

Query Optimization

Thomas Neumann

April 21, 2009

Overview

1. Introduction
2. Textbook Query Optimization
3. Join Ordering
4. Accessing the Data
5. Physical Properties
6. Query Rewriting
7. Self Tuning

1. Introduction

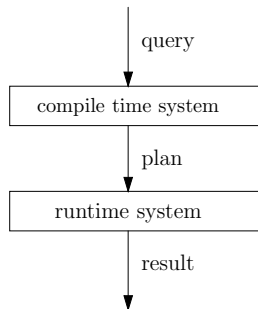
- Overview Query Processing
- Overview Query Optimization
- Overview Query Execution

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

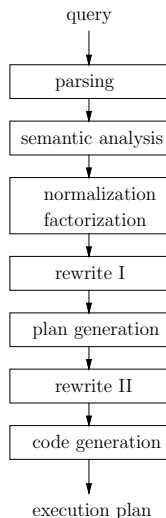
Overview Query Processing



- 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



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

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

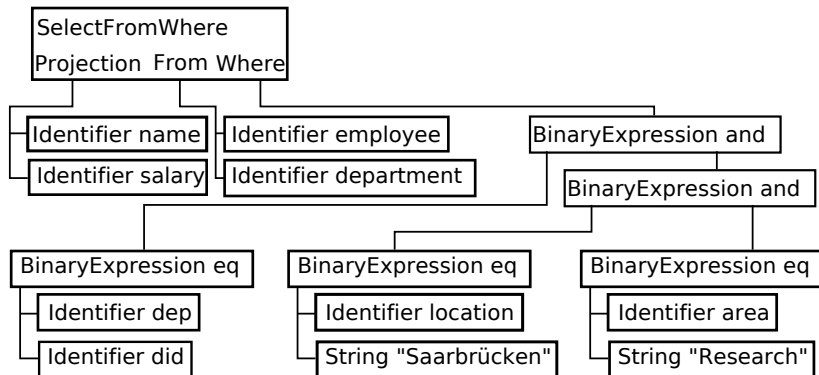
Processing Example - Input

```
select name, salary
from employee, department
where dep=did
and location="Saarbrücken"
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.

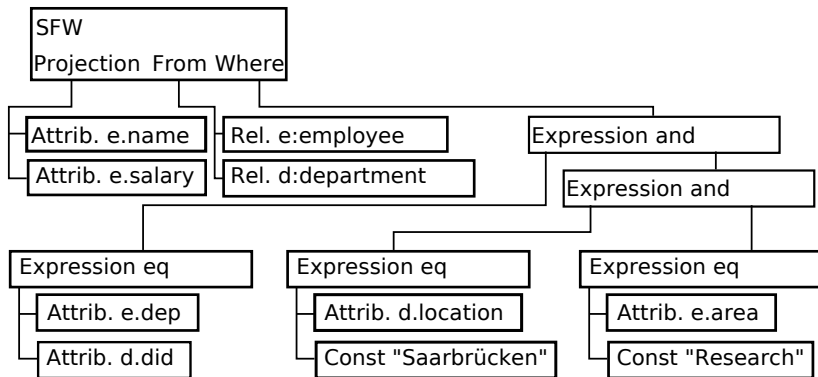
Processing Example - Parsing

Constructs an AST from the input



Processing Example - Semantic Analysis

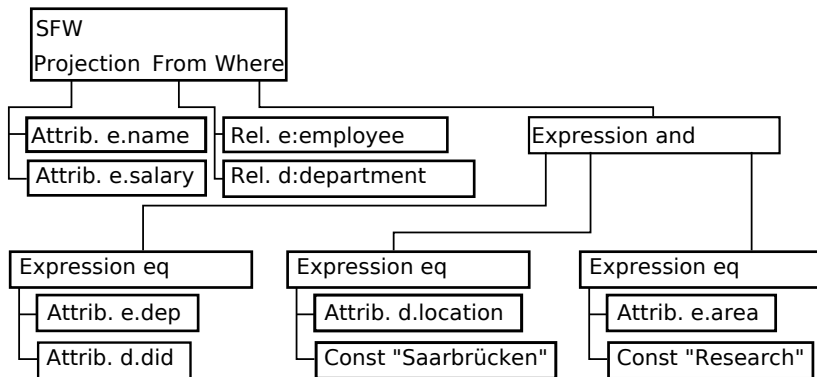
Resolves all variable binding, infers the types and checks semantics



Types omitted here, result is *bag* \langle *string*, *number* \rangle

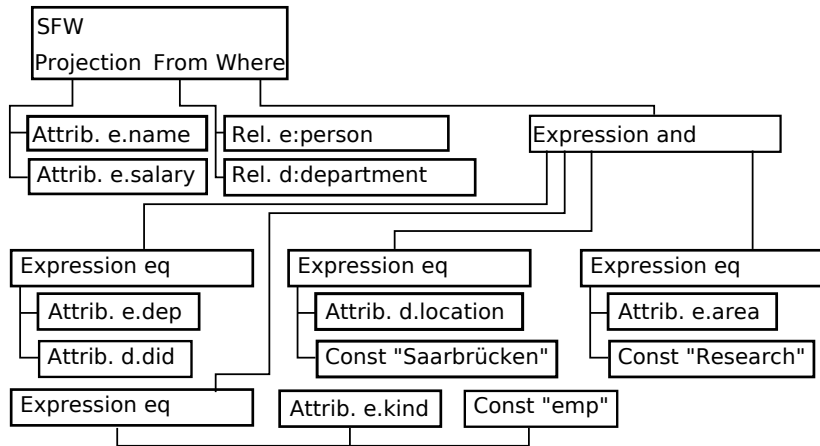
Processing Example - Normalization

Normalizes the representation, factorizes common expressions, folds constant expressions



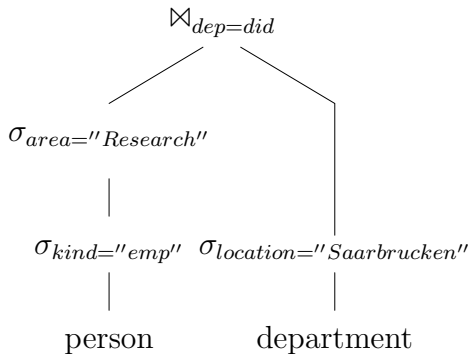
Processing Example - Rewrite I

resolves views, unnests nested expressions, expensive optimizations



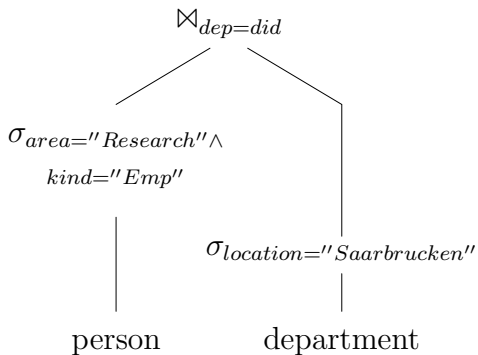
Processing Example - Plan Generation

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 "Saarbr\u00f6cken" @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
    ]
  ]
  <#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:

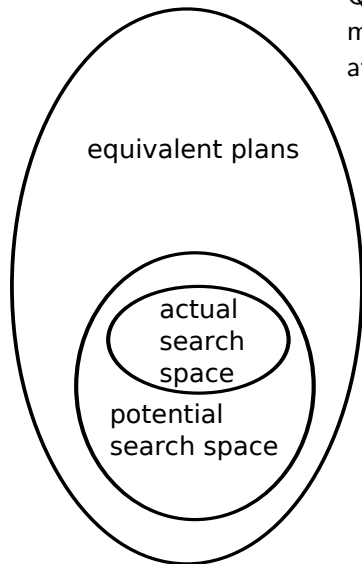
- find a cheaper/the cheapest expression that is equivalent to the first one

Problems:

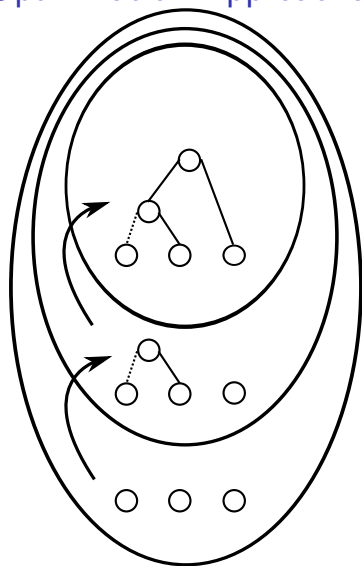
- the set of possible expressions is huge
- testing for equivalence is difficult/impossible in general
- the query is given in a calculus and not an algebra (this is also an advantage, though)
- even "simpler" optimization problems (e.g. join ordering) are typically NP hard in general

Search Space

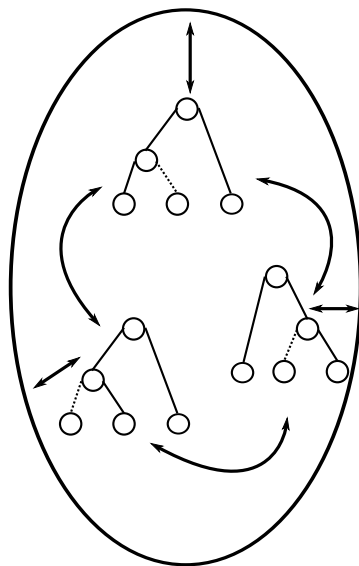
Query optimizers only search the "optimal" solution within the limited space created by known optimization rules



Optimization Approaches



constructive



transformative

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 produced 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, $\{x \mid x \in e_1 \wedge x \notin e_2\}$
$\rho_{a \rightarrow b}(e)$	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, $\{x \circ y \mid_{\mathcal{A}(e_2) \setminus \mathcal{A}(e_1)} \mid x \in e_1 \wedge y \in e_2 \wedge x =_{\mathcal{A}(e_1) \cap \mathcal{A}(e_2)} y\}$

$e_1 \div e_2$ division, $\{x \mid_{\mathcal{A}(e_1) \setminus \mathcal{A}(e_2)} \mid x \in e_1 \wedge \forall y \in e_2 : x =_{\mathcal{A}(e_1) \cap \mathcal{A}(e_2)} y\}$

$e_1 \ltimes_p e_2$ semi-join, $\{x \mid x \in e_1 \wedge \exists y \in e_2 : p(x \circ y)\}$

$e_1 \triangleright_p e_2$ anti-join, $\{x \mid x \in e_1 \wedge \nexists y \in e_2 : p(x \circ y)\}$

$e_1 \bowtie_p e_2$ outer-join, $(e_1 \ltimes_p e_2) \cup \{x \circ \circ_{a \in \mathcal{A}(e_2)} (a : null) \mid x \in (e_1 \triangleright_p e_2)\}$

$e_1 \bowtie_p e_2$ full outer-join, $(e_1 \ltimes_p e_2) \cup (e_2 \ltimes_p e_1)$

Relational Algebra - Extensions

The algebra needs some extensions for real queries:

- map/function evaluation

$$\chi_{a:f}(e) = \{x \circ (a : f(x)) \mid x \in e\}$$

- group by/aggregation

$$\Gamma_{A;a:f}(e) = \{x \circ (a : f(y)) \mid x \in \Pi_A(e) \wedge y = \{z \mid z \in e \wedge \forall a \in A : x.a = z.a\}\}$$

- dependent join (djoin). Requires $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$

$$e_1 \bowtie_p e_2 = \{x \circ y \mid x \in e_1 \wedge y \in e_2(x) \wedge p(x \circ y)\}$$

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