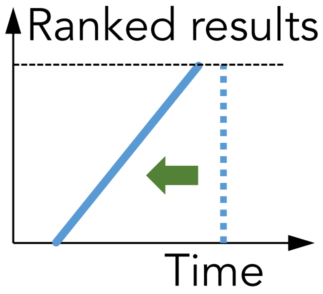


SIGMOD 2020 tutorial

Optimal Join Algorithms meet Top-k

Nikolaos Tziavelis, Wolfgang Gatterbauer, Mirek Riedewald

Northeastern University, Boston



Part 1 : Top-k

Northeastern University Khoury College of Computer Sciences

DATALAB @Northeastern

Slides: https://northeastern-datalab.github.io/topk-join-tutorial/

DOI: https://doi.org/10.1145/3318464.3383132

Data Lab: https://db.khoury.northeastern.edu

Why "Optimal Join Algorithms meet Top-k"?

Optimal Join algorithms

Return all results over joins

⇒ How to avoid large intermediate results?

Top-k

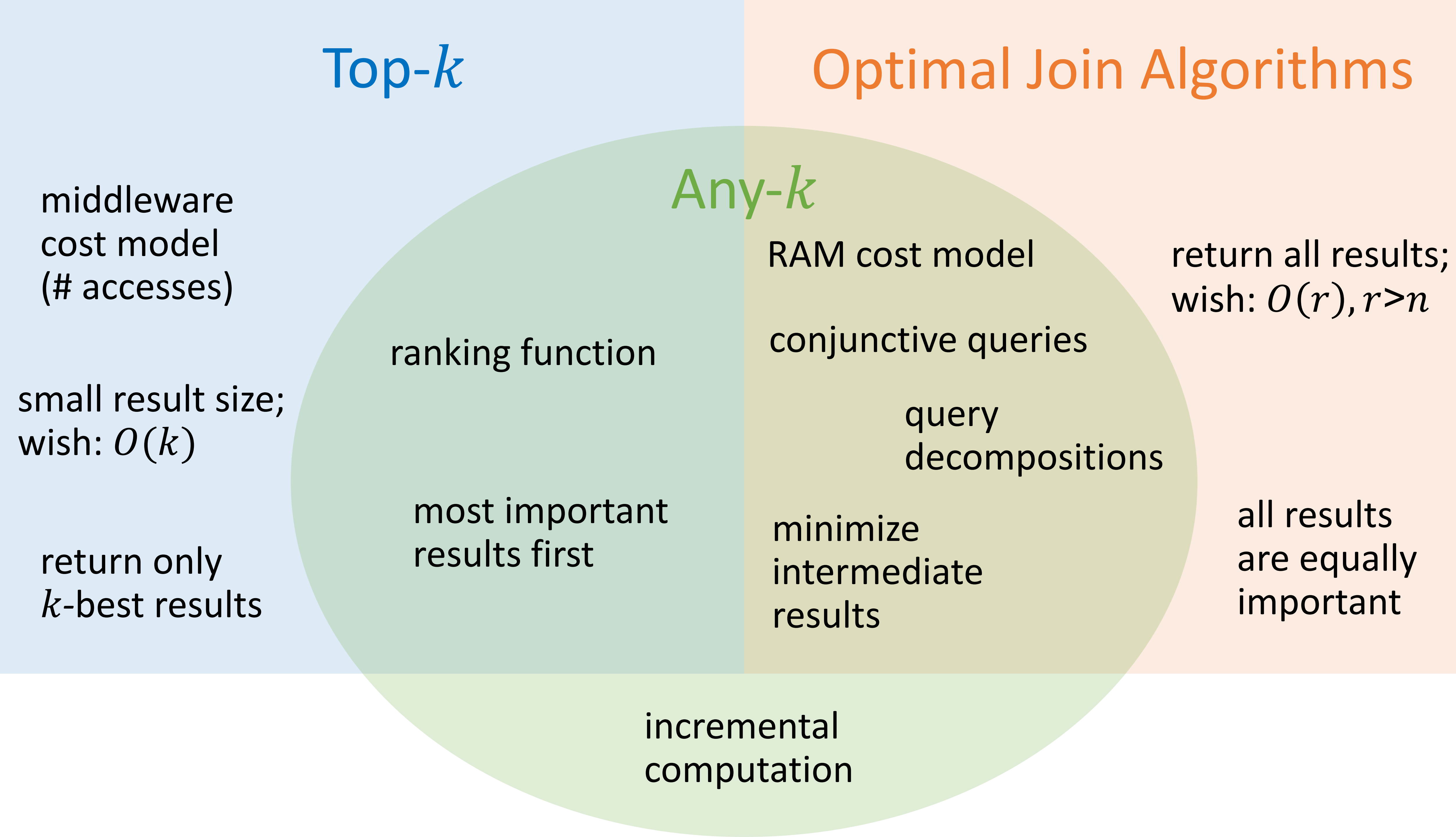
Given k, return k “best” results

⇒ How to avoid working on any lower ranked results?

Ranked Enumeration (Any-k)

Incrementally return the k “best” results over joins (for any k = 1, 2, ...)

⇒ How to most effectively push sorting through joins?


Top-k
Optimal Join Algorithms
Any-k
middleware cost model (# accesses)
small result size; wish: $O(k)$
return only k-best results
ranking function
most important results first
RAM cost model
conjunctive queries
query decompositions
minimize intermediate results
return all results; wish: $O(r), r>n$
all results are equally important
incremental computation

Outline tutorial

- Part 1: Top-k (Wolfgang): ~20min
  - Top-k selection problem
  - Threshold algorithm [Fagin+ '03]
  - Top-k join problem
  - J* algorithm [Natsev+ '01]
  - Discussion on cost models
- Part 2: Optimal Join Algorithms (Mirek): ~30min
- Part 3: Ranked enumeration over joins (Nikolaos): ~40min

Top-k Selection Query: overall setup

- n objects $X_1, X_2, \dots, X_n$ with ℓ numeric weight attributes $w_1, w_2, \dots, w_\ell$
- weight of object = aggregate function over its weights $\rho(w_1, w_2, \dots, w_\ell) = \rho(X)$
- Goal: Find top-k objects according to some order (e.g. min)

In most original papers assumed to be max!

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

$n = 5, \ell = 3, k = 2$

Example aggregate function:
$\rho = \text{sum}\{w_1, w_2, w_3\}$

Top-k: a set of k objects s.t. $\rho(X_i) \leq \rho(X_j)$ for every $X_i \in T$ and every $X_j \notin T$

Top-k Selection Query: information in different relations

- Weights are stored in ℓ distinct relations R_i

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_1	4
X_2	2
X_3	8
X_4	6
X_5	7

R_3

id	w_3
X_1	3
X_2	4
X_3	1
X_4	6
X_5	5

Top-k Selection Query: sorted access

- Weights are stored in ℓ distinct relations R_i
  - each R_i is sorted by attribute w_i

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_1	4
X_2	2
X_3	8
X_4	6
X_5	7

R_3

id	w_3
X_1	3
X_2	4
X_3	1
X_4	6
X_5	5

Top-k Selection Query: sorted access

- Weights are stored in ℓ distinct relations R_i
  - each R_i is sorted by attribute w_i

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

Notice we sort in increasing order

Top-k Selection Query: "middleware" assumption

- Weights are stored in ℓ distinct relations R_i
  - each R_i is sorted by attribute w_i
- Goal: Find top-k with minimal access cost
  - get next object in R_i sequentially: "sorted" sequential access cost c_{seq}
  - obtain the weight for a specific object in R_i : random access (index lookup) cost c_{rand}

Assumption 1: Middleware cost model:
we aggregate rankings of other services.
- *we only pay for accesses to attribute lists*
- *2 types of access: sequential / random*

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

Notice we sort in increasing order

Top-k Selection Query as a Join Problem

- Weights are stored in ℓ distinct relations R_i
  - each R_i is sorted by attribute w_i
- Goal: Find top-k with minimal access cost
  - get next object in R_i sequentially: "sorted" sequential access cost c_{seq}
  - obtain the weight for a specific object in R_i : random access (index lookup) cost c_{rand}

Assumption 1: Middleware cost model:
we aggregate rankings of other services.
- *we only pay for accesses to attribute lists*
- *2 types of access: sequential / random*

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

```
select   R1.id,
    sum(w1,w2,w3) as weight
from     R1, R2, R3
where  R1.id=R2.id
    and  R2.id=R3.id
order by weight
limit 2
```

~ Joins on unique object id:
1-1 relationships

Naive algorithm: retrieve all items

- Weights are stored in ℓ distinct relations R_i
  - each R_i is sorted by attribute w_i
- Goal: Find top-k with minimal access cost
  - get next object in R_i sequentially: "sorted" sequential access cost c_{seq}
  - obtain the weight for a specific object in R_i : random access (index lookup) cost c_{rand}

Assumption 1: Middleware cost model:
we aggregate rankings of other services.
- *we only pay for accesses to attribute lists*
- *2 types of access: sequential / random*

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

```
select   R1.id,
    sum(w1,w2,w3) as weight
from     R1, R2, R3
where  R1.id=R2.id
    and  R2.id=R3.id
order by weight
limit 2
```

Naive algorithm: retrieve all items, sort, return top-k

Cost = $n \cdot \ell \cdot c_{\text{sort}}$

Assumption 2: monotonicity of ρ

- Weights are stored in ℓ distinct relations R_i
  - each R_i is sorted by attribute w_i
- Goal: Find top-k with minimal access cost
  - get next object in R_i sequentially: "sorted" sequential access cost c_{seq}
  - obtain the weight for a specific object in R_i : random access (index lookup) cost c_{rand}

Assumption 1: Middleware cost model:
we aggregate rankings of other services.
- *we only pay for accesses to attribute lists*
- *2 types of access: sequential / random*

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15
X_4	7	6	6	18
X_5	8	7	5	20

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

```
select   R1.id,
    sum(w1,w2,w3) as weight
from     R1, R2, R3
where  R1.id=R2.id
    and  R2.id=R3.id
order by weight
limit 2
```

Assumption 2: The aggregate function ρ is monotone:
$\rho(w_1, w_2, \ldots, w_\ell) \leq \rho(w_1', w_2', \ldots, w_\ell')$ if $w_i \leq w_i'$ for all i

Part 3: tropical semiring (min, sum) is instance of "**selective dioid**" (i.e. min(a,b) = a or b).
ρ is decomposable: $\rho(w_1, w_2, w_3) = \rho\{w_1, w_2, w_3\}$

Important early work making these assumptions

- Fagin's algorithm:
  - Fagin. Combining fuzzy information from multiple systems. PODS 1996. https://doi.org/10.1145/237661.237715
  - Fagin. Fuzzy queries in multimedia database systems. PODS 1998. https://doi.org/10.1145/275487.275488
  - Fagin. Combining fuzzy information from multiple systems. JCSS 1999. https://doi.org/10.1006/jcss.1998.1600

- Threshold Algorithm (TA):
  - Nepal, Ramakrishna. Query processing issues in image (multimedia) databases. ICDE 1999. https://doi.org/10.1109/ICDE.1999.754894
  - Güntzer, Balke, Kießling. Optimizing multifeature queries for image databases. VLDB 2000. https://dl.acm.org/doi/10.5555/645926.671875
  - Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

2014 Gödel Prize on "a framework to design and analyze algorithms where aggregation of information from multiple data sources is needed... introduced the notion of instance optimality"

Outline tutorial

- Part 1: Top-k (Wolfgang): ~20min
  - Top-k selection problem
  - Threshold algorithm [Fagin+ '03]
  - Top-k join problem
  - J* algorithm [Natsev+ '01]
  - Discussion on cost models
- Part 2: Optimal Join Algorithms (Mirek): ~30min
- Part 3: Ranked enumeration over joins (Nikolaos): ~40min

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
    a. Set threshold τ to the aggregate of the weights last seen in sorted access

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \mathrm{sum}(3,2,1) = 6$

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
   a. Set threshold τ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights ρ of all objects seen

id	w_1	w_2	w_3
X_1	3		
X_2		2	
X_3			1

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \text{sum}(3,2,1) = 6$

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
   a. Set threshold τ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights ρ of all objects seen

id	w_1	w_2	w_3
X_1	3	4	3
X_2		2	
X_3			1

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \mathrm{sum}(3,2,1) = 6$

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
   a. Set threshold τ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights ρ of all objects seen

id	w_1	w_2	w_3
X_1	3	4	3
X_2	4	2	4
X_3			1

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \text{sum}(3,2,1) = 6$

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
   a. Set threshold τ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights ρ of all objects seen

id	w_1	w_2	w_3
X_1	3	4	3
X_2	4	2	4
X_3	6	8	1

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \mathrm{sum}(3,2,1) = 6$

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
   a. Set threshold τ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights ρ of all objects seen

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15

k=2

focus only on top-k
(can purge rest)

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \text{sum}(3,2,1) = 6$

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
   a. Set threshold τ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights ρ of all objects seen
   c. Continue until the aggregate weights ρ of the top-k ≤ τ

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15

k=2

focus only on top-k
(can purge rest)

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \text{sum}(3,2,1) = 6$

$10 \not\leq 6$: continue: access next objects sequentially

Threshold algorithm [Fagin+ 03]

1. Access next objects in all R_i sequentially
   a. Set threshold τ to the aggregate of the weights last seen in sorted access
   b. Use random accesses and compute the aggregate weights ρ of all objects seen
   c. Continue until the aggregate weights ρ of the top-k ≤ τ

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10
X_3	6	8	1	15

k=2

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \text{sum}(3,2,1) = 6$

$\tau = \text{sum}(4,4,3) = 11$

focus only on top-k
(can purge rest)

10 ≤ 11: stop!

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Threshold algorithm [Fagin+ 03]

- Why can we avoid looking at X_4?

From the monotonicity property: for any object not seen, the score of the object is bigger than the threshold

id	w_1	w_2	w_3	sum
X_1	3	4	3	10
X_2	4	2	4	10

$4 \leq 7$ $\quad$ $4 \leq 6$ $\quad$ $3 \leq 6$ $\quad$ monotonicity $\Rightarrow$ $\rho(X_4) \geq \tau$

R_1

id	w_1
X_1	3
X_2	4
X_3	6
X_4	7
X_5	8

R_2

id	w_2
X_2	2
X_1	4
X_4	6
X_5	7
X_3	8

R_3

id	w_3
X_3	1
X_1	3
X_2	4
X_5	5
X_4	6

$\tau = \text{sum}(4,4,3) = 11$

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Instance Optimality of Threshold Algorithm (TA)

- The TA algorithm is instance cost-optimal
  - within a constant factor of the best algorithm on any database*

- Let $\text{cost}(A, D)$ = access cost of algorithm A on database D:
  - $\text{cost}(\text{TA}, D) = O(\text{cost}(A, D))$ for all A and D

* Excluding those that make "wild guesses" = random access to object without first seeing it with sorted access

[Fagin+ 03] Fagin, Lotem, Naor. Optimal aggregation algorithms for middleware. JCSS 2003. https://doi.org/10.1016/S0022-0000(03)00026-6

Outline tutorial

- Part 1: Top-k (Wolfgang): ~20min
  - Top-k selection problem
  - Threshold algorithm [Fagin+ '03]
  - Top-k join problem
  - J* algorithm [Natsev+ '01]
  - Discussion on cost models
- Part 2: Optimal Join Algorithms (Mirek): ~30min
- Part 3: Ranked enumeration over joins (Nikolaos): ~40min

Goal: Generalize TA setup to arbitrary join patterns

- Same cost model: measuring access cost
  - to simplify, we ignore random accesses

- *many-to-many relationships*
- *no unique identifier per join result*
- *arbitrary join conditions possible*

natural join

```
select   A1, A2, A3, A4,
   sum(w1,w2,w3) as weight
from     R1, R2, R3
where  R1.A2=R2.A2
   and   R2.A3=R3.A3
order by weight
limit 1
```

R_1

A_1	A_2	w_1
a	b	1
c	b	4
b	d	5

R_2

A_2	A_3	w_2
d	c	1
b	c	2
b	a	3

R_3

A_3	A_4	w_3
c	d	1
a	a	2
a	d	3

Outline tutorial

- Part 1: Top-k (Wolfgang): ~20min
  - Top-k selection problem
  - Threshold algorithm [Fagin+ '03]
  - Top-k join problem
  - J* algorithm [Natsev+ '01]
  - Discussion on cost models
- Part 2: Optimal Join Algorithms (Mirek): ~30min
- Part 3: Ranked enumeration over joins (Nikolaos): ~40min

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
()	R_1:1	0+0+0=0

R_1

	A_1	A_2	w_1
1	a	b	1
2	c	b	4
3	b	d	5

R_2

	A_2	A_3	w_2
1	d	c	1
2	b	c	2
3	b	a	3

R_3

	A_3	A_4	w_3
1	c	d	1
2	a	a	2
3	a	d	3

[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer", one "deeper"

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1)	R_2:1	1+0+0=1
()	R_1:2	1+0+0=1

R_1

	A_1	A_2	w_1
1	a	b	1
2	c	b	4
3	b	d	5

R_2

	A_2	A_3	w_2
1	d	c	1
2	b	c	2
3	b	a	3

R_3

	A_3	A_4	w_3
1	c	d	1
2	a	a	2
3	a	d	3

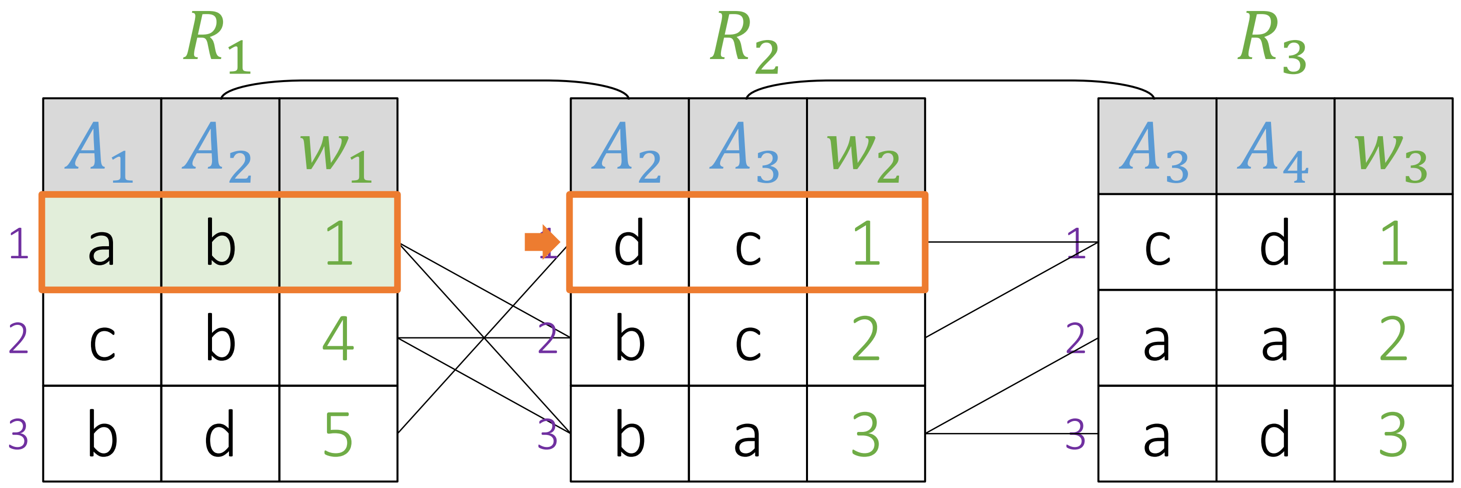
[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer", one "deeper"

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1)	R_2:1	1+0+0=1
()	R_1:2	1+0+0=1



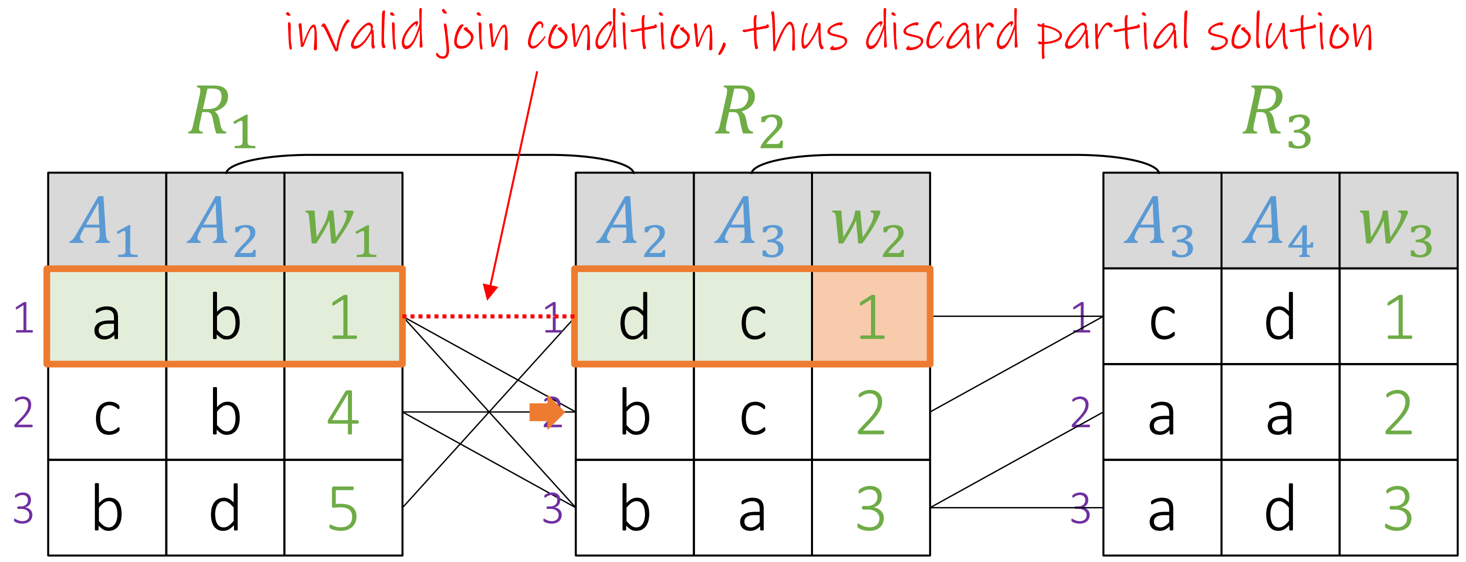
[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer", one "deeper"

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1)	R_2:2	1+1+0=2
()	R_1:2	1+0+0=1

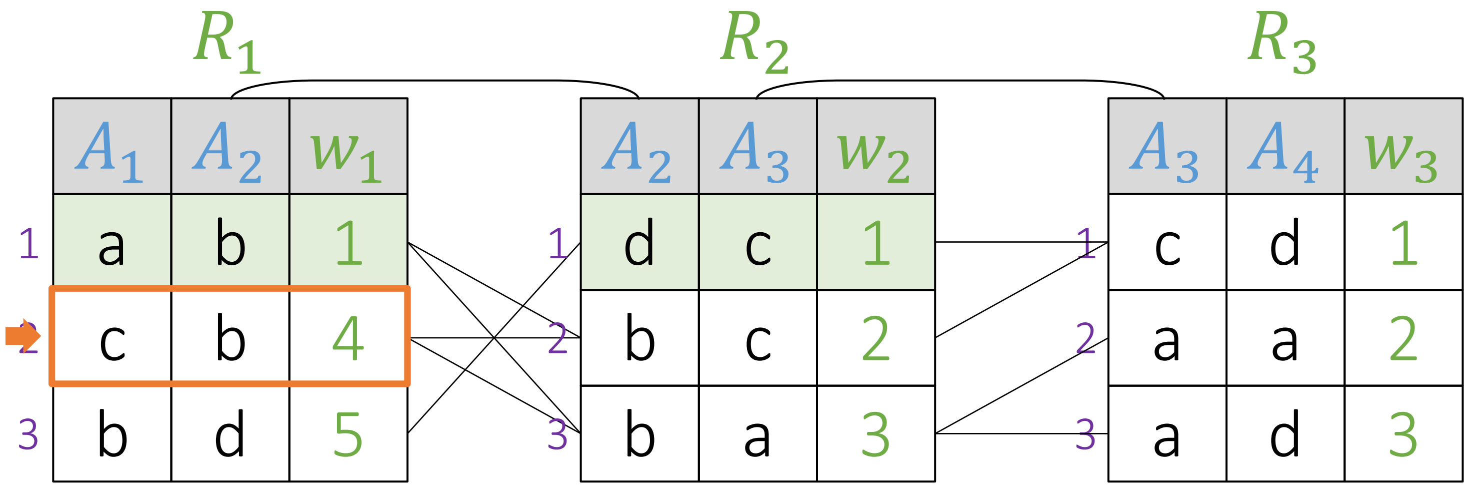


J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1)	R_1:2	1+1+0=2
()	R_1:2	1+0+0=1

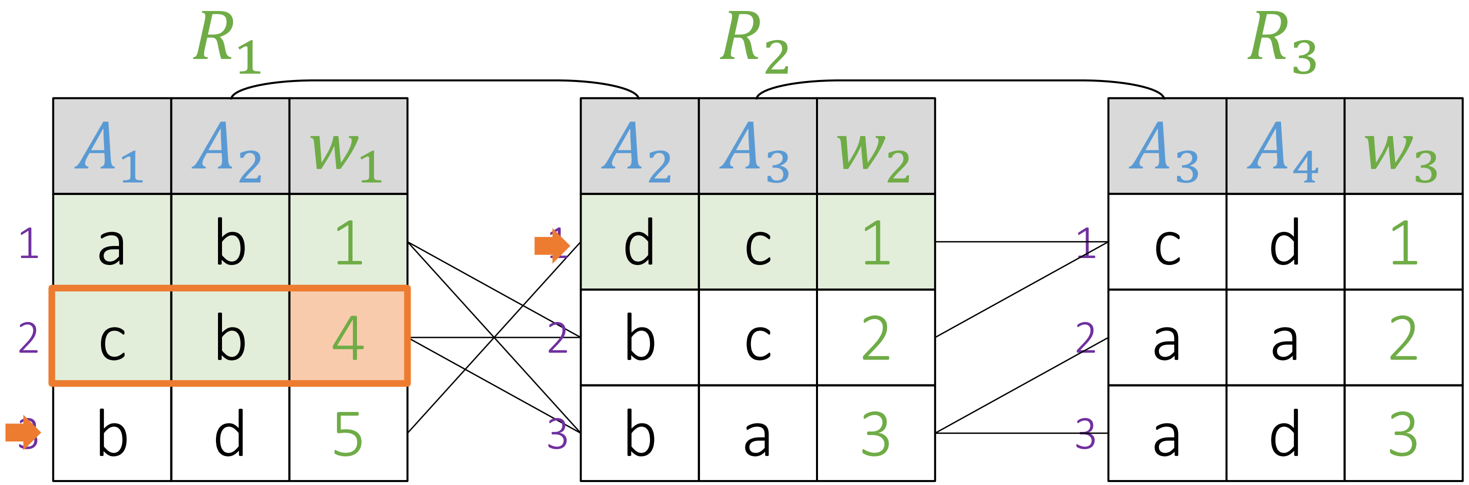


J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer", one "deeper"

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1)	R_2:2	1+1+0=2
(2)	R_2:1	4+0+0=4
()	R_1:3	4+0+0=4

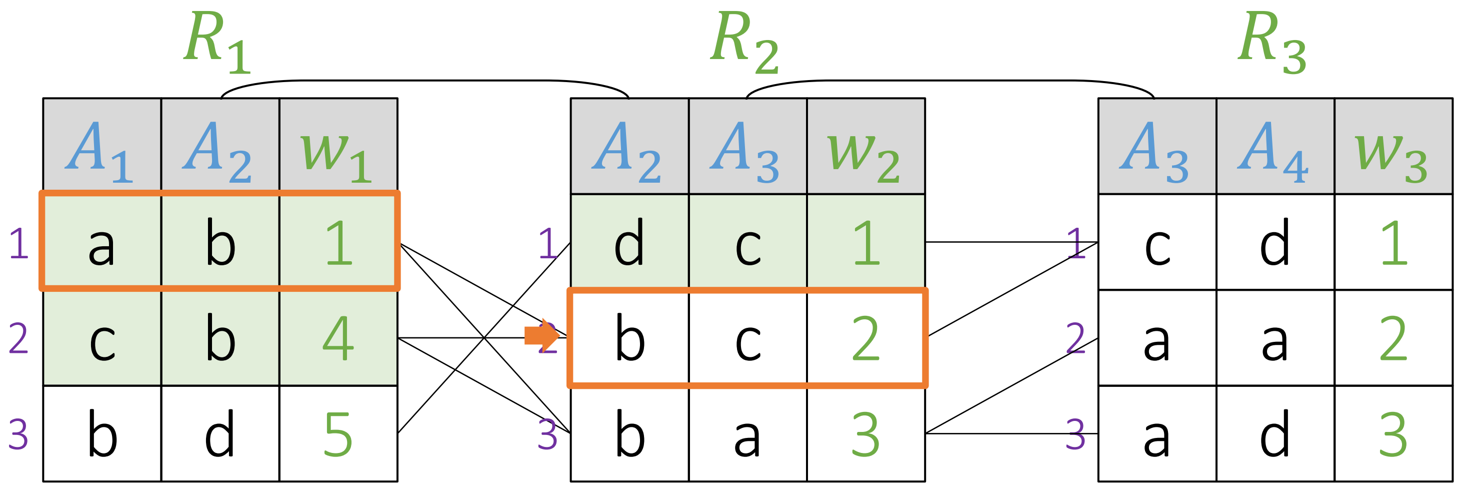


J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1)	R_2:2	1+1+0=2
(2)	R_2:1	4+0+0=4
()	R_1:3	4+0+0=4



[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1,2)	R_3:1	1+2+0=3
(1)	R_2:3	1+2+0=3
(2)	R_2:1	4+0+0=4
()	R_1:3	4+0+0=4

R_1

	A_1	A_2	w_1
1	a	b	1
2	c	b	4
3	b	d	5

R_2

	A_2	A_3	w_2
1	d	c	1
2	b	c	2
3	b	a	3

R_3

	A_3	A_4	w_3
1	c	d	1
2	a	a	2
3	a	d	3

[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer", one "deeper"

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1,2)	R_3:1	1+2+0=3
(1)	R_2:3	1+2+0=3
(2)	R_2:1	4+0+0=4
()	R_1:3	4+0+0=4

R_1

	A_1	A_2	w_1
1	a	b	1
2	c	b	4
3	b	d	5

R_2

	A_2	A_3	w_2
1	d	c	1
2	b	c	2
3	b	a	3

R_3

	A_3	A_4	w_3
1	c	d	1
2	a	a	2
3	a	d	3

[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer", one "deeper"

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1,2,1)		1+2+1=4
(1,2)	R_3:2	1+2+1=4
(1)	R_2:3	1+2+0=3
(2)	R_2:1	4+0+0=4
()	R_1:3	4+0+0=4

R_1

	A_1	A_2	w_1
1	a	b	1
2	c	b	4
3	b	d	5

R_2

	A_2	A_3	w_2
1	d	c	1
2	b	c	2
3	b	a	3

R_3

	A_3	A_4	w_3
1	c	d	1
2	a	a	2
3	a	d	3

[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one "longer", one "deeper"

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound
(1,2,1)		1+2+1=4
(1,2)	R_3:2	1+2+1=4
(1,3)	R_3:1	1+3+0=4
(2)	R_2:1	4+0+0=4
()	R_1:3	4+0+0=4

R_1

	A_1	A_2	w_1
1	a	b	1
2	c	b	4
3	b	d	5

R_2

	A_2	A_3	w_2
1	d	c	1
2	b	c	2
3	b	a	3

R_3

	A_3	A_4	w_3
1	c	d	1
2	a	a	2
3	a	d	3

[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

J* Algorithm [Natsev+ 01]

- Idea: A* search on the Cartesian product to find top-k join results
  - Keep Priority Queue (PQ) of partial results
  - Pop partial result with smallest lower bound (based on what has been seen) and access lists to extend it
  - If still incomplete, push back 2 new ones: one “longer”, one “deeper”

left to right (showing row id's)

Partial Solution	Next Tuple	Lower bound	
(1,2,1)		1+2+1=4	top-1
(1,2)	R_3:2	1+2+1=4	
(1,3)	R_3:1	1+3+0=4	
(2)	R_2:1	4+0+0=4	
()	R_1:3	4+0+0=4	

R_1

	A_1	A_2	w_1
1	a	b	1
2	c	b	4
3	b	d	5

R_2

	A_2	A_3	w_2
1	d	c	1
2	b	c	2
3	b	a	3

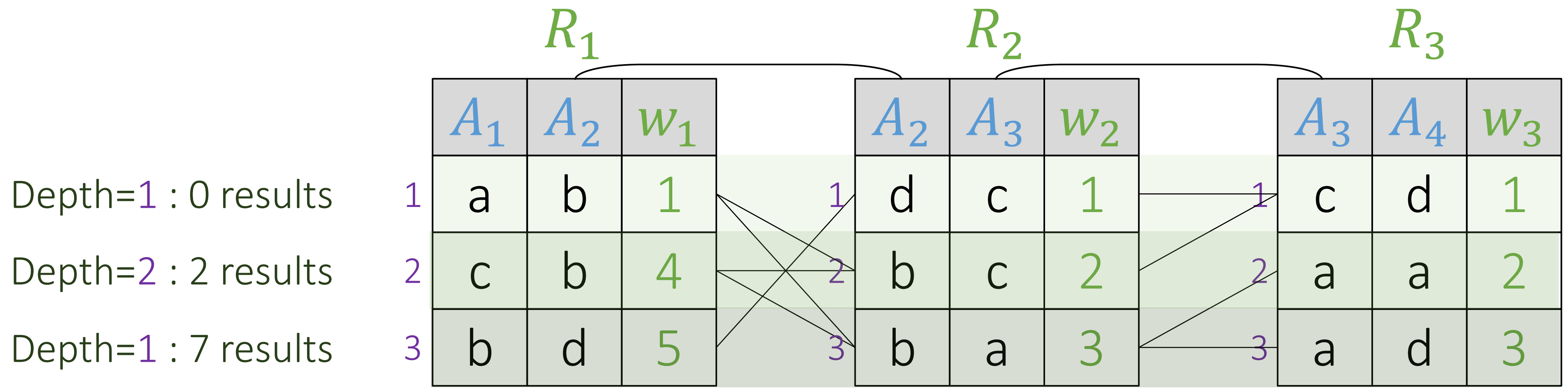
R_3

	A_3	A_4	w_3
1	c	d	1
2	a	a	2
3	a	d	3

[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

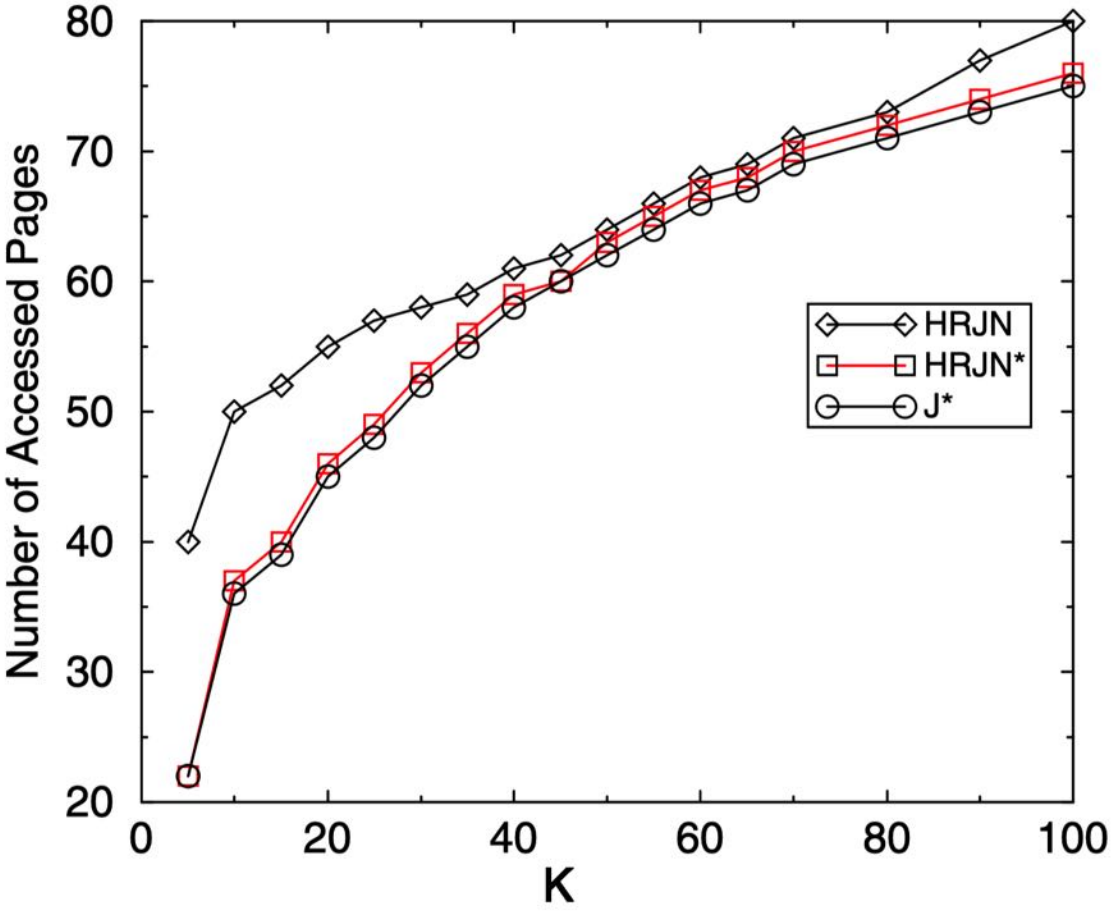
J* w/ iterative deepening [Natsev+ 01] & Rank Join [Ilyas+ 04]

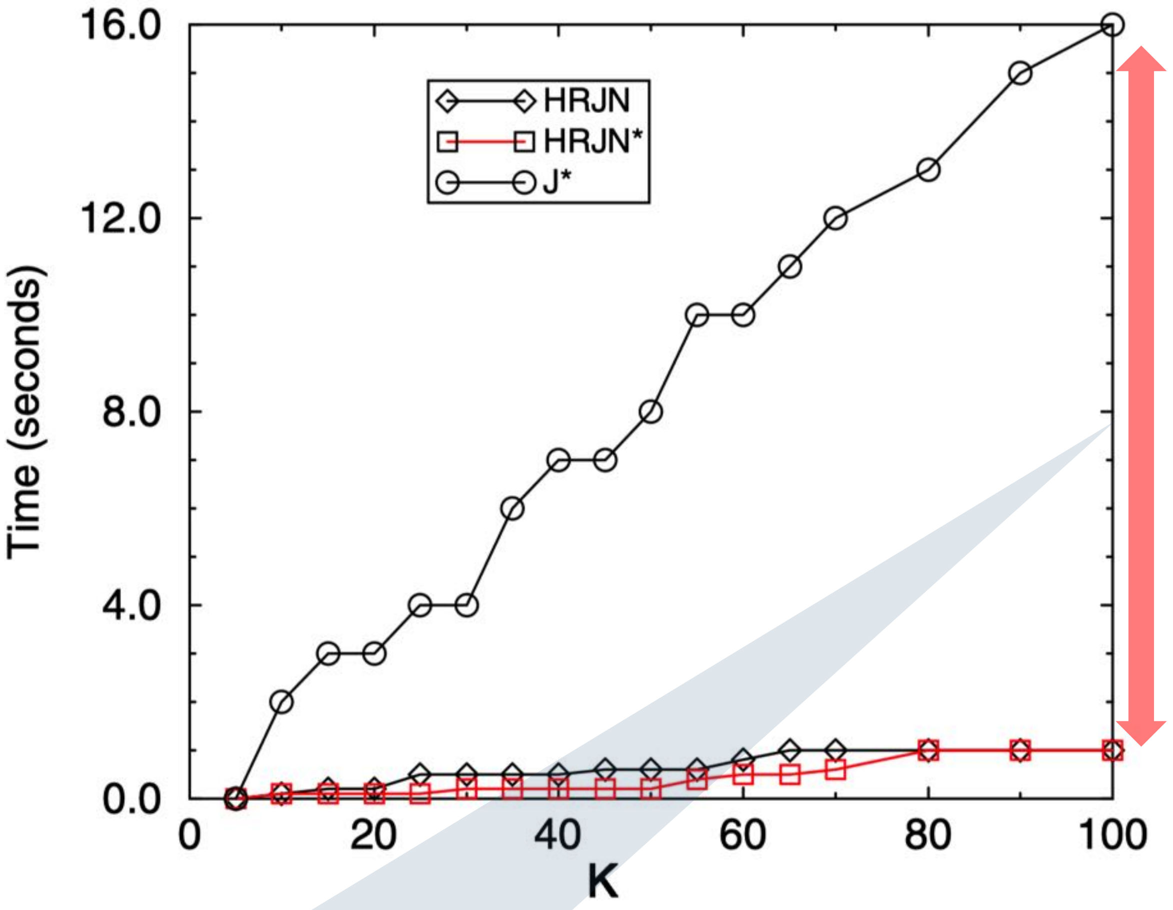
- To guarantee instance optimality for J*, go deeper only after producing all results (iterative deepening) [Natsev+ 01]
- Rank-Join [Ilyas+ 04]: Instead of A* type of search use a threshold value similarly to TA. Also instance-optimal in terms of accesses
- Many variants and much follow-up work (different join strategies, heuristics to prioritize relations, etc.)



[Ilyas+ 04] Ilyas, Aref, Elmagarmid. Supporting top-*k* join queries in relational databases. VLDBJ 2004. https://doi.org/10.1007/s00778-004-0128-2
[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

Figures from [Ilyas+ 04]





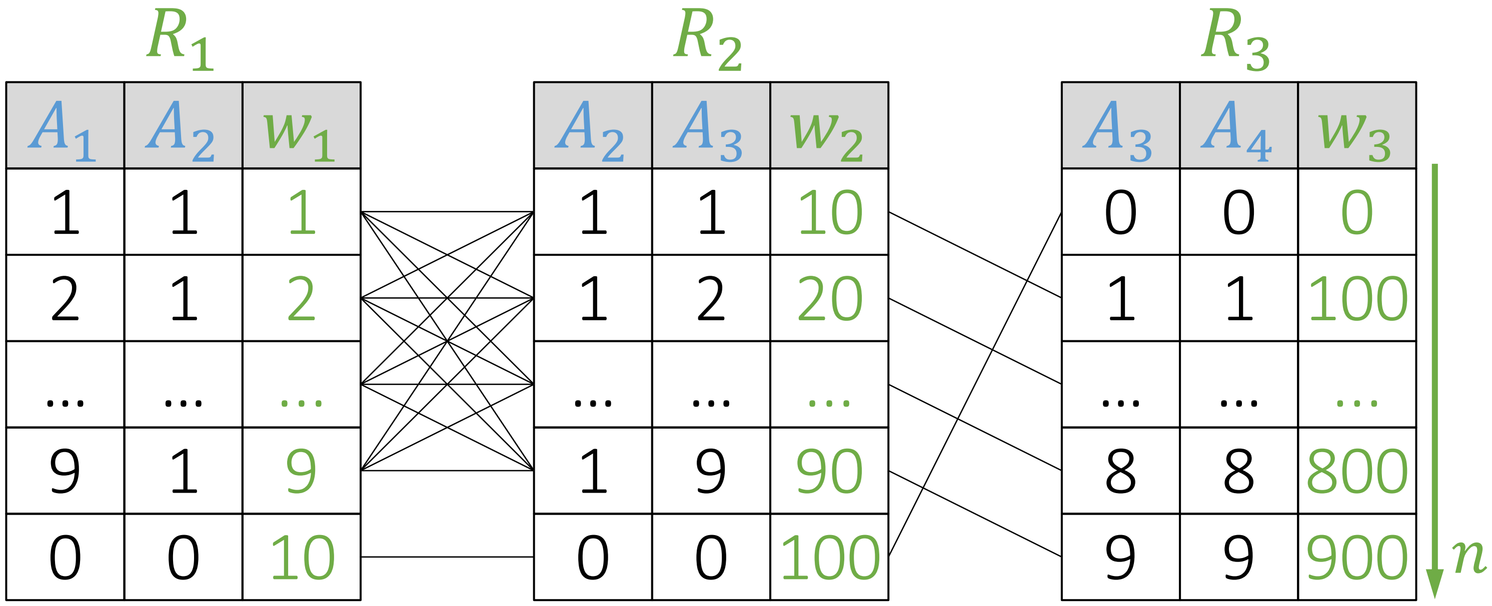
Similar access cost, but different times in practice.
Is # of access cost thus a reasonable cost model?

[Ilyas+ 04] Ilyas, Aref, Elmagarmid. Supporting top-*k* join queries in relational databases. VLDBJ 2004. https://doi.org/10.1007/s00778-004-0128-2

Outline tutorial

- Part 1: Top-k (Wolfgang): ~20min
  - Top-k selection problem
  - Threshold algorithm [Fagin+ '03]
  - Top-k join problem
  - J* algorithm [Natsev+ '01]
  - Discussion on cost models
- Part 2: Optimal Join Algorithms (Mirek): ~30min
- Part 3: Ranked enumeration over joins (Nikolaos): ~40min

Middleware cost model vs. in-database join computations



[Ilyas+ 04] Ilyas, Aref, Elmagarmid. Supporting top-*k* join queries in relational databases. VLDBJ 2004. https://doi.org/10.1007/s00778-004-0128-2
[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

Middleware cost model vs. in-database join computations

- J* and Rank-Join produce n^2 partial results to find top-1 result [§]
  - Are number of accesses a realistic measure for in-database join computation? E.g. if tables are available in a database, we don't have to fetch tuples over a network.

⇒ How to most effectively push sorting through joins?

RAM cost model
- In-memory join comp.
- quadratic cost
- *in-memory processing:* join time matters

R_1

A_1	A_2	w_1
1	1	1
2	1	2
…	…	…
9	1	9
0	0	10

R_2

A_2	A_3	w_2
1	1	10
1	2	20
…	…	…
1	9	90
0	0	100

R_3

A_3	A_4	w_3
0	0	0
1	1	100
…	…	…
8	8	800
9	9	900

n

Middleware cost model
- Minimize access depth
- linear cost
- *Information retrieval:* latency/ access cost matters

§ Assuming sorted accesses only. If random accesses allowed, another slightly more complicated example shows the same issue.

[Ilyas+ 04] Ilyas, Aref, Elmagarmid. Supporting top-*k* join queries in relational databases. VLDBJ 2004. https://doi.org/10.1007/s00778-004-0128-2

[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

A natural question:

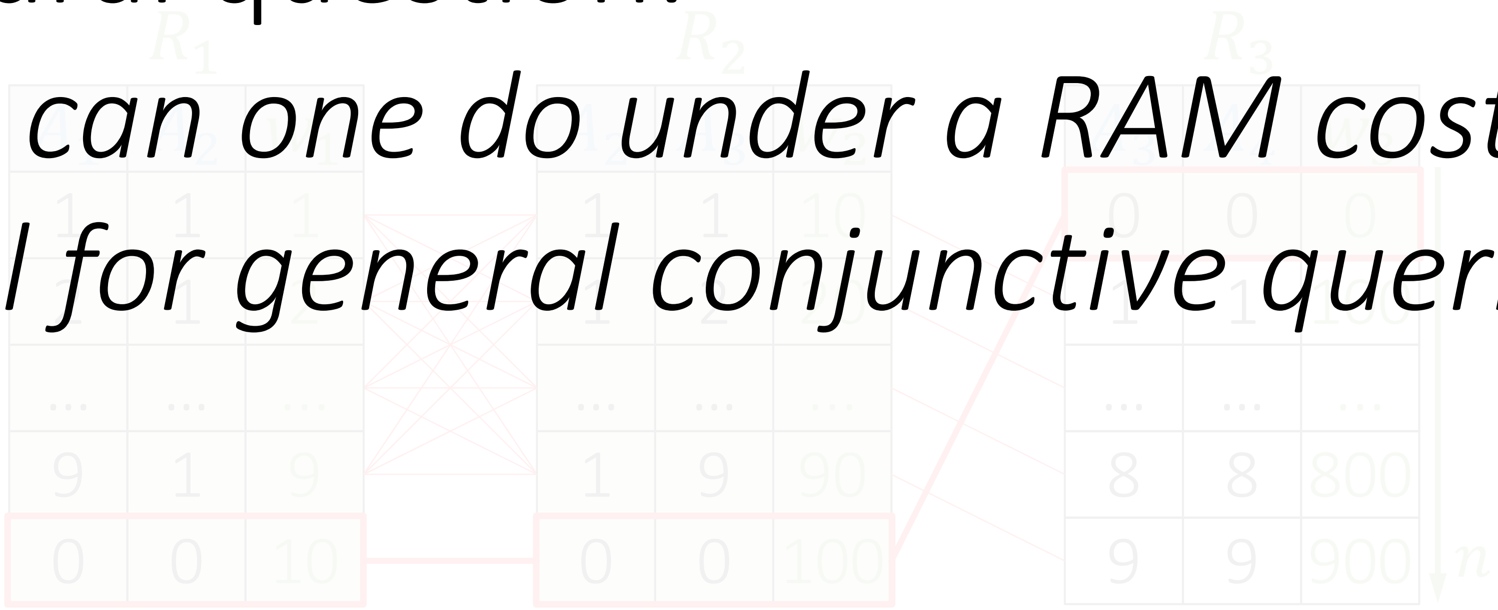
What can one do under a RAM cost model for general conjunctive queries?

Middleware cost model vs. in-database join computations

- J* and Rank-Join produce n^2 partial results to find top-1 result [§]
  - Are number of accesses a realistic measure for in-database join computation? E.g. if tables are available in a database, we don't have to fetch tuples over a network.

⇒ *How to most effectively push sorting through joins?*

RAM cost model
- In-memory join comp.
- quadratic cost
- *in-memory processing:* join time matters



Middleware cost model
- Minimize access depth
- linear cost
- *Information retrieval:* latency/ access cost matters

[§] Assuming sorted accesses only. If random accesses allowed, another slightly more complicated example shows the same issue.

[Ilyas+ 04] Ilyas, Aref, Elmagarmid. Supporting top-*k* join queries in relational databases. VLDBJ 2004. https://doi.org/10.1007/s00778-004-0128-2
[Natsev+ 01] Natsev, Chang, Smith, Li, Vitter. Supporting incremental join queries on ranked inputs. VLDB 2001. https://doi.org/doi/10.5555/645927.672365

An excerpt of rich literature, once access determines cost ...

- What if the ranking function is the distance from a desired (high-dimensional) point?
  - [Bruno+ TODS'02]: Rewrite as a range query and restart if #results < *k*
- What if we are allowed to pre-compute data structures and learn the ranking function at query time?
  - [Tsaparas+ ICDE'03]: Find linear ranking functions that act as "separators" (i.e., they change the top-*k* set)
  - [Chang+ SIGMOD'00]: Construct convex hulls for linear ranking functions
  - [Hristidis+ SIGMOD'01, Das+ VLDB'06]: Materialize ranked views for some selected ranking functions
- What if the ranking function is non-monotone?
  - [Zhang+ SIGMOD'06]: Use continuous function optimization methods
- What if the query model is different?
  - "SMART" [Wu+ VLDB'10]: Query contains disjunctions, partial results allowed to be returned
- ...

Please see dedicated tutorials and surveys on top-k

Bruno, Chaudhuri, Gravano. Top-*k* selection queries over relational databases: Mapping strategies and performance evaluation. TODS 2002. https://doi.org/10.1145/568518.568519
Tsaparas, Palpanas, Kotidis, Koudas, Srivastava. Ranked join indices. ICDE 2003. https://doi.org/10.1109/ICDE.2003.1260799
Chang, Bergman, Castelli, Li, Lo, Smith. The onion technique: Indexing for linear optimization queries. SIGMOD 2000. https://doi.org/10.1145/342009.335433
Hristidis, Koudas, Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. SIGMOD 2001. https://doi.org/10.1145/376284.375690
Das, Gunopulos, Koudas, Tsirogiannis. Answering top-*k* queries using views. VLDB 2006. http://www.vldb.org/conf/2006/p451-das.pdf
Zhang, Hwang, Chang, Wang, Lang, Chang. Boolean + ranking: querying a database by *k*-constrained optimization. SIGMOD 2006. https://doi.org/10.1145/1142473.1142515
Wu, Berti-Equille, Marian, Procopiuc, Srivastava. Processing top-*k* join queries. VLDB 2010. https://doi.org/10.14778/1920841.1920951

Outline tutorial

- Part 1: Top-k (Wolfgang): ~20min
  - Top-k selection problem
  - Threshold algorithm [Fagin+ '03]
  - Top-k join problem
  - J* algorithm [Natsev+ '01]
  - Discussion on cost models
- Part 2: Optimal Join Algorithms (Mirek): ~30min
- Part 3: Ranked enumeration over joins (Nikolaos): ~40min