Data Indexing for Stateful, Large-scale Data Processing

Dionysios Logothetis, Kenneth Yocum
UC San Diego
9500 Gilman Dr.
La Jolla, CA
{dlogothetis, kyocum}@cs.ucsd.edu

ABSTRACT

Bulk data processing models like MapReduce are popular because they enable users to harness the power of tens of thousands of commodity machines with little programming effort. However, these systems have recently come under fire for lacking features common to parallel relational databases. One key weakness of these architectures is that they do not provide any underlying data indexing. Indexing could potentially provide large increases in performance for workloads that join data across distinct inputs, a common operation. This paper explores the challenges of incorporating indexed data into these processing systems.

In particular we explore using indexed data to support *stateful* groupwise processing. Access to persistent state is a key requirement for incremental processing, allowing operations to incorporate data updates without recomputing from scratch. With indexing, groupwise processing can randomly access state, avoiding costly sequential scans. While random access performance of current table-based stores (Bigtable) is disappointing, the characteristics of solid-state drives (SSDs) promises to make this a relevant optimization for such systems. Experience with an initial prototype and a simple model of system performance indicate that such techniques can halve job runtime in the common case. Finally, we outline the integration and fault tolerance issues this system architecture presents.

1. INTRODUCTION

Large-scale analysis of unstructured data has received a lot of attention recently. For instance, Internet companies, like Google, Yahoo and Microsoft, process terabytes of crawled webpages to generate search engine indices[6], or in the case of social networking sites, like Facebook, to data mine the user graph. Systems like MapReduce[6] and Dryad[9] have enabled such computations, leveraging the power of thousands of commodity PCs.

However, recent studies have raised concerns that programmers are sacrificing significant performance gains by opting for these data processing systems over parallel database systems [14]. For a number of processing tasks, parallel databases may outperform these systems by a large factor. A critical advantage of modern parallel databases is that they may store relations (or columns of) in separate partitions and in multiple sort orders, using multiple indices to optimize their query plans [17]. In contrast, systems such as MapReduce [6] or Dryad [9] default to reading all input data sequentially, no matter the particular processing step.

This paper lays the groundwork for a general approach to leveraging indexed data for groupwise processing in these systems. Groupwise processing is a core abstraction for these systems as it allows parallel data processing, and it underlies basic relational operations like groupby and join. However, current systems are oblivious to whether input data is indexed, and leveraging indices for efficient groupwise processing requires changes to their programming models and runtime systems.

We are currently exploring the use of indexed storage in the context of *bulk-incremental processing* systems (BIPS). We designed the BIPS processing model to support data-intensive applications whose input data arrives continually (e.g. web crawls, video uploads, etc.,). Like MapReduce, BIPS provides a groupwise processing operator to enable data parallelism, but a BIPS operator also has access to persistent state and can store partial results, indexed by a key, to minimize reprocessing as input data changes. Efficient state management is a fundamental aspect of performance in BIPS programs. Typically, applications modify only a fraction of the state, and performance depends on the ability to selectively access state.

While this work considers the use of scalable table-based stores, such as Bigtable [4], to allow random access to state, we find their performance disappointing. Random reads from table-based stores are still a factor of four to ten slower than sequential reads from commonly used distributed file systems (e.g., GFS[8]). Thus, random access is only beneficial when the operation against the table is sufficiently selective. Specifically, while we modified our BIPS implementation to store and retrieve state from Hypertable (an open-source Bigtable), running times only improved when the operation touched less than 15% of the state keys.

However, the importance of this optimization remained unclear. The Hypertable implementation is not mature (relative to published Bigtable performance numbers), and emerging solid-state storage technologies (SSDs) are rapidly improving random access performance. To better answer this question we developed a simple cost model to explore ex-

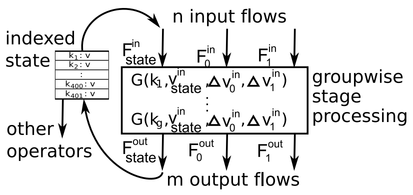

Figure 1: A single logical step of a BIPS dataflow.

```
G(url, $F^{in}_{state}$[], $F^{in}_{urls}$[])
1      newcnt ← $F^{in}_{urls}.size()$
2      if $F^{in}_{state}[0] \neq$ NULL then
3          newcnt ← newcnt + $F^{in}_{state}[0].cnt$
4      $F^{out}_{state}$.write({url, newcnt})
5      $F^{out}_{updates}$.write({url, newcnt})
```

Figure 2: Pseudocode for a BIPS processing step for maintaining counts of observed URLs.

pected benefits given the ratio of random to sequential data access performance. Using a simplified table store built directly above an Intel X25-E SSD, we predict that this optimization can significantly drop running times for a range of workloads. Finally, we report on how we handled the key prefetching and fault tolerance challenges such architectures present.

2. BACKGROUND AND MOTIVATION

This paper explores the use of data indexing for groupwise processing in unstructured processing systems. These popular large-scale data processing systems provide flexible but unstructured programming models. The downside to this approach is that the system is largely oblivious to how a user's query or dataflow manipulates data, meaning that most optimizations must be performed in an ad-hoc fashion by individual users.

Even so, most such systems include a groupwise processing construct because it enables data parallelism by allowing processing to proceed in parallel for each group. Groupwise processing underlies the join and groupby relational operations, as well as the *reduce* function in MapReduce[6]. Upper-layer languages, like Pig [13] and DryadLinq [20], also emulate similar relational operations with groupwise processing.

2.1 Stateful groupwise processing

We use and maintain indexed data as input to a groupwise operator in these non-relational systems by making the operators *stateful*. The index leverages the definition of the primary key, and sort and partition function that is defined with the groupwise operator.

We are currently exploring the use of such processing in the context of *bulk-incremental* processing systems or BIPS. The goal of BIPS is to allow for efficient incremental computation of bulk data that arrive periodically (e.g., weekly web crawls). In such applications, some processing steps operate over data in the current batch only (e.g., extracting new hypertext links), while other steps combine new data with data derived from previous batches (e.g., maintaining inlink counts). A basic requirement of incremental processing is state. Here, we focus on the features salient to optimizing stateful groupwise processing with indexed storage systems.[1]

In BIPS, each processing step or *stage* is a groupwise *operator* with multiple logical inputs and outputs that contain key-value pairs (or *records*), $\{k, v\}$. When an operator executes, it reads records from the inputs, sorts and groups by k, and calls a user-defined *groupwise function*, $G(\cdot)$, for each grouping key k. Critically, BIPS supports incremental functions by giving the function $G(\cdot)$ access to persistent state. We model state as an implicit, *loopback* flow, adding an input/output flow pair, F^{in}_S and F^{out}_S, to the operator. Figure 1 illustrates a groupwise processing stage with $n = m = 2$ and a loopback flow.

We call a single parallel execution of an operator an *epoch*, and it proceeds similarly to MapReduce. When the stage is ready to execute, the system partitions input data into R sets, and creates R replicas of the operator, assigning each a unique partition. Each replica operator performs a grouping operation on its partition (e.g., a merge sort), and then calls $G(\cdot)$ for each key. Since state is just another flow, it may also be subjected to these same steps each time the operator executes.

2.2 Efficient state management

Managing state efficiently is critical to high performance for an incremental system. The state is expected to become big relative to the other inputs, and a naive implementation that treats state as just another input will be grossly inefficient. For example, if one used MapReduce, it would blindly re-process, re-partition, and re-sort state. Instead, BIPS leverages the fact that state has already been partitioned and output in a sorted order during the prior invocation of the operator. Instead, it stores state in *partition files*. When an operator replica starts, it simply merge-sorts the state partition file with input.

Current unstructured systems read inputs in their entirety. MapReduce provides a *full-outer grouping*, calling the reduce function for every key present. BIPS, on the other hand, allows users to specify *right-outer groupings* with respect to state and all other inputs. Right-outer groupings only update state records that have matching keys from other inputs, allowing the system to avoid reading all state records. Unfortunately, even if a user specifies a right-outer grouping, a file-based approach will read and write the entire state during processing. As [14] illustrated, this can be a significant overhead.

2.3 Example

We illustrate BIPS with a single-stage dataflow that maintains the frequency of URLs seen on an input flow. This basic component is found in more sophisticated dataflows such as the one we developed to incrementally compute a web crawl queue [12]. This stateful processing stage has a single input flow that contains URLs extracted from a set of crawled web pages. The grouping key is the URL, the stage maintains state records of the type $\{url, count\}$, and the output is the set of URLs and counts that changed with the last input.

Figure 2 shows pseudocode for a groupwise function $G(\cdot)$ that performs a right-outer grouping with state. The system calls $G(\cdot)$ for each unique input key, passing the state and

[1]A more detailed discussion of BIPS, including the data model and example applications, can be found in [12].

input records that share the given key. It counts the new input records, and stores the updated count to state. The function also sends these state changes, the delta, to the output flow for downstream stages to use. Note that in this simple example writes to state are only for records assigned to this replica's partition. In fact, operators may write to keys "owned" by other partitions as well, but those writes are not visible until the next execution epoch.

3. GROUPWISE PROCESSING WITH INDEXED STATE

While the current BIPS implementation partitions and sorts state, it must still sequentially read the entire state even if the groupwise processing is extremely selective. This section explores the possibility of storing state in table-based storage systems, like Bigtable, that allow random access. We develop a simple model to predict the relative gain when using table-based storage over sequential scans. This allows us to reason about the impact future table-based storage architectures, such as those built over Solid State Disks (SSDs) with better random access performance, will have on this optimization.

3.1 Using table-based stores

The default BIPS implementation stores state partition files to distributed file systems, like HDFS, as a sequence of key-value pairs. Accessing a specific key-value pair forces the system to scan all preceding ones. Instead, storage systems like Bigtable appear to be a better match for our purposes. They provide a table abstraction, allowing applications to randomly access individual rows by a key. This reduces the amount of data accessed and may, therefore, result in reduced running time.

Ultimately, their efficacy depends on reading and writing matched keys randomly from the table faster than reading and writing all keys sequentially using a distributed file system. Published Bigtable performance figures indicate a four to ten times reduction in performance for random reads relative to sequential reads from distributed file systems like GFS[8].

To evaluate the efficiency of such systems in managing the state of BIPS operators, we modified the file-based BIPS implementation to store state records to Hypertable[1], an open source implementation of Bigtable. Instead of sequentially reading whole state partition files from HDFS, operator replicas issue random reads to Hypertable only for state records with matching keys in the input. After computing the new state values, they update the corresponding entries in the table.

We compared these two variants to identify the *break-even* hit rate, the hit rate below which the table-based implementation outperforms the file-based implementation. To do so, we ran the application described in Section 2.3 and varied the workload's *hit rate*, the fraction of state with matching keys in the input. We ran the experiments on a 16-node cluster consisting of Dual Intel Xeon 2.4GHz machines with 4GB of RAM, connected by a Gigabit switch. We pre-loaded the state with 1 million $\{item, count\}$ records (500MB). Subsequently, we processed a varying number of unique input records. Each input record has a matching record in state and, therefore, changing their number allows us to change the hit rate. For simplicity, we use one data partition and, hence, one operator replica.

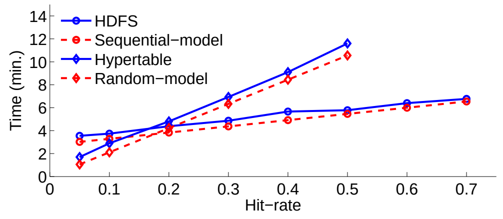

Figure 3: Experimental model verification. We compare file-based and table-based state storage. Solid lines correspond to our prototype experiments. Dotted lines show the predicted cost.

In Figure 3, the solid lines show the running time of the operator replica using both HDFS and Hypertable for storing state as a function of the hit rate. Because HDFS is an order of magnitude faster for retrieving records, using the table is only beneficial when the operator reads less than 17% of the state's keys. Thus, only workloads that update a small fraction of the state will see a performance improvement.

3.2 Predicting the benefits of random access

While these initial results were disappointing, could adopting systems with more mature implementations (Bigtable) or storage technologies with high random access throughput (SSD's) significantly improve performance? This section describes a cost model that predicts the gain in operator running time of random over sequential state access as a function of (i) the workload hit rate, and (ii) the relative performance of random to sequential reads of the underlying storage system.

Because BIPS performs grouping in parallel, we model the running time of a single operator replica. We assume the operator replica has access to state S with N_S records and one input I with N_I records. We use $N_{S \bowtie I}$ to denote the number of records in the state with matching keys in the input. The operator replica can access data sequentially or randomly at specific rates. We use R_{seq} (R_{ran}) and W_{seq} (W_{ran}) to denote the rate in records/sec at which the system can read and write data sequentially (randomly).

An operator groups state and input in two steps. First, it reads the input and sort-merges it with state. It then calls the operator function $G(\cdot)$ for every group, updating, removing or creating new state records. Here, we consider only the cost of accessing state and do not include the cost of computing the updated state. Therefore, the total cost T of the grouping is:

$$T = T_{read,I} + T_{sort,I} + T_{read,S} + T_{write,S}$$

When state is stored as a file, $T_{read,S}$ reflects the time to read N_S records at a rate R_{seq} and $T_{write,S}$ is the time to write all N_S records at a rate W_{seq}. When state can be accessed randomly, $T_{read,S}$ is the time to read $N_{S \bowtie I}$ records at a rate R_{ran} and $T_{write,S}$ is the time to write only the $N_{S \bowtie I}$ updated records.

A benefit of this simple model is that it only requires knowledge of read, write and sort performance rates and the size of input and state. We parameterized the model by measuring read and write rates of HDFS and Hypertable

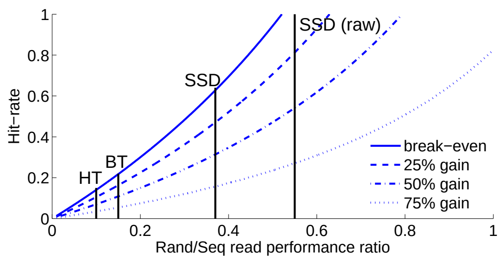

Figure 4: The hit rate required to achieve a given performance gain for different rand/seq read performance ratios. Each line corresponds to different % gain. The vertical lines correspond to the random-to-sequential performance ratios for Hypertable (HT), Bigtable (BT) and the observed performance of our SSD prototype.

on our 16-node cluster using 1 million $\{item, count\}$ state records. We parameterized the sort cost by measuring the time to sort all records on a single BIPS node. Figure 3 shows both our experimental results and the predicted cost as a function of the hit rate for each state access method. Each data point in the experimental lines is an average of 5 runs. We ran a variety of configurations (not shown) and the predicted cost is, on average, within 90% of the experimental.

The low break-even hit rate observed in Figure 3 is mainly attributed to the performance gap between random and sequential reads. The question that naturally arises is what the performance requirement is for table-based storage to be usable for a wide range of workloads. In Figure 4, we use our model to predict the hit rate (y-axis) below which table-based storage reduces running time by a certain percentage as a function of the random-to-sequential read performance ratio (x-axis). The different lines correspond to different performance improvements.

This graph allows us to infer the benefits of current and future systems. To do so, we draw vertical lines for the random-to-sequential performance of Hypertable (HT) and Bigtable (BT). While Bigtable, the state-of-art distributed table, closes the gap between random and sequential access performance, it still requires the workload to touch less than 20% of the state to see a 25% improvement in running time.

3.3 Fast random access with *SSDs*

Despite the relatively low random read performance in current systems, technological trends in storage devices are promising. Solid State Disks offer one to two orders of magnitude better random read throughput[15] than magnetic disks, closing the gap between random and sequential access performance. Recent work has proposed an architecture for more efficient clusters for data-intensive applications using SSDs [3].

Here, we explore the potential of a SSD-based system for indexed storage. We developed a proof-of-concept indexed storage system that allows selective retrieval of records using the random access capabilities of the file system. We benchmarked our prototype by measuring the time to sequentially read state and the time to randomly read records. The use of the SSD boosts the random-to-sequential read performance ratio to 37%. In Figure 4, this corresponds

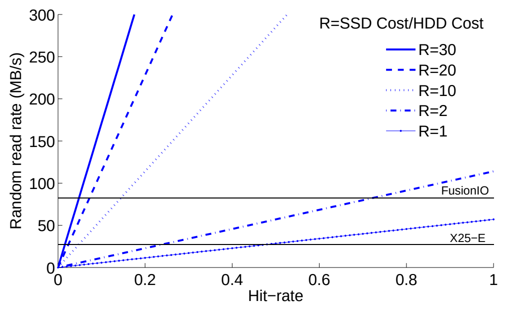

Figure 5: Required random read performance for SSDs to be as cost-efficient as disks. The different lines correspond to different ratios, R, of the cost per capacity between SSDs and disks.

to a 65% break-even hit rate, indicating that indexed storage may even benefit workloads that update the majority of their state. Note that the raw random-to-sequential ratio of our SSD, obtained from the manufacturer datasheet, is in fact 55%. We expect that with better file system support for SSDs, an indexed storage system will approach the raw performance, widening the range of workloads that benefit from indexed state.

Unfortunately, SSDs high random read rates come at considerable monetary cost; the cost per capacity of an SSD is up to 30 times higher than traditional disks [15]. However, recent studies have shown that SSDs can challenge the cost-efficiency of traditional disks [15, 3] in terms of cost per bandwidth. For random reads SSDs may deliver up to 50 times more bandwidth per dollar than disks [15].

Here we compare the cost-efficiency of a SSD-based system that can access a fraction of the state randomly to a disk-based system that accesses the whole state sequentially. We model cost efficiency as the cost of the system per performance delivered. The cost is defined as price per capacity or C. We measure performance in terms of job throughput. Using the notation from Section 3.2, if $T_{read,S}$ is the time to access state, then job throughput is equal to $1/T_{read,S}$. Note that unlike Section 3.2, here we compare the time to read N_S records sequentially on a disk-based system to the time to read $N_{S \bowtie I}$ records randomly on a SSD-based system. We can access state sequentially at a rate $R_{seq,HDD}$ and randomly at a rate $R_{ran,SSD}$. By setting the job throughput of the two alternatives equal to each other, we can then solve for the read rate of the SSD required to make SSD's as cost efficient as disks:

$$\frac{C_{SSD}}{R_{ran,SSD}/h} = \frac{C_{HDD}}{R_{seq,HDD}}$$

where $h = N_{S \bowtie I}/N_S$ is the hit rate.

Figure 5 shows the random read rate required for an SSD-based system as a function of the hit rate. We set the sequential read performance of the disk to 57MB/s, matching the faster disk benchmarked in [15]. Each line indicates a particular cost penalty for using SSD's versus disks, $R = \frac{C_{SSD}}{C_{HDD}}$. At this time $R = 30$, and the SSD rate must climb steeply as the hit rate increases to remain competitive with disk. Studies have shown that the SSD price per capacity is dropping at a 50% annual decline, faster than that of magnetic disks [10]. Within the next five years it is conceivable that

SSD's will cost about twice that of disks per unit storage ($R = 2$).

Using this graph we can estimate the hit rate at which SSD's become a cost-effective choice. To do so, we draw horizontal lines to indicate the random read performance of our SSD, the X25-E, and another enterprise-class SSD, the FusionIO device. These rates are obtained from the benchmarks in [15]. The point where the horizontal line intersects a particular cost ratio line R indicates the break-even hit rate randomly accessing state with an SSD is more cost efficient than sequentially accessing the whole state with the disk.

Today, even the high-performing FusionIO device is only cost effective when accessing less than 5% of the state. However, if SSD's drop to within a factor of two of disk cost (assuming SSD and disk performance remain constant), SSD's can become cost effective even when accessing a majority or 70% of the state. Thus, while an indexed storage system of a given size benefits from changing disks to SSDs, at current prices it may be better to simply increase the disk count. However, given price trends, we expect SSDs to become a viable alternative to disks for our workloads in the near future. While we have not considered it here, tiered architectures, where SSDs are used as caches or for specialized workloads, rather than long-term storage, may also improve their cost efficiency [11].

4. DESIGN ISSUES

As mentioned previously, these bulk processing systems typically read inputs in their entirety. Incorporating the ability to randomly access inputs (and outputs) affects the manner in which the system performs groupwise processing. While our BIPS prototype supports indexed inputs to store and retrieve state, these techniques apply generally to any groupwise inputs (or outputs) that can be accessed randomly. In particular, the system should pipeline random data retrievals while fetching the other "outer" inputs. At the same time, care must be taken to ensure that the data tables remain consistent during node failures.

4.1 State prefetching and caching

The efficiency of our optimization depends on promptly fetching state with matching keys in the input. While databases can use semantic information (e.g. a select predicate) during query optimization, to identify in advance which part of a relation to read, the unstructured nature of the data prevents BIPS from eagerly fetching state. As in the MapReduce *map phase*, BIPS extracts keys while processing input data. Thus the keys are known in their entirety only after processing all input.

Our design attempts to reduce the likelihood that operator replicas stall processing while fetching individual key records from state. To achieve this, operators learn the set of keys to fetch while sort-merging inputs (typically called the shuffle in MapReduce). When the system observes a threshold number of keys, it issues a batched read, allowing state retrieval to proceed in parallel with the sort of other inputs. The system places fetched values in a cache, limiting the memory footprint. In the common case, the operator replica finds the state values in the cache as it iterates across found keys; misses fault in the offending state from the table store.

4.2 Fault tolerance

Failures are common in data centers consisting of thousands of commodity PCs. Our system provides failure semantics similar to that of MapReduce and Dryad. Either all operator replicas of a stage complete and the output of the stage is the same as if there was no failure, or the stage fails. However, BIPS must additionally guarantee the consistency of operator state. In particular, after a stage execution, either all state updates are applied, or, in the case of a failed stage, the state is unmodified.

Typically, these systems handle machine failures by restarting failed replicas. Operator replicas write output to a private temporary directory that they move (rename) to the final output directory upon successful completion. They rely on the underlying distributed file system (e.g., HDFS) to atomically rename operator replica output. This allows the execution of *backup* (or speculative) executions of the same operator replica and ensures that the final output contains data by only one execution.

Using similar techniques for the state partition files would require BIPS to copy the data to the table. To avoid this copy, BIPS applies state updates directly to the table. However, since operator replicas can randomly access and modify only parts of state, failures can leave the system in an inconsistent state. Consider, for instance, an operator replica that fails after updating only part of the state. If the system simply restarts the replica, it will read an incorrect version of the state.

To provide consistent state snapshots, operator replicas label updated state values with a monotonically increasing version number that corresponds to the execution *epoch*. An operator that runs in epoch E labels all updated state values with version number E. Therefore, the version of a state value is equal to the epoch during which it was last updated. Notice that different state values can have different versions, since an operator may update only a fraction of the state during an epoch.

An operator replica that runs in epoch E will read the state value with the highest version number that is less than or equal to $E - 1$. This ensures that restarted replicas do not read updates written during the failed execution. Furthermore, because multiple backup replicas can update the same value, BIPS keeps the values written by only one successful backup replica. Note that current table-based stores like Bigtable and Hypertable can support such functionality through logical timestamps. However these systems only support simple range queries on timestamps; we are currently investigating simple extensions for supporting our queries efficiently.

5. RELATED WORK

The right-outer grouping of state and input in BIPS resembles a right-outer join operation. Map-Reduce-Merge[19] is an extension to the MapReduce abstraction that supports relational operations between heterogeneous datasets and can implement joins. In [2], the proposed computation model reduces the communication cost of joins relative to MapReduce. However, neither approach uses an index, resulting in scanning entire input relations.

Our system depends on the ability to selectively read state based on keys. There have been many variants of key-value stores, including Distributed Hash Tables (DHTs), like

Chord[16]. These systems focus mainly on providing a low latency key lookup service in large wide-area networks[16] or high-availability[7]. They are not designed for bulk data accesses and typically provide relaxed consistency guarantees. Although systems like Bigtable[4], Hypertable[1] and PNUTS[5] are better candidates for the underlying indexed storage, adopting them would replicate work (partitioning and sorting) our integrated design already performs.

Recent work has developed join algorithms to leverage the access characteristics of SSDs [18]. While focused on optimizing join algorithms in the context of PostgreSQL, their FlashScan algorithm also improves performance by leveraging random reads to avoid unnecessary data transfers. Interestingly, part of their improvements stem from using column-based layouts on SSDs. Such layouts may become beneficial for BIPS-like systems as query languages (Pig or DryadLinq) and the underlying execution systems become increasingly integrated.

6. CONCLUSION

Stateful large-scale data processing is a key step to providing the ability to incrementally process bulk data. However, efficiency dictates careful orchestration of processing to avoid full input scans and re-sorting/partitioning of state. While our initial prototype can provide significant decreases in running time when using indexed inputs, it was only for a narrow range of workloads. However, the performance characteristics of SSD's seem uniquely suited to this environment. This initial performance study suggests that adopting a robust, but simplified, table storage system over emerging SSD technology can reduce runtime by more than 50% in common incremental processing scenarios, while also being cost-effective in the near future.

Acknowledgements

This work benefited from numerous discussions with Chris Olston and Ben Reed, with whom the BIPS processing model was developed. This work is supported by the National Science Foundation through grant CNS-834784.

7. REFERENCES

[1] The Hypertable project. http://hypertable.org/.

[2] F. N. Afrati and J. D. Ullman. A new computation model for rack-based computing. In *PODS'09*, Providence, Rhode Island, June 2009.

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS'09*, Washington, DC, 2009.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Ch, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI'06*, Seattle, WA, November 2006.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!s Hosted Data Serving Platform. In *VLDB'08*, Auckland, New Zealand, August 2008.

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

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP'07*, Stevenson, WA, October 2007.

[8] S. Ghemawat, H. Gogioff, and S. Leung. The Google file system. In *SOSP'03*, Bolton Landing, NY, December 2003.

[9] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys'07*, Lisbon, Portugal, March 2007.

[10] J. Janukowicz, D. Reinsel, and J. Rydning. Worldwide solid state drive 2008-2012 forecast and analysis: Entering the no-spin zone. Technical Report 212736, IDC, June 2008.

[11] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. In *VLDB'08*, Auckland, New Zealand, August 2008.

[12] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Programming bulk-incremental dataflows. Technical Report CS2009-0944, UC San Diego, Computer Science dept., La Jolla, CA, 2009.

[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD'08*, June 2008.

[14] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD'09*, Providence, Rhode Island, June 2009.

[15] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *3rd Petascale Data Storage Workshop*, Austin, TX, November 2008.

[16] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM'01*, San Diego, CA, August 2001.

[17] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB'05*, Trondheim, Norway, 2005.

[18] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD'07*, Beijing, China, June 2007.

[19] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *SIGMOD'07*, Beijing, China, June 2007.

[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 *OSDI'08*, San Diego, CA, December 2008.