Building Query Compilers

Guido Moerkotte

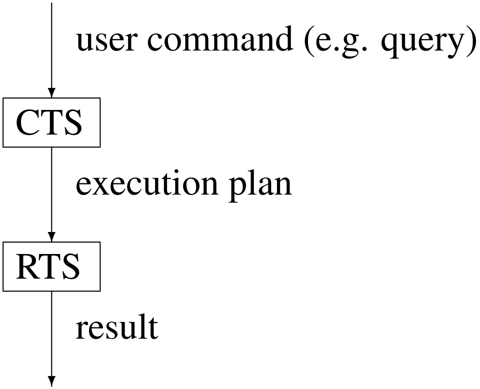
Skript siehe Internet.

1 Introduction

1.0 General Remarks

- Query languages like SQL or OQL are declarative.
- The task of the query compiler is to generate a *query evaluation plan*.
  (query execution plan, QEP)
- The QEP is an operator tree with physical algebraic operators as nodes.
- The QEP is interpreted by the runtime system.

1.1 DBMS Architecture

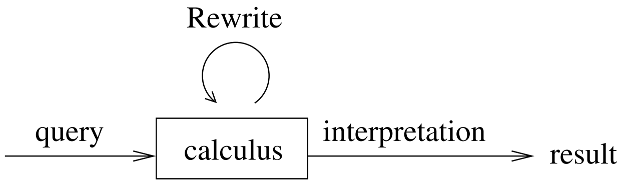


1.2 Interpretation versus Compilation

There are two approaches to process a query:

- *interpretation*
- *compilation*

1.2 Interpretation

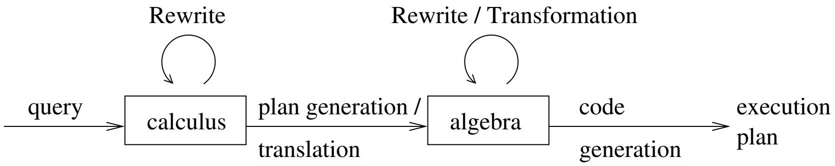


1.2 Interpretation

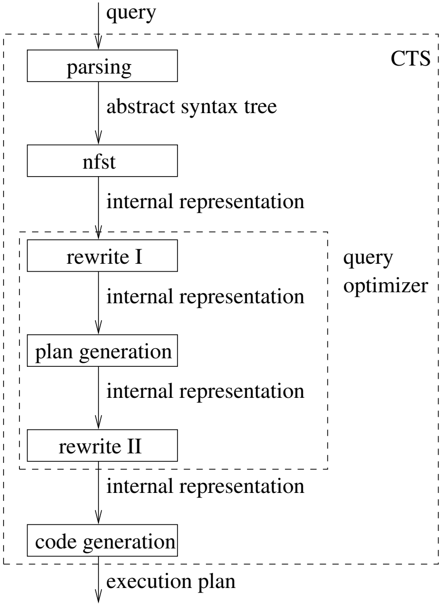
```
interprete(SQLBlock $x$) {
        /* possible rewrites go here */
        $s$ := $x$.select();
        $f$ := $x$.from();
        $w$ := $x$.where();
        $R$ := $\emptyset$; /* result */
        $t$ := []; /* empty tuple */
        eval($s$, $f$, $w$, $t$, $R$);
        return R;
}
```

```
eval($s$, $f$, $w$, $t$, $R$) {
        if($f$.empty())
            if($w$($t$))
                $R$ += $s$($t$);
        else
            foreach($t' \in$ first($f$))
                eval($s$, tail($f$), $w$, $t \circ t'$, $R$);
}
```

1.2 Compilation



1.2 Compiler Architecture



1.2 Compiler Tasks

1. Parser
2. NFST: normalization, factorization, semantic analysis, translation
3. Rewrite I: simple rewrites, deriving new predicates, view resolution and merging, unnesting nested queries
4. Plan Generation: find cheapest QEP (include e.g. join ordering)
5. Rewrite II: polishing
6. Code Generation

1.2 TPC-D Query 1

```
SELECT    RETURNFLAG, LINESTATUS,
          SUM(QUANTITY) as SUM_QTY,
          SUM(EXTENDEDPRICE) as SUM_EXTPR,
          SUM(EXTENDEDPRICE * (1 - DISCOUNT)),
          SUM(EXTENDEDPRICE * (1 - DISCOUNT)*
             (1 + TAX)),
          AVG(QUANTITY),
          AVG(EXTENDEDPRICE),
          AVG(DISCOUNT),
          COUNT(*)
FROM      LINEITEM
WHERE     SHIPDDATE <= DATE '1998-12-01'
GROUP BY RETURNFLAG, LINESTATUS
ORDER BY RETURNFLAG, LINESTATUS
```

1.2 Query Evaluation Plan

```
(group
  (tbscan
     {segment 'lineitem.C4Kseg' 0 4096}
     {nalslottedpage 4096}
     {ctuple 'lineitem.cschema'}
     [ 20
       LOAD_PTR       1
       LOAD_SC1_C     8  1  2 // L_RETURNFLAG
       LOAD_SC1_C     9  1  3 // L_LINESTATUS
       LOAD_DAT_C    10  1  4 // L_SHIPDATE
       LEQ_DAT_ZC_C 4 '1998-02-09'  1
     ] 2 1  // number of help-registers and selection
  ) 10 22 // hash table size,  number of registers
```

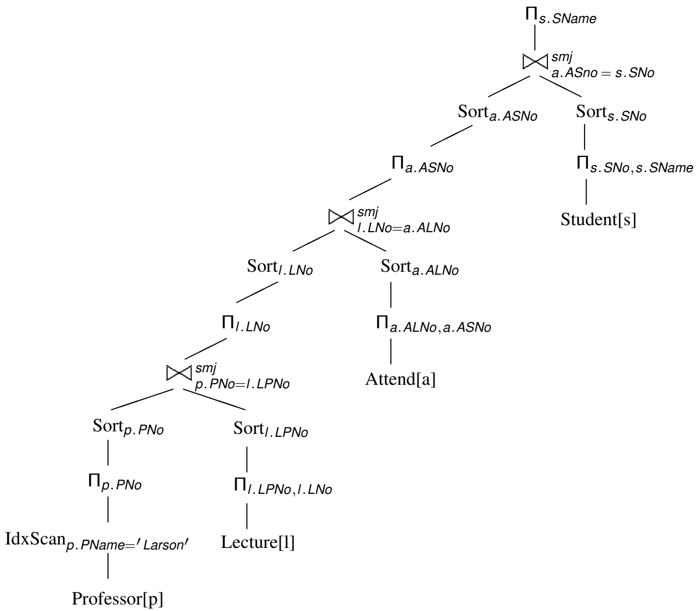
```
[ // init
  MV_UI4_C_C     1        0     // COUNT(*) = 0
  LOAD_SF8_C     4   1  6     // L_QUANTITY
  LOAD_SF8_C     5   1  7     // L_EXTENDEDPRICE
  LOAD_SF8_C     6   1  8     // L_DISCOUNT
  LOAD_SF8_C     7   1  9     // L_TAX
  MV_SF8_Z_C     6      10     // SUM/AVG(L_QUANTITY)
  MV_SF8_Z_C     7      11     // SUM/AVG(L_EXTENDEDPRICE)
  MV_SF8_Z_C     8      12     // AVG(L_DISCOUNT)
  SUB_SF8_CZ_C  1.0 8 13     // 1 - L_DISCOUNT
  ADD_SF8_CZ_C  1.0 9 14     // 1 + L_TAX
  MUL_SF8_ZZ_C  7  13 15     // SUM(L_EXTDPRICE * (1 - L_D
  MUL_SF8_ZZ_C 15  14 16     // SUM((...) * (1 + L_TAX))
]
```

```
[ // advance
  INC_UI4       0             // inc COUNT(*)
  MV_PTR_Y      1       1
  LOAD_SF8_C    4   1   6     // L_QUANTITY
  LOAD_SF8_C    5   1   7     // L_EXTENDEDPRICE
  LOAD_SF8_C    6   1   8     // L_DISCOUNT
  LOAD_SF8_C    7   1   9     // L_TAX
  MV_SF8_Z_A    6      10     // SUM/AVG(L_QUANTITY)
  MV_SF8_Z_A    7      11     // SUM/AVG(L_EXTENDEDPRICE)
  MV_SF8_Z_A    8      12     // AVG(L_DISCOUNT)
  SUB_SF8_CZ_C  1.0 8 13      // 1 - L_DISCOUNT
  ADD_SF8_CZ_C  1.0 9 14      // 1 + L_TAX
  MUL_SF8_ZZ_B  7  13 17 15 // SUM(L_EXTDPRICE * (1 - L_D
  MUL_SF8_ZZ_A 17  14 16      // SUM((...) * (1 + L_TAX))
]
[ // finalize
  UIFC_C          0    18
  DIV_SF8_ZZ_C  10 18 19     // AVG(L_QUANTITY)
  DIV_SF8_ZZ_C  11 18 20     // AVG(L_EXTENDEDPRICE)
  DIV_SF8_ZZ_C  12 18 21     // AVG(L_DISCOUNT)
] [ // hash program
  HASH_SC1  2 HASH_SC1 3
] [ // compare program
  CMPA_SC1_ZY_C 2  2  0
  EXIT_NEQ      0
  CMPA_SC1_ZY_C 3  3  0
])
```

1.2 Query's Result

RETURNFLAG	LINESTATUS	SUM_QTY	SUM_EXTPR	. . .
A	F	3773034	5319329289.68	. . .
N	F	100245	141459686.10	. . .
N	O	7464940	10518546073.98	. . .
R	F	3779140	5328886172.99	. . .

1.2 Abstracted Operator Tree



1.3 What to Optimize?

Cost function:

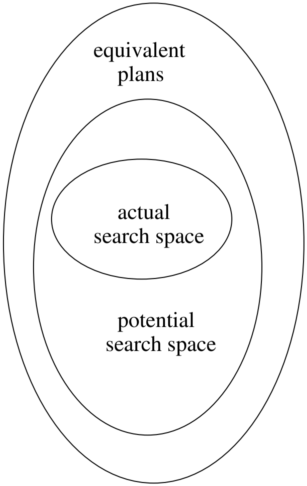
- Minimize resource consumption
- Maximize throughput
- Minimize response time
- Minimize time to first tuple

Which one? At least two of them.

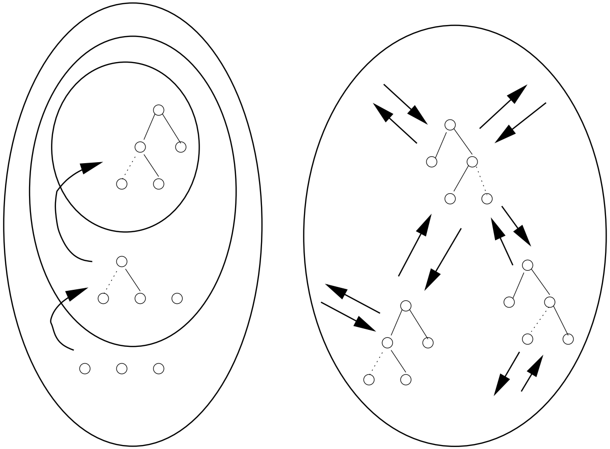
1.4 Requirements for a Query Compiler

1. Correctness
2. Completeness
3. Generate optimal plan (viz avoid the worst case)
4. Efficiency, generate the plan fast, do not waste memory
5. Graceful degradation
6. Robustness, maintainability, and all the other general requirements for software

1.5 Search Space



1.6 Transformation vs. Generation



a) Generative Approach b) Transformational Approach

2 Textbook Query Optimization

- repeat basics
- pinpoint problems
- reveal gaps

2.1 Example Query and Outline

Sample schema:

Student($\underline{\text{SNo}}$, SName, SAge, SYear)
Attend($\underline{\text{ASNo, ALNo, AGrade}}$)
Lecture($\underline{\text{LNo}}$, LTitle, LPNo)
Professor($\underline{\text{PNo}}$, PName)

Keys are underlined

2.1 Example Query and Outline

Sample query:

select **distinct** s.SName
from Student s, Attend a, Lecture l, Professor p
where s.SNo = a.ASNo **and** a.ALNo = l.LNo
and l.LPNo = p.PNo **and** p.PName = ‘Larson’

2.1 Example Query and Outline

- Review relational algebra
- Translation into the algebra
- Logical query optimization
- Physical query optimization

2.2 Relational Algebra

- Set operators: union ($\cup$), intersection ($\cap$), and setdifference ($\setminus$)
- Algebraic operators

2.2 Relational Algebra

Tuples and relations:

- Tuple: mapping from attribute names to values of a domain (Unordered!)
- Sample tuple: `[name: ``Anton'', age: 2]`
- Schema: set of attributes (with domain)
- Relation: set of tuples with the same schema
- Schema of a relation R: schema of the tuples in R, sometimes denoted by `sch(`R`)`, we use $\mathcal{A}(R)$

The argument relations of the set operators must have the same schema.

2.2 Relational Algebra

Tuple concatenation:

- Tuple concatentation: '$\circ$'
- Example:
  `[name: ``Anton'', age: 2]` $\circ$ `[toy: ``digger'']`
  results in
  `[name: ``Anton'', age: 2, toy: ``digger'']`

2.2 Relational Algebra

Tuple projection:

- Let $A' \subseteq A$ be sets of attributes and t a tuple with schema A
- Then $t.A'$ is the tuple that contains only the attributes in A'; others are discarded.
- Example:
  Let t be `[name: ``Anton'', age: 2, toy: ``digger'']` and
  $A = \{\texttt{name}, \texttt{age}\}$
  Then: $t.A$ = `[name: ``Anton'', age: 2]`

2.2 Relational Algebra

Free attributes (variables)

- Consider the predicate: `age` $= 2$
  where `age` is an attribute name
- `age` behaves like a free variable
- We must provide a value for `age` before the predicate can be evaluated
- Set of free attributes/variables of an expression e: $\mathcal{F}(e)$

2.2 Relational Algebra

Abbreviation of predicates:

- Let $A = \langle a_1, \ldots, a_k \rangle$ and $B = \langle b_1, \ldots, b_k \rangle$ be two attribute sequences.
- Then for any comparison operator $\theta \in \{=, \leq, <, \geq, >, \neq\}$, the expression $A \theta B$ abbreviates $a_1 \theta b_1 \wedge a_2 \theta b_2 \wedge \ldots \wedge a_k \theta b_k$.

2.2 Relational Algebra

Algebraic operators:

$$\begin{aligned}
\sigma_p(R) &:= \{r \mid r \in R, p(r)\} \\
\Pi_A(R) &:= \{r.A \mid r \in R\} \\
R_1 \times R_2 &:= \{r_1 \circ r_2 \mid r_1 \in R_1, r_2 \in R_2\} \\
R_1 \bowtie_p R_2 &:= \sigma_p(R_1 \times R_2)
\end{aligned}$$

2.2 Relational Algebra

Natural join:

- Consider two relations R_1 and R_2.
- Define $A_i := \mathcal{A}(R_i)$ for $i \in \{1, 2\}$, and $A := A_1 \cap A_2$.
- Assume that A is non-empty and $A = \langle a_1, \ldots, a_n \rangle$.

If A is non-empty, the natural join is defined as

$$R_1 \bowtie R_2 \quad := \quad \Pi_{A_1 \cup A_2}(R_1 \bowtie_p \rho_{A:A'}(R_2))$$

where $\rho_{A:A'}$ renames the attributes a_i in A to a'_i in A' and the predicate p has the form $A = A'$, i.e. $a_1 = a'_1 \wedge \ldots \wedge a_n = a'_n$.

2.2 Relational Algebra

Equivalences:

$$
\begin{aligned}
\sigma_{p_1 \wedge \ldots \wedge p_k}(R) &\equiv \sigma_{p_1}(\ldots(\sigma_{p_k}(R))\ldots) && (1)\\
\sigma_{p_1}(\sigma_{p_2}(R)) &\equiv \sigma_{p_2}(\sigma_{p_1}(R)) && (2)\\
\Pi_{A_1}(\Pi_{A_2}(\ldots(\Pi_{A_k}(R))\ldots)) &\equiv \Pi_{A_1}(R) && \\
& \quad \text{if } A_i \subseteq A_j \text{ for } i < j && (3)\\
\Pi_A(\sigma_p(R)) &\equiv \sigma_p(\Pi_A(R)) && \\
& \quad \text{if } \mathcal{F}(p) \subseteq A && (4)
\end{aligned}
$$

2.2 Relational Algebra

$$
\begin{aligned}
(R_1 \times R_2) \times R_3 &\equiv R_1 \times (R_2 \times R_3) && (5)\\
(R_1 \bowtie_{p_{1,2}} R_2) \bowtie_{p_{2,3}} R_3 &\equiv R_1 \bowtie_{p_{1,2}} (R_2 \bowtie_{p_{2,3}} R_3) && \\
&\quad \text{if } \mathcal{F}(p_{1,2}) \subseteq \mathcal{A}(R_1) \cup \mathcal{A}(R_2) && \\
&\quad \text{and } \mathcal{F}(p_{2,3}) \subseteq \mathcal{A}(R_2) \cup \mathcal{A}(R_3) && (6)\\
R_1 \times R_2 &\equiv R_2 \times R_1 && (7)\\
R_1 \bowtie_p R_2 &\equiv R_2 \bowtie_p R_1 && (8)
\end{aligned}
$$

2.2 Relational Algebra

$$\begin{aligned}
\sigma_p(R_1 \times R_2) &\equiv \sigma_p(R_1) \times R_2 \\
&\quad \text{if } \mathcal{F}(p) \subseteq \mathcal{A}(R_1) && (9) \\
\sigma_p(R_1 \bowtie_q R_2) &\equiv \sigma_p(R_1) \bowtie_q R_2 \\
&\quad \text{if } \mathcal{F}(p) \subseteq \mathcal{A}(R_1) && (10) \\
\Pi_A(R_1 \times R_2) &\equiv \Pi_{A_1}(R_1) \times \Pi_{A_2}(R_2) \\
&\quad \text{if } A = A_1 \cup A_2,\ A_i \subseteq \mathcal{A}(R_i) && (11) \\
\Pi_A(R_1 \bowtie_p R_2) &\equiv \Pi_{A_1}(R_1) \bowtie_p \Pi_{A_2}(R_2) \\
&\quad \text{if } \mathcal{F}(p) \subseteq A,\ A = A_1 \cup A_2, \\
&\quad \text{and } A_i \subseteq \mathcal{A}(R_i) && (12)
\end{aligned}$$

2.2 Relational Algebra

$$\begin{aligned}
\sigma_p(R_1 \theta R_2) &\equiv \sigma_p(R_1)\theta\sigma_p(R_2) \\
&\quad \text{where } \theta \text{ is any of } \cup, \cap, \setminus && (13) \\
\Pi_A(R_1 \cup R_2) &\equiv \Pi_A(R_1) \cup \Pi_A(R_2) && (14) \\
\sigma_p(R_1 \times R_2) &\equiv R_1 \bowtie_p R_2 && (15)
\end{aligned}$$

2.2 Relational Algebra

Remarks on validity/conditions:

- Relations joined must have disjoint attribute sets
  Attribute names must be unique.
  Means: notation $R.a$ for a relation R or $v.a$ for a variable ranging over tuples with an attribute a.
  Another possibility is to use the renaming operator ρ.
- Consumer/producer relationship must be o.k.
  Attributes must be provided before they are used

2.3 Canonical Translation

- Only: **select distinct**
  Reason: sets, not bags
- Not: **group by**, **order by**, **union**, **intersection**, **except**
- Only: attributes in **select** clause (no other expressions)
- Not: disjunction, negation
- Not discussed: NULL-values
- Not allowed: nested queries, views

2.3 Canonical Translation

Canonical translation:

1. Let $R_1 \ldots R_k$ be the entries in the **from** clause of the query. Construct the expression

$$F := \begin{cases} R_1 & \text{if } k = 1 \\ ((\ldots(R_1 \times R_2) \times \ldots) \times R_k) & \text{else} \end{cases}$$

2. The **where** clause is optional in SQL. Therefore, we distinguish the cases that there is no **where** clause and that the **where** clause exists and contains a predicate p. Construct the expression

$$W := \begin{cases} F & \text{if there is no } \textbf{where} \text{ clause} \\ \sigma_p(F) & \text{if the } \textbf{where} \text{ clause contains } p \end{cases}$$

2.3 Canonical Translation

3. Let s be the content of the **from** clause. For the canonical translation it must be of either '*' or a list $a_1, \ldots, a_n$ of attribute names. Construct the expression

$$S := \begin{cases} W & \text{if } s = \text{'*'} \\ \Pi_{a_1,\ldots,a_n}(W) & \text{if } s = a_1, \ldots, a_n \end{cases}$$

4. Return S.

2.3 Canonical Translation

Sample query:

select **distinct** s.SName
from Student s, Attend a, Lecture l, Professor p
where s.SNo = a.ASNo **and** a.ALNo = l.LNo
and l.LPNo = p.PNo **and** p.PName = ‘Larson’

Translation:

$$\Pi_{s.SName}(\sigma_p(((Student[s] \times Attend[a]) \times Lecture[l]) \times Professor[p]))$$

where p equals

s.SNo = a.ASNo **and** a.ALNo = l.LNo **and** l.LPNo = p.PNo **and** p.PName = ‘Larson’.

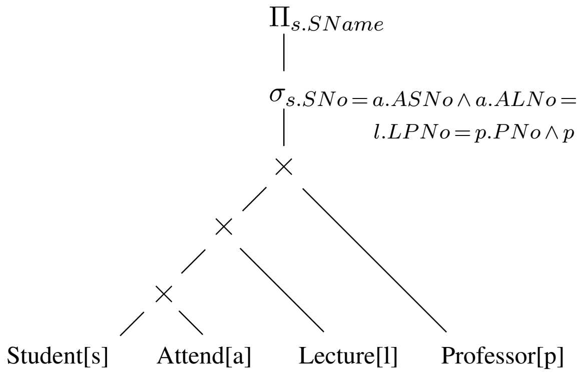
2.3 Canonical Translation

Remarks:

- We used the notation $R[r]$ to say that a relation named R has the correlation name r.
- Another interpretation:
  r is a variable successively bound to the elements (tuples) in R

2.3 Canonical Translation

Graphically:



2.4 Logical Query Optimization

Steps of textbook query optimization:

1. translate query into its canonical algebraic expression
2. perform logical query optimization
3. perform physical query optimization

2.4 Logical Query Optimization

- Foundation: algebraic equivalences
- Algebraic equivalences span the potential search space
- Given an initial algebraic expression: apply algebraic equivalences to derive new (equivalent) algebraic expressions
- Note: algebraic equivalences can be applied in two directions:
  from left to right and from right to left.
- Care has to be taken that the conditions attached to the equivalences are obeyed.

New equivalences increase the potential search space:

+ better plans
- larger search space

2.5 Logical Query Optimization

Better(?!):

- Plans can only be compared if there is a *cost function*
- Cost functions need details not available at the level of the logical algebra
- Consequence: logical query optimization remains a heuristic

2.5 Logical Query Optimization

Most algorithms for logical query optimization require

- organization of equivalences into groups
- directing equivalences

A *directed* equivalence is called *rewrite rule*. The groups of rewrite rules are then successively applied to the initial algebraic expression.
General idea behind: reduce intermediate result sizes.

2.5 Logical Query Optimization

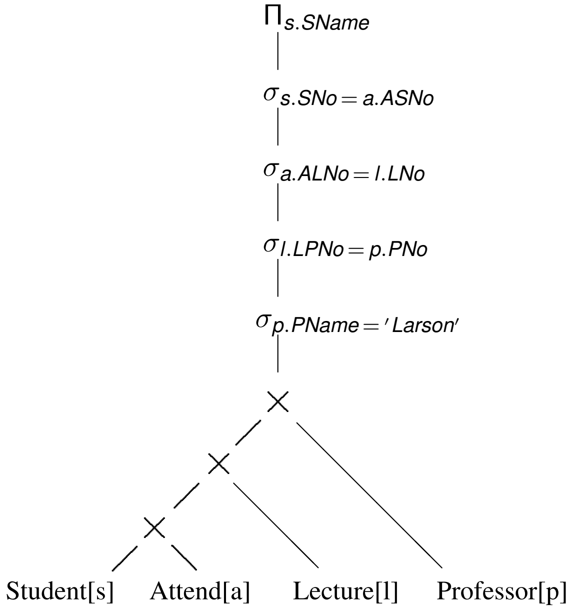
1. break up conjunctive selection predicates
   (Eqv. 1: $\rightarrow$)
2. push down selections
   (Eqv. 2: $\rightarrow$), (Eqv. 9: $\rightarrow$)
3. introduce joins
   (Eqv. 15: $\rightarrow$)
4. determine join order
   Eqv. 8, Eqv. 6, Eqv. 5, Eqv. 7
5. introduce and push down projections
   (Eqv. 3: $\leftarrow$), (Eqv. 4: $\rightarrow$),
   (Eqv. 11: $\rightarrow$), (Eqv. 12: $\rightarrow$)

2.5 Logical Query Optimization

Step 1:

- Break up conjunctive selection predicates.
- Motivation: selections with simple predicates can be moved around easier

For our example query Step 1 results in



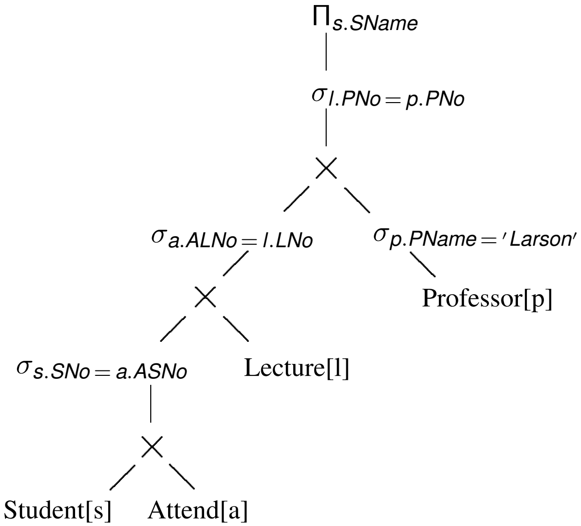
$\Pi_{s.SName}$
$\sigma_{s.SNo = a.ASNo}$
$\sigma_{a.ALNo = l.LNo}$
$\sigma_{l.LPNo = p.PNo}$
$\sigma_{p.PName = 'Larson'}$
$\times$
$\times$
$\times$
Student[s]
Attend[a]
Lecture[l]
Professor[p]

2.5 Logical Query Optimization

Step 2:

- Push selections down
- Motivation: reduce number of tuples early, especially inputs to expensive operators like join

For our example query Step 2 results in

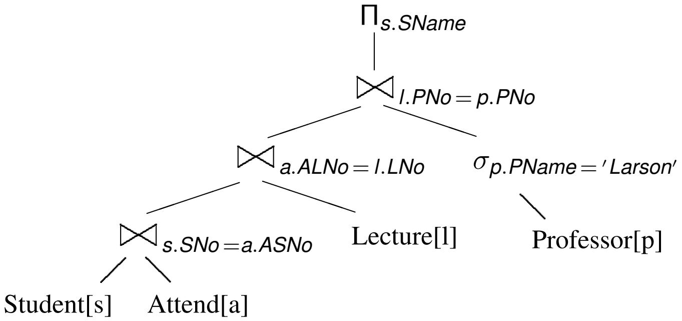

$\Pi_{s.SName}$
$\sigma_{l.PNo=p.PNo}$
$\times$
$\sigma_{a.ALNo=l.LNo}$
$\sigma_{p.PName='Larson'}$
$\times$
Professor[p]
$\sigma_{s.SNo=a.ASNo}$
Lecture[l]
$\times$
Student[s]
Attend[a]

2.5 Logical Query Optimization

Step 3:

- Convert cross products to joins
- Motivation: joins are cheaper than cross products

For our example query Step 3 results in


$\Pi_{s.SName}$
$\bowtie_{l.PNo = p.PNo}$
$\bowtie_{a.ALNo = l.LNo}$
$\sigma_{p.PName = 'Larson'}$
$\bowtie_{s.SNo = a.ASNo}$
Lecture[l]
Professor[p]
Student[s]
Attend[a]

2.5 5ogical Query Optimization

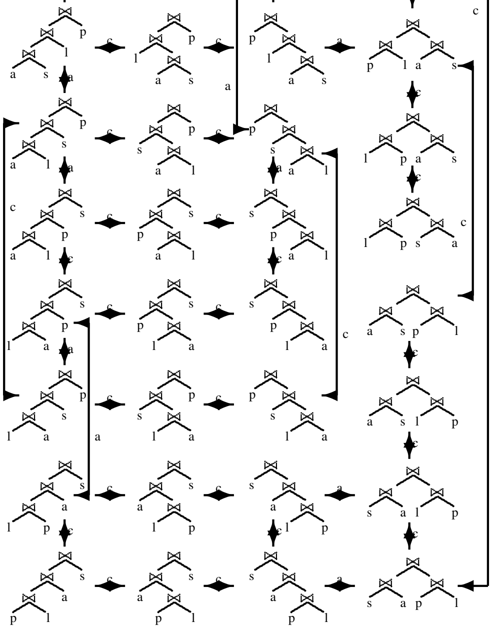
Step 4:

- Reorder joins (exploit associativity and commutativity)
- Motivation: find cheap join order

Remarks:

- Cost differ vastly for different orders
- Difficult problem (many alternatives, NP-hard)
- next chapter discusses only join ordering

Next slide: different join plans



2.5 Logical Query Optimization

Step 4 (Join Ordering) for example query:

- Currently, the bottom-most expression is

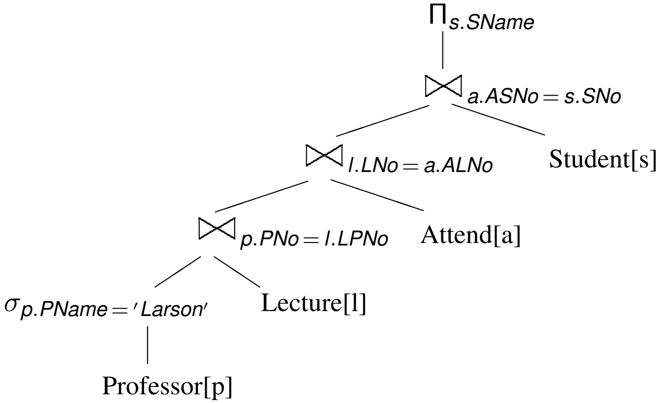
$$Student[s] \bowtie_{s.SNo=a.ASNo} Attend[a].$$

- Result is huge: all students, all lectures
- On the other hand: only one professor selected by

$$\sigma_{p.PName=\text{`}Larson\text{'}}(Professor[p])$$

- Join this with Lectures: only lectures by ‘Larson’: few compared to all lectures

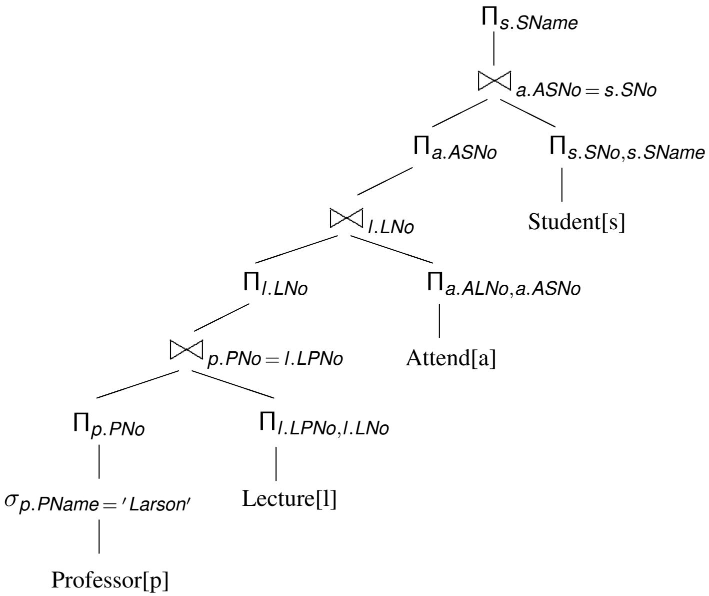
Result of join ordering:


$\Pi_{s.SName}$
$\bowtie_{a.ASNo = s.SNo}$
$\bowtie_{l.LNo = a.ALNo}$
Student[s]
$\bowtie_{p.PNo = l.LPNo}$
Attend[a]
$\sigma_{p.PName = 'Larson'}$
Lecture[l]
Professor[p]

2.5 Logical Query Optimization

Step 5:

- Push down projections
- Motivation: eliminate irrelevant attributes
- Do the elimination just before pipeline breakers


$\Pi_{s.SName}$
$\bowtie_{a.ASNo = s.SNo}$
$\Pi_{a.ASNo}$
$\Pi_{s.SNo, s.SName}$
$\bowtie_{l.LNo}$
Student[s]
$\Pi_{l.LNo}$
$\Pi_{a.ALNo, a.ASNo}$
$\bowtie_{p.PNo = l.LPNo}$
Attend[a]
$\Pi_{p.PNo}$
$\Pi_{l.LPNo, l.LNo}$
$\sigma_{p.PName = 'Larson'}$
Lecture[l]
Professor[p]

2.5 Logical Query Optimization: Excursion

We might encounter problems when pushing down selections:

select **distinct** s.SName
from Student s, Lecture l, Attend a
where s.SNo = a.ASNo **and** a.ALNo = l.LNo
and l.LTitle = 'Databases I'

After translation and Steps 1 and 2 the algebraic expression looks like

$$\Pi_{\mathrm{s}.SName}(\sigma_{\mathrm{s}.SNo=a.ASNo}(\sigma_{\mathrm{a}.ALNo=l.LNo}((Student[s] \times \sigma_{\mathrm{l}.LTitle=\text{`}Databases\ I'}(Lecture[l])) \times Attend[a]))).$$

Neither of $\sigma_{\mathrm{s}.SNo=a.ASNo}$ and $\sigma_{\mathrm{a}.ALNo=l.LNo}$ can be pushed down further.

2.5 Logical Query Optimization: Excursion

Only after reordering the cross products such as in

$$
\begin{array}{l}
\Pi_{\mathrm{s}.SName}(\\
\quad \sigma_{\mathrm{s}.SNo=a.ASNo}(\\
\quad\quad \sigma_{\mathrm{a}.ALNo=l.LNo}(\\
\quad\quad\quad (Student[s] \times Attend[a]) \\
\quad\quad\quad\quad \times \sigma_{\mathrm{l}.LTitle=`Databases\ I'}(Lecture[l]))))
\end{array}
$$

can $\sigma_{\mathrm{s}.SNo=a.ASNo}$ be pushed down:

$$
\begin{array}{l}
\Pi_{\mathrm{s}.SName}(\\
\quad\quad \sigma_{\mathrm{a}.ALNo=l.LNo}(\\
\quad\quad\quad \sigma_{\mathrm{s}.SNo=a.ASNo}(Student[s] \times Attend[a]) \\
\quad\quad \times \sigma_{\mathrm{l}.LTitle=`Databases\ I'}(Lecture[l])))
\end{array}
$$

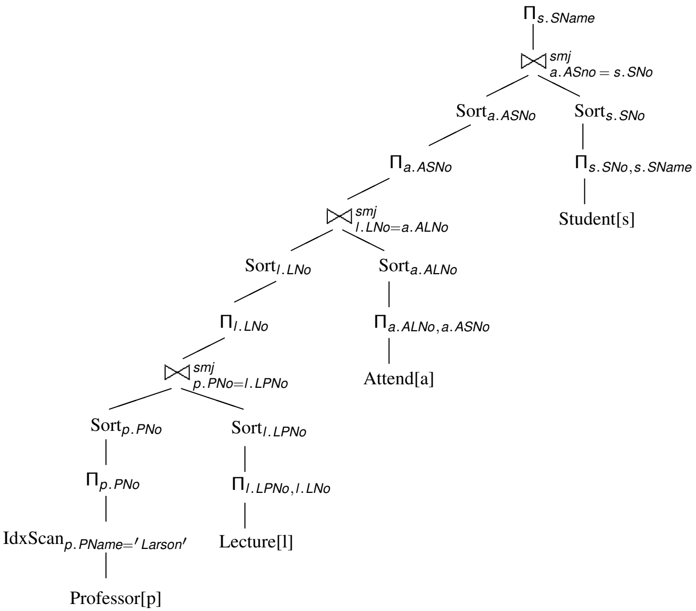
Lesson learned: Steps 2 and 4 are highly interdependent.

2.6 Physical Query Optimization

- Add more execution information to plan
- Select index structures/access paths
- Specify operator implementations
- Property enforcers
  (e.g. sort-merge join requires both inputs to be sorted)
  (e.g. in distributed databases the argument relations of a join must be at the side where the join is executed)
- DAG's
  (tmp-Operator)

QEP for example query

$$
\begin{array}{l}
\Pi_{\text{s}.SName}(\\
\quad \text{Sort}_{\text{a}.ASNo}(\Pi_{\text{a}.ASNo}(\\
\quad\quad \text{Sort}_{\text{l}.LNo}(\Pi_{\text{l}.LNO}(\\
\quad\quad\quad \text{Sort}_{\text{p}.PNo}(\Pi_{\text{p}.PNo}(\text{IdxScan}_{\text{p}.PName=\text{`}Larson\text{'}}(Professor[p])) \\
\quad\quad\quad \bowtie^{smj}_{\text{p}.PNo=l.LPNo} \\
\quad\quad\quad \text{Sort}_{\text{l}.LPNo}(\Pi_{\text{l}.LPno,l.LNo}(Lecture[l]))) \\
\quad\quad \bowtie^{smj}_{\text{l}.LNo=a.ALNo} \\
\quad\quad \text{Sort}_{\text{a}.ALNo}(\Pi_{\text{a}.ALNo,a.ASNo}(Attend[a])))) \\
\quad \bowtie^{smj}_{\text{a}.ASno=s.SNo} \\
\quad \text{Sort}_{\text{s}.SNo}(\Pi_{\text{s}.SNo,s.SName}(Student[s])))
\end{array}
$$


$\Pi_{s.SName}$
$\bowtie^{smj}_{a.ASno=s.SNo}$
$\text{Sort}_{a.ASNo}$
$\text{Sort}_{s.SNo}$
$\Pi_{a.ASNo}$
$\Pi_{s.SNo,s.SName}$
$\bowtie^{smj}_{l.LNo=a.ALNo}$
Student[s]
$\text{Sort}_{l.LNo}$
$\text{Sort}_{a.ALNo}$
$\Pi_{l.LNo}$
$\Pi_{a.ALNo,a.ASNo}$
$\bowtie^{smj}_{p.PNo=l.LPNo}$
Attend[a]
$\text{Sort}_{p.PNo}$
$\text{Sort}_{l.LPNo}$
$\Pi_{p.PNo}$
$\Pi_{l.LPNo,l.LNo}$
$\text{IdxScan}_{p.PName='Larson'}$
Lecture[l]
Professor[p]

2.7 Discussion

- Only a small fraction of SQL discussed
  (see omissions at beginning of this chapter)
- Separation into two phases (looses optimality)
- Join ordering
- Query rewrite (views, nested queries)

Outline:

- Join ordering (algorithms)
- Physical query optimization (details, costs)
- Rewrite (views, nested queries)

Join Ordering

Queries Considered

- conjunctive queries, i.e.
- conjunctions of simple predicates
- predicates of the form $e_1 \theta e_2$
  e_1 is an attribute, e_2 is either an attribute or a constant
- join predicates: θ must be '$=$'
  (equi-joins only)

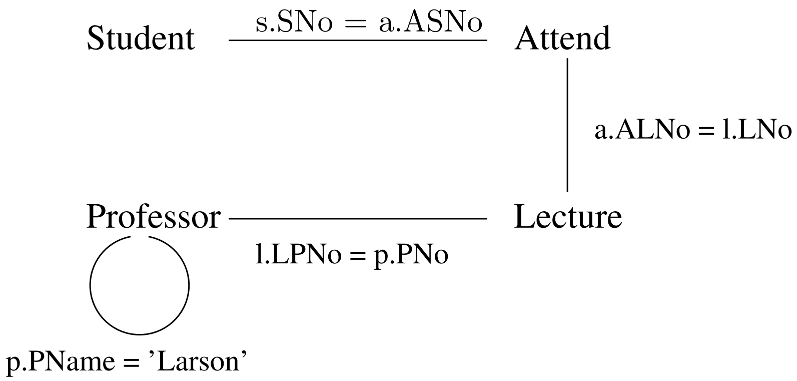
We join relations $R_1, \ldots, R_n$ where R_i can be

- a base relation
- a base relation to which a selection has been applied
- a more complex building block or access path

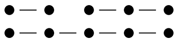
Query Graph

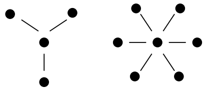
A query graph is an undirected graph with nodes $R_1, \ldots, R_n$.
For every join predicate in the conjunction P whose attributes belong to the relations R_i and R_j, we add an edge between R_i and R_j.
This edge is labeled by the join predicate.
For simple predicates applicable to a single relation, we add a self-edge.
However, our algorithms will not consider simple selection predicates. They have to be pushed down before.

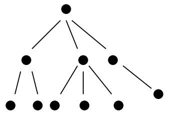
Example Query Graph

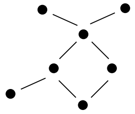


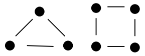
Shapes of Query Graphs

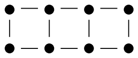

chain queries

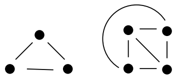

star queries


tree query


cyclic query


cycle queries


grid query


clique queries

Join Trees

are binary trees

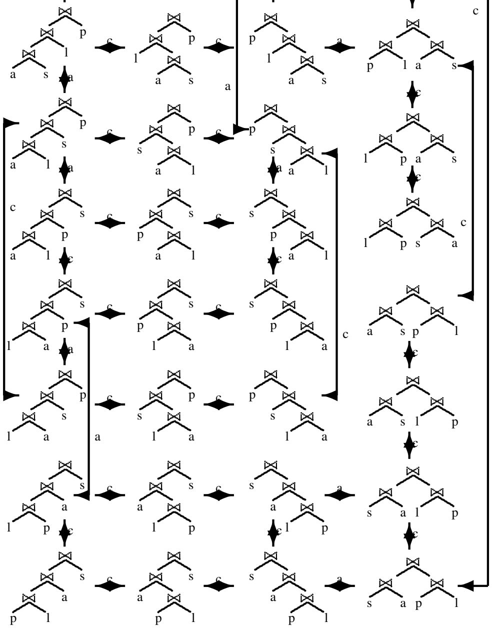
- with relation names attached to leaf nodes and
- join operators as inner nodes.

Some algorithms will produce ordered binary trees, others unordered binary trees.
Distinguish whether cross products are allowed or not.

Join Tree Shapes

- left-deep join trees
- right-deep join trees
- zig-zag trees
- bushy trees

Left-deep, right-deep, and zig-zag trees can be summarized under the notion of *linear trees*



a
c

Simple Cost Functions

Input:

- cardinalities: $|R_i|$
- selectivities: $f_{i,j}$ of $p_{i,j}$ is then defined as

$$f_{i,j} = \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i| * |R_j|}$$

Calculate:

- result cardinality:

$$|R_i \bowtie_{p_{i,j}} R_j| = f_{i,j}|R_i||R_j|$$

Result Cardinalities of Join Trees

Consider a join tree $T = T_1 \bowtie T_2$.
Then, $|T|$ can be calculated as follows:

- If T is a leaf R_i, then $|T| = |R_i|$.
- Otherwise,

$$|T| = (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j})\ |T_1|\ |T_2|.$$

This formula assumes independence.

Example

The table

$$\begin{aligned} |R_1| &= 10 \\ |R_2| &= 100 \\ |R_3| &= 1000 \\ f_{1,2} &= 0.1 \\ f_{2,3} &= 0.2 \end{aligned}$$

implicitly defines the query graph $R_1 - -R_2 - -R_3$.
(We assume $f_{i,j} = 1$ for all i, j for which $f_{i,j}$ is not explicitly given.)

The cost function C_{out}

$$C_{\text{out}}(T) = \begin{cases} 0 & \text{if } T \text{ is a single relation} \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

Other cost functions

For single join operators:

$$\begin{aligned}
C_{\text{nlj}}(e_1 \bowtie_p e_2) &= |e_1||e_2| \\
C_{\text{hj}}(e_1 \bowtie_p e_2) &= h|e_1| \\
C_{\text{smj}}(e_1 \bowtie_p e_2) &= |e_1|log(|e_1|) + |e_2|log(|e_2|)
\end{aligned}$$

For sequences of join operators (relations):

$$\begin{aligned}
C_{\text{hj}}(s) &= \sum_{i=2}^{n} 1.2|s_1, \ldots, s_{i-1}| \\
C_{\text{smj}}(s) &= \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| \log(|s_1, \ldots, s_{i-1}|) + \sum_{i=2}^{n} |s_i| \log(|s_i|) \\
C_{\text{nlj}}(s) &= \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| * s_i
\end{aligned}$$

Remarks on Cost Functions

- cost functions are simplistic
- cost functions designed for left-deep trees
- C_{hj} and C_{smj} do not work for cross products
  (Fix: define them to be equal to the output cardinality which happens to be the costs of the nested-loop cost function)
- in reality: other parameters besides cardinality play a role
- the above cost functions assume that the same join algorithm is chosen throughout the whole plan

Example Calculations

	C_{out}	C_{nlj}	C_{hj}	C_{smj}
$R_1 \bowtie R_2$	100	1000	12	697.61
$R_2 \bowtie R_3$	20000	100000	120	10630.26
$R_1 \times R_3$	10000	10000	10000	10000.00
$(R_1 \bowtie R_2) \bowtie R_3$	20100	101000	132	11327.86
$(R_2 \bowtie R_3) \bowtie R_1$	40000	300000	24120	32595.00
$(R_1 \times R_3) \bowtie R_2$	30000	1010000	22000	143542.00

Observations

- Costs differ vastly
- different cost functions result in different costs
- the cheapest join tree is the cheapest one under all cost functions
- join trees with cross products are expensive
- the order in which relations are joined is essential under all (and other) cost functions

Another Example

Query: $|R_1| = 1000$, $|R_2| = 2$, $|R_3| = 2$, $f_{1,2} = 0.1$, $f_{1,3} = 0.1$
For C_{out} we have costs

Join Tree	C_{out}
$R_1 \bowtie R_2$	200
$R_2 \times R_3$	4
$R_1 \bowtie R_3$	200
$(R_1 \bowtie R_2) \bowtie R_3$	240
$(R_2 \times R_3) \bowtie R_1$	44
$(R_1 \bowtie R_3) \bowtie R_2$	240

Plan with cross product is best.

Yet Another Example

Query: $|R_1| = 10$, $|R_2| = 20$, $|R_3| = 20$, $|R_4| = 10$, $f_{1,2} = 0.01$, $f_{2,3} = 0.5$, $f_{3,4} = 0.01$

Join Tree	C_{out}
$R_1 \bowtie R_2$	2
$R_2 \bowtie R_3$	200
$R_3 \bowtie R_4$	2
$((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$	24
$((R_2 \bowtie R_3) \bowtie R_1) \bowtie R_4$	222
$(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$	6

Bushy tree better than any left-deep tree

Properties of Cost Functions: Symmetry and ASI

A cost function C_{impl} is called *symmetric*, if $C_{\text{impl}}(R_1 \bowtie^{\text{impl}} R_2) = C_{\text{impl}}(R_2 \bowtie^{\text{impl}} R_1)$ for all relations R_1 and R_2.
For symmetric cost functions, it does not make sense to consider commutativity.
ASI: adjacent sequence interchange (see below)

	ASI	$\neg$ ASI
symmetric	C_{out}	C_{smj}
$\neg$ symmetric	C_{hj}	(listen)

Classification of Join Ordering Problems

Query Graph Classes × *Possible Join Tree Classes* × *Cost Function Classes*

- Query Graph Classes: *chain*, *star*, *tree*, and *cyclic*
- Join trees: left-deep, zig-zag, or bushy trees: w/o cross products
- Cost functions: w/o ASI property

In total, we have $4 * 3 * 2 * 2 = 48$ different join ordering problems.

Search Space Size

1. with cross products
2. without cross products

Number of Linear Trees with Cross Products

- left-deep: $n!$
- right-deep: $n!$
- zig-zag: $2^{n-2}n!$

Remember: Catalan Numbers

For n leave nodes, the number of binary trees is given by $\mathcal{C}(n-1)$ where $\mathcal{C}(n)$ is defined by the recurrence

$$\mathcal{C}(n) = \sum_{k=0}^{n-1} \mathcal{C}(k)\mathcal{C}(n-k-1)$$

with $\mathcal{C}(0) = 1$. They can also be computed by the following formula:

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

The Catalan Numbers grow in the order of $\Theta(4^n/n^{3/2})$.

Number of Bushy Trees with Cross Products

$$\begin{aligned}
n!\,\mathcal{C}(n-1) &= n!\,\frac{1}{n}\binom{2(n-1)}{n-1} \\
&= n!\,\frac{1}{n}\,\frac{(2n-2)!}{(n-1)!\,((2n-2)-(n-1))!} \\
&= \frac{(2n-2)!}{(n-1)!}
\end{aligned}$$

Chain Queries, Left-Deep Join Trees, No Cross Product

Let us denote the number of join trees for a chain query in n relations with query graph $R_1 - R_2 - \ldots - R_{n-1} - R_n$ as $f(n)$.
Obvious: $f(0) = 1$ and $f(1) = 1$.

. . . for $n > 1$

For larger n:
Consider the join trees for $R_1 - \ldots - R_{n-1}$ where

- R_{n-1} is the k-th relation from the bottom
  where k ranges from 1 to $n - 1$.

From such a join tree we can derive join trees for all n relations
by adding relation R_n at any position $\underline{\text{following}}$ R_{n-1}.
There are $n - k$ such join trees.
Only for $k = 1$, we can also add R_n below R_{n-1}. Hence, for
$k = 1$ we have n join trees.
How many join trees with R_{n-1} at position k are there?

For $k = 1$, R_{n-1} must be the first relation to be joined.
Since we do not consider cross products, it must be joined with R_{n-2}.
The next relation must be R_{n-3}, and so on.
Hence, there is only one such join tree.
For $k = 2$, the first relation must be R_{n-2} which is then joined with R_{n-1}.
Then $R_{n-3}, \ldots, R_1$ must follow in this order.
Again, there is only one such join tree.
For higher k, for R_{n-1} to occur savely at position k (no cross products) the $k - 1$ relations $R_{n-2}, \ldots, R_{n-k}$ must occur before R_{n-1}.
There are exactly f($k - 1$) join trees for the $k - 1$ relations.
On each such join tree we just have to add R_{n-1} on top of it to yield a join tree with R_{n-1} at position k.

Recurrence

Now we can compute the $f(n)$ as

$$f(n) = n + \sum_{k=2}^{n-1} f(k-1) * (n-k)$$

for $n > 1$.
Solving the recurrence gives us

$$f(n) = 2^{n-1}$$

(Exercise)

Chain Queries, Bushy Join Trees, No Cross Product

Let $f(n)$ be the number of bushy trees without cross products for a chain query in n relations with query graph $R_1 - R_2 - \ldots - R_{n-1} - R_n$.
Obvious: $f(0) = 1$ and $f(1) = 1$.

. . . for $n > 1$

Every subtree of the join tree must contain a subchain in order to avoid cross products
Every subchain can be joined in either the left or the right argument of the join.
Thus:

$$f(n) = \sum_{k=1}^{n-1} 2f(k)f(n-k)$$

This is equal to

$$2^{n-1} * \mathcal{C}(n-1)$$

(Exercise)

Star Queries, No Cartesian Product

Star: R_0 in the center, $R_1, \dots, R_{n-1}$ as satellites
The first join must involve R_0.
The order of the remaining relations does not matter.

- left-deep trees: $2 * (n-1)!$
- right-deep trees: $2 * (n-1)!$
- zig-zag trees: $2 * (n-1)! * 2^{n-2} = 2^{n-1} * (n-1)!$

Remark

The numbers for star queries are also upper bounds for tree queries.
For clique queries, there is no join tree possible that does contain a cross product.
Hence, all join trees are valid join trees and the search space size is the same as the corresponding search space for join trees with cross products.

Numbers

	Join Trees Without Cross Products				
	Chain Query			Star Query	
	Left-Deep	Zig-Zag	Bushy	Left-Deep	Zig-Zag/Bushy
n	2^{n-1}	2^{2n-3}	$2^{n-1}\mathcal{C}(n-1)$	$2*(n-1)!$	$2^{n-1}(n-1)!$
1	1	1	1	1	1
2	2	2	2	2	2
3	4	8	8	4	8
4	8	32	40	12	48
5	16	128	224	48	384
6	32	512	1344	240	3840
7	64	2048	8448	1440	46080
8	128	8192	54912	10080	645120
9	256	32768	366080	80640	10321920
10	512	131072	2489344	725760	185794560

Numbers

	With Cross Products/Clique		
	Left-Deep	Zig-Zag	Bushy
n	$n!$	$2^{n-2} * n!$	$n!\mathcal{C}(n-1)$
1	1	1	1
2	2	2	2
3	6	12	12
4	24	96	120
5	120	960	1680
6	720	11520	30240
7	5040	161280	665280
8	40320	2580480	17297280
9	362880	46448640	518918400
10	3628800	928972800	17643225600

Complexity

Query Graph	Join Tree	Cross Prod.	Cost Function	Complexity
general	left-deep	no	ASI	NP-hard
tree/star/chain	left-deep	no	one join method (ASI)	P
star	left-deep	no	two join methods (NLJ+SMJ)	NP-hard
general/tree/star	left-deep	yes	ASI	NP-hard
chain	left-deep	yes	—	open
general	bushy	no	ASI	NP-hard
tree	bushy	no	—	open
star	bushy	no	ASI	P
chain	bushy	no	any	P
general	bushy	yes	ASI	NP-hard
tree/star/chain	bushy	yes	ASI	NP-hard

Join Ordering Algorithms

1. Deterministic Algorithms
   1.1 Heuristics
   1.2 IKKBZ
   1.3 MVP
   1.4 Dynamic Programming (DP)
   1.5 Memoization
   1.6 Join Ordering by Generating Permutations
   1.7 . . .
2. Probabilistic Algorithms
3. Hybrid Algorithms
4. Ordering Order-Preserving Joins

Heuristics: Greedy Algorithms

First Algorithm:

- Left-deep trees, i.e. a permutation
- Relations are ordered according to some weight function
- W/O cross products

Heuristics: Greedy Algorithms

```
GreedyJoinOrdering-1($\{R_1,\ldots,R_n\}$, (*weight)(Relation))
Input: a set of relations to be joined and a weight function
Output: a join order
$S$ = $\epsilon$; // initialize $S$ to the empty sequence
$R$ = $\{R_1,\ldots,R_n\}$; // let $R$ be the set of all relations
while(!empty($R$)) {
  Let $k$ be such that: weight($R_k$) = min$_{R_i \in R}$(weight($R_i$));
  $R\backslash = R_k$; // eliminate $R_k$ from $R$
  $S\circ = R_k$; // append $R_k$ to S
}
return $S$
```

Examples for weight: cardinality of a relation
Remark: speed up by sorting

Heuristics: Greedy Algorithms

Second Algorithm:

- not all heuristics fit the above scheme
- Example: smallest intermediate result size next
- The set of relations before the current one influences the weight of a relation

The second algorithm fixes this.

Heuristics: Greedy Algorithms

```
GreedyJoinOrdering-2($\{R_1, \ldots, R_n\}$,
                    (*weight)(Sequence of Relations, Relation))
**Input:** a set of relations to be joined and a weight function
**Output:** a join order
$S$ = $\epsilon$; // initialize $S$ to the empty sequence
$R$ = $\{R_1, \ldots, R_n\}$; // let $R$ be the set of all relations
**while**(!empty($R$)) {
  Let $k$ be such that: weight($S, R_k$) = min$_{R_i \in R}$(weight($S$, $R_i$));
  $R \setminus = R_k$; // eliminate $R_k$ from $R$
  $S \circ = R_k$; // append $R_k$ to S
}
**return** $S$
```

Heuristics: Greedy Algorithms

```
GreedyJoinOrdering-3($\{R_1,\ldots,R_n\}$,
                (*weight)(Sequence of Relations, Relation))
Solutions = $\emptyset$;
for ($i = 1$; $i \leq n$; $++i$) {
  $S$ = $R_i$; // initialize $S$ to a singleton sequence
  $R = \{R_1,\ldots,R_n\} \setminus \{R_i\}$; // let $R$ be the set of all relations
  while(!empty($R$)) {
    Let $k$ be such that: weight($S, R_k$) = min$_{R_i \in R}$(weight($S$, $R_i$));
    $R \setminus= R_k$; // eliminate $R_k$ from $R$
    $S \circ= R_k$; // append $R_k$ to S
  }
  Solutions += $S$;
}
return cheapest in Solutions
```

Heuristics: Greedy Algorithms

Often used relative weight function:

> *the product of the selectivities connecting relations in S with the new relation.*

This heuristics is sometimes called *MinSel*.

Heuristics: Greedy Algorithms

- The above algorithms produce left-deep trees.
- We now discuss Greedy Operator Ordering (GOO) which produces bushy trees.

Idea:

- A set of join trees `Trees` is initialized such that it contains all the relations to be joined.
- It then investigates all pairs of trees contained in `Tree`.
- Among all of these, the algorithm joins the two trees that result in the smallest intermediate result.
- The two trees are then eliminated from `Trees` and the new join tree joining them is added to it.

Heuristics: Greedy Algorithms

```
GOO($\{R_1, \ldots, R_n\}$)
**Input:** a set of relations to be joined
**Output:** join tree
Trees := $\{R_1, \ldots, R_n\}$
**while** (|Trees| != 1) {
  **find** $T_i, T_j$ ∈ Trees such that $|T_i \bowtie T_j|$ is minimal
       among all pairs of trees in Trees
  Trees −= $T_i$;
  Trees −= $T_j$;
  Trees += $T_i \bowtie T_j$;
}
**return** the tree contained in Trees;
```

IKKBZ

- Now: IKKBZ-Algorithm
- The query graph must be acyclic
- Produces optimal left-deep tree without cross products
- Cost function must have ASI property
- Join method must be fixed before hand

Scheme for the Cost Function

The IKKBZ-Algorithm considers only join operations that have a cost function of the form:

$$cost(R_i \bowtie R_j) = |R_i| * h_j(|R_j|).$$

where each R_j can have its own cost function h_j. We denote the set of h_j by H and parameterize cost functions with it. Example instanciations are

- $h_j \equiv 1.2$ for main memory hash-based joins
- $h_j \equiv id$ for nested-loops joins

Let us denote by n_i the cardinality of the relation R_i ($n_i := |R_i|$). Then, the $h_i(n_i)$ represent the costs per input tuple to be joined with R_i.

Overview of the Algorithm

For every relation R_k it computes the optimal join order under the assumption that R_k is the first relation in the join sequence. The resulting subproblems then resemble job-scheduling problems that can be solved by the Monma-Sidney-Algorithm.

Precedence Graph

Given a query graph and a starting relation R_k, we can compute the *precedence graph* $G = (V, E)$ rooted at R_k as follows:

- Make some relation $R_k \in V$ the root node of the precedence graph.
- As long as not all relations are included in the precedence graph: Choose a relation $R_i \in V$, such that $(R_j, R_i) \in E$ is an edge in the query graph and R_j is already contained in the (partial) precedence graph constructed so far and R_i is not. Add R_j and the edge $R_j \rightarrow R_i$ to the precedence graph.

Conformance to a Precedence Graph

A sequence $S = v_1, \ldots, v_k$ of nodes conforms to a precedence graph $G = (V, E)$ if the following conditions are satisfied:

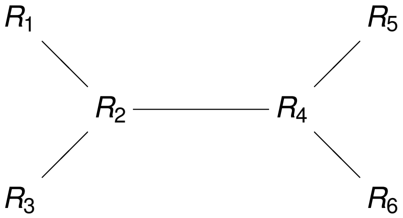
1. for all i $(2 \leq i \leq k)$ there exists a j $(1 \leq j < i)$ with $(v_j, v_i) \in E$ and
2. there is no edge $(v_i, v_j) \in E$ for $i > j$.

Notations

For non-empty sequences U and V in a precedence graph, we write $U \to V$ if, according to the precedence graph U must occur before V. This requires U and V to be disjoint. More precisely, if $U \to V$ then there can only be paths from nodes in U to nodes in V and at least one such path exists.
Define

$$\begin{aligned} R_{1,2,\ldots,k} &:= R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k \\ n_{1,2,\ldots,k} &:= |R_{1,2,\ldots,k}| \end{aligned}$$

Sample Query Graph



Sample Precedence Graph and Corresponding Join Tree

$$\begin{tikzcd}
R_1 \arrow[d] \\
R_2 \arrow[d] \\
R_3 \arrow[d] \\
R_4 \arrow[d] \\
R_5 \arrow[d] \\
R_6
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & & \bowtie \arrow[dl] \arrow[dr] & \\
 & & & & \bowtie \arrow[dl] \arrow[dr] & & R_6 \\
 & & & \bowtie \arrow[dl] \arrow[dr] & & R_5 & \\
 & & \bowtie \arrow[dl] \arrow[dr] & & R_4 & & \\
 & \bowtie \arrow[dl] \arrow[dr] & & R_3 & & & \\
R_1 & & R_2 & & & &
\end{tikzcd}$$

Relations Joined Before a Given One

For a given precedence graph, let R_i be a relation and $\mathcal{R}_i$ be the set of relations from which there exists a path to R_i.
Then, in any join tree adhering to the precedence graph, all relations in $\mathcal{R}_i$ and only those will be joined before R_i.
Hence, we can define $s_i = \prod_{R_j \in \mathcal{R}_i} f_{i,j}$ for $i > 1$.
Note that for any i only one j with $f_{i,j} \neq 1$ exists in the product.

If the Precedence Graph is Totally Ordered

If the precedence graph is a chain, then the following holds:

$$n_{1,2,\ldots,k+1} \quad = \quad n_{1,2\ldots,k} * s_{k+1} * n_{k+1}$$

We define $s_1 = 1$. Then we have

$$n_{1,2} \quad = \quad s_2 * (n_1 * n_2) = (s_1 * s_2) * (n_1 * n_2)$$

and in general

$$n_{1,2,\ldots,k} = \prod_{i=1}^{k} (s_i * n_i).$$

We call the s_i selectivities although they depend on the precedence graph.

Costs of a Totally Ordered Precedence Graph

The costs for a totally ordered precedence graph G can be computed as follows:

$$
\begin{aligned}
Cost_H(G) &= \sum_{i=2}^{n} [n_{1,2,\ldots,i-1} * h_i(n_i)] \\
&= \sum_{i=2}^{n} [(\prod_{j=1}^{i-1} s_j * n_j) * h_i(n_i)]
\end{aligned}
$$

- If we define $h_i(n_i) = s_i n_i$, then $Cost_H \equiv C_{\text{out}}$.
- The factor $s_i n_i$ determines by how much the input relation to be joined with R_i changes its cardinality after the join has been performed.
- If $s_i n_i$ is less than one, we call the join *decreasing*, if it is larger than one, we call the join *increasing*.

Recursive Formulation of the Costs

Define the cost function C_H as follows:

$$\begin{array}{rcl} C_H(\epsilon) & = & 0 \\ C_H(R_j) & = & 0 \quad \text{if } R_j \text{ is the root} \\ C_H(R_j) & = & h_j(n_j) \quad \text{else} \\ C_H(S_1 S_2) & = & C_H(S_1) + T(S_1) * C_H(S_2) \end{array}$$

where

$$\begin{array}{rcl} T(\epsilon) & = & 1 \\ T(S) & = & \prod_{R_i \in S} (s_i * n_i) \end{array}$$

C_H is well-defined and $C_H(G) = Cost_H(G)$.

ASI-Property

Let A and B be two sequences and V and U two non-empty sequences. We say a cost function C has the *adjacent sequence interchange property* (ASI property), if and only if there exist two sequences T and S and a rank function defined as

$$rank(S) = \frac{T(S) - 1}{C(S)}$$

such that for non-empty sequences S the following holds

$$C(AUVB) \leq C(AVUB) \quad \prec\succ \quad rank(U) \leq rank(V) \qquad (16)$$

if $AUVB$ and $AVUB$ satisfy the precedence constraints imposed by a given precedence graph.

The First Lemma

Lemma: The cost function C_H defined has the ASI-Property.
The proof is very simple. Using the definition of C_H, we have

$$\begin{aligned} C_H(AUVB) &= C_H(A) \\ &\quad + T(A)C_H(U) \\ &\quad + T(A)T(U)C_H(V) \\ &\quad + T(A)T(U)T(V)C_H(B) \end{aligned}$$

and, hence,

$$\begin{aligned} C_H(AUVB) - C_H(AVUB) &= T(A)[C_H(V)(T(U)-1) - C_H(U)(T(V)-1)] \\ &= T(A)C_H(U)C_H(V)[rank(U) - rank(V)] \end{aligned}$$

The lemma follows. $\square$

Module

Let $M = \{A_1, \ldots, A_n\}$ be a set of sequences of nodes in a given precedence graph.
Then, M is called a *module*, if for all sequences B that do not overlap with the sequences in M one of the following conditions holds:

- $B \to A_i, \forall\ 1 \leq i \leq n$
- $A_i \to B, \forall\ 1 \leq i \leq n$
- $B \not\to A_i$ and $A_i \not\to B, \forall\ 1 \leq i \leq n$

The Second Lemma

Lemma: Let C be any cost function with the ASI property and $\{A, B\}$ a module. If $A \rightarrow B$ and additionally $rank(B) \leq rank(A)$, then we find an optimal sequence among those in which B directly follows A. $\square$

Proof: Every optimal permutation must have the form (U, A, V, B, W) since $A \rightarrow B$.
Assumption: $V \neq \epsilon$.
If $rank(V) \leq rank(A)$, then we can exchange V and A without increasing the costs.
If $rank(A) \leq rank(V)$, we have $rank(B) \leq rank(V)$ due to the transitivity of $\leq$.
Hence, we can exchange B and V without increasing the costs.
Both exchanges produce legal sequences obeying the precedence graph since $\{A, B\}$ is a module. $\square$

Contradictory Sequence

If the precedence graph demands $A \rightarrow B$ but $rank(B) \leq rank(A)$, we speak of *contradictory sequences* A and B.

Compound Relation

Since the lemma shows that no non-empty subsequence can occur between A and B, we will combine A and B into a new single node replacing A and B.
This node represents a *compound relation* comprising of all relations in A and B.
Its cardinality is computed by multiplying the cardinalities of all relations occurring in A and B, and its selectivity s is the product of all the selectivities s_i of the relations R_i contained in A and B.

Normalization and Denormalization

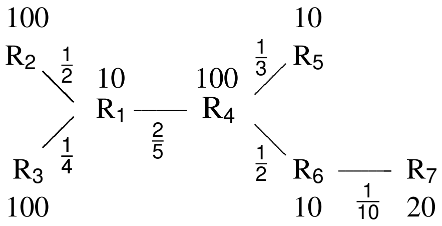
The continued process of this step until no more contradictory sequences exist is called *normalization*.
The opposite step, replacing a compound node by the sequence of relations it was derived from, is called *denormalization*.

```
IKKBZ($G$)
**Input:** an acyclic query graph $G$ for relations $R_1, \ldots, R_n$
**Output:** the best left-deep tree
$R = \emptyset$;
for ($i = 1$; $i \leq n$; $++i$) {
  Let $G_i$ be the precedence graph derived from $G$
      and rooted at $R_i$;
  $T$ = IKKBZ-Sub($G_i$);
  $R += T$;
}
return best of $R$;
```

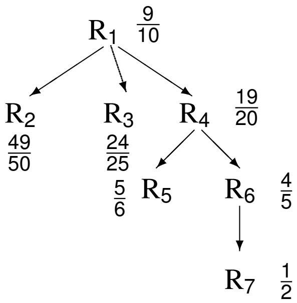
```
IKKBZ-Sub($G_i$)
Input:  a precedence graph $G_i$ for relations $R_1,\ldots,R_n$
         rooted at some $R_i$
Output:  the optimal left-deep tree under $G_i$
while ($G_i$ is not a chain) {
  let $r$ be the root of a subtree whose subtrees are chains;
  IKKBZ-Normalize($r$);
  merge the chains under $r$ according to the rank function
    in ascending order;
}
```

```
IKKBZ-Normalize($r$)
Input:  root $r$ of a subtree $T$ of a precedence graph $G = (V, E)$
Output:  a normalized subchain
while  ($\exists$ $r', c \in V$,  $r \rightarrow^* r', (r', c) \in E$:  rank($r'$) > rank($c$)) {
  replace $r'$ by a compound relation $r''$ that represents $r'c$;
};
```

Sample Query Graph and One Precedence Graph



A)



B)

Round One

$$\begin{tikzcd}
 & R_1\ \frac{9}{10} \arrow[dl] \arrow[d, dotted] \arrow[dr] & & \\
R_2\ \frac{49}{50} & R_3\ \frac{24}{25} & R_4\ \frac{19}{20} \arrow[dl] \arrow[dr] & \\
 & \frac{5}{6}\ R_5 & & R_6\ \frac{4}{5} \arrow[d] \\
 & & & R_7\ \frac{1}{2}
\end{tikzcd}$$

B)

$$\begin{tikzcd}
 & R_1\ \frac{9}{10} \arrow[dl] \arrow[d, dotted] \arrow[dr] & & \\
\frac{49}{50}\ R_2 & \frac{24}{25}\ R_3 & R_4\ \frac{19}{20} \arrow[dl] \arrow[dr] & \\
 & \frac{5}{6}\ R_5 & & R_{6,7}\ \frac{9}{10}
\end{tikzcd}$$

C)

Round Two

$$\begin{tikzcd}
 & R_1\ \frac{9}{10} \arrow[dl] \arrow[d, dashed] \arrow[dr] & & \\
\frac{49}{50}\ R_2 & \frac{24}{25}\ R_3 & R_4\ \frac{19}{20} \arrow[dl] \arrow[dr] & \\
 & \frac{5}{6}\ R_5 & & R_{6,7}\ \frac{9}{10}
\end{tikzcd}$$

C)

$$\begin{tikzcd}
 & R_1\ \frac{9}{10} \arrow[dl] \arrow[d, dashed] \arrow[dr] & \\
R_2 & R_3 & R_4\ \frac{19}{20} \arrow[d] \\
\frac{49}{50} & \frac{24}{25} & R_5\ \frac{5}{6} \arrow[d] \\
 & & R_{6,7}\ \frac{9}{10}
\end{tikzcd}$$

D)

Round Three

$$\begin{tikzcd}
 & R_1 \quad \frac{9}{10} \arrow[dl] \arrow[d, dotted] \arrow[dr] & \\
R_2 \atop \frac{49}{50} & R_3 \atop \frac{24}{25} & R_4 \quad \frac{19}{20} \arrow[d] \\
 & & R_5 \quad \frac{5}{6} \arrow[d] \\
 & & R_{6,7} \quad \frac{9}{10}
\end{tikzcd}$$

D)

$$\begin{tikzcd}
 & R_1 \quad \frac{9}{10} \arrow[dl] \arrow[d, dotted] \arrow[dr] & \\
R_2 \atop \frac{49}{50} & R_3 \atop \frac{24}{25} & R_{4,5} \quad \frac{119}{120} \arrow[d] \\
 & & R_{6,7} \quad \frac{9}{10}
\end{tikzcd}$$

E)

Round Four and Final Result

$$\begin{tikzcd}
& R_1 \quad \frac{9}{10} \arrow[dl] \arrow[d, dotted] \arrow[dr] & \\
R_2 & R_3 & R_{4,5} \quad \frac{119}{120} \arrow[d] \\
\frac{49}{50} & \frac{24}{25} & R_{6,7} \quad \frac{9}{10}
\end{tikzcd}$$

E)

$$\begin{tikzcd}
& R_1 \quad \frac{9}{10} \arrow[dl] \arrow[d, dotted] \arrow[dr] & \\
R_2 & R_3 & R_{4,5,6,7} \\
\frac{49}{50} & \frac{24}{25} & \frac{1119}{1200}
\end{tikzcd}$$

F)

$$\begin{tikzcd}
R_1 \arrow[r] & R_2 \arrow[r] & R_3 \arrow[r] & R_{4,5,6,7}
\end{tikzcd}$$

G)

$$\begin{tikzcd}
R_1 \arrow[r] & R_2 \arrow[r] & R_3 \arrow[r] & R_4 \arrow[r] & R_5 \arrow[r] & R_6 \arrow[r] & R_7
\end{tikzcd}$$

H)

Maximum-Value-Precedence Algorithm

Given a conjunctive query with join predicates P.
For a join predicate $p \in P$, we denote by $\mathcal{R}(p)$ the relations whose attributes are mentioned in p.
Definition The *directed join graph* of a conjunctive query with join predicates P is a triple $G = (V, E_p, E_v)$ where V is the set of nodes and E_p and E_v are sets of directed edges defined as follows.
For any two nodes $u, v \in V$, if $\mathcal{R}(u) \cap \mathcal{R}(v) \neq \emptyset$ then $(u, v) \in E_p$ and $(v, u) \in E_p$.
If $\mathcal{R}(u) \cap \mathcal{R}(v) = \emptyset$, then $(u, v) \in E_v$ and $(v, u) \in E_v$.
The edges in E_p are called *physical edges*, those in E_v *virtual edges*. $\square$
Note in G for every two nodes u, v, there is an edge (u, v) that is either physical or virtual. Hence, G is a clique.

Example: Query Graph and Directed Join Graph

I

$$\begin{tikzcd}
R_1 \arrow[r, dash] & R_2 \arrow[r, dash] & R_3 \arrow[r, dash] & R_4
\end{tikzcd}$$

II

$$\begin{tikzcd}
& p_{2,3} \arrow[dl, shift right] \arrow[dr, shift left] & \\
p_{1,2} \arrow[ur, shift right] \arrow[rr, dotted, shift left] & & p_{3,4} \arrow[ul, shift left] \arrow[ll, dotted, shift left]
\end{tikzcd}$$

Extracting Join Trees from the Directed Join Graph

- Every spanning tree in the Directed Join Graph leads to a join tree.

Example: Spanning Tree and Join Tree

I
$$\begin{tikzcd} R_1 \arrow[r, dash] & R_2 \arrow[r, dash] & R_3 \arrow[r, dash] & R_4 \end{tikzcd}$$

II
$$\begin{tikzcd} & p_{2,3} \arrow[dl, shift right] \arrow[dr, shift left] & \\ p_{1,2} \arrow[ur, shift right] \arrow[rr, dotted, shift left] & & p_{3,4} \arrow[ul, shift left] \arrow[ll, dotted, shift left] \end{tikzcd}$$

III a)
$$\begin{tikzcd} p_{3,4} \\ p_{2,3} \arrow[u] \\ p_{1,2} \arrow[u] \end{tikzcd}$$

III b)
$$\begin{tikzcd} & & & \bowtie \arrow[dl, dash] \arrow[dr, dash] & \\ & & \bowtie \arrow[dl, dash] \arrow[dr, dash] & & R_4 \\ & \bowtie \arrow[dl, dash] \arrow[dr, dash] & & R_3 & \\ R_1 & & R_2 & & \end{tikzcd}$$

Example: Spanning Tree and Join Tree

I

$$\begin{tikzcd}
R_1 \arrow[r, dash] & R_2 \arrow[r, dash] & R_3 \arrow[r, dash] & R_4
\end{tikzcd}$$

II

$$\begin{tikzcd}
& p_{2,3} \arrow[dl, shift right] \arrow[dr, shift left] & \\
p_{1,2} \arrow[ur, shift right] \arrow[rr, dotted, shift left] & & p_{3,4} \arrow[ul, shift left] \arrow[ll, dotted, shift left]
\end{tikzcd}$$

IV a)

$$\begin{tikzcd}
p_{2,3} \\
p_{3,4} \arrow[u] \\
p_{1,2} \arrow[u, dotted]
\end{tikzcd}$$

IV b)

$$\begin{tikzcd}
& & \bowtie \arrow[dl, dash] \arrow[dr, dash] & & \\
& \bowtie \arrow[dl, dash] \arrow[d, dash] & & \bowtie \arrow[d, dash] \arrow[dr, dash] & \\
R_1 & R_2 & & R_3 & R_4
\end{tikzcd}$$

Example: Spanning Tree and Join Tree

I

$$\begin{tikzcd}
R_1 \arrow[r, dash] & R_2 \arrow[r, dash] & R_3 \arrow[r, dash] & R_4
\end{tikzcd}$$

II

$$\begin{tikzcd}
 & p_{2,3} \arrow[dl, shift right] \arrow[dr, shift left] & \\
p_{1,2} \arrow[ur, shift right] \arrow[rr, dotted, shift left] & & p_{3,4} \arrow[ul, shift left] \arrow[ll, dotted, shift left]
\end{tikzcd}$$

V a)

$$\begin{tikzcd}
 & p_{2,3} & \\
p_{1,2} \arrow[ur] & & p_{3,4} \arrow[ul]
\end{tikzcd}$$

V b)

$$\begin{tikzcd}
 & & & \bowtie \arrow[dll, dash] \arrow[drr, dash] & & & \\
 & \bowtie \arrow[dl, dash] \arrow[dr, dash] & & & & \bowtie \arrow[dl, dash] \arrow[dr, dash] & \\
R_1 & & R_2 & & R_3 & & R_4
\end{tikzcd}$$

Problems

The next example shows that problems can occur.

Example

I

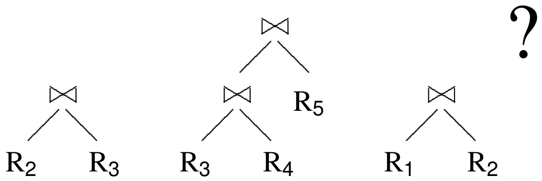
$$\begin{tikzcd} R_1 \arrow[r, dash] & R_2 \arrow[r, dash] & R_3 \arrow[r, dash] & R_4 \arrow[r, dash] & R_5 \end{tikzcd}$$

II

$$\begin{tikzcd} p_{1,2} \arrow[r, shift left] & p_{2,3} \arrow[l, shift left] \arrow[r, shift left] & p_{3,4} \arrow[l, shift left] \arrow[r, shift left] & p_{4,5} \arrow[l, shift left] \end{tikzcd}$$

$$\begin{tikzcd} & p_{4,5} & \\ p_{2,3} \arrow[ur, dotted] & & p_{1,2} \arrow[ul, dotted] \\ & & p_{3,4} \arrow[u, dotted] \end{tikzcd}$$

III a)



III b)

Effective Spanning Trees

It can be shown that a spanning tree $T = (V, E)$ is *effective*, if it satisfies the following conditions:

1. T is a binary tree,
2. for all inner nodes v and node u with $(u, v) \in E$: $\mathcal{R}^*(T(u)) \cap \mathcal{R}(v) \neq \emptyset$, and
3. for all nodes v, u_1, u_2 with $u_1 \neq u_2$, $(u_1, v) \in E$, and $(u_2, v) \in E$ one of the following conditions holds:
   3.1 $((\mathcal{R}^*(T(u_1)) \cap \mathcal{R}(v)) \cap (\mathcal{R}^*(T(u_2)) \cap \mathcal{R}(v))) = \emptyset$ or
   3.2 $(\mathcal{R}^*(T(u_1)) = \mathcal{R}(v)) \vee (\mathcal{R}^*(T(u_2)) = \mathcal{R}(v))$.

Thereby, we denote by $T(v)$ the partial tree rooted at v and by $\mathcal{R}^*(T') = \cup_{v \in T'} \mathcal{R}(v)$ the set of all relations in subtree T'.

Adding Weights to the Edges

For two nodes $v, u \in V$ define $u \sqcap v := \mathcal{R}(u) \cap \mathcal{R}(v)$.
For simplicity, we assume that every predicate involves exactly two relations.
Then for all $u, v \in V$, $u \sqcap v$ contains a single relation.
Let $v \in V$ be a node with $\mathcal{R}(v) = \{R_i, R_j\}$.
We abbreviate $R_i \bowtie_v R_j$ by $\bowtie_v$.
Using these notations, we can attach weights to the edges to define the *weighted directed join graph*.

Adding Weights to the Edges

Definition Let $G = (V, E_p, E_v)$ be a directed join graph for a conjunctive query with join predicates P. The *weighted directed join graph* is derived from G by attaching a weight with each edge as follows:

- Let $(u, v) \in E_p$ be a physical edge. The weight $w_{u,v}$ of (u, v) is defined as

$$w_{u,v} = \frac{| \bowtie_u |}{|u \sqcap v|}.$$

- For virtual edges $(u, v) \in E_v$, we define $w_{u,v} = 1$.

Remark on Edge Weights

The weights of physical edges are equal to the s_i used in the IKKBZ-Algorithm.
Assume $\mathcal{R}(u) = \{R_1, R_2\}$, $\mathcal{R}(v) = \{R_2, R_3\}$. Then

$$\begin{aligned} w_{u,v} &= \frac{|\bowtie_u|}{|u \sqcap v|} \\ &= \frac{|R_1 \bowtie_u R_2|}{|R_2|} \\ &= \frac{f_{1,2}\,|R_1|\,|R_2|}{|R_2|} \\ &= f_{1,2}\,|R_1| \end{aligned}$$

Hence, if the join $R_1 \bowtie_u R_2$ is executed before the join $R_2 \bowtie_v R_3$, the input size to the latter join changes by a factor $w_{u,v}$.

Adding Weights to the Nodes

The weight of a node reflects the change in cardinality to be expected when certain other joins have been executed before. They depend on a (partial) spanning tree S.
Given S, we denote by $\bowtie^S_{p_{i,j}}$ the result of the join $\bowtie_{p_{i,j}}$ if all joins preceding $p_{i,j}$ in S have been executed. Then the weight attached to node $p_{i,j}$ is defined as

$$w(p_{i,j}, S) = \frac{|\bowtie^S_{p_{i,j}}|}{|R_i \bowtie_{p_{i,j}} R_j|}.$$

Similarily, we define the cost of a node $p_{i,j}$ depending on other joins preceding it in some given spanning tree S. We denote this by `cost`$(p_{i,j}, S)$.
The actual cost function an be chosen arbitrarily.
If we have several join implementations: take minimum.

MVP-Algorithm: Outline

Two phases to build an effective spanning tree:

1. Take first those edges with a weight < 1.
2. Treat other edges

Thereby take care of the effectivness.

```
MVP($G$)
Input:  a weighted directed join graph $G = (V, E_p, E_v)$
Output:  an effective spanning tree
// $Q_1$, $Q_2$:  two priority queues with
$Q_1$.insert($V$); // smallest node weights $w(\cdot)$ first
$Q_2 = \emptyset$; // largest node weights $w(\cdot)$ first
$G' = (V', E')$ with $V' = V$ and $E' = E_p$; /* working graph */
// result:  effective spanning tree:
$S = (V_S, E_S)$ with $V_S = V$ and $E_S = \emptyset$;
```

```
while (!$Q_1$.empty() && $|E_S| < |V| - 1$) { /* Phase I */
  $v$ = $Q_1$.head();
  among all $(u, v) \in E'$, $w_{u,v} < 1$ such that
    $S' = (V, E'_S)$ with $E'_S = E_S \cup \{(u, v)\}$ is acyclic and effective
    select one that maximizes cost($\bowtie_v, S$) - cost($\bowtie_v, S'$);
  if (no such edge exists) {
    $Q_1$.remove($v$);
    $Q_2$.insert($v$);
    continue;
  }
  MvpUpdate($(u, v)$);
  recompute $w(\cdot)$ for $v$ and its ancestors; /* rearranges $Q_1$ */
}
```

```
while (!$Q_2$.empty() && $|E_S| < |V| - 1$) { /* Phase II */
  $v$ = $Q_2$.head();
  among all $(u, v), (v, u) \in E'$ denoted by $(x, y)$ henceforth
    such that
    $S' = (V, E'_S)$ with $E'_S = E_S \cup \{(x, y)\}$ is acyclic and effective
    select the one that minimizes cost($\bowtie_v, S'$) - cost($\bowtie_v, S$);
  MvpUpdate($(x, y)$);
  recompute $w(\cdot)$ for $y$ and its ancestors; /* rearranges $Q_2$ */
}
return $S$;
```

```
MvpUpdate($(u, v)$)
Input: an edge to be added to $S$
Output: side-effects on $S$, $G'$,
  $E_S \cup= \{(u, v)\}$;
  $E' \setminus= \{(u, v), (v, u)\}$;
  $E' \setminus= \{(u, w) | (u, w) \in E'\}$; /* (1) */
  $E' \cup= \{(v, w) | (u, w) \in E_p, (v, w) \in E_v\}$; /* (3) */
  if ($v$ has two inflowing edges in $S$) { /* (2) */
    $E' \setminus= \{(w, v) | (w, v) \in E'\}$;
  }
  if ($v$ has one outflowing edge in $S$) { /* (1) */
    $E' \setminus= \{(v, w) | (v, w) \in E'\}$;
  }
```

Dynamic Programming

- Optimality Principle
- Avoid duplicate work

Optimality Principle

Consider the two join trees

$$(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) \bowtie R_5$$

and

$$(((R_3 \bowtie R_1) \bowtie R_2) \bowtie R_4) \bowtie R_5.$$

If we know that $((R_1 \bowtie R_2) \bowtie R_3)$ is cheaper than $((R_3 \bowtie R_1) \bowtie R_2)$, we know that the first join tree is cheaper than the second join tree. Hence, we could avoid generating the second alternative and still won't miss the optimal join tree.

Optimality Principle

Optimality Principle for join ordering:

> *Let T be an optimal join tree for relations $R_1, \ldots, R_n$. Then, every subtree S of T must be an optimal join tree for the relations contained in it.*

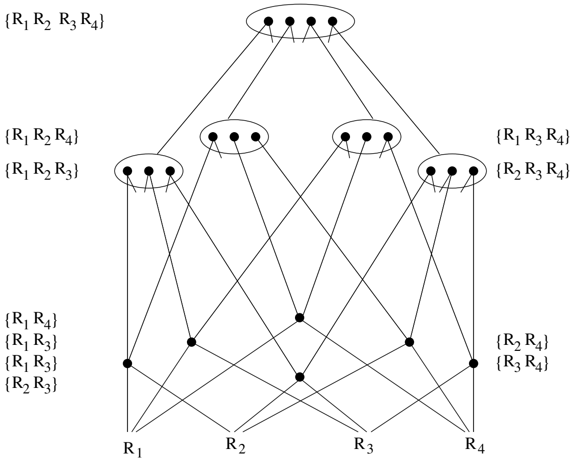
Remark: Optimality principle does not hold in the presence of properties.

Dynamic Programming

- Generate optimal join trees bottom up
- Start from optimal join trees of size one
- Build larger join trees for sizes $n > 1$ by (re-) using those of smaller sizes
- We use subroutine `CreateJoinTree` that joins two (sub-) trees

```
CreateJoinTree($T_1$, $T_2$)
Input:  two (optimal) join trees $T_1$ and $T_2$.
        for linear trees:  assume that $T_2$ is a single relation
Output:  an (optimal) join tree for joining $T_1$ and $T_2$.
BestTree = NULL;
for all implementations impl do {
  if(!RightDeepOnly)
    Tree = $T_1 \bowtie^{impl} T_2$
    if (BestTree == NULL || cost(BestTree) > cost(Tree))
      BestTree = Tree;
  if(!LeftDeepOnly)
    Tree = $T_2 \bowtie^{impl} T_1$
    if (BestTree == NULL || cost(BestTree) > cost(Tree))
      BestTree = Tree;
}
return BestTree;
```

Search space with sharing under Optimality Principle



```
DP-Linear-1({$R_1, \ldots, R_n$})
Input:  a set of relations to be joined
Output:  an optimal left-deep (right-deep, zig-zag) join tree
for (i = 1; i <= n; ++i) BestTree({$R_i$}) = $R_i$;
for (i = 1; i < n; ++i) {
  for all $S \subseteq \{R_1, \ldots, R_n\}$, $|S| = i$ do {
    for all $R_i \in \{R_1, \ldots, R_n\}$, $R_i \notin S$ do {
      if (NoCrossProducts && !connected({$R_i$}, $S$)) { continue; }
      CurrTree = CreateJoinTree(BestTree($S$),$R_i$);
      $S' = S \cup \{R_i\}$;
      if (BestTree($S'$) == NULL
         || cost(BestTree($S'$)) $>$ cost(CurrTree)) {
         BestTree($S'$) = CurrTree;
      }
    }
  }
}
return BestTree({$R_1, \ldots, R_n$});
```

Order in which subtrees are generated

The order in which subtrees are generated does not matter as long as the following condition is not violated:

> *Let S be a subset of $\{R_1, \ldots, R_n\}$. Then, before a join tree for S can be generated, the join trees for all relevant subsets of S must already be available.*

Exercise: fix the semantics of *relevant*

Generation in integer order

000	$\{\}$
001	$\{R_1\}$
010	$\{R_2\}$
011	$\{R_1, R_2\}$
100	$\{R_3\}$
101	$\{R_1, R_3\}$
110	$\{R_2, R_3\}$
111	$\{R_1, R_2, R_3\}$

```
DP-Linear-2($\{R_1, \ldots, R_n\}$)
Input:  a set of relations to be joined
Output:  an optimal left-deep (right-deep, zig-zag) join tree
for (i = 1; i <= n; ++i) { BestTree($1 << i-1$) = $R_i$; }
for (S = 1; S < $2^n$; ++S) {
  if (BestTree($S$) != NULL) continue;
  for all $i \in S$ do {
    $S' = S \setminus \{i\}$;
    CurrTree = CreateJoinTree(BestTree($S'$),$R_i$);
    if (cost(BestTree($S$)) > cost(CurrTree)) {
      BestTree($S$) = CurrTree;
    }
  }
}
return BestTree($2^n - 1$);
```

```
DP-Bushy($\{R_1, \ldots, R_n\}$)
**Input:** a set of relations to be joined
**Output:** an optimal bushy join tree
for (i = 1; i <= n; ++i)
  BestTree(1 $<<$ $i-1$) = $R_i$;
for (S = 1; S $<$ $2^n$; ++S) {
  if (BestTree($S$) != NULL) continue;
  for all $S_1 \subset S$, $S_1 \neq \emptyset$ do
    $S_2 = S \setminus S_1$;
    CurrTree = CreateJoinTree(BestTree($S_1$), BestTree($S_2$));
    if (BestTree($S$) == NULL
      || cost(BestTree($S$)) $>$ cost(CurrTree))
      BestTree($S$) = CurrTree;
}
return BestTree($2^n - 1$);
```

Subset Generation for Bushy Trees

```
$S_1$ = S & - S;
do {
  /* do something with subset $S_1$ */
  $S_1$ = S & ($S_1$ - S);
} while ($S_1$ != S);
```

S represents the input set. S_1 iterates through all subsets of S where S itself and the empty set are not considered.

Number of join trees investigated by DP

	without cross products			with cross products	
	chain		star	any query graph	
	linear	bushy	linear	linear	bushy
n	$(n-1)^2$	$(n^3-n)/6$	$(n-1)2^{n-2}$	$n2^{n-1}-n(n+1)/2$	$(3^n-2^{n+1}+1)/2$
2	1	1	1	1	1
3	4	4	4	6	6
4	9	10	12	22	25
5	16	20	32	65	90
6	25	35	80	171	301
7	36	56	192	420	966
8	49	84	448	988	3025
9	64	120	1024	2259	9330
10	81	165	2304	5065	28501

Optimal Bushy Trees without Cross Products

Given:	Connected join graph
Problem:	Generate optimal bushy trees without cross products

Csg-Cmp-Pairs

Let S_1 and S_2 be subsets of the nodes (relations) of the query graph. We say (S_1, S_2) is a *csg-cmp-pair*, if and only if

1. S_1 induces a connected subgraph of the query graph,
2. S_2 induces a connected subgraph of the query graph,
3. S_1 and S_2 are disjoint, and
4. there exists at least one edge connected a node in S_1 to a node in S_2.

If (S_1, S_2) is a csg-cmp-pair, then (S_2, S_1) is a valid csg-cmp-pair.

Csg-Cmp-Pairs and Join Trees

Let (S_1, S_2) be a csg-cmp-pair and T_i be a join tree for S_i. Then we can construct two valid join tree:

$$T_1 \bowtie T_2 \text{ and } T_2 \bowtie T_1$$

Hence, the number of csg-cmp-pairs coincides with the search space DP explores. In fact, the number of csg-cmp-pairs is a lower bound for the complexity of DP.

If `CreateJoinTree` considers commutativity of joins, the number of calls to it is precisely expressed by the count of non-symmetric csg-cmp-pairs. In other implementations `CreateJoinTree` might be called for all csg-cmp-pairs and, thus, may not consider commutativity.

The Number of Csg-Cmp-Pairs

Let us denote the number of non-symmetric csg-cmp-pairs by $\#\texttt{ccp}$. Then

$$\begin{aligned}
\#\texttt{ccp}^{\text{chain}}(n) &= \frac{1}{6}(n^3 - n) \\
\#\texttt{ccp}^{\text{cycle}}(n) &= (n^3 - 2n^2 + n)/2 \\
\#\texttt{ccp}^{\text{star}}(n) &= (n-1)2^{n-2} \\
\#\texttt{ccp}^{\text{clique}}(n) &= (3^n - 2^{n+1} + 1)/2
\end{aligned}$$

These numbers have to be multiplied by two if we want to count all csg-cmp-pairs.

DPsize

```
for all $R_i \in R$ BestPlan($\{R_i\}$) = $R_i$;
for all $1 < s \leq n$ ascending // size of plan
for all $1 \leq s_1 \leq s/2$ // size of left subplan
  $s_2 = s - s_1$; // size of right subplan
  for all $p_1$ = BestPlan($S_1 \subset R : |S_1| = s_1$)
       $p_2$ = BestPlan($S_2 \subset R : |S_2| = s_2$)
    ++InnerCounter;
    if ($\emptyset \neq S_1 \cap S_2$) continue;
    if not ($S_1$ connected to $S_2$) continue;
    ++CsgCmpPairCounter;
    CurrPlan = CreateJoinTree($p_1, p_2$);
    if (cost(BestPlan($S_1 \cup S_2$)) $>$ cost(CurrPlan))
      BestPlan($S_1 \cup S_2$) = CurrPlan;
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan($\{R_0, \ldots, R_{n-1}\}$);
```

Analysis: DPsize

$$I^{chain}_{DPsize}(n) = \begin{cases} 1/48(5n^4 + 6n^3 - 14n^2 - 12n) & n \text{ even} \\ 1/48(5n^4 + 6n^3 - 14n^2 - 6n + 11) & n \text{ odd} \end{cases}$$

$$I^{cycle}_{DPsize}(n) = \begin{cases} \frac{1}{4}(n^4 - n^3 - n^2) & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n) & n \text{ odd} \end{cases}$$

$$I^{star}_{DPsize}(n) = \begin{cases} 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + q(n) & n \text{ even} \\ 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + 1/4\binom{n-1}{(n-1)/2} + q(n) & n \text{ odd} \end{cases}$$

with $q(n) = n2^{n-1} - 5 * 2^{n-3} + 1/2(n^2 - 5n + 4)$

$$I^{clique}_{DPsize}(n) = \begin{cases} 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} - 1/4\binom{n}{n/2} + 1 & n \text{ even} \\ 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} + 1 & n \text{ odd} \end{cases}$$

DPsub

```
DPsub ($\{R_0, \ldots, R_{n-1}\}$)
for all $R_i \in R$ BestPlan($\{R_i\}$) = $R_i$;
for $1 \leq i < 2^n - 1$ ascending
    $S = \{R_j \in R | (\lfloor i/2^j \rfloor \mathit{mod}\, 2) = 1\}$
    if not (connected $S$) continue;
    for all $S_1 \subset S$, $S_1 \neq \emptyset$ do
        ++InnerCounter;
        $S_2 = S \setminus S_1$;
        if not (connected $S_1$) continue;
        if not (connected $S_2$) continue;
        if not ($S_1$ connected to $S_2$) continue;
        ++CsgCmpPairCounter;
        $p_1$ = BestPlan($S_1$); $p_2$ = BestPlan($S_2$);
        CurrPlan = CreateJoinTree($p_1$, $p_2$);
        if (cost(BestPlan($S$)) > cost(CurrPlan))
            BestPlan($S$) = CurrPlan;
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan($\{R_0, \ldots, R_{n-1}\}$);
```

Analysis: DPsub

$$\begin{aligned}
I^{\text{chain}}_{\text{DPsub}}(n) &= 2^{n+2} - n^2 - 3n - 4 \\
I^{\text{cycle}}_{\text{DPsub}}(n) &= n2^n + 2^n - 2n^2 - 2 \\
I^{\text{star}}_{\text{DPsub}}(n) &= 2 * 3^{n-1} - 2^n \\
I^{\text{clique}}_{\text{DPsub}}(n) &= 3^n - 2^{n+1} + 1
\end{aligned}$$

Sample Numbers for InnerCounter

	Chain		
n	#ccp	DPsub	DPsize
5	20	84	73
10	165	3.962	1.135
15	560	130.798	5.628
20	1330	4.193.840	17.545
	Cycle		
	#ccp	DPsub	DPsize
5	40	140	120
10	405	11.062	2.225
15	1470	523.836	11.760
20	3610	22.019.294	37.900

	Star		
n	`#ccp`	`DPsub`	`DPsize`
5	32	130	110
10	2.304	38.342	57.888
15	114.688	9.533.170	57.305.929
20	4.980.736	2.323.474.358	59.892.991.338
	Clique		
n	`#ccp`	`DPsub`	`DPsize`
5	90	180	280
10	28.501	57.002	306.991
15	7.141.686	14.283.372	307.173.877
20	1.742.343.625	3.484.687.250	309.338.182.241

Algorithm DPccp

```
for all ($R_i \in \mathcal{R}$) BestPlan($\{R_i\}$) = $R_i$;
forall csg-cmp-pairs $(S_1, S_2)$, $S = S_1 \cup S_2$
  ++InnerCounter;
  ++OnoLohmanCounter;
  $p_1$ = BestPlan($S_1$);
  $p_2$ = BestPlan($S_2$);
  CurrPlan = CreateJoinTree($p_1$, $p_2$);
  if (cost(BestPlan($S$)) > cost(CurrPlan))
    BestPlan($S$) = CurrPlan;
  CurrPlan = CreateJoinTree($p_2$, $p_1$);
  if (cost(BestPlan($S$)) > cost(CurrPlan))
    BestPlan($S$) = CurrPlan;
CsgCmpPairCounter = 2 * OnoLohmanCounter;
return BestPlan($\{R_0, \ldots, R_{n-1}\}$);
```

Notation

Let $G = (V, E)$ be an undirected graph.
For a node $v \in V$ define the *neighborhood* $\boldsymbol{N}(v)$ of v as

$$\boldsymbol{N}(v) := \{v' | (v, v') \in E\}$$

For a subset $S \subseteq V$ of V we define the *neighborhood* of S as

$$\boldsymbol{N}(S) := \cup_{v \in S} \boldsymbol{N}(v) \setminus S$$

The neighborhood of a set of nodes thus consists of all nodes reachable by a single edge.
Note that for all $S, S' \subset V$ we have
$\boldsymbol{N}(S \cup S') = (\boldsymbol{N}(S) \cup \boldsymbol{N}(S')) \setminus (S \cup S')$. This allows for an efficient bottom-up calculation of neighborhoods.
$\mathcal{B}_i = \{v_j | j \leq i\}$

Algorithm EnumerateCsg

```
EnumerateCsg
Input: a connected query graph $G = (V, E)$
Output: emits all subsets of $V$ inducing a connected subgraph
of $G$

for all $i \in [n-1, \ldots, 0]$ descending {
    emit $\{v_i\}$;
    EnumerateCsgRec($G$, $\{v_i\}$, $\mathcal{B}_i$);
}
```

Subroutine EnumerateCsgRec

```
EnumerateCsgRec($G$, $S$, $X$)
$N = \boldsymbol{N}(S) \setminus X$;
**for all** $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    **emit** $(S \cup S')$;
}
**for all** $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec($G$, $(S \cup S')$, $(X \cup N)$);
}
```

Example

$$\begin{tikzcd}
 & R_0 \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
R_1 \arrow[dr, dash] & R_2 \arrow[r, dash] \arrow[d, dash] & R_3 \arrow[dl, dash] \\
 & R_4 &
\end{tikzcd}$$

S	X	N	emit/S
$\{4\}$	$\{0, 1, 2, 3, 4\}$	$\emptyset$	
$\{3\}$	$\{0, 1, 2, 3\}$	$\{4\}$	
			$\{3, 4\}$
$\{2\}$	$\{0, 1, 2\}$	$\{3, 4\}$	
			$\{2, 3\}$
			$\{2, 4\}$
			$\{2, 3, 4\}$
$\{1\}$	$\{0, 1\}$	$\{4\}$	
			$\{1, 4\}$
$\rightarrow \{1, 4\}$	$\{0, 1, 4\}$	$\{2, 3\}$	
			$\{1, 2, 4\}$
			$\{1, 3, 4\}$
			$\{1, 2, 3, 4\}$

Algorithm EnumerateCmp

```
EnumerateCmp
Input: a connected query graph $G = (V, E)$, a connected sub-
set $S_1$
Output: emits all complements $S_2$ for $S_1$ such that $(S_1, S_2)$ is a
csg-cmp-pair

$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending $i$) {
    emit $\{v_i\}$;
    EnumerateCsgRec($G$, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}
```

where $\min(S_1) := \min(\{i | v_i \in S_1\})$.

Memoization

- Recursive generation of join trees
- Easier code
- Memoize already generated join trees in order to avoid duplicate work

```
MemoizationJoinOrdering($R$)
**Input:** a set of relations $R$
**Output:** an optimal join tree for $S$
for (i = 1; i <= n; ++i) {
  BestTree($\{R_i\}$) = $R_i$;
}
return MemoizationJoinOrderingSub($R$);
```

```
MemoizationJoinOrderingSub($S$)
**Input:** a (sub-) set of relations $S$
**Output:** an optimal join tree for $S$
**if**(NULL == BestTree($S$)) {
  **for all** $S_1 \subset S$ **do** {
    $S_2 = S \setminus S_1$;
    CurrTree = CreateJoinTree(
          MemoizationJoinOrderingSub($S_1$),
          MemoizationJoinOrderingSub($S_2$));
    **if** (cost(BestTree($S$)) > cost(CurrTree)) {
      BestTree($S$) = CurrTree;
    }
  }
}
**return** BestTree($S$);
```

Join Ordering by Generating Permutations

- Left-deep trees are permutations of the relations to be joined
- Permutations can be generated directly
- Generating all permutations is too expensive
- Some permutations can be ignored:
  Consider the join sequence $R_1 R_2 R_3 R_4$. If we now that $R_1 R_3 R_2$ is cheaper than $R_1 R_2 R_3$, then we do not have to consider $R_1 R_2 R_3 R_4$.

Idea: successively add a relation.
Thereby: an extended sequence is only explored if exchanging the last two relations does not result in a cheaper sequence.

```
ConstructPermutations(Query Specification)
**Input:** query specification for relations $\{R_1, \ldots, R_n\}$
**Output:** optimal left-deep tree
BestPermutation = NULL;
Prefix = $\epsilon$;
Rest = $\{R_1, \ldots, R_n\}$;
ConstructPermutationsSub(Prefix, Rest);
**return** BestPermutation
```

```
ConstructPermutationsSub(Prefix, Rest)
Input:  a prefix of a permutation, relations to be added (Rest)
Ouput:  none, side-effect on BestPermutation
if (Rest == $\emptyset$)
  if (cost(Prefix) $<$ cost(BestPermutation))
    BestPermutation = Prefix;
  return
foreach ($R_i$, $R_j$ $\in$ Rest)
  if (cost(Prefix $\circ$ $\langle R_i, R_j \rangle$) $\leq$ cost(Prefix $\circ$ $\langle R_j, R_i \rangle$))
    ConstructPermutationsSub(Prefix $\circ$ $\langle R_i \rangle$, Rest $\setminus$ $\{R_i\}$);
  if (cost(Prefix $\circ$ $\langle R_j, R_i \rangle$) $\leq$ cost(Prefix $\circ$ $\langle R_i, R_j \rangle$))
    ConstructPermutationsSub(Prefix $\circ$ $\langle R_j \rangle$, Rest $\setminus$ $\{R_j\}$);
return
```

Discussion

- Good: linear memory
- Good: immediatly produces plan alternatives
- Bad: worst-case if ties occur
- Bad: worst-case if no ties occur is an open problem

Chain Queries

- Relations $R_1, \ldots, R_n$
- Query graph is a chain: R_1 — R_2 — $\ldots$ — R_n
- For every edge (R_i, R_{i+1}) there is an associated selectivity $f_{i,i+1} = |R_i \bowtie R_{i+1}|/|R_i \times R_{i+1}|$
- We define all other selectivities $f_{i,j} = 1$ for $|i - j| \neq 1$.

The Problem

- Generate optimal left-deep tree possibly containing cross products

Hence: every permutation corresponds to a valid join tree

Connected

- Two relations R_i and R_j are connected if they are connected in the query graph
- Two sequences s and t are connected if there exist R_i in s and R_j in t such that R_i and R_j are connected

Observation

- If a (sub)sequence s' does not contain a cross product, it uniquely corresponds to a subchain of the query graph.

In this case we speak of (sub)chains or (sub)sequences

Goal

- DP generates $n2^{n-1} - n(n+1)/2$ left-deep trees when cross products are considered
- Can we do better?

Relativized Rank

The costs (C_{out}) of a sequence s *relative* to a sequence u are defined follows:

$$\begin{aligned}
C_u(\epsilon) &:= 0 \\
C_u(R_i) &:= 0 \ \text{ if } u = \epsilon \\
C_u(R_i) &:= \Big(\prod_{R_j <_u R_i} f_{j,i}\Big) n_i \quad \text{if } u \neq \epsilon \\
C_u(s_1 s_2) &:= C_u(s_1) + T_u(s_1) * C_{us_1}(s_2)
\end{aligned}$$

with

$$\begin{aligned}
T_u(\epsilon) &:= 1 \\
T_u(s) &:= \prod_{R_i \in s} \Big(\prod_{R_j <_{us} R_i} f_{j,i} \Big) * n_i
\end{aligned}$$

Here, $R_i <_s R_j$ is true if and only if R_i appears before R_j in s.

Observations

- $C_{us}(t) = C_u(t)$ holds if there is no connection between relations in s and t
- $T_\epsilon(R_i) = |R_i|$ and $T_\epsilon(s) = |s|$
- $C_u(\epsilon) = 0$ for all u but $C_\epsilon(s) = 0$ only if s does not contain more than one relation

We abbreviate C_ϵ by C.

Remarks

- The special case that $C_\epsilon(R) = 0$ for a single relation R causes some problems in the homogeneity of definitions and proofs.
  Hence, we abandon this case from all definitions and lemmata of this section.
- Two versions for the two algorithms:
  The first version is simpler and relies on a modified cost function C' and only the second version will apply to the original cost function C.
  C' differs from C in exactly the problematic case in which it is defined as $C'_u(R_i) := |R_i|$.

Example 1

$$|R_1| = 1, |R_2| = 100, |R_3| = 10,\ f_{1,2} = f_{2,3} = 0.9$$
$$T(R_1 R_2 R_3) = \cdots = T(R_3 R_2 R_1) = 100 * 10 * 1 * 0.9 * 0.9 = 810$$

$$C(R_1 R_2 R_3) = 1 * 100 * 0.9 + 1 * 100 * 10 * 0.9 * 0.9 = 900$$
$$C(R_1 R_3 R_2) = 1 * 10 + 1 * 10 * 100 * 0.9 * 0.9 = 820$$
$$C(R_2 R_3 R_1) = 100 * 10 * 0.9 + 100 * 10 * 1 * 0.9 * 0.9 = 1710$$
$$C(R_2 R_1 R_3) = C(R_1 R_2 R_3)$$
$$C(R_3 R_1 R_2) = C(R_1 R_3 R_2)$$
$$C(R_3 R_2 R_1) = C(R_2 R_3 R_1)$$

Relativized Rank

The *rank* of a sequence *s* relative to a non-empty sequence *u* is given by

$$rank_u(s) \quad := \quad \frac{T_u(s) - 1}{C_u(s)}$$

Observation

Let $s = R_i$ be a singleton relation.
Let f_i be the product of the selectivities between relations in u and R_i.
Then

$$rank_u(R_i) = \frac{f_i|R_i| - 1}{f_i|R_i|}$$

Hence, the *rank* becomes a function of the form $f(x) = \frac{x-1}{x}$. This function is monotonously increasing in x for $x > 0$.
The argument is the $f_i|R_i|$.
This is the factor by which the next intermediate result will increase (or decrease).
Since we sum up intermediate results, this is an essential number.

Furthermore, from the monotonicity of $f(x)$ it follows that

$rank_u(R_i) \leq rank_u(R_j)$ if and only if $f_i|R_i| \leq f_j|R_j|$ where f_j is the product of all selectivities between R_j and relations in u.

Example 1 (cont'd)

The optimal sequence $R_1 R_3 R_2$ gives rise to the following ranks:

$$rank_{R_1}(R_2) = \frac{T_{R_1}(R_2)-1}{C_{R_1}(R_2)} = \frac{100*0.9-1}{100*0.9} \approx 0.9888$$
$$rank_{R_1}(R_3) = \frac{T_{R_1}(R_3)-1}{C_{R_1}(R_3)} = \frac{10*1.0-1}{10*1.0} = 0.9$$
$$rank_{R_1 R_3}(R_2) = \frac{T_{R_1 R_3}(R_2)-1}{C_{R_1 R_3}(R_2)} = \frac{100*0.9*0.9-1}{100*0.9*0.9} \approx 0.9877$$

Lemma I (API)

For sequences

$$\begin{aligned} S &= r_1 \cdots r_{k-1} r_k r_{k+1} r_{k+2} \cdots r_n \\ S' &= r_1 \cdots r_{k-1} r_{k+1} r_k r_{k+2} \cdots r_n \end{aligned}$$

the following holds:

$$C(S) \leq C(S') \Leftrightarrow rank_u(r_k) \leq rank_u(r_{k+1})$$

where $u = r_1 \cdots r_{k-1}$. Equality only holds if it holds on both sides.

Lemma II (ASI, unconnected)

Let u, x and y be three subchains where x and y are not interconnected. Then we have:

$$C(uxy) \leq C(uyx) \Leftrightarrow rank_u(x) \leq rank_u(y)$$

Equality only holds if it holds on both sides.

Contradictory Pair of Subchains

Let u, x, y be nonempty sequences. We call (x, y) a contradictory pair of subchains if and only if

$$C_u(xy) \leq C_u(yx) \quad \wedge \quad rank_u(x) > rank_{ux}(y)$$

A special case occurs when x and y are single relations. Then the above condition simplifies to

$$rank_{ux}(y) < rank_u(x) \leq rank_u(y)$$

Example 2

$|R_1| = 1, |R_2| = |R_3| = 10$, $f_{1,2} = 0.5$, $f_{2,3} = 0.2$. Consider $R_1 R_2 R_3$ and $R_1 R_3 R_2$

$$\begin{aligned}
rank_{R_1}(R_2) &= 0.8 \\
rank_{R_1 R_2}(R_3) &= 0.0 \\
rank_{R_1}(R_3) &= 0.9 \\
rank_{R_1 R_3}(R_2) &= 0.5 \\
C(R_1 R_2 R_3) &= 15 \\
C(R_1 R_3 R_2) &= 20 \\
rank_{R_1}(R_2) &> rank_{R_1 R_2}(R_3) \\
rank_{R_1}(R_3) &> rank_{R_1 R_3}(R_2) \\
C(R_1 R_2 R_3) &< C(R_1 R_3 R_2)
\end{aligned}$$

(R_2, R_3) is a contradictory pair within $R_1 R_2 R_3$.

Observation

If there is no connection between two subchains x and y, then they cannot build a contradictory pair (x, y).

Lemma 3

Let $S = usvtw$ be a sequence.
If there is no connection between relations in s and v and relations in v and t, and $rank_u(s) \geq rank_{us}(t)$, then there exists a sequence S' of not higher cost, where s immediately precedes t.

Lemma 4

Let $S = x_1 \cdots x_n$ and $S' = y_1 \cdots y_n$ be two different rank-sorted chains containing exactly the relations $R_1, \ldots, R_n$, i.e.

$$\begin{aligned} rank_{x_1 \cdots x_{i-1}}(x_i) &\leq rank_{x_1 \cdots x_i}(x_{i+1}) \text{ for all } 1 \leq i \leq n, \\ rank_{y_1 \cdots y_{i-1}}(y_i) &\leq rank_{y_1 \cdots y_i}(y_{i+1}) \text{ for all } 1 \leq i \leq n, \end{aligned}$$

then S and S' have equal costs and, furthermore

$$rank_{x_1 \cdots x_{i-1}}(x_i) = rank_{y_1 \cdots y_{i-1}}(y_i) \text{ for all } 1 < i \leq n$$

Compound Relation

If we tie together relations $r_1, \ldots, r_n$ to a new relation $r_{1,\ldots,n}$ then

- we define size of $r_{1,\ldots,n}$ as $|r_{1,\ldots,n}| = |r_1 \bowtie \ldots \bowtie r_n|$ and
- if some r_i ($1 \leq i \leq n$) does have a connection to some $r_k \notin \{r_1, \ldots, r_n\}$ then we define the selectivity factor $f_{r_{1,\ldots,n},r_k}$ between r_k and $r_{1,\ldots,n}$ as $f_{r_{1,\ldots,n},r_k} = f_{i,k}$.

Normalization

```
Normalize($p, s$)
  **while** there exist subsequences $u, v$ $(u \neq \epsilon)$ and
        compound relations $x, y$ such that $s = uxyv$
        and $C_{pu}(xy) \leq C_{pu}(yx)$
        and $rank_{pu}(x) > rank_{pux}(y)$ {
      replace $xy$ by a compound relation $(x, y)$;
  }
return $(p, s)$;
```

Maximal Contradictory Subchain

The compound relations in the result of the procedure `Normalize` are called *contradictory chains*.
A *maximal contradictory subchain* is a contradictory subchain that cannot be made longer by further tying steps.
Observation: Every chain can be decomposed into a sequence of adjacent maximal contradictory subchains.

Observation

The decomposition into adjacent maximal contradictory subchains is not unique:
Given an optimal subchain $r_1 r_2 r_3$ and a sequence u of preceding relations:
If $rank_u(r_1) > rank_{ur_1}(r_2) > rank_{ur_1 r_2}(r_3)$ one can easily show that both $(r_1, (r_2, r_3))$ and $((r_1, r_2), r_3)$ are contradictory subchains.
This ambiguity is not important since in the following we are only interested in contradictory subchains which are *optimal* and in this case the condition $C_u(xy) \leq C_u(yx)$ is certainly true and can therefore be neglected.

Lemma 5

Let $S = s_1 \dots s_m$ be an optimal chain consisting of the maximal contradictory subchains $s_1, \dots, s_m$ (as determined by the function `normalize`). Then

$$rank(s_1) \leq rank_{s_1}(s_2) \leq rank_{s_1 s_2}(s_3)$$

$$\leq \dots \leq rank_{s_1 \dots s_{m-1}}(s_m)$$

In other words, the (maximal) contradictory subchains in an optimal chain are always sorted by ascending ranks.

Lemma 6

Let x and y be two optimal sequences of relations where x and y are not interconnected. Then the sequence obtained by merging the maximal contradictory subchains in x and y (as obtained by `normalize`) according to their ascending rank is optimal.

The Cost Function C’

$$C'_u(s) = \begin{cases} C(s) + |n_R|, & \text{if } u = \epsilon \text{ and } s = Rs' \\ C_u(s), & \text{otherwise} \end{cases}$$

$$rank_u(s) := (T_u(s) - 1)/C'_u(s)$$

Neighbourhood

We call the set of relations that are directly connected to a subchain (with respect to the query graph G) the *complete neighbourhood* of that subchain. A *neighbourhood* is a subset of the complete neighbourhood. The *complement* of a neighbourhood u of a subchain s is defined as $v - u$, where v denotes the complete neighbourhood of s.
Obviously: the neighborhood that precedes a sequence influences its rank.
Denote a pair consisting of a connected sequence s and a neighbourhood u by $[s]_u$.

Contradictory Subchain, Extent

A *contradictory subchain* $[s]_u$ is inductively defined:

1. For a single relation s, $[s]_u$ is a contradictory subchain.
2. There is a decomposition $s = vw$ such that (v, w) is a contradictory pair with respect to the preceding subsequence u and both $[v]_u$ and $[w]_{uv}$ are themselves contradictory subchains.

The *extent* of a contradictory chain $[s]_u$ is defined to be the pair consisting of the neighbourhood u and the set of relations occurring in s.
Since contradictory subchains are connected, the set of occurring relations has always the form $\{R_i, R_{i+1}, \ldots, R_{i+l}\}$ for some $1 \leq i \leq n,\ 0 \leq l \leq n - i$.
An *optimal contradictory subchain* to a given extent is a contradictory subchain with lowest cost among all contradictory subchains of the same extent.

Number of Extents

The number of different extents of contradictory subchains for a chain query of n relations is

$$2n^2 - 2n + 1$$

Recursively Decomposable Subchain

A *recursively decomposable subchain* $[s]_u$ is inductively defined as follows.

1. If s is a single relation then $[s]_u$ is recursively decomposable.
2. There is a decomposition $s = vw$ such that v is connected to w and both $[v]_u$ and $[w]_{uv}$ are recursively decomposable subchains.

Remark The extent of a recursively decomposable chain is defined in the same way as for contradictory chains. Note that every contradictory subchain is recursively decomposable. Consequently, the set of all contradictory subchains for a certain extent is a subset of all recursively decomposable subchains of the same extent.

Example

Consider the sequence of relations

$$s = R_2 R_4 R_3 R_6 R_5 R_1 .$$

Using parenthesis to indicate the recursive decompositions we have the following two possibilities

$$(((R_2(R_4 R_3))(R_6 R_5))R_1)$$

$$((R_2((R_4 R_3)(R_6 R_5)))R_1)$$

The extent of the recursively decomposable subchain

$$R_4 R_3 R_6 R_5$$

of s is $(\{R_2\}, \{R_3, R_4, R_5, R_6\})$

Number of Recursively Decomposable Chains

The number of different recursively decomposable chains involving the relations $R_1, \ldots, R_n$ is r_n, where r_n denotes the n-th Schröder number: Hence, the total number of recursively decomposable chains is

$$r_n + 2(n-1)r_{n-1} + 4\sum_{i=1}^{n-2}\binom{n-2}{i} r_i$$

It can be shown that

$$r_n \approx \frac{C(2+\sqrt{8})^n}{n^{3/2}}$$

where $C = 1/2\sqrt{\frac{2\sqrt{2}-4}{\pi}}$. Using Stirling's formula for $n!$ it is easy to show that $\lim_{n\to\infty} \frac{r_n}{n!} = 0$. Thus, the probability of a random permutation to be recursively decomposable strives to zero for large n.

Optimality Principle

- An *optimal recursively decomposable subchain* to a given extent is a recursively decomposable subchain with lowest cost among all recursively decomposable subchains of the same extent.
- Bellman's optimality principle holds: every optimal recursively decomposable subchain can be decomposed into smaller optimal recursively decomposable subchains.

Hence: the optimal recursively decomposable subchains can be computed using dynamic programming

Example 5

In order to compute an optimal recursively decomposable subchain for the extent

$$(\{R_2, R_7\}, \{R_3, R_4, R_5, R_6\})$$

the algorithm makes use of optimal recursively decomposable subchains for the extents

$$\begin{array}{rl}
(\{R_2\}, \{R_3\}) & (\{R_7, R_3\}, \{R_4, R_5, R_6\}) \\
(\{R_2\}, \{R_3, R_4\}) & (\{R_7, R_4\}, \{R_5, R_6\}) \\
(\{R_2\}, \{R_3, R_4, R_5\}) & (\{R_5, R_7\}, \{R_6\}) \\
(\{R_7\}, \{R_4, R_5, R_6\}) & (\{R_2, R_4\}, \{R_3\}) \\
(\{R_7\}, \{R_5, R_6\}) & (\{R_2, R_5\}, \{R_3, R_4\}) \\
(\{R_7\}, \{R_6\}) & (\{R_2, R_6\}, \{R_3, R_4, R_5\})
\end{array}$$

which have been computed in earlier steps.

The splitting of extents induces a partial order on the set of extents.

Partial Order on Extents, M

Let E be the set of all possible extents.
We define the following partial order $\mathcal{P} = (E, \prec)$ on E:
For all extents $e_1, e_2 \in E$:
$e_1 \prec e_2$ if and only if e_1 can be obtained by splitting the extent e_2.
The set of maximal extents M then corresponds to a set of incomparable elements in $\mathcal{P}$ such that for all extents e enumerated so far, there is an extent $e' \in M$ with $e \prec e'$.

Example

$$(\{R_7\}, \{R_5, R_6\}) \prec (\{R_2, R_7\}, \{R_3, R_4, R_5, R_6\})$$

Algorithm Chain-I’

1. Use dynamic programming to determine all optimal contradictory subchains.
   This step can be made faster by keeping track of the set *M* of all maximal extents (with respect to the partial order induced by splitting extents).
2. Determine all optimal recursively decomposable subchains for all extents included in some maximal extent in *M*.
3. Compare the results from steps 1 and 2 and retain only matching subchains.
4. Sort the contradictory subchains according to their ranks.
5. Eliminate contradictory subchains that cannot be part of a solution.
6. Use backtracking to enumerate all sequences of rank-ordered optimal contradictory subchains and keep track of the sequence with lowest cost.

Algorithm Chain-II'

1. Use dynamic programming to compute an optimal recursive decomposable chain for the whole set of relations $\{R_1, \ldots, R_n\}$.
2. Normalize the resulting chain.
3. Reorder the contradictory subchains according to their ranks.
4. De-normalize the sequence.

Discussion

Algorithm	Runtime	Optimality
Chain-I	?	yes
Chain-II	$O(n^4)$	?

Questionsmarks: exercise

Transformation-Based Approaches

Main idea:

- Use equivalences directly (associativity, commutativity)

Rule Set

$R_1 \bowtie R_2$	$\rightsquigarrow$	$R_2 \bowtie R_1$	Commutativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$R_1 \bowtie (R_2 \bowtie R_3)$	Right Associativity
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$(R_1 \bowtie R_2) \bowtie R_3$	Left Associativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$(R_1 \bowtie R_3) \bowtie R_2$	Left Join Exchange
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$R_2 \bowtie (R_1 \bowtie R_3)$	Right Join Exchange

Two more rules are often used to transform left-deep trees:

- *swap* exchanges two arbitrary relations in a left-deep tree
- *3Cycle* performs a cyclic rotation of three arbitrary relations in a left-deep tree.

To try another join method, another rule called *join method exchange* is introduced.

Rule Set RS-0

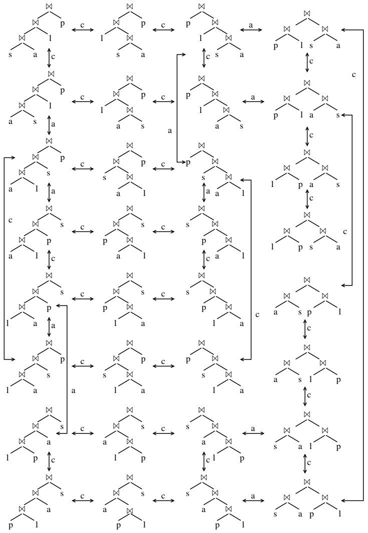
- commutativity
- left-associativity
- right-associativity

```
ExhaustiveTransformation($\{R_1, \ldots, R_n\}$)
Input: a set of relations
Output: an optimal join tree
Let $T$ be an arbitrary join tree for all relations
Done = $\emptyset$; // contains all trees processed
ToDo = $\{T\}$; // contains all trees to be processed
while (!empty(ToDo)) {
  Let $T$ be an arbitrary tree in ToDo
  ToDo $\setminus=$ $T$;
  Done $+=$ $T$;
  Trees = ApplyTransformations($T$);
  for all $T \in$ Trees do { if ($T \notin$ ToDo $\cup$ Done) { ToDo $+=$ $T$; } }
}
return cheapest tree found in Done;
```

```
ApplyTransformations($T$)
Input: join tree
Output: all trees derivable by associativity and commutativity
Trees = $\emptyset$;
Subtrees = all subtrees of $T$ rooted at inner nodes
for all $S$ $\in$ Subtrees do {
  if ($S$ is of the form $S_1 \bowtie S_2$) { Trees $+=$ $S_2 \bowtie S_1$; }
  if ($S$ is of the form $(S_1 \bowtie S_2) \bowtie S_3$) {
    Trees $+=$ $S_1 \bowtie (S_2 \bowtie S_3)$;
  }
  if ($S$ is of the form $S_1 \bowtie (S_2 \bowtie S_3)$) {
    Trees $+=$ $(S_1 \bowtie S_2) \bowtie S_3$;
  }
}
return Trees;
```

Remarks

- If no cross products are to be considered, extend **if** conditions for associativity rules.
- Problem 1: explores the whole search space
- Problem 2: generates join trees more than once
- Problem 3: sharing of subtrees is non-trivial



Introducing the Memo Structure

- For any subset of relations, dynamic programming remembers the best join tree.
- This does not quite suffice for the transformation-based approach.
- Instead, we have to keep all join trees generated so far including those differing in the order of the arguments of a join operator.
- However, subtrees can be shared.
- This is done by keeping pointers into the data structure (see below).

Memo Structure Example

$\{R_1, R_2, R_3\}$	$\{R_1, R_2\} \bowtie R_3, R_3 \bowtie \{R_1, R_2\},$ $\{R_1, R_3\} \bowtie R_2, R_2 \bowtie \{R_1, R_3\},$ $\{R_2, R_3\} \bowtie R_1, R_1 \bowtie \{R_2, R_3\}$
$\{R_2, R_3\}$	$\{R_2\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_2\}$
$\{R_1, R_3\}$	$\{R_1\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_1\}$
$\{R_1, R_2\}$	$\{R_1\} \bowtie \{R_2\}, \{R_2\} \bowtie \{R_1\}$
$\{R_3\}$	R_3
$\{R_2\}$	R_2
$\{R_1\}$	R_1

Remarks

- in Memo Structure: arguments are pointers to classes
- Algorithm: `ExploreClass` expands a class
- Algorithm: `ApplyTransformation2` expands a member of a class

```
ExhaustiveTransformation2(Query Graph $G$)
**Input:** a query specification for relations $\{R_1, \ldots, R_n\}$.
**Output:** an optimal join tree
  initialize MEMO structure
  ExploreClass($\{R_1, \ldots, R_n\}$)
  **return** best of class $\{R_1, \ldots, R_n\}$
```

```
ExploreClass(C)
Input:   a class $\mathcal{C} \subseteq \{R_1, \ldots, R_n\}$
Output:  none, but has side-effect on MEMO-structure
  while (not all join trees in C have been explored) {
    choose an unexplored join tree $T$ in $\mathcal{C}$
    ApplyTransformation2($T$)
    mark $T$ as explored
  }
  return
```

```
ApplyTransformations2($T$)
Input:  a join tree of a class $\mathcal{C}$
Output:  none, but has side-effect on MEMO-structure
  ExploreClass(left-child($T$));
  ExploreClass(right-child($T$));
  foreach transformation $\mathcal{T}$ and class member of child classes {
    foreach $T'$ resulting from applying $\mathcal{T}$ to $T$ {
      if $T'$ not in MEMO structure {
        add $T'$ to class $\mathcal{C}$ of MEMO structure
      }
    }
  }
  return
```

Remarks

- Applying `ExhaustiveTransformation2` with a rule set consisting of Commutativity and Left and Right Associativity generates $4^n - 3^{n+1} + 2^{n+2} - n - 2$ duplicates
- Contrast this with the number of join trees contained in a completely filled MEMO structure: $3^n - 2^{n+1} + n + 1$
- Solve the problem of duplicate generation by disabling applied rules.
- Note that this does not solve the problem that transformation-based approaches explore the whole search space.

Rule Set RS-1

T_1: Commutativity $C_1 \bowtie_0 C_2 \leadsto C_2 \bowtie_1 C_1$
Disable all transformations T_1, T_2, and T_3 for $\bowtie_1$.

T_2: Right Associativity $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \leadsto C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$
Disable transformations T_2 and T_3 for $\bowtie_2$ and enable all rules for $\bowtie_3$.

T_3: Left associativity $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \leadsto (C_1 \bowtie_2 C_2) \bowtie_3 C_3$
Disable transformations T_2 and T_3 for $\bowtie_3$ and enable all rules for $\bowtie_2$.

Example for chain query $R_1 - R_2 - R_3 - R_4$:

Class	Initialization	Transformation	Step
$\{R_1, R_2, R_3, R_4\}$	$\{R_1, R_2\} \bowtie_{111} \{R_3, R_4\}$	$\{R_3, R_4\} \bowtie_{000} \{R_1, R_2\}$	3
		$R_1 \bowtie_{100} \{R_2, R_3, R_4\}$	4
		$\{R_1, R_2, R_3\} \bowtie_{100} R_4$	5
		$\{R_2, R_3, R_4\} \bowtie_{000} R_1$	8
		$R_4 \bowtie_{000} \{R_1, R_2, R_3\}$	10
$\{R_2, R_3, R_4\}$		$R_2 \bowtie_{111} \{R_3, R_4\}$	4
		$\{R_3, R_4\} \bowtie_{000} R_2$	6
		$\{R_2, R_3\} \bowtie_{100} R_4$	6
		$R_4 \bowtie_{000} \{R_2, R_3\}$	7
$\{R_1, R_3, R_4\}$			
$\{R_1, R_2, R_4\}$			
$\{R_1, R_2, R_3\}$		$\{R_1, R_2\} \bowtie_{111} R_3$	5
		$R_3 \bowtie_{000} \{R_1, R_2\}$	9
		$R_1 \bowtie_{100} \{R_2, R_3\}$	9
		$\{R_2, R_3\} \bowtie_{000} R_1$	9
$\{R_3, R_4\}$	$R_3 \bowtie_{111} R_4$	$R_4 \bowtie_{000} R_3$	2
$\{R_2, R_4\}$			
$\{R_2, R_3\}$			
$\{R_1, R_4\}$			
$\{R_1, R_3\}$			
$\{R_1, R_2\}$	$R_1 \bowtie_{111} R_2$	$R_2 \bowtie_{000} R_1$	1

Rule Set RS-2

Bushy Trees: Rule set for clique queries and if cross products are allowed:

T_1: Commutativity $C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_1$.

T_2: Right Associativity $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$
Disable transformations T_2, T_3, and T_4 for $\bowtie_2$.

T_3: Left Associativity $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3$
Disable transformations T_2, T_3 and T_4 for $\bowtie_3$.

T_4: Exchange $(C_1 \bowtie_0 C_2) \bowtie_1 (C_3 \bowtie_2 C_4) \rightsquigarrow (C_1 \bowtie_3 C_3) \bowtie_4 (C_2 \bowtie_5 C_4)$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_4$.

If we initialize the MEMO structure with left-deep trees, we can strip down the above rule set to Commutativity and Right Associativity.
Reason: from a left-deep join tree we can generate all bushy trees with only these two rules

Rule Set RS-3

Left-deep trees:

T_1 Commutativity $R_1 \bowtie_0 R_2 \rightsquigarrow R_2 \bowtie_1 R_1$
Here, the R_i are restricted to classes with exactly one relation. T_1 is disabled for $\bowtie_1$.

T_2 Right Join Exchange $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow (C_1 \bowtie_2 C_3) \bowtie_3 C_2$
Disable T_2 for $\bowtie_3$.

Generating Random Join Trees

- Randomized optimization procedures
- Basis for Simulated Annealing, Iterative Improvement
- With Cross Products first, since they are easier

Ranking/Unranking

Let S be a set with n elements.

- a bijective mapping $f : S \to [0, n[$ is called *ranking*
- a bijective mapping $f : [0, n[\to S$ is called *unranking*

Given an unranking function, we can generate random elements in S by generating a random number in $[0, n[$ and unranking this number.
Challenge: making unranking fast.

Random Permutations

Every permutation corresponds to a left-deep join tree possibly with cross products.
Standard algorithm to generate random permutations is the starting point for the algorithm:

```
for ($k = n - 1$; $k \geq 0$; $--k$)
    swap($\pi[k], \pi[\text{random}(k)]$);
```

Array π initialized with elements $[0, n[$.
random(k) generates a random number in $[0, k]$.

Random Permutations

- Assume the random elements produced by the algorithm are $r_{n-1}, \dots, r_0$ where $0 \le r_i \le i$.
- Thus, there are exactly $n(n-1)(n-2)\dots 1 = n!$ such sequences and there is a one to one correspondance between these sequences and the set of all permutations.
- Unrank $r \in [0, n![$ by turning it into a unique sequence of values $r_{n-1}, \dots, r_0$.
  Note that after executing the swap with r_{n-1} every value in $[0, n[$ is possible at position $\pi[n-1]$.
  Further, $\pi[n-1]$ is never touched again.
- Hence, we can unrank r as follows. We first set $r_{n-1} = r \mod n$ and perform the swap. Then, we define $r' = \lfloor r/n \rfloor$ and iteratively unrank r' to construct a permutation of $n-1$ elements.

```
Unrank(n, r)
Input:  the number $n$ of elements to be permuted
        the rank $r$ of the permutation to be constructed
Output:  a permutation $\pi$
for ($i = 0$; $i < n$; $++i$) $\pi[i] = i$;
Unrank-Sub($n, r, \pi$);
return $\pi$;

Unrank-Sub($n, r, \pi$)
for ($i = n$; $i > 0$; $--i$)
   swap($\pi[i-1]$, $\pi[r \mod i]$);
   $r = \lfloor r/i \rfloor$;
```

Generating Random Bushy Trees with Cross Products

Steps of the algorithm:

1. Generate a random number b in $[0, C(n-1)[$.
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 to the binary tree obtained in Step 2.

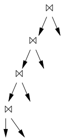
The only step that we have still to discuss is Step 2.

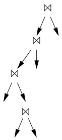
Tree Encoding

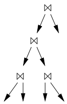
- Preordertraversal:
  - Inner node: '('
  - Leaf Node: ')'

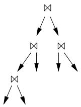
  Skip last leaf node.
- Replace '(' by 1 and ')' by 0
- Just take positions of 1s.
- The ranks are in $[0, 14[$

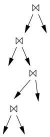
Example: all trees with four inner nodes:

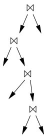

(((( ))))
11110000
1, 2, 3, 4
0

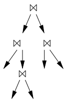

((( )( )))
11101000
1, 2, 3, 5
1

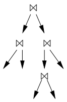

((( )) ( ))
11100100
1, 2, 3, 6
2

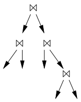

((( ))) ( )
11100010
1, 2, 3, 7
3

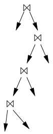

(( ) (( )))
11011000
1, 2, 4, 5
4

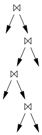

(( ) ( ) ( ))
11010100
1, 2, 4, 6
5

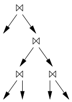

(( ) ( )) ( )
11010010
1, 2, 4, 7
6

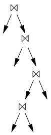

(( )) (( ))
11001100
1, 2, 5, 6
7

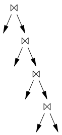

(( )) ( ) ( )
11001010
1, 2, 5, 7
8


( ) ((( )))
10111000
1, 3, 4, 5
9


( ) (( ) ( ))
10110100
1, 3, 4, 6
10

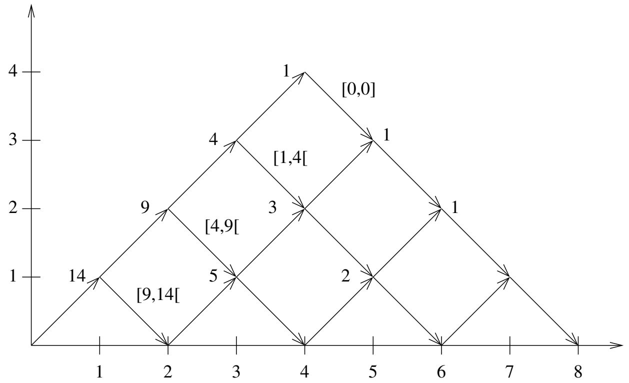

( ) (( )) ( )
10110010
1, 3, 4, 7
11


( ) ( ) (( ))
10101100
1, 3, 5, 6
12


( ) ( ) ( ) ( )
10101010
1, 3, 5, 7
13

Unranking Binary Trees

Establish bijection between Dyck words and paths in a grid:



Every path from $(0, 0)$ to $(2n, 0)$ uniquely corresponds to a Dyck word.

Counting Paths

The number of different paths from $(0,0)$ to (i,j) can be computed by

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

These numbers are the *Ballot numbers*.
The number of paths from (i,j) to $(2n,0)$ can thus be computed as:

$$q(i,j) = p(2n-i,j)$$

Note the special case $q(0,0) = p(2n,0) = C(n)$.

Unranking Outline

- We open a parenthesis (go from (i, j) to $(i + 1, j + 1)$) as long as the number of paths from that point does no longer exceed our rank r.
- If it does, we close a parenthesis (go from (i, j) to $(i - 1, j + 1)$).
- Assume, that we went upwards to (i, j) and then had to go down to $(i - 1, j + 1)$.
  We subtract the number of paths from $(i + 1, j + 1)$ from our rank r and proceed iteratively from $(i - 1, j + 1)$ by going up as long as possible and going down again.
- Remembering the number of parenthesis opened and closed along our way results in the required encoding.

```
UnrankTree($n, r$)
**Input:** a number of inner nodes $n$ and a rank $r \in [0, C(n-1)]$
**Output:** encoding of the inner nodes of a tree
lNoParOpen = lNoParClose = 0;
i = 1; // current encoding
j = 0; // current position in encoding array
while ($j < n$) {
  k = q(lNoParOpen + lNoParClose + 1,
        lNoParOpen - lNoParClose + 1);
  if ($k \leq r$) {
    $r$ -= $k$;
    ++lNoParClose;
  } else {
    aTreeEncoding[j++] = i;
    ++lNoParOpen;
  }
  ++i;
}
```

Generating Random Trees Without Cross Products

Tree queries only!

- query graph: $G = (V, E)$, $|V| = n$, G must be a tree.
- level: root has level 0, children thereof 1, etc.
- $\mathcal{T}_G$: join trees for G

Partitioning $\mathcal{T}_G$

$\mathcal{T}_G^{v(k)} \subseteq \mathcal{T}_G$: subset of join trees where the leaf node (i.e. relation) v occurs at level k.
Observations:

- $n = 1$: $|\mathcal{T}_G| = |\mathcal{T}_G^{v(0)}| = 1$
- $n > 1$: $|\mathcal{T}_G^{v(0)}| = 0$ (top is a join and no relation)
- The maximum level that can occur in any join tree is $n - 1$. Hence: $|\mathcal{T}_G^{v(k)}| = 0$ if $k \geq n$.
- $\mathcal{T}_G = \cup_{k=0}^{n} \mathcal{T}_G^{v(k)}$
- $\mathcal{T}_G^{v(i)} \cap \mathcal{T}_G^{v(j)} = \emptyset$ for $i \neq j$
- Thus: $|\mathcal{T}_G| = \sum_{k=0}^{n} |\mathcal{T}_G^{v(k)}|$

The Specification

- The algorithm will generate an unordered tree with n leaf nodes.
- If we wish to have a random ordered tree, we have to pick one of the 2^{n-1} possibilities to order the $(n-1)$ joins within the tree.

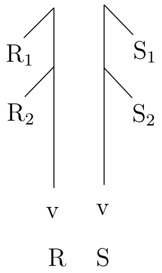
The Procedure

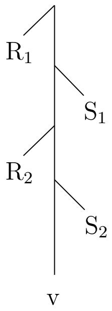
1. List merges (notation, specification, counting, unranking)
2. Join tree construction: leaf-insertion and tree-merging
3. Standard Decomposition Graph (SDG): describes all valid join trees
4. Counting
5. Unranking algorithm

List Merge

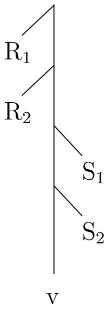
- Lists: Prolog-Notation: $\langle a|t\rangle$
- Property P on elements
- A list l' is the *projection* of a list L on P, if L' contains all elements of L satisfying the property P.
  Thereby, the order is retained.
- A list L is a *merge* of two disjoint lists L_1 and L_2, if L contains all elements from L_1 and L_2 and both are projections of L.

Example

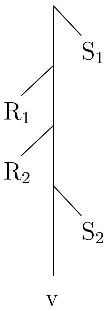




(R, S, [1, 1, 0])



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



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

List Merge: Specification

A merge of a list L_1 with a list L_2 whose respective lengths are l_1 and l_2 can be described by an array $\alpha = [\alpha_0, \ldots, \alpha_{l_2}]$ of non-negative integers whose sum is equal to l_1, i.e. $\sum_{i=0}^{l_2} \alpha_i = |l_1|$.

- We obtain the merged list L by first taking α_0 elements from L_1.
- Then, an element from L_2 follows. Then follow α_1 elements from L_1 and the next element of L_2 and so on.
- Finally follow the last α_{l_2} elements of L_1.

List Merge: Counting (1)

Non-negative integer decomposition:

- What is the number of decompositions of a non-negative integer n into k non-negative integers α_i with $\sum_{i=1}^{k} \alpha_k = n$.

Answer: $\binom{n+k-1}{k-1}$

List Merge: Counting (2)

Since we have to decompose l_1 into $l_2 + 1$ non-negative integers, the number of possible merges is $M(l_1, l_2) = \binom{l_1+l_2}{l_2}$.
The observation $M(l_1, l_2) = M(l_1 - 1, l_2) + M(l_1, l_2 - 1)$ allows us to construct an array of size $n * n$ in $O(n^2)$ that materializes the values for M.
This array will allow us to rank list merges in $O(l_1 + l_2)$.

List Merge: Unranking: General Idea

The idea for establishing a bijection between $[1, M(l_1, l_2)]$ and the possible αs is a general one and used for all subsequent algorithms of this section.
Assume we want to rank the elements of some set S and $S = \cup_{i=0}^{n} S_i$ is partitioned into disjoint S_i.

1. If we want to rank $x \in S_k$, we first find the *local rank* of $x \in S_k$.
2. The rank of x is then $\sum_{i=0}^{k-1} |S_i| + \texttt{local-rank}(x, S_k)$.
3. To unrank some number $r \in [1, N]$, we first find k such that $k = \min_j r \leq \sum_{i=0}^{j} |S_i|$.
4. We proceed by unranking with the new local rank $r' = r - \sum_{i=0}^{k-1} |S_i|$ within S_k.

List Merge: Unranking

We partition the set of all possible merges into subsets.

- Each subset is determined by α_0.
  For example, the set of possible merges of two lists L_1 and L_2 with length $l_1 = l_2 = 4$ is partitioned into subsets with $\alpha_0 = j$ for $0 \leq j \leq 4$.
- In each partition, we have $M(l_1 - j, l_2 - 1)$ elements.
- To unrank a number $r \in [1, M(l_1, l_2)]$ we first determine the partition by computing $k = \min_j r \leq \sum_{i=0}^{j} M(j, l_2 - 1)$.
  Then, $\alpha_0 = l_1 - k$.
- With the new rank $r' = r - \sum_{i=0}^{k} M(j, l_2 - 1)$, we start iterating all over.

Example

k	α_0	$(k, l_2 - 1)$	$M(k, l_2 - 1)$	rank intervals
0	4	$(0, 3)$	1	$[1, 1]$
1	3	$(1, 3)$	4	$[2, 5]$
2	2	$(2, 3)$	10	$[6, 15]$
3	1	$(3, 3)$	20	$[16, 35]$
4	0	$(4, 3)$	35	$[36, 70]$

```
UnrankDecomposition($r, l_1, l_2$)
Input:  a rank $r$, two list sizes $l_1$ and $l_2$
Output:  a merge specification $\alpha$.
for ($i = 0$; $i \le l_2$; $++i$) {
  alpha[$i$] = 0;
}
$i = k = 0$;
```

```
while ($l_1 > 0$ && $l_2 > 0$) {
   $m = M(k, l_2 - 1)$;
   if ($r \leq m$) {
      alpha[$i++$] = $l_1 - k$;
      $l_1 = k$;
      $k = 0$;
      $--l_2$;
   } else {
      $r -= m$;
      $++k$;
   }
}
alpha[$i$] = $l_1$;
return alpha;
```

Anchored List Representation of Join Trees

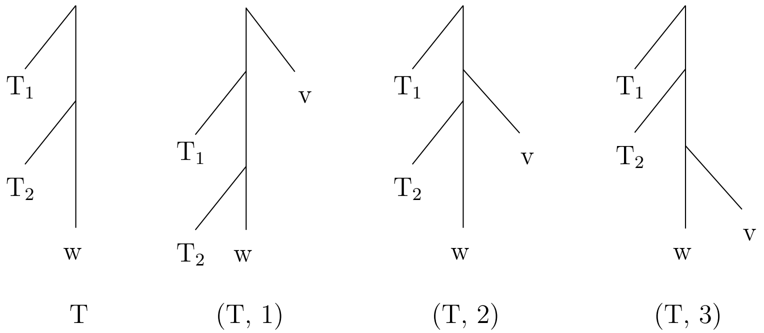
Definition Let T be a join tree and v be a leaf of T. The *anchored list representation* L of T is constructed as follows:

- If T consists of the single leaf node v, then $L = \langle\rangle$.
- If $T = (T_l \bowtie T_2)$ and without loss of generality v occurs in T_2, then $L = \langle T_1 | L_2 \rangle$ where L_2 is the anchored list representation of T_2.

We then write $T = (L, v)$.

Observation If $T = (L, v) \in \mathcal{T}_G$ then $T \in \mathcal{T}_G^{v(k)} \prec\succ |L| = k$

Leaf-Insertion: Example



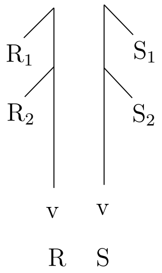
Leaf-Insertion

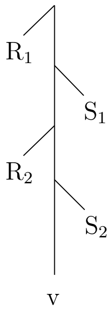
Definition Let $G = (V, E)$ be a query graph, T a join tree of G. $v \in V$ be such that $G' = G|_{V\setminus\{v\}}$ is connected, $(v, w) \in E$, $1 \leq k < n$, and

$$\begin{aligned} T &= (\langle T_1, \ldots, T_{k-1}, v, T_{k+1}, \ldots, T_n\rangle, w) && (17) \\ T' &= (\langle T_1, \ldots, T_{k-1}, T_{k+1}, \ldots, T_n\rangle, w). && (18) \end{aligned}$$

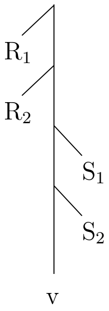
Then we call (T', k) an *insertion pair* on v and say that T is *decomposed into* (or *constructed from*) the pair (T', k) on v.
Observation: Leaf-insertion defines a bijective mapping between $\mathcal{T}_G^{v(k)}$ and insertion pairs (T', k) on v, where T' is an element of the disjoint union $\cup_{i=k-1}^{n-2} \mathcal{T}_{G'}^{w(i)}$.

Tree-Merging: Example

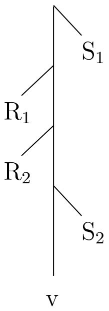




(R, S, [1, 1, 0])



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



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

Tree-Merging

Two trees $R = (L_R, w)$ and $S = (L_S, w)$ on a common leaf w are merged by merging their anchored list representations.
Definition. Let $G = (V, E)$ be a query graph, $w \in V$, $T = (L, w)$ a join tree of G, $V_1, V_2 \subseteq V$ such that $G_1 = G|_{V_1}$ and $G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{w\}$.
For $i = 1, 2$:

- Define the property P_i to be "every leaf of the subtree is in V_i",
- Let L_i be the projection of L on P_i.
- $T_i = (L_i, w)$.

Let α be the integer decomposition such that L is the result of merging L_1 and L_2 on α. Then, we call (T_1, T_2, α) a *merge triplet*. We say that T is *decomposed into* (*constructed from*) (T_1, T_2, α) on V_1 and V_2.

Observation

Tree-Merging defines a bijective mapping between $\mathcal{T}_G^{w(k)}$ and merge triplets (T_1, T_2, α), where $T_1 \in \mathcal{T}_{G_1}^{w(i)}$, $T_2 \in \mathcal{T}_{G_2}^{w(k-i)}$, and α specifies a merge of two lists of sizes i and $k-i$. Further, the number of these merges (i.e. the number of possibilities for α) is $\binom{i+(k-i)}{k-i} = \binom{k}{i}$.

Standard Decomposition Graph (SDG)

A *standard decomposition graph* of a query graph describes the possible constructions of join trees.
It is not unique (for $n > 1$) but anyone can be used to construct all possible unordered join trees.
For each of our two operations it has one kind of inner nodes:

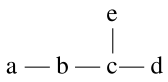
- A unary node labeled $+_v$ stands for leaf-insertion of v.
- A binary node labeled $*_w$ stands for tree-merging its subtrees whose only common leaf is w.

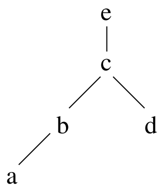
Constructing a Standard Decomposition Graph

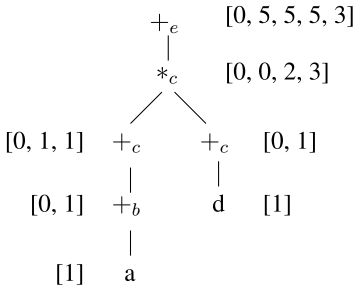
The standard decomposition graph of a query graph $G = (V, E)$ is constructed in three steps:

1. pick an arbitrary node $r \in V$ as its root node
2. transform G into a tree G' by directing all edges away from r;
3. call `QG2SDG(`G'`,`r`)`

```
QG2SDG($G', r$)
Input:  a query tree $G' = (V, E)$ and its root $r$
Output:  a standard query decomposition tree of $G'$
Let $\{w_1, \ldots, w_n\}$ be the children of $v$;
switch ($n$) {
  case 0:  label $v$ with "$v$";
  case 1:
      label $v$ as "$+_v$";
      QG2SDG($G', w_1$);
  otherwise:
      label $v$ as "$*_v$";
      create new nodes $l$, $r$ with label $+_v$;
      $E \setminus = \{(v, w_i) | 1 \leq i \leq n\}$;
      $E \cup = \{(v, l), (v, r), (l, w_1)\} \cup \{(r, w_i) | 2 \leq i \leq n\}$;
      QG2SDG($G', l$);
      QG2SDG($G', r$);
}
return $G'$;
```


e
a
b
c
d


e
c
b
d
a


$+_e$
[0, 5, 5, 5, 3]
$*_c$
[0, 0, 2, 3]
[0, 1, 1]
$+_c$
$+_c$
[0, 1]
[0, 1]
$+_b$
d
[1]
[1]
a

Counting (1)

For efficient access to the number of join trees in some partition $\mathcal{T}_G^{v(k)}$ in the unranking algorithm, we materialize these numbers. This is done in the `count` array.
The semantics of a `count` array $[c_0, c_1, \ldots, c_n]$ of a node u with label $\circ_v$ ($\circ \in \{+, *\}$) of the SDG is that

- u can construct c_i different trees in which leaf v is at level i.

Then, the total number of trees for a query can be computed by summing up all the c_i in the `count` array of the root node of the decomposition tree.

Counting (2)

To compute the `count` and an additional `summand` adornment of a node labeled $+_v$, we use the following lemma:
Lemma. Let $G = (V, E)$ be a query graph with n nodes, $v \in V$ such that $G' = G|_{V \setminus v}$ is connected, $(v, w) \in E$, and $1 \leq k < n$. Then

$$|\mathcal{T}_G^{v(k)}| = \sum_{i \geq k-1} |\mathcal{T}_{G'}^{w(i)}|$$

Counting (3)

The sets $\mathcal{T}_{G'}^{w(i)}$ used in the summands of the former Lemma directly correspond to subsets $\mathcal{T}_{G}^{v(k),i}$ $(k-1 \leq i \leq n-2)$ defined such that $T \in \mathcal{T}_{G}^{v(k),i}$ if

1. $T \in \mathcal{T}_{G}^{v(k)}$,
2. the insertion pair on v of T is (T', k), and
3. $T' \in \mathcal{T}_{G'}^{w(i)}$.

Further, $|\mathcal{T}_{G}^{v(k),i}| = |\mathcal{T}_{G'}^{w(i)}|$. For efficiency, we materialize the summands in an array of arrays `summands`.

Counting (4)

To compute the `count` and `summand` adornment of a node labeled $*_v$, we use the following lemma.
Lemma. Let $G = (V, E)$ be a query graph, $w \in V$, $T = (L, w)$ a join tree of G, $V_1, V_2 \subseteq V$ such that $G_1 = G|_{V_1}$ and $G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$. Then

$$|\mathcal{T}_G^{v(k)}| = \sum_i \binom{k}{i} \, |\mathcal{T}_{G_1}^{v(i)}| \, |\mathcal{T}_{G_2}^{v(k-i)}|$$

Counting (5)

The sets $\mathcal{T}_{G'}^{w(i)}$ used in the summands of the previous Lemma directly correspond to subsets $\mathcal{T}_{G}^{v(k),i}$ ($0 \leq i \leq k$) defined such that $T \in \mathcal{T}_{G}^{v(k),i}$ if

1. $T \in \mathcal{T}_{G}^{v(k)}$,
2. the merge triplet on V_1 and V_2 of T is (T_1, T_2, α), and
3. $T_1 \in \mathcal{T}_{G_1}^{v(i)}$.

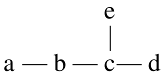
Further, $|\mathcal{T}_{G}^{v(k),i}| = \binom{k}{i} \, |\mathcal{T}_{G_1}^{v(i)}| \, |\mathcal{T}_{G_2}^{v(k-i)}|$.

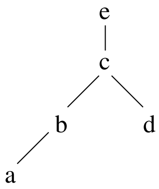
Counting (6)

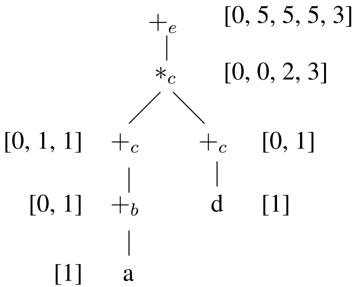
Observation: Assume a node v whose `count` array is $[c_1, \ldots, c_m]$ and whose `summands` is $s = [s^0, \ldots, s^n]$ with $s_i = [s^i_0, \ldots, s^i_m]$, then

$$c_i = \sum_{j=0}^{m} s^i_j$$

holds.
The following algorithm has worst-case complexity $O(n^3)$.
Looking at the `count` array of the root node of the following SDG, we see that the total number of join trees for our example query graph is 18.


e
a — b — c — d


e
c
b
d
a


$+_e$ [0, 5, 5, 5, 3]
$*_c$ [0, 0, 2, 3]
[0, 1, 1] $+_c$
$+_c$ [0, 1]
[0, 1] $+_b$
d [1]
[1] a

```
Adorn($v$)
**Input:** a node $v$ of the SDG
**Output:** $v$ and nodes below are adorned by count and summands
Let $\{w_1, \ldots, w_n\}$ be the children of $v$;
**switch** ($n$) {
  **case 0:** count($v$) := $[1]$; // no summands for $v$
  **case 1:**
      Adorn($w_1$);
      assume count($w_1$) = $[c^1_0, \ldots, c^1_{m_1}]$;
      count($v$) := $[0, c_1, \ldots, c_{m_1+1}]$ where $c_k = \sum_{i=k-1}^{m_1} c^1_i$;
      summands($v$) = $[s^0, \ldots, s^{m_1+1}]$ where $s^k = [s^k_0, \ldots, s^k_{m_1+1}]$ and
      $s^k_i = \begin{cases} c^1_i & \text{if } 0 < k \text{ and } k-1 \le i \\ 0 & \text{else} \end{cases}$
```

case 2:

```
    Adorn($w_1$);
    Adorn($w_2$);
    assume count($w_1$) = $[c^1_0, \ldots, c^1_{m_1}]$;
    assume count($w_2$) = $[c^2_0, \ldots, c^2_{m_2}]$;
    count($v$) = $[c_0, \ldots, c_{m_1+m_2}]$ where
        $c_k = \sum_{i=0}^{m_1} \binom{k}{i} c^1_i c^2_{k-i}$;  // $c^2_i = 0$ for $i \notin \{0, \ldots, m_2\}$
    summands($v$) = $[s^0, \ldots, s^{m_1+m_2}]$ where $s^k = [s^k_0, \ldots, s^k_{m_1}]$ and
    $s^k_i = \begin{cases} \binom{k}{i} c^1_i c^2_{k-i} & \text{if } 0 \leq k - i \leq m_2 \\ 0 & \text{else} \end{cases}$
}
```

Unranking: top-level procedure

The algorithm `UnrankLocalTreeNoCross` called by `UnrankTreeNoCross` adorns the standard decomposition graph with `insert-at` and `merge-using` annotations. These can then be used to extract the join tree.

```
UnrankTreeNoCross(r,v)
Input:  a rank $r$ and the root $v$ of the SDG
Output:  adorned SDG
let count($v$) = $[x_0, \ldots, x_m]$;
$k := \min_j r \leq \sum_{i=0}^{j} x_i$;
$r' := r - \sum_{i=0}^{k-1} x_i$;
UnrankLocalTreeNoCross($v$, $r'$, $k$);
```

Unranking: Example

The following table shows the intervals associated with the partitions $\mathcal{T}_G^{e(k)}$ for our standard decomposition graph:

Partition	*Interval*
$\mathcal{T}_G^{e(1)}$	$[1, 5]$
$\mathcal{T}_G^{e(2)}$	$[6, 10]$
$\mathcal{T}_G^{e(3)}$	$[11, 15]$
$\mathcal{T}_G^{e(4)}$	$[16, 18]$

Unranking: the last utility function

The unranking procedure makes use of unranking decompositions and unranking triples. For the latter and a given X, Y, Z, we need to assign each member in

$$\{(x, y, z) | 1 \leq x \leq X, 1 \leq y \leq Y, 1 \leq z \leq Z\}$$

a unique number in $[1, XYZ]$ and base an unranking algorithm on this assignment. We call the function `UnrankTriplet`(r, X, Y, Z). r is a rank and X, Y, and Z are the upper bounds for the numbers in the triplets.

```
UnrankingTreeNoCrossLocal($v, r, k$)
Input:   an SDG node $v$, a rank $r$,
         a number $k$ identifying a partition
Output:  adornments of the SDG as a side-effect
Let $\{w_1, \ldots, w_n\}$ be the children of $v$
switch ($n$) {
  case 0:
    assert($r = 1$ && $k = 0$);
    // no additional adornment for $v$
```

```
case 1:
  let count($v$) = $[c_0, \ldots, c_n]$;
  let summands($v$) = $[s^0, \ldots, s^n]$;
  assert($k \leq n$ && $r \leq c_k$);
  $k_1 = \min_j r \leq \sum_{i=0}^{j} s_i^k$;
  $r_1 = r - \sum_{i=0}^{k_1 - 1} s_i^k$;
  insert-at($v$) = $k$;
  UnrankingTreeNoCrossLocal($w_1, r_1, k_1$);
  case 2:
    let count($v$) = $[c_0, \ldots, c_n]$;
    let summands($v$) = $[s^0, \ldots, s^n]$;
    let count($w_1$) = $[c^1_0, \ldots, c^1_{n_1}]$;
    let count($w_2$) = $[c^2_0, \ldots, c^2_{n_2}]$;
    assert($k \leq n$ && $r \leq c_k$);
    $k_1 = \min_j r \leq \sum_{i=0}^{j} s^k_i$;
    $q = r - \sum_{i=0}^{k_1-1} s^k_i$;
    $k_2 = k - k_1$;
    $(r_1, r_2, a)$ = UnrankTriplet($q, c^1_{k_1}, c^2_{k_2}, \binom{k}{i}$);
    $\alpha$ = UnrankDecomposition($a$);
    merge-using($v$) = $\alpha$;
    UnrankingTreeNoCrossLocal($w_1, r_1, k_1$);
    UnrankingTreeNoCrossLocal($w_2, r_2, k_2$);
}
```

Quick Pick

- build Pseudo-Random join trees fast
- Idea: randomly select an edge in the query graph
- extend join tree by selected edge

```
QuickPick(Query Graph $G$)
**Input:** a query graph $G = (\{R_1, \ldots, R_n\}, E)$
**Output:** a bushy join tree
BestTreeFound = any join tree
**while** stopping criterion not fulfilled
  $E' = E$;
  Trees = $\{R_1, \ldots, R_n\}$;
  **while** ($|$Trees$| > 1$)
    choose $e \in E'$;
    $E' -= e$;
    **if** ($e$ connects two rels in subtrees $T_1, T_2 \in$ Trees, $T_1 \neq T_2$)
      Trees \= $\{T_1, T_2\}$ ;
      Trees += CreateJoinTree($T_1, T_2$);
  Tree = single tree contained in Trees;
  **if** (cost(Tree) $<$ cost(BestTreeFound)) BestTreeFound = Tree;
**return** BestTreeFound
```

Iterative Improvement

- Start with random join tree
- Select rule that improves join tree
- Stop when no further improvement possible

```
IterativeImprovementBase(Query Graph $G$)
Input:  a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output:  a join tree
do {
  JoinTree = random tree
  JoinTree = IterativeImprovement(JoinTree)
  if (cost(JoinTree) < cost(BestTree)) {
    BestTree = JoinTree;
  }
} while (time limit not exceeded)
return BestTree
```

```
IterativeImprovement(JoinTree)
**Input:** a join tree
**Output:** improved join tree
do {
  JoinTree' = randomly apply transformation from the rule set
  if (cost(JoinTree') < cost(JoinTree)) {
    JoinTree = JoinTree';
  }
} while (local minimum not reached)
return JoinTree
```

Simulated Annealing

- II: stuck in local minimum
- SA: allow moves that result in more expensive join trees
- lower the threshold for worsening

```
SimulatedAnnealing(Query Graph $G$)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a join tree
BestTreeSoFar = random tree;
Tree = BestTreeSoFar;
```

```
do {
  do {
    Tree′ = apply random transformation to Tree;
    if (cost(Tree′) < cost(Tree)) {
      Tree = Tree′;
    } else {
      with probability $e^{-(cost(Tree')-cost(Tree))/temperature}$
        Tree = Tree′;
    }
    if (cost(Tree) < cost(BestTreeSoFar)) {
      BestTreeSoFar = Tree′;
    }
  } while (equilibrium not reached)
reduce temperature;
} while (not frozen)
return BestTreeSoFar
```

Tabu Search

- Select cheapest reachable neighbor (even if it is more expensive)
- Maintain tabu set to avoid running into circles

```
TabuSearch(Query Graph)
Input:  a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output:  a join tree
Tree = random join tree;
BestTreeSoFar = Tree;
TabuSet = $\emptyset$;
do {
  Neighbors = all trees generated by applying
              a transformation to Tree;
  Tree = cheapest in Neighbors $\setminus$ TabuSet; $(\star)$
  if (cost(Tree) $<$ cost(BestTreeSoFar)) {
    BestTreeSoFar = Tree;
  }
  if ($|$TabuSet$|$ $>$ limit) remove oldest tree from TabuSet;
  TabuSet += Tree;
} while (not stopping condition satisfied)
return $\pi$;
```

Stop: no improvement in the last *n* cycles, or no tree can be derived in (*)

Genetic Algorithms

- Join trees seen as population
- Successor generations generated by crossover and mutation
- Only the fittest survive

Problem: Encoding

- Chromosome $\longleftrightarrow$ string
- Gene $\longleftrightarrow$ character

Encoding

We distinguish *ordered list* and *ordinal number* encodings.
Both encodings are used for left-deep and bushy trees.
In all cases we assume that the relations $R_1, \ldots, R_n$ are to be joined and use the index i to denote R_i.

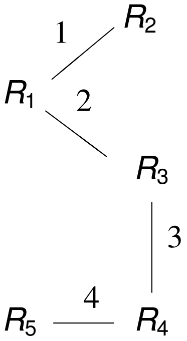
Ordered List Encoding

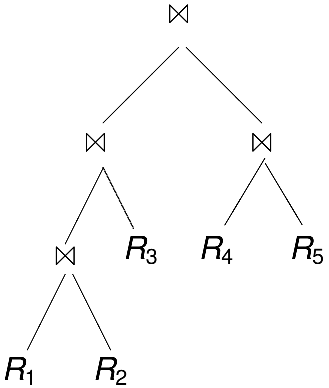
1. left-deep trees
   A left-deep join tree is encoded by a permutation of $1, \ldots, n$. For instance, $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as “1423”.
2. bushy trees
   A bushy join-tree without cartesian products is encoded as an ordered list of the edges in the join graph. Therefore, we number the edges in the join graph. Then, the join tree is encoded in a bottom-up, left-to-right manner.

Ordinal Number Encoding

In both cases, we start with the list $L = \langle R_1, \ldots, R_n \rangle$.

1) left-deep trees
   Within L we find the index of first relation to be joined. If this relation be R_i then the first character in the chromosome string is i. We eliminate R_i from L. For every subsequent relation joined, we again determine its index in L, remove it from L and append the index to the chromosome string.
   For instance, starting with $\langle R_1, R_2, R_3, R_4 \rangle$, the left-deep join tree $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as “1311”.





`1243`

2) bushy trees

   We encode a bushy join tree in a bottom-up, left-to-right manner. Let $R_i \bowtie R_j$ be the first join in the join tree under this ordering. Then we look up their positions in L and add them to the encoding. Then we eliminate R_i and R_j from L and push $R_{i,j}$ to the front of it. We then proceed for the other joins by again selecting the next join which now can be between relations and or subtrees. We determine their position within L, add these positions to the encoding, remove them from L, and insert a composite relation into L such that the new composite relation directly follows those already present.

   For instance, starting with the list $\langle R_1, R_2, R_3, R_4 \rangle$, the bushy join tree $((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$ is encoded as "12 23 12".

Crossover

1. Subsequence exchange
2. Subset exchange

Crossover: Subsequence exchange

The subsequence exchange for the ordered list encoding:

- Assume two individuals with chromosomes $u_1 v_1 w_1$ and $u_2 v_2 w_2$.
- From these we generate $u_1 v_1' w_1$ and $u_2 v_2' w_2$ where v_i' is a permutation of the relations in v_i such that the order of their appearence is the same as in $u_{3-i} v_{3-i} w_{3-i}$.

Subsequence exchange for ordinal number encoding:

- We require that the v_i are of equal length ($|v_1| = |v_2|$) and occur at the same offset ($|u_1| = |u_2|$).
- We then simply swap the v_i.
- That is, we generate $u_1 v_2 w_1$ and $u_2 v_1 w_2$.

Crossover: Subset exchange

The subset exchange is defined only for the ordered list encoding.
Within the two chromosomes, we find two subsequences of equal length comprising the same set of relations. These sequences are then simply exchanged.

Mutation

A mutation randomly alters a character in the encoding.
If duplicates may not occur— as in the ordered list encoding—swapping two characters is a perfect mutation.

Selection

- The probability of survival is determined by its rank in the population.
- We calculate the costs of the join trees encoded for each member in the population.
- Then, we sort the population according to their associated costs and assign probabilities to each individual such that the best solution in the population has the highest probability to survive and so on.
- After probabilities have been assigned, we randomly select members of the population taking into account these probabilities.
- That is, the higher the probability of a member the higher its chance to survive.

The Algorithm

1. Create a random population of a given size (say 128).
2. Apply crossover and mutation with a given rate.
   For example such that 65% of all members of a population participate in crossover, and 5% of all members of a population are subject to random mutation.
3. Apply selection until we again have a population of the given size.
4. Stop after no improvement within the population was seen for a fixed number of iterations (say 30).

Two Phase Optimization

1. For a number of randomly generated initial trees, Iterative Improvement is used to find a local minima.
2. Then Simulated Annealing is started to find a better plan in the neighborhood of the local minima.
   The initial temperature of Simulated Annealing can be lower as is its original variants.

AB Algorithm

1. If the query graph is cyclic, a spanning tree is selected.
2. Assign join methods randomly
3. Apply IKKBZ
4. Apply iterative improvement

Toured Simulated Annealing

The basic idea is that simulated annealing is called *n* times with different initial join trees, if *n* is the number of relations to be joined.

- Each join sequence in the set `Solutions` produced by `GreedyJoinOrdering-3` is used to start an independent run of simulated annealing.

As a result, the starting temperature can be descreased to 0.1 times the cost of the initial plan.

GOO-II

Append an iterative improvement step to GOO

Iterative Dynamic Programming

- Two variants: IDP-1, IDP-2
- Here: Only IDP-1 base version

Idea:

- create join trees with up to k relations
- replace cheapest one by a compound relation
- start all over again

```
IDP-1({$R_1,\ldots,R_n$}, $k$)
Input:  a set of relations to be joined, maximum block size $k$
Output:  a join tree
for (i = 1; i <= n; ++i) {
  BestTree({$R_i$}) = $R_i$;
}
ToDo = {$R_1,\ldots,R_n$};
```

```
while (|ToDo| > 1) {
  $k = \min(k, |\text{ToDo}|)$;
  for (i = 2; i < k; ++i)
    for all $S \subseteq$ ToDo, $|S| = i$ do
      for all $O \subset S$ do
        BestTree($S$) = CreateJoinTree(BestTree($S \setminus O$),
                                      BestTree($O$));
  find $V \subset$ ToDo, $|V| = k$
  with cost(BestTree($V$)) = min{cost(BestTree($W$)) |
                                $W \subset$ ToDo, $|W| = k$};
  generate new symbol $T$;
  BestTree({$T$}) = BestTree($V$);
  ToDo = (ToDo \ $V$) ∪ {$T$};
  for all $O \subset V$ do delete(BestTree($O$));
}
return BestTree({$R_1, \ldots, R_n$});
```

Ordering Order-Preserving Joins

- Motivation: XQuery
- Order-Preserving Selection:

$$\hat{\sigma}_p(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ \alpha(e) \oplus \hat{\sigma}_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \hat{\sigma}_p(\tau(e)) & \text{else} \end{cases}$$

Order-Preserving Join

Order-preserving cross product:

$$e_1 \hat{\times} e_2 := (\alpha(e_1) \overline{\times} e_2) \oplus (\tau(e_1) \times e_2)$$

where

$$e_1 \overline{\times} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \overline{\times} \tau(e_2)) & \text{else} \end{cases}$$

Order-preserving join:

$$e_1 \hat{\bowtie}_p e_2 := \hat{\sigma}_p(e_1 \hat{\times} e_2)$$

Equivalences

$$
\begin{array}{lcll}
\hat{\sigma}_{p_1}(\hat{\sigma}_{p_2}(e)) & = & \hat{\sigma}_{p_2}(\hat{\sigma}_{p_1}(e)) & \\
\hat{\sigma}_{p_1}(e_1 \hat{\bowtie}_{p_2} e_2) & = & \hat{\sigma}_{p_1}(e_1) \hat{\bowtie}_{p_2} e_2 & \text{if} \quad \mathcal{F}(p_2) \subseteq \mathcal{A}(e_1) \\
\hat{\sigma}_{p_1}(e_1 \hat{\bowtie}_{p_2} e_2) & = & e_1 \hat{\bowtie}_{p_2} \hat{\sigma}_{p_1}(e_2) & \text{if} \quad \mathcal{F}(p_2) \subseteq \mathcal{A}(e_2) \\
e_1 \hat{\bowtie}_{p_1}(e_2 \hat{\bowtie}_{p_2} e_3) & = & (e_1 \hat{\bowtie}_{p_1} e_2) \hat{\bowtie}_{p_2} e_3 & \text{if} \quad \mathcal{F}(p_i) \subseteq \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1})
\end{array}
$$

Example

Let $R_1 = \langle [a:1], [a:2] \rangle$ and $R_2 = \langle [b:1], [b:2] \rangle$. Then

$$\begin{array}{rcl} R_1 \hat{\bowtie}_{true} R_2 & = & \langle [a:1, b:1], [a:1, b:2], [a:2, b:1], [a:2, b:2] \rangle \\ R_2 \hat{\bowtie}_{true} R_1 & = & \langle [a:1, b:1], [a:2, b:1], [a:1, b:2], [a:2, b:2] \rangle \end{array}$$

Hence, order-preserving join is not commutative.

Algorithm

- Analogous to matrix multiply
- Algorithm also pushes selections down
- `construct-bushy-tree` fills arrays p,s,c,t

p	remember applicable predicates
s	remember statistics
c	remember costs
t	remember split

- `extract-plan` extracts plan

```
construct-bushy-tree($\mathcal{R}$, $\mathcal{P}$)

$n = |\mathcal{R}|$
for $i = 1$ to $n$
    $\mathcal{B}$ =applicable-predicates($R_i$, $\mathcal{P}$)
    $\mathcal{P} = \mathcal{P} \setminus \mathcal{B}$
    $p[i,i] = \mathcal{B}$
    $s[i,i] = S_0(R_i, \mathcal{B})$
    $c[i,i] = C_0(R_i, \mathcal{B})$
```

```
for $l = 2$ to $n$
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $\mathcal{B} =$ applicable-predicates($R_{i...j}$, $\mathcal{P}$)
        $\mathcal{P} = \mathcal{P} \setminus \mathcal{B}$
        $p[i, j] = \mathcal{B}$
        $s[i, j] = S_1(s[i, j - 1], s[j, j], \mathcal{B})$
        $c[i, j] = \infty$
        for $k = i$ to $j - 1$
            $q = c[i, k] + c[k + 1, j] + C_1(s[i, k], s[k + 1, j], \mathcal{B})$
            if (q < c[i,j])
                $c[i, j] = q$
                $t[i, j] = k$
```

```
extract-plan($\mathcal{R}, t, p$)

    return extract-subplan($\mathcal{R}, t, p, 1, |\mathcal{R}|$)

extract-subplan($\mathcal{R}, t, p, i, j$)

    if ($j > i$)
        X = extract-subplan($\mathcal{R}, t, p, i, t[i, j]$)
        Y = extract-subplan($\mathcal{R}, t, p, t[i, j] + 1, j$)
        return $X \hat{\bowtie}_{p[i,j]} Y$
    else
        return $\sigma_{p[i,i]}(R_i)$
```

Database Items, Access Paths, and Building Blocks

In this chapter we go into some details:

- Deep into the (Runtime) System
- Close to the Hardware

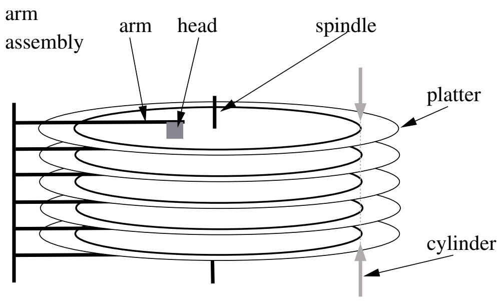
Goal:

- Estimation and Optimization of Disk Access Costs

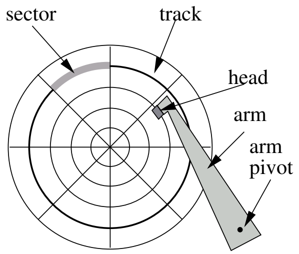
Database Items, Access Paths, and Building Blocks

1. Disk Drives
2. Database Buffer
3. Physical Database Organization
4. Slotted Pages and Tuple Identifier (TID)
5. Physical Record Layouts
6. Physical Algebra
7. Simple Scan
8. Scan and Attribute Access
9. Temporal Relations
10. Table Functions
11. Indexes
12. Single Index Access Paths
13. Multi Index Access Paths
14. Indexes and Joins
15. Counting the Number of Accesses
16. Disk Drive Costs for *N* Uniform Accesses

4.1 Disk Drive: Assembly



a. side view



b. top view

4.1 Disk Drive: Zones I

- Outer tracks/sectors longer than inner ones
- Highest density is fixed
- Results in waste in outer sectors
- Thus: cylinders organized into zones

4.2 Disk Drive: Zones II

- Every zone contains a fixed number of consecutive cylinders
- Every cylinder in a zone has the same number of sectors per track
- Outer zones have more sectors per track than inner zones
- Since rotation speed is fixed: higher throughput on outer cylinders

4.1 Disk Drive: Track Skew

Read all sectors of all tracks of some consecutive cylinders:

- Read all sectors of one track
- Switch to next track: small adjustment of head necessary called: *head switch*
- This causes tiny delay
- Thus, if all tracks start at the same angular position then we miss the start of the first sector of the next track
- Remedy: *track skew*

4.1 Disk Drive: Cylinder Skew

Read all sectors of all tracks of some consecutive cylinders:

- Read all sectors of all tracks of some cylinder
- Switching to the next cylinder causes some delay
- Again, we miss the start of the first sector, if the tracks start all start at the same angular position
- Remedy: *cylinder skew*

4.1 Disk Drive: Addressing Sectors

- Physical Address: cylinder number, head (surface) number, sector number
- Logical Address: LBN (logical block number)

4.2 Disk Drive: LBN to Physical Address

Mapping:

Cylinder	Track	LBN	number of sectors per track
0	0	0	573
	1	573	573
...	...	...	...
	5	2865	573
1	0	3438	573
...	...	...	...
15041	0	35841845	253
...	...	...	...

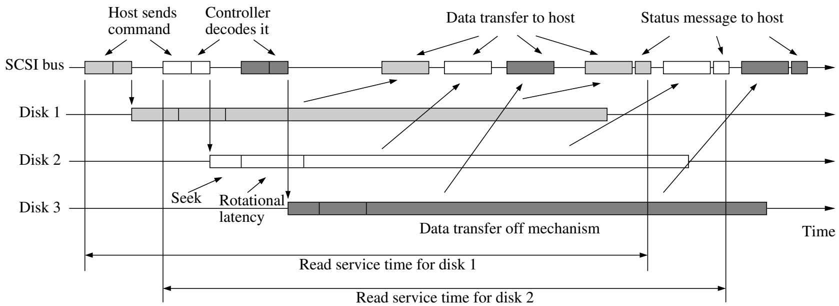
4.1 Disk Drive: LBN to Physical Address

This ideal view of the mapping is disturbed by *bad blocks*

- due to the high density, no perfect manufacturing is possible
- As a consequence *bad blocks* occur (sectors that cannot be used)
- Reserve some blocks, tracks, cylinders for remapping bad blocks

Bad blocks may cause hickups during sequential reads

4.1 Disk Drive: Reading/Writing a Block



4.1 Disk Drive: Reading/Writing a Block

1. The host sends the SCSI command.
2. The disk controller decodes the command and calculates the physical address.
3. During the seek the disk drive's arm is positioned such that the according head is correctly placed over the cylinder where the requested block resides. This step consists of several phases.
   - 3.1 The disk controler accelerates the arm.
   - 3.2 For long seeks, the arm moves with maximum velocity (coast).
   - 3.3 The disk controler slows down the arm.
   - 3.4 The disk arm settles for the desired location. Note: settle times differ for reads and writes
4. The disk has to wait until the sector where the requested block resides comes under the head (rotation latency).
5. The disk reads the sector and transfers data to the host.
6. Finally, it sends a status message.

4.2 Disk Drive: Optimizing Round Trip Time

- caching
- read-ahead
- command queuing

4.1 Disk Drive: Seek Time

A good approximation of the seek time where d cylinders have to be travelled is given by

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

where the constants c_i are disk specific. The constant c_0 indicates the maximum number cylinders where no coast takes place: seeking over a distance of more than c_0 cylinders results in a phase where the disk arm moves with maximum velocity.

4.1 Disk Drive: Cost model: initial thoughts

Disk access costs depend on

- the current position of the disk arm and
- the angular position of the platters

Both are not known at query compilation time
Consequence:

- Estimating the costs of a single disk access at query compilation time may result in large estimation error

Better: costs of many accesses
Nonetheless: First Simplistic Cost Model to give a feeling for disk drive access costs

4.1 Disk Drive: Simplistic Cost Model

We introduce some disk drive parameters for out simplistic cost model:

- average latency time: average time for positioning (seek+rotational delay)
  - use average access time for a single request
  - Estimation error can (on the average) be as “low” as 35%
- sustained read/write rate:
  - after positioning, rate at which data can be delivered using sequential read

4.1 Disk Drive: Model 2004

A hypothetical disk (inspired by disks available in 2004) then has the following parameters:

Model 2004		
Parameter	Value	Abbreviated Name
capacity	180 GB	$\mathtt{D}_{\mathtt{cap}}$
average latency time	5 ms	$\mathtt{D}_{\mathtt{lat}}$
sustained read rate	100 MB/s	$\mathtt{D}_{\mathtt{srr}}$
sustained write rate	100 MB/s	$\mathtt{D}_{\mathtt{swr}}$

The time a disk needs to read and transfer *n* bytes is then approximated by $\mathtt{D}_{\mathtt{lat}} + n/\mathtt{D}_{\mathtt{srr}}$.

4.1 Disk Drive: Sequential vs. Random I/O

Database management system developers distinguish between

- *sequential* I/O and
- *random* I/O.

In our simplistic cost model:

- For sequential I/O, there is only one positioning at the beginning and then, we can assume that data is read with the sustained read rate.
- For random I/O, one positioning for every unit of transfer—typically a page of say 8 KB—is assumed.

4.2 Disk Drive: Simplistic Cost Model

Read 100 MB

- Sequential read: 5 ms + 1 s
- Random read (8K pages): 65 s

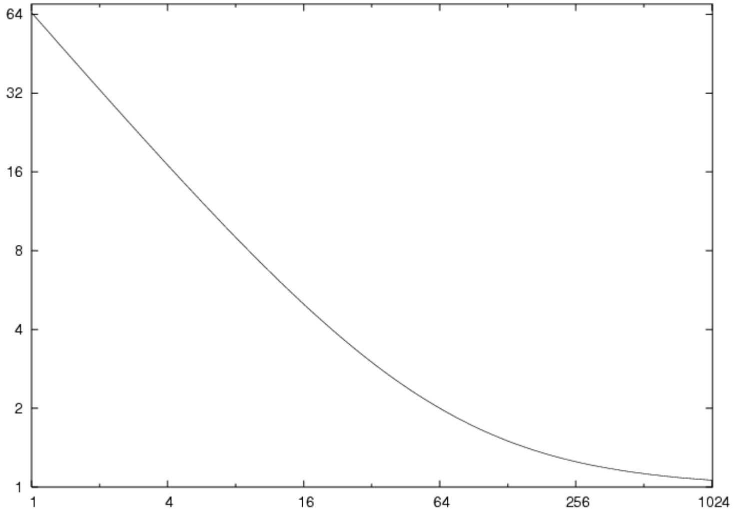
4.1 Disk Drive: Simplistic Cost Model

Problems:

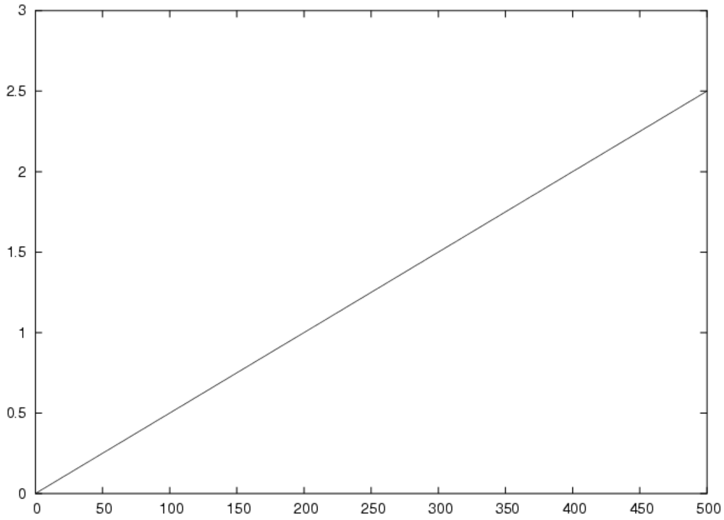
- other applications
- other transactions
- other read operations in the same QEP

may request blocks from the same disk and move away the head(s) from the current position
Further: 100 MB sequential search poses problem to buffer manager

4.1 Disk Drive: Time to Read 100 MB (x: number of 8 KB chunks)



4.1 Disk Drive: Time to Read *n* Random Pages



4.1 Disk Drive: Simplistic Cost Model

100 MB can be stored on 12800 8 KB pages.
In our simplistic cost model, reading 200 pages randomly costs about the same as reading 100 MB sequentially.
That is, reading 1/64th of 100 MB randomly takes as long as reading the 100 MB sequentially.

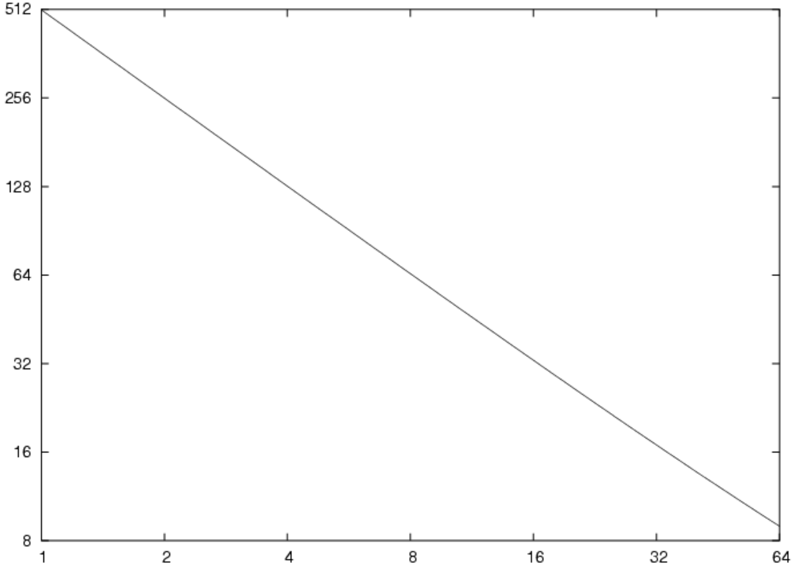
4.1 Disk Drive: Simplistic Cost Model

Let us denote by a the positioning time, s the sustained read rate, p the page size, and d some amount of consecutively stored bytes. Let us calculate the break even point

$$\begin{aligned} n * (a + p/s) &= a + d/s \\ n &= (a + d/s)/(a + p/s) \\ &= (as + d)/(as + p) \end{aligned}$$

a and s are disk parameters and, hence, fixed. For a fixed d, the break even point depends on the page size.
Next Figure: x-axis: is the page size p in multiples of 1 K;
y-axis: $(d/p)/n$ for d = 100 MB.

4.2 Disk Drive: Break Even Point (depending on page size)



4.1 Disk Drive: Two Lessons Learned

- sequential read is much faster than random read
- the runtime system should secure sequential read

The latter point can be generalized:

- the runtime system of a database management system has, as far as query execution is concerned, two equally important tasks:
    - allow for efficient query evaluation plans and
    - allow for smooth, simple, and robust cost functions.

4.1 Disk Drive: Measures to Achieve the Above

Typical measures on the database side are

- carefully chosen physical layout on disk (e.g. cylinder or track-aligned extents, clustering)
- disk scheduling, multi-page requests
- (asynchronous) prefetching,
- piggy-back scans,
- buffering (e.g. multiple buffers, replacement strategy) and last but not least
- efficient and robust algorithms for algebraic operators

4.1 Disk Drive: Parameters

D_{cyl}	total number of cylinders
D_{track}	total number of tracks
D_{sector}	total number of sectors
D_{tpc}	number of tracks per cylinder (= number of surfaces)
D_{cmd}	command interpretation time
D_{rot}	time for a full rotation
D_{rdsettle}	time for settle for read
D_{wrsettle}	time for settle for write
D_{hdswitch}	time for head switch

D_{Zone}	total number of zones
$D_{\text{Zcyl}}(i)$	number of cylinders in zone i
$D_{\text{Zspt}}(i)$	number of sectors per track in zone i
$D_{\text{Zspc}}(i)$	number of sectors per cylinder in zone i ($= D_{\text{tpc}} D_{\text{Zspt}}(i)$)
$D_{\text{Zscan}}(i)$	time to scan a sector in zone i ($= D_{\text{rot}}/Dzspti$)

D_{avgseek}	average seek costs
D_{c_0}	parameter for seek cost function
D_{c_1}	parameter for seek cost function
D_{c_2}	parameter for seek cost function
D_{c_3}	parameter for seek cost function
D_{c_4}	parameter for seek cost function

$D_{\text{seek}}(d)$ cost of a seek of d cylinders

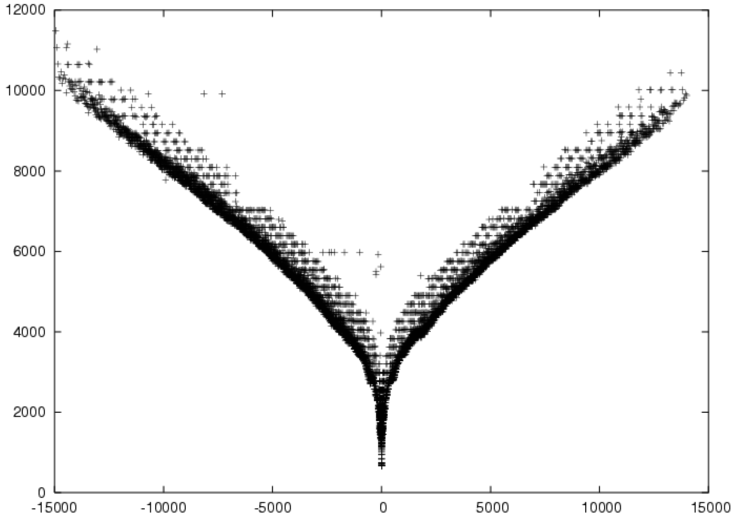
$$D_{\text{seek}}(d) = \begin{cases} D_{c_1} + D_{c_2}\sqrt{d} & \text{if } d \leq D_{c_0} \\ D_{c_3} + D_{c_4} d & \text{if } d > D_{c_0} \end{cases}$$

$D_{\text{rot}}(s, i)$ rotation cost for s sectors of zone i $(= sD_{\text{Zscan}}(i))$

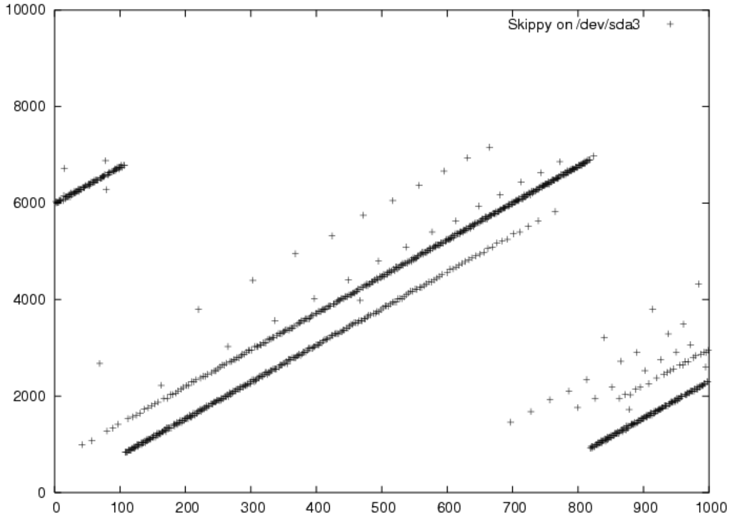
4.1 Disk Drive: Extraction of Disk Drive Parameters

- Documentation: often not sufficient
- Mapping: Interrogation via SCSI-Mapping command (disk drives lie)
- Use benchmarking tools, e.g.:
  - Diskbench
  - Skippy (Microbenchmark)
  - Zoned

4.1 Disk Drive: Seek Curve Measured with Diskbench



4.1 Disk Drive: Skippy Benchmark Example



4.2 Disk Drive: Interpretation of Skippy Results

- x-axis: distance (sectors)
- y-axis: time
- difference topmost/bottommost line: rotational latency
- difference two lowest ‘lines’: head switch time
- difference lowest ‘line’ topmost spots: cylinder switch time
- start lowest ‘line’: minimal time to media
- plus other parameters

4.1 Disk Drive: Upper bound on Seek Time

Theorem (Qyang)

If the disk arm has to travel over a region of C cylinders, it is positioned on the first of the C cylinders, and has to stop at $s - 1$ of them, then $sD_{seek}(C/s)$ is an upper bound for the seek time.

4.2 Database Buffer

The database buffer

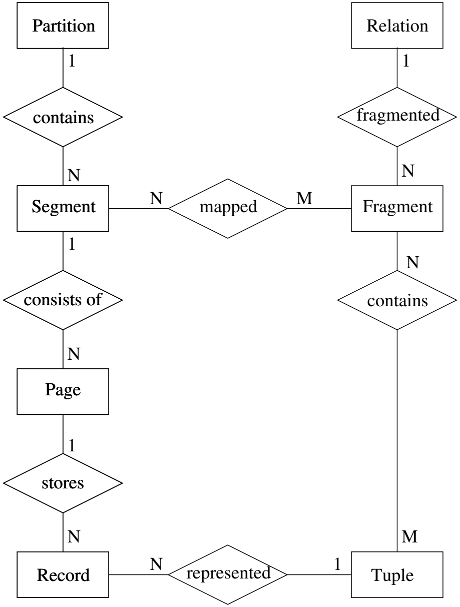
1. is a finite piece of memory,
2. typically supports a limited number of different page sizes (mostly one or two),
3. is often fragmented into several buffer pools,
4. each having a replacement strategy (typically enhanced by hints).

Given the page identifier, the buffer frame is found by a hashtable lookup.
Accesses to the hash table and the buffer frame need to be synchronized.
Before accessing a page in the buffer, it must be fixed.
These points account for the fact that the costs of accessing a page in the buffer are therefore greater than zero.

4.3 Physical Database Organization

1. partition: sequence of pages (consecutive on disk)
2. extent: subsequence of a partition
3. segment (file): logical sequence of pages (implemented e.g. as set of extents)
4. record: sequence of bytes stored on a page

Mapping of a relation's tuples onto records stored on pages in segments:



Partition
1
contains
N
Segment
N
mapped
M
Fragment
1
consists of
N
Page
1
stores
N
Record
N
represented
1
Tuple
Relation
1
fragmented
N
N
contains
M

4.3 Access to Database Items

- database item: something stored in DB
- database item can be set (bag, sequence) of items
- access to a database item then produces stream of smaller database items
- the operation that does so is called *scan*

4.3 Example

Using a relation scan `rscan`, the query

select *
from Student

can be answered by `rscan(Student)`
(segments? extents?): Assumption:

- segment scans and each relation stored in one segment
- segment and relation name identical

Then `fscan(Student)` and `Student` denote scans of all tuples in a relation

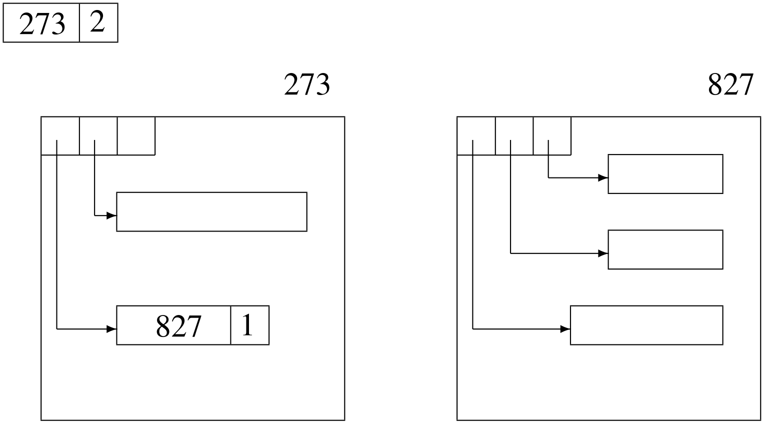
4.3 Model of a Segment

- For our cost model, we need a model of segments.
- We assume an extent-based segment implementation.
- Every segment then is a sequence of extents.
- Every extent can be described by a pair (F_j, L_j) containing its first and last cylinder.
  (For simplicity, we assume that extents span whole cylinders.)
- An extent may cross a zone boundary.
- Hence: split extents to align them with zone boundaries.
- Segment can be described by a sequence of triples (F_i, L_i, z_i) ordered on F_i where z_i is the zone number in which the extent lies.

4.3 Model of a Segment

S_{ext}	number of extents in the segment
$S_{\text{first}}(i)$	first cylinder in extent i (F_i)
$S_{\text{last}}(i)$	last cylinder in extent i (L_i)
$S_{\text{zone}}(i)$	zone of extent i (z_i)
$S_{\text{cpe}}(i)$	number of cylinders in extent i ($= S_{\text{last}}(i) - S_{\text{first}}(i) + 1$)
S_{sec}	total number of sectors in the segment ($= \sum_{i=1}^{S_{\text{ext}}} S_{\text{cpe}}(i) D_{Zspc}(S_{\text{zone}}(i))$)

4.4 Slotted Page



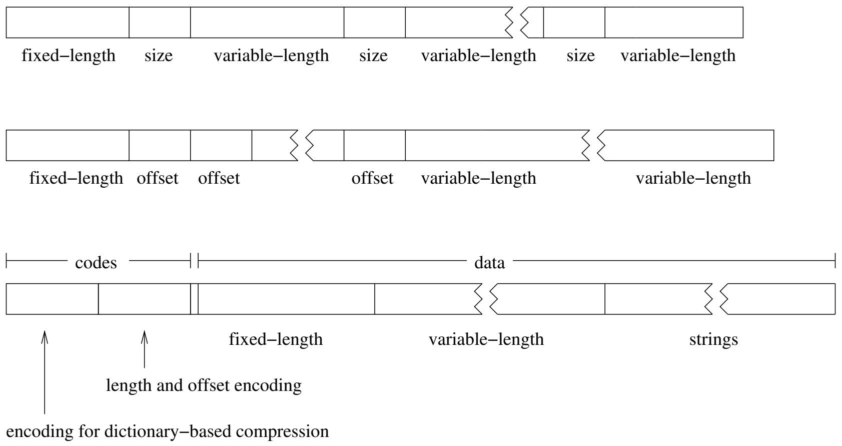
4.4 Tuple Identifier (TID)

TID is conjunction of

- page identifier (e.g. partition/segment no, page no)
- slot number

TID sometimes called Row Identifier (RID)

4.5 Record Layout



4.5 Record Layout

Consequence:

- Accessing an attribute value has non-zero cost

4.6 Physical Algebra (Iterator Concept)

Remember from DBS I:

- open
- next
- close

Interface for *iterators*.
All physical algebraic operators are implemented as iterators.

4.7 Simple Scan

- An `rscan` operation is rarely supported.
- Instead: scans on segments (files).
- Since a (data) segment is sometimes called *file*, the correct plan for the above query is often denoted by `fscan(Student)`.

Several assumptions must hold:

- the `Student` relation is not fragmented, it is stored in a single segment,
- the name of this segment is the same as the relation name, and
- no tuples from other relations are stored in this segment.

Until otherwise stated, we assume that these assumptions hold. Instead of `fscan(Student)`, we could then simply use `Student` to denote leaf nodes in a query execution plan.

4.8 Attributes/Variables and their Binding

```
select  *
from    Student
```

can be expressed as `Student`[*s*] instead of `Student`.
Result type: set of tuples with a single attribute *s*.
s is assumed to bind a pointer

- to the physical record in the buffer holding the current tuple
  or
- a pointer to the slot pointing to the record holding the current tuple

4.8 Building Block

- scan
- a leaf of a query execution plan

Leaf can be complex.
But: Plan generator does not try to reorder within building blocks
Nonetheless:

- building block organized around a single database item

If more than a single database item is involved: *access path*

4.8 Scan and Attribute Access

Strictly speaking, the plan

$$\sigma_{\texttt{age}>30}(\texttt{Student}[s])$$

is incorrect (`age` is not bound!)
We have a choice:

- implicit attribute access
- make attribute accesses explicit

4.8 Scan and Attribute Access

Explicit attribute access:

$$\sigma_{s.\texttt{age} > 30}(\texttt{Student}[s])$$

Advantage: makes attribute access costs explicit

4.8 Scan and Attribute Access

Consider:

$$\sigma_{s.\texttt{age}>30 \wedge s.\texttt{age}<40}(\texttt{Student}[s])$$

Problem: accesses `age` twice

4.8 Scan and Attribute Access

Map operator:

$$\chi_{a_1:e_1,\ldots,a_n:e_n}(e) := \{t \circ [a_1 : c_1, \ldots, a_n : c_n] | t \in e, c_i = e_i(t) \; \forall \; (1 \leq i \leq n)\}$$

Scan and Attribute Access

The above problem can now be solved by

$$\sigma_{\texttt{age}>30 \wedge \texttt{age}<40}(\chi_{\texttt{age}:\boldsymbol{s}.\texttt{age}}(\texttt{Student}[\boldsymbol{s}])).$$

In general, it is beneficial to load attributes as late as possible. The latest point at which all attributes must be read from the page is typically just before a pipeline breaker.

4.8 Scan and Attribute Access

select name
from Student
where age $>$ 30

The plan

$$\Pi_n(\chi_{n:s.\mathtt{name}}(\sigma_{a>30}(\chi_{a:s.\mathtt{age}}(\mathtt{Student}[s]))))$$

is better than

$$\Pi_n(\sigma_{a>30}(\chi_{n:s.\mathtt{name},a:s.\mathtt{age}}(\mathtt{Student}[s])))$$

4.8 Scan and Attribute Access

Alternative to this selective successive attribute access:

- scan has list of attributes to be projected (accessed, copied)
- predicate is applied before processing the projection list

4.8 Scan and Attribute Access

predicate evaluable on disk representation is called *SARGable* (search argument)

- boolean expression in simple predicates of the form $A\theta c$

If a predicate can be used for an index lookup: index SARGable
Other predicates: residual predicates

4.8 Scan and Attribute Access

$\mathtt{R}[v; p]$ equivalent to $\sigma_p(\mathtt{R[v]})$ but cheaper to evaluate

Remark

- if p is conjunct, order by $(f_i - 1)/c_i$

Example:

$$\mathtt{Student}[s; \mathtt{age} > 30, \mathtt{name\ like}\ `\%\mathrm{m}\%']$$

4.9 Temporal Relations

select e.name, d.name
from Emp e, Dept d
where e.age $>$ 30 **and** e.age $<$ 40 **and** e.dno = d.dno

can be evaluated by

$$\texttt{Dept}[d] \bowtie^{\texttt{nl}}_{e.\texttt{dno}=d.\texttt{dno}} \sigma_{e.\texttt{age}>30 \wedge e.\texttt{age}<40}(\texttt{Emp}[d]).$$

Better:

$$\texttt{Dept}[d] \bowtie^{\texttt{nl}}_{e.\texttt{dno}=d.\texttt{dno}} \mathrm{Tmp}(\sigma_{e.\texttt{age}>30 \wedge e.\texttt{age}<40}(\texttt{Emp}[d])).$$

Or:

1. $R_{\texttt{tmp}} = \sigma_{e.\texttt{age}>30 \wedge e.\texttt{age}<40}(\texttt{Emp}[d]);$
2. $\texttt{Dept}[d] \bowtie^{\texttt{nl}}_{e.\texttt{dno}=d.\texttt{dno}} R_{\texttt{tmp}}[e]$

4.10 Table Functions

A *table function* is a function that returns a relation.
Example query:

select *
from TABLE(Primes(1,100)) as p

Translation:

$$\texttt{Primes}(1, 100)[p]$$

Looks the same as regular scan, but is of course computed differently.

4.10 Table Functions

Special birthdays of Anton:

```
select  *
from    Friends f,
        TABLE(Primes(
        CURRENT_YEAR, EXTRACT(YEAR FROM f.birthday) + 100)) as p
where   f.name = 'Anton'
```

Note: The result of the table function depends on our friend Anton.
Translation: uses d-join

4.10 Table Functions

Definition d-join:

$$R \Join S = \{r \circ s | r \in R, s \in S(t)\}.$$

Translation of the above query:

$$\chi_{\text{b}:XTRY(f.birthday)+100}(\sigma_{\text{f}.name=\text{'}Anton\text{'}}(\texttt{Friends}[f]))\Join\texttt{Primes}(\texttt{c},\texttt{b})[p]$$

where we assume that some global entity c holds the value of `CURRENT_YEAR`.

4.10 Table Functions

The same for all friends:

```
select *
from   Friends f,
       TABLE(Primes(
       CURRENT_YEAR, EXTRACT(YEAR FROM f.birthday) + 100)) as p
```

Better:

```
select *
from   Friends f,
       TABLE(Primes(
       CURRENT_YEAR, (select max(birthday) from Friends) + 100)) as p
where  p.prime ≥ f.birthday
```

At the algebraic level: this optimization requires some knowledge

4.11 Indexes

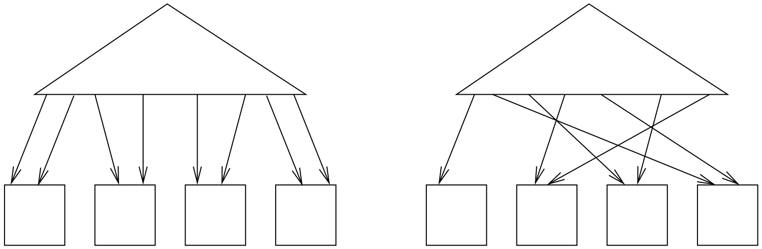
We consider B-Trees only

- key attributes: $a_1, \ldots, a_n$
- data attributes: $d_1, \ldots, d_m$
- Often: one special data attribute holding the TID of a tuple

Some notions:

- simple/complex key
- unique/non-unique index
- index-only relation (no TIDs available!)
- clustered/non-clustered index

4.11 Indexes



4.12 Single Index Access Path

Exact match query:

select name
from Emp
where eno = 1077

Translation:

$$\Pi_{\mathtt{name}}(\chi_{e:*(x.\mathrm{TID}),\mathtt{name}:\mathtt{e.name}}(\mathtt{Emp}_{\mathtt{eno}}[x;\mathtt{eno}=1077]))$$

Alternative translation using d-join:

$$\Pi_{\mathtt{name}}(\mathtt{Emp}_{\mathtt{eno}}[x;\mathtt{eno}=1077] \ltimes \chi_{e:*(x.\mathrm{TID}),\mathtt{name}:\mathtt{e.name}}(\Box))$$

(x: holds ptr to index entry; *: dereference TID)

4.12 Single Index Access Path

Range query:

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35

Translation:

$$\Pi_{\mathtt{name}}(\chi_{e:*(x.\mathrm{TID}),\mathtt{name}:\mathtt{e.name}}(\mathtt{Emp}_{\mathtt{age}}[x; 25 \leq \mathtt{age}; \mathtt{age} \leq 35]))$$

(Start and Stop condition)

4.12 Single Index Access Path

Turning random I/O into sequential I/O:

$\Pi_{\mathtt{name}}(\chi_{e:*(\mathsf{TID}),\mathtt{name}:\mathtt{e.name}}(\mathsf{Sort}_{\mathsf{TID}}(\mathtt{Emp}_{\mathtt{age}}[x; 25 \leq \mathtt{age}; \mathtt{age} \leq 35; \mathsf{TID}])))$

Note: explicit projection the TID attribute of the index within the index scan.

4.12 Single Index Access Path

Query demanding ordered output:

select	name, age
from	Emp
where	age $\geq$ 25 **and** age $\leq$ 35
order by	age

Translation:

$$\Pi_{\texttt{name},\texttt{age}}(\chi_{e:*(x.\text{TID}),\texttt{name}:e.\texttt{name}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35]))$$

Note: output of index scan ordered on its key attributes
This order can be exploited in many ways: e.g.: subsequent merge join

4.12 Single Index Access Path

Turning random I/O into sequential I/O requires resort:

$$\Pi_{\mathtt{name},\mathtt{age}}(\mathrm{Sort}_{\mathtt{age}}(\chi_{e:*(\mathrm{TID}),\mathtt{name}:e.\mathtt{name}}(\\ \mathrm{Sort}_{\mathrm{TID}}(\mathtt{Emp}_{\mathtt{age}}[x; 25 \leq \mathtt{age}; \mathtt{age} \leq 35; \mathrm{TID}]))))$$

Possible speedup of sort by dense numbering:

$$\Pi_{\mathtt{name},\mathtt{age}}(\\ \mathrm{Sort}_{\mathtt{rank}}(\\ \chi_{e:*(\mathrm{TID}),\mathtt{name}:e.\mathtt{name}}(\\ \mathrm{Sort}_{\mathrm{TID}}(\\ \chi_{\mathtt{rank}:\mathtt{counter}{+}{+}}(\\ \mathtt{Emp}_{\mathtt{age}}[x; 25 \leq \mathtt{age}; \mathtt{age} \leq 35; \mathrm{TID}])))))$$

4.12 Single Index Access Path

Some predicates not index sargable but still useful as residual predicates:

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35 **and** age $\neq$ 30

Translation:

$$\Pi_{\texttt{name}}(\chi_{t:x.\text{TID},e:*t,\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35; \texttt{age} \neq 30]))$$

4.12 Single Index Access Path

Non-inclusive bounds:

select name
from Emp
where age $>$ 25 **and** age $<$ 35

Supported by index:

$$\Pi_{\texttt{name}}(\chi_{t:x.\text{TID},e:*t,\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{age}}[x; 25 < \texttt{age}; \texttt{age} < 35]))$$

Unsupported:

$$\Pi_{\texttt{name}}(\chi_{t:x.\text{TID},e:*t,\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35; \texttt{age} \neq 25, \texttt{age} \neq 35]))$$

Especially for predicates on strings this might be expensive.

4.12 Single Index Access Path

Start and stop conditions are optional:

select name
from Emp
where age $\geq$ 60

or

select name
from Emp
where age $\leq$ 20

4.12 Single Index Access Path

Full index scan also useful:

```
select  count(*)
from    Emp
```

Also works for sum/avg.
(notion: index only query)

4.12 Single Index Access Path

Min/max even more efficient:

select min/max(salary)
from Emp

4.12 Single Index Access Path

```
select  name
from    Emp
where   salary = (select  max(salary)
                  from    Emp)
```

Alternatives: one or two descents into the index.

4.12 Single Index Access Path

Full index scan:

select	salary
from	Emp
order by	salary

Translation:

$$\texttt{Emp}_{\texttt{salary}}$$

4.12 Single Index Access Path

Predicate on string attribute:

select name, salary
from Emp
where name $\geq$ 'Maaa'

Start condition: $'\texttt{Maaa}' \leq \texttt{name}$

select name, salary
from Emp
where name **like** 'M%'

Start condition: $'\texttt{M}' \leq \texttt{name}$

4.12 Single Index Access Path

An *access path* is a plan fragment with building blocks concerning a single database item.
Hence, every building block is an access path.
The above plans mostly touch two database items: a relation and an index on some attribute of that relation.
If we say that an index concerns the relation that it indexes, such a fragment is an access path.
For relational systems, the most general case of an access path uses several indexes to retrieve the tuples of a single relation.
We will see examples of these more complex access paths in the following section.
A query that can be answered solely by accessing indexes is called an *index only query*.

4.12 Single Index Access Path

Query with `IN`:

select name
from Emp
where age in $\{28, 29, 31, 32\}$

Take min/max value for start/stop key plus one of the following as the residual predicate:

- $\texttt{age} = 28 \lor \texttt{age} = 29 \lor \texttt{age} = 31 \lor \texttt{age} = 32$
- $\texttt{age} \neq 30$

4.12 Single Index Access Path

A case for the d-join:

select name
from Emp
where salary in $\{1111, 11111, 111111\}$

With Sal = $\{[s : 1111], [s : 11111], [s : 111111]\}$:

$$\texttt{Sal}[S] \Join \chi_{e:*\mathrm{TID},\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{salary}}[x; \texttt{salary} = S.s; \mathrm{TID}])$$

(explain: gap skipping/zig-zag skipping)

4.12 Single Index Access Path

In general an index can have a complex key comprising of key attributes $k_1, \ldots, k_n$ and data attributes $d_1, \ldots, d_m$.
Besides a full index scan, the index can be descended to directly search for the desired tuple(s):
If the search predicate is of the form

$$k_1 = c_1 \wedge k_2 = c_2 \wedge \ldots \wedge k_j = c_j$$

for some constants c_i and some $j <= n$, we can generate the start and stop condition

$$k_1 = c_1 \wedge \ldots \wedge k_j = c_j.$$

4.12 Single Index Access Path

With ranges things become more complex and highly dependent on the implementation of the facilities of the B-Tree:

$$k_1 = c_1 \wedge k_2 \geq c_2 \wedge k_3 = c_3$$

Obviously, we can generate the start condition $k_1 = c_1 \wedge k_2 \geq c_2$ and the stop condition $k_1 = c_1$.
Here, we neglected the condition on k_3 which becomes a residual predicate.
However, with some care we can extend the start condition to $k_1 = c_1 \wedge k_2 \geq c_2 \wedge k_3 = c_3$:
we only have to keep $k_3 = c_3$ as a residual predicate since for k_2 values larger than c_2 values different from c_3 can occur for k_3.

4.12 Single Index Access Path

If closed ranges are specified for a prefix of the key attributes as in

$$a_1 \leq k_1 \leq b_1 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

we can generate the start key $k_1 = a_1 \wedge \ldots \wedge k_j = a_j$, the stop key $k_1 = b_1 \wedge \ldots \wedge k_j = b_j$, and

$$a_2 \leq k_2 \leq b_2 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

as the residual predicate.
If for some search key attribute k_j the lower bound a_j is not specified, the start condition can not contain k_j and any k_{j+i}.
If for some search key attribute k_j the upper bound b_j is not specified, the stop condition can not contain k_j and any k_{j+i}.

4.12 Single Index Access Path

Two further enhancements of the B-Tree functionality possibly allow for alternative start/stop conditions:

- The B-Tree implemenation allows to specify the order (ascending or descending) for each key attribute individually.
- The B-Tree implementation implements forward and backward scans

4.12 Single Index Access Path

Consider search predicate:

```
haircolor = 'blond' and height between 180 and 190
```

and index on

```
sex, haircolor, height
```

There are only the two values `male` and `female` available for `sex`.
Rewrite:

```
(sex = 'm' and haircolor = 'blond'
     and height between 180 and 190)
or
(sex = 'f' and haircolor = 'blond'
     and height between 180 and 190)
```

Improvement: determine rewrite at query execution time in conjunction with gap skipping.

4.13 Multi Index Access Path

Query:

select *
from Camera
where megapixel $>$ 5 **and** distortion $<$ 0.05
and noise $<$ 0.01
zoomMin $<$ 35 **and** zoomMax $>$ 105

Indexes on all attributes

4.13 Multi Index Access Path

Translation:

$$
((((\\
\quad \texttt{Camera}_{\texttt{megapixel}}[c; \texttt{megapixel} > 5; \text{TID}]\\
\quad \cap\\
\quad\quad \texttt{Camera}_{\texttt{distortion}}[c; \texttt{distortion} < 0.05; \text{TID}])\\
\quad\quad \cap\\
\quad\quad\quad \texttt{Camera}_{\texttt{noise}}[c; \texttt{noise} < 0.01; \text{TID}])\\
\quad\quad\quad \cap\\
\quad\quad\quad\quad \texttt{Camera}_{\texttt{zoomMin}}[c; \texttt{zoomMin} < 35; \text{TID}])\\
\quad\quad\quad\quad \cap\\
\quad\quad\quad\quad\quad \texttt{Camera}_{\texttt{zoomMax}}[c; \texttt{zoomMax} > 105; \text{TID}])
$$

Then dereference
(Notion: index and-ing/and merge) (bitmap index)

4.13 Multi Index Access Path

Questions:

- In which order do we intersect the TID sets resulting from the index scans?
- Do we really apply all indexes before dereferencing the TIDs?

The answer to the latter question is clearly *"no"*, if the next index scan is more expensive than accessing the records in the current TID list.
It can be shown that the indexes in the cascade of intersections are ordered on increasing $(f_i - 1)/c_i$ terms where f_i is the selectivity of the index and c_i its access cost.
Further, we can stop as soon as accessing the original tuples in the base relation becomes cheaper than intersecting with another index and subsequently accessing the base relation.

4.13 Multi Index Access Path

Index-oring (or merge):

select *
from Emp
where yearsOfEmployment $\geq$ 30
or age $\geq$ 65

Translation:

$\texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} \geq 30; \mathrm{TID}]$
$\cup\ \texttt{Emp}_{\texttt{age}}[c; \texttt{age} \geq 65; \mathrm{TID}]$

Attention: duplicates
Optimal translation of complex boolean expressions?
Factorization?

4.13 Multi Index Access Path

Index differencing:

select *
from Emp
where yearsOfEmployment $\neq$ 10
and age $\geq$ 65

Translation:

$\texttt{Emp}_{\texttt{age}}[c; \texttt{age} \geq 65; \text{TID}]$
$\setminus$
$\texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} = 10; \text{TID}]$

4.13 Multi Index Access Path

Non-restrictive index sargable predicates (more than half of the index has to be read):

select *
from Emp
where yearsOfEmployment $\leq$ 5
and age $\leq$ 60

Then

$$\texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} \leq 5; \text{TID}]$$
$$\setminus \texttt{Emp}_{\texttt{age}}[c; \texttt{age} > 60; \text{TID}]$$

could be more efficient than

$$\texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} \leq 5; \text{TID}]$$
$$\cap \texttt{Emp}_{\texttt{age}}[c; \texttt{age} \leq 60; \text{TID}]$$

4.14 Indexes and Join

1. speed up joins by index exploitation
2. make join a general index processing operation

(intersection is similar to join (for sets))

4.14 Indexes and Join

Turn map

$$\chi_{e:*\text{TID},\mathtt{name}:\mathtt{e.name}}(\mathtt{Emp}_{\mathtt{salary}}[x; 25 \leq \mathtt{age} \leq 35; \text{TID}])$$

into d-join

$$\mathtt{Emp}_{\mathtt{salary}}[x; 25 \leq \mathtt{age} \leq 35; \text{TID}] \Join \chi_{e:*\text{TID},\mathtt{name}:\mathtt{e.name}}(\Box)$$

or even join

$$\mathtt{Emp}_{\mathtt{salary}}[x; 25 \leq \mathtt{age} \leq 35] \Join_{x.\text{TID}=e.\text{TID}} \mathtt{Emp}[e]$$

Variants: sorting at different places (by plan generator)

- pro: flexibility
- contra: large search space

4.14 Indexes and Join

Query:

select name,age
from Person
where name like 'R%' and age between 40 and 50

Translation:

$$\Pi_{\texttt{name,age}}(\\ \quad \texttt{Emp}_{\texttt{age}}[a; 40 \le \texttt{age} \le 50; \texttt{TIDa}, \texttt{age}] \\ \qquad \bowtie_{\texttt{TIDa=TIDn}} \\ \quad \texttt{Emp}_{\texttt{name}}[n; \texttt{name} \ge' R'; \texttt{name} \le' R'; \texttt{TIDn}, \texttt{name}])$$

4.14 Indexes and Join

select	*
from	*Emp e, Dept d*
where	*e.name = ‘Maier’ and e.dno = d.dno*

The query
can be directly translated to

$$\sigma_{e.\mathtt{name}=\text{‘}\mathtt{Maier}\text{’}}(\mathtt{Emp}[e]) \bowtie_{e.\mathtt{dno}=d.\mathtt{dno}} \mathtt{Dept}[d]$$

4.14 Indexes and Join

If there are indexes on `Emp.name` and `Dept.dno`, we can replace $\sigma_{e.\mathtt{name}=\text{`}\mathtt{Maier}\text{'}}(\mathtt{Emp}[e])$ by an index scan as we have seen previously:

$$\chi_{e:*(x.TID),\mathcal{A}(\mathtt{Emp}):e.*}(\mathtt{Emp}_{\mathtt{name}}[x;\mathtt{name}=\text{`}\mathtt{Maier}\text{'}])$$

4.14 Indexes and Join

With a d-join:

$$\texttt{Emp}_{\texttt{name}}[x; \texttt{name} = \text{`}\texttt{Maier}\text{'}] \bowtie \chi_{t:*(x.TID),\mathcal{A}(e)t.*}(\Box)$$

Abbreviate $\texttt{Emp}_{\texttt{name}}[x; \texttt{name} = \text{`}\texttt{Maier}\text{'}]$ by E_i
Abbreviate $\chi_{t:*(x.TID),\mathcal{A}(e)t.*}(\Box)$ by E_a.

4.14 Indexes and Join

Use index on `Dept.dno`:

$$E_i \bowtie E_a \bowtie \mathtt{Dept}_{\mathtt{dno}}[y; y.dno = dno]$$

Dereference TIDs (*index nested loop join*):

$$\begin{aligned}&E_i \bowtie E_a\\ &\qquad \bowtie \mathtt{Dept}_{\mathtt{dno}}[y; y.\mathtt{dno} = \mathtt{dno}; \mathtt{dTID} : y.\mathrm{TID}]\\ &\qquad\quad \bowtie \chi_{u:*\mathtt{dTID},\mathcal{A}(\mathtt{Dept})u.*}(\Box)\end{aligned}$$

Abbreviate $\mathtt{Dept}_{\mathtt{dno}}[y; y.\mathtt{dno} = \mathtt{dno}; \mathtt{dTID} : y.\mathrm{TID}]$ by D_i
Abbreviate $\chi_{u:*\mathtt{dTID},\mathcal{A}(\mathtt{Dept})u.*}(\Box)$ by D_a
Fully abbreviated, the expression then becomes

$$E_i \bowtie E_a \bowtie D_i \bowtie D_a$$

4.14 Indexes and Join

Optimizations: sorting the *outer* of a d-join is useful under several circumstances since it may

- turn random I/O into sequential I/O and/or
- avoid reading the same page twice.

In our example expression:

- We can sort the result of expression E_i on `TID` in order to turn random I/O into sequential I/O, if there are many employees named “Maier”.
- We can sort the result of the expression $E_i \ltimes E_a$ on `dno` for two reasons:
  - If there are duplicates for `dno`, i.e. there are many employees named “Maier” in each department, then this guarantees that no index page (of the index `Dept.dno`) has to be read more than once.
  - If additionally `Dept.dno` is a clustered index or `Dept` is an index-only table contained in `Dept.dno` then large parts of the random I/O can be turned into sequential I/O.
  - If the result of the inner is materialized (see below), then only one result needs to be stored. Note that sorting is not necessary but grouping would suffice to avoid duplicate work.
- We can sort the result of the expression $E_i \ltimes E_a \ltimes D_i$ on `dTID` for the same reasons as mentioned above for sorting the result of E_i on `TID`.

4.14 Indexes and Join: Temping the Inner

Typically, many employees will work in a single department and possibly several of them are called “Maier”.
For everyone of them, we can be sure that there exists at most one department.
Let us assume that referential intregrity has been specified.
Then there exists exactly one department for every employee.
We have to find a way to rewrite the expression

$$E_i \bowtie E_a \bowtie \texttt{Dept}_{\texttt{dno}}[y; y.\texttt{dno} = \texttt{dno}; \texttt{dTID} : y.\text{TID}]$$

such that the mapping $\texttt{dno} \longrightarrow \texttt{dTID}$ is explicitly materialized (or, as one could also say, *cached*).

4.14 Indexes and Join: Temping the Inner

Use $\chi^{\mathtt{mat}}$:

$$E_i \bowtie E_a \bowtie \chi^{\mathtt{mat}}_{dTID:(\mathtt{IdxAcc}^{Dept}_{dno}[y;y.\mathtt{dno}=\mathtt{dno}]).\mathrm{TID}}(\square)$$

4.14 Indexes and Joins: Temping the Inner

If we further assume that the outer ($E_i \bowtie E_a$) is sorted on `dno`, then it suffices to remember only the TID for the latest `dno`.
We define the map operator $\chi^{\mathtt{mat},1}$ to do exactly this.
A more efficient plan could thus be

$$\text{Sort}_{\mathtt{dno}}(E_i \bowtie E_a) \bowtie \chi^{\mathtt{mat},1}_{dTID:(\mathtt{IdxAcc}^{Dept}_{dno}[y;y.\mathtt{dno}=\mathtt{dno}]).\text{TID}}(\Box)$$

where, strictly speaking, sorting is not necessary: grouping would suffice.

4.14 Indexes and Joins: Temping the Inner

Consider: $e_1 \bowtie e_2$
The free variables used in e_2 must be a subset of the variables (attributes) produced by e_1, i.e. $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$.
Even if e_1 does not contain duplicates, the projection of e_1 on $\mathcal{F}(e_2)$ may contain duplicates.
If so, materialization could pay off.
However, in general, for every binding of the variables $\mathcal{F}(e_2)$, the expression e_2 may produce several tuples.
This means that using $\chi^{\texttt{mat}}$ is not sufficient.

4.14 Indexes and Joins: Temping the Inner

The query

select *
from Emp e, Wine w
where e.yearOfBirth = w.year

has the usual suspects as plans.
Assume we have only wines from a few years.
Then, it might make sense to consider the following alternative:

$$\texttt{Wine}[w] \ltimes \sigma_{e.\texttt{yearOfBirth}=w.\texttt{year}}(\texttt{Emp}[e])$$

Problem: scan `Emp` once for each `Wine` tuple
Duplicates in `Wine.year`: scan `Emp` only once per `Wine.year` value

4.14 Indexes and Joins: Temping the Inner

The memox operator performs caching:

$$\texttt{Wine}[w] \ltimes \mathfrak{M}(\sigma_{\texttt{e.yearOfBirth=w.year}}(\texttt{Emp}[e]))$$

Sorting still beneficial:

$$\text{Sort}_{w.\texttt{yearOfBirth}}(\texttt{Wine}[w]) \ltimes \mathfrak{M}^1(\sigma_{\texttt{e.yearOfBirth=w.year}}(\texttt{Emp}[e]))$$

4.14 Indexes and Join

Things can become even more efficient if there is an index on `Emp.yearOfBirth`:

$$\text{Sort}_{\texttt{w.yearOfBirth}}(\texttt{Wine}[\texttt{w}])$$
$$\quad \ltimes(\mathfrak{M}^1(\texttt{Emp}_{\texttt{yearOfBirth}}[\texttt{x}; \texttt{x.yearOfBirth} = \texttt{w.year}]$$
$$\qquad \ltimes(\chi_{e:*(x.\text{TID}),\mathcal{A}(\texttt{Emp}):*e}(\Box))))$$

4.14 Indexes and Join

Indexes on `Emp.yearOfBirth` and `Wine.year`.
Join result of index scans.
Since the index scan produces its output ordered on the key attributes, a simple merge join suffices (and we are back at the latter):

$$\mathtt{Emp}_{\mathtt{yearOfBirth}}[x] \bowtie^{\mathtt{merge}}_{x.\mathtt{yearOfBirth}=y.\mathtt{year}} \mathtt{Wine}_{\mathtt{year}}[y]$$

4.15 Remarks on Access Path Generation

Side-ways information passing

Consider $R \bowtie_{R.a=S.b} S$

- min/max for restriction on other join argument
- full projection on join attributes (leads to semi-join)
- bitmap representation of the projection

Cardinalities and Costs

Given: number of TIDs to dereference
Question: disk access costs?
Two step solution:

1. estimate number of pages to be accessed
2. estimate costs for accessing these pages

4.16 Counting the Number of Accesses

Given a set of k TIDs after an index access:

How many pages do we have to access to dereference them?

Let R be the relation for which we have to retrieve the tuples. Then we use the following abbreviations

N	$\lvert R\rvert$	number of tuples in the relation R
m	$\lvert\lvert R\rvert\rvert$	number of pages on which tuples of R are stored
B	N/m	number of tuples per page
k		number of (distinct) TIDs for which tuples have to be retrieved

We assume that the tuples are uniformely distributed among the m pages.

Then, each page stores $B = N/m$ tuples. B is called *blocking factor*.

4.16 Counting the Number of Accesses

Let us consider some border cases.
If $k > N - N/m$ or $m = 1$, then all pages are accessed.
If $k = 1$ then exactly one page is accessed.

4.16 Counting the Number of Accesses

The answer to the general question will be expressed in terms of

- *buckets* (pages in the above case) and
- *items* contained therein (tuples in the above case).

Later on, we will also use extents, cylinders, or tracks as buckets and tracks or sectors/blocks as items.

4.16 Counting the Number of Accesses

Outline:

1. random/direct access
   1.1 items uniformly distributed among the buckets
      1.1.1 request k distinct items
      1.1.2 request k non-distinct items
   1.2 non-uniform distribution of items among buckets
2. sequential access

Always: uniform access probability

4.16 Direct, Uniform, Distinct

Additional assumption:
The probability that we request a set with k items is

$$\frac{1}{\binom{N}{k}}$$

for all of the

$$\binom{N}{k}$$

possibilities to select a k-set.
[Every k-set is accessed with the same probability.]

4.16 Direct, Uniform, Distinct

Theorem [Waters/Yao]
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

$$\overline{\mathcal{Y}}_n^{N,m}(k) = m * \mathcal{Y}_n^N(k) \tag{19}$$

where $\mathcal{Y}_n^N(k)$ is the probability that a bucket contains at least one item. This probability is equal to

$$\mathcal{Y}_n^N(k) = \begin{cases} [1 - p] & k \leq N - n \\ 1 & k > N - n \end{cases}$$

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

The following alternative expressions can be used to calculate p:

$$
\begin{aligned}
p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} \qquad (20)\\
&= \prod_{i=0}^{k-1} \frac{N-n-i}{N-i} \qquad (21)\\
&= \prod_{i=0}^{n-1} \frac{N-k-i}{N-i} \qquad (22)
\end{aligned}
$$

4.16 Direct, Uniform, Distinct

Proof (1): The total number of possibilities to pick the k items from all N items is

$$\binom{N}{k}$$

The number of possibilities to pick k items from all items not contained in a fixed single bucket is

$$\binom{N-n}{k}$$

Hence, the probability p that a bucket does not qualify is

$$p = \binom{N-n}{k} / \binom{N}{k}$$

4.16 Direct, Uniform, Distinct

Proof (2):

$$
\begin{aligned}
p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} \\
&= \frac{(N-n)!\;\; k!(N-k)!}{k!((N-n)-k)!\;\; N!} \\
&= \prod_{i=0}^{k-1} \frac{N-n-i}{N-i}
\end{aligned}
$$

4.16 Direct, Uniform, Distinct

Proof(3):

$$\begin{aligned} p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} \\ &= \frac{(N-n)!\;\; k!(N-k)!}{k!((N-n)-k)!\;\; N!} \\ &= \frac{(N-n)!\;\; (N-k)!}{N!\;\; ((N-k)-n)!} \\ &= \prod_{i=0}^{n-1} \frac{N-k-i}{N-i} \end{aligned}$$

4.16 Direct, Uniform, Distinct

Implementation remark:

> *The fraction $m = N/n$ may not be an integer.*
> *For these cases, it is advisable to have a Gamma-function based implementation of binomial coeffcients at hand*

Evaluation of Yao's formula is expensive. Approximations are more efficient to calculate.

4.16 Direct, Uniform, Distinct

Special cases:

If	then $\mathcal{Y}_m^N(k) =$
$n = 1$	k/N
$n = N$	1
$k = 0$	0
$k = 1$	B/N
$k = N$	1

4.16 Direct, Uniform, Distinct

Let N items be distributed over N buckets such that every bucket contains exactly one item.
Further let us be interested in a subset of m buckets ($1 \leq m \leq N$).
If we pick k items then the number of buckets within the subset of size m that qualify is

$$m\mathcal{Y}_1^N(k) = m\frac{k}{N} \tag{23}$$

qualify.
Proof:

$$\begin{aligned}
\mathcal{Y}_1^N(k) &= (1 - \frac{\binom{N-1}{k}}{\binom{N}{k}}) \\
&= (1 - \frac{\frac{(N-1)!}{k!((N-1)-k)!}}{\frac{N!}{k!(N-k)!}}) \\
&= (1 - \frac{(N-1)!k!(N-k)!}{N!k!((N-1)-k)!}) \\
&= (1 - \frac{N-k}{N}) \\
&= (\frac{N}{N} - \frac{N-k}{N}) \\
&= \frac{N-N+k}{N} \\
&= \frac{k}{N}
\end{aligned}$$

4.16 Direct, Uniform, Distinct

Approximation of Yao's formula (1):

$$p \approx (1 - k/N)^n$$

[Waters]

4.16 Direct, Uniform, Distinct

Approximation of Yao's formula (2):
$\overline{\mathcal{Y}}_n^{N,m}(k)$ can be approximated by:

$$\begin{aligned} m * [\quad & (1 - (1 - 1/m)^k) + \\ & (1/(m^2 b) * k(k-1)/2 * (1 - 1/m)^{k-1}) + \\ & (1.5/(m^3 p^4) * k(k-1)(2k-1)/6 * (1 - 1/m)^{k-1}) \quad] \end{aligned}$$

[Whang, Wiederhold, Sagalowicz]

4.16 Direct, Uniform, Distinct

Approximation of Yao's formula (3):

$$\overline{\mathcal{Y}}_n^{N,m}(k) \approx \begin{cases} k & \text{if } k < \frac{m}{2} \\ \frac{k+m}{2} & \text{if } \frac{m}{2} \leq k < 2m \\ m & \text{if } 2m \leq k \end{cases}$$

[Bernstein, Goodman, Wong, Reeve, Rothnie]

4.16 Direct, Uniform, Distinct

Upper and lower bounds for p:

$$\begin{aligned} p_{\text{lower}} &= (1 - \frac{k}{N - \frac{n-1}{2}})^{n} \\ p_{\text{upper}} &= ((1 - \frac{k}{N}) * (1 - \frac{k}{N - n + 1}))^{n/2} \end{aligned}$$

for $n = N/m$.
Dihr and Saharia claim that the maximal difference resulting from the use of the lower and the upper bound to compute the number of page accesses is 0.224—far less than a single page access.

4.16 Direct, Uniform, Non-Distinct

Lemma

Let S be a set with $|S| = N$ elements. Then, the number of multisets with cardinality k containing only elements from S is

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

Proof: For a prove we just note that there is a bijection between the k-multisets and the k-subsets of a $N+k-1$-set.
We can go from a multiset to a set by f with

$$f(\{x_1 \leq \ldots \leq x_k\}) = \{x_1 + 0 < x_2 + 1 < \ldots < x_k + (k-1)\}$$

and from a set to a multiset via g with

$$g(\{x_1 < \ldots < x_k\}) = \{x_1 - 0 < x_2 - 1 < \ldots < x_k - (k-1)\}$$

4.16 Direct, Uniform, Non-Distinct

Theorem [Cheung] Consider m buckets with n items each. Then there is a total of $N = nm$ items. If we randomly select k not necessarily distinct items from all items, then the number of qualifying buckets is

$$\overline{\text{Cheung}}_n^{N,m}(k) = m * \text{Cheung}_n^N(k) \tag{24}$$

where

$$\text{Cheung}_n^N(k) = [1 - \tilde{p}] \tag{25}$$

with the following equivalent expressions for $\tilde{p}$:

$$\tilde{p} = \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \tag{26}$$

$$= \prod_{i=0}^{k-1} \frac{N-n+i}{N+i} \tag{27}$$

$$= \prod_{i=0}^{n-1} \frac{N-1-i}{N-1+k-i} \tag{28}$$

Proof(1):
Eq. 26 follows from the observation that the probability that some bucket does not contain any of the k possibly duplicate items is $\frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}}$.

Proof(2):
Eq. 27 follows from

$$\begin{aligned}
\tilde{p} &= \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \\
&= \frac{(N-n+k-1)!\ \ k!((N+k-1)-k)!}{k!((N-n+k-1)-k)!\ \ (N+k-1)!} \\
&= \frac{(N-n-1+k)!\ \ (N-1)!}{(N-n-1)!\ \ (N-1+k)!} \\
&= \prod_{i=0}^{k-1} \frac{N-n+i}{N+i}
\end{aligned}$$

Proof(3):
Eq. 28 follows from

$$\begin{aligned}
\tilde{p} &= \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \\
&= \frac{(N-n+k-1)!\ \ k!((N+k-1)-k)!}{k!((N-n+k-1)-k)!\ \ (N+k-1)!} \\
&= \frac{(N+k-1-n)!\ \ (N-1)!}{(N+k-1)!\ \ (N-1-n)!} \\
&= \prod_{i=0}^{n-1} \frac{N-n+i}{N+k-n+i} \\
&= \prod_{i=0}^{n-1} \frac{N-1-i}{N-1+k-i}
\end{aligned}$$

4.16 Direct, Uniform, Non-Distinct

Approximation for $\tilde{p}$:

$$(1 - n/N)^k$$

[Cardenas]

4.16 Direct, Uniform, Non-Distinct

Estimate for the number of distinct values in a bag:

Corollary

Let S be a k-multiset containing elements from an N-set T. Then the number of distinct items contained in S is

$$\mathcal{D}(N, k) = \frac{Nk}{N + k - 1} \tag{29}$$

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

4.16 Direct, Uniform, Non-Distinct

Model switching:

$$\overline{\mathcal{Y}}_n^{N,m}(\mathit{Distinct}(N,k)) \approx \overline{\text{Cheung}}_n^{N,m}(k)$$

[for $n \geq 5$]

4.16 Direct, Non-Uniform, Distinct

So far:

1. every page contains the same number of records, and
2. every record is accessed with the same probability.

Now:

> *Model the distribution of items to buckets by m numbers n_i (for $1 \leq i \leq m$) if there are m buckets.*
> *Each n_i equals the number of records in some bucket i.*

4.16 Direct, Non-Uniform, Distinct

The following theorem is a simple application of Yao's formula:

Theorem (Yao/Waters/Christodoulakis)

Assume a set of m buckets. Each bucket contains $n_j > 0$ items ($1 \leq j \leq m$). The total number of items is $N = \sum_{j=1}^{m} n_j$. If we lookup k distinct items, then the probability that bucket j qualifies is

$$\mathcal{W}_{n_j}^{N}(k,j) = [1 - \frac{\binom{N-n_j}{k}}{\binom{N}{k}}] \quad (= \mathcal{Y}_{n_j}^{N}(k)) \tag{30}$$

and the expected number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) := \sum_{j=1}^{m} \mathcal{W}_{n_j}^{N}(k,j) \tag{31}$$

4.16 Direct, Non-Uniform, Distinct

The product formulation in Eq. 22 of Theorem 455 results in a more efficient computation:

Corollary

If we lookup k distinct items, then the expected number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{j=1}^{m}(1 - p_j) \tag{32}$$

with

$$p_j = \begin{cases} \prod_{i=0}^{n_j-1} \frac{N-k-i}{N-i} & k \leq n_j \\ 0 & N - n_j < k \leq N \end{cases} \tag{33}$$

4.16 Direct, Non-Uniform, Distinct

If we compute the p_j after we have sorted the n_j in ascending order, we can use the fact that

$$p_{j+1} = p_j * \prod_{i=n_j}^{n_{j+1}-1} \frac{N-k-i}{N-i}.$$

4.16 Direct, Non-Uniform, Distinct

Many buckets: statistics too big. Better: Histograms

Corollary

For $1 \leq i \leq L$ let there be l_i buckets containing n_i items. Then, the total number of buckets is $m = \sum_{i=1}^{L} l_i$ and the total number of items in all buckets is $N = \sum_{i=1}^{L} l_i n_i$. For k randomly selected items the number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{i=1}^{L} l_i \mathcal{Y}_{n_j}^{N}(k) \qquad (34)$$

4.16 Direct, Non-Uniform, Distinct

Distribution function. The probability that $x \leq n_j$ items in a bucket j qualify, can be calculated as follows:

- The number of possibilities to select x items in bucket n_j is

$$\binom{n_j}{x}$$

- The number of possibilites to draw the remaining $k - x$ items from the other buckets is

$$\binom{N - n_j}{k - x}$$

- The total number of possibilities to distributed k items over the buckets is

$$\binom{N}{k}$$

This shows the following:

4.16 Direct, Non-Uniform, Distinct

Theorem

Assume a set of m buckets. Each bucket contains $n_j > 0$ items $(1 \leq j \leq m)$. The total number of items is $N = \sum_{j=1}^{m} n_j$. If we lookup k distinct items, then the probability that x items in bucket j qualify is

$$\mathcal{X}_{n_j}^{N}(k,x) = \frac{\binom{n_j}{x} \binom{N-n_j}{k-x}}{\binom{N}{k}} \tag{35}$$

Further, the expected number of qualifying items in bucket j is

$$\overline{\mathcal{X}}_{n_j}^{N,m}(k) = \sum_{x=0}^{\min(k,n_j)} x\mathcal{X}_{n_j}^{N}(k,x) \tag{36}$$

In standard statistics books the probability distribution $\mathcal{X}_{n_j}^{N}(k,x)$ is called *hypergeometric distribution*.

4.16 Direct, Non-Uniform, Distinct

Let us consider the case where all n_j are equal to n. Then, we can calculate the average number of qualifying items in a bucket. With $y := \min(k, n)$ we have

$$\begin{aligned}
\overline{\mathcal{X}}^{N,m}_{n_j}(k) &= \sum_{x=0}^{\min(k,n)} x\mathcal{X}^{N}_{n}(k, x) \\
&= \sum_{x=1}^{\min(k,n)} x\mathcal{X}^{N}_{n}(k, x) \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} x \binom{n}{x} \binom{N-n}{k-x}
\end{aligned}$$

$$\begin{aligned}
\overline{\mathcal{X}}^{N,m}_{n_j}(k) &= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} x \binom{n}{x}\binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{x}{1}\binom{n}{x}\binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{n}{1}\binom{n-1}{x-1}\binom{N-n}{k-x} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \sum_{x=0}^{y-1} \binom{n-1}{0+x}\binom{N-n}{(k-1)-x} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{n-1+N-n}{0+k-1} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{N-1}{k-1} \\
&= n\frac{k}{N} \;=\; \frac{k}{m}
\end{aligned}$$

4.16 Direct, Non-Uniform, Distinct

Let us consider the even more special case where every bucket contains a single item. That is, $N = m$ and $n_i = 1$. The probability that a bucket contains a qualifying item reduces to

$$\begin{aligned} \mathcal{X}_1^N(k,x) &= \frac{\binom{1}{x}\ \binom{N-1}{k-1}}{\binom{N}{k}} \\ &= \frac{\binom{N-1}{k-1}}{\binom{N}{k}} \\ &= \frac{k}{N}\ \ (= \frac{k}{m}) \end{aligned}$$

Since x can then only be zero or one, the average number of qualifying items a bucket contains is also $\frac{k}{N}$.

4.16 Sequential: Vector of Bits

When estimating seek costs, we need to calculate the probability distribution for the distance between two subsequent qualifying cylinders.
We model the situation as a bitvector of length B with b bits set to one.
Then, B corresponds to the number of cylinders and a one indicates that a cylinder qualifies.
[Later: Vector of Buckets]

4.16 Sequential: Vector of Bits

Theorem

Assume a bitvector of length B. Within it b ones are uniformly distributed. The remaining $B - b$ bits are zero. Then, the probability distribution of the number j of zeros

1. *between two consecutive ones,*
2. *before the first one, and*
3. *after the last one*

is given by

$$\mathcal{B}_b^B(j) = \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \tag{37}$$

Proof: To see why the formula holds, consider the total number of bitvectors having a one in position *i* followed by *j* zeros followed by a one.
This number is

$$\binom{B-j-2}{b-2}$$

We can chose $B-j-1$ positions for *i*.
The total number of bitvectors is

$$\binom{B}{b}$$

and each bitvector has $b-1$ sequences of the form that a one is followed by a sequence of zeros is followed by a one.

Hence,

$$\begin{aligned}\mathcal{B}_b^B(j) &= \frac{(B-j-1)\binom{B-j-2}{b-2}}{(b-1)\binom{B}{b}} \\ &= \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}}\end{aligned}$$

Part (1) follows.
To prove (2), we count the number of bitvectors that start with j zeros before the first one.
There are $B-j-1$ positions left for the remaining $b-1$ ones.
Hence, the number of these bitvectors is $\binom{B-j-1}{b-1}$ and part (2) follows.
Part (3) follows by symmetry.

4.16 Sequential: Vector of Bits

We can derive a less expensive way to calculate formula for $\mathcal{B}_b^B(j)$ as follows.
For $j = 0$, we have $\mathcal{B}_b^B(0) = \frac{b}{B}$.
If $j > 0$, then

$$
\begin{aligned}
\mathcal{B}_b^B(j) &= \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \\
&= \frac{\frac{(B-j-1)!}{(b-1)!((B-j-1)-(b-1))!}}{\frac{B!}{b!(B-b)!}} \\
&= \frac{(B-j-1)! \;\; b!(B-b)!}{(b-1)!((B-j-1)-(b-1))! \;\; B!}
\end{aligned}
$$

$$\begin{aligned}
\mathcal{B}_b^B(j) &= \frac{(B-j-1)!\ \ b!(B-b)!}{(b-1)!((B-j-1)-(b-1))!\ \ B!} \\
&= b\frac{(B-j-1)!\ \ (B-b)!}{((B-j-1)-(b-1))!\ \ B!} \\
&= b\frac{(B-j-1)!\ \ (B-b)!}{(B-j-b)!\ \ B!} \\
&= \frac{b}{B-j}\,\frac{(B-j)!\ \ (B-b)!}{(B-b-j)!\ \ B!} \\
&= \frac{b}{B-j}\prod_{i=0}^{j-1}\left(1-\frac{b}{B-i}\right)
\end{aligned}$$

This formula is useful when $\mathcal{B}_b^B(j)$ occurs in sums over j.

4.16 Sequential: Vector of Bits

Corollary

Using the terminology of Theorem 0.8, the expected value for the number of zeros

1. *before the first one,*
2. *between two successive ones, and*
3. *after the last one*

is

$$\overline{\mathcal{B}}_b^B = \sum_{j=0}^{B-b} j\mathcal{B}_b^B(j) = \frac{B-b}{b+1} \tag{38}$$

Proof:

$$\begin{aligned}
\sum_{j=0}^{B-b} j\binom{B-j-1}{b-1} &= \sum_{j=0}^{B-b}(B-(B-j))\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{B-j-1}{b-1} - \sum_{j=0}^{B-b}(B-j)\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{b-1} - b\sum_{j=0}^{B-b}\binom{B-j}{b} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{j} - b\sum_{j=0}^{B-b}\binom{b+j}{b}
\end{aligned}$$

$$\begin{aligned}
\sum_{j=0}^{B-b} j\binom{B-j-1}{b-1} &= B\sum_{j=0}^{B-b}\binom{b-1+j}{j} - b\sum_{j=0}^{B-b}\binom{b+j}{b} \\
&= B\binom{(b-1)+(B-b)+1}{(b-1)+1} - b\binom{b+(B-b)+1}{b+1} \\
&= B\binom{B}{b} - b\binom{B+1}{b+1} \\
&= (B - b\frac{B+1}{b+1})\binom{B}{b}
\end{aligned}$$

With

$$\begin{aligned}
B - b\frac{B+1}{b+1} &= \frac{B(b+1)-(Bb+b)}{b+1} \\
&= \frac{B-b}{b+1}
\end{aligned}$$

the claim follows.

4.16 Sequential: Vector of Bits

Corollary

Using the terminology of Theorem 0.8, the expected total number of bits from the first bit to the last one, both included, is

$$\overline{\mathcal{B}}_{tot}(B, b) = \frac{Bb + b}{b + 1} \tag{39}$$

4.16 Sequential: Vector of Bits

Proof:
We subtract from B the average expected number of zeros between the last one and the last bit:

$$\begin{aligned} B - \frac{B-b}{b+1} &= \frac{B(b+1)}{b+1} - \frac{B-b}{b+1} \\ &= \frac{Bb + B - B + b}{b+1} \\ &= \frac{Bb + b}{b+1} \end{aligned}$$

4.16 Sequential: Vector of Bits

Corollary

Using the terminology of Theorem 0.8, the number of bits from the first one and the last one, both included, is

$$\overline{\mathcal{B}}_{\textit{1-span}}(B, b) = \frac{Bb - B + 2b}{b + 1} \tag{40}$$

Proof (alternative 1):
Subtract from B the number of zeros at the beginning and the end:

$$\begin{aligned}
\overline{\mathcal{B}}_{\text{1-span}}(B, b) &= B - 2\frac{B-b}{b+1} \\
&= \frac{Bb + B - 2B + 2b}{b+1} \\
&= \frac{Bb - B + 2b}{b+1}
\end{aligned}$$

Proof (alternative 2):
Add the number of zeros between the first and the last one and the number of ones:

$$\begin{aligned}
\overline{\mathcal{B}}_{\text{1-span}}(B, b) &= (b-1)\overline{\mathcal{B}}_b^B + b \\
&= (b-1)\frac{B-b}{b+1} + \frac{b(b+1}{b+1} \\
&= \frac{Bb - b^2 - B + b + b^2 + b}{b+1} \\
&= \frac{Bb - B + 2b}{b+1}
\end{aligned}$$

4.16 Sequential: Applications for Bitvector Model

- If we look up one record in an array of B records and we search sequentially, how many array entries do we have to examine on average if the search is successful?
- Let a file consist of B consecutive cylinders. We search for k different keys all of which occur in the file. These k keys are distributed over b different cylinders. Of course, we can stop as soon as we have found the last key. What is the expected total distance the disk head has to travel if it is placed on the first cylinder of the file at the beginning of the search?
- Assume we have an array consisting of B different entries. We sequentially go through all entries of the array until we have found all the records for b different keys. We assume that the B entries in the array and the b keys are sorted. Further all b keys occur in the array. On the average, how many comparisons do we need to find all keys?

4.16 Sequential: Vector of Buckets

Theorem (Yao)

Consider a sequence of m buckets. For $1 \leq i \leq m$, let n_i be the number of items in a bucket i. Then there is a total of $N = \sum_{i=1}^{m} n_i$ items. Let $t_i = \sum_{l=0}^{i} n_l$ be the number of items in the first i buckets. If the buckets are searched sequentially, then the probability that j buckets that have to be examined until k distinct items have been found is

$$\mathcal{C}_{n_i}^{N,m}(k,j) = \frac{\binom{t_j}{k} - \binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{41}$$

Thus, the expected number of buckets that need to be examined in order to retrieve k distinct items is

$$\overline{\mathcal{C}}_{n_i}^{N,m}(k) = \sum_{j=1}^{m} j\mathcal{C}_{n_i}^{N,m}(k,j) = m - \frac{\sum_{j=1}^{m} \binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{42}$$

4.16 Sequential: Vector of Buckets

The following theorem is very useful for deriving estimates for average sequential accesses under different models [Especially: the above theorem follows].

Theorem (Lang/Driscoll/Jou)

Consider a sequence of N items. For a batched search of k items, the expected number of accessed items is

$$A(N,k) = N - \sum_{i=1}^{N-1} Prob[Y \leq i] \tag{43}$$

where Y is a random variable for the last item in the sequence that occurs among the k items searched.

4.17 Disk Drive Costs for *N* Uniform Accesses

The goal of this section is to derive estimates for the costs (time) for retrieving N cache-missed sectors of a segment S from disk.
We assume that the N sectors are read in their physical order on disk.
This can be enforced by the DBMS, by the operating system's disk scheduling policy (SCAN policy), or by the disk drive controler.

4.17 Disk Drive Costs for *N* Uniform Accesses

Remembering the description of disk drives, the total costs can be described as

$$\mathcal{C}_{\text{disk}} = \mathcal{C}_{\text{cmd}} + \mathcal{C}_{\text{seek}} + \mathcal{C}_{\text{settle}} + \mathcal{C}_{\text{rot}} + \mathcal{C}_{\text{headswitch}} \tag{44}$$

For brevity, we omitted the parameter *N* and the parameters describing the segment and the disk drive on which the segment resides.
Subsequently, we devote a (sometimes tiny) section to each summand.
Before that, we have to calculate the number of qualifying cylinders, tracks, and sectors.
These numbers will be used later on.

4.17 Disk Costs: Number of Qualifying Cylinder

- N sectors are to be retrieved.
- We want to find the number of cylinders qualifying in extent i.
- S_{sec} denotes the total number of sectors our segment contains.
- Assume: The N sectors we want to retrieve are uniformly distributed among the S_{sec} sectors of the segment.
- $S_{\text{cpe}}(i) = L_i - F_i + 1$ denotes the number of cylinders of extent i.

4.17 Disk Costs: Number of Qualifying Cylinder

The number of qualifying cylinders in extent *i* is:

$S_{cpe}(i)$ ** (1 - Prob(a cylinder does not qualify))*

The probability that a cylinder does not qualify can be computed by deviding the total number of possibilities to chose the N sectors from sectors outside the cylinder by the total number of possibilities to chose N sectors from all S_{sec} sectors of the segment.
Hence, the number of qualifying cylinders in the considered extent is:

$$Q_c(i) = S_{\text{cpe}}(i)\mathcal{Y}^{S_{\text{sec}}}_{D_{\text{Zspc}}(i)}(N) = S_{\text{cpe}}(i)(1 - \frac{\binom{S_{\text{sec}} - D_{\text{Zspc}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \quad (45)$$

4.17 Disk Costs: Number of Qualifying Tracks

Let us also calculate the number of qualifying tracks in a partion *i*.
It can be calculated by

$$S_{\text{cpe}}(i) D_{\text{tpc}}(1 - \text{Prob}(\texttt{a track does not qualify}))$$

The probability that a track does not qualify can be computed by dividing the number of ways to pick *N* sectors from sectors not belonging to a track divided by the number of possible ways to pick *N* sectors from all sectors:

$$Q_t(i) = S_{\text{cpe}}(i) D_{\text{tpc}} \mathcal{Y}^{S_{\text{sec}}}_{D_{\text{Zspt}}(i)}(N) = S_{\text{cpe}}(i) D_{\text{tpc}}(1 - \frac{\binom{S_{\text{sec}} - D_{\text{Zspt}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \tag{46}$$

4.17 Disk Costs: Number of Qualifying Tracks

Just for fun, we calculate the number of qualifying sectors of an extent in zone i. It can be approximated by

$$Q_s(i) = S_{\text{cpe}}(i) D_{\text{Zspc}}(i) \frac{N}{S_{\text{sec}}} \tag{47}$$

Since all $S_{\text{cpe}}(i)$ cylinders are in the same zone, they have the same number of sectors per track and we could also use Waters/Yao to approximate the number of qualifying cylinders by

$$Q_c(i) = \overline{\mathcal{Y}}^{S_{\text{cpe}}(i) D_{\text{Zspc}}(S_{\text{zone}}(i)), S_{\text{cpe}}(i)}_{D_{\text{Zspc}}(S_{\text{zone}}(i))}(Q_s(i)) \tag{48}$$

If $Q_s(i)$ is not too small (e.g. > 4).

4.17 Disk Costs: Command Costs

The command costs $\mathcal{C}_{\text{cmd}}$ are easy to compute. Every read of a sector requires the execution of a command. Hence

$$\mathcal{C}_{\text{cmd}} = ND_{\text{cmd}}$$

estimates the total command costs.

4.17 Disk Costs: Seek Costs

We look at several alternatives from less to more precise models.

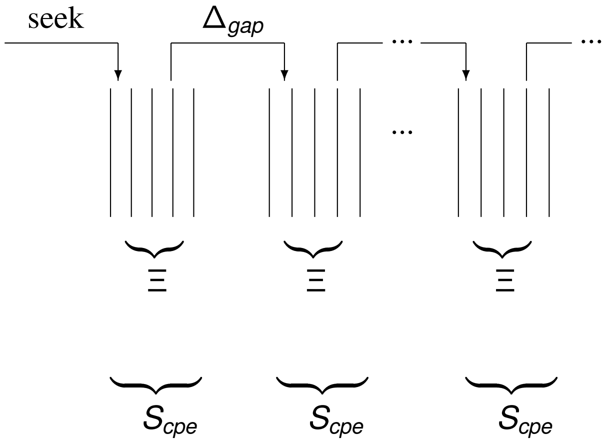
4.17 Disk Costs: Seek Costs: Upper Bound

The first cylinder we have to visit requires a random seek with cost D_{avgseek}. (Truely upper bound: $D_{\text{seek}}(D_{\text{cyl}} - 1)$)
After that, we have to visit the remaining $Q_c(i) - 1$ qualifying cylinders.
The segment spans a total of $S_{\text{last}}(S_{\text{ext}}) - S_{\text{first}}(1) + 1$ cylinders.
Let us assume that the first qualifying cylinder is the first cylinder and the last qualifying cylinder is the last cylinder of the segment.
Then, applying Qyang's Theorem 0.1 gives us the upper bound

$$\mathcal{C}_{\text{seek}}(i) \leq (Q_c(i) - 1) D_{\text{seek}}\left(\frac{S_{\text{last}}(S_{\text{ext}}) - S_{\text{first}}(1) + 1}{Q_c(i) - 1}\right)$$

after we have found the first qualifying cylinder.

4.17 Disk Costs: Seek Costs: Illustration



4.17 Disk Costs: Seek Costs

Steps:

1. Estimate for $\mathcal{C}_{\text{seekgap}}$
2. Estimates for $\mathcal{C}_{\text{seekext}}(i)$

4.17 Disk Costs: Seek Costs: Interextent Costs

The average seek cost for reaching the first qualifying cylinder is D_{avgseek}. How far within the first extent are we now? We use Corollary 0.9 to derive that the number of non-qualifying cylinders preceding the first qualifying one in some extent i is

$$\overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)} = \frac{S_{\text{cpe}}(i) - Q_c(i)}{Q_c(i) + 1}.$$

The same is found for the number of non-qualifying cylinders following the last qualifying cylinder. Hence, for every gap between the last and the first qualifying cylinder of two extents i and $i + 1$, the disk arm has to travel the distance

$$\Delta_{\text{gap}}(i) := \overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)} + S_{\text{first}}(i + 1) - S_{\text{last}}(i) - 1 + \overline{\mathcal{B}}^{S_{\text{cpe}}(i+1)}_{Q_c(i+1)}$$

Using this, we get

$$\mathcal{C}_{\text{seekgap}} = D_{\text{avgseek}} + \sum_{i=1}^{S_{\text{ext}}-1} D_{\text{seek}}(\Delta_{\text{gap}}(i))$$

4.17 Disk Costs: Seek Costs: Intraextent Costs

Let us turn to $\mathcal{C}_{\text{seekext}}(i)$. We first need the number of cylinders between the first and the last qualifying cylinder, both included, in extent i. It can be calculated using Corollary 0.11:

$$\Xi_{\text{ext}}(i) = \overline{\mathcal{B}}_{\text{1-span}}(S_{\text{cpe}}(i), Q_c(i))$$

Hence, $\Xi(i)$ is the minimal span of an extent that contains all qualifying cylinders.

4.17 Disk Costs: Seek Costs: Intraextent Costs

Using $\Xi(i)$ and Qyang's Theorem 0.1, we can derive an upper bound for $\mathcal{C}_{\text{seekext}}(i)$:

$$\mathcal{C}_{\text{seekext}}(i) \leq (Q_c(i) - 1) D_{\text{seek}}(\frac{\Xi(i)}{Q_c(i) - 1}) \tag{49}$$

Alternatively, we could formulate this as

$$\mathcal{C}_{\text{seekext}}(i) \leq (Q_c(i) - 1) D_{\text{seek}}(\overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)}) \tag{50}$$

by applying Corollary 0.9.

4.17 Disk Costs: Seek Costs: Intraextent Costs

A seemingly more precise estimate for the expected seek cost within the qualifying cylinders of an extent is derived by using Theorem 0.8:

$$\mathcal{C}_{\text{seekext}}(i) = Q_c(i) \sum_{j=0}^{S_{\text{cpe}}(i)-Q_c(i)} D_{\text{seek}}(j+1)\mathcal{B}_{Q_c(i)}^{S_{\text{cpe}}(i)}(j) \qquad (51)$$

4.17 Disk Costs: Settle Costs

The average settle cost is easy to calculate. For every qualifying cylinder, one head settlement takes place:

$$\mathcal{C}_{\text{settle}}(i) = Q_c(i) D_{\text{rdsettle}} \tag{52}$$

4.17 Disk Costs: Rotational Delay

Let us turn to the rotational delay.
For some given track in zone i,
we want to read the $Q_t(i)$ qualifying sectors contained in it.
On average, we would expect that the read head is ready to start reading in the middle of some sector of a track.
If so, we have to wait for $\frac{1}{2} D_{\text{Zscan}}(S_{\text{zone}}(i))$ before the first whole sector ocurs under the read head.
However, due to track and cylinder skew, this event does not occur after a head switch or a cylinder switch.
Instead of being overly precise here, we igore this half sector pass by time and assume we are always at the beginning of a sector.
This is also justified by the fact that we model the head switch time explicitly.

4.17 Disk Costs: Rotational Delay

Assume that the head is ready to read at the beginning of some sector of some track.
Then, in front of us is a — cyclic, which does not matter — bitvector of qualifying and non-qualifying sectors.
We can use Corollary 0.10 to estimate the total number of qualifying and non-qualifying sectors that have to pass under the head until all qualifying sectors have been seen.
The total rotational delay for the tracks of zone *i* is

$$\mathcal{C}_{\mathrm{rot}}(i) = Q_t(i) \;\; D_{\mathrm{Zscan}}(S_{\mathrm{zone}}(i)) \;\; \overline{\mathcal{B}}_{\mathrm{tot}}(D_{\mathrm{Zspt}}(S_{\mathrm{zone}}(i)), Q_{\mathrm{spt}}(i))$$

where $Q_{\mathrm{spt}}(i) = \overline{\mathcal{W}}_1^{S_{\mathrm{sec}}, D_{\mathrm{Zspt}}(S_{\mathrm{zone}}(i))}(N) = D_{\mathrm{Zspt}}(S_{\mathrm{zone}}(i))\frac{N}{S_{\mathrm{sec}}}$ is the expected number of qualifying sectors per track in extent *i*.
In case $Q_{\mathrm{spt}}(i) < 1$, we set $Q_{\mathrm{spt}}(i) := 1$.

4.17 Disk Costs: Rotational Delay

A more precise model is derived as follows.
We sum up for all j the product of (1) the probability that j sectors in a track qualify and (2) the average number of sectors that have to be read if j sectors qualify.
This gives us the number of sectors that have to pass the head in order to read all qualifying sectors.
We only need to multiply this number by the time to scan a single sector and the number of qualifying tracks.
We can estimate (1) using Theorem 0.7. For (2) we again use Corollary 0.10.

$$
\begin{aligned}
\mathcal{C}_{\text{rot}}(i) &= S_{\text{cpe}}(i)\ D_{\text{tpc}}\ D_{\text{Zscan}}(S_{\text{zone}}(i)) \\
&\quad * \sum_{j=1}^{\min(N, D_{\text{Zspt}}(S_{\text{zone}}(i)))} \mathcal{X}^{S_{\text{sec}}}_{D_{\text{Zspt}}(S_{\text{zone}}(i))}(N, j)\ \overline{\mathcal{B}}_{\text{tot}}(D_{\text{Zspt}}(S_{\text{zone}}(i)), j)
\end{aligned}
$$

4.17 Disk Costs: Rotational Delay

Yet another approach:
Split the total rotational delay into two components:

1. $\mathcal{C}_{\text{rotpass}}(i)$ measures the time needed to skip unqualifying sectors
2. $\mathcal{C}_{\text{rotread}}(i)$ that for scanning the qualifying sectors

Then

$$\mathcal{C}_{\text{rot}} = \sum_{i=1}^{S_{\text{ext}}} \mathcal{C}_{\text{rotpass}}(i) + \mathcal{C}_{\text{rotread}}(i)$$

where the total transfer cost of the qualifying sectors can be estimated as

$$\mathcal{C}_{\text{rotread}}(i) = Q_s(i) \;\; D_{\text{Zscan}}(S_{\text{zone}}(i))$$

4.17 Disk Costs: Rotational Delay

Let us treat the first component ($\mathcal{C}_{\text{rotpass}}(i)$).
Assume that j sectors of a track in extent i qualify.
The expected position on a track where the head is ready to read is the middle between two qualifying sectors.
Since the expected number of sectors between two qualifying sectors is $D_{\text{Zspt}}(S_{\text{zone}}(i))/j$, the expected number of sectors scanned before the first qualifying sector comes under the head is

$$\frac{D_{\text{Zspt}}(S_{\text{zone}}(i))}{2j}$$

The expected positions of *j* qualifying sectors on the same track is such that the number non-qualifying sectors between two successively qualifying sectors is the same.
Hence, after having read a qualifying sector $\frac{D_{\text{Zspt}}(S_{\text{zone}}(i))}{j}$ unqualifying sectors must be passed until the next qualifying sector shows up.
The total number of unqualifying sectors to be passed if *j* sectors qualify in a track of zone *i* is

$$N_s(j,i) = \frac{D_{\text{Zspt}}(S_{\text{zone}}(i))}{2j} + (j-1)\frac{D_{\text{Zspt}}(S_{\text{zone}}(i)) - j}{j}$$

Using again Theorem 0.7, the expected rotational delay for the unqualifying sectors then is

$$\begin{aligned}\mathcal{C}_{\text{rotpass}}(i) &= S_{\text{cpe}}(i)\ \ D_{\text{tpc}}\ \ D_{\text{Zscan}}(S_{\text{zone}}(i)) \\ &\quad * \sum_{j=1}^{\min(N, D_{\text{Zspt}}(S_{\text{zone}}(i)))} \mathcal{X}^{S_{\text{sec}}}_{D_{\text{Zspt}}(S_{\text{zone}}(i))}(N, j) N_s(j, i)\end{aligned}$$

4.17 Disk Costs: Head Switch Costs

The average head switch cost is equal to the average number of head switches that occur times the average head switch cost. The average number of head switch is equal to the number of tracks that qualify minus the number of cylinders that qualify since a head switch does not occur for the first track of each cylinder.
Summarizing

$$\mathcal{C}_{\text{headswitch}} = \sum_{i=1}^{S_{\text{ext}}} (Q_t(i) - Q_c(i)) \;\; D_{\text{hdswitch}} \tag{53}$$

where Q_t is the average number of tracks qualifying in an extent.

4.17 Disk Costs: Discussion

We neglected many problems in our disk access model:

- partially filled cylinders,
- pages larger than a block,
- disk drive's cache,
- remapping of bad blocks,
- non-uniformly distributed accesses,
- clusteredness,
- and so on.

Whereas the first two items are easy to fix, the rest is not so easy.