Database Languages and their Compilers

Prof. Dr. Torsten Grust

Database Systems Research Group
U Tübingen

Winter 2010

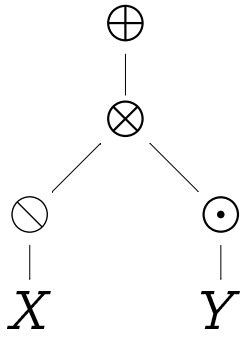
4 Query Normalization

- 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**:

$$\left[\, e \mid qs \,\right]^{M} \quad \overset{?}{\dashrightarrow}$$

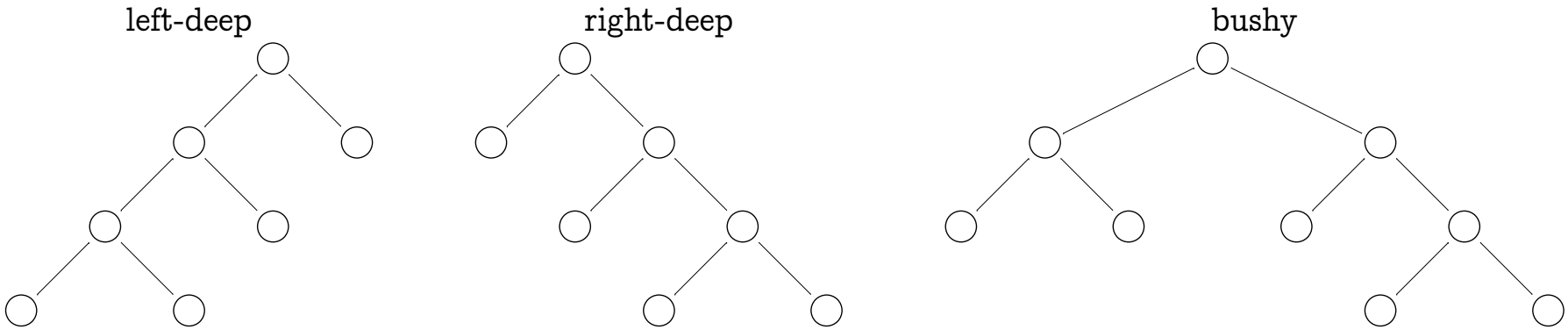


- We will, of course, aim to assemble execution plans which

  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 ○-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 ⇢ MCC ⇢ *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 $\mathbb{T}$):

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

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

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

> ✎ **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):

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

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

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

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

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

  Let p, p' denote two SQL predicates:

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

- 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:**

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

  - **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 $\mathbb{T}$:

$\mathbb{T}(q)$

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

- If we apply "$\mathbb{T}^{-1}$" to the rewriting results, we now see that q is nothing else but a join between X and Y:

$$\begin{array}{rl} \texttt{SELECT DISTINCT} & x \\ \texttt{FROM} & X\ \texttt{AS}\ x, Y\ \texttt{AS}\ y \\ \texttt{WHERE} & p'(x, (y : y)) \\ \texttt{AND} & p(x, y)\ . \end{array}$$

  - 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:

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

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

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

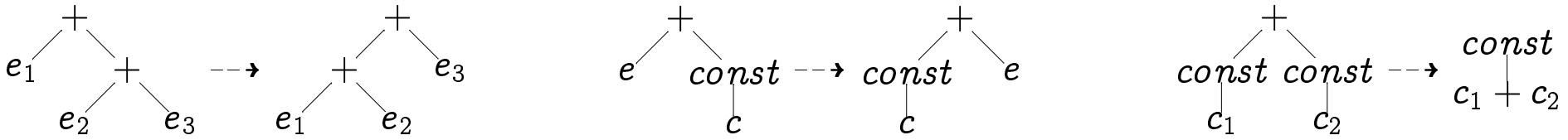
$$[\, e \mid qs, v \leftarrow z^{M'}, qs' \,]^{M} = z^{M} \quad (7)$$

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**.

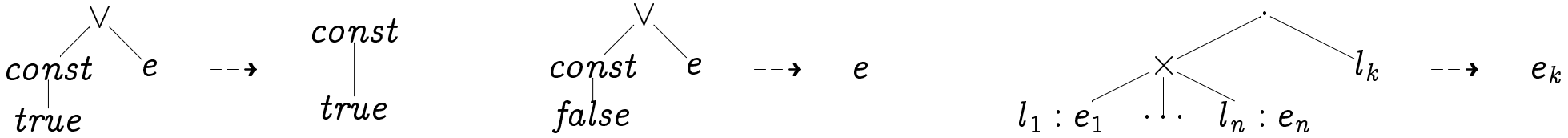
4.2.1 Constant Folding



- 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:

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

4.2.2 Reordering the Qualifier List

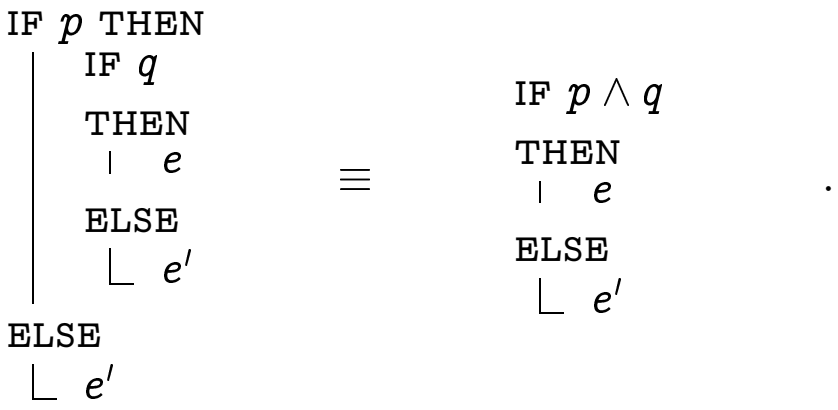
- For a comprehension $[\, e \mid qs \,]^M$, the **order** of qualifiers in qs obviously makes a difference:
  - **moving a predicate** towards the beginning of qs allows to **evaluate filters early**;
  - **moving generators** inside qs may lead to **interesting join orders** (later).

① **Exchanging predicates:**

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

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

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



Finally:

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

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):

$$\underset{\substack{\uparrow \\ E}}{[\, e \mid} \; \underset{\substack{\uparrow \\ E}}{v \leftarrow e'}, \; \underset{\substack{\uparrow \\ E+\{v \mapsto \tau\}}}{p} \,]^M \quad = \quad [\, e \mid \; \underset{\substack{\uparrow \\ E}}{p}, v \leftarrow e' \,]^M \;\; .$$

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

$$\underset{\substack{\uparrow \\ E}}{[\, e \mid} \; \underset{\substack{\uparrow \\ E}}{v \leftarrow e'}, \; \underset{\substack{\uparrow \\ E+\{v \mapsto \tau\}}}{v' \leftarrow e''} \,]^M \quad = \quad [\, e \mid \; \underset{\substack{\uparrow \\ E}}{v' \leftarrow e''}, \; \underset{\substack{\uparrow \\ E+\{v' \mapsto \tau'\}}}{v \leftarrow e'} \,]^M \;\; .$$

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

Contents