Information Retrieval
정보검색론

Younghoon Kim
(nongaussian@hanyang.ac.kr)

QUERY PROCESSING WITH AN INVERTED INDEX

Google's Query Language Model

- Boolean and positional model
  - Boolean query
    - Conjunction: Hanyang University
    - Disjunction: Hanyang |University
    - Negation: Hanyang -University
  - Phrase query
    - "Hanyang University"
  - Phrase query with wildcards
    - "Hanyang * University"

Query Language of Our Search Engine

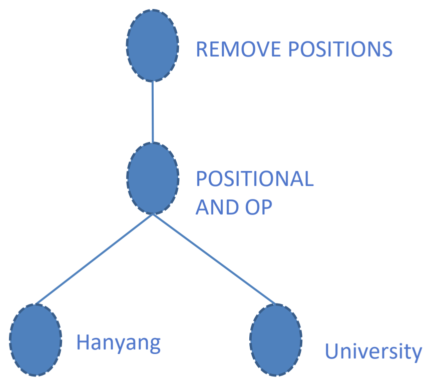
- Query language model
  - Simple conjunctive Boolean and positional query language
  - E.g.,
    - Conjunction: Hanyang University
    - Phrase search: “Hanyang University "

Query Processing

Query processing

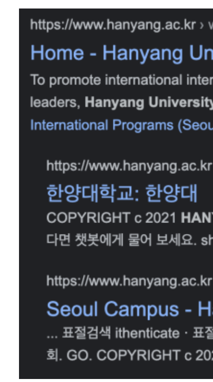
“Hanyang University”

Query parsing



Merging inverted lists

…

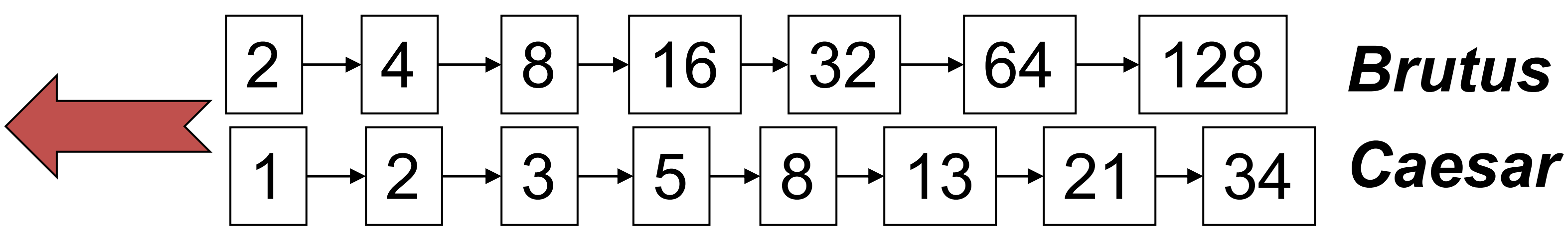


Query processing: AND

- Consider processing the query:

  Brutus *AND* ***Caesar***

  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings (intersect the document sets):



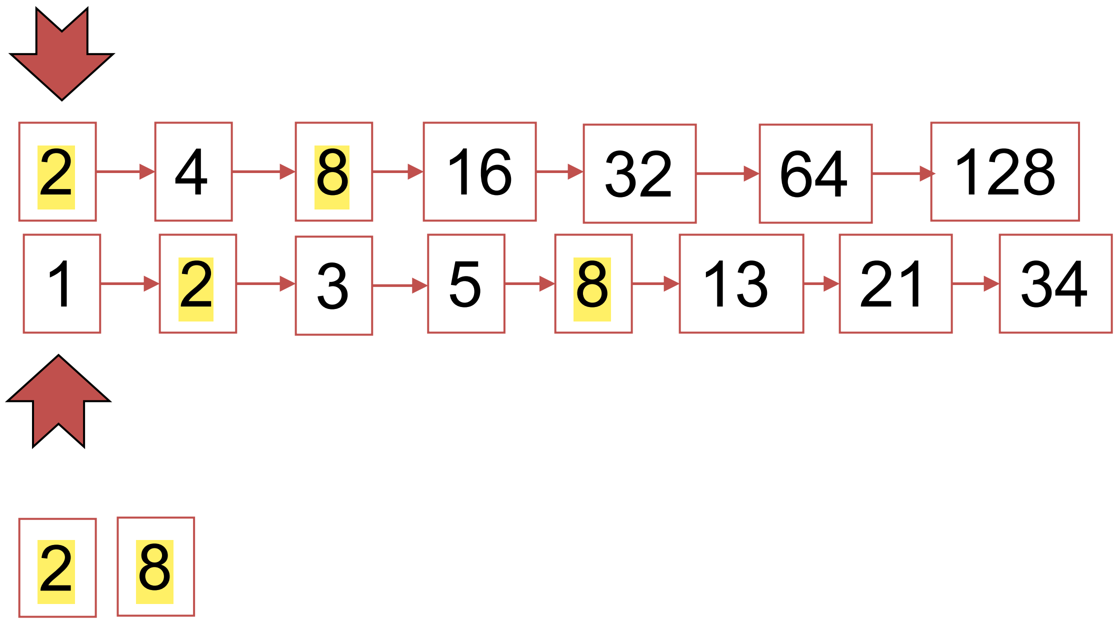
Merge of Unsorted Posting Lists

4	1	6	9	3	2		: len = x
5	3	4	7	9	8	1	: len = y

Time complexity = $O(xy)$

Merge of Sorted Posting Lists

- Walk through the two postings simultaneously, outputs the common doc IDs



Outputs:

Crucial: postings sorted by docID.

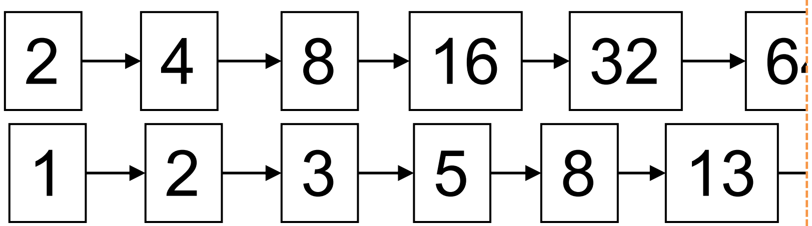
Intersecting two posting lists (a “merge” algorithm)

INTERSECT(p_1, p_2)

answer $\leftarrow \langle\ \rangle$
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
    then ADD($answer, docID(p_1)$)
        $p_1 \leftarrow next(p_1)$
        $p_2 \leftarrow next(p_2)$
    else if $docID(p_1) < docID(p_2)$
        then $p_1 \leftarrow next(p_1)$
        else $p_2 \leftarrow next(p_2)$
return *answer*

Time Complexity

- For the intersection of two
of lengths x and y,




```
INTERSECT(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
     then ADD(answer, docID(p1))
          p1 ← next(p1)
          p2 ← next(p2)
     else if docID(p1) < docID(p2)
            then p1 ← next(p1)
            else p2 ← next(p2)
return answer
```

Time complexity = $O(x + y)$

Exercise

- Write an algorithm to process an OR query
- **or** (p_1, p_2)
  - answer $\leftarrow$ < >
  - while p_1 is not null and p_2 is not null
    - if docID(p_1) < docID(p_2)
      - Add(answer, p_1)
      - $p_1 \leftarrow$ next(p_1)
    - else if docID(p_1) > docID(p_2)
      - Add(answer, p_2)
      - $p_2 \leftarrow$ next(p_2)
    - else
      - Add(answer, p_1)
      - $p_1 \leftarrow$ next(p_1)
      - $p_2 \leftarrow$ next(p_2)

Exercise

- Write an algorithm to process an AND-NOT query
- **and-not** (p_1, p_2)
  - answer ← < >
  - while p_1 is not null and p_2 is not null
    - if docID(p_1) < docID(p_2)
      - Add(answer, p_1)
      - p_1 ← next(p_1)
    - else if docID(p_1) > docID(p_2)
      - p_2 ← next(p_2)
    - else
      - p_1 ← next(p_1)
      - p_2 ← next(p_2)
  - while p_1 is not null
      - Add(answer, p_1)
      - p_1 ← next(p_1)

N-Way Merge

- Remind that
  - We have merged multiple runs more than two at a time using n-way merge in external mergesort
- N-way merge in intersection
  - A merge of runs in external sort aggregates all runs and the total length is not reduced
  - But intersection may produce <u>a much shorter output</u> → smaller intermediate runs to be materialized on disk

Binary vs. N-Way Merges

- N-way merge
  - Beneficial when the merged list are still so large that it should be written temporary on disk
- Binary merge
  - Advantageous if a binary merge can reduce the intermediate result enough to be maintained in main memory

PHRASE QUERIES AND POSITIONAL INDEXES

Phrase queries

- We want to be able to answer queries such as ***"Hanyang university"*** – as a phrase
- Thus, the sentence *"I went to university near Hanyang high school"* is not a match.
  - The concept of phrase queries has proven easily understood by users; one of the few "advanced search" ideas that works
  - Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only <*term* : *docs*> entries

Solution 1: Biword Indexes

- Index every consecutive pair of terms in the text as a phrase
- For example, the text "Friends, Romans, Countrymen" would generate the biwords
  - ***friends romans***
  - ***romans countrymen***
- Each of these biwords is now a dictionary term
- Two-word phrase query-processing is now immediate.

Longer Phrase Queries

- Longer phrases can be processed by breaking them down
- ***Hanyang university at Ansan*** can be broken into the Boolean query on biwords:

*“**Hanyang university**” AND “**university at**” AND “**at Ansan**”*

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

Can have false positives!

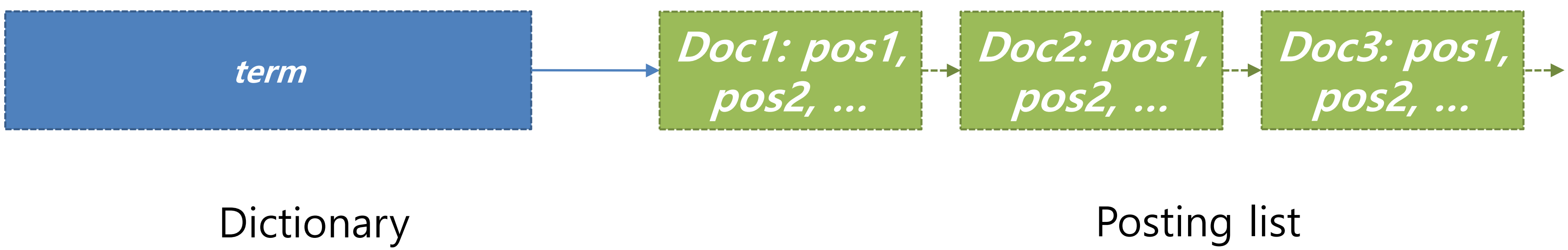
Issues For Biword Indexes

- False positives, as noted before
- Index blowup due to bigger dictionary
  - Infeasible for more than biwords, big even for them

- Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy

Solution 2: Positional indexes

- In the postings, store, for each ***term*** the position(s) in which tokens of it appear:



Example: Positional Inverted List

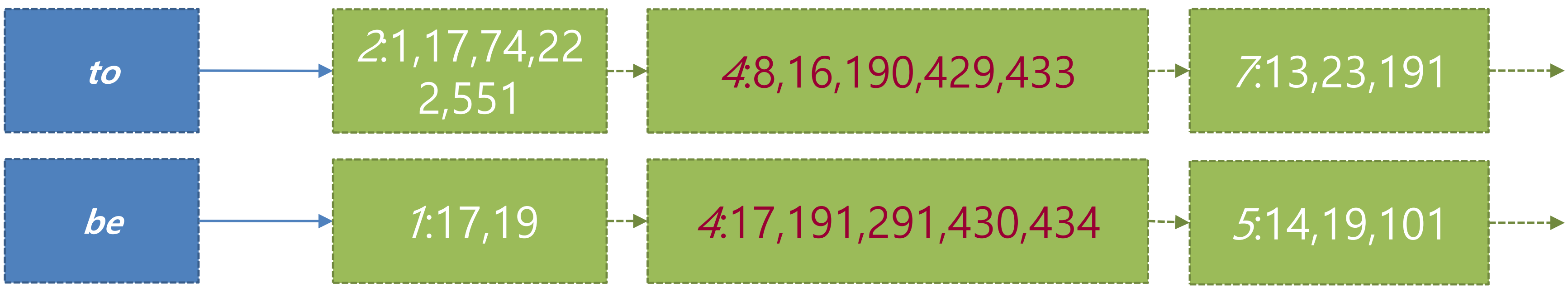
be → *1*: 7, 18, 33, 72, 86, 23 → *2*: 3, 149 → *4*: 17, 191, 291, 430, 434 → *5*: 363, 367

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality

Processing a phrase query

- Extract inverted index entries for each distinct term: ***to, be, or, not.***
- Merge their *doc:position* lists to enumerate all positions with "***to be ~~or not~~ to be***".



- Same general method for proximity searches

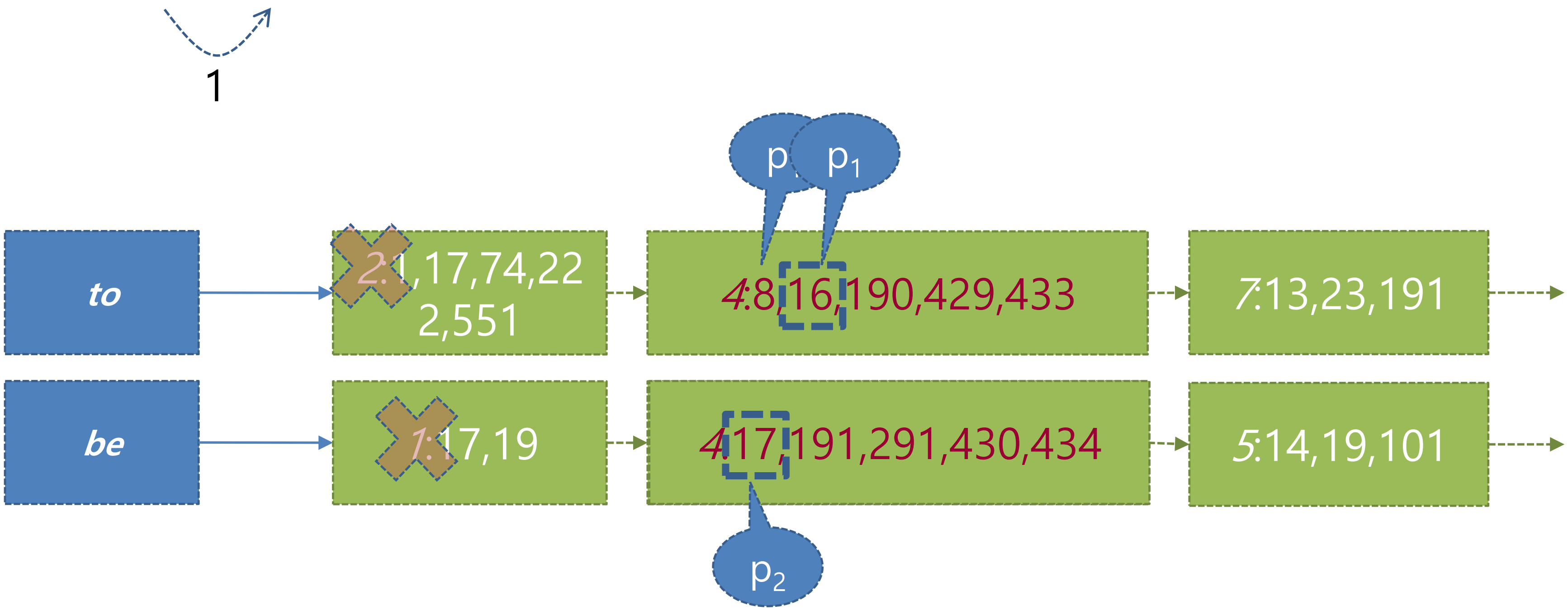
Processing a phrase query with two keywords

- "*to be*".

1

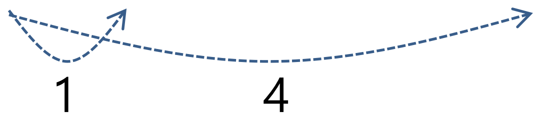
Processing a phrase query with two keywords

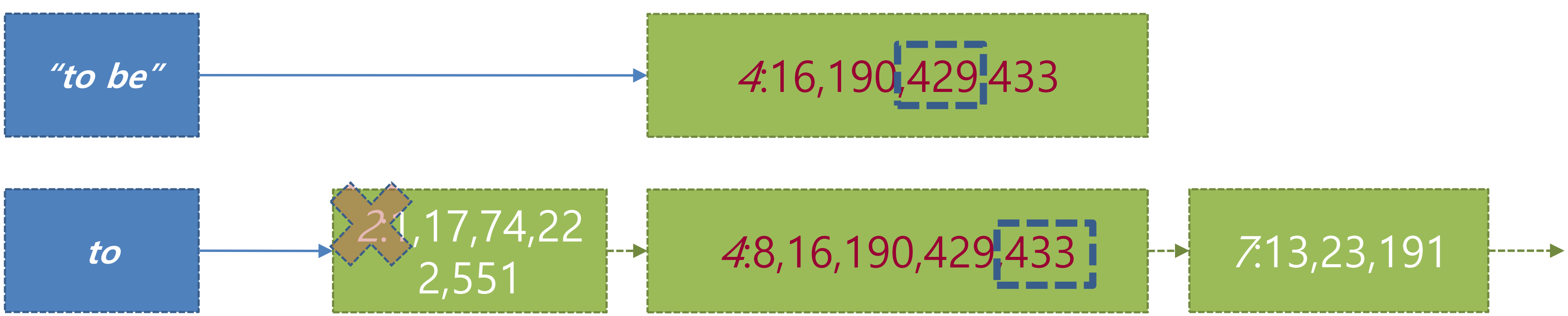
- "***to be***".



Processing a phrase query with multiple keywords

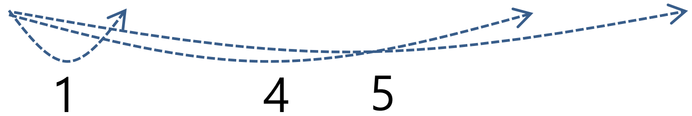
- "***to be ~~or not~~ to***".





Processing a phrase query with multiple keywords

- "***to be ~~or not~~ to be***".



Exercise

- Write a pseudocode for processing phrase query with two terms (distanced by d)
- **proximity** (p_1, p_2, d)
  - // p_1, p_2 : cursor on each posting list
  - // d: the distance of terms for p_1 and p_2 in query
  - // DocID(p_1) : doc. ID pointed by the current cursor
  - // Pos(q_1) : position pointed by a secondary cursor

Exercise

- **proximity** (p_1, p_2, d)
  - answer ← <>
  - while p_1 is not null and p_2 is not null
    - if docID(p_1) < docID(p_2)
      - p_1 ← next(p_1)
    - else if docID(p_1) > docID(p_2)
      - p_2 ← next(p_2)
    - else
      - q_1 ← init(p_1), q_2 ← init(p_2)
      - While q_1 is not null and q_2 is not null
        - » If Pos(q_1) + d < Pos(q_2) then q_1 ← next(q_1)
        - » Else if Pos(q_1) + > Pos(q_2) then q_2 ← next(q_2)
        - » Else Add(answer, docID(p_1)) and exit while loop

Positional Index Size

- A positional index expands postings storage *substantially*
  - Even though indices can be compressed

Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries … whether used explicitly or implicitly in a ranking retrieval system.

Rules of Thumb

- A positional index is 2–4 times as large as a non-positional index
- Positional index size 35–50% of volume of original text
  - – Caveat: all of this holds for "English-like" languages

QUERY PROCESSING OPTIMIZATION

Query optimization

- What is the **best order** for query processing?
- Consider a query that is an AND of **n** terms.
- For each of the n terms, get its postings, then AND them together.

Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	16	21	34
Calpurnia	13	16						

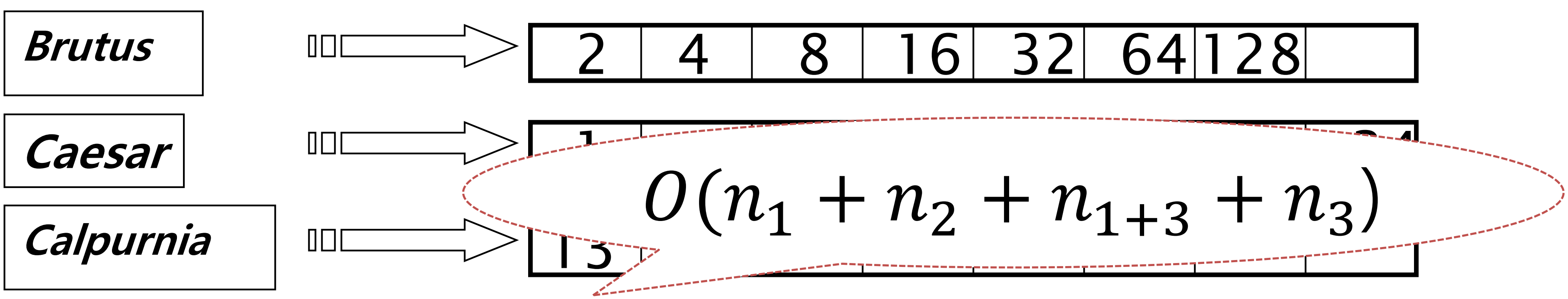
Query: ***(Brutus*** *AND* ***Caesar)*** *AND* ***Calpurnia***

$$O(n_1 + n_2 + n_{1+2} + n_3)$$

Query optimization example

- Process in order of increasing freq:
  - *Heuristic: start with smallest set, then keep cutting further.*

This is one of the reasons we kept document freq. in dictionary



Execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***.

More general optimization

- e.g., *(**madding** OR **crowd**) AND **ignoble***

x y z

- Optimization algorithm
  - Get doc. freq.'s for all terms.
  - Estimate the size of each *merge* by the sum of its doc. freq.'s (conservative).
  - Process in increasing order of *posting list* sizes.

$$O(x + y + \boxed{?} + z)$$

More general optimization

- e.g., *(**madding** AND **ignoble**) OR (**crowd** AND **ignoble**)*

x

y

z

- Optimization algorithm
  - Enumerate all equivalent forms with the given query
  - Choose the one with smallest cost

$O($? $)$

Better if $\max(x, y) > z + \min(x, z) + \min(y, z)$

Optimizing

- Limitation
  - Simple Boolean operation set
  - Simple summary information about a posting lists such as size and min / max doc IDs
- Optimization
  - Simple heuristic only possible
  - What if there is an advanced method to estimate the size of intersection more exact? ➔ the order of merges really matters!

Exercise

- Find an optimal order for
  - Hanyang AND University AND ERICA
- Given information

Term	Freq.	Min DocID	Max DocID
Hanyang	100	1	115
University	290	89	782
ERICA	40	3	456

Intersection Size Estimation

- Given two terms t_1 and t_2
  - Minimum of
    - min $(\mathrm{freq}(t_1), \mathrm{freq}(t_2))$
    - min $(\mathrm{MaxDocID}(t_1), \mathrm{MaxDocID}(t_2))$ – max $(\mathrm{MinDocID}(t_1), \mathrm{MinDocID}(t_2)) + 1$

<table>
<tr><th>Term</th><th>Freq.</th><th>Min DocID</th><th>Max DocID</th></tr>
<tr><td>Hanyang</td><td>100</td><td>1</td><td>115</td></tr>
<tr><td>University</td><td>290</td><td>89</td><td>782</td></tr>
<tr><td rowspan="3">Estimate size</td><td rowspan="2">Min = 100</td><td>Max = 89</td><td>Min = 115</td></tr>
<tr><td colspan="2">115-89+1 = 27</td></tr>
<tr><td colspan="3">Min(100, 87) = 27</td></tr>
<tr><td>Min DocID</td><td>89</td><td>Max DocID</td><td>115</td></tr>
</table>

Intersection Size Estimation

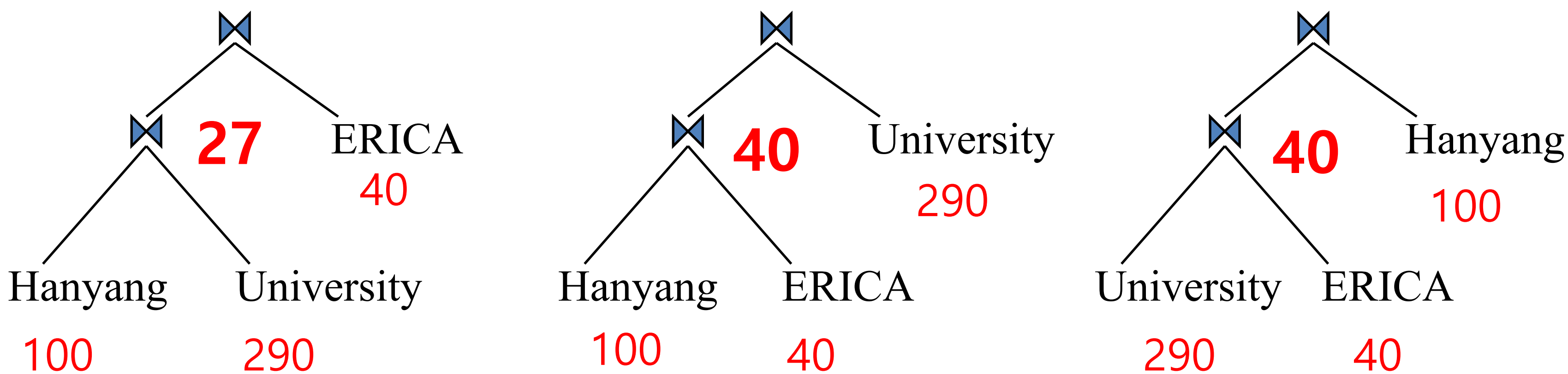
<table>
<tr><th>Term</th><th>Freq.</th><th>Min DocID</th><th>Max DocID</th></tr>
<tr><td>Hanyang</td><td>100</td><td>1</td><td>115</td></tr>
<tr><td>ERICA</td><td>40</td><td>3</td><td>456</td></tr>
<tr><td rowspan="3">Estimate size</td><td rowspan="2">Min = 40</td><td>Max = 3</td><td>Min = 115</td></tr>
<tr><td colspan="2">115-3+1 = 113</td></tr>
<tr><td colspan="3">Min(40, 113) = 40</td></tr>
<tr><td>Min DocID</td><td>3</td><td>Max DocID</td><td>115</td></tr>
</table>

Intersection Size Estimation

<table>
<tr><th>Term</th><th>Freq.</th><th>Min DocID</th><th>Max DocID</th></tr>
<tr><td>University</td><td>290</td><td>89</td><td>782</td></tr>
<tr><td>ERICA</td><td>40</td><td>3</td><td>456</td></tr>
<tr><td rowspan="3">Estimate size</td><td rowspan="2">Min = 40</td><td>Max = 89</td><td>Min = 456</td></tr>
<tr><td colspan="2">456-89+1 = 368</td></tr>
<tr><td colspan="3">Min(40, 368) = 40</td></tr>
<tr><td>Min DocID</td><td>89</td><td>Max DocID</td><td>456</td></tr>
</table>

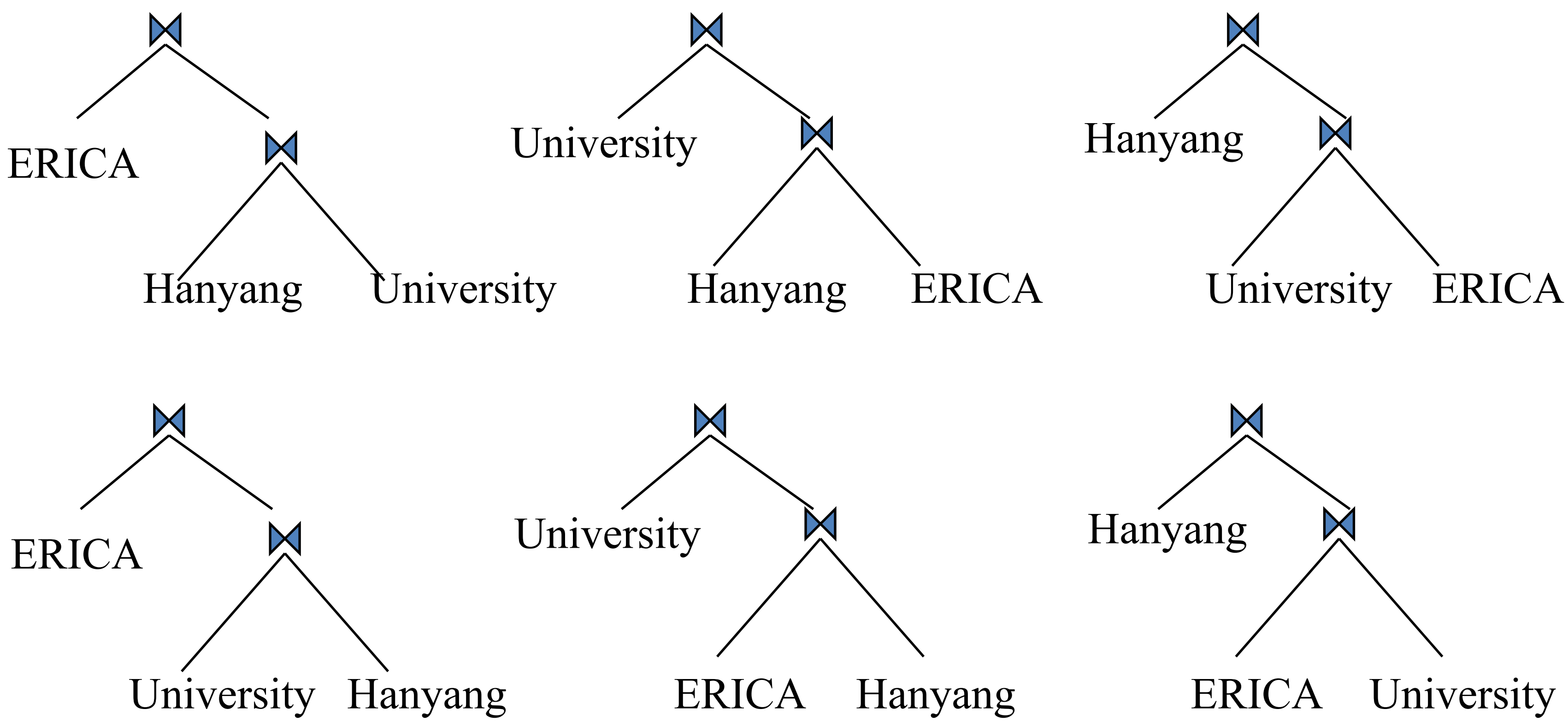
Query Plan Trees

- Repeating two intermediate results must be materialized by one of following trees:



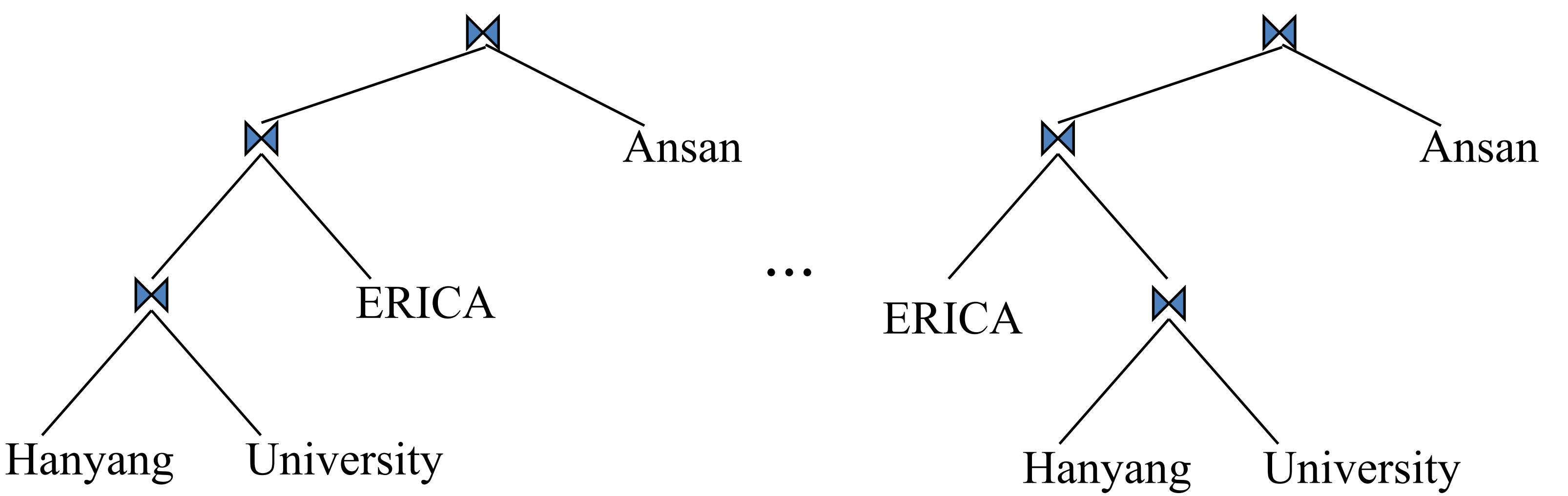
Query Plan Trees

- Repeating two intermediate results must be materialized by one of following trees:



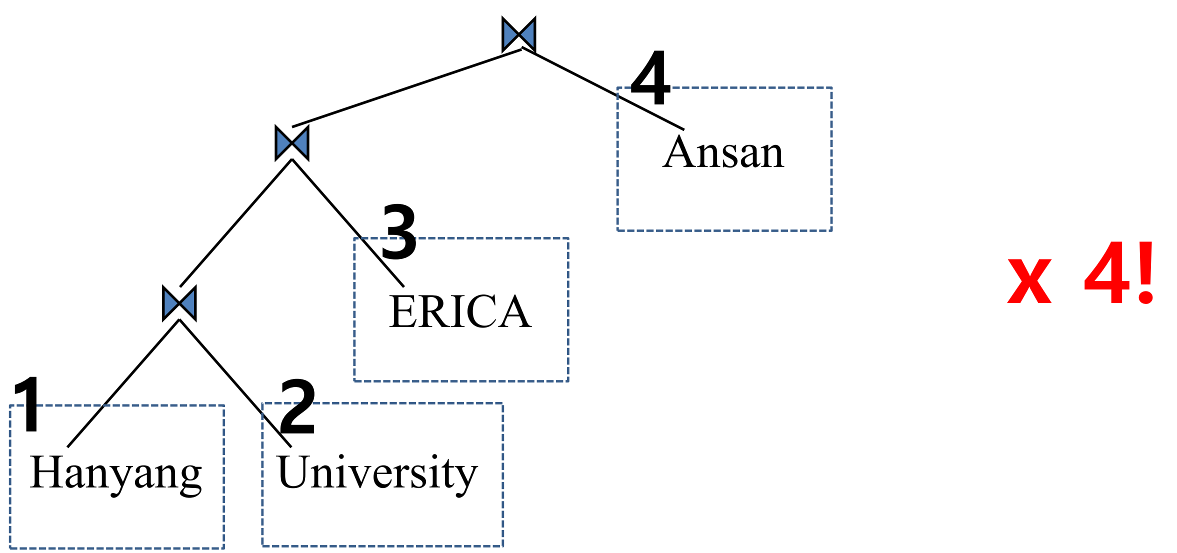
Query Plan Trees

- How many query plan trees are there for
  - Hanyang AND University AND ERICA AND Ansan?
- Let us
  - Count all different query trees without considering equivalents



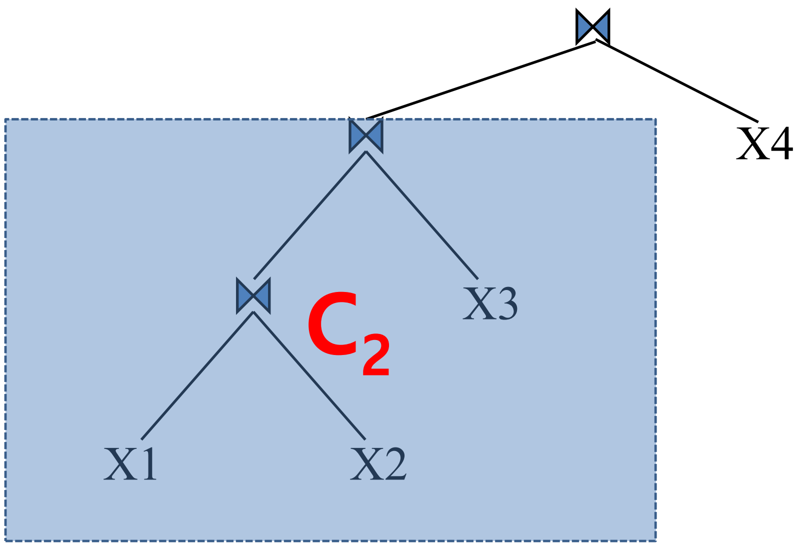
Query Plan Trees

- 1. Enumerate all possible trees
- 2. For each tree, put n terms with all leaf nodes for all possible permutations



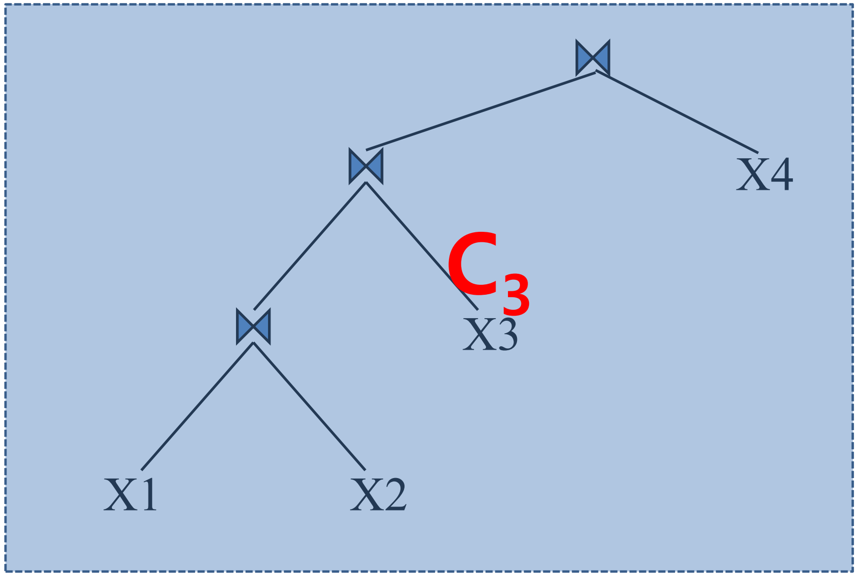
Query Plan Trees

- Let
  - C_k: the number of all possible trees for merging k+1 posting list by k merge operations



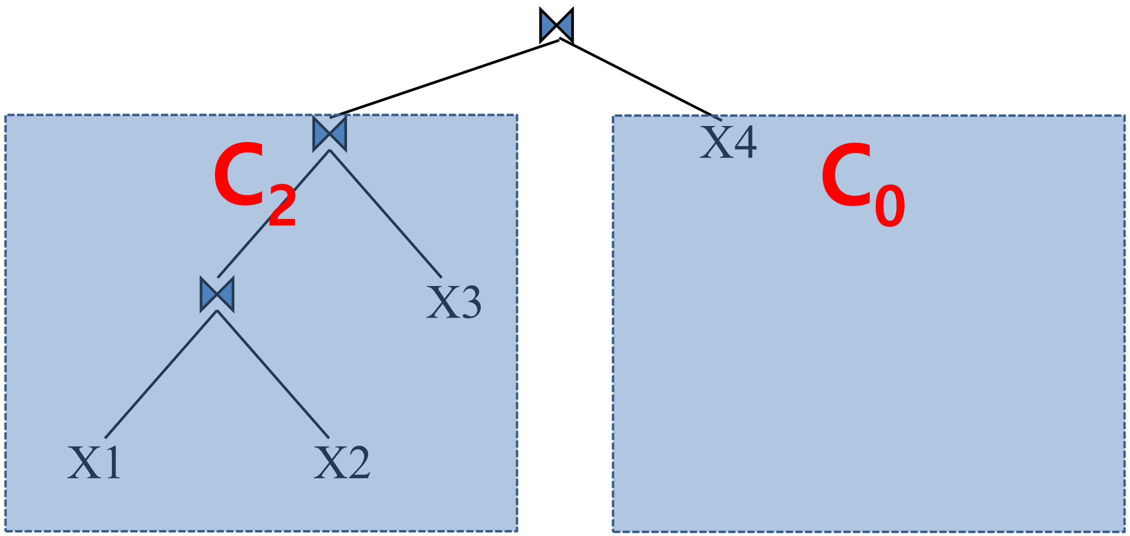
Query Plan Trees

- Let
  - C_k: the number of all possible trees for merging k+1 posting list by k merge operations



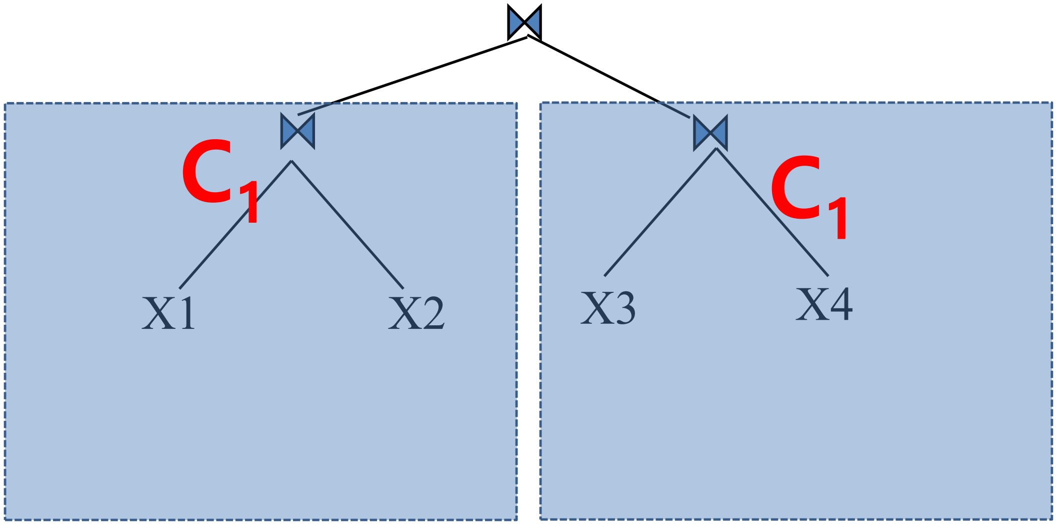
Query Plan Trees

- $C_3 = C_2 + C_0 +$



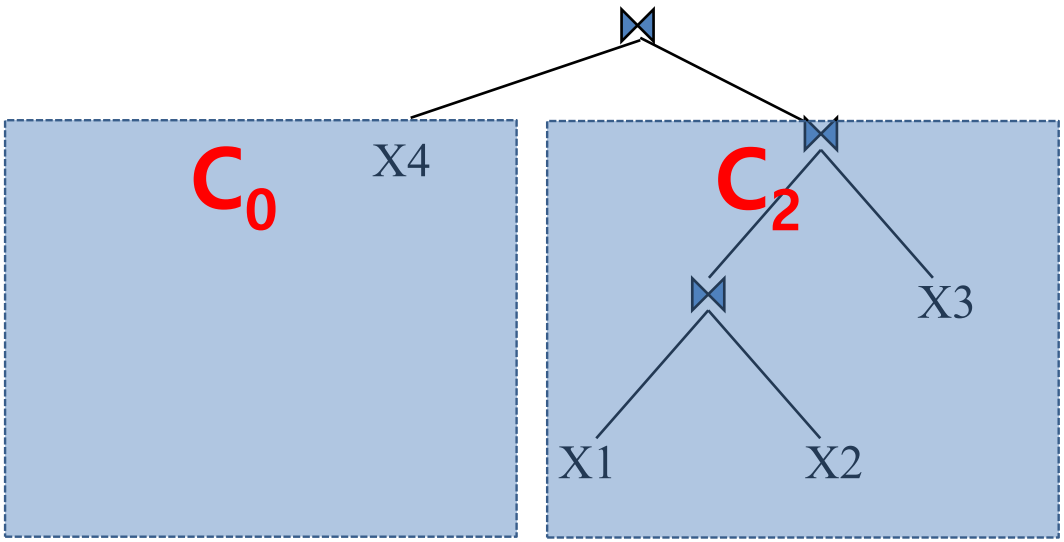
Query Plan Trees

- $C_3 = C_2 + C_0 + C_1 + C_1 +$



Query Plan Trees

- $C_3 = C_2 + C_0 + C_1 + C_1 + C_0 + C_2$



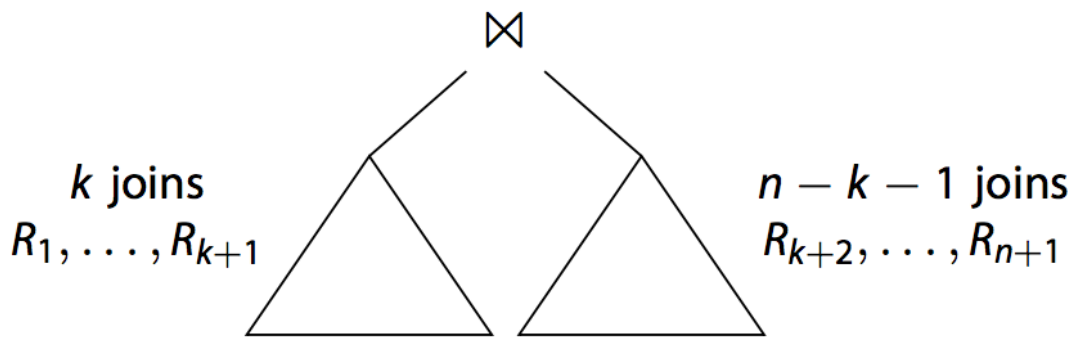
Number of Query Plan Trees

- Consider finding the best merge-order for n posting lists r_1 r_2 . . . r_n
- There are $(2(n-1))!/(n-1)!$ different merge orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!

How Many Such Combinations Are There?

Slide from https://db.inf.uni-tuebingen.de/staticfiles/teaching/ws1011/db2/db2-optimization.pdf

- A join over $n + 1$ relations $R_1, \dots, R_{n+1}$ requires n **binary joins.**
- Its **root-level operator** joins sub-plans of k and $n - k - 1$ join operators ($0 \leqslant k \leqslant n - 1$):



- Let C_i be the **number of possibilities** to construct a binary tree of i inner nodes (join operators):

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} \quad .$$

Catalan Numbers

This recurrence relation is satisfied by **Catalan numbers**:

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1} = \frac{(2n)!}{(n+1)!n!} \ ,$$

describing the number of ordered binary trees with $n + 1$ leaves.

For **each** of these trees, we can **permute** the input relations (why?) $R_1, \ldots, R_{n+1}$, leading to:

Number of possible join trees for an $(n + 1)$-way relational join

$$\frac{(2n)!}{(n+1)!n!} \cdot (n+1)! = \frac{(2n)!}{n!}$$

Cost-Based Optimization

- Consider finding the best merge-order for n posting lists r_1 r_2 . . . r_n
- There are $(2(n-1))!/(n-1)!$ different merge orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Dynamic Programming in Optimization

- To find best merge tree for a set of n relations:
  - To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S
  - Recursively compute costs for merging subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 1$ alternatives
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of re-computing it

Optimization Algorithm

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
            return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
            set bestplan[S].plan and bestplan[S].cost based on the best way
            of accessing S (= size of S)
     else for each non-empty subset S1 of S such that S1 ≠ S
           P1= findbestplan(S1)
           P2= findbestplan(S - S1)
           A = best algorithm for merging results of P1 and P2
           cost = P1.cost + P2.cost + cost of A
           if cost < bestplan[S].cost
                     bestplan[S].cost = cost
                     bestplan[S].plan =  "execute P1.plan; execute P2.plan;
                                          join results of P1 and P2 using A"
    return bestplan[S]
```

Optimization Algorithm

```
procedure findbestplan(S)                                    T(n)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on the best way
        of accessing S (= size of S)                         k = |S1| = 1, ..., n-1
    else for each non-empty subset S1 of S such that S1 ≠ S
T(k)        P1= findbestplan(S1)
T(n-k)      P2= findbestplan(S - S1)
            A = best algorithm for merging results of P1 and P2
            cost = P1.cost + P2.cost + cost of A
            if cost < bestplan[S].cost
                bestplan[S].cost = cost
                bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                    join results of P1 and P2 using A"
    return bestplan[S]
```

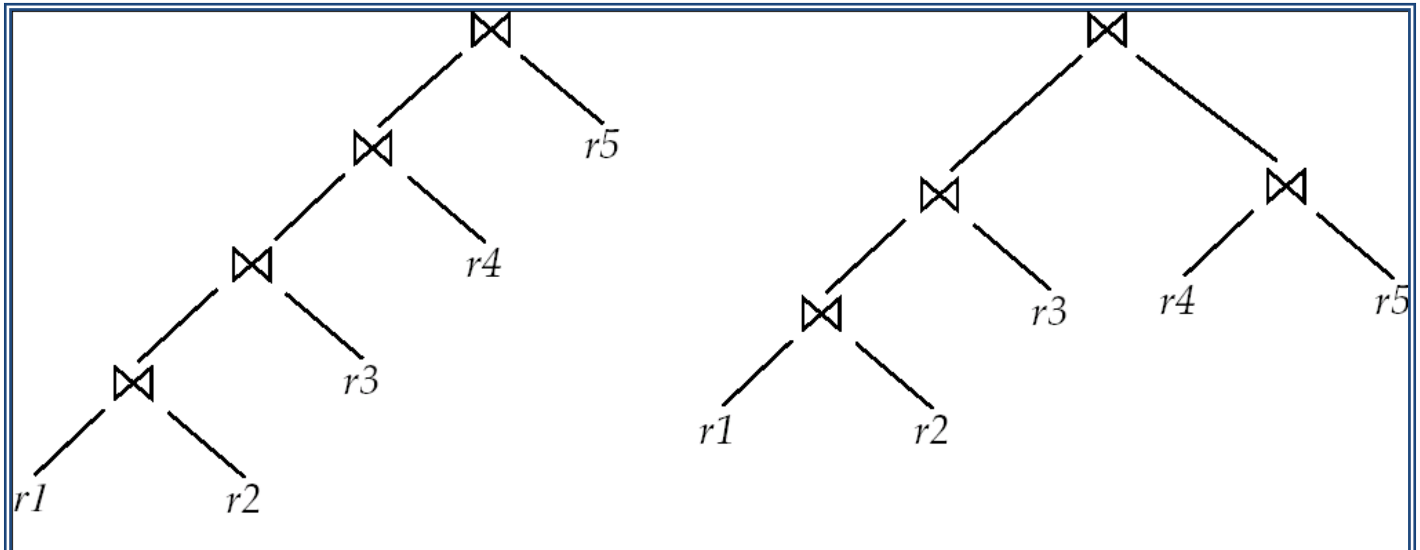
$$T(n) = \sum_{k=0}^{n} \binom{n}{k} 2^k 1^{n-k} = 3^n$$

Cost of Optimization

- With dynamic programming, time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!
  - time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input
- Space complexity is $O(2^n)$
  - space complexity = amount of memory an algorithm needs

Left Deep Trees

- In **left-deeptrees,** the right-hand-side input for each join is a relation, not the result of an intermediate merge



(a) Left-deep join tree (b) Non-left-deep join tree

See http://iggyfernandez.wordpress.com/2010/11/27/sql-101-deep-left-trees-deep-right-trees-and-bushy-trees-oh-my/ for a discussion of tree types and optimization

Left-Deep Merge Optimization Algorithm

procedure <u>findbestplan</u>(*S*)
    if (*bestplan*[*S*].*cost* $\neq \infty$)
        return *bestplan*[*S*]
    // else *bestplan*[*S*] has not been computed earlier, compute it now
    if (*S* contains only 1 relation)
        set *bestplan*[*S*].*plan* and *bestplan*[*S*].*cost* based on the best way of accessing *S (= size of S)*
    else for each non-empty subset *S*1 of *S* such that |*S*1| = 1
        P1= findbestplan(*S*1)
        P2= findbestplan(*S* - *S*1)
        A = best algorithm for merging results of *P*1 and *P*2
        cost = *P*1.*cost* + *P*2.*cost* + cost of *A*
        if *cost* < *bestplan*[*S*].*cost*
            bestplan[*S*].*cost* = cost
            bestplan[*S*].*plan* = "execute *P*1.*plan*; execute *P*2.*plan*; join results of *P*1 and *P*2 using *A*"
    return *bestplan*[*S*]

Cost of Optimization

- To find best left-deep join tree for a set of *n* relations:
  - Consider *n* alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Using (recursively computed and stored) least-cost join order for each alternative on left-hand-side, choose the cheapest of the *n* alternatives.
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)