



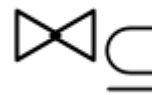
Datenbanksysteme II: Set Containment Join

Ulf Leser

Set Containment Join

DEFINITION 1 (SET CONTAINMENT JOIN). *Given two relations R, S with set-valued attributes $R.a$ and $S.b$, a set-containment join $R \bowtie_{\subseteq} S$ returns all pairs $(r, s), r \in R, s \in S$ where $r.a \subseteq s.b$.*

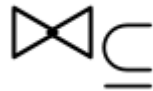
<i>Job ID</i>	<i>Required skills</i>
r1	{Java,C++}
r2	{Java,C++,SQL,EJB}
r3	{Java,C++,EAI}
r4	{Java,SOA}
r5	{Cloud}



<i>Applicant ID</i>	<i>Qualifications</i>
s1	{Java}
s2	{Java,C++,SOA}
s3	{Java,C++,SQL,EAI}
s4	{C++,EJB}
s5	{Java,EAI,EJB}
s6	{C++,SQL,EJB}

Set Containment Join

<i>JID</i>	<i>Required skills</i>
r1	{Java,C++}
r2	{Java,C++,SQL,EJB}
r3	{Java,C++,EAI}
r4	{Java,SOA}
r5	{Cloud}



=

<i>AID</i>	<i>Qualifications</i>
s1	{Java}
s2	{Java,C++,SOA}
s3	{Java,C++,SQL,EAI}
s4	{C++,EJB}
s5	{Java,EAI,EJB}
s6	{C++,SQL,EJB}

<i>Job ID</i>	<i>AppID</i>	<i>Required skills</i>
r1	s2	{ Java,C++ , SOA}
r1	s3	{ Java,C++ ,SQL,EAI}
r3	s3	{ Java,C++ ,SQL, EAI }
r4	s2	{ Java , C++, SOA }

SQL Formulation [BMGT15]

Requires

Skill	Course
Systems	RDBMS-1
Systems	OS
Databases	RDBMS-1
Databases	RDBMS-2

Passes

Student	Course
Ana	RDBMS-1
Ana	OS
Peter	RDBMS-1

```
select P1.Student, R1.Skill
from Passes as P1, Requires as R1
where not exists (select R2.Course
                  from Requires as R2
                  where R1.Skill = R2.Skill
                  and not exists (select P2.Course
                                from Passes as P2
                                where P2.Student=P1.Student
                                and P2.Course=R2.Course));
```

NF2 Formulation

Requires

Skill	Course*
Systems	{RDBMS-1, OS}
Databases	{RDBMS-1, RDBMS-2}

Passes

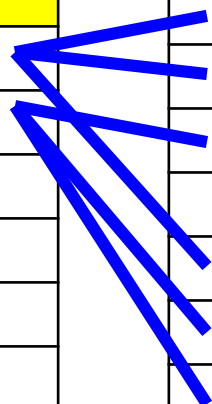
Student	Course*
Ana	{RDBMS-1, OS}
Peter	{RDBMS-1}

```
SELECT student, skill
FROM   requires R, student S
WHERE  S.course  $\subseteq$  R.course;
```

Computing SCJ – The Early Years [HM03, MGM03]

<i>Job ID</i>	<i>Required skills</i>
r1	{Java,C++}
r2	{Java,C++,SQL,EJB}
r3	{Java,C++,EAI}
r4	{Java,SOA}
r5	{Cloud}

<i>Job ID</i>	<i>Req.</i>	<i>A-ID</i>	<i>Qual.</i>
r1	C++	s1	Java
r1	Java	s2	C++
r2	C++	s2	Java
r2	EJB	s2	SOA
r2	Java	s3	C++
r2	SQL	s3	EAI
r3	C++	s3	Java
r3	EAI	s3	SQL
r3	Java	s4	C++
r3	Java	s4	EJB
...

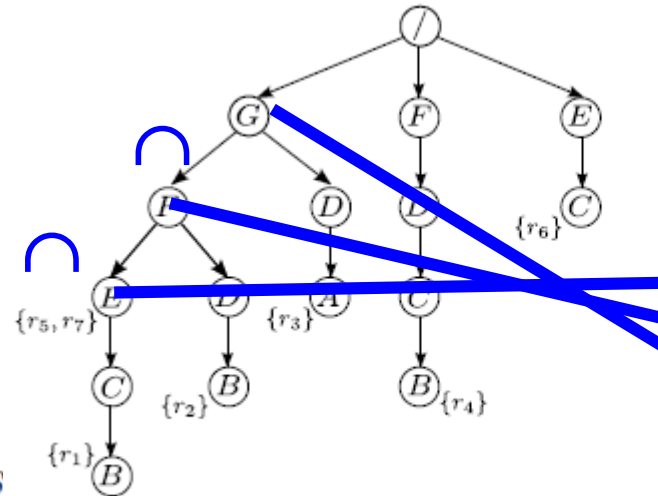


- Build on relational model
- Sort by ID and skills
- Sort-merge join, hash-join, partitioning, signatures etc.
- Problem: Much duplicate work

PRETTI: Prefix Tree for R / Inverted file for S [JP05]

$r_1: \{G, F, E, C, B\}$
 $r_2: \{G, F, D, B\}$
 $r_3: \{G, D, A\}$
 $r_4: \{F, D, C, B\}$
 $r_5: \{G, F, E\}$
 $r_6: \{E, C\}$
 $r_7: \{G, F, E\}$

$s_1: \{D, C, A\}$
 $s_2: \{G, F, E, D, C, A\}$
 $s_3: \{D, B\}$
 $s_4: \{G, F, C, B\}$
 $s_5: \{G, F, E, B\}$
 $s_6: \{F, E, D, C, B\}$
 $s_7: \{G, E, D, C, B\}$
 $s_8: \{G, E, D, C, B\}$
 $s_9: \{G, F, E, D\}$
 $s_{10}: \{G, F, E, D\}$
 $s_{11}: \{G, F\}$
 $s_{12}: \{G, F, E\}$



$A: \{s_1, s_2\}$
 $B: \{s_3, s_4, s_5, s_6, s_7, s_8\}$
 $C: \{s_1, s_2, s_4, s_6, s_7, s_8\}$
 $D: \{s_1, s_2, s_3, s_6, s_7, s_8, s_9, s_{10}\}$
 $E: \{s_2, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{12}\}$
 $F: \{s_2, s_4, s_5, s_6, s_9, s_{10}, s_{11}, s_{12}\}$
 $G: \{s_2, s_4, s_5, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\}$

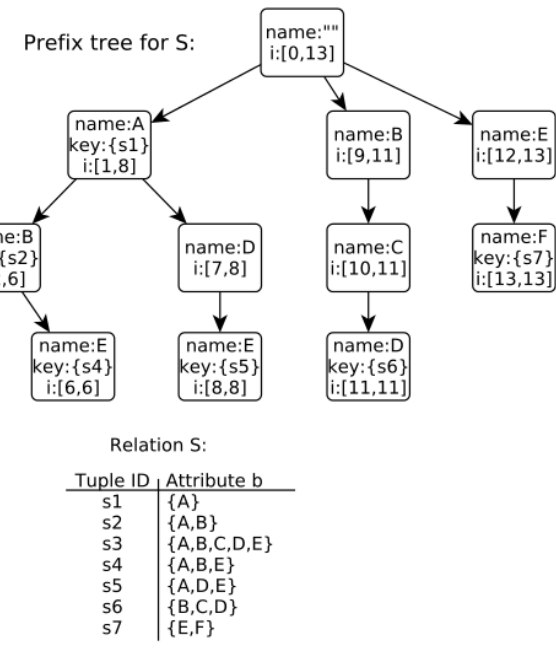
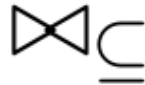
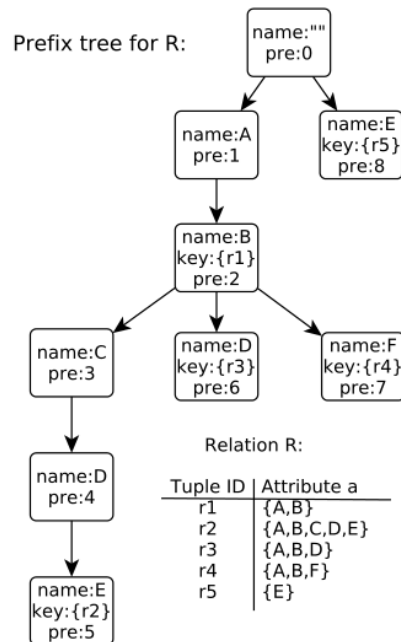
(a) left-hand collection R (b) right-hand collection S

(a) prefix tree T_R

(b) inverted index I_S

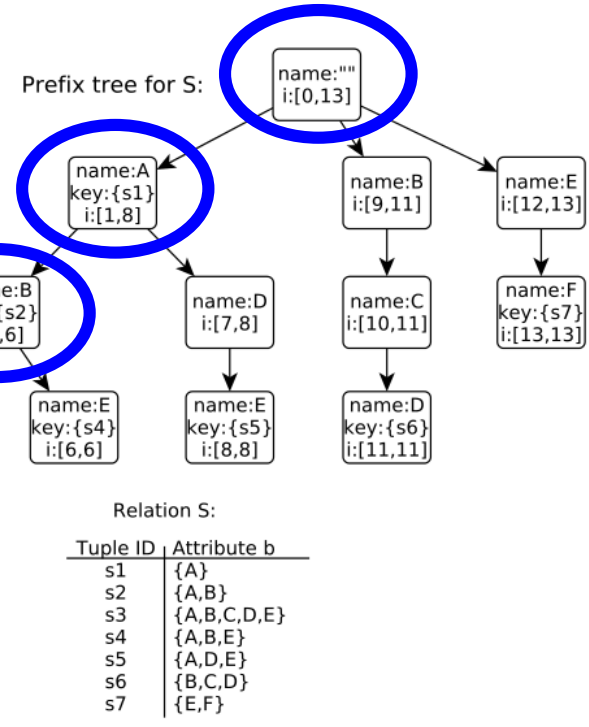
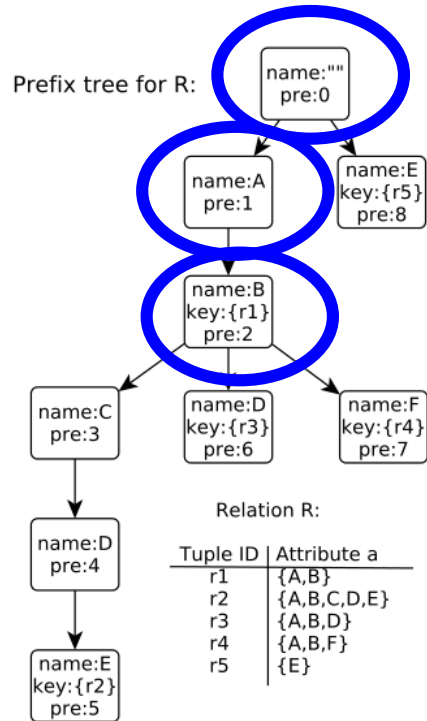
- Traverse $PT(R)$ and look up lists in $IF(S)$
- Progressively compute **list intersections** from $IF(S)$
- Output Cartesian products when matches are found
- Problem: Still **redundant work** regarding $IF(S)$

PIEJoin: Use Two Prefix Trees

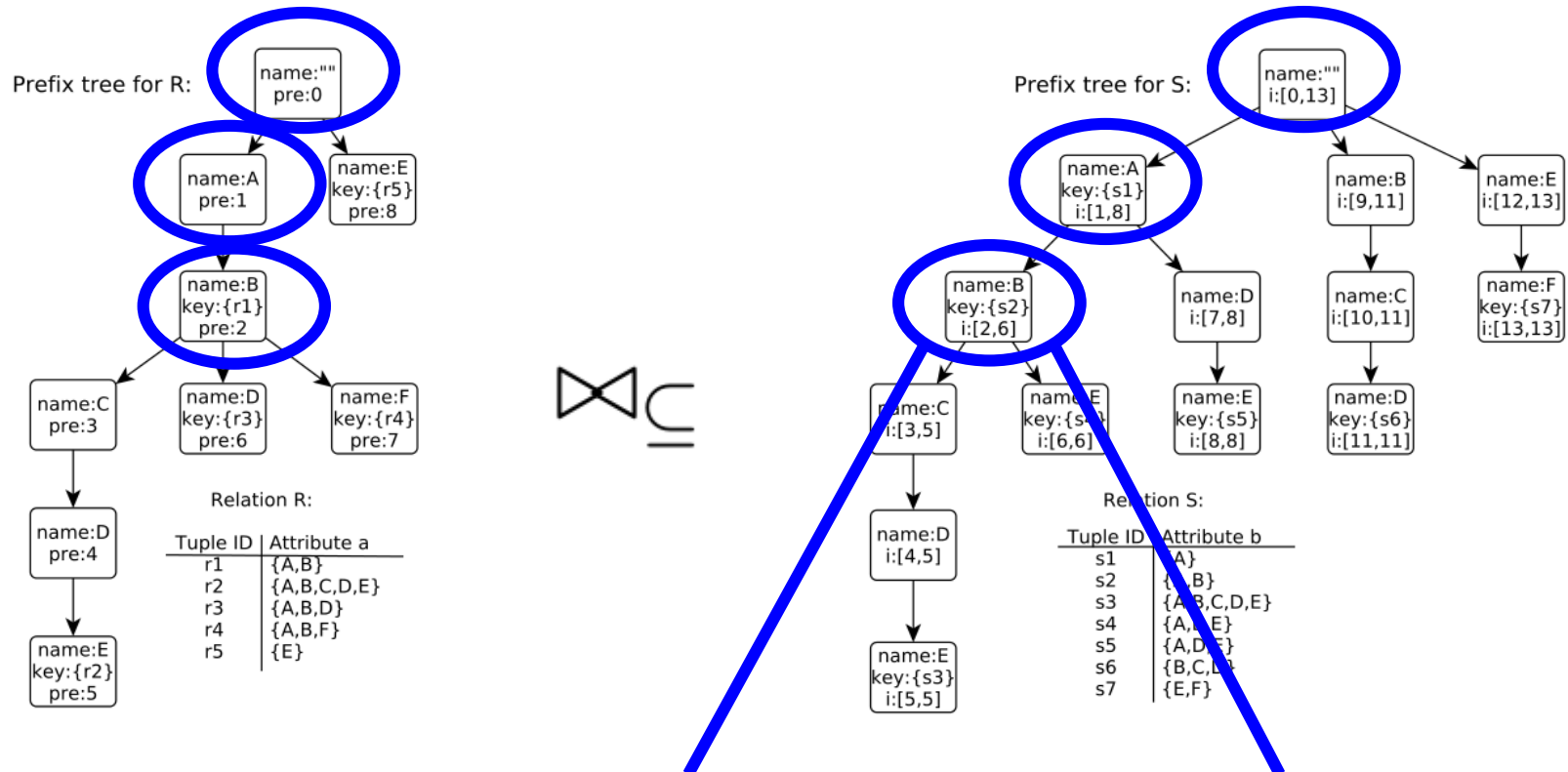


- Fix an **arbitrary order** of elements (frequency, alphabetical)
- Traverse both trees simultaneously
- When finding a key in R, **"extract" matching keys** in S

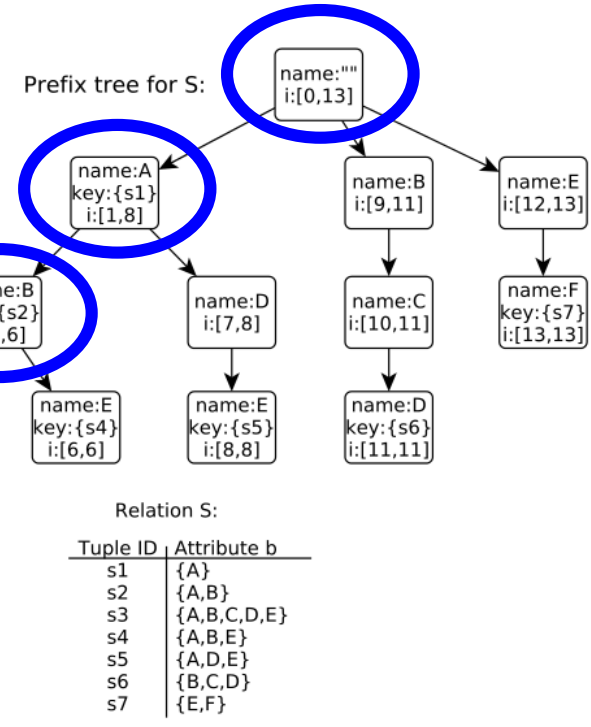
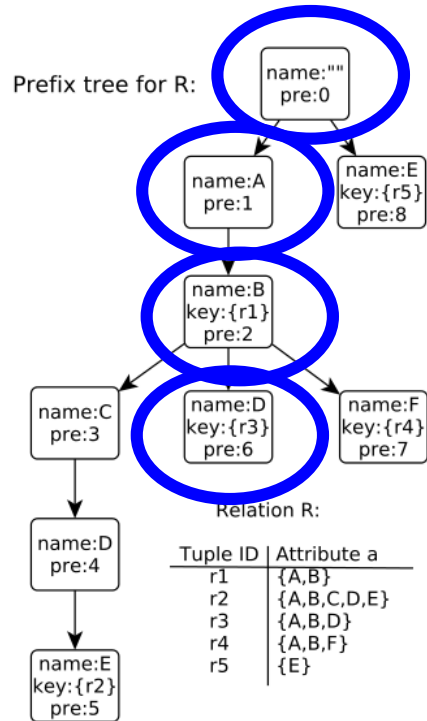
Traverse ...



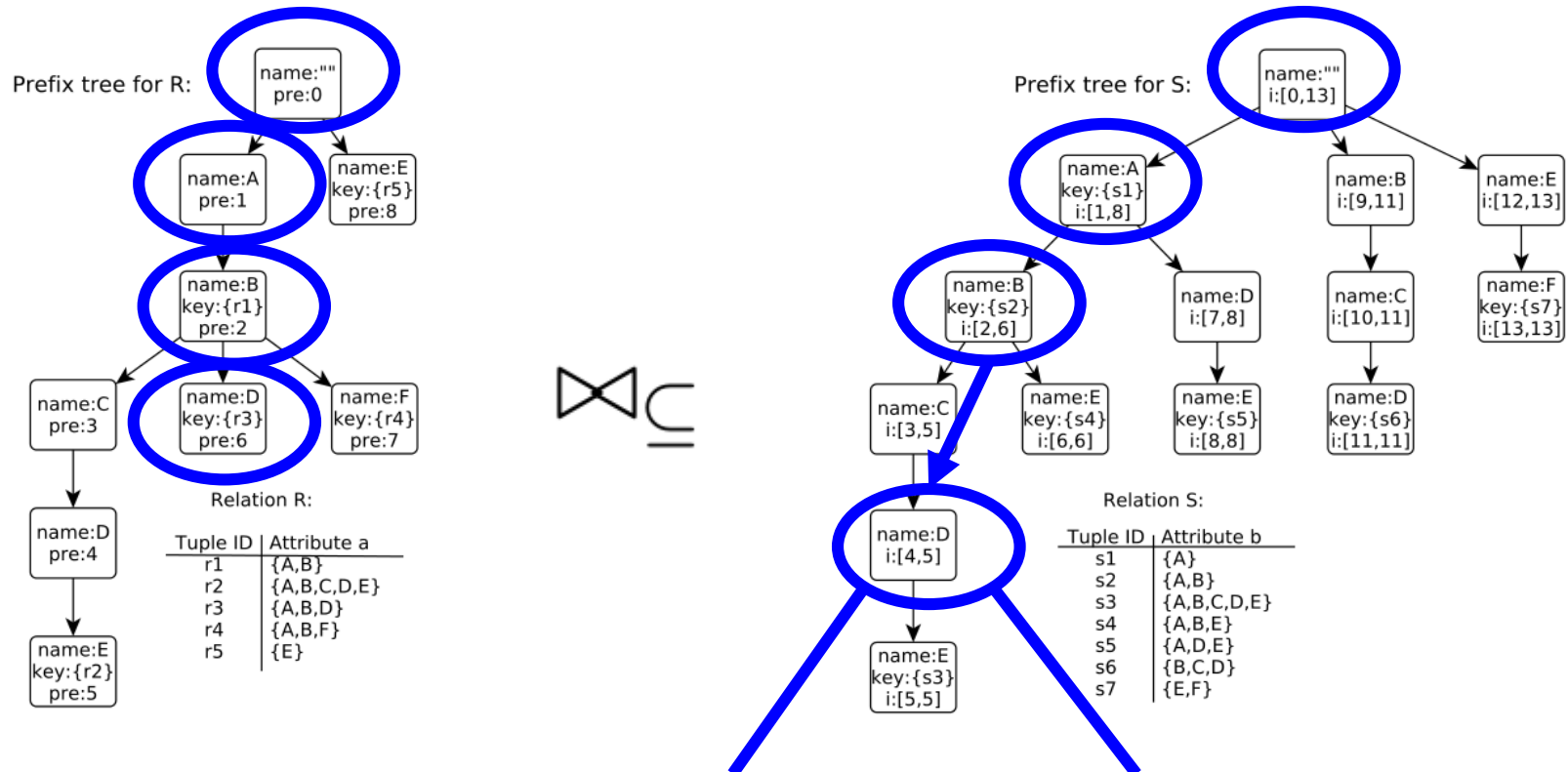
Problem P1: Find all Keys in Subtree



Traverse ...

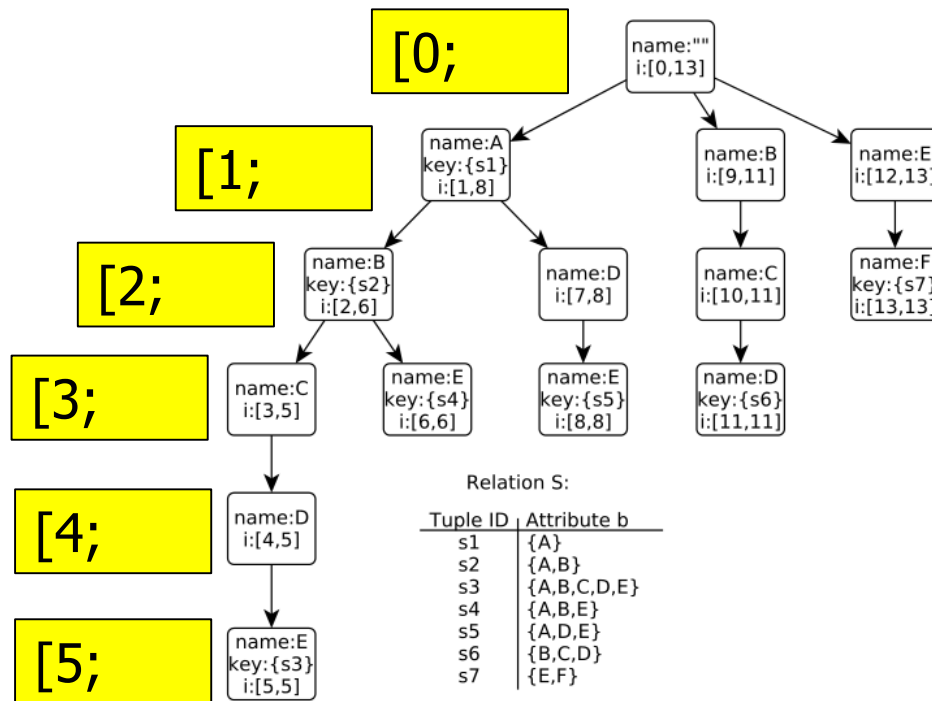


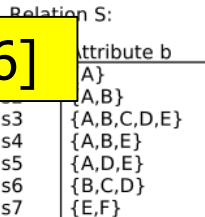
Problem P2: Find all „D“s underneath {A,B}



PIEJoin: Jump the Queue

Use **preorder indexing** of PT(S) to solve both problems
(with a few lookups)

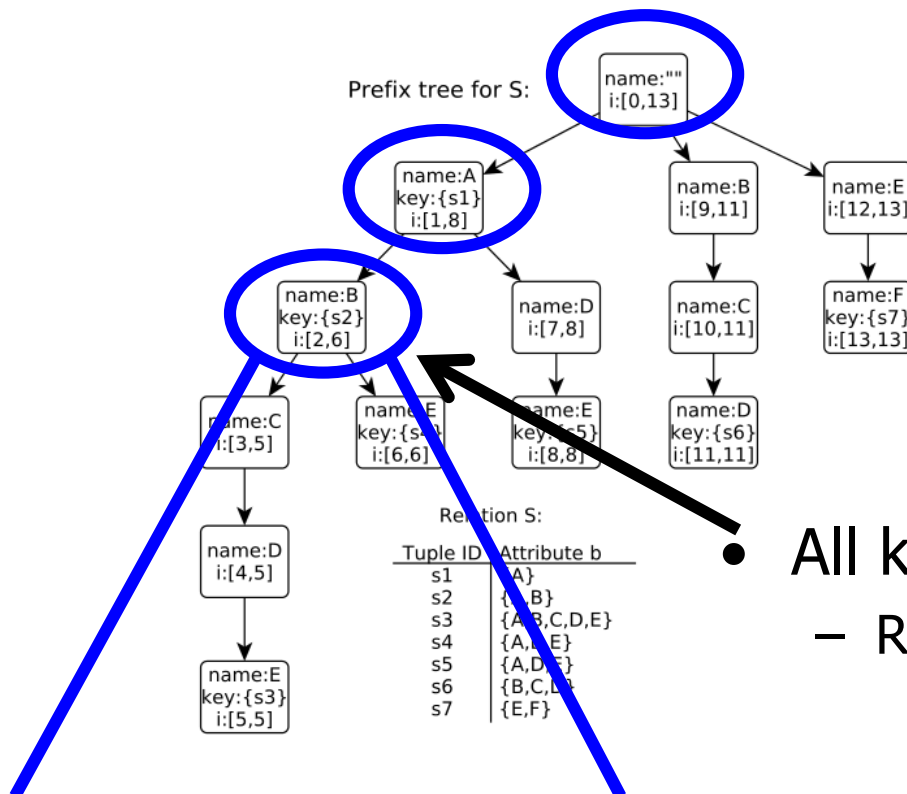




- All nodes p below a node n :
 $p.pre > n.pre$ and $p.post \leq n.post$
 - Build two lists sorted by pre/post
 - Finding all p : $2 * \log(|R|) + \text{list intersection}$

P1: Find all Keys in Subtree

Use Preorder indexing to solve both problems
(with a few lookups)

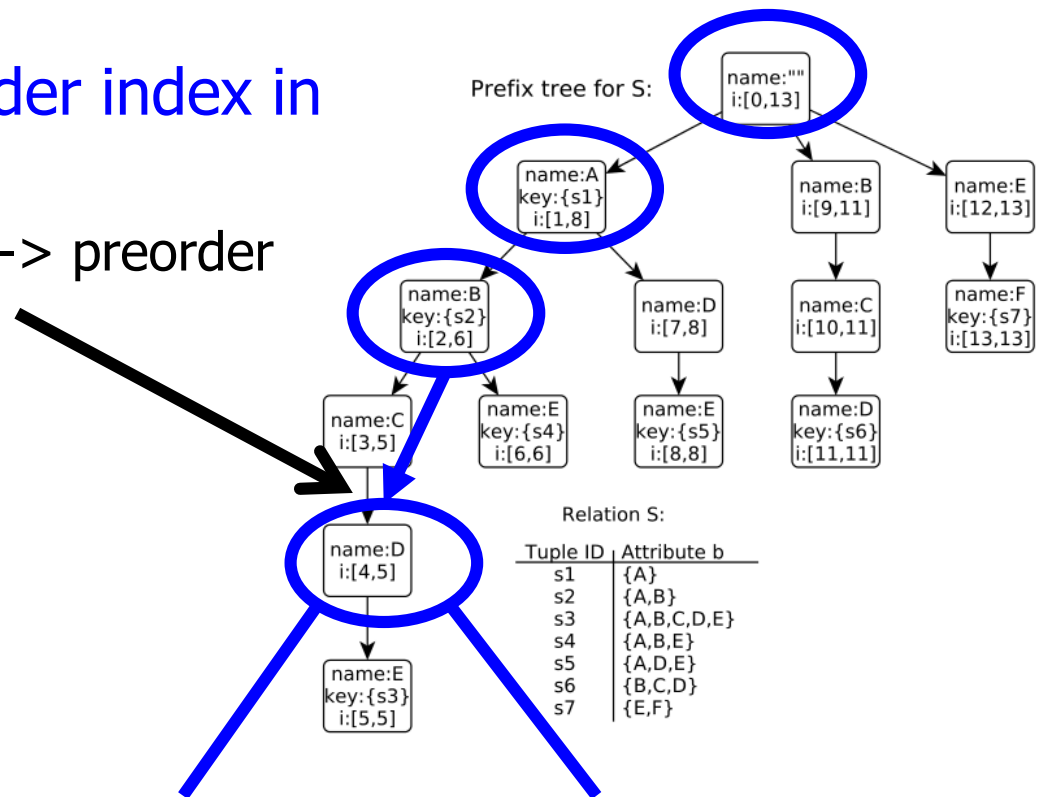


- All keys with preorder index in [2,6]
 - Requires index preorder -> keys

P2: Find Highest Occurrence of Search Key

Use Preorder indexing to solve both problems
(with a few lookups)

- All nodes with preorder index in $[2,6]$ and label "D"
 - Requires index label \rightarrow preorder



Trees can be very Space-Consuming

- Efficient implementation
 - Open source
- All based on **arrays and integers**, no pointers or objects
- See paper

Evaluation

- Comp.: PRETTI [JP05], PRETTI+ [FLH+15], LIMIT+ [BMGT16]
 - On-the-fly indexing in main memory - included in measurements
- Eight real-world data sets; non-self-joins in paper
- Some data sets properties

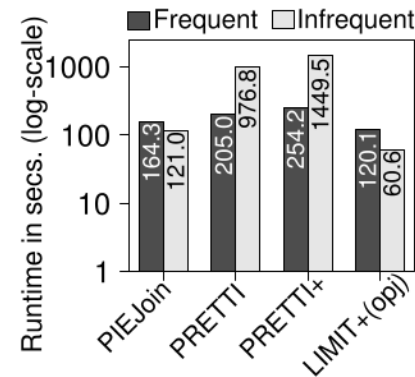
- Number of sets
- Number **unique elements**
 - Fan-Out of prefix trees
- Average **size of sets**
 - Depth of prefix trees

Data set	Domain size	No. of tuples	Max. set size	Avg. set size	Join cardinality	Size in MB
BMS	1,657	515,597	164	6.53	$3.2 * 10^9$	11.2
Flickr	810,660	1,680,490	102	9.78	$1.6 * 10^9$	79.6
Flickr-LC	618,971	3,546,729	1,230	5.36	$6.3 * 10^9$	132.9
Kosarak	41,270	990,002	2,497	8.10	$5.5 * 10^{10}$	31.6
Netflix	17,770	480,189	17,653	209.25	$1.6 * 10^8$	426.4
Orkut	15,293,693	1,853,285	2,958	57.16	$1.9 * 10^6$	881.7
Twitter	1,318	371,586	687	65.96	$1.2 * 10^8$	82.3
Webbase	15,146,263	168,707	3,842	463.64	$2.3 * 10^7$	709.6

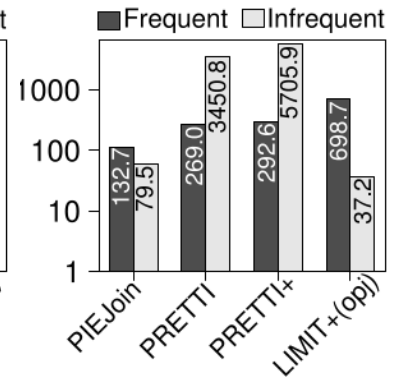
- **Skewness** of element frequencies
 - Real-world data sets are skewed!
- **Sort-order** (frequent first, infrequent first)
 - Frequent first: Large intermediate sets, quickly shrinking

Results

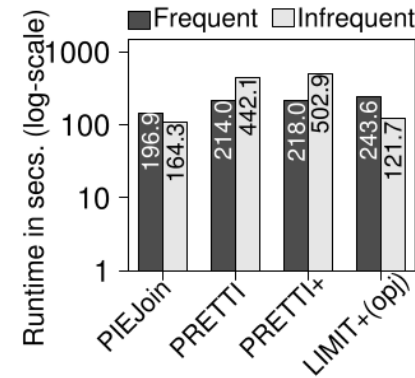
- **PIEJoin outperforms PRETTI / PRETTI+** in 7/8 data sets
- **LIMIT+ outperforms PIEJoin** in 11/16 data sets
- Contradicting [FLH+15], our results indicate that PRETTI is faster than PRETTI+



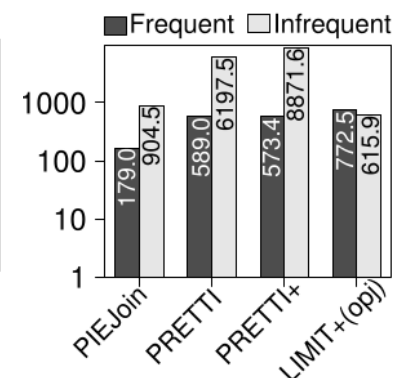
(a) BMS



(b) Flickr



(c) Flickr-LC

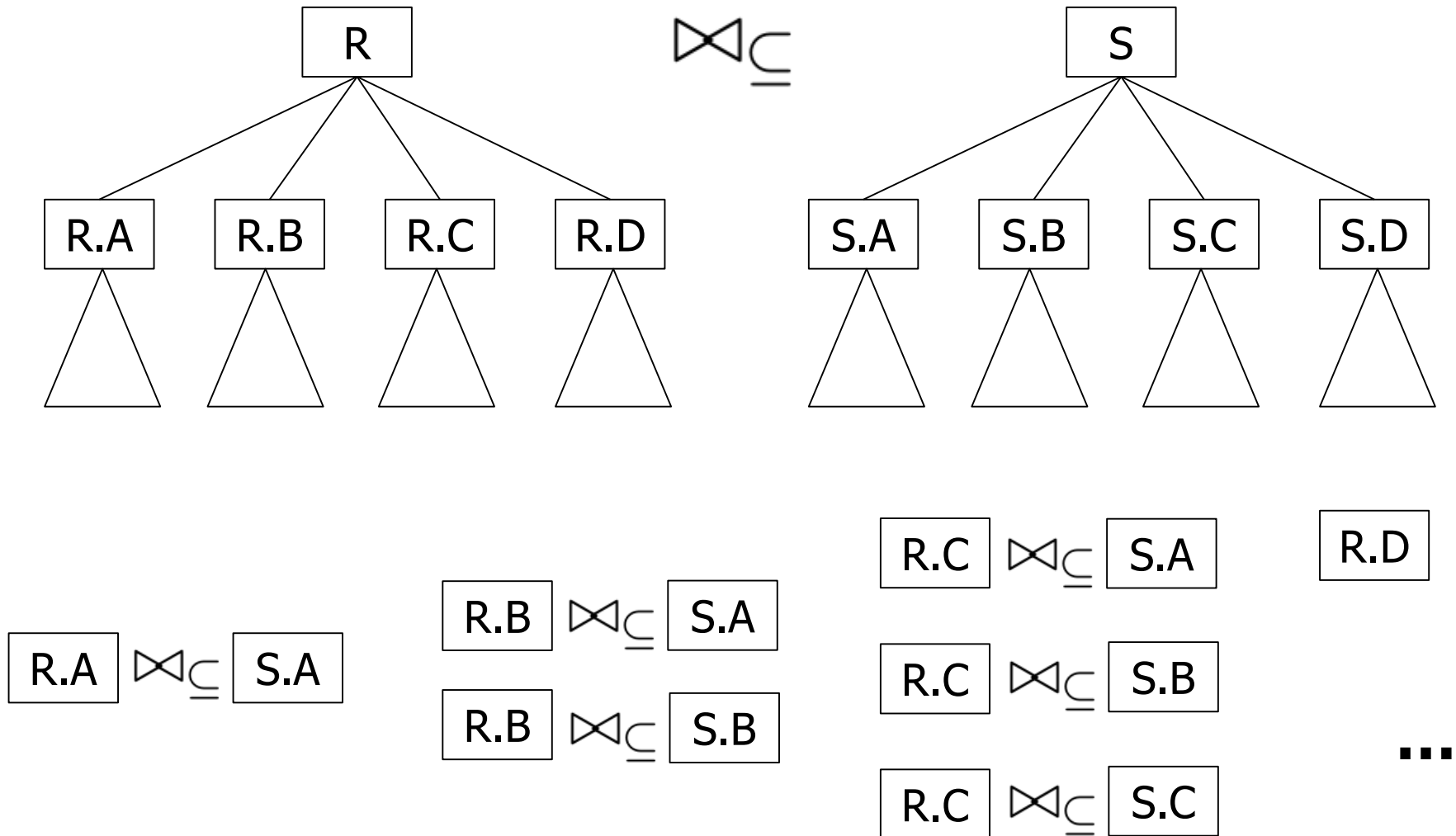


(d) Kosarak

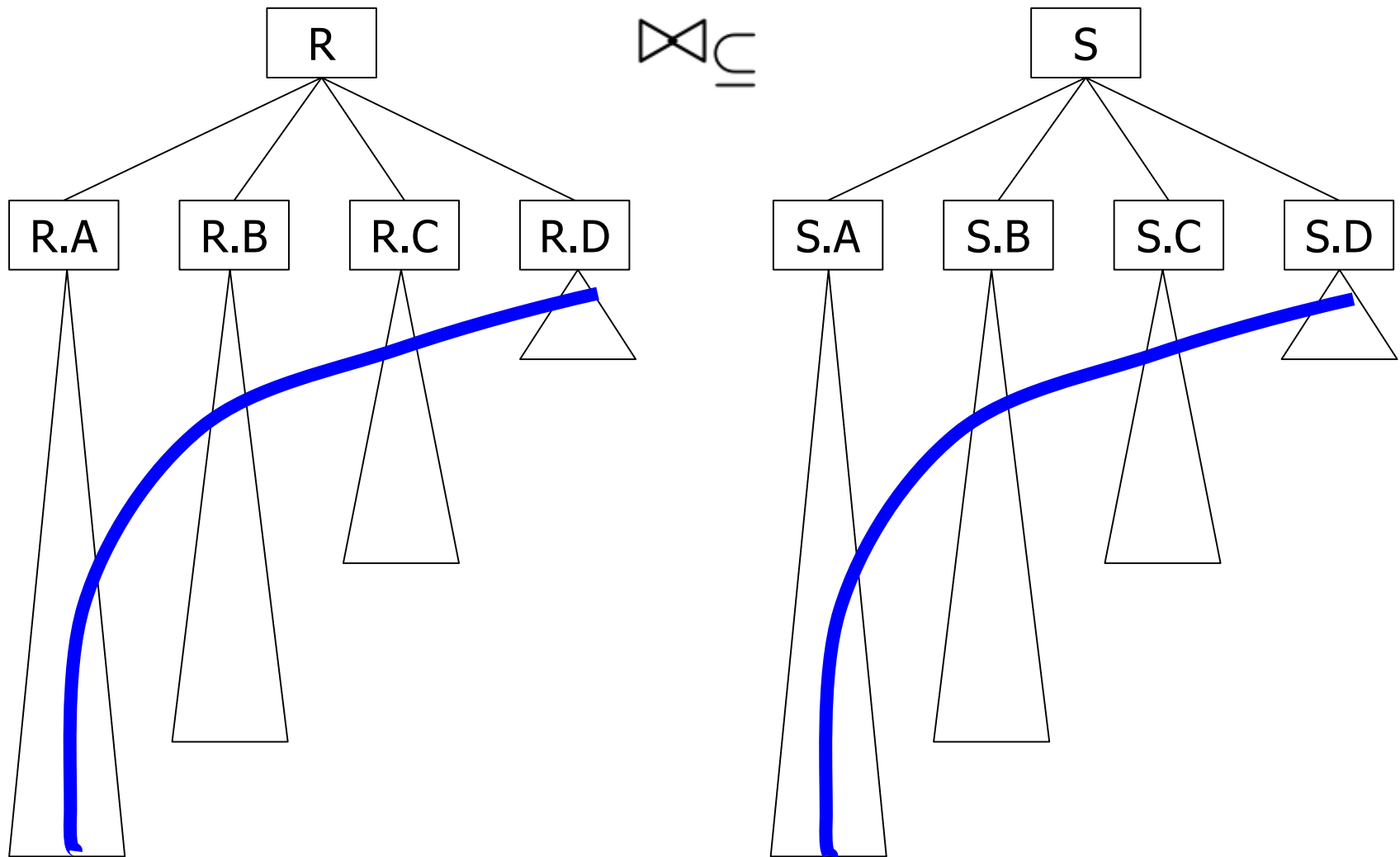
Intermediate Result

- PRETTI / PRETTI+ very sensitive to **sort order**
- **LIMIT+** is the overall fastest method
 - But no orders-of-magnitude differences to PIEJoin
- LIMIT+ also fastest in RxS: 5x PIEJoin, 50x PRETTI
- PIEJoin has **lowest memory footprint** (factors 2-20)
- Prefix-Tree-Joins call for **partitioning and parallelization**

Partitioning SCJ on Prefix Trees

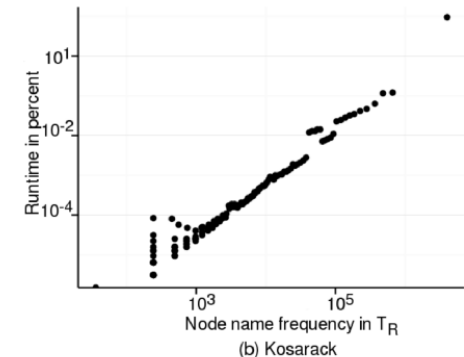
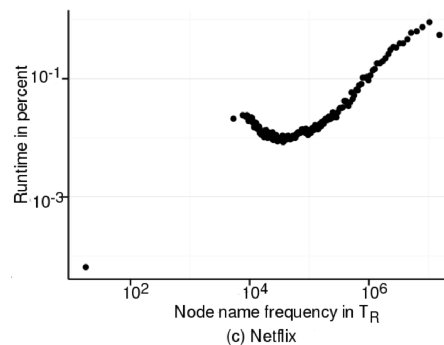
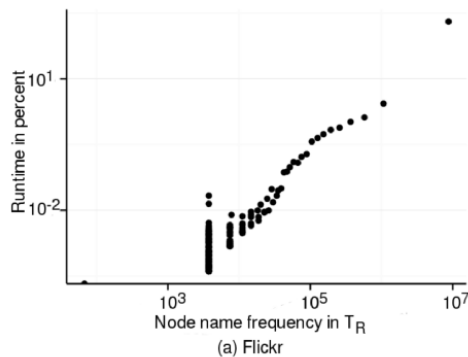


REAL CASE: Partitioning SCJ on Prefix Trees



Parallelization Issues

- Different sub-trees have largely different sizes
 - Different **sub-tree-joins** need largely different run times
 - Stragglers – **bad scalability**
- Concrete behavior depends on **frequencies of sub sets**
 - Fortunately, **element frequency** does roughly correlate to work load

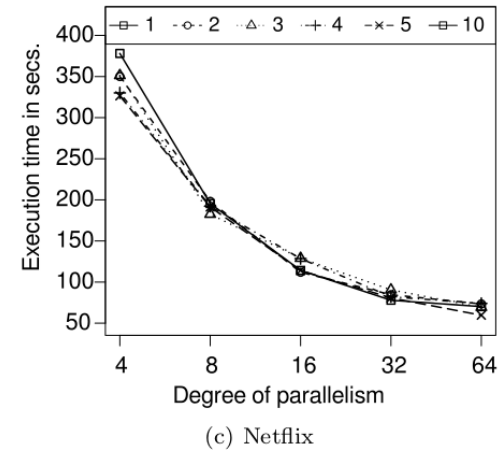


Best we Have so Far

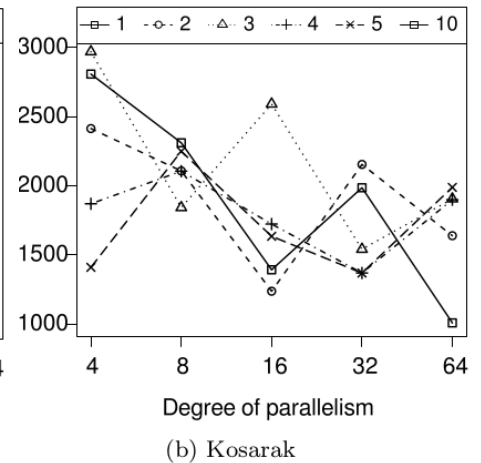
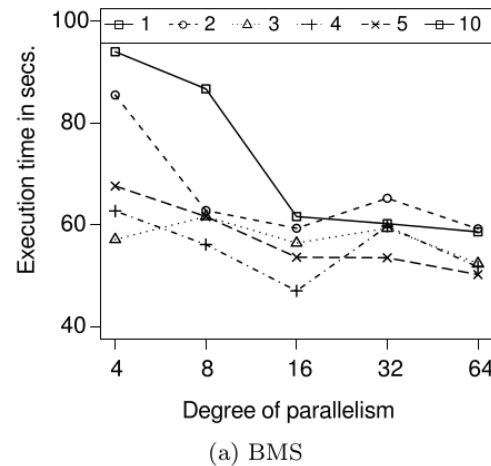
- As always: More partitions create better load balancing yet more overhead
 - E.g. synchronization of result list
 - Parameter rf : Create $rf \cdot DOP$ work groups
 - DOP: Number of available (hardware) threads
- Observation: For large alphabets (>1000), partitioning at level 1 already creates dozens of millions of tasks
 - Too much overhead
- Solution: Sub-tree-joins are **adaptively range-partitioned** into work groups at level 1 or level 2
 - Most work is done at upper levels – very large list intersections
 - **Greedy partitioning** – find range with $\sim 1/rf \cdot DOP$ fraction of work

Evaluation

- Sometimes wonderful



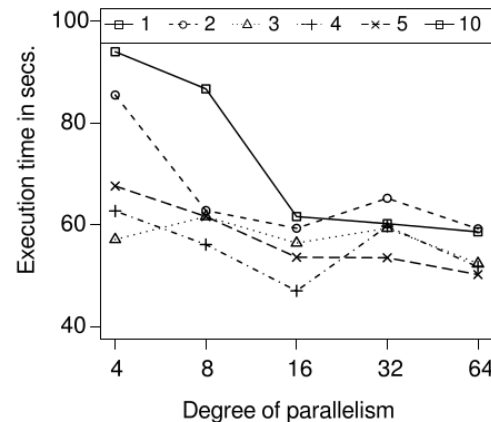
- Often ugly



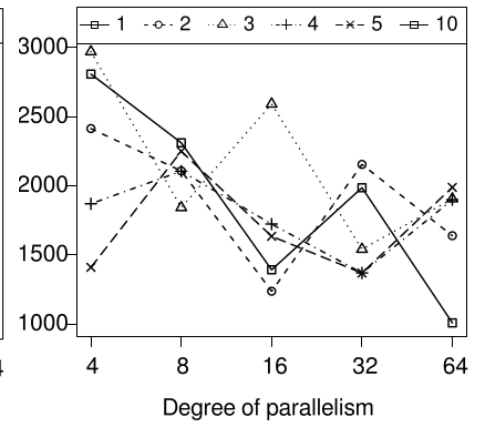
Reasons

- Greedy is not optimal
- Element frequency is not a perfect predictor of work
- Work loads are very heterogeneous
- ...

- Often ugly



(a) BMS



(b) Kosarak

Conclusions

- Set containment join: Operation with many applications
- Parallel PIEJoin: By far **the fastest SCJ algorithm** so far
 - Efficient basic design + parallelization
 - With 64 threads: Speedup of 30% to 550% compared to LIMIT+
- All code and data available
- Further ideas
 - Adaptive, **scalable parallelization strategies**
 - No fixed “level 2”
 - Better estimation of work
 - Partition the sets, not the items?
 - Given data set characteristics – **determine optimal combination** of sort-order and algorithm
- Related problems: **Set similarity join**

References

- Bouros, P., Mamoulis, N., Ge, S. and Terrovitis, M. (2015). "Set containment join revisited." Knowledge and Information Systems.
- Kunkel, A., Rheinländer, A., Schiefer, C., Helmer, S., Bouros, P. and Leser, U. (2016). "PIEJoin: Towards Parallel Set Containment Joins". Int. Conf. on Scientific and Statistical Database Management, Budapest, Hungary.
- Jampani, R., & Pudi, V. (2005). „Using prefix-trees for efficiently computing set joins“. International Conference on Database Systems for Advanced Applications. Springer, Berlin, Heidelberg.
- Luo, Y., Fletcher, G. H., Hidders, J., & De Bra, P. (2015). „Efficient and scalable trie-based algorithms for computing set containment relations“. International Conference on Data Engineering
- ... three new algorithms since 2016: FreshJoin, TT-join, LC-Join