Analyzing Massive Astrophysical Datasets: Can Pig/Hadoop or a Relational DBMS Help?

Sarah Loebman[1], Dylan Nunley[2], YongChul Kwon[3], Bill Howe[4], Magdalena Balazinska[5], and Jeffrey P. Gardner[6]

University of Washington, Seattle, WA

[1]sloebman@astro.washington.edu

{[2]dnunley,[3]yongchul,[4]billhowe,[5]magda}@cs.washington.edu

[6]gardnerj@phys.washington.edu

***Abstract*— As the datasets used to fuel modern scientific discovery grow increasingly large, they become increasingly difficult to manage using conventional software. Parallel database management systems (DBMSs) and massive-scale data processing systems such as MapReduce hold promise to address this challenge. However, since these systems have not been expressly designed for scientific applications, their efficacy in this domain has not been thoroughly tested.**

In this paper, we study the performance of these engines in one specific domain: massive astrophysical simulations. We develop a use case that comprises five representative queries. We implement this use case in one distributed DBMS and in the Pig/Hadoop system. We compare the performance of the tools to each other and to hand-written IDL scripts. We find that certain representative analyses are easy to express in each engine's high-level language and both systems provide competitive performance and improved scalability relative to current IDL-based methods.

I. Introduction

Advances in high-performance computing are having a transformative impact on many scientific disciplines. Advances in compute power and an increased ability to harness this power enable scientists, among other tasks, to run simulations at an unprecedented scale. Simulations are used to model the behavior of complex natural systems ranging from the interaction of subatomic particles to the evolution of the universe. These simulations produce an ever more massive amount of data that must be analyzed, interacted with, and understood by the scientist.

While scientists have access to tools and an unprecedented amount of computational resources to run increasingly complex simulations, their ability to analyze the resulting data remains limited. The reason is not lack of expertise, but simple economics. A simulation code is typically used by very large number of researchers, and it is often in use for 10 years or more. Data analysis applications, however, are often unique to individual researchers and evolve much more quickly. Therefore, while it may be affordable for a science discipline to invest the time and effort required to develop highly scalable simulation applications using hand-written code in languages like Fortran or C, it is often infeasible for individual researchers to each invest a similar effort in developing their own scalable hand-written data analysis applications. Consequently, data analysis is becoming the bottleneck to discovery: It is not uncommon for scientists to artificially limit the scale or resolution of their simulations in order to accommodate inadequate data analysis tools.

Existing parallel database management systems, such as Oracle [1], DB2 [2], Teradata [3], and Greenplum [4], and new types of massive-scale data processing platforms, such as MapReduce [5], Hadoop [6], and Dryad [7], can potentially facilitate these data analysis tasks. Each system is equipped with a high-level language (*e.g.*, SQL [8], DryadLINQ [9], Pig Latin [10], or Sawzall [11]). Programs written in these languages are compiled into a graph of operators called a *plan*. The plan is then executed as a parallel program distributed across a cluster. However, the ability of these tools to support realistic scientific analysis is not clear; some argue that these systems are wholly ill-suited to the task [12]. Although these systems have limitations, the absence of fully developed alternatives [12] raises the question: are these systems still better than the state of the art, which consists of writing custom scripts and code to carry out the analysis?

In this paper, we strive to answer this question for one specific domain: astrophysical simulations. DBMSs have not been previously applied to this data type: DBMSs are used by observational astronomers to store sky survey images [13], [14], but not to store astrophysical simulation results.

We explore the emergent data management needs of the University of Washington's "N-body Shop" group, which specializes in the development and utilization of large-scale simulations (specifically, "N-body tree codes" [15]). These simulations serve to investigate the formation and evolution of large scale structures in the universe. The UW N-body Shop is representative of the current state-of-the-art in astrophysical cosmological simulation. In 2008, the N-body Shop was the 10th largest consumer of NSF Teragrid [16] time, using 7.5 million CPU hours and generated 50 terabytes of raw data, with an additional 25 terabytes of post-processing information. In this paper, we discuss the analysis of the results of three simulations with snapshots (*i.e.* a snapshot of the state of the simulated system at a single instant in time) ranging in size from 170 MB to 36 GB. The total amount of data generated by each simulation (*i.e.*, all snapshots together) ranges in size from 55 GB to a few TB.

We find that the required analysis involves three types of tasks: filtering and correlating the data at different times in

978-1-4244-5012-1/09/$25.00 ©2009 IEEE

the simulation, clustering data within one simulated timestep, and querying the clustered data. In this paper, we focus on the first group of tasks and study how well two existing types of tools — a distributed relational DBMS and a MapReduce-like system with a declarative front-end called Pig/Hadoop [10], [6] — support this analysis. We implement the use case in each of the two tools and measure its performance on clusters of different sizes. We compare the performance and usability of these two tools to the state-of-the-art manually written programs and scripts. We report our experimental results and also discuss our general experience with using these tools in this specific context including what we find to be some benefits and limitations of each type of system.

Overall, we find that both types of engines offer a compelling alternative to current scientific data management strategies. In particular, the relational DBMS offers excellent performance when manipulating datasets whose size significantly outweighs the available memory. On the other hand, the Pig/Hadoop system offers somewhat more consistent speed-up and performs as well as the DBMS once the cluster becomes sufficiently large.

II. Astronomy Simulation Application Domain

Cosmological simulations are used to study how structure forms and evolves in the universe on size scales ranging from a few million light-years to several billion light-years. Typically, the simulations start shortly after the Big Bang and run the full lifespan of the universe, roughly 14 billion years. Simulations provide an invaluable contribution to our understanding of how the universe evolved, because the overwhelming majority of matter in the universe does not emit light and is therefore invisible to direct detection by telescopes. Furthermore, simulations represent our only way to experiment with cosmic objects such as stars, galaxies, black holes, and clusters of galaxies. For these reasons, astrophysics was among the first scientific disciplines to utilize leading-edge computational resources, a heritage that has continued to the present day.

In the simulations under discussion, the universe is represented by a set of particles. There are three varieties of particles: dark matter, gas, and stars. Some properties (*e.g.*, position, mass, velocity) are common to all particles. Other properties (*e.g.*, formation time, temperature, chemical content) are specific only to certain types. All particles are points in a 3D space and are simulated over a series of discrete timesteps. Every few timesteps, the simulator outputs a snapshot of the state of the simulated universe. Each snapshot records all properties of all particles at the time of the snapshot. Simulations of this type currently have between $10^8 - 10^9$ particles (with approximately 100 bytes per particle) and output between a few dozen and a few hundred snapshots per run. Table I shows the names of each simulation used in our experiments and the sizes of their snapshot files. The largest run, cosmo25, has a total of a few TB of snapshot files.

For the UW N-body group, the state-of-the-art approach to data analysis in the absence of relational DBMSs and MapReduce is to use a combination of many handwritten tools, programs, and scripts created over the past 15 years. Since the simulation code generates one output file per snapshot, all post-processing tools are written to read this standard snapshot-based format. Many of these tools generate additional data, which are then stored in separate files.

TABLE I

Simulation datasets studied in the paper.

Name	No. Particles	Snapshot Size (binary)
dbtest128g	4.2 million	169 MB
cosmo50	33.6 million	1.4 GB
cosmo25	916.8 million	36 GB

Some tools were developed by the N-body Shop, and are widely used within the astrophysical simulation community. One such tools is TIPSY [17], an X11-based particle visualization tool modeled after the Santa Cruz/Lick image processing package VISTA. There are also several codes, "group finders" [18], [19], [20], [21] that identify physically meaningful collections of particles within the simulation snapshots. Other tools are scripts that are usually written in interpreted languages such as Python, Perl, or Interactive Data Language (IDL) [22]. A trait common to these tools is that they operate in main memory — they require at least one full snapshot to be read into RAM before any processing can begin.

Astrophysicists are facing two big challenges in continuing to use existing strategies for data analysis. The first is scalability: the size of the simulations is growing much faster than the memory capacity of shared-memory platforms on which to deploy serial data analysis software. Secondly, the I/O bandwidth of a single node has increased little over the past decade, meaning that simulation snapshots are incurring an increasingly long delay when loading data into memory. Consequently, the queries that filter and correlate data from different snapshots require very large main memories RAM and become highly I/O constrained.

Scalability in RAM and I/O bandwidth can be achieved by harnessing large numbers of distributed-memory nodes, provided one's data analysis software can scale on such architectures. The astrophysical community has investigated the efficacy of application-specific parallel libraries to reduce the development time of scalable in-RAM data analysis pipelines. For example, *N*tropy [23], [24] provides a parallel library for the analysis of massive particle datasets, and *N*tropy applications can scale to thousands of distributed-memory nodes. The principle disadvantages of *N*tropy are that 1) it still requires a steep learning curve and 2) this knowledge is not transferable outside of particle-based scientific datasets.

DBMSs and frameworks like Hadoop [6] and Dryad [7] offer potentially the same scalability as application-specific libraries, but can also be used across a broad range of science applications. They may also be easier to learn, especially if the researcher can utilize the declarative languages built on top of the frameworks, such as Pig Latin [10] and DryadLINQ [9]. Consequently, we wish to assess the effectiveness of these

TABLE II
ATTRIBUTES DESCRIBING PARTICLES IN A SIMPLE COSMOLOGY SCHEMA

Attribute	Description	Particles described
iOrder	unique identifier	all
X,Y,Z	position in Cartesian coordinates	all
V_x,V_y,V_z	velocity components	all
Phi	gravitational potential	all
Metals	proportion of heavy elements	gas, stars
Tform	formation time	stars
Eps	gravitational softening radius	stars, dark matter
DenSmooth	local smoothed density	dark matter
Hsmooth	smoothing radius	gas
Rho	density	gas

tools for scientific data analysis in the regime where current state-of-the-art strategies have become constrained by I/O bandwidth and shared memory requirements.

III. USE CASE

In order to study how well existing data management systems support the analysis tasks described above, we develop a concrete use case comprising a small number of representative queries.

A. Data Model

Since different species of particles have different properties, we model the data as three different relations, or tables: one per type of particle. The attributes for each relation are the ones shown in Table II. Furthermore, we horizontally partition each table: Each partition holds the data for one snapshot and one species of particles.

For each dataset, we analyzed two snapshots of data. Snapshots are output from the simulation in TIPSY binary format. In this binary format, this amounted to 169 MB, 1.4 GB, and 36 GB of data from simulations dbtest128g, cosmo50, and cosmo25 respectively.

B. Selection and Correlation Queries

The simplest form of analysis that astronomers perform on the output of a simulation is to look-up specific particles that match given conditions.

Q1: Return all particles whose property X is above a given threshold at step S1.

Q1 corresponds to one of the simplest possible filtering queries: find all gas particles with temperature greater than 150000 K. Gas particles at temperatures at or above 10^5 K are categorized as "warm/hot"; these particles are typically found in the regions between galaxies known as the Intergalactic Medium (IGM). Astronomers seek to understand properties of the IGM such as spatial extent and total mass, and a temperature threshold is an effective way to distinguish particles belonging to the IGM from the rest of the simulation.

These filtering queries correspond in SQL to what is known as *select* queries, since they return only selected rows.

Q2: Return all particles of type T within distance R of point P.

Q2 is another common type of select query that involves returning particles based upon physical proximity. In contrast to Q1, Q2 demonstrates the need for spatial operators and help us assess the expressiveness of the competing tools. A concrete version of Q2 which we use in our experiments finds all the stars that lie within the *virial radius* of the center of mass of a given *halo*. A halo is the ensemble of particles that surrounds a galaxy or cluster of galaxies. The virial radius is the radius of a sphere, centered on a galaxy, within which *virial equilibrium* holds. Virial equilibrium describes a system in dynamic balance; stars within the virial radius are gravitationally bound to the system. Astronomers use the virial radius to consistently define the edge of the halo.

Q3: Return all particles of type T within distance R of point P whose property X is above a threshold computed at timestep S_1.

Complex calculations can be expressed in a database as *user-defined functions* (UDFs). A UDF provides astronomers a framework for expressing more sophisticated selection conditions based upon data interdependencies.

Our concrete example of Q3 uses a UDF to calculate the virial temperature from a galaxy's mass and radius. An astronomer might ask to return all gas particles whose temperature is above the virial temperature at a given step within a given halo. Such particles are considered *shocked*. The proportion of shocked gas to total gas tells astronomers about how the galaxy has been assembled over time.

Q4: Return gas particles destroyed between step S_1 and S_2.

In addition to the select queries discussed above, scientists often need to correlate information across two snapshots. In particular, astronomers often want to track the evolution of particles. For example, a scientist may want to see the identifiers of all particles destroyed between two snapshots. Such queries correspond in SQL to what are known as *join* queries since they "join" together the information from two different snapshots.

A join operator finds pairs of tuples, one from each of two input relations, that share some property. In this query, the join condition associates identical particles using the unique identifier, `iOrder`. Destroyed particles are identified as those values of `iOrder` from timestep S_1 that do not have a corresponding partner in timestep S_2.

In these simulations, stars form from gas; each gas particle can form up to four star particles before it is deleted from the simulation. For our concrete version of Q4, we select deleted gas particles, as they indicate regions of vigorous star formation.

Q5: Return all particles whose property X changes from S_1 to S_2.

Along with tracking how many particles are created or deleted, astronomers are often also interested in the change

of properties over time. For example, when stars explode in supernovae, they release heavy elements called *metals* into the intergalactic medium. Astronomers are interested in tracking what happens to the gas that encounters these metals and becomes enriched. Our concrete version of Q5 selects gas that starts with zero metal content at one step and has a non-zero metal content at a later time step so as to return recently enriched material.

In summary, the simplest form of data analysis can thus be expressed in the form of select-project-join (SPJ) queries (the "project" operator simply discards unwanted attributes). The general class of SPJ queries is considered to be the simplest non-trivial subset of the full *relational algebra*, the formal basis for SQL.

Simple queries such as Q1, Q2, and Q3 admit a trivial parallel evaluation strategy: partition the data across N nodes, evaluate the same query on each node, and merge the results. These search-oriented queries are sometimes amenable to the use of indexes to quickly locate particles that match a specific condition. If indexes are not available, the best strategy is to do an exhaustive scan of every particle as quickly as possible.

Queries Q4 & Q5 can be expensive to compute because they require correlating two large datasets. A naive algorithm might build a large data structure in memory for particles in S_1 then probe this data structure for each value from S_2. This algorithm is limited by the available memory of the computer. Alternatively, relational databases perform a similar algorithm, but automatically dividing the work into pieces that fit in main memory — the limiting factor is then the size of the disk rather than available main memory[1]. Parallel programming models such as MapReduce use a similar mechanism to provide scalability across multiple computers: each computer handles a particular set of particle identifiers, so no one computer performs too much work. We evaluate both approaches in Section VI.

IV. Relational DBMS Implementation

Relational DBMSs (RDBMSs) have a long history of providing high performance when querying or updating disk-resident data [25]. In our experiments, we use a common, commercial RDBMS, which we call Database Z. We performed no special tuning of Database Z.

In a RDBMS, queries are expressed in SQL — a *declarative* language in which users describe the result they need, but not specifically how to go about obtaining it. For example, the queries in our use case can be expressed in SQL as shown in Table III. The RDBMS analyzes each such query and decides on an evaluation strategy, called a *query plan*. The space of possible plans is large in part since the underlying formalism of SQL called the *relational algebra* admits a variety of algebraic rewrite rules. Additionally, each logical operator has many different physical implementations that the system can choose from. For example, in Q5, the system can choose to apply the condition `metals=0` either before or after joining timesteps $g1$ and $g2$. It can then choose from one of several join algorithms depending on the properties of the input data. Out of this space of plans, the system selects a good plan by estimating the cost of the candidates. All modern DBMS perform this style of *cost-based algebraic optimization*.

RDBMSs use indexes to reduce the number of I/O operations in a query plan. An index is a data structure that enables the DBMS to skip over irrelevant data without reading it into memory. The two most common indexes used in an RDBMS are the *hash index* (amortized constant time access) and the *B+ tree* index (logarithmic time access with robust performance on varying data). Indexes may have additional properties that affect their performance [26].

Declarative queries provide a means to automatically achieve "disk scalability" — as long as the dataset fits on disk, the RDBMS is guaranteed to finish every query. A RDBMS will never crash with an out-of-memory error, and, if configured properly, will never thrash virtual memory. The reason these guarantees are possible is that the user cannot control which algorithms are used, and therefore cannot instruct the RDBMS to anything untoward such as loading the entire dataset into memory before operating on it. An SQL query is also frequently far shorter and simpler than an equivalent implementation in a general-purpose programming language.

Query 1. As shown in Table III, Q1 specifies that we are interested in the `iOrder` of all particles inside table `gas43`, which represents all gas particles in snapshot 43 of our largest simulation, cosmo25, with attribute `temp` above a threshold. To improve performance, we created a B-tree index on the `temp` attribute using a simple `CREATE INDEX` command.

Query 2. Q2 is an example of query with a spatial predicate that is awkward to express in plain SQL (though ,any relational DBMSs now include spatial extensions). Database Z supports a spatial index but we did not use it in order to study query performance when all input data must be read, as in the case of Pig and other MapReduce-like systems.

Query 2 also exhibits an unusual idiom: Multiple relations referenced in the `FROM` clause, but no join condition between them. In this case, SQL generates the *cross product* of the relations: every possible pairwise combination of tuples is produced in the output. In this case, this idiom is used only to bring just one appropriate tuple into context from the `stat43` table, so the number of tuples does not actually increase.

Query 3. Q3 is similar to Q2, but 1) adds an additional condition on the `temp` attribute, and 2) expresses the temperature threshold as a calculation involving a user-defined function. The index on temperature we created for Q1 can potentially be reused in this query.

Query 4 & 5. Q4 and Q5 both involve joins on the `iOrder` attribute, so we create an index on `iOrder`. In a production system, `iOrder` would likely be designated a *primary key*, which would enable additional optimizations that we discuss in Section VI. Q4 could be expressed using either a join or a set difference operation. We chose the former formulation to test the general performance of correlation queries. In particular,

[1]DBMSs also use other algorithms for joining large relations such as "nested-loop" and "sort-merge".

we identified particles at one timestep that have no matching partner in a later timestep, implying that the particle was destroyed during the course of the simulation.

Distributed query processing. Several RDBMSs [4], [27], [28] support *parallel* queries, where data can be partitioned across several nodes and accessed simultaneously. An RDBMS can support *inter-operator* parallelism or *intra-operator* parallelism or both. Inter-operator parallelism refers to the ability of the query scheduler to map different subplans to different nodes for simultaneous execution. Intra-operator parallelism refers to the ability to instantiate multiple copies of one operator that work in concert to process a partitioned dataset. Both forms rely crucially on partitioning the data appropriately.

Database Z does not support intra-operator parallelism, but it does support a robust and thorough form of inter-operator parallelism. We use the term *distributed queries* to distinguish the class of queries supported by this system from those involving intra-operator parallelism [4], [27], [28], [29], [30].

In the system that we study, remote database servers may be *linked* to a head instance of Database Z, allowing them to be referenced in queries using an extra layer of qualification. For example, the following query accesses data from a table named `gas43` in the default namespace `dbo` of a database named `cosmo50` on a remote server named `orlando`:

```
SELECT * FROM orlando.cosmo50.dbo.gas43
```

It would be tedious and error-prone to explicitly reference every node in the cluster each time one wished to query a partitioned table. Worse, if the data is reorganized to utilize more nodes, existing queries would not function until they were updated. Thankfully, relational databases natively provide a mechanism for *logical data independence* in the form of *views*. A view is simply a named query that can be referenced as a table in another query. For example, the following statement creates a view named `gas60_dist` computed as the union of three subqueries, each referencing a different table on a different node in the cluster. The output of this query is not pre-computed. The view is like a virtual table.

```
CREATE VIEW gas43_dist AS
SELECT * FROM orlando.cosmo50.dbo.gas43
  UNION ALL
SELECT * FROM newyork.cosmo50.dbo.gas43
  UNION ALL
SELECT * FROM beijing.cosmo50.dbo.gas43
```

This kind of view is referred to as a *distributed partitioned view* in the Database Z documentation. The `UNION ALL` clause merges the results from two subqueries. This clause is distinguished from `UNION`; the latter checks for and removes any duplicate tuples, which is an expensive operation that can be avoided if the partitions are known to be disjoint a priori.

With this view established, we can write the following query to access data across all three nodes transparently:

```
SELECT * FROM gas43_dist
WHERE temp > 150000
```

No explicit knowledge of or reference to the distribution policy is necessary — the user need not care if this database is running on a single enterprise-scale server or across many commodity workstation-class machines.

Since the remote servers are known to support full SQL queries, the optimizer on the head node that receives the query is free to reorder operations to push more work down to the data nodes. For example, in the query above, a naïve evaluation strategy would be to stream all tuples from all nodes back to the head node, where the filter `temp > 150000` would be applied. A better plan, however, is one found automatically by Database Z. It pushes the filter down to each node, essentially generating an equivalent query of the form:

```
SELECT * FROM orlando.cosmo50.dbo.gas43
WHERE temp > 150000
  UNION ALL
SELECT * FROM newyork.cosmo50.dbo.gas43
WHERE temp > 150000
```

The Database Z optimizer does not actually generate a new SQL statement; queries are represented internally as relational algebra expressions. This example simply illustrates the semantics of the optimization applied.

In our experiments, we partitioned the data across 1, 2, 4 and 8 nodes to assess scalability and performance of distributed queries in Database Z. We created the appropriate views for each configuration. We manually adjusted a few of these views to compel the optimizer to use a more efficient plan.

V. Pig/Hadoop Implementation

MapReduce [5] is a programming model for processing massive-scale data sets in large shared-nothing clusters. Users specify a *map* function that generates a set of key/value pairs, and a *reduce* function that merges or aggregates all values associated with the same key. A single combination of a map function and a reduce function is called a *job*. In SQL, a MapReduce job can usually be expressed as an aggregation query (*i.e.*, a query involving a `GROUP BY` clause).

MapReduce programs are automatically parallelized and executed on a cluster. Data partitioning, scheduling, and inter-machine communication are all handled by the run-time system. Note, however, that these systems do not offer any "smart" partitioning. The input data is simply split into fixed-size chunks (a recommended 128MB in our experiments) that are spread across nodes. In contrast, relational DBMSs enable users to specify how the data should be partitioned and exploit the partitioning knowledge during query optimization. Overall, however, MapReduce provides a lightweight alternative to parallel programming and is becoming popular in variety of domains.

Hadoop [6] is an open-source implementation of MapReduce written in Java. Because Hadoop has no predefined schema, data must be repeatedly parsed at the beginning of each query. However, because the parsing algorithm we used is extremely straight-forward, this extra step did not introduce any significant overhead.

The Pig [10] engine provides a high-level language over the low level map and reduce primitives to simplify programming. Users write procedural scripts in *Pig Latin* that are compiled

TABLE III

IMPLEMENTATIONS OF Q1-Q5 IN SQL AND PIG LATIN.

Q#	SQL	Pig Latin

Q1	SELECT iOrder
	FROM gas43
	WHERE temp > 150000
		// LOAD and STORE statements are only presented for Q1
		rawGas = LOAD 'cosmo25cmb.768g.00043_gas.bin'
		USING quark.pig.BinCosmoLoad('gas');
		gas = FOREACH rawGas
		GENERATE $0 AS pid:long, $10 AS temp:double;
		filteredGas = FILTER gas BY temp > 150000;
		filteredGasPid = FOREACH filteredGas GENERATE pid;
		STORE filteredGasPid INTO 'q1_cosmo25.00043_gas';

Q2 and Q3	SELECT g.iOrder
	FROM gas43 g, stat43 h
	WHERE (g.x-(h.Xc-12.5)/25)*(g.x-(h.Xc-12.5)/25)
	+ (g.y-(h.Yc-12.5)/25)*(g.y-(h.Yc-12.5)/25)
	+ (g.z-(h.Zc-12.5)/25)*(g.z-(h.Zc-12.5)/25)
	<= (h.Rvir*h.Rvir)/6.25e8
	AND h.HaloID = 1
	// The following condition appears only in Q3
	AND g.temp > VirialTemp(h.Rvir, h.Mvir)
		dark = FOREACH rawDark
		GENERATE $0 AS pid:long, $2 AS px:double,
		$3 AS py:double, $4 AS pz:double;
		filteredDark = FILTER dark BY
		(px-($XC-12.5)/25)*(px-($XC-12.5)/25) +
		(py-($YC-12.5)/25)*(py-($YC-12.5)/25) +
		(pz-($ZC-12.5)/25)*(pz-($ZC-12.5)/25)
		<= ($RVIR*$RVIR)/6.25e8
		// The following condition appears only in Q3
		AND g.temp > VirialTemp(h.Rvir, h.Mvir)
		filteredDarkPid = FOREACH filteredDark GENERATE pid;

Q4	SELECT g1.iOrder
	FROM gas43 g1 FULL OUTER JOIN gas60 g2
	ON g1.iOrder=g2.iOrder
	WHERE g2.iOrder is NULL
		star43 = FOREACH rawGas43 GENERATE $0 AS pid:long;
		star60 = FOREACH rawGas60 GENERATE $0 AS pid:long;
		groupedGas = COGROUP star43 BY pid, star60 BY pid;
		selectedGas = FOREACH groupedGas GENERATE
		FLATTEN((IsEmpty(gas43) ? null : gas43)) as s43,
		FLATTEN((IsEmpty(gas60) ? null : gas60)) as s60;
		destroyed = FILTER selectedGas BY s60 is null;

Q5	SELECT g1.iOrder
	FROM gas43 g1, gas60 g2
	WHERE g1.iOrder = g2.iOrder
	AND g1.metals = 0
	AND g2.metals > 0
		gas43 = FOREACH rawGas43
		GENERATE $0 AS pid:long, $2 AS px:double,
		$3 AS py:double, $4 AS pz:double,
		$12 AS metals:double;
		gas60 = FOREACH rawGas60
		GENERATE $0 AS pid:long, $2 AS px:double,
		$3 AS py:double, $4 AS pz:double,
		$12 AS metals:double;
		gas43 = FILTER gas43 BY metals == 0.0;
		gas60 = FILTER gas60 BY metals > 0.0;
		gasGrouped = COGROUP gas43 BY pid, gas60 BY pid;
		filteredGas = FILTER gasGrouped BY NOT IsEmpty(gas43)
		AND NOT IsEmpty(gas60);
		filteredGasPid = FOREACH filteredGas GENERATE group;

into a network of MapReduce jobs. Each query in our astronomy use case can be encoded as a Pig Latin script, as shown in the right-hand column of Table III. We explain these scripts in more detail in the remainder of this section.

Query 1. Q1 is a simple selection query. Here, the `LOAD` command specifies which file the system should read and how that file should be parsed. We chose to split the particle into files based on type, just as we used separate tables in the RDBMS implementation. Since Hadoop is not aware of the schema of the data before it loads it and because it provides no indexing capability, the system must load and process files in their entirety.

The two `FOREACH-GENERATE` commands simply project out unnecessary attributes to reduce the data volume. `FILTER` performs the actual selection. Finally `STORE` dumps the result into a new HDFS file. There is no other way for users to get output data in Pig/Hadoop.

Since this query requires no grouping, the entire query is performed in one MapReduce job comprising only map tasks.

Query 2. Q2 is similar to Q1 with a more complex predicate.

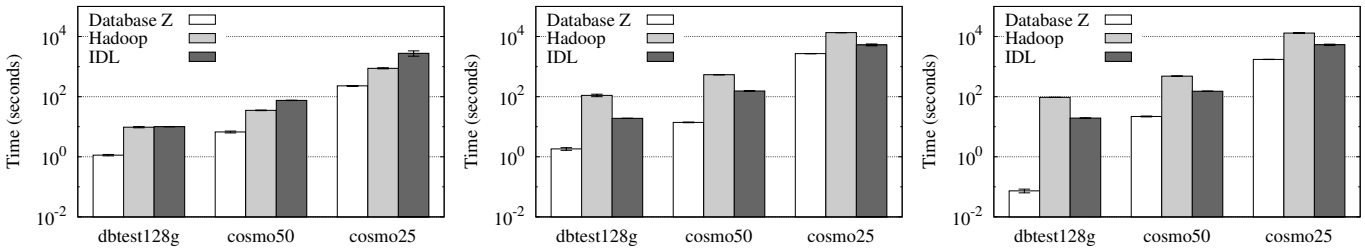
Query 3. The third query, the "select-project with UDF" query, is syntactically similar to Q2 and Q1, but uses a UDF called VirialTemp as additional filtering condition. Pig Latin posed no difficulty registering and calling a custom Java function.

Query 4. This query is most naturally expressed with a relational-style join operator. In the current version of Pig, a full outer join can be written in two steps. First, the two joined relations are `COGROUP`ed by join attribute (`pid`). For each distinct `pid` value, the COGROUP operator generates a tuple comprised of two *bags*[2] containing tuples from each relation sharing a common `pid`. The following FOREACH statement transforms the output of COGROUP into a standard full outer join output by substituting nulls for empty bags[3]. Finally, a `FILTER` is used to select destroyed particles by examining null value in `s60` field.

This query is converted into a single MapReduce job with both map and reduce tasks. The map step loads the data and sends tuples to the reduce job based on their pid values. The reduce job performs the join and filter operations. Different

[2]A bag is a collection that allows duplicates, in contrast to a *set*.

[3]In Q4, each bag contains at most one element because `pid` is unique in each input.



(a) Q1 (b) Q4 (c) Q5

Fig. 1. Single node experiments. Since the results for Q1, Q2 and Q3 are very similar, only Q1, Q4, and Q5 are presented here for brevity's sake. Please note that the log scale on the y-axis.

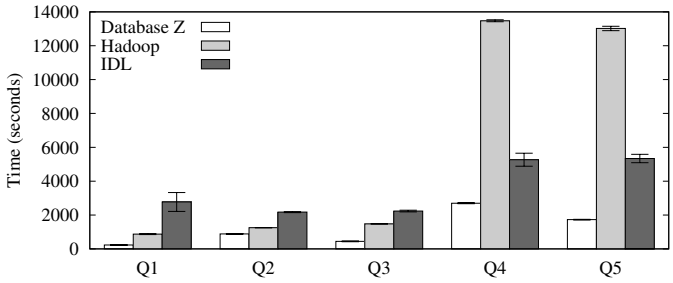


Fig. 2. Query times from single-node implementation of RDBMS, Hadoop & IDL for largest data set.

reduce tasks process tuples with different sets of pid values.

Query 5. This join query is similar to Query 4 and thus our implementation is also similar. For Q5, we load the input data into two bags because the selection conditions differ. As with Query 4, this script compiles into a single MapReduce job.

Overall, we found it quite natural to express our use case queries in the Pig Latin scripting language. We note that most of our queries only touch a few columns/variables in the dataset, and as such, a column-store system [31] [32] might have a significant advantage over a traditional RDBMS or Hadoop. We leave the study of CStore DBMSs in the context of astrophysical simulation analysis for future work.

VI. Evaluation

In this section, we present results from running all five queries on both systems and different cluster configurations. Given the relatively manageable size of our data, we use a cluster of only eight nodes. We measure two primary values: performance on a single node and speedup achieved by increasing the cluster size.

For all Database Z and Pig/Hadoop tests, each node had two quad-core Intel Xeon E5335 processors running at 2.0 GHz, 16 GB RAM, and two 500 GB SATA disks in RAID 0. The IDL platform was an SGI Altix with 16 Opteron 880 processors running at 2.4 GHz, 128 GB RAM, and 3.1 TB of RAID 6 SATA disk. The IDL system was thus significantly different from the Database Z and Hadoop platforms, and therefore the runtimes are not directly comparable.

A. Single-Node Query Performance

Figure 1 shows the single-node query latencies for three of the five queries for three datasets and three engines: Database Z, Pig/Hadoop, and IDL. The latter runs the original analysis script used by the astronomers. The figure shows the average of three executions of each query. The variance is too small to be seen. Between executions, we restarted the machine to clear all caches. Figure 2 highlights the same results for the largest dataset on a linear scale.

Overall, IDL shows the most consistent numbers across queries. We find that its execution time is completely dominated by load times. As noted previously, IDL loads all data into memory before it operates on it (which is why it needs to run with the 128 GB of RAM!). In Q1 - Q3, IDL had to read in one snapshot, while in Q4 and Q5 it had to read in two.

Interestingly, the Hadoop runtime in Figure 2, although presumably also dominated by load times, shows significantly more variance than IDL. This is because the IDL system had enough RAM to completely fit two snapshots of cosmo25, but a Hadoop node only had 16GB of RAM, enough for only half of a single snapshot.

Database Z performs better than Hadoop under the same limited memory conditions. To see why, consider Q2. The performance of all engines is most similar on Q2, where the RDBMS, like the other engines, simply scans the input data and filters it. On this query, Pig/Hadoop shows a higher overhead compared with Database Z. One reason for this is the query startup cost; Hadoop must start several new processes on remote nodes to execute a query.

Now consider Q1, where the benefits of indexing are apparent. While the performance of Pig/Hadoop and IDL remain approximately the same as for Q2, the query time for the DBMS is significantly reduced relative to Q1. The reason is that the DBMS uses the index to identify exactly those pages on disk that must be reads and only reads those. In general, the performance benefit of an index is sensitive to properties of the data, the index, and the optimizer and is therefore not generally predictable [26].

Our results for Q1 do not exploit *covering indexes*, an optimization that would typically be available in practice. Consider an index over two attributes (`temp`, `iOrder`). In this case, the DBMS does not need to use the index to

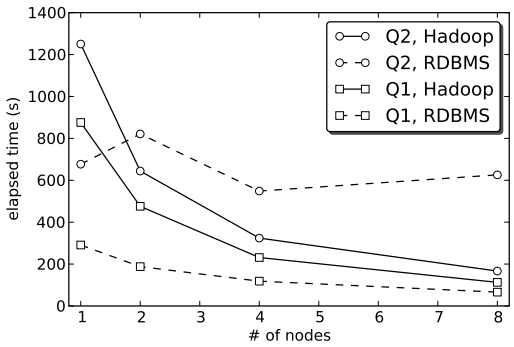


Fig. 3. Query 1 and Query 2 parallel speedup.

look up pages in the original table — all the data needed to answer the query are available in the index itself. This optimization produces remarkable performance benefits — Q1 can be evaluated in under 100 ms!

Q3 shows a similar effect as Q1 because the UDF to compute VirialTemp does not prevent the use of the index on temperature.

The improved performance of the RDBMS is especially clear on the two queries involving joins, Q4 and Q5. The RDBMS is once again able to exploit indexes. For Q4, the DBMS can answer the whole query using the (covering) index. This index was also sorted on iOrder, allowing sequential I/O instead of random I/O during the evaluation of the join.

In these experiments, we cleared all caches between runs. In practice, query times could go down for all systems thanks to caching. This would lead to the most significant runtime improvement for the IDL approach that runs on a system with 128 GB RAM and can thus cache entire snapshots at the time. To exploit such a cache, however, a scientist would need to access the same snapshot repeatedly.

In summary, in contrast to IDL, on a single node, both Pig/Hadoop and Database Z give a researcher the ability to analyze datasets that are significantly larger than the RAM of that node, and do so in a time comparable to or better than IDL on a much more expensive system with enough RAM to hold the dataset in memory. The RDBMS is especially efficient in its memory management.

B. Parallel Query Processing Performance

We evaluate speedup through experiments on the largest of the three datasets. Speedup refers to improved query execution time when using additional nodes to process the same dataset. For perfect speedup, given a data set and a query, the execution time for a cluster of N nodes should be equal to $\frac{T_1}{N}$, where T_1 is the query execution time on a single node. Figures 3 through 6 show the results. Overall, both Pig/Hadoop and the RDBMS offer adequate speedup. Hadoop exhibits the most consistent speedup for all 5 of our queries. In the worse case, (Q1) speedup is still over 80% linear on 8 nodes, with most of the other queries achieving over 90% linearity on 8 nodes. While Database Z was faster on the single node experiments, its speedup is inconsistent on some queries, especially Q2. The speedup limitations of Database Z stem from a parallel execution model that involves funneling results back to a head node, a bottleneck for scalability. We expect this problem to become more apparent as the number of nodes increases. For our test dataset, Hadoop seems to be at least as fast as Database Z on 8 nodes, and in Q2 it is significantly faster.

More specifically, Figure 3 compares performance for both Q1 and Q2. For Q1 (squares), we see smooth speedup in both Hadoop and Database Z. Both systems exhibit diminishing returns as the number of nodes increases. An index on the `temp` attribute and a relatively small query result size minimizes the overhead of streaming all result tuples back to the head node, so Database Z outperforms Hadoop easily.

In Q2 (circles), Hadoop exhibits a smooth speedup with some diminishing returns. Database Z however is erratic — additional nodes do not always reduce elapsed time. This result is consistently reproducible: two nodes are slower than one node, and eight nodes are more expensive than four nodes. This result demonstrates the overhead of providing parallel processing. Although it is difficult to ascertain exactly where the additional costs are incurred, it is clear from the other queries that there is some cost to coordinating a distributed query. Query latency improves only when this overhead is outweighed by the time savings in processing each data partition much faster.

In Figure 4, we see that Q3 exhibits similar characteristics to Q2: the overhead of managing a distributed query is apparent in the slope between the 1-node case and the 2-node case, but the parallelism still pays off in this case. Once again, 4 nodes appears to be the "sweet spot." Speedup and overall performance between Hadoop and Database Z are comparable for this query.

Q4 and Q5 are the most expensive queries for both systems (Figure 5 and Figure 6). In both cases, Database Z significantly outperforms Hadoop, though the effect is diminished as the number of nodes increases. Interestingly, the speedup curve for Database Z is quite flat. Although this is generally a bad sign, the comparison with Hadoop shows that the flatness is better attributed to the strong performance of the single node case than simply poor speedup.

It is important to point out that these experiments do not properly represent the operational performance of Database Z — the database was unfairly penalized by not allowing it to exercise some of its most important features such as caching. We also did almost no physical database tuning (selecting views to materialize, which is a form of caching, changing properties of indexes, etc.).

However, this strength also illustrates a well-known weakness of relational databases: they are complicated pieces of software that require significant experience to operate properly. A typical research lab cannot afford a Database Administrator on staff, so the full potential of a RDBMS may be difficult to realize in practice. This problem is well-known in the database community, who are studying self-managing and self-tuning enhancements to databases to simplify the cognitive load.

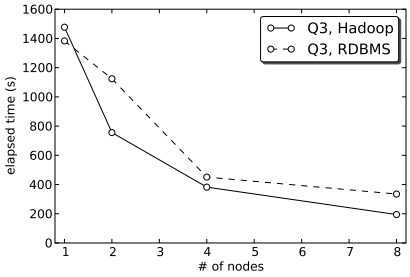


Fig. 4. Query 3 parallel speedup.

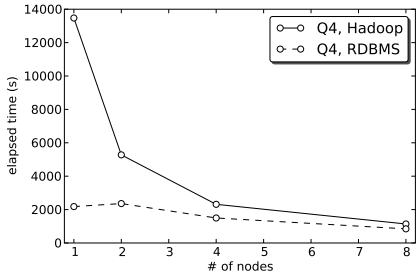


Fig. 5. Query 4 parallel speedup.

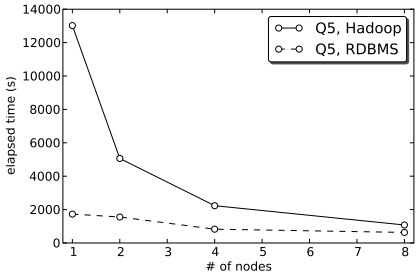


Fig. 6. Query 5 parallel speedup.

Hadoop is also somewhat unfairly represented. Hadoop is designed for massive parallelism — hundreds or thousands of nodes. Testing on such few nodes means that important features such as intra-query fault tolerance are never exercised. The overhead of having central coordination of a distributed query is also not evident with only eight nodes.

In summary, on these larger cluster configurations, Database Z outperforms Hadoop on join queries and one of the selection queries. This makes sense as our queries are I/O dominated, and Database Z minimizes I/O whereas Hadoop must read in the entire dataset.

VII. Related Work

Parallel query processing in a shared-nothing architecture has a long history [33], [34], [35], [36], [37], [38], [39] and many high-end commercial products are on the market today [4], [27], [28], [29], [30]. Recently, a new generation of systems have been introduced for massive-scale data processing. These systems include MapReduce [5], [40] and similar massively parallel data processing systems (*e.g.*, Clustera [41], Dryad [7], and Hadoop [6]) along with their specialized languages [9], [10], [11], [42]). All these systems, however, have not been designed specifically for scientific workloads and many argue that that they are ill-suited for this purpose [12]. Our goal in this paper was to evaluate how well two examples of such systems, one of each type, could be used for scientific data analysis in astronomy simulation research.

There is little published work that compares the performance of different parallel data processing systems. Recently, Pavlo *et al.* [31], comparatively benchmarked a parallel relational DBMS, Hadoop [6], and Vertica [28], a column-store parallel DBMS. The goal of their work was to evaluate the performance of these three systems in general. The study thus used synthetic data. In contrast, our goal is to measure how suitable such systems are for scientific data analysis and, in particular, astronomy simulation analysis. Hence, our work focuses on real queries important in the astronomy domain and uses real data from that domain. Additionally, we compared the DBMS to Pig, which brings a declarative layer to Hadoop. For these reasons (and also due to differences in settings and relational DBMS used), some of our findings differ from those in this previous study. In particular, in our setting, we did not find that the relational DBMS consistently outperformed Hadoop. Instead, different engines performed better on different queries.

Benchmarks demonstrating the performance of relational DBMSs [25], Hadoop [43], or Pig [44] have previously been published. These benchmarks, however, evaluate these systems in finely tuned environments, using synthetic data, and synthetic tasks such as sorting. In contrast, we evaluated these systems within the context of a specific application and without extensive tunings.

Cary *et al.* [45] studied the applicability of MapReduce to spatial data processing workloads and confirmed the excellent scalability of MapReduce in that domain. In contrast, our work focuses on data from a different domain, experiments with a different engine (Pig instead of raw MapReduce) and also compares the performance with that of a RDBMS.

The Sloan Digital Sky Survey [46] has successfully used relational DBMSs to store and serve astronomy data. Their data, however, comes from telescope imagery rather than large-scale simulations.

Palankar *et al.* [47] recently evaluated Amazon S3 [48] as a feasible and cost effective alternative for hosting scientific datasets, particularly those produced by large community collaborations such as LSST [49]. This work is orthogonal to ours since S3 is purely a storage system. It does not provide any data management capabilities beyond storing and retrieving objects based on their unique keys [48].

There are several ongoing efforts to build new types of database management systems for sciences [12], [50]. Evaluating the applicability of these systems to astronomy simulation or other scientific analysis tasks is an area of future work.

VIII. Conclusion

In this paper, we evaluated the performance of a commercial RDBMS and Hadoop on astronomy simulation analysis tasks. We found that it was natural and concise to express the required analysis tasks in these tools' respective languages. On small-to-medium scale clusters (<10 nodes), we found the modern RDBMS to offer a powerful platform for organizing data and an excellent constellation of features for improving performance: indexes, distributed queries, cost-based algebraic optimization, and logical data independence. Our tests also indicate that overall a RDBMS outperforms Hadoop and IDL for representative I/O-dominated queries in the astronomy

domain. However, we acknowledge that Hadoop is designed for scalability to hundreds or thousands of nodes, and is likely to outperform a RDBMS in this context.

ACKNOWLEDGMENTS

It is a pleasure to acknowledge the help we have received from Tom Quinn, both during the project and in writing this publication. Simulations "Cosmo25" and "Cosmo50" were graciously supplied by Tom Quinn and Fabio Governato of the University of Washington Department of Astronomy. The simulations were produced using allocations of advanced NSF–supported computing resources operated by the Pittsburgh Supercomputing Center, NCSA, and the TeraGrid.

This work was funded in part by the NASA Advanced Information Systems Research Program grants NNG06GE23G, NNX08AY72G, NSF CAREER award IIS-0845397, NSF CRI grant CNS-0454425, the eScience Institute at the University of Washington, gifts from Microsoft Research, and Balazinska's Microsoft Research New Faculty Fellowship.

REFERENCES

[1] "Oracle," http://www.oracle.com/database.
[2] "Db2," http://www.ibm.com/db2/.
[3] "Teradata," http://www.teradata.com/.
[4] "Greenplum database," http://www.greenplum.com/.
[5] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proc. of the 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.
[6] "Hadoop," http://hadoop.apache.org/.
[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. of the European Conference on Computer Systems (EuroSys)*, 2007, pp. 59–72.
[8] ISO/IEC 9075-*:2003, *Database Languages - SQL*. ISO, Geneva, Switzerland.
[9] 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," 2008.
[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proc. of the SIGMOD Conf.*, 2008, pp. 1099–1110.
[11] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, 2005.
[12] M. Stonebraker, J. Becla, D. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik, "Requirements for science data bases and SciDB," in *Fourth CIDR Conf. Perspectives*, 2009.
[13] "SDSS SkyServer DR7." [Online]. Available: http://skyserver.sdss.org
[14] V. Singh, J. Gray, A. Thakar, A. S. Szalay, J. Raddick, B. Boroski, S. Lebedeva, and B. Yanny, "Skyserver traffic report - the first five years," 2007. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0701173
[15] J. G. Stadel, "Cosmological N-body simulations and their analysis," Ph.D. dissertation, University of Washington, 2001.
[16] "The NSF TeraGrid," http://www.teragrid.org.
[17] "TIPSY: A Theoretical Image Processing SYstem," http://hpcc.astro.washington.edu/tools/tipsy/tipsy.html.
[18] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White, "The evolution of large-scale structure in a universe dominated by cold dark matt er," *Astroph J*, vol. 292, pp. 371–394, May 1985.
[19] D. H. Weinberg, L. Hernquist, and N. Katz, "Photoionization, Numerical Resolution, and Galaxy Formation," *Astroph. J.*, vol. 477, pp. 8–+, Mar. 1997.
[20] "An Overview of SKID," http://www-hpcc.astro.washington.edu/tools/skid.html.
[21] S. R. Knollmann and A. Knebe, "AHF: Amiga's Halo Finder," *Astroph. J. Suppl.*, vol. 182, pp. 608–624, June 2009.
[22] "IDL - Data Visualization Solutions," http://www.ittvis.com/ProductServices/IDL.aspx.
[23] J. P. Gardner, A. Connolly, and C. McBride, "Enabling rapid development of parallel tree search applications," in *Proceedings of the 2007 Symposium on Challenges of Large Applications in Distri buted Environments (CLADE 2007)*. ACM Press, 2007.
[24] ——, "Enabling knowledge discovery in a virtual universe," in *Proceedings of TeraGrid '07: Broadening Participation in the TeraGrid*. ACM Press, 2007.
[25] "The TPC-H benchmark," http://www.tpc.org.
[26] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The complete book*, 2nd ed. Prentice Hall, 2009.
[27] C. Ballinger, "Born to be parallel: Why parallel origins give Teradata database an enduring performance edge," http://www.teradata.com/t/page/87083/index.html.
[28] "Vertica, inc." http://www.vertica.com/.
[29] "IBM zSeries SYSPLEX," http://publib.boulder.ibm.com/infocenter/\\dzichelp/v2r2/index.jsp?topic=/com.ibm.db2.doc.admin/xf6495.htm.
[30] A. Pruscino, "Oracle RAC: Architecture and performance," in *Proc. of the SIGMOD Conf.*, 2003, p. 635.
[31] Pavlo A. et. al., "A comparison of approaches to large-scale data analysis," in *Proc. of the SIGMOD Conf.*, 2009.
[32] M. Ivanova, M. L. Kersten, N. Nes, and R. Goncalves, "An Architecture for Recycling Intermediates in a Column-store," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Providence, RI, USA, June 2009, accepted for publication.
[33] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, a highly parallel database system," *IEEE TKDE*, vol. 2, no. 1, pp. 4–24, 1990.
[34] L. Chen, C. Olston, and R. Ramakrishnan, "Parallel evaluation of composite aggregate queries," in *Proc. of the 24th International Conference on Data Engineering (ICDE)*, 2008.
[35] A. Deshpande and L. Hellerstein, "Flow algorithms for parallel query optimization," in *Proc. of the 24th International Conference on Data Engineering (ICDE)*, 2008.
[36] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.
[37] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The Gamma database machine project," *IEEE TKDE*, vol. 2, no. 1, pp. 44–62, 1990.
[38] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," *SIGMOD Record*, vol. 19, no. 2, pp. 102–111, 1990.
[39] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *Proc. of the SIGMOD Conf.*, 2008, pp. 1043–1052.
[40] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proc. of the SIGMOD Conf.*, 2007, pp. 1029–1040.
[41] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov, "Clustera: an integrated computation and data management system," in *Proc. of the 34th International Conference on Very Large DataBases (VLDB)*, 2008, pp. 28–41.
[42] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," in *Proc. of the 34th International Conference on Very Large DataBases (VLDB)*, 2008, pp. 1265–1276.
[43] O. O'Malley and A. C. Murthy, "Winning a 60 second dash with a yellow elephant," http://developer.yahoo.net/blogs/hadoop/Yahoo2009.pdf, 2009.
[44] "PigMix," http://wiki.apache.org/pig/PigMix.
[45] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," in *Proc. of the 21st SSDBM Conf.*, 2009.
[46] "Sloan Digital Sky Survey," http://cas.sdss.org.
[47] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon S3 for science grids: a viable solution?" in *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, 2008, pp. 55–64.
[48] "Amazon Simple Storage Service (Amazon S3)," http://www.amazon.com/gp/browse.html?node=16427261.
[49] "Large Synoptic Survey Telescope," http://www.lsst.org/.
[50] "Rasdaman: The intelligent RasterServer," http://www.rasdaman.com/.