MAD Skills: New Analysis Practices for Big Data

Jeffrey Cohen
Greenplum

Brian Dolan
Fox Audience Network

Mark Dunlap
Evergreen Technologies

Joseph M. Hellerstein
U.C. Berkeley

Caleb Welton
Greenplum

ABSTRACT

As massive data acquisition and storage becomes increasingly affordable, a wide variety of enterprises are employing statisticians to engage in sophisticated data analysis. In this paper we highlight the emerging practice of Magnetic, Agile, Deep (MAD) data analysis as a radical departure from traditional Enterprise Data Warehouses and Business Intelligence. We present our design philosophy, techniques and experience providing MAD analytics for one of the world's largest advertising networks at Fox Audience Network, using the Greenplum parallel database system. We describe database design methodologies that support the agile working style of analysts in these settings. We present data-parallel algorithms for sophisticated statistical techniques, with a focus on *density* methods. Finally, we reflect on database system features that enable agile design and flexible algorithm development using both SQL and MapReduce interfaces over a variety of storage mechanisms.

1. INTRODUCTION

If you are looking for a career where your services will be in high demand, you should find something where you provide a scarce, complementary service to something that is getting ubiquitous and cheap. So what's getting ubiquitous and cheap? Data. And what is complementary to data? Analysis.
– Prof. Hal Varian, UC Berkeley, Chief Economist at Google [5]

mad (adj.): an adjective used to enhance a noun.
1- dude, you got skills.
2- dude, you got mad skills.
– UrbanDictionary.com [12]

Standard business practices for large-scale data analysis center on the notion of an "Enterprise Data Warehouse" (EDW) that is queried by "Business Intelligence" (BI) software. BI tools produce reports and interactive interfaces that summarize data via basic aggregation functions (e.g., counts and averages) over various hierarchical breakdowns of the data into groups. This was the topic of significant academic research and industrial development throughout the 1990's.

Traditionally, a carefully designed EDW is considered to have a central role in good IT practice. The design and evolution of a comprehensive EDW schema serves as the rallying point for disciplined data integration within a large enterprise, rationalizing the outputs and representations of all business processes. The resulting database serves as the repository of record for critical business functions. In addition, the database server storing the EDW has traditionally been a major computational asset, serving as the central, scalable engine for key enterprise analytics. The conceptual and computational centrality of the EDW makes it a mission-critical, expensive resource, used for serving data-intensive reports targeted at executive decision-makers. It is traditionally controlled by a dedicated IT staff that not only maintains the system, but jealously controls access to ensure that executives can rely on a high quality of service. [13]

While this orthodox EDW approach continues today in many settings, a number of factors are pushing towards a very different philosophy for large-scale data management in the enterprise. First, storage is now so cheap that small subgroups within an enterprise can develop an isolated database of astonishing scale within their discretionary budget. The world's largest data warehouse from just over a decade ago can be stored on less than 20 commodity disks priced at under $100 today. A department can pay for 1-2 orders of magnitude more storage than that without coordinating with management. Meanwhile, the number of massive-scale data sources in an enterprise has grown remarkably: massive databases arise today even from single sources like clickstreams, software logs, email and discussion forum archives, etc. Finally, the value of data analysis has entered common culture, with numerous companies showing how sophisticated data analysis leads to cost savings and even direct revenue. The end result of these opportunities is a grassroots move to collect and leverage data in multiple organizational units. While this has many benefits in fostering efficiency and data-driven culture [15], it adds to the force of data decentralization that data warehousing is supposed to combat.

In this changed climate of widespread, large-scale data collection, there is a premium on what we dub *MAD* analysis skills. The acronym arises from three aspects of this environment that differ from EDW orthodoxy:

- **Magnetic:** Traditional EDW approaches "repel" new data sources, discouraging their incorporation until they are carefully cleansed and integrated. Given the ubiquity of data in modern organizations, a data ware-

VLDB '09, August 24-28, 2009, Lyon, France

house can keep pace today only by being "magnetic": attracting all the data sources that crop up within an organization regardless of data quality niceties.

- **Agile:** Data Warehousing orthodoxy is based on long-range, careful design and planning. Given growing numbers of data sources and increasingly sophisticated and mission-critical data analyses, a modern warehouse must instead allow analysts to easily ingest, digest, produce and adapt data at a rapid pace. This requires a database whose physical and logical contents can be in continuous rapid evolution.
- **Deep:** Modern data analyses involve increasingly sophisticated statistical methods that go well beyond the rollups and drilldowns of traditional BI. Moreover, analysts often need to see both the forest and the trees in running these algorithms – they want to study enormous datasets without resorting to samples and extracts. The modern data warehouse should serve both as a deep data repository and as a sophisticated algorithmic runtime engine.

As noted by Varian, there is a growing premium on analysts with MAD skills in data analysis. These are often highly trained statisticians, who may have strong software skills but would typically rather focus on deep data analysis than database management. They need to be complemented by MAD approaches to data warehouse design and database system infrastructure. These goals raise interesting challenges that are different than the traditional focus in the data warehousing research and industry.

1.1 Contributions

In this paper, we describe techniques and experiences we have developed in our development of MAD analytics for Fox Audience Network, using a large installation of the Greenplum Database system. We discuss our database design methodology that focuses on enabling an agile yet organized approach to data analysis (Section 4). We present a number of data-parallel statistical algorithms developed for this setting, which focus on modeling and comparing the *densities* of distributions. These include specific methods like Ordinary Least Squares, Conjugate Gradiant, and Mann-Whitney U Testing, as well as general purpose techniques like matrix multiplication and Bootstrapping (Section 5). Finally, we reflect on critical database system features that enable agile design and flexible algorithm development, including high-performance data ingress/egress, heterogeneous storage facilities, and flexible programming via both extensible SQL and MapReduce interfaces to a single system (Section 6).

Underlying our discussion are challenges to a number of points of conventional wisdom. In designing and analyzing data warehouses, we advocate the theme "Model Less, Iterate More". This challenges data warehousing orthodoxy, and argues for a shift in the locus of power from DBAs toward analysts. We describe the need for unified systems that embrace and integrate a wide variety of data-intensive programming styles, since analysts come from many walks of life. This involves moving beyond religious debates about the advantages of SQL over MapReduce, or R over Java, to focus on evolving a single parallel dataflow engine that can support a diversity of programming styles, to tackle substantive statistical analytics. Finally, we argue that many data sources and storage formats can and should be knitted together by the parallel dataflow engine. This points toward more fluid integration or consolidation of traditionally diverse tools including traditional relational databases, column stores, ETL tools, and distributed file systems.

2. BACKGROUND: IF YOU'RE NOT MAD

Data analytics is not a new area. In this section we describe standard practice and related work in Business Intelligence and large-scale data analysis, to set the context for our MAD approach.

2.1 OLAP and Data Cubes

Data Cubes and On-Line Analytic Processing (OLAP) were popularized in the 1990's, leading to intense commercial development and significant academic research. The SQL `CUBE BY` extension translated the basic idea of OLAP into a relational setting [8]. BI tools package these summaries into fairly intuitive "cross-tabs" visualizations. By grouping on few dimensions, the analyst sees a coarse "roll-up" bar-chart; by grouping on more dimensions they "drill down" into finer grained detail. Statisticians use the phrase *descriptive* statistics for this kind of analysis, and traditionally apply the approach to the results of an experimental study. This functionality is useful for gaining intuition about a process underlying the experiment. For example, by describing the clickstream at a website one can get better inuition about underlying properties of the user population.

By contrast, *inferential* or *inductive* statistics try to directly capture the underlying properties of the population. This includes fitting models and parameters to data and computing likelihood functions. Inferential statistics require more computation than the simple summaries provided by OLAP, but provide more probabilistic power that can be used for tasks like prediction (e.g., "which users would be likely to click on this new ad?"), causality analysis ("what features of a page result in user revisits?"), and distributional comparison (e.g., "how do the buying patterns of truck owners differ from sedan owners?") Inferential approaches are also more robust to outliers and other particulars of a given dataset. While OLAP and Data Cubes remain useful for intuition, the use of inferential statistics has become an imperative in many important automated or semi-automated business processes today, including ad placement, website optimization, and customer relationship management.

2.2 Databases and Statistical Packages

BI tools provide fairly limited statistical functionality. It is therefore standard practice in many organizations to extract portions of a database into desktop software packages: statistical package like SAS, Matlab or R, spreadsheets like Excel, or custom code written in languages like Java.

There are various problems with this approach. First, copying out a large database extract is often much less efficient than pushing computation to the data; it is easy to get orders of magnitude performance gains by running code in the database. Second, most stat packages require their data to fit in RAM. For large datasets, this means sampling the database to form an extract, which loses detail. In modern settings like advertising placement, *microtargeting* requires an understanding of even small subpopulations. Samples and synopses can lose the "long tail" in a data set, and that is increasingly where the competition for effectiveness lies.

A better approach is to tightly integrate statistical computation with a massively parallel database. Unfortunately, many current statistical packages do not provide parallel implementations of any kind. Those statistical libraries that have been parallelized, e.g., via ScaLAPACK [2], rely on MPI-based message-passing protocols across processors, and do not integrate naturally with the dataflow parallelism of popular data-intensive solutions.

2.3 MapReduce and Parallel Programming

While BI and EDW methodologies were being packaged for enterprise settings, the MapReduce programming model popularized by Google has captured the attention of many developers. Google's very visible successes in ad placement and text processing – and their public embrace of statistical machine learning – has driven this phenomenon forward quickly. A recent paper on implementing machine learning algorithms in MapReduce [3] highlighted a number of standard techniques that can be computed in a data-parallel fashion via summations. The Apache Mahout project is an effort to implement these techniques in the open-source Hadoop MapReduce engine. The observation of that paper applies equally well to SQL, but the technical-social phenomenon surrounding MapReduce is important: it has caused a number of statistically-minded researchers and developers to focus on Big Data and data-parallel programming, rather than on multiprocessing via MPI. This spirit of data-parallel programming informs the design of our algorithms in Section 5 as well. But programming style is only one aspect of a MAD approach to managing the process of analytics, as we describe in Section 4.

2.4 Data Mining and Analytics in the Database

There is a significant literature on parallel data mining algorithms; see for example the collection by Zaki and Ho [24]. The most common data mining techniques – clustering, classification, association rules – concern themselves with what we might call *pointwise decisions*. Classifiers and clustering assign individual points to cohorts (class labels or cluster IDs); association rules form combinatorial collections of individual points at the output. Though these problems are non-trivial, the field of statistical modeling covers quite a bit more ground in addition. For example, a common technique in advertising analysis is *A/B testing*, which takes response rates of a subpopulation and a control group, and compares their statistical densities on various metrics.

Standard data mining methods in commercial databases are useful but quite targeted: they correspond to only a small number of the hundreds of statistical libraries that would ship with a stat package like R, SAS, or Matlab. Moreover, they are typically "black box" implementations, whose code is compiled into an engine plugin. By contrast, statistical package like R or Matlab are flexible programming environments, with library routines that can be modified and extended by analysts. MAD analysis requires similar programming abilities to be brought to Big Data scenarios, via extensible SQL and/or MapReduce. Black-box data mining routines can sometimes be useful in that context, but only in a modest fraction of cases.

In addition to our work, there are other interesting efforts to do significant scientific computation in SQL documented in the literature, most notably in the Sloan Digital Sky Survey [22]. The management of experiments [14] and complex SQL code [16] are also relevant. The interested reader is also referred to new research systems on scientific data management [21] and scalability for R [25]. The idea of "dataspaces" is related to our MAD philosophy as well, with a focus on data integration [6].

3. FOX AUDIENCE NETWORK

Fox Audience Network serves ads across several Fox online publishers, including MySpace.com, IGN.com and Scout.com. With a reach of over one hundred and fifty million users, it is one of the world's largest ad networks.

The FAN Data Warehouse today is implemented via the Greenplum Database system running on 42 nodes: 40 for query processing, and two master nodes (one for failover). The query processing nodes are Sun X4500 ("Thumper") machines, configured with 2 dual-core Opteron processors, 48 500-GB drives, and 16 GB of RAM. The FAN EDW currently holds over 200 terabytes of unique production data that is then mirrored for failover. It is growing rapidly: every day the FAN data warehouse ingests four to seven billion rows of ad server logs amounting to approximately five terabytes. The major impression fact table stretches back to October of 2007, creating a single table of over 1.5 trillion rows. FAN's Customer Relationship Management (CRM) solution also provides millions of rows of advertiser and campaign dimension data every day. Additionally there is extensive data on each of over 150 million users. The FAN EDW is the sole repository where all three data sources are integrated for use by research or reporting teams.

The EDW is expected to support very disparate users, from sales account managers to research scientists. These users' needs are very different, and a variety of reporting and statistical software tools are leveraged against the warehouse every day. The MicroStrategy BI tool has been implemented directly against the warehouse to service Sales, Marketing and most other basic reporting needs. Research scientists also have direct command-line access to the same warehouse. Thus, the query ecosystem is very dynamic.

No set of pre-defined aggregates could possibly cover every question. For example, it is easy to imagine questions that combine both advertiser and user variables. At FAN, this typically means hitting the fact tables directly. For instance, one question that is easy for a salesperson to pose is: *How many female WWF enthusiasts under the age of 30 visited the Toyota community over the last four days and saw a medium rectangle?* (A "medium rectangle" is a standard-sized web ad.) The goal is to provide answers within minutes to such ad hoc questions, and it is not acceptable to refuse to answer questions that were not precomputed.

This example is satisfied by a simple SQL query, but the follow-up question in such scenarios is invariably a comparative one: *How are these people similar to those that visited Nissan?* At this stage we begin to engage in open-ended multi-dimensional statistical analysis requiring more sophisticated methods. At FAN, *R* is a popular choice for research and the *RODBC* package is often used to interface directly with the warehouse. When these questions can be reduced to fewer than 5 million records, the data is often exported and analyzed in *R*. The cases where it cannot be reduced form a main impetus for this research paper.

In addition to the data warehouse, the machine learning team at FAN also makes use of several large Hadoop clusters. That team employs dozens of algorithms for classification, supervised and unsupervised learning, neural network and natural language processing. These are not techniques traditionally addressed by an RDBMS, and the implementation in Hadoop results in large data migration efforts for specific single-purpose studies. The availability of machine learning methods directly within the warehouse would offer a significant savings in time, training, and system management, and is one of the goals of the work described here.

4. MAD DATABASE DESIGN

Traditional Data Warehouse philosophy revolves around a disciplined approach to modeling information and processes in an enterprise. In the words of warehousing advocate Bill Inmon, it is an "architected environment" [13]. This view of warehousing is at odds with the magnetism and agility desired in many new analysis settings, as we describe below.

4.1 New Requirements

As data savvy people, analysts introduce a new set of requirements to a database environment. They have a deep understanding of the enterprise data and tend to be early adopters of new data sources. In the same way that systems engineers always want the latest-and-greatest hardware technologies, analysts are always hungry for new sources of data. When new data-generating business processes are launched, analysts demand the new data immediately.

These desires for speed and breadth of data raise tensions with Data Warehousing orthodoxy. Inmon describes the traditional view:

> There is no point in bringing data ... into the data warehouse environment without integrating it. If the data arrives at the data warehouse in an unintegrated state, it cannot be used to support a corporate view of data. And a corporate view of data is one of the essences of the architected environment. [13]

Unfortunately, the challenge of perfectly integrating a new data source into an "architected" warehouse is often substantial, and can hold up access to data for months – or in many cases, forever. The architectural view introduces friction into analytics, repels data sources from the warehouse, and as a result produces shallow incomplete warehouses. It is the opposite of the MAD ideal.

Given the growing sophistication of analysts and the growing value of analytics, we take the view that it is much more important to provide agility to analysts than to aspire to an elusive ideal of full integration. In fact, analysts serve as key data magnets in an organization, scouting for interesting data that should be part of the corporate big picture. They can also act as an early warning system for data quality issues. For the privilege of being the first to see the data, they are more tolerant of dirty data, and will act to apply pressure on operational data producers upstream of the warehouse to rectify the data before it arrives. Analysts typically have much higher standards for the data than a typical business unit working with BI tools. They are undaunted by big, flat fact tables that hold complete data sets, scorning samples and aggregates, which can both mask errors and lose important features in the tails of distributions.

Hence it is our experience that a good relationship with the analytics team is an excellent preventative measure for data management issues later on. Feeding their appetites and responding to their concerns improves the overall health of the warehouse.

Ultimately, the analysts produce new data products that are valuable to the enterprise. They are not just consumers, but producers of enterprise data. This requires the warehouse to be prepared to "productionalize" the data generated by the analysts into standard business reporting tools.

It is also useful, when possible, to leverage a single parallel computing platform, and push as much functionality as possible into it. This lowers the cost of operations, and eases the evolution of software from analyst experiments to production code that affects operational behavior. For example, the lifecycle of an ad placement algorithm might start in a speculative analytic task, and end as a customer facing feature. If it is a data-driven feature, it is best to have that entire lifecycle focused in a single development environment on the full enterprise dataset. In this respect we agree with a central tenet of Data Warehouse orthodoxy: there is tangible benefit to getting an organization's data into one repository. We differ on how to achieve that goal in a useful and sophisticated manner.

In sum, a healthy business should not assume an architected data warehouse, but rather an evolving structure that iterates through a continuing cycle of change:

1. The business performs analytics to identify areas of potential improvement.
2. The business either reacts to or ignores this analysis.
3. A reaction results in new or different business practices – perhaps new processes or operational systems – that typically generate new data sets.
4. Analysts incorporate new data sets into their models.
5. The business again asks itself "How can we improve?"

A healthy, competitive business will look to increase the pace of this cycle. The MAD approach we describe next is a design pattern for keeping up with that increasing pace.

4.2 Getting More MAD

The central philosophy in MAD data modeling is to get the organization's data into the warehouse as soon as possible. Secondary to that goal, the cleaning and integration of the data should be staged intelligently.

To turn these themes into practice, we advocate the three-layer approach. A *Staging* schema should be used when loading raw fact tables or logs. Only engineers and some analysts are permitted to manipulate data in this schema. The *Production Data Warehouse* schema holds the aggregates that serve most users. More sophisticated users comfortable in a large-scale SQL environment are given access to this schema. A separate *Reporting* schema is maintained to hold specialized, static aggregates that support reporting tools and casual users. It should be tuned to provide rapid access to modest amounts of data.

These three layers are not physically separated. Users with the correct permissions are able to cross-join between layers and schemas. In the FAN model, the Staging schema holds raw action logs. Analysts are given access to these logs for research purposes and to encourage a laboratory approach to data analysis. Questions that start at the event log level often become broader in scope, allowing custom aggregates. Communication between the researchers and the DBAs uncovers common questions and often results in aggregates that were originally personalized for an analyst being promoted into the production schema.

The Production schema provides quick answers to common questions that are not yet so common as to need reports. Many of these are anticipated during installation, but many are not. Data tends to create "feeding frenzies" as organizations, starved for data only months before, begin launching question after question at the database. Being nimble in this stage is crucial, as the business analysts begin to enter the environment. They want to know things at the daily or monthly level. Questions about daily performance are turned into dashboards.

Analysts should also be given a fourth class of schema within the warehouse, which we call a "sandbox"[1]. The sandbox schema is under the analysts' full control, and is to be used for managing their experimental processes. Analysts are data-savvy developers, and they often want to track and record their work and work products in a database. For example, when developing complex SQL like we will see in Section 5, it is typical to use SQL views as "subroutines" to structure a coding task. During development, these views are likely to be defined and edited frequently. Similarly, analysts may want to materialize query results as they are doing their work, and reuse the results later; this materialization can also help the software engineering process, and improve efficiency during iterative design of an analysis workflow. In addition, analysts often want to squirrel away pet data sets for their own convenience over time, to use in prototyping new techniques on known inputs.

The ability to leap from very specific to very general encourages investigation and creativity. As the data gets used, transformed, discussed and adopted, the organization learns and changes its practices. The pace of progress is bound by the pace and depth of investigation. MAD design is intended to accelerate this pace.

5. DATA-PARALLEL STATISTICS

Analysts and statisticians are an organization's most data-savvy agents, and hence key to its MAD skills. In this section, we focus on powerful, general statistical methods that make the data warehouse more "Magnetic" and "Agile" for these analysts, encouraging them to go "Deep" and substantially increase the sophistication and scale of data analytics.

Our general approach is to develop a hierarchy of mathematical concepts in SQL, and encapsulate them in a way that allows analysts to work in relatively familiar statistical terminology, without having to develop statistical methods in SQL from first principles for each computation. Similar functionality could be coded up in a MapReduce syntax.

Traditional SQL databases provide data types and functions for simple (scalar) arithmetic. The next layer of abstraction is *vector arithmetic*, which brings its own suite of operators. Vector objects combined with vector operators bring us the language of *linear algebra.* We suggest methods for these operators in Section 5.1. It is this level that allows us to speak in the language of machine learning, mathematical modeling, and statistics. The next layer of abstraction is the *function* level; probability densities are specialized functions. Inductively, there is another layer of abstraction, where functions are the base objects and algebras are created using operators called "functionals" acting upon functions. This is the realm of functional analysis. Methods like $t-$tests or likelihood ratios are functionals. A/B testing involves functionals, treating two mathematical objects at a time: probability density functions $f_1(\cdot)$ and $f_2(\cdot)$.

Our job, then, is to advance database methods from scalars to vectors to functions to functionals. Further, we must do this in a massively parallel environment. This is not trivial. Even the apparently "simple" problem of representing matrices does not have a unique optimal solution. In the next few sections, we describe methods we have used to convince a parallel database to behave like a massively scalable statistical package. We begin with vector arithmetic and work toward functionals, with additional powerful statistical methods along the way.

5.1 Vectors and Matrices

Relational databases are designed to scale with cardinality. We describe how we have represented large "vector" and "matrix" objects as relations, and implemented basic operations for them. This gives us vector arithmetic.

Before constructing operators, we need to define what a vector (often in the form of a matrix) would mean in the context of a database. There are many ways to partition ("block", "chunk") such matrices across nodes in a parallel system (e.g., see [2], Chapter 4). A simple scheme that fits well with parallel databases is to represent the matrix as a relation with schema `(row_number integer, vector numeric[])` and allow the DBMS to partition the rows across processors arbitrarily – e.g. via hashing or a round-robin scheme. In practice, we often materialize both A and A' to allow this row-representation method to work more efficiently.

Given matrices represented as horizontally partitioned relations in this manner, we need to implement basic matrix arithmetic as queries over those relations, which can be executed in a parallel fashion.

Consider two matrices A and B of identical dimensions. Matrix addition $A + B$ is easy to express in SQL:

```
SELECT A.row_number, A.vector + B.vector
  FROM A, B
 WHERE A.row_number = B.row_number;
```

Note that the `+` operator here is operating on arrays of numeric types and returns a similarly-dimensioned array, so the output is a matrix whose dimensions are equal to the inputs. If vector addition is not provided by the DBMS, it is very easy to implement via object-relational extensions and register as an infix operator [19]. A query optimizer is likely to choose a hash join for this query, which parallelizes well.

Multiplication of a matrix and a vector Av is also simple:

```
SELECT 1, array_accum(row_number, vector*v) FROM A;
```

Again, the `*` operator here is operating on arrays of numerics, but in this case returns a single numeric value – the dot product of its inputs $\vec{x} \cdot \vec{y} = \Sigma_i x_i y_i$. This too can be implemented via a user-defined function, and registered as an infix operator with the query language [19]. The pairs `(row_number,vector*v)` represent a vector as (index, value) pairs. To get back to our canonical row-wise representation, we convert to an array type in a single output row via the custom aggregation function `array_accum(x,v)`, which accumulates a single array field, setting position `x` to value `v` for each row of input.

[1] This is not to be confused with "sandboxing" software processes for computer security. Our usage is intended to convey a sense of play.

Most RDBMSs have sequence functions. In PostgreSQL and Greenplum, the command `generate_series(1, 50)` will generate $1, 2, \ldots 50$. One way to compute a matrix transpose A' of an $m \times n$ matrix A is expressible as follows (for $n = 3$):

```
SELECT S.col_number,
       array_accum(A.row_number, A.vector[S.col_number])
  FROM A, generate_series(1,3) AS S(col_number)
 GROUP BY S.col_number;
```

Unfortunately if `A` stores n-dimensional vectors, then this results in up to n copies of the table `A` being fed into the grouping operator. An alternative is to convert to a different matrix representation, for example a sparse representation of the form `(row_number, column_number, value)`. An advantage to this approach is that the SQL is much easier to construct for multiplication of matrices AB.

```
SELECT A.row_number, B.column_number, SUM(A.value * B.value)
FROM A, B
WHERE A.column_number = B.row_number
GROUP BY A.row_number, B.column_number
```

This query is very efficient on sparse matrices, as 0 values would not be stored. In general, it is well-known in parallel computation that no single storage representation will service all needs, and in practice blends need to be used. Without proper abstractions, this can lead to confusion as custom operators need to be defined for each representation. In SQL, different representations can be achieved via naming conventions on (materialized) views, but the choice of views typically is done by the analyst, since a traditional query optimizer is unaware of the equivalence of these views. Work on this front can be found in the parallel computing literature [23], but has yet to be integrated with relational query optimization and data-intensive computing.

The above conversation applies to scalar multiplication, vector addition and vector/matrix multiplication, which are essentially single-pass methods. The task of matrix division is not definitively solved in a parallel context. One awkwardness in SQL is the lack of convenient syntax for iteration. The fundamental routines for finding a matrix inverse involve two or more passes over the data. However, recursive or iterative procedures can be driven by external processes with a minimal amount of data flow over the master node. For instance, in the conjugate gradient method described in Section 5.2.2, only a single value is queried between iterations. Although matrix division is complicated, we are able to develop the rest of our methods in this paper via pseudoinverse routines (with textbook math programming caveats on existence and convergence.)

A comprehensive suite of (now distributed) vector objects and their operators generate the nouns and verbs of statistical mathematics. From there, functions follow as sentences. We continue with a familiar example, cast in this language.

5.1.1 tf-idf and Cosine Similarity

The introduction of vector operations allows us to speak much more compactly about methods. We consider a specific example here: document similarity, a common tool in web advertising. One usage is in fraud detection. When many different advertisers link to pages that are very similar, it is typical that they are actually the same malicious party, and very likely using stolen credit cards for payment. It is therefore wise to follow advertisers' outlinks and look for patterns of similar documents.

The common document similarity metric tf-idf involves three or four steps, all of which can be easily distributed and lend themselves very well to SQL methods. First, triples of *(document, term, count)* must be created. Then marginals along *document* and *term* are calculated, via simple SQL `GROUP BY` queries. Next each original triple can be expanded with a tf-idf score along every "dimension" – i.e., for every word in the resulting dictionary – by joining the triples with the *document* marginals and dividing out the document counts from the term counts. From there, the cosine similarity of two document vectors is calculated on tf-idf scores and the standard "distance" metric is obtained.

Specifically, it is well known that given two term-weight vectors x and y, the cosine similarity θ is given by $\theta = \frac{x \cdot y}{\|x\|^2 \|y\|^2}$. It is not difficult to construct "vanilla" SQL to reproduce this equation. But analysts with backgrounds in statistics do not think that way – this approach (and the sparse matrix example of Section 5.1) focuses on pairing up scalar values, rather than higher-level "whole-object" reasoning on vectors. Being able to express ideas like tf-idf in terms of linear algebra lowers the barrier to entry for statisticians to program against the database. The dot-product operator reduces tf-idf to a very natural syntax. Suppose `A` has one row per document vector.

```
SELECT a1.row_id AS document_i, a2.row_id AS document_j,
  (a1.row_v * a2.row_v) /
    ((a1.row_v * a1.row_v) * (a2.row_v * a2.row_v)) AS theta
FROM a AS a1, a AS a2
WHERE a1.row_id > a2.row_id
```

To any analyst comfortable with scripting environments such as SAS or `R`, this formation is perfectly acceptable. Further, the DBA has done the work of distributing the objects and defining the operators. When the objects and operators become more complicated, the advantages to having predefined operators increases. From a practical standpoint, we have moved the database closer to an interactive, analytic programming environment and away from a simple data retrieval system.

5.2 Matrix Based Analytical Methods

The matrices of primary interest in our setting are large, dense matrices. A common subject is a distance matrix D where $D(i, j) > 0$ for almost all (i, j). Another theme is covariance matrices Σ in tightly correlated data sets.

5.2.1 Ordinary Least Squares

We begin with Ordinary Least Squares (OLS), a classical method for fitting a curve to data, typically with a polynomial function. In web advertising, one standard application is in modeling seasonal trends. More generally, in many ad-hoc explorations it is where analysis starts. Given a few simple vector-oriented user-defined functions, it becomes natural for an analyst to express OLS in SQL.

In our case here we find a statistical estimate of the parameter β^* best satisfying $Y = X\beta$. Here, $X = n \times k$ is a set of fixed (independent) variables, and Y is a set of n observations (dependent variables) that we wish to model via a function of X with parameters β.

As noted in [3],

$$\beta^* = (X'X)^{-1}X'y \qquad (1)$$

can be calculated by computing $A = X'X$ and $b = X'y$ as summations. In a parallel database this can be executed by having each partition of the database calculate the local A and b in parallel and then merge these intermediate results in a final sequential calculation.

This will produce a square matrix and a vector based on the size of the independent variable vector. The final calculation is computed by inverting the small A matrix and multiplying by the vector to derive the coefficients β^*.

Additionally, calculation of the coefficient of determination R^2 can be calculated concurrently by

$$\begin{aligned} SSR &= b'\beta^* - \frac{1}{n}\left(\sum y_i\right)^2 \\ TSS &= \sum y_i^2 - \frac{1}{n}\left(\sum y_i\right)^2 \\ R^2 &= \frac{SSR}{TSS} \end{aligned}$$

In the following SQL query, we compute the coefficients β^*, as well as the components of the coefficient of determination:

```
CREATE VIEW ols AS
SELECT pseudo_inverse(A) * b as beta_star,
  (transpose(b) * (pseudo_inverse(A) * b)
    - sum_y2/count)                             -- SSR
  / (sum_yy - sumy2/n)                          -- TSS
      as r_squared
FROM (
  SELECT sum(transpose(d.vector) * d.vector) as A,
         sum(d.vector * y) as b,
         sum(y)^2 as sum_y2, sum(y^2) as sum_yy,
         count(*) as n
  FROM design d
) ols_aggs;
```

Note the use of a user-defined function for vector transpose, and user-defined aggregates for summation of (multidimensional) array objects. The array `A` is a small in-memory matrix that we treat as a single object; the `pseudo-inverse` function implements the textbook Moore-Penrose pseudoinverse of the matrix.

All of the above can be efficiently calculated in a single pass of the data. For convenience, we encapsulated this yet further via two user-defined aggregate functions:

```
SELECT ols_coef(d.y, d.vector), ols_r2(d.y, d.vector)
FROM design d;
```

Prior to the implementation of this functionality within the DBMS, one Greenplum customer was accustomed to calculating the OLS by exporting data and importing the data into R for calculation, a process that took several hours to complete. They reported significant performance improvement when they moved to running the regression within the DBMS. Most of the benefit derived from running the analysis in parallel close to the data with minimal data movement.

5.2.2 Conjugate Gradient

In this subsection we develop a data-parallel implementation of the Conjugate Gradiant method for solving a system of linear equations. We can use this to implement Support Vector Machines, a state-of-the-art technique for binary classification. Binary classifiers are a common tool in modern ad placement, used to turn complex multi-dimensional user features into simple boolean labels like "is a car enthusiast" that can be combined into *enthusiast charts*. In addition to serving as a building block for SVMs, the Conjugate Gradiant method allows us to optimize a large class of functions that can be approximated by second order Taylor expansions.

To a mathematician, the solution to the matrix equation $Ax = b$ is simple when it exists: $x = A^{-1}b$. As noted in Section 5.1, we cannot assume we can find A^{-1}. If matrix A is $n \times n$ symmetric and positive definite (SPD), we can use the Conjugate Gradient method. This method requires neither $df(y)$ nor A^{-1} and converges in no more than n interations. A general treatment is given in [17]. Here we outline the solution to $Ax = b$ as an extremum of $f(x) = \frac{1}{2}x'Ax + b'x + c$. Broadly, we have an estimate $\hat{x}$ to our solution x^*. Since $\hat{x}$ is only an estimate, $r_0 = A\hat{x} - b$ is non-zero. Subtracting this error r_0 from the estimate allows us to generate a series $p_i = r_{i-1} - \{A\hat{x} - b\}$ of orthogonal vectors. The solution will be $x^* = \sum_i \alpha_i p_i$ for α_i defined below. We end at the point $\|r_k\|^2 < \epsilon$ for a suitable ϵ. There are several update algorithms, we have written ours in matrix notation.

$$\begin{array}{ll} r_0 = b - A\hat{x}_0, & \alpha_0 = \frac{r_0' r_0}{v_0' A v_0}, \\ v_0 = r_0, & i = 0 \end{array}$$

Begin iteration over i.

$$\begin{aligned} \alpha_i &= \frac{r_i' r_i}{v_i' A v_i} \\ x_{i+1} &= x_i + \alpha_i v_i \\ r_{i+1} &= r_i - \alpha_i A v_i \\ & \quad \text{check } \|r_{i+1}\|^2 \leq \epsilon \\ v_{i+1} &= r_{i+1} + \frac{r_{i+1}' r_{i+1}}{r_i' r_i} v_i \end{aligned}$$

To incorporate this method into the database, we stored $(v_i, x_i, r_i, \alpha_i)$ as a row and inserted row $i+1$ in one pass. This required the construction of functions `update_alpha(r_i, p_i, A)`, `update_x(x_i, alpha_i, v_i)`, `update_r(x_i, alpha_i, v_i, A)`, and `update_v(r_i, alpha_i, v_i, A)`. Though the function calls were redundant (for instance, `update_v()` also runs the update of r_{i+1}), this allowed us to insert one full row at a time. An external driver process then checks the value of r_i before proceeding. Upon convergence, it is rudimentary to compute x^*.

The presence of the conjugate gradient method enables even more sophisticated techniques like Support Vector Machines (SVM). At their core, SVMs seek to maximize the distance between a set of points and a candiate hyperplane. This distance is denoted by the magnitude of the normal vectors $\|w\|^2$. Most methods incorporate the integers $\{0, 1\}$ as labels c, so the problem becomes

$$\underset{w,b}{\operatorname{argmax}} \; f(w) = \frac{1}{2}\|w\|^2, \quad \text{subject to } c'w - b \geq 0.$$

This method applies to the more general issue of high dimensional functions under a Taylor expansion $f_{x_0}(x) \approx f(x_0) + df(x)(x - x_0) + \frac{1}{2}(x - x_o)'d^2f(x)(x - x_0)$ With a good initial guess for x_* and the common assumption of continuity of $f(\cdot)$, we know the the matrix will be SPD near x^*. See [17] for details.

5.3 Functionals

Basic statistics are not new to relational databases – most support means, variances and some form of quantiles. But modeling and comparative statistics are not typically built-in functionality. In this section we provide data-parallel implementations of a number of comparative statistics expressed in SQL.

In the previous section, scalars or vectors were the atomic unit. Here a probability density function is the foundational object. For instance the Normal (Gaussian) density $f(x) = e^{(x-\mu)^2/2\sigma^2}$ is considered by mathematicians as a single "entity" with two attributes: the mean μ and variance σ. A common statistical question is to see how well a data set fits to a target density function. The $z-$score of a datum x is given $z(x) = \frac{(x-\mu)}{\sigma/\sqrt{n}}$ and is easy to obtain in standard SQL.

```
SELECT x.value, (x.value - d.mu) * d.n / d.sigma AS z_score
FROM x, design d
```

5.3.1 Mann-Whitney *U* Test

Rank and order statistics are quite amenable to relational treatments, since their main purpose is to evaluate a set of data, rather then one datum at a time. The next example illustrates the notion of comparing two entire sets of data without the overhead of describing a parameterized density.

The Mann-Whitney U Test (MWU) is a popular substitute for Student's t-test in the case of non-parametric data. The general idea it to take two populations A and B and decide if they are from the same underlying population by examining the rank order in which members of A and B show up in a general ordering. The cartoon is that if members of A are at the "front" of the line and members of B are at the "back" of the line, then A and B are different populations. In an advertising setting, click-through rates for web ads tend to defy simple parametric models like Gaussians or log-normal distributions. But it is often useful to compare click-through rate distributions for different ad campaigns, e.g., to choose one with a better median click-through. MWU addresses this task.

Given a table `T` with columns `SAMPLE_ID, VALUE`, row numbers are obtained and summed via SQL windowing functions.

```
CREATE VIEW R AS
SELECT sample_id, avg(value) AS sample_avg
       sum(rown) AS rank_sum, count(*) AS sample_n,
       sum(rown) - count(*) * (count(*) + 1) AS sample_us
  FROM (SELECT sample_id, row_number() OVER
                   (ORDER BY value DESC) AS rown,
               value
          FROM T) AS ordered
 GROUP BY sample_id
```

Assuming the condition of large sample sizes, for instance greater then 5,000, the normal approximation can be justified. Using the previous view `R`, the final reported statistics are given by

```
SELECT r.sample_u, r.sample_avg, r.sample_n
       (r.sample_u - a.sum_u / 2) /
        sqrt(a.sum_u * (a.sum_n + 1) / 12) AS z_score
FROM R as r, (SELECT sum(sample_u) AS sum_u,
                     sum(sample_n) AS sum_n
              FROM R) AS a
GROUP BY r.sample_u, r.sample_avg, r.sample_n,
     a.sum_n, a.sum_u
```

The end result is a small set of numbers that describe a relationship of functions. This simple routine can be encapsulated by stored procedures and made available to the analysts via a simple `SELECT mann_whitney(value) FROM table` call, elevating the vocabulary of the database tremendously.

5.3.2 Log-Likelihood Ratios

Likelihood ratios are useful for comparing a subpopulation to an overall population on a particular attributed. As an example in advertising, consider two attributes of users: beverage choice, and family status. One might want to know whether coffee attracts new parents more than the general population.

This is a case of having two density (or mass) functions for the same data set X. Denote one distribution as null hypothesis f_0 and the other as alternate f_A. Typically, f_0 and f_A are different parameterizations of the same density. For instance, $N(\mu_0, \sigma_0)$ and $N(\mu_A, \sigma_A)$. The likelihood L under f_i is given by

$$L_{f_i} = L(X|f_i) = \prod_k f_i(x_k).$$

The log-likelihood ratio is given by the quotient $-2\log\left(L_{f_0}/L_{f_A}\right)$. Taking the log allows us to use the well-known χ^2 approximation for large n. Also, the products turn nicely into sums and an RDBMS can handle it easily in parallel.

$$LLR = 2\sum_k \log f_A(x_k) - 2\sum_k \log f_0(x_k).$$

This calculation distributes nicely if $f_i : \mathbb{R} \to \mathbb{R}$, which most do. If $f_i : \mathbb{R}^n \to \mathbb{R}$, then care must be taken in managing the vectors as distributed objects. Suppose the values are in table `T` and the function $f_A(\cdot)$ has been written as a user-defined function `f_llk(x numeric, param numeric)`. Then the entire experiment is can be performed with the call

```
SELECT 2 * sum(log(f_llk(T.value, d.alt_param))) -
       2 * sum(log(f_llk(T.value, d.null_param))) AS llr
FROM T, design AS d
```

This represents a significant gain in flexibility and sophistication for any RDBMS.

Example: The Multinomial Distribution

The multinomial distribution extends the binomial distribution. Consider a random variable X with k discrete outcomes. These have probabilities $p = (p_1, \ldots, p_k)$. In n trials, the joint probability distribution is given by

$$\mathbb{P}(X|p) = \binom{n}{(n_1, \ldots, n_k)} p_1^{n_1} \cdots p_k^{n_k}.$$

To obtain p_i, we assume a table `outcome` with column `outcome` representing the base population.

```
CREATE VIEW B AS
SELECT outcome,
       outcome_count / sum(outcome_count) over () AS p
  FROM (SELECT outcome, count(*)::numeric AS outcome_count
          FROM input
         GROUP BY outcome) AS a
```

In the context of model selection, it is often convenient to compare the same data set under two different multinomial distributions.

$$\begin{aligned} LLR &= -2\log\left(\frac{\mathbb{P}(X|p)}{\mathbb{P}(X|\tilde{p})}\right) \\ &= -2\log\left(\frac{\binom{n}{n_1,\ldots,n_k} p_1^{n_1}\cdots p_k^{n_k}}{\binom{n}{n_1,\ldots,n_k} \tilde{p}_1^{n_1}\cdots \tilde{p}_k^{n_k}}\right) \\ &= 2\sum_i n_i \log \tilde{p}_i - \sum_i n_i \log p_i. \end{aligned}$$

Or in SQL:

```
SELECT 2 * sum(T.outcome_count * log B.p)
     - 2 * sum(T.outcome_count * log T.p)
  FROM B, test_population AS T
 WHERE B.outcome = T.outcome
```

5.4 Resampling Techniques

Parametric modeling assumes that data comes from some process that is well-represented by mathematical models with a number of parameters – e.g., the mean and variance of a Normal distribution. The parameters of the real-world process need to be estimated from existing data that serves as a "sample" of that process. One might be tempted to simply use the SQL AVERAGE and STDDEV aggregates to do this on a big dataset, but that is typically not a good idea. Large real-world datasets like those at FAN invariable contain outlier values and other artifacts. Naive "sample statistics" over the data – i.e., simple SQL aggregates – are not *robust* [10], and will "overfit" to those artifacts. This can keep the models from properly representing the real-world process of interest.

The basic idea in resampling is to repeatedly take samples of a data set in a controlled fashion, compute a summary statistic over each sample, and carefully combine the samples to estimate of a property of the entire data set more robustly. Intuitively, rare outliers will appear in few or no samples, and hence will not perturb the estimators used.

There are two standard resampling techniques in the statistics literature. The *bootstrap* method is straightforward: from a population of size N, pick k members (a *subsample* of size k) from the population and compute the desired statistic θ_0. Now replace your subsample and pick another random k members. Your new statistic θ_1 will be different from your previous statistic. Repeat this "sampling" process tens of thousands of times. The distribution of the resulting θ_i's is called the *sampling distribution*. The Central Limit Theorem says that the sampling distribution is normal, so the mean of a large sampling distribution produces an accurate measure θ^*. The alternative to bootstrapping is the *jackknife* method, which repeatedly recomputes a summary statistic θ_i by *leaving out* one or more data items from the full data set to measure the influence of certain subpopulations. The resulting set of observations is used as a sampling distribution in the same way as in bootstrapping, to generate a good estimator for the underlying statistic of interest.

Importantly, it is not required that all subsamples be of the exact same size, though widely disparate subsample sizes can lead to incorrect error bounds.

Assuming the statistic of interest θ is easy to obtain via SQL, for instance the average of a set of values, then the only work needing to be done is to orchestrate the resampling via a sufficiently random number generator. We illustrate this with an example of bootstrapping. Consider a table T with two columns *(row_id, value)* and N rows. Assume that the *row_id* column ranges densely from $1 \ldots N$. Because each sample is done with replacement, we can pre-assign the subsampling. That is, if we do M samples, we can decide in advance that record i will appear in subsamples $1, 2, 25, \ldots$ etc.

The function random() generates a uniformly random element of $(0, 1)$ and floor(x) truncates to the integer portion of x. We use these functions to design a resampling experiment. Suppose we have $N = 100$ subjects and we wish to have $10,000$ trials each with subsample size 3.

```
CREATE VIEW design AS
SELECT a.trial_id,
       floor (100 * random()) AS row_id
 FROM generate_series(1,10000) AS a (trial_id),
      generate_series(1,3) AS b (subsample_id)
```

The reliance upon the random number generator here is system dependent and the researcher needs to verify that scaling the random() function still returns a uniform random variable along the scale. Performing the experiment over the view now takes a single query:

```
CREATE VIEW trials AS
SELECT d.trial_id, AVG(a.values) AS avg_value
  FROM design d, T
 WHERE d.row_id = T.row_id
 GROUP BY d.trial_id
```

This returns the sampling distribution: the average values of each subsample. The final result of the bootstrapping process is a simple query over this view:

```
SELECT AVG(avg_value), STDDEV(avg_value)
  FROM trials;
```

This query returns the statistic of interest after the given number of resamples. The AVG() and STDDEV() functions are already done in parallel, so the entire technique is done in parallel. Note that the design view is relatively small ($\sim 30,000$ rows) so it will fit in memory; hence all $10,000$ "experiments" are performed in a single parallel pass of the table T; i.e., roughly the same cost as computing a naive SQL aggregate.

Jackknifing makes use of a similar trick to generate multiple experiments in a single pass of the table, excluding a random subpopulation in each.

6. MAD DBMS

The MAD approach we sketch in Section 4 requires support from the DBMS. First, getting data into a "Magnetic" database must be painless and efficient, so analysts will play with new data sources within the warehouse. Second, to encourage "Agility", the system has to make physical storage evolution easy and efficient. Finally, "Depth" of analysis – and really all aspects of MAD analytics – require the database to be a powerful, flexible programming environment that welcomes developers of various stripes.

6.1 Loading and Unloading

The importance of high-speed data loading and dumping for big parallel DBMSs was highlighted over a decade ago [1], and it is even more important today. Analysts load new data sets frequently, and quite often like to "slosh" large data sets between systems (e.g., between the DBMS and a Hadoop cluster) for specific tasks. As they clean and refine data and engage in developing analysis processes, they iterate over tasks frequently. If load times are measured in days, the flow of an analyst's work qualitatively changes. These kinds of delays repel data from the warehouse.

In addition to loading the database quickly, a good DBMS for MAD analytics should enable database users to run queries directly against *external tables*: raw feeds from files or services that are accessed on demand during query processing. By accessing external data directly and in parallel, a good DBMS can eliminate the overhead of landing data and keeping it refreshed. External tables ("wrappers") are typically discussed in the context of data integration [18]. But the focus in a MAD warehouse context is on massively parallel access to file data that lives on a local high-speed network.

Greenplum implements fully parallel access for both loading and query processing over external tables via a technique called *Scatter/Gather Streaming*. The idea is similar to traditional shared-nothing database internals [7], but requires coordination with external processes to "feed" all the DBMS nodes in parallel. As the data is streamed into the system it can be landed in database tables for subsequent access, or used directly as a purely external table with parallel I/O. Using this technology, Greenplum customers have reporting loading speeds of a fully-mirrored, production database in excess of four terabytes per hour with negligible impact on concurrent database operations.

6.1.1 ETL and ELT

Traditional data warehousing is supported by custom tools for the *Extract-Transform-Load (ETL)* task. In recent years, there is increasing pressure to push the work of transformation into the DBMS, to enable parallel execution via SQL transformation scripts. This approach has been dubbed *ELT* since transformation is done after loading. The ELT approach becomes even more natural with external tables. Transformation queries can be written against external tables, removing the need to ever load untransformed data. This can speed up the design loop for transformations substantially – especially when combined with SQL's `LIMIT` clause as a "poor man's Online Aggregation" [11] to debug transformations.

In addition to transformations written in SQL, Greenplum supports MapReduce scripting in the DBMS, which can run over either external data via Scatter/Gather, or in-database tables (Section 6.3). This allows programmers to write transformation scripts in the dataflow-style programming used by many ETL tools, while running at scale using the DBMS' facilities for parallelism.

6.2 Data Evolution: Storage and Partitioning

The data lifecycle in a MAD warehouse includes data in various states. When a data source is first brought into the system, analysts will typically iterate over it frequently with significant analysis and transformation. As transformations and table definitions begin to settle for a particular data source, the workload looks more like traditional EDW settings: frequent appends to large "fact" tables, and occasional updates to "detail" tables. This mature data is likely to be used for ad-hoc analysis as well as for standard reporting tasks. As data in the "fact" tables ages over time, it may be accessed less frequently or even "rolled off" to an external archive. Note that all these stages co-occur in a single warehouse at a given time.

Hence a good DBMS for MAD analytics needs to support multiple storage mechanisms, targeted at different stages of the data lifecycle. In the early stage, external tables provide a lightweight approach to experiment with transformations. Detail tables are often modest in size and undergo periodic updates; they are well served by traditional transactional storage techniques. Append-mostly fact tables can be better served by compressed storage, which can handle appends and reads efficiently, at the expense of making updates slower. It should be possible to roll this data off of the warehouse as it ages, without disrupting ongoing processing.

Greenplum provides multiple storage engines, with a rich SQL partitioning specification to apply them flexibly across and within tables. As mentioned above, Greenplum includes external table support. Greenplum also provides a traditional "heap" storage format for data that sees frequent updates, and a highly-compressed "append-only" (AO) table feature for data that is not going to be updated; both are integrated within a transactional framework. Greenplum AO storage units can have a variety of compression modes. At one extreme, with compression off, bulk loads run very quickly. Alternatively, the most aggressive compression modes are tuned to use as little space as possible. There is also a middle ground with "medium" compression to provide improved table scan time at the expense of slightly slower loads. In a recent version Greenplum also adds "column-store" partitioning of append-only tables, akin to ideas in the literature [20]. This can improve compression, and ensures that queries over large archival tables only do I/O for the columns they need to see.

A DBA should be able to specify the storage mechanism to be used in a flexible way. Greenplum supports many ways to partition tables in order to increase query and data load performance, as well as to aid in managing large data sets. The top-most layer of partitioning is a *distribution policy* specified via a `DISTRIBUTED BY` clause in the `CREATE TABLE` statement that determines how the rows of a table are distributed across the individual nodes that comprise a Greenplum cluster. While all tables have a distribution policy, users can optionally specify a *partitioning policy* for a table, which separates the data in the table into partitions by *range* or *list*. A range partitioning policy lets users specify an ordered, non-overlapping set of partitions for a partitioning column, where each partition has a `START` and `END` value. A list partitioning policy lets users specify a set of partitions for a collection of columns, where each partition corresponds to a particular value. For example, a `sales` table may be hash-distributed over the nodes by `sales_id`. On each node, the rows are further partitioned by range into separate partitions for each month, and each of these partitions is subpartitioned into three separate sales regions. Note that the partitioning structure is completely mutable: a user can add new partitions or drop existing partitions or subpartitions at any point.

Partitioning is important for a number of reasons. First, the query optimizer is aware of the partitioning structure, and can analyze predicates to perform *partition exclusion*: scanning only a subset of the partitions instead of the entire table. Second, each partition of a table can have a different *storage format*, to match the expected workload. A typical arrangement is to partition by a timestamp field, and have older partitions be stored in a highly-compressed append-only format while newer, "hotter" partitions are stored in a more update-friendly format to accommodate auditing updates. Third, it enables *atomic partition exchange*. Rather than inserting data a row at a time, a user can use ETL or ELT to stage their data to a temporary table. After the data is scrubbed and transformed, they can use the `ALTER TABLE ... EXCHANGE PARTITION` command to bind the temporary table as a new partition of an existing table in a quick atomic operation. This capability makes partitioning particularly useful for businesses that perform bulk data loads on a daily, weekly, or monthly basis, especially if they drop or archive older data to keep some fixed size "window" of data online in the warehouse. The same idea also allows users to do physical migration of tables and storage format modifications in a way that mostly isolates production tables from loading and transformation overheads.

6.3 MAD Programming

Although MAD design favors quick import and frequent iteration over careful modeling, it is not intended to reject structured databases per se. As mentioned in Section 4, the structured data management features of a DBMS can be very useful for organizing experimental results, trial datasets, and experimental workflows. In fact, shops that use tools like Hadoop typically have DBMSs in addition, and/or evolve light database systems like Hive. But as we also note in Section 4, it is advantageous to unify the structured environment with the analysts' favorite programming environments.

Data analysts come from many walks of life. Some are experts in SQL, but many are not. Analysts that come from a scientific or mathematical background are typically trained in statistical packages like R, SAS, or Matlab. These are memory-bound, single-machine solutions, but they provide convenient abstractions for math programming, and access to libraries containing hundreds of statistical routines. Other analysts have facility with traditional programming languages like Java, Perl, and Python, but typically do not want to write parallel or I/O-centric code.

The kind of database extensibility pioneered by Postgres [19] is no longer an exotic DBMS feature – it is a key to modern data analytics, enabling code to run close to the data. To be inviting to a variety of programmers, a good DBMS extensibility interface should accommodate multiple languages. PostgreSQL has become quite powerful in this regard, supporting a wide range of extension languages including R, Python and Perl. Greenplum takes these interfaces and enables them to run data-parallel on a cluster. This does not provide automatic parallelism of course: developers must think through how their code works in a data-parallel environment without shared memory, as we did in Section 5.

In addition to work like ours to implement statistical methods in extensible SQL, there is a groundswell of effort to implement methods with the MapReduce programming paradigm popularized by Google [4] and Hadoop. From the perspective of programming language design, MapReduce and modern SQL are quite similar takes on parallelism: both are data-parallel programming models for shared-nothing architectures that provide extension hooks ("upcalls") to intercept individual tuples or sets of tuples within a dataflow. But as a cultural phenomenon, MapReduce has captured the interest of many developers interested in running large-scale analyses on Big Data, and is widely viewed as a more attractive programming environment than SQL. A MAD data warehouse needs to attract these programmers, and allow them to enjoy the familiarity of MapReduce programming in a context that both integrates with the rest of the data in the enterprise, and offers more sophisticated tools for managing data products.

Greenplum approached this challenge by implementing a MapReduce programming interface whose runtime engine is the same query executor used for SQL [9]. Users write Map and Reduce functions in familiar languages like Python, Perl, or R, and connect them up into MapReduce scripts via a simple configuration file. They can then execute these scripts via a command line interface that passes the configuration and MapReduce code to the DBMS, returning output to a configurable location: command line, files, or DBMS tables. The only required DBMS interaction is the specification of an IP address for the DBMS, and authentication credentials (user/password, PGP keys, etc.) Hence developers who are used to traditional open source tools continue to use their favorite code editors, source code management, and shell prompts; they do not need to learn about database utilities, SQL syntax, schema design, etc.

The Greenplum executor accesses files for MapReduce jobs via the same Scatter/Gather technique that it uses for external tables in SQL. In addition, Greenplum MapReduce scripts interoperate with all the features of the database, and vice versa. MapReduce scripts can use database tables or views as their inputs, and/or store their results as database tables that can be directly accessed via SQL. Hence complex pipelines can evolve that include some stages in SQL, and some in MapReduce syntax. Execution can be done entirely on demand – running the SQL and MapReduce stages in a pipeline – or via materialization of steps along the way either inside or outside the database. Programmers of different stripes can interoperate via familiar interfaces: database tables and views, or MapReduce input streams, incorporating a variety of languages for the Map and Reduce functions, and for SQL extension functions.

This kind of interoperability between programming metaphors is critical for MAD analytics. It attracts analysts – and hence data – to the warehouse. It provides agility to developers by facilitating familiar programming interfaces and enabling interoperability among programming styles. Finally, it allows analysts to do deep development using the best tools of the trade, including many domain specific modules written for the implementation languages.

In experience with a variety of Greenplum customers, we have found that developers comfortable with both SQL and MapReduce will choose among them flexibly for different tasks. For example, MapReduce has proved more convenient for writing ETL scripts on files where the input order is known and should be exploited in the transformation. MapReduce also makes it easy to specify transformations that take one input and produce multiple outputs – this is also common in ETL settings that "shred" input records and produce a stream of output tuples with mixed formats. SQL, surprisingly, has been more convenient than MapReduce for tasks involving graph data like web links and social networks, since most of the algorithms in that setting (PageRank, Clustering Coefficients, etc.) can be coded compactly as "self-joins" of a link table.

7. DIRECTIONS AND REFLECTIONS

The work in this paper resulted from a fairly quick, iterative discussion among data-centric people with varying job descriptions and training. The process of arriving at the paper's lessons echoed the lessons themselves. We did not design a document up front, but instead "got MAD": we brought many datapoints together, fostered quick iteration among multiple parties, and tried to dig deeply into details.

As in MAD analysis, we expect to arrive at new questions and new conclusions as more data is brought to light. A few of the issues we are currently considering include the following:

Package management and reuse: In many cases, an analyst simply wants to string together and parameterize textbook techniques like linear regression or resampling "off the shelf". To support this, there is a pressing need – in both the SQL and MapReduce environments – for a package management solution and repository akin to the CRAN repository for R, to enable very easy code reuse. Among other challenges, this requires standardizing a vocabulary for objects like vectors, matrices, functions and functionals.

Co-optimizing storage and queries for linear algebra: There are many choices for laying out matrices across nodes in a cluster [2]. We believe that all of them can be captured in records, with linear algebra methods written over them in SQL or MapReduce syntax. The next step of sophistication is for a query optimizer to reason about (a) multiple, redundant layouts for matrices, and (b) choosing among equivalent libraries of linear algebra routines that are customized to the different layouts.

Automating physical design for iterative tasks: Analysts engaged in either ETL/ELT or in core analytics often take multiple passes over massive data sets in various forms. They are currently required to think about how to store that data: leave it external, land it into one of many storage formats, materialize repeated computations, etc. Analysts are not interested in these questions, they are interested in the data. It would be useful for a system to (perhaps semi-) automatically tune these decisions.

Online query processing for MAD analytics: Agility in the design of analytics depends on how frequently an analyst can iterate. Techniques like Online Aggregation [11] can be useful to radically speed up this process, but the literature on the topic needs to be extended substantially to cover the depth of analysis discussed here. This includes techniques to provide useful running estimates for more complex computations, including ad-hoc code like MapReduce programs. It also includes techniques to appropriately prioritize data for processing in a way that captures interesting data in the tails of distributions.

7.1 When is the Future?

We are not just speculating when we say that MAD approaches are the future of analytics. For a number of leading organizations, that future has already arrived, and brings clear and growing value. But for more conservative organizations, there may not even be a roadmap to get MAD, and they may have been burned in the past by staff statisticians or data mining initiatives that failed to gain ground. Those lessons have to be taken in the context of their time. Returning to Varian's point that opens the paper, the economics of data are changing exponentially fast. The question is not whether to get MAD, but how and when. In nearly all environments an evolutionary approach makes sense: traditionally conservative data warehousing functions are maintained even as new skills, practices and people are brought together to develop MAD approaches. This coexistence of multiple styles of analysis is itself an example of MAD design.

8. ACKNOWLEDGMENTS

Thanks to Noelle Sio and David Hubbard of FAN for their input, and to James Marca of UC Irvine for a fix to our bootstrapping discussion.

9. REFERENCES

[1] T. Barclay et al. Loading databases using dataflow parallelism. *SIGMOD Record*, 23(4), 1994.

[2] J. Choi et al. ScaLAPACK: a portable linear algebra library for distributed memory computers – design issues and performance. *Computer Physics Communications*, 97(1-2), 1996. High-Performance Computing in Science.

[3] C.-T. Chu et al. Map-Reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[5] S. Dubner. Hal Varian answers your questions, February 2008.

[6] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec.*, 34(4), 2005.

[7] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *SIGMOD Rec.*, 19(2), 1990.

[8] J Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1), 1997.

[9] Greenplum. A unified engine for RDBMS and MapReduce, 2009. `http://www.greenplum.com/resources/mapreduce/`.

[10] F. R. Hampel et al. *Robust Statistics – The Approach Based on Influence Functions.* Wiley, 1986.

[11] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM SIGMOD*, 1997.

[12] W. Holland, February 2009. Downloaded from `http://www.urbandictionary.com/define.php?term=mad`.

[13] W. H. Inmon. *Building the Data Warehouse.* Wiley, 2005.

[14] Y. E. Ioannidis et al. Zoo: A desktop experiment management environment. In *VLDB*, 1996.

[15] A. Kaushik. *Web Analytics: An Hour a Day.* Sybex, 2007.

[16] N. Khoussainova et al. A case for a collaborative query management system. In *CIDR*, 2009.

[17] K. Lange. *Optimization.* Springer, 2004.

[18] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *VLDB*, 1997.

[19] M. Stonebraker. Inclusion of new types in relational data base systems. In *ICDE*, 1986.

[20] M. Stonebraker et al. C-store: a column-oriented dbms. In *VLDB*, 2005.

[21] M. Stonebraker et al. Requirements for science data bases and SciDB. In *CIDR*, 2009.

[22] A. S. Szalay et al. Designing and mining multi-terabyte astronomy archives: the sloan digital sky survey. *SIGMOD Rec.*, 29(2), 2000.

[23] R. Vuduc, J. Demmel, and K. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *SciDAC*, 2005.

[24] M.J. Zaki and C.-T. Ho. *Large-Scale Parallel Data Mining.* Springer, 2000.

[25] Y. Zhang, H. Herodotou, and J. Yang. Riot: I/O-efficient numerical computing without SQL. In *CIDR*, 2009.