Adaptive Processing of User-Defined Aggregates in Jaql

Andrey Balmin
abalmin@us.ibm.com

Vuk Ercegovac
vercego@us.ibm.com

Rares Vernica*
rares.vernica@hp.com

Kevin Beyer
dr.beyer@gmail.com

Abstract

Adaptive techniques can dramatically improve performance and simplify tuning for MapReduce jobs. However, their implementation often requires global coordination between map tasks, which breaks a key assumption of MapReduce that mappers run in isolation. We show that it is possible to preserve fault-tolerance, scalability, and ease of use of MapReduce by allowing map tasks to utilize a limited set of high-level coordination primitives. We have implemented these primitives on top of an open source distributed coordination service. We expose adaptive features in a high-level declarative query language, Jaql, by utilizing unique features of the language, such as higher-order functions and physical transparency. For instance, we observe that maintaining a small amount of global state could help improve performance for a class of aggregate functions that are able to limit the output based on a global threshold. Such algorithms arise, for example, in Top-K processing, skyline queries, and exception handling. We provide a simple API that facilitates safe and efficient development of such functions.

1 Introduction

The MapReduce parallel data-processing framework, pioneered by Google, is quickly gaining popularity in industry [1, 2, 3, 4] as well as in academia [5, 6, 7]. Hadoop [1] is the dominant open-source MapReduce implementation backed by Yahoo!, Facebook, and others. IBM's BigInsights [8] platform is one of the enterprise "Big Data" products based on Hadoop that have recently appeared on the market.

A MapReduce job consist of two phases: map and reduce. In the map phase, input data is partitioned into a number of *splits* and a *map task* processes a single split. A map task scans the split's data and calls a *map function* for every record of its input to produce intermediate key,value pairs. The map task uses the intermediate keys to partition its outputs amongst a fixed number of *reduce tasks* and for sorting. Each reduce task starts with a *shuffle* where it copies over and merges all of its input partitions. After shuffle, the reducer calls a *reduce function* on each set of map output records that share the same intermediate key.

In order to provide a simple programming environment for users, MapReduce offers a limited choice of execution strategies, which can adversely affect performance and usability of the platform. Much of this rigidity is due to one key assumption of MapReduce that map tasks run in isolation and do not depend on each other. This *independent mappers* assumption allows MapReduce to flexibly partition the inputs, arbitrarily order their processing, and safely reprocess them in case of failures. This assumption also implies that only embarrassingly

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Work done at IBM Almaden Research Center.

parallel algorithms can run during the map phase. All other computation must be shifted to the reduce phase, which requires, potentially very expensive, redistribution of the data during the shuffle. Consequently, MapReduce users are often forced to choose less efficient embarrassingly parallel versions of the algorithms, even if much more efficient options are available that require modest communication between tasks.

In this paper we describe our Adaptive MapReduce approach that breaks the independent mappers assumption by introducing an asynchronous communication channel between mappers using a transactional, distributed meta-data store (DMDS). This store enables the mappers to post some metadata about their state and see state of all other mappers. Such "situation-aware mappers" (SAMs) can get an aggregate view of the job execution state to make globally coordinated optimization decisions. However, while implementing SAM-based features, one has to be careful to satisfy key MapReduce assumptions about scalability and fault tolerance, and not introduce noticeable performance overhead.

To externalize SAMs, we propose an API for developing user defined aggregates (UDAs) for the Jaql [9] query language. This API is powerful enough to allow efficient implementation of an interesting and diverse set of functions, ranging from Top-K to Skyline queries, sampling, event triggering, and many others. At the same time, it provides simple primitives to UDA developers, and transparently takes care of DMDS communication, ensuring fault-tolerance and scalability of the approach. Jaql's unique modularization and data heterogeneity features make the resulting UDAs applicable for a wide variety of scenarios.

In our prior work [10] we employed SAM-based adaptive techniques to enhance Hadoop with variable checkpoint intervals, best effort hash-based local aggregation, and flexible, sample based, partitioning of map outputs. Note that Hadoop's flexible programming environment enabled us to implement all of these adaptive techniques without any changes to Hadoop. We chose an adaptive runtime to avoid performance tuning, which is currently challenging [11]. Adaptive algorithms demonstrate superior performance stability, as they are robust to tuning errors and changing runtime conditions, such as other jobs running on the same cluster. Furthermore, adaptive techniques do not rely on cardinality and cost estimation, which in turn requires accurate analytical modeling of job execution. Such modeling is extremely difficult for large scale MapReduce environments, mainly for two reasons. First, is the scale and interference from other software running on the same cluster. Parallel databases, for instance, rarely scale to thousands of nodes and always assume full control of the cluster. Second, MapReduce is a programming environment where much of the processing is done by black-box user code. Even in higher-level query processing systems, such as Jaql, Pig [3], and Hive [4], queries typically include user-defined functions that are written in Java. That is why all these systems offer various query "hint" mechanisms instead of traditional cost-based optimizers. In this environment, the use of adaptive run-time algorithms is an attractive strategy.

One of the adaptive techniques in [10], called Adaptive Combiners, helps speed up generic UDAs by keeping their state in a fixed-size hash-based buffer in local memory. However, to preserve generality, we did not allow communication between UDA implementations. This prevented efficient implementation for a wide class of UDAs. A simple example of such a UDA is a Top-K query. Consider a query over a webserver click-log dataset that asks for the last million records. Unless the dataset is partitioned by time, the simplest way to execute this request in MapReduce is to produce the last million keys from every map task, and then merge these map output lists in a single reducer until one million results are produced. Running this job with a thousand map tasks will result in up to a billion records output by map tasks and copied by the reducer. Needless to say, this may be very expensive. Imagine if the map tasks were able to maintain the current one-millionth most recent time-stamp. It could be used to filter the map outputs, to minimize the number of false positives sent to the reducer. Smaller map outputs means quicker local sorts and less data for the reducer to shuffle and merge.

Of course, maintaining this one-millionth key *threshold* exactly will require a lot of communication. Recent studies [12, 13] show that it is possible to efficiently minimize the amount of communication between worker nodes for the Top-K queries and other similar algorithms. The high level approach is to use relatively few communications to maintain a small global state, e.g., in case of Top-K, a histogram of the keys processed so far. This state is used to compute an approximate threshold. In this case, most false positives could be filtered

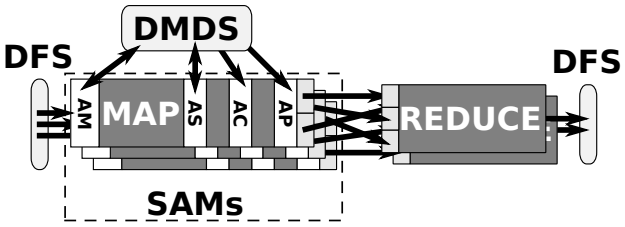

Figure 1: Adaptive techniques in SAMs and their communication with DMDS.

out from map outputs. Thus, Top-K algorithm could be much more efficient if map tasks were able to coordinate their efforts.

The same approach works for all other *thresholding* aggregation algorithms [12], which are able to filter out the data that requires aggregation, based on some global threshold. Examples of such algorithms include skyline queries, data sampling, and event triggering, among others. Thresholding algorithms do not fit a MapReduce paradigm as their efficient implementation require communication between map tasks. Allowing arbitrary communication amongst map tasks introduces potential dependencies which may lead to deadlocks and would prevent re-execution of tasks in case of failures.

In [10] we focused on system-level adaptive runtime options that could not be customized by the user. Thus, we could address fault-tolerance and scalability issues for each technique individually in an ad-hoc fashion. For each adaptive technique we implemented a unique recovery mechanism that allowed SAMs to be re-executed at any time. We also avoided synchronization barriers and paid special attention to recovery from task failures in the critical path.

Our experience with these techniques led us to believe that while SAMs are a very useful mechanism for system programming, the DMDS API is too low level for developers. Thus we have designed more straightforward APIs that could abstract out the details needed for fault-tolerance and performance. In this paper we describe one such API that enables developers to efficiently implement arbitrary thresholding functions in a very straightforward manner.

2 Background

2.1 Adaptive MapReduce

In our prior work we utilized SAMs in a number of adaptive techniques, some of which are already available in IBM's BigInsights product. Adaptive Mappers (AMs) dynamically "stitch" together multiple splits to be processed by a single mapper, thus changing the checkpoint interval, Adaptive Combiners (ACs) use best-effort hash-based aggregation of map outputs, Adaptive Sampling (AS) uses some early map outputs to produce a global sample of their keys, and Adaptive Partitioning (AP) dynamically partitions map outputs based on the sample. Figure 1 shows the adaptive techniques in the SAMs and their communication with DMDS.

The API proposed in this paper builds upon **Adaptive Combiners (ACs)** to perform local aggregation in a fixed-size hash table kept in a mapper, as does Hive and as was suggested in [14]. However, unlike these systems, once the hash table is filled up, it is not flushed. An AC keeps the hash table and starts using it as a cache. Before outputting a result, the mapper probes the table and calls the local aggregation function (i.e., *combiner*) in case of a cache hit. The behavior in case of a cache miss is determined by a pluggable replacement policy. We study two replacement policies: No Replacement (NR) and Least Recently Used (LRU). Our experimental evaluation shows that NR is very efficient - its performance overhead is barely noticeable even when map outputs are nearly unique. LRU overhead is substantially higher, however it can outperform NR, if map output keys are very skewed or clustered, due to a better cache hit ratio.

While ACs use an adaptive algorithm, their decisions are local and thus do not utilize SAMs. However, we employ SAM-based Adaptive Sampling technique, to predict if AC will benefit query execution and decide which replacement policy and cache size to use. Also, AMs further improve performance benefit of ACs as they increase the amount of data that gets combined in the same hash table.

All adaptive techniques mentioned above are packaged as a library that can be used by Hadoop developers through a simple API. Notice that the original programming API of MapReduce remains completely unchanged. In order to make the adaptive techniques completely transparent to the user, we also implemented them inside the Jaql [9] query processor.

2.2 ZooKeeper

One of the main components of our SAM-based techniques is a distributed meta-data store (DMDS). The store has to perform efficient distributed read and writes of small amounts of data in a transactional manner. We use Apache ZooKeeper [15, 16], an open-source distributed coordination service. The service is highly available, if configured with three or more servers, and fault tolerant. Data is organized in a hierarchical structure similar to a file system, except that each node can contain both data and sub-nodes. A node's content is a sequence of bytes and has a version number attached to it. A ZooKeeper server keeps the entire structure and the associated data cached in memory. Reads are extremely fast, but writes are slightly slower because the data needs to be serialized to disk and agreed upon by the majority of the servers. Transactions are supported by versioning the data. The service provides a basic set of primitives, like `create`, `delete`, `exists`, `get` and `set`, which can be easily used to build more complex services such as synchronization and leader election. Clients can connect to any of the servers and, in case the server fails, they can reconnect to any other server while sequential consistency is preserved. Moreover, clients can set watches on certain ZooKeeper nodes and they get a notification if there are any changes to those nodes.

2.3 Jaql

IBM's BigInsights [8] platform includes the Jaql system [9] to flexibly and scalably process large datasets. The Jaql system consists of a scripting language that features: 1) core operators that are based on relational algebra, 2) functions and higher-order functions for extensibility, modularity, and encapsulation, and 3) a data model that is based on JSON [17] to flexibly model the heterogeneity and nesting that often arises in big data. The Jaql compiler parses and transforms scripts into a DAG of MapReduce jobs for scalable evaluation.

Jaql's scripting language enables developers to blend high-level, declarative queries with low-level expressions that directly control the evaluation plan. This feature is referred to as *physical transparency* since it allows developers to judiciously design part of their script to directly access physical operators – either to tune performance or to exploit new operators. We exploit Jaql's *physical transparency* to expose Adaptive MapReduce functionality so that developers can efficiently implement a large class of aggregate functions. Since Jaql is a functional language, we also exploit (higher-order)functions to encapsulate the low-level details of Adaptive MapReduce so that such *adaptive* aggregates are accessible to a broader user base.

3 Adaptive Thresholding Functions

All adaptive thresholding functions in Jaql implement the following Java interface, which completely hide DMDS interactions, allowing the developer to concentrate purely on the semantics of the thresholding functions.

- `boolean filter(inVal)`: Compares the `inVal` against the current `threshold` structure, possibly using a comparator passed as a parameter to the Jaql thresholding function, and filters out the inputs that

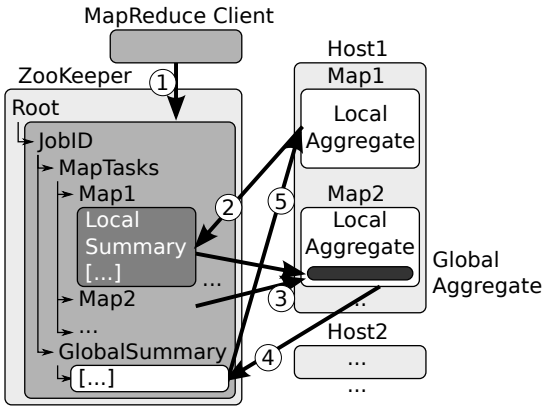

Figure 2: Communication between map tasks and ZooKeeper for a job computing an aggregate.

don't pass the threshold.

- `void aggregate(inVal)`: Combines inputs that pass the `filter()`, with an internal `accumulator` structure. This function is already implemented by Jaql's *algebraic* aggregate functions.
- `val summarize()`: Produces a `local summary` based on the current state of the aggregate function runtime, including the `accumulator`. If it returns a new summary, it is automatically written to ZooKeeper and eventually used to update the `threshold`.
- `val combineSummaries([val])`: Produces a global summary given a set of local ones. This function is called automatically every time some local summary changes,
- `val computeThreshold(val)`: Computes `threshold` structure given a global summary. This function is called automatically every time the global summary changes.

Jaql translates aggregate functions into a low-level expression called `mrAggregate()` that acts as a MapReduce client, directly submitting a MapReduce job which is parameterized by the appropriate map, combine (e.g., partial aggregate), and reduce functions. `mrAggregate()` invokes the functions listed above, hiding from the developer the low-level interactions with ZooKeeper that are shown in Figure 2. We describe this process in detail below.

Before the job is launched, a ZooKeeper structure is created for the job by the MapReduce client. The structure starts with a `JobID` node which contains two sub-nodes: a *MapTasks* node where mappers post local information, and a *GlobalSummary* node where the leader-mapper posts global information (Step 1 in Figure 2.)

Once the map task is running, `mrAggregate()` initializes the aggregate function runtime, and calls `filter()` for every map output record. If the result is *true*, it calls `aggregate()` and then `summarize()` for this record. If `summarize()` returns a non-empty local summary, the operator connects to ZooKeeper (or uses an already existing connection) and updates the *LocalSummary* corresponding to map task ID of the current task (Step 2 in Figure 2.)

One of the mappers is elected a *leader* using standard ZooKeeper techniques. We also use ZooKeeper to efficiently maintain a list of *potential leaders*, to facilitate quick recovery in case of a leader failure.

The leader uses the *watch* mechanism of ZooKeeper to keep track of updates to *LocalSummary* structures (Step 3 in Figure 2.) Once some local summary has been updated, subject to throttling rules that prevent the leader from becoming a bottleneck, the leader calls `combineSummaries()` with a set of all local summaries. If the resulting global summary is different from the one currently in ZooKeeper, the leader also calls `computeThreshold()` on the new global summary. The leader writes both global summary and the threshold

data structure to ZooKeeper, in their respective nodes under *GlobalSummary* (Step 4 in Figure 2.) The *GlobalSummary* is stored to expedite leader recovery. Otherwise, the new leader would have to recompute it during the recovery.

Every map task, during the initialization stage, sets a watch on the ZooKeeper node that stores the threshold. Thus, as soon as the leader updates the threshold in ZooKeeper, every map task gets notified and asynchronously copies the new threshold into their local aggregate data structure (Step 5 in Figure 2.)

In the remainder of this section, we illustrate how an adaptive version of the *Top-K* aggregate is implemented in Jaql.

Example 1: Consider a collection of web pages that has been parsed, hyper-links extracted, and the top 1000000 pages by their number of out links need to be selected for further processing. This can be easily specified using Jaql's `topN` aggregate function:

```
read( parsedPages ) -> topN( 1000000, cmp(x) [ x.numOutLinks desc ] );
```

The `read` function takes an I/O descriptor, `parsedPages`, as input and returns an array of objects, also referred to as *records*. The `parsedPages` descriptor specifies where the data is located (e.g., a path to a file) and how the data is formatted (e.g., binary JSON). In the example above, each record returned by `read` represents a parsed web page. One of the fields of each parsed page is `numOutLinks` and we want to obtain the top 1000000 pages according to the highest number of out links. The `topN` aggregate function takes an array as input, a threshold `k`, and a comparator, and returns an array whose length is $\leq k$ according to the ordering specified by the comparator. The input array is implicitly passed to `topN` using Jaql's `->` symbol, which is similar to the Unix pipe operator (`|`). The comparator is specified using a lambda function, `cmp`, whose input x is bound to each parsed page. Given two parsed pages, p_1 and p_2, $p_1 > p_2$ iff p_1.`numOutLinks`$> p_2$.`numOutLinks`.

Jaql transforms the above query into a single MapReduce job where the map phase reads the input, computes a partial Top-K aggregate, and the reduce phase aggregates the partial Top-K results into a final result. In the case of `topN`, a heap of size k is maintained that is then sent to the reducers for further aggregation. Since the example computes a global aggregate, all Top-K partial aggregates are sent to a single reduce task.

Note that each map task computes and sends its entire Top-K list to the reduce phase, which is conservative. With Adaptive MapReduce, a little global state allows us to be more aggressive so that substantially less data is sent to the reducers. Our Top-K aggregate function exploits Adaptive MapReduce by leveraging Jaql's physical transparency to control the MapReduce job and is packaged up as a Jaql function, *amrTopN*, so that its usage is identical to the `topN` aggregate:

```
read( parsedPages ) -> amrTopN( 1000000, cmp(x) [ x.outLinks desc ] );
```

The `amrTopN` can be implemented by slightly modifying `topN` as follows. The function runtime keeps a single key, such as `outLinks` in the above example, as the threshold. `filter(x)` simply calls `cmp(x,threshold)`, to purge all values (above)below the threshold. `aggregate(x)` remains the same as in `topN` - it still maintains its Top-K heap accumulator. `summarize()` returns the current (upper)lower-bound, in case a new one has been fund by the last `aggregate()` call. If a new local (upper)lower-bound is produced, ZooKeeper will notify the leader, who will invoke `combineSummaries()` to compute min(max) of all the local (upper)lower-bounds. That min(max) is both the global summary and a threshold in this case, thus `computeThreshold()` is the identity function. Once this global (upper)lower-bound is updated, all map tasks are notified of the change, so they update their local threshold and start using the new one for their `filter()` calls.

When the mappers complete, they sort all outputs and run the combiner function that does final local aggregation. In this case, the combiners will do some final purging using the latest global threshold. After that, map outputs, i.e., local Top-K aggregates will be sent to the reduce phase. Usually, mappers will discard much of their input due to threshold filtering and will send over less data to the reducer.

We intentionally used a very basic implementation of `amrTopN` above for demonstration purposes. A more efficient implementation uses a small (e.g., $log_2(k)$ points) histogram of the top k keys as local state. `combineSummaries()` creates a single histogram by combining all the local ones, and `computeThreshold()` produces the max threshold that it can guarantee to be outside top k keys produced so far *globally*. For example, for $k = 1000$, we'll keep 10-point local histograms, each consisting of v_{100}, v_{200}, ..., v_{1000}, where v_n is the current n-th top value. Having 10 local histograms, each with $v_{100} \geq N$, guarantees that the global threshold is at least N.

This approach is a generalization of the basic algorithm, which essentially always uses a histogram of size one. It is obviously going to produce thresholds that are at least as tight as those of the basic algorithm, and in practice, provide much better filtering for a very minor increase in communication.

Group-by queries. Up to now we considered top-level aggregate queries, which computed a single instance of a thresholding aggregate function. However, aggregate functions are often used in `group by` queries, where records are partitioned by some criteria and an aggregate is computed per partition. For example, suppose our collection of web pages also includes the top level domain of each page's URL, and that the top 1000000 pages by their number of out links need to be found, per domain. This can be done by the following Jaql query:

```
read( parsedPages ) -> group by g = $.domain into
  { domain: g, top: amrTopN( 1000000, cmp(x) [ x.outLinks desc ] )};
```

To implement this case we utilize our prior work in Adaptive Combiners (AC), in particular their Jaql implementation that enhances the `mrAggregate()` implementation with a cache of the most popular group keys along with their accumulator structures. To support thresholding functions, we use a similar fixed-size cache structure in ZooKeeper by creating a child node per group key under the *JobID* node. Each of these group nodes, contains child *MapTasks* and the *GlobalSummary*. However, we only allow up to a certain number of group nodes. If a group node is evicted (deleted) from ZooKeeper, its corresponding group may remain in local cache(s), where its local copy of the threshold will be maintained, but not synchronized with other mappers. This caching approach provides best-effort performance improvement. Mappers will filter out as many false positives as possible, subject to the cache size constraints.

Fault Tolerance. We implemented a number of techniques to facilitate correct and efficient recovery from leader failure. As soon as the leader's ZooKeeper connection is interrupted, the next task on the list of "potential leaders" is notified and it becomes a leader. Since both local and global summaries are stored in ZooKeeper and are versioned, the new leader can quickly pick up where the last one left off.

Non-leader task recovery is handled transparently by MapReduce framework and does not impact other tasks, as long as thresholds are monotonic, as they are in all of our examples. We currently do not support non-monotonic functions, however recent work on the subject [12] takes a promising approach of implementing general thresholding functions using monotonic ones as a building block.

Skyline Computation. Support for Skyline queries is another interesting example of thresholding functions. They have been initially proposed in [18] and gained popularity due to their applicability in multi-criteria decision making. Given a d-dimensional dataset D, the skyline is composed of points that are not dominated by any other point, with respect with some specific preference on (e.g., min, max) on each dimension. Various algorithms have been proposed in the literature for computing the skyline of a dataset distributed over a set of nodes [19, 20]. Some of the previous work [19] assumes a specific partitioning (e.g., grid-based, angular) of the points over the nodes. In this work, we make no assumptions regarding the data distribution over the nodes. We implement the same thresholding function API as follows. Each mapper computes a local skyline and maintains the local summary of the most "promising" data points. More precisely, the local summary is composed of the k points with the highest domination probability, as defined in [20]. The global summary contains the skyline computed from the union of the points contained in the local summaries. The intuition behind this approach is to identify points that are likely to make the greatest impact, i.e., dominate and filter out the most inputs. These points are shared with other map tasks to greatly reduce the number of false positives they produce.

Adaptive Sampling Another example of thresholding function is data sampling, which we already explored in [10]. In *Adaptive Sampling (AS)*, local aggregates collect samples of map output keys and the leader aggregates them into a global histogram. AS uses a pluggable threshold function to stop sampling if a sufficient sample has been accumulated. The default for this "stopping condition" function is to generate k samples. AS enabled us to produce balanced range partitioning of map outputs, which are required by the global sort operator.

In [10] we evaluated performance of AS and found it to be very robust and vastly superior to currently popular "client-side" sampling, which essentially executes the job on the sample of the dataset using a single node. In our experiments, client-side sampling had overhead of up to 250 seconds. In contrast, AS is able to leverage the full cluster. In our experiments, sampling was done in 5-20 seconds, with only a minor fraction of that time attributed to AS overhead, since we overlap sampling and query execution.

4 Conclusions and Future Work

We have presented adaptive techniques for the MapReduce framework that dramatically improve performance for user-defined aggregates. We expect SAMs to become an important MapReduce extension. Advanced users can implement essentially system-level enhancements using this mechanism directly. We demonstrated one approach for exposing SAMs to less advanced developers in a safe and controlled fashion. We plan to continue extending the programming API of Adaptive MapReduce and provide further integration points with Jaql.

References

[1] Apache Hadoop, http://hadoop.apache.org.

[2] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, 2010.

[3] A. Gates et al., "Building a highlevel dataflow system on top of MapReduce: the Pig experience," *PVLDB*, vol. 2, no. 2, pp. 1414–1425, 2009.

[4] A. Thusoo et al., "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*, 2010, pp. 996–1005.

[5] Y. Bu and et al., "Haloop: Efficient iterative data processing on large clusters," *PVLDB*, vol. 3, no. 1, 2010.

[6] D. Jiang et al., "The performance of MapReduce: an in-depth study," *PVLDB*, vol. 3, no. 1, pp. 472–483, 2010.

[7] T. Nykiel et al., "MRShare: sharing across multiple queries in MapReduce," *PVLDB*, vol. 3, no. 1, pp. 494–505, 2010.

[8] "IBM InfoSphere BigInsights," http://www-01.ibm.com/software/data/infosphere/biginsights.

[9] K. Beyer et. al., "Jaql: A scripting language for large scale semistructured data analysis," in *VLDB*, 2011.

[10] R. Vernica, A. Balmin, K. Beyer, and V. Ercegovac, "Adaptive mapreduce using situation-aware mappers," IBM, Tech. Rep. RJ 10494, 2011.

[11] S. Babu, "Towards automatic optimization of MapReduce programs," in *SoCC*, 2010, pp. 137–142.

[12] G. Sagy et. al., "Distributed threshold querying of general functions by a difference of monotonic representation," in *VLDB*, 2011.

[13] H. Yu et. al., "Efficient processing of distributed top-k queries," in *DEXA*, 2005.

[14] A. Shatdal et al., "Adaptive parallel aggregation algorithms," in *SIGMOD Conf.*, 1995, pp. 104–114.

[15] Apache ZooKeeper http://hadoop.apache.org/zookeeper.

[16] P. Hunt et al., "ZooKeeper: wait-free coordination for internet-scale systems," in *USENIX Conf.*, 2010.

[17] "JSON," http://www.json.org/.

[18] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001.

[19] G. Valkanas and A. N. Papadopoulos, "Efficient and adaptive distributed skyline computation," in *SSDBM*, 2010.

[20] B. Cui et. al., "Parallel distributed processing of constrained skyline queries by filtering," in *ICDE*, 2008.