

Scalable computation of acyclic joins

Rasmus Pagh
IT University of Copenhagen

Joint work with Anna Östlin Pagh
PODS 2006

Outline

- ▶ What is an acyclic join?
- ▶ Case study: *Star schemas*
 - Worst case performance of known algorithms
 - Using our new approach
- ▶ Our general result
- ▶ Cyclic joins

Natural join

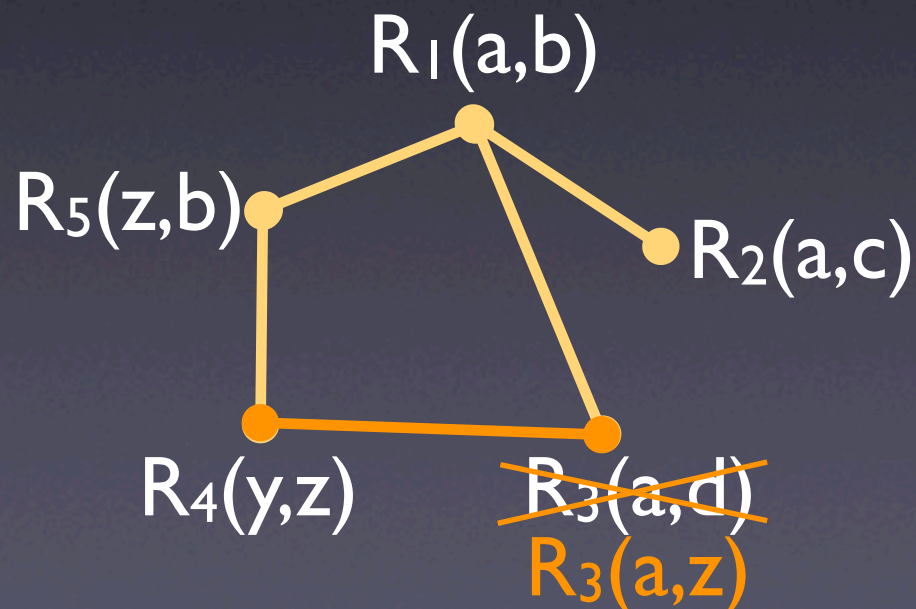
- Given relations R_1, \dots, R_k , compute the relation R consisting of all tuples whose projections (to appropriate attributes) can be found in R_1, \dots, R_k .
- **NP-hard** in general [MSY '81].
- Special case of *acyclic joins* known to be polynomial time computable [G, YO '79], [Y '81].

Acyclic join

- **Definition.** A join of relations R_1, \dots, R_k is *acyclic* if there exists a tree with vertices R_1, \dots, R_k such that the join can be expressed as a theta-join with equality conditions only between adjacent relations.

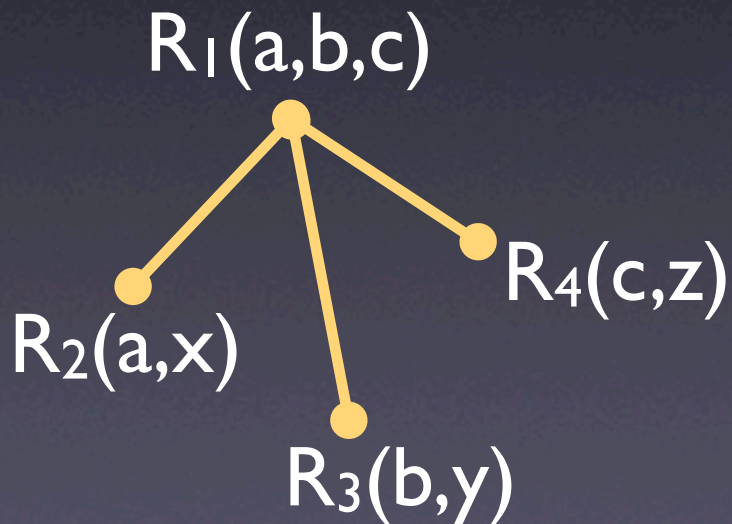
- **Example:**

cyclic
join



Special case: Star schema

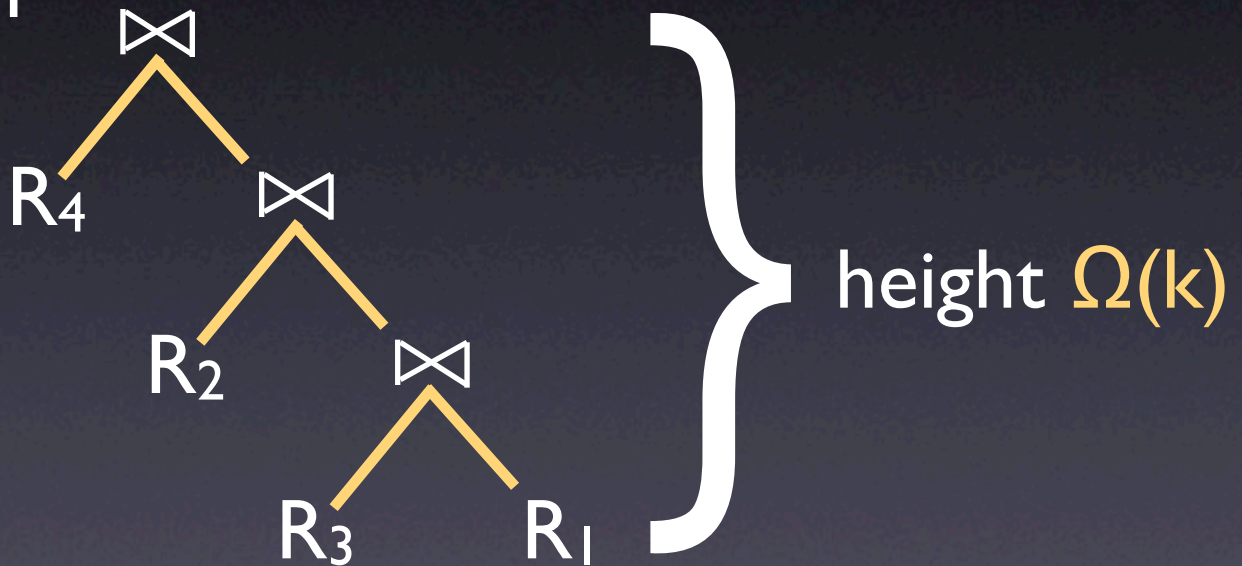
- **Star schema:** The attribute sets of the relations are disjoint, except R_1 which contains an attribute from each of R_2, \dots, R_k .
- **Example:**



Star schema join

- Natural join of R_1, \dots, R_k having a star schema.
- Any join not involving R_1 is a **cartesian product**.

- Best query plan usually highly **unbalanced**:



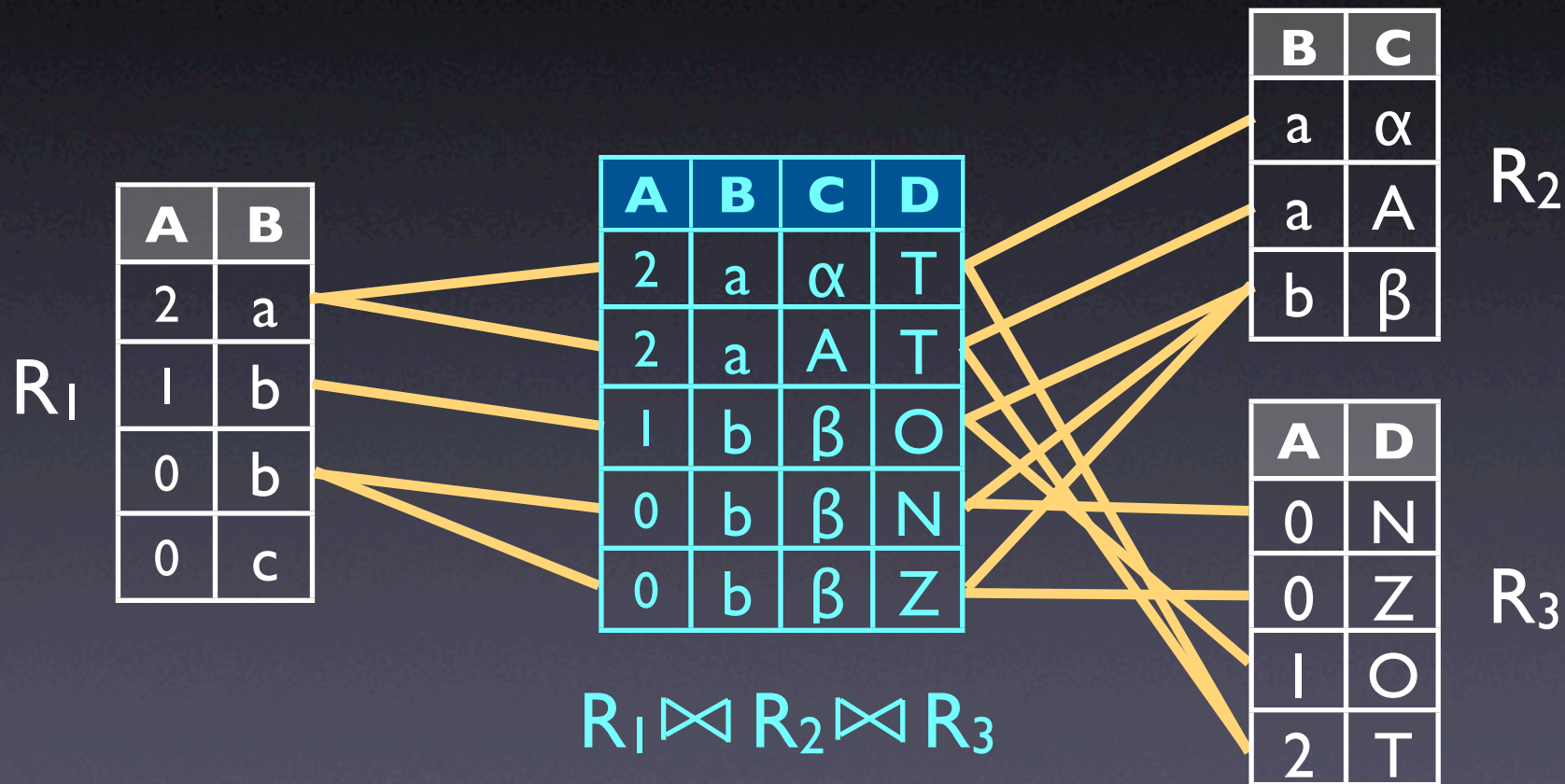
- All intermediate results, and the final result, may have roughly the same size as R_1 .

Worst-case complexity

- Let n denote the size of input (\approx size of output).
- Total size of inputs to binary joins $\Omega(kn) \Rightarrow$
 - $\left\{ \begin{array}{l} \Omega(kn) \text{ time on a RAM} \\ \Omega(kn/B) \text{ I/Os in the I/O model} \end{array} \right.$
- Pipelining and indexing don't help asymptotically.
- Next: Eliminating the dependence on k .

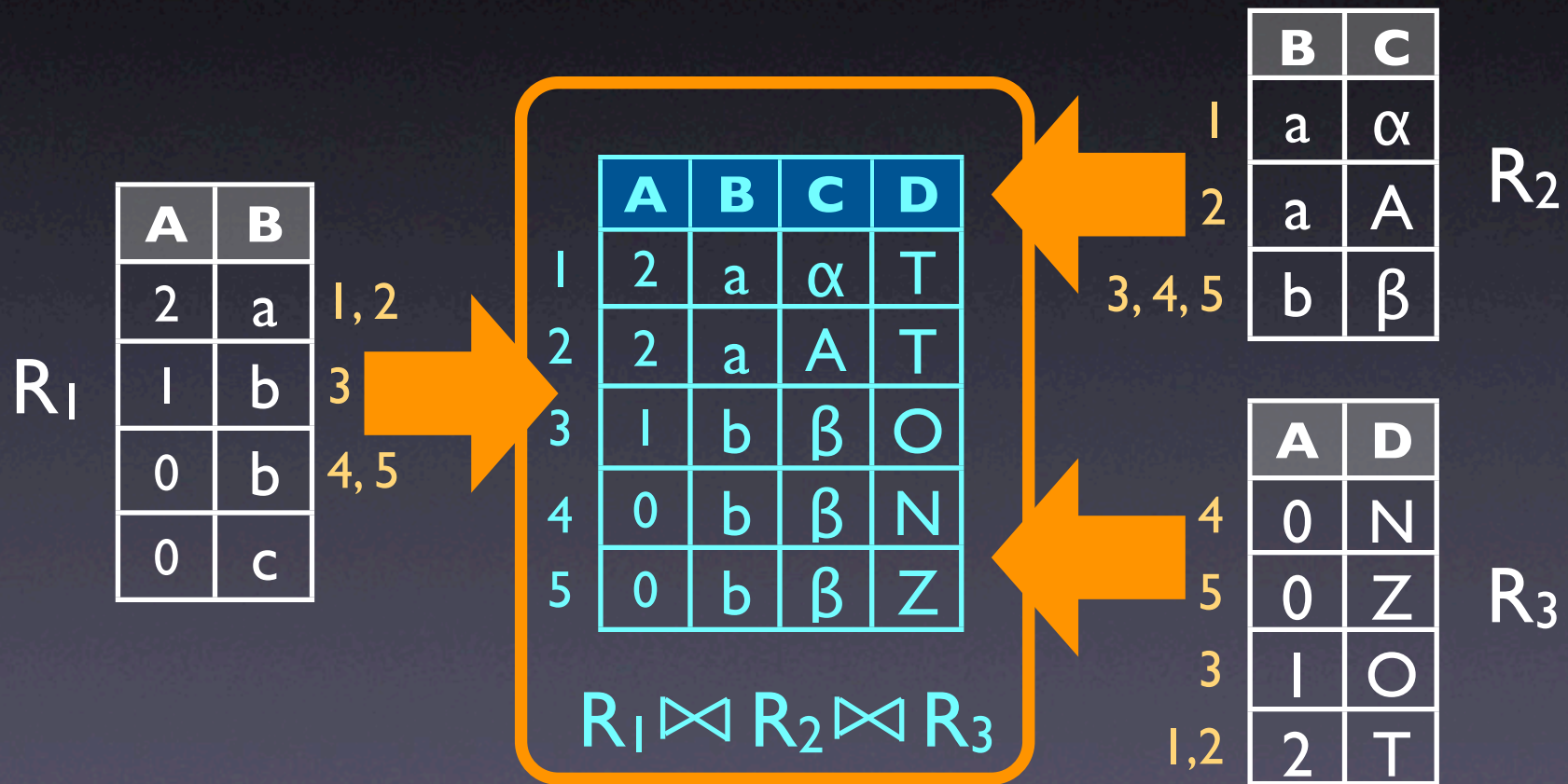
New approach

- *Abandon* the idea of intermediate results.
- Compute relationship between input and output.



New approach

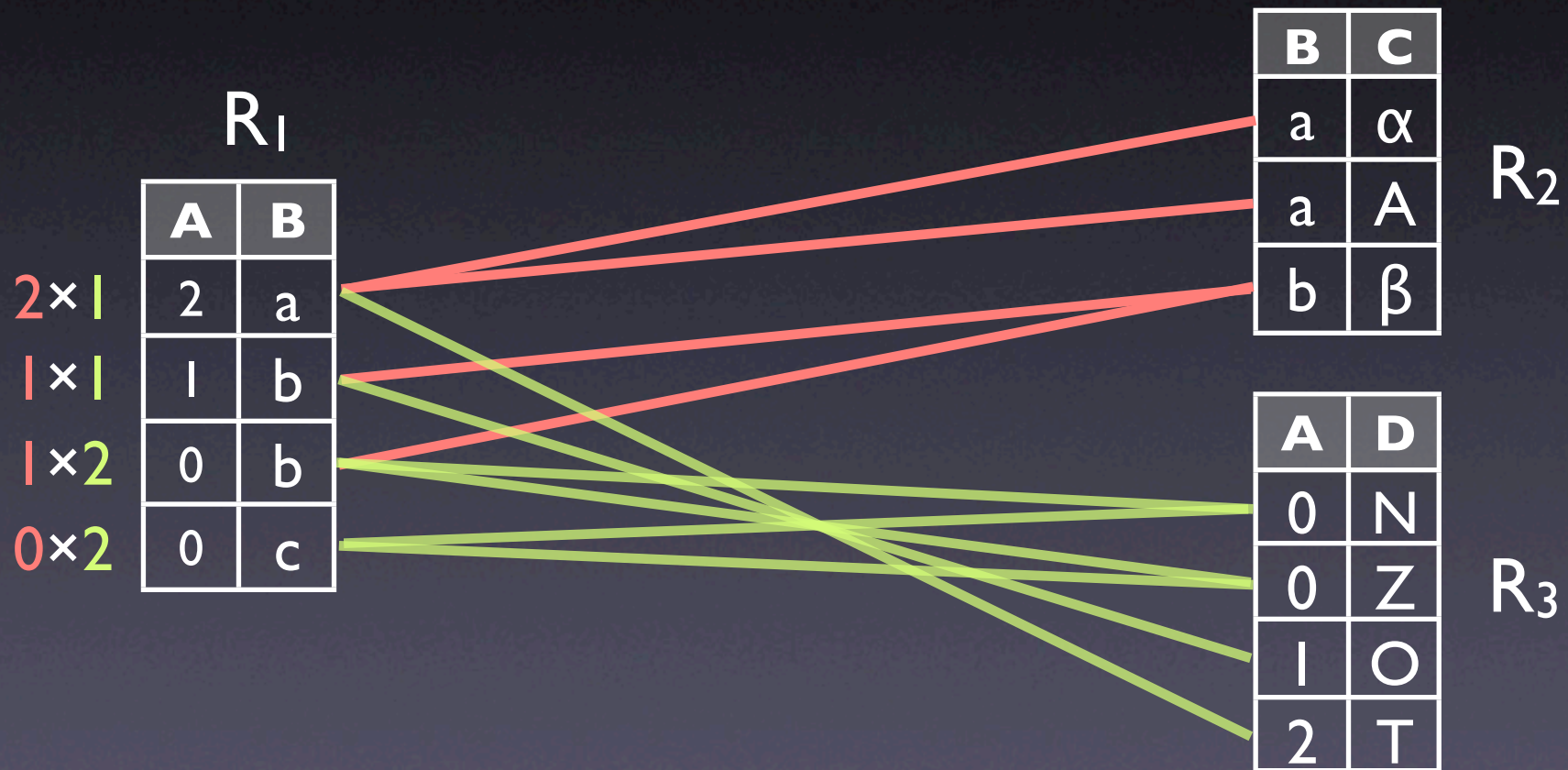
- *Abandon* the idea of intermediate results.
- Compute relationship between input and output.



sort by row numbers

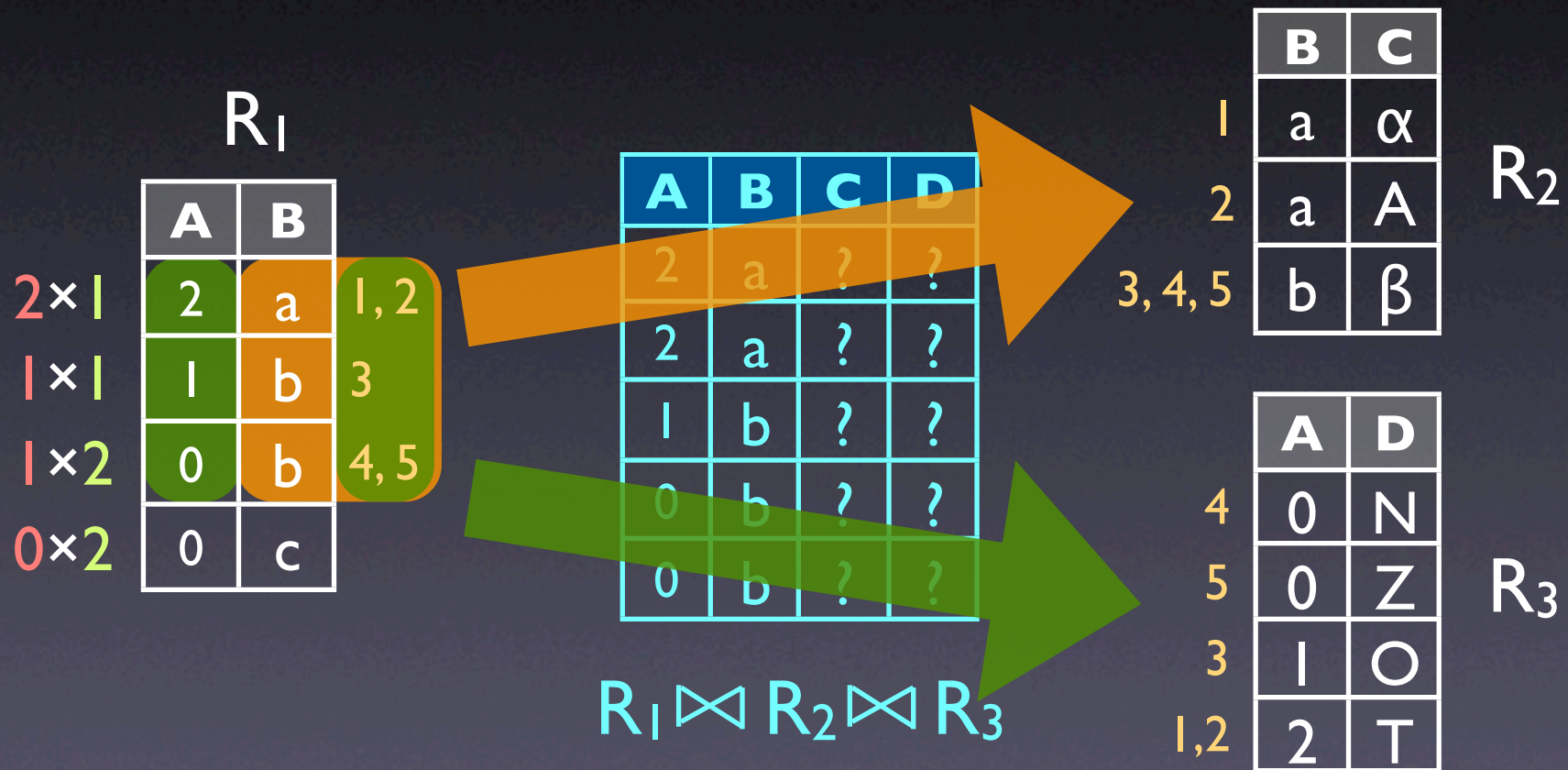
Implementation

I. Count number of tuples matching each tuple in R_1 .



Implementation

1. Count number of tuples matching each tuple in R_1 .
2. Enumerate the output tuples.

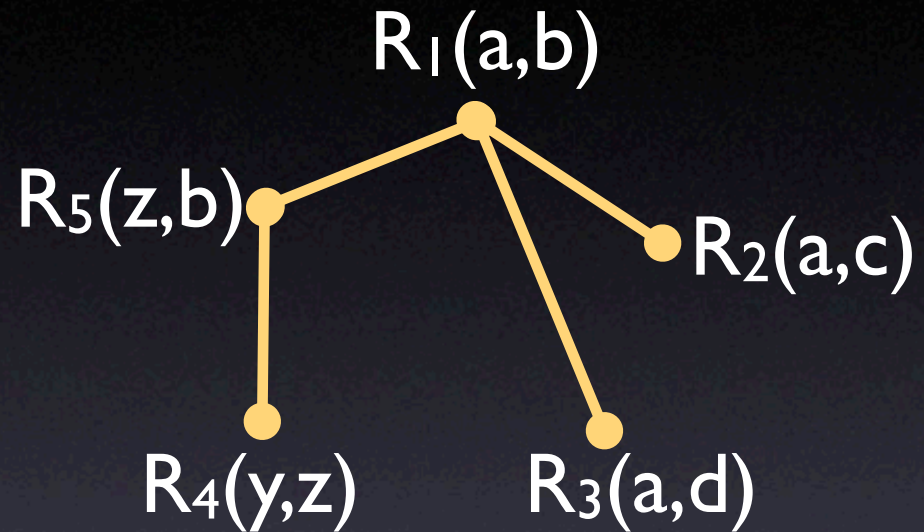


General algorithm

- More complicated application of the same ideas — guided by the tree of the acyclic join.
- *Counting phase* proceeds bottom-up in tree — computes #tuple occurrences in sub-joins.
- *Enumeration phase* proceeds top-down — assigns output tuple numbers to all tuples.
- *Sort* to produce the result.

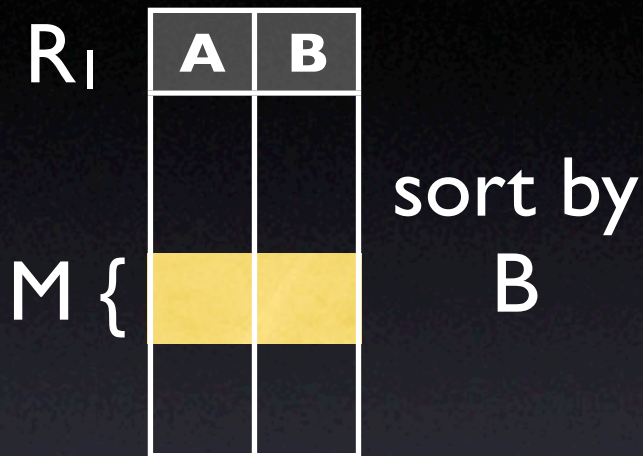
General result

- Acyclic join of k relations.
- Let n denote the input size plus the size of data involved in the equality conditions of the join.
Let z denote the output size.

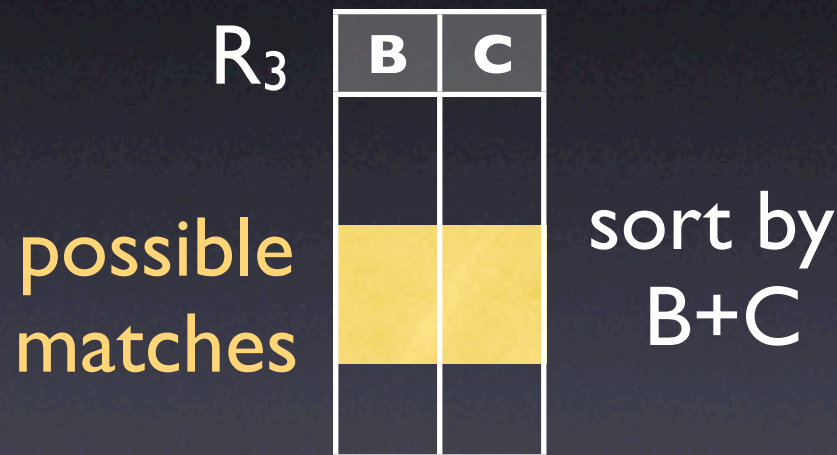
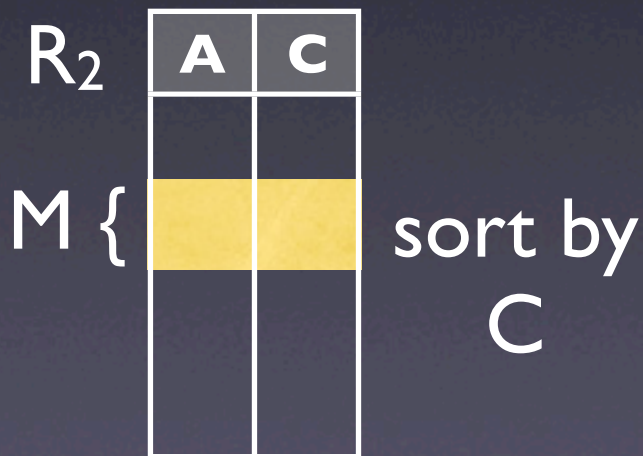


- We can compute the join in
 - $O(\text{sort}(n+z))$ I/Os on external memory
 - $O(n+z)$ expected time on a RAM

Cyclic joins: 3-cycles



Analysis: Each tuple in R_3 is a possible match $O(n/M)$ times.



$O(n^2/(MB) + \text{sort}(n))$ I/Os

(previously: $O(n^2/B)$ I/Os)

Conclusion

- Relational algebra forms the basis of query languages, but it is not always the best basis for query processing!
- We saw two examples where evaluation based on a relational algebra expression is suboptimal.

Open problems

- Experimental evaluation — how many relations are needed before the new algorithm is better?
- Are any cyclic joins computable in sorting complexity (worst case)?
- Which other join graphs can be handled in $\tilde{O}(n^2/(MB))$ I/Os (worst case)?
Status: 3-cycles and 4-cycles.

Thank you!