The Declarative Imperative: Experiences and Conjectures in Distributed Logic

Joseph M. Hellerstein

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-90
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-90.html

June 1, 2010

The Declarative Imperative

Experiences and Conjectures in Distributed Logic

Joseph M. Hellerstein
University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

The rise of multicore processors and cloud computing is putting enormous pressure on the software community to find solutions to the difficulty of parallel and distributed programming. At the same time, there is more—and more varied—interest in data-centric programming languages than at any time in computing history, in part because these languages parallelize naturally. This juxtaposition raises the possibility that the theory of declarative database query languages can provide a foundation for the next generation of parallel and distributed programming languages.

In this paper I reflect on my group's experience over seven years using Datalog extensions to build networking protocols and distributed systems. Based on that experience, I present a number of theoretical conjectures that may both interest the PODS community, and clarify important practical issues in distributed computing. Most importantly, I make a case for PODS researchers to take a leadership role in addressing the impending programming crisis.

This is an extended version of an invited lecture at the ACM PODS 2010 conference entitled "Datalog Redux: Experience and Conjecture." [31].

1. INTRODUCTION

This year marks the forty-fifth anniversary of Gordon Moore's paper laying down the Law: exponential growth in the density of transistors on a chip. Of course Moore's Law has served more loosely to predict the doubling of computing efficiency every eighteen months. This year is a watershed: by the loose accounting, computers should be 1 Billion times faster than they were when Moore's paper appeared in 1965.

Technology forecasters appear cautiously optimistic that Moore's Law will hold steady over the coming decade, in its strict interpretation. But they also predict a future in which continued exponentiation in hardware performance will only be available via parallelism. Given the difficulty of parallel programming, this prediction has led to an unusually gloomy outlook for computing in the coming years.

At the same time that these storm clouds have been brewing, there has been a budding resurgence of interest across the software disciplines in data-centric computation, including declarative programming and Datalog. There is more—and more varied—applied activity in these areas than at any point in memory.

The juxtaposition of these trends presents stark alternatives. Will the forecasts of doom and gloom materialize in a storm that drowns out progress in computing? Or is this the long-delayed catharsis that will wash away today's thicket of imperative languages, preparing the ground for a more fertile declarative future? And what role might PODS play in shaping this future, having sowed the seeds of Datalog over the last quarter century?

Before addressing these issues directly, a few more words about both crisis and opportunity are in order.

1.1 Urgency: Parallelism

> *I would be panicked if I were in industry.*
> — John Hennessy, President, Stanford University [34]

The need for parallelism is visible at micro and macro scales. In microprocessor development, the connection between the "strict" and "loose" definitions of Moore's Law has been severed: while transistor density is continuing to grow exponentially, it is no longer improving processor speeds. Instead, chip manufacturers are packing increasing numbers of processor cores onto each chip, in reaction to challenges of power consumption and heat dissipation. Hence Moore's Law no longer predicts the clock speed of a chip, but rather its offered degree of parallelism. And as a result, traditional sequential programs will get no faster over time. For the first time since Moore's paper was published, the hardware community is at the mercy of software: only programmers can deliver the benefits of the Law to the people.

At the same time, Cloud Computing promises to commoditize access to large compute clusters: it is now within the budget of individual developers to rent massive resources in the worlds' largest computing centers. But again, this computing potential will go untapped unless those developers can write programs that harness parallelism, while managing the heterogeneity and component failures endemic to very large clusters of distributed computers.

Unfortunately, parallel and distributed programming today is challenging even for the best programmers, and unworkable for the majority. In his Turing lecture, Jim Gray pointed to discouraging trends in the cost of software development, and presented *Automatic Programming* as the twelfth of his dozen grand challenges for computing [25]: develop methods to build software with orders of magnitude less code and effort. As presented in the Turing lecture, Gray's challenge concerned sequential programming. The urgency and difficulty of his twelfth challenge has grown markedly with the technology trends toward parallelism. Hence the spreading cloud of doom and gloom.

1.2 Resurgency: Springtime for Datalog

> *In the spring time, the only pretty ring time*
> *When birds do sing, hey ding a ding, ding;*
> *Sweet lovers love the spring.*
> — Shakespeare

With these storm clouds on the horizon, it should be a matter of some cheer for database theoreticians that Datalog variants, like

crocuses in the snow, have recently been seen cropping up outside the walled garden of PODS. Datalog and related languages have been proposed for use in a wide range of practical settings including security and privacy protocols [36, 19, 78], program analysis [42, 72, 27], natural language processing [20, 69], probabilistic inference [7, 70], modular robotics [6], multiplayer games [73], networking [45] and distributed systems [2]. The renewed interest appears not to be the result of a coordinated effort, but rather (to hybridize metaphors) a grassroots phenomenon arising independently in different communities within computer science.

Over the past few years, my group has nurtured a patch of this activity in the unlikely ground of Berkeley's systems projects, with a focus on inherently parallel tasks in networking and distributed systems. The effort has been quite fruitful: we have demonstrated full-featured Datalog-style implementations of distributed systems that are orders of magnitude more compact than popular imperatively implemented systems, with competitive performance and significantly accelerated software evolution [45, 2]. Evidence is mounting that Datalog can serve as the rootstock of a much simpler family of languages for programming serious parallel and distributed software. Encouraged by these results, we are cultivating a new language in this style for Cloud Computing, which we call *Bloom*[1].

1.3 Synergy: The Long-Awaited Question

> *It shall be:*
> *when I becloud the earth with clouds,*
> *and in the clouds my bow is seen,*
> *I will call to mind my covenant*
> *that is between me and you and all living beings—all flesh: never again shall the waters become a Deluge,*
> *to bring all flesh to ruin!*
> – *Genesis*, 8:14-15 [23]

Though Gray speaks only vaguely about "non-procedural" languages in his Turing lecture, it is hard to imagine he did not have in mind the success of SQL over COBOL as one model for progress[2]. And parallelism has proved quite tractable in the SQL context. Recently, David Patterson wrote soberly of the "Dead Computer Society" of parallel hardware vendors in the 1980's [34], but notably omitted the survivor from that era: parallel database pioneer Teradata. It happens that the relational algebra parallelizes very naturally over large datasets, and SQL programmers benefit without modifications to their code. This point has been rediscovered and amplified via the recent enthusiasm for MapReduce programming and "Big Data," which have turned data-parallelism into common culture across computing. It seems that we are all database people nowadays.

The Parallel Computing literature traditionally pooh-poohs these examples as "embarrassingly parallel." But should we really be embarrassed? Perhaps after a quarter century of fighting the "hard" problems of parallelism, the rational way forward is to start with an "easy" kernel to parallelize—something like the relational algebra—and then extend that kernel to more general-purpose computation. As PODS veterans well know, database languages have natural Turing-complete extensions (e.g., [10, 67]).

[1]In tribute to Gray's twelfth challenge, our research project is called *BOOM: the Berkeley Orders Of Magnitude* project. Bloom is the language of BOOM. We hope BOOM and Bloom can be an antidote to doom and gloom.

[2]Butler Lampson filled this gap in his follow-up article in the 50^{th} anniversary issue of *J. ACM*, though he questioned the generality of declarative approaches [39].

This direction for tackling parallelism and distribution raises questions that should warm the heart of a database theoretician. How does the complexity hierarchy of logic languages relate to parallel models of computation? What are appropriate complexity models for the realities of modern distributed systems, where computation is cheap and coordination is expensive? Can the lens of logic provide better focus on what is "hard" to parallelize, what is "embarrassingly easy," and what falls in between? And finally, a question close to the heart of the PODS conference: if Datalog has been The Answer all these years, is the crisis in parallel and distributed programming The Question it has been waiting for?

I explore some of these issues below, by way of both experience and conjecture.

2. BACKGROUND: DEDALUS

> *We work on the other side of time.*
> — Sun Ra

It has been seven years since my group began exploring the use of recursive queries to implement systems, based on languages including NDlog [46], Overlog [16], and SNLog [14]. But only in the last twelve months have we settled on a Datalog variant that cleanly captures what we see as the salient semantic issues for parallel and distributed computing. We call the language Dedalus, and its key contribution is the use of time as an organizing principle for distributed systems, rather than distance in physical space[3] [4]. The design of Dedalus captures the main semantic reality of distribution: two computers are effectively "distributed" if they cannot directly reason about each other's perception of time. The time dimension in Dedalus succinctly captures two important aspects of time in distributed systems: intra-node atomicity and sequencing of state transitions, and inter-node temporal relations induced by the receipt of networked data.

Dedalus clarifies many issues that were semantically ambiguous in our early work, and I will use it throughout this paper, even for examples that predate the language. Before proceeding, I pause for a casual introduction to Dedalus targeted at PODS readers familiar with Datalog.

Dedalus is a simple temporal extension to stratified Datalog in which each relation schema has a "timestamp" attribute in its rightmost position. For intuition, this attribute can be considered to contain sequence numbers from a logical clock. The use of this attribute will always be clear from context, so we can omit it from the syntax of Dedalus predicates as we will see shortly.

There are three kinds of rules in Dedalus:

- *Deductive* rules, in which all predicates share the same variable in the timestamp attribute. For such rules, we omit the timestamps completely, and the result looks like traditional Datalog. The first two rules of Figure 1 are deductive; all predicates in those rules should be considered to have a variable `T` in their missing rightmost position. Intuitively, they express deduction within each timestep.

- *Inductive* rules have the same timestamp variable in all body predicates, but the head's timestamp variable is equal to the

[3]My student Peter Alvaro explains the name as follows: "Dedalus is intended as a precursor language for Bloom in the BOOM project. As such, it is derived from the character Stephen Dedalus in James Joyce's *Ulysses*, whose dense and precise chapters precede those of the novel's hero, Leopold Bloom. The character Dedalus, in turn, was partly derived from Daedalus, the greatest of the Greek engineers and father of Icarus. Unlike Overlog, which flew too close to the sun, Dedalus remains firmly grounded." [4]

```
toggle(1) :- state(0).                  toggle(1, T) :- state(0, T).
toggle(0) :- state(1).                  toggle(0, T) :- state(1, T).
state(X)@next :- toggle(X).             state(X, S) :- toggle(X, T), succ(T, S).
announce(X)@async :- toggle(X).         announce(X, S) :- toggle(X, T), choice({X,T}, {S}).
```

Figure 1: A simple Dedalus program, written with syntactic sugar (left), and with standard Datalog notation (right).

successor of the body predicates' timestamp variable. In this case we omit the body predicates' timestamp variable, and mark the head predicate with the suffix `@next`. The third rule of Figure 1 is inductive; all the body predicates have an omitted rightmost variable `T`, the head has an omitted rightmost variable `S`, and there is an implicit body predicate `succ(T, S)`. Intuitively, this rule says that the `state` predicate at timestep `T + 1` will contain the contents of `toggle` from timestep `T`.

- *Asynchronous* rules are like inductive rules, except that the head's timestamp variable is chosen non-deterministically for each binding of the body variables in time, using Greco and Zaniolo's `choice` construct [26]. We notate asynchronous rules with the head suffix `@async`. The final rule of Figure 1 is asynchronous. It can be read akin to the inductive rule case, but with a different implicit body predicate: `choice({X, T}, {S})`, which indicates that for each pair of assignments to variables `{X, T}`, a value `S` is non-deterministically chosen. Intuitively, this syntax says that `announce` tuples are copies of `toggle` tuples, but the `announce` tuples contain (or "appear at") a non-deterministic timestep. Positive infinity is included in the domain of timestamps, corresponding to the possibility of failure in computing or communicating an asynchronous result. Most useful programs constrain the head timestamp to be larger than the body timestamp, but this is not a requirement of the language. In Section 4.2 I return to the topic of Dedalus programs that can send messages into their own past.

Dedalus includes timestamps for three reasons: to capture state visibility via timestamp unification, to capture sequential atomic update via inductive rules, and to account for the unpredictable network delays, failures and machine-clock discrepancies that occur in distributed systems via asynchronous rules. I return to these issues below, in contexts where the predecessors to Dedalus ran into difficulties.

3. EXPERIENCE

> *No practical applications of recursive query theory ...*
> *have been found to date.*
> —Michael Stonebraker, 1998
> *Readings in Database Systems, 3rd Edition*
> Stonebraker and Hellerstein, eds. [33]

Over the last seven years we have implemented and published a wide range of algorithms, protocols and complete systems specified declaratively in Datalog-like languages. These include distributed crawlers [17, 48], network routing protocols [49], overlay networks including Chord [47], distributed Bayesian inference via message passing on junction trees [7], relational query optimization [16], distributed consensus (Paxos) and two-phase commit [3], sensornet protocols [14], network caching and proxying [13, 15], file systems and job schedulers [2].

Many of these efforts were justified in terms of radical reductions in code size, typically orders of magnitude smaller than competing imperative implementations. In some cases [46, 13, 15], the results also demonstrated the advantages of automatic optimizations for declarative programs.

As a student I had little love for Datalog, and it is tempting to make amends by documenting my growing appreciation of the language and its literature. But my learning process has been slow and disorderly, and remains far from complete; certainly not a useful organizing structure for sharing the experiences from my group. Instead, this section is organized thematically. I start by describing some general behaviors and design patterns we encountered, some deficiencies of the languages we have struggled with, and implications of these for parallelism and distribution.

3.1 Recursion (Rewriting The Classics)

Our work on declarative programming began in reaction to the Web, with its emphasis on large graphs and networks. As we began working directly on this idea, we found that Datalog-style recursion had natural applications and advantages in many settings. There is no question in our minds today that 1980's-era arguments against the relevance of general recursion were short-sighted. Unfortunately, there has been too little success connecting the dots between potential and reality in this domain. Critical Web infrastructure for managing large graphs is still written in imperative languages. Closer to home, traditional RDBMS internals such as dynamic programming are also coded imperatively. Part of our agenda has been to simultaneously highlight the importance of recursion to practitioners in the database field, and to highlight the importance of declarative programming to practitioners in the systems field.

3.1.1 Finding Closure Without the Ancs

Classic discussions of Datalog start with examples of transitive closure on family trees: the dreaded `anc` and `desc` relations that afflicted a generation of graduate students[4]. My group's work with Datalog began with the observation that more interesting examples were becoming hot topics: Web infrastructure such as webcrawlers and PageRank computation were essentially transitive closure computations, and recursive queries should simplify their implementation. To back up this claim, we began by building a Deep Web data crawler using recursive streaming SQL in the Telegraph project [17]. Subsequently we built a distributed crawler for the Gnutella peer-to-peer network as a cyclic dataflow of relational algebra in the PIER p2p query engine [48]. Both of these examples were simple monotonic programs that accumulated a list of the nodes reached from one or more starting points. We later built more sophisticated distributed programs with transitive closure at their core, including network routing protocols for Internet and wireless settings [49, 46, 14], and distributed versions of Bayesian belief propagation algorithms that pass weights along the edges [7]. As expected, Datalog was an excellent language for expressing transitive closures and graph traversals, and these tasks were almost

[4]The tedium of tiresome table-names (*l'ennui de l'entité*) goes back to the founders of Datalog; the original victims can be identified by a same-generation query. However, victims often grow into abusers—a form of transitive closure—and I confess to occasional pedagogical lapses myself. This phenomenon is of course not limited to Datalog; any student of SQL can `emp`athize.

trivial to code.

Building upon previous experience implementing RDBMS internals, my group found it relatively easy to build single-node implementations of the recursive query engines supporting these ideas. But to move to a distributed setting, two issues remained to be worked out: specification of distributed joins, and modifications to recursive query evaluation to allow asynchronous streaming of derivations across networks. These issues are discussed in Section 3.2.

3.1.2 DP and Optimization: Datalog in the Mirror

Another recursive design pattern we saw frequently was Dynamic Programming (DP). Our first work in this area was motivated by the Systems community imperative to "sip your own champagne"[5]: we wanted to implement a major part of our Overlog runtime in Overlog. To this end we built a cost-based query optimizer for Overlog named Evita Raced, itself written in Overlog as a "metacompiler" allowing for program reflection[6] [16]. Evita Raced is an Overlog program that runs in a centralized fashion without parallelism, and its DP kernel is quite similar to Greco and Zaniolo's general presentation of greedy search in extended Datalog [26]. Evita Raced makes the connection between the System R optimizer—a warhorse of the SIGMOD canon—and the compact implementation of DP in stratified Datalog. If this had been demonstrated in the 1980's during the era of extensible query optimizer architectures, it might have alleviated doubts about the utility of generalized recursion[7]. In addition to cost-based search via DP, Evita Raced also uses Overlog to implement classic Datalog optimizations and analyses, including magic sets and stratification, which are themselves based on simple transitive closures. The fact that traversals of Datalog rule/goal graphs are not described in terms of Datalog is also something of a pity, both in terms of conceptual elegance and compactness of code. But as a student of Datalog I am sypathetic to the pragmatics of exposition: it is asking a lot of one's readers to learn about recursive query optimization via metacircular logic programming[8]!

More recently, we used recursive SQL to implement the Viterbi DP algorithm for probabilistic inference on Conditional Random Fields, a technique commonly used in statistical Information Extraction [70, 69]. This connection may be more surprising to database researchers than to the machine learning community: at roughly the same time as our work on distributed systems, researchers in AI have been using logic and forward chaining to do efficient dynamic programming and search [20, 21].

Moving to a distributed setting, the main challenge that arises from Dynamic Programming is the handling of stratified aggregation (minima and maxima, etc.) across machines. I revisit this issue in Section 3.4.3.

3.2 Space, Communication and Synchrony

Much of our work has been focused on using Datalog-like languages for networking and distributed systems. This led us to a series of designs to address spatial partitioning and network communication in languages such as Overlog. Also inherent in these languages was the notion of network delay and its relationship to asynchronous evaluation.

3.2.1 Distributed State: Network Data Independence

One of our first extensions of Datalog was to model the partitioning of relations across machines in the style of parallel databases. As a matter of syntax, we required each relation to have a distinguished *location specifier* attribute. This attribute (marked with the prefix '@') had to be from the domain of network addresses in the system. Using this notation, a traditional set of network neighbor tables at each node can be represented by a global relation:

```
link(@Src, Dest, Weight)}
```

The location specifier in this declaration states that each tuple is stored at the network address of the source machine, leading to an interestingly declarative approach to networking. Location specifiers simply denote where data must be stored; communication is induced automatically (and flexibly) to achieve this specification. As one example, consider a simple version of a multicast protocol that sends messages to designated groups of recipients:

```
received(@Node, Src, Content)@async
    :- msg(@Src, GroupID, Content),
       members(@Src, GroupID, Node).
```

The simplicity here comes from two features: representing multicast group lookup as a relational join, and the bulk specification of message shipping to all recipients via a single head variable `@Node`. Note also that this rule must be asynchronous due to the communication: we cannot say when each message will be received.

As a richer example, consider the inductive rule in path-finding:

```
path(@Src, Dest)@async
    :- link(@Src, X), path(@X, Dest).
```

Here, the unification in the body involves a variable `X` that is not a location specifier in the first predicate of the rule body; as a result, communication of some sort is required to evaluate the body. That communication can happen in at least two ways: (1) link tuples can be passed to the locations in their `X` values, and resulting join tuples shipped to the values in their `Src` attribute, or (2) the rule can be rewritten to be "left-recursive":

```
path(@Src, Dest)@async
    :- path(@Src, X), link(@X, Dest).
```

In this case path tuples can be sent to the address in their `X` attribute, joined with link tuples there, and the results returned to the address in their `Src` attribute. As it happens, evaluating these two programs in the standard left-to-right order corresponds to executing two different well-known routing protocols, one used in Internet infrastructure and another common in wireless communication [48]. Raising the abstraction of point-to-point communication to join specification leads to a wide range of optimizations for rendezvous and proxying [13, 15]. This thrust is reminiscent of the shift from navigational data models to the relational model, but in the network routing domain—an increasingly attractive idea as Internet devices and subnets evolve quickly and heterogeneously [30].

We considered this notion of Network Data Independence to be one of the key contributions we were able to bring to the Networking research community. On the other hand, as I discuss in Section 3.5.2, the global database abstraction inherent in this syntax caused us difficulty in a more general setting of distributed systems.

[5] This is a more palatable (and self-congratulatory) version of the phenomenon sometimes called *dogfooding* [74].

[6] As Tyson Condie notes in his paper, the name "Evita Raced" is itself a reflection on our work: the imperfection in the name's mirroring captures the imperfect declarativity of Overlog, subsequently addressed in Dedalus.

[7] In his influential work on this topic, Guy Lohman makes an intriguing reference to logic programming, but then steps away from the idea, contrasting the deduction of "relations" from the deduction of "operators." [43]

[8] The NAIL! implementers mention using CProlog to generate rule/goal graphs for Datalog, but present imperative pseudocode for their algorithms [55].

```
q(V,R)@next :- q(V,R), !del_q(V,R).
qmin(V, min<R>) :- q(V,R).
p(V,R)@next :- q(V,R), qmin(V,R).
del_q(V,R) :- q(V,R), qmin(V,R).
```

Figure 2: A queue implementation in Dedalus. Predicate `q` represents the queue; items are being dequeued into a predicate `p`. Throughout, the variable `V` is a value being enqueued, and the variable `R` represents a position (or priority) in the queue.

3.2.2 Embracing Time, Evolving State

Prior to the development of Dedalus, we had significant problems modeling state updates in our languages. For example, Overlog provided an operational model of persistent state with updates, including SQL-like syntax for deletion, and tuple "overwrites" via primary key specifications in head predicates. But, unlike SQL, there was no notion of transactions, and issues of update visibility were left ambiguous. The lack of clear update semantics caused us ongoing frustration, and led to multiple third-party efforts at clarifying our intended operational semantics by examining our interpreter, P2, more closely than perhaps it merited [57, 53].

An example of the difficulty of state update arose early in modeling the Symphony distributed hash table [52]. In Symphony, the asymptotic structure of small-world networks is simulated in practice by constraints: each new node tries to choose $\log n$ neighbors at random, subject to the constraint that no node can have more than $2 \log n$ neighbors. A simple protocol ensures this constraint: when a node wishes to join the network, it ships link establishment requests to $\log n$ randomly chosen nodes. Each recipient responds with either an agreement (establishing a bidirectional link), or a message saying it has reached its maximum degree. The recipient logic for a successful request requires a read-only check (counting the size of its neighbor table), and two updates (adding a new edge to the neighbor table, and adding a response tuple to the network stream). The check and the updates must be done in one atomic step: if two such requests are handled in an interleaved fashion at a node with $2 \log n - 1$ neighbors, they can both pass the check and lead to a violation of the maximum-degree constraint.

One solution to this "race condition" is to have the language runtime implement a queue of request messages at each recipient, dequeuing only one request at a time into the "database" considered in a given Overlog fixpoint computation. We implemented this approach in the P2 Overlog interpreter, and a similar approach is taken in the recent Reactor programming language [22]. But the use of an operational feature outside the logic is unsatisfying, and forces any program-analysis techniques to rely on operational semantics rather than model- or proof-theoretic arguments.

It is not immediately clear how to express a queue in Datalog, and our search for a suitably declarative solution to such update problems led to the design of Dedalus. The problem can be solved via the Dedalus timestamp convention, as shown in Figure 2. The first rule of Figure 2 is the Dedalus boilerplate for "persistence" via induction rather than a storage model. It asserts persistence of the head predicate `q` across consecutive timesteps, except in the presence of tuples in a designated "deletion" predicate `del_q`. The existence of a deletion tuple in timestep N breaks the induction, and the tuple no longer appears in the predicate beginning in timestep $N + 1$.

The second rule identifies the minimum item in the queue.

The third and fourth rules together atomically dequeue the minimum item in a single timestep, placing it (ephemerally) in predicate `p`. This pair of rules illustrates how multiple updates are specified to occur atomically in Dedalus. Recall that in all Dedalus rules there is an implicit but enforced unification of all body predicates on the (omitted) timestamp attribute: this enforcement of simultaneity ensures that exactly one state of the database (one timestamp) is "visible" for deduction. Inductive rules ensure that all modifications to the state at timestep N are visible atomically in the unique successor timestep $N + 1$. In Figure 2, the insertion into the relation `p` via the third rule, and the breaking of the induction in `q` via the last and first rules occur together atomically.

This queue example provides one declarative solution to operational concurrency problems in Overlog. But many other such solutions can be expressed in Dedalus. The point is that by reifying time into the language, Dedalus allows programmers to declare their desires for concurrency and isolation in the same logic that they use for other forms of deduction. Timestamp unification and the infinite successor function serve as a monotonic, stratifying construct for treating isolation at its core: as a constraint on data dependencies. We were not the first to invent this idea (Starlog and Statelog have related constructs for time and state modification [50, 41]). But we may be the most enthusiastic proponents of its utility for reasoning about parallel and distributed programming, and the role of time in computation. I return to this topic in the Conjectures of Section 4 below.

3.3 Events and Dispatch

The crux of our work has been to apply declarative database metaphors and languages to more general-purpose programs. This required us to wrestle with program features that have not traditionally been modeled in databases, including communication and task management. Both of these features fell out relatively cleanly as we developed our systems.

3.3.1 Ephemera: Timeouts, Events, and Soft State

A central issue in distributed systems is the inability to establish the state of a remote machine. To address this limitation, distributed systems maintain evidence of the liveness of disparate nodes via periodic "heartbeat" messages and "timeouts" on those messages. This design pattern has been part of every distributed algorithm and system we have built. To support it, we needed our languages to capture the notion of physical clocks at each node.

Early on, we incorporated the notion of physical time as a relation in our languages. In Overlog we provided a built-in predicate `periodic` that could be declared to contain a set (possibly infinite) of tuples with regularly-spaced wall-clock times; the P2 runtime for Overlog would cause these tuples to "appear" in the dataflow of the query engine at the wall-clock times they contained. We would use this predicate to drive subsequent tuple derivations, for example the generation of scheduled heartbeat messages. Tuples in Overlog's `periodic` table were intended to be "ephemeral events", formed to kick off a computation once, and then be "forgotten."

Ephemeral state is often desired in network protocols, where messages are "handled" and then "consumed." A related pattern in networking is *soft state* maintenance, a loosely-coupled protocol for maintaining caches or views across machines. Consider a situation where a receiver node is caching objects (e.g., routing table entries) known at some sender node. The sender and receiver agree upon a time-to-live (TTL) for this "soft" state. The sender must try to send "refresh" messages—essentially, per-object heartbeat messages—to the receiver before the TTL expires. Upon receipt, the receiver resets the TTL of the relevant object to the agreed-upon maximum; in the absence of a refresh within the TTL, the receiver

deletes the object.

While these are common patterns in networking, their inclusion in Overlog complicated our language semantics. We included persistence properties as an aspect of Overlog's table declaration: tables could be marked as persistent, soft-state (with a fixed TTL) or as event tables (data streams). This feature introduced various subtleties for rules that mixed persistent predicates with soft state or event predicates [44]. Consider an Overlog rule for logging events: it has a persistent table in the head that represents the log, and an ephemeral stream of events in the body. But for such an Overlog rule, what does it mean when an ephemeral body tuple "disappears"? We would like the logged tuple in the head to remain, but it is no longer supported by an existing body fact.

Dedalus clears up the confusion by treating *all* tuples as ephemeral "events." Persistence of a table is ensured by the deduction of new (ephemeral) tuples at each timestep from the same tuples at the preceding timestep, as in the first rule of Figure 2 above[9]. Ambiguous "race conditions" are removed by enforcing the convention of unification on timestamp attributes. Soft state can be achieved by modifying the persistence rules to capture a wall-clock time attribute of tuples in soft-state tables (via join with a built-in wallclock-time relation), and by including a TTL-checking clause in the persistence rule as follows:

```
q(A, TTL, Birth)@next :-
   q(A, TTL, Birth),!del_q(A),
   now() - Birth < TTL.
```

In this example, `now()` returns the current wall-clock time at the local node, and can be implement as a foreign function in the spirit of LDL [12].

Having reified time into an attribute in Dedalus, any ambiguities about persistence that were inherent in Overlog are required to be explicit in a programmer's Dedalus specification. There is no need to resort to operational semantics to explain why a tuple "persists," "disappears," or is "overwritten" at a given time: each Dedalus tuple is grounded in its provenance. All issues of state mutation and persistence are captured within that logical framework.

3.3.2 Dispatch as Join: A Third Way

At the heart of any long-running service or system is a dispatching model for the management of multiple tasks. There are two foundational design patterns for task dispatch: concurrent processes and event loops. In a classic paper, Lauer and Needham demonstrate a duality between these patterns [40], but in applied settings in the last decade there has been significant back-and-forth on the performance superiority of one model or the other (e.g., [71, 68]).

We found ourselves handling this issue with a third design pattern based on dataflow. Our crawlers specified dispatch via the streaming join of an event table and a persistent system-state table. To illustrate, consider the simple example of Figure 3, which handles service `request` tuples with parameter `P`. At the timestep when a particular request arrives, it is recorded in the `pending` requests table, where it persists until it receives a matching `response`. The invocation of `service_in` is specified to happen at the same atomic timestep as the request arrival; it is evaluated by an asynchronous external function call that will eventually place its results in the relation `service_out`. Because this computation is asynchronous, the system need not "wait" for results before beginning the next timestep. This approach follows the common model of lightweight event handlers. As `service_out` results arrive

[9]An intelligent evaluation strategy for this logic should in most cases use traditional storage technology rather than re-deriving tuples each timestep.

```
pending(Id, Sender, P) :-
    request(Id, Sender, P).
pending(Id, Sender, P)@next :-
    pending(Id, Sender, P),
    !response(Id, Sender, _).
service_out(P, Out)@async :-
    request(Id, Sender, P),
    service_in(P, Out).
response(Sender, Id, O) :-
    pending(Id, Sender, P),
    service_out(P, O).
```

Figure 3: An asynchronous service

(likely in a different order than their input), they need to be looked up in the "rendezvous buffer" of `pending` requests to be routed back to the caller.

Evaluating this logic requires nothing more than the execution of a number of pipelined join algorithms such as that of Wilschut and Apers [75]. The application of pipelining joins to asynchronous dispatch was first explored in database literature for querying remote services on the Web [24, 62]. But the implication is much broader: any server dispatch loop can be coded as a few simple joins in a high-level language. This data-centric approach parallelizes beautifully (using a hash of `Id` as a location specifier), it does not incur the context-switching overhead associated with the process model, nor does it require the programmer to write explicit logic for state loops, event handling, and request-response rendezvous.

Moreover, by writing even low-level tasks such as request dispatch in logic, more of the system can enjoy the benefits of higher level reasoning, including simplicity of specification, and automatic query optimization across multiple software layers. For example, application logic that filters messages can be automatically merged into the scheduler via "selection pushdown" or magic sets rewrites, achieving something akin to kernel packet filters without any programmer intervention. Scheduling can be easily spread across multiple nodes, with task and data partitioning aligned for locality in a straightforward manner. It has yet to be demonstrated that the inner loop of a dispatcher built on a query engine can compete for performance with the best implementations of threads or events. I believe this is achievable. I also believe that optimization and parallelization opportunities that fall out from the data-centric approach can make it substantially out-perform the best thread and event packages.

3.4 Parallel and Distributed Implications

Having discussed our experience with these design patterns in some detail, I would like to step back and evaluate the implications for Datalog-like languages in parallel and distributed settings.

3.4.1 Monotonic? Embarrassing!

The Pipelined Semi-Naive (PSN) evaluation strategy [46] lies at the heart of our experience running Datalog in parallel and distributed settings. The intuition for PSN comes from our crawler experience. The basic idea in the crawlers was to act immediately on the discovery a new edge in two ways: add its endpoints to the table of nodes seen so far, and if either endpoint is new, send a request to probe that node for its neighbors in turn, producing more new edges.

It should be clear that this approach produces a correct traversal of the network graph regardless of the order of node visits. But

it is a departure from classical semi-naive evaluation, which proceeds in strict rounds corresponding to a breadth-first traversal of the graph. The need to wait for each level of the traversal to complete before moving on to the next requires undesirable (unacceptable!) coordination overhead in a distributed or parallel setting. Moreover, it is unnecessary: in monotonic programs, deductions can only "accumulate," and need never be postponed. PSN makes this idea work for general monotonic Datalog programs, avoiding redundant work via a sequencing scheme borrowed from the Urhan-Franklin Xjoin algorithm [64]. The result is that monotonic (sub)programs can proceed without any synchronization between individual deductions, and without any redundant derivations. Simply put, PSN makes monotonic logic embarrassingly parallel. This statement is significant: a large class of recursive programs—all of basic Datalog—can be parallelized without any need for coordination! This simple point is at the core of the Conjectures in Section 4 below.

As a side note, this insight appears to have eluded the MapReduce community as well, where join is necessarily a blocking operator. The issue that arises in MapReduce results from an improper conflation of a physical operation (repartitioning data) with a non-monotonic functional operation (Reduce). In Google's MapReduce framework, the only way to achieve physical network repartitioning—a key component to parallel joins—is to use a Reducer. The framework assumes Reducers need to see all their inputs at once, so it introduces a parallel barrier: no node in the system may begin Reducing until all the Map tasks are complete. This defeats the pipelining approach of Wilschut and Apers, which would otherwise perform the monotonic logic of join using physical network partitioning as a primitive. A clean implementation should be able to choose between the efficiency of pipelining and the simple fault-tolerance that comes from materialization, without tying the decision unnecessarily to the programming model.

3.4.2 Monotonic? Eventually Consistent!

A related feature we exploited in our crawlers was to accomodate "insertions" to the database via simple ongoing execution of PSN evaluation. The goal, formalized by Loo [46], was to have an *eventually consistent* semantics for the links and paths in the graph: in a quiescent database without communication failure, the set of derived data across all machines should eventually reach a consistent state. It is easy to achieve this consistency for monotonic insertion of edges into a crawler. When a new edge is added, paths in the graph radiating out from the new edge can be crawled and added to the transitive closure. When edges cease to arrive, this process eventually leads to a consistent transitive closure. This approach can be seen as a materialized view scheme for transitive closure, but in effect it is no different than the *de novo* PSN query evaluation scheme sketched above: updates simply play the role of edges that "appear very late" in the evaluation strategy.

More generally, "monotonic updates" (i.e. "appends") to a monotonic program guarantee eventual consistency. And this result can be achieved without any redundant work, by simply leaving the standard pipelined query-processing algorithm running indefinitely.

Note that in Dedalus, non-monotonic updates (deletion, overwriting) are expressed by non-monotonic programs: the negated `del` clause in a persistence rule such as the one in Figure 2. A monotonic Dedalus program can persist appends via a simpler rule:

```
p(X)@next :- p(X).
```

But modeling deletion or overwrite requires negation. Hence using Dedalus we can simply speak of whether a program is monotonic or not; this description includes the monotonicity of its state manipulation. This point informs much of the discussion in Section 4.

3.4.3 Counting Waits; Waiting Counts

If coordination is not required for monotonic programs, when is it required? The answer should be clear: at non-monotonic stratification boundaries. To establish the veracity of a negated predicate in a distributed setting, an evaluation strategy has to start "counting to 0" to determine emptiness, and wait until the distributed counting process has definitely terminated. Aggregation is the generalization of this idea.

It is tricky to compute aggregates in a distributed system that can include network delays, reordering, and failures. This problem has been the topic of significant attention in the last decade [51, 65, 35, 56], etc.) For recursive strata that span machines, the task is even trickier: no node can establish in isolation that it has fully "consumed" its input, since recursive deductions may be in flight from elsewhere.

In order to compute the outcome of an aggregate, nodes must wait and coordinate. And the logic of the next stratum of the program must wait until the coordination is complete: in general, no node may start stratum $N + 1$ until all nodes are known to have completed stratum N. In parallel programming parlance, a stratification boundary is a "global barrier." More colloquially, we can say that counting requires waiting.

This idea can be seen from the other direction as well. Coordination protocols are themselves aggregations, since they entail voting: Two-Phase Commit requires unanimous votes, Paxos consensus requires majority votes, and Byzantine protocols require a 2/3 majority. Waiting requires counting.

Combining this discussion with the previous two observations, it should be clear that there is a deep connection between non-monotonic reasoning and parallel coordination. Monotonic reasoning can be done without any coordination among nodes; non-monotonic reasoning in general requires global barriers. This line of thought suggests that non-monotonicity—a property of logic programs that can sometimes be easily identified statically—is key to understanding the limits of parallelization. I return to this point in the Conjectures section.

3.4.4 Unstratifiable? Spend Some Time.

The Dedalus state-update examples presented earlier show how the sequentiality of time can be used to capture atomic updates and persistence. Time can also be used to make sense of otherwise ambiguous, unstratifiable programs. Consider the following program, a variation on Figure 1 that toggles the emptiness of a predicate:

```
state(X)@next :- state(X), !del_state(X).
state(1) :- !state(X).
del_state(X) :- state(X)
```

The first rule is boilerplate Dedalus persistence. The last two rules toggle the emptiness of state. The second rule is clearly not stratifiable. But if we make the second rule inductive, things change:

```
state(1)@next :- !state(X).
```

In this revised program, the state table only depends negatively on itself across timesteps, never within a single timestep. The resulting program has a unique minimal model, which has `1` in the state relation every other timestep.[10]

[10]Technically, the minimal model here is infinitely periodic, but with a minimal number of distinguished states (two). Capturing this point requires a slightly modified notion of safety and minimality [41].

If we expand the syntax of this Dedalus program with all the omitted attributes and clauses, we can see that it provides what Ross defined as *Universal Constraint Stratification* [63] by virtue of the semantics of the successor function used in time. Universal Constraint Stratification is a technique to establish the acyclicity of individual derivations by manipulating constraints on the semantics of functions in a program. In this program, the successor function ensures that all derivations of state produce monotonically increasing timesteps, and hence no derivation can cycle through negation.

Many programs we have written—including the queue example above—are meaningful only because time flies like an arrow: monotonically forward[11]. Again, this temporal construct hints at a deeper point that I will expand upon in Section 4.3: in some cases the meaning of a program can only be established by "spending time."

3.5 Gripes and Problems

Datalog-based languages have enabled my students to be very productive coders. That said, it is not the case that they have always been happy with the languages at hand. Here I mention some of the common complaints, with an eye toward improving them in Bloom.

3.5.1 Syntax and Encapsulation

The first frustration programmers have with Datalog is the difficulty of unifying predicates "by eyeball," especially for predicates of high arity. Off-by-one errors in variable positions are easy to make, hard to recognize, and harder to debug. Code becomes burdensome to read and understand because of the effort involved in visually matching the index of multiple variables in multiple lists.

Datalog often requires redundant code. Disjunction, in our Datalog variants, involves writing multiple rules with the same head predicate. Conditional logic is worse. Consider the following example comparing a view of the number of "yes" votes to a view of the total number of votes:

```
outcome('succeed')
    :- yes(X), total(Y), X > Y/2.
outcome('fail')
    :- yes(X), total(Y), X < Y/2.
outcome('tie')
    :- yes(X), total(Y), X = Y/2.
```

Not only is this code irritatingly chatty, but the different "branches" of this conditional expression are independent, and need not appear near each other in the program text. Only programmer discipline can ensure that such branches are easy to read and maintain over the lifetime of a piece of software.

Finally, Datalog offers none of the common constructs for modularity: variable scoping, interface specification, encapsulation, etc. The absence of these constructs often leads to disorganized, redundant code that is hard to read and evolve.

Many of these gripes are addressable with syntactic sugar, and some Datalog variants offer help [5]. One observation then is that such sugar is very important in practice, and the academic syntax of Datalog has not improved its odds of adoption.

3.5.2 The Myth of the Global Database

The routing examples discussed above illustrate how communication can be induced via a partitioned global database. The metaphor of a global database becomes problematic when we consider programs in which semantics under failure are important. In practice, individual participating machines may become disconnected from and reconnected to the network over time, taking their partitions with them. Moreover time (and hence "state") may evolve at different rates on different nodes. Exposing a unified global database abstraction to the programmer is therefore a lie. In the context of routing it is a little white lie, because computation of the true "best" paths in the network at any time is not practically achievable in any language, and not considered important to the task. But the lie can cause trouble in cases where assumptions about global state affect correctness. False abstractions in distributed protocols have historically been quite problematic [76]. In our recent Overlog code we have rarely written rules with distributed joins in the body, in part because it seems like bad programming style, and in part because we have been focused on distributed systems protocols where message failure handling needs to be explicit in the logic. In Dedalus such rules are forbidden.

Note that the myth of the global database can be "made true" via additional code. We have implemented distributed consensus protocols such as Two-Phase Commit and Paxos that can ensure consistent, network-global updates. These protocols slow down a distributed system substantially, but in cases where it is important, distributed joins can be made "real" by incorporating these protocols [3]. On top of these protocols, an abstraction of a consistent global database can be made true (though unavailable in the face of network partitions, as pointed out in Brewer's CAP theorem.)

Given that distributed state semantics are fundamental to parallel programming, it seems important for the language syntax to require programmers to address it, and the language parser to provide built-in reasoning about the use of different storage types. For example, the Dedalus boilerplate for persistence can be "sugared" via a persistence modifier to a schema declaration, as in Overlog. Similarly, the rules for a persistent distributed table protected via two-phase commit updates could be sugared via a "globally consistent" schema modifier. While these may seem like syntactic sugar, from a programmer's perspective these are critical metaphors: the choice of the proper storage semantics can determine the meaning and efficiency of a distributed system. Meanwhile, the fact that all these options compile down to Dedalus suggests that program analysis can be done to capture the stated meaning of the program and reflect it to the user. In the other direction, program analysis can in some cases relax the user's choice of consistency models without changing program semantics.

4. CONJECTURES

> *In placid hours well-pleased we dream*
> *Of many a brave unbodied scheme.*
> — Herman Melville

The experiences described above are offered as lessons of construction. But a PODS audience may prefer the construction of more questions. Are there larger theoretical issues that arise here, and can they inform practice in a fundamental way?

I like to think the answer is "yes," though I am sensitive to both the hubris and irresponsibility of making up problems for other people. As a start, I offer some conjectures that have arisen from discussion in my group. Perhaps the wider PODS audience will find aspects of them amenable to formalization, and worthy of deeper investigation.

4.1 Parallelism, Distribution and Monotonicity

Earlier I asserted that basic Datalog programs—monotonic programs without negation or aggregation—can be implemented in

[11] Groucho Marx's corollary comes to mind: "Time flies like an arrow. Fruit flies like a banana."

an embarrassingly parallel or eventually consistent manner without need for any coordination. As a matter of conjecture, it seems that the other direction should hold as well:

CONJECTURE 1. Consistency And Logical Monotonicity (CALM). *A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.*

The "coordination-free" property is key to the CALM conjecture. Clearly one can achieve eventual consistency via a coordination mechanism such as two-phase commit or Paxos. But this "instantaneous consistency" approach violates the spirit of eventual-consistency methods, which typically proceed without coordination and still produce consistent states in periods of quiescence.

I have yet to argue one direction of this conjecture: that a non-monotonic program can have no eventually consistent implementation without coordination. Consider the case of a two-stratum non-monotonic program and some eventually consistent implementation. Any node in the system can begin evaluating the second stratum only when it can prove it has received "everything" in the first stratum's predicates. For global consistency, "everything" in this context means any data that is interdependent with what any other node has received. If any of that data resides on remote nodes, distributed coordination is required.

A proper treatment of this conjecture requires crisp definitions of eventual consistency, coordination, and relevant data dependencies. In particular, trivially partitionable programs with no cross-node interdependencies need to be treated as a special (easy) case. But I suspect the conjecture holds for most practical cases of interest in distributed and parallel computing.

It is worth recalling here Vardi's well-known result that (monotonic) Datalog can be evaluated in time polynomial in the size of the database [66]. If the conjecture above is true, then the expressive power of "eventually-consistent" implementations is similarly bounded, where the "database" includes all data and messages introduced up to the time of quiescence.

This conjecture, if true, would have both analytic and constructive uses. On the analytic front, existing systems that offer eventual consistency can be modeled in Datalog and checked for monotonicity. In many cases the core logic will be trivially monotonic, but with special-purpose escapes into non-monotonicity that should either be "protected" by coordination, or managed via compensatory exception handling (Helland and Campbell's "apologies." [29].) As a classic example, general ledger entries (debits and credits) accumulate monotonically, but account balance computation is non-monotonic; Amazon uses a (mostly) eventually-consistent implementation for this pattern in their shopping carts [18]. An interesting direction here is to incorporate exception-handling logic into the program analysis: ideally, a program with proper exception handlers can be shown to be monotonic even though it would not b monotonic in the absence of the handlers. Even more interesting is the prospect of automatically generating (perhaps conservative) exception handlers to enforce a provable notion of monotonicity.

On the constructive front, implementations in Datalog-style languages should be amenable to (semi-)automatic rewriting techniques that expose further monotonicity, expanding the base of highly-available, eventually-consistent functionality. As an example, a predicate testing for non-negative account balances can be evaluated without coordination for accounts that see only credit entries, since the predicate will eventually transition to truth at each node as information propagates, and will never fail thereafter. This example is simplistic, but the idea can be used in a more fine-grained manner to enable sets or time periods of safe operation (e.g., a certain number of "small" debits) to run in an eventually consistent fashion, only enforcing barriers as special-case logic near monotonicity thresholds on predicates (e.g., when an account balance is too low to accomodate worst-case scenarios of in-flight debits). It would be interesting to enable program annotations, in the spirit of Universal Constraint Stratification [63], that would allow program rewrites to relax initially non-monotonic kernels in this fashion.

Finally, confirmation of this conjecture would shed some much-needed light on heated discussions of the day regarding the utility or necessity of non-transactional but eventually consistent systems, including the so-called "NoSQL" movement. It is sorely tempting to underscore this conjecture with the slogan "NoSQL is Datalog." But my student Neil Conway views this idea as pure mischief-making, so I have refrained from including the slogan here.

4.2 Asynchrony, Traces, and Trace Equivalence

Expanding on the previous point, consider asynchronous rules, which introduce non-determinism into Dedalus timestamps and hence Dedalus semantics. It is natural to ask under what conditions this explicit non-determinism affects program outcomes, and how.

We can say that the timestamps in asynchronous head predicates capture possible *traces* of a program: each trace is an assignment of timestamps that describes a non-deterministic "run" of an evaluation strategy. We can define *trace equivalence* with respect to a given program: two traces can be considered equivalent if they lead to the same "final" outcome of the database modulo timestamp attributes. If all traces of a program can be shown to be equivalent in this sense, we have demonstrated the Church-Rosser confluence property. In cases where this property does not hold, other notions of equivalence classes may be of interest. Serializability theory provides a model: we might try to prove that every trace of a program is in an equivalence class with some "good" trace.

The theory of distributed systems has developed various techniques to discuss the possible traces of a program. The seminal work is Lamport's paper on "Time, Clocks and the Ordering of Events" [38], which lays out the notion of causal ordering in time that requires logical clocks to respect a *happens-before* relation. Based on these observations, he shows that multiple independent clocks (at different distributed nodes) can be made to respect the happens-before relation of each individual node. Coordination protocols have been developed to enforce this kind of synchrony, and related issues involving data consistency. We have coded some of these protocols in Dedalus and its predecessor languages [3], and they can be incorporated into programs to constrain the class of traces that can be produced.

Classical PODC work on casuality tends to assume black-box state machines at the communication endpoints. With Dedalus programs at the endpoints, we can extract logical data dependency and provenance properties of the programs, including tests for various forms of stratifiability. Can the combination of causality analysis and logic-programming tests enrich our understanding of distributed systems, and perhaps admit new program-specific cleverness in coordination?

As an extreme example, suppose we ignore causality entirely, and allow asynchronous Dedalus rules to send messages into the past. Are temporal paradoxes—the absence of a unique minimal model—an inevitable result? On this front, I have a simple conjecture:

CONJECTURE 2. Causality Required Only for Non-monotonicity (CRON). *Program semantics require causal message ordering if and only if the messages participate in non-monotonic derivations.*

Said differently, temporal paradoxes arise from messages sent into the past if and only if the messages have non-monotonic implica-

tions.

This conjecture follows the intuition of the CALM Conjecture. Purely monotonic logic does not depend on message ordering, but if the facts being "sent into the past" are part of a non-monotonic cycle of deduction, the program lacks a unique minimal model: it will either admit multiple possible worlds, or none.

The idea of sending messages into the past may seem esoteric, but it arises in practical techniques like recovery. If a node fails and is restarted at time T, it may reuse results from logs that recorded the (partial) output of a run of the same logic from some earlier time $S < T$. In effect the derived messages can "appear" at time S, and be used in the redo execution beginning at T without causing problems. Speculative execution strategies have a similar flavor, a point I return to in Section 4.4.

Exploiting the monotonic case may be constructive. It should be possible to relax the scheduling and spatial partitioning of programs—allow for more asynchrony via a broader class of traces—by examining the program logic, and enforcing causal orderings only to control non-monotonic reasoning. This marriage of PODS-style program analysis and PODC-style causality analysis has many attractions.

4.3 Coordination Complexity, Time and Fate

In recent years, the exponentiation of Moore's Law has brought the cost of computational units so low that to infrastructure services they seem almost free. For example, O'Malley and Murthy at Yahoo! reported sorting a petabyte of data with Hadoop using a cluster of 3,800 machines each with 8 processor cores, 4 disks, and 8GB of RAM each [58]. That means each core was responsible for sorting only about 32 MB (just 1/64th of their available share of RAM!), while 3799/3800 of the petabyte was passed through the cluster interconnection network during repartitioning. In rough terms, they maximized parallelism while ignoring resource utilization. But if computation and communication are nearly free in practice, what kind of complexity model captures the practical constraints of modern datacenters?

This anecdote involves an embarrassingly parallel, monotonic binning algorithm, with a focus on bulk data throughput rather than latency of individual interactions. By contrast, non-monotonic stratified programs require latency-sensitive distributed coordination to proceed from one parallel monotonic stratum to the next. Non-monotonic stratum boundaries are global barriers: in general, no node can proceed until all tasks in the lower stratum are guaranteed to be finished. This restriction means that the slowest-performing task in the cluster—the "weakest link"— slows all nodes down to its speed. Dynamic load balancing and reassignment can mitigate the worst-case performance of the weakest link, but even with those techniques in place, coordination is the key remaining bottleneck in a world of free computation [8].

In this worldview, the running time of a logic program might be best measured by the number of strata it must proceed through sequentially; call this the Coordination Complexity of a program[12]. This notion differs from other recent models of parallel complexity for MapReduce proposed by Afrati and Ullman [1] and Karloff, et al. [37], which still concern themselves in large part with measuring communication and computation. It resembles a simplified ("embarrassingly" simplified?) form of Valiant's Bulk Synchronous-Parallel (BSP) model, with the weights for communication and computation time set to zero. Results for BSP tend to involve complicated analyses of communication and computation metrics that are treated as irrelevant here, with good reason. First, the core operations for bottom-up Dedalus evaluation (join, aggregation) typically require all-to-all communication that only varies between $\frac{1}{2}$ and 1 for any non-trivial degree of parallelism[13]. Second, at scale, the practical running time of the slowest node in the cluster is often governed less by computational complexity than by non-deterministic effects of the environment (heterogeneous machines and workloads, machine failures, software misconfiguration and bugs, etc.)

Conveniently, a narrow focus on Coordination Complexity fits nicely with logic programming techniques, particularly if the previous conjectures hold: i.e., if coordination is required precisely to manage non-monotonic boundaries. In that case, the Coordination Complexity of a stratified program is the maximum stratum number in the program, which can be analyzed syntactically. In the more general case of locally stratified [61] or universally constraint-stratified programs [63], the program's rule-goal graph may have cycles through negation or aggregation, which in practice might be traversed many times. The coordination complexity in these cases depends not only on the rule syntax but also on the database instance (in the case of local stratification) and on the semantics of monotonic predicates and aggregations in the program (in universal constraint stratification).

As noted above, many natural Dedalus programs are not stratified, but are instead universally constraint-stratified by the monotonicity of timestamps. In those cases, the number of times around the non-monotonic loops corresponds to the number of Dedalus timesteps needed to complete the computation. In essence, Dedalus timesteps become units for measuring the complexity of a parallel algorithm.

This idea is intuitively appealing in the following sense. Recall that a Dedalus timestep results in an atomic batch of inductions, corresponding to traditional "state changes." To lower the number of Dedalus timesteps for a program, one needs to find a way to batch together more state modifications within a single timestep—i.e., shift some rules from inductive to deductive (by removing the `@next` suffix). If a program is expressed to use a minimal number of timesteps, it has reached its inherent limit on "batching up" state changes—or conversely, the program accurately captures the inherent need for sequentiality of its state modifications.

This argument leads to our next conjecture:

CONJECTURE 3. Dedalus Time $\Leftrightarrow$ Coordination Complexity. *The minimum number of Dedalus timesteps required to evaluate a program on a given input data set is equivalent to the program's Coordination Complexity.*

The equivalence posited here is within a constant factor of the minimum number of sequential coordination steps required, which accounts for multiple deductive strata within a single timestep. The stratification depth per timestep is bounded by the program's length, and hence is a constant with respect to data complexity (the appropriate measure for analyzing a specific program [66]).[14]

Clearly one can do a poor job writing an algorithm in Dedalus, e.g., by overuse of the `@next` modifier. So when are timesteps

[12] In some algorithms it may be worth refining this further to capture the fraction of nodes involved in each coordination step; this two-dimensional "depth" and "breadth" might be called a coordination *surface* or lattice.

[13] The exception here is cases where communication can be "colored away" entirely, due to repeated partitioning by functionally dependent keys [28].

[14] David Maier notes that it should be possible convert a fixed number of timestamps into data and account for these timestamps within a single Dedalus timestep, in the manner of loop unrolling. This conversion of time into space may require refining the complexity measure proposed here.

truly required? We can distinguish two cases. The first is when the removal of an `@next` suffix changes the meaning (i.e. the minimal model) of the program. The second is when the removal causes a form of non-monotonicity that leaves the program with no unique minimal model: either a proliferation of minimal models, or no models because a contradiction requires some variable to take on multiple values "at once." The examples we have seen for the first of these cases seem trivial, in the spirit of Figure 1: a finite number of possible states are infinitely repeated. Such programs can be rewritten without an infinite successor relation: the finite set of possible states can be distinguished via data-oriented predicates on finite domains, rather than contemporaneity in unbounded (but cyclic) time[15]. This leads us to the following more aggressive conjecture:

CONJECTURE 4. Fateful Time. *Any Dedalus program P can be rewritten into an equivalent* temporally-minimized *program P' such that each inductive or asynchronous rule of P' is* necessary*: converting that rule to a deductive rule would result in a program with no unique minimal model.*

I call this the "Fateful Time" conjecture because it argues that *the inherent purpose of time is to seal fate*. If a program's semantics inherently rely upon time as an unbounded source of monotonicity, then the requirement for simultaneity in unification resolves multiple concurrent possible worlds into a series of irrevocable individual worlds, one at each timestep. If the conjecture holds, then any other use of temporal induction is literally a waste of time.[16].

This conjecture is an appealing counterpart to CRON. Time *does* matter, exactly in those cases where ignoring it would result in logically ambiguous fate[17].

[15]This observation, by my student William Marczak, is eerily reminiscent of the philosophy underlying Jorge Luis Borges' stories of circular ruins, weary immortals, infinite libraries and labyrinths. In his "New Refutation of Time," Borges presents a temporal extension of the idealism of Berkeley and Hume (which denies the existence of matter) based on the following argument: "The denial of time involves two negations: the negation of the succession of the terms of a series, negation of the synchronism of the terms in two different series." The similarity to a rewriting of Dedalus' successors and timestamp unification is striking. More allegorically, Borges puts it this way: "I suspect, however, that the number of circumstantial variants is not infinite: we can postulate, in the mind of an individual (or of two individuals who do not know of each other but in whom the same process works), two identical moments. Once this identity is postulated, one may ask: Are not these identical moments the same? Is not one single repeated term sufficient to break down and confuse the series of time? Do not the fervent readers who surrender themselves to Shakespeare become, literally, Shakespeare?" [9]

[16]Temporally-minimized Dedalus programs might be called *daidala*, Homer's term for finely crafted basic objects: "The 'daidala' in Homer seem to possess mysterious powers. They are luminous—they reveal the reality that they represent." [60]

[17]In his Refutation, Borges concludes with a very similar inescapable association of time and fate: "*And yet, and yet...* Denying temporal succession, denying the self, denying the astronomical universe are apparent desperations and secret consolations. Our destiny ... is frightful because it is irreversible and iron-clad. Time is the substance I am made of. Time is a river which sweeps me along, but I am the river; it is a tiger which destroys me, but I am the tiger; it is a fire which consumes me, but I am the fire. The world, unfortunately, is real; I, unfortunately, am Borges." [9] To close the cycle here, note that Daedalus was the architect of the Labyrinth of Crete; labyrinths are a signature metaphor in Borges' writing.

4.4 Approximate and Randomized Algorithms

Given the cost of coordination at stratum boundaries, it is tempting to try and go further: let the evaluation of a Dedalus program press ahead without waiting for the completion of a stratum or timestep. In some cases this trick can be done safely: for example, temporal aggregates such as `count` and `min` can provide "early returns" in the spirit of online aggregation [32], and range predicates on the results of those aggregates can sometimes be evaluated correctly in parallel with the computation of the final aggregate result [77]. These cases exploit monotonicity of predicates on monotonic aggregates, and are in the spirit of the CALM conjecture above.

But what about approximable but non-monotonic aggregates, such as averages, which can produce early estimates that oscillate non-monotonically, but provide probabilistic confidence intervals? If predicates on the result of those aggregates are "acted upon" probabilistically, in parallel with the completion of the lower stratum, what happens to the program outcome?

Two execution strategies come to mind, based on "optimistically" allowing higher strata to proceed on the basis of tentative results in lower strata. The first takes the tentative results and acts upon them directly to compute a fixpoint, which may or may not be the same as the minimal model of the program. The challenge then is to characterize the distribution of possible worlds that can arise from such evaluations, provide a meaningful probabilistic interpretation on the outcomes of the computation, and perhaps provide execution strategies to ensure that an individual outcome has high likelihood. It would be interesting to understand how this relates to more traditional approximation algorithms and complexity, especially with respect to parallel computation.

A second strategy is to ensure the correct minimal model by "rolling back" any deductions based on false assumptions using view maintenance techniques [11]. Here the challenge is not to characterize the answer, but to produce it efficiently by making good guesses. This requires characterizing the expected utility of a given optimistic decision, both in terms of its likely benefit if performance is correct, and its likely recomputation cost if incorrect. This is in the spirit of speculation techniques that are common currency in the Architecture and Systems communities, but with the benefit of program analyses provided by logic. It also seems feasible to synthesize logic here, including both speculative moves, and compensating actions for incorrect speculations.

Practically, these approaches are important for getting around fundamental latencies in communication. From a theoretical perspective, speculation on non-monotonic boundaries seems to be the natural path to bring randomized algorithms into the evaluation of logic: the previous conjectures suggest that there is no interesting non-determinism in monotonic programs, so the power of randomized execution seems to reside at non-monotonic boundaries. It would be interesting to understand how to bridge this idea to the complexity structures known for randomized algorithms.

This line of thinking is not as well developed as the earlier discussion, so I close this discussion without stating a particular conjecture.

5. CARPE DIEM

Gather ye rosebuds while ye may
Old Time is still a-flying,
And this same flower that smiles to-day
To-morrow will be dying.
— Robert Herrick

Like most invited papers, I would have rejected this one had I been asked to review it[18]. Its claims are imprecise and unsubstantiated, relying on anecdotes and intuition. It contains an unseemly number of references to the author's prior work. It is too long, and was submitted after the conference deadline. And as a paper that treads outside the author's area of expertise, it also undoubtedly overlooks important earlier work by others. For this last flaw in particular I extend sincere apologies, and an earnest request to be schooled regarding my omissions and errors.

However, my chief ambition in writing this paper was not to present a scholarly survey. It was instead to underscore—in as urgent and ambitious terms as possible—the current opportunity for database theory to have broad impact. Under most circumstances it is very hard to change the way people program computers [54]. But as noted by Hennessy and others, programming is entering an unusually dark period, with dire implications for computer science in general. "Data folk" seem to have one of the best sources of light: we have years of success parallelizing SQL, we have the common culture of MapReduce as a bridge to colleagues, and we have the well-tended garden of declarative logic languages to transplant into practice.

Circumstance has presented a rare opportunity—call it an imperative—for the PODS community to take its place in the sun, and help create a new environment for parallel and distributed computation to flourish. I hope that the discussion in this paper will encourage more work in that direction.

6. ACKNOWLEDGMENTS

I am indebted to mentors who read and commented on early drafts of this paper: David Maier, Jeff Naughton, Christos Papadimitriou, Raghu Ramakrishnan, Jeff Ullman, and Moshe Vardi. I am grateful for their time, logic, and bracing candor.

The conjectures in the paper grew out of relatively recent discussions with students on the BOOM project: Peter Alvaro, Tyson Condie, Neil Conway, William Marczak and Rusty Sears. I am unreasonably lucky and genuinely grateful to be able to work with a group of students this strong.

Much of the fun of systems work comes from the group effort involved. In certain cases in the text I was able to give specific references to ideas from individuals, but many of the experiences I described arose in joint work with a wonderful group of students and colleagues, especially on the P2 [45], DSN [14] and BOOM [2] projects. The following list is the best accounting I can give for these experiences. In alphabetical order, credit is due to: Peter Alvaro, Ashima Atul, Ras Bodik, Kuang Chen, Owen Cooper, David Chu, Tyson Condie, Neil Conway, Khaled Elmeleegy, Stanislav Funiak, Minos Garofalakis, David Gay, Carlos Guestrin, Thibaud Hottelier, Ryan Huebsch, Philip Levis, Boon Thau Loo, David Maier Petros Maniatis, Lucian Popa, Raghu Ramakrishnan, Timothy Roscoe, Rusty Sears, Scott Shenker, Ion Stoica, and Arsalan Tavakoli.

7. REFERENCES

[1] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[2] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. Boom: Data-centric programming in the datacenter. In *Eurosys*, April 2010.

[3] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: consensus in a logic language. *SIGOPS Oper. Syst. Rev.*, 43(4):25–30, 2010.

[4] P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[5] M. Aref. Datalog for enterprise applications: from industrial applications to research. In *Datalog 2.0 Workshop*, 2010. `http://www.datalog20.org/slides/aref.pdf`.

[6] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. S. Pillai. Meld: A declarative approach to programming ensembles. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2007.

[7] A. Atul. Compact implementation of distributed inference algorithms for network. Master's thesis, EECS Department, University of California, Berkeley, Mar 2009.

[8] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[9] J. L. Borges. A new refutation of time. In D. A. Yates and J. E. Irby, editors, *Labyrinths: Selected Stories and Other Writings*. New Directions Publishing, 1964. Translation: James E. Irby.

[10] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[11] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. In *VLDB*, 2009.

[12] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.

[13] D. Chu and J. Hellerstein. Automating rendezvous and proxy selection in sensor networks. In *Eighth International Conference on Information Processing in Sensor Networks (IPSN)*, 2009.

[14] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys*, pages 175–188, 2007.

[15] D. C. Chu. *Building and Optimizing Declarative Networked Systems*. PhD thesis, EECS Department, University of California, Berkeley, Jun 2009.

[16] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *Proc. VLDB Endow.*, 1(1):1153–1165, 2008.

[17] O. Cooper. TelegraphCQ: From streaming database to information crawler. Master's thesis, EECS Department, University of California, Berkeley, 2004.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007.

[19] N. Durgin, J. C. Mitchell, and D. Pavlovic. A compositional logic for proving security properties of protocols. *Journal of Computer Security*, 11(4):677–721, 2003.

[20] J. Eisner, E. Goldlust, and N. A. Smith. Dyna: a declarative language for implementing dynamic programs. In *Proc.*

[18] My benchmark of quality for invited papers is Papadimitriou's "Database Metatheory." [59] Although I looked to it for inspiration, I had no pretensions to its breadth or charm. In deference, I refrain from matching Papadimitriou's two dozen footnotes. In tribute, however, I adopt his spirit of proud classical reference with something from my own tradition: 18 footnotes, corresponding to the auspicious *gematria* of חי (life).

ACL, 2004.

[21] P. F. Felzenszwalb and D. A. McAllester. The generalized A* architecture. *J. Artif. Intell. Res. (JAIR)*, 29:153–190, 2007.

[22] J. Field, M.-C. V. Marinescu, and C. Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theor. Comput. Sci.*, 410(2-3), 2009.

[23] E. Fox. *The Five Books of Moses: A New Translation With Introductions, Commentary and Notes*. Schocken Books, 1995.

[24] R. Goldman and J. Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *In SIGMOD*, pages 285–296, 2000.

[25] J. Gray. What next?: A dozen information-technology research goals. *J. ACM*, 50(1):41–57, 2003.

[26] S. Greco and C. Zaniolo. Greedy algorithms in datalog with choice and negation, 1998.

[27] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder. Codequest: querying source code with datalog. In *OOPSLA*, 2005.

[28] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *VLDB*, 1995.

[29] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[30] J. M. Hellerstein. Toward network data independence. *SIGMOD Rec.*, 32(3):34–40, 2003.

[31] J. M. Hellerstein. Datalog redux: Experience and conjectures. In *PODS*, 2010.

[32] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[33] J. M. Hellerstein and M. Stonebraker, editors. *Readings in Database Systems, Third Edition*. Morgan Kaufmann, Mar. 1998.

[34] J. Hennessy and D. Patterson. A conversation with John Hennessy and David Patterson, 2006. `http://queue.acm.org/detail.cfm?id=1189286`.

[35] N. Jain, M. Dahlin, Y. Zhang, D. Kit, P. Mahajan, and P. Yalagandula. Star: Self-tuning aggregation for scalable monitoring. In *VLDB*, 2007.

[36] T. Jim. Sd3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, page 106, Washington, DC, USA, 2001. IEEE Computer Society.

[37] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Symposium on Discrete Algorithms (SODA)*, 2010.

[38] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[39] B. Lampson. Getting computers to understand. *J. ACM*, 50(1):70–72, 2003.

[40] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[41] G. Lausen, B. Ludascher, and W. May. On active deductive databases: The statelog approach. In *In Transactions and Change in Logic Databases*, pages 69–106. Springer-Verlag, 1998.

[42] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *PLDI*, 2004.

[43] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. *SIGMOD Rec.*, 17(3):18–27, 1988.

[44] B. T. Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, University of California, Berkeley, Dec. 2006.

[45] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, 2009.

[46] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD Conference*, pages 97–108, 2006.

[47] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.

[48] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *VLDB*, 2004.

[49] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *SIGCOMM*, pages 289–300, 2005.

[50] L. Lu and J. G. Cleary. An operational semantics of starlog. In *PPDP*, pages 294–310, 1999.

[51] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[52] G. S. Manku, M. Bawa, P. Raghavan, and V. Inc. Symphony: Distributed hashing in a small world. In *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, pages 127–140, 2003.

[53] Y. Mao. On the declarativity of declarative networking. *SIGOPS Oper. Syst. Rev.*, 43(4), 2010.

[54] E. Meijer. Confessions of a used programming language salesman. In *OOPSLA*, pages 677–694, 2007.

[55] K. A. Morris. An algorithm for ordering subgoals in nail? In *PODS*, 1988.

[56] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. *TOSN*, 4(2), 2008.

[57] J. A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In *PADL*, 2009.

[58] O. O'Malley and A. Murthy. Hadoop sorts a petabyte in 16.25 hours and a terabyte in 62 seconds, 2009. `http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html`.

[59] C. H. Papadimitriou. Database metatheory: asking the big queries. *SIGACT News*, 26(3), 1995.

[60] A. Perez-Gomez. The myth of daedalus. *Architectural Association Files*, 10:49–52, 1985.

[61] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[62] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[63] K. A. Ross. A syntactic stratification condition using constraints. In *ILPS*, 1994.

[64] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelines join operator. *Bulletin of the Technical Committee on Data Engineering*, 23(2), 2000.

[65] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[66] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

[67] V. Vianu and D. V. Gucht. Computationally complete relational query languages. In *Encyclopedia of Database Systems*, pages 406–411. Springer, 2009.

[68] J. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. A. Brewer. Capriccio: scalable threads for internet services. In *SOSP*, pages 268–281, 2003.

[69] D. Z. Wang, M. J. Franklin, M. Garofalakis, and J. M. Hellerstein. Querying probabilistic declarative information extraction. In *VLDB*, 2010. To appear.

[70] D. Z. Wang, E. Michelakis, M. J. Franklin, M. Garofalakis, and J. M. Hellerstein. Probabilistic declarative information extraction. In *ICDE*, 2010.

[71] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[72] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, 2004.

[73] W. White, B. Sowell, J. Gehrke, and A. Demers. Declarative processing for computer games. In *ACM SIGGRAPH Sandbox Symposium*, 2008.

[74] Wikipedia. Eating one's own dog food, 2010. `http://en.wikipedia.org/wiki/Eating_one's_own_dog_food`.

[75] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, 1991.

[76] A. Wollrath, G. Wyant, J. Waldo, and S. C. Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories, 1994.

[77] C. Zaniolo and H. Wang. Logic-based user-defined aggregates for the next generation of database systems. In K. Apt, V. Marek, M. Truszczynski, and D.S.Warren, editors, *The Logic Programming Paradigm: Current Trends and Future Directions*. Springer Verlag, 1999.

[78] W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified declarative platform for secure netwoked information systems. In *ICDE*, 2009.