Query Optimization

Algebra Revisited

Tuple: [name: "Sokrates", age: 69]
Schema: {(name, string), (age, number)} ← $\mathcal{A}(t)$ ≡ columns

Concatenation : $t_1 \circ t_2$
Projection : $t.a$ or $t_{|A}$
Relations : set of tuples with the same schema
Set : no duplicates
Multi-set / Bag : can contain duplicates
Set operations : $\cup, \cap, \setminus$

Free variables $\mathcal{F}(e)$: age = 62 → 'age' is a free variable

Selection : $\sigma_p(R)$
Projection : $\Pi_A(R)$
Rename : $\rho_{a \to b}(R)$
Join : $\bowtie, \times$

Logical Query Optimization

- need a cost function

Phases :

1: break conjunctive selection predicates – $\sigma_A(\sigma_B(x))$ instead of $\sigma_{A \cap B}$
2: push selections down – reduce the size of the items
3: introduce joins
4: determine join order – best join order based on cost
5: introduce and push down projections – consider only relevant variables

Physical Query Optimization

- cost calculations
- add more execution info to the plan
- scan + selection
  - ↳ index vs. no index trade-offs

Operator selection

- logical operator "$\bowtie$" → physical operator "$\bowtie_{HH}$"

$\bowtie^{BNL}$ better than $\bowtie^{NL}$
$\bowtie^{SM}$ and $\bowtie^{HH}$ are usually better than both above (NL is optimal if left side has only one tuple)
$\bowtie^{HH}$ is often the best

- $\bowtie^{SM}$ requires sorted elements
- Materializing can be a good idea — temp operator stored in disk
  - ↳ first pass expensive, further passes cheap

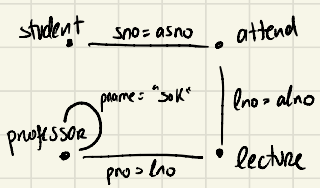
Note: Usually, $\bowtie^{HH}$ is better than $\bowtie^{SM}$ — only use $\bowtie^{SM}$ when data is much larger than memory (data ≈ mem size?)

Outlook

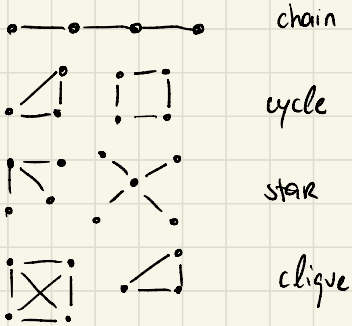
- separation in two phases looses optimality
- many decisions
- textbook physical optimization is incomplete
- did not discuss cost calculations

Join Ordering

Query Graph



Shapes



Join Trees:

- inner nodes: operators (e.g. join, cross product)
- leaves: relations
- shapes:
  - linear (left-deep, right-deep, zigzag)
  - bushy
- desired shape influences size of the search space
  - with cross-products: num tree shapes × num leaf permutations
  - without cross-products: depends on the query graph's shape

Selectivity, Cardinality

$$\boxed{f_p = \frac{|\sigma_p(R)|}{|R|}} \qquad \boxed{f_{i,j} = \frac{|R_i \bowtie R_j|}{|R_i \times R_j|}}$$

- cardinality assumes independence of predicates → not always true

$$\left(\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}\right) |T_1| \, |T_2|$$

- Independence: Data distributions of different columns are independent
- Uniformity: Values in a column are distributed uniformly
- Containment: Smaller set is contained within the larger set

selectivity of $\sigma_{R_1.x=c}$: if x is the key : $\frac{1}{|R_1|}$

if not : $\frac{1}{|R_1.x|}$ ↳ $D(R_1.x)$ = distinct values of x

selectivity of $\bowtie_{R_1.x=R_2.y}$: x and y keys : $\frac{1}{\max(|R_1|,|R_2|)}$

x is key : $\frac{1}{|R_1|}$

x and y not keys : $\frac{1}{\max(|R_1.x|,|R_2.y|)}$

Costs

$$C_{out}(R) = 0$$

$$C_{out}(R_i \bowtie R_j) = |R_i \bowtie R_j| + \underbrace{C_{out}(R_i)}_{\text{left side}} + \underbrace{C_{out}(R_j)}_{\text{right side}}$$

- sums up the costs of the (intermediate) results
- we ignore costs of single relations

More cost functions:

nested loop join

$$C_{nlj}(e_1 \bowtie e_2) = |e_1|\,|e_2|$$

$$C_{hj}(e_1 \bowtie e_2) = 1.2\,|e_1|$$

$$C_{smj}(e_1 \bowtie e_2) = |e_1| \log(|e_1|) + |e_2| \log(|e_2|)$$

Properties of C_{out}:

- symmetry : $C(A \bowtie B) = C(B \bowtie A)$
- ASI : rank function r such that $r(A \cup B) \leq r(A \cup B)$ $\Leftrightarrow$ $C(A \cup B) \leq C(A \cup B)$

  ↳ adjacent sequence interchange

Search Space

Catalan Numbers

- number of binary tree nodes with n leave nodes : $C(n-1)$

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

Number of Join Trees :

with cross products :

- left / right deep - $n!$
- zig-zag - $n!\, 2^{n-2}$
- bushy - $n!\, C(n-1) = \frac{(2n-2)!}{(n-1)!}$

rational : number of leaf combinations $(n!)$ × number of unlabeled trees

chain, no cross-products :

- left / right deep - 2^{n-1}
- bushy - $2^{n-1}\, C(n-1)$

star, no cross-products :

- left / right deep - $2 \times (n-1)!$
- zig-zag - $2 \times (n-1)! \times 2^{n-2}$
- bushy - n

Algorithms

Greedy Heuristics

- Linear (left-deep / right-deep / zig-zag) join tree
  - GJO-1 : start with smallest R_i. Pick next R_i with smaller cardinality
    $\arg\min_{R_i \in R} w(R_i)$ → doesn't look at intermediate results
    only left-deep
  - GJO-2 : start with smallest R_i. Pick next R_i that minimizes tree cardinality
    $\arg\min_{R_i \in R} w(R_i, S)$
    only left-deep, compute relative weights
  - GJO-3 : GJO-2, but try every start relation

Greedy Operator Ordering (GOO)

can pick any edge

- greedily pick edges such that the intermediate result is minimized
- merge nodes connected by the picked edge
- calculate cardinality of merged node
- calculate selectivities of collapsed edges (product of individual selectivities)
- can produce bushy trees.

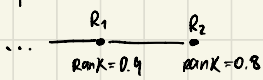
$$(T_i, T_j) = \arg\min_{(T_i \in T,\, T_j \in T),\, T_i \neq T_j} |T_i \bowtie T_j|$$

- bottom-up, small trees combined into larger trees

IKKBZ

- requires acyclic query graph
- requires cost function with ASI property
- generates optimal left deep tree for acyclic queries in polynomial time

- for each relation R in the query
  - build the precedence graph rooted in R_i
  - find subtree whose children are chains
  - build compound relations to eliminate contradictory sequences (normalize)



$\text{rank}(R_1) > \text{rank}(R_2) \longrightarrow$ create $R_{1,2}$

  - merge chains (ascending in rank)
  - repeat until the whole tree is a chain
  - denormalize previously normalized compound relations
- pick the cheapest of all generated sequences

Notation:

cardinality n_i

selectivity s_i

cost $C(R_i) = n_i s_i$ (0 if R_i is the root)

rank $R_i = \dfrac{T(R_i) - 1}{C(R_i)} = \dfrac{n_i s_i - 1}{n_i s_i}$

Moreover: $C(S_1 S_2) = C(S_1) + T(S_1) C(S_2)$

$T(S) = \prod_{R_i \in S} (s_i n_i)$

rank: $R(S) = \dfrac{T(S) - 1}{C(S)}$

Query graph has cycles

1. answer to query subgraph is superset of the answer for original query
2. very selective join → likely to be influential

Build MST

MST

- Kruskal algorithm: pick edges with lowest selectivity until every node is connected

- Union-Find: each set as a representative member (parent)
  find(x) finds the parent
  union(a,b) merges the sets a and b
  (in case rank(a) = rank(b), b is parent and rank(b)++)

Maximum Value Precedence (MVP)

- heuristic: prefer to perform joins that reduce the input size of expensive the most
- algorithm builds an effective spanning tree in the weighted direct join graph (edges and nodes have weights)

  - physical edge: $w_{u,v} = \frac{|\bowtie_u|}{|R(u) \cap R(v)|}$
  - virtual edge: $w_{u,v} = 1$
  - node: $w(p_{i,j}, S) = \frac{|\bowtie^S_{p_{i,j}}|}{|R_i \bowtie_{p_{i,j}} R_j|}$

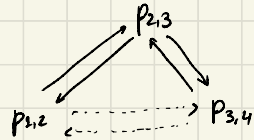
$R(u) = \{R_1, R_2\}$ $R(v) = \{R_2, R_3\}$

$$w_{u,v} = \frac{|\bowtie_u|}{|u \sqcap v|} = \frac{|R_1 \bowtie R_2|}{|R_2|} = f_{1,2}|R_1|$$

- take edges with weight < 1 (reduce work for later operators)
- add remaining edge (increase input sizes as late as possible)

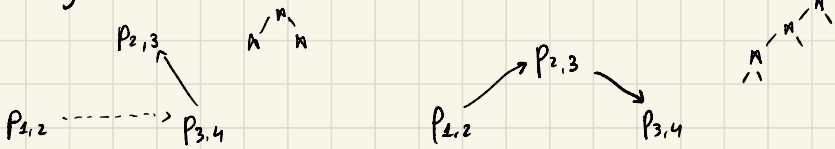
$R_1 - R_2 - R_3 - R_4$

query graph



directed join graph

possible (effective) spanning trees:

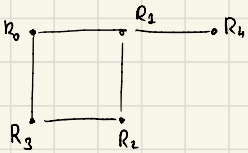


Dynamic Programming

Optimality principle: ~~If subtrees are optimal, full tree is optimal~~
If full tree is optimal, then subtrees <u>must</u> be optimal

Overview:
- generate optimal join trees <u>bottom</u> up
- start from size one
- construct larger trees from optimal smaller ones
- try all combinations that might be optimal
- different possibilities to enumerate sets of relations

▶ <u>DPsize</u>: enumerate sets in ascending order
1 element → 2 elements → ...
- cannot produce <u>bushy trees</u>



start from 1 relation at the left and go until $\left\lfloor \frac{\text{level}}{2} \right\rfloor$ to avoid <u>duplicates</u>.

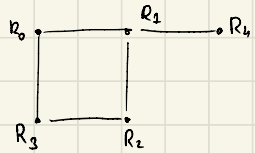
1: 0, 1, 2, 3, 4
2: 0|1[1], 0|3[2], 1|2[3], 1|4[4], 2|3[5] ✓
3: 0|12[6], 0|14[7], 0|23[8], 1|03[9], 1|23[10], 2|01[11], 2|14[12], 2|03[13], 3|01[14], 3|12[15], 4|01[16], 4|12[17] ✓
~~01|2~~ ~~01|3~~ ~~01|4~~ ~~03|1~~ ~~03|2~~ ~~12|0~~ ~~12|3~~ ~~12|4~~

...

- **DPsub**: enumerate in integer order (exponential)

Integer order:

000 - {}
001 - {R_1}
010 - {R_2}
011 - {R_1, R_2}
...
01001 - {R_1, R_4}



1: 0
10: 1
11: 0|1
100: 2
101: 0|2
110: 1|2
111: 0|12, 1|02, 01|2
(1, 10, 11)
...

1000: 3
1001: 0|3
1010: 1|3
1011: 0|13, 1|03, 01|3
1100: 2|3
1101: 0|23, 2|03, 02|3
1110: 1|23, 2|13, 12|3
1111: 0|123, 1|023, 01|23, 2|013, 02|13, 12|03, 3|012

Subset generation:

```
S1 = S & (-S)
do {
    S2 = S - S1
    S1 = S & (S1 - S) = S & (~S + S + 1)
```

selects last set bit

- enumerates all subsets except $\emptyset$ and S itself
- very fast

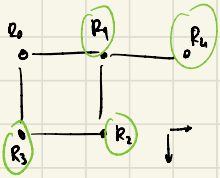
Asymptotic Search Space

	chains	cycles	stars	cliques	
DPsize	$O(n^4)$	$O(n^4)$	$O(4^n)$	$O(4^n)$	→ good for chains, very bad cliques
DPsub	$O(2^n)$	$O(n2^n)$	$O(3^n)$	$O(3^n)$	→ good for cliques, but adapts badly

- DPsize vs DPsub: Neither dominates the other ⇒ none is optimal!
- Both generate many pairs that are pruned away (connectness, overlap)
- Do not consider query graph
- Keep list of plans for each size

▶ **DPccp**: enumerate connected component complement pairs
→ only expand to subgraphs > current index

- Enumerate Csg + Enumerate Cmp produce all ccp
- DPccp considers exactly # ccp pairs
  ↳ lower bound for all DPs



- each connected subgraph - complement pair (ccp) can be joined
- adaps to the shape of the query graph
- Main problem: enumerating the pairs
  ↳ $O(1)$, at most $O(n)$

$\{4\}$: –
$\{3\}$: –
$\{2\}$: $\{3\}$
$\{2,3\}$: –
$\{1\}$: $\{4\}, \{2\}, \{2,3\}$
$\{1,2\}$: $\{4\}, \{3\}$
$\{1,2,3\}$: $\{4\}$
$\{1,4\}$: $\{2\}, \{2,3\}$
$\{1,2,4\}$: $\{3\}$
$\{1,2,3,4\}$: –

valid neighbors: $\{2\}, \{4\}$

here we consider $2>0$, not $2<3$ so this pair is valid.

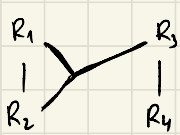
$\{0\}$: $\{3\}, \{2,3\}, \{1\}, \{1,2\}, \{1,2,3\}, \{1,4\}, \{1,2,4\}, \{1,2,3,4\}$ – ✓
$\{0,1\}$: $\{4\}, \{3\}, \{2\}, \{2,3\}$
...

Complexity:	chains	cycles	stars	cliques
DPccp	$O(n^3)$	$O(n^3)$	$O(n2^n)$	$O(3^n)$ ←
DPsize	$O(n^4)$	$O(n^4)$	$O(4^n)$	$O(4^n)$
DPsub	$O(2^n)$	$O(n2^n)$	$O(3^n)$	$O(3^n)$

#ccp is lower bound for all DP enumeration algorithms
↳ minimal number of pairs

▶ **DPhyp**: handles hypergraphs (join predicates between more than two relations, reordering constraints for non inner-joins, graph simplification

Hypergraph



- Problems: DPccp no longer optimal!
  - must still guarantee DP order
  - temporaly disconnected graphs

Memoization

- recursive top-down approach
- memoize already generated trees to avoid duplicate work
- might be faster / more efficient, as more knowledge allows for more pruning
- usually slower than DP
- RS-1 – commutativity, right-associativity, left-associativity
- RS-2, RS-3 slightly modify/add to RS-1

• Algorithm calls if $B[S] = \varepsilon$ A LOT of times recursively

---

Non-Inner Joins:

some operators are not freely reordable

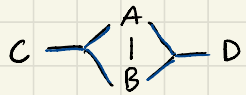
Extract reordering constraints from operator trees

1. build the syntactic eligibility set (SES) for each operator
2. bottom-up up transversal, building total eligibity set (TES)
   - initialize TES with SES
   - check for conflicts with other operators
   - if conflict, add other TES to own TES

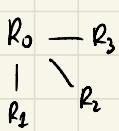
captures reordering restrictions

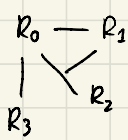
SES	TES
{B, D}	{A, B, D}
{A, C}	{A, B, C}
{A, B}	{A, B}

• Construct hyper-edges



Simplifying Query-Graph





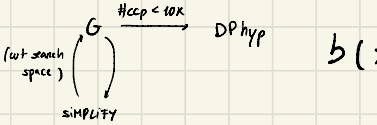
search space size

$\cancel{6}$ = 3!
3

joins: $R_0 \bowtie R_1$, $R_0 \bowtie R_2$, $R_0 \bowtie R_3$ $\longrightarrow$ $R_0 \bowtie R_1$, $\{R_0, R_1\} \bowtie R_2$, $R_0 \bowtie R_3$ $\longrightarrow$ Keep simplifying ...

Procedure:

1. Examine all joins $\bowtie_1, \bowtie_2 \in E$ that are neighboring (1 relation in common)
2. make sure $\bowtie_2$ could be ordered before $\bowtie_1$
3. compute ordering Benefit($\bowtie_1, \bowtie_2$)



$$b(\times \bowtie_1 R_1, \times \bowtie_2 R_2) = \frac{C((\times \bowtie_1 R_1) \bowtie_2 R_2)}{C((\times \bowtie_2 R_2) \bowtie_1 R_1)}$$

4. retain relations with maximal benefit
   if $b(R_1 \bowtie R_2, R_2 \bowtie R_3)$ is high, $R_2 \bowtie R_3$ should be done first!

Adaptative Optimization

- DP optimal but slow for large queries
- Small: DP
- Medium: LinDP (MST + IKKBZ + linearized DP)
- Large: GOO / linDP / DP

Small queries:
- $\leq 10K$ DP entries (chains: ≤ 100 relations ; cliques: < 14 relations)
- Run DPhyp
  - adapts to the query graph's structure
  - complete and minimal enumeration of all join orders
- Optimal plan guaranteed
- Optimization will be fast.

Medium queries: fast optimization → search space linearization

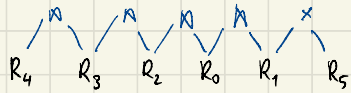
- Assume order of relations in the optimal plan is Known
- Optimally combine optimal solutions for subchains of increasing size
- How to Know the optimal order: IKKBZ
  Optimal left-deep in $O(n^2)$ ↵ → Perform n times $O(n)$ IKKBZ

– Final result is very close to optimal!

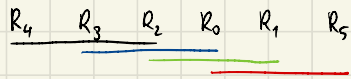
▸ LinDP : complexity $O(n^3)$

example :

	R_4	R_3	R_2	R_0	R_1	R_5
R_4	0	600	6400	24800	24800	267.020
R_3		0	400	1800	1820	19120
R_2			0	200	220	2420
R_0				0	20	1120
R_1					0	550
R_5						0

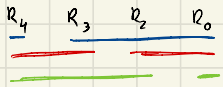


C = 600 400 200 20 550



- $C((R_4 \bowtie R_3) \bowtie R_2) = 600 + 6000 = 6600$
  $C(R_4 \bowtie (R_3 \bowtie R_2)) = 400 + 6000 = \underline{6400}$

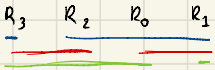
• • •



1. $C_{out} = \overbrace{1800}^{200+1600} + \frac{3}{10} \times 1600 \times 50 = \times 25.800$
2. $C_{out} = \overset{600}{50} + 200 + \frac{1}{5} \times \overset{600}{50} \times 200 = 24.800$

Vere ovima!

3. $C_{out} = \overbrace{6400}^{400+6000} + \frac{1}{5} \times 6000 \times 20 = 30.400$



1. $C_{out} = \overbrace{220}^{20+200} + \frac{1}{5} \times 200 \times 40 = 1820$
2. $C_{out} = 400 + 20 + \frac{1}{5} \times 400 \times 20 = \overset{2020}{2200}$
3. $C_{out} = \overbrace{1800}^{200+1600} \times \frac{1}{10} \times 1600 \times 10 = 3400$

• • •

Procedure :

1. Linearize using IKKBZ
2. Build best bushy plan for linearization

Properties :

- Runs in $O(n^3)$ — for $2 \le l \le n$, for start point, for split point
- Result at least as good as optimal left-deep plan
- With proper linearization, discovers optimal bushy plan

Large Queries :

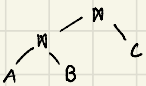
- even linearized DP too expensive
- Iterative Dynamic Programming
  - Greedily build query plan (GOO)
  - Optimize most expensive subtrees using DP

Transformative Approaches

- apply equivalences to initial join tree
- makes it easy to add new equivalences / rules (in theory)
- use memoization (keep all trees generated so far)
  Memo-structure: arguments are pointers to classes
- naive implementation generates a massive amount of duplicates
- duplicates can be avoided by disabling certain rules after a transformation has been applied (introduction of new rules becomes harder)
  ↳ RS-1, RS-2, RS-3

example: (RS-1)

- Memo structure



Class	Initialization	Transformation	Step
{A, B, C}	{A, B} $\bowtie_{111}$ C	C $\bowtie_{000}$ {A, B}	2
		A $\bowtie_{100}$ {B, C} (right associativity)	3 -
		B $\bowtie_{100}$ {A, C} (right associativity)	4 -
		{B, C} $\bowtie_{000}$ A (left associativity)	6
		{A, C} $\bowtie_{000}$ B (left associativity)	8
{A, B}	A $\bowtie_{111}$ B	B $\bowtie_{000}$ A	1
{B, C}		B $\bowtie_{111}$ C	3 -
		C $\bowtie_{000}$ B	5
{A, C}		A $\bowtie_{111}$ C	4 -
		C $\bowtie_{000}$ A	7

when add here, add one class as well

every transformation disabled

Random Approaches

Random Permutations

- construct permutations of relations (left deep trees)
- choose each relation as starting once
  - succesively add a relation to the existing chain
  - only explore the resulting chain if exchanging the last two relations does not result in a cheaper chain
  - recursion base: all relations are contained in the chain
    ⇒ Keep chain if cheaper than cheapest chain seen so far
- anytime algorithm (can be stopped after first generation)
- finds optimal plan eventually

Algorithm:

```
Unrank(n, R)
    π[i] = i
    for each n ≥ i > 0 descending {
        swap(π[i-1], π[R mod i])
        R = ⌊R/i⌋
    }
```

ex: rank = 64

- $i=8$ $R \bmod i = 0$ $R = 8$ swap (7, 0)
  $[\underline{8}\ 2\ 3\ 4\ 5\ 6\ 7\ \underline{1}]$
- $i=7$ $R \bmod i = 1$ $R = 1$ swap (6, 1)
  $[8\ \underline{7}\ 3\ 4\ 5\ 6\ \underline{2}\ 1]$
- $i=6$ $R \bmod i = 1$ $R = 0$ swap (5, 1)
  $[8\ \underline{6}\ 3\ 4\ 5\ \underline{7}\ 2\ 1]$

...

Random Join Trees (uniformly distributed, with cross products)

Steps of the algorithm:

1. Generate a random number b in $[0, C(n)[$
2. Unrank b to obtain a bushy tree with $n-1$ inner nodes
3. Generate a random number p in $[0, n![$
4. Unrank p to obtain a permutation
5. Attach the relations in order p from left to right as leaf nodes

Tree encoding: Inner node '(' Leaf node ')'

- Skip last node
- Replace '(' by 1 and ')' by 0

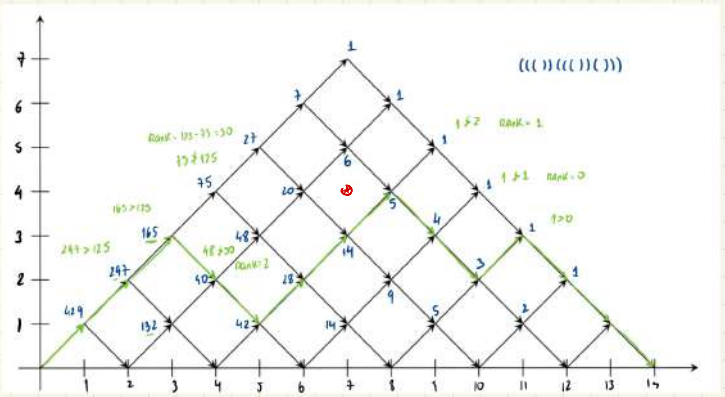
(())()
↓
110010

1, 2, 5 → pos of 1's

- order: left down

Dyck word example: ()(())

Unranking trees:

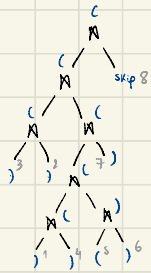


$$p(i,j) = \frac{j+1}{i+1}\binom{i+1}{\frac{1}{2}(i+j)+1}$$

(*) Note: 20 is the sum of 6 (up) and 14 (down)

→ Can construct tree iteratively from last row

- Dyck word: ((( ))((( ))( )))



Permutation: 3 2 1 4 5 6 7 8

(((3 2)((1 4)(5 6)) 7))8)

List Merge

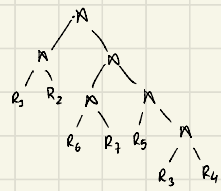
- Lists: Prolog - Notation < a | t >
- List L is a merge of 2 disjoint lists L_1 and L_2
- $\alpha = [\alpha_0, \dots, \alpha_n]$ represents a merge
  - Obtain merged list by taking α_0 elements from L_1
  - Then an element from L_2 follows.
  - Then α_1 elements from L_1 and next element from L_2. And so on

example: T_1: $(<(R_1 \bowtie R_2), R_3>, \underline{4}) = (1\ 2), 3, 4$

T_2: $(<(R_6 \bowtie R_7), R_5>, \underline{4}) = (6\ 7), 5, 4$

$\alpha = [1, 0, 1]$

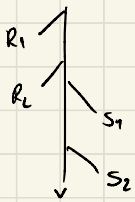
<u>Merged</u>: $(<(R_1 \bowtie R_2), (R_6 \bowtie R_7), R_5, R_3>, R_4)$
$\rightarrow (1\ 2), (6\ 7), 5, 3, 4$



Anchored List Representation: $R = (L_R, w)$ ↳ leaf node

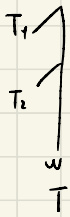
- Common leaf node $\underline{w}$ as to the the <u>same anchor node</u> when merging

- (T_1, T_2, α) : merge triplet



$(R, S, [2,0,0])$

Leaf-insertion:



T

(T, 1)

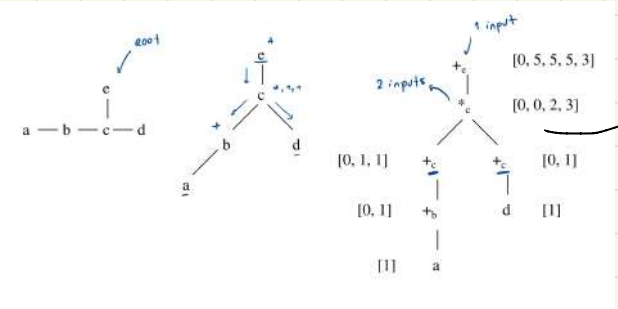
(T, 2)

Standard Decomposition Graph (SDG)

- (all) possible constructions of join trees
- not unique
- Two operations :

  $+v$ – leaf-insertion of v.

  $*w$ – tree merging of subtrees with w in common



2 inputs

↳ add $*c$

$+c$ $+c$

Quick – Pick

- generates pseudo-random trees
- <u>randomly pick</u> an edge from the query graph

  "choose a random $e \in E'$"
- <u>no longer uniformly distributed</u> ⇒ no guarantees BUT <u>very fast</u>
- <u>use union-find</u> datastructure to identify subsets containing nodes connected by an edge

- Normally used to get <u>cost bounds</u>. → its <u>very cheap</u>

Metaheuristics

- universal optimization strategy → considered a "brute force" method
- applicable for many different problems
- work well even for large problems

Iterative Improvement

- start with random join tree
- apply random transformation until minimum is reached
- stop when no further improvement possible
- Problem: can be stuck in local minimum

Simulated Annealing

best

- start with random join tree
- apply random transformation
- accept transformed tree either if it is cheap or – with a dependent temperature – even if it is more expensive
  - → probability of exploring trees with higher cost
    - ↳ when cost difference is low, probability is higher.
- decrease temperature over time
- allows to escape local minima
- Problems: parameter tunning; initial temperature; temperature decrease rate

Tabu Search

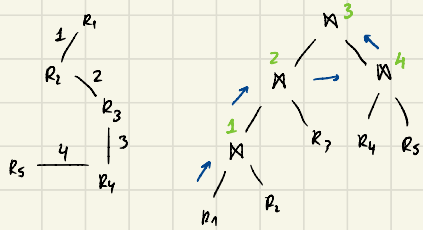
- start with random join trees
- investigate cheapest neighbor even if it is more expensive
- Keep (recently) investigated solutions in TabuSet to avoid running into circles

Genetic Algorithms

- population of join trees
- use different tree encoding schemes
- simulate crossover and mutation
- survival of the fittest

Ordered List Encoding:

1. left-deep trees: permutation of 1 ... n
   $(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) = 1234$
2. bushy trees: ordered list of edges in the join graph (bottom up, left-right)



1243

Ordinal Number Encoding:

- List $L = \langle R_1, \dots, R_n \rangle$

1. left-deep trees: index of first relation to be joined. Eliminate from L. Index in L, remove, keep going

example: $L = \langle R_1, R_2, R_3, R_4 \rangle$ and $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$

R_1: pos 1 in $\langle R_1, \dots, R_4 \rangle$, remove R_1
R_4: pos 3 in $\langle R_2, R_3, R_4 \rangle$, remove R_4
...

Result: 1311

2. bushy trees: bottom-up, left to right. Positions in L for both. Eliminate both R_i and R_j and create $R_{i,j}$. Push $R_{i,j}$ to the front of L, after already existing $R_{x,y}$.

example: $\langle R_1, R_2, R_3, R_4 \rangle$ and $((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$

$$\langle \overset{1}{R_1}, \overset{2}{R_2}, R_3, R_4 \rangle \rightarrow \langle R_{1,2}, \overset{2}{R_3}, \overset{3}{R_4} \rangle \rightarrow \langle \overset{1}{R_{1,2}}, \overset{2}{R_{3,4}} \rangle$$

Result: 12 23 12

Combinations and Hybrid Approaches

Two Phase Optimization:

- II to find local minima
- SA to find a better plan in the neighborhood

AB Algorithms:

- if query graph is cyclic → spanning tree
- Assign join methods randomly
- Apply IKKBZ
- Apply II

Toured simulated Annealing:

- Run SA n times with different initial join trees (e.g. results of GOO-3)

GOO-II:

- Run II on the result of GOO

Iterative Dynamic Programming: (IDP-I)

- build join trees up to K relations, replace cheapest with compound, repeat
- compromise between runtime and optimality
- combines greedy heuristics with dynamic programming
- scales well for large problems
- finds optimal solution for smaller problems

Order Preserving Joins

Algorithm:

- non-commutative operators!
- similar to matrix multiplication
- in addition: selection push down
- DP table is n x n array (or rather 4 arrays)
- algorithm fills arrays p, s, c, t:
  - p: applicable predicates
  - s: statistics (cardinality, perhaps more)
  - c: costs
  - t: split position for larger plans
- plan is extracted from the arrays afterwards

$$(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) \quad \underline{vs.} \quad ((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$$

example:

cardinality statistics ($\Pi_{S_i N_i}$)

	R_1	R_2	R_3	R_4
R_1	100	100	8	1
R_2		25	200	25
R_3			40	500
R_4				100

cost costs (C_{out})

	R_1	R_2	R_3	R_4
R_1	0	100	108	109
R_2		0	200	225
R_3			0	500
R_4				0

split points

	R_1	R_2	R_3	R_4
R_1		1	2	3
R_2			2	3
R_3				3
R_4				

$C_{out}((R_1 \bowtie R_2) \bowtie R_3) = 100 + 0.2 \times 0.01 \times 100 \times 40 = 108$

$C_{out}(R_1 \bowtie (R_1 \bowtie R_3)) = 200 + \ldots > 108$

$C_{out}(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) = 100 + 8 + 0.01 \times 0.125 \times 100 \times 8 = 109$

Resulting tree: $(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4)$ Cost $= \underline{109}$

Accessing the Data

- estimation and optimization of disk access costs

Track skew: optimizing data retrieval by intentionally misaligning sector positions on adjacent tracks in disk.

Cylinder skew: optimizing data access by misaligning track positions across multiple disk platters in storage systems

Bad blocks: sector that cannot be used – may cause hickups during sequential reads → jump over it

Seek Time:

$$\text{seektime}(d) = \begin{cases} c_1 + c_2\sqrt{d} & d \le c_0 \\ c_3 + c_4 d & d > 0 \end{cases}$$

where constants c_i are disk specific. Constant c_0 indicates the maximum number of cylinders where no coast takes place: seeking over a distance over c_0 cylinders ⇒ arm moves at maximum velocity.

Simplistic Cost Model:

The time a disk needs to read and transfer n bytes is aprox. by:

$$T = D_{lat} + \frac{n}{D_{SRR}}$$

Sequential I/O: one positioning at the beginning; data is read with the sustained read rate $\rightarrow T = D_{lat} + \frac{n}{D_{SRR}}$

Random I/O: positioning for every unit of transfer – typically a page of 8KB – is assumed.

$$\rightarrow T = \left(D_{lat} + \frac{|\text{page size}|}{D_{SRR}}\right) \cdot \#\text{pages}$$

Yao (Theorem) – Direct, uniform, distinct

Consider m buckets with n items each. Then there is a total of $N = nm$ items. If we randomly select K distinct items from all items then the number of qualifying buckets is:

$$\bar{Y}_n^{N,m}(K) = m \times Y_n^N(K)$$

where $Y_n^N(K)$ is the probability that a bucket contains at least one item.

$$Y_n^N(K) = \begin{cases} [1-p] & \text{if } K \leq N-n \\ 1 & \text{if } K > N-n \end{cases}$$

$\rightarrow$ should read everything

where p is the probability that a bucket contains none of the K items.

exclude n items of the current bucket

$$p = \binom{N-n}{K} \Big/ \binom{N}{K}$$

number of possibility of picking K items from all N items

$$= \prod_{i=0}^{K-1} \frac{N-n-i}{N-i} = \prod_{i=0}^{n-1} \frac{N-K-i}{N-i}$$

Approximations:

Waters: $p \approx \left(1 - \frac{K}{N}\right)^n$ (rest follows Yao's formula)

Bernstein:

$$\bar{Y}_n^{N,m}(K) = \begin{cases} K & \text{if } K < \frac{m}{2} \\ \frac{K+m}{3} & \text{if } \frac{m}{2} \leq K < 2m \\ m & \text{if } 2m \leq K \end{cases}$$

Cheung (Theorem) - Direct, Uniform, non-distinct

Consider $\underline{m}$ buckets with $\underline{n}$ items each. Then there is a total of $\underline{N = nm}$ items. If we randomly select $\underline{K}$ not necessarily distinct items from all items then the number of qualifying buckets is:

$$\overline{\text{Cheung}}_n^{N,m}(K) = m \times \text{Cheung}_n^N(K)$$

where $\quad \text{Cheung}_n^N(K) = [1 - \tilde{p}]$

account for possible duplicates

with $\quad \tilde{p} = \binom{N - n + K - 1}{K} \Big/ \binom{N + K - i}{K}$

$$= \prod_{i=0}^{n-1} \frac{N - 1 - i}{N - 1 + K - i}$$

- **Note**: The number of multiset with cardinality $\underline{K}$ containing only elements from a set S with $|S| = N$ is:

$$\binom{N + K - 1}{K}$$

Approximations: $\quad$ Cardenas : $\quad \tilde{p} \approx \left(1 - \frac{n}{N}\right)^K$

the larger the K, the more wrong it gets

Corollary (Number of distinct values in a multiset)

Let S be a K-multiset containing elements from N-set T. Then the <u>number</u> of <u>distinct items</u> contained in S is

$$D(N, K) = \frac{NK}{N + K - 1}$$

if the elements in T occur with the same probability in S.

Bitvectors

- estimate sequential disk access costs for a sequence of $\underline{K}$ pages sorted in the order they reside in disk

Theorem:

Bitvector of length B. $\underline{b}$ 1's uniformly distributed. The remaining $B-b$ bits are zero.

The probability distribution of the number j of zeros

1. between two consecutive 1's,
2. before the first 1,
3. after the last 1

is given by

$$B_b^B(j) = \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}}$$

Corollary:

Expected value for the number of zeros:

1. between two consecutive 1's,
2. before the first 1,
3. after the last 1

is

$$\bar{B}_b^B = \sum_{j=0}^{B-b} j\, B_b^B(j) = \frac{B-b}{b+1} \quad //$$

Corollary:

Number of bits from the first bit to the last 1,

$$\bar{B}_{tot}(B,b) = \frac{Bb+b}{b+1} = B - \frac{B-b}{b+1} \quad //$$

number of 0's after last 1

Corollary:

Number of bits from the first 1 and the last 1,

$$\bar{B}_{1\text{-span}}(B,b) = \frac{Bb-B+2b}{b+1} = B - 2\cdot\frac{B-b}{b+1} \quad //$$

Selectivity Estimation

- Using histograms – aggregated data

Given a histogram, we can approximate selectivities as follows

$A = c$
$$\frac{\sum_{b \in B : c \in b} H_A(b)}{\sum_{b \in B} H_A(b)}$$

$A > c$
$$\frac{\sum_{b \in B : c \in b} \frac{\max(b) - c}{\max(b) - \min(b)} H_A(b) + \sum_{b \in B : \min(b) > c} H_A(b)}{\sum_{b \in B} H_A(b)}$$

$A_1 = A_2$
$$\frac{\sum_{b_1 \in B_1, b_2 \in B_2, b' = b_1 \cap b_2 : b' \neq 0} \frac{\max(b') - \min(b')}{\max(b_1) - \min(b_1)} H_{A_1}(b_1) \frac{\max(b') - \min(b')}{\max(b_1) - \min(b_1)} H_{A_2}(b_2)}{\sum_{b_1 \in B_1} H_{A_1}(b_1) \sum_{b_2 \in B_2} H_{A_2}(b_2)}$$

Unnesting

- Eliminate dependent joins by transforming them into normal joins (with additional predicates)
- The attributes of the left input must be propagated up through the right input
- May need to join right input with the domain
- Transforms an $O(n^2)$ into (ideally) $O(n)$ operation
- Unnesting is essential for good performance
  - improve the asymptotics
  - can lead to arbitrary gains

Simple Unnesting:

$l_1 \bowtie (\sigma_{l_1.okey = l_2.okey}(l_2))$

↓ pulling the predicate up

$l_1 \bowtie_{l_1.okey = l_2.okey} l_2$

General Unnesting:

$T_1 \bowtie_p T_2 \equiv T_1 \bowtie_{p \wedge T_1 = A(D), D} (D \bowtie T_2)$

where $D := \Pi_{F(T_2) \cap A(T_1)}(T_1)$

Physical Properties

- Physical properties (ordering, grouping) of tuples can be exploited by operators to reduce the cost of a plan
  - example: sort-merge vs. hash join
    hash could be cheaper, but sort-merge still pay of later

Physical Property P: defines a partial relation $\leq_P$

If two plans p_1 and p_2 are logically equivalent:
- $p_1 \leq_P p_2$ if p_2 dominates p_1 concerning P
- $p_1 =_P p_2$ is p_1 and p_2 are comparable concerning P

A plan can be pruned if it is dominated or comparable

- Minimal cost subplan does not necessarily lead to minimal cost plan for entire query
- Need to consider physical properties
- Limit orderings to interesting orderings
  - ↳ requested by the user
  - ↳ physically available
  - ↳ used for a planed operator

Physical ordering: actual order of tuples on disk / in a tuple stream
Logical ordering: ordering satisfied by the tuples

Functional Dependencies (FD)

- $\sigma_{a=\cos(b)} \Rightarrow \{b \rightarrow a\}$
- $\sigma_{a=b} \Rightarrow \{a \rightarrow b,\ b \rightarrow a\}$
- $\sigma_{a=10} \Rightarrow \{\emptyset \rightarrow a\}$

- complex operators can induce multiple FDs

example:

```
select a, b, c
from s a,
    ( select b: b, c: count(*), d: max(d)
      from tablefunc(a) group by b )
order by a, b, c
```

Interesting orderings: (a), (b), (a,b) and (a,b,c)
Interesting groupings: {b}
Functional dependencies: $b \to c$, $b \to d$ → compute max after group by

↳ because after group by we compute the count c

Reducing orderings

Ordering (b, d, e), test for (a, b, c, e), FDs $\{a \to c, \emptyset \to a, b \to d\}$

1. reduce ordering to (b, e)
2. reduce test to (a, b, e) since $a \to c$
3. reduce test to (b, e) $\emptyset \to a$
4. test for prefix

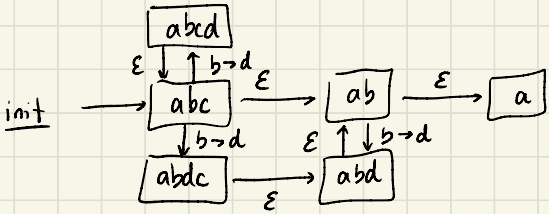
but:

- what would happen if we applied $\emptyset \to a$ first?
- Reductions must be applied back to front.

- memory management is a problem
- better than materializing orderings, but not optimal.

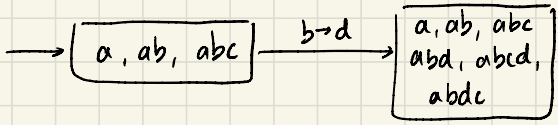
Encoding orderings as FSMs → finite-state machines

Use an FSM (ordering (a, b, c), FD $\{b \to d\}$)



- FSM describes physical orderings
- might be non-deterministic
- has to become deterministic
- conversion to DFSM (via NFA → DFA)

DFSM



- node contains all possible physical orderings
- operating on the DFSM is very efficient
- Problem: how to construct it (efficiently)

Ordering FSM Construction

- interesting orderings (requested, required, index)
- $O_I = O_P \cup O_T$ (produced vs. tested, allows pruning)
- functional dependencies (operators, keys)

E.g. $F = \{\{b \to c\}, \{b \to d\}\}$

$O_I = \{(b), (a,b)\} \cup \{(a,b,c)\}$

1. Initial nodes for O_I (b, ab, abc)
2. Edges for F. Creates artificial nodes (b,c) and (a)
3. Edges for initialization. (b,c) was pruned
4. Standard conversion algorithm

Pruning Techniques

- reducing the NFSM reduces conversion time
- reducing the DFSM reduces search space
- FDs can be removed if no interesting orderings reachable
- artificial nodes can be merged if they behave identicaly
- artificial nodes can be dropped if they only have ε edges

Note: search space reduction is a major benefit!