# 16 Important Data Science Papers

All the ideas which you see in sci-fi movies can actually turn into reality by Data Science. Data Science is the future of Artificial Intelligence. In this book, we have listed some of the most influential papers in the history of Data Science. We've compiled these papers based on recommendations by big data enthusiasts on various social media channels.

**Contents**

Knowl Inf Syst (2008) 14:1–37
DOI 10.1007/s10115-007-0114-2

SURVEY PAPER

# Top 10 algorithms in data mining

**Xindong Wu · Vipin Kumar · J. Ross Quinlan · Joydeep Ghosh · Qiang Yang · Hiroshi Motoda · Geoffrey J. McLachlan · Angus Ng · Bing Liu · Philip S. Yu · Zhi-Hua Zhou · Michael Steinbach · David J. Hand · Dan Steinberg**

Received: 9 July 2007 / Revised: 28 September 2007 / Accepted: 8 October 2007
Published online: 4 December 2007


**Abstract** This paper presents the top 10 data mining algorithms identified by the IEEE International Conference on Data Mining (ICDM) in December 2006: C4.5, *k*-Means, SVM, Apriori, EM, PageRank, AdaBoost, *k*NN, Naive Bayes, and CART. These top 10 algorithms are among the most influential data mining algorithms in the research community. With each algorithm, we provide a description of the algorithm, discuss the impact of the algorithm, and review current and further research on the algorithm. These 10 algorithms cover classification,

X. Wu (✉)
Department of Computer Science, University of Vermont, Burlington, VT, USA
e-mail: xwu@cs.uvm.edu

V. Kumar
Department of Computer Science and Engineering,
University of Minnesota, Minneapolis, MN, USA
e-mail: kumar@cs.umn.edu

J. Ross Quinlan
Rulequest Research Pty Ltd,
St Ives, NSW, Australia
e-mail: quinlan@rulequest.com

J. Ghosh
Department of Electrical and Computer Engineering,
University of Texas at Austin, Austin, TX 78712, USA
e-mail: ghosh@ece.utexas.edu

Q. Yang
Department of Computer Science,
Hong Kong University of Science and Technology,
Honkong, China
e-mail: qyang@cs.ust.hk

H. Motoda
AFOSR/AOARD and Osaka University,
7-23-17 Roppongi, Minato-ku, Tokyo 106-0032, Japan
e-mail: motoda@ar.sanken.osaka-u.ac.jp

clustering, statistical learning, association analysis, and link mining, which are all among the most important topics in data mining research and development.

## 0 Introduction

In an effort to identify some of the most influential algorithms that have been widely used in the data mining community, the IEEE International Conference on Data Mining (ICDM, http://www.cs.uvm.edu/~icdm/) identified the top 10 algorithms in data mining for presentation at ICDM '06 in Hong Kong.

As the first step in the identification process, in September 2006 we invited the ACM KDD Innovation Award and IEEE ICDM Research Contributions Award winners to each nominate up to 10 best-known algorithms in data mining. All except one in this distinguished set of award winners responded to our invitation. We asked each nomination to provide the following information: (a) the algorithm name, (b) a brief justification, and (c) a representative publication reference. We also advised that each nominated algorithm should have been widely cited and used by other researchers in the field, and the nominations from each nominator as a group should have a reasonable representation of the different areas in data mining.

G. J. McLachlan
Department of Mathematics,
The University of Queensland, Brisbane, Australia
e-mail: gjm@maths.uq.edu.au

A. Ng
School of Medicine, Griffith University,
Brisbane, Australia

B. Liu
Department of Computer Science,
University of Illinois at Chicago, Chicago, IL 60607, USA

P. S. Yu
IBM T. J. Watson Research Center,
Hawthorne, NY 10532, USA
e-mail: psyu@us.ibm.com

Z.-H. Zhou
National Key Laboratory for Novel Software Technology,
Nanjing University, Nanjing 210093, China
e-mail: zhouzh@nju.edu.cn

M. Steinbach
Department of Computer Science and Engineering,
University of Minnesota, Minneapolis, MN 55455, USA
e-mail: steinbac@cs.umn.edu

D. J. Hand
Department of Mathematics,
Imperial College, London, UK
e-mail: d.j.hand@imperial.ac.uk

D. Steinberg
Salford Systems,
San Diego, CA 92123, USA
e-mail: dsx@salford-systems.com

After the nominations in Step 1, we verified each nomination for its citations on Google Scholar in late October 2006, and removed those nominations that did not have at least 50 citations. All remaining (18) nominations were then organized in 10 topics: association analysis, classification, clustering, statistical learning, bagging and boosting, sequential patterns, integrated mining, rough sets, link mining, and graph mining. For some of these 18 algorithms such as *k*-means, the representative publication was not necessarily the original paper that introduced the algorithm, but a recent paper that highlights the importance of the technique. These representative publications are available at the ICDM website (http://www.cs.uvm.edu/~icdm/algorithms/CandidateList.shtml).

In the third step of the identification process, we had a wider involvement of the research community. We invited the Program Committee members of KDD-06 (the 2006 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining), ICDM '06 (the 2006 IEEE International Conference on Data Mining), and SDM '06 (the 2006 SIAM International Conference on Data Mining), as well as the ACM KDD Innovation Award and IEEE ICDM Research Contributions Award winners to each vote for up to 10 well-known algorithms from the 18-algorithm candidate list. The voting results of this step were presented at the ICDM '06 panel on Top 10 Algorithms in Data Mining.

At the ICDM '06 panel of December 21, 2006, we also took an open vote with all 145 attendees on the top 10 algorithms from the above 18-algorithm candidate list, and the top 10 algorithms from this open vote were the same as the voting results from the above third step. The 3-hour panel was organized as the last session of the ICDM '06 conference, in parallel with 7 paper presentation sessions of the Web Intelligence (WI '06) and Intelligent Agent Technology (IAT '06) conferences at the same location, and attracting 145 participants to this panel clearly showed that the panel was a great success.

## 1 C4.5 and beyond

### 1.1 Introduction

Systems that construct classifiers are one of the commonly used tools in data mining. Such systems take as input a collection of cases, each belonging to one of a small number of classes and described by its values for a fixed set of attributes, and output a classifier that can accurately predict the class to which a new case belongs.

These notes describe C4.5 [64], a descendant of CLS [41] and ID3 [62]. Like CLS and ID3, C4.5 generates classifiers expressed as decision trees, but it can also construct classifiers in more comprehensible ruleset form. We will outline the algorithms employed in C4.5, highlight some changes in its successor See5/C5.0, and conclude with a couple of open research issues.

### 1.2 Decision trees

Given a set S of cases, C4.5 first grows an initial tree using the divide-and-conquer algorithm as follows:

- If all the cases in S belong to the same class or S is small, the tree is a leaf labeled with the most frequent class in S.
- Otherwise, choose a test based on a single attribute with two or more outcomes. Make this test the root of the tree with one branch for each outcome of the test, partition S into corresponding subsets $S_1, S_2, \ldots$ according to the outcome for each case, and apply the same procedure recursively to each subset.

There are usually many tests that could be chosen in this last step. C4.5 uses two heuristic criteria to rank possible tests: information gain, which minimizes the total entropy of the subsets $\{S_i\}$ (but is heavily biased towards tests with numerous outcomes), and the default gain ratio that divides information gain by the information provided by the test outcomes.

Attributes can be either numeric or nominal and this determines the format of the test outcomes. For a numeric attribute $A$ they are $\{A \leq h, A > h\}$ where the threshold $h$ is found by sorting $S$ on the values of $A$ and choosing the split between successive values that maximizes the criterion above. An attribute $A$ with discrete values has by default one outcome for each value, but an option allows the values to be grouped into two or more subsets with one outcome for each subset.

The initial tree is then pruned to avoid overfitting. The pruning algorithm is based on a pessimistic estimate of the error rate associated with a set of $N$ cases, $E$ of which do not belong to the most frequent class. Instead of $E/N$, C4.5 determines the upper limit of the binomial probability when $E$ events have been observed in $N$ trials, using a user-specified confidence whose default value is 0.25.

Pruning is carried out from the leaves to the root. The estimated error at a leaf with $N$ cases and $E$ errors is $N$ times the pessimistic error rate as above. For a subtree, C4.5 adds the estimated errors of the branches and compares this to the estimated error if the subtree is replaced by a leaf; if the latter is no higher than the former, the subtree is pruned. Similarly, C4.5 checks the estimated error if the subtree is replaced by one of its branches and when this appears beneficial the tree is modified accordingly. The pruning process is completed in one pass through the tree.

C4.5's tree-construction algorithm differs in several respects from CART [9], for instance:

- Tests in CART are always binary, but C4.5 allows two or more outcomes.
- CART uses the Gini diversity index to rank tests, whereas C4.5 uses information-based criteria.
- CART prunes trees using a cost-complexity model whose parameters are estimated by cross-validation; C4.5 uses a single-pass algorithm derived from binomial confidence limits.
- This brief discussion has not mentioned what happens when some of a case's values are unknown. CART looks for surrogate tests that approximate the outcomes when the tested attribute has an unknown value, but C4.5 apportions the case probabilistically among the outcomes.

### 1.3 Ruleset classifiers

Complex decision trees can be difficult to understand, for instance because information about one class is usually distributed throughout the tree. C4.5 introduced an alternative formalism consisting of a list of rules of the form "if A and B and C and ... then class X", where rules for each class are grouped together. A case is classified by finding the first rule whose conditions are satisfied by the case; if no rule is satisfied, the case is assigned to a default class.

C4.5 rulesets are formed from the initial (unpruned) decision tree. Each path from the root of the tree to a leaf becomes a prototype rule whose conditions are the outcomes along the path and whose class is the label of the leaf. This rule is then simplified by determining the effect of discarding each condition in turn. Dropping a condition may increase the number $N$ of cases covered by the rule, and also the number $E$ of cases that do not belong to the class nominated by the rule, and may lower the pessimistic error rate determined as above. A hill-climbing algorithm is used to drop conditions until the lowest pessimistic error rate is found.

To complete the process, a subset of simplified rules is selected for each class in turn. These class subsets are ordered to minimize the error on the training cases and a default class is chosen. The final ruleset usually has far fewer rules than the number of leaves on the pruned decision tree.

The principal disadvantage of C4.5's rulesets is the amount of CPU time and memory that they require. In one experiment, samples ranging from 10,000 to 100,000 cases were drawn from a large dataset. For decision trees, moving from 10 to 100K cases increased CPU time on a PC from 1.4 to 61 s, a factor of 44. The time required for rulesets, however, increased from 32 to 9,715 s, a factor of 300.

### 1.4 See5/C5.0

C4.5 was superseded in 1997 by a commercial system See5/C5.0 (or C5.0 for short). The changes encompass new capabilities as well as much-improved efficiency, and include:

- A variant of boosting [24], which constructs an ensemble of classifiers that are then voted to give a final classification. Boosting often leads to a dramatic improvement in predictive accuracy.
- New data types (e.g., dates), "not applicable" values, variable misclassification costs, and mechanisms to pre-filter attributes.
- Unordered rulesets—when a case is classified, all applicable rules are found and voted. This improves both the interpretability of rulesets and their predictive accuracy.
- Greatly improved scalability of both decision trees and (particularly) rulesets. Scalability is enhanced by multi-threading; C5.0 can take advantage of computers with multiple CPUs and/or cores.

More details are available from http://rulequest.com/see5-comparison.html.

### 1.5 Research issues

We have frequently heard colleagues express the view that decision trees are a "solved problem." We do not agree with this proposition and will close with a couple of open research problems.

*Stable trees.* It is well known that the error rate of a tree on the cases from which it was constructed (the resubstitution error rate) is much lower than the error rate on unseen cases (the predictive error rate). For example, on a well-known letter recognition dataset with 20,000 cases, the resubstitution error rate for C4.5 is 4%, but the error rate from a leave-one-out (20,000-fold) cross-validation is 11.7%. As this demonstrates, leaving out a single case from 20,000 often affects the tree that is constructed!

Suppose now that we could develop a non-trivial tree-construction algorithm that was hardly ever affected by omitting a single case. For such stable trees, the resubstitution error rate should approximate the leave-one-out cross-validated error rate, suggesting that the tree is of the "right" size.

*Decomposing complex trees.* Ensemble classifiers, whether generated by boosting, bagging, weight randomization, or other techniques, usually offer improved predictive accuracy. Now, given a small number of decision trees, it is possible to generate a single (very complex) tree that is exactly equivalent to voting the original trees, but can we go the other way? That is, can a complex tree be broken down to a small collection of simple trees that, when voted together, give the same result as the complex tree? Such decomposition would be of great help in producing comprehensible decision trees.

C4.5 Acknowledgments

Research on C4.5 was funded for many years by the Australian Research Council.

C4.5 is freely available for research and teaching, and source can be downloaded from http://rulequest.com/Personal/c4.5r8.tar.gz.

## 2 The *k*-means algorithm

### 2.1 The algorithm

The `k-means` algorithm is a simple iterative method to partition a given dataset into a user-specified number of clusters, $k$. This algorithm has been discovered by several researchers across different disciplines, most notably Lloyd (1957, 1982) [53], Forgey (1965), Friedman and Rubin (1967), and McQueen (1967). A detailed history of `k-means` alongwith descriptions of several variations are given in [43]. Gray and Neuhoff [34] provide a nice historical background for `k-means` placed in the larger context of hill-climbing algorithms.

The algorithm operates on a set of $d$-dimensional vectors, $D = \{\mathbf{x_i} \mid i = 1, \ldots, N\}$, where $\mathbf{x_i} \in \Re^d$ denotes the $i$th data point. The algorithm is initialized by picking $k$ points in $\Re^d$ as the initial $k$ cluster representatives or "centroids". Techniques for selecting these initial seeds include sampling at random from the dataset, setting them as the solution of clustering a small subset of the data or perturbing the global mean of the data $k$ times. Then the algorithm iterates between two steps till convergence:

*Step 1: Data Assignment*. Each data point is assigned to its *closest* centroid, with ties broken arbitrarily. This results in a partitioning of the data.

*Step 2: Relocation of "means"*. Each cluster representative is relocated to the center (mean) of all data points assigned to it. If the data points come with a probability measure (weights), then the relocation is to the expectations (weighted mean) of the data partitions.

The algorithm converges when the assignments (and hence the $\mathbf{c_j}$ values) no longer change. The algorithm execution is visually depicted in Fig. 1. Note that each iteration needs $N \times k$ comparisons, which determines the time complexity of one iteration. The number of iterations required for convergence varies and may depend on $N$, but as a first cut, this algorithm can be considered linear in the dataset size.

One issue to resolve is how to quantify "closest" in the assignment step. The default measure of closeness is the Euclidean distance, in which case one can readily show that the non-negative cost function,

$$\sum_{i=1}^{N} \left( \operatorname*{argmin}_{j} ||\mathbf{x_i} - \mathbf{c_j}||_2^2 \right) \tag{1}$$

will decrease whenever there is a change in the assignment or the relocation steps, and hence convergence is guaranteed in a finite number of iterations. The greedy-descent nature of `k-means` on a non-convex cost also implies that the convergence is only to a local optimum, and indeed the algorithm is typically quite sensitive to the initial centroid locations. Figure 2 [1] illustrates how a poorer result is obtained for the same dataset as in Fig. 1 for a different choice of the three initial centroids. The local minima problem can be countered to some

[1] Figures 1 and 2 are taken from the slides for the book, *Introduction to Data Mining*, Tan, Kumar, Steinbach, 2006.

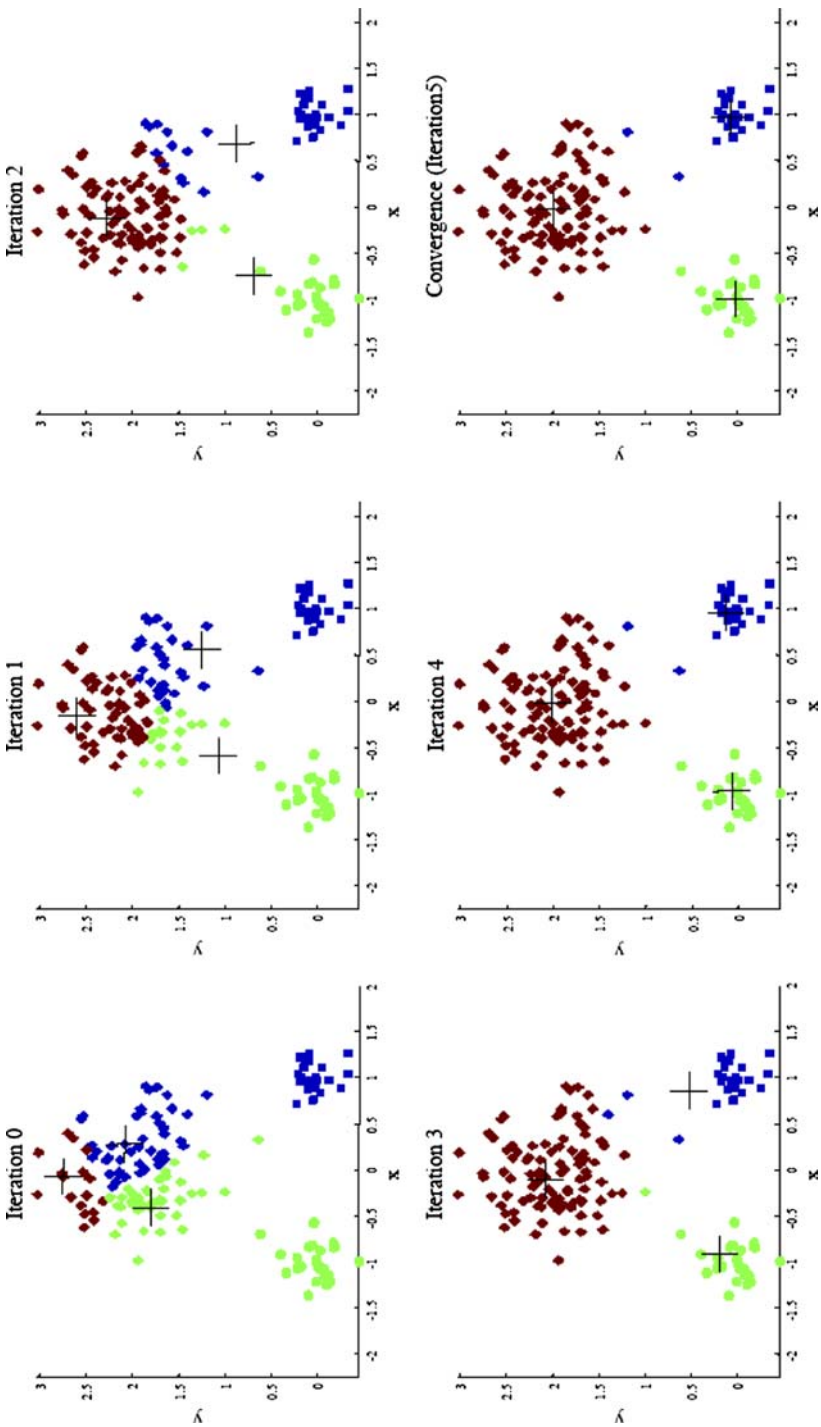



**Fig. 1** Changes in cluster representative locations (indicated by '+' signs) and data assignments (indicated by color) during an execution of the `k-means` algorithm

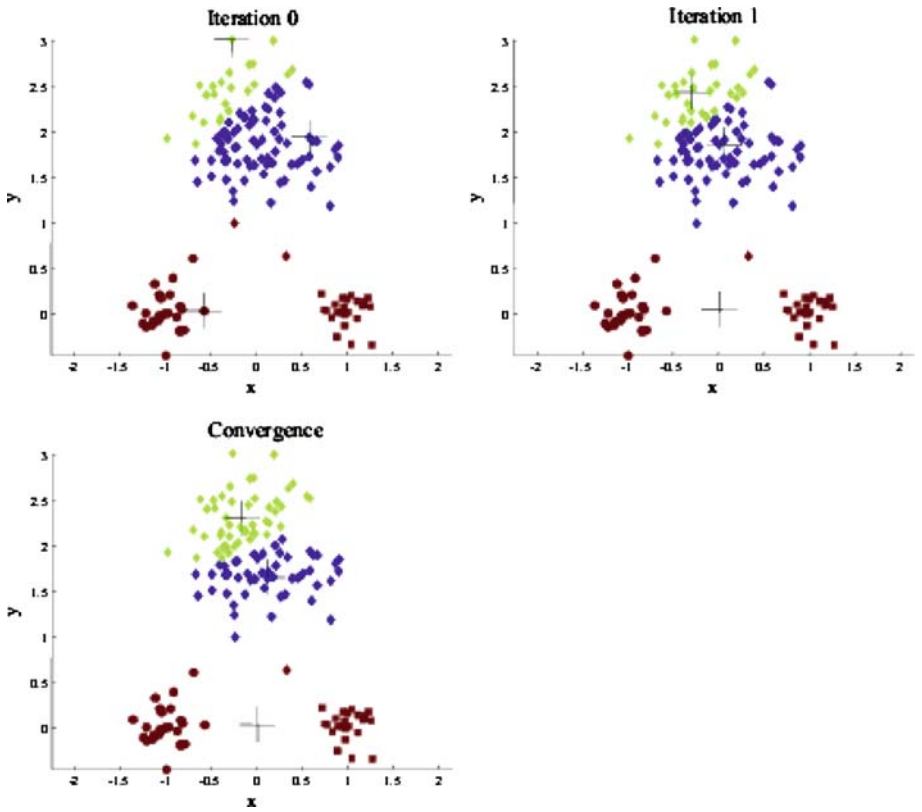



**Fig. 2** Effect of an inferior initialization on the `k-means` results

extent by running the algorithm multiple times with different initial centroids, or by doing limited local search about the converged solution.

## 2.2 Limitations

In addition to being sensitive to initialization, the `k-means` algorithm suffers from several other problems. First, observe that `k-means` is a limiting case of fitting data by a mixture of $k$ Gaussians with identical, isotropic covariance matrices ($\Sigma = \sigma^2\mathbf{I}$), when the soft assignments of data points to mixture components are hardened to allocate each data point solely to the most likely component. So, it will falter whenever the data is not well described by reasonably separated spherical balls, for example, if there are non-covex shaped clusters in the data. This problem may be alleviated by rescaling the data to "whiten" it before clustering, or by using a different distance measure that is more appropriate for the dataset. For example, information-theoretic clustering uses the KL-divergence to measure the distance between two data points representing two discrete probability distributions. It has been recently shown that if one measures distance by selecting any member of a very large class of divergences called Bregman divergences during the assignment step and makes no other changes, the essential properties of `k-means`, including guaranteed convergence, linear separation boundaries and scalability, are retained [3]. This result makes `k-means` effective for a much larger class of datasets so long as an appropriate divergence is used.

k-means can be paired with another algorithm to describe non-convex clusters. One first clusters the data into a large number of groups using k-means. These groups are then agglomerated into larger clusters using single link hierarchical clustering, which can detect complex shapes. This approach also makes the solution less sensitive to initialization, and since the hierarchical method provides results at multiple resolutions, one does not need to pre-specify $k$ either.

The cost of the optimal solution decreases with increasing $k$ till it hits zero when the number of clusters equals the number of distinct data-points. This makes it more difficult to (a) directly compare solutions with different numbers of clusters and (b) to find the optimum value of $k$. If the desired $k$ is not known in advance, one will typically run k-means with different values of $k$, and then use a suitable criterion to select one of the results. For example, SAS uses the cube-clustering-criterion, while X-means adds a complexity term (which increases with $k$) to the original cost function (Eq. 1) and then identifies the $k$ which minimizes this adjusted cost. Alternatively, one can progressively increase the number of clusters, in conjunction with a suitable stopping criterion. Bisecting k-means [73] achieves this by first putting all the data into a single cluster, and then recursively splitting the least compact cluster into two using 2-means. The celebrated LBG algorithm [34] used for vector quantization doubles the number of clusters till a suitable code-book size is obtained. Both these approaches thus alleviate the need to know $k$ beforehand.

The algorithm is also sensitive to the presence of outliers, since "mean" is not a robust statistic. A preprocessing step to remove outliers can be helpful. Post-processing the results, for example to eliminate small clusters, or to merge close clusters into a large cluster, is also desirable. Ball and Hall's ISODATA algorithm from 1967 effectively used both pre- and post-processing on k-means.

### 2.3 Generalizations and connections

As mentioned earlier, k-means is closely related to fitting a mixture of $k$ isotropic Gaussians to the data. Moreover, the generalization of the distance measure to all Bregman divergences is related to fitting the data with a mixture of $k$ components from the exponential family of distributions. Another broad generalization is to view the "means" as probabilistic models instead of points in $R^d$. Here, in the assignment step, each data point is assigned to the most likely model to have generated it. In the "relocation" step, the model parameters are updated to best fit the assigned datasets. Such *model-based* k-means allow one to cater to more complex data, e.g. sequences described by Hidden Markov models.

One can also "kernelize" k-means [19]. Though boundaries between clusters are still linear in the implicit high-dimensional space, they can become non-linear when projected back to the original space, thus allowing kernel k-means to deal with more complex clusters. Dhillon et al. [19] have shown a close connection between kernel k-means and spectral clustering. The *K-medoid* algorithm is similar to k-means except that the centroids have to belong to the data set being clustered. Fuzzy c-means is also similar, except that it computes fuzzy membership functions for each clusters rather than a hard one.

Despite its drawbacks, k-means remains the most widely used partitional clustering algorithm in practice. The algorithm is simple, easily understandable and reasonably scalable, and can be easily modified to deal with streaming data. To deal with very large datasets, substantial effort has also gone into further speeding up k-means, most notably by using kd-trees or exploiting the triangular inequality to avoid comparing each data point with all the centroids during the assignment step. Continual improvements and generalizations of the

basic algorithm have ensured its continued relevance and gradually increased its effectiveness as well.

## 3 Support vector machines

In today's machine learning applications, support vector machines (SVM) [83] are considered a must try—it offers one of the most robust and accurate methods among all well-known algorithms. It has a sound theoretical foundation, requires only a dozen examples for training, and is insensitive to the number of dimensions. In addition, efficient methods for training SVM are also being developed at a fast pace.

In a two-class learning task, the aim of SVM is to find the best classification function to distinguish between members of the two classes in the training data. The metric for the concept of the "best" classification function can be realized geometrically. For a linearly separable dataset, a linear classification function corresponds to a separating hyperplane $f(x)$ that passes through the middle of the two classes, separating the two. Once this function is determined, new data instance $x_n$ can be classified by simply testing the sign of the function $f(x_n)$; $x_n$ belongs to the positive class if $f(x_n) > 0$.

Because there are many such linear hyperplanes, what SVM additionally guarantee is that the best such function is found by maximizing the margin between the two classes. Intuitively, the margin is defined as the amount of space, or separation between the two classes as defined by the hyperplane. Geometrically, the margin corresponds to the shortest distance between the closest data points to a point on the hyperplane. Having this geometric definition allows us to explore how to maximize the margin, so that even though there are an infinite number of hyperplanes, only a few qualify as the solution to SVM.

The reason why SVM insists on finding the maximum margin hyperplanes is that it offers the best generalization ability. It allows not only the best classification performance (e.g., accuracy) on the training data, but also leaves much room for the correct classification of the future data. To ensure that the maximum margin hyperplanes are actually found, an SVM classifier attempts to maximize the following function with respect to $\vec{\mathbf{w}}$ and $b$:

$$L_P = \frac{1}{2} \parallel \vec{\mathbf{w}} \parallel - \sum_{i=1}^{t} \alpha_i y_i (\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}_\mathbf{i} + b) + \sum_{i=1}^{t} \alpha_i \tag{2}$$

where $t$ is the number of training examples, and $\alpha_i$, $i = 1, \ldots, t$, are non-negative numbers such that the derivatives of $L_P$ with respect to $\alpha_i$ are zero. $\alpha_i$ are the Lagrange multipliers and $L_P$ is called the Lagrangian. In this equation, the vectors $\vec{\mathbf{w}}$ and constant $b$ define the hyperplane.

There are several important questions and related extensions on the above basic formulation of support vector machines. We list these questions and extensions below.

1. Can we understand the meaning of the SVM through a solid theoretical foundation?
2. Can we extend the SVM formulation to handle cases where we allow errors to exist, when even the best hyperplane must admit some errors on the training data?
3. Can we extend the SVM formulation so that it works in situations where the training data are not linearly separable?
4. Can we extend the SVM formulation so that the task is to predict numerical values or to rank the instances in the likelihood of being a positive class member, rather than classification?

5. Can we scale up the algorithm for finding the maximum margin hyperplanes to thousands and millions of instances?

*Question* 1 Can we understand the meaning of the SVM through a solid theoretical foundation?

Several important theoretical results exist to answer this question.

A learning machine, such as the SVM, can be modeled as a function class based on some parameters $\alpha$. Different function classes can have different capacity in learning, which is represented by a parameter $h$ known as the VC dimension [83]. The VC dimension measures the maximum number of training examples where the function class can still be used to learn perfectly, by obtaining zero error rates on the training data, for any assignment of class labels on these points. It can be proven that the actual error on the future data is bounded by a sum of two terms. The first term is the training error, and the second term if proportional to the square root of the VC dimension $h$. Thus, if we can minimize $h$, we can minimize the future error, as long as we also minimize the training error. In fact, the above maximum margin function learned by SVM learning algorithms is one such function. Thus, theoretically, the SVM algorithm is well founded.

*Question* 2 Can we extend the SVM formulation to handle cases where we allow errors to exist, when even the best hyperplane must admit some errors on the training data?

To answer this question, imagine that there are a few points of the opposite classes that cross the middle. These points represent the training error that existing even for the maximum margin hyperplanes. The "soft margin" idea is aimed at extending the SVM algorithm [83] so that the hyperplane allows a few of such noisy data to exist. In particular, introduce a slack variable $\xi_i$ to account for the amount of a violation of classification by the function $f(x_i)$; $\xi_i$ has a direct geometric explanation through the distance from a mistakenly classified data instance to the hyperplane $f(x)$. Then, the total cost introduced by the slack variables can be used to revise the original objective minimization function.

*Question* 3 Can we extend the SVM formulation so that it works in situations where the training data are not linearly separable?

The answer to this question depends on an observation on the objective function where the only appearances of $\vec{\mathbf{x}}_\mathbf{i}$ is in the form of a dot product. Thus, if we extend the dot product $\vec{\mathbf{x}}_\mathbf{i} \cdot \vec{\mathbf{x}}_\mathbf{j}$ through a functional mapping $\Phi(\vec{\mathbf{x}}_\mathbf{i})$ of each $\vec{\mathbf{x}}_\mathbf{i}$ to a different space $\mathcal{H}$ of larger and even possibly infinite dimensions, then the equations still hold. In each equation, where we had the dot product $\vec{\mathbf{x}}_\mathbf{i} \cdot \vec{\mathbf{x}}_\mathbf{j}$, we now have the dot product of the transformed vectors $\Phi(\vec{\mathbf{x}}_\mathbf{i}) \cdot \Phi(\vec{\mathbf{x}}_\mathbf{j})$, which is called a kernel function.

The kernel function can be used to define a variety of nonlinear relationship between its inputs. For example, besides linear kernel functions, you can define quadratic or exponential kernel functions. Much study in recent years have gone into the study of different kernels for SVM classification [70] and for many other statistical tests. We can also extend the above descriptions of the SVM classifiers from binary classifiers to problems that involve more than two classes. This can be done by repeatedly using one of the classes as a positive class, and the rest as the negative classes (thus, this method is known as the one-against-all method).

*Question* 4 Can we extend the SVM formulation so that the task is to learn to approximate data using a linear function, or to rank the instances in the likelihood of being a positive class member, rather a classification?

SVM can be easily extended to perform numerical calculations. Here we discuss two such extensions. The first is to extend SVM to perform regression analysis, where the goal is to produce a linear function that can approximate that target function. Careful consideration goes into the choice of the error models; in support vector regression, or SVR, the error is defined to be zero when the difference between actual and predicted values are within a epsilon amount. Otherwise, the *epsilon insensitive* error will grow linearly. The support vectors can then be learned through the minimization of the Lagrangian. An advantage of support vector regression is reported to be its insensitivity to outliers.

Another extension is to learn to rank elements rather than producing a classification for individual elements [39]. Ranking can be reduced to comparing pairs of instances and producing a $+1$ estimate if the pair is in the correct ranking order, and $-1$ otherwise. Thus, a way to reduce this task to SVM learning is to construct new instances for each pair of ranked instance in the training data, and to learn a hyperplane on this new training data.

This method can be applied to many areas where ranking is important, such as in document ranking in information retrieval areas.

*Question* 5 Can we scale up the algorithm for finding the maximum margin hyperplanes to thousands and millions of instances?

One of the initial drawbacks of SVM is its computational inefficiency. However, this problem is being solved with great success. One approach is to break a large optimization problem into a series of smaller problems, where each problem only involves a couple of carefully chosen variables so that the optimization can be done efficiently. The process iterates until all the decomposed optimization problems are solved successfully. A more recent approach is to consider the problem of learning an SVM as that of finding an approximate minimum enclosing ball of a set of instances.

These instances, when mapped to an $N$-dimensional space, represent a core set that can be used to construct an approximation to the minimum enclosing ball. Solving the SVM learning problem on these core sets can produce a good approximation solution in very fast speed. For example, the core-vector machine [81] thus produced can learn an SVM for millions of data in seconds.

## 4 The Apriori algorithm

### 4.1 Description of the algorithm

One of the most popular data mining approaches is to find frequent itemsets from a transaction dataset and derive association rules. Finding frequent itemsets (itemsets with frequency larger than or equal to a user specified minimum support) is not trivial because of its combinatorial explosion. Once frequent itemsets are obtained, it is straightforward to generate association rules with confidence larger than or equal to a user specified minimum confidence.

Apriori is a seminal algorithm for finding frequent itemsets using candidate generation [1]. It is characterized as a level-wise complete search algorithm using anti-monotonicity of itemsets, "if an itemset is not frequent, any of its superset is never frequent". By convention, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. Let the set of frequent itemsets of size $k$ be $F_k$ and their candidates be $C_k$. Apriori first scans the database and searches for frequent itemsets of size 1 by accumulating the count for each item and collecting those that satisfy the minimum support requirement. It then iterates on the following three steps and extracts all the frequent itemsets.

1. Generate $C_{k+1}$, candidates of frequent itemsets of size $k+1$, from the frequent itemsets of size $k$.
2. Scan the database and calculate the support of each candidate of frequent itemsets.
3. Add those itemsets that satisfies the minimum support requirement to $F_{k+1}$.

The Apriori algorithm is shown in Fig. 3. Function apriori-gen in line 3 generates $C_{k+1}$ from $F_k$ in the following two step process:

1. Join step: Generate $R_{K+1}$, the initial candidates of frequent itemsets of size $k+1$ by taking the union of the two frequent itemsets of size $k$, $P_k$ and $Q_k$ that have the first $k-1$ elements in common.

$$R_{K+1} = P_k \cup Q_k = \{item_l, \ldots, item_{k-1}, item_k, item_{k'}\}$$
$$P_k = \{item_l, item_2, \ldots, item_{k-1}, item_k\}$$
$$Q_k = \{item_l, item_2, \ldots, item_{k-1}, item_{k'}\}$$

   where, $iteml < item2 < \cdots < item_k < item_{k'}$.
2. Prune step: Check if all the itemsets of size $k$ in $R_{k+1}$ are frequent and generate $C_{k+1}$ by removing those that do not pass this requirement from $R_{k+1}$. This is because any subset of size $k$ of $C_{k+1}$ that is not frequent cannot be a subset of a frequent itemset of size $k+1$.

Function subset in line 5 finds all the candidates of the frequent itemsets included in transaction $t$. Apriori, then, calculates frequency only for those candidates generated this way by scanning the database.

It is evident that Apriori scans the database at most $k_{\max+1}$ times when the maximum size of frequent itemsets is set at $k_{\max}$.

The Apriori achieves good performance by reducing the size of candidate sets (Fig. 3). However, in situations with very many frequent itemsets, large itemsets, or very low minimum support, it still suffers from the cost of generating a huge number of candidate sets

**Algorithm 1** *Apriori*

$F_l$=(Frequent itemsets of cardinality 1);
for($k = 1; F_k \neq \phi; k++$) do begin
  $C_{k+1}$ = apriori-gen($F_k$); //New candidates
  for all transactions $t \in$ Database do begin
    $C'_t$ = subset($C_{k+1}, t$); //Candidates contained in $t$
    for all candidate $c \in C'_t$ do
      $c.count++$;
    end
    $F_{k+1} = \{C \in C_{k+1} \quad | c.count \geq$ minimum support $\}$
  end
end
Answer $\cup_k F_k$;

**Fig. 3** Apriori algorithm

and scanning the database repeatedly to check a large set of candidate itemsets. In fact, it is necessary to generate $2^{100}$ candidate itemsets to obtain frequent itemsets of size 100.

### 4.2 The impact of the algorithm

Many of the pattern finding algorithms such as decision tree, classification rules and clustering techniques that are frequently used in data mining have been developed in machine learning research community. Frequent pattern and association rule mining is one of the few exceptions to this tradition. The introduction of this technique boosted data mining research and its impact is tremendous. The algorithm is quite simple and easy to implement. Experimenting with Apriori-like algorithm is the first thing that data miners try to do.

### 4.3 Current and further research

Since Apriori algorithm was first introduced and as experience was accumulated, there have been many attempts to devise more efficient algorithms of frequent itemset mining. Many of them share the same idea with Apriori in that they generate candidates. These include hash-based technique, partitioning, sampling and using vertical data format. Hash-based technique can reduce the size of candidate itemsets. Each itemset is hashed into a corresponding bucket by using an appropriate hash function. Since a bucket can contain different itemsets, if its count is less than a minimum support, these itemsets in the bucket can be removed from the candidate sets. A partitioning can be used to divide the entire mining problem into $n$ smaller problems. The dataset is divided into $n$ non-overlapping partitions such that each partition fits into main memory and each partition is mined separately. Since any itemset that is potentially frequent with respect to the entire dataset must occur as a frequent itemset in at least one of the partitions, all the frequent itemsets found this way are candidates, which can be checked by accessing the entire dataset only once. Sampling is simply to mine a random sampled small subset of the entire data. Since there is no guarantee that we can find all the frequent itemsets, normal practice is to use a lower support threshold. Trade off has to be made between accuracy and efficiency. Apriori uses a horizontal data format, i.e. frequent itemsets are associated with each transaction. Using vertical data format is to use a different format in which transaction IDs (TIDs) are associated with each itemset. With this format, mining can be performed by taking the intersection of TIDs. The support count is simply the length of the TID set for the itemset. There is no need to scan the database because TID set carries the complete information required for computing support.

The most outstanding improvement over Apriori would be a method called FP-growth (frequent pattern growth) that succeeded in eliminating candidate generation [36]. It adopts a divide and conquer strategy by (1) compressing the database representing frequent items into a structure called FP-tree (frequent pattern tree) that retains all the essential information and (2) dividing the compressed database into a set of conditional databases, each associated with one frequent itemset and mining each one separately. It scans the database only twice. In the first scan, all the frequent items and their support counts (frequencies) are derived and they are sorted in the order of descending support count in each transaction. In the second scan, items in each transaction are merged into a prefix tree and items (nodes) that appear in common in different transactions are counted. Each node is associated with an item and its count. Nodes with the same label are linked by a pointer called node-link. Since items are sorted in the descending order of frequency, nodes closer to the root of the prefix tree are shared by more transactions, thus resulting in a very compact representation that stores all the necessary information. Pattern growth algorithm works on FP-tree by choosing an

item in the order of increasing frequency and extracting frequent itemsets that contain the chosen item by recursively calling itself on the conditional FP-tree. FP-growth is an order of magnitude faster than the original Apriori algorithm.

There are several other dimensions regarding the extensions of frequent pattern mining. The major ones include the followings: (1) incorporating taxonomy in items [72]: Use of taxonomy makes it possible to extract frequent itemsets that are expressed by higher concepts even when use of the base level concepts produces only infrequent itemsets. (2) incremental mining: In this setting, it is assumed that the database is not stationary and a new instance of transaction keeps added. The algorithm in [12] updates the frequent itemsets without restarting from scratch. (3) using numeric valuable for item: When the item corresponds to a continuous numeric value, current frequent itemset mining algorithm is not applicable unless the values are discretized. A method of subspace clustering can be used to obtain an optimal value interval for each item in each itemset [85]. (4) using other measures than frequency, such as information gain or $\chi^2$ value: These measures are useful in finding discriminative patterns but unfortunately do not satisfy anti-monotonicity property. However, these measures have a nice property of being convex with respect to their arguments and it is possible to estimate their upperbound for supersets of a pattern and thus prune unpromising patterns efficiently. AprioriSMP uses this principle [59]. (5) using richer expressions than itemset: Many algorithms have been proposed for sequences, tree and graphs to enable mining from more complex data structure [90,42]. (6) closed itemsets: A frequent itemset is closed if it is not included in any other frequent itemsets. Thus, once the closed itemsets are found, all the frequent itemsets can be derived from them. LCM is the most efficient algorithm to find the closed itemsets [82].

## 5 The EM algorithm

Finite mixture distributions provide a flexible and mathematical-based approach to the modeling and clustering of data observed on random phenomena. We focus here on the use of normal mixture models, which can be used to cluster continuous data and to estimate the underlying density function. These mixture models can be fitted by maximum likelihood via the EM (Expectation–Maximization) algorithm.

### 5.1 Introduction

Finite mixture models are being increasingly used to model the distributions of a wide variety of random phenomena and to cluster data sets [57]. Here we consider their application in the context of cluster analysis.

We let the $p$-dimensional vector ( $\mathbf{y} = (y_1, \ldots, y_p)^{\mathrm{T}}$) contain the values of $p$ variables measured on each of $n$ (independent) entities to be clustered, and we let $\mathbf{y}_j$ denote the value of $\mathbf{y}$ corresponding to the $j$th entity ($j = 1, \ldots, n$). With the mixture approach to clustering, $\mathbf{y}_1, \ldots, \mathbf{y}_n$ are assumed to be an observed random sample from mixture of a finite number, say $g$, of groups in some unknown proportions $\pi_1, \ldots, \pi_g$.

The mixture density of $\mathbf{y}_j$ is expressed as

$$f(y_i; \Psi) = \sum_{i=1}^{g} \pi_i f_i(y_j; \theta_i) \quad (j = 1, \ldots, n), \tag{3}$$

where the mixing proportions $\pi_1, \ldots, \pi_g$ sum to one and the group-conditional density $f_i(y_j; \theta_i)$ is specified up to a vector $\theta_i$ of unknown parameters ($i = 1, \ldots, g$). The vector of all the unknown parameters is given by

$$\Psi = (\pi_1, \ldots, \pi_{g-1}, \theta_1^{\mathrm{T}}, \ldots, \theta_g^{\mathrm{T}})^{\mathrm{T}},$$

where the superscript "T" denotes vector transpose. Using an estimate of $\Psi$, this approach gives a probabilistic clustering of the data into $g$ clusters in terms of estimates of the posterior probabilities of component membership,

$$\tau_i(y_j, \Psi) = \frac{\pi_i f_i(y_j; \theta_i)}{f(y_j; \Psi)}, \tag{4}$$

where $\tau_i(\mathbf{y}_j)$ is the posterior probability that $y_j$ (really the entity with observation $y_j$) belongs to the $i$th component of the mixture ($i = 1, \ldots, g$; $j = 1, \ldots, n$).

The parameter vector $\Psi$ can be estimated by maximum likelihood. The maximum likelihood estimate (MLE) of $\Psi$, $\hat{\Psi}$, is given by an appropriate root of the likelihood equation,

$$\partial \log L(\Psi)/\partial \Psi = 0, \tag{5}$$

where

$$\log L(\Psi) = \sum_{j=1}^{n} \log f(y_j; \Psi) \tag{6}$$

is the log likelihood function for $\Psi$. Solutions of (6) corresponding to local maximizers can be obtained via the expectation–maximization (EM) algorithm [17].

For the modeling of continuous data, the component-conditional densities are usually taken to belong to the same parametric family, for example, the normal. In this case,

$$f_i(y_j; \theta_i) = \phi(y_j; \mu_i, \Sigma_i), \tag{7}$$

where $\phi(y_j; \mu, \Sigma)$ denotes the $p$-dimensional multivariate normal distribution with mean vector $\mu$ and covariance matrix $\Sigma$.

One attractive feature of adopting mixture models with elliptically symmetric components such as the normal or $t$ densities, is that the implied clustering is invariant under affine transformations of the data (that is, under operations relating to changes in location, scale, and rotation of the data). Thus the clustering process does not depend on irrelevant factors such as the units of measurement or the orientation of the clusters in space.

### 5.2 Maximum likelihood estimation of normal mixtures

McLachlan and Peel [57, Chap. 3] described the E- and M-steps of the EM algorithm for the maximum likelihood (ML) estimation of multivariate normal components; see also [56]. In the EM framework for this problem, the unobservable component labels $z_{ij}$ are treated as being the "missing" data, where $z_{ij}$ is defined to be one or zero according as $y_j$ belongs or does not belong to the $i$th component of the mixture ($i = 1, \ldots, g;$ , $j = 1, \ldots, n$).

On the $(k+1)$th iteration of the EM algorithm, the E-step requires taking the expectation of the complete-data log likelihood $\log L_c(\Psi)$, given the current estimate $\Psi^k$ for $\Psi$. As is linear in the unobservable $z_{ij}$, this E-step is effected by replacing the $z_{ij}$ by their conditional expectation given the observed data $y_j$, using $\Psi^{(k)}$. That is, $z_{ij}$ is replaced by $\tau_{ij}^{(k)}$, which is the posterior probability that $y_j$ belongs to the $i$th component of the mixture, using the

current fit $\Psi^{(k)}$ for $\Psi$ ($i = 1, \ldots, g$; $j = 1, \ldots, n$). It can be expressed as

$$\tau_{ij}^{(k)} = \frac{\pi_i^{(k)} \phi(y_j; \mu_i^{(k)}, \Sigma_i^{(k)})}{f(y_j; \Psi^{(k)})}. \tag{8}$$

On the M-step, the updated estimates of the mixing proportion $\pi_j$, the mean vector $\mu_i$, and the covariance matrix $\Sigma_i$ for the $i$th component are given by

$$\pi_i^{(k+1)} = \sum_{j=1}^{n} \tau_{ij}^{(k)} \Big/ n, \tag{9}$$

$$\mu_i^{(k+1)} = \sum_{j=1}^{n} \tau_{ij}^{(k)} y_j \Big/ \sum_{j=1}^{n} \tau_{ij}^{(k)} \tag{10}$$

and

$$\Sigma_i^{(k+1)} = \frac{\sum_{j=1}^{n} \tau_{ij}^{(k)} (y_j - \mu_i^{(k+1)})(y_j - \mu_i^{(k+1)})^T}{\sum_{j=1}^{n} \tau_{ij}^{(k)}}. \tag{11}$$

It can be seen that the M-step exists in closed form.

These E- and M-steps are alternated until the changes in the estimated parameters or the log likelihood are less than some specified threshold.

### 5.3 Number of clusters

We can make a choice as to an appropriate value of $g$ by consideration of the likelihood function. In the absence of any prior information as to the number of clusters present in the data, we monitor the increase in the log likelihood function as the value of $g$ increases.

At any stage, the choice of $g = g_0$ versus $g = g_1$, for instance $g_1 = g_0 + 1$, can be made by either performing the likelihood ratio test or by using some information-based criterion, such as BIC (Bayesian information criterion). Unfortunately, regularity conditions do not hold for the likelihood ratio test statistic $\lambda$ to have its usual null distribution of chi-squared with degrees of freedom equal to the difference d in the number of parameters for $g = g_1$ and $g = g_0$ components in the mixture models. One way to proceed is to use a resampling approach as in [55]. Alternatively, one can apply BIC, which leads to the selection of $g = g_1$ over $g = g_0$ if $-2 \log \lambda$ is greater than $d \log(n)$.

## 6 PageRank

### 6.1 Overview

PageRank [10] was presented and published by Sergey Brin and Larry Page at the Seventh International World Wide Web Conference (WWW7) in April 1998. It is a search ranking algorithm using hyperlinks on the Web. Based on the algorithm, they built the search engine Google, which has been a huge success. Now, every search engine has its own hyperlink based ranking method.

PageRank produces a static ranking of Web pages in the sense that a PageRank value is computed for each page off-line and it does not depend on search queries. The algorithm relies on the democratic nature of the Web by using its vast link structure as an indicator of an individual page's quality. In essence, PageRank interprets a hyperlink from page $x$ to page $y$ as a vote, by page $x$, for page $y$. However, PageRank looks at more than just the sheer

number of votes, or links that a page receives. It also analyzes the page that casts the vote. Votes casted by pages that are themselves "important" weigh more heavily and help to make other pages more "important". This is exactly the idea of rank prestige in social networks [86].

### 6.2 The algorithm

We now introduce the PageRank formula. Let us first state some main concepts in the Web context.

In-links of page $i$: These are the hyperlinks that point to page $i$ from other pages. Usually, hyperlinks from the same site are not considered.

Out-links of page $i$: These are the hyperlinks that point out to other pages from page $i$. Usually, links to pages of the same site are not considered.

The following ideas based on rank prestige [86] are used to derive the PageRank algorithm:

1. A hyperlink from a page pointing to another page is an implicit conveyance of authority to the target page. Thus, the more in-links that a page i receives, the more prestige the page $i$ has.
2. Pages that point to page $i$ also have their own prestige scores. A page with a higher prestige score pointing to $i$ is more important than a page with a lower prestige score pointing to $i$. In other words, a page is important if it is pointed to by other important pages.

According to rank prestige in social networks, the importance of page $i$ ($i$'s PageRank score) is determined by summing up the PageRank scores of all pages that point to $i$. Since a page may point to many other pages, its prestige score should be shared among all the pages that it points to.

To formulate the above ideas, we treat the Web as a directed graph $G = (V, E)$, where $V$ is the set of vertices or nodes, i.e., the set of all pages, and $E$ is the set of directed edges in the graph, i.e., hyperlinks. Let the total number of pages on the Web be $n$ (i.e., $n = |V|$). The PageRank score of the page $i$ (denoted by $P(i)$) is defined by

$$P(i) = \sum_{(j,i)\in E} \frac{P(j)}{O_j}, \tag{12}$$

where $O_j$ is the number of out-links of page $j$. Mathematically, we have a system of $n$ linear equations (12) with $n$ unknowns. We can use a matrix to represent all the equations. Let $P$ be a $n$-dimensional column vector of PageRank values, i.e.,

$$P = (P(1), P(2), \ldots, P(n))T.$$

Let $A$ be the adjacency matrix of our graph with

$$A_{ij} = \begin{cases} \frac{1}{O_i} & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

We can write the system of n equations with

$$\mathbf{P} = \mathbf{A}^{\mathrm{T}}\mathbf{P}. \tag{14}$$

This is the characteristic equation of the *eigensystem*, where the solution to $P$ is an *eigenvector* with the corresponding *eigenvalue* of 1. Since this is a circular definition, an iterative algorithm is used to solve it. It turns out that if some conditions are satisfied, 1 is

**Fig. 4** The power iteration method for PageRank

**PageRank-Iterate**($G$)
  $\boldsymbol{P}_0 \leftarrow \boldsymbol{e}/n$
  $k \leftarrow 1$
  **repeat**
    $\boldsymbol{P}_k \leftarrow (1-d)\boldsymbol{e} + d\boldsymbol{A}^T\boldsymbol{P}_{k-1}$;
    $k \leftarrow k+1$;
  **until** $||\boldsymbol{P}_k - \boldsymbol{P}_{k-1}||_1 < \varepsilon$
  return $\boldsymbol{P}_k$

the largest eigenvalue and the PageRank vector $P$ is the principal eigenvector. A well known mathematical technique called power iteration [30] can be used to find $P$.

However, the problem is that Eq. (14) does not quite suffice because the Web graph does not meet the conditions. In fact, Eq. (14) can also be derived based on the Markov chain. Then some theoretical results from Markov chains can be applied. After augmenting the Web graph to satisfy the conditions, the following PageRank equation is produced:

$$\mathbf{P} = (1-d)\mathbf{e} + d\mathbf{A}^{\mathrm{T}}\mathbf{P}, \tag{15}$$

where $\mathbf{e}$ is a column vector of all 1's. This gives us the PageRank formula for each page $i$:

$$P(i) = (1-d) + d\sum_{j=1}^{n} A_{ji}P(j), \tag{16}$$

which is equivalent to the formula given in the original PageRank papers [10,61]:

$$P(i) = (1-d) + d\sum_{(j,i)\in E} \frac{P(j)}{O_j}. \tag{17}$$

The parameter $d$ is called the *damping factor* which can be set to a value between 0 and 1. $d = 0.85$ is used in [10,52].

The computation of PageRank values of the Web pages can be done using the power iteration method [30], which produces the principal eigenvector with the eigenvalue of 1. The algorithm is simple, and is given in Fig. 1. One can start with any initial assignments of PageRank values. The iteration ends when the PageRank values do not change much or converge. In Fig. 4, the iteration ends after the 1-norm of the residual vector is less than a pre-specified threshold e.

Since in Web search, we are only interested in the ranking of the pages, the actual convergence may not be necessary. Thus, fewer iterations are needed. In [10], it is reported that on a database of 322 million links the algorithm converges to an acceptable tolerance in roughly 52 iterations.

### 6.3 Further references on PageRank

Since PageRank was presented in [10,61], researchers have proposed many enhancements to the model, alternative models, improvements for its computation, adding the temporal dimension [91], etc. The books by Liu [52] and by Langville and Meyer [49] contain in-depth analyses of PageRank and several other link-based algorithms.

## 7 AdaBoost

### 7.1 Description of the algorithm

*Ensemble learning* [20] deals with methods which employ multiple learners to solve a problem. The generalization ability of an ensemble is usually significantly better than that of a single learner, so ensemble methods are very attractive. The AdaBoost algorithm [24] proposed by Yoav Freund and Robert Schapire is one of the most important ensemble methods, since it has solid theoretical foundation, very accurate prediction, great simplicity (Schapire said it needs only "just 10 lines of code"), and wide and successful applications.

Let $\mathcal{X}$ denote the instance space and $\mathcal{Y}$ the set of class labels. Assume $\mathcal{Y} = \{-1, +1\}$. Given a *weak* or *base learning algorithm* and a training set $\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$ where $\boldsymbol{x}_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$ $(i = 1, \ldots, m)$, the AdaBoost algorithm works as follows. First, it assigns equal weights to all the training examples $(\boldsymbol{x}_i, y_i)(i \in \{1, \ldots, m\})$. Denote the distribution of the weights at the $t$-th learning round as $D_t$. From the training set and $D_t$ the algorithm generates a *weak* or *base learner* $h_t : \mathcal{X} \to \mathcal{Y}$ by calling the base learning algorithm. Then, it uses the training examples to test $h_t$, and the weights of the incorrectly classified examples will be increased. Thus, an updated weight distribution $D_{t+1}$ is obtained. From the training set and $D_{t+1}$ AdaBoost generates another weak learner by calling the base learning algorithm again. Such a process is repeated for $T$ rounds, and the final model is derived by weighted majority voting of the $T$ weak learners, where the weights of the learners are determined during the training process. In practice, the base learning algorithm may be a learning algorithm which can use weighted training examples directly; otherwise the weights can be exploited by sampling the training examples according to the weight distribution $D_t$. The pseudo-code of AdaBoost is shown in Fig. 5.

In order to deal with multi-class problems, Freund and Schapire presented the AdaBoost.M1 algorithm [24] which requires that the weak learners are strong enough even on hard distributions generated during the AdaBoost process. Another popular multi-class version of AdaBoost is AdaBoost.MH [69] which works by decomposing multi-class task to a series of binary tasks. AdaBoost algorithms for dealing with regression problems have also been studied. Since many variants of AdaBoost have been developed during the past decade, *Boosting* has become the most important "family" of ensemble methods.

### 7.2 Impact of the algorithm

As mentioned in Sect. 7.1, AdaBoost is one of the most important ensemble methods, so it is not strange that its high impact can be observed here and there. In this short article we only briefly introduce two issues, one theoretical and the other applied.

In 1988, Kearns and Valiant posed an interesting question, i.e., whether a *weak* learning algorithm that performs just slightly better than random guess could be "boosted" into an arbitrarily accurate *strong* learning algorithm. In other words, whether two complexity classes, *weakly learnable* and *strongly learnable* problems, are equal. Schapire [67] found that the answer to the question is "yes", and the proof he gave is a construction, which is the first Boosting algorithm. So, it is evident that AdaBoost was born with theoretical significance. AdaBoost has given rise to abundant research on theoretical aspects of ensemble methods, which can be easily found in machine learning and statistics literature. It is worth mentioning that for their AdaBoost paper [24], Schapire and Freund won the Godel Prize, which is one of the most prestigious awards in theoretical computer science, in the year of 2003.

**Input:** Data set $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \cdots, (\boldsymbol{x}_m, y_m)\}$;

Base learning algorithm $\mathcal{L}$;

Number of learning rounds $T$.

**Process:**

$D_1(i) = 1/m$. % Initialize the weight distribution

for $t = 1, \cdots, T$:

$h_t = \mathcal{L}(\mathcal{D}, D_t)$; % Train a weak learner $h_t$ from $\mathcal{D}$ using distribution $D_t$

$\epsilon_t = \Pr_{i \sim D_i}[h_t(\boldsymbol{x}_i \neq y_i)]$; % Measure the error of $h_t$

$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$; % Determine the weight of $h_t$

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} \exp(-\alpha_t) & \text{if } h_t(\boldsymbol{x}_i) = y_i \\ \exp(\alpha_t) & \text{if } h_t(\boldsymbol{x}_i) \neq y_i \end{cases}$$

$$= \frac{D_t(i)\exp(-\alpha_t y_i h_t(\boldsymbol{x}_i))}{Z_t}$$ % Update the distribution, where $Z_t$ is

% a normalization factor which enables $D_{t+1}$ be a distribution

end.

**Output:** $H(\boldsymbol{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\boldsymbol{x})\right)$

**Fig. 5** The AdaBoost algorithm

AdaBoost and its variants have been applied to diverse domains with great success. For example, Viola and Jones [84] combined AdaBoost with a cascade process for face detection. They regarded rectangular features as weak learners, and by using AdaBoost to weight the weak learners, they got very intuitive features for face detection. In order to get high accuracy as well as high efficiency, they used a cascade process (which is beyond the scope of this article). As the result, they reported a very strong face detector: On a 466 MHz machine, face detection on a $384 \times 288$ image cost only 0.067 seconds, which is 15 times faster than state-of-the-art face detectors at that time but with comparable accuracy. This face detector has been recognized as one of the most exciting breakthroughs in computer vision (in particular, face detection) during the past decade. It is not strange that "Boosting" has become a buzzword in computer vision and many other application areas.

### 7.3 Further research

Many interesting topics worth further studying. Here we only discuss on one theoretical topic and one applied topic.

Many empirical study show that AdaBoost often does not overfit, i.e., the test error of AdaBoost often tends to decrease even after the training error is zero. Many researchers have studied this and several theoretical explanations have been given, e.g. [38]. Schapire et al. [68] presented a margin-based explanation. They argued that AdaBoost is able to increase the margins even after the training error is zero, and thus it does not overfit even after a large number of rounds. However, Breiman [8] indicated that larger margin does not necessarily mean

better generalization, which seriously challenged the margin-based explanation. Recently, Reyzin and Schapire [65] found that Breiman considered minimum margin instead of average or median margin, which suggests that the margin-based explanation still has chance to survive. If this explanation succeeds, a strong connection between AdaBoost and SVM could be found. It is obvious that this topic is well worth studying.

Many real-world applications are born with high dimensionality, i.e., with a large amount of input features. There are two paradigms that can help us to deal with such kind of data, i.e., dimension reduction and feature selection. Dimension reduction methods are usually based on mathematical projections, which attempt to transform the original features into an appropriate feature space. After dimension reduction, the original meaning of the features is usually lost. Feature selection methods directly select some original features to use, and therefore they can preserve the original meaning of the features, which is very desirable in many applications. However, feature selection methods are usually based on heuristics, lacking solid theoretical foundation. Inspired by Viola and Jones's work [84], we think AdaBoost could be very useful in feature selection, especially when considering that it has solid theoretical foundation. Current research mainly focus on images, yet we think general AdaBoost-based feature selection techniques are well worth studying.

## 8 *k*NN: *k*-nearest neighbor classification

### 8.1 Description of the algorithm

One of the simplest, and rather trivial classifiers is the Rote classifier, which memorizes the entire training data and performs classification only if the attributes of the test object match one of the training examples exactly. An obvious drawback of this approach is that many test records will not be classified because they do not exactly match any of the training records. A more sophisticated approach, $k$-nearest neighbor ($k$NN) classification [23,75], finds a group of $k$ objects in the training set that are closest to the test object, and bases the assignment of a label on the predominance of a particular class in this neighborhood. There are three key elements of this approach: a set of labeled objects, e.g., a set of stored records, a distance or similarity metric to compute distance between objects, and the value of $k$, the number of nearest neighbors. To classify an unlabeled object, the distance of this object to the labeled objects is computed, its $k$-nearest neighbors are identified, and the class labels of these nearest neighbors are then used to determine the class label of the object.

Figure 6 provides a high-level summary of the nearest-neighbor classification method. Given a training set $D$ and a test object $x = (\mathbf{x}', y')$, the algorithm computes the distance (or similarity) between $z$ and all the training objects $(\mathbf{x}, y) \in D$ to determine its nearest-neighbor list, $D_z$. ($\mathbf{x}$ is the data of a training object, while $y$ is its class. Likewise, $\mathbf{x}'$ is the data of the test object and $y'$ is its class.)

Once the nearest-neighbor list is obtained, the test object is classified based on the majority class of its nearest neighbors:

$$\text{Majority Voting: } y' = \operatorname*{argmax}_{v} \sum_{(\mathbf{x}_i, y_i) \in D_z} I(v = y_i), \tag{18}$$

where $v$ is a class label, $y_i$ is the class label for the $i$th nearest neighbors, and $I(\cdot)$ is an indicator function that returns the value 1 if its argument is true and 0 otherwise.

**Input:** $D$, the set of $k$ training objects, and test object $z = (\mathbf{x}', y')$

**Process:**

Compute $d(\mathbf{x}', \mathbf{x})$, the distance between $z$ and every object, $(\mathbf{x}, y) \in D$.

Select $D_z \subseteq D$, the set of $k$ closest training objects to $z$.

**Output:** $y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} I(v = y_i)$

**Fig. 6** The $k$-nearest neighbor classification algorithm

## 8.2 Issues

There are several key issues that affect the performance of *k*NN. One is the choice of $k$. If $k$ is too small, then the result can be sensitive to noise points. On the other hand, if $k$ is too large, then the neighborhood may include too many points from other classes.

Another issue is the approach to combining the class labels. The simplest method is to take a majority vote, but this can be a problem if the nearest neighbors vary widely in their distance and the closer neighbors more reliably indicate the class of the object. A more sophisticated approach, which is usually much less sensitive to the choice of $k$, weights each object's vote by its distance, where the weight factor is often taken to be the reciprocal of the squared distance: $w_i = 1/d(\mathbf{x}', \mathbf{x}_i)^2$. This amounts to replacing the last step of the kNN algorithm with the following:

$$\text{Distance-Weighted Voting: } y' = \underset{v}{\operatorname{argmax}} \sum_{(\mathbf{x}_i, y_i) \in D_z} w_i \times I(v = y_i). \tag{19}$$

The choice of the distance measure is another important consideration. Although various measures can be used to compute the distance between two points, the most desirable distance measure is one for which a smaller distance between two objects implies a greater likelihood of having the same class. Thus, for example, if *k*NN is being applied to classify documents, then it may be better to use the cosine measure rather than Euclidean distance. Some distance measures can also be affected by the high dimensionality of the data. In particular, it is well known that the Euclidean distance measure become less discriminating as the number of attributes increases. Also, attributes may have to be scaled to prevent distance measures from being dominated by one of the attributes. For example, consider a data set where the height of a person varies from 1.5 to 1.8 m, the weight of a person varies from 90 to 300 lb, and the income of a person varies from $10,000 to $1,000,000. If a distance measure is used without scaling, the income attribute will dominate the computation of distance and thus, the assignment of class labels. A number of schemes have been developed that try to compute the weights of each individual attribute based upon a training set [32].

In addition, weights can be assigned to the training objects themselves. This can give more weight to highly reliable training objects, while reducing the impact of unreliable objects. The PEBLS system by by Cost and Salzberg [14] is a well known example of such an approach.

*K*NN classifiers are lazy learners, that is, models are not built explicitly unlike eager learners (e.g., decision trees, SVM, etc.). Thus, building the model is cheap, but classifying unknown objects is relatively expensive since it requires the computation of the $k$-nearest neighbors of the object to be labeled. This, in general, requires computing the distance of the unlabeled object to all the objects in the labeled set, which can be expensive particularly for large training sets. A number of techniques have been developed for efficient computation

of $k$-nearest neighbor distance that make use of the structure in the data to avoid having to compute distance to all objects in the training set. These techniques, which are particularly applicable for low dimensional data, can help reduce the computational cost without affecting classification accuracy.

### 8.3 Impact

*K*NN classification is an easy to understand and easy to implement classification technique. Despite its simplicity, it can perform well in many situations. In particular, a well known result by Cover and Hart [15] shows that the the error of the nearest neighbor rule is bounded above by twice the Bayes error under certain reasonable assumptions. Also, the error of the general *k*NN method asymptotically approaches that of the Bayes error and can be used to approximate it.

*K*NN is particularly well suited for multi-modal classes as well as applications in which an object can have many class labels. For example, for the assignment of functions to genes based on expression profiles, some researchers found that *k*NN outperformed SVM, which is a much more sophisticated classification scheme [48].

### 8.4 Current and future research

Although the basic *k*NN algorithm and some of its variations, such as weighted *k*NN and assigning weights to objects, are relatively well known, some of the more advanced techniques for *k*NN are much less known. For example, it is typically possible to eliminate many of the stored data objects, but still retain the classification accuracy of the *k*NN classifier. This is known as 'condensing' and can greatly speed up the classification of new objects [35]. In addition, data objects can be removed to improve classification accuracy, a process known as "editing" [88]. There has also been a considerable amount of work on the application of proximity graphs (nearest neighbor graphs, minimum spanning trees, relative neighborhood graphs, Delaunay triangulations, and Gabriel graphs) to the *k*NN problem. Recent papers by Toussaint [79,80], which emphasize a proximity graph viewpoint, provide an overview of work addressing these three areas and indicate some remaining open problems. Other important resources include the collection of papers by Dasarathy [16] and the book by Devroye et al. [18]. Finally, a fuzzy approach to *k*NN can be found in the work of Bezdek [4].

## 9 Naive Bayes

### 9.1 Introduction

Given a set of objects, each of which belongs to a known class, and each of which has a known vector of variables, our aim is to construct a rule which will allow us to assign future objects to a class, given only the vectors of variables describing the future objects. Problems of this kind, called problems of supervised classification, are ubiquitous, and many methods for constructing such rules have been developed. One very important one is the naive Bayes method—also called idiot's Bayes, simple Bayes, and independence Bayes. This method is important for several reasons. It is very easy to construct, not needing any complicated iterative parameter estimation schemes. This means it may be readily applied to huge data sets. It is easy to interpret, so users unskilled in classifier technology can understand why it is making the classification it makes. And finally, it often does surprisingly well: it may not

be the best possible classifier in any particular application, but it can usually be relied on to be robust and to do quite well. General discussion of the naive Bayes method and its merits are given in [22,33].

### 9.2 The basic principle

For convenience of exposition here, we will assume just two classes, labeled $i = 0, 1$. Our aim is to use the initial set of objects with known class memberships (the training set) to construct a score such that larger scores are associated with class 1 objects (say) and smaller scores with class 0 objects. Classification is then achieved by comparing this score with a threshold, $t$. If we define $P(i|x)$ to be the probability that an object with measurement vector $x = (x_1, \ldots, x_p)$ belongs to class $i$, then any monotonic function of $P(i|x)$ would make a suitable score. In particular, the ratio $P(1|x)/P(0|x)$ would be suitable. Elementary probability tells us that we can decompose $P(i|x)$ as proportional to $f(x|i)P(i)$, where $f(x|i)$ is the conditional distribution of $x$ for class $i$ objects, and $P(i)$ is the probability that an object will belong to class $i$ if we know nothing further about it (the 'prior' probability of class $i$). This means that the ratio becomes

$$\frac{P(1|x)}{P(0|x)} = \frac{f(x|1)P(1)}{f(x|0)P(0)}. \tag{20}$$

To use this to produce classifications, we need to estimate the $f(x|i)$ and the $P(i)$. If the training set was a random sample from the overall population, the $P(i)$ can be estimated directly from the proportion of class $i$ objects in the training set. To estimate the $f(x|i)$, the naive Bayes method assumes that the components of $x$ are independent, $f(x|i) = \prod_{j=1}^{p} f(x_j|i)$, and then estimates each of the univariate distributions $f(x_j|i)$, $j = 1, \ldots, p$; $i = 0, 1$, separately. Thus the $p$ dimensional multivariate problem has been reduced to $p$ univariate estimation problems. Univariate estimation is familiar, simple, and requires smaller training set sizes to obtain accurate estimates. This is one of the particular, indeed unique attractions of the naive Bayes methods: estimation is simple, very quick, and does not require complicated iterative estimation schemes.

If the marginal distributions $f(x_j|i)$ are discrete, with each $x_j$ taking only a few values, then the estimate $\hat{f}(x_j|i)$ is a multinomial histogram type estimator (see below)—simply counting the proportion of class $i$ objects which fall into each cell. If the $f(x_j|i)$ are continuous, then a common strategy is to segment each of them into a small number of intervals and again use multinomial estimator, but more elaborate versions based on continuous estimates (e.g. kernel estimates) are also used.

Given the independence assumption, the ratio in (20) becomes

$$\frac{P(1|x)}{P(0|x)} = \frac{\prod_{j=1}^{p} f(x_j|1)P(1)}{\prod_{j=1}^{p} f(x_j|0)P(0)} = \frac{P(1)}{P(0)} \prod_{j=1}^{p} \frac{f(x_j|1)}{f(x_j|0)}. \tag{21}$$

Now, recalling that our aim was merely to produce a score which was monotonically related to $P(i|x)$, we can take logs of (21)—log is a monotonic increasing function. This gives an alternative score

$$\ln \frac{P(1|x)}{P(0|x)} = \ln \frac{P(1)}{P(0)} + \sum_{j=1}^{p} \ln \frac{f(x_j|1)}{f(x_j|0)}. \tag{22}$$

If we define $w_j = \ln(f(x_j|1)/f(x_j|0))$ and a constant $k = \ln(P(1)/P(0))$ we see that (22) takes the form of a simple sum

$$\ln\frac{P(1|x)}{P(0|x)} = k + \sum_{j=1}^{p} w_j, \tag{23}$$

so that the classifier has a particularly simple structure.

The assumption of independence of the $x_j$ within each class implicit in the naive Bayes model might seem unduly restrictive. In fact, however, various factors may come into play which mean that the assumption is not as detrimental as it might seem. Firstly, a prior variable selection step has often taken place, in which highly correlated variables have been eliminated on the grounds that they are likely to contribute in a similar way to the separation between classes. This means that the relationships between the remaining variables might well be approximated by independence. Secondly, assuming the interactions to be zero provides an implicit regularization step, so reducing the variance of the model and leading to more accurate classifications. Thirdly, in some cases when the variables are correlated the optimal decision surface coincides with that produced under the independence assumption, so that making the assumption is not at all detrimental to performance. Fourthly, of course, the decision surface produced by the naive Bayes model can in fact have a complicated nonlinear shape: the surface is linear in the $w_j$ but highly nonlinear in the original variables $x_j$, so that it can fit quite elaborate surfaces.

### 9.3 Some extensions

Despite the above, a large number of authors have proposed modifications to the naive Bayes method in an attempt to improve its predictive accuracy.

One early proposed modification was to shrink the simplistic multinomial estimate of the proportions of objects falling into each category of each discrete predictor variable. So, if the $j$th discrete predictor variable, $x_j$, has $c_r$ categories, and if $n_{jr}$ of the total of $n$ objects fall into the $r$th category of this variable, the usual multinomial estimator of the probability that a future object will fall into this category, $n_{jr}/n$, is replaced by $(n_{jr} + c_r^{-1})/(n + 1)$. This shrinkage also has a direct Bayesian interpretation. It leads to estimates which have lower variance.

Perhaps the obvious way of easing the independence assumption is by introducing extra terms in the models of the distributions of $x$ in each class, to allow for interactions. This has been attempted in a large number of ways, but we must recognize that doing this necessarily introduces complications, and so sacrifices the basic simplicity and elegance of the naive Bayes model. Within either (or any, more generally) class, the joint distribution of $x$ is

$$f(x) = f(x_1)f(x_2|x_1)f(x_3|x_1, x_2)\cdots f(x_p|x_1, x_2, \ldots, x_{p-1}), \tag{24}$$

and this can be approximated by simplifying the conditional probabilities. The extreme arises with $f(x_i|x_1, \ldots, x_{i-1}) = f(x_i)$ for all $i$, and this is the naive Bayes method. Obviously, however, models between these two extremes can be used. For example, one could use the Markov model

$$f(x) = f(x_1)f(x_2|x_1)f(x_3|x_2)\cdots f(x_p|x_{p-1}). \tag{25}$$

This is equivalent to using a subset of two way marginal distributions instead of the univariate marginal distributions in the naive Bayes model.

Another extension to the naive Bayes model was developed entirely independently of it. This is the logistic regression model. In the above we obtained the decomposition (21) by

adopting the naive Bayes independence assumption. However, exactly the same structure for the ratio results if we model $f(x|1)$ by $g(x)\prod_{j=1}^{p} h_1(x_j)$ and $f(x|0)$ by $g(x)\prod_{j=1}^{p} h_0(x_j)$, where the function $g(x)$ is the same in each model. The ratio is thus

$$\frac{P(1|x)}{P(0|x)} = \frac{P(1)g(x)\prod_{j=1}^{p} h_1(x_j)}{P(0)g(x)\prod_{j=1}^{p} h_0(x_j)} = \frac{P(1)}{P(0)} \cdot \frac{\prod_{j=1}^{p} h_1(x_j)}{\prod_{j=1}^{p} h_0(x_j)}. \tag{26}$$

Here, the $h_i(x_j)$ do not even have to be probability density functions—it is sufficient that the $g(x)\prod_{j=1}^{p} h_i(x_j)$ are densities. The model in (26) is just as simple as the naive Bayes model, and takes exactly the same form—take logs and we have a sum as in (23)—but it is much more flexible because it does not assume independence of the $x_j$ in each class. In fact, it permits arbitrary dependence structures, via the $g(x)$ function, which can take any form. The point is, however, that this dependence is the same in the two classes, so that it cancels out in the ratio in (26). Of course, this considerable extra flexibility of the logistic regression model is not obtained without cost. Although the resulting model form is identical to the naive Bayes model form (with different parameter values, of course), it cannot be estimated by looking at the univariate marginals separately: an iterative procedure has to be used.

### 9.4 Concluding remarks on naive Bayes

The naive Bayes model is tremendously appealing because of its simplicity, elegance, and robustness. It is one of the oldest formal classification algorithms, and yet even in its simplest form it is often surprisingly effective. It is widely used in areas such as text classification and spam filtering. A large number of modifications have been introduced, by the statistical, data mining, machine learning, and pattern recognition communities, in an attempt to make it more flexible, but one has to recognize that such modifications are necessarily complications, which detract from its basic simplicity. Some such modifications are described in [27,66].

## 10 CART

The 1984 monograph, "CART: Classification and Regression Trees," co-authored by Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone, [9] represents a major milestone in the evolution of Artificial Intelligence, Machine Learning, non-parametric statistics, and data mining. The work is important for the comprehensiveness of its study of decision trees, the technical innovations it introduces, its sophisticated discussion of tree-structured data analysis, and its authoritative treatment of large sample theory for trees. While CART citations can be found in almost any domain, far more appear in fields such as electrical engineering, biology, medical research and financial topics than, for example, in marketing research or sociology where other tree methods are more popular. This section is intended to highlight key themes treated in the CART monograph so as to encourage readers to return to the original source for more detail.

### 10.1 Overview

The CART decision tree is a binary recursive partitioning procedure capable of processing continuous and nominal attributes both as targets and predictors. Data are handled in their raw form; no binning is required or recommended. Trees are grown to a maximal size without the use of a stopping rule and then pruned back (essentially split by split) to the root via cost-complexity pruning. The next split to be pruned is the one contributing least to the

overall performance of the tree on training data (and more than one split may be removed at a time). The procedure produces trees that are invariant under any order preserving transformation of the predictor attributes. The CART mechanism is intended to produce not one, but a sequence of nested pruned trees, all of which are candidate optimal trees. The "right sized" or "honest" tree is identified by evaluating the predictive performance of every tree in the pruning sequence. CART offers no internal performance measures for tree selection based on the training data as such measures are deemed suspect. Instead, tree performance is always measured on independent test data (or via cross validation) and tree selection proceeds only after test-data-based evaluation. If no test data exist and cross validation has not been performed, CART will remain agnostic regarding which tree in the sequence is best. This is in sharp contrast to methods such as C4.5 that generate preferred models on the basis of training data measures.

The CART mechanism includes automatic (optional) class balancing, automatic missing value handling, and allows for cost-sensitive learning, dynamic feature construction, and probability tree estimation. The final reports include a novel attribute importance ranking. The CART authors also broke new ground in showing how cross validation can be used to assess performance for every tree in the pruning sequence given that trees in different CV folds may not align on the number of terminal nodes. Each of these major features is discussed below.

### 10.2 Splitting rules

CART splitting rules are always couched in the form

*An instance goes left if CONDITION, and goes right otherwise,*

where the CONDITION is expressed as "attribute $X_i <= C$" for continuous attributes. For nominal attributes the CONDITION is expressed as membership in an explicit list of values. The CART authors argue that binary splits are to be preferred because (1) they fragment the data more slowly than multi-way splits, and (2) repeated splits on the same attribute are allowed and, if selected, will eventually generate as many partitions for an attribute as required. Any loss of ease in reading the tree is expected to be offset by improved performance. A third implicit reason is that the large sample theory developed by the authors was restricted to binary partitioning.

The CART monograph focuses most of its discussion on the Gini rule, which is similar to the better known entropy or information-gain criterion. For a binary (0/1) target the "Gini measure of impurity" of a node $t$ is

$$G(t) = 1 - p(t)^2 - (1 - p(t))^2, \tag{27}$$

where $p(t)$ is the (possibly weighted) relative frequency of class 1 in the node, and the improvement (gain) generated by a split of the parent node $P$ into left and right children $L$ and $R$ is

$$I(P) = G(P) - qG(L) - (1 - q)G(R). \tag{28}$$

Here, $q$ is the (possibly weighted) fraction of instances going left. The CART authors favor the Gini criterion over information gain because the Gini can be readily extended to include symmetrized costs (see below) and is computed more rapidly than information gain. (Later versions of CART have added information gain as an optional splitting rule.) They introduce the modified twoing rule, which is based on a direct comparison of the target attribute

distribution in two child nodes:

$$I(split) = \left[ .25(q(1-q))^u \sum_k |p_L(k) - p_R(k)| \right]^2, \quad (29)$$

where $k$ indexes the target classes, $p_L()$ and $p_R()$ are the probability distributions of the target in the left and right child nodes respectively, and the power term $u$ embeds a user-trollable penalty on splits generating unequal-sized child nodes. (This splitter is a modified version of Messenger and Mandell [58].) They also introduce a variant of the twoing split criterion that treats the classes of the target as ordered; ordered twoing attempts to ensure target classes represented on the left of a split are ranked below those represented on the right. In our experience the twoing criterion is often a superior performer on multi-class targets as well as on inherently difficult-to-predict (e.g. noisy) binary targets. For regression (continuous targets), CART offers a choice of Least Squares (LS) and Least Absolute Deviation (LAD) criteria as the basis for measuring the improvement of a split. Three other splitting rules for cost-sensitive learning and probability trees are discussed separately below.

### 10.3 Prior probabilities and class balancing

In its default classification mode CART always calculates class frequencies in any node relative to the class frequencies in the root. This is equivalent to automatically reweighting the data to balance the classes, and ensures that the tree selected as optimal minimizes balanced class error. The reweighting is implicit in the calculation of all probabilities and improvements and requires no user intervention; the reported sample counts in each node thus reflect the unweighted data. For a binary (0/1) target any node is classified as class 1 if, and only if,

$$N_1(node)/N_1(root) > N_0(node)/N_0(root). \quad (30)$$

This default mode is referred to as "priors equal" in the monograph. It has allowed CART users to work readily with any unbalanced data, requiring no special measures regarding class rebalancing or the introduction of manually constructed weights. To work effectively with unbalanced data it is sufficient to run CART using its default settings. Implicit reweighting can be turned off by selecting the "priors data" option, and the modeler can also elect to specify an arbitrary set of priors to reflect costs, or potential differences between training data and future data target class distributions.

### 10.4 Missing value handling

Missing values appear frequently in real world, and especially business-related databases, and the need to deal with them is a vexing challenge for all modelers. One of the major contributions of CART was to include a fully automated and highly effective mechanism for handling missing values. Decision trees require a missing value-handling mechanism at three levels: (a) during splitter evaluation, (b) when moving the training data through a node, and (c) when moving test data through a node for final class assignment. (See [63] for a clear discussion of these points.) Regarding (a), the first version of CART evaluated each splitter strictly on its performance on the subset of data for which the splitter is available. Later versions offer a family of penalties that reduce the split improvement measure as a function of the degree of missingness. For (b) and (c), the CART mechanism discovers "surrogate" or substitute splitters for every node of the tree, whether missing values occur in the training data or not. The surrogates are thus available should the tree be applied to new data that

does include missing values. This is in contrast to machines that can only learn about missing value handling from training data that include missing values. Friedman [25] suggested moving instances with missing splitter attributes into both left and right child nodes and making a final class assignment by pooling all nodes in which an instance appears. Quinlan [63] opted for a weighted variant of Friedman's approach in his study of alternative missing value-handling methods. Our own assessments of the effectiveness of CART surrogate performance in the presence of missing data are largely favorable, while Quinlan remains agnostic on the basis of the approximate surrogates he implements for test purposes [63]. Friedman et al. [26] noted that 50% of the CART code was devoted to missing value handling; it is thus unlikely that Quinlan's experimental version properly replicated the entire CART surrogate mechanism.

In CART the missing value handling mechanism is fully automatic and locally adaptive at every node. At each node in the tree the chosen splitter induces a binary partition of the data (e.g., X1 <= c1 and X1 > c1). A surrogate splitter is a single attribute Z that can predict this partition where the surrogate itself is in the form of a binary splitter (e.g., Z <= d and Z > d). In other words, every splitter becomes a new target which is to be predicted with a single split binary tree. Surrogates are ranked by an association score that measures the advantage of the surrogate over the default rule predicting that all cases go to the larger child node. To qualify as a surrogate, the variable must outperform this default rule (and thus it may not always be possible to find surrogates). When a missing value is encountered in a CART tree the instance is moved to the left or the right according to the top-ranked surrogate. If this surrogate is also missing then the second ranked surrogate is used instead, (and so on). If all surrogates are missing the default rule assigns the instance to the larger child node (possibly adjusting node sizes for priors). Ties are broken by moving an instance to the left.

### 10.5 Attribute importance

The importance of an attribute is based on the sum of the improvements in all nodes in which the attribute appears as a splitter (weighted by the fraction of the training data in each node split). Surrogates are also included in the importance calculations, which means that even a variable that never splits a node may be assigned a large importance score. This allows the variable importance rankings to reveal variable masking and nonlinear correlation among the attributes. Importance scores may optionally be confined to splitters and comparing the splitters-only and the full importance rankings is a useful diagnostic.

### 10.6 Dynamic feature construction

Friedman [25] discussed the automatic construction of new features within each node and, for the binary target, recommends adding the single feature

$$x * w,$$

where $x$ is the original attribute vector and $w$ is a scaled difference of means vector across the two classes (the direction of the Fisher linear discriminant). This is similar to running a logistic regression on all available attributes in the node and using the estimated logit as a predictor. In the CART monograph, the authors discuss the automatic construction of linear combinations that include feature selection; this capability has been available from the first release of the CART software. BFOS also present a method for constructing Boolean

combinations of splitters within each node, a capability that has not been included in the released software.

### 10.7 Cost-sensitive learning

Costs are central to statistical decision theory but cost-sensitive learning received only modest attention before Domingos [21]. Since then, several conferences have been devoted exclusively to this topic and a large number of research papers have appeared in the subsequent scientific literature. It is therefore useful to note that the CART monograph introduced two strategies for cost-sensitive learning and the entire mathematical machinery describing CART is cast in terms of the costs of misclassification. The cost of misclassifying an instance of class $i$ as class $j$ is $C(i, j)$ and is assumed to be equal to 1 unless specified otherwise; $C(i, i) = 0$ for all $i$. The complete set of costs is represented in the matrix $C$ containing a row and a column for each target class. Any classification tree can have a total cost computed for its terminal node assignments by summing costs over all misclassifications. The issue in cost-sensitive learning is to induce a tree that takes the costs into account during its growing and pruning phases.

The first and most straightforward method for handling costs makes use of weighting: instances belonging to classes that are costly to misclassify are weighted upwards, with a common weight applying to all instances of a given class, a method recently rediscovered by Ting [78]. As implemented in CART. the weighting is accomplished transparently so that all node counts are reported in their raw unweighted form. For multi-class problems BFOS suggested that the entries in the misclassification cost matrix be summed across each row to obtain relative class weights that approximately reflect costs. This technique ignores the detail within the matrix but has now been widely adopted due to its simplicity. For the Gini splitting rule the CART authors show that it is possible to embed the entire cost matrix into the splitting rule, but only after it has been symmetrized. The "symGini" splitting rule generates trees sensitive to the difference in costs $C(i, j)$ and $C(i, k)$, and is most useful when the symmetrized cost matrix is an acceptable representation of the decision maker's problem. In contrast, the instance weighting approach assigns a single cost to all misclassifications of objects of class $i$. BFOS report that pruning the tree using the full cost matrix is essential to successful cost-sensitive learning.

### 10.8 Stopping rules, pruning, tree sequences, and tree selection

The earliest work on decision trees did not allow for pruning. Instead, trees were grown until they encountered some stopping condition and the resulting tree was considered final. In the CART monograph the authors argued that no rule intended to stop tree growth can guarantee that it will not miss important data structure (e.g., consider the two-dimensional XOR problem). They therefore elected to grow trees without stopping. The resulting overly large tree provides the raw material from which a final optimal model is extracted.

The pruning mechanism is based strictly on the training data and begins with a cost-complexity measure defined as

$$R_a(T) = R(T) + a|T|, \tag{31}$$

where $R(T)$ is the training sample cost of the tree, $|T|$ is the number of terminal nodes in the tree and $a$ is a penalty imposed on each node. If $a = 0$ then the minimum cost-complexity tree is clearly the largest possible. If $a$ is allowed to progressively increase the minimum cost-complexity tree will become smaller since the splits at the bottom of the tree that reduce

$R(T)$ the least will be cut away. The parameter a is progressively increased from 0 to a value sufficient to prune away all splits. BFOS prove that any tree of size $Q$ extracted in this way will exhibit a cost $R(Q)$ that is minimum within the class of all trees with $Q$ terminal nodes.

The optimal tree is defined as that tree in the pruned sequence that achieves minimum cost on test data. Because test misclassification cost measurement is subject to sampling error, uncertainty always remains regarding which tree in the pruning sequence is optimal. BFOS recommend selecting the "1 SE" tree that is the smallest tree with an estimated cost within 1 standard error of the minimum cost (or "0 SE") tree.

### 10.9 Probability trees

Probability trees have been recently discussed in a series of insightful articles elucidating their properties and seeking to improve their performance (see Provost and Domingos 2000). The CART monograph includes what appears to be the first detailed discussion of probability trees and the CART software offers a dedicated splitting rule for the growing of "class probability trees." A key difference between classification trees and probability trees is that the latter want to keep splits that generate terminal node children assigned to the same class whereas the former will not (such a split accomplishes nothing so far as classification accuracy is concerned). A probability tree will also be pruned differently than its counterpart classification tree, therefore, the final structure of the two optimal trees can be somewhat different (although the differences are usually modest). The primary drawback of probability trees is that the probability estimates based on training data in the terminal nodes tend to be biased (e.g., towards 0 or 1 in the case of the binary target) with the bias increasing with the depth of the node. In the recent ML literature the use of the LaPlace adjustment has been recommended to reduce this bias (Provost and Domingos 2002). The CART monograph offers a somewhat more complex method to adjust the terminal node estimates that has rarely been discussed in the literature. Dubbed the "Breiman adjustment", it adjusts the estimated misclassification rate $r^*(t)$ of any terminal node upwards by

$$r^*(t) = r(t) + e/(q(t) + S) \tag{32}$$

where $r(t)$ is the train sample estimate within the node, $q(t)$ is the fraction of the training sample in the node and $S$ and $e$ are parameters that are solved for as a function of the difference between the train and test error rates for a given tree. In contrast to the LaPlace method, the Breiman adjustment does not depend on the raw predicted probability in the node and the adjustment can be very small if the test data show that the tree is not overfit. Bloch et al. [5] reported very good performance for the Breiman adjustment in a series of empirical experiments.

### 10.10 Theoretical foundations

The earliest work on decision trees was entirely atheoretical. Trees were proposed as methods that appeared to be useful and conclusions regarding their properties were based on observing tree performance on a handful of empirical examples. While this approach remains popular in Machine Learning, the recent tendency in the discipline has been to reach for stronger theoretical foundations. The CART monograph tackles theory with sophistication, offering important technical insights and proofs for several key results. For example, the authors derive the expected misclassification rate for the maximal (largest possible) tree, showing that it is bounded from above by twice the Bayes rate. The authors also discuss the bias variance tradeoff in trees and show how the bias is affected by the number of attributes.

Based largely on the prior work of CART co-authors Richard Olshen and Charles Stone, the final three chapters of the monograph relate CART to theoretical work on nearest neighbors and show that as the sample size tends to infinity the following hold: (1) the estimates of the regression function converge to the true function, and (2) the risks of the terminal nodes converge to the risks of the corresponding Bayes rules. In other words, speaking informally, with large enough samples the CART tree will converge to the true function relating the target to its predictors and achieve the smallest cost possible (the Bayes rate). Practically speaking. such results may only be realized with sample sizes far larger than in common use today.

### 10.11 Selected biographical details

CART is often thought to have originated from the field of Statistics but this is only partially correct. Jerome Friedman completed his PhD in Physics at UC Berkeley and became leader of the Numerical Methods Group at the Stanford Linear Accelerator Center in 1972, where he focused on problems in computation. One of his most influential papers from 1975 presents a state-of-the-art algorithm for high speed searches for nearest neighbors in a database. Richard Olshen earned his BA at UC Berkeley and PhD in Statistics at Yale and focused his earliest work on large sample theory for recursive partitioning. He began his collaboration with Friedman after joining the Stanford Linear Accelerator Center in 1974. Leo Breiman earned his BA in Physics at the California Institute of Technology, his PhD in Mathematics at UC Berkeley, and made notable contributions to pure probability theory (Breiman, 1968) [7] while a Professor at UCLA. In 1967 he left academia for 13 years to work as an industrial consultant; during this time he encountered the military data analysis problems that inspired his contributions to CART. An interview with Leo Breiman discussing his career and personal life appears in [60].

Charles Stone earned his BA in mathematics at the California Institute of Technology, and his PhD in Statistics at Stanford. He pursued probability theory in his early years as an academic and is the author of several celebrated papers in probability theory and nonparametric regression. He worked with Breiman at UCLA and was drawn by Breiman into the research leading to CART in the early 1970s. Breiman and Friedman first met at an Interface conference in 1976, which shortly led to collaboration involving all four co-authors. The first outline of their book was produced in a memo dated 1978 and the completed CART monograph was published in 1984.

The four co-authors have each been distinguished for their work outside of CART. Stone and Breiman were elected to the National Academy of Sciences (in 1993 and 2001, respectively) and Friedman was elected to the American Academy of Arts and Sciences in 2006. The specific work for which they were honored can be found on the respective academy websites. Olshen is a Fellow of the Institute of Mathematical Statistics, a Fellow of the IEEE, and Fellow, American Association for the Advancement of Science.

## 11 Concluding remarks

Data mining is a broad area that integrates techniques from several fields including machine learning, statistics, pattern recognition, artificial intelligence, and database systems, for the analysis of large volumes of data. There have been a large number of data mining algorithms rooted in these fields to perform different data analysis tasks. The 10 algorithms identified by the IEEE International Conference on Data Mining (ICDM) and presented in

this article are among the most influential algorithms for classification [47,51,77], clustering [11,31,40,44–46], statistical learning [28,76,92], association analysis [2,6,13,50,54,74], and link mining.

With a formal tie with the ICDM conference, *Knowledge and Information Systems* has been publishing the best papers from ICDM every year, and several of the above papers cited for classification, clustering, statistical learning, and association analysis were selected by the previous years' ICDM program committees for journal publication in *Knowledge and Information Systems* after their revisions and expansions. We hope this survey paper can inspire more researchers in data mining to further explore these top-10 algorithms, including their impact and new research issues.

**Acknowledgments** The initiative of identifying the top-10 data mining algorithms started in May 2006 out of a discussion between Dr. Jiannong Cao in the Department of Computing at the Hong Kong Polytechnic University (PolyU) and Dr. Xindong Wu, when Dr. Wu was giving a seminar on 10 Challenging Problems in Data Mining Research [89] at PolyU. Dr. Wu and Dr. Kumar continued this discussion at KDD-06 in August 2006 with various people, and received very enthusiastic support. Naila Elliott in the Department of Computer Science and Engineering at the University of Minnesota collected and compiled the algorithm nominations and voting results in the 3-step identification process. Yan Zhang in the Department of Computer Science at the University of Vermont converted the 10 section submissions in different formats into the same LaTeX format, which was a time-consuming process.

## References

1. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceedings of the 20th VLDB conference, pp 487–499
2. Ahmed S, Coenen F, Leng PH (2006) Tree-based partitioning of date for association rule mining. Knowl Inf Syst 10(3):315–331
3. Banerjee A, Merugu S, Dhillon I, Ghosh J (2005) Clustering with Bregman divergences. J Mach Learn Res 6:1705–1749
4. Bezdek JC, Chuah SK, Leep D (1986) Generalized k-nearest neighbor rules. Fuzzy Sets Syst 18(3):237–256. http://dx.doi.org/10.1016/0165-0114(86)90004-7
5. Bloch DA, Olshen RA, Walker MG (2002) Risk estimation for classification trees. J Comput Graph Stat 11:263–288
6. Bonchi F, Lucchese C (2006) On condensed representations of constrained frequent patterns. Knowl Inf Syst 9(2):180–201
7. Breiman L (1968) Probability theory. Addison-Wesley, Reading. Republished (1991) in Classics of mathematics. SIAM, Philadelphia
8. Breiman L (1999) Prediction games and arcing classifiers. Neural Comput 11(7):1493–1517
9. Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) Classification and regression trees. Wadsworth, Belmont
10. Brin S, Page L (1998) The anatomy of a large-scale hypertextual Web Search Sngine. Comput Networks 30(1–7):107–117
11. Chen JR (2007) Making clustering in delay-vector space meaningful. Knowl Inf Syst 11(3):369–385
12. Cheung DW, Han J, Ng V, Wong CY (1996) Maintenance of discovered association rules in large databases: an incremental updating technique. In: Proceedings of the ACM SIGMOD international conference on management of data, pp. 13–23
13. Chi Y, Wang H, Yu PS, Muntz RR (2006) Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. Knowl Inf Syst 10(3):265–294
14. Cost S, Salzberg S (1993) A weighted nearest neighbor algorithm for learning with symbolic features. Mach Learn 10:57.78 (PEBLS: Parallel Examplar-Based Learning System)
15. Cover T, Hart P (1967) Nearest neighbor pattern classification. IEEE Trans Inform Theory 13(1):21–27
16. Dasarathy BV (ed) (1991) Nearest neighbor (NN) norms: NN pattern classification techniques. IEEE Computer Society Press
17. Dempster AP, Laird NM, Rubin DB (1977) Maximum likelihood from incomplete data via the EM algorithm (with discussion). J Roy Stat Soc B 39:1–38

18. Devroye L, Gyorfi L, Lugosi G (1996) A probabilistic theory of pattern recognition. Springer, New York. ISBN 0-387-94618-7
19. Dhillon IS, Guan Y, Kulis B (2004) Kernel k-means: spectral clustering and normalized cuts. KDD 2004, pp 551–556
20. Dietterich TG (1997) Machine learning: Four current directions. AI Mag 18(4):97–136
21. Domingos P (1999) MetaCost: A general method for making classifiers cost-sensitive. In: Proceedings of the fifth international conference on knowledge discovery and data mining, pp 155–164
22. Domingos P, Pazzani M (1997) On the optimality of the simple Bayesian classifier under zero-one loss. Mach Learn 29:103–130
23. Fix E, Hodges JL, Jr (1951) Discriminatory analysis, nonparametric discrimination. USAF School of Aviation Medicine, Randolph Field, Tex., Project 21-49-004, Rept. 4, Contract AF41(128)-31, February 1951
24. Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55(1):119–139
25. Friedman JH, Bentley JL, Finkel RA (1977) An algorithm for finding best matches in logarithmic time. ACM Trans. Math. Software 3, 209. Also available as Stanford Linear Accelerator Center Rep. SIX-PUB-1549, February 1975
26. Friedman JH, Kohavi R, Yun Y (1996) Lazy decision trees. In: Proceedings of the thirteenth national conference on artificial intelligence, San Francisco, CA. AAAI Press/MIT Press, pp. 717–724
27. Friedman N, Geiger D, Goldszmidt M (1997) Bayesian network classifiers. Mach Learn 29:131–163
28. Fung G, Stoeckel J (2007) SVM feature selection for classification of SPECT images of Alzheimer's disease using spatial information. Knowl Inf Syst 11(2):243–258
29. Gates GW (1972) The reduced nearest neighbor rule. IEEE Trans Inform Theory 18:431–433
30. Golub GH, Van Loan CF (1983) Matrix computations. The Johns Hopkins University Press
31. Gondek D, Hofmann T (2007) Non-redundant data clustering. Knowl Inf Syst 12(1):1–24
32. Han E (1999) Text categorization using weight adjusted k-nearest neighbor classification. PhD thesis, University of Minnesota, October 1999
33. Hand DJ, Yu K (2001) Idiot's Bayes—not so stupid after all?. Int Stat Rev 69:385–398
34. Gray RM, Neuhoff DL (1998) Quantization. IEEE Trans Inform Theory 44(6):2325–2384
35. Hart P (1968) The condensed nearest neighbor rule. IEEE Trans Inform Theory 14:515–516
36. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceedings of ACM SIGMOD international conference on management of data, pp 1–12
37. Hastie T, Tibshirani R (1996) Discriminant adaptive nearest neighbor classification. IEEE Trans Pattern Anal Mach Intell 18(6):607–616
38. Friedman J, Hastie T, Tibshirani R (2000) Additive logistic regression: a statistical view of boosting with discussions. Ann Stat 28(2):337–407
39. Herbrich R, Graepel T, Obermayer K (2000) Rank boundaries for ordinal regression. Adv Mar Classif pp 115–132
40. Hu T, Sung SY (2006) Finding centroid clusterings with entropy-based criteria. Knowl Inf Syst 10(4):505–514
41. Hunt EB, Marin J, Stone PJ (1966) Experiments in induction. Academic Press, New York
42. Inokuchi A, Washio T, Motoda H (2005) General framework for mining frequent subgraphs from labeled graphs. Fundament Inform 66(1-2):53–82
43. Jain AK, Dubes RC (1988) Algorithms for clustering data. Prentice-Hall, Englewood Cliffs
44. Jin R, Goswami A, Agrawal G (2006) Fast and exact out-of-core and distributed *k*-means clustering. Knowl Inf Syst 10(1):17–40
45. Kobayashi M, Aono M (2006) Exploring overlapping clusters using dynamic re-scaling and sampling. Knowl Inf Syst 10(3):295–313
46. Koga H, Ishibashi T, Watanabe T (2007) Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing. Knowl Inf Syst 12(1):25–53
47. Kukar M (2006) Quality assessment of individual classifications in machine learning and data mining. Knowl Inf Syst 9(3):364–384
48. Kuramochi M, Karypis G (2005) Gene Classification using Expression Profiles: A Feasibility Study. Int J Artif Intell Tools 14(4):641–660
49. Langville AN, Meyer CD (2006) Google's PageRank and beyond: the science of search engine rankings. Princeton University Press, Princeton
50. Leung CW-k, Chan SC-f, Chung F-L (2006) A collaborative filtering framework based on fuzzy association rules and multiple-level similarity. Knowl Inf Syst 10(3):357–381
51. Li T, Zhu S, Ogihara M (2006) Using discriminant analysis for multi-class classification: an experimental investigation. Knowl Inf Syst 10(4):453–472

52. Liu B (2007) Web data mining: exploring hyperlinks, contents and usage Data. Springer, Heidelberg
53. Lloyd SP (1957) Least squares quantization in PCM. Unpublished Bell Lab. Tech. Note, portions presented at the Institute of Mathematical Statistics Meeting Atlantic City, NJ, September 1957. Also, IEEE Trans Inform Theory (Special Issue on Quantization), vol IT-28, pp 129–137, March 1982
54. Leung CK-S, Khan QI, Li Z, Hoque T (2007) CanTree: a canonical-order tree for incremental frequent-pattern mining. Knowl Inf Syst 11(3):287–311
55. McLachlan GJ (1987) On bootstrapping the likelihood ratio test statistic for the number of components in a normal mixture.. Appl Stat 36:318–324
56. McLachlan GJ, Krishnan T (1997) The EM algorithm and extensions. Wiley, New York
57. McLachlan GJ, Peel D (2000) Finite Mixture Models. Wiley, New York
58. Messenger RC, Mandell ML (1972) A model search technique for predictive nominal scale multivariate analysis. J Am Stat Assoc 67:768–772
59. Morishita S, Sese J (2000) Traversing lattice itemset with statistical metric pruning. In: Proceedings of PODS'00, pp 226–236
60. Olshen R (2001) A conversation with Leo Breiman. Stat Sci 16(2):184–198
61. Page L, Brin S, Motwami R, Winograd T (1999) The PageRank citation ranking: bringing order to the Web. Technical Report 1999–0120, Computer Science Department, Stanford University
62. Quinlan JR (1979) Discovering rules by induction from large collections of examples. In: Michie D (ed), Expert systems in the micro electronic age. Edinburgh University Press, Edinburgh
63. Quinlan R (1989) Unknown attribute values in induction. In: Proceedings of the sixth international workshop on machine learning, pp. 164–168
64. Quinlan JR (1993) C4.5: Programs for machine learning. Morgan Kaufmann Publishers, San Mateo
65. Reyzin L, Schapire RE (2006) How boosting the margin can also boost classifier complexity. In: Proceedings of the 23rd international conference on machine learning. Pittsburgh, PA, pp. 753–760
66. Ridgeway G, Madigan D, Richardson T (1998) Interpretable boosted naive Bayes classification. In: Agrawal R, Stolorz P, Piatetsky-Shapiro G (eds) Proceedings of the fourth international conference on knowledge discovery and data mining.. AAAI Press, Menlo Park pp 101–104
67. Schapire RE (1990) The strength of weak learnability. Mach Learn 5(2):197–227
68. Schapire RE, Freund Y, Bartlett P, Lee WS (1998) Boosting the margin: A new explanation for the effectiveness of voting methods. Ann Stat 26(5):1651–1686
69. Schapire RE, Singer Y (1999) Improved boosting algorithms using confidence-rated predictions. Mach Learn 37(3):297–336
70. Scholkopf B, Smola AJ (2002) Learning with kernels. MIT Press
71. Seidl T, Kriegel H (1998) Optimal multi-step k-nearest neighbor search. In: Tiwary A, Franklin M (eds) Proceedings of the 1998 ACM SIGMOD international conference on management of data, Seattle, Washington, United States, 1–4 June, 1998. ACM Press, New York pp 154–165
72. Srikant R, Agrawal R (1995) Mining generalized association rules. In: Proceedings of the 21st VLDB conference. pp. 407–419
73. Steinbach M, Karypis G, Kumar V (2000) A comparison of document clustering techniques. In: Proceedings of the KDD Workshop on Text Mining
74. Steinbach M, Kumar V (2007) Generalizing the notion of confidence. Knowl Inf Syst 12(3):279–299
75. Tan P-N, Steinbach M, Kumar V (2006) Introduction to data mining. Pearson Addison-Wesley
76. Tao D, Li X, Wu X, Hu W, Maybank SJ (2007) Supervised tensor learning. Knowl Inf Syst 13(1):1–42
77. Thabtah FA, Cowling PI, Peng Y (2006) Multiple labels associative classification. Knowl Inf Syst 9(1):109–129
78. Ting KM (2002) An instance-weighting method to induce cost-sensitive trees. IEEE Trans Knowl Data Eng 14:659–665
79. Toussaint GT (2002) Proximity graphs for nearest neighbor decision rules: recent progress. In: Interface-2002, 34th symposium on computing and statistics (theme: Geoscience and Remote Sensing). Ritz-Carlton Hotel, Montreal, Canada, 17–20 April, 2002
80. Toussaint GT (2002) Open problems in geometric methods for instance-based learning. JCDCG 273–283
81. Tsang IW, Kwok JT, Cheung P-M (2005) Core vector machines: Fast SVM training on very large data sets. J Mach Learn Res 6:363–392
82. Uno T, Asai T, Uchida Y, Arimura H (2004) An efficient algorithm for enumerating frequent closed patterns in transaction databases. In: Proc. of the 7th international conference on discovery science. LNAI vol 3245, Springe, Heidelberg, pp 16–30
83. Vapnik V (1995) The nature of statistical learning theory. Springer, New York
84. Viola P, Jones M (2001) Rapid object detection using a boosted cascade of simple features. In: Proceedings of the IEEE computer society conference on computer vision and pattern recognition. pages 511–518, Kauai, HI

85. Washio T, Nakanishi K, Motoda H (2005) Association rules based on levelwise subspace clustering. In: Proceedings. of 9th European conference on principles and practice of knowledge discovery in databases. LNAI, vol 3721, pp. 692–700 Springer, Heidelberg
86. Wasserman S, Raust K (1994) Social network analysis. Cambridge University Press, Cambridge
87. Wettschereck D, Aha D, Mohri T (1997) A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. Artif Intell Rev 11:273–314
88. Wilson DL (1972) Asymptotic properties of nearest neighbor rules using edited data. IEEE Trans Syst Man Cyberne 2:408–420
89. Yang Q, Wu X (2006) 10 challenging problems in data mining research. Int J Inform Technol Decis Making 5(4):597–604
90. Yan X, Han J (2002) gSpan: Graph-based substructure pattern mining. In: Proceedings of ICDM'02, pp 721–724
91. Yu PS, Li X, Liu B (2005) Adding the temporal dimension to search—a case study in publication search. In: Proceedings of Web Intelligence (WI'05)
92. Zhang J, Kang D-K, Silvescu A, Honavar V (2006) Learning accurate and concise naïve Bayes classifiers from attribute value taxonomies and data. Knowl Inf Syst 9(2):157–179

# MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

*Google, Inc.*

## Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

## 1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

for a rewrite of our production indexing system. Section 7 discusses related and future work.

## 2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

*Map*, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key $I$ and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key $I$ and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

### 2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

### 2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

| | | |
|---|---|---|
| map | `(k1,v1)` | $\rightarrow$ `list(k2,v2)` |
| reduce | `(k2,list(v2))` | $\rightarrow$ `list(v2)` |

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

### 2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

**Distributed Grep:** The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

**Count of URL Access Frequency:** The map function processes logs of web page requests and outputs $\langle$`URL`, `1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL`, `total count`$\rangle$ pair.

**Reverse Web-Link Graph:** The map function outputs $\langle$`target`, `source`$\rangle$ pairs for each link to a `target` URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle$`target`, $list$(`source`)$\rangle$

**Term-Vector per Host:** A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname`, `term vector`$\rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle$`hostname`, `term vector`$\rangle$ pair.

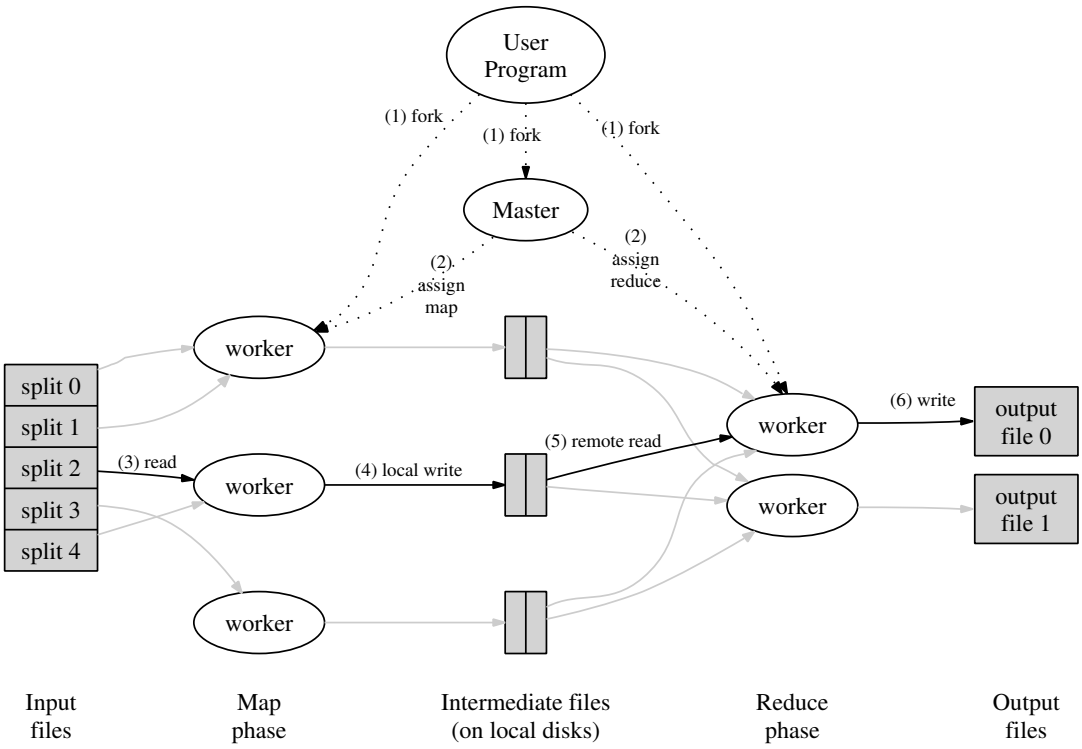



Figure 1: Execution overview

**Inverted Index:** The map function parses each document, and emits a sequence of $\langle$`word, document ID`$\rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle$`word`, $list$(`document ID`)$\rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

**Distributed Sort:** The map function extracts the key from each record, and emits a $\langle$`key, record`$\rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

# 3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

## 3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of $M$ *splits*. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into $R$ pieces using a partitioning function (e.g., $hash(key) \bmod R$). The number of partitions ($R$) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into $M$ pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are $M$ map tasks and $R$ reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into $R$ regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the `MapReduce` call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the $R$ output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these $R$ output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

## 3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the $R$ intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

## 3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

### Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker $A$ and then later executed by worker $B$ (because $A$ failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker $A$ will read the data from worker $B$.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

#### Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

#### Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces $R$ such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the $R$ temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of $R$ files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task $R_1$ is equivalent to the output for $R_1$ produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task $R_2$ may correspond to the output for $R_2$ produced by a different sequential execution of the non-deterministic program.

Consider map task $M$ and reduce tasks $R_1$ and $R_2$. Let $e(R_i)$ be the execution of $R_i$ that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of $M$ and $e(R_2)$ may have read the output produced by a different execution of $M$.

### 3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

### 3.5 Task Granularity

We subdivide the map phase into $M$ pieces and the reduce phase into $R$ pieces, as described above. Ideally, $M$ and $R$ should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large $M$ and $R$ can be in our implementation, since the master must make $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (The constant factors for memory usage are small however: the $O(M * R)$ piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, $R$ is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose $M$ so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make $R$ a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

### 3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

## 4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

### 4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire ($R$). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key) \bmod R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey)) \bmod R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

### 4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

### 4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

### 4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

## 4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

## 4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

## 4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

## 4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

## 4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
    EmitIntermediate(w, "1");
```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

# 5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

## 5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

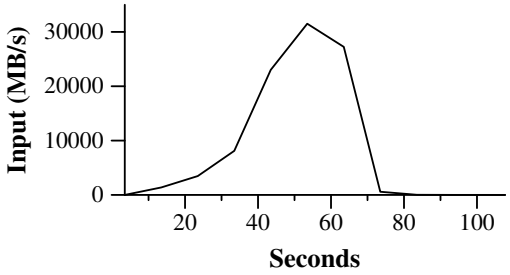



Figure 2: Data transfer rate over time

## 5.2 Grep

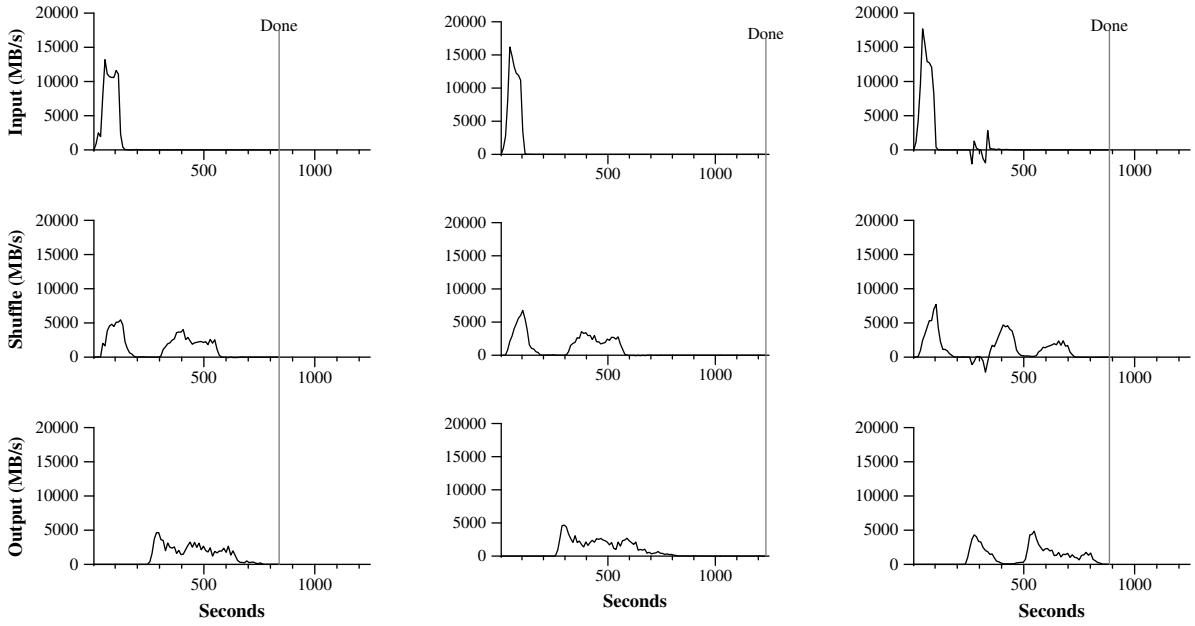
The *grep* program scans through $10^{10}$ 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

## 5.3 Sort

The *sort* program sorts $10^{10}$ 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the




(a) Normal execution (b) No backup tasks (c) 200 tasks killed

Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of $R$ pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

## 5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

## 5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

# 6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

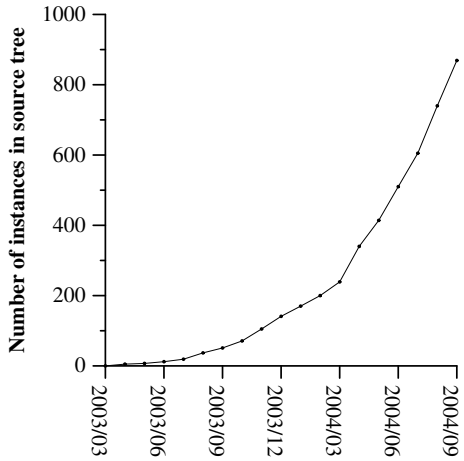



Figure 4: MapReduce instances over time

| | |
|---|---|
| Number of jobs | 29,423 |
| Average job completion time | 634 secs |
| Machine days used | 79,186 days |
| Input data read | 3,288 TB |
| Intermediate data produced | 758 TB |
| Output data written | 193 TB |
| Average worker machines per job | 157 |
| Average worker deaths per job | 1.2 |
| Average map tasks per job | 3,351 |
| Average reduce tasks per job | 55 |
| Unique *map* implementations | 395 |
| Unique *reduce* implementations | 269 |
| Unique *map/reduce* combinations | 426 |

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

## 6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

## 7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an $N$ element array in $\log N$ time on $N$ processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of $R$ reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

## 8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

## Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

## References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

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

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

## A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(start,i-start),"1");
    }
  }
};
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }

    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);

  MapReduceSpecification spec;

  // Store list of input files into "spec"
  for (int i = 1; i < argc; i++) {
    MapReduceInput* input = spec.add_input();
    input->set_format("text");
    input->set_filepattern(argv[i]);
    input->set_mapper_class("WordCounter");
  }

  // Specify the output files:
  //    /gfs/test/freq-00000-of-00100
  //    /gfs/test/freq-00001-of-00100
  //    ...
  MapReduceOutput* out = spec.output();
  out->set_filebase("/gfs/test/freq");
  out->set_num_tasks(100);
  out->set_format("text");
  out->set_reducer_class("Adder");

  // Optional: do partial sums within map
  // tasks to save network bandwidth
  out->set_combiner_class("Adder");

  // Tuning parameters: use at most 2000
  // machines and 100 MB of memory per task
  spec.set_machines(2000);
  spec.set_map_megabytes(100);
  spec.set_reduce_megabytes(100);

  // Now run it
  MapReduceResult result;
  if (!MapReduce(spec, &result)) abort();

  // Done: 'result' structure contains info
  // about counters, time taken, number of
  // machines used, etc.

  return 0;
}
```

# The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

## ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

## Categories and Subject Descriptors

D [**4**]: 3—*Distributed file systems*

## General Terms

Design, reliability, performance, measurement

## Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com*.


*SOSP'03,* October 19–22, 2003, Bolton Landing, New York, USA.


## 1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

For example, we have relaxed GFS's consistency model to vastly simplify the file system without imposing an onerous burden on the applications. We have also introduced an atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them. These will be discussed in more details later in the paper.

Multiple GFS clusters are currently deployed for different purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

# 2. DESIGN OVERVIEW

## 2.1 Assumptions

In designing a file system for our needs, we have been guided by assumptions that offer both challenges and opportunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

- The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
- The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.
- The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. Our files are often used as producer-consumer queues or for many-way merging. Hundreds of producers, running one per machine, will concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.
- High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

## 2.2 Interface

GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. We support the usual operations to *create*, *delete*, *open*, *close*, *read*, and *write* files.

Moreover, GFS has *snapshot* and *record append* operations. Snapshot creates a copy of a file or a directory tree at low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append. It is useful for implementing multi-way merge results and producer-consumer queues that many clients can simultaneously append to without additional locking. We have found these types of files to be invaluable in building large distributed applications. Snapshot and record append are discussed further in Sections 3.4 and 3.3 respectively.

## 2.3 Architecture

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size *chunks*. Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in *HeartBeat* messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

## 2.4 Single Master

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placement

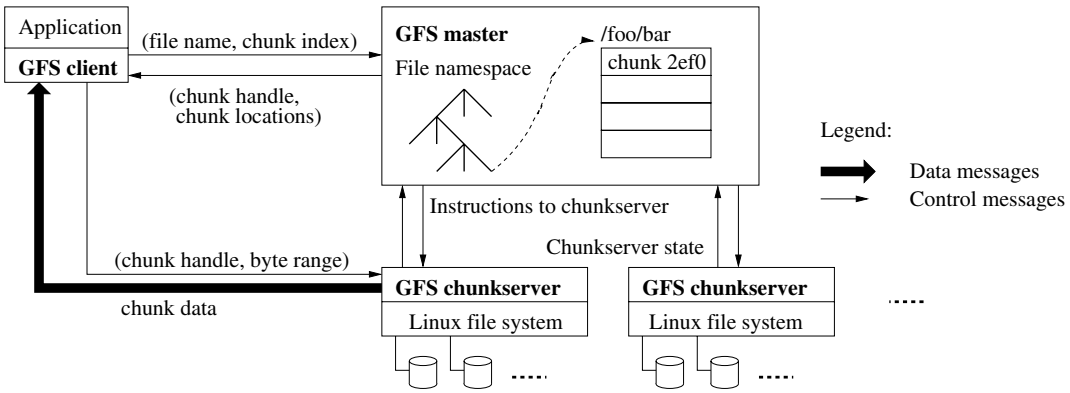



**Figure 1: GFS Architecture**

and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with reference to Figure 1. First, using the fixed chunk size, the client translates the file name and byte offset specified by the application into a chunk index within the file. Then, it sends the master a request containing the file name and chunk index. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key.

The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested. This extra information sidesteps several future client-master interactions at practically no extra cost.

## 2.5 Chunk Size

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunkserver and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages. First, it reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially. Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time. The few chunkservers storing this executable were overloaded by hundreds of simultaneous requests. We fixed this problem by storing such executables with a higher replication factor and by making the batch-queue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

## 2.6 Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas. All metadata is kept in the master's memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an *operation log* stored on the master's local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location information persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

### *2.6.1 In-Memory Data Structures*

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver failures, and chunk migration to balance load and disk space

usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less then 64 bytes per file because it stores file names compactly using prefix compression.

If necessary to support even larger file systems, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility we gain by storing the metadata in memory.

### 2.6.2 Chunk Locations

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with regular *HeartBeat* messages.

We initially attempted to keep chunk location information persistently at the master, but we decided that it was much simpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.

Another way to understand this design decision is to realize that a chunkserver has the final word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

### 2.6.3 Operation Log

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created.

Since the operation log is critical, we must store it reliably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.

The master recovers its file system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the limited number of log records after that. The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without extra parsing. This further speeds up recovery and improves availability.

Because building a checkpoint can take a while, the master's internal state is structured in such a way that a new checkpoint can be created without delaying incoming mutations. The master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million files. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint and subsequent log files. Older checkpoints and log files can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not affect correctness because the recovery code detects and skips incomplete checkpoints.

| | Write | Record Append |
|---|---|---|
| Serial success | *defined* | *defined* interspersed with *inconsistent* |
| Concurrent successes | *consistent* but *undefined* | |
| Failure | *inconsistent* | |

**Table 1: File Region State After Mutation**

## 2.7 Consistency Model

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement. We now discuss GFS's guarantees and what they mean to applications. We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

### 2.7.1 Guarantees by GFS

File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1); the master's operation log defines a global total order of these operations (Section 2.6.3).

The state of a file region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summarizes the result. A file region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writers, the affected region is defined (and by implication consistent): all clients will always see what the mutation has written. Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times. We describe below how our applications can distinguish defined regions from undefined

regions. The applications do not need to further distinguish between different kinds of undefined regions.

Data mutations may be *writes* or *record appends*. A write causes data to be written at an application-specified file offset. A record append causes data (the "record") to be appended *atomically at least once* even in the presence of concurrent mutations, but at an offset of GFS's choosing (Section 3.3). (In contrast, a "regular" append is merely a write at an offset that the client believes to be the current end of file.) The offset is returned to the client and marks the beginning of a defined region that contains the record. In addition, GFS may insert padding or record duplicates in between. They occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.

After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Since clients cache chunk locations, they may read from a stale replica before that information is refreshed. This window is limited by the cache entry's timeout and the next open of the file, which purges from the cache all chunk information for that file. Moreover, as most of our files are append-only, a stale replica usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get current chunk locations.

Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming (Section 5.2). Once a problem surfaces, the data is restored from valid replicas as soon as possible (Section 4.3). A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

### 2.7.2 Implications for Applications

GFS applications can accommodate the relaxed consistency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate files by appending rather than overwriting. In one typical use, a writer generates a file from beginning to end. It atomically renames the file to a permanent name after writing all the data, or periodically checkpoints how much has been successfully written. Checkpoints may also include application-level checksums. Readers verify and process only the file region up to the last checkpoint, which is known to be in the defined state. Regardless of consistency and concurrency issues, this approach has served us well. Appending is far more efficient and more resilient to application failures than random writes. Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from the application's perspective.

In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append's append-at-least-once semantics preserves each writer's output. Readers deal with the occasional padding and duplicates as follows. Each record prepared by the writer contains extra information like checksums so that its validity can be verified. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent operations), it can filter them out using unique identifiers in the records, which are often needed anyway to name corresponding application entities such as web documents. These functionalities for record I/O (except duplicate removal) are in library code shared by our applications and applicable to other file interface implementations at Google. With that, the same sequence of records, plus rare duplicates, is always delivered to the record reader.

## 3. SYSTEM INTERACTIONS

We designed the system to minimize the master's involvement in all operations. With that background, we now describe how the client, master, and chunkservers interact to implement data mutations, atomic record append, and snapshot.

### 3.1 Leases and Mutation Order

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk's replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the *primary*. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the *HeartBeat* messages regularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

In Figure 2, we illustrate this process by following the control flow of a write through these numbered steps.

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary

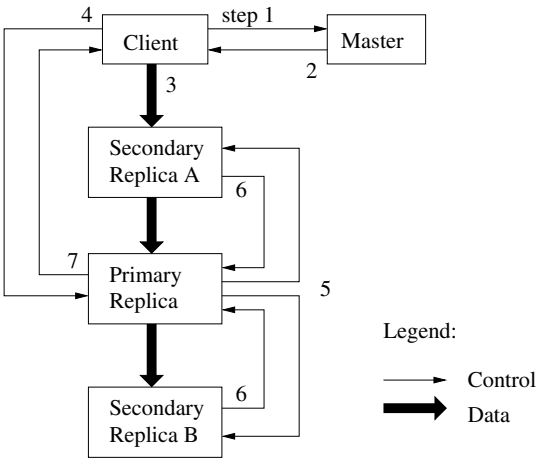



**Figure 2: Write Control and Data Flow**

becomes unreachable or replies that it no longer holds a lease.

3. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.
5. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.
6. The secondaries all reply to the primary indicating that they have completed the operation.
7. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations. They all follow the control flow described above but may be interleaved with and overwritten by concurrent operations from other clients. Therefore, the shared file region may end up containing fragments from different clients, although the replicas will be identical because the individual operations are completed successfully in the same order on all replicas. This leaves the file region in consistent but undefined state as noted in Section 2.7.

## 3.2 Data Flow

We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion. Our goals are to fully utilize each machine's network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data.

To fully utilize each machine's network bandwidth, the data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree). Thus, each machine's full outbound bandwidth is used to transfer the data as fast as possible rather than divided among multiple recipients.

To avoid network bottlenecks and high-latency links (e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the "closest" machine in the network topology that has not received it. Suppose the client is pushing data to chunkservers S1 through S4. It sends the data to the closest chunkserver, say S1. S1 forwards it to the closest chunkserver S2 through S4 closest to S1, say S2. Similarly, S2 forwards it to S3 or S4, whichever is closer to S2, and so on. Our network topology is simple enough that "distances" can be accurately estimated from IP addresses.

Finally, we minimize latency by pipelining the data transfer over TCP connections. Once a chunkserver receives some data, it starts forwarding immediately. Pipelining is especially helpful to us because we use a switched network with full-duplex links. Sending the data immediately does not reduce the receive rate. Without network congestion, the ideal elapsed time for transferring $B$ bytes to $R$ replicas is $B/T + RL$ where $T$ is the network throughput and $L$ is latency to transfer bytes between two machines. Our network links are typically 100 Mbps ($T$), and $L$ is far below 1 ms. Therefore, 1 MB can ideally be distributed in about 80 ms.

## 3.3 Atomic Record Appends

GFS provides an atomic append operation called *record append*. In a traditional write, the client specifies the offset at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. In a record append, however, the client specifies only the data. GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client. This is similar to writing to a file opened in `O_APPEND` mode in Unix without the race conditions when multiple writers do so concurrently.

Record append is heavily used by our distributed applications in which many clients on different machines append to the same file concurrently. Clients would need additional complicated and expensive synchronization, for example through a distributed lock manager, if they do so with traditional writes. In our workloads, such files often

serve as multiple-producer/single-consumer queues or contain merged results from many different clients.

Record append is a kind of mutation and follows the control flow in Section 3.1 with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the file Then, it sends its request to the primary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worst-case fragmentation at an acceptable level.) If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

If a record append fails at any replica, the client retries the operation. As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This property follows readily from the simple observation that for the operation to report success, the data must have been written at the same offset on all replicas of some chunk. Furthermore, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary. In terms of our consistency guarantees, the regions in which successful record append operations have written their data are defined (hence consistent), whereas intervening regions are inconsistent (hence undefined). Our applications can deal with inconsistent regions as we discussed in Section 2.7.2.

## 3.4 Snapshot

The snapshot operation makes a copy of a file or a directory tree (the "source") almost instantaneously, while minimizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

Like AFS [5], we use standard copy-on-write techniques to implement snapshots. When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to find the lease holder. This will give the master an opportunity to create a new copy of the chunk first.

After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

The first time a client wants to write to a chunk C after the snapshot operation, it sends a request to the master to find the current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk handle C'. It then asks each chunkserver that has a current replica of C to create a new chunk called C'. By creating the new chunk on the same chunkservers as the original, we ensure that the data can be copied locally, not over the network (our disks are about three times as fast as our 100 Mb Ethernet links). From this point, request handling is no different from that for any chunk: the master grants one of the replicas a lease on the new chunk C' and replies to the client, which can write the chunk normally, not knowing that it has just been created from an existing chunk.

# 4. MASTER OPERATION

The master executes all namespace operations. In addition, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage. We now discuss each of these topics.

## 4.1 Namespace Management and Locking

Many master operations can take a long time: for example, a snapshot operation has to revoke chunkserver leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization.

Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.

Each master operation acquires a set of locks before it runs. Typically, if it involves `/d1/d2/.../dn/leaf`, it will acquire read-locks on the directory names `/d1`, `/d1/d2`, ..., `/d1/d2/.../dn`, and either a read lock or a write lock on the full pathname `/d1/d2/.../dn/leaf`. Note that `leaf` may be a file or directory depending on the operation.

We now illustrate how this locking mechanism can prevent a file `/home/user/foo` from being created while `/home/user` is being snapshotted to `/save/user`. The snapshot operation acquires read locks on `/home` and `/save`, and write locks on `/home/user` and `/save/user`. The file creation acquires read locks on `/home` and `/home/user`, and a write lock on `/home/user/foo`. The two operations will be serialized properly because they try to obtain conflicting locks on `/home/user`. File creation does not require a write lock on the parent directory because there is no "directory", or *inode*-like, data structure to be protected from modification. The read lock on the name is sufficient to protect the parent directory from deletion.

One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple file creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the file name. The read lock on the directory name suffices to prevent the directory from being deleted, renamed, or snapshotted. The write locks on

file names serialize attempts to create a file with the same name twice.

Since the namespace can have many nodes, read-write lock objects are allocated lazily and deleted once they are not in use. Also, locks are acquired in a consistent total order to prevent deadlock: they are first ordered by level in the namespace tree and lexicographically within the same level.

## 4.2 Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to distribute data for scalability, reliability, and availability.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks, a tradeoff we make willingly.

## 4.3 Creation, Re-replication, Rebalancing

Chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing.

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several factors. (1) We want to place new replicas on chunkservers with below-average disk space utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of "recent" creations on each chunkserver. Although creation itself is cheap, it reliably predicts imminent heavy write traffic because chunks are created when demanded by writes, and in our append-once-read-many workload they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher priority to a chunk that has lost two replicas than to a chunk that has lost only one. In addition, we prefer to first re-replicate chunks for live files as opposed to chunks that belong to recently deleted files (see Section 4.4). Finally, to minimize the impact of failures on running applications, we boost the priority of any chunk that is blocking client progress.

The master picks the highest priority chunk and "clones" it by instructing some chunkserver to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks. To keep cloning traffic from overwhelming client traffic, the master limits the numbers of active clone operations both for the cluster and for each chunkserver. Additionally, each chunkserver limits the amount of bandwidth it spends on each clone operation by throttling its read requests to the source chunkserver.

Finally, the master *rebalances* replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually fills up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traffic that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which existing replica to remove. In general, it prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.

## 4.4 Garbage Collection

After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels. We find that this approach makes the system much simpler and more reliable.

### *4.4.1 Mechanism*

When a file is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the file is just renamed to a hidden name that includes the deletion timestamp. During the master's regular scan of the file system namespace, it removes any such hidden files if they have existed for more than three days (the interval is configurable). Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden file is removed from the namespace, its in-memory metadata is erased. This effectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identifies orphaned chunks (i.e., those not reachable from any file) and erases the metadata for those chunks. In a *HeartBeat* message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master's metadata. The chunkserver is free to delete its replicas of such chunks.

### *4.4.2 Discussion*

Although distributed garbage collection is a hard problem that demands complicated solutions in the context of programming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the file-to-chunk mappings maintained exclusively by the master. We can also easily identify all the chunk replicas: they are Linux files under designated directories on each chunkserver. Any such replica not known to the master is "garbage."

The garbage collection approach to storage reclamation offers several advantages over eager deletion. First, it is simple and reliable in a large-scale distributed system where component failures are common. Chunk creation may succeed on some chunkservers but not others, leaving replicas that the master does not know exist. Replica deletion messages may be lost, and the master has to remember to resend them across failures, both its own and the chunkserver's. Garbage collection provides a uniform and dependable way to clean up any replicas not known to be useful. Second, it merges storage reclamation into the regular background activities of the master, such as the regular scans of namespaces and handshakes with chunkservers. Thus, it is done in batches and the cost is amortized. Moreover, it is done only when the master is relatively free. The master can respond more promptly to client requests that demand timely attention. Third, the delay in reclaiming storage provides a safety net against accidental, irreversible deletion.

In our experience, the main disadvantage is that the delay sometimes hinders user effort to fine tune usage when storage is tight. Applications that repeatedly create and delete temporary files may not be able to reuse the storage right away. We address these issues by expediting storage reclamation if a deleted file is explicitly deleted again. We also allow users to apply different replication and reclamation policies to different parts of the namespace. For example, users can specify that all the chunks in the files within some directory tree are to be stored without replication, and any deleted files are immediately and irrevocably removed from the file system state.

## 4.5 Stale Replica Detection

Chunk replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a *chunk version number* to distinguish between up-to-date and stale replicas.

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-to-date replicas. The master and these replicas all record the new version number in their persistent state. This occurs before any client is notified and therefore before it can start writing to the chunk. If another replica is currently unavailable, its chunk version number will not be advanced. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. If the master sees a version number greater than the one in its records, the master assumes that it failed when granting the lease and so takes the higher version to be up-to-date.

The master removes stale replicas in its regular garbage collection. Before that, it effectively considers a stale replica not to exist at all when it replies to client requests for chunk information. As another safeguard, the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk or when it instructs a chunkserver to read the chunk from another chunkserver in a cloning operation. The client or the chunkserver verifies the version number when it performs the operation so that it is always accessing up-to-date data.

# 5. FAULT TOLERANCE AND DIAGNOSIS

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Component failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose problems when they inevitably occur.

## 5.1 High Availability

Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. We keep the overall system highly available with two simple yet effective strategies: fast recovery and replication.

### 5.1.1 Fast Recovery

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experience a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry. Section 6.2.2 reports observed startup times.

### 5.1.2 Chunk Replication

As discussed earlier, each chunk is replicated on multiple chunkservers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go offline or detect corrupted replicas through checksum verification (see Section 5.2). Although replication has served us well, we are exploring other forms of cross-server redundancy such as parity or erasure codes for our increasing read-only storage requirements. We expect that it is challenging but manageable to implement these more complicated redundancy schemes in our very loosely coupled system because our traffic is dominated by appends and reads rather than small random writes.

### 5.1.3 Master Replication

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally. When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. Clients use only the canonical name of the master (e.g. gfs-test), which is a DNS alias that can be changed if the master is relocated to another machine.

Moreover, "shadow" masters provide read-only access to the file system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results. In fact, since file content is read from chunkservers, applications do not observe stale file content. What could be

stale within short windows is file metadata, like directory contents or access control information.

To keep itself informed, a shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does. Like the primary, it polls chunkservers at startup (and infrequently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status. It depends on the primary master only for replica location updates resulting from the primary's decisions to create and delete replicas.

## 5.2 Data Integrity

Each chunkserver uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of GFS mutations, in particular atomic record append as discussed earlier, does not guarantee identical replicas. Therefore, each chunkserver must independently verify the integrity of its own copy by maintaining checksums.

A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

Checksumming has little effect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserver are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the checksum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the first and last blocks of the range being overwritten, then perform the write, and finally compute and record the new checksums. If we do not verify the first and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corruption in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

## 5.3 Diagnostic Tools

Extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost. Without logs, it is hard to understand transient, non-repeatable interactions between machines. GFS servers generate diagnostic logs that record many significant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without affecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records on different machines, we can reconstruct the entire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the benefits) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

# 6. MEASUREMENTS

In this section we present a few micro-benchmarks to illustrate the bottlenecks inherent in the GFS architecture and implementation, and also some numbers from real clusters in use at Google.

## 6.1 Micro-benchmarks

We measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this configuration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

### *6.1.1 Reads*

*N* clients read simultaneously from the file system. Each client reads a randomly selected 4 MB region from a 320 GB file set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunkservers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buffer cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for $N$ clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The efficiency drops from 80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

### 6.1.2 Writes

$N$ clients write simultaneously to $N$ distinct files. Each client writes 1 GB of data to a new file in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or 2.2 MB/s per client), about half the theoretical limit. As in the case of reads, it becomes more likely that multiple clients write concurrently to the same chunkserver as the number of clients increases. Moreover, collision is more likely for 16 writers than for 16 readers because each write involves three different replicas.

Writes are slower than we would like. In practice this has not been a major problem because even though it increases the latencies as seen by individual clients, it does not significantly affect the aggregate write bandwidth delivered by the system to a large number of clients.

### 6.1.3 Record Appends

Figure 3(c) shows record append performance. $N$ clients append simultaneously to a single file. Performance is limited by the network bandwidth of the chunkservers that store the last chunk of the file, independent of the number of clients. It starts at 6.0 MB/s for one client and drops to 4.8 MB/s for 16 clients, mostly due to congestion and variances in network transfer rates seen by different clients.

Our applications tend to produce multiple such files concurrently. In other words, $N$ clients append to $M$ shared files simultaneously where both $N$ and $M$ are in the dozens or hundreds. Therefore, the chunkserver network congestion in our experiment is not a significant issue in practice because a client can make progress on writing one file while the chunkservers for another file are busy.

## 6.2 Real World Clusters

We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hundred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single "task" consists of many processes on many machines reading and writing many files simultaneously.

| Cluster | A | B |
|---|---|---|
| Chunkservers | 342 | 227 |
| Available disk space | 72 TB | 180 TB |
| Used disk space | 55 TB | 155 TB |
| Number of Files | 735 k | 737 k |
| Number of Dead files | 22 k | 232 k |
| Number of Chunks | 992 k | 1550 k |
| Metadata at chunkservers | 13 GB | 21 GB |
| Metadata at master | 48 MB | 60 MB |

**Table 2: Characteristics of two GFS clusters**

### 6.2.1 Storage

As shown by the first five entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. "Used space" includes all chunk replicas. Virtually all files are replicated three times. Therefore, the clusters store 18 TB and 52 TB of file data respectively.

The two clusters have similar numbers of files, though B has a larger proportion of dead files, namely files which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its files tend to be larger.

### 6.2.2 Metadata

The chunkservers in aggregate store tens of GBs of metadata, mostly the checksums for 64 KB blocks of user data. The only other metadata kept at the chunkservers is the chunk version number discussed in Section 4.5.

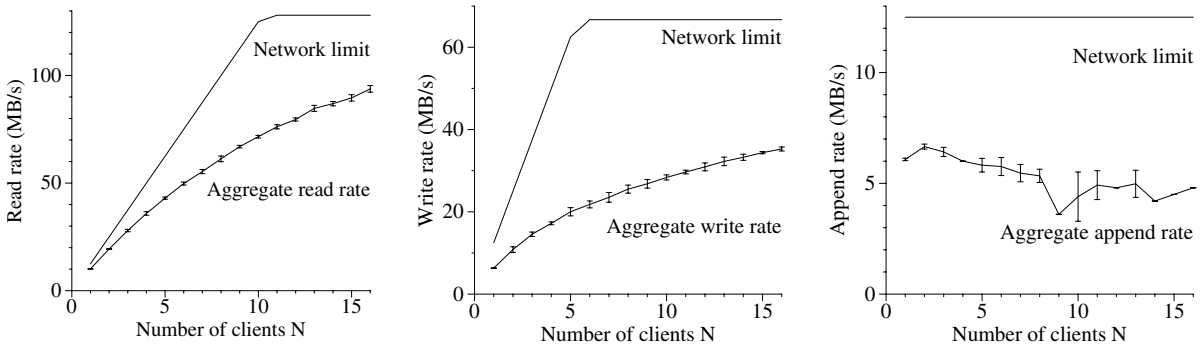
The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per file on average. This agrees with our assumption that the size of the master's memory does not limit the system's capacity in practice. Most of the per-file metadata is the file names stored in a prefix-compressed form. Other metadata includes file ownership and permissions, mapping from files to chunks, and each chunk's current version. In addition, for each chunk we store the current replica locations and a reference count for implementing copy-on-write.

Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from disk before the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to 60 seconds – until it has fetched chunk location information from all chunkservers.

### 6.2.3 Read and Write Rates

Table 3 shows read and write rates for various time periods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of GFS.)

The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load because writes are propagated to three replicas.




(a) Reads (b) Writes (c) Record appends

**Figure 3: Aggregate Throughputs.** Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

| Cluster | A | B |
|---|---|---|
| Read rate (last minute) | 583 MB/s | 380 MB/s |
| Read rate (last hour) | 562 MB/s | 384 MB/s |
| Read rate (since restart) | 589 MB/s | 49 MB/s |
| Write rate (last minute) | 1 MB/s | 101 MB/s |
| Write rate (last hour) | 2 MB/s | 117 MB/s |
| Write rate (since restart) | 25 MB/s | 13 MB/s |
| Master ops (last minute) | 325 Ops/s | 533 Ops/s |
| Master ops (last hour) | 381 Ops/s | 518 Ops/s |
| Master ops (since restart) | 202 Ops/s | 347 Ops/s |

**Table 3: Performance Metrics for Two GFS Clusters**

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network configuration can support 750 MB/s, so it was using its resources efficiently. Cluster B can support peak read rates of 1300 MB/s, but its applications were using just 380 MB/s.

### 6.2.4 Master Load

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of GFS, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which contained hundreds of thousands of files) looking for particular files. We have since changed the master data structures to allow efficient binary searches through the namespace. It can now easily support many thousands of file accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data structures.

### 6.2.5 Recovery Time

After a chunkserver fails, some chunks will become underreplicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about 15,000 chunks containing 600 GB of data. To limit the impact on running applications and provide leeway for scheduling decisions, our default parameters limit this cluster to 91 concurrent clonings (40% of the number of chunkservers) where each clone operation is allowed to consume at most 6.25 MB/s (50 Mbps). All chunks were restored in 23.2 minutes, at an effective replication rate of 440 MB/s.

In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660 GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

## 6.3 Workload Breakdown

In this section, we present a detailed breakdown of the workloads on two GFS clusters comparable but not identical to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data processing.

### 6.3.1 Methodology and Caveats

These results include only client originated requests so that they reflect the workload generated by our applications for the file system as a whole. They do not include interserver requests to carry out client requests or internal background activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by GFS servers. For example, GFS client code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise. Explicit logging by applications might have provided slightly more accurate data, but it is logistically impossible to recompile and restart thousands of running clients to do so and cumbersome to collect the results from as many machines.

One should be careful not to overly generalize from our workload. Since Google completely controls both GFS and its applications, the applications tend to be tuned for GFS, and conversely GFS is designed for these applications. Such mutual influence may also exist between general applications

| Operation | Read | | Write | | Record Append | |
|---|---|---|---|---|---|---|
| Cluster | X | Y | X | Y | X | Y |
| 0K | 0.4 | 2.6 | 0 | 0 | 0 | 0 |
| 1B..1K | 0.1 | 4.1 | 6.6 | 4.9 | 0.2 | 9.2 |
| 1K..8K | 65.2 | 38.5 | 0.4 | 1.0 | 18.9 | 15.2 |
| 8K..64K | 29.9 | 45.1 | 17.8 | 43.0 | 78.0 | 2.8 |
| 64K..128K | 0.1 | 0.7 | 2.3 | 1.9 | < .1 | 4.3 |
| 128K..256K | 0.2 | 0.3 | 31.6 | 0.4 | < .1 | 10.6 |
| 256K..512K | 0.1 | 0.1 | 4.2 | 7.7 | < .1 | 31.2 |
| 512K..1M | 3.9 | 6.9 | 35.5 | 28.7 | 2.2 | 25.5 |
| 1M..inf | 0.1 | 1.8 | 1.5 | 12.3 | 0.7 | 2.2 |

**Table 4: Operations Breakdown by Size (%).** For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

and file systems, but the effect is likely more pronounced in our case.

### 6.3.2 Chunkserver Workload

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (under 64 KB) come from seek-intensive clients that look up small pieces of data within huge files. The large reads (over 512 KB) come from long sequential reads through entire files.

A significant number of reads return no data at all in cluster Y. Our applications, especially those in the production systems, often use files as producer-consumer queues. Producers append concurrently to a file while a consumer reads the end of file. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.

Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from significant buffering within the writers. Writers that buffer less data, checkpoint or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).

As for record appends, cluster Y sees a much higher percentage of large record appends than cluster X does because our production systems, which use cluster Y, are more aggressively tuned for GFS.

Table 5 shows the total amount of data transferred in operations of various sizes. For all kinds of operations, the larger operations (over 256 KB) generally account for most of the bytes transferred. Small reads (under 64 KB) do transfer a small but significant portion of the read data because of the random seek workload.

### 6.3.3 Appends versus Writes

Record appends are heavily used especially in our production systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts. For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ratios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two applications with particular buffer size choices.

As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This approximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload *per se* but a consequence of the retry mechanism.

| Operation | Read | | Write | | Record Append | |
|---|---|---|---|---|---|---|
| Cluster | X | Y | X | Y | X | Y |
| 1B..1K | < .1 | < .1 | < .1 | < .1 | < .1 | < .1 |
| 1K..8K | 13.8 | 3.9 | < .1 | < .1 | < .1 | 0.1 |
| 8K..64K | 11.4 | 9.3 | 2.4 | 5.9 | 2.3 | 0.3 |
| 64K..128K | 0.3 | 0.7 | 0.3 | 0.3 | 22.7 | 1.2 |
| 128K..256K | 0.8 | 0.6 | 16.5 | 0.2 | < .1 | 5.8 |
| 256K..512K | 1.4 | 0.3 | 3.4 | 7.7 | < .1 | 38.4 |
| 512K..1M | 65.9 | 55.1 | 74.1 | 58.0 | .1 | 46.8 |
| 1M..inf | 6.4 | 30.1 | 3.3 | 28.0 | 53.9 | 7.4 |

**Table 5: Bytes Transferred Breakdown by Operation Size (%).** For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

| Cluster | X | Y |
|---|---|---|
| Open | 26.1 | 16.3 |
| Delete | 0.7 | 1.5 |
| FindLocation | 64.3 | 65.8 |
| FindLeaseHolder | 7.8 | 13.4 |
| FindMatchingFiles | 0.6 | 2.2 |
| All other combined | 0.5 | 0.8 |

**Table 6: Master Requests Breakdown by Type (%)**

### 6.3.4 Master Workload

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (*FindLocation*) for reads and lease holder information (*FindLeaseLocker*) for data mutations.

Clusters X and Y see significantly different numbers of *Delete* requests because cluster Y stores production data sets that are regularly regenerated and replaced with newer versions. Some of this difference is further hidden in the difference in *Open* requests because an old version of a file may be implicitly deleted by being opened for write from scratch (mode "w" in Unix open terminology).

*FindMatchingFiles* is a pattern matching request that supports "ls" and similar file system operations. Unlike other requests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the file system to understand global application state. In contrast, cluster X's applications are under more explicit user control and usually know the names of all needed files in advance.

## 7. EXPERIENCES

In the process of building and deploying GFS, we have experienced a variety of issues, some operational and some technical.

Initially, GFS was conceived as the backend file system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production systems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked, but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption, while concurrently we modified the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of `fsync()`. Its cost is proportional to the size of the file rather than the size of the modified portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modifies the address space in an `mmap()` call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Eventually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by replacing `mmap()` with `pread()` at the cost of an extra copy.

Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel and share the changes with the open source community.

## 8. RELATED WORK

Like other large distributed file systems such as AFS [5], GFS provides a location independent namespace which enables data to be moved transparently for load balance or fault tolerance. Unlike AFS, GFS spreads a file's data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, GFS currently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], GFS does not provide any caching below the file system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed file systems like Frangipani, xFS, Minnesota's GFS[11] and GPFS [10] remove the centralized server and rely on distributed algorithms for consistency and management. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain flexibility. In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully replicated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persistent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp [7] to provide high availability with stronger consistency guarantees than our current scheme.

We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simplified the problem significantly by focusing on the needs of our applications rather than building a POSIX-compliant file system. Additionally, GFS assumes large number of unreliable components and so fault tolerance is central to our design.

GFS most closely resembles the NASD architecture [4]. While the NASD architecture is based on network-attached disk drives, GFS uses commodity machines as chunkservers, as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated fixed-size chunks rather than variable-length objects. Additionally, GFS implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota's GFS and NASD, we do not seek to alter the model of the storage device. We focus on addressing day-to-day data processing needs for complicated distributed systems with existing commodity components.

The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queues in River [2]. While River uses memory-based queues distributed across machines and careful data flow control, GFS uses a persistent file that can be appended to concurrently by many producers. The River model supports m-to-n distributed queues but lacks the fault tolerance that comes with persistent storage, while GFS only supports m-to-1 queues efficiently. Multiple consumers can read the same file, but they must coordinate to partition the incoming load.

## 9. CONCLUSIONS

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness.

We started by reexamining traditional file system assumptions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically different points in the design space. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequentially), and both extend and relax the standard file system interface to improve the overall system.

Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver

failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, centralized master that does not become a bottleneck. We believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.

GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

## ACKNOWLEDGMENTS

We wish to thank the following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David desJardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing framework and security enhancements. David Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

## REFERENCES

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.

[4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems*, pages 92–103, San Jose, California, October 1998.

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[6] InterMezzo. http://www.inter-mezzo.org, 2003.

[7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991.

[8] Lustre. http://www.lustreorg, 2003.

[9] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, September 1988.

[10] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, California, January 2002.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Gobal File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, Maryland, September 1996.

[12] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224–237, Saint-Malo, France, October 1997.

# Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

*Google, Inc.*

## Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

## 1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

## 2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

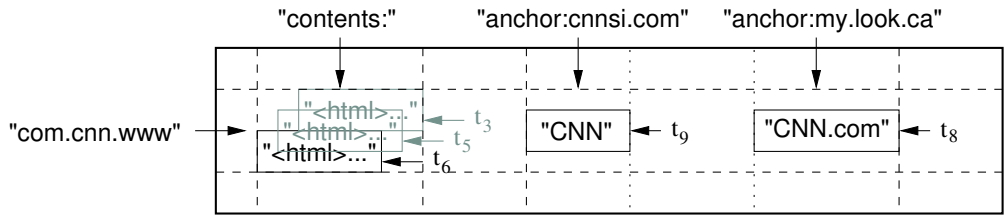



Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps $t_3$, $t_5$, and $t_6$.

We settled on this data model after examining a variety of potential uses of a Bigtable-like system. As one concrete example that drove some of our design decisions, suppose we want to keep a copy of a large collection of web pages and related information that could be used by many different projects; let us call this particular table the *Webtable*. In Webtable, we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the contents: column under the timestamps when they were fetched, as illustrated in Figure 1.

### Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row.

Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned. Each row range is called a *tablet*, which is the unit of distribution and load balancing. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines. Clients can exploit this property by selecting their row keys so that they get good locality for their data accesses. For example, in Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname components of the URLs. For example, we store data for maps.google.com/index.html under the key com.google.maps/index.html. Storing pages from the same domain near each other makes some host and domain analyses more efficient.

### Column Families

Column keys are grouped into sets called *column families*, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used. It is our intent that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation. In contrast, a table may have an unbounded number of columns.

A column key is named using the following syntax: *family*:*qualifier*. Column family names must be printable, but qualifiers may be arbitrary strings. An example column family for the Webtable is language, which stores the language in which a web page was written. We use only one column key in the language family, and it stores each web page's language ID. Another useful column family for this table is anchor; each column key in this family represents a single anchor, as shown in Figure 1. The qualifier is the name of the referring site; the cell contents is the link text.

Access control and both disk and memory accounting are performed at the column-family level. In our Webtable example, these controls allow us to manage several different types of applications: some that add new base data, some that read the base data and create derived column families, and some that are only allowed to view existing data (and possibly not even to view all of the existing families for privacy reasons).

### Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent "real time" in microseconds, or be explicitly assigned by client

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

Figure 3: Reading from Bigtable.

applications. Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first.

To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last $n$ versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were written in the last seven days).

In our Webtable example, we set the timestamps of the crawled pages stored in the contents: column to the times at which these page versions were actually crawled. The garbage-collection mechanism described above lets us keep only the most recent three versions of every page.

## 3 API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.

Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. Figure 2 shows C++ code that uses a `RowMutation` abstraction to perform a series of updates. (Irrelevant details were elided to keep the example short.) The call to `Apply` performs an atomic mutation to the Webtable: it adds one anchor to www.cnn.com and deletes a different anchor.

Figure 3 shows C++ code that uses a `Scanner` abstraction to iterate over all anchors in a particular row. Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows, columns, and timestamps produced by a scan. For example, we could restrict the scan above to only produce anchors whose columns match the regular expression anchor:*.cnn.com, or to only produce anchors whose timestamps fall within ten days of the current time.

Bigtable supports several other features that allow the user to manipulate data in more complex ways. First, Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients. Second, Bigtable allows cells to be used as integer counters. Finally, Bigtable supports the execution of client-supplied scripts in the address spaces of the servers. The scripts are written in a language developed at Google for processing data called Sawzall [28]. At the moment, our Sawzall-based API does not allow client scripts to write back into Bigtable, but it does allow various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators.

Bigtable can be used with MapReduce [12], a framework for running large-scale parallel computations developed at Google. We have written a set of wrappers that allow a Bigtable to be used both as an input source and as an output target for MapReduce jobs.

## 4 Building Blocks

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data files. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status.

The Google *SSTable* file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified

key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [8]. A Chubby service consists of five active replicas, one of which is elected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm [9, 23] to keep its replicas consistent in the face of failure. Chubby provides a namespace that consists of directories and small files. Each directory or file can be used as a lock, and reads and writes to a file are atomic. The Chubby client library provides consistent caching of Chubby files. Each Chubby client maintains a *session* with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby files and directories for notification of changes or session expiration.

Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data (see Section 5.1); to discover tablet servers and finalize tablet server deaths (see Section 5.2); to store Bigtable schema information (the column family information for each table); and to store access control lists. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances. The average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%. The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

# 5 Implementation

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. Tablet servers can be dynamically added (or removed) from a cluster to accomodate changes in workloads.

The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of files in GFS. In addition, it handles schema changes such as table and column family creations.

Each tablet server manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large.

As with many single-master distributed storage systems [17, 21], client data does not move through the master: clients communicate directly with tablet servers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice.

A Bigtable cluster stores a number of tables. Each table consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

## 5.1 Tablet Location

We use a three-level hierarchy analogous to that of a $B^+$-tree [10] to store tablet location information (Figure 4).

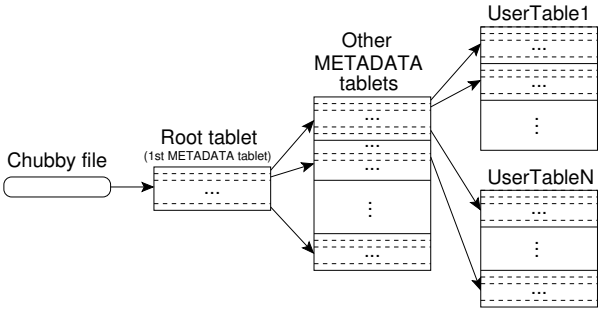



Figure 4: Tablet location hierarchy.

The first level is a file stored in Chubby that contains the location of the *root tablet*. The *root tablet* contains the location of all tablets in a special `METADATA` table. Each `METADATA` tablet contains the location of a set of user tablets. The *root tablet* is just the first tablet in the `METADATA` table, but is treated specially—it is never split—to ensure that the tablet location hierarchy has no more than three levels.

The `METADATA` table stores the location of a tablet under a row key that is an encoding of the tablet's table

identifier and its end row. Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address $2^{34}$ tablets (or $2^{61}$ bytes in 128 MB tablets).

The client library caches tablet locations. If the client does not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy. If the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby. If the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses (assuming that METADATA tablets do not move very frequently). Although tablet locations are stored in memory, so no GFS accesses are required, we further reduce this cost in the common case by having the client library prefetch tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table.

We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

## 5.2 Tablet Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory. The master monitors this directory (the *servers directory*) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock: e.g., due to a network partition that caused the server to lose its Chubby session. (Chubby provides an efficient mechanism that allows a tablet server to check whether it still holds its lock without incurring network traffic.) A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists. If the file no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates (e.g., because the cluster management system is removing the tablet server's machine from the cluster), it attempts to release its lock so that the master will reassign its tablets more quickly.

The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last several attempts, the master attempts to acquire an exclusive lock on the server's file. If the master is able to acquire the lock, then Chubby is live and the tablet server is either dead or having trouble reaching Chubby, so the master ensures that the tablet server can never serve again by deleting its server file. Once a server's file has been deleted, the master can move all the tablets that were previously assigned to that server into the set of unassigned tablets. To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. However, as described above, master failures do not change the assignment of tablets to tablet servers.

When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique *master* lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to find the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned. Therefore, before starting this scan (step 4), the master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet was not discovered during step 3. This addition ensures that the root tablet will be assigned. Because the root tablet contains the names of all METADATA tablets, the master knows about all of them after it has scanned the root tablet.

The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets. The master is able to keep track of these changes because it initiates all but the last. Tablet splits are treated specially since they are initiated by a tablet server. The tablet server commits the split by recording information for the new tablet in the METADATA table. When the split has committed, it notifies the master. In case the split notification is lost (either

because the tablet server or the master died), the master detects the new tablet when it asks a tablet server to load the tablet that has now split. The tablet server will notify the master of the split, because the tablet entry it finds in the METADATA table will specify only a portion of the tablet that the master asked it to load.

## 5.3 Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a *memtable*; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server

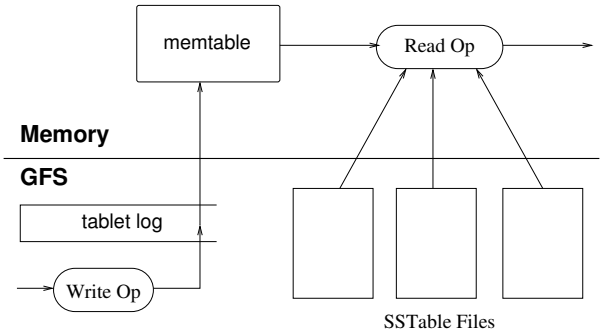



Figure 5: Tablet Representation

reads its metadata from the METADATA table. This metadata contains the list of SSTables that comprise a tablet and a set of a redo points, which are pointers into any commit logs that may contain data for the tablet. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all of the updates that have committed since the redo points.

When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby file (which is almost always a hit in the Chubby client cache). A valid mutation is written to the commit log. Group commit is used to improve the throughput of lots of small mutations [13, 16]. After the write has been committed, its contents are inserted into the memtable.

When a read operation arrives at a tablet server, it is similarly checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and the memtable. Since the SSTables and the memtable are lexicographically sorted data structures, the merged view can be formed efficiently.

Incoming read and write operations can continue while tablets are split and merged.

## 5.4 Compactions

As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Every minor compaction creates a new SSTable. If this behavior continued unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we bound the number of such files by periodically executing a *merging compaction* in the background. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has finished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactions to them. These major compactions allow Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

# 6 Refinements

The implementation described in the previous section required a number of refinements to achieve the high performance, availability, and reliability required by our users. This section describes portions of the implementation in more detail in order to highlight these refinements.

### Locality groups

Clients can group multiple column families together into a *locality group*. A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more efficient reads. For example, page metadata in Webtable (such as language and checksums) can be in one locality group, and the contents of the page can be in a different group: an ap-

plication that wants to read the metadata does not need to read through all of the page contents.

In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory. SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This feature is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the `METADATA` table.

### Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which compression format is used. The user-specified compression format is applied to each SSTable block (whose size is controllable via a locality group specific tuning parameter). Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire file. Many clients use a two-pass custom compression scheme. The first pass uses Bentley and McIlroy's scheme [6], which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast—they encode at 100–200 MB/s, and decode at 400–1000 MB/s on modern machines.

Even though we emphasized speed instead of space reduction when choosing our compression algorithms, this two-pass compression scheme does surprisingly well. For example, in Webtable, we use this compression scheme to store Web page contents. In one experiment, we stored a large number of documents in a compressed locality group. For the purposes of the experiment, we limited ourselves to one version of each document instead of storing all versions available to us. The scheme achieved a 10-to-1 reduction in space. This is much better than typical Gzip reductions of 3-to-1 or 4-to-1 on HTML pages because of the way Webtable rows are laid out: all pages from a single host are stored close to each other. This allows the Bentley-McIlroy algorithm to identify large amounts of shared boilerplate in pages from the same host. Many applications, not just Webtable, choose their row names so that similar data ends up clustered, and therefore achieve very good compression ratios. Compression ratios get even better when we store multiple versions of the same value in Bigtable.

### Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTables blocks that were read from GFS. The Scan Cache is most useful for applications that tend to read the same data repeatedly. The Block Cache is useful for applications that tend to read data that is close to the data they recently read (e.g., sequential reads, or random reads of different columns in the same locality group within a hot row).

### Bloom filters

As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end up doing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters [7] should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

### Commit-log implementation

If we kept the commit log for each tablet in a separate log file, a very large number of files would be written concurrently in GFS. Depending on the underlying file system implementation on each GFS server, these writes could cause a large number of disk seeks to write to the different physical log files. In addition, having separate log files per tablet also reduces the effectiveness of the group commit optimization, since groups would tend to be smaller. To fix these issues, we append mutations to a single commit log per tablet server, co-mingling mutations for different tablets in the same physical log file [18, 20].

Using one log provides significant performance benefits during normal operation, but it complicates recovery. When a tablet server dies, the tablets that it served will be moved to a large number of other tablet servers: each server typically loads a small number of the original server's tablets. To recover the state for a tablet, the new tablet server needs to reapply the mutations for that tablet from the commit log written by the original tablet server. However, the mutations for these tablets

were co-mingled in the same physical log file. One approach would be for each new tablet server to read this full commit log file and apply just the entries needed for the tablets it needs to recover. However, under such a scheme, if 100 machines were each assigned a single tablet from a failed tablet server, then the log file would be read 100 times (once by each server).

We avoid duplicating log reads by first sorting the commit log entries in order of the keys ⟨table, row name, log sequence number⟩. In the sorted output, all mutations for a particular tablet are contiguous and can therefore be read efficiently with one disk seek followed by a sequential read. To parallelize the sorting, we partition the log file into 64 MB segments, and sort each segment in parallel on different tablet servers. This sorting process is coordinated by the master and is initiated when a tablet server indicates that it needs to recover mutations from some commit log file.

Writing commit logs to GFS sometimes causes performance hiccups for a variety of reasons (e.g., a GFS server machine involved in the write crashes, or the network paths traversed to reach the particular set of three GFS servers is suffering network congestion, or is heavily loaded). To protect mutations from GFS latency spikes, each tablet server actually has two log writing threads, each writing to its own log file; only one of these two threads is actively in use at a time. If writes to the active log file are performing poorly, the log file writing is switched to the other thread, and mutations that are in the commit log queue are written by the newly active log writing thread. Log entries contain sequence numbers to allow the recovery process to elide duplicated entries resulting from this log switching process.

**Speeding up tablet recovery**

If the master moves a tablet from one tablet server to another, the source tablet server first does a minor compaction on that tablet. This compaction reduces recovery time by reducing the amount of uncompacted state in the tablet server's commit log. After finishing this compaction, the tablet server stops serving the tablet. Before it actually unloads the tablet, the tablet server does another (usually very fast) minor compaction to eliminate any remaining uncompacted state in the tablet server's log that arrived while the first minor compaction was being performed. After this second minor compaction is complete, the tablet can be loaded on another tablet server without requiring any recovery of log entries.

**Exploiting immutability**

Besides the SSTable caches, various other parts of the Bigtable system have been simplified by the fact that all of the SSTables that we generate are immutable. For example, we do not need any synchronization of accesses to the file system when reading from SSTables. As a result, concurrency control over rows can be implemented very efficiently. The only mutable data structure that is accessed by both reads and writes is the memtable. To reduce contention during reads of the memtable, we make each memtable row copy-on-write and allow reads and writes to proceed in parallel.

Since SSTables are immutable, the problem of permanently removing deleted data is transformed to garbage collecting obsolete SSTables. Each tablet's SSTables are registered in the `METADATA` table. The master removes obsolete SSTables as a mark-and-sweep garbage collection [25] over the set of SSTables, where the `METADATA` table contains the set of roots.

Finally, the immutability of SSTables enables us to split tablets quickly. Instead of generating a new set of SSTables for each child tablet, we let the child tablets share the SSTables of the parent tablet.

## 7 Performance Evaluation

We set up a Bigtable cluster with $N$ tablet servers to measure the performance and scalability of Bigtable as $N$ is varied. The tablet servers were configured to use 1 GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. $N$ client machines generated the Bigtable load used for these tests. (We used the same number of clients as tablet servers to ensure that clients were never a bottleneck.) Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments.

$R$ is the distinct number of Bigtable row keys involved in the test. $R$ was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server.

The *sequential write* benchmark used row keys with names 0 to $R - 1$. This space of row keys was partitioned into $10N$ equal-sized ranges. These ranges were assigned to the $N$ clients by a central scheduler that as-

| Experiment | # of Tablet Servers | | | |
|---|---|---|---|---|
| | 1 | 50 | 250 | 500 |
| random reads | 1212 | 593 | 479 | 241 |
| random reads (mem) | 10811 | 8511 | 8000 | 6250 |
| random writes | 8850 | 3745 | 3425 | 2000 |
| sequential reads | 4425 | 2463 | 2625 | 2469 |
| sequential writes | 8547 | 3623 | 2451 | 1905 |
| scans | 15385 | 10526 | 9524 | 7843 |

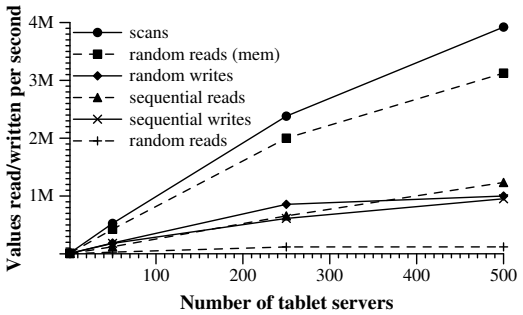



Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

signed the next available range to a client as soon as the client finished processing the previous range assigned to it. This dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines. We wrote a single string under each row key. Each string was generated randomly and was therefore uncompressible. In addition, strings under different row key were distinct, so no cross-row compression was possible. The *random write* benchmark was similar except that the row key was hashed modulo $R$ immediately before writing so that the write load was spread roughly uniformly across the entire row space for the entire duration of the benchmark.

The *sequential read* benchmark generated row keys in exactly the same way as the sequential write benchmark, but instead of writing under the row key, it read the string stored under the row key (which was written by an earlier invocation of the sequential write benchmark). Similarly, the *random read* benchmark shadowed the operation of the random write benchmark.

The *scan* benchmark is similar to the sequential read benchmark, but uses support provided by the Bigtable API for scanning over all values in a row range. Using a scan reduces the number of RPCs executed by the benchmark since a single RPC fetches a large sequence of values from a tablet server.

The *random reads (mem)* benchmark is similar to the random read benchmark, but the locality group that contains the benchmark data is marked as *in-memory*, and therefore the reads are satisfied from the tablet server's memory instead of requiring a GFS read. For just this benchmark, we reduced the amount of data per tablet server from 1 GB to 100 MB so that it would fit comfortably in the memory available to the tablet server.

Figure 6 shows two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.

**Single tablet-server performance**

Let us first consider performance with just one tablet server. Random reads are slower than all other operations by an order of magnitude or more. Each random read involves the transfer of a 64 KB SSTable block over the network from GFS to a tablet server, out of which only a single 1000-byte value is used. The tablet server executes approximately 1200 reads per second, which translates into approximately 75 MB/s of data read from GFS. This bandwidth is enough to saturate the tablet server CPUs because of overheads in our networking stack, SSTable parsing, and Bigtable code, and is also almost enough to saturate the network links used in our system. Most Bigtable applications with this type of an access pattern reduce the block size to a smaller value, typically 8KB.

Random reads from memory are much faster since each 1000-byte read is satisfied from the tablet server's local memory without fetching a large 64 KB block from GFS.

Random and sequential writes perform better than random reads since each tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes efficiently to GFS. There is no significant difference between the performance of random writes and sequential writes; in both cases, all writes to the tablet server are recorded in the same commit log.

Sequential reads perform better than random reads since every 64 KB SSTable block that is fetched from GFS is stored into our block cache, where it is used to serve the next 64 read requests.

Scans are even faster since the tablet server can return a large number of values in response to a single client RPC, and therefore RPC overhead is amortized over a large number of values.

**Scaling**

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500. For example, the

| # of tablet servers | # of clusters |
|---|---|
| 0 .. 19 | 259 |
| 20 .. 49 | 47 |
| 50 .. 99 | 20 |
| 100 .. 499 | 50 |
| $> 500$ | 12 |

Table 1: Distribution of number of tablet servers in Bigtable clusters.

performance of random reads from memory increases by almost a factor of 300 as the number of tablet server increases by a factor of 500. This behavior occurs because the bottleneck on performance for this benchmark is the individual tablet server CPU.

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

The random read benchmark shows the worst scaling (an increase in aggregate throughput by only a factor of 100 for a 500-fold increase in number of servers). This behavior occurs because (as explained above) we transfer one large 64KB block over the network for every 1000-byte read. This transfer saturates various shared 1 Gigabit links in our network and as a result, the per-server throughput drops significantly as we increase the number of machines.

# 8 Real Applications

As of August 2006, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. Many of these clusters are used for development purposes and therefore are idle for significant periods. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traffic of about 16 GB/s.

Table 2 provides some data about a few of the tables currently in use. Some tables store data that is served to users, whereas others store data for batch processing; the tables range widely in total size, average cell size, percentage of data served from memory, and complexity of the table schema. In the rest of this section, we briefly describe how three product teams use Bigtable.

## 8.1 Google Analytics

Google Analytics (analytics.google.com) is a service that helps webmasters analyze traffic patterns at their web sites. It provides aggregate statistics, such as the number of unique visitors per day and the page views per URL per day, as well as site-tracking reports, such as the percentage of users that made a purchase, given that they earlier viewed a specific page.

To enable the service, webmasters embed a small JavaScript program in their web pages. This program is invoked whenever a page is visited. It records various information about the request in Google Analytics, such as a user identifier and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters.

We briefly describe two of the tables used by Google Analytics. The raw click table (˜200 TB) maintains a row for each end-user session. The row name is a tuple containing the website's name and the time at which the session was created. This schema ensures that sessions that visit the same web site are contiguous, and that they are sorted chronologically. This table compresses to 14% of its original size.

The summary table (˜20 TB) contains various predefined summaries for each website. This table is generated from the raw click table by periodically scheduled MapReduce jobs. Each MapReduce job extracts recent session data from the raw click table. The overall system's throughput is limited by the throughput of GFS. This table compresses to 29% of its original size.

## 8.2 Google Earth

Google operates a collection of services that provide users with access to high-resolution satellite imagery of the world's surface, both through the web-based Google Maps interface (maps.google.com) and through the Google Earth (earth.google.com) custom client software. These products allow users to navigate across the world's surface: they can pan, view, and annotate satellite imagery at many different levels of resolution. This system uses one table to preprocess data, and a different set of tables for serving client data.

The preprocessing pipeline uses one table to store raw imagery. During preprocessing, the imagery is cleaned and consolidated into final serving data. This table contains approximately 70 terabytes of data and therefore is served from disk. The images are efficiently compressed already, so Bigtable compression is disabled.

| Project name | Table size (TB) | Compression ratio | # Cells (billions) | # Column Families | # Locality Groups | % in memory | Latency-sensitive? |
|---|---|---|---|---|---|---|---|
| *Crawl* | 800 | 11% | 1000 | 16 | 8 | 0% | No |
| *Crawl* | 50 | 33% | 200 | 2 | 2 | 0% | No |
| *Google Analytics* | 20 | 29% | 10 | 1 | 1 | 0% | Yes |
| *Google Analytics* | 200 | 14% | 80 | 1 | 1 | 0% | Yes |
| *Google Base* | 2 | 31% | 10 | 29 | 3 | 15% | Yes |
| *Google Earth* | 0.5 | 64% | 8 | 7 | 2 | 33% | Yes |
| *Google Earth* | 70 | – | 9 | 8 | 3 | 0% | No |
| *Orkut* | 9 | – | 0.9 | 8 | 5 | 1% | Yes |
| *Personalized Search* | 4 | 47% | 6 | 93 | 11 | 5% | Yes |

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

Each row in the imagery table corresponds to a single geographic segment. Rows are named to ensure that adjacent geographic segments are stored near each other. The table contains a column family to keep track of the sources of data for each segment. This column family has a large number of columns: essentially one for each raw data image. Since each segment is only built from a few images, this column family is very sparse.

The preprocessing pipeline relies heavily on MapReduce over Bigtable to transform data. The overall system processes over 1 MB/sec of data per tablet server during some of these MapReduce jobs.

The serving system uses one table to index data stored in GFS. This table is relatively small (˜500 GB), but it must serve tens of thousands of queries per second per datacenter with low latency. As a result, this table is hosted across hundreds of tablet servers and contains in-memory column families.

### 8.3 Personalized Search

Personalized Search (www.google.com/psearch) is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news. Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.

Personalized Search stores each user's data in Bigtable. Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table. A separate column family is reserved for each type of action (for example, there is a column family that stores all web queries). Each data element uses as its Bigtable timestamp the time at which the corresponding user action occurred. Personalized Search generates user profiles using a MapReduce over Bigtable. These user profiles are used to personalize live search results.

The Personalized Search data is replicated across several Bigtable clusters to increase availability and to reduce latency due to distance from clients. The Personalized Search team originally built a client-side replication mechanism on top of Bigtable that ensured eventual consistency of all replicas. The current system now uses a replication subsystem that is built into the servers.

The design of the Personalized Search storage system allows other groups to add new per-user information in their own columns, and the system is now used by many other Google properties that need to store per-user configuration options and settings. Sharing a table amongst many groups resulted in an unusually large number of column families. To help support sharing, we added a simple quota mechanism to Bigtable to limit the storage consumption by any particular client in shared tables; this mechanism provides some isolation between the various product groups using this system for per-user information storage.

## 9 Lessons

In the process of designing, implementing, maintaining, and supporting Bigtable, we gained useful experience and learned several interesting lessons.

One lesson we learned is that large distributed systems are vulnerable to many types of failures, not just the standard network partitions and fail-stop failures assumed in many distributed protocols. For example, we have seen problems due to all of the following causes: memory and network corruption, large clock skew, hung machines, extended and asymmetric network partitions, bugs in other systems that we are using (Chubby for example), overflow of GFS quotas, and planned and unplanned hardware maintenance. As we have gained more experience with these problems, we have addressed them by changing various protocols. For example, we added checksumming to our RPC mechanism. We also handled

some problems by removing assumptions made by one part of the system about another part. For example, we stopped assuming a given Chubby operation could return only one of a fixed set of errors.

Another lesson we learned is that it is important to delay adding new features until it is clear how the new features will be used. For example, we initially planned to support general-purpose transactions in our API. Because we did not have an immediate use for them, however, we did not implement them. Now that we have many real applications running on Bigtable, we have been able to examine their actual needs, and have discovered that most applications require only single-row transactions. Where people have requested distributed transactions, the most important use is for maintaining secondary indices, and we plan to add a specialized mechanism to satisfy this need. The new mechanism will be less general than distributed transactions, but will be more efficient (especially for updates that span hundreds of rows or more) and will also interact better with our scheme for optimistic cross-data-center replication.

A practical lesson that we learned from supporting Bigtable is the importance of proper system-level monitoring (i.e., monitoring both Bigtable itself, as well as the client processes using Bigtable). For example, we extended our RPC system so that for a sample of the RPCs, it keeps a detailed trace of the important actions done on behalf of that RPC. This feature has allowed us to detect and fix many problems such as lock contention on tablet data structures, slow writes to GFS while committing Bigtable mutations, and stuck accesses to the `METADATA` table when `METADATA` tablets are unavailable. Another example of useful monitoring is that every Bigtable cluster is registered in Chubby. This allows us to track down all clusters, discover how big they are, see which versions of our software they are running, how much traffic they are receiving, and whether or not there are any problems such as unexpectedly large latencies.

The most important lesson we learned is the value of simple designs. Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging. One example of this is our tablet-server membership protocol. Our first protocol was simple: the master periodically issued leases to tablet servers, and tablet servers killed themselves if their lease expired. Unfortunately, this protocol reduced availability significantly in the presence of network problems, and was also sensitive to master recovery time. We redesigned the protocol several times until we had a protocol that performed well. However, the resulting protocol was too complex and depended on the behavior of Chubby features that were seldom exercised by other applications. We discovered that we were spending an inordinate amount of time debugging obscure corner cases, not only in Bigtable code, but also in Chubby code. Eventually, we scrapped this protocol and moved to a newer simpler protocol that depends solely on widely-used Chubby features.

## 10 Related Work

The Boxwood project [24] has components that overlap in some ways with Chubby, GFS, and Bigtable, since it provides for distributed agreement, locking, distributed chunk storage, and distributed B-tree storage. In each case where there is overlap, it appears that the Boxwood's component is targeted at a somewhat lower level than the corresponding Google service. The Boxwood project's goal is to provide infrastructure for building higher-level services such as file systems or databases, while the goal of Bigtable is to directly support client applications that wish to store data.

Many recent projects have tackled the problem of providing distributed storage or higher-level services over wide area networks, often at "Internet scale." This includes work on distributed hash tables that began with projects such as CAN [29], Chord [32], Tapestry [37], and Pastry [30]. These systems address concerns that do not arise for Bigtable, such as highly variable bandwidth, untrusted participants, or frequent reconfiguration; decentralized control and Byzantine fault tolerance are not Bigtable goals.

In terms of the distributed data storage model that one might provide to application developers, we believe the key-value pair model provided by distributed B-trees or distributed hash tables is too limiting. Key-value pairs are a useful building block, but they should not be the only building block one provides to developers. The model we chose is richer than simple key-value pairs, and supports sparse semi-structured data. Nonetheless, it is still simple enough that it lends itself to a very efficient flat-file representation, and it is transparent enough (via locality groups) to allow our users to tune important behaviors of the system.

Several database vendors have developed parallel databases that can store large volumes of data. Oracle's Real Application Cluster database [27] uses shared disks to store data (Bigtable uses GFS) and a distributed lock manager (Bigtable uses Chubby). IBM's DB2 Parallel Edition [4] is based on a shared-nothing [33] architecture similar to Bigtable. Each DB2 server is responsible for a subset of the rows in a table which it stores in a local relational database. Both products provide a complete relational model with transactions.

Bigtable locality groups realize similar compression and disk read performance benefits observed for other systems that organize data on disk using column-based rather than row-based storage, including C-Store [1, 34] and commercial products such as Sybase IQ [15, 36], SenSage [31], KDB+ [22], and the ColumnBM storage layer in MonetDB/X100 [38]. Another system that does vertical and horizontal data partioning into flat files and achieves good data compression ratios is AT&T's Daytona database [19]. Locality groups do not support CPU-cache-level optimizations, such as those described by Ailamaki [2].

The manner in which Bigtable uses memtables and SSTables to store updates to tablets is analogous to the way that the Log-Structured Merge Tree [26] stores updates to index data. In both systems, sorted data is buffered in memory before being written to disk, and reads must merge data from memory and disk.

C-Store and Bigtable share many characteristics: both systems use a shared-nothing architecture and have two different data structures, one for recent writes, and one for storing long-lived data, with a mechanism for moving data from one form to the other. The systems differ significantly in their API: C-Store behaves like a relational database, whereas Bigtable provides a lower level read and write interface and is designed to support many thousands of such operations per second per server. C-Store is also a "read-optimized relational DBMS", whereas Bigtable provides good performance on both read-intensive and write-intensive applications.

Bigtable's load balancer has to solve some of the same kinds of load and memory balancing problems faced by shared-nothing databases (e.g., [11, 35]). Our problem is somewhat simpler: (1) we do not consider the possibility of multiple copies of the same data, possibly in alternate forms due to views or indices; (2) we let the user tell us what data belongs in memory and what data should stay on disk, rather than trying to determine this dynamically; (3) we have no complex queries to execute or optimize.

## 11 Conclusions

We have described Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production use since April 2005, and we spent roughly seven person-years on design and implementation before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time.

Given the unusual interface to Bigtable, an interesting question is how difficult it has been for our users to adapt to using it. New users are sometimes uncertain of how to best use the Bigtable interface, particularly if they are accustomed to using relational databases that support general-purpose transactions. Nevertheless, the fact that many Google products successfully use Bigtable demonstrates that our design works well in practice.

We are in the process of implementing several additional Bigtable features, such as support for secondary indices and infrastructure for building cross-data-center replicated Bigtables with multiple master replicas. We have also begun deploying Bigtable as a service to product groups, so that individual groups do not need to maintain their own clusters. As our service clusters scale, we will need to deal with more resource-sharing issues within Bigtable itself [3, 5].

Finally, we have found that there are significant advantages to building our own storage solution at Google. We have gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addition, our control over Bigtable's implementation, and the other Google infrastructure upon which Bigtable depends, means that we can remove bottlenecks and inefficiencies as they arise.

## Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper. The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition, we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen, Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin, and Arthur Zwiegincew.

## References

[1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in column-oriented database systems. *Proc. of SIGMOD* (2006).

[2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. Weaving relations for cache performance. In *The VLDB Journal* (2001), pp. 169–180.

[3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd OSDI* (Feb. 1999), pp. 45–58.

[4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND,

G. P., AND WILSON, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2 (1995), 292–322.

[5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In *Proc. of the 1st NSDI* (Mar. 2004), pp. 253–266.

[6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In *Data Compression Conference* (1999), pp. 287–295.

[7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *CACM 13*, 7 (1970), 422–426.

[8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th OSDI* (Nov. 2006).

[9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J. Paxos made live — An engineering perspective. In *Proc. of PODC* (2007).

[10] COMER, D. Ubiquitous B-tree. *Computing Surveys 11*, 2 (June 1979), 121–137.

[11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In *Proc. of SIGMOD* (1988), pp. 99–108.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 137–150.

[13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proc. of SIGMOD* (June 1984), pp. 1–8.

[14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. *CACM 35*, 6 (June 1992), 85–98.

[15] FRENCH, C. D. One size fits all database architectures do not work for DSS. In *Proc. of SIGMOD* (May 1995), pp. 449–450.

[16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. *Database Engineering Bulletin 8*, 2 (1985), 3–10.

[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. of the 19th ACM SOSP* (Dec. 2003), pp. 29–43.

[18] GRAY, J. Notes on database operating systems. In *Operating Systems — An Advanced Course*, vol. 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[19] GREER, R. Daytona and the fourth-generation language Cymbal. In *Proc. of SIGMOD* (1999), pp. 525–526.

[20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th SOSP* (Dec. 1987), pp. 155–162.

[21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In *Proc. of the 14th SOSP* (Asheville, NC, 1993), pp. 29–43.

[22] KX.COM. kx.com/products/database.php. Product page.

[23] LAMPORT, L. The part-time parliament. *ACM TOCS 16*, 2 (1998), 133–169.

[24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. of the 6th OSDI* (Dec. 2004), pp. 105–120.

[25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *CACM 3*, 4 (Apr. 1960), 184–195.

[26] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4 (1996), 351–385.

[27] ORACLE.COM. www.oracle.com/technology/products/-database/clustering/index.html. Product page.

[28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal 13*, 4 (2005), 227–298.

[29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug. 2001), pp. 161–172.

[30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware 2001* (Nov. 2001), pp. 329–350.

[31] SENSAGE.COM. sensage.com/products-sensage.htm. Product page.

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of SIGCOMM* (Aug. 2001), pp. 149–160.

[33] STONEBRAKER, M. The case for shared nothing. *Database Engineering Bulletin 9*, 1 (Mar. 1986), 4–9.

[34] STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A column-oriented DBMS. In *Proc. of VLDB* (Aug. 2005), pp. 553–564.

[35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In *Proc. of the Tenth ICDE* (1994), IEEE Computer Society, pp. 54–65.

[36] SYBASE.COM. www.sybase.com/products/database-servers/sybaseiq. Product page.

[37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.

[38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 — A DBMS in the CPU cache. *IEEE Data Eng. Bull. 28*, 2 (2005), 17–22.

# The PageRank Citation Ranking: Bringing Order to the Web

January 29, 1998

**Abstract**

The importance of a Web page is an inherently subjective matter, which depends on the readers interests, knowledge and attitudes. But there is still much that can be said objectively about the relative importance of Web pages. This paper describes PageRank, a method for rating Web pages objectively and mechanically, effectively measuring the human interest and attention devoted to them.

We compare PageRank to an idealized random Web surfer. We show how to efficiently compute PageRank for large numbers of pages. And, we show how to apply PageRank to search and to user navigation.

## 1 Introduction and Motivation

The World Wide Web creates many new challenges for information retrieval. It is very large and heterogeneous. Current estimates are that there are over 150 million web pages with a doubling life of less than one year. More importantly, the web pages are extremely diverse, ranging from "What is Joe having for lunch today?" to journals about information retrieval. In addition to these major challenges, search engines on the Web must also contend with inexperienced users and pages engineered to manipulate search engine ranking functions.

However, unlike "flat" document collections, the World Wide Web is hypertext and provides considerable auxiliary information on top of the text of the web pages, such as link structure and link text. In this paper, we take advantage of the link structure of the Web to produce a global "importance" ranking of every web page. This ranking, called PageRank, helps search engines and users quickly make sense of the vast heterogeneity of the World Wide Web.

### 1.1 Diversity of Web Pages

Although there is already a large literature on academic citation analysis, there are a number of significant differences between web pages and academic publications. Unlike academic papers which are scrupulously reviewed, web pages proliferate free of quality control or publishing costs. With a simple program, huge numbers of pages can be created easily, artificially inflating citation counts. Because the Web environment contains competing profit seeking ventures, attention getting strategies evolve in response to search engine algorithms. For this reason, any evaluation strategy which counts replicable features of web pages is prone to manipulation. Further, academic papers are well defined units of work, roughly similar in quality and number of citations, as well as in their purpose – to extend the body of knowledge. Web pages vary on a much wider scale than academic papers in quality, usage, citations, and length. A random archived message posting

asking an obscure question about an IBM computer is very different from the IBM home page. A research article about the effects of cellular phone use on driver attention is very different from an advertisement for a particular cellular provider. The average web page quality experienced by a user is higher than the quality of the average web page. This is because the simplicity of creating and publishing web pages results in a large fraction of low quality web pages that users are unlikely to read.

There are many axes along which web pages may be differentiated. In this paper, we deal primarily with one - an approximation of the overall relative importance of web pages.

### 1.2 PageRank

In order to measure the relative importance of web pages, we propose PageRank, a method for computing a ranking for every web page based on the graph of the web. PageRank has applications in search, browsing, and traffic estimation.

Section 2 gives a mathematical description of PageRank and provides some intuitive justification. In Section 3, we show how we efficiently compute PageRank for as many as 518 million hyperlinks. To test the utility of PageRank for search, we built a web search engine called Google (Section 5). We also demonstrate how PageRank can be used as a browsing aid in Section 7.3.

## 2 A Ranking for Every Page on the Web

### 2.1 Related Work

There has been a great deal of work on academic citation analysis [Gar95]. Goffman [Gof71] has published an interesting theory of how information flow in a scientific community is an epidemic process.

There has been a fair amount of recent activity on how to exploit the link structure of large hypertext systems such as the web. Pitkow recently completed his Ph.D. thesis on "Characterizing World Wide Web Ecologies" [Pit97, PPR96] with a wide variety of link based analysis. Weiss discuss clustering methods that take the link structure into account [WVS+96]. Spertus [Spe97] discusses information that can be obtained from the link structure for a variety of applications. Good visualization demands added structure on the hypertext and is discussed in [MFH95, MF95]. Recently, Kleinberg [Kle98] has developed an interesting model of the web as Hubs and Authorities, based on an eigenvector calculation on the co-citation matrix of the web.

Finally, there has been some interest in what "quality" means on the net from a library community [Til].

It is obvious to try to apply standard citation analysis techniques to the web's hypertextual citation structure. One can simply think of every link as being like an academic citation. So, a major page like http://www.yahoo.com/ will have tens of thousands of backlinks (or citations) pointing to it.

This fact that the Yahoo home page has so many backlinks generally imply that it is quite important. Indeed, many of the web search engines have used backlink count as a way to try to bias their databases in favor of higher quality or more important pages. However, simple backlink counts have a number of problems on the web. Some of these problems have to do with characteristics of the web which are not present in normal academic citation databases.

## 2.2 Link Structure of the Web

While estimates vary, the current graph of the crawlable Web has roughly 150 million nodes (pages) and 1.7 billion edges (links). Every page has some number of forward links (outedges) and backlinks (inedges) (see Figure 1). We can never know whether we have found all the backlinks of a particular page but if we have downloaded it, we know all of its forward links at that time.

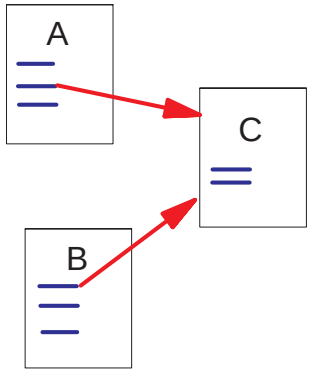



Figure 1: A and B are Backlinks of C

Web pages vary greatly in terms of the number of backlinks they have. For example, the Netscape home page has 62,804 backlinks in our current database compared to most pages which have just a few backlinks. Generally, highly linked pages are more "important" than pages with few links. Simple citation counting has been used to speculate on the future winners of the Nobel Prize [San95]. PageRank provides a more sophisticated method for doing citation counting.

The reason that PageRank is interesting is that there are many cases where simple citation counting does not correspond to our common sense notion of importance. For example, if a web page has a link off the Yahoo home page, it may be just one link but it is a very important one. This page should be ranked higher than many pages with more links but from obscure places. PageRank is an attempt to see how good an approximation to "importance" can be obtained just from the link structure.

## 2.3 Propagation of Ranking Through Links

Based on the discussion above, we give the following intuitive description of PageRank: a page has high rank if the sum of the ranks of its backlinks is high. This covers both the case when a page has many backlinks and when a page has a few highly ranked backlinks.

## 2.4 Definition of PageRank

Let $u$ be a web page. Then let $F_u$ be the set of pages $u$ points to and $B_u$ be the set of pages that point to $u$. Let $N_u = |F_u|$ be the number of links from $u$ and let $c$ be a factor used for normalization (so that the total rank of all web pages is constant).

We begin by defining a simple ranking, $R$ which is a slightly simplified version of PageRank:

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v}$$

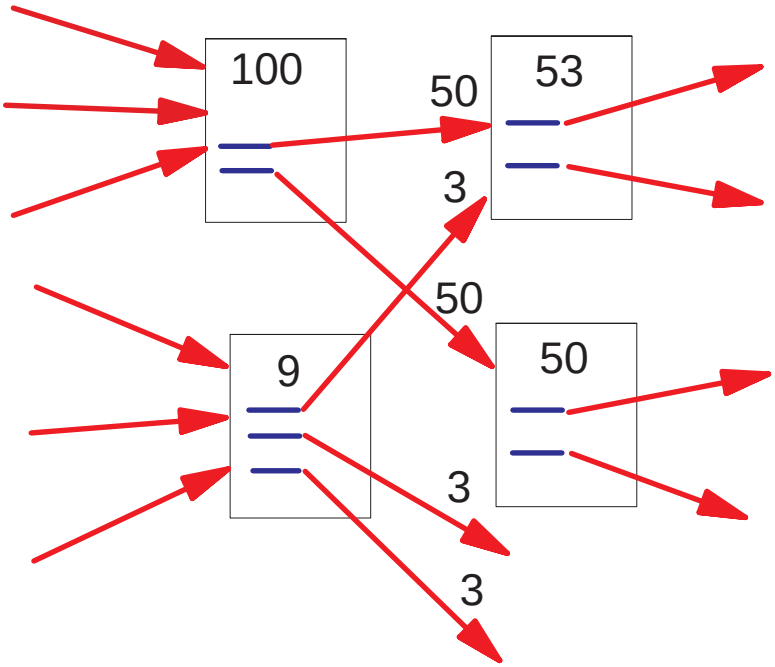



Figure 2: Simplified PageRank Calculation

This formalizes the intuition in the previous section. Note that the rank of a page is divided among its forward links evenly to contribute to the ranks of the pages they point to. Note that $c < 1$ because there are a number of pages with no forward links and their weight is lost from the system (see section 2.7). The equation is recursive but it may be computed by starting with any set of ranks and iterating the computation until it converges. Figure 2 demonstrates the propagation of rank from one pair of pages to another. Figure 3 shows a consistent steady state solution for a set of pages.

Stated another way, let $A$ be a square matrix with the rows and column corresponding to web pages. Let $A_{u,v} = 1/N_u$ if there is an edge from $u$ to $v$ and $A_{u,v} = 0$ if not. If we treat $R$ as a vector over web pages, then we have $R = cAR$. So $R$ is an eigenvector of $A$ with eigenvalue $c$. In fact, we want the dominant eigenvector of $A$. It may be computed by repeatedly applying $A$ to any nondegenerate start vector.

There is a small problem with this simplified ranking function. Consider two web pages that point to each other but to no other page. And suppose there is some web page which points to one of them. Then, during iteration, this loop will accumulate rank but never distribute any rank (since there are no outedges). The loop forms a sort of trap which we call a rank sink.

To overcome this problem of rank sinks, we introduce a rank source:

**Definition 1** *Let $E(u)$ be some vector over the Web pages that corresponds to a source of rank. Then, the PageRank of a set of Web pages is an assignment, $R'$, to the Web pages which satisfies*

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u) \tag{1}$$

*such that $c$ is maximized and $||R'||_1 = 1$ ($||R'||_1$ denotes the $L_1$ norm of $R'$).*

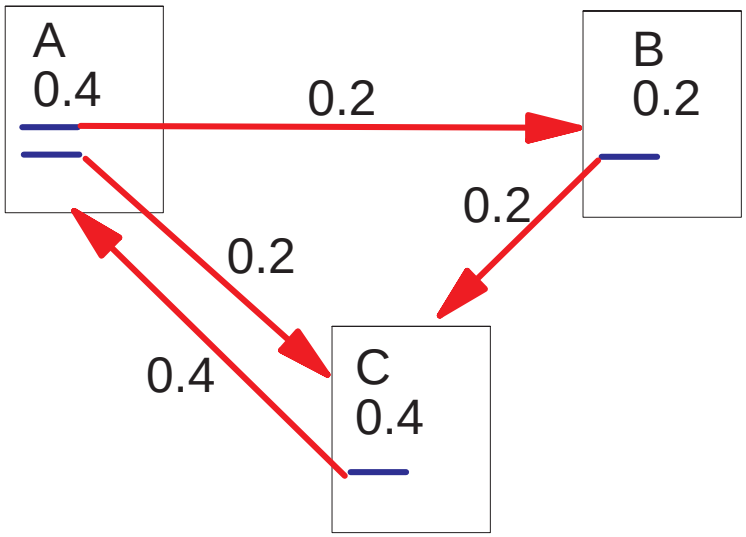



Figure 3: Simplified PageRank Calculation

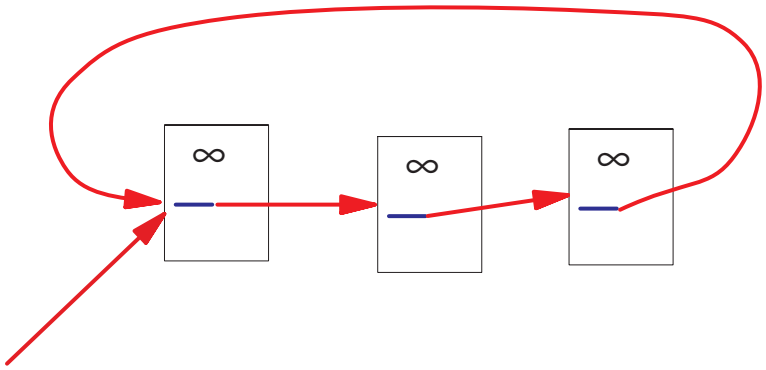



Figure 4: Loop Which Acts as a Rank Sink

where $E(u)$ is some vector over the web pages that corresponds to a source of rank (see Section 6). Note that if $E$ is all positive, $c$ must be reduced to balance the equation. Therefore, this technique corresponds to a decay factor. In matrix notation we have $R' = c(AR' + E)$. Since $||R'||_1 = 1$, we can rewrite this as $R' = c(A + E \times \mathbf{1})R'$ where $\mathbf{1}$ is the vector consisting of all ones. So, $R'$ is an eigenvector of $(A + E \times \mathbf{1})$.

### 2.5 Random Surfer Model

The definition of PageRank above has another intuitive basis in random walks on graphs. The simplified version corresponds to the standing probability distribution of a random walk on the graph of the Web. Intuitively, this can be thought of as modeling the behavior of a "random surfer". The "random surfer" simply keeps clicking on successive links at random. However, if a real Web surfer ever gets into a small loop of web pages, it is unlikely that the surfer will continue in the loop forever. Instead, the surfer will jump to some other page. The additional factor $E$ can be viewed as a way of modeling this behavior: the surfer periodically "gets bored" and jumps to a

random page chosen based on the distribution in $E$.

So far we have left $E$ as a user defined parameter. In most tests we let $E$ be uniform over all web pages with value $\alpha$. However, in Section 6 we show how different values of $E$ can generate "customized" page ranks.

### 2.6 Computing PageRank

The computation of PageRank is fairly straightforward if we ignore the issues of scale. Let $S$ be almost any vector over Web pages (for example $E$). Then PageRank may be computed as follows:

$$\begin{aligned} R_0 &\leftarrow S \\ \text{loop}: & \\ R_{i+1} &\leftarrow AR_i \\ d &\leftarrow ||R_i||_1 - ||R_{i+1}||_1 \\ R_{i+1} &\leftarrow R_{i+1} + dE \\ \delta &\leftarrow ||R_{i+1} - R_i||_1 \\ \text{while } \delta > \epsilon & \end{aligned}$$

Note that the $d$ factor increases the rate of convergence and maintains $||R||_1$. An alternative normalization is to multiply $R$ by the appropriate factor. The use of $d$ may have a small impact on the influence of $E$.

### 2.7 Dangling Links

One issue with this model is dangling links. Dangling links are simply links that point to any page with no outgoing links. They affect the model because it is not clear where their weight should be distributed, and there are a large number of them. Often these dangling links are simply pages that we have not downloaded yet, since it is hard to sample the entire web (in our 24 million pages currently downloaded, we have 51 million URLs not downloaded yet, and hence dangling). Because dangling links do not affect the ranking of any other page directly, we simply remove them from the system until all the PageRanks are calculated. After all the PageRanks are calculated, they can be added back in, without affecting things significantly. Notice the normalization of the other links on the same page as a link which was removed will change slightly, but this should not have a large effect.

## 3 Implementation

As part of the Stanford WebBase project [PB98], we have built a complete crawling and indexing system with a current repository of 24 million web pages. Any web crawler needs to keep a database of URLs so it can discover all the URLs on the web. To implement PageRank, the web crawler simply needs to build an index of links as it crawls. While a simple task, it is non-trivial because of the huge volumes involved. For example, to index our current 24 million page database in about five days, we need to process about 50 web pages per second. Since there about about 11 links on an average page (depending on what you count as a link) we need to process 550 links per second. Also, our database of 24 million pages references over 75 million unique URLs which each link must be compared against.

Much time has been spent making the system resilient in the face of many deeply and intricately flawed web artifacts. There exist infinitely large sites, pages, and even URLs. A large fraction of web pages have incorrect HTML, making parser design difficult. Messy heuristics are used to help the crawling process. For example, we do not crawl URLs with /cgi-bin/ in them. Of course it is impossible to get a correct sample of the "entire web" since it is always changing. Sites are sometimes down, and some people decide to not allow their sites to be indexed. Despite all this, we believe we have a reasonable representation of the actual link structure of publicly accessible web.

### 3.1 PageRank Implementation

We convert each URL into a unique integer, and store each hyperlink in a database using the integer IDs to identify pages. Details of our implementation are in [PB98]. In general, we have implemented PageRank in the following manner. First we sort the link structure by Parent ID. Then dangling links are removed from the link database for reasons discussed above (a few iterations removes the vast majority of the dangling links). We need to make an initial assignment of the ranks. This assignment can be made by one of several strategies. If it is going to iterate until convergence, in general the initial values will not affect final values, just the rate of convergence. But we can speed up convergence by choosing a good initial assignment. We believe that careful choice of the initial assignment and a small finite number of iterations may result in excellent or improved performance.

Memory is allocated for the weights for every page. Since we use single precision floating point values at 4 bytes each, this amounts to 300 megabytes for our 75 million URLs. If insufficient RAM is available to hold all the weights, multiple passes can be made (our implementation uses half as much memory and two passes). The weights from the current time step are kept in memory, and the previous weights are accessed linearly on disk. Also, all the access to the link database, $A$, is linear because it is sorted. Therefore, $A$ can be kept on disk as well. Although these data structures are very large, linear disk access allows each iteration to be completed in about 6 minutes on a typical workstation. After the weights have converged, we add the dangling links back in and recompute the rankings. Note after adding the dangling links back in, we need to iterate as many times as was required to remove the dangling links. Otherwise, some of the dangling links will have a zero weight. This whole process takes about five hours in the current implementation. With less strict convergence criteria, and more optimization, the calculation could be much faster. Or, more efficient techniques for estimating eigenvectors could be used to improve performance. However, it should be noted that the cost required to compute the PageRank is insignificant compared to the cost required to build a full text index.

## 4 Convergence Properties

As can be seen from the graph in Figure 4 PageRank on a large 322 million link database converges to a reasonable tolerance in roughly 52 iterations. The convergence on half the data takes roughly 45 iterations. This graph suggests that PageRank will scale very well even for extremely large collections as the scaling factor is roughly linear in $\log n$.

One of the interesting ramifications of the fact that the PageRank calculation converges rapidly is that the web is an expander-like graph. To understand this better, we give a brief overview of the theory of random walks on graphs; refer to Motwani-Raghavan [MR95] for further details. A random walk on a graph is a stochastic process where at any given time step we are at a particular node of the graph and choose an outedge uniformly at random to determine the node to visit at the next time step. A graph is said to be an expander if it is the case that every (not too large) subset of nodes $S$ has a neighborhood (set of vertices accessible via outedges emanating from nodes

in $S$) that is larger than some factor $\alpha$ times $|S|$; here, $\alpha$ is called the expansion factor. It is the case that a graph has a good expansion factor if and only if the largest eigenvalue is sufficiently larger than the second-largest eigenvalue. A random walk on a graph is said to be rapidly-mixing if it quickly (time logarithmic in the size of the graph) converges to a limiting distribution on the set of nodes in the graph. It is also the case that a random walk is rapidly-mixing on a graph if and only if the graph is an expander or has an eigenvalue separation.

To relate all this to the PageRank computation, note that it is essentially the determination of the limiting distribution of a random walk on the Web graph. The importance ranking of a node is essentially the limiting probability that the random walk will be at that node after a sufficiently large time. The fact that the PageRank computation terminates in logarithmic time is equivalent to saying that the random walk is rapidly mixing or that the underlying graph has a good expansion factor. Expander graphs have many desirable properties that we may be able to exploit in the future in computations involving the Web graph.

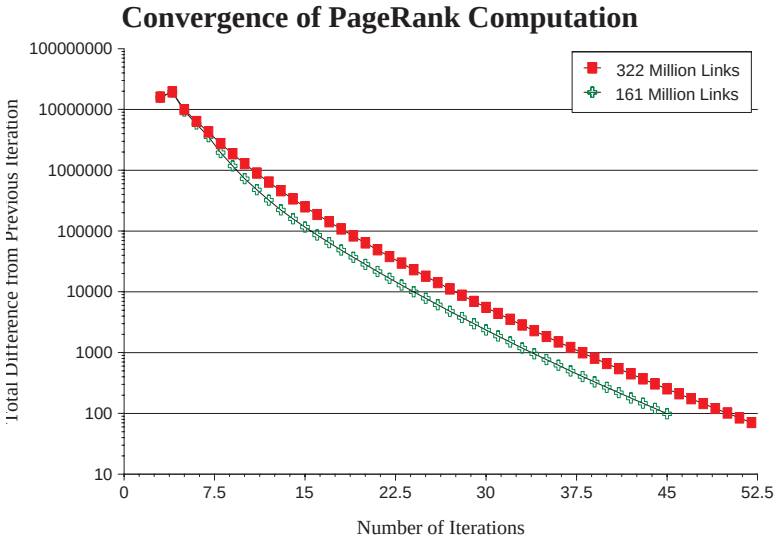



Figure 5: Rates of Convergence for Full Size and Half Size Link Databases

## 5 Searching with PageRank

A major application of PageRank is searching. We have implemented two search engines which use PageRank. The first one we will discuss is a simple title-based search engine. The second search engine is a full text search engine called Google [BP]. Google utilizes a number of factors to rank search results including standard IR measures, proximity, anchor text (text of links pointing to web pages), and PageRank. While a comprehensive user study of the benefits of PageRank is beyond the scope of this paper, we have performed some comparative experiments and provide some sample results in this paper.

The benefits of PageRank are the greatest for underspecified queries. For example, a query for "Stanford University" may return any number of web pages which mention Stanford (such as publication lists) on a conventional search engine, but using PageRank, the university home page is listed first.

## 5.1 Title Search

To test the usefulness of PageRank for search we implemented a search engine that used only the titles of 16 million web pages. To answer a query, the search engine finds all the web pages whose titles contain all of the query words. Then it sorts the results by PageRank. This search engine is very simple and cheap to implement. In informal tests, it worked remarkably well. As can be seen in Figure 6, a search for "University" yields a list of top universities. This figure shows our MultiQuery system which allows a user to query two search engines at the same time. The search engine on the left is our PageRank based title search engine. The bar graphs and percentages shown are a log of the actual PageRank with the top page normalized to 100%, not a percentile which is used everywhere else in this paper. The search engine on the right is Altavista. You can see that Altavista returns random looking web pages that match the query "University" and are the root page of the server (Altavista seems to be using URL length as a quality heuristic).

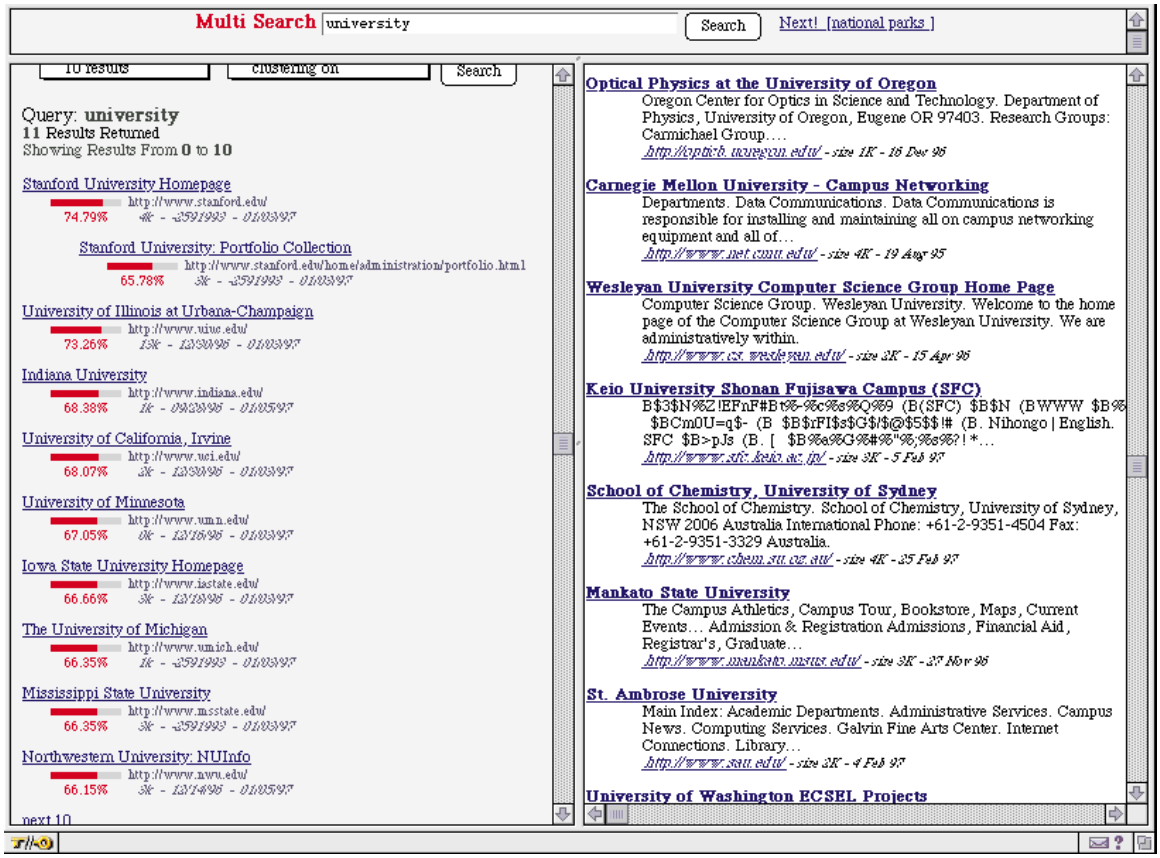



Figure 6: Comparison of Query for "University"

| Web Page | PageRank (average is 1.0) |
|---|---|
| Download Netscape Software | 11589.00 |
| http://www.w3.org/ | 10717.70 |
| Welcome to Netscape | 8673.51 |
| Point: It's What You're Searching For | 7930.92 |
| Web-Counter Home Page | 7254.97 |
| The Blue Ribbon Campaign for Online Free Speech | 7010.39 |
| CERN Welcome | 6562.49 |
| Yahoo! | 6561.80 |
| Welcome to Netscape | 6203.47 |
| Wusage 4.1: A Usage Statistics System For Web Servers | 5963.27 |
| The World Wide Web Consortium (W3C) | 5672.21 |
| Lycos, Inc. Home Page | 4683.31 |
| Starting Point | 4501.98 |
| Welcome to Magellan! | 3866.82 |
| Oracle Corporation | 3587.63 |

Table 1: Top 15 Page Ranks: July 1996

### 5.2 Rank Merging

The reason that the title based PageRank system works so well is that the title match ensures high precision, and the PageRank ensures high quality. When matching a query like "University" on the web, recall is not very important because there is far more than a user can look at. For more specific searches where recall is more important, the traditional information retrieval scores over full-text and the PageRank should be combined. Our Google system does this type of rank merging. Rank merging is known to be a very difficult problem, and we need to spend considerable additional effort before we will be able to do a reasonable evaluation of these types of queries. However, we do believe that using PageRank as a factor in these queries is quite beneficial.

### 5.3 Some Sample Results

We have experimented considerably with Google, a full-text search engine which uses PageRank. While a full-scale user study is beyond the scope of this paper, we provide a sample query in Appendix A. For more queries, we encourage the reader to test Google themselves [BP].

Table 1 shows the top 15 pages based on PageRank. This particular listing was generated in July 1996. In a more recent calculation of PageRank, Microsoft has just edged out Netscape for the highest PageRank.

### 5.4 Common Case

One of the design goals of PageRank was to handle the common case for queries well. For example, a user searched for "wolverine", remembering that the University of Michigan system used for all administrative functions by students was called something with a wolverine in it. Our PageRank based title search system returned the answer "Wolverine Access" as the first result. This is sensible since all the students regularly use the Wolverine Access system, and a random user is quite likely to be looking for it given the query "wolverine". The fact that the Wolverine Access site is a good common case is not contained in the HTML of the page. Even if there were a way of defining good

meta-information of this form within a page, it would be problematic since a page author could not be trusted with this kind of evaluation. Many web page authors would simply claim that their pages were all the best and most used on the web.

It is important to note that the goal of finding a site that contains a great deal of information about wolverines is a very different task than finding the common case wolverine site. There is an interesting system [Mar97] that attempts to find sites that discuss a topic in detail by propagating the textual matching score through the link structure of the web. It then tries to return the page on the most central path. This results in good results for queries like "flower"; the system will return good navigation pages from sites that deal with the topic of flowers in detail. Contrast that with the common case approach which might simply return a commonly used commercial site that had little information except how to buy flowers. It is our opinion that both of these tasks are important, and a general purpose web search engine should return results which fulfill the needs of both of these tasks automatically. In this paper, we are concentrating only on the common case approach.

### 5.5 Subcomponents of Common Case

It is instructive to consider what kind of common case scenarios PageRank can help represent. Besides a page which has a high usage, like the Wolverine Access cite, PageRank can also represent a collaborative notion of authority or trust. For example, a user might prefer a news story simply because it is linked is linked directly from the New York Times home page. Of course such a story will receive quite a high PageRank simply because it is mentioned by a very important page. This seems to capture a kind of collaborative trust, since if a page was mentioned by a trustworthy or authoritative source, it is more likely to be trustworthy or authoritative. Similarly, quality or importance seems to fit within this kind of circular definition.

## 6 Personalized PageRank

An important component of the PageRank calculation is $E$ – a vector over the Web pages which is used as a source of rank to make up for the rank sinks such as cycles with no outedges (see Section 2.4). However, aside from solving the problem of rank sinks, $E$ turns out to be a powerful parameter to adjust the page ranks. Intuitively the $E$ vector corresponds to the distribution of web pages that a random surfer periodically jumps to. As we see below, it can be used to give broad general views of the Web or views which are focussed and personalized to a particular individual.

We have performed most experiments with an $E$ vector that is uniform over all web pages with $||E||_1 = 0.15$. This corresponds to a random surfer periodically jumping to a random web page. This is a very democratic choice for $E$ since all web pages are valued simply because they exist. Although this technique has been quite successful, there is an important problem with it. Some Web pages with many related links receive an overly high ranking. Examples of these include copyright warnings, disclaimers, and highly interlinked mailing list archives.

Another extreme is to have $E$ consist entirely of a single web page. We tested two such $E$'s – the Netscape home page, and the home page of a famous computer scientist, John McCarthy. For the Netscape home page, we attempt to generate page ranks from the perspective of a novice user who has Netscape set as the default home page. In the case of John McCarthy's home page we want to calculate page ranks from the perspective of an individual who has given us considerable contextual information based on the links on his home page.

In both cases, the mailing list problem mentioned above did not occur. And, in both cases, the respective home page got the highest PageRank and was followed by its immediate links. From

| Web Page Title | John McCarthy's View PageRank Percentile | Netscape's View PageRank Percentile |
|---|---|---|
| John McCarthy's Home Page | 100.00% | 99.23% |
| John Mitchell (Stanford CS Theory Group) | 100.00% | 93.89% |
| Venture Law (Local Startup Law Firm) | 99.94% | 99.82% |
| Stanford CS Home Page | 100.00% | 99.83% |
| University of Michigan AI Lab | 99.95% | 99.94% |
| University of Toronto CS Department | 99.99% | 99.09% |
| Stanford CS Theory Group | 99.99% | 99.05% |
| Leadershape Institute | 95.96% | 97.10% |

Table 2: Page Ranks for Two Different Views: Netscape vs. John McCarthy

that point, the disparity decreased. In Table 2, we show the resulting page rank percentiles for an assortment of different pages. Pages related to computer science have a higher McCarthy-rank than Netscape-rank and pages related to computer science at Stanford have a considerably higher McCarthy-rank. For example, the Web page of another Stanford Computer Science Dept. faculty member is more than six percentile points higher on the McCarthy-rank. Note that the page ranks are displayed as percentiles. This has the effect of compressing large differences in PageRank at the top of the range.

Such personalized page ranks may have a number of applications, including personal search engines. These search engines could save users a great deal of trouble by efficiently guessing a large part of their interests given simple input such as their bookmarks or home page. We show an example of this in Appendix A with the "Mitchell" query. In this example, we demonstrate that while there are many people on the web named Mitchell, the number one result is the home page of a colleague of John McCarthy named John Mitchell.

### 6.1 Manipulation by Commercial Interests

These types of personalized PageRanks are virtually immune to manipulation by commercial interests. For a page to get a high PageRank, it must convince an important page, or a lot of non-important pages to link to it. At worst, you can have manipulation in the form of buying advertisements(links) on important sites. But, this seems well under control since it costs money. This immunity to manipulation is an extremely important property. This kind of commercial manipulation is causing search engines a great deal of trouble, and making features that would be great to have very difficult to implement. For example fast updating of documents is a very desirable feature, but it is abused by people who want to manipulate the results of the search engine.

A compromise between the two extremes of uniform $E$ and single page $E$ is to let $E$ consist of all the root level pages of all web servers. Notice this will allow some manipulation of PageRanks. Someone who wished to manipulate this system could simply create a large number of root level servers all pointing at a particular site.

# 7 Applications

## 7.1 Estimating Web Traffic

Because PageRank roughly corresponds to a random web surfer (see Section 2.5), it is interesting to see how PageRank corresponds to actual usage. We used the counts of web page accesses from NLANR [NLA] proxy cache and compared these to PageRank. The NLANR data was from several national proxy caches over the period of several months and consisted of 11,817,665 unique URLs with the highest hit count going to Altavista with 638,657 hits. There were 2.6 million pages in the intersection of the cache data and our 75 million URL database. It is extremely difficult to compare these datasets analytically for a number of different reasons. Many of the URLs in the cache access data are people reading their personal mail on free email services. Duplicate server names and page names are a serious problem. Incompleteness and bias a problem is both the PageRank data and the usage data. However, we did see some interesting trends in the data. There seems to be a high usage of pornographic sites in the cache data, but these sites generally had low PageRanks. We believe this is because people do not want to link to pornographic sites from their own web pages. Using this technique of looking for differences between PageRank and usage, it may be possible to find things that people like to look at, but do not want to mention on their web pages. There are some sites that have a very high usage, but low PageRank such as netscape.yahoo.com. We believe there is probably an important backlink which simply is omitted from our database (we only have a partial link structure of the web). It may be possible to use usage data as a start vector for PageRank, and then iterate PageRank a few times. This might allow filling in holes in the usage data. In any case, these types of comparisons are an interesting topic for future study.

## 7.2 PageRank as Backlink Predictor

One justification for PageRank is that it is a predictor for backlinks. In [CGMP98] we explore the issue of how to crawl the web efficiently, trying to crawl better documents first. We found on tests of the Stanford web that PageRank is a better predictor of future citation counts than citation counts themselves.

The experiment assumes that the system starts out with only a single URL and no other information, and the goal is to try to crawl the pages in as close to the optimal order as possible. The optimal order is to crawl pages in exactly the order of their rank according to an evaluation function. For the purposes here, the evaluation function is simply the number of citations, given complete information. The catch is that all the information to calculate the evaluation function is not available until after all the documents have been crawled. It turns out using the incomplete data, PageRank is a more effective way to order the crawling than the number of known citations. In other words, PageRank is a better predictor than citation counting even when the measure is the number of citations! The explanation for this seems to be that PageRank avoids the local maxima that citation counting gets stuck in. For example, citation counting tends to get stuck in local collections like the Stanford CS web pages, taking a long time to branch out and find highly cited pages in other areas. PageRank quickly finds the Stanford homepage is important, and gives preference to its children resulting in an efficient, broad search.

This ability of PageRank to predict citation counts is a powerful justification for using PageRank. Since it is very difficult to map the citation structure of the web completely, PageRank may even be a better citation count approximation than citation counts themselves.

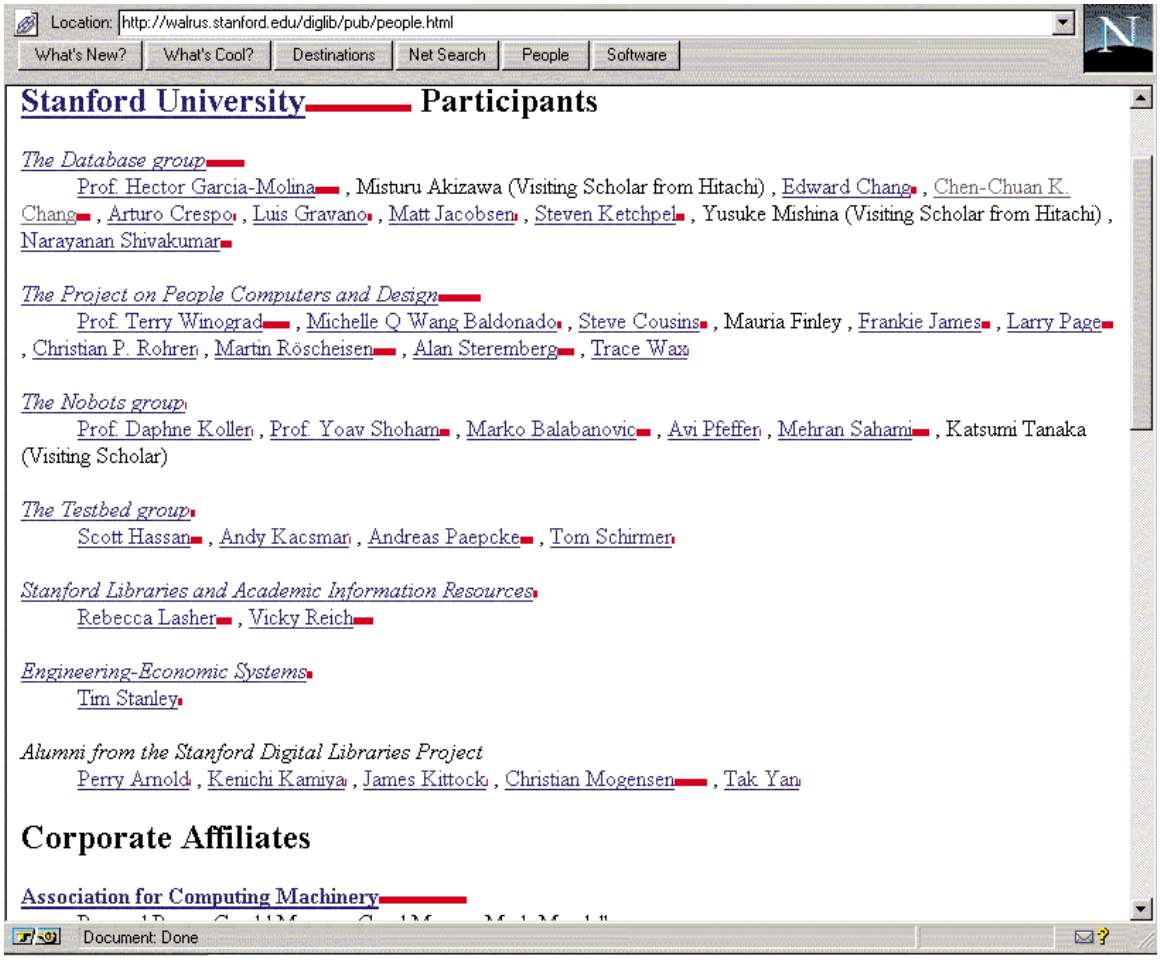



Figure 7: PageRank Proxy

### 7.3 User Navigation: The PageRank Proxy

We have developed a web proxy application that annotates each link that a user sees with its PageRank. This is quite useful, because users receive some information about the link before they click on it. In Figure 7 is a screen shot from the proxy. The length of the red bars is the log of the URL's PageRank. We can see that major organizations, like Stanford University, receive a very high ranking followed by research groups, and then people, with professors at the high end of the people scale. Also notice ACM has a very high PageRank, but not as high as Stanford University. Interestingly, this PageRank annotated view of the page makes an incorrect URL for one of the professors glaringly obvious since the professor has a embarrassingly low PageRank. Consequently this tool seems useful for authoring pages as well as navigation. This proxy is very helpful for looking at the results from other search engines, and pages with large numbers of links such as Yahoo's listings. The proxy can help users decide which links in a long listing are more likely to be interesting. Or, if the user has some idea where the link they are looking for should fall in the "importance" spectrum, they should be able to scan for it much more quickly using the proxy.

### 7.4 Other Uses of PageRank

The original goal of PageRank was a way to sort backlinks so if there were a large number of backlinks for a document, the "best" backlinks could be displayed first. We have implemented such a system. It turns out this view of the backlinks ordered by PageRank can be very interesting when trying to understand your competition. For example, the people who run a news site always want to keep track of any significant backlinks the competition has managed to get. Also, PageRank can help the user decide if a site is trustworthy or not. For example, a user might be inclined to trust information that is directly cited from the Stanford homepage.

## 8 Conclusion

In this paper, we have taken on the audacious task of condensing every page on the World Wide Web into a single number, its PageRank. PageRank is a global ranking of all web pages, regardless of their content, based solely on their location in the Web's graph structure.

Using PageRank, we are able to order search results so that more important and central Web pages are given preference. In experiments, this turns out to provide higher quality search results to users. Th intuition behind PageRank is that it uses information which is exernal to the Web pages themselves - their backlinks, which provide a kind of peer review. Furthermore, backlinks from "important" pages are more significant than backlinks from average pages. This is encompassed in the recursive definition of PageRank (Section 2.4).

PageRank could be used to separate out a small set of commonly used documents which can answer most queries. The full database only needs to be consulted when the small database is not adequate to answer a query. Finally, PageRank may be a good way to help find representative pages to display for a cluster center.

We have found a number of applications for PageRank in addition to search which include traffic estimation, and user navigation. Also, we can generate personalized PageRanks which can create a view of Web from a particular perspective.

Overall, our experiments with PageRank suggest that the structure of the Web graph is very useful for a variety of information retrieval tasks.

## References


[BP] Sergey Brin and Larry Page. Google search engine. `http://google.stanford.edu`.

[CGMP98] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. In *To Appear: Proceedings of the Seventh International Web Conference (WWW 98)*, 1998.

[Gar95] Eugene Garfield. New international professional society signals the maturing of scientometrics and informetrics. *The Scientist*, 9(16), Aug 1995. `http://www.the-scientist.library.upenn.edu/yr1995/august/issi_950821.ht%ml`.

[Gof71] William Goffman. A mathematical method for analyzing the growth of a scientific discipline. *Journal of the ACM*, 18(2):173–185, April 1971.

[Kle98] Jon Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the Nineth Annual ACM-SIAM Symposium on Discrete Algorithms.*, 1998.

[Mar97] Massimo Marchiori. The quest for correct information on the web: Hyper search engines. In *Proceedings of the Sixth International WWW Conference, Santa Claram USA, April, 1997*, 1997. `http://www6.nttlabs.com/HyperNews/get/PAPER222.html`.

[MF95] Sougata Mukherjea and James D. Foley. Showing the context of nodes in the world-wide web. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 2 of *Short Papers: Web Browsing*, pages 326–327, 1995.

[MFH95] Sougata Mukherjea, James D. Foley, and Scott Hudson. Visualizing complex hypermedia networks through multiple hierarchical views. In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1 of *Papers: Creating Visualizations*, pages 331–337, 1995.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[NLA] NLANR. A distributed testbed for national information provisioning. `http://ircache.nlanr.net/Cache/`.

[PB98] Lawrence Page and Sergey Brin. The anatomy of a large-scale hypertextual web search engine. In *To Appear: Proceedings of the Seventh International Web Conference (WWW 98)*, 1998.

[Pit97] James E. Pitkow. *Characterizing World Wide Web Ecologies*. PhD thesis, Georgia Institue of Technology, June 1997.

[PPR96] Peter Pirolli, James Pitkow, and Ramana Rao. Silk from a sow's ear: Extracting usable structure from the web. In Michael J. Tauber, Victoria Bellotti, Robin Jeffries, Jock D. Mackinlay, and Jakob Nielsen, editors, *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, pages 118–125, New York, 13–18 April 1996. ACM Press.

[San95] Neeraja Sankaran. Speculation in the biomedical community abounds over likely candidates for nobel. *The Scientist*, 9(19), Oct 1995. `http://www.the-scientist.library.upenn.edu/yr1995/oct/nobel_951002.html%`.

[Spe97] Ellen Spertus. Parasite: Mining structural information on the web. In *Proceedings of the Sixth International WWW Conference, Santa Claram USA, April, 1997*, 1997. `http://www6.nttlabs.com/HyperNews/get/PAPER206.html`.

[Til] Hope N. Tillman. Evaluating quality on the net. `http://www.tiac.net/users/hope/findqual.html`.

[WVS+96] Ron Weiss, Bienvenido Vélez, Mark A. Sheldon, Chanathip Manprempre, Peter Szilagyi, Andrzej Duda, and David K. Gifford. HyPursuit: A hierarchical network search engine that exploits content-link hypertext clustering. In *Proceedings of the 7th ACM Conference on Hypertext*, pages 180–193, New York, 16–20 March 1996. ACM Press.

# Appendix A

This Appendix contains several sample query results using Google. The first is a query for "Digital Libraries" using a uniform $E$. The next is a query for "Mitchell" using $E$ consisting just of John McCarthy's home page.

# Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

## ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

## Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

## General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

## 1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.


*SOSP'07,* October 14–17, 2007, Stevenson, Washington, USA.


One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

a gossip based distributed failure detection and membership protocol. Dynamo is a completely decentralized system with minimal need for manual administration. Storage nodes can be added and removed from Dynamo without requiring any manual partitioning or redistribution.

In the past year, Dynamo has been the underlying storage technology for a number of the core services in Amazon's e-commerce platform. It was able to scale to extreme peak loads efficiently without any downtime during the busy holiday shopping season. For example, the service that maintains shopping cart (Shopping Cart Service) served tens of millions requests that resulted in well over 3 million checkouts in a single day and the service that manages session state handled hundreds of thousands of concurrently active sessions.

The main contribution of this work for the research community is the evaluation of how different techniques can be combined to provide a single highly-available system. It demonstrates that an eventually-consistent storage system can be used in production with demanding applications. It also provides insight into the tuning of these techniques to meet the requirements of production systems with very strict performance demands.

The paper is structured as follows. Section 2 presents the background and Section 3 presents the related work. Section 4 presents the system design and Section 5 describes the implementation. Section 6 details the experiences and insights gained by running Dynamo in production and Section 7 concludes the paper. There are a number of places in this paper where additional information may have been appropriate but where protecting Amazon's business interests require us to reduce some level of detail. For this reason, the intra- and inter-datacenter latencies in section 6, the absolute request rates in section 6.2 and outage lengths and workloads in section 6.3 are provided through aggregate measures instead of absolute details.

# 2. BACKGROUND

Amazon's e-commerce platform is composed of hundreds of services that work in concert to deliver functionality ranging from recommendations to order fulfillment to fraud detection. Each service is exposed through a well defined interface and is accessible over the network. These services are hosted in an infrastructure that consists of tens of thousands of servers located across many data centers world-wide. Some of these services are stateless (i.e., services which aggregate responses from other services) and some are stateful (i.e., a service that generates its response by executing business logic on its state stored in persistent store).

Traditionally production systems store their state in relational databases. For many of the more common usage patterns of state persistence, however, a relational database is a solution that is far from ideal. Most of these services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by an RDBMS. This excess functionality requires expensive hardware and highly skilled personnel for its operation, making it a very inefficient solution. In addition, the available replication technologies are limited and typically choose consistency over availability. Although many advances have been made in the recent years, it is still not easy to scale-out databases or use smart partitioning schemes for load balancing.

This paper describes Dynamo, a highly available data storage technology that addresses the needs of these important classes of services. Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates. Each service that uses Dynamo runs its own Dynamo instances.

## 2.1 System Assumptions and Requirements

The storage system for this class of services has the following requirements:

*Query Model*: simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema. Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

*ACID Properties:* ACID (*Atomicity, Consistency, Isolation, Durability*) is a set of properties that guarantee that database transactions are processed reliably. In the context of databases, a single logical operation on the data is called a transaction. Experience at Amazon has shown that data stores that provide ACID guarantees tend to have poor availability. This has been widely acknowledged by both the industry and academia [5]. Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability. Dynamo does not provide any isolation guarantees and permits only single key updates.

*Efficiency*: The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the $99.9^{th}$ percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

Other Assumptions: Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts. We will discuss the scalability limitations of Dynamo and possible scalability related extensions in later sections.

## 2.2 Service Level Agreements (SLA)

To guarantee that the application can deliver its functionality in a bounded time, each and every dependency in the platform needs to deliver its functionality with even tighter bounds. Clients and services engage in a Service Level Agreement (SLA), a formally negotiated contract where a client and a service agree on several system-related characteristics, which most prominently include the client's expected request rate distribution for a particular API and the expected service latency under those conditions. An example of a simple SLA is a service guaranteeing that it will

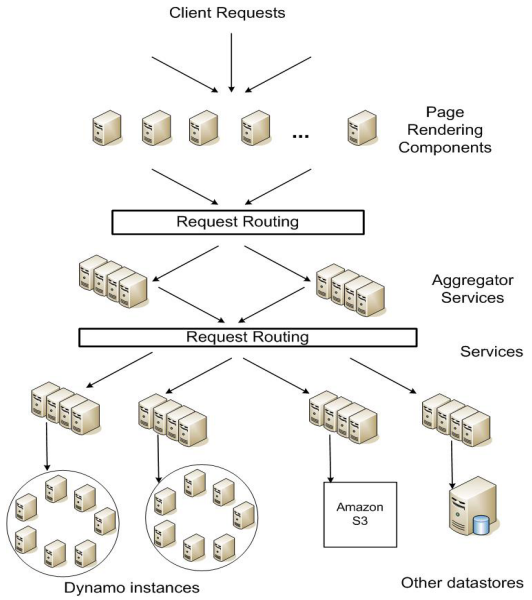



**Figure 1: Service-oriented architecture of Amazon's platform**

provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second.

In Amazon's decentralized service oriented infrastructure, SLAs play an important role. For example a page request to one of the e-commerce sites typically requires the rendering engine to construct its response by sending requests to over 150 services. These services often have multiple dependencies, which frequently are other services, and as such it is not uncommon for the call graph of an application to have more than one level. To ensure that the page rendering engine can maintain a clear bound on page delivery each service within the call chain must obey its performance contract.

Figure 1 shows an abstract view of the architecture of Amazon's platform, where dynamic web content is generated by page rendering components which in turn query many other services. A service can use different data stores to manage its state and these data stores are only accessible within its service boundaries. Some services act as aggregators by using several other services to produce a composite response. Typically, the aggregator services are stateless, although they use extensive caching.

A common approach in the industry for forming a performance oriented SLA is to describe it using average, median and expected variance. At Amazon we have found that these metrics are not good enough if the goal is to build a system where **all** customers have a good experience, rather than just the majority. For example if extensive personalization techniques are used then customers with longer histories require more processing which impacts performance at the high-end of the distribution. An SLA stated in terms of mean or median response times will not address the performance of this important customer segment. To address this issue, at Amazon, SLAs are expressed and measured at the 99.9$^{th}$ percentile of the distribution. The choice for 99.9% over an even higher percentile has been made based on a cost-benefit analysis which demonstrated a significant increase in cost to improve performance that much. Experiences with Amazon's production systems have shown that this approach provides a better overall experience compared to those systems that meet SLAs defined based on the mean or median.

In this paper there are many references to this 99.9$^{th}$ percentile of distributions, which reflects Amazon engineers' relentless focus on performance from the perspective of the customers' experience. Many papers report on averages, so these are included where it makes sense for comparison purposes. Nevertheless, Amazon's engineering and optimization efforts are not focused on averages. Several techniques, such as the load balanced selection of write coordinators, are purely targeted at controlling performance at the 99.9$^{th}$ percentile.

Storage systems often play an important role in establishing a service's SLA, especially if the business logic is relatively lightweight, as is the case for many Amazon services. State management then becomes the main component of a service's SLA. One of the main design considerations for Dynamo is to give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.

## 2.3 Design Considerations

Data replication algorithms used in commercial systems traditionally perform synchronous replica coordination in order to provide a strongly consistent data access interface. To achieve this level of consistency, these algorithms are forced to tradeoff the availability of the data under certain failure scenarios. For instance, rather than dealing with the uncertainty of the correctness of an answer, the data is made unavailable until it is absolutely certain that it is correct. From the very early replicated database works, it is well known that when dealing with the possibility of network failures, strong consistency and high data availability cannot be achieved simultaneously [2, 11]. As such systems and applications need to be aware which properties can be achieved under which conditions.

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where changes are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to conflicting changes which must be detected and resolved. This process of conflict resolution introduces two problems: when to resolve them and who resolves them. Dynamo is designed to be an eventually consistent data store; that is all updates reach all replicas eventually.

An important design consideration is to decide *when* to perform the process of resolving update conflicts, i.e., whether conflicts should be resolved during reads or writes. Many traditional data stores execute conflict resolution during writes and keep the read complexity simple [7]. In such systems, writes may be rejected if the data store cannot reach all (or a majority of) the replicas at a given time. On the other hand, Dynamo targets the design space of an "always writeable" data store (i.e., a data store that is highly available for writes). For a number of Amazon services, rejecting customer updates could result in a poor customer experience. For instance, the shopping cart service must allow customers to add and remove items from their shopping cart even amidst network and server failures. This requirement forces us to push the complexity of conflict resolution to the reads in order to ensure that writes are never rejected.

The next design choice is *who* performs the process of conflict resolution. This can be done by the data store or the application. If conflict resolution is done by the data store, its choices are rather limited. In such cases, the data store can only use simple policies, such as "last write wins" [22], to resolve conflicting updates. On the other hand, since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client's experience. For instance, the application that maintains customer shopping carts can choose to "merge" the conflicting versions and return a single unified shopping cart. Despite this flexibility, some application developers may not want to write their own conflict resolution mechanisms and choose to push it down to the data store, which in turn chooses a simple policy such as "last write wins".

Other key principles embraced in the design are:

*Incremental scalability*: Dynamo should be able to scale out one storage host (henceforth, referred to as "*node"*) at a time, with minimal impact on both operators of the system and the system itself.

*Symmetry*: Every node in Dynamo should have the same set of responsibilities as its peers; there should be no distinguished node or nodes that take special roles or extra set of responsibilities. In our experience, symmetry simplifies the process of system provisioning and maintenance.

*Decentralization*: An extension of symmetry, the design should favor decentralized peer-to-peer techniques over centralized control. In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible. This leads to a simpler, more scalable, and more available system.

*Heterogeneity*: The system needs to be able to exploit heterogeneity in the infrastructure it runs on. e.g. the work distribution must be proportional to the capabilities of the individual servers. This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once.

# 3. RELATED WORK

## 3.1 Peer to Peer Systems

There are several peer-to-peer (P2P) systems that have looked at the problem of data storage and distribution. The first generation of P2P systems, such as Freenet and Gnutella[1], were predominantly used as file sharing systems. These were examples of unstructured P2P networks where the overlay links between peers were established arbitrarily. In these networks, a search query is usually flooded through the network to find as many peers as possible that share the data. P2P systems evolved to the next generation into what is widely known as structured P2P networks. These networks employ a globally consistent protocol to ensure that any node can efficiently route a search query to some peer that has the desired data. Systems like Pastry [16] and Chord [20] use routing mechanisms to ensure that queries can be answered within a bounded number of hops. To reduce the additional latency introduced by multi-hop routing, some P2P systems (e.g., [14]) employ O(1) routing where each peer maintains enough routing information locally so that it can route requests (to access a data item) to the appropriate peer within a constant number of hops.

[1] http://freenetproject.org/, http://www.gnutella.org

Various storage systems, such as Oceanstore [9] and PAST [17] were built on top of these routing overlays. Oceanstore provides a global, transactional, persistent storage service that supports serialized updates on widely replicated data. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, it uses an update model based on conflict resolution. Conflict resolution was introduced in [21] to reduce the number of transaction aborts. Oceanstore resolves conflicts by processing a series of updates, choosing a total order among them, and then applying them atomically in that order. It is built for an environment where the data is replicated on an untrusted infrastructure. By comparison, PAST provides a simple abstraction layer on top of Pastry for persistent and immutable objects. It assumes that the application can build the necessary storage semantics (such as mutable files) on top of it.

## 3.2 Distributed File Systems and Databases

Distributing data for performance, availability and durability has been widely studied in the file system and database systems community. Compared to P2P storage systems that only support flat namespaces, distributed file systems typically support hierarchical namespaces. Systems like Ficus [15] and Coda [19] replicate files for high availability at the expense of consistency. Update conflicts are typically managed using specialized conflict resolution procedures. The Farsite system [1] is a distributed file system that does not use any centralized server like NFS. Farsite achieves high availability and scalability using replication. The Google File System [6] is another distributed file system built for hosting the state of Google's internal applications. GFS uses a simple design with a single master server for hosting the entire metadata and where the data is split into chunks and stored in chunkservers. Bayou is a distributed relational database system that allows disconnected operations and provides eventual data consistency [21].

Among these systems, Bayou, Coda and Ficus allow disconnected operations and are resilient to issues such as network partitions and outages. These systems differ on their conflict resolution procedures. For instance, Coda and Ficus perform system level conflict resolution and Bayou allows application level resolution. All of them, however, guarantee eventual consistency. Similar to these systems, Dynamo allows read and write operations to continue even during network partitions and resolves updated conflicts using different conflict resolution mechanisms. Distributed block storage systems like FAB [18] split large size objects into smaller blocks and stores each block in a highly available manner. In comparison to these systems, a key-value store is more suitable in this case because: (a) it is intended to store relatively small objects (size < 1M) and (b) key-value stores are easier to configure on a per-application basis. Antiquity is a wide-area distributed storage system designed to handle multiple server failures [23]. It uses a secure log to preserve data integrity, replicates each log on multiple servers for durability, and uses Byzantine fault tolerance protocols to ensure data consistency. In contrast to Antiquity, Dynamo does not focus on the problem of data integrity and security and is built for a trusted environment. Bigtable is a distributed storage system for managing structured data. It maintains a sparse, multi-dimensional sorted map and allows applications to access their data using multiple attributes [2]. Compared to Bigtable, Dynamo targets applications that require only key/value access with primary focus on high availability where updates are not rejected even in the wake of network partitions or server failures.

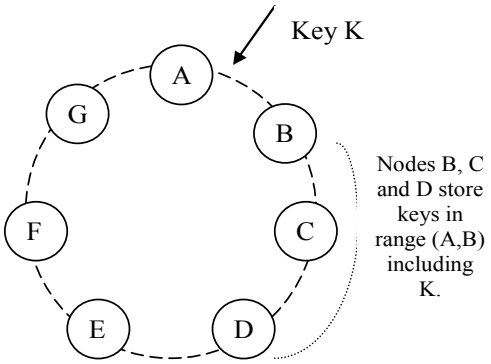



**Figure 2: Partitioning and replication of keys in Dynamo ring.**

Traditional replicated relational database systems focus on the problem of guaranteeing strong consistency to replicated data. Although strong consistency provides the application writer a convenient programming model, these systems are limited in scalability and availability [7]. These systems are not capable of handling network partitions because they typically provide strong consistency guarantees.

## 3.3 Discussion

Dynamo differs from the aforementioned decentralized storage systems in terms of its target requirements. First, Dynamo is targeted mainly at applications that need an "always writeable" data store where no updates are rejected due to failures or concurrent writes. This is a crucial requirement for many Amazon applications. Second, as noted earlier, Dynamo is built for an infrastructure within a single administrative domain where all nodes are assumed to be trusted. Third, applications that use Dynamo do not require support for hierarchical namespaces (a norm in many file systems) or complex relational schema (supported by traditional databases). Fourth, Dynamo is built for latency sensitive applications that require at least 99.9% of read and write operations to be performed within a few hundred milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple nodes (which is the typical design adopted by several distributed hash table systems such as Chord and Pastry). This is because multi-hop routing increases variability in response times, thereby increasing the latency at higher percentiles. Dynamo can be characterized as a zero-hop DHT, where each node maintains enough routing information locally to route a request to the appropriate node directly.

# 4. SYSTEM ARCHITECTURE

The architecture of a storage system that needs to operate in a production setting is complex. In addition to the actual data persistence component, the system needs to have scalable and robust solutions for load balancing, membership and failure detection, failure recovery, replica synchronization, overload handling, state transfer, concurrency and job scheduling, request marshalling, request routing, system monitoring and alarming, and configuration management. Describing the details of each of the solutions is not possible, so this paper focuses on the core distributed systems techniques used in Dynamo: partitioning, replication, versioning, membership, failure handling and scaling.

**Table 1: Summary of techniques used in *Dynamo* and their advantages.**

| Problem | Technique | Advantage |
|---|---|---|
| Partitioning | Consistent Hashing | Incremental Scalability |
| High Availability for writes | Vector clocks with reconciliation during reads | Version size is decoupled from update rates. |
| Handling temporary failures | Sloppy Quorum and hinted handoff | Provides high availability and durability guarantee when some of the replicas are not available. |
| Recovering from permanent failures | Anti-entropy using Merkle trees | Synchronizes divergent replicas in the background. |
| Membership and failure detection | Gossip-based membership protocol and failure detection. | Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information. |

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

## 4.1 System Interface

Dynamo stores objects associated with a key through a simple interface; it exposes two operations: get() and put(). The get(*key*) operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a *context*. The put(*key, context, object*) operation determines where the replicas of the *object* should be placed based on the associated *key*, and writes the replicas to disk. The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

Dynamo treats both the key and the object supplied by the caller as an opaque array of bytes. It applies a MD5 hash on the key to generate a 128-bit identifier, which is used to determine the storage nodes that are responsible for serving the key.

## 4.2 Partitioning Algorithm

One of the key design requirements for Dynamo is that it must scale incrementally. This requires a mechanism to dynamically partition the data over the set of nodes (i.e., storage hosts) in the system. Dynamo's partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts. In consistent hashing [10], the output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value). Each node in the system is assigned a random value within this space which represents its "position" on the ring. Each data item identified by a key is assigned to a node by hashing the data item's key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the item's position.

Thus, each node becomes responsible for the region in the ring between it and its predecessor node on the ring. The principle advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors and other nodes remain unaffected.

The basic consistent hashing algorithm presents some challenges. First, the random position assignment of each node on the ring leads to non-uniform data and load distribution. Second, the basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address these issues, Dynamo uses a variant of consistent hashing (similar to the one used in [10, 20]): instead of mapping a node to a single point in the circle, each node gets assigned to multiple points in the ring. To this end, Dynamo uses the concept of "virtual nodes". A virtual node looks like a single node in the system, but each node can be responsible for more than one virtual node. Effectively, when a new node is added to the system, it is assigned multiple positions (henceforth, "tokens") in the ring. The process of fine-tuning Dynamo's partitioning scheme is discussed in Section 6.

Using virtual nodes has the following advantages:

- If a node becomes unavailable (due to failures or routine maintenance), the load handled by this node is evenly dispersed across the remaining available nodes.
- When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
- The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

## 4.3 Replication

To achieve high availability and durability, Dynamo replicates its data on multiple hosts. Each data item is replicated at N hosts, where N is a parameter configured *"per-instance"*. Each key, *k*, is assigned to a coordinator node (described in the previous section). The coordinator is in charge of the replication of the data items that fall within its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the N-1 clockwise successor nodes in the ring. This results in a system where each node is responsible for the region of the ring between it and its $N^{th}$ predecessor. In Figure 2, node B replicates the key *k* at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges (A, B], (B, C], and (C, D].

The list of nodes that is responsible for storing a particular key is called the *preference list*. The system is designed, as will be explained in Section 4.8, so that every node in the system can determine which nodes should be in this list for any particular key. To account for node failures, preference list contains more than N nodes. Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.

## 4.4 Data Versioning

Dynamo provides eventual consistency, which allows for updates to be propagated to all replicas asynchronously. A put() call may return to its caller before the update has been applied at all the replicas, which can result in scenarios where a subsequent get() operation may return an object that does not have the latest updates.. If there are no failures then there is a bound on the update propagation times. However, under certain failure scenarios (e.g., server outages or network partitions), updates may not arrive at all replicas for an extended period of time.

There is a category of applications in Amazon's platform that can tolerate such inconsistencies and can be constructed to operate under these conditions. For example, the shopping cart application requires that an "*Add to Cart"* operation can never be forgotten or rejected. If the most recent state of the cart is unavailable, and a user makes changes to an older version of the cart, that change is still meaningful and should be preserved. But at the same time it shouldn't supersede the currently unavailable state of the cart, which itself may contain changes that should be preserved. Note that both "*add to cart*" and "*delete item from cart*" operations are translated into put requests to Dynamo. When a customer wants to add an item to (or remove from) a shopping cart and the latest version is not available, the item is added to (or removed from) the older version and the divergent versions are reconciled later.

In order to provide this kind of guarantee, Dynamo treats the result of each modification as a new and immutable version of the data. It allows for multiple versions of an object to be present in the system at the same time. Most of the time, new versions subsume the previous version(s), and the system itself can determine the authoritative version (syntactic reconciliation). However, version branching may happen, in the presence of failures combined with concurrent updates, resulting in conflicting versions of an object. In these cases, the system cannot reconcile the multiple versions of the same object and the client must perform the reconciliation in order to *collapse* multiple branches of data evolution back into one (semantic reconciliation). A typical example of a collapse operation is "merging" different versions of a customer's shopping cart. Using this reconciliation mechanism, an "add to cart" operation is never lost. However, deleted items can resurface.

It is important to understand that certain failure modes can potentially result in the system having not just two but several versions of the same data. Updates in the presence of network partitions and node failures can potentially result in an object having distinct version sub-histories, which the system will need to reconcile in the future. This requires us to design applications that explicitly acknowledge the possibility of multiple versions of the same data (in order to never lose any updates).

Dynamo uses vector clocks [12] in order to capture causality between different versions of the same object. A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object. One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examine their vector clocks. If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.

In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if

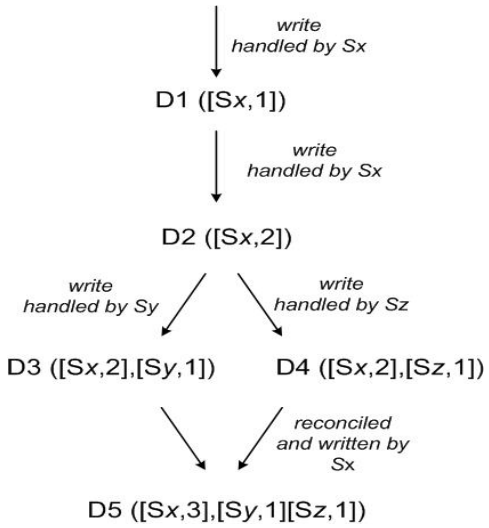



**Figure 3: Version evolution of an object over time.**

Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.

To illustrate the use of vector clocks, let us consider the example shown in Figure 3. A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock [(Sx, 1)]. The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock [(Sx, 2)]. D2 *descends* from D1 and therefore over-writes D1, however there may be replicas of D1 lingering at nodes that have not yet seen D2. Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write. The system now has D4 (descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)]. A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected. A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other. Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)]. If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock: [(Sx, 3), (Sy, 1), (Sz, 1)].

A possible issue with vector clocks is that the size of vector clocks may grow if many servers coordinate the writes to an object. In practice, this is not likely because the writes are usually handled by one of the top N nodes in the preference list. In case of network partitions or multiple server failures, write requests may be handled by nodes that are not in the top N nodes in the preference list causing the size of vector clock to grow. In these scenarios, it is desirable to limit the size of vector clock. To this end, Dynamo employs the following clock truncation scheme: Along with each (node, counter) pair, Dynamo stores a timestamp that indicates the last time the node updated the data item. When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock. Clearly, this truncation scheme can lead to inefficiencies in reconciliation as the descendant relationships cannot be derived accurately. However, this problem has not surfaced in production and therefore this issue has not been thoroughly investigated.

## 4.5 Execution of get () and put () operations

Any storage node in Dynamo is eligible to receive client get and put operations for any key. In this section, for sake of simplicity, we describe how these operations are performed in a failure-free environment and in the subsequent section we describe how read and write operations are executed during failures.

Both get and put operations are invoked using Amazon's infrastructure-specific request processing framework over HTTP. There are two strategies that a client can use to select a node: (1) route its request through a generic load balancer that will select a node based on load information, or (2) use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. The advantage of the first approach is that the client does not have to link any code specific to Dynamo in its application, whereas the second strategy can achieve lower latency because it skips a potential forwarding step.

A node handling a read or write operation is known as the *coordinator*. Typically, this is the first among the top N nodes in the preference list. If the requests are received through a load balancer, requests to access a key may be routed to any random node in the ring. In this scenario, the node that receives the request will not coordinate it if the node is not in the top N of the requested key's preference list. Instead, that node will forward the request to the first among the top N nodes in the preference list.

Read and write operations involve the first N healthy nodes in the preference list, skipping over those that are down or inaccessible. When all nodes are healthy, the top N nodes in a key's preference list are accessed. When there are node failures or network partitions, nodes that are lower ranked in the preference list are accessed.

To maintain consistency among its replicas, Dynamo uses a consistency protocol similar to those used in quorum systems. This protocol has two key configurable values: R and W. R is the minimum number of nodes that must participate in a successful read operation. W is the minimum number of nodes that must participate in a successful write operation. Setting R and W such that R + W > N yields a quorum-like system. In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

Upon receiving a put() request for a key, the coordinator generates the vector clock for the new version and writes the new version locally. The coordinator then sends the new version (along with

the new vector clock) to the N highest-ranked reachable nodes. If at least W-1 nodes respond then the write is considered successful.

Similarly, for a get() request, the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key, and then waits for R responses before returning the result to the client. If the coordinator ends up gathering multiple versions of the data, it returns all the versions it deems to be causally unrelated. The divergent versions are then reconciled and the reconciled version superseding the current versions is written back.

## 4.6 Handling Failures: Hinted Handoff

If Dynamo used a traditional quorum approach it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this it does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N *healthy* nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring.

Consider the example of Dynamo configuration given in Figure 2 with N=3. In this example, if node A is temporarily down or unreachable during a write operation then a replica that would normally have lived on A will now be sent to node D. This is done to maintain the desired availability and durability guarantees. The replica sent to D will have a hint in its metadata that suggests which node was the intended recipient of the replica (in this case A). Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically. Upon detecting that A has recovered, D will attempt to deliver the replica to A. Once the transfer succeeds, D may delete the object from its local store without decreasing the total number of replicas in the system.

Using hinted handoff, Dynamo ensures that the read and write operations are not failed due to temporary node or network failures. Applications that need the highest level of availability can set W to 1, which ensures that a write is accepted as long as a single node in the system has durably written the key it to its local store. Thus, the write request is only rejected if all nodes in the system are unavailable. However, in practice, most Amazon services in production set a higher W to meet the desired level of durability. A more detailed discussion of configuring N, R and W follows in section 6.

It is imperative that a highly available storage system be capable of handling the failure of an entire data center(s). Data center failures happen due to power outages, cooling failures, network failures, and natural disasters. Dynamo is configured such that each object is replicated across multiple data centers. In essence, the preference list of a key is constructed such that the storage nodes are spread across multiple data centers. These datacenters are connected through high speed network links. This scheme of replicating across multiple datacenters allows us to handle entire data center failures without a data outage.

## 4.7 Handling permanent failures: Replica synchronization

Hinted handoff works best if the system membership churn is low and node failures are transient. There are scenarios under which hinted replicas become unavailable before they can be returned to the original replica node. To handle this and other threats to durability, Dynamo implements an anti-entropy (replica synchronization) protocol to keep the replicas synchronized.

To detect the inconsistencies between replicas faster and to minimize the amount of transferred data, Dynamo uses Merkle trees [13]. A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children. The principal advantage of Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Moreover, Merkle trees help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas. For instance, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the tree are equal and the nodes require no synchronization. If not, it implies that the values of some replicas are different. In such cases, the nodes may exchange the hash values of children and the process continues until it reaches the leaves of the trees, at which point the hosts can identify the keys that are "out of sync". Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads performed during the anti-entropy process.

Dynamo uses Merkle trees for anti-entropy as follows: Each node maintains a separate Merkle tree for each key range (the set of keys covered by a virtual node) it hosts. This allows nodes to compare whether the keys within a key range are up-to-date. In this scheme, two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common. Subsequently, using the tree traversal scheme described above the nodes determine if they have any differences and perform the appropriate synchronization action. The disadvantage with this scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated. This issue is addressed, however, by the refined partitioning scheme described in Section 6.2.

## 4.8 Membership and Failure Detection

### *4.8.1 Ring Membership*

In Amazon's environment node outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. A node outage rarely signifies a permanent departure and therefore should not result in rebalancing of the partition assignment or repair of the unreachable replicas. Similarly, manual error could result in the unintentional startup of new Dynamo nodes. For these reasons, it was deemed appropriate to use an explicit mechanism to initiate the addition and removal of nodes from a Dynamo ring. An administrator uses a command line tool or a browser to connect to a Dynamo node and issue a membership change to join a node to a ring or remove a node from a ring. The node that serves the request writes the membership change and its time of issue to persistent store. The membership changes form a history because nodes can be removed and added back multiple times. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of membership. Each node contacts a peer chosen at random every second and the two nodes efficiently reconcile their persisted membership change histories.

When a node starts for the first time, it chooses its set of tokens (virtual nodes in the consistent hash space) and maps nodes to their respective token sets. The mapping is persisted on disk and

initially contains only the local node and token set. The mappings stored at different Dynamo nodes are reconciled during the same communication exchange that reconciles the membership change histories. Therefore, partitioning and placement information also propagates via the gossip-based protocol and each storage node is aware of the token ranges handled by its peers. This allows each node to forward a key's read/write operations to the right set of nodes directly.

### *4.8.2 External Discovery*

The mechanism described above could temporarily result in a logically partitioned Dynamo ring. For example, the administrator could contact node A to join A to the ring, then contact node B to join B to the ring. In this scenario, nodes A and B would each consider itself a member of the ring, yet neither would be immediately aware of the other. To prevent logical partitions, some Dynamo nodes play the role of seeds. Seeds are nodes that are discovered via an external mechanism and are known to all nodes. Because all nodes eventually reconcile their membership with a seed, logical partitions are highly unlikely. Seeds can be obtained either from static configuration or from a configuration service. Typically seeds are fully functional nodes in the Dynamo ring.

### *4.8.3 Failure Detection*

Failure detection in Dynamo is used to avoid attempts to communicate with unreachable peers during get() and put() operations and when transferring partitions and hinted replicas. For the purpose of avoiding failed attempts at communication, a purely local notion of failure detection is entirely sufficient: node A may consider node B failed if node B does not respond to node A's messages (even if B is responsive to node C's messages). In the presence of a steady rate of client requests generating inter-node communication in the Dynamo ring, a node A quickly discovers that a node B is unresponsive when B fails to respond to a message; Node A then uses alternate nodes to service requests that map to B's partitions; A periodically retries B to check for the latter's recovery. In the absence of client requests to drive traffic between two nodes, neither node really needs to know whether the other is reachable and responsive.

Decentralized failure detection protocols use a simple gossip-style protocol that enable each node in the system to learn about the arrival (or departure) of other nodes. For detailed information on decentralized failure detectors and the parameters affecting their accuracy, the interested reader is referred to [8]. Early designs of Dynamo used a decentralized failure detector to maintain a globally consistent view of failure state. Later it was determined that the explicit node join and leave methods obviates the need for a global view of failure state. This is because nodes are notified of permanent node additions and removals by the explicit node join and leave methods and temporary node failures are detected by the individual nodes when they fail to communicate with others (while forwarding requests).

## 4.9 Adding/Removing Storage Nodes

When a new node (say X) is added into the system, it gets assigned a number of tokens that are randomly scattered on the ring. For every key range that is assigned to node X, there may be a number of nodes (less than or equal to N) that are currently in charge of handling keys that fall within its token range. Due to the allocation of key ranges to X, some existing nodes no longer have to some of their keys and these nodes transfer those keys to X. Let us consider a simple bootstrapping scenario where node X is added to the ring shown in Figure 2 between A and B. When X is added to the system, it is in charge of storing keys in the ranges (F, G], (G, A] and (A, X]. As a consequence, nodes B, C and D no longer have to store the keys in these respective ranges. Therefore, nodes B, C, and D will offer to and upon confirmation from X transfer the appropriate set of keys. When a node is removed from the system, the reallocation of keys happens in a reverse process.

Operational experience has shown that this approach distributes the load of key distribution uniformly across the storage nodes, which is important to meet the latency requirements and to ensure fast bootstrapping. Finally, by adding a confirmation round between the source and the destination, it is made sure that the destination node does not receive any duplicate transfers for a given key range.

# 5. IMPLEMENTATION

In Dynamo, each storage node has three main software components: request coordination, membership and failure detection, and a local persistence engine. All these components are implemented in Java.

Dynamo's local persistence component allows for different storage engines to be plugged in. Engines that are in use are Berkeley Database (BDB) Transactional Data Store[2], BDB Java Edition, MySQL, and an in-memory buffer with persistent backing store. The main reason for designing a pluggable persistence component is to choose the storage engine best suited for an application's access patterns. For instance, BDB can handle objects typically in the order of tens of kilobytes whereas MySQL can handle objects of larger sizes. Applications choose Dynamo's local persistence engine based on their object size distribution. The majority of Dynamo's production instances use BDB Transactional Data Store.

The request coordination component is built on top of an event-driven messaging substrate where the message processing pipeline is split into multiple stages similar to the SEDA architecture [24]. All communications are implemented using Java NIO channels. The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes). Each client request results in the creation of a state machine on the node that received the client request. The state machine contains all the logic for identifying the nodes responsible for a key, sending the requests, waiting for responses, potentially doing retries, processing the replies and packaging the response to the client. Each state machine instance handles exactly one client request. For instance, a read operation implements the following state machine: (i) send read requests to the nodes, (ii) wait for minimum number of required responses, (iii) if too few replies were received within a given time bound, fail the request, (iv) otherwise gather all the data versions and determine the ones to be returned and (v) if versioning is enabled, perform syntactic reconciliation and generate an opaque write context that contains the vector clock that subsumes all the remaining versions. For the sake of brevity the failure handling and retry states are left out.

After the read response has been returned to the caller the state

[2] http://www.oracle.com/database/berkeley-db.html

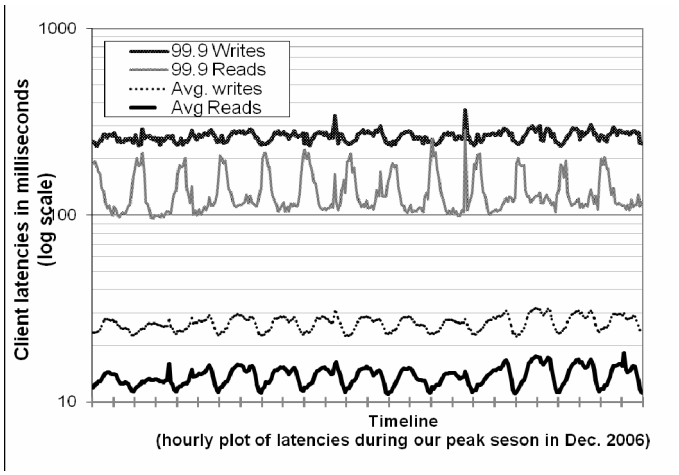



**Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages**

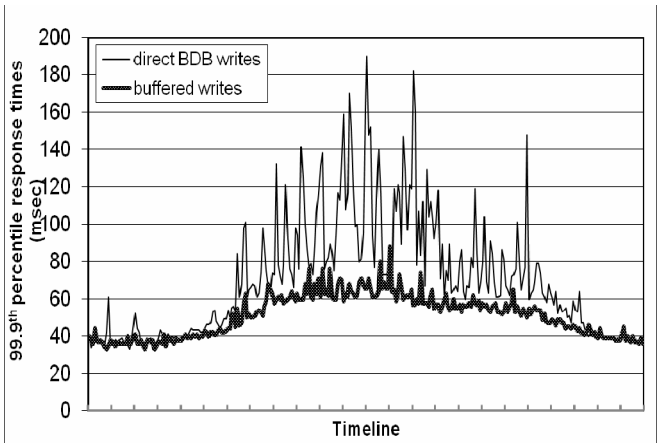



**Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.**

machine waits for a small period of time to receive any outstanding responses. If stale versions were returned in any of the responses, the coordinator updates those nodes with the latest version. This process is called *read repair* because it repairs replicas that have missed a recent update at an opportunistic time and relieves the anti-entropy protocol from having to do it.

As noted earlier, write requests are coordinated by one of the top N nodes in the preference list. Although it is desirable always to have the first node among the top N to coordinate the writes thereby serializing all writes at a single location, this approach has led to uneven load distribution resulting in SLA violations. This is because the request load is not uniformly distributed across objects. To counter this, any of the top N nodes in the preference list is allowed to coordinate the writes. In particular, since each write usually follows a read operation, the coordinator for a write is chosen to be the node that replied fastest to the previous read operation which is stored in the context information of the request. This optimization enables us to pick the node that has the data that was read by the preceding read operation thereby increasing the chances of getting "read-your-writes" consistency. It also reduces variability in the performance of the request handling which improves the performance at the 99.9 percentile.

# 6. EXPERIENCES & LESSONS LEARNED

Dynamo is used by several services with different configurations. These instances differ by their version reconciliation logic, and read/write quorum characteristics. The following are the main patterns in which Dynamo is used:

- *Business logic specific reconciliation:* This is a popular use case for Dynamo. Each data object is replicated across multiple nodes. In case of divergent versions, the client application performs its own reconciliation logic. The shopping cart service discussed earlier is a prime example of this category. Its business logic reconciles objects by merging different versions of a customer's shopping cart.
- *Timestamp based reconciliation:* This case differs from the previous one only in the reconciliation mechanism. In case of divergent versions, Dynamo performs simple timestamp based reconciliation logic of "last write wins"; i.e., the object with the largest physical timestamp value is chosen as the correct version. The service that maintains customer's session information is a good example of a service that uses this mode.
- *High performance read engine:* While Dynamo is built to be an "always writeable" data store, a few services are tuning its quorum characteristics and using it as a high performance read engine. Typically, these services have a high read request rate and only a small number of updates. In this configuration, typically R is set to be 1 and W to be N. For these services, Dynamo provides the ability to partition and replicate their data across multiple nodes thereby offering incremental scalability. Some of these instances function as the authoritative persistence cache for data stored in more heavy weight backing stores. Services that maintain product catalog and promotional items fit in this category.

The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability. For instance, the value of N determines the durability of each object. A typical value of N used by Dynamo's users is 3.

The values of W and R impact object availability, durability and consistency. For instance, if W is set to 1, then the system will never reject a write request as long as there is at least one node in the system that can successfully process a write request. However, low values of W and R can increase the risk of inconsistency as write requests are deemed successful and returned to the clients even if they are not processed by a majority of the replicas. This also introduces a vulnerability window for durability when a write request is successfully returned to the client even though it has been persisted at only a small number of nodes.

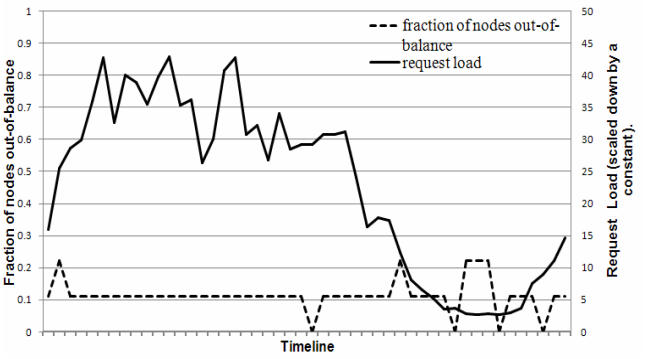



**Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.**

Traditional wisdom holds that durability and availability go hand-in-hand. However, this is not necessarily true here. For instance, the vulnerability window for durability can be decreased by increasing W. This may increase the probability of rejecting requests (thereby decreasing availability) because more storage hosts need to be alive to process a write request.

The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2). These values are chosen to meet the necessary levels of performance, durability, consistency, and availability SLAs.

All the measurements presented in this section were taken on a live system operating with a configuration of (3,2,2) and running a couple hundred nodes with homogenous hardware configurations. As mentioned earlier, each instance of Dynamo contains nodes that are located in multiple datacenters. These datacenters are typically connected through high speed network links. Recall that to generate a successful get (or put) response R (or W) nodes need to respond to the coordinator. Clearly, the network latencies between datacenters affect the response time and the nodes (and their datacenter locations) are chosen such that the applications target SLAs are met.

## 6.1 Balancing Performance and Durability

While Dynamo's principle design goal is to build a highly available data store, performance is an equally important criterion in Amazon's platform. As noted earlier, to provide a consistent customer experience, Amazon's services set their performance targets at higher percentiles (such as the 99.9$^{th}$ or 99.99$^{th}$ percentiles). A typical SLA required of services that use Dynamo is that 99.9% of the read and write requests execute within 300ms.

Since Dynamo is run on standard commodity hardware components that have far less I/O throughput than high-end enterprise servers, providing consistently high performance for read and write operations is a non-trivial task. The involvement of multiple storage nodes in read and write operations makes it even more challenging, since the performance of these operations is limited by the slowest of the R or W replicas. Figure 4 shows the average and 99.9$^{th}$ percentile latencies of Dynamo's read and write operations during a period of 30 days. As seen in the figure, the latencies exhibit a clear diurnal pattern which is a result of the diurnal pattern in the incoming request rate (i.e., there is a significant difference in request rate between the daytime and night). Moreover, the write latencies are higher than read latencies obviously because write operations always results in disk access. Also, the 99.9$^{th}$ percentile latencies are around 200 ms and are an order of magnitude higher than the averages. This is because the 99.9$^{th}$ percentile latencies are affected by several factors such as variability in request load, object sizes, and locality patterns.

While this level of performance is acceptable for a number of services, a few customer-facing services required higher levels of performance. For these services, Dynamo provides the ability to trade-off durability guarantees for performance. In the optimization each storage node maintains an object buffer in its main memory. Each write operation is stored in the buffer and gets periodically written to storage by a *writer thread*. In this scheme, read operations first check if the requested key is present in the buffer. If so, the object is read from the buffer instead of the storage engine.

This optimization has resulted in lowering the 99.9$^{th}$ percentile latency by a factor of 5 during peak traffic even for a very small buffer of a thousand objects (see Figure 5). Also, as seen in the figure, write buffering smoothes out higher percentile latencies. Obviously, this scheme trades durability for performance. In this scheme, a server crash can result in missing writes that were queued up in the buffer. To reduce the durability risk, the write operation is refined to have the coordinator choose one out of the N replicas to perform a "durable write". Since the coordinator waits only for W responses, the performance of the write operation is not affected by the performance of the durable write operation performed by a single replica.

## 6.2 Ensuring Uniform Load distribution

Dynamo uses consistent hashing to partition its key space across its replicas and to ensure uniform load distribution. A uniform key distribution can help us achieve uniform load distribution assuming the access distribution of keys is not highly skewed. In particular, Dynamo's design assumes that even where there is a significant skew in the access distribution there are enough keys in the popular end of the distribution so that the load of handling popular keys can be spread across the nodes uniformly through partitioning. This section discusses the load imbalance seen in Dynamo and the impact of different partitioning strategies on load distribution.

To study the load imbalance and its correlation with request load, the total number of requests received by each node was measured for a period of 24 hours - broken down into intervals of 30 minutes. In a given time window, a node is considered to be "in-balance", if the node's request load deviates from the average load by a value a less than a certain threshold (here 15%). Otherwise the node was deemed "out-of-balance". Figure 6 presents the fraction of nodes that are "out-of-balance" (henceforth, "imbalance ratio") during this time period. For reference, the corresponding request load received by the entire system during this time period is also plotted. As seen in the figure, the imbalance ratio decreases with increasing load. For instance, during low loads the imbalance ratio is as high as 20% and during high loads it is close to 10%. Intuitively, this can be explained by the fact that under high loads, a large number of popular keys are accessed and due to uniform distribution of keys the load is evenly distributed. However, during low loads (where load is 1/8$^{th}$

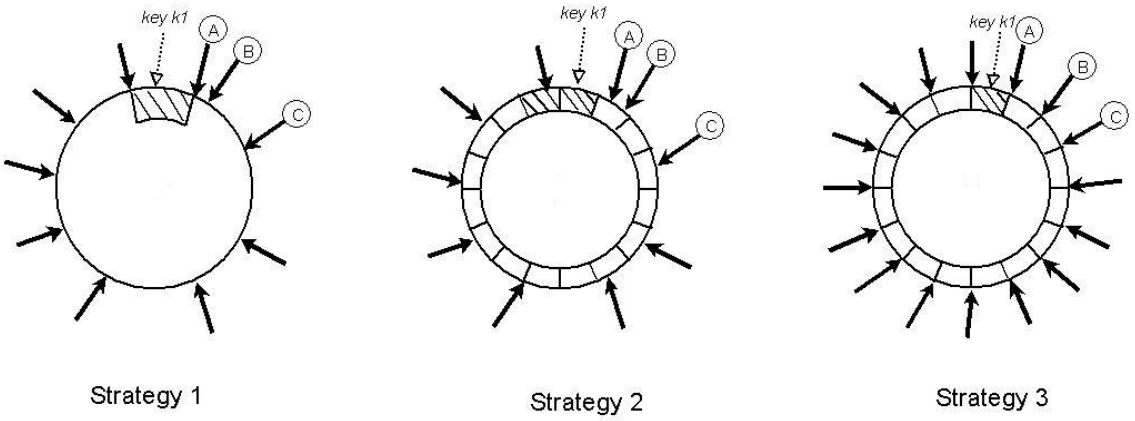



**Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.**

of the measured peak load), fewer popular keys are accessed, resulting in a higher load imbalance.

This section discusses how Dynamo's partitioning scheme has evolved over time and its implications on load distribution.

*Strategy 1: T random tokens per node and partition by token value*: This was the initial strategy deployed in production (and described in Section 4.2). In this scheme, each node is assigned T tokens (chosen uniformly at random from the hash space). The tokens of all nodes are ordered according to their values in the hash space. Every two consecutive tokens define a range. The last token and the first token form a range that "wraps" around from the highest value to the lowest value in the hash space. Because the tokens are chosen randomly, the ranges vary in size. As nodes join and leave the system, the token set changes and consequently the ranges change. Note that the space needed to maintain the membership at each node increases linearly with the number of nodes in the system.

While using this strategy, the following problems were encountered. First, when a new node joins the system, it needs to "steal" its key ranges from other nodes. However, the nodes handing the key ranges off to the new node have to scan their local persistence store to retrieve the appropriate set of data items. Note that performing such a scan operation on a production node is tricky as scans are highly resource intensive operations and they need to be executed in the background without affecting the customer performance. This requires us to run the bootstrapping task at the lowest priority. However, this significantly slows the bootstrapping process and during busy shopping season, when the nodes are handling millions of requests a day, the bootstrapping has taken almost a day to complete. Second, when a node joins/leaves the system, the key ranges handled by many nodes change and the Merkle trees for the new ranges need to be recalculated, which is a non-trivial operation to perform on a production system. Finally, there was no easy way to take a snapshot of the entire key space due to the randomness in key ranges, and this made the process of archival complicated. In this scheme, archiving the entire key space requires us to retrieve the keys from each node separately, which is highly inefficient.

The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement. To this end, following strategies were evaluated:

*Strategy 2: T random tokens per node and equal sized partitions:* In this strategy, the hash space is divided into Q equally sized partitions/ranges and each node is assigned T random tokens. Q is usually set such that Q >> N and Q >> S*T, where S is the number of nodes in the system. In this strategy, the tokens are only used to build the function that maps values in the hash space to the ordered lists of nodes and not to decide the partitioning. A partition is placed on the first N unique nodes that are encountered while walking the consistent hashing ring clockwise from the end of the partition. Figure 7 illustrates this strategy for N=3. In this example, nodes A, B, C are encountered while walking the ring from the end of the partition that contains key k1. The primary advantages of this strategy are: (i) decoupling of partitioning and partition placement, and (ii) enabling the possibility of changing the placement scheme at runtime.

*Strategy 3: Q/S tokens per node, equal-sized partitions:* Similar to strategy 2, this strategy divides the hash space into Q equally sized partitions and the placement of partition is decoupled from the partitioning scheme. Moreover, each node is assigned Q/S tokens where S is the number of nodes in the system. When a node leaves the system, its tokens are randomly distributed to the remaining nodes such that these properties are preserved. Similarly, when a node joins the system it "steals" tokens from nodes in the system in a way that preserves these properties.

The efficiency of these three strategies is evaluated for a system with S=30 and N=3. However, comparing these different strategies in a fair manner is hard as different strategies have different configurations to tune their efficiency. For instance, the load distribution property of strategy 1 depends on the number of tokens (i.e., T) while strategy 3 depends on the number of partitions (i.e., Q). One fair way to compare these strategies is to

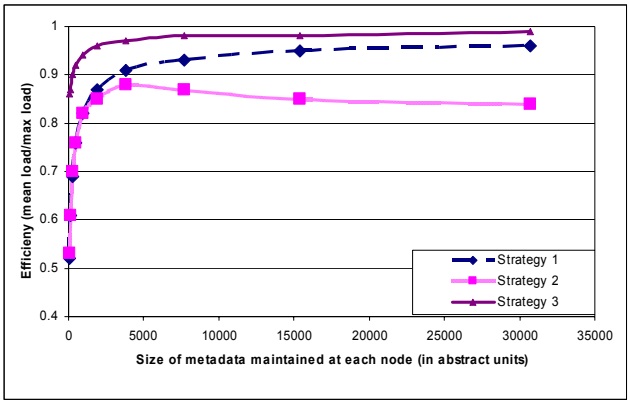



**Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.**

evaluate the skew in their load distribution while all strategies use the same amount of space to maintain their membership information. For instance, in strategy 1 each node needs to maintain the token positions of all the nodes in the ring and in strategy 3 each node needs to maintain the information regarding the partitions assigned to each node.

In our next experiment, these strategies were evaluated by varying the relevant parameters (T and Q). The load balancing efficiency of each strategy was measured for different sizes of membership information that needs to be maintained at each node, where *Load balancing efficiency* is defined as the ratio of average number of requests served by each node to the maximum number of requests served by the hottest node.

The results are given in Figure 8. As seen in the figure, strategy 3 achieves the best load balancing efficiency and strategy 2 has the worst load balancing efficiency. For a brief time, Strategy 2 served as an interim setup during the process of migrating Dynamo instances from using Strategy 1 to Strategy 3. Compared to Strategy 1, Strategy 3 achieves better efficiency and reduces the size of membership information maintained at each node by three orders of magnitude. While storage is not a major issue the nodes gossip the membership information periodically and as such it is desirable to keep this information as compact as possible. In addition to this, strategy 3 is advantageous and simpler to deploy for the following reasons: (i) *Faster bootstrapping/recovery:* Since partition ranges are fixed, they can be stored in separate files, meaning a partition can be relocated as a unit by simply transferring the file (avoiding random accesses needed to locate specific items). This simplifies the process of bootstrapping and recovery. (ii) *Ease of archival*: Periodical archiving of the dataset is a mandatory requirement for most of Amazon storage services. Archiving the entire dataset stored by Dynamo is simpler in strategy 3 because the partition files can be archived separately. By contrast, in Strategy 1, the tokens are chosen randomly and, archiving the data stored in Dynamo requires retrieving the keys from individual nodes separately and is usually inefficient and slow. The disadvantage of strategy 3 is that changing the node membership requires coordination in order to preserve the properties required of the assignment.

## 6.3 Divergent Versions: When and How Many?

As noted earlier, Dynamo is designed to tradeoff consistency for availability. To understand the precise impact of different failures on consistency, detailed data is required on multiple factors: outage length, type of failure, component reliability, workload etc. Presenting these numbers in detail is outside of the scope of this paper. However, this section discusses a good summary metric: the number of divergent versions seen by the application in a live production environment.

Divergent versions of a data item arise in two scenarios. The first is when the system is facing failure scenarios such as node failures, data center failures, and network partitions. The second is when the system is handling a large number of concurrent writers to a single data item and multiple nodes end up coordinating the updates concurrently. From both a usability and efficiency perspective, it is preferred to keep the number of divergent versions at any given time as low as possible. If the versions cannot be syntactically reconciled based on vector clocks alone, they have to be passed to the business logic for semantic reconciliation. Semantic reconciliation introduces additional load on services, so it is desirable to minimize the need for it.

In our next experiment, the number of versions returned to the shopping cart service was profiled for a period of 24 hours. During this period, 99.94% of requests saw exactly one version; 0.00057% of requests saw 2 versions; 0.00047% of requests saw 3 versions and 0.00009% of requests saw 4 versions. This shows that divergent versions are created rarely.

Experience shows that the increase in the number of divergent versions is contributed not by failures but due to the increase in number of concurrent writers. The increase in the number of concurrent writes is usually triggered by busy robots (automated client programs) and rarely by humans. This issue is not discussed in detail due to the sensitive nature of the story.

## 6.4 Client-driven or Server-driven Coordination

As mentioned in Section 5, Dynamo has a request coordination component that uses a state machine to handle incoming requests. Client requests are uniformly assigned to nodes in the ring by a load balancer. Any Dynamo node can act as a coordinator for a read request. Write requests on the other hand will be coordinated by a node in the key's current preference list. This restriction is due to the fact that these preferred nodes have the added responsibility of creating a new version stamp that causally subsumes the version that has been updated by the write request. Note that if Dynamo's versioning scheme is based on physical timestamps, any node can coordinate a write request.

An alternative approach to request coordination is to move the state machine to the client nodes. In this scheme client applications use a library to perform request coordination locally. A client periodically picks a random Dynamo node and downloads its current view of Dynamo membership state. Using this information the client can determine which set of nodes form the preference list for any given key. Read requests can be coordinated at the client node thereby avoiding the extra network hop that is incurred if the request were assigned to a random Dynamo node by the load balancer. Writes will either be forwarded to a node in the key's preference list or can be

**Table 2: Performance of client-driven and server-driven coordination approaches.**

| | 99.9th percentile read latency (ms) | 99.9th percentile write latency (ms) | Average read latency (ms) | Average write latency (ms) |
|---|---|---|---|---|
| Server-driven | 68.9 | 68.5 | 3.9 | 4.02 |
| Client-driven | 30.4 | 30.4 | 1.55 | 1.9 |

coordinated locally if Dynamo is using timestamps based versioning.

An important advantage of the client-driven coordination approach is that a load balancer is no longer required to uniformly distribute client load. Fair load distribution is implicitly guaranteed by the near uniform assignment of keys to the storage nodes. Obviously, the efficiency of this scheme is dependent on how fresh the membership information is at the client. Currently clients poll a random Dynamo node every 10 seconds for membership updates. A pull based approach was chosen over a push based one as the former scales better with large number of clients and requires very little state to be maintained at servers regarding clients. However, in the worst case the client can be exposed to stale membership for duration of 10 seconds. In case, if the client detects its membership table is stale (for instance, when some members are unreachable), it will immediately refresh its membership information.

Table 2 shows the latency improvements at the 99.9$^{th}$ percentile and averages that were observed for a period of 24 hours using client-driven coordination compared to the server-driven approach. As seen in the table, the client-driven coordination approach reduces the latencies by at least 30 milliseconds for 99.9$^{th}$ percentile latencies and decreases the average by 3 to 4 milliseconds. The latency improvement is because the client-driven approach eliminates the overhead of the load balancer and the extra network hop that may be incurred when a request is assigned to a random node. As seen in the table, average latencies tend to be significantly lower than latencies at the 99.9th percentile. This is because Dynamo's storage engine caches and write buffer have good hit ratios. Moreover, since the load balancers and network introduce additional variability to the response time, the gain in response time is higher for the 99.9$^{th}$ percentile than the average.

## 6.5 Balancing background vs. foreground tasks

Each node performs different kinds of background tasks for replica synchronization and data handoff (either due to hinting or adding/removing nodes) in addition to its normal foreground put/get operations. In early production settings, these background tasks triggered the problem of resource contention and affected the performance of the regular put and get operations. Hence, it became necessary to ensure that background tasks ran only when the regular critical operations are not affected significantly. To this end, the background tasks were integrated with an admission control mechanism. Each of the background tasks uses this controller to reserve runtime slices of the resource (e.g. database), shared across all background tasks. A feedback mechanism based on the monitored performance of the foreground tasks is employed to change the number of slices that are available to the background tasks.

The admission controller constantly monitors the behavior of resource accesses while executing a "foreground" put/get operation. Monitored aspects include latencies for disk operations, failed database accesses due to lock-contention and transaction timeouts, and request queue wait times. This information is used to check whether the percentiles of latencies (or failures) in a given trailing time window are close to a desired threshold. For example, the background controller checks to see how close the 99$^{th}$ percentile database read latency (over the last 60 seconds) is to a preset threshold (say 50ms). The controller uses such comparisons to assess the resource availability for the foreground operations. Subsequently, it decides on how many time slices will be available to background tasks, thereby using the feedback loop to limit the intrusiveness of the background activities. Note that a similar problem of managing background tasks has been studied in [4].

## 6.6 Discussion

This section summarizes some of the experiences gained during the process of implementation and maintenance of Dynamo. Many Amazon internal services have used Dynamo for the past two years and it has provided significant levels of availability to its applications. In particular, applications have received successful responses (without timing out) for 99.9995% of its requests and no data loss event has occurred to date.

Moreover, the primary advantage of Dynamo is that it provides the necessary knobs using the three parameters of (N,R,W) to tune their instance based on their needs.. Unlike popular commercial data stores, Dynamo exposes data consistency and reconciliation logic issues to the developers. At the outset, one may expect the application logic to become more complex. However, historically, Amazon's platform is built for high availability and many applications are designed to handle different failure modes and inconsistencies that may arise. Hence, porting such applications to use Dynamo was a relatively simple task. For new applications that want to use Dynamo, some analysis is required during the initial stages of the development to pick the right conflict resolution mechanisms that meet the business case appropriately. Finally, Dynamo adopts a full membership model where each node is aware of the data hosted by its peers. To do this, each node actively gossips the full routing table with other nodes in the system. This model works well for a system that contains couple of hundreds of nodes. However, scaling such a design to run with tens of thousands of nodes is not trivial because the overhead in maintaining the routing table increases with the system size. This limitation might be overcome by introducing hierarchical extensions to Dynamo. Also, note that this problem is actively addressed by O(1) DHT systems(e.g., [14]).

# 7. CONCLUSIONS

This paper described Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon.com's e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based on their current

request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

The production use of Dynamo for the past year demonstrates that decentralized techniques can be combined to provide a single highly-available system. Its success in one of the most challenging application environments shows that an eventual-consistent storage system can be a building block for highly-available applications.

## ACKNOWLEDGEMENTS

The authors would like to thank Pat Helland for his contribution to the initial design of Dynamo. We would also like to thank Marvin Theimer and Robert van Renesse for their comments. Finally, we would like to thank our shepherd, Jeff Mogul, for his detailed comments and inputs while preparing the camera ready version that vastly improved the quality of the paper.

## REFERENCES

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 1-14.

[2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. on Database Systems, 9(4):596-615, December 1984

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006). USENIX Association, Berkeley, CA, 15-15.

[4] Douceur, J. R. and Bolosky, W. J. 2000. Process-based regulation of low-importance processes. *SIGOPS Oper. Syst. Rev.* 34, 2 (Apr. 2000), 26-27.

[5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 05 - 08, 1997). W. M. Waite, Ed. SOSP '97. ACM Press, New York, NY, 78-91.

[6] Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.

[7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996). J. Widom, Ed. SIGMOD '96. ACM Press, New York, NY, 173-182.

[8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing* (Newport, Rhode Island, United States). PODC '01. ACM Press, New York, NY, 170-179.

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News* 28, 5 (Dec. 2000), 190-201.

[10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997). STOC '97. ACM Press, New York, NY, 654-663.

[11] Lindsay, B.G., et. al., "Notes on Distributed Databases", Research Report RJ2571(33471), IBM Research, July 1979

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. ACM Communications, 21(7), pp. 558-565, 1978.

[13] Merkle, R. A digital signature based on a conventional encryption function. Proceedings of CRYPTO, pages 369–378. Springer-Verlag, 1988.

[14] Ramasubramanian, V., and Sirer, E. G. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation,* San Francisco, CA, March 29 - 31, 2004.

[15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994. Resolving file conflicts in the Ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994). USENIX Association, Berkeley, CA, 12-12..

[16] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware, pages 329-350, November, 2001.

[17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. Proceedings of Symposium on Operating Systems Principles, October 2001.

[18] Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. 2004. FAB: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev.* 38, 5 (Dec. 2004), 48-58.

[19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System. IEEE Workshop on Workstation Operating Systems, Nov. 1987.

[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM Press, New York, NY, 149-160.

[21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, United States, December 03 - 06, 1995). M. B. Jones, Ed. SOSP '95. ACM Press, New York, NY, 172-182.

[22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems 4 (2): 180-209, 1979.

[23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007. Antiquity: exploiting a secure log for wide-area distributed storage. *SIGOPS Oper. Syst. Rev.* 41, 3 (Jun. 2007), 371-384.

[24] Welsh, M., Culler, D., and Brewer, E. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada, October 21 - 24, 2001). SOSP '01. ACM Press, New York, NY, 230-243.

# A Few Useful Things to Know about Machine Learning

Pedro Domingos
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, U.S.A.
pedrod@cs.washington.edu

## ABSTRACT

Machine learning algorithms can figure out how to perform important tasks by generalizing from examples. This is often feasible and cost-effective where manual programming is not. As more data becomes available, more ambitious problems can be tackled. As a result, machine learning is widely used in computer science and other fields. However, developing successful machine learning applications requires a substantial amount of "black art" that is hard to find in textbooks. This article summarizes twelve key lessons that machine learning researchers and practitioners have learned. These include pitfalls to avoid, important issues to focus on, and answers to common questions.

## 1. INTRODUCTION

Machine learning systems automatically learn programs from data. This is often a very attractive alternative to manually constructing them, and in the last decade the use of machine learning has spread rapidly throughout computer science and beyond. Machine learning is used in Web search, spam filters, recommender systems, ad placement, credit scoring, fraud detection, stock trading, drug design, and many other applications. A recent report from the McKinsey Global Institute asserts that machine learning (a.k.a. data mining or predictive analytics) will be the driver of the next big wave of innovation [15]. Several fine textbooks are available to interested practitioners and researchers (e.g, [16, 24]). However, much of the "folk knowledge" that is needed to successfully develop machine learning applications is not readily available in them. As a result, many machine learning projects take much longer than necessary or wind up producing less-than-ideal results. Yet much of this folk knowledge is fairly easy to communicate. This is the purpose of this article.

Many different types of machine learning exist, but for illustration purposes I will focus on the most mature and widely used one: classification. Nevertheless, the issues I will discuss apply across all of machine learning. A *classifier* is a system that inputs (typically) a vector of discrete and/or continuous *feature values* and outputs a single discrete value, the *class*. For example, a spam filter classifies email messages into "spam" or "not spam," and its input may be a Boolean vector $\mathbf{x} = (x_1, \ldots, x_j, \ldots, x_d)$, where $x_j = 1$ if the $j$th word in the dictionary appears in the email and $x_j = 0$ otherwise. A *learner* inputs a *training set* of *examples* $(\mathbf{x}_i, y_i)$, where $\mathbf{x_i} = (x_{i,1}, \ldots, x_{i,d})$ is an observed input and $y_i$ is the corresponding output, and outputs a classifier. The test of the learner is whether this classifier produces the correct output $y_t$ for future examples $\mathbf{x_t}$ (e.g., whether the spam filter correctly classifies previously unseen emails as spam or not spam).

## 2. LEARNING = REPRESENTATION + EVALUATION + OPTIMIZATION

Suppose you have an application that you think machine learning might be good for. The first problem facing you is the bewildering variety of learning algorithms available. Which one to use? There are literally thousands available, and hundreds more are published each year. The key to not getting lost in this huge space is to realize that it consists of combinations of just three components. The components are:

**Representation.** A classifier must be represented in some formal language that the computer can handle. Conversely, choosing a representation for a learner is tantamount to choosing the set of classifiers that it can possibly learn. This set is called the *hypothesis space* of the learner. If a classifier is not in the hypothesis space, it cannot be learned. A related question, which we will address in a later section, is how to represent the input, i.e., what features to use.

**Evaluation.** An evaluation function (also called *objective function* or *scoring function)* is needed to distinguish good classifiers from bad ones. The evaluation function used internally by the algorithm may differ from the external one that we want the classifier to optimize, for ease of optimization (see below) and due to the issues discussed in the next section.

**Optimization.** Finally, we need a method to search among the classifiers in the language for the highest-scoring one. The choice of optimization technique is key to the efficiency of the learner, and also helps determine the classifier produced if the evaluation function has more than one optimum. It is common for new learners to start out using off-the-shelf optimizers, which are later replaced by custom-designed ones.

Table 1 shows common examples of each of these three components. For example, $k$-nearest neighbor classifies a test example by finding the $k$ most similar training examples and predicting the majority class among them. Hyperplane-based methods form a linear combination of the features per class and predict the class with the highest-valued combination. Decision trees test one feature at each internal node,

**Table 1: The three components of learning algorithms.**

| Representation | Evaluation | Optimization |
|---|---|---|
| Instances | Accuracy/Error rate | Combinatorial optimization |
| $K$-nearest neighbor | Precision and recall | Greedy search |
| Support vector machines | Squared error | Beam search |
| Hyperplanes | Likelihood | Branch-and-bound |
| Naive Bayes | Posterior probability | Continuous optimization |
| Logistic regression | Information gain | Unconstrained |
| Decision trees | K-L divergence | Gradient descent |
| Sets of rules | Cost/Utility | Conjugate gradient |
| Propositional rules | Margin | Quasi-Newton methods |
| Logic programs | | Constrained |
| Neural networks | | Linear programming |
| Graphical models | | Quadratic programming |
| Bayesian networks | | |
| Conditional random fields | | |

with one branch for each feature value, and have class predictions at the leaves. Algorithm 1 shows a bare-bones decision tree learner for Boolean domains, using information gain and greedy search [20]. InfoGain($x_j$,$y$) is the mutual information between feature $x_j$ and the class $y$. MakeNode($x$,$c_0$,$c_1$) returns a node that tests feature $x$ and has $c_0$ as the child for $x = 0$ and $c_1$ as the child for $x = 1$.

Of course, not all combinations of one component from each column of Table 1 make equal sense. For example, discrete representations naturally go with combinatorial optimization, and continuous ones with continuous optimization. Nevertheless, many learners have both discrete and continuous components, and in fact the day may not be far when every single possible combination has appeared in some learner!

Most textbooks are organized by representation, and it's easy to overlook the fact that the other components are equally important. There is no simple recipe for choosing each component, but the next sections touch on some of the key issues. And, as we will see below, some choices in a machine learning project may be even more important than the choice of learner.

## 3. IT'S GENERALIZATION THAT COUNTS

The fundamental goal of machine learning is to *generalize* beyond the examples in the training set. This is because, no matter how much data we have, it is very unlikely that we will see those exact examples again at test time. (Notice that, if there are 100,000 words in the dictionary, the spam filter described above has $2^{100,000}$ possible different inputs.) Doing well on the training set is easy (just memorize the examples). The most common mistake among machine learning beginners is to test on the training data and have the illusion of success. If the chosen classifier is then tested on new data, it is often no better than random guessing. So, if you hire someone to build a classifier, be sure to keep some of the data to yourself and test the classifier they give you on it. Conversely, if you've been hired to build a classifier, set some of the data aside from the beginning, and only use it to test your chosen classifier at the very end, followed by learning your final classifier on the whole data.

**Algorithm 1 LearnDT**(*TrainSet*)

```
if all examples in TrainSet have the same class y* then
  return MakeLeaf(y*)
if no feature x_j has InfoGain(x_j,y) > 0 then
  y* ← Most frequent class in TrainSet
  return MakeLeaf(y*)
x* ← argmax_{x_j} InfoGain(x_j,y)
TS_0 ← Examples in TrainSet with x* = 0
TS_1 ← Examples in TrainSet with x* = 1
return MakeNode(x*, LearnDT(TS_0), LearnDT(TS_1))
```

Contamination of your classifier by test data can occur in insidious ways, e.g., if you use test data to tune parameters and do a lot of tuning. (Machine learning algorithms have lots of knobs, and success often comes from twiddling them a lot, so this is a real concern.) Of course, holding out data reduces the amount available for training. This can be mitigated by doing cross-validation: randomly dividing your training data into (say) ten subsets, holding out each one while training on the rest, testing each learned classifier on the examples it did not see, and averaging the results to see how well the particular parameter setting does.

In the early days of machine learning, the need to keep training and test data separate was not widely appreciated. This was partly because, if the learner has a very limited representation (e.g., hyperplanes), the difference between training and test error may not be large. But with very flexible classifiers (e.g., decision trees), or even with linear classifiers with a lot of features, strict separation is mandatory.

Notice that generalization being the goal has an interesting consequence for machine learning. Unlike in most other optimization problems, we don't have access to the function we want to optimize! We have to use training error as a surrogate for test error, and this is fraught with danger. How to deal with it is addressed in some of the next sections. On the positive side, since the objective function is only a proxy for the true goal, we may not need to fully optimize it; in fact, a local optimum returned by simple greedy search may be better than the global optimum.

## 4. DATA ALONE IS NOT ENOUGH

Generalization being the goal has another major consequence: data alone is not enough, no matter how much of it you have. Consider learning a Boolean function of (say) 100 variables from a million examples. There are $2^{100} - 10^6$ examples whose classes you don't know. How do you figure out what those classes are? In the absence of further information, there is just no way to do this that beats flipping a coin. This observation was first made (in somewhat different form) by the philosopher David Hume over 200 years ago, but even today many mistakes in machine learning stem from failing to appreciate it. Every learner must embody some knowledge or assumptions beyond the data it's given in order to generalize beyond it. This was formalized by Wolpert in his famous "no free lunch" theorems, according to which no learner can beat random guessing over all possible functions to be learned [25].

This seems like rather depressing news. How then can we ever hope to learn anything? Luckily, the functions we want to learn in the real world are *not* drawn uniformly from the set of all mathematically possible functions! In fact, very general assumptions—like smoothness, similar examples having similar classes, limited dependences, or limited complexity—are often enough to do very well, and this is a large part of why machine learning has been so successful. Like deduction, induction (what learners do) is a knowledge lever: it turns a small amount of input knowledge into a large amount of output knowledge. Induction is a vastly more powerful lever than deduction, requiring much less input knowledge to produce useful results, but it still needs more than zero input knowledge to work. And, as with any lever, the more we put in, the more we can get out.

A corollary of this is that one of the key criteria for choosing a representation is which kinds of knowledge are easily expressed in it. For example, if we have a lot of knowledge about what makes examples similar in our domain, instance-based methods may be a good choice. If we have knowledge about probabilistic dependencies, graphical models are a good fit. And if we have knowledge about what kinds of preconditions are required by each class, "IF ... THEN ..." rules may be the the best option. The most useful learners in this regard are those that don't just have assumptions hard-wired into them, but allow us to state them explicitly, vary them widely, and incorporate them automatically into the learning (e.g., using first-order logic [21] or grammars [6]).

In retrospect, the need for knowledge in learning should not be surprising. Machine learning is not magic; it can't get something from nothing. What it does is get more from less. Programming, like all engineering, is a lot of work: we have to build everything from scratch. Learning is more like farming, which lets nature do most of the work. Farmers combine seeds with nutrients to grow crops. Learners combine knowledge with data to grow programs.

## 5. OVERFITTING HAS MANY FACES

What if the knowledge and data we have are not sufficient to completely determine the correct classifier? Then we run the risk of just hallucinating a classifier (or parts of it) that is not grounded in reality, and is simply encoding random

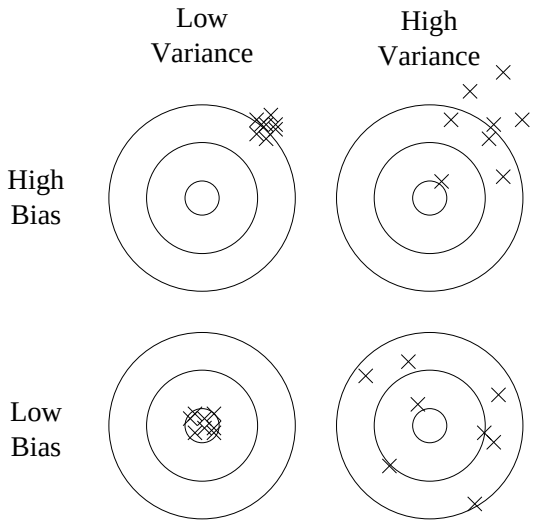



**Figure 1: Bias and variance in dart-throwing.**

quirks in the data. This problem is called *overfitting*, and is the bugbear of machine learning. When your learner outputs a classifier that is 100% accurate on the training data but only 50% accurate on test data, when in fact it could have output one that is 75% accurate on both, it has overfit.

Everyone in machine learning knows about overfitting, but it comes in many forms that are not immediately obvious. One way to understand overfitting is by decomposing generalization error into *bias* and *variance* [9]. Bias is a learner's tendency to consistently learn the same wrong thing. Variance is the tendency to learn random things irrespective of the real signal. Figure 1 illustrates this by an analogy with throwing darts at a board. A linear learner has high bias, because when the frontier between two classes is not a hyperplane the learner is unable to induce it. Decision trees don't have this problem because they can represent any Boolean function, but on the other hand they can suffer from high variance: decision trees learned on different training sets generated by the same phenomenon are often very different, when in fact they should be the same. Similar reasoning applies to the choice of optimization method: beam search has lower bias than greedy search, but higher variance, because it tries more hypotheses. Thus, contrary to intuition, a more powerful learner is not necessarily better than a less powerful one.

Figure 2 illustrates this.[1] Even though the true classifier is a set of rules, with up to 1000 examples naive Bayes is more accurate than a rule learner. This happens despite naive Bayes's false assumption that the frontier is linear! Situations like this are common in machine learning: strong false assumptions can be better than weak true ones, because a learner with the latter needs more data to avoid overfitting.

[1]Training examples consist of 64 Boolean features and a Boolean class computed from them according to a set of "IF ... THEN ..." rules. The curves are the average of 100 runs with different randomly generated sets of rules. Error bars are two standard deviations. See Domingos and Pazzani [10] for details.

Cross-validation can help to combat overfitting, for example by using it to choose the best size of decision tree to learn. But it's no panacea, since if we use it to make too many parameter choices it can itself start to overfit [17].

Besides cross-validation, there are many methods to combat overfitting. The most popular one is adding a *regularization term* to the evaluation function. This can, for example, penalize classifiers with more structure, thereby favoring smaller ones with less room to overfit. Another option is to perform a statistical significance test like chi-square before adding new structure, to decide whether the distribution of the class really is different with and without this structure. These techniques are particularly useful when data is very scarce. Nevertheless, you should be skeptical of claims that a particular technique "solves" the overfitting problem. It's easy to avoid overfitting (variance) by falling into the opposite error of underfitting (bias). Simultaneously avoiding both requires learning a perfect classifier, and short of knowing it in advance there is no single technique that will always do best (no free lunch).

A common misconception about overfitting is that it is caused by noise, like training examples labeled with the wrong class. This can indeed aggravate overfitting, by making the learner draw a capricious frontier to keep those examples on what it thinks is the right side. But severe overfitting can occur even in the absence of noise. For instance, suppose we learn a Boolean classifier that is just the disjunction of the examples labeled "true" in the training set. (In other words, the classifier is a Boolean formula in disjunctive normal form, where each term is the conjunction of the feature values of one specific training example). This classifier gets all the training examples right and every positive test example wrong, regardless of whether the training data is noisy or not.

The problem of *multiple testing* [13] is closely related to overfitting. Standard statistical tests assume that only one hypothesis is being tested, but modern learners can easily test millions before they are done. As a result what looks significant may in fact not be. For example, a mutual fund that beats the market ten years in a row looks very impressive, until you realize that, if there are 1000 funds and each has a 50% chance of beating the market on any given year, it's quite likely that one will succeed all ten times just by luck. This problem can be combatted by correcting the significance tests to take the number of hypotheses into account, but this can lead to underfitting. A better approach is to control the fraction of falsely accepted non-null hypotheses, known as the *false discovery rate* [3].

## 6. INTUITION FAILS IN HIGH DIMENSIONS

After overfitting, the biggest problem in machine learning is the *curse of dimensionality*. This expression was coined by Bellman in 1961 to refer to the fact that many algorithms that work fine in low dimensions become intractable when the input is high-dimensional. But in machine learning it refers to much more. Generalizing correctly becomes exponentially harder as the dimensionality (number of features) of the examples grows, because a fixed-size training set covers a dwindling fraction of the input space. Even with a moderate dimension of 100 and a huge training set of a trillion examples, the latter covers only a fraction of about $10^{-18}$ of the input space. This is what makes machine learning both necessary and hard.

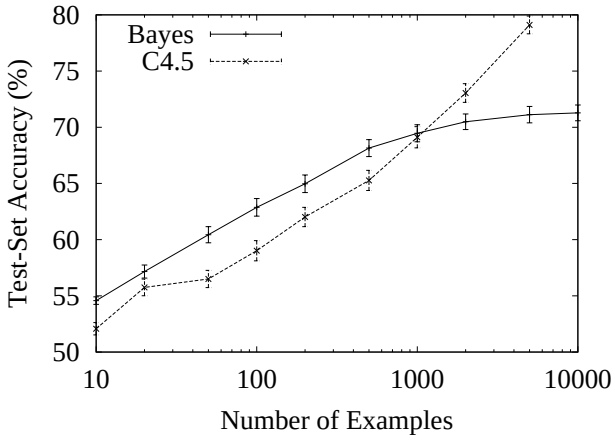



**Figure 2: Naive Bayes can outperform a state-of-the-art rule learner (C4.5rules) even when the true classifier is a set of rules.**

More seriously, the similarity-based reasoning that machine learning algorithms depend on (explicitly or implicitly) breaks down in high dimensions. Consider a nearest neighbor classifier with Hamming distance as the similarity measure, and suppose the class is just $x_1 \wedge x_2$. If there are no other features, this is an easy problem. But if there are 98 irrelevant features $x_3, \ldots, x_{100}$, the noise from them completely swamps the signal in $x_1$ and $x_2$, and nearest neighbor effectively makes random predictions.

Even more disturbing is that nearest neighbor still has a problem even if all 100 features are relevant! This is because in high dimensions all examples look alike. Suppose, for instance, that examples are laid out on a regular grid, and consider a test example $\mathbf{x_t}$. If the grid is $d$-dimensional, $\mathbf{x_t}$'s $2d$ nearest examples are all at the same distance from it. So as the dimensionality increases, more and more examples become nearest neighbors of $\mathbf{x_t}$, until the choice of nearest neighbor (and therefore of class) is effectively random.

This is only one instance of a more general problem with high dimensions: our intuitions, which come from a three-dimensional world, often do not apply in high-dimensional ones. In high dimensions, most of the mass of a multivariate Gaussian distribution is not near the mean, but in an increasingly distant "shell" around it; and most of the volume of a high-dimensional orange is in the skin, not the pulp. If a constant number of examples is distributed uniformly in a high-dimensional hypercube, beyond some dimensionality most examples are closer to a face of the hypercube than to their nearest neighbor. And if we approximate a hypersphere by inscribing it in a hypercube, in high dimensions almost all the volume of the hypercube is outside the hypersphere. This is bad news for machine learning, where shapes of one type are often approximated by shapes of another.

Building a classifier in two or three dimensions is easy; we

can find a reasonable frontier between examples of different classes just by visual inspection. (It's even been said that if people could see in high dimensions machine learning would not be necessary.) But in high dimensions it's hard to understand what is happening. This in turn makes it difficult to design a good classifier. Naively, one might think that gathering more features never hurts, since at worst they provide no new information about the class. But in fact their benefits may be outweighed by the curse of dimensionality.

Fortunately, there is an effect that partly counteracts the curse, which might be called the "blessing of non-uniformity." In most applications examples are not spread uniformly throughout the instance space, but are concentrated on or near a lower-dimensional manifold. For example, $k$-nearest neighbor works quite well for handwritten digit recognition even though images of digits have one dimension per pixel, because the space of digit images is much smaller than the space of all possible images. Learners can implicitly take advantage of this lower effective dimension, or algorithms for explicitly reducing the dimensionality can be used (e.g., [22]).

## 7. THEORETICAL GUARANTEES ARE NOT WHAT THEY SEEM

Machine learning papers are full of theoretical guarantees. The most common type is a bound on the number of examples needed to ensure good generalization. What should you make of these guarantees? First of all, it's remarkable that they are even possible. Induction is traditionally contrasted with deduction: in deduction you can guarantee that the conclusions are correct; in induction all bets are off. Or such was the conventional wisdom for many centuries. One of the major developments of recent decades has been the realization that in fact we can have guarantees on the results of induction, particularly if we're willing to settle for probabilistic guarantees.

The basic argument is remarkably simple [5]. Let's say a classifier is bad if its true error rate is greater than $\epsilon$. Then the probability that a bad classifier is consistent with $n$ random, independent training examples is less than $(1-\epsilon)^n$. Let $b$ be the number of bad classifiers in the learner's hypothesis space $H$. The probability that at least one of them is consistent is less than $b(1-\epsilon)^n$, by the union bound. Assuming the learner always returns a consistent classifier, the probability that this classifier is bad is then less than $|H|(1-\epsilon)^n$, where we have used the fact that $b \leq |H|$. So if we want this probability to be less than $\delta$, it suffices to make $n > \ln(\delta/|H|)/\ln(1-\epsilon) \geq \frac{1}{\epsilon}\left(\ln|H| + \ln\frac{1}{\delta}\right)$.

Unfortunately, guarantees of this type have to be taken with a large grain of salt. This is because the bounds obtained in this way are usually extremely loose. The wonderful feature of the bound above is that the required number of examples only grows logarithmically with $|H|$ and $1/\delta$. Unfortunately, most interesting hypothesis spaces are *doubly* exponential in the number of features $d$, which still leaves us needing a number of examples exponential in $d$. For example, consider the space of Boolean functions of $d$ Boolean variables. If there are $e$ possible different examples, there are $2^e$ possible different functions, so since there are $2^d$ possible examples, the total number of functions is $2^{2^d}$. And even for hypothesis spaces that are "merely" exponential, the bound is still very loose, because the union bound is very pessimistic. For example, if there are 100 Boolean features and the hypothesis space is decision trees with up to 10 levels, to guarantee $\delta = \epsilon = 1\%$ in the bound above we need half a million examples. But in practice a small fraction of this suffices for accurate learning.

Further, we have to be careful about what a bound like this means. For instance, it does not say that, if your learner returned a hypothesis consistent with a particular training set, then this hypothesis probably generalizes well. What it says is that, given a large enough training set, with high probability your learner will either return a hypothesis that generalizes well or be unable to find a consistent hypothesis. The bound also says nothing about how to select a good hypothesis space. It only tells us that, if the hypothesis space contains the true classifier, then the probability that the learner outputs a bad classifier decreases with training set size. If we shrink the hypothesis space, the bound improves, but the chances that it contains the true classifier shrink also. (There are bounds for the case where the true classifier is not in the hypothesis space, but similar considerations apply to them.)

Another common type of theoretical guarantee is asymptotic: given infinite data, the learner is guaranteed to output the correct classifier. This is reassuring, but it would be rash to choose one learner over another because of its asymptotic guarantees. In practice, we are seldom in the asymptotic regime (also known as "asymptopia"). And, because of the bias-variance tradeoff we discussed above, if learner A is better than learner B given infinite data, B is often better than A given finite data.

The main role of theoretical guarantees in machine learning is not as a criterion for practical decisions, but as a source of understanding and driving force for algorithm design. In this capacity, they are quite useful; indeed, the close interplay of theory and practice is one of the main reasons machine learning has made so much progress over the years. But *caveat emptor*: learning is a complex phenomenon, and just because a learner has a theoretical justification and works in practice doesn't mean the former is the reason for the latter.

## 8. FEATURE ENGINEERING IS THE KEY

At the end of the day, some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used. If you have many independent features that each correlate well with the class, learning is easy. On the other hand, if the class is a very complex function of the features, you may not be able to learn it. Often, the raw data is not in a form that is amenable to learning, but you can construct features from it that are. This is typically where most of the effort in a machine learning project goes. It is often also one of the most interesting parts, where intuition, creativity and "black art" are as important as the technical stuff.

First-timers are often surprised by how little time in a machine learning project is spent actually doing machine learning. But it makes sense if you consider how time-consuming

it is to gather data, integrate it, clean it and pre-process it, and how much trial and error can go into feature design. Also, machine learning is not a one-shot process of building a data set and running a learner, but rather an iterative process of running the learner, analyzing the results, modifying the data and/or the learner, and repeating. Learning is often the quickest part of this, but that's because we've already mastered it pretty well! Feature engineering is more difficult because it's domain-specific, while learners can be largely general-purpose. However, there is no sharp frontier between the two, and this is another reason the most useful learners are those that facilitate incorporating knowledge.

Of course, one of the holy grails of machine learning is to automate more and more of the feature engineering process. One way this is often done today is by automatically generating large numbers of candidate features and selecting the best by (say) their information gain with respect to the class. But bear in mind that features that look irrelevant in isolation may be relevant in combination. For example, if the class is an XOR of $k$ input features, each of them by itself carries no information about the class. (If you want to annoy machine learners, bring up XOR.) On the other hand, running a learner with a very large number of features to find out which ones are useful in combination may be too time-consuming, or cause overfitting. So there is ultimately no replacement for the smarts you put into feature engineering.

## 9. MORE DATA BEATS A CLEVERER ALGORITHM

Suppose you've constructed the best set of features you can, but the classifiers you're getting are still not accurate enough. What can you do now? There are two main choices: design a better learning algorithm, or gather more data (more examples, and possibly more raw features, subject to the curse of dimensionality). Machine learning researchers are mainly concerned with the former, but pragmatically the quickest path to success is often to just get more data. As a rule of thumb, a dumb algorithm with lots and lots of data beats a clever one with modest amounts of it. (After all, machine learning is all about letting data do the heavy lifting.)

This does bring up another problem, however: scalability. In most of computer science, the two main limited resources are time and memory. In machine learning, there is a third one: training data. Which one is the bottleneck has changed from decade to decade. In the 1980's it tended to be data. Today it is often time. Enormous mountains of data are available, but there is not enough time to process it, so it goes unused. This leads to a paradox: even though in principle more data means that more complex classifiers can be learned, in practice simpler classifiers wind up being used, because complex ones take too long to learn. Part of the answer is to come up with fast ways to learn complex classifiers, and indeed there has been remarkable progress in this direction (e.g., [11]).

Part of the reason using cleverer algorithms has a smaller payoff than you might expect is that, to a first approximation, they all do the same. This is surprising when you consider representations as different as, say, sets of rules

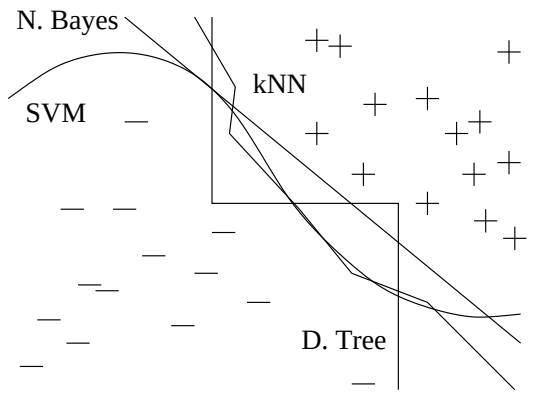



**Figure 3: Very different frontiers can yield similar class predictions. (+ and − are training examples of two classes.)**

and neural networks. But in fact propositional rules are readily encoded as neural networks, and similar relationships hold between other representations. All learners essentially work by grouping nearby examples into the same class; the key difference is in the meaning of "nearby." With non-uniformly distributed data, learners can produce widely different frontiers while still making the same predictions in the regions that matter (those with a substantial number of training examples, and therefore also where most test examples are likely to appear). This also helps explain why powerful learners can be unstable but still accurate. Figure 3 illustrates this in 2-D; the effect is much stronger in high dimensions.

As a rule, it pays to try the simplest learners first (e.g., naive Bayes before logistic regression, $k$-nearest neighbor before support vector machines). More sophisticated learners are seductive, but they are usually harder to use, because they have more knobs you need to turn to get good results, and because their internals are more opaque.

Learners can be divided into two major types: those whose representation has a fixed size, like linear classifiers, and those whose representation can grow with the data, like decision trees. (The latter are sometimes called non-parametric learners, but this is somewhat unfortunate, since they usually wind up learning many more parameters than parametric ones.) Fixed-size learners can only take advantage of so much data. (Notice how the accuracy of naive Bayes asymptotes at around 70% in Figure 2.) Variable-size learners can in principle learn any function given sufficient data, but in practice they may not, because of limitations of the algorithm (e.g., greedy search falls into local optima) or computational cost. Also, because of the curse of dimensionality, no existing amount of data may be enough. For these reasons, clever algorithms—those that make the most of the data and computing resources available—often pay off in the end, provided you're willing to put in the effort. There is no sharp frontier between designing learners and learning classifiers; rather, any given piece of knowledge could be encoded in the learner or learned from data. So machine learning projects often wind up having a significant component of learner design, and practitioners need to have some expertise in it [12].

In the end, the biggest bottleneck is not data or CPU cycles, but human cycles. In research papers, learners are typically compared on measures of accuracy and computational cost. But human effort saved and insight gained, although harder to measure, are often more important. This favors learners that produce human-understandable output (e.g., rule sets). And the organizations that make the most of machine learning are those that have in place an infrastructure that makes experimenting with many different learners, data sources and learning problems easy and efficient, and where there is a close collaboration between machine learning experts and application domain ones.

## 10. LEARN MANY MODELS, NOT JUST ONE

In the early days of machine learning, everyone had their favorite learner, together with some *a priori* reasons to believe in its superiority. Most effort went into trying many variations of it and selecting the best one. Then systematic empirical comparisons showed that the best learner varies from application to application, and systems containing many different learners started to appear. Effort now went into trying many variations of many learners, and still selecting just the best one. But then researchers noticed that, if instead of selecting the best variation found, we combine many variations, the results are better—often much better—and at little extra effort for the user.

Creating such *model ensembles* is now standard [1]. In the simplest technique, called *bagging*, we simply generate random variations of the training set by resampling, learn a classifier on each, and combine the results by voting. This works because it greatly reduces variance while only slightly increasing bias. In *boosting*, training examples have weights, and these are varied so that each new classifier focuses on the examples the previous ones tended to get wrong. In *stacking*, the outputs of individual classifiers become the inputs of a "higher-level" learner that figures out how best to combine them.

Many other techniques exist, and the trend is toward larger and larger ensembles. In the Netflix prize, teams from all over the world competed to build the best video recommender system (http://netflixprize.com). As the competition progressed, teams found that they obtained the best results by combining their learners with other teams', and merged into larger and larger teams. The winner and runner-up were both stacked ensembles of over 100 learners, and combining the two ensembles further improved the results. Doubtless we will see even larger ones in the future.

Model ensembles should not be confused with Bayesian model averaging (BMA). BMA is the theoretically optimal approach to learning [4]. In BMA, predictions on new examples are made by averaging the individual predictions of *all* classifiers in the hypothesis space, weighted by how well the classifiers explain the training data and how much we believe in them *a priori*. Despite their superficial similarities, ensembles and BMA are very different. Ensembles change the hypothesis space (e.g., from single decision trees to linear combinations of them), and can take a wide variety of forms. BMA assigns weights to the hypotheses in the original space according to a fixed formula. BMA weights are extremely different from those produced by (say) bagging or boosting: the latter are fairly even, while the former are extremely skewed, to the point where the single highest-weight classifier usually dominates, making BMA effectively equivalent to just selecting it [8]. A practical consequence of this is that, while model ensembles are a key part of the machine learning toolkit, BMA is seldom worth the trouble.

## 11. SIMPLICITY DOES NOT IMPLY ACCURACY

Occam's razor famously states that entities should not be multiplied beyond necessity. In machine learning, this is often taken to mean that, given two classifiers with the same training error, the simpler of the two will likely have the lowest test error. Purported proofs of this claim appear regularly in the literature, but in fact there are many counter-examples to it, and the "no free lunch" theorems imply it cannot be true.

We saw one counter-example in the previous section: model ensembles. The generalization error of a boosted ensemble continues to improve by adding classifiers even after the training error has reached zero. Another counter-example is support vector machines, which can effectively have an infinite number of parameters without overfitting. Conversely, the function $\text{sign}(\sin(ax))$ can discriminate an arbitrarily large, arbitrarily labeled set of points on the $x$ axis, even though it has only one parameter [23]. Thus, contrary to intuition, there is no necessary connection between the number of parameters of a model and its tendency to overfit.

A more sophisticated view instead equates complexity with the size of the hypothesis space, on the basis that smaller spaces allow hypotheses to be represented by shorter codes. Bounds like the one in the section on theoretical guarantees above might then be viewed as implying that shorter hypotheses generalize better. This can be further refined by assigning shorter codes to the hypothesis in the space that we have some *a priori* preference for. But viewing this as "proof" of a tradeoff between accuracy and simplicity is circular reasoning: we made the hypotheses we prefer simpler by design, and if they are accurate it's because our preferences are accurate, not because the hypotheses are "simple" in the representation we chose.

A further complication arises from the fact that few learners search their hypothesis space exhaustively. A learner with a larger hypothesis space that tries fewer hypotheses from it is less likely to overfit than one that tries more hypotheses from a smaller space. As Pearl [18] points out, the size of the hypothesis space is only a rough guide to what really matters for relating training and test error: the procedure by which a hypothesis is chosen.

Domingos [7] surveys the main arguments and evidence on the issue of Occam's razor in machine learning. The conclusion is that simpler hypotheses should be preferred because simplicity is a virtue in its own right, not because of a hypothetical connection with accuracy. This is probably what Occam meant in the first place.

## 12. REPRESENTABLE DOES NOT IMPLY LEARNABLE

Essentially all representations used in variable-size learners have associated theorems of the form "Every function can be represented, or approximated arbitrarily closely, using this representation." Reassured by this, fans of the representation often proceed to ignore all others. However, just because a function can be represented does not mean it can be learned. For example, standard decision tree learners cannot learn trees with more leaves than there are training examples. In continuous spaces, representing even simple functions using a fixed set of primitives often requires an infinite number of components. Further, if the hypothesis space has many local optima of the evaluation function, as is often the case, the learner may not find the true function even if it is representable. Given finite data, time and memory, standard learners can learn only a tiny subset of all possible functions, and these subsets are different for learners with different representations. Therefore the key question is not "Can it be represented?", to which the answer is often trivial, but "Can it be learned?" And it pays to try different learners (and possibly combine them).

Some representations are exponentially more compact than others for some functions. As a result, they may also require exponentially less data to learn those functions. Many learners work by forming linear combinations of simple basis functions. For example, support vector machines form combinations of kernels centered at some of the training examples (the support vectors). Representing parity of $n$ bits in this way requires $2^n$ basis functions. But using a representation with more layers (i.e., more steps between input and output), parity can be encoded in a linear-size classifier. Finding methods to learn these deeper representations is one of the major research frontiers in machine learning [2].

## 13. CORRELATION DOES NOT IMPLY CAUSATION

The point that correlation does not imply causation is made so often that it is perhaps not worth belaboring. But, even though learners of the kind we have been discussing can only learn correlations, their results are often treated as representing causal relations. Isn't this wrong? If so, then why do people do it?

More often than not, the goal of learning predictive models is to use them as guides to action. If we find that beer and diapers are often bought together at the supermarket, then perhaps putting beer next to the diaper section will increase sales. (This is a famous example in the world of data mining.) But short of actually doing the experiment it's difficult to tell. Machine learning is usually applied to *observational* data, where the predictive variables are not under the control of the learner, as opposed to *experimental* data, where they are. Some learning algorithms can potentially extract causal information from observational data, but their applicability is rather restricted [19]. On the other hand, correlation is a sign of a potential causal connection, and we can use it as a guide to further investigation (for example, trying to understand what the causal chain might be).

Many researchers believe that causality is only a convenient fiction. For example, there is no notion of causality in physical laws. Whether or not causality really exists is a deep philosophical question with no definitive answer in sight, but the practical points for machine learners are two. First, whether or not we call them "causal," we would like to predict the effects of our actions, not just correlations between observable variables. Second, if you can obtain experimental data (for example by randomly assigning visitors to different versions of a Web site), then by all means do so [14].

## 14. CONCLUSION

Like any discipline, machine learning has a lot of "folk wisdom" that can be hard to come by, but is crucial for success. This article summarized some of the most salient items. Of course, it's only a complement to the more conventional study of machine learning. Check out http://www.cs.washington.edu/homes/pedrod/class for a complete online machine learning course that combines formal and informal aspects. There's also a treasure trove of machine learning lectures at http://www.videolectures.net. A good open source machine learning toolkit is Weka [24]. Happy learning!

## 15. REFERENCES


[1] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting and variants. *Machine Learning*, 36:105–142, 1999.

[2] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2:1–127, 2009.

[3] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society, Series B*, 57:289–300, 1995.

[4] J. M. Bernardo and A. F. M. Smith. *Bayesian Theory*. Wiley, New York, NY, 1994.

[5] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, 1987.

[6] W. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[7] P. Domingos. The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3:409–425, 1999.

[8] P. Domingos. Bayesian averaging of classifiers and the overfitting problem. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 223–230, Stanford, CA, 2000. Morgan Kaufmann.

[9] P. Domingos. A unified bias-variance decomposition and its applications. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 231–238, Stanford, CA, 2000. Morgan Kaufmann.

[10] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.

[11] G. Hulten and P. Domingos. Mining complex models from arbitrarily large databases in constant time. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 525–531, Edmonton, Canada, 2002. ACM Press.

[12] D. Kibler and P. Langley. Machine learning as an experimental science. In *Proceedings of the Third European Working Session on Learning*, London, UK, 1988. Pitman.

[13] A. J. Klockars and G. Sax. *Multiple Comparisons*. Sage, Beverly Hills, CA, 1986.

[14] R. Kohavi, R. Longbotham, D. Sommerfield, and R. Henne. Controlled experiments on the Web: Survey and practical guide. *Data Mining and Knowledge Discovery*, 18:140–181, 2009.

[15] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, 2011.

[16] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, NY, 1997.

[17] A. Y. Ng. Preventing "overfitting" of cross-validation data. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 245–253, Nashville, TN, 1997. Morgan Kaufmann.

[18] J. Pearl. On the connection between the complexity and credibility of inferred models. *International Journal of General Systems*, 4:255–264, 1978.

[19] J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK, 2000.

[20] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[21] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62:107–136, 2006.

[22] J. Tenenbaum, V. Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, 2000.

[23] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, NY, 1995.

[24] I. Witten, E. Frank, and M. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Mateo, CA, 3rd edition, 2011.

[25] D. Wolpert. The lack of *a priori* distinctions between learning algorithms. *Neural Computation*, 8:1341–1390, 1996.

# 15
# Random Forests

## 15.1 Introduction

Bagging or *bootstrap aggregation* (section 8.7) is a technique for reducing the variance of an estimated prediction function. Bagging seems to work especially well for high-variance, low-bias procedures, such as trees. For regression, we simply fit the same regression tree many times to bootstrap-sampled versions of the training data, and average the result. For classification, a *committee* of trees each cast a vote for the predicted class.

Boosting in Chapter 10 was initially proposed as a committee method as well, although unlike bagging, the committee of *weak learners* evolves over time, and the members cast a weighted vote. Boosting appears to dominate bagging on most problems, and became the preferred choice.

*Random forests* (Breiman, 2001) is a substantial modification of bagging that builds a large collection of *de-correlated* trees, and then averages them. On many problems the performance of random forests is very similar to boosting, and they are simpler to train and tune. As a consequence, random forests are popular, and are implemented in a variety of packages.

## 15.2 Definition of Random Forests

The essential idea in bagging (Section 8.7) is to average many noisy but approximately unbiased models, and hence reduce the variance. Trees are ideal candidates for bagging, since they can capture complex interaction

**Algorithm 15.1** *Random Forest for Regression or Classification.*

1. For $b = 1$ to $B$:
   (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size $N$ from the training data.
   (b) Grow a random-forest tree $T_b$ to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size $n_{min}$ is reached.
      i. Select $m$ variables at random from the $p$ variables.
      ii. Pick the best variable/split-point among the $m$.
      iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point $x$:

*Regression:* $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^B T_b(x)$.

*Classification:* Let $\hat{C}_b(x)$ be the class prediction of the $b$th random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \textit{majority vote}\ \{\hat{C}_b(x)\}_1^B$.

structures in the data, and if grown sufficiently deep, have relatively low bias. Since trees are notoriously noisy, they benefit greatly from the averaging. Moreover, since each tree generated in bagging is identically distributed (i.d.), the expectation of an average of $B$ such trees is the same as the expectation of any one of them. This means the bias of bagged trees is the same as that of the individual trees, and the only hope of improvement is through variance reduction. This is in contrast to boosting, where the trees are grown in an adaptive way to remove bias, and hence are not i.d.

An average of $B$ i.i.d. random variables, each with variance $\sigma^2$, has variance $\frac{1}{B}\sigma^2$. If the variables are simply i.d. (identically distributed, but not necessarily independent) with positive pairwise correlation $\rho$, the variance of the average is (Exercise 15.1)

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2. \tag{15.1}$$

As $B$ increases, the second term disappears, but the first remains, and hence the size of the correlation of pairs of bagged trees limits the benefits of averaging. The idea in random forests (Algorithm 15.1) is to improve the variance reduction of bagging by reducing the correlation between the trees, without increasing the variance too much. This is achieved in the tree-growing process through random selection of the input variables.

Specifically, when growing a tree on a bootstrapped dataset:

> *Before each split, select $m \leq p$ of the input variables at random as candidates for splitting.*

Typically values for $m$ are $\sqrt{p}$ or even as low as 1.

After $B$ such trees $\{T(x;\Theta_b)\}_1^B$ are grown, the random forest (regression) predictor is

$$\hat{f}_{\mathrm{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T(x;\Theta_b). \tag{15.2}$$

As in Section 10.9 (page 356), $\Theta_b$ characterizes the $b$th random forest tree in terms of split variables, cutpoints at each node, and terminal-node values. Intuitively, reducing $m$ will reduce the correlation between any pair of trees in the ensemble, and hence by (15.1) reduce the variance of the average.

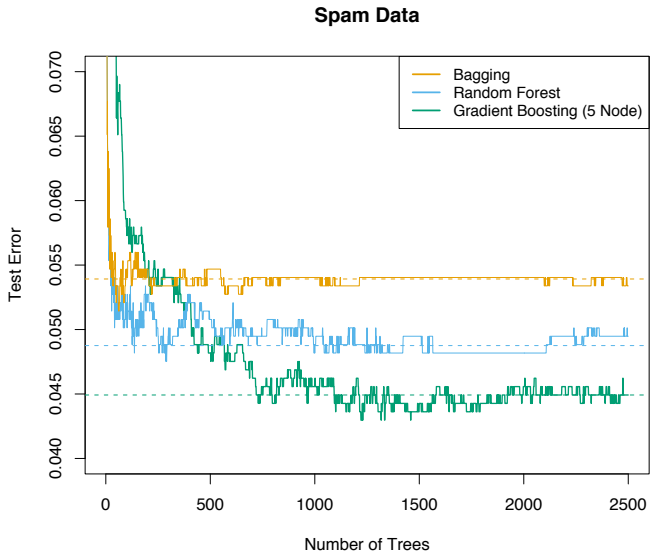



**FIGURE 15.1.** *Bagging, random forest, and gradient boosting, applied to the spam data. For boosting,* 5*-node trees were used, and the number of trees were chosen by* 10*-fold cross-validation (*2500 *trees). Each "step" in the figure corresponds to a change in a single misclassification (in a test set of* 1536*).*

Not all estimators can be improved by shaking up the data like this. It seems that highly nonlinear estimators, such as trees, benefit the most. For bootstrapped trees, $\rho$ is typically small (0.05 or lower is typical; see Figure 15.9), while $\sigma^2$ is not much larger than the variance for the original tree. On the other hand, bagging does not change *linear* estimates, such as the sample mean (hence its variance either); the pairwise correlation between bootstrapped means is about 50% (Exercise 15.4).

Random forests are popular. Leo Breiman's[1] collaborator Adele Cutler maintains a random forest website[2] where the software is freely available, with more than 3000 downloads reported by 2002. There is a `randomForest` package in R, maintained by Andy Liaw, available from the `CRAN` website.

The authors make grand claims about the success of random forests: "most accurate," "most interpretable," and the like. In our experience random forests do remarkably well, with very little tuning required. A random forest classifier achieves 4.88% misclassification error on the `spam` test data, which compares well with all other methods, and is not significantly worse than gradient boosting at 4.5%. Bagging achieves 5.4% which is significantly worse than either (using the McNemar test outlined in Exercise 10.6), so it appears on this example the additional randomization helps.

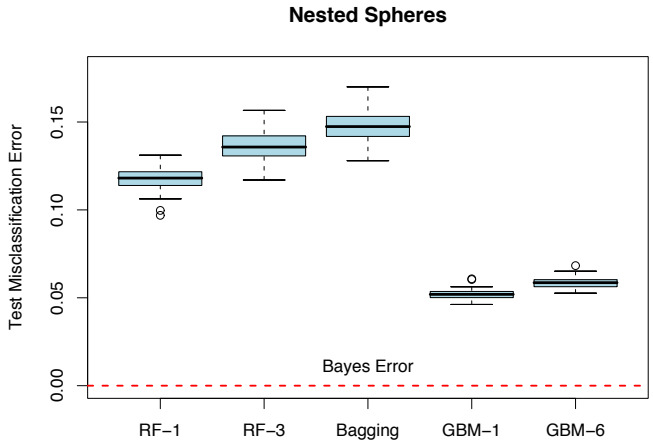



**FIGURE 15.2.** *The results of* 50 *simulations from the "nested spheres" model in* $\mathbb{R}^{10}$. *The Bayes decision boundary is the surface of a sphere (additive). "RF-3" refers to a random forest with* $m = 3$, *and "GBM-6" a gradient boosted model with interaction order six; similarly for "RF-1" and "GBM-1." The training sets were of size* 2000, *and the test sets* $10,000$.

Figure 15.1 shows the test-error progression on 2500 trees for the three methods. In this case there is some evidence that gradient boosting has started to overfit, although 10-fold cross-validation chose all 2500 trees.

[1]Sadly, Leo Breiman died in July, 2005.

[2]`http://www.math.usu.edu/~adele/forests/`

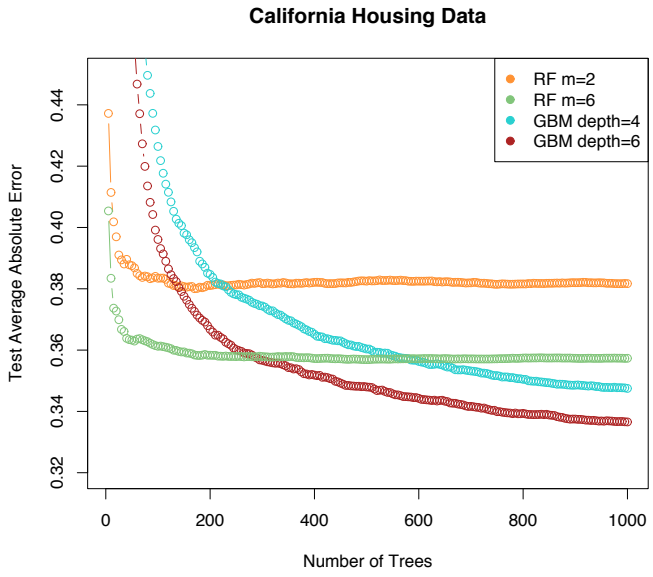



**FIGURE 15.3.** *Random forests compared to gradient boosting on the California housing data. The curves represent mean absolute error on the test data as a function of the number of trees in the models. Two random forests are shown, with* $m = 2$ *and* $m = 6$*. The two gradient boosted models use a shrinkage parameter* $\nu = 0.05$ *in (10.41), and have interaction depths of* 4 *and* 6*. The boosted models outperform random forests.*

Figure 15.2 shows the results of a simulation[3] comparing random forests to gradient boosting on the *nested spheres* problem [Equation (10.2) in Chapter 10]. Boosting easily outperforms random forests here. Notice that smaller $m$ is better here, although part of the reason could be that the true decision boundary is additive.

Figure 15.3 compares random forests to boosting (with shrinkage) in a regression problem, using the California housing data (Section 10.14.1). Two strong features that emerge are

- Random forests stabilize at about 200 trees, while at 1000 trees boosting continues to improve. Boosting is slowed down by the shrinkage, as well as the fact that the trees are much smaller.

- Boosting outperforms random forests here. At 1000 terms, the weaker boosting model (GBM depth 4) has a smaller error than the stronger

[3]Details: The random forests were fit using the R package `randomForest 4.5-11`, with 500 trees. The gradient boosting models were fit using R package `gbm 1.5`, with shrinkage parameter set to 0.05, and 2000 trees.

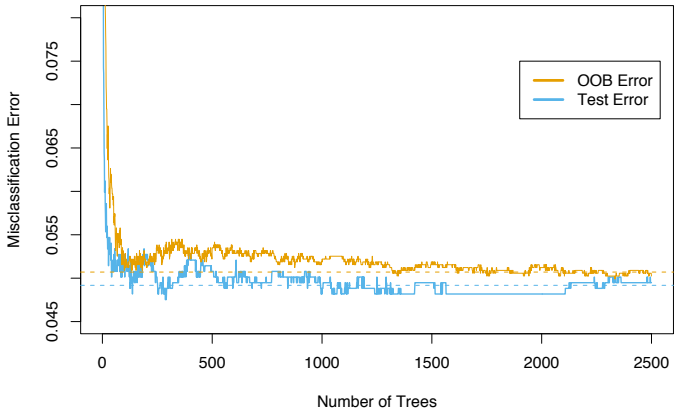



**FIGURE 15.4.** OOB *error computed on the* `spam` *training data, compared to the test error computed on the test set.*

random forest (RF $m = 6$); a Wilcoxon test on the mean differences in absolute errors has a p-value of 0.007. For larger $m$ the random forests performed no better.

## 15.3 Details of Random Forests

We have glossed over the distinction between random forests for classification versus regression. When used for classification, a random forest obtains a class vote from each tree, and then classifies using majority vote (see Section 8.7 on bagging for a similar discussion). When used for regression, the predictions from each tree at a target point $x$ are simply averaged, as in (15.2). In addition, the inventors make the following recommendations:

- For classification, the default value for $m$ is $\lfloor\sqrt{p}\rfloor$ and the minimum node size is one.
- For regression, the default value for $m$ is $\lfloor p/3\rfloor$ and the minimum node size is five.

In practice the best values for these parameters will depend on the problem, and they should be treated as tuning parameters. In Figure 15.3 the $m = 6$ performs much better than the default value $\lfloor 8/3\rfloor = 2$.

### *15.3.1 Out of Bag Samples*

An important feature of random forests is its use of *out-of-bag* (OOB) samples:

> *For each observation $z_i = (x_i, y_i)$, construct its random forest predictor by averaging* only *those trees corresponding to bootstrap samples in which $z_i$* did not *appear.*

An OOB error estimate is almost identical to that obtained by $N$-fold cross-validation; see Exercise 15.2. Hence unlike many other nonlinear estimators, random forests can be fit in one sequence, with cross-validation being performed along the way. Once the OOB error stabilizes, the training can be terminated.

Figure 15.4 shows the OOB misclassification error for the `spam` data, compared to the test error. Although 2500 trees are averaged here, it appears from the plot that about 200 would be sufficient.

### *15.3.2 Variable Importance*

Variable importance plots can be constructed for random forests in exactly the same way as they were for gradient-boosted models (Section 10.13). At each split in each tree, the improvement in the split-criterion is the importance measure attributed to the splitting variable, and is accumulated over all the trees in the forest separately for each variable. The left plot of Figure 15.5 shows the variable importances computed in this way for the `spam` data; compare with the corresponding Figure 10.6 on page 354 for gradient boosting. Boosting ignores some variables completely, while the random forest does not. The candidate split-variable selection increases the chance that any single variable gets included in a random forest, while no such selection occurs with boosting.

Random forests also use the OOB samples to construct a different *variable-importance* measure, apparently to measure the prediction strength of each variable. When the $b$th tree is grown, the OOB samples are passed down the tree, and the prediction accuracy is recorded. Then the values for the $j$th variable are randomly permuted in the OOB samples, and the accuracy is again computed. The decrease in accuracy as a result of this permuting is averaged over all trees, and is used as a measure of the importance of variable $j$ in the random forest. These are expressed as a percent of the maximum in the right plot in Figure 15.5. Although the rankings of the two methods are similar, the importances in the right plot are more uniform over the variables. The randomization effectively voids the effect of a variable, much like setting a coefficient to zero in a linear model (Exercise 15.7). This does not measure the effect on prediction were this variable not available, because if the model was refitted without the variable, other variables could be used as surrogates.

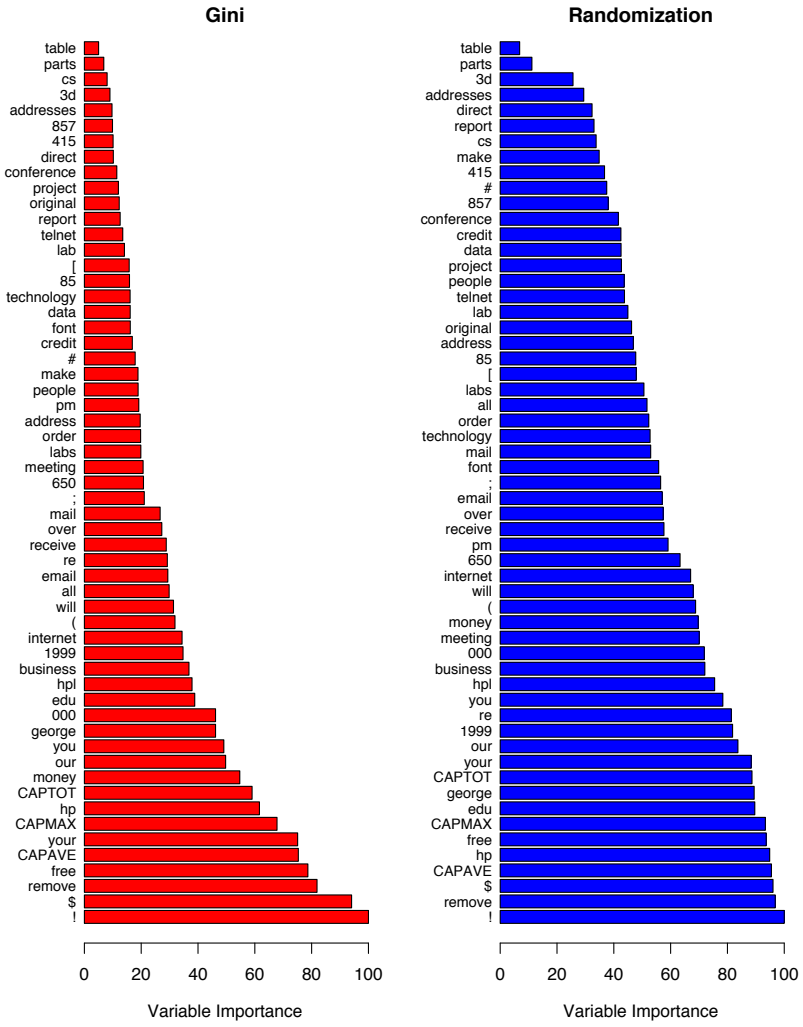



**FIGURE 15.5.** *Variable importance plots for a classification random forest grown on the* `spam` *data. The left plot bases the importance on the Gini splitting index, as in gradient boosting. The rankings compare well with the rankings produced by gradient boosting (Figure 10.6 on page 354). The right plot uses* OOB *randomization to compute variable importances, and tends to spread the importances more uniformly.*

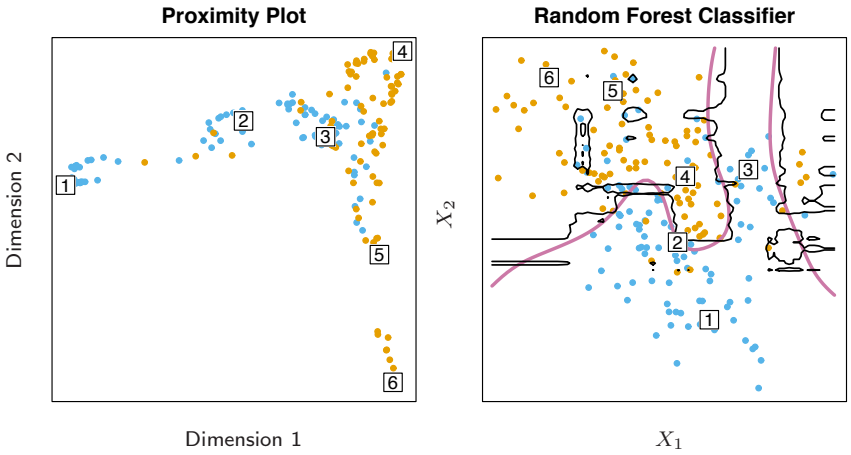



**FIGURE 15.6.** *(Left): Proximity plot for a random forest classifier grown to the mixture data. (Right): Decision boundary and training data for random forest on mixture data. Six points have been identified in each plot.*

### *15.3.3 Proximity Plots*

One of the advertised outputs of a random forest is a *proximity plot*. Figure 15.6 shows a proximity plot for the mixture data defined in Section 2.3.3 in Chapter 2. In growing a random forest, an $N \times N$ proximity matrix is accumulated for the training data. For every tree, any pair of OOB observations sharing a terminal node has their proximity increased by one. This proximity matrix is then represented in two dimensions using multidimensional scaling (Section 14.8). The idea is that even though the data may be high-dimensional, involving mixed variables, etc., the proximity plot gives an indication of which observations are effectively close together in the eyes of the random forest classifier.

Proximity plots for random forests often look very similar, irrespective of the data, which casts doubt on their utility. They tend to have a star shape, one arm per class, which is more pronounced the better the classification performance.

Since the mixture data are two-dimensional, we can map points from the proximity plot to the original coordinates, and get a better understanding of what they represent. It seems that points in pure regions class-wise map to the extremities of the star, while points nearer the decision boundaries map nearer the center. This is not surprising when we consider the construction of the proximity matrices. Neighboring points in pure regions will often end up sharing a bucket, since when a terminal node is pure, it is no longer

split by a random forest tree-growing algorithm. On the other hand, pairs of points that are close but belong to different classes will sometimes share a terminal node, but not always.

### *15.3.4 Random Forests and Overfitting*

When the number of variables is large, but the fraction of relevant variables small, random forests are likely to perform poorly with small $m$. At each split the chance can be small that the relevant variables will be selected. Figure 15.7 shows the results of a simulation that supports this claim. Details are given in the figure caption and Exercise 15.3. At the top of each pair we see the hyper-geometric probability that a relevant variable will be selected at any split by a random forest tree (in this simulation, the relevant variables are all equal in stature). As this probability gets small, the gap between boosting and random forests increases. When the number of relevant variables increases, the performance of random forests is surprisingly robust to an increase in the number of noise variables. For example, with 6 relevant and 100 noise variables, the probability of a relevant variable being selected at any split is 0.46, assuming $m = \sqrt{(6+100)} \approx 10$. According to Figure 15.7, this does not hurt the performance of random forests compared with boosting. This robustness is largely due to the relative insensitivity of misclassification cost to the bias and variance of the probability estimates in each tree. We consider random forests for regression in the next section.

Another claim is that random forests "cannot overfit" the data. It is certainly true that increasing $B$ does not cause the random forest sequence to overfit; like bagging, the random forest estimate (15.2) approximates the expectation

$$\hat{f}_{\mathrm{rf}}(x) = \mathrm{E}_{\Theta} T(x;\Theta) = \lim_{B\to\infty} \hat{f}(x)^{B}_{\mathrm{rf}} \tag{15.3}$$

with an average over $B$ realizations of $\Theta$. The distribution of $\Theta$ here is conditional on the training data. However, *this limit can overfit the data*; the average of fully grown trees can result in too rich a model, and incur unnecessary variance. Segal (2004) demonstrates small gains in performance by controlling the depths of the individual trees grown in random forests. Our experience is that using full-grown trees seldom costs much, and results in one less tuning parameter.

Figure 15.8 shows the modest effect of depth control in a simple regression example. Classifiers are less sensitive to variance, and this effect of overfitting is seldom seen with random-forest classification.

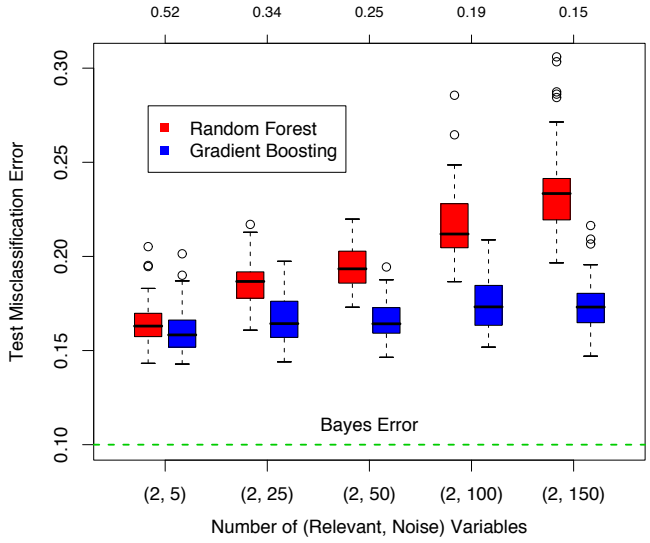



**FIGURE 15.7.** *A comparison of random forests and gradient boosting on problems with increasing numbers of noise variables. In each case the true decision boundary depends on two variables, and an increasing number of noise variables are included. Random forests uses its default value* $m = \sqrt{p}$. *At the top of each pair is the probability that one of the relevant variables is chosen at any split. The results are based on* 50 *simulations for each pair, with a training sample of* 300*, and a test sample of* 500.

## 15.4 Analysis of Random Forests

In this section we analyze the mechanisms at play with the additional randomization employed by random forests. For this discussion we focus on regression and squared error loss, since this gets at the main points, and bias and variance are more complex with 0–1 loss (see Section 7.3.1). Furthermore, even in the case of a classification problem, we can consider the random-forest average as an estimate of the class posterior probabilities, for which bias and variance are appropriate descriptors.

### *15.4.1 Variance and the De-Correlation Effect*

The limiting form ($B \to \infty$) of the random forest regression estimator is

$$\hat{f}_{\mathrm{rf}}(x) = \mathrm{E}_{\Theta|\mathbf{Z}} T(x; \Theta(\mathbf{Z})), \tag{15.4}$$

where we have made explicit the dependence on the training data $\mathbf{Z}$. Here we consider estimation at a single target point $x$. From (15.1) we see that

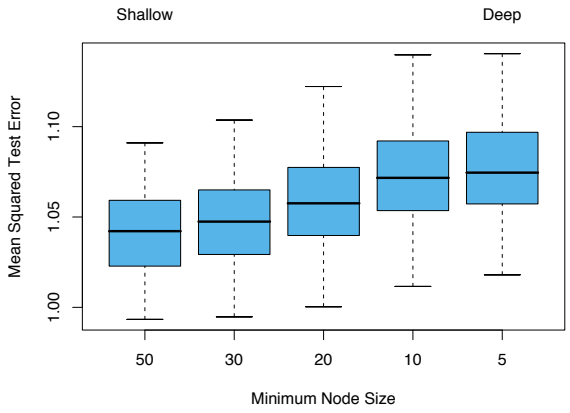



**FIGURE 15.8.** *The effect of* tree size *on the error in random forest regression. In this example, the true surface was additive in two of the* 12 *variables, plus additive unit-variance Gaussian noise. Tree depth is controlled here by the minimum node size; the smaller the minimum node size, the deeper the trees.*

$$\operatorname{Var}\hat{f}_{\mathrm{rf}}(x) = \rho(x)\sigma^2(x). \tag{15.5}$$

Here

- $\rho(x)$ is the *sampling* correlation between any pair of trees used in the averaging:
$$\rho(x) = \operatorname{corr}[T(x;\Theta_1(\mathbf{Z})), T(x;\Theta_2(\mathbf{Z}))], \tag{15.6}$$
where $\Theta_1(\mathbf{Z})$ and $\Theta_2(\mathbf{Z})$ are a randomly drawn pair of random forest trees grown to the randomly sampled $\mathbf{Z}$;
- $\sigma^2(x)$ is the sampling variance of any single randomly drawn tree,
$$\sigma^2(x) = \operatorname{Var} T(x;\Theta(\mathbf{Z})). \tag{15.7}$$

It is easy to confuse $\rho(x)$ with the average correlation between fitted trees in a *given* random-forest ensemble; that is, think of the fitted trees as $N$-vectors, and compute the average pairwise correlation between these vectors, conditioned on the data. This is *not* the case; this conditional correlation is not directly relevant in the averaging process, and the dependence on $x$ in $\rho(x)$ warns us of the distinction. Rather, $\rho(x)$ is the theoretical correlation between a pair of random-forest trees evaluated at $x$, induced by repeatedly making training sample draws $\mathbf{Z}$ from the population, and then drawing a pair of random forest trees. In statistical jargon, this is the correlation induced by the *sampling distribution* of $\mathbf{Z}$ and $\Theta$.

More precisely, the variability averaged over in the calculations in (15.6) and (15.7) is both

- conditional on **Z**: due to the bootstrap sampling and feature sampling at each split, and
- a result of the sampling variability of **Z** itself.

In fact, the conditional covariance of a pair of tree fits at $x$ is zero, because the bootstrap and feature sampling is i.i.d; see Exercise 15.5.

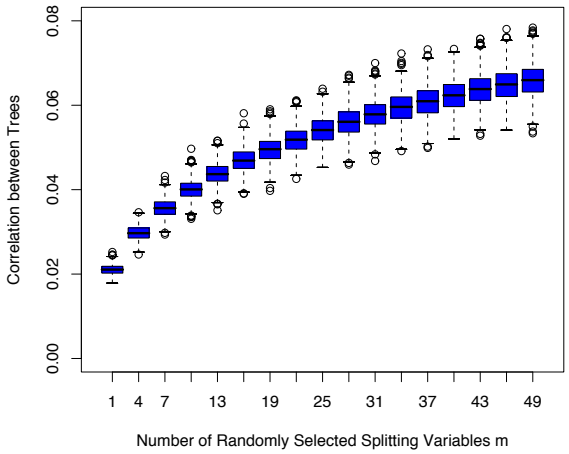



**FIGURE 15.9.** *Correlations between pairs of trees drawn by a random-forest regression algorithm, as a function of $m$. The boxplots represent the correlations at* 600 *randomly chosen prediction points $x$.*

The following demonstrations are based on a simulation model

$$Y = \frac{1}{\sqrt{50}} \sum_{j=1}^{50} X_j + \varepsilon, \tag{15.8}$$

with all the $X_j$ and $\varepsilon$ iid Gaussian. We use 500 training sets of size 100, and a single set of test locations of size 600. Since regression trees are nonlinear in **Z**, the patterns we see below will differ somewhat depending on the structure of the model.

Figure 15.9 shows how the correlation (15.6) between pairs of trees decreases as $m$ decreases: pairs of tree predictions at $x$ for different training sets **Z** are likely to be less similar if they do not use the same splitting variables.

In the left panel of Figure 15.10 we consider the variances of single tree predictors, $\mathrm{Var}T(x;\Theta(\mathbf{Z}))$ (averaged over 600 prediction points $x$ drawn randomly from our simulation model). This is the total variance, and can be

decomposed into two parts using standard conditional variance arguments (see Exercise 15.5):

$$\begin{aligned} \mathrm{Var}_{\Theta,\mathbf{Z}}T(x;\Theta(\mathbf{Z})) &= \mathrm{Var}_{\mathbf{Z}}\mathrm{E}_{\Theta|\mathbf{Z}}T(x;\Theta(\mathbf{Z})) + \mathrm{E}_{\mathbf{Z}}\mathrm{Var}_{\Theta|\mathbf{Z}}T(x;\Theta(\mathbf{Z})) \\ \textit{Total Variance} &= \mathrm{Var}_{\mathbf{Z}}\hat{f}_{\mathrm{rf}}(x) + \textit{within-}\mathbf{Z}\textit{ Variance} \end{aligned} \tag{15.9}$$

The second term is the within-$\mathbf{Z}$ variance—a result of the randomization, which increases as $m$ decreases. The first term is in fact the sampling variance of the random forest ensemble (shown in the right panel), which decreases as $m$ decreases. The variance of the individual trees does not change appreciably over much of the range of $m$, hence in light of (15.5), the variance of the ensemble is dramatically lower than this tree variance.

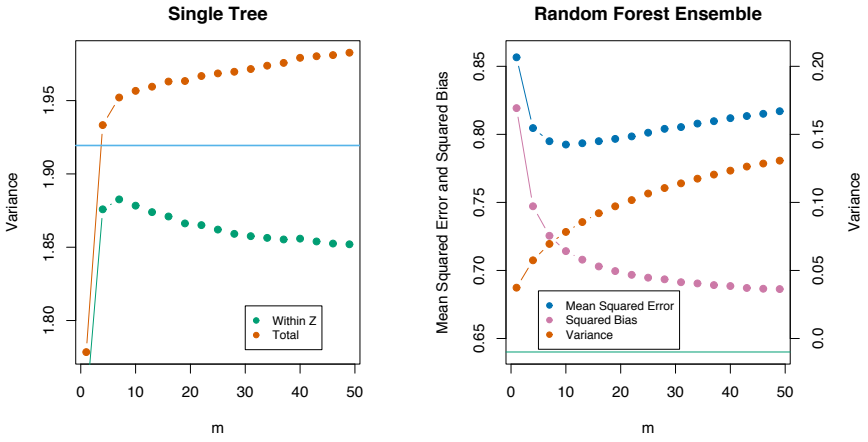



**FIGURE 15.10.** *Simulation results. The left panel shows the average variance of a single random forest tree, as a function of $m$. "Within $\mathbf{Z}$" refers to the average within-sample contribution to the variance, resulting from the bootstrap sampling and split-variable sampling (15.9). "Total" includes the sampling variability of $\mathbf{Z}$. The horizontal line is the average variance of a single fully grown tree (without bootstrap sampling). The right panel shows the average mean-squared error, squared bias and variance of the ensemble, as a function of $m$. Note that the variance axis is on the right (same scale, different level). The horizontal line is the average squared-bias of a fully grown tree.*

## *15.4.2 Bias*

As in bagging, the bias of a random forest is the same as the bias of any of the individual sampled trees $T(x;\Theta(\mathbf{Z}))$:

$$\begin{aligned} \text{Bias}(x) &= \mu(x) - \text{E}_{\mathbf{Z}}\hat{f}_{\text{rf}}(x) \\ &= \mu(x) - \text{E}_{\mathbf{Z}}\text{E}_{\Theta|\mathbf{Z}}T(x;\Theta(\mathbf{Z})). \end{aligned} \tag{15.10}$$

This is also typically greater (in absolute terms) than the bias of an unpruned tree grown to $\mathbf{Z}$, since the randomization and reduced sample space impose restrictions. Hence the improvements in prediction obtained by bagging or random forests are *solely a result of variance reduction*.

Any discussion of bias depends on the unknown true function. Figure 15.10 (right panel) shows the squared bias for our additive model simulation (estimated from the 500 realizations). Although for different models the shape and rate of the bias curves may differ, the general trend is that as $m$ decreases, the bias increases. Shown in the figure is the mean-squared error, and we see a classical bias-variance trade-off in the choice of $m$. For all $m$ the squared bias of the random forest is greater than that for a single tree (horizontal line).

These patterns suggest a similarity with ridge regression (Section 3.4.1). Ridge regression is useful (in linear models) when one has a large number of variables with similarly sized coefficients; ridge shrinks their coefficients toward zero, and those of strongly correlated variables toward each other. Although the size of the training sample might not permit all the variables to be in the model, this regularization via ridge stabilizes the model and allows all the variables to have their say (albeit diminished). Random forests with small $m$ perform a similar averaging. Each of the relevant variables get their turn to be the primary split, and the ensemble averaging reduces the contribution of any individual variable. Since this simulation example (15.8) is based on a linear model in all the variables, ridge regression achieves a lower mean-squared error (about 0.45 with $\text{df}(\lambda_{\text{opt}}) \approx 29$).

### *15.4.3 Adaptive Nearest Neighbors*

The random forest classifier has much in common with the $k$-nearest neighbor classifier (Section 13.3); in fact a weighted version thereof. Since each tree is grown to maximal size, for a particular $\Theta^*$, $T(x;\Theta^*(\mathbf{Z}))$ is the response value for one of the training samples[4]. The tree-growing algorithm finds an "optimal" path to that observation, choosing the most informative predictors from those at its disposal. The averaging process assigns weights to these training responses, which ultimately vote for the prediction. Hence via the random-forest voting mechanism, those observations *close* to the target point get assigned weights—an equivalent kernel—which combine to form the classification decision.

Figure 15.11 demonstrates the similarity between the decision boundary of 3-nearest neighbors and random forests on the mixture data.

---

[4]We gloss over the fact that pure nodes are not split further, and hence there can be more than one observation in a terminal node

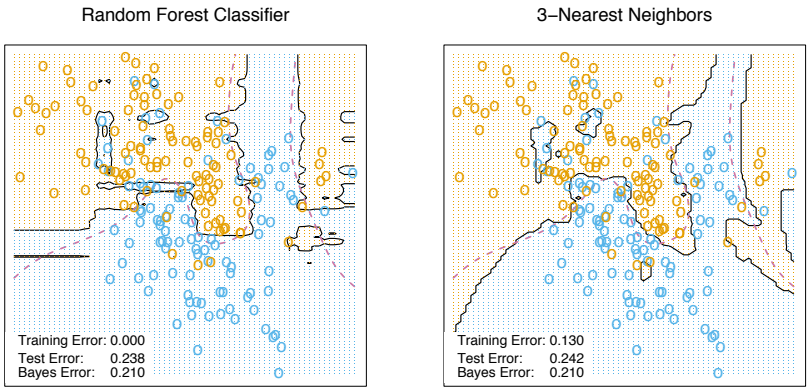



**FIGURE 15.11.** *Random forests versus 3-NN on the mixture data. The axis-oriented nature of the individual trees in a random forest lead to decision regions with an axis-oriented flavor.*

## Bibliographic Notes

Random forests as described here were introduced by Breiman (2001), although many of the ideas had cropped up earlier in the literature in different forms. Notably Ho (1995) introduced the term "random forest," and used a consensus of trees grown in random subspaces of the features. The idea of using stochastic perturbation and averaging to avoid overfitting was introduced by Kleinberg (1990), and later in Kleinberg (1996). Amit and Geman (1997) used randomized trees grown on image features for image classification problems. Breiman (1996a) introduced bagging, a precursor to his version of random forests. Dietterich (2000b) also proposed an improvement on bagging using additional randomization. His approach was to rank the top 20 candidate splits at each node, and then select from the list at random. He showed through simulations and real examples that this additional randomization improved over the performance of bagging. Friedman and Hall (2007) showed that sub-sampling (without replacement) is an effective alternative to bagging. They showed that growing and averaging trees on samples of size $N/2$ is approximately equivalent (in terms bias/variance considerations) to bagging, while using smaller fractions of $N$ reduces the variance even further (through decorrelation).

There are several free software implementations of random forests. In this chapter we used the `randomForest` package in R, maintained by Andy Liaw, available from the `CRAN` website. This allows both split-variable selection, as well as sub-sampling. Adele Cutler maintains a random forest website `http://www.math.usu.edu/~adele/forests/` where (as of August 2008) the software written by Leo Breiman and Adele Cutler is freely

available. Their code, and the name "random forests", is exclusively licensed to Salford Systems for commercial release. The `Weka` machine learning archive `http://www.cs.waikato.ac.nz/ml/weka/` at Waikato University, New Zealand, offers a free `java` implementation of random forests.

## Exercises

Ex. 15.1 Derive the variance formula (15.1). This appears to fail if $\rho$ is negative; diagnose the problem in this case.

Ex. 15.2 Show that as the number of bootstrap samples $B$ gets large, the OOB error estimate for a random forest approaches its $N$-fold CV error estimate, and that in the limit, the identity is exact.

Ex. 15.3 Consider the simulation model used in Figure 15.7 (Mease and Wyner, 2008). Binary observations are generated with probabilities

$$\Pr(Y = 1|X) = q + (1 - 2q) \cdot 1\left[\sum_{j=1}^{J} X_j > J/2\right], \tag{15.11}$$

where $X \sim U[0, 1]^p$, $0 \leq q \leq \frac{1}{2}$, and $J \leq p$ is some predefined (even) number. Describe this probability surface, and give the Bayes error rate.

Ex. 15.4 Suppose $x_i$, $i = 1, \ldots, N$ are iid $(\mu, \sigma^2)$. Let $\bar{x}_1^*$ and $\bar{x}_2^*$ be two bootstrap realizations of the sample mean. Show that the sampling correlation $\text{corr}(\bar{x}_1^*, \bar{x}_2^*) = \frac{n}{2n-1} \approx 50\%$. Along the way, derive $\text{var}(\bar{x}_1^*)$ and the variance of the bagged mean $\bar{x}_{bag}$. Here $\bar{x}$ is a *linear* statistic; bagging produces no reduction in variance for linear statistics.

Ex. 15.5 Show that the sampling correlation between a pair of random-forest trees at a point $x$ is given by

$$\rho(x) = \frac{\text{Var}_{\mathbf{Z}}[\text{E}_{\Theta|\mathbf{Z}} T(x; \Theta(\mathbf{Z}))]}{\text{Var}_{\mathbf{Z}}[\text{E}_{\Theta|\mathbf{Z}} T(x; \Theta(\mathbf{Z}))] + \text{E}_{\mathbf{Z}}\text{Var}_{\Theta|\mathbf{Z}}[T(x, \Theta(\mathbf{Z})]}. \tag{15.12}$$

The term in the numerator is $\text{Var}_{\mathbf{Z}}[\hat{f}_{\text{rf}}(x)]$, and the second term in the denominator is the expected conditional variance due to the randomization in random forests.

Ex. 15.6 Fit a series of random-forest classifiers to the `spam` data, to explore the sensitivity to the parameter $m$. Plot both the OOB error as well as the test error against a suitably chosen range of values for $m$.

Ex. 15.7 Suppose we fit a linear regression model to $N$ observations with response $y_i$ and predictors $x_{i1}, \ldots, x_{ip}$. Assume that all variables are standardized to have mean zero and standard deviation one. Let $RSS$ be the mean-squared residual on the training data, and $\hat{\beta}$ the estimated coefficient. Denote by $RSS_j^*$ the mean-squared residual on the training data using the same $\hat{\beta}$, but with the $N$ values for the $j$th variable randomly permuted before the predictions are calculated. Show that

$$\mathrm{E}_P[RSS_j^* - RSS] = 2\hat{\beta}_j^2, \tag{15.13}$$

where $\mathrm{E}_P$ denotes expectation with respect to the permutation distribution. Argue that this is approximately true when the evaluations are done using an independent test set.

# Spanner: Google's Globally Distributed Database

JAMES C. CORBETT, JEFFREY DEAN, MICHAEL EPSTEIN, ANDREW FIKES, CHRISTOPHER FROST, J. J. FURMAN, SANJAY GHEMAWAT, ANDREY GUBAREV, CHRISTOPHER HEISER, PETER HOCHSCHILD, WILSON HSIEH, SEBASTIAN KANTHAK, EUGENE KOGAN, HONGYI LI, ALEXANDER LLOYD, SERGEY MELNIK, DAVID MWAURA, DAVID NAGLE, SEAN QUINLAN, RAJESH RAO, LINDSAY ROLIG, YASUSHI SAITO, MICHAL SZYMANIAK, CHRISTOPHER TAYLOR, RUTH WANG, and DALE WOODFORD, Google, Inc.



Spanner is Google's scalable, multiversion, globally distributed, and synchronously replicated database. It is the first system to distribute data at global scale and support externally-consistent distributed transactions. This article describes how Spanner is structured, its feature set, the rationale underlying various design decisions, and a novel time API that exposes clock uncertainty. This API and its implementation are critical to supporting external consistency and a variety of powerful features: nonblocking reads in the past, lock-free snapshot transactions, and atomic schema changes, across all of Spanner.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Concurrency, distributed databases, transaction processing*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Distributed databases, concurrency control, replication, transactions, time management

**ACM Reference Format:**
Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. 2013. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (August 2013), 22 pages.
DOI: http://dx.doi.org/10.1145/2491245

## 1. INTRODUCTION

Spanner is a scalable, globally distributed database designed, built, and deployed at Google. At the highest level of abstraction, it is a database that shards data across many sets of Paxos [Lamport 1998] state machines in datacenters spread all over the world. Replication is used for global availability and geographic locality; clients automatically failover between replicas. Spanner automatically reshards data across machines as the amount of data or the number of servers changes, and it automatically migrates data across machines (even across datacenters) to balance load and in

---

This article is essentially the same (with minor reorganizations, corrections, and additions) as the paper of the same title that appeared in the Proceedings of OSDI 2012.
Authors' address: J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh (corresponding author), S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, Google, Inc. 1600 Amphitheatre Parkway, Mountain View, CA 94043; email: wilsonh@google.com.

0734-2071/2013/08-ART8
DOI: http://dx.doi.org/10.1145/2491245

response to failures. Spanner is designed to scale up to millions of machines across hundreds of datacenters and trillions of database rows.

Applications can use Spanner for high availability, even in the face of wide-area natural disasters, by replicating their data within or even across continents. Our initial customer was F1 [Shute et al. 2012], a rewrite of Google's advertising backend. F1 uses five replicas spread across the United States. Most other applications will probably replicate their data across 3 to 5 datacenters in one geographic region, but with relatively independent failure modes. That is, most applications will choose lower latency over higher availability, as long as they can survive 1 or 2 datacenter failures.

Spanner's main focus is managing cross-datacenter replicated data, but we have also spent a great deal of time in designing and implementing important database features on top of our distributed-systems infrastructure. Even though many projects happily use Bigtable [Chang et al. 2008], we have also consistently received complaints from users that Bigtable can be difficult to use for some kinds of applications: those that have complex, evolving schemas, or those that want strong consistency in the presence of wide-area replication. (Similar claims have been made by other authors [Stonebraker 2010b].) Many applications at Google have chosen to use Megastore [Baker et al. 2011] because of its semirelational data model and support for synchronous replication, despite its relatively poor write throughput. As a consequence, Spanner has evolved from a Bigtable-like versioned key-value store into a temporal multiversion database. Data is stored in schematized semirelational tables; data is versioned, and each version is automatically timestamped with its commit time; old versions of data are subject to configurable garbage-collection policies; and applications can read data at old timestamps. Spanner supports general-purpose transactions, and provides an SQL-based query language.

As a globally distributed database, Spanner provides several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage across datacenters. Second, Spanner has two features that are difficult to implement in a distributed database: it provides externally consistent [Gifford 1982] reads and writes, and globally consistent reads across the database at a timestamp. These features enable Spanner to support consistent backups, consistent MapReduce executions [Dean and Ghemawat 2010], and atomic schema updates, all at global scale, and even in the presence of ongoing transactions.

These features are enabled by the fact that Spanner assigns globally meaningful commit timestamps to transactions, even though transactions may be distributed. The timestamps reflect serialization order. In addition, the serialization order satisfies external consistency (or equivalently, linearizability [Herlihy and Wing 1990]): if a transaction $T_1$ commits before another transaction $T_2$ starts, then $T_1$'s commit timestamp is smaller than $T_2$'s. Spanner is the first system to provide such guarantees at global scale.

The key enabler of these properties is a new TrueTime API and its implementation. The API directly exposes clock uncertainty, and the guarantees on Spanner's timestamps depend on the bounds that the implementation provides. If the uncertainty is large, Spanner slows down to wait out that uncertainty. Google's cluster-management software provides an implementation of the TrueTime API. This implementation keeps uncertainty small (generally less than 10ms) by using multiple modern clock references (GPS and atomic clocks). Conservatively reporting

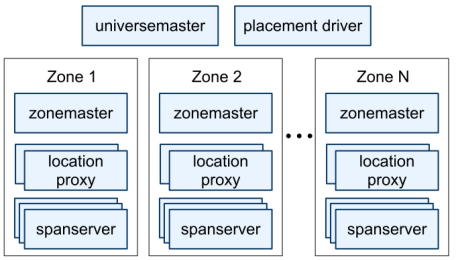



Fig. 1. Spanner server organization.

uncertainty is necessary for correctness; keeping the bound on uncertainty small is necessary for performance.

Section 2 describes the structure of Spanner's implementation, its feature set, and the engineering decisions that went into their design. Section 3 describes our new TrueTime API and sketches its implementation. Section 4 describes how Spanner uses TrueTime to implement externally-consistent distributed transactions, lock-free snapshot transactions, and atomic schema updates. Section 5 provides some benchmarks on Spanner's performance and TrueTime behavior, and discusses the experiences of F1. Sections 6, 7, and 8 describe related and future work, and summarize our conclusions.

## 2. IMPLEMENTATION

This section describes the structure of and rationale underlying Spanner's implementation. It then describes the *directory* abstraction, which is used to manage replication and locality, and is the unit of data movement. Finally, it describes our data model, why Spanner looks like a relational database instead of a key-value store, and how applications can control data locality.

A Spanner deployment is called a *universe*. Given that Spanner manages data globally, there will be only a handful of running universes. We currently run a test/playground universe, a development/production universe, and a production-only universe.

Spanner is organized as a set of *zones*, where each zone is the rough analog of a deployment of Bigtable servers [Chang et al. 2008]. Zones are the unit of administrative deployment. The set of zones is also the set of locations across which data can be replicated. Zones can be added to or removed from a running system as new datacenters are brought into service and old ones are turned off, respectively. Zones are also the unit of physical isolation: there may be one or more zones in a datacenter, for example, if different applications' data must be partitioned across different sets of servers in the same datacenter.

Figure 1 illustrates the servers in a Spanner universe. A zone has one *zonemaster* and between one hundred and several thousand *spanservers*. The former assigns data to spanservers; the latter serve data to clients. The per-zone *location proxies* are used by clients to locate the spanservers assigned to serve their data. The *universe master* and the *placement driver* are currently singletons. The universe master is primarily a console that displays status information about all the zones for interactive debugging. The placement driver handles automated movement of data across zones on the timescale of minutes. The placement driver periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load. For space reasons, we will only describe the spanserver in any detail.

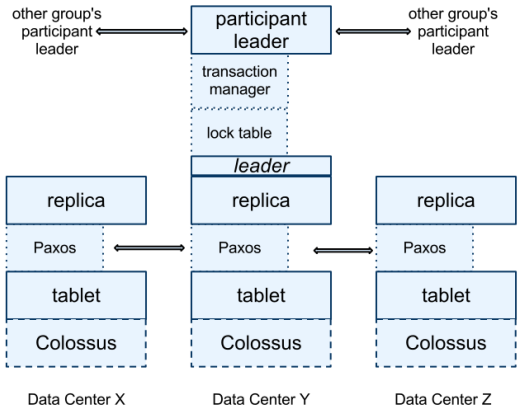



Fig. 2. Spanserver software stack.

### 2.1. Spanserver Software Stack

This section focuses on the spanserver implementation to illustrate how replication and distributed transactions have been layered onto our Bigtable-based implementation. The software stack is shown in Figure 2. At the bottom, each spanserver is responsible for between 100 and 1000 instances of a data structure called a *tablet*. A tablet is similar to Bigtable's tablet abstraction, in that it implements a bag of the following mappings.

$$(\text{key:string, timestamp:int64}) \rightarrow \text{string}$$

Unlike Bigtable, Spanner assigns timestamps to data, which is an important way in which Spanner is more like a multiversion database than a key-value store. A tablet's state is stored in a set of B-tree-like files and a write-ahead log, all on a distributed file system called Colossus (the successor to the Google File System [Ghemawat et al. 2003]).

To support replication, each spanserver implements a single Paxos state machine on top of each tablet. (An early Spanner incarnation supported multiple Paxos state machines per tablet, which allowed for more flexible replication configurations. The complexity of that design led us to abandon it.) Each state machine stores its metadata and log in its corresponding tablet. Our Paxos implementation supports long-lived leaders with time-based leader leases, whose length defaults to 10 seconds. The current Spanner implementation logs every Paxos write twice: once in the tablet's log, and once in the Paxos log. This choice was made out of expediency, and we are likely to remedy this eventually. Our implementation of Paxos is pipelined, so as to improve Spanner's throughput in the presence of WAN latencies. By "pipelined," we mean Lamport's "multi-decree parliament" [Lamport 1998], which both amortizes the cost of electing a leader across multiple decrees and allows for concurrent voting on different decrees. It is important to note that although decrees may be approved out of order, the decrees are applied in order (a fact on which we will depend in Section 4).

The Paxos state machines are used to implement a consistently replicated bag of mappings. The key-value mapping state of each replica is stored in its corresponding tablet. Writes must initiate the Paxos protocol at the leader; reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date. The set of replicas is collectively a *Paxos group*.

At every replica that is a leader, a spanserver implements a *lock table* to implement concurrency control. The lock table contains the state for two-phase locking: it maps

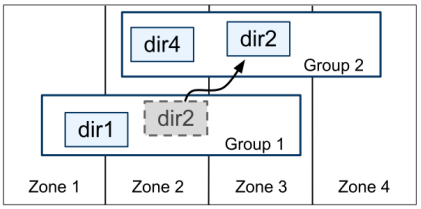



Fig. 3. Directories are the unit of data movement between Paxos groups.

ranges of keys to lock states. (Note that having a long-lived Paxos leader is critical to efficiently managing the lock table.) In both Bigtable and Spanner, we designed for long-lived transactions (for example, for report generation, which might take on the order of minutes), which perform poorly under optimistic concurrency control in the presence of conflicts. Operations that require synchronization, such as transactional reads, acquire locks in the lock table; other operations bypass the lock table. The state of the lock table is mostly volatile (i.e., not replicated via Paxos): we explain the details further in Section 4.2.1.

At every replica that is a leader, each spanserver also implements a *transaction manager* to support distributed transactions. The transaction manager is used to implement a *participant leader*; the other replicas in the group will be referred to as *participant slaves*. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since the lock table and Paxos together provide transactionality. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two-phase commit. One of the participant groups is chosen as the coordinator: the participant leader of that group will be referred to as the *coordinator leader*, and the slaves of that group as *coordinator slaves*. The state of each transaction manager is stored in the underlying Paxos group (and therefore is replicated).

### 2.2. Directories and Placement

On top of the bag of key-value mappings, the Spanner implementation supports a bucketing abstraction called a *directory*, which is a set of contiguous keys that share a common prefix. (The choice of the term *directory* is a historical accident; a better term might be *bucket*.) We will explain the source of that prefix in Section 2.3. Supporting directories allows applications to control the locality of their data by choosing keys carefully.

A directory is the unit of data placement. All data in a directory has the same replication configuration. When data is moved between Paxos groups, it is moved directory by directory, as shown in Figure 3. Spanner might move a directory to shed load from a Paxos group; to put directories that are frequently accessed together into the same group; or to move a directory into a group that is closer to its accessors. Directories can be moved while client operations are ongoing. One would expect that a 50MB directory could be moved in a few seconds.

The fact that a Paxos group may contain multiple directories implies that a Spanner tablet is different from a Bigtable tablet: the former is not necessarily a single lexicographically contiguous partition of the row space. Instead, a Spanner tablet is a container that may encapsulate multiple partitions of the row space. We made this decision so that it would be possible to colocate multiple directories that are frequently accessed together.

*Movedir* is the background task used to move directories between Paxos groups [Douceur and Howell 2006]. Movedir is also used to add to or remove replicas

from Paxos groups [Lorch et al. 2006] by moving all of a group's data to a new group with the desired configuration, because Spanner does not yet support in-Paxos configuration changes. Movedir is not implemented as a single transaction, so as to avoid blocking ongoing reads and writes on a bulky data move. Instead, movedir registers the fact that it is starting to move data and moves the data in the background. When it has moved all but a nominal amount of the data, it uses a transaction to atomically move that nominal amount and update the metadata for the two Paxos groups.

A directory is also the smallest unit whose geographic-replication properties (or *placement*, for short) can be specified by an application. The design of our placement-specification language separates responsibilities for managing replication configurations. Administrators control two dimensions: the number and types of replicas, and the geographic placement of those replicas. They create a menu of named options in these two dimensions (e.g., *North America, replicated 5 ways with 1 witness*). An application controls how data is replicated, by tagging each database and/or individual directories with a combination of those options. For example, an application might store each end-user's data in its own directory, which would enable user *A*'s data to have three replicas in Europe, and user *B*'s data to have five replicas in North America.

For expository clarity we have oversimplified. In fact, Spanner will shard a directory into multiple *fragments* if it grows too large. Fragments may be served from different Paxos groups (and therefore different servers). Movedir actually moves fragments, and not whole directories, between groups.

### 2.3. Data Model

Spanner exposes the following set of data features to applications: a data model based on schematized semirelational tables, a query language, and general-purpose transactions. The move towards supporting these features was driven by many factors. The need to support schematized semirelational tables and synchronous replication is supported by the popularity of Megastore [Baker et al. 2011]. At least 300 applications within Google use Megastore (despite its relatively low performance) because its data model is simpler to manage than Bigtable's, and because of its support for synchronous replication across datacenters. (Bigtable only supports eventually-consistent replication across datacenters.) Examples of well-known Google applications that use Megastore are Gmail, Picasa, Calendar, Android Market, and AppEngine. The need to support an SQL-like query language in Spanner was also clear, given the popularity of Dremel [Melnik et al. 2010] as an interactive data-analysis tool. Finally, the lack of cross-row transactions in Bigtable led to frequent complaints; Percolator [Peng and Dabek 2010] was in part built to address this failing. Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings [Chang et al. 2008; Cooper et al. 2008; Helland 2007]. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. Running two-phase commit over Paxos mitigates the availability problems.

The application data model is layered on top of the directory-bucketed key-value mappings supported by the implementation. An application creates one or more databases in a universe. Each database can contain an unlimited number of schematized tables. Tables look like relational-database tables, with rows, columns, and versioned values. We will not go into detail about the query language for Spanner. It looks like SQL with some extensions to support protocol-buffer-valued [Google 2008] fields.

Spanner's data model is not purely relational, in that rows must have names. More precisely, every table is required to have an ordered set of one or more primary-key columns. This requirement is where Spanner still looks like a key-value store: the

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

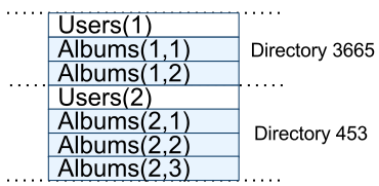



Fig. 4. Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

Table I. TrueTime API. The Argument $t$ is of Type *TTstamp*

| Method | Returns |
|---|---|
| $TT.now()$ | *TTinterval*: $[earliest, latest]$ |
| $TT.after(t)$ | true if $t$ has definitely passed |
| $TT.before(t)$ | true if $t$ has definitely not arrived |

primary keys form the name for a row, and each table defines a mapping from the primary-key columns to the non-primary-key columns. A row has existence only if some value (even if it is NULL) is defined for the row's keys. Imposing this structure is useful because it lets applications control data locality through their choices of keys.

Figure 4 contains an example Spanner schema for storing photo metadata on a per-user, per-album basis. The schema language is similar to Megastore's, with the additional requirement that every Spanner database must be partitioned by clients into one or more hierarchies of tables. Client applications declare the hierarchies in database schemas via the `INTERLEAVE IN` declarations. The table at the top of a hierarchy is a *directory table*. Each row in a directory table with key $K$, together with all of the rows in descendant tables that start with $K$ in lexicographic order, forms a directory. `ON DELETE CASCADE` says that deleting a row in the directory table deletes any associated child rows. The figure also illustrates the interleaved layout for the example database: for example, `Albums(2,1)` represents the row from the `Albums` table for `user_id 2, album_id 1`. This interleaving of tables to form directories is significant because it allows clients to describe the locality relationships that exist between multiple tables, which is necessary for good performance in a sharded, distributed database. Without it, Spanner would not know the most important locality relationships.

## 3. TRUETIME

This section describes the TrueTime API and sketches its implementation. We leave most of the details for another article: our goal is to demonstrate the power of having such an API. Table I lists the methods of the API. TrueTime explicitly represents time as a *TTinterval*, which is an interval with bounded time uncertainty (unlike standard time interfaces that give clients no notion of uncertainty). The endpoints of a *TTinterval* are of type *TTstamp*. The $TT.now()$ method returns a *TTinterval* that is guaranteed to contain the absolute time during which $TT.now()$ was invoked. The time epoch is analogous to UNIX time with leap-second smearing. Define the instantaneous

error bound as $\epsilon$, which is half of the interval's width, and the average error bound as $\bar{\epsilon}$. The *TT.after*() and *TT.before*() methods are convenience wrappers around *TT.now*().

Denote the absolute time of an event $e$ by the function $t_{abs}(e)$. In more formal terms, TrueTime guarantees that for an invocation $tt = TT.now()$, $tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$, where $e_{now}$ is the invocation event.

The underlying time references used by TrueTime are GPS and atomic clocks. TrueTime uses two forms of time reference because they have different failure modes. GPS reference-source vulnerabilities include antenna and receiver failures, local radio interference, correlated failures (e.g., design faults such as incorrect leap-second handling and spoofing), and GPS system outages. Atomic clocks can fail in ways uncorrelated to GPS and each other, and over long periods of time can drift significantly due to frequency error.

TrueTime is implemented by a set of *time master* machines per datacenter and a *timeslave daemon* per machine. The majority of masters have GPS receivers with dedicated antennas; these masters are separated physically to reduce the effects of antenna failures, radio interference, and spoofing. The remaining masters (which we refer to as *Armageddon masters*) are equipped with atomic clocks. An atomic clock is not that expensive: the cost of an Armageddon master is of the same order as that of a GPS master. All masters' time references are regularly compared against each other. Each master also cross-checks the rate at which its reference advances time against its own local clock, and evicts itself if there is substantial divergence. Between synchronizations, Armageddon masters advertise a slowly increasing time uncertainty that is derived from conservatively applied worst-case clock drift. GPS masters advertise uncertainty that is typically close to zero.

Every daemon polls a variety of masters [Mills 1981] to reduce vulnerability to errors from any one master. Some are GPS masters chosen from nearby datacenters; the rest are GPS masters from farther datacenters, as well as some Armageddon masters. Daemons apply a variant of Marzullo's algorithm [Marzullo and Owicki 1983] to detect and reject liars, and synchronize the local machine clocks to the non-liars. To protect against broken local clocks, machines that exhibit frequency excursions larger than the worst-case bound derived from component specifications and operating environment are evicted. Correctness depends on ensuring that the worst-case bound is enforced.

Between synchronizations, a daemon advertises a slowly increasing time uncertainty. $\epsilon$ is derived from conservatively applied worst-case local clock drift. $\epsilon$ also depends on time-master uncertainty and communication delay to the time masters. In our production environment, $\epsilon$ is typically a sawtooth function of time, varying from about 1 to 7 ms over each poll interval. $\bar{\epsilon}$ is therefore 4 ms most of the time. The daemon's poll interval is currently 30 seconds, and the current applied drift rate is set at 200 microseconds/second, which together account for the sawtooth bounds from 0 to 6 ms. The remaining 1 ms comes from the communication delay to the time masters. Excursions from this sawtooth are possible in the presence of failures. For example, occasional time-master unavailability can cause datacenter-wide increases in $\epsilon$. Similarly, overloaded machines and network links can result in occasional localized $\epsilon$ spikes. Correctness is not affected by $\epsilon$ variance because Spanner can wait out the uncertainty, but performance can degrade if $\epsilon$ increases too much.

## 4. CONCURRENCY CONTROL

This section describes how TrueTime is used to guarantee the correctness properties around concurrency control, and how those properties are used to implement features such as externally consistent transactions, lock-free snapshot transactions, and nonblocking reads in the past. These features enable, for example, the guarantee that

Table II. Types of Reads and Writes in Spanner, and How They Compare

| Operation | Timestamp Discussion | Concurrency Control | Replica Required |
|---|---|---|---|
| Read-Write Transaction | § 4.1.2 | pessimistic | leader |
| Snapshot Transaction | § 4.1.4 | lock-free | leader for timestamp; any for read, subject to § 4.1.3 |
| Snapshot Read, client-chosen timestamp | — | lock-free | any, subject to § 4.1.3 |
| Snapshot Read, client-chosen bound | § 4.1.3 | lock-free | any, subject to § 4.1.3 |

a whole-database audit read at a timestamp $t$ will see exactly the effects of every transaction that has committed as of $t$.

Going forward, it will be important to distinguish writes as seen by Paxos (which we will refer to as *Paxos writes* unless the context is clear) from Spanner client writes. For example, two-phase commit generates a Paxos write for the prepare phase that has no corresponding Spanner client write.

### 4.1. Timestamp Management

Table II lists the types of operations that Spanner supports. The Spanner implementation supports *read-write transactions*, *snapshot transactions* (predeclared snapshot-isolation transactions), and *snapshot reads*. Standalone writes are implemented as read-write transactions; non-snapshot standalone reads are implemented as snapshot transactions. Both are internally retried (clients need not write their own retry loops).

A snapshot transaction is a kind of transaction that has the performance benefits of snapshot isolation [Berenson et al. 1995]. A snapshot transaction must be predeclared as not having any writes; it is not simply a read-write transaction without any writes. Reads in a snapshot transaction execute at a system-chosen timestamp without locking, so that incoming writes are not blocked. The execution of the reads in a snapshot transaction can proceed on any replica that is sufficiently up-to-date (Section 4.1.3).

A snapshot read is a read in the past that executes without locking. A client can either specify a timestamp for a snapshot read, or provide an upper bound on the desired timestamp's staleness and let Spanner choose a timestamp. In either case, the execution of a snapshot read proceeds at any replica that is sufficiently up-to-date.

For both snapshot transactions and snapshot reads, commit is inevitable once a timestamp has been chosen, unless the data at that timestamp has been garbage-collected. As a result, clients can avoid buffering results inside a retry loop. When a server fails, clients can internally continue the query on a different server by repeating the timestamp and the current read position.

*4.1.1. Paxos Leader Leases.* Spanner's Paxos implementation uses timed leases to make leadership long-lived (10 seconds by default). A potential leader sends requests for timed *lease votes*; upon receiving a quorum of lease votes the leader knows it has a lease. A replica extends its lease vote implicitly on a successful write, and the leader requests lease-vote extensions if they are near expiration. Define a leader's *lease interval* to start when it discovers it has a quorum of lease votes, and to end when it no longer has a quorum of lease votes (because some have expired). Spanner depends on the following disjointness invariant: for each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's. Section 4.2.5 describes how this invariant is enforced.

The Spanner implementation permits a Paxos leader to abdicate by releasing its slaves from their lease votes. To preserve the disjointness invariant, Spanner

constrains when abdication is permissible. Define $s_{max}$ to be the maximum timestamp used by a leader. Subsequent sections will describe when $s_{max}$ is advanced. Before abdicating, a leader must wait until $TT.after(s_{max})$ is true.

*4.1.2. Assigning Timestamps to RW Transactions.* Transactional reads and writes use strict two-phase locking. As a result, they can be assigned timestamps at any time when all locks have been acquired, but before any locks have been released. For a given transaction, Spanner assigns it the timestamp that Paxos assigns to the Paxos write that represents the transaction commit.

Spanner depends on the following monotonicity invariant: within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders. A single leader replica can trivially assign timestamps in monotonically increasing order. This invariant is enforced across leaders by making use of the disjointness invariant: a leader must only assign timestamps within the interval of its leader lease. Note that whenever a timestamp $s$ is assigned, $s_{max}$ is advanced to $s$ to preserve disjointness.

Spanner also enforces the following external-consistency invariant: if the start of a transaction $T_2$ occurs after the commit of a transaction $T_1$, then the commit timestamp of $T_2$ must be greater than the commit timestamp of $T_1$. Define the start and commit events for a transaction $T_i$ by $e_i^{start}$ and $e_i^{commit}$; and the commit timestamp of a transaction $T_i$ by $s_i$. The invariant becomes $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$. The protocol for executing transactions and assigning timestamps obeys two rules, which together guarantee this invariant, as shown in the following. Define the arrival event of the commit request at the coordinator leader for a write $T_i$ to be $e_i^{server}$.

*Start.* The coordinator leader for a write $T_i$ assigns a commit timestamp $s_i$ no less than the value of $TT.now().latest$, computed after $e_i^{server}$. Note that the participant leaders do not matter here; Section 4.2.1 describes how they are involved in the implementation of the next rule.

*Commit Wait.* The coordinator leader ensures that clients cannot see any data committed by $T_i$ until $TT.after(s_i)$ is true. Commit wait ensures that $s_i$ is less than the absolute commit time of $T_i$, or $s_i < t_{abs}(e_i^{commit})$. The implementation of commit wait is described in Section 4.2.1. Proof:

$$
\begin{aligned}
s_1 &< t_{abs}(e_1^{commit}) && \text{(commit wait)}\\
t_{abs}(e_1^{commit}) &< t_{abs}(e_2^{start}) && \text{(assumption)}\\
t_{abs}(e_2^{start}) &\le t_{abs}(e_2^{server}) && \text{(causality)}\\
t_{abs}(e_2^{server}) &\le s_2 && \text{(start)}\\
s_1 &< s_2 && \text{(transitivity)}
\end{aligned}
$$

*4.1.3. Serving Reads at a Timestamp.* The monotonicity invariant described in Section 4.1.2 allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read. Every replica tracks a value called *safe time*, $t_{safe}$, which is the maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at a timestamp $t$ if $t <= t_{safe}$.

Define $t_{safe} = min(t_{safe}^{Paxos}, t_{safe}^{TM})$, where each Paxos state machine has a safe time $t_{safe}^{Paxos}$ and each transaction manager has a safe time $t_{safe}^{TM}$. $t_{safe}^{Paxos}$ is simpler: it is the timestamp of the highest-applied Paxos write. Because timestamps increase monotonically and

writes are applied in order, writes will no longer occur at or below $t_{safe}^{Paxos}$ with respect to Paxos.

$t_{safe}^{TM}$ is $\infty$ at a replica if there are zero prepared (but not committed) transactions—that is, transactions in between the two phases of two-phase commit. (For a participant slave, $t_{safe}^{TM}$ actually refers to the replica's leader's transaction manager, whose state the slave can infer through metadata passed on Paxos writes.) If there are any such transactions, then the state affected by those transactions is indeterminate: a participant replica does not yet know whether such transactions will commit. As we discuss in Section 4.2.1, the commit protocol ensures that every participant knows a lower bound on a prepared transaction's timestamp. Every participant leader (for a group $g$) for a transaction $T_i$ assigns a prepare timestamp $s_{i,g}^{prepare}$ to its prepare record. The coordinator leader ensures that the transaction's commit timestamp $s_i >= s_{i,g}^{prepare}$ over all participant groups $g$. Therefore, for every replica in a group $g$, over all transactions $T_i$ prepared at $g$, $t_{safe}^{TM} = min_i(s_{i,g}^{prepare}) - 1$ over all transactions prepared at $g$.

*4.1.4. Assigning Timestamps to RO Transactions.* A snapshot transaction executes in two phases: assign a timestamp $s_{read}$ [Chan and Gray 1985], and then execute the transaction's reads as snapshot reads at $s_{read}$. The snapshot reads can execute at any replicas that are sufficiently up-to-date.

The simple assignment of $s_{read} = TT.now().latest$, at any time after a transaction starts, preserves external consistency by an argument analogous to that presented for writes in Section 4.1.2. However, such a timestamp may require the execution of the data reads at $s_{read}$ to block if $t_{safe}$ has not advanced sufficiently. (In addition, note that choosing a value of $s_{read}$ may also advance $s_{max}$ to preserve disjointness.) To reduce the chances of blocking, Spanner should assign the oldest timestamp that preserves external consistency. Section 4.2.2 explains how such a timestamp can be chosen.

### 4.2. Details

This section explains some of the practical details of read-write transactions and snapshot transactions elided earlier, as well as the implementation of a special transaction type used to implement atomic schema changes. It then describes some refinements of the basic schemes as described.

*4.2.1. Read-Write Transactions.* Like Bigtable, writes that occur in a transaction are buffered at the client until commit. As a result, reads in a transaction do not see the effects of the transaction's writes. This design works well in Spanner because a read returns the timestamps of any data read, and uncommitted writes have not yet been assigned timestamps.

Reads within read-write transactions use wound-wait [Rosenkrantz et al. 1978] to avoid deadlocks. The client issues reads to the leader replica of the appropriate group, which acquires read locks and then reads the most recent data. While a client transaction remains open, it sends keepalive messages to prevent participant leaders from timing out its transaction. When a client has completed all reads and buffered all writes, it begins two-phase commit. The client chooses a coordinator group and sends a commit message to each participant's leader with the identity of the coordinator and any buffered writes. Having the client drive the two-phase commit protocol avoids sending data across wide-area links twice.

A non-coordinator-participant leader first acquires write locks. It then chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous

transactions (to preserve monotonicity), and logs a prepare record through Paxos. Each participant then notifies the coordinator of its prepare timestamp.

The coordinator leader also first acquires write locks, but skips the prepare phase. It chooses a timestamp for the entire transaction after hearing from all other participant leaders. The commit timestamp $s$ must be greater than or equal to all prepare timestamps (to satisfy the constraints discussed in Section 4.1.3), greater than $TT.now().latest$ at the time the coordinator received its commit message, and greater than any timestamps the leader has assigned to previous transactions (again, to preserve monotonicity). The coordinator leader then logs a commit record through Paxos (or an abort if it timed out while waiting for the other participants).

Only Paxos leaders acquire locks. Lock state is only logged during transaction prepare. If any locks have been lost before prepare (either due to deadlock avoidance, timeout, or a Paxos leader change), the participant aborts. Spanner ensures that a prepare or commit record is only logged while all locks are still held. In the event of a leader change, the new leader restores any lock state for prepared but uncommitted transactions before it accepts any new transactions.

Before allowing any coordinator replica to apply the commit record, the coordinator leader waits until $TT.after(s)$, so as to obey the commit-wait rule described in Section 4.1.2. Because the coordinator leader chose $s$ based on $TT.now().latest$, and now waits until that timestamp is guaranteed to be in the past, the expected wait is at least $2 * \bar{\epsilon}$. This wait is typically overlapped with Paxos communication. After commit wait, the coordinator sends the commit timestamp to the client and all other participant leaders. Each participant leader logs the transaction's outcome through Paxos. All participants apply at the same timestamp and then release locks.

*4.2.2. Snapshot Transactions.* Assigning a timestamp requires a negotiation phase between all of the Paxos groups that are involved in the reads. As a result, Spanner requires a *scope* expression for every snapshot transaction, which is an expression that summarizes the keys that will be read by the entire transaction. Spanner automatically infers the scope for standalone queries.

If the scope's values are served by a single Paxos group, then the client issues the snapshot transaction to that group's leader. (The current Spanner implementation only chooses a timestamp for a snapshot transaction at a Paxos leader.) That leader assigns $s_{read}$ and executes the read. For a single-site read, Spanner generally does better than $TT.now().latest$. Define $LastTS()$ to be the timestamp of the last committed write at a Paxos group. If there are no prepared transactions, the assignment $s_{read} = LastTS()$ trivially satisfies external consistency: the transaction will see the result of the last write, and therefore be ordered after it.

If the scope's values are served by multiple Paxos groups, there are several options. The most complicated option is to do a round of communication with all of the groups's leaders to negotiate $s_{read}$ based on $LastTS()$. Spanner currently implements a simpler choice. The client avoids a negotiation round, and just executes its reads at $s_{read} = TT.now().latest$ (which may wait for safe time to advance). All reads in the transaction can be sent to replicas that are sufficiently up-to-date.

*4.2.3. Schema-Change Transactions.* TrueTime enables Spanner to support atomic schema changes. It would be infeasible to use a standard transaction, because the number of participants (the number of groups in a database) could be in the millions. Bigtable supports atomic schema changes in one datacenter, but its schema changes block all operations.

A Spanner schema-change transaction is a generally nonblocking variant of a standard transaction. First, it is explicitly assigned a timestamp in the future, which is registered in the prepare phase. As a result, schema changes across thousands

of servers can complete with minimal disruption to other concurrent activity. Second, reads and writes, which implicitly depend on the schema, synchronize with any registered schema-change timestamp at time $t$: they may proceed if their timestamps precede $t$, but they must block behind the schema-change transaction if their timestamps are after $t$. Without TrueTime, defining the schema change to happen at $t$ would be meaningless.

*4.2.4. Refinements.* $t_{safe}^{TM}$ as defined here has a weakness, in that a single prepared transaction prevents $t_{safe}$ from advancing. As a result, no reads can occur at later timestamps, even if the reads do not conflict with the transaction. Such false conflicts can be removed by augmenting $t_{safe}^{TM}$ with a fine-grained mapping from key ranges to prepared-transaction timestamps. This information can be stored in the lock table, which already maps key ranges to lock metadata. When a read arrives, it only needs to be checked against the fine-grained safe time for key ranges with which the read conflicts.

$LastTS()$ as defined here has a similar weakness: if a transaction has just committed, a nonconflicting snapshot transaction must still be assigned $s_{read}$ so as to follow that transaction. As a result, the execution of the read could be delayed. This weakness can be remedied similarly by augmenting $LastTS()$ with a fine-grained mapping from key ranges to commit timestamps in the lock table. (We have not yet implemented this optimization.) When a snapshot transaction arrives, its timestamp can be assigned by taking the maximum value of $LastTS()$ for the key ranges with which the transaction conflicts, unless there is a conflicting prepared transaction (which can be determined from fine-grained safe time).

$t_{safe}^{Paxos}$ as defined here has a weakness in that it cannot advance in the absence of Paxos writes. That is, a snapshot read at $t$ cannot execute at Paxos groups whose last write happened before $t$. Spanner addresses this problem by taking advantage of the disjointness of leader-lease intervals. Each Paxos leader advances $t_{safe}^{Paxos}$ by keeping a threshold above which future writes' timestamps will occur: it maintains a mapping $MinNextTS(n)$ from Paxos sequence number $n$ to the minimum timestamp that may be assigned to Paxos sequence number $n + 1$. A replica can advance $t_{safe}^{Paxos}$ to $MinNextTS(n) - 1$ when it has applied through $n$.

A single leader can enforce its $MinNextTS()$ promises easily. Because the timestamps promised by $MinNextTS()$ lie within a leader's lease, the disjointness invariant enforces $MinNextTS()$ promises across leaders. If a leader wishes to advance $MinNextTS()$ beyond the end of its leader lease, it must first extend its lease. Note that $s_{max}$ is always advanced to the highest value in $MinNextTS()$ to preserve disjointness.

A leader by default advances $MinNextTS()$ values every 8 seconds. Thus, in the absence of prepared transactions, healthy slaves in an idle Paxos group can serve reads at timestamps greater than 8 seconds old in the worst case. A leader may also advance $MinNextTS()$ values on demand from slaves.

*4.2.5. Paxos Leader-Lease Management.* The simplest means to ensure the disjointness of Paxos-leader-lease intervals would be for a leader to issue a synchronous Paxos write of the lease interval, whenever it would be extended. A subsequent leader would read the interval and wait until that interval has passed.

TrueTime can be used to ensure disjointness without these extra log writes. The potential $i$th leader keeps a lower bound on the start of a lease vote from replica $r$ as $v_{i,r}^{leader} = TT.now().earliest$, computed before $e_{i,r}^{send}$ (defined as when the lease

Table III. Operation Microbenchmarks. Mean and Standard Deviation over 10 Runs. 1D Means One Replica with Commit Wait Disabled

| | latency (ms) | | | throughput (Kops/sec) | | |
|---|---|---|---|---|---|---|
| replicas | write | snapshot transaction | snapshot read | write | snapshot transaction | snapshot read |
| 1D | 10.1 ± 0.3 | — | — | 4.2 ± 0.03 | — | — |
| 1 | 14.1 ± 1.0 | 1.3 ± 0.04 | 1.3 ± 0.02 | 4.2 ± 0.07 | 10.7 ± 0.6 | 11.4 ± 0.2 |
| 3 | 14.3 ± 0.5 | 1.4 ± 0.08 | 1.4 ± 0.08 | 1.8 ± 0.03 | 11.2 ± 0.4 | 32.0 ± 1.8 |
| 5 | 15.4 ± 2.3 | 1.4 ± 0.07 | 1.6 ± 0.04 | 1.2 ± 0.2 | 11.1 ± 0.5 | 46.8 ± 5.8 |

request is sent by the leader). Each replica $r$ grants a lease at time $e_{i,r}^{grant}$, which happens after $e_{i,r}^{receive}$ (when the replica receives a lease request); the lease ends at $t_{i,r}^{end} = TT.now().latest + lease\_length$, computed after $e_{i,r}^{receive}$. A replica $r$ obeys the **single-vote** rule: it will not grant another lease vote until $TT.after(t_{i,r}^{end})$ is true. To enforce this rule across different incarnations of $r$, Spanner logs a lease vote at the granting replica before granting the lease; this log write can be piggybacked upon existing Paxos-protocol log writes.

When the $i$th leader receives a quorum of votes (event $e_i^{quorum}$), it computes its lease interval as $lease_i = [TT.now().latest, min_r(v_{i,r}^{leader}) + lease\_length]$. The lease is deemed to have expired at the leader when $TT.before(min_r(v_{i,r}^{leader}) + lease\_length)$ is false. To prove disjointness, we make use of the fact that the $i$th and $(i+1)$th leaders must have one replica in common in their quorums. Call that replica $r0$. Proof:

$$
\begin{aligned}
lease_i.end &= min_r(v_{i,r}^{leader}) + lease\_length && \text{(by definition)}\\
min_r(v_{i,r}^{leader}) + lease\_length &\le v_{i,r0}^{leader} + lease\_length && \text{(min)}\\
v_{i,r0}^{leader} + lease\_length &\le t_{abs}(e_{i,r0}^{send}) + lease\_length && \text{(by definition)}\\
t_{abs}(e_{i,r0}^{send}) + lease\_length &\le t_{abs}(e_{i,r0}^{receive}) + lease\_length && \text{(causality)}\\
t_{abs}(e_{i,r0}^{receive}) + lease\_length &\le t_{i,r0}^{end} && \text{(by definition)}\\
t_{i,r0}^{end} &< t_{abs}(e_{i+1,r0}^{grant}) && \text{(single-vote)}\\
t_{abs}(e_{i+1,r0}^{grant}) &\le t_{abs}(e_{i+1}^{quorum}) && \text{(causality)}\\
t_{abs}(e_{i+1}^{quorum}) &\le lease_{i+1}.start && \text{(by definition)}
\end{aligned}
$$

## 5. EVALUATION

We first measure Spanner's performance with respect to replication, transactions, and availability. We then provide some data on TrueTime behavior, and a case study of our first client, F1.

### 5.1. Microbenchmarks

Table III presents some microbenchmarks for Spanner. These measurements were taken on timeshared machines: each spanserver ran on scheduling units of 4GB RAM and 4 cores (AMD Barcelona 2200MHz). Clients were run on separate machines. Each zone contained one spanserver. Clients and zones were placed in a set of datacenters with network distance of less than 1ms. (Such a layout should be commonplace: most

Table IV. Two-Phase Commit Scalability. Mean and Standard Deviations over 10 Runs

| participants | latency (ms) | |
|---|---|---|
| | mean | 99th percentile |
| 1 | 14.6 ± 0.2 | 26.550 ± 6.2 |
| 2 | 20.7 ± 0.4 | 31.958 ± 4.1 |
| 5 | 23.9 ± 2.2 | 46.428 ± 8.0 |
| 10 | 22.8 ± 0.4 | 45.931 ± 4.2 |
| 25 | 26.0 ± 0.6 | 52.509 ± 4.3 |
| 50 | 33.8 ± 0.6 | 62.420 ± 7.1 |
| 100 | 55.9 ± 1.2 | 88.859 ± 5.9 |
| 200 | 122.5 ± 6.7 | 206.443 ± 15.8 |

applications do not need to distribute all of their data worldwide.) The test database was created with 50 Paxos groups with 2500 directories. Operations were standalone reads and writes of 4KB. All reads were served out of memory after a compaction, so that we are only measuring the overhead of Spanner's call stack. In addition, one unmeasured round of reads was done first to warm any location caches.

For the latency experiments, clients issued sufficiently few operations so as to avoid queuing at the servers. From the 1-replica experiments, commit wait is about 4ms, and Paxos latency is about 10ms. As the number of replicas increases, the latency increases slightly because Paxos must commit at more replicas. Because Paxos executes in parallel at a group's replicas, the increase is sublinear: the latency depends on the slowest member of the quorum. We can see the most measurement noise in write latency.

For the throughput experiments, clients issued sufficiently many operations so as to saturate the servers' CPUs. In addition, Paxos leaders were pinned to one zone, so as to keep constant across experiments, the number of machines on which leaders run. Snapshot reads can execute at any up-to-date replicas, so their throughput increases with the number of replicas. Single-read snapshot transactions only execute at leaders because timestamp assignment must happen at leaders. As a result, snapshot-transaction throughput stays roughly constant with the number of replicas. Finally, write throughput decreases with the number of replicas, since the amount of Paxos work increases linearly with the number of replicas. Again, there is a fair amount of noise in our measurements, since we ran in production datacenters: we had to discard some data that could only be explained by interference from other jobs.

Table IV demonstrates that two-phase commit can scale to a reasonable number of participants: it summarizes a set of experiments run across 3 zones, each with 25 spanservers. Scaling up to 50 participants is reasonable in both mean and 99th-percentile; latencies start to rise noticeably at 100 participants.

### 5.2. Availability

Figure 5 illustrates the availability benefits of running Spanner in multiple datacenters. It shows the results of three experiments on throughput in the presence of datacenter failure, all of which are overlaid onto the same time scale. The test universe consisted of 5 zones $Z_i$, each of which had 25 spanservers. The test database was sharded into 1250 Paxos groups, and 100 test clients constantly issued nonsnapshot reads at an aggregate rate of 50K reads/second. All of the leaders were explicitly placed in $Z_1$. Five seconds into each test, all of the servers in one zone were killed: *nonleader* kills $Z_2$; *leader-hard* kills $Z_1$; *leader-soft* kills $Z_1$, but it gives notifications to all of the servers that they should handoff leadership first.

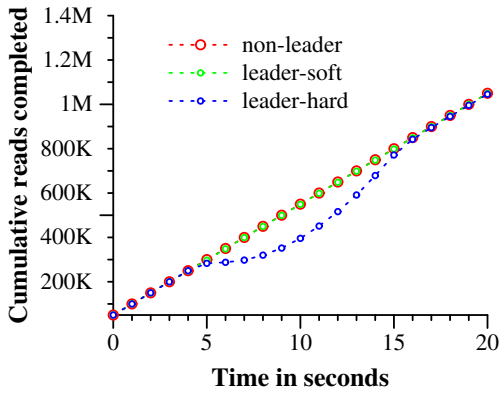



Fig. 5. Effect of killing servers on throughput.

Killing $Z_2$ does not affect read throughput. Killing $Z_1$ while giving the leaders time to handoff leadership to a different zone has a minor effect: the throughput drop is not visible in the graph, but is around 3–4%. On the other hand, killing $Z_1$ with no warning has a severe effect: the rate of completion drops almost to 0. As leaders get reelected, though, the throughput of the system rises to approximately 100K reads/second because of two artifacts of our experiment: there is extra capacity in the system, and operations are queued while the leader is unavailable. As a result, the throughput of the system rises before leveling off again at its steady-state rate.

We can also see the effect of the fact that Paxos leader leases are set to 10 seconds. When we kill the zone, the leader-lease expiration times for the groups should be evenly distributed over the next 10 seconds. Soon after each lease from a dead leader expires, a new leader is elected. Approximately 10 seconds after the kill time, all of the groups have leaders and throughput has recovered. Shorter lease times would reduce the effect of server deaths on availability, but would require greater amounts of lease-renewal network traffic. We are in the process of designing and implementing a mechanism that will cause slaves to release Paxos leader leases upon leader failure.

### 5.3. TrueTime

Two questions must be answered with respect to TrueTime: is $\epsilon$ truly a bound on clock uncertainty, and how bad does $\epsilon$ get? For the former, the most serious problem would be if a local clock's drift were greater than 200us/sec: that would break assumptions made by TrueTime. Our machine statistics show that bad CPUs are 6 times more likely than bad clocks. That is, clock issues are extremely infrequent, relative to much more serious hardware problems. As a result, we believe that TrueTime's implementation is as trustworthy as any other piece of software upon which Spanner depends.

Figure 6 presents TrueTime data taken at several thousand spanserver machines across datacenters up to 2200 km apart. It plots the 90th, 99th, and 99.9th percentiles of $\epsilon$, sampled at timeslave daemons immediately after polling the time masters. This sampling elides the sawtooth in $\epsilon$ due to local-clock uncertainty, and therefore measures time-master uncertainty (which is generally 0) plus communication delay to the time masters.

The data shows that these two factors in determining the base value of $\epsilon$ are generally not a problem. However, there can be significant tail-latency issues that cause higher values of $\epsilon$. The reduction in tail latencies beginning on March 30 were due to networking improvements that reduced transient network-link congestion. The increase in $\epsilon$ on April 13, approximately one hour in duration, resulted from the

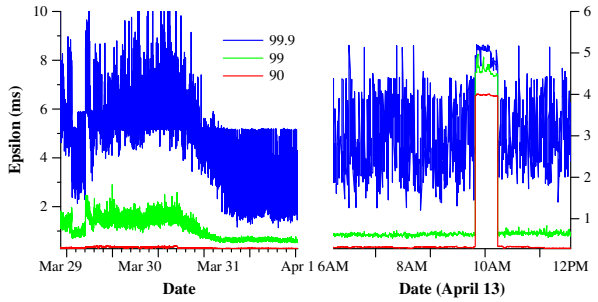



Fig. 6. Distribution of TrueTime $\epsilon$ values, sampled right after timeslave daemon polled the time masters. 90th, 99th, and 99.9th percentiles are graphed.

shutdown of 2 time masters at a datacenter for routine maintenance. We continue to investigate and remove causes of TrueTime spikes.

### 5.4. F1

Spanner started being experimentally evaluated under production workloads in early 2011, as part of a rewrite of Google's advertising backend, called F1 [Shute et al. 2012]. This backend was originally based on a MySQL database that was manually sharded many ways. The uncompressed dataset is tens of terabytes, which is small compared to many NoSQL instances, but was large enough to cause difficulties with sharded MySQL. The MySQL sharding scheme assigned each customer and all related data to a fixed shard. This layout enabled the use of indexes and complex query processing on a per-customer basis, but required some knowledge of the sharding in application business logic. Resharding this revenue-critical database as it grew in the number of customers and the size of their data was extremely costly. The last resharding took over two years of intense effort, and involved coordination and testing across dozens of teams to minimize risk. This operation was too complex to do regularly: as a result, the team had to limit growth on the MySQL database by storing some data in external Bigtables, which compromised transactional behavior and the ability to query across all data.

The F1 team chose to use Spanner for several reasons. First, Spanner removes the need to manually reshard. Second, Spanner provides synchronous replication and automatic failover. With MySQL master-slave replication, failover was difficult, and risked data loss and downtime. Third, F1 requires strong transactional semantics, which made using other NoSQL systems impractical. Application semantics requires transactions across arbitrary data, and consistent reads. The F1 team also needed secondary indexes on their data (since Spanner does not yet provide automatic support for secondary indexes), and was able to implement their own consistent global indexes using Spanner transactions.

All application writes are now by default sent through F1 to Spanner, instead of the MySQL-based application stack. F1 has 2 replicas on the west coast of the US, and 3 on the east coast. This choice of replica sites was made to cope with outages due to potential major natural disasters, and also the choice of their frontend sites. Anecdotally, Spanner's automatic failover has been nearly invisible to them. Although there have been unplanned cluster failures in the last few months, the most that the F1 team has had to do is update their database's schema to tell Spanner where to preferentially place Paxos leaders, so as to keep them close to where their frontends moved.

Table V. Distribution of Directory-Fragment Counts in F1

| # fragments | # directories |
|---|---|
| 1 | >100M |
| 2–4 | 341 |
| 5–9 | 5336 |
| 10–14 | 232 |
| 15–99 | 34 |
| 100–500 | 7 |

Table VI. F1-Perceived Operation Latencies Measured over the Course of 24 Hours

| operation | latency (ms) | | count |
|---|---|---|---|
| | mean | std dev | |
| all reads | 8.7 | 376.4 | 21.5B |
| single-site commit | 72.3 | 112.8 | 31.2M |
| multi-site commit | 103.0 | 52.2 | 32.1M |

Spanner's timestamp semantics made it efficient for F1 to maintain in-memory data structures computed from the database state. F1 maintains a logical history log of all changes, which is written into Spanner itself as part of every transaction. F1 takes full snapshots of data at a timestamp to initialize its data structures, and then reads incremental changes to update them.

Table V illustrates the distribution of the number of fragments per directory in F1. Each directory typically corresponds to a customer in the application stack above F1. The vast majority of directories (and therefore customers) consist of only 1 fragment, which means that reads and writes to those customers' data are guaranteed to occur on only a single server. The directories with more than 100 fragments are all tables that contain F1 secondary indexes: writes to more than a few fragments of such tables are extremely uncommon. The F1 team has only seen such behavior when they do untuned bulk data loads as transactions.

Table VI presents Spanner operation latencies as measured from F1 servers. Replicas in the east coast data centers are given higher priority in choosing Paxos leaders. The data in the table is measured from F1 servers in those data centers. The large standard deviation in write latencies is caused by a pretty fat tail due to lock conflicts. The even larger standard deviation in read latencies is partially due to the fact that Paxos leaders are spread across two data centers, only one of which has machines with SSDs. In addition, the measurement includes every read in the system from two datacenters: the mean and standard deviation of the bytes read were roughly 1.6KB and 119KB, respectively.

## 6. RELATED WORK

Consistent replication across datacenters as a storage service has been provided by Megastore [Baker et al. 2011] and DynamoDB [Amazon 2012]. DynamoDB presents a key-value interface, and only replicates within a region. Spanner follows Megastore in providing a semirelational data model, and even a similar schema language. Megastore does not achieve high performance. It is layered on top of Bigtable, which imposes high communication costs. It also does not support long-lived leaders: multiple replicas may initiate writes. All writes from different replicas necessarily conflict in the Paxos protocol, even if they do not logically conflict: throughput collapses on a Paxos group

at several writes per second. Spanner provides higher performance, general-purpose transactions, and external consistency.

Pavlo et al. [2009] have compared the performance of databases and MapReduce [Dean and Ghemawat 2010]. They point to several other efforts that have been made to explore database functionality layered on distributed key-value stores [Abouzeid et al. 2009; Armbrust et al. 2011; Brantner et al. 2008; Thusoo et al. 2010] as evidence that the two worlds are converging. We agree with the conclusion, but demonstrate that integrating multiple layers has its advantages: integrating concurrency control with replication reduces the cost of commit wait in Spanner, for example.

The notion of layering transactions on top of a replicated store dates at least as far back as Gifford's dissertation [Gifford 1982]. Scatter [Glendenning et al. 2011] is a recent DHT-based key-value store that layers transactions on top of consistent replication. Spanner focuses on providing a higher-level interface than Scatter does. Gray and Lamport [2006] describe a nonblocking commit protocol based on Paxos. Their protocol incurs more messaging costs than two-phase commit, which would aggravate the cost of commit over widely distributed groups. Walter [Sovran et al. 2011] provides a variant of snapshot isolation that works within, but not across, datacenters. In contrast, our snapshot transactions provide a more natural semantics, because we support external consistency over all operations.

There has been a spate of recent work on reducing or eliminating locking overheads. Calvin [Thomson et al. 2012] eliminates concurrency control: it preassigns timestamps and then executes the transactions in timestamp order. H-Store [Stonebraker et al. 2007] and Granola [Cowling and Liskov 2012] each supported their own classification of transaction types, some of which could avoid locking. None of these systems provides external consistency. Spanner addresses the contention issue by providing support for snapshot isolation.

VoltDB [2012] is a sharded in-memory database that supports master-slave replication over the wide area for disaster recovery, but not more general replication configurations. It is an example of what has been called NewSQL, which is a marketplace push to support scalable SQL [Stonebraker 2010a]. A number of commercial databases implement reads in the past, such as MarkLogic [2012] and Oracle's Total Recall [Oracle 2012]. Lomet and Li [2009] describe an implementation strategy for such a temporal database.

Farsite derived bounds on clock uncertainty (much looser than TrueTime's) relative to a trusted clock reference [Douceur and Howell 2003]: server leases in Farsite were maintained in the same way that Spanner maintains Paxos leases. Loosely synchronized clocks have been used for concurrency-control purposes in prior work [Adya et al. 1995; Liskov 1993]. We have shown that TrueTime lets one reason about global time across sets of Paxos state machines.

## 7. FUTURE WORK

We have spent most of the last year working with the F1 team to transition Google's advertising backend from MySQL to Spanner. We are actively improving its monitoring and support tools, as well as tuning its performance. In addition, we have been working on improving the functionality and performance of our backup/restore system. We are currently implementing the Spanner schema language, automatic maintenance of secondary indices, and automatic load-based resharding. Longer term, there are a couple of features that we plan to investigate. Optimistically doing reads in parallel may be a valuable strategy to pursue, but initial experiments have indicated that the right implementation is nontrivial. In addition, we plan to eventually support direct changes of Paxos configurations [Lamport et al. 2010; Shraer et al. 2012].

Given that we expect many applications to replicate their data across datacenters that are relatively close to each other, TrueTime $\epsilon$ may noticeably affect performance. We see no insurmountable obstacle to reducing $\epsilon$ below 1ms. Time-master-query intervals can be reduced, and better clock crystals are relatively cheap. Time-master query latency could be reduced with improved networking technology, or possibly even avoided through alternate time-distribution technology.

Finally, there are obvious areas for improvement. Although Spanner is scalable in the number of nodes, the node-local data structures have relatively poor performance on complex SQL queries, because they were designed for simple key-value accesses. Algorithms and data structures from DB literature could improve single-node performance a great deal. Second, moving data automatically between datacenters in response to changes in client load has long been a goal of ours, but to make that goal effective, we would also need the ability to move client-application processes between datacenters in an automated, coordinated fashion. Moving processes raises the even more difficult problem of managing resource acquisition and allocation between datacenters.

## 8. CONCLUSIONS

To summarize, Spanner combines and extends ideas from two research communities: from the database community, a familiar, easy-to-use, semirelational interface, transactions, and an SQL-based query language; from the systems community, scalability, automatic sharding, fault tolerance, consistent replication, external consistency, and wide-area distribution. Since Spanner's inception, we have taken more than 5 years to iterate to the current design and implementation. Part of this long iteration phase was due to a slow realization that Spanner should not only tackle the problem of a globally replicated namespace, it should also focus on database features that Bigtable was missing.

One aspect of our design stands out: the linchpin of Spanner's feature set is TrueTime. We have shown that reifying clock uncertainty in the time API makes it possible to build distributed systems with much stronger time semantics. In addition, as the underlying system enforces tighter bounds on clock uncertainty, the overhead of the stronger semantics decreases. As a community, we should no longer depend on loosely synchronized clocks and weak time APIs in designing distributed algorithms.

## ACKNOWLEDGMENTS

Many people have helped to improve this article: our shepherd Jon Howell, who went above and beyond his responsibilities; the anonymous referees; and many Googlers: Atul Adya, Fay Chang, Frank Dabek, Sean Dorward, Bob Gruber, David Held, Nick Kline, Alex Thomson, and Joel Wein. Our management has been very supportive of both our work and of publishing this article: Aristotle Balogh, Bill Coughran, Urs Hölzle, Doron Meyer, Cos Nicolaou, Kathy Polizzi, Sridhar Ramaswany, and Shivakumar Venkataraman. We have built upon the work of the Bigtable and Megastore teams. The F1 team, and Jeff Shute in particular, worked closely with us in developing our data model and helped immensely in tracking down performance and correctness bugs. The Platforms team, and Luiz Barroso and Bob Felderman in particular, helped to make TrueTime happen. Finally, a lot of Googlers used to be on our team: Ken Ashcraft, Paul Cychosz, Krzysztof Ostrowski, Amir Voskoboynik, Matthew Weaver, Theo Vassilakis, and Eric Veach; or have joined our team recently: Nathan Bales, Adam Beberg, Vadim Borisov, Ken Chen, Brian Cooper, Cian Cullinan, Robert-Jan Huijsman, Milind Joshi, Andrey Khorlin, Dawid Kuroczko, Laramie Leavitt, Eric Li, Mike Mammarella, Sunil Mushran, Simon Nielsen, Ovidiu Platon, Ananth Shrinivas, Vadim Suvorov, and Marcel van der Holst.

## REFERENCES

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., and Rasin, A. 2009. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. In *Proceedings of the International Conference on Very Large Data Bases*. 922–933.

Adya, A., Gruber, R., Liskov, B., and Maheshwari, U. 1995. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 23–34.

Amazon. 2012. Amazon dynamodb. http://aws.amazon.com/dynamodb.

Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M., and Patterson, D. 2011. PIQL: Success-tolerant query processing in the cloud. In *Proceedings of the International Conference on Very Large Data Bases*. 181–192.

Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Léon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of CIDR*. 223–234.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1–10.

Brantner, M., Florescu, D., Graf, D., Kossmann, D., and Kraska, T. 2008. Building a database on S3. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 251–264.

Chan, A. and Gray, R. 1985. Implementing distributed read-only transactions. *IEEE Trans. Softw. Eng. SE-11,* 2, 205–212.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26,* 2, 4:1–4:26.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. 2008. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of the International Conference on Very Large Data Bases*. 1277–1288.

Cowling, J. and Liskov, B. 2012. Granola: Low-overhead distributed transaction coordination. In *Proceedings of USENIX ATC*. 223–236.

Dean, J. and Ghemawat, S. 2010. MapReduce: A flexible data processing tool. *Comm. ACM 53,* 1, 72–77.

Douceur, J. and Howell, J. 2003. Scalable Byzantine-fault-quantifying clock synchronization. Tech. rep. MSR-TR-2003-67, MS Research.

Douceur, J. R. and Howell, J. 2006. Distributed directory service in the Farsite file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 321–334.

Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In *Proceedings of the Symposium on Operating Systems Principles*. 29–43.

Gifford, D. K. 1982. Information storage in a decentralized computer system. Tech. rep. CSL-81-8, Xerox PARC.

Glendenning, L., Beschastnikh, I., Krishnamurthy, A., and Anderson, T. 2011. Scalable consistency in scatter. In *Proceedings of the Symposium on Operating Systems Principles*.

Google. 2008. Protocol buffers — Google's data interchange format. https://code.google.com/p/protobuf.

Gray, J. and Lamport, L. 2006. Consensus on transaction commit. *ACM Trans. Datab. Syst. 31,* 1, 133–160.

Helland, P. 2007. Life beyond distributed transactions: An apostate's opinion. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*. 132–141.

Herlihy, M. P. and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12,* 3, 463–492.

Lamport, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16,* 2, 133–169.

Lamport, L., Malkhi, D., and Zhou, L. 2010. Reconfiguring a state machine. *SIGACT News 41,* 1, 63–73.

Liskov, B. 1993. Practical uses of synchronized clocks in distributed systems. *Distrib. Comput. 6,* 4, 211–219.

Lomet, D. B. and Li, F. 2009. Improving transaction-time DBMS performance and functionality. In *Proceedings of the International Conference on Data Engineering*. 581–591.

Lorch, J. R., Adya, A., Bolosky, W. J., Chaiken, R., Douceur, J. R., and Howell, J. 2006. The SMART way to migrate replicated stateful services. In *Proceedings of EuroSys*. 103–115.

MarkLogic. 2012. Marklogic 5 product documentation. http://community.marklogic.com/docs.

Marzullo, K. and Owicki, S. 1983. Maintaining the time in a distributed system. In *Proceedings of the Symposium on Principles of Distributed Computing*. 295–305.

Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., and Vassilakis, T. 2010. Dremel: Interactive analysis of Web-scale datasets. In *Proceedings of the International Conference on Very Large Data Bases*. 330–339.

Mills, D. 1981. Time synchronization in DCNET hosts. Internet project rep. IEN–173, COMSAT Laboratories.

Oracle. 2012. Oracle total recall. http://www.oracle.com/technetwork/database/focus-areas/storage/total-recall-whitepaper-171749.pdf.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., and Stonebraker, M. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 165–178.

Peng, D. and Dabek, F. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 1–15.

Rosenkrantz, D. J., Stearns, R. E., and Lewis II, P. M. 1978. System level concurrency control for distributed database systems. *ACM Trans. Datab. Syst. 3,* 2, 178–198.

Shraer, A., Reed, B., Malkhi, D., and Junqueiera, F. 2012. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of USENIX ATC*. 425–438.

Shute, J., Oancea, M., Ellner, S., Handy, B., Rollins, E., Samwel, B., Vingralek, R., Whipkey, C., Chen, X., Jegerlehner, B., Littlefield, K., and Tong, P. 2012. F1 — The fault-tolerant distributed RDBMS supporting Google's ad business. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 777–778.

Sovran, Y., Power, R., Aguilera, M. K., and Li, J. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the Symposium on Operating Systems Principles*. 385–400.

Stonebraker, M. 2010a. Six SQL urban myths. http://highscalability.com/blog/2010/6/28/voltdb-decapitates-six-sql-urban-myths-and-delivers-internet.html.

Stonebraker, M. 2010b. Why enterprises are uninterested in NoSQL. http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. 2007. The end of an architectural era: (It's time for a complete rewrite). In *Proceedings of the International Conference on Very Large Data Bases*. 1150–1160.

Thomson, A., Diamond, T., Shu-Chun Weng, T. D., Ren, K., Shao, P., and Abadi, D. J. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1–12.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R. 2010. Hive — A petabyte scale data warehouse using Hadoop. In *Proceedings of the International Conference on Data Engineering*. 996–1005.

VoltDB. 2012. VoltDB resources. http://voltdb.com/resources/whitepapers.

Received January 2013; accepted May 2013

Machine Learning 36, 85–103 (1999)


# Pasting Small Votes for Classification in Large Databases and On-Line

LEO BREIMAN leo@stat.berkeley.edu
*Statistics Department, University of California, Berkeley, CA 94708*

**Editors:** David Wolpert, Philip Chan and Salvatore Stolfo

**Abstract.** Many databases have grown to the point where they cannot fit into the fast memory of even large memory machines, to say nothing of current workstations. If what we want to do is to use these data bases to construct predictions of various characteristics, then since the usual methods require that all data be held in fast memory, various work-arounds have to be used. This paper studies one such class of methods which give accuracy comparable to that which could have been obtained if all data could have been held in core and which are computationally fast.The procedure takes small pieces of the data, grows a predictor on each small piece and then pastes these predictors together. A version is given that scales up to terabyte data sets. The methods are also applicable to on-line learning.

**Keywords:** combining, database, votes, pasting

## 1. Introduction

Suppose that the data base $D$ is a large collection of examples $(y_n, \mathbf{x}_n)$ where the $\mathbf{x}_n$ are input vectors and the $y_n$ are class labels. What we want to do is use this data to construct a classifier which can accurately assign a class label to future inputs $\mathbf{x}$. But $D$ is too large to hold in the memory of any currently available computer.

What we show in this paper is that the aggregation of many classifiers, each grown on a training set of modest size $N$ selected from the data base $D$, can achieve almost optimum classification accuracy. The procedure, which we refer to as pasting votes together, has two key ideas in its implementation:

(i) Suppose that up to the present, $k$ predictors have been constructed. A new training set of size $N$ is selected from $D$ either by random or importance sampling. The $(k+1)$st predictor is grown on this new training set and aggregated with the previous $k$. The aggregation is by unweighted plurality voting. If random sampling is used to select the training set, the training set is called an *Rprecinct* and the procedure is called *pasting Rvotes* (R = random). If it is done using importance sampling, the training set is called an *Iprecinct* and the procedure as *pasting Ivotes* (I = importance). The importance sampling used (see Section 2) samples more heavily from the instances more likely to be misclassified.

(ii) An estimate $e(k)$ of the generalization error for the $k$th aggregation $e(k)$ is updated. The pasting stops when $e(k)$ stops decreasing. The estimate of $e(k)$ can be gotten using a test set, but our implementation uses out-of-bag estimation (Breiman, 1996).

CART (Breiman et al., 1984) is used as a test bed predictor, but it is clear that pasting votes will work with other prediction methods. In Section 2, we explore two versions in classification-pasting Ivotes and pasting Rvotes. Using Rvotes is less complicated, but pasting Ivotes gives considerably more accuracy. We experiment on five moderate sized data sets. The accuracy of pasting Ivotes is compared to trees, single and multiple, grown using the entire data set.

Section 3 applies a version of pasting Ivotes designed to minimize disk accesses to a synthetic data base of a million records, each containing data on 61 variables. The total disk access and read times were tallied together with the tree construction times and the time needed to construct the Iprecincts. They had similar magnitudes. An analysis shows that this version of pasting Ivotes scales up to terabyte data bases. In Section 4 pasting Ivotes is applied to on-line learning using a synthetic data set as a test bed. Comments and conclusions are given in Section 5 and a look at related work in Section 6.

## 2. Pasting votes

### *2.1. Method description*

The simplest version of pasting is to select each training set of size $N$ by random sampling from the data base $D$, grow the classifier, repeat a preassigned number $K$ of times, stop and aggregate the classifiers by voting. This is certainly workable and cheap. If, after aggregation, accuracy is checked on a test set, then further runs can be used to optimize the values of $K$ and $N$. A more sophisticated version estimates $e(k)$ after the $k$th aggregation and stops when $e(k)$ stops decreasing.

There are three methods that can be used to estimate $e(k)$:

*First*: Set aside a fixed test set of examples in $D$. Run the $k$th aggregated classifier on the test set. Estimate $e(k)$ by the error on this test set.

*Second*: If $T$ is the $(k+1)$st training set, let $r(k)$ be the error rate of the $k$th aggregated classifier on $T$. Since $N$ is small, $r(k)$ with be a noisy estimate of $e(k)$. Smooth it by defining $e(k) = p * e(k-1) + (1-p)^* r(k)$. The value $p = 0.75$ was used in all of our experiments, but results are not sensitive to the value of $p$.

If the total number of examples used in the repeated sampling of training sets gets above an appreciable fraction of the number in $D$, the second estimate will be biased downward because some of the examples in the $(k+1)$st training set will have been used to construct the previous classifiers.

*Third*: To eliminate the bias in the second method, if $T_h$ is the $h$th training set, and $\mathrm{C}(\mathbf{x}, h)$ the classifier for input vector $\mathbf{x}$ constructed using $T_h$, then classify an example $(y, \mathbf{x})$ that is a candidate for the $(k+1)$st training set by aggregating all of the classifiers $\mathrm{C}(\mathbf{x}, h)$, $h < k+1$, such that $(y, \mathbf{x})$ is not in $T_h$. This is the out-of-bag classifier $\mathrm{C}^{\mathrm{OB}}(\mathbf{x}, k)$. Estimate the error $r(k)$ as the proportion of misclassifications made by $\mathrm{C}^{\mathrm{OB}}$. Smooth the $r(k)$ as in the second method to get the estimate $e^{\mathrm{OB}}(k)$.

Out-of-bag (OB) estimation is shown in Breiman (1996) to give effective estimates of the generalization error. (Regarding OB estimation, see also Wolpert (1996) and

Tibshirani (1996)) In the examples we run, both the test set $e^{\mathrm{TS}}(k)$ and the estimate $e^{\mathrm{OB}}(k)$ are computed and compared. Also compared are two methods for selecting the examples for the $(k+1)$st training sets.

*Random*: This is simple random selection from $D$ with all examples having the same probability of being selected. Continue until $N$ examples are selected. The classifier grown on this training set is called a Rvote.

*Importance*: In this procedure, an example $(y, \mathbf{x})$ is selected at random from $D$ with all examples having the same probability of being selected. Let $\mathrm{C}^{\mathrm{OB}}(\mathbf{x}, k)$ be the out-of-bag classifier at stage $k$. If $y \neq \mathrm{C}^{\mathrm{OB}}(\mathbf{x}, k)$ then put this example in the training set. Otherwise, put it in the training set with probability $e(k)/(1-e(k))$. Repeat until $N$ examples have been collected. Refer to the classifier grown on this training sets as an Ivote.

The rationale for the importance sampling procedure is that the new training set will contain about equal numbers of incorrectly and correctly classified examples. The probability that the next instance sampled is incorrectly classified and put into the training set is $e(k)$. The probability that the next instance sampled is correctly classified and put into the training set is also $e(k) = (1-e(k)) * [e(k)/(1-e(k))]$. The sampling probabilities change as more classifiers are pasted together. In this respect it is an arcing algorithm, where arcing is an acronym standing for **a**daptive **r**esampling and **c**ombin**ing** (Breiman, 1998).

An apparent question is: if one wants to get equal numbers of misclassified and correctly classified instances into a training set of size $N$, why not sample until we get $N/2$ of each? One answer is that for $e(k) < 1/2$, the expected number of instances we have to sample to accomplish this is about the same as the expected number needed using the rejection sampling. The rejection sampling adds a randomness to the training set selection that may be important in sequential sampling from large data bases (see Section 3).

Although the algorithm for importance sampling used here differs essentially from those used in the Breiman (1998) paper where the entire data base was used to grow each classifier, it retains the basic idea—put increased weight on those examples more likely to be misclassified. The first effective arcing algorithm is due to Freund and Schapire (1995, 1996) and named Adaboost (see also (Drucker & Cortes, 1996; Quinlan, 1996; Breiman, 1997). An idea similar to pasting Ivotes was suggested by Schapire (1990) in the context of boosting in PAC learning, but used a sequence of training sets increasing in size.

In our experiments, $N$ is taken to be a few hundred examples out of data sets having up to 43,500 examples; In four of the data sets we use in our experiments Adaboost-CART was shown to have a lower overall error rate than any of the other 22 well-known classification methods reported on in the Statlog Project (Michie, Spiegelhalter, & Taylor, 1994). The fifth data base is the famous Post Office handwritten digit data on which Adaboosted CART is competitive with hand tailored neural nets.

Two surprising and gratifying things emerge in the experiments:

(1) *Pasting together Ivotes, each one grown using only a few hundred examples, gives accuracy comparable to running Adaboost using the whole data set at each iteration.*

(2) *The computing times to construct the pasted classifiers are very nominal, making the procedure computationally feasible even for data bases much larger than fast memory.*

Pasting never requires storage of the entire data base $D$ in core. Examples are selected, tested, and pulled out to form the small training sets. The $K$ trees constructed to date need to be stored. This takes about 14KN bytes—very workstation feasible. The trees produced are not pruned—the aggregation seems to eliminate the overfitting. Each tree construction takes order $MN \log N$ flops where $M$ is the number of input variables.

The selection of the small training sets adds a computational burden. Assuming that the estimated error $e = e(k)$ is about equal to the true error rate over the whole database and $e < 1/2$, the expected number of instances sampled from the database to form the $k$th training set is $N/2e$ (see Appendix). To decide whether to accept an instance, it must be run through all trees constructed to date that do not use the instance in the training set. This is accomplished as follows: For each instance, we store an integer $N_L$ equal to the tree number when the instance was last run through the previous trees together with $J$ integers $nc(1), \ldots, nc(J)$ where $nc(j)$ is the number of times the instance was classified as class $j$ in the run through the trees constructed prior to $N_L$.

Then, to update the $nc(1), \ldots, nc(J)$ the instance is passed through the trees constructed from tree $N_L$ up to the last tree constructed. If $N_B$ is the number of instances in the entire database, then the expected number of flops needed to form the training set for each tree construction is proportional to $\log(N)\ N_B$ (see Appendix). For large $N_B$, this becomes appreciable. But keeping track of which trees to run the instances through requires only the updating of $N_L$ and the $J$-vector **nc** for each instance selected in forming the current Iprecinct.

### *2.2. Pasting Ivotes*

The data sets used these experiments are summarized in Table 1.

The first four of these data sets were used in the Statlog project and are described in Michie, Spiegelhalter, and Taylor (1994). The last data set is the well-known handwritten digit recognition data set. The data exists as separated into test and training set. The results of 10cv-CART and Adaboost-CART are given in Table 2. On the digit data 100 iterations were used in Adaboost. The other Adaboost results are based on 50 iterations.

To track the effect of $N$, the size of the training sets, we ran pasting on all data sets with $N = 100, 200, 400, 800$. The number of total iterations was chosen, in each data set, to be

*Table 1.* Data set summary.

| Data set | Classes | Inputs | Training | Test |
|---|---|---|---|---|
| letters | 26 | 16 | 15000 | 5000 |
| satellite | 6 | 36 | 4435 | 2000 |
| shuttle | 7 | 9 | 43500 | 14500 |
| dna | 3 | 60 | 2000 | 1186 |
| digit | 10 | 256 | 7291 | 2007 |

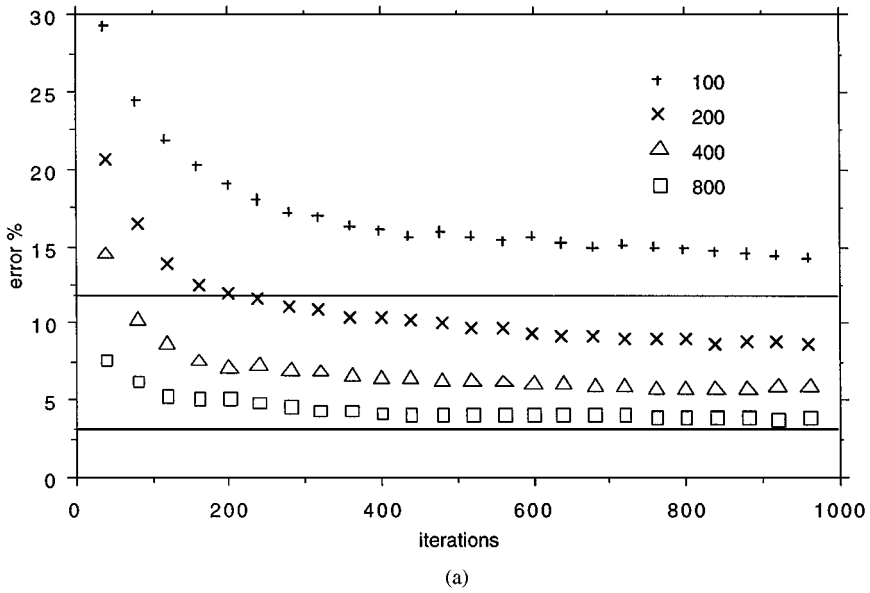
*Table 2.* Test set error (% misclassification).

| Data set | 10cv-CART | Adaboost-CART |
|---|---|---|
| letters | 12.4 | 3.4 |
| satellite | 14.8 | 8.8 |
| shuttle | 0.062 | 0.007 |
| dna | 6.2 | 4.2 |
| digit | 27.1 | 6.2 |

past the point of decreasing test set error. In letters, this was 1000, 500 for the digits data, 250 for the satellite data, 100 for the dna data, and 50 for the shuttle data. In these runs, we kept track of the test set error, the OB estimates and compute times.

***2.2.1. Test set error.*** The test set error $e^{\mathrm{TS}}(k)$ is computed by running the set aside test set through the classifiers pasted together after the $k$th Ibite. These are shown in figure 1(a–e) which give graphs of the test set error $e^{\mathrm{TS}}(k)$ at the $k$th iteration as a function of $k$. Each graph contains plots of $e^{\mathrm{TS}}(k)$ for $N = 100, 200, 400, 800$. These can usually be distinguished by the fact that for a given value of $k$, test set error is highest for $N = 100$, and decreases to $N = 800$. The higher horizontal line is the error rate for 10cv-CART; the lower for Adaboost-CART.

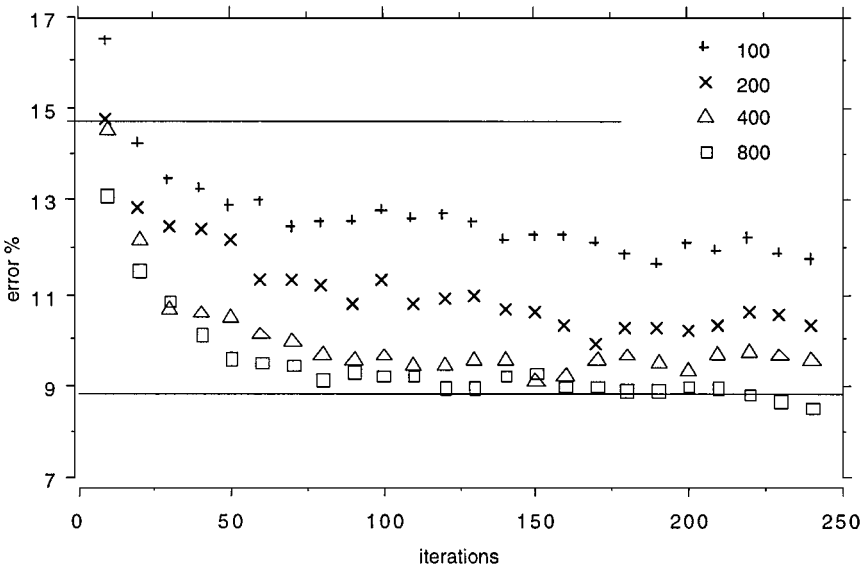
These conclusions can be read from the graphs:



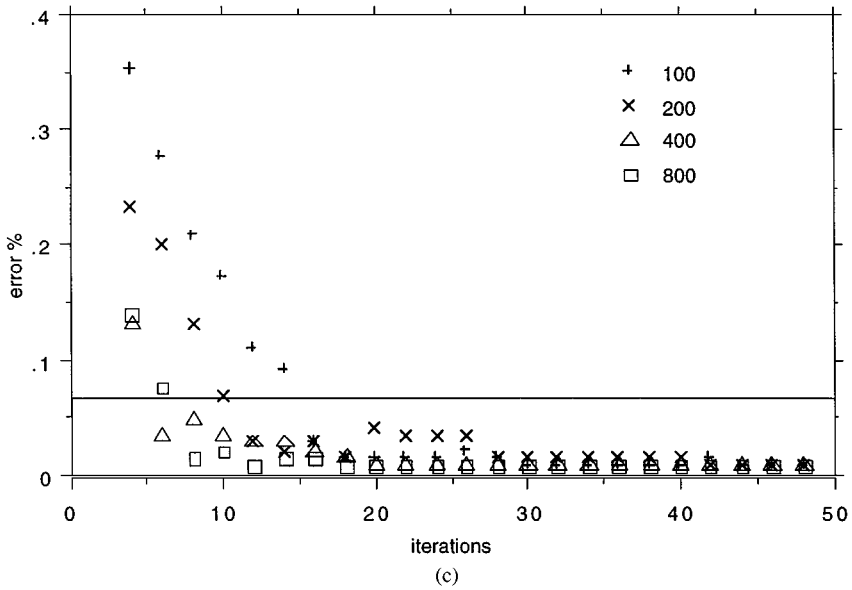

(a)

*Figure 1.* Test set error (%): (a) letter data; (b) satellite data; (c) shuttle data; (d) dna data; (e) digit data.

(*Continued on next page.*)



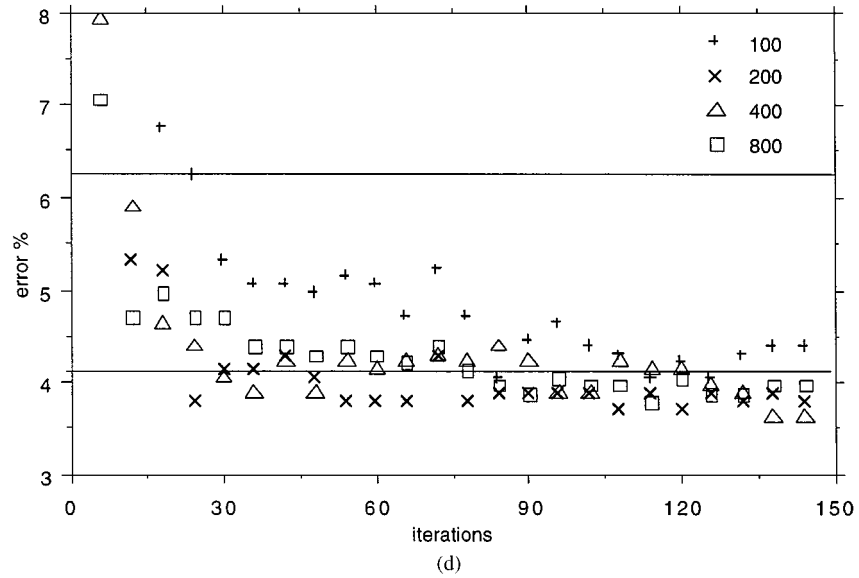

(b)



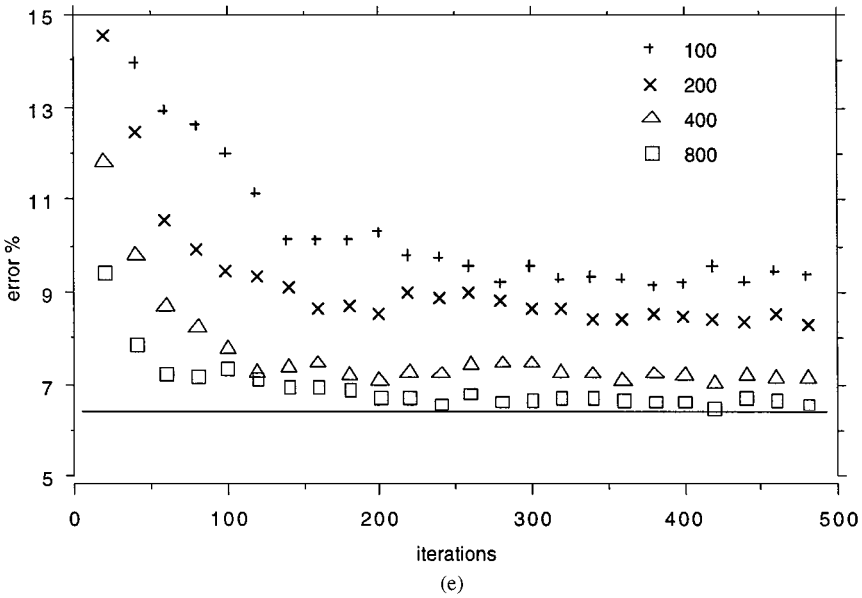

(c)

*Figure 1.* (*Continued*).

(*Continued on next page.*)




(d)




(e)

*Figure 1.* (*Continued*).

(i) The test set error falls rapidly as the number $k$ of iterations increases, and then reaches an asymptotic value for large $k$.

(ii) The accuracy in pasting Ivotes together is similar to that of Adaboost, especially for the larger values of $N$. For instance, the test set error percentages at the end of the $N = 800$ runs are: letters 3.8%, satellite 8.7%, shuttle .007%, dna 3.8%, digit 6.5%.

(iii) In all of the data sets, using even the smallest training sets ($N = 100, 200$) gave test set error comparable to or lower than 10cv-CART after a small number of iterations.
(iv) The effect of increasing $N$ is data dependent. In the dna and shuttle data sets, $N = 100$ gives the same accuracy as $N = 800$. Generally, the asymptotic value is approached faster in the large $N$ runs.
(v) In all data sets, the major decrease in test set error has occurred by 100 iterations. In the letters data set, it appears as though the error is decreasing out to $K = 1000$, although the decrease is small after $K = 200$. In the other data sets, the decrease has stopped by $K = 200$.

***2.2.2. Monitoring and selecting using the OB estimates.*** Suppose that the out-of-bag test set estimates $e^{\mathrm{OB}}(k)$ are used to monitor the pasting procedures in the sense that we stop when the values of $e^{\mathrm{OB}}(k)$ become flat and select the pasted classifier corresponding to the lowest value of $e^{\mathrm{OB}}(k)$ seen to date. Then if we decide to stop after $k$ iterations, the true test set error will be $e^{\mathrm{TS}}(h(k))$ where $h(k) = \mathrm{argmin}\{e^{\mathrm{OB}}(h), h = 1, \ldots, k\}$. The loss in using this method depends on how close or far apart the values of $e^{\mathrm{TS}}(k)$ and $e^{\mathrm{TS}}(h(k))$ are.

Figure 2(a–e) give plots of $e^{\mathrm{OB}}(k)$, $e^{\mathrm{TS}}(k)$, and $e^{\mathrm{TS}}(h(k))$ vs. $k$ for each of the five data sets for $N = 200$. The plot of $e^{\mathrm{OB}}(k)$ can be recognized as the noisiest. The plots of $e^{\mathrm{TS}}(k)$ and $e^{\mathrm{TS}}(h(k))$ lie of top of each other. In all five data sets $e^{\mathrm{TS}}(k)$ and $e^{\mathrm{TS}}(h(k))$ differ very little. Using the out-of-bag estimates to select a pasted classifier works well. However, one thing that emerges from these graphs is that while $e^{\mathrm{OB}}$ tracks $e^{\mathrm{TS}}$ quite well as $k$ increases, it may be systematically low or high.

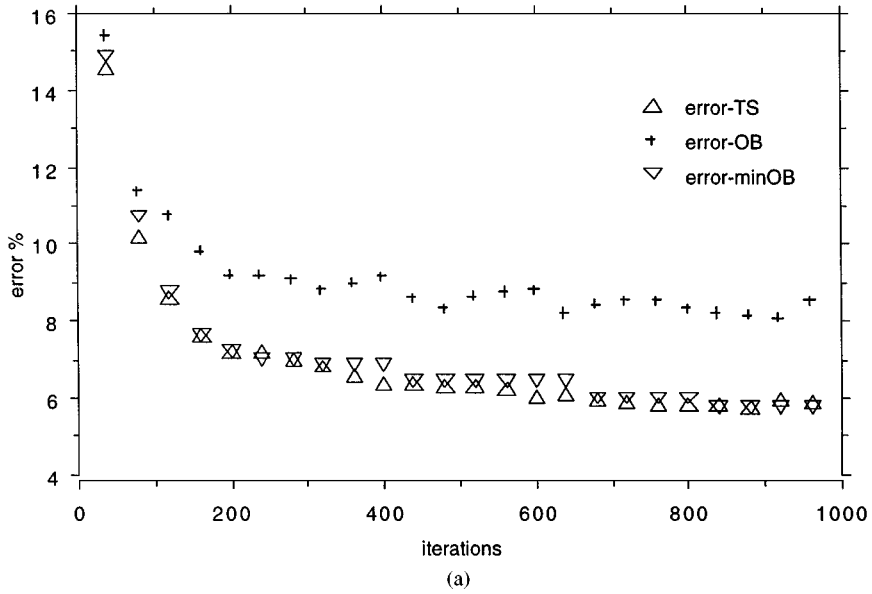
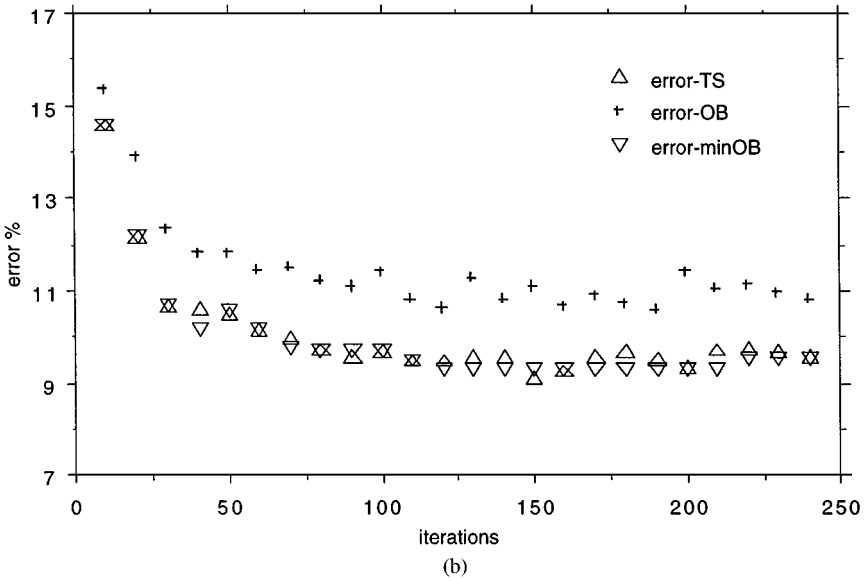
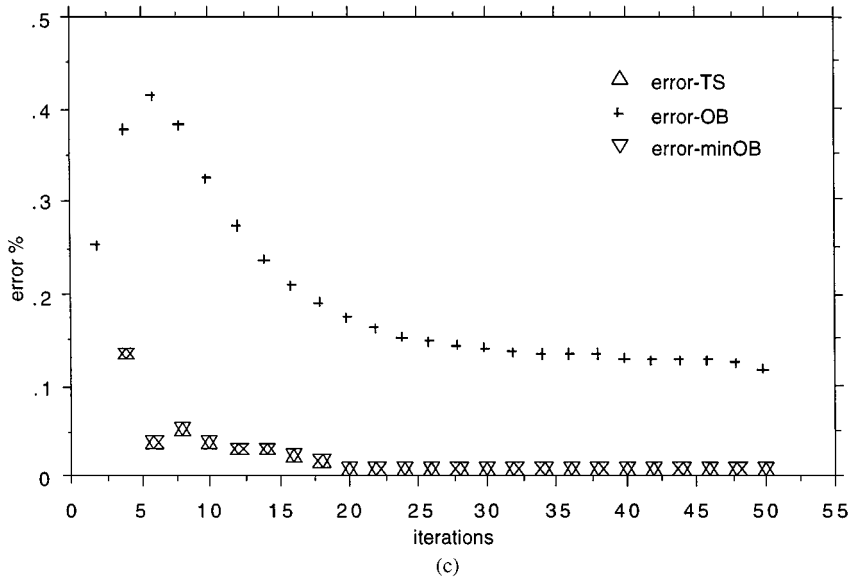



*Figure 2.* Test set, OB, and minimum OB error estimates: (a) letter data; (b) satellite data; (c) shuttle data; (d) dna data; (e) digit data.

(*Continued on next page.*)



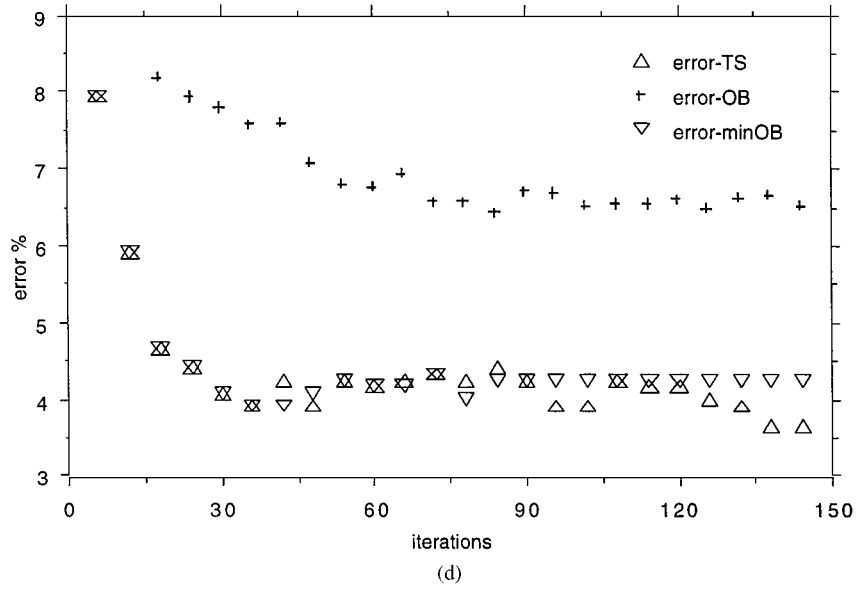

(b)



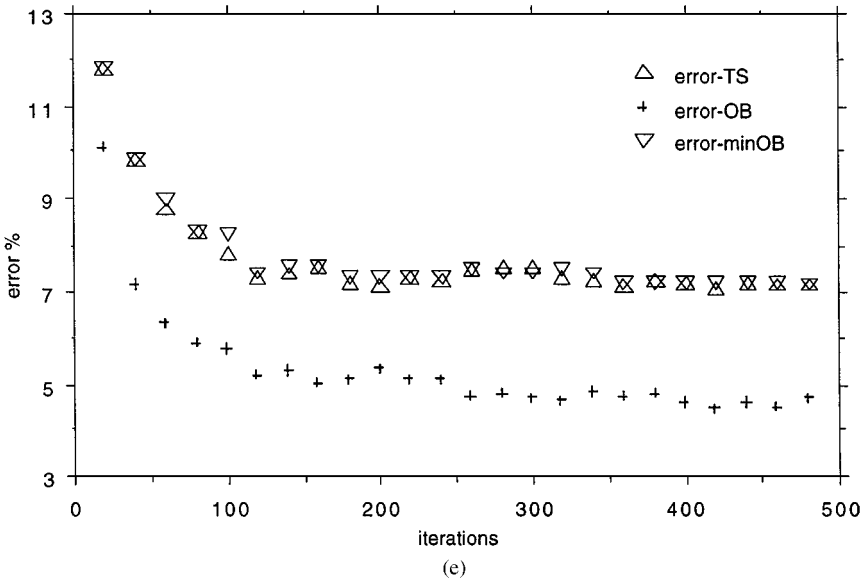

(c)

*Figure 2.* (*Continued*).

Initially, we thought that this bias might be due to the fact that for the same classifier, the "true" test set classification error may differ from the "true" training set classification error. That is, the data sets used above come already separated into training and test sets. It may be that the test set is intrinsically either more or less difficult to classify than the training




(d)




(e)

*Figure 2.* (*Continued*).

set. We ran a test of this hypothesis by interchanging test and training sets. The result is that if the OB estimate is biased high on the training set, it is also biased high on the test set. Our conclusion, after additional research, is that the bias comes from some complex interaction of the data set and the importance sampling.

In the initial stages of pasting, the out-of-bag estimates are usually too high. One obvious cause is that since $e^{\mathrm{OB}}(k)$ is a smoothed version of past out-of-bag estimates, it reflects the earlier and higher values of the error. The less obvious is this: in data sets of moderate size with low misclassification error, examples that are prone to be misclassified are used over and over again in the Iprecinct training sets. These examples will tend to be out-of-bag in a relatively small number of the Ivotes. Therefore, their misclassification rate will be elevated. Another question in using Ivotes is how big to take $N$. In general, it seems that the bigger, the better. But taking $N$ larger also slows down the compute time. The out-of-bag estimates can be used to resolve this issue, since modulo a possible offset due to systematic bias, they will track the true test set error consistently. That is, looking at the out-of-bag estimates for different $N$ will give a fairly accurate idea of how much one buys by using larger $N$.

***2.2.3. Compute times.*** Table 3 gives the compute times per 100 iterations for the five data sets by training set size. The times are scaled to a SUN Ultrasparc 2 and do not include the time to load the data.

These times were computed from the total elapsed time of the run. Thus, if the run was 500 iterations, the time was divided by 5.

### *2.3. Pasting Rvotes*

Pasting Rvotes does not work as well as pasting Ivotes. To illustrate, random sampling was used on the five data sets to form training sets of size $N = 200$. The number of iterations was set at the upper limits specified in Section 2.2. The final test set error was divided by the corresponding test set error for pasting Ivotes. These ratios are given in Table 4.

These ratios are disappointingly large and show that pasting Rvotes is not competitive with pasting Ivotes in terms of accuracy. Still, except for the shuttle data, pasting Rvotes has lower test set error than 10cv-CART.

*Table 3.* Compute time per 100 iterations (min).

| Data set | $N = 100$ | 200 | 400 | 800 |
|---|---|---|---|---|
| letters | 15 | 0.24 | 0.59 | 0.84 |
| satellite | 0.06 | 0.12 | 0.26 | 0.55 |
| shuttle | 0.40 | 0.48 | 0.62 | 0.70 |
| dna | 0.10 | 0.18 | 0.43 | 0.47 |
| digit | 0.34 | 0.92 | 1.94 | 3.61 |

*Table 4.* Ratio of test set errors (Rvotes/Ivotes).

| letters | satellite | shuttle | dna | digit |
|---|---|---|---|---|
| 2.41 | 1.32 | 73.95 | 1.26 | 1.68 |

### *2.4. Of-bag can't be ignored*

Keeping track of which examples are in and out-of-bag is a simple but extra complication. A natural question is what happens if this complication is ignored and all incoming examples treated as if they had never been seen—that is, everything is out-of-bag. If the database is semi-infinite, so that duplicate examples in the training sets used in the pasting is infrequent, then this works. But if the database is finite in the sense that duplication is not infrequent, then performance degenerates.

One consequence is that the error, being a resubstitution estimate, is increasingly biased low and generally goes to zero as the iterations continue. Then the Iprecincts have to search more instances to find enough misclassified ones and the procedure bogs down. For instance, using the letters data and $N = 400$, it bogged down at around 90 iterations with a test set error of 10.3, almost twice as large as the error gotten using out-of-bag.

## 3. Minimizing disk access—An alternative version

A point strongly made by the referees is that the timings given in Section 2 excluded the many random disk accesses needed and that the time taken by these would swamp the cpu times. This point was valid and to deal with it, the following algorithm was constructed. Let a record consist of all data for a single instance.

(a) read a record
if at eof, rewind
check to see if instance is acceptable
if the number of instances accepted is $<N$, goto (a)
construct tree
goto (a)

The algorithm stops after a specified number of trees have been constructed or after a specified number of epochs, where an epoch consists of running through the entire database from beginning to a rewind.

The records are read sequentially, so no random accesses are required. To get an idea of the times needed in a larger database, one million instances of synthetic data with 61 input variables and 10 classes was generated, comprising about 250 Mbyte. For a description of this data, see the Appendix. We set $N = 1000$ and ran for one epoch with 310 trees constructed. On a 10,000 instance test set the error rate dropped to 17.1%. (Constructing a single tree using 100,000 instances resulted in 21.1% test set error). We kept track of the disk read and access times, the time to select instances for the training sets and the time for tree construction. These are tabled below for one epoch.

The machine used is a 250 MHz Macintosh. The total time for the epoch was about 50 min, and the dominant factor is not the disk reads, but the selection cpu usage. The select time is sublinear at the first, and then becomes linear in the number of trees. The disk read and tree construction times are linear. Let $R = N_B/N$. Per epoch the select time $T_S$ is proportional to $2eN\log(N)R^2$. The tree time $T_T$ is proportional to $2eM\log(N)N_B$. Dividing gives $T_T/T_S = cM/R$ (see Appendix). From Table 5, it appears that $c$ is about 3.

*Table 5*. Accumulated times—one epoch (sec).

| Disk time | Selection time | Tree time | Total |
|---|---|---|---|
| 797 | 1829 | 333 | 2949 |

### *3.1. Scaling up*

To get some idea of how the above algorithm scales up, assume a database with $N_B = 10^8$ records such that each record contains data for $M = 10^3$ variables. Assuming numerical variables, this is about half a terabyte. Also assume that $2e = .1$. From Table 5, it took 333 seconds to construct trees for one epoch with $e = 0.2$, $M = 61$, $N = 1000$, and $N_B = 10^6$. Thus, the tree construction time for an epoch of the larger data set using $N = 10^5$ is about $T_T = 62.5$ h on my Macintosh. Making the assumption that we are using a server five times as fast as my machine gives an epoch tree building time of about 12 h.

Using $cM/R$ for the ratio $T_T/T_S$ with $c = 3$ leads to $T_S = .33T_T$. So add 4 h to the running time. Allow about the same magnitude for disk read time, say 16 h. Then, the time needed is about 32 h per epoch—reasonable for a half terabyte database. This is only about double the time needed to sequentially access the entire database. No other algorithm that accesses the entire database can improve on this time by more than 50%.

These results depend on the $MN\log(N)$ time for tree construction. The Appendix gives justification. One consequence of the $MN\log(N)$ time is that even if there was a machine with a terabyte of RAM, constructing a tree using the entire database would take over 11 days on a server five times as fast as my machine.

Here is how memory scales up. Keeping track of $N_L$ and $nc(1), \ldots, nc(J)$ requires about $2JN_B$ bytes. For $J = 2$, this comes to about 0.4 of a Gbyte. The training set consists of $10^5$ records, each containing 1000 4-byte variables—another 0.4 Gbyte. Storing all trees in an epoch costs about $N_B$ bytes—100 Mbytes. The only memory requirements that expand as we go to multiple epochs is the number of trees stored. But this is the least memory extensive requirement. Since gigabyte servers are getting common nowadays, the scaling on memory is reasonable.

### *3.2. Accuracy of the alternative*

An important question is how much accuracy is lost using this less randomized alternative (version 2). Our experimental results indicate that the loss is small. We ran version 2 on the databases used in Section 2. Training set sizes of 800 were used and the algorithm run for the same number of trees as used by version 1 in reaching the optimum result. A comparison of the test set errors at the end of the runs is given in Table 6.

### *3.3. A caveat*

Version 2 will not work if the database has a highly non-uniform distribution of classes. For instance, if all class 1 instances come first in the database. If this structure is known, it

*Table 6.* Comparison of test set error (%).

| Data set | Version 1 | Version 2 |
|---|---|---|
| letters | 3.8 | 4.3 |
| satellite | 8.7 | 9.4 |
| shuttle | 0.007 | 0.000 |
| dna | 3.8 | 5.4 |
| digit | 6.5 | 6.4 |

can be dealt with using additional disk accesses. For instance, a sequential disk read could be used to accept $N/2$ class 1 instances, and then jump to the last accessed class 2 instance and do a sequential read to accept the other $N/2$ instances. This costs two disk accesses per tree constructed.

## 4. On-line learning

In situations where there is a steady flow of new examples being formed, incremental or on-line learning research has focused on the continual updating of a prespecified architecture. For instance, given a flow of examples how is a neural net with specified architecture updated as each new example occurs? Or given a binary tree structure, find efficient ways to update the tree as new examples are added. There has been research on incremental decision tree building starting with Utgoff (1989).

What we propose instead is to do on-line learning by the steady pasting on of new Ivotes. Thus the architecture grows as the information flows in. The drawback is that the storage requirements cannot be set in advance. Ivotes are added until an asymptote is reached. But this is offset by two important advantages—accuracy and speed. As seen in the previous section, pasting Ivotes gives generally low test set error. Further, the compute times required per example are small.

In on-line learning the possibility of a sampled example being sampled again is zero. This simplifies the algorithms. In forming test set error estimates, one can assume that none of the incoming examples have been previously used in a training set. Similarly, in deciding whether an incoming example is correctly classified or not, we can assume that it has not previously been used as a training example.

The on-line procedure generates training sets of size $N$ in the way similar to the Section 2 method. But instead of dealing with a fixed database $D$, there is a flow of incoming examples. As the new examples come in, they are checked to see how they are classified by the current pasted together classifier. If misclassified, they are accepted into the training set. If not, they are accepted with probability $e^{\mathrm{TR}}/(1-e^{\mathrm{TR}})$. Continue this procedure until there are $N$ examples in the training set. The error estimates $e^{\mathrm{TR}}(k)$ are computed as the smoothed version of $r(k)$, where $r(k)$ is the proportion of misclassified examples found in forming the $k$th training set.

To study on-line performance, 10 class, 61 input synthetic data was manufactured (see Appendix). Before going on-line, a test set of 5000 examples was generated. At the $k$th

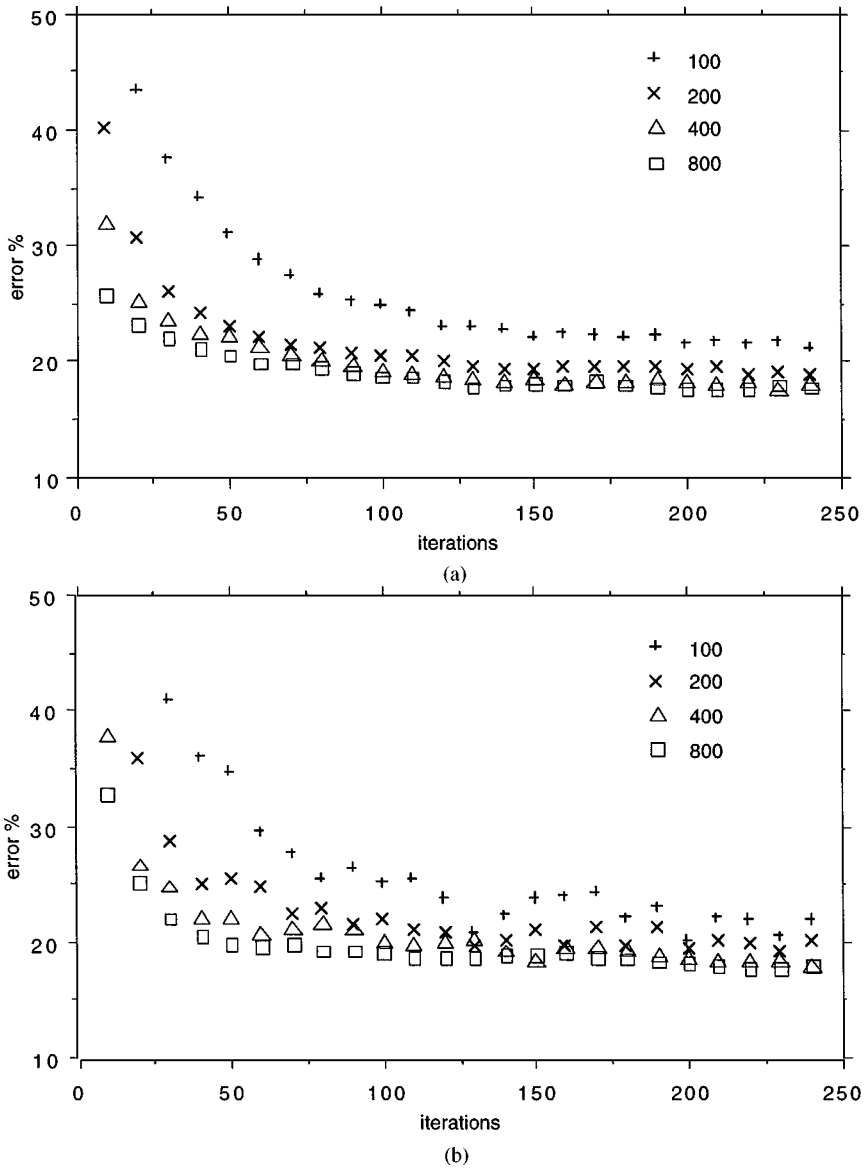



*Figure 3.* On-line synthetic data: (a) test set error (%); (b) training set error estimates (%).

iteration, both $e^{\mathrm{TS}}(k)$ and $e^{\mathrm{TR}}(k)$ are computed. Again, we use $N = 100, 200, 400, 800$. Figure 3(a) shows the test set errors vs. the number of iterations out to 250 iterations. All the test set plots show a rapid decrease to an error of about 20% followed by a slow decrease. Although we stopped at 250 iterations, the test set error was still slowly decreasing. To see what happens if we kept going, we did the $N = 400$ pasting out to 750 iterations. The

test set error dropped from 17.7 at 250 to 16.6 at 750. In application, there is usually no test set available, and the stopping decision must be made on the basis of the $e^{\mathrm{TR}}$ values. Figure 3(b) plots the values of $e^{\mathrm{TR}}$ for the 250 iterations. The values are noisy and need more smoothing, but generally agree with the test set values.

### *4.1. A modification to bound compute times and memory requirements*

At the $(k+1)$st iteration each candidate example for the training set has to be passed down $k$ trees. The compute time to do this for the $k$th iteration increases linearly with $k$, and the total compute time increases quadratically with $k$. Furthermore, the memory used grows unboundedly.

This is not as problematic with finite data sets, since the out-of-bag condition bounds the number of trees used to classify each candidate example and the size of the run is limited. To explore a possible remedy for the problem in on-line learning, we revised the procedure so that the Iprecinct training set is selected based only on the misclassifications in the pasting together of the last $K_B$ trees where $K_B$ is set by the user. The compute times now increase linearly. At 500 iterations the total compute time using $K_B = 100$, and $N = 400$ is 8.3 min compared to 22.0 min for the unrestricted procedure.

The final test set error is similar—17.2% for the unrestricted vs. 17.4% for the restricted version. One drawback is that the $e^{\mathrm{TR}}$ estimates in the restricted arcing now reflect the accuracy of the last $K_B$ Ivotes, instead of all Ivotes. Therefore they have a pessimistic bias. In the original procedure, the final value of $e^{\mathrm{TR}}$ is 18.0%. In the restricted procedure it is 19.8%. Further experiments are needed to see if the $K_B$ restriction gives generally satisfactory results and whether the pessimistic bias in $e^{\mathrm{TR}}$ can be corrected. If it works, then the memory requirements for the on-line learning can be bounded.

## 5. Comments and conclusions

Our prime conclusion is that pasting small Ivotes together is a promising approach to constructing classifiers for large data sets and for on-line learning. On the data sets we experimented with, it is fast, accurate, and has reasonable memory requirements. It will scale up to terabyte databases. Its performance raises interesting issues and possibilities.

The increase in accuracy using Ivotes as compared to Rvotes is newsworthy. It emphasizes that in classification, concentrating on the examples near the classification boundaries pays off in terms of reduced generalization error. The importance sampling approach used here is differs from the Freud-Schapire algorithm. In Adaboost sampling weights for the entire training set are updated in terms of the classifications done by the last classifier constructed. In pasting Ivotes, the sampling for the new training set is done on the basis of the classifications of the entire past sequence of classifiers and the current estimate of test set error.

The process of creating Iprecincts does not depend on the classifier used—it is simply a method of forming the successive training sets. This leads to a question that will be explored in the near future: suppose that there are a number of classifications methods available in our repertoire, i.e., trees, nearest neighbor, neural nets, etc. Suppose that each of these is

run on the $K$th Iprecinct and one is selected to paste onto the previous classifiers. Can this significantly upgrade performance over the use of a single type of classifier?

## 6. Related work for large databases

There has been much interest lately on prediction in large databases. A good survey of is given by Provost and Kolluri (1997a, 1997b). Interesting related work appears in Chan and Stolfo (1997a, 1997b). Their approach, called meta-learning, is to split the data set into pieces, run a classifier on each piece and then to grow a tree that arbitrates or combines the different classifiers. Meta-learning is shown to give good accuracy on two moderate sized databases, but no timings are given.

Provost and Hennessey (1996) distribute the data over multiple workstations, run a rule learning program on each and then use an algorithm to select a subset of the rules generated. Timings were done on what they characterized as a massive database—over 1,000,000 records with 31 variables per record. Using 5 Sparc10 workstations, the run took about 1200 sec. Since their database is about half of the size of the synthetic database used in Section 3, the times are comparable. Modifications of the basic algorithm are suggested that cut the computing time significantly. No data was given concerning the accuracy of the basic or modified algorithm. This paper also contains a nice summary of earlier work.

Breiman and Spector (1994) distributed construction of CART out to a workstation network where each workstation owned a subset of the variables. The vector of the values of each variable was sorted once and converted to ranks which eliminated the need for further sorting at the nodes. At each node, each workstation searched its variable rankings for the best split and reported the decrease in Gini due to that split to the master machine. The master machine reported the best split to all slave machines, and they rearranged the their variable rank values to correspond to the two new nodes. Unfortunately, with a 10 Mbyte network, the communication time proved to be a bottle neck and performance degenerated if more than five machines were used. We plan to rerun on our recent 100 Mbyte network.

Shafer, Agrawal, and Mehta (1996) design SPRINT—a clever parallel version of decision tree construction based also on sort once at the top. To get timings they generated a synthetic 1,600,000 instance database with 9 variables per instance. Using a 16 node IBM SP 2 Model 9076 with the cpu's running at 62.5 MHz, growing a tree took 375 sec. On the same database pasting Ivotes, using training sets of size 1600, took 242 sec for an epoch (22 trees) on the 250 MHz Macintosh. Of this 242 sec, 71% was disk read time, 27% the select time and 2% the tree construction time. Their paper does not specify accuracies so comparison is not possible.

## Appendix

(a) *Synthetic data*: Let $w(k, m)$ be a triangular function in $m$, with a maximum of 10 at $m = 10k$, becoming zero at $10k - 10, 10k + 10$, and zero for $m$ outside this range.

There are 10 distinct subsets of size 3 of the integers {1, 2, 3, 4, 5}. Assigning each of these subsets to a class and denote by $T(j)$ the subset assigned to class $j$. Then the 61 dimensional inputs corresponding to class $j$ are given by:

$$x(m) = \sum_{k \in T(j)} z_k w(k, m) + u_m$$

where the $u_m$ are independent mean zero normals with sd = 10, and the $z_k$ are independent mean zero normals with sd = 1.

The data for each of the classes has a multivariate 61-dimensional mean-zero normal distribution. The optimal classification scheme for this data is quadratic discrimination, and the curved classification boundaries are hard for trees to approximate.

(b) *Tree construction is MNlog(N) cpu time*: My present version of CART, which dates from Breiman and Spector (1994) sorts all variables only at the root node using Quicksort and replaces them by their ranks. This takes about $MN \log(N)$ flops. Then at each node, the best split is found by a sequential search through the ranks of each variable. This takes time about $M^*$(node population). At each tree depth, the flops needed to split each node at that depth is $MN$. Assuming a balanced tree, the tree grows to depth $\log(N)$ adding another $MN \log(N)$ flops.

Some empirical support comes from this fact—evaluate the constant $c$ in $c\,MN \log(N)$ using a training set of size 1000 of my 61 variable synthetic data. Consider running on 10,000 instances of the 9-variable synthetic data used in the Shafer, Agrawal, and Mehta (1996) paper cited in Section 6. Then the predicted time is 2.1 sec. The actual time is 1.5 sec. The faster than predicted run time on the SPRINT data was probably due to the many tied values in each variable. CART does not compute the floating point Gini update on values tied with the value ahead. So the effective number of operations per tree level is smaller than $MN$.

(c) *Select time*: Assume $e < 1/2$. The expected number of picks until $K$ incorrectly classified examples are selected is $K/e$. So the number of picks until $N/2$ examples are selected is $N/(2e)$. But in selecting these $N/2$ incorrects, $N/2$ corrects are also gathered. Therefore, the total number of selections needed for a training set of size $N$ is about $L = N/(2e)$. So in constructing each tree, $L$ instances are drawn at random out of $N_B$.

For any given instance, the probability that it appeared in the last draw is $P = L/N_B$. The probability that it occurred last $I$ draws ago is $P(1-P)I-1$. The expected number of draws ago that it last occurred is $1/P$. On average, each instance used in constructing the current tree has to be run through $1/P$ trees. It takes about $\log(N)$ flops to run an instance down a tree built using $N$ instances. To construct a tree, $L$ instances need to be checked. Therefore, it takes time proportional to $\log(N)L/P = \log(N)NR$ to check the instances needed to grow a tree, where $R = N_B/N$. Since construction time for each tree is proportional to $MN \log(N)$, this results in $T_T/T_S = cM/R$.

**Acknowledgments**

I'm grateful to Yoav Freund and Harold Drucker for numbers of suggestions that led to improvements on the first draft. The comments of the referees and an editor were especially

helpful in making this a better paper. This research was partially supported by NSF Grant 1-444063-21445.

## References

Breiman, L. (1996). Out-of-bag estimation, available at ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation.ps.

Breiman, L. (1998). Arcing classifiers. *Annals of Statistics*, *26*, 801–824.

Breiman, L., Freidman, J., Olshen, R., & Stone, C. (1984). *Classification and regression trees*. Wadsworth.

Breiman, L., & Spector, P. (1994). Parallelizing cart using a workstation network. *Proceedings Annual American Statistical Association Meeting*, San Francisco, available at ftp.stat.berkeley.edu/usrs/breiman/pcart.ps.Z.

Chan, P., & Stolfo, S. (1997a). Scalability of hierarchical meta-learning on partitioned data, submitted to the Journal of Data Mining and Knowledge Discovery, available at www.cs.fit.edu/∼pkc/papers/dmkd-scale.ps.

Chan, P., & Stolfo, S. (1997b). On the accuracy of meta-learning for scalable data mining. *Journal of Intelligent Information Systems*, *9*, 5–28.

Drucker, H., & Cortes, C. (1996). Boosting decision trees. *Neural information processing 8* (pp. 479–485). Morgan Kaufmann.

Freund, Y., & Schapire, R. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. http://www.research.att.com/orgs/ssr/people/yoav or http://www.research.att.com/orgs/ssr/people/schapire.

Freund, Y., & Schapire, R. (1996). Experiments with a new boosting algorithm. *Machine Learning: Proceedings of the Thirteenth International Conference* (pp. 148–156).

Michie, D., Spiegelhalter, D., & Taylor, C. (1994). *Machine learning, neural and statistical classification*. London: Ellis Horwood.

Provost, F.J., & Hennessey, D. (1996). Scaling up: Distributed machine learning with cooperation. *Proceedings AAAI-96*.

Provost, F.J., & Kolluri, V. (1997a). A survey of methods for scaling up inductive learning algorithms. *Accepted by Data Mining and Knowledge Discovery Journal: Special Issue on Scalable High-Performance Computing for KDD* available at www.pitt.edu/∼uxkst/survey-paper.ps.

Provost, F.J., & Kolluri, V. (1997b). Scaling up inductive algorithms: An overview. *Proc. of Knowledge Discovery in Databases* (Vol. KDD'97, pp. 239–242).

Quinlan, J.R. (1996). Bagging, boosting, and C4.5. *Proceedings of AAAI'96 National Conference* (Vol. 1, pp. 725–730).

Schapire, R.E. (1990). The strength of weak learnability. *Machine Learning*, *5*(2), 197–226.

Shafer, J., Agrawal, R., & Mehta, M. (1996). SPRINT: A scalable parallel classifier for data mining. *Proceedings of the 22nd VLDB Conference* (pp. 544–555).

Tibshirani, R. (1996). *Bias, variance, and prediction error for classification rules* (Technical Report). Statistics Department, University of Toronto.

Utgoff, P. (1989). Incremental induction of decision trees. *Machine Learning*, *4*, 161–186 .

Wolpert, D.H., & Macready, W.G. (1996). An efficient method to estimate bagging's generalization error. *Machine Learning*, to appear.

Received September 26, 1997
Accepted September 3, 1998

# Map-Reduce for Machine Learning on Multicore

**Cheng-Tao Chu** * chengtao@stanford.edu

**Sang Kyun Kim** * skkim38@stanford.edu

**Yi-An Lin** * ianl@stanford.edu

**YuanYuan Yu** * yuanyuan@stanford.edu

**Gary Bradski** *† garybradski@gmail

**Andrew Y. Ng** * ang@cs.stanford.edu

**Kunle Olukotun** * kunle@cs.stanford.edu

* CS. Department, Stanford University 353 Serra Mall,
Stanford University, Stanford CA 94305-9025.
† Rexee Inc.

## Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to *many* different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a *single* algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain "summation form," which allows them to be easily parallelized on multicore computers. We adapt Google's map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

## 1 Introduction

Frequency scaling on silicon—the ability to drive chips at ever higher clock rates—is beginning to hit a power limit as device geometries shrink due to leakage, and simply because CMOS consumes power every time it changes state [9, 10]. Yet Moore's law [20], the density of circuits doubling every generation, is projected to last between 10 and 20 more years for silicon based circuits [10]. By keeping clock frequency fixed, but doubling the number of processing cores on a chip, one can maintain lower power while doubling the speed of many applications. This has forced an industry-wide shift to multicore.

We thus approach an era of increasing numbers of cores per chip, but there is as yet no good framework for machine learning to take advantage of massive numbers of cores. There are many parallel programming languages such as Orca, Occam ABCL, SNOW, MPI and PARLOG, but none of these approaches make it obvious how to parallelize a particular algorithm. There is a vast literature on distributed learning and data mining [18], but very little of this literature focuses on our goal: A general means of programming machine learning on multicore. Much of this literature contains a long

and distinguished tradition of developing (often ingenious) ways to speed up or parallelize *individual* learning algorithms, for instance cascaded SVMs [11]. But these yield no general parallelization technique for machine learning and, more pragmatically, specialized implementations of popular algorithms rarely lead to widespread use. Some examples of more general papers are: Caregea et. al. [5] give some general data distribution conditions for parallelizing machine learning, but restrict the focus to decision trees; Jin and Agrawal [14] give a general machine learning programming approach, but only for shared memory machines. This doesn't fit the architecture of cellular or grid type multiprocessors where cores have local cache, even if it can be dynamically reallocated.

In this paper, we focuses on developing a general and exact technique for parallel programming of a large class of machine learning algorithms for multicore processors. The central idea of this approach is to allow a future programmer or user to speed up machine learning applications by "throwing more cores" at the problem rather than search for specialized optimizations. This paper's contributions are:

$(i)$ We show that any algorithm fitting the Statistical Query Model may be written in a certain "summation form." This form does not change the underlying algorithm and so is not an approximation, but is instead an exact implementation. $(ii)$ The summation form does not depend on, but can be easily expressed in a map-reduce [7] framework which is easy to program in. $(iii)$ This technique achieves basically linear speed-up with the number of cores.

We attempt to develop a pragmatic and general framework. What we do **not** claim:

$(i)$ We make no claim that our technique will necessarily run faster than a specialized, one-off solution. Here we achieve linear speedup which in fact often does beat specific solutions such as cascaded SVM [11] (see section 5; however, they do handle kernels, which we have not addressed). $(ii)$ We make no claim that following our framework (for a *specific* algorithm) always leads to a novel parallelization undiscovered by others. What is novel is the larger, broadly applicable framework, together with a pragmatic programming paradigm, map-reduce. $(iii)$ We focus here on exact implementation of machine learning algorithms, not on parallel approximations to algorithms (a worthy topic, but one which is beyond this paper's scope).

In section 2 we discuss the Statistical Query Model, our summation form framework and an example of its application. In section 3 we describe how our framework may be implemented in a Google-like map-reduce paradigm. In section 4 we choose 10 frequently used machine learning algorithms as examples of what can be coded in this framework. This is followed by experimental runs on 10 moderately large data sets in section 5, where we show a good match to our theoretical computational complexity results. Basically, we often achieve linear speedup in the number of cores. Section 6 concludes the paper.

## 2 Statistical Query and Summation Form

For multicore systems, Sutter and Larus [25] point out that multicore mostly benefits concurrent applications, meaning ones where there is little communication between cores. The best match is thus if the data is subdivided and stays local to the cores. To achieve this, we look to Kearns' Statistical Query Model [15].

The Statistical Query Model is sometimes posed as a restriction on the Valiant PAC model [26], in which we permit the learning algorithm to access the learning problem only through a *statistical query oracle*. Given a function $f(x, y)$ over instances, the statistical query oracle returns an estimate of the expectation of $f(x, y)$ (averaged over the training/test distribution). Algorithms that calculate sufficient statistics or gradients fit this model, and since these calculations may be batched, they are expressible as a sum over data points. This class of algorithms is large; We show 10 popular algorithms in section 4 below. An example that does not fit is that of learning an XOR over a subset of bits. [16, 15]. However, when an algorithm does sum over the data, we can easily distribute the calculations over multiple cores: We just divide the data set into as many pieces as there are cores, give each core its share of the data to sum the equations over, and aggregate the results at the end. We call this form of the algorithm the "summation form."

As an example, consider ordinary least squares (linear regression), which fits a model of the form $y = \theta^T x$ by solving: $\theta^* = \min_\theta \sum_{i=1}^m (\theta^T x_i - y_i)^2$ The parameter $\theta$ is typically solved for by

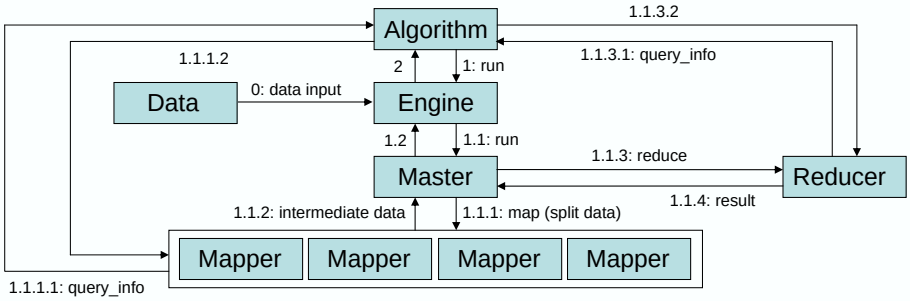



Figure 1: Multicore map-reduce framework

defining the design matrix $X \in \mathbb{R}^{m \times n}$ to be a matrix whose rows contain the training instances $x_1, \ldots, x_m$, letting $\vec{y} = [y_1, \ldots, y_m]^m$ be the vector of target labels, and solving the normal equations to obtain $\theta^* = (X^T X)^{-1} X^T \vec{y}$.

To put this computation into summation form, we reformulate it into a two phase algorithm where we first compute sufficient statistics by summing over the data, and then aggregate those statistics and solve to get $\theta^* = A^{-1} b$. Concretely, we compute $A = X^T X$ and $b = X^T \vec{y}$ as follows: $A = \sum_{i=1}^{m} (x_i x_i^T)$ and $b = \sum_{i=1}^{m} (x_i y_i)$. The computation of $A$ and $b$ can now be divided into equal size pieces and distributed among the cores. We next discuss an architecture that lends itself to the summation form: Map-reduce.

## 3 Architecture

Many programming frameworks are possible for the summation form, but inspired by Google's success in adapting a functional programming construct, map-reduce [7], for wide spread parallel programming use inside their company, we adapted this same construct for multicore use. Google's map-reduce is specialized for use over clusters that have unreliable communication and where individual computers may go down. These are issues that multicores do not have; thus, we were able to developed a much lighter weight architecture for multicores, shown in Figure 1.

Figure 1 shows a high level view of our architecture and how it processes the data. In step 0, the map-reduce engine is responsible for splitting the data by training examples (rows). The engine then caches the split data for the subsequent map-reduce invocations. Every algorithm has its own engine instance, and every map-reduce task will be delegated to its engine (step 1). Similar to the original map-reduce architecture, the engine will run a master (step 1.1) which coordinates the mappers and the reducers. The master is responsible for assigning the split data to different mappers, and then collects the processed intermediate data from the mappers (step 1.1.1 and 1.1.2). After the intermediate data is collected, the master will in turn invoke the reducer to process it (step 1.1.3) and return final results (step 1.1.4). Note that some mapper and reducer operations require additional scalar information from the algorithms. In order to support these operations, the mapper/reducer can obtain this information through the query_info interface, which can be customized for each different algorithm (step 1.1.1.1 and 1.1.3.2).

## 4 Adopted Algorithms

In this section, we will briefly discuss the algorithms we have implemented based on our framework. These algorithms were chosen partly by their popularity of use in NIPS papers, and our goal will be to illustrate how each algorithm can be expressed in summation form. We will defer the discussion of the theoretical improvement that can be achieved by this parallelization to Section 4.1. In the following, $x$ or $x_i$ denotes a training vector and $y$ or $y_i$ denotes a training label.

- **Locally Weighted Linear Regression (LWLR)** LWLR [28, 3] is solved by finding the solution of the normal equations $A\theta = b$, where $A = \sum_{i=1}^{m} w_i(x_i x_i^T)$ and $b = \sum_{i=1}^{m} w_i(x_i y_i)$. For the summation form, we divide the computation among different mappers. In this case, one set of mappers is used to compute $\sum_{subgroup} w_i(x_i x_i^T)$ and another set to compute $\sum_{subgroup} w_i(x_i y_i)$. Two reducers respectively sum up the partial values for $A$ and $b$, and the algorithm finally computes the solution $\theta = A^{-1}b$. Note that if $w_i = 1$, the algorithm reduces to the case of ordinary least squares (linear regression).
- **Naive Bayes (NB)** In NB [17, 21], we have to estimate $P(x_j = k|y = 1)$, $P(x_j = k|y = 0)$, and $P(y)$ from the training data. In order to do so, we need to sum over $x_j = k$ for each $y$ label in the training data to calculate $P(x|y)$. We specify different sets of mappers to calculate the following: $\sum_{subgroup} 1\{x_j = k|y = 1\}$, $\sum_{subgroup} 1\{x_j = k|y = 0\}$, $\sum_{subgroup} 1\{y = 1\}$ and $\sum_{subgroup} 1\{y = 0\}$. The reducer then sums up intermediate results to get the final result for the parameters.
- **Gaussian Discriminative Analysis (GDA)** The classic GDA algorithm [13] needs to learn the following four statistics $P(y), \mu_0, \mu_1$ and $\Sigma$. For all the summation forms involved in these computations, we may leverage the map-reduce framework to parallelize the process. Each mapper will handle the summation (i.e. $\Sigma\ 1\{y_i = 1\}, \Sigma\ 1\{y_i = 0\}, \Sigma\ 1\{y_i = 0\}x_i$, etc) for a subgroup of the training samples. Finally, the reducer will aggregate the intermediate sums and calculate the final result for the parameters.
- **k-means** In k-means [12], it is clear that the operation of computing the Euclidean distance between the sample vectors and the centroids can be parallelized by splitting the data into individual subgroups and clustering samples in each subgroup separately (by the mapper). In recalculating new centroid vectors, we divide the sample vectors into subgroups, compute the sum of vectors in each subgroup in parallel, and finally the reducer will add up the partial sums and compute the new centroids.
- **Logistic Regression (LR)** For logistic regression [23], we choose the form of hypothesis as $h_\theta(x) = g(\theta^T x) = 1/(1 + \exp(-\theta^T x))$ Learning is done by fitting $\theta$ to the training data where the likelihood function can be optimized by using Newton-Raphson to update $\theta := \theta - H^{-1}\nabla_\theta \ell(\theta)$. $\nabla_\theta \ell(\theta)$ is the gradient, which can be computed in parallel by mappers summing up $\sum_{subgroup}(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$ each NR step $i$. The computation of the hessian matrix can be also written in a summation form of $H(j,k) := H(j,k) + h_\theta(x^{(i)})(h_\theta(x^{(i)}) - 1)x_j^{(i)}x_k^{(i)}$ for the mappers. The reducer will then sum up the values for gradient and hessian to perform the update for $\theta$.
- **Neural Network (NN)** We focus on backpropagation [6] By defining a network structure (we use a three layer network with two output neurons classifying the data into two categories), each mapper propagates its set of data through the network. For each training example, the error is back propagated to calculate the partial gradient for each of the weights in the network. The reducer then sums the partial gradient from each mapper and does a batch gradient descent to update the weights of the network.
- **Principal Components Analysis (PCA)** PCA [29] computes the principle eigenvectors of the covariance matrix $\Sigma = \frac{1}{m}\left(\sum_{i=1}^{m} x_i x_i^T\right) - \mu\mu^T$ over the data. In the definition for $\Sigma$, the term $\left(\sum_{i=1}^{m} x_i x_i^T\right)$ is already expressed in summation form. Further, we can also express the mean vector $\mu$ as a sum, $\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$. The sums can be mapped to separate cores, and then the reducer will sum up the partial results to produce the final empirical covariance matrix.
- **Independent Component Analysis (ICA)** ICA [1] tries to identify the independent source vectors based on the assumption that the observed data are linearly transformed from the source data. In ICA, the main goal is to compute the unmixing matrix W. We implement batch gradient ascent to optimize the $W$'s likelihood. In this scheme, we can independently calculate the expression $\begin{bmatrix} 1 - 2g(w_1^T x^{(i)}) \\ \vdots \end{bmatrix} x^{(i)^T}$ in the mappers and sum them up in the reducer.
- **Expectation Maximization (EM)** For EM [8] we use Mixture of Gaussian as the underlying model as per [19]. For parallelization: In the E-step, every mapper processes its subset

of the training data and computes the corresponding $w_j^{(i)}$ (expected pseudo count). In M-phase, three sets of parameters need to be updated: $p(y)$, $\mu$, and $\Sigma$. For $p(y)$, every mapper will compute $\sum_{subgroup}(w_j^{(i)})$, and the reducer will sum up the partial result and divide it by $m$. For $\mu$, each mapper will compute $\sum_{subgroup}(w_j^{(i)} * x^{(i)})$ and $\sum_{subgroup}(w_j^{(i)})$, and the reducer will sum up the partial result and divide them. For $\Sigma$, every mapper will compute $\sum_{subgroup}(w_j^{(i)} * (x^{(i)} - \mu_j) * (x^{(i)} - \mu_j)^T)$ and $\sum_{subgroup}(w_j^{(i)})$, and the reducer will again sum up the partial result and divide them.

- **Support Vector Machine (SVM)** Linear SVM's [27, 22] primary goal is to optimize the following primal problem $\min_{w,b} \|w\|^2 + C\sum_{i:\xi_i>0} \xi_i^p \quad s.t. \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i$ where $p$ is either 1 (hinge loss) or 2 (quadratic loss). [2] has shown that the primal problem for quadratic loss can be solved using the following formula where $sv$ are the support vectors: $\nabla = 2w + 2C\sum_{i \in sv}(w \cdot x_i - y_i)x_i$ & Hessian $H = I + C\sum_{i \in sv} x_i x_i^T$ We perform batch gradient descent to optimize the objective function. The mappers will calculate the partial gradient $\sum_{subgroup(i \in sv)}(w \cdot x_i - y_i)x_i$ and the reducer will sum up the partial results to update $w$ vector.

Some implementations of machine learning algorithms, such as ICA, are commonly done with stochastic gradient ascent, which poses a challenge to parallelization. The problem is that in every step of gradient ascent, the algorithm updates a common set of parameters (e.g. the unmixing $W$ matrix in ICA). When one gradient ascent step (involving one training sample) is updating $W$, it has to lock down this matrix, read it, compute the gradient, update $W$, and finally release the lock. This "lock-release" block creates a bottleneck for parallelization; thus, instead of stochastic gradient ascent, our algorithms above were implemented using batch gradient ascent.

### 4.1 Algorithm Time Complexity Analysis

Table 1 shows the theoretical complexity analysis for the ten algorithms we implemented on top of our framework. We assume that the dimension of the inputs is $n$ (i.e., $x \in \mathbb{R}^n$), that we have $m$ training examples, and that there are $P$ cores. The complexity of iterative algorithms is analyzed for one iteration, and so their actual running time may be slower.[1] A few algorithms require matrix inversion or an eigen-decomposition of an $n$-by-$n$ matrix; we did not parallelize these steps in our experiments, because for us $m >> n$, and so their cost is small. However, there is extensive research in numerical linear algebra on parallelizing these numerical operations [4], and in the complexity analysis shown in the table, we have assumed that matrix inversion and eigen-decompositions can be sped up by a factor of $P'$ on $P$ cores. (In practice, we expect $P' \approx P$.) In our own software implementation, we had $P' = 1$. Further, the reduce phase can minimize communication by combining data as it's passed back; this accounts for the $\log(P)$ factor.

As an example of our running-time analysis, for single-core LWLR we have to compute $A = \sum_{i=1}^{m} w_i(x_i x_i^T)$, which gives us the $mn^2$ term. This matrix must be inverted for $n^3$; also, the reduce step incurs a covariance matrix communication cost of $n^2$.

## 5 Experiments

To provide fair comparisons, each algorithm had two different versions: One running map-reduce, and the other a serial implementation without the framework. We conducted an extensive series of experiments to compare the speed up on data sets of various sizes (table 2), on eight commonly used machine learning data sets from the UCI Machine Learning repository and two other ones from a [anonymous] research group (Helicopter Control and sensor data). Note that not all the experiments make sense from an output view – regression on categorical data – but our purpose was to test speedup so we ran every algorithm over all the data.

The first environment we conducted experiments on was an Intel X86 PC with two Pentium-III 700 MHz CPUs and 1GB physical memory. The operating system was Linux RedHat 8.0 Kernel 2.4.20-

[1] If, for example, the number of iterations required grows with $m$. However, this would affect single- and multi-core implementations equally.

| | single | multi |
|---|---|---|
| LWLR | $O(mn^2 + n^3)$ | $O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$ |
| LR | $O(mn^2 + n^3)$ | $O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$ |
| NB | $O(mn + nc)$ | $O(\frac{mn}{P} + nc \log(P))$ |
| NN | $O(mn + nc)$ | $O(\frac{mn}{P} + nc \log(P))$ |
| GDA | $O(mn^2 + n^3)$ | $O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$ |
| PCA | $O(mn^2 + n^3)$ | $O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$ |
| ICA | $O(mn^2 + n^3)$ | $O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$ |
| k-means | $O(mnc)$ | $O(\frac{mnc}{P} + mn \log(P))$ |
| EM | $O(mn^2 + n^3)$ | $O(\frac{mn^2}{P} + \frac{n^3}{P'} + n^2 \log(P))$ |
| SVM | $O(m^2 n)$ | $O(\frac{m^2 n}{P} + n \log(P))$ |

Table 1: time complexity analysis

| Data Sets | samples (m) | features (n) |
|---|---|---|
| Adult | 30162 | 14 |
| Helicopter Control | 44170 | 21 |
| Corel Image Features | 68040 | 32 |
| IPUMS Census | 88443 | 61 |
| Synthetic Time Series | 100001 | 10 |
| Census Income | 199523 | 40 |
| ACIP Sensor | 229564 | 8 |
| KDD Cup 99 | 494021 | 41 |
| Forest Cover Type | 581012 | 55 |
| 1990 US Census | 2458285 | 68 |

Table 2: data sets size and description

8smp. In addition, we also ran extensive comparison experiments on a 16 way Sun Enterprise 6000, running Solaris 10; here, we compared results using 1,2,4,8, and 16 cores.

### 5.1 Results and Discussion

Table 3 shows the speedup on dual processors over all the algorithms on all the data sets. As can be seen from the table, most of the algorithms achieve more than 1.9x times performance improvement. For some of the experiments, e.g. gda/covertype, ica/ipums, nn/colorhistogram, etc., we obtain a greater than 2x speedup. This is because the original algorithms do not utilize all the cpu cycles efficiently, but do better when we distribute the tasks to separate threads/processes.

Figure 2 shows the speedup of the algorithms over all the data sets for 2,4,8 and 16 processing cores. In the figure, the thick lines shows the average speedup, the error bars show the maximum and minimum speedups and the dashed lines show the variance. Speedup is basically linear with number

| | lwlr | gda | nb | logistic | pca | ica | svm | nn | kmeans | em |
|---|---|---|---|---|---|---|---|---|---|---|
| Adult | 1.922 | 1.801 | 1.844 | 1.962 | 1.809 | 1.857 | 1.643 | 1.825 | 1.947 | 1.854 |
| Helicopter | 1.93 | 2.155 | 1.924 | 1.92 | 1.791 | 1.856 | 1.744 | 1.847 | 1.857 | 1.86 |
| Corel Image | 1.96 | 1.876 | 2.002 | 1.929 | 1.97 | 1.936 | 1.754 | 2.018 | 1.921 | 1.832 |
| IPUMS | 1.963 | 2.23 | 1.965 | 1.938 | 1.965 | 2.025 | 1.799 | 1.974 | 1.957 | 1.984 |
| Synthetic | 1.909 | 1.964 | 1.972 | 1.92 | 1.842 | 1.907 | 1.76 | 1.902 | 1.888 | 1.804 |
| Census Income | 1.975 | 2.179 | 1.967 | 1.941 | 2.019 | 1.941 | 1.88 | 1.896 | 1.961 | 1.99 |
| Sensor | 1.927 | 1.853 | 2.01 | 1.913 | 1.955 | 1.893 | 1.803 | 1.914 | 1.953 | 1.949 |
| KDD | 1.969 | 2.216 | 1.848 | 1.927 | 2.012 | 1.998 | 1.946 | 1.899 | 1.973 | 1.979 |
| Cover Type | 1.961 | 2.232 | 1.951 | 1.935 | 2.007 | 2.029 | 1.906 | 1.887 | 1.963 | 1.991 |
| Census | 2.327 | 2.292 | 2.008 | 1.906 | 1.997 | 2.001 | 1.959 | 1.883 | 1.946 | 1.977 |
| avg. | 1.985 | 2.080 | 1.950 | 1.930 | 1.937 | 1.944 | 1.819 | 1.905 | 1.937 | 1.922 |

Table 3: Speedups achieved on a dual core processor, without load time. Numbers reported are dual-core time / single-core time. Super linear speedup sometimes occurs due to a reduction in processor idle time with multiple threads.

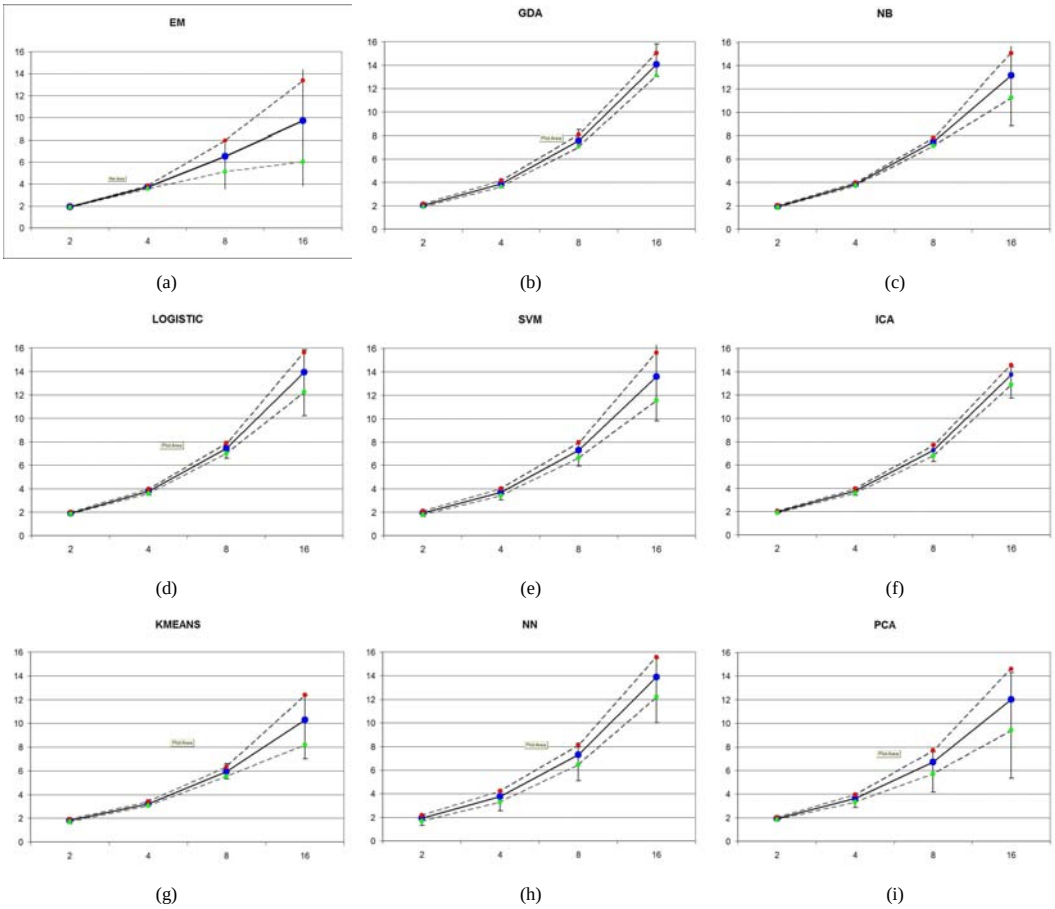



Figure 2: (a)-(i) show the speedup from 1 to 16 processors of all the algorithms over all the data sets. The Bold line is the average, error bars are the max and min speedups and the dashed lines are the variance.

of cores, but with a slope $< 1.0$. The reason for the sub-unity slope is increasing communication overhead. For simplicity and because the number of data points $m$ typically dominates reduction phase communication costs (typically a factor of $n^2$ but $n << m$), we did not parallelize the reduce phase where we could have combined data on the way back. Even so, our simple SVM approach gets about 13.6% speed up on average over 16 cores whereas the specialized SVM cascade [11] averages only 4%.

Finally, the above are runs on multiprocessor machines. We finish by reporting some confirming results and higher performance on a proprietary multicore simulator over the sensor dataset.[2] NN speedup was [16 cores, 15.5x], [32 cores, 29x], [64 cores, 54x]. LR speedup was [16 cores, 15x], [32 cores, 29.5x], [64 cores, 53x]. Multicore machines are generally faster than multiprocessor machines because communication internal to the chip is much less costly.

## 6 Conclusion

As the Intel and AMD product roadmaps indicate [24], the number of processing cores on a chip will be doubling several times over the next decade, even as individual cores cease to become significantly faster. For machine learning to continue reaping the bounty of Moore's law and apply to ever larger datasets and problems, it is important to adopt a programming architecture which takes advantage of multicore. In this paper, by taking advantage of the summation form in a map-reduce

[2]This work was done in collaboration with Intel Corporation.

framework, we could parallelize a wide range of machine learning algorithms and achieve a 1.9 times speedup on a dual processor on up to 54 times speedup on 64 cores. These results are in line with the complexity analysis in Table 1. We note that the speedups achieved here involved no special optimizations of the algorithms themselves. We have demonstrated a simple programming framework where in the future we can just "throw cores" at the problem of speeding up machine learning code.

## Acknowledgments

We would like to thank Skip Macy from Intel for sharing his valuable experience in VTune performance analyzer. Yirong Shen, Anya Petrovskaya, and Su-In Lee from Stanford University helped us in preparing various data sets used in our experiments. This research was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under the ACIP program and grant number NBCH104009.

## References

[1] Sejnowski TJ. Bell AJ. An information-maximization approach to blind separation and blind deconvolution. In *Neural Computation*, 1995.

[2] O. Chapelle. Training a support vector machine in the primal. *Journal of Machine Learning Research (submitted)*, 2006.

[3] W. S. Cleveland and S. J. Devlin. Locally weighted regression: An approach to regression analysis by local fitting. In *J. Amer. Statist. Assoc. 83*, pages 596–610, 1988.

[4] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5(4):618–623, 1976.

[5] A. Silvescu D. Caragea and V. Honavar. A framework for learning from distributed data using sufficient statistics and its application to learning decision trees. *International Journal of Hybrid Intelligent Systems*, 2003.

[6] R. J. Williams D. E. Rumelhart, G. E. Hinton. Learning representation by back-propagating errors. In *Nature*, volume 323, pages 533–536, 1986.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Operating Systems Design and Implementation*, pages 137–149, 2004.

[8] N.M. Dempster A.P., Laird and Rubin D.B.

[9] D.J. Frank. Power-constrained cmos scaling limits. *IBM Journal of Research and Development*, 46, 2002.

[10] P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities and new frontiers. In *ISSCC Tech. Digest*, pages 22–25, 2001.

[11] Leon Bottou Igor Durdanovic Hans Peter Graf, Eric Cosatto and Vladimire Vapnik. Parallel support vector machines: The cascade svm. In *NIPS*, 2004.

[12] J. Hartigan. *Clustering Algorithms.* Wiley, 1975.

[13] T. Hastie and R. Tibshirani. Discriminant analysis by gaussian mixtures. *Journal of the Royal Statistical Society B*, pages 155–176, 1996.

[14] R. Jin and G. Agrawal. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. In *Second SIAM International Conference on Data Mining,*, 2002.

[15] M. Kearns. Efficient noise-tolerant learning from statistical queries. pages 392–401, 1999.

[16] Michael Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[17] David Lewis. Naive (bayes) at forty: The independence asssumption in information retrieval. In *ECML98: Tenth European Conference On Machine Learning*, 1998.

[18] Kun Liu and Hillow Kargupta. Distributed data mining bibliography. *http://www.cs.umbc.edu/ hillol/DDMBIB/*, 2006.

[19] T. K. MOON. The expectation-maximization algorithm. In *IEEE Trans. Signal Process*, pages 47–59, 1996.

[20] G. Moore. Progress in digital integrated electronics. In *IEDM Tech. Digest*, pages 11–13, 1975.

[21] Wayne Iba Pat Langley and Kevin Thompson. An analysis of bayesian classifiers. In *AAAI*, 1992.

[22] John C. Platt. Fast training of support vector machines using sequential minimal optimization. pages 185–208, 1999.

[23] Daryl Pregibon. Logistic regression diagnostics. In *The Annals of Statistics*, volume 9, pages 705–724, 1981.

[24] T. Studt. There's a multicore in your future, http://tinyurl.com/ohd2m, 2006.

[25] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[26] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 3(11):1134–1142, 1984.

[27] V. Vapnik. *Estimation of Dependencies Based on Empirical Data.* Springer Verlag, 1982.

[28] R. E. Welsch and E. KUH. Linear regression diagnostics. In *Working Paper 173, Nat. Bur. Econ. Res.Inc*, 1977.

[29] K. Esbensen Wold, S. and P. Geladi. Principal component analysis. In *Chemometrics and Intelligent Laboratory Systems*, 1987.

# Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.

{jasonbaker,chrisbond,jcorbett,jfurman,akhorlin,jimlarson,jm,yaweili,alloyd,vadimy}@google.com

## ABSTRACT

Megastore is a storage system developed to meet the requirements of today's interactive online services. Megastore blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS in a novel way, and provides both strong consistency guarantees and high availability. We provide fully serializable ACID semantics within fine-grained partitions of data. This partitioning allows us to synchronously replicate each write across a wide area network with reasonable latency and support seamless failover between datacenters. This paper describes Megastore's semantics and replication algorithm. It also describes our experience supporting a wide range of Google production services built with Megastore.

## Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases; H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases*

## General Terms

Algorithms, Design, Performance, Reliability

## Keywords

Large databases, Distributed transactions, Bigtable, Paxos

## 1. INTRODUCTION

Interactive online services are forcing the storage community to meet new demands as desktop applications migrate to the cloud. Services like email, collaborative documents, and social networking have been growing exponentially and are testing the limits of existing infrastructure. Meeting these services' storage demands is challenging due to a number of conflicting requirements.

First, the Internet brings a huge audience of potential users, so the applications must be *highly scalable*. A service can be built rapidly using MySQL [10] as its datastore, but scaling the service to millions of users requires a complete redesign of its storage infrastructure. Second, services must compete for users. This requires *rapid development* of features and fast time-to-market. Third, the service must be responsive; hence, the storage system must have *low latency*. Fourth, the service should provide the user with a *consistent view of the data*—the result of an update should be visible immediately and durably. Seeing edits to a cloud-hosted spreadsheet vanish, however briefly, is a poor user experience. Finally, users have come to expect Internet services to be up 24/7, so the service must be *highly available*. The service must be resilient to many kinds of faults ranging from the failure of individual disks, machines, or routers all the way up to large-scale outages affecting entire datacenters.

These requirements are in conflict. Relational databases provide a rich set of features for easily building applications, but they are difficult to scale to hundreds of millions of users. NoSQL datastores such as Google's Bigtable [15], Apache Hadoop's HBase [1], or Facebook's Cassandra [6] are highly scalable, but their limited API and loose consistency models complicate application development. Replicating data across distant datacenters while providing low latency is challenging, as is guaranteeing a consistent view of replicated data, especially during faults.

Megastore is a storage system developed to meet the storage requirements of today's interactive online services. It is novel in that it blends the scalability of a NoSQL datastore with the convenience of a traditional RDBMS. It uses synchronous replication to achieve high availability and a consistent view of the data. In brief, it provides fully serializable ACID semantics over distant replicas with low enough latencies to support interactive applications.

We accomplish this by taking a middle ground in the RDBMS vs. NoSQL design space: we partition the datastore and replicate each partition separately, providing full ACID semantics within partitions, but only limited consistency guarantees across them. We provide traditional database features, such as secondary indexes, but only those features that can scale within user-tolerable latency limits, and only with the semantics that our partitioning scheme can support. We contend that the data for most Internet services can be suitably partitioned (e.g., by user) to make this approach viable, and that a small, but not spartan, set of features can substantially ease the burden of developing cloud applications.

Contrary to conventional wisdom [24, 28], we were able to use Paxos [27] to build a highly available system that pro-


$5^{th}$ *Biennial Conference on Innovative Data Systems Research (CIDR '11)* January 9-12, 2011, Asilomar, California, USA.

vides reasonable latencies for interactive applications while synchronously replicating writes across geographically distributed datacenters. While many systems use Paxos solely for locking, master election, or replication of metadata and configurations, we believe that Megastore is the largest system deployed that uses Paxos to replicate primary user data across datacenters on every write.

Megastore has been widely deployed within Google for several years [20]. It handles more than three billion write and 20 billion read transactions daily and stores nearly a petabyte of primary data across many global datacenters.

The key contributions of this paper are:

1. the design of a data model and storage system that allows rapid development of interactive applications where high availability and scalability are built-in from the start;
2. an implementation of the Paxos replication and consensus algorithm optimized for low-latency operation across geographically distributed datacenters to provide high availability for the system;
3. a report on our experience with a large-scale deployment of Megastore at Google.

The paper is organized as follows. Section 2 describes how Megastore provides availability and scalability using partitioning and also justifies the sufficiency of our design for many interactive Internet applications. Section 3 provides an overview of Megastore's data model and features. Section 4 explains the replication algorithms in detail and gives some measurements on how they perform in practice. Section 5 summarizes our experience developing the system. We review related work in Section 6. Section 7 concludes.

# 2. TOWARD AVAILABILITY AND SCALE

In contrast to our need for a storage platform that is global, reliable, and arbitrarily large in scale, our hardware building blocks are geographically confined, failure-prone, and suffer limited capacity. We must bind these components into a unified ensemble offering greater throughput and reliability.

To do so, we have taken a two-pronged approach:

- for availability, we implemented a synchronous, fault-tolerant log replicator optimized for long distance-links;
- for scale, we partitioned data into a vast space of small databases, each with its own replicated log stored in a per-replica NoSQL datastore.

## 2.1 Replication

Replicating data across hosts within a single datacenter improves availability by overcoming host-specific failures, but with diminishing returns. We still must confront the networks that connect them to the outside world and the infrastructure that powers, cools, and houses them. Economically constructed sites risk some level of facility-wide outages [25] and are vulnerable to regional disasters. For cloud storage to meet availability demands, service providers must replicate data over a wide geographic area.

### 2.1.1 Strategies

We evaluated common strategies for wide-area replication:

**Asynchronous Master/Slave** A master node replicates write-ahead log entries to at least one slave. Log appends are acknowledged at the master in parallel with transmission to slaves. The master can support fast ACID transactions but risks downtime or data loss during failover to a slave. A consensus protocol is required to mediate mastership.

**Synchronous Master/Slave** A master waits for changes to be mirrored to slaves before acknowledging them, allowing failover without data loss. Master and slave failures need timely detection by an external system.

**Optimistic Replication** Any member of a homogeneous replica group can accept mutations [23], which are asynchronously propagated through the group. Availability and latency are excellent. However, the global mutation ordering is not known at commit time, so transactions are impossible.

We avoided strategies which could lose data on failures, which are common in large-scale systems. We also discarded strategies that do not permit ACID transactions. Despite the operational advantages of eventually consistent systems, it is currently too difficult to give up the read-modify-write idiom in rapid application development.

We also discarded options with a heavyweight master. Failover requires a series of high-latency stages often causing a user-visible outage, and there is still a huge amount of complexity. Why build a fault-tolerant system to arbitrate mastership and failover workflows if we could avoid distinguished masters altogether?

### 2.1.2 Enter Paxos

We decided to use Paxos, a proven, optimal, fault-tolerant consensus algorithm with no requirement for a distinguished master [14, 27]. We replicate a write-ahead log over a group of symmetric peers. Any node can initiate reads and writes. Each log append blocks on acknowledgments from a majority of replicas, and replicas in the minority catch up as they are able—the algorithm's inherent fault tolerance eliminates the need for a distinguished "failed" state. A novel extension to Paxos, detailed in Section 4.4.1, allows local reads at any up-to-date replica. Another extension permits single-roundtrip writes.

Even with fault tolerance from Paxos, there are limitations to using a single log. With replicas spread over a wide area, communication latencies limit overall throughput. Moreover, progress is impeded when no replica is current or a majority fail to acknowledge writes. In a traditional SQL database hosting thousands or millions of users, using a synchronously replicated log would risk interruptions of widespread impact [11]. So to improve availability and throughput we use multiple replicated logs, each governing its own partition of the data set.

## 2.2 Partitioning and Locality

To scale our replication scheme and maximize performance of the underlying datastore, we give applications fine-grained control over their data's partitioning and locality.

### 2.2.1 Entity Groups

To scale throughput and localize outages, we partition our data into a collection of *entity groups* [24], each independently and synchronously replicated over a wide area. The underlying data is stored in a scalable NoSQL datastore in each datacenter (see Figure 1).

Entities within an entity group are mutated with single-phase ACID transactions (for which the commit record is

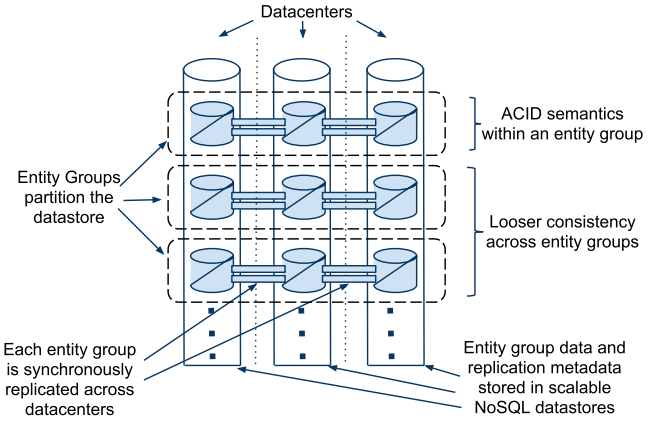



**Figure 1: Scalable Replication**

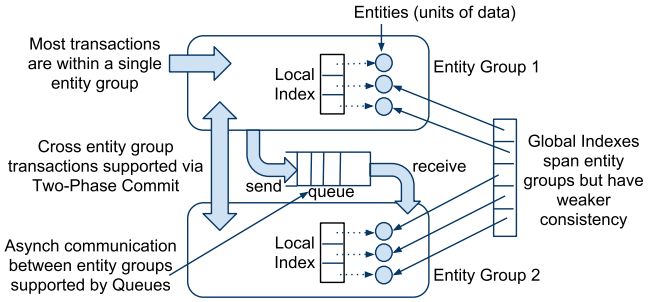



**Figure 2: Operations Across Entity Groups**

replicated via Paxos). Operations across entity groups could rely on expensive two-phase commits, but typically leverage Megastore's efficient asynchronous messaging. A transaction in a sending entity group places one or more messages in a queue; transactions in receiving entity groups atomically consume those messages and apply ensuing mutations.

Note that we use asynchronous messaging between logically distant entity groups, not physically distant replicas. All network traffic between datacenters is from replicated operations, which are synchronous and consistent.

Indexes local to an entity group obey ACID semantics; those across entity groups have looser consistency. See Figure 2 for the various operations on and between entity groups.

### 2.2.2 *Selecting Entity Group Boundaries*

The entity group defines the *a priori* grouping of data for fast operations. Boundaries that are too fine-grained force excessive cross-group operations, but placing too much unrelated data in a single group serializes unrelated writes, which degrades throughput.

The following examples show ways applications can work within these constraints:

**Email** Each email account forms a natural entity group. Operations within an account are transactional and consistent: a user who sends or labels a message is guaranteed to observe the change despite possible failover to another replica. External mail routers handle communication between accounts.

**Blogs** A blogging application would be modeled with multiple classes of entity groups. Each user has a profile, which is naturally its own entity group. However, blogs are collaborative and have no single permanent owner. We create a second class of entity groups to hold the posts and metadata for each blog. A third class keys off the unique name claimed by each blog. The application relies on asynchronous messaging when a single user operation affects both blogs and profiles. For a lower-traffic operation like creating a new blog and claiming its unique name, two-phase commit is more convenient and performs adequately.

**Maps** Geographic data has no natural granularity of any consistent or convenient size. A mapping application can create entity groups by dividing the globe into non-overlapping patches. For mutations that span patches, the application uses two-phase commit to make them atomic. Patches must be large enough that two-phase transactions are uncommon, but small enough that each patch requires only a small write throughput. Unlike the previous examples, the number of entity groups does not grow with increased usage, so enough patches must be created initially for sufficient aggregate throughput at later scale.

Nearly all applications built on Megastore have found natural ways to draw entity group boundaries.

### 2.2.3 *Physical Layout*

We use Google's Bigtable [15] for scalable fault-tolerant storage within a single datacenter, allowing us to support arbitrary read and write throughput by spreading operations across multiple rows.

We minimize latency and maximize throughput by letting applications control the placement of data: through the selection of Bigtable instances and specification of locality within an instance.

To minimize latency, applications try to keep data near users and replicas near each other. They assign each entity group to the region or continent from which it is accessed most. Within that region they assign a triplet or quintuplet of replicas to datacenters with isolated failure domains.

For low latency, cache efficiency, and throughput, the data for an entity group are held in contiguous ranges of Bigtable rows. Our schema language lets applications control the placement of hierarchical data, storing data that is accessed together in nearby rows or denormalized into the same row.

## 3. A TOUR OF MEGASTORE

Megastore maps this architecture onto a feature set carefully chosen to encourage rapid development of scalable applications. This section motivates the tradeoffs and describes the developer-facing features that result.

### 3.1 API Design Philosophy

ACID transactions simplify reasoning about correctness, but it is equally important to be able to reason about performance. Megastore emphasizes *cost-transparent* APIs with runtime costs that match application developers' intuitions.

Normalized relational schemas rely on joins at query time to service user operations. This is not the right model for Megastore applications for several reasons:

- High-volume interactive workloads benefit more from predictable performance than from an expressive query language.

- Reads dominate writes in our target applications, so it pays to move work from read time to write time.
- Storing and querying hierarchical data is straightforward in key-value stores like Bigtable.

With this in mind, we designed a data model and schema language to offer fine-grained control over physical locality. Hierarchical layouts and declarative denormalization help eliminate the need for most joins. Queries specify scans or lookups against particular tables and indexes.

Joins, when required, are implemented in application code. We provide an implementation of the merge phase of the merge join algorithm, in which the user provides multiple queries that return primary keys for the same table in the same order; we then return the intersection of keys for all the provided queries.

We also have applications that implement outer joins with parallel queries. This typically involves an index lookup followed by parallel index lookups using the results of the initial lookup. We have found that when the secondary index lookups are done in parallel and the number of results from the first lookup is reasonably small, this provides an effective stand-in for SQL-style joins.

While schema changes require corresponding modifications to the query implementation code, this system guarantees that features are built with a clear understanding of their performance implications. For example, when users (who may not have a background in databases) find themselves writing something that resembles a nested-loop join algorithm, they quickly realize that it's better to add an index and follow the index-based join approach above.

## 3.2 Data Model

Megastore defines a data model that lies between the abstract tuples of an RDBMS and the concrete row-column storage of NoSQL. As in an RDBMS, the data model is declared in a *schema* and is strongly typed. Each schema has a set of *tables*, each containing a set of *entities*, which in turn contain a set of *properties*. Properties are named and typed values. The types can be strings, various flavors of numbers, or Google's Protocol Buffers [9]. They can be required, optional, or repeated (allowing a list of values in a single property). All entities in a table have the same set of allowable properties. A sequence of properties is used to form the primary key of the entity, and the primary keys must be unique within the table. Figure 3 shows an example schema for a simple photo storage application.

Megastore tables are either *entity group root* tables or *child* tables. Each child table must declare a single distinguished foreign key referencing a root table, illustrated by the `ENTITY GROUP KEY` annotation in Figure 3. Thus each child entity references a particular entity in its root table (called the *root entity*). An entity group consists of a root entity along with all entities in child tables that reference it. A Megastore instance can have several root tables, resulting in different classes of entity groups.

In the example schema of Figure 3, each user's photo collection is a separate entity group. The root entity is the `User`, and the `Photo`s are child entities. Note the `Photo.tag` field is repeated, allowing multiple tags per `Photo` without the need for a sub-table.

### 3.2.1 Pre-Joining with Keys

While traditional relational modeling recommends that

```
CREATE SCHEMA PhotoApp;

CREATE TABLE User {
 required int64 user_id;
 required string name;
} PRIMARY KEY(user_id), ENTITY GROUP ROOT;

CREATE TABLE Photo {
 required int64 user_id;
 required int32 photo_id;
 required int64 time;
 required string full_url;
 optional string thumbnail_url;
 repeated string tag;
} PRIMARY KEY(user_id, photo_id),
  IN TABLE User,
  ENTITY GROUP KEY(user_id) REFERENCES User;

CREATE LOCAL INDEX PhotosByTime
  ON Photo(user_id, time);

CREATE GLOBAL INDEX PhotosByTag
  ON Photo(tag) STORING (thumbnail_url);
```

**Figure 3: Sample Schema for Photo Sharing Service**

all primary keys take surrogate values, Megastore keys are chosen to cluster entities that will be read together. Each entity is mapped into a single Bigtable row; the primary key values are concatenated to form the Bigtable row key, and each remaining property occupies its own Bigtable column.

Note how the `Photo` and `User` tables in Figure 3 share a common `user_id` key prefix. The `IN TABLE User` directive instructs Megastore to colocate these two tables into the same Bigtable, and the key ordering ensures that `Photo` entities are stored adjacent to the corresponding `User`. This mechanism can be applied recursively to speed queries along arbitrary join depths. Thus, users can force hierarchical layout by manipulating the key order.

Schemas declare keys to be sorted ascending or descending, or to avert sorting altogether: the `SCATTER` attribute instructs Megastore to prepend a two-byte hash to each key. Encoding monotonically increasing keys this way prevents hotspots in large data sets that span Bigtable servers.

### 3.2.2 Indexes

Secondary indexes can be declared on any list of entity properties, as well as fields within protocol buffers. We distinguish between two high-level classes of indexes: *local* and *global* (see Figure 2). A local index is treated as separate indexes for each entity group. It is used to find data within an entity group. In Figure 3, `PhotosByTime` is an example of a local index. The index entries are stored in the entity group and are updated atomically and consistently with the primary entity data.

A global index spans entity groups. It is used to find entities without knowing in advance the entity groups that contain them. The `PhotosByTag` index in Figure 3 is global and enables discovery of photos marked with a given tag, regardless of owner. Global index scans can read data owned by many entity groups but are not guaranteed to reflect all recent updates.

Megastore offers additional indexing features:

#### 3.2.2.1 Storing Clause.

Accessing entity data through indexes is normally a two-step process: first the index is read to find matching primary keys, then these keys are used to fetch entities. We provide a way to denormalize portions of entity data directly into index entries. By adding the `STORING` clause to an index, applications can store additional properties from the primary table for faster access at read time. For example, the `PhotosByTag` index stores the photo thumbnail URL for faster retrieval without the need for an additional lookup.

#### 3.2.2.2 Repeated Indexes.

Megastore provides the ability to index repeated properties and protocol buffer sub-fields. Repeated indexes are a efficient alternative to child tables. `PhotosByTag` is a repeated index: each unique entry in the `tag` property causes one index entry to be created on behalf of the `Photo`.

#### 3.2.2.3 Inline Indexes.

*Inline indexes* provide a way to denormalize data from source entities into a related target entity: index entries from the source entities appear as a virtual repeated column in the target entry. An inline index can be created on any table that has a foreign key referencing another table by using the first primary key of the target entity as the first components of the index, and physically locating the data in the same Bigtable as the target.

Inline indexes are useful for extracting slices of information from child entities and storing the data in the parent for fast access. Coupled with repeated indexes, they can also be used to implement many-to-many relationships more efficiently than by maintaining a many-to-many link table.

The `PhotosByTime` index could have been implemented as an inline index into the parent `User` table. This would make the data accessible as a normal index or as a virtual repeated property on `User`, with a time-ordered entry for each contained `Photo`.

### 3.2.3 Mapping to Bigtable

The Bigtable column name is a concatenation of the Megastore table name and the property name, allowing entities from different Megastore tables to be mapped into the same Bigtable row without collision. Figure 4 shows how data from the example photo application might look in Bigtable.

Within the Bigtable row for a root entity, we store the transaction and replication metadata for the entity group, including the transaction log. Storing all metadata in a single Bigtable row allows us to update it atomically through a single Bigtable transaction.

Each index entry is represented as a single Bigtable row; the row key of the cell is constructed using the indexed property values concatenated with the primary key of the indexed entity. For example, the `PhotosByTime` index row keys would be the tuple (*user_id*, *time*, *primary key*) for each photo. Indexing repeated fields produces one index entry per repeated element. For example, the primary key for a photo with three tags would appear in the `PhotosByTag` index thrice.

## 3.3 Transactions and Concurrency Control

Each Megastore entity group functions as a mini-database

| **Row key** | User. name | Photo. time | Photo. tag | Photo. _url |
|---|---|---|---|---|
| **101** | John | | | |
| **101,500** | | 12:30:01 | Dinner, Paris | ... |
| **101,502** | | 12:15:22 | Betty, Paris | ... |
| **102** | Mary | | | |

**Figure 4: Sample Data Layout in Bigtable**

that provides serializable ACID semantics. A transaction writes its mutations into the entity group's write-ahead log, then the mutations are applied to the data.

Bigtable provides the ability to store multiple values in the same row/column pair with different timestamps. We use this feature to implement multiversion concurrency control (MVCC): when mutations within a transaction are applied, the values are written at the timestamp of their transaction. Readers use the timestamp of the last fully applied transaction to avoid seeing partial updates. Readers and writers don't block each other, and reads are isolated from writes for the duration of a transaction.

Megastore provides *current*, *snapshot*, and *inconsistent* reads. Current and snapshot reads are always done within the scope of a single entity group. When starting a current read, the transaction system first ensures that all previously committed writes are applied; then the application reads at the timestamp of the latest committed transaction. For a snapshot read, the system picks up the timestamp of the last known fully applied transaction and reads from there, even if some committed transactions have not yet been applied. Megastore also provides inconsistent reads, which ignore the state of the log and read the latest values directly. This is useful for operations that have more aggressive latency requirements and can tolerate stale or partially applied data.

A write transaction always begins with a current read to determine the next available log position. The commit operation gathers mutations into a log entry, assigns it a timestamp higher than any previous one, and appends it to the log using Paxos. The protocol uses optimistic concurrency: though multiple writers might be attempting to write to the same log position, only one will win. The rest will notice the victorious write, abort, and retry their operations. Advisory locking is available to reduce the effects of contention. Batching writes through session affinity to a particular front-end server can avoid contention altogether.

The complete transaction lifecycle is as follows:

1. **Read**: Obtain the timestamp and log position of the last committed transaction.
2. **Application logic**: Read from Bigtable and gather writes into a log entry.
3. **Commit**: Use Paxos to achieve consensus for appending that entry to the log.
4. **Apply**: Write mutations to the entities and indexes in Bigtable.
5. **Clean up**: Delete data that is no longer required.

The write operation can return to the client at any point after Commit, though it makes a best-effort attempt to wait for the nearest replica to apply.

### 3.3.1 Queues

Queues provide transactional messaging between entity groups. They can be used for cross-group operations, to

batch multiple updates into a single transaction, or to defer work. A transaction on an entity group can atomically send or receive multiple messages in addition to updating its entities. Each message has a single sending and receiving entity group; if they differ, delivery is asynchronous. (See Figure 2.)

Queues offer a way to perform operations that affect many entity groups. For example, consider a calendar application in which each calendar has a distinct entity group, and we want to send an invitation to a group of calendars. A single transaction can atomically send invitation queue messages to many distinct calendars. Each calendar receiving the message will process the invitation in its own transaction which updates the invitee's state and deletes the message.

There is a long history of message queues in full-featured RDBMSs. Our support is notable for its scale: declaring a queue automatically creates an inbox on each entity group, giving us millions of endpoints.

#### 3.3.2 Two-Phase Commit

Megastore supports two-phase commit for atomic updates across entity groups. Since these transactions have much higher latency and increase the risk of contention, we generally discourage applications from using the feature in favor of queues. Nevertheless, they can be useful in simplifying application code for unique secondary key enforcement.

### 3.4 Other Features

We have built a tight integration with Bigtable's full-text index in which updates and searches participate in Megastore's transactions and multiversion concurrency. A full-text index declared in a Megastore schema can index a table's text or other application-generated attributes.

Synchronous replication is sufficient defense against the most common corruptions and accidents, but backups can be invaluable in cases of programmer or operator error. Megastore's integrated backup system supports periodic full snapshots as well as incremental backup of transaction logs. The restore process can bring back an entity group's state to any point in time, optionally omitting selected log entries (as after accidental deletes). The backup system complies with legal and common sense principles for expiring deleted data.

Applications have the option of encrypting data at rest, including the transaction logs. Encryption uses a distinct key per entity group. We avoid granting the same operators access to both the encryption keys and the encrypted data.

## 4. REPLICATION

This section details the heart of our synchronous replication scheme: a low-latency implementation of Paxos. We discuss operational details and present some measurements of our production service.

### 4.1 Overview

Megastore's replication system provides a single, consistent view of the data stored in its underlying replicas. Reads and writes can be initiated from any replica, and ACID semantics are preserved regardless of what replica a client starts from. Replication is done per entity group by synchronously replicating the group's transaction log to a quorum of replicas. Writes typically require one round of inter-datacenter communication, and healthy-case reads run locally. Current reads have the following guarantees:

- A read always observes the last-acknowledged write.
- After a write has been observed, all future reads observe that write. (A write might be observed before it is acknowledged.)

### 4.2 Brief Summary of Paxos

The Paxos algorithm is a way to reach consensus among a group of replicas on a single value. It tolerates delayed or reordered messages and replicas that fail by stopping. A majority of replicas must be active and reachable for the algorithm to make progress—that is, it allows up to $F$ faults with $2F + 1$ replicas. Once a value is *chosen* by a majority, all future attempts to read or write the value will reach the same outcome.

The ability to determine the outcome of a single value by itself is not of much use to a database. Databases typically use Paxos to replicate a transaction log, where a separate instance of Paxos is used for each position in the log. New values are written to the log at the position following the last chosen position.

The original Paxos algorithm [27] is ill-suited for high-latency network links because it demands multiple rounds of communication. Writes require at least two inter-replica roundtrips before consensus is achieved: a round of *prepares*, which reserves the right for a subsequent round of *accepts*. Reads require at least one round of prepares to determine the last chosen value. Real world systems built on Paxos reduce the number of roundtrips required to make it a practical algorithm. We will first review how master-based systems use Paxos, and then explain how we make Paxos efficient.

### 4.3 Master-Based Approaches

To minimize latency, many systems use a dedicated master to which all reads and writes are directed. The master participates in all writes, so its state is always up-to-date. It can serve reads of the current consensus state without any network communication. Writes are reduced to a single round of communication by piggybacking a prepare for the next write on each accept [14]. The master can batch writes together to improve throughput.

Reliance on a master limits flexibility for reading and writing. Transaction processing must be done near the master replica to avoid accumulating latency from sequential reads. Any potential master replica must have adequate resources for the system's full workload; slave replicas waste resources until the moment they become master. Master failover can require a complicated state machine, and a series of timers must elapse before service is restored. It is difficult to avoid user-visible outages.

### 4.4 Megastore's Approach

In this section we discuss the optimizations and innovations that make Paxos practical for our system.

#### 4.4.1 Fast Reads

We set an early requirement that current reads should usually execute on any replica without inter-replica RPCs. Since writes usually succeed on all replicas, it was realistic to allow local reads everywhere. These *local reads* give us better utilization, low latencies in all regions, fine-grained read failover, and a simpler programming experience.

We designed a service called the *Coordinator*, with servers in each replica's datacenter. A coordinator server tracks a set of entity groups for which its replica has observed all Paxos writes. For entity groups in that set, the replica has sufficient state to serve local reads.

It is the responsibility of the write algorithm to keep coordinator state conservative. If a write fails on a replica's Bigtable, it cannot be considered committed until the group's key has been evicted from that replica's coordinator.

Since coordinators are simple, they respond more reliably and quickly than Bigtable. Handling of rare failure cases or network partitions is described in Section 4.7.

### 4.4.2 Fast Writes

To achieve fast single-roundtrip writes, Megastore adapts the pre-preparing optimization used by master-based approaches. In a master-based system, each successful write includes an implied prepare message granting the master the right to issue accept messages for the next log position. If the write succeeds, the prepares are honored, and the next write skips directly to the accept phase. Megastore does not use dedicated masters, but instead uses *leaders*.

We run an independent instance of the Paxos algorithm for each log position. The leader for each log position is a distinguished replica chosen alongside the preceding log position's consensus value. The leader arbitrates which value may use proposal number zero. The first writer to submit a value to the leader wins the right to ask all replicas to accept that value as proposal number zero. All other writers must fall back on two-phase Paxos.

Since a writer must communicate with the leader before submitting the value to other replicas, we minimize writer-leader latency. We designed our policy for selecting the next write's leader around the observation that most applications submit writes from the same region repeatedly. This leads to a simple but effective heuristic: use the closest replica.

### 4.4.3 Replica Types

So far all replicas have been *full* replicas, meaning they contain all the entity and index data and are able to service current reads. We also support the notion of a *witness* replica. Witnesses vote in Paxos rounds and store the write-ahead log, but do not apply the log and do not store entity data or indexes, so they have lower storage costs. They are effectively tie breakers and are used when there are not enough full replicas to form a quorum. Because they do not have a coordinator, they do not force an additional roundtrip when they fail to acknowledge a write.

*Read-only* replicas are the inverse of witnesses: they are non-voting replicas that contain full snapshots of the data. Reads at these replicas reflect a consistent view of some point in the recent past. For reads that can tolerate this staleness, read-only replicas help disseminate data over a wide geographic area without impacting write latency.

## 4.5 Architecture

Figure 5 shows the key components of Megastore for an instance with two full replicas and one witness replica.

Megastore is deployed through a client library and auxiliary servers. Applications link to the client library, which implements Paxos and other algorithms: selecting a replica for read, catching up a lagging replica, and so on.

Each application server has a designated *local replica*. The

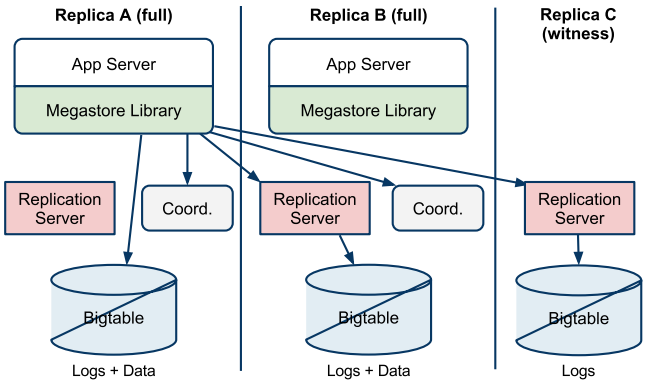



**Figure 5: Megastore Architecture Example**

client library makes Paxos operations on that replica durable by submitting transactions directly to the local Bigtable. To minimize wide-area roundtrips, the library submits remote Paxos operations to stateless intermediary *replication servers* communicating with their local Bigtables.

Client, network, or Bigtable failures may leave a write abandoned in an indeterminate state. Replication servers periodically scan for incomplete writes and propose no-op values via Paxos to bring them to completion.

## 4.6 Data Structures and Algorithms

This section details data structures and algorithms required to make the leap from consensus on a single value to a functioning replicated log.

### 4.6.1 Replicated Logs

Each replica stores mutations and metadata for the log entries known to the group. To ensure that a replica can participate in a write quorum even as it recovers from previous outages, we permit replicas to accept out-of-order proposals. We store log entries as independent cells in Bigtable.

We refer to a log replica as having "holes" when it contains an incomplete prefix of the log. Figure 6 demonstrates this scenario with some representative log replicas for a single Megastore entity group. Log positions 0-99 have been fully scavenged and position 100 is partially scavenged, because each replica has been informed that the other replicas will never request a copy. Log position 101 was accepted by all replicas. Log position 102 found a bare quorum in A and C. Position 103 is noteworthy for having been accepted by A and C, leaving B with a hole at 103. A conflicting write attempt has occurred at position 104 on replica A and B preventing consensus.

### 4.6.2 Reads

In preparation for a current read (as well as before a write), at least one replica must be brought up to date: all mutations previously committed to the log must be copied to and applied on that replica. We call this process *catchup*.

Omitting some deadline management, the algorithm for a current read (shown in Figure 7) is as follows:

1. **Query Local**: Query the local replica's coordinator to determine if the entity group is up-to-date locally.
2. **Find Position**: Determine the highest possibly-committed log position, and select a replica that has ap-

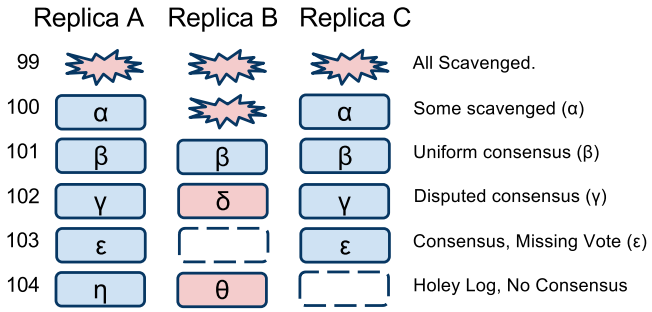



**Figure 6: Write Ahead Log**

plied through that log position.

(a) (*Local read*) If step 1 indicates that the local replica is up-to-date, read the highest accepted log position and timestamp from the local replica.

(b) (*Majority read*) If the local replica is not up-to-date (or if step 1 or step 2a times out), read from a majority of replicas to find the maximum log position that any replica has seen, and pick a replica to read from. We select the most responsive or up-to-date replica, not always the local replica.

3. **Catchup**: As soon as a replica is selected, catch it up to the maximum known log position as follows:

   (a) For each log position in which the selected replica does not know the consensus value, read the value from another replica. For any log positions without a known-committed value available, invoke Paxos to propose a no-op write. Paxos will drive a majority of replicas to converge on a single value—either the no-op or a previously proposed write.

   (b) Sequentially apply the consensus value of all unapplied log positions to advance the replica's state to the distributed consensus state.

   In the event of failure, retry on another replica.

4. **Validate**: If the local replica was selected and was not previously up-to-date, send the coordinator a *validate* message asserting that the (*entity group*, *replica*) pair reflects all committed writes. Do not wait for a reply—if the request fails, the next read will retry.

5. **Query Data**: Read the selected replica using the timestamp of the selected log position. If the selected replica becomes unavailable, pick an alternate replica, perform catchup, and read from it instead. The results of a single large query may be assembled transparently from multiple replicas.

Note that in practice 1 and 2a are executed in parallel.

### 4.6.3 Writes

Having completed the read algorithm, Megastore observes the next unused log position, the timestamp of the last write, and the next leader replica. At commit time all pending changes to the state are packaged and proposed, with a timestamp and next leader nominee, as the consensus value for the next log position. If this value wins the distributed

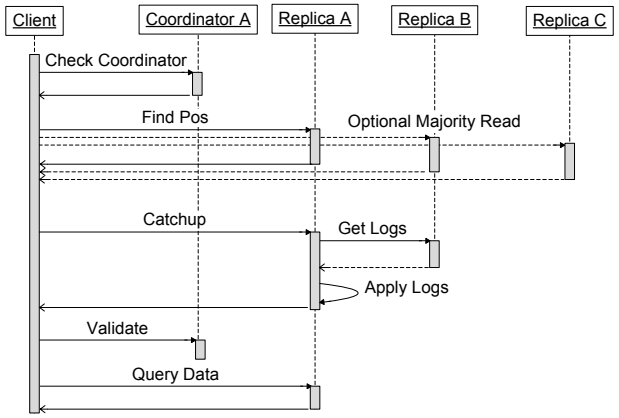



**Figure 7: Timeline for reads with local replica $A$**

consensus, it is applied to the state at all full replicas; otherwise the entire transaction is aborted and must be retried from the beginning of the read phase.

As described above, coordinators keep track of the entity groups that are up-to-date in their replica. If a write is not accepted on a replica, we must remove the entity group's key from that replica's coordinator. This process is called *invalidation*. Before a write is considered committed and ready to apply, all full replicas must have accepted or had their coordinator invalidated for that entity group.

The write algorithm (shown in Figure 8) is as follows:

1. **Accept Leader**: Ask the leader to accept the value as proposal number zero. If successful, skip to step 3.
2. **Prepare**: Run the Paxos Prepare phase at all replicas with a higher proposal number than any seen so far at this log position. Replace the value being written with the highest-numbered proposal discovered, if any.
3. **Accept**: Ask remaining replicas to accept the value. If this fails on a majority of replicas, return to step 2 after a randomized backoff.
4. **Invalidate**: Invalidate the coordinator at all full replicas that did not accept the value. Fault handling at this step is described in Section 4.7 below.
5. **Apply**: Apply the value's mutations at as many replicas as possible. If the chosen value differs from that originally proposed, return a conflict error.

Step 1 implements the "fast writes" of Section 4.4.2. Writers using single-phase Paxos skip Prepare messages by sending an Accept command at proposal number zero. The next leader replica selected at log position $n$ arbitrates the value used for proposal zero at $n + 1$. Since multiple proposers may submit values with proposal number zero, serializing at this replica ensures only one value corresponds with that proposal number for a particular log position.

In a traditional database system, the *commit point* (when the change is durable) is the same as the *visibility point* (when reads can see a change and when a writer can be notified of success). In our write algorithm, the commit point is after step 3 when the write has won the Paxos round, but the visibility point is after step 4. Only after all full replicas have accepted or had their coordinators invalidated can the write be acknowledged and the changes applied. Acknowledging before step 4 could violate our consistency guarantees: a current read at a replica whose invalidation was skipped might

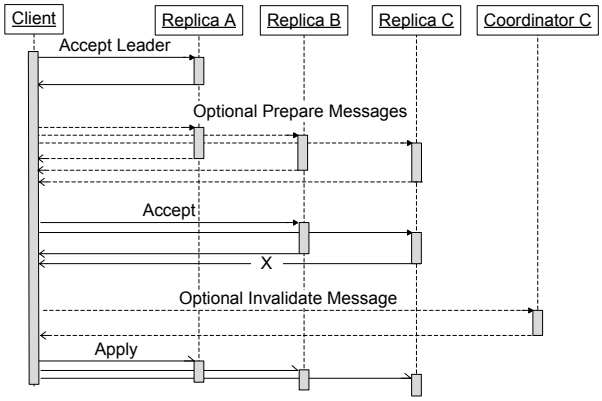



**Figure 8: Timeline for writes**

fail to observe the acknowledged write.

## 4.7 Coordinator Availability

Coordinator processes run in each datacenter and keep state only about their local replica. In the write algorithm above, each full replica must either accept or have its coordinator invalidated, so it might appear that any single replica failure (Bigtable and coordinator) will cause unavailability.

In practice this is not a common problem. The coordinator is a simple process with no external dependencies and no persistent storage, so it tends to be much more stable than a Bigtable server. Nevertheless, network and host failures can still make the coordinator unavailable.

### *4.7.1 Failure Detection*

To address network partitions, coordinators use an out-of-band protocol to identify when other coordinators are up, healthy, and generally reachable.

We use Google's Chubby lock service [13]: coordinators obtain specific Chubby locks in remote datacenters at startup. To process requests, a coordinator must hold a majority of its locks. If it ever loses a majority of its locks from a crash or network partition, it will revert its state to a conservative default, considering all entity groups in its purview to be out-of-date. Subsequent reads at the replica must query the log position from a majority of replicas until the locks are regained and its coordinator entries are revalidated.

Writers are insulated from coordinator failure by testing whether a coordinator has lost its locks: in that scenario, a writer knows that the coordinator will consider itself invalidated upon regaining them.

This algorithm risks a brief (tens of seconds) write outage when a datacenter containing live coordinators suddenly becomes unavailable—all writers must wait for the coordinator's Chubby locks to expire before writes can complete (much like waiting for a master failover to trigger). Unlike after a master failover, reads and writes can proceed smoothly while the coordinator's state is reconstructed. This brief and rare outage risk is more than justified by the steady state of fast local reads it allows.

The coordinator liveness protocol is vulnerable to asymmetric network partitions. If a coordinator can maintain the leases on its Chubby locks, but has otherwise lost contact with proposers, then affected entity groups will experience a write outage. In this scenario an operator performs a manual step to disable the partially isolated coordinator. We have faced this condition only a handful of times.

### *4.7.2 Validation Races*

In addition to availability issues, protocols for reading and writing to the coordinator must contend with a variety of race conditions. Invalidate messages are always safe, but validate messages must be handled with care. Races between validates for earlier writes and invalidates for later writes are protected in the coordinator by always sending the log position associated with the action. Higher numbered invalidates always trump lower numbered validates. There are also races associated with a crash between an invalidate by a writer at position $n$ and a validate at some position $m < n$. We detect crashes using a unique epoch number for each incarnation of the coordinator: validates are only allowed to modify the coordinator state if the epoch remains unchanged since the most recent read of the coordinator.

In summary, using coordinators to allow fast local reads from any datacenter is not free in terms of the impact to availability. But in practice most of the problems with running the coordinator are mitigated by the following factors:

- Coordinators are much simpler processes than Bigtable servers, have many fewer dependencies, and are thus naturally more available.
- Coordinators' simple, homogeneous workload makes them cheap and predictable to provision.
- Coordinators' light network traffic allows using a high network QoS with reliable connectivity.
- Operators can centrally disable coordinators for maintenance or unhealthy periods. This is automatic for certain monitoring signals.
- A quorum of Chubby locks detects most network partitions and node unavailability.

## 4.8 Write Throughput

Our implementation of Paxos has interesting tradeoffs in system behavior. Application servers in multiple datacenters may initiate writes to the same entity group and log position simultaneously. All but one of them will fail and need to retry their transactions. The increased latency imposed by synchronous replication increases the likelihood of conflicts for a given per-entity-group commit rate.

Limiting that rate to a few writes per second per entity group yields insignificant conflict rates. For apps whose entities are manipulated by a small number of users at a time, this limitation is generally not a concern. Most of our target customers scale write throughput by sharding entity groups more finely or by ensuring replicas are placed in the same region, decreasing both latency and conflict rate.

Applications with some server "stickiness" are well positioned to batch user operations into fewer Megastore transactions. Bulk processing of Megastore queue messages is a common batching technique, reducing the conflict rate and increasing aggregate throughput.

For groups that must regularly exceed a few writes per second, applications can use the fine-grained advisory locks dispensed by coordinator servers. Sequencing transactions back-to-back avoids the delays associated with retries and the reversion to two-phase Paxos when a conflict is detected.

## 4.9 Operational Issues

When a particular full replica becomes unreliable or loses connectivity, Megastore's performance can degrade. We have

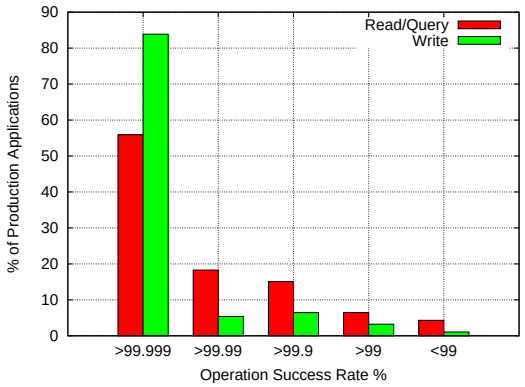



**Figure 9: Distribution of Availability**

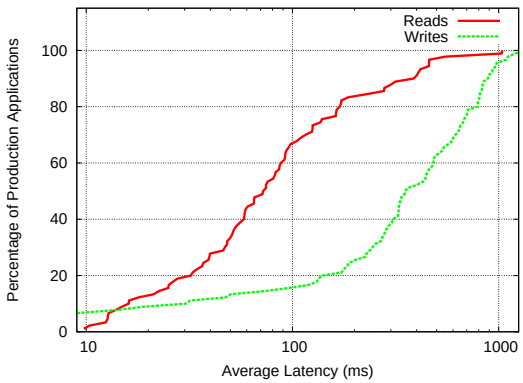



**Figure 10: Distribution of Average Latencies**

a number of ways to respond to these failures, including: routing users away from the problematic replica, disabling its coordinators, or disabling it entirely. In practice we rely on a combination of techniques, each with its own tradeoffs.

The first and most important response to an outage is to disable Megastore clients at the affected replica by rerouting traffic to application servers near other replicas. These clients typically experience the same outage impacting the storage stack below them, and might be unreachable from the outside world.

Rerouting traffic alone is insufficient if unhealthy coordinator servers might continue to hold their Chubby locks. The next response is to disable the replica's coordinators, ensuring that the problem has a minimal impact on write latency. (Section 4.7 described this process in more detail.) Once writers are absolved of invalidating the replica's coordinators, an unhealthy replica's impact on write latency is limited. Only the initial "accept leader" step in the write algorithm depends on the replica, and we maintain a tight deadline before falling back on two-phase Paxos and nominating a healthier leader for the next write.

A more draconian and rarely used action is to disable the replica entirely: neither clients nor replication servers will attempt to communicate with it. While sequestering the replica can seem appealing, the primary impact is a hit to availability: one less replica is eligible to help writers form a quorum. The valid use case is when attempted operations might cause harm—e.g. when the underlying Bigtable is severely overloaded.

## 4.10 Production Metrics

Megastore has been deployed within Google for several years; more than 100 production applications use it as their storage service. In this section, we report some measurements of its scale, availability, and performance.

Figure 9 shows the distribution of availability, measured on a per-application, per-operation basis. Most of our customers see extremely high levels of availability (at least five nines) despite a steady stream of machine failures, network hiccups, datacenter outages, and other faults. The bottom end of our sample includes some pre-production applications that are still being tested and batch processing applications with higher failure tolerances.

Average read latencies are tens of milliseconds, depending on the amount of data, showing that most reads are local. Most users see average write latencies of 100–400 milliseconds, depending on the distance between datacenters, the size of the data being written, and the number of full replicas. Figure 10 shows the distribution of average latency for read and commit operations.

# 5. EXPERIENCE

Development of the system was aided by a strong emphasis on testability. The code is instrumented with numerous (but cheap) assertions and logging, and has thorough unit test coverage. But the most effective bug-finding tool was our network simulator: the *pseudo-random test* framework. It is capable of exploring the space of all possible orderings and delays of communications between simulated nodes or threads, and deterministically reproducing the same behavior given the same seed. Bugs were exposed by finding a problematic sequence of events triggering an assertion failure (or incorrect result), often with enough log and trace information to diagnose the problem, which was then added to the suite of unit tests. While an exhaustive search of the scheduling state space is impossible, the pseudo-random simulation explores more than is practical by other means. Through running thousands of simulated hours of operation each night, the tests have found many surprising problems.

In real-world deployments we have observed the expected performance: our replication protocol optimizations indeed provide local reads most of the time, and writes with about the overhead of a single WAN roundtrip. Most applications have found the latency tolerable. Some applications are designed to hide write latency from users, and a few must choose entity group boundaries carefully to maximize their write throughput. This effort yields major operational advantages: Megastore's latency tail is significantly shorter than that of the underlying layers, and most applications can withstand planned and unplanned outages with little or no manual intervention.

Most applications use the Megastore schema language to model their data. Some have implemented their own *entity-attribute-value* model within the Megastore schema language, then used their own application logic to model their data (most notably, Google App Engine [8]). Some use a hybrid of the two approaches. Having the dynamic schema built on top of the static schema, rather than the other way around, allows most applications to enjoy the performance, usability,

and integrity benefits of the static schema, while still giving the option of a dynamic schema to those who need it.

The term "high availability" usually signifies the ability to mask faults to make a collection of systems more reliable than the individual systems. While fault tolerance is a highly desired goal, it comes with it its own pitfalls: it often hides persistent underlying problems. We have a saying in the group: "Fault tolerance is fault masking". Too often, the resilience of our system coupled with insufficient vigilance in tracking the underlying faults leads to unexpected problems: small transient errors on top of persistent uncorrected problems cause significantly larger problems.

Another issue is flow control. An algorithm that tolerates faulty participants can be heedless of slow ones. Ideally a collection of disparate machines would make progress only as fast as the least capable member. If slowness is interpreted as a fault, and tolerated, the fastest majority of machines will process requests at their own pace, reaching equilibrium only when slowed down by the load of the laggers struggling to catch up. We call this anomaly *chain gang throttling*, evoking the image of a group of escaping convicts making progress only as quickly as they can drag the stragglers.

A benefit of Megastore's write-ahead log has been the ease of integrating external systems. Any idempotent operation can be made a step in applying a log entry.

Achieving good performance for more complex queries requires attention to the physical data layout in Bigtable. When queries are slow, developers need to examine Bigtable traces to understand why their query performs below their expectations. Megastore does not enforce specific policies on block sizes, compression, table splitting, locality group, nor other tuning controls provided by Bigtable. Instead, we expose these controls, providing application developers with the ability (and burden) of optimizing performance.

# 6. RELATED WORK

Recently, there has been increasing interest in NoSQL data storage systems to meet the demand of large web applications. Representative work includes Bigtable [15], Cassandra [6], and Yahoo PNUTS [16]. In these systems, scalability is achieved by sacrificing one or more properties of traditional RDBMS systems, e.g., transactions, schema support, query capability [12, 33]. These systems often reduce the scope of transactions to the granularity of single key access and thus place a significant hurdle to building applications [18, 32]. Some systems extend the scope of transactions to multiple rows within a single table, for example the Amazon SimpleDB [5] uses the concept of *domain* as the transactional unit. Yet such efforts are still limited because transactions cannot cross tables or scale arbitrarily. Moreover, most current scalable data storage systems lack the rich data model of an RDBMS, which increases the burden on developers. Combining the merits from both database and scalable data stores, Megastore provides transactional ACID guarantees within a entity group and provides a flexible data model with user-defined schema, database-style and full-text indexes, and queues.

Data replication across geographically distributed datacenters is an indispensable means of improving availability in state-of-the-art storage systems. Most prevailing data storage systems use asynchronous replication schemes with a weaker consistency model. For example, Cassandra [6], HBase [1], CouchDB [7], and Dynamo [19] use an eventual consistency model and PNUTS uses "timeline" consistency [16]. By comparison, synchronous replication guarantees strong transactional semantics over wide-area networks and improves the performance of current reads.

Synchronous replication for traditional RDBMS systems presents a performance challenge and is difficult to scale [21]. Some proposed workarounds allow for strong consistency via asynchronous replication. One approach lets updates complete before their effects are replicated, passing the synchronization delay on to transactions that need to read the updated state [26]. Another approach routes writes to a single master while distributing read-only transactions among a set of replicas [29]. The updates are asynchronously propagated to the remaining replicas, and reads are either delayed or sent to replicas that have already been synchronized. A recent proposal for efficient synchronous replication introduces an ordering preprocessor that schedules incoming transactions deterministically, so that they can be independently applied at multiple replicas with identical results [31]. The synchronization burden is shifted to the preprocessor, which itself would have to be made scalable.

Until recently, few have used Paxos to achieve synchronous replication. SCALARIS is one example that uses the Paxos commit protocol [22] to implement replication for a distributed hash table [30]. Keyspace [2] also uses Paxos to implement replication on a generic key-value store. However the scalability and performance of these systems is not publicly known. Megastore is perhaps the first large-scale storage systems to implement Paxos-based replication across datacenters while satisfying the scalability and performance requirements of scalable web applications in the cloud.

Conventional database systems provide mature and sophisticated data management features, but have difficulties in serving large-scale interactive services targeted by this paper [33]. Open source database systems such as MySQL [10] do not scale up to the levels we require [17], while expensive commercial database systems like Oracle [4] significantly increase the total cost of ownership in large deployments in the cloud. Furthermore, neither of them offer fault-tolerant synchronous replication mechanism [3, 11], which is a key piece to build interactive services in the cloud.

# 7. CONCLUSION

In this paper we present Megastore, a scalable, highly available datastore designed to meet the storage requirements of interactive Internet services. We use Paxos for synchronous wide area replication, providing lightweight and fast failover of individual operations. The latency penalty of synchronous replication across widely distributed replicas is more than offset by the convenience of a single system image and the operational benefits of carrier-grade availability. We use Bigtable as our scalable datastore while adding richer primitives such as ACID transactions, indexes, and queues. Partitioning the database into entity group sub-databases provides familiar transactional features for most operations while allowing scalability of storage and throughput.

Megastore has over 100 applications in production, facing both internal and external users, and providing infrastructure for higher levels. The number and diversity of these applications is evidence of Megastore's ease of use, generality, and power. We hope that Megastore demonstrates the viability of a middle ground in feature set and replication consistency for today's scalable storage systems.

## 8. ACKNOWLEDGMENTS

Steve Newman, Jonas Karlsson, Philip Zeyliger, Alex Dingle, and Peter Stout all made substantial contributions to Megastore. We also thank Tushar Chandra, Mike Burrows, and the Bigtable team for technical advice, and Hector Gonzales, Jayant Madhavan, Ruth Wang, and Kavita Guliani for assistance with the paper. Special thanks to Adi Ofer for providing the spark to make this paper happen.

## 9. REFERENCES

[1] Apache HBase. http://hbase.apache.org/, 2008.

[2] Keyspace: A consistently replicated, highly-available key-value store. http://scalien.com/whitepapers/.

[3] MySQL Cluster. http://dev.mysql.com/tech-resources/articles/mysql_clustering_ch5.html, 2010.

[4] Oracle Database. http://www.oracle.com/us/-products/database/index.html, 2007.

[5] Amazon SimpleDB. http://aws.amazon.com/simpledb/, 2007.

[6] Apache Cassandra. http://incubator.apache.org/cassandra/, 2008.

[7] Apache CouchDB. http://couchdb.apache.org/, 2008.

[8] Google App Engine. http://code.google.com/appengine/, 2008.

[9] Google Protocol Buffers: Google's data interchange format. http://code.google.com/p/protobuf/, 2008.

[10] MySQL. http://www.mysql.com, 2009.

[11] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the performance of wide-area synchronous database replication. Technical Report CNDS-2002-4, Johns Hopkins University, 2002.

[12] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. Scads: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[13] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[14] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, New York, NY, USA, 2010. ACM.

[18] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174, New York, NY, USA, 2010. ACM.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[20] J. Furman, J. S. Karlsson, J.-M. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A scalable data system for user facing applications. In *ACM SIGMOD/PODS Conference*, 2008.

[21] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[22] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[23] S. Gustavsson and S. F. Andler. Self-stabilization and eventual consistency in replicated real-time databases. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 105–107, New York, NY, USA, 2002. ACM.

[24] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[25] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[26] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. Strongly consistent replication for a bargain. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 52 –63, 2010.

[27] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[28] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. Technical Report MSR-TR-2009-63, Microsoft Research, 2009.

[29] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[30] F. Schintke, A. Reinefeld, S. e. Haridi, and T. Schutt. Enhanced paxos commit for transactions on dhts. In *10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, pages 448–454, 2010.

[31] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

[32] S. Wu, D. Jiang, B. C. Ooi, and K. L. W. Towards elastic transactional cloud storage with range query support. In *Int'l Conference on Very Large Data Bases (VLDB)*, 2010.

[33] F. Yang, J. Shanmugasundaram, and R. Yerneni. A scalable data platform for a large number of small applications. In *CIDR*, 2009.

# F1: A Distributed SQL Database That Scales

Jeff Shute, Chad Whipkey, David Menestrina, Radek Vingralek, Eric Rollins, Stephan Ellner, Traian Stancescu, Bart Samwel, Mircea Oancea, John Cieslewicz, Himani Apte, Ben Handy, Kyle Littlefield, Ian Rae*

Google, Inc.
*University of Wisconsin-Madison

## ABSTRACT

F1 is a distributed relational database system built at Google to support the AdWords business. F1 is a hybrid database that combines high availability, the scalability of NoSQL systems like Bigtable, and the consistency and usability of traditional SQL databases. F1 is built on Spanner, which provides synchronous cross-datacenter replication and strong consistency. Synchronous replication implies higher commit latency, but we mitigate that latency by using a hierarchical schema model with structured data types and through smart application design. F1 also includes a fully functional distributed SQL query engine and automatic change tracking and publishing.

## 1. INTRODUCTION

F1[1] is a fault-tolerant globally-distributed OLTP and OLAP database built at Google as the new storage system for Google's AdWords system. It was designed to replace a sharded MySQL implementation that was not able to meet our growing scalability and reliability requirements.

The key goals of F1's design are:

1. **Scalability**: The system must be able to scale up, trivially and transparently, just by adding resources. Our sharded database based on MySQL was hard to scale up, and even more difficult to rebalance. Our users needed complex queries and joins, which meant they had to carefully shard their data, and resharding data without breaking applications was challenging.
2. **Availability**: The system must never go down for any reason – datacenter outages, planned maintenance, schema changes, etc. The system stores data for Google's core business. Any downtime has a significant revenue impact.
3. **Consistency**: The system must provide ACID transactions, and must always present applications with consistent and correct data.
   Designing applications to cope with concurrency anomalies in their data is very error-prone, time-consuming, and ultimately not worth the performance gains.
4. **Usability**: The system must provide full SQL query support and other functionality users expect from a SQL database. Features like indexes and ad hoc query are not just nice to have, but absolute requirements for our business.

Recent publications have suggested that these design goals are mutually exclusive [5, 11, 23]. A key contribution of this paper is to show how we achieved all of these goals in F1's design, and where we made trade-offs and sacrifices. The name F1 comes from genetics, where a *Filial 1 hybrid* is the first generation offspring resulting from a cross mating of distinctly different parental types. The F1 database system is indeed such a hybrid, combining the best aspects of traditional relational databases and scalable NoSQL systems like Bigtable [6].

F1 is built on top of Spanner [7], which provides extremely scalable data storage, synchronous replication, and strong consistency and ordering properties. F1 inherits those features from Spanner and adds several more:

- Distributed SQL queries, including joining data from external data sources
- Transactionally consistent secondary indexes
- Asynchronous schema changes including database reorganizations
- Optimistic transactions
- Automatic change history recording and publishing

Our design choices in F1 result in higher latency for typical reads and writes. We have developed techniques to hide that increased latency, and we found that user-facing transactions can be made to perform as well as in our previous MySQL system:

- An F1 schema makes data clustering explicit, using tables with hierarchical relationships and columns with structured data types. This clustering improves data locality and reduces the number and cost of RPCs required to read remote data.

[1]Previously described briefly in [22].


*Proceedings of the VLDB Endowment, Vol. 6, No. 11*

- F1 users make heavy use of batching, parallelism and asynchronous reads. We use a new ORM (object-relational mapping) library that makes these concepts explicit. This places an upper bound on the number of RPCs required for typical application-level operations, making those operations scale well by default.

The F1 system has been managing all AdWords advertising campaign data in production since early 2012. AdWords is a vast and diverse ecosystem including 100s of applications and 1000s of users, all sharing the same database. This database is over 100 TB, serves up to hundreds of thousands of requests per second, and runs SQL queries that scan tens of trillions of data rows per day. Availability reaches five nines, even in the presence of unplanned outages, and observable latency on our web applications has not increased compared to the old MySQL system.

We discuss the AdWords F1 database throughout this paper as it was the original and motivating user for F1. Several other groups at Google are now beginning to deploy F1.

# 2. BASIC ARCHITECTURE

Users interact with F1 through the F1 *client* library. Other tools like the command-line ad-hoc SQL shell are implemented using the same client. The client sends requests to one of many F1 *servers*, which are responsible for reading and writing data from remote data sources and coordinating query execution. Figure 1 depicts the basic architecture and the communication between components.

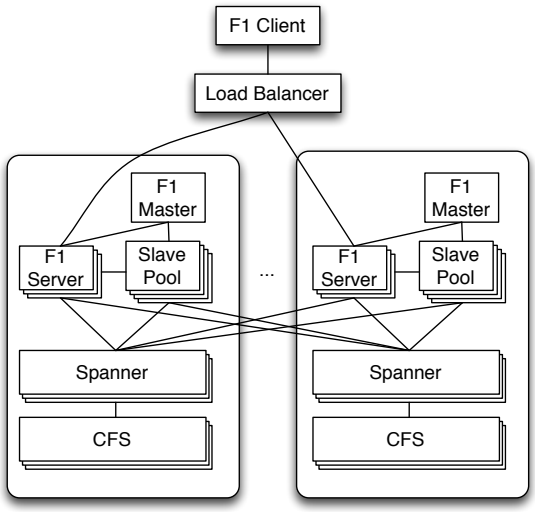



**Figure 1: The basic architecture of the F1 system, with servers in two datacenters.**

Because of F1's distributed architecture, special care must be taken to avoid unnecessarily increasing request latency. For example, the F1 client and load balancer prefer to connect to an F1 server in a nearby datacenter whenever possible. However, requests may transparently go to F1 servers in remote datacenters in cases of high load or failures.

F1 servers are typically co-located in the same set of datacenters as the Spanner servers storing the data. This co-location ensures that F1 servers generally have fast access to the underlying data. For availability and load balancing, F1 servers can communicate with Spanner servers outside their own datacenter when necessary. The Spanner servers in each datacenter in turn retrieve their data from the Colossus File System (CFS) [14] in the same datacenter. Unlike Spanner, CFS is not a globally replicated service and therefore Spanner servers will never communicate with remote CFS instances.

F1 servers are mostly stateless, allowing a client to communicate with a different F1 server for each request. The one exception is when a client uses pessimistic transactions and must hold locks. The client is then bound to one F1 server for the duration of that transaction. F1 transactions are described in more detail in Section 5. F1 servers can be quickly added (or removed) from our system in response to the total load because F1 servers do not own any data and hence a server addition (or removal) requires no data movement.

An F1 cluster has several additional components that allow for the execution of distributed SQL queries. Distributed execution is chosen over centralized execution when the query planner estimates that increased parallelism will reduce query processing latency. The shared *slave pool* consists of F1 processes that exist only to execute parts of distributed query plans on behalf of regular F1 servers. Slave pool membership is maintained by the F1 *master*, which monitors slave process health and distributes the list of available slaves to F1 servers. F1 also supports large-scale data processing through Google's MapReduce framework [10]. For performance reasons, MapReduce workers are allowed to communicate directly with Spanner servers to extract data in bulk (not shown in the figure). Other clients perform reads and writes exclusively through F1 servers.

The throughput of the entire system can be scaled up by adding more Spanner servers, F1 servers, or F1 slaves. Since F1 servers do not store data, adding new servers does not involve any data re-distribution costs. Adding new Spanner servers results in data re-distribution. This process is completely transparent to F1 servers (and therefore F1 clients).

The Spanner-based remote storage model and our geographically distributed deployment leads to latency characteristics that are very different from those of regular databases. Because the data is synchronously replicated across multiple datacenters, and because we've chosen widely distributed datacenters, the commit latencies are relatively high (50-150 ms). This high latency necessitates changes to the patterns that clients use when interacting with the database. We describe these changes in Section 7.1, and we provide further detail on our deployment choices, and the resulting availability and latency, in Sections 9 and 10.

## 2.1 Spanner

F1 is built on top of Spanner. Both systems were developed at the same time and in close collaboration. Spanner handles lower-level storage issues like persistence, caching, replication, fault tolerance, data sharding and movement, location lookups, and transactions.

In Spanner, data rows are partitioned into clusters called *directories* using ancestry relationships in the schema. Each directory has at least one *fragment*, and large directories can have multiple fragments. *Groups* store a collection of directory fragments. Each group typically has one replica *tablet* per datacenter. Data is replicated synchronously using the *Paxos* algorithm [18], and all tablets for a group store

the same data. One replica tablet is elected as the Paxos *leader* for the group, and that leader is the entry point for all transactional activity for the group. Groups may also include *readonly replicas*, which do not vote in the Paxos algorithm and cannot become the group leader.

Spanner provides serializable pessimistic transactions using strict two-phase locking. A transaction includes multiple reads, taking shared or exclusive locks, followed by a single write that upgrades locks and atomically commits the transaction. All commits are synchronously replicated using Paxos. Transactions are most efficient when updating data co-located in a single group. Spanner also supports transactions across multiple groups, called transaction *participants*, using a two-phase commit (2PC) protocol on top of Paxos. 2PC adds an extra network round trip so it usually doubles observed commit latency. 2PC scales well up to 10s of participants, but abort frequency and latency increase significantly with 100s of participants [7].

Spanner has very strong consistency and timestamp semantics. Every transaction is assigned a commit timestamp, and these timestamps provide a global total ordering for commits. Spanner uses a novel mechanism to pick globally ordered timestamps in a scalable way using hardware clocks deployed in Google datacenters. Spanner uses these timestamps to provide multi-versioned consistent reads, including snapshot reads of current data, without taking read locks. For guaranteed non-blocking, globally consistent reads, Spanner provides a *global safe timestamp*, below which no in-flight or future transaction can possibly commit. The global safe timestamp typically lags current time by 5-10 seconds. Reads at this timestamp can normally run on any replica tablet, including readonly replicas, and they never block behind running transactions.

# 3. DATA MODEL

## 3.1 Hierarchical Schema

The F1 data model is very similar to the Spanner data model. In fact, Spanner's original data model was more like Bigtable, but Spanner later adopted F1's data model. At the logical level, F1 has a relational schema similar to that of a traditional RDBMS, with some extensions including explicit table hierarchy and columns with Protocol Buffer data types.

Logically, tables in the F1 schema can be organized into a *hierarchy*. Physically, F1 stores each child table *clustered* with and *interleaved* within the rows from its parent table. Tables from the logical schema cannot be arbitrarily interleaved: the child table must have a foreign key to its parent table as a prefix of its primary key. For example, the AdWords schema contains a table Customer with primary key `(CustomerId)`, which has a child table Campaign with primary key `(CustomerId, CampaignId)`, which in turn has a child table AdGroup with primary key `(CustomerId, CampaignId, AdGroupId)`. A row of the root table in the hierarchy is called a *root row*. All child table rows corresponding to a root row are clustered together with that root row in a single Spanner *directory*, meaning that cluster is normally stored on a single Spanner server. Child rows are stored under their parent row ordered by primary key. Figure 2 shows an example.

The hierarchically clustered physical schema has several advantages over a flat relational schema. Consider the corresponding traditional schema, also depicted in Figure 2. In this traditional schema, fetching all Campaign and AdGroup records corresponding to a given `CustomerId` would take two sequential steps, because there is no direct way to retrieve AdGroup records by `CustomerId`. In the F1 version of the schema, the hierarchical primary keys allow the fetches of Campaign and AdGroup records to be started in parallel, because both tables are keyed by `CustomerId`. The primary key prefix property means that reading all AdGroups for a particular Customer can be expressed as a single range read, rather than reading each row individually using an index. Furthermore, because the tables are both stored in primary key order, rows from the two tables can be joined using a simple ordered merge. Because the data is clustered into a single directory, we can read it all in a single Spanner request. All of these properties of a hierarchical schema help mitigate the latency effects of having remote data.

Hierarchical clustering is especially useful for updates, since it reduces the number of Spanner groups involved in a transaction. Because each root row and all of its descendant rows are stored in a single Spanner directory, transactions restricted to a single root will usually avoid 2PC and the associated latency penalty, so most applications try to use single-root transactions as much as possible. Even when doing transactions across multiple roots, it is important to limit the number of roots involved because adding more participants generally increases latency and decreases the likelihood of a successful commit.

Hierarchical clustering is not mandatory in F1. An F1 schema often has several root tables, and in fact, a completely flat MySQL-style schema is still possible. Using hierarchy however, to the extent that it matches data semantics, is highly beneficial. In AdWords, most transactions are typically updating data for a single advertiser at a time, so we made the advertiser a root table (Customer) and clustered related tables under it. This clustering was critical to achieving acceptable latency.

## 3.2 Protocol Buffers

The F1 data model supports table columns that contain structured data types. These structured types use the schema and binary encoding format provided by Google's open source Protocol Buffer [16] library. Protocol Buffers have typed fields that can be required, optional, or repeated; fields can also be nested Protocol Buffers. At Google, Protocol Buffers are ubiquitous for data storage and interchange between applications. When we still had a MySQL schema, users often had to write tedious and error-prone transformations between database rows and in-memory data structures. Putting protocol buffers in the schema removes this impedance mismatch and gives users a universal data structure they can use both in the database and in application code.

Protocol Buffers allow the use of *repeated fields*. In F1 schema designs, we often use repeated fields instead of child tables when the number of child records has a low upper bound. By using repeated fields, we avoid the performance overhead and complexity of storing and joining multiple child records. The entire protocol buffer is effectively treated as one blob by Spanner. Aside from performance impacts, Protocol Buffer columns are more natural and reduce semantic complexity for users, who can now read and write their logical business objects as atomic units, without having to

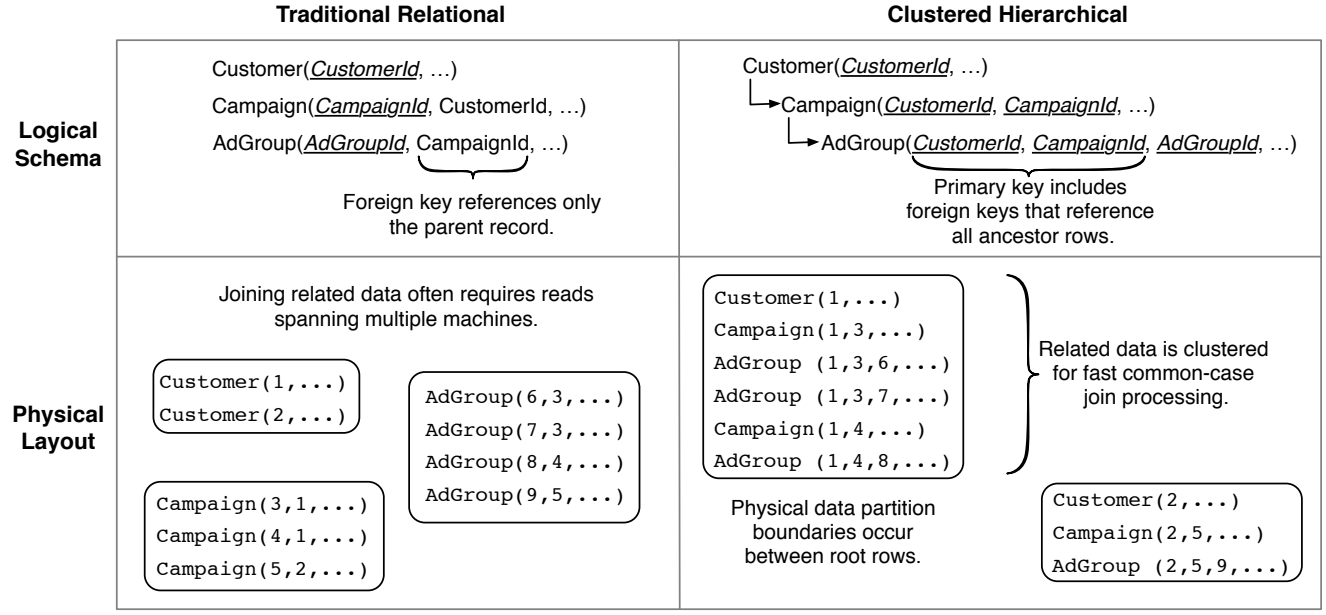



**Figure 2: The logical and physical properties of data storage in a traditional normalized relational schema compared with a clustered hierarchical schema used in an F1 database.**

think about materializing them using joins across several tables. The use of Protocol Buffers in F1 SQL is described in Section 8.7.

Many tables in an F1 schema consist of just a single Protocol Buffer column. Other tables split their data across a handful of columns, partitioning the fields according to access patterns. Tables can be partitioned into columns to group together fields that are usually accessed together, to separate fields with static and frequently updated data, to allow specifying different read/write permissions per column, or to allow concurrent updates to different columns. Using fewer columns generally improves performance in Spanner where there can be high per-column overhead.

## 3.3 Indexing

All indexes in F1 are transactional and fully consistent. Indexes are stored as separate tables in Spanner, keyed by a concatenation of the index key and the indexed table's primary key. Index keys can be either scalar columns or fields extracted from Protocol Buffers (including repeated fields). There are two types of physical storage layout for F1 indexes: local and global.

*Local index* keys must contain the root row primary key as a prefix. For example, an index on `(CustomerId, Keyword)` used to store unique keywords for each customer is a local index. Like child tables, local indexes are stored in the same Spanner directory as the root row. Consequently, the index entries of local indexes are stored on the same Spanner server as the rows they index, and local index updates add little additional cost to any transaction.

In contrast, *global index* keys do not include the root row primary key as a prefix and hence cannot be co-located with the rows they index. For example, an index on `(Keyword)` that maps from all keywords in the database to Customers that use them must be global. Global indexes are often large and can have high aggregate update rates. Consequently, they are sharded across many directories and stored on multiple Spanner servers. Writing a single row that updates a global index requires adding a single extra participant to a transaction, which means the transaction must use 2PC, but that is a reasonable cost to pay for consistent global indexes.

Global indexes work reasonably well for single-row updates, but can cause scaling problems for large transactions. Consider a transaction that inserts 1000 rows. Each row requires adding one or more global index entries, and those index entries could be arbitrarily spread across 100s of index directories, meaning the 2PC transaction would have 100s of participants, making it slower and more error-prone. Therefore, we use global indexes sparingly in the schema, and encourage application writers to use small transactions when bulk inserting into tables with global indexes.

Megastore [3] makes global indexes scalable by giving up consistency and supporting only asynchronous global indexes. We are currently exploring other mechanisms to make global indexes more scalable without compromising consistency.

# 4. SCHEMA CHANGES

The AdWords database is shared by thousands of users and is under constant development. Batches of schema changes are queued by developers and applied daily. This database is mission critical for Google and requires very high availability. Downtime or table locking during schema changes (e.g. adding indexes) is not acceptable.

We have designed F1 to make all schema changes fully non-blocking. Several aspects of the F1 system make non-blocking schema changes particularly challenging:

- F1 is a massively distributed system, with servers in multiple datacenters in distinct geographic regions.

- Each F1 server has a schema locally in memory. It is not practical to make an update occur atomically across all servers.
- Queries and transactions must continue on all tables, even those undergoing schema changes.
- System availability and latency must not be negatively impacted during schema changes.

Because F1 is massively distributed, even if F1 had a global F1 server membership repository, synchronous schema change across all servers would be very disruptive to response times. To make changes atomic, at some point, servers would have to block transactions until confirming all other servers have received the change. To avoid this, F1 schema changes are applied *asynchronously*, on different F1 servers at different times. This implies that two F1 servers may update the database concurrently using different schemas.

If two F1 servers update the database using different schemas that are not compatible according to our schema change algorithms, this could lead to anomalies including database corruption. We illustrate the possibility of database corruption using an example. Consider a schema change from schema $S_1$ to schema $S_2$ that adds index $I$ on table $T$. Because the schema change is applied asynchronously on different F1 servers, assume that server $M_1$ is using schema $S_1$ and server $M_2$ is using schema $S_2$. First, server $M_2$ inserts a new row $r$, which also adds a new index entry $I(r)$ for row $r$. Subsequently, row $r$ is deleted by server $M_1$. Because the server is using schema $S_1$ and is not aware of index $I$, the server deletes row $r$, but fails to delete the index entry $I(r)$. Hence, the database becomes corrupt. For example, an index scan on $I$ would return spurious data corresponding to the deleted row $r$.

We have implemented a schema change algorithm that prevents anomalies similar to the above by

1. Enforcing that across all F1 servers, at most two different schemas are active. Each server uses either the current or next schema. We grant leases on the schema and ensure that no server uses a schema after lease expiry.
2. Subdividing each schema change into multiple phases where consecutive pairs of phases are mutually compatible and cannot cause anomalies. In the above example, we first add index $I$ in a mode where it only executes delete operations. This prohibits server $M_1$ from adding $I(r)$ into the database. Subsequently, we upgrade index $I$ so servers perform all write operations. Then we initiate a MapReduce to backfill index entries for all rows in table $T$ with carefully constructed transactions to handle concurrent writes. Once complete, we make index $I$ visible for normal read operations.

The full details of the schema change algorithms are covered in [20].

# 5. TRANSACTIONS

The AdWords product ecosystem requires a data store that supports ACID transactions. We store financial data and have hard requirements on data integrity and consistency. We also have a lot of experience with eventual consistency systems at Google. In all such systems, we find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date. We think this is an unacceptable burden to place on developers and that consistency problems should be solved at the database level. Full transactional consistency is one of the most important properties of F1.

Each F1 transaction consists of multiple reads, optionally followed by a single write that commits the transaction. F1 implements three types of transactions, all built on top of Spanner's transaction support:

1. **Snapshot transactions**. These are read-only transactions with snapshot semantics, reading repeatable data as of a fixed Spanner *snapshot timestamp*. By default, snapshot transactions read at Spanner's global safe timestamp, typically 5-10 seconds old, and read from a local Spanner replica. Users can also request a specific timestamp explicitly, or have Spanner pick the current timestamp to see current data. The latter option may have higher latency and require remote RPCs.

   Snapshot transactions are the default mode for SQL queries and for MapReduces. Snapshot transactions allow multiple client servers to see consistent views of the entire database at the same timestamp.

2. **Pessimistic transactions**. These transactions map directly on to Spanner transactions [7]. Pessimistic transactions use a stateful communications protocol that requires holding locks, so all requests in a single pessimistic transaction get directed to the same F1 server. If the F1 server restarts, the pessimistic transaction aborts. Reads in pessimistic transactions can request either shared or exclusive locks.

3. **Optimistic transactions**. Optimistic transactions consist of a read phase, which can take arbitrarily long and never takes Spanner locks, and then a short write phase. To detect row-level conflicts, F1 returns with each row its last modification timestamp, which is stored in a hidden *lock column* in that row. The new commit timestamp is automatically written into the lock column whenever the corresponding data is updated (in either pessimistic or optimistic transactions). The client library collects these timestamps, and passes them back to an F1 server with the write that commits the transaction. The F1 server creates a short-lived Spanner pessimistic transaction and re-reads the last modification timestamps for all read rows. If any of the re-read timestamps differ from what was passed in by the client, there was a conflicting update, and F1 aborts the transaction. Otherwise, F1 sends the writes on to Spanner to finish the commit.

F1 clients use optimistic transactions by default. Optimistic transactions have several benefits:

- Tolerating misbehaved clients. Reads never hold locks and never conflict with writes. This avoids any problems caused by badly behaved clients who run long transactions or abandon transactions without aborting them.

- Long-lasting transactions. Optimistic transactions can be arbitrarily long, which is useful in some cases. For example, some F1 transactions involve waiting for end-user interaction. It is also hard to debug a transaction that always gets aborted while single-stepping. Idle transactions normally get killed within ten seconds to avoid leaking locks, which means long-running pessimistic transactions often cannot commit.

- Server-side retriability. Optimistic transaction commits are self-contained, which makes them easy to retry transparently in the F1 server, hiding most transient Spanner errors from the user. Pessimistic transactions cannot be retried by F1 servers because they require re-running the user's business logic to reproduce the same locking side-effects.

- Server failover. All state associated with an optimistic transaction is kept on the client. Consequently, the client can send reads and commits to different F1 servers after failures or to balance load.

- Speculative writes. A client may read values outside an optimistic transaction (possibly in a MapReduce), and remember the timestamp used for that read. Then the client can use those values and timestamps in an optimistic transaction to do speculative writes that only succeed if no other writes happened after the original read.

Optimistic transactions do have some drawbacks:

- Insertion phantoms. Modification timestamps only exist for rows present in the table, so optimistic transactions do not prevent insertion phantoms [13]. Where this is a problem, it is possible to use parent-table locks to avoid phantoms. (See Section 5.1)

- Low throughput under high contention. For example, in a table that maintains a counter which many clients increment concurrently, optimistic transactions lead to many failed commits because the read timestamps are usually stale by write time. In such cases, pessimistic transactions with exclusive locks avoid the failed transactions, but also limit throughput. If each commit takes 50ms, at most 20 transactions per second are possible. Improving throughput beyond that point requires application-level changes, like batching updates.

F1 users can mix optimistic and pessimistic transactions arbitrarily and still preserve ACID semantics. All F1 writes update the last modification timestamp on every relevant lock column. Snapshot transactions are independent of any write transactions, and are also always consistent.

## 5.1 Flexible Locking Granularity

F1 provides row-level locking by default. Each F1 row contains one *default lock column* that covers all columns in the same row. However, concurrency levels can be changed in the schema. For example, users can increase concurrency by defining additional lock columns in the same row, with each lock column covering a subset of columns. In an extreme case, each column can be covered by a separate lock column, resulting in column-level locking.

One common use for column-level locking is in tables with concurrent writers, where each updates a different set of columns. For example, we could have a front-end system allowing users to change bids for keywords, and a back-end system that updates serving history on the same keywords. Busy customers may have continuous streams of bid updates at the same time that back-end systems are updating stats. Column-level locking avoids transaction conflicts between these independent streams of updates.

Users can also selectively reduce concurrency by using a lock column in a parent table to cover columns in a child table. This means that a set of rows in the child table share the same lock column and writes within this set of rows get serialized. Frequently, F1 users use lock columns in parent tables to avoid insertion phantoms for specific predicates or make other business logic constraints easier to enforce. For example, there could be a limit on keyword count per AdGroup, and a rule that keywords must be distinct. Such constraints are easy to enforce correctly if concurrent keyword insertions (in the same AdGroup) are impossible.

# 6. CHANGE HISTORY

Many database users build mechanisms to log changes, either from application code or using database features like triggers. In the MySQL system that AdWords used before F1, our Java application libraries added change history records into all transactions. This was nice, but it was inefficient and never 100% reliable. Some classes of changes would not get history records, including changes written from Python scripts and manual SQL data changes.

In F1, Change History is a first-class feature at the database level, where we can implement it most efficiently and can guarantee full coverage. In a change-tracked database, all tables are change-tracked by default, although specific tables or columns can be opted out in the schema. Every transaction in F1 creates one or more *ChangeBatch* Protocol Buffers, which include the primary key and before and after values of changed columns for each updated row. These ChangeBatches are written into normal F1 tables that exist as children of each root table. The primary key of the ChangeBatch table includes the associated root table key and the transaction commit timestamp. When a transaction updates data under multiple root rows, possibly from different root table hierarchies, one ChangeBatch is written for each distinct root row (and these ChangeBatches include pointers to each other so the full transaction can be reassembled if necessary). This means that for each root row, the change history table includes ChangeBatches showing all changes associated with children of that root row, in commit order, and this data is easily queryable with SQL. This clustering also means that change history is stored close to the data being tracked, so these additional writes normally do not add additional participants into Spanner transactions, and therefore have minimal latency impact.

F1's ChangeHistory mechanism has a variety of uses. The most common use is in applications that want to be notified of changes and then do some incremental processing. For example, the approval system needs to be notified when new ads have been inserted so it can approve them. F1 uses a publish-and-subscribe system to push notifications that particular root rows have changed. The publish happens in Spanner and is guaranteed to happen at least once after any series of changes to any root row. Subscribers normally remember a checkpoint (i.e. a high-water mark) for each root

row and read all changes newer than the checkpoint whenever they receive a notification. This is a good example of a place where Spanner's timestamp ordering properties are very powerful since they allow using checkpoints to guarantee that every change is processed exactly once. A separate system exists that makes it easy for these clients to see only changes to tables or columns they care about.

Change History also gets used in interesting ways for caching. One client uses an in-memory cache based on database state, distributed across multiple servers, and uses this while rendering pages in the AdWords web UI. After a user commits an update, it is important that the next page rendered reflects that update. When this client reads from the cache, it passes in the root row key and the commit timestamp of the last write that must be visible. If the cache is behind that timestamp, it reads Change History records beyond its checkpoint and applies those changes to its in-memory state to catch up. This is much cheaper than reloading the cache with a full extraction and much simpler and more accurate than comparable cache invalidation protocols.

# 7. CLIENT DESIGN

## 7.1 Simplified ORM

The nature of working with a distributed data store required us to rethink the way our client applications interacted with the database. Many of our client applications had been written using a MySQL-based ORM layer that could not be adapted to work well with F1. Code written using this library exhibited several common ORM anti-patterns:

- Obscuring database operations from developers.
- Serial reads, including `for` loops that do one query per iteration.
- Implicit traversals: adding unwanted joins and loading unnecessary data "just in case".

Patterns like these are common in ORM libraries. They may save development time in small-scale systems with local data stores, but they hurt scalability even then. When combined with a high-latency remote database like F1, they are disastrous. For F1, we replaced this ORM layer with a new, stripped-down API that forcibly avoids these anti-patterns. The new ORM layer does not use any joins and does not implicitly traverse any relationships between records. All object loading is explicit, and the ORM layer exposes APIs that promote the use of parallel and asynchronous read access. This is practical in an F1 schema for two reasons. First, there are simply fewer tables, and clients are usually loading Protocol Buffers directly from the database. Second, hierarchically structured primary keys make loading all children of an object expressible as a single range read without a join.

With this new F1 ORM layer, application code is more explicit and can be slightly more complex than code using the MySQL ORM, but this complexity is partially offset by the reduced impedance mismatch provided by Protocol Buffer columns. The transition usually results in better client code that uses more efficient access patterns to the database. Avoiding serial reads and other anti-patterns results in code that scales better with larger data sets and exhibits a flatter overall latency distribution.

With MySQL, latency in our main interactive application was highly variable. Average latency was typically 200-300 ms. Small operations on small customers would run much faster than that, but large operations on large customers could be much slower, with a latency tail of requests taking multiple seconds. Developers regularly fought to identify and fix cases in their code causing excessively serial reads and high latency. With our new coding style on the F1 ORM, this doesn't happen. User requests typically require a fixed number (fewer than 10) of reads, independent of request size or data size. The minimum latency is higher than in MySQL because of higher minimum read cost, but the average is about the same, and the latency tail for huge requests is only a few times slower than the median.

## 7.2 NoSQL Interface

F1 supports a NoSQL key/value based interface that allows for fast and simple programmatic access to rows. Read requests can include any set of tables, requesting specific columns and key ranges for each. Write requests specify inserts, updates, and deletes by primary key, with any new column values, for any set of tables.

This interface is used by the ORM layer under the hood, and is also available for clients to use directly. This API allows for batched retrieval of rows from multiple tables in a single call, minimizing the number of round trips required to complete a database transaction. Many applications prefer to use this NoSQL interface because it's simpler to construct structured read and write requests in code than it is to generate SQL. This interface can be also be used in MapReduces to specify which data to read.

## 7.3 SQL Interface

F1 also provides a full-fledged SQL interface, which is used for low-latency OLTP queries, large OLAP queries, and everything in between. F1 supports joining data from its Spanner data store with other data sources including Bigtable, CSV files, and the aggregated analytical data warehouse for AdWords. The SQL dialect extends standard SQL with constructs that allow accessing data stored in Protocol Buffers. Updates are also supported using SQL data manipulation statements, with extensions to support updating fields inside protocol buffers and to deal with repeated structure inside protocol buffers. Full syntax details are beyond the scope of this paper.

# 8. QUERY PROCESSING

The F1 SQL query processing system has the following key properties which we will elaborate on in this section:

- Queries are executed either as low-latency centrally executed queries or distributed queries with high parallelism.
- All data is remote and batching is used heavily to mitigate network latency.
- All input data and internal data is arbitrarily partitioned and has few useful ordering properties.
- Queries use many hash-based repartitioning steps.
- Individual query plan operators are designed to stream data to later operators as soon as possible, maximizing pipelining in query plans.

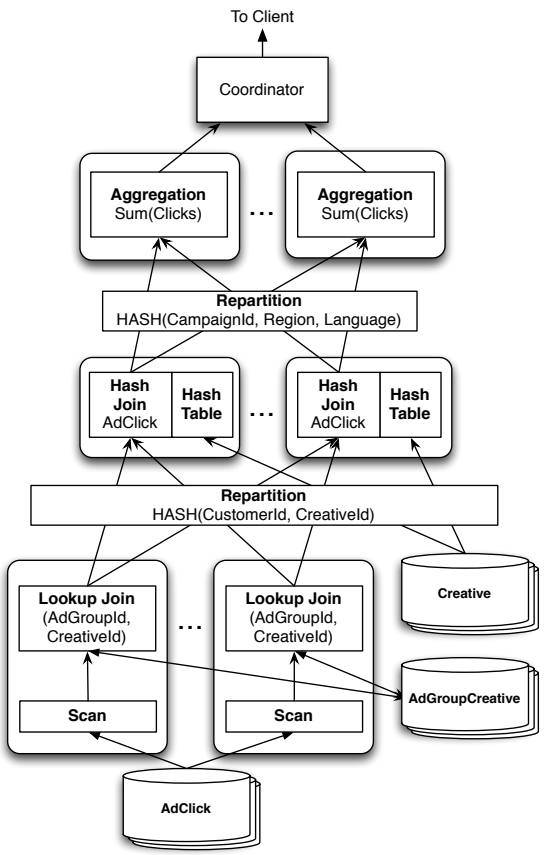



**Figure 3: A distributed query plan. Rounded boxes represent processes running on separate machines. Arrows show data flow within a process or over the network in the form of RPCs.**

- Hierarchically clustered tables have optimized access methods.
- Query data can be consumed in parallel.
- Protocol Buffer-valued columns provide first-class support for structured data types.
- Spanner's snapshot consistency model provides globally consistent results.

## 8.1 Central and Distributed Queries

F1 SQL supports both centralized and distributed execution of queries. Centralized execution is used for short OLTP-style queries and the entire query runs on one F1 server node. Distributed execution is used for OLAP-style queries and spreads the query workload over worker tasks in the F1 slave pool (see Section 2). Distributed queries always use snapshot transactions. The query optimizer uses heuristics to determine which execution mode is appropriate for a given query. In the sections that follow, we will mainly focus our attention on distributed query execution. Many of the concepts apply equally to centrally executed queries.

## 8.2 Distributed Query Example

The following example query would be answered using distributed execution:

```
SELECT agcr.CampaignId, click.Region,
       cr.Language, SUM(click.Clicks)
FROM AdClick click
  JOIN AdGroupCreative agcr
    USING (AdGroupId, CreativeId)
  JOIN Creative cr
    USING (CustomerId, CreativeId)
WHERE click.Date = '2013-03-23'
GROUP BY agcr.CampaignId, click.Region,
         cr.Language
```

This query uses part of the AdWords schema. An *AdGroup* is a collection of ads with some shared configuration. A *Creative* is the actual ad text. The *AdGroupCreative* table is a link table between AdGroup and Creative; Creatives can be shared by multiple AdGroups. Each *AdClick* records the Creative that the user was shown and the AdGroup from which the Creative was chosen. This query takes all AdClicks on a specific date, finds the corresponding AdGroupCreative and then the Creative. It then aggregates to find the number of clicks grouped by campaign, region and language.

A possible query plan for this query is shown in Figure 3. In the query plan, data is streamed bottom-up through each of the operators up until the aggregation operator. The deepest operator performs a scan of the AdClick table. In the same worker node, the data from the AdClick scan flows into a *lookup join* operator, which looks up AdGroupCreative records using a secondary index key. The plan then *repartitions* the data stream by a hash of the CustomerId and CreativeId, and performs a lookup in a hash table that is partitioned in the same way (a distributed hash join). After the distributed hash join, the data is once again repartitioned, this time by a hash of the CampaignId, Region and Language fields, and then fed into an aggregation operator that groups by those same fields (a distributed aggregation).

## 8.3 Remote Data

SQL query processing, and join processing in particular, poses some interesting challenges in F1, primarily because F1 does not store its data locally. F1's main data store is Spanner, which is a remote data source, and F1 SQL can also access other remote data sources and join across them. These remote data accesses involve highly variable network latency [9]. In contrast, traditional database systems generally perform processing on the same machine that hosts their data, and they mostly optimize to reduce the number of disk seeks and disk accesses.

Network latency and disk latency are fundamentally different in two ways. First, network latency can be mitigated by *batching* or *pipelining* data accesses. F1 uses extensive batching to mitigate network latency. Secondly, disk latency is generally caused by contention for a single limited resource, the actual disk hardware. This severely limits the usefulness of sending out multiple data accesses at the same time. In contrast, F1's network based storage is typically distributed over many disks, because Spanner partitions its data across many physical servers, and also at a finer-grained level because Spanner stores its data in CFS. This makes it much less likely that multiple data accesses will contend for the same resources, so scheduling multiple data accesses in parallel often results in near-linear speedup until the underlying storage system is truly overloaded.

The prime example of how F1 SQL takes advantage of batching is found in the *lookup join* query plan operator. This operator executes a join by reading from the inner table using equality lookup keys. It first retrieves rows from the outer table, extracting the lookup key values from them and deduplicating those keys. This continues until it has gathered 50MB worth of data or 100,000 unique lookup key values. Then it performs a simultaneous lookup of all keys in the inner table. This returns the requested data in arbitrary order. The lookup join operator joins the retrieved inner table rows to the outer table rows which are stored in memory, using a hash table for fast lookup. The results are streamed immediately as output from the lookup join node.

F1's query operators are designed to stream data as much as possible, reducing the incidence of pipeline stalls. This design decision limits operators' ability to preserve interesting data orders. Specifically, an F1 operator often has many reads running asynchronously in parallel, and streams rows to the next operator as soon as they are available. This emphasis on data streaming means that ordering properties of the input data are lost while allowing for maximum read request concurrency and limiting the space needed for row buffering.

## 8.4 Distributed Execution Overview

The structure of a distributed query plan is as follows. A full query plan consists of potentially tens of *plan parts*, each of which represents a number of workers that execute the same query subplan. The plan parts are organized as a directed acyclic graph (DAG), with data flowing up from the leaves of the DAG to a single *root node*, which is the only node with no out edges, i.e. the only sink. The root node, also called the *query coordinator*, is executed by the server that received the incoming SQL query request from a client. The query coordinator plans the query for execution, receives results from the penultimate plan parts, performs any final aggregation, sorting, or filtering, and then streams the results back to the client, except in the case of partitioned consumers as described in Section 8.6.

A technique frequently used by distributed database systems is to take advantage of an explicit co-partitioning of the stored data. Such co-partitioning can be used to push down large amounts of query processing onto each of the processing nodes that host the partitions. F1 cannot take advantage of such partitioning, in part because the data is always remote, but more importantly, because Spanner applies an arbitrary, effectively random partitioning. Moreover, Spanner can also dynamically change the partitioning. Hence, to perform operations efficiently, F1 must frequently resort to repartitioning the data. Because none of the input data is range partitioned, and because range partitioning depends on correct statistics, we have eschewed range partitioning altogether and opted to only apply hash partitioning.

Traditionally, repartitioning like this has been regarded as something to be avoided because of the heavy network traffic involved. Recent advances in the scalability of network switch hardware have allowed us to connect clusters of several hundred F1 worker processes in such a way that all servers can simultaneously communicate with each other at close to full network interface speed. This allows us to repartition without worrying much about network capacity and concepts like rack affinity. A potential downside of this solution is that it limits the size of an F1 cluster to the limits of available network switch hardware. This has not posed a problem in practice for the queries and data sizes that the F1 system deals with.

The use of hash partitioning allows us to implement an efficient distributed hash join operator and a distributed aggregation operator. These operators were already demonstrated in the example query in Section 8.2. The hash join operator repartitions both of its inputs by applying a hash function to the join keys. In the example query, the hash join keys are CustomerId and CreativeId. Each worker is responsible for a single partition of the hash join. Each worker loads its smallest input (as estimated by the query planner) into an in-memory hash table. It then reads its largest input and probes the hash table for each row, streaming out the results. For distributed aggregation, we aggregate as much as possible locally inside small buffers, then repartition the data by a hash of the grouping keys, and finally perform a full aggregation on each of the hash partitions. When hash tables grow too large to fit in memory, we apply standard algorithms that spill parts of the hash table to disk.

F1 SQL operators execute in memory, without checkpointing to disk, and stream data as much as possible. This avoids the cost of saving intermediate results to disk, so queries run as fast as the data can be processed. This means, however, that any server failure can make an entire query fail. Queries that fail get retried transparently, which usually hides these failures. In practice, queries that run for up to an hour are sufficiently reliable, but queries much longer than that may experience too many failures. We are exploring adding checkpointing for some intermediate results into our query plans, but this is challenging to do without hurting latency in the normal case where no failures occur.

## 8.5 Hierarchical Table Joins

As described in Section 3.1, the F1 data model supports hierarchically clustered tables, where the rows of a child table are interleaved in the parent table. This data model allows us to efficiently join a parent table and a descendant table by their shared primary key prefix. For instance, consider the join of table `Customer` with table `Campaign`:

```
SELECT *
FROM Customer JOIN
     Campaign USING (CustomerId)
```

The hierarchically clustered data model allows F1 to perform this join using a *single* request to Spanner in which we request the data from both tables. Spanner will return the data to F1 in interleaved order (a pre-order depth-first traversal), ordered by primary key prefix, e.g.:

```
Customer(3)
  Campaign(3,5)
  Campaign(3,6)
Customer(4)
  Campaign(4,2)
  Campaign(4,4)
```

While reading this stream, F1 uses a merge-join-like algorithm which we call *cluster join*. The cluster join operator only needs to buffer one row from each table, and returns the joined results in a streaming fashion as the Spanner input data is received. Any number of tables can be cluster joined this way using a single Spanner request, as long as all tables fall on a single ancestry path in the table hierarchy.

For instance, in the following table hierarchy, F1 SQL can only join `RootTable` to either `ChildTable1` or `ChildTable2` in this way, but not both:

```
RootTable
  ChildTable1
  ChildTable2
```

When F1 SQL has to join between sibling tables like these, it will perform one join operation using the cluster join algorithm, and select an alternate join algorithm for the remaining join. An algorithm such as lookup join is able to perform this join without disk spilling or unbounded memory usage because it can construct the join result piecemeal using bounded-size batches of lookup keys.

## 8.6 Partitioned Consumers

F1 queries can produce vast amounts of data, and pushing this data through a single query coordinator can be a bottleneck. Furthermore, a single client process receiving all the data can also be a bottleneck and likely cannot keep up with many F1 servers producing result rows in parallel. To solve this, F1 allows multiple client processes to consume sharded streams of data from the same query in parallel. This feature is used for partitioned consumers like MapReduces[10]. The client application sends the query to F1 and requests distributed data retrieval. F1 then returns a set of endpoints to connect to. The client must connect to all of these endpoints and retrieve the data in parallel. Due to the streaming nature of F1 queries, and the cross-dependencies caused by frequent hash repartitioning, slowness in one distributed reader may slow other distributed readers as well, as the F1 query produces results for all readers in lock-step. A possible, but so far unimplemented mitigation strategy for this horizontal dependency is to use disk-backed buffering to break the dependency and to allow clients to proceed independently.

## 8.7 Queries with Protocol Buffers

As explained in Section 3, the F1 data model makes heavy use of Protocol Buffer valued columns. The F1 SQL dialect treats these values as first class objects, providing full access to all of the data contained therein. For example, the following query requests the `CustomerId` and the entire Protocol Buffer valued column `Info` for each customer whose country code is `US`.

```
SELECT c.CustomerId, c.Info
FROM Customer AS c
WHERE c.Info.country_code = 'US'
```

This query illustrates two aspects of Protocol Buffer support. First, queries use path expressions to extract individual fields (`c.Info.country_code`). Second, F1 SQL also allows for querying and passing around entire protocol buffers (`c.Info`). Support for full Protocol Buffers reduces the impedance mismatch between F1 SQL and client applications, which often prefer to receive complete Protocol Buffers.

Protocol Buffers also allow *repeated fields*, which may have zero or more instances, i.e., they can be regarded as variable-length arrays. When these repeated fields occur in F1 database columns, they are actually very similar to hierarchical child tables in a 1:N relationship. The main difference between a child table and a repeated field is that the child table contains an explicit foreign key to its parent table, while the repeated field has an *implicit* foreign key to the Protocol Buffer containing it. Capitalizing on this similarity, F1 SQL supports access to repeated fields using *PROTO JOIN*, a JOIN variant that joins by the implicit foreign key. For instance, suppose that we have a table `Customer`, which has a Protocol Buffer column `Whitelist` which in turn contains a repeated field `feature`. Furthermore, suppose that the values of this field `feature` are themselves Protocol Buffers, each of which represents the whitelisting status of a particular feature for the parent `Customer`.

```
SELECT c.CustomerId, f.feature
FROM Customer AS c
     PROTO JOIN c.Whitelist.feature AS f
WHERE f.status = 'STATUS_ENABLED'
```

This query joins the `Customer` table with its virtual child table `Whitelist.feature` by the foreign key that is implied by containment. It then filters the resulting combinations by the value of a field `f.status` inside the child table `f`, and returns another field `f.feature` from that child table. In this query syntax, the `PROTO JOIN` specifies the parent relation of the repeated field by qualifying the repeated field name with `c`, which is the alias of the parent relation. The implementation of the `PROTO JOIN` construct is straightforward: in the read from the outer relation we retrieve the entire Protocol Buffer column containing the repeated field, and then for each outer row we simply enumerate the repeated field instances in memory and join them to the outer row.

F1 SQL also allows subqueries on repeated fields in Protocol Buffers. The following query has a scalar subquery to count the number of `Whitelist.feature`s, and an `EXISTS` subquery to select only `Customer`s that have at least one feature that is not `ENABLED`. Each subquery iterates over repeated field values contained inside Protocol Buffers from the current row.

```
SELECT c.CustomerId, c.Info,
  (SELECT COUNT(*) FROM c.Whitelist.feature) nf
FROM Customer AS c
WHERE EXISTS (SELECT * FROM c.Whitelist.feature f
              WHERE f.status != 'ENABLED')
```

Protocol Buffers have performance implications for query processing. First, we always have to fetch entire Protocol Buffer columns from Spanner, even when we are only interested in a small subset of fields. This takes both additional network and disk bandwidth. Second, in order to extract the fields that the query refers to, we always have to parse the contents of the Protocol Buffer fields. Even though we have implemented an optimized parser to extract only requested fields, the impact of this decoding step is significant. Future versions will improve this by pushing parsing and field selection to Spanner, thus reducing network bandwidth required and saving CPU in F1 while possibly using more CPU in Spanner.

# 9. DEPLOYMENT

The F1 and Spanner clusters currently deployed for AdWords use five datacenters spread out across mainland US. The Spanner configuration uses 5-way Paxos replication to

ensure high availability. Each region has additional read-only replicas that do not participate in the Paxos algorithm. Read-only replicas are used only for snapshot reads and thus allow us to segregate OLTP and OLAP workloads.

Intuitively, 3-way replication should suffice for high availability. In practice, this is not enough. When one datacenter is down (because of either an outage or planned maintenance), both surviving replicas must remain available for F1 to be able to commit transactions, because a Paxos commit must succeed on a majority of replicas. If a second datacenter goes down, the entire database becomes completely unavailable. Even a single machine failure or restart temporarily removes a second replica, causing unavailability for data hosted on that server.

Spanner's Paxos implementation designates one of the replicas as a *leader*. All transactional reads and commits must be routed to the leader replica. User transactions normally require at least two round trips to the leader (reads followed by a commit). Moreover, F1 servers usually perform an extra read as a part of transaction commit (to get old values for Change History, index updates, optimistic transaction timestamp verification, and referential integrity checks). Consequently, transaction latency is best when clients and F1 servers are co-located with Spanner leader replicas. We designate one of the datacenters as a *preferred leader location*. Spanner locates leader replicas in the preferred leader location whenever possible. Clients that perform heavy database modifications are usually deployed close to the preferred leader location. Other clients, including those that primarily run queries, can be deployed anywhere, and normally do reads against local F1 servers.

We have chosen to deploy our five read/write replicas with two each on the east and west coasts of the US, and the fifth centrally. With leaders on the east coast, commits require round trips to the other east coast datacenter, plus the central datacenter, which accounts for the 50ms minimum latency. We have chosen this deployment to maximize availability in the presence of large regional outages. Other F1 and Spanner instances could be deployed with closer replicas to reduce commit latency.

# 10. LATENCY AND THROUGHPUT

In our configuration, F1 users see read latencies of 5-10 ms and commit latencies of 50-150 ms. Commit latency is largely determined by network latency between datacenters. The Paxos algorithm allows a transaction to commit once a majority of voting Paxos replicas acknowledge the transaction. With five replicas, commits require a round trip from the leader to the two nearest replicas. Multi-group commits require 2PC, which typically doubles the minimum latency.

Despite the higher database latency, overall user-facing latency for the main interactive AdWords web application averages about 200ms, which is similar to the preceding system running on MySQL. Our schema clustering and application coding strategies have successfully hidden the inherent latency of synchronous commits. Avoiding serial reads in client code accounts for much of that. In fact, while the average is similar, the MySQL application exhibited tail latency much worse than the same application on F1.

For non-interactive applications that apply bulk updates, we optimize for throughput rather than latency. We typically structure such applications so they do small transactions, scoped to single Spanner directories when possible, and use parallelism to achieve high throughput. For example, we have one application that updates billions of rows per day, and we designed it to perform single-directory transactions of up to 500 rows each, running in parallel and aiming for 500 transactions per second. F1 and Spanner provide very high throughput for parallel writes like this and are usually not a bottleneck – our rate limits are usually chosen to protect downstream Change History consumers who can't process changes fast enough.

For query processing, we have mostly focused on functionality and parity so far, and not on absolute query performance. Small central queries reliably run in less than 10ms, and some applications do tens or hundreds of thousands of SQL queries per second. Large distributed queries run with latency comparable to MySQL. Most of the largest queries actually run faster in F1 because they can use more parallelism than MySQL, which can only parallelize up to the number of MySQL shards. In F1, such queries often see linear speedup when given more resources.

Resource costs are usually higher in F1, where queries often use an order of magnitude more CPU than similar MySQL queries. MySQL stored data uncompressed on local disk, and was usually bottlenecked by disk rather than CPU, even with flash disks. F1 queries start from data compressed on disk and go through several layers, decompressing, processing, recompressing, and sending over the network, all of which have significant cost. Improving CPU efficiency here is an area for future work.

# 11. RELATED WORK

As a hybrid of relational and NoSQL systems, F1 is related to work in both areas. F1's relational query execution techniques are similar to those described in the shared-nothing database literature, e.g., [12], with some key differences like the ignoring of interesting orders and the absence of co-partitioned data. F1's NoSQL capabilities share properties with other well-described scalable key-value stores including Bigtable [6], HBase [1], and Dynamo [11]. The hierarchical schema and clustering properties are similar to Megastore [3].

Optimistic transactions have been used in previous systems including Percolator [19] and Megastore [3]. There is extensive literature on transaction, consistency and locking models, including optimistic and pessimistic transactions, such as [24] and [4].

Prior work [15] also chose to mitigate the inherent latency of remote lookups though the use of asynchrony in query processing. However, due to the large volumes of data processed by F1, our system is not able to make the simplifying assumption that an unlimited number of asynchronous requests can be made at the same time. This complication, coupled with the high variability of storage operation latency, led to the out-of-order streaming design described in Section 8.3.

MDCC [17] suggests some Paxos optimizations that could be applied to reduce the overhead of multi-participant transactions.

Using Protocol Buffers as first class types makes F1, in part, a kind of object database [2]. The resulting simplified ORM results in a lower impedance mismatch for most client applications at Google, where Protocol Buffers are used pervasively. The hierarchical schema and clustering properties are similar to Megastore and ElasTraS [8]. F1

treats repeated fields inside protocol buffers like nested relations [21].

## 12. CONCLUSION

In recent years, conventional wisdom in the engineering community has been that if you need a highly scalable, high-throughput data store, the only viable option is to use a NoSQL key/value store, and to work around the lack of ACID transactional guarantees and the lack of conveniences like secondary indexes, SQL, and so on. When we sought a replacement for Google's MySQL data store for the AdWords product, that option was simply not feasible: the complexity of dealing with a non-ACID data store in every part of our business logic would be too great, and there was simply no way our business could function without SQL queries. Instead of going NoSQL, we built F1, a distributed relational database system that combines high availability, the throughput and scalability of NoSQL systems, and the functionality, usability and consistency of traditional relational databases, including ACID transactions and SQL queries.

Google's core AdWords business is now running completely on F1. F1 provides the SQL database functionality that our developers are used to and our business requires. Unlike our MySQL solution, F1 is trivial to scale up by simply adding machines. Our low-level commit latency is higher, but by using a coarse schema design with rich column types and improving our client application coding style, the observable end-user latency is as good as before and the worst-case latencies have actually improved.

F1 shows that it is actually possible to have a highly scalable and highly available distributed database that still provides all of the guarantees and conveniences of a traditional relational database.

## 13. ACKNOWLEDGEMENTS

We would like to thank the Spanner team, without whose great efforts we could not have built F1. We'd also like to thank the many developers and users across all AdWords teams who migrated their systems to F1, and who played a large role influencing and validating the design of this system. We also thank all former and newer F1 team members, including Michael Armbrust who helped write this paper, and Marcel Kornacker who worked on the early design of the F1 query engine.

## 14. REFERENCES

[1] Apache Foundation. Apache HBase. `http://hbase.apache.org/`.

[2] M. Atkinson et al. The object-oriented database system manifesto. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an object-oriented database system*, pages 1–20. Morgan Kaufmann, 1992.

[3] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[4] H. Berenson et al. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[5] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC*, 2000.

[6] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[7] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[8] S. Das et al. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *TODS*, 38(1):5:1–5:45, Apr. 2013.

[9] J. Dean. Evolution and future directions of large-scale storage and computation systems at Google. In *SOCC*, 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[12] D. J. Dewitt et al. The Gamma database machine project. *TKDE*, 2(1):44–62, Mar. 1990.

[13] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *CACM*, 19(11):624–633, Nov. 1976.

[14] A. Fikes. Storage architecture and challenges. Google Faculty Summit, July 2010.

[15] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *SIGMOD*, 2000.

[16] Google, Inc. Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[17] T. Kraska et al. MDCC: Multi-data center consistency. In *EuroSys*, 2013.

[18] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[19] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[20] I. Rae et al. Online, asynchronous schema change in F1. *PVLDB*, 6(11), 2013.

[21] M. A. Roth et al. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst.*, 13(4):389–417, Oct. 1988.

[22] J. Shute et al. F1: The fault-tolerant distributed RDBMS supporting Google's ad business. In *SIGMOD*, 2012.

[23] M. Stonebraker. SQL databases v. NoSQL databases. *CACM*, 53(4), 2010.

[24] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.

# A Relational Model of Data for Large Shared Data Banks

E. F. CODD
*IBM Research Laboratory, San Jose, California*

**Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.**

**Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.**

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity
CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

## 1. Relational Model and Normal Form

### 1.1. Introduction

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Levein and Maron [2] provide numerous references to work in this area.

In contrast, the problems treated here are those of *data independence*—the independence of application programs and terminal activities from growth in data types and changes in data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive systems.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for noninferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

### 1.2. Data Dependencies in Present Systems

The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed *without logically impairing some application programs* is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items). Three of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

1.2.1. *Ordering Dependence.* Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of addresses. For example, the records of a file concerning parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or is a subordering of) the

stored ordering. Those application programs which take advantage of the stored ordering of a file are likely to fail to operate correctly if for some reason it becomes necessary to replace that ordering by a different one. Similar remarks hold for a stored ordering implemented by means of pointers.

It is unnecessary to single out any system as an example, because all the well-known information systems that are marketed today fail to make a clear distinction between order of presentation on the one hand and stored ordering on the other. Significant implementation problems must be solved to provide this kind of independence.

1.2.2. *Indexing Dependence.* In the context of formatted data, an index is usually thought of as a purely performance-oriented component of the data representation. It tends to improve response to queries and updates and, at the same time, slow down response to insertions and deletions. From an informational standpoint, an index is a redundant component of the data representation. If a system uses indices at all and if it is to perform well in an environment with changing patterns of activity on the data bank, an ability to create and destroy indices from time to time will probably be necessary. The question then arises: Can application programs and terminal activities remain invariant as indices come and go?

Present formatted data systems take widely different approaches to indexing. TDMS [7] unconditionally provides indexing on all attributes. The presently released version of IMS [5] provides the user with a choice for each file: a choice between no indexing at all (the hierarchic sequential organization) or indexing on the primary key only (the hierarchic indexed sequential organization). In neither case is the user's application logic dependent on the existence of the unconditionally provided indices. IDS [8], however, permits the file designers to select attributes to be indexed and to incorporate indices into the file structure by means of additional chains. Application programs taking advantage of the performance benefit of these indexing chains must refer to those chains by name. Such programs do not operate correctly if these chains are later removed.

1.2.3. *Access Path Dependence.* Many of the existing formatted data systems provide users with tree-structured files or slightly more general network models of the data. Application programs developed to work with these systems tend to be logically impaired if the trees or networks are changed in structure. A simple example follows.

Suppose the data bank contains information about parts and projects. For each part, the part number, part name, part description, quantity-on-hand, and quantity-on-order are recorded. For each project, the project number, project name, project description are recorded. Whenever a project makes use of a certain part, the quantity of that part committed to the given project is also recorded. Suppose that the system requires the user or file designer to declare or define the data in terms of tree structures. Then, any one of the hierarchical structures may be adopted for the information mentioned above (see Structures 1–5).

Structure 1. Projects Subordinate to Parts

| *File* | *Segment* | *Fields* |
|---|---|---|
| F | PART | part # |
| | | part name |
| | | part description |
| | | quantity-on-hand |
| | | quantity-on-order |
| | PROJECT | project # |
| | | project name |
| | | project description |
| | | quantity committed |

Structure 2. Parts Subordinate to Projects

| *File* | *Segment* | *Fields* |
|---|---|---|
| F | PROJECT | project # |
| | | project name |
| | | project description |
| | PART | part # |
| | | part name |
| | | part description |
| | | quantity-on-hand |
| | | quantity-on-order |
| | | quantity committed |

Structure 3. Parts and Projects as Peers Commitment Relationship Subordinate to Projects

| *File* | *Segment* | *Fields* |
|---|---|---|
| F | PART | part # |
| | | part name |
| | | part description |
| | | quantity-on-hand |
| | | quantity-on-order |
| G | PROJECT | project # |
| | | project name |
| | | project description |
| | PART | part # |
| | | quantity committed |

Structure 4. Parts and Projects as Peers Commitment Relationship Subordinate to Parts

| *File* | *Segment* | *Fields* |
|---|---|---|
| F | PART | part # |
| | | part description |
| | | quantity-on-hand |
| | | quantity-on-order |
| | PROJECT | project # |
| | | quantity committed |
| G | PROJECT | project # |
| | | project name |
| | | project description |

Structure 5. Parts, Projects, and Commitment Relationship as Peers

| *File* | *Segment* | *Fields* |
|---|---|---|
| F | PART | part # |
| | | part name |
| | | part description |
| | | quantity-on-hand |
| | | quantity-on-order |
| G | PROJECT | project # |
| | | project name |
| | | project description |
| H | COMMIT | part # |
| | | project # |
| | | quantity committed |

Now, consider the problem of printing out the part number, part name, and quantity committed for every part used in the project whose project name is "alpha." The following observations may be made regardless of which available tree-oriented information system is selected to tackle this problem. If a program $P$ is developed for this problem assuming one of the five structures above—that is, $P$ makes no test to determine which structure is in effect—then $P$ will fail on at least three of the remaining structures. More specifically, if $P$ succeeds with structure 5, it will fail with all the others; if $P$ succeeds with structure 3 or 4, it will fail with at least 1, 2, and 5; if $P$ succeeds with 1 or 2, it will fail with at least 3, 4, and 5. The reason is simple in each case. In the absence of a test to determine which structure is in effect, $P$ fails because an attempt is made to exceute a reference to a nonexistent file (available systems treat this as an error) or no attempt is made to execute a reference to a file containing needed information. The reader who is not convinced should develop sample programs for this simple problem.

Since, in general, it is not practical to develop application programs which test for all tree structurings permitted by the system, these programs fail when a change in structure becomes necessary.

Systems which provide users with a network model of the data run into similar difficulties. In both the tree and network cases, the user (or his program) is required to exploit a collection of user access paths to the data. It does not matter whether these paths are in close correspondence with pointer-defined paths in the stored representation—in IDS the correspondence is extremely simple, in TDMS it is just the opposite. The consequence, regardless of the stored representation, is that terminal activities and programs become dependent on the continued existence of the user access paths.

One solution to this is to adopt the policy that once a user access path is defined it will not be made obsolete until all application programs using that path have become obsolete. Such a policy is not practical, because the number of access paths in the total model for the community of users of a data bank would eventually become excessively large.

### 1.3. A Relational View of Data

The term *relation* is used here in its accepted mathematical sense. Given sets $S_1, S_2, \cdots, S_n$ (not necessarily distinct), $R$ is a relation on these $n$ sets if it is a set of $n$-tuples each of which has its first element from $S_1$, its second element from $S_2$, and so on.[1] We shall refer to $S_j$ as the $j$th *domain* of $R$. As defined above, $R$ is said to have *degree* $n$. Relations of degree 1 are often called *unary*, degree 2 *binary*, degree 3 *ternary*, and degree $n$ *n-ary*.

For expository reasons, we shall frequently make use of an array representation of relations, but it must be remembered that this particular representation is not an essential part of the relational view being expounded. An array which represents an $n$-ary relation $R$ has the following properties:

(1) Each row represents an $n$-tuple of $R$.
(2) The ordering of rows is immaterial.
(3) All rows are distinct.
(4) The ordering of columns is significant—it corresponds to the ordering $S_1, S_2, \cdots, S_n$ of the domains on which $R$ is defined (see, however, remarks below on domain-ordered and domain-unordered relations).
(5) The significance of each column is partially conveyed by labeling it with the name of the corresponding domain.

The example in Figure 1 illustrates a relation of degree 4, called *supply*, which reflects the shipments-in-progress of parts from specified suppliers to specified projects in specified quantities.

| *supply* | (*supplier* | *part* | *project* | *quantity*) |
|---|---|---|---|---|
| | 1 | 2 | 5 | 17 |
| | 1 | 3 | 5 | 23 |
| | 2 | 3 | 7 | 9 |
| | 2 | 7 | 5 | 4 |
| | 4 | 1 | 1 | 12 |

Fig. 1. A relation of degree 4

One might ask: If the columns are labeled by the name of corresponding domains, why should the ordering of columns matter? As the example in Figure 2 shows, two columns may have identical headings (indicating identical domains) but possess distinct meanings with respect to the relation. The relation depicted is called *component*. It is a ternary relation, whose first two domains are called *part* and third domain is called *quantity*. The meaning of *component* $(x, y, z)$ is that part $x$ is an immediate component (or subassembly) of part $y$, and $z$ units of part $x$ are needed to assemble one unit of part $y$. It is a relation which plays a critical role in the parts explosion problem.

| *component* | (*part* | *part* | *quantity*) |
|---|---|---|---|
| | 1 | 5 | 9 |
| | 2 | 5 | 7 |
| | 3 | 5 | 2 |
| | 2 | 6 | 12 |
| | 3 | 6 | 3 |
| | 4 | 7 | 1 |
| | 6 | 7 | 1 |

Fig. 2. A relation with two identical domains

It is a remarkable fact that several existing information systems (chiefly those based on tree-structured files) fail to provide data representations for relations which have two or more identical domains. The present version of IMS/360 [5] is an example of such a system.

The totality of data in a data bank may be viewed as a collection of time-varying relations. These relations are of assorted degrees. As time progresses, each $n$-ary relation may be subject to insertion of additional $n$-tuples, deletion of existing ones, and alteration of components of any of its existing $n$-tuples.

[1] More concisely, $R$ is a subset of the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$.

In many commercial, governmental, and scientific data banks, however, some of the relations are of quite high degree (a degree of 30 is not at all uncommon). Users should not normally be burdened with remembering the domain ordering of any relation (for example, the ordering *supplier*, then *part*, then *project*, then *quantity* in the relation *supply*). Accordingly, we propose that users deal, not with relations which are domain-ordered, but with *relationships* which are their domain-unordered counterparts.[2] To accomplish this, domains must be uniquely identifiable at least within any given relation, without using position. Thus, where there are two or more identical domains, we require in each case that the domain name be qualified by a distinctive *role name*, which serves to identify the role played by that domain in the given relation. For example, in the relation *component* of Figure 2, the first domain *part* might be qualified by the role name *sub*, and the second by *super*, so that users could deal with the relationship *component* and its domains—*sub.part super.part, quantity*—without regard to any ordering between these domains.

To sum up, it is proposed that most users should interact with a relational model of the data consisting of a collection of time-varying relationships (rather than relations). Each user need not know more about any relationship than its name together with the names of its domains (role qualified whenever necessary).[3] Even this information might be offered in menu style by the system (subject to security and privacy constraints) upon request by the user.

There are usually many alternative ways in which a relational model may be established for a data bank. In order to discuss a preferred way (or normal form), we must first introduce a few additional concepts (active domain, primary key, foreign key, nonsimple domain) and establish some links with terminology currently in use in information systems programming. In the remainder of this paper, we shall not bother to distinguish between relations and relationships except where it appears advantageous to be explicit.

Consider an example of a data bank which includes relations concerning parts, projects, and suppliers. One relation called *part* is defined on the following domains:

(1) part number
(2) part name
(3) part color
(4) part weight
(5) quantity on hand
(6) quantity on order

and possibly other domains as well. Each of these domains is, in effect, a pool of values, some or all of which may be represented in the data bank at any instant. While it is conceivable that, at some instant, all part colors are present, it is unlikely that all possible part weights, part names, and part numbers are. We shall call the set of values represented at some instant the *active domain* at that instant.

Normally, one domain (or combination of domains) of a given relation has values which uniquely identify each element ($n$-tuple) of that relation. Such a domain (or combination) is called a *primary key*. In the example above, part number would be a primary key, while part color would not be. A primary key is *nonredundant* if it is either a simple domain (not a combination) or a combination such that none of the participating simple domains is superfluous in uniquely identifying each element. A relation may possess more than one nonredundant primary key. This would be the case in the example if different parts were always given distinct names. Whenever a relation has two or more nonredundant primary keys, one of them is arbitrarily selected and called *the* primary key of that relation.

A common requirement is for elements of a relation to cross-reference other elements of the same relation or elements of a different relation. Keys provide a user-oriented means (but not the only means) of expressing such cross-references. We shall call a domain (or domain combination) of relation $R$ a *foreign key* if it is not the primary key of $R$ but its elements are values of the primary key of some relation $S$ (the possibility that $S$ and $R$ are identical is not excluded). In the relation *supply* of Figure 1, the combination of *supplier, part, project* is the primary key, while each of these three domains taken separately is a foreign key.

In previous work there has been a strong tendency to treat the data in a data bank as consisting of two parts, one part consisting of entity descriptions (for example, descriptions of suppliers) and the other part consisting of relations between the various entities or types of entities (for example, the *supply* relation). This distinction is difficult to maintain when one may have foreign keys in any relation whatsoever. In the user's relational model there appears to be no advantage to making such a distinction (there may be some advantage, however, when one applies relational concepts to machine representations of the user's set of relationships).

So far, we have discussed examples of relations which are defined on simple domains—domains whose elements are atomic (nondecomposable) values. Nonatomic values can be discussed within the relational framework. Thus, some domains may have relations as elements. These relations may, in turn, be defined on nonsimple domains, and so on. For example, one of the domains on which the relation *employee* is defined might be *salary history*. An element of the salary history domain is a binary relation defined on the domain *date* and the domain *salary*. The *salary history* domain is the set of all such binary relations. At any instant of time there are as many instances of the *salary history* relation in the data bank as there are employees. In contrast, there is only one instance of the *employee* relation.

The terms attribute and repeating group in present data base terminology are roughly analogous to simple domain

[2] In mathematical terms, a relationship is an equivalence class of those relations that are equivalent under permutation of domains (see Section 2.1.1).

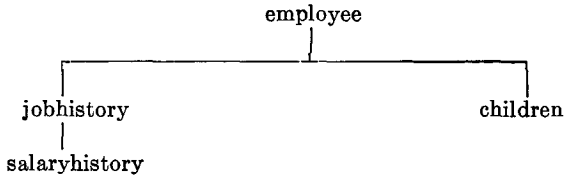
[3] Naturally, as with any data put into and retrieved from a computer system, the user will normally make far more effective use of the data if he is aware of its meaning.

and nonsimple domain, respectively. Much of the confusion in present terminology is due to failure to distinguish between type and instance (as in "record") and between components of a user model of the data on the one hand and their machine representation counterparts on the other hand (again, we cite "record" as an example).

### 1.4. Normal Form

A relation whose domains are all simple can be represented in storage by a two-dimensional column-homogeneous array of the kind discussed above. Some more complicated data structure is necessary for a relation with one or more nonsimple domains. For this reason (and others to be cited below) the possibility of eliminating nonsimple domains appears worth investigating.[4] There is, in fact, a very simple elimination procedure, which we shall call normalization.

Consider, for example, the collection of relations exhibited in Figure 3(a). *Job history* and *children* are nonsimple domains of the relation *employee*. *Salary history* is a nonsimple domain of the relation *job history*. The tree in Figure 3(a) shows just these interrelationships of the nonsimple domains.




employee (*man#*, name, birthdate, jobhistory, children)
jobhistory (*jobdate*, title, salaryhistory)
salaryhistory (*salarydate*, salary)
children (*childname*, birthyear)

Fig. 3(a). Unnormalized set

employee′ (*man#*, name, birthdate)
jobhistory′ (*man#*, *jobdate*, title)
salaryhistory′ (*man#*, *jobdate*, *salarydate*, salary)
children′ (*man#*, *childname*, birthyear)

Fig. 3(b). Normalized set

Normalization proceeds as follows. Starting with the relation at the top of the tree, take its primary key and expand each of the immediately subordinate relations by inserting this primary key domain or domain combination. The primary key of each expanded relation consists of the primary key before expansion augmented by the primary key copied down from the parent relation. Now, strike out from the parent relation all nonsimple domains, remove the top node of the tree, and repeat the same sequence of operations on each remaining subtree.

The result of normalizing the collection of relations in Figure 3(a) is the collection in Figure 3(b). The primary key of each relation is italicized to show how such keys are expanded by the normalization.

[4] M. E. Sanko of IBM, San Jose, independently recognized the desirability of eliminating nonsimple domains.

If normalization as described above is to be applicable, the unnormalized collection of relations must satisfy the following conditions:

(1) The graph of interrelationships of the nonsimple domains is a collection of trees.

(2) No primary key has a component domain which is nonsimple.

The writer knows of no application which would require any relaxation of these conditions. Further operations of a normalizing kind are possible. These are not discussed in this paper.

The simplicity of the array representation which becomes feasible when all relations are cast in normal form is not only an advantage for storage purposes but also for communication of bulk data between systems which use widely different representations of the data. The communication form would be a suitably compressed version of the array representation and would have the following advantages:

(1) It would be devoid of pointers (address-valued or displacement-valued).

(2) It would avoid all dependence on hash addressing schemes.

(3) It would contain no indices or ordering lists.

If the user's relational model is set up in normal form, names of items of data in the data bank can take a simpler form than would otherwise be the case. A general name would take a form such as

$$R(g).r.d$$

where $R$ is a relational name; $g$ is a generation identifier (optional); $r$ is a role name (optional); $d$ is a domain name. Since $g$ is needed only when several generations of a given relation exist, or are anticipated to exist, and $r$ is needed only when the relation $R$ has two or more domains named $d$, the simple form $R.d$ will often be adequate.

### 1.5. Some Linguistic Aspects

The adoption of a relational model of data, as described above, permits the development of a universal data sublanguage based on an applied predicate calculus. A first-order predicate calculus suffices if the collection of relations is in normal form. Such a language would provide a yardstick of linguistic power for all other proposed data languages, and would itself be a strong candidate for embedding (with appropriate syntactic modification) in a variety of host languages (programming, command- or problem-oriented). While it is not the purpose of this paper to describe such a language in detail, its salient features would be as follows.

Let us denote the data sublanguage by $R$ and the host language by $H$. $R$ permits the declaration of relations and their domains. Each declaration of a relation identifies the primary key for that relation. Declared relations are added to the system catalog for use by any members of the user community who have appropriate authorization. $H$ permits supporting declarations which indicate, perhaps less permanently, how these relations are represented in stor-

age. $R$ permits the specification for retrieval of any subset of data from the data bank. Action on such a retrieval request is subject to security constraints.

The universality of the data sublanguage lies in its descriptive ability (not its computing ability). In a large data bank each subset of the data has a very large number of possible (and sensible) descriptions, even when we assume (as we do) that there is only a finite set of function subroutines to which the system has access for use in qualifying data for retrieval. Thus, the class of qualification expressions which can be used in a set specification must have the descriptive power of the class of well-formed formulas of an applied predicate calculus. It is well known that to preserve this descriptive power it is unnecessary to express (in whatever syntax is chosen) every formula of the selected predicate calculus. For example, just those in prenex normal form are adequate [9].

Arithmetic functions may be needed in the qualification or other parts of retrieval statements. Such functions can be defined in $H$ and invoked in $R$.

A set so specified may be fetched for query purposes only, or it may be held for possible changes. Insertions take the form of adding new elements to declared relations without regard to any ordering that may be present in their machine representation. Deletions which are effective for the community (as opposed to the individual user or subcommunities) take the form of removing elements from declared relations. Some deletions and updates may be triggered by others, if deletion and update dependencies between specified relations are declared in $R$.

One important effect that the view adopted toward data has on the language used to retrieve it is in the naming of data elements and sets. Some aspects of this have been discussed in the previous section. With the usual network view, users will often be burdened with coining and using more relation names than are absolutely necessary, since names are associated with paths (or path types) rather than with relations.

Once a user is aware that a certain relation is stored, he will expect to be able to exploit[5] it using any combination of its arguments as "knowns" and the remaining arguments as "unknowns," because the information (like Everest) is there. This is a system feature (missing from many current information systems) which we shall call (logically) *symmetric exploitation* of relations. Naturally, symmetry in performance is not to be expected.

To support symmetric exploitation of a single binary relation, two directed paths are needed. For a relation of degree $n$, the number of paths to be named and controlled is $n$ factorial.

Again, if a relational view is adopted in which every $n$-ary relation ($n > 2$) has to be expressed by the user as a nested expression involving only binary relations (see Feldman's LEAP System [10], for example) then $2n - 1$ names have to be coined instead of only $n + 1$ with direct $n$-ary notation as described in Section 1.2. For example, the 4-ary relation *supply* of Figure 1, which entails 5 names in $n$-ary notation, would be represented in the form

$$P\ (supplier,\ Q\ (part,\ R\ (project,\ quantity)))$$

in nested binary notation and, thus, employ 7 names.

A further disadvantage of this kind of expression is its asymmetry. Although this asymmetry does not prohibit symmetric exploitation, it certainly makes some bases of interrogation very awkward for the user to express (consider, for example, a query for those parts and quantities related to certain given projects via $Q$ and $R$).

### 1.6. Expressible, Named, and Stored Relations

Associated with a data bank are two collections of relations: the *named set* and the *expressible set*. The named set is the collection of all those relations that the community of users can identify by means of a simple name (or identifier). A relation $R$ acquires membership in the named set when a suitably authorized user declares $R$; it loses membership when a suitably authorized user cancels the declaration of $R$.

The expressible set is the total collection of relations that can be designated by expressions in the data language. Such expressions are constructed from simple names of relations in the named set; names of generations, roles and domains; logical connectives; the quantifiers of the predicate calculus;[6] and certain constant relation symbols such as $=$, $>$. The named set is a subset of the expressible set—usually a very small subset.

Since some relations in the named set may be time-independent combinations of others in that set, it is useful to consider associating with the named set a collection of statements that define these time-independent constraints. We shall postpone further discussion of this until we have introduced several operations on relations (see Section 2).

One of the major problems confronting the designer of a data system which is to support a relational model for its users is that of determining the class of stored representations to be supported. Ideally, the variety of permitted data representations should be just adequate to cover the spectrum of performance requirements of the total collection of installations. Too great a variety leads to unnecessary overhead in storage and continual reinterpretation of descriptions for the structures currently in effect.

For any selected class of stored representations the data system must provide a means of translating user requests expressed in the data language of the relational model into corresponding—and efficient—actions on the current stored representation. For a high level data language this presents a challenging design problem. Nevertheless, it is a problem which must be solved—as more users obtain concurrent access to a large data bank, responsibility for providing efficient response and throughput shifts from the individual user to the data system.

[5] Exploiting a relation includes query, update, and delete.

[6] Because each relation in a practical data bank is a finite set at every instant of time, the existential and universal quantifiers can be expressed in terms of a function that counts the number of elements in any finite set.

## 2. Redundancy and Consistency

### 2.1. Operations on Relations

Since relations are sets, all of the usual set operations are applicable to them. Nevertheless, the result may not be a relation; for example, the union of a binary relation and a ternary relation is not a relation.

The operations discussed below are specifically for relations. These operations are introduced because of their key role in deriving relations from other relations. Their principal application is in noninferential information systems—systems which do not provide logical inference services—although their applicability is not necessarily destroyed when such services are added.

Most users would not be directly concerned with these operations. Information systems designers and people concerned with data bank control should, however, be thoroughly familiar with them.

2.1.1. *Permutation.* A binary relation has an array representation with two columns. Interchanging these columns yields the converse relation. More generally, if a permutation is applied to the columns of an $n$-ary relation, the resulting relation is said to be a *permutation* of the given relation. There are, for example, $4! = 24$ permutations of the relation *supply* in Figure 1, if we include the identity permutation which leaves the ordering of columns unchanged.

Since the user's relational model consists of a collection of relationships (domain-unordered relations), permutation is not relevant to such a model considered in isolation. It is, however, relevant to the consideration of stored representations of the model. In a system which provides symmetric exploitation of relations, the set of queries answerable by a stored relation is identical to the set answerable by any permutation of that relation. Although it is logically unnecessary to store both a relation and some permutation of it, performance considerations could make it advisable.

2.1.2. *Projection.* Suppose now we select certain columns of a relation (striking out the others) and then remove from the resulting array any duplication in the rows. The final array represents a relation which is said to be a *projection* of the given relation.

A selection operator $\pi$ is used to obtain any desired permutation, projection, or combination of the two operations. Thus, if $L$ is a list of $k$ indices[7] $L = i_1, i_2, \cdots, i_k$ and $R$ is an $n$-ary relation ($n \geq k$), then $\pi_L(R)$ is the $k$-ary relation whose $j$th column is column $i_j$ of $R$ ($j = 1, 2, \cdots, k$) except that duplication in resulting rows is removed. Consider the relation *supply* of Figure 1. A permuted projection of this relation is exhibited in Figure 4. Note that, in this particular case, the projection has fewer $n$-tuples than the relation from which it is derived.

2.1.3. *Join.* Suppose we are given two binary relations, which have some domain in common. Under what circumstances can we combine these relations to form a ternary relation which preserves all of the information in the given relations?

The example in Figure 5 shows two relations $R$, $S$, which are joinable without loss of information, while Figure 6 shows a join of $R$ with $S$. A binary relation $R$ is *joinable* with a binary relation $S$ if there exists a ternary relation $U$ such that $\pi_{12}(U) = R$ and $\pi_{23}(U) = S$. Any such ternary relation is called a *join* of $R$ with $S$. If $R$, $S$ are binary relations such that $\pi_2(R) = \pi_1(S)$, then $R$ is joinable with $S$. One join that always exists in such a case is the *natural join* of $R$ with $S$ defined by

$$R*S = \{(a, b, c) : R(a, b) \wedge S(b, c)\}$$

where $R(a, b)$ has the value *true* if $(a, b)$ is a member of $R$ and similarly for $S(b, c)$. It is immediate that

$$\pi_{12}(R*S) = R$$

and

$$\pi_{23}(R*S) = S.$$

Note that the join shown in Figure 6 is the natural join of $R$ with $S$ from Figure 5. Another join is shown in Figure 7.

$\Pi_{31}$(*supply*)

| (*project* | *supplier*) |
|---|---|
| 5 | 1 |
| 5 | 2 |
| 1 | 4 |
| 7 | 2 |

Fig. 4. A permuted projection of the relation in Figure 1

R

| (*supplier* | *part*) |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 2 | 2 |

S

| (*part* | *project*) |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |

Fig. 5. Two joinable relations

R*S

| (*supplier* | *part* | *project*) |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 2 | 1 | 1 |
| 2 | 1 | 2 |
| 2 | 2 | 1 |

Fig. 6. The natural join of R with S (from Figure 5)

U

| (*supplier* | *part* | *project*) |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 1 | 1 |
| 2 | 2 | 1 |

Fig. 7. Another join of R with S (from Figure 5)

Inspection of these relations reveals an element (element 1) of the domain *part* (the domain on which the join is to be made) with the property that it possesses more than one relative under $R$ and also under $S$. It is this ele-

[7] When dealing with relationships, we use domain names (role-qualified whenever necessary) instead of domain positions.

ment which gives rise to the plurality of joins. Such an element in the joining domain is called a *point of ambiguity* with respect to the joining of $R$ with $S$.

If either $\pi_{21}(R)$ or $S$ is a function,[8] no point of ambiguity can occur in joining $R$ with $S$. In such a case, the natural join of $R$ with $S$ is the only join of $R$ with $S$. Note that the reiterated qualification "of $R$ with $S$" is necessary, because $S$ might be joinable with $R$ (as well as $R$ with $S$), and this join would be an entirely separate consideration. In Figure 5, none of the relations $R$, $\pi_{21}(R)$, $S$, $\pi_{21}(S)$ is a function.

Ambiguity in the joining of $R$ with $S$ can sometimes be resolved by means of other relations. Suppose we are given, or can derive from sources independent of $R$ and $S$, a relation $T$ on the domains *project* and *supplier* with the following properties:

$$(1)\ \pi_1(T) = \pi_2(S),$$

$$(2)\ \pi_2(T) = \pi_1(R),$$

$$(3)\ T(j, s) \rightarrow \exists p(R(S, p) \wedge S(p, j)),$$

$$(4)\ R(s, p) \rightarrow \exists j(S(p, j) \wedge T(j, s)),$$

$$(5)\ S(p, j) \rightarrow \exists s(T(j, s) \wedge R(s, p)),$$

then we may form a three-way join of $R$, $S$, $T$; that is, a ternary relation such that

$$\pi_{12}(U) = R, \qquad \pi_{23}(U) = S, \qquad \pi_{31}(U) = T.$$

Such a join will be called a *cyclic 3-join* to distinguish it from a *linear 3-join* which would be a quaternary relation $V$ such that

$$\pi_{12}(V) = R, \qquad \pi_{23}(V) = S, \qquad \pi_{34}(V) = T.$$

While it is possible for more than one cyclic 3-join to exist (see Figures 8, 9, for an example), the circumstances under which this can occur entail much more severe constraints

| R | ($s$ | $p$) | S | ($p$ | $j$) | T | ($j$ | $s$) |
|---|---|---|---|---|---|---|---|---|
| | 1 | a | | a | d | | d | 1 |
| | 2 | a | | a | e | | d | 2 |
| | 2 | b | | b | d | | e | 2 |
| | | | | b | e | | e | 2 |

FIG. 8. Binary relations with a plurality of cyclic 3-joins

| U | ($s$ | $p$ | $j$) | U′ | ($s$ | $p$ | $j$) |
|---|---|---|---|---|---|---|---|
| | 1 | a | d | | 1 | a | d |
| | 2 | a | e | | 2 | a | d |
| | 2 | b | d | | 2 | a | e |
| | 2 | b | e | | 2 | b | d |
| | | | | | 2 | b | e |

FIG. 9. Two cyclic 3-joins of the relations in Figure 8

than those for a plurality of 2-joins. To be specific, the relations $R$, $S$, $T$ must possess points of ambiguity with respect to joining $R$ with $S$ (say point $x$), $S$ with $T$ (say $y$), and $T$ with $R$ (say $z$), and, furthermore, $y$ must be a relative of $x$ under $S$, $z$ a relative of $y$ under $T$, and $x$ a relative of $z$ under $R$. Note that in Figure 8 the points $x = a$; $y = d$; $z = 2$ have this property.

The natural linear 3-join of three binary relations $R$, $S$, $T$ is given by

$$R*S*T = \{(a, b, c, d): R(a, b) \wedge S(b, c) \wedge T(c, d)\}$$

where parentheses are not needed on the left-hand side because the natural 2-join ($*$) is associative. To obtain the cyclic counterpart, we introduce the operator $\gamma$ which produces a relation of degree $n - 1$ from a relation of degree $n$ by tying its ends together. Thus, if $R$ is an $n$-ary relation ($n \geq 2$), the *tie* of $R$ is defined by the equation

$$\gamma(R) = \{(a_1, a_2, \cdots, a_{n-1}): R(a_1, a_2, \cdots, a_{n-1}, a_n) \wedge a_1 = a_n\}.$$

We may now represent the natural cyclic 3-join of $R$, $S$, $T$ by the expression

$$\gamma(R*S*T).$$

Extension of the notions of linear and cyclic 3-join and their natural counterparts to the joining of $n$ binary relations (where $n \geq 3$) is obvious. A few words may be appropriate, however, regarding the joining of relations which are not necessarily binary. Consider the case of two relations $R$ (degree $r$), $S$ (degree $s$) which are to be joined on $p$ of their domains ($p < r$, $p < s$). For simplicity, suppose these $p$ domains are the last $p$ of the $r$ domains of $R$, and the first $p$ of the $s$ domains of $S$. If this were not so, we could always apply appropriate permutations to make it so. Now, take the Cartesian product of the first $r$-$p$ domains of $R$, and call this new domain $A$. Take the Cartesian product of the last $p$ domains of $R$, and call this $B$. Take the Cartesian product of the last $s$-$p$ domains of $S$ and call this $C$.

We can treat $R$ as if it were a binary relation on the domains $A$, $B$. Similarly, we can treat $S$ as if it were a binary relation on the domains $B$, $C$. The notions of linear and cyclic 3-join are now directly applicable. A similar approach can be taken with the linear and cyclic $n$-joins of $n$ relations of assorted degrees.

2.1.4. *Composition.* The reader is probably familiar with the notion of composition applied to functions. We shall discuss a generalization of that concept and apply it first to binary relations. Our definitions of composition and composability are based very directly on the definitions of join and joinability given above.

Suppose we are given two relations $R$, $S$. $T$ is a *composition* of $R$ with $S$ if there exists a join $U$ of $R$ with $S$ such that $T = \pi_{13}(U)$. Thus, two relations are composable if and only if they are joinable. However, the existence of more than one join of $R$ with $S$ does not imply the existence of more than one composition of $R$ with $S$.

Corresponding to the natural join of $R$ with $S$ is the

[8] A function is a binary relation, which is one-one or many-one, but not one-many.

*natural composition*[9] of $R$ with $S$ defined by

$$R \cdot S = \pi_{13}(R * S).$$

Taking the relations $R$, $S$ from Figure 5, their natural composition is exhibited in Figure 10 and another composition is exhibited in Figure 11 (derived from the join exhibited in Figure 7).

| R·S | (*project* | *supplier*) |
|---|---|---|
| | 1 | 1 |
| | 1 | 2 |
| | 2 | 1 |
| | 2 | 2 |

FIG. 10. The natural composition of R with S (from Figure 5)

| T | (*project* | *supplier*) |
|---|---|---|
| | 1 | 2 |
| | 2 | 1 |

FIG. 11. Another composition of R with S (from Figure 5)

When two or more joins exist, the number of distinct compositions may be as few as one or as many as the number of distinct joins. Figure 12 shows an example of two relations which have several joins but only one composition. Note that the ambiguity of point $c$ is lost in composing $R$ with $S$, because of unambiguous associations made via the points $a$, $b$, $d$, $e$.

| R | (*supplier* | *part*) | S | (*part* | *project*) |
|---|---|---|---|---|---|
| | 1 | a | | a | g |
| | 1 | b | | b | f |
| | 1 | c | | c | f |
| | 2 | c | | c | g |
| | 2 | d | | d | g |
| | 2 | e | | e | f |

FIG. 12. Many joins, only one composition

Extension of composition to pairs of relations which are not necessarily binary (and which may be of different degrees) follows the same pattern as extension of pairwise joining to such relations.

A lack of understanding of relational composition has led several systems designers into what may be called the *connection trap*. This trap may be described in terms of the following example. Suppose each supplier description is linked by pointers to the descriptions of each part supplied by that supplier, and each part description is similarly linked to the descriptions of each project which uses that part. A conclusion is now drawn which is, in general, erroneous: namely that, if all possible paths are followed from a given supplier via the parts he supplies to the projects using those parts, one will obtain a valid set of all projects supplied by that supplier. Such a conclusion is correct only in the very special case that the target relation between projects and suppliers is, in fact, the natural composition of the other two relations—and we must normally add the phrase "for all time," because this is usually implied in claims concerning path-following techniques.

[9] Other writers tend to ignore compositions other than the natural one, and accordingly refer to this particular composition as *the* composition—see, for example, Kelley's "General Topology."

2.1.5. *Restriction.* A subset of a relation is a relation. One way in which a relation $S$ may act on a relation $R$ to generate a subset of $R$ is through the operation *restriction* of $R$ by $S$. This operation is a generalization of the restriction of a function to a subset of its domain, and is defined as follows.

Let $L$, $M$ be equal-length lists of indices such that $L = i_1, i_2, \cdots, i_k$, $M = j_1, j_2, \cdots, j_k$ where $k \leqq$ degree of $R$ and $k \leqq$ degree of $S$. Then the $L, M$ restriction of $R$ by $S$ denoted $R_{L|M}S$ is the maximal subset $R'$ of $R$ such that

$$\pi_L(R') = \pi_M(S).$$

The operation is defined only if equality is applicable between elements of $\pi_{i_h}(R)$ on the one hand and $\pi_{j_h}(S)$ on the other for all $h = 1, 2, \cdots, k$.

The three relations $R$, $S$, $R'$ of Figure 13 satisfy the equation $R' = R_{(2,3)|(1,2)}S$.

| R | (*s* | *p* | *j*) | S | (*p* | *j*) | R′ | (*s* | *p* | *j*) |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | a | A | | a | A | | 1 | a | A |
| | 2 | a | A | | c | B | | 2 | a | A |
| | 2 | a | B | | b | B | | 2 | b | B |
| | 2 | b | A | | | | | | | |
| | 2 | b | B | | | | | | | |

FIG. 13. Example of restriction

We are now in a position to consider various applications of these operations on relations.

## 2.2. Redundancy

Redundancy in the named set of relations must be distinguished from redundancy in the stored set of representations. We are primarily concerned here with the former. To begin with, we need a precise notion of derivability for relations.

Suppose $\theta$ is a collection of operations on relations and each operation has the property that from its operands it yields a unique relation (thus natural join is eligible, but join is not). A relation $R$ is *$\theta$-derivable* from a set $S$ of relations if there exists a sequence of operations from the collection $\theta$ which, for all time, yields $R$ from members of $S$. The phrase "for all time" is present, because we are dealing with time-varying relations, and our interest is in derivability which holds over a significant period of time. For the named set of relationships in noninferential systems, it appears that an adequate collection $\theta_1$ contains the following operations: projection, natural join, tie, and restriction. Permutation is irrelevant and natural composition need not be included, because it is obtainable by taking a natural join and then a projection. For the stored set of representations, an adequate collection $\theta_2$ of operations would include permutation and additional operations concerned with subsetting and merging relations, and ordering and connecting their elements.

2.2.1. *Strong Redundancy.* A set of relations is *strongly redundant* if it contains at least one relation that possesses a projection which is derivable from other projections of relations in the set. The following two examples are intended to explain why strong redundancy is defined this way, and to demonstrate its practical use. In the first ex-

ample the collection of relations consists of just the following relation:

*employee* (*serial #*, *name*, *manager#*, *managername*)

with *serial#* as the primary key and *manager#* as a foreign key. Let us denote the active domain by $\Delta_t$, and suppose that

$$\Delta_t(manager\#) \subset \Delta_t(serial\#)$$

and

$$\Delta_t(managername) \subset \Delta_t(name)$$

for all time $t$. In this case the redundancy is obvious: the domain *managername* is unnecessary. To see that it is a strong redundancy as defined above, we observe that

$$\pi_{34}(employee) = \pi_{12}(employee)_1|_1\pi_3(employee).$$

In the second example the collection of relations includes a relation $S$ describing suppliers with primary key $s\#$, a relation $D$ describing departments with primary key $d\#$, a relation $J$ describing projects with primary key $j\#$, and the following relations:

$$P(s\#, d\#, \cdots), \qquad Q(s\#, j\#, \cdots), \qquad R(d\#, j\#, \cdots),$$

where in each case $\cdots$ denotes domains other than $s\#$, $d\#$, $j\#$. Let us suppose the following condition $C$ is known to hold independent of time: supplier $s$ supplies department $d$ (relation $P$) if and only if supplier $s$ supplies some project $j$ (relation $Q$) to which $d$ is assigned (relation $R$). Then, we can write the equation

$$\pi_{12}(P) = \pi_{12}(Q) \cdot \pi_{21}(R)$$

and thereby exhibit a strong redundancy.

An important reason for the existence of strong redundancies in the named set of relationships is user convenience. A particular case of this is the retention of semi-obsolete relationships in the named set so that old programs that refer to them by name can continue to run correctly. Knowledge of the existence of strong redundancies in the named set enables a system or data base administrator greater freedom in the selection of stored representations to cope more efficiently with current traffic. If the strong redundancies in the named set are directly reflected in strong redundancies in the stored set (or if other strong redundancies are introduced into the stored set), then, generally speaking, extra storage space and update time are consumed with a potential drop in query time for some queries and in load on the central processing units.

2.2.2. *Weak Redundancy.* A second type of redundancy may exist. In contrast to strong redundancy it is not characterized by an equation. A collection of relations is *weakly redundant* if it contains a relation that has a projection which is not derivable from other members but is at all times a projection of *some* join of other projections of relations in the collection.

We can exhibit a weak redundancy by taking the second example (cited above) for a strong redundancy, and assuming now that condition $C$ does not hold at all times. The relations $\pi_{12}(P)$, $\pi_{12}(Q)$, $\pi_{12}(R)$ are complex[10] relations with the possibility of points of ambiguity occurring from time to time in the potential joining of any two. Under these circumstances, none of them is derivable from the other two. However, constraints do exist between them, since each is a projection of some cyclic join of the three of them. One of the weak redundancies can be characterized by the statement: for all time, $\pi_{12}(P)$ is *some* composition of $\pi_{12}(Q)$ with $\pi_{21}(R)$. The composition in question might be the natural one at some instant and a nonnatural one at another instant.

Generally speaking, weak redundancies are inherent in the logical needs of the community of users. They are not removable by the system or data base administrator. If they appear at all, they appear in both the named set and the stored set of representations.

2.3. Consistency

Whenever the named set of relations is redundant in either sense, we shall associate with that set a collection of statements which define all of the redundancies which hold independent of time between the member relations. If the information system lacks—and it most probably will—detailed semantic information about each named relation, it cannot deduce the redundancies applicable to the named set. It might, over a period of time, make attempts to *in*duce the redundancies, but such attempts would be fallible.

Given a collection $C$ of time-varying relations, an associated set $Z$ of constraint statements and an instantaneous value $V$ for $C$, we shall call the state $(C, Z, V)$ *consistent* or *inconsistent* according as $V$ does or does not satisfy $Z$. For example, given stored relations $R$, $S$, $T$ together with the constraint statement "$\pi_{12}(T)$ is a composition of $\pi_{12}(R)$ with $\pi_{12}(S)$", we may check from time to time that the values stored for $R$, $S$, $T$ satisfy this constraint. An algorithm for making this check would examine the first two columns of each of $R$, $S$, $T$ (in whatever way they are represented in the system) and determine whether

(1) $\pi_1(T) = \pi_1(R)$,

(2) $\pi_2(T) = \pi_2(S)$,

(3) for every element pair $(a, c)$ in the relation $\pi_{12}(T)$ there is an element $b$ such that $(a, b)$ is in $\pi_{12}(R)$ and $(b, c)$ is in $\pi_{12}(S)$.

There are practical problems (which we shall not discuss here) in taking an instantaneous snapshot of a collection of relations, some of which may be very large and highly variable.

It is important to note that consistency as defined above is a property of the instantaneous state of a data bank, and is independent of how that state came about. Thus, in particular, there is no distinction made on the basis of whether a user generated an inconsistency due to an act of omission or an act of commission. Examination of a simple

[10] A binary relation is complex if neither it nor its converse is a function.

example will show the reasonableness of this (possibly unconventional) approach to consistency.

Suppose the named set $C$ includes the relations $S$, $J$, $D$, $P$, $Q$, $R$ of the example in Section 2.2 and that $P$, $Q$, $R$ possess either the strong or weak redundancies described therein (in the particular case now under consideration, it does not matter which kind of redundancy occurs). Further, suppose that at some time $t$ the data bank state is consistent and contains no project $j$ such that supplier 2 supplies project $j$ and $j$ is assigned to department 5. Accordingly, there is no element (2, 5) in $\pi_{12}(P)$. Now, a user introduces the element (2, 5) into $\pi_{12}(P)$ by inserting some appropriate element into $P$. The data bank state is now inconsistent. The inconsistency could have arisen from an act of omission, if the input (2, 5) is correct, and there does exist a project $j$ such that supplier 2 supplies $j$ and $j$ is assigned to department 5. In this case, it is very likely that the user intends in the near future to insert elements into $Q$ and $R$ which will have the effect of introducing $(2, j)$ into $\pi_{12}(Q)$ and $(5, j)$ in $\pi_{12}(R)$. On the other hand, the input (2, 5) might have been faulty. It could be the case that the user intended to insert some other element into $P$—an element whose insertion would transform a consistent state into a consistent state. The point is that the system will normally have no way of resolving this question without interrogating its environment (perhaps the user who created the inconsistency).

There are, of course, several possible ways in which a system can detect inconsistencies and respond to them. In one approach the system checks for possible inconsistency whenever an insertion, deletion, or key update occurs. Naturally, such checking will slow these operations down. If an inconsistency has been generated, details are logged internally, and if it is not remedied within some reasonable time interval, either the user or someone responsible for the security and integrity of the data is notified. Another approach is to conduct consistency checking as a batch operation once a day or less frequently. Inputs causing the inconsistencies which remain in the data bank state at checking time can be tracked down if the system maintains a journal of all state-changing transactions. This latter approach would certainly be superior if few nontransitory inconsistencies occurred.

### 2.4. Summary

In Section 1 a relational model of data is proposed as a basis for protecting users of formatted data systems from the potentially disruptive changes in data representation caused by growth in the data bank and changes in traffic. A normal form for the time-varying collection of relationships is introduced.

In Section 2 operations on relations and two types of redundancy are defined and applied to the problem of maintaining the data in a consistent state. This is bound to become a serious practical problem as more and more different types of data are integrated together into common data banks.

Many questions are raised and left unanswered. For example, only a few of the more important properties of the data sublanguage in Section 1.4 are mentioned. Neither the purely linguistic details of such a language nor the implementation problems are discussed. Nevertheless, the material presented should be adequate for experienced systems programmers to visualize several approaches. It is also hoped that this paper can contribute to greater precision in work on formatted data systems.

*Acknowledgment.* It was C. T. Davies of IBM Poughkeepsie who convinced the author of the need for data independence in future information systems. The author wishes to thank him and also F. P. Palermo, C. P. Wang, E. B. Altman, and M. E. Senko of the IBM San Jose Research Laboratory for helpful discussions.

Received September, 1969; revised February, 1970

### References

1. Childs, D. L. Feasibility of a set-theoretical data structure —a general structure based on a reconstituted definition of relation. Proc. IFIP Cong., 1968, North Holland Pub. Co., Amsterdam, p. 162–172.
2. Levein, R. E., and Maron, M. E. A computer system for inference execution and data retrieval. *Comm. ACM 10*, 11 (Nov. 1967), 715–721.
3. Bachman, C. W. Software for random access processing. *Datamation* (Apr. 1965), 36–41.
4. McGee, W. C. Generalized file processing. In *Annual Review in Automatic Programming 5*, 13, Pergamon Press, New York, 1969, pp. 77–149.
5. Information Management System/360, Application Description Manual H20-0524-1. IBM Corp., White Plains, N. Y., July 1968.
6. GIS (Generalized Information System), Application Description Manual H20-0574. IBM Corp., White Plains, N. Y., 1965.
7. Bleier, R. E. Treating hierarchical data structures in the SDC time-shared data management system (TDMS). Proc. ACM 22nd Nat. Conf., 1967, MDI Publications, Wayne, Pa., pp. 41–49.
8. IDS Reference Manual GE 625/635, GE Inform. Sys. Div., Pheonix, Ariz., CPB 1093B, Feb. 1968.
9. Church, A. *An Introduction to Mathematical Logic I.* Princeton U. Press, Princeton, N.J., 1956.
10. Feldman, J. A., and Rovner, P. D. An Algol-based associative language. Stanford Artificial Intelligence Rep. AI-66, Aug. 1, 1968.

# Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank

**Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang,**
**Christopher D. Manning, Andrew Y. Ng and Christopher Potts**
Stanford University, Stanford, CA 94305, USA
richard@socher.org,{aperelyg,jcchuang,ang}@cs.stanford.edu
{jeaneis,manning,cgpotts}@stanford.edu

## Abstract

Semantic word spaces have been very useful but cannot express the meaning of longer phrases in a principled way. Further progress towards understanding compositionality in tasks such as sentiment detection requires richer supervised training and evaluation resources and more powerful models of composition. To remedy this, we introduce a Sentiment Treebank. It includes fine grained sentiment labels for 215,154 phrases in the parse trees of 11,855 sentences and presents new challenges for sentiment compositionality. To address them, we introduce the Recursive Neural Tensor Network. When trained on the new treebank, this model outperforms all previous methods on several metrics. It pushes the state of the art in single sentence positive/negative classification from 80% up to 85.4%. The accuracy of predicting fine-grained sentiment labels for all phrases reaches 80.7%, an improvement of 9.7% over bag of features baselines. Lastly, it is the only model that can accurately capture the effects of negation and its scope at various tree levels for both positive and negative phrases.

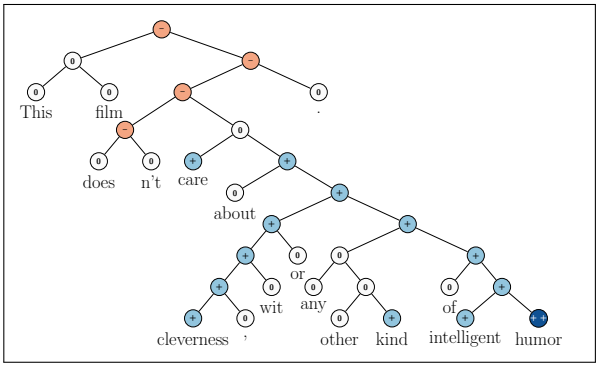



Figure 1: Example of the Recursive Neural Tensor Network accurately predicting 5 sentiment classes, very negative to very positive (– –, –, 0, +, + +), at every node of a parse tree and capturing the negation and its scope in this sentence.

## 1 Introduction

Semantic vector spaces for single words have been widely used as features (Turney and Pantel, 2010). Because they cannot capture the meaning of longer phrases properly, compositionality in semantic vector spaces has recently received a lot of attention (Mitchell and Lapata, 2010; Socher et al., 2010; Zanzotto et al., 2010; Yessenalina and Cardie, 2011; Socher et al., 2012; Grefenstette et al., 2013). However, progress is held back by the current lack of large and labeled compositionality resources and models to accurately capture the underlying phenomena presented in such data. To address this need, we introduce the Stanford Sentiment Treebank and a powerful Recursive Neural Tensor Network that can accurately predict the compositional semantic effects present in this new corpus.

The *Stanford Sentiment Treebank* is the first corpus with fully labeled parse trees that allows for a complete analysis of the compositional effects of sentiment in language. The corpus is based on the dataset introduced by Pang and Lee (2005) and consists of 11,855 single sentences extracted from movie reviews. It was parsed with the Stanford parser (Klein and Manning, 2003) and includes a total of 215,154 unique phrases from those parse trees, each annotated by 3 human judges. This new dataset allows us to analyze the intricacies of sentiment and to capture complex linguistic phenomena. Fig. 1 shows one of the many examples with clear compositional structure. The granularity and size of

this dataset will enable the community to train compositional models that are based on supervised and structured machine learning techniques. While there are several datasets with document and chunk labels available, there is a need to better capture sentiment from short comments, such as Twitter data, which provide less overall signal per document.

In order to capture the compositional effects with higher accuracy, we propose a new model called the Recursive Neural Tensor Network (RNTN). Recursive Neural Tensor Networks take as input phrases of any length. They represent a phrase through word vectors and a parse tree and then compute vectors for higher nodes in the tree using the same tensor-based composition function. We compare to several supervised, compositional models such as standard recursive neural networks (RNN) (Socher et al., 2011b), matrix-vector RNNs (Socher et al., 2012), and baselines such as neural networks that ignore word order, Naive Bayes (NB), bi-gram NB and SVM. All models get a significant boost when trained with the new dataset but the RNTN obtains the highest performance with 80.7% accuracy when predicting fine-grained sentiment for all nodes. Lastly, we use a test set of positive and negative sentences and their respective negations to show that, unlike bag of words models, the RNTN accurately captures the sentiment change and scope of negation. RNTNs also learn that sentiment of phrases following the contrastive conjunction 'but' dominates.

The complete training and testing code, a live demo and the Stanford Sentiment Treebank dataset are available at `http://nlp.stanford.edu/sentiment`.

## 2 Related Work

This work is connected to five different areas of NLP research, each with their own large amount of related work to which we cannot do full justice given space constraints.

**Semantic Vector Spaces.** The dominant approach in semantic vector spaces uses distributional similarities of single words. Often, co-occurrence statistics of a word and its context are used to describe each word (Turney and Pantel, 2010; Baroni and Lenci, 2010), such as tf-idf. Variants of this idea use more complex frequencies such as how often a word appears in a certain syntactic context (Pado and Lapata, 2007; Erk and Padó, 2008). However, distributional vectors often do not properly capture the differences in antonyms since those often have similar contexts. One possibility to remedy this is to use neural word vectors (Bengio et al., 2003). These vectors can be trained in an unsupervised fashion to capture distributional similarities (Collobert and Weston, 2008; Huang et al., 2012) but then also be fine-tuned and trained to specific tasks such as sentiment detection (Socher et al., 2011b). The models in this paper can use purely supervised word representations learned entirely on the new corpus.

**Compositionality in Vector Spaces.** Most of the compositionality algorithms and related datasets capture two word compositions. Mitchell and Lapata (2010) use e.g. two-word phrases and analyze similarities computed by vector addition, multiplication and others. Some related models such as holographic reduced representations (Plate, 1995), quantum logic (Widdows, 2008), discrete-continuous models (Clark and Pulman, 2007) and the recent compositional matrix space model (Rudolph and Giesbrecht, 2010) have not been experimentally validated on larger corpora. Yessenalina and Cardie (2011) compute matrix representations for longer phrases and define composition as matrix multiplication, and also evaluate on sentiment. Grefenstette and Sadrzadeh (2011) analyze subject-verb-object triplets and find a matrix-based categorical model to correlate well with human judgments. We compare to the recent line of work on supervised compositional models. In particular we will describe and experimentally compare our new RNTN model to recursive neural networks (RNN) (Socher et al., 2011b) and matrix-vector RNNs (Socher et al., 2012) both of which have been applied to bag of words sentiment corpora.

**Logical Form.** A related field that tackles compositionality from a very different angle is that of trying to map sentences to logical form (Zettlemoyer and Collins, 2005). While these models are highly interesting and work well in closed domains and on discrete sets, they could only capture sentiment distributions using separate mechanisms beyond the currently used logical forms.

**Deep Learning.** Apart from the above mentioned

work on RNNs, several compositionality ideas related to neural networks have been discussed by Bottou (2011) and Hinton (1990) and first models such as Recursive Auto-associative memories been experimented with by Pollack (1990). The idea to relate inputs through three way interactions, parameterized by a tensor have been proposed for relation classification (Sutskever et al., 2009; Jenatton et al., 2012), extending Restricted Boltzmann machines (Ranzato and Hinton, 2010) and as a special layer for speech recognition (Yu et al., 2012).

**Sentiment Analysis.** Apart from the abovementioned work, most approaches in sentiment analysis use bag of words representations (Pang and Lee, 2008). Snyder and Barzilay (2007) analyzed larger reviews in more detail by analyzing the sentiment of multiple aspects of restaurants, such as food or atmosphere. Several works have explored sentiment compositionality through careful engineering of features or polarity shifting rules on syntactic structures (Polanyi and Zaenen, 2006; Moilanen and Pulman, 2007; Rentoumi et al., 2010; Nakagawa et al., 2010).

## 3 Stanford Sentiment Treebank

Bag of words classifiers can work well in longer documents by relying on a few words with strong sentiment like 'awesome' or 'exhilarating.' However, sentiment accuracies even for binary positive/negative classification for single sentences has not exceeded 80% for several years. For the more difficult multiclass case including a neutral class, accuracy is often below 60% for short messages on Twitter (Wang et al., 2012). From a linguistic or cognitive standpoint, ignoring word order in the treatment of a semantic task is not plausible, and, as we will show, it cannot accurately classify hard examples of negation. Correctly predicting these hard cases is necessary to further improve performance.

In this section we will introduce and provide some analyses for the new *Sentiment Treebank* which includes labels for every syntactically plausible phrase in thousands of sentences, allowing us to train and evaluate compositional models.

We consider the corpus of movie review excerpts from the `rottentomatoes.com` website originally collected and published by Pang and Lee (2005). The original dataset includes 10,662 sentences, half of which were considered positive and the other half negative. Each label is extracted from a longer movie review and reflects the writer's overall intention for this review. The normalized, lowercased text is first used to recover, from the original website, the text with capitalization. Remaining HTML tags and sentences that are not in English are deleted. The Stanford Parser (Klein and Manning, 2003) is used to parses all 10,662 sentences. In approximately 1,100 cases it splits the snippet into multiple sentences. We then used Amazon Mechanical Turk to label the resulting 215,154 phrases. Fig. 3 shows the interface annotators saw. The slider has 25 different values and is initially set to neutral. The phrases in each hit are randomly sampled from the set of all phrases in order to prevent labels being influenced by what follows. For more details on the dataset collection, see supplementary material.

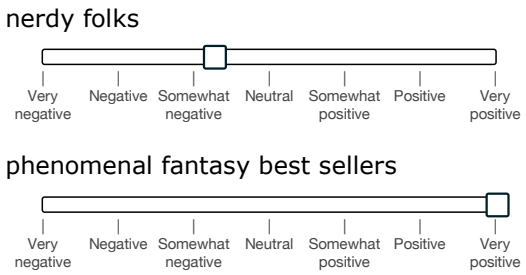



Figure 3: The labeling interface. Random phrases were shown and annotators had a slider for selecting the sentiment and its degree.

Fig. 2 shows the normalized label distributions at each $n$-gram length. Starting at length 20, the majority are full sentences. One of the findings from labeling sentences based on *reader's perception* is that many of them could be considered neutral. We also notice that stronger sentiment often builds up in longer phrases and the majority of the shorter phrases are neutral. Another observation is that most annotators moved the slider to one of the five positions: negative, somewhat negative, neutral, positive or somewhat positive. The extreme values were rarely used and the slider was not often left in between the ticks. Hence, *even a 5-class classification into these categories captures the main variability of the labels*. We will name this *fine-grained sentiment* classification and our main experiment will be to recover these five labels for phrases of all lengths.

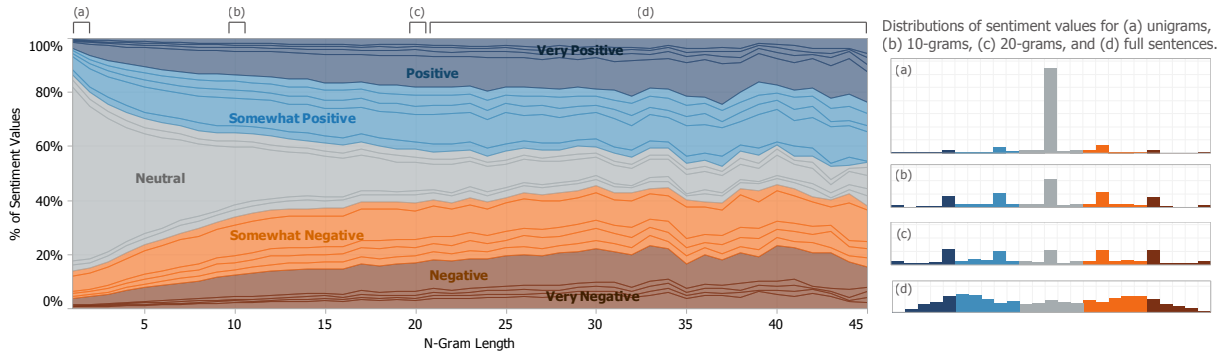



Figure 2: Normalized histogram of sentiment annotations at each $n$-gram length. Many shorter $n$-grams are neutral; longer phrases are well distributed. Few annotators used slider positions between ticks or the extreme values. Hence the two strongest labels and intermediate tick positions are merged into 5 classes.

## 4 Recursive Neural Models

The models in this section compute compositional vector representations for phrases of variable length and syntactic type. These representations will then be used as features to classify each phrase. Fig. 4 displays this approach. When an $n$-gram is given to the compositional models, it is parsed into a binary tree and each leaf node, corresponding to a word, is represented as a vector. Recursive neural models will then compute parent vectors in a bottom up fashion using different types of compositionality functions $g$. The parent vectors are again given as features to a classifier. For ease of exposition, we will use the tri-gram in this figure to explain all models.

We first describe the operations that the below recursive neural models have in common: word vector representations and classification. This is followed by descriptions of two previous RNN models and our RNTN.

Each word is represented as a $d$-dimensional vector. We initialize all word vectors by randomly sampling each value from a uniform distribution: $\mathcal{U}(-r, r)$, where $r = 0.0001$. All the word vectors are stacked in the word embedding matrix $L \in \mathbb{R}^{d \times |V|}$, where $|V|$ is the size of the vocabulary. Initially the word vectors will be random but the $L$ matrix is seen as a parameter that is trained jointly with the compositionality models.

We can use the word vectors immediately as parameters to optimize and as feature inputs to a $\mathrm{softmax}$ classifier. For classification into five classes, we compute the posterior probability over

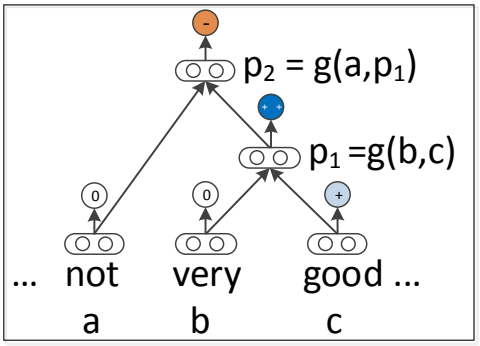



Figure 4: Approach of Recursive Neural Network models for sentiment: Compute parent vectors in a bottom up fashion using a compositionality function $g$ and use node vectors as features for a classifier at that node. This function varies for the different models.

labels given the word vector via:

$$y^a = \mathrm{softmax}(W_s a), \tag{1}$$

where $W_s \in \mathbb{R}^{5 \times d}$ is the sentiment classification matrix. For the given tri-gram, this is repeated for vectors $b$ and $c$. The main task of and difference between the models will be to compute the hidden vectors $p_i \in \mathbb{R}^d$ in a bottom up fashion.

### 4.1 RNN: Recursive Neural Network

The simplest member of this family of neural network models is the standard recursive neural network (Goller and Küchler, 1996; Socher et al., 2011a). First, it is determined which parent already has all its children computed. In the above tree example, $p_1$ has its two children's vectors since both are words. RNNs use the following equations to compute the parent vectors:

$$p_1 = f\left(W \begin{bmatrix} b \\ c \end{bmatrix}\right), p_2 = f\left(W \begin{bmatrix} a \\ p_1 \end{bmatrix}\right),$$

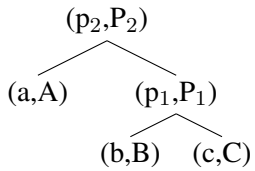
where $f = \tanh$ is a standard element-wise nonlinearity, $W \in \mathbb{R}^{d \times 2d}$ is the main parameter to learn and we omit the bias for simplicity. The bias can be added as an extra column to $W$ if an additional 1 is added to the concatenation of the input vectors. The parent vectors must be of the same dimensionality to be recursively compatible and be used as input to the next composition. Each parent vector $p_i$, is given to the same $\mathrm{softmax}$ classifier of Eq. 1 to compute its label probabilities.

This model uses the same compositionality function as the recursive autoencoder (Socher et al., 2011b) and recursive auto-associate memories (Pollack, 1990). The only difference to the former model is that we fix the tree structures and ignore the reconstruction loss. In initial experiments, we found that with the additional amount of training data, the reconstruction loss at each node is not necessary to obtain high performance.

### 4.2 MV-RNN: Matrix-Vector RNN

The MV-RNN is linguistically motivated in that most of the parameters are associated with words and each composition function that computes vectors for longer phrases depends on the actual words being combined. The main idea of the MV-RNN (Socher et al., 2012) is to represent every word and longer phrase in a parse tree as both a vector and a matrix. When two constituents are combined the matrix of one is multiplied with the vector of the other and vice versa. Hence, the compositional function is parameterized by the words that participate in it.

Each word's matrix is initialized as a $d \times d$ identity matrix, plus a small amount of Gaussian noise. Similar to the random word vectors, the parameters of these matrices will be trained to minimize the classification error at each node. For this model, each $n$-gram is represented as a list of (vector,matrix) pairs, together with the parse tree. For the tree with (vector,matrix) nodes:




the MV-RNN computes the first parent vector and its matrix via two equations:

$$p_1 = f\left(W \begin{bmatrix} Cb \\ Bc \end{bmatrix}\right), P_1 = f\left(W_M \begin{bmatrix} B \\ C \end{bmatrix}\right),$$

where $W_M \in \mathbb{R}^{d \times 2d}$ and the result is again a $d \times d$ matrix. Similarly, the second parent node is computed using the previously computed (vector,matrix) pair $(p_1, P_1)$ as well as $(a, A)$. The vectors are used for classifying each phrase using the same $\mathrm{softmax}$ classifier as in Eq. 1.

### 4.3 RNTN:Recursive Neural Tensor Network

One problem with the MV-RNN is that the number of parameters becomes very large and depends on the size of the vocabulary. It would be cognitively more plausible if there was a single powerful composition function with a fixed number of parameters. The standard RNN is a good candidate for such a function. However, in the standard RNN, the input vectors only implicitly interact through the nonlinearity (squashing) function. A more direct, possibly multiplicative, interaction would allow the model to have greater interactions between the input vectors.

Motivated by these ideas we ask the question: Can a single, more powerful composition function perform better and compose aggregate meaning from smaller constituents more accurately than many input specific ones? In order to answer this question, we propose a new model called the Recursive Neural Tensor Network (RNTN). The main idea is to use the same, tensor-based composition function for all nodes.

Fig. 5 shows a single tensor layer. We define the output of a tensor product $h \in \mathbb{R}^d$ via the following vectorized notation and the equivalent but more detailed notation for each slice $V^{[i]} \in \mathbb{R}^{d \times d}$:

$$h = \begin{bmatrix} b \\ c \end{bmatrix}^T V^{[1:d]} \begin{bmatrix} b \\ c \end{bmatrix}; h_i = \begin{bmatrix} b \\ c \end{bmatrix}^T V^{[i]} \begin{bmatrix} b \\ c \end{bmatrix}.$$

where $V^{[1:d]} \in \mathbb{R}^{2d \times 2d \times d}$ is the tensor that defines multiple bilinear forms.

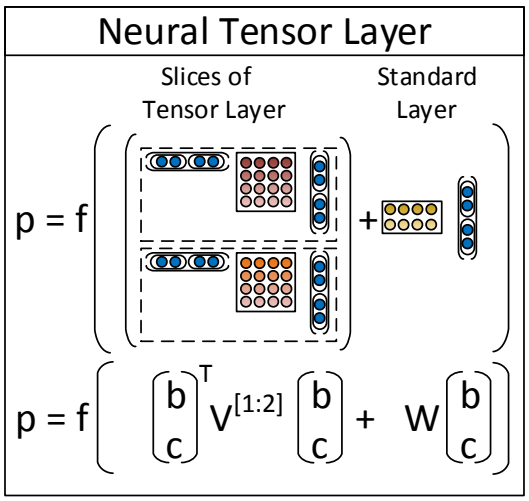



Figure 5: A single layer of the Recursive Neural Tensor Network. Each dashed box represents one of $d$-many slices and can capture a type of influence a child can have on its parent.

The RNTN uses this definition for computing $p_1$:

$$p_1 = f\left(\begin{bmatrix} b \\ c \end{bmatrix}^T V^{[1:d]} \begin{bmatrix} b \\ c \end{bmatrix} + W \begin{bmatrix} b \\ c \end{bmatrix}\right),$$

where $W$ is as defined in the previous models. The next parent vector $p_2$ in the tri-gram will be computed with the same weights:

$$p_2 = f\left(\begin{bmatrix} a \\ p_1 \end{bmatrix}^T V^{[1:d]} \begin{bmatrix} a \\ p_1 \end{bmatrix} + W \begin{bmatrix} a \\ p_1 \end{bmatrix}\right).$$

The main advantage over the previous RNN model, which is a special case of the RNTN when $V$ is set to 0, is that the tensor can directly relate input vectors. Intuitively, we can interpret each slice of the tensor as capturing a specific type of composition.

An alternative to RNTNs would be to make the compositional function more powerful by adding a second neural network layer. However, initial experiments showed that it is hard to optimize this model and vector interactions are still more implicit than in the RNTN.

### 4.4 Tensor Backprop through Structure

We describe in this section how to train the RNTN model. As mentioned above, each node has a $\mathrm{softmax}$ classifier trained on its vector representation to predict a given ground truth or target vector $t$. We assume the target distribution vector at each node has a 0-1 encoding. If there are $C$ classes, then it has length $C$ and a 1 at the correct label. All other entries are 0.

We want to maximize the probability of the correct prediction, or minimize the cross-entropy error between the predicted distribution $y^i \in \mathbb{R}^{C\times 1}$ at node $i$ and the target distribution $t^i \in \mathbb{R}^{C\times 1}$ at that node. This is equivalent (up to a constant) to minimizing the KL-divergence between the two distributions. The error as a function of the RNTN parameters $\theta = (V, W, W_s, L)$ for a sentence is:

$$E(\theta) = \sum_i \sum_j t_j^i \log y_j^i + \lambda \|\theta\|^2 \qquad (2)$$

The derivative for the weights of the $\mathrm{softmax}$ classifier are standard and simply sum up from each node's error. We define $x^i$ to be the vector at node $i$ (in the example trigram, the $x^i \in \mathbb{R}^{d\times 1}$'s are $(a, b, c, p_1, p_2)$). We skip the standard derivative for $W_s$. Each node backpropagates its error through to the recursively used weights $V, W$. Let $\delta^{i,s} \in \mathbb{R}^{d\times 1}$ be the $\mathrm{softmax}$ error vector at node $i$:

$$\delta^{i,s} = \left(W_s^T(y^i - t^i)\right) \otimes f'(x^i),$$

where $\otimes$ is the Hadamard product between the two vectors and $f'$ is the element-wise derivative of $f$ which in the standard case of using $f = \tanh$ can be computed using only $f(x^i)$.

The remaining derivatives can only be computed in a top-down fashion from the top node through the tree and into the leaf nodes. The full derivative for $V$ and $W$ is the sum of the derivatives at each of the nodes. We define the complete incoming error messages for a node $i$ as $\delta^{i,com}$. The top node, in our case $p_2$, only received errors from the top node's $\mathrm{softmax}$. Hence, $\delta^{p_2,com} = \delta^{p_2,s}$ which we can use to obtain the standard backprop derivative for $W$ (Goller and Küchler, 1996; Socher et al., 2010). For the derivative of each slice $k = 1, \ldots, d$, we get:

$$\frac{\partial E^{p_2}}{\partial V^{[k]}} = \delta_k^{p_2,com} \begin{bmatrix} a \\ p_1 \end{bmatrix} \begin{bmatrix} a \\ p_1 \end{bmatrix}^T,$$

where $\delta_k^{p_2,com}$ is just the $k$'th element of this vector. Now, we can compute the error message for the two

children of $p_2$:

$$\delta^{p_2,down} = \left(W^T\delta^{p_2,com} + S\right) \otimes f'\left(\begin{bmatrix} a \\ p_1 \end{bmatrix}\right),$$

where we define

$$S = \sum_{k=1}^{d} \delta_k^{p_2,com} \left(V^{[k]} + \left(V^{[k]}\right)^T\right) \begin{bmatrix} a \\ p_1 \end{bmatrix}$$

The children of $p_2$, will then each take half of this vector and add their own $\mathrm{softmax}$ error message for the complete $\delta$. In particular, we have

$$\delta^{p_1,com} = \delta^{p_1,s} + \delta^{p_2,down}[d+1:2d],$$

where $\delta^{p_2,down}[d+1:2d]$ indicates that $p_1$ is the right child of $p_2$ and hence takes the 2nd half of the error, for the final word vector derivative for $a$, it will be $\delta^{p_2,down}[1:d]$.

The full derivative for slice $V^{[k]}$ for this trigram tree then is the sum at each node:

$$\frac{\partial E}{\partial V^{[k]}} = \frac{E^{p_2}}{\partial V^{[k]}} + \delta_k^{p_1,com} \begin{bmatrix} b \\ c \end{bmatrix} \begin{bmatrix} b \\ c \end{bmatrix}^T,$$

and similarly for $W$. For this nonconvex optimization we use AdaGrad (Duchi et al., 2011) which converges in less than 3 hours to a local optimum.

## 5 Experiments

We include two types of analyses. The first type includes several large quantitative evaluations on the test set. The second type focuses on two linguistic phenomena that are important in sentiment.

For all models, we use the dev set and cross-validate over regularization of the weights, word vector size as well as learning rate and minibatch size for AdaGrad. Optimal performance for all models was achieved at word vector sizes between 25 and 35 dimensions and batch sizes between 20 and 30. Performance decreased at larger or smaller vector and batch sizes. This indicates that the RNTN does not outperform the standard RNN due to simply having more parameters. The MV-RNN has orders of magnitudes more parameters than any other model due to the word matrices. The RNTN would usually achieve its best performance on the dev set after training for 3 - 5 hours. Initial experiments showed that the recursive models worked significantly worse (over 5% drop in accuracy) when no nonlinearity was used. We use $f = \tanh$ in all experiments.

| Model | Fine-grained | | Positive/Negative | |
|---|---|---|---|---|
| | All | Root | All | Root |
| NB | 67.2 | 41.0 | 82.6 | 81.8 |
| SVM | 64.3 | 40.7 | 84.6 | 79.4 |
| BiNB | 71.0 | 41.9 | 82.7 | 83.1 |
| VecAvg | 73.3 | 32.7 | 85.1 | 80.1 |
| RNN | 79.0 | 43.2 | 86.1 | 82.4 |
| MV-RNN | 78.7 | 44.4 | 86.8 | 82.9 |
| RNTN | **80.7** | **45.7** | **87.6** | **85.4** |

Table 1: Accuracy for fine grained (5-class) and binary predictions at the sentence level (root) and for all nodes.

We compare to commonly used methods that use bag of words features with Naive Bayes and SVMs, as well as Naive Bayes with bag of bigram features. We abbreviate these with NB, SVM and biNB. We also compare to a model that averages neural word vectors and ignores word order (VecAvg).

The sentences in the treebank were split into a train (8544), dev (1101) and test splits (2210) and these splits are made available with the data release. We also analyze performance on only positive and negative sentences, ignoring the neutral class. This filters about 20% of the data with the three sets having 6920/872/1821 sentences.

### 5.1 Fine-grained Sentiment For All Phrases

The main novel experiment and evaluation metric analyze the accuracy of fine-grained sentiment classification for all phrases. Fig. 2 showed that a fine grained classification into 5 classes is a reasonable approximation to capture most of the data variation.

Fig. 6 shows the result on this new corpus. The RNTN gets the highest performance, followed by the MV-RNN and RNN. The recursive models work very well on shorter phrases, where negation and composition are important, while bag of features baselines perform well only with longer sentences. The RNTN accuracy upper bounds other models at most $n$-gram lengths.

Table 1 (left) shows the overall accuracy numbers for fine grained prediction at all phrase lengths and full sentences.

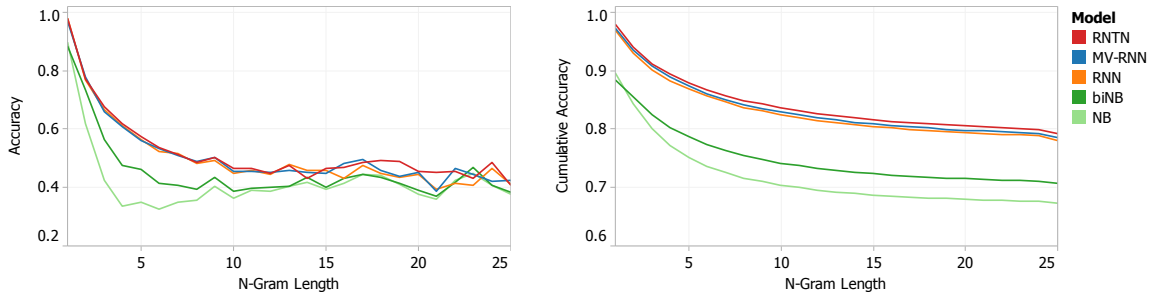



Figure 6: Accuracy curves for fine grained sentiment classification at each $n$-gram lengths. Left: Accuracy separately for each set of $n$-grams. Right: Cumulative accuracy of all $\leq n$-grams.

### 5.2 Full Sentence Binary Sentiment

This setup is comparable to previous work on the original rotten tomatoes dataset which only used full sentence labels and binary classification of positive/negative. Hence, these experiments show the improvement even baseline methods can achieve with the sentiment treebank. Table 1 shows results of this binary classification for both all phrases and for only full sentences. The previous state of the art was below 80% (Socher et al., 2012). With the coarse bag of words annotation for training, many of the more complex phenomena could not be captured, even by more powerful models. The combination of the new sentiment treebank and the RNTN pushes the state of the art on short phrases up to 85.4%.

### 5.3 Model Analysis: Contrastive Conjunction

In this section, we use a subset of the test set which includes only sentences with an '$X$ *but* $Y$' structure: A phrase $X$ being followed by *but* which is followed by a phrase $Y$. The conjunction is interpreted as an argument for the second conjunct, with the first functioning concessively (Lakoff, 1971; Blakemore, 1989; Merin, 1999). Fig. 7 contains an example. We analyze a strict setting, where $X$ and $Y$ are phrases of different sentiment (including neutral). The example is counted as correct, if the classifications for both phrases $X$ and $Y$ are correct. Furthermore, the lowest node that dominates both of the word *but* and the node that spans $Y$ also have to have the same correct sentiment. For the resulting 131 cases, the RNTN obtains an accuracy of 41% compared to MV-RNN (37), RNN (36) and biNB (27).

### 5.4 Model Analysis: High Level Negation

We investigate two types of negation. For each type, we use a separate dataset for evaluation.

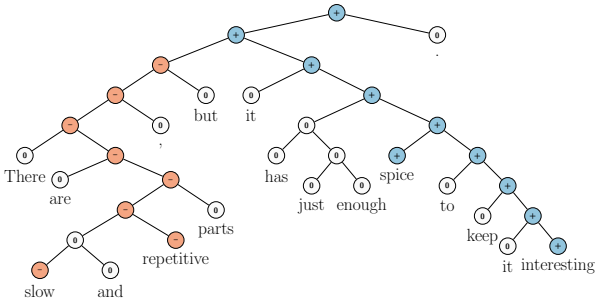



Figure 7: Example of correct prediction for contrastive conjunction $X$ *but* $Y$.

**Set 1: Negating Positive Sentences.** The first set contains positive sentences and their negation. In this set, the negation changes the overall sentiment of a sentence from positive to negative. Hence, we compute accuracy in terms of correct sentiment reversal from positive to negative. Fig. 9 shows two examples of positive negation the RNTN correctly classified, even if negation is less obvious in the case of 'least'. Table 2 (left) gives the accuracies over 21 positive sentences and their negation for all models. The RNTN has the highest reversal accuracy, showing its ability to structurally learn negation of positive sentences. But what if the model simply makes phrases very negative when negation is in the sentence? The next experiments show that the model captures more than such a simplistic negation rule.

**Set 2: Negating Negative Sentences.** The second set contains negative sentences and their negation. When negative sentences are negated, the sentiment treebank shows that overall sentiment should become *less negative*, but not necessarily positive. For instance, 'The movie was terrible' is negative but the 'The movie was not terrible' says only that it was less bad than a terrible one, not that it was good (Horn, 1989; Israel, 2001). Hence, we evaluate ac-

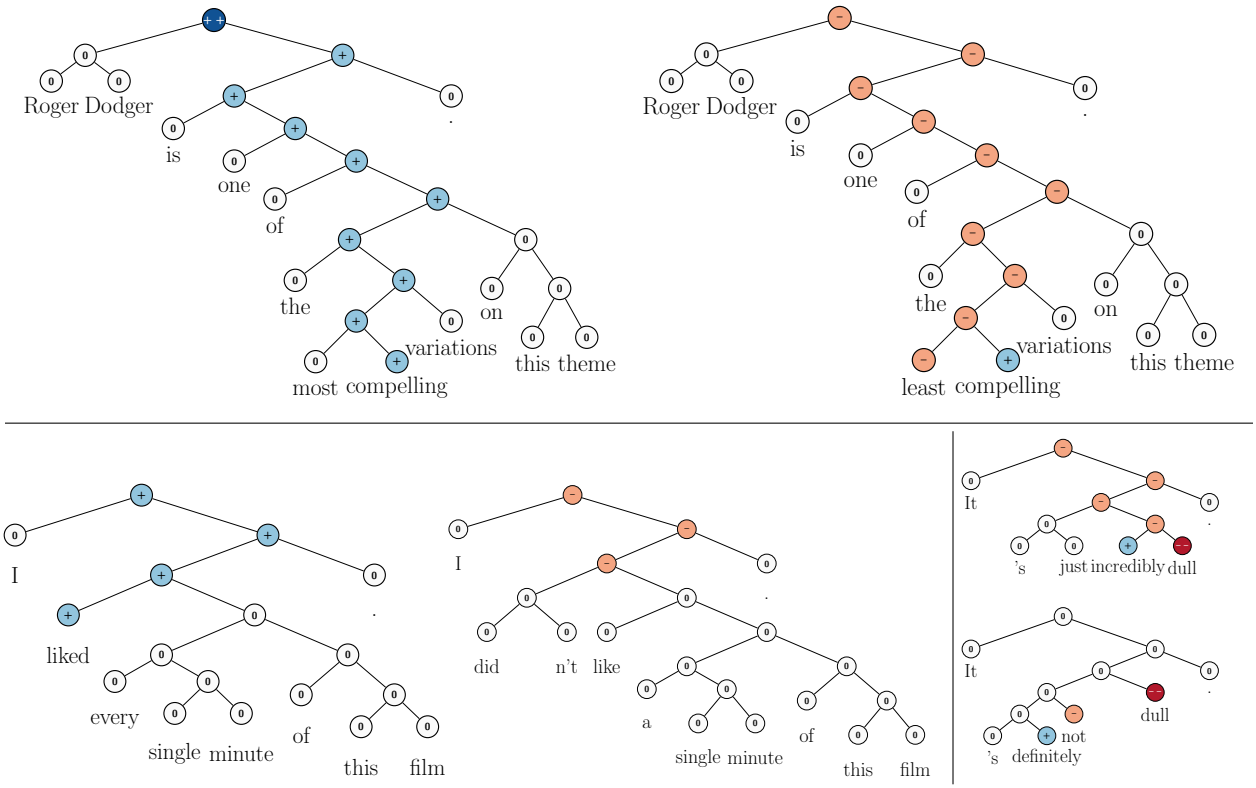



Figure 9: RNTN prediction of positive and negative (bottom right) sentences and their negation.

| Model | Accuracy | |
|---|---|---|
| | Negated Positive | Negated Negative |
| biNB | 19.0 | 27.3 |
| RNN | 33.3 | 45.5 |
| MV-RNN | 52.4 | 54.6 |
| RNTN | **71.4** | **81.8** |

Table 2: Accuracy of negation detection. Negated positive is measured as correct sentiment inversions. Negated negative is measured as increases in positive activations.

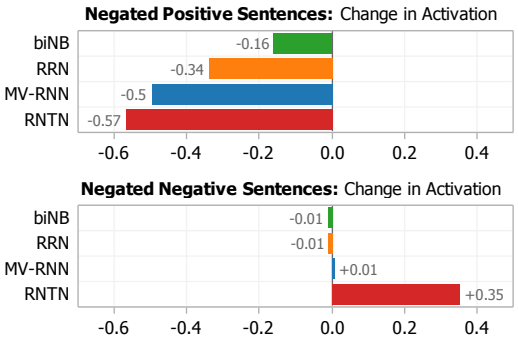



Figure 8: Change in activations for negations. Only the RNTN correctly captures both types. It decreases positive sentiment more when it is negated and learns that negating negative phrases (such as *not terrible*) should increase neutral and positive activations.

curacy in terms of how often each model was able to increase non-negative activation in the sentiment of the sentence. Table 2 (right) shows the accuracy. In over 81% of cases, the RNTN correctly increases the positive activations. Fig. 9 (bottom right) shows a typical case in which sentiment was made more positive by switching the main class from negative to neutral even though both *not* and *dull* were negative. Fig. 8 shows the changes in activation for both sets. Negative values indicate a decrease in average positive activation (for set 1) and positive values mean an increase in average positive activation (set 2). The RNTN has the largest shifts in the correct directions. Therefore we can conclude that the RNTN is best able to identify the effect of negations upon both positive and negative sentiment sentences.

| $n$ | Most positive $n$-grams | Most negative $n$-grams |
|---|---|---|
| 1 | engaging; best; powerful; love; beautiful | bad; dull; boring; fails; worst; stupid; painfully |
| 2 | excellent performances; A masterpiece; masterful film; wonderful movie; marvelous performances | worst movie; very bad; shapeless mess; worst thing; instantly forgettable; complete failure |
| 3 | an amazing performance; wonderful all-ages triumph; a wonderful movie; most visually stunning | for worst movie; A lousy movie; a complete failure; most painfully marginal; very bad sign |
| 5 | nicely acted and beautifully shot; gorgeous imagery, effective performances; the best of the year; a terrific American sports movie; refreshingly honest and ultimately touching | silliest and most incoherent movie; completely crass and forgettable movie; just another bad movie. A cumbersome and cliche-ridden movie; a humorless, disjointed mess |
| 8 | one of the best films of the year; A love for films shines through each frame; created a masterful piece of artistry right here; A masterful film from a master filmmaker, | A trashy, exploitative, thoroughly unpleasant experience ; this sloppy drama is an empty vessel.; quickly drags on becoming boring and predictable.; be the worst special-effects creation of the year |

Table 3: Examples of $n$-grams for which the RNTN predicted the most positive and most negative responses.

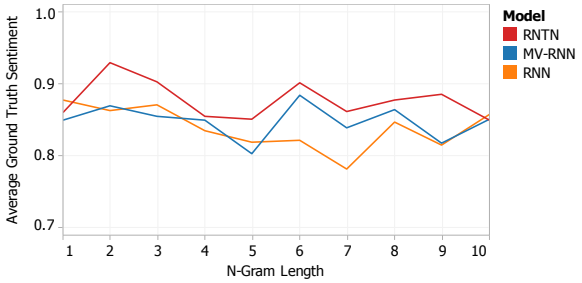



Figure 10: Average ground truth sentiment of top 10 most positive $n$-grams at various $n$. The RNTN correctly picks the more negative and positive examples.

### 5.5 Model Analysis: Most Positive and Negative Phrases

We queried the model for its predictions on what the most positive or negative $n$-grams are, measured as the highest activation of the most negative and most positive classes. Table 3 shows some phrases from the dev set which the RNTN selected for their strongest sentiment.

Due to lack of space we cannot compare top phrases of the other models but Fig. 10 shows that the RNTN selects more strongly positive phrases at most $n$-gram lengths compared to other models.

For this and the previous experiment, please find additional examples and descriptions in the supplementary material.

## 6 Conclusion

We introduced Recursive Neural Tensor Networks and the Stanford Sentiment Treebank. The combination of new model and data results in a system for single sentence sentiment detection that pushes state of the art by 5.4% for positive/negative sentence classification. Apart from this standard setting, the dataset also poses important new challenges and allows for new evaluation metrics. For instance, the RNTN obtains 80.7% accuracy on fine-grained sentiment prediction across all phrases and captures negation of different sentiments and scope more accurately than previous models.


### Acknowledgments

We thank Rukmani Ravisundaram and Tayyab Tariq for the first version of the online demo. Richard is partly supported by a Microsoft Research PhD fellowship. The authors gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) Deep Exploration and Filtering of Text (DEFT) Program under Air Force Research Laboratory (AFRL) prime contract no. FA8750-13-2-0040, the DARPA Deep Learning program under contract number FA8650-10-C-7020 and NSF IIS-1159679. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of DARPA, AFRL, or the US government.

## References


M. Baroni and A. Lenci. 2010. Distributional memory: A general framework for corpus-based semantics. *Computational Linguistics*, 36(4):673–721.

Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. 2003. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3, March.

D. Blakemore. 1989. Denial and contrast: A relevance theoretic analysis of 'but'. *Linguistics and Philosophy*, 12:15–37.

L. Bottou. 2011. From machine learning to machine reasoning. *CoRR*, abs/1102.1808.

S. Clark and S. Pulman. 2007. Combining symbolic and distributional models of meaning. In *Proceedings of the AAAI Spring Symposium on Quantum Interaction*, pages 52–55.

R. Collobert and J. Weston. 2008. A unified architecture for natural language processing: deep neural networks with multitask learning. In *ICML*.

J. Duchi, E. Hazan, and Y. Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12, July.

K. Erk and S. Padó. 2008. A structured vector space model for word meaning in context. In *EMNLP*.

C. Goller and A. Küchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of the International Conference on Neural Networks (ICNN-96)*.

E. Grefenstette and M. Sadrzadeh. 2011. Experimental support for a categorical compositional distributional model of meaning. In *EMNLP*.

E. Grefenstette, G. Dinu, Y.-Z. Zhang, M. Sadrzadeh, and M. Baroni. 2013. Multi-step regression learning for compositional distributional semantics. In *IWCS*.

G. E. Hinton. 1990. Mapping part-whole hierarchies into connectionist networks. *Artificial Intelligence*, 46(1-2).

L. R. Horn. 1989. *A natural history of negation*, volume 960. University of Chicago Press Chicago.

E. H. Huang, R. Socher, C. D. Manning, and A. Y. Ng. 2012. Improving Word Representations via Global Context and Multiple Word Prototypes. In *ACL*.

M. Israel. 2001. Minimizers, maximizers, and the rhetoric of scalar reasoning. *Journal of Semantics*, 18(4):297–331.

R. Jenatton, N. Le Roux, A. Bordes, and G. Obozinski. 2012. A latent factor model for highly multi-relational data. In *NIPS*.

D. Klein and C. D. Manning. 2003. Accurate unlexicalized parsing. In *ACL*.

R. Lakoff. 1971. If's, and's, and but's about conjunction. In Charles J. Fillmore and D. Terence Langendoen, editors, *Studies in Linguistic Semantics*, pages 114–149. Holt, Rinehart, and Winston, New York.

A. Merin. 1999. Information, relevance, and social decisionmaking: Some principles and results of decision-theoretic semantics. In Lawrence S. Moss, Jonathan Ginzburg, and Maarten de Rijke, editors, *Logic, Language, and Information*, volume 2. CSLI, Stanford, CA.

J. Mitchell and M. Lapata. 2010. Composition in distributional models of semantics. *Cognitive Science*, 34(8):1388–1429.

K. Moilanen and S. Pulman. 2007. Sentiment composition. In *In Proceedings of Recent Advances in Natural Language Processing*.

T. Nakagawa, K. Inui, and S. Kurohashi. 2010. Dependency tree-based sentiment classification using CRFs with hidden variables. In *NAACL*, HLT.

S. Pado and M. Lapata. 2007. Dependency-based construction of semantic space models. *Computational Linguistics*, 33(2):161–199.

B. Pang and L. Lee. 2005. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *ACL*, pages 115–124.

B. Pang and L. Lee. 2008. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135.

T. A. Plate. 1995. Holographic reduced representations. *IEEE Transactions on Neural Networks*, 6(3):623–641.

L. Polanyi and A. Zaenen. 2006. Contextual valence shifters. In W. Bruce Croft, James Shanahan, Yan Qu, and Janyce Wiebe, editors, *Computing Attitude and Affect in Text: Theory and Applications*, volume 20 of *The Information Retrieval Series*, chapter 1.

J. B. Pollack. 1990. Recursive distributed representations. *Artificial Intelligence*, 46, November.

M. Ranzato and A. Krizhevsky G. E. Hinton. 2010. Factored 3-Way Restricted Boltzmann Machines For Modeling Natural Images. *AISTATS*.

V. Rentoumi, S. Petrakis, M. Klenner, G. A. Vouros, and V. Karkaletsis. 2010. United we stand: Improving sentiment analysis by joining machine learning and rule based methods. In *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta.

S. Rudolph and E. Giesbrecht. 2010. Compositional matrix-space models of language. In *ACL*.

B. Snyder and R. Barzilay. 2007. Multiple aspect ranking using the Good Grief algorithm. In *HLT-NAACL*.

R. Socher, C. D. Manning, and A. Y. Ng. 2010. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*.

R. Socher, C. Lin, A. Y. Ng, and C.D. Manning. 2011a. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *ICML*.

R. Socher, J. Pennington, E. H. Huang, A. Y. Ng, and C. D. Manning. 2011b. Semi-Supervised Recursive Autoencoders for Predicting Sentiment Distributions. In *EMNLP*.

R. Socher, B. Huval, C. D. Manning, and A. Y. Ng. 2012. Semantic compositionality through recursive matrix-vector spaces. In *EMNLP*.

I. Sutskever, R. Salakhutdinov, and J. B. Tenenbaum. 2009. Modelling relational data using Bayesian clustered tensor factorization. In *NIPS*.

P. D. Turney and P. Pantel. 2010. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37:141–188.

H. Wang, D. Can, A. Kazemzadeh, F. Bar, and S. Narayanan. 2012. A system for real-time twitter sentiment analysis of 2012 u.s. presidential election cycle. In *Proceedings of the ACL 2012 System Demonstrations*.

D. Widdows. 2008. Semantic vector products: Some initial investigations. In *Proceedings of the Second AAAI Symposium on Quantum Interaction*.

A. Yessenalina and C. Cardie. 2011. Compositional matrix-space models for sentiment analysis. In *EMNLP*.

D. Yu, L. Deng, and F. Seide. 2012. Large vocabulary speech recognition using deep tensor neural networks. In *INTERSPEECH*.

F.M. Zanzotto, I. Korkontzelos, F. Fallucchi, and S. Manandhar. 2010. Estimating linear models for compositional distributional semantics. In *COLING*.

L. Zettlemoyer and M. Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*.

# A New Approach to Linear Filtering and Prediction Problems[1]

**R. E. KALMAN**
**Research Institute for Advanced Study,[2] Baltimore, Md.**

*The classical filtering and prediction problem is re-examined using the Bode-Shannon representation of random processes and the "state transition" method of analysis of dynamic systems. New results are:*

*(1) The formulation and methods of solution of the problem apply without modification to stationary and nonstationary statistics and to growing-memory and infinite-memory filters.*

*(2) A nonlinear difference (or differential) equation is derived for the covariance matrix of the optimal estimation error. From the solution of this equation the coefficients of the difference (or differential) equation of the optimal linear filter are obtained without further calculations.*

*(3) The filtering problem is shown to be the dual of the noise-free regulator problem.*

*The new method developed here is applied to two well-known problems, confirming and extending earlier results.*

*The discussion is largely self-contained and proceeds from first principles; basic concepts of the theory of random processes are reviewed in the Appendix.*

## Introduction

AN IMPORTANT class of theoretical and practical problems in communication and control is of a statistical nature. Such problems are: (i) Prediction of random signals; (ii) separation of random signals from random noise; (iii) detection of signals of known form (pulses, sinusoids) in the presence of random noise.

In his pioneering work, Wiener [1][3] showed that problems (i) and (ii) lead to the so-called Wiener-Hopf integral equation; he also gave a method (spectral factorization) for the solution of this integral equation in the practically important special case of stationary statistics and rational spectra.

Many extensions and generalizations followed Wiener's basic work. Zadeh and Ragazzini solved the finite-memory case [2]. Concurrently and independently of Bode and Shannon [3], they also gave a simplified method [2] of solution. Booton discussed the nonstationary Wiener-Hopf equation [4]. These results are now in standard texts [5-6]. A somewhat different approach along these main lines has been given recently by Darlington [7]. For extensions to sampled signals, see, e.g., Franklin [8], Lees [9]. Another approach based on the eigenfunctions of the Wiener-Hopf equation (which applies also to nonstationary problems whereas the preceding methods in general don't), has been pioneered by Davis [10] and applied by many others, e.g., Shinbrot [11], Blum [12], Pugachev [13], Solodovnikov [14].

In all these works, the objective is to obtain the specification of a linear dynamic system (Wiener filter) which accomplishes the prediction, separation, or detection of a random signal.[4]

Present methods for solving the Wiener problem are subject to a number of limitations which seriously curtail their practical usefulness:

(1) The optimal filter is specified by its impulse response. It is not a simple task to synthesize the filter from such data.

(2) Numerical determination of the optimal impulse response is often quite involved and poorly suited to machine computation. The situation gets rapidly worse with increasing complexity of the problem.

(3) Important generalizations (e.g., growing-memory filters, nonstationary prediction) require new derivations, frequently of considerable difficulty to the nonspecialist.

(4) The mathematics of the derivations are not transparent. Fundamental assumptions and their consequences tend to be obscured.

This paper introduces a new look at this whole assemblage of problems, sidestepping the difficulties just mentioned. The following are the highlights of the paper:

(5) *Optimal Estimates and Orthogonal Projections.* The Wiener problem is approached from the point of view of conditional distributions and expectations. In this way, basic facts of the Wiener theory are quickly obtained; the scope of the results and the fundamental assumptions appear clearly. It is seen that all statistical calculations and results are based on first and second order averages; no other statistical data are needed. Thus difficulty (4) is eliminated. This method is well known in probability theory (see pp. 75–78 and 148–155 of Doob [15] and pp. 455–464 of Loève [16]) but has not yet been used extensively in engineering.

(6) *Models for Random Processes.* Following, in particular, Bode and Shannon [3], arbitrary random signals are represented (up to second order average statistical properties) as the output of a linear dynamic system excited by independent or uncorrelated random signals ("white noise"). This is a standard trick in the engineering applications of the Wiener theory [2–7]. The approach taken here differs from the conventional one only in the way in which linear dynamic systems are described. We shall emphasize the concepts of *state* and *state transition;* in other words, linear systems will be specified by systems of first-order difference (or differential) equations. This point of view is

[1] This research was supported in part by the U. S. Air Force Office of Scientific Research under Contract AF 49 (638)-382.

[2] 7212 Bellona Ave.

[3] Numbers in brackets designate References at end of paper.

[4] Of course, in general these tasks may be done better by nonlinear filters. At present, however, little or nothing is known about how to obtain (both theoretically and practically) these nonlinear filters.

Contributed by the Instruments and Regulators Division and presented at the Instruments and Regulators Conference, March 29– Apri1 2, 1959, of THE AMERICAN SOCIETY OF MECHANICAL ENGINEERS.

NOTE: Statements and opinions advanced in papers are to be understood as individual expressions of their authors and not those of the Society. Manuscript received at ASME Headquarters, February 24, 1959. Paper No. 59—IRD-11.

natural and also necessary in order to take advantage of the simplifications mentioned under (5).

(7) *Solution of the Wiener Problem.* With the state-transition method, a single derivation covers a large variety of problems: growing and infinite memory filters, stationary and nonstationary statistics, etc.; difficulty (3) disappears. Having guessed the "state" of the estimation (i.e., filtering or prediction) problem correctly, one is led to a nonlinear difference (or differential) equation for the covariance matrix of the optimal estimation error. This is vaguely analogous to the Wiener-Hopf equation. Solution of the equation for the covariance matrix starts at the time $t_0$ when the first observation is taken; at each later time $t$ the solution of the equation represents the covariance of the optimal prediction error given observations in the interval $(t_0, t)$. From the covariance matrix at time $t$ we obtain at once, without further calculations, the coefficients (in general, time-varying) characterizing the optimal linear filter.

(8) *The Dual Problem.* The new formulation of the Wiener problem brings it into contact with the growing new theory of control systems based on the "state" point of view [17–24]. It turns out, *surprisingly*, that the Wiener problem is the *dual* of the noise-free optimal regulator problem, which has been solved previously by the author, using the state-transition method to great advantage [18, 23, 24]. The mathematical background of the two problems is identical—this has been suspected all along, but until now the analogies have never been made explicit.

(9) *Applications.* The power of the new method is most apparent in theoretical investigations and in numerical answers to complex practical problems. In the latter case, it is best to resort to machine computation. Examples of this type will be discussed later. To provide some feel for applications, two standard examples from nonstationary prediction are included; in these cases the solution of the nonlinear difference equation mentioned under (7) above can be obtained even in closed form.

For easy reference, the main results are displayed in the form of theorems. Only Theorems 3 and 4 are original. The next section and the Appendix serve mainly to review well-known material in a form suitable for the present purposes.

## Notation Conventions

Throughout the paper, we shall deal mainly with *discrete* (or *sampled)* dynamic systems; in other words, signals will be observed at equally spaced points in time *(sampling instants).* By suitable choice of the time scale, the constant intervals between successive sampling instants *(sampling periods)* may be chosen as unity. Thus variables referring to time, such as $t$, $t_0$, $\tau$, $T$ will always be integers. The restriction to discrete dynamic systems is not at all essential (at least from the engineering point of view); by using the discreteness, however, we can keep the mathematics rigorous and yet elementary. Vectors will be denoted by small bold-face letters: **a**, **b**, ..., **u**, **x**, **y**, ... A *vector* or more precisely an *n-vector* is a set of $n$ numbers $x_1$, ... $x_n$; the $x_i$ are the *co-ordinates* or *components* of the vector **x**.

Matrices will be denoted by capital bold-face letters: **A**, **B**, **Q**, **Φ**, **Ψ**, …; they are $m \times n$ arrays of elements $a_{ij}$, $b_{ij}$, $q_{ij}$,... The *transpose* (interchanging rows and columns) of a matrix will be denoted by the prime. In manipulating formulas, it will be convenient to regard a vector as a matrix with a single column.

Using the conventional definition of matrix multiplication, we write the *scalar product* of two $n$-vectors **x**, **y** as

$$\mathbf{x'y} = \sum_{i=1}^{n} x_i y_i = \mathbf{y'x}$$

The scalar product is clearly a scalar, i.e., not a vector, quantity. Similarly, the quadratic form associated with the $n \times n$ matrix **Q** is,

$$\mathbf{x'Qx} = \sum_{i,j=1}^{n} x_i q_{ij} x_j$$

We define the expression **xy'** where **x'** is an $m$-vector and **y** is an $n$-vector to be the $m \times n$ matrix with elements $x_i y_j$.

We write $E(\mathbf{x}) = E\mathbf{x}$ for the expected value of the random vector **x** (see Appendix). It is usually convenient to omit the brackets after $E$. This does not result in confusion in simple cases since constants and the operator $E$ commute. Thus $E\mathbf{xy'}$ = matrix with elements $E(x_i y_j)$; $E\mathbf{x}E\mathbf{y'}$ = matrix with elements $E(x_i)E(y_j)$.

For ease of reference, a list of the principal symbols used is given below.

**Optimal Estimates**

| | |
|---|---|
| $t$ | time in general, present time. |
| $t_0$ | time at which observations start. |
| $x_1(t)$, $x_2(t)$ | basic random variables. |
| $y(t)$ | observed random variable. |
| $x_1^*(t_1\|t)$ | optimal estimate of $x_1(t_1)$ given $y(t_0)$, …, $y(t)$. |
| $L$ | loss function (non random function of its argument). |
| $\varepsilon$ | estimation error (random variable). |

**Orthogonal Projections**

| | |
|---|---|
| $\mathcal{Y}(t)$ | linear manifold generated by the random variables $y(t_0)$, …, $y(t)$. |
| $\bar{x}(t_1\|t)$ | orthogonal projection of $x(t_1)$ on $\mathcal{Y}(t)$. |
| $\tilde{x}(t_1\|t)$ | component of $x(t_1)$ orthogonal to $\mathcal{Y}(t)$. |

**Models for Random Processes**

| | |
|---|---|
| $\mathbf{\Phi}(t+1; t)$ | transition matrix |
| $\mathbf{Q}(t)$ | covariance of random excitation |

**Solution of the Wiener Problem**

| | |
|---|---|
| $\mathbf{x}(t)$ | basic random variable. |
| $\mathbf{y}(t)$ | observed random variable. |
| $\mathcal{Y}(t)$ | linear manifold generated by $\mathbf{y}(t_0)$, …, $\mathbf{y}(t)$. |
| $\mathcal{Z}(t)$ | linear manifold generated by $\mathbf{y}(t\|t-1)$. |
| $\mathbf{x}^*(t_1\|t)$ | optimal estimate of $\mathbf{x}(t_1)$ given $\mathcal{Y}(t)$. |
| $\tilde{\mathbf{x}}(t_1\|t)$ | error in optimal estimate of $\mathbf{x}(t_1)$ given $\mathcal{Y}(t)$. |

## Optimal Estimates

To have a concrete description or the type of problems to be studied, consider the following situation. We are given signal $x_1(t)$ and noise $x_2(t)$. Only the sum $y(t) = x_1(t) + x_2(t)$ can be observed. Suppose we have observed and know exactly the values of $y(t_0)$, ..., $y(t)$. What can we infer from this knowledge in regard to the (unobservable) value of the signal at $t = t_1$, where $t_1$ may be less than, equal to, or greater than $t$? If $t_1 < t$, this is a *data-smoothing (interpolation)* problem. If $t_1 = t$, this is called *filtering.* If $t_1 > t$, we have a *prediction* problem. Since our treatment will be general enough to include these and similar problems, we shall use hereafter the collective term *estimation.*

As was pointed out by Wiener [1], the natural setting of the estimation problem belongs to the realm of probability theory and statistics. Thus signal, noise, and their sum will be random variables, and consequently they may be regarded as random processes. From the probabilistic description of the random processes we can determine the probability with which a particular sample of the signal and noise will occur. For any given set of measured values $\eta(t_0)$, ..., $\eta(t)$ of the random variable $y(t)$ one can then also determine, in principle, the probability of simultaneous occurrence of various values $\xi_1(t)$ of the random variable $x_1(t_1)$. This is the conditional probability distribution function

Transactions of the ASME–Journal of Basic Engineering, 82 (Series D): 35-45. Copyright © 1960 by ASME

$$Pr[x_1(t_1) \le \xi_1 | y(t_0) = \eta(t_0), \ldots, y(t) = \eta(t)] = F(\xi_1) \quad (1)$$

Evidently, $F(\xi_1)$ represents all the information which the measurement of the random variables $y(t_0)$, ..., $y(t)$ has conveyed about the random variable $x_1(t_1)$. Any statistical estimate of the random variable $x_1(t_1)$ will be some function of this distribution and therefore a (nonrandom) function of the random variables $y(t_0)$, ..., $y(t)$. This statistical estimate is denoted by $X_1(t_1|t)$, or by just $X_1(t_1)$ or $X_1$ when the set of observed random variables or the time at which the estimate is required are clear from context.

Suppose now that $X_1$ is given as a fixed function of the random variables $y(t_0)$, ..., $y(t)$. Then $X_1$ is itself a random variable and its actual value is known whenever the actual values of $y(t_0)$, ..., $y(t)$ are known. In general, the actual value of $X_1(t_1)$ will be different from the (unknown) actual value of $x_1(t_1)$. To arrive at a rational way of determining $X_1$, it is natural to assign a *penalty* or *loss* for incorrect estimates. Clearly, the loss should be a (i) positive, (ii) nondecreasing function of the *estimation error* $\varepsilon = x_1(t_1) - X_1(t_1)$. Thus we define a *loss function* by

$$L(0) = 0$$

$$L(\varepsilon_2) \ge L(\varepsilon_1) \ge 0 \quad \text{when} \quad \varepsilon_2 \ge \varepsilon_1 \ge 0 \quad (2)$$

$$L(\varepsilon) = L(-\varepsilon)$$

Some common examples of loss functions are: $L(\varepsilon) = a\varepsilon^2$, $a\varepsilon^4$, $a|\varepsilon|$, $a[1 - \exp(-\varepsilon^2)]$, etc., where $a$ is a positive constant.

One (but by no means the only) natural way of choosing the random variable $X_1$ is to require that this choice should minimize the average loss or risk

$$E\{L[x_1(t_1) - X_1(t_1)]\} = E[E\{L[x(t_1) - X_1(t_1)]|y(t_0), \ldots, y(t)\}] \quad (3)$$

Since the first expectation on the right-hand side of (3) does not depend on the choice of $X_1$ but only on $y(t_0)$, ..., $y(t)$, it is clear that minimizing (3) is equivalent to minimizing

$$E\{L[x_1(t_1) - X_1(t_1)]|y(t_0), ..., y(t)\} \quad (4)$$

Under just slight additional assumptions, optimal estimates can be characterized in a simple way.

**Theorem 1.** *Assume that L is of type* (2) *and that the conditional distribution function* $F(\xi)$ *defined by* (1) *is:*

(*A*) *symmetric about the mean* $\bar{\xi}$ :

$$F(\xi - \bar{\xi}) = 1 - F(\bar{\xi} - \xi)$$

(*B*) *convex for* $\xi \le \bar{\xi}$ :

$$F(\lambda\xi_1 + (1 - \lambda)\xi_2) \le \lambda F(\xi_1) + (1 - \lambda)F(\xi_2)$$

*for all* $\xi_1, \xi_2 \le \bar{\xi}$ *and* $0 \le \lambda \le 1$

*Then the random variable* $x_1^*(t_1|t)$ *which minimizes the average loss* (3) *is the conditional expectation*

$$x_1^*(t_1|t) = E[x_1(t_1)|y(t_0), \ldots, y(t)] \quad (5)$$

**Proof:** As pointed out recently by Sherman [25], this theorem follows immediately from a well-known lemma in probability theory.

**Corollary.** *If the random processes* $\{x_1(t)\}$, $\{x_2(t)\}$, and $\{y(t)\}$ *are gaussian, Theorem 1 holds.*

**Proof:** By Theorem 5, (*A*) (see Appendix), conditional distributions on a gaussian random process are gaussian. Hence the requirements of Theorem 1 are always satisfied.

In the control system literature, this theorem appears sometimes in a form which is more restrictive in one way and more general in another way:

**Theorem I-a.** *If* $L(\varepsilon) = \varepsilon^2$, *then Theorem 1 is true without assumptions* (*A*) and (*B*).

**Proof:** Expand the conditional expectation (4):

$$E[x_1^2(t_1)|y(t_0), \ldots, y(t)] - 2X_1(t_1)E[x_1(t_1)|y(t_0), \ldots, y(t)] + X_1^2(t_1)$$

and differentiate with respect to $X_1(t_1)$. This is not a completely rigorous argument; for a simple rigorous proof see Doob [15], pp. 77–78.

**Remarks.** (*a*) As far as the author is aware, it is not known what is the most general class of random processes $\{x_1(t)\}$, $\{x_2(t)\}$ for which the conditional distribution function satisfies the requirements of Theorem 1.

(*b*) Aside from the note of Sherman, Theorem 1 apparently has never been stated explicitly in the control systems literature. In fact, one finds many statements to the effect that loss functions of the general type (2) cannot be conveniently handled mathematically.

(*c*) In the sequel, we shall be dealing mainly with vector-valued random variables. In that case, the estimation problem is stated as: Given a vector-valued random process $\{\mathbf{x}(t)\}$ and observed random variables $\mathbf{y}(t_0)$, ..., $\mathbf{y}(t)$, where $\mathbf{y}(t) = \mathbf{M}\mathbf{x}(t)$ ($\mathbf{M}$ being a singular matrix; in other words, not all co-ordinates of $\mathbf{x}(t)$ can be observed), find an estimate $\mathbf{X}(t_1)$ which minimizes the expected loss $E[L(\|\mathbf{x}(t_1) - \mathbf{X}(t_1)\|)]$, $\| \ \|$ being the norm of a vector.

Theorem 1 remains true in the vector case also, provided we re- quire that the conditional distribution function of the $n$ coordi- nates of the vector $\mathbf{x}(t_1)$,

$$Pr[x_1(t_1) \le \xi_1, \ldots, x_n(t_1) \le \xi_n | \mathbf{y}(t_0), \ldots, \mathbf{y}(t)] = F(\xi_1, \ldots, \xi_n)$$

be symmetric with respect to the $n$ variables $\xi_1 - \bar{\xi}_1, \ldots, \xi_n - \bar{\xi}_n$ and convex in the region where all of these variables are negative.

## Orthogonal Projections

The explicit calculation of the optimal estimate as a function of the observed variables is, in general, impossible. There is an important exception: The processes $\{x_1(t)\}$, $\{x_2(t)\}$ are gaussian.

On the other hand, if we attempt to get an optimal estimate under the restriction $L(\varepsilon) = \varepsilon^2$ and the additional requirement that the estimate be a linear function of the observed random variables, we get an estimate which is identical with the optimal estimate in the gaussian case, without the assumption of linearity or quadratic loss function. This shows that results obtainable by linear estimation can be bettered by nonlinear estimation only when (i) the random processes are nongaussian and even then (in view of Theorem 5, (*C*)) only (ii) by considering at least third-order probability distribution functions.

In the special cases just mentioned, the explicit solution of the estimation problem is most easily understood with the help of a geometric picture. This is the subject of the present section.

Consider the (real-valued) random variables $y(t_0)$, …, $y(t)$. The set of all linear combinations of these random variables with real coefficients

$$\sum_{i=t_0}^{t} a_i y(i) \quad (6)$$

forms a *vector space (linear manifold)* which we denote by $\mathcal{Y}(t)$. We regard, abstractly, any expression of the form (6) as "point" or "vector" in $\mathcal{Y}(t)$; this use of the word "vector" should not be confused, of course, with "vector-valued" random variables, etc. Since we do not want to fix the value of $t$ (i.e., the total number of possible observations), $\mathcal{Y}(t)$ should be regarded as a finite-dimensional subspace of the space of all possible observations.

Given any two vectors $u$, $v$ in $\mathcal{Y}(t)$ (i.e., random variables expressible in the form (6)), we say that $u$ and $v$ are *orthogonal* if $Euv = 0$. Using the Schmidt orthogonalization procedure, as described for instance by Doob [15], p. 151, or by Loève [16], p. 459, it is easy to select an *orthonormal basis* in $\mathcal{Y}(t)$. By this is meant a set of vectors $e_{t_0}, \ldots, e_t$ in $\mathcal{Y}(t)$ such that any vector in $\mathcal{Y}(t)$ can be expressed as a unique linear combination of $e_{t_0}, \ldots, e_t$ and

$$\left.\begin{aligned} Ee_ie_j = \delta_{ij} = 1 \quad &\text{if} \quad i = j \\ = 0 \quad &\text{if} \quad i \neq j \end{aligned}\right\} \quad (i, j = t_0, \ldots, t) \tag{7}$$

Thus any vector $\bar{x}$ in $\mathcal{Y}(t)$. is given by

$$\bar{x} = \sum_{i=t_0}^{t} a_i e_i$$

and so the coefficients $a_i$ can be immediately determined with the aid of (7):

$$E\bar{x}e_j = E\left(\sum_{i=t_0}^{t} a_i e_i\right) e_j = \sum_{i=t_0}^{t} a_i Ee_ie_j = \sum_{i=t_0}^{t} a_i \delta_{ij} = a_j \tag{8}$$

It follows further that any random variable $x$ (not necessarily in $\mathcal{Y}(t)$) can be uniquely decomposed into two parts: a part $\bar{x}$ in $\mathcal{Y}(t)$ and a part $\tilde{x}$ orthogonal to $\mathcal{Y}(t)$ (i.e., orthogonal to every vector in $\mathcal{Y}(t)$). In fact, we can write

$$x = \bar{x} + \tilde{x} = \sum_{i=t_0}^{t} (Exe_i)e_i + \tilde{x} \tag{9}$$

Thus $\bar{x}$ is uniquely determined by equation (9) and is obviously a vector in $\mathcal{Y}(t)$. Therefore $\tilde{x}$ is also uniquely determined; it remains to check that it is orthogonal to $\mathcal{Y}(t)$:

$$E\tilde{x}e_i = E(x - \bar{x})e_i = Exe_i - E\bar{x}e_i$$

Now the co-ordinates of $\bar{x}$ with respect to the basis $e_{t_0}, \ldots, e_t$ are given either in the form $E\bar{x}e_i$ (as in (8)) or in the form $Exe_i$ (as in (9)). Since the co-ordinates are unique, $Exe_i = E\bar{x}e_i$ $(i = t_0, \ldots, t)$; hence $E\tilde{x}e_i = 0$ and $\tilde{x}$ is orthogonal to every base vector $e_i$; and therefore to $\mathcal{Y}(t)$. We call $\bar{x}$ the *orthogonal projection* of $x$ on $\mathcal{Y}(t)$.

There is another way in which the orthogonal projection can be characterized: $\bar{x}$ is that vector in $\mathcal{Y}(t)$ (i.e., that *linear* function of the random variables $y(t_0), \ldots, y(t)$) which minimizes the quadratic loss function. In fact, if $\bar{w}$ is any other vector in $\mathcal{Y}(t)$, we have

$$E(x - \bar{w})^2 = E(\tilde{x} + \bar{x} - \bar{w})^2 = E[(x - \bar{x}) + (\bar{x} - \bar{w})]^2$$

Since $\tilde{x}$ is orthogonal to every vector in $\mathcal{Y}(t)$ and in particular to $\bar{x} - \bar{w}$ we have

$$E(x - \bar{w})^2 = E(x - \bar{x})^2 + E(\bar{x} - \bar{w})^2 \geq E(x - \bar{x})^2 \tag{10}$$

This shows that, if $\bar{w}$ also minimizes the quadratic loss, we must have $E(\bar{x} - \bar{w})^2 = 0$ which means that the random variables $\bar{x}$ and $\bar{w}$ are equal (except possibly for a set of events whose probability is zero).

These results may be summarized as follows:

**Theorem 2.** Let $\{x(t)\}$, $\{y(t)\}$ *random processes with zero mean* (*i.e.*, $Ex(t) = Ey(t) = 0$ *for all* $t$). *We observe* $y(t_0), \ldots, y(t)$.

*If either*

(*A*) *the random processes* $\{x(t)\}$, $\{y(t)\}$ *are gaussian; or*

(*B*) *the optimal estimate is restricted to be a linear function of the observed random variables and* $L(\varepsilon) = \varepsilon^2$;

*then*

$$\begin{aligned} x^*(t_1|t) &= \textit{optimal estimate of } x(t_1) \text{ given } y(t_0), \ldots, y(t) \\ &= \textit{orthogonal projection } \bar{x}\,(t_1|t) \textit{ of } x(t_1) \text{ on } \mathcal{Y}(t). \end{aligned} \tag{11}$$

These results are well-known though not easily accessible in the control systems literature. See Doob [15], pp. 75–78, or Pugachev [26]. It is sometimes convenient to denote the orthogonal projection by

$$\bar{x}(t_1 \mid t) \equiv x^*(t_1 \mid t) = \hat{E}[x(t_1) \mid \mathcal{Y}(t)]$$

The notation $\hat{E}$ is motivated by part (*b*) of the theorem: If the stochastic processes in question are gaussian, then orthogonal projection is actually identical with conditional expectation.

**Proof.** (*A*) This is a direct consequence of the remarks in connection with (10).

(*B*) Since $x(t)$, $y(t)$ are random variables with zero mean, it is clear from formula (9) that the orthogonal part $\tilde{x}\,(t_1|t)$ of $x(t_1)$ with respect to the linear manifold $\mathcal{Y}(t)$ is also a random variable with zero mean. Orthogonal random variables with zero mean are uncorrelated; if they are also gaussian then (by Theorem 5 (*B*)) they are independent. Thus

$$\begin{aligned} 0 = E\,\tilde{x}\,(t_1|t) \quad &= E[\,\tilde{x}\,(t_1|t)|y(t_0), \ldots, y(t)] \\ &= E[x\,(t_1) - \bar{x}\,(t_1|t)|y(t_0), \ldots, y(t)] \\ &= E[x\,(t_1)|y(t_0), \ldots, y(t)] - \bar{x}\,(t_1|t) = 0 \end{aligned}$$

**Remarks.** (*d*) A rigorous formulation of the contents of this section as $t \to \infty$ requires some elementary notions from the theory of Hilbert space. See Doob [15] and Loève [16 ].

(*e*) The physical interpretation of Theorem 2 is largely a matter of taste. If we are not worried about the assumption of gaussianness, part (*A*) shows that the orthogonal projection is the optimal estimate for all reasonable loss functions. If we do worry about gaussianness, even if we are resigned to consider only linear estimates, we know that orthogonal projections are *not* the optimal estimate for many reasonable loss functions. Since in practice it is difficult to ascertain to what degree of approximation a random process of physical origin is gaussian, it is hard to decide whether Theorem 2 has very broad or very limited significance.

(*f*) Theorem 2 is immediately generalized for the case of vector-valued random variables. In fact, we define the linear manifold $\mathcal{Y}(t)$ generated by $\mathbf{y}(t_0), \ldots, \mathbf{y}(t)$ to be the set of all linear combinations

$$\sum_{i=t_0}^{t} \sum_{j=1}^{m} a_{ij} y_j(i)$$

of all $m$ co-ordinates of each of the random vectors $\mathbf{y}(t_0), \ldots, \mathbf{y}(t)$. The rest of the story proceeds as before.

(*g*) Theorem 2 states in effect that the optimal estimate under conditions (*A*) or (*B*) is a linear combination of all previous observations. In other words, the optimal estimate can be regarded as the output of a linear filter, with the input being the actually occurring values of the observable random variables; Theorem 2 gives a way of computing the impulse response of the optimal filter. As pointed out before, knowledge of this impulse response is not a complete solution of the problem; for this reason, no explicit formulas for the calculation of the impulse response will be given.

## Models for Random Processes

In dealing with physical phenomena, it is not sufficient to give an empirical description but one must have also some idea of the underlying causes. Without being able to separate in some sense causes and effects, i.e., without the assumption of causality, one can hardly hope for useful results.

It is a fairly generally accepted fact that primary macroscopic sources of random phenomena are independent gaussian processes.[5] A well-known example is the noise voltage produced in a resistor due to thermal agitation. In most cases, *observed* random phenomena are not describable by independent random variables. The statistical dependence (correlation) between random signals observed at different times is usually explained by the presence of a dynamic system between the primary random source and the observer. *Thus a random function of time may be thought of as the output of a dynamic system excited by an independent gaussian random process.*

An important property of gaussian random signals is that they remain gaussian after passing through a linear system (Theorem 5 (*A*)). Assuming independent gaussian primary random sources, if the observed random signal is also gaussian, we may assume that the dynamic system between the observer and the primary source is *linear*. This conclusion may be forced on us also because of lack of detailed knowledge of the statistical properties of the observed random signal: Given any random process with known first and second-order averages, we can find a gaussian random process with the same properties (Theorem 5 (*C*)). Thus gaussian distributions and linear dynamics are natural, mutually plausible assumptions particularly when the statistical data are scant.

How is a dynamic system (linear or nonlinear) described? The fundamental concept is the notion of the *state*. By this is meant, intuitively, some quantitative information (a set of numbers, a function, etc.) which is the least amount of data one has to know about the past behavior of the system in order to predict its future behavior. The dynamics is then described in terms of *state transitions*, i.e., one must specify how one state is transformed into another as time passes.

A *linear* dynamic system may be described in general by the vector differential equation

$$\left.\begin{aligned} d\mathbf{x}/dt &= \mathbf{F}(t)\mathbf{x} + \mathbf{D}(t)\mathbf{u}(t) \\ \text{and} \qquad \mathbf{y}(t) &= \mathbf{M}(t)\mathbf{x}(t) \end{aligned}\right\} \tag{12}$$

where $\mathbf{x}$ is an $n$-vector, the *state* of the system (the components $x_i$ of $\mathbf{x}$ are called *state variables*); $\mathbf{u}(t)$ is an $m$-vector ($m \le n$) representing the *inputs* to the system; $\mathbf{F}(t)$ and $\mathbf{D}(t)$ are $n \times n$, respectively, $n \times m$ matrices. If all coefficients of $\mathbf{F}(t)$, $\mathbf{D}(t)$, $\mathbf{M}(t)$ are constants, we say that the dynamic system (12) is *time-invariant* or *stationary*. Finally, $\mathbf{y}(t)$ is a *p-vector* denoting the outputs of the system; $\mathbf{M}(t)$ is an $n \times p$ matrix; $p \le n$

The physical interpretation of (12) has been discussed in detail elsewhere [18, 20, 23]. A look at the block diagram in Fig. 1 may be helpful. This is not an ordinary but a matrix block diagram (as revealed by the fat lines indicating signal flow). The integrator in

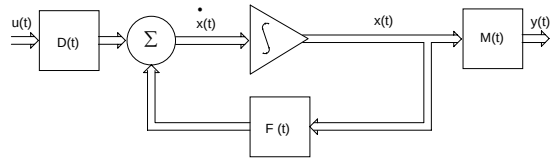



**Fig 1. Matrix block diagram of the general linear continuous-dynamic system**

[5] The probability distributions will be gaussian because macroscopic random effects may be thought of as the superposition of very many microscopic random effects; under very general conditions, such aggregate effects tend to be gaussian, regardless of the statistical properties of the microscopic effects. The assumption of independence in this context is motivated by the fact that microscopic phenomena tend to take place much more rapidly than macroscopic phenomena; thus primary random sources would appear to be independent on a macroscopic time scale.

Fig. 1 actually stands for $n$ integrators such that the output of each is a state variable; $\mathbf{F}(t)$ indicates how the outputs of the integrators are fed back to the inputs of the integrators. Thus $f_{ij}(t)$ is the coefficient with which the output of the $j$th integrator is fed back to the input of the $i$th integrator. It is not hard to relate this formalism to more conventional methods of linear system analysis.

If we assume that the system (12) is stationary and that $\mathbf{u}(t)$ is constant during each sampling period, that is

$$\mathbf{u}(t + \tau) = \mathbf{u}(t); \quad 0 \le \tau < 1, \quad t = 0, 1, \ldots \tag{13}$$

then (12) can be readily transformed into the more convenient discrete form.

$$\mathbf{x}(t + 1) = \mathbf{\Phi}(1)\mathbf{x}(t) + \mathbf{\Delta}(1)\mathbf{u}(t); \quad t = 0, 1, \ldots$$

where [18, 20]

$$\mathbf{\Phi}(1) = \exp \mathbf{F} = \sum_{i=0}^{\infty} \mathbf{F}^i/i! \quad (\mathbf{F}^0 = \text{unit matrix})$$

and

$$\mathbf{\Delta}(1) = \left( \int_0^1 \exp \mathbf{F}\tau d\tau \right) \mathbf{D}$$

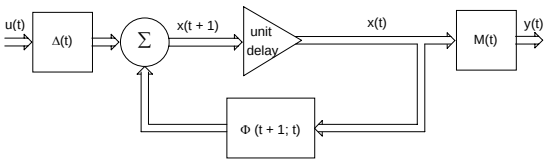



**Fig 2. Matrix block diagram of the general linear discrete-dynamic system**

See Fig. 2. One could also express $\exp \mathbf{F}\tau$ in closed form using Laplace transform methods [18, 20, 22, 24]. If $\mathbf{u}(t)$ satisfies (13) but the system (12) is nonstationary, we can write analogously

$$\left.\begin{aligned} \mathbf{x}(t + 1) &= \mathbf{\Phi}(t + 1; t) + \mathbf{\Delta}(t)\mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{M}(t)\mathbf{x}(t) \end{aligned}\right\} \quad t = 0, 1, \ldots \tag{14}$$

but of course now $\mathbf{\Phi}(t + 1; t)$, $\mathbf{\Delta}(t)$ cannot be expressed in general in closed form. Equations of type (14) are encountered frequently also in the study of complicated sampled-data systems [22]. See Fig. 2

$\mathbf{\Phi}(t + 1; t)$ is the *transition matrix* of the system (12) or (14). The notation $\mathbf{\Phi}(t_2; t_1)$ ($t_2$, $t_1$ = integers) indicates transition from time $t_1$ to time $t_2$. Evidently $\mathbf{\Phi}(t; t) = \mathbf{I}$ = unit matrix. If the system (12) is stationary then $\mathbf{\Phi}(t + 1; t) = \mathbf{\Phi}(t + 1 - t) = \mathbf{\Phi}(1)$ = const. Note also the product rule: $\mathbf{\Phi}(t; s)\mathbf{\Phi}(s; r) = \mathbf{\Phi}(t; r)$ and the inverse rule $\mathbf{\Phi}^{-1}(t; s) = \mathbf{\Phi}(s; t)$, where $t$, $s$, $r$ are integers. In a stationary system, $\mathbf{\Phi}(t; \tau) = \exp \mathbf{F}(t - \tau)$.

As a result of the preceding discussion, we shall represent random phenomena by the model

$$\mathbf{x}(t + 1) = \mathbf{\Phi}(t + 1; t)\mathbf{x}(t) + \mathbf{u}(t) \tag{15}$$

where $\{\mathbf{u}(t)\}$ is a vector-valued, independent, gaussian random process, with zero mean, which is completely described by (in view of Theorem 5 (*C*))

$$E\mathbf{u}(t) = \mathbf{0} \quad \text{for all } t;$$

$$E\mathbf{u}(t)\mathbf{u}'(s) = \mathbf{0} \quad \text{if } t \ne s$$

$$E\mathbf{u}(t)\mathbf{u}'(t) = \mathbf{G}(t).$$

Of course (Theorem 5 (*A*)), $\mathbf{x}(t)$ is then also a gaussian random process with zero mean, but it is no longer independent. In fact, if we consider (15) in the steady state (assuming it is a stable system), in other words, if we neglect the initial state $\mathbf{x}(t_0)$, then

Transactions of the ASME–Journal of Basic Engineering, 82 (Series D): 35-45. Copyright © 1960 by ASME

$$\mathrm{x}(t) = \sum_{r=-\infty}^{t-1} \mathbf{\Phi}(t;\, r+1)\mathbf{u}(r).$$

Therefore if $t \geq s$ we have

$$E\mathbf{x}(t)\mathbf{x}'(s) = \sum_{r=-\infty}^{s-1} \mathbf{\Phi}(t;\, r+1)\mathbf{Q}(r)\,\mathbf{\Phi}'(s;\, r+1).$$

Thus if we assume a linear dynamic model and know the statistical properties of the gaussian random excitation, it is easy to find the corresponding statistical properties of the gaussian random process $\{\mathbf{x}(t)\}$.

In real life, however, the situation is usually reversed. One is given the covariance matrix $E\mathbf{x}(t)\mathbf{x}'(s)$ (or rather, one attempts to estimate the matrix from limited statistical data) and the problem is to get (15) and the statistical properties of $\mathbf{u}(t)$. This is a subtle and presently largely unsolved problem in experimentation and data reduction. As in the vast majority of the engineering literature on the Wiener problem, we shall find it convenient to start with the model (15) and regard the problem of obtaining the model itself as a separate question. To be sure, the two problems *should* be optimized jointly if possible; the author is not aware, however, of any study of the *joint* optimization problem.

In summary, the following assumptions are made about random processes:

*Physical random phenomena may be thought of* as *due to primary random sources exciting dynamic systems. The primary sources are assumed to be independent gaussian random processes with zero mean; the dynamic systems will be linear. The random processes are therefore described by models such as* (15). *The question of how the numbers specifying the model are obtained from experimental data will not be considered.*

## Solution of the Wiener problem

Let us now define the principal problem of the paper.

**Problem I.** *Consider the dynamic model*

$$\mathbf{x}(t+1) = \mathbf{\Phi}(t+1;\, t)\mathbf{x}(t) + \mathbf{u}(t) \tag{16}$$

$$\mathbf{y}(t) = \mathbf{M}(t)\mathbf{x}(t) \tag{17}$$

*where* $\mathbf{u}(t)$ is *an independent gaussian random process of n-vectors with zero mean,* $\mathbf{x}(t)$ is *an n-vector,* $\mathbf{y}(t)$ is *a p-vector* ($p \leq n$), $\mathbf{\Phi}(t+1;\, t)$, $\mathbf{M}(t)$ *are* $n \times n$, *resp.* $p \times n$, *matrices whose elements are nonrandom functions of time.*

*Given the observed values of* $\mathbf{y}(t_0)$, ..., $\mathbf{y}(t)$ *find an estimate* $\mathbf{x}^*(t_1|t)$ *of* $\mathbf{x}(t_1)$ *which minimizes the expected loss.* (See Fig. 2, where $\mathbf{\Delta}(t) = \mathbf{I}$.)

This problem includes as a special case the problems of filtering, prediction, and data smoothing mentioned earlier. It includes also the problem of reconstructing all the state variables of a linear dynamic system from noisy observations of some of the state variables ($p < n$!).

From Theorem 2-*a* we know that the solution of Problem I is simply the orthogonal projection of $\mathbf{x}(t_1)$ on the linear manifold $\mathcal{Y}(t)$ generated by the observed random variables. As remarked in the Introduction, this is to be accomplished by means of a linear (not necessarily stationary!) dynamic system of the general form (14). With this in mind, we proceed as follows.

Assume that $\mathbf{y}(t_0)$, ..., $\mathbf{y}(t-1)$ have been measured, i.e., that $\mathcal{Y}(t-1)$ is known. Next, at time $t$, the random variable $\mathbf{y}(t)$ is measured. As before let $\tilde{\mathbf{y}}(t|t-1)$ be the component of $\mathbf{y}(t)$ orthogonal to $\mathcal{Y}(t-1)$. If $\tilde{\mathbf{y}}(t|t-1) \equiv 0$, which means that the values of all components of this random vector are zero for almost every possible event, then $\mathcal{Y}(t)$ is obviously the same as $\mathcal{Y}(t-1)$ and therefore the measurement of $\mathbf{y}(t)$ does not convey any additional information. This is not likely to happen in a physically meaningful situation. In any case, $\tilde{\mathbf{y}}(t|t-1)$ generates a linear manifold (possibly 0) which we denote by $\mathcal{Z}(t)$. By definition, $\mathcal{Y}(t-1)$ and $\mathcal{Z}(t)$ taken together are the same manifold as $\mathcal{Y}(t)$, and every vector in $\mathcal{Z}(t)$ is orthogonal to every vector in $\mathcal{Y}(t-1)$.

Assuming by induction that $\mathbf{x}^*(t_1 - 1|t-1)$ is known, we can write:

$$\begin{aligned}\mathbf{x}^*(t_1|t) &= \hat{E}\,[\mathbf{x}(t_1)|\mathcal{Y}(t)] = \hat{E}\,[\mathbf{x}(t_1)|\mathcal{Y}(t-1)] + \hat{E}\,[\mathbf{x}(t_1)|\mathcal{Z}(t)] \\ &= \mathbf{\Phi}(t+1;\, t)\,\mathbf{x}^*(t_1-1|t-1) + \hat{E}\,[\mathbf{u}(t_1-1)|\mathcal{Y}(t-1)] \\ &\quad + \hat{E}\,[\mathbf{x}(t_1)|\mathcal{Z}(t)] \end{aligned} \tag{18}$$

where the last line is obtained using (16).

Let $t_1 = t + s$, where $s$ is any integer. If $s \geq 0$, then $\mathbf{u}(t_1 - 1)$ is independent of $\mathcal{Y}(t-1)$. This is because $\mathbf{u}(t_1 - 1) = \mathbf{u}(t + s - 1)$ is then independent of $\mathbf{u}(t-2)$, $\mathbf{u}(t-3)$, ... and therefore by (16–17), independent of $\mathbf{y}(t_0)$, ..., $\mathbf{y}(t-1)$, hence independent of $\mathcal{Y}(t-1)$. Since, for all $t$, $\mathbf{u}(t_0)$ has zero mean by assumption, it follows that $\mathbf{u}(t_1 - 1)$ ($s \geq 0$) is orthogonal to $\mathcal{Y}(t-1)$. Thus if $s \geq 0$, the second term on the right-hand side of (18) vanishes; if $s < 0$, considerable complications result in evaluating this term. We shall consider only the case $t_1 \geq t$. Furthermore, it will suffice to consider in detail only the case $t_1 = t + 1$ since the other cases can be easily reduced to this one.

The last term in (18) must be a linear operation on the random variable $\tilde{\mathbf{y}}(t|t-1)$:

$$\hat{E}\,[\mathbf{x}(t+1)|\mathcal{Z}(t)] = \mathbf{\Delta}^*(t)\,\tilde{\mathbf{y}}(t|t-1) \tag{19}$$

where $\mathbf{\Delta}^*(t)$ is an $n \times p$ matrix, and the star refers to "optimal filtering".

The component of $\mathbf{y}(t)$ lying in $\mathcal{Y}(t-1)$ is $\bar{\mathbf{y}}(t|t-1) = \mathbf{M}(t)\mathbf{x}^*(t|t-1)$. Hence

$$\tilde{\mathbf{y}}(t|t-1) = \mathbf{y}(t) - \bar{\mathbf{y}}(t|t-1) = \mathbf{y}(t) - \mathbf{M}(t)\mathbf{x}^*(t|t-1). \tag{20}$$

Combining (18-20) (see Fig. 3) we obtain

$$\mathbf{x}^*(t+1|t) = \mathbf{\Phi}^*(t+1;\, t)\mathbf{x}^*(t|t-1) + \mathbf{\Delta}^*(t)\mathbf{y}(t) \tag{21}$$

where

$$\mathbf{\Phi}^*(t+1;\, t) = \mathbf{\Phi}(t+1;\, t) - \mathbf{\Delta}^*(t)\mathbf{M}(t) \tag{22}$$

Thus optimal estimation is performed by a linear dynamic system of the same form as (14). The state of the estimator is the previous estimate, the input is the last measured value of the observable random variable $\mathbf{y}(t)$, the transition matrix is given by (22). Notice that physical realization of the optimal filter requires only (*i*) the model of the random process (*ii*) the operator $\mathbf{\Delta}^*(t)$.

The estimation error is also governed by a linear dynamic system. In fact,

$$\begin{aligned}\tilde{\mathbf{x}}(t+1|t) &= \mathbf{x}(t+1) - \mathbf{x}^*(t+1|t) \\ &= \mathbf{\Phi}(t+1;\, t)\mathbf{x}(t) + \mathbf{u}(t) - \mathbf{\Phi}^*(t+1;\, t)\mathbf{x}^*(t|t-1) \\ &\quad - \mathbf{\Delta}^*(t)\mathbf{M}(t)\mathbf{x}(t)\end{aligned}$$

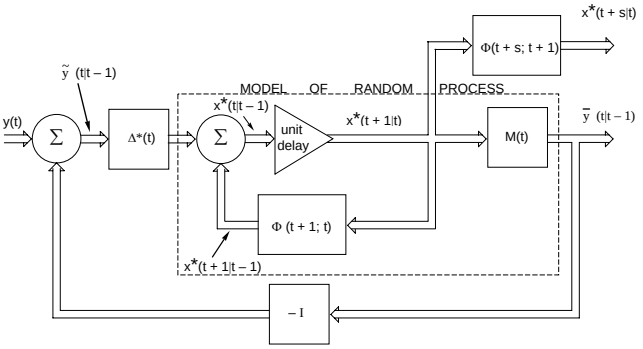



**Fig. 3 Matrix block diagram of optimal filter**

$$= \mathbf{\Phi}^*(t+1;\, t)\, \tilde{\mathbf{x}}\,(t|t-1) + \mathbf{u}(t) \tag{23}$$

Thus $\mathbf{\Phi}^*$ is also the transition matrix of the linear dynamic system governing the error.

From (23) we obtain at once a recursion relation for the covariance matrix $\mathbf{P}^*(t)$ of the optimal error $\tilde{\mathbf{x}}\,(t|t-1)$. Noting that $\mathbf{u}(t)$ is independent of $\mathbf{x}(t)$ and therefore of $\tilde{\mathbf{x}}\,(t|t-1)$ we get

$$\begin{aligned}\mathbf{P}^*(t+1) &= E\,\tilde{\mathbf{x}}\,(t+1|t)\,\tilde{\mathbf{x}}\,'(t+1|t)\\ &= \mathbf{\Phi}^*(t+1;\, t)E\,\tilde{\mathbf{x}}\,(t|t-1)\,\tilde{\mathbf{x}}\,'(t|t-1)\mathbf{\Phi}^{*\prime}\,(t+1;\, t) + \mathbf{Q}(t)\\ &= \mathbf{\Phi}^*(t+1;\, t)E\,\tilde{\mathbf{x}}\,(t|t-1)\,\tilde{\mathbf{x}}\,'(t|t-1)\mathbf{\Phi}'(t+1;\, t) + \mathbf{Q}(t)\\ &= \mathbf{\Phi}^*(t+1;\, t)\mathbf{P}^*(t)\mathbf{\Phi}'(t+1;\, t) + \mathbf{Q}(t) \end{aligned} \tag{24}$$

where $\mathbf{Q}(t) = E\mathbf{u}(t)\mathbf{u}'(t)$.

There remains the problem of obtaining an explicit formula for $\mathbf{\Delta}^*$ (and thus also for $\mathbf{\Phi}^*$). Since,

$$\tilde{\mathbf{x}}\,(t+1)|Z(t)) = \mathbf{x}(t+1) - \hat{E}\,[\mathbf{x}(t+1)|Z(t)]$$

is orthogonal to $\tilde{\mathbf{y}}\,(t\,|t-1)$, it follows that by (19) that

$$\begin{aligned} 0 &= E[\mathbf{x}(t+1) - \mathbf{\Delta}^*(t)\,\tilde{\mathbf{y}}\,(t|t-1)]\,\tilde{\mathbf{y}}\,'(t|t-1)\\ &= E\mathbf{x}(t+1)\,\tilde{\mathbf{y}}\,'(t|t-1) - \mathbf{\Delta}^*(t)E\;\tilde{\mathbf{y}}\,(t|t-1)\,\tilde{\mathbf{y}}\,'(t|t-1). \end{aligned}$$

Noting that $\bar{\mathbf{x}}\,(t+1|t-1)$ is orthogonal to $Z(t)$, the definition of $\mathbf{P}(t)$ given earlier, and (17), it follows further

$$\begin{aligned} 0 &= E\,\tilde{\mathbf{x}}\,(t+1|t-1)\,\tilde{\mathbf{y}}\,'(t|t-1) - \mathbf{\Delta}^*(t)\mathbf{M}(t)\mathbf{P}^*(t)\mathbf{M}'(t)\\ &= E[\mathbf{\Phi}(t+1;\, t)\,\tilde{\mathbf{x}}\,(t|t-1) + \mathbf{u}(t|t-1)]\,\tilde{\mathbf{x}}\,'(t|t-1)\mathbf{M}'(t)\\ &\qquad - \mathbf{\Delta}^*(t)\mathbf{M}(t)\mathbf{P}^*(t)\mathbf{M}'(t). \end{aligned}$$

Finally, since $\mathbf{u}(t)$ is independent of $\mathbf{x}(t)$,

$$0 = \mathbf{\Phi}(t+1;\, t)\mathbf{P}^*(t)\mathbf{M}'(t) - \mathbf{\Delta}^*(t)\mathbf{M}(t)\mathbf{P}^*(t)\mathbf{M}'(t).$$

Now the matrix $\mathbf{M}(t)\mathbf{P}^*(t)\mathbf{M}'(t)$will be positive definite and hence invertible whenever $\mathbf{P}^*(t)$ is positive definite, provided that none of the rows of $\mathbf{M}(t)$ are linearly dependent at any time, in other words, that none of the observed scalar random variables $y_1(t)$, ..., $y_m(t)$, is a linear combination of the others. Under these circumstances we get finally:

$$\mathbf{\Delta}^*(t) = \mathbf{\Phi}(t+1;\, t)\mathbf{P}^*(t)\mathbf{M}'(t)[\mathbf{M}(t)\mathbf{P}^*(t)\mathbf{M}'(t)]^{-1} \tag{25}$$

Since observations start at $t_0$, $\tilde{\mathbf{x}}\,(t_0|t_0-1) = \mathbf{x}(t_0)$; to begin the iterative evaluation of $\mathbf{P}^*(t)$ by means of equation (24), we must obviously specify $\mathbf{P}^*(t_0) = E\mathbf{x}(t_0)\mathbf{x}'(t_0)$. Assuming this matrix is positive definite, equation (25) then yields $\mathbf{\Delta}^*(t_0)$; equation (22) $\mathbf{\Phi}^*(t_0+1;\, t_0)$, and equation (24) $\mathbf{P}^*(t_0+1)$, completing the cycle. If now $\mathbf{Q}(t)$ is positive definite, then all the $\mathbf{P}^*(t)$ will be positive definite and the requirements in deriving (25) will be satisfied at each step.

Now we remove the restriction that $t_1 = t+1$. Since $\mathbf{u}(t)$ is orthogonal to $\mathcal{Y}(t)$, we have

$$\mathbf{x}^*(t+1|t) = \hat{E}\,[\mathbf{\Phi}(t+1;\, t)\mathbf{x}(t) + \mathbf{u}(t)|\mathcal{Y}(t)] = \mathbf{\Phi}(t+1;\, t)\mathbf{x}^*(t|t)$$

Hence if $\mathbf{\Phi}(t+1;\, t)$ has an inverse $\mathbf{\Phi}(t;\, t+1)$ (which is always the case when $\mathbf{\Phi}$ is the transition matrix of a dynamic system describable by a differential equation) we have

$$\mathbf{x}^*(t|t) = \mathbf{\Phi}(t;\, t+1)\mathbf{x}^*(t+1|t)$$

If $t_1 \geq t+1$, we first observe by repeated application of (16) that

$$\mathbf{x}(t+s) = \mathbf{\Phi}(t+s;\, t+1)\mathbf{x}(t+1) + \sum_{r1}^{s-1} \mathbf{\Phi}(t+s;\, t+r)\mathbf{u}(t+r) \qquad (s \geq 1)$$

Since $\mathbf{u}(t+s-1)$, …, $\mathbf{u}(t+1)$ are all orthogonal to $\mathcal{Y}(t)$,

$$\begin{aligned}\mathbf{x}^*(t+s|t) &= \hat{E}\,[\mathbf{x}(t+s)|\mathcal{Y}(t)]\\ &= \hat{E}\,[\mathbf{\Phi}(t+s;\, t+1)\mathbf{x}(t+1)|\mathcal{Y}(t)]\\ &= \mathbf{\Phi}(t+s;\, t+1)\mathbf{x}^*(t+1|t) \qquad (s \geq 1)\end{aligned}$$

If $s < 0$, the results are similar, but $\mathbf{x}^*(t-s|t)$ will have $(1-s)(n-p)$ co-ordinates.

The results of this section may be summarized as follows:

**Theorem 3.** (*Solution of the Wiener Problem*)

*Consider Problem I. The optimal estimate* $\mathbf{x}^*(t+1|t)$ *of* $\mathbf{x}(t+1)$ *given* $\mathbf{y}(t_0)$, ..., $\mathbf{y}(t)$ is *generated by the linear dynamic system*

$$\mathbf{x}^*(t+1|t) = \mathbf{\Phi}^*(t+1;\, t)\mathbf{x}^*(t|t-1) + \mathbf{\Delta}^*(t)\mathbf{y}(t) \tag{21}$$

*The estimation error is given by*

$$\tilde{\mathbf{x}}\,(t+1|t) = \mathbf{\Phi}^*(t+1;\, t)\,\tilde{\mathbf{x}}\,(t|t-1) + \mathbf{u}(t) \tag{23}$$

*The covariance matrix of the estimation error is*

$$\operatorname{cov}\,\tilde{\mathbf{x}}\,(t|t-1) = E\,\tilde{\mathbf{x}}\,(t|t-1)\,\tilde{\mathbf{x}}\,'(t|t-1) = \mathbf{P}^*(t) \tag{26}$$

*The expected quadratic loss is*

$$\sum_{i=1}^{n} E\tilde{x}_i^{\,2}(t\,|\,t-1) \;=\; \operatorname{trace}\,\mathbf{P}^*(t) \tag{27}$$

*The matrices* $\mathbf{\Delta}^*(t)$, $\mathbf{\Phi}^*(t+1;\, t)$, $\mathbf{P}^*(t)$ *are generated by the recursion relations*

$$\left.\begin{aligned} \mathbf{\Delta}^*(t) &= \mathbf{\Phi}(t+1;\, t)\mathbf{P}^*(t)\mathbf{M}'(t)[\mathbf{M}(t)\mathbf{P}^*(t)\mathbf{M}'(t)]^{-1} && (28)\\ \mathbf{\Phi}^*(t+1;\, t) &= \mathbf{\Phi}(t+1;\, t) - \mathbf{\Delta}^*(t)\mathbf{M}(t) && (29)\\ \mathbf{P}^*(t+1) &= \mathbf{\Phi}^*(t+1;\, t)\mathbf{P}^*(t)\mathbf{\Phi}'(t+1;\, t) + \mathbf{Q}(t) && (30)\end{aligned}\right\} \quad t \geq t_0$$

*In order to carry out the iterations, one must specify the covariance* $\mathbf{P}^*(t_0)$ *of* $\mathbf{x}(t_0)$ *and the covariance* $\mathbf{Q}(t)$ *of* $\mathbf{u}(t)$. *Finally, for any* $s \geq 0$, *if* $\mathbf{\Phi}$ is *invertible*

$$\begin{aligned}\mathbf{x}^*(t+s|t) &= \mathbf{\Phi}(t+s;\, t+1)\mathbf{x}^*(t+1|t)\\ &= \mathbf{\Phi}(t+s;\, t+1)\mathbf{\Phi}^*(t+1;\, t)\mathbf{\Phi}(t;\, t+s-1)\\ &\qquad \times \mathbf{x}^*(t+s-1|t-1)\\ &\quad + \mathbf{\Phi}(t+s;\, t+1)\mathbf{\Delta}^*(t)\mathbf{y}(t) \end{aligned} \tag{31}$$

*so that the estimate* $\mathbf{x}^*(t+s|t)$ $(s \geq 0)$ is *also given by a linear dynamic system of the type* (21).

**Remarks.** (*h*) Eliminating $\mathbf{\Delta}^*$ and $\mathbf{\Phi}^*$ from (28–30), a nonlinear difference equation is obtained for $\mathbf{P}^*(t)$:

$$\begin{aligned}\mathbf{P}^*(t+1) &= \mathbf{\Phi}(t+1;\, t)\{\mathbf{P}^*(t) - \mathbf{P}^*(t)\mathbf{M}'(t)[\mathbf{M}(t)\mathbf{P}^*(t)\mathbf{M}'(t)]^{-1}\\ &\qquad \times \mathbf{P}^*(t)\mathbf{M}(t)\}\mathbf{\Phi}'(t+1;\, t) + \mathbf{Q}(t) \qquad t \geq t_0 \end{aligned} \tag{32}$$

This equation is linear only if $\mathbf{M}(t)$ is invertible but then the problem is trivial since all components of the random vector $\mathbf{x}(t)$ are observable $\mathbf{P}^*(t+1) = \mathbf{Q}(t)$. Observe that equation (32) plays a role in the present theory analogous to that of the Wiener-Hopf equation in the conventional theory.

Once $\mathbf{P}^*(t)$ has been computed via (32) starting at $t = t_0$, the explicit specification of the optimal linear filter is immediately available from formulas (29-30). Of course, the solution of Equation (32), or of its differential-equation equivalent, is a much simpler task than solution of the Wiener-Hopf equation.

(*i*) The results stated in Theorem 3 do not resolve completely Problem I. Little has been said, for instance, about the physical significance of the assumptions needed to obtain equation (25), the convergence and stability of the nonlinear difference equation (32), the stability of the optimal filter (21), etc. This can actually be done in a completely satisfactory way, but must be left to a future paper. In this connection, the principal guide and

tool turns out to be the duality theorem mentioned briefly in the next section. See [29].

(*j*) By letting the sampling period (equal to one so far) approach zero, the method can be used to obtain the specification of a differential equation for the optimal filter. To do this, i.e., to pass from equation (14) to equation (12), requires computing the logarithm $\mathbf{F}^*$ of the matrix $\mathbf{\Phi}^*$. But this can be done only if $\mathbf{\Phi}^*$ is nonsingular—which is easily seen *not* to be the case. This is because it is sufficient for the optimal filter to have $n - p$ state variables, rather than $n$, as the formalism of equation (22) would seem to imply. By appropriate modifications, therefore, equation (22) can be reduced to an equivalent set of only $n - p$ equations whose transition matrix is nonsingular. Details of this type will be covered in later publications.

(*k*) The dynamic system (21) is, in general, nonstationary. This is due to two things: (1) The time dependence of $\mathbf{\Phi}(t+1;\, t)$ and $\mathbf{M}(t)$; (2) the fact that the estimation starts at $t = t_0$ and improves as more data are accumulated. If $\mathbf{\Phi}$, $\mathbf{M}$ are constants, it can be shown that (21) becomes a stationary dynamic system in the limit $t \to \infty$. This is the case treated by the classical Wiener theory.

(*l*) It is noteworthy that the derivations given are not affected by the nonstationarity of the model for $\mathbf{x}(t)$ or the finiteness of available data. In fact, as far as the author is aware, the only explicit recursion relations given before for the growing-memory filter are due to Blum [12]. However, his results are much more complicated than ours.

(*m*) By inspection of Fig. 3 we see that the optimal filter is a feedback system, and that the signal after the first summer is white noise since $\tilde{\mathbf{y}}(t|t-1)$ is obviously an orthogonal random process. This corresponds to some well-known results in Wiener filtering, see, e.g., Smith [28], Chapter 6, Fig. 6–4. However, this is apparently the first *rigorous* proof that every Wiener filter is realizable by means of a feedback system. Moreover, it will be shown in another paper that such a filter is always *stable*, under very mild assumptions on the model (16–17). See [29].

## The Dual Problem

Let us now consider another problem which is conceptually very different from optimal estimation, namely, the noise-free regulator problem. In the simplest cases, this is:

**Problem II.** *Consider the dynamic system*

$$\mathbf{x}(t+1) = \hat{\mathbf{\Phi}}(t+1;\, t)\mathbf{x}(t) + \hat{\mathbf{M}}(t)\mathbf{u}(t) \tag{33}$$

*where* $\mathbf{x}(t)$ *is an n-vector,* $\mathbf{u}(t)$ *is an m-vector* ($m \le n$), $\hat{\mathbf{\Phi}}$, $\hat{\mathbf{M}}$ *are* $n \times n$ *resp.* $n \times m$ *matrices whose elements are nonrandom functions of time. Given any state* $\mathbf{x}(t)$ *at time t, we are to find a sequence* $\mathbf{u}(t)$, ..., $\mathbf{u}(T)$ *of control vectors which minimizes the performance index*

$$V[\mathbf{x}(t)] = \sum_{\tau=t}^{T+1} \mathbf{x}'(\tau)\mathbf{Q}(\tau)\mathbf{x}(\tau)$$

*Where* $\hat{\mathbf{Q}}(t)$ *is a positive definite matrix whose elements are nonrandom functions of time.* See Fig. 2, where $\mathbf{\Delta} = \hat{\mathbf{M}}$ and $\mathbf{M} = \mathbf{I}$.

Probabilistic considerations play no part in Problem II; it is implicitly assumed that every state variable can be measured exactly at each instant $t$, $t+1$, ..., $T$. It is customary to call $T \ge t$ the *terminal time* (it may be infinity).

The first general solution of the noise-free regulator problem is due to the author [18]. The main result is that the optimal control vectors $\mathbf{u}^*(t)$ are nonstationary linear functions of $\mathbf{x}(t)$. After a change in notation, the formulas of the Appendix, Reference [18] (see also Reference [23]) are as follows:

$$\mathbf{u}^*(t) = -\hat{\mathbf{\Delta}}^*(t)\mathbf{x}(t) \tag{34}$$

Under optimal control as given by (34), the "closed-loop" equations for the system are (see Fig. 4)

$$\mathbf{x}(t+1) = \hat{\mathbf{\Phi}}^*(t+1;\, t)\mathbf{x}(t)$$

and the minimum performance index at time $t$ is given by

$$V^*[\mathbf{x}(t)] = \mathbf{x}'(t)\mathbf{P}^*(t-1)\mathbf{x}(t)$$

The matrices $\hat{\mathbf{\Delta}}^*(t)$, $\hat{\mathbf{\Phi}}^*(t+1;\, t)$, $\hat{\mathbf{P}}^*(t)$ are determined by the recursion relations:

$$\left.\begin{aligned} \hat{\mathbf{\Delta}}^*(t) &= [\hat{\mathbf{M}}'(t)\,\hat{\mathbf{P}}^*(t)\,\hat{\mathbf{M}}(t)]^{-1}\hat{\mathbf{M}}'(t)\hat{\mathbf{P}}^*(t)\hat{\mathbf{\Phi}}(t+1;\, t) && (35)\\ \hat{\mathbf{\Phi}}^*(t+1;\, t) &= \hat{\mathbf{\Phi}}(t+1;\, t) - \hat{\mathbf{M}}(t)\,\hat{\mathbf{\Delta}}^*(t) && (36)\\ \hat{\mathbf{P}}^*(t-1) &= \hat{\mathbf{\Phi}}'(t+1;\, t)\hat{\mathbf{P}}^*(t)\,\hat{\mathbf{\Phi}}^*(t+1;\, t) \\ &\quad + \hat{\mathbf{Q}}(t) && (37) \end{aligned}\right\} t \le T$$

Initially we must set $\hat{\mathbf{P}}^*(T) = \hat{\mathbf{Q}}(T+1)$.

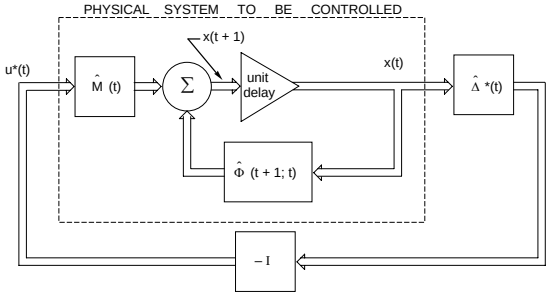



**Fig. 4 Matrix block diagram of optimal controller**

Comparing equations (35–37) with (28–30) and Fig. 3 with Fig. 4 we notice some interesting things which are expressed precisely by

**Theorem 4.** (*Duality Theorem*) *Problem I and Problem II are duals of each other in the following sense:*

*Let* $\tau \ge 0$. *Replace every matrix* $\mathbf{X}(t) = \mathbf{X}(t_0 + \tau)$ *in* (28–30) *by* $\hat{\mathbf{X}}'(t) = \hat{\mathbf{X}}'(T - \tau)$. *Then One has* (35–37). *Conversely, replace every matrix* $\hat{\mathbf{X}}(T - \tau)$ *in* (35–37) *by* $\mathbf{X}'(t_0 + \tau)$. *Then one has* (28–30).

**Proof.** Carry out the substitutions. For ease of reference, the dualities between the two problems are given in detail in Table 1.

**Table 1**

| | Problem I | Problem II |
|---|---|---|
| 1 | $\mathbf{x}(t)$ (unobservable) state variables of random process. | $\mathbf{x}(t)$ (observable) state variables of plant to be regulated. |
| 2 | $\mathbf{y}(t)$ observed random variables. | $\mathbf{u}(t)$ control variables |
| 3 | $t_0$ first observation. | $T$ last control action. |
| 4 | $\mathbf{\Phi}(t_0 + \tau + 1;\, t_0 + \tau)$ transition matrix. | $\hat{\mathbf{\Phi}}(T - \tau + 1;\, T - \tau)$ transition matrix. |
| 5 | $\mathbf{P}^*(t_0 + \tau)$ covariance of optimized estimation error. | $\hat{\mathbf{P}}^*(T - \tau)$ matrix of quadratic form for performance index under optimal regulation. |
| 6 | $\mathbf{\Delta}^*(t_0 + \tau)$ weighting of observation for optimal estimation. | $\hat{\mathbf{\Delta}}^*(T - \tau)$ weighting of state for optimal control. |
| 7 | $\mathbf{\Phi}^*(t_0 + \tau + 1;\, t_0 + \tau)$ transition matrix for optimal estimation error. | $\hat{\mathbf{\Phi}}^*(T - \tau + 1;\, T - \tau)$ transition matrix under optimal regulation. |
| 8 | $\mathbf{M}(t_0 + \tau)$ effect of state on observation. | $\hat{\mathbf{M}}(T - \tau)$ effect of control vectors on state. |
| 9 | $\mathbf{Q}(t_0 + \tau)$ covariance of random excitation. | $\hat{\mathbf{Q}}(T - \tau)$ matrix of quadratic form defining error criterion. |

**Remarks.** (*n*) The *mathematical* significance of the duality between Problem I and Problem II is that both problems reduce to the solution of the Wiener-Hopf-like equation (32).

(*o*) The *physical* significance of the duality is intriguing. Why are observations and control dual quantities?

Recent research [29] has shown that the essence of the Duality Theorem lies in the duality of constraints at the output (represented by the matrix $\hat{\mathbf{M}}(t)$ in Problem I) and constraints at the input (represented by the matrix $\hat{\mathbf{M}}(t)$ in Problem II).

(*p*) Applications of Wiener's methods to the solution of noise-free regulator problem have been known for a long time; see the recent textbook of Newton, Gould, and Kaiser [27]. However, the connections between the two problems, and in particular the duality, have apparently never been stated precisely before.

(*q*) The duality theorem offers a powerful tool for developing more deeply the theory (as opposed to the computation) of Wiener filters, as mentioned in Remark (*i*). This will be published elsewhere [29].

## Applications

The power of the new approach to the Wiener problem, as expressed by Theorem 3, is most obvious when the data of the problem are given in numerical form. In that case, one simply performs the numerical computations required by (28–30). Results of such calculations, in some cases of practical engineering interest, will be published elsewhere.

When the answers are desired in closed analytic form, the iterations (28–30) may lead to very unwieldy expressions. In a few cases, $\mathbf{\Delta}^*$ and $\mathbf{\Phi}^*$ can be put into "closed form." Without discussing here how (if at all) such closed forms can be obtained, we now give two examples indicative of the type of results to be expected.

**Example 1.** Consider the problem mentioned under "Optimal Estimates." Let $x_1(t)$ be the signal and $x_2(t)$ the noise. We assume the model:

$$x_1(t+1) = \phi_{11}(t+1;\, t)x_1(t) + u_1(t)$$

$$x_2(t+1) = u_2(t)$$

$$y_1(t) = x_1(t) + x_2(t)$$

The specific data for which we desire a solution of the estimation problem are as follows:

**1** $t_1 = t + 1;\ t_0 = 0$
**2** $Ex_1^2(0) = 0$, i.e., $x_1(0) = 0$
**3** $Eu_1^2(t) = a^2$, $Eu^2(t) = b^2$, $Eu_1(t)\, u_2(t) = 0$ (for all $t$)
**4** $\phi_{11}(t+1;\, t) = \phi_{11} = \text{const.}$

A simple calculation shows that the following matrices satisfy the difference equations (28–30), for all $t \geq t_0$:

$$\mathbf{\Delta}^*(t) = \begin{bmatrix} \phi_{11}C(t) \\ 0 \end{bmatrix}$$

$$\mathbf{\Phi}^*(t+1;\, t) = \begin{bmatrix} \phi_{11}[1-C(t)] & 0 \\ 0 & 0 \end{bmatrix}$$

$$\mathbf{P}^*(t+1) = \begin{bmatrix} a^2 + \phi_{11}^2 b^2 C(t) & 0 \\ 0 & b^2 \end{bmatrix}$$

where
$$C(t+1) = 1 - \frac{b^2}{a^2 + b^2 + \phi_{11}^2 b^2 C(t)} \qquad t \geq 0 \qquad (38)$$

Since it was assumed that $x_1(0) = 0$, neither $x_1(1)$ nor $x_2(1)$ can be predicted from the measurement of $y_1(0)$. Hence the measurement at time $t = 0$ is useless, which shows that we should set $C(0) = 0$. This fact, with the iterations (38), completely determines the function $C(t)$. The nonlinear difference equation (38) plays the role of the Wiener-Hopf equation.

If $b^2/a^2 \ll 1$, then $C(t) \approx 1$ which is essentially pure prediction. If $b^2/a^2 \gg 1$, then $C(t) \approx 0$, and we depend mainly on $x_1^*(t|t-1)$ for the estimation of $x_1^*(t+1|t)$ and assign only very small weight to the measurement $y_1(t)$ ; this is what one would expect when the measured data are very noisy.

In any case, $x_2^*(t|t-1) = 0$ at all times; one cannot predict independent noise! This means that $\phi^*_{12}$ can be set equal to zero. The optimal predictor is a first-order dynamic system. See Remark (*j*).

To find the stationary Wiener filter, let $t = \infty$ on both sides of (38), solve the resulting quadratic equation in $C(\infty)$, etc.

**Example 2.** A number or particles leave the origin at time $t_0 = 0$ with random velocities; after $t = 0$, each particle moves with a constant (unknown) velocity. Suppose that the position of one of these particles is measured, the data being contaminated by stationary, additive, correlated noise. What is the optimal estimate of the position and velocity of the particle at the time of the last measurement?

Let $x_1(t)$ be the position and $x_2(t)$ the velocity of the particle; $x_3(t)$ is the noise. The problem is then represented by the model,

$$x_1(t+1) = x_1(t) + x_2(t)$$
$$x_2(t+1) = x_2(t)$$
$$x_3(t+1) = \phi_{33}(t+1;\, t)x_3(t) + u_3(t)$$
$$y_1(t) = x_1(t) + x_3(t)$$

and the additional conditions

**1** $t_1 = t;\ t_0 = 0$
**2** $Ex_1^2(0) = Ex_2(0) = 0$, $Ex_2^2(0) = a^2 > 0$;
**3** $Eu_3(t) = 0$, $Eu_3^2(t) = b^2$.
**4** $\phi_{33}(t+1;\, t) = \phi_{33} = \text{const.}$

According to Theorem 3, $\mathbf{x}^*(t|t)$ is calculated using the dynamic system (31).

First we solve the problem of predicting the position and velocity of the particle one step ahead. Simple considerations show that

$$\mathbf{P}^*(1) = \begin{bmatrix} a^2 & a^2 & 0 \\ a^2 & a^2 & 0 \\ 0 & 0 & b^2 \end{bmatrix} \quad \text{and} \quad \mathbf{\Delta}^*(0) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

It is then easy to check by substitution into equations (28–30) that

$$\mathbf{P}^*(t) = \frac{b^2}{C_1(t-1)} \times \begin{bmatrix} t^2 & t & -\phi_{33}t(t-1) \\ t & 1 & -\phi_{33}(t-1) \\ -\phi_{33}t(t-1) & -\phi_{33}(t-1) & \phi_{33}^2(t-1)^2 + C_1(t-1) \end{bmatrix}$$

is the correct expression for the covariance matrix of the prediction error $\tilde{\mathbf{x}}(t|t-1)$ for all $t \geq 1$, provided that we define

$$C_1(0) = b^2/a^2$$
$$C_1(t) = C_1(t-1) + [t - \phi_{33}(t-1)]^2, \quad t \geq 1$$

It is interesting to note that the results just obtained are valid also when $\phi_{33}$ depends on $t$. This is true also in Example 1. In conventional treatments of such problems there *seems* to be an essential difference between the cases of stationary and nonstationary noise. This misleading impression created by the conventional theory is due to the very special *methods* used in solving the Wiener-Hopf equation.

Introducing the abbreviation

$$C_2(0) = 0$$
$$C_2(t) = t - \phi_{33}(t-1), \quad t \geq 1$$

and observing that

$$\text{cov}\ \tilde{\mathbf{x}}(t+1|t) = \mathbf{P}^*(t+1)$$
$$= \mathbf{\Phi}(t+1;\, t)[\text{cov}\ \tilde{\mathbf{x}}(t|t)]\mathbf{\Phi}'(t+1;\, t) + \mathbf{Q}(t)$$

the matrices occurring in equation (31) and the covariance matrix of $\tilde{\mathbf{x}}(t|t)$ are found after simple calculations. We have, for all $t \geq 0$,

$$\mathbf{\Phi}(t;\, t+1)\mathbf{\Delta}^*(t) = \frac{1}{C_1(t)}\begin{bmatrix} tC_2(t) \\ C_2(t) \\ C_1(t) - tC_2(t) \end{bmatrix}$$

$$\mathbf{\Phi}(t;\, t+1)\mathbf{\Phi}^*(t+1;\, t)\mathbf{\Phi}(t+1;\, t)$$

$$= \frac{1}{C_1(t)}\begin{bmatrix} C_1(t) - tC_2(t) & C_1(t) - tC_3(t) & -\phi_{33}tC_2(t) \\ -C_2(t) & C_1(t) - C_2(t) & -\phi_{33}C_2(t) \\ -C_1(t) + tC_2(t) & -C_1(t) + tC_2(t) & +\phi_{33}tC_2(t) \end{bmatrix}$$

and

$$\operatorname{cov}\ \tilde{\mathbf{x}}(t|t) = E\,\tilde{\mathbf{x}}(t|t)\,\tilde{\mathbf{x}}'(t|t) = \frac{b^2}{C_1(t)}\begin{bmatrix} t^2 & t & -t^2 \\ t & 1 & -t \\ -t^2 & -t & t^2 \end{bmatrix}$$

To gain some insight into the behavior of this system, let us examine the limiting case $t \to \infty$ of a large number of observations. Then $C_1(t)$ obeys approximately the differential equation

$$dC_1(t)/dt \approx C_2^2(t) \qquad (t >> 1)$$

from which we find

$$C_1(t) \approx (1 - \phi_{33})^2 t^3/3 + \phi_{33}(1 - \phi_{33})t^2 + \phi_{33}^2 t + b^2/a^2 \qquad (t >> 1) \qquad (39)$$

Using (39), we get further,

$$\mathbf{\Phi}^{-1}\mathbf{\Phi}^*\mathbf{\Phi} \approx \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{\Phi}^{-1}\mathbf{\Delta}^* \approx \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad (t >> 1)$$

Thus as the number of observations becomes large, we depend almost exclusively on $x_1^*(t|t)$ and $x_2^*(t|t)$ to estimate $x_1^*(t+1|t+1)$ and $x_2^*(t+1|t+1)$. Current observations are used almost exclusively to estimate the noise

$$x_3^*(t|t) \approx y_1^*(t) - x_1^*(t|t) \qquad (t >> 1)$$

One would of course expect something like this since the problem is analogous to fitting a straight line to an increasing number of points.

As a second check on the reasonableness of the results given, observe that the case $t >> 1$ is essentially the same as prediction based on continuous observations. Setting $\phi_{33} = 0$, we have

$$E\,\tilde{x}_1^{\,2}(t\,|\,t) \approx \frac{a^2b^2t^2}{b^2 + a^2t^3/3} \qquad (t >> 1;\ \phi_{33} = 0)$$

which is identical with the result obtained by Shinbrot [11], Example 1, and Solodovnikov [14], Example 2, in their treatment of the Wiener problem in the finite-length, continuous-data case, using an approach entirely different from ours.

## Conclusions

This paper formulates and solves the Wiener problem from the "state" point of view. On the one hand, this leads to a very general treatment including cases which cause difficulties when attacked by other methods. On the other hand, the Wiener problem is shown to be closely connected with other problems in the theory of control. Much remains to be done to exploit these connections.

## References


1 N. Wiener, "The Extrapolation, Interpolation and Smoothing of Stationary Time Series," John Wiley & Sons, Inc., New York, N.Y., 1949.

2 L. A. Zadeh and J. R. Ragazzini, "An Extension of Wiener's Theory of Prediction," *Journal of Applied Physics*, vol. 21, 1950, pp. 645–655.

3 H. W. Bode and C. E. Shannon, "A Simplified Derivation of Linear Least-Squares Smoothing and Prediction Theory," *Proceedinqs IRE*, vol. 38, 1950, pp. 417–425.

4 R. C. Booton, "An Optimization Theory for Time-Varying Linear Systems With Nonstationary Statistical Inputs," *Proceedings IRE*, vol. 40, 1952, pp. 977–981.

5 J. H. Laning and R. H. Battin, "Random Processes in Automatic Control," McGraw-Hill Book Company, Inc., New York, N. Y., 1956.

6 W. B. Davenport, Jr., and W. L. Root, "An Introduction to the Theory of Random Signals and Noise," McGraw-Hill Book Company, Inc., New York, N. Y., 1958.

7 S. Darlington, "Linear Least-Squares Smoothing and Prediction, With Applications," *Bell System Tech. Journal*, vol. 37, 1958, pp.1221–1294.

8 G. Franklin, "The Optimum Synthesis of Sampled-Data Systems" Doctoral dissertation, Dept. of Elect. Engr., Columbia University, 1955.

9 A. B. Lees, "Interpolation and Extrapolation of Sampled Data," *Trans. IRE Prof. Grou.p on Information Theory*, IT-2, 1956, pp. 173–175.

10 R. C. Davis. "On the Theory of Prediction of Nonstationary Stochastic Processes," *Journal of Applied Physics*, vol. 23. 1952, pp. 1047–1053.

11 M. Shinbrot, "Optimization of Tirne-Varying Linear Systems With Nonstationary Inputs," TRANS. ASME, vol. 80, 1958. pp. 457–462.

12 M. Blum, "Recursion Formulas for Growing Memory Digital Filters," *Trans. IRE Prof. Group on Information Theory*. IT-4. 1958, pp. 24–30.

13 V. S. Pugachev, "The Use of Canonical Expansions of Random Functions in Determining an Optimum Linear System," *Automatics and Remote Control* (USSR), vol. 17, 1956, pp. 489–499; translation pp. 545–556.

14 V. V. Solodovnikov and A. M. Batkov. "On the Theory of Self-Optimizing Systems (in German and Russian)," Proc. Heidelberg Conference on Automatic Control, 1956. pp. 308–323.

15 J. L. Doob, "Stochastic Processes," John Wiley & Sons, Inc., New York, N. Y., 1955.

16 M. Loève, "Probability Theory," Van Nostrand Company, Inc., New York, N. Y., 1955.

17 R. E. Bellman, I. Glicksberg, and 0. A. Gross, "Some Aspects of the Mathematical Theory of Control Processes," RAND Report R-313, 1958, 244 pp.

18 R. E. Kalman and R. W. Koepcke, "Optimal Synthesis of Linear Sampling Control Systems Using Generalized Performance Indexes," TRANS. ASME, vol. 80, 1958, pp. 1820–1826.

19 J. E. Bertram, "Effect of Quantization in Sampled-Feedback Systems," *Trans. AlEE*, vol. 77, II, 1958, pp. 177–182.

20 R. E. Kalman and J. E. Bertram, "General Synthesis Procedure for Computer Control of Single and Multi-Loop Linear Systems" *Trans. AlEE*, vol. 77, II, 1958, pp. 602–609.

21 C. W. Merriam, III, "A Class of Optimum Control Systems," *Journal of the Franklin Institute*, vol. 267, 1959, pp. 267–281.

22 R. E. Kalman and J. E. Bertram, "A Unified Approach to the Theory of Sampling Systems," *Journal of the Franklin Institute*, vol. 267, 1959, pp. 405–436.

23 R. E. Kalman and R. W. Koepcke, "The Role of Digital Computers in the Dynamic Optimization of Chemical Reactors," Proc. Western Joint Computer Conference, 1959, pp. 107–116.

24 R. E. Kalman, "Dynamic Optimization of Linear Control Systems, I. Theory," to appear.

25 S. Sherman, "Non-Mean-Square Error Criteria," *Trans. IRE Prof. Group on Information Theory*, IT-4, 1958, pp. 125–126.

26 V. S. Pugachev, "On a Possible General Solution of the Problem of Determining Optimum Dynamic Systems," *Automatics and Remote Control* (USSR), vol. 17, 1956, pp. 585–589.

27 G. C. Newton, Jr., L. A. Gould, and J. F. Kaiser, "Analytical Design of Linear Feedback Controls," John Wiley & Sons, Inc., New York, N. Y., 1957.

28 0. J. M. Smith, "Feedback Control Systems," McGraw-Hill Book Company, Inc., New York, N. Y., 1958.

29 R. E. Kalman, "On the General Theory of Control Systems," Proceedings First International Conference on Automatic Control, Moscow, USSR, 1960.


Transactions of the ASME–Journal of Basic Engineering, 82 (Series D): 35-45. 

# APPENDIX
# RANDOM PROCESSES: BASIC CONCEPTS

For convenience of the reader, we review here some elementary definitions and facts about probability and random processes. Everything is presented with the utmost possible simplicity; for greater depth and breadth, consult Laning and Battin [5] or Doob [15].

A *random variable* is a function whose values depend on the outcome of a chance event. The *values* of a random variable may be any convenient mathematical entities; real or complex numbers, vectors, etc. For simplicity, we shall consider here only real-valued random variables, but this is no real restriction. Random variables will be denoted by $x, y,$ ... and their values by $\xi$, $\eta$, …. Sums, products, and functions of random variables are also random variables.

A random variable $x$ can be explicitly defined by stating the probability that $x$ is less than or equal to some real constant $\xi$. This is expressed symbolically by writing

$$Pr(x \le \xi) = F_x(\xi);\ F_x(-\infty) = 0,\ F_x(+\infty) = 1$$

$F_x(\xi)$ is called the *probability distribution function* of the random variable $x$. When $F_x(\xi)$ is differentiable with respect to $\xi$, then $f_x(\xi) = dF_x(\xi)/d\xi$ is called the *probability density function* of $x$.

The *expected value* (*mathematical expectation, statistical average, ensemble average, mean*, etc., are commonly used synonyms) of any nonrandom function $g(x)$ of a random variable $x$ is defined by

$$Eg(x) = E[g(x)] = \int_{-\infty}^{\infty} g(\xi)dF_x(\xi) = \int_{-\infty}^{\infty} g(\xi)\, f_x(\xi)d\xi \qquad (40)$$

As indicated, it is often convenient to omit the brackets after the symbol $E$. A sequence of random variables (finite or infinite)

$$\{x(t)\} = \ldots, x(-1), x(0), x(1), \ldots \qquad (41)$$

is called a *discrete* (or *discrete-parameter*) *random* (or *stochastic*) *process.* One particular set of observed values of the random process (41)

$$\ldots, \xi(-1), \xi(0), \xi(1), \ldots$$

is called a *realization* (or a *sample function)* of the process. Intuitively, a random process is simply a set of random variables which are indexed in such a way as to bring the notion of time into the picture.

A random process is *uncorrelated* if

$$Ex(t)x(s) = Ex(t)Ex(s) \qquad (t \ne s)$$

If, furthermore,

$$Ex(t)x(s) = 0 \qquad (t \ne s)$$

then the random process is *orthogonal.* Any uncorrelated random process can be changed into orthogonal random process by replacing $x(t)$ by $x'(t) = x(t) - Ex(t)$ since then

$$Ex'(t)x'(s) = E[x(t) - Ex(t)]\cdot[x(s) - Ex(s)]$$
$$= Ex(t)x(s) - Ex(t)Ex(s) = 0$$

It is useful to remember that, if a random process is orthogonal, then

$$E[x(t_1) + x(t_2) + \ldots]^2 = Ex^2(t_1) + Ex^2(t_2) + \ldots \quad (t_1 \ne t_2 \ne \ldots)$$

If $\mathbf{x}$ is a vector-valued random variable with components $x_1, \ldots, x_n$ (which are of course random variables), the matrix

$$[E(x_i - Ex_i)(x_j - Ex_j)] = E(\mathbf{x} - E\mathbf{x})(\mathbf{x}' - E\mathbf{x}')$$
$$= \operatorname{cov} \mathbf{x} \qquad (42)$$

is called the *covariance matrix of x.*

A random process may be specified explicitly by stating the probability of simultaneous occurrence of any finite number of events of the type

$x(t_1) \le \xi_1, \ldots, x(t_n) \le \xi_n$; $(t_1 \ne \ldots \ne t_n)$, i.e.,

$$Pr[(x(t_1) \le \xi_1, \ldots, x(t_n) \le \xi_n)] = F_{x(t_1),\ \ldots,\ x(t_n)}(\xi_1, \ldots, \xi_n) \qquad (43)$$

where $F_{x(t_1),\ \ldots,\ x(t_n)}$ is called the *joint probability distribution function* of the random variables $x(t_1), \ldots, x(t_n)$. The *joint probability density function* is then

$$f_{x(t_1),\ \ldots,\ x(t_n)}(\xi_1, \ldots, \xi_n) = \partial^n F_{n(t_1),\ \ldots,\ x(t_n)}/\partial\xi_1, \ldots, \partial\xi_n$$

provided the required derivatives exist. The expected value $Eg[x(t_1), \ldots, x(t_n)]$ of any nonrandom function of $n$ random variables is defined by an $n$-fold integral analogous to (40).

A random process is *independent* if for any finite $t_1 \ne \ldots \ne t_n$, (43) is equal to the product of the first-order distributions

$$Pr[x(t_1) \le \xi_1] \ldots Pr[x(t_n) \le \xi_n]$$

If a set of random variables is independent, then they are obviously also uncorrelated. The converse is not true in general. For a set of more than 2 random variables to be independent, it is not sufficient that any pair of random variables be independent.

Frequently it is of interest to consider the probability distribution of a random variable $x(t_{n+1})$ of a random process given the actual values $\xi(t_1), \ldots, \xi(t_n)$ with which the random variables $x(t_1), \ldots, x(t_n)$ have occurred. This is denoted by

$$Pr[x(t_{n+1}) \le \xi_{n+1} | x(t_1) = \xi_1, \ldots, x(t_n) = \xi_n]$$

$$= \frac{\int_{-\infty}^{\xi_{n+1}} f_{x(t_1),\ldots,x(t_{n+1})}(\xi_1,\ldots,\xi_{n+1})d\xi_{n+1}}{f_{x(t_1),\ldots,x(t_n)}(\xi_1,\ldots,\xi_n)} \qquad (44)$$

which is called the *conditional probability distribution function* of $x(t_{n+1})$ given $x(t_1), \ldots, x(t_n)$. The *conditional expectation*

$$E\{g[x(t_{n+1})] | x(t_1), \ldots, x(t_n)\}$$

is defined analogously to (40). The conditional expectation is a random variable; it follows that

$$E[E\{g[x(t_{n+1})] | x(t_1), \ldots, x(t_n)\}] = E\{g[x(t_{n+1})]\}$$

In all cases of interest in this paper, integrals of the type (40) or (44) need never be evaluated explicitly, only the *concept* of the expected value is needed.

A random variable $x$ is *gaussian* (or *normally distributed)* if

$$f_x(\xi) = \frac{1}{[2\pi E(x - Ex)^2]^{1/2}} \exp\left[-\frac{1}{2}\frac{(\xi - Ex)^2}{E(x - Ex)^2}\right]$$

which is the well-known bell-shaped curve. Similarly, a random vector $\mathbf{x}$ is *gaussian* if

$$f_x(\boldsymbol{\xi}) = \frac{1}{(2\pi)^{n/2}(det\ \mathbf{C})^{1/2}} \exp\left[-\frac{1}{2}(\boldsymbol{\xi} - E\mathbf{x})'\mathbf{C}^{-1}(\boldsymbol{\xi} - E\mathbf{x})\right]$$

where $\mathbf{C}^{-1}$ is the inverse of the covariance matrix (42) of $\mathbf{x}$. A *gaussian random process* is defined similarly.

The importance of gaussian random variables and processes is largely due to the following facts:

**Theorem 5.** (*A*) *Linear functions* (*and therefore conditional expectations*) *on a gaussian random process are gaussian random variables.*

(*B*) *Orthogonal gaussian random variables are independent.*

(*C*) *Given any random process with means* $Ex(t)$ *and covariances* $Ex(t)x(s)$*, there exists a unique gaussian random process with the same means and covariances.*

Transactions of the ASME–Journal of Basic Engineering, 82 (Series D): 35-45. Copyright © 1960 by ASME

Explanation of this transcription, John Lukesh, 20 January 2002.

Using a photo copy of R. E. Kalman's 1960 paper from an original of the ASME "Journal of Basic Engineering", March 1960 issue, I did my best to make an accurate version of this rather significant piece, in an up-to-date computer file format. For this I was able to choose page formatting and type font spacings that resulted in a document that is a close match to the original. (All pages start and stop at about the same point, for example; even most individual lines of text do.) I used a recent version of Word for Windows and a recent Hewlett Packard scanner with OCR (optical character recognition) software. The OCR software is very good on plain text, even distinguishing between italic versus regular characters quite reliably, but it does not do well with subscripts, superscripts, and special fonts, which were quite prevalent in the original paper. And I found there was no point in trying to work from the OCR results for equations. A lot of manual labor was involved.

Since I wanted to make a faithful reproduction of the original, I did not make any changes to correct (what I believed were) mistakes in it. For example, equation (32) has a $\mathbf{P}^*(t)\mathbf{M}(t)$ product that should be reversed, I think. I left this, and some other things that I thought were mistakes in the original, as is. (I didn't find very many other problems with the original.) There may, *of course*, be problems with my transcription. The plain text OCR results, which didn't require much editing, are pretty accurate I think. But the subscripts etc and the equations which I copied essentially manually, are suspect. I've reviewed the resulting document quite carefully, several times finding mistakes in what I did each time. The last time there were five, four cosmetic and one fairly inconsequential. There are probably more. I would be very pleased to know about it if any reader of this finds some of them; jlukesh@deltanet.com.