The Quest for Faster Join Algorithms

Paris Koutris

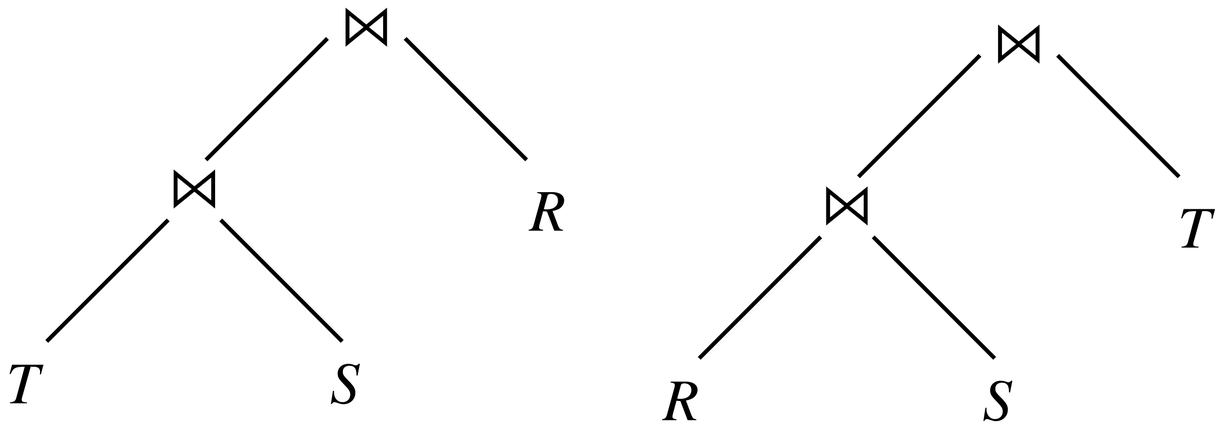
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Joins are a core relational operator

The Long History of Join Algorithms

- typically computed via a query plan of **binary join operators**
- many different algorithms to compute binary joins
  - nested loop join
  - hash join
  - sort-merge join
  - .. and a lot more!

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



But haven’t we already solved everything?

But haven’t we already solved everything?

No!

1. Query optimizers are notoriously sensitive to cardinality estimation errors leading to suboptimal plans
2. Binary join plans perform well for most joins, but poorly for others:
   - pattern matching on graphs
   - cyclic joins
3. There is a large gap between theory and practice!

What is the fastest algorithm to compute a join query?

We still don’t know!

What is the fastest algorithm to compute a join query?

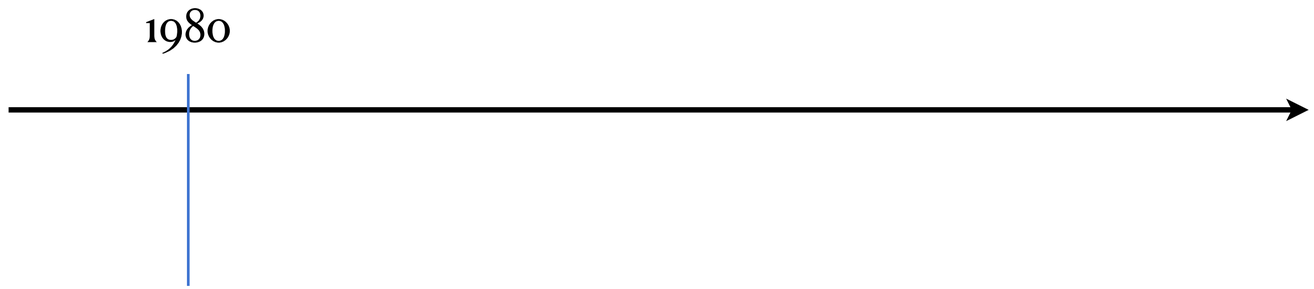
But we have made some amazing progress!

In this talk:

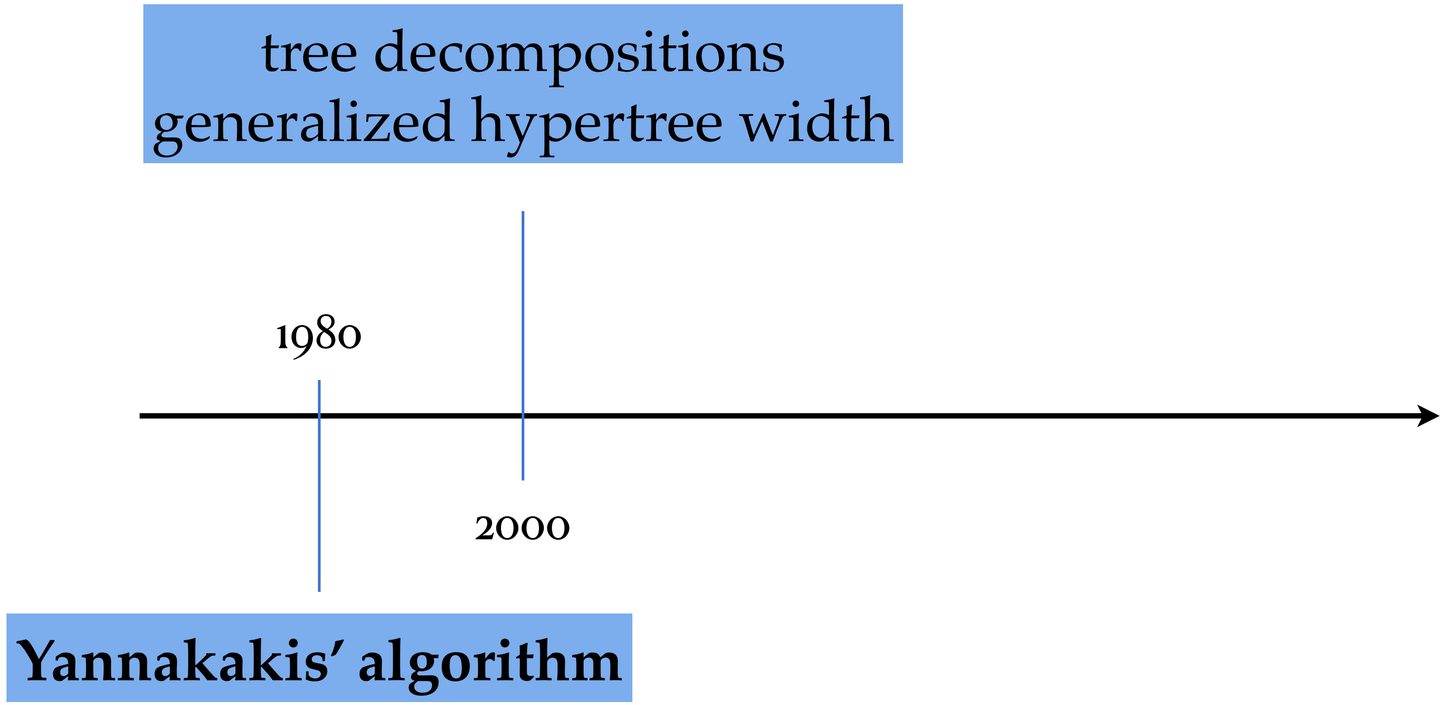
the story about the progress and a few key ideas behind it

Roadmap

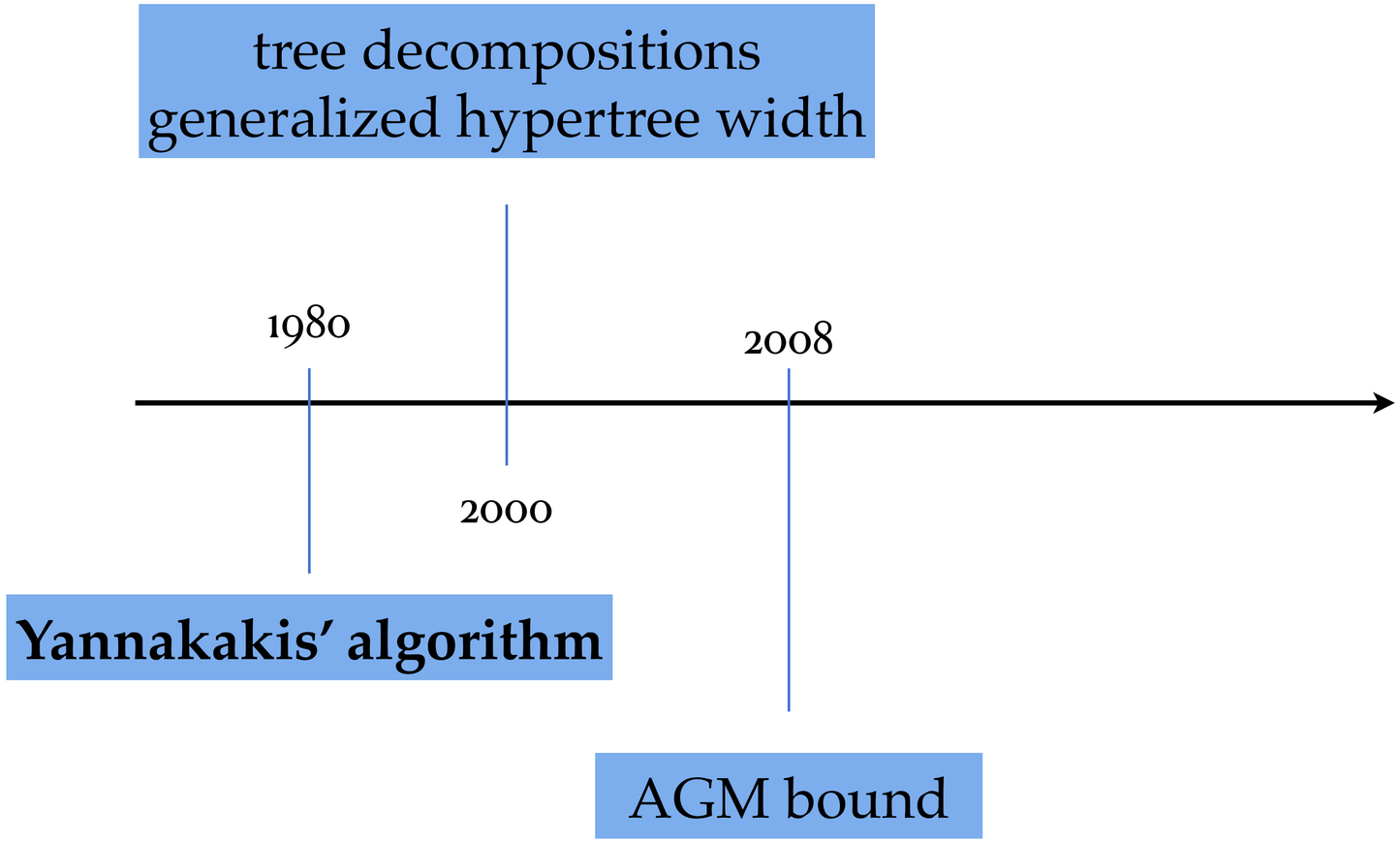
Roadmap



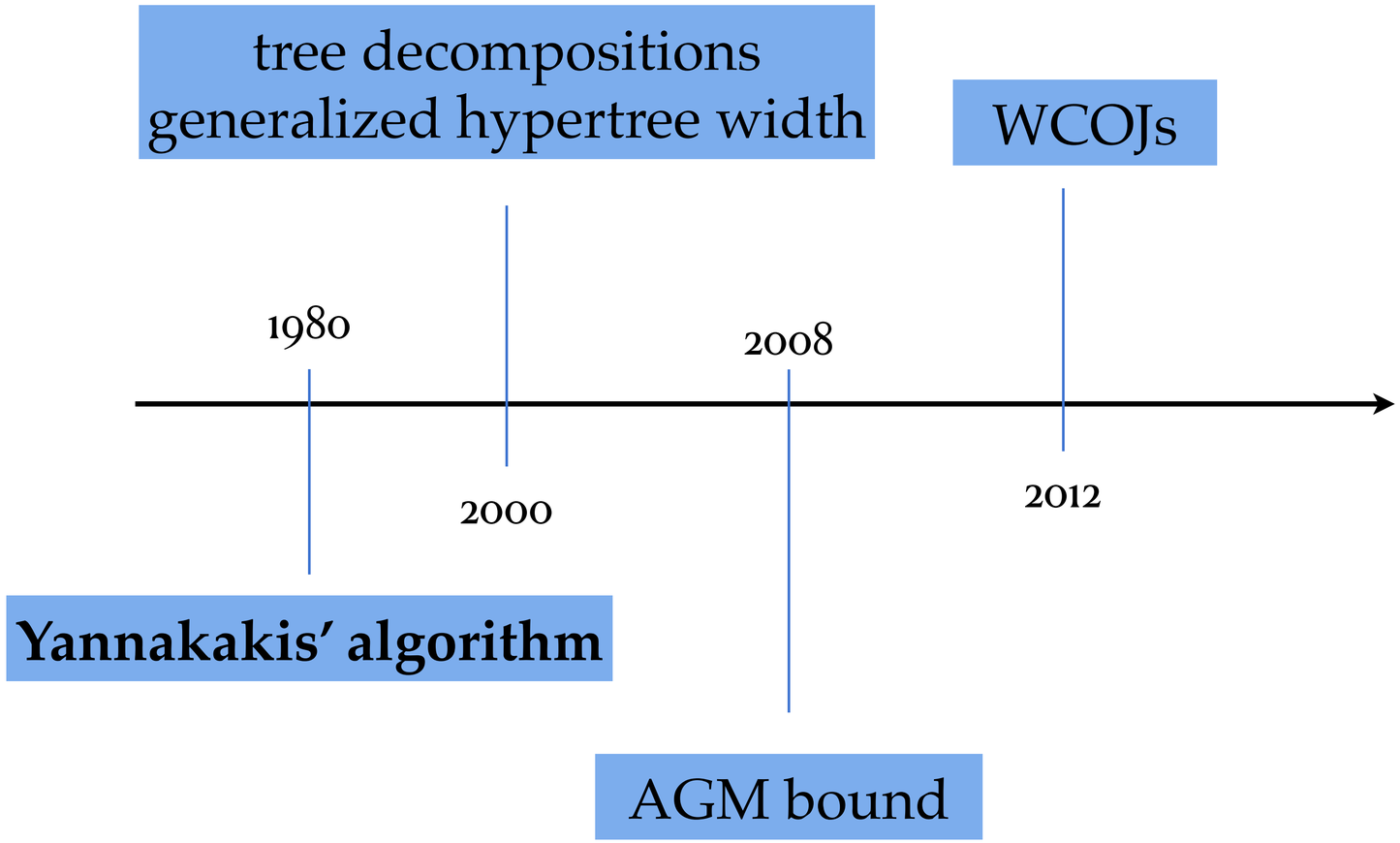
Roadmap


tree decompositions
generalized hypertree width
1980
2000
Yannakakis' algorithm

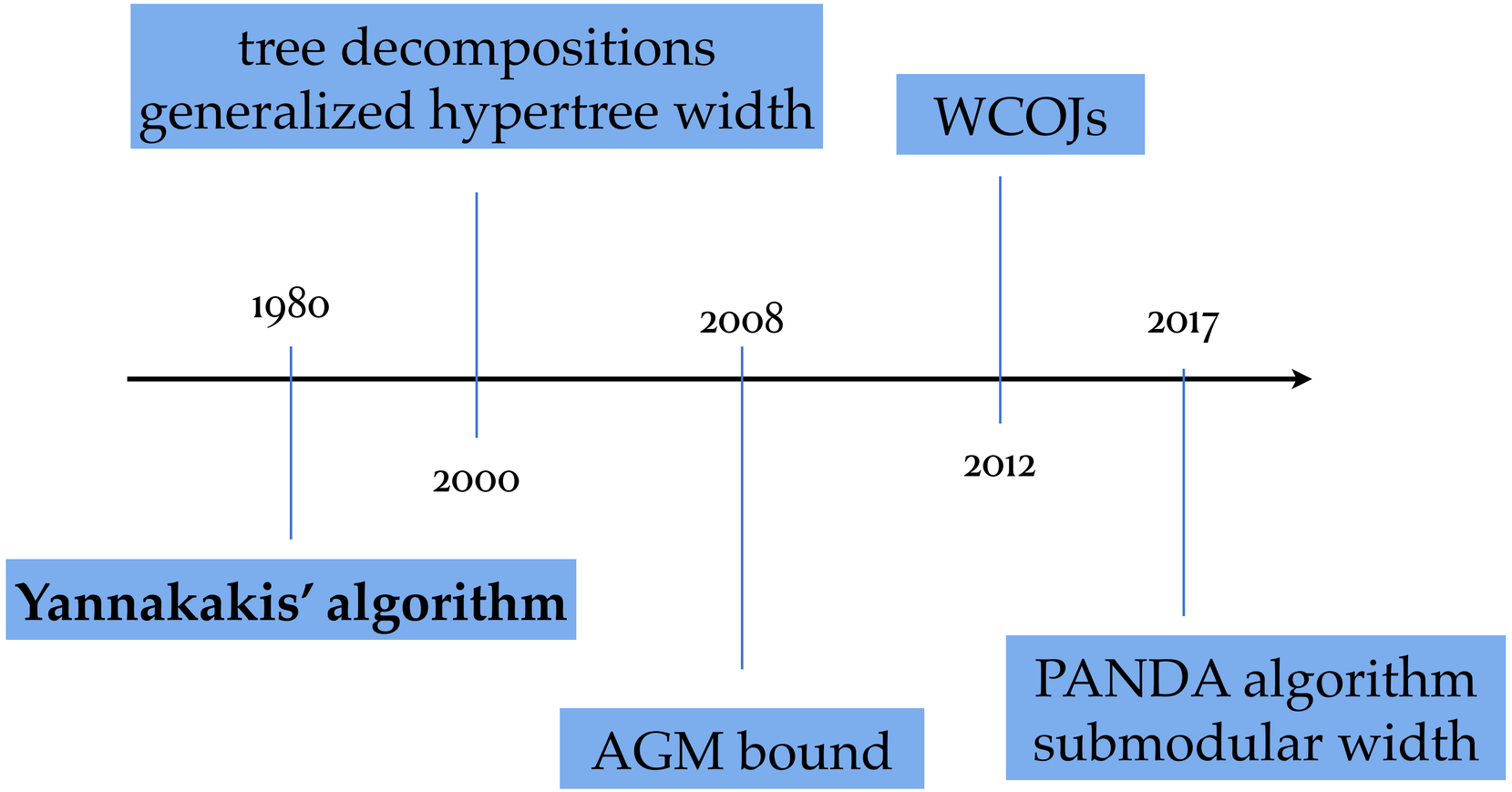
Roadmap


tree decompositions
generalized hypertree width
1980
2000
2008
Yannakakis' algorithm
AGM bound

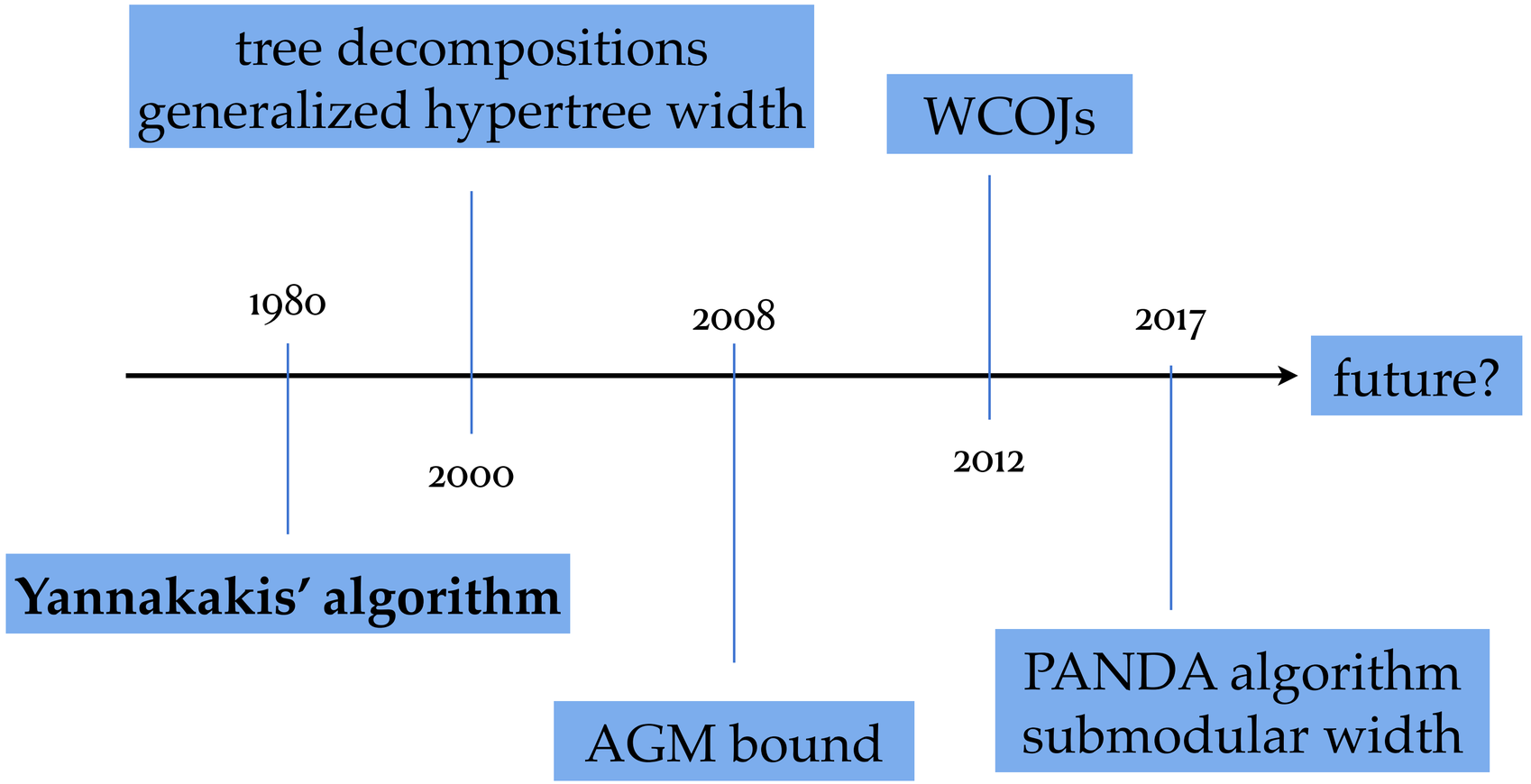
Roadmap


tree decompositions
generalized hypertree width
WCOJs
1980
2000
2008
2012
Yannakakis' algorithm
AGM bound

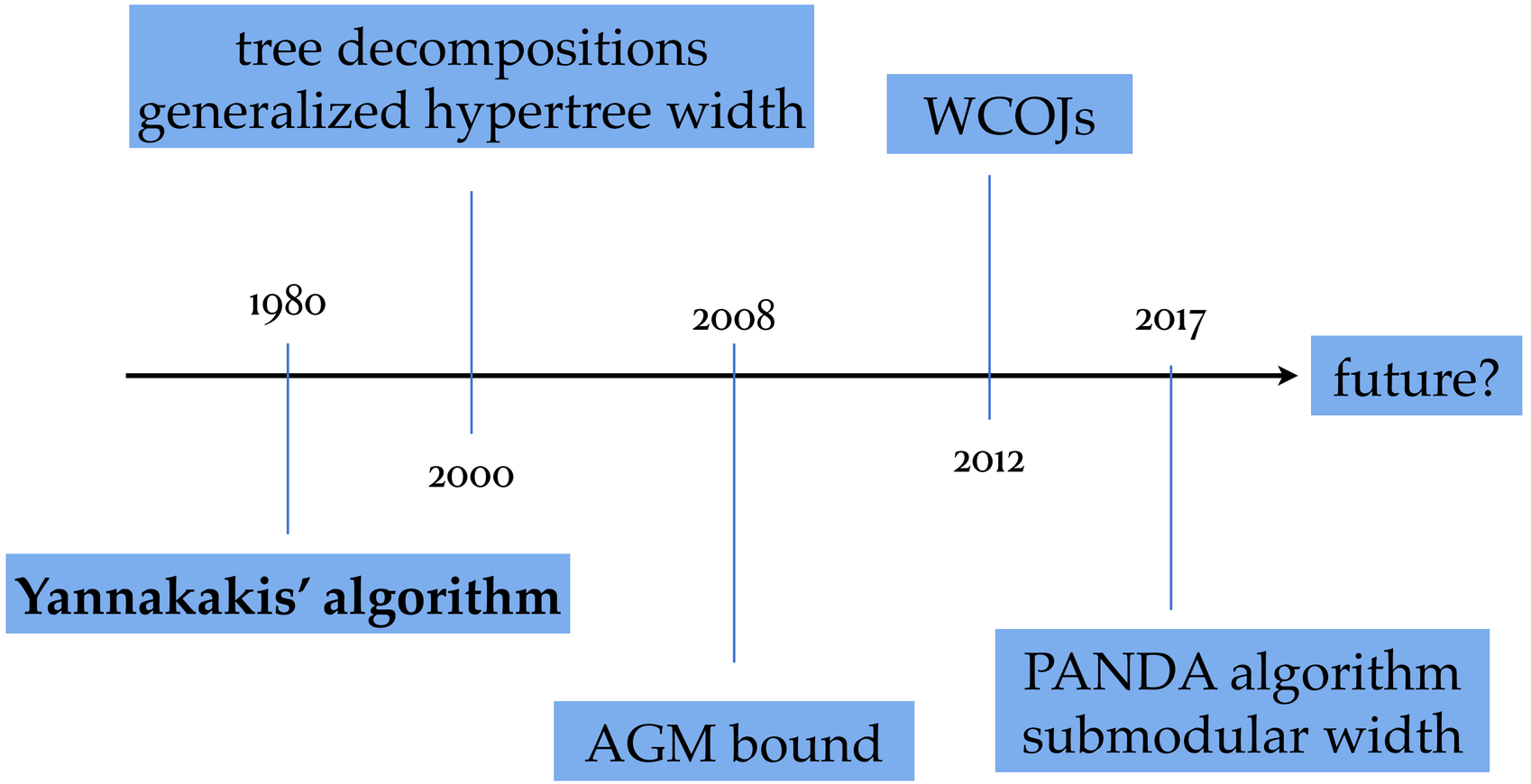
Roadmap


tree decompositions
generalized hypertree width
WCOJs
1980
2000
2008
2012
2017
Yannakakis' algorithm
AGM bound
PANDA algorithm
submodular width

Roadmap


tree decompositions
generalized hypertree width
WCOJs
1980
2000
2008
2012
2017
future?
Yannakakis' algorithm
AGM bound
PANDA algorithm
submodular width

Roadmap



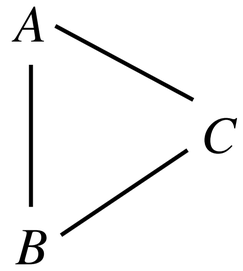
5 Key Ideas:

1. tree decompositions
2. data partitioning
3. using statistics
4. enumeration
5. the algebraic lens

Some background

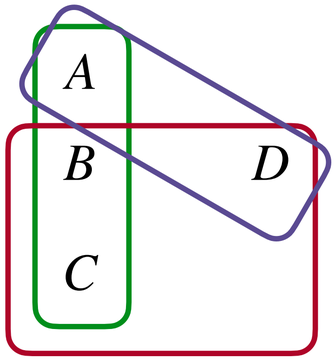
this talk: queries with **natural joins** + **projections**

$\pi_{\emptyset}(R(A,B) \bowtie S(B,C) \bowtie T(C,A))$



Boolean: project out everything

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



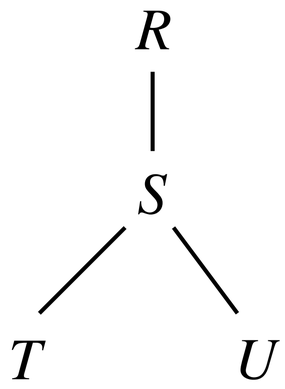
Full: no projections

Yannakakis' Algorithm

[Yannakakis - '81]

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

join tree

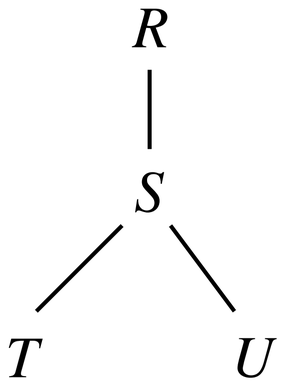


Yannakakis' Algorithm

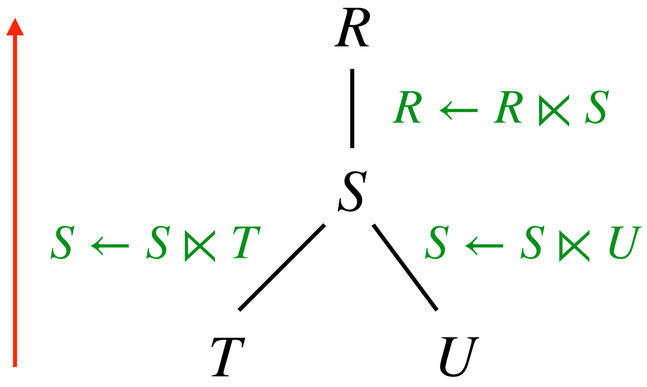
[Yannakakis - '81]

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

join tree



1 - semijoin phase

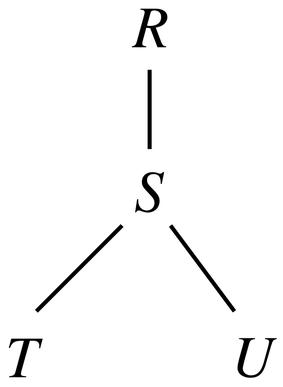


Yannakakis' Algorithm

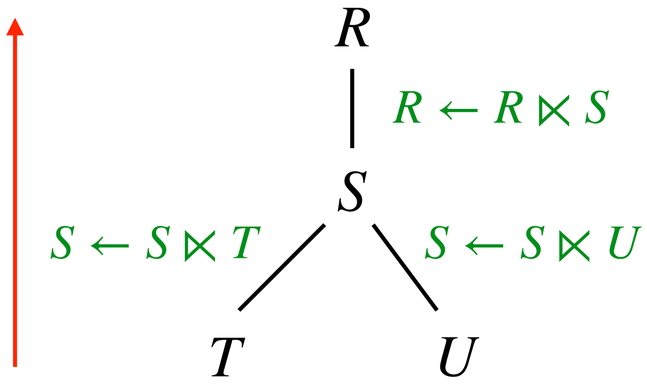
[Yannakakis - '81]

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

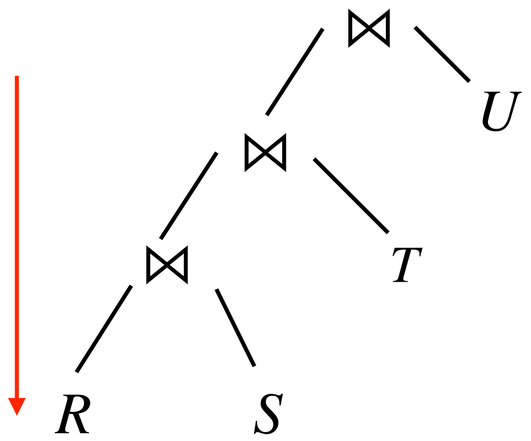
join tree



1 - semijoin phase



2 - join phase



Key property: any intermediate join size is bounded by the output size

Yannakakis' Algorithm

[Yannakakis - '81]

joins that have a join tree

Yannakakis' algorithm comes with runtime guarantees for **acyclic joins**

Full	Boolean	General
$O(N + OUT)$	$O(N)$	$O(N \cdot OUT + OUT)$

input output

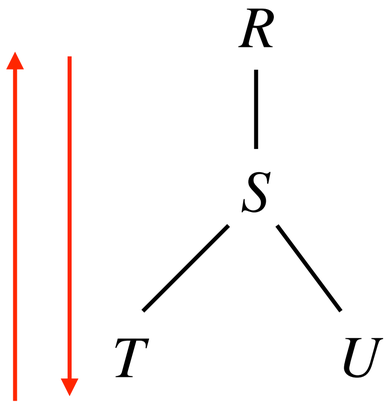
Predicate Transfer: Yannakakis in Practice

- Yannakakis can be slow because of the high cost of a semi-join
- We can replace the semi-joins by (approximate) Bloom filters [Yang et al. - '24]
- We obtain **speedup** + **robustness** to bad query plans

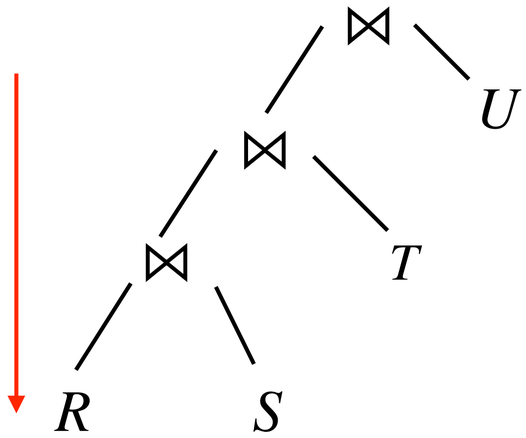
$$R(A,B) \bowtie S(B,C) \bowtie T(C,D) \bowtie U(C,E)$$

1 - transfer phase

forward-backward transfer of Bloom filters



2 - join phase



Predicate Transfer: Yannakakis in Practice

Robustness Factor := max execution time / min execution time

[Zhang et al. - **'25**]

worst join order **best join order**

Predicate Transfer: Yannakakis in Practice

Robustness Factor := max execution time / min execution time

[Zhang et al. - '25]

worst join order **best join order**

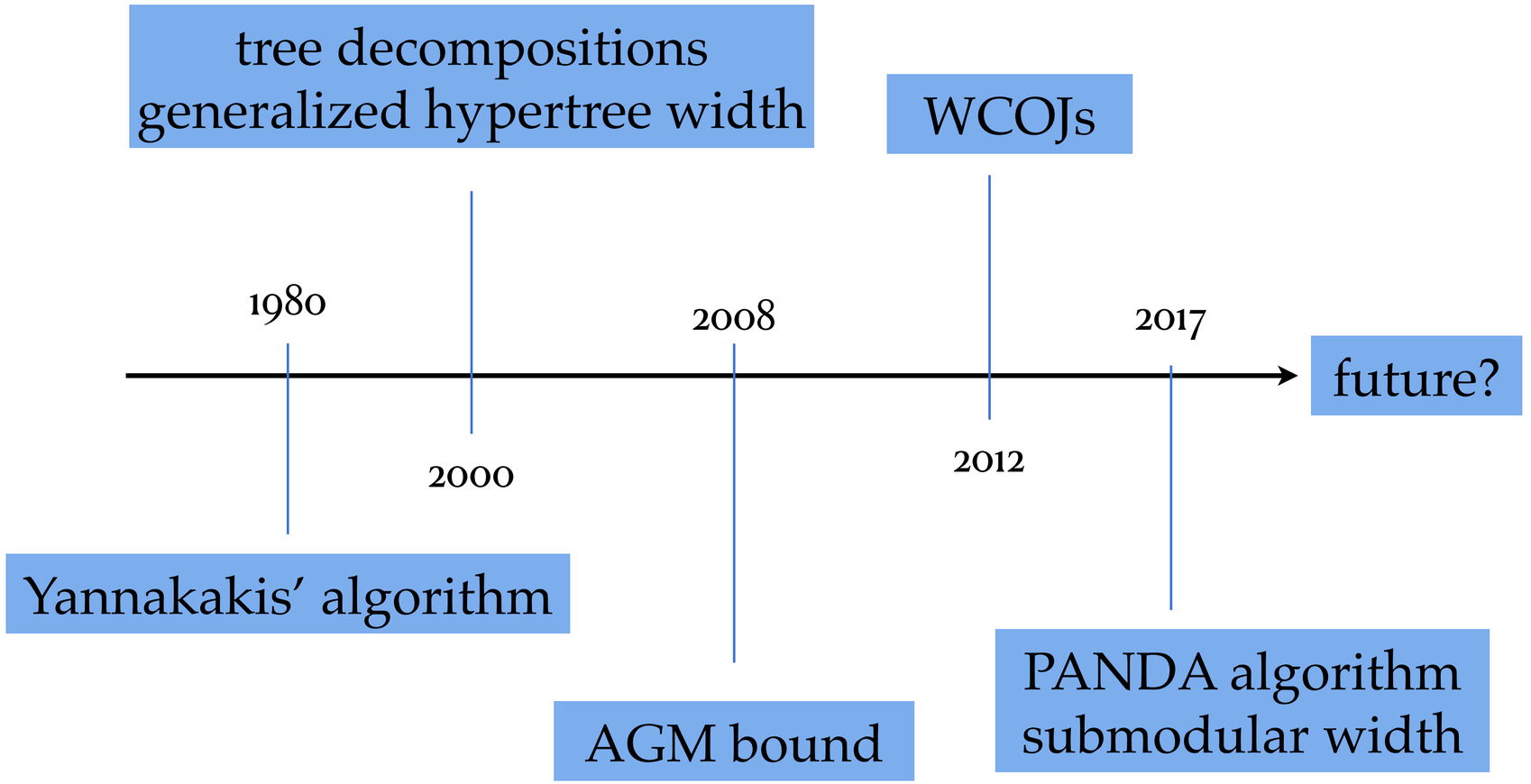
Table 1: Robustness Factors for left-deep joins.

RF	TPC-H			JOB			TPC-DS		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
DuckDB	2.7	1.2	9.3	30.4	1.1	371	7.2	1.0	224
RPT	1.3	1.2	1.5	1.2	1.0	1.6	1.1	1.0	1.5

Table 2: Robustness Factors for bushy joins.

RF	TPC-H			JOB			TPC-DS		
	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max
DuckDB	5.1	1.2	13.7	120	1.1	1747	35.0	1.0	1226
RPT	1.8	1.2	3.0	1.6	1.1	7.7	1.8	1.0	4.2

Roadmap

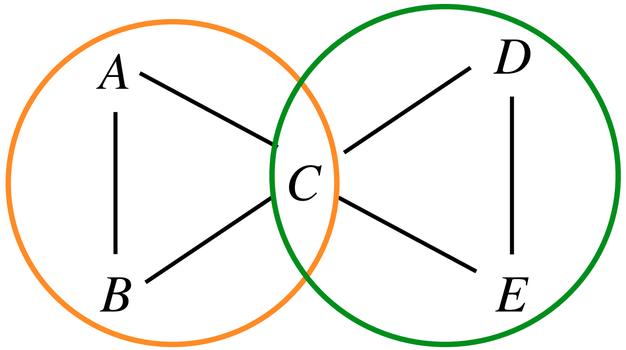


5 Key Ideas:

1. **tree decompositions**
2. data partitioning
3. using statistics
4. enumeration
5. the algebraic lens

1 - Tree Decompositions

- Acyclic joins have plans where the intermediate size is bounded by the output size.
- What can we do for cyclic joins?



$T_1(A, B, C) \bowtie T_2(C, D, E)$

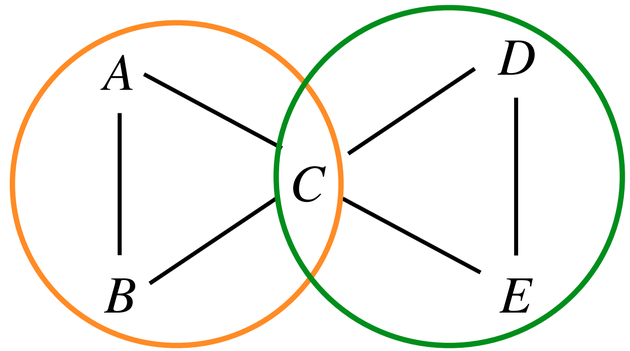
$O(N^2)$ $O(N^2)$

IDEA: make the join acyclic by computing larger intermediate relations

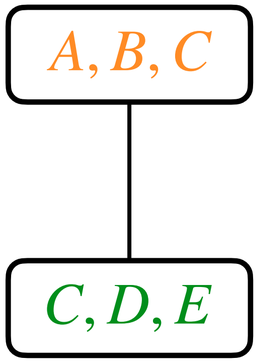
$O(N^2 + OUT)$

Tree Decompositions - Definition

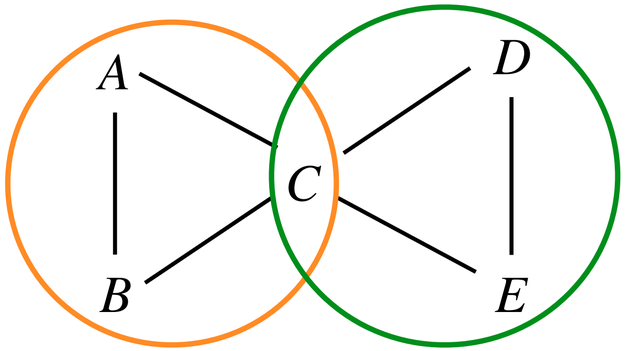
- Like a join tree, but each node is an intermediate relation (**bag**)
- Every relation must be included in some bag



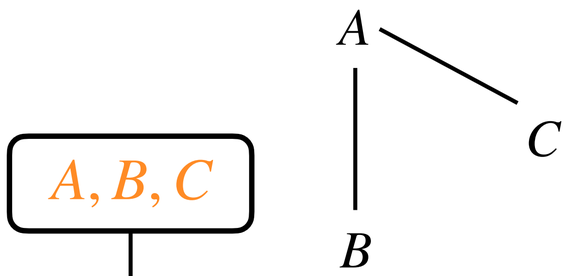
$T_1(A, B, C) \bowtie T_2(C, D, E)$



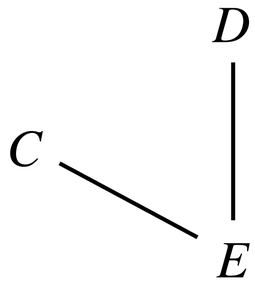
Generalized HyperTree Width



integral edge cover w: smallest subset of relations that "covers" all attributes in a bag

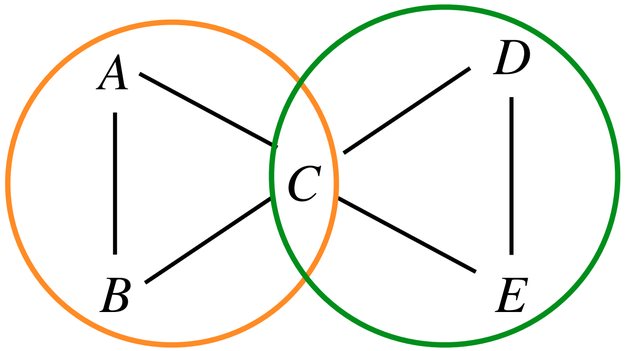


$w = 2$

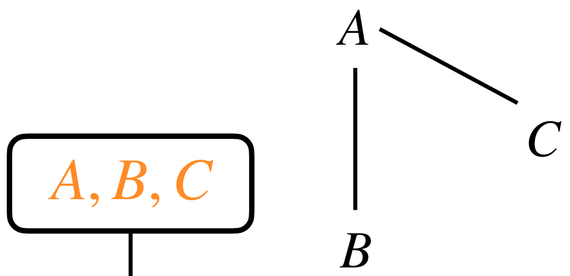


$w = 2$

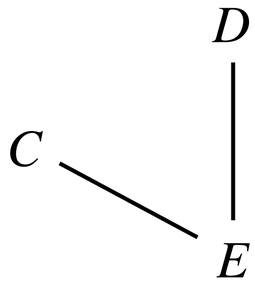
Generalized HyperTree Width



integral edge cover w: smallest subset of relations that "covers" all attributes in a bag



$w = 2$



$w = 2$

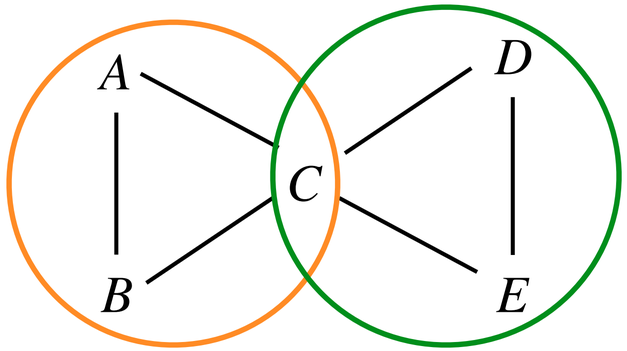
$$\text{ghw}(Q) := \min_{\text{decomp } T} \max_{\text{bag } B} w(X_B)$$

ghw = 2

Runtime Result

If Q is a full join, we can evaluate it in time

$$O\left(N^{ghw(Q)} + OUT\right)$$

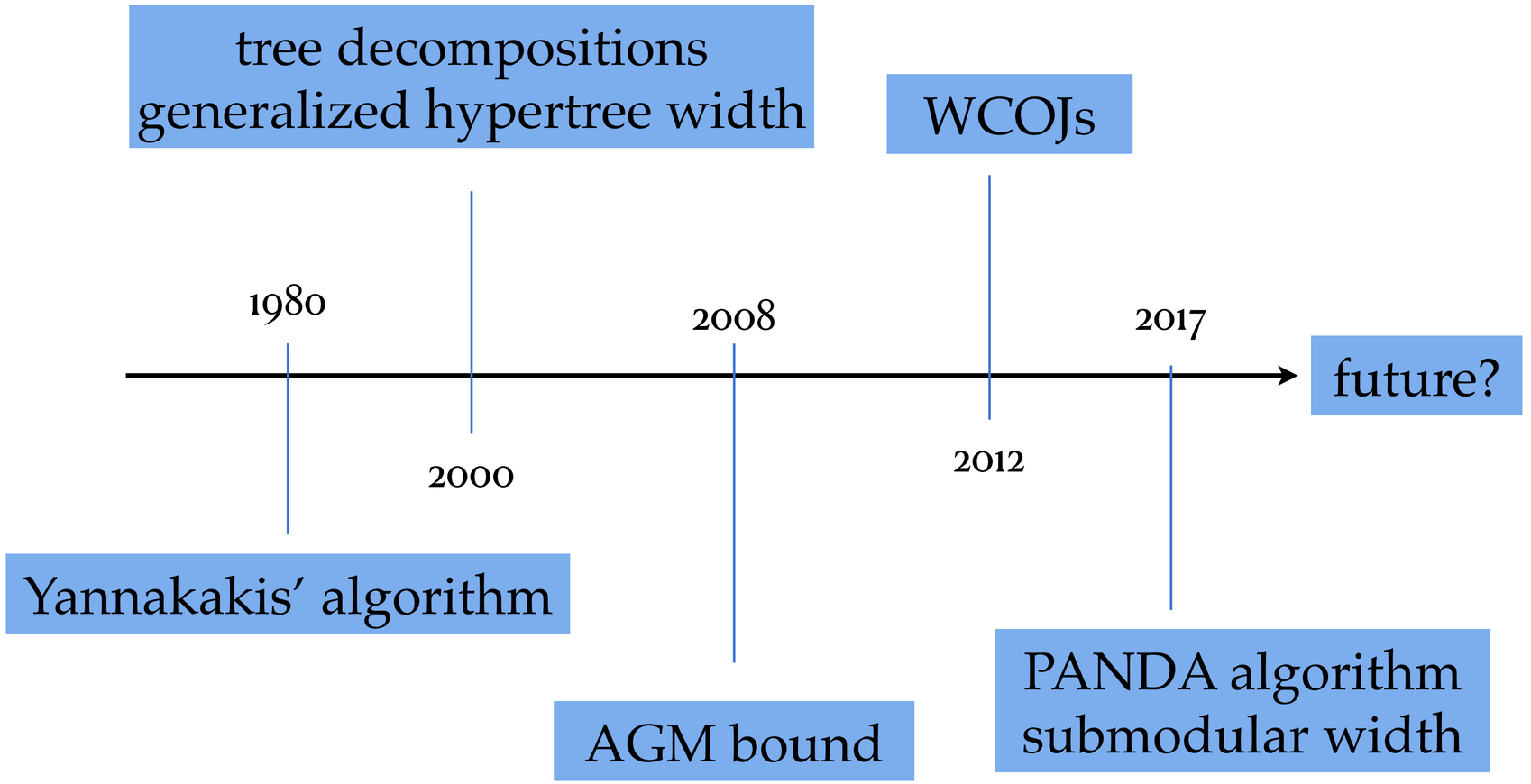


$T_1(A, B, C) \bowtie T_2(C, D, E)$

$O(N^2 + OUT)$

The algorithm can be expressed as a join-project plan!

Roadmap

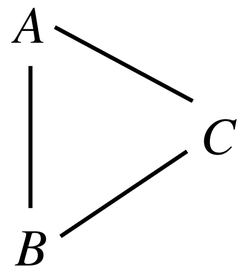


5 Key Ideas:

1. tree decompositions
2. **data partitioning**
3. using statistics
4. enumeration
5. the algebraic lens

2 - Data Partitioning

Join-at-a-time algorithms are **suboptimal!**

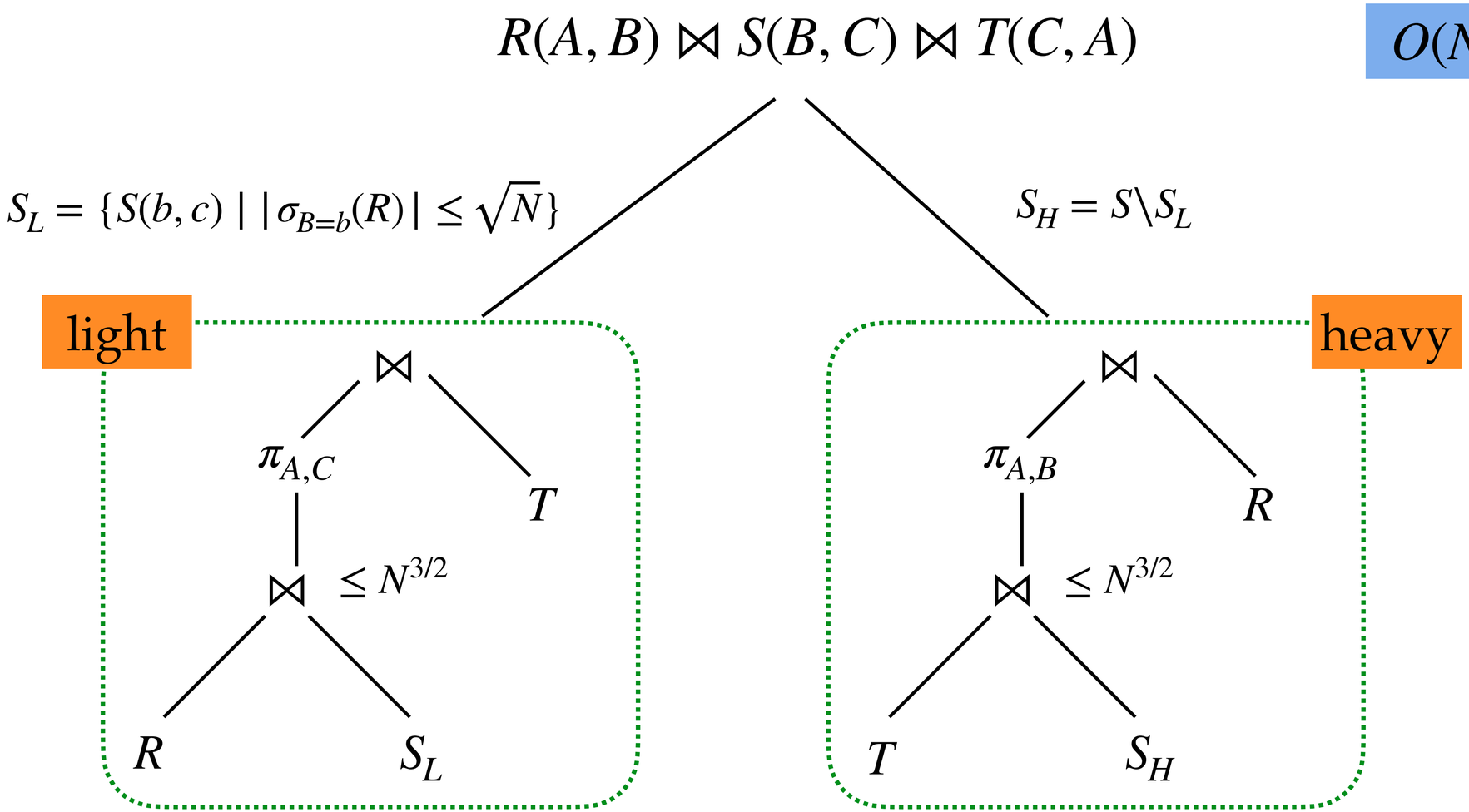


best binary join algorithm: $O(N^2)$

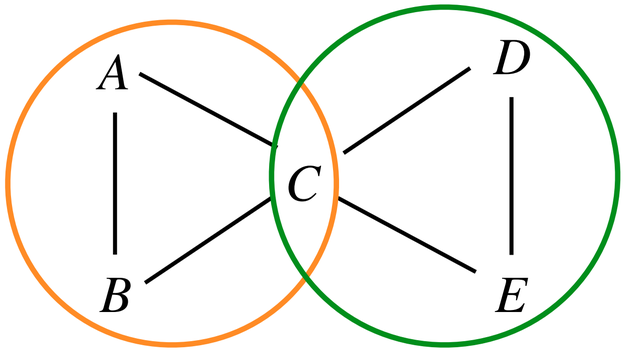
we can do better: $O(N^{3/2})$

We can do better by **partitioning** the input data and applying a different join plan to each partition

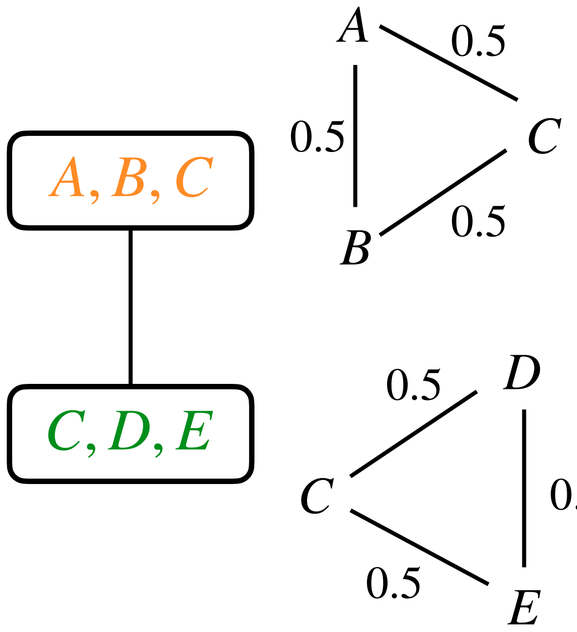
Data Partitioning: Example



Fractional Hypertree Width



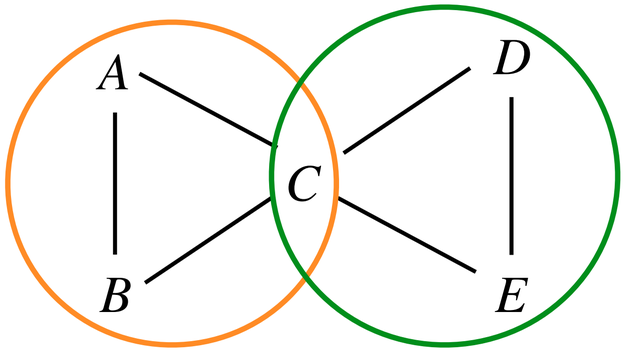
fractional edge cover f: assign weights to relations to "cover" all attributes in a bag



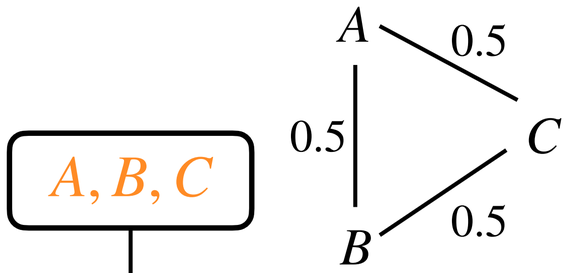
$f = 3/2$

$f = 3/2$

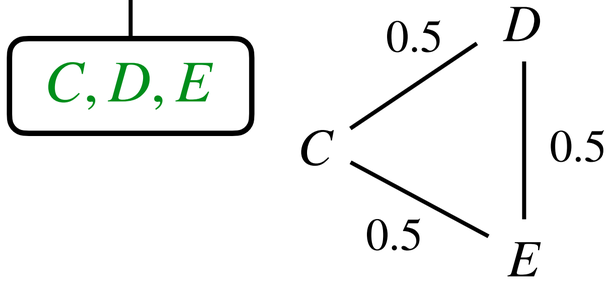
Fractional Hypertree Width



fractional edge cover f: assign weights to relations to "cover" all attributes in a bag



$f = 3/2$



$f = 3/2$

$$\text{fhw}(Q) := \min_{\text{decomp } T} \max_{\text{bag } B} f(X_B)$$

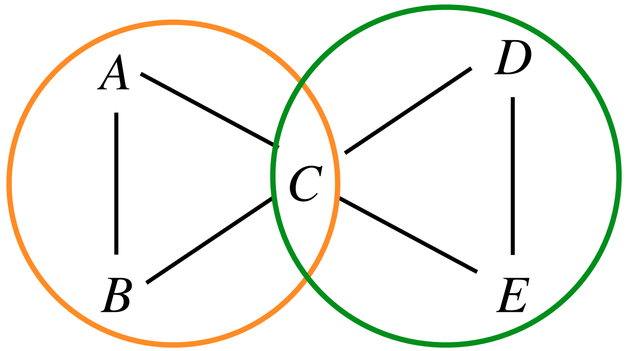
fhw = 3/2

Runtime Result Revisited

If Q is a full join, we can evaluate it in time

$$O\left(N^{fhw(Q)} + OUT\right)$$

[Ngo, Re, Rudra - '13]



$T_1(A, B, C) \bowtie T_2(C, D, E)$ ..... $O(N^{3/2} + OUT)$

The algorithm can be expressed as a join-project-**partition** plan!

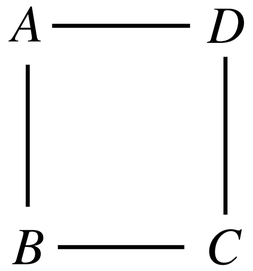
(Worst-case Optimal Joins)

- WCOJs have a running time matching the worst-case output size
- They often work in a attribute-at-a-time manner
  - Leapfrog Triejoin [Veldhuizen - '14]
  - GenericJoin [Ngo, Re, Rudra - '13]
  - FreeJoin [Wang, Willsey, Suciu - '23]

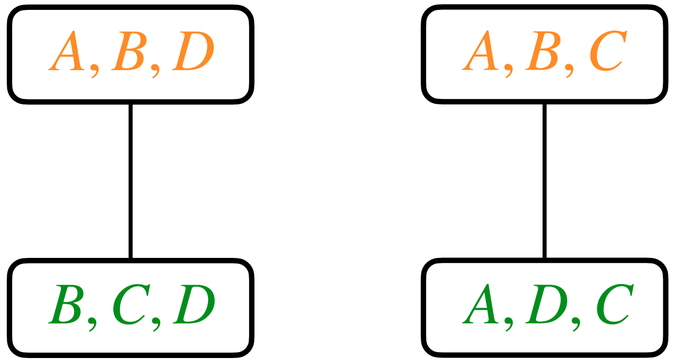
AGM bound [Atserias, Grohe, Marx - '08]

A lot of successful practical implementations!

Data Partitioning Again



We can partition data to use different tree decompositions!



fhw = 2

Runtime Result (Re)Revisited

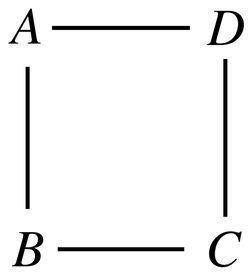
If Q is a full join, we can evaluate it in time

$$\tilde{O}\left(N^{subw(Q)} + OUT\right)$$

[Khamis, Ngo, Suciu - '17]

$$subw(Q) := \max_{\text{submodular } h} \min_{\text{decomp } T} \max_{\text{bag } B} h(X_B)$$

subw ≤ fhw ≤ ghw



$O(N^{3/2} + OUT)$

PANDA algorithm

Output-Sensitive Yannakakis

- When we have projections in an acyclic join, Yannakakis runs in time $O(N + N \cdot OUT)$
- We can design faster join algorithms by taking the output size OUT into account (using **data partitioning** again)

If Q is an acyclic join, we can evaluate it in time

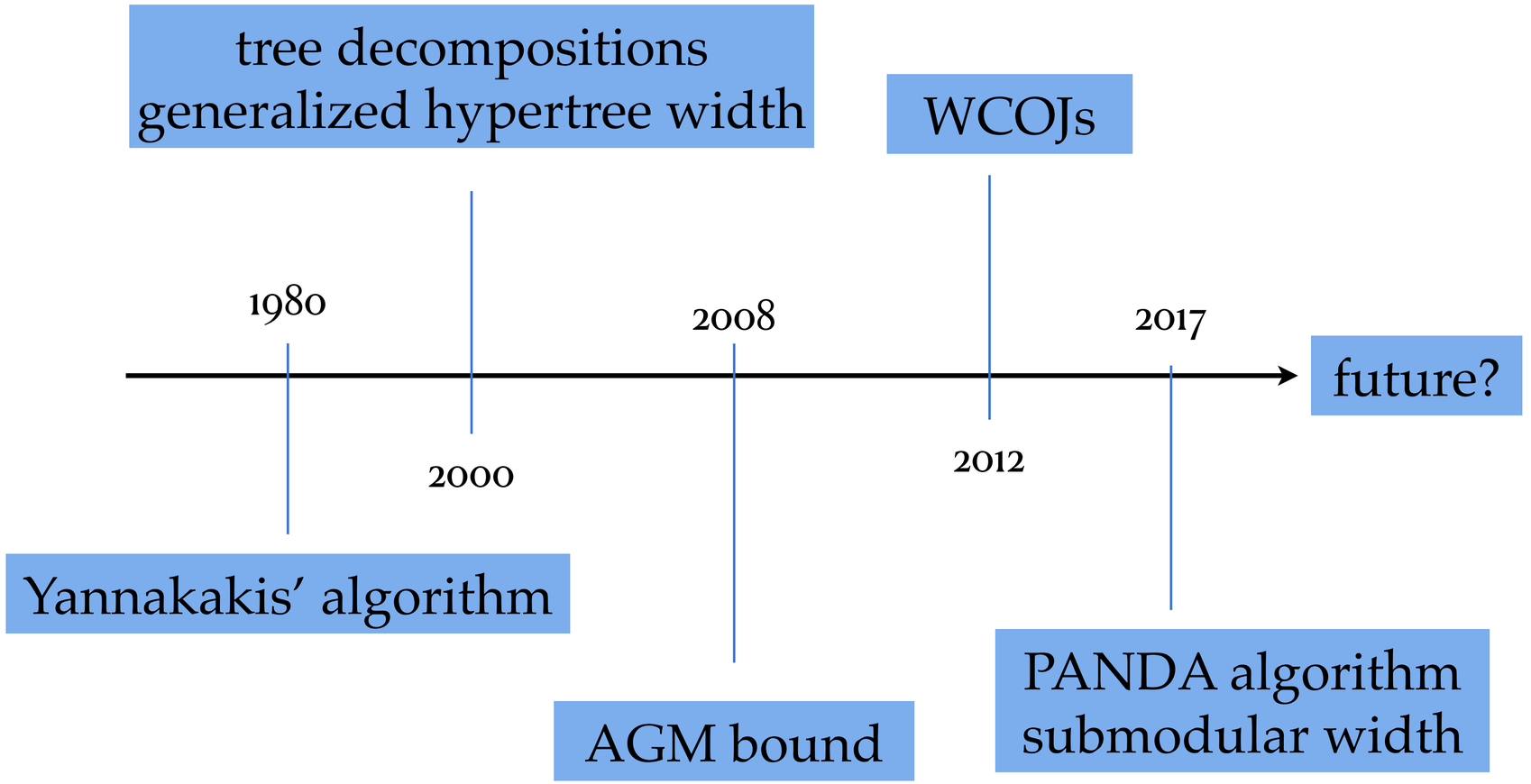
$$O(N + N \cdot OUT^{1-\epsilon})$$

[Deep, Zhao, Fan, Koutris - '**25**]

[Hu - '**25**]

$\pi_{A,C}(R(A,B) \bowtie T(B,C))$ ..... $O(N + N \cdot OUT^{1/2})$

Roadmap

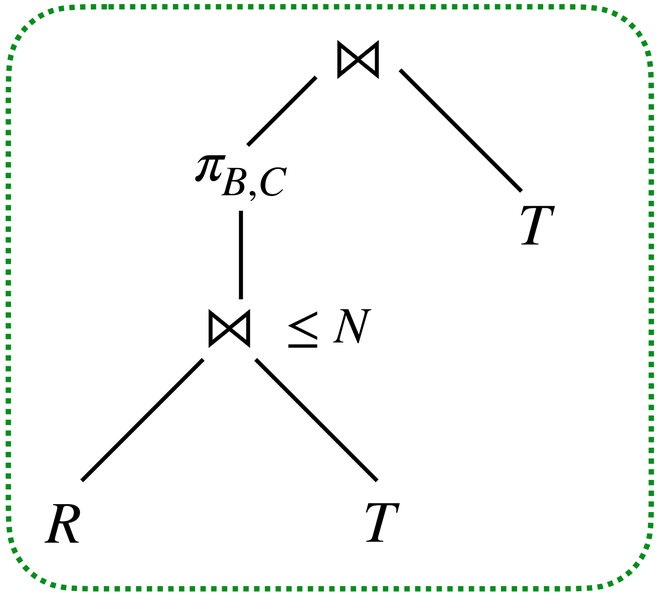


5 Key Ideas:

1. tree decompositions
2. data partitioning
3. **using statistics**
4. enumeration
5. the algebraic lens

3 - Using Statistical Information

By using more **statistical information**, we can design faster join algorithms



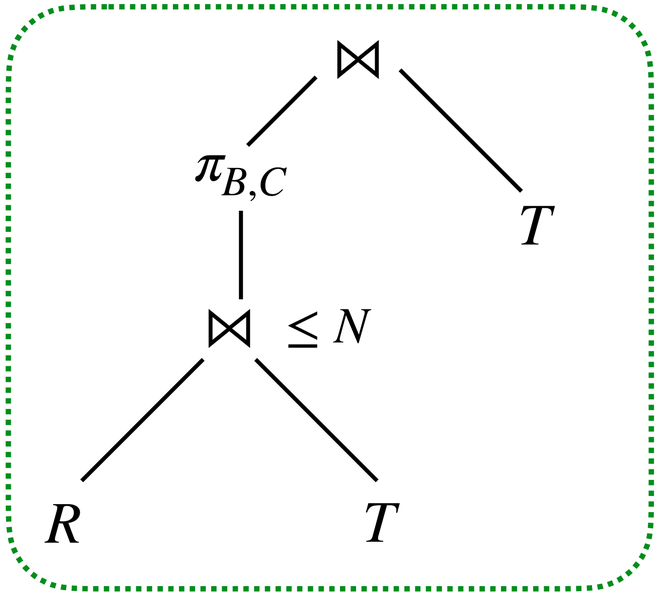
primary key

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

runs in time only $O(N)$

3 - Using Statistical Information

By using more **statistical information**, we can design faster join algorithms



primary key

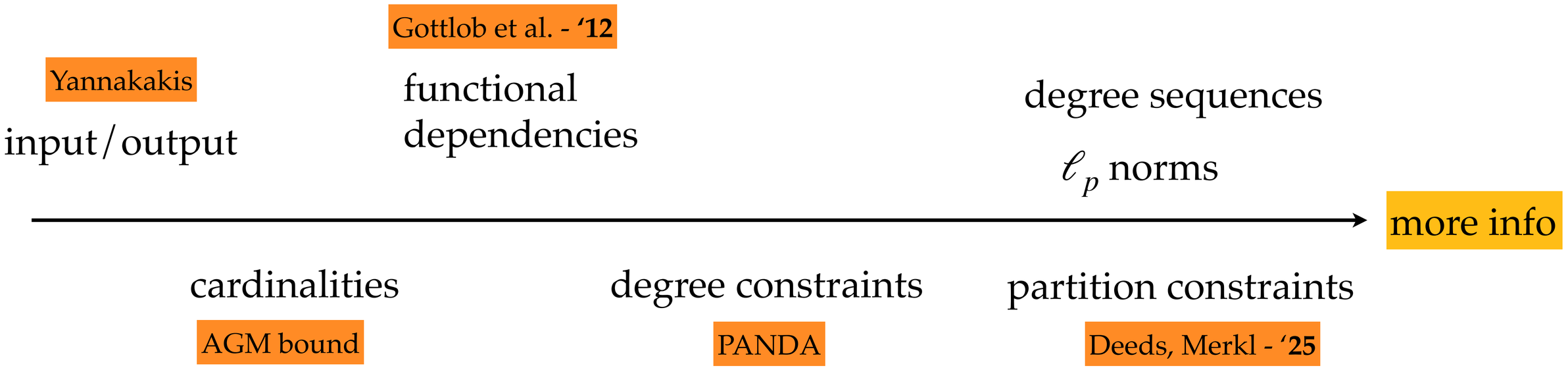
$R(A, B) \bowtie S(B, C) \bowtie T(C, A)$

runs in time only $O(N)$

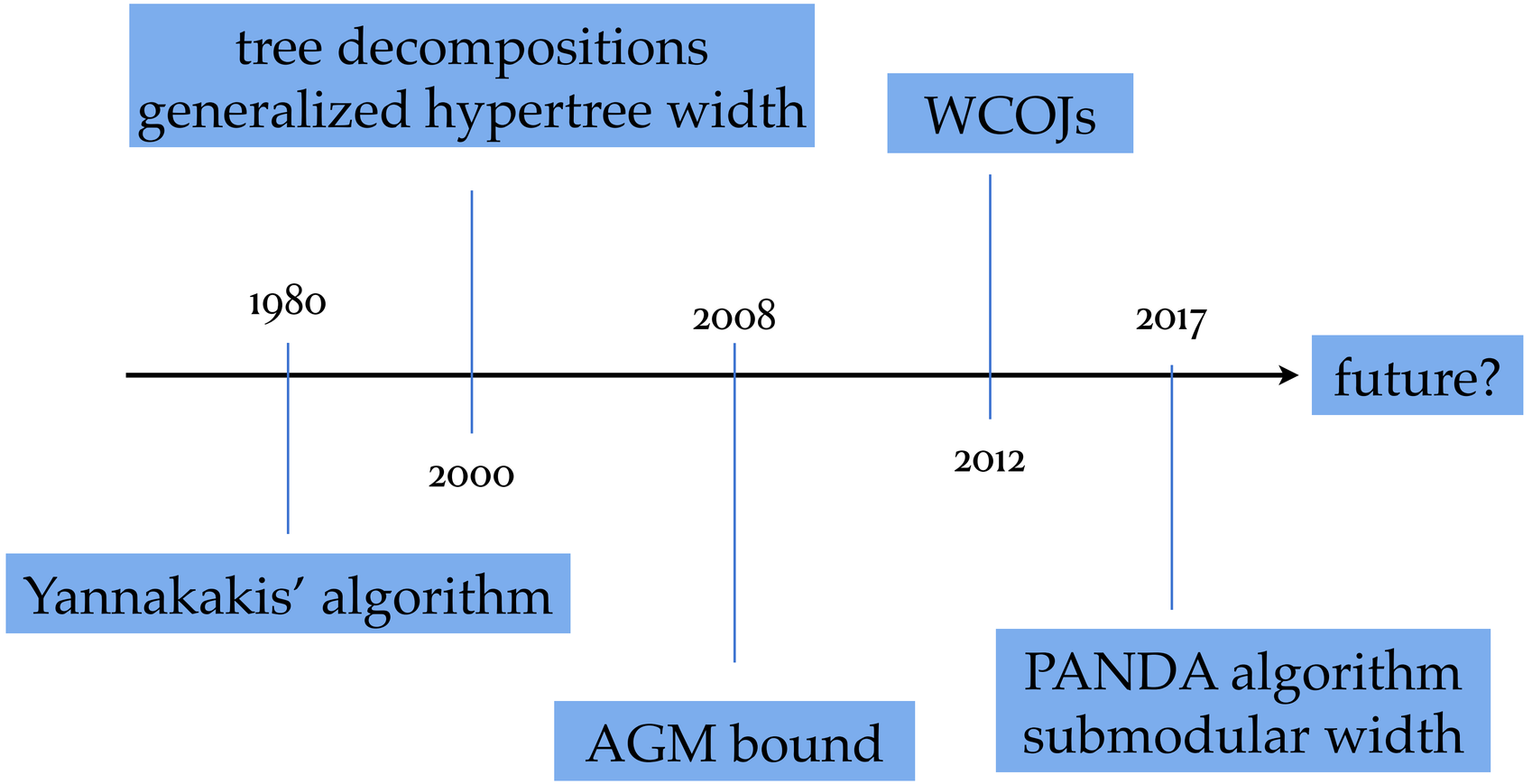
Using more detailed statistics

- restricts what is the worst-case behaved input
- closes the gap between theory and practice

3 - Using Statistical Information



Roadmap



5 Key Ideas:

1. tree decompositions
2. data partitioning
3. using statistics
4. **enumeration**
5. the algebraic lens

4 - Enumeration

We can design join algorithms with stronger **enumeration** guarantees

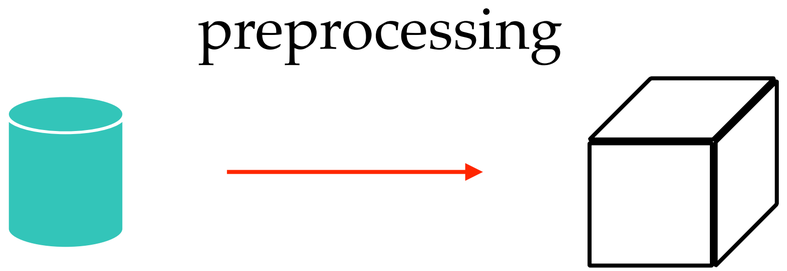
- useful for LIMIT queries
- constant delay enumeration [Bagan et al. - '07]

$O(N + OUT)$ vs After $O(N)$ preprocessing time, we can enumerate the output while spending only constant time between two consecutive outputs

4 - Enumeration

We can design join algorithms with stronger **enumeration** guarantees

- useful for LIMIT queries
- constant delay enumeration [Bagan et al. - '07]

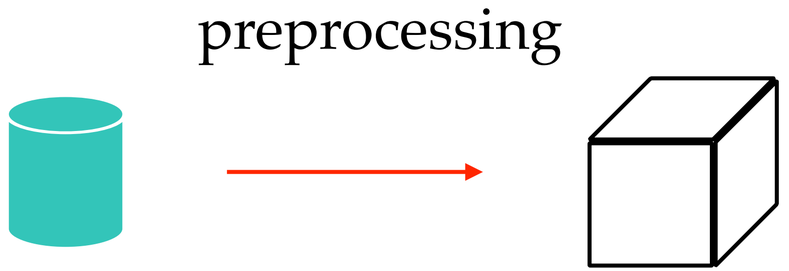


$O(N + OUT)$ vs After $O(N)$ preprocessing time, we can enumerate the output while spending only constant time between two consecutive outputs

4 - Enumeration

We can design join algorithms with stronger **enumeration** guarantees

- useful for LIMIT queries
- constant delay enumeration [Bagan et al. - '07]

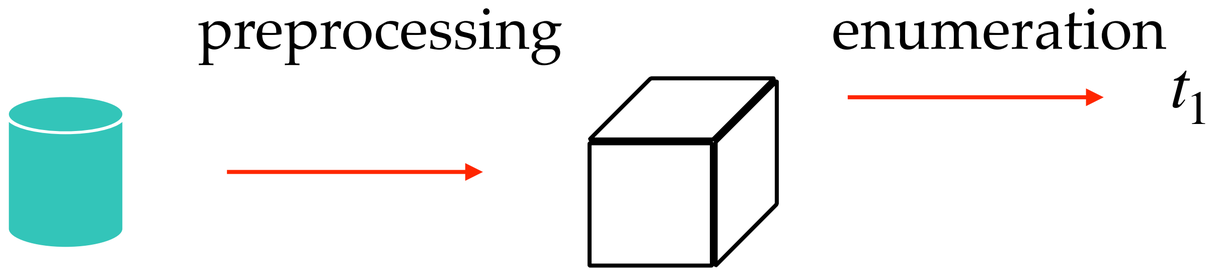


$O(N + OUT)$ vs After $O(N)$ preprocessing time, we can enumerate the output while spending only constant time between two consecutive outputs

4 - Enumeration

We can design join algorithms with stronger **enumeration** guarantees

- useful for LIMIT queries
- constant delay enumeration [Bagan et al. - '07]

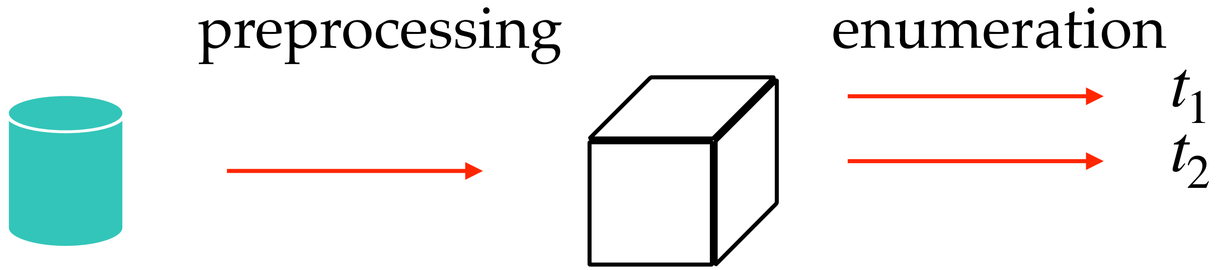


$O(N + OUT)$ vs After $O(N)$ preprocessing time, we can enumerate the output while spending only constant time between two consecutive outputs

4 - Enumeration

We can design join algorithms with stronger **enumeration** guarantees

- useful for LIMIT queries
- constant delay enumeration [Bagan et al. - '07]



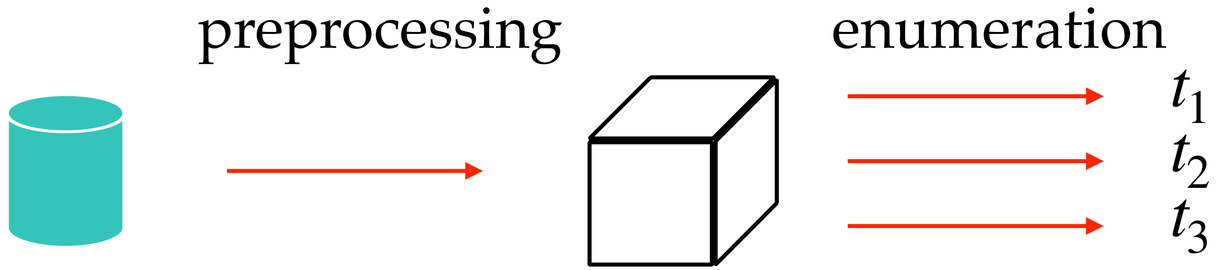
$O(N + OUT)$ vs

After $O(N)$ preprocessing time, we can enumerate the output while spending only constant time between two consecutive outputs

4 - Enumeration

We can design join algorithms with stronger **enumeration** guarantees

- useful for LIMIT queries
- constant delay enumeration [Bagan et al. - '07]



$O(N + OUT)$ vs

After $O(N)$ preprocessing time, we can enumerate the output while spending only constant time between two consecutive outputs

Runtime Result Revisited

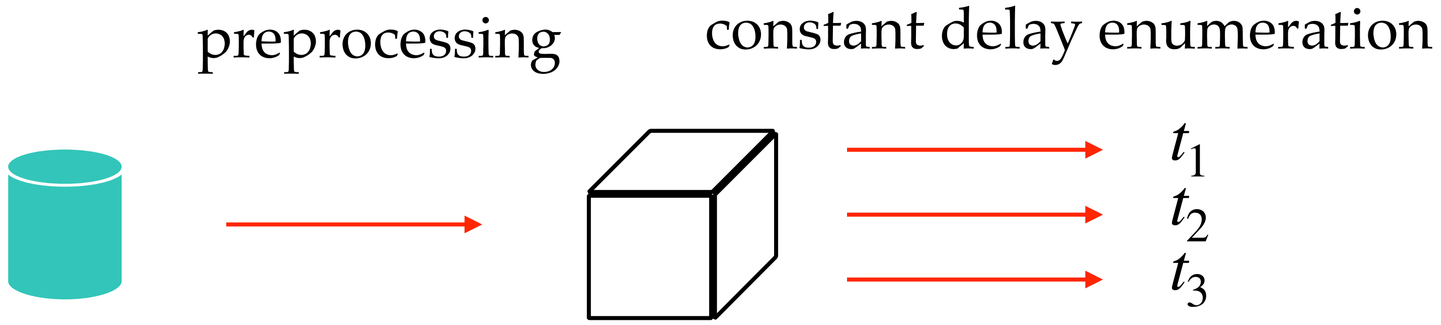
If Q is a full join, we can do constant delay enumeration with preprocessing time

$$\tilde{O}\left(N^{subw(Q)}\right)$$

Acyclic queries only need linear preprocessing time!

Factorized Databases

[Olteanu,Zavodny - '16]



The intermediate data structure can be viewed as a compressed (*factorized*) representation of the join output that can be decompressed efficiently

Joins with Ranked Order

1. Joins where we only want top-ranked results (ORDER BY + LIMIT)
2. We can construct algorithms with strong enumeration guarantees
3. Works well in practice - it avoids intermediate *expensive materializations*

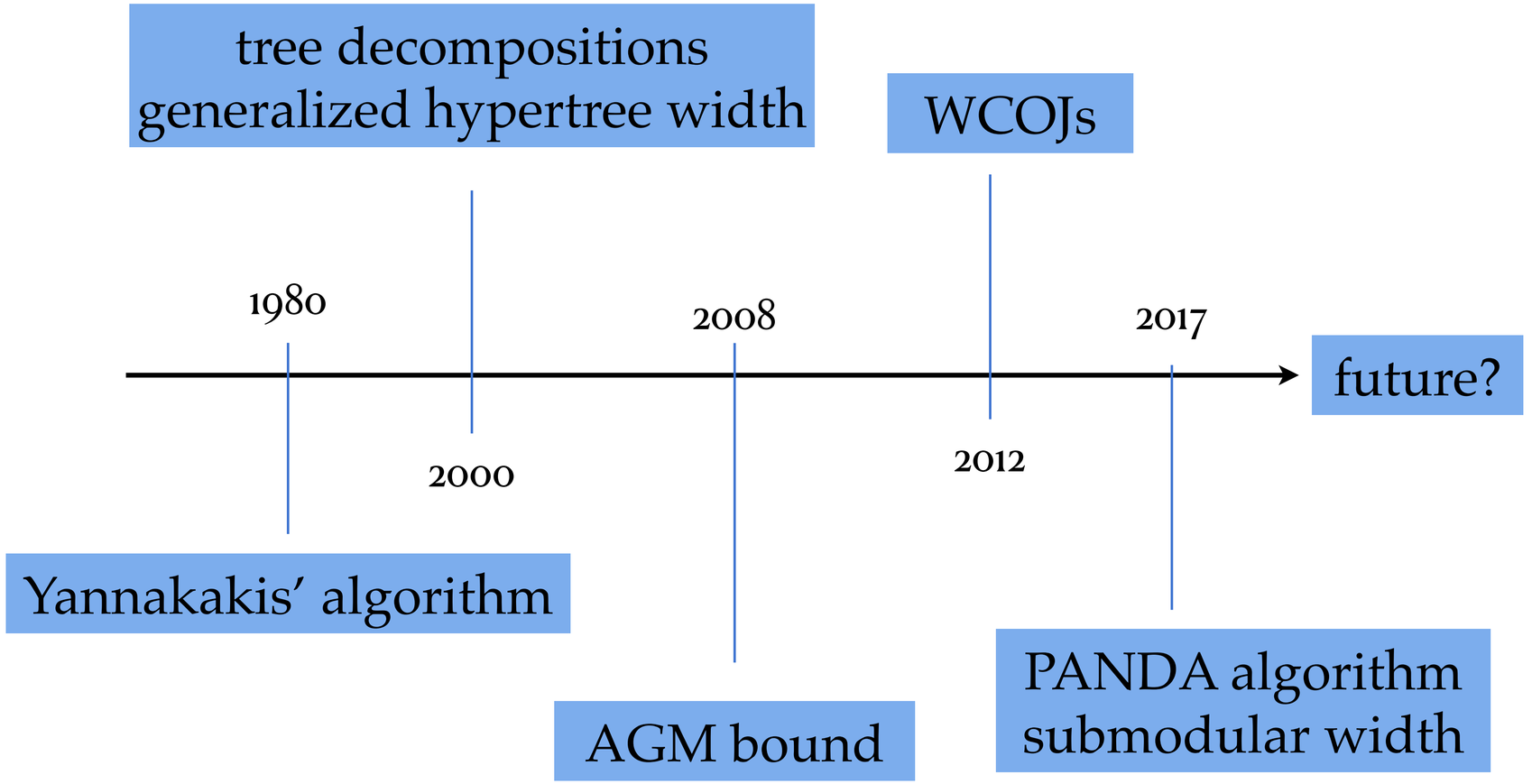
```
SELECT *
FROM R, S
WHERE R.A = S.B
ORDER BY S.C + R.D
LIMIT 10
```

For acyclic joins, after $O(N)$ preprocessing time, we can enumerate the output while spending only **logarithmic** time between two consecutive outputs

[Deep, Koutris - '**21**]

[Tziavelis et al. - '**20**]

Roadmap



5 Key Ideas:

1. tree decompositions
2. data partitioning
3. using statistics
4. enumeration
5. **the algebraic lens**

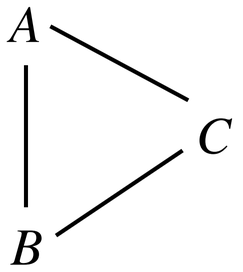
5 - The Algebraic Lens

Most existing join algorithms can be *lifted* to work on general algebraic structures called **semirings** [Green, Karvounarakis, Tannen - '**21**]

- aggregation becomes simple - sum, count, min, max
- extremely useful for ML pipelines (FAQs, factorized databases)

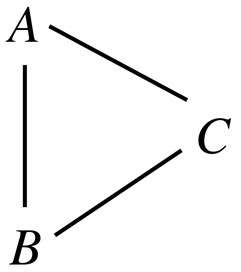
- join: **multiplication** ($\times$)
- project, union: **addition** ($+$)

Triangles with Semirings



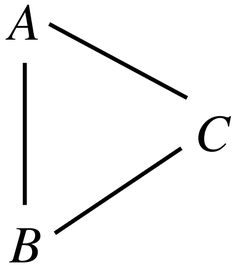
$(\{0,1\}, \vee , \wedge ,0,1)$

\+ **Boolean semiring** = is there is a triangle?



$(\mathbb{N}, \min , + ,\infty,0)$

\+ **tropical semiring** = minimum weight triangle



$(\mathbb{N}, + , \cdot ,0,1)$

\+ **arithmetic semiring** = how many triangles?

The Algebraic Lens

If Q is an acyclic full join, we can compute it over any semiring it in time

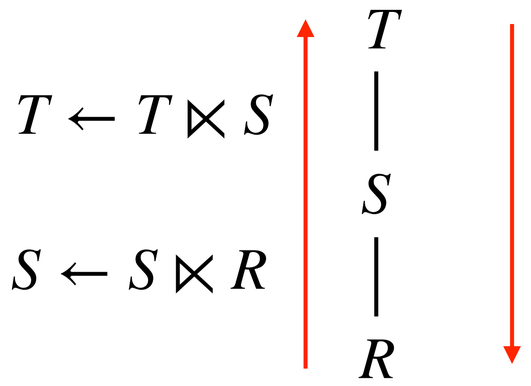
$$O(N + OUT)$$

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

and many other results…

FAQs [Khamis, Ngo, Rudra - '**16**]

[Khamis et al. - '**19**]



1. semijoin

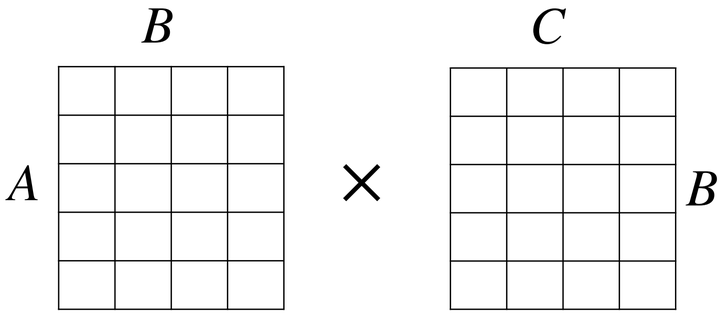
2. join + (partial) aggregation

What comes next?

Towards Even Faster Joins

We can join faster using **Fast Matrix Multiplication (FMM)**

$$\pi_{A,C}(R(A,B) \bowtie S(B,C))$$



- combinatorial: $O(n^3)$
- FMM: $O(n^\omega)$, $\omega = 2.371339$

$$\pi_{\emptyset}(R(A,B) \bowtie S(B,C) \bowtie T(C,A))$$

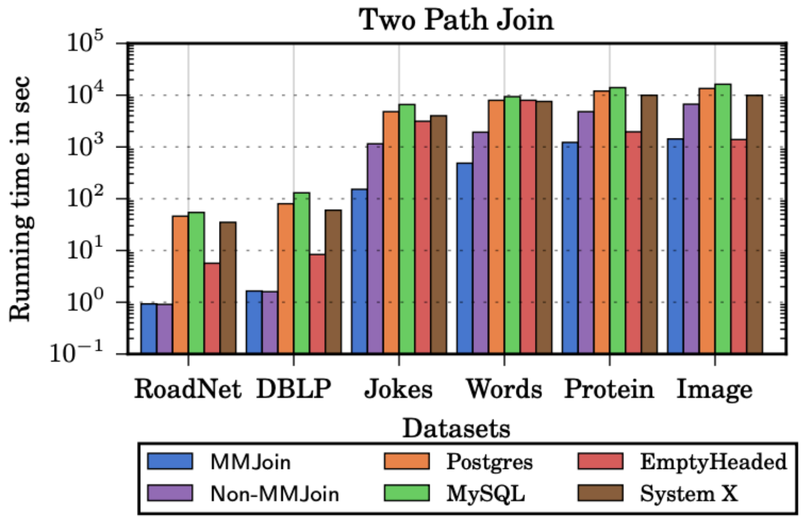
combinatorial: $O(N^{3/2})$
with FMM: $O(N^{\frac{2\omega}{1+\omega}})$

[Alon, Yuster, Zwick - '97]

Towards Even Faster Joins

- FMM can be incorporated as a new operator in a query plan [Khamis, Hu, Suciu - '25]
- Using MM can also help in practice

[Deep, Hu, Koutris - '22]



Q1: Are there other ways to use FMM in query plans?

Q2: What other non-combinatorial techniques can we use?

Joins for any Semiring - how do we count?

- The PANDA algorithm *cannot count*!
- The key problem is the overlap of output tuples in tree decompositions
- We can count the join size in time $O(N^{\#\text{subw}})$ [Khamis et al. - '**19**]

$$\text{subw} \leq \#\text{subw} \leq \text{fhw}$$

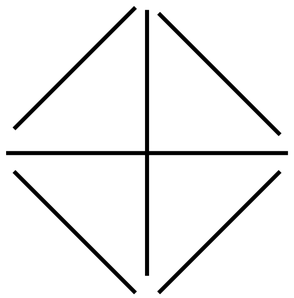
Q3: What is the best algorithm that counts the size of a join?

Lower Bounds for Joins

- To show that join algorithms are optimal, we need lower bounds
- Not (currently) feasible to show unconditional bounds
- We use *conditional lower bounds* by reducing from problems where we believe we have an optimal algorithm

k-clique problem on n vertices

$\Omega(n^k)$



(conditional) lower bounds for all joins

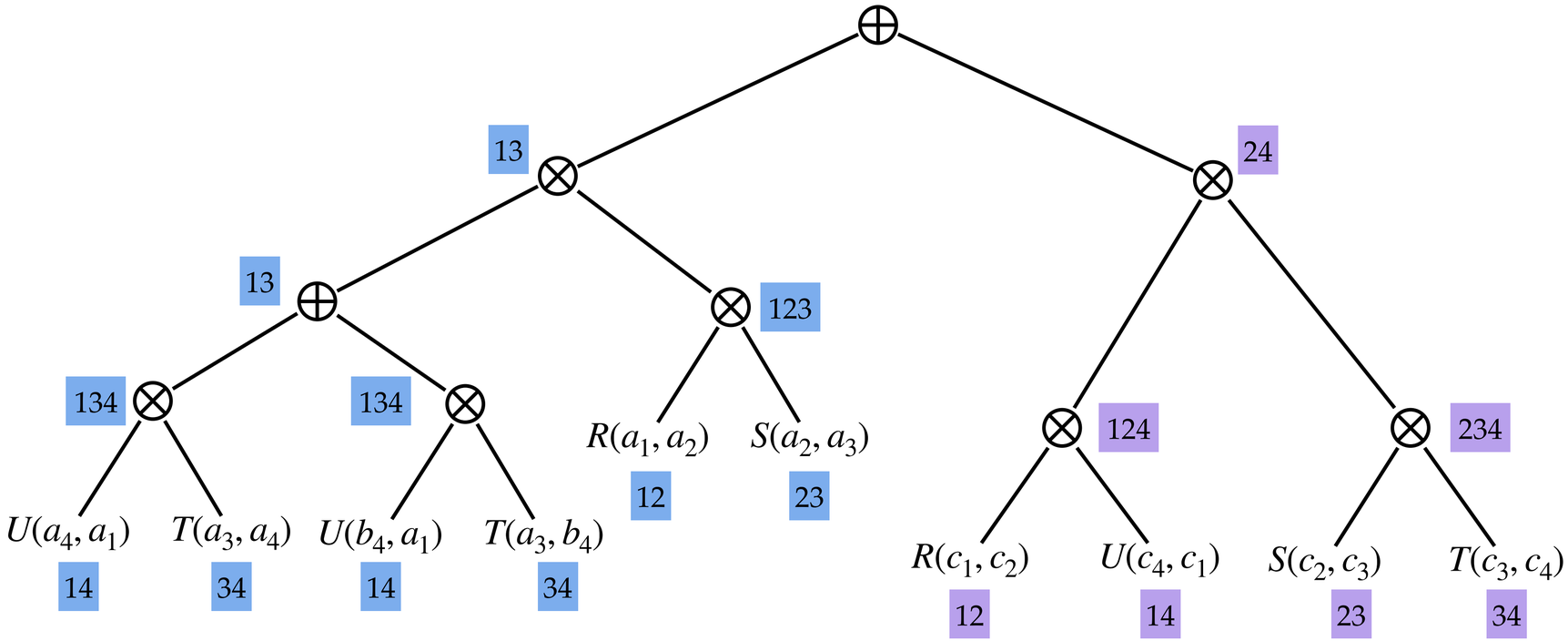
$\Omega(N^{clemb})$ [Fan, Koutris, Zhao - '23]

clemb ≤ subw ≤ fhw

Q4: What other conditional lower bounds should we use?

Lower Bounds for Joins

- Another approach is to restrict the algorithm
- In particular, we ask that the algorithm structure is encoded via a **circuit**
- We can show matching upper + lower bounds *for any join* [Fan, Koutris, Zhao - '**24**]

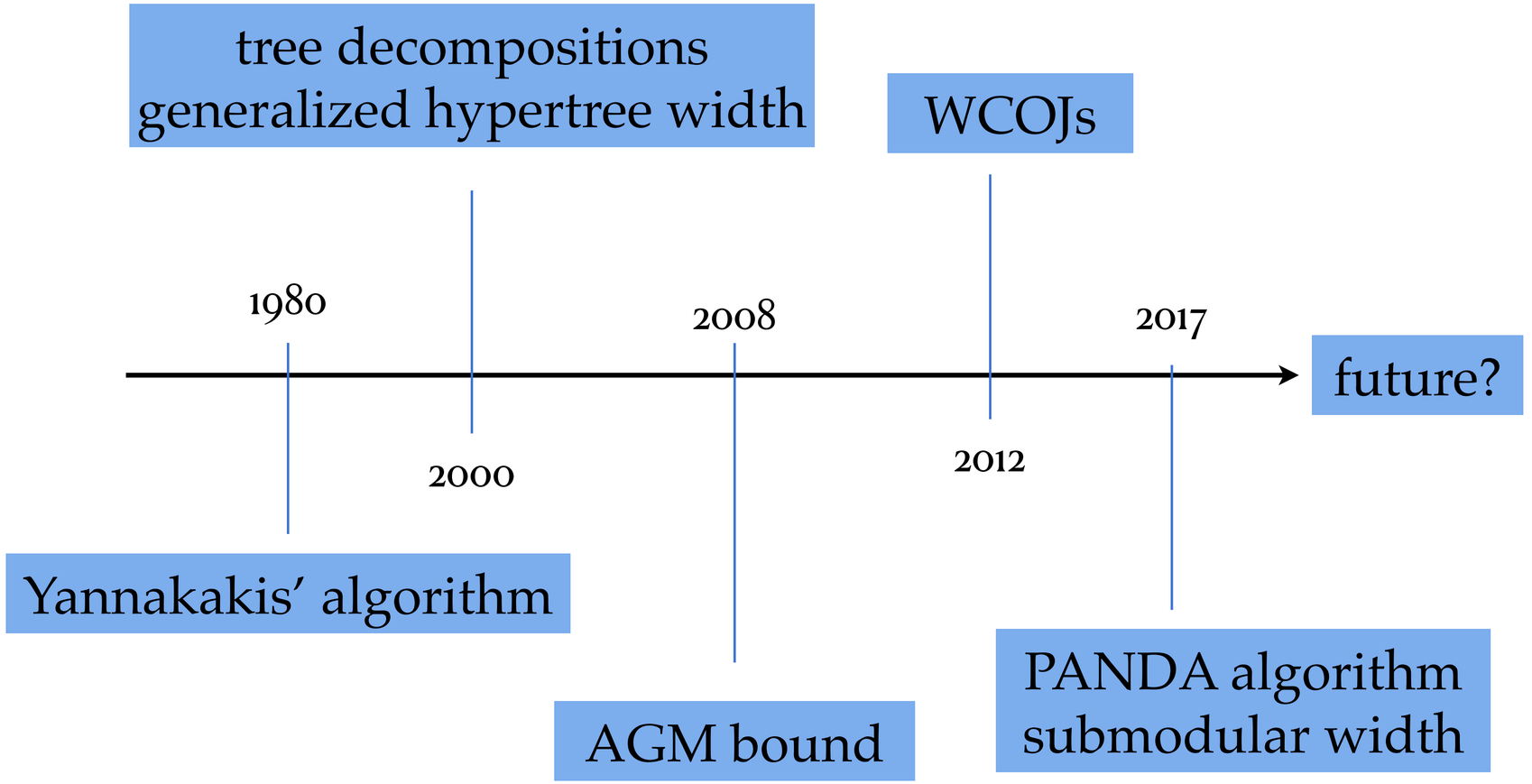


From Theory to Practice

Q5: Which theoretical developments can be effective in practice?

- some have already been successful: WCOJs, predicate transfer
- for some it is still unclear: enumeration?
- … and others are only theoretical (and will remain)

Thank You!



5 Key Ideas:

1. tree decompositions
2. data partitioning
3. using statistics
4. enumeration
5. the algebraic lens