Incremental Stream Processing using Computational Conflict-free Replicated Data Types

David Navalho
Sérgio Duarte Nuno Preguiça
CITI / DI - FCT - Universidade Nova de Lisboa

Marc Shapiro
LIP6 & INRIA

Abstract

Information has become a key commodity for most service providers. Analyzing streams of data efficiently, in real time, has become increasingly more important for supporting new products and applications.

This paper outlines a novel abstraction for performing incremental stream processing based on Computational Conflict-free Replicated Data Types. C-CRDTs are replicated objects that can be updated concurrently without coordination to perform a computation and still converge to a consistent state that reflects all contributions.

Results obtained with a preliminary prototype show that C-CRDTs have the potential to match and improve computational throughput when compared with a state of the art stream processing system.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Distributed programming, Parallel programming; E.1 [*Data Structures*]: Distributed data structures

Keywords Stream Processing, Distributed Processing, Incremental Computation, Real Time

1. Introduction

Information has become a key commodity for all major Internet players providing core services, such as search, social networking or e-commerce. Analyzing personal and business data is now common practice; doing so efficiently, in real time, is increasingly more important for supporting new products and applications.

Data processing solutions for cloud computing infrastructures, such as Map-Reduce [10] and DryadLINQ [20], have been designed for batch processing of bulk data. Effective for their primary goal, they proved inadequate for real-time processing. Latency can be improved by extending batch processing systems, by exposing partial computations and early results, as in (e.g. Map-Reduce Online [8]) or pursuing incremental models [4]. However, pitfalls remain that stem from operating over persisted bulk data and the penalty inherent to secondary storage.

In this paper we outline the foundations of Titan, a system for real-time, incremental data processing that we are currently building. Titan aims to tightly integrate computations with a decentralized storage system, striving for both performance and resilience, to ensure deterministic computations.We expect to achieve the low latency benefits of soft state based computations, with the fault tolerance guarantees of a backing store.

Titan is built on top of the Conflict-free Replicated Data Type (CRDT) abstraction [19]. CRDTs allow multiple replicas of the same data (structure) to be modified without coordination, guaranteeing that replicas can be merged later without need for any conflict resolution policy. This allows Titan to aggressively cache data in memory in multiple nodes for achieving high throughput, while resorting to secondary storage as a fallback for long term persistence.

Performing computations over soft state is one of the advantages of stream processing systems (e.g., Yahoo's S4 [15]), but resilience to faults becomes a major concern, as they impact consistency and determinism of ongoing computations. The use of CRDTs may help with providing deterministic and consistent fault tolerance.

Unlike existing alternatives, such as stream processing, that favor the execution of arbitrary application code, we want to capture much of the processing logic as a set of known operations over specialized Computational CRDTs, with particular semantics and invariants, such as min/max/average/median registers, accumulators, top-N sets, sorted sets/maps, and so on. Keeping state also allows the system to decrease the amount of propagated information.

Preliminary results obtained in a single example show that Titan has an higher throughput when compared with state of the art stream processing systems.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces compu-

CloudDP '13, April 14, 2013, Prague, Czech Republic.

tational CRDTs, while Section 4 present Titan's programming model. Section 5 overviews the system design. Section 6 presents a preliminary evaluation of the system and Section 7 concludes the paper with some final remarks.

2. Related work

Our work has been influenced by research from several areas, which we briefly survey.

MapReduce [10] is actively used by researchers and companies alike, in order to provide computation of massive amounts of data, with a simple programming model. In recent years, it became clear this model was insufficient for the increasing needs of social applications, improved user interactivity, or simply response time.

Many improvements to MapReduce have been proposed as a response, but one or more of the key problems usually remain: computations have to start and stop with a set amount of data, changes require recomputing everything again [21]; jobs are based on batches with a certain size, and even for incremental computation, one small change on a chunk of data requires recomputing the whole chunk and whatever chunks are associated with it [4]; in all cases, computations are intended as, at most, a single pipeline of transformations over a dataset [4, 8, 10, 21] and do not allow reuse of partial computations by multiple data flows; linking several computations requires work; adding more resources or simply new data or computations after being launched is impractical or impossible; and usually it requires some amount of complexity from the programmer to offer incremental computations [4].

More recently, some stream-based systems have been introduced in order to tackle the processing of always-incoming data. Systems such as Yahoo's S4 [15], Twitter's Storm [1] and StreamMine [5] present good approaches for stream-based computing at the large scale, although they each lack in important areas in order to provide scalable systems for true real-time computations.

None of the systems allow adding more nodes or resources after launch, which is a problem, especially when it is expected that participatory sensing systems are able to scale on demand (i.e., elastic). Additionally, we identify that some computations require faster response times, while others can work more similarly to a MapReduce computation, taking their time and less resources. S4 provides mechanisms for periodic messaging, instead of immediate. Storm doesn't take such an approach into account, and adding that functionality can only be done partially, and with effort.

Regarding failure recovery, none of the approaches are very good. Storm guarantees that messages are delivered at least once, but the user has to take that into account on his code - this can be very hard. In the case of S4, the solutions are to either keep going and assume data was lost, or fail the forward Nodes gracefully, and recompute every partial computation again (similar to Incoop [4]). StreamMine [5] presents a checkpoint-based approach for fault-tolerance, although it is unclear how it affects computation latency.

All of the approaches are stuck with the logical notion of a single computational flow, even if they allow multiple applications running in parallel on the system. We think there is a lot of repeated information and computation that may be interesting to reuse for other workflows.

Various solutions for handling concurrent updates have been previously used in storage systems [17]. With Titan, we intend to explore methods of using commutativity for parallel computation. We will take advantage of the novel concept of Conflict-free Replicated Data Types(CRDTs) [19]. CRDTs can guarantee that replicas converge without additional concurrency control, as long as all replicas receive all updates (either by receiving the operations or synchronizing state). Section 3 expands on this concept.

One of S4's intended objectives is to be science friendly: easy to program. We agree with this statement. We have recently taken an interest on social-scale computations (e.g, Participatory Sensing). There has been increasing research into what can be obtained and processed from sensor data, such as building location-based systems in the absence of GPS [3], road conservation and traffic monitoring [11, 12, 14], to extracting and interpreting information from user generated input, such as [6]. Not only that, but there has been increased interest in crossing information from multiple sources, in order to provide richer visualizations and information, that would otherwise not be possible [9].

In order to provide a framework that allows individuals from different backgrounds to create and test topologies at will, building applications that manage to feed off of each other, and where adding just an additional computational step is easy, some more work and engineering is necessary to eventually achieve this.

3. Computational CRDTs

Conflict-free replicated data types (CRDTs) [19] are data types that allow replicas to be modified without coordination, while guaranteeing that upon synchronization all replicas converge to the same state that considers all concurrently executed updates. There are two flavours of CRDTs, with different sufficient conditions for ensuring final consistency. In an operation-based model, all concurrent operations must commute. In a state-based model, the successive states must form a partial order, updates must always advance in the partial order, and merging states must compute a least-upper-bound. A large number of CRDTs have been designed in the past [18], including sets, maps, sequences, graphs, etc.

Computational CRDTs (C-CRDTs) are CRDTs that perform the result of a computation. Figure 1 presents a simple example: the state-based implementation of an object that keeps the result of computing a sequence of add and subtracts to the initial state of 0. The sum CRDT has a payload with two vectors, one for elements with positive value and

```
1: payload number[n] P, number[n] N          -- One entry per replica
2:     initial [0, 0, ..., 0], [0, 0, ..., 0]
3: update add (N)
4:     let g = myID()                        -- g: source replica
5:     P[g] := P[g] + N
6: update subtract (N)
7:     let g = myID()
8:     N[g] := N[g] + N
9: query value () : number v
10:    let v = Σ_i P[i] − Σ_i N[i]
11: compare (X, Y) : boolean b
12:    let b = (∀i ∈ [0, n − 1] : X.P[i] ≤ Y.P[i] ∧ ∀i ∈ [0, n − 1] : X.N[i] ≤ Y.N[i])
13: merge (X, Y) : payload Z
14:    let ∀i ∈ [0, n − 1] : Z.P[i] = max(X.P[i], Y.P[i])
15:    let ∀i ∈ [0, n − 1] : Z.N[i] = max(X.N[i], Y.N[i])
16: merge delta (X, Y) : payload Z
17:    let ∀i ∈ [0, n − 1] : Z.P[i] = X.P[i] + Y.P[i]
18:    let ∀i ∈ [0, n − 1] : Z.N[i] = X.N[i] + Y.N[i]
```

Figure 1. Sum CRDT (state-based)

the other one for elements with negative value. Each replica can independently modify its entry in the vectors, as add and subtract operations are executed. The result of the computation is the difference between the sum of add entries and the sum of subtract entries. On merge, the resulting state will include the maximum for each entry in both vectors.

This guarantees that the state of all replicas will evolve to the same state. The state of each replica includes the computed result of a sequence of updates, thus efficiently propagating the contributions of a a potentially large number of operations in a single synchronization operation.

Although the previous example is very simple, it is clear that the same approach could be used for a large number of other computations -e.g., a counter would include only the vector with positive values; for computing the average, the sum of elements and the number of elements would be kept; for computing top-N, the subset of top-N elements is kept; etc. For example, C-CRDTs can be created by leveraging previous work on the definition of decentralized algorithms for computing aggregation functions [2, 7, 8, 10, 13].

3.1 Hollow replicas

In the context of C-CRDTs, for efficiency purposes, it is often interesting to instantiate pristine replicas of C-CRDTs lacking remote state - we call these replicas hollow replicas. For example, while counting the number of words in a text stream, the application code that processes the stream wants to increase the sum of words whenever a new word appears. In this case, it is only necessary to accumulate the incremental part of the sum computation, without need of having access to the current value.

To this end, we use hollow replicas - a replica with no initial state. In this example, we would instantiate a hollow replica of the sum CRDT. Thus, a hollow replica accumulates and later transports the contribution of the stream processed by the application to the overall result. Merging a hollow replica with a full replica is done by a new merge function defined in C-CRDTs: the *merge delta* function. The merge delta for the sum C-CRDT defined in Figure1 consists in adding the value of each entry in the vectors. For a map, it would consist in: for new keys, adding them to the full replica; for existing keys, merging the sum CRDTs.

The state of a hollow replica can be synchronized with a full replica, thus being promoted to a full replica and maintaining the complete state of the object. In this case, it is still possible to accumulate and propagate the delta of the state to the other replicas (in Figure 1 we omit this code for simplicity). While hollow replicas perform a task similar to combiners in Map-reduce systems [10], keeping the full state of the replica can reduce the information that needs to be further propagated - e.g. in a top-N object, it is not necessary to propagate the local top-N elements that will not feature the global top-N.

In the next section we discuss how Titan combines C-CRDTs to express computations.

4. Programming model

The Titan programming model involves two main levels of algorithmic abstraction. One concerns the assembly of a pipeline of C-CRDTs that transforms the input data into the desired result, in as many intermediate steps as needed. The other pertains to selecting appropriate partitioning strategies for those C-CRDTs to lead to an efficient and balanced use of the available computational resources and potentiate the desired level of performance.

Titan is geared towards continuous, incremental processing, where intermediate results can be exposed to feed other computations or external consumer applications. Sharing common parts of computational pipeline and partial results among applications is transparent to Titan applications. To that end, Titan allows a new computation to reference the output of any pipeline stage of other instantiated computations as one of its inputs.

Chaining C-CRDTs to produce a complex computation in Titan is, first and foremost, centered on exploiting the specific computational semantics and invariants that are embodied in each C-CRDT type. For instance, a *set* besides storing values, also filters duplicates; whereas, a sorted set will extend that contract with an ordering invariant. As such, typical Titan application code will not explicitly perform operations such as testing for duplicates or sort data; likewise, if a computation step consists in a sum of numerical values, that result will be achieved via the inclusion of a *sum* C-CRDT in the computation pipeline, rather than explicitly resorting to the addition operator of the programming language.

In general, complex computations will be decomposed into a series of processing steps, where a partial result will feed the next stage of a processing pipeline, incrementally. The glue that connects the pipeline stages together and carries data from beginning to the end can be abstracted, as it is transparently provided at the system level. However, in most cases, the application is expected to provide the code that takes the output of a C-CRDT of a given type and adapts it to the input type of the next computation stage, such as splitting a sentence into its words before they can be individually added to the hollow replica of a set that is computing the unique words of a text stream in the next pipeline stage.

Decentralized and parallel processing is vital for speeding up Titan computations. The distributed nature of Titan computations arises lenghtwise in the processing pipelines, but also at each pipeline processing stage via partitioning of C-CRDTs. For C-CRDT types that support partitioning, typically containers such as sets and maps, partitioning is achieved by pairing each C-CRDT instance with a *map* function that projects its domain (e.g., the elements of a set) into one of several *logical partitions*. Via partitioning it is possible, for instance, to exploit spatial or temporal coherence to group and process logically related data in the the same node, when each logical partition is mapped to the actual physical partitions supported by the available computing nodes. Note that, unlike logical partitioning, mapping logical to physical partitions is automatic by default.

One important aspect that is reflected in the overall algorithmic logic of Titan computations is that C-CRDT partitioning embodies the *map-reduce* mechanism that is also found in other stream processing architectures. Partitioning implies a *reduce* step that in C-CRDT nomenclature is captured in the *merge* operation specific to each C-CRDT type. The C-CRDT merge operation is fundamental for combining the partial computations performed in the hollow replicas induced by C-CRDT partitioning and, in general, for pushing forth incremental updates through the computation pipeline. As such, when reasoning about the chained effects of the semantics of several C-CRDTs, the particular behaviour of merge operations involved merits close attention. The general rule is that the merge operation preserves the C-CRDT invariants, if needed via some built-in compensation logic. However, for more complex C-CRDTs, such as containers of C-CRDTs, merging may have far reaching side-effects as part of the invariant preservation, possibly producing cascading merge of other C-CRDTs. For instance, when instances of a C-CRDT map are merged, any colliding keys will cause their respective values to be merged. If those values are themselves C-CRDTs then another merge operation will be involved, and so forth. This behaviour can be exploited to perform complex computations, especially when matched with judicious partitioning of the computation pipeline C-CRDTs. To further illustrate the programming model, we next describe a couple of simple computation examples.

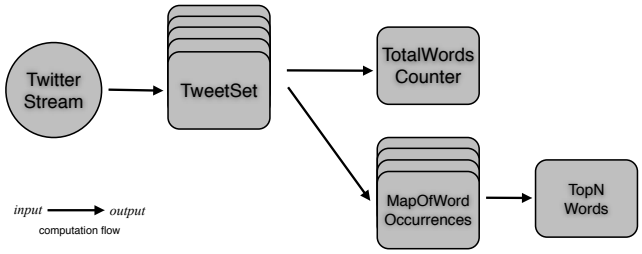

Figure 2. Example of two computations sharing part of the processing pipeline

4.1 Examples

Our two examples consist, respectively, in counting the total of words arriving in a Twitter stream and computing the top N most popular words, as determined by the number of occurrences. For this purpose, consider each individual tweet as message payload comprising a piece of text, made of words, accompanied by some metadata that includes an unique identifier. The computation is incremental and in realtime, meaning the values computed represent the results obtained so far. They will be continuously updated until the source stream runs out. A diagram representing the two computations is presented in Figure 2.

Counting Words Computing the total number of words in the stream begins with storing incoming tweets in a set (TweetSet). Any duplicates will be filtered out by the set semantics. To allow the computation to run in parallel, the TweetSet will be partitioned into a K fixed number of logical partitions. The partition function in this case simply maps the tweet identifier present in the metadata to a numerical value between 0 and K. Using hashing and modular arithmetic, we can expect to distribute tweets uniformly across the K partitions. The total number of words seen so far is produced in the next stage of the computation, which consists of a C-CRDT counter (TotalWords). As such, each of the TweetSet's K partitions will compute (independently) a subtotal from the tweets it stores; that subtotal will be accumulated in a local hollow replica of TotalWords. As the computation progresses TotalWords' hollow replicas are periodically merged with the primary, allowing the TotalWords C-CRDT to converge to the global total count.

TopN Words The most frequent words in a Twitter stream are found by counting the number of occurrences of each word. That is accomplished with a map, whose keys are words and its values C-CRDT counters (MapOfWordOccurrences). Just as before, this C-CRDT will be fed, independently, by the K partitions of TweetSet C-CRDT to populate local hollow replicas of MapOfWordOccurrences. Periodic merging of these hollow replicas will update the counters in the primary, taking advantage of the semantics of the map merge operation, which merges values of colliding keys. The next step in the computation is sorting the words and retain-

ing the top N with highest frequencies. This is done by the topN CRDT (TopNWords) - a sorted (value) map of a maximum fixed size, storing words as its keys and counters for values. However, computing a top requires sorting - an expensive operation best performed in parallel. To achieve that, MapOfWordOccurrences needs to be partitioned. Since updating or merging topN C-CRDT replicas (hollow or otherwise) can discard excess elements, it mandates that the partitioning scheme must not scatter words across different partitions, otherwise the global top-N computed from the partials produced at each MapOfWordOccurrences partition would not be correct. Partitioning MapOfWordOccurrences based on its keys (i.e., words) avoids the problem. As tweets enter the pipeline, they produce a cascading effect that eventually gets to the TopNWords hollow replicas and, consequently via periodic merge, update the primary.

5. Titan Overview

In this section, we briefly present an overview of the current Titan design. Titan is designed to run in a set of nodes in a single cluster. The nodes are organised in a one-hop DHT, which allows any pair of nodes to communicate directly.

Each physical node manages and stores and provides execution for a collection of C-CRDTs or C-CRDT partitions. The C-CRDT storage is versioned, maintaining for each C-CRDT, a sequence of versions. C-CRDT are partitioned according to the partition scheme defined by the programmer, upon instantiation. Logical C-CRDT partitions are mapped automatically to their physical counterparts, by hashing logical partition keys to DHT keys and selecting a node accordingly. Physical partitions are created or loaded on demand, as data arrives and hollow replicas perform their first merge with the primary C-CRDT.

An application installs a computation pipeline by creating and naming the desired C-CRDTs and providing the necessary gluing code. Typically, there will also be some client code that receives the data stream from an external source and starts the computation pipeline. As a means to consume computation results, client applications can request to have C-CRDT update notifications to be streamed to them, or perform an explicit read. In the example of Figure 2, this application would consist in consuming the Twitter stream and adding the new tweets to the TweetSet object, itself exposed as an hollow replica.

At system level, the programmer supplied code that glues C-CRDTs together and populates hollow replicas are used in *triggers*, associated with each C-CRDT instance. As in Percolator [16], *triggers* are the basis for incremental processing. They can run according to multiple policies, depending on how the C-CRDT was instantiated: whenever new data exists; whenever some condition on the data is true; or periodically. Each C-CRDT can have multiple triggers associated if it is involved in multiple computation pipelines. In the example of Figure 2, the TweetSet partitions have an underlying trigger that runs whenever a new tweet is available, which feeds the C-CRDTs that follow in the computational pipeline, by populating their respective hollow replicas.

6. Evaluation

In this section, we present a few experimental results with our preliminary and incomplete Titan prototype. We deployed a similar setup using Storm [1] to have some measure of comparability. We chose Storm instead of S4 [15] because it seems to have a very active user community, with much more frequent fixes and updates.

In the following sections, we will describe our example application and it's deployment strategy on both Titan and Storm. We then describe our testing environment and finally present our initial results using both systems.

Example Application: Word Count Our test application consists in counting the total words found in a Twitter stream, as outlined in Section 4.1. The Storm implementation followed its programming model. We used an initial set of *bolts* (Logical Nodes) for receiving and splitting TweetMessages; followed by several Spouts (Logical Nodes) to gather these words and count them. Finally, one last Spout gathers the word counts into a total word count. To provide a fair comparison, we made an effort to optimize the Storm application, by implementing bulk processing and finding the best deployment topology in our testbed.

Testing and Configurations We ran the experiments presented in this section in a cluster of 8 nodes, each one equipped with a Quad-Core AMD Opteron 2376 at 2.3Ghz, 4x512KB cache L2, and 8GB of RAM. The operating system is Debian 5.0.8, running Linux 2.6.26-2-amd64 kernel, and the nodes are interconnected via a private gigabit ethernet. The installed Java Platform was version 6 (OpenJDK, IcedTea6 1.8.10, 6b18-1.8.10-0 lenny2). For serving Twitter data, we had several clients preload Tweets into memory on three of the cluster computers. The remaining five were used to gather and process the data according to the configurations explained next.

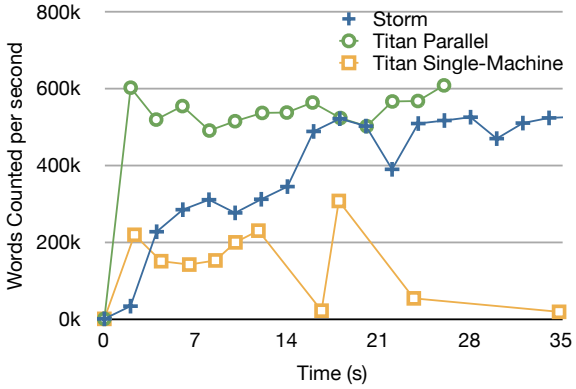

Figure 3. Counting Words experimental results

We ran several Storm topologies, with varying configurations. For the purpose of this article, we present the best configuration we found for running the Storm topology, with 4 initial nodes (spouts) for splitting the incoming tweets into words, 4 nodes (bolts) counting word occurrences, and a final node (bolt) that summed everything. Under this setup, Storm was gathering 100 Tweets per read from the clients.

Regarding the Titan experiment, we present two different runs: one similar to Storm, as described before, using the 5 machines for running Titan computations, and a second configuration using a single machine for Titan computations.

The results from these experiments are found in Figure 3. Despite the preliminary nature of our results and the simplicity of the example, the outlook is promising. Parallel Titan's throughput is both higher and more consistent than Storm's, reaching its maximum performance much earlier. Parallel Titan finishes first and compares well to the single-machine execution, taking advantage of decentralized processing.

7. Conclusions

This paper outlined Titan, a system aiming for real-time, incremental data processing, whose main abstraction, introduced in this paper, is the Computational CRDT.

C-CRDTs perform elementary computations and maintain specific invariants. As any other CRDT, they can be replicated in any number of nodes and modified without coordination, while being guaranteed that replicas can be merged automatically with all updates being considered.

Titan can be seen as a decentralized storage and execution environment for C-CRDTs, supporting computations defined as a graph of C-CRDT nodes interfaced through small pieces of application code. Results obtained with our preliminary prototype are encouraging, showing an improvement in throughput when compared to Storm.

Titan is still under active development. Our future work will focus on dynamic and multi-partitioning of C-CRDTs, resilience to faults and improving persistent storage integration in the system.

Acknowledgments

We sincerely thank our anonymous reviewers for their insightful comments and suggestions. This research has been supported by CITI grant PEst-OE/EEI/UI0527/2011 and FCT/MEC projects PTDC/EIA-EIA/108963/2008 and PTDC/EEI-SCR/1837/2012.

References

[1] Storm -. http://storm-project.net (accessed Jan. 2013).

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications Magazine, IEEE*, 40(8):102–114, November 2002.

[3] P. Bahl and V. N. Padmanabhan. Radar: An in-building rf-based user location and tracking system. In *Proc. INFOCOM*, pages 775–784, 2000.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proc. SOCC '11*, pages 7:1–7:14, 2011.

[5] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and Low-Latency Data Processing with StreamMapReduce. In *Proc. CloudCom 2011*, pages 48 –58, Los Alamitos, CA, USA, 2011. IEEE.

[6] M. Cha, F. Benevenuto, H. Haddadi, and P. K. Gummadi. The world of connections and information flow in twitter. *IEEE TSMC, Part A*, 42(4):991–998, 2012.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proc. 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, 2010.

[9] J. Cranshaw, R. Schwartz, J. I. Hong, and N. M. Sadeh. The livehoods project: Utilizing social media to understand the dynamics of a city. In *Proc. Sixth International Conference on Weblogs and Social Media*, 2012.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, 2004.

[11] J. Eriksson, L. Girod, B. Hull, R. Newton, S. Madden, and H. Balakrishnan. The Pothole Patrol: Using a Mobile Sensor Network for Road Surface Monitoring. In *MobiSys '08*, 2008.

[12] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. K. Miu, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys*, November 2006.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. 2nd EuroSys European Conference on Computer Systems*, EuroSys '07, pages 59–72, 2007.

[14] Mohan, Prashanth and Padmanabhan, Venkata and Ramjee, Ramachandran . Nericell: Rich Monitoring of Road and Traffic Conditions using Mobile Smartphones. In *Proc. of ACM SenSys 2008*, November 2008.

[15] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proc. ICDMW '10*, ICDMW '10, pages 170–177, 2010.

[16] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proc. 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, 2010.

[17] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005. ISSN 0360-0300.

[18] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Convergent and commutative replicated data types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (104):67–88, June 2011.

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Proc. 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes on Computer Science*, pages 386–400, Oct. 2011.

[20] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, 2008.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI'12*, pages 2–2, 2012.