fields are completely different. Area codes are sometimes changed. For example, every (650) number used to be a (415) number, so an old record may have (415) 555-1212 and a newer record (650) 555-1212, and yet these numbers refer to the same phone.

5. *Abbreviations.* Sometimes words in an address are spelled out; other times an abbreviation may be used. Thus, "Sesame St." and "Sesame Street" may be the same street.

Thus, when deciding whether two records represent the same entity, we need to look carefully at the kinds of discrepancies that occur and devise a scoring system or other test that measures the similarity of records. Ultimately, we must turn the score into a yes/no decision: do the records represent the same entity or not? We shall mention below two useful approaches to measuring the similarity of records.

Edit Distance

Values that are strings can be compared by counting the number of insertions and/or deletions of characters it takes to turn one string into another. Thus, `Smythe` and `Smith` are at distance 3 (delete the "y" and "e," then insert the "i").

An alternative edit distance counts 1 for a *mutation*, that is, a replacement of one letter by another. In this measure, `Smythe` and `Smith` are at distance 2 (mutate "y" to "i" and delete "e"). This edit distance makes mistyped characters "cost" less, and therefore may be appropriate if typing errors are common in the data.

Finally, we may devise a specialized distance that takes into account the way the data was constructed. For instance, if we decide that changes of area codes are a major source of errors, we might charge only 1 for changing the entire area code from one to another. We might decide that the problem of misinterpreted family names was severe and allow two components of a name to be swapped at low cost, so `Chen Li` and `Li Chen` are at distance 1.

Once we have decided on the appropriate edit distance for each field, we can define a similarity measure for records. For example, we could sum the edit distances of each of the pairs of corresponding fields in the two records, or we could compute the sum of the squares of those distances. Whatever formula we use, we have then to say that records represent the same entity if their similarity measure is below a given threshold.

Normalization

Before applying an edit distance, we might wish to "normalize" records by replacing certain substrings by others. The goal is that substrings representing the same "thing" will become identical. For instance, it may make sense to use a table of abbreviations and replace abbreviations by what they normally stand for. Thus, `St.` would be replaced by `Street` in street addresses and by `Saint` in town names. Also, we could use a table of nicknames and variant spellings, so `Sue` would become `Susan` and `Jeffery` would become `Geoffrey`.

One could even use the *Soundex* encoding of names, so names that sound the same are represented by the same string. This system, used by telephone information services, for example, would represent `Smith` and `Smythe` identically. Once we have normalized values in the records, we could base our similarity test on identical values only (e.g., a majority of fields have identical values in the two records), or we could further use an edit distance to measure the difference between normalized values in the fields.

21.7.2 Merging Similar Records

In many applications, when we find two records that are similar enough to merge, we would like to replace them by a single record that, in some sense, contains the information of both. For instance, if we want to compile a "dossier" on the entity represented, we might take the union of all the values in each field. Or we might somehow combine the values in corresponding fields to make a single value. If we try to combine values, there are many rules that we might follow, with no obvious best approach. For example, we might assume that a full name should replace a nickname or initials, and a middle initial should be used in place of no middle initial. Thus, "Susan Williams" and "S. B. Williams" would be combined into "Susan B. Williams."

It is less clear how to deal with misspellings. For instance, how would we combine the addresses "123 Oak St." and "123 Yak St."? Perhaps we could look at the town or zip-code and determine that there was an Oak St. there and no Yak St. But if both existed and had 123 in their range of addresses, there is no right answer.

Another problem that arises if we use certain combinations of a similarity test and a merging rule is that our decision to merge one pair of records may preclude our merging another pair. An example may help illustrate the risk.

	name	*address*	*phone*
(1)	`Susan Williams`	`123 Oak St.`	`818-555-1234`
(2)	`Susan Williams`	`456 Maple St.`	`818-555-1234`
(3)	`Susan Williams`	`456 Maple St.`	`213-555-5678`

Figure 21.12: Three records to be merged

Example 21.22: Suppose that we have the three name-address-phone records in Fig. 21.12. and our similarity rule is: "must agree exactly in at least two out of the three fields." Suppose also that our merge rule is: "set the field in which the records disagree to the empty string."

Then records (1) and (2) are similar; so are records (2) and (3). Note that records (1) and (3) are not similar to each other, which serves to remind us that "similarity" is not normally a transitive relationship. If we decide to replace (1) and (2) by their merger, we are left with the two tuples:

	name	*address*	*phone*
(1-2)	Susan Williams		818-555-1234
(3)	Susan Williams	456 Maple St.	213-555-5678

These records disagree in two fields, so they cannot be merged. Had we merged (1) and (3) first, we would again have a situation where the remaining record cannot be merged with the result.

Another choice for similarity and merge rules is:

1. Merge by taking the union of the values in each field, and

2. Declare two records similar if at least two of the three fields have a nonempty intersection.

Consider the three records in Fig. 21.12. Again, (1) is similar to (2) and (2) is similar to (3), but (1) is not similar to (3). If we choose to merge (1) and (2) first, we get:

	name	*address*	*phone*
(1-2)	Susan Williams	{123 Oak St. 456 Maple St.}	818-555-1234
(3)	Susan Williams	456 Maple St.	213-555-5678

Now, the remaining two tuples are similar, because 456 Maple St. is a member of both address sets and Susan Williams is a member of both name sets. The result is a single tuple:

	name	*address*	*phone*
(1-2-3)	Susan Williams	{123 Oak St., 456 Maple St.}	{818-555-1234, 213-555-5678}

□

21.7.3 Useful Properties of Similarity and Merge Functions

Any choice of similarity and merge functions allows us to test pairs of records for similarity and merge them if so. As we saw in the first part of Example 21.22, the result we get when no more records can be merged may depend on which pairs of mergeable records we consider first. Whether or not different ending configurations can result depends on properties of similarity and merger.

There are several properties that we would expect any merge function to satisfy. If $\wedge$ is the operation that produces the merge of two records, it is reasonable to expect:

1. $r \wedge r = r$ (*Idempotence*). That is, the merge of a record with itself should surely be that record.

2. $r \wedge s = s \wedge r$ (*Commutativity*). If we merge two records, the order in which we list them should not matter.

3. $(r \wedge s) \wedge t = r \wedge (s \wedge t)$ (*Associativity*). The order in which we group records for a merger should not matter.

These three properties say that the merge operation is a semilattice. Note that both merger functions in Example 21.22 have these properties. The only tricky point is that we must remember that $r \wedge s$ need not defined for all records r and s. We do, however, assume that:

- If r and s are similar, then $r \wedge s$ is defined.

There are also some properties that we expect the similarity relationship to have, and ways that we expect similarity and merging to interact. We shall use $r \approx s$ to say that records r and s are similar.

a) $r \approx r$ (*Idempotence* for similarity). A record is always similar to itself.

b) $r \approx s$ if and only if $s \approx r$ (*Commutativity* of similarity). That is, in deciding whether two records are similar, it does not matter in which order we list them.

c) If $r \approx s$, then $r \approx (s \wedge t)$ (*Representability*). This rule requires that if r is similar to some other record s (and thus could be merged with s), but s is instead merged with some other record t, then r remains similar to the merger of s and t and can be merged with that record.

Note that representability is the property most likely to fail. In particular, it fails for the first merger rule in Example 21.22, where we merge by setting disagreeing fields to the empty string. In particular, representability fails when r is record (3) of Fig. 21.12, s is (2), and t is (1). On the other hand, the second merger rule of Example 21.22 satisfies the representability rule. If r and s have nonempty intersections in at least two fields, those shared values will still be present if we replace s by $s \wedge t$.

The collection of properties above are called the *ICAR properties.* The letters stand for Idempotence, Commutativity, Associativity, and Representability, respectively.

21.7.4 The R-Swoosh Algorithm for ICAR Records

When the similarity and merge functions satisfy the ICAR properties, there is a simple algorithm that merges all possible records. The representability property guarantees that if two records are similar, then as they are merged with other records, the resulting records are also similar and will eventually be merged. Thus, if we repeatedly replace any pair of similar records by their merger, until no more pairs of similar records remain, then we reach a unique set of records that is independent of the order in which we merge.

A useful way to think of the merger process is to imagine a graph whose nodes are the records. There is an edge between nodes r and s if $r \approx s$. Since similarity need not be transitive, it is possible that there are edges between r and s and between s and t, yet there is no edge between r and t. For instance, the records of Fig. 21.12 have the graph of Fig. 21.13.

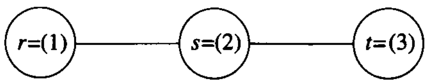

Figure 21.13: Similarity graph from Fig. 21.12

However, representability tells us that if we merge s and t, then because r is similar to s, it will be similar to $s \wedge t$. Thus, we can merge all three of r, s, and t. Likewise, if we merge r and s first, representability says that because $s \approx t$, we also have $(r \wedge s) \approx t$, so we can merge t with $r \wedge s$. Associativity tells us that the resulting record will be the same, regardless of the order in which we do the merge.

The idea described above extends to any set of ICAR nodes (records) that are connected in any way. That is, regardless of the order in which we do the merges, the result is that every connected component of the graph becomes a single record. This record is the merger of all the records in that component. Commutativity and associativity are enough to tell us that the order in which we perform the mergers does not matter.

Although computing connected components of a graph is simple in principle, when we have millions of records or more, it is not feasible to construct the graph. To do so would require us to test similarity of every pair of records. The "R-Swoosh" algorithm is an implementation of this idea that organizes the comparisons so we avoid, in many cases, comparing all pairs of records. Unfortunately, if no records at all are similar, then there is no algorithm that can avoid comparing all pairs of records to determine this fact.

Algorithm 21.23: R-Swoosh.

INPUT: A set of records I, a similarity function $\approx$, and a merge function $\wedge$. We assume that $\approx$ and $\wedge$ satisfy the ICAR properties. If they do not, then the algorithm will still merge some records, but the result may not be the maximum or best possible merging.

OUTPUT: A set of merged records O.

METHOD: Execute the steps of Fig. 21.14. The value of O at the end is the output. □

```
O := emptyset;
WHILE I is not empty DO BEGIN
   let r be any record in I;
   find, if possible, some record s in O that is similar to r;
   IF no record s exists THEN
      move r from I to O
   ELSE BEGIN
      delete r from I;
      delete s from O;
      add the merger of r and s to I;
   END;
END;
```

Figure 21.14: The R-Swoosh Algorithm

Example 21.24: Suppose that I is the three records of Fig. 21.12, and that we use the ICAR similarity and merge functions from Example 21.22, where we take the union of possible values for a field to produce the corresponding field in the merged record. Initially, O is empty. We pick one of the records from I, say record (1) to be the record r in Fig. 21.14. Since O is empty, there is no possible record s, so we move record (1) from I to O.

We next pick a new record r. Suppose we pick record (3). Since record (3) is not similar to record (1), which is the only record in O, we again have no value of s, so we move record (3) from I to O. The third choice of r must be record (2). That record is similar to both of the records in O, so we must pick one to be s; say we pick record (1). Then we merge records (1) and (2) to get the record

	name	*address*	*phone*
(1-2)	Susan Williams	{123 Oak St., 456 Maple St.}	818-555-1234

We remove record (2) from I, remove record (1) from O, and insert the above record into I. At this point, I consists of only the record (1-2) and O consists of only the record (3).

The execution of the R-Swoosh Algorithm ends after we pick record (1-2) as r — the only choice — and pick record (3) as s — again the only choice. These records are merged, to produce

	name	*address*	*phone*
(1-2-3)	Susan Williams	{123 Oak St., 456 Maple St.}	{818-555-1234, 213-555-5678}

and deleted from I and O, respectively. The record (1-2-3) is put in I, at which point it is the only record in I, and O is empty. At the last step, this record is moved from I to O, and we are done. □

21.7.5 Why R-Swoosh Works

Recall that for ICAR similarity and merge functions, the goal is to merge records that form connected components. There is a loop invariant that holds for the while-loop of Fig. 21.14:

- If a connected component C is not completely merged into one record, then there is at least one record in I that is either in C or was formed by the merger of some records from C.

To see why this invariant must hold, suppose that the selected record r in some iteration of the loop is the last record in I from its connected component C. If r is the only record that is the merger of one or more records from C, then it may be moved to O without violating the loop invariant.

However, if there are other records that are the merger of one or more records from C, they are in O. Let r be the merger of the set of records $R \subseteq C$. Note that R could be only one record, or could be many records. However, since R is not all of C, there must be an original record r_1 in R that is similar to another original record r_2 that is in $C-R$. Suppose r_2 is currently merged into a record r' in O. By representability, perhaps applied several times, we can start with the known $r_1 \approx r_2$ and deduce that $r \approx r'$. Thus, r' can be s in Fig. 21.14. As a result, r will surely be merged with some record from O. The resulting merged record will be placed in I and is the merger of some or all records from C. Thus, the loop invariant continues to hold.

21.7.6 Other Approaches to Entity Resolution

There are many other algorithms known to discover and (optionally) merge similar records. We shall outline some of them briefly here.

Non-ICAR Datasets

First, suppose the ICAR properties do not hold, but we want to find all possible mergers of records, including cases where one record r_1 is merged with a record r_2, but later, r_1 (not the merger $r_1 \wedge r_2$) is also merged with r_3. If so, we need to systematically compare all records, including those we constructed by merger, with all other records, again including those constructed by merger.

To help control the proliferation of records, we can define a *dominance* relation $r \leq s$ that means record s contains all the information contained in record r. If so, we can eliminate record r from further consideration. If the merge function is a semilattice, then the only reasonable choice for $\leq$ is $a \leq b$ if and only if $a \wedge b = b$. This dominance function is always a partial order, regardless of what semilattice is used. If the merge operation is not even a semilattice, then the dominance function must be constructed in an ad-hoc manner.

Clustering

In some entity-resolution applications, we do not want to merge at all, but will instead group records into *clusters* such that members of a cluster are in some sense similar to each other and members of different clusters are not similar. For example, if we are looking for similar products sold on eBay, we might want the result to be not a single record for each kind of product, but rather a list of the records that represent a common product for sale. Clustering of large-scale data involves a complex set of options. We shall discuss the matter further in Section 22.5.

Partitioning

Since any algorithm for doing a complete merger of similar records may be forced to examine each pair of records, it may be infeasible to get an exact answer to a large entity-resolution problem. One solution is to group the records, perhaps several times, into groups that are likely to contain similar records, and look only within each group for pairs of similar records.

Example 21.25: Suppose we have millions of name-address-phone records, and our measure of similarity is that the total edit distance of the values in the three fields must be at most 5. We could partition the records into groups such that each group has the same name field. We could also partition the records according to the value in their address field, and a third time according to their phone numbers. Thus, each record appears in three groups and is compared only with the members of those groups. This method will not notice a pair of similar records that have edit distance 2 in their phone fields, 2 in their name fields, and 1 in their address fields. However, in practice, it will catch almost all similar pairs. □

The idea in Example 21.25 is actually a special case of an important idea: "locality-sensitive hashing." We discuss this topic in Section 22.4.

21.7.7 Exercises for Section 21.7

Exercise 21.7.1: A string s is a *subsequence* of a string t if s is formed from t by deleting 0 or more positions of t. For example, if t = "abcab", then substrings of t include "aba" (delete positions 3 and 5), "bc" (delete positions 1, 4, and 5), and the empty string (delete all positions).

a) What are all the other subsequences of "abcab"?

b) What are the subsequences of "aabb"?

! c) If a string consists of n distinct characters, how many subsequences does it have?

Exercise 21.7.2: A *longest common subsequence* of two strings s and t is any string r that is a subsequence of both s and t and is as long as any other string that is a substring of both. For example, the longest common subsequences of `"aba"` and `"bab"` are `"ab"` and `"ba"`. Give a the longest common subsequence for each pair of the following strings: `"she"`, `"hers"`, `"they"`, and `"theirs"`?

Exercise 21.7.3: A *shortest common supersequence* of two strings s and t is any string r of which both s and t are subsequences, such that no string shorter than r has both s and t as subsequences. For example, the some of the shortest common supersequences of `"abc"` and `"cb"` are `"abcb"` and `"acbc"`.

a) What are the shortest common supersequences of each pair of strings in Exercise 21.7.2?

! b) What are all the other shortest common supersequences of `"abc"` and `"cb"`?

!! c) If two strings have no characters in common, and are of lengths m and n, respectively, how many shortest common supersequences do the two strings have?

!! **Exercise 21.7.4:** Suppose we merge records (whose fields are strings) by taking, for each field, the lexicographically first longest common subsequence of the strings in the corresponding fields.

a) Does this definition of merge satisfy the idempotent, commutative, and associative laws?

b) Repeat (a) if instead corresponding fields are merged by taking the lexicographically first shortest common supersequence.

! **Exercise 21.7.5:** Suppose we define the similarity and merge functions by:

i. Records are similar if in all fields, or in all but one field, either both records have the same value or one has `NULL`.

ii. Merge records by letting each field have the common value if both records agree in that field or have value `NULL` if the records disagree in that field. Note that `NULL` disagrees with any nonnull value.

Show that these similarity and merge functions have the ICAR properties.

! **Exercise 21.7.6:** In Section 21.7.6 we suggested that if $\wedge$ is a semilattice, then the dominance relationship defined by $a \leq b$ if and only if $a \wedge b = b$ is a partial order. That is, $a \leq b$ and $b \leq c$ imply $a \leq c$ (transitivity) and $a \leq b$ and $b \leq a$ if and only if $a = b$ (antisymmetry). Prove that $\leq$ is a partial order, using the reflexivity, commutativity, and associativity properties of a semilattice.

21.8 Summary of Chapter 21

✦ *Integration of Information*: When many databases or other information sources contain related information, we have the opportunity to combine these sources into one. However, heterogeneities in the schemas often exist; these incompatibilities include differing types, codes or conventions for values, interpretations of concepts, and different sets of concepts represented in different schemas.

✦ *Approaches to Information Integration*: Early approaches involved "federation," where each database would query the others in the terms understood by the second. A more recent approach is warehousing, where data is translated to a global schema and copied to the warehouse. An alternative is mediation, where a virtual warehouse is created to allow queries to a global schema; the queries are then translated to the terms of the data sources.

✦ *Extractors and Wrappers*: Warehousing and mediation require components at each source, called extractors and wrappers, respectively. A major function of either is to translate queries and results between the global schema and the local schema at the source.

✦ *Wrapper Generators*: One approach to designing wrappers is to use templates, which describe how a query of a specific form is translated from the global schema to the local schema. These templates are tabulated and interpreted by a driver that tries to match queries to templates. The driver may also have the ability to combine templates in various ways, and/or perform additional work such as filtering, to answer more complex queries.

✦ *Capability-Based Optimization*: The sources for a mediator often are able or willing to answer only limited forms of queries. Thus, the mediator must select a query plan based on the capabilities of its sources, before it can even think about optimizing the cost of query plans as conventional DBMS's do.

✦ *Adornments*: These provide a convenient notation in which to describe the capabilities of sources. Each adornment tells, for each attribute of a relation, whether, in queries matching that adornment, this attribute requires or permits a contant value, and whether constants must be chosen from a menu.

✦ *Conjunctive Queries*: A single Datalog rule, used as a query, is a convenient representation for queries involving joins, possibly followed by selection and/or projection.

✦ *The Chain Algorithm*: This algorithm is a greedy approach to answering mediator queries that are in the form of a conjunctive query. Repeatedly look for a subgoal that matches one of the adornments at a source, and obtain the relation for that subgoal from the source. Doing so may provide a set of constant bindings for some variables of the query, so repeat the process, looking for additional subgoals that can be resolved.

- *Local-as-View Mediators*: These mediators have a set of global, virtual predicates or relations at the mediator, and each source is described by views, which are conjunctive queries whose subgoals use the global predicates. A query at the mediator is also a conjunctive query using the global predicates.

- *Answering Queries Using Views*: A local-as-view mediator searches for solutions to a query, which are conjunctive queries whose subgoals use the views as predicates. Each such subgoal of a proposed solution is expanded using the conjunctive query that defines the view, and it is checked that the expansion is contained in the query. If so, the proposed solution does indeed provide (some of the) answers to the query.

- *Containment of Conjunctive Queries*: We test for containment of conjunctive queries by looking for a containment mapping from the containing query to the contained query. A containment mapping is a substitution for variables that turns the head of the first into the head of the second and turns each subgoal of the first into some subgoal of the second.

- *Limiting the Search for Solutions*: The LMSS Theorem says that when seaching for solutions to a query at a local-as-view mediator, it is sufficient to consider solutions that have no more subgoals than the query does.

- *Entity Resolution*: The problem is to take records with a common schema, find pairs or groups of records that are likely to represent the same entity (e.g., a person) and merge these records into a single record that represents the information of the entire group.

- *ICAR Similarity and Merge Functions*: Certain choices of similarity and merge functions satisfy the properties of idempotence, commutativity, associativity, and representability. The latter is the key to efficient algorithms for merging, since it guarantees that if two records are similar, their successors will also be similar even as they are merged into records that represent progressively larger sets of original records.

- *The R-Swoosh Algorithm*: If similarity and merge functions have the ICAR properties, then the complete merger of similar records will group all records that are in a connected component of the graph formed from the similarity relation on the original records. The R-Swoosh algorithm is an efficient way to make all necessary mergers without determining similarity for every pair of records.

21.9 References for Chapter 21

Federated systems are surveyed in [11]. The concept of the mediator comes from [12]. Implementation of mediators and wrappers, especially the wrapper-generator approach, is covered in [4]. Capability-based optimization for mediators was explored in [10, 13]; the latter describes the Chain Algorithm.

Local-as-view mediators come from [7]. The LMSS Theorem is from [6], and the idea of containment mappings to decide containment of conjunctive queries is from [2]. [8] extends the idea to sources with limited capabilities. [5] is a survey of logical information-integration techniques.

Entity resolution was first studied informally by [9] and formally by [3]. The theory presented here, the R-Swoosh Algorithm, and related algorithms are from [1].

1. O. Benjelloun, H. Garcia-Molina, J. Jonas, Q. Su, S. E. Whang, and J. Widom, "Swoosh: a generic approach to entity resolution." Available as `http://dbpubs.stanford.edu:8090/pub/2005-5`.

2. A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational databases," *Proc. Ninth Annual Symposium on Theory of Computing*, pp. 77–90, 1977.

3. I. P. Fellegi and A. B. Sunter, "A theory for record linkage," *J. American Statistical Assn.* **64**, pp. 1183–1210, 1969.

4. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, V. Vassalos, J. D. Ullman, and J. Widom, "The TSIMMIS approach to mediation: data models and languages," *J. Intelligent Information Systems* 8:2 (1997), pp. 117–132.

5. A. Y. Levy, "Logic-based techniques in data integration," *Logic-Based Artificial Intelligence* (J. Minker, ed.), pp. 575–595, Kluwer, Norwell, MA, 2000.

6. A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering queries using views," *Proc. 25th Annual Symposium on Principles of Database Systems*, pp. 95–104, 1995.

7. A. Y. Levy, A. Rajaraman, and J. J. Ordille, "Querying heterogeneous information sources using source descriptions," *Intl. Conf. on Very Large Databases*, pp. 251–262, 1996.

8. A. Y. Levy, A. Rajaraman, and J. D. Ullman, "Answering queries using limited external query processors," *Proc. Fifteenth Annual Symposium on Principles of Database Systems*, pp. 227–237, 1996.

9. H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James, "Automatic linkage of vital records," *Science* **130**, pp. 954–959, 1959.

10. Y. Papakonstantinou, A. Gupta, and L. Haas, "Capabilities-base query rewriting in mediator systems," *Conference on Parallel and Distributed Information Systems* (1996). Available as `http://dbpubs.stanford.edu/pub/1995-2`.

11. A. P. Sheth and J. A. Larson, "Federated databases for managing distributed, heterogeneous, and autonomous databases," *Computing Surveys* **22**:3 (1990), pp. 183–236.

12. G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer* **C-25**:1 (1992), pp. 38–49.

13. R. Yerneni, C. Li, H. Garcia-Molina, and J. D. Ullman, "Optimizing large joins in mediation systems," *Proc. Seventh Intl. Conf. on Database Theory*, pp. 348–364, 1999.

Chapter 22

Data Mining

"Data mining" is the process of examining data and finding simple rules or models that summarize the data. The rules can range from very general, such as "50% of the people who buy hot dogs also buy mustard," to the very specific: "these three individual's pattern of credit-card expenditures indicate that they are running a terrorist cell." Our discussion of data mining will concentrate on mining information from very large databases.

We begin by looking at "market-basket" data, records of the things people buy together, such as at a supermarket. This study leads to a number of efficient algorithms for finding "frequent itemsets" in large databases, including the "A-Priori" Algorithm and its extensions.

We next turn to finding "similar" items in a large collection. Example applications include finding documents on the Web that share a significant amount of common text or finding books that have been bought by many of the same Amazon customers. Two key techniques for this problem are "minhashing" and "locality-sensitive hashing."

We conclude the chapter with a discussion of the problem of large-scale clustering in high dimensions. An example application is clustering Web pages by the words they use. In that case, each word might be a dimension, and a document is placed in this space by counting the number of occurrences of each word.

22.1 Frequent-Itemset Mining

There is a family of problems that arise from attempts by marketers to use large databases of customer purchases to extract information about buying patterns. The fundamental problem is called "frequent itemsets" — what sets of items are often bought together? This information is sometimes further refined into "association rules" — implications that people who buy one set of items are likely to buy another particular item. The same technology has many other uses, from discovering combinations of genes related to certain diseases to finding plagiarism among documents on the Web.

22.1.1 The Market-Basket Model

In several important applications, the data involves a set of *items*, perhaps all the items that a supermarket sells, and a set of *baskets*; each basket is a subset of the set of items, typically a small subset. The baskets each represent a set of items that someone has bought together. Here are two typical examples of where market-basket data appears.

Supermarket Checkout

A supermarket chain may sell 10,000 different items. Daily, millions of customers wheel their shopping carts ("market baskets") to the checkout, and the cash register records the set of items they purchased. Each such set is one basket, in the sense used by the market-basket model. Some customers may have identified themselves, using a discount card that many supermarket chains provide, or by their credit card. However, the identity of the customer often is not necessary to get useful information from the data.

Stores analyze the data to learn what typical customers buy together. For example, if a large number of baskets contain both hot dogs and mustard, the supermarket manager can use this information in several ways.

1. Apparently, many people walk from where the hot dogs are to where the mustard is. We can put them close together, and put between them other foods that might also be bought with hot dogs and mustard, e.g., ketchup or potato chips. Doing so can generate additional "impulse" sales.

2. The store can run a sale on hot dogs and at the same time raise the price of mustard (without advertising that fact, of course). People will come to the store for the cheap hot dogs, and many will need mustard too. It is not worth the trouble to go to another store for cheaper mustard, so they buy that too. The store makes back on mustard what it loses on hot dogs, and also gets more customers into the store.

While the relationship between hot dogs and mustard may be obvious to those who think about the matter, even if they have no data to analyze, there are many pairs of items that are connected but may be less obvious. The most famous example is diapers and beer.[1]

There are some conditions on when a fact about co-occurrence of sets of items can be useful. Any useful pair (or larger set) of items must be bought by many customers. It is not even necessary that there be any connection between purchases of the items, as long as we know lots of customers buy them

[1]One theory: if you buy diapers, you probably have a baby at home. If so, you are not going out to a bar tonight, so you are more likely to buy beer at a supermarket.

all. Conversely, strongly linked, but rarely purchased items (e.g., caviar and champagne) are not very interesting to the supermarket, because it doesn't pay to advertise things that few customers are interested in buying anyway.

On-Line Purchases

Amazon.com offers several million different items for sale, and has several tens of millions of customers. While brick-and-mortar stores such as the supermarket discussed above can only make money on combinations of items that large numbers of people buy, Amazon and other on-line sellers have the opportunity to tailor their offers to every customer. Thus, an interesting question is to find pairs of items that many customers have bought together. Then, if one customer has bought one of these items but not the other, it might be good for Amazon to advertise the second item when this customer next logs in. We can treat the purchase data as a market-basket problem, where each "basket" is the set of items that one particular customer ever has bought.

But there is another way Amazon can use the same data. This approach, often called "collaborative filtering," has us look for customers that are similar in their purchase habits. For example, we could look for pairs, or even larger sets, of customers that have bought many of the same items. Then, if a customer logs in, Amazon might pitch an item that a similar customer bought, but this customer has not.

Finding similar customers also can be couched as a market-basket problem. Here, however, the "items" are the customers and the "baskets" are the items for sale by Amazon. That is, for each item I sold by Amazon there is a "basket" consisting of all the customers who bought I.

It is worth noting that the meaning of "many baskets" differs in the on-line and brick-and-mortar situations. In the brick-and-mortar case, we may need thousands of baskets containing a set of items before we can exploit that information profitably. For on-line stores, we need many fewer baskets containing a set of items, before we can use the information in the limited context we intend (pitching one item to one customer).

On the other hand, the brick-and-mortar store doesn't need too many examples of good sets of items to use; they can't run sales on millions of items. In contrast, the on-line store needs millions of good pairs to work with — at least one for each customer. As a result, the most effective techniques for analyzing on-line purchases may not be those of this section, which exploit the assumption that many occurrences of a pair of items are needed. Rather, we shall resume our discussion of finding correlated, but infrequent, pairs in Section 22.3.

22.1.2 Basic Definitions

Suppose we are given a set of items I and a set of baskets B. Each basket b in B is a subset of I. To talk about frequent sets of items, we need a *support threshold* s, which an integer. We say a set of items $J \subseteq I$ is *frequent* if there are at least s baskets that contain all the items in J (perhaps along with other items). Optionally, we can express the support s as a percentage of $|B|$, the number of baskets in B.

Example 22.1: Suppose our set of items I consists of the six movies

$$\{BI, BS, BU, HP1, HP2, HP3\}$$

standing for the *Bourne Identity, Bourne Supremacy, Bourne Ultimatum*, and *Harry Potter I, II*, and *III*. The table of Fig. 22.1 shows eight viewers (baskets of items) and the movies they have seen. An x indicates they saw the movie.

	BI	BS	BU	$HP1$	$HP2$	$HP3$
V_1	x	x	x			
V_2				x	x	x
V_3	x			x		
V_4	x	x		x	x	
V_5	x	x	x	x		
V_6	x		x			
V_7		x		x	x	
V_8	x	x		x	x	x

Figure 22.1: Market-basket data about viewers and movies

Suppose that $s = 3$. That is, in order for a set of items to be considered a frequent itemset, it must be a subset of at least three baskets. Technically, the empty set is a subset of all baskets, so it is frequent but uninteresting. In this example, all singleton sets except $\{HP3\}$ appear in at least three baskets. For example, $\{BI\}$ is contained in V_1, V_3, V_4, V_5, V_6, and V_8.

Now, consider which doubleton sets (pairs of items) are frequent. Since $HP3$ is not frequent by itself, it cannot be part of a frequent pair. However, each of the 10 pairs involving the other five movies might be frequent. For example, $\{BI, BS\}$ is frequent because it appears in at least three baskets; in fact it appears in four: V_1, V_4, V_5, and V_8.

Also:

- $\{BI, HP1\}$ is frequent appearing in V_3, V_4, V_5, and V_8.
- $\{BS, HP1\}$ is frequent, appearing in V_4, V_5, V_7, and V_8.
- $\{HP1, HP2\}$ is frequent, appearing in V_2, V_4, V_7, and V_8.

No other pair is frequent.

There is one frequent triple: $\{BI, BS, HP1\}$. This set is a subset of the baskets V_4, V_5, and V_8. There are no frequent itemsets of size greater than three. □

22.1.3 Association Rules

A natural query about market-basket data asks for implications among purchases that people make. That is, we want to find pairs of items such that people buying the first are likely to buy the second as well. More generally, people buying a particular set of items are also likely to buy yet another particular item. This idea is formalized by "association rules."

An *association rule* is a statement of the form $\{i_1, i_2, \ldots, i_n\} \Rightarrow j$, where the i's and j are items. In isolation, such a statement asserts nothing. However, three properties that we might want in useful rules of this form are:

1. High *Support*: the support of this association rule is the support of the itemset $\{i_1, i_2, \ldots, i_n, j\}$.

2. High *Confidence*: the probability of finding item j in a basket that has all of $\{i_1, i_2, \ldots, i_n\}$ is above a certain threshold, e.g., 50%, e.g., "at least 50% of the people who buy diapers buy beer."

3. *Interest*: the probability of finding item j in a basket that has all of $\{i_1, i_2, \ldots, i_n\}$ is significantly higher or lower than the probability of finding j in a random basket. In statistical terms, j correlates with

   $$\{i_1, i_2, \ldots, i_n\}$$

   either positively or negatively. The alleged relationship between diapers and beer is really a claim that the association rule $\{\text{diapers}\} \Rightarrow \text{beer}$ has high interest in the positive direction.

Note that even if an association rule has high confidence or interest, it will tend not to be useful unless it also has high support. The reason is that if the support is low, then the number of instances of the rule is not large, which limits the benefit of a strategy that exploits the rule. Also, it is important not to confuse an association rule, even with high values for support, confidence, and interest, with a causal rule. For instance, the "beer and diapers" example mentioned in Section 22.1.1 suggests that the association rule $\{\text{beer}\} \Rightarrow \text{diapers}$ has high confidence, but that does not mean beer "causes" diapers. Rather, the theory suggested there is that both are caused by a "hidden variable" — the baby at home.

Example 22.2: Using the data from Fig. 22.1, consider the association rule

$$\{BI, BS\} \Rightarrow BU$$

Its support is 2, since there are two baskets, V_1 and V_5 that contain all three "Bourne" movies. The confidence of the rule is 1/2, since there are four baskets that contain both BI and BS, and two of these also contain BU. The rule is slightly interesting in the positive direction. That is, BU appears in 3/8 of all baskets, but appears in 1/2 of those baskets that contain the left side of the association rule. □

As long as high support is a significant requirement for a useful association rule, the search for high-confidence or high-interest association rules is really the search for high-support itemsets. Once we have these itemsets, we can consider each member of an itemset as the item on the right of the association rule. We may, as part of the process of finding frequent itemsets, already have computed the counts of baskets for the subsets of this frequent itemset, since they also must be frequent. If so, we can compute easily the confidence and interest of each potential association rule. We shall thus, in what follows, leave aside the problem of finding association rules and concentrate on efficient methods for finding frequent itemsets.

22.1.4 The Computation Model for Frequent Itemsets

Since we are studying database systems, our first thought might be that the market-basket data is stored in a relation such as:

```
Baskets(basket, item)
```

consisting of pairs that are a basket ID and the ID of one of the items in that basket. In principle, we could find frequent itemsets by a SQL query. For instance, the query in Fig. 22.2 finds all frequent pairs. It joins Baskets with itself, grouping the resulting tuples by the two items found in that tuple, and throwing away groups where the number of baskets is below the support threshold s. Note that the condition I.item < J.item in the WHERE-clause is there to prevent the same pair from being considered in both orders, or for a "pair" consisting of the same item twice from being considered at all.

```
SELECT I.item, J.item, COUNT(I.basket)
FROM Baskets I, Baskets J
WHERE I.basket = J.basket AND
      I.item < J.item
GROUP BY I.item, J.item
HAVING COUNT(I.basket) >= s;
```

Figure 22.2: Naive way to find all high-support pairs of items

However, if the size of the Baskets relation is very large, the join of the relation with itself will be too large to construct, or at least too time-consuming to construct. No matter how efficiently we compute the join, the result relation contains one tuple for each pair of items in a basket. For instance, if there are 1,000,000 baskets, and each basket contains 20 items, then there will be 190,000,000 tuples in the join [since $\binom{20}{2} = 190$]. We shall see in Section 22.2 that it is often possible to do much better by preprocessing the Baskets relation.

But in fact, it is not common to store market-basket data as a relation. It is far more efficient to put the data in a file or files consisting of the baskets, in some order. A basket is represented by a list of its items, and there is some punctuation between baskets.

Example 22.3: The data of Fig. 22.1 could be represented by a file that begins:

```
{BI,BS,BU}{HP1,HP2,HP3}{BI,HP1}{BI,BS,HP1,HP2}{...
```

Here, we are using brackets to surround baskets and commas to separate items within a basket. □

When market-basket data is represented this way, the cost of an algorithm is relatively simple to estimate. Since we are interested only in cases where the data is too large to fit in main memory, we can count disk-I/O's as our measure of complexity.

However, the matter is even simpler than disk-I/O's. All the successful algorithms for finding frequent itemsets read the data file several times, in the order given. They thus make several passes over the data, and the information preserved from one pass to the next is small enough to fit in main memory. Thus, we do not even have to count disk-I/O's; it is sufficient to count the number of passes through the data.

22.1.5 Exercises for Section 22.1

Exercise 22.1.1: Suppose we are given the eight "market baskets" of Fig. 22.3.

$$\begin{aligned}
B_1 &= \{\text{milk, coke, beer}\}\\
B_2 &= \{\text{milk, pepsi, juice}\}\\
B_3 &= \{\text{milk, beer}\}\\
B_4 &= \{\text{coke, juice}\}\\
B_5 &= \{\text{milk, pepsi, beer}\}\\
B_6 &= \{\text{milk, beer, juice, pepsi}\}\\
B_7 &= \{\text{coke, beer, juice}\}\\
B_8 &= \{\text{beer, pepsi}\}
\end{aligned}$$

Figure 22.3: Example market-basket data

a) As a percentage of the baskets, what is the support of the set {beer, juice}?

b) What is the support of the itemset {coke, pepsi}?

c) What is the confidence of milk given beer (i.e., of the association rule {beer} ⇒ milk)?

d) What is the confidence of juice given milk?

e) What is the confidence of coke, given beer and juice?

f) If the support threshold is 37.5% (i.e., 3 out of the eight baskets are needed), which pairs of items are frequent?

g) If the support threshold is 50%, which pairs of items are frequent?

! h) What is the most interesting association rule with a singleton set on the left?

22.2 Algorithms for Finding Frequent Itemsets

We now look at how many passes are needed to find frequent itemsets of a certain size. We first argue why, in practice, finding frequent pairs is often the bottleneck. Then, we present the A-Priori Algorithm, a key step in minimizing the amount of main memory needed for a multipass algorithm. Several improvements on A-Priori make better use of main memory on the first pass, in order to make it more feasible to complete the algorithm without exceeding the capacity of main memory on later passes.

22.2.1 The Distribution of Frequent Itemsets

If we pick a support threshold $s = 1$, then all itemsets that appear in any basket are "frequent," so just producing the answer could be infeasible. However, in applications such as managing sales at a store, a small support threshold is not useful. Recall that we need many customers buying a set of items before we can exploit that itemset. Moreover, any data mining of market-basket data must produce a small number of answers, say tens or hundreds. If we get no answers, we cannot act, but if we get millions of answers, we cannot read them all, let alone act on them all.

The consequence of this reasoning is that the support threshold must be set high enough to make few itemsets frequent. Typically, a threshold around 1% of the baskets is used. Since the probability of an itemset being frequent goes down rapidly with size, most frequent itemsets will be small. However, an itemset of size one is generally not useful; we need at least two items in a frequent itemset in order to apply the marketing techniques mentioned in Section 22.1.1, for example.

Our conclusion is that in practical uses of algorithms to find frequent itemsets, we need to use a support threshold so that there will be a small number of frequent pairs, and very few frequent itemsets that are larger. Thus, our algorithms will focus on how to find frequent pairs in a few passes through the data. If larger frequent itemsets are wanted, the computing resources used to find the frequent pairs are usually sufficient to find the small number of frequent triples, quadruples, and so on.

What if Items Aren't Numbered Conveniently

We assume that items have integer ID's starting at 0. However, in practice, items could be represented by long ID's or by their full names. If so, we need to keep in main memory a hash table that maps each true item-ID to a unique integer in the range 0 to $k - 1$. This table consumes main memory proportional to the number of items k. No algorithm for finding frequent pairs or larger itemsets works if the number of items is not small compared with the available main memory. Thus, we neglect the possible need for a main-memory table whose size is proportional to the number of items.

22.2.2 The Naive Algorithm for Finding Frequent Itemsets

Let us suppose that there is some fixed number of bytes of main memory M, perhaps a gigabyte, or 16 gigabytes, or whatever our machine has. Let there be k different items in our market-basket dataset, and assume they are numbered $0, 1, \ldots, k - 1$. Finally, as suggested in Section 22.2.1, we shall focus on the counting of pairs, assuming that is the bottleneck for memory use.

If there is enough room in main memory to count all the pairs of items as we make a single pass over the baskets, then we can solve the frequent-pairs problem in a single pass. In that pass, we read one block of the data file at a time. We shall neglect the amount of main memory needed to hold this block (or even several blocks if baskets span two or more blocks), since we may assume that the space needed to represent a basket is tiny compared with M. For each basket found on this block, we execute a double loop through its items and for each pair of items in the basket, we add one to the count for that pair.

The essential problem we face, then, is how do we store the counts of the pairs of items in M bytes of memory. There are two reasonable ways to do so, and which is better depends on whether it is common or unlikely that a given pair of items occurs in at least one basket. In what follows, we shall make the simplifying assumption that all integers, whether used for a count or to represent an item, require four bytes. Here are the two contending approaches to maintaining counts.

Triangular Matrix

If most of the possible pairs of items are expected to appear at least once in the dataset, then the most efficient use of main memory is a triangular array. That is, let a be a one-dimensional integer array occupying all available main memory. We count the pair (i, j), where $0 \leq i < j < k$ in $a[n]$, where:

$$\begin{array}{ll} n = (i+j)^2/4 + i - 1/4 & \text{if } i+j \text{ is odd} \\ n = (i+j)^2/4 + i & \text{if } i+j \text{ is even} \end{array}$$

As long as $M \geq 2k^2$, there is enough room to store array a, with four bytes per count. Notice that this method takes only half the space that would be used by a square array, of which we used only the upper or lower triangle to count the pairs (i, j) where $i < j$.

Table of Counts

If the probability of a pair of items ever occurring is small, then we can do with space less than $O(k^2)$. We instead construct a hash table of triples (i, j, c), where $i < j$ and $\{i, j\}$ is one of the itemsets that actually occurs in one or more of the baskets. Here, c is the count for that pair. We hash the pair (i, j) to find the bucket in which the count for that itemset is kept.

A triple (i, j, c) requires 12 bytes, so we can maintain counts for $M/12$ pairs.[2] Put another way, if p pairs ever occur in the data, we need main memory at least $M \geq 12p$.

Notice that there are approximately $k^2/2$ possible pairs if there are k different items. If the number of pairs $p = k^2/2$, then the table of counts requires three times as much main memory as the triangular matrix. However, if only 1/3 of all possible pairs occur, then the two methods have the same memory requirements, and if the probability that a given pair occurs is less than 1/3, then the table of counts is preferable.

Additional Comments About the Naive Algorithm

In summary, we can use the naive, one-pass algorithm to find all frequent pairs if the number of bytes of main memory M exceeds either $2k^2$ or $12p$, where k is the number of different items and p is the number of pairs of items that occur in at least one basket of the dataset.

The same approach can be used to count triples, provided that there is enough memory to count either all possible triples or all triples that actually occur in the data. Likewise, we can count quadruples or itemsets of any size, although the likelihood that we have enough memory goes down as the size goes up. We leave the formulas for how much memory is needed as an exercise.

22.2.3 The A-Priori Algorithm

The A-Priori Algorithm is a method for finding frequent itemsets of size n, for any n, in n passes. It normally uses much less main memory than the naive algorithm, and it is certain to use less memory if the support threshold is sufficiently high that some singleton sets are not frequent. The important

[2]Whatever kind of hash table we use, there will be some additional overhead, which we shall neglect. For example, if we use open addressing, then it is generally necessary to leave a small fraction of the buckets unfilled, to limit the average search for a triple.

insight that makes the algorithm work is *monotonicity* of the property of being frequent. That is:

- If an itemset S is frequent, so is each of its subsets.

The truth of the above statement is easy to see. If S is a subset of at least s baskets, where s is the support threshold, and $T \subseteq S$, then T is also a subset of the same baskets that contain S, and perhaps T is a subset of other baskets as well. The use of monotonicity is actually in its contrapositive form:

- If S is not a frequent itemset, then no superset of S is frequent.

On the first pass, the a-priori algorithm counts only the singleton sets of items. If some of those sets are not frequent by themselves, then their items cannot be part of any frequent pair. Thus, the nonfrequent items can be ignored on a second pass through the data, and only the pairs consisting of two frequent items need be counted. For example, if only half the items are frequent, then we need to count only 1/4 of the number of pairs, so we can use 1/4 as much main memory. Or put another way, with a fixed amount of main memory, we can deal with a dataset that has twice as many items.

We can continue to construct the frequent triples on another pass, the frequent quadruples on the fourth pass, and so on, as high as we like and that frequent itemsets exist. The generalization is that for the nth pass we begin with a *candidate set* of itemsets C_n, and we produce a subset F_n of C_n consisting of the frequent itemsets of size n. That is, C_1 is the set of all singletons, and F_1 is those singletons that are frequent. C_2 is the set of pairs of items, both of which are in F_1, and F_2 is those pairs that are frequent. The candidate set for the third pass, C_3, is those triples $\{i, j, k\}$ such that each doubleton subset, $\{i, j\}$, $\{i, k\}$, and $\{j, k\}$, is in F_2. The following gives the algorithm formally.

Algorithm 22.4: A-Priori Algorithm.

INPUT: A file D consisting of baskets of items, a support threshold s, and a size limit q for the size of frequent itemsets.

OUTPUT: The sets of itemsets $F_1, F_2, \ldots, F_q$, where F_i is the set of all itemsets of size i that appear in at least s baskets of D.

METHOD: Execute the algorithm of Fig. 22.4 and output each set F_n of frequent items, for $n = 1, 2, \ldots, q$. □

Example 22.5: Let us execute the A-Priori Algorithm on the data of Fig. 22.1 with support $s = 4$. Initially, C_1 is the set of all six movies. In the first pass, we count the singleton sets, and we find that BI, BS, $HP1$, and $HP2$ occur at least four times; the other two movies do not. Thus, $F_1 = \{BI, BS, HP1, HP2\}$, and C_2 is the set of six pairs that can be formed by choosing two of these four movies.

```
1) LET $C_1$ = all items that appear in file $F$;
2) FOR $n$ := 1 TO $q$ DO BEGIN
3)     $F_n$ := those sets in $C_n$ that occur at least
           $s$ times in $D$;
4)     IF $n$ = $q$ BREAK;
5)     LET $C_{n+1}$ = all itemsets $S$ of size $n+1$ such that
           every subset of $S$ of size $n$ is in $F_n$;
   END
```

Figure 22.4: The A-Priori Algorithm

On the second pass, we count only these six pairs, and we find that $F_2 = \{\{BI, BS\}, \{HP1, HP2\}, \{BI, HP1\}, \{BS, HP1\}\}$; the other two pairs are not frequent. Assuming $q > 2$, we try to find frequent triples. C_3 consists of only the triple $\{BI, BS, HP1\}$, because that is the only set of three movies, all pairs of which are in F_2. However, these three movies appear together only in three rows: V_4, V_5, and V_8. Thus, F_3 is empty, and there are no more frequent itemsets, no matter how large q is. The algorithm returns $F_1 \cup F_2$. □

22.2.4 Implementation of the A-Priori Algorithm

Figure 22.4 is just an outline of the algorithm. We must consider carefully how the steps are implemented. The heart of the algorithm is line (3), which we shall implement, each time through, by a single pass through the input data. The let-statements of lines (1) and (5) are just definitions of what C_n is, rather than assignments to be executed. That is, as we run through the baskets in line (3), the definition of C_n tells us which sets of size n need to be counted in main memory, and which need not be counted.

The algorithm should be used only if there is enough main memory to satisfy the requirements to count all the candidate sets on each pass. If there is not enough memory, then either a more space-efficient algorithm must be used, or several passes must be used for a single value of n. Otherwise, the system will "thrash," with pages being moved in and out of main memory during a pass, thus greatly increasing the running time.

We can use either method discussed in Section 22.2.2 to organize the main-memory counts during a pass. It may not be obvious that the triangular-matrix method can be used with a-priori on the second pass, since the frequent items are not likely to have numbers $0, 1, \ldots$ up to as many frequent items as there are. However, after finding the frequent items on pass 1, we can construct a small main-memory table, no larger than the set of items itself, that translates the original items numbers into consecutive numbers for just the frequent items.

22.2.5 Making Better Use of Main Memory

We expect that the memory bottleneck comes on the second pass of Algorithm 22.4, that is, at the execution of line (3) of Fig. 22.4 with $n = 2$. That is, we assume counting candidate pairs takes more space than counting candidate triples, quadruples, and so on. Thus, let us concentrate on how we could reduce the number of candidate pairs for the second pass. To begin, the typical use of main memory on the first two passes of the A-Priori Algorithm is suggested by Fig. 22.5.

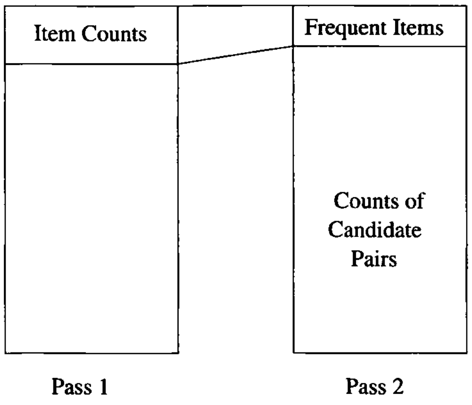

Figure 22.5: Main-memory use by the A-Priori Algorithm

On the first pass ($n = 1$), all we need is space to count all the items, which is typically very small compared with the amount of memory needed to count pairs. On the second pass ($n = 2$), the counts are replaced by a list of the frequent items, which is expected to take even less space than the counts took on the first pass. All the available memory is devoted, as needed, to counts of the candidate pairs.

Could we do anything with the unused memory on the first pass, in order to reduce the number of candidate pairs on the second pass? If so, data sets with larger numbers of frequent pairs could be handled on a machine with a fixed amount of main memory. The *PCY Algorithm*[3] exploits the unused memory by filling it entirely with an unusual sort of hash table. The "buckets" of this table do not hold pairs or other elements. Rather, each bucket is a single integer count, and thus occupies only four bytes. We could even use two-byte buckets if the support threshold were less than 2^{16}, since once a count gets above the threshold, we do not need to see how large it gets.

During the first pass, as we examine each basket, we not only add one to the count for each item in the basket, but we also hash each pair of items to its bucket in the hash table and add one to the count in that bucket. What we

[3]For the authors, J. S. Park, M.-S. Chen, and P. S. Yu.

hope for is that some buckets will wind up with a count less than s, the support threshold. If so, we know that no pair $\{i, j\}$ that hashes to that bucket can be frequent, even if both i and j are frequent as singletons.

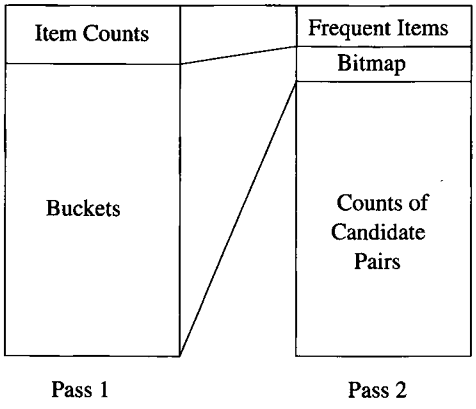

Figure 22.6: Main-memory use by the PCY Algorithm

Between the first and second passes, we replace the buckets by a *bitmap* with one bit per bucket. The bit is 1 if the corresponding bucket is a *frequent bucket*; that is, its count is at least the support threshold s; otherwise the bit is 0. A bucket, occupying 32 bits (4 bytes) is replaced by a single bit, so the bitmap occupies roughly 1/32 of main memory on the second pass. There is thus almost as much space available for counts on the second pass of the PCY Algorithm as there is for the A-Priori Algorithm. Figure 22.6 illustrates memory use during the first two passes of PCY.

On the second pass, $\{i, j\}$ is a candidate pair if and only if the following conditions are satisfied:

1. Both i and j are frequent items.

2. $\{i, j\}$ hashes to a bucket that the bitmap tells us is a frequent bucket.

Then, on the second pass, we can count only this set of candidate pairs, rather than all the pairs that meet the first condition, as in the A-Priori Algorithm.

22.2.6 When to Use the PCY Algorithm

In the PCY Algorithm, the set of candidate pairs is sufficiently irregular that we cannot use the triangular-matrix method for organizing counts; we must use a table of counts. Thus, it does not make sense to use PCY unless the number of candidate pairs is reduced to at most 1/3 of all possible pairs. Passes of the PCY Algorithm after the second can proceed just as in the A-Priori Algorithm, if they are needed.

Further, in order for PCY to be an improvement over A-Priori, a good fraction of the buckets on the first pass must not be frequent. For if most buckets are frequent, condition (3) above does not eliminate many pairs. Any bucket to which even one frequent pair hashes will itself be frequent. However, buckets to which no frequent pair hashes could still be frequent if the sum of the counts of the pairs that do hash there exceeds the threshold s. To a first approximation, if the average count of a bucket is less then s, we can expect at least half the buckets not to be frequent, which suggests some benefit from the PCY approach. However, if the average bucket has a count above s, then most buckets will be frequent.

Suppose the total number of occurrences of pairs of items among all the baskets in the dataset is P. Since most of the main memory M can be devoted to buckets, the number of buckets will be approximately $M/4$. The average count of a bucket will then be $4P/M$. In order that there be many buckets that are not frequent, we need $4P/M < s$, or $M > 4P/s$. The exercises allow you to explore some more concrete examples.

22.2.7 The Multistage Algorithm

Instead of counting pairs on the second pass, as we do in A-Priori or PCY, we could use the same bucketing technique (with a different hash function) on the second pass. To make the average counts even smaller on the second pass, we do not even have to consider a pair on the second pass unless it would be counted on the second pass of PCY; that is, the pair consists of two frequent items and also hashed to a frequent bucket on the first pass.

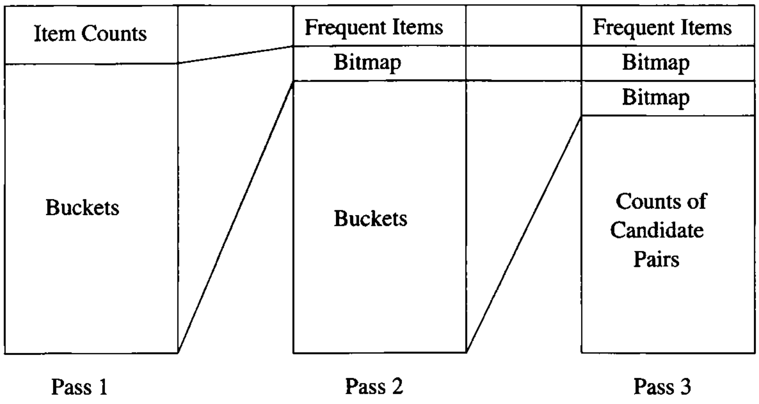

Figure 22.7: Main-memory use in the three-pass version of the multistage algorithm

This idea leads to the three-pass version of the *Multistage Algorithm* for finding frequent pairs. The algorithm is sketched in Fig. 22.7. Pass 1 is just like Pass 2 of PCY, and between Passes 1 and 2 we collapse the buckets to bits and select the frequent items, also as in PCY.

However, on Pass 2, we again use all available memory to hash pairs into as many buckets as will fit. Because there is a bitmap to store in main memory on the second pass, and this bitmap compresses a 4-byte (32-bit) integer into one bit, there will be approximately 31/32 as many buckets on the second pass as on the first. On the second pass, we use a different hash function from that used on Pass 2. We hash a pair $\{i, j\}$ to a bucket and add one to the count there if and only if:

1. Both i and j are frequent items.

2. $\{i, j\}$ hashed to a frequent bucket on the first pass. This decision is made by consulting the bitmap.

That is, we hash only those pairs we would count on the second pass of the PCY Algorithm.

Between the second and third passes, we condense the buckets of the second pass into another bitmap, which must be stored in main memory along with the first bitmap and the set of frequent items. On the third pass, we finally count the candidate pairs. In order to be a candidate, the pair $\{i, j\}$ must satisfy all of:

1. Both i and j are frequent items.

2. $\{i, j\}$ hashed to a frequent bucket on the first pass. This decision is made by consulting the first bitmap.

3. $\{i, j\}$ hashed to a frequent bucket on the second pass. This decision is made by consulting the second bitmap.

As with PCY, subsequent passes can construct frequent triples or larger itemsets, if desired, using the same method as A-Priori.

The third condition often eliminates many pairs that the first two conditions let through. One reason is that on the second pass, not every pair is hashed, so the counts of buckets tend to be smaller than on the first pass, resulting in many more infrequent buckets. Moreover, since the hash functions on the first two passes are different, infrequent pairs that happened to hash to a frequent bucket on the first pass have a good chance of hashing to an infrequent bucket on the second pass.

The Multistage Algorithm is not limited to three passes for computation of frequent pairs. We can have a large number of bucket-filling passes, each using a different hash function. As long as the first pass eliminates some of the pairs because they belong to a nonfrequent bucket, then subsequent passes will eliminate a rapidly growing fraction of the pairs, until it is very unlikely that any candidate pair will turn out not to be frequent. However, there is a point of diminishing returns, since each bitmap requires about 1/32 of the memory.

If we use too many passes, not only will the algorithm take more time, but we can find ourselves with available main memory that is too small to count all the frequent pairs.

22.2.8 Exercises for Section 22.2

Exercise 22.2.1: Simulate the A-Priori Algorithm on the data of Fig. 22.3, with $s = 3$.

! **Exercise 22.2.2:** Suppose we want to count all itemsets of size n using one pass through the data.

a) What is the generalization of the triangular-matrix method for $n > 2$? Give the formula for locating the array element that counts a given set of n elements $\{i_1, i_2, \dots, i_n\}$.

b) How much main memory does the generalized triangular-matrix method take if there are k items?

c) What is the generalization of the table-of-counts method for $n > 2$?

d) How much main memory does the generalized table-of-counts method take if there are p itemsets of size n that appear in the data?

Exercise 22.2.3: Imagine that there are 1100 items, of which 100 are "big" and 1000 are "little." A basket is formed by adding each big item with probability 1/10, and each little item with probability 1/100. Assume the number of baskets is large enough that each itemset appears in a fraction of the baskets that equals its probability of being in any given basket. For example, every pair consisting of a big item and a little item appears in 1/1000 of the baskets. Let s be the support threshold, but expressed as a fraction of the total number of baskets rather than as an absolute number. Give, as a function of s ranging from 0 to 1, the number of frequent items on Pass 1 of the A-Priori Algorithm. Also, give the number of candidate pairs on the second pass.

! **Exercise 22.2.4:** Consider running the PCY Algorithm on the data of Exercise 22.2.3, with 100,000 buckets on the first pass. Assume that the hash function used distributes the pairs to buckets in a conveniently random fashion. Specifically, the 499,500 little-little pairs are divided as evenly as possible (approximately 5 to a bucket). One of the 100,000 big-little pairs is in each bucket, and the 4950 big-big pairs each go into a different bucket.

a) As a function of s, the ratio of the support threshold to the total number of baskets (as in Exercise 22.2.3), how many frequent buckets are there on the first pass?

b) As a function of s, how many pairs must be counted on the second pass?

! **Exercise 22.2.5:** Using the assumptions of Exercise 22.2.4, suppose we run a three-pass Multistage Algorithm on the dataset. Assuming that on the second pass there are again 100,000 buckets, and the hash function distributes pairs randomly among the buckets, answer the following questions, all in terms of s the ratio of the support threshold to the number of baskets.

a) Approximately how many frequent buckets will there be on the second pass?

b) Approximately how many pairs are counted on the third pass?

Exercise 22.2.6: Suppose baskets are in a file that is distributed over many processors. Show how you would use the map-reduce framework of Section 20.2 to:

a) Find the counts of all items.

! b) Find the counts of all pairs of items.

22.3 Finding Similar Items

We now turn to the version of the frequent-itemsets problem that supports marketing activities for on-line merchants and a number of other interesting applications such as finding similar documents on the Web. We may start with the market-basket model of data, but now we search for pairs of items that appear together a large fraction of the times that either appears, even if neither item appears in very many baskets. Such items are said to be *similar*. The key technique is to create a short "signature" for each item, such that the difference between signatures tells us the difference between the items themselves.

22.3.1 The Jaccard Measure of Similarity

Our starting point is to define exactly what we mean by "similar" items. Since we are interested in finding items that tend to appear together in the same baskets, the natural viewpoint is that each item is a set: the set of baskets in which it appears. Thus, we need a definition for how similar two sets are.

The *Jaccard similarity* (or just *similarity*, if this similarity measure is understood) of sets S and T is $|S \cap T|/|S \cup T|$, that is, the ratio of the sizes of their intersection and union. Thus, disjoint sets have a similarity of 0, and the similarity of a set with itself is 1. As another example, the similarity of sets $\{1,2,3\}$ and $\{1,3,4,5\}$ is 2/5, since there are two elements in the intersection and five elements in the union.

22.3.2 Applications of Jaccard Similarity

A number of important data-mining problems can be expressed as finding sets with high Jaccard similarity. We shall discuss two of them in detail here.

Collaborative Filtering

Suppose we are given data about customers' on-line purchases. One way to tell what items to pitch to a customer is to find pairs of customers that bought similar sets of items. When a customer logs in, they can be pitched an item that a similar customer bought, but that they did not buy. To compare customers, represent a customer by the set of items they bought, and compute the Jaccard similarity for each pair of customers.

There is a dual view of the same data. We might want to know which pairs of items are similar, based on their having been bought by similar sets of customers. We can frame this problem in the same terms as finding similar customers. Now, the items are represented by the set of customers that bought them, and we need to find pairs of items that have similar sets of customers.

Notice, incidentally, that the same data can be viewed as market-basket data in two different ways. The products can be the "items" and the customers the "baskets," or vice-versa. You should not be surprised. Any many-many relationship can be seen as market-basket data in two ways. In Section 22.1 we viewed the data in only one way, because when the "baskets" are really shopping carts at a store's checkout stand, there is no real interest in finding similar shopping carts or carts that contain many items in common.

Similar Documents

There are many reasons we would like to find pairs of textually similar documents. If we are crawling the Web, documents that are very similar might be mirrors of one another, perhaps differing only in links to other documents at the local site. A search engine would not want to offer both sites in response to a search query. Other similar pairs might represent an instance of plagiarism. Note that one document d_1 might contain an excerpt from another document d_2, yet d_1 and d_2 are identical in only 10% of each; that could still be an instance of plagiarism.

Telling whether documents are character-for-character identical is easy; just compare characters until you find a mismatch or reach the ends of the documents. Finding whether a sentence or short piece of text appears character-for-character in a document is not much harder. Then you have to consider all places in the document where the sentence of fragment might start, but most of those places will have a mismatch very quickly. What is harder is to find documents that are similar, but are not exact copies in long stretches. For instance, a draft document and its edited version might have small changes in almost every sentence.

A technique that is almost invulnerable to large numbers of small changes is to represent a document by its set of *k-grams*, that is, by the set of substrings of length k. *k-Shingle* is another word for k-gram. For example, the set of 3-grams that we find in the first sentence of Section 22.3.2 ("A number of$\cdots$") contains `"A n"`, `" nu"`, `"num"`, and so on. If we pick k large enough so that the probability of a randomly chosen k-gram appearing in a document is small,

Compressed Shingles

In order that a document be characterized by its set of k-shingles, we have to pick k sufficiently large that it is rare for a given shingle to appear in a document. $k = 5$ is about the smallest we can choose, and it is not unusual to have k around 10. However, then there are so many possible shingles, and the shingles are so long, that certain algorithms take more time than necessary. Therefore, it is common to hash the shingles to integers of 32 bits or less. These hash-values are still numerous enough that they differentiate between documents, but they can be compared and processed quickly.

then a high Jaccard similarity of the sets of k-grams representing a pair of documents is a strong indication that the documents themselves are similar.

22.3.3 Minhashing

Computing the Jaccard similarity of two large sets is time consuming. Moreover, even if we can compute similarities efficiently, a large dataset has far too many pairs of sets for us to compute the similarity of every pair. Thus, there are two "tricks" we need to learn to extract only the similar pairs from a large dataset. Both are a form of "hashing," although the techniques are completely different uses of hashing.

1. *Minhashing* is a technique that lets us form a short signature for each set. We can compute the Jaccard similarity of the sets by measuring the similarity of the signatures. As we shall see, the "similarity" for signatures is simple to compute, but it is not the Jaccard similarity. We take up minhashing in this section.

2. *Locality-Sensitive Hashing* is a technique that lets us focus on pairs of signatures whose underlying sets are likely to be similar, without examining all pairs of signatures. We take up locality-sensitive hashing in Section 22.4.

To introduce minhashing, suppose that the elements of each set are chosen from a "universal" set of n elements $e_0, e_1, \ldots, e_{n-1}$. Pick a random permutation of the n elements. Then the *minhash value* of a set S is the first element, in the permuted order, that is a member of S.

Example 22.6: Suppose the universal set of elements is $\{1, 2, 3, 4, 5\}$ and the permuted order we choose is $(3, 5, 4, 2, 1)$. Then the hash value of any set that contains 3, such as $\{2, 3, 5\}$ is 3. A set that contains 5 but not 3, such as $\{1, 2, 5\}$, hashes to 5. For another example, $\{1, 2\}$ hashes to 2, because 2 appears before 1 in the permuted order. $\square$

Suppose we have a collection of sets. For example, we might be given a collection of documents and think of each document as represented by its set of 10-grams. We compute *signatures* for the sets by picking a list of m permutations of all the possible elements (e.g., all possible character strings of length 10, if the elements are 10-grams). Typically, m would be about 100. The signature of a set S is the list of the minhash values of S, for each of the m permutations, in order.

Example 22.7: Suppose the universal set of elements is again $\{1,2,3,4,5\}$, and choose $m = 3$, that is, signatures of three minhash values. Let the permutations be $\pi_1 = (1,2,3,4,5)$, $\pi_2 = (5,4,3,2,1)$, and $\pi_3 = (3,5,1,4,2)$. The signature of $S = \{2,3,4\}$ is $(2,4,3)$. To see why, first notice that in the order π_1, 2 appears before 3 and 4, so 2 is the first minhash value. In π_2, 4 appears before 2 and 3, so 4 is the second minhash value. In π_3, 3 appears before 2 and 4, so 3 is the third minhash value. □

22.3.4 Minhashing and Jaccard Distance

There is a surprising relationship between the minhash values and the Jaccard similarity:

- If we choose a permutation at random, the probability that it will produce the same minhash values for two sets is the same as the Jaccard similarity of those sets.

Thus, if we have the signatures of two sets S and T, we can estimate the Jaccard similarity of S and T by the fraction of corresponding minhash values for the two sets that agree.

Example 22.8: Let the permutations be as in Example 22.7, and consider another set, $T = \{1,2,3\}$. The signature for T is $(1,3,3)$. If we compare this signature with $(2,4,3)$, the signature of the set $S = \{2,3,4\}$, we see that the signatures agree in only the last of the three components. We therefore estimate the Jaccard similarity of S and T to be $1/3$. Notice that the true Jaccard similarity of S and T is $1/2$. □

In order that the signatures are very likely to estimate the similarity closely, we need to pick considerably more than three permutations. We suggest that 100 permutations may be enough for the "law of large numbers" to hold. However, the exact number of signatures needed depends on how closely we need to estimate the similarity.

22.3.5 Why Minhashing Works

To see why the Jaccard similarity is the probability that two sets have the same minhash value according to a randomly chosen permutation of elements, let S

and T be two sets. Imagine going down the list of elements in the permuted order, until you find an element e that appears in at least one of S and T. There are two cases:

1. If e appears in both S and T, then both sets have the same minhash value, namely e.

2. But if e appears in one of S and T but not the other, then one set gets minhash value e and the other definitely gets some other minhash value.

We do not meet e until the first time we find, in the permuted order, an element that is in $S \cup T$. The probability of Case 1 occuring is the fraction of members of $S \cup T$ that are in $S \cap T$. That fraction is exactly the Jaccard similarity of S and T. But Case 1 is also exactly when S and T have the same minhash value, which proves the relationship.

22.3.6 Implementing Minhashing

While we have spoken of choosing a random permutation of all possible elements, it is not feasible to do so. It would take far too long, and we might have to deal with elements that appeared in none of our sets. Rather, we simulate the choice of a random permutation by instead picking a random hash function h from elements to some large sequence of integers $0, 1, \ldots, B-1$ (i.e., bucket numbers). We pretend that the permutation that h represents places element e in the position $h(e)$. Of course, several elements might thus wind up in the same position, but as long as B is large, we can break ties as we like, and the simulated permutations will be sufficiently random that the relationship between signatures and similarity still holds.

Suppose our dataset is presented one set at a time. To compute the minhash value for a set $S = \{a_1, a_2, \ldots, a_n\}$ using a hash function h, we can execute:

```
$V$ := $\infty$;
FOR $i$ := 1 TO $n$ DO
    IF $h(a_i) < V$ THEN $V$ := $h(a_i)$;
```

As a result, V will be set to the hash value of the element of S that has the smallest hash value. This hash value may not identify a unique element, because several elements in the universe of possible elements may hash to this value, but as long as h hashes to a large number of possible values, the chances of a coincidence is small, and we may continue to assume that a common minhash value suggests two sets have an element in common.

If we want to compute not just one minhash value but the minhash values for set S according to m hash functions $h_1, h_2, \ldots, h_m$, then we can compute m minhash values in parallel, as we process each member of S. The code is suggested in Fig. 22.8.

```
FOR $j$ := 1 TO $m$ DO
    $V_j$ := $\infty$;
FOR $i$ := 1 TO $n$ DO
    FOR $j$ := 1 TO $m$ DO
        IF $h_j(a_i) < V_j$ THEN $V_j$ := $h_j(a_i)$;
```

Figure 22.8: Computing m minhash values at once

It is somewhat harder to compute signatures if the data is presented basket-by-basket as in Section 22.1. That is, suppose we want to compute the signatures of "items," but our data is in a file consisting of baskets. Similarity of items is the Jaccard similarity of the sets of baskets in which these items appear.

Suppose there are k items, and we want to construct their minhash signatures using m different hash functions $h_1, h_2, \ldots, h_m$. Then we need to maintain km values, each of which will wind up being the minhash value for one of the items according to one of the hash functions. Let V_{ij} be the value for item i and hash function h_j. Initially, set all V_{ij}'s to infinity. When we read a basket b, we compute $h_j(b)$ for all $j = 1, 2, \ldots, m$. However, we adjust values only for those items i that are in b. The algorithm is sketched in Fig. 22.9. At the end, V_{ij} holds the jth minhash value for item i.

```
FOR $i$ := 1 TO $k$ DO
    FOR $j$ := 1 TO $m$ DO
        $V_{ij}$ := $\infty$;
FOR EACH basket $b$ DO BEGIN
    FOR $j$ := 1 TO $m$ DO
        compute $h_j(b)$;
    FOR EACH item $i$ in $b$ DO
        FOR $j$ := 1 TO $m$ DO
            IF $h_j(b) < V_{ij}$ THEN $V_{ij}$ := $h_j(b)$;
    END
```

Figure 22.9: Computing minhash values for all items and hash functions

22.3.7 Exercises for Section 22.3

Exercise 22.3.1: Compute the Jaccard similarity of each pair of the following sets: $\{1,2,3,4,5\}$, $\{1,6,7\}$, $\{2,4,6,8\}$.

Exercise 22.3.2: What are all the 4-grams of the following string:

```
"abc def ghi"
```

Do not count the quotation marks as part of the string, but remember that blanks do count.

Exercise 22.3.3: Suppose that the universal set is $\{1, 2, \dots, 10\}$, and signatures for sets are constructed using the following list of permutations:

1. $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$
2. $(10, 8, 6, 4, 2, 9, 7, 5, 3, 1)$
3. $(4, 7, 2, 9, 1, 5, 3, 10, 6, 8)$

Construct minhash signatures for the following sets:

a) $\{3, 6, 9\}$.

b) $\{2, 4, 6, 8\}$

c) $\{2, 3, 4\}$

How does the estimate of the Jaccard similarity for each pair, derived from the signatures, compare with the true Jaccard similarity?

Exercise 22.3.4: Suppose that instead of using particular permutations to construct signatures for the three sets of Exercise 22.3.3, we use hash functions to construct the signatures. The three hash functions we use are:

$$\begin{aligned} f(x) &= x \mod 10 \\ g(x) &= (2x + 1) \mod 10 \\ h(x) &= (3x + 2) \mod 10 \end{aligned}$$

Compute the signatures for the three sets, and compare the resulting estimate of the Jaccard similarity of each pair with the true Jaccard similarity.

! **Exercise 22.3.5:** Suppose data is in a file that is distributed over many processors. Show how you would use the map-reduce framework of Section 20.2 to compute a minhash value, using a single hash function, assuming:

a) The file must be partitioned by rows.

b) The file must be partitioned by columns.

22.4 Locality-Sensitive Hashing

Now, we take up the problem that was not really solved by taking minhash signatures. It is true that these signatures may make it much faster to estimate the similarity of any pair of sets, but there may still be far too many pairs of sets to find all pairs that meet a given similarity threshold. The technique called "locality-sensitive hashing," or LSH, may appear to be magic; it allows us, in a sense, to hash sets or other elements to buckets so that "similar" elements are assigned to the same bucket. There are tradeoffs, of course. There is a (typically small) probability that we shall miss a pair of similar elements, and the lower we want that probability to be, the more work we must do. After some examples, we shall take up the general theory.

22.4.1 Entity Resolution as an Example of LSH

Recall our discussion of entity resolution in Section 21.7. There, we had a large collection of records, and we needed to find similar pairs. The notion of "similarity" was not Jaccard similarity, and in fact we left open what "similarity" meant. Whatever definition we use for similarity of records, there may be far too many pairs to measure them all. For example, if there are a million records — not a very large number — then there are about 500 billion pairs of records. An algorithm like R-Swoosh may allow merging with fewer than that number of comparisons, provided there are many large sets of similar records, but if no records are similar to other records, then there is no way we can discover that fact without doing all possible comparisons.

It would be wonderful to have a way to "hash" records so that similar records fell into the same bucket, and nonsimilar pairs never did, or rarely did. Then, we could restrict our examination of pairs to those that were in the same bucket. If, say, there were 1000 buckets, and records distributed evenly, then we would only have to compare 1/1000 of the pairs. We cannot do exactly what is described above, but we can come surprisingly close.

Example 22.9: Suppose for concreteness that records are as in the running example of Section 21.7: name-address-phone triples, where each of the three fields is a character string. Suppose also that we define records to be similar if the sum of the edit distances of their three corresponding pairs of fields is no greater than 5. Let us use a hash function h that hashes the name field of a record to one of a million buckets. How h works is unimportant, except that it must be a good hash function — one that distributes names roughly uniformly among the buckets.

But we do not stop here. We also hash the records to another set of a million buckets, this time using the address, and a suitable hash function on addresses. If h operates on any strings, we can even use h. Then, we hash records a third time to a million buckets, using the phone number.

Finally, we examine each bucket in each of the three hash tables, a total of 3,000,000 buckets. For each bucket, we compare each pair of records in each bucket, and we report any pair that has total edit distance 5 or less. Suppose there are n records. Assuming even distribution of records in each hash table, there are $n/10^6$ records in each bucket. The number of pairs of records in each bucket is approximately $n^2/(2 \times 10^{12})$. Since there are 3×10^6 buckets, the total number of comparisons is about $1.5n^2/10^6$. And since there are about $n^2/2$ pairs of records, we have managed to look at only fraction 3×10^{-6} of the records, a big improvement.

In fact, since the number of buckets was chosen arbitrarily, it seems we can reduce the number of comparisons to whatever degree we wish. There are limitations, of course. If we choose too large a number of buckets, we run out of main-memory space, and regardless of how many buckets we use, we cannot avoid the pairs of records that are really similar.

Have we given up anything? Yes, we have; we shall miss some similar pairs of records that meet the similarity threshold, because they differ by a few characters in each of the three fields, yet no more than five characters in total. What fraction of the truly similar pairs we lose depends on the distribution of discrepancies among the fields of records that truly represent the same entity. However, if the threshold for total edit distance is 5, we do not expect to miss too many truly similar pairs. □

But what if the threshold on edit distance in Example 22.9 were not 5, but 20? There might be many pairs of similar records that had no one field identical. To deal with this problem, we need to:

1. Increase the number of hash functions and hash tables.

2. Base each hash function on a small part of a field.

Example 22.10: We could break the name into first, middle, and last names, and hash each to buckets. We could break the address into house number, street name, city name, state, and zip code. The phone number could be broken into area code, exchange, and the last four digits. Since phones are numbers, we could even choose any subset of the ten digits in a phone number, and hash on those. Unfortunately, since we are now hashing short subfields, we are limited in the number of buckets that we can use. If we pick too many buckets, most will be empty.

After hashing records many times, we again look in each bucket of each of the hash tables, and we compare each pair of records that fall into the same bucket at least once. However, the total running time is much higher than for our first example, for two reasons. First, the number of record occurrences among all the buckets is proportional to the number of hash functions we use. Second, hash functions based on small pieces of data cannot divide the records into as many buckets as in Example 22.9. □

22.4.2 Locality-Sensitive Hashing of Signatures

The use of locality-sensitive hashing in Example 22.10 is relatively straightforward. For a more subtle application of the general idea, let us return to the problem introduced in Section 22.3, where we saw the advantage of replacing sets by their signatures. When we need to find similar pairs of sets that are represented by signatures, there is a way to build hash functions for a locality-sensitive hashing, for any desired similarity threshold. Think of the signatures of the various sets as a matrix, with a column for each set's signature and a row for each hash function. Divide the matrix into b bands of r rows each, where br is the length of a signature. The arrangement is suggested by Fig. 22.10.

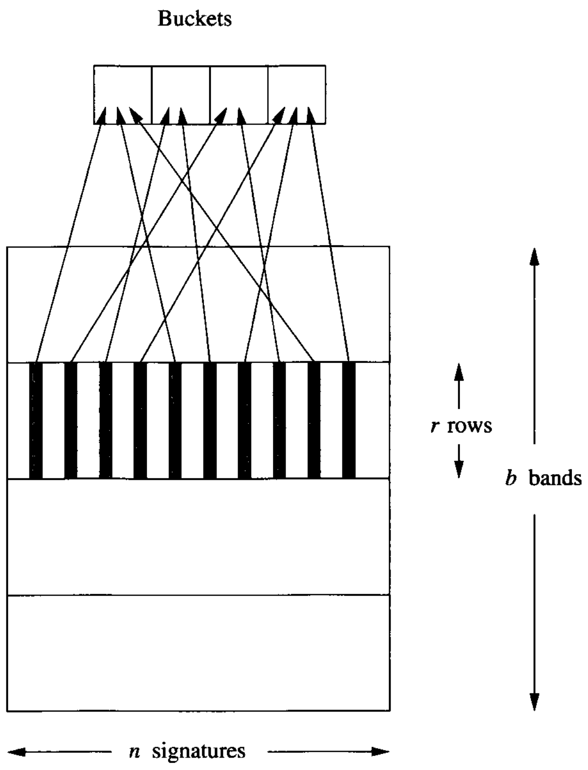

Figure 22.10: Dividing signatures into bands and hashing based on the values in a band

For each band we choose a hash function that maps the portion of a signature in that band to some large number of buckets, B. That is, the hash function applies to sequences of r integers and produces one integer in the range 0 to $B-1$. In Fig. 22.10, $B = 4$. If two signatures agree in all rows of any one band, then they surely will wind up in the same bucket. There is a small chance that they will be in the same bucket even if they do not agree, but by using a very large number of buckets B, we can make sure there are very few "false positives." Every bucket of each hash function has its members compared for similarity, so a pair of signatures that agree in even one band will be compared. Signatures that do not agree in any band probably will not be compared, although as we mentioned, there is a small probability they will hash to the same bucket anyway, and would therefore be compared.

Let us compute the probability that a pair of minhash signatures will be compared, as a function of the Jaccard similarity s of their underlying sets, the number of bands b, and the number of rows r in a band. For simplicity, we shall assume that the number of buckets is so large that there are no coincidences; signatures hash to the same bucket if and only if they have the same values in the entire band on which the hash function is based.

First, the probability that the signatures agree on one row is s, as we saw in Section 22.3.5. The probability that they agree on all r rows of a given band is s^r. The probability that they do not agree on all rows of a band is $1 - s^r$, and the probability that for none of the b bands do they agree in all rows of that band is $(1 - s^r)^b$. Finally, the probability that the signatures will agree in all rows of at least one band is $1 - (1 - s^r)^b$. This function is the probability that the signatures will be compared for similiarity.

Example 22.11: Suppose $r = 5$ and $b = 20$; that is, we have signatures of 100 integers, divided into 20 bands of five rows each. The formula for the probability that two signatures of similarity s will be compared becomes $1 - (1 - s^5)^{20}$. Suppose $s = 0.8$; i.e., the underlying sets have Jaccard similarity 80%. $s^5 = 0.328$. That is, the chance that the two signatures agree in a given band is small, only about 1/3. However, we have 20 chances to "win," and $(1-0.328)^{20}$ is tiny, only about 0.00035. Thus, the chance that we *do* find this pair of signatures together in at least one bucket is $1 - 0.00035$, or 0.99965.

On the other hand, suppose $s = 0.4$. Then $1 - \left(1 - (0.4)^5\right)^{20} = (1 - .01)^{20}$, or approximately 20%. If s is much smaller than 0.4, the probability that the signatures will be compared drops below 20% very rapidly. We conclude that the choice $b = 20$ and $r = 5$ is a good one if we are looking for pairs with a very high similarity, say 80% or more, although it would not be a good choice if the similarity threshold were as small as 40%. □

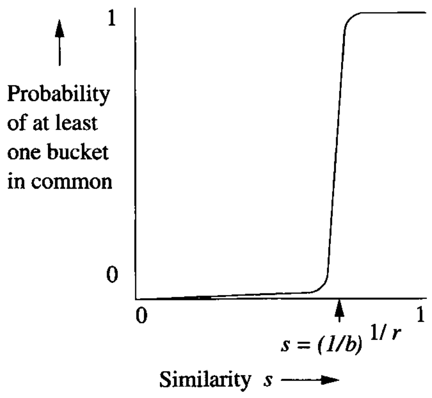

Figure 22.11: The probability that a pair of signatures will appear together in at least one bucket

The function $1 - (1 - s^r)^b$ always looks like Fig. 22.11, but the point of rapid transition from a very small value to a value close to 1 varies, depending on b and r. Roughly, the breakpoint is at similarity $s = (1/b)^{1/r}$.

22.4.3 Combining Minhashing and Locality-Sensitive Hashing

The two ideas, minhashing and LSH, must be combined properly to solve the sort of problems we discussed in Section 22.3.2. Suppose, for example, that we have a large repository of documents, which we have already represented by their sets of shingles of some length. We want to find those documents whose shingle sets have a Jaccard similarity of at least s.

1. Start by computing a minhash signature for each document; how many hash functions to use depends on the desired accuracy, but several hundred should be enough for most purposes.

2. Perform a locality-sensitive hashing to get *candidate* pairs of signatures that hash to the same bucket for at least one band. How many bands and how many rows per band depend on the similarity threshold s, as discussed in Section 22.4.2.

3. For each candidate pair, compute the estimate of their Jaccard similarity by counting the number of components in which their signatures agree.

4. Optionally, for each pair whose signatures are sufficiently similar, compute their true Jaccard similarity by examining the sets themselves.

Of course, this method introduces false positives — candidate pairs that get eliminated in step (2), (3), or (4). However, the second and third steps also allow some false negatives — pairs with a sufficiently high Jaccard similarity that are not candidates or are eliminated from the candidate pool.

a) At step (2), a pair could have very similar signatures, yet there happens to be no band in which the signatures agree in all rows of the band.

b) In step (3), a pair could have Jaccard similarity at least s, but their signatures do not agree in fraction s of the components.

One way to reduce the number of false negatives is to lower the similarity threshold at the initial stages. At step (2), choose a smaller number of rows r or a larger number of bands b than would be indicated by the target similarity s. At step (3) choose a smaller fraction than s of corresponding signature components that allows a pair to move on to step (4). Unfortunately, these changes each increase the number of false positives, so you must consider carefully how small you can afford to make your thresholds.

Another possible way to avoid false negatives is to skip step (3) and go directly to step (4) for each candidate pair. That is, we compute the true

Jaccard similarity of every candidate pair. The disadvantage of doing so is that the minhash signatures were devised to make it easier to compare the underlying sets. For example, if the objects being compared are actually large documents, comparing complete sets of k-shingles is far more time consuming than matching several hundred components of signatures.

In some applications, false negatives are not a problem, so we can tune our LSH to allow a significant fraction of false negatives, in order to reduce false positives and thus to speed up the entire process. For instance, if an on-line retailer is looking for pairs of similar customers, in order to select an item to pitch to each customer, it is not necessary to find every single pair of similar customers. It is sufficient to find a few very similar customers for each customer.

22.4.4 Exercises for Section 22.4

! **Exercise 22.4.1:** This exercise is based on the entity-resolution problem of Example 22.9. For concreteness, suppose that the only pairs records that could possibly be total edit distance 5 or less from each other consist of a true copy of a record and another *corrupted* version of the record. In the corrupted version, each of the three fields is changed independently. 50% of the time, a field has no change. 20% of the time, there is a change resulting in edit distance 1 for that field. There is a 20% chance of edit distance 2 and 10% chance of edit distance 10. Suppose there are one million pairs of this kind in the dataset.

a) How many of the million pairs are within total edit distance 5 of each other?

b) If we hash each field to a large number of buckets, as suggested by Example 22.9, how many of these one million pairs will hash to the same bucket for at least one of the three hashings?

c) How many false negatives will there be; that is, how many of the one million pairs are within total edit distance 5, but will not hash to the same bucket for any of the three hashings?

Exercise 22.4.2: The function $p = 1 - (1 - s^r)^b$ gives the probability p that two minhash signatures that come from sets with Jaccard similarity s will hash to the same bucket at least once, if we use an LSH scheme with b bands of r rows each. For a given similarity threshold s, we want to choose b and r so that $p = 1/2$ at s. we suggested that approximately $s = (1/b)^{1/r}$ is where $p = 1/2$, but that is only an approximation. Suppose signatures have length 24. We can pick any integers b and r whose product is 24. That is, the choices for r are 1, 2, 3, 4, 6, 8, 12, or 24, and b must then be $24/r$.

a) If $s = 1/2$, determine the value of p for each choice of b and r. Which would you choose, if 1/2 were the similarity threshold?

! b) For each choice of b and r, determine the value of s that makes $p = 1/2$.

22.5 Clustering of Large-Scale Data

Clustering is the problem of taking a dataset consisting of "points" and grouping the points into some number of *clusters*. Points within a cluster must be "near" to each other in some sense, while points in different clusters are "far" from each other. We begin with a study of distance measures, since only if we have a notion of distance can we talk about whether points are near or far. An important kind of distance is "Euclidean," a distance based on the location of points within a space. Curiously, not all distances are Euclidean, and an important problem in clustering is dealing with sets of points that do not "live" anywhere in a space, yet have a notion of distance.

We next consider the two major approaches to clustering. One, called "agglomerative," is to start with points each in their own cluster, and repeatedly merge "nearby" clusters. The second, "point assignment," initializes the clusters in some way and then assigns each point to its "best" cluster.

22.5.1 Applications of Clustering

Many discussions of clustering begin with a small example, in which a small number of points are given in a two-dimensional space, such as Fig, 22.12. Algorithms to cluster such data are relatively simple, and we shall mention the techniques only in passing. The problem becomes hard when the dataset is large. It becomes even harder when the number of dimensions of the data is large, or when the data doesn't even belong to a space that has "dimensions." Let us begin by examining some examples of interesting uses of clustering algorithms on large-scale data.

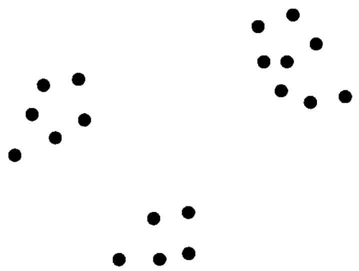
Figure 22.12: Data that can be clustered easily

Collaborative Filtering

In Section 22.3.2 we discussed the problem of finding similar products or similar customers by looking at the set of items each customer bought. The output of analysis using minhashing and locality-sensitive hashing could be a set of pairs of similar products (those bought by many of the same customers. Alternatively, we could look for pairs of similar customers (those buying many of the same products). It may be possible to get a better picture of relationships if we cluster products (points) into groups of similar products. These might represent a natural class of products, e.g., classical-music CD's. Likewise, we might find it useful to cluster customers with similar tastes; e.g., one cluster might be "people who like classical music." For clustering to make sense, we must view the distance between points representing customers or items as "low" if the similarity is high. For example, we shall see in Section 22.5.2 how one minus the Jaccard similarity can serve as a suitable notion of "distance."

Clustering Documents by Topic

We could use the technique described above for products and customers to cluster documents based on their Jaccard similarity. However, another application of document clustering is to group documents into clusters based on their "topics" (e.g., topics such as "sports" or "medicine"), even if documents on the same topic are not very similar character-by-character. A simple approach is to imagine a very high-dimensional space, where there is one dimension for each word that might appear in the document. Place the document at point $(x_1, x_2, \ldots)$, where $x_i = 1$ if the ith word appears in the document and $x_i = 0$ if not. Distance can be taken to be the ordinary Euclidean distance, although as we shall see, this distance measure is not as useful as it might appear at first.

Clustering DNA Sequences

DNA is a sequence of base-pairs, represented by the letters C, G, A, and T. Because these strands sometimes change by substitution of one letter for another or by insertion or deletion of letters, there is a natural edit-distance between DNA sequences. Clustering sequences based on their edit distance allows us to group similar sequences.

Entity Resolution

In Section 21.7.4, we discussed an algorithm for merging records that, in effect, created clusters of records, where each cluster was one connected component of the graph formed by connecting records that met the similarity condition.

SkyCat

In this project, approximately two billion "sky objects" such as stars and galaxies were plotted in a 7-dimensional space, where each dimension represented the radiation of the object in one of seven different bands of the electromagnetic spectrum. By clustering these objects into groups of similar radiation patterns, the project was able to identify approximately 20 different kinds of objects.

Euclidean Spaces

Without going into the theory, for our purposes we may think of a Euclidean space as one with some number of dimensions n. The points in the space are all n-tuples of real numbers $(x_1, x_2, \ldots, x_n)$. The common Euclidean distance is but one of many plausible distance measures in a Euclidean space.

22.5.2 Distance Measures

A *distance measure* on a set of points is a function $d(x, y)$ that satisfies:

1. $d(x, y) \geq 0$ for all points x and y.
2. $d(x, y) = 0$ if and only if $x = y$.
3. $d(x, y) = d(y, x)$ (*symmetry*).
4. $d(x, y) \leq d(x, z) + d(z, y)$ for any points x, y, and z (*triangle inequality*).

That is, the distance from a point to itself is 0, and the distance between any two different points is positive. The distance between points does not depend on which way you travel (symmetry), and it never reduces the distance if you force yourself to go through a particular third point (the triangle inequality).

The most common distance measure is the *Euclidean distance* between points in an n-dimensional Euclidean space. In such a space, points can be represented by n coordinates $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$. The distance $d(x, y)$ is $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$, that is, the square root of the sum of the squares of the differences in each dimension. However, there are many other ways to define distance; we shall examine some below.

Distances Based on Norms

In a Euclidean space, the conventional distance mentioned above is only one possible choice. More generally, we can define the distance

$$d(x, y) = (\sum_{i=1}^{n} |x_i - y_i|^r)^{1/r}$$

for any r. This distance is said to be derived from the L_r*-norm*. The conventional Euclidean distance is the case $r = 2$, and is often called the L_2-norm.

Another common choice is the L_1-norm, that is, the sum of the distances along the coordinates of the space. This distance is often called the *Manhattan distance*, because it is the distance one has to travel along a rectangular grid of streets found in many cities such as Manhattan.

Yet another interesting choice is the L_∞-norm, which is the maximum of the distances in any one coordinate. That is, as r approaches infinity, the value of $\sum_{i=1}^{n} |x_i - y_i|^r)^{1/r}$ approaches the maximum over all i of $|x_i - y_i|$.

Example 22.12: Let $x = (1,2,3)$ and $y = (2,4,1)$. Then the L_2 distance $d(x,y)$ is $\sqrt{|1-2|^2 + |2-4|^2 + |3-1|^2} = \sqrt{(1+4+4)} = 3$. Note that this distance is the conventional Euclidean distance. The Manhattan distance between x and y is $|1-2| + |2-4| + |3-1| = 5$. The L_∞-norm gives distance between x and y of $\max(|1-2|, |2-4|, |3-1|) = 2$. $\square$

Jaccard Distance

The *Jaccard distance* between points that are sets is one minus the Jaccard similarity of those sets. That is, if x and y are sets, then

$$d(x,y) = 1 - (|x \cap y|/|x \cup y|)$$

For example, if the two points represent sets $\{1,2,3\}$ and $\{2,3,4,5\}$, then the Jaccard similarity is $2/5$, so the Jaccard distance is $3/5$.

One might naturally ask whether the Jaccard distance satisfies the axioms of a distance measure. It is easy to see that $d(x,x) = 0$, because

$$1 - (|x \cap x|/|x \cup x|) = 1 - (1/1) = 0$$

It is also easy to see that the Jaccard distance cannot be negative, since the intersection of sets cannot be bigger than their union. Symmetry of the Jaccard distance is likewise straightforward, since both union and intersection are commutative.

The hard part is showing the triangle inequality. Coming to our rescue is the theorem from Section 22.3.4 that says the Jaccard similarity of two sets is the probability that a random permutation will result in the same minhash value for those sets. Thus, the Jaccard *distance* is the probability that the sets will *not* have the same minhash value. Suppose x and y have different minhash values according to a permutation π. Then at least one of the pairs $\{x,z\}$ and $\{z,y\}$ must have different minhash values; possibly both do. Thus, the probability that x and y have different minhash values is no greater than the sum of the probability that x and z have different minhash values plus the probability that z and y have different minhash values. These probabilities are the Jaccard distances mentioned in the triangle inequality. That is, we have shown that the Jaccard distance from x to y is no greater than the sum of the Jaccard distances from x to z and from z to y.

Cosine Distance

Suppose our points are in a Euclidean space. We can think of these points as vectors from the origin of the space. The *cosine distance* between two points is the angle between the vectors.

The Curse of Dimensionality

Our intuition is pretty good when clustering points in one or two dimensions. However, when the points are in a high-dimensional space, our intuition goes awry in several ways. For example, suppose our points are in an n-dimensional hypercube of side 1. If $n = 2$ (i.e., a square), there are many points near the center, and many near the edges. However, for large n, the volume of a hypercube of side just slightly less than 1 is tiny compared with the hypercube of side 1. That means almost every point in the hypercube is very near the surface. There is no "center" and no points to form clusters other than on the surface.

Example 22.13: Suppose documents are characterized by the presence or absence of five words, so points (documents) are vectors of five 0's or 1's. Let $(0,0,1,1,1)$ and $(1,0,0,1,1)$ be the two points. The cosine of the angle between them is computed by taking the dot product of the vectors, and dividing by the product of the lengths of the vectors. In this case, the dot product is $0\times1+0\times0+1\times0+1\times1+1\times1 = 0+0+0+1+1 = 2$. Both vectors have length $\sqrt{3}$. Thus, the cosine of the angle between the vectors is $2/(\sqrt{3}\times\sqrt{3}) = 2/3$. The angle is about 48 degrees. □

Cosine distance satisfies the axioms of a distance measure, as long as points are treated as directions, so two vectors, one of which is a multiple of the other are treated as the same. Angles can only be positive, and if the angle is 0 then the vectors must be in the same direction. Symmetry holds because the angle between x and y is the same as the angle between y and x. The triangle inequality holds because the angle between two vectors is never greater than the sum of the angles between those vectors and a third vector.

Edit Distance

Various forms of edit distance satisfy the axioms of a distance measure. Let us focus on the edit distance that allows only insertions and deletions. If strings x and y are at distance 0 (i.e., no edits are needed) then they surely must be the same. Symmetry follows because insertions and deletions can be reversed. The triangle inequality follows because one way to turn x into y is to first turn x into z and then turn z into y. Thus, the sum of the edit distances from x to z and from z to y is the number of edits needed for one possible way to turn x into y. This number of edits cannot be less than the edit distance from x to y, which is the minimum over all possible ways to get from x to y.

22.5.3 Agglomerative Clustering

We shall now begin our study of algorithms for computing clusters. The first approach is, at the highest level, straightforward. Start with every point in its own cluster. Until some stopping condition is met, repeatedly find the "closest" pair of clusters to merge, and merge them. This methodology is called *agglomerative* or *hierarchical* clustering. The term "hierarchical" comes from the fact that we not only produce clusters, but a cluster itself has a hierarchical substructure that reflects the sequence of mergers that formed the cluster. The devil, as always, is in the details, so we need to answer two questions:

1. How do we measure the "closeness" of clusters?

2. How do we decide when to stop merging?

Defining "Closeness"

There are many ways we could define the closeness of two clusters C and D. Here are two popular ones:

a) Find the minimum distance between any pair of points, one from C and one from D.

b) Average the distance between any pair of points, one from C and one from D.

These measures of closeness work for any distance measure. If the points are in a Euclidean space, then we have additional options. Since real numbers can be averaged, any set of points in a Euclidean space has a *centroid*, the point that is the average, in each coordinate, of the points in the set. For example, the centroid of the set $\{(1,2,3),\ (4,5,6),\ (2,2,2)\}$ is (2.33, 3, 3.67) to two decimal places. For Euclidean spaces, another good choice of closeness measure is:

c) The distance between the centroids of clusters C and D.

Stopping the Merger

One common stopping criterion is to pick a number of clusters k, and keep merging until you are down to k clusters. This approach is good if you have an intuition about how many clusters there should be. For instance, if you have a set of documents that cover three different topics, you could merge until you have three clusters, and hope that these clusters correspond closely to the three topics.

Other stopping criteria involve a notion of *cohesion*, the degree to which the merged cluster consists of points that are all close. Using a cohesion-based stopping policy, we decline to merge two clusters whose combination fails to meet the cohesion condition that we have chosen. At each merger round, we may merge two clusters that are not closest of all pairs of clusters, but are closer than any other pair that meet the cohesion condition. We even could define "closeness" to be the cohesion score, thus combining the merger selection with the stopping criterion. Here are some ways that we could define a cohesion score for a cluster:

i. Let the cohesion of a cluster be the average distance of each point to the centroid. Note that this definition only makes sense in a Euclidean space.

ii. Let the cohesion be the ***diameter***, the largest distance between any pair of points in the cluster.

iii. Let the cohesion be the average distance between pairs of points in the cluster.

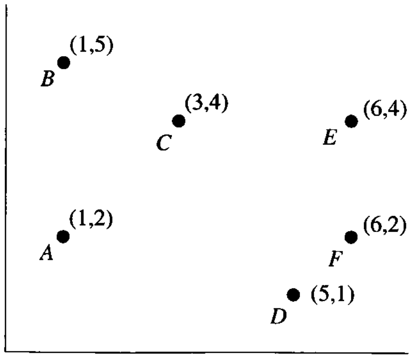

Figure 22.13: Data for Example 22.14

Example 22.14: Consider the six points in Fig. 22.13. Assume the normal Euclidean distance as our distance measure. We shall choose as the distance between clusters the minimum distance between any pair of points, one from each cluster. Initially, each point is in a cluster by itself, so the distances between clusters are just the distances between the points. These distances, to two decimal places, are given in Fig. 22.14

	A	*B*	*C*	*D*	*E*
F	4.00	5.83	3.61	1.41	2.00
E	5.39	5.10	3.00	3.16	
D	4.12	5.66	3.61		
C	2.83	2.24			
B	3.00				

Figure 22.14: Distances between points in Fig. 22.13

The closest two points are D and F, so these get merged into one cluster. We must compute the distance between the cluster DF and each of the other points. By the "closeness" rule we chose, this distance is the minimum of the distances from a node to D or F. The table of distances becomes:

	A	B	C	DF
E	5.39	5.10	3.00	2.00
DF	4.00	5.66	3.61	
C	2.83	2.24		
B	3.00			

The shortest distance above is between E and DF, so we merge these two clusters into a single cluster DEF. The distance to this cluster from each of the other points is the minimum of the distance to any of D, E, and F. This table of distances is:

	A	B	C
DEF	4.00	5.10	3.00
C	2.83	2.24	
B	3.00		

Next, we merge the two closest clusters, which are B and C. The new table of distances is:

	A	BC
DEF	4.00	3.00
BC	2.83	

The last possible merge is A with BC. The result is two clusters, ABC and DEF.

However, we may wish to stop the merging earlier. As an example stopping criterion, let us reject any merger that results in a cluster with an average distance between points over 2.5. Then we can merge D, E, and F; the cohesion (average of the three distances between pairs of these points) is 2.19 (see Fig. 22.14 to check).

At the point where the clusters are A, BC, and DEF, we cannot merge A with BC, even though these are the closest clusters. The reason is that the average distance among the points in ABC is 2.69, which is too high. We might consider merging DEF with BC, which is the second-closest pair of clusters at that time, but the cohesion for the cluster $BCDEF$ is 3.56, also too high. The third option would be to merge A with DEF, but the cohesion of $ADEF$ is 3.35, again too high. □

22.5.4 *k*-Means Algorithms

The second broad approach to clustering is called point-assignment. A popular version, which is typical of the approach is called *k-means*. This approach is really a family of algorithms, just as agglomerative clustering is. The outline of a k-means algorithm is:

1. Start by choosing k initial clusters in some way. These clusters might be single points, or small sets of points.

2. For each unassigned point, place it in the "nearest" cluster.

3. Optionally, after all points are assigned to clusters, fix the centroid of each cluster (assuming the points are in a Euclidean space, since non-Euclidean spaces do not have a notion of "centroid"). Then reassign all points to these k clusters. Occasionally, some of the earliest points to be assigned will thus wind up in another cluster.

One way to initialize a k-means clustering is to pick the first point at random. Then pick a second point as far from the first point as possible. Pick a third point whose minimum distance to either of the other two points is as great as possible. Proceed in this manner, until k points are selected, each with the maximum possible minimum distance to the previously selected points. These points become the initial k clusters.

Example 22.15: Suppose our points are those in Fig. 22.13, $k = 3$, and we choose A as the seed of the first cluster. The point furthest from A is E, so E becomes the seed of the second cluster. For the third point, the minimum distances to A or E are as follows.

$$B\text{: } 3.00,\ C\text{: } 2.83,\ D\text{: } 3.16,\ F\text{: } 2.00$$

The winner is D, with the largest minimum distance of 3.16. Thus, D becomes the third seed. □

Having picked the seeds for the k clusters, we visit each of the remaining points and assign it to a cluster. A simple way is to assign each point to the closest seed. However, if we are in a Euclidean space, we may wish to maintain the centroid for each cluster, and as we assign each point, put it in the cluster with the nearest centroid.

Example 22.16: Let us continue with Example 22.15. We have initialized each of the three clusters A, D, and E, so their centroids are the points themselves. Suppose we assign B to a cluster. The nearest centroid is A, at distance 3.00. Thus, the first cluster becomes AB, and its centroid is $(1, 3.5)$. Suppose we assign C next. Clearly C is closer to the centroid of AB than it is to either D or E, so C is assigned to AB, which becomes ABC with centroid $(1.67, 3.67)$. Last, we assign F; it is closer to D than to E or to the centroid of ABC. Thus, the three clusters are ABC, DF, and E, with centroids $(1.67, 3.67)$, $(5.5, 1.5)$, and $(6, 4)$, respectively. We could reassign all points to the nearest of these three centroids, but the resulting clusters would not change. □

22.5.5 k-Means for Large-Scale Data

We shall now examine an extension of k-means that is designed to deal with sets of points that are so large they cannot fit in main memory. The goal is not to assign every point to a cluster, but to determine where the centroids of the clusters are. If we really wanted to know the cluster of every point, we would have to make another pass through the data, assigning each point to its nearest centroid and writing out the cluster number with the point.

This algorithm, called the BFR Algorithm,[4] assumes an n-dimensional Euclidean space. It may therefore represent clusters, as they are forming, by their centroids. The BFR Algorithm also assumes that the cohesion of a cluster can be measured by the variance of the points within a cluster; the variance of a cluster is the average square of the distance of a point in the cluster from the centroid of the cluster. However, for convenience, it does not record the centroid and variance, but rather the following $2n + 1$ *summary statistics*:

1. N, the number of points in the cluster.

2. For each dimension i, the sum of the ith coordinates of the points in the cluster, denoted SUM_i.

3. For each dimension i, the sum of the squares of the ith coordinates of the points in the cluster, denoted SUMSQ_i.

The reason to use these parameters is that they are easy to compute when we merge clusters. Just add the corresponding values from the two clusters. However, we can compute the centroid and variance from these values. The rules are:

- The ith coordinate of the centroid is SUM_i/N.

- The variance in the ith dimension is $\text{SUMSQ}_i/N - (\text{SUM}_i/N)^2$.

Also remember that σ_i, the standard deviation in the ith dimension is the square root of the variance in that dimension.

The BFR Algorithm reads the data one main-memory-full at a time, leaving space in memory for the summary statistics for the clusters and some other data that we shall discuss shortly. It can initialize by picking k points from the first memory-load, using the approach of Example 22.15. It could also do any sort of clustering on the first memory load to obtain k clusters from that data. During the running of the algorithm, points are divided into three classes:

1. The *discard set*: points that have been assigned to a cluster. These points do not appear in main memory. They are represented only by the summary statistics for their cluster.

[4]For the authors, P. S. Bradley, U. M. Fayyad, and C. Reina.

2. The *compressed set*: There can be many groups of points that are sufficiently close to each other that we believe they belong in the same cluster, but they are not close to any cluster's current centroid, so we do not know to which cluster they belong. Each such group is represented by its summary statistics, just like the clusters are, and the points themselves do not appear in main memory.

3. The *retained set*: These points are not close to any other points; they are "outliers." They will eventually be assigned to the nearest cluster, but for the moment we retain each such point in main memory.

These sets change as we process successive memory-loads of the data. Figure 22.15 suggests the state of the data after some number of memory-loads have been processed by the BFR Algorithm.

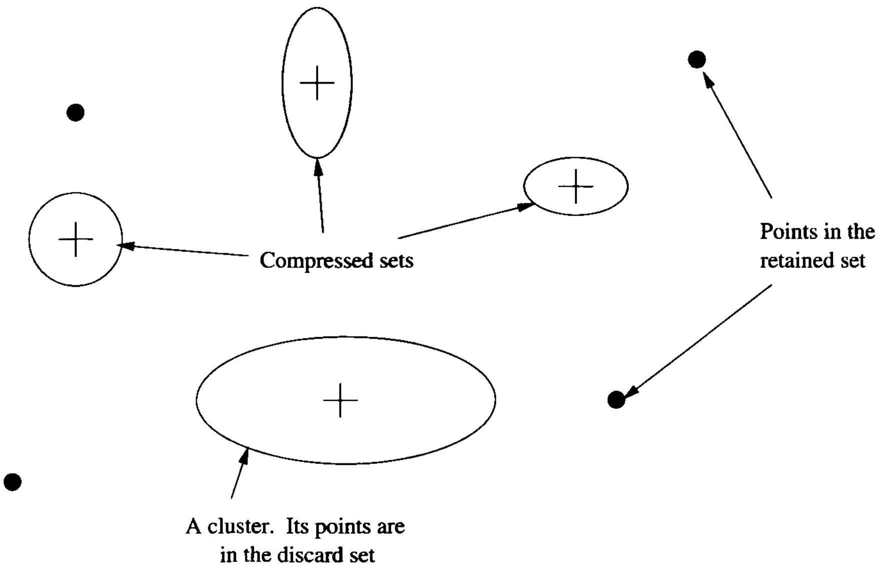
Compressed sets
Points in the retained set
A cluster. Its points are in the discard set

Figure 22.15: A cluster, several compressed sets and several points of the retained set

22.5.6 Processing a Memory Load of Points

We shall now describe how one memory load of points is processed. We assume that main memory current contains the summary statistics for the k clusters and also for zero or more groups of points that are in the compressed set. Main memory also holds the current set of points in the retained set. We do the following steps:

1. For all points $(x_1, x_2, \ldots, x_n)$ that are "sufficiently close" (a term we shall define shortly) to the centroid of a cluster, add the point to this cluster. The point itself goes into the discard set. We add 1 to N in the summary statistics for that cluster. We also add x_i to SUM_i and add ${x_i}^2$ to SUMSQ_i for that cluster.

2. If this memory load is the last, then merge each group from the compressed set and each point of the retained set into its nearest cluster. Remember that it is easy to merge clusters and groups using their summary statistics. Just add the counts N, and add corresponding components of the SUM and SUMSQ vectors. The algorithm ends at this point.

3. Otherwise (the memory load is not the last), use any main-memory clustering algorithm to cluster the remaining points from this memory load, along with all points in the current retained set. Set a threshold on the cohesiveness of a cluster, so we do not merge points unless they are reasonably close.

4. Those points that remain in clusters of size 1 (i.e., they are not near any other point) become the new retained set. Clusters of more than one point become groups in the compressed set and are replaced by their summary statistics.

5. Consider merging groups in the compressed set. Use some cohesiveness threshold to decide whether groups are close enough; we shall discuss how to make this decision shortly. If they can be merged, then it is easy to combine their summary statistics, as in (2) above.

Deciding Whether a Point is Close Enough to a Cluster

Intuitively, each cluster has a size in each dimension that indicates how far out in that dimension typical points extend. Since we have only the summary statistics to work with, the appropriate statistic is the standard deviation in that dimension. Recall from Section 22.5.5 that we can compute the standard deviations from the summary statistics, and in particular, the standard deviation is the square root of the variance. However, clusters may be "cigar-shaped, so the standard deviations could vary widely. We want to include a point if its distance from the cluster centroid is not too many standard deviations in any dimension.

Thus, the first thing to do with a point $p = (x_1, x_2, \ldots, x_n)$ that we are considering for inclusion in a cluster is to normalize p relative to the centroid and the standard deviations of the cluster. That is, we transform the point into $p' = (y_1, y_2, \ldots, y_n)$, where $y_i = (x_i - c_i)/\sigma_i$; here c_i is the coordinate of the centroid in the ith dimension and σ_i is the standard deviation of the cluster in that dimension. The normalized distance of p from the centroid is the absolute distance of p' from the origin, that is, $\sqrt{\sum_{i=1}^{n} {y_i}^2}$. This distance is sometimes called the *Mahalanobis distance*, although it is actually a simplifed version of the concept.

Example 22.17: Suppose p is the point $(5, 10, 15)$, and we are considering whether to include p in a cluster with centroid $(10, 20, 5)$. Also, let the standard deviation of the cluster in the three dimensions be 1, 2, and 10, respectively. Then the Mahalanobis distance of p is

$$\sqrt{((5-10)/1)^2 + ((10-20)/2)^2 + ((15-5)/10)^2} = \sqrt{25+25+1} = 7.14$$

□

Having computed the Mahalanobis distance of point p, we can apply a threshold to decide whether or not to include p in the cluster. For instance, suppose we use 3 as the threshold; that is, we shall include the point if and only if its Mahalanobis distance from the centroid is not greater than 3. If values are normally distributed, then very few of these values will be more than 3 standard deviations from the mean (approximately one in a million will be that far from the mean). Thus, we would only reject one in a million points that belong in the cluster. There is a good chance that, at the end, the rejected points would wind up in the cluster anyway, since there may be no closer cluster.

Deciding Whether to Merge Groups of the Compressed Set

We discussed methods of computing the cohesion of a prospective cluster in Section 22.5.3. However, for the BFR algorithm, these ideas must be modified so we can make a decision using only the summary statistics for the two groups. Here are some options:

1. Choose an upper bound on the sum of the variances of the combined group in each dimension. Recall that we compute the summary statistics for the combined group by adding corresponding components, and compute the variance in each dimension using the formula in Section 22.5.5. This approach has the effect of limiting the region of space in which the points of a group exist. Groups in which the distances between typical pairs of points is too large will exceed the upper bound on variance, no matter how many points are in the group and how dense the points are within the region of space the group occupies.

2. Put an upper limit on the diameter in any dimension. Since we do not know the locations of the points exactly, we cannot compute the exact diameter. However, we could estimate the diameter in the ith dimension as the distance between the centroids of the two groups in dimension i plus the standard deviation of each group in dimension i. This approach also limits the size of the region of space occupied by a group.

3. Use one of the first two approaches, but divide the figure of merit (sum of variances or maximum diameter) by a quantity such as N or $\sqrt{N}$ that grows with the number of points in the group. That way, groups can occupy more space, as long as they remain dense within that space.

22.5.7 Exercises for Section 22.5

Exercise 22.5.1: For each pair of the points in Fig. 22.13:

a) Compute the Manhattan distance (L_1-norm).

b) Compute the L_∞-norm.

! **Exercise 22.5.2:** Show that for any $r \geq 1$, the distance based on the L_r norm satisfies the axioms of a distance measure. What happens if $r < 1$?

Exercise 22.5.3: In Example 22.14 we performed a hierarchical clustering of the points in Fig. 22.13, using minimum distance between points as the measure of closeness of clusters. Repeat the example using each of the following ways of measuring the distance between clusters.

a) The distance between the centroids of the clusters.

b) The maximum distance between points, one from each cluster.

c) The average distance between points, one from each cluster.

Exercise 22.5.4: We could also modify Example 22.14 by using a different distance measure. Suppose we use the L_∞-norm as the distance measure. Note that this distance is the maximum of the distances along any axis, but when comparing distances you can break ties according to the next largest dimension. Show the sequence of mergers of the points in Fig. 22.13 that result from the use of this distance measure.

Exercise 22.5.5: Suppose we want to select three nodes in Fig. 22.13 to start three clusters, and we want them to be as far from each other as possible, as in Example 22.15. What points are selected if we start with (a) point B (b) point C?

Exercise 22.5.6: The BFR Algorithm represents clusters by summary statistics, as described in Section 22.5.5. Suppose the current members of a cluster are $\{(1,2),\ (3,4),\ (2,1),\ (0,5)\}$. What are the summary statistics for this cluster?

Exercise 22.5.7: For the cluster described in Example 22.17, compute the Mahalanobis distance of the points: (a) $(8,21,0)$ (b) $(10,25,25)$.

22.6 Summary of Chapter 22

✦ *Data Mining*: This term refers to the discovery of simple summaries of data.

✦ *The Market-Basket Model of Data*: A common way to represent a many-many relation is as a collection of baskets, each of which contains a set of items. Often, this data is presented not as a relation but as a file of baskets. Algorithms typically make passes through this file, and the cost of an algorithm is the number of passes it makes.

✦ *Frequent Itemsets*: An important summary of some market-basket data is the collection of frequent itemsets: sets of items that occur in at least some fixed number of baskets. The minimum number of baskets that make an itemset frequent is called the support threshold.

✦ *Association Rules*: These are statements of the form that say if a certain set of items appears in a basket, then there is at least some minimum probability that another particular item is also in that basket. The probability is called the confidence of the rule.

✦ *The A-Priori Algorithm*: This algorithm finds frequent itemsets by exploiting the fact that if a set of items occurs at least s times, then so does each of its subsets. For each size of itemset, we start with the candidate itemsets, which are all those whose every immediate subset (the set minus one element) is known to be frequent. We then count the occurrences of the candidates in a single pass, to determine which are truly frequent.

✦ *The PCY Algorithm*: This algorithm makes better use of main memory than A-priori does, while counting the singleton items. PCY additionally hashes all pairs to buckets and counts the total number of baskets that contain a pair hashing to each bucket. To be a candiate on the second pass, a pair has to consist of items that not only are frequent as singletons, but also hash to a bucket whose count exceeded the support threshold.

✦ *The Multistage Algorithm*: This algorithm improves on PCY by using several passes in which pairs are hashed to buckets using different hash functions. On the final pass, a pair can only be a candidate if it consists of frequent items and also hashed each time to a bucket that had a count at least equal to the support threshold.

✦ *Similar Sets and Jaccard Similarity*: Another important use of market-basket data is to find similar baskets, that is, pairs of baskets with many elements in common. A useful measure is Jaccard similarity, which is the ratio of the sizes of the intersection and union of the two sets.

- *Shingling Documents*: We can find similar documents if we convert each document into its set of k-shingles — all substrings of k consecutive characters in the document. In this manner, the problem of finding similar documents can be solved by any technique for finding similar sets.

- *Minhash Signatures*: We can represent sets by short signatures that enable us to estimate the Jaccard similarity of any two represented sets. The technique known as minhashing chooses a sequence of random permutations, implemented by hash functions. Each permutation maps a set to the first, in the permuted order, of the members of that set, and the signature of the set is the list of elements that results by applying each permutation in this way.

- *Minhash Signatures and Jaccard Similarity*: The reason minhash signatures serve to represent sets is that the Jaccard similarity of sets is also the probability that two sets will agree on their minhash values. Thus, we can estimate the Jaccard similarity of sets by counting the number of components on which their minhash signatures agree.

- *Locality-Sensitive Hashing*: To avoid having to compare all pairs of signatures, locality-sensitive hashing divides the signatures into bands, and compares two signatures only if they agree exactly in at least one band. By tuning the number of bands and the number of components per band, we can focus attention on only the pairs that are likely to meet a given similarity threshold.

- *Clustering*: The problem is to find groups (clusters) of similar items (points) in a space with a distance measure. One approach, called agglomerative, is to build bigger and bigger clusters by merging nearby clusters. A second approach is to estimate the clusters initially and assign points to the nearest cluster.

- *Distance Measures*: A distance on a set of points is a function that assigns a nonnegative number to any pair of points. The function is 0 only if the points are the same, and the function is commutative. It must also satisfy the triangle inequality.

- *Commonly Used Distance Measures*: If points occupy a Euclidean space, essentially a space with some number of dimensions and a coordinate system, we can use the ordinary Euclidean distance, or modifications such as the Manhattan distance (sum of the distances along the coordinates). In non-Euclidean spaces, we can use distance measures such as the Jaccard distance between sets (one minus Jaccard similiarity) or the edit distance between strings.

- *BFR Algorithm*: This algorithm is a variant of k-means, where points are assigned to k clusters. Since the BFR Algorithm is intended for data sets that are two large to fit in main memory, it compresses most points into sets that are represented only by their count and, for each dimension, the sum of their coordinates and the sum of the squares of their coordinates. each

22.7 References for Chapter 22

Two useful books on data mining are [7] and [10].

The A-Priori Algorithm comes from [1] and [2]. The PCY Algorithm is from [9] and the multistage algorithm is from [6].

The use of shingling and minhashing to discover similar documents is from [4] and the theory of minhashing is in [5]. Locality-sensitive hashing is from [8].

Clustering of non-main-memory data sets was first considered in [11]. The BFR Algorithm is from [3].

1. R. Agrawal, T. Imielinski, and A. Swami, "Mining associations between sets of items in massive databases," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 207–216, 1993.

2. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," *Intl. Conf. on Very Large Databases*, pp. 487–499, 1994.

3. P. S. Bradley, U. M. Fayyad, and C. Reina, "Scaling clustering algorithms to large databases," *Proc. Knowledge Discovery and Data Mining*, pp. 9–15, 1998.

4. A. Z. Broder, "On the resemblance and containment of documents," *Proc. Compression and Complexity of Sequences*, pp. 21–29, Positano Italy, 1997.

5. A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *J. Computer and System Sciences* **60**:3 (2000), pp. 630–659.

6. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," *Intl. Conf. on Very Large Databases*, pp. 299-310, 1998.

7. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, MIT Press, 1996.

8. P. Indyk and R. Motwani, "Approximate nearest neighbors: toward removing the curse of dimensionality," *ACM Symp. on Theory of Computing*, pp. 604–613, 1998.

9. J. S. Park, M.-S. Chen, and P. S. Yu, "An effective hash-based algorithm for mining association rules," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 175–186, 1995.

10. P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, Addison-Wesley, Boston MA, 2006.

11. T. Zhang, R. Ramakrishnan, and M. Livny, “BIRCH: an efficient data clustering method for very large databases,” *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 103–114, 1996.

Chapter 23

Database Systems and the Internet

The age of the World-Wide Web has had a profound effect on database technology. Conventional relational databases sit behind, and power, many of the most important Web applications, as we discussed in Section 9.1. But Web applications have also forced databases to assume new forms. Often, massive databases are not found inside a relational DBMS, but in complex, ad-hoc file structures. One of the most important examples of this phenomenon is the way search engines manage their data. Thus, in this chapter we shall examine algorithms for crawling the Web and for answering search-engine queries.

Other sources of data are dynamic in nature. Rather than existing in a database, the data is a stream of information that must either be processed and stored as it arrives, or thrown away. One example is the click streams (sequence of URL requests) received at major Web sites. Non-Web-related streams of data also exist, such as the "call-detail records" generated by all the telephone calls traveling through a network, and data generated by satellites and networks of sensors. Thus, the second part of this chapter addresses the stream data model and the technology needed to manage massive data in the form of streams.

23.1 The Architecture of a Search Engine

The search engine has become one of the most important tools of the 21st century. The repositories managed by the major search engines are among the largest databases on the planet, and surely no other database is accessed so frequently and by so many users. In this section, we shall examine the key components of a search engine, which are suggested schematically in Fig. 23.1.

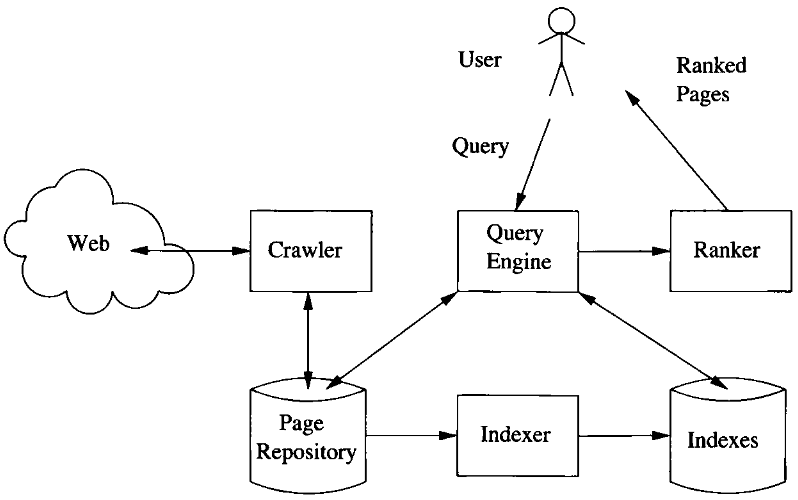

Figure 23.1: The components of a search engine

23.1.1 Components of a Search Engine

There are two main functions that a search engine must perform.

1. The Web must be *crawled*. That is, copies of many of the pages on the Web must be brought to the search engine and processed.

2. Queries must be answered, based on the material gathered from the Web. Usually, the query is in the form of a word or words that the desired Web pages should contain, and the answer to a query is a ranked list of the pages that contain all those words, or at least some of them.

Thus, in Fig. 23.1, we see the *crawler* interacting with the Web and with the *page repository*, a database of pages that the crawler has found. We shall discuss crawling in more detail in Section 23.1.2.

The pages in the page repository are indexed. Typically, these indexes are inverted indexes, of the type discussed in Section 14.1.8. That is, for each word, there is a list of the pages that contain that word. Additional information in the index for the word may include its location(s) within the page or its role, e.g., whether the word is in the header.

We also see in Fig. 23.1 a user issuing a query that consists of one or more words. A *query engine* takes those words and interacts with the indexes, to determine which pages satisfy the query. These pages are then ordered by a *ranker*, and presented to the user, typically 10 at a time, in ranked order. We shall have more to say about the query process in Section 23.1.3.

23.1.2 Web Crawlers

A crawler can be a single machine that is started with a set S, containing the URL's of one or more Web pages to crawl. There is a repository R of pages, with the URL's that have already been crawled; initially R is empty.

Algorithm 23.1: A Simple Web Crawler.

INPUT: An initial set of URL's S.

OUTPUT: A repository R of Web pages.

METHOD: Repeatedly, the crawler does the following steps.

```
1. If $S$ is empty, end.
2. Select a page $p$ from the set $S$ to "crawl" and delete $p$ from $S$.
3. Obtain a copy of $p$, using its URL. If $p$ is already in repository $R$, return
   to step (1) to select another page.
4. If $p$ is not already in $R$:
   (a) Add $p$ to $R$.
   (b) Examine $p$ for links to other pages. Insert into $S$ the URL of each
       page $q$ that $p$ links to, but that is not already in $R$ or $S$.
5. Go to step (1).
```

□

Algorithm 23.1 raises several questions.

a) How do we terminate the search if we do not want to search the entire Web?

b) How do we check efficiently whether a page is already in repository R?

c) How do we select a page p from S to search next?

d) How do we speed up the search, e.g., by exploiting parallelism?

Terminating Search

Even if we wanted to search the "entire Web," we must limit the search somehow. The reason is that some pages are generated dynamically, so when the crawler asks a site for a URL, the site itself constructs the page. Worse, that page may have URL's that also refer to dynamically constructed pages, and this process could go on forever.

As a consequence, it is generally necessary to cut off the search at some point. For example, we could put a limit on the number of pages to crawl, and stop when that limit is reached. The limit could be either on each site or on the total number of pages. Alternatively, we could limit the *depth* of the crawl. That is, say that the pages initially in set S have depth 1. If the page p selected for crawling at step (2) of Algorithm 23.1 has depth i, then any page q that we add to S at step (4b) is given depth $i+1$. However, if p has depth equal to the limit, then we do not examine links out of p at all. Rather we simply add p to R, if it is not already there.

Managing the Repository

There are two points where we must avoid duplication of effort. First, when we add a new URL for a page q to the set S, we should check that it is not already there or among the URL's of pages in R. There may be billions of URL's in R and/or S, so this job requires an efficient index structure, such as those in Chapter 14.

Second, when we decide to add a new page p to R at step (4a) of Algorithm 23.1, we should be sure the page is not already there. How could it be, since we make sure to search each URL only once? Unfortunately, the same page can have several different URL's, so our crawler may indeed encounter the same page via different routes. Moreover, the Web contains mirror sites, where large collection of pages are duplicated, or nearly duplicated (e.g., each may have different internal links within the site, and each may refer to the other mirror sites). Comparing a page p with all the pages in R can be much too time-consuming. However, we can make this comparison efficient as follows:

1. If we only want to detect exact duplicates, hash each Web page to a signature of, say, 64 bits. The signatures themselves are stored in a hash table T; i.e., they are further hashed into a smaller number of buckets, say one million buckets. If we are considering inserting p into R, compute the 64-bit signature $h(p)$, and see whether $h(p)$ is already in the hash table T. If so, do not store p; otherwise, store p in R. Note that we could get some false positives; it could be that $h(p)$ is in T, yet some page other than p produced the same signature. However, by making signatures sufficiently long, we can reduce the probability of a false positive essentially to zero.

2. If we want to detect near duplicates of p, then we can store minhash signatures (see Section 22.3) in place of the simple hash-signatures mentioned in (1). Further, we need to use locality-sensitive hashing (see Section 22.4) in place of the simple hash table T of option (1).

Selecting the Next Page

We could use a completely random choice of next page. A better strategy is to manage S as a queue, and thus do a breadth-first search of the Web from the starting point or points with which we initialized S. Since we presumably start the search from places in the Web that have "important" pages, we thus are assured of visiting preferentially those portions of the Web that the authors of these "important" pages thought were also important.

An alternative is to try to estimate the importance of pages in the set S, and to favor those pages we estimate to be most important. We shall take up in Section 23.2 the idea of PageRank as a measure of the importance that the Web attributes to certain pages. It is impossible to compute PageRank exactly while the crawl is in progress. However, a simple approximation is to count the number of known in-links for each page in set S. That is, each time we examine a link to a page q at step (4b) of Algorithm 23.1, we add one to the count of in-links for q. Then, when selecting the next page p to crawl at step (2), we always pick one of the pages with the highest number of in-links.

Speeding Up the Crawl

We do not need to limit ourselves to one crawling machine, and we do not need to limit ourselves to one process per machine. Each process that acts on the set of available URL's (what we called S in Algorithm 23.1) must lock the set, so we do not find two processes obtaining the same URL to crawl, or two processes writing the same URL into the set at the same time. If there are so many processes that the lock on S becomes a bottleneck, there are several options.

We can assign processes to entire hosts or sites to be crawled, rather than to individual URL's. If so, a process does not have to access the set of URL's S so often, since it knows no other process will be accessing the same site while it does.

There is a disadvantage to this approach. A crawler gathering pages at a site can issue page requests at a very rapid rate. This behavior is essentially a denial-of-service attack, where the site can do no useful work while it strives to answer all the crawler's requests. Thus, a responsible crawler does not issue frequent requests to a single site; it might limit itself to one every several seconds. If a crawling process is visiting a single site, then it must slow down its rate of requests to the point that it is often idle. That in itself is not a problem, since we can run many crawling processes at a single machine. However, operating-system software has limits on how many processes can be alive at any time.

An alternative way to avoid bottlenecks is to partition the set S, say by hashing URL's into several buckets. Each process is assigned to select new URL's to crawl from a particular one of the buckets. When a process follows a link to find a new URL, it hashes that URL to determine which bucket it belongs in. That bucket is the only one that needs to be examined to see if the new URL is already there, and if it is not, that is the bucket into which the new URL is placed.

The same bottleneck issues that arise for the set S of active URL's also come up in managing the page repository R and its set of URL's. The same two techniques — assigning processes to sites or partitioning the set of URL's by hashing — serve to avoid bottlenecks in the accessing of R as well.

23.1.3 Query Processing in Search Engines

Search engine queries are not like SQL queries. Rather they are typically a set of words, for which the search engine must find and rank all pages containing all, or perhaps a subset of, those words. In some cases, the query can be a boolean combination of words, e.g., all pages that contain the word "data" or the word "base." Possibly, the query may require that two words appear consecutively, or appear near each other, say within 5 words.

Answering queries such as these requires the use of inverted indexes. Recall from our discussion of Fig. 23.1 that once the crawl is complete, the indexer constructs an inverted index for all the words on the Web. Note that there will be hundreds of millions of words, since any sequence of letters and digits surrounded by punctuation or whitespace is an indexable word. Thus, "words" on the Web include not only the words in any of the world's natural languages, but all misspellings of these words, error codes for all sorts of systems, acronyms, names, and jargon of many kinds.

The first step of query processing is to use the inverted index to determine those pages that contain the words in the query. To offer the user acceptable response time, this step must involve few, if any, disk accesses. Search engines today give responses in fractions of a second, an amount of time so small that it amounts to only a few disk-access times.

On the other hand, the vectors that represent occurrences of a single word have components for each of the pages indexed by the search engine, perhaps tens of billions of pages. Very rare words might be represented by listing their occurrences, but for common, or even reasonably rare words, it is more efficient to represent by a bit vector the pages in which they occur. The AND of bit vectors gives the pages containing both words, and the OR of bit vectors gives the pages containing one or both. To speed up the selection of pages, it is essential to keep as many vectors as possible in main memory, since we cannot afford disk accesses. Teams of machines may partition the job, say each managing the portion of bit vectors corresponding to a subset of the Web pages.

23.1.4 Ranking Pages

Once the set of pages that match the query is determined, these pages are ranked, and only the highest-ranked pages are shown to the user. The exact way that pages are ranked is a secret formula, as closely guarded by search engines as the formula for Coca Cola. One important component is the "PageRank," a measure of how important the Web itself believes the page to be. This measure is based on links to the page in question, but is significantly more complex than that. We discuss PageRank in detail in Section 23.2.

Some of the other measures of how likely a page is to be a relevant response to the query are fairly easy to reason out. The following is a list of typical components of a relevance measure for pages.

1. The presence of all the query words. While search engines will return

pages with only a proper subset of the query words, these pages are generally ranked lower than pages having all the words.

2. The presence of query words in important positions in the page. For example, we would expect that a query word appearing in a title of the page would indicate more strongly that the page was relevant to that word than its mere occurrence in the middle of a paragraph. Likewise, appearance of the word in a header cell of a table would be a more favorable indication than its appearance in a data cell of the same table.

3. Presence of several query words near each other would be a more favorable indication than if the words appeared in the page, but widely separated. For example, if the query consists of the words "sally" and "jones," we are probably looking for pages that mention a certain person. Many pages have lists of names in them. If "sally" and "jones" appear adjacent, or perhaps separated by a middle initial, then there is a better chance the page is about the person we want than if "sally" appeared, but nowhere near "jones." In that case, there are probably two different people, one with first name Sally, and the other with last name Jones.

4. Presence of the query words in or near the anchor text in links leading to the page in question. This insight was one of the two key ideas that made the Google search engine the standard for the field (the other is PageRank, to be discussed next). A page may lie about itself, by using words designed to make it appear to be a good answer to a query, but it is hard to make other people confirm your lie in their own pages.

23.2 PageRank for Identifying Important Pages

One of the key technological advances in search is the PageRank[1] algorithm for identifying the "importance" of Web pages. In this section, we shall explain how the algorithm works, and show how to compute PageRank for very large collections of Web pages.

23.2.1 The Intuition Behind PageRank

The insight that makes Google and other search engines able to return the "important" pages on a topic is that the Web itself points out the important pages. When you create a page, you tend to link that page to others that you think are important or valuable, rather than pages you think are useless. Of course others may differ in their opinions, but on balance, the more ways one can get to a page by following links, the more likely the page is to be important.

We can formalize this intuition by imagining a random walker on the Web. At each step, the random walker is at one particular page p and randomly

[1] After Larry Page, who first proposed the algorithm.

picks one of the pages that p links to. At the next step, the walker is at the chosen successor of p. The structure of the Web links determines the long-run probability that the walker is at each individual page. This probability is termed the *PageRank* of the page.

Intuitively, pages that a lot of other pages point to are more likely to be the location of the walker than pages with few in-links. But all in-links are not equal. It is better for a page to have a few links from pages that themselves are likely places for the walker to be than to have many links from pages that the walker visits infrequently or not at all. Thus, it is not sufficient to count the in-links to compute the PageRank. Rather, we must solve a recursive equation that formalizes the idea:

- A Web page is important if many important pages link to it.

23.2.2 Recursive Formulation of PageRank — First Try

To describe how the random walker moves, we can use the *transition matrix of the Web*. Number the pages $1, 2, \ldots, n$. The matrix $\mathbf{M}$, the transition matrix of the Web has element m_{ij} in row i and column j, where:

1. $m_{ij} = 1/r$ if page j has a link to page i, and there are a total of $r \geq 1$ pages that j links to.
2. $m_{ij} = 0$ otherwise.

If every page has at least one link out, then the transition matrix will be (*left*) *stochastic* — elements are nonnegative, and its columns each sum to exactly 1. If there are pages with no links out, then the column for that page will be all 0's, and the transition matrix is said to be *substochastic* (all columns sum to at most 1).

Example 23.2: As we all know, the Web has been growing exponentially, so if you extrapolate back to 1839, you find that the Web consisted of only three pages. Figure 23.2 shows what the Web looked like in 1839.

We have numbered the pages 1, 2, and 3, so the transition matrix for this graph is:

$$\mathbf{M} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

For example, node 3, the page for Microsoft, links only to node 2, the page for Amazon. Thus, in column 3, only row 2 is nonzero, and its value is 1 divided by the number of out-links of node 3, which is 1. As another example, node 1, Yahoo!, links to itself and to Amazon (node 2). Thus, in column 1, row 3 is 0, and rows 1 and 2 are each 1 divided by the number of out-links from node 1, i.e., 1/2. □

PageRank Combats Spam

Before Google and PageRank, search engines had a great deal of trouble recognizing important pages on the Web. It was common for unscrupulous Web sites ("spammers") to put bogus content on their pages, often in ways that could not be seen by users, but that search engines would see in the text of the page (e.g., by making the writing have the same color as the background). If Google had simply counted in-links to measure the importance of pages, then the spammers could have created massive numbers of other bogus pages that linked to the page they wanted the search engines to think was important. However, simply creating a page doesn't give it much PageRank, since truly important pages are unlikely to link to it. Thus, PageRank defeated the spammers of the day.

Interestingly, the war between spammers and search engines continues. The spammers eventually learned how to increase the PageRank of bogus pages, which led to techniques for combating new forms of spam, often called "link spam." We shall address link spam in Section 23.3.3.

Suppose y, a, and m represent the fractions of the time the random walker spends at the three pages of Fig. 23.2. Then multiplying the column-vector of these three values by **M** will not change their values. The reason is that, after a large number of moves, the walker's distribution of possible locations is the same at each step, regardless where the walker started. That is, the unknowns y, a, and m must satisfy:

$$\begin{bmatrix} y \\ a \\ m \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} y \\ a \\ m \end{bmatrix}$$

Although there are three equations in three unknowns, you cannot solve these equations for more than the ratios of y, a, and m. That is, if $[y, a, m]$ is a solution to the equations, then $[cy, ca, cm]$ is also a solution, for any constant c. However, since y, a, and m form a probability distribution, we also know $y + a + m = 1$.

While we could solve the resulting equations without too much trouble, solving large numbers of simultaneous linear equations takes time $O(n^3)$, where n is the number of variables or equations. If n is in the billions, as it would be for the Web of today, it is utterly infeasible to solve for the distribution of the walker's location by Gaussian elimination or another direct solution method. However, we can get a good approximation by the method of *relaxation*, where we start with some estimate of the solution and repeatedly multiply the estimate by the matrix **M**. As long as the columns of **M** each add up to 1, then the sum of the values of the variables will not change, and eventually they converge to

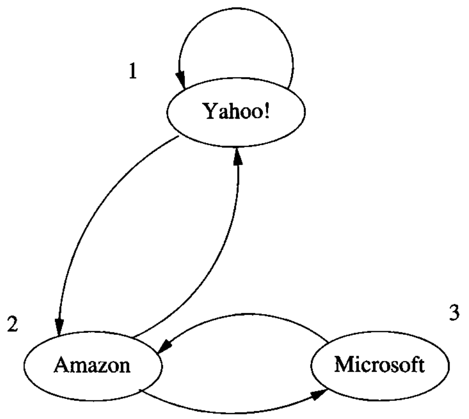

Figure 23.2: The Web in 1839

the distribution of the walker's location. In practice, 50 to 100 iterations of this process suffice to get very close to the exact solution.

Example 23.3: Suppose we start with $[y, a, m] = [1/3, 1/3, 1/3]$. Multiply this vector by $\mathbf{M}$ to get

$$\begin{bmatrix} 2/6 \\ 3/6 \\ 1/6 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

At the next iteration, we multiply the new estimate [2/6,3/6,1/6] by $\mathbf{M}$, as:

$$\begin{bmatrix} 5/12 \\ 4/12 \\ 3/12 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 2/6 \\ 3/6 \\ 1/6 \end{bmatrix}$$

If we repeat this process, we get the following sequence of vectors:

$$\begin{bmatrix} 9/24 \\ 11/24 \\ 4/24 \end{bmatrix}, \begin{bmatrix} 20/48 \\ 17/48 \\ 11/48 \end{bmatrix}, \ldots, \begin{bmatrix} 2/5 \\ 2/5 \\ 1/5 \end{bmatrix}$$

That is, asymptotically, the walker is equally likely to be at Yahoo! or Amazon, and only half as likely to be at Microsoft as either one of the other pages. □

23.2.3 Spider Traps and Dead Ends

The graph of Fig. 23.2 is atypical of the Web, not only because of its size, but for two structural reasons:

1. Some Web pages (called *dead ends*) have no out-links. If the random walker arrives at such a page, there is no place to go next, and the walk ends.

2. There are sets of Web pages (called *spider traps*) with the property that if you enter that set of pages, you can never leave, because there are no links from any page in the set to any page outside the set.

Any dead end is, by itself, a spider trap. However, one also finds on the Web spider traps all of whose pages have out-links. For example, any page that links only to itself is a spider trap.

If a spider trap can be reached from outside, then the random walker may wind up there eventually, and never leave. Put another way, applying relaxation to the matrix of the Web with spider traps can result in a limiting distribution where all probabilities outside a spider trap are 0.

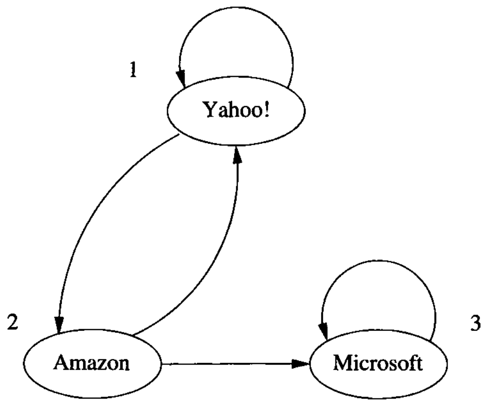

Figure 23.3: The Web, if Microsoft becomes a spider trap

Example 23.4: Suppose Microsoft decides to link only to itself, rather than Amazon, resulting in the Web of Fig. 23.3. Then the set of pages consisting of Microsoft alone is a spider trap, and that trap can be reached from either of the other pages. The matrix $\mathbf{M}$ for this Web graph is

$$\mathbf{M} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix}$$

Here is the sequence of approximate distributions that is obtained if we start, as we did in Example 23.3, with $[y, a, m] = [1/3, 1/3, 1/3]$ and repeatedly multiply by the matrix $\mathbf{M}$ for Fig. 23.3:

$$\begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}, \begin{bmatrix} 2/6 \\ 1/6 \\ 3/6 \end{bmatrix}, \begin{bmatrix} 3/12 \\ 2/12 \\ 7/12 \end{bmatrix}, \begin{bmatrix} 5/24 \\ 3/24 \\ 16/24 \end{bmatrix}, \begin{bmatrix} 8/48 \\ 5/48 \\ 35/48 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

That is, with probability 1, the walker will eventually wind up at the Microsoft page and stay there. □

If we interpret these PageRank probabilities as "importance" of pages, then the Microsoft page has gathered all importance to itself simply by choosing not to link outside. That situation intuitively violates the principle that other pages, not you yourself, should determine your importance on the Web. The other problem we mentioned — dead ends — also cause the PageRank not to reflect importance of pages, as we shall see in the next example.

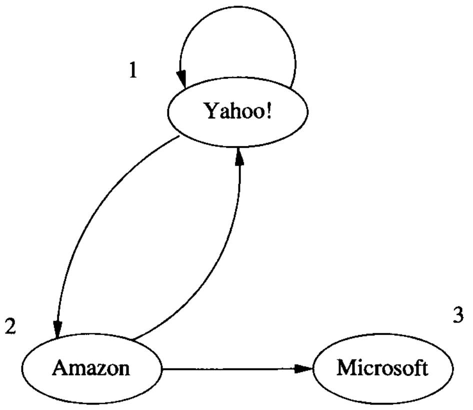

Figure 23.4: Microsoft becomes a dead end

Example 23.5: Suppose that instead of linking to itself, Microsoft links nowhere, as suggested in Fig. 23.4. The matrix **M** for this Web graph is

$$\mathbf{M} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 0 \end{bmatrix}$$

Notice that this matrix is not stochastic, because its columns do not all add up to 1. If we try to apply the method of relaxation to this matrix, with initial vector [1/3, 1/3, 1/3], we get the sequence:

$$\begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}, \begin{bmatrix} 2/6 \\ 1/6 \\ 1/6 \end{bmatrix}, \begin{bmatrix} 3/12 \\ 2/12 \\ 1/12 \end{bmatrix}, \begin{bmatrix} 5/24 \\ 3/24 \\ 2/24 \end{bmatrix}, \begin{bmatrix} 8/48 \\ 5/48 \\ 3/48 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

That is, the walker will eventually arrive at Microsoft, and at the next step has nowhere to go. Eventually, the walker disappears. □

23.2.4 PageRank Accounting for Spider Traps and Dead Ends

The solution to both spider traps and dead ends is to limit the time the random walker is allowed to wander at random. We pick a constant $\beta < 1$, typically in the range 0.8 to 0.9, and at each step, we let the walker follow a random out-link, if there is one, with probability β. With probability $1 - \beta$ (called the *taxation rate*), we remove that walker and deposit a new walker at a randomly chosen Web page. This modification solves both problems.

- If the walker gets stuck in a spider trap, it doesn't matter, because after a few time steps, that walker will disappear and be replaced by a new walker.
- If the walker reaches a dead end and disappears, a new walker will take over shortly.

Example 23.6: Let us use $\beta = 0.8$ and reformulate the calculation of PageRank for the Web of Fig. 23.3. If $\mathbf{p}_{new}$ and $\mathbf{p}_{old}$ are the new and old distributions of the location of the walker after one iteration, the relationship between these two can be expressed as:

$$\mathbf{p}_{new} = 0.8 \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \mathbf{p}_{old} + 0.2 \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

That is, with probability 0.8, we multiply $\mathbf{p}_{old}$ by the matrix of the Web to get the new location of the walker, and with probability 0.2 we start with a new walker at a random place. If we start with $\mathbf{p}_{old} = [1/3, 1/3, 1/3]$ and repeatedly compute $\mathbf{p}_{new}$ and then replace $\mathbf{p}_{old}$ by $\mathbf{p}_{new}$, we get the following sequence of approximations to the asymptotic distribution of the walker:

$$\begin{bmatrix} .333 \\ .333 \\ .333 \end{bmatrix}, \begin{bmatrix} .333 \\ .200 \\ .467 \end{bmatrix}, \begin{bmatrix} .280 \\ .200 \\ .520 \end{bmatrix}, \begin{bmatrix} .259 \\ .179 \\ .563 \end{bmatrix}, \ldots, \begin{bmatrix} 7/33 \\ 5/33 \\ 21/33 \end{bmatrix}$$

Notice that Microsoft, because it is a spider trap, gets a large share of the importance. However, the effect of the spider trap has been mitigated considerably by the policy of redistributing the walker with probability 0.2. $\square$

The same idea fixes dead ends as well as spider traps. The resulting matrix that describes transitions is substochastic, since a column will sum to 0 if there are no out-links. Thus, there will be a small probability that the walker is "nowhere" at any given time. That is, the sums of the probabilities of the walker being at each of the pages will be less than one. However, the relative sizes of the probabilities will still be a good measure of the importance of the page.

Teleportation of Walkers

Another view of the random-walking process is that there are no "new" walkers, but rather the walker *teleports* to a random page with probability $1-\beta$. For this view to make sense, we have to assume that if the walker is at a dead end, then the probability of teleport is 100%. Equivalently, we can scale up the probabilities to sum to one at each step of the iteration. Doing so does not affect the ratios of the probabilities, and therefore the relative PageRank of pages remains the same. For instance, in Example 23.7, the final pageRank vector would be $[35/81, 25/81, 21/81]$.

Example 23.7: Let us reconsider Example 23.5, using $\beta = 0.8$. The formula for iteration is now:

$$\mathbf{p}_{new} = 0.8 \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 0 \\ 0 & 1/2 & 0 \end{bmatrix} \mathbf{p}_{old} + 0.2 \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

Starting with $\mathbf{p}_{old} = [1/3, 1/3, 1/3]$, we get the following sequence of approximations to the asymptotic distribution of the walker:

$$\begin{bmatrix} .333 \\ .333 \\ .333 \end{bmatrix}, \begin{bmatrix} .333 \\ .200 \\ .200 \end{bmatrix}, \begin{bmatrix} .280 \\ .200 \\ .147 \end{bmatrix}, \begin{bmatrix} .259 \\ .179 \\ .147 \end{bmatrix}, \ldots, \begin{bmatrix} 35/165 \\ 25/165 \\ 21/165 \end{bmatrix}$$

Notice that these probabilities do not sum to one, and there is slightly more than 50% probability that the walker is "lost" at any given time. However, the ratio of the importances of Yahoo!, and Amazon are the same as in Example 23.6. That makes sense, because in neither Fig. 23.3 nor Fig. 23.4 are there links from the Microsoft page to influence the importance of Yahoo! or Amazon. □

23.2.5 Exercises for Section 23.2

Exercise 23.2.1: Compute the PageRank of the four nodes in Fig. 23.5, assuming no "taxation."

Exercise 23.2.2: Compute the PageRank of the four nodes in Fig. 23.5, assuming a taxation rate of: (a) 10% (b) 20%.

Exercise 23.2.3: Repeat Exercise 23.2.2 for the Web graph of

i. Fig. 23.6.

ii. Fig. 23.7.

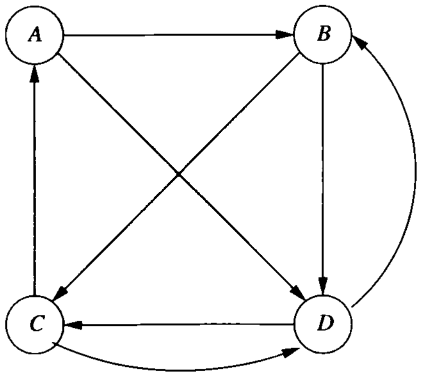

Figure 23.5: A Web graph with no dead-ends or spider traps

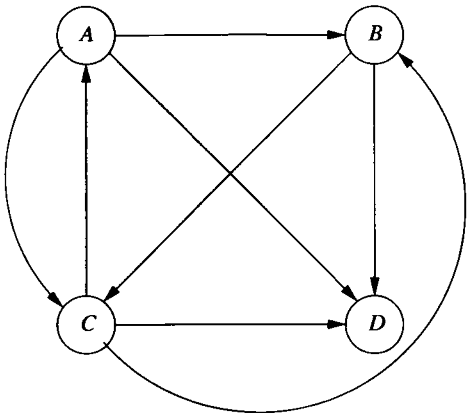

Figure 23.6: A Web graph with a dead end

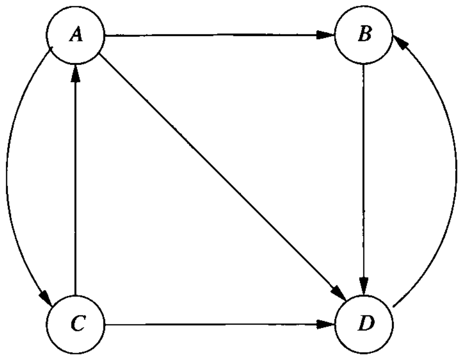

Figure 23.7: A Web graph with a spider trap

! **Exercise 23.2.4:** Suppose that we want to use the map-reduce framework of Section 20.2 to compute one iteration of the PageRank computation. That is, we are given data that represents the transition matrix of the Web and the current estimate of the PageRank for each page, and we want to compute the next estimate by multiplying the old estimate by the matrix of the Web. Suppose it is possible to break the data into chunks that correspond to sets of pages — that is, the PageRank estimates for those pages and the columns of the matrix for the same pages. Design map and reduce functions that implement the iteration, so that the computation can be partitioned onto any number of processors.

23.3 Topic-Specific PageRank

The calculation of PageRank is unbiased as to the content of pages. However, there are several reasons why we might want to bias the calculation to favor certain pages. For example, suppose we are interested in answering queries only about sports. We would want to give a higher PageRank to a page that discusses some sport than we would to another page that had similar links from the Web, but did not discuss sports. Or, we might want to detect and eliminate "spam" pages — those that were placed on the Web only to increase the PageRank of some other pages, or which were the beneficiaries of such planned attempts to increase PageRank illegitimately.

In this section, we shall show how to modify the PageRank computation to favor pages of a certain type. We then show how the technique yields solutions to the two problems mentioned above.

23.3.1 Teleport Sets

In Section 23.2.4, we "taxed" each page $1-\beta$ of its estimated PageRank and distributed the tax equally among all pages. Equivalently, we allowed random walkers on the graph of the Web to choose, with probability $1-\beta$, to teleport to a randomly chosen page. We are forced to have some taxation scheme in any calculation of PageRank, because of the presence of dead-ends and spider traps on the Web. However, we are not obliged to distribute the tax (or random walkers) equally. We could, instead, distribute the tax or walkers only among a selected set of nodes, called the *teleport set*. Doing so has the effect not only of increasing the PageRank of nodes in the teleport set, but of increasing the PageRank of the nodes they link to, and with diminishing effect, the nodes reachable from the teleport set by paths of lengths two, three, and so on.

Example 23.8: Let us reconsider the original Web graph of Fig. 23.2, which we reproduce here as Fig. 23.8. Assume we are interested only in retail sales, so we chose a teleport set that consists of Amazon alone. We shall use $\beta = 0.8$, i.e., a taxation rate of 20%. If y, a, and m are variables representing the PageRanks

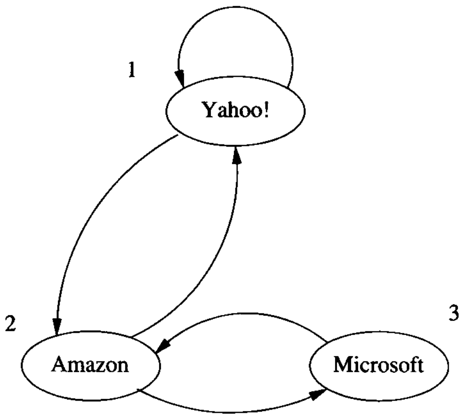

Figure 23.8: Web graph for Example 23.8

of Yahoo!, Amazon, and Microsoft, respectively, then the equations we need to solve are:

$$\begin{bmatrix} y \\ a \\ m \end{bmatrix} = 0.8 \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} y \\ a \\ m \end{bmatrix} + 0.2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The vector $[0, 1, 0]$ added at the end represents the fact that all the tax is distributed equally among the members of the teleport set. In this case, there is only one member of the teleport set, so the vector has 1 for that member (Amazon) and 0's elsewhere. We can solve the equations by relaxation, as we have done before. However, the example is small enough to apply Gaussian elimination and get the exact solution; it is $y = 10/31$, $a = 15/31$, and $m = 6/31$. The expected thing has happened; the PageRank of Amazon is elevated, because it is a member of the teleport set. □

The general rule for setting up the equations in a topic-specific PageRank problem is as follows. Suppose there are k pages in the teleport set. Let $\mathbf{t}$ be a column-vector that has $1/k$ in the positions corresponding to members of the teleport set and 0 elsewhere. Let $1 - \beta$ be the taxation rate, and let $\mathbf{M}$ be the transition matrix of the Web. Then we must solve by relaxation the following iterative rule:

$$\mathbf{p}_{new} = \beta \mathbf{M} \mathbf{p}_{old} + (1 - \beta)\mathbf{t}$$

Example 23.8 was an illustration of this process, although we set both $\mathbf{p}_{new}$ and $\mathbf{p}_{old}$ to $[y, a, m]$ and solved for the fixedpoint of the equations, rather than iterating to converge to the solution.

23.3.2 Calculating A Topic-Specific PageRank

Suppose we had a set of pages that we were certain were about a particular topic, say sports. We make these pages the teleport set, which has the effect of increasing their PageRank. However, it also increases the PageRank of pages linked to by pages in the teleport set, the pages linked to by those pages, and so on. We hope that many of these pages are also about sports, even if they are not in the teleport set. For example, the page `mlb.com`, the home page for major-league baseball, would probably be in the teleport set for the sports topic. That page links to many other pages on the same site — pages that sell baseball-related products, offer baseball statistics, and so on. It also links to news stories about baseball. All these pages are, in some sense, about sports.

Suppose we issue a search query "batter." If the PageRank that the search engine uses to rank the importance of pages were the general PageRank (i.e., the version where all pages are in the teleport set), then we would expect to find pages about baseball batters, but also cupcake recipes. If we used the PageRank that is specific to sports, i.e., one where only sports pages are in the teleport set, then we would expect to find, among the top-ranked pages, nothing about cupcakes, but only pages about baseball or cricket.

It is not hard to reason that the home page for a major-league sport will be a good page to use in the teleport set for sports. However, we might want to be sure we got a good sample of pages that were about sports into our teleport set, including pages we might not think of, even if we were an expert on the subject. For example, starting at major-league baseball might not get us to pages for the Springfield Little League, even though parents in Springfield would want that page in response to a search involving the words "baseball" and "Springfield." To get a larger and wider selection of pages on sports to serve as our teleport set, some approaches are:

1. Start with a curated selection of pages. For example, the Open Directory (`www.dmoz.org`) has human-selected pages on sixteen topics, including sports, as well as many subtopics.

2. Learn the keywords that appear, with unusually high frequency, in a small set of pages on a topic. For instance if the topic were sports, we would expect words like "ball," "player," and "goal" to be among the selected keywords. Then, examine the entire Web, or a larger subset thereof, to identify other pages that also have unusually high concentrations of some of these keywords.

The next problem we have to solve, in order to use a topic-specific PageRank effectively, is determining which topic the user is interested in. Several possibilities exist.

a) The easiest way is to ask the user to select a topic.

b) If we have keywords associated with different topics, as described in (2) above, we can try to discover the likely topic on the user's mind. We can examine pages that we think are important to the user, and find, in these pages, the frequency of keywords that are associated with each of the topics. Topics whose keywords occur frequently in the pages of interest are assumed to be the preference(s) of the user. To find these "pages of interest," we might:

i. Look at the pages the user has bookmarked.

ii. Look at the pages the user has recently searched.

23.3.3 Link Spam

Another application of topic-specific PageRank is in combating "link spam." Because it is known that many search engines use PageRank as part of the formula to rank pages by importance, it has become financially advantageous to invest in mechanisms to increase the PageRank of your pages. This observation spawned an industry: *spam farming.* Unscrupulous individuals create networks of millions of Web pages, whose sole purpose is to accumulate and concentrate PageRank on a few pages.

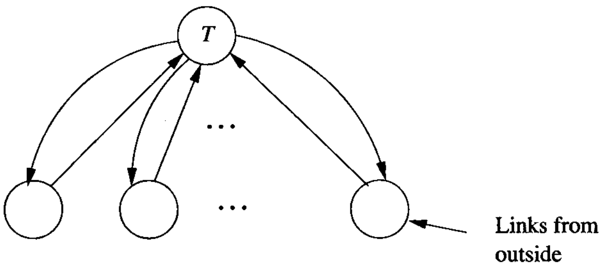

Figure 23.9: A spam farm concentrates PageRank in page T

A simple structure that accumulates PageRank in a target page T is shown in Fig. 23.9. Suppose that, in a PageRank calculation with taxation $1 - \beta$, the pages shown in the bottom row of Fig. 23.9 get, from the outside, a total PageRank of r, and let the total PageRank of these pages be x. Also, let the PageRank of page T be t. Then, in the limit, $t = \beta x$, because T gets all the PageRank of the other pages, except for the tax. Also, $x = r + \beta t$, because the other pages collectively get r from the outside and a total of βt from T. If we solve these equations for t, we get $t = \beta r/(1-\beta^2)$. For instance, if $\beta = .85$, then we have amplified the external PageRank by factor $0.85/\left(1 - (0.85)^2\right) = 3.06$. Moreover, we have concentrated this PageRank in a single page, T.

Of course, if $r = 0$ then T still gets no PageRank at all. In fact, it is cut off from the rest of the Web and would be invisible to search engines. However, it is not hard for spam farmers to get a reasonable value for r. As one example, they create links to the spam farm from publicly accessible blogs, with messages like "I agree with you. See x123456.mySpamFarm.com." Moreover, if the number of pages in the bottom row is large, and the "tax" is distributed among all pages, then r will include the share of the tax that is given to these pages. That is why spam farmers use many pages in their structure, rather than just one or two.

23.3.4 Topic-Specific PageRank and Link Spam

A search engine needs to detect pages that are on the Web for the purpose of creating link spam. A useful tool is to compute the *TrustRank* of pages. Although the original definition is somewhat different, we may take the TrustRank to be the topic-specific PageRank computed with a teleport set consisting of only "trusted" pages. Two possible methods for selecting the set of trusted pages are:

1. Examine pages by hand and do an evaluation of their role on the Web. It is hard to automate this process, because spam farmers often copy the text of perfectly legitimate pages and populate their spam farm with pages containing that text plus the necessary links.

2. Start with a teleport set that is likely to contain relatively little spam. For example, it is generally believed that the set of university home pages form a good choice for a widely distributed set of trusted pages. In fact, it is likely that modern search engines routinely compute PageRank using a teleport set similar to this one.

Either of these approaches tends to assign lower PageRank to spam pages, because it is rare that a trusted page would link to a spam page. Since TrustRank, like normal PageRank, is computed with a positive taxation factor $1 - \beta$, the trust imparted by a trusted page attenuates, the further we get from that trusted page. The TrustRank of pages may substitute for PageRank, when the search engine chooses pages in response to a query. So doing reduces the likelihood that spam pages will be offered to the queryer.

Another approach to detecting link-spam pages is to compute the *spam mass* of pages as follows:

a) Compute the ordinary PageRank, that is, using all pages as the teleport set.

b) Compute the TrustRank of all pages, using some reasonable set of trusted pages.

c) Compute the difference between the PageRank and TrustRank for each page. This difference is the *negative TrustRank*.

d) The spam mass of a page is the ratio of its negative TrustRank to its ordinary PageRank, that is, the fraction of its PageRank that appears to come from spam farms.

While TrustRank alone can bias the PageRank to minimize the effect of link spam, computing the spam mass also allows us to see where the link spam is coming from. Sites that have many pages with high spam mass may be owned by spam farmers, and a search engine can eliminate from its database all pages from such sites.

23.3.5 Exercises for Section 23.3

Exercise 23.3.1: Compute the topic-specific PageRank for Fig. 23.5, assuming

a) Only A is in the teleport set.

b) The teleport set is $\{A, B\}$.

Assume a taxation rate of 20%.

Exercise 23.3.2: Repeat Exercise 23.3.1 for the graph of Fig. 23.6.

Exercise 23.3.3: Repeat Exercise 23.3.1 for the graph of Fig. 23.7.

!! **Exercise 23.3.4:** Suppose we fix the taxation rate and compute the topic-specific PageRank for a graph G, using only node a as the teleport set. We then do the same using only another node b as the teleport set. Prove that the average of these PageRanks is the same as what we get if we repeated the calculation with $\{a, b\}$ as the teleport set.

!! **Exercise 23.3.5:** What is the generalization of Exercise 23.3.4 to a situation where there are two disjoint teleport sets S_1 and S_2, perhaps with different numbers of elements? That is, suppose we compute the PageRanks with just S_1 and then just S_2 as the teleport sets. How could we use these results to compute the PageRank with $S_1 \cup S_2$ as the teleport set?

23.4 Data Streams

We now turn to an extension of the ideas contained in the traditional DBMS to deal with *data streams*. As the Internet has made communication among machines routine, a class of applications has developed that stress the traditional model of a database system. Recall that a typical database system is primarily a repository of data. Input of data is done as part of the query language or a special data-load utility, and is assumed to occur at a rate controlled by the DBMS.

However, in some applications, the inputs arrive at a rate the DBMS cannot control. For example, Yahoo! may wish to record every "click," that is, every page request made by any user anywhere. The sequence of URL's representing these requests arrive at a very high rate that is determined only by the desires of Yahoo!'s customers.

23.4.1 Data-Stream-Management Systems

If we are to allow queries on such streams of data, we need some new mechanisms. While we may be able to store the data on high-rate streams, we cannot do so in a way that allows instantaneous queries using a language like SQL. Further, it is not even clear what some queries mean; for instance, how can we take the join of two streams, when we never can see the completed streams? The rough structure of a data-stream-management system (*DSMS*) is shown in Fig. 23.10.

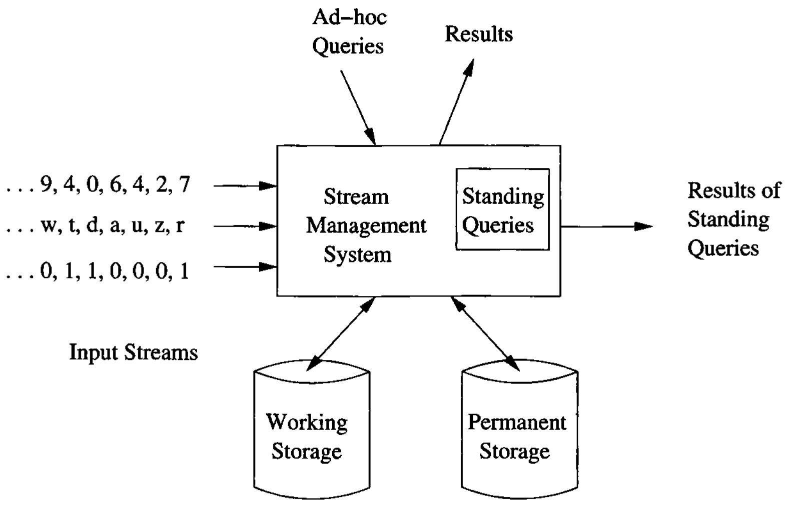

Figure 23.10: A data-stream-management system

The system accepts data streams as input, and also accepts queries. These queries may be of two kinds:

1. Conventional ad-hoc queries.

2. *Standing queries* that are stored by the system and run on the input stream(s) at all times.

Example 23.9: Whether ad-hoc or standing, queries in a DSMS need to be expressed so they can be answered using limited portions of the streams. As an example, suppose we are receiving streams of radiation levels from sensors around the world. While the DSMS cannot store and query streams from arbitrarily far back in time, it can store a *sliding window* of each input stream. It might be able to keep on disk, in the "working storage" referred to in Fig. 23.10, all readings from all sensors for the past 24 hours. Data from further back in time could be dropped, could be summarized (e.g., replaced by the daily average), or copied in its entirety to the permanent store (*archive*).

An ad-hoc query might ask for the average radiation level over the past hour for all locations in North Korea. We can answer this query, because we have all data from all streams over the past 24 hours in our working store. A standing query might ask for a notification if any reading on any stream exceeds a certain limit. As each data element of each stream enters the system, it is compared with the threshold, and an output is made if the entering value exceeds the threshold. This sort of query can be answered from the streams themselves, although we would need to examine the working store if, say, we asked to be alerted if the average over the past 5 minutes for any one stream exceeded the threshold. □

23.4.2 Stream Applications

Before addressing the mechanics of data-stream-management systems, let us look at some of the applications where the data is in the form of a stream or streams.

1. *Click Streams.* As we mentioned, a common source of streams is the clicks by users of a large Web site. A Web site might wish to analyze the clicks it receives for a number of reasons; an increase in clicks on a link may indicate that link is broken, or that it has become of much more interest recently. A search engine may want to analyze clicks on the links to ads that it shows, to determine which ads are most attractive.

2. *Packet Streams.* We may wish to analyze the sources and destinations of IP packets that pass through a switch. An unusual increase in packets for a destination may warn of a denial-of-service attack. Examination of the recent history of destinations may allow us to predict congestion in the network and to reroute packets accordingly.

3. *Sensor Data.* We also mentioned a hypothetical example of a network of radiation sensors. There are many kinds of sensors whose outputs need to be read and considered collectively, e.g., tsunami warning sensors that record ocean levels at subsecond frequencies or the signals that come from seismometers around the world, recording the shaking of the earth. Cities that have networks of security cameras can have the video from these cameras read and analyzed for threats.

4. *Satellite Data.* Satellites send back to earth incredible streams of data, often petabytes per day. Because scientists are reluctant to throw any of this data away, it is often stored in raw form in archival memory systems. These are half-jokingly referred to as "write-only memory." Useful products are extracted from the streams as they arrive and stored in more accessible storage places or distributed to scientists who have made standing requests for certain kinds of data.

5. *Financial Data.* Trades of stocks, commodities, and other financial instruments are reported as a stream of tuples, each representing one financial transaction. These streams are analyzed by software that looks for events or patterns that trigger actions by traders. The most successful traders have access to the largest amount of data and process it most quickly, because opportunities involving stock trades often last for only fractions of a second.

23.4.3 A Data-Stream Data Model

We shall now offer a data model useful for discussing algorithms on data streams. First, we shall assume the following about the streams themselves:

- Each stream consists of a sequence of tuples. The tuples have a fixed relation schema (list of attributes), just as the tuples of relations do. However, unlike relations, the sequence of tuples in a stream may be unbounded.

- Each tuple has an associated *arrival time*, at which time it becomes available to the data-stream-management system for processing. The DSMS has the option of placing it in the working storage or in the permanent storage, or of dropping the tuple from memory altogether. The tuple may also be processed in simple ways before storing it.

For any stream, we can define a *sliding window* (or just "window"), which is a set consisting of the most recent tuples to arrive. A window can be *time-based* with a constant τ, in which case it consists of the tuples whose arrival time is between the current time t and $t-\tau$. Or, a window can be *tuple-based*, in which case it consists of the most recent n tuples to arrive, for some fixed n.

We shall describe windows on a stream S by the notation S $[W]$, where W is the window description, either:

1. `Rows` n, meaning the most recent n tuples of the stream, or

2. `Range` τ, meaning all tuples that arrived within the previous amount of time τ.

Example 23.10: Let `Sensors(sensID, temp, time)` be a stream, each of whose tuples represent a temperature reading of `temp` at a certain `time` by the sensor named `sensID`. It might be more common for each sensor to produce its own stream, but all readings could also be merged into one stream if the data were accumulated outside the data-stream-management system. The expression

```
Sensors [Rows 1000]
```

describes a window on the `Sensors` stream consisting of the most recent 1000 tuples. The expression

```
Sensors [Range 10 Seconds]
```

describes a window on the same stream consisting of all tuples that arrived in the past 10 seconds. □

23.4.4 Converting Streams Into Relations

Windows allow us to convert streams into relations. That is, the window expressions as in Example 23.10 describe a relation at any time. The contents of the relation typically changes rapidly. For example, consider the expression `Sensors [Rows 1000]`. Each time a new tuple of `Sensors` arrives, it is inserted into the described relation, and the oldest of the tuples is deleted. For the expression `Sensors [Range 10 Seconds]`, we must insert tuples of the stream when they arrive and delete tuples 10 seconds after they arrive.

Window expressions can be used like relations in an extended SQL for streams. The following example suggests what such an extended SQL looks like.

Example 23.11: Suppose we would like to know, for each sensor, the highest recorded temperature to arrive at the DSMS in the past hour. We form the appropriate time-based window and query it as if it were an ordinary relation. The query looks like:

```
SELECT sensID, MAX(temp)
FROM Sensors [Range 1 Hour]
GROUP BY sensID;
```

This query can be issued as an ad-hoc query, in which case it is executed once, based on the window that exists at the instant the query is issued. Of course the DSMS must have made available to the query processor a window on `Sensors` of at least one hour's length.[2] The same query could be a standing query, in which case the current result relation should be maintained as if it were a materialized view that changes from time to time. In Section 23.4.5 we shall consider an alternative way to represent the result of this query as a standing query. □

Window relations can be combined with other window relations, or with "ordinary" relations — those that do not come from streams. An example will suggest what is possible.

Example 23.12: Suppose that our DSMS has the stream `Sensors` as an input stream and also maintains in its working storage an ordinary relation

```
Calibrate(sensID, mult, add)
```

[2]Strictly speaking, the DSMS only needs to have retained enough information to answer the query. For example, it could still answer the query at any time if it threw away every tuple for which there was a later reading from the same sensor with a higher temperature.

which gives a multiplicative factor and additive term that are used to correct the reading from each sensor. The query

```
SELECT MAX(mult*temp + add)
FROM Sensors [Range 1 Hour], Calibrate
WHERE Sensors.sensID = Calibrate.sensID;
```

finds the highest, properly calibrated temperature reported by any sensor in the past hour. Here, we have joined a window relation from Sensors with the ordinary relation Calibrate. □

We can also compute joins of window-relations. The following query illustrates a self-join by means of a subquery, but all the SQL tools for expressing joins are available.

Example 23.13: Suppose we wanted to give, for each sensor, its maximum temperature over the past hour (as in Example 23.11), but we also wanted the resulting tuples to give the most recent time at which that maximum temperature was recorded. Figure 23.11 is one way to write the query using window relations.

```
SELECT s.sensID, s.temp, s.time
FROM Sensors [Range 1 Hour] s
WHERE NOT EXISTS (
    SELECT * FROM Sensors [Range 1 Hour]
    WHERE sensID = s.sensID AND (
        temp > s.temp OR
            (temp = s.temp AND time > s.time)
    )
);
```

Figure 23.11: Including time with the maximum temperature readings of sensors

That is, the subquery checks if there is not another tuple in the window-relation Sensors [Range 1 Hour] that refers to the same sensor as the tuple *s*, and has either a higher temperature or has the same temperature but a more recent time. If no such tuple exists, then the tuple *s* is part of the result. □

23.4.5 Converting Relations Into Streams

When we issue queries such as that of Example 23.11 as standing queries, the resulting relations change frequently. Maintaining these relations as materialized views may result in a lot of effort making insertions and deletions that no one ever looks at. An alternative is to convert the relation that is the result of the query back into streams, which may be processed like any other streams.

For example, we can issue an ad-hoc query to construct the query result at a particular time when we are interested in its value.

If R is a relation, define `Istream(`R`)` to be the stream consisting of each tuple that is inserted into R. This tuple appears in the stream at the time the insertion occurs. Similarly, define `Dstream(`R`)` to be the stream of tuples deleted from R; each tuple appears in this stream at the moment it is deleted. An update to a tuple can be represented by an insertion and deletion at the same time.

Example 23.14: Let R be the relation constructed by the query of Example 23.13, that is, the relation that has, for each sensor, the maximum temperature it recorded in any tuple that arrived in the past hour, and the time at which that temperature was most recently recorded. Then `Istream(`R`)` has a tuple for every event in which a new tuple is added to R. Note that there are two events that add tuples to R:

1. A `Sensors` tuple arrives with a temperature that is at least as high as any tuple currently in R with the same sensor ID. This tuple is inserted into R and becomes an element of `Istream(`R`)` at that time.

2. The current maximum temperature for a sensor i was recorded an hour ago, and there has been at least one tuple for sensor i in the `Sensors` stream in the past hour. In that case, the new tuple for R and for `Istream(`R`)` is the `Sensors` tuple for sensor i that arrived in the past hour, but no other tuple for i that also arrived in the past hour has:

   (a) A higher temperature, or

   (b) The same temperature and a more recent time.

The same two events may generate tuples for the stream `Dstream(`R`)` as well. In (1) above, if there was any other tuple in R for the same sensor, then that tuple is deleted from R and becomes an element of `Dstream(`R`)`. In (2), the hour-old tuple of R for sensor i is deleted from R and becomes an element of `Dstream(`R`)`. □

If we compute the Istream and Dstream for a relation like that constructed by the query of Fig. 23.11, then we do not have to maintain that relation as a materialized view. Rather, we can query its Istream and Dstream to answer queries about the relation when we wish.

Example 23.15: Suppose we form the Istream I and the Dstream D for the relation R of Fig. 23.11. When we wish, we can issue an ad-hoc query to these streams. For instance, suppose we want to find the maximum temperature recorded by sensor 100 that arrived over the past hour. That will be the temperature in the tuple in I for sensor 100 that:

1. Has a `time` in the past hour.

2. Was not deleted from R (i.e., is not in D restricted to the past hour).

This query can be written as shown in Fig. 23.12. The keyword `Now` represents the current time.

Note that we must check that a tuple of I both arrived in the past hour and that it has a timestamp within the past hour. To see why these conditions are not the same, consider the case of a tuple of I that arrived in the past hour, because it became the maximum temperature t for sensor 100 thirty minutes ago. However, that temperature itself has an associated `time` that is eighty minutes ago. The reason is that a temperature higher than t was recorded by sensor 100 ninety minutes ago. It wasn't until 30 minutes ago that t became the highest temperature for sensor 100 in the sixty minutes preceding. □

```
(SELECT * FROM I [Range 1 Hour]
 WHERE sensID = 100 AND
     time >= [Now - 1 Hour])
    EXCEPT
(SELECT * FROM D [Range 1 Hour]
 WHERE sensID = 100);
```

Figure 23.12: Querying an Istream and a Dstream

23.4.6 Exercises for Section 23.4

Exercise 23.4.1: Using the `Sensors` stream from Example 23.11, write the following queries:

a) Find the oldest tuple (lowest `time`) among the last 1000 tuples to arrive.

b) Find those sensors for which at least two readings have arrived in the past minute.

! c) Find those sensors for which more readings arrived in the past minute than arrived between one and two minutes ago.

Exercise 23.4.2: Following the example of sensor data from this section, suppose that the following temperature-time readings are generated by sensor 100, and each arrives at the DSMS at the time generated: (80, 0), (70, 50), (60, 70), (65, 100). Times are in minutes. If R is the query of Fig. 23.11, What are the tuples of `Istream(`R`)` and `Dstream(`R`)`, and at what time is each of these tuples generated?

! **Exercise 23.4.3:** Suppose our stream consists of baskets of items, as in the market-basket model of Section 22.1.1. Since we assume elements of streams are tuples, the contents of a basket must be represented by several consecutive tuples with the schema `Baskets(basket, item)`. Write the following queries:

a) Find those items that have appeared in at least 1% of the baskets that arrived over the past hour.[3]

b) Find those pairs of items that have appeared in at least twice as many baskets in the previous half hour as in the half hour before that.

c) Find the most frequent pair(s) of items over the past hour.

23.5 Data Mining of Streams

When processing streams, there are a number of problems that become quite hard, even though the analogous problems for relations are easy. In this section, we shall concentrate on representing the contents of windows more succinctly than by listing the current set of tuples in the window. Surely, we are not then able to answer all possible queries about the window, but if we know what kinds of queries we are expected to support, we might be able to compress the window and answer those queries. Another possibility is that we cannot compress the window and answer our selected queries exactly, but we can guarantee to be able to answer them within a fixed error bound.

We shall consider two fundamental problems of this type. First, we consider binary streams (streams of 0's and 1's), and ask whether we can answer queries about the number of 1's in any time range contained within the window. Obviously, if we keep the exact sequence of bits and their timestamps, we can manage to answer those questions exactly. However, it is possible to compress the data significantly and still answer this family of queries within a fixed error bound. Second, we address the problem of counting the number of different values within a sliding window. Here is another family of problems that cannot be answered exactly without keeping the data in the window exactly. However, we shall see that a good approximation is possible using much less space than the size of the window.

23.5.1 Motivation

Suppose we wish to have a stream with a window of a billion integers. Such a window could fit in a large main memory of four gigabytes, and it would have no trouble fitting on disk. Surely, if we are only interested in recent data from the stream, a billion tuples should suffice. But what if there are a million such streams?

For example, we might be trying to integrate the data from a million sensors placed around a city. Or we might be given a stream of market baskets, and try to compute the frequency, over any time range, of all sets of items contained in

[3]Technically, some but not all of a basket could arrive within the past hour. Ignore this "edge effect," and assume that either all or none of a basket's tuples appear in any given window.

those baskets. In that case, we need a window for each set, with bits indicating whether or not that set was contained in each of the baskets.

In situations such as these, the amount of space needed to store all the windows exceeds what is available using disk storage. Moreover, for efficient response, we might want to keep all windows in main memory. Then, a few windows of length a billion, or a few thousand windows of length a million exceed what even a large main memory can hold. We are thus led to consider compressing the data in windows. Unfortunately, even some very simple queries cannot be answered if we compress the window, as the next example suggests.

Example 23.16: Suppose we have a sliding window that stores stream elements that are integers, and we have a standing query that asks for an alert any time the sum of the integers in the window exceeds a certain threshold t. We thus only need to maintain the sum of the integers in the window in order to answer this query. When a new integer comes in, we can add it to the sum.

However, at certain times, integers leave the window and must be subtracted from the sum. If the window is tuple-based, then we must subtract the last integer from the sum each time a new integer arrives. If the window is time-based, then when the time of an integer in the window expires, it must be subtracted from the sum.

Unfortunately, if we don't know exactly what integers are in the window, or we don't know their order of arrival (for tuple-based windows) or their time of arrival (for time-based windows), then we cannot maintain the sum properly. To see why we cannot compress, observe the following. If there is any compression at all, then two different window-contents, W_1 and W_2, must have the same compressed value. Since $W_1 \neq W_2$, there is some time t at which the integers for time t are different in W_1 and W_2. Consider what happens when t is the oldest time in the window, and another integer arrives. We must have to do different subtractions from the sum, to maintain the sums for W_1 and W_2. But since the compressed representation does not tell us which of W_1 and W_2 is the true contents of the window, we cannot maintain the proper sum in both cases. □

Example 23.16 tells us that we cannot compress the sum of a sliding window if we are to get exact answers for the sum at all times. However, suppose we are willing to accept an approximate sum. Then there are many options, and we shall look at a very simple one here. We can group the stream elements into groups of 100; say the first hundred elements of the stream ever to arrive, then the next hundred, and so on. Each group is represented by the sum of elements in that group. Thus, we have a compression factor of 100; i.e., the window is represented by $1/100^{\text{th}}$ of the number of integers that are theoretically "in" in window.

Suppose for simplicity that we have a tuple-based window, and the number of tuples in the window is a multiple of 100. When the number of stream elements that have arrived is also a multiple of 100, then we can get the sum of the elements in the window exactly, just by summing the sums of the groups.

Suppose another integer arrives. That integer starts another group, so we keep it as the sum of that group. Now, we can only estimate the sum of all the integers in the window. The reason is that the last group has only 99 of its 100 members in the window, and we don't know the value of the integer, from the last group, that is no longer in the window.

The best estimate of the deleted integer is 1% of the sum of the last group. That is, we estimate the sum of all the integers in the window by taking 0.99 times the recorded sum of the last group, plus the recorded sums of all the other groups.

Forty-nine arrivals later, there are fifty integers in the group formed from the most recent arrivals, and the sum of the window includes exactly half of the last group. Our best estimate of the sum of the fifty integers of the last group that remain in the window is half the group's sum. After another fifty arrivals, the most recent group is complete, and the last group has left the window entirely. We therefore can drop the recorded sum of the last group and prepare to start another group with the next arrival.

Intuitively, this method gives a "good" approximation to the sum. If integers are nonnegative, and there is not too much variance in the values of the integers, then assuming that the missing integers are average for their group is a close estimate. Unfortunately, if the variance is high, or integers can be both positive and negative, there is no worst-case bound on how bad the estimate of the sum can be. Consider what happens if integers can range from minus infinity to plus infinity, and the last group consists of fifty large negative numbers followed by fifty large positive numbers, such that the sum for the group is 0. Then the estimate of the contribution of the last group, when only half of it is in the window is zero, but in fact the true sum is very large — perhaps much larger than the sum of all the integers that followed them in the stream.

One can modify this compression approach in various ways. For example, we can increase the size of the groups to reduce the amount of space taken by the representation. Doing so increases the error in the estimate, however. In the next section, we shall see how to get a bounded error rate, while getting significant compression, for the binary version of this problem, where stream elements are either 0 or 1. The same method extends to streams of positive integers with an upper bound, if we treat each position in the binary representation of the integers as a bit stream (see Exercise 23.5.3).

23.5.2 Counting Bits

In this section, we shall examine the following problem. Assume that the length of the sliding window is N, and the stream consists of bits, 0 or 1. We assume that the stream began at some time in the past, and we associate a ***time*** with each arriving bit that is its position in the stream; i.e., the first to arrive is at time 1, the next at time 2, and so on.

Our queries, which may be asked at any time, are of the form "how many 1's are there in the most recent k bits?" where k is any integer between 1 and

N. Obviously, if we stored the window with no compression, we could answer any such query exactly, although we would have to sum the last k bits to do so. Since k could be very large, the time needed to answer queries could itself be large. Suppose, however, that along with the bits themselves we stored the sums of certain groups of consecutive bits — groups of size 2, 4, 8,.... We could then decrease the time needed to answer the queries exactly to $O(\log N)$. However, if we also stored sums of these groups, then even more space would be needed than what we use to store the window elements themselves.

An attractive alternative is to keep an amount of information about the window that is logarithmic in N, and yet be able to answer any query of the type described above, with a fractional error that is as low as we like. Formally, for any $\epsilon > 0$, we can produce an estimate that is in the range of $1 - \epsilon$ to $1 + \epsilon$ times the true result. We shall give the method for $\epsilon = 1/2$, and we leave the generalization to any $\epsilon > 0$ as an exercise with hints (see Exercise 23.5.4).

Buckets

To describe the algorithm for approximate counting of 1's, we need to define a *bucket* of size m; it is a section of the window that contains exactly m 1's. The window will be partitioned completely into such buckets, except possibly for some 0's that are not part of any bucket. Thus, we can represent any such bucket by (m, t), where m is the size of the bucket, and t is the time of the most recent 1 belonging to that bucket. There are a number of rules that we shall follow in determining the buckets that represent the current window:

1. The size of every bucket is a power of 2.

2. As we look back in time, the sizes of the buckets never decrease.

3. For $m = 1, 2, 4, 8, \ldots$ up to some largest-size bucket, there are one or two buckets of each size, never zero and never more than two.

4. Each bucket begins somewhere within the current window, although the last (largest) bucket maybe partially outside the window.

Figure 23.13 suggests what a window partitioned into buckets might look like.

Representing Buckets

We shall see that under these assumptions, a bucket can be represented by $O(\log N)$ bits. Further, there are at most $O(\log N)$ buckets that must be represented. Thus, a window of length N can be represented in space $O(\log^2 N)$, rather than $O(N)$ bits. To see why only $O(\log^2 N)$ bits are needed, observe the following:

- A bucket (m, t) can be represented in $O(\log N)$ bits. First, m, the size of a bucket, can never get above N. Moreover, m is always a power of 2, so

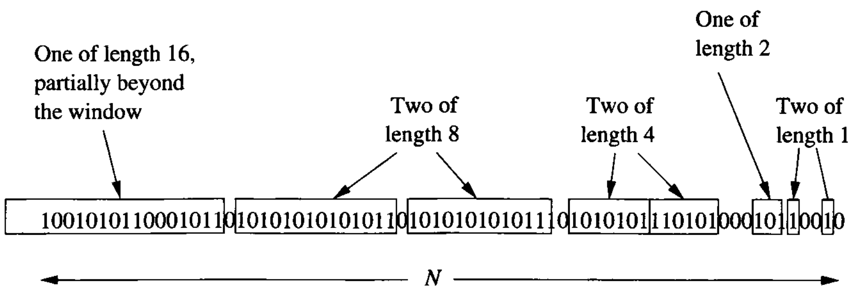

Figure 23.13: Bucketizing a sliding window

we don't have to represent m itself; rather we can represent $\log_2 m$. That requires $O(\log\log N)$ bits. However, we also need to represent t, the time of the most recent 1 in the bucket. In principle, t can be an arbitrarily large integer, but it is sufficient to represent t modulo N, since we know t has to be in the window of length N. Thus, $O(\log N)$ bits suffice to represent both m and t. So that we can know the time of newly arriving 1's, we maintain the current time, but also represent it modulo N, so $O(\log N)$ bits suffice for this count.

- There can be only $O(\log N)$ buckets. The sum of the sizes of the buckets is at most N, and there can be at most two of any size. If there are more than $2 + 2\log_2 N$ buckets, then the largest one is of size at least $2 \times 2^{\log_2 N}$, which is $2N$. There must be a smaller bucket of half that size, so the supposed largest bucket is certainly completely outside the window.

Answering Queries Approximately, Using Buckets

Notice that we can answer a query to count the 1's in the most recent k bits approximately, as follows. Find the least recent bucket B whose most recent bit arrived within the last k time units. All later buckets are entirely within the range of k time units. We know exactly how many 1's are in each of these buckets; it is their size. The bucket B is partially in the query's range, and partially outside it. We cannot tell how much is in and how much is out, so we choose half its size as the best guess.

Example 23.17: Suppose $k = N$ and the window is represented by the buckets of Fig. 23.13. We see two buckets of size 1 and one of size 2, which implies four 1's. Then, there are two buckets of size 4, giving another eight 1's, and two buckets of size 4, implying another sixteen 1's. Finally, the last bucket, of size 16, is partially in the window, so we add another 8 to the estimate. The approximate answer is thus $2 \times 1 + 1 \times 2 + 2 \times 4 + 2 \times 8 + 8 = 36$. □

Maintaining Buckets

There are two reasons the buckets change as new bits arrive. The first is easy to handle: if a new bit arrives, and the last bucket now has a most recent bit that is more than N lower than the time of the arriving bit, then we can drop that bucket from the representation. Such a bucket can never be part of the answer to any query.

Now, suppose a new bit arrives. If the bit is a 0, there are no changes, except possibly the deletion of the last bucket as mentioned above. Suppose the new bit is a 1. We create a new bucket of size 1 representing just that bit. However, we may now have three buckets of size 1, which violates the rule that there can be only one or two buckets of each size. Thus, we enter a recursive combining-buckets phase.

Suppose we have three consecutive buckets of size m, say (m, t_1), (m, t_2), and (m, t_3), where $t_1 < t_2 < t_3$. We combine the two least recent of the buckets, (m, t_1) and (m, t_2), into one bucket of size $2m$. The time of the most recent bit for the combined bucket is that of the most recent bit for the more recent of the two combined buckets. That is, (m, t_1) and (m, t_2) are replaced by a bucket $(2m, t_2)$.

This combination may cause there to be three consecutive buckets of size $2m$, if there were two of that size previously. Thus, we apply the combination algorithm recursively, with the size now $2m$. It can take no more than $O(\log N)$ time to do all the necessary combinations.

Example 23.18: Suppose we have the list of bucket sizes implied by Fig. 23.13, that is, $16, 8, 8, 4, 4, 2, 1, 1$. If a 1 arrives, we have three buckets of size 1, so we combine the two earlier 1's, to get the list $16, 8, 8, 4, 4, 2, 2, 1$. As this combination gives us only two buckets of size 2, no recursive combining is needed. If another 1 arrives, no combining at all is needed, and we get sequence of bucket sizes $16, 8, 8, 4, 4, 2, 2, 1, 1$. When the next 1 arrives, we must combine 1's, leaving $16, 8, 8, 4, 4, 2, 2, 2, 1$. Now we have three 2's, so we recursively combine the least recent of them, leaving $16, 8, 8, 4, 4, 4, 2, 1$. Now there are three 4's, and the least recent of them are combined to give $16, 8, 8, 8, 4, 2, 1$. Again, we must combine the least recent of the three 8's, giving us the final list of bucket sizes $16, 16, 8, 4, 2, 1$. □

A Bound on the Error

Suppose that in answer to a query the last bucket whose represented 1's are in the range of the query has size m. Since we estimate $m/2$ for its contribution to the count, we cannot be off by more than $m/2$. The correct answer is at least the sum of all the smaller buckets, and there is at least one bucket of each size $m/2, m/4, m/8, \ldots, 1$. This sum is $m - 1$. Thus, the fractional error is at most $(m/2)/(m - 1)$, or approximately 50%. In fact, if we look more carefully, 50% is an exact upper bound. The reason is that when we underestimate (i.e., all m 1's from the last bucket are in the query range), the error is no more than $1/3$.

When we overestimate, we can really only overestimate by $(m/2) - 1$, not $m/2$, since we know that at least one 1 contributes to the query. Since $(m/2) - 1$ is less than half $m - 1$, the error is truly upper bounded by 50%.

23.5.3 Counting the Number of Distinct Elements

We now turn to another important problem: counting the distinct elements in a (window on) a stream. The problem has a number of applications, such as the following:

1. The popularity of a Web site is often measured by unique visitors per month or similar statistics. Think of the logins at a site like Yahoo! as a stream. Using a window of size one month, we want to know how many different logins there are.

2. Suppose a crawler is examining sites. We can think of the words encountered on the pages as forming a stream. If a site is legitimate, the number of distinct words will fall in a range that is neither too high (few repetitions of words) nor too low (excessive repetition of words). Falling outside that range suggests that the site could be artificial, e.g., a spam site.

To get an exact answer to the question, we must store the entire window and apply the δ operator to it, in order to find the distinct elements. However, we don't want to *see* the distinct elements; we just want to know how many there are. Even getting this count requires that we maintain the window in its entirety, but we can get an approximation to the count by several different methods. The following technique actually computes the number of distinct elements in the entire stream, rather than in a finite window. However, we can, if we like, restart the process periodically, e.g., once a month to count unique visitors or each time we visit a new site (to count distinct words).

The necessary tools are a number N that is certain to be at least as large as the number of distinct values in the stream, and a hash function h that maps values to $\log_2 N$ bits. We maintain a number R that is initially 0. As each stream value v arrives, do the following:

1. Compute $h(v)$.

2. Let r be the number of trailing 0's in $h(v)$.

3. If $r > R$, set R to be r.

Then, the estimate of the number of distinct values seen so far is 2^R. To see why this estimate makes sense, note the following.

a) The probability that $h(v)$ ends in at least i 0's is 2^{-i}.

b) If there are m distinct elements in the stream so far, the probability that $R > i$ is $(1 - 2^{-i})^m$.

c) If i is much less than $\log_2 m$, then this probability is close to 1, and if i is much greater than $\log_2 m$, then this probability is close to 0.

d) Thus, R will frequently be near $\log_2 m$, and 2^R, our estimate, will frequently be near m.

While the above reasoning is comforting, it is actually inaccurate, to say the least. The reason is that the expected value of 2^R is infinite, or at least it is as large as possible given that N is finite. The intuitive reason is that, for large R, when R increases by 1, the probability of R being that large halves, but the value of R doubles, so each possible value of R contributes the same to the expected value.

It is therefore necessary to get around the fact that there will occasionally be a value of R that is so large it biases the estimate of m upwards. While we shall not go into the exact justification, we can avoid this bias by:

1. Take many estimates of R, using different hash functions.
2. Group these estimates into small groups and take the median of each group. Doing so eliminates the effect of occasional large R's.
3. Take the average of the medians of the groups.

23.5.4 Exercises for Section 23.5

Exercise 23.5.1: Starting with the window of Fig. 23.13, suppose that the next ten bits to arrive are all 1's. What will be the sequence of buckets at that time?

Exercise 23.5.2: What buckets are used in Fig. 23.13 to answer queries of the form "how many 1's in the most recent k bits?" if k is (a) 10 (b) 15 (c) 20? What are the estimates for each of these queries? How close are the estimates?

! **Exercise 23.5.3:** Suppose that we have a stream of integers in the range 0 to 1023. How can you adapt the method of Section 23.5.2 to estimate the sum of the integers in a window of size N, keeping the error to 50%? *Hint*: treat each of the ten bits that represent an integer as a separate stream.

! **Exercise 23.5.4:** We can modify the algorithm of Section 23.5.2 to use buckets whose sizes are powers of 2, but there are between p and $p+1$ buckets of each size, for a chosen integer $p \geq 1$. As before, sizes do not decrease as we go further back in time.

a) Give the recursive rule for combining buckets when there are too many buckets of a given size.

b) Show that the fractional error of this scheme is at most $1/2p$.

Exercise 23.5.5: Suppose that we wish to estimate the number of distinct values in a stream of integers. The integers are in the range 0 to 1023. We'll use the following hash functions, each of which hashes to a 9-bit integer:

a) $h_1(v) = v$ modulo 512.

b) $h_2(v) = v + 159$ modulo 512.

c) $h_3(v) = v + 341$ modulo 512.

Compute the estimate of the number of distinct values in the following stream, using each of these hash functions:

$$24, 45, 102, 24, 78, 222, 45, 24, 670, 78, 999, 576, 222, 24$$

Exercise 23.5.6: In Example 23.11 we observed that if all we wanted was the maximum of N temperature readings in a sliding window of time-temperature tuples, then when a reading of t arrives, we can delete immediately any earlier reading that is smaller than t.

! a) Does this rule always compress the data in the window?

!! b) Suppose temperatures are real numbers chosen uniformly and at random from some fixed range of values. On average, how many tuples will be retained, as a function of N?

23.6 Summary of Chapter 23

✦ *Search Engines*: A search engine requires a crawler to gather information about pages and a query engine to answer search queries.

✦ *Crawlers*: A crawler consists of one or more processes that visit Web pages and follow links found in those pages. The crawler must maintain a repository of pages already visited, so it does not revisit the same page too frequently. Shingling and minhashing can be used to detect duplicate pages with different URL's.

✦ *Limiting the Crawl*: Crawlers normally limit the depth to which they will search, declining to follow links from pages that are too far from their root page or pages. They also can prioritize the search to visit preferentially pages that are estimated to be popular.

✦ *Preparing Crawled Pages to Be Searched*: The search engine creates an inverted index on the words of the crawled pages. The index may also include information about the role of the word (e.g., is it part of a header?), and the index for each word may be represented by a bit-vector indicating on which pages the word appears.

- *Answering Search Queries*: A search query normally consists of a set of words. The query engine uses the inverted index to find the Web pages containing all these words. The pages are then ranked, using a formula that is determined by each search engine, but typically favors pages with close occurrences of the words, use of the words in important places (e.g., headers), and favors important pages using a measure such as PageRank.

- *The Transition Matrix of the Web*: This matrix is an important analytic tool for estimating the importance of Web pages. There is a row and column for each page, and the column for page j has $1/r$ in the ith row if page i is one of r pages with links from page j, and 0 otherwise.

- *PageRank*: The PageRank of Web pages is the principal eigenvector of the transition matrix of the Web. If there are n pages, we can compute the PageRank vector by starting with a vector of length n, and repeatedly multiplying the current vector by the transition matrix of the Web.

- *Taxation of PageRank*: Because of Web artifacts such as dead ends (pages without out-links) and spider traps (sections of the Web that cannot be exited), it is normal to introduce a small tax, say 15%, and redistribute that fraction of a page's PageRank equally among all pages, after each matrix-vector multiplication.

- *Teleport Sets*: Instead of redistributing the tax equally among all pages during an iteration of the PageRank computation, we can distribute the tax only among a subset of the pages, called the teleport set. Then, the computation of PageRank simulates a walker on the graph of the Web who normally follows a randomly chosen out-link from their current page, but with a small probability instead jumps to a random member of the teleport set.

- *Topic-Specific PageRank*: One application of the teleport-set idea is to pick a teleport set consisting of a set of pages known to be about a certain topic. Then, the PageRank will measure not only the importance of the page in general, but to what extent it is relevant to the selected topic.

- *Link Spam*: Spam farmers create large collections of Web pages whose sole purpose is to increase the PageRank of certain target pages, and thus make them more likely to be displayed by a search engine. One way to combat such spam farms is to compute PageRank using a teleport set consisting of known, trusted pages — those that are unlikely to be spam.

- *Data Streams*: A data stream is a sequence of tuples arriving at a fixed place, typically at a rate so fast as to make processing and storage in its entirety difficult. Examples include streams of data from satellites and click streams of requests at a Web site.

- *Data-Stream-Management Systems*: A DSMS accepts data in the form of streams. It maintains working storage and permanent (archival) storage. Working storage is limited, although it may involve disks. The DSMS accepts both ad-hoc and standing queries about the streams.

- *Sliding Windows*: To query a stream, it helps to be able to talk about portions of the stream as a relation. A sliding window is the most recent portion of the stream. A window can be time-based, in which case it consists of all tuples arriving over some fixed time interval, or tuple-based, in which case it is a fixed number of the most recently arrived tuples.

- *Compressing Windows*: If the DSMS must maintain large windows on many streams, it can run out of main memory, or even disk space. Depending on the family of queries that will be asked about the window, it may be possible to compress the window so it uses significantly less space. However, in many cases, we can compress a window only if we are willing to accept approximate answers to queries.

- *Counting Bits*: A fundamental problem that allows a space/accuracy trade-off is that of counting the number of 1's in a window of a bit-stream. We partition the window into buckets representing exponentially increasing numbers of 1's. The last bucket may be partially outside the window, leading to inaccuracy in the count of 1's, but the error is limited to a fixed fraction of the count and can be any $\epsilon > 0$.

- *Counting Distinct Elements*: Another important stream problem is counting the number of distinct elements in the stream without keeping a table of all the distinct elements ever seen. An unbiased estimate of this number can be made by picking a hash function, hashing elements to bit strings, and estimating the number of distinct elements to be 2 raised to the power that is the largest number of consecutive 0's ever seen at the end of the hash function of any stream element.

23.7 References for Chapter 23

References [3] and [8] summarize issues in crawling, based on the Stanford WebBase system. An analysis of the degree to which crawlers reach the entire Web was given in [15].

PageRank and the Google search engine are described in [6] and [16]. An alternative formulation of Web structure, often referred to as "hubs and authorities," is in [14].

Topic-specific PageRank, as described here, is from [12]. TrustRank and combating link spam are discussed in [11].

Two on-line histories of search engines are [17] and [18].

The study of data streams as a data model can be said to begin with the "chronicle data model" of [13]. References [7] and [2] describe the architecture of early data-stream management systems. Reference [5] surveys data-stream systems.

The algorithm described here for approximate counting of 1's in a sliding window is from [9].

The problem of estimating the number of distinct elements in a stream originated with [10] and [4]. The method described here is from [1], which also generalizes the technique to estimate higher moments of the data, e.g., the sum of the squares of the number of occurrences of each element.

1. N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating frequency moments," *Twenty-Eighth ACM Symp. on Theory of Computing* (1996), pp. 20–29.

2. A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution,"

   `http://dbpubs.stanford.edu/pub/2003-67`

   Dept. of Computer Science, Stanford Univ., Stanford CA, 2003.

3. A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan, "Searching the Web," *ACM Trans. on Internet Technologies* **1**:1 (2001), pp. 2–43.

4. M. M. Astrahan, M. Schkolnick, and K.-Y. Whang, "Approximating the number of unique values of an attribute without sorting," *Information Systems* **12**:1 (1987), pp. 11-15.

5. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," *Twenty-First ACM Symp. on Principles of Database Systems* (2002), pp. 261–272.

6. S. Brin and L. Page, "Anatomy of a large-scale hypertextual Web search engine," *Proc. Seventh Intl. World-Wide Web Conference*, 1998.

7. D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams — a new class of data management applications," *Proc. Intl. Conf. on Very Large Database Systems* (2002), pp. 215–226.

8. J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley, "Stanford WebBase components and applications," *ACM Trans. on Internet Technologies* **6**:2 (2006), pp. 153–186.

9. M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM J. Computing* **31** (2002), pp. 1794–1813.

10. P. Flagolet and G. N. Martin, "Probabilistic counting for database applications," *J. Computer and System Sciences* **31**:2 (1985), pp. 182–209.

11. Z. Gyongyi, H. Garcia-Molina, and J. Pedersen, "Combating Web spam with TrustRank," *Proc. Intl. Conf. on Very Large Database Systems* (2004), pp. 576–587.

12. T. Haveliwala, "Topic-sensitive PageRank," *Proc. Eleventh Intl. World-Wide Web Conference* (2002).

13. H. V. Jagadish, I. S. Mumick, and A Silberschatz, "View maintenance issues for the chronicle data model," *Fourteenth ACM Symp. on Principles of Database Systems* (1995), pp. 113–124.

14. J. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM* **46**:5 (1999), pp. 604–632.

15. S. Lawrence and C. L. Giles, "Searching the World-Wide Web," *Science* **280**(5360):98, 1998.

16. L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the Web," unpublished manuscript, Dept. of CS, Stanford Univ., Stanford CA, 1998.

17. L. Underwood, "A brief history of search engines,"
`www.webreference.com/authoring/search_history`

18. A. Wall, "Search engine history," `www.searchenginehistory.com`.

Index