Large-Scale Computation Not at the Cost of Expressiveness

Sangjin Han and Sylvia Ratnasamy
University of California, Berkeley

Abstract

We present Celias, a new concurrent programming model for data-intensive scalable computing. Celias supports many virtues commonly found in existing distributed programming frameworks, such as elastic scaling and fault tolerance, without sacrificing expressiveness. The key design idea of Celias is the concept of a microtask, as a scalable, fault-tolerant, and completely data-driven unit of computation. By combining Tuplespace and microtasks, Celias provides an intuitive yet powerful programming abstraction for large and complex problems.

1 Introduction

The explosive growth in "big data" applications fueled research on scalable, data-centric computing systems, with MapReduce [18] as the most notable example. The allure of MapReduce lies largely in its highly abstracted programming model which shifts the responsibility for scalable and fault-tolerant distribution of computation from the programmer to the execution engine, thus allowing a programmer to focus on problem solving instead of the inherently complex details of distributing computation.

Since a single MapReduce job simply does not fit many classes of real-world problems, recent research projects have proposed solutions based on iteration over MapReduce jobs [10, 21] or structured representations of embarrassingly parallel operations (e.g., map, filter, join, and group-by) [5, 8, 14, 15, 19, 25, 30, 37, 43, 46]. All these systems are designed for an application model based on *bulk transformation over immutable collections* as noted by Zaharia et al [45]. However, for certain classes of problems – especially where sparse operations over mutable data sets gradually occur based on fine-grained, inter-data dependencies – the coarse-grained building blocks of the above systems fundamentally limit their efficiency at best or applicability at worst [27, 32, 36].

As an illustrative example, consider breadth-first search on a large graph with a series of MapReduce jobs. Each iteration takes the entire graph and the nodes in the current search horizon (queue) to compute the next horizon, and this step repeats until the horizon is empty. This is unacceptably inefficient, since each step processes the entire graph although the "useful" computation only happens at a small fraction of the graph. While there are frameworks that efficiently support incremental computation for graph algorithms [29, 30], can we devise a general programming model for non-graph algorithms?

In this work, we thus aim to answer the following question: *Can we design a programming model that is powerful enough to represent algorithms with unpredictable, fine-grained data access patterns and control flow, while retaining the virtues of MapReduce, such as simplicity, automatic scaling, and fault tolerance?* We revisit Linda [24] as a promising starting point, as Linda offers the advantages that we seek, such as the fine-grained interface to mutable global state and expressivity with dynamic control flow. In order to solve a large, complex problem in a distributed manner, Linda follows the oft-cited blackboard metaphor [17].

> *"Imagine a group of human specialists seated next to a large blackboard. The specialists are working cooperatively to solve a problem, using the blackboard as the workplace for developing the solution.*
>
> *Problem solving begins when the problem and initial data are written onto the blackboard. The specialists watch the blackboard, looking for an opportunity to apply their expertise to the developing solution. When a specialist finds sufficient information to make a contribution, she records the contribution on the blackboard, hopefully enabling other specialists to apply their expertise. This process of adding contributions to the blackboard continues until the problem has been solved."*

Linda utilizes Tuplespace, a logically-shared associative memory, as the "blackboard" to allow distributed processes to collaborate. While the Linda model greatly simplifies many complex aspects of distributed programming, it is not, per se, an alternative to MapReduce since it is neither automatically scalable nor inherently fault tolerant, due to Linda's process-oriented abstraction.

In this paper, we present the early design aspects of the Celias programming model. Celias introduces the concept of a *microtask* as the basic unit of computation. Microtasks are triggered by data availability in the tuplespace, run in their entirety as a transaction, and are characterized by explicit input/output and side-effect free computation. These properties give microtasks three important benefits: task mobility, automatic scalability, and fault tolerance.

2 Reviving Tuplespace for Tomorrow

We briefly review Linda/Tuplespace (§2.1), discuss issues that limit the applicability of Linda to datacenter contexts (§2.2), and finally introduce our Celias programming model that aims to overcome these issues (§2.3).

2.1 Tuplespace and Linda

In contrast to message-passing schemes, Linda processes, distributed across a network, interact only with the tuplespace [12]. The tuplespace stores a collection of tuples and offers the logical abstraction of a global associative memory shared by all processes. Processes store/retrieve tuples to/from the tuplespace in order to indirectly communicate with other processes.

The tuplespace stores a collection of ordered sequences, called tuples. A tuple contains typed attributes, each of which is a primitive (integer, string, etc.) or composite object. Tuples can be used as either shared data or structured messages between processes, depending on the semantics imposed by the application. All tuplespace operations are atomic, and partial updates on existing tuples are not allowed. These properties avoid difficult concurrency issues commonly seen in mutable stores, such as key-value tables.

Each Linda process runs a sequential program written in a general-purpose language and performs tuple-related operations in an arbitrary fashion. Linda defines three primitive operations by which processes interact with a tuplespace: `in()`, `rd()`, and `out()`. `in()` fetches (consumes) a tuple from the tuplespace, and `rd()` is its non-destructive version. For both operations, a tuple can be referenced either as a concrete instance (i.e., all attributes are bound with specific values), or as a template with unbound variables. For example, a process may invoke `in('employee', 'Jane', ?age, 'Manager')` to retrieve a tuple in an associative manner, and the variable `age` will be bound according to the value of the fetched tuple. `out()` adds a tuple to the tuplespace.

2.2 Issues with Linda for Big-Data Applications

In Linda, the basic unit of scheduling is a process. We discuss four main issues that stem from this process-based granularity.

Programmability: The burden of spawning processes and orchestrating them is left to programmers, not the framework. Ideally, a programmer need only provide the computation logic for the data, leaving all the coordination aspects to the runtime.

Mobility: Linda processes, together with their internal state, have unbounded lifetime. Given the significant performance overhead of process migration, this limits process mobility across servers. The lack of mobility in turn limits the ability for long-running applications to adapt to the constant change in the availability of the underlying computing resources. In addition, poor mobility limits the runtime's ability to optimize for data locality by moving computation rather than data.

Adaptive Scalability: In cluster computing, replicating computation is crucial to achieving data parallelism. Unfortunately, Linda processes cannot be systematically replicated without understanding the application's semantics. Instead, Linda programs must explicitly spawn new processes to scale out, with limited information about the degree of exploitable parallelism and resource availability. While a Linda extension, Piranha [11], addresses this issue by letting the scheduler (not processes) spawn workers automatically, its applicability is limited to simple master/-worker architectures.

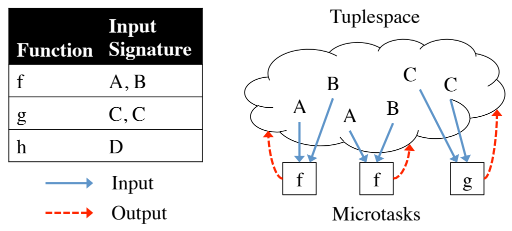
Fault Tolerance: Linda programs are not fault tolerant, even if we assume the underlying tuplespace is robust [44]. Process failures may leave the tuplespace in an *inconsistent* state. Transactions [4] (atomic execution of multiple Linda operations) can solve this integrity issue, but failure recovery is still solely the programmer's responsibility; the lost internal state of the process, such as the program counter and local variables, must be manually recovered. While process checkpointing [26] can automate this failure recovery, it does so at the cost of runtime overhead. Furthermore, both approaches rely on explicit annotations carefully made by application programmers.

2.3 Celias Programming Model

In Celias, we address the above issues by introducing the concept of a *microtask*. A microtask is an instance of a user-specified function with well-defined input and output. Unlike Linda processes, Celias microtasks interact with the tuplespace in a more constrained manner: tuple retrieval only happens at the beginning of the microtask, and tuple store at the end. This well-defined behavior of microtasks enables many benefits. First, microtasks are stateless and location independent – this makes them highly mobile giving the scheduler significant freedom in where and how it places data and computation. Second, the scheduler can easily duplicate microtasks to scale out computation. Third, fault tolerance is naturally achieved with the reliable tuplespace implementation, which is always in a consistent state due to the transactional nature of microtasks.

Figure 1 illustrates the Celias programming model. A Celias application defines a set of functions, akin to how a MapReduce job consists of a map() function and a reduce() function. Each function is defined with an input signature, which specifies one or more tuple templates. When all of the specified tuples are available, the scheduler triggers a microtask for the function. The microtask performs its computation and returns output tuples which are added to the tuplespace. These newly-added output tuples, perhaps together with pre-existing tuples, may in turn satisfy the input signatures for some other microtasks which are then triggered for execution. To allow flexible, data-dependent control flow, the output tuples that Celias microtasks produce can be arbitrary in number, type, and value.

A Celias function is represented as an input signature and a portion of code. The code performs computation with input tuples and, upon completion, may emit tuples



Function	Input Signature
f	A, B
g	C, C
h	D

Figure 1: Overview of Celias programming model

back to the tuplespace as output. Celias expects functions to resemble pure mathematical functions; the computation should be deterministic and side-effect free so that the output of a function is always the same given the same input. Functions may be written in many different programming languages, to expose a trade-off between productivity and performance for programmers.

An input signature declares one or more tuples, each of which could be either a concrete instance or a template, as in Linda. The scheduler is responsible for finding tuples that satisfy the signature of a function and for launching a microtask with those input tuples. We show how signatures can be utilized to specify constraints for input tuples in §3. If multiple and distinct candidate tuple sets exist, the scheduler may initiate multiple microtasks concurrently in order to exploit data/task parallelism.

Throughout the lifetime of a microtask, all the operations (fetching input tuples, performing computation, and returning output tuples back to the tuplespace) are performed *atomically*. In other words, the execution of a microtask appears to be instantaneous, and the tuplespace always remains "consistent" by not exposing an intermediate state at any given time. This atomic execution of microtasks greatly simplifies failure recovery. Upon a node failure, we can simply regard microtasks that were running on the node as aborted transactions, and retry those microtasks on other nodes.

A Celias "job" thus proceeds as follows. Before the job begins, the tuplespace is empty. A client provides input tuples (as a seed) to the tuplespace and initiates a job by providing a set of functions. The scheduler launches microtasks, possibly with chain reactions if the job is designed for such behavior, until no matching signature remains. The client then retrieves the remaining tuples in the tuplespace as the output of the job.

3 Celias with Examples

In this section, we walk through some simple applications to understand the early design details of signature specification and how Celias can be used to implement common parallel programming patterns. Here we follow the convention of using a human-readable string for the first attribute of tuples for the sake of readability.

Parallel Reduction: The following example performs parallel reduction over some given input numbers. We assume that the input tuples (seed) in the form of ('number', value) are provided by the client before the job begins.

```
# Seed input:
# ('number', value) values to add up

func sum: ('number', ?v1), ('number', ?v2)
  emit ('number', v1 + v2)
```

The signature of function `sum` specifies that two tuples will be used as input for the microtask. The `?` preceding variables `v1` and `v2` specifies that any values will match the unbound variable, so the scheduler can arbitrarily choose two tuples whose first attribute is the string `number`. The function `sum` adds two values from the matched tuples and emits a new tuple with the sum as its output. The job runs until there are no tuples that match the function's signature. For this application, the scheduler will stop when there is only one tuple left, which is the final output (total sum).

Note that exploiting data parallelism is solely handled by the Celias scheduler, not the application. Assuming there are an infinite number of processors available, the job will take $\log_2 n$ parallel steps, if the scheduling is done in lockstep. At the other extreme, the execution on a uniprocessor would be sequential in an arbitrary order.

Vector Addition: In this example we introduce the concept of *common variables* as a simple way of instructing the scheduler on how it should relate multiple tuples within a signature. We illustrate common variables through the following example for simple vector addition $\mathbf{c}_i = \mathbf{a}_i + \mathbf{b}_i$.

```
# ('a', index, value) vector a
# ('b', index, value) vector b

func vadd: ('a', ?i, ?v1), ('b', ?i, ?v2)
  emit ('c', i, v1 + v2)
```

The function `vadd` takes two tuples, both of which have `i` as a common variable. The scheduler chooses two tuples with the same index for each microtask to perform pairwise addition.

Quicksort: Celias can also emulate the general fork/join scheme to support recursive algorithms with dynamic parallelism. In the following example, we utilize the `stack` variable to combine the result of "forked" subroutines.

```
# ('list', list, []) the entire unsorted list

func sort: ('list', ?list, ?stack)
  if list is small enough:
    list = sorted(list)
    emit ('sorted', list, stack)
  else:
    pivot = list[0]
    left = filter(<, list[1:])
    right = filter(>=, list[1:])
    emit('pivot', pivot, [stack])
    emit('list', left, ['l' | stack])
    emit('list', right, ['r' | stack])

func combine: ('pivot', ?pivot, [?stack]),
              ('sorted', ?left, ['l' | ?stack]),
              ('sorted', ?right, ['r' | ?stack])
  list = left + [pivot] + right
  emit('sorted', list, stack)
```

Note that this simple code is not meant to be an efficient implementation, as the partitioning is not done in parallel.

MapReduce: In the MapReduce programming model, an application specifies map: (key, value) → list(key, value) and reduce: (key, list(value)) → list(key, value) functions. The following example shows how we can mimic the map-shuffle-reduce phases of the MapReduce model in Celias.

```
# ('input', k, v) key value pairs

func map: ('input', ?k, ?v)
  # Perform the mapper function.
  # Emit tuples ('intermediate', key, value).

func shuffle1: ('intermediate', ?k, ?v),
               !('bucket', ?k, _)
  list = [v]
  emit ('bucket', ?k, list)

func shuffle2: ('intermediate', ?k, ?v),
               ('bucket', ?k, ?list)
  list = [v | list]
  emit ('bucket', ?k, list)

func reduce: ('bucket', ?k, ?list),
             !('intermediate', ?k, _),
             !('input', _, _)
  # Perform the reducer function over the list.
  # Emit output tuples ('output', key, value).
```

The shuffle phase is done with two functions. `shuffle1` creates a bucket for each intermediate key; this bucket will subsequently collect all values for that key. The prefix "`!`" for the second tuple of the signature indicates that the specified tuple should *not* exist in the tuplespace. The "`_`" attribute means that any value can match and it is not bound to a variable. Once `shuffle1` creates the bucket for a key[1], the `shuffle2` function collects the intermediate values for that key into the bucket. When a key's bucket settles (i.e. no intermediate values remain for the key and the entire map phase is complete), `reduce` performs the reducer operation for the bucket. The two negated tuples in the `reduce` signature express these conditions.

If we have a series of MapReduce "jobs" expressed in a single Celias application, the scheduler can feed the output key-value pairs of a job to its next job, allowing pipelining between jobs [16]. Also, if the reducer only performs simple aggregation, such as max, count, and sum, we can also achieve pipelining within a MapReduce job by modifying the code above to perform online aggregation. Celias can exploit fine-grained task parallelism without any artificial barriers.

Shortest Path: In this example, we show how to implement a shortest path algorithm in Celias. The following code resembles the algorithm demonstrated in Pregel [30]. While this algorithm constructs a single-source shortest path tree, we can find multi-source shortest paths in a similar way.

[1]We assume that the scheduler enforces once-and-only-once execution when the signature matching involves negation. For example, the scheduler may hold a lock for (shuffle1, 'bucket', k) to avoid accidentally having multiple buckets for the same key under race conditions.

```
# ('outneigh', u, neighbors) ∀u ∈ V
# ('distance', u, v, dist) ∀(u, v) ∈ E
# ('updated', src) for source vertex src
# ('mindist', src, 0, NIL) for source vertex src
# ('mindist', u, ∞, NIL) ∀u ∈ V-{src}

func signal: ('updated', ?u),
             =('mindist', ?u, ?mindist, _),
             =('outneigh', ?u, ?neighbors)
  foreach v in neighbors:
    emit ('signal', u, v, mindist)

func update: ('signal', ?u, ?v, ?new),
             ('mindist', ?v, ?old, ?from),
             =('distance', ?u, ?v, ?dist)
  if new + dist < old:
    emit('updated', v)
    emit('mindist', v, new + dist, u)
  else:
    emit('mindist', v, old, from)
```

The tuples with the prefix "`=`" indicate that they are not removed when the signature matches. This behavior is similar to `rd()` in Linda. This read-only operation is useful when some tuples need to be read repeatedly.

4 Celias in Context

Many parallel programming systems, such as Clustera [19], Dryad [25], Hyracks [8], Nephele/PACTs [5], Scope [14], and Spark [46], represent a data flow with connections between parallel operations. Other systems using MapReduce jobs as building blocks (e.g., Pig [37], Hive [43], and FlumeJava [15]) also follow this approach.

There are two fundamental issues in these systems. First, the building blocks assume bulk transformation over immutable data collections. This coarse-grained unit of computation is not efficient for algorithms that require sparse and incremental computation with fine-grained, inter-data dependencies (e.g., recursive queries in SQL:1999 [20]). Second, the static dataflow systems cannot support dynamic algorithms with irregular parallelism, whose control path is data-dependent and thus must evolve with the computation itself (e.g., Quicksort in §3).

Incremental/Differential Processing: There is a growing body of work to address the granularity issues of the aforementioned systems. For example, CBP [28], Incoop [6], REX [33], and Naiad [32] support incremental computation over growing data and/or differential processing over iterations. Celias shares the same goals with those systems, but we also aim to support dynamic control flow in Celias.

CIEL: Unlike static dataflow systems, CIEL can modify the control flow at runtime to formulate algorithms with irregular parallelism [36]. CIEL is based on the classic fork/join scheme to build an explicit DAG of *tasks*, while the scheduling is completely driven by *data* in Celias.

We note two implications of the task-based dynamic dataflow systems. First, as the basic scheduling unit is a task, not data, such systems do not automatically capture data parallelism; instead applications must explicitly partition data across tasks and iterate within that data split. Second, the centralized scheduler must keep track of the task DAG for scheduling and fault tolerance purposes. This enforces coarse-grained parallelism on applications, as fine-grained data splits in combination with complex control flow may overload the scheduler with both computation and space overheads [36, 42].

Logic Programming: The sinature matching scheme of Celias closely resembles the Datalog query language and its variants [13], except that Celias allows microtasks to perform any computations on input tuples to produce arbitrary output, rather than predefined output with no computation.

Bloom extends Datalog with the explicit notion of mutable state and asynchronous message passing to support distributed programming on purely declarative foundations [2,3]. In Celias, we intentionally ruled out the concept of communication (done via location specifiers and channels in Bloom) to make microtasks completely stateless and location indepdendent.

Trigger-Based Systems: Celias has parallels to Percolator [40] and Oolong [34], in that they are based on fine-grained updates to mutable shared memory for incremental/asynchronous computation with triggers. Those systems activate triggers when a key-value entry is updated, in a similar sense to traditional database triggers [39].

In theory, the signature scheme of Celias provides a more expressive means to specify conditions to activate microtasks. Also, unlike triggers in those systems – which permit arbitrary access to the key-value table during execution – Celias microtasks are completely side-effect free. This allows simple replay of microtasks to suffice for failure recovery in Celias, instead of explicit transactions (Percolator) or checkpointing (Oolong). However, the inherent complexity of signature matching in Celias leaves the feasibility of its efficient implementation as an open question, as discussed in the next section.

5 Design and Implementation Challenges

This section briefly discusses design aspects that are still open-ended and the many challenges that remain in achieving a robust, efficient implementation of Celias.

Design Challenges: Celias's expressiveness greatly depends on the flexibility of signatures. It is unclear whether our current features (common variables, negatation, and read-only tuples) are powerful enough to express complex algorithms. However, adopting advanced features such as range queries may compromise the feasibility of an efficient implementation. We continue to weigh options to strike the right balance.

Another issue we are investigating is non-determinism. We expect functions to be deterministic for the sake of fault tolerance, yet there are many non-deterministic aspects to scheduling in Celias, e.g., which microtasks are chosen first and which combination of tuples match a signature. Improperly designed programs may suffer the curse of non-determinism but, on the other hand, non-deterministic parallelism creates new opportunities [35].

In contrast to static dataflow systems, the execution of a Celias job is unbound; an application may never terminate if its microtasks cause a continuous chain reaction without reaching a "fixpoint". For example, many machine learning algorithms run iteratiely towards convergence, but the required number of iterations is unknown in advance. Similarly, combinatorial optimization algorithms can explode the tuplespace with exponential growth of state. We plan to devise work-bounding mechanisms, from the in-band (e.g., function priority) and out-of-band (e.g., allowing external clients to observe and control the execution of a job) perspectives.

Implementation Challenges: An obvious concern with the Celias programming model is that it might be too difficult to implement efficiently: that microtasks are too fine-grained to schedule efficiently or that signature matching is too expensive. However, we see cause for optimism. A lesson we learn from MapReduce is that the granularity of computation (key-value pairs) can be orthogonal to the granularity of scheduling (block splits). We can adopt this idea by scheduling a collection of microtasks in a batch to minimize the per-microtask overhead.

For signature matching, we are witnessing the emergence of many parallel logic query systems [7, 9, 23, 31] based on Datalog. We believe that an efficient implementation is feasible since Celias's signature mathching is simpler than Datalog (e.g., no inductive/recursive inference) though we also see that the *online* aspects of Celias bring another implementation challenge.

The efficiency of the system will depend greatly on our ability to exploit data locality thus minimizing communication cost. The original Tuplespace abstraction imposes high communication cost as the exchange of tuples happens between processes, most likely running on separate machines. Since tasks are location independent in Celias, the scheduler can make a trade-off between moving data and moving computation. We can also allow programmers to use annotations that give Celias scheduling hints, e.g., regarding where data should be stored [18, 41] or which function will take output tuples as input next.

The fault tolerance of Celias relies on a robust implementation of tuplespace with microtask transaction support. Since the abstraction of Sinfonia minitransactions [1] is conceptually similar to microtasks, we believe that its scalable transaction implementation is also applicable to Celias. We are also currently evaluating the approaches proposed by RAMCloud [38] and HyperDex [22], to implement a high-performance, scalable, and reliable tuplespace tailored to the Celias programming model.

References

[1] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of ACM SOSP* (2007).

[2] ALVARO, P., CONWAY, N., HELLERSTEIN, J. M., AND MARCZAK, W. R. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Proceedings of CIDR* (2011).

[3] ALVARO, P., MARCZAK, W. R., CONWAY, N., HELLERSTEIN, J. M., MAIER, D., AND SEARS, R. Dedalus: Datalog in Time and Space. In *Proceedings of Datalog 2.0 Workshop* (2011).

[4] BAKKEN, D., AND SCHLICHTING, R. Supporting Fault-Tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Systems 6*, 3 (1995), 287–302.

[5] BATTRÉ, D., EWEN, S., HUESKE, F., KAO, O., MARKL, V., AND WARNEKE, D. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *Proceedings of ACM SoCC* (2010).

[6] BHATOTIA, P., WIEDER, A., RODRIGUES, R., ACAR, U. A., AND PASQUIN, R. Incoop: MapReduce for Incremental Computations. In *Proceedings of ACM SoCC* (2011).

[7] BORKAR, V., BU, Y., CAREY, M., ROSEN, J., POLYZOTIS, N., CONDIE, T., WEIMER, M., RAMAKRISHNAN, R., DROR, G., KOENIGSTEIN, N., ET AL. Declarative Systems for Large-Scale Machine Learning. *Bulletin of the Technical Committee on Data Engineering 35*, 2 (2012), 24–32.

[8] BORKAR, V., CAREY, M., GROVER, R., ONOSE, N., AND VERNICA, R. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *Proceedings of IEEE ICDE* (2011).

[9] BU, Y., BORKAR, V., CAREY, M., ROSEN, J., POLYZOTIS, N., CONDIE, T., WEIMER, M., AND RAMAKRISHNAN, R. Scaling Datalog for Machine Learning on Big Data. *arXiv:1203.0160 [cs.DB]* (2012).

[10] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB 3*, 1 (2010), 285–296.

[11] CARRIERO, N., FREEMAN, E., GELERNTER, D., AND KAMINSKY, D. Adaptive Parallelism and Piranha. *IEEE Computer 28*, 1 (1995), 40–49.

[12] CARRIERO, N., GELERNTER, D., MATTSON, T., AND SHERMAN, A. The Linda alternative to message-passing systems. *Parallel Computing 20*, 4 (1994), 633–655.

[13] CERI, S., GOTTLOB, G., AND TANCA, L. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering 1*, 1 (1989), 146–166.

[14] CHAIKEN, R., JENKINS, B., LARSON, P.-Å., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB 1*, 2 (2008), 1265–1276.

[15] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R., BRADSHAW, R., AND WEIZENBAUM, N. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proceedings of ACM PLDI* (2010).

[16] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J., ELMELEEGY, K., AND SEARS, R. MapReduce Online. In *Proceedings of USENIX NSDI* (2010).

[17] CORKILL, D. Blackboard systems. *AI Expert 6*, 9 (1991), 40–47.

[18] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of USENIX OSDI* (2004).

[19] DEWITT, D. J., PAULSON, E., ROBINSON, E., NAUGHTON, J. F., ROYALTY, J., SHANKAR, S., AND KRIOUKOV, A. Clustera: An Integrated Computation And Data Management System. *PVLDB 1*, 1 (2008), 28–41.

[20] EISENBERG, A., AND MELTON, J. SQL: 1999, formerly known as SQL3. *SIGMOD Record 28*, 1 (1999), 131–138.

[21] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S., QIU, J., AND FOX, G. Twister: A Runtime for Iterative MapReduce. In *Proceedings of ACM HPDC* (2010).

[22] ESCRIVA, R., WONG, B., AND SIRER, E. HyperDex: A Distributed, Searchable Key-Value Store. In *Proceedings of ACM SIGCOMM* (2012).

[23] GANGULY, S., SILBERSCHATZ, A., AND TSUR, S. A Framework for the Parallel Processing of Datalog Queries. *SIGMOD Record 19*, 2 (1990), 143–152.

[24] GELERNTER, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), 80–112.

[25] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of ACM EuroSys* (2007).

[26] JEONG, K., AND SHASHA, D. PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda. In *IEEE SRDS* (1994).

[27] LIN, J. MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail! *Big Data 1*, 1 (2013), 28–37.

[28] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful Bulk Processing for Incremental Analytics. In *Proceedings of ACM SoCC* (2010).

[29] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB 5*, 8 (2012), 716–727.

[30] MALEWICZ, G., AUSTERN, M., BIK, A., DEHNERT, J., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of ACM SIGMOD* (2010).

[31] MARZ, N. Cascalog. `http://cascalog.org`.

[32] MCSHERRY, F., MURRAY, D. G., ISAACS, R., AND ISARD, M. Differential dataflow. In *Proceedings of CIDR* (2013).

[33] MIHAYLOV, S. R., IVES, Z. G., AND GUHA, S. Rex: Recursive, Delta-Based Data-Centric Computation. *PVLDB 5*, 11 (2012), 1280–1291.

[34] MITCHELL, C., POWER, R., AND LI, J. Oolong: Asynchronous Distributed Applications Made Easy. In *Proceedings of APSys* (2012).

[35] MURRAY, D., AND HAND, S. Non-deterministic parallelism considered useful. In *Proceedings of USENIX HotOS* (2011).

[36] MURRAY, D., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of USENIX NSDI* (2011).

[37] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign Language for Data Processing. In *Proceedings of ACM SIGMOD* (2008).

[38] ONGARO, D., RUMBLE, S., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast Crash Recovery in RAMCloud. In *Proceedings of ACM SOSP* (2011).

[39] PATON, N. W., AND DÍAZ, O. Active Database Systems. *ACM Computing Surveys 31*, 1 (1999), 63–103.

[40] PENG, D., AND DABEK, F. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of USENIX OSDI* (2010).

[41] POWER, R., AND LI, J. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of USENIX OSDI* (2010).

[42] QIAN, Z., CHEN, X., KANG, N., CHEN, M., YU, Y., MOSCIBRODA, T., AND ZHANG, Z. MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud. In *Proceedings of ACM EuroSys* (2012).

[43] THUSOO, A., SARMA, J., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB 2*, 2 (2009), 1626–1629.

[44] XU, A., AND LISKOV, B. A Design for a Fault-Tolerant, Distributed Implementation of Linda. In *Proceedings of IEEE FTCS* (1989).

[45] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of USENIX NSDI* (2011).

[46] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *Proceedings of USENIX HotCloud* (2010).