Scalable Progressive Analytics on Big Data in the Cloud

Badrish Chandramouli[1] Jonathan Goldstein[1] Abdul Quamar[2*]
[1]Microsoft Research, Redmond [2]University of Maryland, College Park
{badrishc, jongold}@microsoft.com, abdul@cs.umd.edu

ABSTRACT

Analytics over the increasing quantity of data stored in the Cloud has become very expensive, particularly due to the pay-as-you-go Cloud computation model. Data scientists typically manually extract samples of increasing data size (*progressive samples*) using domain-specific sampling strategies for exploratory querying. This provides them with user-control, repeatable semantics, and result provenance. However, such solutions result in tedious workflows that preclude the reuse of work across samples. On the other hand, existing *approximate query processing* systems report early results, but do not offer the above benefits for complex ad-hoc queries. We propose a new *progressive analytics* system based on a progress model called Prism that (1) allows users to communicate progressive samples to the system; (2) allows efficient and deterministic query processing over samples; and (3) provides repeatable semantics and provenance to data scientists. We show that one can realize this model for *atemporal* relational queries using an unmodified *temporal* streaming engine, by re-interpreting temporal event fields to denote progress. Based on Prism, we build Now!, a progressive data-parallel computation framework for Windows Azure, where progress is understood as a first-class citizen in the framework. Now! works with "progress-aware reducers"- in particular, it works with streaming engines to support progressive SQL over big data. Extensive experiments on Windows Azure with real and synthetic workloads validate the scalability and benefits of Now! and its optimizations, over current solutions for progressive analytics.

1. INTRODUCTION

With increasing volumes of data stored and processed in the Cloud, analytics over such data is becoming very expensive. The pay-as-you-go paradigm of the Cloud causes computation costs to increase linearly with query execution time, making it possible for a *data scientist* to easily spend large amounts of money analyzing data. The problem is exacerbated by the exploratory nature of analytics, where queries are iteratively discovered and refined, including the submission of many off-target and erroneous queries (e.g., bad parameters). In traditional systems, queries must execute to completion before such problems are diagnosed, often after hours of expensive compute time are exhausted.

*Work performed during internship at Microsoft Research.

Data scientists therefore typically choose to perform their ad-hoc querying on extracted *samples* of data. This approach gives them the control to carefully choose from a huge variety [11, 28, 27] of sampling strategies in a domain-specific manner. For a given sample, it provides precise (e.g., relational) query semantics, repeatable execution using a query processor and optimizer, result provenance in terms of what data contributed to an observed result, and query composability. Further, since choosing a fixed sample size a priori for all queries is impractical, data scientists usually create and operate over multiple *progressive samples* of increasing size [28].

1.1 Challenges

In an attempt to help data scientists, the database community has proposed *approximate query processing* (*AQP*) systems such as CONTROL [20] and DBO [23] that perform *progressive analytics*. We define progressive analytics as the generation of early results to analytical queries based on partial data, and the progressive refinement of these results as more data is received. Progressive analytics allows users to get early results using significantly fewer resources, and potentially end (and possibly refine) computations early once sufficient accuracy or query incorrectness is observed.

The general focus of AQP systems has, however, been on automatically providing confidence intervals for results, and selecting processing orders to reduce bias [21, 9, 15, 17, 31]. The premise of AQP systems is that users are not involved in specifying the semantics of early results; rather, the system takes up the responsibility of defining and providing accurate early results. To be useful, the system needs to automatically select effective sampling strategies for a particular combination of query and data. This can work for narrow classes of workloads, but does not generalize to complex ad-hoc queries. A classic example is the infeasibility of sampling for join trees [10]. In these cases, a lack of user involvement with "fast and loose" progress has shortcomings; hence, data scientists tend to prefer the more laborious but controlled approach presented earlier. We illustrate this using a running example.

Example 1 (CTR) *Consider an advertising platform where an analyst wishes to compute the* click-through-rate (CTR*) for each ad. We require two sub-queries (Q_c and Q_i) to compute (per ad) the number of clicks and impressions, respectively. Each query may be non-trivial; for example, Q_c needs to process clicks on a per-user basis to consider only legitimate (non-automated) clicks from a webpage whitelist. Further, Q_i may need to process a different set of logged data. The final query Q_{ctr} joins (for each ad) the results of Q_c and Q_i, and computes their ratio as the CTR. Figure 1 shows a toy input sorted by user, and the final results for Q_c, Q_i, and Q_{ctr}.*

(a)

User	Ad	...
u_0	a_0	...
u_1	a_0	...
u_2	a_0	...

(b)

User	Ad	...
u_0	a_0	...
u_0	a_0	...
u_1	a_0	...
u_2	a_0	...
u_2	a_0	...

(c)

Ad	Clicks
a_0	3

Ad	Imprs
a_0	5

(d)

Ad	CTR
a_0	0.6

Figure 1: (a) Click data; (b) Impression data; (c) Final result of Q_c and Q_i; (d) Final result of Q_{ctr}.

Next, Figure 3 (a) and (b) show progressive *results for the same queries Q_c and Q_i. Without user involvement in defining progressive samples, the exact sequence of progressive counts can be non-deterministic across runs, although the final counts are precise. Further, depending on the relative speed and sequence of results for Q_c and Q_i, Q_{ctr} may compose arbitrary progressive results, resulting in significant variations in progressive CTR results. Figures 3(c) and (d) show two possible results for Q_{ctr}. For example, a CTR of* 2.0 *would result from combining the first tuple from Q_c and Q_i. Some results that are not even meaningful (e.g., CTR $>$ 1.0) are possible. Although both results eventually get to the same final CTR, there is no mechanism to ensure that the inputs being correlated to compute progressive CTRs are deterministic and comparable (e.g., computed using the same sample of users).*

The above example illustrates several challenges:
1) User-Control: Data scientists usually have domain expertise that they leverage to select from a range of sampling strategies [11, 28, 27] based on their specific needs and context. In Example 1, we may progressively sample both datasets identically in user-order for meaningful progress, avoiding the join sampling problem [10]. Users may also need more flexibility; for instance, with a star-schema dataset, they may wish to *fully process* the small dimension table before sampling the fact table, for better progressive results.
2) Semantics: Relational algebra provides precise semantics for SQL queries. Given a set of input tables, the correct output is defined by the input and query alone, and is independent of dynamic properties such as the order of processing tuples. However, for complex queries, existing AQP systems use *operational semantics*, where early results are on a best-effort basis. Thus, it is unclear what a particular early result means to the user.
3) Repeatability & Optimization: Two runs of a query in AQP may provide a different sequence of early results, although they have to both converge to the same final answer. Thus, without limiting the class of queries which are progressively executed, it is hard to understand what early answers mean, or even recognize anomalous early answers. Even worse, changing the physical operators in the plan (e.g., changing operators within the ripple join family [16]) can significantly change what early results are seen!
4) Provenance: Users cannot easily establish the provenance of early results, e.g., link an early result (CTR=3.0) to particular contributing tuples, which is useful to debug and reason about results.
5) Query Composition: The problem of using operational semantics is exacerbated when one starts to compose queries. Example 1 shows that one may get widely varying results (e.g., spurious CTR values) that are hard to reason about.
6) Scale-Out: Performing progressive analytics at scale exacerbates the above challenges. The CTR query from Example 1 is expressed as two *map-reduce* (*MR*) jobs that partition data by UserId, feeding a third job that partitions data by a different key (AdId); see Figure 2. In a complex distributed multi-stage workflow, accurate deterministic progressive results can be very

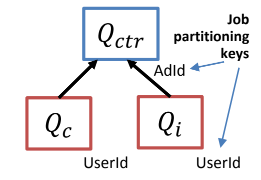


Figure 2: CTR; MR jobs.

(a)

Ad	Clicks
a_0	2
a_0	3

(b)

Ad	Imprs
a_0	1
a_0	4
a_0	5

(c)

Ad	CTR
a_0	2.0
a_0	0.5
a_0	0.6

(d)

Ad	CTR
a_0	3.0
a_0	0.75
a_0	0.6

Figure 3: (a) Progressive Q_c output; (b) Progressive Q_i output; (c) & (d) Two possible progressive Q_{ctr} results.

useful. *Map-reduce-online* (*MRO*) [12] adds a limited form of pipelining to MR, but MRO reports a heuristic progress metric (average fraction of data processed across mappers) that does not eliminate the problems discussed above (§ 6 covers related work).

To summarize, data scientists prefer user-controlled progressive sampling because it helps avoid the above issues, but the lack of system support results in a tedious and error-prone workflow that precludes the reuse of work across progressive samples. We need a system that (1) allows users to communicate progressive samples to the system; (2) allows efficient and deterministic query processing over progressive samples, without the system *itself* trying to reason about specific sampling strategies or confidence estimation; and yet (3) continues to offer the desirable features outlined above.

1.2 Contributions

1) Prism*: A New Progress Model & Implementation*

We propose (§ 2) a new progress model called Prism (*Progressive sampling model*). The key idea is for users to encode their chosen progressive sampling strategy into the data by augmenting tuples with explicit *progress intervals* (*PIs*). PIs denote logical points where tuples enter and exit the computation, and explicitly assign tuples to progressive samples. PIs offer remarkable flexibility for encoding sampling strategies and ordering for early results, including arbitrarily overlapping sample sequences and special cases such as the star-schema join mentioned earlier (§ 2.5 has more details).

PIs propagate through Prism operators. Combined with progressive operator semantics, PIs provide closed-world determinism: the exact sequence of early results is a deterministic function of augmented inputs and the logical query alone. They are independent of physical plans, which enables side-effect-free query optimization. Provenance is explicit; result tuples have PIs that denote the exact set of contributing inputs. Prism also allows meaningful query composition, as operators respect PIs. If desired, users can encode confidence interval computations as part of their queries.

The introduction of a new progress model into an existing relational engine appears challenging. However, interestingly, we show (§ 2.4) that a progressive in-memory relational engine based on Prism can be realized immediately using an *unmodified temporal streaming engine*, by carefully reusing its temporal fields to denote progress. Tuples from successive progressive samples get incrementally processed when possible, giving a significant performance benefit. Note here that the temporal engine is unaware that it is processing *atemporal* relational queries; we simply re-interpret its temporal fields to denote progress points. While it may appear that in-memory queries can be memory intensive since the final answer is computed over the entire dataset, Prism allows us to exploit sort orders and foreign key dependencies in the input data and queries to reduce memory usage significantly (§ 2.6).

Prism generalizes AQP Our progress semantics are compatible with queries for which prior AQP techniques with statistical guarantees apply, and thus don't require user involvement. These techniques simply correspond to different PI assignment policies for input data. For instance, variants of ripple join [16] are different PI assignments for a temporal symmetric-hash-join, with confidence intervals computed as part of the query. Thus, Prism is orthogonal to and can leverage this rich area of prior work, while adding

PI	User	Ad
$[0,\infty)$	u_0	a_0
$[1,\infty)$	u_1	a_0
$[2,\infty)$	u_2	a_0

(a)

PI	User	Ad
$[0,\infty)$	u_0	a_0
$[0,\infty)$	u_0	a_0
$[1,\infty)$	u_1	a_0
$[2,\infty)$	u_2	a_0
$[2,\infty)$	u_2	a_0

(b)

PI	Ad	Clicks
$[0,1)$	a_0	1
$[1,2)$	a_0	2
$[2,\infty)$	a_0	3

PI	Ad	Imprs
$[0,1)$	a_0	2
$[1,2)$	a_0	3
$[2,\infty)$	a_0	5

(c)

PI	Ad	CTR
$[0,1)$	a_0	0.5
$[1,2)$	a_0	0.66
$[2,\infty)$	a_0	0.6

(d)

Figure 4: (a,b) Input data with progress intervals; (c) Progressive results of Q_c and Q_i; (d) Progressive output of Q_{ctr}.

the benefit of repeatable and deterministic semantics. In summary, *Prism gives progressive results the same form of determinism and user control that relational algebra provides final results.*

2) Applying Prism in a Scaled-Out Cloud Setting

The Prism model is particularly suitable for progressive analytics on big data in the Cloud, since queries in this setting are complex, and memory- and CPU-intensive. Traditional scalable distributed frameworks such as MR are not pipelined, making them unsuitable for progressive analytics. MRO adds pipelining, but does not offer the semantic underpinnings of progress necessary to achieve the desirable features outlined earlier.

We address this problem by designing and building a new framework for progressive analytics called Now! (§ 3). Now! runs on Windows Azure; it understands and propagates progress (based on the Prism model) as a first-class citizen inside the framework. Now! generalizes the popular data-parallel MR model and supports *progress-aware reducers* that understand explicit progress in the data. In particular, Now! can work with a temporal engine (we use StreamInsight [3]) as a progress-aware reducer to enable scaled-out progressive relational (SQL) query support in the Cloud. Now! is a novel contribution in its own right, with several important features:

- Fully pipelined progressive computation and data movement across multiple stages with different partitioning keys, in order to avoid the high cost of sending intermediate results to Cloud storage.
- Elimination of sorting in the framework using progress-ordered data movement, partitioned computation pushed inside progress-aware reducers, and support for the traditional reducer API.
- Progress-based merge of multiple map outputs at a reducer node.
- Concurrent scheduling of multi-stage map and reduce jobs with a new scheduling policy and flow control scheme.

We also extend Now! (§ 4) with a high performance mode that eliminates disk writes, and discuss high availability (by leveraging progress semantics in a new way) and straggler management.

We perform a detailed evaluation (§ 5) of Now! on Windows Azure (with StreamInsight) over real and benchmark datasets up to 100GB, with up to 75 large-sized Azure compute instances. Experiments show that we can scale effectively and produce meaningful early results, making Now! suitable in a pay-as-you-go environment. Now! provides a substantial reduction in processing time, memory and CPU usage as compared to current schemes; performance is significantly enhanced by exploiting sort orders and using our memory-only processing mode.

Paper Outline § 2 provides the details of our proposed model for progressive computation Prism. We present Now! in detail in § 3; and discuss several extensions in § 4. The detailed evaluation of Now! is covered in § 5, and related work is discussed in § 6.

2. Prism SEMANTICS & CONSTRUCTION

At a high level, our progress model (called Prism) defines a logical linear *progress domain* that represents the progress of a query. Sampling strategies desired by data scientists are encoded into the data before query processing, using *augmented tuples with progress intervals* that precisely define how data progressively contributes to result computation. Users express their data analytics as relational queries that consist of a DAG of *progressive operators*. An extension of traditional database operators, progressive operators understand and propagate progress intervals based on precisely defined operator semantics. The result of query processing is a sequence of augmented tuples whose progress intervals denote *early results and their associated regions of validity in the progress domain*. Each of these steps is elaborated in the following subsections.

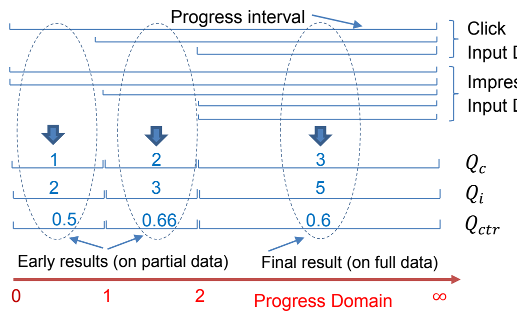


Figure 5: Input and output progress intervals, query semantics.

2.1 Logical Progress and Progress Intervals

Prism defines a logical linear *progress domain* $\mathcal{P}$ as the range of non-negative integers $[0,\infty)$. Progress made by a query at any given point during computation is explicitly indicated by a non-decreasing *progress point* $p \in \mathcal{P}$. Progress point ∞ indicates the point of query completion.

Next, we associate a *progress interval* (*PI*) from the progress domain to every tuple in the input data. More formally, each tuple T is augmented with two new attributes, a *progress-start* P^+ and a *progress-end* P^-, that jointly denote a PI $[P^+, P^-)$. P^+ indicates the progress point at which a tuple T starts participating in the computation, and P^- (if not ∞) denotes the progress point at which tuple T stops contributing to the computation. PIs enable users to specify domain-specific progressive sampling strategies. PI assignment can be controlled by data scientists to ensure quicker and more meaningful early results, either directly or using a layer between the system and the user. Figures 4(a) and (b) show PIs for our running example inputs; they are also depicted in Figure 5 (top). We provide several concrete examples of PI assignment in Section 2.5.

2.2 Progressive Operators and Queries

Progressive Operators Every relational operator O has a progressive counterpart, which computes augmented output tuples from augmented input tuples. Logically, the output at progress point p is the operation O applied to input tuples whose PIs are stabbed by p. Figures 4(c) and 5 (bottom) show the results of Q_c and Q_i, which behave as Count operators. We see that Q_c produces a progressive count of 1 at progress point 0, which it revises to 2 and 3 at progress points 1 and 2. As a result, the PIs for these tuples are $[0,1)$, $[1,2)$ and $[2,\infty)$ respectively.

The P^- for an output tuple may not always be known at the same time as when the operator determine its P^+. Thus, an operator may output a tuple having an eventual PI of $[P^+,P^-)$ in two separate pieces: (1) at progress point P^+, it generates a *start-edge* tuple T_1 with a PI $[P^+,\infty)$ indicating that the tuple participates in the result forever; (2) at the later progress point P^-, it generates an *end-edge* tuple T_2 with the actual PI $[P^+,P^-)$. We use the term *progress-sync* to denote the progress point associated with a tuple (or its subsequent update). The start-edge tuple T_1 has a progress-sync of P^+, whereas the end-edge tuple T_2 has a progress-sync of P^-.

Every operator both processes and generates augmented tuples in *non-decreasing progress-sync order*. The eventual P^- values for early results that get refined later are less than ∞, to indicate that the result is not final. For example, consider an Average operator that reports a value a_0 from progress point 0 to 10, and revises it to a_1 from progress point 10 onwards. Tuple a_0 has an eventual PI of $[0, 10)$. This is reported as a start-edge $[0, \infty)$ at progress point 0. At progress point 10, the operator reports an end-edge $[0, 10)$ for the old average a_0, followed immediately by a start-edge $[10, \infty)$ for the revised average a_1. Similarly, a progressive Join operator with one tuple on each input with PIs $[10, 20)$ and $[15, 25)$ – if the join condition is satisfied – produces a result tuple with PI $[15, 20)$, the intersection of the two input PIs. Note here that the output tuple's PI ends at 20 because its left input is no longer valid at that point.

Progressive Queries Based on the above semantics, operators can be composed meaningfully to produce progressive queries. We define Prism output for a relational query Q as:

Definition 1 (Prism Output) *Associated with each input tuple is a progress interval (PI). At every unique progress point p across all PI endpoints in the input data, there exists a set O_p of output results with PIs stabbed by p. O_p is defined to be exactly the result of the query Q evaluated over input tuples with PIs stabbed by p.*

2.3 Summary of Benefits of the Prism Model

The results of Q_{ctr} for our running example are shown in Figures 4(d) and 5 (bottom); every CTR is meaningful as it is computed on some prefix of users (for our chosen progress assignment), and CTR provenance is provided by PIs. The final CTR of 0.6 is the only tuple active at progress point ∞, as expected.

It is easy to see that the output of a progressive query is a deterministic function of the (augmented) input data and the logical query alone. Further, these progressive results are fixed for a given input and logical query, and are therefore repeatable. Prism enables data scientists to use their domain knowledge to control progressive samples; Section 2.5 provides several concrete examples. Early results in Prism carry the added benefit of provenance that helps debug and reason about early results: the set of output tuples with PIs stabbed by progress point p denote the progressive result of the query at p. The provenance of these output tuples is simply all tuples along their input paths whose PIs are stabbed by p.

One can view Prism as a generalization of relational algebra with progressive sampling as a first-class concept. Relational algebra prescribes the final answer to a relational query but does not cover how we get there using partial results. The Prism algebra explicitly specifies, for any query, not only the final answer, but every intermediate (progressive) result and its position in the progress domain.

2.4 Implementing Prism

One can modify a database engine to add PI support to all operators in the engine. However, we can realize Prism *without* incurring this effort. The idea is to leverage a *stream processing engine* (*SPE*) as the progressive query processor. In particular, the semantics underlying a temporal SPE such as NILE [19], STREAM [4], or StreamInsight [3] (based on temporal databases [22]) can be leveraged to denote progress, with the added benefit of incremental processing across samples when possible. With StreamInsight's temporal model, for example, the event validity time interval [6] $[V_s, V_e)$ directly denotes the PI $[\mathsf{P}^+, \mathsf{P}^-)$. T_1 is an insertion and T_2 is a retraction (or revision [33]). Likewise, T_1 and T_2 correspond to Istreams and Dstreams in STREAM, and positive and negative tuples in NILE. We feed the input tuples converted into events to a *continuous query* corresponding to the original atemporal SQL query. The unmodified SPE operates on these tuples as though they were temporal events, and produces output events with timestamp fields that we re-interpret as tuples with PIs.

Note that with this construction, the SPE is unaware that it is being used as a progressive SQL processor. It processes and produces events whose temporal fields are re-interpreted to denote progress of an *atemporal* (relational) query. For instance, the temporal symmetric-hash-join in an SPE effectively computes a sequence of joins over a sequence of progressive samples very efficiently. The resulting query processor transparently handles all of SQL, including user-defined functions, with all the desirable features of our new progress model.

2.5 PI Assignment

Any progressive sampling strategy at the inputs corresponds to a PI assignment; several are discussed next.

Inclusive & Non-inclusive Samples With *inclusive samples* (as used, for example, in EARL [25]), each sample is a superset of the previous one. To specify these, input tuples are assigned a P^- of ∞, and non-decreasing P^+ values based on when tuples become a part of the sample, as shown in Figure 5 (top). In case of *non-inclusive samples*, tuples have a finite P^- to denote that they no longer participate in computation beyond P^-, and can even reappear with a greater P^+ for a later sample (our technical report [7] includes a concrete example of expressing non-inclusive sampling using PIs).

Reporting Granularity Progress reporting granularity can be controlled by individual queries, by adjusting the way P^+ moves forward. Data is often materialized in a statistically relevant order, and we may wish to include k additional tuples in each successive sample. We use a streaming AlterLifetime [8] operator that sets P^+ for the nth tuple to $\lfloor n/k \rfloor$ and P^- to ∞. This increases P^+ by 1 after every k tuples, resulting in the engine producing a new progressive result every k tuples. We refer to the set of tuples with the same P^+ as a *progress-batch*. Data scientists often start with small progress-batches to get quick estimates, and then increase batch sizes (e.g., exponentially) as they get diminishing returns with more data.

Joins & Star Schemas In case of queries involving an equi-join, we may apply an identical sampling strategy (e.g., pseudo-random) over the join key in both inputs as this increases the likelihood of getting useful early results. With a star-schema, we may set all tuples in the small dimension table to have a PI of $[0, \infty)$, while progressively sampling from the fact table as $[0, \infty), [1, \infty), \ldots$. This causes a Join operator to "preload" the dimension table before progressively sampling the fact table for meaningful early results.

Stratified Sampling Stratified sampling groups data on a certain key and applies a sampling strategy (e.g., uniform) within each group to ensure that rare subgroups are sufficiently represented. BlinkDB [2] pre-computes stratified samples of different sizes and responds to queries within a given error and response time by choosing the correct sample to compute the query on. Stratified sampling is easy to implement with Prism: we perform a GroupApply operation [8] by the key, with an AlterLifetime inside the GroupApply to create progress-batches as before. The temporal Union that merges groups respects timestamp ordering, resulting in a final dataset with PIs that exactly represent stratified sampling. Stratified samples of increasing size can be constructed similarly.

Other Examples For online aggregation, we may assign non-decreasing P^+ values over a pre-defined random order of tuples for quick result convergence. Active learning [11] changes the sampling strategy based on outcomes from prior samples. Prior proposals for ordering data for quick convergence [17, 16, 30, 9] simply correspond to different PI assignment schemes in Prism.

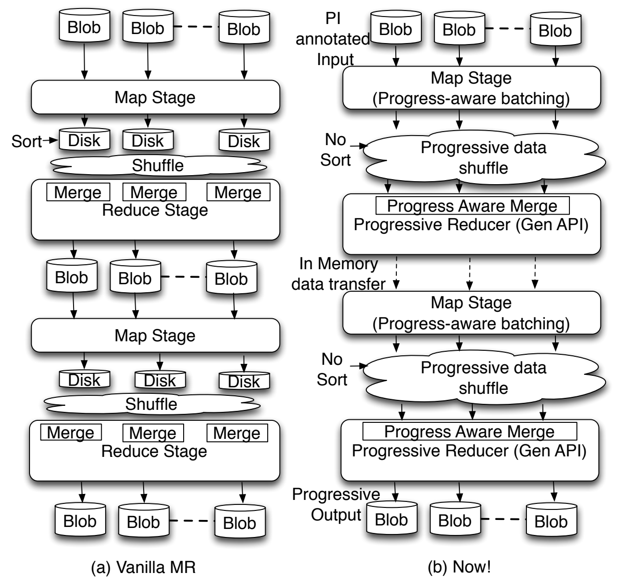


Figure 6: System architecture (MR vs. Now!).

2.6 Performance Optimizations

Query processing using an in-memory streaming engine can be expensive since the final answer is over the entire dataset. Prism enables crucial performance optimizations that can improve performance significantly in practical situations. Consider computation Q_c, which is partitionable by UserId. We can exploit the compile-time property that progress-sync ordering is the same as (or correlated to) the partitioning key, to reduce memory usage and consequently throughput. The key intuition is that although every tuple with PI $[P^+, \infty)$ logically has a P^- of ∞, it does not contribute to any progress point beyond P^+. Thus, we can temporarily set P^- to P^++1 before feeding the tuples to the SPE. This effectively causes the SPE to not have to retain information related to progress point P^+ in memory once computation for P^+ is done. The result tuples have their P^- set back to ∞ to retain the original query semantics (these query modifications are introduced using compile-time query rewrites). A similar optimization applies to equi-joins; see [7] for details. We will see in Section 5 that this optimization can result in orders-of-magnitude performance benefits.

We next discuss our new big data framework called Now!, that implements Prism for MR-style computation at scale in a distributed setting (applying Prism to other models such as graphs is an interesting area of future work).

3. Now! ARCHITECTURE AND DESIGN

3.1 Overview

At a high level, Now!'s architecture is based on the Map-Reduce (MR) [14] computation paradigm. Figure 6 shows the overall design of Now! (right) as compared to vanilla MR (left), for a query with two stages and different partitioning keys. *Blobs* in the figure indicate the format of input and output data on Windows Azure's distributed Cloud storage, and can be replaced by any distributed persistent storage such as HDFS. The key points are as follows:

1) Progress-aware data flow: Now! implements the Prism progress model and provides support for data flow (§ 3.2) in strict progress-sync order. The main components of progress-aware data flow are:

- **Batching** Now! reads input data annotated with PIs (progressive samples) and creates *batches* (§ 3.2.1) of tuples with the same progress-sync. Data movement in Now! is fully pipelined in terms of these *progress-batches*, in progress-sync order.
- **Sort-free data shuffle** MR sorts the map output by key, followed by a merge to enable grouping by key at reducers. This sort-merge operation in MR is a performance bottleneck [26]. In contrast, the batched map output in Now! is partitioned and shuffled across the network to reducers *without sorting* (§ 3.2.2), thus retaining progress-sync order with improved performance.
- **Progress-aware merge** A progress-aware merge at reducers is key to enabling the Prism model for progressive query results. Each reducer groups together batches received from different mappers, that belong to the same PI, into a single progress-batch, and ensures that all progress-batches are processed in strict progress-sync order (§ 3.2.3) along all data flow paths.

Data flow between map and reduce in Now! uses TCP connections which guarantee FIFO delivery. Since the input data is read in progress-sync order and all components retain this invariant, we are guaranteed global progress-sync order for progress-batches.

2) Progress-aware reducers: Now! introduces the notion of a *progress-aware reducer* (Section 3.2.4), that accepts and produces augmented tuples in progress-sync order, and logically adheres to the Prism query model. The progress-aware merge generates progress-batches in progress-sync order; these are fed directly to reducers that produce early results in progress-sync order. While one could write custom reducers, we use an unmodified SPE (§ 2.4) as a progress-aware reducer for progressive relational queries.

3) Multi-stage support: Now! supports *concurrent scheduling* of all jobs in a multi-stage query and *co-location* of mappers of dependent jobs with the reducers of feeding jobs on the same slave machine (Section 3.3). Data transfer between jobs is in-memory providing significant savings in a Cloud deployment where blob access is expensive.

4) Flow control: Now! provides end-to-end flow control to avoid buffer overflows at intermediate stages such as mapper output, reducer input and reducer output for multi-stage MR. The flow control mechanism ensures data flows at a speed that can be sustained by downstream consumers. We use a *blocking concurrent queue* (*BCQ*), a lock-free data structure which supports concurrent enqueue and dequeue operations, for implementing an end-to-end flow control mechanism for Now! (our technical report [7] has more details on flow control in Now!).

5) In-memory data processing: By default, Now! materializes map output on disk to provide better data availability during failure recovery. For better interactivity, we also support a high-performance in-memory mode (see Section 4).

3.2 Progress-aware data flow & computation

Data flow in Now! is at the granularity of progress-batches and governed by PIs. This section describes the generation and flow of these progress-batches in the framework.

3.2.1 Progress-aware batching

The input data is partitioned into a number of input splits (one for each mapper), data tuples in each of which are assigned progress intervals in progress-sync order. The mapper reads its input split as progress annotated tuples (progressive samples), and invokes the user's map function.The resulting augmented key-value pairs are partitioned by key to produce a sequence of *progress-batches* for each partition (downstream reducer). A progress-batch consists of all tuples with the same progress-sync value (within the specific partition) and has a unique ID. Each progress-batch sequence is in strictly increasing progress-sync order. The input text reader appends an *end-of-file* (*eof*) marker to the mapper's input when it

PI	User	Ad
$[0,\infty)$	u_0	a_0
$[0,\infty)$	u_0	a_0
$[1,\infty)$	u_1	a_0
$[1,\infty)$	u_1	a_1
$[2,\infty)$	u_2	a_1
$[2,\infty)$	u_2	a_1
$[2,\infty)$	u_2	a_1
$[2,\infty)$	u_2	a_1

(a)

PI	User	Ad
$[0,\infty)$	u_0	a_0
$[0,\infty)$	u_0	a_0

PI	User	Ad
$[1,\infty)$	u_1	a_0
$[1,\infty)$	u_1	a_1

PI	User	Ad
$[2,\infty)$	u_2	a_1
$[2,\infty)$	u_2	a_1
$[2,\infty)$	u_2	a_1
$[2,\infty)$	u_2	a_1

(b)

PI	User	Ad
$[0,\infty)$	u_0	a_0
$[0,\infty)$	u_0	a_0
$[0,\infty)$	u_1	a_0
$[0,\infty)$	u_1	a_1

PI	User	Ad
$[1,\infty)$	u_2	a_1
$[1,\infty)$	u_2	a_1
$[1,\infty)$	u_2	a_1
$[1,\infty)$	u_2	a_1

(c)

Figure 7: (a) Input data annotated with PIs; (b) Progress-batches according to input data PI assignment; (c) Progress-batches with modified granularity using a batching function.

reaches the end of its input split. The mapper, on receipt of the *eof* marker, appends it to all progress-batch sequences.

Batching granularity. The batching granularity in the framework is determined by the PI assignment scheme (§ 2.5) of the input data. Now!, also provides a *control knob* to the user, in terms of a *parameterized batching function*, to vary the batching granularity of the map output as a factor of the PI annotation granularity of the actual input. This avoids re-annotating the input data with PIs if the user decides to alter the granularity of the progressive output.

Example 2 (Batching) *Figure 7(a) shows a PI annotated input split with three progressive samples. Figure 7(b) shows the corresponding batched map output, where each tuple in a batch has the same progress-sync value. Figure 7(c) shows how progress granularity is varied using a batching function that modifies P^+. Here, $P^+ = \lfloor \frac{P^+}{b} \rfloor$ is the batching function, with the batching parameter b set to* 2.

3.2.2 Progressive data shuffle

Now! shuffles data between the mappers and reducers in terms of progress-batches without sorting. As an additional performance enhancement, Now! supports a mode for in-memory transfer of data between the mappers and reducers with flow control to avoid memory overflow. We pipeline progress-batches from the mapper to the reducers using a *fine-grained signaling mechanism*, which allows the mappers to inform the job tracker (master) the availability of a progress-batch. The job tracker then passes the progress-batch ID and location information to the appropriate reducers, triggering the respective map output downloads.

The download mechanism on the reducer side has been designed to support progress-sync ordered batch movement. Each reducer maintains a separate blocking concurrent queue (BCQ) for each mapper associated with the job. As mentioned earlier, the BCQ is a lock-free in-memory data structure which supports concurrent enqueue and dequeue operations and enables appropriate flow control to avoid swamping of the reducer. The maximum size of the BCQ is a tunable parameter which can be set according to the available memory at the reducer . The reducer enqueues progress-batches, downloaded from each mapper, into the corresponding BCQ associated with the mapper, in strict progress-sync order. Note that our batched sequential mode of data transfer means that continuous connections do not need to be maintained between mappers and reducers, which aids scalability.

3.2.3 Progress-aware merge

Now! implements the Prism model using a *progress-aware merge mechanism* which ensures flow of data in progress-sync order along all paths in the framework. Figure 8 shows the high level design of the progress-aware merge module within each reducer. Once a map output is available in each of the map output queues, the reducer invokes the progress-aware merge mechanism the details of which

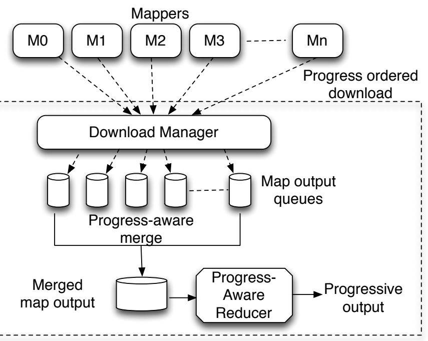


Figure 8: Progress-aware merge.

are given in Algorithm 1. The algorithm takes as input the number of mappers M, a set of BCQs $\mathcal{B}$ where $q_i \in \mathcal{B}$ denotes the blocking concurrent queue for mapper i, the current progress-sync value c_{min} of the merged batch that needs to be produced (c_{min} is initialized to the minimum progress-sync across the heads of the BCQs), and $\mathcal{H}$, where $h_i \in \mathcal{H}$ is the progress-sync value currently at the head of q_i (h_i is initialized to the progress-sync value at the head of q_i).

The algorithm initializes an empty set O as output. It iterates over all mapper queues to find and dequeue the batches whose progress-sync values match c_{min}, adds them to O and updates h_i to the new value at the head of q_i. It finally updates c_{min} and returns O, a merged batch with all tuples having the same progress-sync value. O is then fed to the progressive reducer. If $O = \emptyset$, indicating end of input on all BCQs, the framework passes an *eof* marker to the progressive reducer signaling termination of input.

Algorithm 1: Progress-aware merge

```
input  : # of Mappers $M$, $\mathcal{B} = \{q_1, \ldots, q_M\}$, $c_{min}$, $\mathcal{H} = \{h_1, \ldots, h_M\}$
output : Merged batch $O$
begin
    $O = \emptyset$;
    for each $q_i \in Q$ do
        if ($h_i$==$\infty$) then continue;
        progress-sync = peek($q_i$);  // peek blocks if $q_i = \emptyset$
        if (progress-sync==eof) then
            $h_i = \infty$; continue;
        $h_i$ =progress-sync;
        if ($h_i == c_{min}$) then
            $O = O \bigcup dequeue(q_i)$;
            progress-sync = peek($q_i$);
            if (progress-sync==eof) then $h_i = \infty$;
            else $h_i$ =progress-sync;
    $c_{min}$ = min($\mathcal{H}$); return $O$;
end
```

3.2.4 Progress-aware reducer

Let *partition* denote the set of keys that a particular reducer is responsible for. In traditional MR, the reducer gathers all values for each key in the partition and invokes a reduce function for each key, passing the group of values associated with that key. Now! instead uses progress-aware reducers whose input is a sequence of progress-batches associated with that partition in progress-sync order. The reducer is responsible for per-key grouping and computation, and produces a sequence of progress-batches in progress-sync order as output. We use the following API to achieve this:

```
Unchanged map API:
void map(K1 key, V1 value, Context context

Generalized Reduce API:
void reduce( Iterable<K2, V2> input, Context context)
```

Algorithm 2: Scheduling

```
input   : $R_f, R_o, M_i, M_d$, dependency table
begin
    for each $r \in R_f$ do
        Dispatch $r$;
        if Dispatch successful then Make a note of tracker ID;
    for each $r \in R_o$ do Dispatch $r$;
    for each $m \in M_d$ do
        Dispatch $m$, co-locating it with its feeder reducer;
    for each $m \in M_i$ do
        Dispatch $m$ closest to input data location;
end
```

Here, V1 and V2 include PIs. Now! also supports the traditional reducer API to support older workflows, using a layer that groups active tuples by key for each progress point, invokes the traditional reduce function for each key, and uses the reduce output to generate tuples with PIs corresponding to that progress point.

Progressive SQL While users can write custom progress-aware reducers, we advocate using an unmodified temporal streaming engine (such as StreamInsight) as a reducer to handle progressive relational queries (§ 2.4). Streaming engines process data in timestamp order, which matches with our progress-sync ordered data movement. Temporal notions in events can be reinterpreted as progress points in the query. Further, streaming engines naturally handle efficient grouped subplans using hash-based key partitioning, which is necessary to process tuples in progress-sync order.

3.3 Support for Multi-stage

We find that most analytics queries need to be expressed as multi-stage MR jobs. Now! supports a fully pipelined progressive job execution across different stages using concurrent job scheduling and co-location of processes that need to exchange data across jobs.

Concurrent Job Scheduling The scheduler in Now! has been designed to receive all the jobs in a multi-stage query as a job graph, from the application controller. Each job is converted into a set of map and reduce tasks. The scheduler extracts the type information from the job to construct a dependency table that tracks, for each task within each job, where it reads from and writes to (a blobs or some other job). The scheduler uses this dependency table to partition map tasks into a set of *independent* map tasks M_i which read their input from a blob/HDFS, and a set of *dependent* map tasks M_d whose input is the output of some previous stage reducer. Similarly, reduce tasks are partitioned into a set of *feeder* tasks R_f that provide output to mappers of subsequent jobs, and a set of *output* reduce tasks R_o that write their output to a blob/HDFS.

Algorithm 2 shows the details of how the map and reduce tasks corresponding to different jobs are scheduled[1]. First, all the reduce tasks in R_f are scheduled on slave machines that have at least one map slot available to schedule a corresponding dependent map task in M_d which would consume the feeder reduce task's output. The scheduler maintains a state of the task tracker IDs of the slave machines on which these feeder reduce tasks have been scheduled. Second, all the reducers in R_o are scheduled depending on the availability of reduce slots on various slave machines in a round robin manner. Third, all the map tasks in M_d are dispatched, co-locating

[1]If the scheduler is given additional information such as the streaming query plan executing inside reducers, we may be able to leverage database cost estimation techniques to improve the scheduling algorithm. This is a well studied topic in prior database research, and the ideas translate well to our setting.

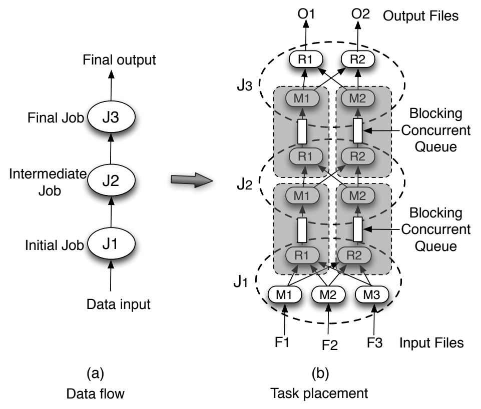


Figure 9: Multi-stage map reduce data flow.

them with the reducers of the previous stage in accordance with the dependency table and using the task tracker information retained in step 1 of the algorithm. Finally, all the map tasks in M_i are scheduled closest to the input data location. Placing tasks in this order ensures that if there exists a feasible placement of all MR tasks that would satisfy all job dependencies, we will find such a placement.

Data flow between jobs Figure 9 shows a sample placement of map and reduce tasks for processing a query that constitutes three jobs, J_1, J_2 and J_3. Figure 9(a) shows the data flow between jobs and Figure 9(b) shows the placement of map and reduce tasks as per Now!'s scheduling algorithm (Ref Algorithm 2). The shaded portions in the figure indicate that the corresponding map and reduce tasks have been co-scheduled on the same slave machine. The scheduler also verifies that the number of dependent map tasks are equal to the number of feeder reduce tasks of a preceding job, thus ensuring that there is one dependent map task for each feeder reduce task that is co-scheduled on the same slave machine.

Data flow between jobs is modeled on the producer-consumer paradigm using a BCQ and takes place completely in memory avoiding data materialization and shuffling overheads. Further, co-location of the reducers and mappers of dependent jobs does away with the overhead of data serialization, de-serialization and expensive network I/O between stages in a Cloud setting.

4. DISCUSSION AND EXTENSIONS

4.1 High Availability (HA)

Upadhyaya et al. [35] have recently shown how a multi-stage pipelined map-reduce system can support hybrid strategies of replay and checkpointing; these solutions are applicable in our setting. Specifically, the failure semantics for Now! are:

Map task failure: Any map task in progress or completed on a failed worker node needs to be rescheduled as in vanilla MR.

Reduce task failure: After a reduce task fails, one can replay its input starting from the last checkpoint (map output is materialized on local storage to allow replay). Interestingly, Prism can further reduce the cost of replay after a failure. The key insight is that processing at progress point p depends only on input tuples whose PIs are stabbed by p. We can leverage this in two ways:

- We can filter out tuples with $P^- \leq p$ during replay to significantly reduce the amount of data replayed and prune the intermediate map output saved on local storage[2].

[2]This optimization does not apply to external input which has P^- set to ∞, but can apply to intermediate results in multi-stage jobs.

- During replay, we can set P$^+$= max(p, P$^+$) for replayed tuples so that the reducer does not need to re-generate early results for progress points earlier than p.

Prior research [32] has reported that input sizes on production clusters are usually less than 100GB. Further, progressive queries are usually expected to end early. Therefore, Now! supports an efficient *no-HA* mode, where intermediate map output is not materialized on local storage and no checkpointing is done. This requires a failure to cascade back to the source data (we simply restart the job). Restarting the job on failure is a cheap and practical solution for such systems as compared to traditional long-running jobs. That said, we acknowledge that high availability with low recovery time (e.g., by restarting only the failed parts of the DAG) is important in some cases. Prior work [35, 38] has studied this problem; these ideas apply in our setting. We leave the implementation and evaluation of such fine-grained HA in Now! as future work.

4.2 Straggler and Skew Management

Stragglers A consequence of progress-sync merge is that if a previous task makes slow progress, we need to slow down overall progress to ensure global progress-sync order. While progress-sync order is necessary to derive the benefits of Prism, there are fixes that avoid sacrificing semantics and determinism:

- Consider n nodes with 1 straggler. If the processing skew is a result of imbalanced load, we can dynamically move partitions from the straggler to a new node (we need to also move reducer state). We may instead fail the straggler altogether and re-start its computation by partitioning its load equally across the remaining $n-1$ nodes. The catch-up work gets done $n-1$ times faster, resulting in a quicker restoration of balance [3].
- We could add support for *compensating reducers*, which can continue to process new progress points, but maintain enough information to revise or compensate their state once late data is received. Several engines have discussed support for compensations [6, 33], and fit well in this setting.

As we have not found stragglers to be a problem in our experiments on Windows Azure VMs, the current version of Now! does not address this issue. A deeper investigation is left as future work.

Data Skew Data skew can result from several reasons:

- Some sampling strategies encoded using PIs may miss out on outliers or rare sub-populations within a population. This can be resolved using stratified sampling which can be easily implemented in Prism as discussed in Section 2.5.
- Skew in the data may result in some progress-batches being larger than others at the reducers. However, this is no different from skew in traditional map-reduce systems, and solutions such as [24] are applicable here.

Since skew is closely related to the straggler problem, techniques mentioned earlier for stragglers may also help mitigate skew.

5. EVALUATION

5.1 Implementation Details

Now! is written in C# and deployed over Windows Azure. Now! uses the same master-slave architecture as Hadoop [36] with JobTracker and TaskTracker nodes. TaskTracker nodes are allocated a fixed number of map and reduce slots. Heartbeats are used to ensure that slave machines are available. We modified and extended this baseline to incorporate our new design features (see Section 3) such as pipelining, progress-based batching, progress-sync merge, multi-stage job support, concurrent job scheduling, etc. Now! deployed on the Windows Azure Cloud platform, uses Azure blobs as persistent storage and Azure VM roles as JobTracker and TaskTracker nodes. Multi-stage *job graphs* are generated by users and provided to Now!'s JobTracker as input; each job consists of input files, a partitioning key (or mapper), and a progressive reducer. Although Now! has been developed in C# and evaluated on Windows Azure, its design features are not tied to any specific platform. For example, Now! could be implemented over Hadoop using HDFS and deployed on the Amazon EC2 cloud.

Now! makes it easy to employ StreamInsight as a reducer for progressive SQL, by providing an additional API that allow users to directly submit a graph of $\langle$key, query$\rangle$ pairs, where *query* is a SQL query specified using LINQ [34]. Each node in this graph is automatically converted into a job. The job uses a special progressive reducer that uses StreamInsight to process tuples. The Now! API can be used to build front-ends that automatically convert larger Hive, SQL, or LINQ queries into job graphs. Although the system has been designed for the Cloud and uses Cloud storage, it also supports deployment on a cluster of machines (or private Cloud). Now! includes diagnostics for monitoring CPU, memory, and I/O usage statistics. These statistics are collected by an instance of a log manager running on each machine which outputs these in the form of logs which are stored as blobs in a separate container.

5.2 Experimental Setup

System Configuration The input and final output of a job graph are stored in Azure blobs. Each Azure VM role (instance) is a large-sized machine with 4 1.6GHz cores, 7GB RAM, 850GB of local storage, and 400Mbps allocated I/O bandwidth. Each instance was configured to support 5 map slots and 2 reduce slots. We experiment with up to 75 instances in our tests[4].

Datasets We use the following datasets in our evaluation, with dataset sizes based upon the aggregate amount of memory needed to run our queries over them:

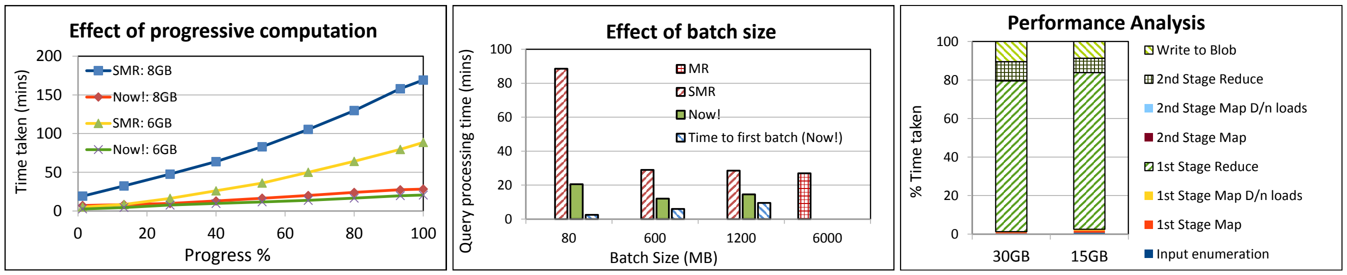
- *Search data.* This is a real 100GB search dataset from Bing, that consists of userids and their search terms. The input splits were created by sharding the data into a number of files/partitions, and annotating with fine-grained PI values.
- *TPC-H data.* We used the *dbgen* tool to generate a 100GB TPC-H benchmark dataset, for experiments using TPC-H queries.
- *Click data.* This is a real 12GB dataset from the Microsoft AdCenter advertising platform, that comprises of clicks and impressions on various ads over a 3 month period.

Queries We use the following progressive queries:

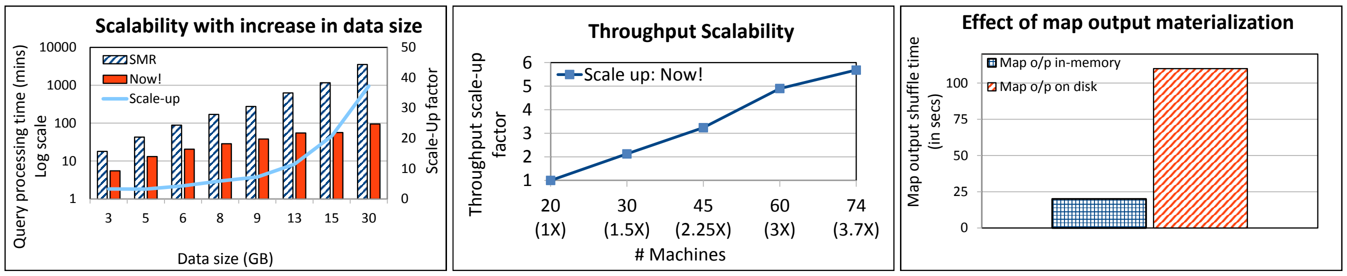
- *Top-k correlated search.* The query reports the top-k words that are most correlated with an input search term, according to a *goodness score*, in the search dataset. The query consists of two Now! jobs, one feeding the other. The first stage job uses the data set as input and partitions by userid. Each reducer computes a histogram that reports, for each word, the number of searches with and without the input term, and the total number of searches. The second stage job groups by word, and aggregates the histograms from the first stage, computes a per-word goodness, and performs top-k to report the k most correlated words to the input term. We use "music" as the default term.

[3]If failures occur halfway through a job on average, jobs run for $2.5/(n-1)$ times as long due to a straggler with this scheme.

[4]Our Windows Azure subscription allowed no more than 300 cores; this limited us to 75 4-core VM instances.



(a) (b) (c)



(d) (e) (f)

Figure 10: Performance analysis.(a) Time taken to process a query in progress-sync order; (b) Effect of batching granularity; (c) Analysis of time taken by different elements for a two-stage Map-Reduce query. Scalability: (d) Effect of data size on query processing time; (e) Throughput scalability with increase in #machines; (f) Overheads of disk I/O (Map output materialization).

- *TPC-H Q3.* We use a generalization of TPC-H query 3:

```
SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS
REVENUE, O_ORDERDATE, O_SHIPPRIORITY FROM ORDERS, LINEITEM
WHERE L_ORDERKEY = O_ORDERKEY
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
```

- *CTR.* The CTR (click-through-rate) query computes the MR job graph shown in Figure 2 (our running example). It consists of three queries (Q_c, Q_i, and Q_{ctr}) where Q_c is a *click query* which computes the number of clicks from the click dataset, Q_i is an *impressions query* which computes the number of ad impressions from the impression data set and Q_{ctr} computes the CTR.

Baselines We evaluate Now! against several baseline systems:

- **Map-Reduce (MR).** For standard map-reduce, we use Daytona [13], a C# implementation of vanilla hadoop for Windows Azure. This baseline provides an estimate of time taken to process a partitioned query without progressive results.
- **Stateful MR (SMR).** Stateful MR is an extension of MR for iterative queries [5], that maintains reducer state across MR jobs. We use it for progressive results by chunking the input into batches, and submitting each chunk (in progress-sync order) as a separate MR job. Subsequent chunks use reducers that retain the prior job's state. For each chunk, we run each MR stage as a vanilla MR job. With multi-stage jobs, we process one chunk through all stages before submitting the next chunk to the first stage.
- **MRO [12].** MRO pipelines data between the mappers and reducers, but is unaware of progress semantics and does not use progress-sync merge at the reducers. This can lead to different nodes progressing at different speeds. We approximate MRO in Now! by replacing the progress-aware merge with a union[5].

Job configuration and parameter settings. The configuration for a two-stage job (with one job feeding another) is depicted as $M_1 - R_1 - M_2 - R_2$ where M and R represent the number of mappers and reducers and their subscripts (1, 2) represent the stage to which they belong (note that $R_1 = M_2$). A single stage job is depicted as $M_1 - R_1$. In our experiments, the number of mappers is equal to the number of input splits (stored as blobs). The number of reducers is chosen based on the memory capacity of each worker node (7GB RAM) and the number of mappers feeding the reducers.

5.3 Experiments and Results

5.3.1 *Effect of Progressive Computation*

We evaluate Now!'s performance vs. SMR in terms of time taken to produce progressive results. The first experiment (see Figure 10(a)) plots the time taken to run the top-*k* correlated search query which provides the top 100 words that were searched with "weather", in terms of progress-batches plotted in progress-sync order. The input data set was batched by the mapper into 75 progress-batches by Now!. For the SMR baseline, the data was ordered and split into 75 chunks (one per PI). Each chunk representing one PI, was processed as a separate MR job and the time taken taken for the same was recorded. Each point on the plot represents an average of five runs. We used datasets of two sizes (6 and 8GB). The experimental results show that Now! performs much better (6X improvement) than SMR, which processes each progress batch as a separate job and resorts to expensive intermediate output materialization, hurting performance, particularly in a Cloud setting. Also, the time taken for the first 50% of the progress batches is under 20mins as opposed to 105mins for SMR, for the 8GB dataset, highlighting the benefit of platform support for progressive early results.

5.3.2 *Effect of Batching*

We evaluate the performance of Now! for different progress-batch sizes and compare the same with SMR and MR. The MR baseline processes the entire input as a single batch. The granularity of batch size controls the number of progress batches. The dataset size used in this experiment is 6GB and the configuration is 94-26-26-4. The experiment shows the results for 3 different batch sizes: 80MB (75 batches), 600MB (10 batches) and 1200MB (5

[5]This baseline benefits from our other optimizations such as concurrent job scheduling, no sorting, and pipelining across stages.

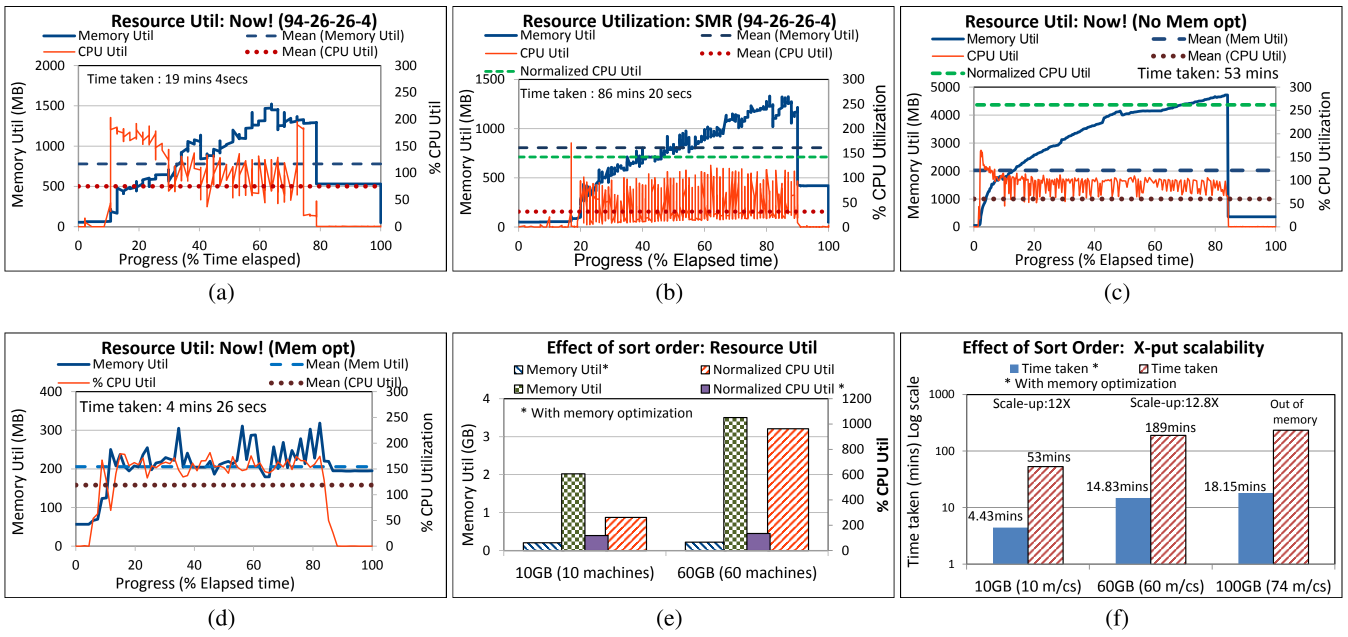


Figure 11: Resource Utilization. (a) CPU and memory utilization Now!; (b) CPU and memory utilization SMR;(c) CPU and memory utilization without memory optimization; (d) CPU and memory utilization with memory optimization. (e) Effect of sort order on memory and % CPU utilization for different data sizes; (f) Memory optimization effects on query processing time.

batches), and compares them against vanilla MR which processes the entire input of 6GB at once.

Figure 10(b) shows the change in total query processing time with change in batch size. As the batch-size decreases from 1200MB to 80MB, the number of batches processed by the system increases from 5 to 75. The query processing time of SMR increases drastically with the increase in the number of batches, which can be attributed to the fact that it processes each batch as a separate MR job and resorts to intermediate data materialization. The MR baseline which processes the entire input as a single batch does better than SMR , but does not provide early results.

On the other hand, the query processing time for Now! does not vary much with increase in number of batches as it is pipelined, does not start a new MR job for each batch, and does not materialize intermediate results between jobs. We do see a slight increase in query processing time when the number of batches increases from 10 to 75, which can be attributed to a moderate increase in batching overheads. However, the smallest batch-size provides the earliest progressive results and at the finest granularity. The figure shows the time to generate the first progress batch i.e., the time when the user starts getting progressive results. The time to first batch increases with increase in batch size (or progressive sample size), but is significantly lower than the total query processing time.

5.3.3 *Performance Breakdown*

We analyzed the performance of Now! using our diagnostic monitoring module which logs CPU, memory, and I/O usage. Figure 10(c) analyses the performance of the two-stage top-k correlated search query with $k = 100$, and plots the % time taken by different components in Now!. Each data point in the figure is an average over 10 runs, for two different datasets (15GB and 30GB) on 30 machines. The results indicate that the maximum time is spent in the first stage reducer followed by the second stage reduce and writing the final output to the blobs. The framework does not have any major bottlenecks in terms of pipelining of progress-batches. The time taken by the two reduce stages would vary depending on the choice of progressive reducer and the type of query. Our current results use StreamInsight as the progressive reducer.

5.3.4 *Scalability*

Figure 10(d) evaluates the effect of increase in data size on query processing time in Now! as compared to SMR. We used the top-k correlated search query for the experiment and varied the data size from 2.8GB to 30GB. The results show that Now! provides a scale-up of up to 38X over SMR in terms of the ratio of their query processing times. This can be attributed to pipelining, no sorting in the framework and no intermediate data materialization between jobs. Figure 10(e) shows the scale-up provided by Now! in terms of throughput (#rows processed per second) with the increase in #machines. For the top-k correlated search query (top 100 words correlated to "music"), we achieved a 6X scale-up with 74 machines as compared to the throughput on 20 machines, for 15GB data.

5.3.5 *Data Materialization Overheads*

Writing map outputs on the local disk, has a significant performance penalty, while on the other hand, intermediate data materialization provides higher availability in presence of failures . Figure 10(f) shows the overhead of disk I/O in materializing map output on disk and subsequent disk access to shuffle the data to the reducers within a job. Our results show an overhead of approx 90 secs for a dataset of 8GB for the 94-26-26-4 configuration.

Now! is tunable to work in both modes (with and without disk I/O) and can be chosen by the user depending on the application needs and the execution environment. It is also pertinent to note here that there is no data materialization on persistent storage (HDFS or Cloud) between different Map-Reduce stages in Now! which provides a similar performance advantage for multi-stage jobs over MR/SMR as seen in section 5.3.1.

5.3.6 *Resource Utilization*

We evaluated Now! for its resource utilization in terms of memory and CPU. Figures 11(a,b) compare the memory and CPU utilization of Now! and SMR for the 94-26-26-4 configuration for a dataset size of 8GB. The figures show the average real time memory and CPU utilization over 30 slave machines each running 4 mappers and 1 reducer plotted against time. The results indicate that there is no significant difference in the average memory utilization for both platforms, and the average CPU utilization of Now! is actually higher than that of SMR. However, we also show the *normalized* %CPU utilization for SMR which is the product of the average CPU utilization and the *normalization factor* (ratio of time taken by SMR to the time taken by Now!.) The normalized %CPU utilization is much higher as SMR takes approx 4.5X more time to complete as compared to Now!. Thus, Now! is ideal for progressive computation on the Cloud, where resources are charged by time.

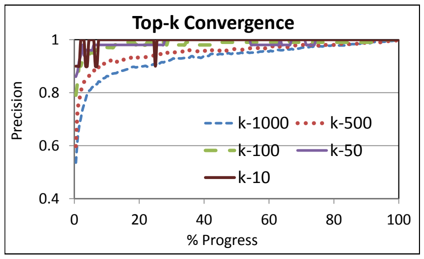
5.3.7 *Memory Optimization using Sort Orders*

The next experiment investigates the benefit of our memory optimization (cf. Section 2.6) in case the progress-sync order is correlated with the partitioning key. Our TPC-H dataset uses progress in terms of the L_ORDERKEY attribute, and TPC-H Q3 also partitions by the same key. An optimized run can detect this at compile-time and set $P^{-}=P^{+}+1$, allowing the query to "forget" previous tuples when we move to the next progress-batch. An unoptimized run would retain all tuples in memory in order to compute future join and aggregation results. We experiment with 10GB, 60GB and 100GB TPC-H datasets. Figures 11(c) and 11(d) show the variation of memory and CPU utilization with progress with and without memory optimization for the 10GB dataset. Figure 11(e) shows that the memory footprint of the optimized approach is much lower than the unoptimized approach, as expected. Further, it indicates that the lower memory utilization directly impacts CPU utilization since the query needs to maintain and lookup much smaller join synopses. Figure 11(f) shows that memory optimization gives an orders of magnitude reduction in time taken to process the TPC-H Q3 for all the three datasets providing a throughput scale-up of approx 12X in two cases (10GB and 60GB). As indicated in the figure, the 100GB run without memory optimization ran out of memory (OOM) as the data per machine was much higher.

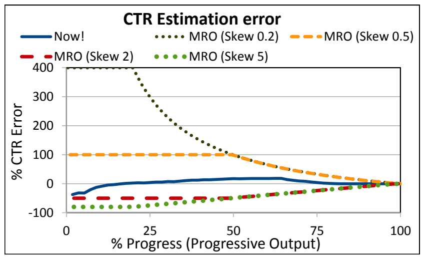
5.3.8 *Qualitative Evaluation*

Result Convergence In order to determine the speed of convergence we compute the precision (for the top-*k* correlated search query) of the progressive output values that we get as intermediate results. Figure 12(a) varies *k* and plots precision against the number of progress-batches processed for a data size of 15GB, with a configuration of 60-43-43-1 and 200 progress batches. The precision metric measures how close progressive results are to the final top-*k*. We see that precision quickly reaches 90%, after a progress of less than 20% as the top k values do not change much after sampling 20% of the data (lower *k* values converge quicker as expected). This shows the utility of early results for real-world queries where the results converge very quickly to the final answer after processing small amounts of data.

Progress Semantics We compare result quality against an MRO-style processing approach using the clicks dataset to compute CTR (Figure 2) progressively. We model variation in processing time using a *skew factor* that measures how much faster Q_i is, as compared to Q_c. A skew of 1 represents the hypothetical case where perfect CTR information is known a priori, and queries follow this relative processing speed. Figure 12(b) shows the % error in CTR estimation plotted against % progress. The experiment shows that if different queries proceed at different speeds, early results without user-defined progress semantics can become inaccurate (although all techniques converge to the same final result). We see that even moderate skew values can result in significant inaccuracy. On the other hand, progress semantics ensure that the data being correlated always belongs to the same subset of users, which allows CTR to converge quickly and reliably, as expected.



(a)



(b)

Figure 12: Qualitative analysis. (a) Top-k Convergence; (b) Error estimation of progressive results.

6. RELATED WORK

Approximate Query Processing Online aggregation was originally proposed by Hellerstein et al. [21], where the focus was on grouped aggregation with statistically robust confidence intervals based on random sampling. This was extended to handle join queries using the ripple join [17] family of operators. Specialized sampling techniques have been widely studied in subsequent years (e.g., see [9, 20, 30]). Laptev et al. [25] propose iteratively computing MR jobs on increasing data samples until a desired approximation goal is achieved. BlinkDB [2] constructs a large number of multi-dimensional samples offline using a particular sampling technique (stratified sampling) and chooses samples automatically based on a user-specified budget.

We follow a different approach: instead of the system taking responsibility for query accuracy (e.g., as sampling techniques) which may not be possible in general, we involve the query writer in the specification of progress semantics. A query processor using Prism can support a variety of user-defined progressive sampling schemes; we view prior work described above as part of a layer between our generic progress engine and the user, that helps with the assignment of PIs in a semantically appropriate manner.

MR Framework Variants Map-Reduce Online *(MRO)* [12] supports progressive output by adding pipelining to MR. Early result snapshots are produced by reducers, each annotated with a rough progress estimate based on averaging progress scores from different map tasks. Unlike our techniques, progress in MRO is an operational and non-deterministic metric that cannot be controlled by users or used to formally correlate progress to query accuracy or to specific input samples. From a data processing standpoint, unlike Now!, MRO sorts subsets of data by key and can incur redundant computations as reducers repeat aggregations over increasing subsets (see [7] for more details).

Li et al. [26] propose scalable one pass analytics (SOPA), where they replace sort-merge in MR with a hash based grouping mechanism inside the framework. Our focus is on *progressive* queries, with a goal of establishing and propagating explicit progress in the platform. Like SOPA, we eliminate sorting in the framework, but leave it to the reducer to process progress-sync ordered data. Streaming engines use efficient hash-based grouping, allowing us to realize similar performance gains as SOPA inside our reducers.

Distributed Stream Processing SPEs answer real-time temporal queries over windowed streams of data. We tackle a different problem: progressive results for atemporal queries over atemporal offline data, and show that our new progress model can in fact be realized by leveraging and re-interpreting the notion of time used by temporal SPEs. Now! is an MR-style distributed framework for progressive queries; it is markedly different from distributed SPEs [1] as it leverages the explicit notion of progress to build a batched-sequential data-parallel framework that does not target real-time data or low-latency queries. The use of progress-batched files for data movement allows Now! to amortize transfer costs across reducer per-tuple computation cost. Now!'s architecture is designed along the lines of MR with extended map and reduce APIs, and is designed for a Cloud setting.

Interactive Full-Data Analytics Dremel [29] and PowerDrill [18] are distributed system for interactive analysis of read-only large columnar datasets. Spark [37] provides in-memory data structures to persist intermediate results in memory, and can be used to interactively query big data sets or get medium-latency batchwise results on real-time data [38]. These engines have a different goal from us; by fully committing memory and compute resources a priori, they provide full results to queries on hot in-memory data in milliseconds, for which they use careful techniques such as columnar in-memory data organization for the (smaller) subset of data that needs such interactivity. On the other hand, we provide generic interactivity over large datasets, in terms of early meaningful results on progressive samples and refining results as more data is processed. Based on the early results, users can choose to potentially end (or possibly refine) computations once sufficient accuracy or query incorrectness is observed.

7. CONCLUSIONS

Data scientists typically perform progressive sampling to extract data for exploratory querying, which provides them user-control, determinism, repeatable semantics, and provenance. However, the lack of system support for such progressive analytics results in a tedious and error-prone workflow that precludes the reuse of work across samples. We proposed a new progress model called Prism that (1) allows users to communicate progressive samples to the system; (2) allows efficient and deterministic query processing over samples; and yet (3) provides repeatable semantics and provenance to data scientists. We showed that one can realize this model for *atemporal* relational queries using an unmodified *temporal* streaming engine, by re-interpreting temporal event fields to denote progress. Based on this model, we built Now!, a new progressive data-parallel computation framework for Windows Azure, where progress is understood and propagated as a first-class citizen in the framework. Now! works with StreamInsight to provide progressive SQL support over big data in Azure. Large-scale experiments showed orders-of-magnitude performance gains achieved by our solutions, without sacrificing the benefits offered by our underlying progress model. While we have studied the application of Prism to MR-style computation, applying it to other computation models (e.g., graphs) is an interesting area of future work.

8. REFERENCES

[1] D. Abadi et al. The design of the Borealis stream processing engine. 2005.

[2] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[3] M. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. 2009.

[4] B. Babcock et al. Models and issues in data stream systems. 2002.

[5] R. Barga, J. Ekanayake, and W. Lu. Iterative mapreduce research on Azure. In *SC*, 2011.

[6] R. Barga et al. Consistent streaming through time: A vision for event stream processing. 2007.

[7] B. Chandramouli et al. Scalable progressive analytics on big data in the cloud. Technical report, MSR. `http://aka.ms/Jpe5f5`.

[8] B. Chandramouli et al. Temporal analytics on big data for web advertising. In *ICDE*, 2012.

[9] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD*, 2004.

[10] S. Chaudhuri et al. On random sampling over joins. In *SIGMOD*, 1999.

[11] D. Cohn et al. Improving generalization with active learning. *Mach. Learn.*, 15, 1994.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[13] Daytona for Azure. `http://aka.ms/unkcbq`.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI'04.

[15] A. Doucet, M. Briers, and S. Senecal. Efficient block sampling strategies for sequential monte carlo methods. *Journal of Computational and Graphical Statistics*, 2006.

[16] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD 1999*.

[17] P. J. Haas and J. M. Hellerstein. Join algorithms for online aggregation. In *IBM Research Report RJ 10126*, 1998.

[18] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB July 2012*.

[19] M. Hammad et al. Nile: A query processing engine for data streams. 2004.

[20] J. M. Hellerstein and R. Avnur. Informix under control: Online query processing. *Data Mining and Knowledge Discovery Journal*, 2000.

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

[22] C. Jensen and R. Snodgrass. Temporal specialization. 1992.

[23] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. SIGMOD '07.

[24] Y. Kwon et al. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[25] N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on mapreduce. *PVLDB 2012*.

[26] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. J. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD 2011*.

[27] O. Maron et al. Hoeffding races: Accelerating model selection search for classification and function approximation. In *NIPS*, 1993.

[28] M. D. McKay et al. Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 21, 1979.

[29] S. Melnik et al. Dremel: interactive analysis of web-scale datasets. *PVLDB 2010*.

[30] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 2011.

[31] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. VLDB '99.

[32] A. Rowstron et al. Nobody ever got fired for using hadoop on a cluster. In *HotCDP*, 2012.

[33] E. Ryvkina et al. Revision processing in a stream processing engine: A high-level design. In *ICDE*, 2006.

[34] The LINQ Project. `http://aka.ms/rjhi00`.

[35] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *SIGMOD*, 2011.

[36] T. White. *Hadoop: The Definitive Guide*. 2009.

[37] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI'12.

[38] M. Zaharia et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.