Query Optimization

Usually: Prof. Thomas Neumann
Today: Andrey Gubichev

Overview

Disclaimer

- This course is about how query optimizers work and what are they good for
- That is, about general principles and specific algorithms that are employed by real database systems
- (With lots of algorithms)
- Sometimes, we will talk about optimization of some general classes of SQL queries
- However, we will not study system-specific settings (how to tune Oracle/MS SQL/MySQL). Read manuals!

1. Introduction

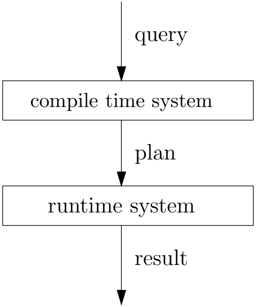
- Overview Query Processing
- Overview Query Optimization
- Overview Query Execution

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

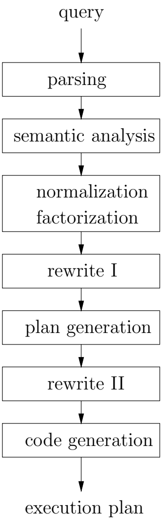
Overview Query Processing



- 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

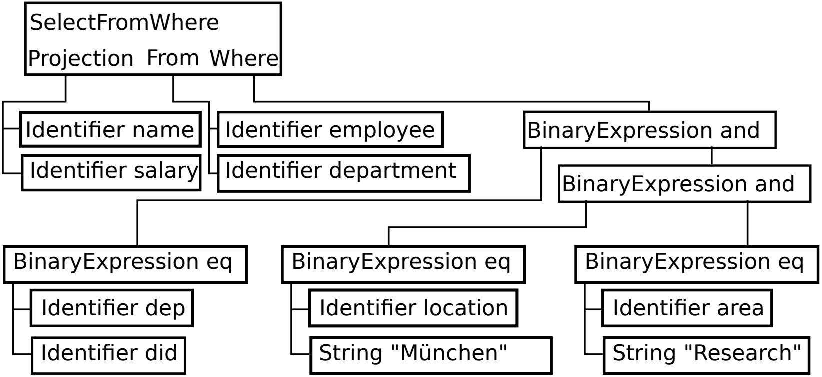
Processing Example - Input

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.

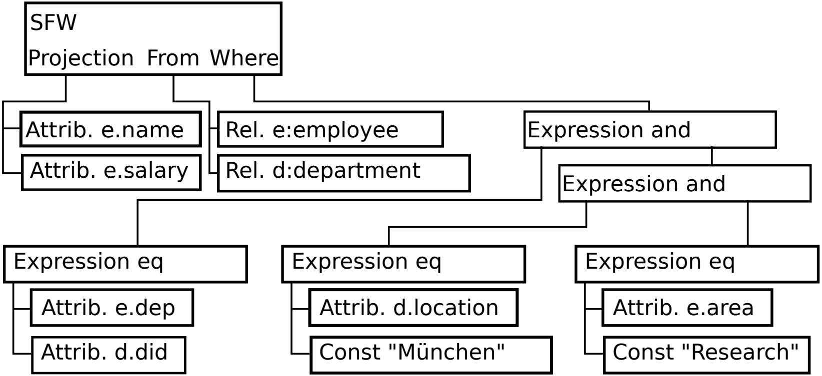
Processing Example - Parsing

Constructs an AST from the input



Processing Example - Semantic Analysis

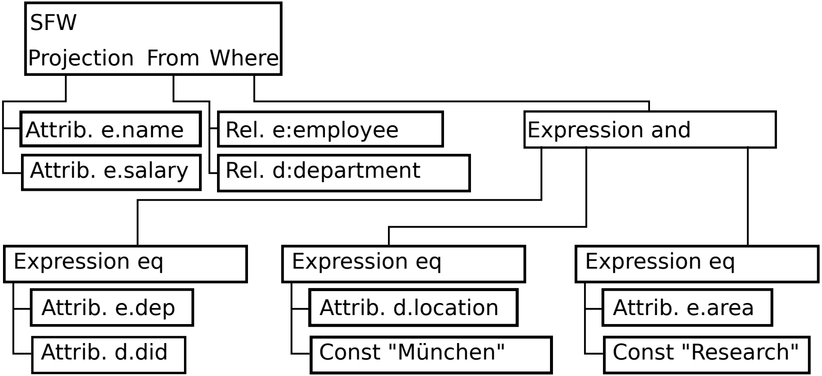
Resolves all variable binding, infers the types and checks semantics



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

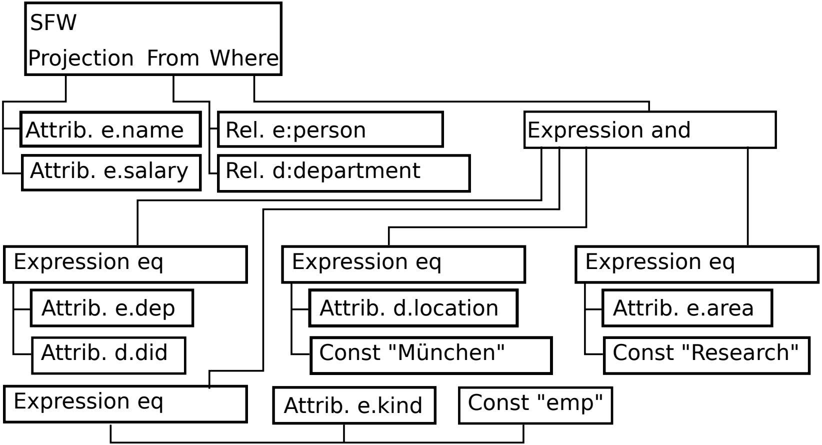
Processing Example - Normalization

Normalizes the representation, factorizes common expressions, folds constant expressions



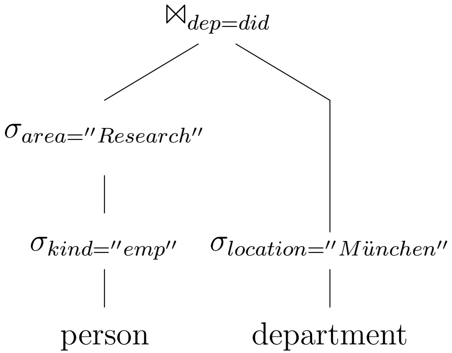
Processing Example - Rewrite I

resolves views, unnests nested expressions, expensive optimizations



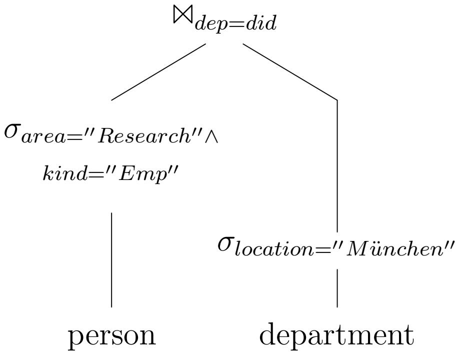
Processing Example - Plan Generation

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

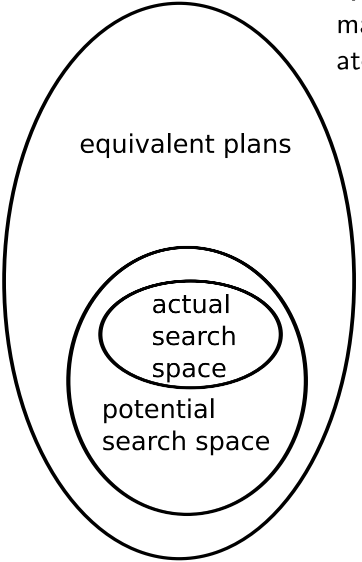
When given an algebraic expression:

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

Problems:

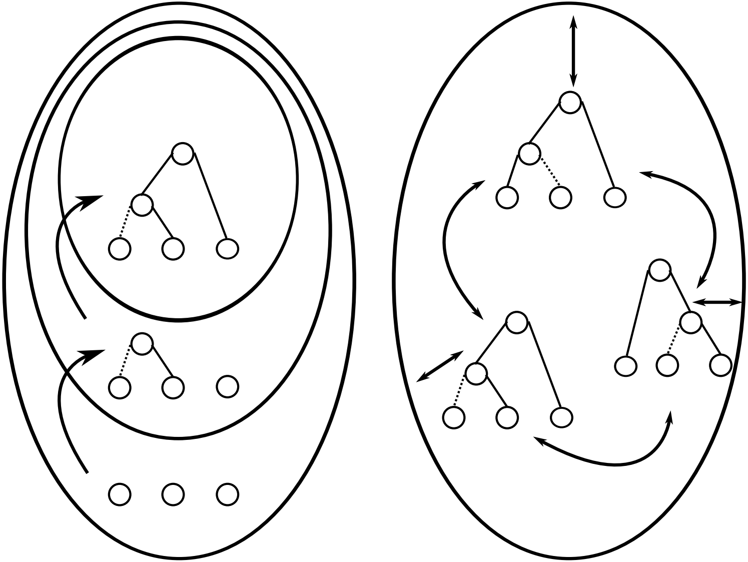
- the set of possible expressions is huge
- testing for equivalence is difficult/impossible in general
- the query is given in a calculus and not an algebra (this is also an advantage, though)
- even "simpler" optimization problems (e.g. join ordering) are typically NP hard in general

Search Space



Query optimizers only search the "optimal" solution within the limited space created by known 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 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, $\{x \mid x \in e_1 \wedge x \notin e_2\}$
$\rho_{a \rightarrow b}(e)$	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, $\{x \circ y_{\vert \mathcal{A}(e_2) \setminus \mathcal{A}(e_1)} \vert x \in e_1 \land y \in e_2 \land x =_{\vert \mathcal{A}(e_1) \cap \mathcal{A}(e_2)} y\}$
$e_1 \div e_2$	division, $\{x_{\vert \mathcal{A}(e_1) \setminus \mathcal{A}(e_2)} \vert x \in e_1 \land \forall y \in e_2 \exists z \in e_1 : y =_{\vert \mathcal{A}(e_2)} z \land x =_{\vert \mathcal{A}(e_1) \setminus \mathcal{A}(e_2)} z\}$
$e_1 \ltimes_p e_2$	semi-join, $\{x \vert x \in e_1 \land \exists y \in e_2 : p(x \circ y)\}$
$e_1 \triangleright_p e_2$	anti-join, $\{x \vert x \in e_1 \land \nexists y \in e_2 : p(x \circ y)\}$
$e_1 ⟕_p e_2$	outer-join, $(e_1 \bowtie_p e_2) \cup \{x \circ \circ_{a \in \mathcal{A}(e_2)}(a : null) \vert x \in (e_1 \triangleright_p e_2)\}$
$e_1 ⟗_p e_2$	full outer-join, $(e_1 ⟕_p e_2) \cup (e_2 ⟕_p e_1)$

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
  $\Gamma_{A;a:f}(e) = \{x \circ (a : f(y)) | x \in \Pi_A(e) \land y = \{z | z \in e \land \forall a \in A : x.a = z.a\}\}$
- dependent join (djoin). Requires $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$
  $e_1 \ltimes_p e_2 = \{x \circ y | x \in e_1 \land y \in e_2(x) \land p(x \circ y)\}$

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

Requirements/Effects:

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

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:

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

Equivalences

Equivalences for selection and projection:

$$
\begin{array}{rcll}
\sigma_{p_1 \wedge p_2}(e) & \equiv & \sigma_{p_1}(\sigma_{p_2}(e)) & (1)\\
\sigma_{p_1}(\sigma_{p_2}(e)) & \equiv & \sigma_{p_2}(\sigma_{p_1}(e)) & (2)\\
\Pi_{A_1}(\Pi_{A_2}(e)) & \equiv & \Pi_{A_1}(e) & (3)\\
 & & \text{if } A_1 \subseteq A_2 & \\
\sigma_p(\Pi_A(e)) & \equiv & \Pi_A(\sigma_p(e)) & (4)\\
 & & \text{if } \mathcal{F}(p) \subseteq A & \\
\sigma_p(e_1 \cup e_2) & \equiv & \sigma_p(e_1) \cup \sigma_p(e_2) & (5)\\
\sigma_p(e_1 \cap e_2) & \equiv & \sigma_p(e_1) \cap \sigma_p(e_2) & (6)\\
\sigma_p(e_1 \setminus e_2) & \equiv & \sigma_p(e_1) \setminus \sigma_p(e_2) & (7)\\
\Pi_A(e_1 \cup e_2) & \equiv & \Pi_A(e_1) \cup \Pi_A(e_2) & (8)
\end{array}
$$

Equivalences

Equivalences for joins:

$$
\begin{aligned}
e_1 \times e_2 &\equiv e_2 \times e_1 && (9)\\
e_1 \Join_p e_2 &\equiv e_2 \Join_p e_1 && (10)\\
(e_1 \times e_2) \times e_3 &\equiv e_1 \times (e_2 \times e_3) && (11)\\
(e_1 \Join_{p_1} e_2) \Join_{p_2} e_3 &\equiv e_1 \Join_{p_1} (e_2 \Join_{p_2} e_3) && (12)\\
\sigma_p(e_1 \times e_2) &\equiv e_1 \Join_p e_2 && (13)\\
\sigma_p(e_1 \times e_2) &\equiv \sigma_p(e_1) \times e_2 && (14)\\
& \quad \text{if } \mathcal{F}(p) \subseteq \mathcal{A}(e_1) &&\\
\sigma_{p_1}(e_1 \Join_{p_2} e_2) &\equiv \sigma_{p_1}(e_1) \Join_{p_2} e_2 && (15)\\
& \quad \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) &&\\
\Pi_A(e_1 \times e_2) &\equiv \Pi_{A_1}(e_1) \times \Pi_{A_2}(e_2) && (16)\\
& \quad \text{if } A = A_1 \cup A_2, A_1 \subseteq \mathcal{A}(e_1), A_2 \subseteq \mathcal{A}(e_2) &&
\end{aligned}
$$