Automatic Optimization of Parallel Dataflow Programs

Christopher Olston, Benjamin Reed, Adam Silberstein, Utkarsh Srivastava

Yahoo! Research
{*olston, breed, silberst, utkarsh*}*@yahoo-inc.com*

Abstract

Large-scale parallel dataflow systems, e.g., Dryad and Map-Reduce, have attracted significant attention recently. High-level dataflow languages such as Pig Latin and Sawzall are being layered on top of these systems, to enable faster program development and more maintainable code. These languages engender greater transparency in program structure, and open up opportunities for automatic optimization. This paper proposes a set of optimization strategies for this context, drawing on and extending techniques from the database community.

1 Introduction

There is significant recent interest in parallel dataflow systems and programming models, e.g., Dryad [19], Jaql [17], Map-Reduce [9], Pig [22] and Sawzall [23]. While the roots of this work date back several decades in the database, programming language and systems communities, the emergence of new and well-funded application areas requiring very large-scale parallel processing is driving this work in somewhat different directions than in the past. In particular, recent work concentrates on much larger-scale systems, simpler fault-tolerance and consistency mechanisms, and stylistically different languages. This work is leading to a new computing paradigm, which some have dubbed *Data-Intensive Scalable Computing* (DISC)[1] [8].

The DISC world is being built bottom-up. Google's Map-Reduce [9] system introduced scalable, fault-tolerant implementations of two key dataflow primitives: independent processing of (groups of) records, and agglomeration of records that contain matching values. Then came Dryad [19], with built-in support for general dataflow graphs, including operator chains of arbitrary length and in- and out-bound branching. Now higher-level languages are being layered on top of these systems, to translate abstract user-supplied dataflow or query expressions into underlying parallel dataflow graphs, e.g., DryadLINQ [21], Jaql [17], Pig Latin [22] and Sawzall [23].

These high-level languages engender a relatively high degree of transparency in program structure. For example, standard data manipulation operations such as *join* and *filter* are expressed via declarative primitives. Also, multistep processing is explicitly broken down into the constituent steps, e.g., join followed by filter followed by face recognition, rather than being buried inside low-level constructs such as Map functions.

These forms of transparency, in addition to making programs easier to write, understand and maintain, also open up opportunities for automatic optimization. This paper proposes a set of optimization strategies for the DISC paradigm. The aim is not to provide concrete or proven results, but rather to suggest some jumping-off points for work in this area. Our belief is that good optimization technology, combined with the economics of programmer resources becoming more expensive relative to computer resources, will trigger a mass migration from low-level programming (e.g., direct Map-Reduce or Dryad programs) to high-level programming (e.g., Jaql, Pig Latin, SQL), analogous to the migration from assembly to C-style languages to Java-style languages.

1.1 Background: Pig

This paper is motivated by our experience with *Pig*, an open-source [4] dataflow engine used extensively at Yahoo! to process large data sets. Pig compiles Pig Latin programs, which are abstract dataflow expressions, into one or more physical dataflow jobs, and then orchestrates the execution of these jobs. Pig currently uses the Hadoop [3] open-source Map-Reduce implementation as its physical dataflow engine.

The current Pig implementation incorporates two simple but critical optimizations: (1) Pig automatically forms efficient pipelines out of sequences of per-record processing steps. (2) Pig exploits the *distributive* and *algebraic* [15] properties of certain aggregation functions, such as COUNT, SUM, AVERAGE and some user-defined functions, and automatically performs partial aggregation early (known as *combining* in the Map-Reduce framework), to reduce data sizes prior to the expensive data partitioning operation.

Our users are demanding more aggressive optimization of their programs, and indeed there is room to do much more on the optimization front.

[1] Previously "Data-Intensive Supercomputing."

1.2 DISC vs. DBMS Optimization

In developing automatic optimization techniques for DISC, we can draw upon many ideas and techniques from the database community, which has studied set-oriented data processing for decades, including parallel processing on clusters [7, 11, 13]. While DISC systems bear a strong resemblance to parallel query processors in database management systems (DBMS), the context is somewhat different: The DBMS context emphasizes highly declarative languages, normalized data and strong consistency, whereas DISC is geared toward procedural code, flexible data models, and cost-effective scalability through weak consistency and commodity hardware.

Traditional DBMS optimization techniques [18] are *model-based*, i.e., they search for "optimal" execution strategies over models of the data, operators and execution environment. In the DISC context, accurate models may not be available a priori, because: (1) Data resides in plain files for ease of interoperability with other tools, and the user may not instruct the system how to parse the data until the last minute; (2) Many of the operators are governed by custom user-supplied code whose cost and data reduction/blowup properties are not known a priori; (3) DISC uses large pools of unreliable and perhaps heterogeneous machines, and formulating simple and accurate models of the execution environment is a challenge.

Starting around 2000, motivated in part by considerations related to the ones stated above, the database community has begun studying *adaptive* approaches to one optimization problem: query plan selection [6, 10]. Adaptive query planning does not rely on the a-priori existence of accurate models, and instead adopts a trial-and-error, feedback-driven approach.

1.3 Model-Light Approach

DISC provides an interesting opportunity to revisit many of the specifics of query optimization in a new light. In particular, it makes sense to pursue a *model-light* approach, guided by the following principles:

1. **Discriminative use of information.** Optimization decisions should not be made on the basis of unreliable information. Conversely, information known to be reliable should be leveraged. For example, reliable file size metadata can typically be obtained, and knowing file sizes can be useful in selecting join algorithms (Section 3.2) and scheduling overlapping programs (Section 4.1). As another example, although the system may not have reliable cost and selectivity estimates for all operations, certain ones such as projection, simple filtering and counting are known to be cheap and data-reducing, and hence ought to be placed early in the execution sequence when possible (Section 3.1).
2. **Risk avoidance.** In cases where key optimization parameters are missing or unreliable, the optimization process should be geared toward minimizing the risk of a bad outcome. For example, when selecting derived data to cache, in the absence of reliable models for the size and utility of various derived data sets, the system should construct a diverse portfolio of cached content (Section 4.2.2). This strategy is less risky than betting on one particular category of derived data being the most useful, according to an unreliable model.
3. **Adaptivity.** Key parameters like intermediate data sizes and black-box function costs, which are hard to estimate a priori, can be measured at runtime. Based on measurements taken at runtime, the system may be able to adjust its execution and storage tactics on the fly, to converge to a better strategy over time. Aspects that are, at least in principle, amenable to adaptive optimization at runtime include dataflow graph structure (see [6, 10, 16]), load balancing across partitions (see [5, 26]), data placement (Section 4.2.1), and caching and reuse of derived data (Section 4.2.2).

In this paper we lay out some possible optimization strategies for DISC that align with the above principles. Section 3 focuses on *single-program optimizations* that optimize one program at a time, and highlights ideas from the database community that may be applicable in the DISC context. Section 4 focuses on *cross-program optimizations* that amortize IO and CPU work across related programs, and proposes several novel approaches.

Before discussing these optimizations we briefly describe our Pig Latin language, as a concrete example of a dataflow language that can be optimized.

2 Pig Latin Language Overview

Pig Latin is our high-level dataflow language for expressing computations or transformations over data. We illustrate the salient features of Pig Latin through an example.

Example 1 *Suppose we have search query logs for May 2005 and June 2005, and we wish to find "add-on terms" that spike in June relative to May. An add-on term is a term that is used in conjunction with standard query phrases, e.g., due to media coverage of the 2012 Olympics selection process there may be a spike in queries like "New York olympics" and "London olympics," with "olympics" being the add-on term. Similarly, the term "scientology" may suddenly co-occur with "Tom Cruise," "depression treatment," and other phrases. The following Pig Latin program describes a dataflow for identifying June add-on terms (the details of the syntax are not important for this paper).*[2]

[2] In reality, additional steps would be needed to eliminate pre-existing add-ons like "City" in "New York City"; due to space constraints we leave the additional steps as an exercise for the reader.

```
# load and clean May search logs
1.  M = load '/logs/may05' as (user, query, time);
2.  M = filter M by not isURL(query);
3.  M = filter M by not isBot(user);

# determine frequent queries in May
4.  M_groups = group M by query;
5.  M_frequent = filter M_groups by COUNT(M) > 10^4;

# load and clean June search logs
6.  J = load '/logs/june05' as (user, query, time);
7.  J = filter J by not isURL(query);
8.  J = filter J by not isBot(user);

# determine June add-ons to May frequent queries
9.  J_sub = foreach J generate query,
            flatten(Subphrases(query)) as subphr;
10. eureka = join J_sub by subphr,
                 M_frequent by query;
11. addons = foreach eureka generate
    Residual(J_sub::query, J_sub::subphr) as residual;

# count add-on occurrences, and filter by count
12. addon_groups = group addons by residual;
13. counts = foreach addon_groups generate residual,
             COUNT(addons) as count;
14. frequent_addons = filter counts by count > 10^5;
15. store frequent_addons into 'myoutput.txt';
```

Line 1 specifies the filename and schema of the May query log. Lines 2 and 3 filter out search queries that consist of URLs or are made by suspected "bots" (the filters are governed by the custom Boolean functions `isURL` *and* `isBot`*, which have been manually ordered to optimize performance). Lines 4–5 identify frequent queries in May.*

Lines 6–8 are identical to Lines 1–3, but for the June log. Lines 9–10 match sub-phrases in the June log (enumerated via a custom set-valued function `Subphrases`*) against frequent May queries. Line 11 then extracts the add-on portion of the query using a custom function* `Residual` *(e.g., "olympics" is an add-on to the frequent May query "New York").*

Lines 12–14 count the number of occurrences of each add-on term, and filter out add-ons that did not occur frequently. Line 15 specifies that the output should be written to a file called `myoutput.txt`.

In general, Pig Latin programs express acyclic dataflows in a step-by-step fashion using variable assignments (the variables on the left-hand side denote sets of records). Each step performs one of: (1) data input or output, e.g., Lines 1, 6, 15; (2) relational-algebra-style transformations, e.g., Lines 10, 14; (3) custom processing, e.g., Lines 9, 11, governed by *user-defined functions* (UDFs). A complete description of the language is omitted here due to space constraints; see [22].

3 Single-Program Optimizations

Optimization of a single dataflow program can occur in two stages:

- *Logical* optimizations restructure the logical dataflow graph supplied by the user. These optimizations produce a new graph that is semantically equivalent to the original one but implies a more efficient evaluation strategy. An example is to move a cheap filter ahead of more expensive operations with which the filter commutes.
- *Physical* optimizations pertain to how the logical dataflow graph is converted into a physical execution plan, e.g., as a sequence of Map-Reduce jobs. An example is to encode a sequence of two filters as a single Map operation.

3.1 Logical Optimizations

Examples of textbook [24] logical optimizations that are consistent with our model-light optimization philosophy (Section 1.3) include:

Early projection. Projection refers to the process of retaining only a subset of the fields of each record, and discarding the rest. A well-known optimization is to project out unneeded fields as early as possible.

Early filtering. Entire unneeded *records* can be eliminated early via filtering. In view of our discriminative information use and risk avoidance tenets, a filter should only be moved to an earlier position if there is enough confidence that it is cheap relative to its data reduction power. Thus, if the filter involves a UDF, it is better to leave the filter at the user-specified position since the UDF might be expensive and moving it earlier may cause it to be invocated more times than the user intended. As a possible extension, the cost and data-reducing power of UDFs can be discovered and exploited on the fly, using adaptive query processing techniques [10].

Operator rewrites. Sometimes, certain operator sequences can be recognized and converted into equivalent operators that have much more efficient implementations. For example, if a user writes a cross product of two data sets followed by a filter on the equality of two attributes, the cross and filter operators can be collapsed into a join operator that can typically be implemented more efficiently than a cross product.

3.2 Physical Optimizations

In the current Pig implementation, each logical dataflow graph is compiled into a sequence of Map-Reduce [9] jobs. A Map-Reduce job consists of five processing stages:

1. *Map*: process fragments of the input data set(s) independently, and assign *reduce keys* to outgoing data items.
2. *Combine* (optional): process all data items output by a given map instance that have the same reduce key, as a unit.
3. *Shuffle*: transmit all data items with a given reduce key to a single location.

4. *Sort*: sort all items received at a location, with the reduce key as a prefix of the sort specification.
5. *Reduce*: process all data items that have the same reduce key as a unit.

Logical operations need to be mapped into these stages. Sometimes an efficient mapping requires splitting a single logical operation into multiple physical ones. For example, a logical duplicate elimination operation that is to occur in the reduce stage can be converted into a sort operation (which can be incorporated into the Map-Reduce sort stage) followed by an operator that eliminates adjacent duplicate data items in a streaming fashion, in the reduce stage.

Join is perhaps the most critical operation, because it can be very expensive (quadratic cost in the worst case). There are several alternative ways to translate a logical join operation into a physical execution plan [14], including:

- **Symmetric join**: Pass both data sets through the same shuffle-sort sequence, using the join attribute(s) as the reduce key. Then match pairs of joining records in the reduce stage.
- **Symmetric join over prepartitioned data**: If the data is already partitioned on (a subset of) the join attribute(s), do the matching in the map stage, at which point the join is complete. (The shuffle, sort and reduce stages are not used.)
- **Asymmetric join**: Perform a map stage over fragments of the larger data set, and in each map instance read a full copy of the smaller data set to perform matching. (This execution method avoids having to shuffle the larger data set.)

The optimal join execution strategy in a given situation depends on whether the data is prepartitioned, and on the sizes of the input data sets (asymmetric join can be best if one data set is very small relative to the other). The cost differences among strategies can span orders of magnitude, so it is important to choose wisely.

If the join occurs early in the dataflow and processes data directly out of stored files, then the choice of join method can be driven by basic system metadata such as file sizes and file partitioning method (if any). If the join occurs late in the dataflow, following operators whose data reduction/blowup behavior has not been modeled well, then there is less hope of selecting a good join strategy in advance. In the latter case, an adaptive "wait and see" approach may make sense, even though doing so may incur additional materialization overhead and/or wasted work due to aborted trials.

4 Cross-Program Optimizations

We now consider combined optimization of multiple dataflow programs, perhaps submitted independently by different users. This type of optimization is of interest if the number of (popular) data sets is much less than the number of users processing those data sets. At internet companies like Yahoo!, hundreds of users pour over a handful of high-value data sets, such as the web crawl and the search query log. In data-intensive environments, disk and network IO represent substantial if not dominant execution cost components, and it is desirable to amortize these costs across multiple programs that access the same data set.

In some cases, programs that access the same data set also perform redundant computation steps. Dataflow programs tend to propagate among users via a cut-paste-modify model whereby users pass around code fragments over email lists or forums. In some cases users explicitly link their dataflow graphs to subgraphs published by other users, using tools like Yahoo! Pipes [27]. Hence it is often the case that programs submitted by different users exhibit a common prefix. For example, a processing prefix that occurs frequently at Yahoo! is: (1) access the web crawl, (2) remove spam pages, (3) remove foreign-language pages, (4) group pages by host for subsequent host-wise processing.

Mechanisms to amortize work across related programs fall into two categories: *concurrent* and *nonconcurrent* work sharing. Concurrent work sharing entails executing related programs in a joint fashion so as to perform common work only once. Nonconcurrent sharing entails caching the result of IO or CPU work performed while evaluating one program, and leveraging it for future programs. We discuss each category in turn.

4.1 Concurrent Work Sharing

If a set of programs sharing a common prefix or subexpression are encountered in the system's work queue at the same time, an optimizing compiler can create a single branching dataflow for the simultaneous evaluation of all the programs [25]. DISC workloads are dominated by IO-bound programs that scan large data files from end to end. In this context the most important "shared prefix" is the scan of the input file(s). Techniques exist for sharing file scans among concurrent programs [12].

In the presence of opportunities and mechanisms to share work among concurrent programs, the key open question is how to *schedule* programs to maximize the sharing benefit. In particular:

1. Given that coupling a slow program with a fast one may increase the response time of the latter, under what conditions should the system couple them?

2. Under what conditions is it beneficial to defer execution of an enqueued program in anticipation of receiving future sharable programs?

Typically, DISC systems do not attempt to guarantee a fast response time for any *individual* program. Rather, the aim is to make efficient use of resources and achieve low *average* response time. Hence we address the two questions above in the context of minimizing average response time. We begin with Question 1.

Suppose a pair of sharable programs P_1 and P_2 have individual execution times t_1 and t_2, respectively, but incur a lesser total time $t_{1+2} < t_1 + t_2$ if merged and evaluated jointly. Let t^s denote the sharable component of these programs, such that $t_{1+2} = t^s + t_1^n + t_2^n$ where t_i^n represents the remaining, nonsharable component of program P_i (i.e., $t_1 = t^s + t_1^n$ and $t_2 = t^s + t_2^n$).

If P_1 is shorter than P_2 ($t_1 < t_2$), then the serial schedule that minimizes average response time executes P_1 first, followed by P_2. Under this schedule the average response time is $(t_1 + (t_1 + t_2))/2 = (3t^s + 2t_1^n + t_2^n)/2$.

If we choose to execute P_1 and P_2 jointly, then the response time is $t_{1+2} = t^s + t_1^n + t_2^n$, which is less than the average response time given by the sequential schedule iff $t^s > t_2^n$. In other words, if more than half of P_2's execution time is sharable with P_1, then it benefits average response time to merge P_1 and P_2 into a single jointly-executed dataflow program. In the DISC context, where file scan IO is often the dominant cost, it makes sense to merge sets of programs that read the same file.

Question 2 is more difficult to answer, because it involves reasoning about future program arrivals, e.g., using a stochastic arrival model based on the popularity of each data set. We have studied this question extensively [1]. Our main result is the following.

Suppose programs that read data file F_i arrive according to a Poisson process with rate parameter λ_i. Suppose also that the dominant cost of these programs is the IO to scan F_i, which is a large file, and hence $t_i^s \propto |F_i|$.

The quantity $|F_i| \cdot \lambda_i$ represents the *sharability* of programs that access F_i: If F_i is large, then a substantial amount of IO work can be amortized by sharing a scan of F_i among multiple programs. If programs that access F_i arrive frequently (large λ_i), then this IO amortization can be realized without imposing excessive delay on individual programs.

In our formal analysis of priority-based scheduling policies, the factor $|F| \cdot \lambda$ plays a prominent role in the priority formulae we derive mathematically, thereby confirming our intuition about sharability. Our analytical model is based on a stationary program arrival process (Poisson arrivals), but the resultant scheduling policies appear to be robust to bursty workloads. (The arrival rate parameter λ can be estimated *adaptively* based on recent arrival patterns, using standard techniques.) Our scheduling policies tend to outperform conventional ones like FIFO and shortest-job-first, in terms of average response time, due to the ability to anticipate future sharing opportunities.

Importantly, effective sharability-aware scheduling does not depend on the ability to model the full execution time of a given program, which can be error prone. Instead, it is only necessary to model the *sharable* time t^s, which for IO-bound programs over large files is directly proportional to the file size $|F|$. This quantity can be obtained from file system metadata.

4.2 Nonconcurrent Work Sharing

We now consider ways to amortize work across programs that occur far apart in time. The mechanisms we propose can be thought of as forms of caching. Fortunately, in the DISC context most data is write-once and therefore cache consistency is not a major concern.

4.2.1 Amortizing Communication

Network IO can be amortized by caching data fragments at particular nodes in the distributed system, to avoid repeatedly transferring the same data. Let us assume that a scheduler assigns computation tasks to nodes in a manner mindful of data locality and load balancing considerations, as in the Hadoop scheduler [3]. Given such a scheduler, then ideally the placement of data fragments onto nodes should be such that the following properties hold:

1. Popular fragments, used by many programs, have replicas on many nodes. This property enables balancing of load across nodes without incurring network overhead to transfer data from busy nodes to idle ones.
2. Fragments that tend to be accessed together (many programs access the data residing in a pair of fragments, e.g., to perform a join) have some replicas co-located on the same node. This property facilitates strong locality of computation and data.
3. Popular fragments that are seldom accessed together are *not* co-located, to avoid hotspots.

The data placement problem has been studied before, with various static placement schemes being proposed and evaluated, including a model-free "round-robin" approach [20]. The round-robin tactic is consistent with the risk avoidance aspect of our philosophy (Section 1.3). However, we believe it is possible to achieve the desired data placement properties outlined above, while still avoiding reliance on explicit workload models, by making use of *adaptivity* (also mentioned in Section 1.3). Our proposed adaptive data placement scheme is:

- Each time the scheduler places a computation task on a node that does not contain all data fragments

read by the task, thereby forcing a network transfer to retrieve the needed fragment(s), store a copy of the newly-transfered fragment(s) at that node.

- Evict fragment replicas deemed to be of little utility, according to some eviction policy, subject to a constraint on the minimum replication level of each fragment for fault tolerance.

The rationale is as follows.

If the scheduler was forced to place a computation task away from its input data fragment(s), then one of three situations has likely occurred: (a) the input fragment(s) are so popular that all nodes containing replicas are busy, (b) all nodes containing replicas are busy due to the popularity of other fragments that happen to be co-located with them, or (c) no single node presently contains a copy of every fragment required by the operation (e.g., a join).

In the first case, our mechanism will increase the number of replicas of the needed fragment(s), thereby making the no-available-copies situation less likely to occur in the future. In the second case, the number of replicas will temporarily increase, followed by eviction of unused replicas on the busy nodes, resulting in a net movement of data (rather than a copy). In the third case, our mechanism brings all the required fragments to one place, so if they are accessed together again in the future full locality will be possible. If the fragments are seldom accessed independently, the eviction policy will eventually remove some of the non-co-located replicas, again yielding a net movement of data rather than a copy.

The success of this scheme hinges on the degree of temporal coherency in the pattern of programs accessing data. We are presently investigating the extent of such coherency in Yahoo!'s workloads. We are also studying the choice of eviction policy and scheduling policy, especially in regards to susceptibility to thrashing.

4.2.2 Amortizing Processing

In cases where multiple programs perform the same initial operation(s), e.g., canonicalize URL strings, remove spam pages, group pages by host, it may be beneficial to cache the result of the common operations. (The cached result may be stored on disk.) In databases, a derived data set that has been cached is called a *materialized view* [2].

One approach is to create materialized views manually, and instruct users to use these materialized views as inputs to their programs when possible. This approach is problematic. For one, selecting materialized views by hand is impractical in large, distributed organizations where no one person has a complete grasp of how data is being used around the organization. More importantly, if users start referencing materialized views explicitly in their programs, it becomes impossible to remove ones that are no longer desirable to keep. A preferred approach is for the system to select and adjust the set of materialized views *automatically*, and when applicable automatically rewrite user programs to use these views.

Standard automated methods of selecting materialized views [2] rely heavily on models to predict the size of a view, and the cost saved by using the view in lieu of the original input data. Ones that are expected to yield high cost savings relative to their size are selected.

In lieu of reliance on models, we are pursuing an approach based on adaptation and risk minimization (as motivated in Section 1.3). Our approach leverages the fact that DISC architectures store large data sets as a collection of relatively small *fragments*, each managed separately by the underlying file system. The idea is to hold small fragments of many view candidates simultaneously, continuously monitor their effectiveness, and increase the number of fragments of the most useful ones while eliminating the least useful ones. Concretely:

1. Enumerate a set of *candidate* materialized views, based on identifying common subexpressions among submitted programs. For example, if removing spam pages from an underlying web page data set is a common operation, then the spam-free derived data set is considered a candidate view.
2. Materialize a few fragments of each candidate view, e.g., materialize a spam-free copy of one fragment of the underlying web pages data set.
3. Over time, as new programs are processed by the system, compare the execution time of branches of the program that are able to leverage view fragments, to the execution time for branches that do not benefit from view fragments. The overall execution time savings attributed to a particular materialized view in a period of time is called the view's *utility*.
4. Add more fragments of views whose utility is large relative to their size, and conversely remove fragments of views with a low utility-to-size ratio.
5. Continue to adjust the set of materialized view fragments over time, as the workload evolves.

It is likely that a few of the candidate views are highly beneficial, while others are of little use or take up too much space. By materializing some fragments of each, the system ensures that it benefits at least partially from the "good" views. This approach mirrors the one advocated in investment portfolio theory for risk minimization in the presence of uncertainty. Since we do expect some degree of stationarity in our context, adaptively increasing the percentage of good-performing views in the portfolio ought to cause the portfolio to converge to a (local) optimum. Of course, the optimum may be a moving target if the workload shifts over time,which motivates continual adjustment of the portfolio.

We are currently investigating the viability of this approach, primarily from an empirical standpoint. A theoretical treatment of this problem would be of significant interest. It would likely touch upon aspects of investment portfolio theory, online learning theory (e.g., multi-armed bandit models), statistical sampling theory, and submodular optimization theory (because the utility of materializing multiple views in the same derivation chain is subadditive).

5 Summary

In this paper we discussed optimization of data-centric programs in the DISC context, a topic that has strong similarities but also important differences from database query optimization. We began by contrasting the two contexts, and offering three principles for success in the DISC context: discriminative use of information, risk avoidance, and adaptivity. Then we highlighted some database techniques that are, or can trivially be made to be, compatible with those principles.

We then turned to *cross-program optimization*. We motivated the need to optimize ensembles of interrelated dataflow programs, and argued that existing database techniques for this problem are insufficient in the DISC context. We sketched some possible approaches that to our knowledge have not been explored in either context.

Acknowledgments

We are grateful to the Yahoo! Grid team, especially the Pig engineering leads Alan Gates and Olga Natkovich, for their immense contributions to the engineering aspects of Pig. We also wish to thank Sandeep Pandey and Jayavel Shanmugasundaram, for inspiring some of the ideas in Section 4.2.2. Lastly, we thank Steve Gribble and Joe Hellerstein for helpful feedback on the paper.

References

[1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files, Mar. 2008. In submission.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes for SQL databases. In *Proceedings of the International Conference on Very Large Data Bases*, 2000.

[3] Apache Software Foundation. Hadoop software. `http://lucene.apache.org/hadoop`.

[4] Apache Software Foundation. Pig software. `http://incubator.apache.org/pig`.

[5] R. H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. on Computing Systems*, 21(1):36–86, Feb. 2003.

[6] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.

[7] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):4–24, 1990.

[8] R. E. Bryant. Data-intensive supercomputing: The case for DISC. Technical report, Carnegie Mellon, 2007. `http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-07-128.pdf`.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, 2004.

[10] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[11] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. In *Proceedings of the International Conference on Very Large Data Bases*, 1986.

[12] P. M. Fernandez. Red brick warehouse: A read-mostly RDBMS for open SMP platforms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[13] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1990.

[14] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1), 1997.

[16] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2004.

[17] IBM Research. Jaql software. `http://www.jaql.org`.

[18] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dyad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2007.

[20] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6(1):53–72, 1997.

[21] Microsoft Research. DryadLINQ software. `http://research.microsoft.com/research/sv/DryadLINQ`.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[23] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal*, 13(4), 2005.

[24] R. Ramakrishnan and J. Gehrke. *Database Management Systems, 3rd edition*. McGraw-Hill, 2003.

[25] T. K. Sellis. Multiple query optimization. *ACM Trans. on Database Systems*, 13(1), 1988.

[26] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the International Conference on Data Engineering*, 2003.

[27] Yahoo! Inc. Pipes software. `http://pipes.yahoo.com`.