Ad-Hoc Data Processing in the Cloud

Dionysios Logothetis
UCSD Department of Computer Science
dlogothetis@cs.ucsd.edu

Kenneth Yocum
UCSD Department of Computer Science
kyocum@cs.ucsd.edu

ABSTRACT

Ad-hoc data processing has proven to be a critical paradigm for Internet companies processing large volumes of unstructured data. However, the emergence of cloud-based computing, where storage and CPU are outsourced to multiple third-parties across the globe, implies large collections of highly distributed and continuously evolving data. Our demonstration combines the power and simplicity of the MapReduce abstraction with a wide-scale distributed stream processor, Mortar. While our incremental MapReduce operators avoid data re-processing, the stream processor manages the placement and physical data flow of the operators across the wide area. We demonstrate a distributed web indexing engine against which users can submit and deploy continuous MapReduce jobs. A visualization component illustrates both the incremental indexing and index searches in real time.

1. MOTIVATION

Ad-hoc data processing is a powerful abstraction for mining terabytes of data. Systems for massive parallel data processing, such as MapReduce [5] and Dryad [8], allow Internet companies, e.g., Google, Yahoo, and Microsoft, to mine large web crawls, click streams, and system logs across shared-nothing clusters of unreliable servers. While traditionally the job of parallel databases [6], these systems give programmers a familiar procedural interface to process unstructured data, and, since the data is often relatively transient, avoid the overhead of importing the data into a traditional RDBMS. Today, such systems manage parallel processing tasks across tens of thousands of machines in a single datacenter.

To date, ad-hoc data processing systems process data located at a single site in a snap-shot fashion. However, current trends in distributed computing are challenging these assumptions. For example, the *cloud-based* compute model allows one to acquire computation, storage, and network connectivity from a disparate collection of remote infrastructure providers. Such resource providers, such as Amazon's S3 and EC2 web services or Akamai's EdgeComputing platform, allow companies to dynamically out-source CPU and storage to remote data centers. Recent performance and availability studies of Amazon's offerings have shown respectable performance [7]. Indeed, a growing body of small to medium-sized companies now forgo the costs of managing computing resources in-house [1]. Similarly, larger companies may simply manage many remote sites themselves, e.g., a big-box retailer recording local customer transactions across hundreds of locations.

An additional challenge is that the data is continuously updated and evolving across time. For example, servers at a single site generate system and event logs, and peers within a distributed application may write data to local storage, such as a distributed web crawler [3]. Because many of the ad-hoc processing tasks programmers use are amenable to incremental computation, there is a large opportunity to reuse previous computations and avoid materializing intermediate results. However, current ad-hoc data processing architectures require the programmer to explicitly break tasks into multiple jobs to build pipelines. And, as we show for MapReduce, even these can leave large opportunities for reuse on the table.

In this demo we address these challenges by implementing a popular ad-hoc data processing abstraction, MapReduce [5], over a distributed stream processor. We take advantage of the fact that MapReduce forces programmers to specify two separable, parallel phases: map and reduce. In many cases the entire MapReduce job can be performed as in-network aggregate computation. Thus instead of bringing data to a central location, the distributed stream processor orchestrates incremental map and reduce computations to process unstructured data distributed across the wide area.

We demonstrate the benefits of continuous, incremental MapReduce by building and querying a distributed web corpus across multiple sites. Beyond reducing the cost of downloading the web to a single location, distributed crawling provides increased opportunities for performing deep web indexing. For example, giving web publishers control over intranet crawling allows them to address the presence of dynamic content, access restrictions, emerging content types (e.g., video), and privacy concerns when creating indices.

For our demo, multiple crawlers explore disjoint areas of the web while a continuous MapReduce job builds each local index. The system transforms user queries into continuous, in-network, MapReduce tasks to query the global index. A visualization component illustrates the output of

PVLDB '08, August 23-28, 2008, Auckland, New Zealand

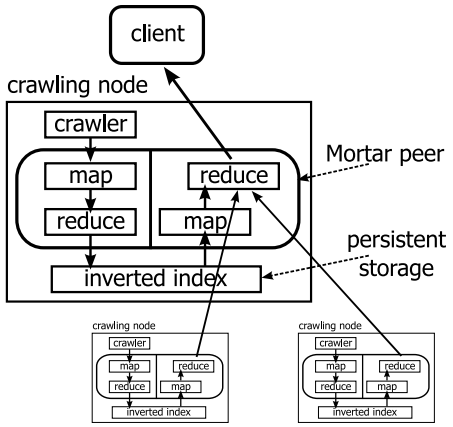

Figure 1: Incremental MapReduce producing a searchable inverted index across a set of distributed web crawlers. A client's continuous MapReduce job queries the index for a given set of keywords.

user queries, the rate of growth of the index across the system, and the impact of node failure on results. Experience with our prototype indicates that wide-area incremental MapReduce is a powerful technique for managing data in the cloud.

2. WIDE-SCALE DATA PROCESSING

This section gives an overview of the technical aspects of our demonstration. We review the MapReduce abstraction, and discuss the benefits of incremental MapReduce computations. We then describe the distributed stream processing platform, Mortar [9], used to create and manage the physical MapReduce dataflows. Finally we address the challenges and implications of adapting the MapReduce abstraction to a data stream model.

Figure 1 illustrates our demonstration scenario. Here three nodes distributed across the wide area run a distributed crawler, such as the Ubi crawler [3] that ensures each peer walks disjoint pieces of the web. Each node runs a Mortar peer that can install, execute, and remove continuous map and reduce operators. The Mortar peer feeds data from the crawler into an incremental MapReduce job, which in turn builds an inverted index that is written to persistent storage. The index maps keywords to the set of documents containing the keyword. Users query the system by writing a new continuous query, which Mortar executes as a continuous, in-network MapReduce job. This illustration shows a simple binary tree of reduce operators aggregating query results.

2.1 Incremental MapReduce

A MapReduce job splits data processing tasks into two phases: map and reduce. The map function operates on individual key-value pairs, $\{k_1, v_1\}$, and outputs a new pair, $\{k_2, v_2\}$. The system creates a list of values, $[v]_2$, for each key k_2. The reduce function then creates a final value v_3 from each key-value list pair. MapReduce implementations [5] transparently manage the parallel execution of the map phase, the grouping of all values with a given key (called the sort), and the parallel execution of the reduce phase. The system restarts failed tasks, gracefully dealing with machine failures

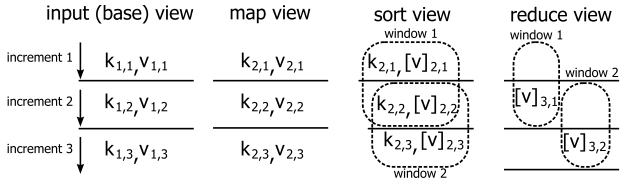

Figure 2: Incremental updates to views in MapReduce. Here the window range is two increments.

and ensuring reliable operation even when operating across thousands of cheap PC's. However, current MapReduce implementations process data in isolated snap-shots.

In contrast, we define an *incremental* MapReduce job as one that processes data in large batches of tuples. Like stream processing systems, an incremental MapReduce task runs continuously and, to bound processing, specifies a window range and slide or increment. The system produces a MapReduce result that includes all data within a window (of time or data size) every slide[1]. We also consider *landmark* MapReduce jobs where the trailing edge of the window is fixed (tuples never "expire"), and the system incorporates new data into the existing result.

Figure 2 illustrates an incremental MapReduce as a collection of three views, each corresponding to a phase in a single MapReduce job: map, sort, and reduce. From left to right, each phase transforms the preceding view into a new relation[2]. Time progresses from top to bottom, as the system processes arriving increments; the second index for each key or value signifies the increment of data that it represents. Here the window range is equal to two increments, suggesting that each may be re-used in the next window.

Note that Figure 2 shows materialized windows (the dotted boxes) in the sort and reduce phases. Basic materialized view maintenance in a RDBMS ensures that a cached realization of a common query remains valid as new tuples arrive. Efficient maintenance avoids re-evaluating the relational expression on all data when only a few base tuples change. Such incremental view maintenance depends on whether the relational function is distributive with respect to insert or delete operations [10]. A goal of this work is to provide transparent incremental update support for unstructured data kept in simplistic, opaque key-value relations.

In that respect, the MapReduce model makes it simple to reason about updates to the intermediate views relative to less structured systems, like Dryad [8]. A window of mapped data is a `union` of processed increments, and a window of sorted data may be created through a `groupby` operation. Because both operations are distributive aggregates with respect to insertion and deletion, the system can maintain these views in an incremental fashion independent of the user-defined map or reduce function. However, holistic reduce functions (as in this figure) require the system to recalculate the reduce output for each window from the materialized sort. Our current prototype delegates responsibility for efficiently updating those views to the user-defined reduce function.

[1]Here the terms increment, delta, and slide are interchangeable.

[2]Programmers may also specify a combiner function that runs as part of the map phase. For simplicity we do not show that intermediate view.

As a point of comparison, one may emulate incremental computation with traditional MapReduce by using several, pipelined MapReduce jobs. Such an approach may require running additional reduce tasks to combine the multiple intermediate outputs [4]. However, there are limitations to this high-level approach. First, intermediate results are always written to disk. Second, there is no obvious way to re-use the output of intermediate sorts, which is critical, as sort requires each of N reduce tasks to request data from each of M map tasks.

We address many of these issues by implementing map and reduce as continuous operators inside an in-network stream processor.

2.2 Distributed stream processing

We consider a distributed stream processor as the substrate for incremental MapReduce because such systems naturally support continuous operators, placement of operators across the wide-area networks, and in-network aggregate functions. In particular, we chose Mortar [9], a platform for instrumenting end hosts with user-defined stream processing operators. The platform manages the creation and removal of operators, and orchestrates the flow of data between them. By using Mortar we are free to focus principally on how to create efficient incremental map and reduce operators. Mortar takes care of arranging them in network-aware aggregation trees, routing data around failed nodes, and ensuring accurate processing in the face of unsynchronized clocks.

Mortar presents a simple API that facilitates programming sophisticated user-defined streaming operators. We distinguish between the operator, the code that supports the MapReduce abstraction, and the function, the programmer-supplied code. Our incremental map and reduce Mortar operators up call the programmer-defined map and reduce functions as new data arrives. Those function API's are nearly identical to those of the original MapReduce abstraction, making it easier to "port" existing MapReduce programs to the wide area. A key difference is that reduce functions may need to implement an appropriate "remove" operation to efficiently maintain the view for data exiting the window.

2.3 Continuous map and reduce

Unlike traditional MapReduce, stream-based map and reduce operators run continuously, processing arriving increments of tuples to update the current views without recomputing them from scratch. Map functions are trivially continuous, and process data on a tuple-by-tuple basis. However, before the reduce function may process the mapped data, the data must be partitioned across the reduce operators and sorted.

When the map operator first receives a new key-value pair, it up calls the map function and inserts the result into the latest increment in the map view. The operator then assigns output key-value pairs to reduce tasks, grouping them according to the partition function. Then the map operator participates in the sort, grouping values by the mapped keys, k_2. While traditional reduce tasks pull data from finished map tasks, our continuous map operator pushes each partition of the increment to the appropriate downstream reduce operator.

Continuous reduce operators participate in the sort as well, grouping values by their keys before up calling the reduce function. Both operators maintain the sort in nearly an identical fashion with respect to updates or removals. In both cases, materialization is simply a union across the constituent increments followed by a flatten operation. By keeping a small internal indexing structure, both operators allow O(1) removals from the mapped or sorted materialized view. Continuous reduce operators can integrate a new increment without re-reading, re-partitioning, and re-fetching the prior sorted data.

```
Indexing query:
MAP:      words = WordBreak(sites.WebPages) [tuple=1,slide=1]
REDUCE:   index = BuildIndex(words,T[0]) [time=10sec,slide=0]
```

Figure 3: The indexing query written in the Mortar Stream Language.

However, the reduce operator may require the programmer to supply an "un-remove" or deletion function. This is straightforward for many algebraic aggregates, such as `sum`. However, if deletion functions do not exist or are difficult to write, then our design re-computes the window output from the base data, the constituent slides. If the function is non-holistic and distributive with respect to insertions, this will be relatively efficient. If it is a holistic function, such re-computation is unavoidable.

2.4 Observations

In general there may be arbitrary sequences of map and reduce operators. The type of operations in such a sequence determines whether operators only need the last increment or the materialized window to produce the correct output. In our demo, when the indexing query integrates a new increment of crawled documents, our simple MapReduce search query needs the entire local index, not just the last changes, to produce the search result delta. However, both the index and search MapReduce jobs remain internally incremental. We discuss the query in detail in the next section.

For many applications including this one, partial results remain useful, and we allow individual map and stream operators to fail, potentially losing their data. This allows us to avoid sophisticated consistency protocols to tolerate node failures found in other stream processors [2], or support for a shared filesystem that makes reliably re-generating data after failure simpler for snap-shot MapReduce implementations. In contrast, Mortar makes no attempt to salvage data that was lost due to node or network failures. It does, however, reliably re-start (or re-install) operators when nodes recover. Note too, that continuous operation means the output of a MapReduce task can improve as nodes recover or connectivity improves. To take advantage of this, our distributed crawler rebuilds lost state, ensuring that query results eventually reflect the restored data.

3. DEMONSTRATION SCENARIO

This demonstration allows participants to interact directly with a distributed index engine based on continuous MapReduce jobs. Our prototype is a 30 node distributed web indexing application where each node runs a Heritrix crawler [3] and Mortar peer. A custom protocol, loosely based on prior work using consistent hashing [3], ensures disjoint crawl queues across the nodes.

```
Search query:
MAP:      search = QueryIndex(index) [tuple=1,slide=1]
MAP:      filter = RankResults(search) [tuple=1,slide=1]
REDUCE:  results = TopK(filter,50) [time=10sec,slide=0]
```

Figure 4: The search query uses a `TopK` reduce to rank results by word frequency.

We write the indexing and search queries in the Mortar Stream Language, a simple, text-based "boxes-and-arrows" language. Figure 3 illustrates the indexing query; Mortar compiles each line into a set of operators and installs them across the crawling nodes. First, Mortar installs an incremental MapReduce job on every node to build the inverted index. A `WordBreak` map function sources data (WebPages) fetched by the crawler and outputs (token, document location) pairs. The `BuildIndex` reduce function groups incoming pairs by word, generating an inverted index. A secondary sort in the function maintains a word occurrence count per document.

Users submit keyword queries through a web interface, and Mortar dynamically installs each user query across the crawlers. The system transforms each user search into a continuous query, shown in Figure 4. The first map operator subscribes to the inverted index at each node, executes a `QueryIndex` function to find keyword matches, and outputs pairs of (search keywords, results). The second map computes a rank using the word frequency present in the results, and sends its output to the reduce operator that implements an in-network `TopK` aggregate. The reduce outputs the top 50 results across the collection of inverted indices ordered by rank. Both queries use landmark windows, incrementally updating the index and search results with the latest data every ten seconds.

We run the system across an emulated wide-area network using ModelNet [11]. A ModelNet emulation tests real, deployable prototypes over unmodified, commodity operating systems and network stacks. A Mortar configuration running across the wide area (e.g., Planetlab) requires zero code changes to use ModelNet; the primary difference is that, in ModelNet, network traffic is subjected to the bandwidth, delay, and loss constraints of an arbitrary network topology. Our demo uses an Internet-like network topology created by the Inet topology generator.

We visualize the indexing system by animating the installation, processing, and removal of each user's MapReduce operators across the crawling nodes. Selecting a node displays information about its status, such as the number of user queries installed, the rate of the indexing, and the current size of the index. Upon query submission, a user observes (i) the progress of the query installation on the nodes and (ii) the streaming of the query results back to the web search interface, which dynamically updates until the user cancels the query.

4. APPLICABILITY

While our demonstration scenario leverages MapReduce computations that are both distributed and incremental, other applications may not benefit from both. Beyond cloud computing, a growing number of applications produce large waves of unstructured data, including sky surveys, medical imaging studies, and digital video surveillance applications. While these systems may benefit from an incremental processing abstraction, the benefits for distributed MapReduce will be greatest when the processing tasks are highly selective, avoiding transferring enormous amounts of data between sites. If the tasks do not sufficiently reduce the input data, or there are many queries, it may be more efficient to bring the data to a central location.

There remain significant research issues for future investigation. For instance, while it is likely that weakly consistent results are sufficient for an important class of applications, others may require increased guarantees of correctness. Though Mortar annotates individual operator results with the percentage of base data used to produce results (completeness), it is a weak metric on which to base accuracy. Another important goal is to develop techniques for orchestrating incremental computation across multiple non-distributive MapReduce tasks. However, even with these limitations, our demonstration indicates that this is a powerful approach for managing the increasing presence of unstructured data across the Internet.

5. REFERENCES

[1] Amazon's Cloud Storage Hiccups. http://www.wjla.com/news/stories/0208/496511.html.

[2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *Proc. of ACM SIGMOD*, Baltimore, MD, June 2005.

[3] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. In *Software: Practice & Experience*, October 2004.

[4] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *Proc. of ACM SIGMOD*, June 2007.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, San Francisco, CA, December 2004.

[6] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[7] S. Garfinkel. An evaluation for Amazon's Grid Computing Services: EC2, S3, SQS. Technical Report TR-08-07, School for Engineering and Applied Sciences, Harvard University, Cambridge, MA, July 2007.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, , and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, March 2007.

[9] D. Logothetis and K. Yocum. Wide-scale data sream management. In *Usenix Annual Technical Conference*, Boston, MA, June 2008.

[10] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *Proc. of 28th VLDB*, September 2002.

[11] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI'02*, Boston, MA, December 2002.