Heraclitus: A Delta-Based DB Programming Language

Zachary G. Ives
University of Pennsylvania

April 16, 2003

Many Non-traditional Models for Representing Data

- We know the relational (OO, semistructured, …) models well
  - Describes world "as we know it"
  - Assumption generally of "closed world"
- But there are also other ideas:
  - Incomplete databases: some values might be missing but we may know something about them
  - Constraint databases: we may know certain relationships between items
  - Today: "what if" databases

Heraclitus

- Named after a Greek philosopher
  - “World is an ongoing process governed by a law of change”
- Idea: let’s explore the affects of possibly applying changes to the database
  - Possible changes are described as “deltas”
- Heraclitus[Alg,C] is an implementation using a “database programming language” (DBPL)

Background: DBPLs

- Strong “impedance mismatch” between SQL and programming languages
  - Painful to use ODBC (or even JDBC) to manipulate data – need to map between objects, open cursors, execute SQL and get rowsets, etc.
- OO databases tried to abolish most of these
  - The database holds objects; the programming language (e.g., C++) is OO
- DBPLs go even further: seamlessly integrate database types into the language

Heraclitus Data Model

- Relations (in the usual sense)
  - Semantics are set-oriented only – no bags
- Deltas – additions or deletions to relations
  - Atoms are insertions or deletions
  - Restriction: delta shouldn't do useless work
  - Special delta, analogous to null/undefined: *fail*

Operations

- Need to be able to apply deltas (speculatively) to relations

  R' = apply(R, Δ)

  Δ can be divided into Δ^+ (adds) and Δ^- (removes)

- Want to be able to merge deltas in meaningful ways
  - For disjoint atoms, no problem
  - But what to do with conflicting atoms:
    - Δ_1 = {+Suppliers(Bob, shoe)}, Δ_2 = {-Suppliers(Bob, shoe)}

First delta combinator: “smash”

- $\Delta_1 ! \Delta_2$ favors Δ_2 for any conflict
  - $(\Delta_1 ! \Delta_2)^+ = \Delta_2^+ \cup (\Delta_1^+ - \Delta_2^-)$
  - $(\Delta_1 ! \Delta_2)^- = \Delta_2^- \cup (\Delta_1^- - \Delta_2^+)$

- Think about query optimization: what do we depend on in terms of our operators?
- What are the implications of smash on query optimization?

Second delta combinator : “merge”

- Δ_1 & Δ_2 fails on any conflict

- Does this have better properties for optimization?

- What about *weak-merge*, which simply deletes conflicting atoms?

Third delta combinator: “compose”

- Δ_1 # Δ_2 applies Δ_1, then applies Δ_2 to the result

- How does this differ from smash?

Smash vs. Compose

Updates:

δ_1: +Ord("light", 400, "Cat Paw", "9/18/93")

δ_2: Ord(*,400, *, *) -> Ord(*,450, *, *)

Applied to:

Ord("frame", 400, "Trek", "8/18/93")

What it Looks Like to the Programmer

- Heraclitus[Alg,C] adds to C:
  - Datatypes for relations
  - Definitions of schemas
  - Specifications of deltas
  - Operators
- Example:

```
relation temp;
Delta D1;
D1 = [del Supp("Campy", "pedals")];
temp = project([part, qty*2], select({foo(sup)>qty}, Ord)) when D1;
```

How Do We Build It?

- Let’s do it!
- What’s new?
  - “when” / “apply” operations
  - “smash” etc.
  - How do we optimize?
  - How do we execute?

Summary

- A very interesting and different way of modeling data
  - Based on deltas (the “reverse” of Monday’s paper)
  - Allows us to query the results of applying speculative changes
  - Sadly, little follow-up work seems to have been done
- Notice that given the specs, we can use our standard class of techniques and build it!
  - Algebraic laws affect optimization
  - Cost model depends on expected # of probes, size, etc.
  - Basic iterate, sort, hash, index techniques useful for execution