Bulletin of the Technical Committee on

Data Engineering

March 2013 Vol. 36 No. 1

IEEE Computer Society

Letters

Special Issue on Query Optimization for Big Data Systems

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

Technical Committee on Data Engineering

Kyu-Young Whang, our new Chair of the Technical Committee on Data Engineering has now appointed a new executive committee. Kyu-Young describes his appointments in his letter on page two. I have changed the inside front cover of the Bulletin to reflect Kyu-Young's appointments. These appointments reflect both continuity and new initiatives. I would urge you to read Kyu-Young's letter and become familiar with the new Executive Committee.

The Current Issue

There is no question but that we live in the age of "big data". I believe this is the result not only of new business models (online advertizing, etc.) but also of economics. The continuing decline in prices for storage and compute cycles means that ever larger amounts of both can be utilized for economic gain, as business pursue profits in a widening sphere of activity, both traditional and new. But it is not only of commercial interest. Science is now exploiting data at scales undreamed of in the past, for dealing with everything from searching for Higgs bosons to conducting vast environmental studies.

While distributed database systems have for a long time provided scalability, "big data" stretches well beyond what these systems have historically been able to handle. The map-reduce framework was the first to emerge to deal with the current vast flood of data. Not surprisingly, however, users always want more and better.

Map-reduce systems are now being augmented in fundamental ways. We are learning new ways (and some old ways) to describe what we want to do with our big data. Hence the re-emergence of SQL. With SQL and other high level ways of describing the way we wish to process big data, one needs a compelling story that performance will not suffer when using these idioms. Once again, query optimization enters the picture in a very important way.

The current issue of the bulletin addresses how the handling of big data has evolved. Sudarshan, as issue editor, has collected a good representative sample of work going on in this area. This includes industrial as well as university based approaches. The scale of data represents a technical challenge not faced in the past. But also a great commercial and scientific opportunity. So you can expect to see work continue in this area for many years to come.

I want to thank Sudarshan for the fine work he has done in assembling the issue. The issue should serve as a partial snapshot of the state of "big data" processing. But more, it could serve as a gateway to understanding how the field is evolving. I think you will enjoy the issue and greatly appreciate the efforts both of authors and of Sudarshan, the issue editor.

David Lomet
Microsoft Corporation

Letter from the TCDE Chair

Dear IEEE Technical Committee on Data Engineering (TCDE) members,

It is my pleasure to announce the organization of TCDE Executive Committee. I retained most of the previous members for their valuable experience, but assigning different roles, while adding new members to initiate new activities. As a result, we have the following names and assigned roles:

Chair: Kyu-Young Whang, KAIST(kywhang@mozart.kaist.ac.kr)

Advisor: Paul Larson, Microsoft

Executive Vice Chair: Masaru Kitsuregawa (ICDE Steering Committee Chair), Tokyo University

Vice Chair for Conferences/TCDE Representative to ICDE Steering Committee: Malu Castallanous, HP Labs

Editor-in-Chief of Data Engineering Bulletin: David Lomet, Microsoft

Secretary/Treasurer/Web Management: Thomas Risse, L3S Research Center (risse@L3S.de)

Awards Program Coordinator: Amr El Abbadi, University of California Santa Babara

Membership Promotion: Xiaofang Zhou, The University of Queensland (zxf@itee.uq.edu.au)

Member: (open assignment)

- Erich Neuhold, University of Vienna
- Alan Fekete, University of Sydney
- Wookey Lee, Inha University

ACM SIGMOD Liaison: Christian Jensen, Aarhus University

Chair of Self Managing Database Systems Working Group: Shivnath Babu, Duke University

Co-Chair of Cloud Data Management Working Group: Hakan Hacigmus, NEC Lab America

Paul Larson and Erich Neuhold are previous TCDE chairs and have served TCDE for many years revitalizing its activities. Their valuable experience will be greatly appreciated for promoting TCDE and for serving its members. David Lomet, a previous TCDE chair, has been Chief Editor of Data Engineering Bulletin since 1992. It is his leadership that has been at the heart of the success of IEEE Data Engineering Bulletin and will continue to be so. Masaru Kitsuregawa, Steering Committee Chair of ICDE, our flagship conference, has contributed a lot to enhancing the quality of ICDE as the chair and member of the steering committee for many years, and will continue to serve in this capacity. Malu Castallanous has been a very active member of TCDE Executive Committee and will oversee the issues on holding and sponsoring TCDE related conferences. Thomas Risse has been successively managing the TCDE and ICDE meetings and TCDE Web for many years, and will continue to serve the TCDE in that capacity with added responsibility of TCDE Treasurer.

Amr El Abbadi is a new member and will oversee and create new programs regarding TCDE awards. Recognition is a very important aspect of academic advancement, and we will expand the award program to encourage our members to make bigger achievements and contributions. Xiaofang Zhou is a new member who will be responsible for TCDE membership promotion. We will make every effort to enlarge the TCDE community, encourage active participation, and provide benefits to our members. Xiaofang has been very active and successful in academic activities and is committed to this new role. Alan Fekete has been an active member of TCDE

Executive Committee, and will remain to be so with possible new roles in coming years. Wookey Lee is a new member who is expected to help the TCDE operations with various roles to be needed in the future.

Christian Jensen, being ACM SIGMOD Vice Chair, has been serving the TCDE as SIGMOD Liaison, and will continue to serve in this capacity. Shivanath Babu is a new member representing Self Managing Database Systems Working Group that he is leading. Hakan Hacigmus is a new member representing Cloud Data Management Working Group that he is co-chairing. Working groups are an important part of TCDE activity, and we will encourage their creative activities and try to create more working groups to cover timely research areas that will be of interest to TCDE members.

We have a very strong team of competent Executive Committee Members who are committed to serve for the benefit of the TCDE community. I am very grateful to them for accepting my invitation to join the committee and their willingness to serve. We also invite all TCDE members to actively participate in TCDE activities, and we will be there to help you.

One of the first events that we are planning for the benefit of TCDE members is the new TCDE reception to be held in the evening of April 8th (Monday) at the coming ICDE in Brisbane. We invite all the TCDE members or members to be (those who are willing to sign up for the membership) to this reception. Xiaofang Zhou, the Exec member in charge of membership promotion, who is organizing this reception will give you more details on time and place etc.

Besides, there will be a TCDE sponsored student travel award granted at this ICDE. TCDE also sponsors two workshops at ICDE: Self Managing Database Systems (SMDB) and Data Management in the Cloud (DMC). We hope many of you will be able to attend these workshops.

Kyu-Young Whang
KAIST

Letter from the Special Issue Editor

SQL Strikes Back! (a.k.a. The SEQUEL to NO-SQL)

Extremely large amounts of data are generated routinely today, by a number of sources such as Web sites, genome sequence and microarray data, network monitoring data, sensor data, and many other kinds of sources; such data is commonly referred to as "Big Data". The map-reduce paradigm has proven particularly successful for processing such Big Data. The availability of open-source map-reduce implementations such as Hadoop has lead to widespread use of the map-reduce paradigm.

However, many data processing needs that can be expressed very concisely in SQL, need an inordinate amount of coding effort in a map-reduce system, since the programmer has to express the logic of the computation imperatively, and further has to make choices about the implementation. The above problems with processing of Big Data have been addressed using two complementary approaches, which are represented by different papers in this special issue.

The first approach continues to use the map-reduce programming paradigm, but seeks to automate the task of optimizing map-reduce programs. For example, the paper by Herodotou and Babu discusses how to automate the choice of parameters for the map-reduce system, which has been shown to significantly improve performance. The paper by Dittrich et al. shows how to exploit the redundancy already present in the HDFS (and other similar) storage system to create and store different indices and different physical layouts in the different copies, which can significantly speed up map-reduce jobs. The paper by Kwon et al. addresses the problem of execution skew in Hadoop, showing how to reduce skew by more sophisticated initial partitioning, and by run-time adaptation through dynamic repartitioning.

The second approach provides programmers a declarative means of specifying their requirements, allowing the implementation to choose the best way of executing the declarative specification. Three systems that follow this approach are represented in this special issue: Apache Pig, SCOPE from Microsoft, and Hyracks from UC Irvine. The use of such declarative specifications has already overtaken low-level map-reduce specifications for many applications; for example, it has been reported that a large fraction of the map-reduce applications at Facebook have been replaced by queries in Hive, which supports an SQL-like declarative language.

The paper by Gates et al. from Hortonworks describes several optimizations that have been implemented in the Pig system, as well as some that are being considered for implementation. The SCOPE system from Microsoft allows programmers to integrate declarative specifications in an SQL-like language with imperative code written using a generalized map-reduce-combine framework. The declarative specification enables cost-based optimization, but a particular problem with optimization for many Big Data applications is the lack of statistics as well as the lack of detailed cost models for user-defined operations. The paper by Bruno et al. describes how information from earlier runs can be used to better estimate costs of alternative plans, and thus speed up future runs of a job. Since many jobs are executed repeatedly, their approach is particularly useful for such recurring jobs. Finally, the paper by Borkar and Carey describes a common algebraic framework which can support multiple Big Data languages such as SQL, Pig, or HiveQL. The algebraic framework implemented in the Alegbricks platform, gives users a choice of language appropriate for their needs, while allowing a single underlying implementation layer. The algebraic nature of the Algebrics platform lends itself to cost-based optimization (an area of future work for the Algebricks project).

With support for declarative querying, and query optimization, combined with transaction support in a new generation of massively parallel data management systems such as Google's Megastore and Spanner, it is clear that database technologies are back in the Big Data world. The articles in this issue provide an excellent insight into recent developments in query optimization for Big Data systems, and I'm sure you will find them as exciting a read as I did.

S. Sudarshan
IIT Bombay

A What-if Engine for Cost-based MapReduce Optimization

Herodotos Herodotou
Microsoft Research

Shivnath Babu
Duke University

Abstract

The Starfish project at Duke University aims to provide MapReduce users and applications with good performance automatically, without any need on their part to understand and manipulate the numerous tuning knobs in a MapReduce system. This paper describes the What-if Engine, an indispensable component in Starfish, which serves a similar purpose as a costing engine used by the query optimizer in a Database system. We discuss the problem and challenges addressed by the What-if Engine. We also discuss the techniques used by the What-if Engine and the design decisions that led us to these techniques.

1 Introduction

Modern organizations are collecting data ("big data") at an unprecedented scale. MapReduce systems like Hadoop have emerged to help these organizations store large datasets and run analytics on the data to generate useful insights [15]. Getting good performance from a MapReduce system can be a nontrivial exercise due to the large number of tuning knobs present in the system. Practitioners of big data analytics like business analysts, computational scientists, and systems researchers usually lack the expertise to tune these knobs.

The Starfish project at Duke University aims to provide MapReduce users and applications with good performance automatically, without any need on their part to understand and manipulate the numerous tuning knobs. Starfish uses a "profile-predict-optimize" approach:

1. *Profile*: Starfish observes and records the actual run-time behavior of MapReduce workloads executing on the system. This part is handled by the *Profiler* in Starfish.
2. *Predict*: Starfish analyzes and learns from these observations, and reasons about hypothetical tuning choices. This part is handled by the *What-if Engine* in Starfish.
3. *Optimize*: *Optimizers* in Starfish choose appropriate settings for the various tuning knobs in order to improve workload performance along one or more dimensions (e.g., completion time, resource utilization, pay-as-you-go monetary costs on cloud platforms).

Optimizers have been developed or are being developed in Starfish for setting a number of tuning knobs including: (i) choice of query execution plans for SQL-like queries run over MapReduce, (ii) degree of task-level parallelism for MapReduce jobs, (iii) physical design choices such as partitioning, ordering, and compression,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

(iv) provisioning of resources on the cluster, (v) selection of the virtual machine type on cloud platforms like Amazon Web Services, (vi) choice of job and task scheduling policies, and others.

The effectiveness of the Optimizers in finding close-to-optimal settings of the various tuning knobs depends heavily on the accuracy of the What-if Engine in predicting performance for different settings of these knobs. Thus, the What-if Engine is a critical component in a big data ecosystem. Unfortunately, the importance of the What-if Engine is often underestimated and undervalued.

In this paper, we focus on the "predict" piece of the "profile-predict-optimize" approach and describe Starfish's What-if Engine. We will first describe the overall problem and challenges that the What-if Engine addresses, and then outline the approach taken to solve them. We also report an experimental evaluation of the What-if Engine.

2 Design of Starfish's What-if Engine

2.1 Problem and Design Decisions

Consider a MapReduce job $j = \langle p, d, r, c \rangle$ that runs program p on input data d and cluster resources r using configuration parameter settings c. Job j's performance can be represented as:

$$perf = F_1(p, d, r, c) \tag{1}$$

Here:

- *perf* is some performance metric (e.g., execution time) of interest for jobs that is captured by the *performance model* F_1.
- The MapReduce program p is written directly in terms of map and reduce functions by a user or generated from a higher-level query specified in languages like HiveQL and Pig Latin [15].
- The input dataset d is represented by information such as d's size, the block layout of files that comprise d in the distributed file-system used by MapReduce, and whether d is stored compressed.
- The cluster resources r are represented by information such as the number of nodes and node types in r, the number of map and reduce task execution slots per node, and the maximum memory available per task slot.
- The job configuration c includes parameters such as the number of map and reduce tasks, the partitioning function, settings for memory-related parameters, the use of compression, and the use of the combine function [15].

Tuning knobs to optimize the performance of the MapReduce job $j = \langle p, d, r, c \rangle$ are present in each one of p, d, r, and c. For example, if p represents a join operation, then a map-side join or a reduce-side join can be done [1]. The data layout can be tuned by using row-based or column-based layouts [8]. The cluster resources can be tuned by choosing appropriate virtual machine types [6]. The job configuration can be tuned by adjusting the number of reduce tasks [5].

If the function F_1 from Equation 1 can be estimated accurately by the *What-if Engine*, then Starfish's Optimizers can be very effective at finding optimal settings for the above tuning knobs. However, accurate modeling of F_1 has to overcome many challenges that arise in distributed processing such as task parallelism, scheduling, and interactions among the large number of tuning knobs involved.

Furthermore, MapReduce jobs usually run as part of a MapReduce workflow W, where W is a directed acyclic graph (DAG) G_W that represents a set of MapReduce jobs and their dataflow dependencies. Consider a MapReduce workflow W that runs the programs $\{p_i\}$ from the corresponding jobs j_i in G_W on input base datasets

	What-if Questions on MapReduce Job Execution
WIF_1	How will the execution time of job j change if I increase the number of reduce tasks from the current value of 20 to 40?
WIF_2	What is the new estimated execution time of job j if 5 more nodes are added to the cluster, bringing the total to 20 nodes?
WIF_3	How will the execution time of job j change when I execute j on the production cluster instead of the development cluster, and the input data size increases by 60%?
WIF_4	What is the estimated execution time of job j if I execute j on a new cluster with 10 EC2 nodes of type m1.large rather than the current in-house cluster?

Table 1: Example questions the What-if Engine can answer.

$\{b_i\}$ and cluster resources r using configuration parameter settings $\{c_i\}$. W's performance can be represented as:

$$perf = F_2(\{p_i\}, \{b_i\}, r, \{c_i\}) \tag{2}$$

Building the *performance model* F_2 to predict workflow performance brings in additional complications due to concurrent execution of independent jobs in W on the cluster resources r.

It is quite clear that performance prediction is a very hard problem. At the same time, it is a problem that needs a reasonable solution if we want to generate good settings for tuning knobs automatically. One of the contributions we have made in Starfish is to identify an abstraction of the prediction problem that strikes a good balance between being applicable and admitting to good solutions. This abstraction can be characterized by the following three design decisions:

1. Address *relative* prediction problems instead of *absolute* prediction problems.
2. Develop *profiles* that capture the salient aspects of MapReduce execution. Use profiles consistently to represent the observed performance of actual MapReduce jobs that run as well as the predicted performance of hypothetical MapReduce jobs.
3. Break the overall prediction problem into composable components and attack each component with the best modeling technique available among analytical models, black-box models, and simulation models.

2.2 Relative What-if Questions

Starfish's What-if Engine can answer any *what-if question* of the following general form:[1]

> Given the profile of a job $j = \langle p, d_1, r_1, c_1 \rangle$ that runs a MapReduce program p over input data d_1 and cluster resources r_1 using configuration c_1, what will the performance of program p be if p is run over input data d_2 and cluster resources r_2 using configuration c_2? That is, how will job $j' = \langle p, d_2, r_2, c_2 \rangle$ perform?

Note the relative format in which the what-if question is specified. Being relative here means that the What-if Engine is given complete information about the execution of the job j, and then asked about how a related, but different, job j' will perform. Intuitively, relative what-if questions are easier to answer than absolute questions. In the relative case, only an estimate of the change in performance is needed; whereas the absolute case as represented by Equation 1 needs a prediction starting from scratch with limited input information provided.

Table 1 lists several interesting what-if questions that users or applications can express directly to the Starfish What-if Engine. For example, consider question WIF_1 from Table 1. Here, the performance of a MapReduce

[1] For simplicity of presentation, this specification considers a single MapReduce job instead of a workflow.

Algorithm for predicting MapReduce workflow performance

Input: Profile of jobs in workflow, Cluster resources, Base dataset properties, Configuration settings
Output: Prediction for the MapReduce workflow performance

For each (job profile in workflow in topological sort order) {
    Estimate the virtual job profile for the hypothetical job (Sections 3.1, 3.2, and 3.3);
    Simulate the job execution on the cluster resources (Section 3.4);
    Estimate the data properties of the hypothetical derived dataset(s) and the overall job performance;
}

Figure 1: Overall process used by the What-if Engine to predict the performance of a MapReduce workflow.

job $j = \langle p, d, r, c \rangle$ is known when 20 reduce tasks are used to run j. The number of reduce tasks is one of the job configuration parameters in c. WIF_1 asks for an estimate of the execution time of a hypothetical job $j' = \langle p, d, r, c' \rangle$ whose configuration c' is the same as c except that c' specifies using 40 reduce tasks. The MapReduce program p, input data d, and cluster resources r remain unchanged in j'.

As indicated in Table 1, the What-if Engine can answer questions on real and hypothetical input data as well as cluster resources. Furthermore, as we will describe in Section 3, the What-if Engine can answer what-if questions for MapReduce workflows composed of multiple jobs that exhibit producer-consumer relationships or are run concurrently.

2.3 Profiles

Note that a relative what-if question needs the profile of a MapReduce job as input. A profile consists of fields that together form a concise summary of the job's execution. These fields are partitioned into four categories:[2]

- *Cost fields:* These fields capture information about execution time at the level of tasks as well as phases within the tasks for a MapReduce job execution. Some example fields are the execution time of the Collect and Spill phases of a map task.
- *Cost statistics fields:* These fields capture statistical information about execution time for a MapReduce job—e.g., the average time to read a record from the distributed file-system, or the average time to execute the map function per input record—that is expected to remain unchanged across different executions of the job unless the CPU and/or I/O resources available per node change.
- *Dataflow fields:* These fields capture information about the amount of data, both in terms of bytes as well as records (key-value pairs), flowing through the different tasks and phases of a MapReduce job execution. Some example fields are the number of map output records and the amount of bytes shuffled among the map and reduce tasks.
- *Dataflow statistics fields:* These fields capture statistical information about the dataflow—e.g., the average number of records output by map tasks per input record (the Map selectivity) or the compression ratio of the map output—that is expected to remain unchanged across different executions of the MapReduce job unless the data distribution in the input dataset changes significantly across these executions.

3 MapReduce Performance Prediction

Figure 1 shows the What-if Engine's overall process for predicting the performance of a MapReduce workflow. The DAG of job profiles in the workflow is traversed in topological sort order to ensure that the What-if Engine

[2]Due to space constraints, we refer the reader to [4] for the full listing and description of fields in all four categories.

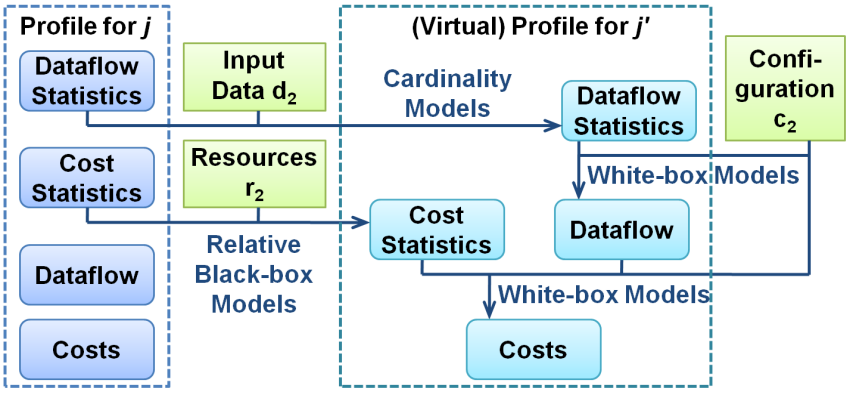


Figure 2: Overall process used by the What-if Engine to estimate a virtual job profile.

respects the dataflow dependencies among the jobs. For each job profile, a *virtual job profile* is estimated for the new hypothetical job j' based on the new configuration settings, the cluster resources, and the properties of the data processed by j' (Sections 3.1, 3.2, and 3.3). The virtual profile is then used to simulate the execution of j' on the (perhaps hypothetical) cluster (Section 3.4). Finally, the simulated execution is used to estimate the data properties for the derived dataset(s) produced by j' as well as the overall performance of j'. The overall workflow performance is predicted by combining the performance predictions for each job in the workflow.

The process of virtual profile estimation forms the foundation on which Starfish's ability to answer what-if questions is based. Specifically, given the profile of a job $j = \langle p, d_1, r_1, c_1 \rangle$ and the properties of input data d_2, cluster resources r_2, and configuration parameter settings c_2 of a hypothetical job $j' = \langle p, d_2, r_2, c_2 \rangle$, the virtual profile of j' has to be estimated. Our solution for virtual profile estimation is based on a mix of black-box and white-box models. The overall estimation process has been broken down into smaller steps as shown in Figure 2, and a suitable modeling technique was picked for each step. These smaller steps correspond to the four categories of fields in a job profile. We use conventional cardinality (white-box) models from database query optimization to estimate dataflow statistics fields (described in Section 3.1), relative black-box models to estimate the cost statistics fields (described in Section 3.2), and new analytical (white-box) models that we developed to estimate the dataflow fields, and in turn, the cost fields (described in Section 3.3).

3.1 Cardinality Models to Estimate Dataflow Statistics Fields

Database query optimizers keep fine-grained, data-level statistics such as histograms to estimate the dataflow in execution plans for declarative queries. However, MapReduce frameworks lack the declarative query semantics and structured data representations of Database systems. Thus, the common case in the What-if Engine is to not have detailed statistical information about the input data d_2 in the hypothetical job j'. By default, the What-if Engine makes a *dataflow proportionality assumption* which says that the logical dataflow sizes through the job's phases are proportional to the input data size. It follows from this assumption that the dataflow statistics fields in the virtual profile of j' will be the same as those in the profile of job j given as input. When additional information is available, the What-if Engine allows the default assumption to be overridden by providing dataflow statistics fields of the virtual profile directly as input.

3.2 Relative Black-box Models to Estimate Cost Statistics Fields

Clusters with identical resources will have the same CPU and I/O (local and remote) costs. Thus, if the cluster resource properties of r_1 are the same as those of r_2 in the what-if question, then the cost statistics fields in the virtual profile of the hypothetical job j' can be copied directly from the profile of job j given as input. This copying cannot be used when $r_1 \neq r_2$, in particular, when job j' will run on a *target cluster* containing nodes with a different type from the *source cluster* where job j was run to collect the profile that was given as input.

The technique we use when $r_1 \neq r_2$ is based on a *relative* black-box model $M_{src \to tgt}$ that can predict the cost statistics fields CS_{tgt} in the virtual profile for the target cluster from the cost statistics fields CS_{src} in the job profile for the source cluster.

$$CS_{tgt} = M_{src \to tgt}(CS_{src}) \tag{3}$$

Generating training samples for the $M_{src \to tgt}$ model: Let r_{src} and r_{tgt} denote the cluster resources respectively for the source and target clusters. Suppose some MapReduce program p is run on input data d and configuration parameter settings c on both the source and target clusters. That is, we run the two jobs $j_{src} = \langle p, d, r_{src}, c \rangle$ and $j_{tgt} = \langle p, d, r_{tgt}, c \rangle$. From these runs, we can generate the profiles for each individual task in these two jobs by direct measurement. Therefore, from the i^{th} task in these two jobs,[3] we get a training sample $\langle \mathrm{CS}_{src}^{(i)}, \mathrm{CS}_{tgt}^{(i)}, \rangle$ for the $M_{src \to tgt}$ model.

The above training samples were generated by running a related pair of jobs j_{src} and j_{tgt} that have the same MapReduce program p, input data d, and configuration parameter settings c. We can generate a complete set of training samples by using a *training benchmark* containing jobs with different $\langle p, d, c \rangle$ combinations. Selecting an appropriate training benchmark is nontrivial because the two main requirements of effective black-box modeling have to be satisfied. First, for accurate prediction, the training samples must have good coverage of the prediction space. Second, for efficiency, the time to generate the complete set of training samples must be small. Both actual and synthetic jobs can be used to generate the training samples. Different methods to generate the training benchmark are described in [4].

Learning all the $M_{src \to tgt}$ models needed: It is important to note that the training benchmark has to be run only once (or with a few repetitions) per target cluster resource; giving only a linear number of benchmark runs, and not quadratic as one might expect from the relative nature of the $M_{src \to tgt}$ models. The training samples for each source-to-target cluster pair are available from these runs. For example, to address question WIF_3 from Table 1 (i.e., planning for workload transition from a development cluster to production), one run each of the training benchmark on the development and the production cluster will suffice.

Once the training samples are generated, there are many *supervised learning* techniques available for generating the black-box model in Equation 3. Since cost statistics are real-valued, we selected the *M5 Tree Model* [12]. An M5 Tree Model first builds a regression-tree using a typical decision-tree induction algorithm. Then, the tree goes through pruning and smoothing phases to generate a linear regression model for each leaf of the tree.

In summary, given the cost statistics fields CS_{src} in the job profile for the source cluster r_{src}, the relative black-box model $M_{src \to tgt}$ is used to predict the cost statistics fields CS_{tgt} in the virtual profile for the target cluster r_{tgt} when $r_{src} \neq r_{tgt}$.

3.3 Analytical Models to Estimate Dataflow and Cost Fields

The What-if Engine uses a detailed set of analytical (white-box) models to calculate the dataflow fields in the virtual profile given (i) the dataflow statistics fields estimated above, and (ii) the configuration parameter settings c_2 in the hypothetical job j'. These models give good accuracy by capturing the subtleties of MapReduce job execution at the fine granularity of phases within map and reduce tasks. The current models were developed for Hadoop, but the overall approach applies to any MapReduce implementation. A second set of analytical models combines the estimated cost statistics and dataflow fields in order to estimate the cost fields in the virtual profile. The full set of models is described in [4].

[3] Ensuring that the tasks have the same input data d and configuration parameter settings c ensures that there is a one-to-one correspondence between the tasks in these two jobs.

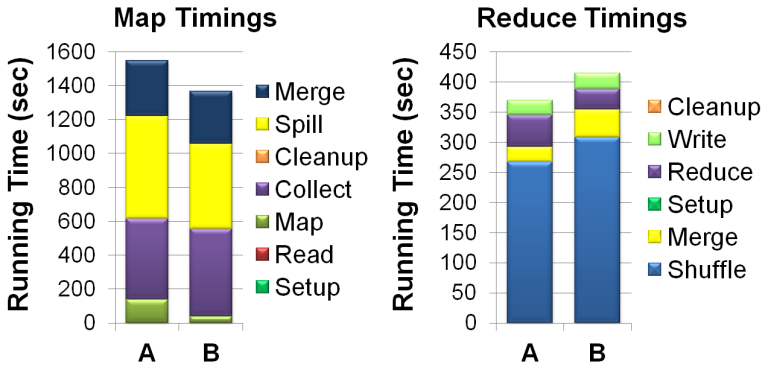


Figure 3: Map and reduce time breakdown for Word Co-occurrence jobs from (A) an actual run and (B) as predicted by the What-if Engine.

3.4 Simulating the Execution of a MapReduce Workload

The virtual job profile contains detailed dataflow and cost information estimated at the task and phase level for the hypothetical job j'. The What-if Engine uses a *Task Scheduler Simulator*, along with the job profiles and information on the cluster resources, to simulate the scheduling and execution of map and reduce tasks in j' (recall Figure 1). The Task Scheduler Simulator is a pluggable component that takes into account the dataflow and resource dependencies among the MapReduce jobs in a workflow. Our current implementation is a lightweight discrete event simulation of Hadoop's default FIFO scheduler.

The final output—after all jobs have been simulated—is a description of the complete (hypothetical) workflow execution in the cluster. The desired answer to the what-if question—e.g., predicted workflow running time, amount of local I/O, or a visualization of the task execution timeline—can be computed from the workflow's simulated execution.

4 Accuracy of What-if Analysis

We report some empirical results on the accuracy of the What-if Engine in predicting the overall execution time of different MapReduce workflows. In our evaluation, we used Hadoop clusters running on Amazon EC2 nodes of various sizes and node types. Figure 3 compares the actual MapReduce task and phase timings with the corresponding predictions from the What-if Engine for a Word-Cooccurrence MapReduce program run over 10GB of real data obtained from Wikipedia. Note that even though the predicted timings are slightly different from the actual ones, the relative percentage of time spent in each MapReduce phase is captured fairly accurately.

We also ran multiple MapReduce workflows under 40 different configuration settings. We then asked the What-if Engine to predict the job execution time for each setting. Figures 4(a)-(c) show scatter plots of the actual and predicted times for these 40 jobs. Observe the proportional correspondence between the actual and predicted times, and the clear identification of settings with the top-k best and worst performance (indicated by the green and red dotted circles respectively).

The fairly uniform gap between the actual and predicted timings in Figure 4 is due to inaccuracies in the profiles given as input to the What-if Engine. These inaccuracies are caused by the overhead of the Java profiling technology that Starfish uses currently. The gap is largest when the MapReduce program runs under CPU contention—which was the case when the Word Co-occurrence job was profiled. Unlike the case of Word Co-occurrence in Figure 4(a), the predicted values in Figures 4(b) and 4(c) are closer to the actual values; indicating that the profiling overhead is reflected less in the costs captured in the job profile. We expect to close this gap using commercial Java profiling technology.

Another common use case we considered in our evaluation is the presence of a development cluster, and the need to stage jobs from the development cluster to the production cluster. In our evaluation, we used a

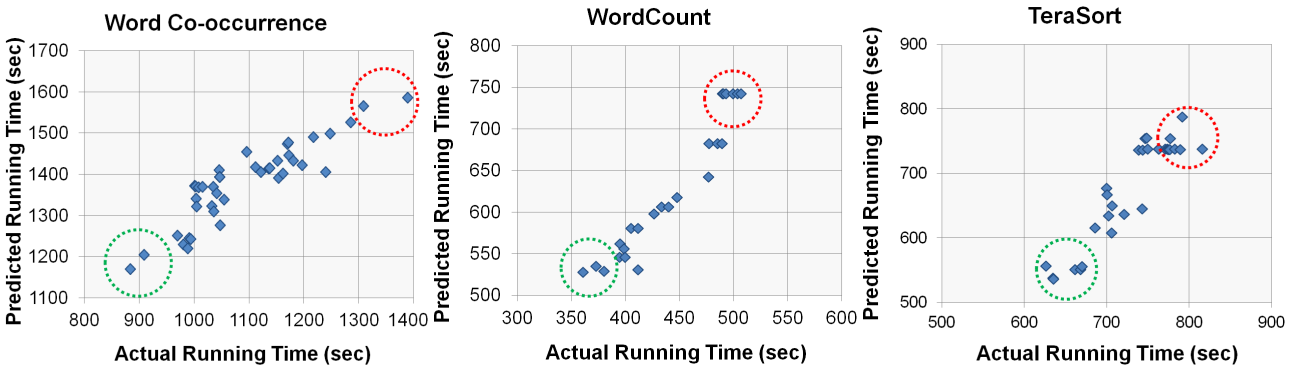


Figure 4: Actual Vs. predicted running times for (a) Word Co-occurrence, (b) WordCount, and (c) TeraSort jobs running with different configuration parameter settings.

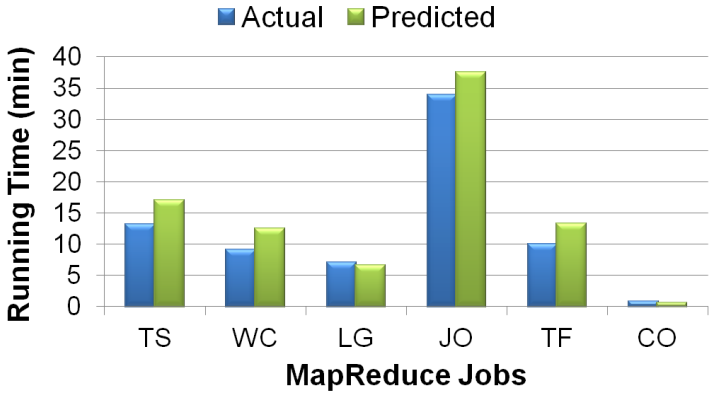


Figure 5: Actual and predicted running times for MapReduce jobs when run on the production cluster. The predictions used job profiles obtained from the development cluster.

10-node Hadoop cluster with m1.large EC2 nodes (4 EC2 units CPU[4], 7.5 GB memory, 850 GB storage) as the development cluster, and a 30-node Hadoop cluster with m1.xlarge EC2 nodes (8 EC2 units CPU, 15 GB memory, 1690 GB storage) as the production one. We profiled a number of MapReduce programs on the development cluster. We then executed the MapReduce programs on the production cluster using three times as much data as used in the development cluster.

Figure 5 shows the actual and predicted running times for each job when run on the production cluster. The What-if Engine used job profiles obtained from the development cluster for making the predictions. Apart from the overall running time, the What-if Engine can also predict several other aspects of the job execution like the amount of I/O and network traffic, the running time and scheduling of individual tasks, as well as data and computational skew.

Overall, the What-if Engine is capable of capturing the performance trends when varying the configuration parameter settings or the cluster resources.

5 Related Work

MapReduce is emerging rapidly as a viable competitor to existing systems for big data analytics, even though it currently trails existing systems in peak query performance [11]. When users are asked to tune the performance of their MapReduce workloads, they have to rely on their experience, knowledge of the data being processed, and rules of thumb from human experts or tuning manuals to complete the task [15]. Automated approaches for such

[4]One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz processor.

performance tuning are based typically on white-box models, black-box models, or simulation. Our approach is unique as it using a carefully-designed combination of all three aforementioned techniques to achieve the task.

White-box models are created by human experts who have detailed knowledge of how workload execution is affected by the properties of cluster resources, input data, system configuration, and scheduling policies. If the models are developed correctly, then the predictions from them will be accurate; as is the case in the *epiC* system, which supports System-R-style join ordering in Hive using white-box modeling [16]. However, white-box models can quickly become inaccurate if they have to capture the impact of evolving features such as hardware properties [6].

There has been considerable interest recently in using black-box models like regression trees to build workload performance predictors [2]. These models can be trained automatically from samples of system behavior, and retrained when major changes happen. However, these models are only as good as the predictive behavior of the independent variables they use and how well the training samples cover the prediction space. Relative (fitness) modeling is a black-box approach proposed recently for complex storage devices [9]. These models drastically simplify the model-learning process as long as actual workload performance can be measured on the devices. The What-if Engine borrows ideas from relative modeling and applies them to MapReduce clusters.

Simulation is often a valid alternative to modeling, but developing accurate simulators faces many of the same challenges as analytical white-box models. Some discrete event simulators have been proposed for Hadoop (e.g., *Mumak* [13] and *MRPerf* [14]), but none of them meet the needs of the What-if Engine. Mumak needs a job execution trace as input, and cannot simulate job execution with a different cluster size, network topology, or even different numbers of map or reduce tasks from what the execution trace contains. MRPerf is able to simulate job execution at the task level like our What-if Engine. However, MRPerf uses an external network simulator to simulate the data transfers and communication among the cluster nodes; which leads to a per-job simulation time on the order of minutes. Such a high simulation overhead prohibits MRPerf's use by a cost-based optimizer that needs 100-1000s of what-if calls done over different configuration parameter settings.

A MapReduce program has semantics similar to a *Select-Project-Aggregate* (SPA) in SQL with User-defined functions (UDFs) for the selection and projection (map) as well as the aggregation (reduce). *HadoopToSQL* and *Manimal* perform static analysis of MapReduce programs in order to extract declarative constructs like filters and projections, which are then used for database-style optimizations [3, 7]. Manimal does not perform profiling, what-if analysis, or cost-based optimization; it uses rule-based optimization instead. *MRShare* performs multi-query optimization by running multiple SPA programs in a single MapReduce job [10]. MRShare proposes a (simplified) cost model for this application. Overall, previous work related to MapReduce job optimization targets semantic optimizations for MapReduce programs and is either missing or lacks comprehensive profiling and what-if analysis.

6 Summary

In this paper, we described the design and implementation of the Starfish What-if Engine. Such a What-if Engine is an indispensable component of any optimizer for MapReduce systems, just like a costing engine is for a query optimizer in a Database system. The uses of the What-if Engine go beyond optimization: it may be used by physical design tools for deciding data layouts and partitioning schemes; it may be used as part of a simulator that helps make critical design decisions during a Hadoop setup—like the size of the cluster, the network topology, and the node compute capacity; or it may help make administrative decisions—like how to allocate resources effectively or which scheduling algorithm to use.

References

[1] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proc. of the 2010 ACM SIGMOD Intl. Conf. on Management of Data*, pages 975–986. ACM, 2010.

[2] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *Proc. of the 1st USENIX Conf. on Hot Topics in Cloud Computing*. USENIX Association, 2009.

[3] M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. In *Proc. of the 13th Intl. Workshop on the Web and Databases*, pages 10:1–10:6. ACM, 2010.

[4] H. Herodotou. *Automatic Tuning of Data-Intensive Analytical Workloads*. PhD thesis, Duke University, May 2012.

[5] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *Proc. of the VLDB Endowment*, 4(11):1111–1122, 2011.

[6] H. Herodotou, F. Dong, and S. Babu. No One (cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *Proc. of the 2nd Symposium on Cloud Computing*. ACM, 2011.

[7] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: A MapReduce Query Optimizer. In *Proc. of the 5th European Conf. on Computer Systems*, pages 251–264. ACM, 2010.

[8] A. Jindal, J.-A. Quian-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *Proc. of the 2nd Symposium on Cloud Computing*. ACM, 2011.

[9] M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger. Modeling the Relative Fitness of Storage. *SIGMETRICS*, 35(1):37–48, 2007.

[10] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *Proc. of the VLDB Endowment*, 3(1):494–505, 2010.

[11] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proc. of the 2009 ACM SIGMOD Intl. Conf. on Management of Data*, pages 165–178. ACM, 2009.

[12] R. J. Quinlan. Learning with Continuous Classes. In *Proc. of the 5th Australian Joint Conference on Artificial Intelligence*, pages 343–348. Springer Berlin / Heidelberg, 1992.

[13] H. Tang. *Mumak: Map-Reduce Simulator*, 2009. `https://issues.apache.org/jira/browse/MAPREDUCE-728`.

[14] G. Wang, A. Butt, P. Pandey, and K. Gupta. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. In *Proc. of the IEEE Intl. Symp. on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–11. IEEE, 2009.

[15] T. White. *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[16] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query Optimization for Massively Parallel Data Processing. In *Proc. of the 2nd Symposium on Cloud Computing*. ACM, 2011.

Efficient OR Hadoop: Why not both?

Jens Dittrich Stefan Richter Stefan Schuh Jorge-Arnulfo Quiané-Ruiz

Abstract

In this article[1], we give an overview of research related to Big Data processing in Hadoop going on at the Information Systems Group at Saarland University. We discuss how to make Hadoop efficient. We briefly survey four of our projects in this context: Hadoop++, Trojan Layouts, HAIL, and LIAH. All projects aim to provide efficient physical layouts in Hadoop including vertically partitioned data layouts, clustered indexes, and adaptively created clustered indexes. Most of our techniques come (almost) for free: they create little to no overhead in comparison to standard Hadoop.

1 Introduction

In the past years, the Hadoop ecosystem has become the de facto standard to handle so-called Big Data [3]. Hadoop's main components are Hadoop Distributed File System (HDFS) and Hadoop MapReduce.

HDFS allows users to store petabytes of data on large clusters. HDFS provides high fault-tolerance capabilities in an environment where failures of single disks or whole nodes are not the exception but the rule. Hadoop MapReduce allows users to query the data with the simple yet expressive map()/reduce() paradigm without a need for the user to care about parallelization, scheduling, or failover. In contrast to parallel DBMS, Hadoop MapReduce scales easily to very large clusters of thousands of machines. In addition, the upfront investment for using MapReduce is small: no need to use schemas, no integrity constraints, no data cleaning, no normalization, and NoSQL [12]. Moreover, installing and configuring a MapReduce cluster is relatively easy compared to a parallel DBMS: Almost any user with minimal knowledge of Java is able to write and run Hadoop jobs. All of this explains the popularity of MapReduce among non-database people.

On the flip side, Hadoop MapReduce processes MapReduce jobs by default in a non-optimized, scan-oriented fashion — in fact the entire system design is centered around the idea of executing long running jobs. Furthermore, several classes of tasks cannot be expressed naturally with the map()/reduce() paradigm, e.g., joins, iterative tasks, and updates. And finally, the performance of MapReduce is in many cases far from the one of an optimized parallel DBMS.

One might conclude that there is a deep divide among the two classes of systems — parallel DBMS and Hadoop MapReduce. And in fact: in 2009, the database community triggered a heated discussion with a paper by Pavlo et.al. [14] which unfortunately widened that divide. However, given the recent popularity of Hadoop, one might get the idea that there must be a reason for this popularity. If database systems are so great, why isn't everyone using them?

We believe that the key to this discussion is not about the 'new kid on the block' Hadoop solely learning from mature database technology, but that it is key for databases to also learn from Hadoop. Our research question is:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] A short version of this article appeared in the German Database magazine „Datenbankspektrum" in January 2013.

is there a way to preserve the properties of Hadoop while fixing its performance issues AND without turning it into yet another parallel DBMS? As a consequence, in 2009, we started a series of projects investigating this. These projects are Hadoop++, Trojan Layouts, HAIL, and LIAH. We will briefly sketch these projects in the following.

2 Hadoop++

The very first question we had in mind in 2009 was *can we substantially improve the runtimes of MapReduce jobs without touching the Hadoop source code?*

We investigated this question in [4]. In that work, we analyzed the query processing pipeline of Hadoop. The major observation was that Hadoop implements a hard-coded processing pipeline whose structure is very hard to change. However, Hadoop's processing pipeline also provides at least ten different user-defined functions (UDFs) — `map()` and `reduce()` being just two of them. These different UDFs may be exploited to place arbitrary code inside the Hadoop processing pipeline and turn Hadoop into a versatile distributed runtime. We exploited this to *inject* indexing and co-partitioning algorithms into Hadoop. This idea is somehow similar to injecting a trojan, however: this time for good. Hence, we coined the resulting index structure a *trojan index*. For instance, we change the `group()` and `shuffle()` UDFs that control grouping and shuffling. This allows us to create separate indexes for each HDFS block. We evaluated our indexes using the benchmark proposed in [14]. We could show that the query runtimes of Hadoop++ are by up to a factor 20 faster than Hadoop[2]. These performance improvements are possible, as we spend additional time creating indexes and copartitions before executing any MapReduce job, i.e. we create appropriate indexes matching the expected workload and we partition both tables on their join keys to be able to execute joins locally. The time spent for creating those indexes may be considerable [4].

The idea of improving the performance of a closed-source system by injecting code afterwards may also be applied to traditional database systems. In [11] we investigated how to change the data layout of a closed-source row-store into using compressed column-oriented layouts yielding up to a factor of 20 performance improvements.

3 Trojan Layouts

When working in the Hadoop++ project, we could observed that Hadoop MapReduce wastes a significant amount of time reading unnecessary data from disk as it stores data in row layout. Obviously, one could simply store all data in a column layout and hope for similar speed-ups as known from traditional column stores. However, in a distributed system there is a major issue with this approach: column data representing the same rows should be stored physically close as to avoid expensive network I/O for tuple reconstruction. Therefore, the question we had at that point in time was: *How could we change the data layout in Hadoop to be better suitable for analytical query processing?*

We investigated this question in [9]. As a first step, we studied the three most popular data layouts in the literature (namely Row Layout, Column Layout, and PAX Layout) and compared then with the optimal layout[3]. Figure 1 shows results of a simulation of the access costs for different layouts in Hadoop. The horizontal axis depicts the number of referenced attributes for a query. The vertical axis depicts the estimated data access costs. For a Column Layout the costs for network transfer have to be factored in and ruin the overall performance. For Row Layout, the number of referenced attributes does not have an effect. Therefore, a popular layout in the context of MapReduce is the hybrid layout PAX [1]: in this approach, all data inside an HDFS block, i.e.,

[2] A companion paper explores the pitfalls when measuring distributed systems like MapReduce in a cloud environment [18]. Other works look at the efficiency of the Hadoop failover algorithms [16, 15].

[3] Where the optimal layout denotes the data layout that has all attributes required by an incoming query in row format.

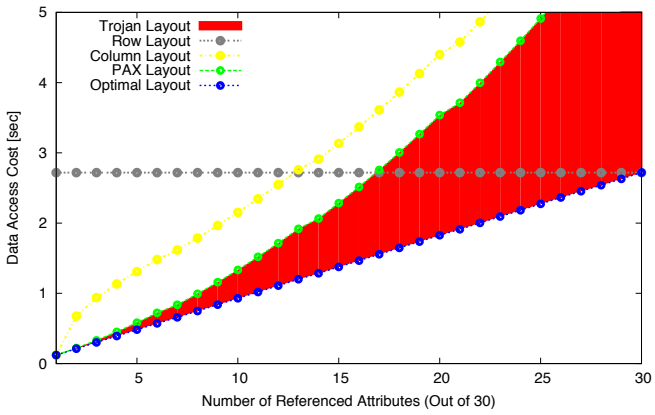


Figure 1: Simulation of data access costs for different data layouts in Hadoop [9].

a horizontal partition of data of 64MB by default, is stored in column layout. This avoids the problems with network I/O for tuple reconstruction and still gives column-like access. However, for some workloads, PAX is not the best layout. The interesting space in Figure 1 is the red area. It depicts data layouts that are enabled by Trojan Layouts and perform better than PAX. Some of those layouts are at the same time better than Row Layout. The latter layouts are the ones we wanted to identify.

In [9] we follow the PAX philosophy in that we keep data belonging to a particular HDFS block on that HDFS block i.e. there is no global reorganization of data *across* HDFS blocks. However, in contrast to PAX, we introduce an important change: Hadoop's Distibuted File Systems (HDFS) stores three copies of an HDFS block for fault-tolerance anyway. All of these copies are byte-identical. We change this to allow the different copies of a *logical* HDFS block to have different *physical*s layouts. As we do not remove any data from the different copies, we fully preserve the fault-tolerance properties of HDFS. At the same time, we are able to optimize the different copies for different types of queries. In [9], we explore this to compute different vertical partitionings for each copy, i.e. we end up with three different vertical partitionings which in turn are then exploited at query time. In summary, Trojan Layouts improves query runtimes both over row layouts and over PAX layouts by up to a factor 5. Recently, in a follow-up work, we investigated the performance trade-offs of vertical partitioning (aka column grouping) in more detail [8]. A major lesson of that study was that the performance of these layouts depend heavily on the database buffer size. In particular, vertically partitioned layouts only pay off for very small input buffers.

However, more interesting research questions remained: *Would it be possible to also store different clustered indexes for each replica rather than different layouts? And what happens if the query workload is not known at data upload time? Is there a way to adaptively create those clustered indexes?*

We will answer these questions in Sections 4 and 5.

4 HAIL

When we observed the big benefits of having different data representations per data block replica in the Trojan Layouts project, we immediately felt the need of applying the same idea for creating a different clustered *index* per data block replica. This triggered two interesting challenges:

How could we instrument the different copies of an HDFS block to use different clustered indexes? And how could we teach Hadoop to create those indexes without paying a high price for expensive index creation jobs as observed for Hadoop++?

For this we started the HAIL (Hadoop Aggressive Indexing Library) project with the goal of answering these questions [5]. HAIL is an enhancement of HDFS and Hadoop MapReduce that keeps the existing physical

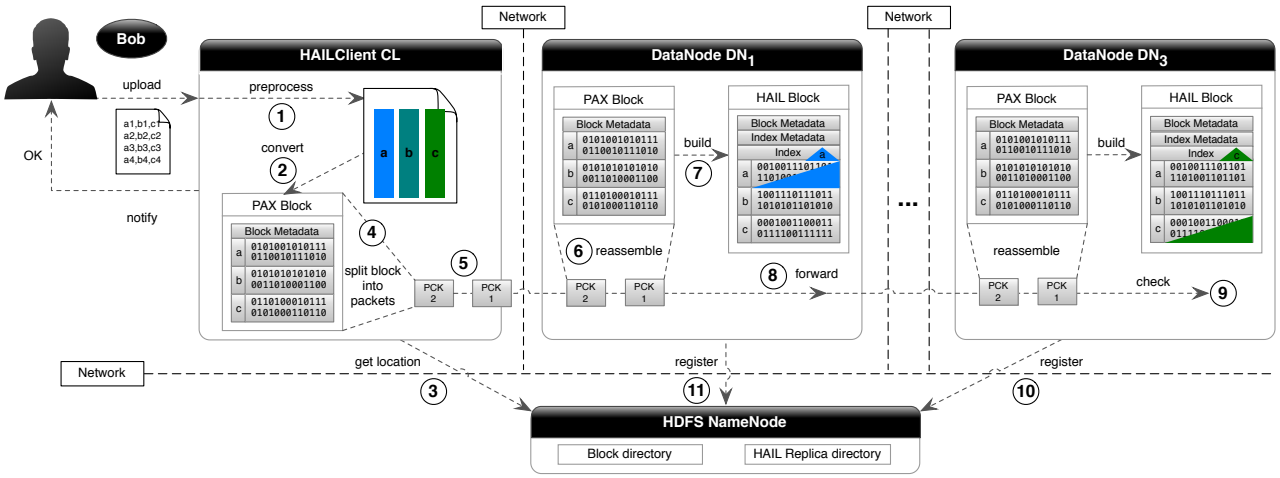


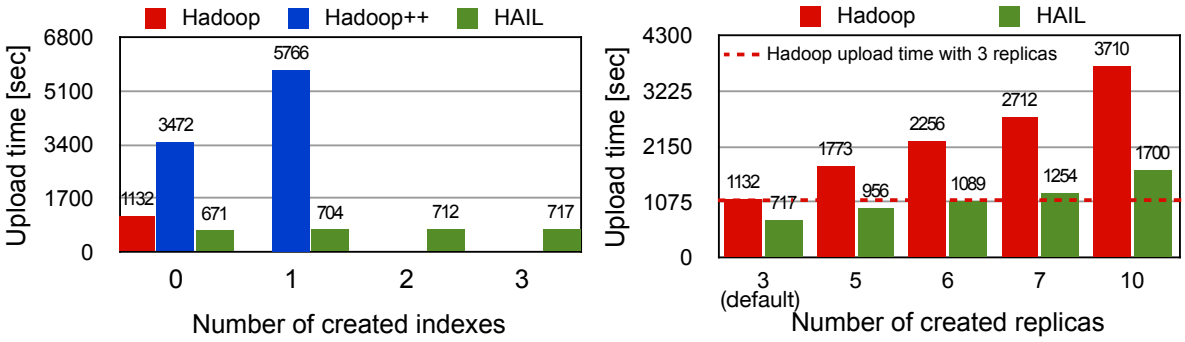
Figure 2: Overview of the HAIL upload pipeline [5]

replicas of an HDFS block in different sort orders and with different clustered indexes. Hence, for a default replication factor of three, three different sort orders and indexes are available for MapReduce job processing. Thus, the likelihood to find a suitable index increases and hence the runtime for a workload improves. In fact, the HAIL upload pipeline is so effective when compared to HDFS that the additional overhead for sorting and index creation is hardly noticeable in the overall process. Why don't we have high costs at upload time? We basically exploit the unused CPU ticks which are not used by standard HDFS. As the standard HDFS upload pipeline is I/O-bound, the effort for our sorting and index creation in the HAIL upload pipeline is hardly noticeable. In addition, as we already parse data to binary while uploading, we often benefit from a smaller binary representation triggering less network and disk I/O.

Let's assume we have a world population table stored in HDFS containing records of type [city, country, population]. If we now want to analyze the population of China, Hadoop MapReduce has to scan the whole world population table and filter for people living in China. While this might be relatively efficient for a country like China, we would still waste our time with reading data that is not needed, and this becomes even more wasteful for a small country such as Luxembourg. If we are now interested in data for a specific city, the traditional Hadoop approach feels like finding a needle in a haystack. This is a typical situation that could be solved with an index, e.g., like the trojan index from Hadoop++. However, creating Trojan indexes is a very costly operation that needs many queries that select on the indexed attribute to amortize [4]. Additionally, the Trojan index will only help when selecting one particular attribute. But what happens if our workload consists of queries selecting on many different attributes like age or name?

Figure 2 shows the HAIL upload pipeline. When uploading a data file to HDFS using the HAIL client, we first analyze the schema of the input (1) and convert the textual data into PAX layout (2). This allows us to save bandwidth, because the binary format is often more space efficient than the textual representation. Like in normal Hadoop, HAIL first asks the Namenode for the locations of all Datanodes that should store a replica of the current block (3). Then, HAIL divides each block into packets (4) and sends them to the first Datanode (the node that was chosen by the Namenode to store the first replica) (5) The first Datanode then reassembles the blocks from the packets (6), sorts the tuples on the index attribute and creates the actual clustered index (7). In parallel, the first Datanode immediately forwards each incoming packet to the next Datanode that stores replica 2. This procedure is repeated for all Datanodes that store a replica until the packets reach the last Datanode. This allows us to create different indexes in parallel on all Datanodes. After reaching the last Datanode, all packets are validated against their checksums (9) Finally, if the blocks could be verified, all Datanodes register their created indexes with the Namenode (10 and 11). With this approach, HAIL even allows us to create more

than three indexes at reasonable costs. Figure 3 shows a comparison of upload times for Hadoop, Hadoop++, and HAIL on our ten-node cluster with a dataset of 130GB. This dataset resembles a typical scientific dataset. A more detailed description of the experiments and the used datasets can be found in our HAIL paper [5].



(a) Upload time for `Synthetic`

(b) Varying replication for `Synthetic`

Figure 3: Upload times when (a) varying the number of created indexex, and (b) the number of data block replicas [5]

In Figure 3(a), we vary the number of indexes from 0 to 3 for HAIL and for Hadoop++ from 0 to 1 (this is because Hadoop++ cannot create more than one index). Notice that we only report numbers for 0 indexes for standard Hadoop as it cannot create any indexes. For all measurements we use three replicas. We observe that HAIL significantly outperforms Hadoop++ by a factor of 5.2 when creating no index and by a factor of 8.2 when creating one index. We observe that HAIL also outperforms Hadoop by a factor of 1.6 even when creating three indexes. This is because HAIL's binary representation of the dataset has a reduced size which allows HAIL to outperform Hadoop even when creating one, two or three indexes.

We now analyze how well HAIL performs when increasing the number of replicas. In particular, we aim at finding out how many indexes HAIL can create for a given dataset in the same time standard Hadoop needs to upload the same dataset with the default replication factor of three and creating no indexes. Those results are presented in Figure 3(b). The dotted line marks the time Hadoop takes to upload with the default replication factor of three. We see that HAIL significantly outperforms Hadoop for any replication factor by up to a factor of 2.5. More interestingly, we observe that HAIL stores six replicas (and hence it creates six different clustered indexes) in a little less than the same time Hadoop uploads the same dataset with only three replicas without creating any index. Still, when increasing the replication factor even further for HAIL, we see that HAIL has only a minor overhead over Hadoop with three replicas only.

A more detailed description of the HAIL upload pipeline that discusses some interesting implementation challenges like adapting Hadoop's packeting and checksumming, Namenode extension, index structure, and fault tolerance can be found in our paper [5].

From these result, we can see a huge improvement for indexing overhead when compared to Hadoop++ and conclude that HAIL provides efficient indexing of many attributes with no or almost invisible overhead. But how can we now use the HAIL indexes and what are the corresponding improvements in terms of query performance? There are at least three options:

1. We can analyze the user-provided map()-function using static code analysis. Then we rewrite the map()-function automatically against our indexes. This approach is fully user transparent. This type of code analysis has already been successfully done in [2] and could be extended to exploit HAIL indexes as well.

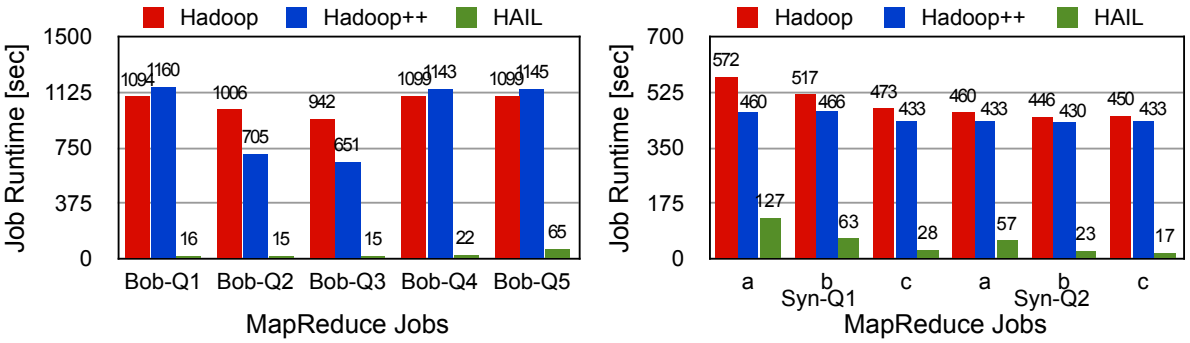
2. We allow users to annotate the map-functions slightly. This approach is not fully user transparent yet minimally invasive. A simple example would be to find the names of all people living in Luxembourg. If

we assume that 'name' is the first attribute and 'country' is the second attribute in the world-population dataset, we simply annotate the map function in Java with

```
@HailQuery(filter="@2='Luxembourg'",
           projection={@1}).
```

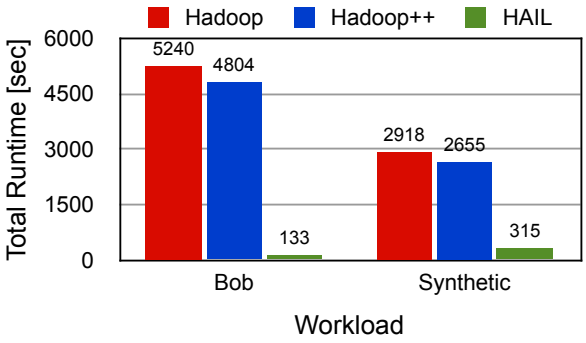
This has the effect that the dataset is pre-filtered and only the attribute name from tuples where country equals to Luxembourg are passed to the map function.

3. The third approach is to modify the applications sitting on top of HDFS or Hadoop MapReduce. As HAIL is a replacement for HDFS, user transparency may be achieved by modifying any software layer on top. For instance, Hive [6] and Pig [13] output machine-generated MapReduce programs; Impala [7] operates directly on flat HDFS files. For these systems, it would be straight-forward to change their MapReduce program generation to exploit HAIL indexes — similar to changing a DB-optimizer to create physical plans using index access paths.



(a) Bob queries

(b) `Synthetic` queries



(c) Total workload

Figure 4: End-to-end job runtimes for two different workloads [5]

Figure 4 illustrates the query performance of HAIL compared to Hadoop and Hadoop++. We clearly observe that HAIL significantly outperforms both Hadoop and Hadoop++. We see in Figure 4(a) that HAIL outperforms Hadoop up to a factor of 68 and Hadoop++ up to a factor of 73 for a log analysis workload (Bob queries). For a Synthetic workload (Figure 4(b)), we observe that HAIL outperforms Hadoop up to a factor of 26 and Hadoop++ up to a factor of 25. Overall, we observe in Figure 4(c) that using HAIL we can run all five queries 39x faster than Hadoop and 36x faster than Hadoop++. We also observe that HAIL runs all six Synthetic queries 9x faster than Hadoop and 8x faster than Hadoop++.

When developing HAIL we learned that the high scheduling overhead of MapReduce tasks is a severe problem when improving the performance of block accesses. All improvements can be eaten up by this overhead. HAIL reduces this overhead significantly using a novel splitting policy at query time (HAIL scheduling). At its core, HAIL scheduling assigns multiple index accesses to a single map task. Thus, we avoid the Hadoop MapReduce overheads for scheduling multiple map waves (see [5] for details and also failover experiments). Overall, using HAIL scheduling we achieve the performance seen in Figure 4.

5 LIAH

One of the disadvantages of HAIL is that the performance of an incoming MapReduce job can only be improved if HAIL creates indexes on the "right" attributes during upload. The set of "right" indexes may be computed by a what-if-analysis. However, what-if-analyzes requires the workload to be known *before* upload — this might not always be the case. Therefore, another challenge remains: *What happens if we do not know the workload beforehand? How can we teach HAIL to adapt its indexes to the workload?*

For this we propose LIAH (Lazy Indexing and Adaptivity in Hadoop) [17] to improve the total runtime of those tasks dramatically.

LIAH effectively piggybacks adaptive index creation on the existing MapReduce job execution pipeline. In particular, we show how to parallelise indexing with ongoing computations of map tasks and disk I/O. All this without any additional data copy in main memory and minimal synchronisation. A particularity of LIAH is that we always index a data block entirely, i.e. in a single pass. As a result, LIAH allows map tasks of future jobs to perform index accessess. In addition, it also frees future map tasks from costly extra I/O-operations for refining indexes. This is in contrast to existing work on adaptive indexing which requires a large number of queries in order to fully refine the indexes. In addition, these works are not designed to preserve the failover properties of HDFS. Hence we had to come up with new ways to enable adaptivity in a distributed system like Hadoop.

LIAH may be used either on top of an unmodifed Hadoop installation or on top of HAIL. This means, it does not matter whether we already have indexes on the data. Similarly to existing adaptive indexing approaches, we analyze the incoming map()-function for its access patterns — using the same analysis techniques used at query time for HAIL and already explained above (code analysis, annotations or application information). Based on these access patterns, we decide which indexes to create. As an ongoing map()-function not finding a suitable index has to scan all data anyways, we simply piggy back on the I/O performed for this scan. However, we do not create indexes for each block in the input, but rather index only one out of ρ blocks. Here, ρ is the *offer rate* determining the fraction of blocks being indexed for a given MapReduce-job, e.g. for $\rho = 1/3$ we require three MapReduce-jobs to adaptively create indexes on all blocks. Notice that $\rho = 1/3$ does *not* imply that the input file is fully read three times. This is because as the first MapReduce-job indexes $1/3$ as a side-effect, the second MapReduce-job may already perform index access on the indexed blocks, i.e. for $1/3$ of the blocks we perform index accesses, for $2/3$ we perform scans.

Many more indexing options exist including *Eager Indexing*, i.e. throttle the adaptive indexing effort to a user-defined SLA; and *Lazy Projection*, i.e. create clustered indexes only for a subset of the attributes, then recluster the missing attributes lazily on demand. We encourage the reader to see [17] for details.

Here we just want to briefly show some performance results of our techniques. Figure 5 shows the job runtimes for Hadoop, HAIL (without precreated indexes) and LIAH (using different offer rates ρ). Overall, we clearly see in both computing clusters that LIAH improves the performance of MapReduce jobs linearly with the number of indexed data blocks.

Figure 5 shows the job runtimes for the `UserVisit` dataset for two different clusters. Overall, we clearly see in both computing clusters that LIAH improves the performance of MapReduce jobs linearly with the number of indexed data blocks. In particular, we observe that the higher the offer rate, the faster LIAH converges to a complete index. However, the higher the offer rate, the higher the adaptive indexing overhead for the initial job

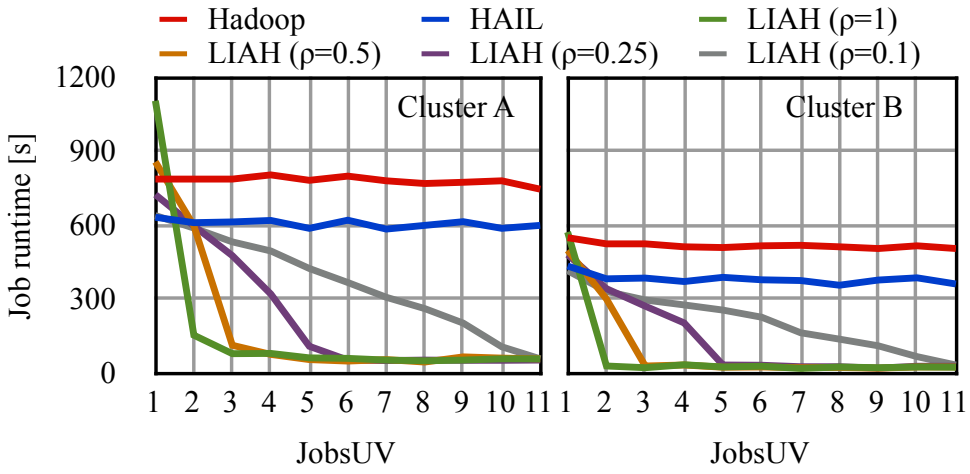


Figure 5: LIAH performance when running a sequence of MapReduce jobs over `UserVisits` [17]

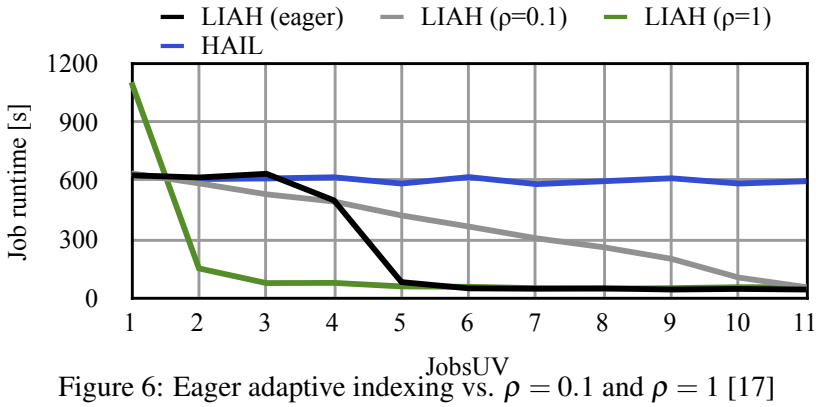


Figure 6: Eager adaptive indexing vs. $\rho = 0.1$ and $\rho = 1$ [17]

(`JobUV1`). Thus, users are faced with a natural tradeoff between indexing overhead and the required number of jobs to index all blocks. But, it is worth noting that users can use low offer rates (e.g. $\rho = 0.1$) and still quickly converge to a complete index (e.g. after 10 job executions for $\rho = 0.1$). In particular, we observe that after executing only a few jobs LIAH already outperforms Hadoop and HAIL (without appropriate indexes) significantly. As soon as LIAH converges to a completely indexed dataset, LIAH significantly outperforms HAIL by up to a factor of 24 and Hadoop by up to a factor of 32.

Figure 6 show the result for eager adaptive indexing. Here we adapt the offer rate ρ such that the execution time of LIAH is less than or equal to that of standard HAIL. As expected, we observe that LIAH (eager) has the same performance as LIAH ($\rho = 0.1$) for `JobUV1`. However, in contrast to LIAH ($\rho = 0.1$), LIAH (eager) keeps its performance constant for `JobUV2`. This is because LIAH (eager) automatically increases ρ from 0.1 to 0.17 in order to exploit saved runtimes. For `JobUV3`, LIAH (eager) still keeps its performance constant by increasing ρ from 0.17 to 0.33. Now, even though LIAH (eager) increases ρ from 0.33 to 1 for `JobUV4`, LIAH (eager) now improves the job runtime as only 40% of the data blocks remain unindexed. As a result of adapting its offer rate, LIAH (eager) converges to a complete index only after 4 jobs while incurring almost no overhead over HAIL. From `JobUV5`, LIAH (eager) ensures the same performance as LIAH ($\rho = 1$) since all data blocks are indexed, while LIAH ($\rho = 0.1$) takes 6 more jobs to converge to a complete index. See [17] for more experimental results.

6 Lessons Learned and Conclusion

We learned that it is possible to introduce indexing in Hadoop without changing its source-code (Hadoop++). We also saw that substantial performance improvements are possible when HDFS is changed to support multiple physical layouts (Trojan Layouts). An interesting challenge was to instrument HDFS to provide efficient index creation and query processing at the same time (HAIL). We also explored how to teach HAIL to create indexes adaptivley (LIAH). Future work aims at generalizing the different projects into a common storage optimizer as discussed in [10].

Yes, parallel DBMS and Hadoop MapReduce are very different systems — at first sight. In comparison, Hadoop is a young system compared to parallel DBMS and can still be improved in many different ways. The Hadoop ecosystem provides an opportunity for the database community to broaden the impact of our research. It is also an opportunity to revisit design decisions taken in the past and take different routes than the ones we took before. In this spirit, we believe that it will be important to teach efficiency to Hadoop without turning it into yet another parallel DBMS.

Acknowledgments. Research partially supported by BMBF. We would like to thank all authors and team members of the Hadoop++, Cloud Variance, RAFT, Trojan Layouts, HAIL, and LIAH projects for their support.

References

[1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. *VLDB*, pages 169–180, 2001.

[2] Michael J. Cafarella and Christopher Ré. Manimal: Relational Optimization for Data-Intensive Programs. *WebDB*, 2010.

[3] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient Big Data Processing in Hadoop MapReduce. *PVLDB*, 5(12):2014–2015, 2012.

[4] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1-2):515–529, 2010.

[5] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.

[6] Hive, http://hive.apache.org.

[7] Cloudera Impala, https://github.com/cloudera/impala.

[8] Alekh Jindal, Endre Palatinus, Vladimir Pavlov, and Jens Dittrich. A Comparison of Knives for Bread Slicing. *PVLDB*, 6(to appear), 2013.

[9] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *SOCC*, 2011.

[10] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. WWHow! Freeing Data Storage from Cages. *CIDR*, 2013.

[11] Alekh Jindal, Felix Martin Schuhknecht, Jens Dittrich, Karen Khachatryan, and Alexander Bunte. How Achaeans Would Construct Columns in Troy. *CIDR*, 2013.

[12] say No! No! and No! (=NoSQL Parody), `http://youtu.be/fXc-QDJBXpw`.

[13] Pig, http://pig.apache.org.

[14] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD*, pages 165–178, 2009.

[15] Jorge-Arnulfo Quiané-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich. RAFT at Work: Speeding-Up MapReduce Applications under Task and Node Failures. In *SIGMOD*, pages 1225–1228, 2011.

[16] Jorge-Arnulfo Quiané-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich. RAFTing MapReduce: Fast recovery on the RAFT. *ICDE*, pages 589–600, 2011.

[17] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards Zero-Overhead Adaptive Indexing in Hadoop. `http://arxiv.org/pdf/1212.3480v1.pdf` [cs.db], TR 12/2012.

[18] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1):460–471, 2010.

Managing Skew in Hadoop

YongChul Kwon[1], Kai Ren[2], Magdalena Balazinska[1], and Bill Howe[1]
[1] University of Washington, [2] Carnegie Mellon University
{yongchul,magda,billhowe}@cs.washington.edu,kair@cs.cmu.edu

Abstract

*Challenges in Big Data analytics stem not only from volume, but also variety: extreme diversity in both data types (*e.g.*, text, images, and graphs) and in operations beyond relational algebra (*e.g.*, machine learning, natural language processing, image processing, and graph analysis). As a result, any competitive Big Data system must support some form of parallel user-defined operations (UDOs) that can capture complex data processing tasks over complex data types without changing the core of the parallel data processing engine. Hadoop and other popular systems have been shown to provide a convenient programming model for implementing parallel UDOs, but the "black-box" nature of UDOs complicates the automatic load balancing required to achieve parallel scalability. In this paper, we present an overview of some of our recent work that tackles the problem of load imbalance (*a.k.a. *skew) in parallel UDO evaluation. We first discuss the prevalence of skew in today's applications and clusters. We then discuss our experience with static and dynamic methods for mitigating it.*

1 Introduction

Users today are increasingly demanding support for complex Big Data analytics that go beyond traditional relational algebra operations: they need to process unusual data types (*e.g.*, text, arrays, and graphs), apply sophisticated statistical models, and adapt specialized domain-specific techniques. These requirements translate into an increased demand for parallel user-defined operations (UDOs). The MapReduce abstraction [4] and its implementation in Hadoop [9] are well-known for their effective support of UDOs in the form of pairs of map and reduce operations. As an example, the Mahout library [27] provides a set of popular machine learning algorithms in Hadoop — tasks that are difficult to implement in conventional database systems.

Although Hadoop simplifies the expression of UDOs, adequate performance is by no means guaranteed (*e.g.*, [18, 24]). UDOs complicate algebraic reasoning and other simplifying assumptions relied on by the database community to optimize query execution. Instead Hadoop developers commonly rely on non-generalizable "tricks" to achieve high performance: ordering properties of intermediate results, custom partitioning functions, and assumptions about the number of partitions. For example, the Hadoop-based sort algorithm that won the terasort benchmark in 2008 required a custom partition function to prescribe a global order on the data, and relied on assumptions about key distribution that were part of the published benchmark [21].

One of the key challenges in the parallel execution of UDOs is, of course, load balancing across UDO partitions. While load balancing has extensively been studied for relational operators [5, 26], the problem has long

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

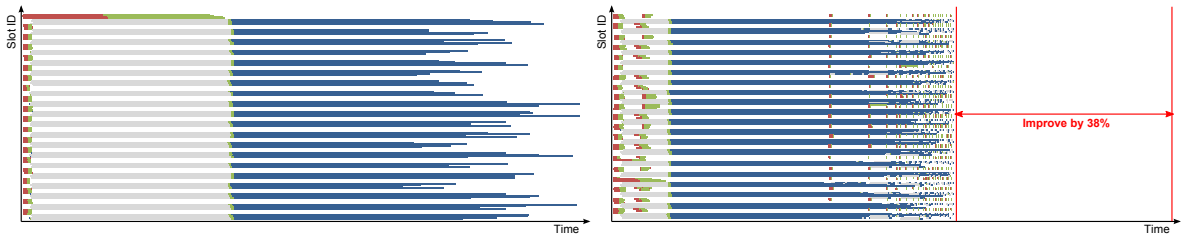


Figure 1: Execution details of the CloudBurst application (a) Original execution (b) Execution with load rebalancing is 38% faster. Red bars correspond to map tasks and blue ones to reduce tasks.

been under-studied for UDOs. In this paper, we summarize some of our recent work [14, 15, 16, 17, 24] that tackles the challenge of effective UDO parallelization in a Hadoop cluster in face of load imbalances: We first show a concrete example of load imbalance in an existing application and its impact on the execution time. We then summarize results from a measurement study in three Hadoop clusters showing, more generally, the prevalence of load imbalance problems in Hadoop clusters. Third, we discuss two techniques that we have developed for mitigating load imbalances in Hadoop applications either statically before execution or dynamically at runtime. We discuss the lessons learned from these two approaches and also point readers to other recent work that falls under each umbrella.

2 Prevalence of Skew

We first consider a concrete example of a user-defined operation that exhibits a significant load imbalance, also referred to as *skew*.

CloudBurst [25] is a MapReduce implementation of the RMAP algorithm for short-read gene alignment.[1] CloudBurst aligns a set of genome sequence reads with a reference sequence. This application is thus somewhat similar to a similarity join: it takes two datasets as input (the reads and the reference sequence) and identifies similar pairs drawing one item from each dataset. In the map phase, it extracts and partitions the reads and reference on their n-grams. Pairs of substrings that share an n-gram are tested for approximate alignment in the reducers. Frequent n-grams are handled similarly to frequent join keys in SkewedJoin [5]: frequent n-grams from a reference sequence are replicated, and frequent n-grams from a read are distributed in a round-robin fashion. Figure 1(a) shows the timing chart of an execution of the CloudBurst application.[2] The total runtime for the job is over 8 hours. The runtimes of the map tasks exhibit a bimodal distribution. We verified that the two modes correspond to the two input datasets. Although there is no significant skew within each mode, the MapReduce job is experiencing skew because the two modes coexist in a single job. The reduce phase also exhibits skew. Even though the partition function distributes keys evenly across reducers (not shown in the figure), some reducers are still assigned more data simply because the assigned key groups contain significantly more values. As a result, the runtime of the reducers is also highly uneven. Figure 1(b) shows how much these types of load imbalances affect the overall runtime of a job. In this application, re-balancing load to eliminate skew, cuts the runtime by nearly 40%. The figure shows the runtime when using the SkewTune system [17] that we discuss in Section 5.

The example illustrates how skew can arise in a UDO and how significantly it can affect the runtime. Our case study paper [15] shows additional such examples. But how prevalent is this problem?

[1] `http://rulai.cshl.edu/rmap/`

[2] We ran the CloudBurst job on a biology dataset [11]. For each alignment, we allowed up to 4 mismatches including insertion and deletion. We used 160 map tasks and 128 reduce tasks for the entire alignment. We used 64 reduce tasks to process low-complexity fragments. The reduce phase processed 128 sequences at a time (first loading data from the reference dataset in memory, then processing 128 sequences from the query dataset in a batch).

Cluster	Duration	Start Date	End Date	Successful/Failed/Killed Jobs	Users
OPENCLOUD	20 months	2010 May	2011 Dec	51975/4614/1762	78
M45	9 months	2009 Jan	2009 Oct	42825/462/138	30
WEB MINING	5 months	2011 Nov	2012 Apr	822/196/211	5

Table 2: Summary of analyzed workloads

Stragglers	Map	Reduce	Map $\wedge$ Reduce	Map $\vee$ Reduce
OPENCLOUD	2967 (58%)	1940 (38%)	1109 (22%)	3798 (74%)
M45	3643 (55%)	2659 (40%)	1310 (20%)	4992 (75%)
WEB MINING	140 (38%)	35 (9%)	18 (5%)	157 (43%)

Table 3: Number and fraction of jobs that have stragglers in map, in reduce, and both phases among jobs longer than five minutes. More than 40% of jobs running longer than five minutes have at least one straggler.

To answer this question, we analyzed execution logs from three Hadoop MapReduce clusters used for research: OPENCLOUD, M45, and WEB MINING. The three clusters have different hardware and software configurations and range in size from nine nodes (WEB MINING), to 64 nodes (OPENCLOUD), to 400 nodes (M45). The clusters are used primarily for scientific research in a wide variety of domains: computational astrophysics, computational biology, computational neurolinguistics, information retrieval, text and web mining, image and video analysis, security malware analysis, graphs and social networking analysis, cloud computing systems development, and others. Table 2 summarizes the duration of each collected log and the number of Hadoop jobs that it covers. The data in each log comes from the Hadoop job configuration files and job history files. More details about the collected data (and its analysis) can be found in our technical report [24].

To assess the prevalence of the skew problem, we count the number of jobs with stragglers (*i.e.*, long-running tasks) considering only jobs with runtimes of at least 5 min (the penalty from skew is insignificant in a short job) and at least two tasks (a straggler task is only defined relative to some other task).

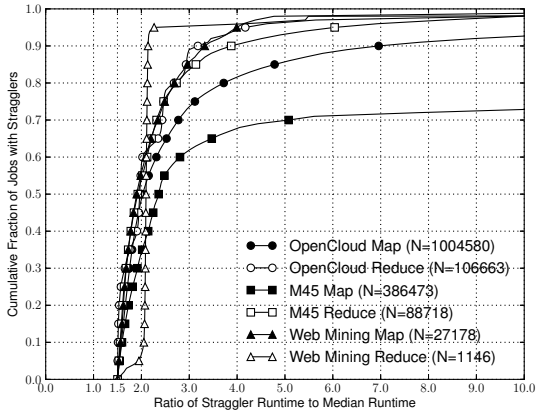
Ananthanarayanan *et al.* [2] categorize a task as a straggler if the task takes at least 50% longer than the median task in the same phase of the same job. Following this definition, Table 3 shows the number of jobs that have stragglers in the map phase, the reduce phase, or both. Overall, more than 40% (and up to 75%) of jobs that run longer than five minutes have at least one straggler. We also find that even the map phase, which is usually regularized by assigning a fixed amount of bytes to each task, frequently experiences the straggler problem. This finding confirms that the input data size alone is not a good indicator of task runtime in these three clusters. The same finding was reported in the analysis of enterprise clusters [3].

Figure 2 shows the distribution of relative runtimes of straggler tasks with respect to the median task runtime of the same phase in the job. In all three clusters, 75% of reduce-side stragglers complete within 2.5x of the median runtime. Map stragglers have larger variations across clusters: 55% and 65% of map straggler tasks complete within 2.5x of median runtime in M45 and OPENCLOUD respectively. As observed in Ananthanarayanan *et al.* [2], the distribution is heavy-tailed. We also show the values at the 99%, 99.9%, and 100% percentiles in the table. Some straggler tasks run orders of magnitudes longer than the median runtime.

The straggler problem is thus prevalent in all clusters, and slow tasks frequently run more than 2.5x slower and sometimes orders of magnitude slower than the median.

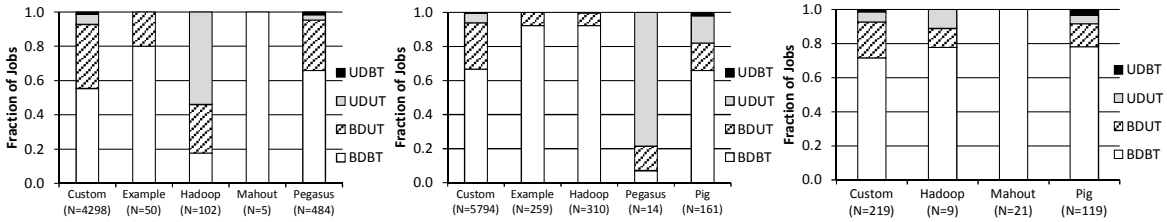
3 Causes of Skew

It is well-known that skew has many causes [2, 4, 30]. The original MapReduce paper [4] considered the problem of skew caused by hardware malfunction or resource contention. To address this type of skew, MapReduce and Hadoop include a mechanism where the last few tasks in a job are speculatively replicated on a different machine



Percentile		99	99.9	100
OPENCLOUD	Map	70.0	1106.0	23068.5
	Reduce	15.3	64.9	54692.5
M45	Map	15.3	153.0	1537.5
	Reduce	11.4	143.8	1267.4
WEB MINING	Map	11.2	66.9	170.7
	Reduce	46.4	404.3	409.3

Figure 2: Cumulative fraction of the ratio of straggler runtime to median task runtime. *N* is the number of straggler tasks per cluster, per phase. The table shows straggler runtime ratios at specific percentiles.



(a) OPENCLOUD (b) M45 (c) WEB MINING

Figure 3: Distribution of map task runtimes with respect to # of input records, grouped by application types. *N* is the number of successful jobs that run map functions in each category. The labels indicate the category: (U)BD(U)BT stands for (un)balanced input data, (un)balanced runtime.

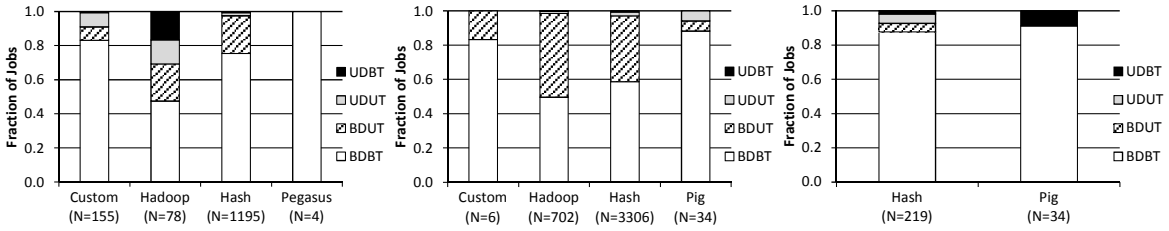
and the job completes when the fastest replicas of these final tasks complete. More commonly, skew occurs when data is not evenly partitioned across tasks. This type of skew, called *data skew*, typically affects reducers since map tasks are normally assigned same-size chunks of input data. For reducers, the problem can occur either when a reducer processes a larger number of keys or a larger number of values than other reducers. In our work, we further find that a surprisingly large number of applications suffer from skew even when all UDO partitions are assigned the same amount of data [15]. The map phase of CloudBurst is one such example.

We analyze the prevalence of skew caused by uneven data allocation or uneven processing times in the three Hadoop cluster logs. For each phase of each job, we compute the ratio of the maximum task runtime in the phase to the average task runtime in that phase. We classify phases where this ratio is greater than 0.5 (meaning that at least one straggler took twice as long to process its data as the average) as *unbalanced in time (UT)*. Otherwise, the phase is said to be *balanced in time (BT)*. We compute the same ratio for the input data and classify phases as either balanced or unbalanced in their data (BD or UD).

Map Phase: Figure 3 shows the result for the map phases. We group the jobs by application type based on the mappers package names,[3] then break the results into four different types based on whether the input data and/or the runtime are balanced or not (*i.e.*, (U)BD(U)BT as defined in the previous paragraph).

As expected for the map phase, most jobs are balanced with respect to data allocated to tasks. However, a

[3] *Hadoop* corresponds to map functions that are part of the Hadoop MapReduce framework (*e.g.*, *IdentityMapper*). *Example* represents map functions defined in example applications in Hadoop (*e.g.*, *WordCount*). *Pegasus* is a graph mining system running on top of Hadoop [12]. If we do not recognize the package name, we assume that the users wrote the functions, and label them as *Custom*.



(a) OPENCLOUD (b) M45 (c) WEB MINING

Figure 4: Distribution of reduce task runtimes with respect to # of reduce keys grouped by partition functions. The labels indicate the category: (U)BD(U)BT stands for (un)balanced input data, (un)balanced runtime.

significant fraction of jobs, more than 20% for all but Mahout in the OPENCLOUD cluster, remain unbalanced in runtime and are categorized as BDUT. These results agree with the result of the previous study in enterprise clusters [3]. Mahout is effective at reducing skew in the map phase, clearly demonstrating the advantage of specialized implementations. Interestingly, Pig produces unbalanced map phases similar to users' custom implementations. Overall, allocating data to compute nodes simply based on data size alone is insufficient to eliminate skew.

Reduce Phase: We perform a similar analysis for the reduce phase by using the number of reduce keys as the input-size measure. Instead of application types, we group the jobs by the partition function employed. The partition function is responsible for redistributing the reduce keys among the reduce tasks. We label the partition functions by inspecting their package names.[4] If the package name is not well-known (*e.g.*, `foo.bar`), we assume that it is customized by the user (labeled *Custom*). Figure 4 shows the results.[5] Overall, hash partitioning effectively redistributes reduce keys among reduce tasks for more than 92% of jobs in all clusters (*i.e.*, BDBT+BDUT). Interestingly, we observe that as many as 5% of all jobs in all three clusters experienced the *empty reduce* problem, where a job has reduce tasks that processed *no* data at all due to either a suboptimal partition function or because there were more reduce tasks than reduce keys. For the jobs with a balanced data distribution, the runtime is still unbalanced (*i.e.*, BDUT jobs) for 22% and 38% of jobs in the OPENCLOUD and M45 clusters, respectively. In both these clusters, custom data partitioning is more effective than the default scheme in terms of balancing both the keys and the computation. Other partitioning schemes that come with the Hadoop distribution do not outperform hash partitioning in terms of balancing data and runtime. A noticeable portion of UDBT jobs (in which data are not balanced but runtime is balanced) use the total order partitioner, which tries to balance the keys in terms of the number of values. Pig, which uses multiple partitioners, consistently performs well in both M45 and WEB MINING clusters.

Overall, UDOs in Hadoop can thus suffer from skew when the data is unevenly distributed to UDO partitions but also when the computation is uneven across the partitions.

4 Tackling Skew with Static Optimization

When the cause of skew is an uneven data distribution across UDO partitions or an uneven processing time across partitions, MapReduce's speculative execution mechanism is not helpful. This approach re-executes the same task on the same input data but on a different machine. It would lead to the same, slow execution time.

Instead, an alternate approach is to use finer-grained partitions so as to minimize the size of serial units of work. The problem with this approach is that finer-grained partitions increase overheads, especially in a system

[4] *Hadoop* represents all partition functions in the Hadoop distribution except the default hash partitioning (labeled *Hash*).

[5] In the M45 workload, the number of reduce keys was not recorded for the jobs that use the new MapReduce API due to a bug in the Hadoop API. The affected jobs are not included in the figure.

such as Hadoop, where the overhead of each new task is not negligible.

In SkewReduce [14], we developed instead a static optimizer that takes a UDO as input and outputs a data partition plan that strives to balance load across processing nodes. SkewReduce does not simply balance the amount of data assigned to each operator partition, but rather the expected processing time for the assigned data.

While a static optimizer could be applied to different types of UDOs, SkewReduce was designed for *feature extracting applications*. In these applications, data items (events, particles, pixels) are embedded in a metric space, and the task is to identify and extract emergent features (populations, galaxies) from the low-level data. These applications are captured by two UDOs: one that extracts features from sub-spaces and another that merges features that cross partition boundaries.

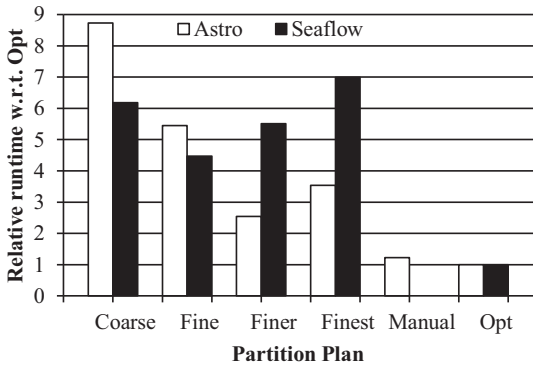
The key idea behind SkewReduce is to ask the user for extra information about their UDOs (the feature extraction and merge functions). The user creates additional *cost functions* that characterize the UDOs runtimes. A cost function takes as input a sample S of the input data, the sampling rate α that this sample corresponds to, and B, the bounding hypercube where the sample was taken from. The function returns a real number, which should represent an estimate of the processing time for the data. The sampling rate α allows the cost function to properly scale up the estimate to the overall dataset. The ideal cost function returns values proportional to the actual cost of processing the data partition delimited by the hypercube. The quality of the SkewReduce plans depends on the quality of the cost functions. However, we found that even inaccurate cost functions could help reduce runtimes compared to no optimization at all. For the domain experts such as scientists and statisticians, we posit that writing a cost model is easier than debugging and optimizing distributed programs. The cost models are specific to the implementation and can be reused across many data sets for the same UDOs.

Given the cost models, a sample of the input data, and the cluster configuration (*e.g.*, the number of nodes and the scheduling algorithm), SkewReduce searches a good partition plan for the input data by (a) applying finer grained data partitioning if significant data skew is expected for some part of the input data, (b) keeping coarse grained partitions when data skew is not anticipated, and (c) balancing the partition granularity in a way that minimizes expected job completion time under the given cluster configuration.

More specifically, the algorithm proceeds as follows. Starting from a single partition that corresponds to the entire hypercube bounding the input data I, and thus also the data sample, S, the optimizer greedily splits the most expensive leaf partition in the current partition plan. The optimizer stops splitting partitions when two conditions are met: (a) All partitions can be processed and merged without running out of memory; (b) No further leaf-node split improves the runtime: i.e., further splitting a node increases the expected runtime compared to the current plan. To verify the second condition, the SkewReduce optimizer first checks that the original execution time of the partition is greater than the expected execution time of the two sub-partitions running in parallel, followed by a newly added merge function to reconcile features found in individual sub-partitions. If the condition is satisfied, SkewReduce further verifies that the resulting tasks can also be scheduled in the cluster in a manner that reduces the runtime. It proceeds with the split only when both conditions are met.

To show the effectiveness of SkewReduce, we built a prototype running on top of Hadoop. Figure 5 shows the relative runtimes when applying the Friends-of-Friends clustering algorithm, a popular feature-extraction application used in astronomy, to two datasets, one from the astronomy domain and the other from oceanography. As the figure shows, SkewReduce yields runtimes 6X to 8X lower than a default Hadoop configuration. It is at least 2X faster than the optimal uniform data partition, whose choice depends on the dataset being processed.

Lessons learned: The key lesson learned from our SkewReduce work is that static optimization of the input data partition can dramatically improve runtimes in the presence of significant skew, caused by uneven processing times. The optimization time was short compared to the actual query execution time (*i.e.*, minutes vs hours or seconds vs minutes). Interestingly, runtimes improved even when the user-provided cost functions were not perfectly accurate but were good enough to identify expensive regions in the data. The drawback of this approach is two-fold: (1) it puts an extra burden on the user who must provide cost functions and (2) it cannot handle any unexpected conditions at runtime. We refer readers to our full paper for details [14].



Completion time (hours for Astro, minutes for Seaflow)

Dataset	**Coarse**	**Fine**	**Finer**	**Finest**	**Manual**	**Opt**
Astro	14.1	8.8	4.1	5.7	2.0	1.6
Seaflow	87.2	63.1	77.7	98.7	-	14.1

Figure 5: Relative runtimes of different partitioning strategies compared with SkewReduces' optimized plan (Opt). The table shows the actual completion times for each strategy (units are hours for Astro and minutes for Seaflow). Manual plan is shown only for the Astro dataset.

5 Mitigating Skew Dynamically

To address the limitations of SkewReduce, we built SkewTune [17], a version of Hadoop extended with the ability to *dynamically* re-balance load across map or reduce tasks in a job. Because it re-balances load automatically, SkewTune must make an assumption about Hadoop programs: that separate invocations of the user-defined map and reduce functions are independent of each other. Load can thus be re-balanced at the granularity of records for map tasks and key-groups for reduce tasks.

SkewTune is designed for MapReduce-type systems, where each operator reads data from disk and writes data to disk and is thus decoupled from other operators in the data flow. SkewTune continuously monitors the execution of a UDO and detects the current bottleneck task that dominates and delays the job completion. Once such a task is identified, the task is stopped, and its unprocessed input data is repartitioned among idle nodes. The repartition only occurs when there is an idle resource (*i.e.*, the MapReduce scheduler runs out of other tasks to schedule) or when new nodes are dynamically added to accelerate the job (*e.g.*, using spot instances in Amazon EC2). Thus, if there are no tasks experiencing significant data skew, SkewTune imposes no overhead. If a node is idle and at least one task is expected to take a long time to complete, SkewTune stops the task with the longest expected time-remaining. It repartitions and thus parallelizes the remaining input data for that task taking into consideration to expected future availability of all nodes in the cluster. SkewTune continuously detects and removes bottleneck tasks until the job completes.

When repartitioning a task, SkewTune takes into account the predicted availability of all nodes in the cluster to minimize the job completion time. The availability is estimated from the progress of running tasks. The remaining unprocessed input data of a task is repartitioned in a way that fully utilizes nodes that are available immediately or are expected to become available in the near future. SkewTune also minimizes the side-effect of repartitioning so that the original output can be reconstructed simply by concatenating the output of split tasks.

We implemented the SkewTune strategy in the Hadoop MapReduce engine. Experimental results show that SkewTune can significantly improve completion time by up to factor of four without code changes. At the same time, it imposes, negligible overheads in the absence of skew.

Lessons learned: The most important lesson learned from the SkewTune approach is that dynamic load re-allocation is a workable solution for UDOs that follow a well-specified API, such as Hadoop, where the system knows how to rebalance load without breaking the application. The overhead of load re-allocations is small compared to the potential savings. At the same time, this approach is more flexible than static planning since it can handle skew independent of its cause: cluster conditions, hardware malfunction, mis-configuration, unlucky input data order, etc. We refer readers to our full paper for details on the SkewTune approach [17].

Both SkewTune and SkewReduce are freely available at `http://nuage.cs.washington.edu/`

6 Related Approaches

Skew-resilient operators: One approach to handling skew is through the implementation of skew-resilient operators. This approach has extensively been studied for the relational join operator (*e.g.*, [5, 30]). Pig, a declarative layer on top of Hadoop, implements an algorithm proposed in the parallel database literature [5] to handle data skew in a join algorithm [6]. Pig also includes mechanisms to mitigate skew in ORDER BY and GROUP BY operators [6]. Vernica *et al.* and Metwally *et al.* studied set-similarity joins in MapReduce [20, 29]. To handle skew, Vernica *et al.* first scans the input data to collect statistics on frequencies and load balances according to the statistics. Metwally *et al.* analyze set similarity measures, extracts a common structure in computing similarities, and design a series of MapReduce steps according to the structure. Metwally *et al.* also analyze skew that may occur in each step, and describe skew-handling strategies. There also exist work on handling skew in relational aggregate operators [26] including studies based on Hadoop [19].

Studies of skew: Qiu *et al.* implemented three applications for bioinformatics using cloud technologies and reported their experience and measurement results [22]. Although they also found skew problems in two applications, they discussed a potential solution rather than tackling the problem. For data and computation imbalance problems, Ke *et al.* also found that these problems are prevalent in a variety of industrial applications [13]. Ananthanarayanan *et al.* studied the causes to outlier (*i.e.*, straggler) tasks and ways of mitigating outlier tasks caused by data imbalances or resource contention [2].

Speculative execution enhancements: Several approaches have improved the basic MapReduce speculative execution. The LATE scheduler [31] speculatively runs the task with the longest estimated time remaining. It also schedules these tasks only on fast nodes and it limits resources used for speculation. The Mantri system [2] further only restarts a task when its runtime is inconsistent with its input data size and improves the network-aware task placement.

Static optimizations: Related to SkewReduce, recent work also studied the possibility to leverage cost models to improve load balance [7, 8, 23]. That work was specific to balancing load in the reduce phase of a Hadoop job. Gufler *et al.* [7, 8] support non-linear cost models for reducers as functions of the number of bytes and the number of records a reducer needs to process. Their algorithm splits the reduce input data into a fixed number of small partitions, estimates their cost, and distributes the small partitions accordingly. A second algorithm further splits partitions that grow too large. The LEEN approach [10] further tries to balance load (*i.e.*, fairness) with data locality in the reduce phase of a job.

Dynamic methods: Related to SkewTune, other techniques have recently explored dynamic load allocation in MapReduce [1, 28]. Vernica *et al.* propose an adaptive MapReduce system using situation-aware mappers that continuously monitor the execution of all mappers and adaptively resplit the map input data [28]. With an adaptive combiner and partitioner, the system further tries to balance the reduce input. However, the situation-aware mappers can not handle computational skew at the reducers, where some key-groups take longer to process than others.

7 Conclusion

The simple MapReduce API provides users great flexibility when implementing UDOs but it also makes users responsible for handling various performance problems. Skew is one challenge that users frequently face today. In this paper, we first reviewed the causes of skew in MapReduce applications and the prevalence of the problem in three real-world Hadoop clusters. We then presented an overview of two systems SkewReduce and SkewTune that mitigate skew statically and dynamically. With little or no user effort, the two systems can significantly improve job completion times compared to the default Hadoop setup. Interesting future work includes but is

not limited to further minimizing user efforts in skew mitigation through better programming interfaces and execution models and static and dynamic analysis or optimizations in multi-tenant environments.

Acknowledgments: We thank Jerome Rolia (HP Labs) from his extensive contributions to this project. This work was partially supported by NSF through CAREER award IIS-0845397, Cluster Exploratory Award IIS-0844572, CRI grant CNS-0454425, SCI-0430781, and CCF-1019104, an HP Labs Innovation Research Award, the Gordon and Betty Moore Foundation, the University of Washington eScience Institute, the Qatar National Research Foundation (09-1116-1-172), the Intel Science and Technology Center on Cloud Computing, Yahoo!, Microsoft Research, and the PDL consortium. We also thank the owners of the logs from the three Hadoop clusters for graciously sharing these logs with us.

References

[1] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of the Third SoCC Conf.*, 2012.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proc. of the 9th OSDI Symp.*, 2010.

[3] Y. Chen, S. Alspaugh, and R. H. Katz. Design insights for MapReduce from diverse production workloads. Technical Report UCB/EECS-2012-17, EECS Department, University of California, Berkeley, 2012.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.

[5] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the 18th VLDB Conf.*, pages 27–40, 1992.

[6] A. F. Gates, J. Dai, and T. Nair. Apache Pigś optimizer. *IEEE Data Engineering Bulletin*, 36(1), 2013.

[7] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Handling data skew in MapReduce. In *The First International Conference on Cloud Computing and Services Science*, 2011.

[8] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in MapReduce based on scalable cardinality estimates. In *Proc. of the 28th ICDE Conf.*, 2012.

[9] Hadoop. `http://hadoop.apache.org/`.

[10] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi. LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud. In *IEEE CloudCom 2010*, pages 17–24, 2010.

[11] M. Kalyuzhnaya, D. Beck, and L. Chistoserdova. Functional metagenomics of methylotrophs. *Methods in Enzymology*, 495, 2011.

[12] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: a peta-scale graph mining system implementation and observations. In *ICDM*, 2009.

[13] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *HotOS*, 2011.

[14] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the First SoCC Conf.*, June 2010.

[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in MapReduce applications. In *The 5th International Open Cirrus Summit*, 2011.

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune in action (demonstration). *Proc. of the VLDB Endowment*, 5(12):1934–1937, 2012.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in MapReduce applications. In *Proc. of the SIGMOD Conf.*, pages 25–36, 2012.

[18] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc. of the 22nd SSDBM Conf.*, 2010.

[19] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A Platform for Scalable One-Pass Analytics using MapReduce. In *Proc. of the SIGMOD Conf.*, June 2011.

[20] A. Metwally and C. Faloutsos. V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. of the VLDB Endowment*, 5(8):704–715, 2012.

[21] O. O'Malley. Apache hadoop wins terabyte sort benchmark. `http://developer.yahoo.com/blogs/hadoop/posts/2008/07/ apache_hadoop_wins_terabyte_sort_benchmark/`.

[22] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon. Cloud technologies for bioinformatics applications. In *Proc of the MTAGS'09 Workshop*, pages 1–10, 2009.

[23] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in MapReduce workloads using progressive sampling. In *Proc. of the Third SoCC Conf.*, 2012.

[24] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: a comparative workload analysis from three research clusters. Technical Report UW-CSE-12-06-01, University of Washington, June 2012.

[25] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, June 2009.

[26] A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *Proc. of SIGMOD Conf.*, 1995.

[27] T. M. Team. Apache Mahout project. `http://mahout.apache.org/`.

[28] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using situation-aware mappers. In *Proc. of the EDBT Conf.*, 2012.

[29] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *Proc. of the SIGMOD Conf.*, pages 495–506, 2010.

[30] C. Walton, A. Dale, and R. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of the 17th VLDB Conf.*, 1991.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of the 8th OSDI Symp.*, 2008.

Apache Pig's Optimizer

Alan F. Gates, Jianyong Dai, Thejas Nair
Hortonworks

Abstract

Apache Pig allows users to describe dataflows to be executed in Apache Hadoop. The distributed nature of Hadoop, as well as its execution paradigms, provide many execution opportunities as well as impose constraints on the system. Given these opportunities and constraints Pig must make decisions about how to optimize the execution of user scripts. This paper covers some of those optimization choices, focussing one ones that are specific to the Hadoop ecosystem and Pig's common use cases. It also discusses optimizations that the Pig community has considered adding in the future.

1 Introduction

Apache Pig [10] provides an SQL-like dataflow language on top of Apache Hadoop [11] [7]. With Pig, users write dataflow scripts in a language called Pig Latin. Pig then executes these dataflow scripts in Hadoop using MapReduce. Providing users with a scripting language, rather than requiring them to write MapReduce programs in Java, drastically decreases their development time and enables non-Java developers to use Hadoop. Pig also provides operators for most common data processing operations, such as join, sort, and aggregation. It would otherwise require huge amounts of effort for a handcrafted Java MapReduce program to implement these operators.

Many different types of data processing are done on Hadoop. Pig does not seek to be a general purpose solution for all of them. Pig focusses on use cases where users have a DAG of transformations to be done on their data, involving some combination of standard relational operations (join, aggregation, etc.) and custom processing which can be included in Pig Latin via *User Defined Functions*, or *UDF*s, which can be written in Java or a scripting language.[1] Pig also focusses on situations where data may not yet be cleansed and normalized. It gracefully handles situations where data schemas are unknown until runtime or are inconsistent. These characteristics make Pig a good choice when doing data transformations in preparation for more traditional analysis (the "T" of ETL). It also makes Pig a good choice for operations such as building and maintaining large data models in Hadoop and doing prototyping on raw data before building a full data pipeline. For a more complete discussion of Pig, Pig Latin, and frequent Pig uses cases see [12].

In this paper, we will discuss some of the performance optimization techniques in Pig, focussing on optimizations specific to working in Hadoop's distributed computing environment, MapReduce.

This paper assumes a working knowledge of MapReduce, particularly Hadoop's implementation. For more information on Hadoop's MapReduce see [13].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] As of Pig 0.10 the available scripting languages for UDFs are Python, Ruby, and JavaScript.

1.1 What to Optimize

In designing and implementing Pig's optimizer Pig developers sought to identify which characteristics of Pig and MapReduce caused the most slow downs in the system. The following discusses the issues that were found to cause the greatest slowdown in Pig's execution.

1.1.1 Number of MapReduce Jobs

Most Pig Latin scripts get compiled into multiple MapReduce jobs. Each MapReduce job adds additional overhead to the query. Hadoop's MapReduce was built around large batch jobs as the primary use case. Little effort has gone into minimizing overheads such as job start up time. As a rule of thumb, starting a MapReduce job takes between 5 and 15 seconds, assuming there is capacity in the cluster to run the job immediately. Also, in Hadoop 1.0 and previous versions there is no ability to declare multiple MapReduce jobs as related. In a busy cluster this means that Pig may have to wait to secure resources for each of its MapReduce jobs, potentially introducing lag time between each job.

Also, Pig moves data between MapReduce jobs by storing it in HDFS. This means that data is written to and then read from disk (and written over the network for redundancy) at every MapReduce boundary.

Finally, a MapReduce job is a blocking operator. MapReduce job N+1 cannot start until all tasks from MapReduce job N have completed. Since all reducers rarely finish together this has the effect of bottlenecking the execution on slower tasks.

All these taken together mean that whenever the work of two or more MapReduce jobs can be consolidated into one, Pig will execute the jobs significantly faster.

1.1.2 Key Skew

MapReduce divides its work between many nodes in the cluster, in both the map and reduce phases. In the map phase the work is generally split by how the data is stored. In the most common case this means each map task operates on one block from HDFS. Depending on the cluster these block sizes are generally between 64M and 512M. In the case of jobs with many smaller files that do not fill a block, a map task may operate on multiple blocks in order to avoid spawning too many map tasks. This heuristic is provided to Pig by MapReduce and gives a very even distribution of processing in the map phase.

In the reduce phase work is distributed based on a key chosen by the MapReduce user (in this case, Pig). When the Pig user specifies a `JOIN`, `GROUP BY`, or `ORDER BY` operation the key of that operation will generally be the key that is used to partition data amongst reducers. In the case where the data is uniformly distributed on this key this provides a good parallel distribution of the work. In our experience the vast majority of human generated data is not uniformly distributed. In most cases it is power law distributed. This limits the ability to parallelize the reduce phase by simply assigning different keys to different reducers. Whichever reducer gets the key with the highest cardinality will receive a disproportionate amount of the work and cannot be sped up. Since it will finish last it will define the time for the whole job. In addition, for any algorithms that require the reducers' data to be held in memory (e.g. the build table in a hash join) the accumulation of a significant percent of the data in a single reducer may exceed the available memory in that reducer.

1.1.3 Amount of Data Shuffled

Between the map and reduce phases of MapReduce, data is hash distributed from the mappers to the reducers. This phase is referred to as the *shuffle*. It is frequently the case that this phase takes longer than either the map or the reduce. The size of the data to shuffle can be reduced by compression, and we have found this trade off to often be worth it for compression algorithms that are not too CPU intensive (such as LZO or Google's

Snappy [1]). Whenever possible it is good to reduce the amount of data being passed from the mapper to the reducer.

1.2 Optimizer Design Choices

Every optimizer only considers a subset of the total optimization space [9]. For Pig, the optimization space is drastically shrunk by two facts: Pig's choice to leave much of the control to the user, and the lack of a cost based optimizer. Pig has chosen to implement different operators for different situations (e.g. hash join and sort join) and let the user choose which to use in a given script. This leaves much control to the user, as well as putting a significant burden on him. This is in line with one of the design goals of Pig: "Pig is a domestic animal", i.e., Pig does whatever the user tells it to do [7].

Pig's optimizer is rule based. In section 5.1, we will talk more about why Pig does not have a cost based optimizer and how to move forward. With a rule based optimizer, exhaustive search in the optimization space is converted to a precondition match for each optimization rule Pig has adopted.

There are two layers of the optimizer in Pig: logical layer and MapReduce layer. A Pig Latin script is first converted into a logical plan by the parser. A logical plan is a DAG of all operators used in the Pig query. In this grand graph, Pig applies optimization rules to split, merge, transform, and reorder operators. A logical plan is then converted to a MapReduce plan[2], which is a fragmented view of the Pig query with the MapReduce job boundaries defined. Reordering operators in this graph is clearly more complex but with the information of MapReduce job boundaries, we can do some further optimizations which are not possible in a grand graph.

Most of the logical layer optimization rules aim to reduce the amount of data in the shuffle phase or the temporary files generated between consecutive MapReduce jobs. Here we list several notable rules in this category:

- *PushUpFilter*: Applying selection earlier, reducing the number of records remaining in the pipeline. If there are multiple conditions in the filter and the filter can be split, Pig splits the conditions and pushes up each condition separately.[3]
- *PushDownForEachFlatten*: Applying `FLATTEN`, which produces a cross product between a complex type such as a tuple or a bag and the other fields in the record, as late as possible in the plan. This keeps the number of records low in the pipeline.
- *ColumnPruner*: Omitting columns that are never used or no longer needed, reducing the size of the record. This can applied after each operator, so that fields can be pruned as aggressively as possible.
- *MapKeyPruner*: Omitting map keys that are never used, reducing the size of the record.
- *LimitOptimizer*: Applying the limit earlier, reducing the number of records. If the limit operator is immediately after a load or sort operator, Pig converts the load or sort operator into a limit-sensitive implementation, which does not require processing the whole dataset.

On the other hand, most of the MapReduce layer optimization rules aim to optimize the MapReduce job properties (e.g., number of MapReduce jobs, whether or not use the combiner, etc.). The MapReduce plan is restructured to a more efficient plan according to the optimization rules. The optimizations discussed in sections 2.3 and 3 are both triggered in this optimizer.

[2]We omit the physical plan in this discussion since is not involved in the optimization. For further details please see [8]

[3]Database texts and papers usually discuss pushing filters down. In Pig we tend to think of load as the top of the graph and store as the bottom, so filters are said to be pushed up rather than down.

2 Handling Key Skew

As discussed earlier, key skew is a major bottleneck for Pig performance. In this section, we will discuss how key skew is handled by the `JOIN`, `ORDER BY`, and `GROUP BY` operators. For a more general discussion of handling skew in MapReduce applications see [19].

2.1 Handling Skew in JOIN

The default join implementation in Pig is hash join. Input relations are assigned to the reducers via a hash function on the join key. The hash function is key skew agnostic so it assigns an unfairly large amount of work to reducers with keys that have a large number of records. It is common to see a single reducer slow down the whole MapReduce job due to a single large key.

The issue of skew in parallel implementations has been discussed in [15]. Based on this paper, Pig added a new join operator, *skewed join*. The idea is to split keys with a high number of records and distribute them to more than one reducer. To implement it the following problems needed solved.

Identifying which keys are large. Pig samples the input data to estimate the number of input records for each key and the average size of each record. The criteria for a key with too many records is that the estimated size of the all the records with that key exceeds a certain percentage of the memory on a reducer. The sampling job is a separate MapReduce job. The added sampling MapReduce job is significantly smaller than the join job in most cases, so the added overhead is tolerable.

How many reducers to use for a large key. To efficiently process the key, Pig assumes each reducer can only take a certain amount of input data. This is related to the memory management model of Pig. If a reducer takes more input data than the available amount of memory, Pig has to spill the input data onto the disk, which slows the pipeline down. Pig is configured by default to allow the amount of input data to occupy 30% of the memory allocated to the JVM. With the JVM configuration information of a node and the estimated number and size of records for a given key, Pig can easily calculate the number of reducers to use for that particular key. Note that if the calculated number of reducers is larger than the number of total reducers of the MapReduce job (usually specified by the user in a `PARALLEL` clause in the Pig Latin script), Pig fails. The user needs to increase the number of reducers and try again.

How to distribute the input data. Hadoop allows the user to specify the `Partitioner` class that is used to distribute data between reducers [13]. A special partitioner is used to distribute a key with a large number of records into the predetermined number of reducers in a round robin fashion. Only the left side relation is used to identify and distribute large keys. Records from the right side relation with the chosen key are duplicated to every reducer that received a section of the chosen key and crossed with the left side relation to produce the final result. As discussed in section 1.2, whether or not to use skewed join is controlled by the user.

Achieving an even balance of work between reducers is not always feasible with skewed join. If the distribution of join keys in the right side input is skewed, the work load of reducers will still be skewed. However, in this algorithm no reducer will be overwhelmed and fail, as can happen in the standard join if the build relation is too large and will not fit in memory. If the distribution of the join keys is even in the right side, then skew join will achieve a balance of work between the reducers.

This algorithm does break the MapReduce rule of "one key, one reducer" and thus can produce outputs that will confuse or surprise down stream users.

2.2 Handling Skew in ORDER BY

`ORDER BY` in Pig Latin indicates a global sort, meaning not only that the output of each reducer is sorted but that all output of reducer N is guaranteed to be less than or equal to the output of reducer N+1. As we know, a reducer sorts the data automatically before processing. Thus a global sort could be achieved by a single

reducer easily, but at the price of poor performance. If multiple reducers are used, the output is sorted within each reducer, however, the outputs of different reducers are independent and the global sort cannot be achieved automatically. Pig uses the following steps to achieve a global sort:

1. A sampling algorithm is used to estimate the range of keys to be sorted and how many records exist for each key.

2. Sort key boundaries for each reducer are determined by the number of reducers and number of records per key. These boundaries are chosen such that it is estimated that the same number of records will be sent to each reducer.

3. Input records are distributed to reducers based on these predetermined boundaries. The boundaries are increasing (or decreasing, in a descending sort), i.e. the boundary for reducer #2 is greater than or equal to the boundary for reducer #1, which means all outputs of reducer #2 are greater than or equal to any output of reducer #1.

4. Each reducer sorts its own input data.

A dedicated sampling MapReduce job is added to process step 1 and 2. A second MapReduce job with a special partitioner is used to process step 3 and 4.

Pig deals with key skew in step 2. When determining the key boundary for each reducer, Pig also balances the workload of all reducers. This is achievable because the estimated size of each key is known in this phase. Unlike skewed join, Pig balances the load of each reducer. As noted in step 3, the boundaries between reducers may be set such that a record could go to more than one reducer (in the case where the upper boundary of reducer #2 is equal to the upper boundary of reducer #1). In this case the partitioner will distribute records in round robin fashion between those reducers. This allows Pig to split keys with large numbers of records across multiple reducers, thus avoiding imbalance between reducers. Using this algorithm Pig is generally able to balance a sort workload such that the longest running reducer finishes in 110% of the time of the average reducer.

As with skew join in the previous section, this algorithm breaks the "one key, one reducer" rule that MapReduce users are accustomed to.

2.3 Handling Skew in GROUP BY

By definition an aggregation operation requires that all records with the same key be collected together. This makes the breaking of the "one key, one reducer" rule, which is done for `JOIN` and `ORDER BY` not feasible for `GROUP BY`. However, some aggregate operations can be distributed across multiple phases. Consider for example the `COUNT` function, which can count whatever records are present the first time it is invoked, and on subsequent invocations can sum results of previous invocations. As discussed in [18], a function that can be rewritten in this way is referred to as *algebraic*.

In Pig, a UDF can indicate that it is algebraic by giving Pig three separate implementations of itself, an *initial*, *intermediate*, and *final* implementation. Pig in turn runs these in the map, combine, and reduce phases. The contract is that the initial (map) implementation will be run once per record, the intermediate (combine) implementation will be run zero or more times per key, and the final (reduce) implementation will be run once per key. This contract is inherited directly from MapReduce's contract of how the map, combine, and reduce phases are run. As an example, consider the `COUNT` function. The initial implementation returns a 1. The intermediate and final implementations sum the counts provided to them.

Use of these algebraic aggregation functions greatly reduces the effect of skew. In general, however many records with a given key a mapper has in its input, the associated combiner will only produce one record for

that key.[4] This means that for each reducer the maximum number of records it will see for each key is bounded by the number of mappers in the job. Also, as part of the map and reduce phases, if the number of combiner outputs to be input to a given reducer exceeds a configurable threshold a merge phase will be invoked where the combiner is again run. This is done to avoid inefficient merges with too many input streams during the reduce. This further bounds the maximum number of records per key each reducer sees to MAX(number of mappers, merge threshold). Given that the merge threshold is usually set at one hundred or lower this effectively prevents significant skew for the reducers in cases where algebraic aggregation functions are used.

3 Aggregating on Multiple Key Sets in a Single MapReduce Job

One of Pig's most common uses is in data processing pipelines. Data is loaded into Hadoop and then processed by Pig to prepare it for analysis, use in front line serving systems, etc. Frequently in these use cases a few data sets will serve as a basis for many derived data sets. Consider the example of an online retailer. The logs from the web servers (click and view data) and the logs from the sales transactions will form the basis for much of the downstream processing. Because of this, many Pig users load the same data set multiple times and aggregate it by different keys. This spawns many MapReduce jobs, each of which load and de-serialize the same data set.

To address this common need the Pig community developed a way for Pig to perform aggregations on multiple keys in a single MapReduce job. This optimization is referred to as *multi-query*.[5] Consider the example of an online retailer that collects records of sales transactions from its front end servers that record, amongst other things, the user that visited its site, what he purchased, and how much the purchases cost. The retailer may want to segment users along different dimensions in order to analyze how different types of users are interacting with the site. For example, the retailer's analysts may want to analyze average user purchases by geographic location, by demographic group (e.g. age and gender), and by income level. A Pig Latin script to prepare this data would look like:

```
txns        = LOAD 'data/txns' AS
                (userid, timestamp, itempurchased, cost);
users       = LOAD 'data/users' AS
                (userid, zip, age, gender, income);
joined      = JOIN txns BY userid, users by userid;
bygeo       = GROUP joined BY zip;
avgbygeo    = FOREACH bygeo GENERATE
                group AS zip, AVG(joined.cost);
STORE avgbygeo INTO 'data/avgbygeo';
bydemo      = GROUP joined BY (age, gender);
avgbydemo   = FOREACH bydemo GENERATE
                FLATTEN(group) AS (age, gender), AVG(joined.cost);
STORE avgbydemo INTO 'data/avgbydemo';
byincome    = GROUP joined BY income;
avgbyincome = FOREACH byincome GENERATE
                group AS income, AVG(joined.cost);
```

[4] This is not quite true because a mapper writes data to the combiner's buffer until that buffer is full and then the combiner runs. Each of these combiner outputs becomes an input to the reducer. It is possible that the mapper has sufficient output to overflow the combiner's buffer, thus producing multiple inputs per reducer from that combiner. In a well tuned system the HDFS block size will be set such that the mapper can store the entire contents of the block in memory. Since approximately a 4x expansion in data size from disk to memory is usually observed, an HDFS block size of 256M would indicate 2G of memory per mapper. However overflows can still occur from operators such as joins that grow the memory size of the mapper's output.

[5] The initial implementation of this feature was discussed in [8]

```
STORE avgbyincome INTO 'data/avgbyincome';
```

Notice that each `GROUP BY` operator works on the same input, `joined`. At first glance it would be natural to assume that this Pig Latin script would be translated to four MapReduce jobs, one for the `JOIN` and one for each of the `GROUP BY`s. But Pig translates this to two MapReduce jobs, one for the `JOIN` and one for all three `GROUP BY`s. In this example all three aggregations invoke the same function on the same column, `AVG(joined.cost)`. This is merely for simplicity in the example. There is no need for this. Different functions can be invoked on different columns in some or all of the aggregations.

Pig accomplishes this combination of multiple aggregations into a single MapReduce job as follows. The first MapReduce job is done in the normal manner. In the map phase of the second MapReduce job each record is replicated three times. An additional column is added to the record and set to indicate which of the three aggregates the record belongs to. We will refer to this as the *pipeline indicator* column. In the example above the copy of the record for `bygeo` would have a pipeline indicator of 0, the copy for `bydemo` a 1, and the copy for `byincome` a 2. The reduce key for this job is set to be a tuple. What is placed in that key varies for each record though. In the record bound for `bydemo` (pipeline 0) the field `zip` is placed in the tuple along with the 0 pipeline indicator. The record for `bygeo` (pipeline 1) has `age`, `gender`, and the 1 pipeline indicator in the key tuple. And the record for `byincome` (pipeline 2) has `income` and 2 for the pipeline indicator in its key tuple. MapReduce's default hashing mechanism now works as usual to hash the records across all of the reducers and collect records for each aggregate in each pipeline together.

Example:

Row entering the map phase of the second MapReduce job:

```
{"cost": 19.49, "zip": "94303", "age": 50, "gender": "female",
 "income": 200000}
```

Three rows exiting the map phase:

```
Row 1:
Key: {"pipeline":0, "zip": "94303"}
Value: {"cost": 19.49}

Row 2:
Key: {"pipeline":1, "age": 50, "gender": "female"}
Value: {"cost": 19.49}

Row 3:
Key: {"pipeline":2, "income": 200000}
Value: {"cost": 19.49}
```

Other fields will have been projected out of the row entering the second MapReduce job by Pig's optimizer as it realizes it no longer needs `userid`, and it never needed `timestamp` or `itempurchased` for this script. Similarly, after duplicating the records for each pipeline Pig's optimizer projects out fields it no longer requires for each pipeline.

Each combiner and reducer instantiates separate pipelines, one for each of the aggregations. When it receives a record it examines the pipeline indicator and places the record in the appropriate pipeline. Since the pipeline indicator is a part of the key a combiner or reducer is guaranteed to see all keys for a given group for a given pipeline together. This implies that the combiner or reducer need not cache records for different pipelines simultaneously, thus limiting memory usage. The data is then processed in each pipeline as it normally would be. Special handling is required when the data is written out of the reducer, as MapReduce expects a single output,

an expectation that Pig is violating. Pig handles this by ignoring the MapReduce default output mechanism and writing the output via the required `OutputFormat` itself separately for each pipeline. This also makes it possible to combine aggregates that write out data using different file formats (for example one aggregate might be stored using the default text format, one using `SequenceFile` (a binary format), and one written to Apache HBase [14]).

The speed up provided by this optimization varies depending on how many MapReduce jobs are coalesced together and how much data is being read and aggregated. The testing done by developers and initial users indicated near linear speed ups when coalescing up to ten jobs. For example, in one particular use case, an 8x speed up was observed with ten aggregations.

There are limits to the usefulness of this optimization. One is collecting together aggregations that do and do not use the combiner.[6] Passing non-combined data through the combiner, as is required in this case, is prohibitively expensive because the data is being repeatedly read and written without being reduced in size. When Pig is provided a script that would be a candidate for multi-query optimization that includes aggregations that do and do not use the combiner, it coalesces all of the combiner eligible jobs together but leaves the non-combiner eligible jobs to be run separately.

A second limit is that this optimization trades off the overhead of running additional MapReduce jobs and the associated extra reads and de-serializations of the data against the cost of increasing the amount of data being sent through the shuffle phase. In our experience, as the number of simultaneous aggregations grouped together grows the efficiency of using multi-query over separate jobs lessens. The Pig community has not yet taken the step of implementing a cost estimator for this feature that determines when MapReduce jobs should and should not be coalesced. This means that the user must be aware of this limitation and write their Pig Latin scripts in such a way to avoid over coalescing of aggregations.

4 Sort versus Hash Aggregation in the Hadoop Ecosystem

Section 2.3 describes how Pig does partial aggregation in the map phase. In addition to handling skew problems in `GROUP BY`, it helps in improving performance by reducing the data that needs to be transferred between mappers and reducers.

The hadoop MapReduce framework sorts the mapper's output on the shuffle key before sending the results to the reducers, and as a result the natural way for supporting partial aggregation in the mapper has been to use sort-based aggregation through use of the combiner.

While the combiner makes it easy to optimize aggregation in MapReduce programs, there are significant costs associated with it. Hadoop uses a memory buffer of fixed size (configurable for the MapReduce job), to accumulate the mapper's output before sorting it and running the results through the combiner. It serializes the data while writing into this buffer[7] and de-serializes it for use by the combiner. These serialization and de-serialization costs are high. If there is not sufficient reduction in data size, this overhead cost can degrade performance. For example, in an extreme case where there is no data size reduction (when each input record to the mapper has a distinct key), a MapReduce query using the combiner takes 50% longer than one that does not use the combiner [4].

To avoid this cost, support for hash-based aggregation within the map task has been introduced in Pig [5] [6]. A Java `HashMap` object with the set of `GROUP BY` keys as the key is used to accumulate the output of the aggregate UDF's initial function call (see section 2.3). When the estimated size of the HashMap exceeds a configurable threshold, it calls the UDF's intermediate function on the accumulated values for each HashMap key to compute partial aggregates. The partial aggregates are accumulated in another `HashMap`. When further

[6] See section 2.3 above for a discussion of the combiner and how Pig uses it.

[7] This is done because it is difficult to precisely calculate the memory size of Java objects.

memory thresholds are reached, the accumulated values are further aggregated and Pig starts writing the results as output of the map task.

In our benchmarks, hash-based aggregation has been shown to reduce the runtime of the map phase (including combiner) in jobs containing a `GROUP BY` by up to 50% [5].

If there is not sufficient reduction in output, the cost of performing hash based aggregation can again degrade the performance. But thanks to this being an internal Pig operator (unlike combiner), Pig can automatically disable the aggregation if the output size reduction is below a configurable threshold. This is done at run time. After a number of records are hashed if a sufficient reduction in size is not seen the hash table is dumped as output and Hadoop's default sort based aggregation used.

5 Future Plans

In addition to the areas outlined above where work has been done or is ongoing, there are additional optimizations that the Pig community has discussed. These involve using statistics to make decisions about query optimization and taking advantage of changes to the underlying execution framework to provide a more efficient environment in which Pig can operate.

5.1 On the Fly Replanning

As indicated earlier, in section 1.2, the Pig community chose initially to focus on building various implementations of operators so that Pig users could choose the best operator for their situation. For example, Pig has five join operators from which users can choose when writing their Pig Latin scripts. While this allows users a large degree of control it also pushes a significant burden onto them, as they must do exhaustive testing to determine the best operator for their use case. It is also not adaptive in the face of change, since for any script run on a regular basis different inputs may perform better with different execution choices. Wherever Pig can make reasonable choices to select the correct operator it should do so, while still allowing the user to override that choice.

Traditionally this is done in databases by using statistics and a cost based optimizer (CBO). The CBO uses the statistics about data in the table to estimate the cost of choosing different implementations of operators. There are, however, difficulties with this approach in Pig. First, statistics are often not available for Hadoop's data. Data in a database is generally cleansed and analyzed as part of the loading process. Data in Hadoop is often raw and has not yet been analyzed and had statistics generated for it. This is particularly true of data that Pig is operating on since Pig is often used for the cleansing and transforming process. It makes sense to use statistics when available, but it does not make sense to assume their existence.

Additionally, the depth of operator trees generated by a Pig Latin script can greatly exceed those of an average SQL query. Consider a common data analytics query with a join, an aggregate, and a sort of the results:

```
SELECT userid, storeid, SUM(cost) as total_purchases
FROM users, stores, purchases
WHERE users.userid = purchases.userid
      AND stores.storeid = purchases.storeid
      AND purchases.userid IS NOT NULL
      AND purchases.storeid IS NOT NULL
GROUP BY userid, storeid
ORDER BY total_purchases descending;
```

Ignoring projections the maximum depth of the operator tree generated to execute this query should be six (one for the scans, one for `IS NOT NULL` filters, one for each join, one for the aggregation, and one for the

sort). This means that even if the initial statistics for the data accessed in this query have some errors (which they will) the errors for the estimations for the operators at the top of the tree will usually be tolerable. However, in a Pig Latin script it is not uncommon to have operator trees of much greater depth. ETL processes often operate on data through tens or even hundreds of transformations before producing the final result. With this many operators processing the data, no matter how accurate the initial statistics are, the errors for the estimates for operators near the end of the pipeline will become intolerable.

The lack of statistics for existing data and the limited usability of statistics in longer Pig Latin scripts implies that it makes sense for Pig to instead collect statistics at runtime and modify its execution plan accordingly. This would work as follows. If statistics are available for some or all of the input data of the script, those would be used to plan the initial phases of the execution of the script. If statistics are not available, a sampling pass could be done to quickly generate statistics or a conservative plan could be chosen which would work reasonably well regardless of the input data.[8] As Pig processes the data it could then, at predetermined points, generate simple statistics. These in turn could be used before the next MapReduce job is launched to re-optimize and re-plan the remaining execution of the script. For example, a Pig execution plan that contains four MapReduce jobs linked one after the other may be changed so that statistics are collected as part of generating output of the second MapReduce job. Pig would then re-optimize and re-plan the remaining two MapReduce jobs, including potentially consolidating the two jobs. This is similar to the approaches described in [17] and [16], but adapted for the case where there are no initial statistics and with re-optimization points chosen statically up front rather than dynamically.

This proposed "on the fly" replanning has areas that need investigation in order to determine the best design.

- What is the optimal frequency of replanning? There will be a cost to collecting statistics, so replanning at every MapReduce job will introduce a noticeable drag. The inflection point between the cost of collecting statistics and replanning and the excessive error introduced by the number of operators for which estimates are done will need to be understood.

- What is the correct trade off between the cost of collecting statistics and the benefit of better planning with those statistics? Estimating the number of records in a file and the cardinality of a few key fields in those records can be done efficiently. Estimating the distribution of values in a field (e.g. building a histogram) is more valuable from an optimization standpoint but more expensive. There is also value in knowing statistics such as maximum and minimum values of a field, average length of a string field, etc. Experimentation will be required to understand when the cost of collecting any particular statistic is worth the planning benefit.

- How accurate do the statistics need to be to provide good planning? The more that statistics collection can rely on fast sampling techniques the more efficient it can be. However, the more error is introduced in collection the less accurate the resulting plans will be and the more often the planner will need to re-collect and re-plan.

5.2 Taking Advantage of Hadoop 2.0

In Hadoop 1.0 the only job execution framework available is MapReduce. In Hadoop 2.0 the job execution framework has been re-factored and generalized [3]. MapReduce is now one execution framework available

[8] Which of these would be better is not clear. A sampling pass that generates rough statistics is most likely more expensive than choosing a conservative plan since it will involve adding an additional MapReduce job and an additional read of some of the data. But in an environment where metadata can be stored (e.g. if the user is using Apache HCatalog [2]), it has the advantage that Pig can store the generated statistics for future reads of the data. Since files are written once in Hadoop, these statistics will be valid for future reads without the risk of them being invalidated by intervening updates. This trade off may need to be presented to users so that users or administrators can make the correct choice for their situation.

as part of the system. Hadoop 2.0 has opened up the interfaces so that users can write their own execution frameworks. This is done by providing an *application master* that manages the users' jobs, much as the Job Tracker manages MapReduce jobs in Hadoop 1.0.

Pig was originally written to use MapReduce as its execution engine because that was the only option in Hadoop. However, there are features of MapReduce which are not optimal from Pig's viewpoint. The ability to write an application master designed directly for Pig's use case will allow Pig to more efficiently execute its scripts. For brevity we will consider only one significant improvement Pig could make to the MapReduce paradigm in its application master, removing the map phase from all but the first MapReduce job in a chain of jobs.

When constructing an execution plan Pig uses a series of MapReduce jobs. However, after the map phase of the first job, no additional map phases are required. That is, it would be more efficient to have the chain of phases be `map -> reduce -> reduce` rather than `map -> reduce -> map -> reduce`. This is because whatever processing is in the second map phase can always be done in the preceding reduce. The contract of the map function is that it operates on one record at a time and it arbitrarily partitions its input data. The reduce function can satisfy this contract by applying the mapper's functionality to each output of operators placed in the reducer. Removing this unnecessary map phase removes a read to and write from HDFS, thus lowering overall disk and network operations. This approach is referred to as *MRR* or *MR** to indicate the multiple reduce phases in a single job.

Additionally, removing the MapReduce job boundary removes a synchronization point. When a MapReduce job generates output it writes it to a temporary location and only moves it to the requested output location in an atomic move after all reducers are finished. This avoids the data being read while it is still being written. A side effect of this for a Pig Latin script with multiple MapReduce jobs is that the second job cannot start until the first job is completely finished. In contrast, inside a MapReduce job data is shuffled between a mapper and the reducers, and potentially merged, as soon as a mapper produces it. The reducer cannot start until all mappers have completed, but much of the intermediate work can be done. The same would be true in the MR* case, where data produced by the first set of reducers could be shuffled and merged in preparation for the second set of reducers.

6 Acknowledgements

We would like to thank the Apache Pig community, both developers and users, who have built, maintained, used, tested, written about, and continue to push forward Pig.

References

[1] `http://code.google.com/p/snappy/`

[2] `http://incubator.apache.org/hcatalog/`

[3] `http://hadoop.apache.org/docs/r2.0.2-alpha/`

[4] `https://cwiki.apache.org/confluence/display/PIG/Pig+Performance+Optimization#PigPerformanceOptimization-HashAggvs.Combiner`

[5] `https://issues.apache.org/jira/browse/PIG-2228`

[6] `https://issues.apache.org/jira/browse/PIG-2888`

[7] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: *Pig Latin: A Not-So-Foreign Language for Data Processing*. Proceedings of the 2008 ACM SIGMOD international conference on Management of data, June 09-12, 2008, Vancouver, Canada

[8] Alan F. Gates, *et al.*: *Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience*. Proceedings of the VLDB Endowment, v.2 n.2, August 2009

[9] Matthias Jarke , Jurgen Koch: *Query Optimization in Database Systems*. ACM Computing Surveys (CSUR), v.16 n.2, p.111-152, June 1984

[10] `http://pig.apache.org/`

[11] `http://hadoop.apache.org/`

[12] Alan Gates, *Programming Pig*, O'Reilly, 2011.

[13] `http://hadoop.apache.org/docs/r1.1.1/mapred_tutorial.html`

[14] `http://hbase.apache.org/`

[15] D. DeWitt, J. Naughton, D. Schneider, and S. S. Seshadri: *Practical Skew Handling in Parallel Joins*. In Proc. of the 18th VLDB Conf., pages 2740, 1992.

[16] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, Hamid Pirahesh: *Robust Query Processing through Progressive Optimization*. In Proc. of ACM SIGMOD 2004, pages 659-670.

[17] Navin Kabra and David J. DeWitt: *Efficient Mid-Query ReOptimization of Sub-Optimal Query Execution Plans*. In Proc. of ACM SIGMOD 1998, pages 106-117.

[18] Jim Gray, *et al.*: *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*. In *Data Mining and Knowledge Discovery* 1(1): 29-53 (1997).

[19] YongChul Kwon, Kai Ren, Magdalena Balazinska, Bill Howe, and Jerome Rolia: *Managing Skew in Hadoop*. IEEE Data Eng. Bull., Vol 36, No 1, March 2013

Recurring Job Optimization for Massively Distributed Query Processing

Nicolas Bruno
nicolasb@microsoft.com
Microsoft Corp.

Sapna Jain
sapnakjain@gmail.com
IIT Bombay

Jingren Zhou
jrzhou@microsoft.com
Microsoft Corp.

Abstract

Companies providing cloud-scale data services have increasing needs to store and analyze massive data sets. For cost and performance reasons, processing is typically done on large clusters of tens of thousands of commodity machines. Developers use high-level scripting languages that simplify understanding various system trade-offs, but introduce new challenges for query optimization. One key optimization challenge is missing accurate data statistics, typically due to massive data volumes and their distributed nature, complex computation logic, and frequent usage of user-defined functions. In this paper we describe a technique to optimize a class of jobs that are recurring *over time in a cloud-scale computation environment. By leveraging information gathered during previous executions we are able to obtain accurate statistics for new instances of recurring jobs, resulting in better execution plans. Experiments on a large-scale production system show that our techniques significantly improve cluster utilization.*

1 Introduction

An increasing number of applications require distributed data storage and processing infrastructure over large clusters of commodity hardware for critical business decisions. The MapReduce programming model [7] helps programmers write distributed applications on large clusters, but requires dealing with complex implementation details (e.g., reasoning with data distribution and overall system configuration). High level scripting languages were recently proposed to address these limitations, such as Nephele/PACT [1], Jaql [2], Hyracks [3], Tenzing [5], Dremel [15], Pig [16], Hive [18], DryadLINQ [19], and SCOPE [20, 21]. These languages offer a single machine programming abstraction and allow developers to focus on application logic, while providing systematic optimizations for the underlying distributed computation. As in traditional database systems, such optimization techniques rely on data statistics to choose the best execution plan in a cost-based manner [21, 13].

Consider the sample script shown in Figure 1(a), which first joins the result of user-defined aggregates and performs a global sort on the join result before writing the final answer. Figure 1(b) shows a possible execution plan for this script. The plan first partitions the input data sets on the group by columns (*a* and *c*, respectively) and computes user defined aggregates in parallel. It then re-partitions the aggregate outputs on the join column *sb* and evaluates the join. Finally, it range-partitions the join output on the global sort columns and locally sorts each partition.

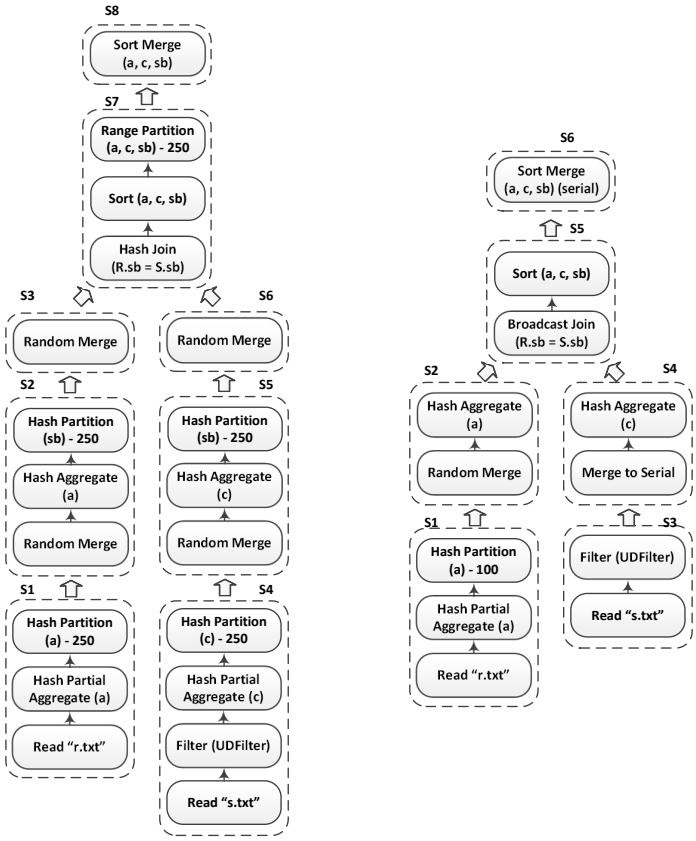
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
R = SELECT a, UDAgg(b) AS sb
    FROM "r.txt" USING RExtractor
    GROUP BY a

S = SELECT c, UDAgg(b) AS sb
    FROM "s.txt" USING SExtractor
    GROUP BY c
    HAVING UDFilter (b, c) > 5

SELECT *
FROM R JOIN S
ON R.sb = S.sb

OUTPUT to "t.txt" USING TOutputter
```



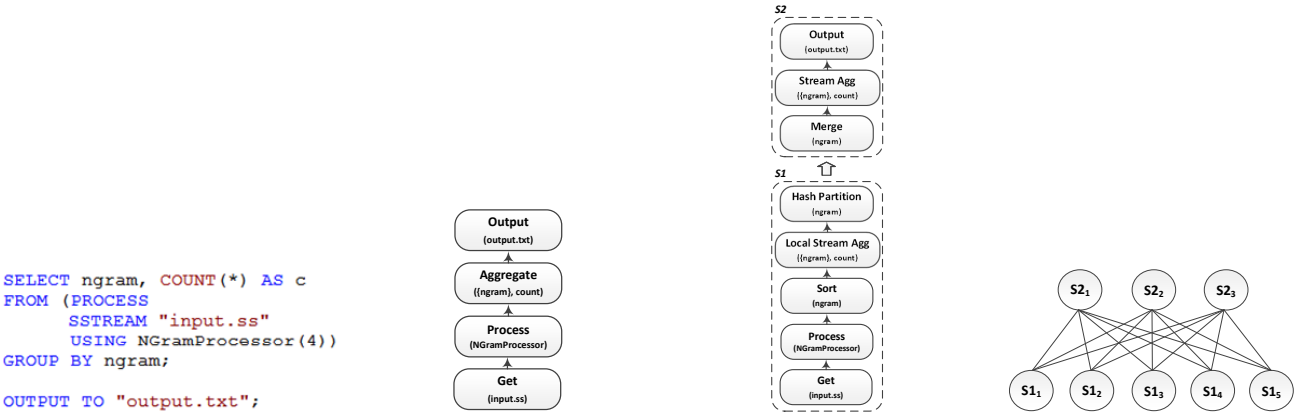
(a) Script (b) Original Plan (c) Alternative Plan

Figure 1: Alternatives to Execute a Distributed Query

Now, suppose that the user-defined filter `UDFilter(b,c) > 5` on `s.txt` is very selective and drastically reduces the data size. In this case, it would be better to merge all data on a single machine to compute the user-defined aggregate and use a broadcast join variant (which does not require re-partitioning on the join column) to obtain the join result, as shown in Figure 1(c). This alternative also assumes that the user-defined aggregate on *r.txt* and the join are selective, so it reduces the degree of parallelism from 250 to 100 and changes the post-join range-partitioning with a serial alternative. Clearly, this alternative plan is better than the previous one for the given cardinality assumptions, but it will perform really badly if those assumptions are not valid.

The previous examples illustrate that the choice of a plan heavily depends on properties of the input data and user-defined functions. Several aspects of highly distributed systems make this problem more challenging compared to traditional database systems. First, it is more difficult to obtain and maintain good quality statistics due to the distributed nature of the data, its scale, and common design choices that do not even account for accurate statistics over input unstructured data. Even if we can collect statistics for base tables, the nature of user scripts, which typically rely on user-defined code, makes the problem of statistical inference beyond selection and projection more difficult during optimization. Additionally, the cost of user defined code is another important source of information for cost-based query optimization. Such information is crucial for the optimizer to choose the optimal degree of parallelism for the final execution plan and when and where to execute the user code. Finally, input scripts typically consist of hundreds of operators, which magnify the well-known problem of propagation of statistical errors. This negatively impacts the quality of the resulting execution plans, with a potential performance impact in the orders of magnitude.

At the same time, a large proportion of queries in such a cloud-scale distributed environments are *parametric* and *recurring* over a time series of data. The input datasets usually come in regularly, say, hourly or daily, and



(a) SCOPE script (b) Optimizer Input tree (c) Optimizer Output tree (d) Scheduled graph

Figure 2: Compiling and Executing in SCOPE

more importantly, they share similar data distribution and characteristics. The same business logic is thus applied to incoming datasets that are similar. We denote those jobs as *recurring*. In this work we describe mechanisms to capture data statistics concurrently with job execution, and automatically exploit them for optimizing recurring jobs. We achieve this goal by instrumenting different job stages and piggybacking statistics collection with the normal execution of a job. After collecting such statistics, we show how to feed them back to the query optimizer so that future invocations of the same (or similar) jobs take advantage of accurate statistics.

The rest of the paper is structured as follows. In Section 2 we review necessary background on distributed computation engines, using the SCOPE system for illustration. In Section 3 we describe our solution for recurring job optimization. Section 4 reports an experimental evaluation of our approach on real-world data. Section 5 reviews related work and Section 6 concludes the paper.

2 Background

We now describe the main components of distributed computation engines, using the SCOPE system as an example. The ideas and results in this paper, however, are applicable to other engines as well.

Language and Data Model. The SCOPE language is declarative and intentionally reminiscing SQL. The select statement is retained along with joins variants, aggregation, and set operators. Like SQL, data is modeled as sets of rows composed of typed columns, and every rowset has a well-defined schema. At the same time, the language is highly extensible and is deeply integrated with the .NET framework. Users can easily define their own functions and implement their own versions of relational operators: *extractors* (parsing and constructing rows from a raw file), *processors* (row-wise processing), *reducers* (group-wise processing), *combiners* (combining rows from two inputs), and *outputters* (formatting and outputting final results). This flexibility allows users to solve problems that cannot be easily expressed in SQL, while at the same time enables sophisticated reasoning of scripts. Figure 2(a) shows a simple SCOPE script that counts the different 4-grams of a given single-column structured stream. In the figure, `NGramProcessor` is a `C#` user-defined operator that outputs, for each input row, all its n-grams (4 in the example). Conceptually, the intermediate output of the processor is a regular rowset that is processed by the main outer query (note that intermediate results are not necessarily materialized between operators at runtime).

Compilation and Optimization. A SCOPE script goes through a series of transformations before it is executed in the cluster. Initially, the SCOPE compiler parses the input script, unfolds views and macro directives, performs syntax and type checking, and resolves names. The result of this step is an annotated abstract

syntax tree, which is passed to the query optimizer. Figure 2(b) shows an input tree for the sample script. The SCOPE optimizer is a cost-based transformation engine that generates efficient execution plans for input trees. The optimizer returns an execution plan that specifies the steps that are required to efficiently execute the script. Figure 2(c) shows the output from the optimizer, which defines specific implementations for each operation (e.g., stream-based aggregation), data partitioning operations (e.g., the partition and merge operators), and additional implementation details (e.g., the initial sort after the processor, and the unfolding of the aggregate into a local/global pair). The backend compiler then generates code for each operator and combines a series of operators into an execution unit or *stage*, obtained by splitting the output tree into components that would be processed by a single node. The output of the compilation of a script thus consists of (i) a graph definition file that enumerates all stages and the data flow relationships among them, and (ii) the assembly itself, which contains the generated code. This package is sent to the cluster for execution. Figure 2(c) shows dotted lines for the two stages corresponding to the input script.

Job Scheduling and Runtime. The execution of a SCOPE script is coordinated by a Job Manager (or JM). The JM is responsible for constructing the job graph and scheduling work across available resources in the cluster. A SCOPE execution plan consists of a DAG of stages that can be scheduled and executed on different machines independent of each other. A stage represents multiple instances, or *vertices*, which operate over different partitions of the data (see Figure 2(d)). The JM maintains the job graph and keeps track of the state and history of each vertex in the graph. When all inputs of a vertex become ready, the JM considers the vertex *runnable* and places it in a scheduling queue. The actual vertex scheduling order is based on vertex priority and resource availability. During execution, a vertex reads inputs either locally or remotely. Operators within a vertex are processed in a pipelined fashion, in a way very similar to a single-node database engine. Every vertex is given enough memory and processing power to satisfy its requirements, which sometimes prevents a new vertex from being run immediately on a busy machine. Similarly to traditional database systems, each machine uses admission control techniques and queues outstanding vertices until the required resources are available. The final result of a vertex is written to local disks (non-replicated for performance reasons), waiting for the next vertex to *pull* data.

3 Recurring Job Optimization

The architecture of our approach to process recurring jobs is shown in Figure 3. We next describe our technique in detail by illustrating the steps a typical job follows through the system:

1. Initially, a script is submitted to the cluster for execution. The script might be recurring or new, and it is assumed to be annotated with parametric information (e.g., usually the input datasets change every day following a simple expression pattern).
2. The compiler parses the input script, performs syntax and type checking, and passes an annotated abstract syntax tree to the query optimizer. The query optimizer explores many semantically equivalent rewritings, estimates the cost of each alternative, and picks the most efficient one. While doing costing, the optimizer relies on cardinality estimates and other statistics for each plan alternative (see Section 3.4).
3. We extend the query optimizer to generate *signatures* for plan subtrees (explained in Section 3.1). During plan exploration we collect all the signatures that are associated with execution alternatives, and before implementation and costing we probe the *statistics repository* for signature matches. The repository is a new service that stores plan signatures and the corresponding runtime statistics gathered during execution of previous jobs. The optimizer relies on such feedback to produce a more effective execution plan. Signatures can be matched not only on the same recurring job, but also on similar jobs that share common subexpressions. During optimization, the optimizer instruments the resulting execution plan to collect additional statistics during execution: the resulting execution plan might contain sub-plans not yet seen by the statistics repository, and also data properties might change over time, invalidating previous estimates.

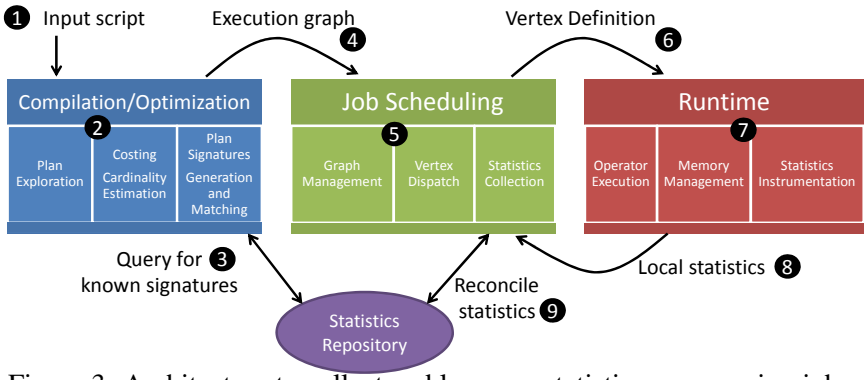


Figure 3: Architecture to collect and leverage statistics on recurring jobs.

4. The resulting execution plan is passed to the job scheduler in the form of a directed acyclic graph, where each vertex is an execution unit which looks similar to a single-node relational query plan, and each edge corresponds to data transfer due to repartitioning operators.
5. The scheduler manages the execution graph to achieve load-balancing, outlier detection and fault tolerance, among others.
6. The job manager transfers the vertex definition of new execution units to be run in cluster machines, and monitors the health and progress of each instance.
7. The runtime, which can be seen as a single-instance database engine, executes the vertex definition and concurrently collects statistics requested by the optimizer and instrumented during code generation (see Section 3.2).
8. When the vertex finishes execution, as part of the last heartbeat to the job scheduler, it sends back the aggregated statistical information, which is collected and further aggregated over all vertex instances.
9. Before finishing execution of the whole job graph, the job manager contacts the statistics repository and inserts the new statistics, to be consumed by future jobs. In case of duplicate signatures, the repository reconciles them using different policies (see Section 3.3). Periodically a background task discards statistics in the repository that exceed a certain age.

We next describe some components in additional detail.

3.1 Plan Signatures

Plan signatures uniquely identify a logical query fragment during optimization. This is similar to the notion of view matching technology in traditional database systems. View matching is a very flexible approach, but very difficult to extend beyond select-project-join queries with aggregation. Traditional view matching is able to perform partial matching on different queries and compensate differences using additional relational operators. For scalability purposes, and because many jobs are naturally recurring in our environment, we take a slightly different approach that can handle any operator tree including user-defined operators in SCOPE. This approach has some advantages over traditional view matching for our scenarios. First, we can handle any logical operator tree including used defined operators in SCOPE, which are not considered in traditional view matching. Second, signatures are very compact and easy to manipulate inside the optimizer and across components. Finally, querying and updating the statistics repository for signature matches is a very efficient lookup operation.

Specifically, for every operator subtree we traverse back the sequence of rules applied during optimization [9] until we reach the initial semantically equivalent expression[1]. Since the optimizer is deterministic, this initial

[1]Every rule transforms one plan into another, and we tag each resulting plan with the rule it produced it and its source. That way, we can always track the plan back to the original logical expression.

expression can be used as the canonical representation for the operator tree. We then recursively serialize the representation of the canonical expression and compute a 64-bit hash value for each subtree, which serves as the plan signature[2]. All fragments producing the same result are grouped together and have the same plan signature.

Parametric signatures. We relax the construction of signatures to accept parametric scripts. The reason is that recurring scripts are virtually identical, but differ on certain constants or input names (e.g., there are predicates that filter the current day). In absence of parametric signatures, every script would result in different signatures and there would not be any possibility of reusing statistical information. In our approach, customers are required to specify that certain constants (including input names) are parameters. In that case, the signature skips the actual value of the constant and replaces it with a canonical value, which allows the system to match subexpressions that differ only on parameters.

3.2 Statistics instrumentation

The compiler instruments generated code to capture statistics during execution and thus enable recurring job optimization. Statistics need to satisfy certain properties to be useful in our framework. First, statistics collection needs to have low overhead and consume little additional memory, as it is always enabled during normal execution. Second, statistics must be composable, because each vertex instance computes partial values that are then sent to and aggregated by the JM. Finally, statistics must be actionable, as there is no point in accurately gathering statistics unless the query optimizer is able to leverage them.

We model statistics as very lightweight user-defined aggregates whose execution is interleaved with normal vertex processing. This design allows to easily add new statistics into the framework as needed. Every operator in the runtime is augmented with a set of statistics to compute. On startup, the operator calls an `initialize` method on the statistics object. On producing each output row, the operator invokes an `increment` method on the statistics object passing the current row. Before closing, the operator invokes a `finalize` method on the statistics object, which computes statistics final results. Each statistics object also implements a `merge` method used by the JM to aggregate partial values across stages. Examples of implemented statistics include:

Cardinality and average row size. The object initializes cardinality and row size counters to zero, increments the cardinality counter and adds the size of the current row to the size counter for each `increment` function call, and returns the cardinality counter and the average row size upon finalization.

Inclusive user-defined operator cost. The object initializes a timer on creation, does nothing on each increment function call, and returns the elapsed time at finalization.

Some common statistics are optimized further by directly generating code that performs the initialization, increment, and finalization methods without requiring virtual function calls.

3.3 Statistics Management

The job manager maintains a *statistics package* data structure while the job is executing. A statistics package is a transient collection of plan signatures and aggregated statistics returned by completed vertices. Every time a vertex finishes execution, it sends the collected statistics to the job manager in the last heartbeat. The job manager uses the `merge` operation on statistics to aggregate all partial statistics for each signature.

The statistics repository is implemented as a key-value service, where keys are signatures and values are serializations of the corresponding statistics. Every time a job finishes, the job manager updates the statistics

[2]Modulo hash collisions, the signatures of two plan fragments match if and only if they produce semantically equivalent results. In practice, hash collisions are so rare that we can hash values as signatures, but other mechanisms are possible (e.g., using the serialization of the logical tree as the signature itself).

repository with all the statistics it gathered during execution. For that purpose, it creates a new entry for each signature (or merges statistics if the signature already exists in the repository). The specific merging behavior is dictated by a policy. Simple alternatives include keeping the last instance, or performing a weighted average with aging. Additionally, the job manager stores all statistics for the job under the signature that corresponds to the root of the corresponding script, to enable the *single-job* optimization mode discussed in Section 3.5.

3.4 Optimizing Recurring Jobs

SCOPE uses a transformation-based optimizer based on the Cascades framework [9], which translates input scripts into efficient execution plans. Transformation-based optimization can be viewed as divided into two phases, namely, logical exploration and physical optimization. Logical exploration applies transformation rules that generate new semantically equivalent logical expressions. All equivalent logical expressions are grouped together in a data structured called *memo*. Examples of such rules include *pushing down* selection predicates, and transforming associative reducers into a local/global pair. Implementation rules, in turn, convert logical operators to physical operators, cost each alternative, and return the most efficient one. Examples include using sort- or hash-based algorithms to implement an aggregate query, or deciding whether to perform a pairwise or broadcast join variant. To enable recurring job optimization, we extended the optimizer to compute signatures of each expression it processes, which is done with very little overhead. Additionally, the optimizer leverages statistical information stored in the global repository. For that purpose, we modify the entry point to any derivation of statistical properties in the optimizer to perform a lookup on the statistics package. For instance, deriving the cardinality of an expression starts by checking whether the signature of the expression matches an instance in the statistics repository. If so, cardinality estimation is bypassed and replaced with the value found in the repository. Since all equivalent logical expressions have the same signature, this technique produces consistent results.

3.5 Optimization modes

There are two different ways to leverage statistics during optimization of recurring jobs, with different trade-offs in both performance and accuracy. The first approach, denoted *single-job mode*, performs a single call to the statistics repository with the signature corresponding to the root of the whole script, and obtains all signatures associated to the given script in previous executions. The second approach, denoted *any-job* model, performs one call to the statistics repository for each signature is about to process. The second approach is clearly more expensive as it requires a single service call per expression, but can be used to optimize queries that share script fragments with other (unrelated) previously executed jobs.

4 Experimental Evaluation

In this section, we report an experimental evaluation of an implementation of our approach on SCOPE using real queries on a large production cluster consisting of tens of thousands of machines at Microsoft. Each machine in the cluster has two six-core AMD Opteron processors running at $1.8GHz$, $24GB$ of DRAM, and four $1TB$ SATA disks. All machines run Windows Server 2008 R2 Enterprise X64 Edition. The cluster runs tens of thousands of jobs daily, reading and writing petabytes of data, powering different online services. Due to confidential data/business information, we report performance trends rather than actual numbers for all the experiments. We present a case study on a single query in Section 4.1 and show how recurring job optimization improves subsequent executions. In Section 4.2 we summarize a performance evaluation on real-world queries.

4.1 Case Study

Consider the simple query in Figure 1(a). Without knowing the selectivity of the user-defined predicate, aggregates and join operators, the optimizer generates the plan shown in Figure 1(b), which consists of eight stages and five complete data repartition stages. Stage $S1$ reads `r.txt`, computes a local aggregate, and hash-partitions the resulting data on column (a) into 250 partitions. Stage $S4$ reads *s.txt*, applies the user-defined predicate, computes a local aggregate, and hash-partitions the data on column (c) into 250 partitions. Stages $S2$ and $S5$ aggregate the tuples that belong to the same partition across all vertices, compute the user-defined aggregate and hash-partition the data on column (sb) into 250 partitions. Stages $S4$ and $S6$ aggregate the tuples that belong to same partition across all vertices. Stage $S7$ performs a pairwise join, locally sorts the join output on columns (a, c, sb) and range-partitions the data into 250 partitions. Finally, stage $S8$ aggregates the tuples that belong to the same partition across all vertices maintaining the sort order. During execution, the runtime collects different data statistics in all operators as described in Section 3.

When the recurring job is submitted the next day, the optimizer knows the selectivity of all operators and other data statistics and chooses the plan shown in Figure 1(c), which consists of only six stages with reduced degree of parallelism, only one complete data repartitioning and two partial data shuffles (moving data to a single machine). Stage $S1$ reads *r.txt*, computes a local aggregate, and hash-partitions the data on column (a) into 100 partitions (instead of 250). Stage $S2$ aggregates the tuples that belong to same partition across all vertices and computes the user-defined aggregate. Stage $S3$ reads *s.txt*, applies the user-defined predicate and computes a local aggregate. Now, instead of partitioning data on the group by column to compute a parallel aggregate, the optimizer leverages the knowledge that data volume is small, and stage $S4$ aggregates all the data on a single machine computing the user-defined aggregate serially. Another data partitioning on the join columns is avoided by using a broadcast join variant in stage $S5$, followed by a local sort of the join output on columns (a, c, sb). Since the join is very selective and significantly reduces the data volume, the repartitioning operation required by the global sort is replaced by a merge operation in stage $S8$.

By gathering and exploiting statistics, the resource consumption of the job is reduced $6.5x$ and latency is reduced $3.2x$. In this example, a significant fraction of the improvement comes from structural changes in the plan (e.g., introducing broadcast join variants), and a better determination of the degree of parallelism of repartitioning operators, which is overestimated due to complex filter predicates and user defined operators.

4.2 Performance Evaluation

In this section we report results of evaluating our approach on a real workload. We observed that roughly 40% of all the jobs submitted to our production clusters in a month have a recurring patterns and potentially benefit from recurring job optimization. We now summarize our findings on a representative workload that consists of 6 queries used internally to administer the cluster [3], and 7 queries from different customers (which include advertising and revenue, click information, and statistics about user searches). The queries in the workload have large variability in resource usage, with latencies that range from minutes to several hours, and process from gigabytes to many terabytes. We computed the resource consumption as total machine-hours used by each query with and without recurring job optimization. Figure 4 summarizes our results, and we can see that resource consumption is reduced from 10% to 80% by using recurring job optimization. Most of these improvements come from better structural plans and changes in the required degree of parallelism, as illustrated in the previous section. At the same time, we observed negligible overhead for collecting runtime statistics during query execution.

[3] All operational data generated by the cluster is stored in the cluster itself as tables, so analysis and mining of interesting cluster properties can be done in SCOPE itself.

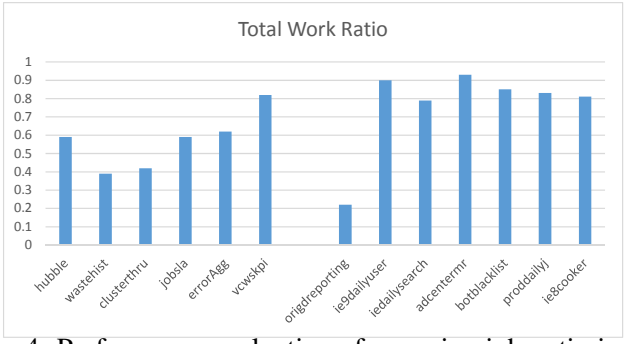


Figure 4: Performance evaluation of recurring job optimization.

5 Related work

The problem of accurate statistics estimation is not unique to cloud-scale data processing systems, as traditional query optimizers also have similar issues. To deal with parameter markers in queries, early work [6, 10] proposes generating multiple plans at compile time. When unknown quantities are discovered at runtime, a decision tree mechanism decides which plan to execute. This approach suffers from a combinatorial explosion in the number of plans that need to be generated and stored at compile time. Progressive query OPtimization [14] uses the optimizer to generate a new plan when cardinality inconsistencies are detected. This work detects cardinality estimation errors in mid-execution by comparing the estimated cardinality values against the actual runtime counts. If the actual count is outside a pre-determined validity range for that part of the plan, a re-optimization of the current plan is triggered. This approach has high overhead of plan switching and wasted work, which need to be paid in every run of the job.

More closely related to our work is the DB2 LEarning Optimizer (LEO) [17], which waits until a plan has finished executing to compare actual row counts to the optimizer's estimates. It learns from mis-estimates by adjusting the statistics to improve the quality of optimizations in future queries. LEO does not capture and detect job recurrence over time series of data. It tries to compute adjustments to base table statistics and selectivity of predicates from output data sizes of operators, which can still be inaccurate and prone to oscillations. Our work can be seen as adapting techniques that exploit statistics on views during optimization [4, 8] to a distributed environment with explicit recurring jobs. The main contributions of our work are the generic signatures used to match plan fragments, the lightweight generated collectors for statistics, the automatic determination of which statistics to collect and store, and the policies around the statistics repository.

Some recent work [11, 12] extends self-tuning techniques to distributed environments. Without user intervention, these approaches can (i) fine tune system parameters based on workload utilization, and (ii) perform some optimizations across jobs by improving interactions between the scheduler and the underlying file system. In contrast to our work, these approaches cannot make structural decisions on how to optimize and execute individual queries, as they are not integrated with a query optimizer. Our techniques complement such approaches by additionally changing the structure of the execution plans (such as switching to broadcast join variants), and further improving cluster utilization.

6 Conclusion

Massive data analysis in cloud-scale data centers plays a crucial role in making critical business decisions and improving quality of service. High-level scripting languages free users from understanding various system trade-offs and complexities, support a transparent abstraction of the underlying system, and provide the system great opportunities and challenges for query optimization. One key optimization challenge, which is amplified in cloud-based systems, is missing accurate data statistics. In this paper, we propose a solution for recurring job

optimization for cloud-scale systems. The approach monitors query execution, collects *actual* runtime statistics, and adapts future execution plans as the recurring job is subsequently submitted again. Experiments on a large-scale production system show that the proposed techniques systematically address the challenges of missing/inaccurate data statistics and improve overall resource consumption.

References

[1] D. Battré et al. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the ACM symposium on Cloud computing*, 2010.

[2] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. In *Proceedings of VLDB Conference*, 2011.

[3] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of ICDE Conference*, 2011.

[4] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings SIGMOD*, 2002.

[5] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL implementation on the mapreduce framework. In *Proceedings of VLDB Conference*, 2011.

[6] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proceedings of SIGMOD Conference*, 1994.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI Conference*, 2004.

[8] C. Galindo-Legaria, M. Joshi, F. Waas, and M.-C. Wu. Statistics on views. In *Proceedings of VLDB*, 2003.

[9] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.

[10] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of SIGMOD Conference*, 1989.

[11] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of VLDB Conference*, 2011.

[12] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of CIDR*, 2011.

[13] H. Lim, H. Herodotou, and S. Babu. Stubby: A transformation-based optimizer for mapreduce workflows. In *Proceedings of VLDB Conference*, 2012.

[14] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proceedings of SIGMOD Conference*, 2004.

[15] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of webscale datasets. In *Proceedings of VLDB Conference*, 2010.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of SIGMOD Conference*, 2008.

[17] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of VLDB Conference*, 2001.

[18] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive – a petabyte scale data warehouse using Hadoop. In *Proceedings of ICDE Conference*, 2010.

[19] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of OSDI Conference*, 2008.

[20] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet mapreduce. *The VLDB Journal*, 21(5), 2012.

[21] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of ICDE Conference*, 2010.

A Common Compiler Framework for Big Data Languages: Motivation, Opportunities, and Benefits

Vinayak R. Borkar Michael J. Carey
University of California, Irvine
{vborkar,mjcarey}@ics.uci.edu

1 Introduction

We are in the era of Big Data and cluster computing. Data sizes have been growing at an exponential rate. At the same time, growth in computing power has been stagnating due to physical limits in processor technology. The only cost effective way to keep up with the growing data trend has been to harness multiple commodity computers in a shared-nothing configuration. Google, needing to manage extremely large amounts of web data, developed the MapReduce [16] platform. The MapReduce system provides a simple way for developers to express a data-oriented computation, by implementing two single-threaded functions (**map** and **reduce**), that is then automatically parallelized to run on large clusters of commodity machines. Yahoo! soon created the Hadoop [5] platform, based on the MapReduce specification, and made it available as open source software. Hadoop has since become the de facto MapReduce implementation outside of Google.

Initially MapReduce (Hadoop) greatly empowered engineers to run parallel jobs on large clusters to crunch virtually unlimited amounts of data while writing what appeared to be simple functions. However, over time many developers found themselves writing similar, yet different, functions to implement new jobs. Having to write MapReduce functions in an imperative language like Java proved to be time consuming and the proliferation of functions created a maintenance problem, prompting developers to explore the possibility of creating declarative high-level languages to express computation. Sawzall [30] was created inside Google for processing large corpora of text in parallel using MapReduce. Yahoo! developed the Pig [29] system along with its Pig Latin [27] language to express data processing in a declarative language resembling the relational algebra [15]. Facebook created Hive [2], an implementation of a SQL-like language. IBM developed the Jaql [23] language for processing large amounts of JSON data. Pig, Hive, and Jaql all compile queries in their respective languages into MapReduce jobs to run on the Hadoop platform. Microsoft proposed SCOPE [13], a system to compile a sequence of SQL-like statements to run in parallel on their own Dryad [21] data-parallel platform. The Dremel [25] system was created by Google for expressing analytical queries interactively in a subset of SQL using a custom column-based runtime platform. At the University of California, Irvine, we developed the AQL language for processing large amounts of semi-structured data, as part of the ASTERIX platform [9, 12, 7]. VXQuery [6] is a project under incubation at the Apache Software Foundation that aims to run XQuery [4] queries over large corpora of XML documents using a cluster of shared-nothing computers.

Declarative languages targeting various data-parallel platforms have seen dramatic growth in popularity in the past few years. In 2010 Facebook told us that upwards of 95% of their Hadoop jobs were automatically

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

generated by the Hive platform as a result of compiling HiveQL queries [32]. In around the same time frame Yahoo! was seeing more than 65% of their Hadoop jobs being submitted through their Pig platform.

The current Big Data movement has unleashed a wave of efforts where researchers and practitioners are creating new languages that better suit various target audiences without having to conform to established data processing standards like SQL. The need to process non-relational data in the Big Data space has also contributed to the invention of new languages. We conjecture that we will continue to witness the creation of new data processing languages that target specific domains, just as we have seen a growth of domain-specific languages in the Scala [26] world. This presents an important opportunity, as declarative languages for data processing share a common set of requirements and properties which suggest a unified platform upon which future languages can be built. Here we analyze four data processing languages (SQL, XQuery, HiveQL, and Pig Latin) with an aim to compare and contrast language features in Section 2. In Section 3 we take a deeper look at the internals of two open source systems that are popular for Big Data processing, Pig and Hive, to understand the commonalities and differences of the two systems. Based on our observations, Section 4 enumerates requirements for a unified platform and describes how one such platform, one that we are building (called Algebricks), is shaping up. Finally, we conclude in Section 5.

2 Languages and Models for Data Processing

In this section, we look at four known declarative languages that have been used successfully for large-scale data processing tasks, namely SQL, XQuery, HiveQL, and Pig Latin.

2.1 SQL: Structured Query Language

SQL emerged long ago as the de facto winner for processing relational data stored in relational database management systems. The relational model [15] that forms the basis of the logical model for data representation in RDBMSs is based on set-theory, and the SQL language is based on first-order logic. Data stored in relational databases is logically modeled as a collection of tables (relations) containing records (tuples) comprised of fields (attributes). Every stored table has a well-defined schema that describes the names and data types of the fields that are contained in the records contained in that table. Every record in a table must have all the fields mentioned in the schema (although fields are allowed to have unknown or missing values in the form of NULLs). Another characteristic of the relational model is that fields of a record in a table must be of a scalar type; a field value cannot itself be a table. In other words, the relational model allows for a strict two-level hierarchical structure, where the outer level is a table and the inner level contains scalar fields. It is important to note that while the logical model in RDBMSs is based on flat tables, an actual implementation is free to store the data using any physical format as long as the logical information is captured completely. For example, relational databases frequently store tables in the form of B+Trees using the unique value in a record (its primary key) as the indexing key. Columnar storage [31] is another example of a different physical storage format, one where tabular information is stored a column at a time. In a parallel database, where data is stored by partitioning the data, more choices with respect to partitioning strategies are available to the implementation.

The SQL language is based on relational algebra, which in turn is based on first-order logic. A SQL query describes what is expected in the result and does not specify how the result is to be computed, making the language "declarative". In contrast, most programming languages such as C, C++, or Java tend to be "imperative"; the programs expressed in those languages actually describe the exact steps to be performed. Declarative languages leave the language implementation free to explore a range of choices with regards to how to actually compute the final result. An implementation can choose the best way to evaluate a query based on the physical storage strategy used to store the tables participating in the query. This property of SQL (and declarative languages, more generally) has made query optimization a flourishing area of research for the last few decades.

2.2 XQuery

Although SQL and relational databases were widely successful for storing and managing enterprise data, the rigid nature of most implementations with respect to schema management made way for XML as a way of representing data that was more semi-structured. As XML started to become more popular for managing semi-structured data, multiple languages were proposed. XML-QL [17] and Quilt [14] were initially proposed as possible languages for querying XML data. Eventually XQuery [4] was developed, including concepts from XML-QL and Quilt, and it was standardized by the W3C for querying XML data.

An abstract data model called the XQuery Data Model (XDM) [19] is the logical model used to describe the semantics of the XQuery language. XDM is more general and subsumes XML for representing data. XQuery allows data to be typed using the XML Schema standard [10], but does not require it, making XQuery well suited for data whose types are not known upfront. The fundamental concept of representing data for processing with XQuery is the "sequence of items". Every query expression in XQuery operates on a sequence of items to produce a sequence of items. The order of items within a sequence is important and the semantics of XQuery specify the exact order of items that every expression must produce.

The FLWOR expression, along with Path Expressions, forms the core of the XQuery language. The FLWOR construct (For-Let-Where-Orderby-Return) is similar to the Select block (Select-From-Where-Orderby) found in SQL. Path Expressions in XQuery can be translated into FLWOR expressions using the Path Step Expression as a primitive building block, as shown in [18]. Researchers have proposed various algebras based on relational algebra and extensions to express XQuery semantics [22, 28], showing that slight extensions to relational algebra are sufficient to capture the meaning of XQuery queries and to provide a basis for optimizing those queries.

2.3 HiveQL

Facebook created the Hive system to be able to process large amounts of data using the Hadoop platform. Hive was created with the express goal of making large-data processing accessible to data analysts who were already familiar with SQL. HiveQL has thus been developed to be a subset of SQL with some extensions to the supported data model and query syntax.

On the data model front, Hive provides support for tables containing standard primitive data types like in SQL. In addition, Hive adds support for nested data through Array, Map, and Struct data types. Fields in tables can be declared to be of these types and can be used to nest data to arbitrary levels.

HiveQL supports most of the standard SQL syntax. HiveQL limits joins to only equality joins. Non-equality joins or cross products are not directly supported by the syntax. Although non-equality joins can be simulated through the use of a "1 = 1" join predicate and a Where clause to capture the non-equality predicate, such queries are not optimized by the HiveQL optimizer. In order to access nested data in the form of Arrays, Structs, and Map types, HiveQL provides functions to navigate values of these types. In addition to scalar values, HiveQL also includes tuple-functions that can be used to explode nested data into tuples for performing standard relational processing. The current release of HiveQL does not allow the construction of a Map-valued field by grouping multiple records. However, third-party packages such as [3] can be used to add this functionality to Hive.

2.4 Pig Latin

PigLatin, the language exposed by the Pig system from Yahoo!, allows users to express queries in the form of steps. Each step in PigLatin converts its input data (a collection of records) into an output collection of records. At each step, the user can use Load, Filter, Group, Cogroup, ForEach, Join, Order, Distinct, Cross, Union, or Split operators to transform data. Most of the resulting steps (except Cogroup and Split) can be directly mapped to relational algebra operators. The Cogroup operation can be expressed as a Group operation followed by a Join operation and hence can be expressed as a combination of relational algebra operators. The Split operation

in PigLatin can be transformed into multiple Filter (Select) operations when translating PigLatin into relational algebra operators.

Records in Pig can contain scalar valued fields or nested data in the form of tuples, bags and maps.

2.5 Wrap-up

Table 4 summarizes the similarities and differences between these four query processing languages.

Feature	SQL	XQuery	HiveQL	PigLatin
Declarative	Yes	Yes	Yes	Yes
Collection data model	Bag	Sequence	Bag	Bag
Instance data model	Tuples of scalar values	XQuery data model item	Tuples of scalar and complex values	Tuples of scalar and complex values
Support for nested data	No	Yes	Yes (Limited)	Yes (Limited)

Table 4: Query language similarities and differences

3 Data Processing Systems: Pig and Hive

In this section we look at two popular Big Data query processing systems that were built independently, Pig and Hive, with an eye for their system architectures and the steps followed in accepting user queries and translating them into executable artifacts to produce results.

3.1 Apache Pig

The Apache Pig platform first parses the textual representation of a user query into a logical plan representing the query. The logical plan is comprised of a DAG structure where nodes represent operators such as Load, Filter, Group, etc. The resulting logical plan is transformed using rules to create an equivalent, more efficient plan. Currently, the rules implemented in the Pig compiler perform the following rewritings (rules meant to purely aid in the rewriting process by normalizing plans are not listed):

1. Push Filters to eliminate records early
2. Push "flatten" style operators that increase tuple cardinality to later in the plan
3. Merge multiple chained ForEach operators when possible
4. Optimize placement of Limit operators to reduce the number of tuples early

After the optimization process, Pig translates the final logical plan into a sequence of one or more MapReduce jobs to be executed using the Hadoop MapReduce platform. More details regarding Pig's optimizer can be found in [20].

3.2 Apache Hive

The architecture of Apache Hive looks very similar to that of Pig. Hive first parses queries in HiveQL into a logical parse tree representation. A Semantic Analyzer consults metadata information in the system catalog to check the existence of the tables named in the query. The semantic analyzer also makes sure that the query is well-formed with respect to the fields of tables accessed in the query and flags any operations that are incorrect with respect to data types. The parse tree is then translated into a logical plan by the Logical Plan Generator. The logical plan is a tree of operators where some of the operators are relational algebra operators while others are specific to Hive to ease the construction of MapReduce jobs. The logical plan is optimized using a rule-based optimizer (just as in Pig). The Hive rules perform logically similar tasks as those performed by the Pig optimizer,

but they have been implemented to process plans represented using the logical operators implemented in Hive. Hive has additional optimizations as compared to Pig, including:

1. Convert a series of joins into a multi-way join
2. Perform map-side partial aggregation for aggregate queries
3. Preserve and exploit interesting orders in subplans when possible
4. Use sampling to better plan queries

After the optimization process, as in Pig, the logical plan is converted into a series of MapReduce jobs to be evaluated on Hadoop [2, 33].

4 An Approach to Unification: Algebricks

Observing the amount of overlap in functionality in Pig and Hive (and query processors for XQuery, SQL, and AQL that the authors have implemented before) and the similarities in algebras used to express their semantics, we have embarked on creating a unified framework to help future implementors of parallel query processors to quickly build new language implementations to be evaluated over Big Data using a data-parallel platform. AQUA [24] was a similar effort undertaken back in the object database era to create a framework where bulk types (collections) and their operations were separated from OO data model details so as to develop one algebra capable of expressing the query semantics for a variety of OO languages.

Our platform, Algebricks [1], is a model-agnostic, algebraic layer for parallel query processing and optimization.

The Algebricks toolkit consists of the following parts:

1. A set of logical operators: Algebricks includes about fifteen logical operators that language implementors can use to construct query plans. In addition to the standard relational operators (Select, Project, etc), Algebricks contains operators for representing nested query plans.
2. A set of physical operators: While most logical operators have one corresponding physical operator, some operators provide alternate physical choices. In Algebricks, the Select operator has a single physical operator while the Join operator provides multiple choices such as Nested-Loop Join, Hybrid-Hash Join, Grace-Hash Join etc.
3. A rewrite rule framework: Algebricks currently includes a rule-based query transformation framework that provides interfaces for users to implement rewrite rules and a rule engine to execute rules to transform queries.
4. A set of generally applicable rewrite rules: Algebricks includes query transformation rules that usually show promise in most scenarios. For example, pushing Select operators to filter data early in a query is usually considered to help with query efficiency. Algebricks includes a rewrite rule to push Select operators lower in the query plans. Another example of a generally applicable rewrite rule for parallel query processing is one that converts an aggregation query into a two-step aggregation query (when the aggregate function lends itself to partial computation). Since aggregation functions are language dependent and implemented by the language author, their properties (such as the ability to be split into two-level aggregation functions) have to be indicated to the Algebricks framework for the parallel aggregation optimization rule to trigger.
5. A metadata provider API that exposes metadata (catalog) information to Algebricks: The metadata API serves as the window through which Algebricks looks at the host language's ecosystem. It is through the metadata API that Algebricks is able to learn about physical data properties of sources (for example), that is then used by the compiler to reason about interesting orders and interesting partitions during query optimization.
6. A mapping of physical operators to the runtime operators in Hyracks [11] (a data-parallel platform built from the ground up at UCI): Currently, Algebricks provides a translation of queries to a single runtime

layer, viz. Hyracks. While this is an artifact of the current implementation of Algebricks, in principle, it should be possible to extend Algebricks to generate runtime plans for other data-parallel platforms.

4.1 Data Model Agnostic Bulk Operators

An important principle that underlies the design of Algebricks, as mentioned earlier, is a data model agnostic design of bulk operators. To be useful for implementing arbitrary parallel data-intensive languages, Algebricks takes a two-level approach to represent operations on data. "Bulk Operators" are used to manipulate collections (sets, bags, lists, etc.) of tuples. Tuples serve as abstract containers that hold scalar data values whose types are defined by the language being implemented using Algebricks. While Algebricks provides bulk operators, language specific scalar data operations have to be implemented by the author of the language. Bulk operators in Algebricks are parametrized with "accessor" functions that expose to them data model properties necessary to perform their task. For example, the Select operator used to restrict the items in a collection based on a boolean predicate only needs a "function" that, when applied to an item, indicates if the item is to be preserved in the result or is to be discarded. By encapsulating data model specific operations in the function, the Select Operator itself remains agnostic to the specifics of data types needed to support a high-level language using Algebricks. Although this principle has been adopted by several data-parallel execution engines [5, 8, 11] to build extensible runtime systems, we use this principle in Algebricks to implement a data model agnostic compiler framework. Every piece of functionality usually present in a parallel query compiler, such as the query transformation framework used to manipulate query plans, the data property computation logic used to reason about interesting orders and interesting partitioning properties of data, and finally the logic necessary to generate runtime artifacts, is implemented in Algebricks following the same two-level approach described earlier. Query transformation rules that depend entirely on the semantics of bulk operators are made available out-of-the-box with Algebricks and can be re-used by every language implemented using the compiler framework. Similarly, runtime plan generation (a process usually referred to as "code generation") of bulk operations is provided by Algebricks requiring the author of the higher-level language to only provide code generation logic for data model specific functions.

4.2 The Anatomy of an Algebricks-based Compiler

Figure 1 shows the typical sequence of steps followed by a query compiler built using Algebricks. An incoming query string is first lexically analyzed and parsed to construct an abstract syntax tree (AST). The resulting AST is then checked for semantic and type errors before transforming it into an Algebricks logical plan. The Algebricks logical plan is composed of logical operators (described next) and serves as an intermediate language for query compilation. The initial logical plan is handed to the logical optimizer to be transformed into an equivalent, but more efficient, logical plan. The resulting optimized logical plan is then translated to an Algebricks physical plan by selecting physical operators for every logical operation in the plan; this is done by the physical optimizer. Both optimizers in Algebricks are rule-based and are configured by selecting the set of rules to execute during the optimization phases. Finally, the resulting physical plan is processed by the Hyracks job generator to produce a Hyracks job that is then parallelized and evaluated by the Hyracks runtime platform.

The query parser and the translator (the first two stages in Figure 1) are query language specific and have to be implemented by the developer of a new language's compiler. The next three stages (the two optimizers and the Hyracks job generator) are provided by the Algebricks library to be used by developers. Algebricks also includes a library of language independent rewrite rules that can be reused by the compiler developer. Additional language-specific rules are usually required in the optimization process and can be implemented by the developer by extending well-defined interfaces in the Algebricks library.

As shown in Figure 1, the various phases of the query compilation process need access to information about the environment in which the query is being compiled (usually described as catalog metadata). The

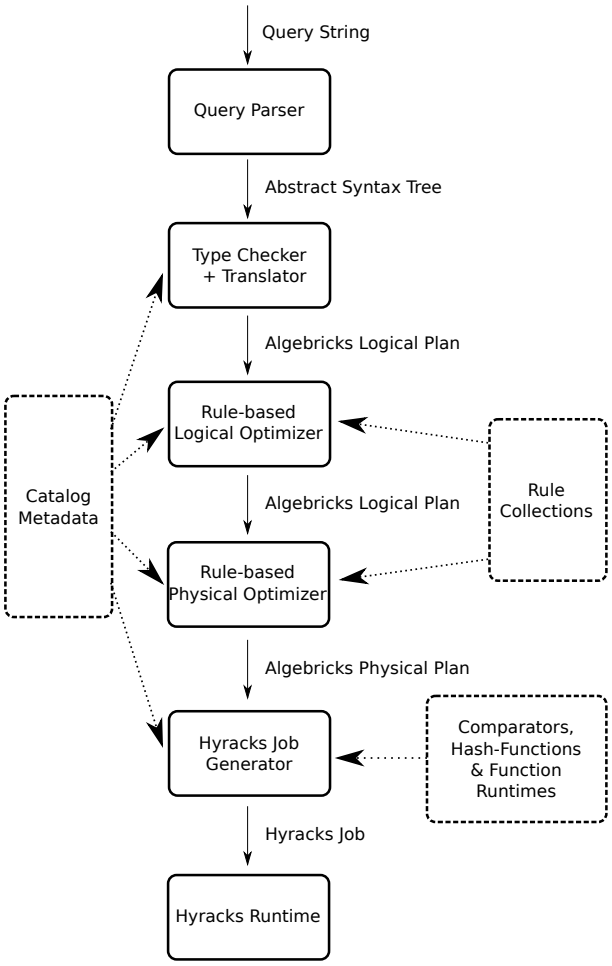


Figure 1: Flowchart of a typical Algebricks-based compiler

catalog contains metadata about the data sources that are accessible in a query. The catalog serves as the authoritative source of information about the logical properties of sources (such as schemas, integrity constraints, key information, etc.) and about their physical properties (access methods, physical locations, etc.). Algebricks provides a metadata interface that must be implemented by the compiler developer so that the various parts of the Algebricks compiler can access the relevant metadata information.

The Hyracks job generator maps the physical operators selected by the physical optimizer into Hyracks runtime operators. In the process of this translation, the Hyracks operators need to be injected with certain data model specific operations. The exact nature of the operations depend on the Hyracks operator being used. For example, the Sort operator in Hyracks must be provided with a comparator to compare two data model instances so that the input records can be sorted. Similarly, the Hash-Join operator needs a hash-function and a comparator to compute its result. When using Algebricks, the compiler developer must provide a family of operations that might be needed for Hyracks job construction.

5 Conclusion and Status

In this paper, we briefly looked at four declarative languages (SQL, XQuery, HiveQL, and PigLatin) used for large-scale query processing. We saw that an abstraction that extends relational algebra, allows the modeling of various types of collections (sets, bags, and lists), and is capable of providing support for nested data and

queries could be used to capture the semantics of queries expressed in the four languages we have seen here. The overlap in functionality in the Pig and Hive compilers shows that a framework that provides commonly required functionality in query compilers would help reduce the amount of code a developer has to write to implement a new language compiler.

To these ends we have developed Algebricks, a model-agnostic query compiler framework that meets the properties listed above. At UCI, we are implementing our own query language to process semi-structured data, called the ASTERIX Query Language (AQL). AQL is being built using Algebricks as the underlying compiler framework. As another proof of concept, we have already ported the HiveQL compiler to use Algebricks to compile HiveQL queries to run on the Hyracks data-parallel platform. We have seen a performance improvement of up to 9x and an average performance improvement of about 4x, at scale, compared to Hive on Hadoop. VXQuery is a project currently being incubated at the Apache Software Foundation that aims to provide a standards compliant XQuery 1.0 query processor that runs on large amounts of XML data using the Hyracks data-parallel platform. VXQuery also uses Algebricks as its compiler framework. Currently, VXQuery reuses Hyracks (90 KLOC) and Algebricks (36 KLOC) and provides a complete parallel XQuery processor for an additional 45 KLOC. More Algebricks implementation and performance details will be available in [1].

References

[1] Algebricks: A Compiler Toolkit for Building Parallel Query Languages for Big Data. In Preparation.

[2] Apache Hive website. `http://hive.apache.org`.

[3] Two Hive UDAF to convert an aggregation to a Map. http://www.adaltas.com/blog/2012/03/06/hive-udaf-map-conversion/.

[4] XQuery 1.0: An XML Query Language (Second Edition), 2010.

[5] Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org.

[6] The VXQuery Project. http://incubator.apache.org/projects/vxquery.html.

[7] ASTERIX Website. `http://asterix.ics.uci.edu/`.

[8] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: a Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC*, pages 119–130, New York, NY, USA, 2010. ACM.

[9] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-World Models. *Distrib. Parallel Databases*, 29:185–216, June 2011.

[10] P. Biron, A. Malhotra, World Wide Web Consortium, et al. XML Schema Part 2: Datatypes. *World Wide Web Consortium Recommendation REC-xmlschema-2-20041028*, 2004.

[11] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *ICDE*, pages 1151–1162, 2011.

[12] Vinayak R. Borkar, Michael J. Carey, and Chen Li. Inside "Big Data Management": Ogres, Onions, or Parfaits? In *EDBT*, 2012.

[13] Ronnie Chaiken, Bob Jenkins, Per A. Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[14] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Dan Suciu, and Gottfried Vossen, editors, *The World Wide Web and Databases*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer Berlin Heidelberg, 2001.

[15] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, December 2004.

[17] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. *Computer Networks*, 31:1155 – 1169, 1999.

[18] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. Xquery 1.0 formal semantics. *W3C Working Draft*, 2001.

[19] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 data model. *W3C working draft*, 15, 2002.

[20] Alan F. Gates, Jianyong Dai, and Thejas Nair. Apache Pig's Optimizer. *IEEE Data Engineering Bulletin*, 35(1), March 2013.

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[22] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Revised Papers from the 8th International Workshop on Database Programming Languages*, DBPL '01, pages 149–164, London, UK, UK, 2002. Springer-Verlag.

[23] Jaql Website. `http://jaql.org/`.

[24] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Bennet Vance, Scott L. Vandenberg, and Stanley B. Zdonik. The AQUA Data Model and Algebra. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *DBPL*, Workshops in Computing, pages 157–175. Springer, 1993.

[25] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[26] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. Technical report, IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[27] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-Foreign Language for Data Processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[28] Yannis Papakonstantinou, Vinayak Borkar, Maxim Orgiyan, Kostas Stathatos, Lucian Suta, Vasilis Vassalos, and Pavel Velikhov. XML Queries and Algebra in the Enosys Integration Platform. *Data and Knowledge Engineering*, 44(3):299 – 322, 2003.

[29] Pig Website. `http://hadoop.apache.org/pig`.

[30] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[31] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564. VLDB Endowment, 2005.

[32] Ashish Thusoo. Personal Communication, August 2009.

[33] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

CALL FOR PARTICIPATION

29th IEEE International Conference on Data Engineering
Brisbane, Australia
April 8-12, 2013

www.icde2013.org

The annual ICDE conference addresses research issues in designing, building, managing, and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences.

Venue:
Brisbane is the capital of the Sunshine State, Queensland. It has a population of about 2 million, making it the third-largest city in Australia. With the Gold Coast to the south and the Sunshine Coast to the north, year-round warm climate, spectacular scenery and pleasant locals, Brisbane has lots to offer. The conference will be held at the Sofitel Hotel located within walking distance of Brisbane 's Central Business District.

Affiliated Workshops:
- ICDE Ph.D. Symposium
- Data-Driven Decision Guidance and Support Systems (DGSS)
- Data Engineering Meets the Semantic Web (DESWEB)
- Data Management and Analytics for Semi-Structured Business Processes (DMA4SP)
- Data Management in the Cloud (DMC)
- Graph Data Management: Techniques and Applications (GDM)
- Intelligent Data Processing on Health (IDP)
- Mobile Data Analytics (MoDA)
- Privacy-Preserving Data Publication and Analysis (PrivDB)
- Self-Managing Database Systems (SMDB)

Conference Events:
- Research papers
- Industrial papers
- Demos
- Keynotes
- Tutorials
- Panels
- Workshops

General Chairs:
- Rao Kotagiri *The University of Melbourne, Australia*
- Beng Chin Ooi *National University of Singapore, Singapore*

Program Committee Chairs:
- Christian S. Jensen *Aarhus University, Denmark*
- Chris Jermaine *Rice University, USA*
- Xiaofang Zhou *The University of Queensland, Australia*

IEEE computer society

SAP Microsoft Events Queensland Queensland Government hp

The University Of Queensland AUSTRALIA CSIRO RMIT UNIVERSITY Oracle Labs

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398