Query Optimization

Thomas Neumann

Overview

1. Introduction

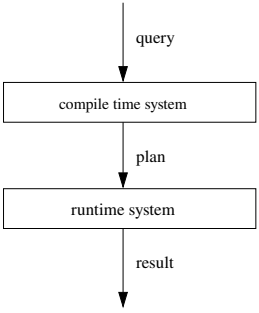
- Overview Query Processing
- Overview Query Optimization
- Overview Query Execution

Reason for Query Optimization

- query languages like SQL are declarative
- query specifies the result, not the exact computation
- multiple alternatives are common
- often vastly different runtime characteristics
- alternatives are the basis of query optimization

Note: Deciding which alternative to choose is not trivial

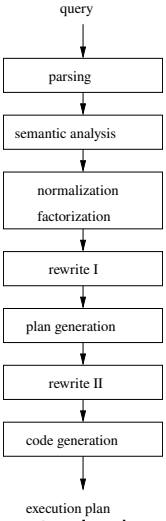
Overview Query Processing



- input: query as text
- compile time system compiles and optimizes the query
- intermediate: query as exact execution plan
- runtime system executes the query
- output: query result

separation can be very strong (embedded SQL/prepared queries etc.)

Overview Compile Time System



1. parsing, AST production
2. schema lookup, variable binding, type inference
3. normalization, factorization, constant folding etc.
4. view resolution, unnesting, deriving predicates etc.
5. constructing the execution plan
6. refining the plan, pushing group by etc.
7. producing the imperative plan

rewrite I, plan generation, and rewrite II form the query optimizer

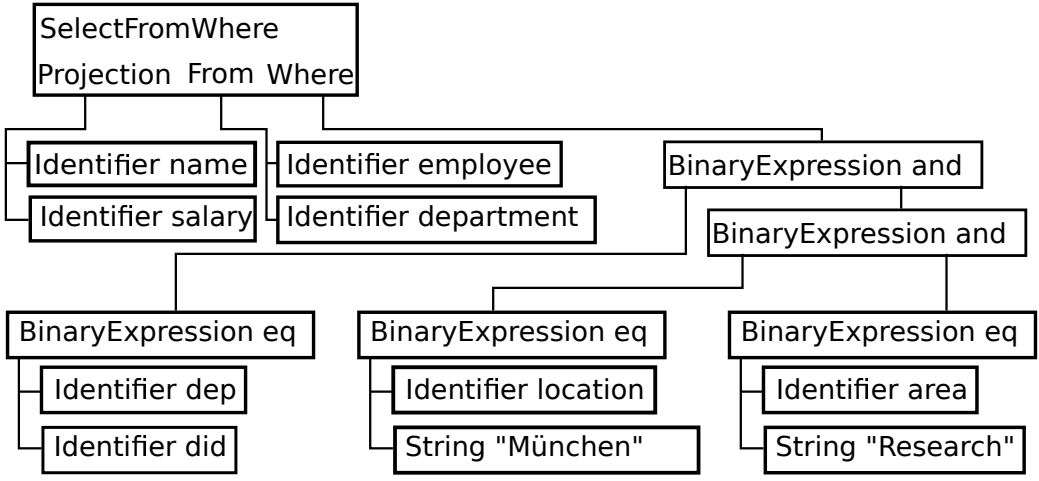
Processing Example - Input

```
select name, salary
from   employee, department
where dep=did
and    location="München"
and    area="Research"
```

Note: example is so simple that it can be presented completely, but does not allow for many optimizations. More interesting (but more abstract) examples later on.

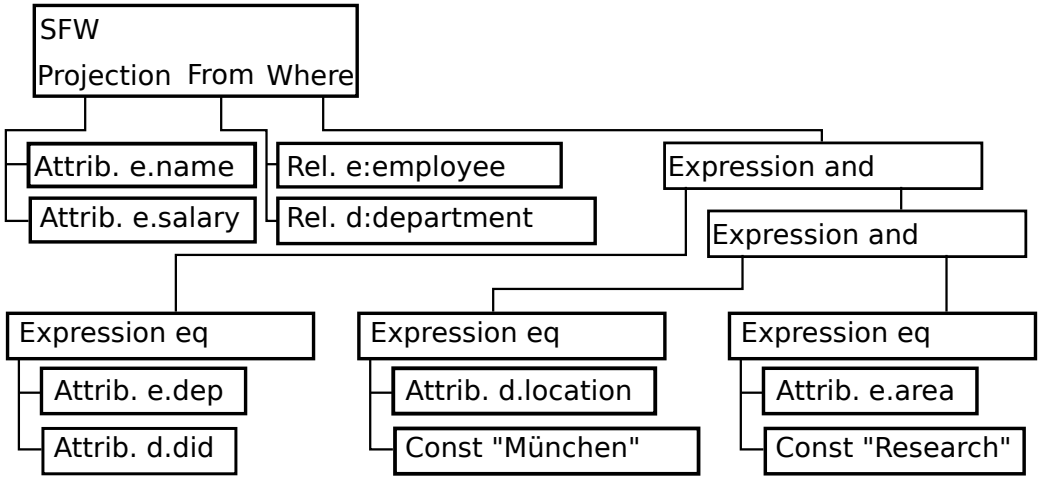
Processing Example - Parsing

Constructs an AST from the input



Processing Example - Semantic Analysis

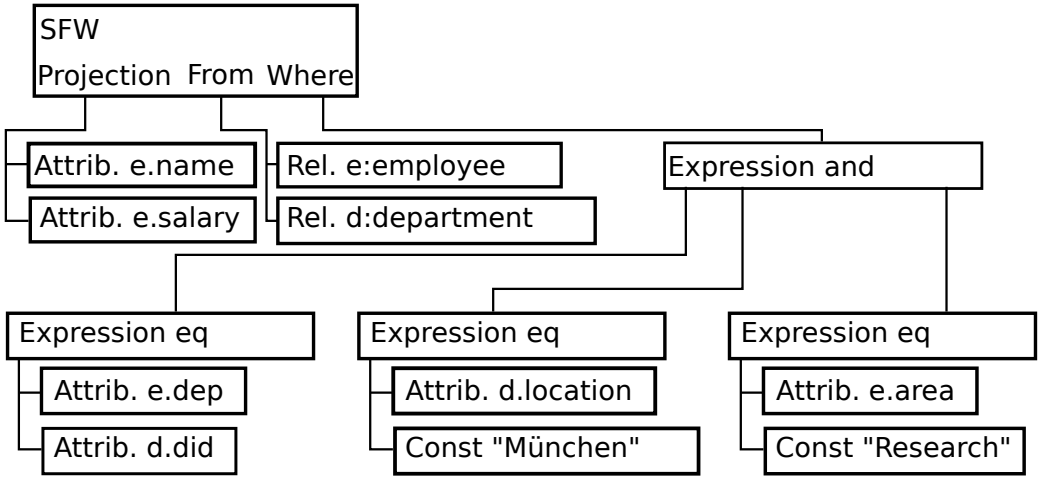
Resolves all variable binding, infers the types and checks semantics



Types omitted here, result is $bag < string, number >$

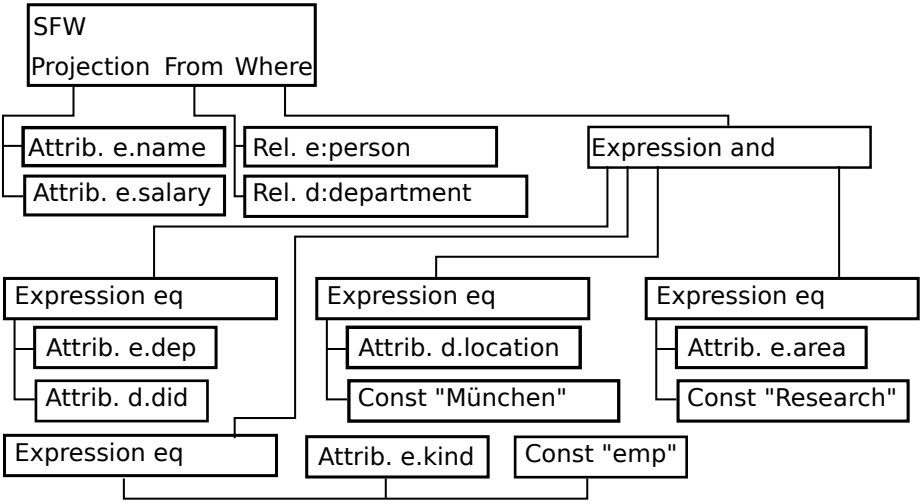
Processing Example - Normalization

Normalizes the representation, factorizes common expressions, folds constant expressions



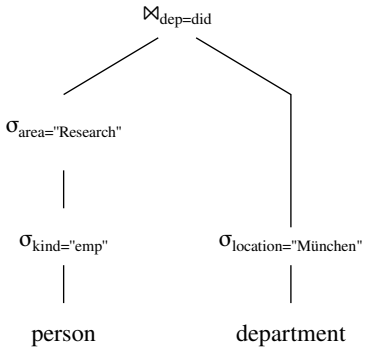
Processing Example - Rewrite I

resolves views, unnests nested expressions, expensive optimizations



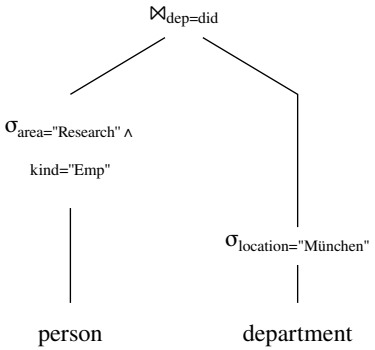
Processing Example - Plan Generation

Finds the best execution strategy, constructs a physical plan



Processing Example - Rewrite II

Polishes the plan



Processing Example - Code Generation

Produces the executable plan

```
<
   @c1 string 0
   @c2 string 0
   @c3 string 0
   @kind string 0
   @name string 0
   @salary float64
   @dep int32
   @area string 0
   @did int32
   @location string 0
   @t1 uint32 local
   @t2 string 0 local
   @t3 bool local
>
[main
   load_string "emp" @c1
   load_string "M\u00fcnchen" @c2
   load_string "Research" @c3
   first_notnull_bool
   <#1 BlockwiseNestedLoopJoin
      memSize 1048576
      [combiner
         unpack_int32 @dep
         eq_int32 @dep @did @t3
         return_if_ne_bool @t3
         unpack_string @name
         unpack_float64 @salary
      ]
[storer
   check_pack 4
   pack_int32 @dep
   pack_string @name
   check_pack 8
   pack_float64 @salary
   load_uint32 0 @t1
   hash_int32 @dep @t1 @t1
   return_uint32 @t1
]
[hasher
   load_uint32 0 @t1
   hash_int32 @did @t1 @t1
   return_uint32 @t1
]
<#2 Tablescan
   segment 1 0 4
   [loader
      unpack_string @kind
      unpack_string @name
      unpack_float64 @salary
      unpack_int32 @dep
      unpack_string @area
      eq_string @kind @c1 @t3
      return_if_ne_bool @t3
      eq_string @area @c3 @t3
      return_if_ne_bool @t3
   ]
>
      <#3 Tablescan
         segment 1 0 5
         [loader
            unpack_int32 @did
            unpack_string @location
            eq_string @location @c2 @t3
            return_if_ne_bool @t3
         ]
      >
   > @t3
   jf_bool 6 @t3
   print_string 0 @name
   cast_float64_string @salary @t2
   print_string 10 @t2
   println
   next_notnull_bool #1 @t3
   jt_bool -6 @t3
]
```

What to Optimize?

Different optimization goals reasonable:

- minimize response time
- minimize resource consumption
- minimize time to first tuple
- maximize throughput

Expressed during optimization as cost function. Common choice: Minimize response time within given resource limitations.

Basic Goal of Algebraic Optimization

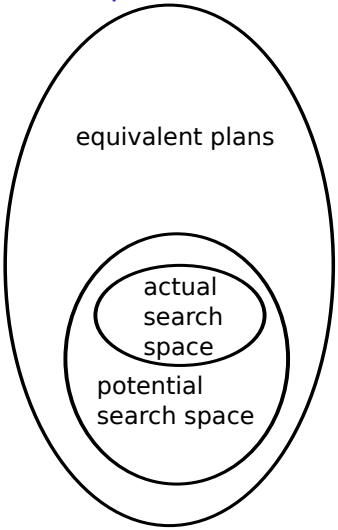
When given an algebraic expression:

- find a cheaper/the cheapest expression that is equivalent to the first one

Problems:

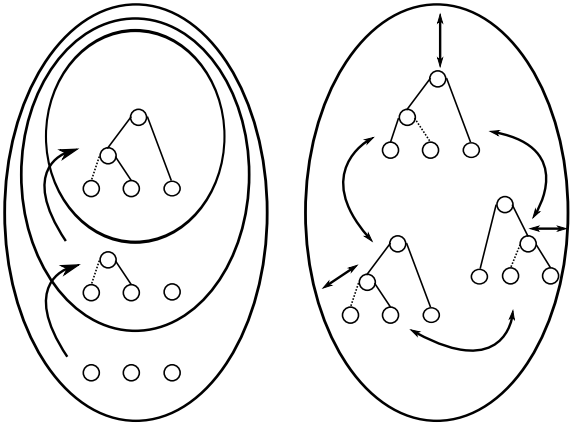
- the set of possible expressions is huge
- testing for equivalence is difficult/impossible in general
- the query is given in a calculus and not an algebra (this is also an advantage, though)
- even "simpler" optimization problems (e.g. join ordering) are typically NP hard in general

Search Space



Query optimizers only search the "optimal" solution within the limited space created by known optimiza-

Optimization Approaches



transformative is simpler, but finding the optimal solution is hard

Query Execution

Understanding query execution is important to understand query optimization

- queries executed using a physical algebra
- operators perform certain specialized operations
- generic, flexible components
- simple base: relational algebra (set oriented)
- in reality: bags, or rather data streams
- each operator produces a tuple stream, consumes streams
- tuple stream model works well, also for OODBMS, XML etc.

Relational Algebra

Notation:

- $\mathcal{A}(e)$ attributes of the tuples produces by e
- $\mathcal{F}(e)$ free variables of the expression e
- binary operators $e_1 \theta e_2$ usually require $\mathcal{A}(e_1) = \mathcal{A}(e_2)$

$e_1 \cup e_2$	union, $\{x \mid x \in e_1 \vee x \in e_2\}$
$e_1 \cap e_2$	intersection, $\{x \mid x \in e_1 \wedge x \in e_2\}$
$e_1 \setminus e_2$	difference, $\{x \mid x \in e_1 \wedge x \notin e_2\}$
$\rho_{a \to b}(e)$	rename, $\{x \circ (b : x.a) \setminus (a : x.a) \mid x \in e\}$
$\Pi_A(e)$	projection, $\{\circ_{a \in A}(a : x.a) \mid x \in e\}$
$e_1 \times e_2$	product, $\{x \circ y \mid x \in e_1 \wedge y \in e_2\}$
$\sigma_p(e)$	selection, $\{x \mid x \in e \wedge p(x)\}$
$e_1 \bowtie_p e_2$	join, $\{x \circ y \mid x \in e_1 \wedge y \in e_2 \wedge p(x \circ y)\}$

per definition set oriented. Similar operators also used bag oriented (no implicit duplicate removal).

Relational Algebra - Derived Operators

Additional (derived) operators are often useful:

- $e_1 \bowtie e_2$ natural join, $\{x \circ y_{|\mathcal{A}(e_2)\setminus\mathcal{A}(e_1)} | x \in e_1 \wedge y \in e_2 \wedge x =_{|\mathcal{A}(e_1)\cap\mathcal{A}(e_2)} y\}$
- $e_1 \div e_2$ division, $\{x_{|\mathcal{A}(e_1)\setminus\mathcal{A}(e_2)} | x \in e_1 \wedge \forall y \in e_2 \exists z \in e_1 : y =_{|\mathcal{A}(e_2)} z \wedge x =_{|\mathcal{A}(e_1)\setminus\mathcal{A}(e_2)} z\}$
- $e_1 \ltimes_p e_2$ semi-join, $\{x | x \in e_1 \wedge \exists y \in e_2 : p(x \circ y)\}$
- $e_1 \triangleright_p e_2$ anti-join, $\{x | x \in e_1 \wedge \nexists y \in e_2 : p(x \circ y)\}$
- $e_1 \bowtie\!\!\!\!\circ_p e_2$ outer-join, $(e_1 \bowtie_p e_2) \cup \{x \circ \circ_{a \in \mathcal{A}(e_2)}(a : null) | x \in (e_1 \triangleright_p e_2)\}$
- $e_1 \bowtie\!\!\!\!\circ\!\!\!\!\circ_p e_2$ full outer-join, $(e_1 \bowtie\!\!\!\!\circ_p e_2) \cup (e_2 \bowtie\!\!\!\!\circ_p e_1)$

Relational Algebra - Extensions

The algebra needs some extensions for real queries:

- map/function evaluation
  $\chi_{a:f}(e) = \{x \circ (a : f(x)) | x \in e\}$
- group by/aggregation
  $\Gamma_{A;a:f}(e) = \{x \circ (a : f(y)) | x \in \Pi_A(e) \wedge y = \{z | z \in e \wedge \forall a \in A : x.a = z.a\}\}$
- dependent join (djoin). Requires $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$
  $e_1 \bowtie_p e_2 = \{x \circ y | x \in e_1 \wedge y \in e_2(x) \wedge p(x \circ y)\}$

Physical Algebra

- relational algebra does not imply an implementation
- the implementation can have a great impact
- therefore more detailed operators (next slides)
- additional operators needed due to stream nature

Physical Algebra - Enforcer

Some operators do not effect the (logical) result but guarantee desired properties:

- sort
  Sorts the input stream according to a sort criteria
- temp
  Materializes the input stream, makes further reads cheap
- ship
  Sends the input stream to a different host (distributed databases)

Physical Algebra - Joins

Different join implementations have different characteristics:

- $e_1 \bowtie^{NL} e_2$ Nested Loop Join
  Reads all of e_2 for every tuple of e_1. Very slow, but supports all kinds of predicates
- $e_1 \bowtie^{BNL} e_2$ Blockwise Nested Loop Join
  Reads chunks of e_1 into memory and reads e_2 once for each chunk. Much faster, but requires memory. Further improvement: Use hashing for equi-joins.
- $e_1 \bowtie^{SM} e_2$ Sort Merge Join
  Scans e_1 and e_2 only once, but requires suitable sorted input. Equi-joins only.
- $e_1 \bowtie^{HH} e_2$ Hybrid-Hash Join
  Partitions e_1 and e_2 into partitions that can be joined in memory. Equi-joins only.

Physical Algebra - Aggregation

Other operators also have different implementations:

- Γ^{SI} Aggregation Sorted Input
  Aggregates the input directly. Trivial and fast, but requires sorted input
- Γ^{QS} Aggregation Quick Sort
  Sorts chunks of input with quick sort, merges sorts
- Γ^{HS} Aggregation Heap Sort
  Like Γ^{QS}. Slower sort, but longer runs
- Γ^{HH} Aggregation Hybrid Hash
  Partitions like a hybrid hash join.

Even more variants with early aggregation etc. Similar for other operators.

Physical Algebra - Summary

- logical algebras describe only the general approach
- physical algebra fixes the exact execution including runtime characteristics
- multiple physical operators possible for a single logical operator
- query optimizer must produce physical algebra
- operator selection is a crucial step during optimization

2. Textbook Query Optimization

- Algebra Revisited
- Canonical Query Translation
- Logical Query Optimization
- Physical Query Optimization

Algebra Revisited

The algebra needs some more thought:

- correctness is critical for query optimization
- can only be guaranteed by a formal model
- the algebra description in the introduction was too cursory

What we ultimately want to do with an algebraic model:

- decide if two algebraic expressions are equivalent (produce the same result)

This is too difficult in practice (not computable in general), so we at least want to:

- guarantee that two algebraic expressions are equivalent (for some classes of expressions)

This still requires a strong formal model. We accept false negatives, but not false positives.

Tuples

Tuple:

- a (unordered) mapping from attribute names to values of a domain
- sample: [name: "Sokrates", age: 69]

Schema:

- a set of attributes with domain, written $\mathcal{A}(t)$
- sample: {(name,string),(age, number)}

Note:

- simplified notation on the slides, but has to be kept in mind
- domain usually omitted when not relevant
- attribute names omitted when schema known

Tuple Concatenation

- notation: $t_1 \circ t_2$
- sample: [name: "Sokrates", age: 69]$\circ$[country: "Greece"]
  $=$ [name: "Sokrates", age: 69, country: "Greece"]
- note: $t_1 \circ t_2 = t_2 \circ t_1$, tuples are unordered

Requirements/Effects:

- $\mathcal{A}(t_1) \cap \mathcal{A}(t_2) = \emptyset$
- $\mathcal{A}(t_1 \circ t_2) = \mathcal{A}(t_1) \cup \mathcal{A}(t_2)$

Tuple Projection

Consider $t =$ [name: "Sokrates", age: 69, country: "Greece"]

Single Attribute:

- notation $t.a$
- sample: $t.name =$ "Sokrates"

Multiple Attributes:

- notation $t_{|A}$
- sample: $t_{|\{name,age\}} =$ [name: "Sokrates", age: 69]

Requirements/Effects:

- $a \in \mathcal{A}(t)$, $A \subseteq \mathcal{A}(t)$
- $\mathcal{A}(t_{|A}) = A$
- notice: $t.a$ produces a value, $t_{|A}$ produces a tuple

Relations

Relation:

- a set of tuples with the same schema
- sample: {[name: "Sokrates", age: 69], [name: "Platon", age: 45]}

Schema:

- schema of the contained tuples, written $\mathcal{A}(R)$
- sample: {(name,string),(age, number)}

Sets vs. Bags

- relations are sets of tuples
- real data is usually a multi set (bag)

Example:

```
select age
from student
```

age
23
24
24
...

- we concentrate on sets first for simplicity
- many (but not all) set equivalences valid for bags

The optimizer must consider three different semantics:

- logical algebra operates on bags
- physical algebra operates on streams (order matters)
- explicit duplicate elimination $\Rightarrow$ sets

Set Operations

Set operations are part of the algebra:

- union ($L \cup R$), intersection ($L \cap R$), difference ($L \setminus R$)
- normal set semantic
- but: schema constraints
- for bags defined via frequencies (union $\rightarrow +$, intersection $\rightarrow$ min, difference $\rightarrow -$)

Requirements/Effects:

- $\mathcal{A}(L) = \mathcal{A}(R)$
- $\mathcal{A}(L \cup R) = \mathcal{A}(L) = \mathcal{A}(R)$, $\mathcal{A}(L \cap R) = \mathcal{A}(L) = \mathcal{A}(R)$, $\mathcal{A}(L \setminus R) = \mathcal{A}(L) = \mathcal{A}(R)$

Free Variables

Consider the predicate $age = 62$

- can only be evaluated when *age* has a meaning
- *age* behaves a free variable
- must be bound before the predicate can be evaluated
- notation: $\mathcal{F}(e)$ are the free variables of e

Note:

- free variables are essential for predicates
- free variables are also important for algebra expressions
- dependent join etc.

Selection

Selection:

- notation: $\sigma_p(R)$
- sample: $\sigma_{a \geq 2}(\{[a:1],[a:2],[a:3]\}) = \{[a:2],[a:3]\}$
- predicates can be arbitrarily complex
- optimizer especially interested in predicates of the form $attrib = attrib$ or $attrib = const$

Requirements/Effects:

- $\mathcal{F}(p) \subseteq \mathcal{A}(R)$
- $\mathcal{A}(\sigma_p(R)) = \mathcal{A}(R)$

Projection

Projection:

- notation: $\Pi_A(R)$
- sample: $\Pi_{\{a\}}(\{[a:1, b:1], [a:2, b:1]\}) = \{[a:1], [a:2]\}$
- eliminates duplicates for set semantic, keeps them for bag semantic
- note: usually written as $\Pi_{a,b}$ instead of the correct $\Pi_{\{a,b\}}$

Requirements/Effects:

- $A \subseteq \mathcal{A}(R)$
- $\mathcal{A}(\Pi_A(R)) = A$

Rename

Rename:

- notation: $\rho_{a \to b}(R)$
- sample: $\rho_{a \to c}(\{[a:1, b:1], [a:2, b:1]\}) = \{[c:1, b:1], [c:2, b:2]\}$?
- often a pure logical operator, no code generation
- important for the data flow

Requirements/Effects:

- $a \in \mathcal{A}(R), b \notin \mathcal{A}(R)$
- $\mathcal{A}(\rho_{a \to b}(R)) = \mathcal{A}(R) \setminus \{a\} \cup \{b\}$

Join

Consider $L = \{[a:1],[a:2]\}, R = \{[b:1],[b:3]\}$

Cross Product:

- notation: $L \times R$
- sample: $L \times R = \{[a:1, b:1],[a:1, b:3],[a:2, b:1],[a:2, b:3]\}$

Join:

- notation: $L \bowtie_p R$
- sample: $L \bowtie_{a=b} R = \{[a:1, b:1]\}$
- defined as $\sigma_p(L \times R)$

Requirements/Effects:

- $\mathcal{A}(L) \cap \mathcal{A}(R) = \emptyset, \mathcal{F}(p) \subseteq (\mathcal{A}(L) \cup \mathcal{A}(R))$
- $\mathcal{A}(L \times R) = \mathcal{A}(L) \cup \mathcal{A}(R)$

Equivalences

Equivalences for selection and projection:

$$\sigma_{p_1 \wedge p_2}(e) \equiv \sigma_{p_1}(\sigma_{p_2}(e)) \quad (1)$$

$$\sigma_{p_1}(\sigma_{p_2}(e)) \equiv \sigma_{p_2}(\sigma_{p_1}(e)) \quad (2)$$

$$\Pi_{A_1}(\Pi_{A_2}(e)) \equiv \Pi_{A_1}(e) \quad (3)$$
$$\text{if } A_1 \subseteq A_2$$

$$\sigma_p(\Pi_A(e)) \equiv \Pi_A(\sigma_p(e)) \quad (4)$$
$$\text{if } \mathcal{F}(p) \subseteq A$$

$$\sigma_p(e_1 \cup e_2) \equiv \sigma_p(e_1) \cup \sigma_p(e_2) \quad (5)$$

$$\sigma_p(e_1 \cap e_2) \equiv \sigma_p(e_1) \cap \sigma_p(e_2) \quad (6)$$

$$\sigma_p(e_1 \setminus e_2) \equiv \sigma_p(e_1) \setminus \sigma_p(e_2) \quad (7)$$

$$\Pi_A(e_1 \cup e_2) \equiv \Pi_A(e_1) \cup \Pi_A(e_2) \quad (8)$$

Equivalences

Equivalences for joins:

$$e_1 \times e_2 \equiv e_2 \times e_1 \tag{9}$$

$$e_1 \bowtie_p e_2 \equiv e_2 \bowtie_p e_1 \tag{10}$$

$$(e_1 \times e_2) \times e_3 \equiv e_1 \times (e_2 \times e_3) \tag{11}$$

$$(e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3 \equiv e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) \tag{12}$$

$$\sigma_p(e_1 \times e_2) \equiv e_1 \bowtie_p e_2 \tag{13}$$

$$\sigma_p(e_1 \times e_2) \equiv \sigma_p(e_1) \times e_2 \quad \text{if } \mathcal{F}(p) \subseteq \mathcal{A}(e_1) \tag{14}$$

$$\sigma_{p_1}(e_1 \bowtie_{p_2} e_2) \equiv \sigma_{p_1}(e_1) \bowtie_{p_2} e_2 \quad \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) \tag{15}$$

$$\Pi_A(e_1 \times e_2) \equiv \Pi_{A_1}(e_1) \times \Pi_{A_2}(e_2) \quad \text{if } A = A_1 \cup A_2, A_1 \subseteq \mathcal{A}(e_1), A_2 \subseteq \mathcal{A}(e_2) \tag{16}$$

Canonical Query Translation

Canonical translation of SQL queries into algebra expressions.
Structure:

select distinct $a_1, \ldots, a_n$
from $R_1, \ldots, R_k$
where p

Restrictions:

- only **select distinct** (sets instead of bags)
- no **group by**, **order by**, **union**, **intersect**, **except**
- only attributes in **select** clause (no computed values)
- no nested queries, no views
- not discussed here: NULL values

From Clause

1. Step: Translating the **from** clause

Let $R_1, \ldots, R_k$ be the relations in the **from** clause of the query.
Construct the expression:

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

Where Clause

2. Step: Translating the **where** clause

Let p be the predicate in the **where** clause of the query (if a **where** clause exists). Construct the expression:

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

Select Clause

3. Step: Translating the **select** clause

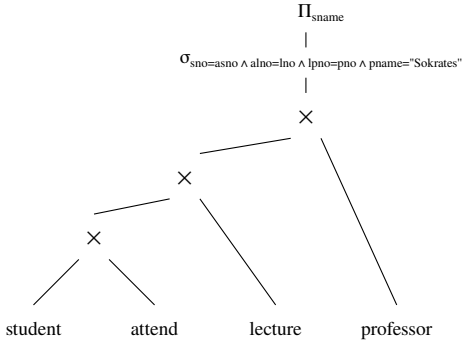
Let $a_1, \ldots, a_n$ (or "*") be the projection in the **select** clause of the query.
Construct the expression:

$$S = \begin{cases} W & \text{if the projection is "*"} \\ \Pi_{a_1,\ldots,a_n}(W) & \text{otherwise} \end{cases}$$

4. Step: S is the canonical translation of the query.

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* =″ *Sokrates*″



Extension - Group By Clause

2.5. Step: Translating the **group by** clause. Not part of the "canonical" query translation!

Let $g_1, \ldots, g_m$ be the attributes in the **group by** clause and agg the aggregations in the **select** clause of the query (if a **group by** clause exists).
Construct the expression:

$$G = \begin{cases} W & \text{if there is no } \textbf{group by} \text{ clause} \\ \Gamma_{g_1,\ldots,g_m;agg}(W) & \text{otherwise} \end{cases}$$

use G instead of W in step 3.

Optimization Phases

Textbook query optimization steps:

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

we have already seen the translation, from now one assume that the algebraic expression is given.

Concept of 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 do not indicate a direction, they can be applied in both ways
- the conditions attached to the equivalences have to be checked

Algebraic equivalences are essential:

- new equivalences increase the potential search space
- better plans
- but search more expensive

Performing Logical Query Optimization

Which plans are better?

- plans can only be compared if there is a *cost function*
- cost functions need details that are not available when only considering logical algebra
- consequence: logical query optimization remains a heuristic

Performing Logical Query Optimization

Most algorithms for logical query optimization use the following strategies:

- organization of equivalences into groups
- directing equivalences

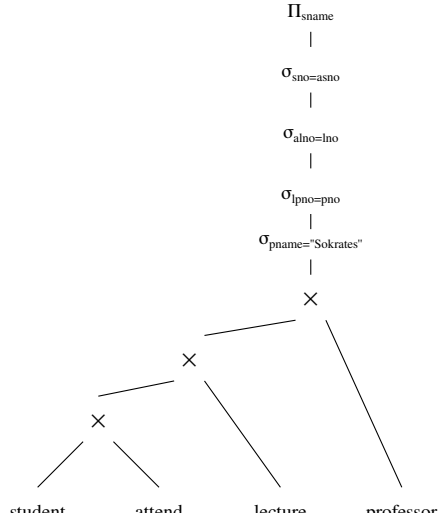
Directing means specifying a preferred side.
A *directed equivalences* is called a *rewrite rule*. The groups of rewrite rules are applied sequentially to the initial algebraic expression. Rough goal: reduce the size of intermediate results

Phases of Logical Query Optimization

1. break up conjunctive selection predicates
   (equivalence (1) $\rightarrow$)
2. push selections down
   (equivalence (2) $\rightarrow$, (14) $\rightarrow$)
3. introduce joins
   (equivalence (13) $\rightarrow$)
4. determine join order
   (equivalence (9), (10), (11), (12))
5. introduce and push down projections
   (equivalence (3) $\leftarrow$, (4) $\leftarrow$, (16) $\rightarrow$)

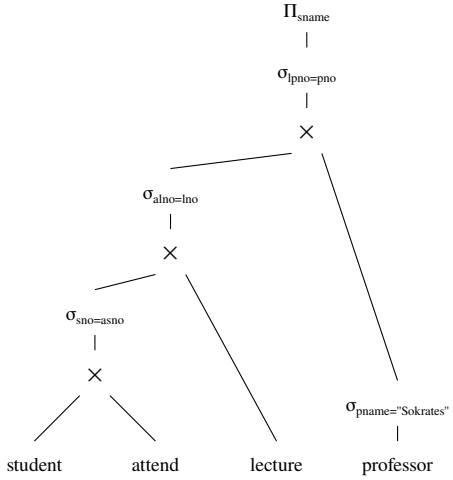
Step 1: Break up conjunctive selection predicates

- selection with simple predicates can be moved around easier



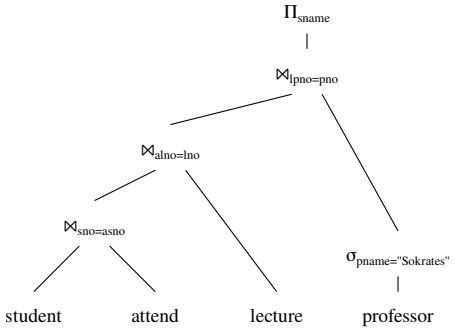
Step 2: Push Selections Down

- reduce the number of tuples early, reduces the work for later operators



Step 3: Introduce Joins

- joins are cheaper than cross products



Step 4: Determine Join Order

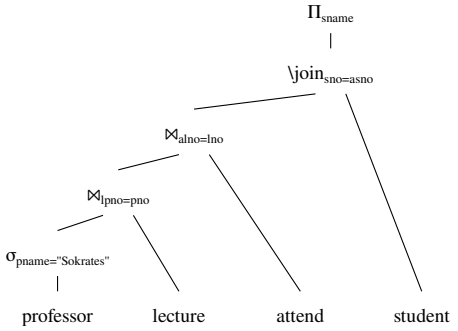
- costs differ vastly
- difficult problem, NP hard (next chapter discusses only join ordering)

Observations in the sample plan:

- bottom most expression is
  $student \bowtie_{sno=asno} attend$
- the result is huge, all students, all their lectures
- in the result only one professor relevant
  $\sigma_{name=''Sokrates''}(professor)$
- join this with lecture first, only lectures by him, much smaller

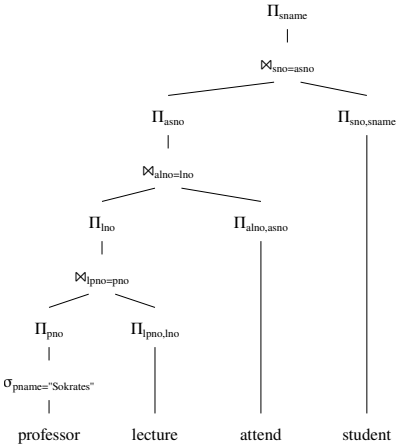
Step 4: Determine Join Order

- intermediate results much smaller



Step 5: Introduce and Push Down Projections

- eliminate redundant attributes
- only before pipeline breakers

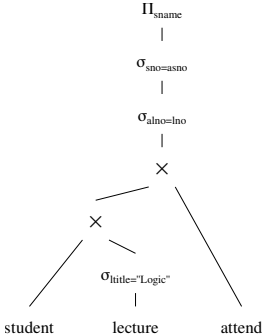


Limitations

Consider the following SQL query

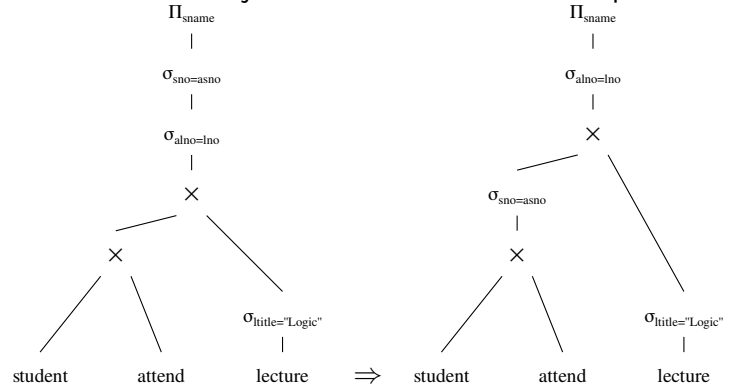
select distinct *s.sname*
from *student s*, *lecture l*, *attend a*
where *s.sno* = *a.asno* **and** *a.alno* = *l.lno* **and** *l.ltitle* =″ *Logic*″

Steps 1-2 could result in plan below. No further selection push down.



Limitations

However a different join order would allow further push down:



- the phases are interdependent
- the separation can loose the optimal solution

Physical Query Optimization

- add more execution information to the plan
- allow for cost calculations
- select index structures/access paths
- choose operator implementations
- add property enforcer
- choose when to materialize (temp/DAGs)

Access Paths Selection

- scan+selection could be done by an index lookup
- multiple indices to choose from
- table scan might be the best, even if an index is available
- depends on selectivity, rule of thumb: 10%
- detailed statistics and costs required
- related problem: materialized views
- even more complex, as more than one operator could be substitued

Operator Selection

- replace a logical operator (e.g. $\bowtie$) with a physical one (e.g. $\bowtie^{HH}$)
- semantic restrictions: e.g. most join operators require equi-conditions
- $\bowtie^{BNL}$ is better than $\bowtie^{NL}$
- $\bowtie^{SM}$ and $\bowtie^{HH}$ are usually better than both
- $\bowtie^{HH}$ is often the best if not reusing sorts
- decission must be cost based
- even $\bowtie^{NL}$ can be optimal!
- not only joins, has to be done for all operators

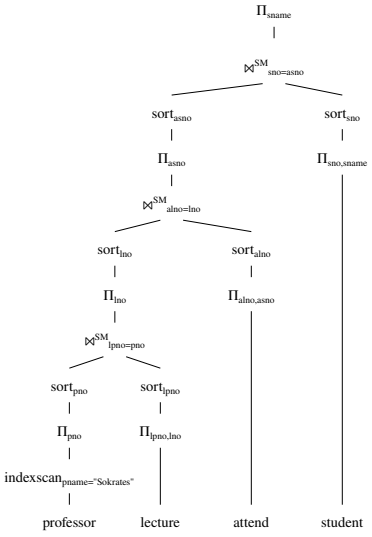
Property Enforcer

- certain physical operators need certain properties
- typical example: sort for $\bowtie^{SM}$
- other example: in a distributed database operators need the data locally to operate
- many operator requirements can be modeled as properties (hashing etc.)
- have to be guaranteed as needed

Materializing

- sometimes materializing is a good idea
- temp operator stores input on disk
- essential for multiple consumers (factorization, DAGs)
- also relevant for $\bowtie^{NL}$
- first pass expensive, further passes cheap

Physical Plan for Sample Query



Outlook

- separation in two phases looses optimality
- many decissions (e.g. view resolution) important for logical optimization
- textbook physical optimization is incomplete
- did not discuss cost calculations
- will look at this again in later chapters

3. Join Ordering

- Basics
- Search Space
- Greedy Heuristics
- IKKBZ
- MVP
- Dynamic Programming
- Simplifying the Query Graph
- Adaptive Optimization
- Generating Permutations
- Transformative Approaches
- Randomized Approaches
- Metaheuristics
- Iterative Dynamic Programming
- Order Preserving Joins
- Complexity of Join Processing

Queries Considered

Concentrate on join ordering, that is:

- conjunctive queries
- simple predicates
- predicates have the form $a_1 = a_2$ where a_1 is an attribute and a_2 is either an attribute or a constant
- even ignore constants in some algorithms

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

- a base relation
- a base relation including selections
- a more complex building block or access path

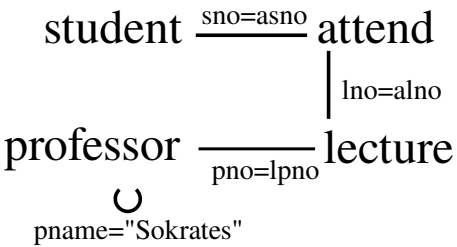
Pretending to have a base relation is ok for now.

Query Graph

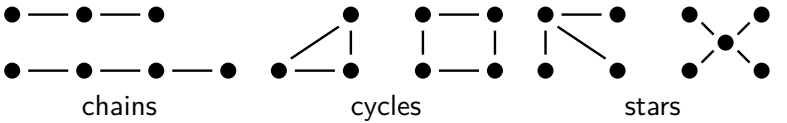
Queries of this type can be characterized by their query graph:

- the query graph is an undirected graph with $R_1, \ldots, R_n$ as nodes
- a predicate of the form $a_1 = a_2$, where $a_1 \in R_i$ and $a_2 \in R_j$ forms an edge between R_i and R_j labeled with the predicate
- a predicate of the form $a_1 = a_2$, where $a_1 \in R_i$ and a_2 is a constant forms a self-edge on R_i labeled with the predicate
- most algorithms will not handle self-edges, they have to be pushed down

Sample Query Graph



Shapes of Query Graphs



- real world queries are somewhere in-between
- chain, cycle, star and clique are interesting to study
- they represent certain kind of problems and queries

Join Trees

A join tree is a binary tree with

- join operators as inner nodes
- relations as leaf nodes

Algorithms will produce different kinds of join trees

- ordered or unordered
- with cross products or without

The most common case is ordered, without cross products

Shape of Join Trees

Commonly used classes of join trees:

- left-deep tree
- right-deep tree
- zigzag tree
- bushy tree

The first three are summarized as *linear trees*.

Join Selectivity

Input:

- cardinalities $|R_i|$
- selectivities $f_{i,j}$: if $p_{i,j}$ is the join predicate between R_i and R_j, define

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

Calculate:

- result cardinality:

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

Rational: The selectivity can be computed/estimated easily (ideally).

Cardinality of Join Trees

Given a join tree T, the result cardinality $|T|$ can be computed recursively as

$$|T| = \begin{cases} |R_i| & \text{if } T \text{ is a leaf } R_i \\ (\prod_{R_i \in T_1, R_j \in T_2} f_{i,j}) |T_1||T_2| & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

- allows for easy calculation of join cardinality
- requires only base cardinalities and selectivities
- assumes independence of the predicates

Sample Statistics

As running example, we use the following statistics:

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

- implies query graph $R_1 - R_2 - R_3$
- assume $f_{i,j} = 1$ for all other combinations

A Basic Cost Function

Given a join tree T, the cost function C_{out} is defined as

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

- sums up the sizes of the (intermediate) results
- rational: larger intermediate results cause more work
- we ignore the costs of single relations as they have to be read anyway

Basic Join Specific Cost Functions

For single joins:

$$\begin{aligned} C_{nlj}(e_1 \bowtie e_2) &= |e_1||e2| \\ C_{hj}(e_1 \bowtie e_2) &= 1.2|e_1| \\ C_{smj}(e_1 \bowtie e_2) &= |e_1|\log(|e_1|) + |e_2|\log(|e_2|) \end{aligned}$$

For sequences of join operators $s = s_1 \bowtie \ldots \bowtie s_n$:

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

Remarks on the Basic Cost Functions

- cost functions are simplistic
- algorithms are modelled very simplified (e.g. 1.2, no n-way sort etc.)
- designed for left-deep trees
- C_{hj} and C_{smj} do not work for cross products (fix: take output cardinality then, which is C_{nl})
- in reality: other parameters than cardinality play a role
- cost functions assume the same join algorithm for the whole join tree

Sample Cost Calculations

	C_{out}	C_{nl}	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

- costs differ vastly between join trees
- different cost functions result in different costs
- the cheapest plan is always the same here, but relative order varies
- join trees with cross products are expensive
- join order is essential under all cost functions

More Examples

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

	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

- here cross product is best
- but relies on the small sizes of $|R_2|$ and $|R_3|$
- attractive if the cardinality of one relation is small

More Examples (2)

For the query $|R_1| = 10, |R_2| = 20, |R_3| = 20, |R_4| = 10, f_{1,2} = 0.01, f_{2,3} = 0.5, f_{3,4} = 0.01$ we have costs:

	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
$((R_2 \times R_3) \bowtie R_1) \bowtie R_4$	222
$(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$	6

- covers all join trees due to the symmetry of the query
- the bushy tree is better than all join trees

Symmetry and ASI

- cost function C_{impl} is called *symmetric* if $C_{impl}(e_1 \bowtie^{impl} e_2) = C_{impl}(e_2 \bowtie^{impl} e_1)$
- for symmetric cost functions commutativity can be ignored
- ASI: *adjacent sequence interchange* (see IKKBZ algorithm for a definition)

Our basic cost functions can be classified as:

	ASI	¬ASI
symmetric	C_{out}	C_{smj}
¬symmetric	C_{hj}	-

- more complex cost functions are usually ¬ASI, often also ¬symmetric
- symmetry and especially ASI can be exploited during optimization

Classification of Join Ordering Problems

We distinguish four different dimensions:

1. query graph class: *chain*, *cycle*, *star*, and *clique*
2. join tree structure: *left-deep*, *zig-zag*, or *bushy* trees
3. join construction: *with* or *without* cross products
4. cost function: *with* or *without* ASI property

In total, 48 different join ordering problems.

Reminder: Catalan Numbers

The number of binary trees with n leave nodes is given by $\mathcal{C}(n-1)$, where $\mathcal{C}(n)$ is defined as

$$\mathcal{C}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{k=0}^{n-1} \mathcal{C}(k)\mathcal{C}(n-k-1) & \text{if } n > 0 \end{cases}$$

It can be written in a closed form as

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

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

Number Of Join Trees with Cross Products

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

- rational: number of leaf combinations ($n!$) $\times$ number of unlabeled trees (varies)
- grows exponentially
- increases even more with a flexible tree structure

Chain Queries, no Cross Products

Let us denote the number of left-deep join trees for a chain query $R_1 - \ldots - R_n$ as $f(n)$

- obviously $f(0) = 1, f(1) = 1$
- for $n > 1$, consider adding R_n to all join trees for $R_1 - \ldots - R_{n-1}$
- R_n can be added at any position following R_{n-1}
- lets denote the position of R_{n-1} from the bottom with k ($[1, n-1]$)
- there are $n - k$ join trees for adding R_n after R_{n-1}
- one additional tree if $k = 1$, R_n can also be added before R_{n-1}
- for R_{n-1} to be at k, $R_{n-k} - \ldots R_{n-2}$ must be below it. $f(k-1)$ trees

for $n > 1$:

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

Chain Queries, no Cross Products (2)

The number of left-deep join trees for chain queries of size n is

$$f(n) = \begin{cases} 1 & \text{if } n < 2 \\ 1 + \sum_{k=1}^{n-1} f(k-1) * (n-k) & \text{if } n \geq 2 \end{cases}$$

solving the recurrence gives the closed form

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

- generalization to zig-zag as before

Chain Queries, no Cross Products (3)

The generalization to bushy trees is not as obvious

- each subtree must contain a subchain to avoid cross products
- thus do not add single relations but subchains
- whole chain must be $R_1 - \ldots - R_n$, cut anywhere
- consider commutativity (two possibilities)

This leads to the formula

$$f(n) = \begin{cases} 1 & \text{if } n < 2 \\ \sum_{k=1}^{n-1} 2f(k)f(n-k) & \text{if } n \geq 2 \end{cases}$$

solving the recurrence gives the closed form

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

Star Queries, no Cross Products

Consider a star query with R_1 at the center and $R_2, \ldots, R_n$ as satellites.

- the first join must involve R_1
- afterwards all other relations can be added arbitrarily

This leads to the following formulas:

- left-deep: $2 * (n-1)!$
- zig-zag: $2 * (n-1)! * 2^{n-2} = (n-1)! * 2^{n-1}$
- bushy: no bushy trees possible (R_1 required), same as zig-zag

Clique Queries, no Cross Products

- in a clique query, every relation is connected to each other
- thus no join tree contains cross products
- all join trees are valid join trees, the number is the same as with cross products

Sample Numbers, without Cross Products

	Chain Queries			Star Queries	
n	Left-Deep 2^{n-1}	Zig-Zag 2^{2n-3}	Bushy $2^{n-1}\mathcal{C}(n-1)$	Left-Deep $2(n-1)!$	Zig-Zag/Bushy $2^{n-1}(n-1)!$
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	18579450

Sample Numbers, with Cross Products

n	Left-Deep $n!$	Zig-Zag $n!2^{n-2}$	Bushy $n!\mathcal{C}(n-1)$
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	968972800	17643225600

Problem Complexity

query graph	join tree	cross products	cost function	complexity
general	left-deep	no	ASI	NP-hard
tree/star/chain	left-deep	no	ASI, 1 joint.	P
star	left-deep	no	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

Greedy Heuristics - First Algorithm

- search space of joins trees is very large
- greedy heuristics produce suitable join trees very fast
- suitable for large queries

For the first algorithm we consider:

- left-deep trees
- no cross products
- relations ordered to some weight function (e.g. cardinality)

Note: the algorithms produces a sequence of relations; it uniquely identifies the left-deep join tree.

Greedy Heuristics - First Algorithm (2)

GreedyJoinOrdering-1($R = \{R_1, \ldots, R_n\}$,$w : R \to \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \epsilon$
while ($|R| > 0$) {
  $m = \arg\min_{R_i \in R} w(R_i)$
  $R = R \setminus \{m\}$
  $S = S \circ < m >$
}
return S

- disadvantage: fixed weight functions
- already chosen relations do not affect the weight
- e.g. does not support minimizing the intermediate result

Greedy Heuristics - Second Algorithm

GreedyJoinOrdering-2($R = \{R_1, \ldots, R_n\}$,$w : R, R^* \rightarrow \mathbb{R}$)
Input: a set of relations to be joined and weight function
Output: a join order
$S = \epsilon$
while $(|R| > 0)$ {
  $m = \arg\min_{R_i \in R} w(R_i, S)$
  $R = R \setminus \{m\}$
  $S = S \circ < m >$
}
return S

- can compute relative weights
- but first relation has a huge effect
- and the fewest information available

Greedy Heuristics - Third Algorithm

GreedyJoinOrdering-3($R = \{R_1, \ldots, R_n\}$, $w : R, R^* \rightarrow \mathbb{R}$)

Input: a set of relations to be joined and weight function

Output: a join order

$S = \emptyset$

for each $R_i \in R$ {

  $R' = R \setminus \{R_i\}$

  $S' =< R_i >$

  while ($|R'| > 0$) {

    $m = \arg\min_{R_j \in R'} w(R_j, S')$

    $R' = R' \setminus \{m\}$

    $S' = S' \circ < m >$

  }

  $S = S \cup \{S'\}$

}

return $\arg\min_{S' \in S} w(S'[n], S'[1 : n-1])$

- commonly used: minimize selectivities (*MinSel*)

Greedy Operator Ordering

- the previous greedy algorithms only construct left-deep trees
- Greedy Operator Ordering (GOO) [1] constructs bushy trees

Idea:

- all relations have to be joined somewhere
- but joins can also happen between whole join trees
- we therefore greedily combine join trees (which can be relations)
- combine join trees such that the intermediate result is minimal

Greedy Operator Ordering (2)

GOO($R = \{R_1, \ldots, R_n\}$)
Input: a set of relations to be joined
Output: a join tree
T = R
while $|T| > 1$ {
  $(T_i, T_j) = \arg\min_{(T_i \in T, T_j \in T), T_i \neq T_j} |T_i \bowtie T_j|$
  $T = (T \setminus \{T_i\}) \setminus \{T_j\}$
  $T = T \cup \{T_i \bowtie T_j\}$
}
return $T_0 \in T$

- constructs the result bottom up
- join trees are combined into larger join trees
- chooses the pair with the minimal intermediate result in each pass

IKKBZ

Polynomial algorithm for join ordering (original [2], improved [3])

- produces optimal left-deep trees without cross products
- requires acyclic join graphs
- cost function must have ASI property
- join method must be fixed

Can be used as heuristic if the requirements are violated

Overview

- the algorithms considers each relation as first relation to be joined
- it tries to order the other relations by "benefit" (rank)
- if the ordering violates the query constraints, it constructs compounds
- the compounds guarantee the constraints (locally) and are again ordered by benefit
- related to a known job-ordering algorithm

Cost Function

The IKKBZ algorithm considers only cost functions of the form

$$C(T_i \bowtie R_j) = |T_i| * h_j(|R_j|)$$

- each relation R_j can have its own h_j
- we denote the set of h_j by H, writing C_H for the parametrized cost function
- examples: $h_j \equiv 1.2$ for C_{hj}, $h_j \equiv id$ for C_{nl}

We will often use cardinalities, thus we define n_i:

- n_i is the cardinality of R_i ($n_i = |R_i|$)
- $h_i(n_i)$ is are the costs per input tuple of a join with R_i

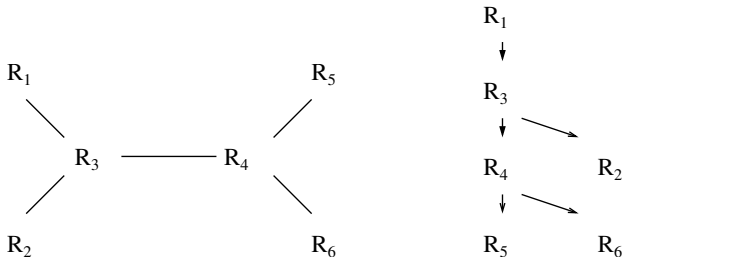
Precedence Graph

Given a query graph $G = (V, E)$ and a starting relation R_k, we construct the directed *precedence graph* $G_k^P = (V_k^P, E_k^P)$ rooted in R_k as follows:

1. choose R_k as the root node of G_k^P, $V_k^P = \{R_k\}$
2. while $|V_k^P| < |V|$, choose a $R_i \in V \setminus V_k^P$ such that $\exists R_j \in V_k^P : (R_j, R_i) \in E$. Add R_i to V_k^P and $R_j \rightarrow R_i$ to E_k^P.

The precedence graph describes the (partial) ordering of joins implied by the query graph.

Sample 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. $\forall i \in [2, k] \exists j \in [1, i[: (v_j, v_i) \in E$
2. $\nexists i \in [1, k], j \in]i, k] : (v_j, v_i) \in E$

Note: IKKBZ constructs left-deep trees, therefore it is sufficient to consider sequences.

Notations

For non-empty sequences S_1 and S_2 and a precedence graph $G = (V, E)$, we write $S_1 \rightarrow S_2$ if S_1 must occur before S_2. More precisely $S_1 \rightarrow S_2$ iff:

1. S_1 and S_2 conform to G
2. $S_1 \cap S_2 = \emptyset$
3. $\exists v_i, v_j \in V : v_i \in S_1 \wedge v_j \in S_2 \wedge (v_i, v_j) \in E$
4. $\nexists v_i, v_j \in V : v_i \in S_1 \wedge v_j \in V \setminus S_1 \setminus S_2 \wedge (v_i, v_j) \in E$

Further, we write

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

Selectivities

For a given precedence graph, let R_i be a relation and $\mathcal{R}_i$ be the set of a relations from which there exists a path to R_i

- in any conforming join tree which includes R_i, all relations from $\mathcal{R}_i$ must be joined first
- all other relations R_j that might be joined before R_i will have no connection to R_i, thus $f_{i,j} = 1$

Hence, we can define the selectivity of the join with R_i as

$$s_i = \begin{cases} 1 & \text{if } |\mathcal{R}_i| = 0 \\ \prod_{R_j \in \mathcal{R}_i} f_{i,j} & \text{if } |\mathcal{R}_i| > 0 \end{cases}$$

Note: we call the s_i a selectivities, although they depend on the precedence graph

Cardinalities

If the query graph is a chain (totally ordered), the following conditions holds:

$$\begin{aligned} n_{1,2,\ldots,k} &= s_k * |R_k| * |R_{1,2,\ldots,k-1}| \\ &= |s_k| * n_k * n_{1,2,\ldots,k-1} \end{aligned}$$

As a closed form, we can write

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

as $s_1 = 1$

Costs

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

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

- if we choose $h_i(n_i) = s_i n_i$ then $C_H \equiv C_{out}$
- the factor $s_i n_i$ determines 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*

Costs (2)

For the algorithm, we prefer a (equivalent) recursive definition of the cost function:

$$\begin{aligned} C_H(\epsilon) &= 0 \\ C_H(R_i) &= 0 \text{ if } R_i \text{ is the root} \\ C_H(R_i) &= h_i(n_i) \text{ else} \\ C_H(S_1 S_2) &= C_H(S_1) + T(S_1) * C_H(S_2) \end{aligned}$$

where

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

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 exists a function T and a rank function defined as

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

such that the following holds

$$C(AUVB) \leq C(AVUB) \Leftrightarrow rank(U) \leq rank(V)$$

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

First Lemma

Lemma: The cost function C_h has the ASI-Property.
Proof: The proof can be derived from the definition of C_H:

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

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.

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:

- $B \rightarrow A_i$, $\forall A_i \in M$
- $A_i \rightarrow B$, $\forall A_i \in M$
- $B \not\rightarrow A_i$ and $A_i \not\rightarrow B$, $\forall A_i \in M$

Second Lemma

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

Proof: by contradiction. Every optimal permutation must have the form $UAVBW$ since $A \rightarrow B$.

Assumption: $V \neq \epsilon$ for all optimal solutions.

- if $rank(V) \leq rank(A)$, we can exchange V and A without increasing the costs.
- if $rank(A) \leq rank(V)$, $rank(B) \leq rank(V)$ due to the transitivity of $\leq$. Hence, we can exchange B and V without increasing the costs.

Both exchanges produces legal sequences since $\{A, B\}$ is a module.

Contradictory Sequences and Compound Relations

- if the precedence graph demands $A \rightarrow B$ but $rank(B) \leq rank(A)$, we speak of *contradictory sequences* A and B
- second lemma $\Rightarrow$ no non-empty subsequence can occur between A and B
- we combine A and B into a new single node replacing A and B
- this nodes represents a *compound relation* comprising of all relations in A and B
- its cardinality is computed by multiplying the cardinalities of all relations in A and B
- its selectivity is the product of all selectivities s_i of relations R_i contained in A and B

Normalization and Denormalization

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

Algorithm

IKKBZ(G, C_H)
Input: an acyclic query graph G for relations $R = \{R_1, \ldots, R_n\}$,
a cost function C_H
Output: the optimal left-deep tree
$S = \emptyset$
for each $R_i \in R$ {
  G_i = the precedence graph derived from G rooted at R_i
  S_i = IKKBZ-Sub(G_i, C_H)
  $S = S \cup \{S_i\}$
}
return $\arg\min_{S_i \in S} C_H(S_i)$

- considers each relation as starting relation
- constructs the precedence graph and starts the main algorithm

Algorithm (2)

IKKBZ-Sub(G_i, C_H)
Input: a precedence graph G_i for relations $R = \{R_1, \ldots, R_n\}$ rooted at R_i,
a cost function C_H
Output: the optimal left-deep tree under G_i
while G_i is not a chain {
  r = a subtree of G_i whose subtrees are chains
  IKKBZ-Normalize(r)
  merge the chains under r according to the rank function (ascending)
}
IKKBZ-Denormalize(G_i)
return G_i

- transforms the precedence graph into a chain
- wherever there are multiple choices, there are serialized according to the rank
- normalization required to preserve the query graph

Algorithm (3)

IKKBZ-Normalize(R)
Input: a subtree R of a precedence graph $G = (V, E)$
Output: a normalized subtree
while $\exists r, c \in T, (r, c) \in E : rank(r) > rank(c)$ {
  replace r and c by a compound relation r' that represent rc
}
return R

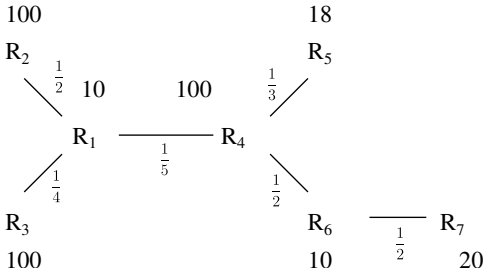
- merges relations that would have been reorder if only considering the rank
- guarantees that the rank is ascending in each subchain

Algorithm (4)

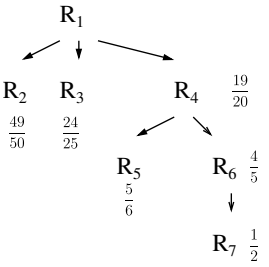
IKKBZ-Denormalize(R)
Input: a precedence graph R containing relations and compound relations
Output: a denormalized precedence graph, containing only relations
while $\exists r \in R : r$ is a compound relation $\{$
  replace r by the sequence of relations it represents
$\}$
return R

- unpacks the compound relations
- required to get a real join tree as final result

Sample Algorithm Execution



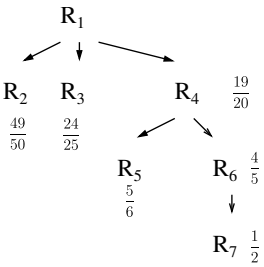
Input: query graph



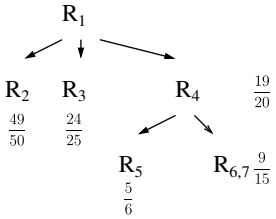
Step 1: precedence graph for R_1

the precedence graph includes the ranks

Sample Algorithm Execution (2)



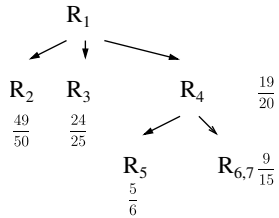
Step 1: precedence graph for R_1



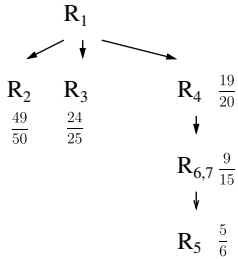
Step 2: normalization

$rank(R_6) > rank(R_7)$, but $R_6 \rightarrow R_7$

Sample Algorithm Execution (3)



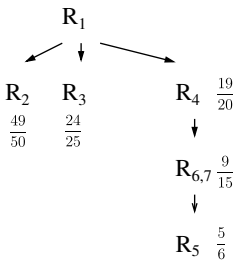
Step 2: normalization



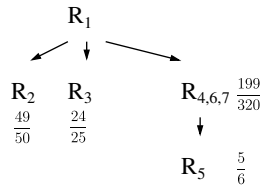
Step 3: merging subchains

$rank(R_{6,7}) < rank(R_5)$

Sample Algorithm Execution (3)



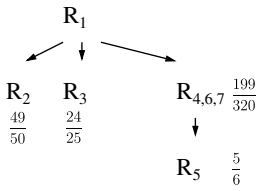
Step 3: merging subchains



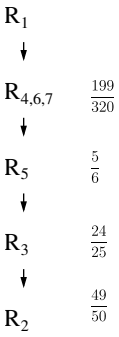
Step 4: normalization

$rank(R_4) > rank(R_{6,7})$, but $R_4 \rightarrow R_{6,7}$

Sample Algorithm Execution (4)



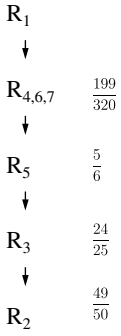
Step 4: normalization



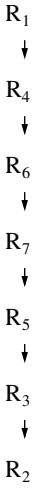
Step 5: merging subchains

$rank(R_{4,6,7}) < rank(R_5) < rank(R_3) < rank(R_2)$

Sample Algorithm Execution (5)



Step 5: merging subchains



Step 6: denormalization

Algorithm has to continue for all other root relations.

Maximum Value Precedence Algorithm

- greedy heuristics can produce poor results
- IKKBZ only support acyclic queries and ASI cost functions
- Maximum Value Precedence (MVP) [4] algorithm is a polynomial time heuristic with good results
- considers join ordering a graph theoretic problem

Directed Join Graph

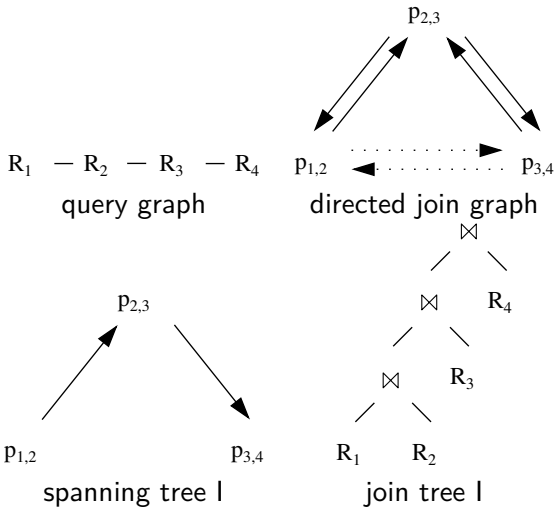
Given a conjunctive query with predicates P.

- for all join predicates $p \in P$, we denote by $\mathcal{R}(p)$ the relations whose attributes are mentioned in p.
- the *directed join graph* of the query is a triple $G = (V, E_p, E_v)$, where V is the set of predicates and E_p and E_v are sets of directed edges defined as follows
- for any 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 $(v, u) \in E_v$
- edges in E_p are called *physical edges*, those in E_v *virtual edges*

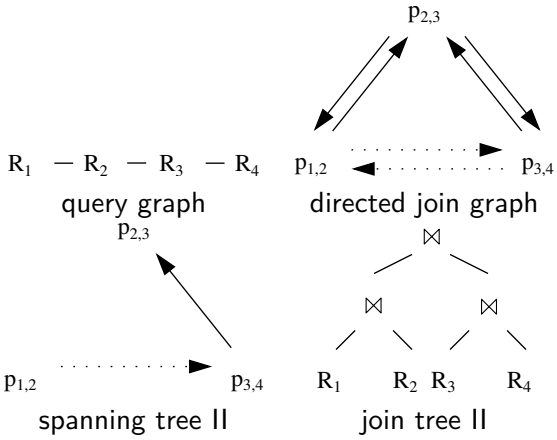
Note: all nodes u, v there is an edge (u, v) that is either physical or virtual. Hence, G is a clique.

Examples: Spanning Tree and Join Tree

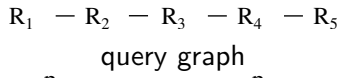
- every spanning tree in the directed join graph leads to a join tree



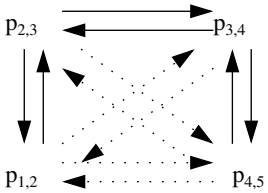
Examples: Spanning Tree and Join Tree (2)



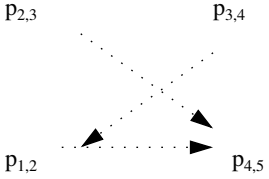
Examples: Spanning Tree and Join Tree (3)



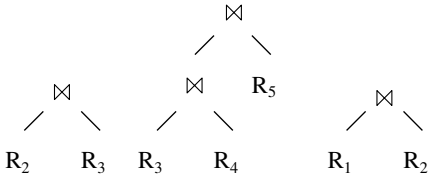
query graph



directed join graph



spanning tree III



join tree III (?)

- spanning tree does not correspond to a (effective) join tree!

Effective Spanning Trees

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

1. T is a binary tree
2. for all inner nodes v and nodes u with $(u, v) \in E$: $\mathcal{R}(T(u))) \cap \mathcal{R}(v) \neq \emptyset$
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 $(\mathcal{R}(T(u_1)) = \mathcal{R}(v)) \vee (\mathcal{R}(T(u_2)) = \mathcal{R}(v))$

We denote by $T(v)$ the partial tree rooted at v.

Adding Weights to the Edges

For two nodes $v, u \in V$ we 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 $u, v \in V$, $a \sqcap v$ contains a single relation (or none)

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 (2)

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 to 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

$$w_{u,v} = 1$$

Note that $w_{u,v}$ is not symmetric.

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

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

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 of $w_{u,v}$

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
- it depends 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

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

For empty sequences we define $w(p_{i,j}, \epsilon) = |R_i \bowtie_{p_{i,j}} R_j|$.
Similarly, 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 $C(p_{i,j}, S)$.

- the actual cost function can be chosen arbitrarily
- if we have several join implementations: take the minimum

Algorithm Overview

The algorithm builds an effective spanning tree in two phases:

1. it takes those edges with a weight < 1
2. it adds the remaining edges

keeping track of effectiveness during the process.

- rational: weight < 1 is good
- decreases the work for later operators
- should be done early
- increasing intermediate results as late as possible

MVP Algorithm

MVP(G)
Input: a weighted directed join graph $G = (V, E_p, E_v)$
Output: an effective spanning tree
Q_1 = a priority queue for nodes, largest w first
Q_2 = a priority queue for nodes, smallest w first
insert all nodes in V to Q_1
$G' = (V', E')$ with $V' = V$ and $E' = E_p$ // working graph
$S = (V_S, E_s)$ with $V_s = V$ and $E_s = \emptyset$ // result
MVP-Phase1(G, G', S, Q_1, Q_2)
MVP-Phase2(G, G', S, Q_1, Q_2)
return S

MVP Algorithm (2)

MVP-Phase1(G, G', S, Q_1, Q_2)
Input: state from MVP
Output: modifies the state
while $|Q_1| > 0 \wedge |E_s| < |V| - 1$ {
  $v =$ head of Q_1
  $U = \{u | (u, v) \in E' \wedge w_{u,v} < 1 \wedge (V, E_S \cup \{(u, v)\})$ is acyclic and effective$\}$
  if $U = \emptyset$ {
    $Q_1 = Q_1 \setminus \{v\}$
    $Q_2 = Q_2 \cup \{v\}$
  } **else** {
    $u = \arg\max_{u \in U} C(\bowtie_v, S) - C(\bowtie_v, (V, E_S \cup \{(u, v)\}))$
    MVPUpdate$(G, G', S, (u, v))$
    recompute w for v and its ancestors
  }
}

MVP Algorithm (3)

MVP-Phase2(G, G', S, Q_1, Q_2)
Input: state from MVP
Output: modifies the state
while $|Q_2| > 0 \wedge |E_s| < |V| - 1$ {
  v = head of Q_2
  $U = \{(x,y) | (x,y) \in E' \wedge (x = v \vee y = v) \wedge (V, E_S \cup \{(x,y)\})$ is acyclic and effective$\}$
  $(x,y) = \arg\min_{(x,y)\in U} C(\bowtie_v, (V, E_S \cup \{(x,y)\})) - C(\bowtie_v, S)$
  MVPUpdate($G, G', S, (x,y)$)
  recompute w for y and its ancestors
}

MVP Algorithm (4)

MVPUpdate$(G, G', S, (u, v))$
Input: state from MVP, an edge to be added to S
Output: modifies the state
$E_S = E_S \cup \{(u, v)\}$
$E' = E' \setminus \{(u, v), (v, u)\}$
$E' = E' \setminus \{(u, w) | (u, w) \in E'\}$
$E' = E' \cup \{(v, w) | (u, w) \in E_p, (v, w) \in E_v\}$
if v has two incoming edges in S $\{$
  $E' = E' \setminus \{(w, v) | (w, v) \in E'\}$
$\}$
if v has one outflowing edge in S $\{$
  $E' = E' \setminus \{(v, w) | (v, w) \in E'\}$
$\}$

- checks that S is a tree (one parent, at most two children)
- detects transitive physical edges

Dynamic Programming

Basic premise:

- optimality principle
- avoid duplicate work

A very generic class of approaches:

- all cost functions (as long as optimality principle holds)
- left-deep/bushy, with/without cross products
- finds the optimal solution

Concrete algorithms can be more specialized of course.

Optimality Principle

Consider the two joins 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 is cheaper than the second join
- hence, we could avoid generating the second alternative and still won't miss the optimal join tree

Optimality Principle (2)

More formally, the optimality 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.*

- optimal substructure: the optimal solution for a problem can be constructed from optimal solutions to its subproblems
- not true with physical properties (but can be fixed)

Overview Dynamic Programming Strategy

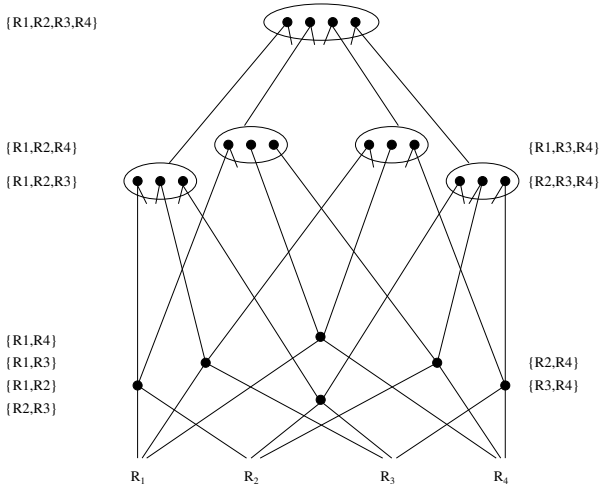
- generate optimal join trees bottom up
- start from optimal join trees of size one (relations)
- build larger join trees by (re-)using those of smaller sizes

To keep the algorithms concise, we use a subroutine *CreateJoinTree* that joins two trees.

Creating Join Trees

CreateJoinTree(T_1, T_2)
Input: two (optimal) join trees T_1, T_2
for linear trees: assume that T_2 is a single relation
Output: an (optimal) join tree for $T_1 \bowtie T_2$
$B = \emptyset$
for each *impl* $\in$ { applicable join implementations } {
  if $\neg$right-deep only {
    $B = B \cup \{T_1 \bowtie^{impl} T_2\}$
  }
  if $\neg$left-deep only {
    $B = B \cup \{T_2 \bowtie^{impl} T_1\}$
  }
}
return $\arg\min_{T \in B} C(T)$

Search Space with Sharing under Optimality Principle



Generating Linear Trees

- a (left-deep) linear tree T with $|T| > 1$ has the form $T' \bowtie R_i$, with $|T| = |T'| + 1$
- if T is optimal, T' must be optimal too
- basic strategy: find the optimal T by joining all optimal T' with $T \setminus T'$

enumeration order varies between algorithms

Generating Linear Trees (2)

DPsizeLinear(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep (right-deep, zig-zag) join tree
B = an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < s \leq n$ **ascending** $\{$
  for each $S \subset R, R_i \in R : |S| = s - 1 \wedge R_i \notin S$ $\{$
    if $\neg$cross products $\wedge \neg S$ connected to R_i **continue**
    $p_1 = B[S], p_2 = B[\{R_i\}]$
    if $p_1 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S \cup \{R_i\}] = \epsilon \vee C(B[S \cup \{R_i\}]) > C(P)$
      $B[S \cup \{R_i\}] = P$
  $\}$
$\}$

Order in which Subtrees are generated

The ordering 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.*

- *relevant* means that they are valid subproblems by the algorithm
- usually this means connected (no cross products)

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	$\{R_1, R_2, R_3\}$

- can be done very efficiently
- set representation is just a number

Generating Linear Trees (3)

DPsubLinear(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep (right-deep, zig-zag) join tree
B = an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < i \leq 2^n - 1$ **ascending** $\{$
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \; mod \; 2) = 1\}$
  for each $R_j \in S$ $\{$
    if $\neg$cross products $\wedge \neg S \setminus \{R_j\}$ connected to R_j **continue**
    $p_1 = B[S \setminus \{R_j\}], p_2 = B[\{R_j\}]$
    if $p_1 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  $\}$
$\}$

Generating Bushy Trees

- a bushy tree T with $|T| > 1$ has the form $T_1 \bowtie T_2$, with $|T| = |T_1| + |T_2|$
- if T is optimal, both T_1 and T_2 must be optimal too
- basic strategy: find the optimal T by joining all pairs of optimal T_1 and T_2

Generating Bushy Trees (2)

DPsize(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < s \leq n$ **ascending** $\{$
  for each $S_1, S_2 \subset R : |S_1| + |S_2| = s$ $\{$
    if ($\neg$cross products $\wedge \neg S_1$ connected to S_2) $\vee$ ($S_1 \cap S_2 \neq \emptyset$) **continue**
    $p_1 = B[S_1], p_2 = B[S_2]$
    if $p_1 = \epsilon \vee p_2 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S_1 \cup S_2] = \epsilon \vee C(B[S_1 \cup S_2]) > C(P)$
      $B[S_1 \cup S_2] = P$
  $\}$
$\}$

Generating Bushy Trees (3)

DPsub(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
for each $1 < i \leq 2^n - 1$ **ascending** $\{$
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \ mod\ 2) = 1\}$
  for each $S_1 \subset S, S_2 = S \setminus S_1$ $\{$
    if $\neg$cross products $\wedge \neg S_1$ connected to S_2 **continue**
    $p_1 = B[S_1], p_2 = B[S_2]$
    if $p_1 = \epsilon \vee p_2 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  $\}$
$\}$

Efficient Subset Generation

If we use integers as set representation, we can enumerate all subsets of S as follows:

$S_1 = S\&(-S)$
do $\{$
  $S_2 = S - S_1$
  // Do something with S_1 and S_2
  $S_1 = S\&(S_1 - S)$
$\}$ **while** $(S_1 != S)$

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

Remarks

- DPsize/DPsizeLinear does not really test for $p_1 = \epsilon$
- it keeps a list of plans for a given size
- candidates can be found very fast
- ensures polynomial time in some cases (we will look at it again)
- DPsub/DPsubLinear is faster if the problem is not polynomial, though

Memoization

- top-down formulation of dynamic programming
- recursive generation of join trees
- memoize already generated join trees to avoid duplicate work
- easier code
- sometimes more efficient (more knowledge, allows for pruning)
- but usually slower than dynamic programming

Memoization (2)

Memoization(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for each $R_i \in R$
  $B[\{R_i\}] = R_i$
MemoizationRec(B, R)
return $B[\{R_1, \ldots, R_n\}]$

- initializes the DP table and triggers the recursive search
- main work done during recursion

Memoization (3)

MemoizationRec(B,S)
Input: a DP table B and a set of relations S to be joined
Output: an optimal bushy join tree for the subproblem
if $B[S] = \epsilon$ {
  for each $S_1 \subset S, S_2 = S \setminus S_1$
    p_1 =MemoizationRec(B, S_1), p_2 =MemoizationRec(B, S_2)
    P=CreateJoinTree(p_1, p_2)
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  }
}
return $B[S]$

- checks for connectedness omitted

Dynamic Programming - Connected Subgraphs

- DP a very versatile strategy
- common usage scenario: bushy, no cross produts
- DPsize and DPsub support it, of course, but not optimal
- enumeration order does not consider the query graph
- many pairs have to be pruned due to conectedness
- especially bad for DPsub

Solution: consider the query graph structure during DP enumeration [5]

Asymptotic Search Space

DPsize:

- organize DP by the size of the join tree
- problem: only few DP slots, many pairs considered

good algorithm for chains, very bad for cliques:

	chains	cycles	stars	cliques
pairs	$O(n^4)$	$O(n^4)$	$O(4^n)$	$O(4^n)$

DPsub:

- organize DP by the set of relations involved
- problem: always 2^n DP slots, fixed enumeration

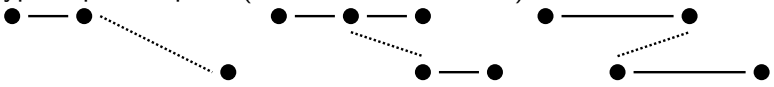
good algorithm for cliques, but adapts badly:

	chains	cycles	stars	cliques
pairs	$O(2^n)$	$O(n2^n)$	$O(3^n)$	$O(3^n)$

Observation

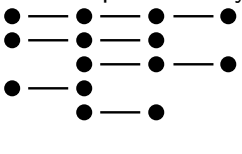
DPsize and DPsub generate many pairs that are pruned anyway (connectedness, overlap).

Typical pruned pairs (chain with 4 relations):



not connected | not disjoint | invalid subproblems

last example ⇒ every join partner must be a connected subgraph:



...

Graph Theoretic Approach

- reformulation as graph theoretic problem:
- enumerate all connected subgraphs of the query graph
- for each subgraph enumerate all other connected subgraphs that are disjoint but connected to it
- each connected subgraph - complement pair (ccp) can be joined
- enumerate them suitable for DP $\Rightarrow$ DP algorithm

algorithm adapts naturally to the graph structure:

	chains	cycles	stars	cliques
pairs	$O(n^3)$	$O(n^3)$	$O(n2^n)$	$O(3^n)$

Lohman et al: #ccp is a lower bound for all DP enumeration algorithms

DP Algorithm using Connected Subgraphs

If we can efficiently enumerate all connected subgraphs/connected complement pairs, the resulting DP algorithm is:

DPccp(R)
Input: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
for $\forall$ csg-cmp-pairs (S_1, S_2), $S = S_1 \cup S_2$ {
  $p_1 = B[S_1]$, $p_2 = B[S_2]$
  $P =$ CreateJoinTree(p_1, p_2);
  if $B[S] = \epsilon \vee C(B[S]) > C(P)$
    $B[S] = P$
}
return $B[\{R_0, \ldots, R_{n-1}\}]$

The main problem is enumerating the pairs

Effect on Search Space

Absolute number of generated pairs

	Chain			Star		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	20	84	73	32	130	110
10	165	3,962	1,135	2,304	38,342	57,888
15	560	130,798	5,628	114,688	9,533,170	57,305,929
20	1,330	4,193,840	17,545	4,980,736	2,323,474,358	59,892,991,338
	Cycle			Clique		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	40	140	120	90	180	280
10	405	11,062	2,225	28,501	57,002	306,991
15	1,470	523,836	11,760	7,141,686	14,283,372	307,173,877
20	3,610	22,019,294	37,900	1,742,343,625	3,484,687,250	309,338,182,241

Enumerating Connected Subgraphs

- two steps: enumerate all connected subgraphs, enumerate disjoint but connected subgraphs for a given one $\Rightarrow$ pairs
- enumerate all pairs, enumerate no duplicates, enumerate for DP
- if (a, b) is enumerated, do not enumerate (b, a)
- requires total ordering of connected subgraphs
- preparation: label nodes breadth-first from 0 to $n - 1$

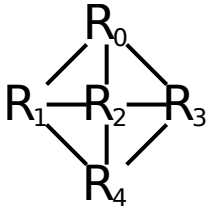
Preliminaries, given query graph $G = (V, E)$:

$$\begin{aligned} V &= \{v_0, \ldots, v_{n-1}\} \\ \mathcal{N}(V') &= \{v' | v \in V' \wedge (v, v') \in E\} \\ \mathcal{B}_i &= \{v_j | j \leq i\} \end{aligned}$$

Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

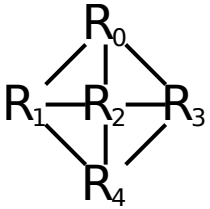


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

Choose all nodes as enumeration start node once

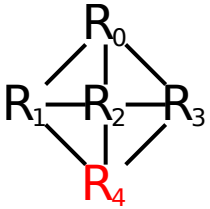


Enumerating Connected Subgraphs (2)

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

First emit only the node itself as subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

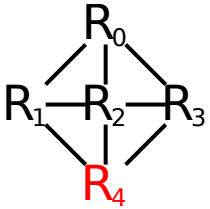


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

Then enlarge the subgraph recursively

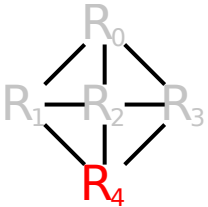


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

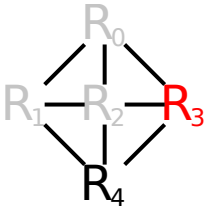
Prohibit nodes with smaller labels. Thus the set of valid nodes increases over time



Enumerating Connected Subgraphs (2)

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

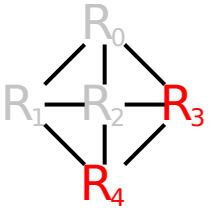
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

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

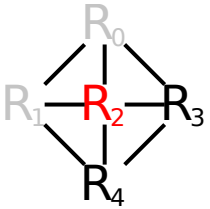
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

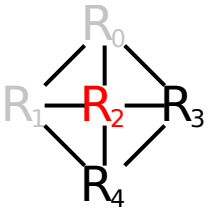


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

In each recursion, find all neighboring nodes that are not prohibited

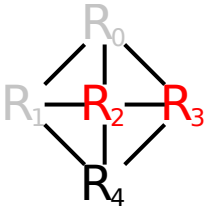


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

Add all combinations to the subgraph and emit the new subgraph

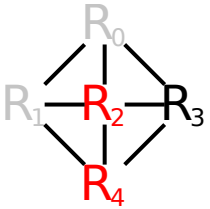


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

Add all combinations to the subgraph and emit the new subgraph

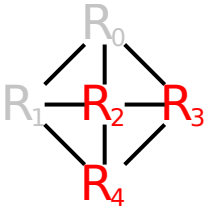


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

Add all combinations to the subgraph and emit the new subgraph

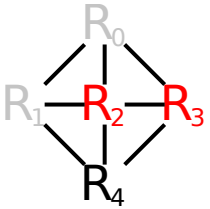


Enumerating Connected Subgraphs (2)

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

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
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, $(S \cup S')$, $(X \cup N)$);
}

Then, add all combinations to the subgraph and increase recursively

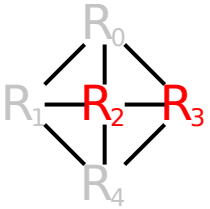


Enumerating Connected Subgraphs (2)

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

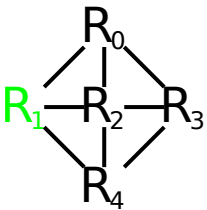
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
  EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

The neighborhood is prohibited during recursion, preventing duplicates



Enumerating Complementary Subgraphs

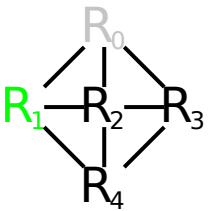
EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}



Enumerating Complementary Subgraphs

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

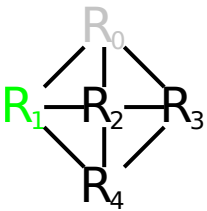
Prohibit all nodes that will be start nodes later on and the primary subgraph



Enumerating Complementary Subgraphs

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

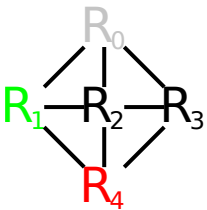
Find all neighboring nodes that are not prohibited



Enumerating Complementary Subgraphs

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

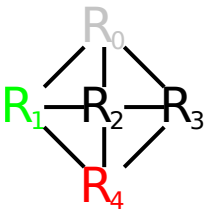
Consider each of the nodes



Enumerating Complementary Subgraphs

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

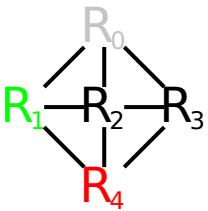
Choose the node as complementary subgraph and emit it



Enumerating Complementary Subgraphs

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

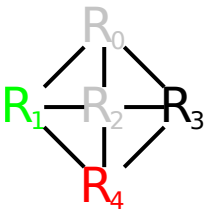
Recursively increase the subgraph re-using EnumerateCsgRec



Enumerating Complementary Subgraphs

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

Again prohibit nodes with a smaller label to prevent duplicates



Enumerating Complementary Subgraphs

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

- EnumerateCsg+EnumerateCmp produce all ccp
- resulting algorithm DPccp considers exactly #ccp pairs
- which is the lower bound for all DP enumeration algorithms

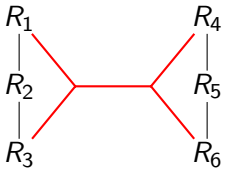
Remarks

- DPsize is good for chains, DPsub for cliques
- implementation of DPccp is more involved
- each enumeration step must be fast (ideally $O(1)$, at most $O(n)$, where n is the number of relations)
- but benefits are huge
- DPccg adopts to query graph structure
- considers minimal number of pairs
- especially for "in-between queries" (e.g. stars) much faster

Beyond (Regular) Query Graphs

Some queries are more complex

select *
from $R_1\ r_1,\ R_2\ r_2,\ R_3\ r_3,$
$R_4\ r_4,\ R_5\ r_5,\ R_6\ r_6$
where $r_1.a{=}r_2.a$ **and** $r_2.b{=}r_3.c$ **and**
$r_4.d{=}r_5.d$ **and** $r_5.e{=}r_6.e$ **and**
$\text{abs}(r_1.f + r_3.f)$
$= \text{abs}(r_4.g + r_6.g)$



- does not induce a graph but a hyper-graph
- graph based DP algorithm not directly applicable
- generic DP algorithms work, but not as efficient

Handling Hypergraphs

A *hypergraph* is a pair $H = (V, E)$ such that

1. V is a non-empty set of nodes and
2. E is a set of hyperedges, where a *hyperedge* is an unordered pair (u, v) of non-empty subsets of V ($u \subset V$ and $v \subset V$) with the additional condition that $u \cap v = \emptyset$.

Nodes in V are totally ordered via an (arbitrary) relation $\prec$.

- enumeration is performed by decreasing $\prec$
- $\prec$ orders the search space (DP order, duplicates)

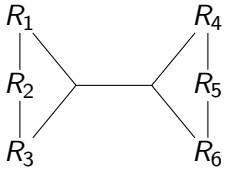
Handling Hypergraphs (2)

In principle same approach as for regular graphs:

- start with one node
- expand recursively by following edges

Problem:

- hyperedges are n:m edges
- where to expand to from $\{R_1, R_2, R_3\}$?
- must still guarantee DP order



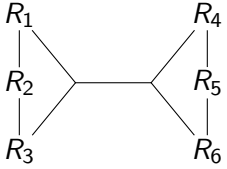
Handling Hypergraphs - Neighborhood

When computing the neighborhood, choose representatives:

- a hyperedge "leads" to the least node (regarding $\prec$)
- therefore $N(\{R_1, R_2, R_3\}) = \{R_4\}$
- ensures DP order (and prevents duplicates)

But:

- leads to (temporarily) disconnected graphs
- $\{R_1, R_2, R_3, R_4\}$ is not connected
- must expand further until R_6 reached



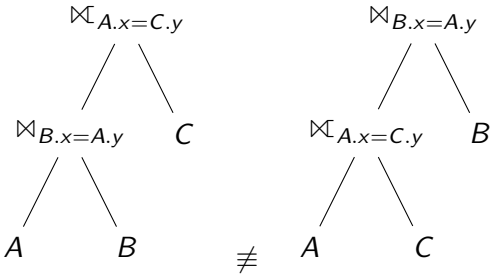
Requires checks for connectedness

- can exploit the DP table for cheap tests
- if it is connected, a DP entry must exist

Non-Inner Joins

Some queries use non-inner joins:

- either explicitly (*OUTER JOIN* etc.) or implicitly (unnesting etc.)
- are not freely reorderable



Must be taken into account during join ordering

Non-Inner Joins - Reordering Constraints

Examine pair-wise reorderings of operators

- for all $\circ_1, \circ_2$, check if $(R \circ_1 S) \circ_2 T \equiv R \circ_1 (S \circ_2 T)$
- assume syntax constraints are satisfied

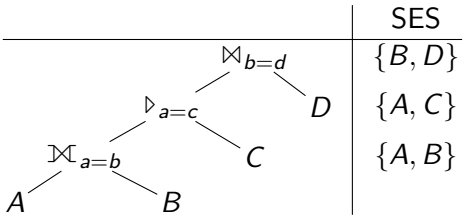
Gives a big compatibility matrix

	$\bowtie$	$\overrightarrow{\bowtie}$	$\overleftrightarrow{\bowtie}$	$\triangleright$	$\ltimes$	$\overrightarrow{\ltimes}$	...
$\bowtie$	+	+	-	+	+	+	...
$\overrightarrow{\bowtie}$	-	+	-	-	-	-	...
$\overleftrightarrow{\bowtie}$	-	+	+	-	-	-	...
$\triangleright$	-	-	-	-	-	-	...
$\ltimes$	-	-	-	-	-	-	...
$\overrightarrow{\ltimes}$	-	-	-	-	-	-	...
...							

Non-Inner Joins - TESs

Extract reordering constraints from operator tree in two steps:

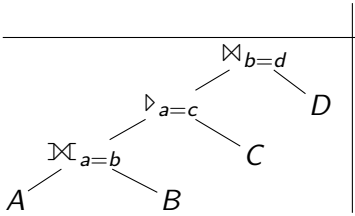
1. build the *syntactic eligibility set* (SES) for each operator
   - set of relations that has to be in the input



Non-Inner Joins - TESs

Extract reordering constraints from operator tree in two steps:

1. build the *syntactic eligibility set* (SES) for each operator
2. bottom up traversal, build the *total eligibility set* (TES)
   - initialize TES with SES
   - check for conflicts with other operators (can be in subtrees!)
   - if conflict, add other TES to own TES



	SES	TES
$\bowtie_{b=d}$	$\{B, D\}$	$\{A, B, D\}$
$\triangleright_{a=c}$	$\{A, C\}$	$\{A, B, C\}$
$\bowtie_{a=b}$	$\{A, B\}$	$\{A, B\}$

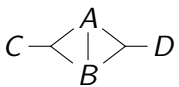
TESs capture reordering restrictions by requiring relations, which imply operators.

Non-Inner Joins - Using TESs

Add the TES to the join edge

- operator "requires" certain relations, so encode it like this
- constructs hyperedges (n:m)
- eliminates invalid reorderings from the search space

Original query graph from previous example: $C - A - B - D$

After adding TESs to the edges: 

Simplifying the Query Graph

The graph-based DP algorithm considers the minimal number of join-pairs

- we therefore cannot expect to get a better runtime for exact solutions
- many problems can be solved exactly, but not all
- depends on the structure of the query graph
- chains are simple, others, e.g., stars, are hard
- how to cope with these queries?

Greedy heuristics would work, but results are much worse than DP solutions.

Simplifying the Query Graph - General Idea

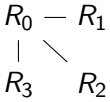
If the problem is too complex to solve exactly, simplify the query graph until it gets tractable.

- the query graph describes all join possibilities
- by modifying the query graph we can rule out some possibilities
- this reduces the search space and the optimization time
- we prefer modifications that are "safe"
- uses greedy steps only for the "easy" problems, then use DP

Note: "simplifying" means simpler for the optimizer.
For a human the query graph tends to get strange.

Simplifying a Star Query

graph

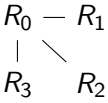
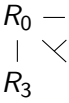


joins

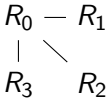
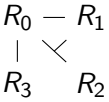
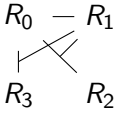
$R_0 \bowtie R_1$
$R_0 \bowtie R_2$
$R_0 \bowtie R_3$
original

search space size
6

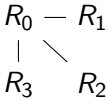
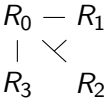
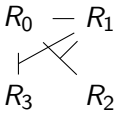
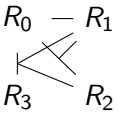
Simplifying a Star Query

graph			
joins	$R_0 \bowtie R_1$ $R_0 \bowtie R_2$ $R_0 \bowtie R_3$ original	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $R_0 \bowtie R_3$ 1st step	search space size ~~6~~ 3

Simplifying a Star Query

graph			
joins	$R_0 \bowtie R_1$ $R_0 \bowtie R_2$ $R_0 \bowtie R_3$ original	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $R_0 \bowtie R_3$ 1st step	search space size ~~6~~ ~~3~~ 2
graph			
joins	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $\{R_0, R_1\} \bowtie R_3$ 2nd step		

Simplifying a Star Query

			search space size
graph			
joins	$R_0 \bowtie R_1$ $R_0 \bowtie R_2$ $R_0 \bowtie R_3$ original	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $R_0 \bowtie R_3$ 1st step	~~6~~ ~~3~~ ~~2~~ 1
graph			
joins	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $\{R_0, R_1\} \bowtie R_3$ 2nd step	$R_0 \bowtie R_1$ $\{R_0, R_1\} \bowtie R_2$ $\{R_0, R_1, R_2\} \bowtie R_3$ 3rd step	

Performing A Simplification Step

Given a query graph $G = (V, E)$

1. examine all joins $\bowtie_1, \bowtie_2 \in E$ that are *neighboring*
   - neighboring $\approx$ have a relation in common (see [6])
2. make sure that $\bowtie_2$ could be ordered before $\bowtie_1$
   - checks for contradictions, requires a fast cycle checker
3. compute the *orderingBenefit*($\bowtie_1$,$\bowtie_2$)
   - this is the heuristical part, different benefit heuristics could be used
4. retain the $S_1^L \bowtie_1 S_1^R, S_2^L \bowtie_2 S_2^R$ with the maximal orderingBenefit
   - maintain priority queues to speed up repeated simplification
5. return $G' = (V, E \setminus \{\bowtie_1\} \cup \{(S_1^L \cup S_2^L \cup S_2^R) \bowtie_1 S_1^R\})$

The resulting query graph is more restrictive, i.e., simpler.

(there are more cases due to different possible ways of neighboring)

Estimating the Ordering Benefit

We want to prefer orderings that are almost certainly a good idea.
Therefore one approach is to maximize

$$\text{orderingBenefit}(X \bowtie_1 R_1, X \bowtie_2 R_2) = \frac{C((X \bowtie_1 R_1) \bowtie_2 R_2)}{C((X \bowtie_2 R_2) \bowtie_1 R_1)}$$

If we cannot compute C due to missing information, use C_{out}.

Adjusting the Problem Complexity

How much should we simplify?

- until optimization fits into resource constraints (memory or time)

How do we know when to stop simplifying?

- count the number of **connected subgraphs** of the query graph
- directly determines memory, indirectly optimization time
- stop counting when the limit is reached

Counting is fast, but not instantaneous

- counting 10,000 subgraphs in a query with 100 relations took $\approx$ 5ms
- we cannot do this after every simplification

Exact limit depends on hardware, a reasonable choice is 10,000 connected subgraphs.

Full Optimization Algorithm

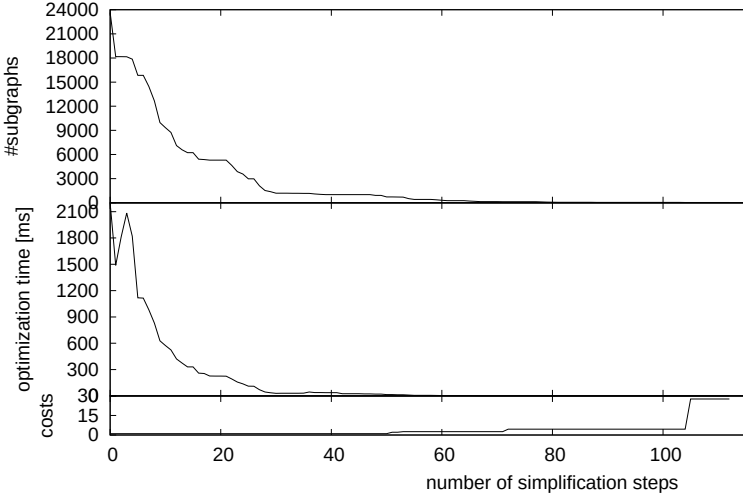
Given a Query Graph $G = (V, E)$ and a complexity budget b

1. compute a list $\bar{G}$ of query graphs
   - repeatedly call the simplification step, stop when no change
2. perform binary search over $\bar{G}$, find G_b
   - for the current element G', $c = \#$connected subgraphs in G' (count at most $b + 1$)
   - if $c > b$ increase, otherwise decrease
3. return $DPhyp(G_b)$

Simplifies as much as needed to meet the constraints, than uses DP.

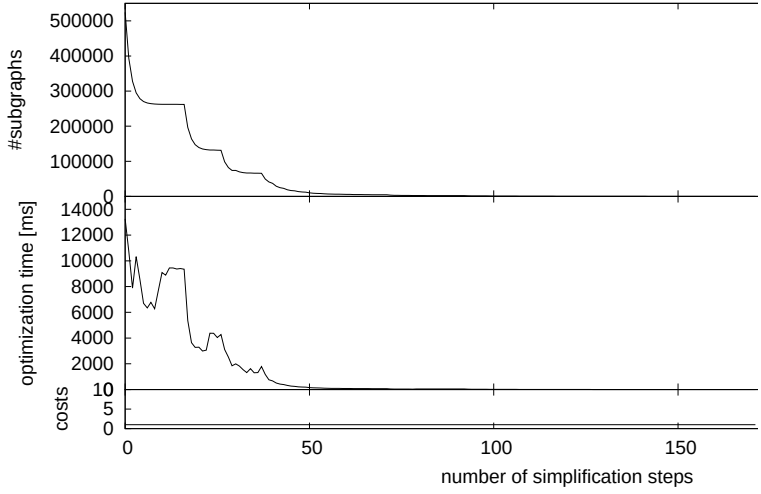
(the algorithm does not materialize $\bar{G}$ explicitly, see [6])

Time/Quality Trade-off - Grid with 20 Relations



- as expected plan quality degrades at some point

Time/Quality Trade-off - Star with 20 Relations

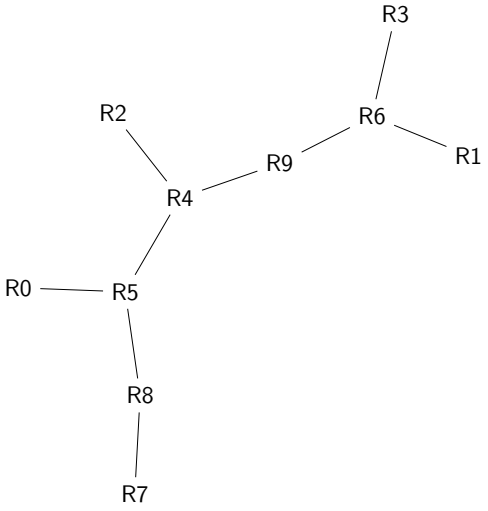


- same optimization time behavior, but plan quality remains perfect

Adaptive Optimization using Search Space Linearization

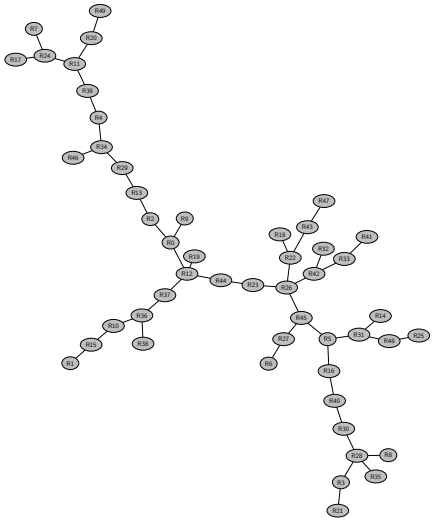
- not all join problems are equal
- most queries are small, but we have a incredible long tail
- must handle all of them reasonably, with the correct expectations
- adapt the algorithm to the query complexity

Join ordering: Solved!

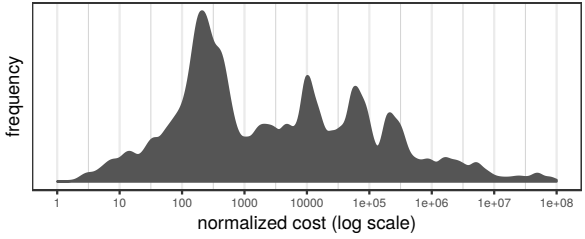


- *Dynamic Programming (DP)*, pioneered by Selinger et al. (1979)
- Large body of follow-up work
  - bushy plans
  - graph awareness
  - non-inner joins
  - top-down formulations
- *Exponential* runtime in general
- Only viable for relatively *small queries*
- Generated queries we are increasingly faced with tend to be *too large*

Solved?

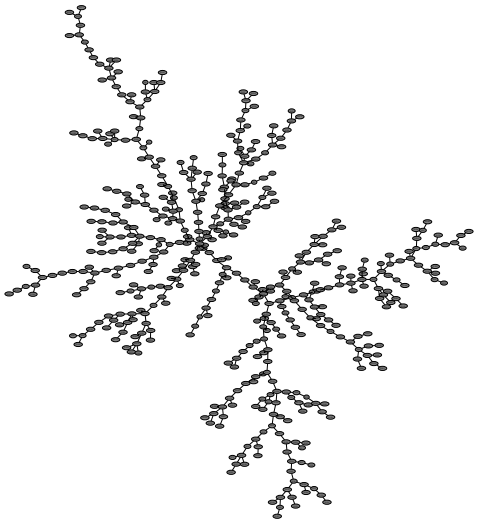


- *Huge* search space (NP-Hard)
- Too hard to solve optimally
- *Heuristics* to the rescue!?

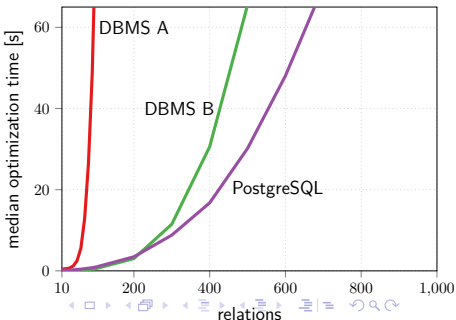


- *Suddenly*, if just slightly too large
- Likely to result in *disastrous plans*
- Not the end of the spectrum

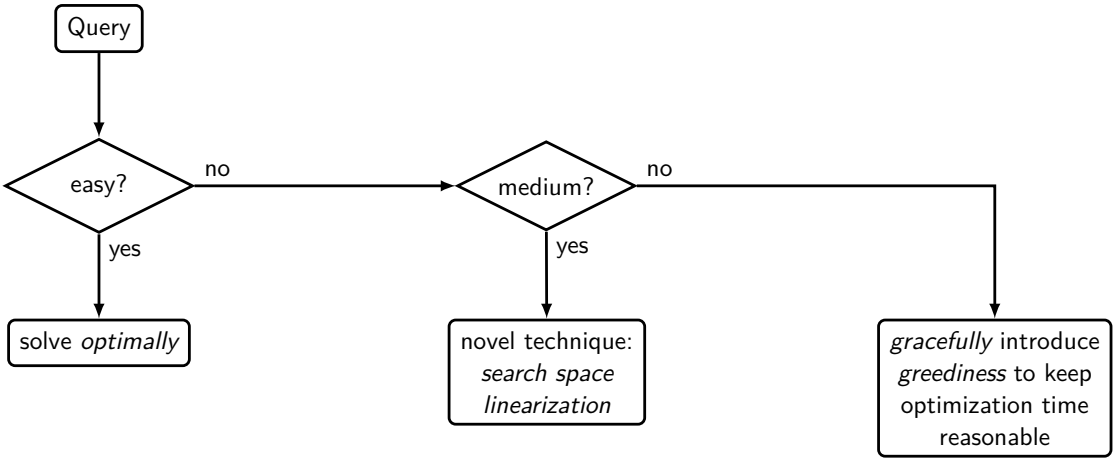
Unsolved!



- *Tableau*: "Get Real: How Benchmarks Fail to Represent the Real World" (DBTEST 2018)
- Queries touching a few *hundred relations* are quite *common*
- *SAP*: 4,598 relations (BTW 2017)

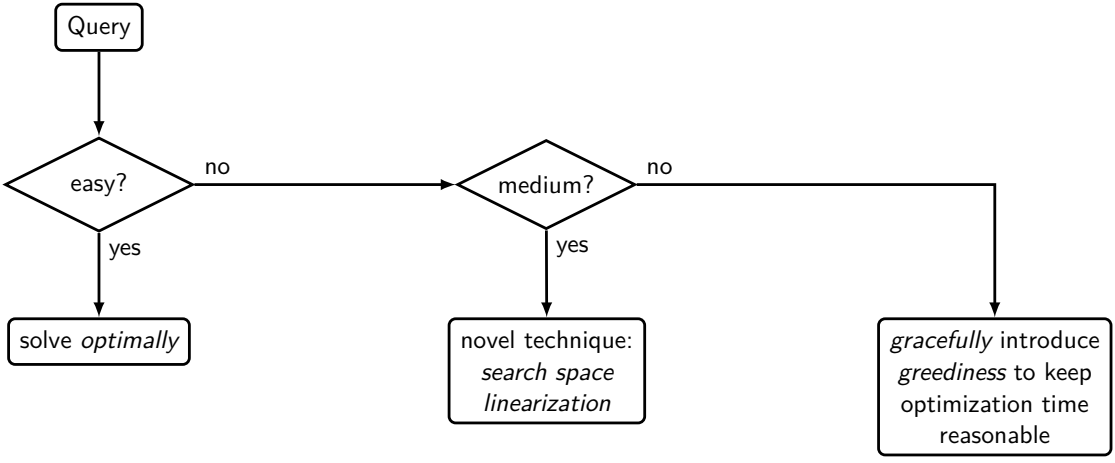


Adaptive Optimization – The Big Picture

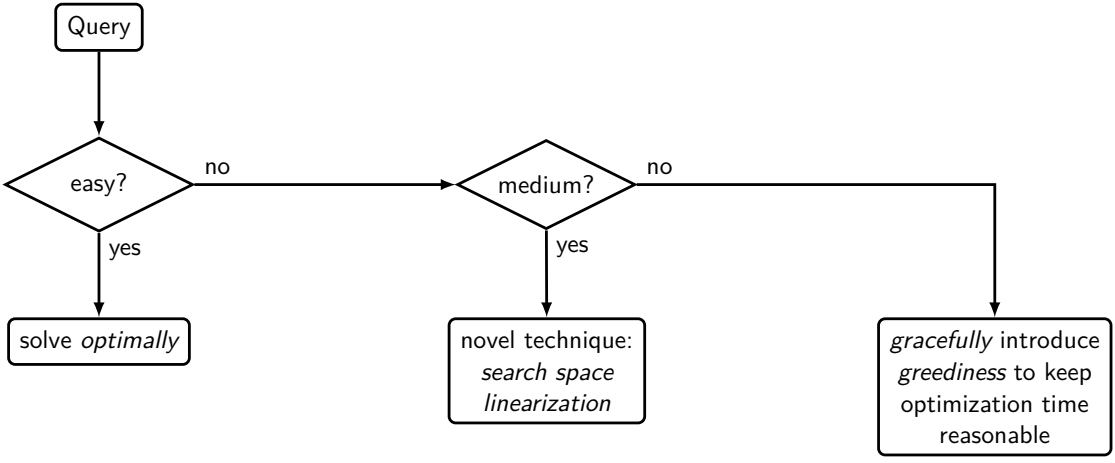


- For *performance* and *correctness* reasons: no cross products

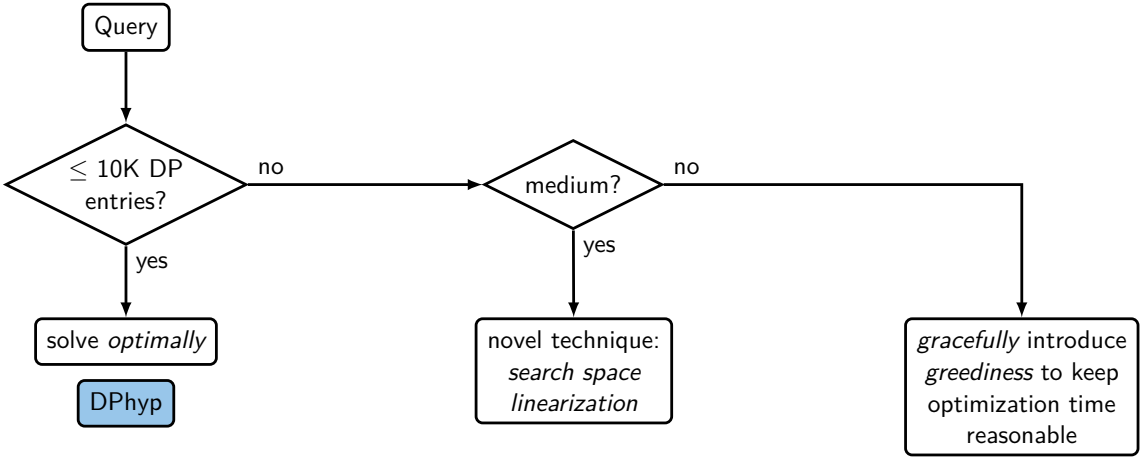
Adaptive Optimization – The Big Picture



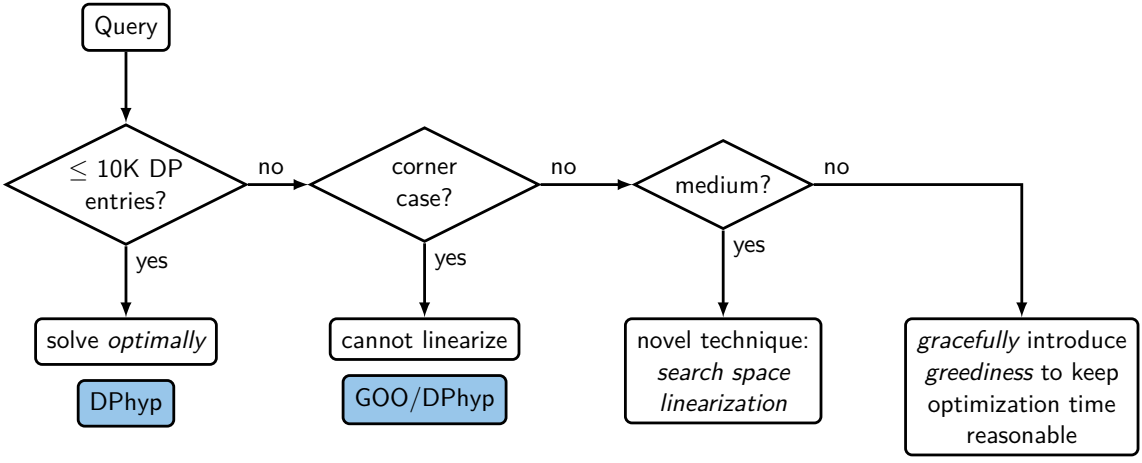
Adaptive Optimization – The Big Picture



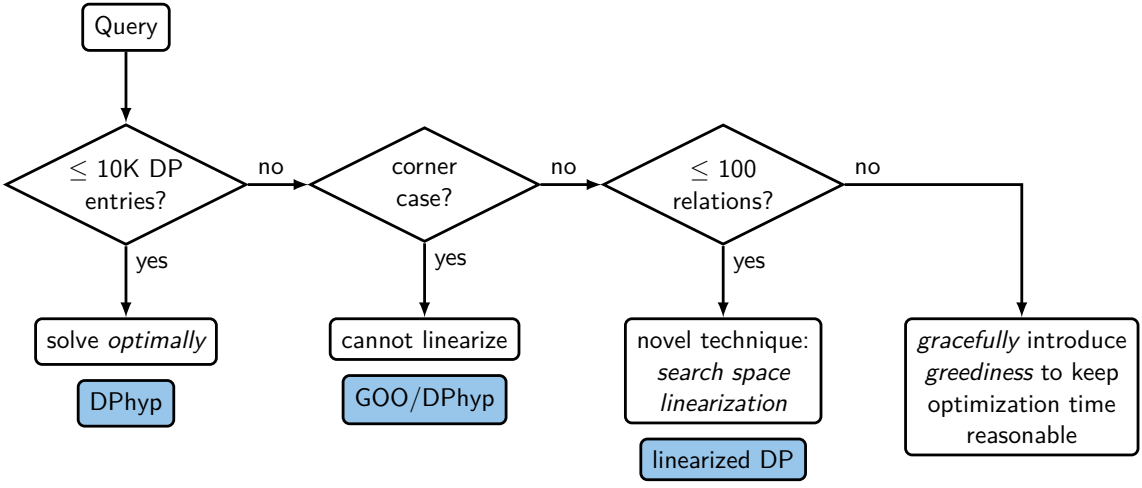
Adaptive Optimization – The Big Picture



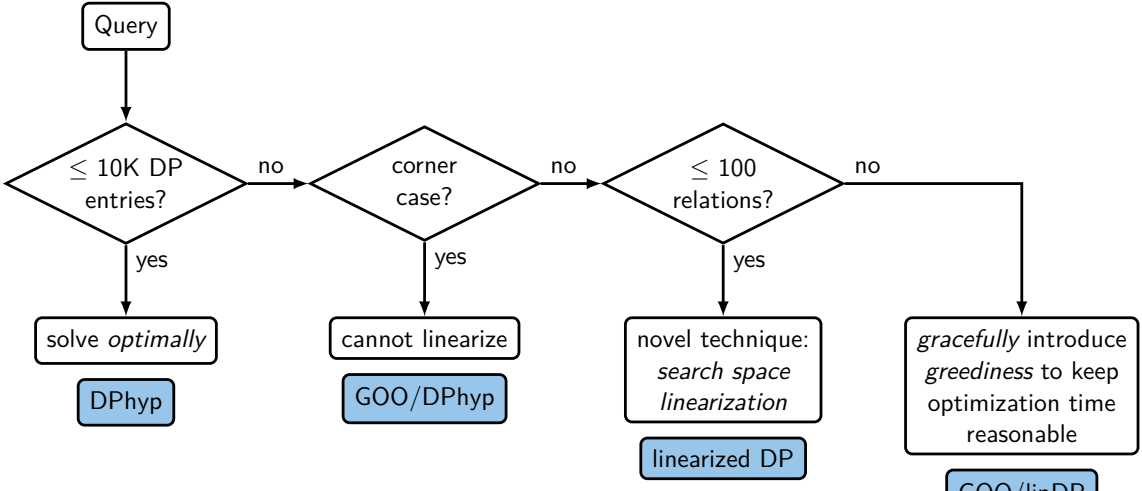
Adaptive Optimization – The Big Picture



Adaptive Optimization – The Big Picture



Adaptive Optimization – The Big Picture



Adaptive Optimization – How to Measure Complexity

Structure	DP complexity	DP table size
chain	$\mathcal{O}(n^3)$	n^2
clique	$\mathcal{O}(3^n)$	2^n

- Complexity depends on the *structure* of the query graph
- *Size of DP table* as measure of complexity
- *Analyze query graph* to determine the size of the DP table

Adaptive Optimization – Small Queries

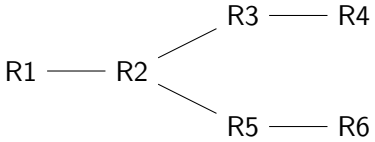
- Up to 10,000 DP entries
  - chains: up to 100 relations
  - cliques: less than 14 relations
- Run *DPhyp*
  - *Adapts* to the query graph's structure
  - *Completely* and *minimally* enumerates all possibly optimal join orders without cross products
- Plan guaranteed to be *optimal*
- Optimization will be *fast*

Adaptive Optimization – Medium Queries

- Complexity depends on the *structure* of the query graph
- Can easily optimize *chain queries* on 100 relations exactly (polynomial runtime)
- Usually queries are not exactly linear
- Still benefit from this *fast optimization* through *search space linearization*

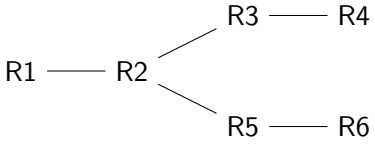
Adaptive Optimization – Search Space Linearization

- Assume the order of relations in the optimal plan is known



Adaptive Optimization – Search Space Linearization

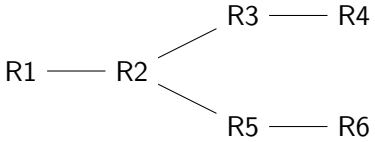
- Assume the order of relations in the optimal plan is known



R2 R1 R3 R5 R6 R4

Adaptive Optimization – Search Space Linearization

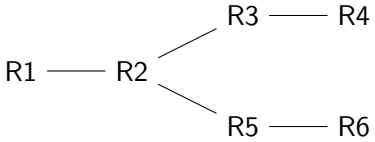
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size



R2 R1 R3 R5 R6 R4

Adaptive Optimization – Search Space Linearization

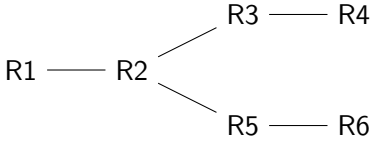
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

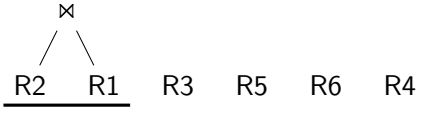


R2 R1 R3 R5 R6 R4

Adaptive Optimization – Search Space Linearization

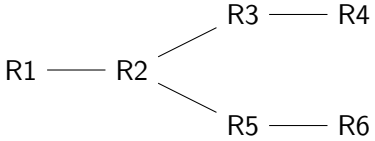
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

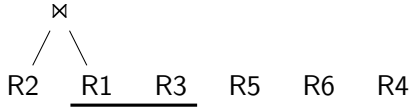




Adaptive Optimization – Search Space Linearization

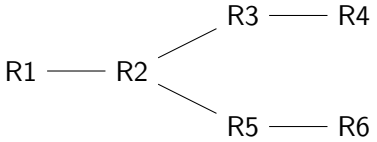
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

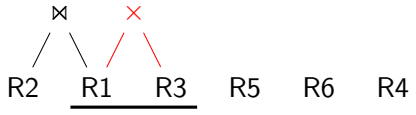




Adaptive Optimization – Search Space Linearization

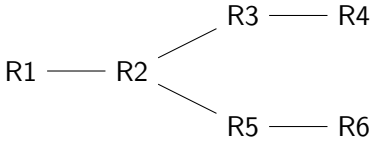
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

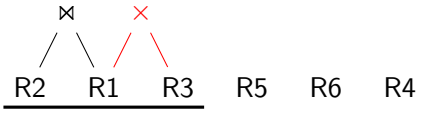




Adaptive Optimization – Search Space Linearization

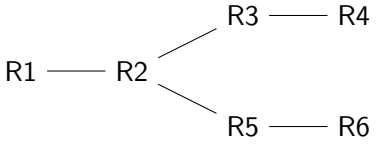
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

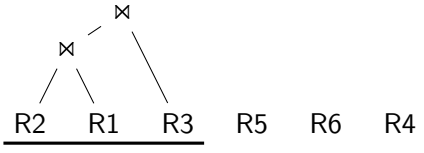




Adaptive Optimization – Search Space Linearization

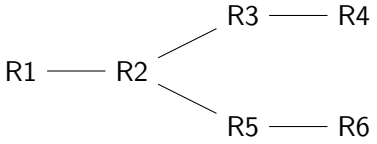
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

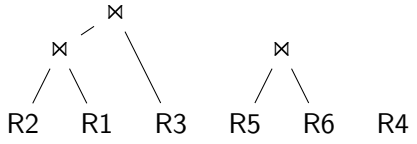




Adaptive Optimization – Search Space Linearization

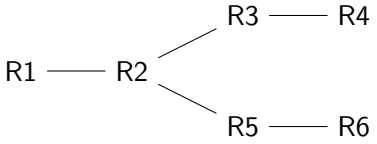
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

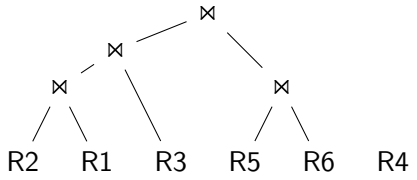




Adaptive Optimization – Search Space Linearization

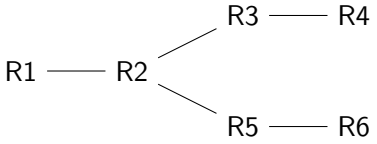
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

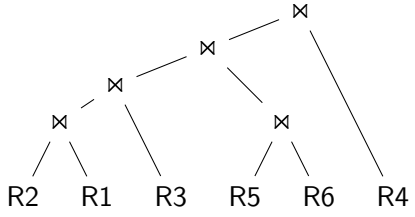




Adaptive Optimization – Search Space Linearization

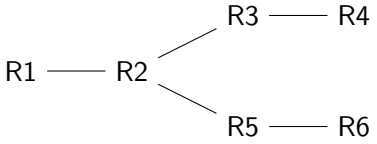
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size

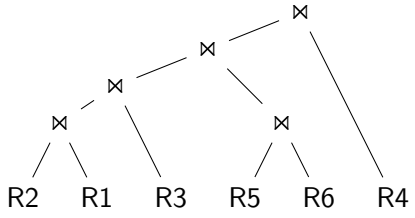




Adaptive Optimization – Search Space Linearization

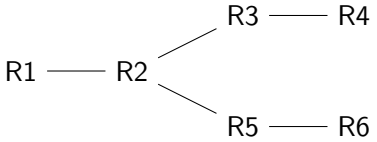
- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size
- But: how to know the optimal order?

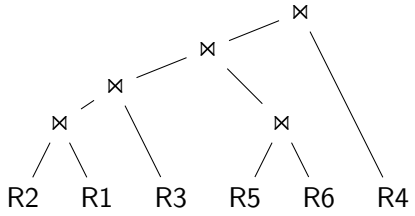




Adaptive Optimization – Search Space Linearization

- Assume the order of relations in the optimal plan is known
- Polynomial DP algorithm to generate optimal plan from this *linearization*
- Optimally combine optimal solutions for subchains of increasing size
- But: how to know the optimal order?
- *IKKBZ* (TODS 3/'84, VLDB '86): Optimal left-deep plan in $\mathcal{O}(n^2)$
- Good alternative to the optimal relative order of relations





Adaptive Optimization – Linearized DP

Procedure

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

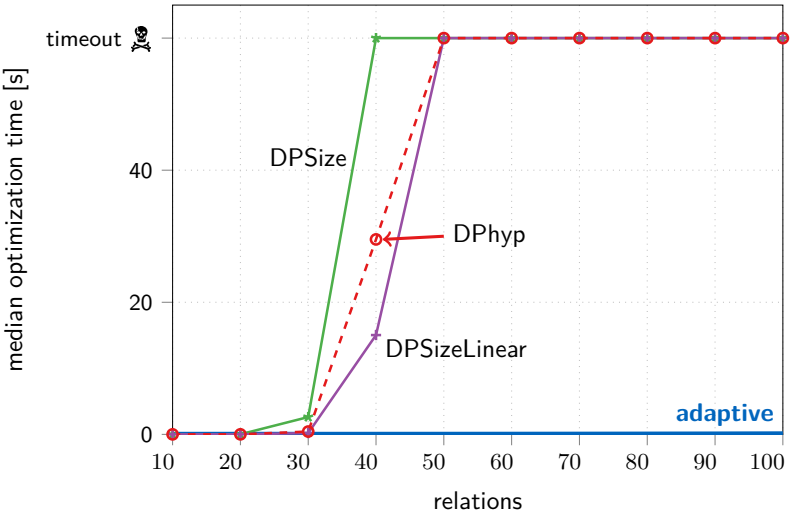
Properties

- Runs in $\mathcal{O}(n^3)$
- Result *at least as good as* the optimal left-deep plan
- With proper linearization, discovers globally *optimal bushy plan*

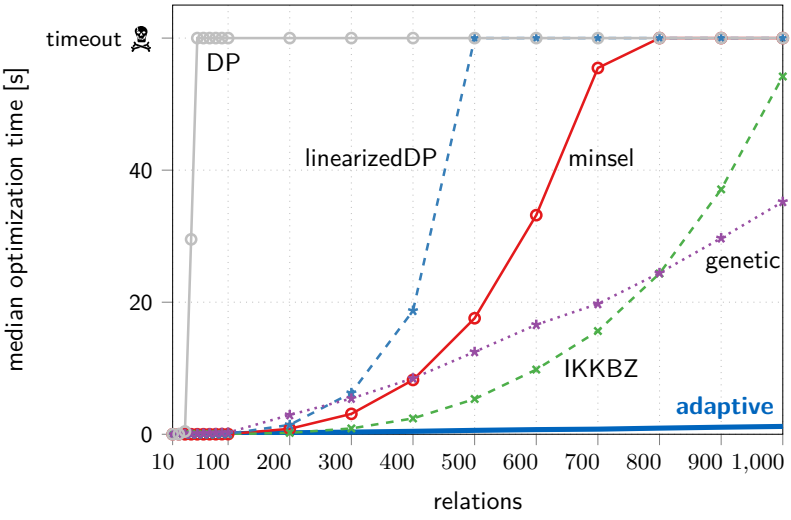
Adaptive Optimization – Large Queries

- Even linearized DP too expensive for the most *complex queries*
- Iterative Dynamic Programming (Kossmann & Stocker, TODS 1/2000):
  1. *Greedily build* query plan, e.g. using Greedy Operator Ordering (GOO)
  2. *Iteratively refine* by optimizing the most expensive sub trees of size k using DP
- Linearization greatly *increases reordering freedom*
  - originally: $k \approx 7$
  - *linearized:* $k = 100$

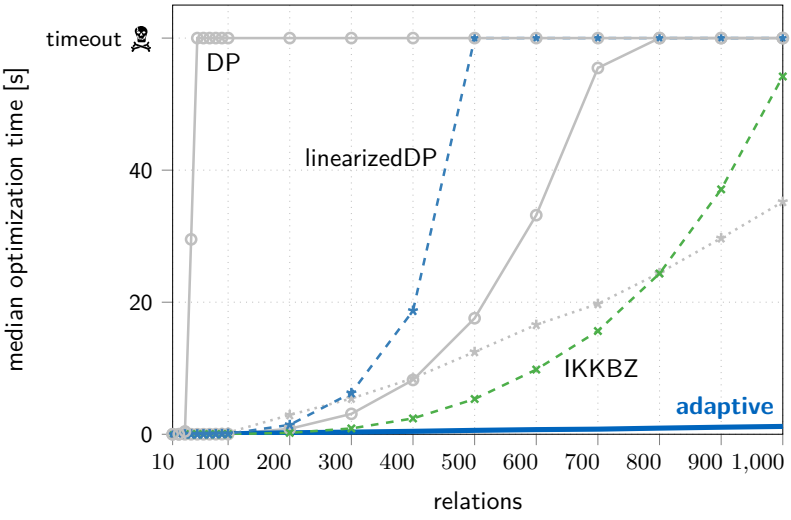
Generated Queries – Optimization Time



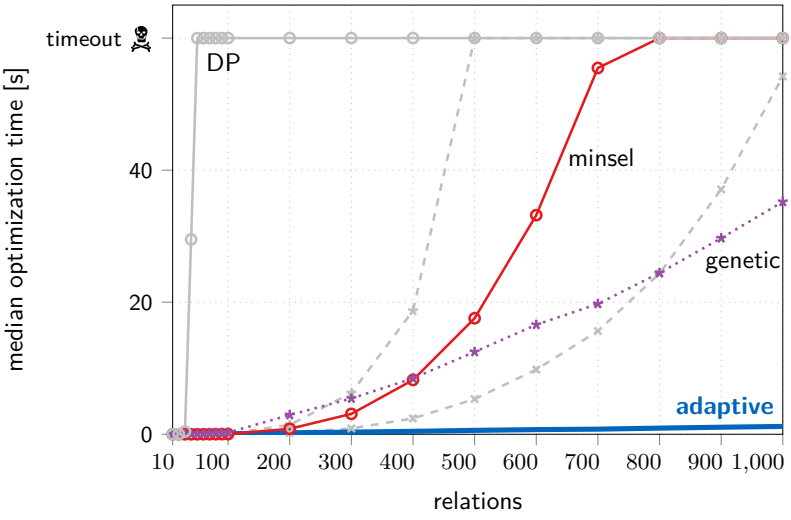
Generated Queries – Optimization Time



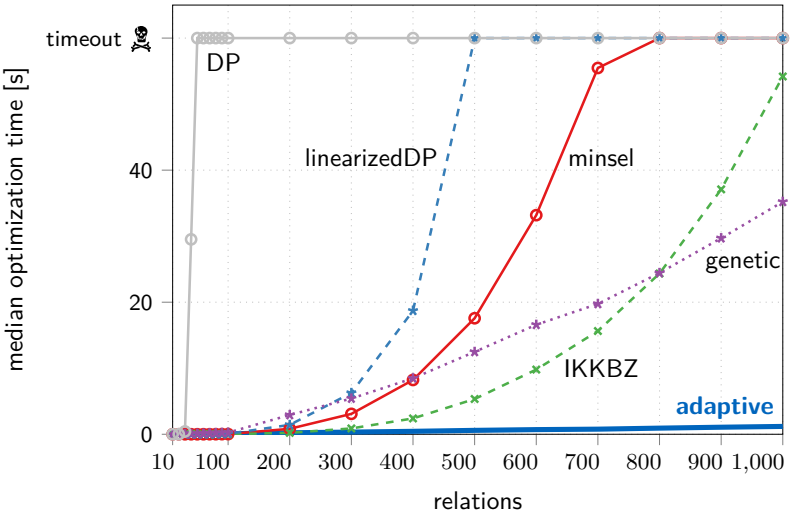
Generated Queries – Optimization Time



Generated Queries – Optimization Time



Generated Queries – Optimization Time



Generated Queries – Plan Quality

Plan cost compared to cost of best plan found by any of the algorithms

Optimal plan known (371 queries)

Algorithm	median	95%	max
DPhyp	1.00	1.00	1.00
Linearized DP	1.00	1.23	2.23
adaptive	1.00	1.10	2.23

- Most of the plans generated by linearized DP are optimal or near-optimal
- Adaptive Optimization additionally benefits from full DPhyp as long as it is fast

Generated Queries – Plan Quality

Linearized DP ($\leq$ 100 relations; 1,000 queries)

Algorithm	median	95%	max
IKKBZ	1.00	1.97	58.47
Linearized DP	1.00	1.12	2.57
adaptive	1.00	1.07	2.57

- DP phase in linearized DP significantly increases plan quality

Generated Queries – Plan Quality

Iterative Dynamic Programming ($\leq$ 5,000 relations; 2,300 queries)

Algorithm	median	95%	max
GOO	1.05	2.81	19.18
GOO/DPhyp	1.01	2.53	19.18
GOO/linDP	1.00	1.60	4.02
adaptive	1.00	1.59	4.02

- Iterative DP benefits from additional freedom induced by linearized DP
- Adaptive Optimization generates good plans across the whole spectrum of queries

Generating Permutations

The algorithms so far have some drawbacks:

- greedy heuristics only heuristics
- will probably not find the optimal solution
- DP algorithms optimal, but very heavy weight
- especially memory consumption is high
- find a solution only after the complete search

Sometimes we want a more light-weight algorithm:

- low memory consumption
- stop if time runs out
- still find the optimal solution if possible

Generating Permutations (2)

We can achieve this when only considering left-deep trees:

- left-deep trees are permutations of the relations to be joined
- permutations can be generated directly
- generating all permutations is too expensive
- but some permutations can be ignored:
  Consider the join sequence $R_1 R_2 R_3 R_4$. If we know that $R_1 R_3 R_2$ is cheaper than $R_1 R_2 R_3$, we do not have to consider $R_1 R_2 R_3 R_4$.

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

Recursive Search

ConstructPermutations(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep join tree
$B = \epsilon$
$P = \epsilon$
for each $R_i \in R$ $\{$
  ConstructPermutationsRec($P \circ < R_i >, R \setminus \{R_i\}, B$)
$\}$ **return** B

- algorithm considers a prefix P and the rest R
- keeps track of the best tree found so far B
- increases the prefix recursively

Recursive Search (2)

ConstructPermutationsRec(P, R, B)
Input: a prefix P, remaining relations R, best plan B
Output: side effects on B

if $|R| = 0$ {
  if $B = \epsilon \vee C(B) > C(P)$ {
    $B = P$
  }
} **else** {
  for each $R_i \in R$ {
    if $C(P \circ < R_i >) \leq C(P[1 : |P| - 1] \circ < R_i, P[|P|] >)$ {
      ConstructPermutationsRec($P \circ < R_i >, R \setminus \{R_i\}, B$)
    }
  }
}

Remarks

Good:

- linear memory
- immediately produces plan alternatives
- anytime algorithm
- finds the optimal plan eventually

Bad:

- worst-case runtime if ties occur
- worst-case runtime if no ties occur is an open problem

Often fast, can be stopped anytime, but may perform poorly.

Transformative Approaches

Main idea: [7]

- use equivalences directly (associativity, commutativity)
- would make integrating new equivalences easy

Problems:

- how to navigate the search space
- equivalences have no order
- how to guarantee finding the optimal solution
- how to avoid exhaustive search

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

Basic Algorithm

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 |ToDo| > 0 {
    T = an arbitrary tree in ToDo
    ToDo = ToDo $\setminus T$;
    Done = Done $\cup \{T\}$;
    Trees = ApplyTransformations(T);
    for each $T \in$ Trees {
        if $T \notin$ ToDo $\cup$ Done
            ToDo = ToDo $\cup \{T\}$
    }
}

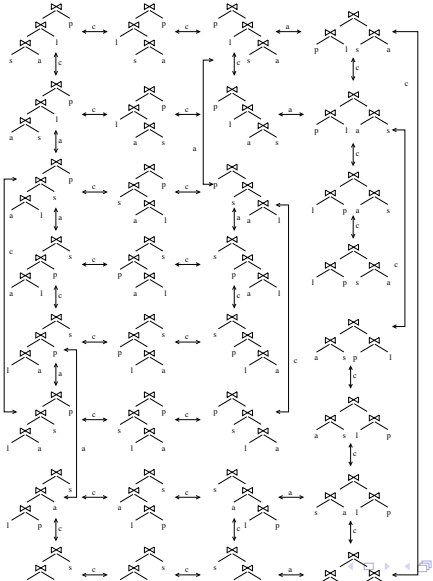
Basic Algorithm (2)

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 each $S \in$ Subtrees $\{$
    if S is of the form $S_1 \bowtie S_2$
        Trees $=$ Trees $\cup \{S_2 \bowtie S_1\}$
    if S is of the form $(S_1 \bowtie S_2) \bowtie S_3$
        Trees $=$ Trees $\cup \{S_1 \bowtie (S_2 \bowtie S_3)\}$
    if S is of the form $S_1 \bowtie (S_2 \bowtie S_3)$
        Trees $=$ Trees $\cup \{(S_1 \bowtie S_2) \bowtie S_3\}$
$\}$
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

Search Space



Introducing the Memo Structure

A memoization strategy is used to keep the runtime reasonable:

- 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 next slide).

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

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

Memoizing Algorithm

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 $\arg\min_{T \in \text{class } \{R_1, \ldots, R_n\}} C(T)$

- stored an arbitrary join tree in the memo structure
- explores alternatives recursively

Memoizing Algorithm (2)

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
}

- considers all alternatives within one class
- transformations themselves are done in ApplyTransformation2

Memoizing Algorithm (3)

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));
for each transformation $\mathcal{T}$ and class member of child classes {
    for each T' resulting from applying $\mathcal{T}$ to T {
        if T' not in MEMO structure {
            add T' to class $\mathcal{C}$ of MEMO structure
        }
    }
}

- first explores subtrees
- then applies all known transformations to the tree
- stores new trees in the memo structure

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.

Rule Set RS-1

T_1: Commutativity $C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1$

Disable all transformations T_1, T_2, and T_3 for $\bowtie_1$.

T_2: Right Associativity $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$

Disable transformations T_2 and T_3 for $\bowtie_2$ and enable all rules for $\bowtie_3$.

T_3: Left associativity $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3$

Disable transformations T_2 and T_3 for $\bowtie_3$ and enable all rules for $\bowtie_2$.

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

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 Left 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

Generating a random join tree is quite useful:

- allows for cost sampling
- randomized optimization procedures
- basis for Simulated Annealing, Iterative Improvement etc.
- easy with cross products, difficult without
- we consider with cross products first

Main problems:

- generating all join trees (potentially)
- creating all with the same probability

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 each $k \in [0, n[$ **descending**
  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 $r_{n-1}, \ldots, r_0$ where $0 \leq r_i \leq i$.
- Thus, there are exactly $n(n-1)(n-2)\ldots 1 = n!$ 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 $r_{n-1}, \ldots, r_0$.
  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.

Generating Random Permutations

Unrank(n, r)
Input: the number n of elements to be permuted
and the rank r of the permutation to be constructed
Output: a permutation π
for each $0 \leq i < n$
  $\pi[i] = i$
for each $n \geq i > 0$ **descending** $\{$
  swap($\pi[i-1]$, $\pi[r \mod i]$)
  $r = \lfloor r/i \rfloor$
$\}$
return π;

Generating Random Bushy Trees with Cross Products

Steps of the algorithm:

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

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

Tree Encoding

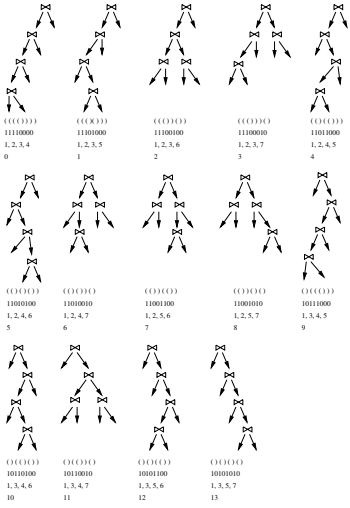
- Preordertraversal:
  - Inner node: '('
  - Leaf Node: ')'

  Skip last leaf node.
- Replace '(' by 1 and ')' by 0
- Just take positions of 1s.

Example: all trees with four inner nodes:

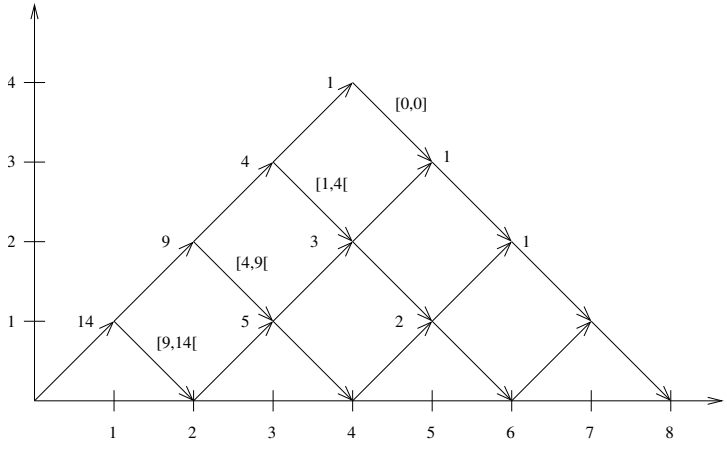
- The ranks are in $[0, 14[$

Tree Ranking Example



Unranking Binary Trees

We establish a 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.

Generating Bushy Trees

UnrankTree(n, r)
Input: a number of inner nodes n and a rank $r \in [0, C(n)[$
Output: encoding of the inner leafes of a tree
open $= 1$, close $= 0$
pos $= 2$, encoding $= < 1 >$
while $|\text{encoding}| < n$ {
  $k = q$(open+close+1,open-close+1)
  if $k \leq r$ {
    $r = r - k$, close=close+1
  } **else** {
    encoding=encoding$\circ < \text{pos} >$, open=open+1
  }
  pos=pos+1
}
return encoding

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

[8]

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.

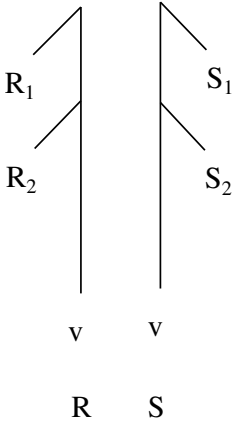
The Procedure

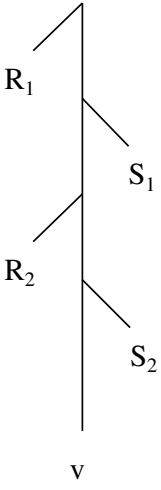
1. List merges (notation, specification, counting, unranking)
2. Join tree construction: leaf-insertion and tree-merging
3. Standard Decomposition Graph (SDG): describes all valid join trees
4. Counting
5. Unranking algorithm

List Merge

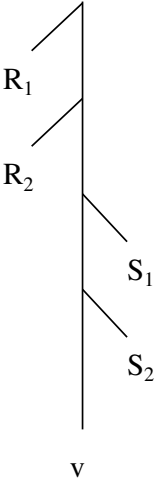
- Lists: Prolog-Notation: $< a|t >$
- Property P on elements
- A list l' is the *projection* of a list L on P, if L' contains all elements of L satisfying the property P.
  Thereby, the order is retained.
- A list L is a *merge* of two disjoint lists L_1 and L_2, if L contains all elements from L_1 and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

Example

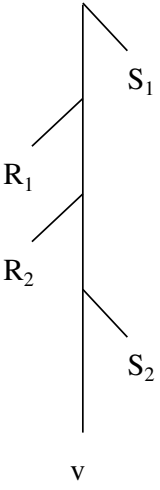




(R, S, [1, 1, 0])



(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

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	$[36, 70]$

Decomposition

UnrankDecomposition(r, l_1, l_2)
Input: a rank r, two list sizes l_1 and l_2
Output: encoding of the inner leafes of a tree
alpha $= <>$, $k = 0$
while $l_1 > 0 \wedge l_2 > 0$ {
  $m = M(k, l_2 - 1)$
  if $r \leq m$ {
    alpha $=$ alpha $\circ < l_1 - k >$
    $l_1 = k, k = 0, l_2 = l_2 - 1$
  } **else** {
    $r = r - m, k = k + 1$
  }
}
return alpha$\circ < l_1 > \circ \bigcirc_{1 \leq i \leq l_2} < 0 >$

Anchored List Representation of Join Trees

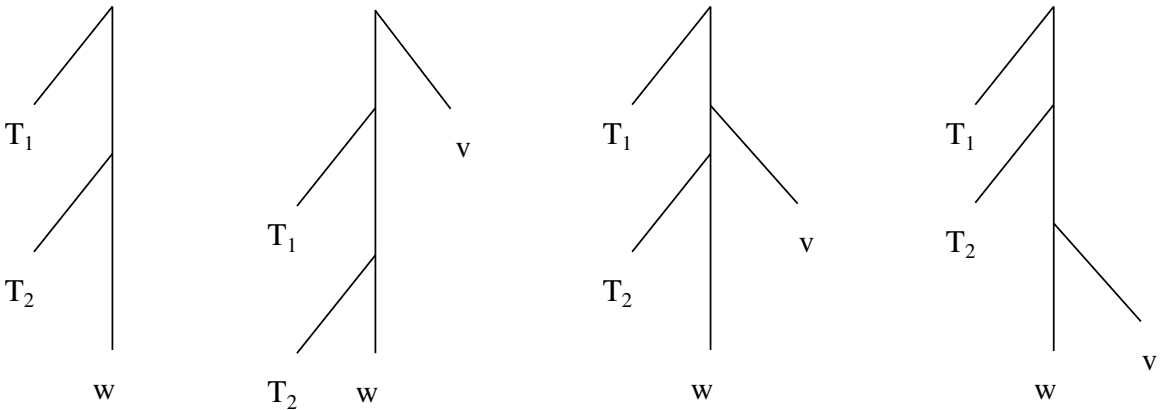
Definition Let T be a join tree and v be a leaf of T. The *anchored list representation* L of T is constructed as follows:

- If T consists of the single leaf node v, then $L =<>$.
- If $T = (T_l \bowtie T_2)$ and without loss of generality v occurs in T_2, then $L =< T_1|L_2 >$ 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

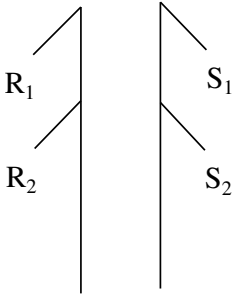
Definition Let $G = (V, E)$ be a query graph, T a join tree of G. $v \in V$ be such that $G' = G|_{V \setminus \{v\}}$ is connected, $(v, w) \in E$, $1 \leq k < n$, and

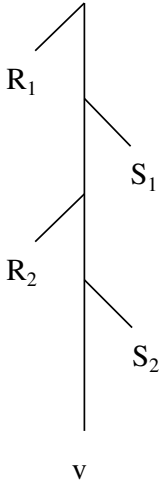
$$\begin{aligned} T &= (< T_1, \ldots, T_{k-1}, v, T_{k+1}, \ldots, T_n >, w) \\ T' &= (< T_1, \ldots, T_{k-1}, T_{k+1}, \ldots, T_n >, w). \end{aligned}$$

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.

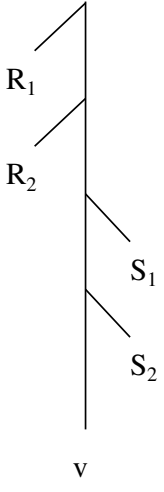
Observation: Leaf-insertion defines a bijective mapping between $\mathcal{T}_G^{v(k)}$ and insertion pairs (T', k) on v, where T' is an element of the disjoint union $\cup_{i=k-1}^{n-2} \mathcal{T}_{G'}^{w(i)}$.

Tree-Merging: Example

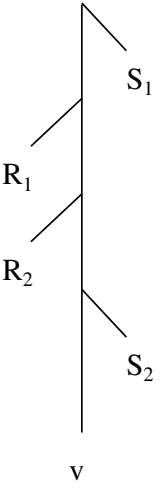




(R, S, [1, 1, 0])



(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

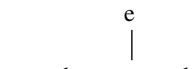
The standard decomposition graph of a query graph $G = (V, E)$ is constructed in three steps:

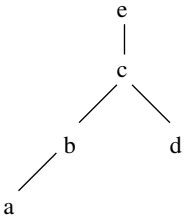
1. pick an arbitrary node $r \in V$ as its root node
2. transform G into a tree G' by directing all edges away from r;
3. call `QG2SDG`(G', r)

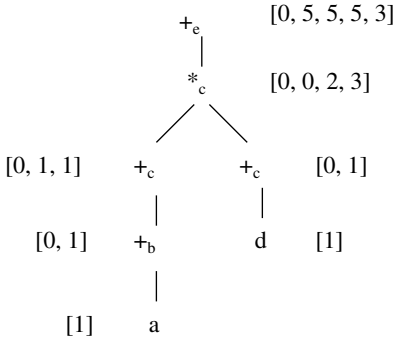
Constructing a Standard Decomposition Graph (2)

QG2SDG(G', v)
Input: a query tree $G' = (V, E)$ and its root v
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 $+_v$
    $E = E \setminus \{(v, w_i) | 1 \leq i \leq n\}$
    $E = E \cup \{(v, l), (v, r), (l, w_1)\} \cup \{(r, w_i) | 2 \leq i \leq n\}$
    QG2SDG(G', l), QG2SDG(G', r)
} **return** G'

Constructing a Standard Decomposition Graph (3)







Counting

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

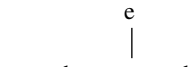
Counting (6)

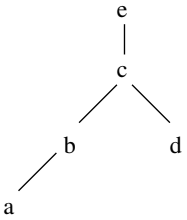
Observation: Assume a node v whose `count` array is $[c_1, \ldots, c_m]$ and whose `summands` is $s = [s^0, \ldots, s^n]$ with $s_i = [s_0^i, \ldots, s_m^i]$, then

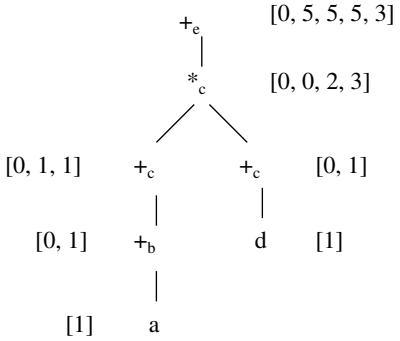
$$c_i = \sum_{j=0}^{m} s_j^i$$

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.

SDG example







Annotating the SDG

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) $= [c_0^1, \ldots, c_{m_1}^1]$;
    count(v) $= [0, c_1, \ldots, c_{m_1+1}]$ where $c_k = \sum_{i=k-1}^{m_1} c_i^1$
    summands(v) $= [s^0, \ldots, s^{m_1+1}]$ where $s^k = [s_0^k, \ldots, s_{m_1+1}^k]$ and
    $s_i^k = \begin{cases} c_i^1 & \text{if } 0 < k \text{ and } k-1 \leq i \\ 0 & \text{else} \end{cases}$

Annotating the SDG (2)

case 2:

Adorn(w_1)
Adorn(w_2)
assume count(w_1) $= [c_0^1, \ldots, c_{m_1}^1]$
assume count(w_2) $= [c_0^2, \ldots, c_{m_2}^2]$
count(v) $= [c_0, \ldots, c_{m_1+m_2}]$ where
$c_k = \sum_{i=0}^{m_1} \binom{k}{i} c_i^1 c_{k-i}^2$; // $c_i^2 = 0$ for $i \notin \{0, \ldots, m_2\}$
summands(v) $= [s^0, \ldots, s^{m_1+m_2}]$ where $s^k = [s_0^k, \ldots, s_{m_1}^k]$ and

$$s_i^k = \begin{cases} \binom{k}{i} c_i^1 c_{k-i}^2 & \text{if } 0 \leq k - i \leq m_2 \\ 0 & \text{else} \end{cases}$$

}

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.

Unranking Without Cross Products

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:
    // no additional adornment for v

Unranking Without Cross Products (2)

case 1:

let $\text{count}(v) = [c_0, \ldots, c_n]$

let $\text{summands}(v) = [s^0, \ldots, s^n]$

$k_1 = \min_j r \leq \sum_{i=0}^{j} s_i^k$

$r_1 = r - \sum_{i=0}^{k_1 - 1} s_i^k$

$\text{insert-at}(v) = k$

$\text{UnrankingTreeNoCrossLocal}(w_1, r_1, k_1)$

Unranking Without Cross Products (3)

case 2:

let $\text{count}(v) = [c_0, \ldots, c_n]$

let $\text{summands}(v) = [s^0, \ldots, s^n]$

let $\text{count}(w_1) = [c_0^1, \ldots, c_{n_1}^1]$

let $\text{count}(w_2) = [c_0^2, \ldots, c_{n_2}^2]$

$k_1 = \min_j r \leq \sum_{i=0}^{j} s_i^k$

$q = r - \sum_{i=0}^{k_1-1} s_i^k$

$k_2 = k - k_1$

$(r_1, r_2, a) = \text{UnrankTriplet}(q, c_{k_1}^1, c_{k_2}^2, \binom{k}{i})$

$\alpha = \text{UnrankDecomposition}(a)$

$\text{merge-using}(v) = \alpha$

$\text{UnrankingTreeNoCrossLocal}(w_1, r_1, k_1)$

$\text{UnrankingTreeNoCrossLocal}(w_2, r_2, k_2)$

}

Quick Pick

- problem: build (pseudo-)random join trees fast
- unranking without cross products is quite involved
- idea: randomly select an edge in the query graph
- extend join tree by selected edge

No longer uniformly distributed, but very fast

Quick Pick (2)

QuickPick(Query Graph G)
Input: a query graph $G = (\{R_1, \ldots, R_n\}, E)$
Output: a bushy join tree
$E' = E$;
Trees $= \{R_1, \ldots, R_n\}$;
while |Trees| > 1 {
  choose a random $e \in E'$
  $E' = E' \setminus \{e\}$
  if e connects two relations in different subtrees $T_1, T_2 \in$ Trees
    Trees $=$ Trees$\setminus\{T_1, T_2\} \cup$CreateJoinTree$(T_1, T_2)$
}
return $T \in$Trees

- repeated multiple times to find a good tree

Metaheuristics

- provide a very general optimization strategy
- applicable for many different problems
- work well even for very large problems
- but are often considered a "brute-force" method

We consider the metaheuristics formulated for the join ordering problem.

Iterative Improvement

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

Iterative Improvement (2)

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

Iterative Improvement (3)

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

- problem: local minimum detection

Simulated Annealing

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

Simulated Annealing (2)

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

Simulated Annealing (3)

```
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
```

Simulated Annealing (4)

Advantages:

- can escape from local minimum
- produces better results than II

Problems:

- parameter tuning
- initial temperature
- when and how to decrease the temperature

Tabu Search

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

Tabu Search (2)

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
  if cost(Tree) $<$ cost(BestTreeSoFar)
    BestTreeSoFar = Tree
  if (|TabuSet| $>$ limit) remove oldest tree from TabuSet
  TabuSet = TabuSet$\cup${Tree}
}
return BestTreeSoFar

Genetic Algorithms

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

Problem: Encoding

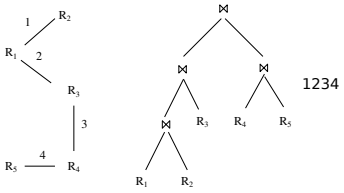
- Chromosome $\longleftrightarrow$ string
- Gene $\longleftrightarrow$ character

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 $L =< R_1, \ldots, R_n >$.

- 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 $< R_1, R_2, R_3, R_4 >$, the left-deep join tree $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as "1311".

Ordinal Number Encoding (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 $< R_1, R_2, R_3, R_4 >$, 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).

Combinations

- metaheuristics are often not used in isolation
- they can be used to improve existing heurstics
- or heuristics can be used to speed up metaheuristics

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 S 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 [9]
- 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

Iterative Dynamic Programming (2)

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 each $1 \leq i \leq n$ {
  BestTree($\{R_i\}$) $= R_i$;
}
ToDo $= \{R_1, \ldots, R_n\}$

Iterative Dynamic Programming (3)

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

Iterative Dynamic Programming (4)

- compromise between runtime and optimality
- combines greedy heuristics with dynamic programming
- scales well to large problems
- finds the optimal solution for smaller problems
- approach can be used for different DP strategies

Order Preserving Joins

- some query languages operatore on lists instead of sets/bags
- order of tuples matters
- examples: XPath/XQuery
- alternatives: either add sort operators or use order preserving operators

Here, we define order preserving operators, $list \rightarrow list$

- let L be a list
- $L[1]$ is the first entry in L
- $L[2 : |L|]$ are the remaining entries

Order Preserving Selection

We define the order preserving selection σ^L as follows:

$$\sigma_p^L(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < e[1] > \circ \sigma_p^L(e[2 : |e|]) & \text{if } p(e[1]) \\ \sigma_p^L(e[2 : |e|]) & \text{otherwise} \end{cases}$$

- filters like a normal selection
- preserves the relative ordering (guaranteed)

Order Preserving Cross Product

We define the order preserving cross product $\times^L$ as follows:

$$e_1 \times^L e_2 := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ (e[1] \hat{\times}^L e_2) \circ (e_1[2 : |e_1|] \times^L e_2) & \text{otherwise} \end{cases}$$

using the tuple/list product defined as:

$$t \hat{\times}^L e := \begin{cases} \epsilon & \text{if } e = \epsilon \\ < t \circ e[1] > \circ (t \hat{\times}^L e[2 : |e|]) & \text{otherwise} \end{cases}$$

- preserves the order of e_1
- order of e_2 is preserved for each e_1 group

Order Preserving Join

The definition of the order preserving join is analogous to the non-order preserving case:

$$e_1 \bowtie_p^L e_2 := \sigma_p^L(e_1 \times^L e_2)$$

- preserves order of e_1, order of e_2 relative to e_1

Equivalences

$$
\begin{array}{rcll}
\sigma^L_{p_1}(\sigma^L_{p_2}(e)) & \equiv & \sigma^L_{p_2}(\sigma^L_{p_1}(e)) & \\
\sigma^L_{p_1}(e_1 \bowtie^L_{p_2} e_2) & \equiv & \sigma^L_{p_1}(e_1) \bowtie^L_{p_2} e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) \\
\sigma^L_{p_2}(e_1 \bowtie^L_{p_2} e_2) & \equiv & e_1 \bowtie^L_{p_2} \sigma^L_{p_1}(e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_2) \\
e_1 \bowtie^L_{p_1} (e_2 \bowtie^L_{p_2} e_3) & \equiv & (e_1 \bowtie^L_{p_1} e_2) \bowtie^L_{p_2} e_3) & \text{if } \mathcal{F}(p_i) \subseteq \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1})
\end{array}
$$

- swap selections
- push selections down
- associativity

Commutativity

Consider the relations $R_1 =< [a:1],[a:2] >$ and $R_2 =< [b:1],[b:2] >$. Then

$$\begin{aligned} R_1 \bowtie^L_{true} R_2 &= < [a:1,b:1],[a:1,b:2],[a:2,b:1],[a:2,b:2] > \\ R_2 \bowtie^L_{true} R_1 &= < [a:1,b:1],[a:2,b:1],[a:1,b:2],[a:2,b:2] > \end{aligned}$$

- the order preserving join is not commutative

Algorithm

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

Algorithm (2)

OrderPreservingJoins($R = \{R_1, \ldots, R_n\}$,P)
Input: a set of relations to be joined and a set of predicates
Output: fills p, s, c, t
for each $1 \leq i \leq n$ {
  $p[i, i]$ =predicates from P applicable to R_i
  $P = P \setminus p[i, i]$
  $s[i, i]$ =statistics for $\sigma_{p[i,i]}(R_i)$
  $c[i, i]$ =costs for $\sigma_{p[i,i]}(R_i)$
}

Algorithm (3)

for each $2 \leq l \leq n$ **ascending** $\{$
  for each $1 \leq i \leq n - l + 1$ $\{$
    $j = i + l - 1$
    $p[i, j]$=predicates from P applicable to $R_i, \ldots, R_j$
    $P = P \setminus p[i, j]$
    $s[i, j]$=statistics derived from $s[i, j-1]$ and $s[j, j]$ including $p[i, j]$
    $c[i, j]$=∞
    for each $i \leq k < j$ $\{$
      $q = c[i, k] + c[k+1, j]$+costs for $s[i, k]$ and $s[k+1, j]$ and $p[i, j]$
      if $q < c[i, j]$ $\{$
        c[i,j]=q
        t[i,j]=k
      $\}$
    $\}$
  $\}$
$\}$

Algorithm (4)

ExtractPlan($R = \{R_1, \ldots, R_n\}, t, p$)
Input: a set of relations, arrays t and p
Output: a bushy join tree
return ExtractPlanRec($R, t, p, 1, n$)

ExtractPlanRec($R = \{R_1, \ldots, R_n\}, t, p, i, j$)
if $i < j$ {
  T_1 =ExtractPlanRec($R, t, p, i, t[i, j]$)
  T_2 =ExtractPlanRec($R, t, p, t[i, j] + 1, j$)
  return $T_1 \bowtie^L_{p[i,j]} T_2$
} **else** {
  return $\sigma_{p[i,j]} R_i$
}

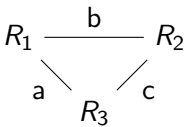
Complexity of Join Processing

- We have focused on how to *optimize* join queries
- But what is the complexity of actually computing a join query?
- Can we do better than a sequence of hash joins for > 2 relations?

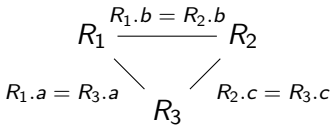
Complexity of Join Processing

Within this section

- We assume set semantics and only inner-joins with equality predicates
- For simplicity, we also assume relations contain no attributes other than join attributes.



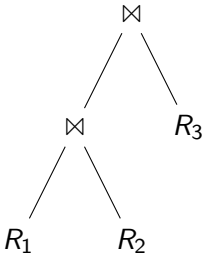
is shorthand for



Complexity of Join Processing

- What is the runtime complexity of a join query?
- The best we can do is $\Omega(|\text{Input}| + |\text{Output}|) = \Omega(\sum_i |R_i| + |R_1 \bowtie R_2 \bowtie \dots|)$
- For acyclic queries there is an algorithm that achieves $\mathcal{O}(k(|\text{Input}| + |\text{Output}|))$, with k as the size of the query graph
- For the general case, the best known algorithm is $\mathcal{O}(k(|\text{Input}| + |\text{Worst Case Output}|))$

Complexity of Join Processing



R_1

a	b
1	1
1	2
2	2
3	2

R_2

b	c
1	1
2	2
2	3

R_3

c
1

$R_1 \bowtie R_2$

a	b	c
1	1	1
1	2	2
1	2	3
2	2	2
	...	

$R_1 \bowtie R_2 \bowtie R_3$

a	b	c
1	1	1

Goal

- Eliminate dangling tuples, i.e. tuples that won't appear in the join result
- $R_i' := \Pi_{\mathcal{A}(R_i)}(R_1 \bowtie ... \bowtie R_k)$

  $\implies$ Intermediate join result sizes are $\mathcal{O}(|\text{Input}| + |\text{Output}|)$ for acyclic queries

  $\implies$ $\mathcal{O}(k(|\text{Input}| + |\text{Output}|))$ runtime
- How do we compute R_i' efficiently without evaluating the full join for acyclic queries?

Semi-Join Reduction & The Yannakakis Algorithm

- Semi join reduction: $R \bowtie S \equiv (R \ltimes S) \bowtie S$
- Goal: Compute $R'_i := \Pi_{\mathcal{A}(R_i)}(R_1 \bowtie ... \bowtie R_k)$ for acyclic QG
- Full Semi-Join Reduction [10]:
  - Root the query graph at any node
  - Apply semi-join reductions from leaf to root
  - Apply semi-join reductions from root to leaf
- The relations are now fully reduced
- Joining the fully reduced relations allows us to compute the acyclic query in polynomial time in the input and output (result due to Yannakakis [11])

Semi-Join Reduction & The Yannakakis Algorithm

$R_1 - R_2 - R_3$

R_1

a	b
1	1
1	2
2	2
3	2

R_2

b	c
1	1
2	2
2	3

R_3

c
1

Semi-Join Reduction & The Yannakakis Algorithm

$R_1 \text{ — } R_2 \text{ — } R_3$

R_1

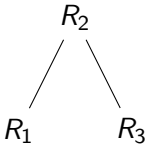
a	b
1	1
1	2
2	2
3	2

R_2

b	c
1	1
2	2
2	3

R_3

c
1



Semi-Join Reduction & The Yannakakis Algorithm

$R_1 - R_2 - R_3$

R_1

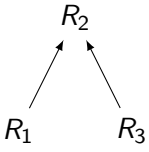
a	b
1	1
1	2
2	2
3	2

R_2

b	c
1	1
2	2
2	3

R_3

c
1



Bottom Up

- $R_2 := R_2 \ltimes R_1$
- $R_2 := R_2 \ltimes R_3$

Semi-Join Reduction & The Yannakakis Algorithm

$R_1 - R_2 - R_3$

R_1

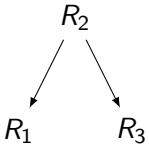
a	b
1	1
1	2
2	2
3	2

R_2

b	c
1	1
2	2
2	3

R_3

c
1



Bottom Up

- $R_2 := R_2 \ltimes R_1$
- $R_2 := R_2 \ltimes R_3$

Top Down

- $R_1 := R_1 \ltimes R_2$
- $R_3 := R_3 \ltimes R_2$

Semi-Join Reduction & The Yannakakis Algorithm

$R_1 - R_2 - R_3$

R_1

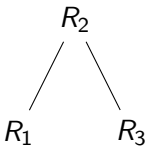
a	b
1	1
1	2
2	2
3	2

R_2

b	c
1	1
2	2
2	3

R_3

c
1



Bottom Up

- $R_2 := R_2 \ltimes R_1$
- $R_2 := R_2 \ltimes R_3$

Top Down

- $R_1 := R_1 \ltimes R_2$
- $R_3 := R_3 \ltimes R_2$

Join

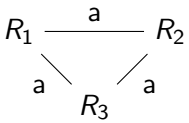
- $(R_1 \bowtie R_2) \bowtie R_3$

Semi-Join Reduction & The Yannakakis Algorithm

- The Yannakakis Algorithm computes the result of an acyclic join query in polynomial time in the input and output size.
- The resulting plan may be better than the best pure inner-join plan.
- However, the resulting plan may be suboptimal as the semi-joins have additional costs.
- The optimizer should decide when to apply semi-join reduction.

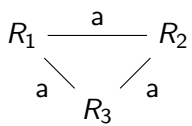
Generalization of Acyclic Queries

- A query is acyclic iff. there is an equivalent query with an acyclic query graph.
- Is the following query cyclic or acyclic?



Generalization of Acyclic Queries

- A query is acyclic iff. there is an equivalent query with an acyclic query graph.
- Is the following query cyclic or acyclic?



- We can find an equivalent query that has an acyclic query graph:

$R_1 \overset{a}{-} R_2 \overset{a}{-} R_3$

GYO (Graham-Yu-Özsoyoğlu) reduction

- Idea: Remove "ear" relations as they do not change whether the query is cyclic.
- A relation R_i is an ear if:
  - R_i has no outgoing edges, or
  - $\exists R_j$: JoinAttributes$(R_i) \subseteq$ JoinAttributes(R_j)
    assuming, w.l.o.g., all equal attributes have the same name
- If no relations remain in the end, the query is acyclic.

GYOReduction(R)
Input: a set of relations R
Output: a reduced set of relations R'
while There is an ear R_i
  $R := R \setminus \{R_i\}$
return R

GYO (Graham-Yu-Özsoyoğlu) reduction

- If no relations remain, the query is acyclic.
  - GYO reduction order $\implies$ Semi join order for full reduction
- If relations remain which cannot be removed, the query is cyclic
  - No known output optimal algorithms for cyclic queries.

Output Size of Join Queries

$$R_1 \overset{\mathsf{a}}{—} R_2$$

Output Size of Join Queries

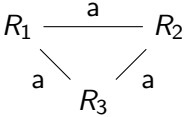
$$R_1 \overset{a}{—} R_2 \qquad \leq n_1 \cdot n_2$$
$$\leq n_1$$
$$\leq n_2$$

$$R_1 \overset{a}{—} R_2 \overset{b}{—} R_3$$

Output Size of Join Queries

$$R_1 \overset{a}{-} R_2 \qquad \leq n_1 \cdot n_2 \qquad \leq n_1 \qquad \leq n_2$$

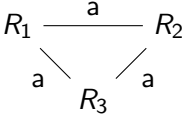
$$R_1 \overset{a}{-} R_2 \overset{b}{-} R_3 \qquad \leq n_1 \cdot n_2 \cdot n_3 \qquad \leq n_1 \cdot n_3 \qquad \leq n_2$$



Output Size of Join Queries

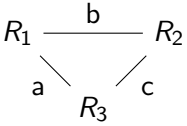
$$R_1 \overset{a}{-} R_2 \qquad \leq n_1 \cdot n_2 \qquad \leq n_1 \qquad \leq n_2$$

$$R_1 \overset{a}{-} R_2 \overset{b}{-} R_3 \qquad \leq n_1 \cdot n_2 \cdot n_3 \qquad \leq n_1 \cdot n_3 \qquad \leq n_2$$

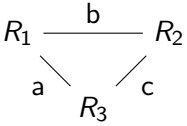


$$\leq n_1 \cdot n_2 \cdot n_3 \qquad \leq \min\{n_1, n_2, n_3\}$$

Output Size of Join Queries



Output Size of Join Queries



$\leq n_1 \cdot n_2 \cdot n_3$

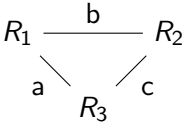
$\leq n_1 \cdot n_2$

$\leq n_2 \cdot n_3$

$\leq n_1 \cdot n_3$

Can we do even better?

Output Size of Join Queries



$\leq n_1 \cdot n_2 \cdot n_3$

$\leq n_1 \cdot n_2$

$\leq n_2 \cdot n_3$

$\leq n_1 \cdot n_3$

Can we do even better?

$\leq \sqrt{n_1 \cdot n_2 \cdot n_3} = n^{1.5}$

Output Size of Join Queries

Suboptimality of hash joins:

- $R_1(a, b) = R_2(b, c) = R_3(c, a) = ([1] \times [n]) \cup ([n] \times [1])$
- $|R_1| = 2n - 1 = \mathcal{O}(n)$
- $R_1 \bowtie R_2 = ([n] \times [1] \times [n]) \cup ([1] \times [n] \times [1])$
- $|R_1 \bowtie R_2| = n^2 + n - 1 = \mathcal{O}(n^2)$
- $R_1 \bowtie R_2 \bowtie R_3 = ([n] \times [1] \times [1]) \cup ([1] \times [n] \times [1]) \cup ([1] \times [1] \times [n])$
- $|R_1 \bowtie R_2 \bowtie R_3| = 3n - 2 = \mathcal{O}(n)$
- No hash join plan is output optimal!

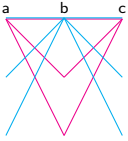
Output Size of Join Queries

Suboptimality of hash joins (visualized for $n = 3$):

R_1

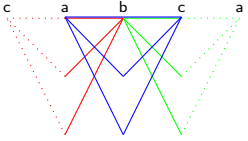
$3 + 3 - 1$

$R_1 \bowtie R_2$



$3^2 + 3 - 1$

$R_1 \bowtie R_2 \bowtie R_3$



$3 + 3 + 3 - 2$

Output Size of Join Queries

Constructing the worst case:

- $m := \sqrt{n}$
- $R_1(a, b) = R_2(b, c) = R_3(c, a) = [m] \times [m]$
- $|R_1| = m^2 = n$
- $|R_1 \bowtie R_2| = m^3$
- $|R_1 \bowtie R_2 \bowtie R_3| = m^3 = n^{1.5}$

Output Size of Join Queries

Constructing the worst case (example for $n = 4$):

- $m = \sqrt{n} = 2$
- $a = b = c = [m] = \{1, 2\}$
- $R_1(a, b) = R_2(b, c) = R_3(c, a) = [m] \times [m] = \{(1, 1), (1, 2), (2, 1), (2, 2)\}$
- $R_1(a, b) \bowtie R_2(b, c) = \{(1, 1, 1), (1, 1, 2), (1, 2, 1), \ldots (2, 2, 2)\}$
- $R_1(a, b) \bowtie R_2(b, c) \bowtie R_3(c, a) = \{(1, 1, 1), (1, 1, 2), (1, 2, 1), \ldots (2, 2, 2)\}$

Lower Bounds on Worst Case Join Size

Goal: Maximize join result size given query graph and base relation sizes n_i:

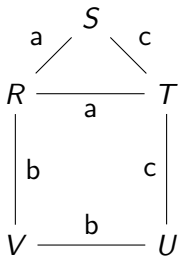
- Idea: Maximize join size by optimizing the domain sizes v_j of the attributes.
- Let $\mathcal{R}$ be a set of relations $\{R_1, R_2, \ldots\}$ and $\mathcal{A}$ a set of attributes $\{a_1, a_2, \ldots\}$.
- Each attribute $a_j \in \mathcal{A}$ is defined to be $a_j := [v_j]$ with variables v_j.
- Each relation is defined to be a cross product of its attributes
  $R_i = \times_{a_j \in \mathcal{A}(R_i)}(a_j) \qquad |R_i| = \prod_{a_j \in \mathcal{A}(R_i)}(v_j)$
- The result of the join is thus a cross product of all the attributes
  $Q = \times_{a_j \in \mathcal{A}}(a_j) \qquad |Q| = \prod_{a_j \in \mathcal{A}}(v_j)$

$$\begin{aligned} \underset{v}{\text{maximize}} \quad & \prod_{a_j \in \mathcal{A}} v_j \\ \text{subject to} \quad & n_i \geq \prod_{a_j \in \mathcal{A}(R_i)} v_j \quad \forall R_i \in \mathcal{R} \end{aligned}$$

Lower Bounds on Worst Case Join Size

Our linear program gives us *lower bounds* on the worst possible join result size.

Example:



- Given $|R| = |S| = |T| = |U| = |V| = 100$
- Candidate solution: $|a| = |b| = |c| = 10$ with $|Q| = 10^3 = 1000$
- We know that the worst possible join result size is *at least* 1000.
- Can there be an even worse case?

Upper Bounds on Worst Case Join Size (AGM Bound)

$$
\begin{aligned}
&\underset{v}{\text{maximize}} && \prod_{a_j \in \mathcal{A}} v_j \\
&\text{subject to} && n_i \geq \prod_{a_j \in \mathcal{A}(R_i)} v_j \quad \forall R_i \in \mathcal{R} \\
= &\underset{w}{\text{minimize}} && \prod_{R_i \in \mathcal{R}} n_i^{w_i} \\
&\text{subject to} && 1 \leq \sum_{i : a_j \in \mathcal{A}(R_i)} w_i \quad \forall a_j \in \mathcal{A}
\end{aligned}
$$

Upper Bounds on Worst Case Join Size (AGM Bound)

$$\begin{aligned} \underset{w}{\text{minimize}} \quad & \prod_{R_i \in \mathcal{R}} n_i^{w_i} \\ \text{subject to} \quad & 1 \leq \sum_{i : a_j \in \mathcal{A}(R_i)} w_i \quad \forall a_j \in \mathcal{A} \end{aligned}$$

- Assign values w_i in range $[0, 1]$ to every relation.
- Make sure that every attribute's connected relations sum up to 1.
- The minimum is equivalent to the maximum of the dual problem.
- Turns out, every correct assignment of values gives a proper upper bound to the worst case join result size (proven by Atserias, Grohe, and Marx [12]).

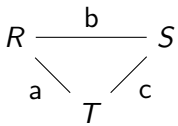
Bounds on Worst Case Join Size

Lower Bounds

- $|a| = |b| = |c| = 1$
- $|Q| \geq |a||b||c| = 1$

- $|a| = |b| = |c| = 10$
- $|Q| \geq |a||b||c| = 1000$

Rel. size 100



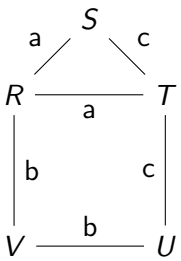
Upper Bounds

- $R : 1, S : 1, T : 0$
- $|Q| \leq |R|^1|S|^1|T|^0 = 10000$

- $R : 0.5, S : 0.5, T : 0.5$
- $|Q| \leq |R|^{0.5}|S|^{0.5}|T|^{0.5} = 1000$

Bounds on Worst Case Join Size

Rel. size 100



Lower Bounds

- $|a| = |b| = |c| = 1$
- $|Q| \geq |a||b||c| = 1$

- $|a| = |b| = |c| = 10$
- $|Q| \geq |a||b||c| = 1000$

Upper Bounds

- $R : 1, T : 1, U : 1, S : 0, V : 0$
- $|Q| \leq |R||T||U| = 1000000$

- $R : 0.5, T : 0.5, U : 0.5, S : 0, V : 0$
- $|Q| \leq |R|^{0.5}|T|^{0.5}|U|^{0.5} = 1000$

Worst Case Optimal Join Algorithms

- All join queries can be computed in time $\mathcal{O}(k(\text{Worst Case Join Result Size}))$
- Not output optimal, but potentially faster than pure hash joins
- Only supports inner-joins with simple equality predicates
- Idea: Compute the result attribute by attribute rather than relation by relation

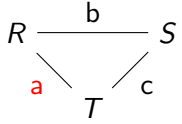
Worst Case Optimal Join Algorithms

GenericJoin(Q)
Input: a query graph Q with some attributes fixed
Output: the join result
if all attributes of Q are fixed
  return the fixed attributes as a result tuple
$J := \emptyset$
Pick arbitrary attribute a
Assume a occurs in relations $R_{i_1}, \ldots, R_{i_k}$
Compute $A := \Pi_a(R_{i_1}) \cap \ldots \cap \Pi_a(R_{i_k})$ in time $\mathcal{O}(\min(|R_{i_1}|, \ldots, |R_{i_k}|))$
for $v \in A$
  $Q' :=$ Q with attribute a fixed to constant v
  $J := J \cup$ GenericJoin(Q')
return J

Worst Case Optimal Join Algorithms

Example execution for the triangle join:

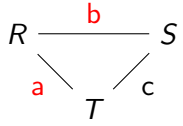
GenericJoin($R(a,b) \bowtie S(b,c) \bowtie T(c,a)$)
Input: a query graph
Output: the join result
$J := \emptyset$
Pick attribute a
Compute $A := \Pi_a(R) \cap \Pi_a(T)$
for $v_a \in A$
  Fix attribute a to v_a
  $R' := \sigma_{a=v_a}(R)$
  $T' := \sigma_{a=v_a}(T)$
  $J := J \cup \text{GenericJoin}(R'(b) \bowtie S(b,c) \bowtie T'(c))$
return J



Worst Case Optimal Join Algorithms

Example execution for the triangle join (2):

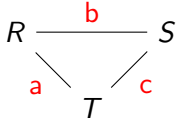
GenericJoin($R'(b) \bowtie S(b,c) \bowtie T'(c)$)
Input: a query graph
Output: the join result
$J := \emptyset$
Pick attribute b
Compute $B := \Pi_b(R') \cap \Pi_b(S)$
for $v_b \in B$
  Fix attribute b to v_b
  $S' := \sigma_{b=v_b}(S)$
  $J := J \cup$ GenericJoin($S'(c) \bowtie T'(c)$)
return J



Worst Case Optimal Join Algorithms

Example execution for the triangle join (3):

GenericJoin($S'(c) \bowtie T'(c)$)
Input: a query graph
Output: the join result
$J := \emptyset$
Pick attribute c
Compute $C := \Pi_c(S') \cap \Pi_c(T')$
for $v_c \in C$
  Fix attribute c to v_c
  $J := J \cup \{(v_a, v_b, v_c)\}$
return J



Worst Case Optimal Join Algorithms

Generic Join:

- Order in which attributes are processed greatly influences execution time.
- Runtime is $\mathcal{O}(k(\text{Worst Case Join Result Size}))$, regardless of attribute order.
- Requires lots of precomputation to ensure *intersection* and *fixing* operations are fast.
- Multiple practical implementations exist [13, 14, 15].

Worst Case Optimal Join Algorithms

- WCOJs are, in general, significantly slower than binary hash joins.
- The optimizer must decide when to apply WCOJs. They are most useful if intermediate results are larger than the worst case result.
- WCOJs and the Yannakakis Algorithm can be combined to improve runtime for complex query graphs [16].

4. Accessing the Data

In this chapter we go into some details:

- deep into the (runtime) system
- close to the hardware

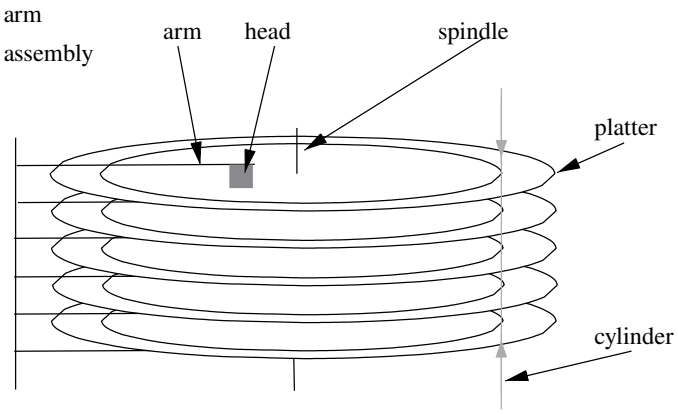
Goal:

- estimation and optimization of disk access costs

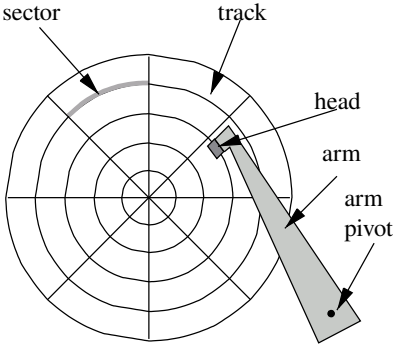
4. Accessing the Data (2)

- disk drives
- database buffer
- physical database organization
- physical algebra
- temporal relations and table functions
- indices
- counting the number of accesses
- disk drive costs
- selectivity estimations

Assembly



a. side view



b. top view

Zones

- outer tracks/sectors longer than inner ones
- highest density is fixed
- results in waste in outer sectors
- thus: cylinders organized into zones

Zones (2)

- 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

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*

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*

Addressing Sectors

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

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
...	...	...	...

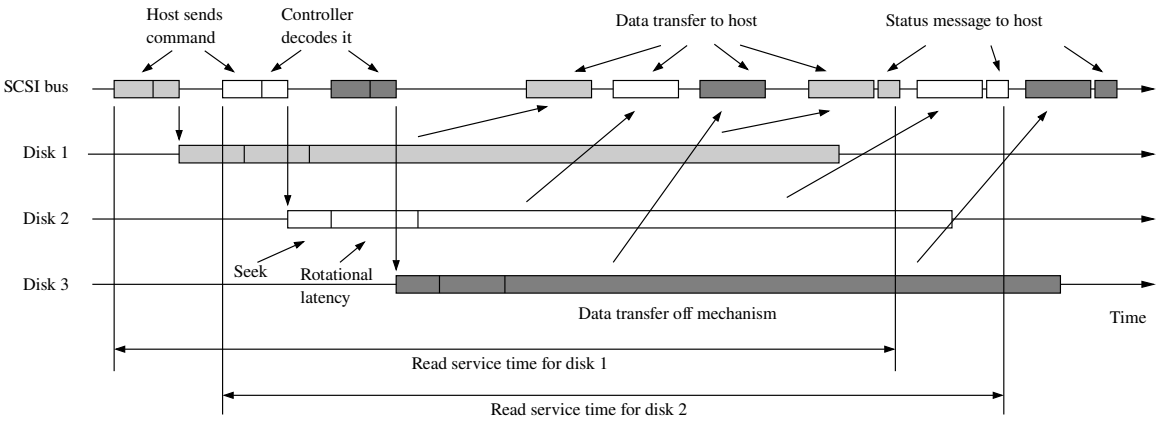
LBN to Physical Address (2)

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

Reading/Writing a Block



Reading/Writing a Block (2)

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. The settle times differ for read and write requests. For reads, an aggressive strategy is used. If, after all, it turns out that the block could not be read correctly, we can just discard it. For writing, a more conservative strategy is in order.
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.

Optimizing Round Trip Time

- caching
- read-ahead
- command queuing

Seek Time

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

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

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.

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

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

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	D_{cap}
average latency time	5 ms	D_{lat}
sustained read rate	100 MB/s	D_{srr}
sustained write rate	100 MB/s	D_{swr}

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

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.

Simplistic Cost Model

Read 100 MB

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

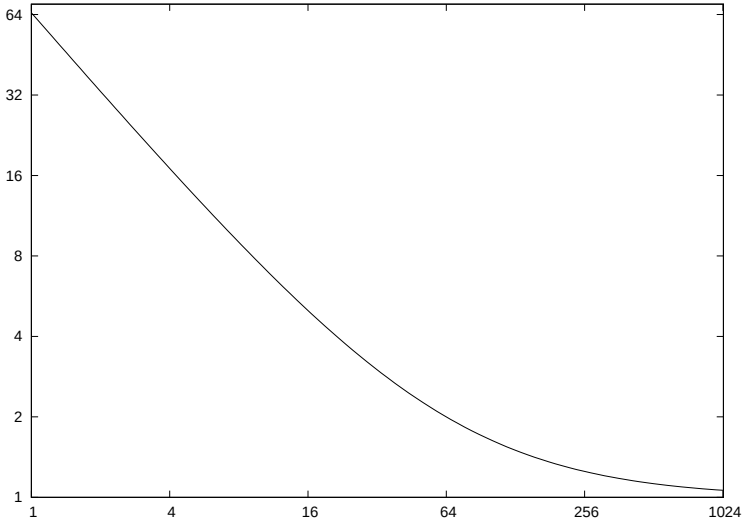
Simplistic Cost Model (2)

Problems:

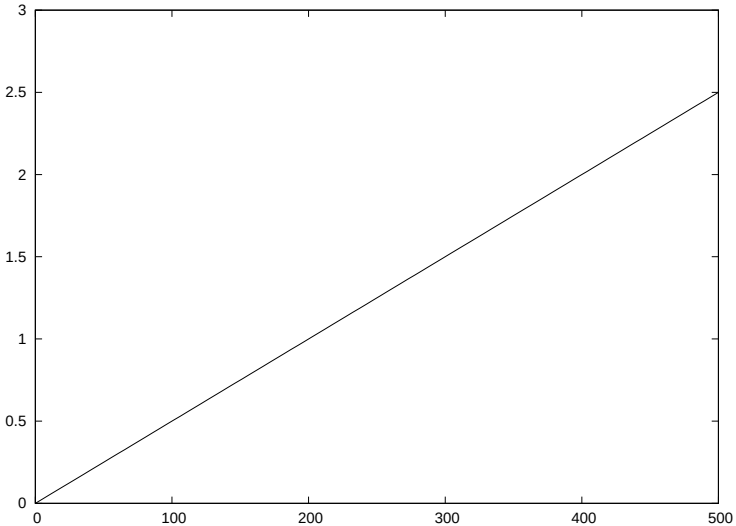
- other applications
- other transactions
- other read operations in the same QEP

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

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



Time to Read n Random Pages



Simplistic Cost Model (3)

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.

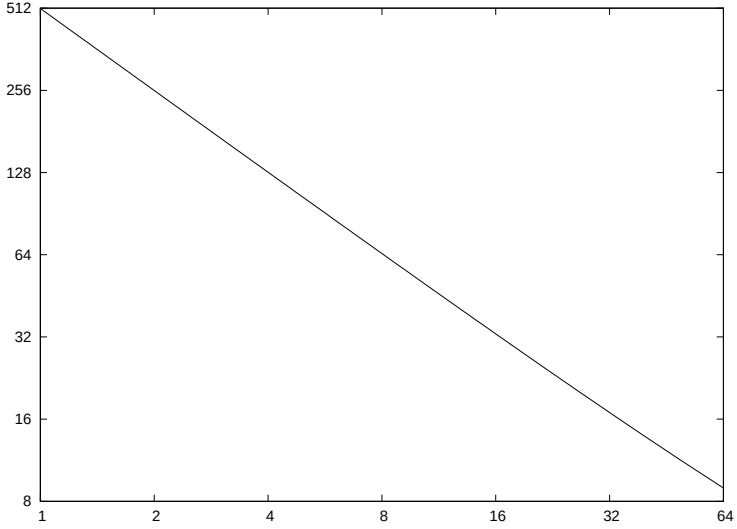
Simplistic Cost Model (4)

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.

Break Even Point (depending on page size)



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.

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

Disk Drive: Parameters

D_{cyl}	total number of cylinders
D_{track}	total number of tracks
D_{sec}	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

Disk Drive: Parameters (2)

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 $(= D_{\text{rot}}/Dzspti)$

Disk Drive: Parameters (3)

D_{seekavg}	average seek costs
D_{clim}	parameter for seek cost function
D_{ca}	parameter for seek cost function
D_{cb}	parameter for seek cost function
D_{cc}	parameter for seek cost function
D_{cd}	parameter for seek cost function

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

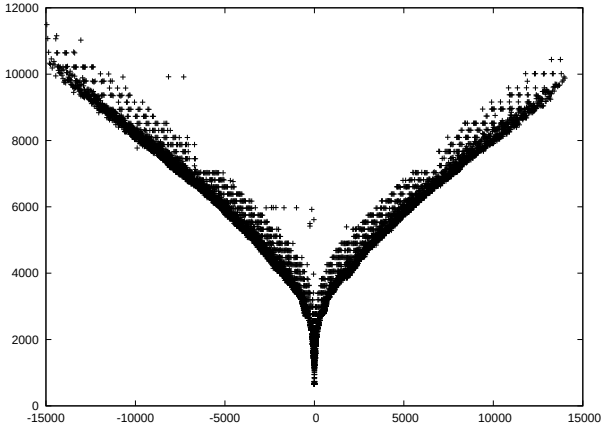
$$D_{\text{fseek}}(d) = \begin{cases} D_{\text{ca}} + D_{\text{cb}}\sqrt{d} & \text{if } d \leq D_{\text{clim}} \\ D_{\text{cc}} + D_{\text{cd}}\, d & \text{if } d > D_{\text{clim}} \end{cases}$$

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

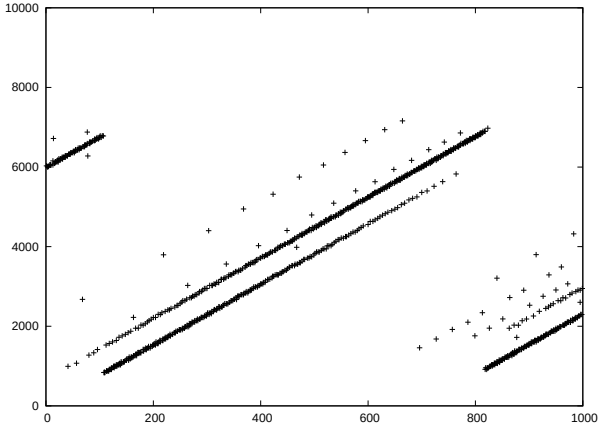
Extraction of Disk Drive Parameters

- documentation: often not sufficient
- mapping: interrogation via SCSI-Mapping command (disk drives lie)
- use benchmarking tools, e.g.:
  - Diskbench
  - Skippy (Microbenchmark)
  - Zoned

Seek Curve Measured with Diskbench



Skippy Benchmark Example



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

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 $sD_{fseek}(C/s)$ is an upper bound for the seek time.

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.

Buffer Accesses

Consider page acceses in a buffer with 2 pages:

page no	action
0	read page 0, place it in buffer
1	read page 1, place it in buffer
0	fix page 0 in buffer
2	swap out a page (e.g. 1), read 2, place it in buffer
0	fix page 0 in buffer
3	swap out a page, read 3, place it in buffer
...	

- replacement strategy is imporant
- unfixes omitted

Replacement Strategies

Some popular replacement strategies:

- random
- fifo
- lru
- Q2

lru is very popular

Replacement Strategies - random

- when a new page slot is needed, remove a random other page from the buffer
- easy to implements, needs no additional memory
- but does not take the access patterns into account
- primarily used as base line
- suitable for analytic results

Replacement Strategies - fifo

- first in - first out
- remove the page that was place in the buffer first
- easy to implement, needs no/few additional memory
- but does not adapt very well do access patterns
- increasing buffer size may hurt it

Fifo Anomaly:

- access pattern: 3 2 1 0 3 2 4 3 2 1 0 4
- buffer sizes: 3 vs. 4

Replacement Strategies - lru

- least recently used
- remove the page that has not been accessed for longest time
- requires a priority queue/linked list
- adapt to access patterns, popular pages stay in memory
- but slow to remove pages

very popular replacement strategy

Replacement Strategies - 2Q

- two queues
- a fifo queue and a lru queue
- place pages first in fifo, if they are accessed again place them in lru
- gets rid of pages that are accessed only once fast
- superior to lru, example of a "real" replacement strategy

Replacement Strategies - Effect on the Cost Model

- replacement affects the costs
- cost model needs predictions, though
- very hard to do in general

Typical approaches:

- ignore buffer effects
- assume random replacement
- make use of known access characteristics

Physical Database Organization

The database organizes the physical storage in multiple layers:

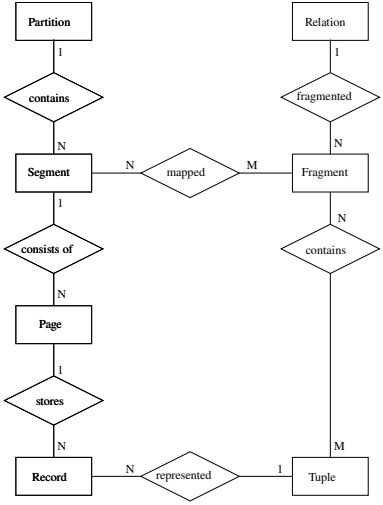
1. partition: sequence of pages (consecutive on disk)
2. extent: subsequence of a partition
3. segment (file): logical sequence of pages (implemented e.g. as set of extents)
4. record: sequence of bytes stored on a page

Note:

- partition/extent/page/record are physical structures
- a segment is a logical structure

Physical Storage of Relations

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



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*

Scan 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

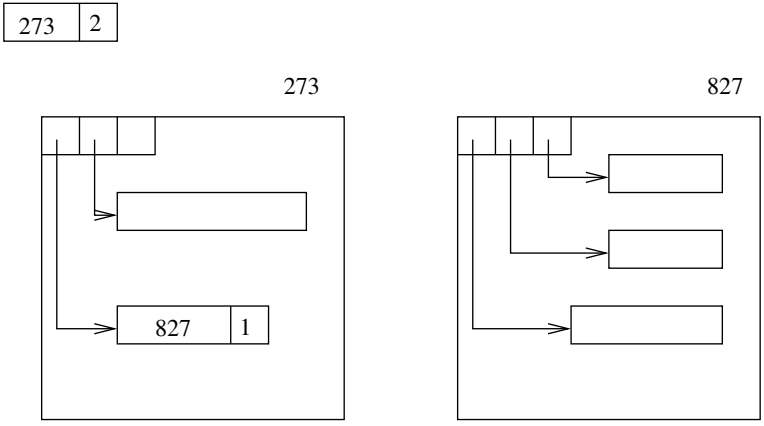
Model of a Segment

- 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.

Model of a Segment

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

Slotted Page



- page is organized into areas (slots)
- slots point to data chunks
- slots may point to other pages

Tuple Identifier (TID)

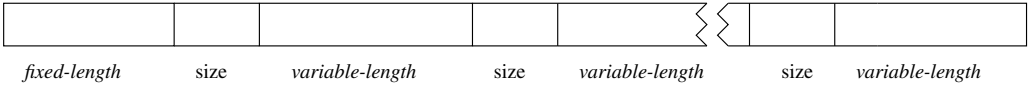
TID is conjunction of

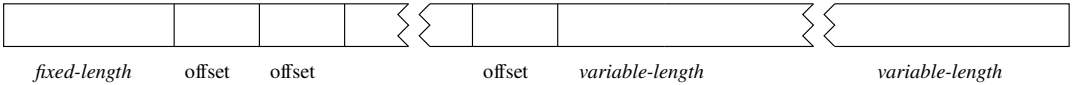
- page identifier (e.g. partition/segment no, page no)
- slot number

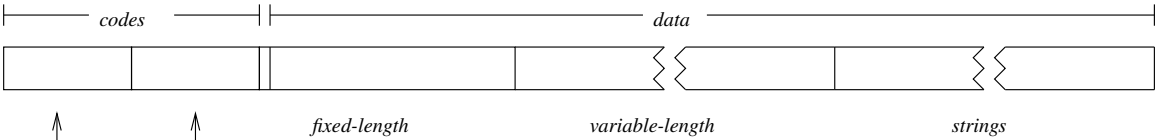
TID sometimes called Row Identifier (RID)

Record Layout

Different layouts possible:







Record Layout (2)

Record layout is a compromise:

- space consumption vs. CPU
- data model specific properties: e.g. generalization
- versioning / easy schema migration
- record layout typically not trivial
- accessing an attribute value has non-zero cost

Physical Algebra

- building blocks for query execution
- implements the algorithms for query execution
- very generic, reusable components
- describes the general execution approach
- annotated with predicates etc. for query specific parts

Iterator Concept

The general interface of each operator is:

- open
- next
- close

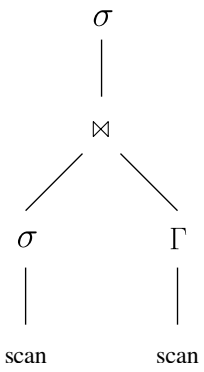
All physical algebraic operators are implemented as iterators.

- produce a stream of data items (tuples)

Implementations vary slightly for performance tuning (concept the same):

- `first`/`next` instead of `next`
- blocks of tuples instead of single tuples

Iterator Example



Note: all details (subscripts, implementations etc.) are omitted here

Pipelining

Pipelining is fundamental for the physical algebra:

- physical operators are iterators over the data
- they produce a stream of single tuples
- tuple stream if passed through other operators
- pipelining operators just pass the data through, they only filter or augment
- data is not copied or materialized
- very efficient processing

pipeline breakers disrupt this pipeline and materialize data:

- very expensive, can cause superfluous work
- sometimes cannot be avoided, though

Simple Scan

- a `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.

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

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*

Scan and Attribute Access

Strictly speaking, the plan

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

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

- implicit attribute access
- make attribute accesses explicit

Scan and Attribute Access (2)

Explicit attribute access:

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

Advantage: makes attribute access costs explicit

Scan and Attribute Access (3)

Consider:

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

Problem: accesses age twice

Scan and Attribute Access (4)

Map operator:

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

Loading Attributes

The above problem can now be solved by

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

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.

Loading Attributes (2)

select name
from Student
where age $>$ 30

The plan

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

is better than

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

Loading Attributes (3)

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

Loading Attributes (4)

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

Loading Attributes (5)

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

Remark

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

Example:

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

Loading Attributes and Pipeline Breakers

- attribute access not only for scans
- likewise all operators that materialize to disk
- most pipeline breakers
- projection and selection should always be integrated into pipeline breakers
- not that important for pipelining operators
- attribute access must happen before breaking the pipeline

Exception:

- RID join/semijoin techniques

Physical Operator - Selection

- consumes a tuple stream
- checks predicate on each tuple
- produces matching tuples

Characteristics:

- pipelining operator
- consumes no memory, causes no IO

Physical Operator - Nested Loop Join

- consumes two tuple streams
- for each tuple from one stream (trad: the left) consumes the whole other stream
- checks predicate on each pair
- produces matching tuples

Characteristics:

- pipelining operator
- consumes no memory, causes no IO (at least not directly)

Physical Operator - Blockwise Nested Loop Join

- consumes two tuple streams
- reads one stream (left) blockwise into memory, consumes the whole other stream for each block
- checks predicate on each pair of tuples
- produces matching tuples

Characteristics:

- pipeline breaker on the left stream
- consumes memory for the blocks, causes no IO (unusual for a pipeline breaker)

Variants (with hashing etc.) behave basically the same

Physical Operator - Sort Merge Join

We only consider the case that the input is already sorted (see *Sort*) and $1 : n$ or $1 : 1$.

- consumes two tuple streams
- skips uniformly through both streams
- checks predicate on each pair (implicitly)
- produces matching tuples

Characteristics:

- pipelining operator
- consumes no memory, causes no IO

Physical Operator - Grace Hash Join

- consumes two tuple streams
- reads one stream and splits it into partitions on disk
- the same of the other stream
- joins the partitions, produces matching tuples

Characteristics:

- full pipeline breaker
- consumes memory for one partition, writes/reads whole data at least once

IO behavior can be predicted relatively easily

Physical Operator - Hybrid Hash Join

- consumes two tuple streams
- reads one stream and splits it into partitions on disk. Tries to keep some partitions in memory
- reads the other stream, also splits it into partitions on disk, but already joins with partitions still in memory
- joins partitions on disk, produces matching tuples

Characteristics:

- (typically) full pipeline breaker. Might keep the pipeline for the second stream
- consumes memory for partitioning (size variable), might write/reads whole data

Behavior difficult to predict, might cause no IO, might write everything

Physical Operator - Sort

- consumes one input stream
- creates sorted runs, spools runs to disk, merges the runs
- produces sorted output stream

Characteristics:

- pipeline breaker
- consumes memory for one run, reads/write data $\log n$ times

Exact behavior depends on implementation, e.g. HeapSort might produce one run, while QuickSort produces fixed number of runs

Physical Operator - Sort Based Group By

We assume that the input is already sorted

- consumes one input stream
- aggregates the input directly
- produces an output tuple whenever the group by attribute changes

Characteristics:

- pipeline breaker (nearly pipelining, though)
- consumes memory for one tuple, causes no IO

Sometimes interleaved with sort (early aggregation)

Physical Operator - Hash Bases Group By

- consumes one input stream
- reads the stream, splits into partitions, writes partitions to disk (if needed)
- aggregates partitions, produces output tuples

Characteristics:

- pipeline breaker
- consumes memory for buffering (variable), might read/write the whole data
- two possibilities, similar to Grace Hash vs. Hybrid Hash

Variants with early aggregation etc.

Physical Operators - Others

Only mainstream operators included, some are missing:

- projection usually implicit
- duplicate elimination is a special kind of aggregation
- dependent join (nested loop, can be done somewhat differently)
- outer join/semi join/anti join etc. roughly similar to normal joins
- specialized operators for query languages: staircase join, twig join etc.
- their characteristics have to be known to the query optimizer

Temporal Relations

The query optimizer might introduce temporal relations:

- a "relations" just for the query
- allows for reusing intermediate results
- related: temporary views
- more efficient nested loop join
- materializes a subquery

Creating a temporary relation is an expensive operation therefore

- should be decided by the query optimizer
- but often done as rewrite
- typically breaks optimization in parts

Temporal Relations (2)

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

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

Better:

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

Or:

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

Table Functions

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

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

Translation:

$$Primes(1, 100)[p]$$

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

Table Functions (2)

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

Table Functions (3)

Definition d-join:

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

Translation of the above query:

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

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

Table Functions (4)

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 ≤ EXTRACT(YEAR FROM f.birthday) + 100
```

At the algebraic level: this optimization requires some knowledge

Indices

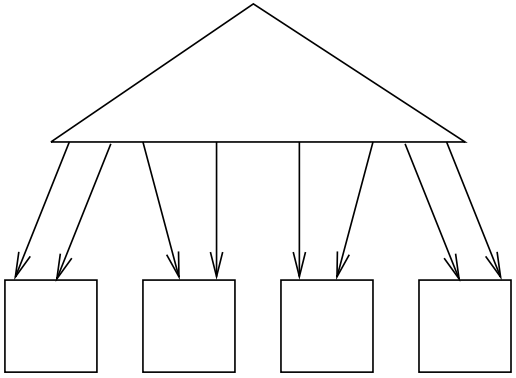
We consider B-Trees only

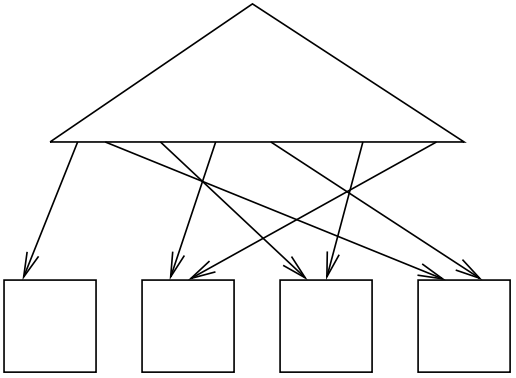
- key attributes: $a_1, \ldots, a_n$
- data attributes: $d_1, \ldots, d_m$
- Often: one special data attribute holding the TID of a tuple

Some notions:

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

Clustered vs. Non-Clustered B-Tree





- clustering is not always possible (or even desireable)

Single Index Access Path - Point Query

Exact match query:

select name
from Emp
where eno = 1077

Translation:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{eno}[x; eno = 1077]))$$

Alternative translation using d-join:

$$\Pi_{name}(Emp_{eno}[x; eno = 1077] \bowtie \chi_{e:*.tid,name:e.name}(\Box))$$

(x: holds ptr to index entry; *: dereference TID, $\Box$ is a singleton scan)

Single Index Access Path - Range Query

Range query:

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

Translation:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35]))$$

(Start and Stop condition)

Single Index Access Path - Sequential I/O

Turning random I/O into sequential I/O:

$$\Pi_{name}(\chi_{e:*tid,name:e.name}(sort_{x.tid}(Emp_{age}[x; 25 \leq age; age \leq 35; tid])))$$

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

Single Index Access Path - Sorted Output

Query demanding ordered output:

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

Translation:

$$\Pi_{name,age}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35]))$$

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

Single Index Access Path - Sorted Output (2)

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

$$\Pi_{name,age}(sort_{age}(\chi_{e:*tid,name:e.name}(sort_{tid}(Emp_{age}[x; 25 \leq age; age \leq 35; tid]))))$$

Possible speedup of sort by dense numbering:

$$\begin{array}{l}\Pi_{name,age}(\\ sort_{rank}(\\ \chi_{e:*tid,name:e.name}(\\ sort_{tid}(\\ \chi_{rank:counter++}(\\ Emp_{age}[x; 25 \leq age; age \leq 35; tid])))))\end{array}$$

Single Index Access Path - Other Predicates

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:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35; age \neq 30]))$$

Single Index Access Path - Other Predicates (2)

Non-inclusive bounds:

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

If supported by index:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 < age; age < 35]))$$

If unsupported:

$$\Pi_{name}(\chi_{e:*x.tid,name:e.name}(Emp_{age}[x; 25 \leq age; age \leq 35; age \neq 25, age \neq 35]))$$

Especially for predicates on strings this might be expensive.

Single Index Access Path - Ranges

Start and stop conditions are optional:

select name
from Emp
where age $\geq$ 60

or

select name
from Emp
where age $\leq$ 20

Single Index Access Path - No Range

Full index scan also useful:

select count(*)
from Emp

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

Single Index Access Path - No Range (2)

Min/max even more efficient:

select min/max(salary)
from Emp

Single Index Access Path - No Range (3)

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

Alternatives: one or two descents into the index.

Single Index Access Path - No Range (4)

Full index scan:

```
select   salary
from     Emp
order by salary
```

Translation:

$$Emp_{salary}$$

Single Index Access Path - String Ranges

Predicate on string attribute:

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

Start condition: $'Maaa' \leq name$

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

Start condition: $'M' \leq name$

Single Index Access Path

- an *access path* is a plan fragment with building blocks concerning a single database items.
- hence, every building block is an access path.
- 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 indices 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*.

Single Index Access Path - Complex Predicates

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:

- $age = 28 \vee age = 29 \vee age = 31 \vee age = 32$
- $age \neq 30$

Single Index Access Path - Complex Predicates (2)

A case for the d-join:

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

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

$$Sal[S] \bowtie \chi_{e:*tid,name:e.name}(Emp_{salary}[x; salary = S.s; tid])$$

- gap skipping/zig-zag skipping

Single Index Access Path - Compound Keys

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

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

Single Index Access Path - Compound Keys

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.

Single Index Access Path - Compound Keys (2)

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}.

Single Index Access Path - Improvements

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

Single Index Access Path - Improvements (2)

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.

Multi Index Access Path - Example

Query:

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

Indexes on all attributes

Multi Index Access Path - Example (2)

Translation:

$$
\begin{array}{l}
((((\\
\quad Camera_{megapixel}[c; megapixel > 5; tid]\\
\quad \cap\\
\quad\quad Camera_{distortion}[c; distortion < 0.05; tid])\\
\quad\quad \cap\\
\quad\quad\quad Camera_{noise}[c; noise < 0.01; tid])\\
\quad\quad\quad \cap\\
\quad\quad\quad\quad Camera_{zoomMin}[c; zoomMin < 35; tid])\\
\quad\quad\quad\quad \cap\\
\quad\quad\quad\quad\quad Camera_{zoomMax}[c; zoomMax > 105; tid])
\end{array}
$$

Then dereference

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

Multi Index Access Path - Combining

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.

Multi Index Access Path - Combining (2)

Index-oring (or merge):

select *
from Emp
where yearsOfEmployment ≥ 30
or age ≥ 65

Translation:

$$Emp_{yearsOfEmployment}[c; yearsOfEmployment \geq 30; tid] \cup Emp_{age}[c; age \geq 65; tid]$$

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

Multi Index Access Path - Combining (3)

Index differencing:

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

Translation:

$$Emp_{age}[c; age \geq 65; tid] \setminus Emp_{yearsOfEmployment}[c; yearsOfEmployment = 10; tid]$$

Multi Index Access Path - Combining (3)

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

select *
from Emp
where yearsOfEmployment ≤ 5
and age ≤ 60

Then

$$Emp_{yearsOfEmployment}[c; yearsOfEmployment \leq 5; tid] \setminus Emp_{age}[c; age > 60; tid]$$

could be more efficient than

$$Emp_{yearsOfEmployment}[c; yearsOfEmployment \leq 5; tid] \cap Emp_{age}[c; age \leq 60; tid]$$

Indices and Join

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

(intersection is similar to join (for sets))

Indices and Join (2)

Turn map

$$\chi_{e:*tid,name:e.name}(Emp_{salary}[x; 25 \leq age \leq 35; tid])$$

into d-join

$$Emp_{salary}[x; 25 \leq age \leq 35; tid] \bowtie \chi_{e:*tid,name:e.name}(\Box)$$

or even join

$$Emp_{salary}[x; 25 \leq age \leq 35] \bowtie_{x.tid=e.tid} Emp[e]$$

Variants: sorting at different places (by plan generator)

- pro: flexibility
- contra: large search space

Indices and Join (3)

Query:

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

Translation:

$$\Pi_{name,age}(\\ Emp_{age}[a; 40 \leq age \leq 50; TIDa, age] \\ \bowtie_{TIDa=TIDn} \\ Emp_{name}[n; name \geq' R'; name <' S'; TIDn, name])$$

Indices and Join (4)

The query

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

can be directly translated to

$$\sigma_{e.name=''Maier''}(Emp[e]) \bowtie_{e.dno=d.dno} Dept[d]$$

Indices and Join (5)

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

$$\chi_{e:*x.tid}(Emp_{name}[x; name =''\ Maier''])$$

Indices and Join (6)

With a d-join:

$$Emp_{name}[x; name ='' Maier''] \bowtie \chi_{e:*x.tid}(\Box)$$

Abbreviate $Emp_{name}[x; name ='' Maier'']$ by E_i
Abbreviate $\chi_{e:*x.tid}(\Box)$ by E_a.

Indices and Join (7)

Use index on `Dept.dno`:

$$E_i \bowtie E_a \bowtie Dept_{dno}[y; y.dno = dno]$$

Dereference TIDs (*index nested loop join*):

$$E_i \bowtie E_a \bowtie Dept_{dno}[y; y.dno = dno; dtid : y.tid] \bowtie \chi_{u:*dtid}(\Box)$$

Abbreviate $Dept_{dno}[y; y.dno = dno; dtid : y.tid]$ by D_i
Abbreviate $\chi_{u:*dtid}(\Box)$ by D_a
Fully abbreviated, the expression then becomes

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

Indices and Join - Performance Improvements

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:

Indices and Join - Performance Improvements (2)

- 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 $E_i \bowtie E_a$ 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 $E_i \bowtie E_a \bowtie D_i$ on `dtid` for the same reasons as mentioned above for sorting the result of E_i on `TID`.

Indices 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

$$E_i \bowtie E_a \bowtie Dept_{dno}[y; y.dno = dno; dtid : y.rid]$$

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

Indices and Join - Temping the Inner (2)

Use χ^{mat}:

$$E_i \bowtie E_a \bowtie \chi^{mat}_{tid:(Dept_{dno}[y;y.dno=dno]).tid}(\Box)$$

Indices and Join - Temping the Inner (3)

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 $\chi^{mat,1}$ to do exactly this.
A more efficient plan could thus be

$$sort_{dno}(E_i \bowtie E_a) \bowtie \chi^{mat,1}_{dtid:(Dept_{dno}[y;y.dno=dno]).tid}(\Box)$$

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

Indices and Join - Temping the Inner (4)

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 χ^{mat} is not sufficient.

Indices and Join - Temping the Inner (5)

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:

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

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

Indices and Join - Temping the Inner (6)

The memox operator performs caching:

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

Sorting still beneficial:

$$sort_{w.year}(Wine[w]) \bowtie memox^{1}(\sigma_{e.yearOfBirth=w.year}(Emp[e]))$$

Indices and Join - Temping the Inner (7)

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

$$sort_{w.year}(Wine[w])$$
$$\bowtie memox^{1}(Emp_{yearOfBirth}[x; x.yearOfBirth = w.year] \bowtie \chi_{e:*(x.tid)}(\Box))$$

Indices and Join - Temping the Inner (8)

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):

$$Emp_{yearOfBirth}[x] \bowtie^{merge}_{x.yearOfBirth=y.year} Wine_{year}[y]$$

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

From Cardinalities to 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

Parameters

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	$\|R\|$	number of tuples in the relation R
m	$\|\|R\|\|$	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*.

Special Cases

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.

General Case

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.

Different Settings

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

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.]

Direct, Uniform, Distinct (2)

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

$$\overline{\mathcal{Y}}_n^{N,m}(k) = m * \mathcal{Y}_n^N(k) \tag{17}$$

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

Direct, Uniform, Distinct (3)

Theorem (Waters/Yao (cont.))

The probability is

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

where p is the probability that a bucket contains none of the k items. The following alternative expressions can be used to calculate p:

$$p = \frac{\binom{N-n}{k}}{\binom{N}{k}} \tag{18}$$

$$= \prod_{i=0}^{k-1} \frac{N-n-i}{N-i} \tag{19}$$

$$= \prod_{i=0}^{n-1} \frac{N-k-i}{N-i} \tag{20}$$

Direct, Uniform, Distinct (4)

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

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

Direct, Uniform, Distinct (5)

Proof (2):

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

Direct, Uniform, Distinct (6)

Proof(3):

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

Direct, Uniform, Distinct (7)

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.

Direct, Uniform, Distinct (8)

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

Direct, Uniform, Distinct (9)

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

$$m\mathcal{Y}_1^N(k) = m\frac{k}{N} \tag{21}$$

qualify.

Direct, Uniform, Distinct (10)

Proof:

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

Direct, Uniform, Distinct (11)

Approximation of Yao's formula (1):

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

[Waters]

Direct, Uniform, Distinct (12)

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

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

[Whang, Wiederhold, Sagalowicz]

Direct, Uniform, Distinct (13)

Approximation of Yao's formula (3):

$$\overline{\mathcal{Y}}_n^{N,m}(k) \approx \begin{cases} k & \text{if } k < \frac{m}{2} \\ \frac{k+m}{3} & \text{if } \frac{m}{2} \leq k < 2m \\ m & \text{if } 2m \leq k \end{cases}$$

[Bernstein, Goodman, Wong, Reeve, Rothnie]

Direct, Uniform, Distinct (14)

Upper and lower bounds for p:

$$p_{\text{lower}} = (1 - \frac{k}{N - \frac{n-1}{2}})^n$$

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

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.

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

$$g(\{x_1 < \ldots < x_k\}) = \{x_1 - 0 \leq x_2 - 1 \leq \ldots \leq x_k - (k-1)\}$$

Direct, Uniform, Non-Distinct (2)

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

$$\overline{Cheung}_n^{N,m}(k) = m * Cheung_n^N(k) \tag{22}$$

where

$$Cheung_n^N(k) = [1 - \tilde{p}] \tag{23}$$

Direct, Uniform, Non-Distinct (3)

Theorem (Cheung (cont.))

with the following equivalent expressions for $\tilde{p}$:

$$\tilde{p} = \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \tag{24}$$

$$= \prod_{i=0}^{k-1} \frac{N-n+i}{N+i} \tag{25}$$

$$= \prod_{i=0}^{n-1} \frac{N-1-i}{N-1+k-i} \tag{26}$$

Direct, Uniform, Non-Distinct (4)

Proof(1):
Eq. 24 follows from the observation that the probability that some bucket does not contain any of the k possibly duplicate items is $\frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}}$.

Direct, Uniform, Non-Distinct (5)

Proof(2):
Eq. 25 follows from

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

Direct, Uniform, Non-Distinct (6)

Proof(3):

Eq. 26 follows from

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

Direct, Uniform, Non-Distinct (7)

Approximation for $\tilde{p}$:

$$(1 - n/N)^k$$

[Cardenas]

Direct, Uniform, Non-Distinct (8)

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

$$\mathcal{D}(N, k) = \frac{Nk}{N + k - 1} \tag{27}$$

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

Direct, Uniform, Non-Distinct (9)

Model switching:

$$\overline{\mathcal{Y}}_n^{N,m}(\mathit{Distinct}(N,k)) \approx \overline{\mathrm{Cheung}}_n^{N,m}(k)$$

[for $n \geq 5$]

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.

Direct, Non-Uniform, Distinct (2)

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

$$\mathcal{W}_{n_j}^{N}(k,j) = [1 - \frac{\binom{N-n_j}{k}}{\binom{N}{k}}] \quad (= \mathcal{Y}_{n_j}^{N}(k)) \tag{28}$$

and the expected number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) := \sum_{j=1}^{m} \mathcal{W}_{n_j}^{N}(k,j) \tag{29}$$

Direct, Non-Uniform, Distinct (3)

The product formulation in Eq. 20 of Theorem 2 results in a more efficient computation:

Corollary

If we lookup k distinct items, then the expected number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{j=1}^{m}(1 - p_j) \tag{30}$$

with

$$p_j = \begin{cases} \prod_{i=0}^{n_j-1} \frac{N-k-i}{N-i} & k \leq n_j \\ 0 & N - n_j < k \leq N \end{cases} \tag{31}$$

Direct, Non-Uniform, Distinct (4)

If we compute the p_j after we have sorted the n_j in ascending order, we can use the fact that

$$p_{j+1} = p_j * \prod_{i=n_j}^{n_{j+1}-1} \frac{N-k-i}{N-i}.$$

Direct, Non-Uniform, Distinct (5)

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

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{i=1}^{L} l_i \mathcal{Y}_{n_j}^{N}(k) \tag{32}$$

Direct, Non-Uniform, Distinct (6)

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:

Direct, Non-Uniform, Distinct (7)

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

$$\mathcal{X}_{n_j}^{N}(k,x) = \frac{\binom{n_j}{x}\binom{N-n_j}{k-x}}{\binom{N}{k}} \tag{33}$$

Further, the expected number of qualifying items in bucket j is

$$\overline{\mathcal{X}}_{n_j}^{N,m}(k) = \sum_{x=0}^{\min(k,n_j)} x\mathcal{X}_{n_j}^{N}(k,x) \tag{34}$$

In standard statistics books the probability distribution $\mathcal{X}_{n_j}^{N}(k,x)$ is called *hypergeometric distribution*.

Direct, Non-Uniform, Distinct (8)

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

$$\begin{aligned}
\overline{\mathcal{X}}^{N,m}_{n_j}(k) &= \sum_{x=0}^{\min(k,n)} x\mathcal{X}^N_n(k,x) \\
&= \sum_{x=1}^{\min(k,n)} x\mathcal{X}^N_n(k,x) \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} x \binom{n}{x} \binom{N-n}{k-x}
\end{aligned}$$

Direct, Non-Uniform, Distinct (9)

$$
\begin{aligned}
\overline{\mathcal{X}}^{N,m}_{n_j}(k) &= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} x \binom{n}{x} \binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{x}{1} \binom{n}{x} \binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{n}{1} \binom{n-1}{x-1} \binom{N-n}{k-x} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \sum_{x=0}^{y-1} \binom{n-1}{0+x} \binom{N-n}{(k-1)-x} \\
&= \dots
\end{aligned}
$$

(cont.)

Direct, Non-Uniform, Distinct (10)

$$
\begin{aligned}
\overline{\mathcal{X}}_{n_j}^{N,m}(k) &= \dots \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{n-1+N-n}{0+k-1} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{N-1}{k-1} \\
&= n\frac{k}{N} = \frac{k}{m}
\end{aligned}
$$

Direct, Non-Uniform, Distinct (11)

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

$$\begin{aligned} \mathcal{X}_1^N(k,x) &= \frac{\binom{1}{x}\binom{N-1}{k-1}}{\binom{N}{k}} \\ &= \frac{\binom{N-1}{k-1}}{\binom{N}{k}} \\ &= \frac{k}{N} \quad (= \frac{k}{m}) \end{aligned}$$

Since x can then only be zero or one, the average number of qualifying items a bucket contains is also $\frac{k}{N}$.

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]

Sequential: Vector of Bits (2)

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

$$\mathcal{B}_b^B(j) = \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \tag{35}$$

Sequential: Vector of Bits (3)

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.

Sequential: Vector of Bits (4)

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.

Sequential: Vector of Bits (5)

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

$$\begin{aligned}
\mathcal{B}_b^B(j) &= \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \\
&= \frac{\frac{(B-j-1)!}{(b-1)!((B-j-1)-(b-1))!}}{\frac{B!}{b!(B-b)!}} \\
&= \frac{(B-j-1)!\ \ b!(B-b)!}{(b-1)!((B-j-1)-(b-1))!\ \ B!}
\end{aligned}$$

Sequential: Vector of Bits (6)

$$
\begin{aligned}
\mathcal{B}_b^B(j) &= \frac{(B-j-1)!\ \ b!(B-b)!}{(b-1)!((B-j-1)-(b-1))!\ \ B!} \\
&= b\frac{(B-j-1)!\ \ (B-b)!}{((B-j-1)-(b-1))!\ \ B!} \\
&= b\frac{(B-j-1)!\ \ (B-b)!}{(B-j-b)!\ \ B!} \\
&= \frac{b}{B-j}\frac{(B-j)!\ \ (B-b)!}{(B-b-j)!\ \ B!} \\
&= \frac{b}{B-j}\prod_{i=0}^{j-1}(1-\frac{b}{B-i})
\end{aligned}
$$

This formula is useful when $\mathcal{B}_b^B(j)$ occurs in sums over j.

Sequential: Vector of Bits (7)

Corollary

Using the terminology of Theorem 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

$$\overline{\mathcal{B}}_b^B = \sum_{j=0}^{B-b} j\mathcal{B}_b^B(j) = \frac{B-b}{b+1} \tag{36}$$

Sequential: Vector of Bits (8)

Proof:

$$\begin{aligned}
\sum_{j=0}^{B-b} j\binom{B-j-1}{b-1} &= \sum_{j=0}^{B-b}(B-(B-j))\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{B-j-1}{b-1} - \sum_{j=0}^{B-b}(B-j)\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{b-1} - b\sum_{j=0}^{B-b}\binom{B-j}{b} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{j} - b\sum_{j=0}^{B-b}\binom{b+j}{b}
\end{aligned}$$

Sequential: Vector of Bits (9)

$$\begin{aligned}
\sum_{j=0}^{B-b} j\binom{B-j-1}{b-1} &= B\sum_{j=0}^{B-b}\binom{b-1+j}{j} - b\sum_{j=0}^{B-b}\binom{b+j}{b} \\
&= B\binom{(b-1)+(B-b)+1}{(b-1)+1} - b\binom{b+(B-b)+1}{b+1} \\
&= B\binom{B}{b} - b\binom{B+1}{b+1} \\
&= (B - b\frac{B+1}{b+1})\binom{B}{b}
\end{aligned}$$

With

$$\begin{aligned}
B - b\frac{B+1}{b+1} &= \frac{B(b+1)-(Bb+b)}{b+1} \\
&= \frac{B-b}{b+1}
\end{aligned}$$

Sequential: Vector of Bits (10)

Corollary

Using the terminology of Theorem 8, the expected total number of bits from the first bit to the last one, both included, is

$$\overline{\mathcal{B}}_{tot}(B, b) = \frac{Bb + b}{b + 1} \tag{37}$$

Sequential: Vector of Bits (11)

Proof:
We subtract from B the average expected number of zeros between the last one and the last bit:

$$\begin{aligned} B - \frac{B-b}{b+1} &= \frac{B(b+1)}{b+1} - \frac{B-b}{b+1} \\ &= \frac{Bb + B - B + b}{b+1} \\ &= \frac{Bb + b}{b+1} \end{aligned}$$

Sequential: Vector of Bits (12)

Corollary

Using the terminology of Theorem 8, the number of bits from the first one and the last one, both included, is

$$\overline{\mathcal{B}}_{1\text{-}span}(B, b) = \frac{Bb - B + 2b}{b + 1} \tag{38}$$

Sequential: Vector of Bits (13)

Proof (alternative 1):
Subtract from B the number of zeros at the beginning and the end:

$$\begin{aligned}\overline{B}_{\text{1-span}}(B,b) &= B - 2\frac{B-b}{b+1} \\ &= \frac{Bb + B - 2B + 2b}{b+1} \\ &= \frac{Bb - B + 2b}{b+1}\end{aligned}$$

Sequential: Vector of Bits (14)

Proof (alternative 2):
Add the number of zeros between the first and the last one and the number of ones:

$$\begin{aligned}
\overline{\mathcal{B}}_{\text{1-span}}(B,b) &= (b-1)\overline{\mathcal{B}}_b^B + b \\
&= (b-1)\frac{B-b}{b+1} + \frac{b(b+1}{b+1} \\
&= \frac{Bb - b^2 - B + b + b^2 + b}{b+1} \\
&= \frac{Bb - B + 2b}{b+1}
\end{aligned}$$

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?

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 $t_i = \sum_{l=0}^{i} n_i$ 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

$$C_{n_i}^{N,m}(k,j) = \frac{\binom{t_j}{k} - \binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{39}$$

Thus, the expected number of buckets that need to be examined in order to retrieve k distinct items is

$$\overline{C}_{n_i}^{N,m}(k) = \sum_{j=1}^{m} j C_{n_i}^{N,m}(k,j) = m - \frac{\sum_{j=1}^{m} \binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{40}$$

Sequential: Vector of Buckets (2)

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

$$A(N,k) = N - \sum_{i=1}^{N-1} Prob[Y \leq i] \tag{41}$$

where Y is a random variable for the last item in the sequence that occurs among the k items searched.

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.

Disk Drive Costs for N Uniform Accesses (2)

Remembering the description of disk drives, the total costs can be described as

$$C_{disk} = C_{cmd} + C_{seek} + C_{settle} + C_{rot} + C_{headswitch} \tag{42}$$

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.

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.

Disk Costs: Number of Qualifying Cylinder

The number of qualifying cylinders in exent 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:

$$Q_c(i) = S_{\text{cpe}}(i)\mathcal{Y}^{S_{\text{sec}}}_{D_{\text{zspc}}(i)}(N) = S_{\text{cpe}}(i)(1 - \frac{\binom{S_{\text{sec}} - D_{\text{zspc}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \tag{43}$$

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:

$$Q_t(i) = S_{\text{cpe}}(i) D_{\text{tpc}} \mathcal{Y}^{S_{\text{sec}}}_{D_{\text{zspt}}(i)}(N) = S_{\text{cpe}}(i) D_{\text{tpc}} (1 - \frac{\binom{S_{\text{sec}} - D_{\text{zspt}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \tag{44}$$

Number of Qualifying Tracks (2)

Just for fun, we calculate the number of qualifying sectors of an extent in zone i. It can be approximated by

$$Q_s(i) = S_{\text{cpe}}(i) D_{\text{zspc}}(i) \frac{N}{S_{\text{sec}}} \tag{45}$$

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

$$Q_c(i) = \overline{\mathcal{Y}}^{S_{\text{cpe}}(i) D_{\text{zspc}}(S_{\text{zone}}(i)), S_{\text{cpe}}(i)}_{D_{\text{zspc}}(S_{\text{zone}}(i))}(Q_s(i)) \tag{46}$$

If $Q_s(i)$ is not too small (e.g. > 4).

Command Costs

The command costs C_{cmd} are easy to compute. Every read of a sector requires the execution of a command. Hence

$$C_{cmd} = ND_{\text{cmd}}$$

estimates the total command costs.

Seek Costs

- often the dominant part of the costs
- we look at several alternatives from less to more precise models

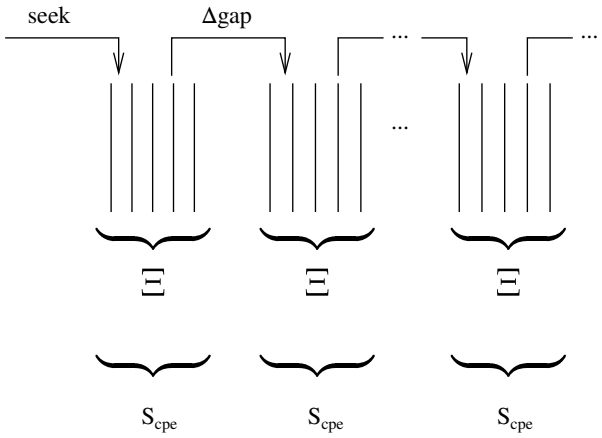
Seek Costs - Upper Bound

The first cylinder we have to visit requires a random seek with cost D_{seekavg}. (Truely upper bound: $D_{\text{fseek}}(D_{\text{cyl}} - 1)$)
After that, we have to visit the remaining $Q_c(i) - 1$ qualifying cylinders.
The segment spans a total of $S_{\text{clast}}(S_{\text{ext}}) - S_{\text{cfirst}}(1) + 1$ 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 1 gives us the upper bound

$$C_{seek}(i) \leq (Q_c(i) - 1) D_{\text{fseek}}\left(\frac{S_{\text{clast}}(S_{\text{ext}}) - S_{\text{cfirst}}(1) + 1}{Q_c(i) - 1}\right)$$

after we have found the first qualifying cylinder.

Seek Costs - Illustration



Seek Costs - Steps

Steps:

1. Estimate for $C_{seekgap}$
2. Estimates for $C_{seekext}(i)$

Seek Costs - Interextent Costs

The average seek cost for reaching the first qualifying cylinder is D_{seekavg}. How far within the first extent are we now? We use Corollary 4 to derive that the number of non-qualifying cylinders preceding the first qualifying one in some extent i is

$$\overline{\mathcal{B}}_{Q_c(i)}^{S_{\text{cpe}}(i)} = \frac{S_{\text{cpe}}(i) - Q_c(i)}{Q_c(i) + 1}.$$

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

$$\Delta_{\text{gap}}(i) := \overline{\mathcal{B}}_{Q_c(i)}^{S_{\text{cpe}}(i)} + S_{\text{cfirst}}(i+1) - S_{\text{clast}}(i) - 1 + \overline{\mathcal{B}}_{Q_c(i+1)}^{S_{\text{cpe}}(i+1)}$$

Using this, we get

$$C_{seekgap} = D_{\text{seekavg}} + \sum_{i=1}^{S_{\text{ext}}-1} D_{\text{fseek}}(\Delta_{\text{gap}}(i))$$

Seek Costs - Intraextent Costs (2)

Let us turn to $C_{seekext}(i)$. 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 6:

$$\Xi_{\text{ext}}(i) = \overline{\mathcal{B}}_{\text{1-span}}(S_{\text{cpe}}(i), Q_c(i))$$

Hence, $\Xi(i)$ is the minimal span of an extent that contains all qualifying cylinders.

Seek Costs - Intraextent Costs

Using $\Xi(i)$ and Qyang's Theorem 1, we can derive an upper bound for $C_{seekext}(i)$:

$$C_{seekext}(i) \leq (Q_c(i) - 1) D_{\text{fseek}}\left(\frac{\Xi(i)}{Q_c(i) - 1}\right) \tag{47}$$

Alternatively, we could formulate this as

$$C_{seekext}(i) \leq (Q_c(i) - 1) D_{\text{fseek}}\left(\overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)}\right) \tag{48}$$

by applying Corollary 4.

Seek Costs - Intraextent Costs (2)

A seemingly more precise estimate for the expected seek cost within the qualifying cylinders of an extent is derived by using Theorem 8:

$$C_{seekext}(i) = Q_c(i) \sum_{j=0}^{S_{\text{cpe}}(i)-Q_c(i)} D_{\text{fseek}}(j+1)\mathcal{B}_{Q_c(i)}^{S_{\text{cpe}}(i)}(j) \tag{49}$$

Settle Costs

The average settle cost is easy to calculate. For every qualifying cylinder, one head settlement takes place:

$$C_{settle}(i) = Q_c(i) D_{\text{rdsettle}} \tag{50}$$

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.

Rotational Delay (2)

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 5 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

$$C_{rot}(i) = Q_t(i) \;\; D_{\text{zscan}}(S_{\text{zone}}(i)) \;\; \overline{\mathcal{B}}_{\text{tot}}(D_{\text{zspt}}(S_{\text{zone}}(i)), Q_{\text{spt}}(i))$$

where $Q_{\text{spt}}(i) = \overline{\mathcal{W}}_1^{S_{\text{sec}}, D_{\text{zspt}}(S_{\text{zone}}(i))}(N) = D_{\text{zspt}}(S_{\text{zone}}(i)) \frac{N}{S_{\text{sec}}}$ is the expected number of qualifying sectors per track in extent i. In case $Q_{\text{spt}}(i) < 1$, we set $Q_{\text{spt}}(i) := 1$.

Rotational Delay (3)

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 7. For (2) we again use Corollary 5.

$$C_{rot}(i) = S_{\text{cpe}}(i)\ D_{\text{tpc}}\ D_{\text{zscan}}(S_{\text{zone}}(i)) * \sum_{j=1}^{\min(N, D_{\text{zspt}}(S_{\text{zone}}(i)))} \mathcal{X}'^{S_{\text{sec}}}_{D_{\text{zspt}}(S_{\text{zone}}(i))}(N, j)\ \overline{\mathcal{B}}_{\text{tot}}(D_{\text{zspt}}(S_{\text{zone}}(i)), j)$$

Rotational Delay (4)

Yet another approach:
Split the total rotational delay into two components:

1. $C_{rotpass}(i)$ measures the time needed to skip unqualifying sectors
2. $C_{rotread}(i)$ that for scanning the qualifying sectors

Then

$$C_{rot} = \sum_{i=1}^{S_{\text{ext}}} C_{rotpass}(i) + C_{rotread}(i)$$

where the total transfer cost of the qualifying sectors can be estimated as

$$C_{rotread}(i) = Q_s(i) \;\; D_{\text{zscan}}(S_{\text{zone}}(i))$$

Rotational Delay (5)

Let us treat the first component ($C_{rotpass}(i)$).
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

$$\frac{D_{\text{zspt}}(S_{\text{zone}}(i))}{2j}$$

Rotational Delay (6)

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

$$N_s(j,i) = \frac{D_{\text{zspt}}(S_{\text{zone}}(i))}{2j} + (j-1)\frac{D_{\text{zspt}}(S_{\text{zone}}(i)) - j}{j}$$

Rotational Delay (7)

Using again Theorem 7, the expected rotational delay for the unqualifying sectors then is

$$
\begin{aligned}
C_{rotpass}(i) &= S_{\text{cpe}}(i)\ \ D_{\text{tpc}}\ \ D_{\text{zscan}}(S_{\text{zone}}(i)) \\
&\quad * \sum_{j=1}^{\min(N, D_{\text{zspt}}(S_{\text{zone}}(i)))} \mathcal{X}^{S_{\text{sec}}}_{D_{\text{zspt}}(S_{\text{zone}}(i))}(N, j) N_s(j, i)
\end{aligned}
$$

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

$$C_{headswitch} = \sum_{i=1}^{S_{\text{ext}}} (Q_t(i) - Q_c(i)) \;\; D_{\text{hdswitch}} \tag{51}$$

where Q_t is the average number of tracks qualifying in an extent.

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.

Selectivity Estimations

- previous slides assume that we "know" how many tuples qualify
- but this has to be estimated somehow
- similar for join ordering algorithms etc.
- cardinalities (and thus selectivities) are fundamental for query optimization
- we will now look at deriving some estimations

Examples

SQL examples for typical selectivity problems:

- **select** *
  from rel r
  where r.a=10
- **select** *
  from rel r
  where r.b>2
- **select** *
  from rel1 r1,rel2 r2
  where r1.a=r2.b

The different problems require different approaches.

Heuristic Estimations

Some commonly used selectivity estimations:

predicate	selectivity	requirement
$A = c$	$1/\|D(A)\|$	if index on A
	$1/10$	otherwise
$A > c$	$(\max(A) - c)/(\max(A) - \min(A))$	if index on A, interpol.
	$1/3$	otherwise
$A_1 = A_2$	$1/\max(\|D(A_1)\|, \|(D(A_2)\|)$	if index on A_1 and A_2
	$1/\|D(A_1)\|$	if index on A_1 only
	$1/\|D(A_2)\|$	if index on A_2 only
	$1/10$	otherwise

Note: Without further statistics, $|D(A)|$ is typically only known (easily estimated) if A is a key or there is an index on A.

Using Histograms

- selectivity can be calculated easily by looking at the real data
- not feasible, therefore look at aggregated data
- histograms partition the data values into buckets

A histogram $H_A : B \to \mathbb{N}$ over a relation R partitions the domain of the aggregated attribute A into disjoint buckets B, such that

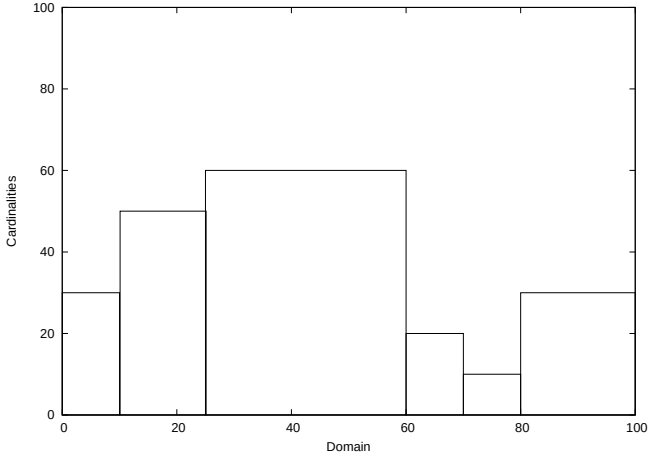
$$H_A(b) = |\{r | r \in R \wedge R.A \in b\}|$$

and thus $\sum_{b \in B} H_A(b) = |R|$.

Choosing B is very important, as we will see on the next slides.

Using Histograms (2)

A rough histogram might look like this:



Using Histograms (3)

Given a histogram, we can approximate the selectivities as follows:

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

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

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

Using Histograms - Remarks

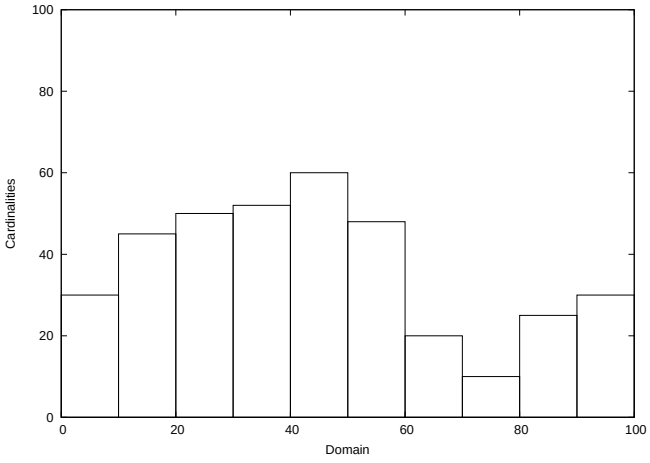
- estimations on previous slide can be improved
- in particular, the $A = c$ case is only a rough approximation
- requires more information
- if we interpret the histogram as a density function, $P(A = c) = 0$!
- a reasonable upper bound, though
- the $A > c$ case is more sound
- $A_1 = A_2$ assumes independence etc.

Building Histograms

- the buckets chosen greatly affect the overall quality
- histogram does not discern items within one bucket
- therefore: try to put items into different buckets
- how to choose the buckets?
- typical constraint: histogram size. n buckets (fixed)
- for a given set of data items, find a good histogram with n buckets
- additional constraint: data distribution is unknown (real data)

Building Histograms - Equiwidth

Partitions the domain into buckets with a fixed width



Building Histograms - Equiwidth (2)

Advantages:

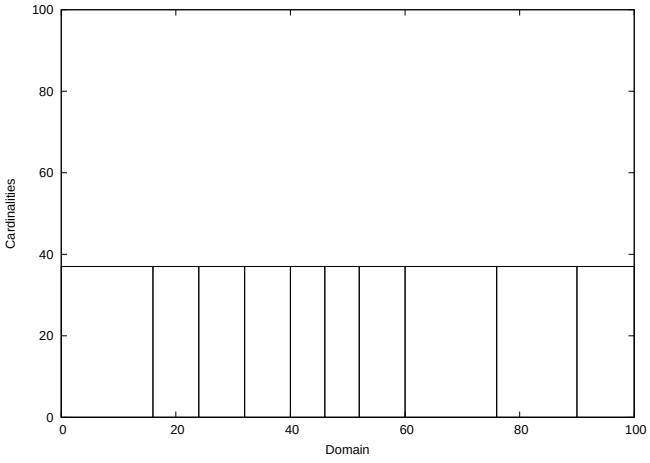
- easy to compute
- bucket boundaries can be computed (require no space)

Disadvantages:

- samples the domain uniformly
- does not handle skewed data well
- skew can lead to very uneven buckets
- greater estimation error in large buckets
- particular bad for zipf-like distributions

Building Histograms - Equidepth

Chooses the buckets to contain the same number of items



Building Histograms - Equidepth (2)

Advantages:

- adopts to data distribution
- reduces maximum error

Disadvantages:

- more involved (sort or similar)
- both boundaries and depth have to be stored (ties)

Very common histogram building technique

Building Histograms - Interpolation

- data is usually not completely random
- can we increase accuracy by interpolation?
- either within buckets (common) or instead of buckets (uncommon)
- histogram is a density function, not continuous, hard to interpolate
- use the equivalent distribution function instead
- very good for estimating $A > c$

Discussion

- estimations more complex in practice
- potentially different goals: maximum vs. average error
- histograms for derived values
- histogram convolution
- handling correlations
- multi-dimensional histograms
- cardinality estimators (sketches, MIPS etc.)

5. Physical Properties

- Why Properties
- Distributed Queries
- Ordering
- Grouping
- DAGs

Why Properties

- query optimizer chooses the cheapest equivalent plan
- join ordering: the cheapest plan with the same set of relations
- but: plans might produce the same result but behave differently
- for example sort-merge vs. hash join
- hash join could be cheaper, but sort-merge still pay of later
- not directly comparable

Why Properties (2)

How to handle logical equivalent but un-comparable plans?

- one alternative: encode differences into search space
- for example, different plans for sorting vs. hashing
- but: search space explodes
- some aspects like "sorting" consist of many alternatives
- further: if sorting is cheaper than hashing, we usually prefer sorting
- direct encoding into search space too wasteful
- use (physical) properties instead

Using Properties

A physical property P defines a partial relation $\leq_P$ with the following characteristics among plans:

If two plans p_1 and p_2 are logically equivalent,

- $p_1 \leq_P p_2$ if p_2 dominates p_1 concerning P
- $p_1 =_P p_2$ is p_1 and p_2 are comparable concerning P ($p_1 \leq_P p_2 \wedge p_2 \leq_P p_1$)

A plan can only be pruned if it is dominated or comparable

Using Properties (2)

With properties, the query optimizer does not maintain a single solution but a set of solutions for each subproblem:

storeSolution(S,p)
  $P = dpTable[S]$
  $P' = \emptyset$
  for $\forall p' \in P$ {
    if $p \leq p' \wedge C(p) \geq C(p')$
      return
    if $\neg(p' \leq p \wedge C(p') \geq C(p))$
      $P' = P' \cup \{p'\}$
  }
  $dpTable[S] = P' \cup \{p\}$

Using Properties (3)

- algorithm too simple
- properties can be *enforced*
- Enforcers make plans comparable
- allows for more pruning
- will see examples for this
- combination of multiple properties needs some care

Distributed Queries

- distributed query processing keeps track of the *site*
- intermediate results can be computed at different sites
- a physical property is therefore the site of the intermediate result
- very simple property, site is either the same or different
- more plans comparable with enforcers

Distributed Queries - Comparing Plans

Two plans are comparable, if they produce their result on the same site or the difference is larger than the shipment costs:

prune(p_1,p_2)
  if $p_1.site = p_2.site$
    return $(C(p_1) \leq C(P_2))?p_1 : p_2$
  if $C(p_1) + C(\text{transfer } p_1) \leq C(P_2)$
    return p_1
  if $C(p_2) + C(\text{transfer } p_2) \leq C(P_1)$
    return p_2
  return $\{p_1, p_2\}$

Distributed Queries - Effect on Search organization

- previous slide described how to compare plans, but not how to generate them
- plans must be generated for desired sites
- one possibility: generate plans for all sites
- can be quite wasteful
- alternative: generate plans (for sites) on demand
- difficult to do bottom-up
- usual technique: determine relevant sites beforehand and generate plans for them
- this sites would be called *interesting*

Ordering

- physical tuple order is the classical physical property
- equivalent plans produce the same tuples, but (potentially) in different order
- tuple ordering is very important for many operators
- sort-merge, group by etc.
- explicit order by
- access optimization

Ordering (2)

An ordering O is a list of attributes $(A_1, \ldots, A_n)$

A tuple stream satisfied an ordering O, if the tuples are sorted according to A_1 and for each $1 < i \leq n$ the tuples are sorted on A_i for identical values of $A_1, \ldots, A_{i-1}$.

Interesting Orderings

- optimizer uses existing orderings, or creates new ones (enforcers)
- set of potential orderings very large
- too many orderings increase the search space
- concentrate on relevant orderings: *interesting orderings*

ordering is interesting, if

- requested by the user
- physically available
- useful for a planed operator

Interesting Orderings (2)

- ordering is characterized by a list of attributes
- if a tuple stream is ordered on $a_1, \ldots, a_n, a_{n+1}$, it is also ordered on $a_1, \ldots, a_n$
- orderings are affected by operators, in particular they can grow
- therefore, each prefix of an interesting ordering is also interesting
- (somewhat implementation dependent)
- non-interesting orderings are "forgotten" by the optimizer to reduce the search space

Physical vs. Logical Ordering

- the *physical* ordering is the actual order of tuples on disk/in a tuple stream
- the *logical* ordering is the ordering satisfied by the tuples
- the query optimizer can usually only reason about the logical ordering
- a tuple stream may satisfy multiple logical orderings
- the logical ordering can change, although the physical ordering did not!

Functional Dependencies

Logical Ordering is affected by functional dependencies:

- induces by operators
- $\sigma_{a=\cos(b)} \Rightarrow \{b \rightarrow a\}$
- $\sigma_{a=b} \Rightarrow \{a \rightarrow b, b \rightarrow a\}$ (even stronger)
- $\sigma_{a=10} \Rightarrow \{\emptyset \rightarrow a\}$
- complex operators can induce multiple FDs
- FDs allow for deriving new logical orderings

Example

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

Interesting ordering:	(a), (b), (a, b) and (a, b, c)
Interesting groupings:	$\{b\}$
Functional dependencies:	$b \rightarrow c$, $b \rightarrow d$

- Note: for $\{b\}$ grouping is sufficient (next section)

Materializing Orderings

- the query optimizer might just maintain a set of all orderings satisfied by a plan
- but FDs increase the set
- $\texttt{sort}(a) \rightarrow \texttt{select}(a = b)$
- is compatible with $(a), (a, b), (b), (b, a)$
- set can grow exponentially
- maintaining set of orderings not feasible

Reducing Orderings

Simmen et al. [17] proposed the following scheme:

- remember the base ordering
- remember all functional dependencies
- whenever testing for an ordering, reduce by base ordering and functional dependency
- apply prefix test after this

Reducing Orderings - Example

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

1. reduce ordering to (b, e)
2. reduce test to (a, b, e)
3. reduce test to (b, e)
4. test for prefix

but:

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

Reducing Orderings - Discussion

- back-to-front rule is not enough $((a),(a,b,c),\{a \rightarrow b, a, b \rightarrow c\})$
- avoiding this requires normalizing the FDs, which is very expensive
- reduction has to be done for each test
- tests happen very frequently (nearly each operator tests)
- memory management is a problem
- better than materializing orderings, but not optimal

Required Interface for Orderings

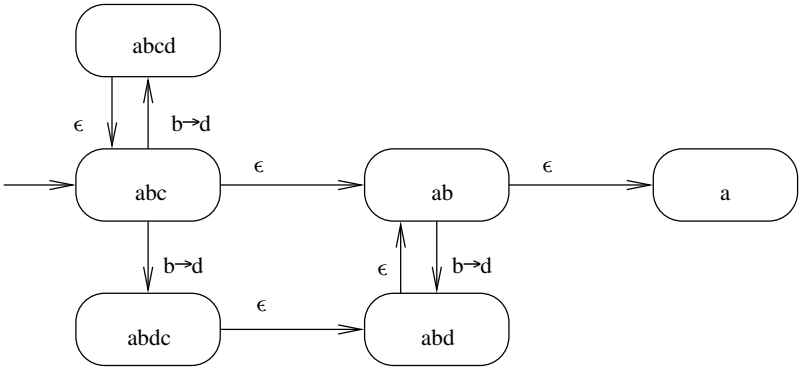
Query optimizer just requires few operations:

- initialization
- test for an ordering
- apply function dependency

Concrete ordering not required

Encoding Orderings as FSMs

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

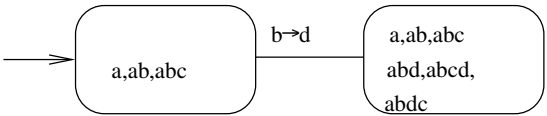


Encoding Orderings as FSMs (2)

- FSM described physical orderings
- pretends that FD changes physical ordering
- might be non-deterministic
- has to become deterministic
- conversion in DFSM (via NFA→DFA)

Encoding Orderings as FSMs (3)

DFSM



- node contains all possible physical orderings $\Longrightarrow$ logical orderings
- operating on the DFSM is very efficient
- only problem: how to construct it (efficiently)

Ordering FSM Construction - Overview

1. Determine the input
   - 1.1 Determine interesting orders
   - 1.2 Determine sets of functional dependencies
2. Construct the NFSM
   - 2.1 Construct nodes of the NFSM
   - 2.2 Filter functional dependencies
   - 2.3 Add edges to the NFSM
   - 2.4 Prune the NFSM
   - 2.5 Add artificial start node and edges
3. Construct the DFSM - convert the NFSM into a DFSM
4. Precompute values
   - 4.1 Precompute the compatibility matrix
   - 4.2 Precompute the transition table

Ordering FSM Construction - Determining the Input

- interesting orders (requested, required, index)
- $O_I = O_P \cup O_T$ (produced vs. tested, allows pruning)
- functional dependencies (operators, keys)
- handles for O(1) comparisons

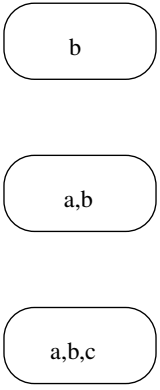
E.g.

$$\mathcal{F} = \{\{b \rightarrow c\}, \{b \rightarrow d\}\}$$

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

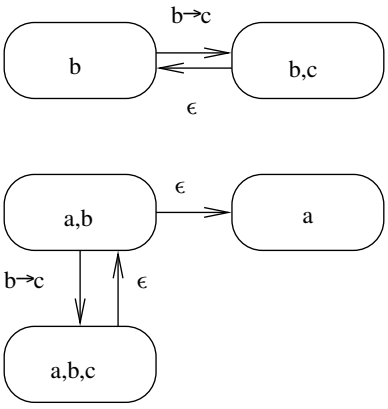
Ordering FSM Construction - Constructing the NFSM

Initial nodes for O_I



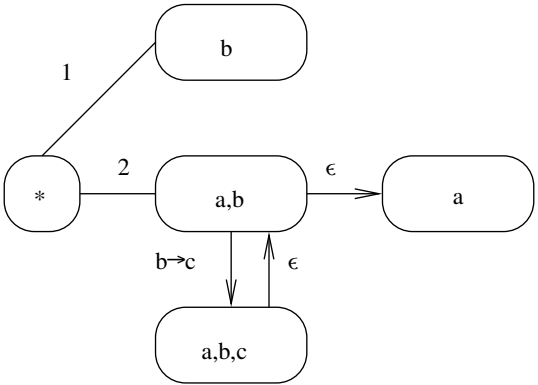
Ordering FSM Construction - Constructing the NFSM (2)

Edges for F. Creates artificial node (can be pruned)



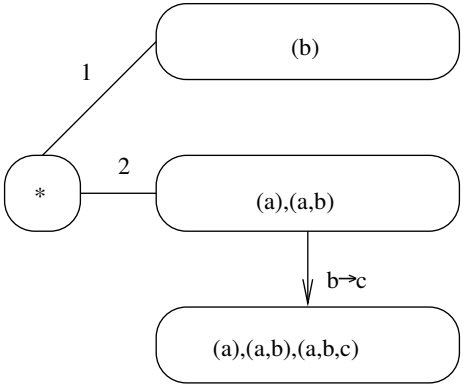
Ordering FSM Construction - Constructing the NFSM (3)

Edges for initialization. (b, c) was pruned.



Ordering FSM Construction - Constructing the DFSM

Standard conversion algorithm



- tests for O_T are precomputed (materialized)

Pruning Techniques

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

Note: search space reduction is a major benefit!

Discussion

- orderings essential for query optimizations
- but orderings increase the search space
- management involved
- FSM representation needs $O(1)$ time and space during optimization
- queried very often, but also very fast
- help reduce the search space

Grouping

- sometimes ordering is a too strong requirement
- some operators do not need an order, they just want continuous blocks for values
- group by operators are a typical example
- therefore: grouping property
- exploiting groupings is similar to exploiting orderings

Grouping (2)

A grouping G is a set of attributes $\{A_1, \ldots, A_n\}$

A tuple stream satisfies a grouping G, if tuples with the same values for $A_1, \ldots, A_n$ are placed next to each other.

Note that the attributes within a grouping are unordered

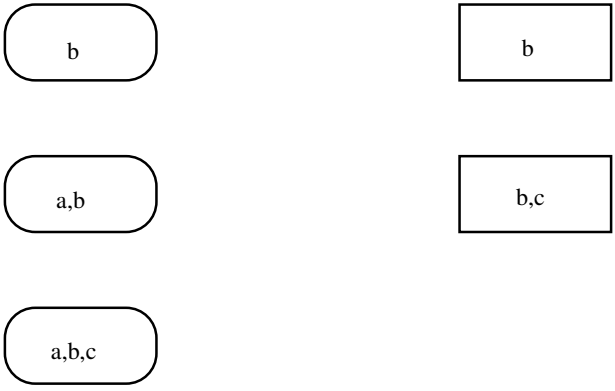
Ordering vs. Grouping

- ordering is a much stronger requirement than grouping
- every tuple stream that satisfies an ordering $O = (A_1, \ldots, A_n)$ also satisfies the grouping $G = \{A_1, \ldots, A_n\}$
- but there is not prefix deduction for groupings
- a tuple stream satisfying $\{A_1, A_2\}$ does not necessarily satisfy $\{A_1\}$
- could be derived from ordering information
- both types should be handled simultaneously

Integrating Grouping into Ordering Processing

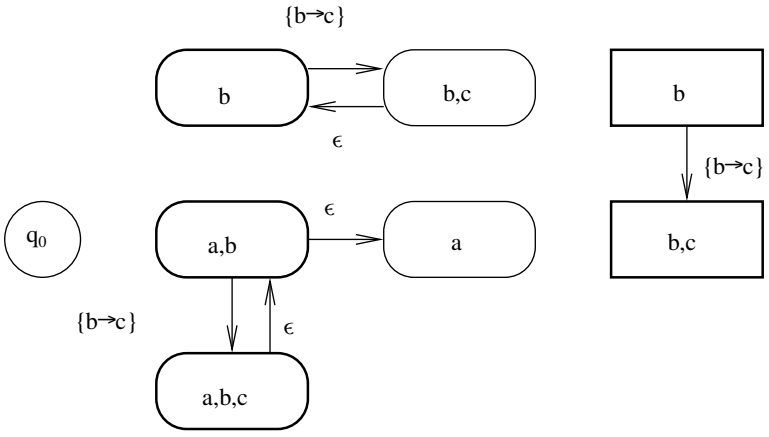
- groupings are similar to orderings
- can be modelled as FSMs, too (less edges, though)
- idea: build one big integrated FSM
- edges from orderings to corresponding groupings
- unifies these properties, makes pruning etc. much easier

Constructing a Unified FSM



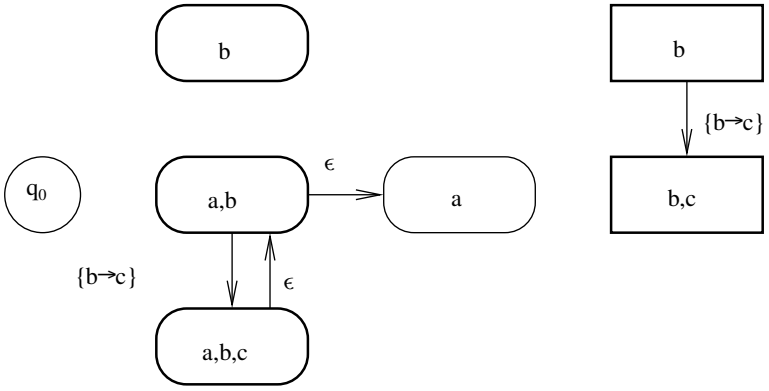
- create states for interesting orderings/groupings

Constructing a Unified FSM (2)



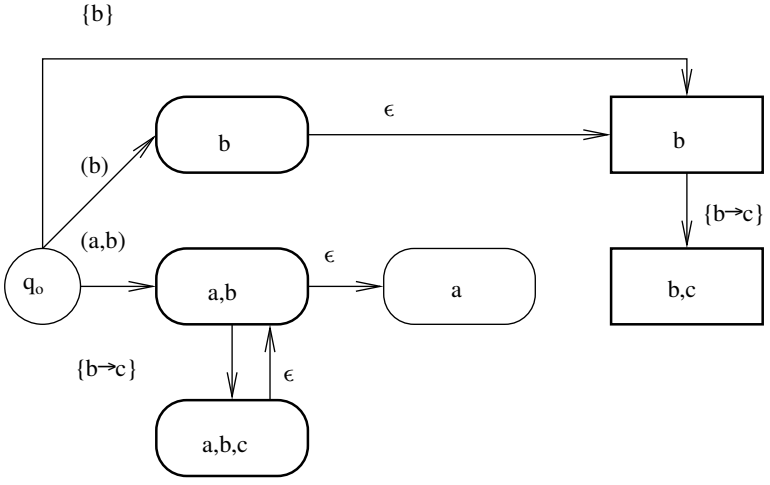
- consider functional dependencies
- note: no ϵ edge between groupings

Constructing a Unified FSM (3)



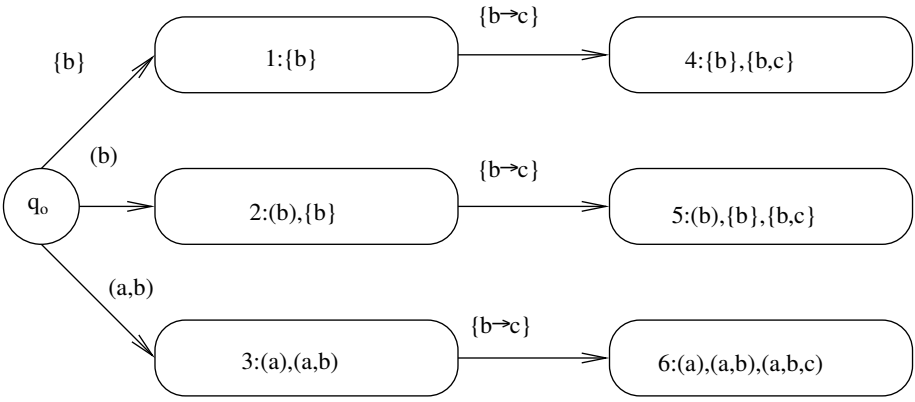
- prune artificial nodes

Constructing a Unified FSM (4)



- add additional edges for initialization

Constructing a Unified FSM (4)



- construct final DFSM

Discussion

- algorithm for groupings similar to orderings
- include pruning etc.
- unified handling very nice
- easy integration of both into the query optimizer
- FSM representation very fast
- only constant space per plan

DAGs

- execution plans until now were trees
- each operator has one consumer (except the root)
- no overlap
- very easy data flow
- but too limited in expressiveness
- a generalized plan structure requires some care (in this case a new kind of properties)

DAGs (2)

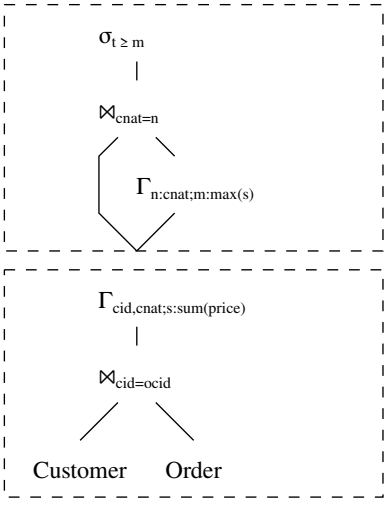
DAG - directed acyclic graph

More general than a tree, an operator can have more than one parent. Allows for more efficient plans.

Motivation for DAGs

common: views or shared expressions

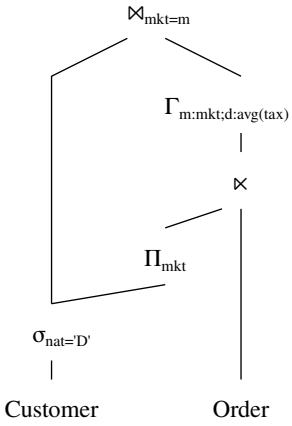
- recognized e.g. by DB2
- uses buffering
- parts optimized independently
- not really a DAG then



Motivation for DAGs (2)

magic sets

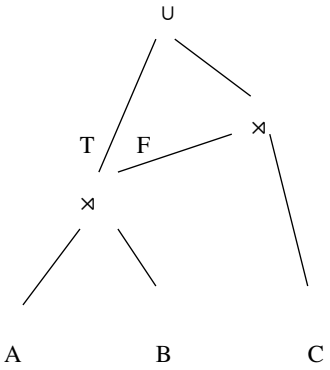
- propagate domain information
- nice optimization, but requires DAGs



Motivation for DAGs (3)

bypass plans

- handle tuples different depending on predicates
- more efficient for disjunctive queries
- more complex data flow

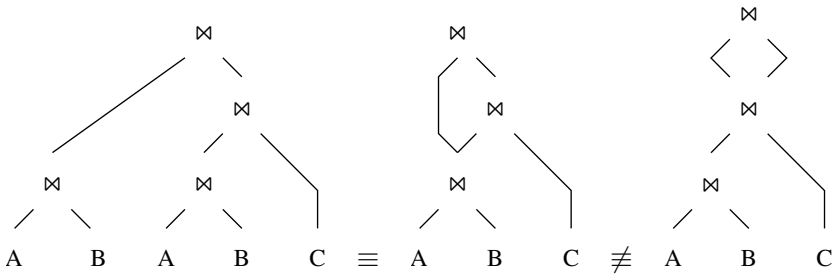


Motivation for DAGs (4)

- also XPath/XQuery evaluation, distributed queries, dependent join optimizations, ...
- optimizations not always beneficial, proper plan generation required
- buffering/temp reduces benefit, "real" execution required

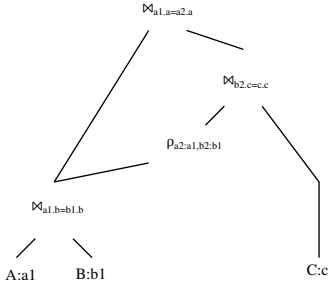
goal: generic DAG support

DAG Generation - Correctness Problems



- equivalences difficult to check
- here joins (apparently) not freely reorderable
- known equivalences not directly applicable

DAG Generation - Correctness Problems (2)



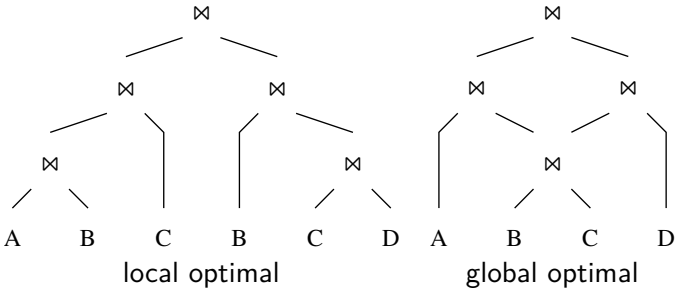
- idea: sharing through renaming $\Longrightarrow$ *share equivalence*
- formal criteria to detect equivalent subproblems
- create logical trees, allows for reusing known equivalences

Share Equivalence

$$A \equiv_S B \text{ iff } \exists_{\delta_{A,B}:\mathcal{A}(A)\to\mathcal{A}(B) \textit{ bijective}} \rho_{\delta_{A,B}}(A) = B$$

- difficult to test in general
- but constructive definition simple
- can be computed easily
- will be the base of a property (next slides)

DAG Generation - Optimal Substructure



- shared plans destroy optimal substructure
- idea: encode sharing into the search space
- *share equivalence* for operators
- creates equivalence classes, describes possibilities to share

DAG Generation - Optimal Substructure (2)

- generalize share equivalence from plans to operators
- would create share equivalent plans if the input were share equivalent
- classifies operators into equivalence classes
- only one operator from an equivalence class is relevant (representative)
- annotate each plan with the equivalence class (property)
- keep plans if they offer more classes (more sharing)
- note: only whole trees can be shared

DAG Generation - Search

Search component has to be adjusted:

- incorporate share equivalence
- try to rewrite problems as representatives
- if completely possible (whole tree) only use representatives
- creates implicit renames
- allows for reusing results
- adjust pruning, too

Discussion

- DAGs allow for much better plans
- generation somewhat involved
- share equivalence as property guarantees optimal solution
- many details omitted here
- cost model
- execution

End of Slides (for now)

[1] Leonidas Fegaras.
A new heuristic for optimizing large queries.
In *DEXA*, pages 726–735, 1998.

[2] Toshihide Ibaraki and Tiko Kameda.
On the optimal nesting order for computing n-relational joins.
ACM Trans. Database Syst., 9(3):482–502, 1984.

[3] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo.
Optimization of nonrecursive queries.
In *VLDB*, pages 128–137, 1986.

[4] Chiang Lee, Chi-Sheng Shih, and Yaw-Huei Chen.
Optimizing large join queries using a graph-based approach.
IEEE Trans. Knowl. Data Eng., 13(2):298–315, 2001.

[5] Guido Moerkotte and Thomas Neumann.
Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products.

In *VLDB*, pages 930–941, 2006.

[6] Thomas Neumann.
Query simplification: graceful degradation for join-order optimization.
In *SIGMOD Conference*, pages 403–414, 2009.

[7] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten.
The complexity of transformation-based join enumeration.
In *VLDB*, pages 306–315, 1997.

[8] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten.
Fast, randomized join-order selection - why use transformations?
In *VLDB*, pages 85–95, 1994.

[9] Donald Kossmann and Konrad Stocker.
Iterative dynamic programming: a new class of query optimization algorithms.
ACM Trans. Database Syst., 25(1):43–82, 2000.

[10] Philip A. Bernstein and Nathan Goodman.
Power of natural semijoins.

SIAM J. Comput., 10(4):751–771, 1981.

[11] Mihalis Yannakakis.
Algorithms for acyclic database schemes.
In *VLDB*, pages 82–94. IEEE Computer Society, 1981.

[12] Albert Atserias, Martin Grohe, and Dániel Marx.
Size bounds and query plans for relational joins.
In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008.

[13] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra.
Worst-case optimal join algorithms: [extended abstract].
In Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012.

[14] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann.

Adopting worst-case optimal joins in relational database systems.
Proc. VLDB Endow., 13(11):1891–1904, 2020.

[15] Todd L. Veldhuizen.
Leapfrog triejoin: a worst-case optimal join algorithm.
CoRR, abs/1210.0481, 2012.

[16] Susan Tu and Christopher Ré.
Duncecap: Query plans using generalized hypertree decompositions.
In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 2077–2078. ACM, 2015.

[17] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus.
Fundamental techniques for order optimization.
In *SIGMOD*, pages 57–67, 1996.