"Clustering and the enabling technology for large data set supplied by Hadoop"

Corso di Laurea Magistrale in Informatica

William Lucia - Matricola 709501

Contents

Chapter 1

Clustering

1.1 Introduction

Clustering is the unsupervised classification of patterns (observations, data items, or feature vectors) into groups (clusters). The clustering problem has been addressed in many contexts and by researcher in many disciplines; this reflect its broad appeal and usefulness as one of the step in exploratory data analysis.

Data analysis underlies many computing applications, either in a design phase or as part of their on-line operations. Data analysis procedures can be dichotomized as either exploratory or confirmatory, based on the availability of appropriate models for the data source, but a key element in both types of procedures (whether for hypothesis formation or decision-making) is the grouping, or classification of measurement based on their

(i) goodness-of-fit to a postulated model, or

(ii) natural groupings (clustering) revealed through analysis.

Cluster analysis is the organizations of a collection of patterns (usually represented as a vector of measurements, or a point in a multidimensional space) into clusters based on similarity. Intuitively, patterns within a valid cluster are are more similar to each other than they are to a pattern belonging to a different cluster.

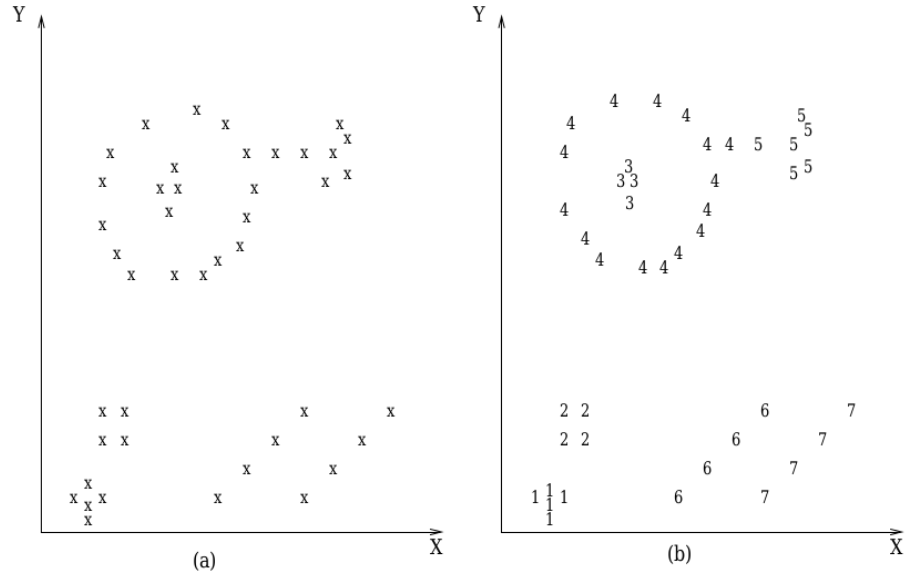

Figure 1.1: Data clustering.

The input patterns are show in Figure 1.1 (a), and the desired clusters are show in Figure 1.1 (b) . Point belonging to the same cluster are given the same label.

It is important to understand the difference between clustering (unsupervised classification) and discriminant analysis (supervised classification). In supervised classifications, we are provided with a collection of labeled (preclassified) patterns; the problem is to label a newly encountered, yet unlabeled, pattern. Typically, the given labeled (training) patterns are used to learn the descriptions of classes which in turn are used to label a new pattern.

In the case of clustering, the problem is to group a given collection of unlabeled patterns into meaningful clusters. Also in this case labels are associated with clusters, but these category labels are data driven; that is, they are obtained solely from the data.

Clustering is useful in several exploratory pattern-analysis, grouping, decision-making, and machine-learning situations, including data mining, document retrieval, image segmentation,and pattern classification. In many such problem, there is little prior information (e.g, statistical models) available about the data, and it must make a few assumptions about the data as possible. It is under this restrictions that clustering methodology is particularly appropriate for the exploration of interrelationship among the data points to make assessment of their structure.

1.2 Components of a clustering task

Typical pattern clustering activity involves the following steps [4]:

1. pattern representation (optionally including feature extraction and/or selection);

2. definition of a pattern proximity measure appropriate to the data domain;

3. clustering or grouping;

4. data abstraction (if needed);

5. assessment of output.

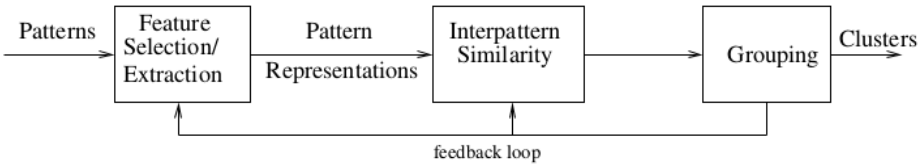

Figure 1.2: Stages in clustering.

Figure 1.2 depicts a typical sequencing of the first three of these steps, including a feedback path where the grouping process output could affect subsequent feature extraction and similarity computations.

Pattern representations refers to the numbers of classes, the number of available patterns, and the number, type, and scale of the features available to the clustering algorithm. *Feature selection* is the process of identifyng the most effective subset of the original features to use in clustering. *Feature extraction* is the use of one or more transformations of the input features to produce new salient features. Either or both of these techniques can be used to obtain an appropriate set of features to use in clustering.

Pattern proximity is usually measured by a distance function defined on pairs of patterns. A simple distance measure like Euclidean distance can often be used to reflect dissimilarity between two patterns, whereas other similarity measures can be used to characterize the conceptual similarity between patterns.

The *Grouping*step can be performed in a number of ways. The output clustering can be hard (a partition of the data into groups) or fuzzy (where each pattern ha a variable degree of membership in each of the output clusters). Hierarchical clustering algorithms produces a nested series of partitions based on a criterion for merging or splitting clusters based on similarity. Partitional clustering algorithms identify the partition that optimizes a clustering criterion.

Data abstraction is the process of extracting a simple and compact representation of a date set. Here, simplicity is either from the perspective of automatic analysis (so that a machine can perform further processing efficiently) or it is human-oriented (so that the representation obtained is easy to comprehend and intuitively appealing).

How is the output of a clustering algorithm evaluated?

All clustering algorithms will, when presented with data, produce clusters, regardless of whether the data contain clusters or not. If the data does contain clusters, some clustering algorithms may obtain 'better clusters than others. The assessment of a clustering procedure's output, then, has several facets. One is actually an assessment of the data domain rather than the clustering algorithm itself, data which do not contain clusters should not be processed by a clustering algorithm.

Cluster validity analysis, by contrast, is the assessment of a clustering procedure's output. Often this analysis uses a specific criterion of optimality; however, these criteria are usually arrived at subjectively. Validity assessments are objective and are performed to determine whether the output is meaningful. A clustering structure is valid if it cannot reasonably have occurred by chance or as an artifact of a clustering algorithm. When statistical approaches to clustering are used, validation is accomplished by carefully applying statistical methods and testing hypotheses. There are three types of validation studies.

(i) An external assessment of validity compares the recovered structure to an a *priori* structure.

(ii) An internal examination of validity tries to determine if the structure is intrinsically appropriate for the data.

(iii) A relative test compares two structures and measures their relative merit.

1.3 Definitions and notation

This section shows the terms and notation used for explain representation of patterns and clustering techniques.

- A *pattern* (or feature vector, observation, or datum) x is a single data item used by the clustering algorithm. It typically consists of a vector of d measurements: $x = (x_1, ..., x_d)$.
- The individual scalar components x_i of a pattern x are called *features* (or attributes).
- A *pattern set* is denoted $\xi = \{x_1, ..., x_n\}$. The *i-th* pattern in ξ is denoted $x = (x_{i,1}, ..., x_{i,d})$. In many cases a pattern set to be clustered is viewed as an $n \times d$ *pattern matrix*.
- A *class* refers to a state of nature that governs the pattern generation process in some cases. More concretely, a class can be viewed as a source of patterns whose distribution in feature space is governed by a probability density specific to the class. Clustering techniques attempt to group patterns so that the classes thereby obtained reflect the different pattern generation processes represented in the pattern set.
- *Hard* clustering techniques assign a *class label* l_i to each patterns x_i, identifying its class. The set of all labels for a pattern set ξ is $L = (l_1, ..., l_n)$ with $l_i \in \{1, ..., k\}$ where k is the numbers of clusters.
- *Fuzzy* clustering procedures assign to each input pattern x_i a fractional degree of membership f_{ij} in each output cluster j .
- A *distance measure* (a specialization of a proximity measure) is a metric on the feature space used to quantify the similarity of patterns.

1.4 Patterns representation

There are no theoretical guidelines that suggest the appropriate patterns and features to use in a specific situation. Indeed, the pattern generation process is often not directly controllable; the user's role in the pattern representation process is to gather facts and conjectures about the data, optionally perform feature selection and extraction, and design the subsequent elements of the clustering system. it is conveniently assumed that the pattern representation is available prior to clustering. Nonetheless, a careful investigation of the available features and any available transformations can yield significantly improved clustering results. A good pattern representation can often yield a simple and easily understood clustering; a poor pattern representation may yield a complex clustering whose true structure is difficult or impossible to discern.

We see an example:

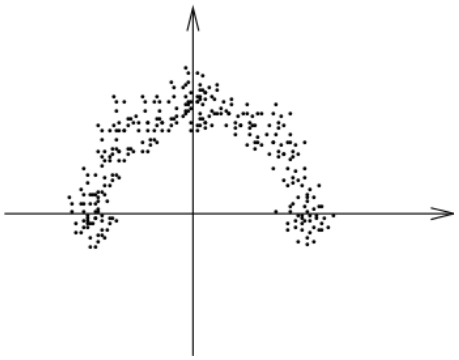

Figure 1.3: A curvilinear cluster.

Figure 1.3 shows a simple example. The points in this 2D feature space are arranged in a curvilinear cluster of approximately constant distance from the origin. If one chooses Cartesian coordinates to represent the patterns, many clustering algorithms would be likely to fragment the cluster into two or more clusters, since it is not compact. If, however, one uses a polar coordinate representation for the clusters, the radius coordinate exhibits tight clustering and a one-cluster solution is likely to be easily obtained.

A pattern can measure either a physical object (e.g., a chair) or an abstract notion (e.g., a style of writing). As noted above, patterns are represented conventionally as multidimensional vectors, where each dimension is a single feature. These features can be either quantitative or qualitative. For example, if *weight* and *color* are the two features used, then (20, black) is the representation of a black object with 20 units of weight.

The features can be subdivided into the following types [2]:

- Quantitative features: e.g.
  (a) continuous values (e.g., weight);
  (b) discrete values (e.g., the number of computers);
  (c) interval values (e.g., the duration of an event).

- Qualitative features:
  (a) nominal or unordered (e.g., color);
  (b) ordinal (e.g., qualitative evaluations of temperature: "cool" or "hot").

Quantitative features can be measured on a ratio scale (with a meaningful reference value, such as temperature), or on nominal or ordinal scales. One can also use structured features which are represented as trees, where the parent node represents a generalization of its child nodes. For example, a parent node "vehicle" may be a generalization of children labeled "cars," "buses," "trucks," and "motorcycles." Further, the node "cars" could be a generalization of cars of the type "Toyota," "Ford," "Benz," etc.

It is often valuable to isolate only the most descriptive and discriminatory features in the input set, and utilize those features exclusively in subsequent analysis. Feature selection techniques identify a subset of the existing features for subsequent use, while feature extraction techniques compute new features from the original set. In either case, the goal is to improve classification performance and/or computational efficiency. Feature selection is a well-explored topic in statistical pattern recognition [1]; however, in a clustering context (i.e., lacking class labels for patterns), the feature selection process is of necessity ad hoc, and might involve a trial-and-error process where various subsets of features are selected, the resulting patterns clustered, and the output evaluated using a validity index.

In contrast, some of the popular feature extraction processes (e.g., principal components analysis) do not depend on labeled data and can be used directly.

1.5 Similarity measures

Since similarity is fundamental to the definition of a cluster, a measure of the similarity between two patterns drawn from the same feature space is essential to most clustering procedures. Because of the variety of feature types and scales, the distance measure (or measures) must be chosen carefully.

It is most common to calculate the *dissimilarity* between two patterns using a distance measure defined on the feature space. The most popular metric for continuous features is the *Euclidean distance*:

$$d_2(X_i, X_j) = (\sum_{k=1}^{d}(X_{i,k} - X_{j,k})^2)^{1/2} = \| x_i - x_j \|_2$$

Which is a special case (p = 2) of the Minkowski metric:

$$d_p(X_i, X_j) = (\sum_{k=1}^{d}(X_{i,k} - X_{j,k})^p)^{1/p} = \| x_i - x_j \|_p$$

The Euclidean distance has an intuitive appeal as it is commonly used to evaluate the proximity of objects in two or three-dimensional space. It works well when a data set has "compact" or "isolated" clusters. The drawback to direct use of the Minkowski metrics is the tendency of the largest-scaled feature to dominate the others. Solutions to this problem include normalization of the continuous features or other weighting schemes. Linear correlation among

features can also distort distance measures; this distortion can be alleviated by applying a whitening transformation to the data or by using the *squared Mahalanobis distance*:

$$d_M(x_i, x_j) = (x_i - x_j)\Sigma^{-1}(x_i - x_j)^T$$

where the patterns x_i and x_j are assumed to be row vectors, and Σ is the sample covariance matrix of the patterns or the known covariance matrix of the pattern generation process; $d_M(\cdot, \cdot)$ assigns different weights to different features based on their variances and pairwise linear correlations. Here, it is implicitly assumed that class conditional densities are unimodal and characterized by multidimensional spread, i.e., that the densities are multivariate Gaussian.

Some clustering algorithms work on a matrix of proximity values instead of on the original pattern set. It is useful in such situations to precompute all the $n(n-1)/2$ pairwise distance values for the n patterns and store them in a symmetric matrix.

There are some distance measures reported in the literature that take into account the effect of surrounding or neighboring points. These surrounding points are called *context* in [8] . The similarity between two points x_i and x_j, given this context, is given by

$$s(x_i, x_j) = f(x_i, x_j, \xi)$$

where ξ is the context (the set of surrounding points). One metric defined using context is the *mutual neighbor distance* (MND), proposed in [3], which is given by

$$MND(x_i, x_j) = NN(x_i, x_j) + NN(x_j, x_i)$$

where $MND(x_i, x_j)$ is the neighbor number of x j with respect to xi.

Figures 1.4 and 1.5 give an example.

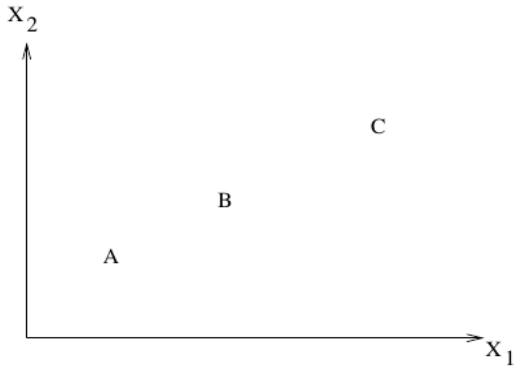

Figure 1.4: A and Bare more similar than A and C.

In Figure 1.4, the nearest neighbor of A is B, and B's nearest neighbor is A. So, $NN(A, B) = NN(B, A) = 1$ and the MND between A and B is 2. However, $NN(B, C) = 1$ but $NN(C, B) = 2$, and therefore $MND(B, C) = 3$.

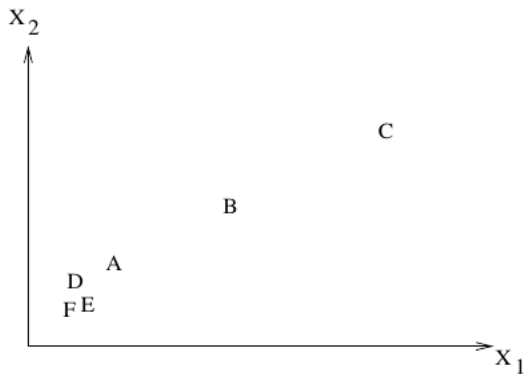

Figure 1.5: After a change in context, B and C are more similar than B and A.

Figure 1.5 was obtained from Figure 1.4 by adding three new points D, E, and F. Now $MND(B, C) = 3$ (as before), but $MND(A, B) = 5$. The MND between A and B has increased by introducing additional points, even though A and B have not moved. In fact, after a change in context, B and C are more similar than B and A.

1.6 Clustering Techniques

Different approaches to clustering data can be described with the help of the hierarchy shown in Figure 1.6

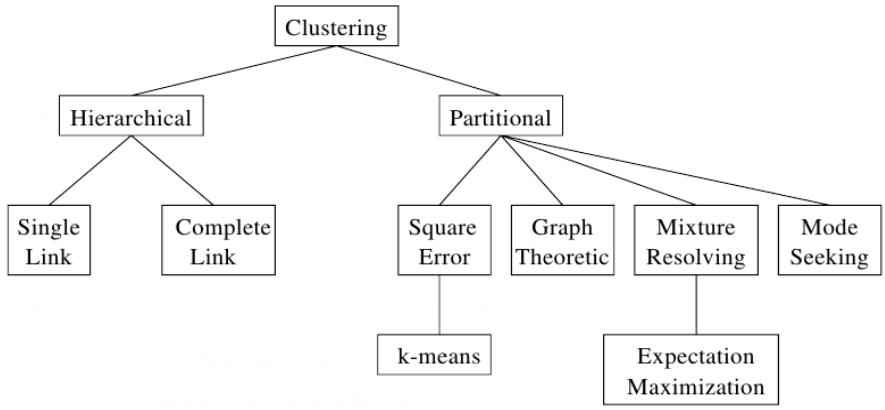

Figure 1.6: A taxonomy of clustering approaches

At the top level, there is a distinction between hierarchical and partitional approaches (hierarchical methods produce a nested series of partitions, while partitional methods produce only one).

The taxonomy shown in Figure 1.6 must be supplemented by a discussion of cross-cutting issues that may affect all of the different ap- proaches regardless of their placement in the taxonomy.

- *Agglomerative* vs. *divisive*: this aspect relates to algorithmic structure

and operation. An agglomerative approach begins with each pattern in a distinct (singleton) cluster, and successively merges clusters together until a *stopping criterion* is satisfied. A divisive method begins with all patterns in a single cluster and performs splitting until a stopping criterion is met.

- *Monothetic* vs. *polythetic* : this aspect relates to the sequential or simultaneous use of features in the clustering process. Most algorithms are polythetic; that is, all features enter into the computation of distances between patterns, and decisions are based on those distances.
- *Hard* vs. *fuzzy* : a hard clustering algorithm allocates each pattern to a single cluster during its operation and in its output. A fuzzy clustering method assigns degrees of membership in several clusters to each input pattern. A fuzzy clustering can be converted to a hard clustering by assigning each pattern to the cluster with the largest measure of membership.
- *Deterministic* vs. *stochastic*: this issue is most relevant to partitional approaches designed to optimize a squared error function. This optimization can be accomplished using traditional techniques or through a random search of the state space consisting of all possible labelings.
- *Incremental* vs. *non-incremental*: this issue arises when the pattern set to be clustered is large, and constraints on execution time or memory space affect the architecture of the algorithm.

1.6.1 Hierarchical Clustering

Given a set of N items to be clustered, and an $N*N$ distance (or similarity) matrix, the basic process of hierarchical clustering [5] is this:

1. Start by assigning each item to a cluster, so that if you have N items, you now have N clusters, each containing just one item. Let the distances (similarities) between the clusters the same as the distances (similarities) between the items they contain.
2. Find the closest (most similar) pair of clusters and merge them into a single cluster, so that now you have one cluster less.
3. Compute distances (similarities) between the new cluster and each of the old clusters.
4. Repeat steps 2 and 3 until all items are clustered into a single cluster of size N.

Of course there is no point in having all the N items grouped in a single cluster but, once you have got the complete hierarchical tree, if you want k clusters you just have to cut the k-1 longest links.

Step 3 can be done in different ways, which is what distinguishes *single-linkage* from *complete-linkage* and *average-linkage* clustering.

In *single-linkage* clustering (also called the connectedness or minimum method), we consider the distance between one cluster and another cluster to be equal to the shortest distance from any member of one cluster to any member of the other cluster. If the data consist of similarities, we consider the

similarity between one cluster and another cluster to be equal to the greatest similarity from any member of one cluster to any member of the other cluster.

In *complete-linkage* clustering (also called the diameter or maximum method), we consider the distance between one cluster and another cluster to be equal to the greatest distance from any member of one cluster to any member of the other cluster.

In *average-linkage* clustering, we consider the distance between one cluster and another cluster to be equal to the *average* distance from any member of one cluster to any member of the other cluster.

This kind of hierarchical clustering is called *agglomerative* because it merges clusters iteratively. There is also a *divisive* hierarchical clustering which does the reverse by starting with all objects in one cluster and subdividing them into smaller pieces. Divisive methods are not generally available, and rarely have been applied.

Single-Linkage Clustering: The Algorithm

Let's now take a deeper look at how hierarchical clustering algorithm works in the case of *single-linkage* clustering. The algorithm is an agglomerative scheme that erases rows and columns in the proximity matrix as old clusters are merged into new ones.

The $N * N$ proximity matrix is $D = [d(i, j)]$. The clusterings are assigned sequence numbers $0, 1, ..., (n-1)$ and $L(k)$ is the level of the $k-th$ clustering. A cluster with sequence number m is denoted (m) and the proximity between clusters (r) and (s) is denoted $d[(r), (s)]$.

The algorithm is composed of the following steps:

1. begin with the disjoint clustering having level $L(0) = 0$ and sequence number $m = 0$;

2. find the least dissimilar pair of clusters in the current clustering, say pair $(r), (s)$, according to

   $d[(r), (s)] = min d[(i), (j)]$ where the minimum is over all pairs of clusters in the current clustering.

3. increment the sequence number : $m = m + 1$. Merge clusters (r) and (s) into a single cluster to form the next clustering m. Set the level of this clustering to $L(m) = d[(r), (s)]$;

4. update the proximity matrix, D, by deleting the rows and columns corresponding to clusters (r) and (s) and adding a row and column corresponding to the newly formed cluster. The proximity between the new cluster, denoted (r, s) and old cluster (k) is defined in this way: $d[(k), (r, s)] = min d[(k), (r)], d[(k), (s)]$;

5. if all objects are in one cluster, stop. Else, go to step 2.

An Example [7]

Let's now see a simple example: a hierarchical clustering of distances in kilometers between some Italian cities. The method used is *single-linkage*.

Input distance matrix(L = 0 for all the clusters):

	BA	FI	MI	NA	RM	TO
BA	0	662	877	255	412	996
FI	662	0	295	468	268	400
MI	877	295	0	754	564	138
NA	255	468	754	0	219	869
RM	412	268	564	219	0	669
TO	996	400	138	869	669	0

The nearest pair of cities is MI and TO, at distance 138. These are merged into a single cluster called "MI/TO". The level of the new cluster is $L(MI/TO) = 138$ and the new sequence number is $m = 1$. Then we compute the distance from this new compound object to all other objects. In single link clustering the rule is that the distance from the compound object to another object is equal to the shortest distance from any member of the cluster to the outside object. So the distance from "MI/TO" to RM is chosen to be 564, which is the distance from MI to RM, and so on.

After merging MI with TO we obtain the following matrix:

	BA	FI	MI/TO	NA	RM
BA	0	662	877	255	412
FI	662	0	295	468	268
MI/TO	877	295	0	754	564
NA	255	468	754	0	219
RM	412	268	564	219	0

Because min $d(i,j) = d(NA, RM) = 219$ then we merge NA and RM into a new cluster called NA/RM. The level of new cluster is $L(NA/RM) = 219$ and the new sequence number is $m = 2$.

	BA	FI	MI/TO	NA/RM
BA	0	662	877	255
FI	662	0	295	268
MI/TO	877	295	0	564
NA/RM	255	268	564	0

Because min $d(i,j) = d(BA, NA/RM) = 255$ then we merge BA and NA/RM into a new cluster called BA/NA/RM. The level of new cluster is $L(BA/NA/RM) = 255$ and the new sequence number is $m = 3$.

	BA/NA/RM	FI	MI/TO
BA/NA/RM	0	268	564
FI	268	0	295
MI/TO	564	295	0

Because $\min d(i,j) = d(BA/NA/RM, FI) = 268$ then we merge BA/NA/RM and FI into a new cluster called BA/FI/NA/RM. The level of new cluster is $L(BA/FI/NA/RM) = 268$ and the new sequence number is $m = 4$.

	BA/FI/NA/RM	MI/TO
BA/FI/NA/RM	0	295
MI/TO	295	0

Finally, we merge the last two clusters at level 295.

The process is summarized by the following hierarchical tree:

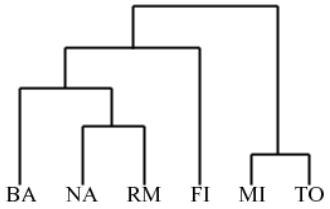

Figure 1.7: The dendogram obtained using the single-linkage algorithm.

Problems

The main weaknesses of agglomerative clustering methods are:

1. they do not scale well: time complexity of at least O(n2), where n is the number of total objects;
2. they can never undo what was done previously.

1.6.2 Partitional Algorithms

A partitional clustering algorithm obtains a single partition of the data instead of a clustering structure, such as the dendrogram produced by a hierarchical technique. Partitional methods have advantages in applications involving large data sets for which the construction of a dendrogram is computationally prohibitive.

A problem accompanying the use of a partitional algorithm is the choice of the number of desired output clusters. The partitional techniques usually produce clusters by optimizing a criterion function defined either locally (on a subset of the patterns) or globally (defined over all of the patterns). Combinatorial search of the set of possible labelings for an optimum value of a criterion is clearly computationally prohibitive. In practice, therefore, the algorithm is typically run multiple times with different starting states, and the best configuration obtained from all of the runs is used as the output clustering.

Squared Error Algorithms

The most intuitive and frequently used criterion function in partitional clustering techniques is the squared error criterion, which tends to work well with isolated and compact clusters. The squared error for a clustering ψ of a pattern set ξ (containing K clusters) is

$$e^2(\xi, \psi) = \sum_{j=1}^{K} \sum_{i=1}^{n_j} \parallel x_i^{(j)} - c_j \parallel^2$$

, where $x_i^{(j)}$ is the i^{th} pattern belonging to the j^{th} cluster and c_j is the centroid of the j^{th} cluster.

The *k -means* is the simplest and most commonly used algorithm employing a squared error criterion. It starts with a random initial partition and keeps

reassigning the patterns to clusters based on the similarity between the pattern and the cluster centers until a convergence criterion is met (e.g., there is no reassignment of any pattern from one cluster to another, or the squared error ceases to decrease significantly after some number of iterations).

The k -means algorithm is popular because it is easy to implement, and its time complexity is $O(n)$, where n is the number of patterns. A major problem with this algorithm is that it is sensitive to the selection of the initial partition and may converge to a local minimum of the criterion function value if the initial partition is not properly chosen.

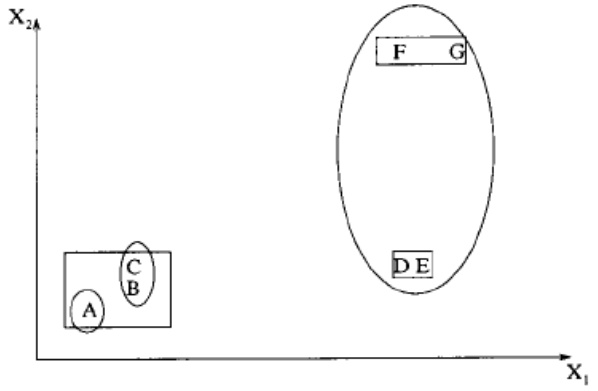

Figure 1.8: The k-means algorithm is sensitive to the initial partition

Figure 1.8 shows seven two-dimensional patterns. If we start with patterns A, B, and C as the initial means around which the three clusters are built, then we end up with the partition A, B, C, D, E, F, G shown by ellipses. The squared error criterion value is much larger for this partition than for the best partition A, B, C, D, E, F, G shown by rectangles, which yields the global minimum value of the squared error criterion function for a clustering containing three clusters. The correct three-cluster solution is obtained by choosing, for example, A, D, and F as the initial cluster means.

Squared Error Clustering Method

(1) Select an initial partition of the patterns with a fixed number of clusters and cluster centers;

(2) assign each pattern to its closest cluster center and compute the new cluster centers as the centroids of the clusters. Repeat this step until convergence is achieved, i.e., until the cluster membership is stable;

(3) merge and split clusters based on some heuristic information, optionally repeating step 2.

K-Means Clustering Algorithm

(1) Choose k cluster centers to coincide with k randomly-chosen patterns or k randomly defined points inside the hypervolume containing the pattern set.

(2) assign each pattern to the closest cluster center,

(3) recompute the cluster centers using the current cluster memberships;

(4) if a convergence criterion is not met, go to step 2. Typical convergence criteria are: no (or minimal) reassignment of patterns to new cluster centers, or minimal decrease in squared error.

Several variants of the k -means algorithm have been reported in the literature. Some of them attempt to select a good initial partition so that the algorithm is more likely to find the global minimum value. Another variation is to permit splitting and merging of the resulting clusters. Typically, a cluster is split when its variance is above a pre-specified threshold, and two clusters are merged when the distance between their centroids is below another pre-specified threshold. Using this variant, it is possible to obtain the optimal partition starting from any arbitrary initial partition, provided proper threshold values are specified.

Nearest Neighbor Clustering

Since proximity plays a key role in our intuitive notion of a cluster, *nearest neighbor* distances can serve as the basis of clustering procedures. An iterative procedure was proposed [6]; it assigns each unlabeled pattern to the cluster of its nearest labeled neighbor pattern, provided the distance to that labeled neighbor is below a threshold. The process continues until all patterns are labeled or no additional labelings occur. The mutual neighborhood value (described earlier in the context of distance computation) can also be used to grow clusters from near neighbors.

Chapter 2

Finding and evaluating community structure in networks

We now analyze an algorithm for discovering community structure in networks, natural divisions of network nodes into densely connected subgroups. This algorithm has two definitive features:

(i) It involves iterative removal of edges from the network to split it into communities, the edges removed being identified using one of a number of possible *"betweenness"* measures;

(ii) these measures are, crucially, recalculated after each removal.

We also study a measure for the strength of the community structure found by this algorithm, which gives us an objective metric for choosing the number of communities into which a network should be divided.

A property that seems to be common to many networks is *community structure*, the division of network nodes into groups within which the network connections are dense, but between which they are sparser.

Although we merely want to find if and how a given network breaks down into communities, we probably don't know how many such communities there are going to be, and there is no reason why they should be roughly the same size. Furthermore, the number of inter-community edges needn't be strictly minimized either, since more such edges are admissible between large communities than between small ones.

A useful approach is that taken by social network analysis with the set of techniques known as hierarchical clustering. These techniques are aimed at discovering natural divisions of (social) networks into groups, based on various metrics (see section 1.5) of similarity or strength of connection between vertices. They fall into two broad classes, *agglomerative* and *divisive*, depending on whether they focus on the addition or removal of edges to or from the network.

In an agglomerative method, similarities are calculated by one method or another between vertex pairs, and edges are then added to an initially empty network (n vertices with no edges) starting with the vertex pairs with

highest similarity. The procedure can be halted at any point, and the resulting components in the network are taken to be the communities. Alternatively, the entire progression of the algorithm from empty graph to complete graph can be represented in the form of a tree or dendrogram such as that shown in Figure 2.1:

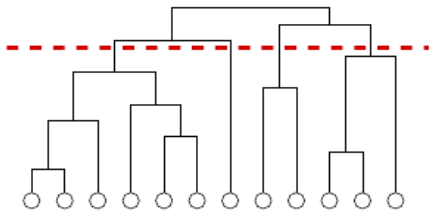

Figure 2.1: A hierarchical tree or dendogram.

Horizontal cuts through the tree represent the communities appropriate to different halting points.

Agglomerative methods based on a wide variety of similarity measures have been applied to different networks. Some networks have natural similarity metrics built in. Other networks have no natural metric, but suitable ones can be devised using correlation coefficients, path lengths, or matrix methods.

Agglomerative methods have their problems however.

(i) they fail with some frequency to find the correct communities in networks were the community structure is known, which makes it difficult to place much trust in them in other cases;

(ii) they have tendency to find only the cores of communities and leave out the periphery.

The core nodes in a community often have strong similarity, and hence are connected early in the agglomerative process, but peripheral nodes that have no strong similarity to others tend to get neglected, leading to structures like that shown in Figure 2.2 .

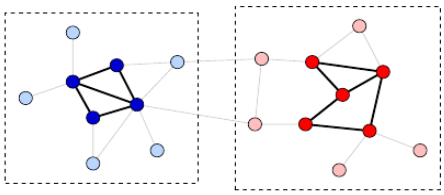

Figure 2.2: Agglomerative clustering methods are typically good at discovering the strongly linked cores of communities (bold vertices and edges) but tend to leave out peripheral vertices,even when, as here, most of them clearly belong to one community or another.

In this figure, there are a number of peripheral nodes whose community membership is obvious to the eye, in most cases they have only a single link to a

specific community, but agglomerative methods often fail to place such nodes correctly. Therefore, we focus on *divisive methods*. These methods have been relatively little studied in the previous literature, either in social network theory or elsewhere, but seem to offer a lot of promise.

In a divisive method, we start with the network of interest and attempt to find the *least* similar connected pairs of vertices and then remove the edges between them. By doing this repeatedly, we divide the network into smaller and smaller components, and again we can stop the process at any stage and take the components at that stage to be the network communities. The process can be represented as a dendrogram depicting the successive splits of the network into smaller and smaller groups.

Rather than looking for the most weakly connected vertex pairs, our approach will be to look for the edges in the network that are most *"between"* other vertices, meaning that the edge is, in some sense, responsible for connecting many pairs of others. Such edges need not be weak at all in the similarity sense.

2.1 Finding communities in a network

We present an algorithm for network clustering, i.e., the discovery of community structure in networks. Our discussion focuses primarily on networks with only a single type of vertex and a single type of undirected, unweighted edge, although generalizations to more complicated network types are certainly possible.

There are two central features that distinguish this algorithm from those that have preceded them. First, this algorithm is divisive rather than agglomerative. This doesn't differ in focusing on removing the edges between vertex pairs with lowest similarity, but on finding edges with the highest *"betweenness"* , where betweenness is some measure that favors edges that lie between communities and disfavors those that lie inside communities. If two communities are joined by only a few intercommunity edges, then all paths through the network from vertices in one community to vertices in the other must pass along one of those few edges. Given a suitable set of paths, one can count how many go along each edge in the graph, and this number we then expect to be largest for the intercommunity edges, thus providing a method for identifying them.

The betweenness measures is based on *shortest path*: we find the shortest path between all pairs of vertices and count how many run along each edge. We call this quantity the *edge betweenness*. We will refer to this betweenness measure as *shortest-path betweenness*. A fast algorithm for calculating the shortest-path betweenness is given in section 2.2.

The second way in which the method we study differ from previous ones is in the inclusion of a *"recalculation step"* in the algorithm. If we were to perform a standard divisive clustering based on edge betweenness it would calculate the edge betweenness for all edges in the network and then remove edges in decreasing order of betweenness to produce a dendrogram like that of Figure 2.1 , showing the order in which the network split up.

However, once the first edge in the network is removed in such an algorithm, the betweenness values for the remaining edges will no longer reflect the network as it now is. This can give rise to unwanted behaviors. For example, if two communities are joined by two edges, but, for one reason or another, most paths between the two flow along just one of those edges, then that edge will have a

high betweenness score and the other will not. An algorithm that calculated betweennesses only once and then removed edges in betweenness order would remove the first edge early in the course of its operation, but the second might not get removed until much later. Thus the obvious division of the network into two parts might not be discovered by the algorithm. In the worst case the two parts themselves might be individually broken up before the division between the two is made. Problems like this crop up in real networks with some regularity and render algorithms of this type ineffective for the discovery of community structure.

The solution, luckily, is obvious. It is necessary recalculate betweenness measure after the removal of each edge. This certainly adds to the computational effort of performing the calculation, but its effect on the results is so desirable that we consider the price worth paying.

Thus the general form of community structure finding algorithm is as follows:

(1) calculate betweenness scores for all edges in the network;

(2) find the edge with the highest score and remove it from the network;

(3) recalculate betweenness for all remaining edges;

(4) Repeat from step 2.

In the next section, we discuss implementation of the algorithm for finding community structure. This algorithm calculate the *shortest-path betweenness* for all edges in time $O(m \times n)$, where m is the numbers of edges in the graph and n is the number of vertices.

2.2 Implementation of shortest-path betweenness

In theory, the descriptions of the last section completely define the methods we consider in this report, but in practice there are a number of tricks to its implementation that are important for turning the description into a workable computer algorithm. Essentially all of the work in the algorithm is in the calculation of the betweenness scores for the edges; the job of finding and removing the highest-scoring edge is trivial and not computationally demanding.

At first sight, it appears that calculating the edge betweenness measure based on shortest paths for all edges will take $O(mn^2)$ operations on a graph with m edges and n vertices: calculating the shortest path between a particular pair of vertices can be done using *breadth-first search* in time $O(m)$ [10, 11], and there are $O(n^2)$ vertex pairs. Recently however new algorithms have been proposed by Newman [9] that can perform the calculation faster than this, finding all betweennesses in $O(mn)$ time. Breadth-first search can find shortest paths from a single vertex s to all others in time $O(m)$.

In the simplest case, when there is only a single shortest path from the source vertex to any other the resulting set of paths forms a shortest-path tree, see Figure 2.3:

We can now use this tree to calculate the contribution to betweenness for each edge from this set of paths as follows. We find first the *"leaves"* of the tree, i.e., those nodes such that no shortest paths to other nodes pass through them, and we assign a score of 1 to the single edge that connects each to the

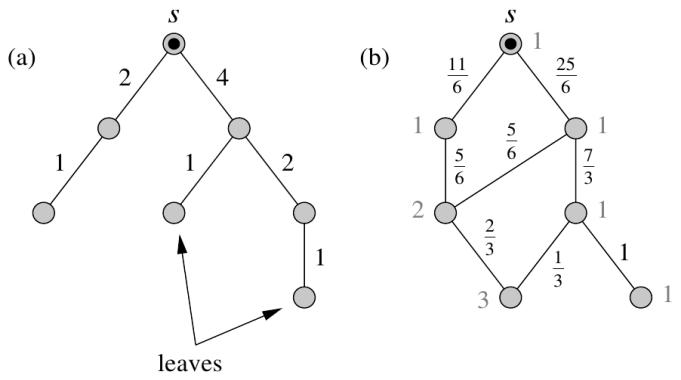

Figure 2.3: Calculation of shortest-path betweenness: (a) When there is only a single shortest path from a source vertex s (top) to all other reachable vertices, those paths necessarily form a tree, which makes the calculation of the contribution to betweenness from this set of paths particularly simple, as describe in the text. (b) For cases in which there is more than one shortest path to some vertices, the calculation is more complex.

rest of the tree, as shown in the figure. Then, starting with those edges that are farthest from the source vertex on the tree, i.e., lowest in Figure 2.3 (a), we work upwards, assigning a score to each edge that is 1 plus the sum of the scores on the neighboring edges immediately below it. When we have gone though all edges in the tree, the resulting scores are the betweenness counts for the paths from vertex s.

Repeating the process for all possible vertices s and summing the scores, we arrive at the full betweenness scores for shortest paths between all pairs. The breadth-first search and the process of working up through the tree both take worst-case time $O(m)$ and there are n vertices total, so the entire calculation takes time $O(mn)$ as claimed.

This simple case serves to illustrate the basic principle behind the algorithm. In general, however, it is not the case that there is only a single shortest path between any pair of vertices. Most networks have at least some vertex pairs between which there are several shortest paths of equal length. Figure 2.3 (b) shows a simple example of a shortest path "tree" for a network with this property.

The resulting structure is in fact no longer a tree, and in such cases an extra step is required in the algorithm to calculate the betweenness correctly. Multiple shortest paths between a pair of vertices are given equal weights summing to 1. For example, if there are three shortest paths, each will be given weight 1/3.

Note that the paths may run along the same edge or edges for some part of their length, resulting in edges with greater weight. To calculate correctly what fraction of the paths flow along each edge in the network, we generalize the breadth-first search part of the calculation, as follows.

Consider Figure 2.3 (b) and suppose we are performing a breadth-first search starting at vertex s. We carry out the following steps:

1. the initial vertex s is given distance $d_s = 0$ and a weight $ws = 1$;

2. every vertex i adjacent to s is given distance $d_i = d_s + 1 = 1$, and weight

$w_i = w_s = 1$;

3. for each vertex j adjacent to one of those vertices i we do one of three things:
   (a) if j has not yet been assigned a distance, it is assigned distance $d_j = d_i + 1$ and weight $w_j = w_i$;
   (b) if j has already been assigned a distance and $d_j = d_i + 1$, then the vertex's weight is increased by w_i, that is $w_j \leftarrow w_j + w_i$;
   (c) if j has already been assigned a distance and $d_j < d_i + 1$, we do nothing;
4. repeat from step 3 until no vertices remain that have assigned distances but whose neighbors do not have assigned distances.

This algorithm can be implemented most efficiently using a queue or first-in/first-out buffer to store the vertices that have been assigned a distance, just as in the standard breadth-first search.

Physically, the weight on a vertex i represents the number of distinct paths from the source vertex to i. These weights are precisely what we need to calculate our edge betweennesses, because if two vertices i and j are connected, with j farther than i from the source s, then the fraction of a shortest path from j through i to s is given by w_i/w_j . Thus, to calculate the contribution to edge betweenness from all shortest paths starting at s, we need only carry out the following steps:

1. find every "leaf" vertex t, i.e., a vertex such that no paths from s to other vertices go though t;
2. for each vertex i neighboring t assign a score to the edge from t to i of w_i/w_t;
3. now, starting with the edges that are farthest from the source vertex s, lower down in a diagram such as Figure 2.3 (b), work up towards s. To the edge from vertex i to vertex j, with j being farther from s than i, assign a score that is 1 plus the sum of the scores on the neighboring edges immediately below it (i.e., those with which it shares a common vertex), all multiplied by wi/wj;
4. Repeat from step 3 until vertex s is reached.

Now repeating this process for all n source vertices s and summing the resulting scores on the edges gives us the total betweenness for all edges in time $O(mn)$.

We now have to repeat this calculation for each edge removed from the network, of which there are m, and hence the complete community structure algorithm based on shortest-path betweenness operates in worst-case time $O(m^2n)$, or $O(n^3)$ time on a sparse graph.

2.2.1 An example

Now we see step by step how the shortest-path betweennesses is calculated for the graph in Figure 2.3, using the algorithm discussed in the previous section.

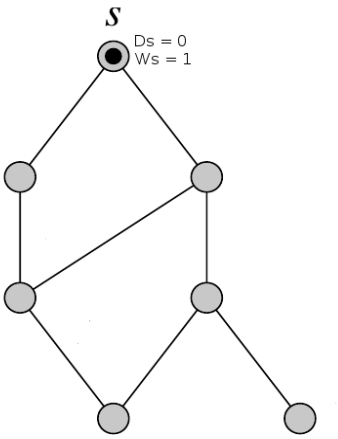

Figure 2.4: calculation of distance and weight of root node.

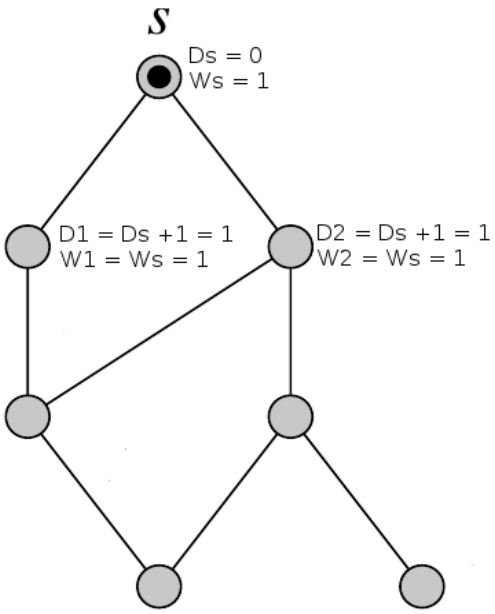

Figure 2.5: calculation of distance and weight of node 1 and node 2.

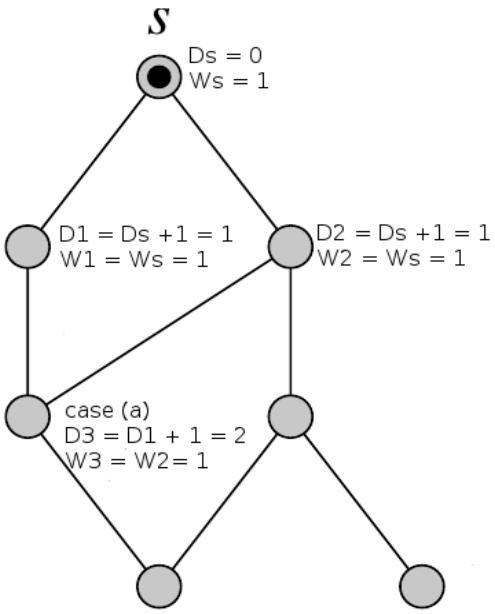

Figure 2.6: calculation of distance and weight of node 3.

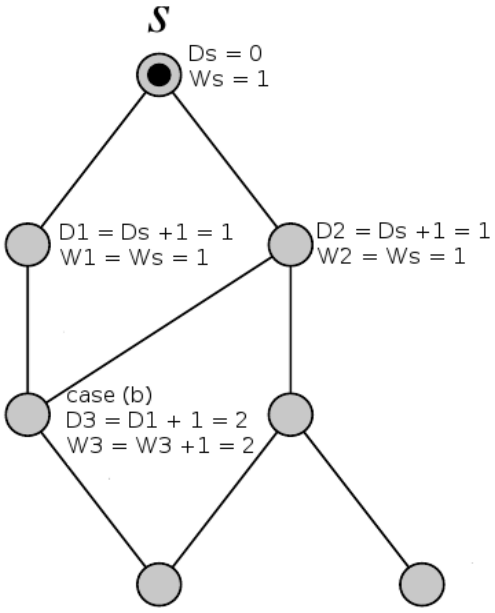

Figure 2.7: updating the weight of node 3.

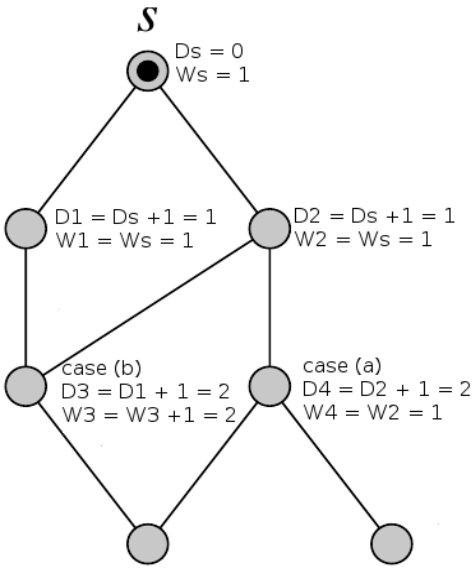

Figure 2.8: calculation of distance and weight of node 4.

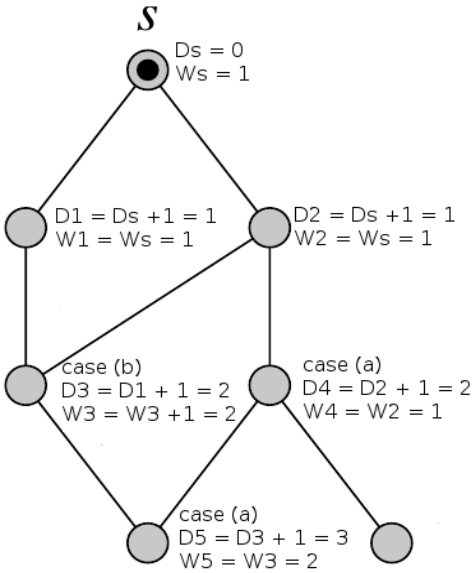

Figure 2.9: calculation of distance and weight of node 5.

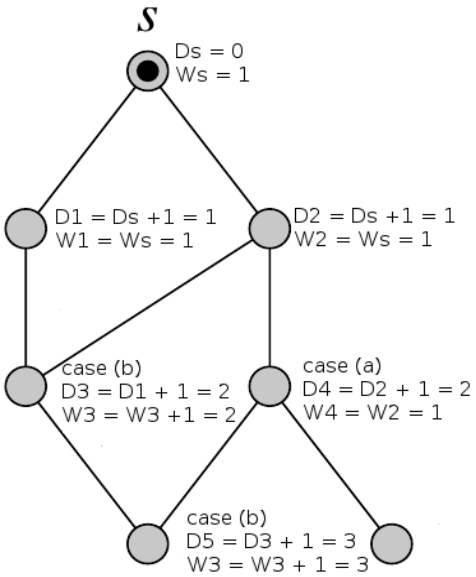

Figure 2.10: updating the weight of node 5.

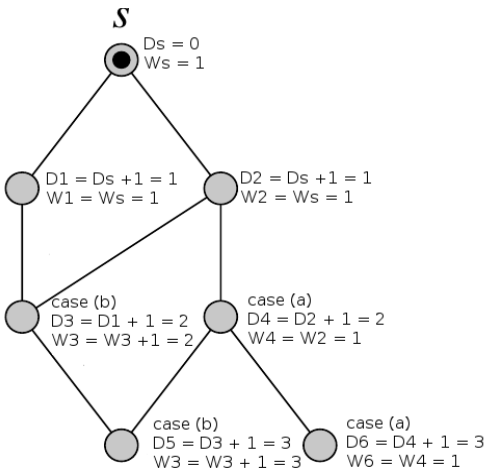

Figure 2.11: calculation of distance and weight of node 6.

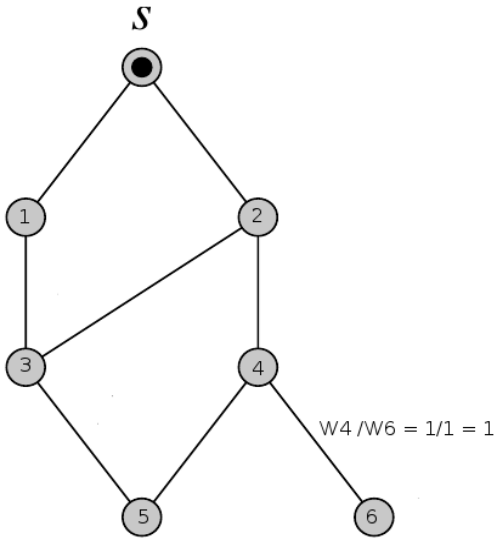

Figure 2.12: calculation of edge betweennesses of edge $< 4, 6 >$.

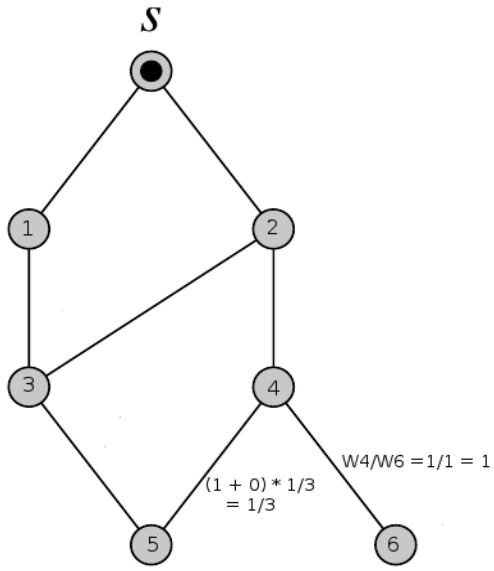

Figure 2.13: calculation of edge betweennesses of edge $< 4, 5 >$.

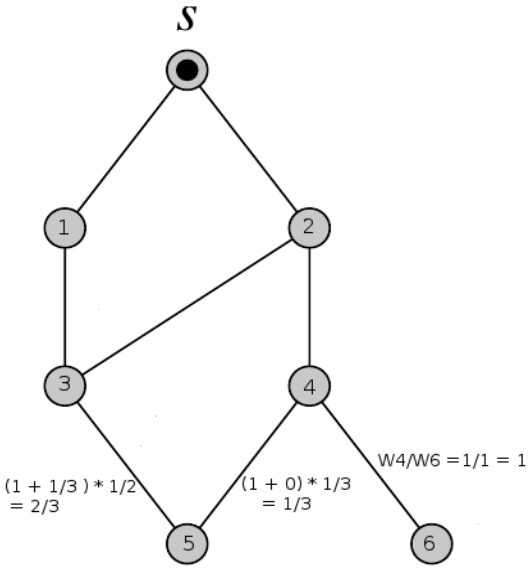

Figure 2.14: calculation of edge betweennesses of edge $< 3, 5 >$.

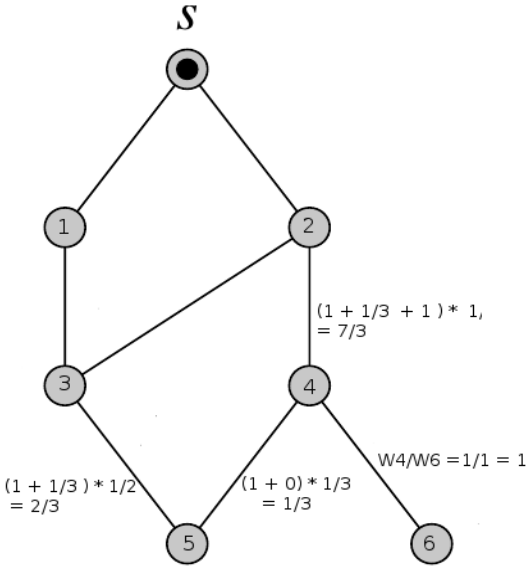

Figure 2.15: calculation of edge betweennesses of edge $< 2, 4 >$.

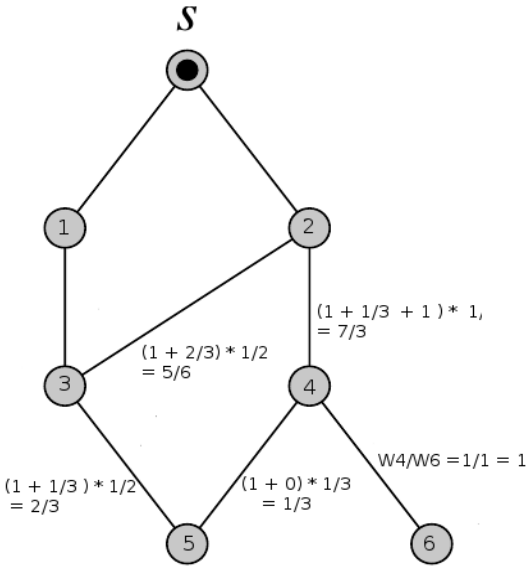

Figure 2.16: calculation of edge betweennesses of edge $< 2, 3 >$.

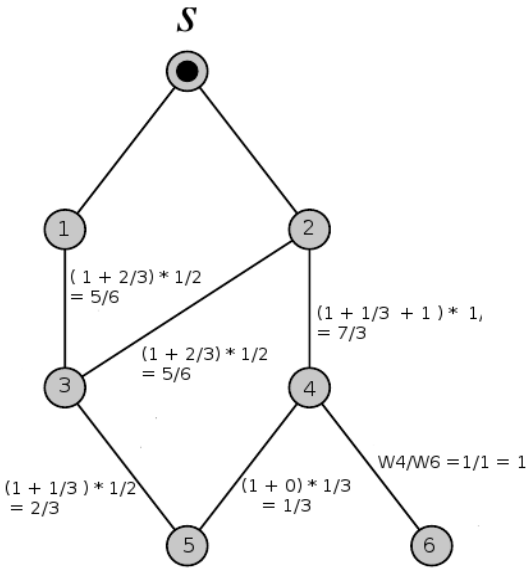

Figure 2.17: calculation of edge betweennesses of edge $< 1, 3 >$.

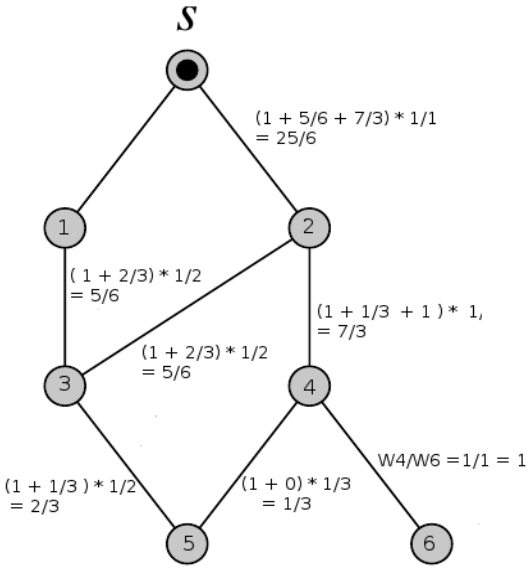

Figure 2.18: calculation of edge betweennesses of edge $< 0, 2 >$.

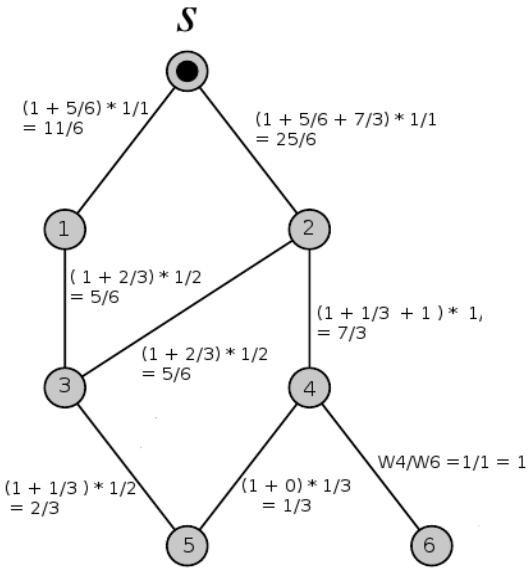

Figure 2.19: calculation of edge betweennesses of edge $< 0, 1 >$.

Chapter 3

The Hadoop Distribuited Filesystem

Hadoop is open source software that enables distributed parallel processing of huge amounts of data across inexpensive, commodity servers. it is not possible to process huge data sets with a single computer because we might take too long. Morover, when a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called *distributed filesystems*. Since they are network-based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss.

Hadoop comes with a distributed filesystem called HDFS, which stands for *Hadoop Distributed Filesystem*.

3.1 The design of HDFS

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. Let's examine this statement in more detail:

(i) *Very large files*: "Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

(ii) *Streming data access*: HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

(iii) *Commodity hardware*: Hadoop doesn't require expensive, highly reliable hardware to run on. It's designed to run on clusters of commodity hardware.

HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

It is also worth examining the applications for which using HDFS does not work so well. While this may change in the future, these are areas where HDFS is not a good fit today:

(i) *Low-latency data access*: Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency.

(ii) *Lots of small files*: Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware.

(iii) *Multiple writers, arbitrary file modifications*: Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

3.2 HDFS Concepts

3.2.1 Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file.

HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

Why is a Block in HDFS so large?

HDFS blocks are large compared to disk blocks, and the reason is to *minimize the cost of seeks*. By making a block large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms, and the transfer rate is 100 MB/s, then to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 64 MB,

although many HDFS installations use 128 MB blocks. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

Having a block abstraction for a distributed filesystem brings several benefits:

(1) a file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster;

(2) making the unit of abstraction a block rather than a file simplifies the storage subsystem. Simplicity is something to strive for all in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (blocks are just a chunk of data to be stored).

Blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level.

3.2.2 Namenodes and Datanodes

An HDFS cluster has two types of node operating in a *master-worker* pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. The client presents a POSIX-like filesystem interface, so the user code does not need to know about the namenode and datanode to function.

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

(i) The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic.

(ii) It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

3.2.3 Data Flow

Anatomy of a file read

To get an idea of how data flows between the client interacting with HDFS, the namenode and the datanodes, consider Figure 3.1, which shows the main sequence of events when reading a file.

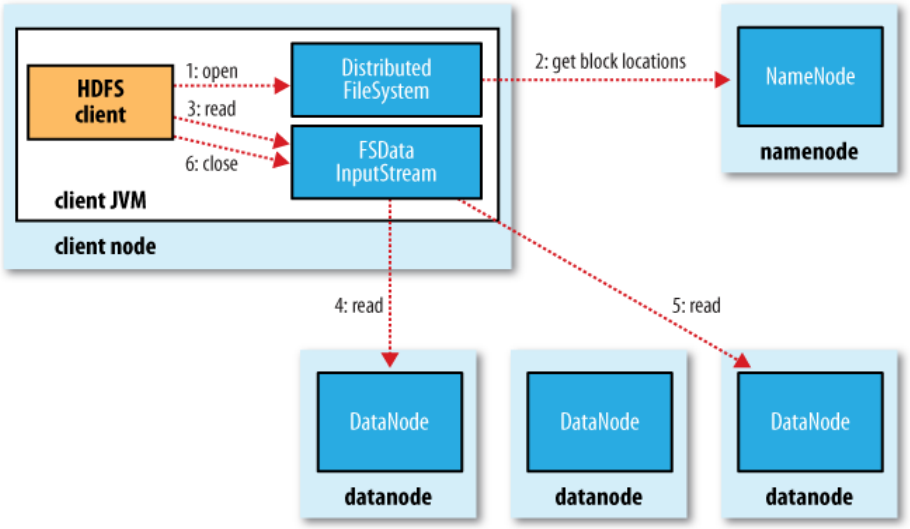

Figure 3.1: A Client reading data from HDFS

The client opens the file it wishes to read by calling **open()** on the **FileSystem** object, which for HDFS is an instance of DistributedFileSystem (step 1 in Figure 3.1). **DistributedFileSystem** calls the namenode, using RPC (Remote Procedure Call), to determine the locations of the blocks for the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client. If the client is itself a datanode, then it will read from the local datanode, if it hosts a copy of the block.

The **DistributedFileSystem** returns an **FSDataInputStream** to the client for it to read data from. **FSDataInputStream** in turn wraps a **DFSInputStream** , which manages the datanode and namenode I/O.

The client then calls **read()** on the stream (step 3). **DFSInputStream**, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls **read()** repeatedly on the stream (step 4). When the end of the block is reached, **DFSInputStream** will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

When the client has finished reading, it calls **close()** on the **FSDataInputStream** (step 6).

During reading, if the **DFSInputStream** encounters an error while communicating with a datanode, then it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The **DFSInputStream** also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode before the **DFSInput Stream** attempts to read a replica of the block from another datanode.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients, since the data traffic is spread across all the datanodes in the cluster. The namenode meanwhile merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

Anatomy of a file write

Next we'll look at how files are written to HDFS.

The case we're going to consider is the case of creating a new file, writing data to it, then closing the file. See Figure 3.2.

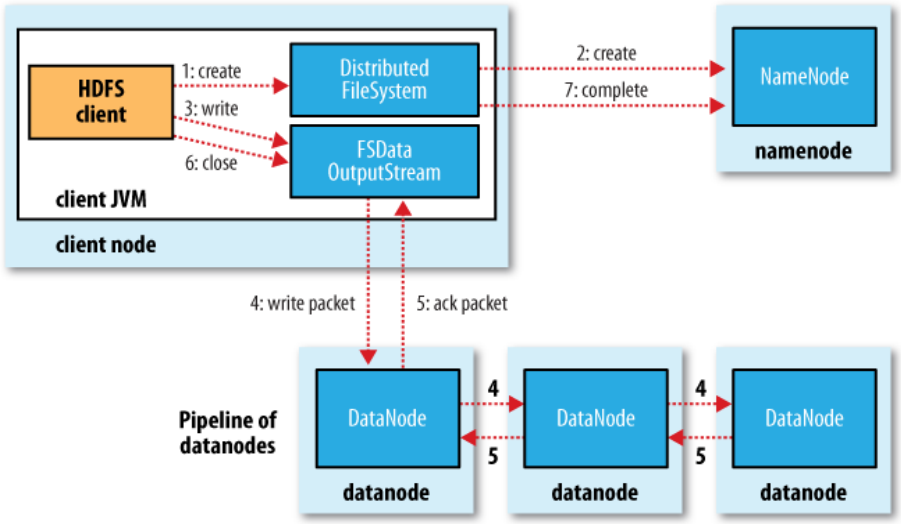

Figure 3.2: A Client writing data from HDFS

The client creates the file by calling **create()** on **DistributedFileSystem** (step 1 in Figure 3.2). **DistributedFileSystem** makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist, and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an **IOException**. The **DistributedFileSystem** returns an **FSDataOutputStream** for the client to start writing data to. Just as in the read case, **FSDataOutputStream** wraps a **DFSOutputStream**, which handles communication with the datanodes and namenode.

As the client writes data (step 3), **DFSOutputStream** splits it into packets, which it writes to an internal queue, called the *data queue*. The data queue is consumed by the **DataStreamer**, whose responsibility it is to ask the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline, we'll assume the replication level is three, so there are three nodes in the pipeline. The **DataStreamer** streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).

If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on. The failed datanode is removed from the pipeline and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as **dfs.replication.min** replicas (default one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (**dfs.replication**, which defaults to three).

When the client has finished writing data, it calls **close()** on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of, so it only has to wait for blocks to be minimally replicated before returning successfully.

3.3 The Command-Line Interface

We're going to have a look at HDFS by interacting with it from the command line and there we are going to run HDFS on one machine *(pseudo-distribuited mode)* .

There are two properties that we set in the pseudo-distributed configuration that deserve further explanation. The first is **fs.default.name**, set to *hdfs://localhost/*, which is used to set a default filesystem for Hadoop. Filesystems are specified by a URI, and here we have used an **hdfs** URI to configure Hadoop to use HDFS by default. The HDFS daemons will use this property to determine the host and port for the HDFS namenode. We'll be running it on localhost, on the default HDFS port, 8020. And HDFS clients will use this property to work out where the namenode is running so they can connect to it.

We set the second property, **dfs.replication** , to 1 so that HDFS doesn't replicate filesystem blocks by the default factor of three. When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would perpetually warn about blocks being under-replicated. This setting solves that problem.

3.3.1 Basic Filesystem Operations

The filesystem is ready to be used, and we can do all of the usual filesystem operations such as reading files, creating directories, moving files, deleting data, and listing directories. You can type **hadoop fs -help** to get detailed help on every command.

Start by copying a file from the local filesystem to HDFS:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt
     hdfs://localhost/user/tom/ quangle.txt
```

This command invokes Hadoop's filesystem shell command **fs**. The local file *quangle.txt* is copied to the file */user/tom/quangle.txt* on the HDFS instance running on localhost. In fact, we could have omitted the scheme and host of the URI and picked up the default, *hdfs://localhost*, as specified in *core-site.xml*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt
     /user/tom/quangle.txt
```

We could also have used a relative path and copied the file to our home directory in HDFS, which in this case is */user/tom*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

Let's copy the file back to the local filesystem and check whether it's the same:

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9
MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

The MD5 digests are the same, showing that the file survived its trip to HDFS and is back intact.

Finally, let's look at an HDFS file listing. We create a directory first just to see how it is displayed in the listing:

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x - tom supergroup   0 2009-04-02 22:41 /user/tom/books
-rw-r--r-- 1 tom supergroup 118 2009-04-02 22:29 /user/tom/quangle.txt
```

The information returned is very similar to the Unix command *ls -l*, with a few minor differences. The first column shows the file mode. The second column is the replication factor of the file. The entry in this column is empty for directories since the concept of replication does not apply to them, directories are treated as metadata and stored by the namenode, not the datanodes. The third and fourth columns show the file owner and group. The fifth column is the size of the file in bytes, or zero for directories. The sixth and seventh columns are the last modified date and time. Finally, the eighth column is the absolute name of the file or directory.

3.4 Hadoop Filesystems

Hadoop has an abstract notion of filesystem, of which HDFS is just one implementation. The Java abstract class **org.apache.hadoop.fs.FileSystem** represents a filesystem in Hadoop, and there are several concrete implementations, which are described in figure 3.3

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	*file*	`fs.LocalFileSystem`	A filesystem for a locally connected disk with client-side checksums. Use `RawLocalFileSystem` for a local filesystem with no checksums. See "LocalFileSystem" on page 76.
HDFS	*hdfs*	`hdfs.DistributedFileSystem`	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
HFTP	*hftp*	`hdfs.HftpFileSystem`	A filesystem providing read-only access to HDFS over HTTP. (Despite its name, HFTP has no connection with FTP.) Often used with *distcp* (see "Parallel Copying with distcp" on page 70) to copy data between HDFS clusters running different versions.
HSFTP	*hsftp*	`hdfs.HsftpFileSystem`	A filesystem providing read-only access to HDFS over HTTPS. (Again, this has no connection with FTP.)
HAR	*har*	`fs.HarFileSystem`	A filesystem layered on another filesystem for archiving files. Hadoop Archives are typically used for archiving files in HDFS to reduce the namenode's memory usage. See "Hadoop Archives" on page 71.
KFS (CloudStore)	*kfs*	`fs.kfs.KosmosFileSystem`	CloudStore (formerly Kosmos filesystem) is a distributed filesystem like HDFS or Google's GFS, written in C++. Find more information about it at *http://kosmosfs.sourceforge.net/*.
FTP	*ftp*	`fs.ftp.FTPFileSystem`	A filesystem backed by an FTP server.
S3 (native)	*s3n*	`fs.s3native.NativeS3FileSystem`	A filesystem backed by Amazon S3. See *http://wiki.apache.org/hadoop/AmazonS3*.
S3 (block-based)	*s3*	`fs.s3.S3FileSystem`	A filesystem backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5 GB file size limit.

Figure 3.3: Hadoop filesystems.

Hadoop provides many interfaces to its filesystems, and it generally uses the URI scheme to pick the correct filesystem instance to communicate with. For example, the filesystem shell that we met in the previous section operates with all Hadoop filesystems. To list the files in the root directory of the local filesystem, type:

```
% hadoop fs -ls file:///
```

Although it is possible (and sometimes very convenient) to run MapReduce programs that access any of these filesystems, when you are processing large volumes of data, you should choose a distributed filesystem that has the data locality optimization, such as HDFS or KFS.

3.4.1 The Java Interface

In this section, we dig into the Hadoop's *FileSystem* class: the API for interacting with one of Hadoop's filesystems.

Readind Data using the FileSystem API

A file in a Hadoop filesystem is represented by a Hadoop Path object (and not a **java.io.File** object, since its semantics are too closely tied to the local filesystem). You can think of a Path as a Hadoop filesystem URI, such as *hdfs://localhost/user/tom/ quangle.txt*.

FileSystem is a general filesystem API, so the first step is to retrieve an instance for the filesystem we want to use, HDFS in this case. There are two static factory methods for getting a FileSystem instance:

```
public static FileSystem get(Configuration conf)
     throws IOException
public static FileSystem get(URI uri, Configuration conf)
     throws IOException
```

A **Configuration** object encapsulates a client or server's configuration, which is set using configuration files read from the classpath, such as *conf/core-site.xml*. The first method returns the default filesystem. The second uses the given **URI**'s scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given URI.

With a **FileSystem** instance in hand, we invoke an **open()** method to get the input stream for a file:

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path f, int bufferSize)
     throws IOException
```

The first method uses a default buffer size of 4 K. Putting this together, we can show a program for displayng files from Hadoop filesystem on standard output, like Unix **cat** command.

Example: Displaying files from a Hadoop filesystem on standard output by using the FileSystem directly

```
public class FileSystemCat {

  public static void main(String[] args) throws Exception {
   String uri = args[0];
   Configuration conf = new Configuration();
   FileSystem fs = FileSystem.get(URI.create(uri), conf);
   InputStream in = null;
   try {
    in = fs.open(new Path(uri));
    IOUtils.copyBytes(in, System.out, 4096, false);
   } finally {
    IOUtils.closeStream(in);
   }
  }
}
```

The program runs as follows:

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream

The **open()** method on **FileSystem** actually returns a **FSDataInputStream** rather than a standard **java.io class**.
This class is a specialization of **java.io.DataInputStream** with support for random access, so you can read from any part of the stream:

```
package org.apache.hadoop.fs;
   public class FSDataInputStream extends DataInputStream
   implements Seekable, PositionedReadable {
     // implementation elided
}
```

The **Seekable** interface permits seeking to a position in the file and a query method for the current offset from the start of the file (**getPos()**):

```
public interface Seekable {
   void seek(long pos) throws IOException;
   long getPos() throws IOException;
}
```

Calling **seek()** with a position that is greater than the length of the file will result in an **IOException**. Unlike the **skip()** method of **java.io.InputStream** that positions the stream at a point later than the current position, **seek()** can move to an arbitrary, absolute position in the file.

Example: program that writes a file from a Hadoop filesystem to standard out twice: after writing it once, it seeks to the start of the file and streams through it once again.

```
public class FileSystemDoubleCat {
 public static void main(String[] args) throws Exception {
  String uri = args[0];
  Configuration conf = new Configuration();
  FileSystem fs = FileSystem.get(URI.create(uri), conf);
  FSDataInputStream in = null;
  try {
    in = fs.open(new Path(uri));
    IOUtils.copyBytes(in, System.out, 4096, false);
    in.seek(0); // go back to the start of the file
    IOUtils.copyBytes(in, System.out, 4096, false);
  } finally {
    IOUtils.closeStream(in);
  }
 }
}
```

Here's the result of running it on a small file:

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream also implements the **PositionedReadable** interface for reading parts of a file at a given offset:

```
public interface PositionedReadable {
public int read(long position, byte[] buffer, int offset,
   int length) throws IOException;
public void readFully(long position, byte[] buffer, int offset,
   int length) throws IOException;
public void readFully(long position, byte[] buffer)
   throws IOException;
}
```

The **read()** method reads up to **length** bytes from the given **position** in the file into the **buffer** at the given **offset** in the buffer. The return value is the number of bytes actually read: callers should check this value as it may be less than length. The **readFully()** methods will read **length** bytes into the buffer, unless the end of the file is reached, in which case an EOFException is thrown.

All of these methods preserve the current offset in the file and are thread-safe, so they provide a convenient way to access another part of the file, while reading the main body of the file. In fact, they are just implemented using the **Seekable** interface using the following pattern:

```
long oldPos = getPos();
  try {
   seek(position);
   // read data
  } finally {
   seek(oldPos);
  }
```

Finally, bear in mind that calling **seek()** is a relatively expensive operation and should be used sparingly. You should structure your application access patterns to rely on streaming data, rather than performing a large number of seeks.

Writing Data

The **FileSystem** class has a number of methods for creating a file. The simplest is the method that takes a **Path** object for the file to be created and returns an output stream to write to:

```
public FSDataOutputStream create(Path f) throws IOException
```

There are overloaded versions of this method that allow you to specify whether to forcibly overwrite existing files, the replication factor of the file, the buffer size to use when writing the file, the block size for the file, and file permissions. The **create()** methods create any parent directories of the file to be written that don't already exist.

There's also an overloaded method for passing a callback interface, **Progressable**, so your application can be notified of the progress of the data being written to the datanodes:

```
package org.apache.hadoop.util;

public interface Progressable {
   public void progress();
}
```

As an alternative to creating a new file, you can append to an existing file using the **append()** method:

```
public FSDataOutputStream append(Path f) throws IOException
```

The append operation allows a single writer to modify an already written file by opening it and writing data from the final offset in the file. The append operation is optional and not implemented by all Hadoop filesystems. For example, HDFS supports append, but S3 filesystems don't.

Example: we show how to copy a local file to a Hadoop filesystem. We illustrate progress by printing a period every time the progress() method is called by Hadoop, which is after each 64 K packet of data is written to the datanode pipeline.

```
public class FileCopyWithProgress {
 public static void main(String[] args) throws Exception {
 String localSrc = args[0];
 String dst = args[1];

 InputStream in = new BufferedInputStream(
                         new FileInputStream(localSrc));

 Configuration conf = new Configuration();
 FileSystem fs = FileSystem.get(URI.create(dst), conf);
 OutputStream out = fs.create(new Path(dst), new Progressable() {
  public void progress() {
   System.out.print(".");
  }
 });

 IOUtils.copyBytes(in, out, 4096, true);
 }
}
```

Typical usage:

```
% hadoop FileCopyWithProgress input/docs/1400-8.txt
    hdfs://localhost/user/tom/1400-8.txt
  ...............
```

FSDataOutputStream

The **create()** method on **FileSystem** returns an **FSDataOutputStream**, which, like **FSDataInputStream**, has a method for querying the current position in the file:

```
package org.apache.hadoop.fs;

public class FSDataOutputStream extends DataOutputStream
   implements Syncable {

public long getPos() throws IOException {
   // implementation elided
}
  // implementation elided
}
```

However, unlike **FSDataInputStream**, **FSDataOutputStream** does not permit seeking. This is because HDFS allows only sequential writes to an open file or appends to an already written file. In other words, there is no support for writing to anywhere other than the end of the file, so there is no value in being able to seek while writing. **Directories FileSystem** provides a method to create a directory:

```
public boolean mkdirs(Path f) throws IOException
```

This method creates all of the necessary parent directories if they don't already exist, just like the **java.io.File**'s **mkdirs()** method. It returns true if the directory was successfully created.

Often, you don't need to explicitly create a directory, since writing a file, by calling **create()**, will automatically create any parent directories.

Querying the Filesystem

An important feature of any filesystem is the ability to navigate its directory structure and retrieve information about the files and directories that it stores. The **FileStatus** class encapsulates filesystem metadata for files and directories, including file length, block size, replication, modification time, ownership, and permission information.

The method **getFileStatus()** on **FileSystems** provides a way of getting a **FileStatus** object for a single file or directory.

we show an example for demonstrating file status information:

```
public class ShowFileStatusTest {
  private MiniDFSCluster cluster; // use an in-process HDFS cluster for testing
  private FileSystem fs;
```

```
@Before
public void setUp() throws IOException {
 Configuration conf = new Configuration();
 if (System.getProperty("test.build.data") == null) {
  System.setProperty("test.build.data", "/tmp");
 }
 cluster = new MiniDFSCluster(conf, 1, true, null);
 fs = cluster.getFileSystem();
 OutputStream out = fs.create(new Path("/dir/file"));
 out.write("content".getBytes("UTF-8"));
 out.close();
}

@After
public void tearDown() throws IOException {
 if (fs != null) { fs.close(); }
 if (cluster != null) { cluster.shutdown(); }
}

@Test(expected = FileNotFoundException.class)
public void throwsFileNotFoundForNonExistentFile() throws IOException {
 fs.getFileStatus(new Path("no-such-file"));
}

@Test
public void fileStatusForFile() throws IOException {
 Path file = new Path("/dir/file");
 FileStatus stat = fs.getFileStatus(file);
 assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
 assertThat(stat.isDir(), is(false));
 assertThat(stat.getLen(), is(7L));
 assertThat(stat.getModificationTime(),
   is(lessThanOrEqualTo(System.currentTimeMillis())));
 assertThat(stat.getReplication(), is((short) 1));
 assertThat(stat.getBlockSize(), is(64 * 1024 * 1024L));
 assertThat(stat.getOwner(), is("tom"));
 assertThat(stat.getGroup(), is("supergroup"));
 assertThat(stat.getPermission().toString(), is("rw-r--r--"));
}

@Test
public void fileStatusForDirectory() throws IOException {
 Path dir = new Path("/dir");
 FileStatus stat = fs.getFileStatus(dir);
 assertThat(stat.getPath().toUri().getPath(), is("/dir"));
 assertThat(stat.isDir(), is(true));
 assertThat(stat.getLen(), is(0L));
 assertThat(stat.getModificationTime(),
   is(lessThanOrEqualTo(System.currentTimeMillis())));
```

```
    assertThat(stat.getReplication(), is((short) 0));
    assertThat(stat.getBlockSize(), is(0L));
    assertThat(stat.getOwner(), is("tom"));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
    }
}
```

If no file or directory exists, a **FileNotFoundException** is thrown. However, if you are interested only in the existence of a file or directory, then the **exists()** method on **FileSystem** is more convenient:

```
public boolean exists(Path f) throws IOException
```

File Patterns

It is a common requirement to process sets of files in a single operation. For example, a MapReduce job for log processing might analyze a month's worth of files contained in a number of directories. Rather than having to enumerate each file and directory to specify the input, it is convenient to use wildcard characters to match multiple files with a single expression, an operation that is known as *globbing*. Hadoop provides two **FileSystem** method for processing globs:

```
public FileStatus[] globStatus(Path pathPattern)
      throws IOException
public FileStatus[] globStatus(Path pathPattern, PathFilter filter)
      throws IOException
```

The **globStatus()** method returns an array of **FileStatus** objects whose paths match the supplied pattern, sorted by path.

Hadoop supports the same set of glob characters as Unix bash (see figure below).

Glob	Name	Matches
*	*asterisk*	Matches zero or more characters
?	*question mark*	Matches a single character
[ab]	*character class*	Matches a single character in the set {a, b}
[^ab]	*negated character class*	Matches a single character that is not in the set {a, b}
[a-b]	*character range*	Matches a single character in the (closed) range [a, b], where a is lexicographically less than or equal to b
[^a-b]	*negated character range*	Matches a single character that is not in the (closed) range [a, b], where a is lexicographically less than or equal to b
{a,b}	*alternation*	Matches either expression a or b
\c	*escaped character*	Matches character c when it is a metacharacter

Figure 3.4: Glob characters and their meanings.

Imagine that logfiles are stored in a directory structure organized hierarchically by date. So, for example, logfiles for the last day of 2007 would go in a directory named /2007/12/31. Suppose that the full file listing is:

- /2007/12/30
- /2007/12/31
- /2008/01/01
- /2008/01/02

Here are some file globs and their expansions:

Glob	Expansion
`/*`	*/2007 /2008*
`/*/*`	*/2007/12 /2008/01*
`/*/12/*`	*/2007/12/30 /2007/12/31*
`/200?`	*/2007 /2008*
`/200[78]`	*/2007 /2008*
`/200[7-8]`	*/2007 /2008*
`/200[^01234569]`	*/2007 /2008*
`/*/*/{31,01}`	*/2007/12/31 /2008/01/01*
`/*/*/3{0,1}`	*/2007/12/30 /2007/12/31*
`/*/{12/31,01/01}`	*/2007/12/31 /2008/01/01*

Figure 3.5: Globs example.

Deleting Data

Use the **delete()** method on **FileSystem** to permanently remove files or directories:

```
public boolean delete(Path f, boolean recursive)
     throws IOException
```

If f is a file or an empty directory, then the value of **recursive** is ignored. A nonempty directory is only deleted, along with its contents, if **recursive** is **true** (otherwise an **IOException** is thrown).

Bibliography

[1] R. O. Duda and Hart. *Pattern Classification and Scene Analysis.* John Wiley and Sons, Inc., 1973.

[2] K.C. Gowda and Diday. Symbolic clustering using a new dissimilarity measure. *IEEE Trans. Syst. Man Cybern.*, 1992.

[3] K.C. Gowda and Krishna. Agglomerative clustering using the concept of mutual nearest neighborhood, 1977.

[4] A. K. Jain and Dubes. *Algorithms for Clustering Data.* Prentice-Hall, 1988.

[5] S. C. Johnson. *Hierarchical Clustering Schemes.* Psychometrika, 1967.

[6] S. Y. LU and K. S. FU. A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Trans. Syst. Man Cybern*, 1978.

[7] Matteo Matteucci. Single-linkage clustering: an example, 2010. [Online; accessed 20-may-2012].

[8] Stepp R. Michalski and Diday. Automated construction of classifications: conceptual clustering versus numerical taxonomy. *IEEE Trans. Pattern Anal. Mach. Intell.*, 1983.

[9] M. E. J. Newman. Scientific collaboration networks: Shortest paths, weighted networks, and centrality. *Phys. Rev. E 64, 016132*, 2001.

[10] T. L. Magnanti Ravindra Ahuja and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, 1993.

[11] R. L. Rivest T. H. Cormen, C. E. Leiserson and C. Stein. Introduction to algorithms. *MIT Press*, 2001.