On Implementing a Language for Specifying Active Database Execution Models*

Shahram Ghandeharizadeh, Richard Hull,† Dean Jacobs
Jaime Castillo, Martha Escobar-Molano, Shih-Hui Lu, Junhui Luo, Chiu Tsang, Gang Zhou

Computer Science Department
University of Southern California
Los Angeles, California 90089

Abstract

A key issue when incorporating rules into database systems concerns understanding alternative semantics of rule application. The database programming language Heraclitus[Alg,C] is an extension of C that supports the relational algebra and novel constructs related to the specification of these semantics. In particular, the language supports *deltas* as "first-class citizens" - these are values corresponding to database updates, which may or may not be applied. Deltas are useful in representing the effect of rule firings, and for representing virtual database states, as they arise in the specification of several active database systems. Unlike previous work on differential files and hypothetical relations, Heraclitus supports operators for combining deltas, and also alternative implementations that incorporate the impact of deltas into conventional database operators (e.g., join). The framework also appears useful in connection with hypothetical database access, version control, specifying concurrency protocols, and the resolution of update conflicts. This paper describes the design and preliminary implementation of Heraclitus[Alg,C]. Two strategies for providing access to deltas have been implemented, one hash-based and the other sort-based. Initial evaluation of system performance demonstrates the feasibility of the language.

*This work was supported in part by the NSF under grants IRI-9110522, IRI-9107055, INT-8817874, NYI award IRI-9258362, and the USC FRIF under grant 22-1509-9440.

†This author's current address: Computer Science Department, University of Colorado, Boulder, CO 80309.

**Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993**

1 Introduction

"Active" databases generally support the automatic triggering of updates as a response to user-requested or system-generated updates. Many active database systems, e.g., [CCCR+90, Coh86, MD89, Han89, dMS88, SIG89, SJGP90, WF90, ZH90], use a paradigm of *rules* to generate these automatic updates, in a manner reminiscent of expert systems. Active databases have been shown useful for constraint maintenance [Mor83, CW90, HJ91a], incremental update of materialized views [CW91], query rewriting [SJGP90], database security [SJGP90]; and hold the promise of providing a new family of solutions to the view and derived data update problem [CHM92] and issues in heterogeneous databases [CW92]. Active database technology will also play an important role in the development of "mediators" [Wie92] for supporting database interoperation.

As discussed in Section 2 (see also [HJ91a, HW92, Sto92]), each of the active database systems described in the literature uses a different semantics or "execution model" for rule application. The variety of alternatives found in active database systems highlights the fact that the "knowledge" represented in them stems from two distinct components: the rule base and the execution model [Abi88]. It appears that different execution models will sometimes be appropriate even within a single database, and that a fixed collection of choices is unlikely to suffice. There is a need for high-level constructs that permit database designers and programmers to specify and implement system modules using customized execution models.

The Heraclitus project [HJ91b, JH91, GHJ92] is focused on the development of database programming language constructs and techniques that can be used to specify and implement alternative, interchangeable execution models for active database systems. Our current focus is to provide language constructs that support (a) the use of multiple virtual states in rule conditions and (b) a wide variety of semantics for applying rules and combining their effects. We are currently working with the pure relational model (no duplicates or tuple-ids). Extensions to incorporate tuple-ids and to generalize to the object-oriented data model are a part of our future research direction.

The basic novelty in the Heraclitus framework is to elevate *deltas*, i.e., values corresponding to database updates, which may or may not be applied, to be "first-class citizens" in database programming languages. Operators are provided for explicitly constructing, accessing and combining deltas. Of particular importance is the **when** operator that permits hypothetical expression evaluation: expression E **when** δ evaluates to the value that E *would* have if the value of δ *were* applied to the current state. This allows deltas to be used to represent virtual states, and also supports hypothetical database access.

We have implemented Heraclitus[Alg,C], a database programming language (DBPL) that extends C to incorporate the relational algebra and deltas and their operators. The implementation has two primary components, a pre-processor and HERALD (HEraclitus Relational ALgebra with Deltas), a library of functions supporting relational and delta operators. Of particular interest is the support of "hypothetical" relational operators, which correspond to the traditional relational operators (e.g., select, join) evaluated under a **when**. HERALD was initially implemented [GHJ92] on the Wisconsin Storage System (WiSS) [CDKK85], and has now been ported to the Exodus system [CDRS86]. HERALD currently supports two strategies for incorporating the effect of deltas on the relational operators, one hash-based and the other sort-based.

This paper describes the design and preliminary implementation of Heraclitus[Alg,C]. Section 2 discusses the conceptual underpinnings of deltas and their use in specifying active database execution models and other database applications. Section 3 introduces Heraclitus[Alg,C], presenting both algebraic operators and language constructs. Section 4 describes the current implementation of the language, along with analysis of the expected running times for the various algebraic operators. Section 5 presents an overview of initial benchmarking activities performed to experimentally verify the analysis of Section 4. Brief conclusions are offered in Section 6

2 Deltas, Virtual States, and Active Database Execution Models

This section lays a conceptual framework for understanding much of the current research in active databases. In particular, we show how access to both deltas and virtual states are useful in the context of active databases, and illustrate how the Heraclitus paradigm can be used to provide this access. Some of this material also appears in [HJ91b], and is included here to make the current paper more self-contained. At the end of the section we briefly sketch other database applications where this paradigm may be useful, and compare our deltas with related work on hypothetical relations and differential files.

2.1 Active databases

A wide range of active database systems have been proposed in the literature. The most crucial differences between their execution models stem from choices concerning (a) how and when rules should be fired, (b) the expressive capabilities of the rules, and (c) how the effects of rule firings should be combined. With regards to (a), three approaches have been proposed: (i) ***transaction boundary*** rule firing, which occurs only at the end of the user transaction (e.g., Starburst, RDL1, LOGRES, AP5); (ii) ***interleaved*** rule firing, where rule application is interleaved with the atomic commands of a user transaction (e.g., POSTGRES [SJGP90], among others [Han89, KDM88, MP90, MD89]); and (iii) *concurrent* rule firing (e.g., [MD89, BM91]), in which rules may spawn concurrent processes in a recursive fashion. The Heraclitus paradigm can be used to specify many of these design choices; in this subsection we focus on transaction boundary rule firing, and briefly discuss interleaved rule firing.

Under transaction boundary rule firing, rule application constructs a sequence of "virtual states"

$$S_{orig},\ S_{prop},\ S_2,\ S_3,\ \ldots,\ S_{curr}$$

of the database, where S_{orig} is the "original" state and S_{prop} is the result of applying to S_{orig} the set of user-proposed updates collected during the transaction. The subsequent virtual states result from a sequence of rule firings according to the execution model. S_{curr} denotes the "current" virtual state that is being considered by the execution model. Execution terminates either when the execution model reaches a fixpoint, in which case the final virtual state replaces S_{orig}, or aborts the transaction. Prominent systems following this paradigm include the Starburst Rule System [WF90, CW90], RDL1 [dMS88], LOGRES [CCCR+90] and AP5 [Coh86, ZH90], and also expert systems such as OPS5 [BFKM85]. (Other paradigms shall be considered below.)

As a simple example, consider a relational database for inventory control in manufacturing. Figure 1 shows two relations used by a hypothetical bicycle manufacturer. The **Suppliers** relation holds suppliers and the parts they supply, and the **Orders** relation shows currently unfilled orders for parts. Other relations, not shown here, might hold information

Supplier	Part
Trek	frame
Campy	brakes
Campy	pedals

Suppliers

Part	Quantity	Supplier	Expected
frame	400	Trek	8/31/93
brakes	150	Campy	9/1/93

Orders

Figure 1: Relations for Inventory Control Example

about the parts usage of different bicycle models, and the expected demand for these parts based on the production schedule of the company.

Consider now the referential integrity constraint stating that if there is an order for part p from supplier s, then the pair (s,p) should be in relation **Suppliers**. A possible rule for enforcing this might be written as

R1 : if $\texttt{Orders}(part, qty, supp, exp)$ and
  not $\texttt{Suppliers}(supp, part)$
 then $-\texttt{Orders}(part, qty, supp, exp)$

In the pidgen syntax used for this rule we follow the style of many active database systems. In particular, (a) the "if" part, or *condition*, is a boolean expression -- the rule can "fire" only if this expression evaluates to true; (b) the "then" part, or *action* is an imperative command that executes when the rule fires; and (c) it is implicit which virtual state(s) are being considered by the conditions and actions. In typical active database systems, if at some point in the application of rules the state S_{curr} satisfies the condition of R1 for some assignment of variables, then the action may be fired, depending on the presence of other rules whose condition is true. We say that rule R1 uses a "one-state" logic, because the rule condition examines a single state, namely the "current" one. RDL1, LOGRES, and most expert systems (e.g., OPS5 [BFKM85]) support only a one-state logic.

In the context of databases, a problem with rule R1 is that the appropriate response to a constraint violation may depend on how the violation arose. Rule R2 below deletes all violating orders if a pair is deleted from the **Suppliers** relation, but if the violation is the result of an update to **Orders**, then R3 undoes that individual update and transmits a warning.

R2 : if $-\texttt{Suppliers}(supp, part)$
 then $-\texttt{Orders}(part, *, supp, *)$

R3 : if $+\texttt{Orders}(part, qty, supp, exp)$ and
  not $\texttt{Suppliers}(supp, part)$
 then $-\texttt{Orders}(part, qty, supp, exp)$ and
  $\texttt{send\_warning}(part, qty, supp, exp)$

The signed atoms in the conditions of these rules refer to proposed updates, rather than any database state. The action of R2 uses "wildcards" (denoted '$*$'); these match any value.

In essence, the conditions of rules R2 and R3 make explicit reference to the delta between two virtual states. Of course, some design choice needs to be made about which pair of virtual states should be considered. The AP5 system focuses on the delta between S_{orig} and S_{curr}:

$$\underbrace{S_{orig},\ S_{prop},\ S_2,\ S_3,\ \ldots,\ S_{curr}}_{\Delta}$$

Assuming this semantics for a moment, note that a one-state execution model cannot simulate the effect of rules R2 and R3 without using "scratch paper relations" that essentially duplicate the contents of S_{orig}. Another natural semantics for rule conditions supporting explicit access to a delta would be to use the delta between S_{prop} and S_{curr}. The Starburst Rule System is even more intricate: it uses the delta between virtual states S_i and S_{curr}, where i is determined by the rule under consideration and the history of previous firings of that rule.

Consider finally the rule

R4 : if the firing of rules results in
  a 20% drop in orders
 then $\texttt{inventory\_warning}()$

Here we need to consider the change in orders between S_{prop} and S_{curr}:

$$S_{orig},\ \boxed{S_{prop}},\ S_2,\ S_3,\ \ldots,\ \boxed{S_{curr}}$$

While this could be expressed using explicit access to a delta, it is much easier to express it in terms of the virtual states, i.e., to write:

$$\text{R4}' : \quad \text{if} \quad \frac{\texttt{count(Orders)}\ \text{"in } S_{curr}\text{"}}{\texttt{count(Orders)}\ \text{"in } S_{prop}\text{"}} < .8$$

 then $\texttt{inventory\_warning}()$

In current DBPL's there is no mechanism to write expressions such as the condition of R4′, because they do not provide explicit access to virtual states. The Heraclitus paradigm provides this by using deltas and the special **when** operator. As mentioned in the introduction, the expression E **when** δ evaluates an arbitrary side-effect free expression E in the state that *would* arise if the value of δ *were* applied to the existing database state. Evaluation of such an expression does not change the existing database state. One way to express rule R4′ in the Heraclitus paradigm is to construct deltas corresponding to the virtual states S_{prop} and S_{curr} as follows:

$$\underbrace{\underbrace{S_{orig},\ \boxed{S_{prop}}}_{\Delta_{prop}},\ S_2,\ S_3,\ \ldots,\ \boxed{S_{curr}}}_{\Delta_{curr}}$$

Rule R4′ can be expressed within the Heraclitus paradigm as:

$$\text{R4}'' :\ \text{if}\ \frac{\texttt{count(Orders) when } \Delta_{curr}}{\texttt{count(Orders) when } \Delta_{prop}} < .8$$

then `inventory_warning()`

We now describe how the Heraclitus paradigm can specify a large family of execution models that use transaction boundary rule firing. During execution, the database state will remain untouched, and deltas will be constructed to represent the virtual states needed for evaluating rule conditions. (An alternative would be to update the database state with each rule firing, and maintain "negative" deltas that simulate previous virtual states in the sequence.) Rules are represented as functions that have as input zero or more deltas (corresponding either to virtual states or deltas between them), and produce as output a delta corresponding to the effect of the rule firing. The rules might also invoke additional procedures such as `inventory_warning()`. Although not done here, triggers (which are logically a part of the condition, but whose value can typically be determined in a very efficient manner) can also be incorporated into the framework. Rules can be arranged to provide either "tuple-at-a-time" or "set-at-a-time" operation [WF90]. Algebraic operators are provided in Heraclitus for manipulating deltas, so that deltas corresponding to new virtual states can be constructed from previous deltas and rule outputs. Using this approach, the execution models of AP5, RDL1, LOGRES and the Starburst Rule System can be specified within Heraclitus[Alg,C] (see also [HJ91b]). Variations on this theme can be developed. As a simple example, a rule-base can be "stratified", and the execution model can fire each layer to a fixpoint before moving to the next layer. More complex firing patterns subsuming the rule algebra of [IN88] are easily expressed.

Returning now to the full range of design choices for active database execution models, the Heraclitus paradigm can also specify interleaved rule firing. In this case, the user transaction and the rule actions are broken into a sequence of atomic updates, and rules are invoked immediately upon a condition becoming true. There is the possibility of intricate recursive rule firing, and it is hard to associate an intuitive meaning to the sequence of virtual states constructed. As a result, the rule conditions in these systems typically give explicit access to the "old" and "new" values of certain tuples, but not to multiple virtual states. Heraclitus also permits "hybrid" execution models, which combine aspects of both interleaved and transaction boundary rule firing. At present, the primary focus of the Heraclitus project is on sequential processing; incorporation of concurrent rule firing is a subject of future research.

Heraclitus gives broad latitude with regards to dimensions (b) and (c) mentioned above. For this reason, the Heraclitus paradigm, and Heraclitus[Alg,C] in particular, can serve as a flexible platform for specifying a wide variety of execution models for active databases. We expect this to be useful both in developing customized execution models, and in comparing them, both experimentally and analytically.

2.2 Other applications

We now briefly outline a few other applications of the Heraclitus paradigm. We feel that the Heraclitus paradigm will be useful in implementing and understanding a variety of database issues, including (1) hypothetical database access, (2) version control, (3) concurrency protocols, and (4) update conflict resolution. With regards to (1), it is possible within Heraclitus to specify deltas that have meanings such as "Add 2 weeks to the `Expected` value for all orders with quantity > 500" or "Cancel all orders with `Expected` in the month of October". Queries are now easily specified against hypothetical states using arbitrary combinations of these deltas and the `when` operator (see Subsection 3.3). With regards to (2), alternative versions might be represented using deltas. Because Heraclitus provides explicit access to deltas, it can provide both a flexible platform for developing customized version control frameworks, and for experimentally comparing them. Turning to (3), deltas appear especially useful in connection with long transactions. For example, protocols could be developed in which certain short transactions can be executed during the running of a long transaction, and a delta recording the impact of the short transaction could be stored and applied after the long transaction finishes. This kind of "soft commit" could increase concurrent access to databases. Finally, (4) addresses situations in which multiple conflicting updates are presented to a database system. This could arise, for example, in managing a forest fire, where different observers give conflicting information about current status of the fire. One approach to finding a coherent update is to extend active database techniques, so that rule conditions can explicitly access multiple deltas corresponding to the different proposed updates.

2.3 Related techniques

This section concludes with a brief comparison of the Heraclitus paradigm with related techniques.

Differential files [SL76] are a low-level implementation technique that support efficient representation of multiple versions of a database. Unlike differential files, deltas in the Heraclitus framework are manipulated directly by constructs in the user-level programming language. Furthermore, we support a family of operators for explicitly constructing and combining deltas, in addition to those for explicitly and hypothetically accessing them.

A version of hypothetical relations is introduced in [WS83]. While the work there describes carefully crafted implementation strategies for such relations, it cannot easily be extended to provide the full generality of delta usage supported in the Heraclitus framework.

It has been suggested that a reasonable approach to support the basic functionality of the **when** operator would be to augment existing concurrency control mechanisms, using the following steps: (a) evaluate E **when** δ by applying δ it to the database (but don't commit), (b) evaluate E in the context of the new database, and (c) rollback the transaction in order to undo δ. While this rollback technique will be useful in some contexts, it is just one of several feasible implementation strategies that warrant investigation. In the case of complex algebraic expressions involving several not necessarily nested deltas, it may be more efficient to incorporate optimization of **when** into the conventional optimization of the other algebraic operators, rather than relegating it to the orthogonal rollback mechanism. Also, the use of rollbacks to support hypothetical database access may cause unacceptable delays in the concurrency system, complicate the transaction protocols, and degrade the performance of the system.

3 Heraclitus[Alg,C]

This section describes the language Heraclitus[Alg,C] from a user's perspective. The discussion begins with an abstract perspective on deltas, then presents a specific realization for the relational model of deltas and their algebraic operators, and finally describes how this is embedded into the C language.

3.1 The abstract perspective

The foundation of the Heraclitus paradigm is the notion of *delta values*, sometimes called simply *deltas*; these are functions that map database states to database states. Intuitively, a delta can be thought of as a "delayed update", i.e., a command that can be used to update a given database state, but is not necessarily applied. Three operations are fundamental to deltas: applying them to the current database state to obtain a new one; *composition*, and **when**. The **when** operator provides hypothetical expression evaluation: the value of E **when** δ in state DB is the value of expression E evaluated in the state resulting from the application of the value of delta expression δ on DB.

The notion of delta and these basic operators provide a powerful paradigm for supporting a wide variety of database applications, across a wide spectrum of database models. In the first phase of the Heraclitus project we are focusing on the development of a comprehensive realization of this paradigm and its application for the pure relational model; we plan to extend the paradigm to an object-oriented database model in the near future.

Several factors affect the design of a specific realization of the Heraclitus paradigm. Obviously, we expect that all deltas considered are computable. Furthermore, the family of deltas that can created should be closed under composition. Even in this case, there is a tradeoff between the expressive power of the family of deltas incorporated, and the efficiency with which they can be stored, manipulated, and accessed. In Heraclitus[Alg,C] we provide a natural tabular representation for a restricted family of deltas that permits efficient manipulation. Importantly, the family of deltas supported is sufficient to specify a wide variety of active database execution models.

3.2 The algebraic perspective

To understand the family of deltas supported in Heraclitus[Alg,C], we first describe the tabular representation used for them, and the function that each represents.

A *signed atom* is an expression of the form $+ <$*reln-name*$> <$*tuple*$>$ or $- <$*reln-name*$> <$*tuple*$>$; intuitively these correspond to "insertions" and "deletions", respectively. In the context of Heraclitus[Alg,C], a *delta*, is represented as a finite set of signed atoms (referring to relations in the current database schema) which does not include both positive and negative versions of the same atom. An example is:

$$\Delta_1 = \left\{ \begin{array}{l} +\mathit{Suppliers}(\mathit{Shimano},\ \mathit{brakes}), \\ +\mathit{Suppliers}(\mathit{Trek},\ \mathit{frame}), \\ -\mathit{Orders}(\mathit{brakes},\ \mathit{150},\ \mathit{Campy},\ \mathit{9/1/93}), \\ +\mathit{Orders}(\mathit{brakes},\ \mathit{150},\ \mathit{Shimano},\ \mathit{9/6/93}) \end{array} \right\}$$

Supplier	Part
Trek	frame
Campy	brakes
Campy	pedals
Shimano	brakes

Suppliers

Part	Quantity	Supplier	Expected
frame	400	Trek	8/31/93
brakes	150	Shimano	9/6/93

Orders

Figure 2: Result of applying Δ_1

We also include a special delta value *fail*, that corresponds to inconsistency.

For non-*fail* delta Δ, we set

$$\begin{aligned}\Delta^+ &= \{A \mid +A \in \Delta\}\\ \Delta^- &= \{A \mid -A \in \Delta\}\end{aligned}$$

The consistency requirement on deltas states that $\Delta^+ \cap \Delta^- = \emptyset$. Δ represents the function which maps a database state [1] DB to $(DB \cup \Delta^+) - \Delta^-$, which, due to the consistency requirement, is equal to $(DB - \Delta^-+) \cup \Delta^+$. Speaking informally, applying Δ has the affect of adding tuples of Δ preceded by a '+', and deletes tuples preceded by a '−'.

The result of applying Δ_1 to the instance of Figure 1 is shown in Figure 2. Because we are working with the pure relational model, the signed tuple +*Suppliers*(*Trek, frame*) can be viewed as a "no-op" in this context; it has no impact when `apply` is used on the instance of Figure 1. Deletes are "no-ops" if the associated tuple is not present in the underlying instance. A mechanism to express "modifies" is also incorporated; see Subsection 3.3

We call the composition operator for these deltas *smash*, denoted '!'. The smash of two delta values is basically their union, with conflicts resolved in favor of the second argument. For example, given

$$\Delta_2 = \left\{\begin{array}{l} +\mathit{Suppliers}(\mathit{Cat\ Paw,\ light}),\\ -\mathit{Suppliers}(\mathit{Campy,\ pedals}),\\ -\mathit{Orders}(\mathit{brakes,\ 150,\ Shimano,\ 9/6/93}),\\ +\mathit{Orders}(\mathit{brakes,\ 500,\ Shimano,\ 9/20/93}) \end{array}\right\}$$

[1] In this context, we view the database state to be a set of atoms, e.g., { *Suppliers*(*Trek, frame*), *Suppliers*(*Campy, brakes*), …, *Orders*(*frame, 400, Trek, 8/31/93*), …}.

then $\Delta_1 ! \Delta_2$ equals

$$\left\{\begin{array}{l} +\mathit{Suppliers}(\mathit{Shimano,\ brakes}),\\ +\mathit{Suppliers}(\mathit{Trek,\ frame}),\\ +\mathit{Suppliers}(\mathit{Cat\ Paw,\ light}),\\ -\mathit{Suppliers}(\mathit{Campy,\ pedals}),\\ -\mathit{Orders}(\mathit{brakes,\ 150,\ Campy,\ 9/1/93}),\\ -\mathit{Orders}(\mathit{brakes,\ 150,\ Shimano,\ 9/6/93}),\\ +\mathit{Orders}(\mathit{brakes,\ 500,\ Shimano,\ 9/20/93}) \end{array}\right\}$$

Formally, for non-*fail* Δ_1 and Δ_2 their smash is defined by

$$\begin{aligned}(\Delta_1 ! \Delta_2)^+ &= \Delta_2^+ \cup (\Delta_1^+ - \Delta_2^-)\\ (\Delta_1 ! \Delta_2)^- &= \Delta_2^- \cup (\Delta_1^- - \Delta_2^+)\end{aligned}$$

It is easily verified that smash realizes function composition for the family of deltas.

Most active database systems use smash when combining the impact of different rule firings. In contrast, AP5 uses a special "merge" operator. The *merge*, denoted '&', of two non-*fail* deltas Δ_1 and Δ_2 is given by:

$$(\Delta_1 \,\&\, \Delta_2) = \begin{cases} \Delta_1 \cup \Delta_2 & \text{if this is consistent}\\ \mathit{fail} & \text{otherwise}\end{cases}$$

Thus, the merge of the two deltas of the previous example is *fail*. The use of merge yields a more declarative flavor than smash; this has been exploited in [ZH90] to obtain sufficient conditions on rule-bases to ensure consistent termination of rule firing sequences.

Several other binary operators for combining deltas can be defined, for example, *weak-merge*, i.e., union but deleting all conflicting pairs of signed atoms (cf. [SdM88, CCCR+90]), or union giving priority to inserts in the case of conflict. At present Heraclitus[Alg,C] provides explicit constructs for smash, merge and weak-merge; other binary operators can be built up from more primitive Heraclitus[Alg,C] constructs.

3.3 Embedding into C

We now describe how relational deltas and the algebraic operators described above are embedded into C. The primary focus is on Heraclitus[Alg,C] expressions for (a) creating deltas, (b) combining deltas, and (c) accessing deltas.

Heraclitus[Alg,C] supports the manipulation of both persistent and transient relations and deltas. Suppose that `Suppliers` and `Orders` are persistent relations as defined in the previous section. The following declares two variables for these, and a variable for transient relation `Big`:

```
relation Supp, Ord, Big;
Supp = access_relation("Suppliers");
Ord = access_relation("Orders");
```

```
Big = empty_relation(Part:char[30],
                     Qty:int,
                     Sup:char[30],
                     Exp:int);
```

Signatures for variables Supp and Ord are taken from the corresponding persistent relations. The signature for transient relation variable Big must be specified explicitly upon initialization. While coordinate names may be associated with relation types as indicated here at present the algebra is based on coordinate positions. However, most of our examples will use coordinate names to simplify the exposition. (We assume that Ord has the same field names as Big, and that Supp has field names Sup and Part.) In Subsection 3.4 we use pure Heraclitus[Alg,C] syntax.

The algebra used is essentially the standard relational algebra, except that system- and user-defined scalar functions can be used in projection target lists, and in selection and join conditions (e.g., `project([Part, Qty*2], select({foo(Sup)>Qty}, Orders))` for user-defined function `foo`).

Deltas are supported in Heraclitus[Alg,C] by the type `delta`. Deltas can be created using *atomic* commands, such as

```
delta D1, D2;
D1 = [del Supp("Campy","pedals")];
D2 = [ins Big("brakes",500,"Shimano",
                                   "9/20/93")];
```

After execution D1 has $\{-Suppliers(Campy, pedals)\}$ and D2 has $\{+templ4(brakes, 500, Shimano, 9/20/93)\}$, where *templ4* is the relation identifier chosen during program execution for the transient relation Big. The *bulk* operator can be used to construct a "large" delta from data currently in the database. For example,

```
bulk(ins Big(Part, Qty, Sup, Exp),
     select({Qty > 300}, Ord))
```

evaluates in the context of Figure 2 to

$$\{\ +templ4(frame, 400, Trek, 8/31/93)\ \}$$

More generally, the first argument to bulk must be, what amounts to, an atomic delta expression containing scalar expressions built up from column names and scalar values. These names are assigned possible values by the second argument to bulk, which must be a relation expression. Thus, a **bulk** operator can be viewed as a composition of relational projection followed by parallel creation of atomic delta expressions.

Heraclitus[Alg,C] also supports atomic *modify* expressions, such as `[mod Ord("brakes", 150, "Campy", "9/1/93; "brakes", 150, "Shimano", "9/6/93")]`. Evaluation of this expression depends on the current state: if $(brakes, 150, Campy, 9/1/93)$ is present in Orders (as it is in Figure 1) this expression evaluates to

$$\left\{ \begin{array}{l} -Orders(brakes, 150, Campy, 9/1/93), \\ +Orders(brakes, 150, Shimano, 9/6/93) \end{array} \right\}$$

On the other hand, if $(brakes, 150, Campy, 9/1/93)$ is not present in Orders (as in Figure 2) then the expression evaluates to the empty delta. We have experimented with permitting explicit modifies inside of delta values, on an equal footing with deletes and inserts. However, as reported in [GHJ92], the semantics for consistency and for smash become quite cumbersome in that framework. This has lead us to the compromise that they can be written explicitly, but their value depends on the state. Regardless of this decision, the presence of modify expressions in a program may give the compiler opportunities for optimization (e.g., by avoiding two traversals of an index).

Heraclitus[Alg,C] also permits "wildcards" in delete and modify commands. Wildcards, denoted by '*', match any value. Evaluation of expressions with wildcards again depends on the current database state.

Deltas may be combined using smash (`!`), merge (`&`), and weak-merge explicitly. A fourth operator, *compose*, is also supported; this is described shortly.

We now turn to the four operators for accessing deltas. The first is *apply*: the command apply δ; first evaluates δ and applies the resulting delta value to the current state. Hypothetical expression evaluation is supported by the **when** operator. As a simple example,

```
Big = select({Qty > 300}, Ord) when
([mod Ord("brakes",150,"Shimano","9/6/93";
        "brakes",500,"Shimano","9/20/93")] &
 [ins Ord("light",300,"Cat Paw","9/3/93")]);
```

when evaluated in Figure 2 yields $\{(frame, 400, Trek, 8/31/93), (brakes, 500, Shimano, 9/20/93)\}$. Importantly, side-effect free functions can be called within the context of a **when**. Nesting of **when**'s is also permitted – it is easily verified that

$$(E\ \texttt{when}\ \delta_1)\ \texttt{when}\ \delta_2\ \equiv\ E\ \texttt{when}\ (\delta_2\ !\ (\delta_1\ \texttt{when}\ \delta_2))$$

This plays a key role in the implementation of delta expressions consisting of nested **when**'s.

The final operators for accessing deltas are *peeking* expressions; these permit the programmer to directly inspect a delta. The expression **peekins**(R, δ) evaluates to the relation containing all tuples that are to be inserted into R according to the value of δ, and the expression **peekdel**(R, δ) evaluates analogously. For example, `peekdel(Supp,[del ("Campy",*)])` evaluates in Figure 2 to $\{(Campy, brakes), (Campy, pedals)\}$.

The *compose* operator, denoted '#', has the property that the command apply (δ_1 # δ_2) is equivalent to (apply δ_1; apply δ_2;). Compose is defined in terms of smash and when, by δ_1 # δ_2 = δ_1 ! (δ_2 when δ_1). This definition indicates the difference between smash and compose. In δ_1 ! δ_2, both δ_1 and δ_2 are evaluated with respect to the current state, then smashed, and then applied to the current state. In δ_1 # δ_2, δ_2 is evaluated in the state resulting from the application of δ_1 to the current state. This is reminiscent of the "phantom" problem in database transaction processing. It is straightforward to verify that compose is associative.

Compose is especially useful in the context of hypothetical database access. We present an example involving two functions. The first function builds a delta that has the effect of canceling all October orders:

```
delta cancel_Oct_orders()
{return bulk(del Ord(Part,Qty,Sup,Exp);
             select({in_Oct(Exp)},Ord);}
```

The second one builds a delta that delays the expected date by two weeks of all orders with Qty > 500:

```
delta delay_big_orders()
{return bulk(mod Ord(Part,Qty,Sup,Exp;
                     Part,Qty,Sup,
                       add_two_weeks(Exp)),
             select({Qty > 500}, Ord));}
```

Suppose that the function total_brakes_on_order computes the total number of brakes on order. Then the expression

```
total_brakes_on_order() when
  cancel_Oct_orders() # delay_big_orders()
```

performs a hypothetical evaluation of total_brakes_on_order, assuming that first the October orders where canceled, and then the big orders were delayed. Note the value resulting from the call to delay_big_orders takes into account the updates proposed by the value of cancel_Oct_orders. The following performs the hypothetical evaluation, but with the application of the two delta functions reversed.

```
total_brakes_on_order() when
  delay_big_orders() # cancel_Oct_orders()
```

In general these two expressions will evaluate to different values.

3.4 Active database examples

This subsection provides a brief indication of how Heraclitus[Alg,C] can be used to specify, and thereby implement, a variety of active database execution models. To simplify, we omit consideration of "triggers", and assume rules to have the form:

if <*condition*> **then** <*action*>

Because Heraclitus[Alg,C] provides explicit peeking, triggers can easily be incorporated into the syntax.

Recall the discussion of Subsection 2.1. We adopt here the convention for this discussion that the original database state remains unchanged during rule firing, and that appropriate virtual states are represented and manipulated using deltas. We now specify in Heraclitus[Alg,C] the rules R2 and R4 of Subsection 2.1. It is assumed that deltas corresponding to S_{prop} and S_{curr} are maintained by the execution model. Both rules will be functions with two arguments, although R2 uses only the delta corresponding to S_{curr}.

In Heraclitus[Alg,C], coordinate positions are indicated using the '@' symbol. Typing information is also included here to simplify the task of pre-processing into C, given the fact that relation signatures can change over the lifetime of a program. Thus, in the rule rule_R2, @c1 refers to the first coordinate of the output of the peekdel, which has type character string.

```
delta rule_R2(prop,curr)
  delta prop,curr;
  { return bulk(del Ord(@c2,*,@c1,*);
                peekdel(Supp,curr)); }

delta rule_R4(prop,curr)
  delta prop,curr;
  { if ( (count(Ord) when curr)
         / (count(Ord) when prop) < .8 )
       inventory_warning();
    return empty_delta; }
```

Suppose now that a total of 25 rules are written to capture the purchasing policy for this application, all using input variables corresponding to S_{prop} and S_{curr}. They can be combined into an array of delta functions as follows:

```
delta (*policy[24])();
policy[0] = rule_R1;
policy[1] = rule_R2;
   ⋮
policy[24] = rule_R25;
```

The following function specifies an execution model that takes in a delta corresponding to a user-requested update and applies the rules according to a specific algorithm. Here we use the *copy* ('<<') operator; 'curr << prop;' copies the signed atoms associated with delta variable prop into the delta variable curr. The assignment temp = empty_delta initializes temp as a transient delta holding the empty delta. The expression curr !<< temp; is equivalent to curr << curr ! temp;, and analogously for &<<. The boolean dequiv checks equality of deltas.

```
boolean apply_policy(prop)
delta prop
{
  delta curr, prev, temp;
  if (prop == fail) return (false);
  curr << prop;
  do { prev << curr;
       temp = empty_delta;
       for (i=0; i<25; i++)
         { temp &<< (*policy[i])(prop,curr) };
      curr !<< temp; }
  while ( curr != fail && !dequiv(prev,curr));
  if (curr == fail)
    { return (false); }
  else
    { apply curr;
      return (true); };
}
```

Here, the inner loop corresponds to a single, independent (set-oriented) application of each rule in policy, and combines the results using merge. Note that in the inner loop, each rule is evaluated on prop and curr, and the resulting deltas are accumulated in variable temp. The outer loop repeatedly performs the inner loop, using smash to fold the results of each iteration into the value of curr already obtained. The outer loop is performed until either a fixpoint is reached, or the inner loop produces the delta *fail* (either because one of the rules explicitly called for an abort by producing *fail*, or because in some execution of the inner loop, two rules produced conflicting deltas).

Suppose now that there is a second array **keys** of rule functions capturing key constraints, and that the above execution model is to be modified so that after each execution of the inner loop the rules in **keys** are to be fired until a fixpoint is reached. Suppose further that these rules use only a single input delta, corresponding to S_{curr}. Now let function **apply_rules** have the following signature

```
delta apply_rules(curr, rule_base, size)
  delta curr;
  delta (*rule_base[])();
  int size;
```

and suppose that it applies the rules in rule_base until a fixpoint is reached. Then the desired modification to apply_policy can be accomplished by adding

```
    curr !<< apply_rules(curr,keys,15);
```

as the last line of the inner loop. This very briefly indicates the kind of flexibility that Heraclitus[Alg,C] provides in specifying active database execution models.

We are currently implementing in Heraclitus[Alg,C] the (kernel of the) execution models of the Starburst Rule System, AP5, and POSTGRES systems. Specifications for Starburst and AP5 in Heraclitus pseudo-code were presented in [HJ91b].

4 The Implementation of Heraclitus[Alg,C]

The implementation of Heraclitus[Alg,C] has two components: HERALD, a library of relational and delta operators built on top of Exodus, and a pre-processor that maps Heraclitus[Alg,C] programs into C programs with calls to HERALD. We discuss the pre-processor first.

4.1 The pre-processor

The pre-processor for HeraclitusAlg,C] was implemented by modifying the GNU C compiler. We mention here only of several significant aspects of the preprocessor, namely, the implementation of when's.

Consider the expression join(<*cond*> , R, S) when D. This cannot be evaluated in the traditional bottom-up manner, because the relationships of D with R and S are lost if the join is performed. Instead, the when must be "pushed" inwards, through the join operator, to directly modify the relations. A naive approach to this problem is to have the compiler "replace" the above expression by join(<*cond*>, R when D, S when D). before passing it to HERALD. A complication arises, however, because Heraclitus[Alg,C] permits functions that reference the database state to be called in the context of a when, e.g., goo(u,v) when D. This means that essentially any expression may have to be evaluated hypothetically, but the relevant delta is known only at runtime. In the current implementation we maintain a "runtime when stack". During the execution of a program the top of the stack holds a delta that reflects the full effect of all deltas relevant to the evaluation of the expression currently under consideration. This has the same impact as pushing when's to the leaves of the syntax tree.

As an aside, we note that in the context of database programming languages such as Heraclitus[Alg,C], queries are generally accessible only at runtime due to the presence of function calls. This highlights one of the key differences between query processing in conventional databases, where the full query tree is available at compile time, and query processing in database programming languages.

4.2 HERALD

A central aspect of the HERALD system is to combine the evaluation of **when**'s with evaluation of the algebraic operators, in a manner reminiscent of the traditional relational optimization of combining selects and projects with joins. For example, HERALD provides a ***hypothetical join*** function **join_when**, that evaluates the expression `join( <cond>, R when D, S when D)`. without materializing `R when D` or `S when D`. HERALD currently supports two strategies for obtaining access to deltas in connection with the hypothetical algebraic operators and other delta operators, one based on hashing and the other on a sort-merge paradigm.

Conceptually, HERALD represents a delta as a collection of pairs (R^+_Δ, R^-_Δ), specifying the proposed inserts and deletes for each relation variable R in the program. Here, R^+_Δ and R^-_Δ are called ***sub-deltas***, and are stored as relations (actually, files) in Exodus. Hash-based access is best suited for the situation where a subdelta pair (R^+_Δ, R^-_Δ) fits into main memory, and sort-based access is better when a subdelta pair is bigger than main memory.

In the remainder of the section we discuss hash-based and sort-based access to deltas.

4.2.1 Hash-based access to deltas

When sub-deltas are small enough to fit in main memory, HERALD maintains a hash index on each sub-delta. The hash index key value to address this hash table is composite and computed based on the values of all fields (or attributes) of a record. As demonstrated in Section 5, this implementation technique is effective as long as a delta fits in main memory. We now describe the low-level algorithms for two representative delta operators, namely **select_when** and **join_when**.

Select_when. The input arguments of this operator are: a relation R, a selection condition, a delta Δ, and an output relation. Logically, this operator selects tuples of R that satisfy the selection condition in the hypothetical state proposed by Δ and stores the resulting tuples in the output relation. Its implementation is as follows:

```
1. open a scan on $R$
2. get the first tuple of $R$ (say $t$)
3. while not EOF(R) do
   a. evaluate the selection condition for $t$. If the tuple
      does not qualify go to step e.
   b. probe the hash index of $R^+_\Delta$ with $t$ for a matching
      tuple, if found go to step e.
   c. probe the hash index of $R^-_\Delta$ with $t$ for a matching
      tuple, if found go to step e.
   d. insert $t$ into the output relation.
   e. get the next tuple $t$ in $R$.
4. for each tuple $t$ of $R^+_\Delta$ do
   a. evaluate the selection condition for $t$. If $t$ satisfies
      this condition, then insert $t$ into the output
      relation.
```

Note that we probe the hash index only if the tuple satisfies the selection condition. This minimizes the number of disk accesses because probing the hash index may result in a disk read operation.

We briefly analyze the expected I/O costs of this implementation of **select_when**. Suppose that R^+_Δ, R^-_Δ are small enough to fit into main memory, and that s% of the tuples in R satisfy the selection condition. Assuming that $s > 0$, the algorithm will call for the following I/Os:

(a) scan R
(b) scan hash tables for R^+_Δ, R^+_Δ.
(c) probe R^-_Δ for s% of R
(d) probe R^+_Δ for s% of R
(e) scan R^+_Δ
(f) write output relation

Thus, the expected overhead in I/O is roughly equal to the number of pages of the hash tables for R^+_Δ and R^+_Δ, and the number of pages of R^+_Δ and R^-_Δ that are read during parts (b) and (c). (An additional scan of all of R^+_Δ and R^-_Δ is needed if hash tables are not maintained.) This was confirmed in our benchmarking experiments.

Join_when. In the current implementation, the binary relational operators use sort-based implementations. In the case of hash-based delta access, a key subroutine for all of them is **sort_when**. Suppose that R is unsorted. The conventional approach to sorting R is to use heap-sort on short (e.g., 100 page) segments of R, and then to perform n-way merges of these segments. In **sort_when**, the impact of a delta is incorporated into the heap-sort. For example, on relation R, as portions of R are read in for heap-sorting, a hash-table for R^-_Δ is probed, and the matching tuples are not placed into the heap. Also redundant tuples in R^+_Δ are marked, to prevent later duplication. After R is completely read, the remainder of R^+_Δ is also processed by the heap sort to provide additional sorted segments. Then one or more merges is invoked to create a sorted file. In the current implementation for join with hash-based delta access, **sort_when** is used to sort R (as impacted by R^+_Δ, R^-_Δ) and S (as impacted by S^+_Δ, S^-_Δ), and then a binary merge is used to create the join. Although not currently implemented, this could be optimized by combining the final merge with the separate merges inside the two calls to **sort_when**.

When using hash-based delta access for these operators, there is an important interaction between the amount of buffer space used by the heap vs. the hash tables. To illustrate, suppose in the abstract that the total available buffer pool consists of 100 frames (and so the heap-sort can perform 100-way merges). Moreover, assume that R consists of 1000 pages, R^- has about 90 pages that will be probed during a pass of R (termed "hitting" pages), and R^+ is empty. In this case a 10-page heap could be established, and $R - R^-$ would be broken into roughly 100 (or fewer) sorted segments. Now a single 100-way merge will yield a sorted version of *apply*(R, R^-); total cost is $2|R| + |R^-|$. Suppose now that R has 2000 pages, R^- has about 80 "hitting" pages, and R^+ is empty. It is now optimal to devote 20 pages to the heap-sort and the other 80 to hash probing. (Fewer pages for the heap-sort results in more merge passes; and fewer pages for the hash probing may result in thrashing.) Thus, providing optimal support for hash-based delta access requires the ability to dynamically partition the buffer pool between these two tasks. This capability is supported by Exodus, and we plan to investigate these trade-offs in our future research.

4.2.2 Sort-based access to deltas

A delta may be so large that it does not fit in main memory, in which case the hash-based implementation will thrash. To remedy this, we have designed and implemented algorithms that access deltas using a sort and merge technique. We now present the low level algorithm for the **select_when** operator; the implementation of other operators is analogous. Heraclitus[Alg,C] maintains information on whether relations and subdeltas are sorted, so that one or more of the sorting steps of these sort-based algorithms can be eliminated.

select_when. The input arguments of this operator are: a relation R, a selection condition, a delta Δ, and an output relation. We assume that no order is maintained for any of the inputs. A key function used here is **select_sort** which takes as input a relation and a selection condition. As with **sort_when**, this implements a two-phase sort, but in the heap-sort phase it deletes all tuples violating the selection condition.

In the following algorithm, if no tuples satisfy the selection condition (i.e., *Temp* is empty), then R^+_Δ is scanned for the qualifying tuples and returns. Otherwise, it sorts the qualifying tuples found in each of R^-_Δ and R^+_Δ into two different temporary relations. Next, it performs a three way merge on these relations, inserting one occurrence of entries of R that match with R^+_Δ (prevent duplicates) and eliminating those that match with R^-_Δ (tuples proposed to be deleted).

```
1. select_sort ($R$, selection condition) into a temporary relation $Temp$.
2. if $Temp$ is empty, then
   a. for each tuple t of $R^+_\Delta$, evaluate the selection condition for $t$. If $t$ satisfies this condition, then insert $t$ into the output relation.
   b. return as the output relation and exit.
3. select_sort ($R^-_\Delta$, selection condition) into a temporary relation $Temp^-$.
4. select_sort ($R^+_\Delta$, selection condition) into a temporary relation $Temp^+$.
5. retrieve the first tuple in $Temp$ (say $r$), $Temp^-$ (say $d-$), and $Temp^+$ (say $d+$).
6. while not EOF($Temp$) OR not EOF($Temp^-$) OR not EOF($Temp^+$) do
   a. assign $t$ to be the tuple with minimum value among $r$, $d+$, and $d-$.
   b. If $t$ is not equivalent to $d-$, then insert $t$ into the output relation.
   c. If $t$ is equivalent to $r$, then get the next tuple $r$ from $Temp$.
   d. If $t$ is equivalent to $d-$, then get the next tuple $d-$ from $Temp^-_\Delta$.
   e. If $t$ is equivalent to $d+$, then get the next tuple $d+$ from $Temp^+_\Delta$.
```

We now analyze the expected I/O cost of this implementation of **select_when**, under the assumption that the inputs are not maintained in sorted order. Let $P(R)$ represent the number of disk pages for relation R, $SP(R)$ represents the number of disk pages that satisfy the selection condition, and analogously for R^+_Δ and R^+_Δ. We assume that $SP(R) \leq$ the square of the number of available pages in the buffer pool (i.e., that only one n-way merge is need to sort R), and similarly for R^+_Δ and R^+_Δ. The total number of I/Os incurred by the above algorithm can be estimated as the sum of:

$$\begin{aligned}
&\textit{select\_sort}(R): && P(R) + 2 * SP(R)\\
&\textit{select\_sort}(R^+_\Delta): && P(R^+_\Delta) + 2 * SP(R^+_\Delta)\\
&\textit{select\_sort}(R^-_\Delta): && P(R^-_\Delta) + 2 * SP(R^-_\Delta)\\
&\textit{merge}: && 2 * SP(R) + 2 * SP(R^+_\Delta) + SP(R^-_\Delta)
\end{aligned}$$

This cost function is a worst case estimate because it assumes: (1) SP(R) is not empty, (2) the tuples of SP(R) are not redundant with those in SP(R^+_Δ), causing all their entries to be written to the output relation, and (3) the tuples of SP(R) do not match with the tuples found in R^-_Δ.

The implementation also handles the case where the input relation and delta are sorted. In this case, only steps (5) and (6) of the algorithm are executed, and the selection condition is incorporated into step (6).

5 An Evaluation of HERALD

Heraclitus[Alg,C] and the underlying library HERALD are currently operational on a Sun SPARCstation 2 using the UNIX operating system. Using this implementation, we characterized the performance of HERALD for executing the delta operators and hypothetical relational operators. The goals of this evaluation were to: (a) characterize the tradeoffs associated with the alternative techniques employed by HERALD, and (b) quantify the different factors that impact the performance of the implementation. We analyzed the alternative implementation of deltas as a function of alternative buffer pool, disk page, relation, and delta sizes. Several other factors were also considered including conflicts between deltas, and the percentage of redundant tuples between two deltas. All experiments confirmed our hypotheses that: (1) the hash based implementation is efficient for small deltas that fit in main memory, and degrades due to thrashing of the buffer pool as the delta size grew larger than main memory, (2) the sort-based implementation is appropriate for large deltas, (3) the sort-based implementation benefits from a "lazy" application of deltas (see below) as long as the deltas are smaller than the referenced relations, (4) the hash-based implementation does not necessarily benefit from a lazy application, (5) the number of I/Os performed by the sort-based implementation decreases as a function of larger disk page sizes, (6) the hash-based implementation benefits from the larger disk page size to a certain point, beyond which, it results in a thrashing behavior, and (7) when executing a program, if the order of records in both the referenced delta and relation are maintained, on small deltas the sort-based access can provide performance identical to hash-based access.

To illustrate our evaluation, we present here the most interesting experiment conducted, which compares lazy vs. eager application of deltas. For this experiment we generated HERALD with 4 kilobyte pages and a 128 page buffer pool. In our experiments we used the number of disk read and write operations performed by our storage manager as the measurement criteria. We did not use the response time of the system, because the obtained results would not have been meaningful as we had no control over the buffer pool replacement policy employed by the underlying UNIX operating system [Sto81]. The use of the number of disk I/O operations is justified as it constitutes the dominant portion of the system response time[2].

Figure 3 presents the percentage savings provided by evaluating the expression apply(DB,Δ_1!Δ_2) (termed "lazy application") as compared to apply(apply(*DB*, Δ_1),Δ_2) (termed "eager application"); these have equivalent impact on the database state. As with all of our experiments, we used benchmark relations based on the standard Wisconsin benchmark relations [BDT83]. Each relation consisted of tuples that are 208 bytes wide. For this experiment we focused on a relation having 10K tuples (about 500 pages), and deltas ranging in size from 5% to 70% of the relation size. The size of the two deltas Δ_1, Δ_2 was identical, with each of Δ_1^+, Δ_1^-, Δ_2^+, and Δ_2^- containing the same number of signed atoms. In addition to varying the size of the deltas, we varied the amount of *conflict* between them. Here, 0% conflict means that the smash of Δ_1 and Δ_2 is equal to their union. At the other extreme, 100% conflict means that each signed atom in Δ_2 "undoes" a signed atom in Δ_1; in this case Δ_1 ! $\Delta_2 = \Delta_2$. For example, then, in the case of 20% deltas (about 100 pages each), the smash with 0% conflict has about 200 pages, the smash with 50% conflict has about 150 pages, and the smash with 100% conflict has about 100 pages.

In these experiments, the system was configured to use both the size of the input deltas and the buffer pool in order to choose the implementation of an operator that resulted in the best performance. For example, each 5% delta is small enough (25 pages) to fit in the buffer pool and the system uses the hash-based implementation of both smash (!) and apply operators. The 70% delta, on the other hand, is larger than the buffer pool and the system uses the sort-based implementation in order to prevent thrashing associated with the buffer pool. The 20% delta required a hybrid

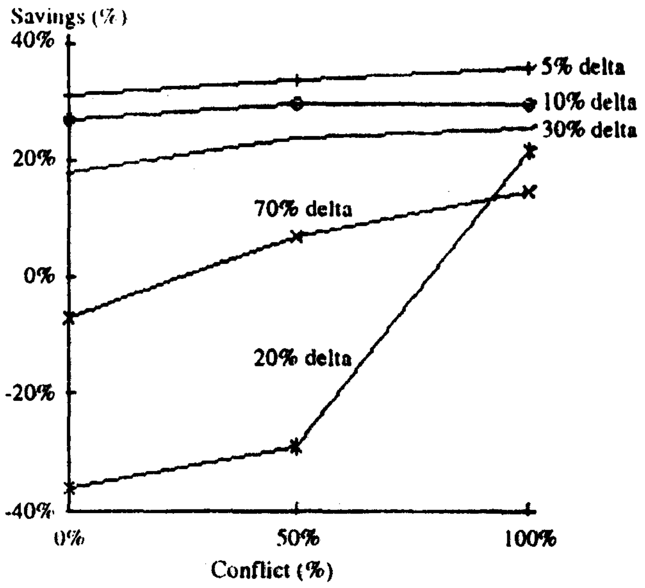

Figure 3: Lazy vs. Eager delta application

[2]With a 25 MIPS SPARC CPU, the CPU processing time of an operator is not as significant as compared to the service time of its disk read or write operations.

implementation: for the lazy case, the hash-based apply could be used, but for the eager case with 0% or 50% conflict, the result of the smash was bigger than the buffer pool, and so the sort-based apply was used.

The obtained results show two general trends. First, as the percentage conflict between two deltas increases the percentage savings obtained by the lazy application of deltas increases. This is because the lazy application requires a smash between Δ_1 and Δ_2 to generate a new conflict free delta (Δ_3). The size of Δ_3 with a 100% conflict is one half that with a 0% conflict, reducing both the number of writes to generate Δ_3 and reads to process the subsequent apply operator. Second, as the size of Δ_1 and Δ_2 increases, the benefits of lazy application decreases. One reason for this is that in the case of lazy application, the processing associated with Δ_3 (e.g., building a hash table for it) has increased cost. A larger granularity reason is that the system starts to use the sort-based implementation of operators, and so the cost of performing the smash (with lazy application) approaches that of apply (with eager application). An exception to this trend is the 20% delta; this is because, as noted above, for the cases of 0% and 50% a sort-based apply was used in the case of lazy evaluation, whereas hash-based applies could be used by the eager evaluation.

6 Conclusions

This paper describes the current status of the Heraclitus project. A long-range goal is to develop and implement language constructs and techniques for the flexible specification and implementation of a wide variety of execution models for active databases. The current focus has been on the development of the language Heraclitus[Alg,C], that extends C with the relational algebra, deltas, and delta operators, and uses Exodus to provide bulk data access. The main research contributions of the implementation have been (a) understanding feasible physical implementations of the algebraic operators, and (b) understanding the implications of embedding the Heraclitus paradigm for database access into an imperative programming language. As shown here, the delta paradigm and Heraclitus[Alg,C] are especially well-suited for working with virtual states, as arise in several active databases in the literature, and for specifying how the results of fired rules should be combined. The preliminary performance evaluation shows that the cost of both explicit and implicit manipulation of deltas is not prohibitive, and provides the first step towards a substantial optimization effort.

A primary short-range goal of the project is the further development and improvement of Heraclitus[Alg,C]. Enhancements will proceed along two dimensions: (a) work on further optimization, both at the compiler and storage manager levels, including investigation of the use of indexes (e.g., B^+-trees) and parallelism,, and (b) the development of high level macros to facilitate the specification of execution models. Over the longer term, we intend to generalize to the framework of object-oriented databases, and to incorporate concurrency. We also plan to investigate the use of deltas in connection with other applications, including hypothetical database access, version control, specifying concurrency protocols, and resolving update conflicts.

7 Acknowledgments

We would like to thank Hsun-Ko Chan, Elton Chen, Vera Choi, Yuji Enzaki, Sanjay Joshi, Yoram Landau, Michel Liu, Aris Prassinos, Babar Saeed, Howard Ungar and Jackie Yu for assisting in the research and implementation effort in connection with this paper. We would also like to thank Serge Abiteboul, Eric Simon, and Roger King for useful discussions on the topics covered in the paper.

References

[Abi88] Serge Abiteboul. Updates, A new frontier. In *Proc. of Intl. Conf. on Database Theory*, 1988.

[BDT83] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Databases Systems: A Systematic Approach. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 8–19, 1983.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading Massachusetts, 1985.

[BM91] C. Beeri and T. Milo. A model for active object oriented database. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 337–349, 1991.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 225–236, 1990.

[CDKK85] H-T. Chou, D. DeWitt, R. Katz, and T. Klug. Design and implementation of the Wisconsin Storage System (WiSS). *Software Practices and Experience*, 15(10), October 1985.

[CDRS86] M. J. Carey, D. J. DeWitt, Joel E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of VLDB*, pages 91–100, 1986.

[CHM92] I.-M. A. Chen, R. Hull, and D. McLeod. Derived data update via limited ambiguity, December 7 1992. USC technical report.

[Coh86] Don Cohen. Programming by specification and annotation. In *Proc. of AAAI*, 1986.

[CW90] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, 1990.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[CW92] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. Technical Report RJ 9064 (80754), IBM Research Division, Almaden Research Center, October 30 1992.

[dMS88] C. de Maindreville and E. Simon. Modeling non-deterministic queries and updates in deductive databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, 1988.

[GHJ92] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Implementation of delayed updates in Heraclitus. In *Proc. of Intl. Conf. on Extending Data Base Technology*, 1992.

[Han89] E.N. Hanson. An initial report on the design of Ariel: A DBMS with an integrated production rule system. *SIGMOD Record*, 18(3):12–19, September 1989.

[HJ91a] R. Hull and D. Jacobs. On the semantics of rules in database programming languages. In J. Schmidt and A. Stogny, editors, *Next Generation Information Systẽm Technology: Proc. of the First International East/West Database Workshop, Kiev, USSR, October 1990*, pages 59–85. Springer-Verlag LNCS, Volume 504, 1991.

[HJ91b] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1991.

[HW92] E. N. Hanson and J. Widom. An overview of production rules in database systems. Technical Report RJ 9023 (80483), IBM Almaden Research Center, October 12, 1992.

[IN88] T. Imielinski and S. Naqvi. Explicit control of logic programs through rule algebra. In *Proc. ACM Symp. on Principles of Database Systems*, pages 103–116, 1988.

[JH91] D. Jacobs and R. Hull. Database programming with delayed updates. In *Intl. Workshop on Database Programming Languages*, pages 416–428, San Mateo, Calif., 1991. Morgan-Kaufmann, Inc.

[KDM88] A.M. Kotz, K.R. Dittrich, and J.A. Mülle. Supporting semantic rules by a generalized event / trigger mechanism. In *Intl. Conf. on Extending Data Base Technology*, pages 76–91, 1988.

[MD89] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 215–224, 1989.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 34–42, 1983.

[MP90] C.B. Medeiros and P. Pfeffer. A mechanism for managing rules in an object-oriented database. Technical report, Altair, 1990.

[SdM88] E. Simon and C. de Maindreville. Deciding whether a production rule is relational computable. In *Proc. of Intl. Conf. on Database Theory*, pages 205–222, 1988.

[SIG89] SIGMOD Record 18:9, "Special Issue on Rule Management and Processing in Expert Database Systems", September 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 281–290, 1990.

[SL76] D.G. Severance and G.M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. on Database Systems*, 1(3), September 1976.

[Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[Sto92] M. Stonebraker. The integration of rule systems and database systems. *IEEE Trans. on Knowledge and Data Engineering*, 4(5):415–423, October 1992.

[WF90] Jennifer Widom and Sheldon J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 259–264, 1990.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

[WS83] J. Woodfuill and M. Stonebraker. An implementation of hypothetical relations. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 157–165, September 1983.

[ZH90] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Intl. Conf. on Extending Data Base Technology*, pages 407–421, 1990.