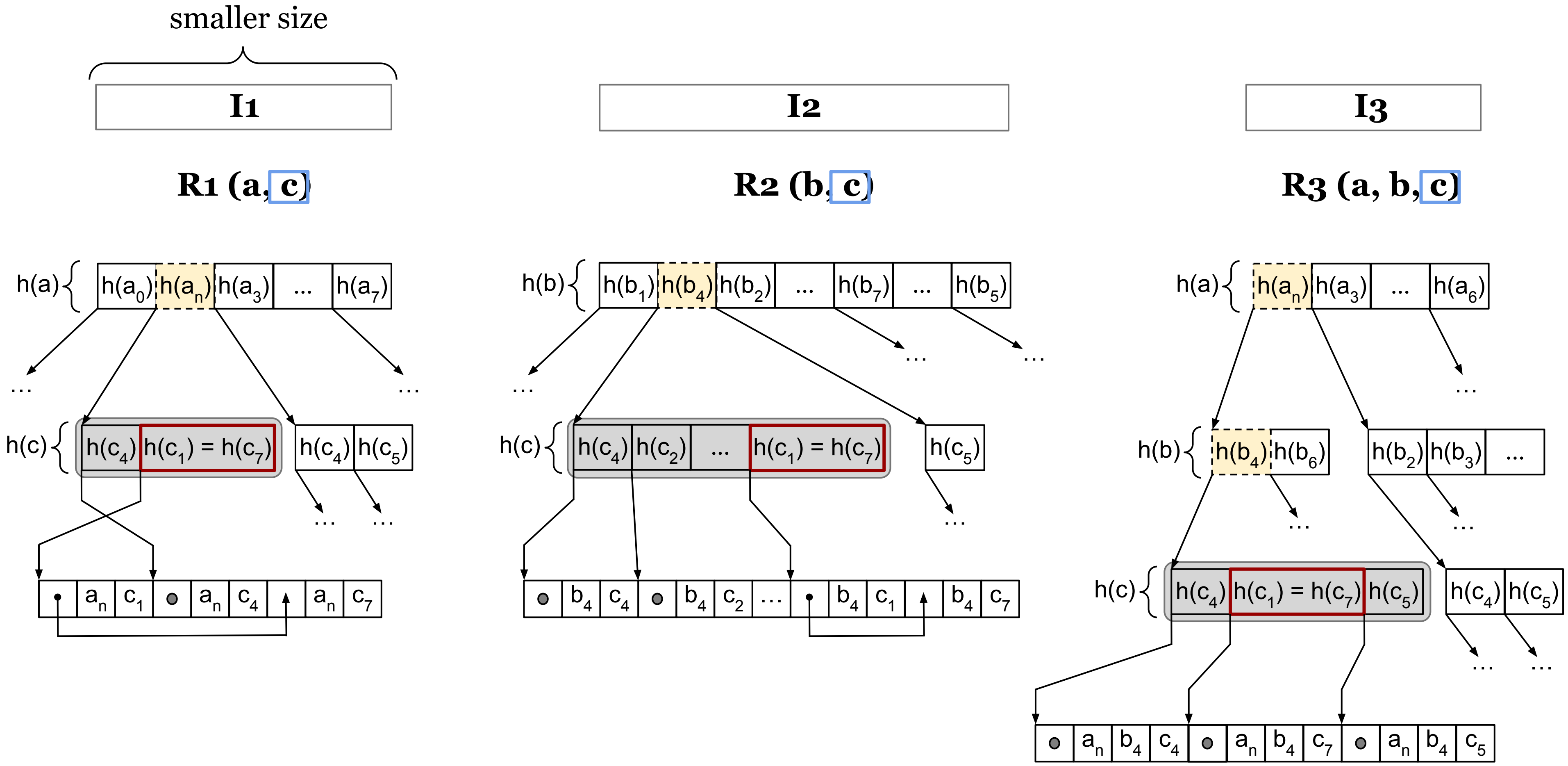


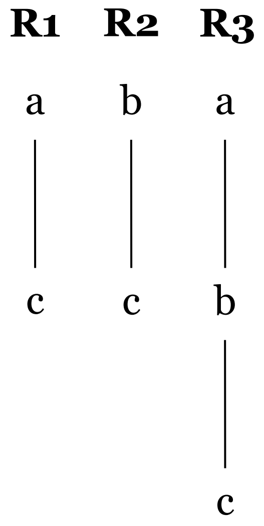
Probe - Enumeration Simulation



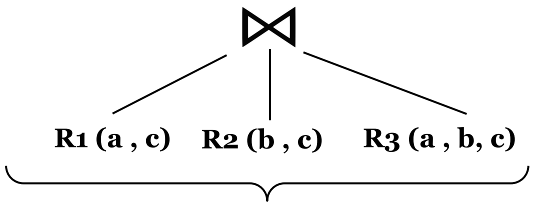
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]



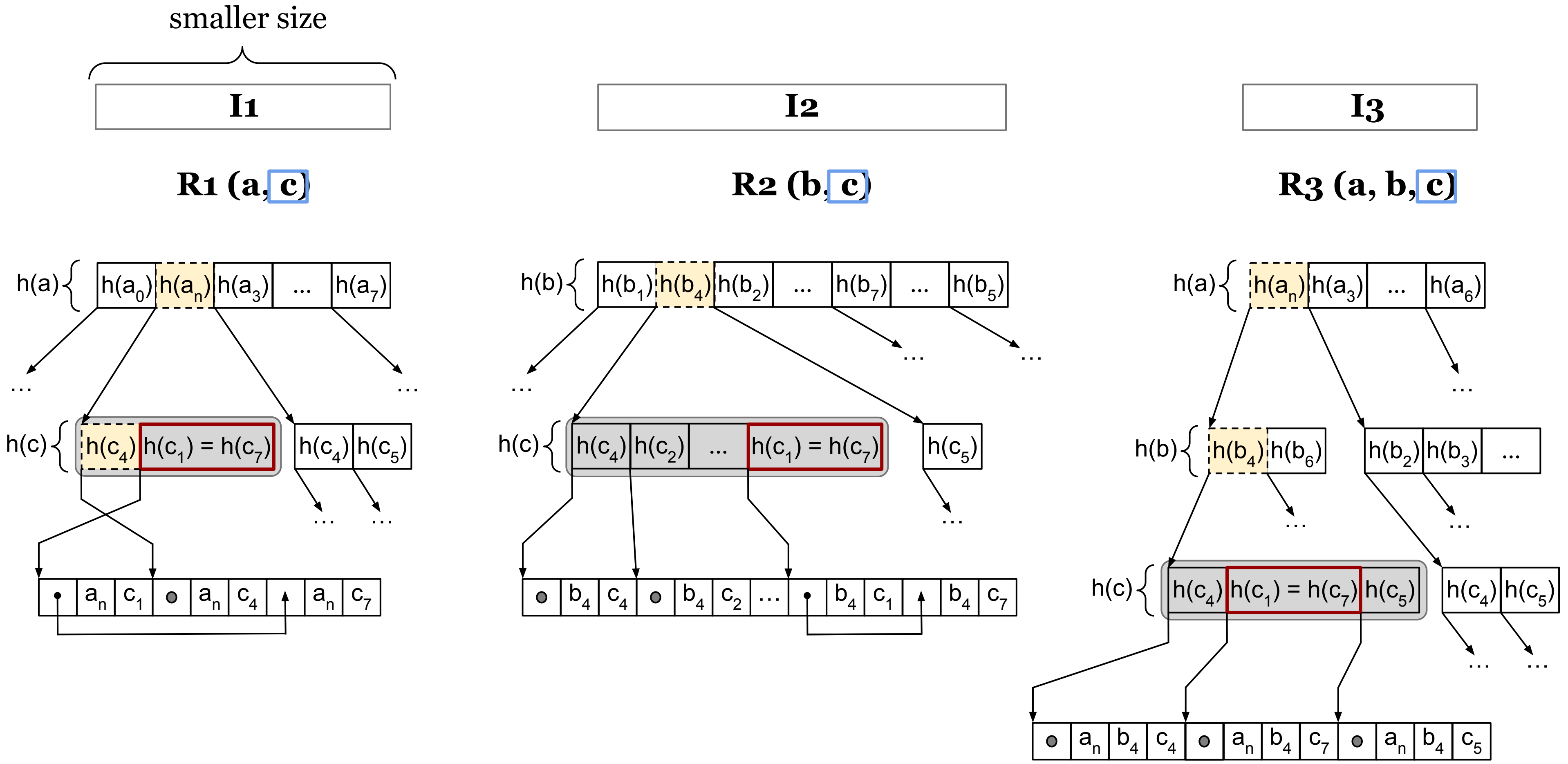


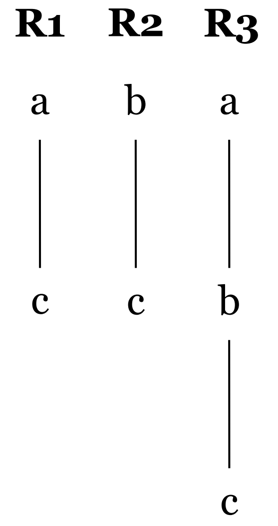
Probe - Enumeration Simulation



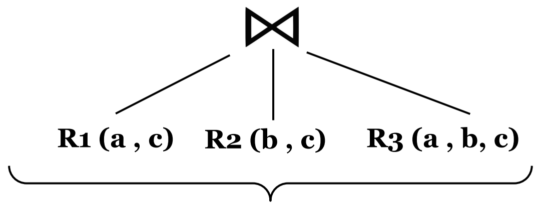
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]



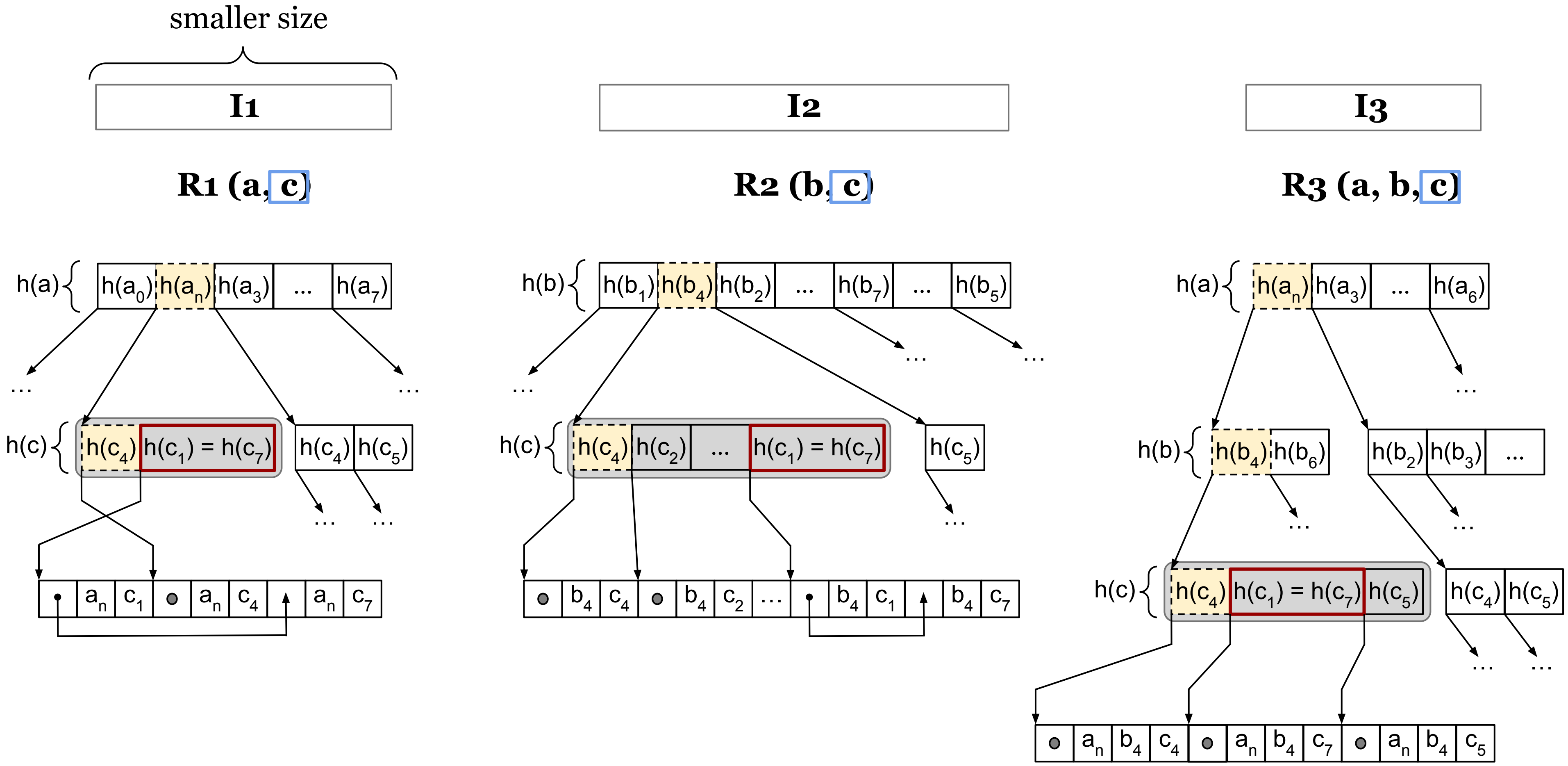


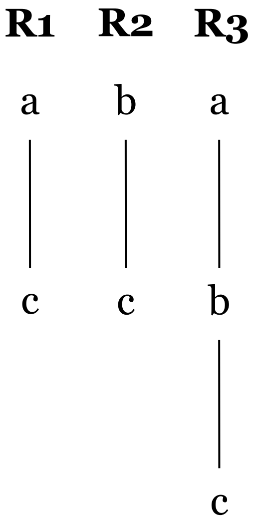
Probe - Enumeration Simulation



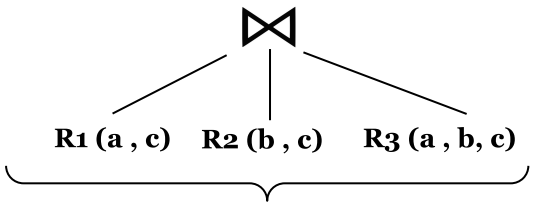
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]



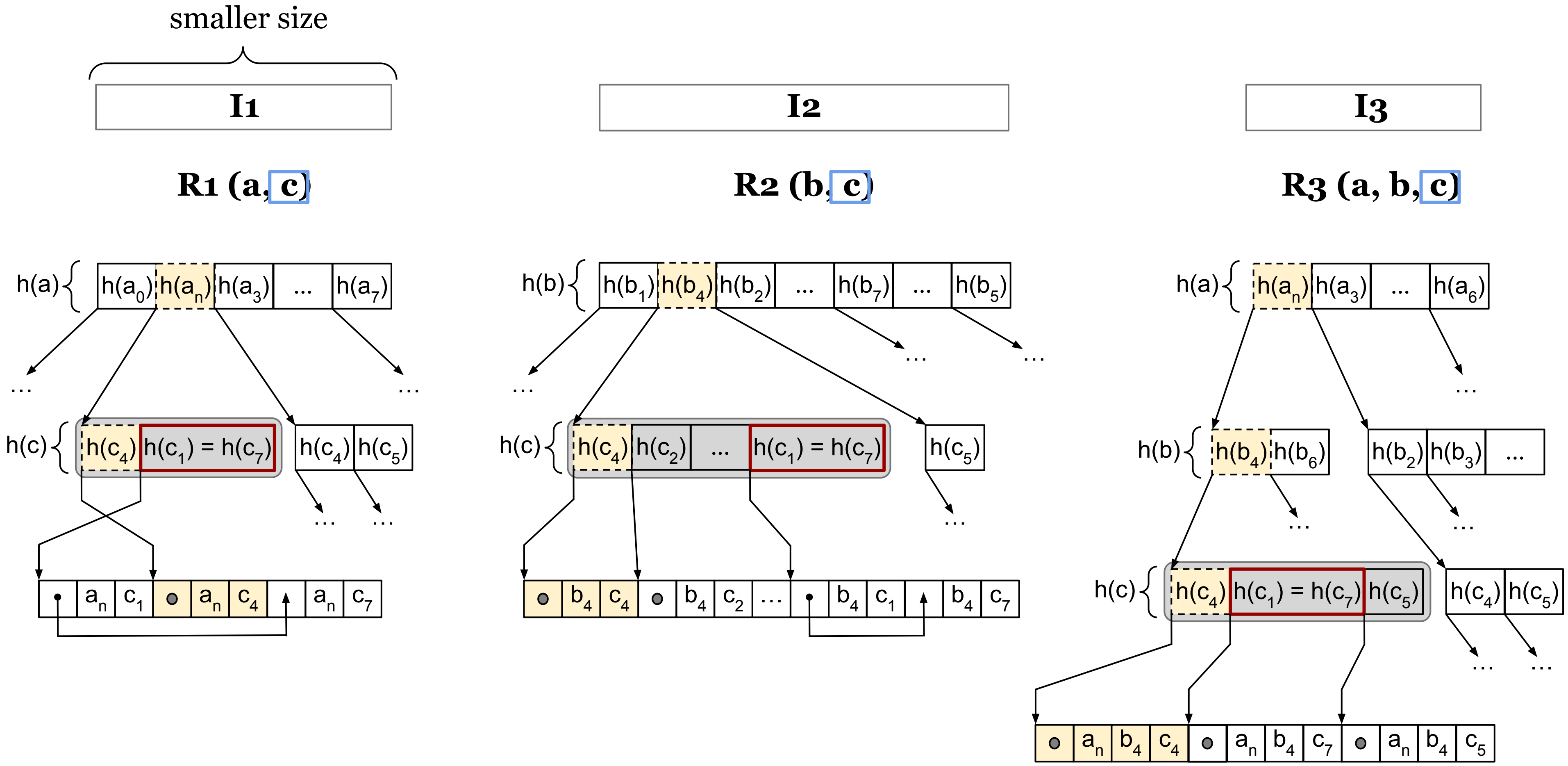


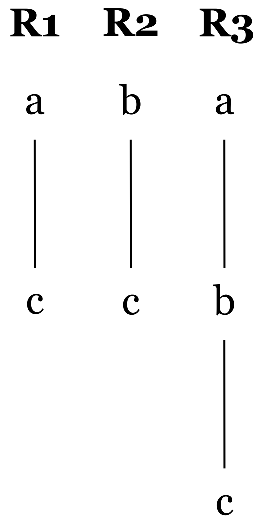
Probe - Enumeration Simulation



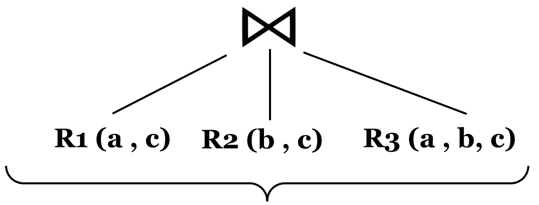
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]



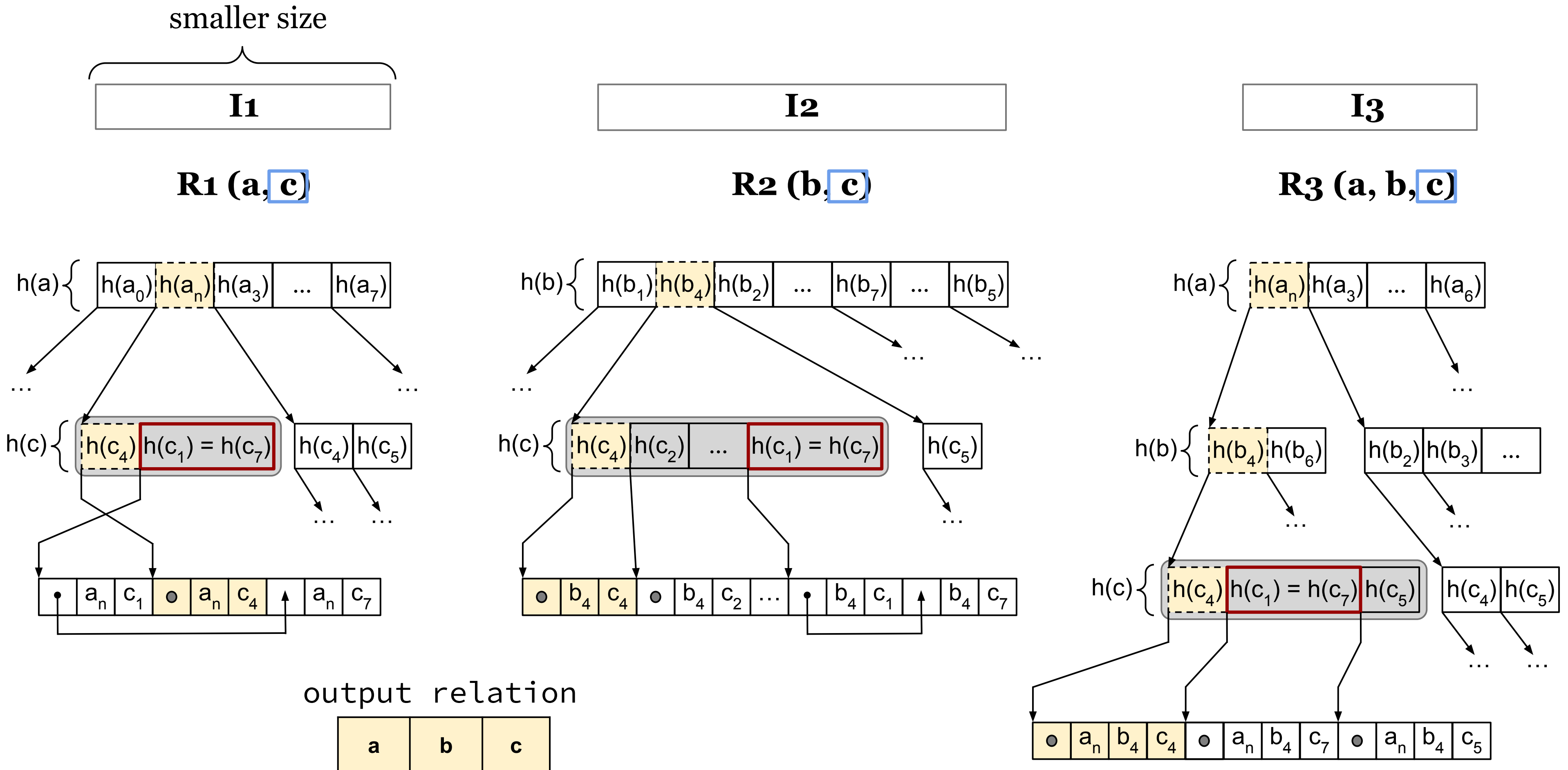


Probe - Enumeration Simulation



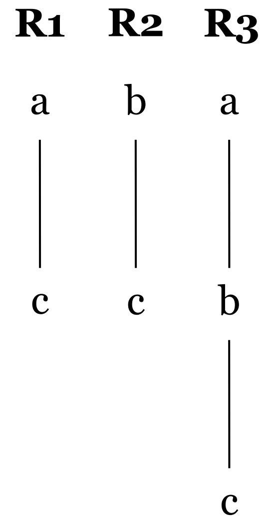
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]



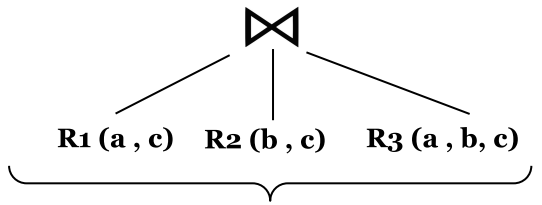
output relation

a	b	c
a_n	b_4	c_4



Probe - Enumeration Simulation

UMBRA



Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

smaller size

I1

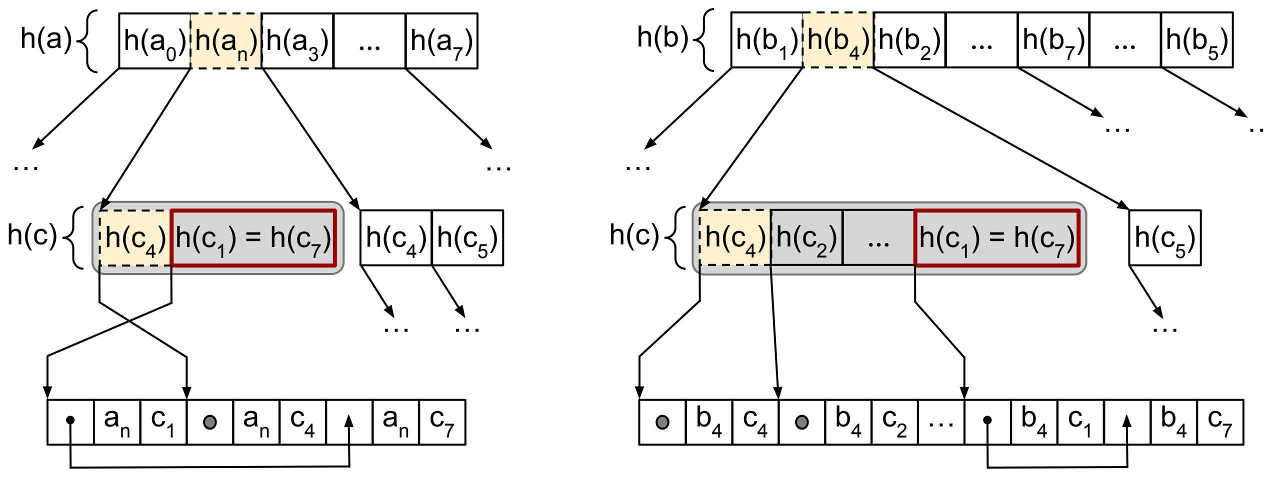
I2

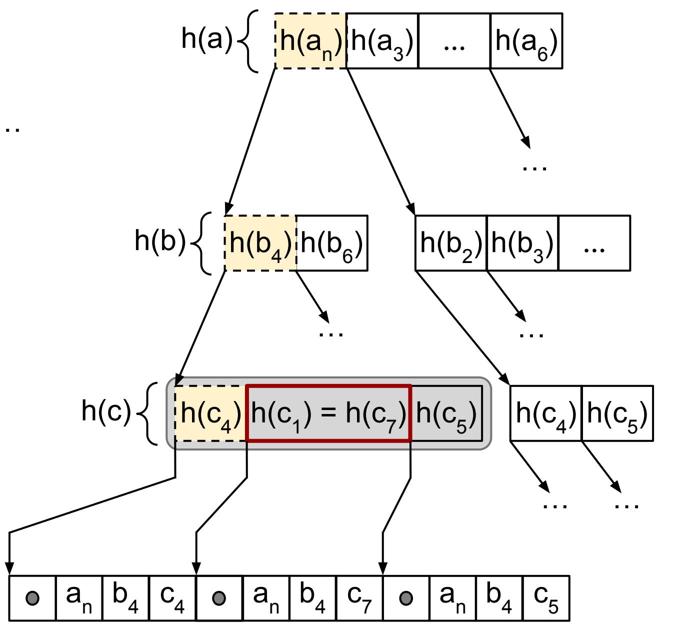
I3

R1 (a, c)

R2 (b, c)

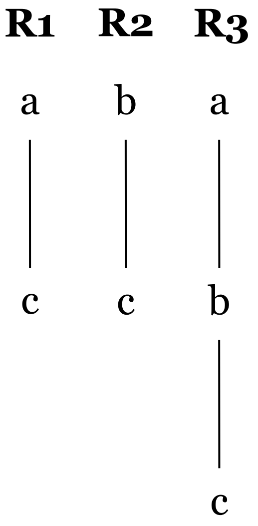
R3 (a, b, c)





output relation

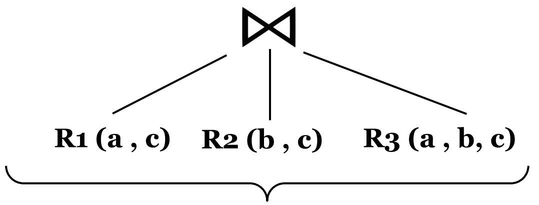
a	b	c
a_n	b_4	c_4



UNIVERSITY OF WATERLOO | DSG Data Systems Group

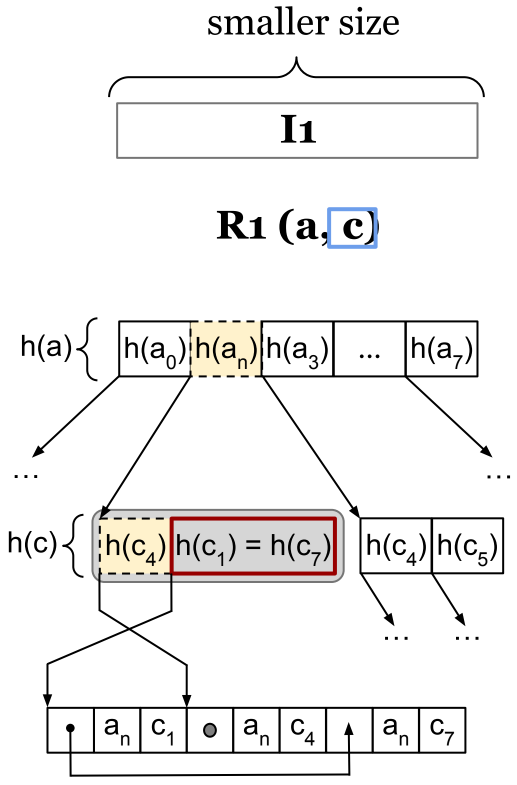
Probe - Enumeration Simulation

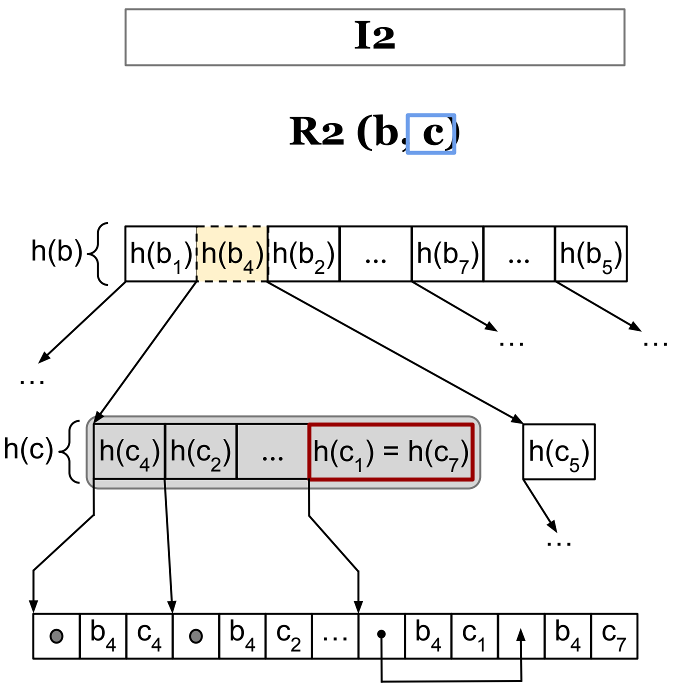
UMBRA

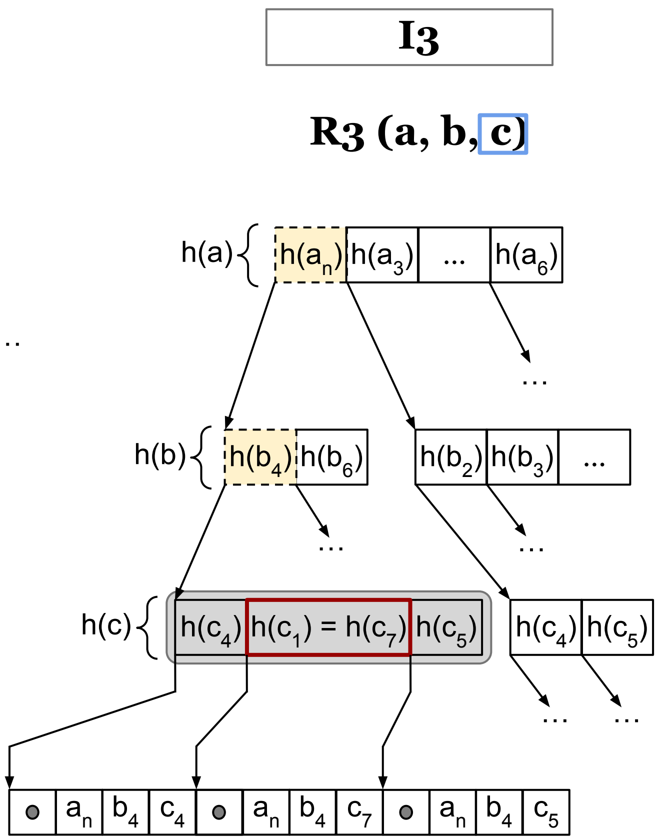


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

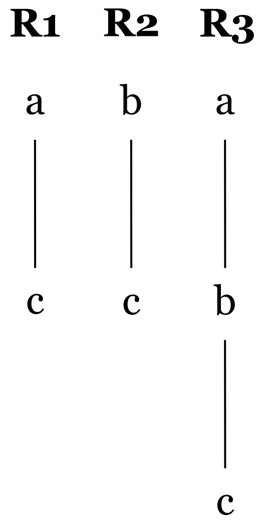




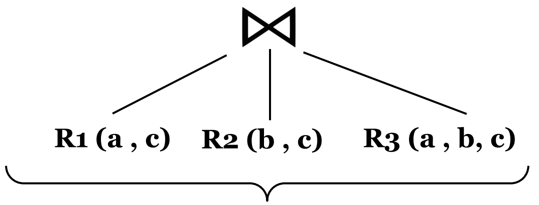


output relation

a	b	c
a_n	b_4	c_4

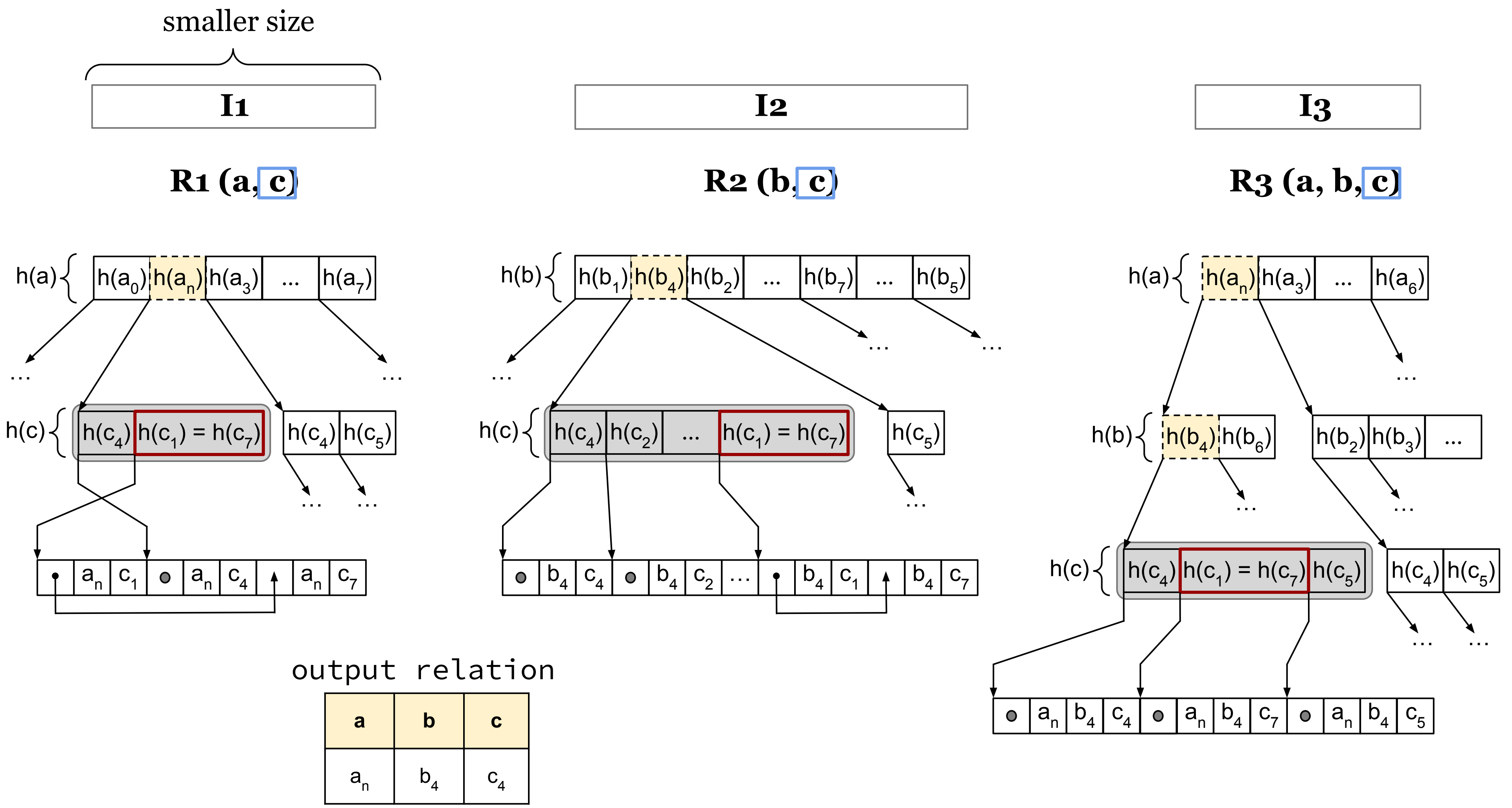


Probe - Enumeration Simulation



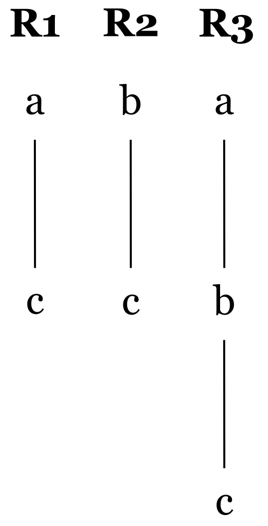
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

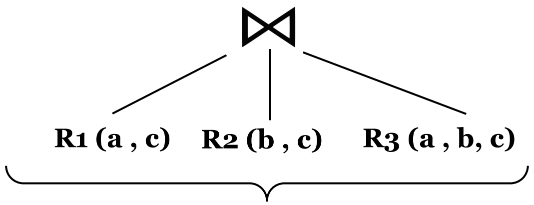


output relation

a	b	c
a$_n$	b$_4$	c$_4$

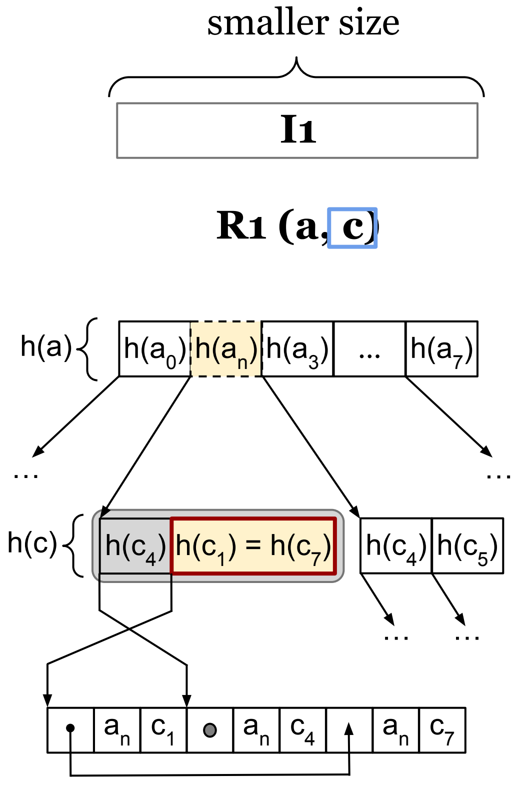


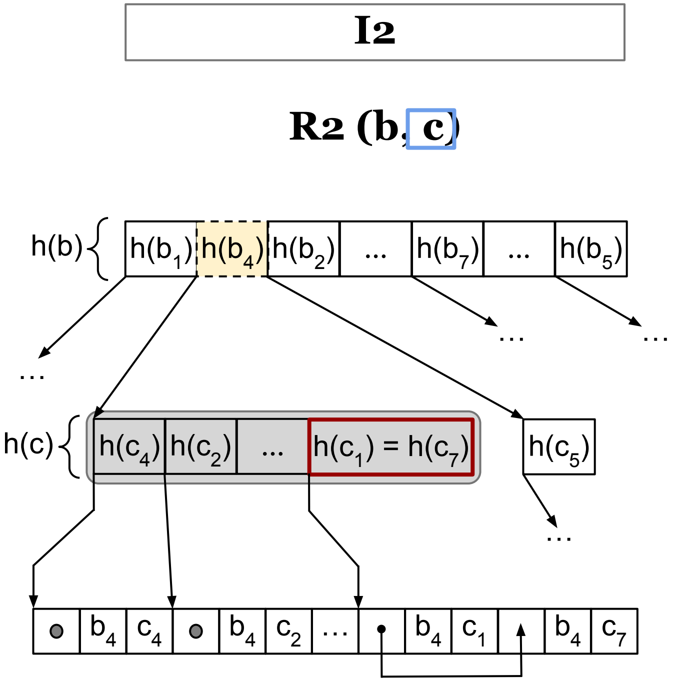
Probe - Enumeration Simulation

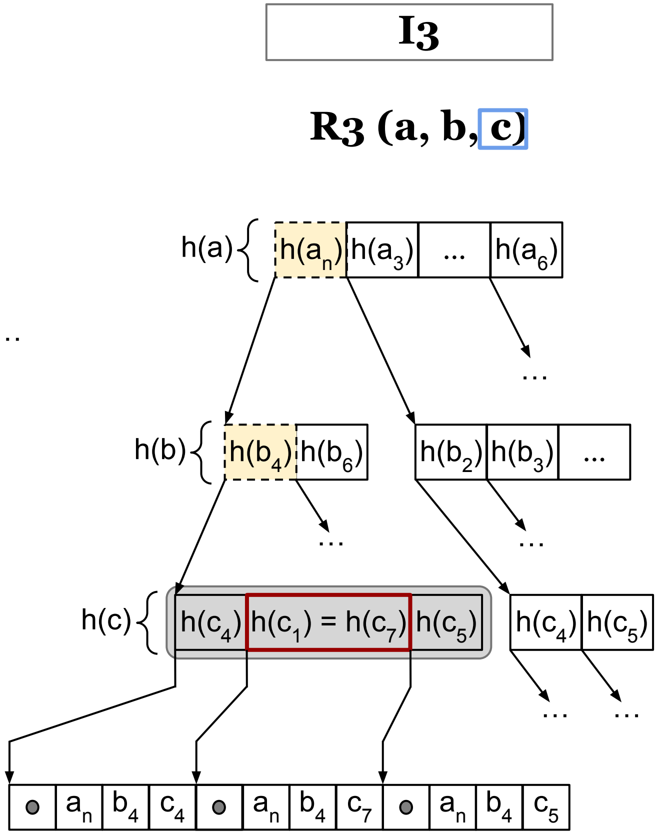


Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

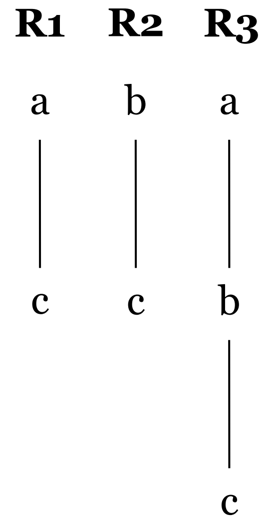






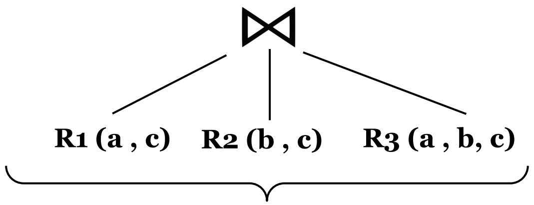
output relation

a	b	c
a_n	b_4	c_4



Probe - Enumeration Simulation

UMBRA



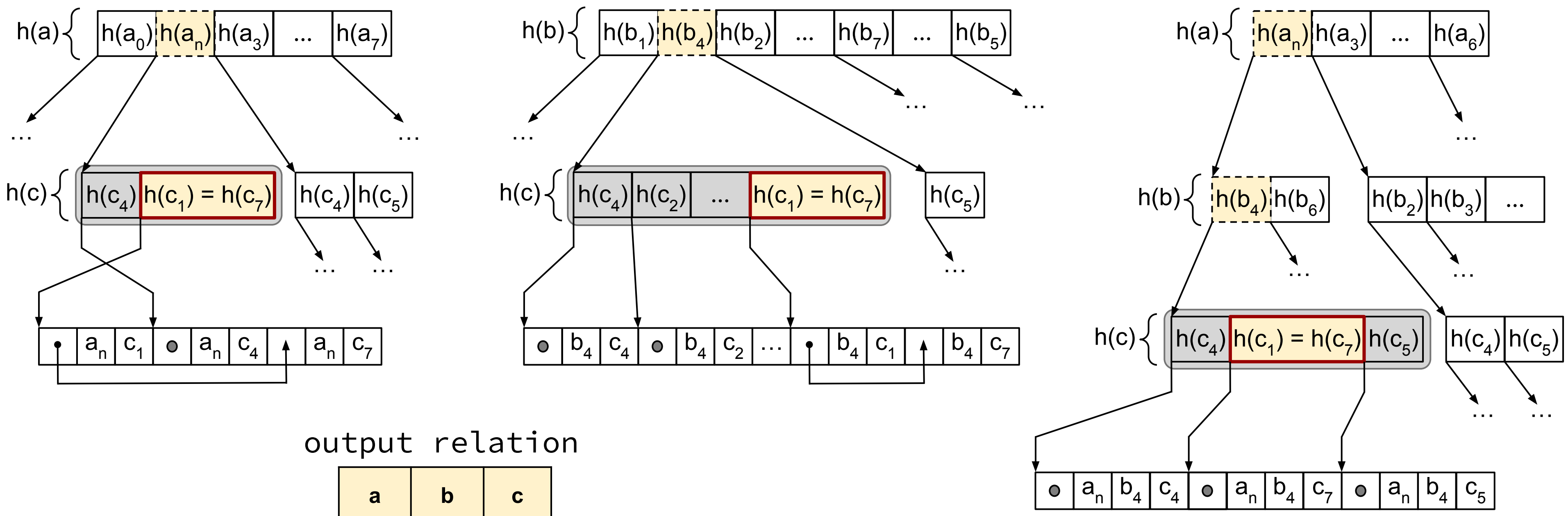
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

smaller size

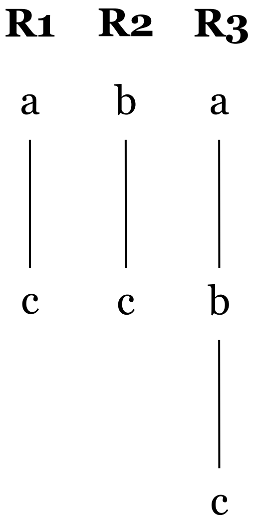
I1 **I2** **I3**

R1 (a, c) **R2 (b, c)** **R3 (a, b, c)**



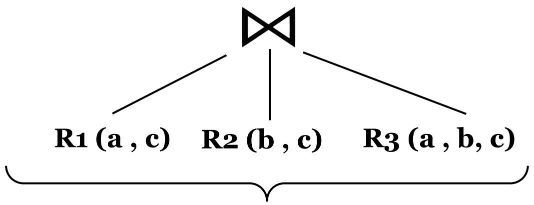
output relation

a	b	c
a_n	b_4	c_4



UNIVERSITY OF WATERLOO | DSG Data Systems Group

Probe - Enumeration Simulation



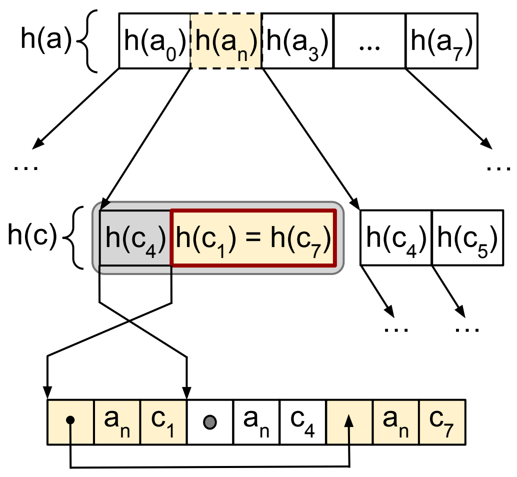
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

smaller size

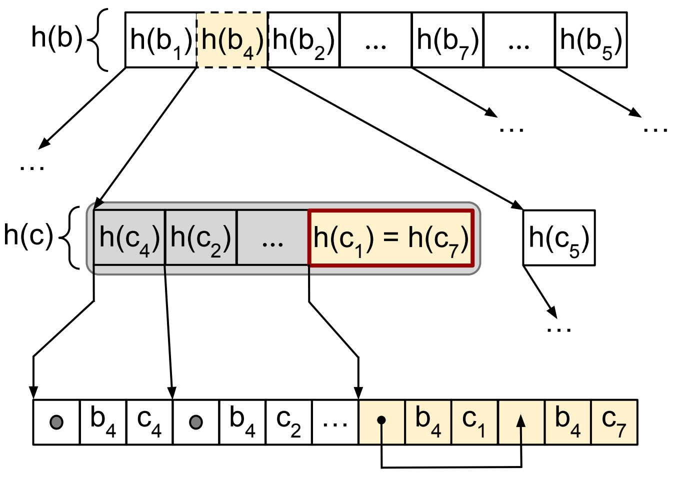
I1

R1 (a, c)



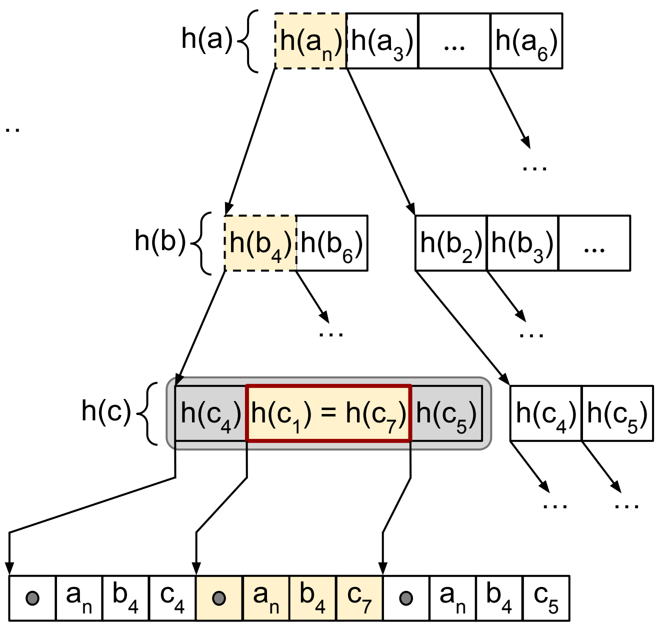
I2

R2 (b, c)



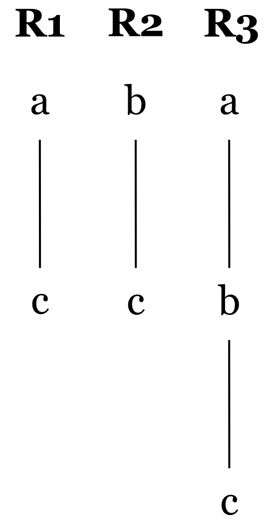
I3

R3 (a, b, c)

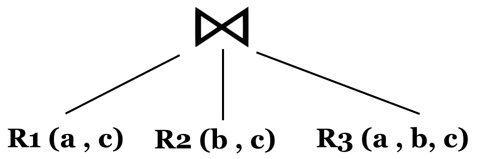


output relation

a	b	c
a_n	b_4	c_4

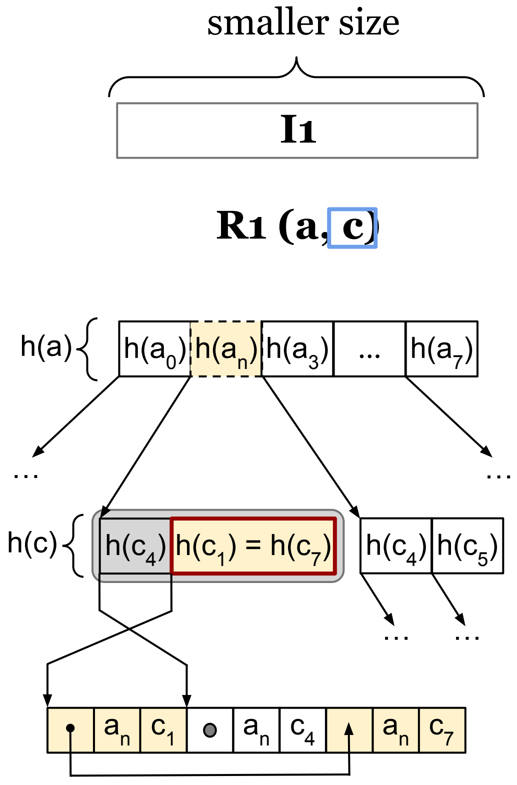


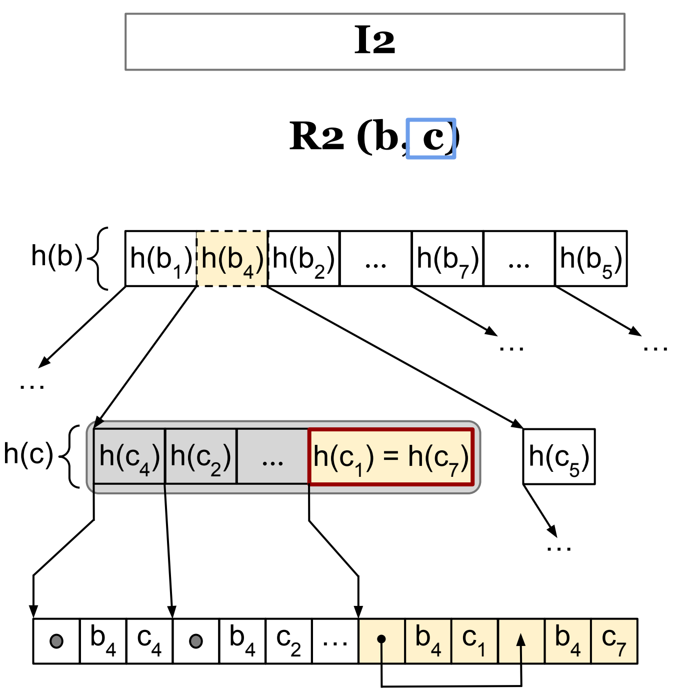
Probe - Enumeration Simulation

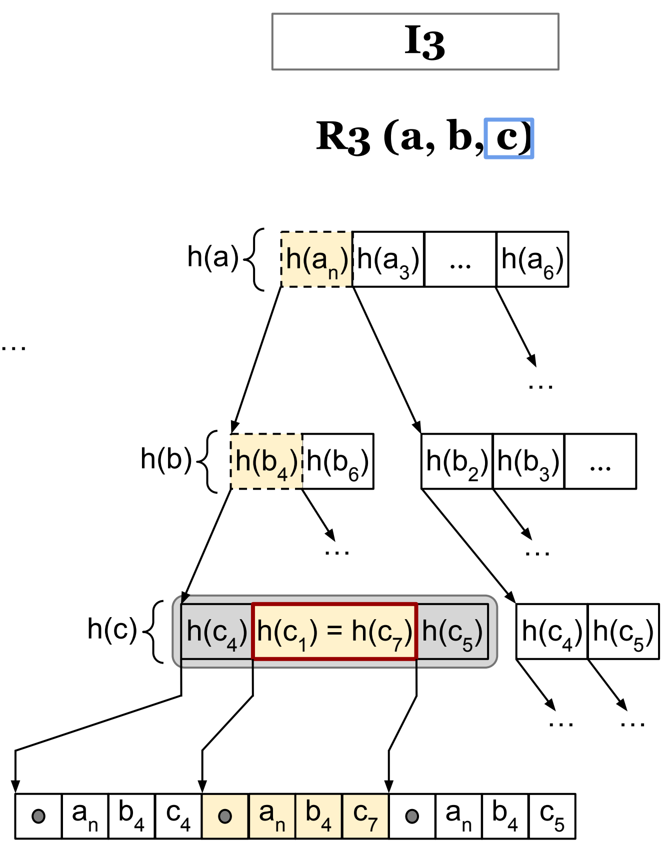


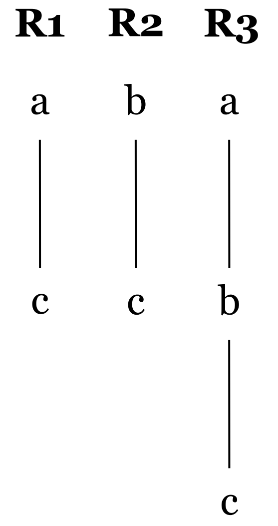
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]





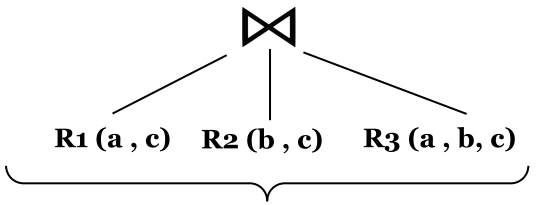




output relation

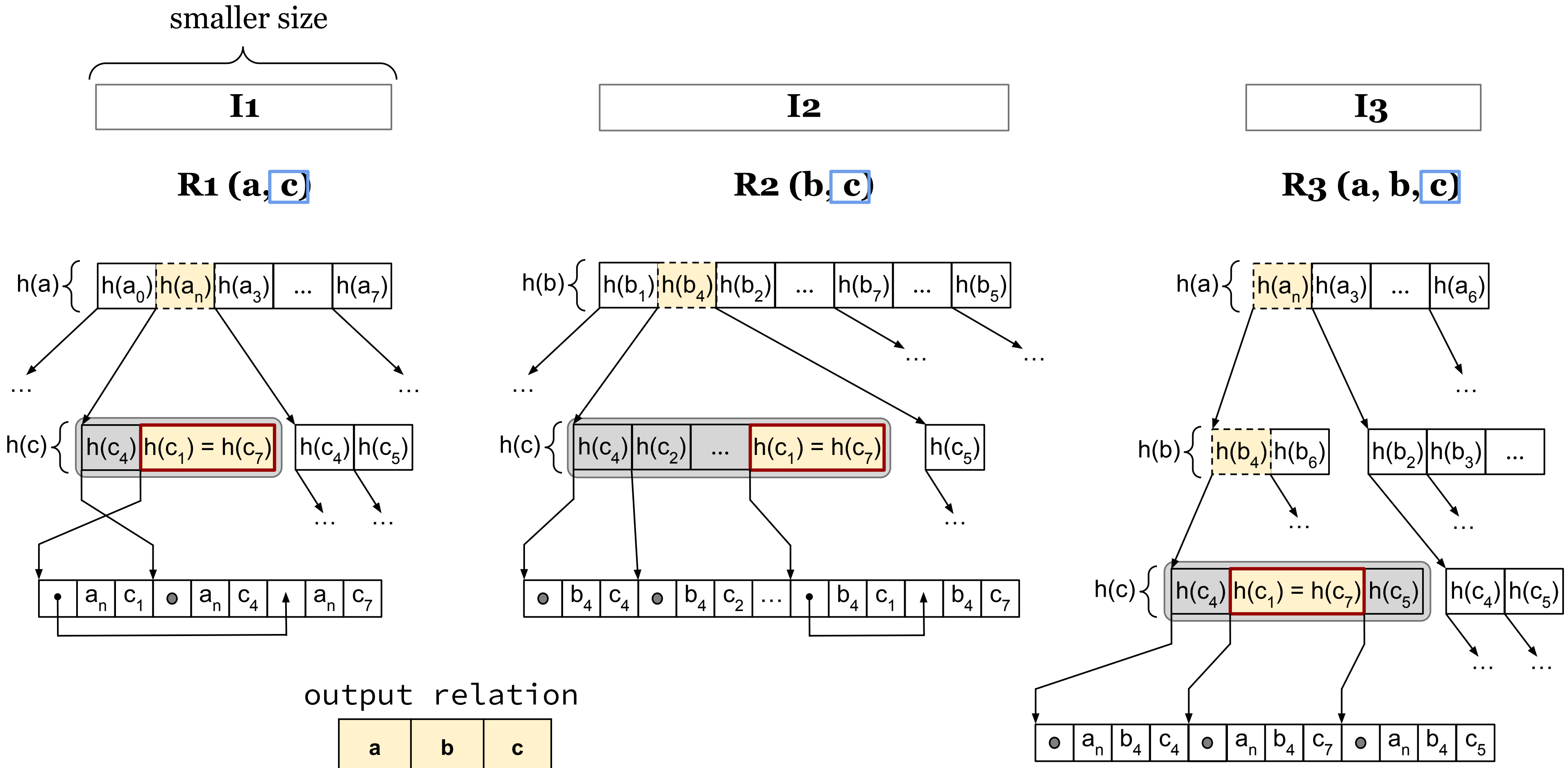
a	b	c
a_n	b_4	c_4
a_n	b_4	c_7

Probe - Enumeration Simulation



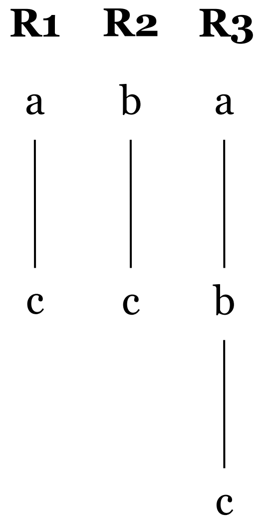
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

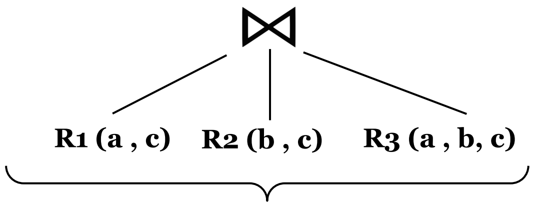


output relation

a	b	c
a_n	b_4	c_4
a_n	b_4	c_7

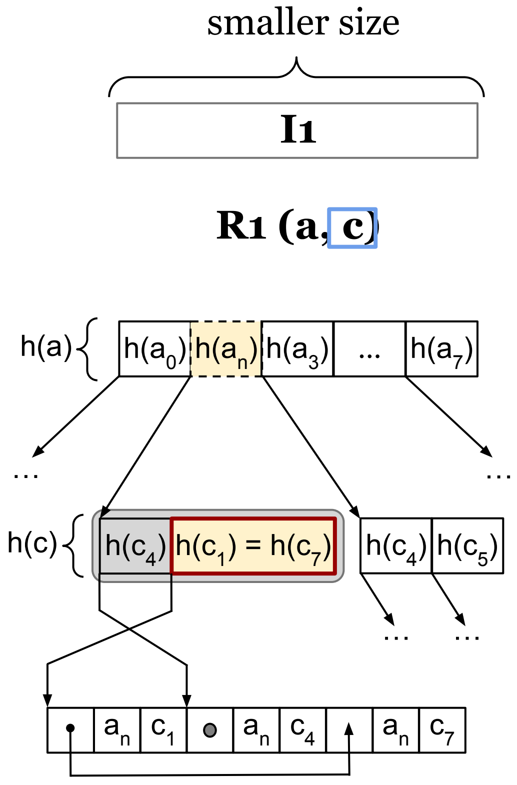


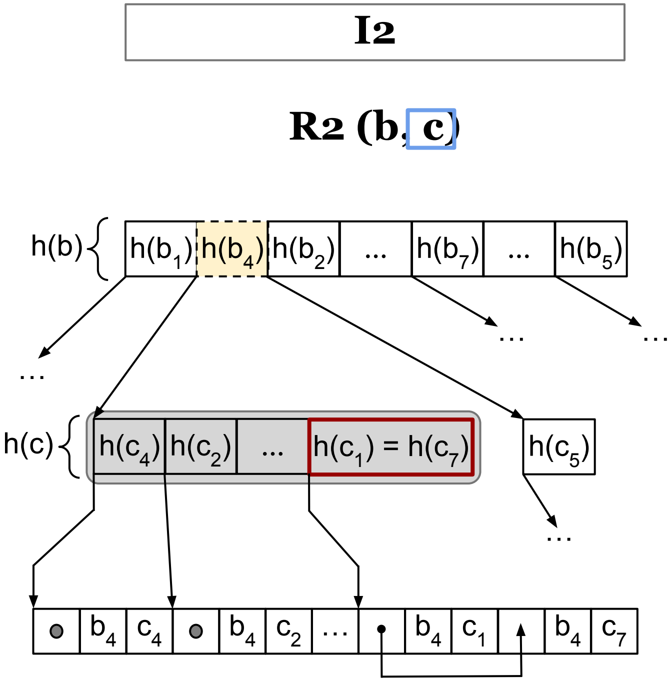
Probe - Enumeration Simulation

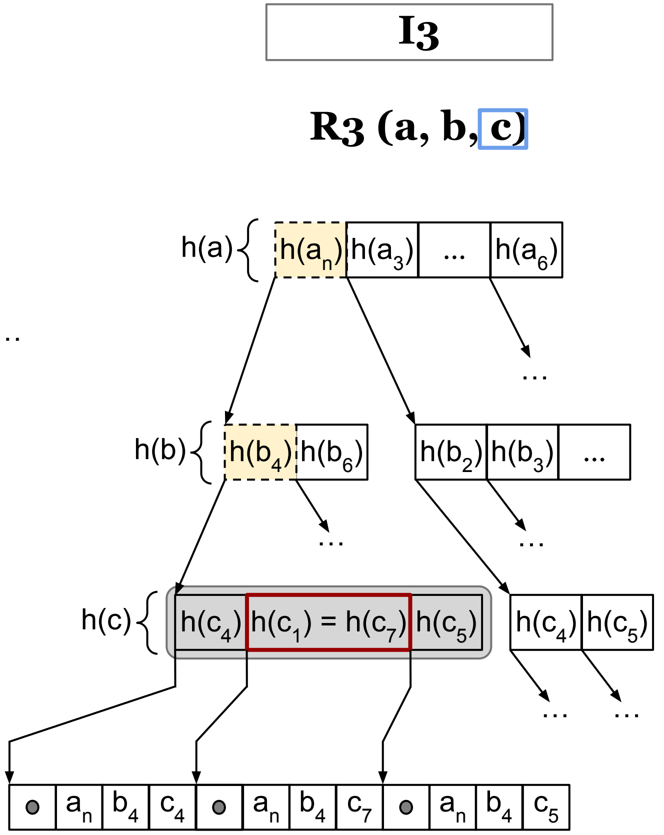


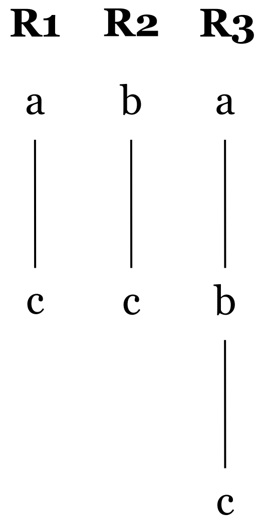
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]





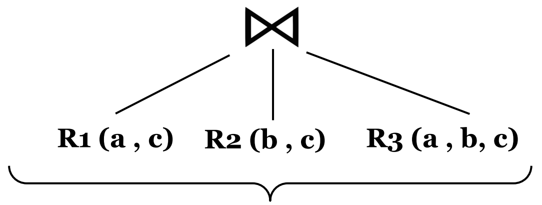




output relation

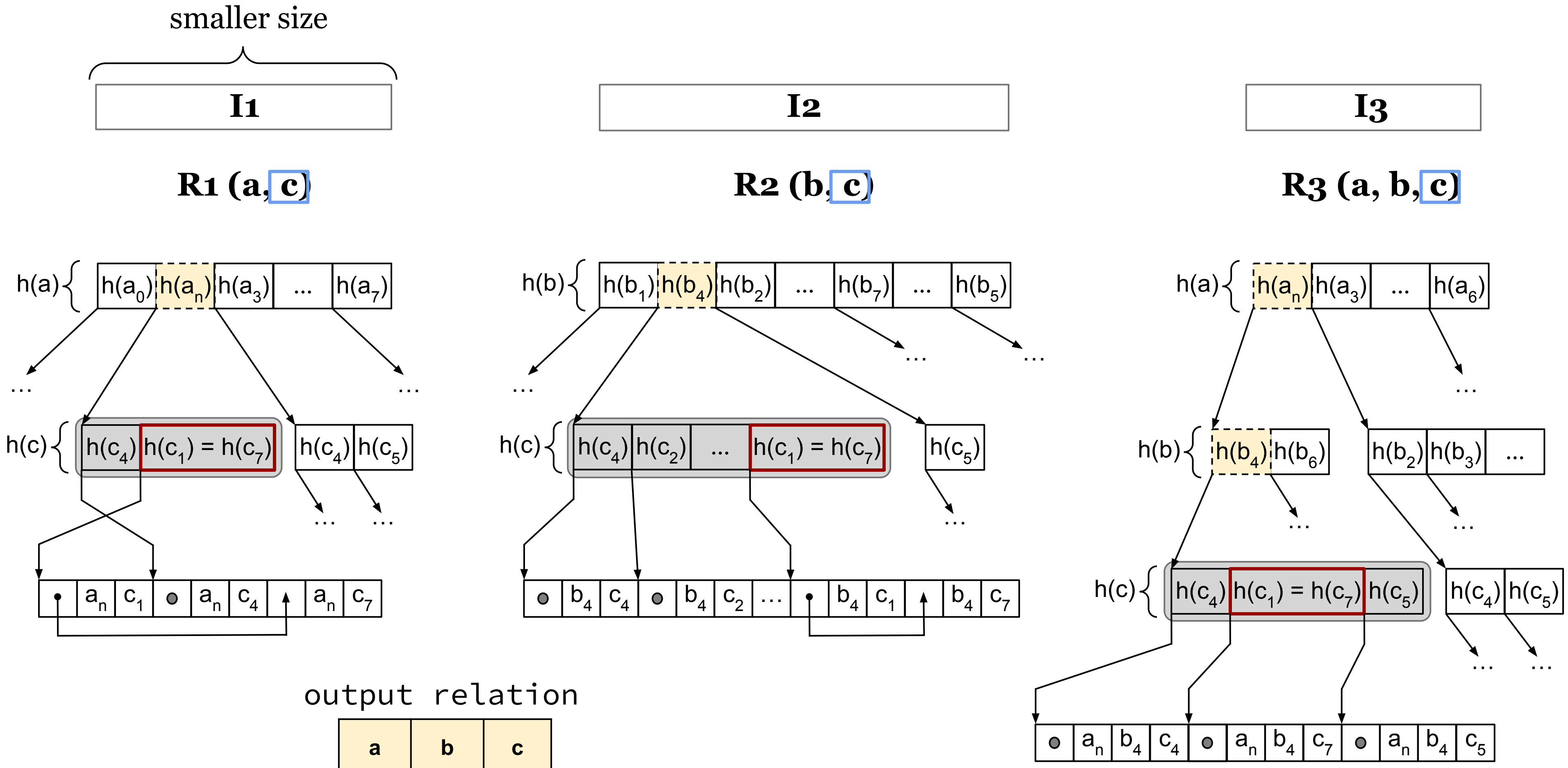
a	b	c
a_n	b_4	c_4
a_n	b_4	c_7

Probe - Enumeration Simulation



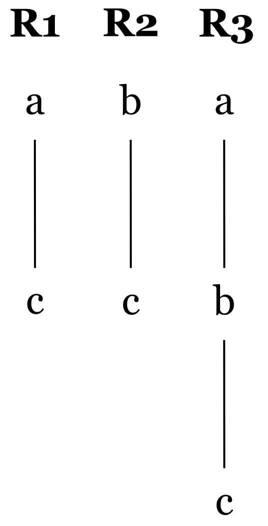
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b , c]

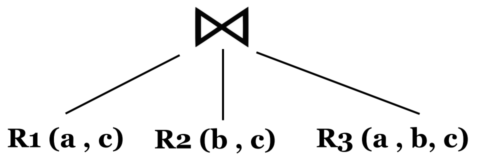


output relation

a	b	c
a_n	b_4	c_4
a_n	b_4	c_7

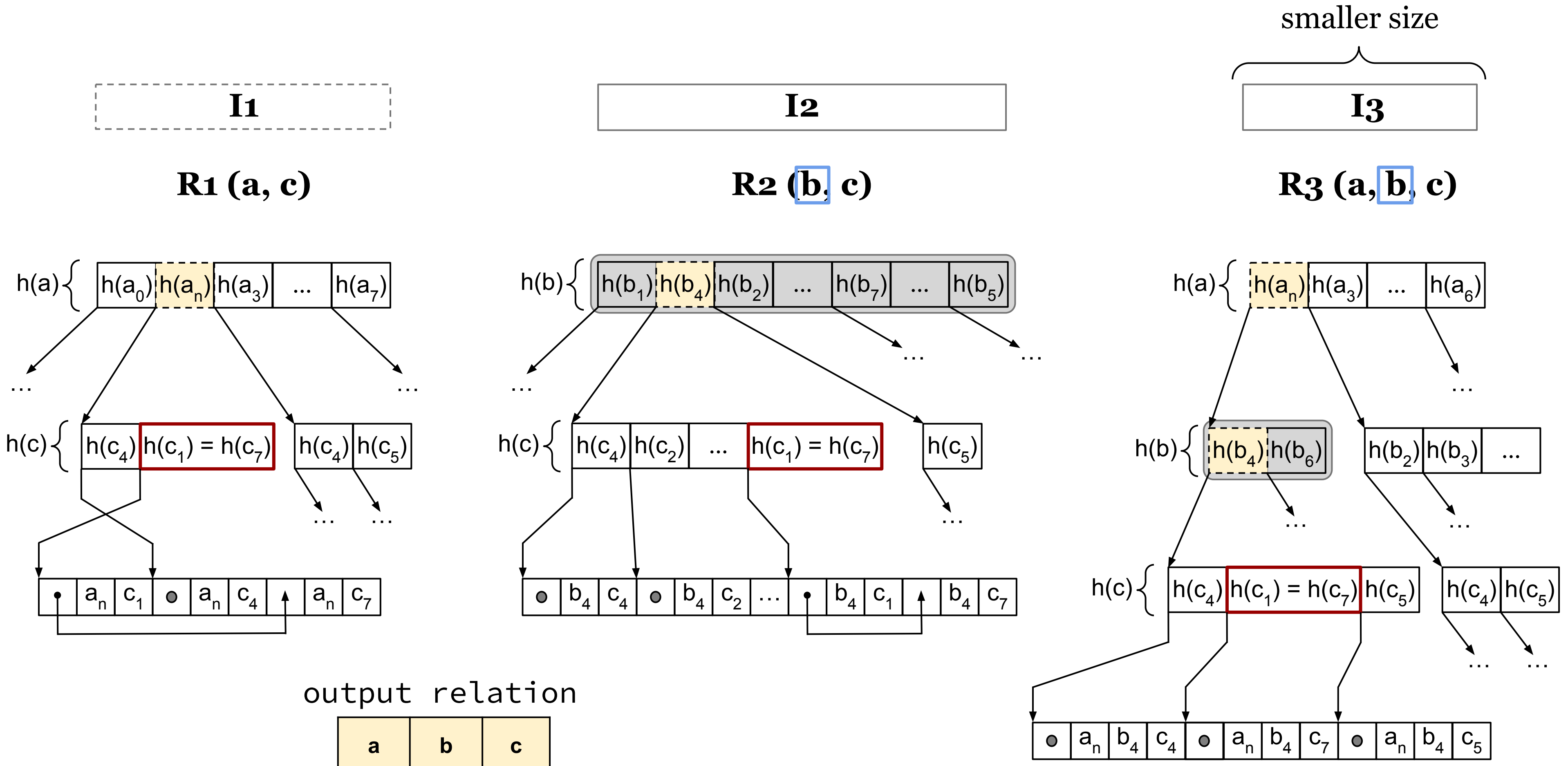


Probe - Enumeration Simulation



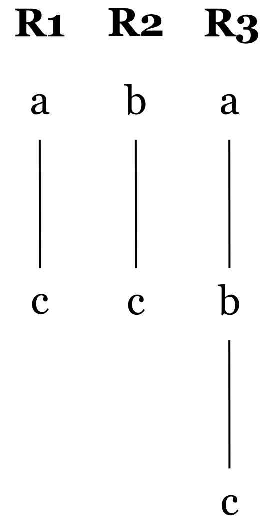
Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b, c]

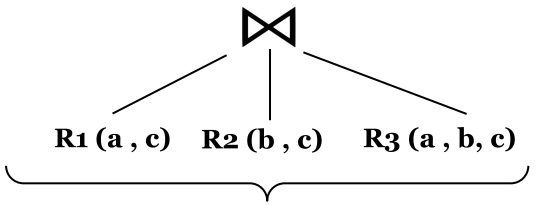


output relation

a	b	c
a_n	b_4	c_4
a_n	b_4	c_7

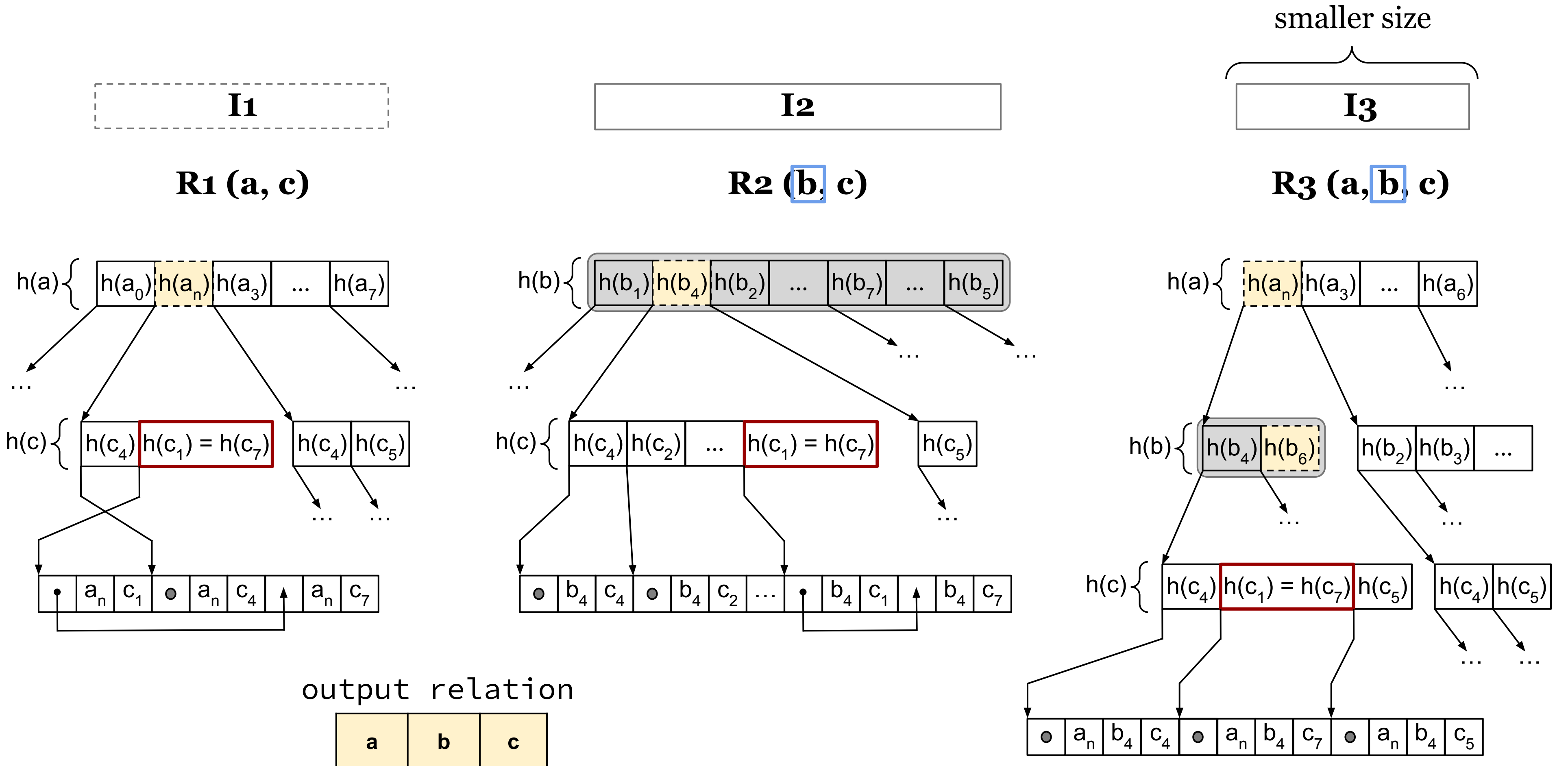


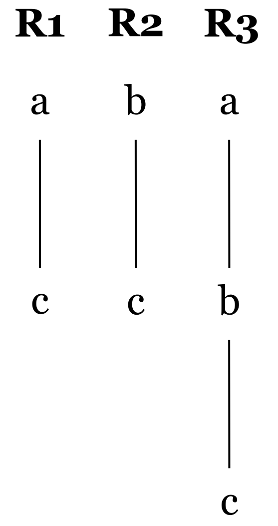
Probe - Enumeration Simulation



Build Hash Tries for R1, R2, and R3 following attribute ordering [a, b, c] in Trie-levels

Attribute ordering: [a , b, c]





output relation

a	b	c
a_n	b_4	c_4
a_n	b_4	c_7

Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case Optimal Joins (WCOJs)**

   Handling Intermediate Size Growth for Cyclic Joins

   2.1. Foundations

   2.2 System Integration Approaches

   i) **Index-based WCOJs (Graphflow & EmptyHeaded)**

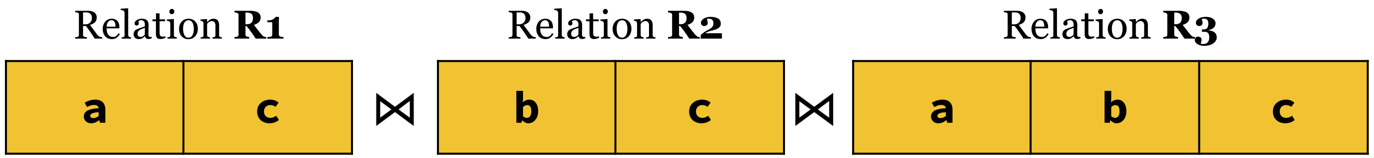
   ii) **Hash-based WCOJs (Umbra)**

- **Optimization approaches** (cost-based DP, GHD, **rule-based**)

3) **Factorized Query Processing**

Query Optimization - Rule-based

Query Optimization - Rule-based



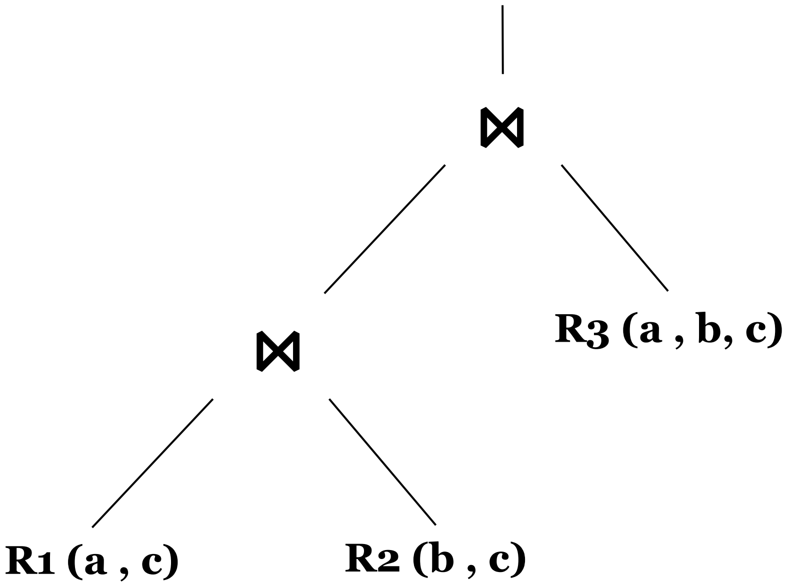
Relation **R1** Relation **R2** Relation **R3**

a	c	⋈	b	c	⋈	a	b	c

Query Optimization - Rule-based

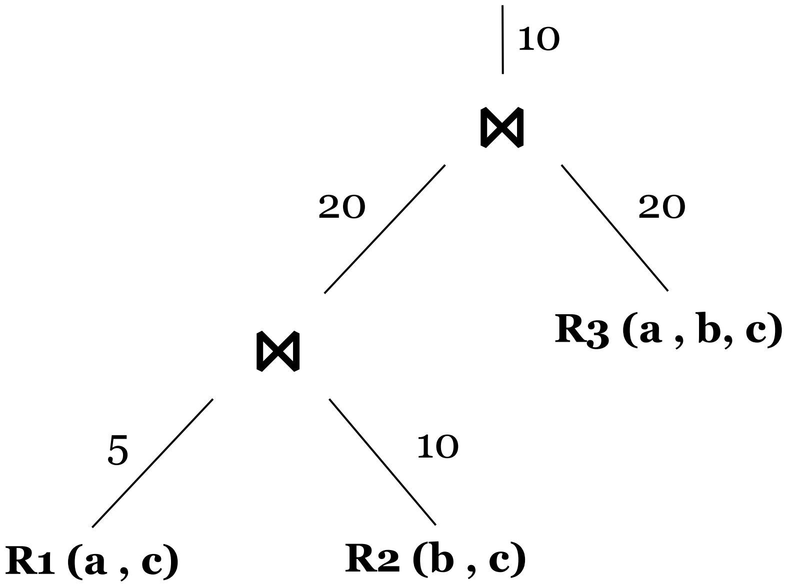
Relation **R1** | Relation **R2** | Relation **R3**

a c ⋈ b c ⋈ a b c



Query Optimization - Rule-based

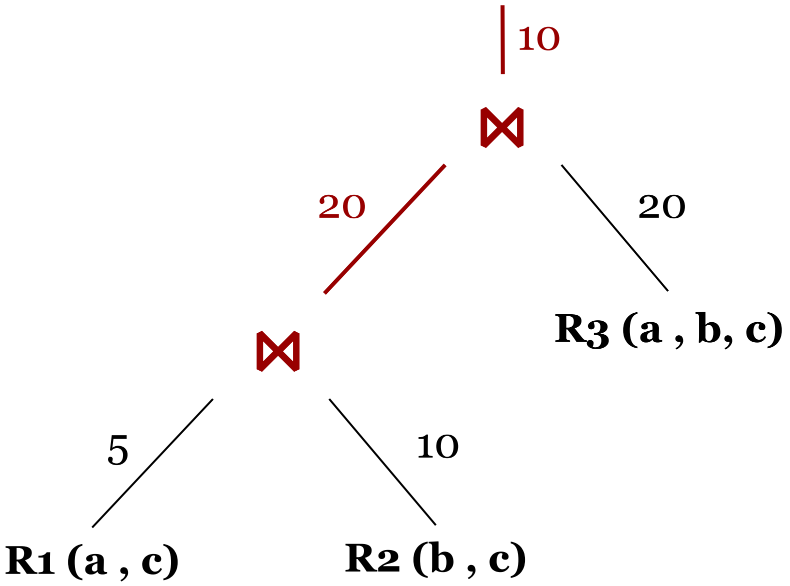
Relation **R1**			Relation **R2**			Relation **R3**		
a	c	⋈	b	c	⋈	a	b	c



Query Optimization - Rule-based

Relation **R1** | Relation **R2** | Relation **R3**

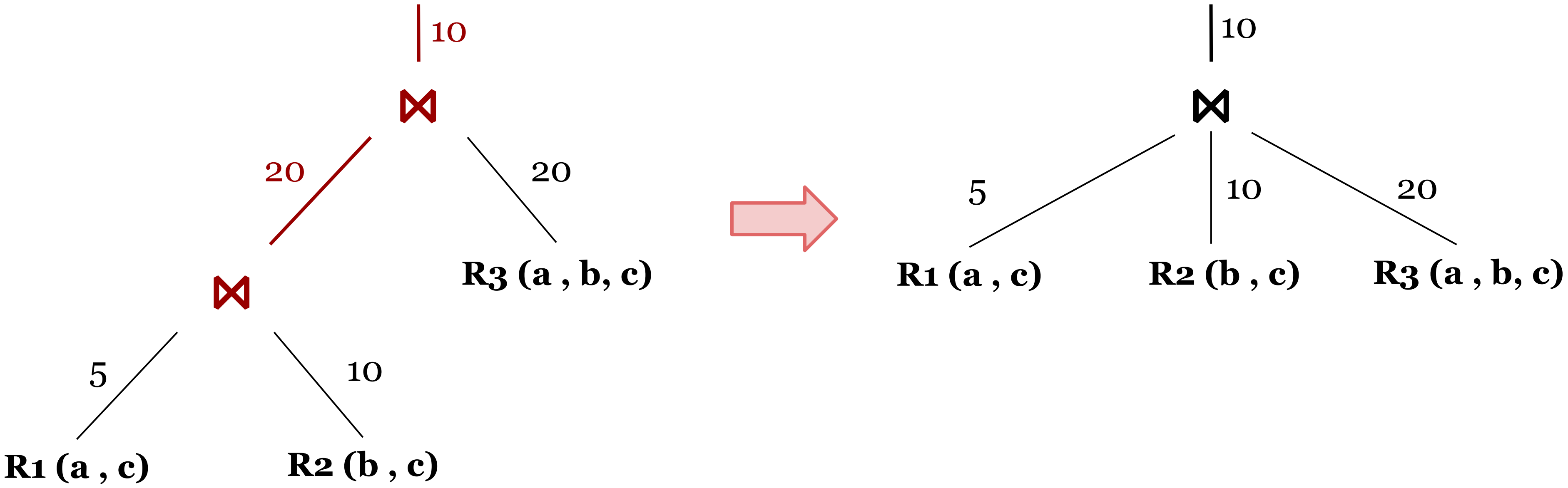
a | **c** ⋈ **b** | **c** ⋈ **a** | **b** | **c**



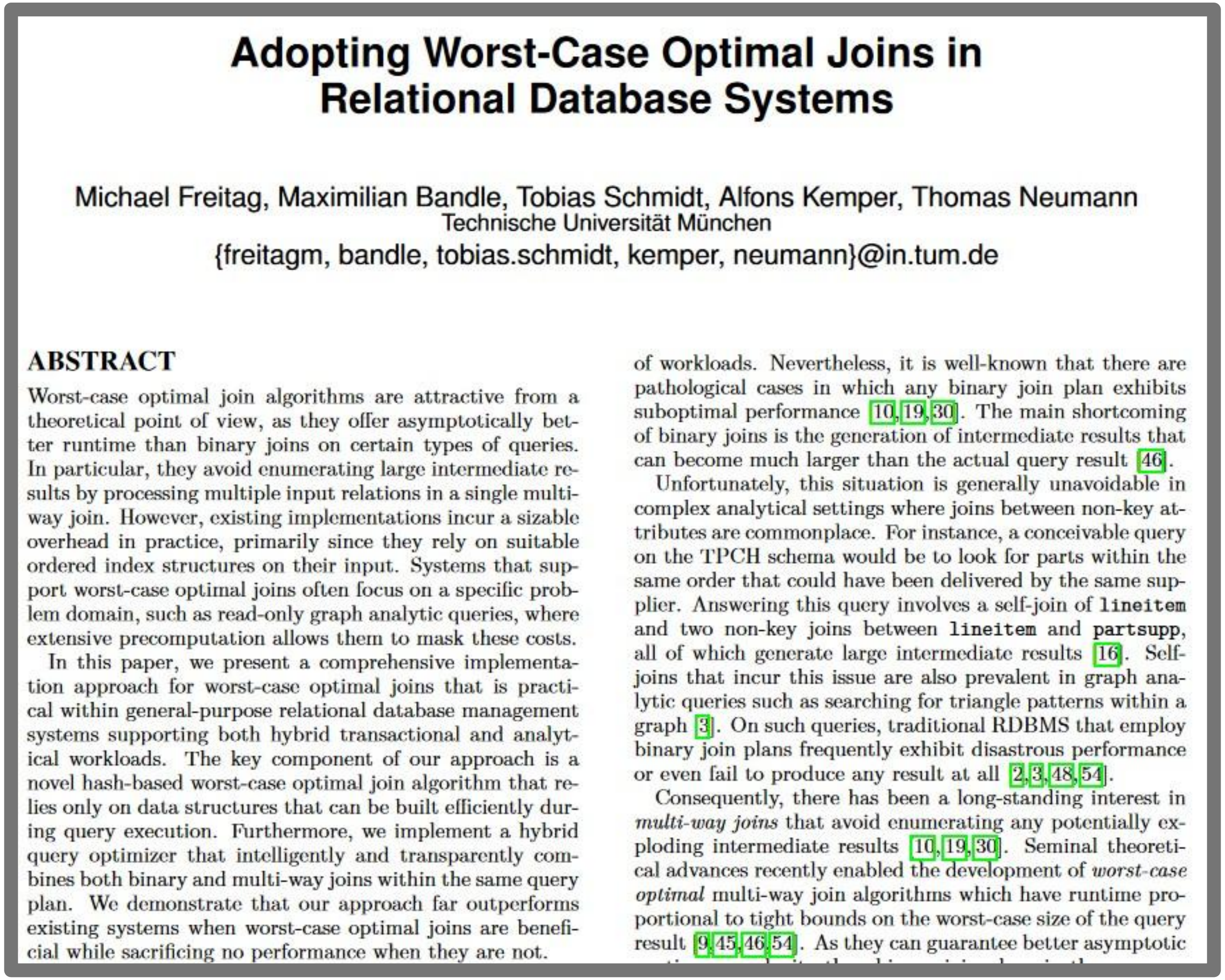
Query Optimization - Rule-based

Relation **R1** | a | c | ⋈ Relation **R2** | b | c | ⋈ Relation **R3** | a | b | c |

A binary join is replaced if it is classified as a ***growing join***, i.e. its output cardinality is greater than the maximum of its input cardinalities.



Further Details



Adopting Worst-Case Optimal Joins in Relational Database Systems

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, Thomas Neumann
Technische Universität München
{freitagm, bandle, tobias.schmidt, kemper, neumann}@in.tum.de

ABSTRACT

Worst-case optimal join algorithms are attractive from a theoretical point of view, as they offer asymptotically better runtime than binary joins on certain types of queries. In particular, they avoid enumerating large intermediate results by processing multiple input relations in a single multi-way join. However, existing implementations incur a sizable overhead in practice, primarily since they rely on suitable ordered index structures on their input. Systems that support worst-case optimal joins often focus on a specific problem domain, such as read-only graph analytic queries, where extensive precomputation allows them to mask these costs.

In this paper, we present a comprehensive implementation approach for worst-case optimal joins that is practical within general-purpose relational database management systems supporting both hybrid transactional and analytical workloads. The key component of our approach is a novel hash-based worst-case optimal join algorithm that relies only on data structures that can be built efficiently during query execution. Furthermore, we implement a hybrid query optimizer that intelligently and transparently combines both binary and multi-way joins within the same query plan. We demonstrate that our approach far outperforms existing systems when worst-case optimal joins are beneficial while sacrificing no performance when they are not.

of workloads. Nevertheless, it is well-known that there are pathological cases in which any binary join plan exhibits suboptimal performance [10,19,30]. The main shortcoming of binary joins is the generation of intermediate results that can become much larger than the actual query result [46].

Unfortunately, this situation is generally unavoidable in complex analytical settings where joins between non-key attributes are commonplace. For instance, a conceivable query on the TPCH schema would be to look for parts within the same order that could have been delivered by the same supplier. Answering this query involves a self-join of `lineitem` and two non-key joins between `lineitem` and `partsupp`, all of which generate large intermediate results [16]. Self-joins that incur this issue are also prevalent in graph analytic queries such as searching for triangle patterns within a graph [3]. On such queries, traditional RDBMS that employ binary join plans frequently exhibit disastrous performance or even fail to produce any result at all [2,3,48,54].

Consequently, there has been a long-standing interest in *multi-way joins* that avoid enumerating any potentially exploding intermediate results [10,19,30]. Seminal theoretical advances recently enabled the development of *worst-case optimal* multi-way join algorithms which have runtime proportional to tight bounds on the worst-case size of the query result [9,45,46,54]. As they can guarantee better asymptotic

Open Challenges

Open Challenges

1. **Sorting-on-the fly:**
    - Direct comparison to hashing-on-the-fly
    - Should be slower to index but faster to join
    - Question: Which performs better and when?

Open Challenges

1. **Sorting-on-the fly:**
   - Direct comparison to hashing-on-the-fly
   - Should be slower to index but faster to join
   - Question: Which performs better and when?

2. **Databases Cracking (Idreos et al. CIDR 2017):**
   - " … addressing index maintenance as part of query processing using continuous physical reorganization …"
   - Don't presort any lists
   - Sort the adjacency lists that are being integrated frequently during query processing
   - Seems easier to integrate into GDBMSs than RDBMSs.

Open Challenges

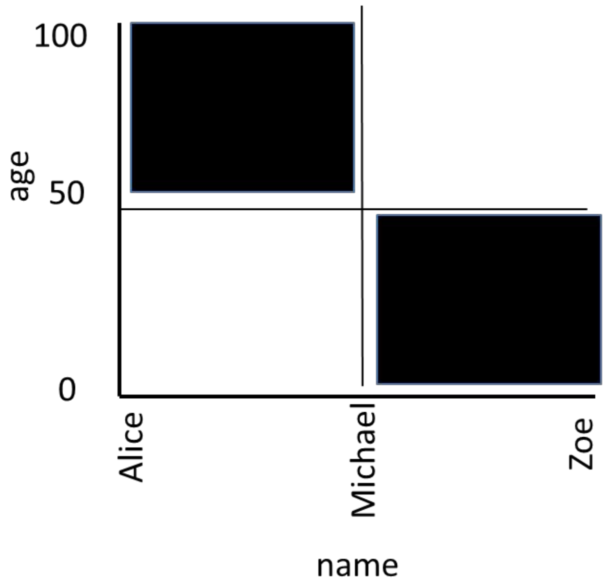
3. Beyond WCOJ algorithms:

- Algorithms with instance optimality guarantees.
- Minesweeper [NGO et al., OIDS ‘14] Tetris [Khamis et al., TODS ‘16]
- *Fundamental Question:*
    - How do we know that a join algorithm’s output is correct?

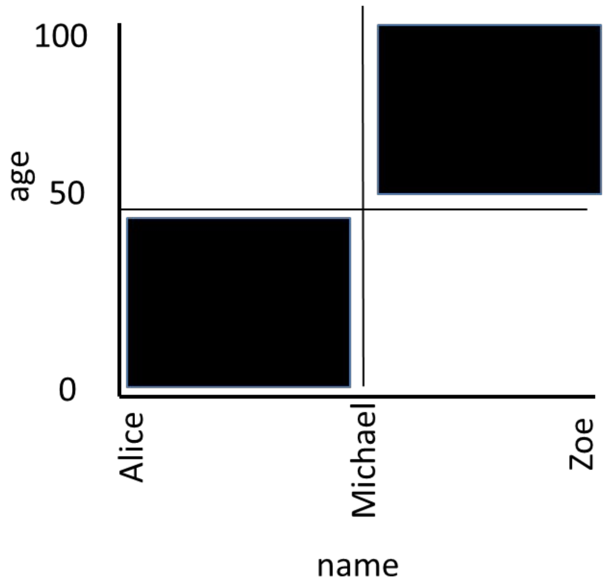
- Answer: Any join algorithm implicitly provides a “proof” of its output through comparisons. Runtime ~ proof size.

- Next question: What is the minimum proof size?

Beyond WCOJ algorithms



R(name, age)



S(name, age)

There is a O(1) “proof” here with right indexing! Standard binary join or WCOJs would take at least linear time in the number of tuples.

Research Question: Can we make Tetris-like algorithms practical.
(If you take a close look, algorithms have very high constants in their asymptotic runtimes)

Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case optimal joins**

3) **Factorized Query Processing**
   Handling Intermediate Size Growth for Acyclic Joins

Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case optimal joins**

3) **Factorized Query Processing**
   Handling Intermediate Size Growth for Acyclic Joins
   3.1. Foundations: Factorized Representations
   3.2. System Integration Approaches: FDB and Factorized Vector Execution in Graphflow

Outline of Query Processing Techniques to Cover

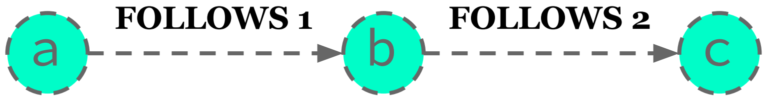
For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

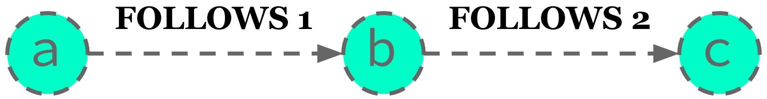
2) **Worst-case optimal joins**

3) **Factorized Query Processing**
   Handling Intermediate Size Growth for Acyclic Joins
   3.1. Foundations: Factorized Representations
   3.2. System Integration Approaches: FDB and Factorized Vector Execution in Graphflow

Flat Representation Example



Flat Representation Example



Relation **FOLLOWS (F1)**

`FROM (a)`	`TO (b)`

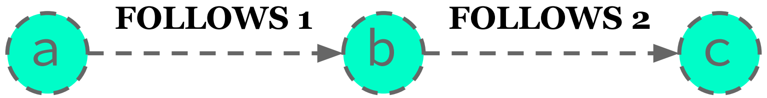
⋈

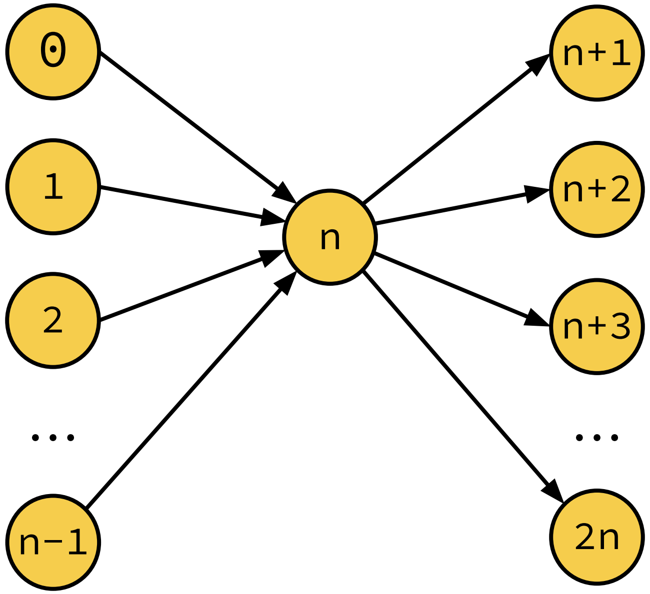
Relation **FOLLOWS (F2)**

`FROM (b)`	`TO (c)`

Flat Representation Example

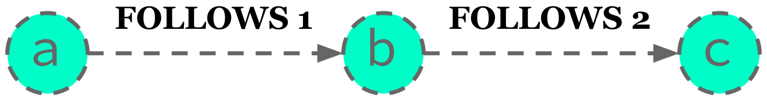
Query Vertex Ordering:
[b, a, c]





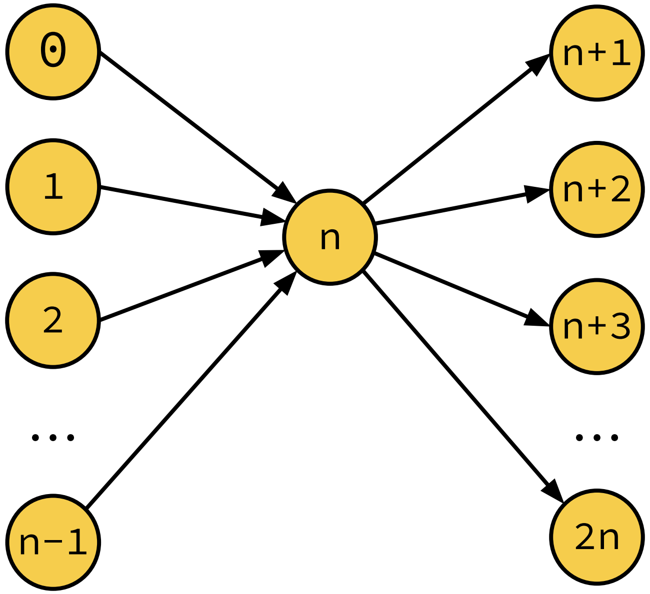
Flat Representation Example

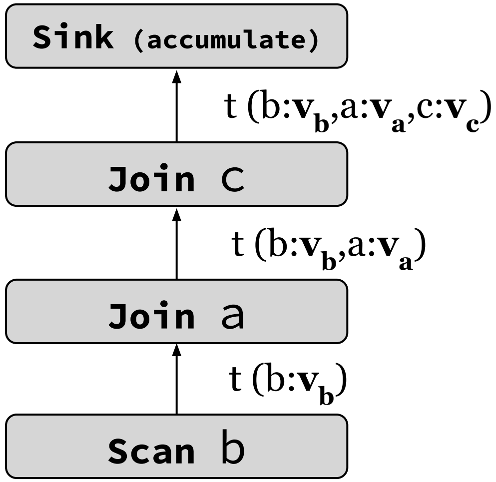
Query Vertex Ordering:
[b, a, c]



Output Relation

b.ID	a.ID	c.ID





`Sink (accumulate)`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

`Join c`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

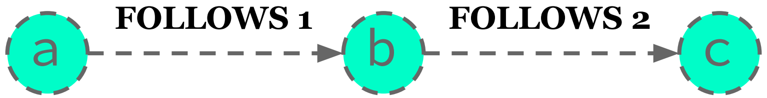
`Join a`

t (b:$\mathbf{v}_b$)

`Scan b`

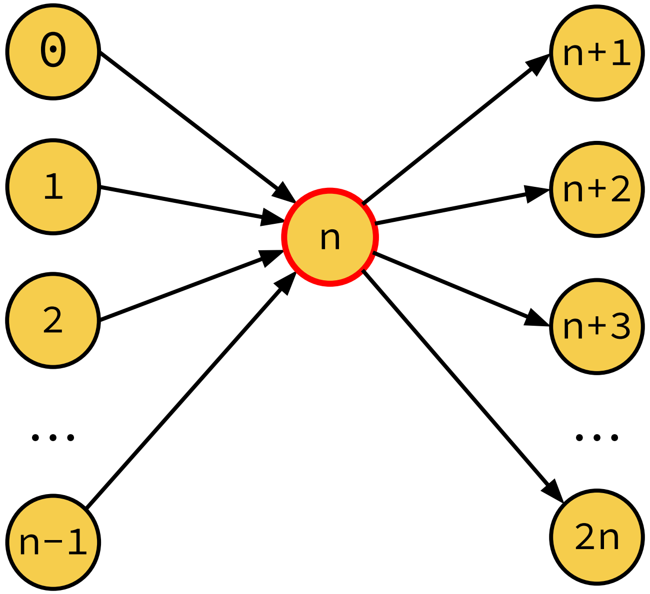
Flat Representation Example

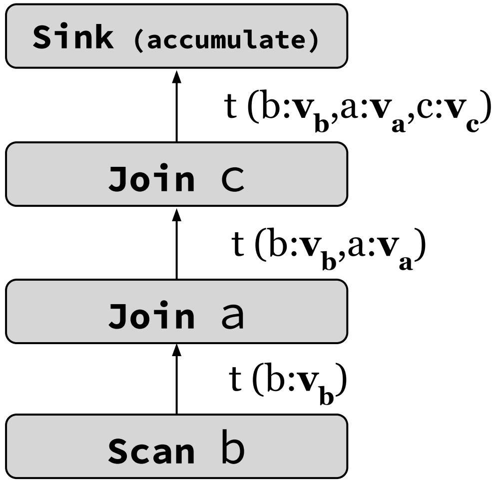
Query Vertex Ordering:
[b, a, c]



Output Relation

b.ID	a.ID	c.ID





Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

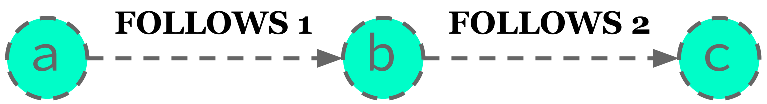
Join a

t (b:$\mathbf{v}_b$)

Scan b

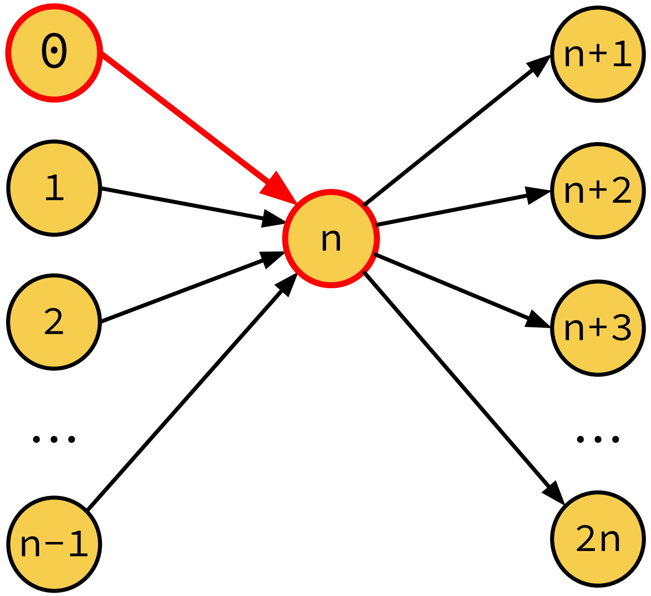
Flat Representation Example

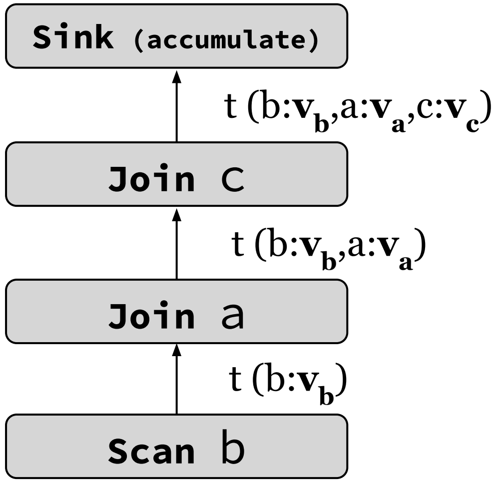
Query Vertex Ordering:
[b, a, c]



Output Relation

b.ID	a.ID	c.ID





Sink (accumulate)

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

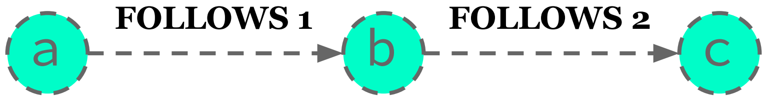
Join a

$t\,(b{:}\mathbf{v}_b)$

Scan b

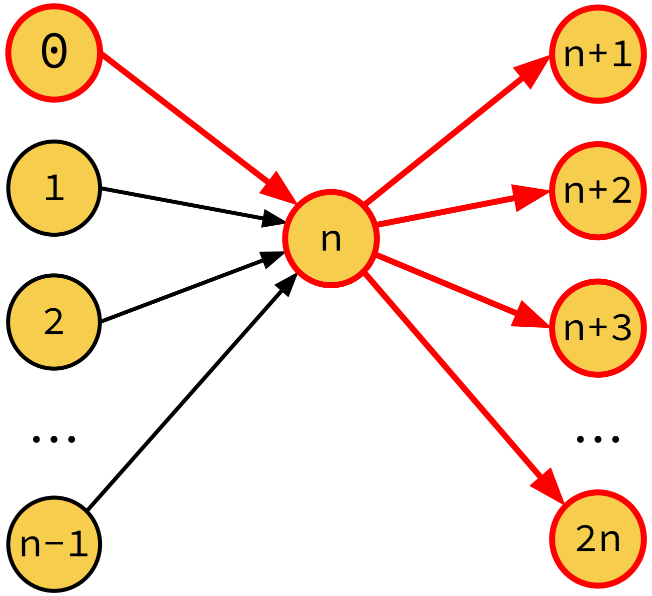
Flat Representation Example

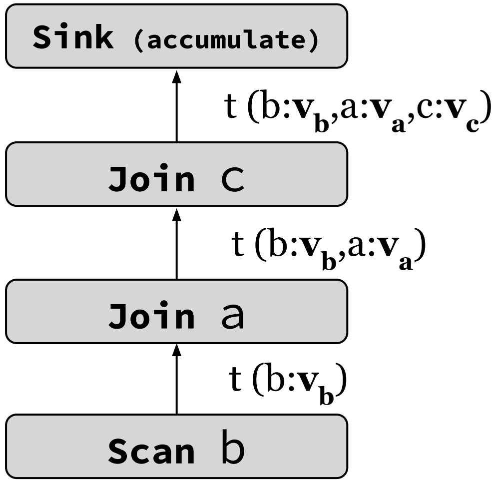
Query Vertex Ordering:
[b, a, c]



Output Relation

b.ID	a.ID	c.ID





Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

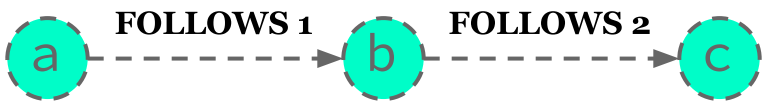
Join a

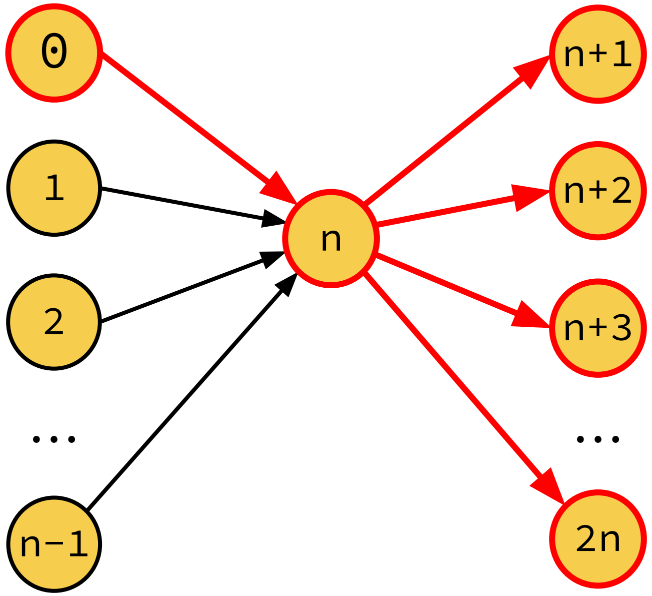
t (b:$\mathbf{v}_b$)

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]

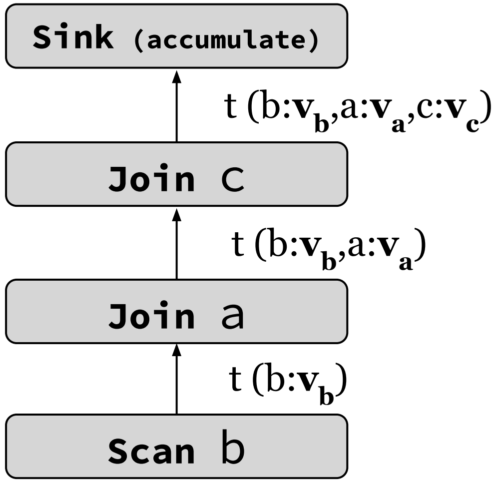




Output Relation

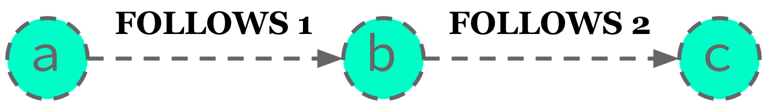
b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n

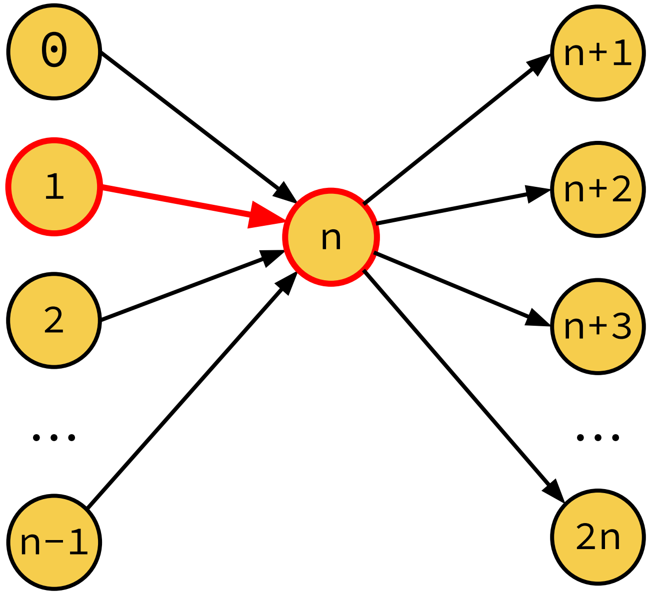
n tuples



Flat Representation Example

Query Vertex Ordering:
[b, a, c]

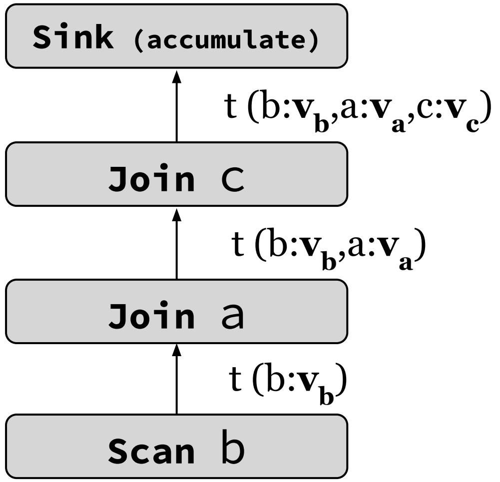




Output Relation

b.ID	**a.ID**	**c.ID**
n	0	n+1
n	0	...
n	0	2n

n tuples



Sink (accumulate)

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

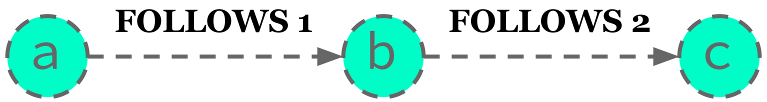
Join a

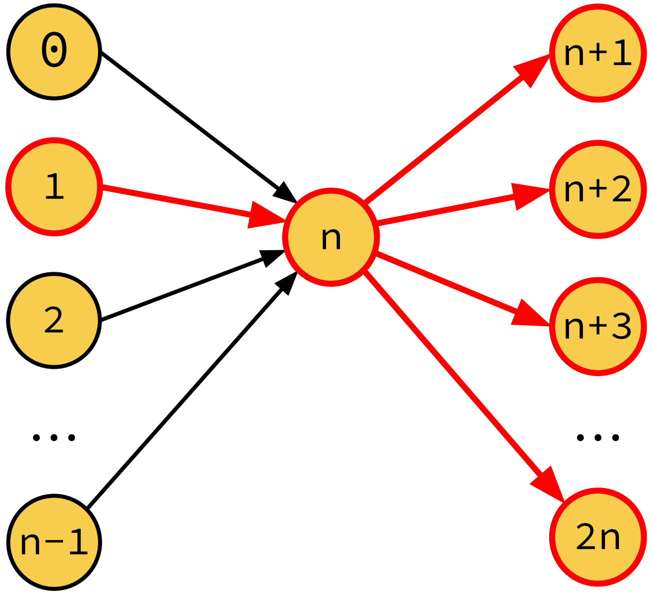
$t\,(b{:}\mathbf{v}_b)$

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]

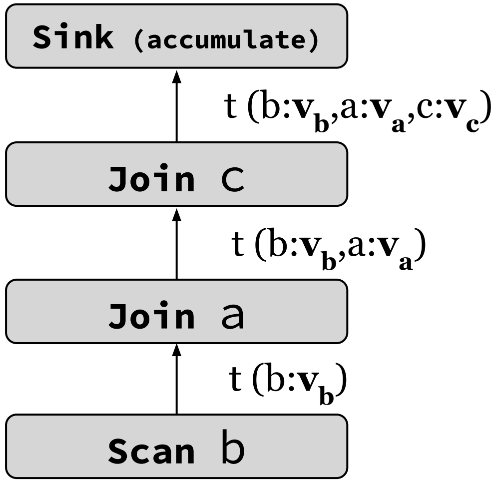




Output Relation

b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n

n tuples



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

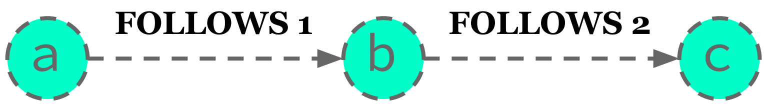
Join a

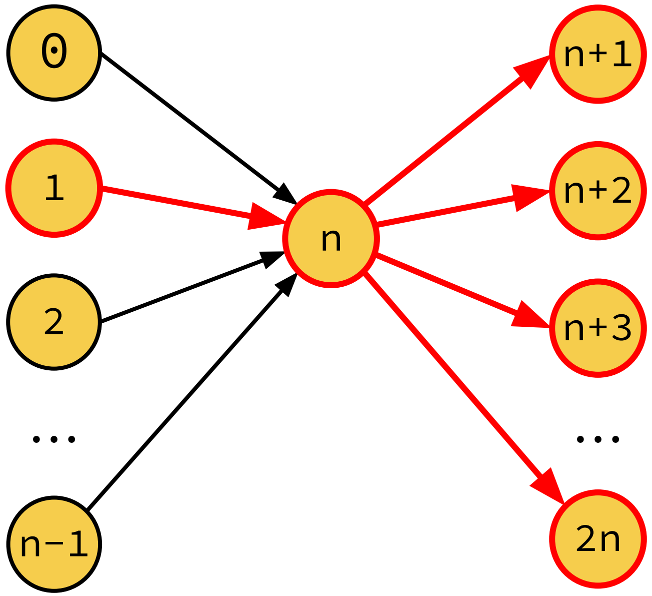
$t\ (b{:}\mathbf{v}_b)$

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]



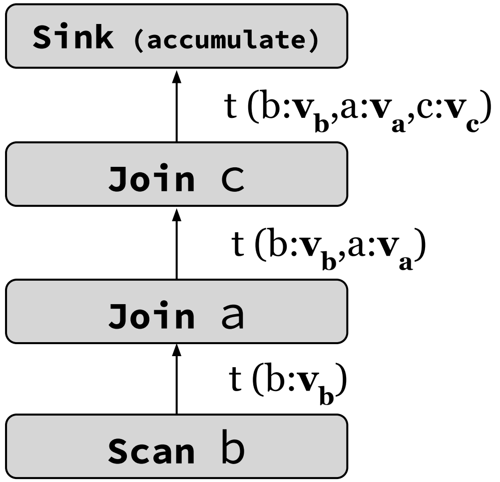


Output Relation

b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n

n tuples

n tuples



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

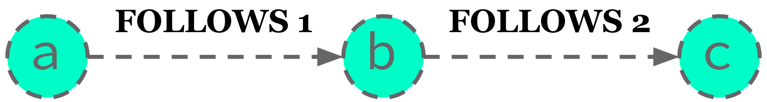
Join a

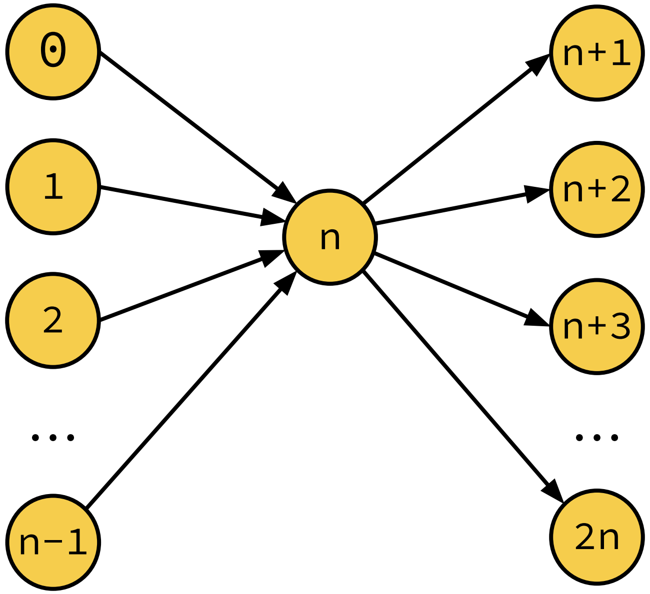
t $(b{:}\mathbf{v}_b)$

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]





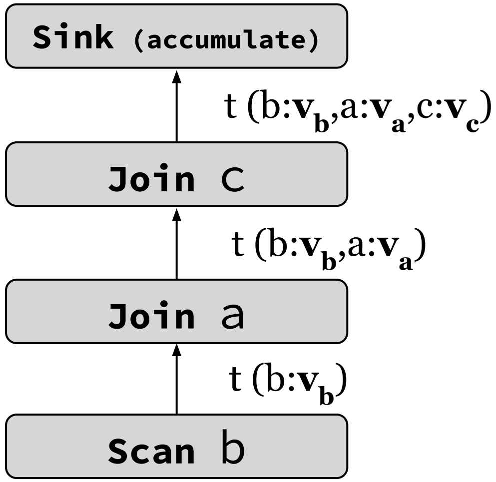
Output Relation

b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n
...	...	...
n	n-1	n+1
n	n-1	...
n	n-1	2n

n tuples

n tuples

n tuples



`Sink (accumulate)`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

`Join c`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

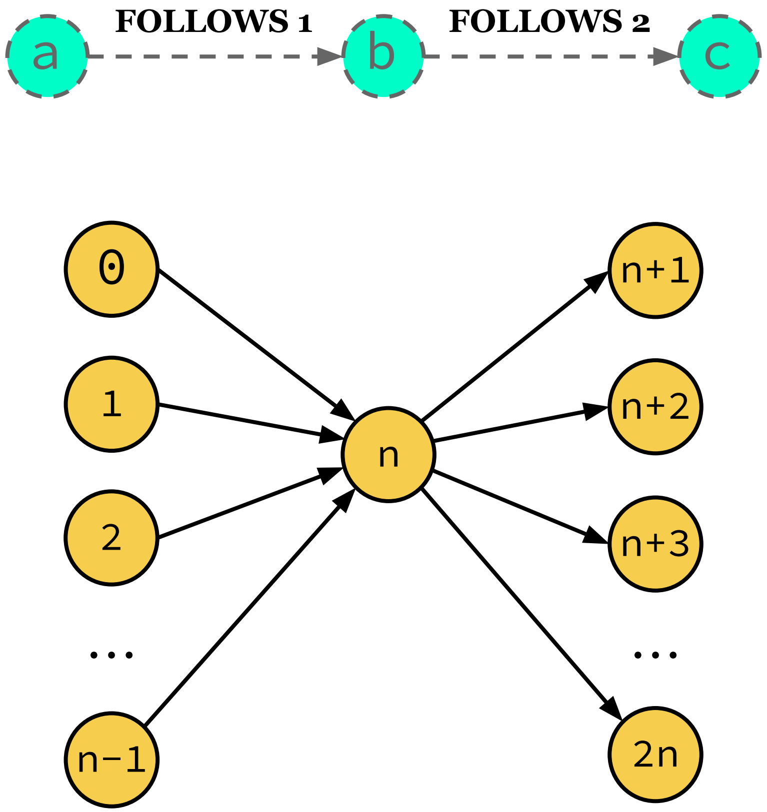
`Join a`

t $(b{:}\mathbf{v}_b)$

`Scan b`

Flat Representation Example

Query Vertex Ordering:
[b, a, c]



Output Relation

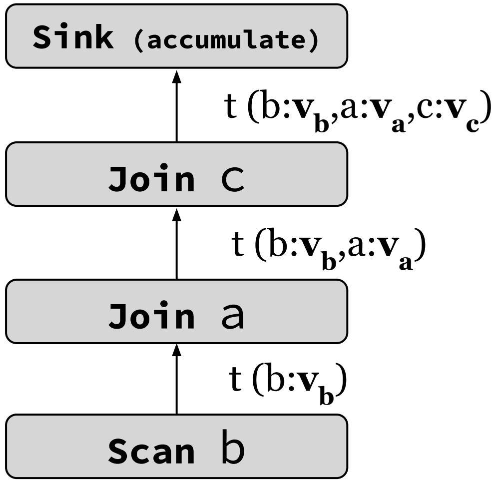
b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n
...	...	...
n	n-1	n+1
n	n-1	...
n	n-1	2n

n^2 tuples

n tuples

n tuples

n tuples



Sink (accumulate)

t ($b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c$)

Join c

t ($b{:}\mathbf{v}_b, a{:}\mathbf{v}_a$)

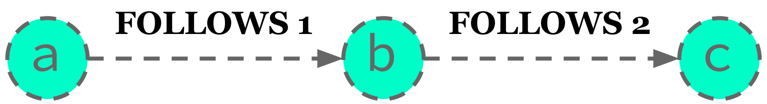
Join a

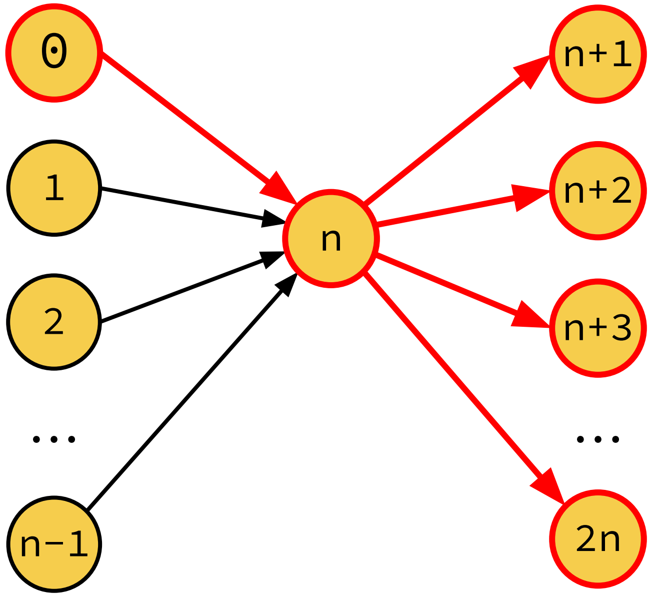
t ($b{:}\mathbf{v}_b$)

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]





Output Relation

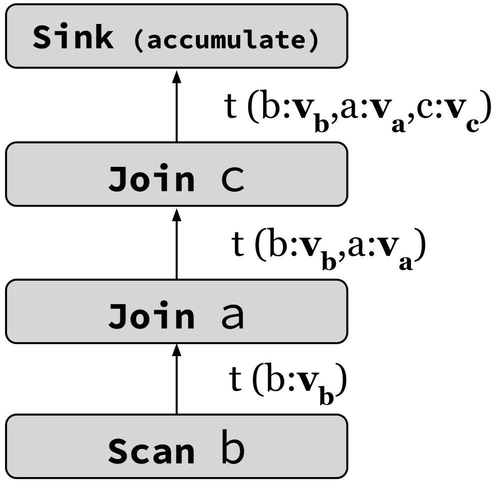
b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n
...	...	...
n	n-1	n+1
n	n-1	...
n	n-1	2n

n^2 tuples

n tuples

n tuples

n tuples



Sink (accumulate)

t ($b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c$)

Join c

t ($b{:}\mathbf{v}_b, a{:}\mathbf{v}_a$)

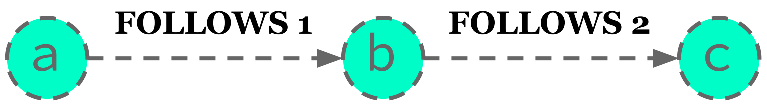
Join a

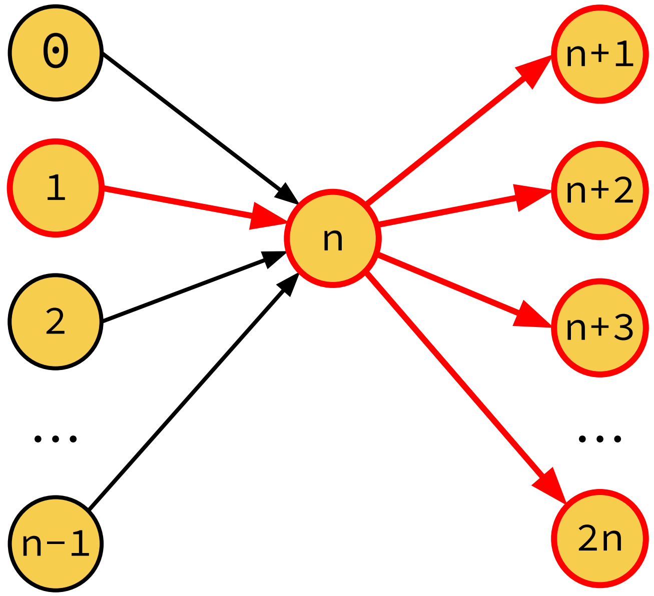
t ($b{:}\mathbf{v}_b$)

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]





Output Relation

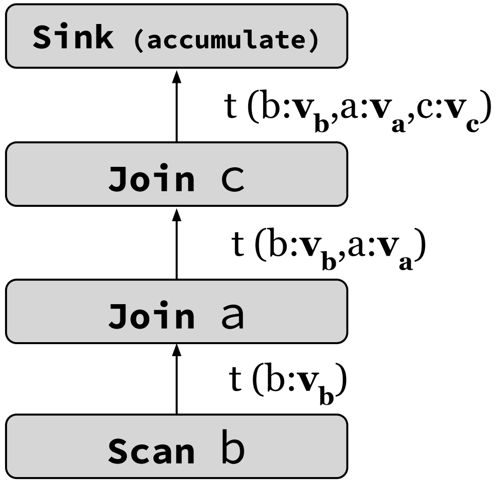
b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n
...	...	...
n	n-1	n+1
n	n-1	...
n	n-1	2n

n^2 tuples

n tuples

n tuples

n tuples



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

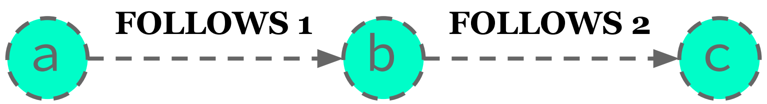
Join a

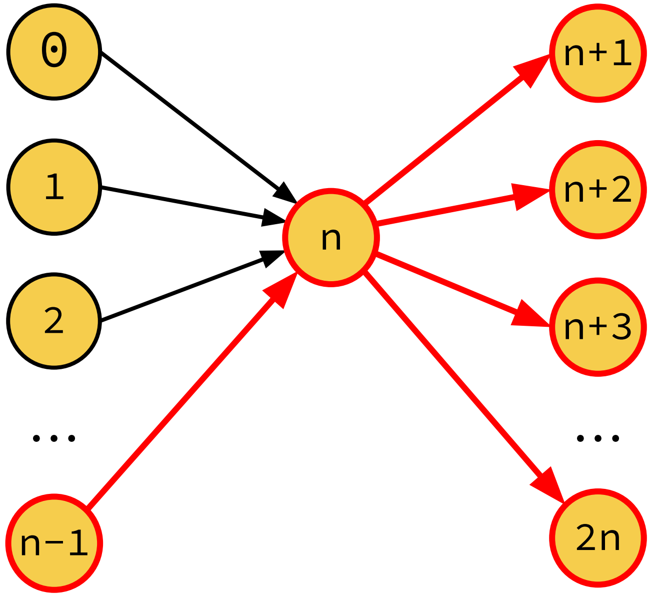
t $(b{:}\mathbf{v}_b)$

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]





Output Relation

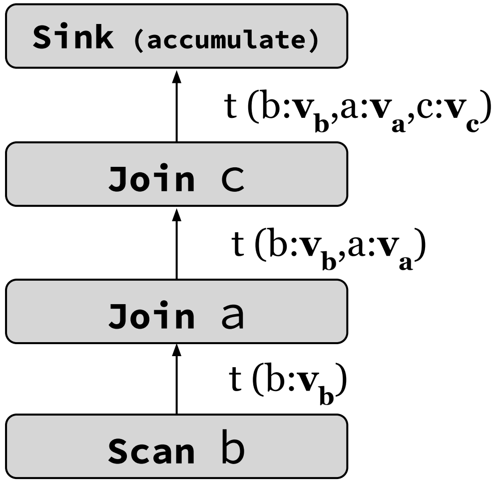
b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n
...	...	...
n	n-1	n+1
n	n-1	...
n	n-1	2n

n^2 tuples

n tuples

n tuples

n tuples



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

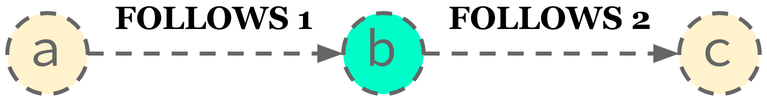
Join a

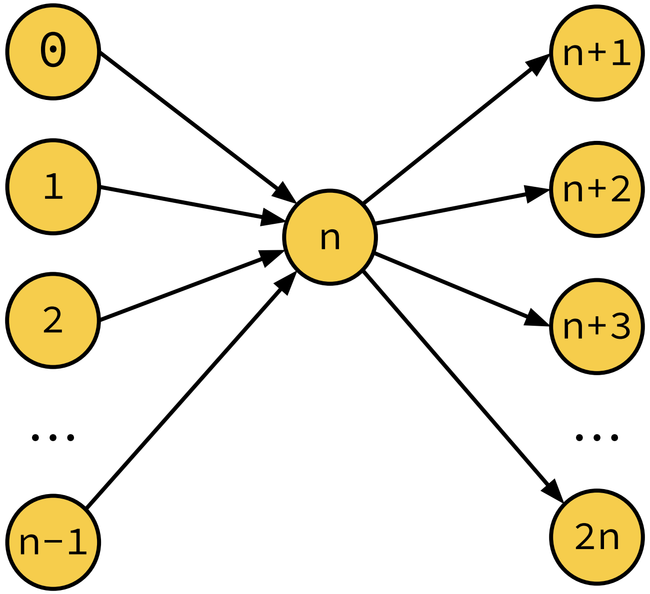
t (b:$\mathbf{v}_b$)

Scan b

Flat Representation Example

Query Vertex Ordering:
[b, a, c]





Output Relation

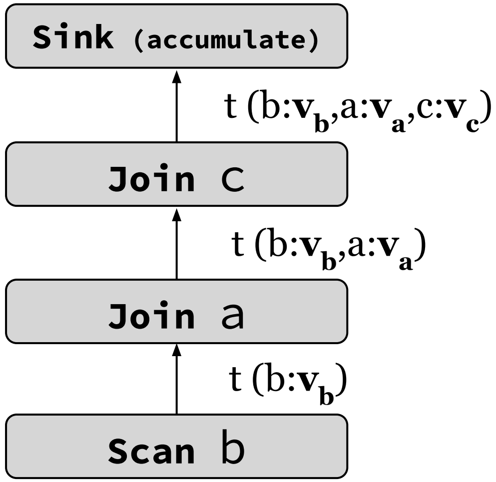
b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n
...	...	...
n	n-1	n+1
n	n-1	...
n	n-1	2n

n^2 tuples

n tuples

n tuples

n tuples



Sink (accumulate)

t $(b{:}\mathbf{v_b}, a{:}\mathbf{v_a}, c{:}\mathbf{v_c})$

Join c

t $(b{:}\mathbf{v_b}, a{:}\mathbf{v_a})$

Join a

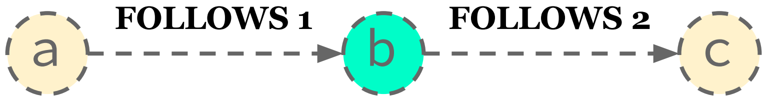
t $(b{:}\mathbf{v_b})$

Scan b

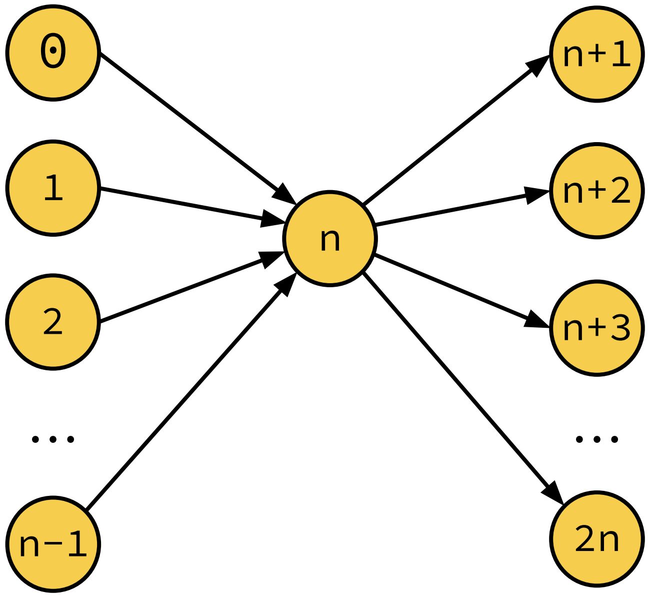
Flat Representation Example

Query Vertex Ordering:
[b, a, c]

a and **c** are conditionally independent on **b**!



For a fixed **b** value, a change in **a** values does not change **c** values and vice-versa.



Output Relation

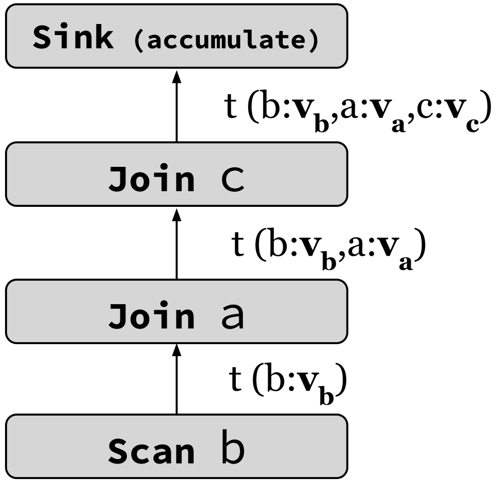
b.ID	a.ID	c.ID
n	0	n+1
n	0	...
n	0	2n
n	1	n+1
n	1	...
n	1	2n
...	...	...
n	n-1	n+1
n	n-1	...
n	n-1	2n

n tuples

n tuples

n tuples

n^2 tuples



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

Join a

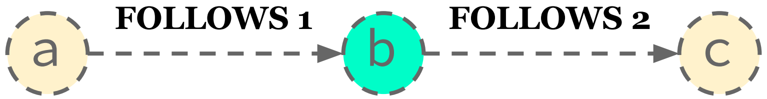
$t\ (b{:}\mathbf{v}_b)$

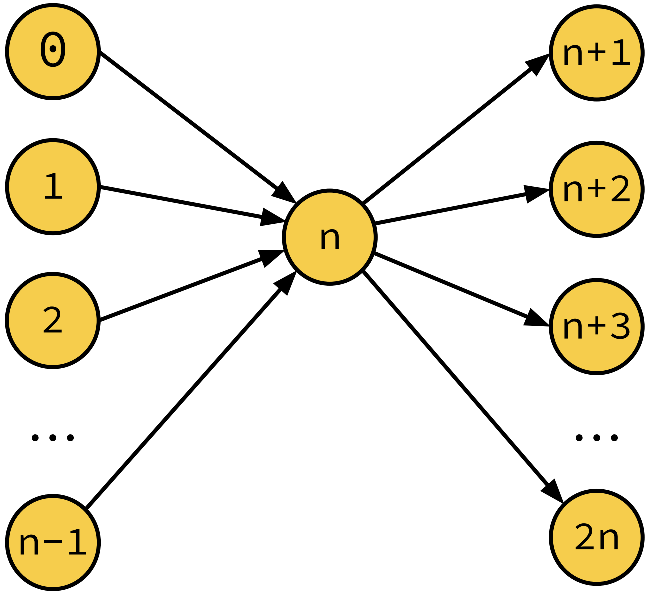
Scan b

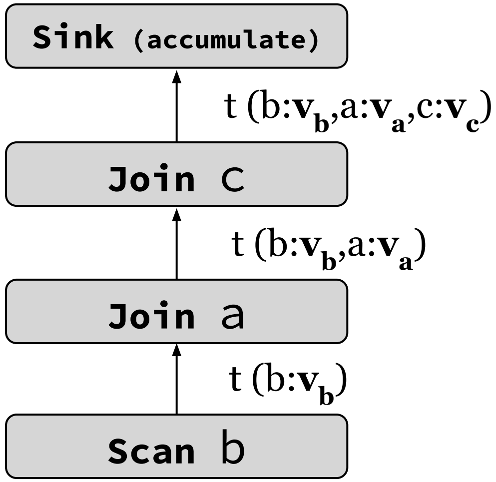
Factorized Representation Example

Query Vertex Ordering:
[b, a, c]

a and **c** are conditionally independent on **b**!







Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

Join a

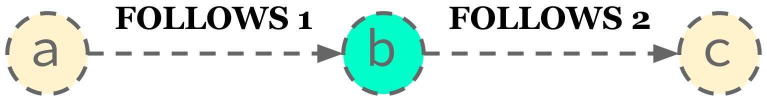
t (b:$\mathbf{v}_b$)

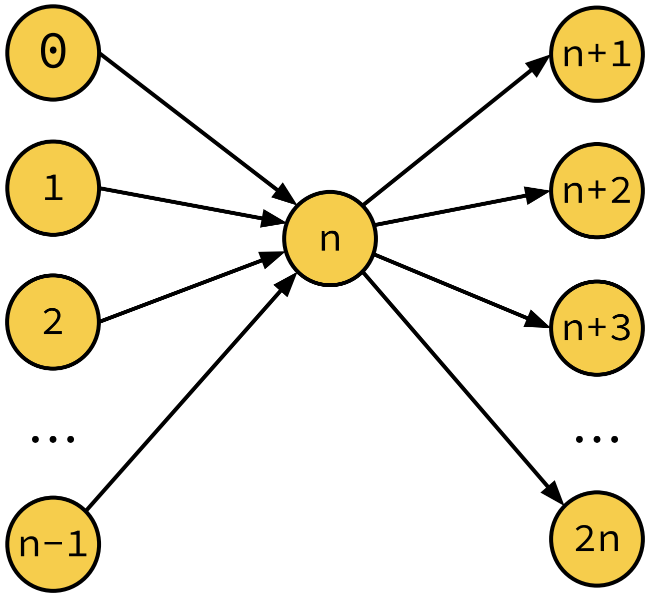
Scan b

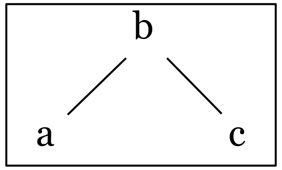
Factorized Representation Example

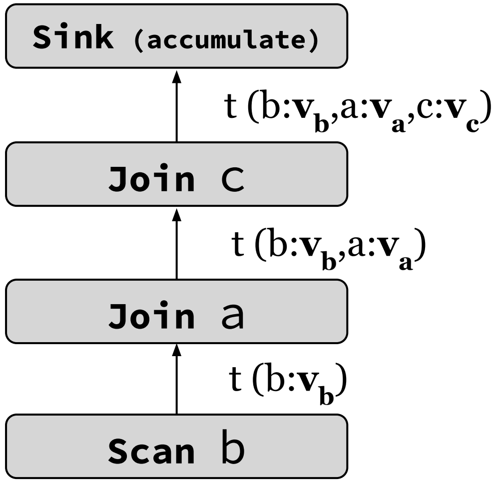
Query Vertex Ordering:
[b, a, c]

`a and c are conditionally independent on b!`









`Sink (accumulate)`

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

`Join c`

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

`Join a`

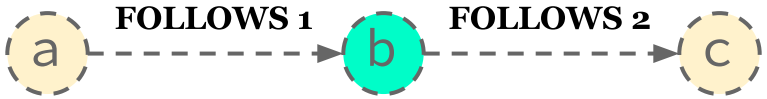
$t\,(b{:}\mathbf{v}_b)$

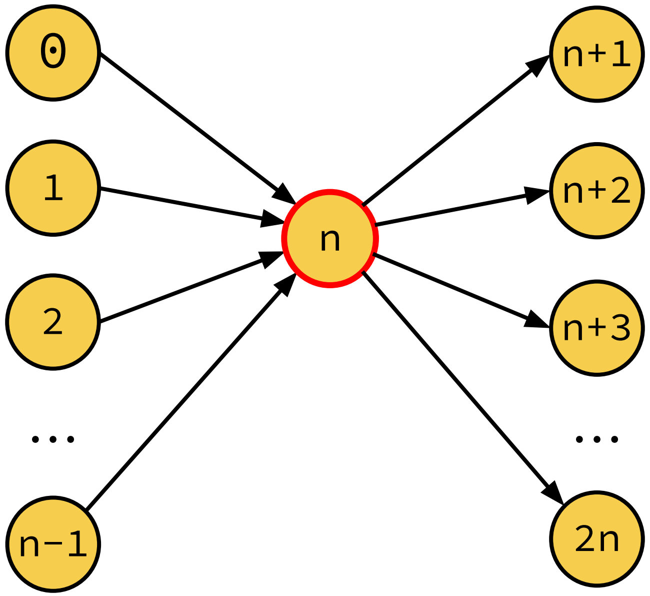
`Scan b`

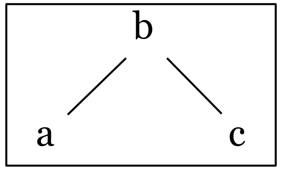
Factorized Representation Example

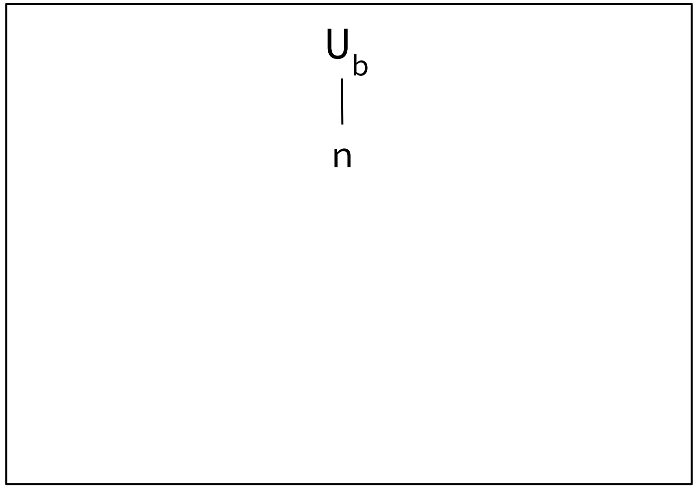
Query Vertex Ordering:
[b, a, c]

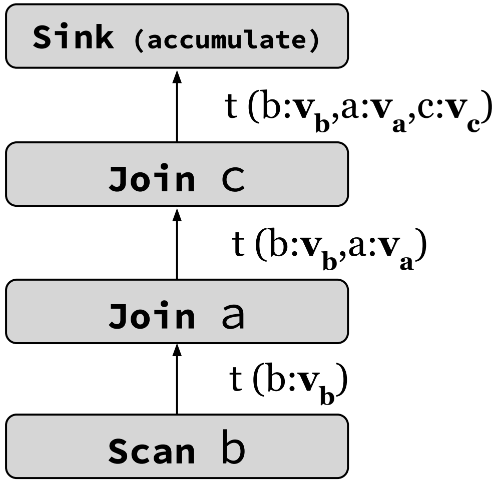
a and **c** are conditionally independent on **b**!











Sink (accumulate)

t (b:$\mathbf{v_b}$,a:$\mathbf{v_a}$,c:$\mathbf{v_c}$)

Join c

t (b:$\mathbf{v_b}$,a:$\mathbf{v_a}$)

Join a

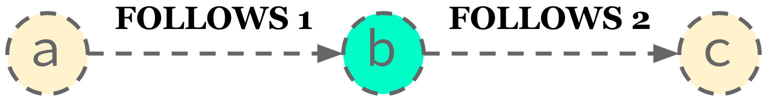
t (b:$\mathbf{v_b}$)

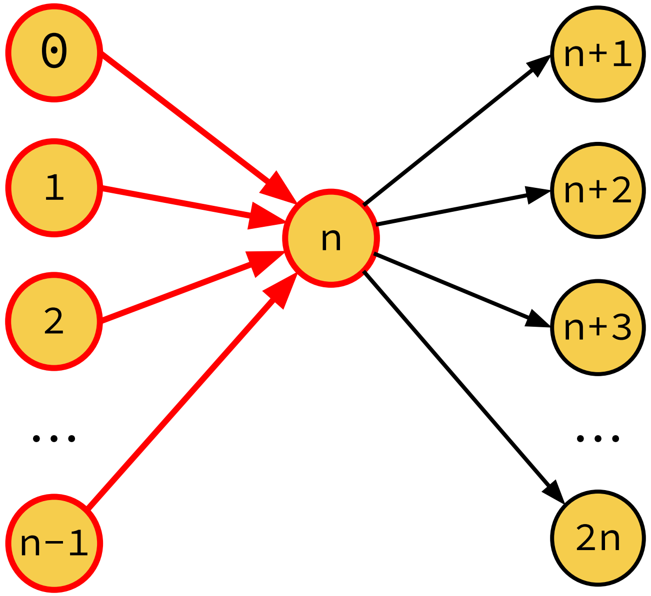
Scan b

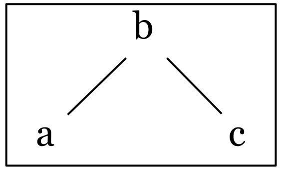
Factorized Representation Example

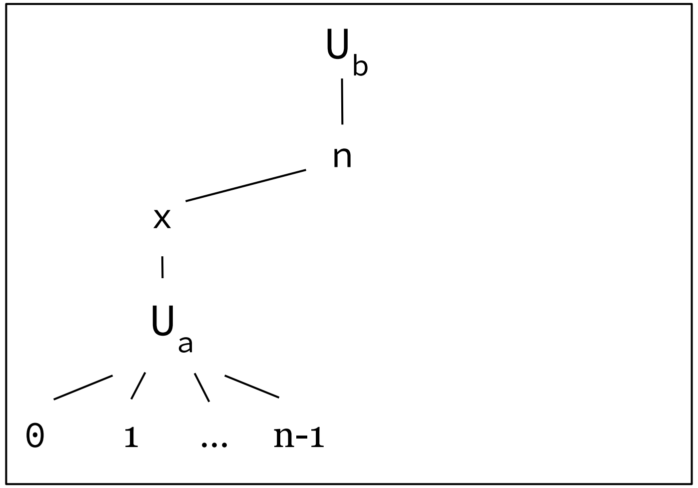
Query Vertex Ordering:
[b, a, c]

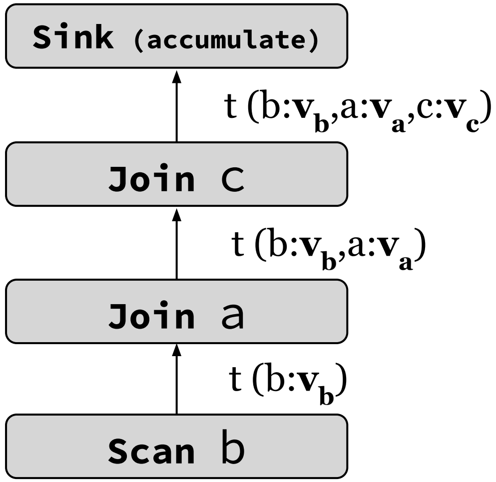
a and **c** are conditionally independent on **b**!











Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

Join a

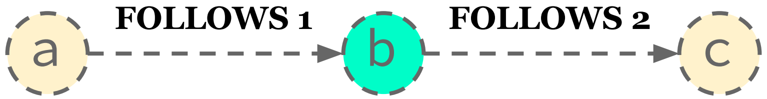
$t\ (b{:}\mathbf{v}_b)$

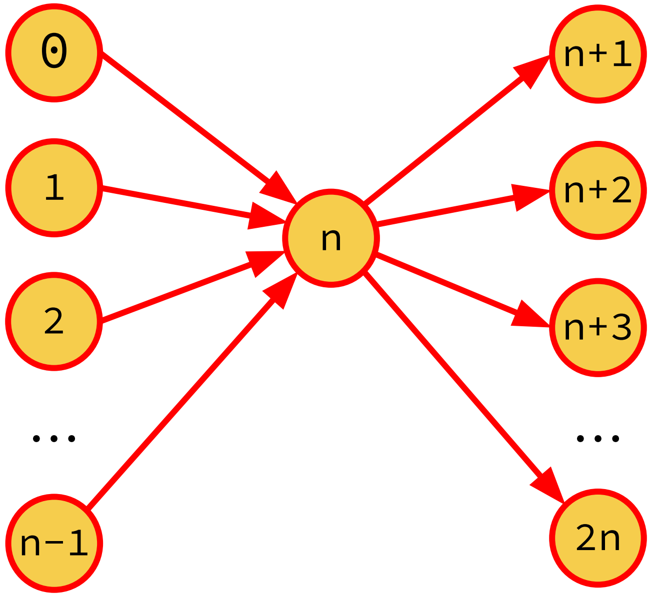
Scan b

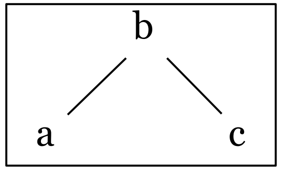
Factorized Representation Example

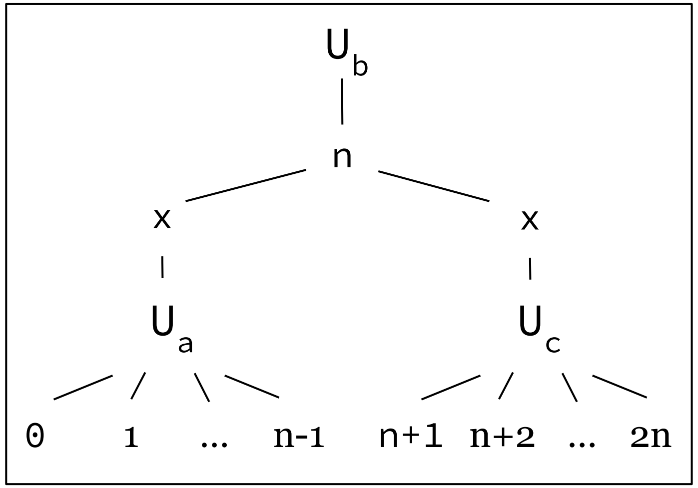
Query Vertex Ordering:
[b, a, c]

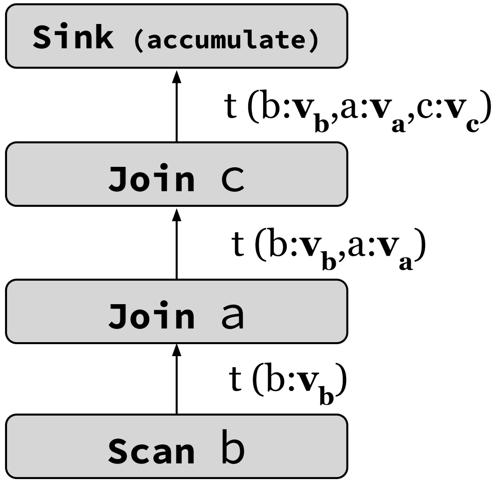
`a and c are conditionally independent on b!`







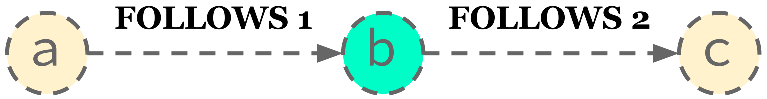


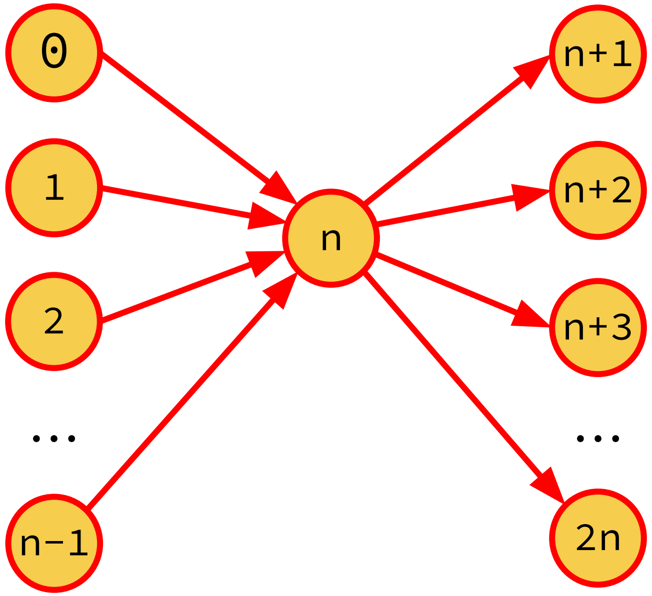


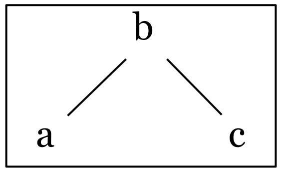
Factorized Representation Example

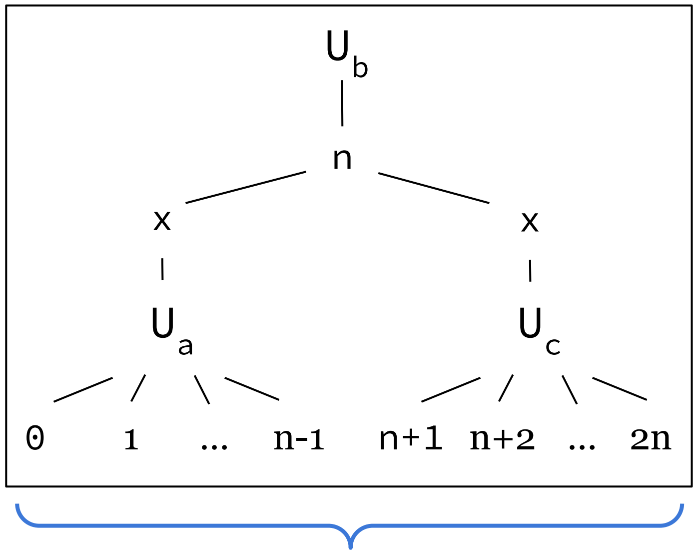
Query Vertex Ordering:
[b, a, c]

a and **c** are conditionally independent on **b**!

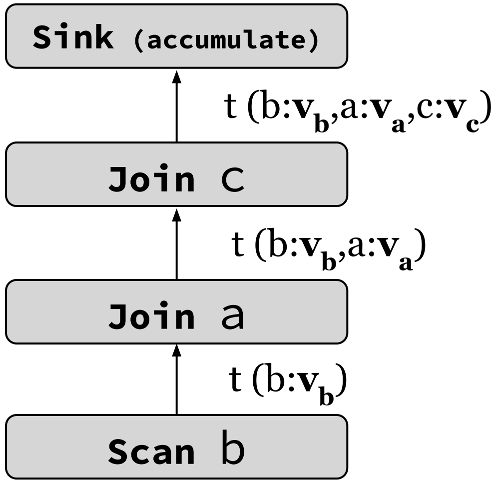








2n+1 fields



Sink (accumulate)

t $(b{:}\mathbf{v_b}, a{:}\mathbf{v_a}, c{:}\mathbf{v_c})$

Join c

t $(b{:}\mathbf{v_b}, a{:}\mathbf{v_a})$

Join a

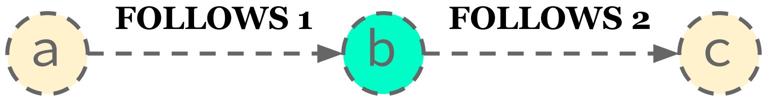
t $(b{:}\mathbf{v_b})$

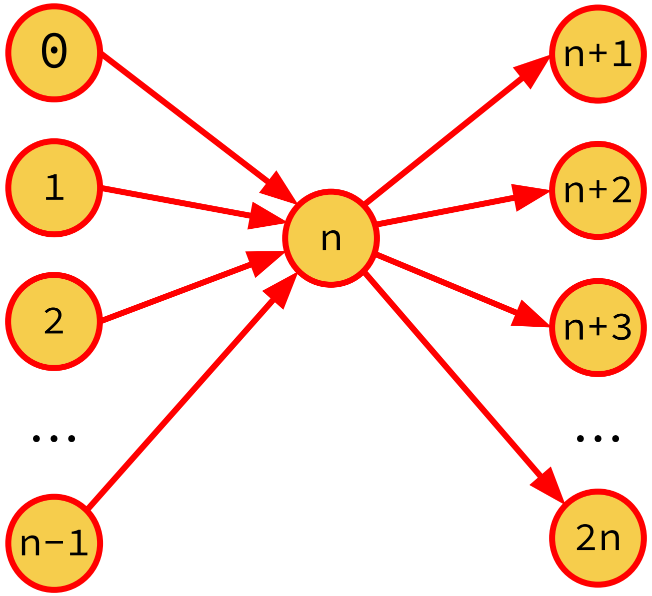
Scan b

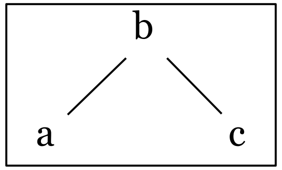
Factorized Representation Example

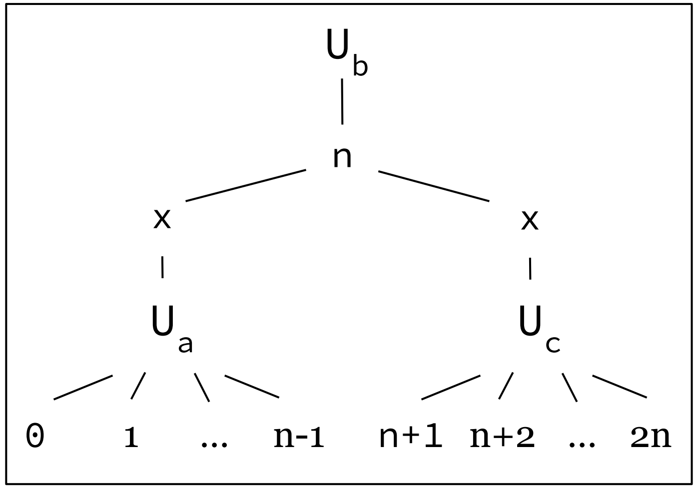
Query Vertex Ordering:
[b, a, c]

a and **c** are conditionally independent on **b**!

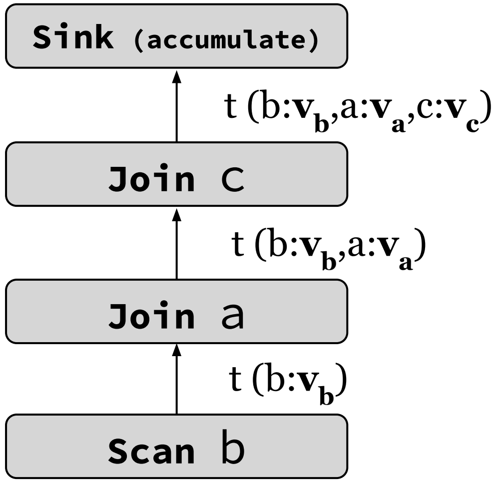








2n+1 fields
vs. flat
$3n^2$ fields



Sink (accumulate)

t ($b{:}\mathbf{v_b}, a{:}\mathbf{v_a}, c{:}\mathbf{v_c}$)

Join c

t ($b{:}\mathbf{v_b}, a{:}\mathbf{v_a}$)

Join a

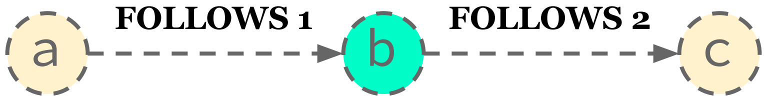
t ($b{:}\mathbf{v_b}$)

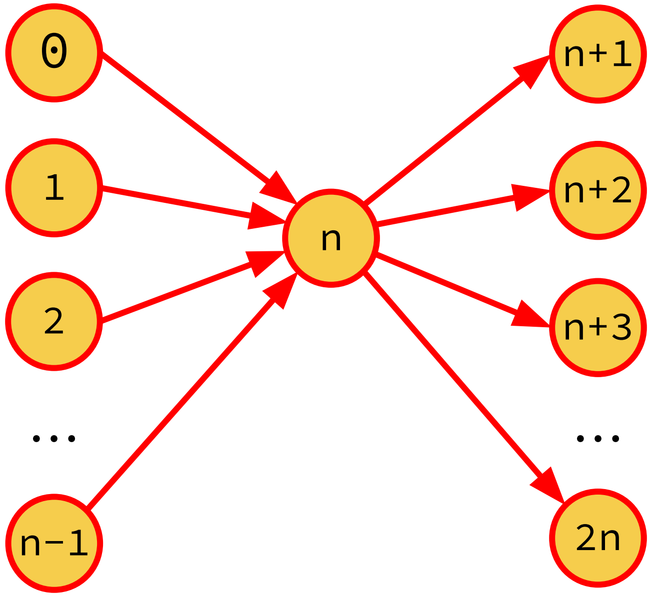
Scan b

Factorized Representation Example

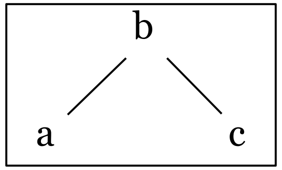
Query Vertex Ordering:
[b, a, c]

a and **c** are conditionally independent on **b**!

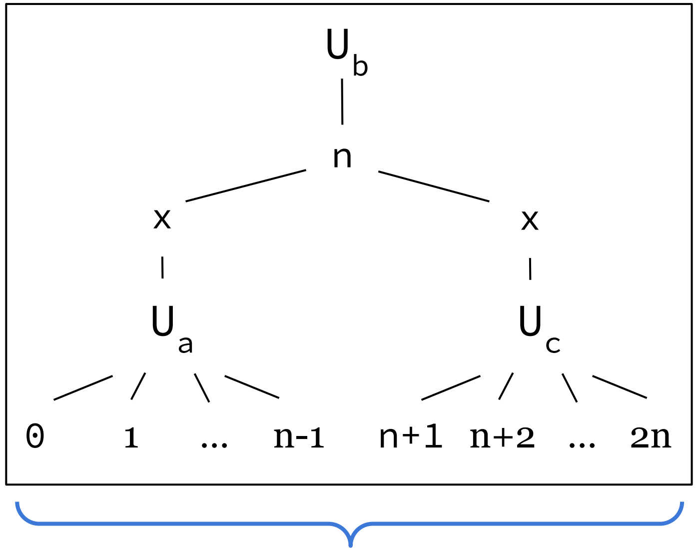




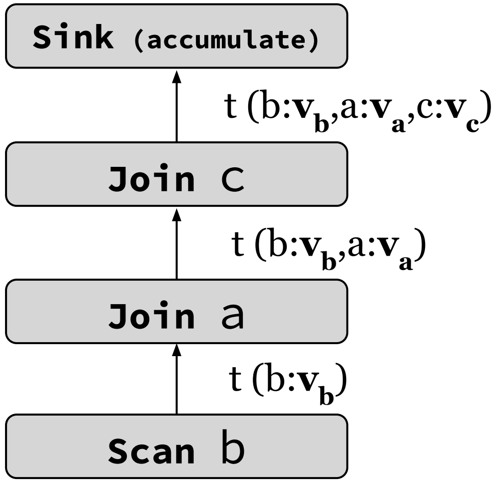
f-tree



f-representation



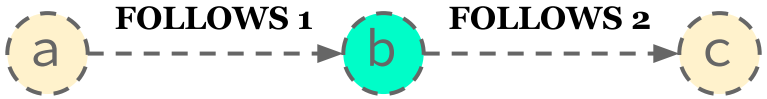
2n+1 fields
vs. flat
$3n^2$ fields

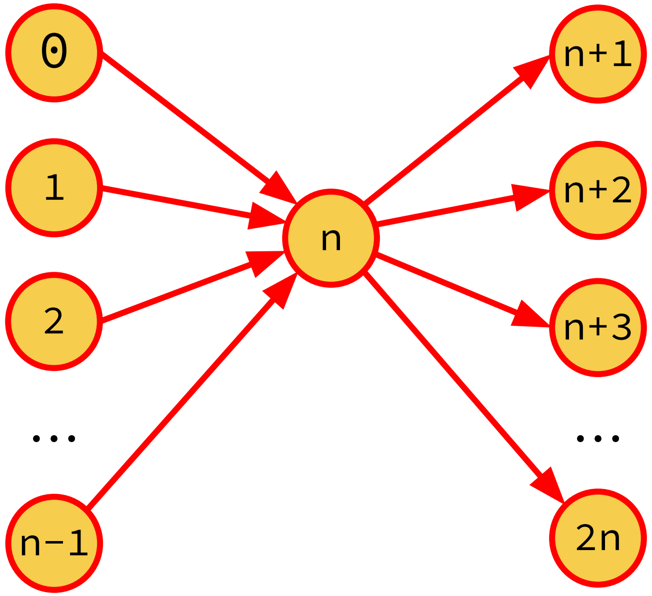


Factorized Representation Example

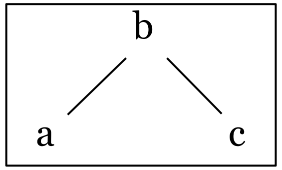
Query Vertex Ordering:
[b, a, c]

a and **c** are conditionally independent on **b**!

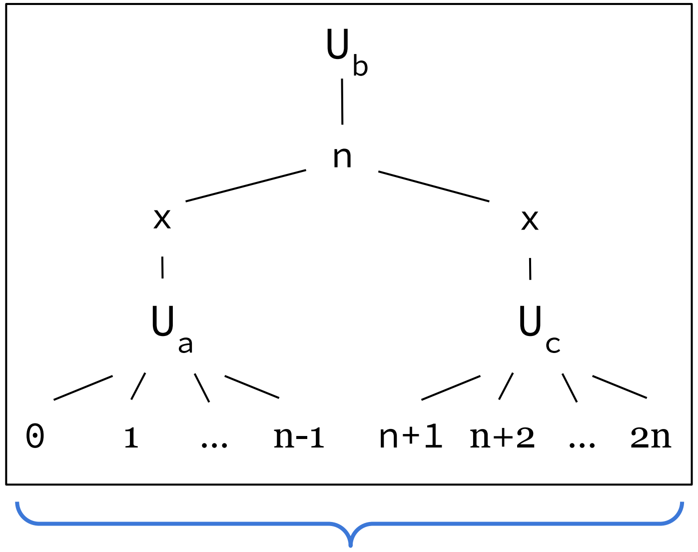




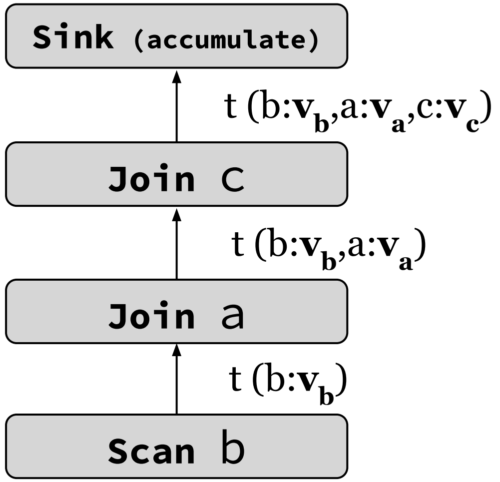
f-tree:
attribute-level description of the f-representation



f-representation



2n+1 fields
vs. flat
$3n^2$ fields



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

Join a

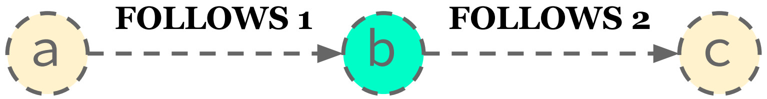
t (b:$\mathbf{v}_b$)

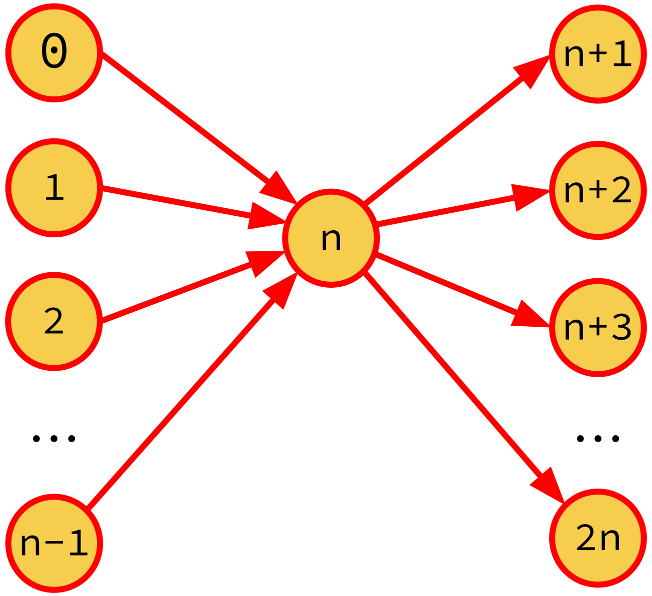
Scan b

Factorized Representation Example

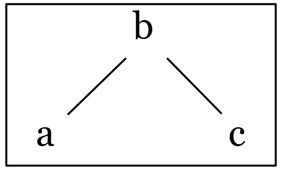
Query Vertex Ordering:
[b, a, c]

a and **c** are conditionally independent on **b**!

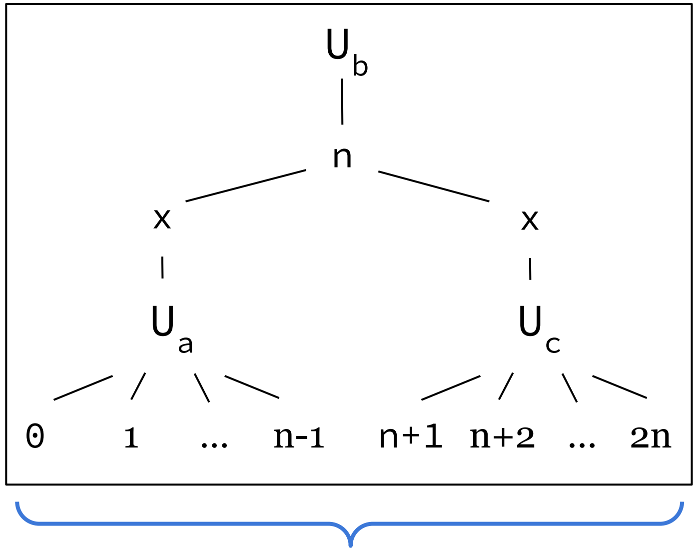




f-tree



f-representation

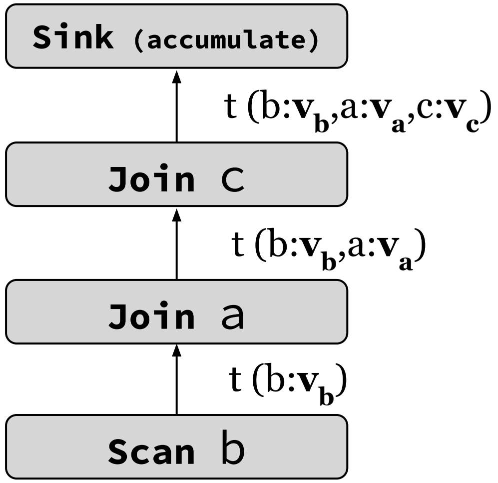


2n+1 fields
vs. flat
$3n^2$ fields

Theory of factorization establishes:
$\sigma(Q) \leq AGM(Q)$ where

$\sigma(Q)$: worst-case size bound over f-representations.

In some cases, $\sigma(Q) < AGM(Q)$



Sink (accumulate)

t $(b{:}\mathbf{v_b}, a{:}\mathbf{v_a}, c{:}\mathbf{v_c})$

Join c

t $(b{:}\mathbf{v_b}, a{:}\mathbf{v_a})$

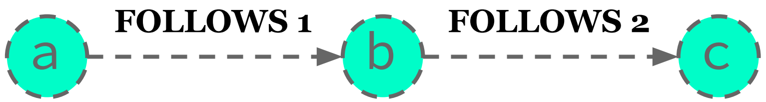
Join a

t $(b{:}\mathbf{v_b})$

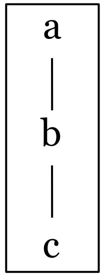
Scan b

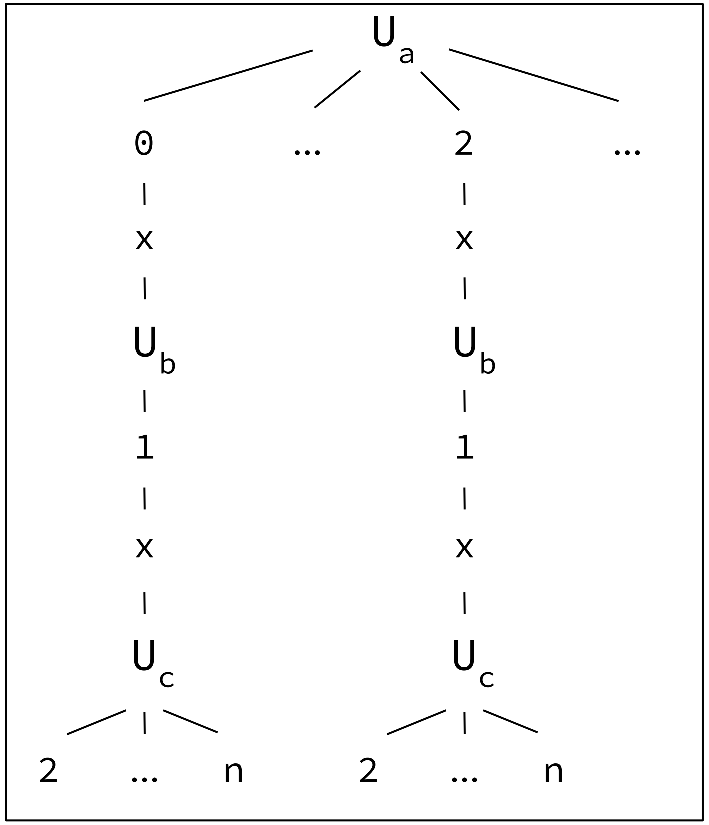
What Are The Possible F-Representations for a Query Q?

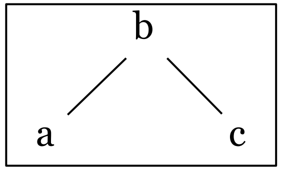
What Are The Possible F-Representations for a Query Q?

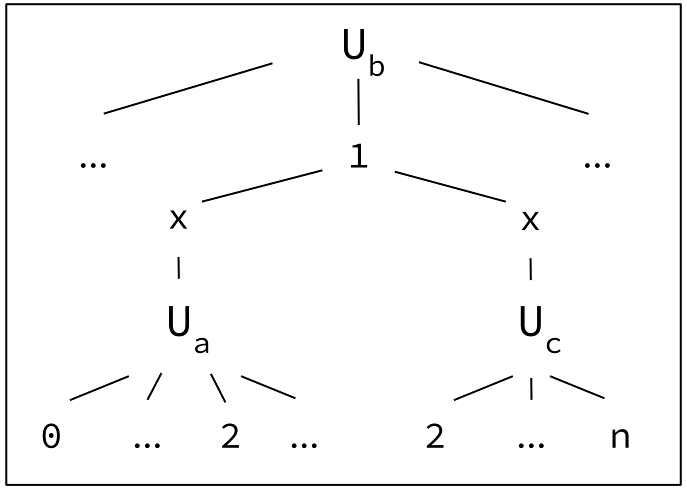


f-representation examples over f-trees

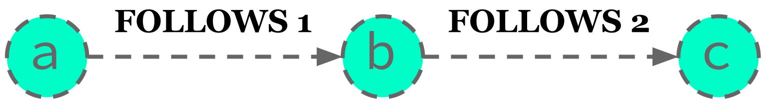




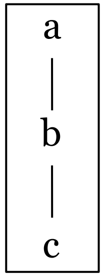


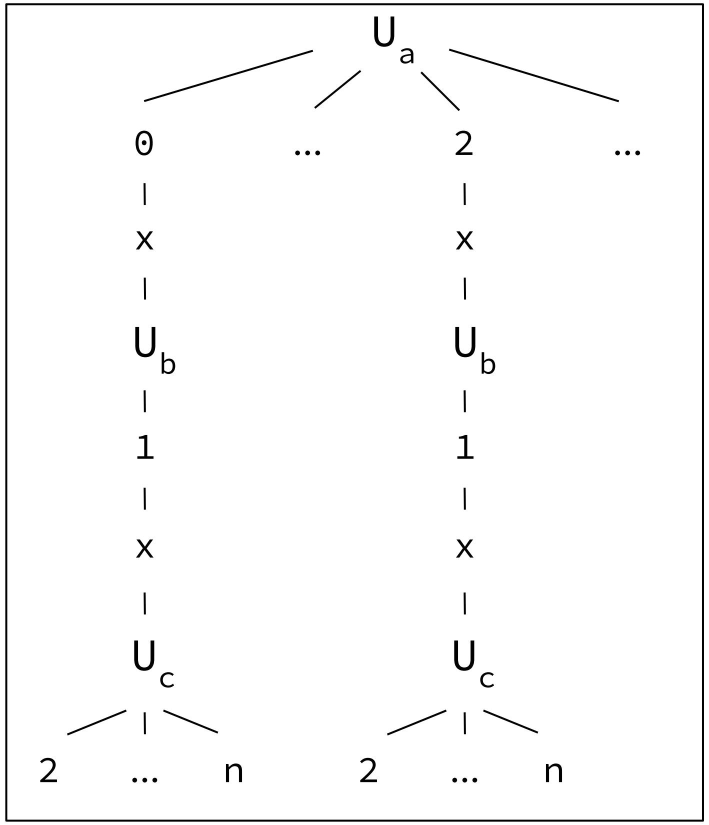


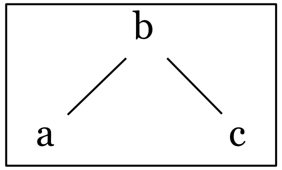
What Are The Possible F-Representations for a Query Q?

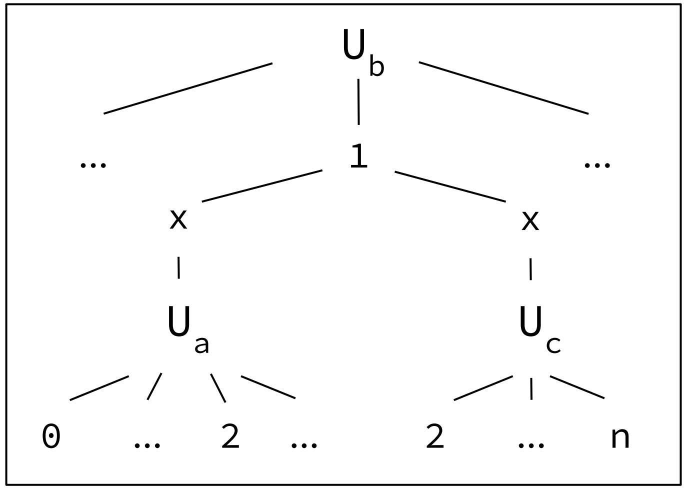


f-representation examples over f-trees





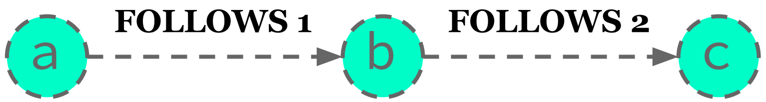




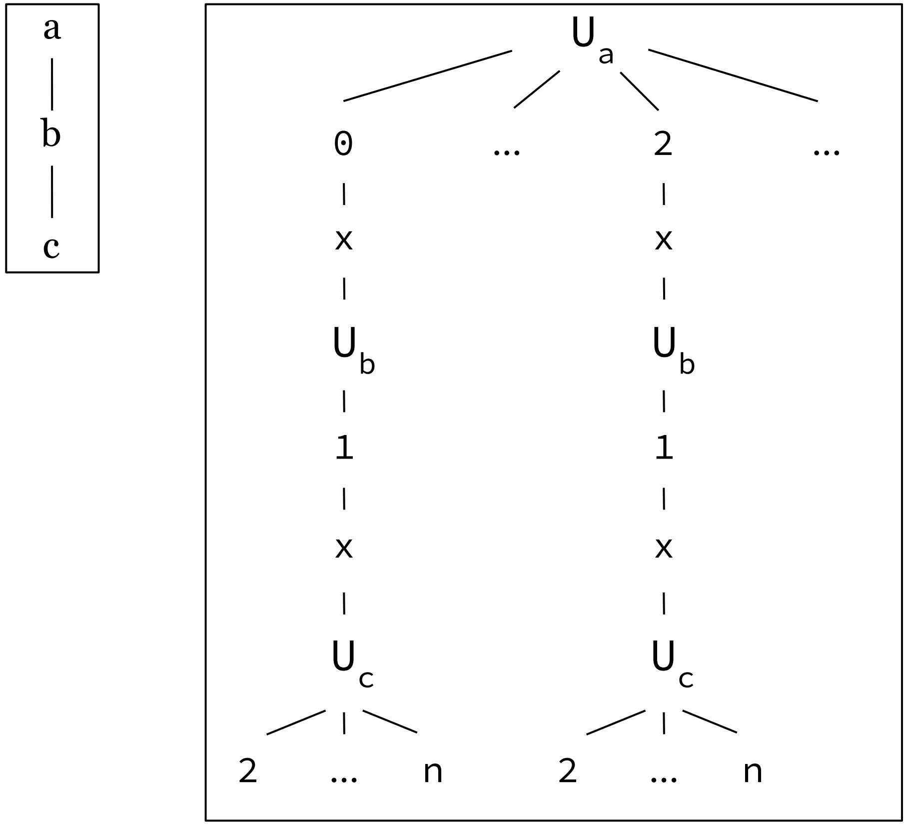
F-tree validity:

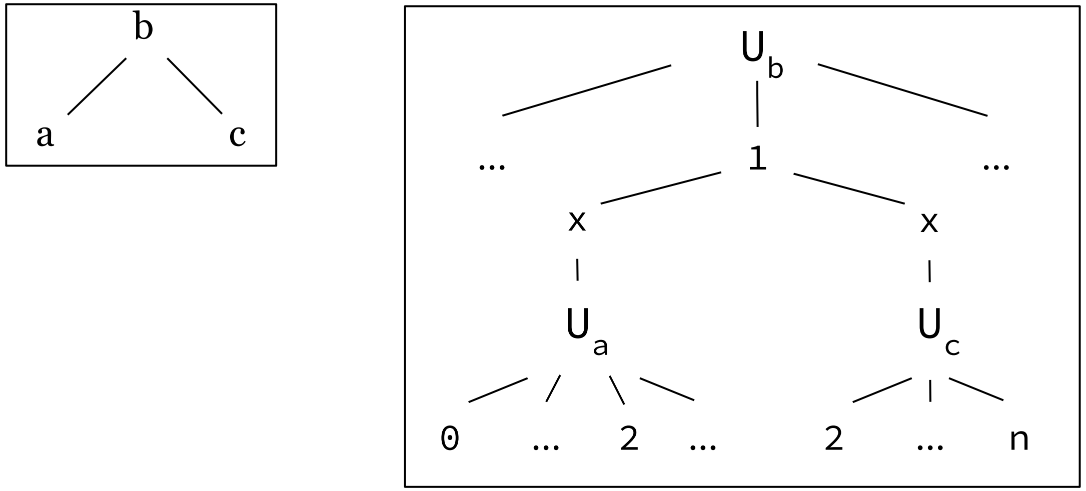
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree.

What Are The Possible F-Representations for a Query Q?



f-representation examples over f-trees



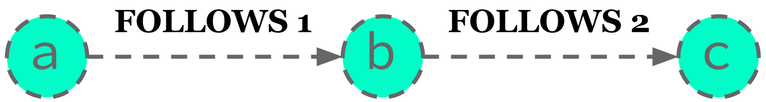


<u>F-tree validity</u>:

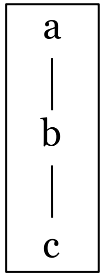
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:

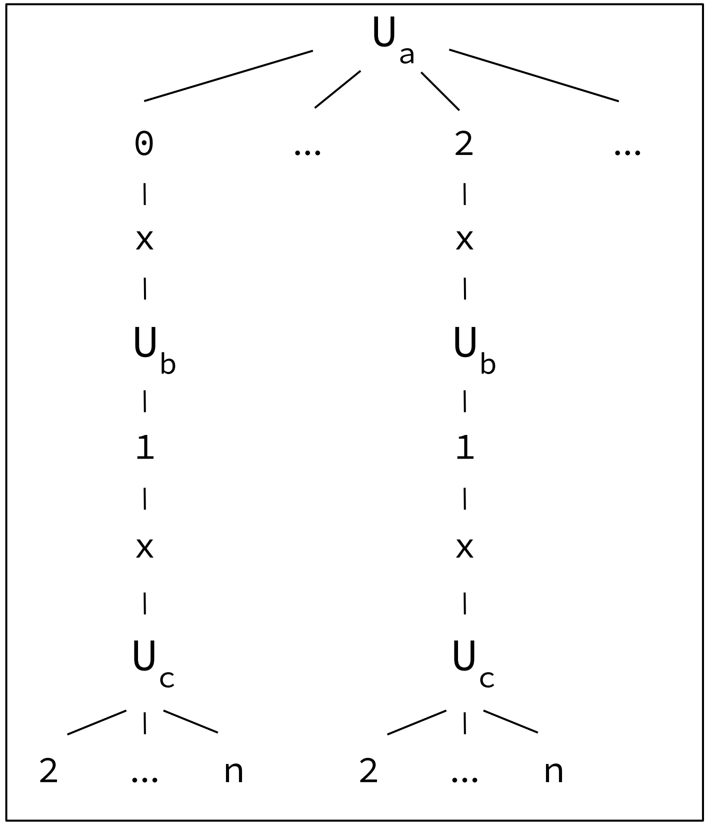
- Any query vertex pair connected by a query edge are Q-dependent.

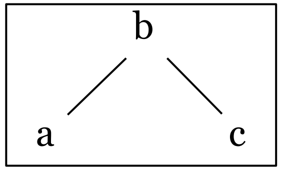
What Are The Possible F-Representations for a Query Q?

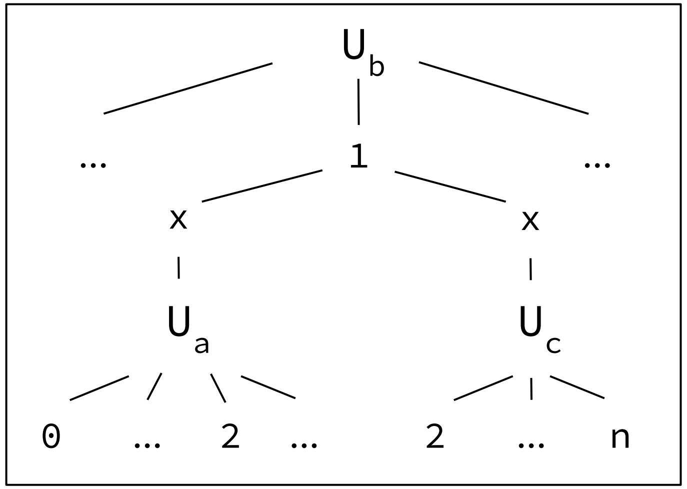


f-representation examples over f-trees







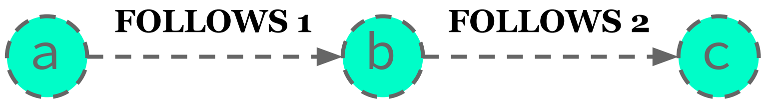


<u>F-tree validity</u>:

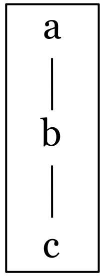
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:

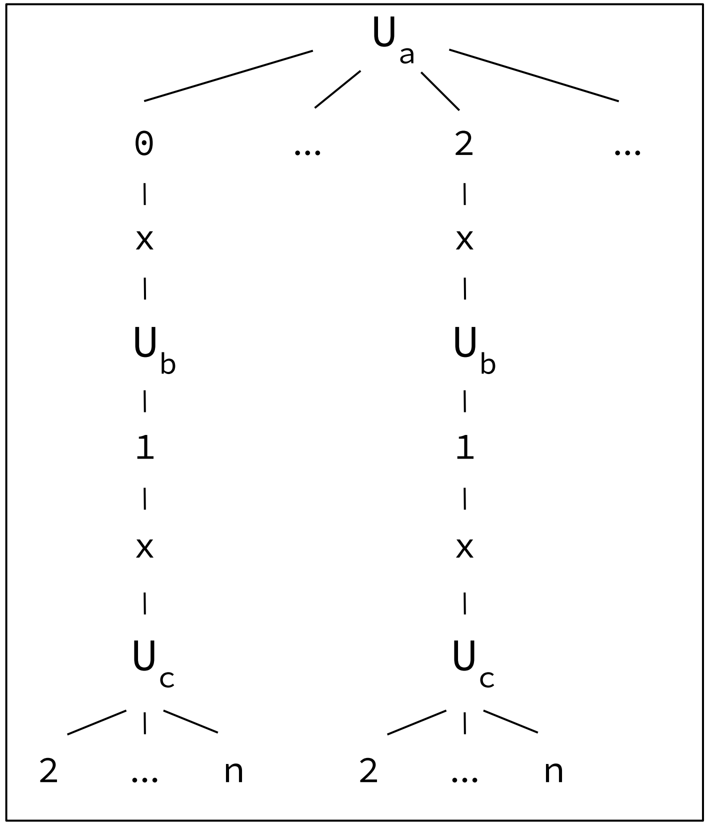
- Any query vertex pair connected by a query edge are Q-dependent.
- Any query vertex/edge pair are dependent based on theta joins.

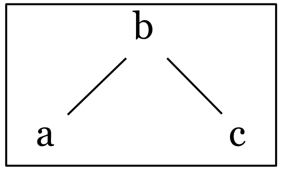
What Are The Possible F-Representations for a Query Q?



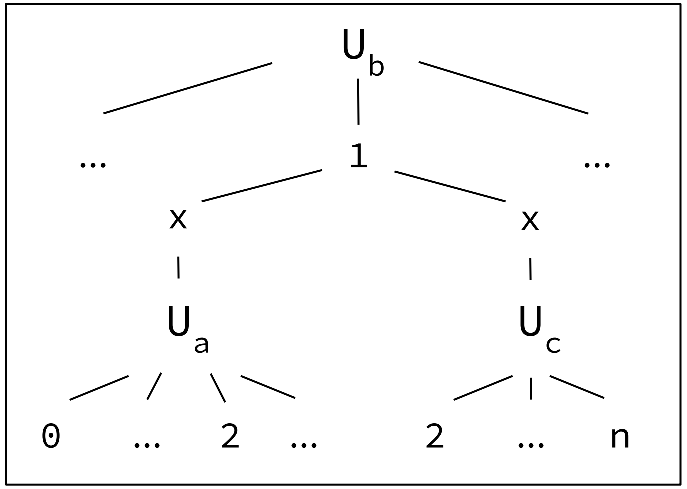
f-representation examples over f-trees







`WHERE a.p < c.p`

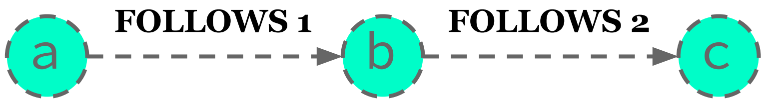


F-tree validity:

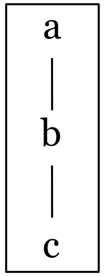
Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:

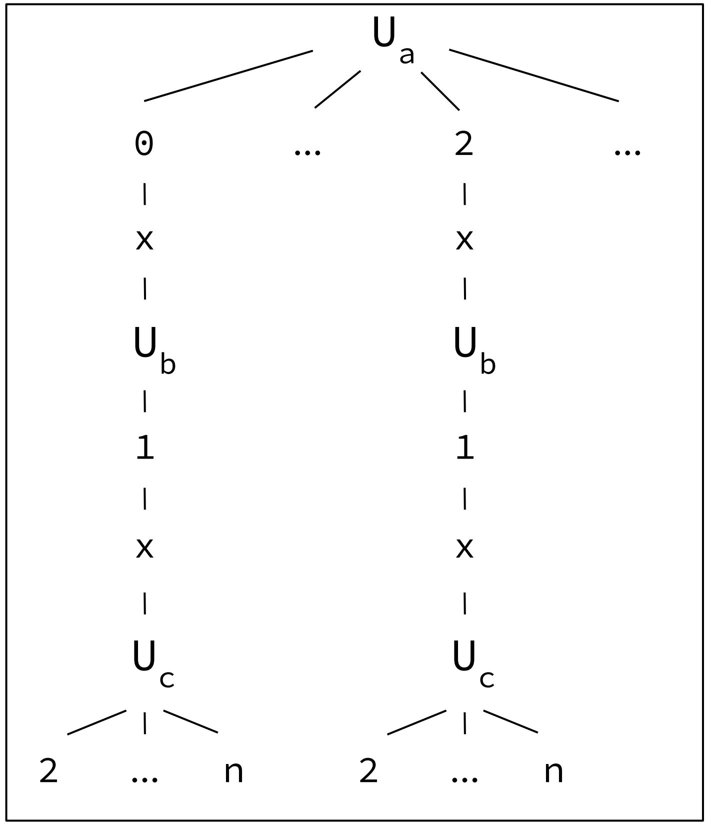
- Any query vertex pair connected by a query edge are Q-dependent.
- Any query vertex/edge pair are dependent based on theta joins.

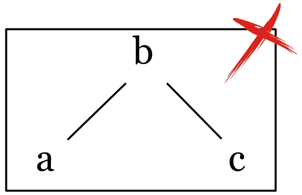
What Are The Possible F-Representations for a Query Q?



f-representation examples over f-trees

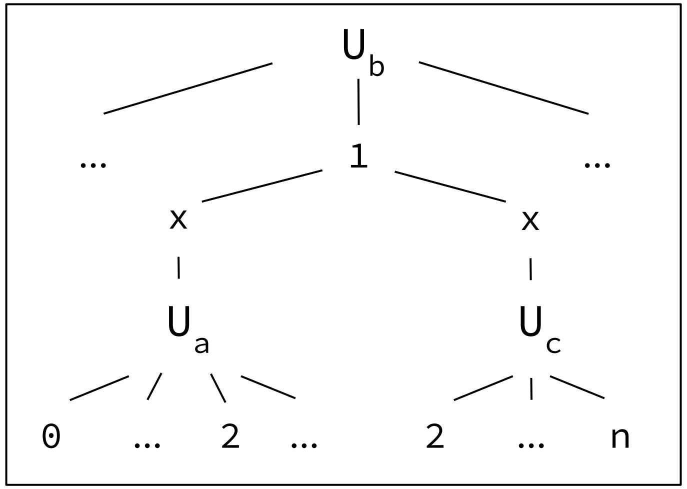






WHERE a.p < c.p

a and b are Q-dependent!!
f-tree is not valid



<u>F-tree validity</u>:

Any two Q-dependent attributes lie along a root-to-leaf path in an f-tree:

- Any query vertex pair connected by a query edge are Q-dependent.
- Any query vertex/edge pair are dependent based on theta joins.

Outline of Query Processing Techniques to Cover

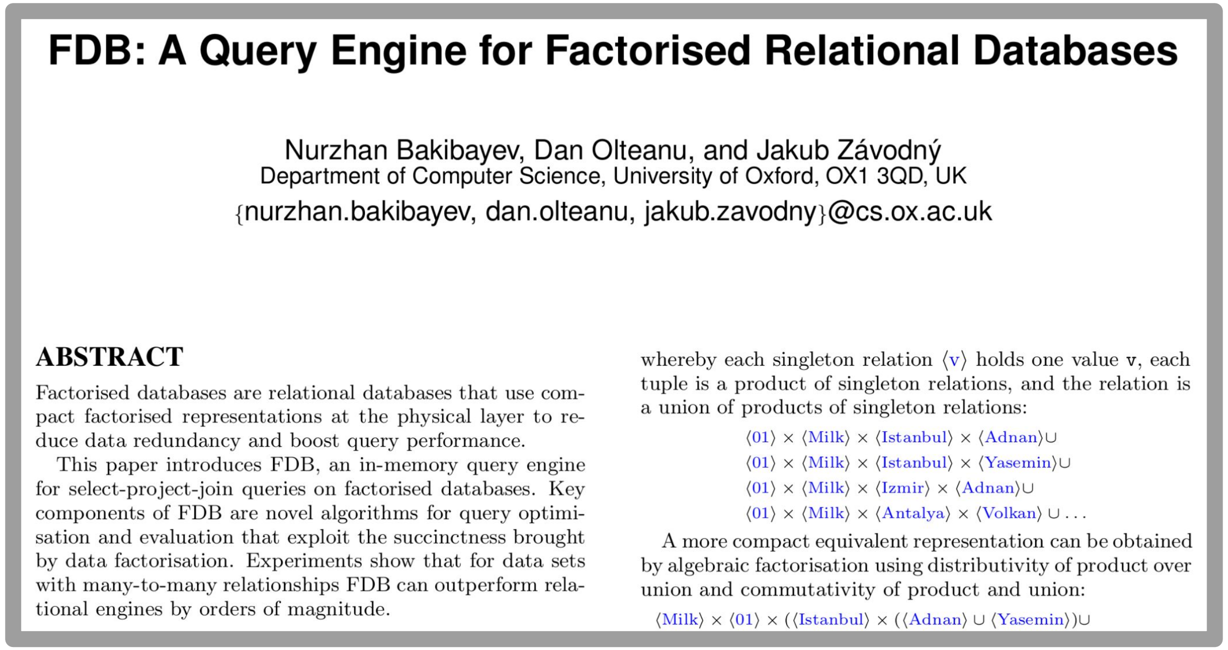
For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

1) **Predefined Joins**

2) **Worst-case optimal joins**

3) **Factorized Query Processing**
   - **3.1. Foundations: Factorized Representations**
   - **3.2. System Integration Approaches: FDB and Factorized Vector Execution in Graphflow**

FDB: Factorized Database Engine



FDB: A Query Engine for Factorised Relational Databases

Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný
Department of Computer Science, University of Oxford, OX1 3QD, UK
{nurzhan.bakibayev, dan.olteanu, jakub.zavodny}@cs.ox.ac.uk

ABSTRACT

Factorised databases are relational databases that use compact factorised representations at the physical layer to reduce data redundancy and boost query performance.

This paper introduces FDB, an in-memory query engine for select-project-join queries on factorised databases. Key components of FDB are novel algorithms for query optimisation and evaluation that exploit the succinctness brought by data factorisation. Experiments show that for data sets with many-to-many relationships FDB can outperform relational engines by orders of magnitude.

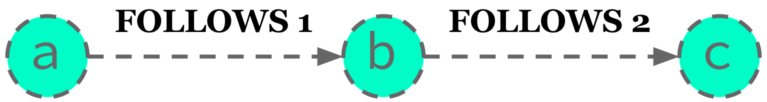
whereby each singleton relation $\langle \text{v} \rangle$ holds one value `v`, each tuple is a product of singleton relations, and the relation is a union of products of singleton relations:

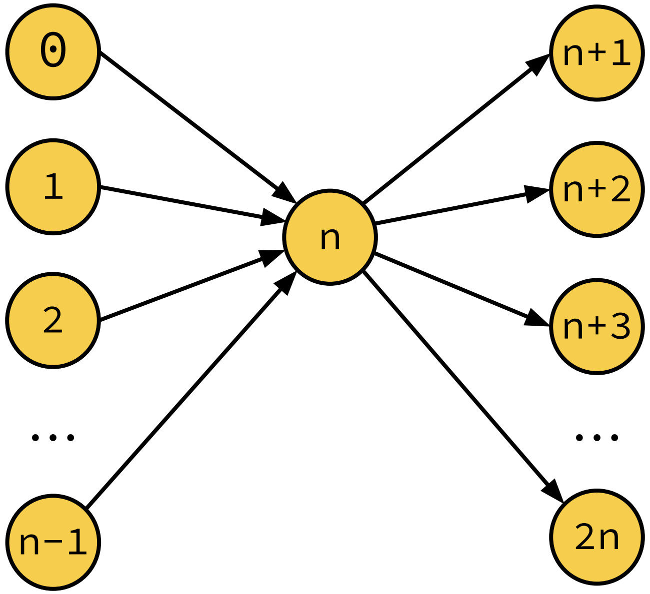
$$\begin{aligned}
&\langle \text{01} \rangle \times \langle \text{Milk} \rangle \times \langle \text{Istanbul} \rangle \times \langle \text{Adnan} \rangle \cup \\
&\langle \text{01} \rangle \times \langle \text{Milk} \rangle \times \langle \text{Istanbul} \rangle \times \langle \text{Yasemin} \rangle \cup \\
&\langle \text{01} \rangle \times \langle \text{Milk} \rangle \times \langle \text{Izmir} \rangle \times \langle \text{Adnan} \rangle \cup \\
&\langle \text{01} \rangle \times \langle \text{Milk} \rangle \times \langle \text{Antalya} \rangle \times \langle \text{Volkan} \rangle \cup \ldots
\end{aligned}$$

A more compact equivalent representation can be obtained by algebraic factorisation using distributivity of product over union and commutativity of product and union:

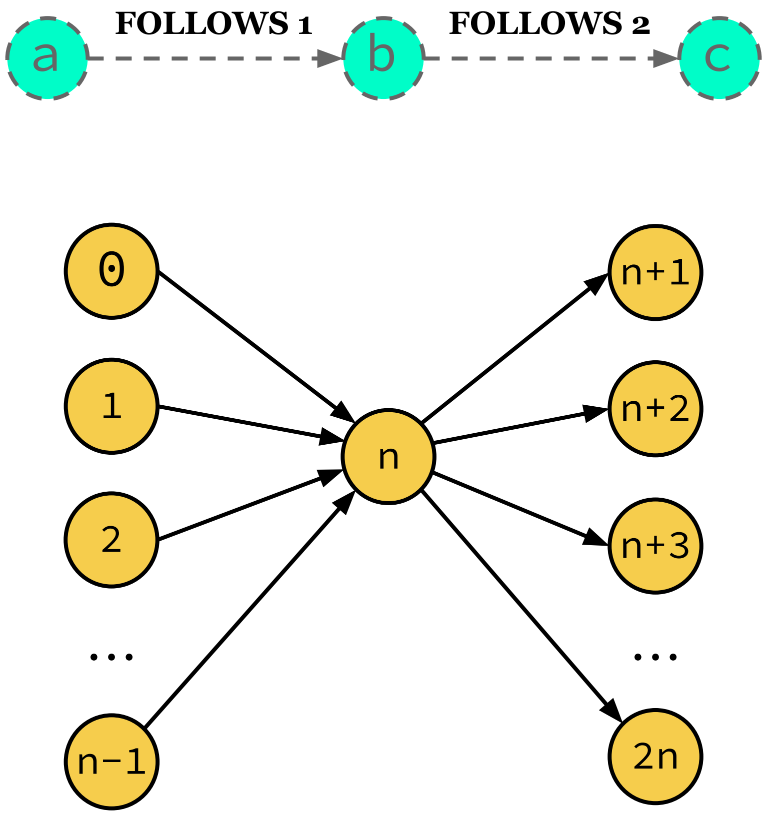
$$\langle \text{Milk} \rangle \times \langle \text{01} \rangle \times (\langle \text{Istanbul} \rangle \times (\langle \text{Adnan} \rangle \cup \langle \text{Yasemin} \rangle) \cup$$

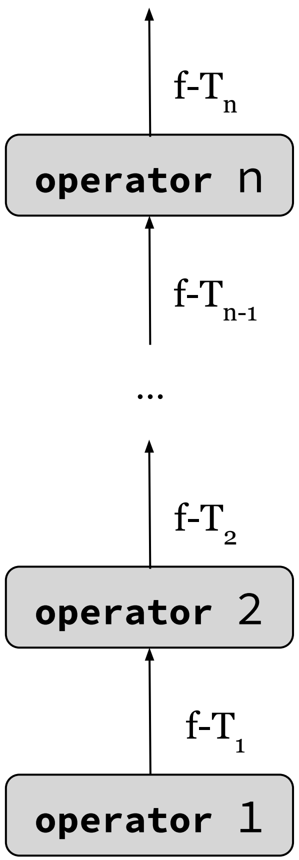
FDB: Factorized Database Engine





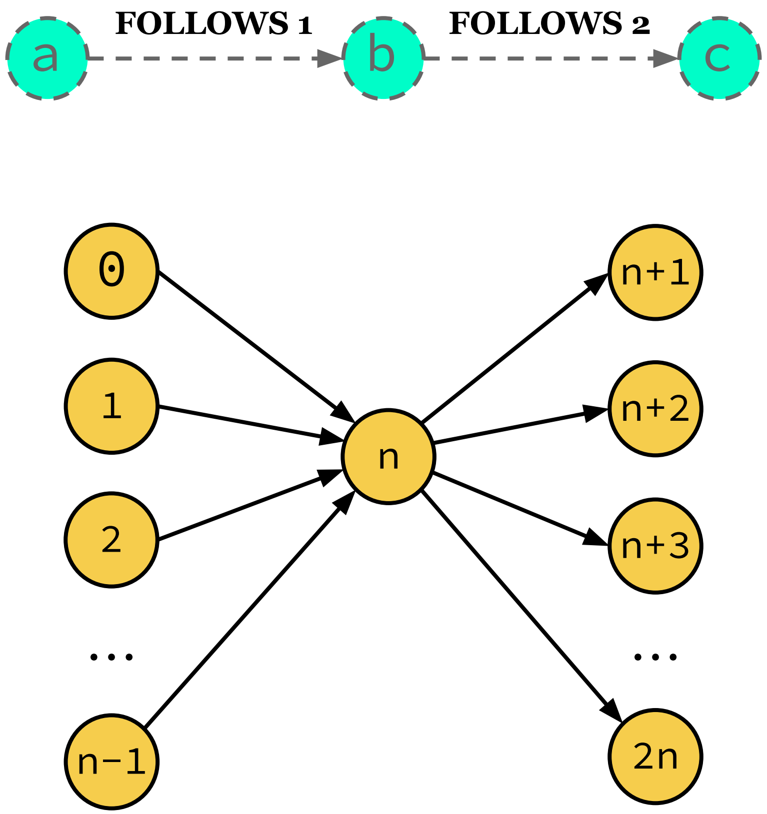
FDB: Factorized Database Engine

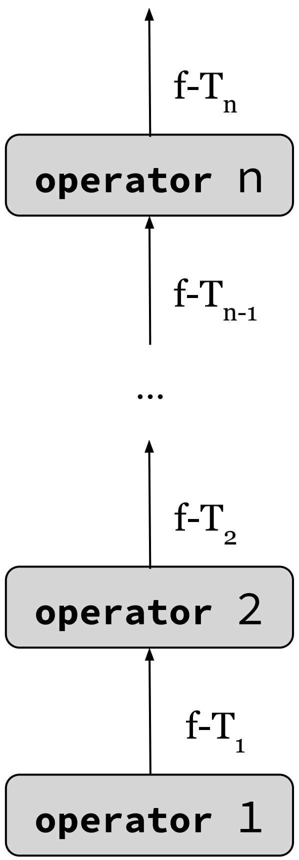


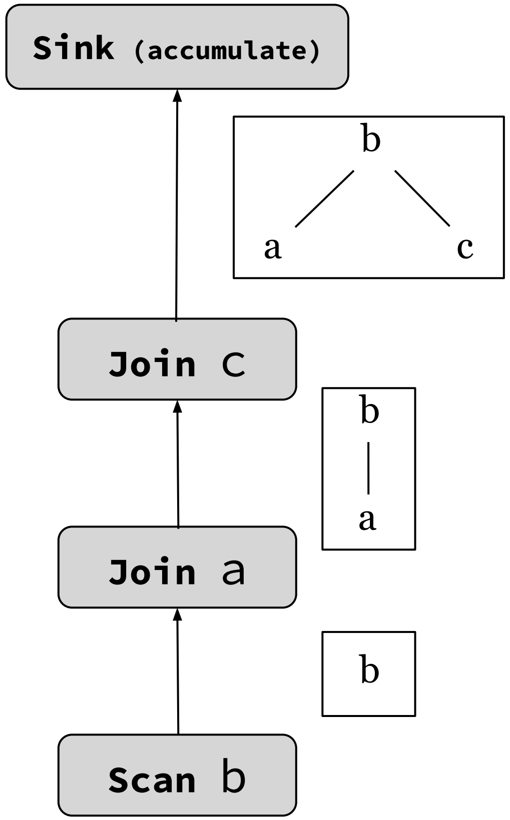


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

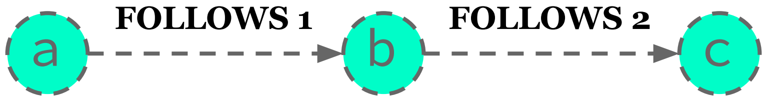


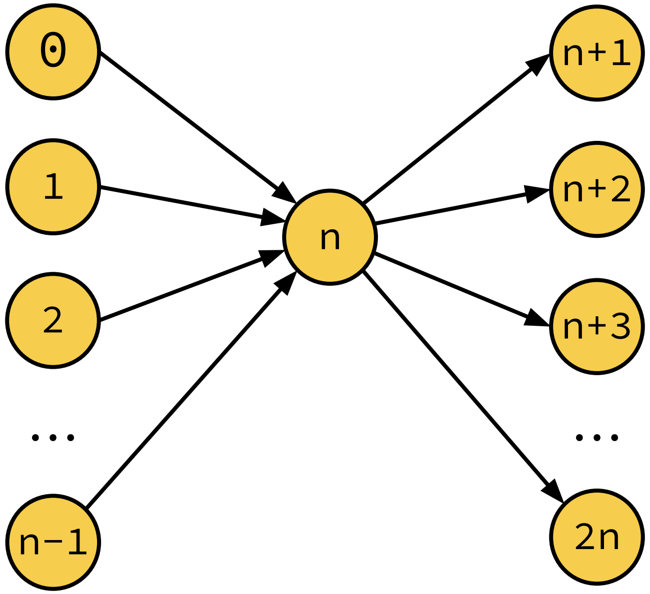


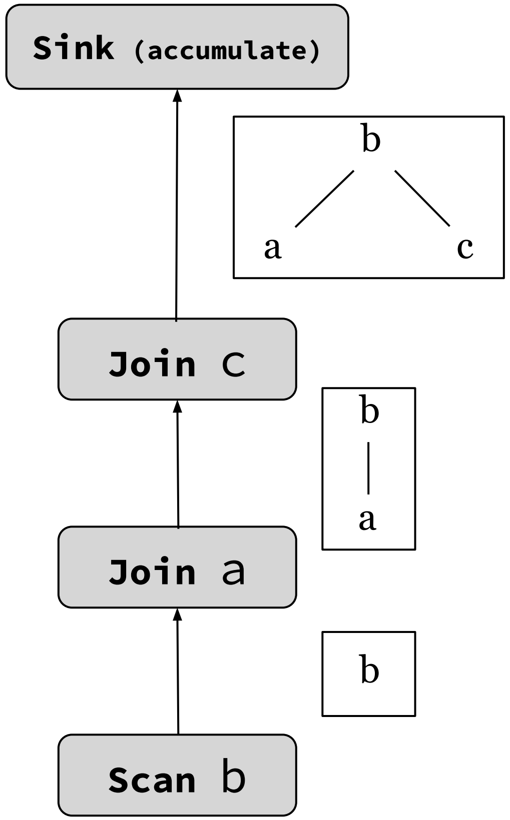


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

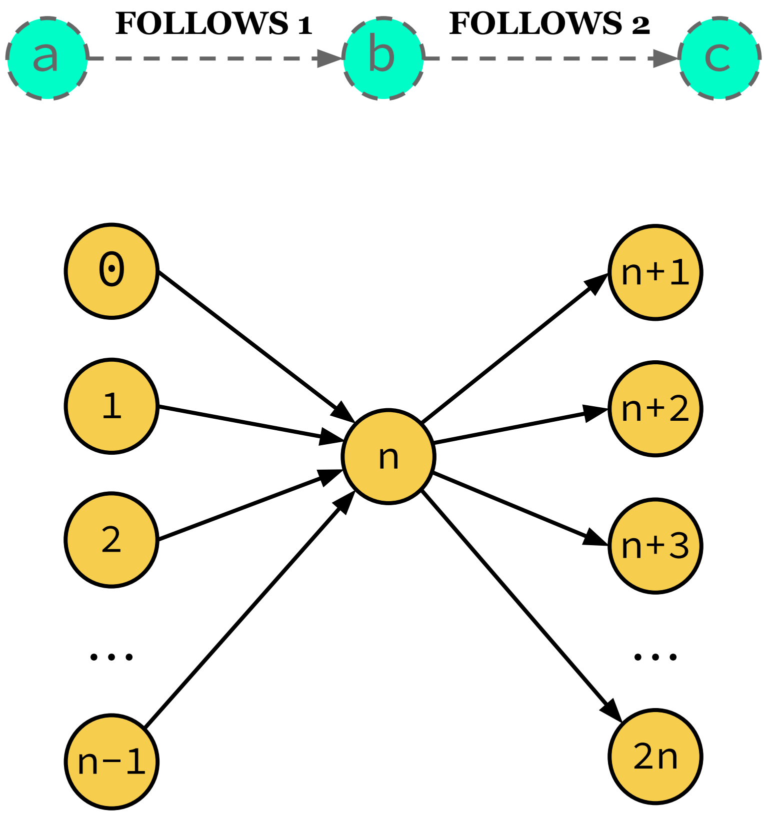


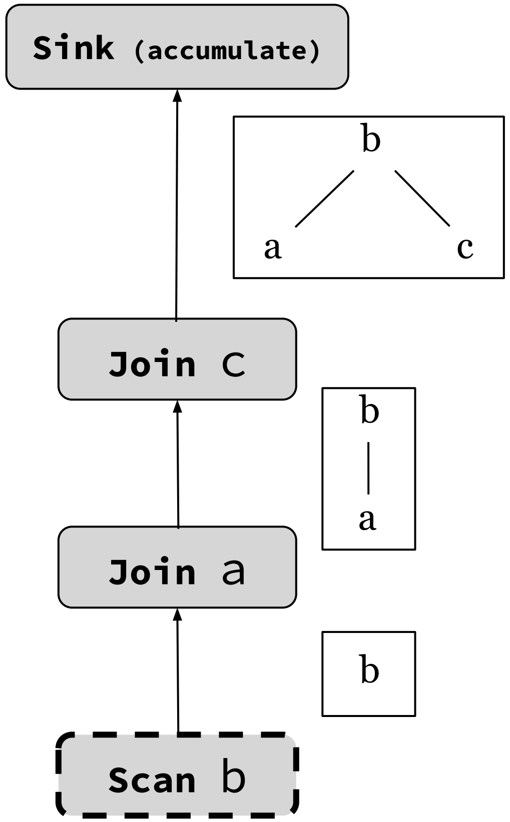




FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]





Sink (accumulate)

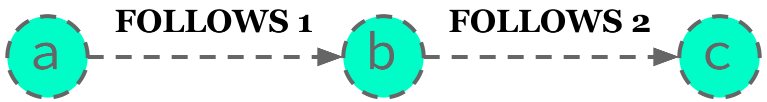
Join c

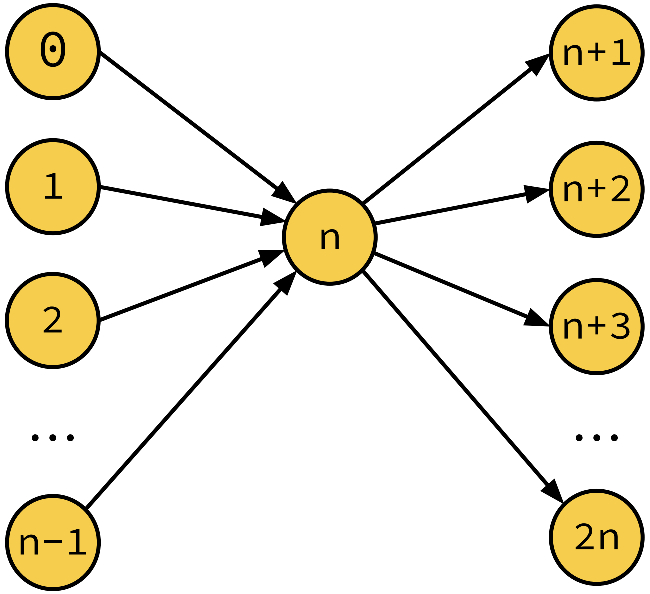
Join a

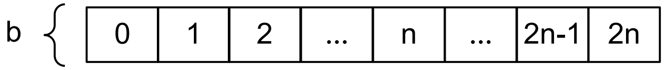
Scan b

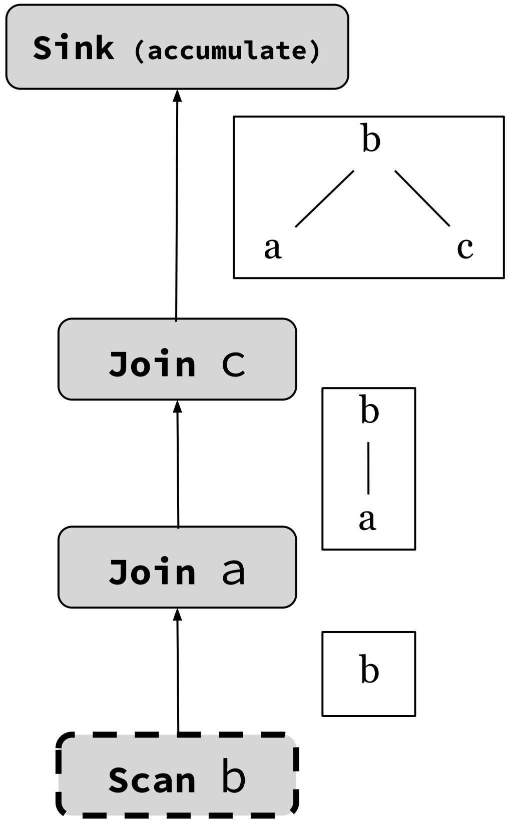
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



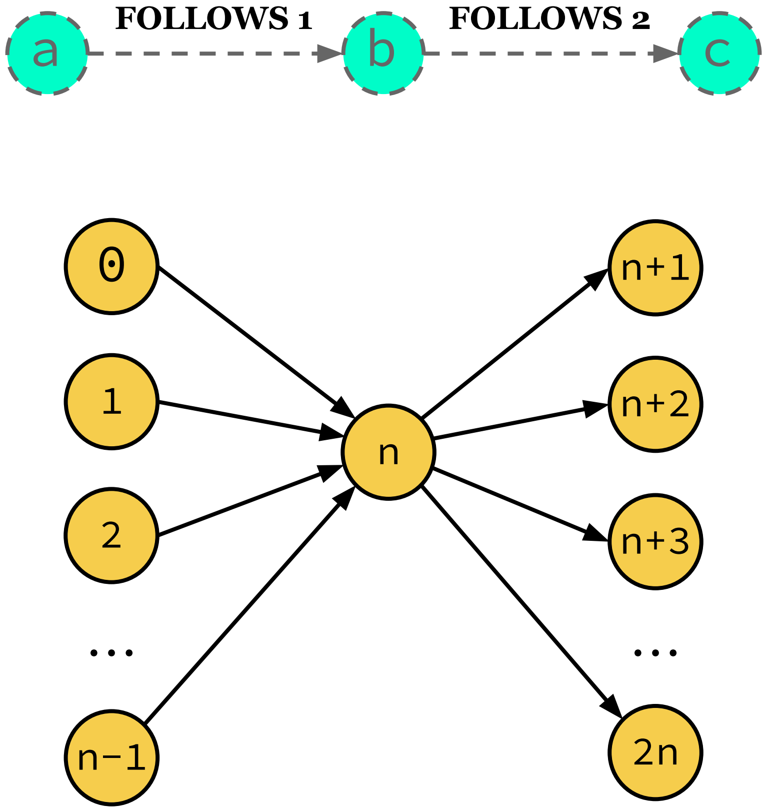




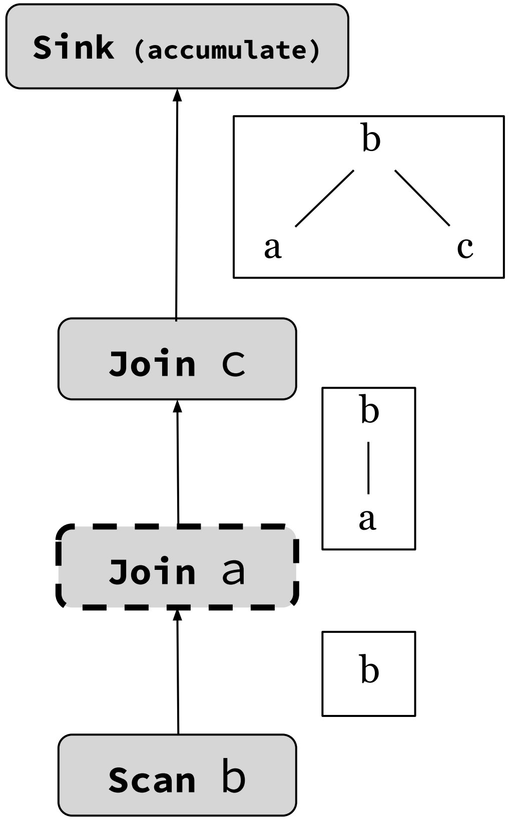


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

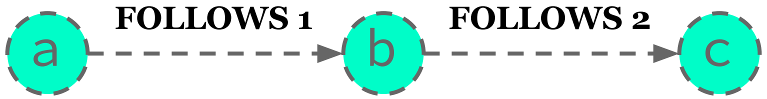


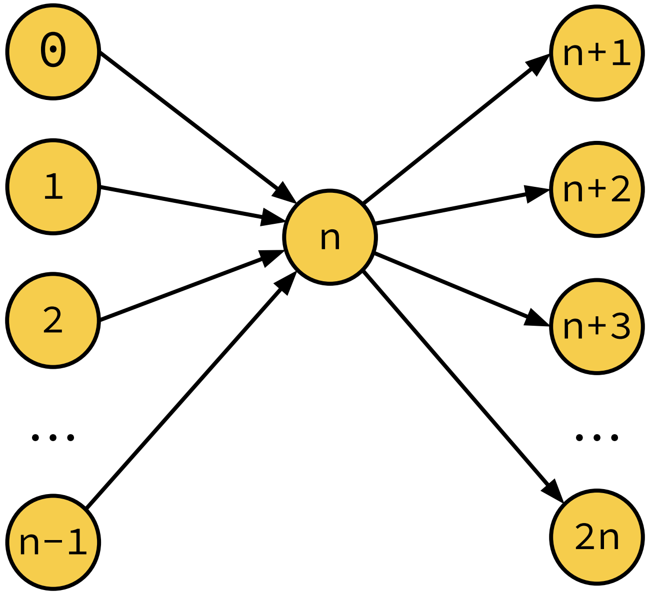
b	0	1	2	...	n	...	2n-1	2n

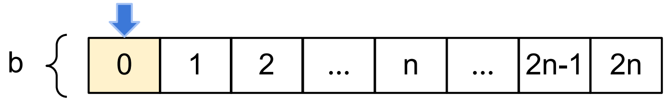


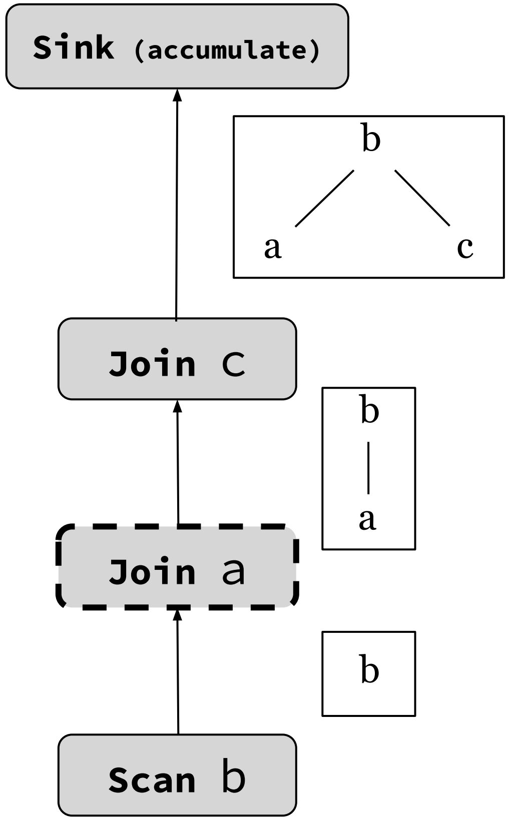
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



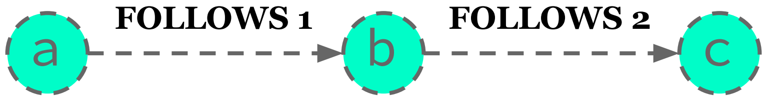


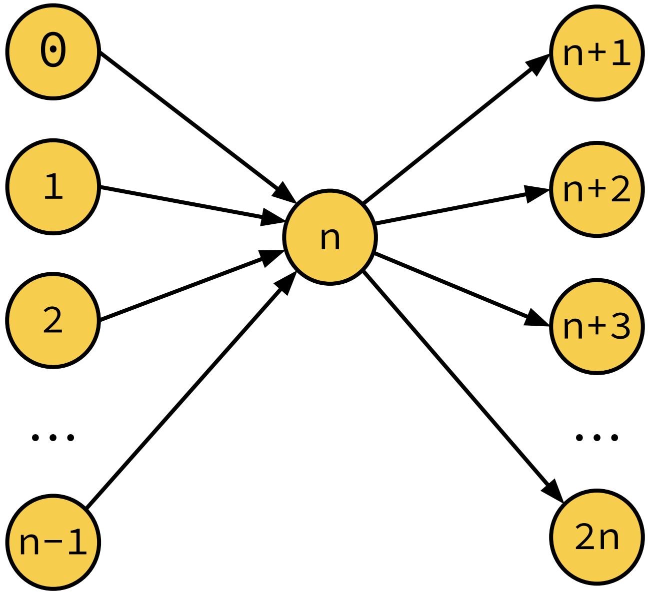


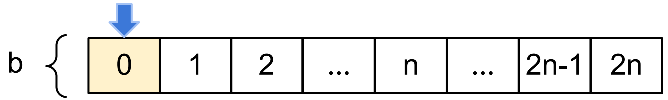


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

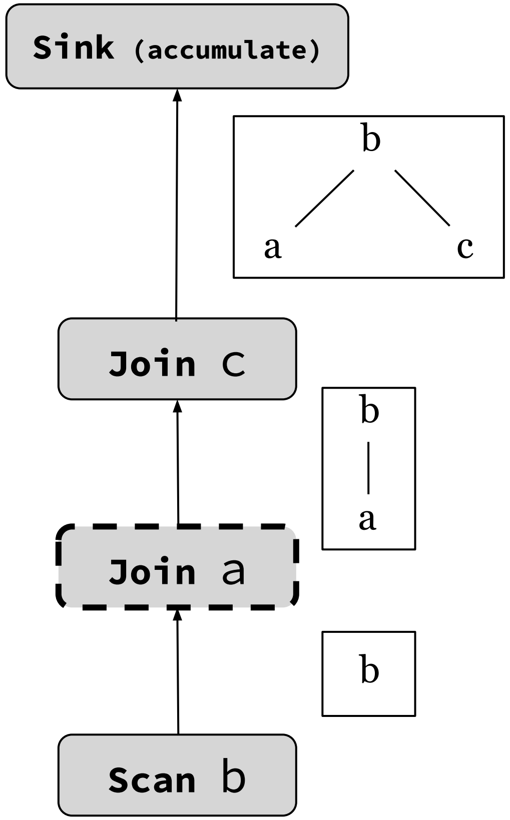






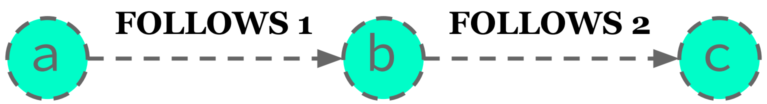
b	0	1	2	...	n	...	2n-1	2n

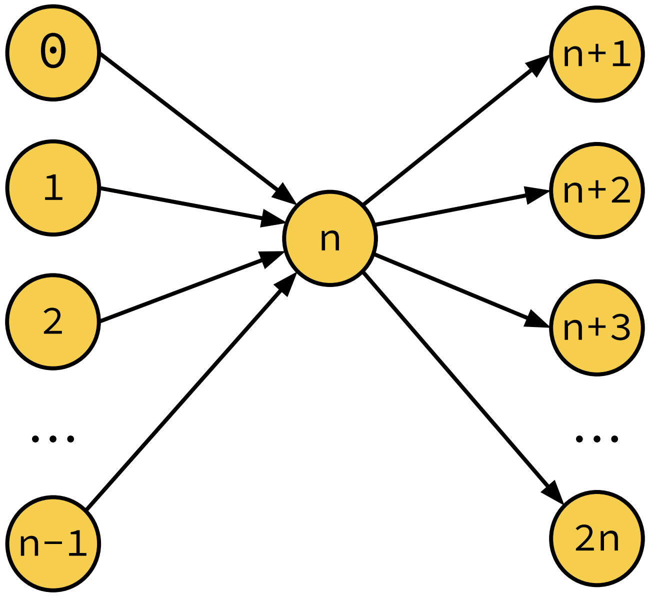
No users follow 0 in the graph

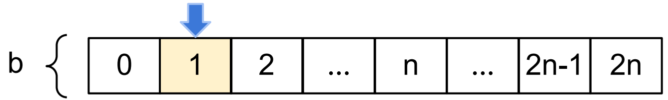


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

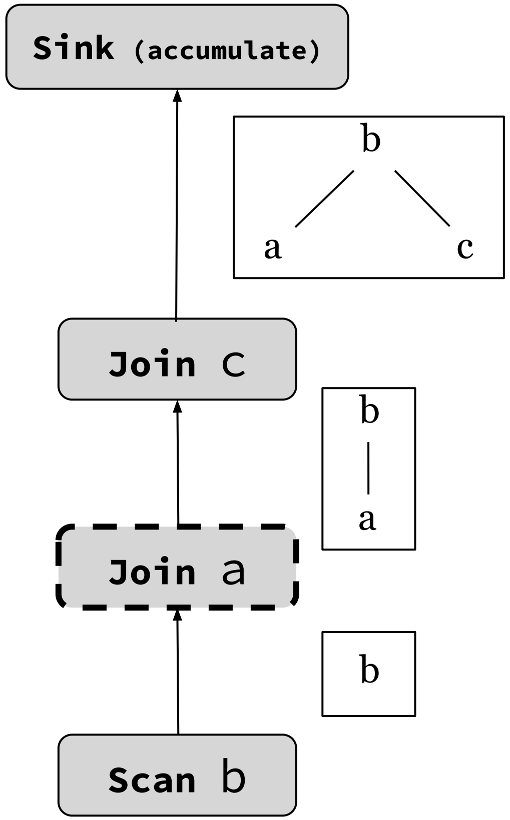






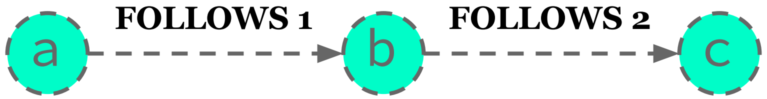
b	0	1	2	...	n	...	2n-1	2n

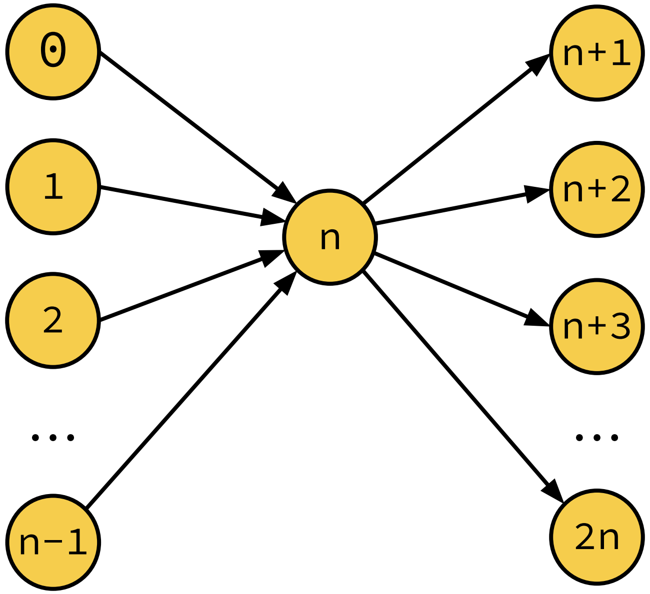
No users follow 1 in the graph

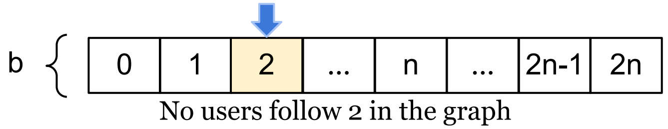


FDB: Factorized Database Engine

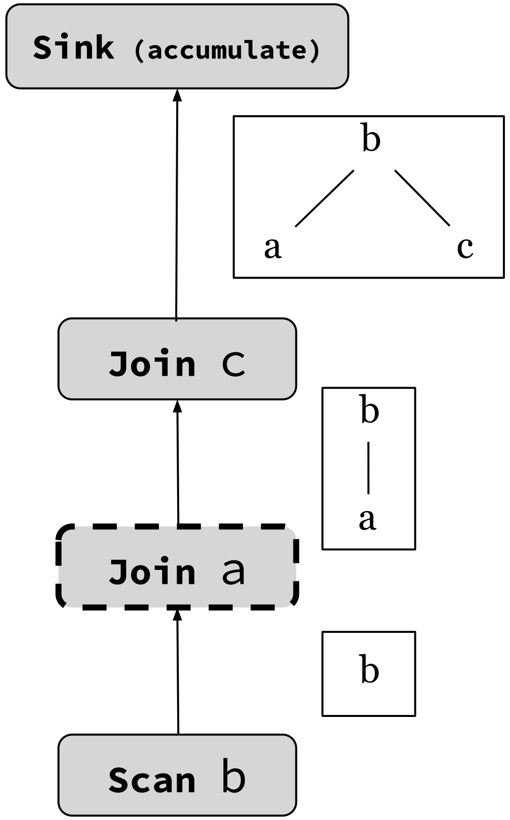
Query Vertex Ordering:
[b, a, c]





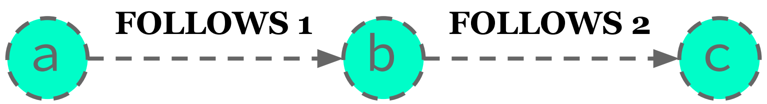


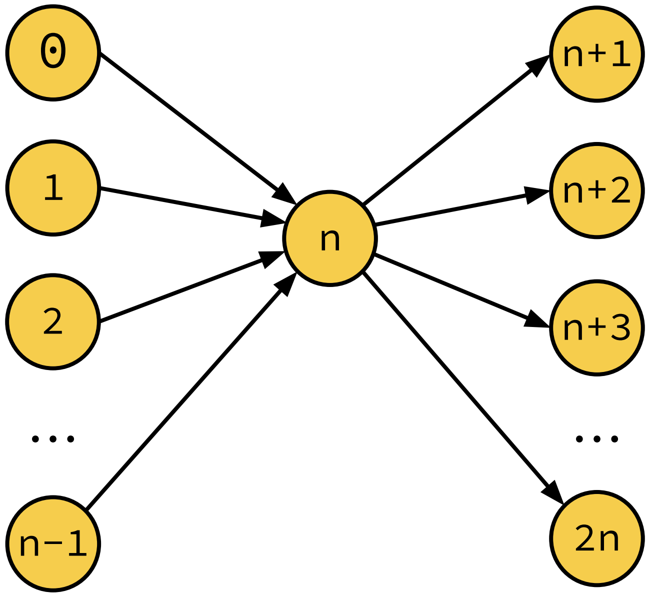
No users follow 2 in the graph

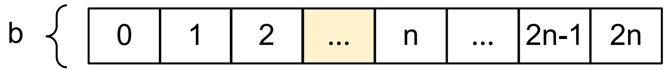


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

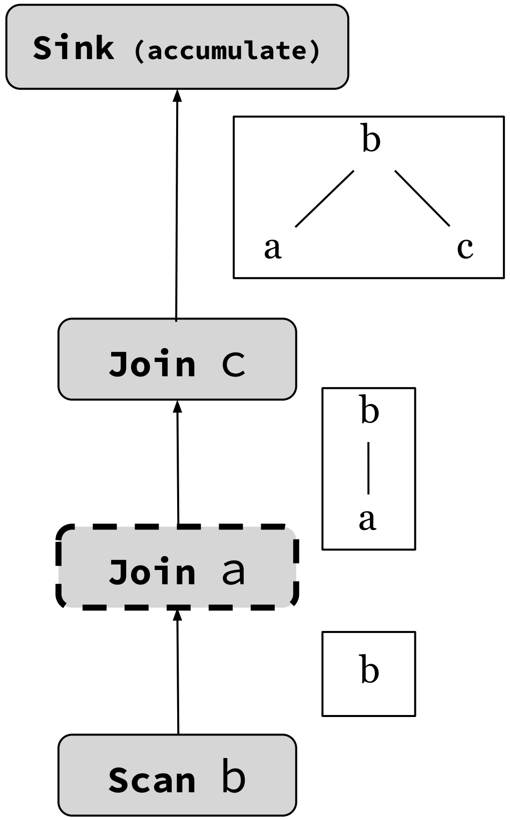






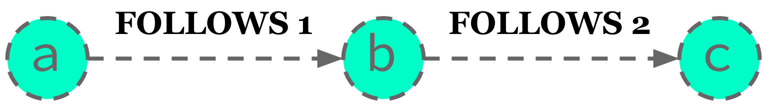
b	0	1	2	...	n	...	2n-1	2n

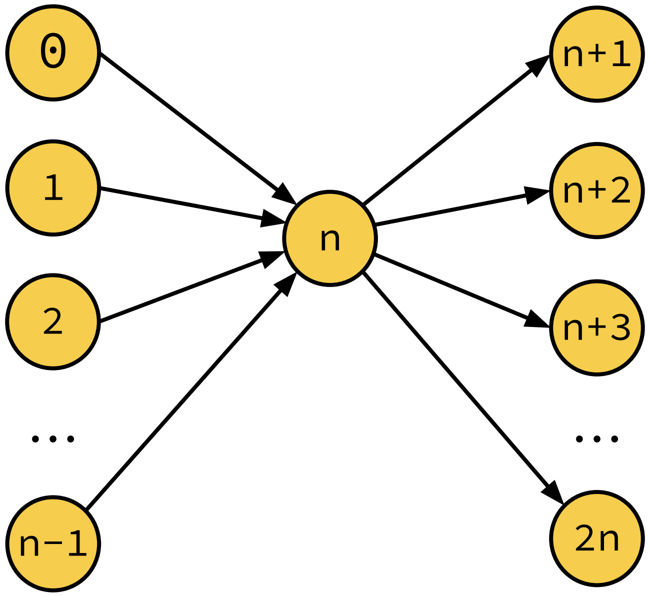
No users follow 2 in the graph

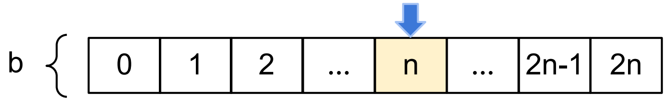


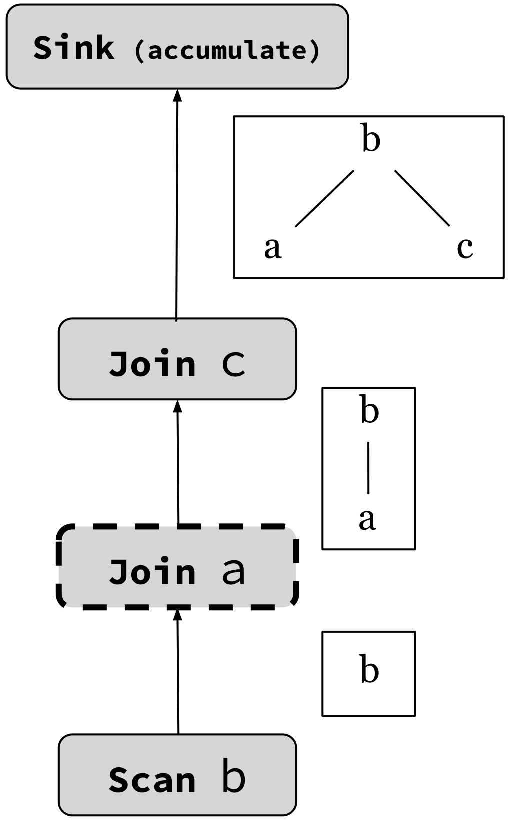
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



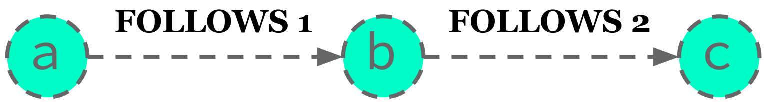


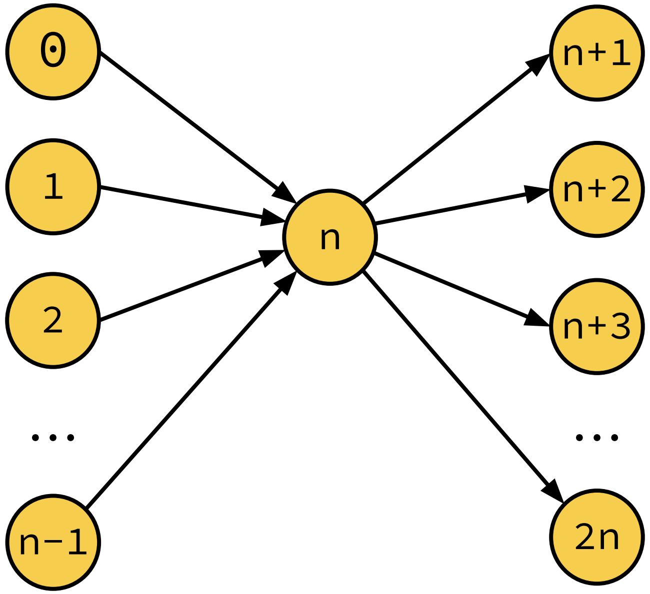


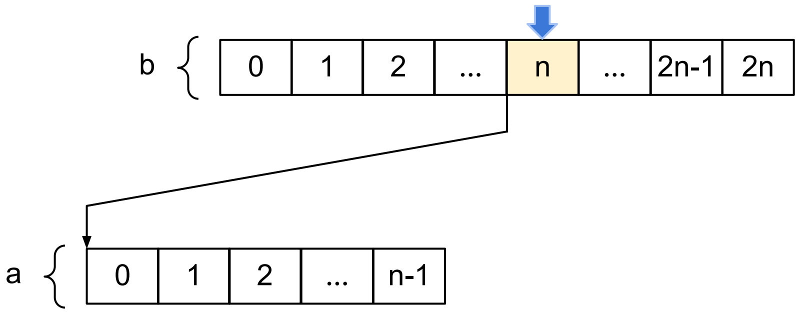


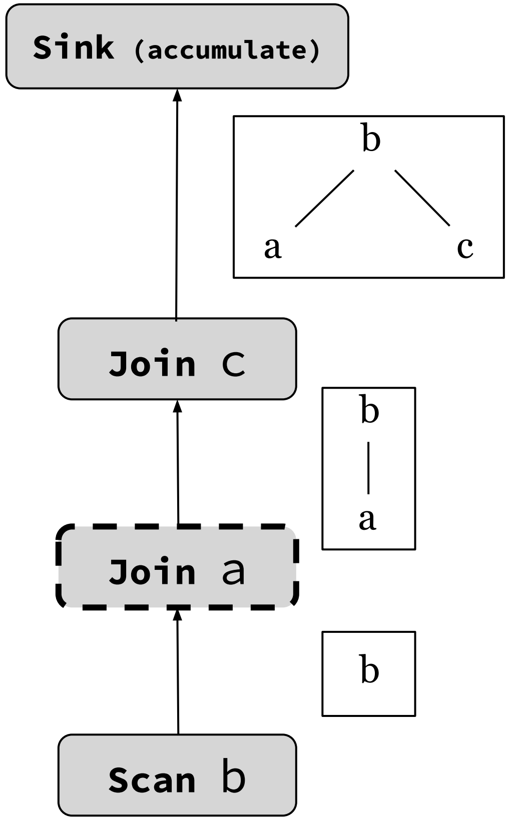
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



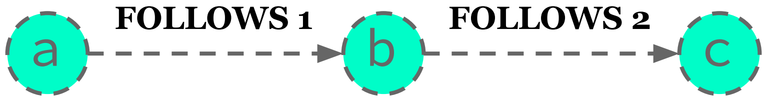


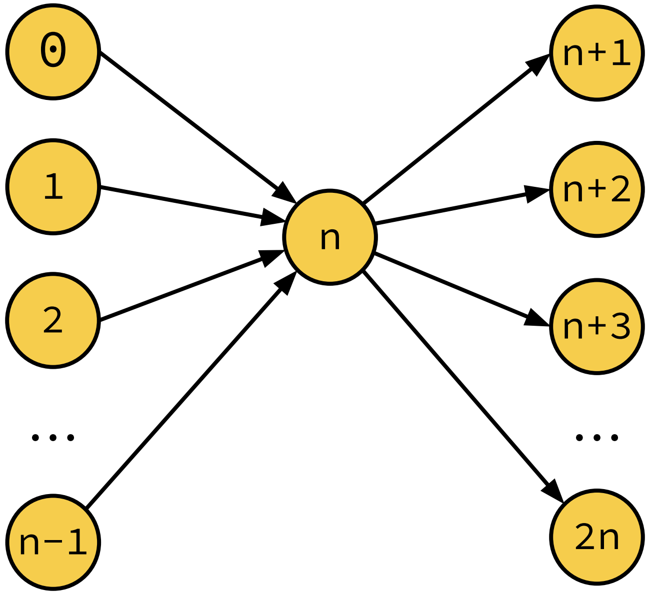


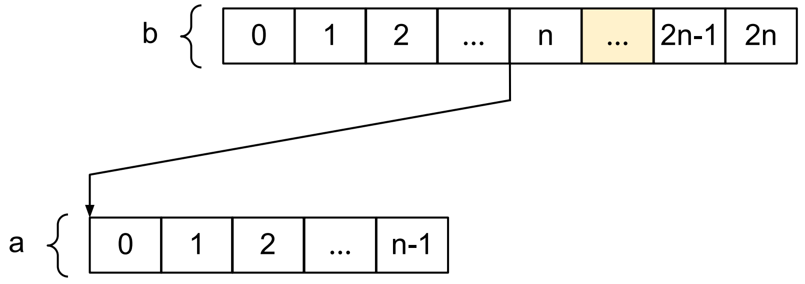


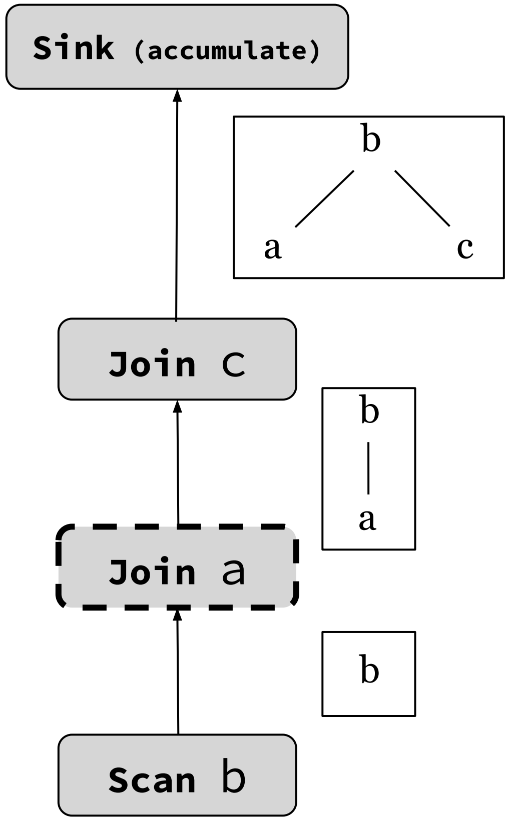
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



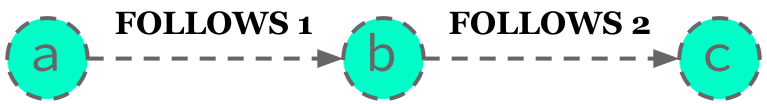


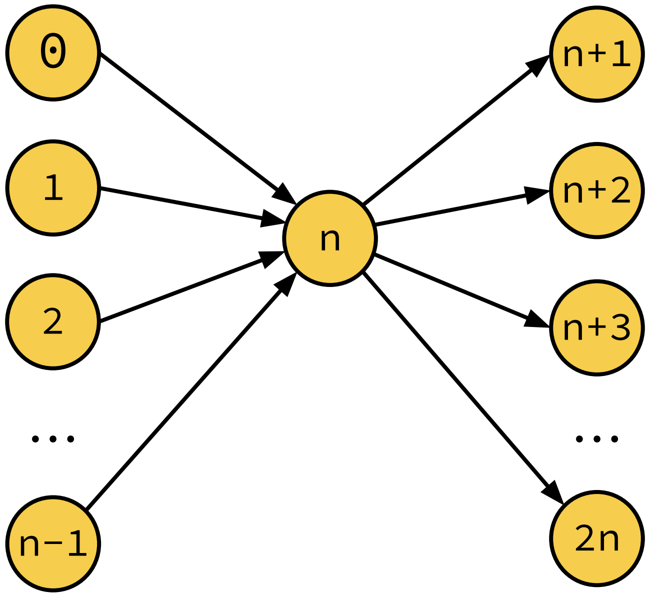


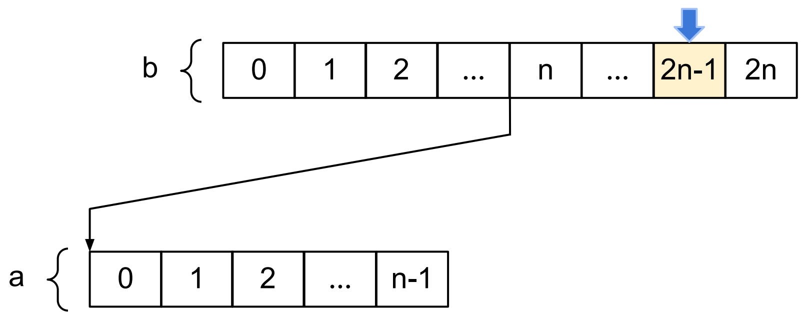


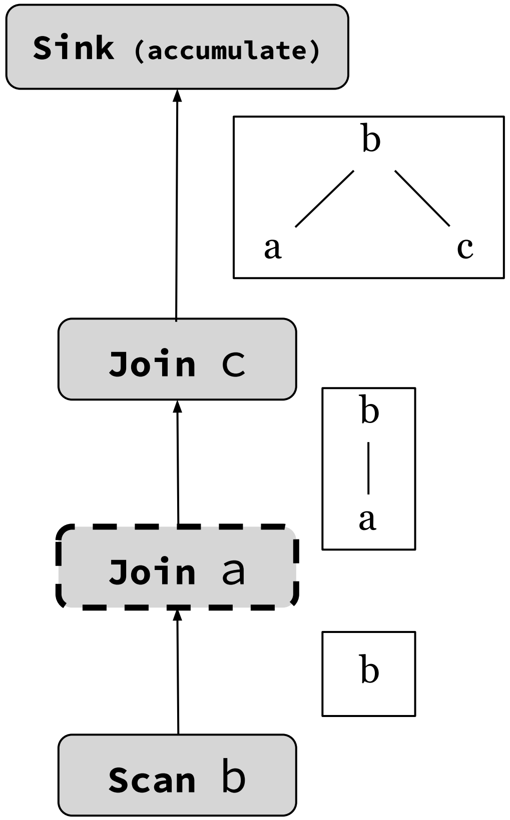
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



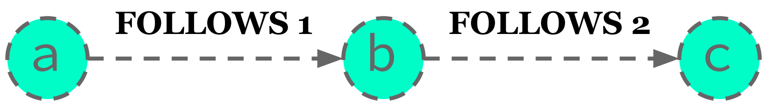


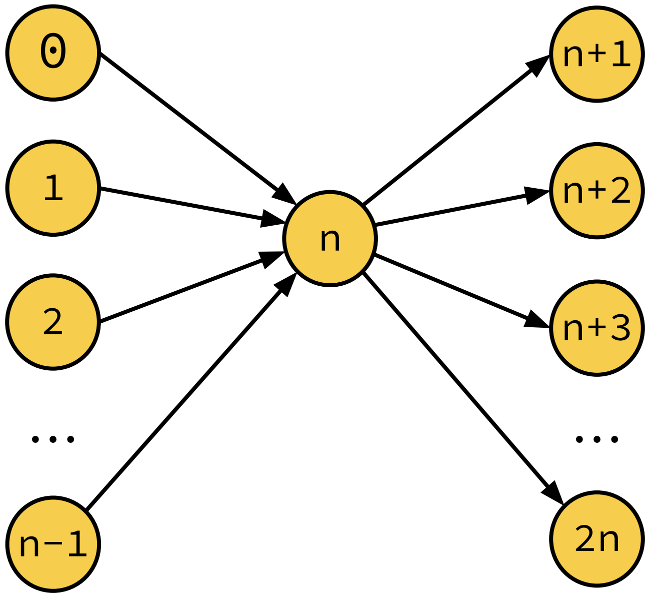


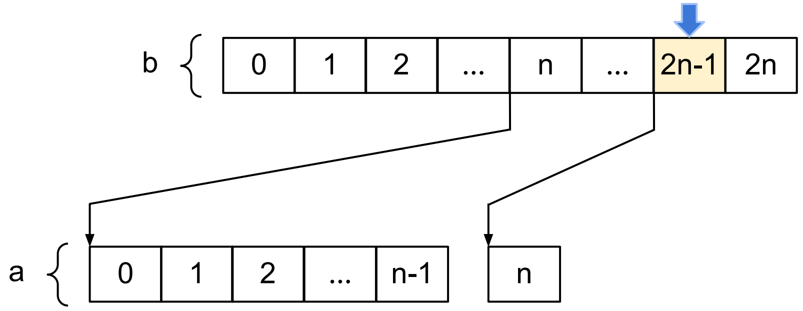


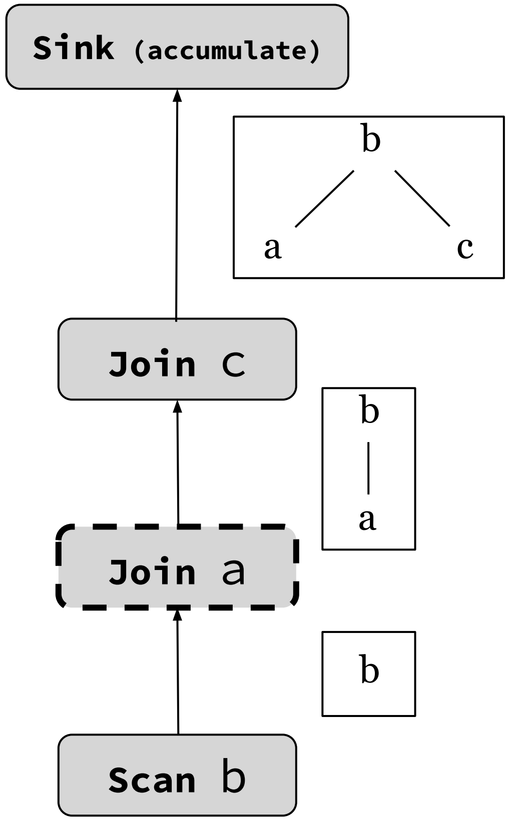
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



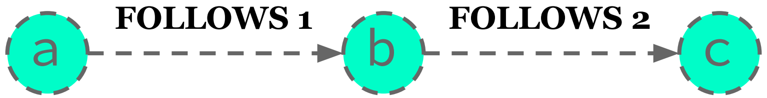


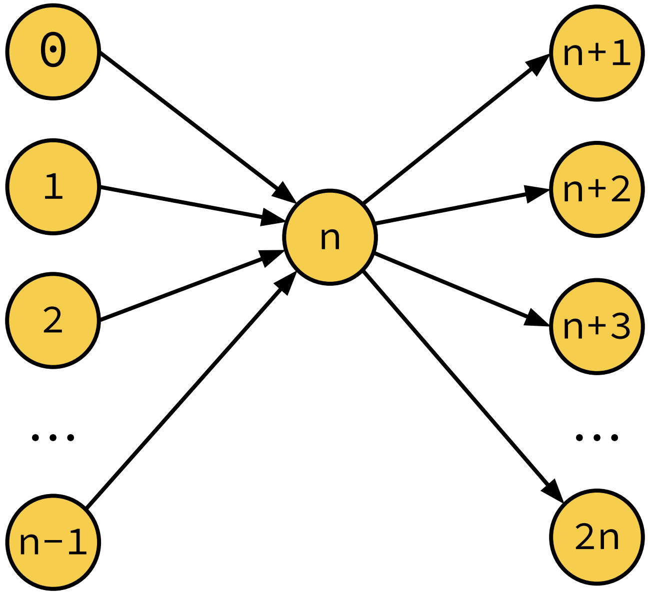


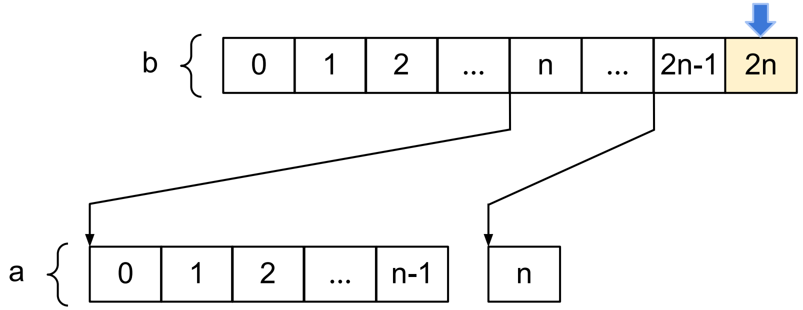


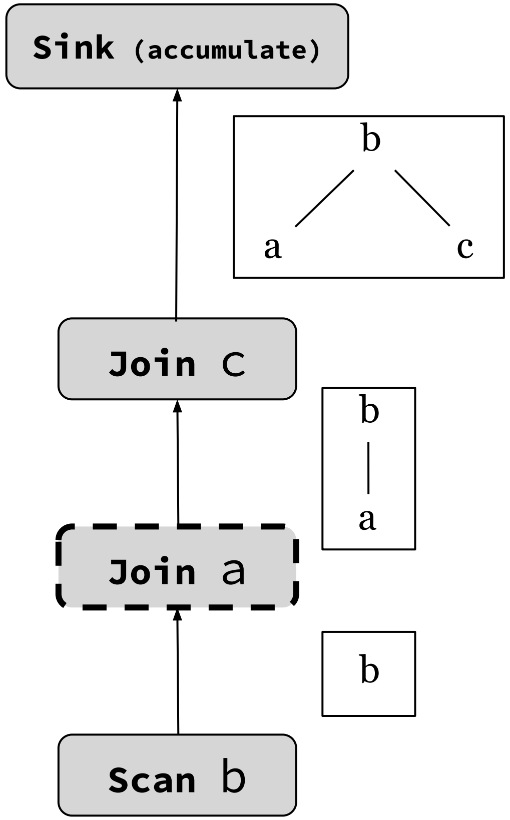
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]








Sink (accumulate)

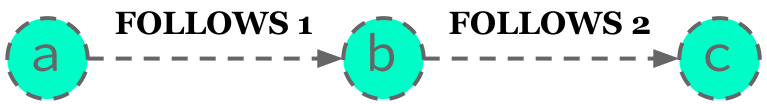
Join c

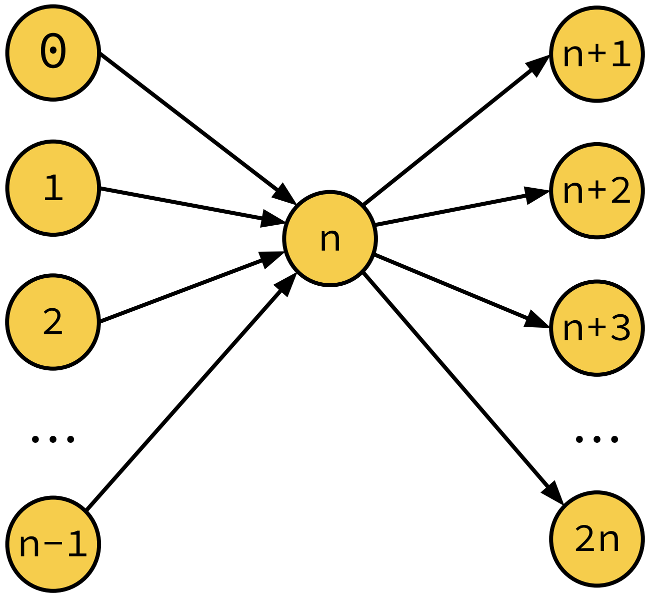
Join a

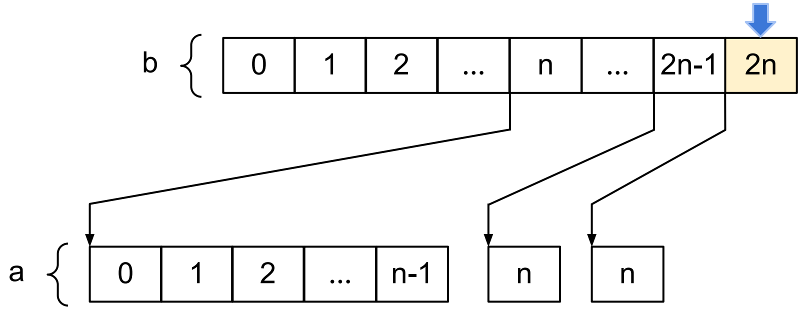
Scan b

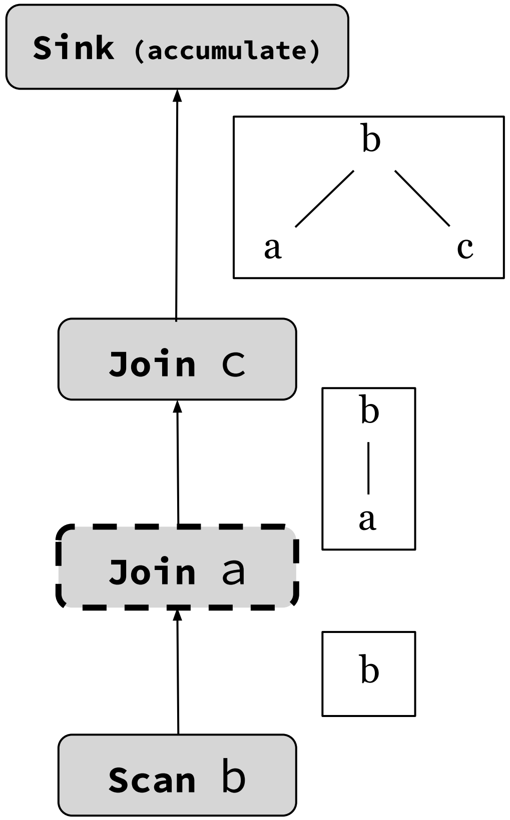
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



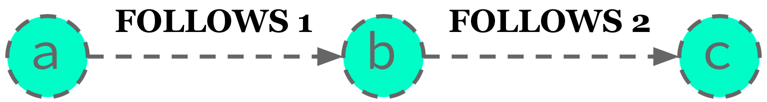


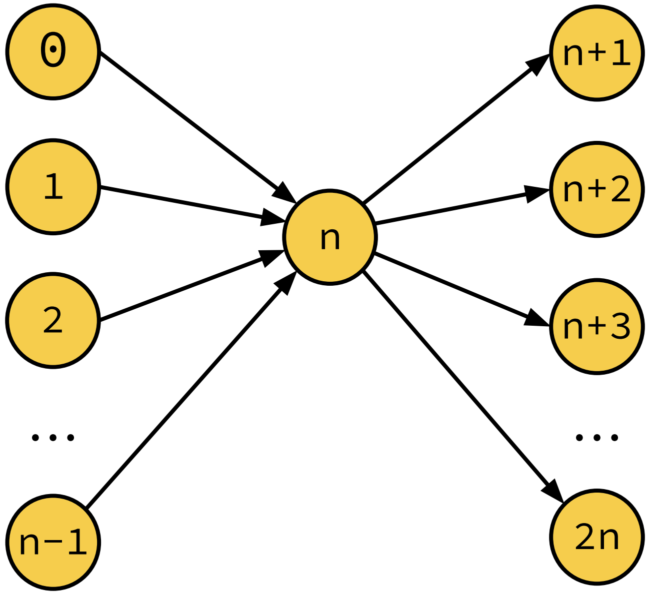


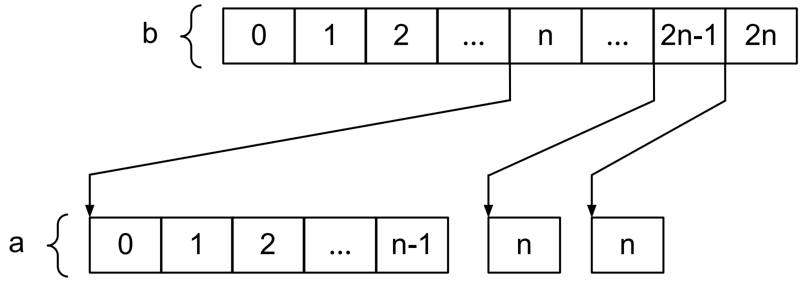


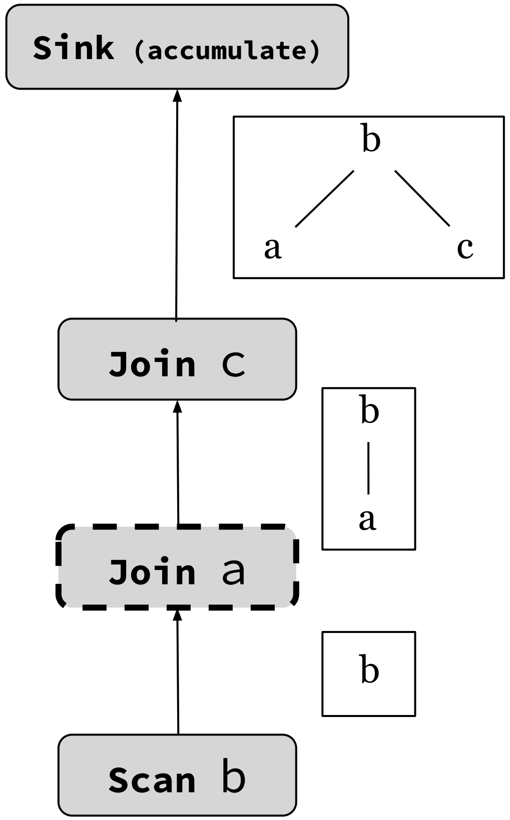
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



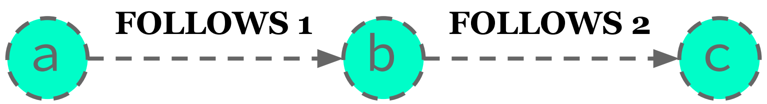


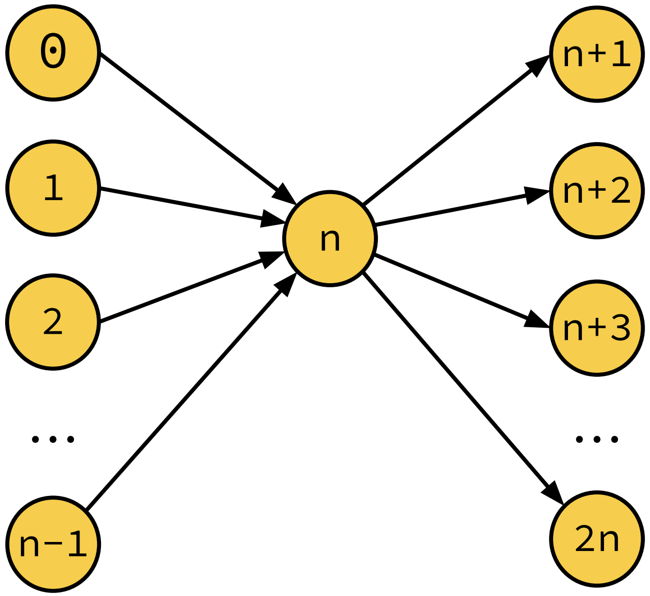


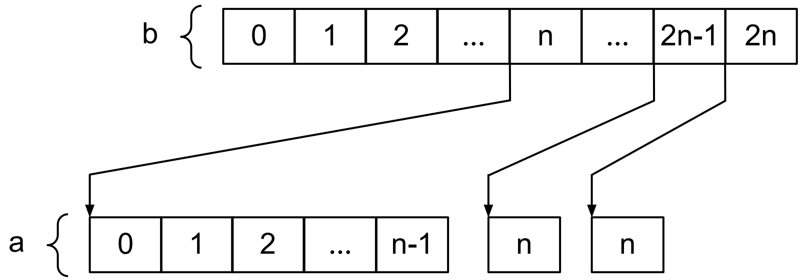


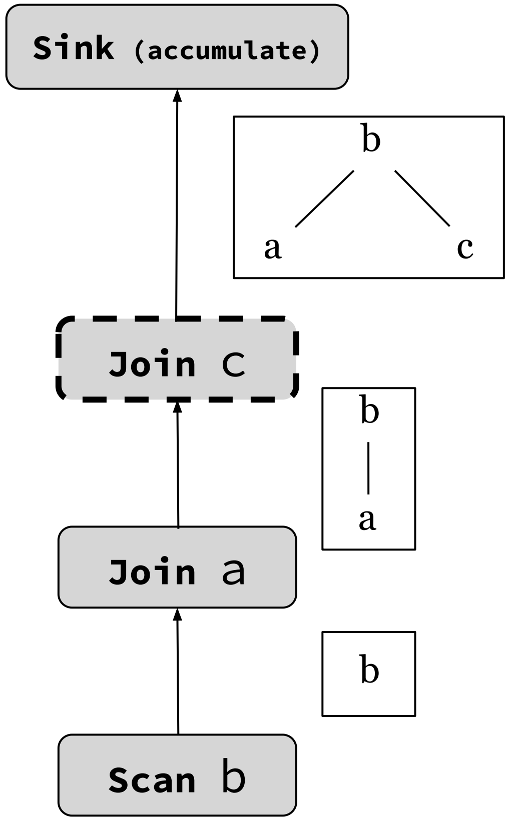
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



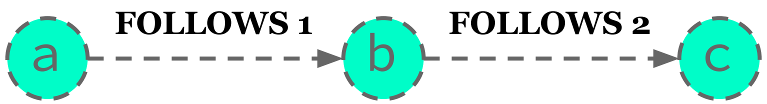


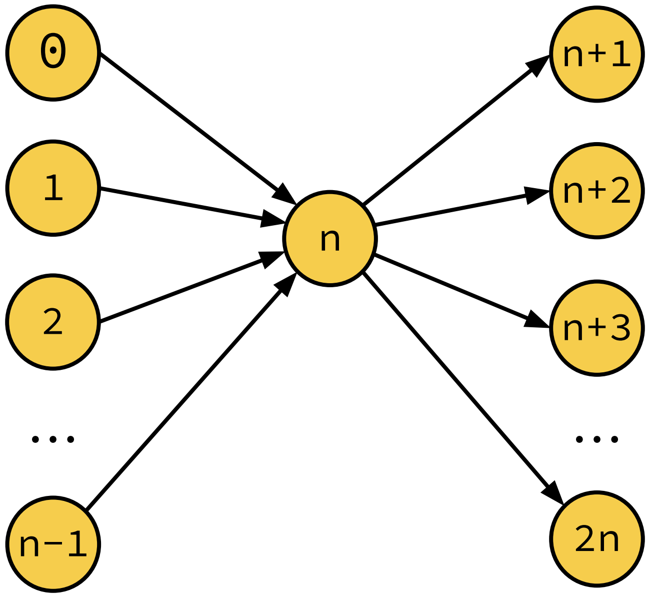


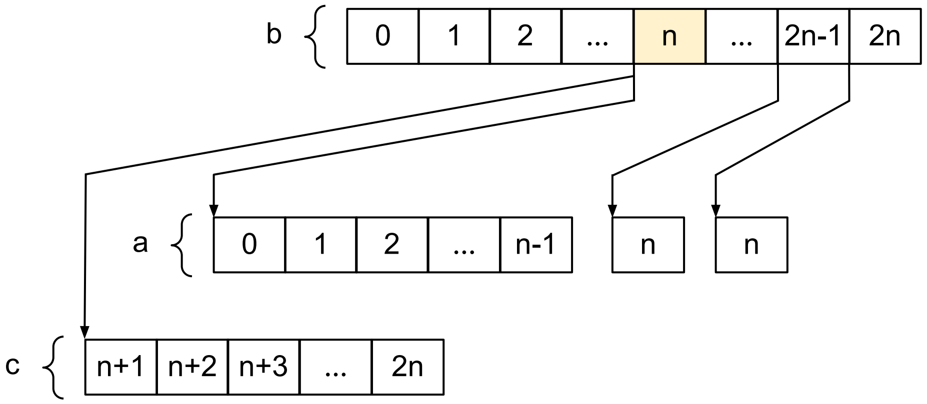


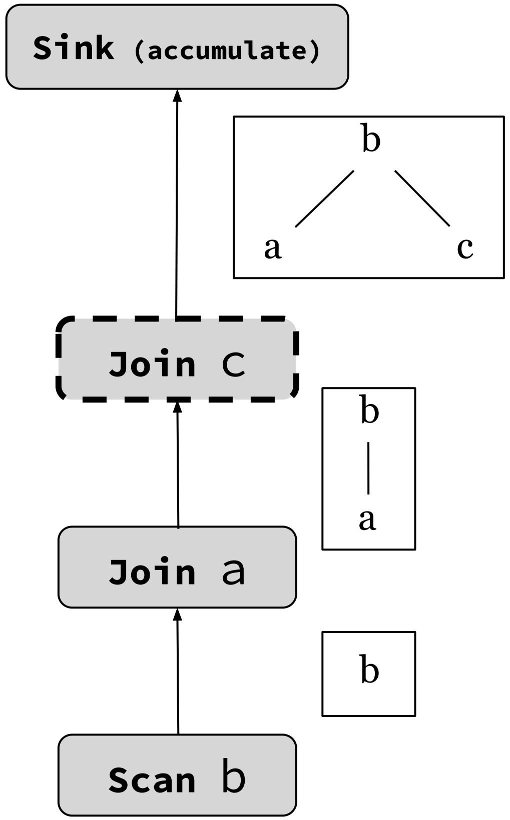
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



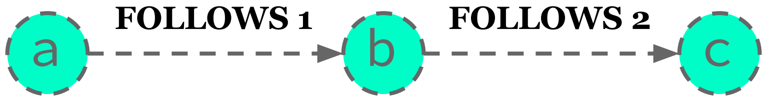


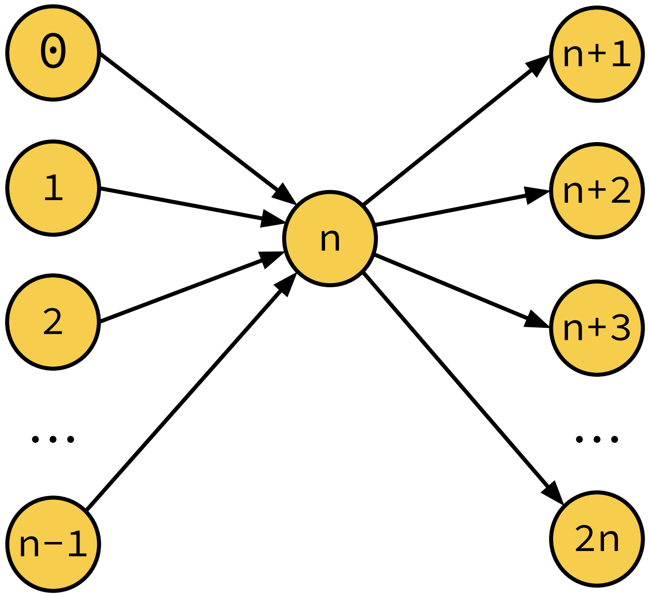


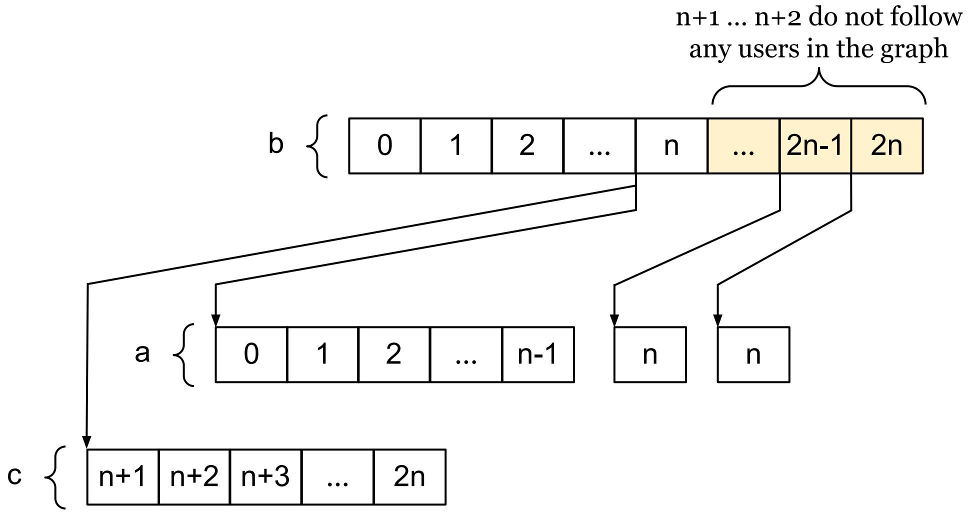


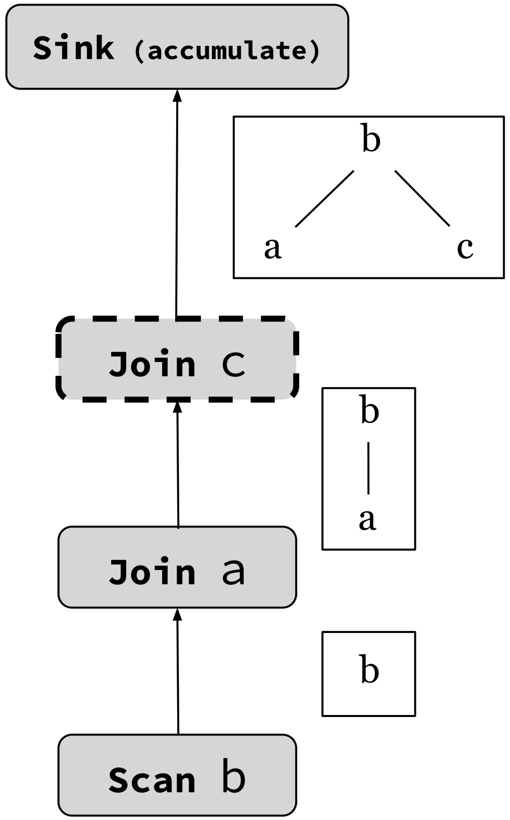
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



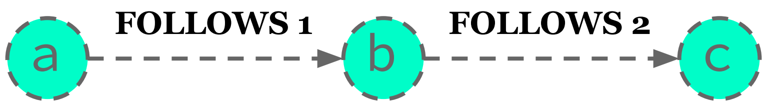


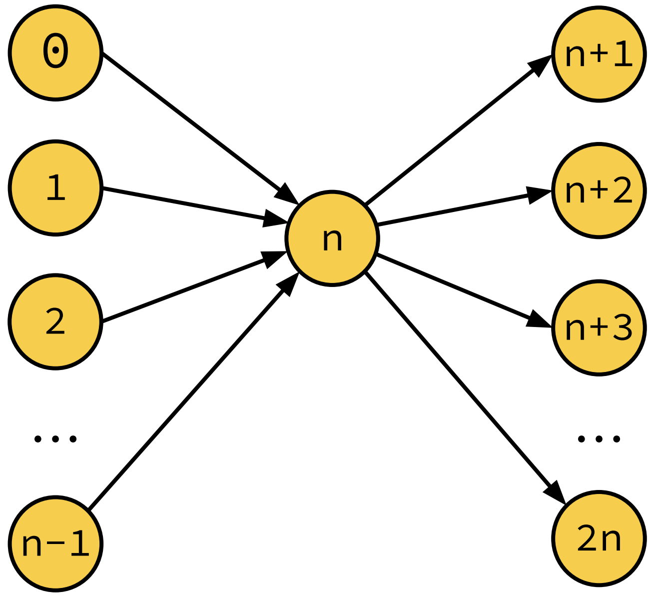

n+1 ... n+2 do not follow any users in the graph

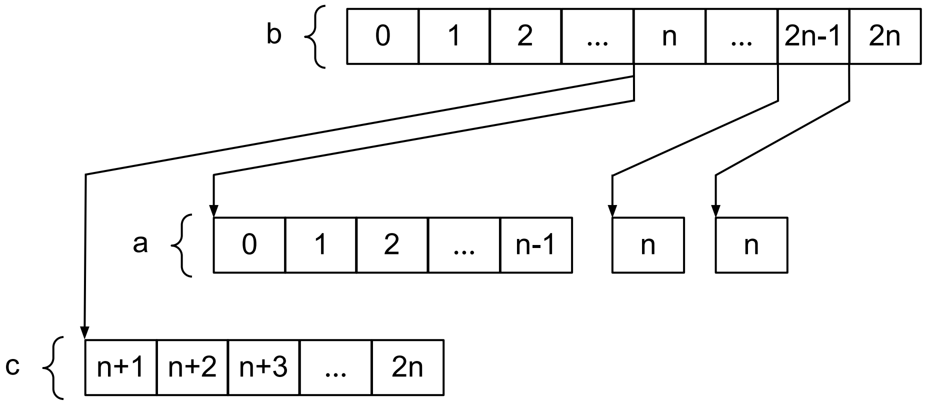


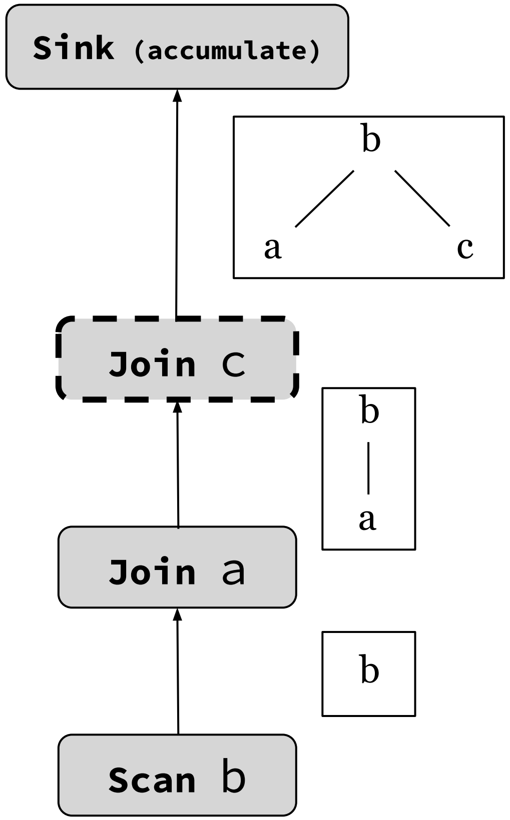
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



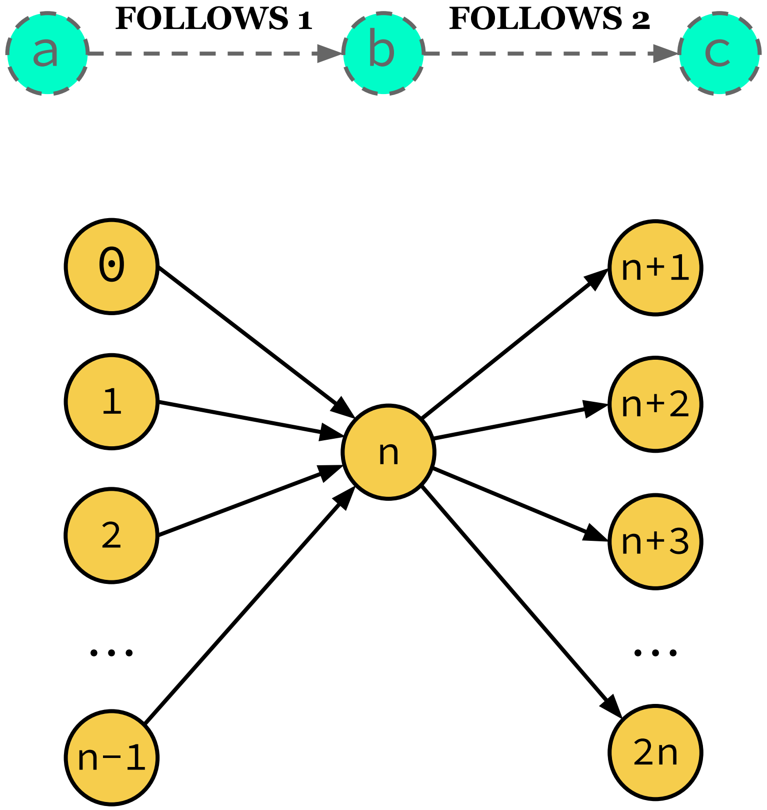




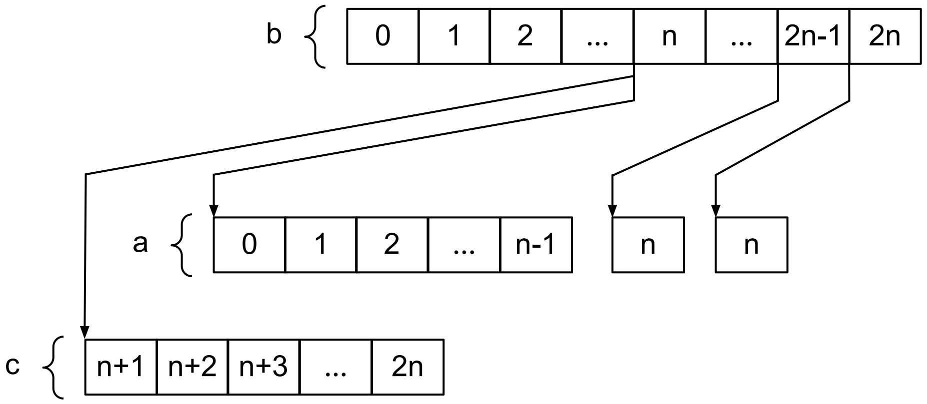


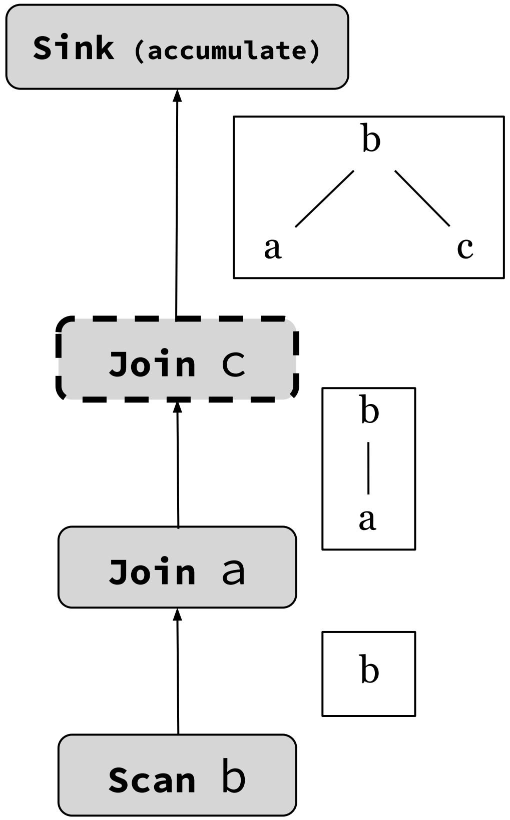
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



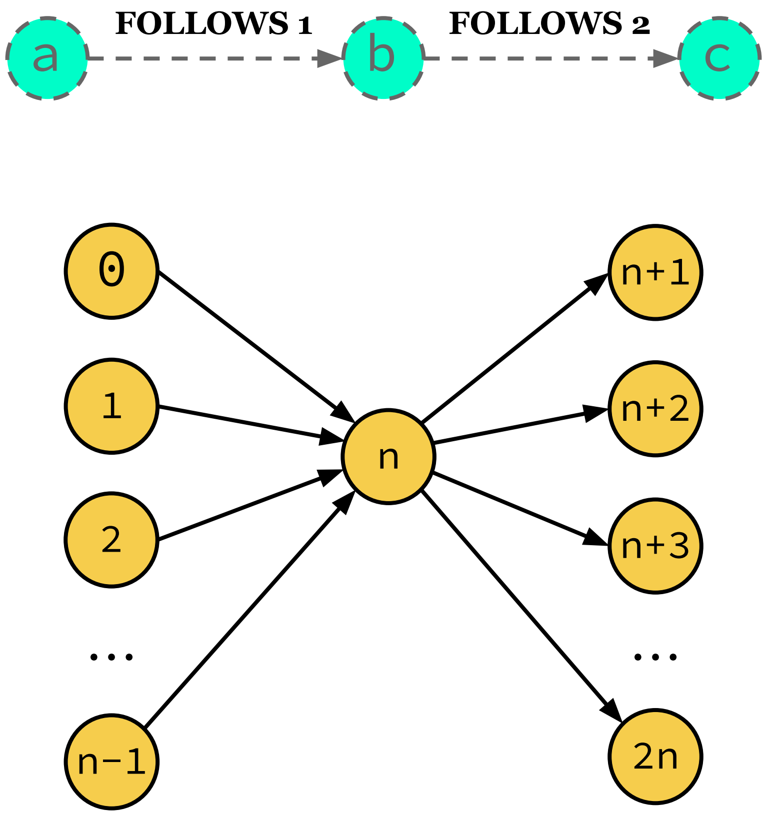
- Implements core algorithms for SJP queries meeting all asymptotic bounds.
- Able to generate all factorized trees.

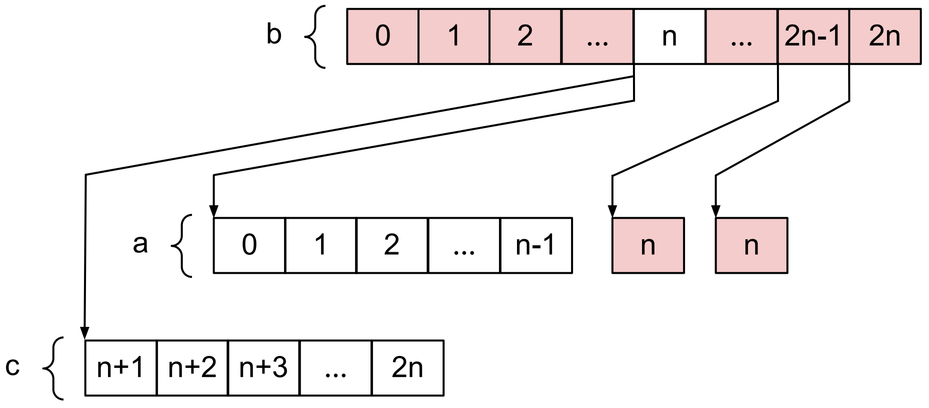


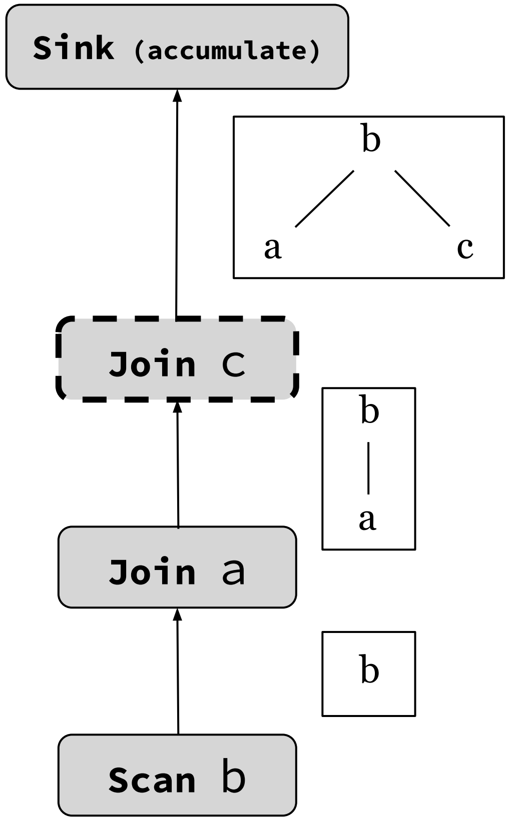


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]

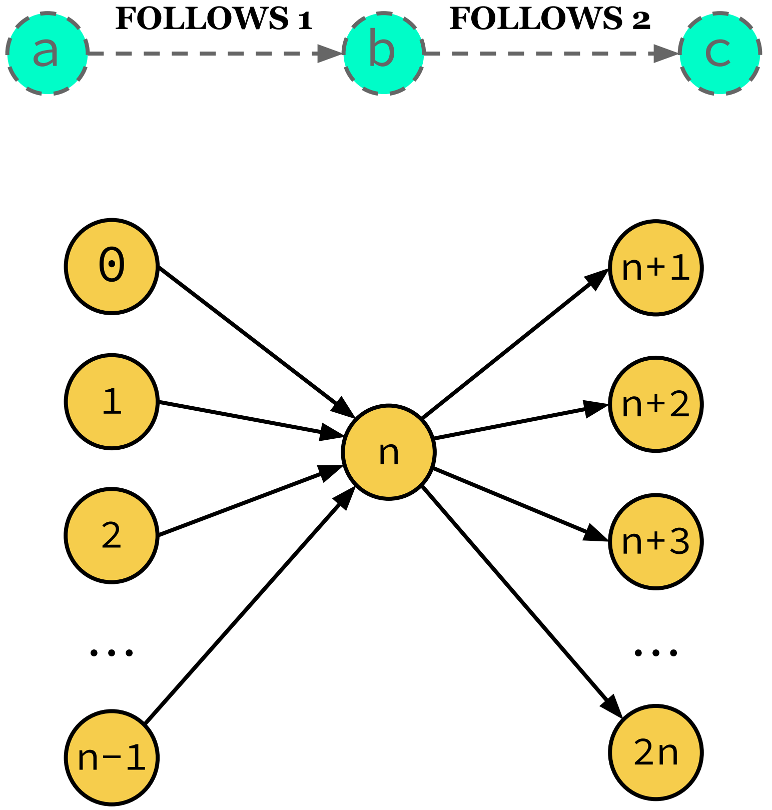




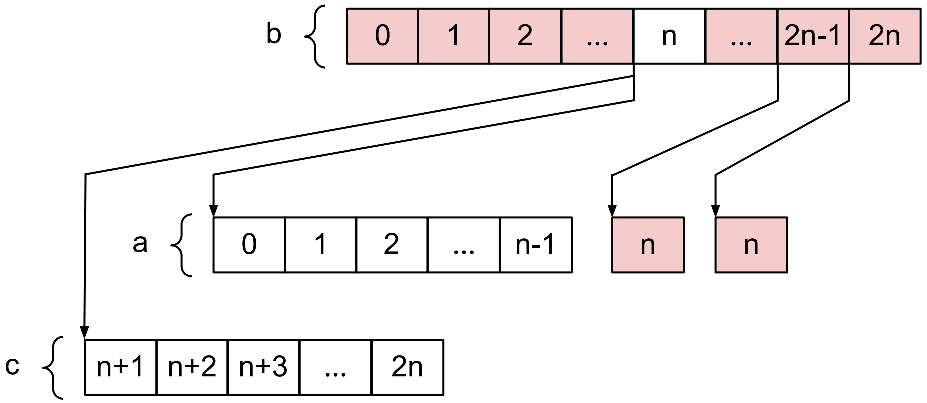


FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



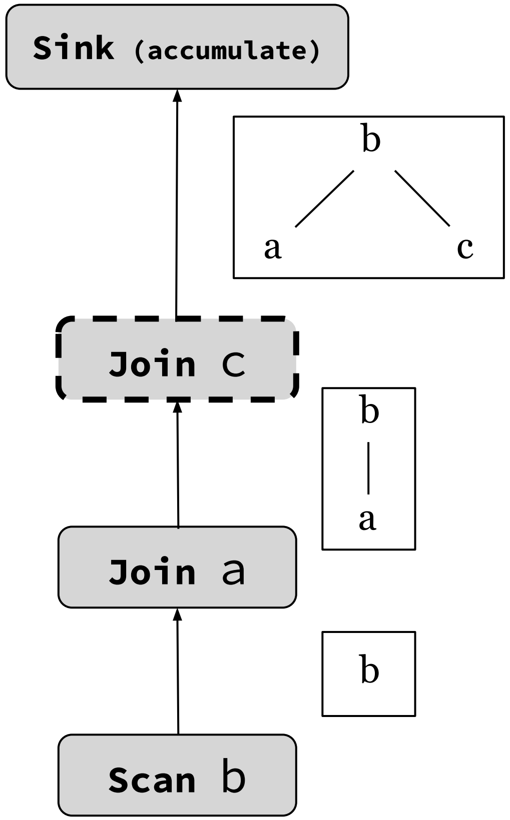
- Materializes more than necessary till the sink! Recursive deletes need to be handled.



b: 0 | 1 | 2 | ... | n | ... | 2n-1 | 2n

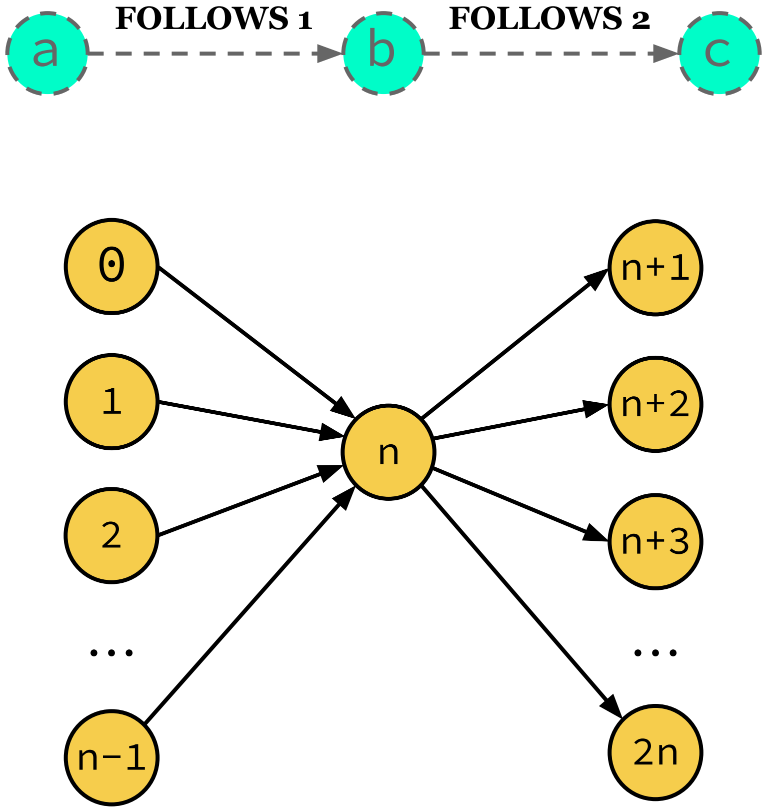
a: 0 | 1 | 2 | ... | n-1 — n — n

c: n+1 | n+2 | n+3 | ... | 2n

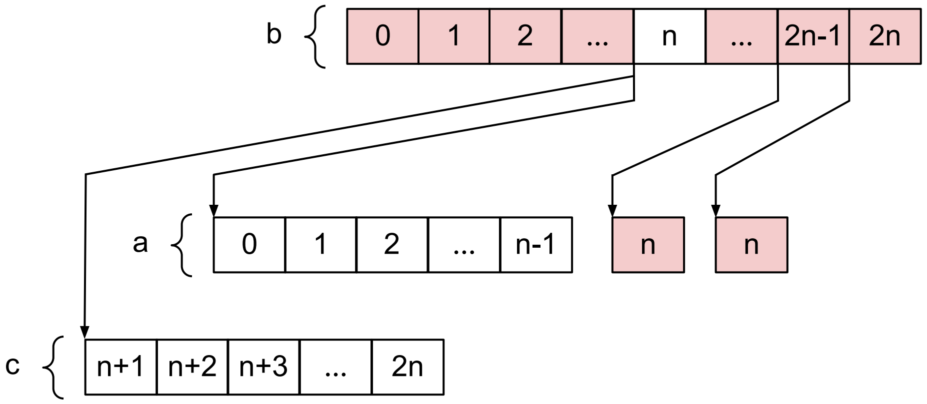


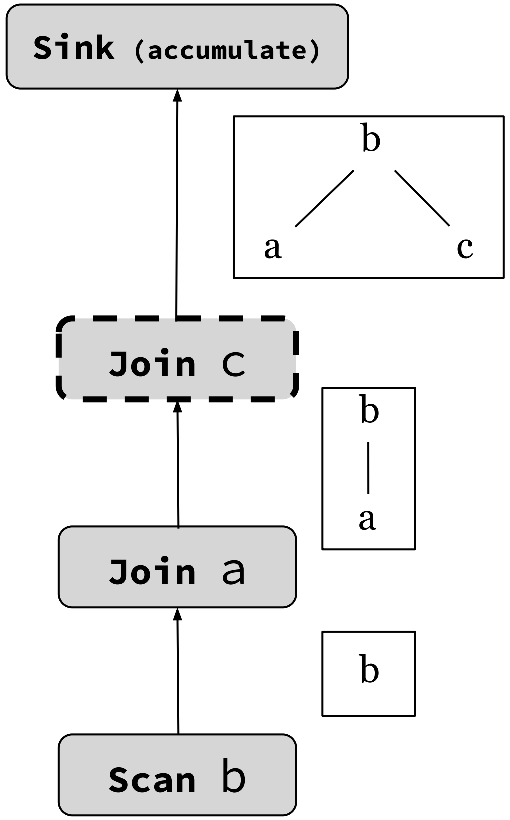
FDB: Factorized Database Engine

Query Vertex Ordering:
[b, a, c]



- Materializes more than necessary till the sink! Recursive deletes need to be handled.
- Full materialization operators: loss of classic optimizations e.g., vectorized execution.





Outline of Query Processing Techniques to Cover

For each we cover: a) **Foundations**; b) **System implementations**; and c) **Open challenges**.

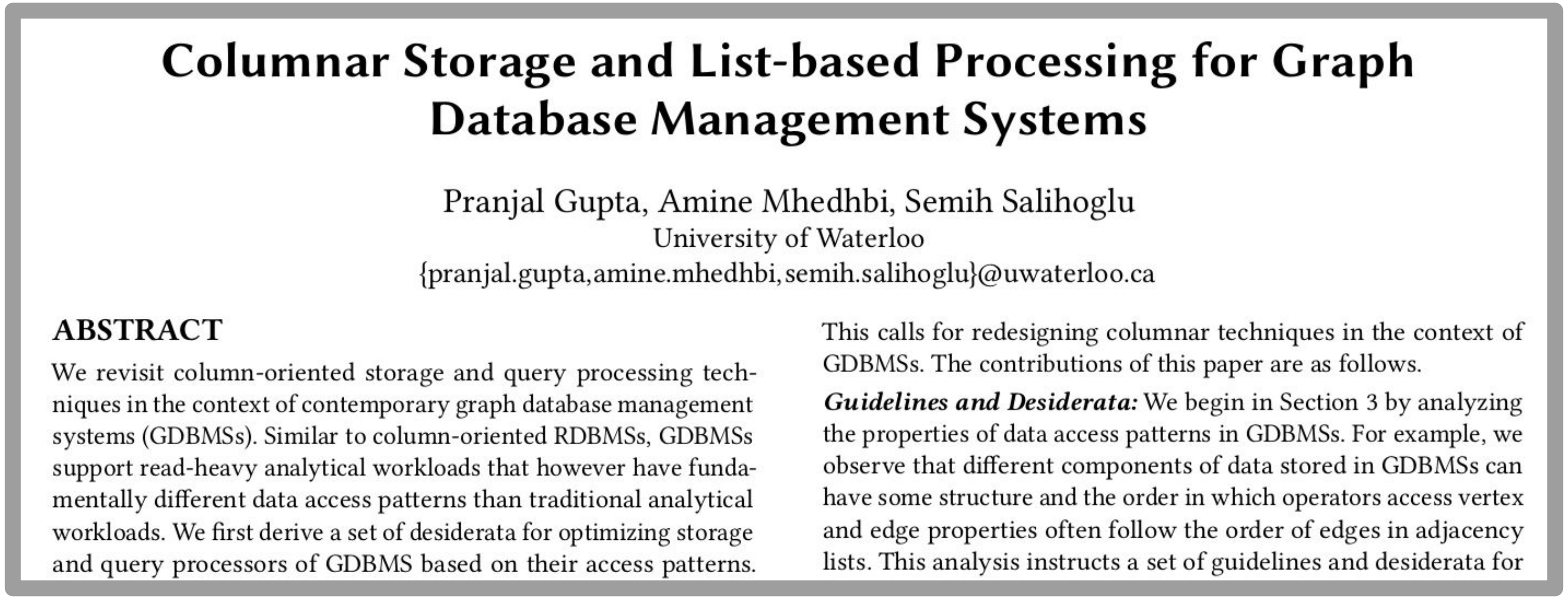
1) **Predefined Joins**

2) **Worst-case optimal joins**

3) **Factorized Query Processing**
   Handling Intermediate Size Growth for Acyclic Joins
   3.1. Foundations: Factorized Representations
   3.2. System Integration Approaches: FDB and **Factorized Vector Execution in Graphflow**

Factorized Vector Execution

Factorized Vector Execution



Columnar Storage and List-based Processing for Graph Database Management Systems

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu
University of Waterloo
{pranjal.gupta,amine.mhedhbi,semih.salihoglu}@uwaterloo.ca

ABSTRACT

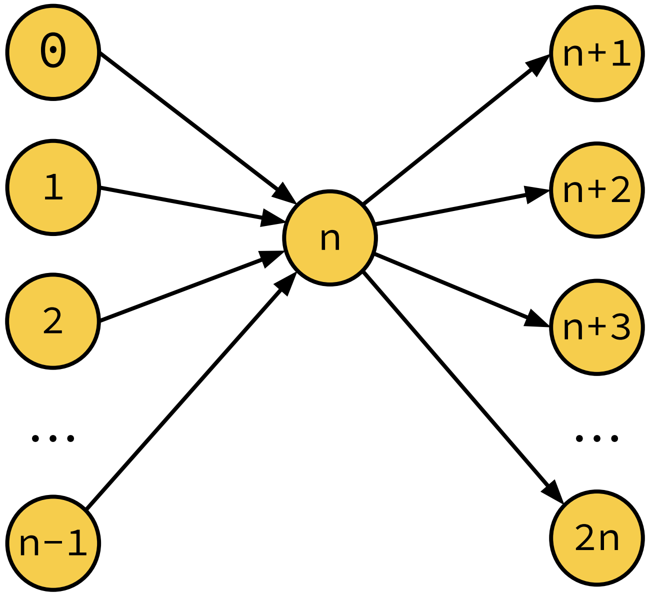
We revisit column-oriented storage and query processing techniques in the context of contemporary graph database management systems (GDBMSs). Similar to column-oriented RDBMSs, GDBMSs support read-heavy analytical workloads that however have fundamentally different data access patterns than traditional analytical workloads. We first derive a set of desiderata for optimizing storage and query processors of GDBMS based on their access patterns.

This calls for redesigning columnar techniques in the context of GDBMSs. The contributions of this paper are as follows.

Guidelines and Desiderata: We begin in Section 3 by analyzing the properties of data access patterns in GDBMSs. For example, we observe that different components of data stored in GDBMSs can have some structure and the order in which operators access vertex and edge properties often follow the order of edges in adjacency lists. This analysis instructs a set of guidelines and desiderata for

Factorized Vector Execution

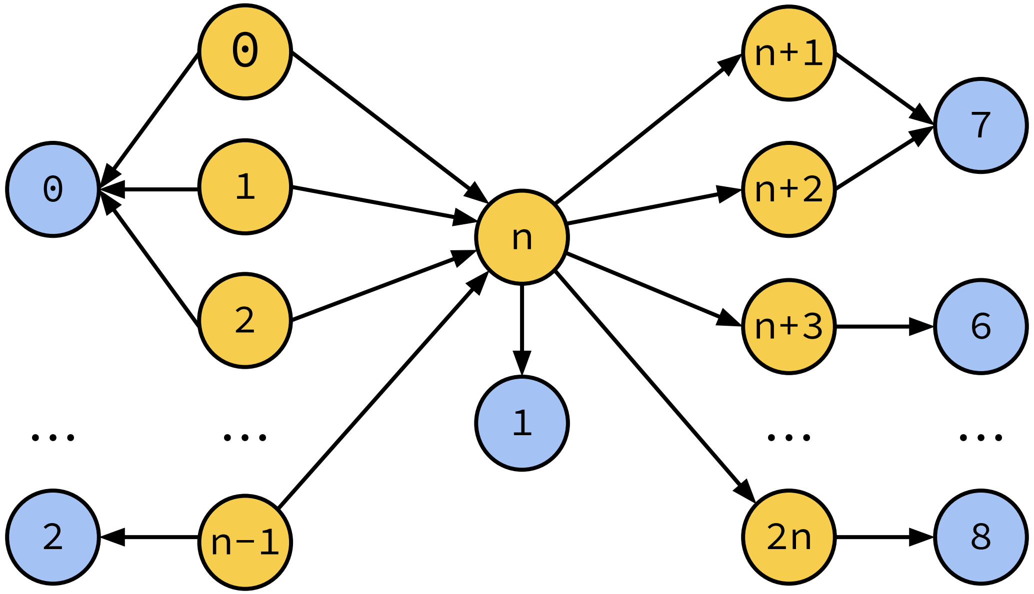
Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

Factorized Vector Execution

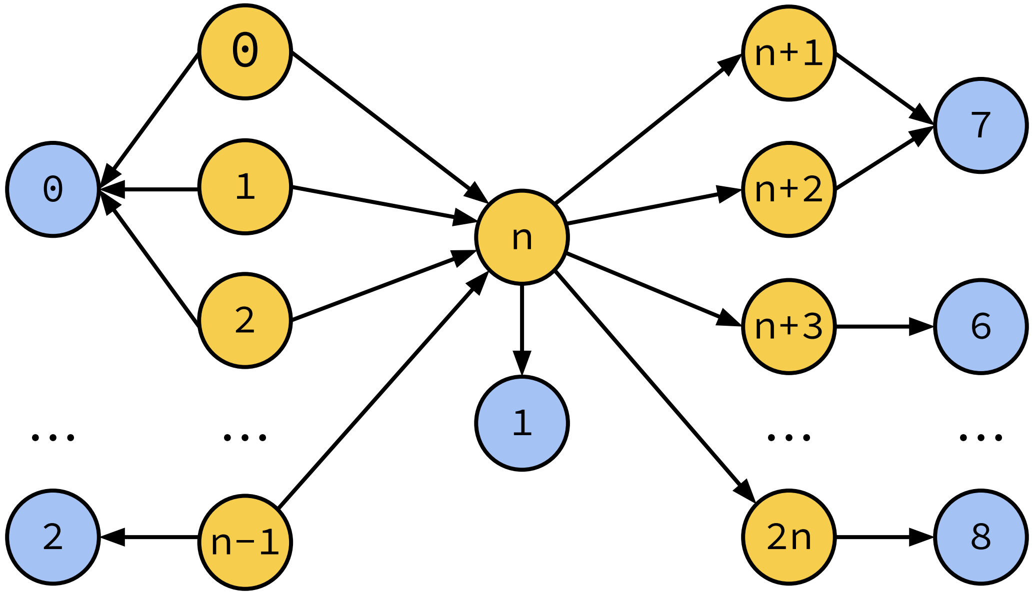
Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

Factorized Vector Execution

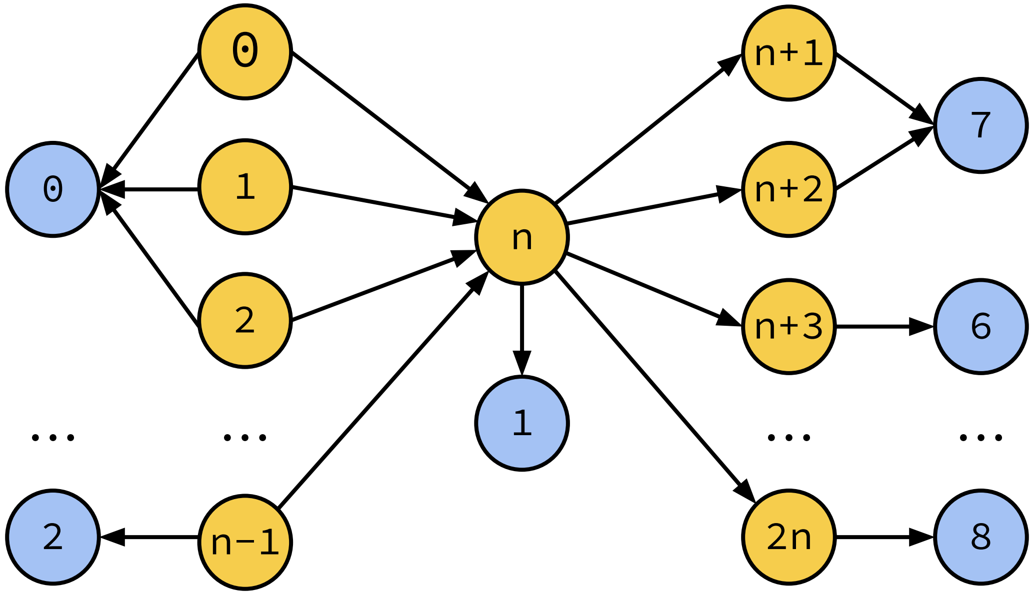
Query Vertex Ordering:
[b, a, c, d]



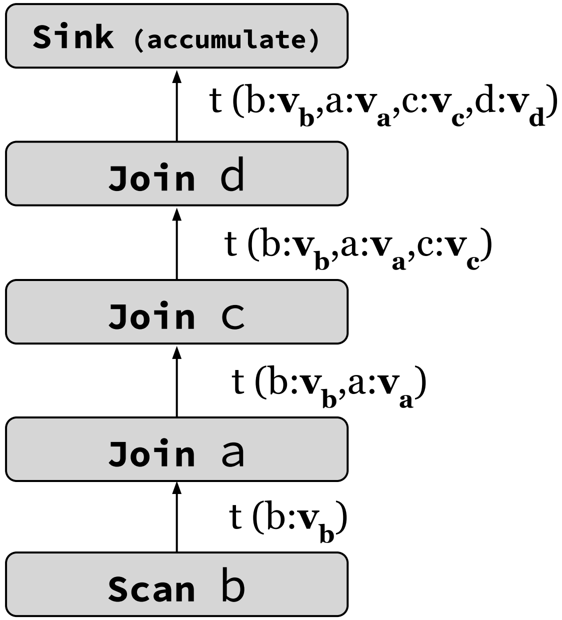
FOLLOWS & LIVES_IN Edges

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

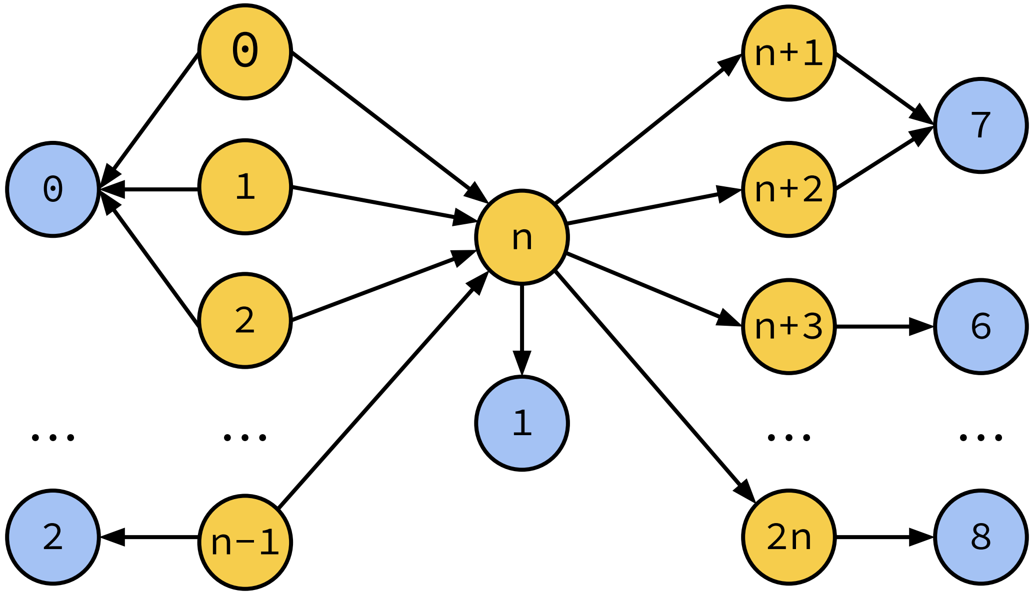


FOLLOWS & LIVES_IN Edges



Factorized Vector Execution

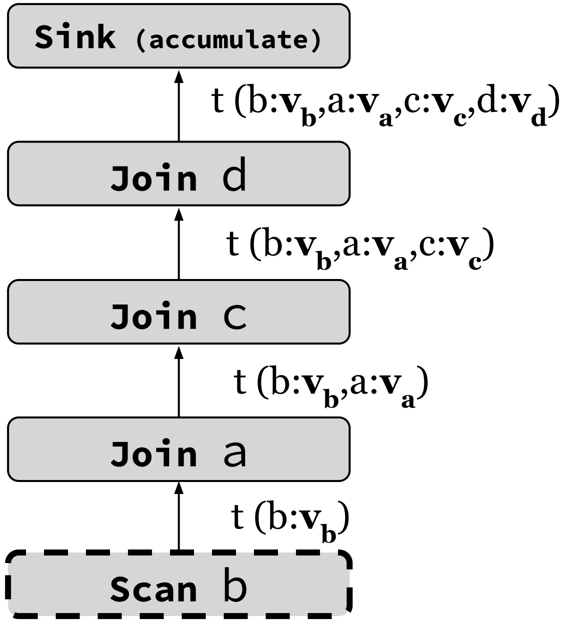
Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

pos → -1 | **size** → 1024



`Sink (accumulate)`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

`Join d`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

`Join c`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

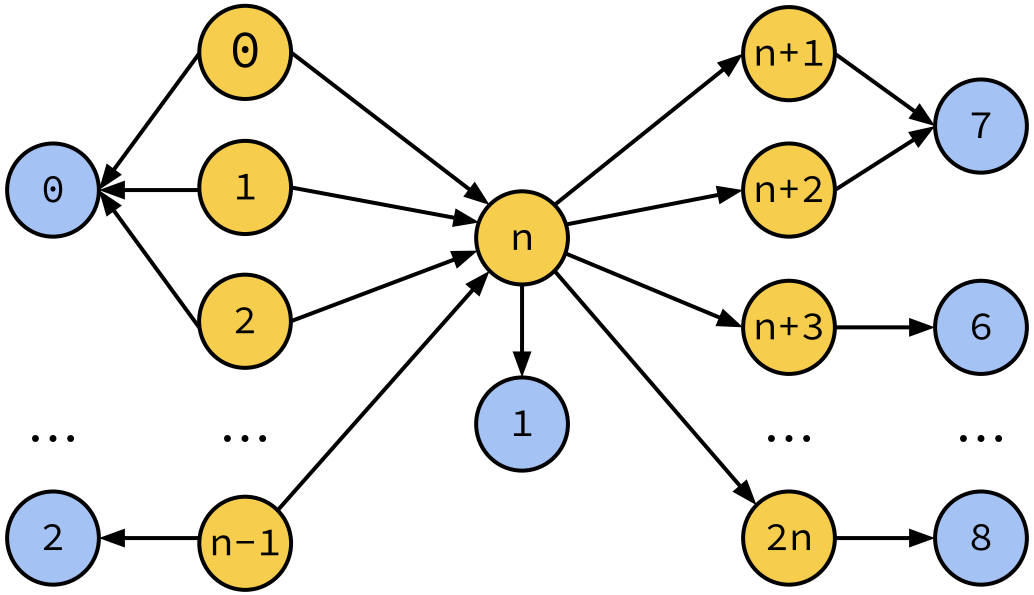
`Join a`

t $(b{:}\mathbf{v}_b)$

`Scan b`

Factorized Vector Execution

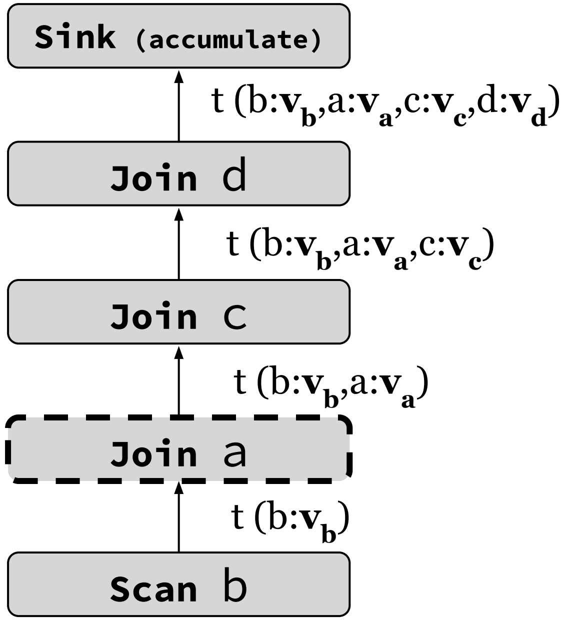
Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

pos → -1 | **size** → 1024



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

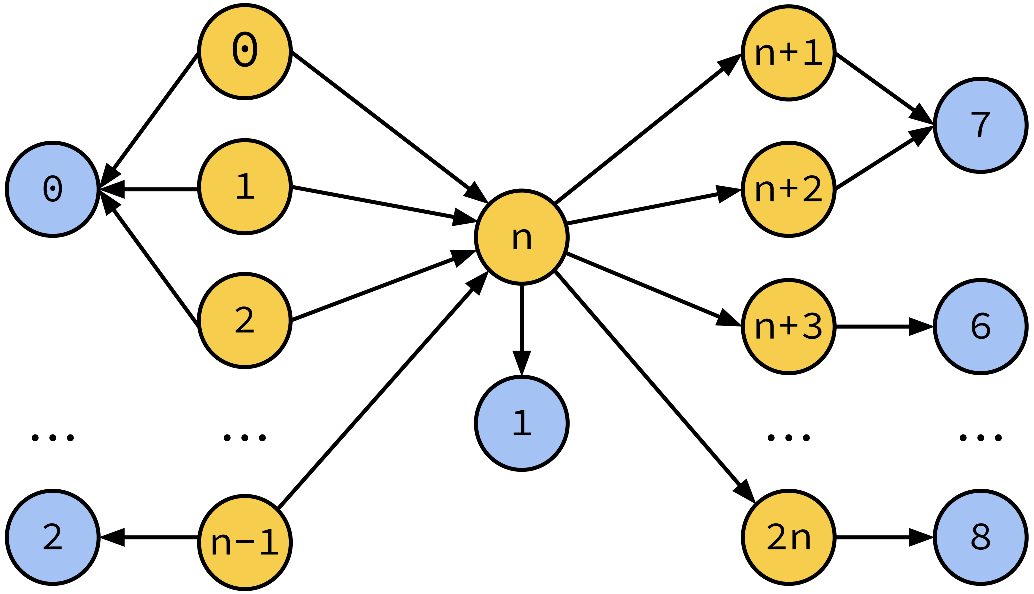
Join a

t (b:$\mathbf{v}_b$)

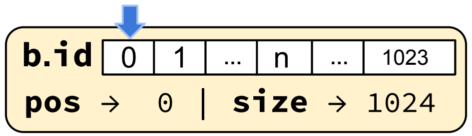
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

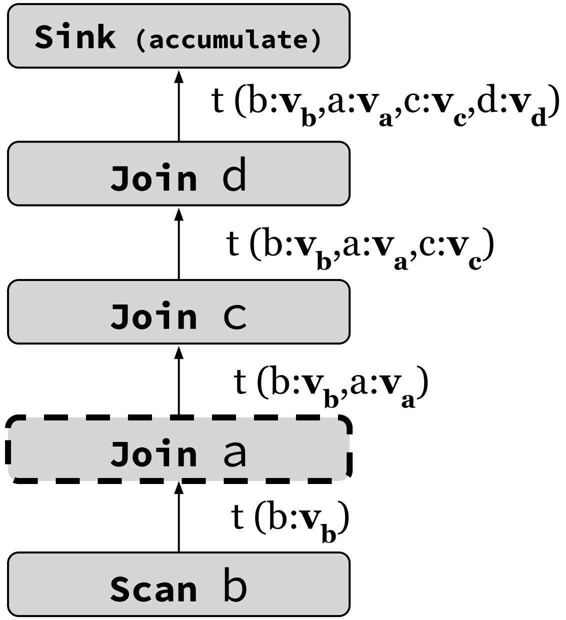


FOLLOWS & LIVES_IN Edges



b.id | 0 | 1 | … | n | … | 1023

pos → 0 | **size** → 1024



`Sink (accumulate)`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

`Join d`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

`Join c`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

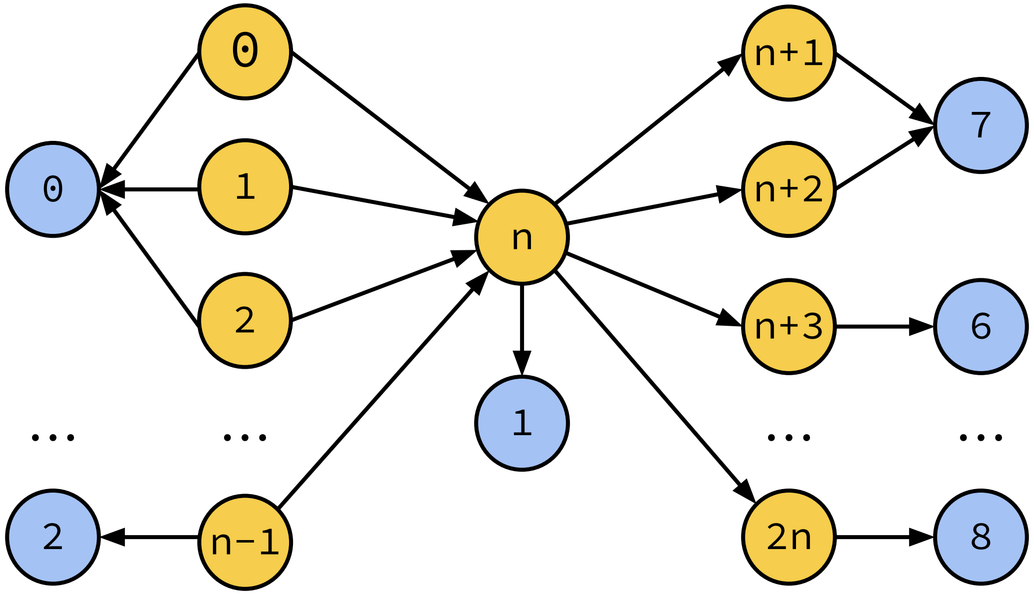
`Join a`

t (b:$\mathbf{v}_b$)

`Scan b`

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

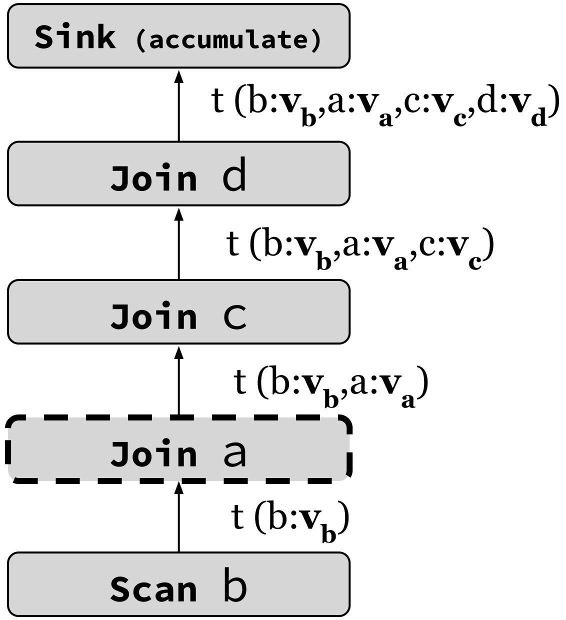


FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

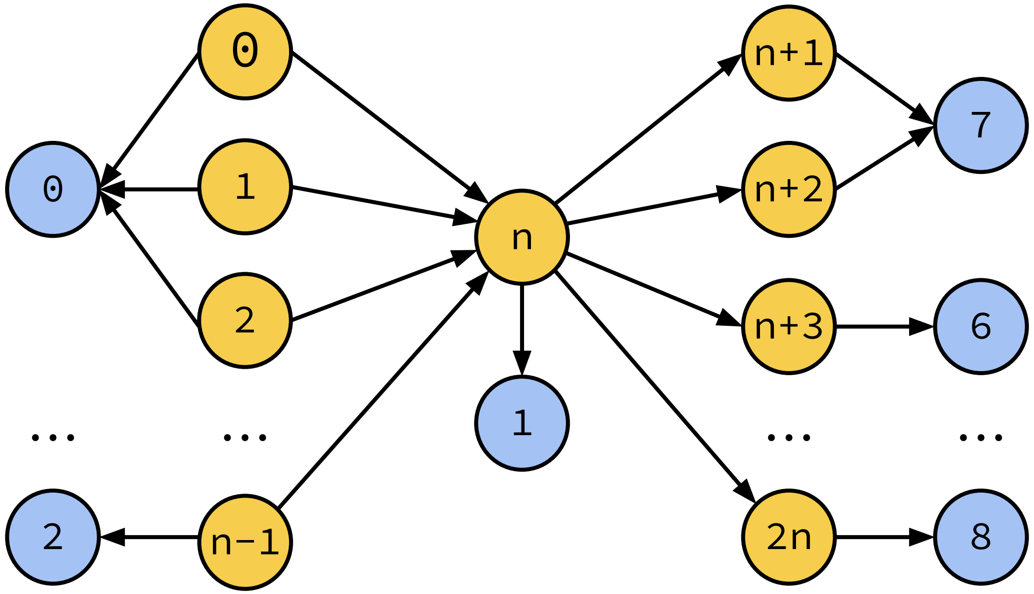
pos → 0 | **size** → 1024

No users follow 0 in the graph

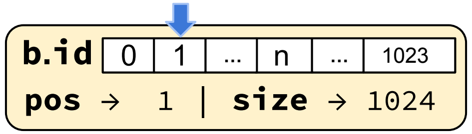


Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



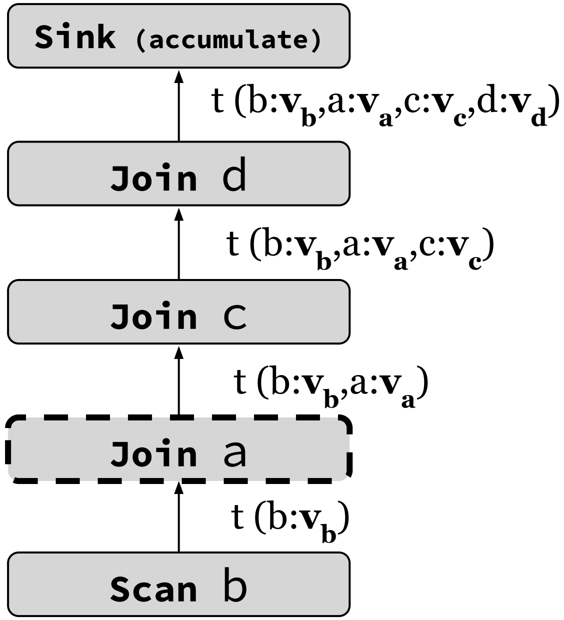
FOLLOWS & LIVES_IN Edges



b.id	0	1	...	n	...	1023

pos → 1 | **size** → 1024

No users follow 1 in the graph



`Sink (accumulate)`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

`Join d`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

`Join c`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

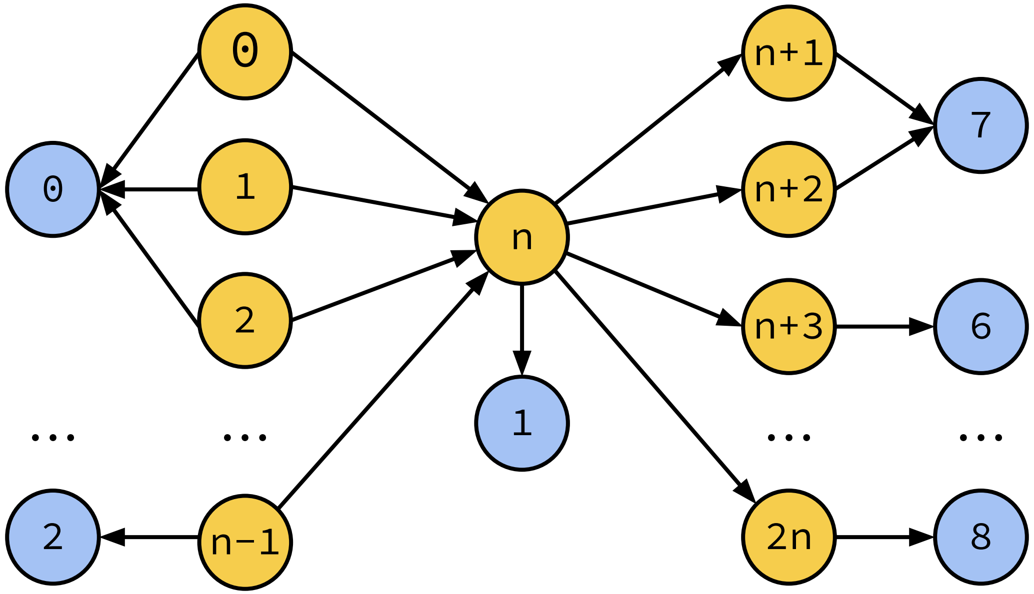
`Join a`

t $(b{:}\mathbf{v}_b)$

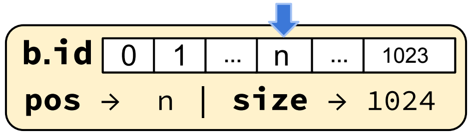
`Scan b`

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

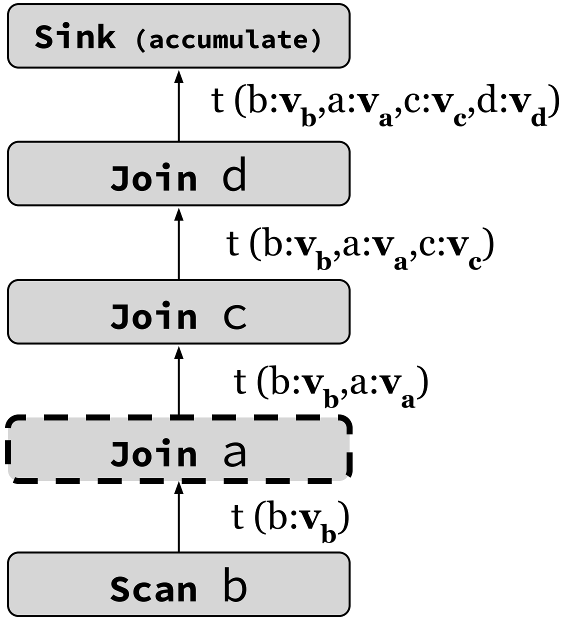


FOLLOWS & LIVES_IN Edges



b.id	0	1	…	n	…	1023

pos → n | **size** → 1024



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

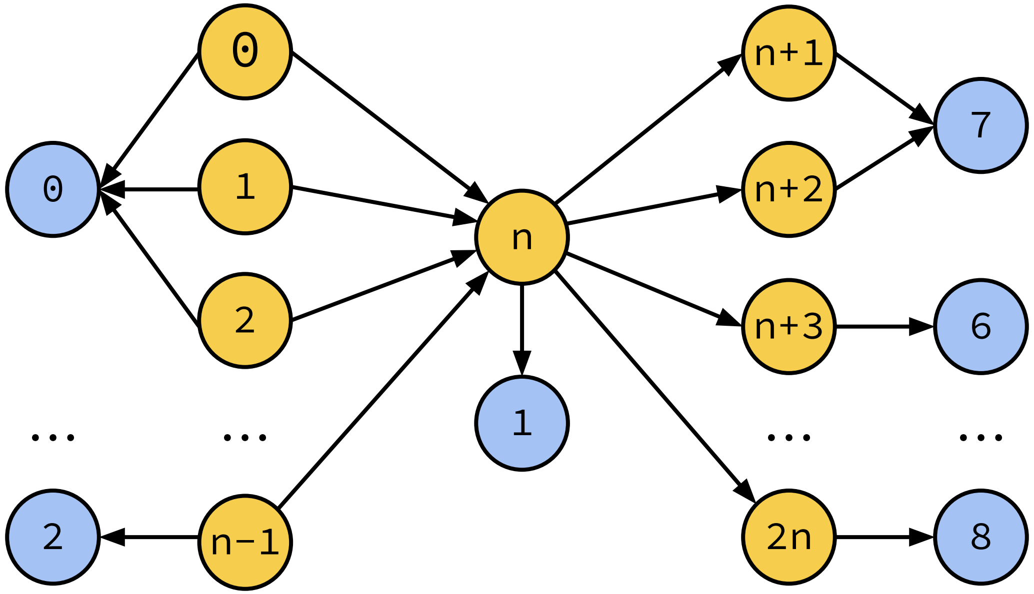
Join a

$t\ (b{:}\mathbf{v}_b)$

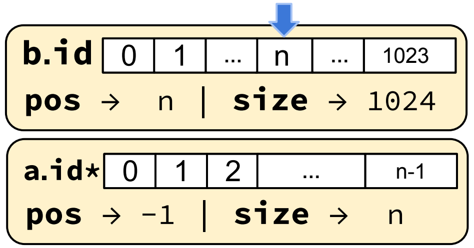
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

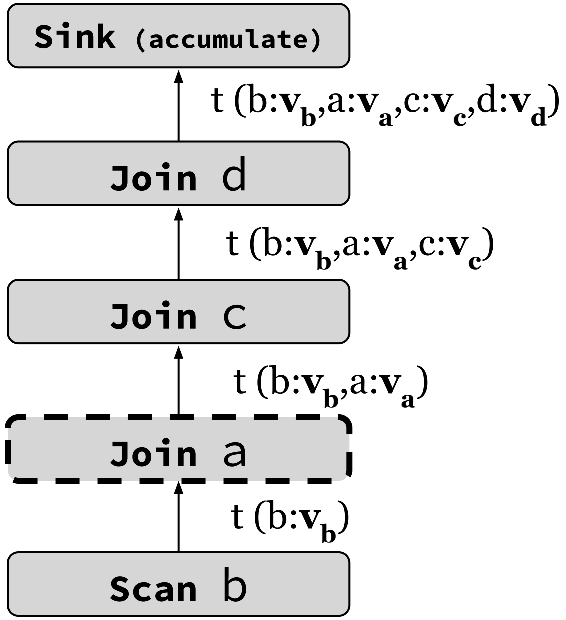


b.id	0	1	...	n	...	1023

pos → n | **size** → 1024

a.id*	0	1	2	...	n-1

pos → -1 | **size** → n



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

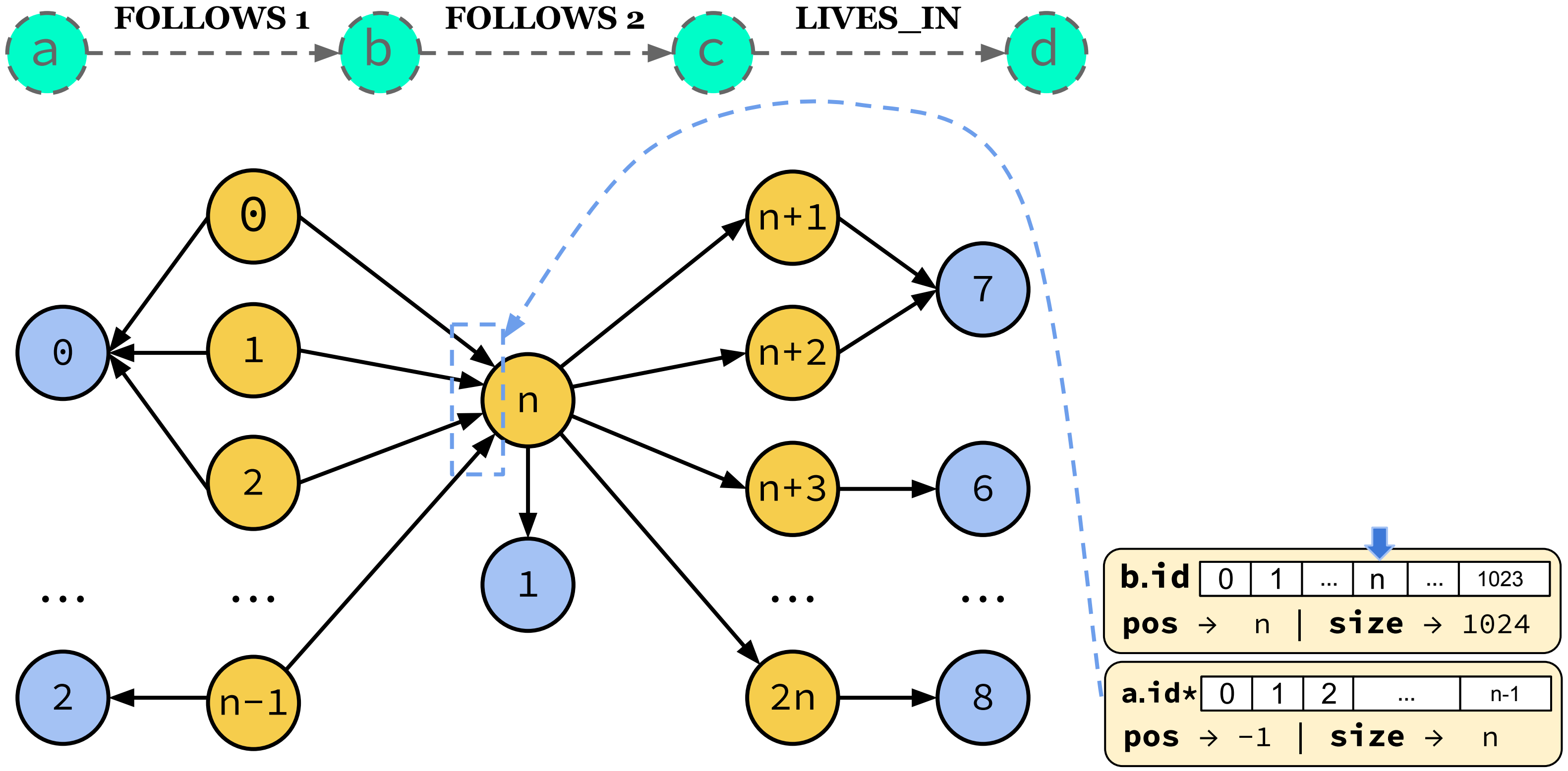
Join a

t (b:$\mathbf{v}_b$)

Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



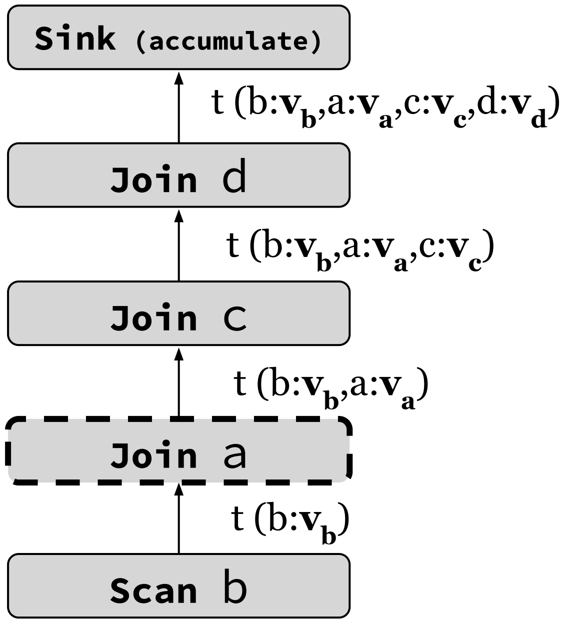
b.id	0	1	…	n	…	1023

pos → n | **size** → 1024

a.id*	0	1	2	…	n-1

pos → -1 | **size** → n

FOLLOWS & LIVES_IN Edges



`Sink (accumulate)`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

`Join d`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

`Join c`

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

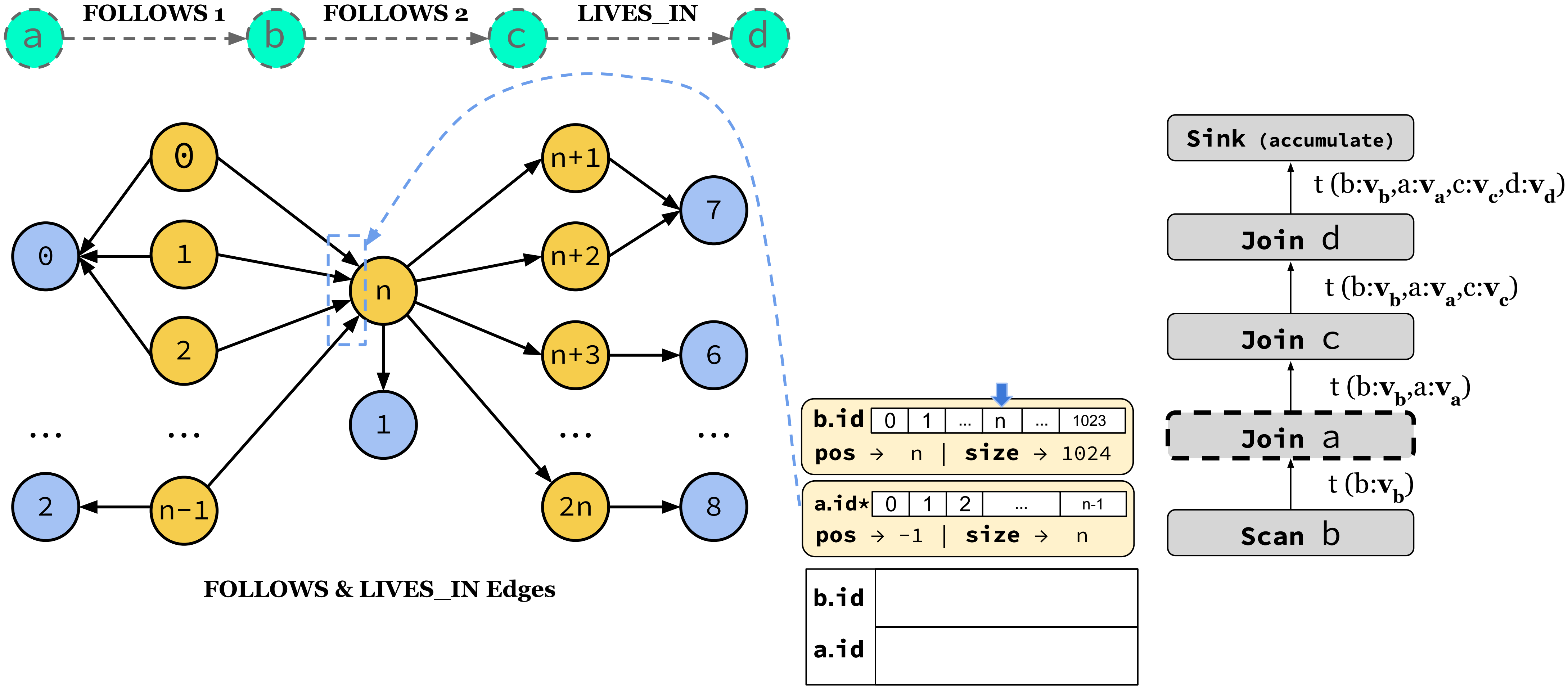
`Join a`

t $(b{:}\mathbf{v}_b)$

`Scan b`

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

pos → n | **size** → 1024

a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

b.id	
a.id	

`Sink (accumulate)`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

`Join d`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

`Join c`

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

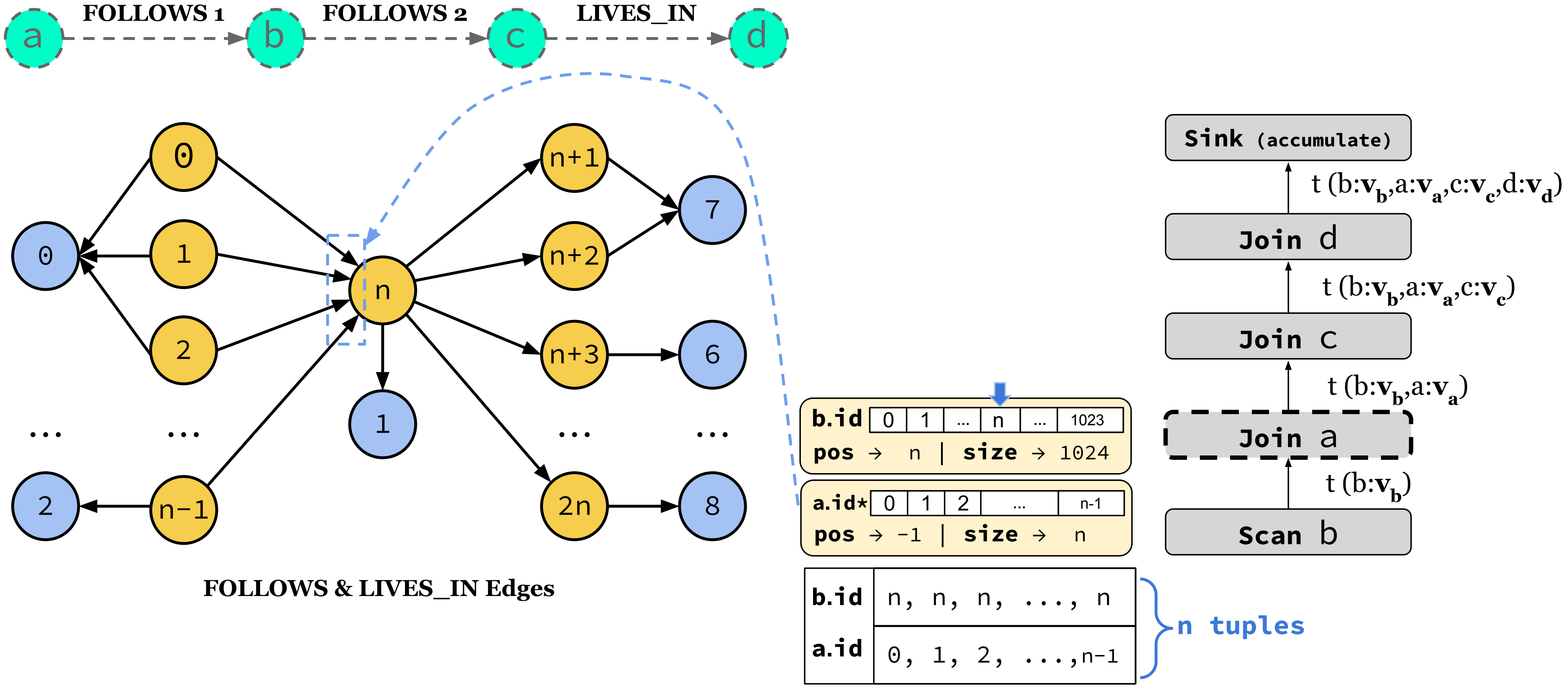
`Join a`

t (b:$\mathbf{v}_b$)

`Scan b`

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id | 0 | 1 | … | n | … | 1023

pos → n | **size** → 1024

a.id* | 0 | 1 | 2 | … | n-1

pos → -1 | **size** → n

b.id	n, n, n, ..., n
a.id	0, 1, 2, ...,n-1

n tuples

Sink (accumulate)

t ($b:\mathbf{v}_b, a:\mathbf{v}_a, c:\mathbf{v}_c, d:\mathbf{v}_d$)

Join d

t ($b:\mathbf{v}_b, a:\mathbf{v}_a, c:\mathbf{v}_c$)

Join c

t ($b:\mathbf{v}_b, a:\mathbf{v}_a$)

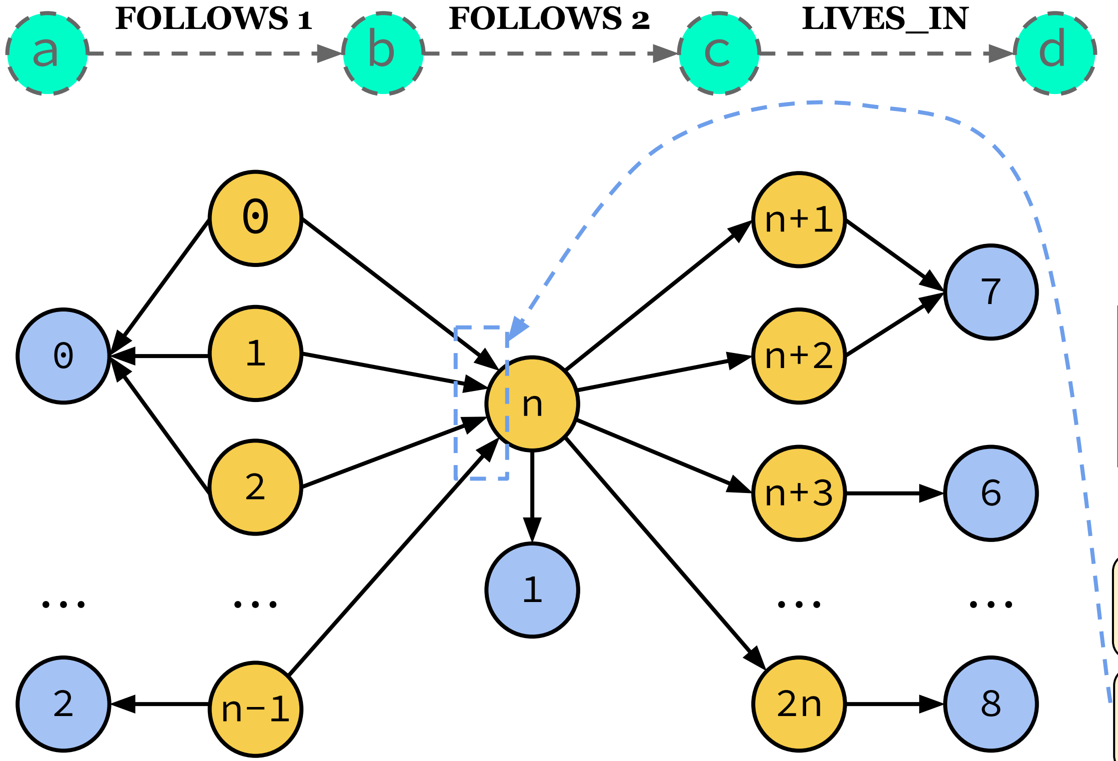
Join a

t ($b:\mathbf{v}_b$)

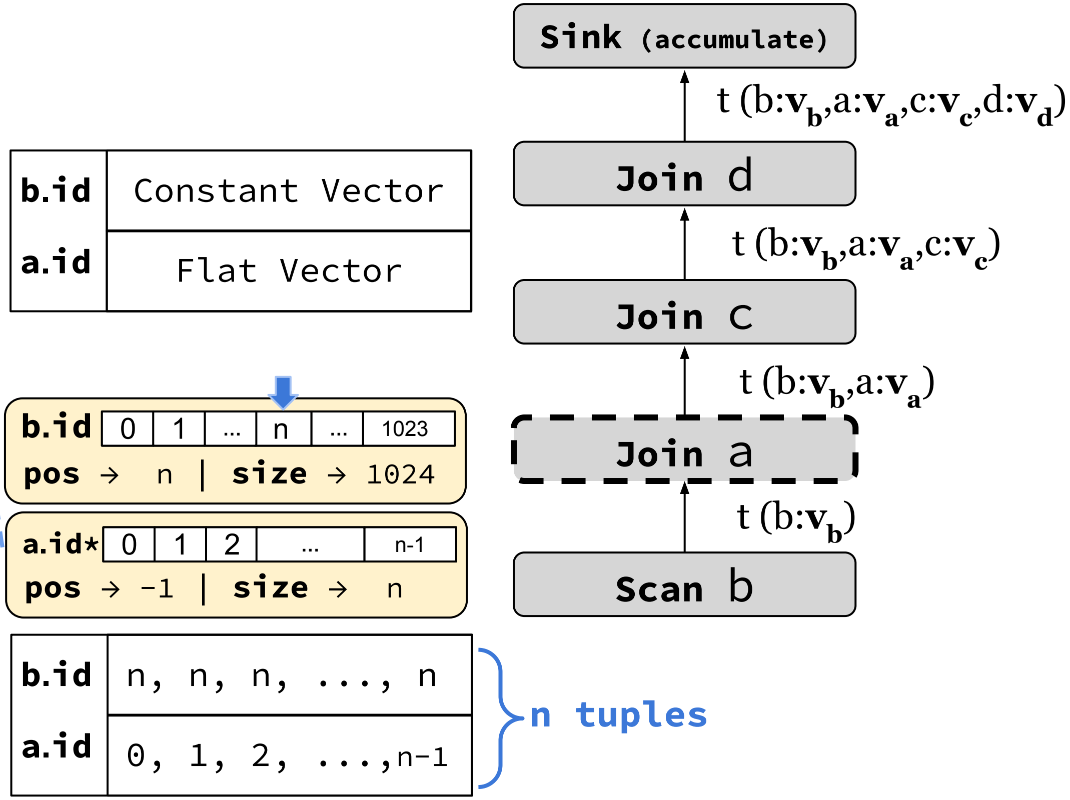
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges



b.id	Constant Vector
a.id	Flat Vector

b.id 0 | 1 | ... | n | ... | 1023
pos → n | **size** → 1024

a.id* 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

b.id	n, n, n, ..., n
a.id	0, 1, 2, ...,n-1

n tuples

Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

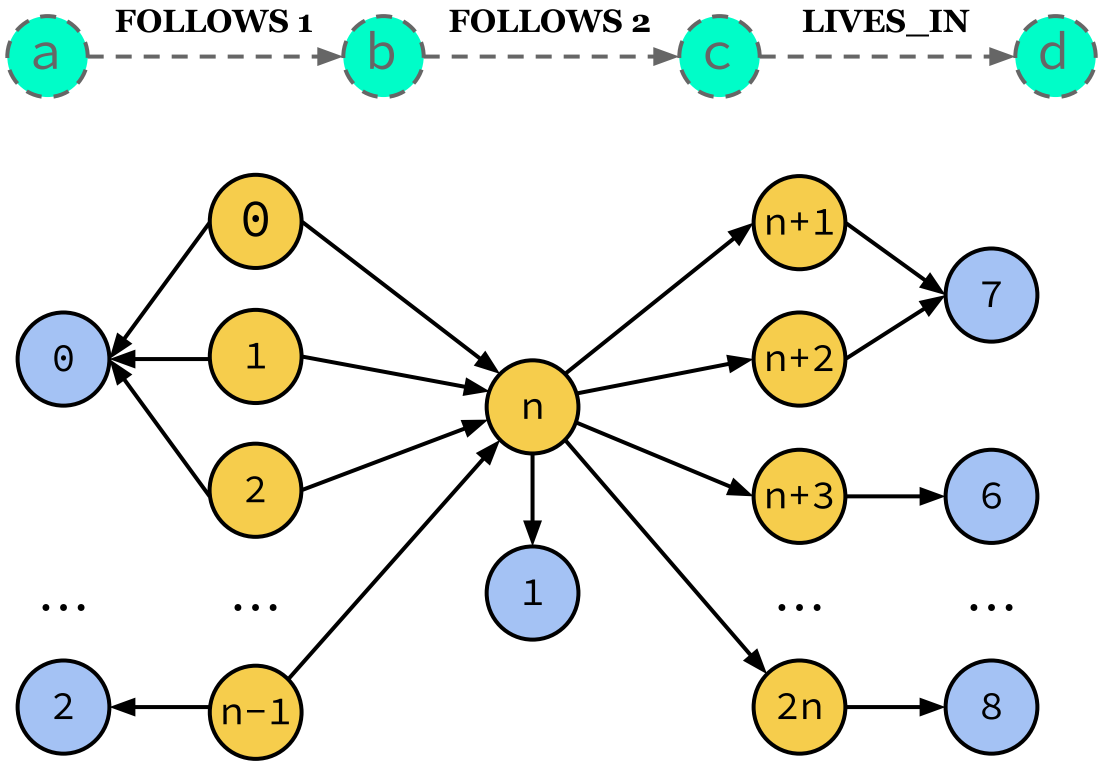
Join a

t $(b{:}\mathbf{v}_b)$

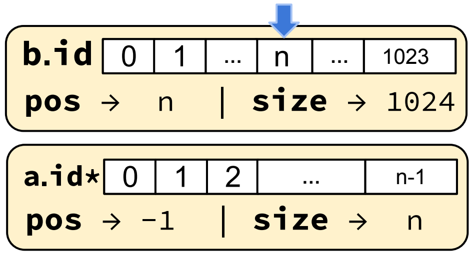
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

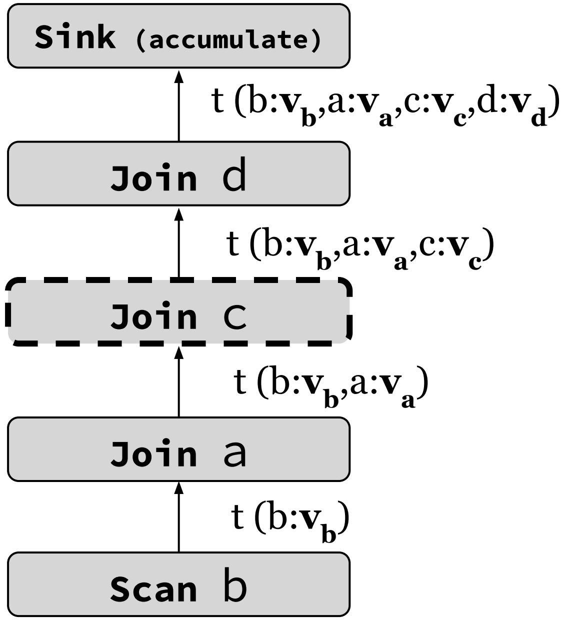


b.id	0	1	…	n	…	1023

pos → n | **size** → 1024

a.id*	0	1	2	…	n-1

pos → -1 | **size** → n



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

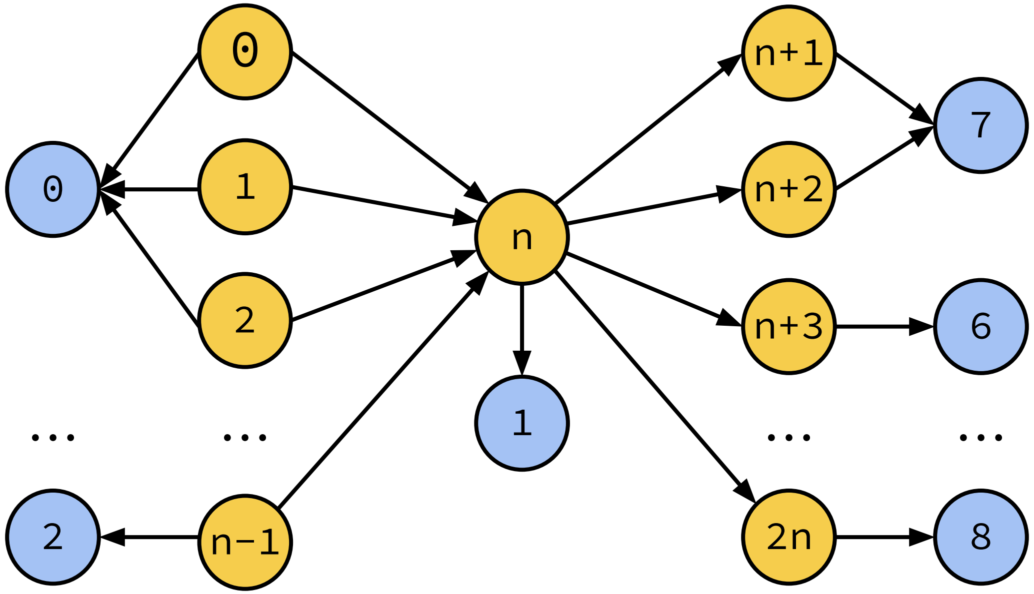
Join a

t (b:$\mathbf{v}_b$)

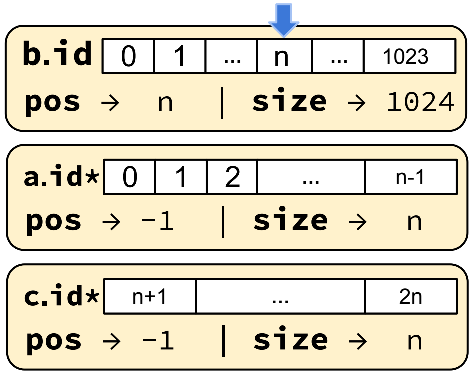
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges



b.id	0	1	...	n	...	1023

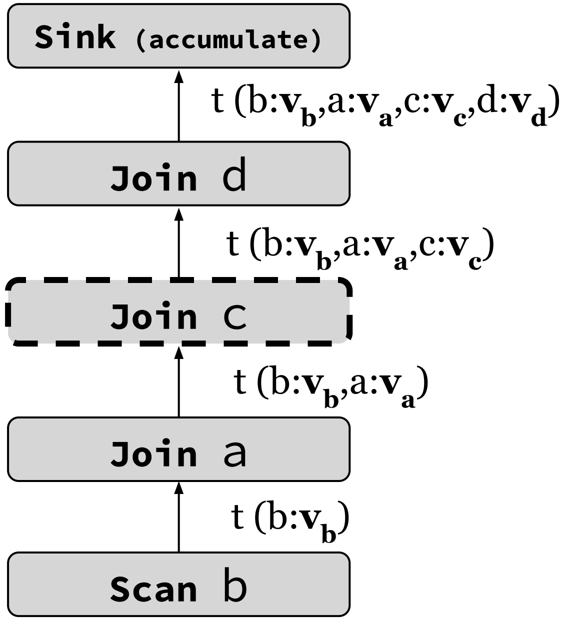
pos → n | **size** → 1024

a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

c.id*	n+1	...	2n

pos → -1 | **size** → n



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

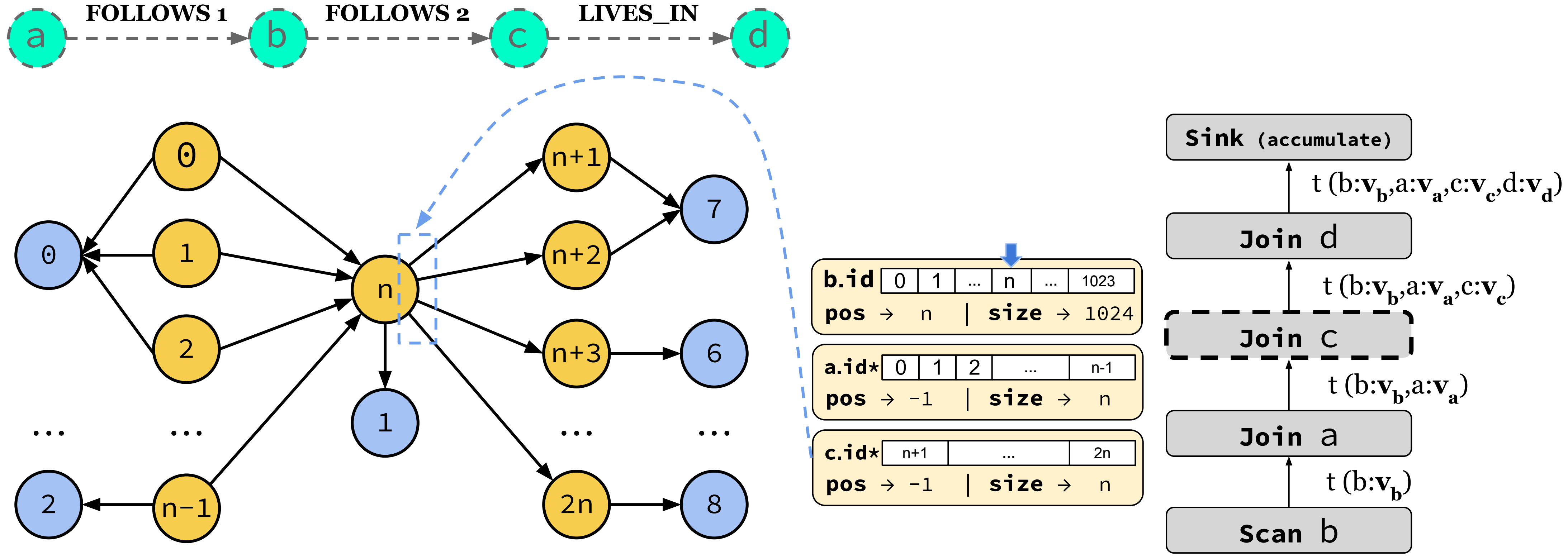
Join a

t $(b{:}\mathbf{v}_b)$

Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id | 0 | 1 | ... | n | ... | 1023
pos → n | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

c.id* | n+1 | ... | 2n
pos → -1 | **size** → n

Sink (accumulate)

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

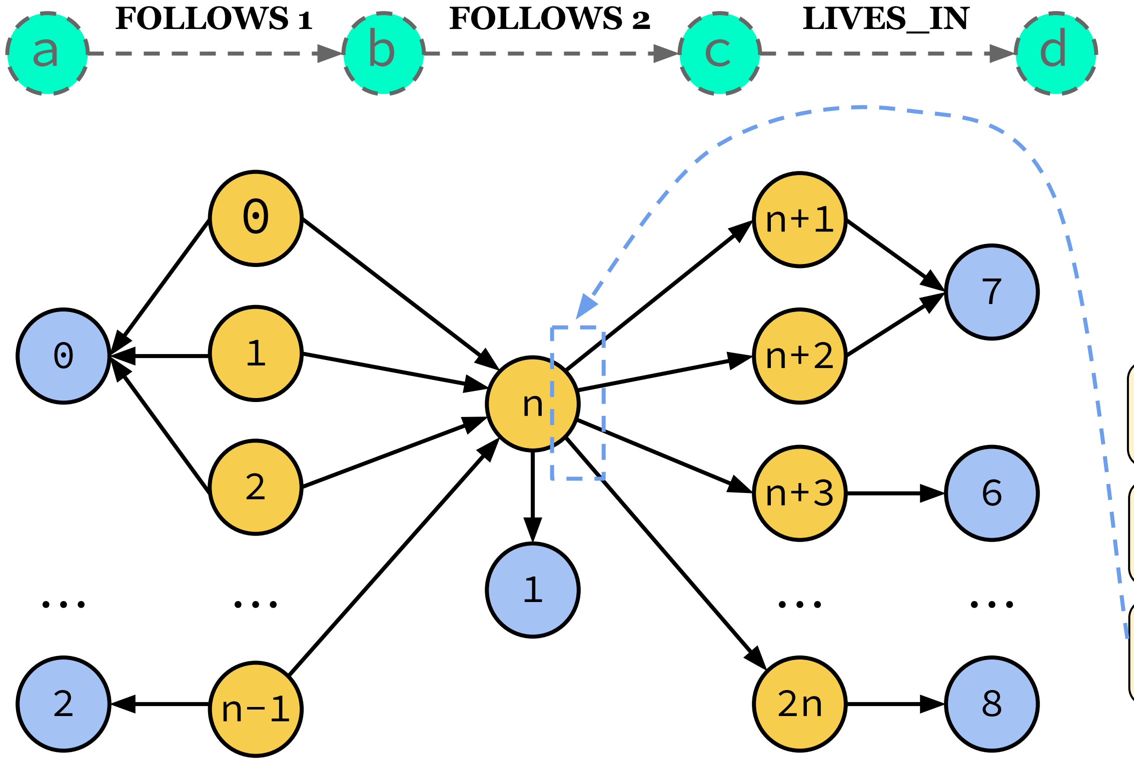
Join a

$t\,(b{:}\mathbf{v}_b)$

Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

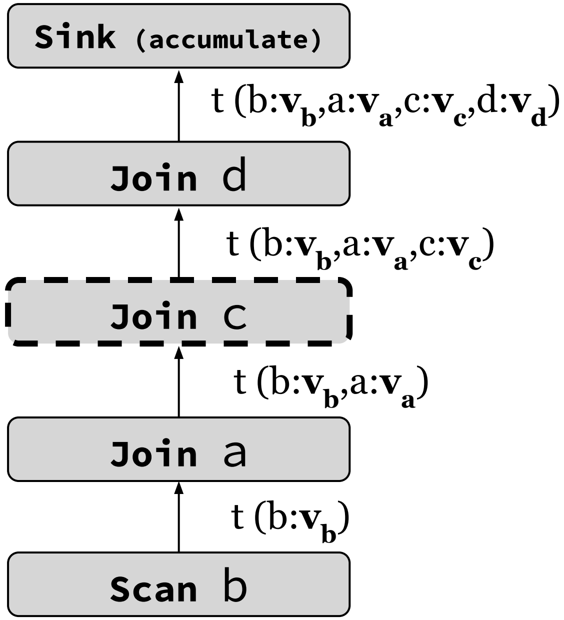
pos → n | **size** → 1024

a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

c.id*	n+1	...	2n

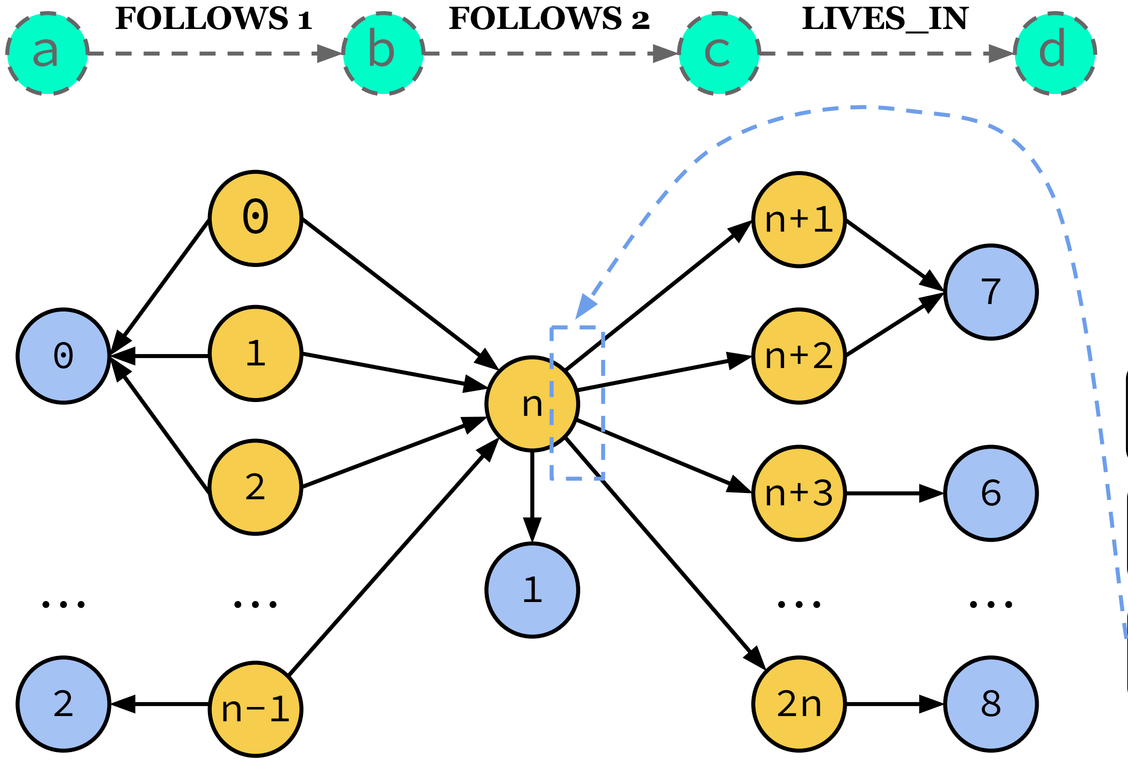
pos → -1 | **size** → n



b.id	
a.id	
c.id	

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

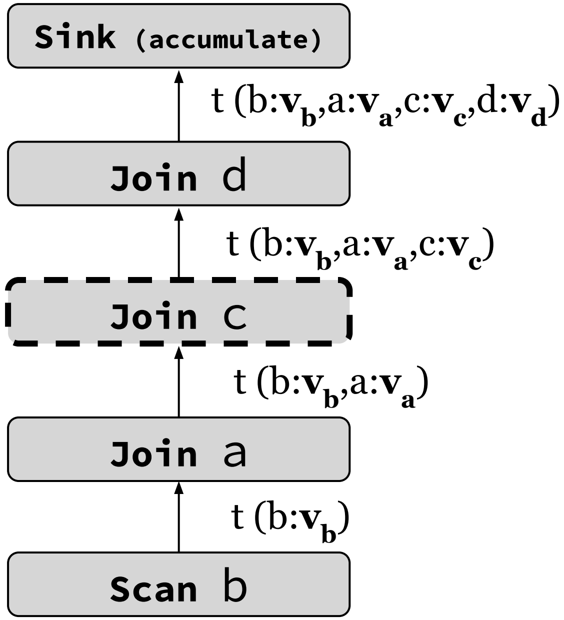


FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023
pos → n	**size** → 1024					

a.id*	0	1	2	...	n-1
pos → -1	**size** → n				

c.id*	n+1	...	2n
pos → -1	**size** → n		



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

Join a

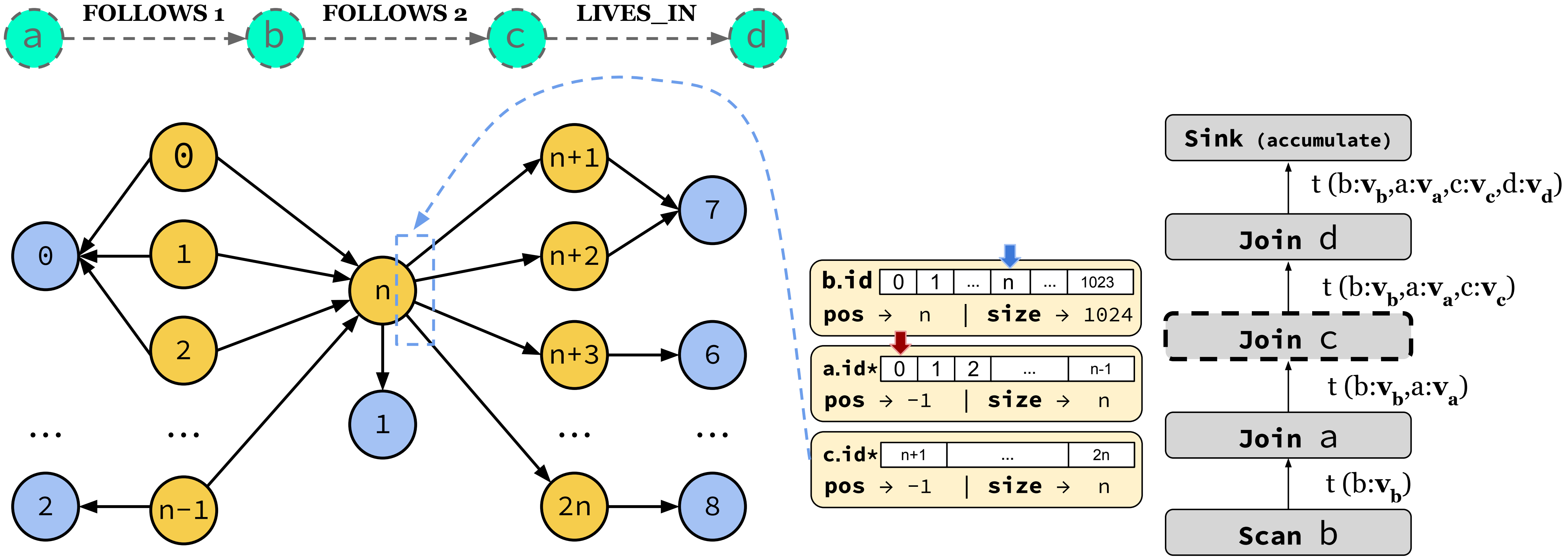
t (b:$\mathbf{v}_b$)

Scan b

b.id	n
a.id	0
c.id	

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



b.id | 0 | 1 | ... | n | ... | 1023
pos → n | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

c.id* | n+1 | ... | 2n
pos → -1 | **size** → n

Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

Join a

t (b:$\mathbf{v}_b$)

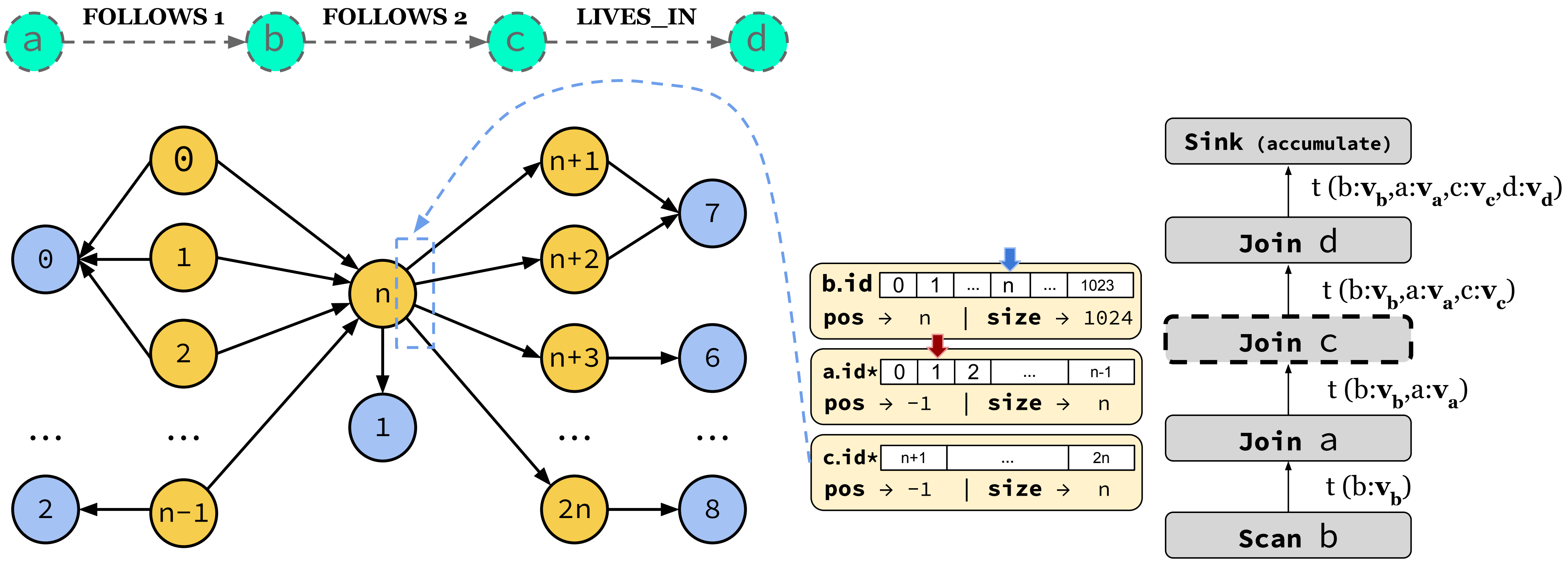
Scan b

FOLLOWS & LIVES_IN Edges

b.id	n,...,n
a.id	0,...,0
c.id	n+1,...,2n

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

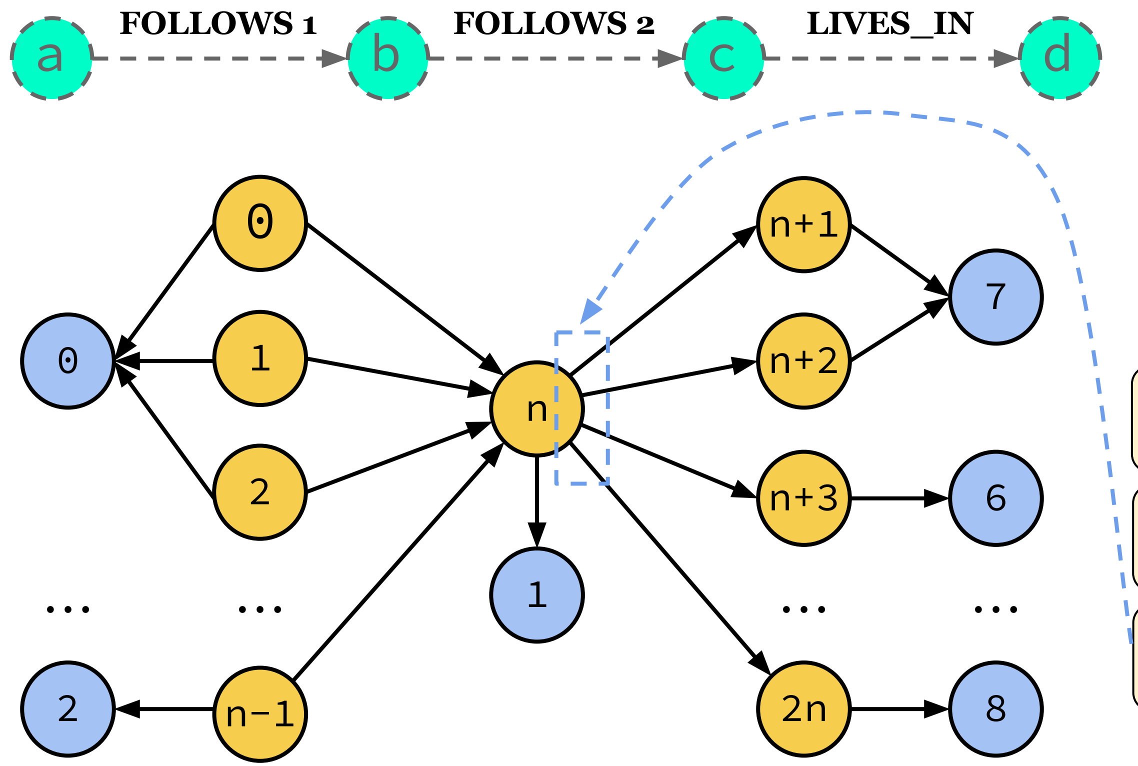


FOLLOWS & LIVES_IN Edges

b.id	n,..., n	n
a.id	0,..., 0	1
c.id	n+1,...,2n	

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

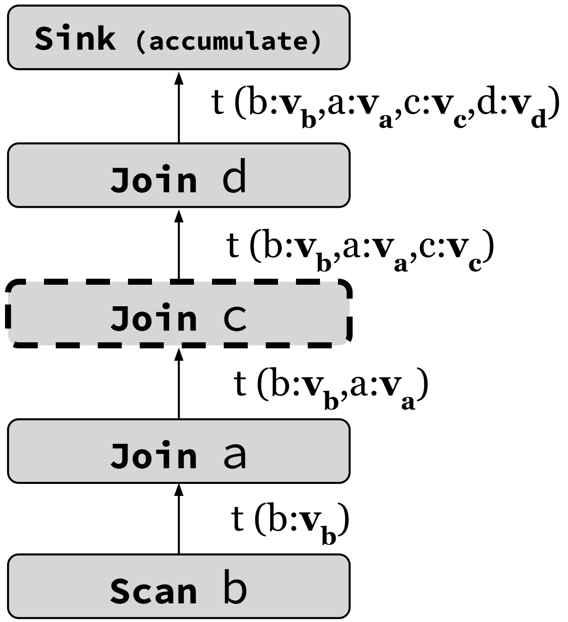
pos → n | **size** → 1024

a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

c.id*	n+1	...	2n

pos → -1 | **size** → n



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

Join a

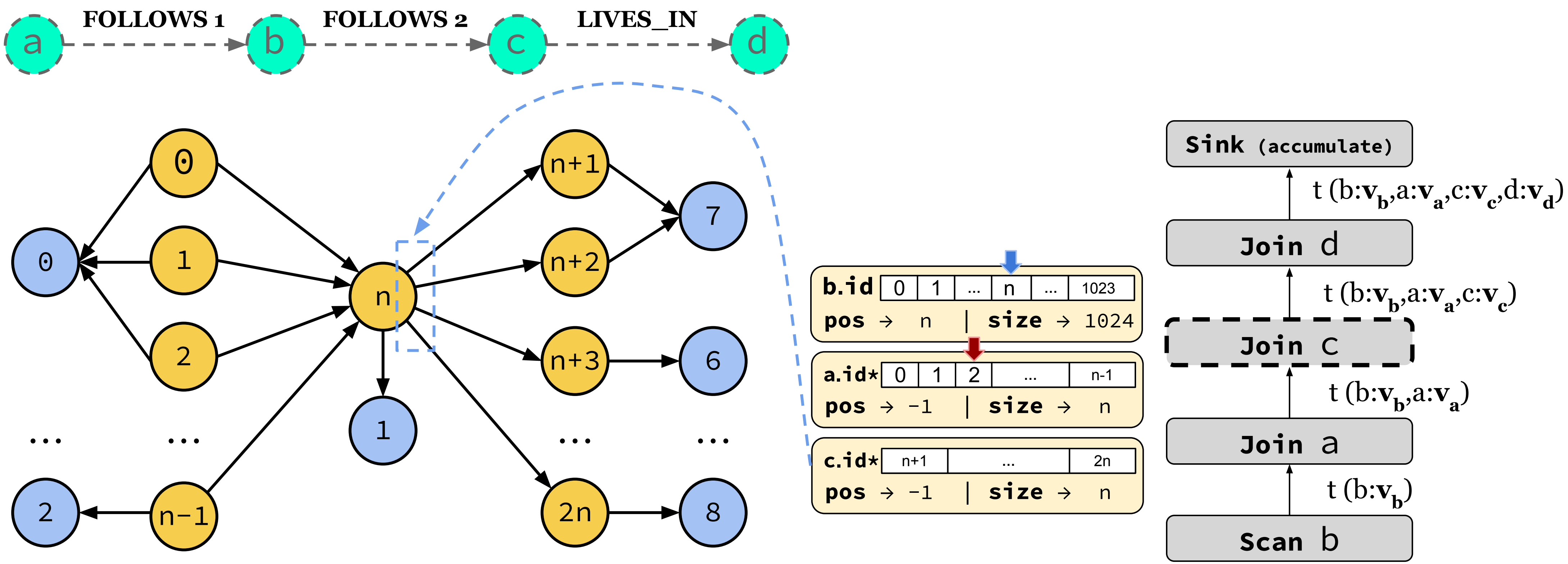
t $(b{:}\mathbf{v}_b)$

Scan b

b.id	n,..., n	n,..., n
a.id	0,..., 0	1,..., 1
c.id	n+1,...,2n	n+1,...,2n

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

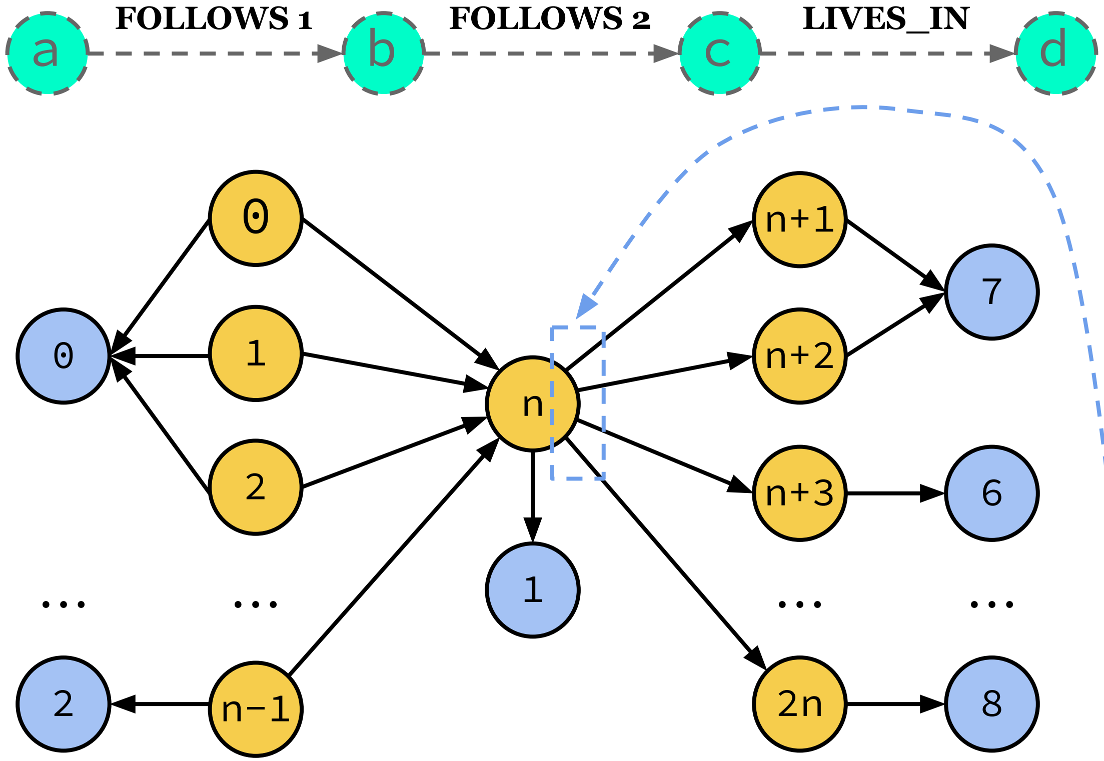


FOLLOWS & LIVES_IN Edges

b.id	n,..., n	n,..., n	n
a.id	0,..., 0	1,..., 1	2
c.id	n+1,...,2n	n+1,...,2n	

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

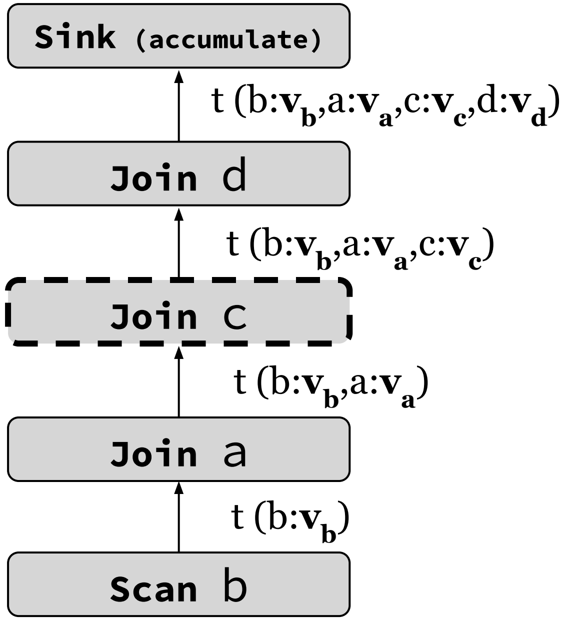


FOLLOWS & LIVES_IN Edges

b.id | 0 | 1 | ... | n | ... | 1023
pos → n | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

c.id* | n+1 | ... | 2n
pos → -1 | **size** → n



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

Join a

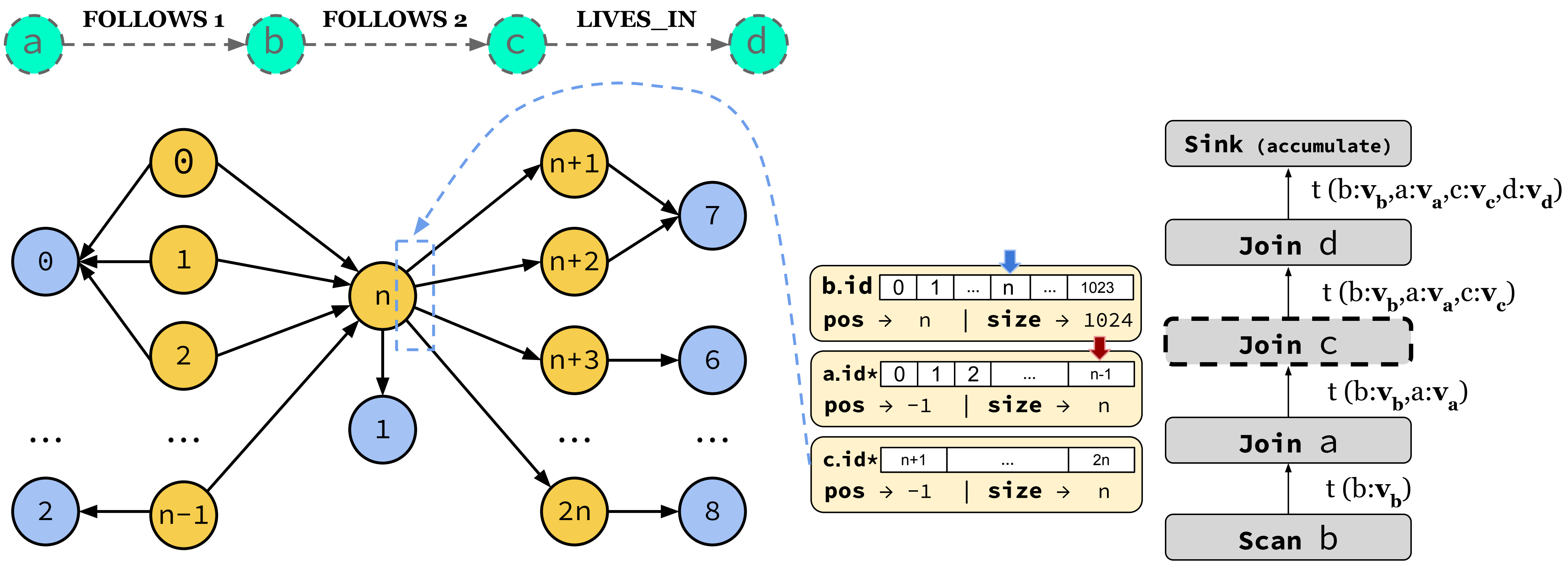
$t\ (b{:}\mathbf{v}_b)$

Scan b

b.id	n,..., n	n,..., n	n,..., n	
a.id	0,..., 0	1,..., 1	2,..., 2	
c.id	n+1,...,2n	n+1,...,2n	n+1,...,2n	

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]

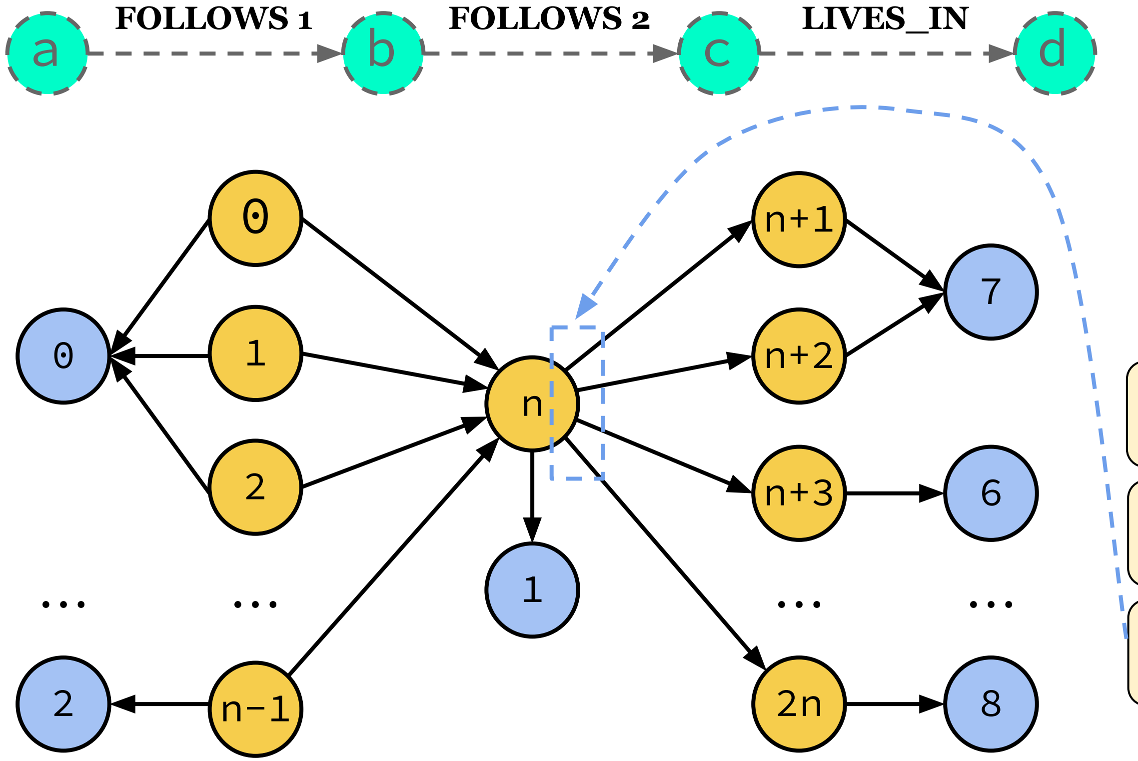


FOLLOWS & LIVES_IN Edges

b.id	n,..., n	n,..., n	n,..., n	...	n,..., n
a.id	0,..., 0	1,..., 1	2,..., 2	...	n-1,..., n-1
c.id	n+1,...,2n	n+1,...,2n	n+1,...,2n	...	n-1,..., n-1

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



b.id | 0 | 1 | ... | n | ... | 1023 |
pos → n | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1 |
pos → -1 | **size** → n

c.id* | n+1 | ... | 2n |
pos → -1 | **size** → n

Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

Join a

$t\ (b{:}\mathbf{v}_b)$

Scan b

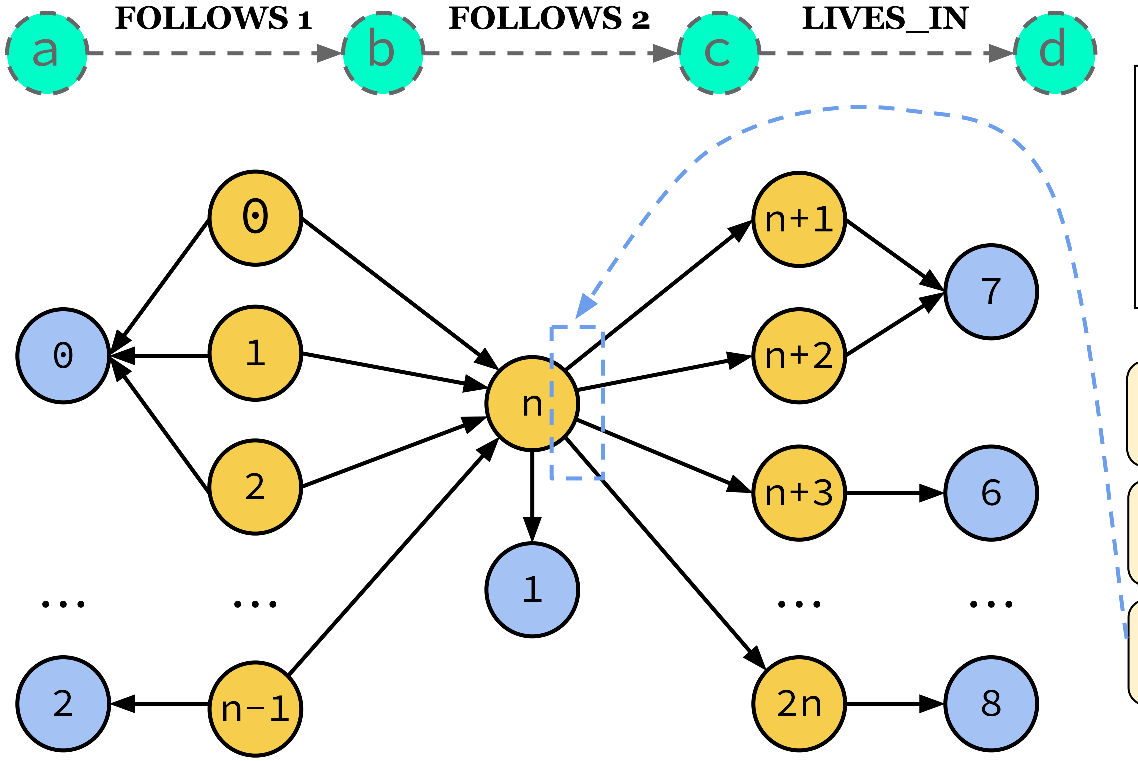
FOLLOWS & LIVES_IN Edges

after flattening:
n^2 tuples

b.id	n,..., n	n,..., n	n,..., n	...	n,..., n
a.id	0,..., 0	1,..., 1	2,..., 2	...	n-1,..., n-1
c.id	n+1,...,2n	n+1,...,2n	n+1,...,2n	...	n-1,..., n-1

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



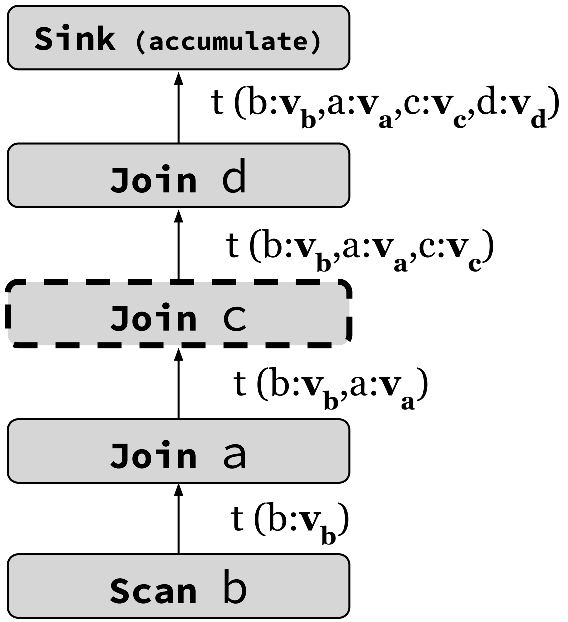
FOLLOWS & LIVES_IN Edges

b.id	Constant Vector
a.id	Flat Vector
c.id	Flat Vector

b.id | 0 | 1 | … | n | … | 1023 |
pos → n | **size** → 1024

a.id* | 0 | 1 | 2 | … | n-1 |
pos → -1 | **size** → n

c.id* | n+1 | … | 2n |
pos → -1 | **size** → n

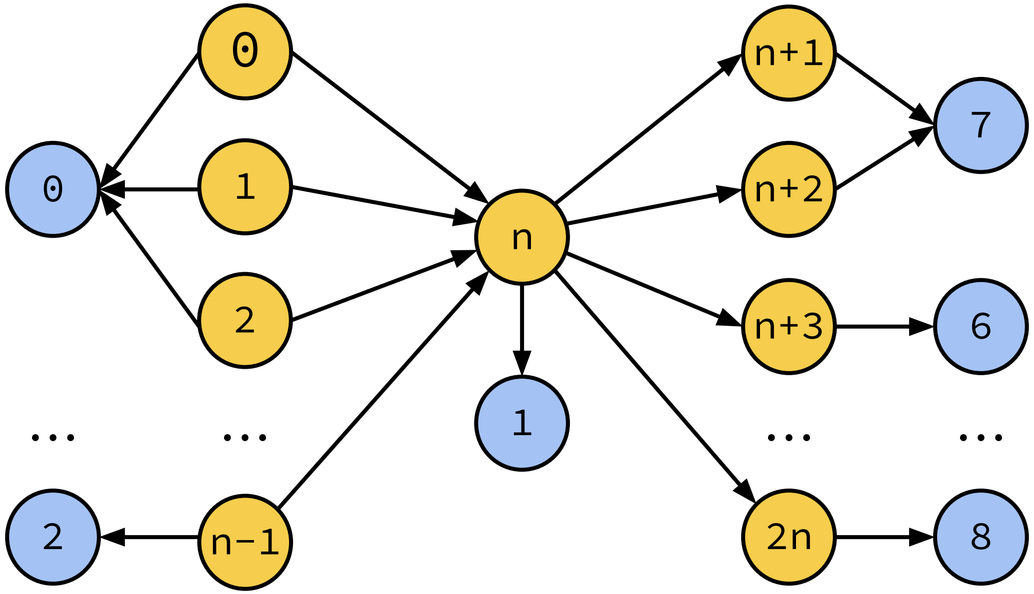


after flattening:
n^2 tuples

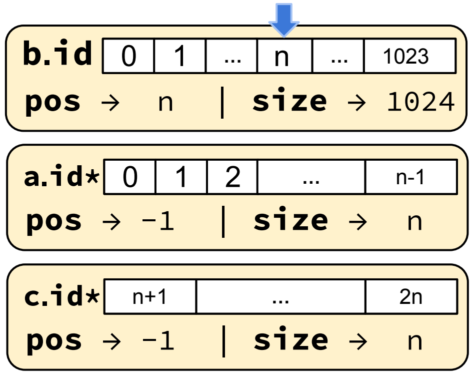
b.id	n,..., n	n,..., n	n,..., n	...	n,..., n
a.id	0,..., 0	1,..., 1	2,..., 2	...	n-1,..., n-1
c.id	n+1,...,2n	n+1,...,2n	n+1,...,2n	...	n-1,..., n-1

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



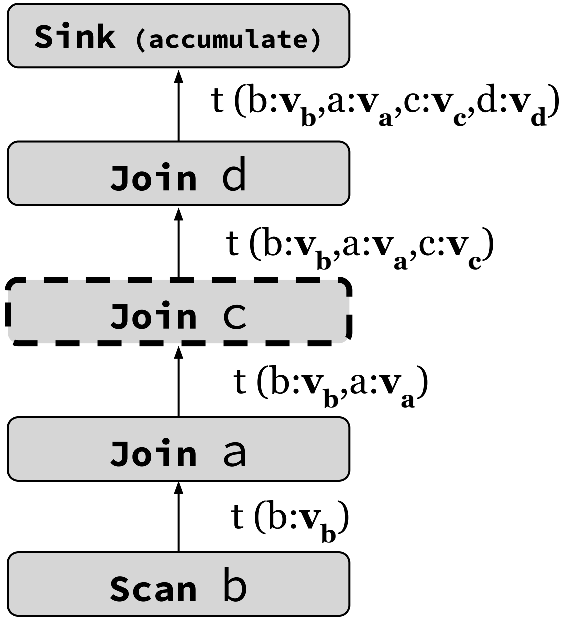
FOLLOWS & LIVES_IN Edges



b.id 0 | 1 | … | n | … | 1023
pos → n | **size** → 1024

a.id* 0 | 1 | 2 | … | n-1
pos → -1 | **size** → n

c.id* n+1 | … | 2n
pos → -1 | **size** → n



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

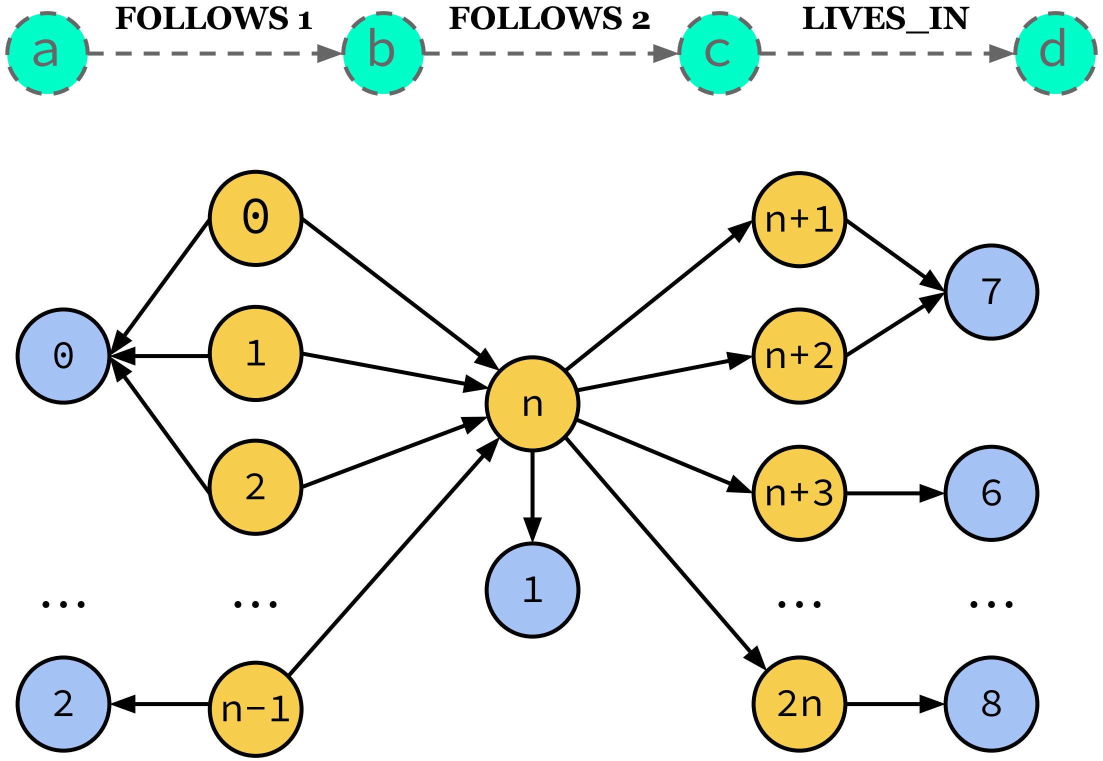
Join a

t (b:$\mathbf{v}_b$)

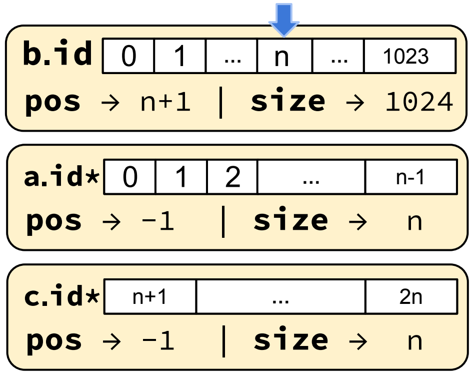
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



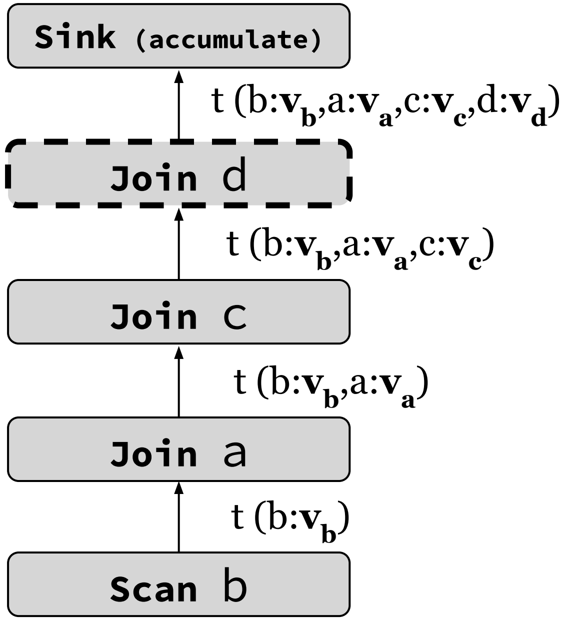
FOLLOWS & LIVES_IN Edges



b.id | 0 | 1 | … | n | … | 1023
pos → n+1 | **size** → 1024

a.id* | 0 | 1 | 2 | … | n-1
pos → -1 | **size** → n

c.id* | n+1 | … | 2n
pos → -1 | **size** → n



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

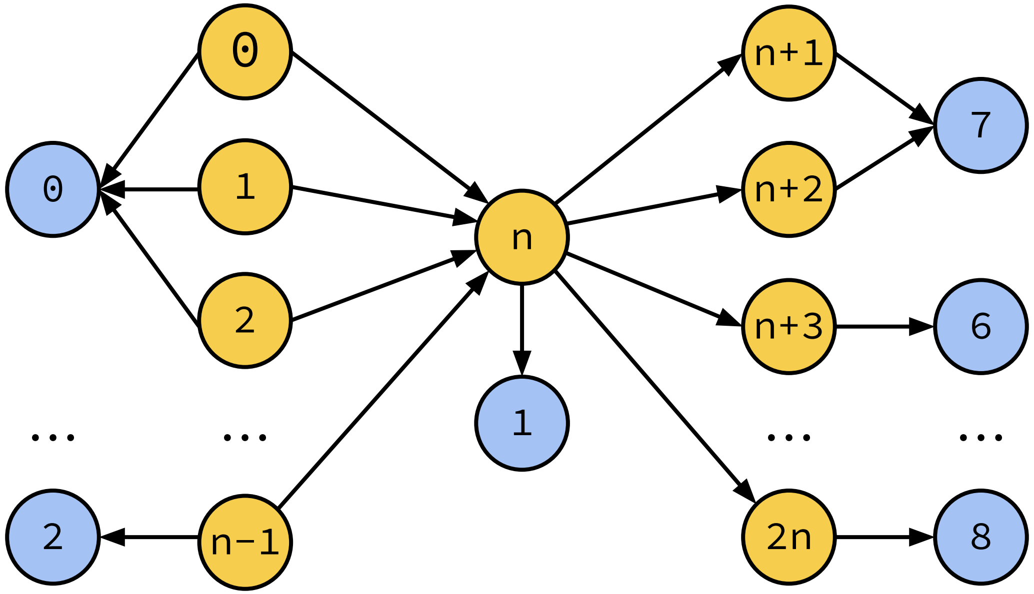
Join a

t $(b{:}\mathbf{v}_b)$

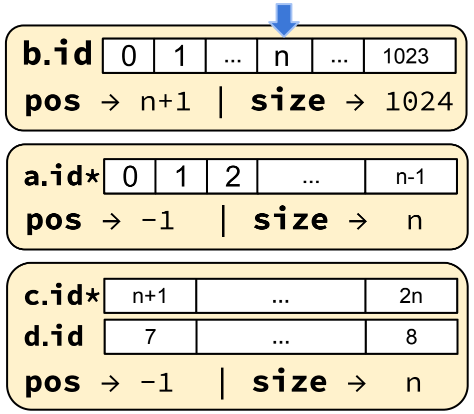
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



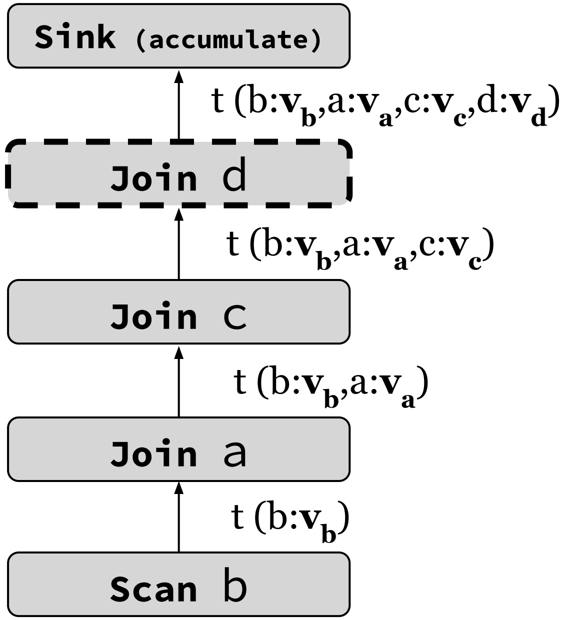
FOLLOWS & LIVES_IN Edges



b.id 0 | 1 | … | n | … | 1023
pos → n+1 | **size** → 1024

a.id* 0 | 1 | 2 | … | n-1
pos → -1 | **size** → n

c.id* n+1 | … | 2n
d.id 7 | … | 8
pos → -1 | **size** → n



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

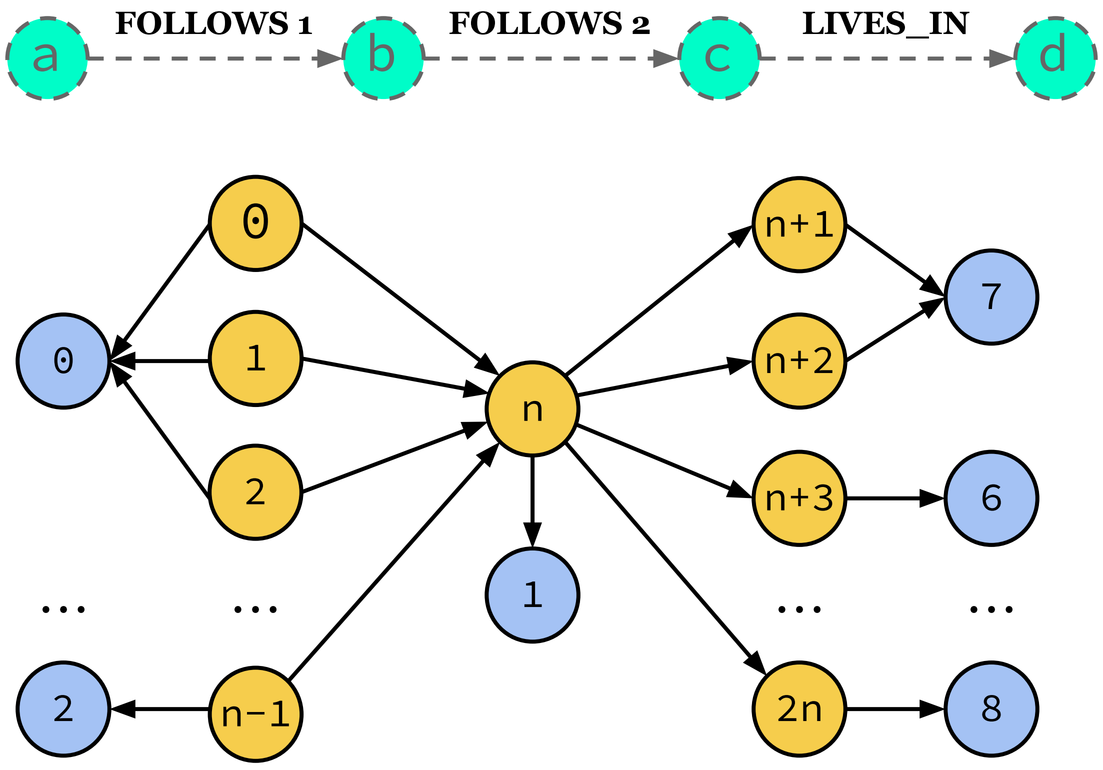
Join a

t $(b{:}\mathbf{v}_b)$

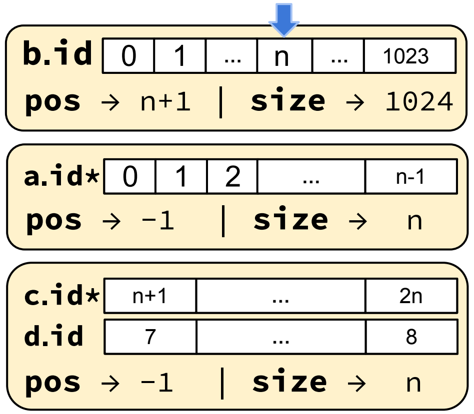
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges



b.id	0	1	…	n	…	1023

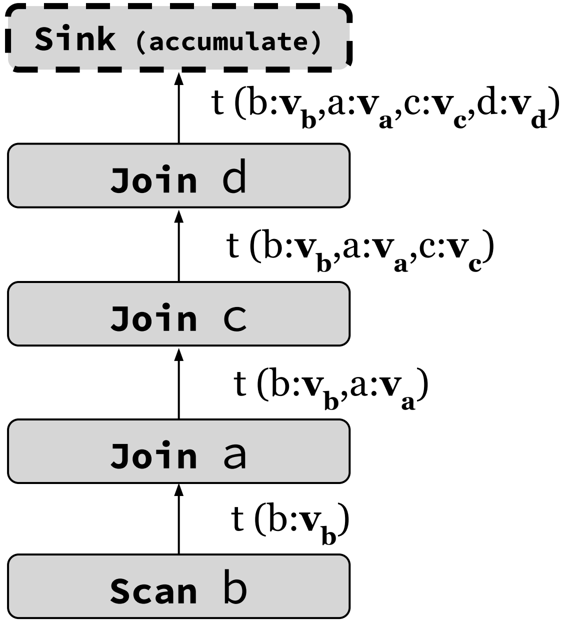
pos → n+1 | **size** → 1024

a.id*	0	1	2	…	n-1

pos → -1 | **size** → n

c.id*	n+1	…	2n
d.id	7	…	8

pos → -1 | **size** → n



Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

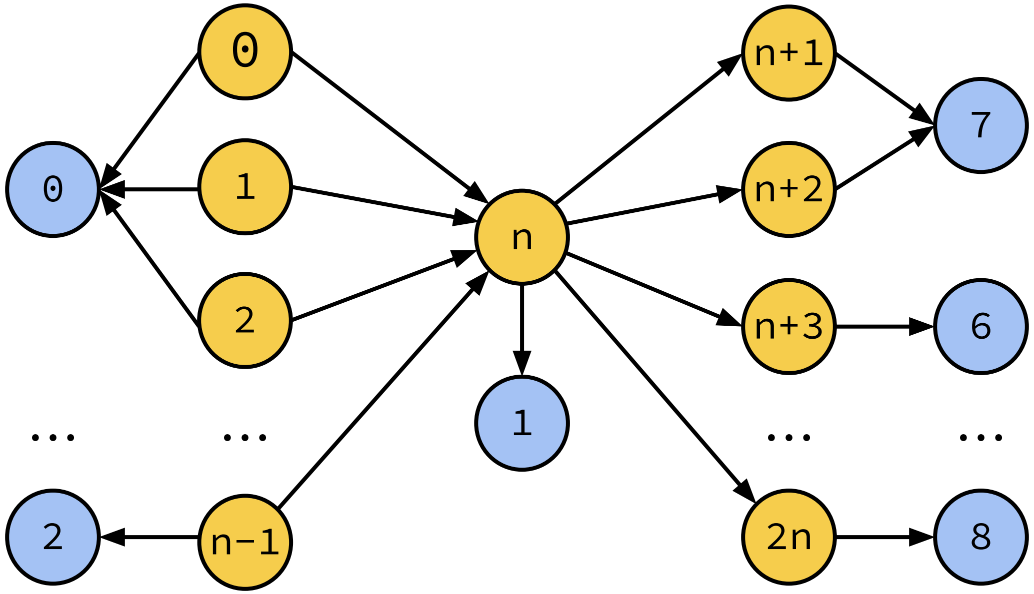
Join a

t (b:$\mathbf{v}_b$)

Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id	0	1	…	n	…	1023

pos → n+1 | **size** → 1024

a.id*	0	1	2	…	n-1

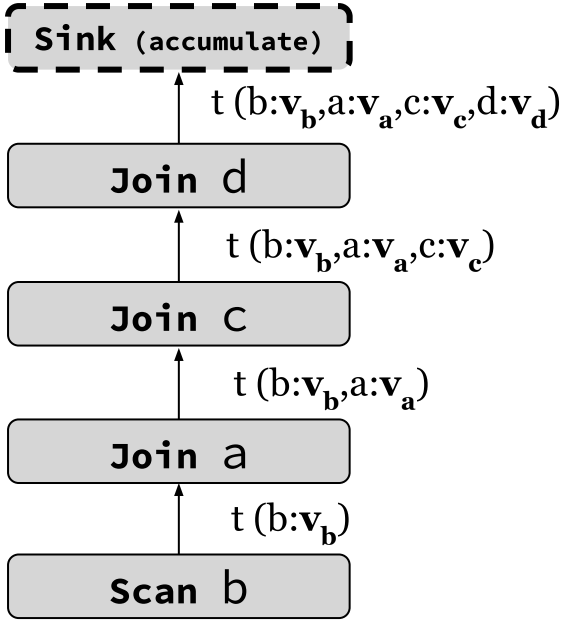
pos → -1 | **size** → n

c.id*	n+1	…	2n
d.id	7	…	8

pos → -1 | **size** → n

Sink output buffer

fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

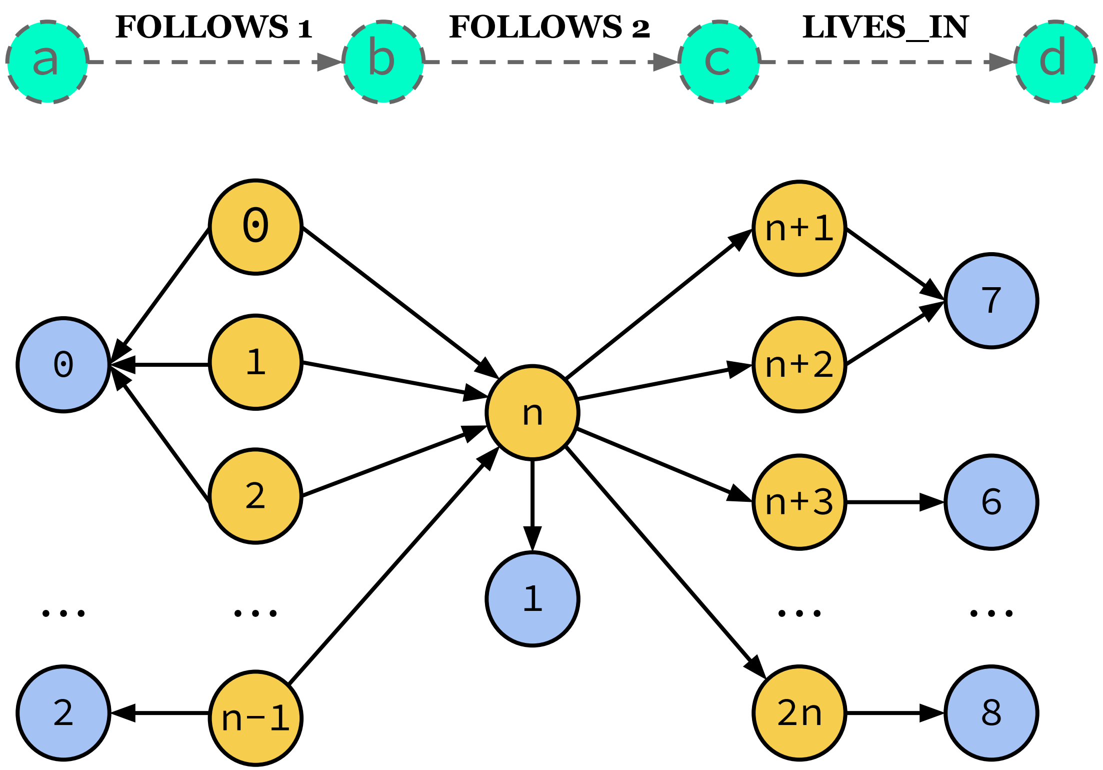
Join a

$t\ (b{:}\mathbf{v}_b)$

Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

pos → n+1 | **size** → 1024

a.id*	0	1	2	...	n-1

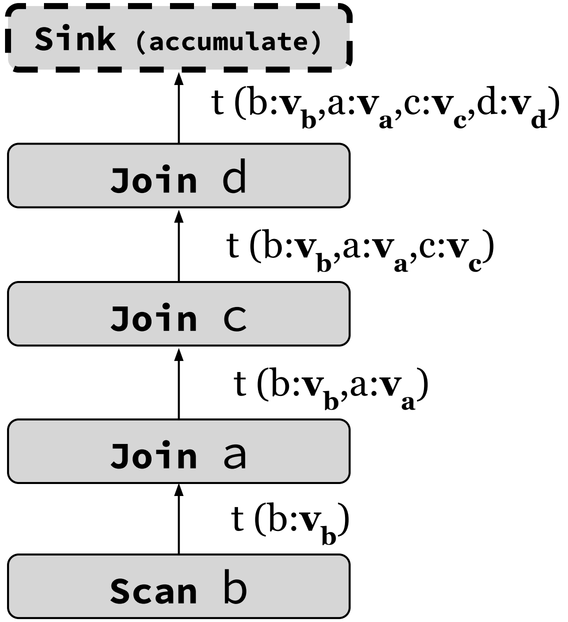
pos → -1 | **size** → n

c.id*	n+1	...	2n
d.id	7	...	8

pos → -1 | **size** → n

Sink output buffer

fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

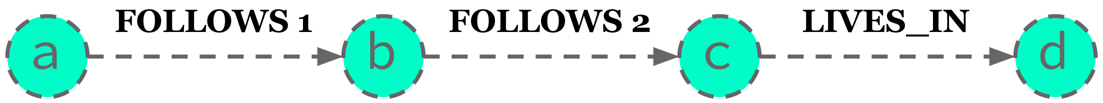
Join a

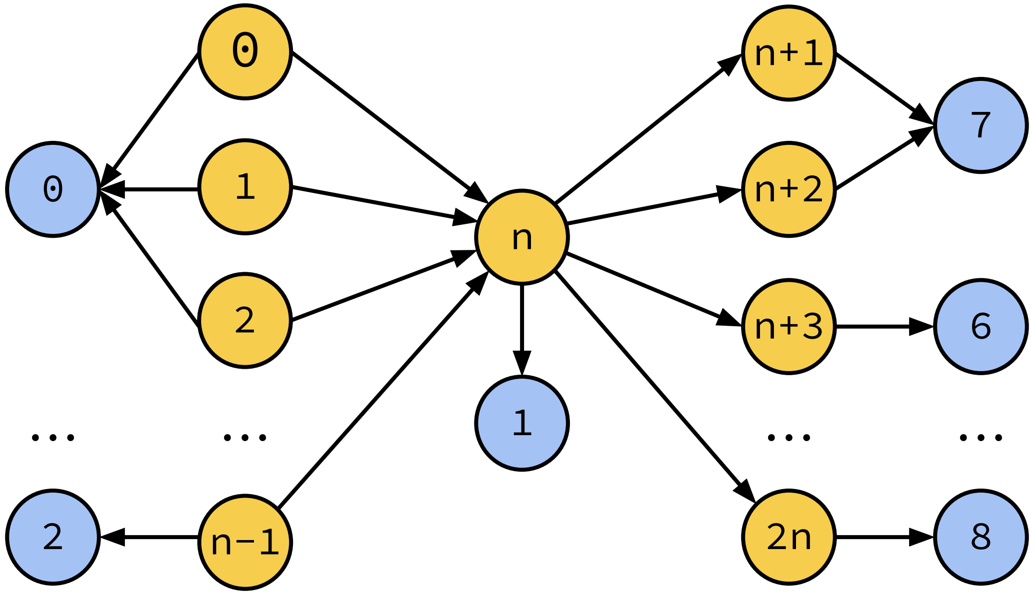
t $(b{:}\mathbf{v}_b)$

Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]





FOLLOWS & LIVES_IN Edges

b.id	0	1	...	n	...	1023

pos → n+1 | **size** → 1024

a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

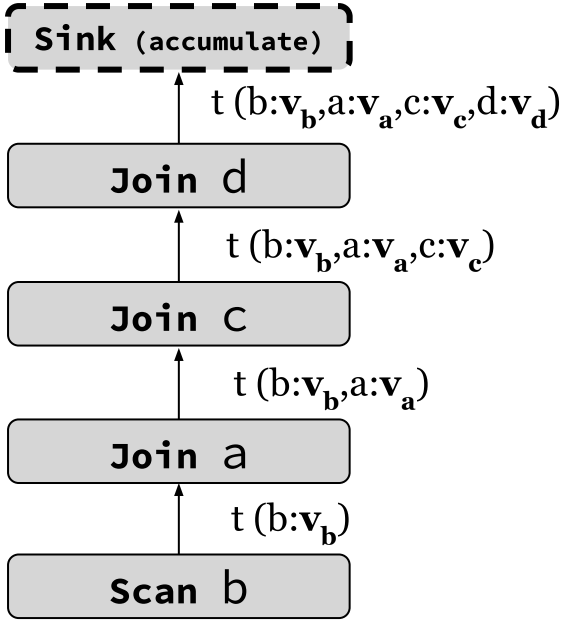
c.id*	n+1	...	2n
d.id	7	...	8

pos → -1 | **size** → n

Sink output buffer

fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

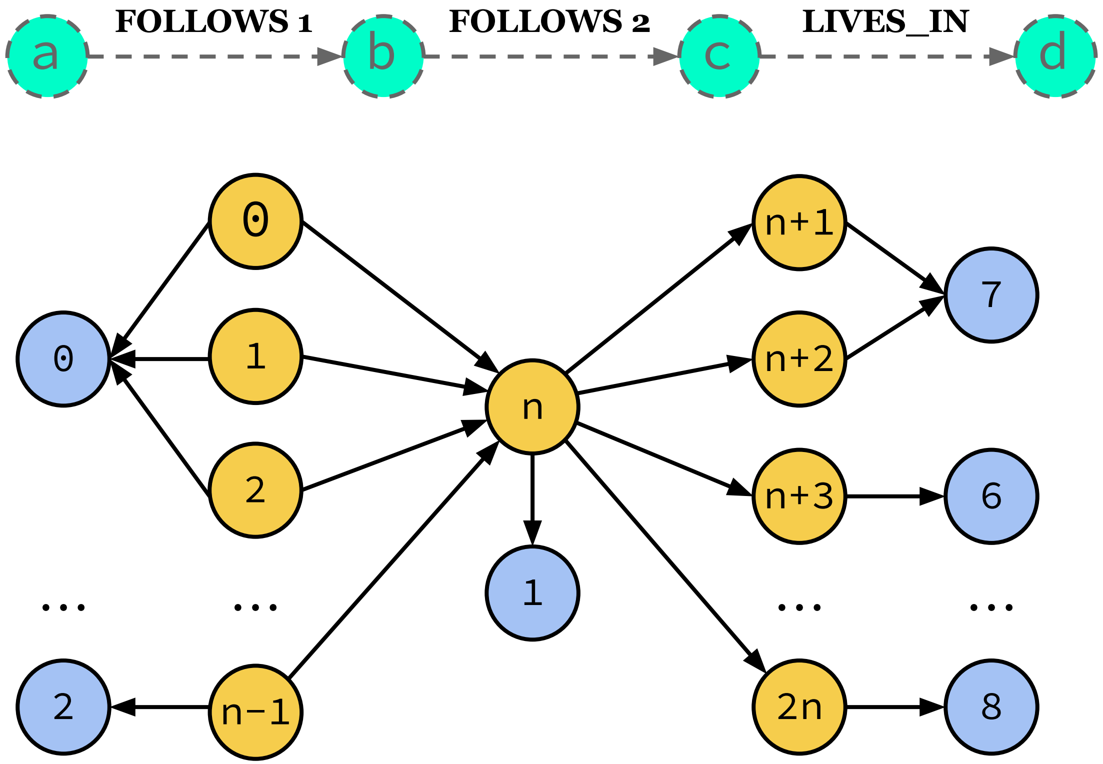
Join a

$t\ (b{:}\mathbf{v}_b)$

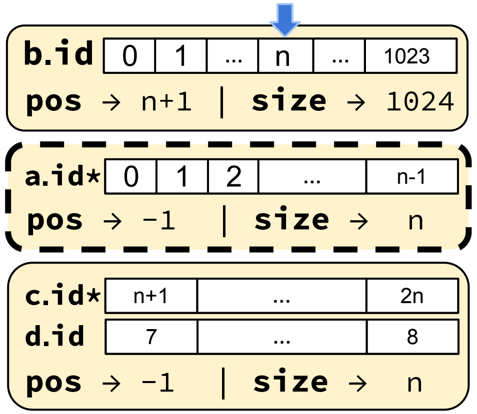
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

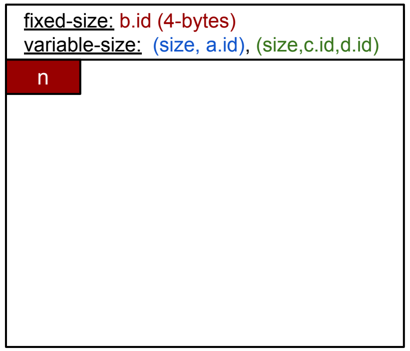


b.id | 0 | 1 | … | n | … | 1023
pos → n+1 | size → 1024

a.id* | 0 | 1 | 2 | … | n-1
pos → -1 | size → n

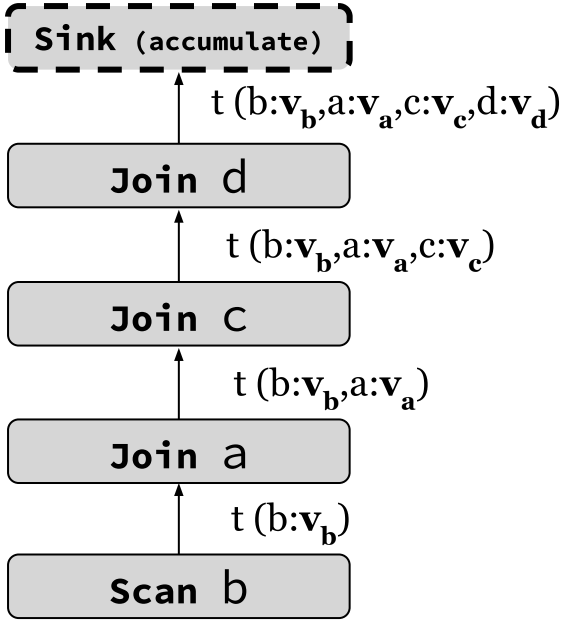
c.id* | n+1 | … | 2n
d.id | 7 | … | 8
pos → -1 | size → n

Sink output buffer



fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

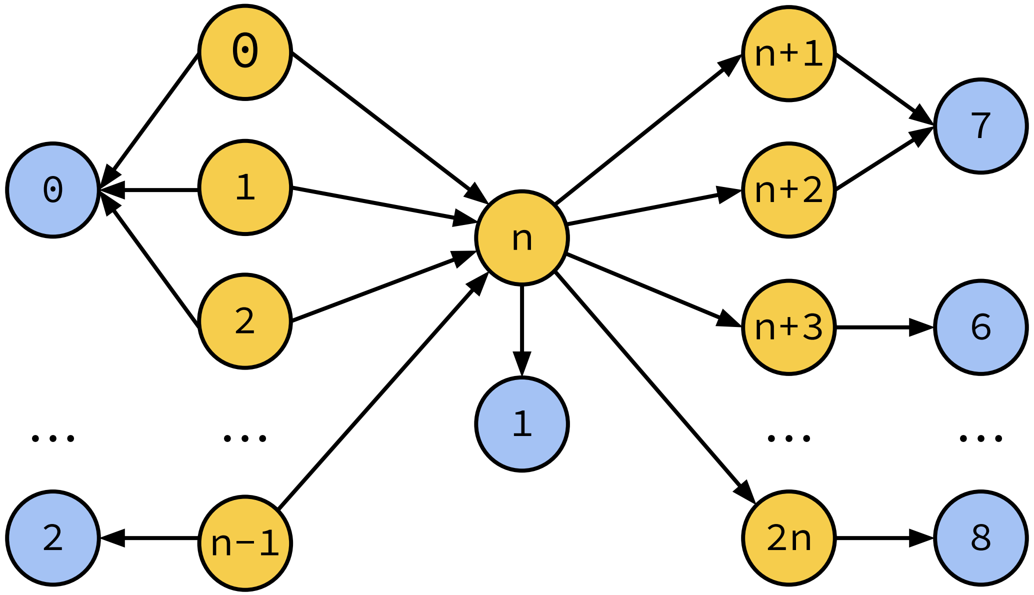
Join a

$t\ (b{:}\mathbf{v}_b)$

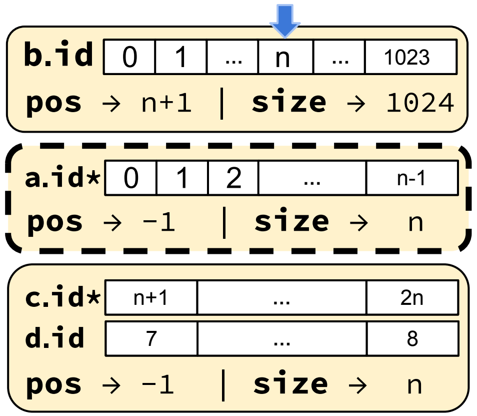
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges



b.id 0 | 1 | ... | n | ... | 1023

pos → n+1 | **size** → 1024

a.id* 0 | 1 | 2 | ... | n-1

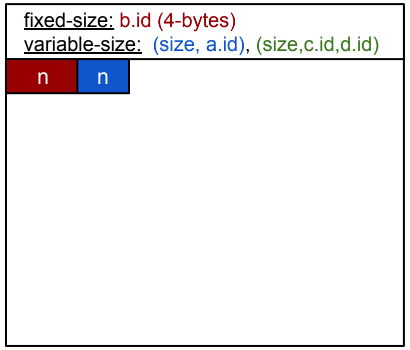
pos → -1 | **size** → n

c.id* n+1 | ... | 2n

d.id 7 | ... | 8

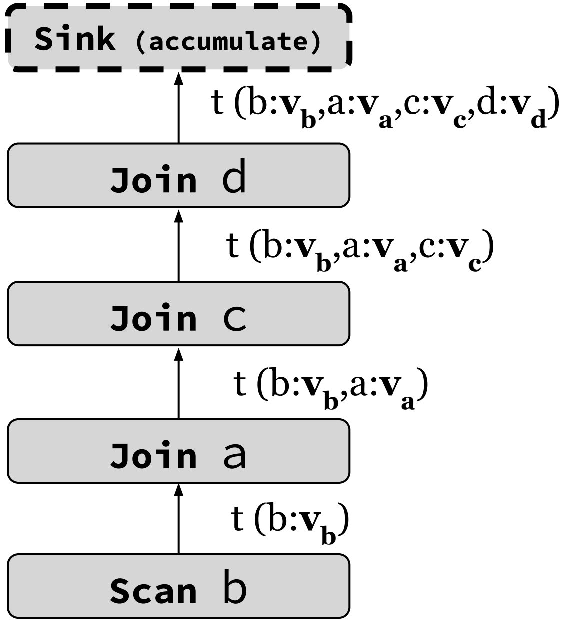
pos → -1 | **size** → n

Sink output buffer



fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n	n



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

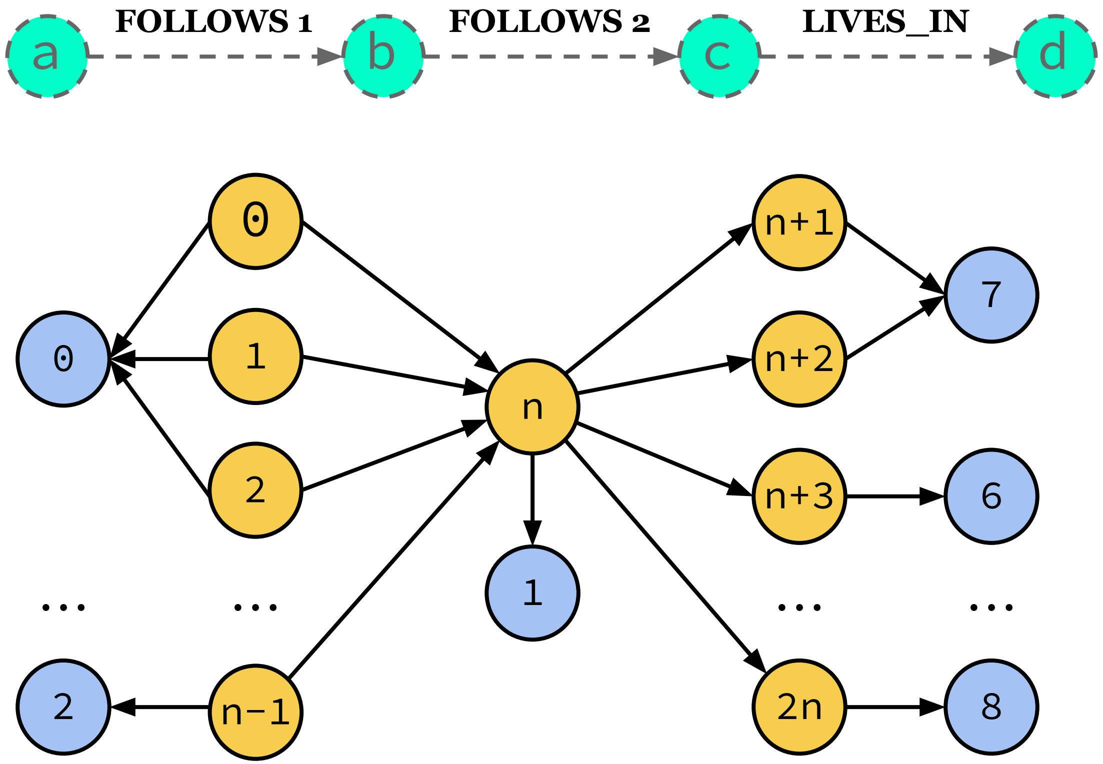
Join a

$t\ (b{:}\mathbf{v}_b)$

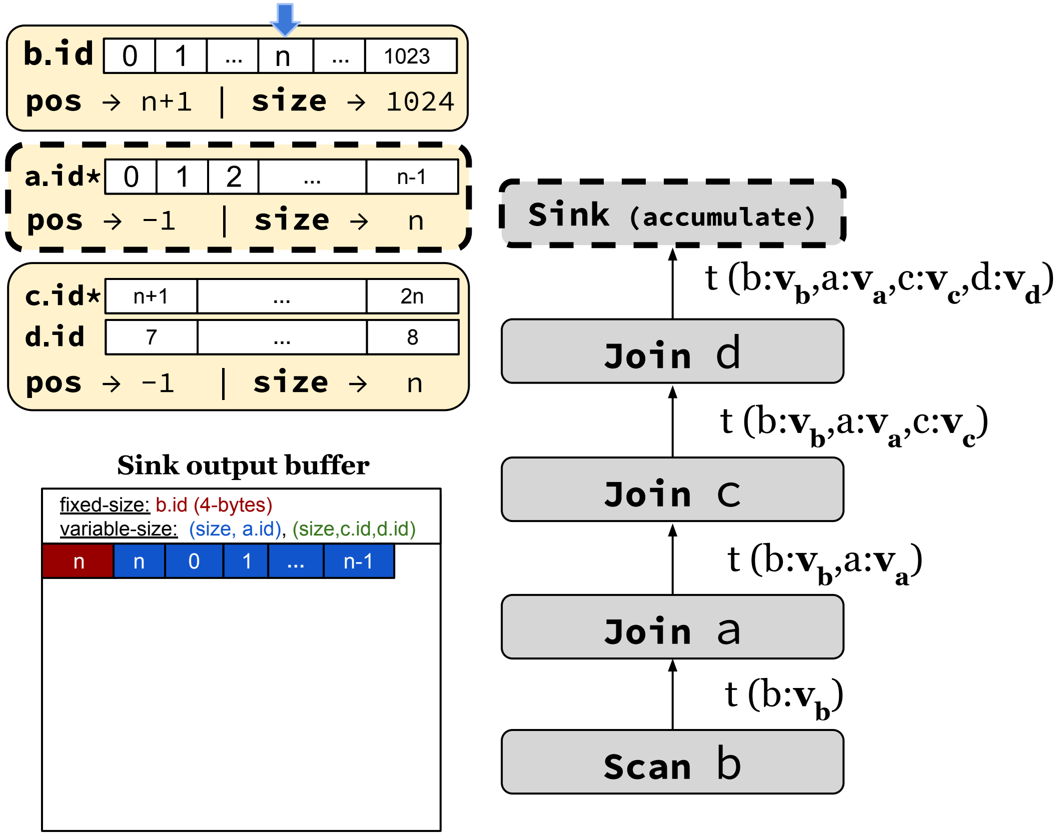
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges



b.id	0	1	...	n	...	1023

pos → n+1 | **size** → 1024

a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

c.id*	n+1	...	2n
d.id	7	...	8

pos → -1 | **size** → n

Sink output buffer

fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n	n	0	1	...	n-1

Sink (accumulate)

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$,d:$\mathbf{v}_d$)

Join d

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$,c:$\mathbf{v}_c$)

Join c

t (b:$\mathbf{v}_b$,a:$\mathbf{v}_a$)

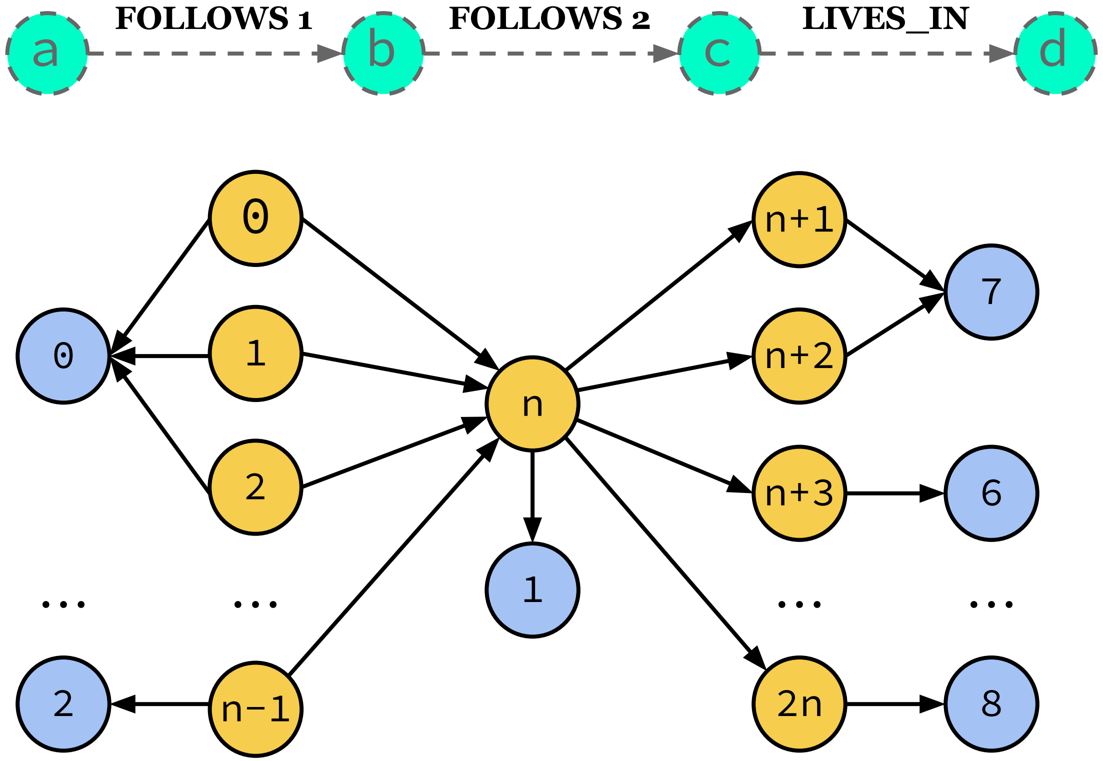
Join a

t (b:$\mathbf{v}_b$)

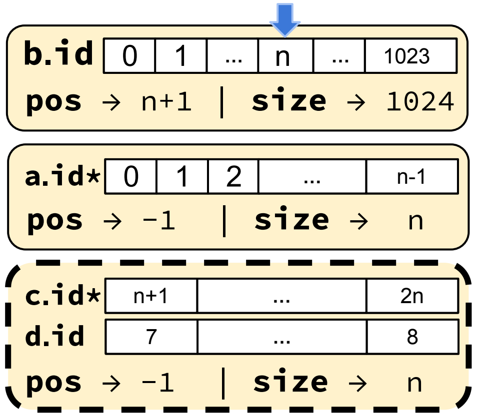
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

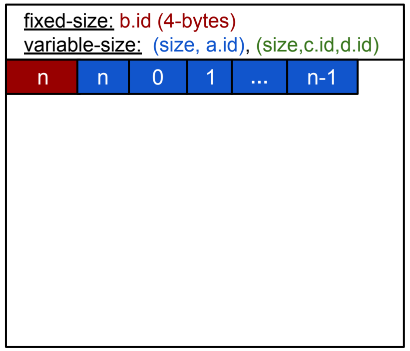


b.id | 0 | 1 | ... | n | ... | 1023
pos → n+1 | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

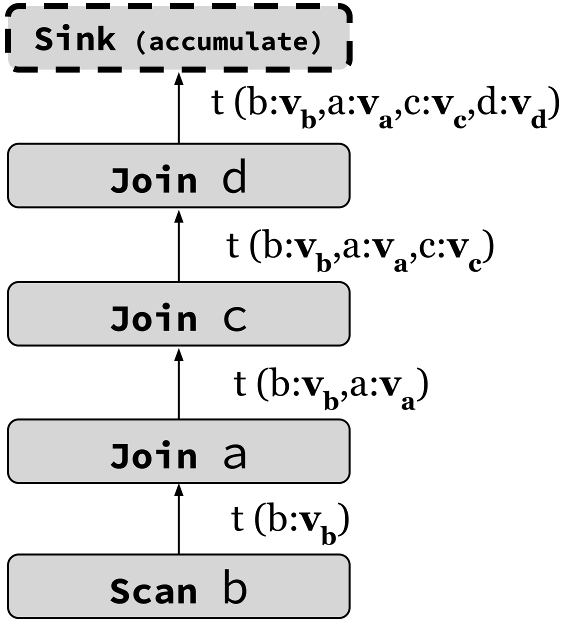
c.id* | n+1 | ... | 2n
d.id | 7 | ... | 8
pos → -1 | **size** → n

Sink output buffer



fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n	n	0	1	...	n-1



Sink (accumulate)

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

t $(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

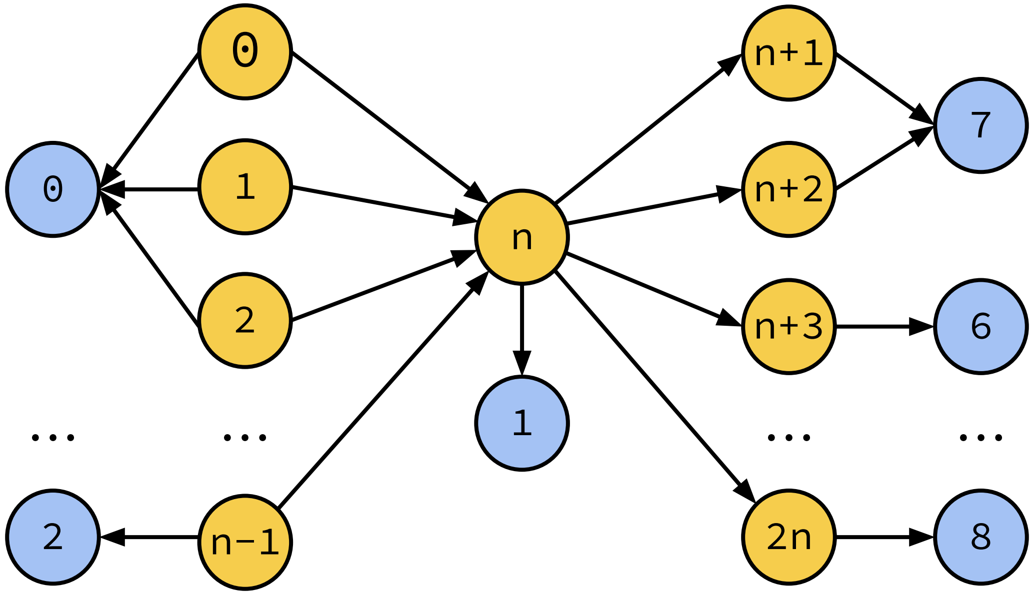
Join a

t $(b{:}\mathbf{v}_b)$

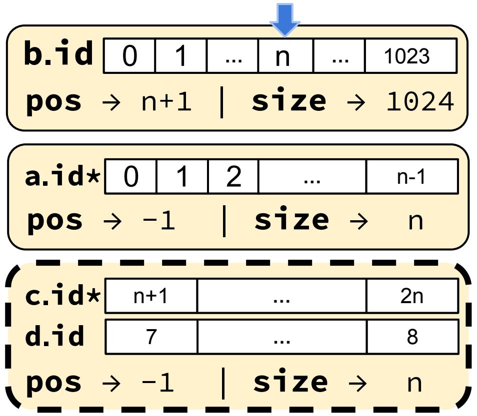
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

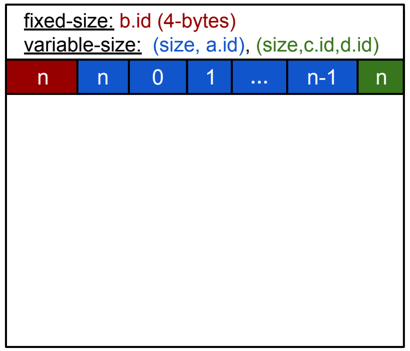


b.id | 0 | 1 | ... | n | ... | 1023
pos → n+1 | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

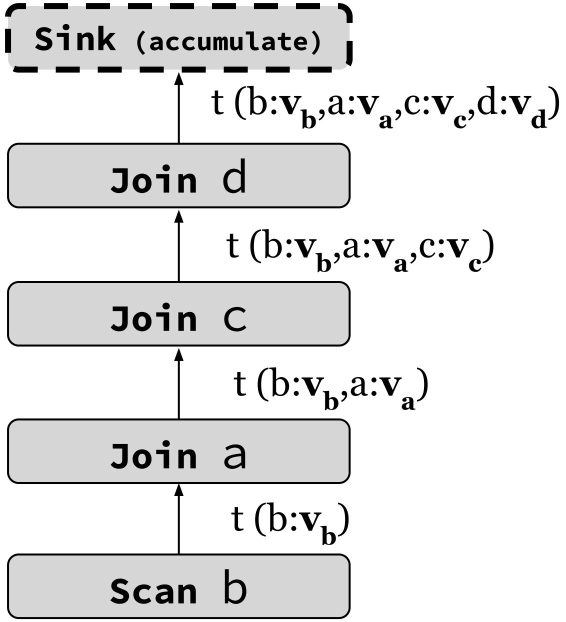
c.id* | n+1 | ... | 2n
d.id | 7 | ... | 8
pos → -1 | **size** → n

Sink output buffer



fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n	n	0	1	...	n-1	n



Sink (accumulate)

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

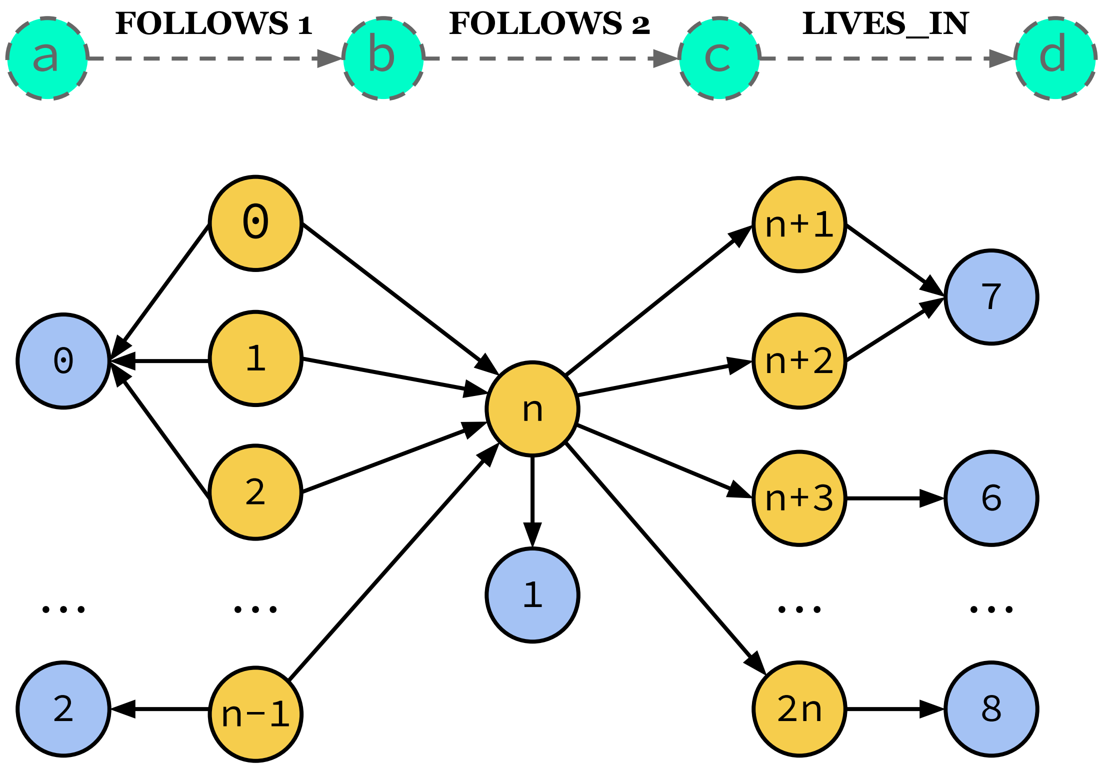
Join a

$t\,(b{:}\mathbf{v}_b)$

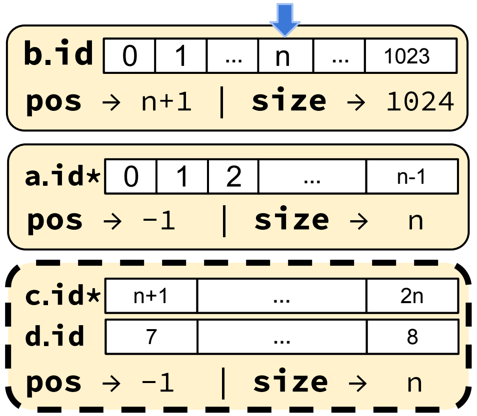
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges



b.id	0	1	...	n	...	1023

pos → n+1 | **size** → 1024

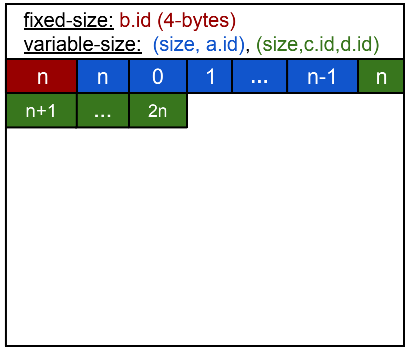
a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

c.id*	n+1	...	2n
d.id	7	...	8

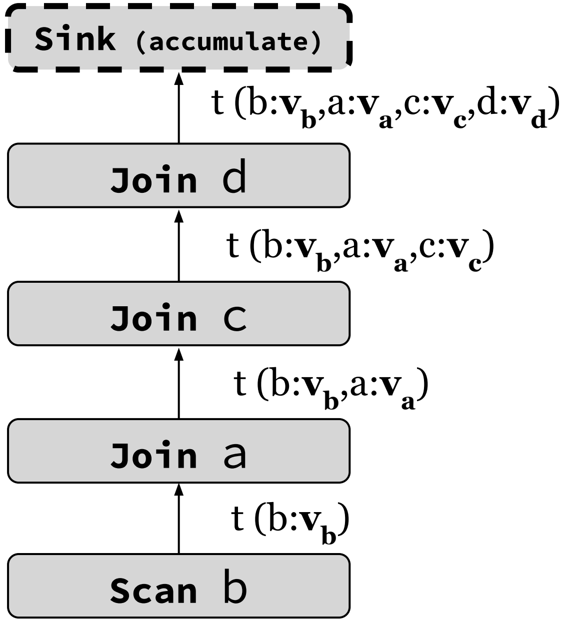
pos → -1 | **size** → n

Sink output buffer



fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n	n	0	1	...	n-1	n
n+1	...	2n				



Sink (accumulate)

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\ (b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

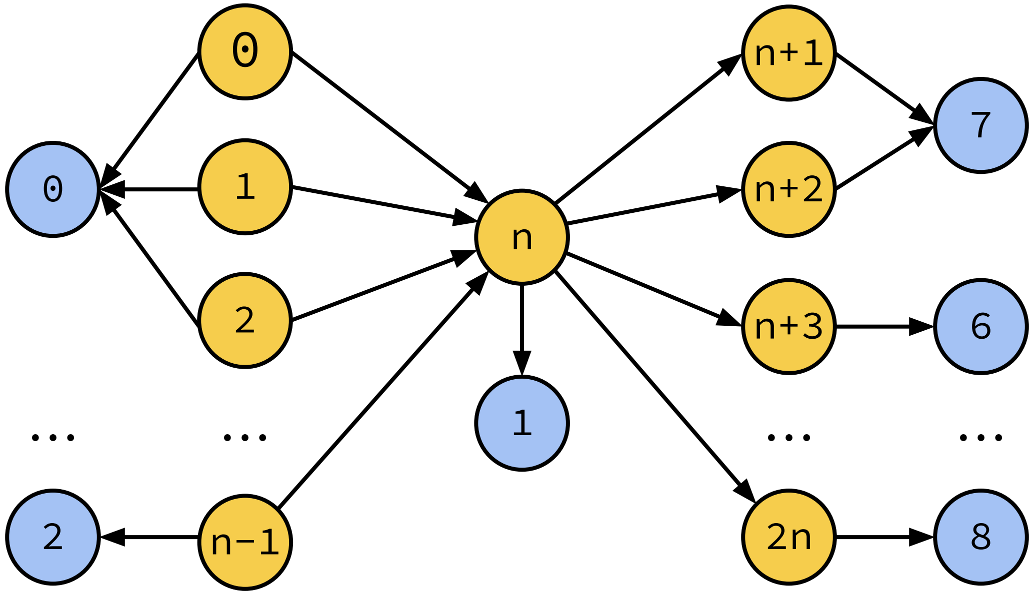
Join a

$t\ (b{:}\mathbf{v}_b)$

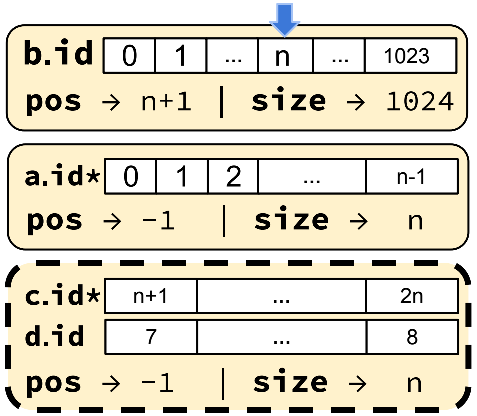
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

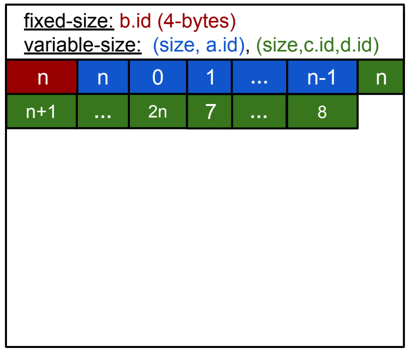


b.id 0 | 1 | ... | n | ... | 1023
pos → n+1 | **size** → 1024

a.id* 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

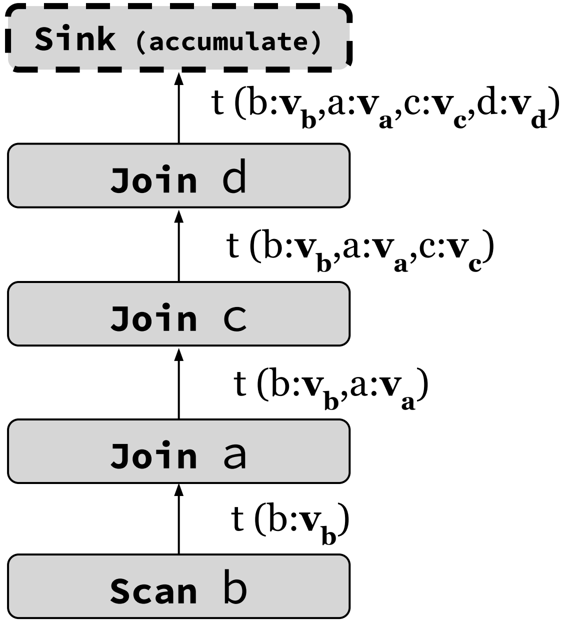
c.id* n+1 | ... | 2n
d.id 7 | ... | 8
pos → -1 | **size** → n

Sink output buffer



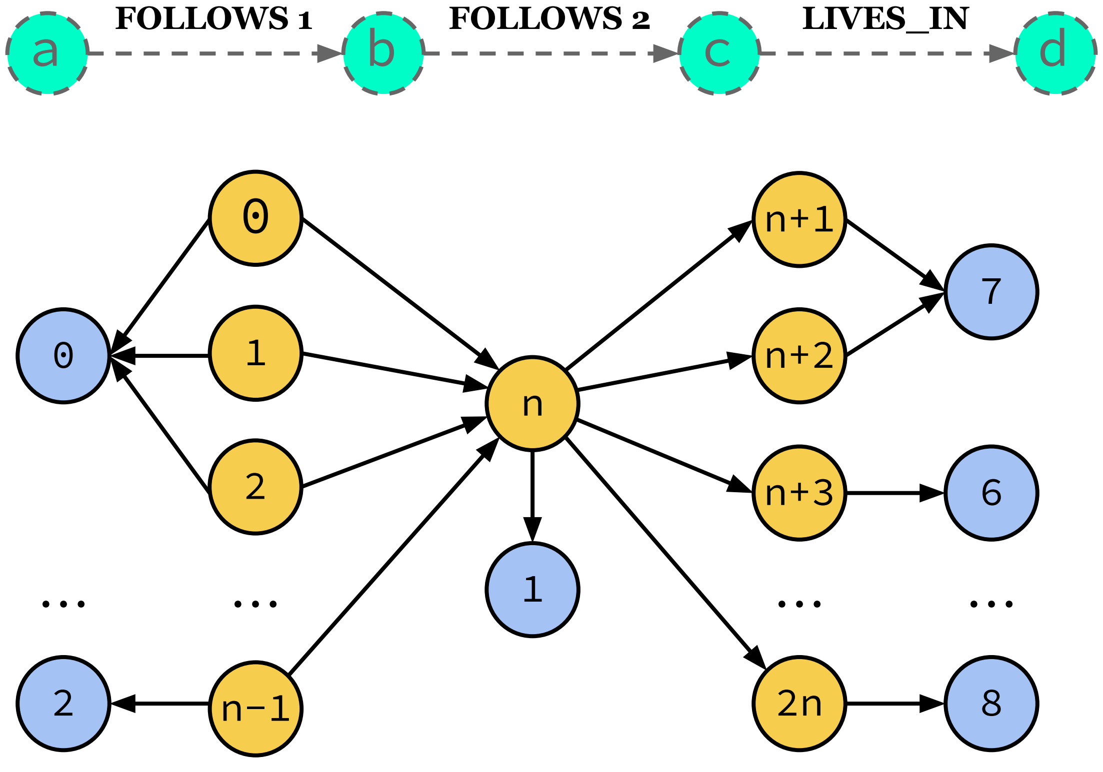
fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n	n	0	1	...	n-1	n
n+1	...	2n	7	...	8	

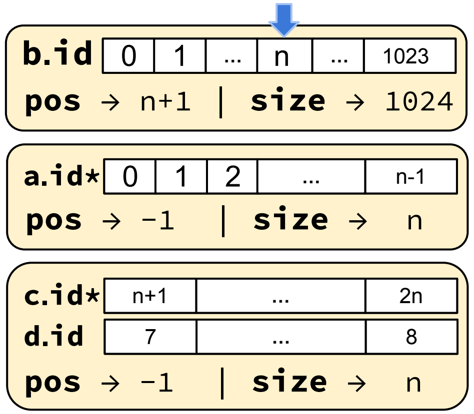


Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

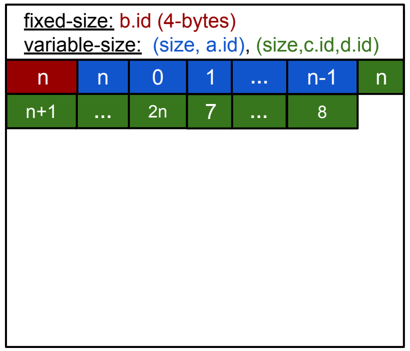


b.id | 0 | 1 | ... | n | ... | 1023
pos → n+1 | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

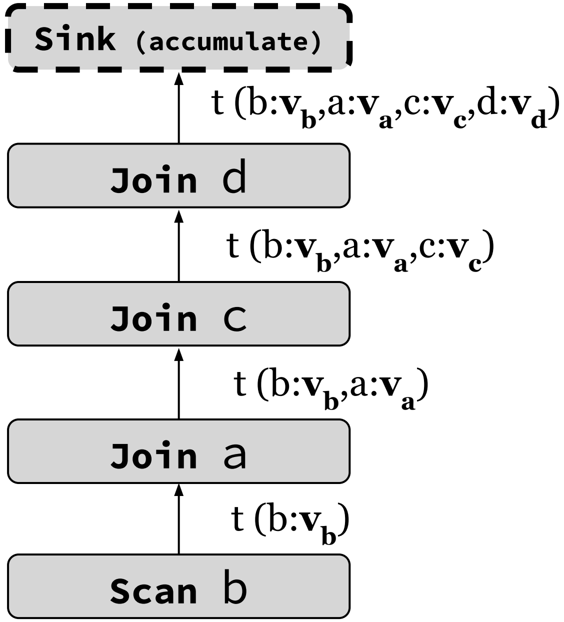
c.id* | n+1 | ... | 2n
d.id | 7 | ... | 8
pos → -1 | **size** → n

Sink output buffer



fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

n	n	0	1	...	n-1	n
n+1	...	2n	7	...	8	



Sink (accumulate)

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c, d{:}\mathbf{v}_d)$

Join d

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a, c{:}\mathbf{v}_c)$

Join c

$t\,(b{:}\mathbf{v}_b, a{:}\mathbf{v}_a)$

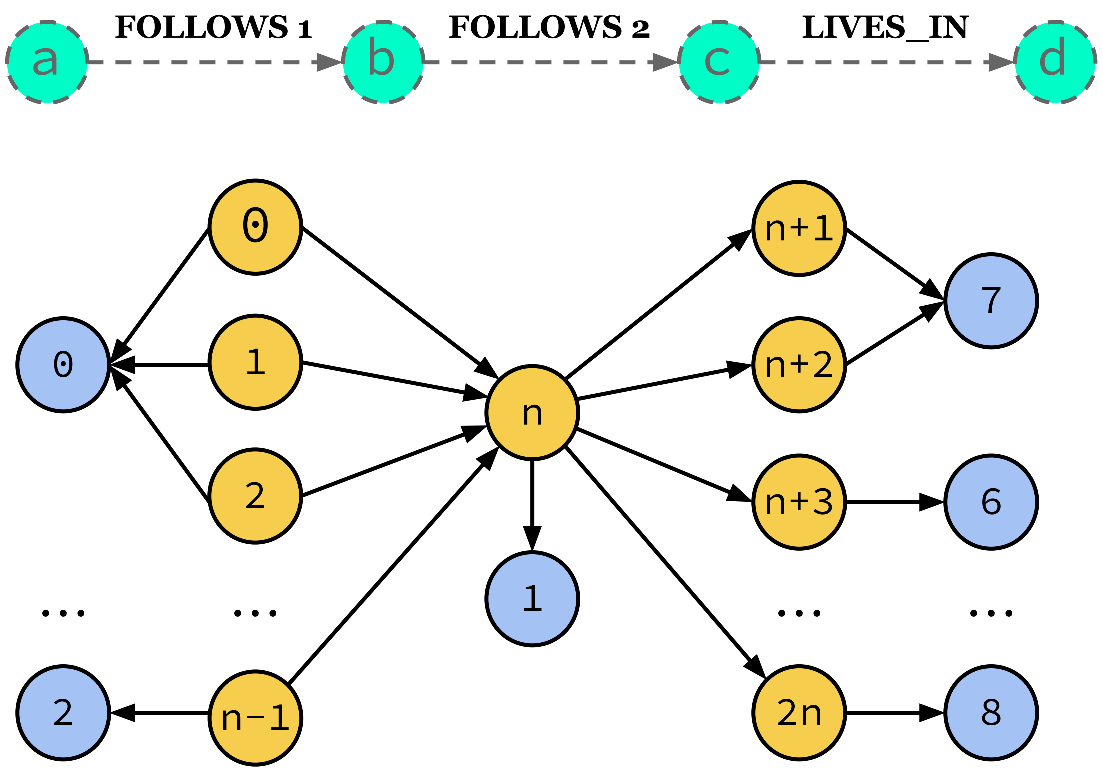
Join a

$t\,(b{:}\mathbf{v}_b)$

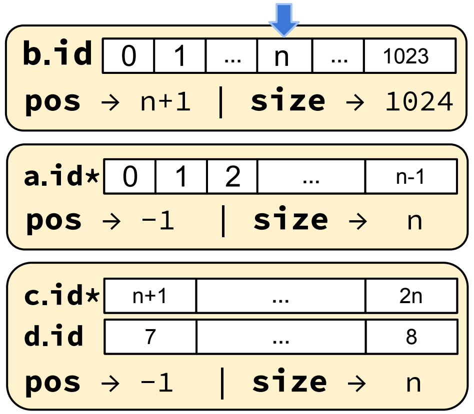
Scan b

Factorized Vector Execution

Query Vertex Ordering:
[b, a, c, d]



FOLLOWS & LIVES_IN Edges

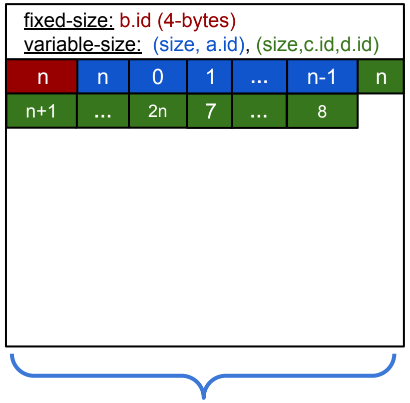


b.id | 0 | 1 | ... | n | ... | 1023
pos → n+1 | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1
pos → -1 | **size** → n

c.id* | n+1 | ... | 2n
d.id | 7 | ... | 8
pos → -1 | **size** → n

Sink output buffer

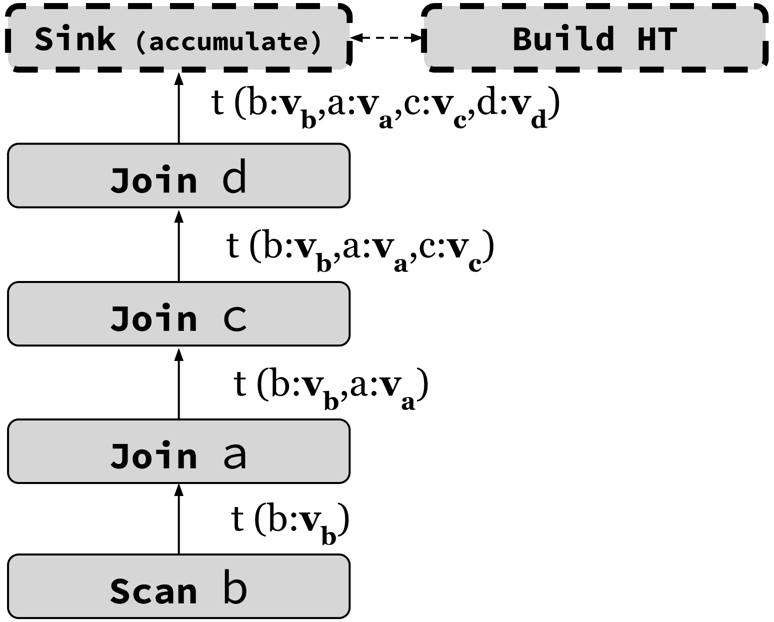


fixed-size: b.id (4-bytes)
variable-size: (size, a.id), (size,c.id,d.id)

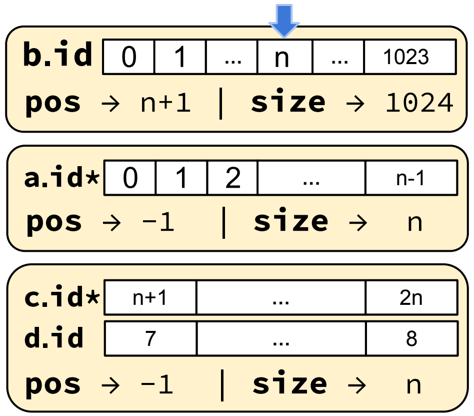
n	n	0	1	...	n-1	n
n+1	...	2n	7	...	8	

3n+1 fields
vs. flat
$4n^2$ fields

Same for BHT
(variable-sizes in
overflow pages)
→ smaller HT



Factorized Vector Benefits & Limits



b.id	0	1	...	n	...	1023

pos → n+1 | **size** → 1024

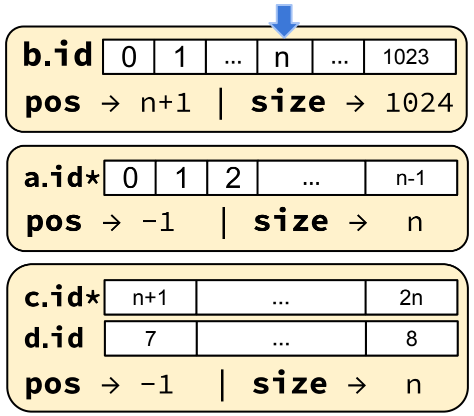
a.id*	0	1	2	...	n-1

pos → -1 | **size** → n

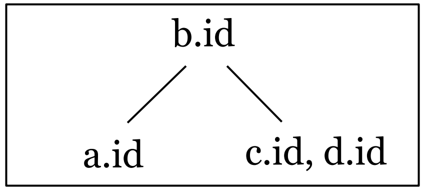
c.id*	n+1	...	2n
d.id	7	...	8

pos → -1 | **size** → n

Factorized Vector Benefits & Limits

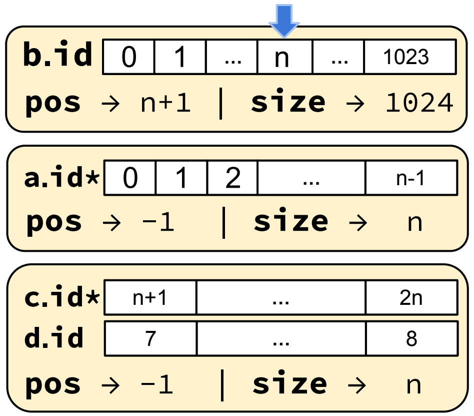


Pipelined factorization with f-tree
saving: reading column scans & Filtering



restricted f-trees (height = 1) to allow pipelining.
Other redundancy is possible

Factorized Vector Benefits & Limits



b.id | 0 | 1 | ... | n | ... | 1023 |

pos → n+1 | **size** → 1024

a.id* | 0 | 1 | 2 | ... | n-1 |

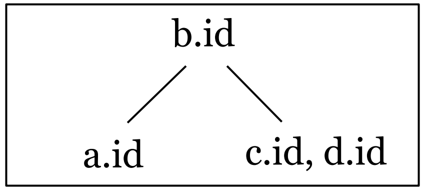
pos → -1 | **size** → n

c.id* | n+1 | ... | 2n |

d.id | 7 | ... | 8 |

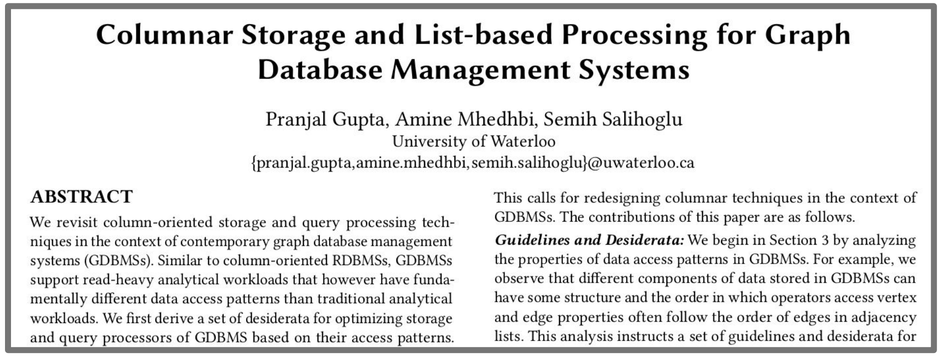
pos → -1 | **size** → n

Pipelined factorization with f-tree saving: reading column scans & Filtering



restricted f-trees (height = 1) to allow pipelining. Other redundancy is possible

Performance speedups up to 19x on LDBC 100



Columnar Storage and List-based Processing for Graph Database Management Systems

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu
University of Waterloo
{pranjal.gupta,amine.mhedhbi,semih.salihoglu}@uwaterloo.ca

ABSTRACT

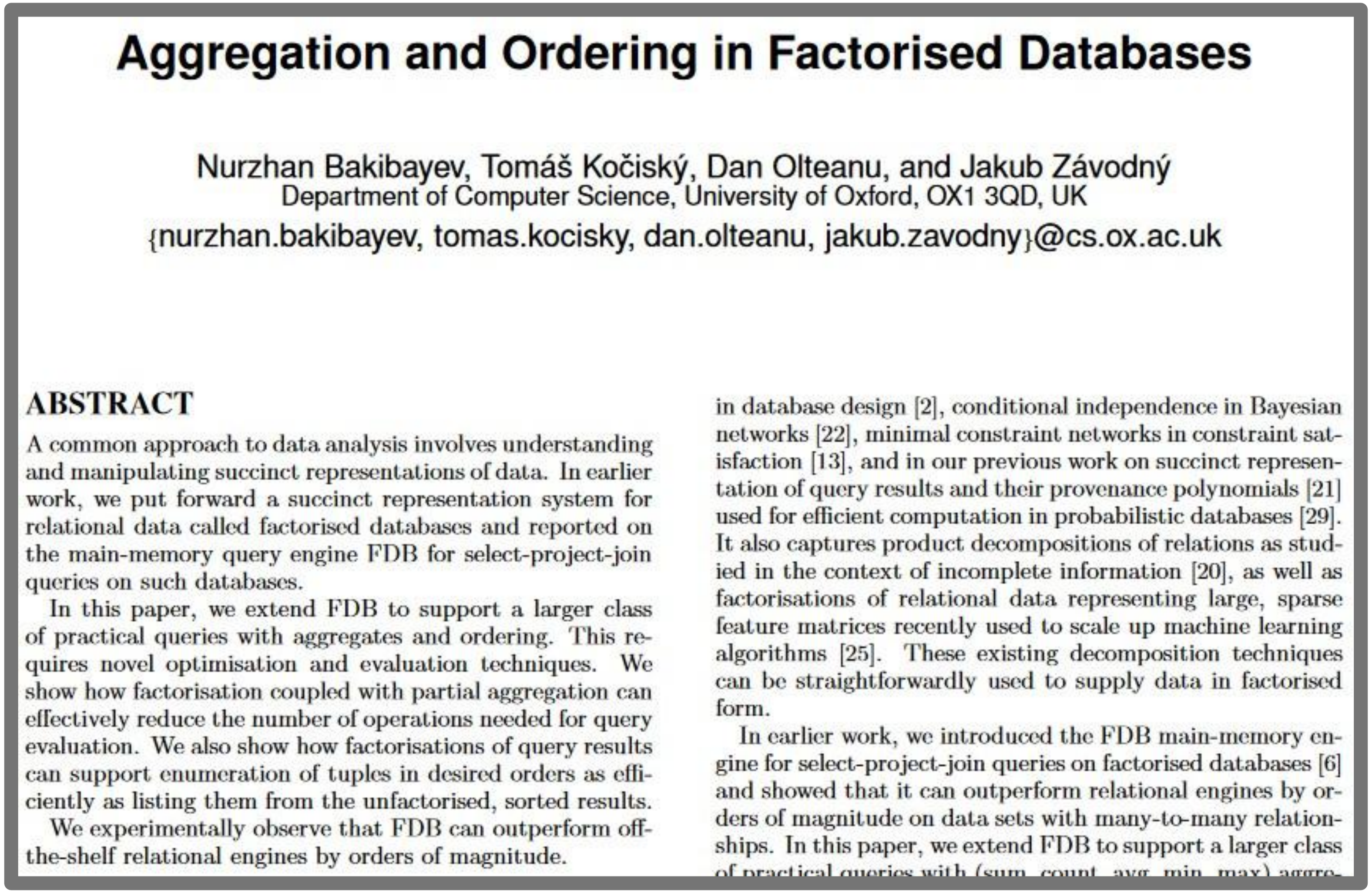
We revisit column-oriented storage and query processing techniques in the context of contemporary graph database management systems (GDBMSs). Similar to column-oriented RDBMSs, GDBMSs support read-heavy analytical workloads that however have fundamentally different data access patterns than traditional analytical workloads. We first derive a set of desiderata for optimizing storage and query processors of GDBMS based on their access patterns.

This calls for redesigning columnar techniques in the context of GDBMSs. The contributions of this paper are as follows.

Guidelines and Desiderata: We begin in Section 3 by analyzing the properties of data access patterns in GDBMSs. For example, we observe that different components of data stored in GDBMSs can have some structure and the order in which operators access vertex and edge properties often follow the order of edges in adjacency lists. This analysis instructs a set of guidelines and desiderata for

Other Work

Other Work



Aggregation and Ordering in Factorised Databases

Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný
Department of Computer Science, University of Oxford, OX1 3QD, UK
{nurzhan.bakibayev, tomas.kocisky, dan.olteanu, jakub.zavodny}@cs.ox.ac.uk

ABSTRACT

A common approach to data analysis involves understanding and manipulating succinct representations of data. In earlier work, we put forward a succinct representation system for relational data called factorised databases and reported on the main-memory query engine FDB for select-project-join queries on such databases.

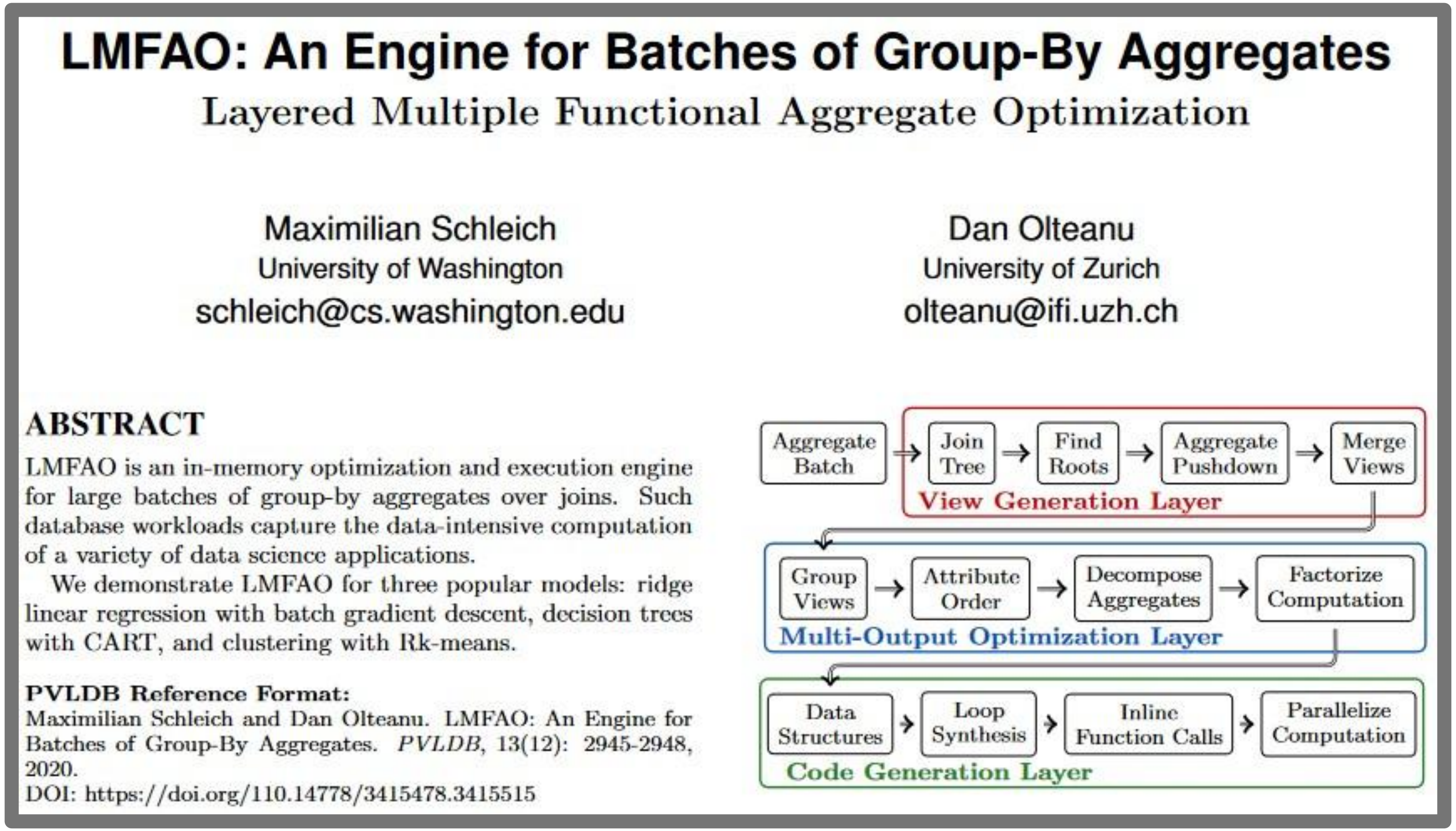
In this paper, we extend FDB to support a larger class of practical queries with aggregates and ordering. This requires novel optimisation and evaluation techniques. We show how factorisation coupled with partial aggregation can effectively reduce the number of operations needed for query evaluation. We also show how factorisations of query results can support enumeration of tuples in desired orders as efficiently as listing them from the unfactorised, sorted results.

We experimentally observe that FDB can outperform off-the-shelf relational engines by orders of magnitude.

in database design [2], conditional independence in Bayesian networks [22], minimal constraint networks in constraint satisfaction [13], and in our previous work on succinct representation of query results and their provenance polynomials [21] used for efficient computation in probabilistic databases [29]. It also captures product decompositions of relations as studied in the context of incomplete information [20], as well as factorisations of relational data representing large, sparse feature matrices recently used to scale up machine learning algorithms [25]. These existing decomposition techniques can be straightforwardly used to supply data in factorised form.

In earlier work, we introduced the FDB main-memory engine for select-project-join queries on factorised databases [6] and showed that it can outperform relational engines by orders of magnitude on data sets with many-to-many relationships. In this paper, we extend FDB to support a larger class of practical queries with (sum, count, avg, min, max) aggre-

Other Work



LMFAO: An Engine for Batches of Group-By Aggregates

Layered Multiple Functional Aggregate Optimization

Maximilian Schleich
University of Washington
schleich@cs.washington.edu

Dan Olteanu
University of Zurich
olteanu@ifi.uzh.ch

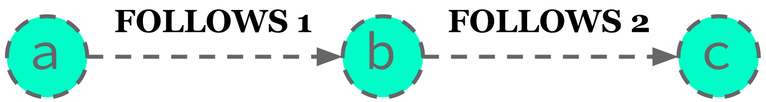
ABSTRACT

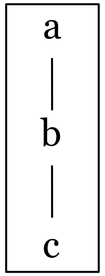
LMFAO is an in-memory optimization and execution engine for large batches of group-by aggregates over joins. Such database workloads capture the data-intensive computation of a variety of data science applications.

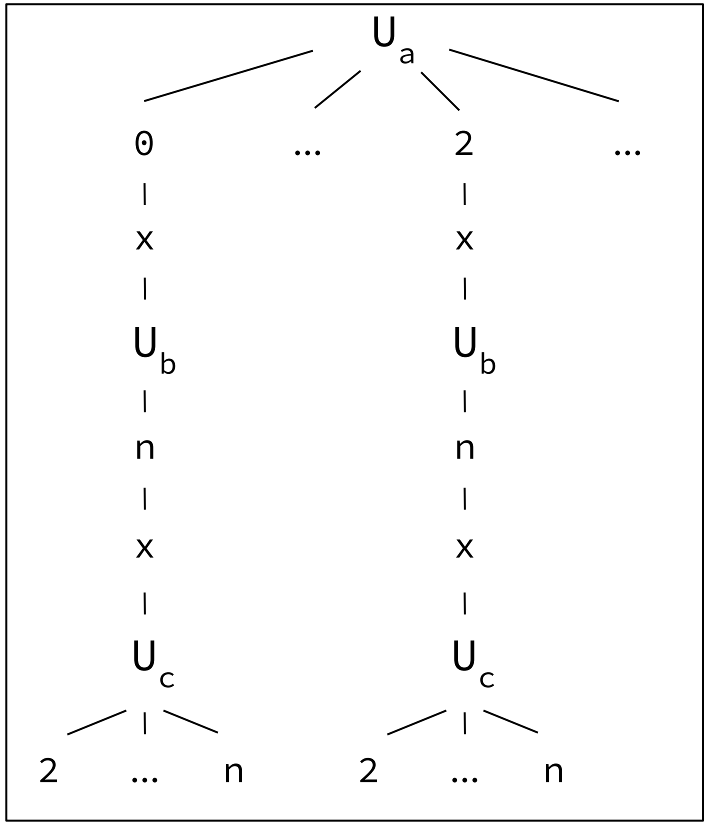
We demonstrate LMFAO for three popular models: ridge linear regression with batch gradient descent, decision trees with CART, and clustering with Rk-means.

PVLDB Reference Format:
Maximilian Schleich and Dan Olteanu. LMFAO: An Engine for Batches of Group-By Aggregates. *PVLDB*, 13(12): 2945-2948, 2020.
DOI: https://doi.org/110.14778/3415478.3415515

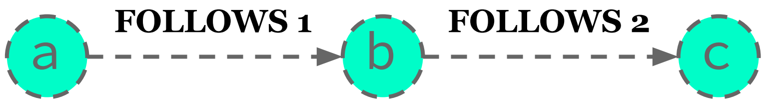
Redundancy in F-Representations Example

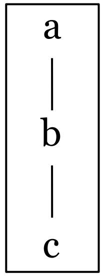


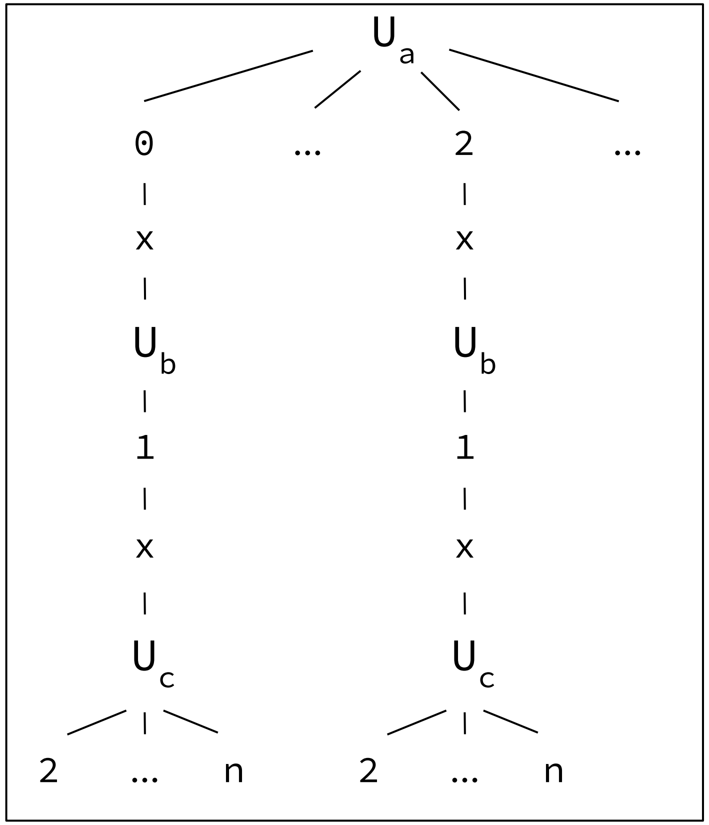




Redundancy in F-Representations Example

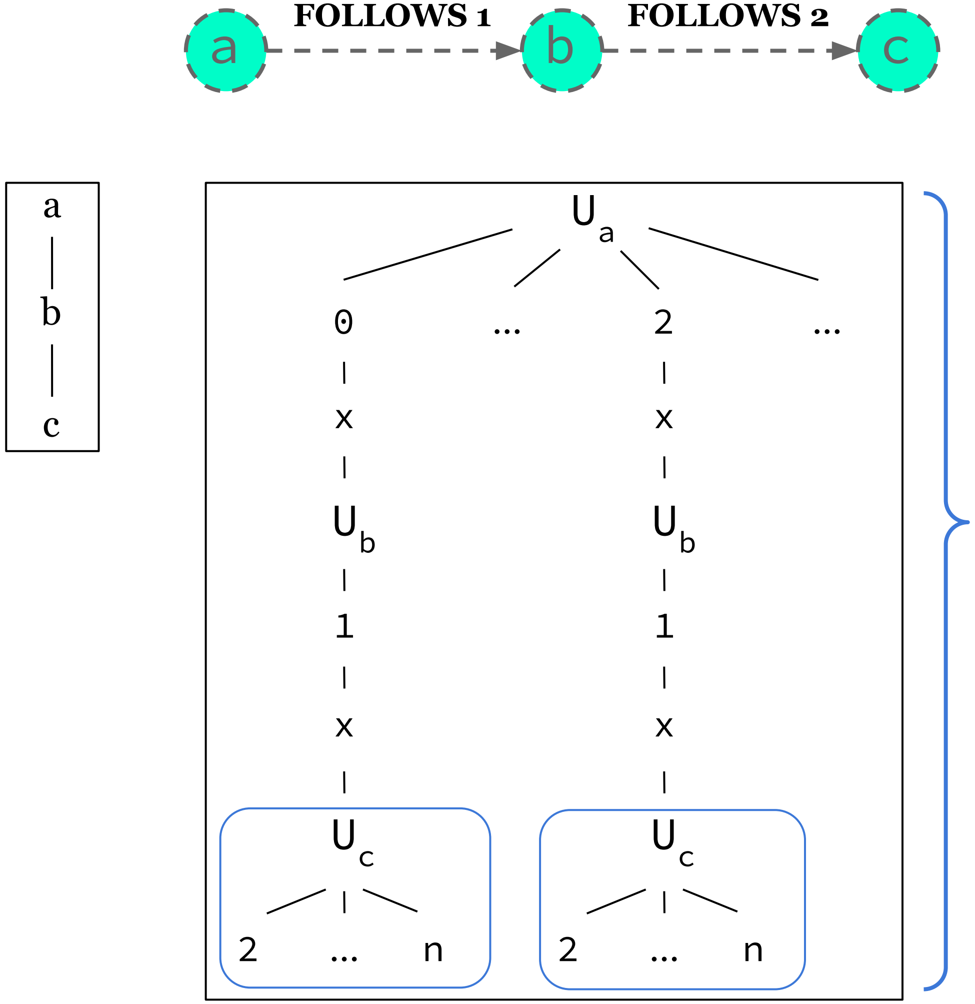






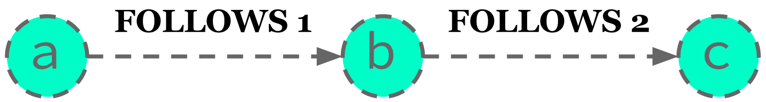
$2n + n^2$ fields

Redundancy in F-Representations Example

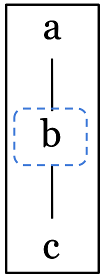


$2n + n^2$ fields

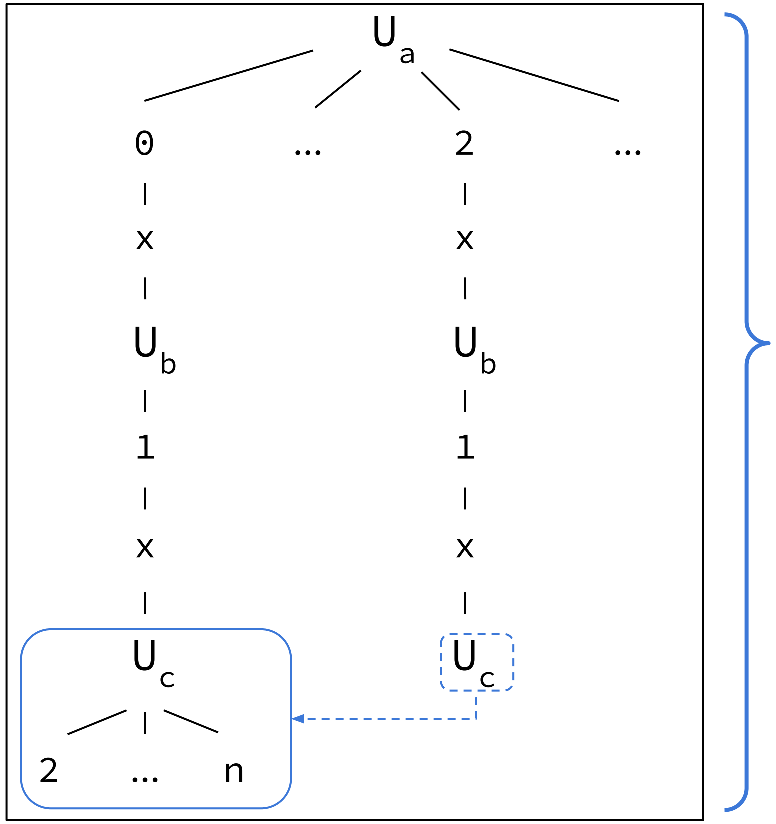
Redundancy in F-Representations Example



d-tree

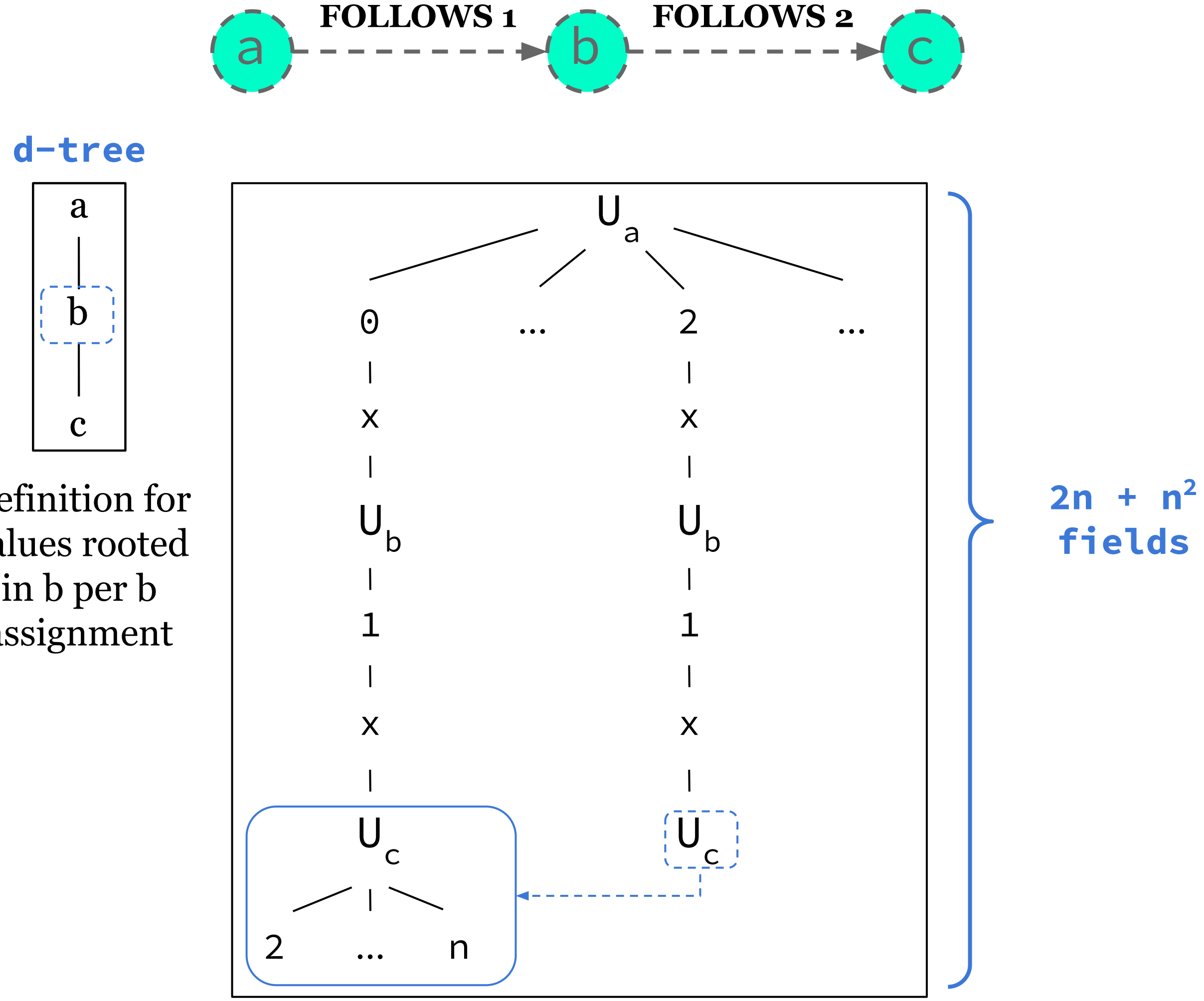


Definition for values rooted in b per b assignment



$2n + n^2$ fields

F-Representations /W Definitions Example



Definition for values rooted in b per b assignment

Theory of factorization establishes:

$$\sigma^{\uparrow}(Q) \leq \sigma(Q) \leq \text{AGM}(Q), \text{ where}$$

$\sigma(Q)$: worst-case size bound over f-representations.
$\sigma^{\uparrow}(Q)$: worst-case size bound over d-representations.

In some cases, $\sigma^{\uparrow}(Q) < \sigma(Q) < \text{AGM}(Q)$

Open Challenges

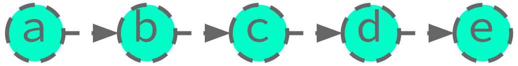
1) How to integrate the use of d-representations in query plans?

2) How to generate and cost such plans?

3) How to characterize the benefits of factorized representations in terms of query and dataset characteristics?

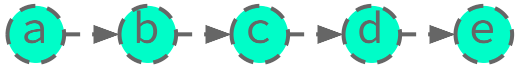
D-Representation for Expression Reuse Example

D-Representation for Expression Reuse Example



WHERE P(a, b, c, d, e)
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

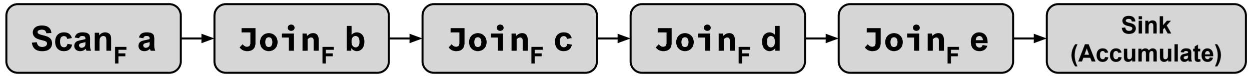
D-Representation for Expression Reuse Example



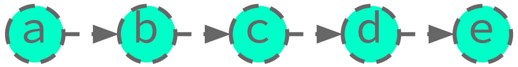
Query Vertex Ordering:
[a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

Op_F c Scan necessary columns and apply filters



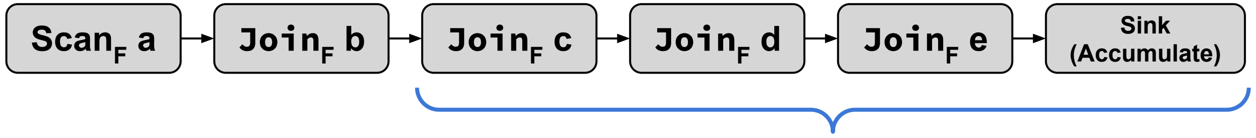
D-Representation for Expression Reuse Example



Query Vertex Ordering:
[a, b, c, d, e]

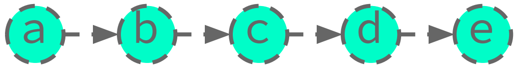
WHERE P(a, b, c, d, e)
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

Op_F c Scan necessary columns and apply filters



same computation given same **b** assignment

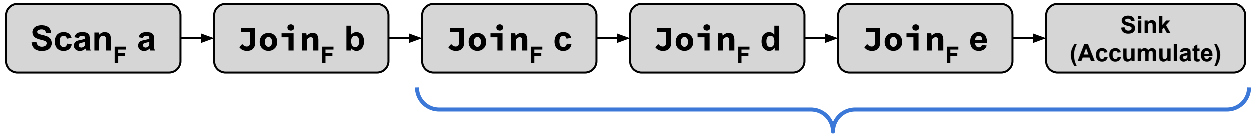
D-Representation for Expression Reuse Example



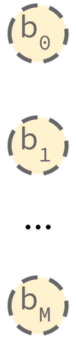
Query Vertex Ordering:
[a, b, c, d, e]

WHERE $P(a, b, c, d, e)$
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

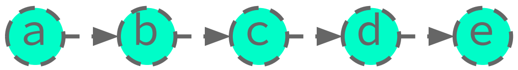
Op$_F$ c Scan necessary columns and apply filters



same computation given same **b** assignment



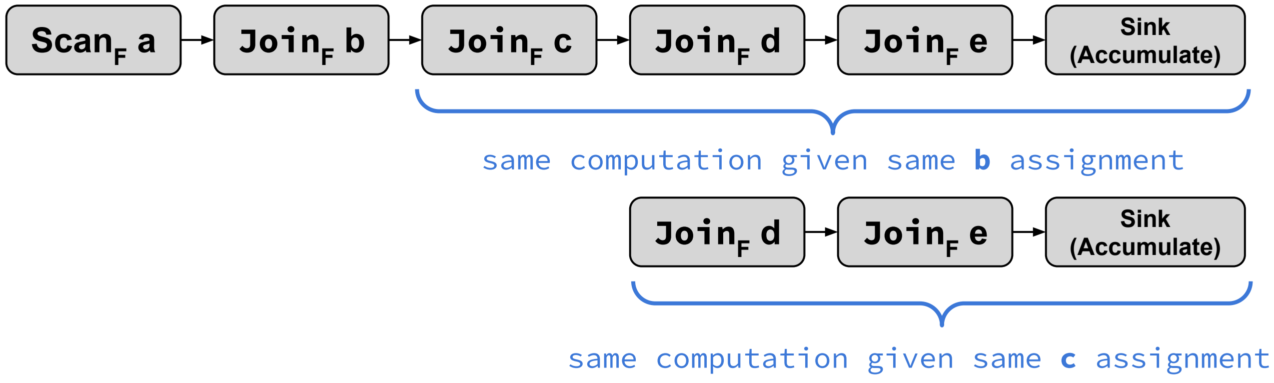
D-Representation for Expression Reuse Example

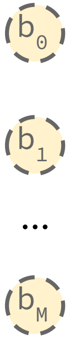


Query Vertex Ordering:
[a, b, c, d, e]

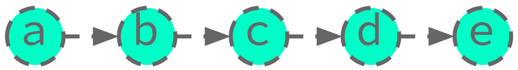
```
WHERE  P(a, b, c, d, e)
RETURN b.p_b,c.p_c1,c.p_c2,d.p_d,e.p_e1
```

Op_F c Scan necessary columns and apply filters





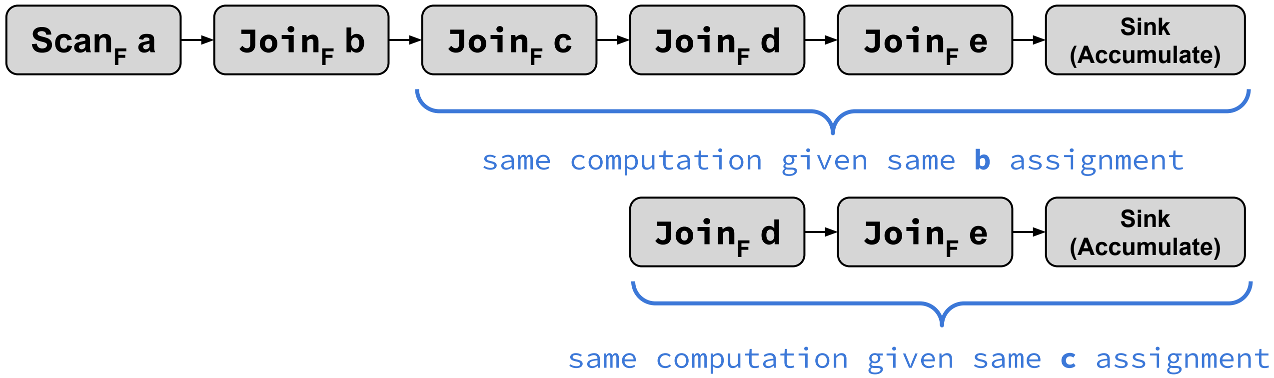
D-Representation for Expression Reuse Example

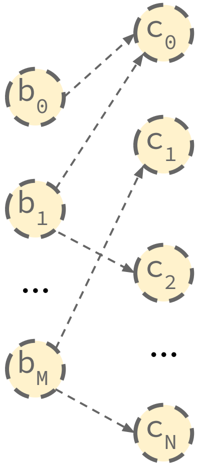


Query Vertex Ordering:
[a, b, c, d, e]

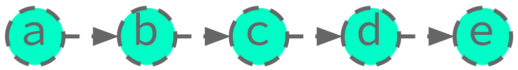
```
WHERE  P(a, b, c, d, e)
```
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

Op_F c Scan necessary columns and apply filters





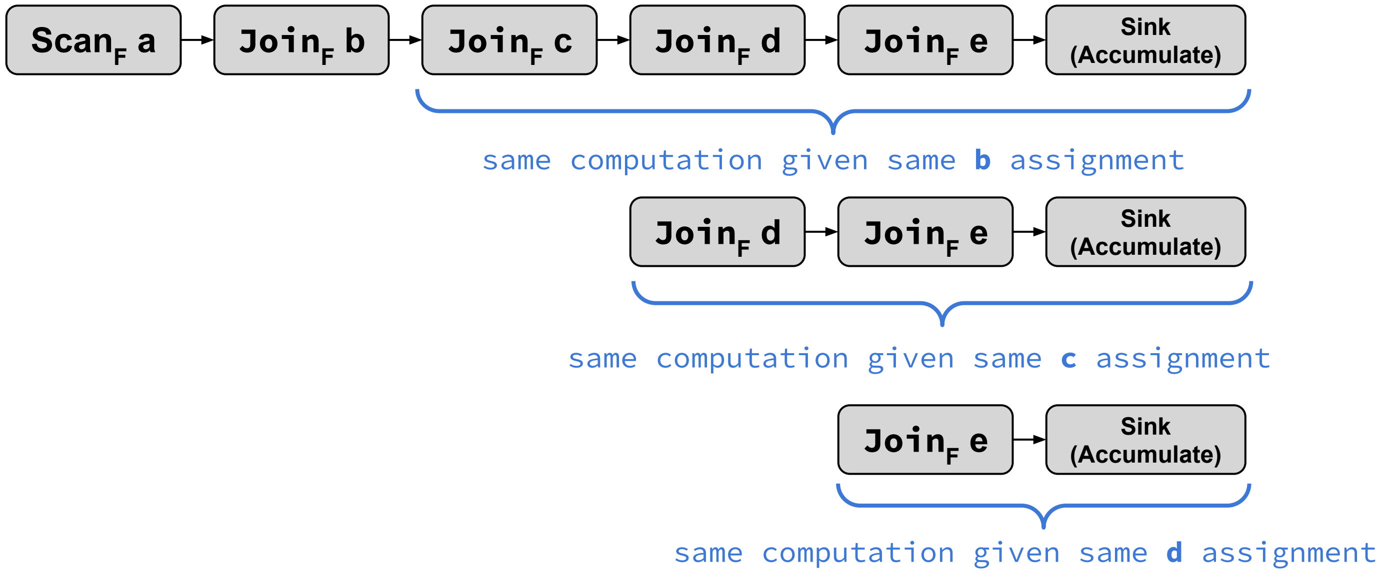
D-Representation for Expression Reuse Example



Query Vertex Ordering:
[a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN $b.p_{b}, c.p_{c1}, c.p_{c2}, d.p_{d}, e.p_{e1}$

Op_F c Scan necessary columns and apply filters



Scan_F a → Join_F b → Join_F c → Join_F d → Join_F e → Sink (Accumulate)

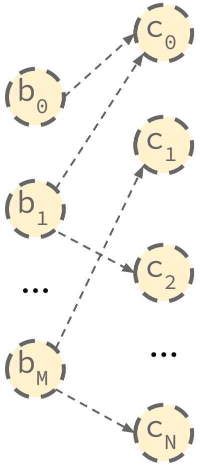
same computation given same **b** assignment

Join_F d → Join_F e → Sink (Accumulate)

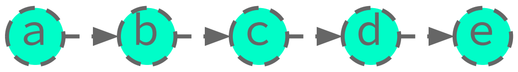
same computation given same **c** assignment

Join_F e → Sink (Accumulate)

same computation given same **d** assignment



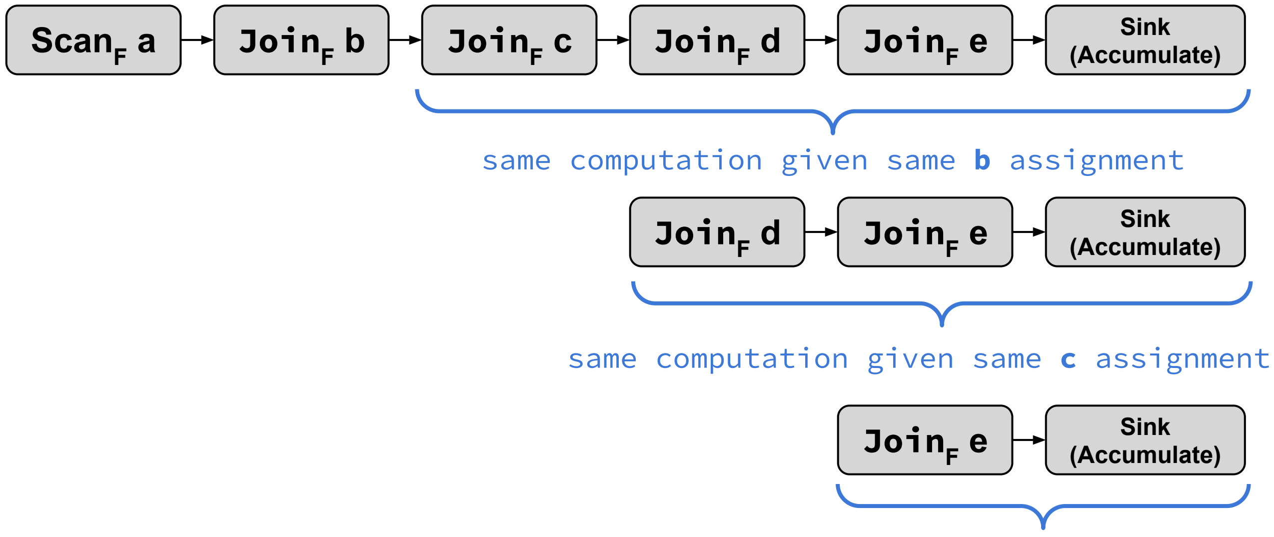
D-Representation for Expression Reuse Example



Query Vertex Ordering:
[a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

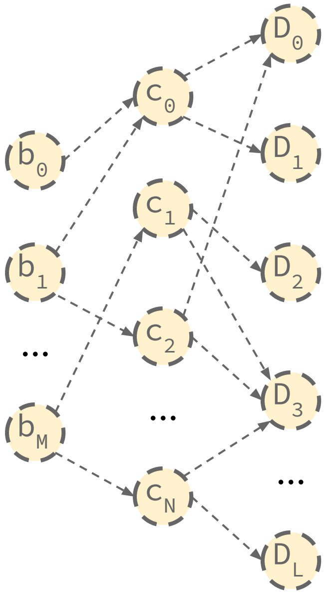
Op$_F$ c Scan necessary columns and apply filters



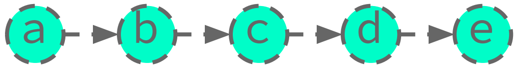
same computation given same **b** assignment

same computation given same **c** assignment

same computation given same **d** assignment



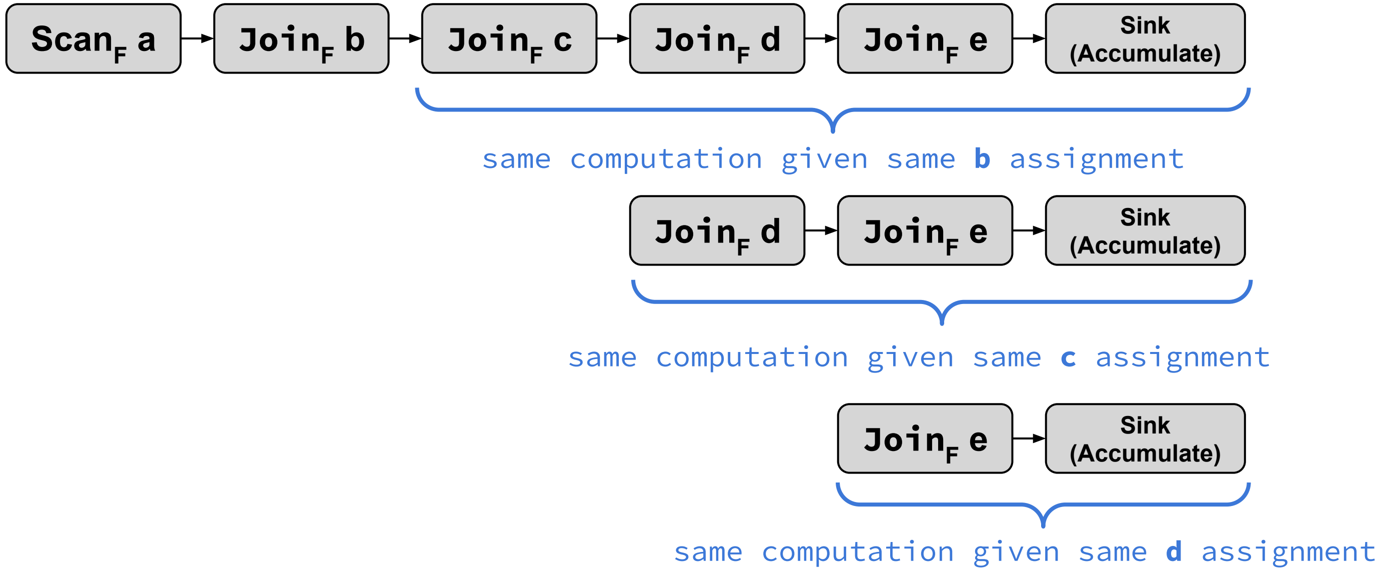
D-Representation for Expression Reuse Example



Query Vertex Ordering:
[a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

Op$_F$ c Scan necessary columns and apply filters



Scan$_F$ a → **Join$_F$ b** → **Join$_F$ c** → **Join$_F$ d** → **Join$_F$ e** → **Sink (Accumulate)**

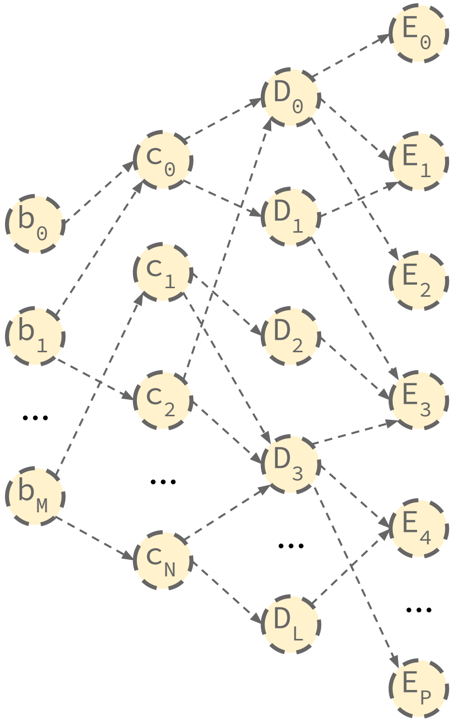
same computation given same **b** assignment

Join$_F$ d → **Join$_F$ e** → **Sink (Accumulate)**

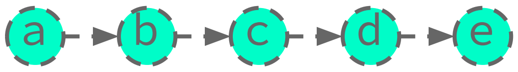
same computation given same **c** assignment

Join$_F$ e → **Sink (Accumulate)**

same computation given same **d** assignment



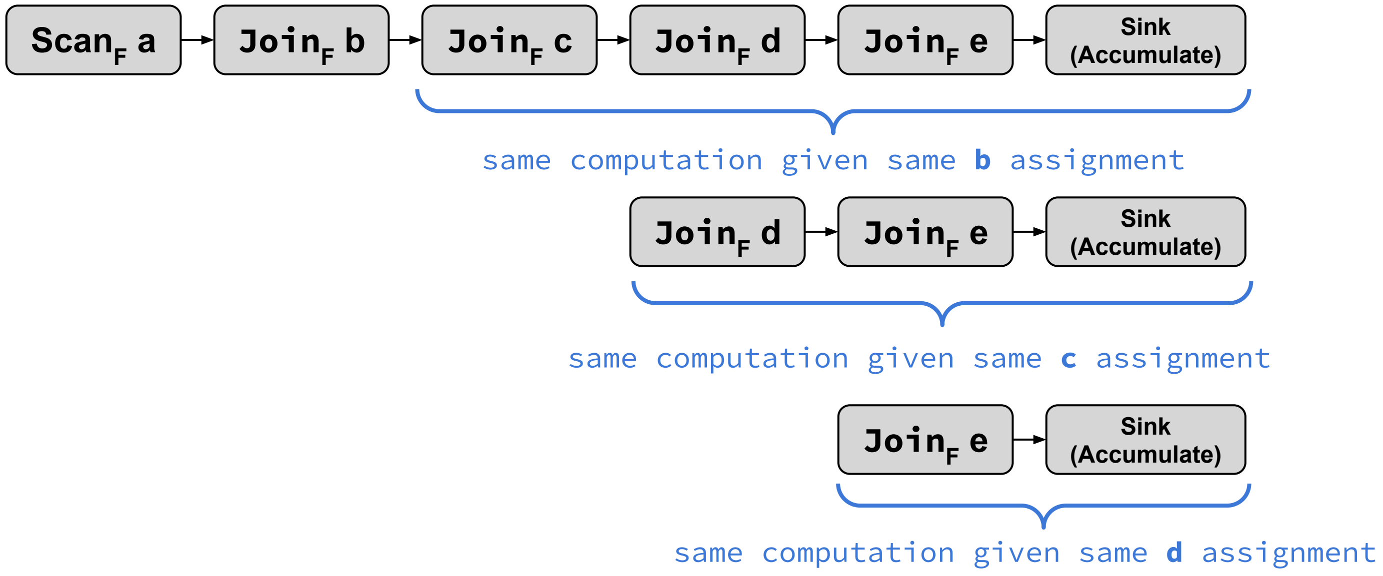
D-Representation for Expression Reuse Example



Query Vertex Ordering:
[a, b, c, d, e]

WHERE P(a, b, c, d, e)
RETURN $b.p_b, c.p_{c1}, c.p_{c2}, d.p_d, e.p_{e1}$

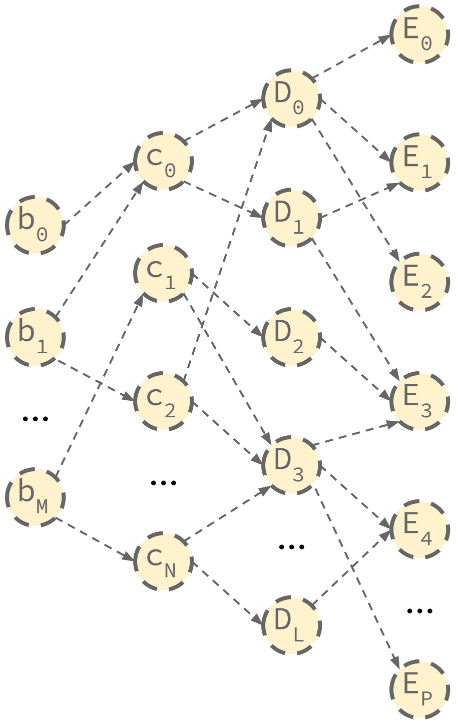
Op_F c Scan necessary columns and apply filters

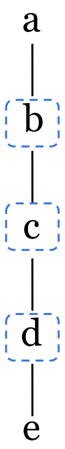


same computation given same **b** assignment

same computation given same **c** assignment

same computation given same **d** assignment





Fin. Questions?

GrainDB: A Relational-core Graph-Relational DBMS. Guodong Jin, Nafisa Anzum, and Semih Salihoglu. In CIDR, 2022.

Making RDBMSs Efficient on Graph Workloads Through Predefined Joins. Guodong Jin and Semih Salihoglu. In VLDB, 2022.

Fast In-memory SQL Analytics on Typed Graphs. Lin et al. In VLDB, 2016.

GQFast: Fast Graph Exploration with Context-aware Autocompletion. Lin et al. In ICDE, 2017.

Extending In-Memory Relational Database Engines with Native Graph Support. Hassan et al. In EDBT, 2017.

GRFusion: Graphs as First-Class Citizens in Main-Memory REIational Database Systems. Hassan et al. In SIGMOD, 2018.

DuckDB: An Embeddable Analytical Database. Hassan Mark Raasveldt, Hannes Muhleisen. In SIGMOD, 2019.

GSQL 2.0: Seamless Querying of Relational and Graph Databases. Deusch, Xu, Wu, Lee, 2018.

IBM DB2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. Tian et al., In SIGMOD 2020.

Beyond Worst-case Analysis for Joins With Minesweeper. Ngo et al., In PODS 2014.

Joins via Geometric Resolutions: Worst case and Beyond. Khamis et al., In TODS 2016.

Size bounds and query plans for relational joins. Albert Atserias, Martin Grohe, and Dániel Marx. In FOCS, 2008.

Worst-case optimal join algorithms. Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. In PODS, 2012.

Skew strikes back: new developments in the theory of join algorithms. Ngo, Hung Q., Christopher Ré and, and Atri Rudra. In SIGMOD R, 2013.

EmptyHeaded: a relational engine for graph processing. Christopher R. Aberger, et al. In TODS, 2017.

FDB: A Query Engine for Factorised Relational Databases. Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. In PVLDB, 2012.

Size Bounds for Factorized Representations of Query Results. Dan Olteanu and Jákub Zavodný. In TODS, 2015.

Fin. Questions?

Will be on the academic job market for 2023! (http://amine.io/)

UNIVERSITY OF WATERLOO | DSG Data Systems Group