

Database Languages and their Compilers

Prof. Dr. Torsten Grust

Database Systems Research Group
U Tübingen

Winter 2010



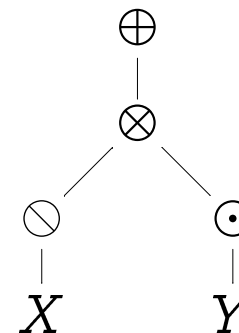
Copyright © 1995 United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited

4 Query Normalization

- ▶ Finally, we have reached a point where our “understanding” (read: syntactic and semantic analysis) of the input query is complete enough to **plan its execution** on the underlying DBMS.
- ▶ What we need now is a mapping from MCC to the primitives of the query engine.
 - These primitives (**operators**) may usually be connected in a tree-like fashion to form a **query execution plan**:

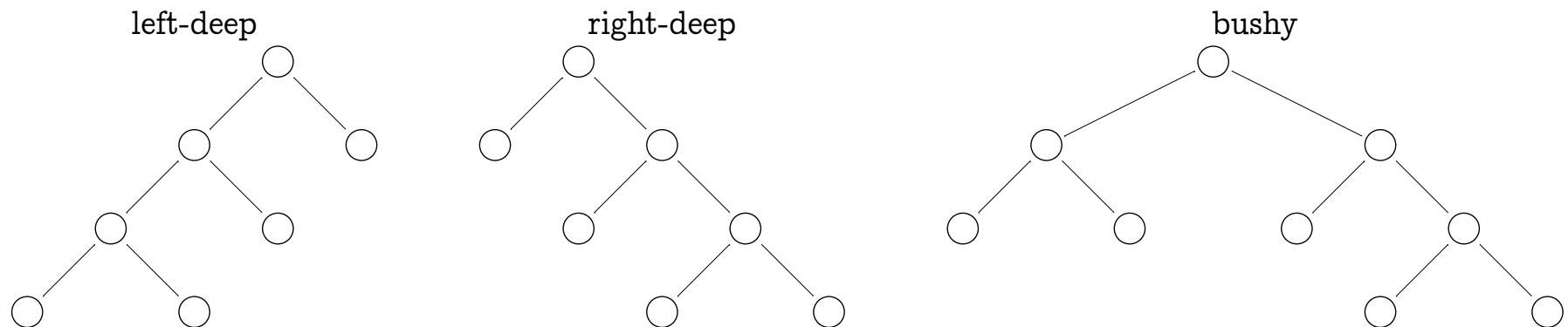
$[e \mid qs]^M$

\dashrightarrow
?



► We will, of course, aim to assemble execution plans which

- ① contain the **minimal number** of necessary operators ○;
 - this will reduce the need to allocate and manage intermediate query results, thus making the overall I/O costs cheaper;
- ② use those operators whose **internal workings are simple** and thus are efficiently executable;
 - this will help to reduce CPU as well as I/O costs (consider the need for temporary storage during operator execution, e.g., sorting).
- ③ adhere to specific **tree shapes**:



(but we will postpone a discussion of this right now.)

- ▶ For a first approach to find such a mapping from MCC to query plans, we can fall back on the *fold*-based MCC semantics (see Chapter 2):
 - We need a single type of \circ -operator implemented in our query engine, namely *fold*;
 - equations (1) through (3) of Section 2.3 provide the mapping from MCC to execution plans;

4.1 Comprehension Normalization

- ▶ SQL \dashrightarrow MCC \dashrightarrow *fold* provides a complete query processor framework for SQL.
 - Unfortunately, *fold*'s internal workings are not simple in the sense of our previous discussion.
 - Each instance of *fold* we can get rid of in a query plan is worth to search for.

► MCC provides a number of hooks to **simplify its expressions** while their meaning is preserved:

■ From a bird's eye point of view, the overall structure of a comprehension computing an SQL SUM-aggregate query is (see cases (g) and (h) of mapping \mathbb{T}):

$$[[G \mid G \leftarrow [e \mid X_1 \leftarrow R]^{bag}]^{sum} \mid]^{bag} .$$

■ **Claim:** Irrespective of e , this computes the same value as

$$[[e \mid X_1 \leftarrow R]^{sum} \mid]^{bag} .$$

 **The second comprehension saves a *fold*.**

A comparison of the corresponding *fold* programs for both comprehensions justifies this **equivalence-preserving simplification step**.

- ▶ This simplification step is an instantiation of a far more general **rewriting** that MCC allows us to do:

- **Comprehension Unnesting** ($M \neq M'$ in general):

$$[e \mid qs, v \leftarrow [e' \mid qs']^{M'}, qs'']^M = [e \mid qs, qs', v \equiv e', qs'']^M$$

The comprehension on the right hand side uses one generator (\leftarrow) less.

- The unnesting rule implements the simplification procedure of the previous slide:

$$\begin{aligned} & [[G \mid \underbrace{G}_{qs=\epsilon} \leftarrow [e \mid X_1 \leftarrow R]^{bag} \underbrace{qs''=\epsilon}]]^{sum} \mid]^{bag} \\ & \quad =_{\text{unnest}} [[G \mid X_1 \leftarrow R, G \equiv e]^{sum} \mid]^{bag} \\ & \quad =_{\text{remove } \equiv} [[e \mid X_1 \leftarrow R]^{sum} \mid]^{bag} \end{aligned}$$

- **Nested SQL queries** of the form shown here may lead to inefficient query execution plans:

Let p, p' denote two SQL predicates:

$$q = \begin{array}{l} \text{SELECT DISTINCT } x \\ \text{FROM } X \text{ AS } x \\ \text{WHERE EXISTS } z \text{ IN } \underbrace{\left(\begin{array}{l} \text{SELECT } y \\ \text{FROM } Y \text{ AS } y \\ \text{WHERE } p(x, y) \end{array} \right)}_{\text{subquery}} : p'(x, z) \end{array}$$

- Note that the nested subquery is **correlated**:

- $p(x, y)$ depends on tuple variable x bound in the outer query;
- consequently, we have to re-evaluate the subquery for every binding of x ;
- it looks like we have to read Y $|X|$ times.

► We can improve the situation with the help of a new MCC rewriting rule:

■ **Unnesting of Existential Quantification:**

$$[e \mid qs, [e' \mid qs']^{some}, qs'']^{set} = [e \mid qs, qs', e', qs'']^{set}$$

■ **N.B.:** Unnesting of existential quantifiers is valid only inside a *set* comprehension.

■ This is applicable to q after we have switched to MCC with the help of \mathbb{T} :

$\mathbb{T}(q)$

$$\begin{aligned} & \stackrel{\mathbb{T}}{=} [(x : x) \mid x \leftarrow X, [p'(x, z) \mid z \leftarrow [(y : y) \mid y \leftarrow Y, p(x, y)]^{bag}]^{some}]^{set} \\ & \stackrel{\text{unnest}}{=} [(x : x) \mid x \leftarrow X, [p'(x, z) \mid y \leftarrow Y, p(x, y), z \equiv (y : y)]^{some}]^{set} \\ & \stackrel{\text{remove } \equiv}{=} [(x : x) \mid x \leftarrow X, [p'(x, (y : y)) \mid y \leftarrow Y, p(x, y)]^{some}]^{set} \\ & \stackrel{\text{unnest } some}{=} [(x : x) \mid x \leftarrow X, y \leftarrow Y, p'(x, (y : y)), p(x, y)]^{set} \end{aligned}$$

- ▶ If we apply “ \top^{-1} ” to the rewriting results, we now see that q is nothing else but a join between X and Y :

```
SELECT DISTINCT   $x$ 
      FROM       $X$  AS  $x$ ,  $Y$  AS  $y$ 
     WHERE       $p'(x, (y : y))$ 
      AND        $p(x, y)$  .
```

- DBMS query engines can cope with queries of this type much more efficiently (various join algorithms are applicable now).

 **Correlation matters.**

If we replace $p(x, y)$ by $p(y)$ in the original query q , can you take advantage of this modification?

How about replacing $p(x, y)$ by $p(x)$?

- In company with two additional laws, the repeated and exhaustive application of the unnesting rules rewrites a comprehension into its **normal form**:



Comprehension Normalization:

$$[e \mid qs, v \leftarrow [e' \mid qs']^{M'}, qs'']^M = [e \mid qs, qs', v \equiv e', qs'']^M \quad (4)$$

$$[e \mid qs, [e' \mid qs']^{some}, qs'']^{set} = [e \mid qs, qs', e', qs'']^{set} \quad (5)$$

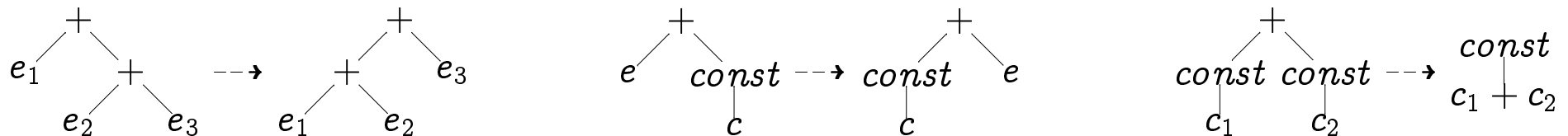
$$[e \mid qs, v \leftarrow lift^{M'}(e'), qs']^M = [e \mid qs, v \equiv e', qs']^M \quad (6)$$

$$[e \mid qs, v \leftarrow z^{M'}, qs']^M = z^M \quad (7)$$

4.2 Further Rewriting Laws

- ▶ The expressions of MCC are subject to a large number of additional equivalence-preserving rewritings and simplifications.
- ▶ If some of these rules appear to be too obvious, recall that
 - we are trying to **implement a completely unguided system** so that we have to declare even the most simple facts;
 - rewriting opportunities might occur due to the **application of (a number of) other rewriting laws**.

4.2.1 Constant Folding



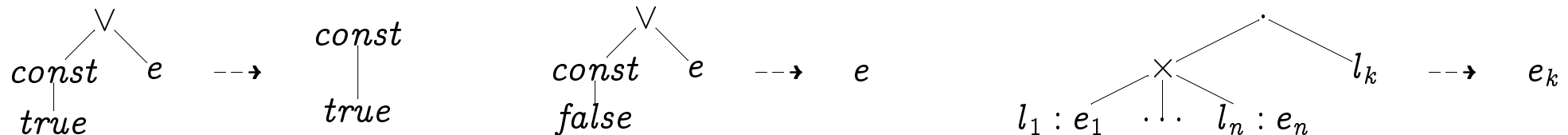
► Note that the rewriting laws are **directed** (\dashrightarrow instead of $=$) to avoid infinite rewriting loops.

■ The first two rules aim to construct a **normal form** of arithmetic expressions in which numeric constants appear left-most, e.g.:

$$((((10 + 30) + 0) + 2) + x) + y) + z ,$$

three applications of the last law finally produce $((42 + x) + y) + z$.

■ Some more examples:



■ These laws, in turn, may then make the following applicable:

$$[e \mid qs, \text{true}, qs']^M \dashrightarrow [e \mid qs, qs']^M \qquad [e \mid qs, \text{false}, qs']^M \dashrightarrow z^M$$

4.2.2 Reordering the Qualifier List

- ▶ For a comprehension $[e \mid qs]^M$, the **order** of qualifiers in qs obviously makes a difference:
 - **moving a predicate** towards the beginning of qs allows to **evaluate filters early**;
 - **moving generators** inside qs may lead to **interesting join orders** (later).

① Exchanging predicates:

To see that two adjacent predicates p, q in a qualifier list may be transposed, note that

$$[e \mid p, q]^M = [e \mid p \wedge q]^M .$$

See Section 2.3 (rule (3)) and use the equivalence

$$\begin{array}{l} \text{IF } p \text{ THEN} \\ \quad | \text{ IF } q \\ \quad | \quad \text{THEN} \\ \quad | \quad | e \\ \quad | \quad \text{ELSE} \\ \quad | \quad \quad \perp e' \\ \text{ELSE} \\ \quad \perp e' \end{array} \equiv \begin{array}{l} \text{IF } p \wedge q \\ \text{THEN} \\ \quad | e \\ \text{ELSE} \\ \quad \perp e' \end{array} .$$

Finally:

$$[e \mid p, q]^M = [e \mid p \wedge q]^M \stackrel{\wedge \text{ commutative}}{=} [e \mid q \wedge p]^M = [e \mid q, p]^M .$$

② Moving generators:

Generators introduce variables, so we have to be careful not to move expressions out of scope and to avoid name capture.

(a) Transposing a generator and a predicate (p may not reference v):

$$\begin{array}{ccc} [e \mid v \leftarrow e', p]^M & = & [e \mid p, v \leftarrow e']^M \\ \begin{array}{c} \uparrow \\ E \end{array} & \begin{array}{c} \uparrow \\ E + \{v \mapsto \tau\} \end{array} & \begin{array}{c} \uparrow \\ E \end{array} \end{array} .$$

(b) Transposing two generators (e' may not reference v' , e'' may not reference v , $M \in \{set, bag\}$):

$$\begin{array}{ccc} [e \mid v \leftarrow e', v' \leftarrow e'']^M & = & [e \mid v' \leftarrow e'', v \leftarrow e']^M \\ \begin{array}{c} \uparrow \\ E \end{array} & \begin{array}{c} \uparrow \\ E + \{v \mapsto \tau\} \end{array} & \begin{array}{c} \uparrow \\ E \end{array} & \begin{array}{c} \uparrow \\ E + \{v' \mapsto \tau'\} \end{array} \end{array} .$$

► **N.B.:** These laws are specified for pairs of qualifiers but they can be easily lifted onto sequences of qualifiers of length > 2 .

Contents

0	Introduction to Query Compilation	1
0.1	Welcome!	1
0.2	Administrativa	3
0.3	Some Remarks on these Slides	4
0.4	Relational Databases and SQL	6
0.5	A Guided Tour through an SQL Query Processor	14
0.5.1	Character Streams and Tokens	15
0.5.2	Identify Valid SQL Syntax	16
0.5.3	Resolve the Meaning of Variables and Identifiers	17
0.5.4	Check Types and Schemas	18

0.5.5	How Could Query Evaluation Look Like?	19
0.5.6	One Query, Many Programs	20
0.5.7	Query Operators	21
0.5.8	Query Operator Trees and Rewriting	22
0.5.9	The Cheaper, the Better: Query Cost Models	23
0.5.10	Executing Operator Trees on a Machine	24
0.6	Recommended Reading	27
1	Query Parsing	28
1.1	The Scanner (Lexer)	28
1.1.1	Regular Expressions	33
1.1.2	Regular Expressions for SQL	35
1.1.3	<u>lex – A Scanner Generator</u>	<u>39</u>

1.2	Syntax Analysis (Parsing)	42
1.2.1	Context-Free Grammars	44
1.2.2	Derivations and Parse Trees	46
1.2.3	Ambiguous Grammars	50
1.2.4	Predictive Parsing and Recursive Descent	54
1.2.5	yacc – A Parser Generator	63
1.3	Recommended Reading	67
2	A Query Calculus for SQL	68
2.1	SQL’s Nested Loops Semantics	69
2.2	<i>fold</i>	71
2.3	The Monoid Comprehension Calculus	75
2.4	Mapping SQL to MCC	81

2.4.1	Applying \mathbb{T} to Translate SQL Queries	91
2.5	Recommended Reading	95
3	Variable Scopes and Type Inference for SQL	96
3.1	Variables in SQL	97
3.1.1	Variable Environments	99
3.2	The Type of an SQL Query	102
3.2.1	Type Inference	103
3.3	Semantic Attributes	105
4	Query Normalization	115
4.1	Comprehension Normalization	117
4.2	Further Rewriting Laws	124
4.2.1	Constant Folding	124

4.2.2 Reordering the Qualifier List 126