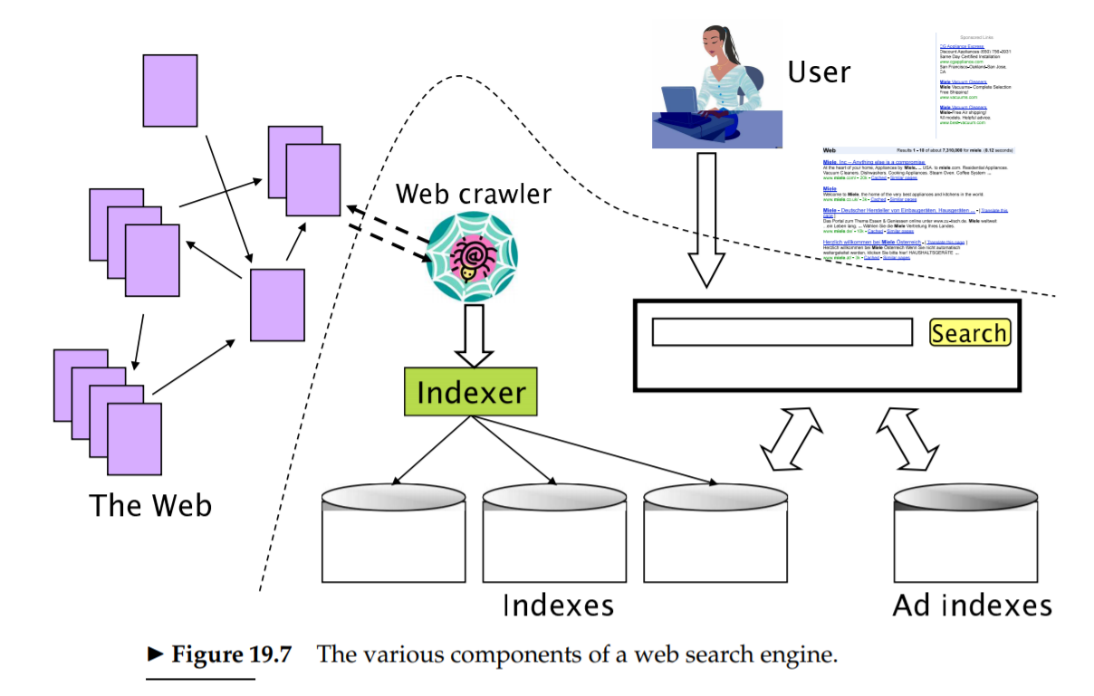
웹 크롤링 및 인덱스

WEB CRAWLER SPIDER

  
20.1 개요  
웹 크롤링은 웹에서 페이지를 수집하여 색인화하고 검색 엔진을 지원하는 과정입니다. 크롤링의 목적은 유용한 웹 페이지를 상호 연결하는 링크 구조와 함께 가능한 한 빠르고 효율적으로 수집하는 것이다. 19장에서 우리는 조정되지 않은 수백만 명의 사람들에 의한 웹의 생성에서 비롯된 웹의 복잡성에 대해 연구했다. 이 장에서는 웹을 기어다니는 데 따른 어려움을 연구한다. 이 장의 초점은 그림 19.7에서 웹 크롤러로 표시된 구성 요소이며, 때때로 거미라고도 합니다.  
이 장의 목표는 전면적인 상용 웹 검색 엔진을 위한 크롤러를 구축하는 방법을 설명하는 것이 아닙니다. 대신 우리는 학생 프로젝트 규모에서 상당한 연구 프로젝트로 이동하는 일반적인 범위의 문제에 초점을 맞춘다. 우리는 웹 크롤러에 대한 결정 데이터를 나열함으로써 (제20.1.1절) 시작한 다음, 이러한 각 문제가 어떻게 해결되는지 20.2절에 논의한다. 이 장의 나머지 부분에서는 이러한 특징들을 만족시키는 분산형 웹 크롤러의 아키텍처와 일부 구현 세부 사항을 설명한다. 20.3절에서는 웹 스케일 구현을 위해 많은 시스템에 인덱스를 배포하는 방법에 대해 설명합니다.

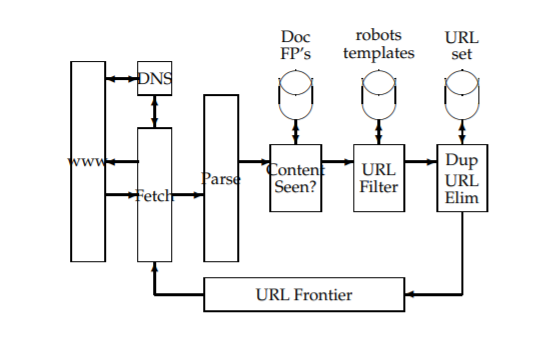
20.1.1 크롤러가 *반드시* 제공해야 하는 특징

우리는 웹 크롤러가 제공해야 하는 특징과 그 다음에 제공해야 하는 특징의 두 가지 범주로 웹 크롤러를 위한 이상적인 데이터를 나열한다.

견고성: 웹은 스파이더 트랩을 만드는 서버를 포함하고 있는데, 이것은 특정 도메인에서 무한히 많은 페이지를 가져오게 하는 크롤러를 오도하는 웹 페이지 생성기입니다. 크롤러는 이러한 함정에 탄력적으로 반응하도록 설계되어야 한다. 그러한 모든 함정이 악의적인 것은 아니다; 어떤 함정은 잘못된 웹사이트 개발의 의도치 않은 부작용이다.  
예의: 웹 서버에는 크롤러가 방문할 수 있는 속도를 규제하는 암시적 정책과 명시적 정책이 모두 있습니다. 이러한 예의 정책은 존중되어야 한다.  
  
20.1.2 크롤러가 제공해야 하는 특징  
분산: 크롤러는 여러 시스템에 분산 방식으로 실행할 수 있는 기능을 가지고 있어야 합니다.  
확장 가능: 크롤러 아키텍처에서는 추가 시스템과 대역폭을 추가하여 크롤러 속도를 확장할 수 있어야 합니다.  
성능 및 효율성: 크롤 시스템은 프로세서, 스토리지 및 네트워크 대역폭을 포함한 다양한 시스템 리소스를 효율적으로 사용해야 합니다.  
품질: 모든 웹 페이지의 상당 부분이 사용자 쿼리 요구를 처리하는 데 유용하지 않다는 점을 고려할 때, 크롤러는 "유용한" 페이지를 먼저 가져오는 쪽으로 치우쳐 있어야 합니다.  
신선도: 대부분의 응용 프로그램에서 크롤러는 연속 모드로 작동해야 합니다. 이전에 가져온 페이지의 새 복사본을 얻어야 합니다. 예를 들어, 검색 엔진 크롤러는 검색 엔진의 인덱스가 각 인덱스 웹 페이지의 최신 표현을 포함하는지 확인할 수 있습니다. 이러한 연속 크롤링의 경우, 크롤러는 해당 페이지의 변경 속도에 가까운 빈도를 사용하여 페이지를 크롤링할 수 있어야 합니다.  
확장 가능: 크롤러는 새로운 데이터 형식, 새로운 가져오기 프로토콜 등에 대처하기 위해 여러 가지 방법으로 확장 가능하도록 설계되어야 합니다. 따라서 크롤러 아키텍처는 모듈식이어야 합니다.

20.2 크롤링 모든 하이퍼텍스트 크롤러(웹, 인트라넷 또는 기타 하이퍼텍스트 문서 모음의 경우)의 기본 작업은 다음과 같습니다. 크롤러는 시드 세트를 구성하는 하나 이상의 URL로 시작합니다. 이 시드 세트에서 URL을 선택한 다음 해당 URL로 웹 페이지를 가져옵니다. 가져온 페이지는 구문 분석되어 페이지에서 텍스트와 링크를 모두 추출합니다(각 페이지는 다른 URL을 가리킵니다). 추출된 텍스트는 텍스트 인덱서로 제공됩니다(제4장 및 제5장에서 설명). 추출된 링크(URL)는 URL 프론티어에 추가되며, 항상 해당 페이지를 크롤러가 가져오지 않은 URL로 구성됩니다. 처음에 URL 프런티어에는 시드 세트가 포함되어 있습니다. 페이지를 가져오면 해당 URL이 URL 프런티어에서 삭제됩니다. 전체 프로세스는 웹 그래프를 통과하는 것으로 볼 수 있습니다(19장 참조). 연속 크롤링에서 가져온 페이지의 URL은 나중에 다시 가져오기 위해 다시 프론티어에 추가됩니다.  
웹 그래프의 단순해 보이는 재귀 통과는 실용적인 웹 크롤링 시스템에 대한 많은 요구로 인해 복잡하다. 크롤러는 높은 품질의 페이지를 가져오는 동안 배포되고, 확장 가능하며, 효율적이며, 공손하고, 확장 가능해야 한다. 우리는 이 이슈들의 영향을 조사한다. 우리의 MERCATOR 치료는 많은 연구와 상업적 크롤러의 기초를 형성한 Mercator 크롤러의 설계를 따른다. 참조 포인트로, 한 달 동안 탐색하는 데 10억 페이지(현재 정적 웹의 일부)를 가져오려면 초당 수백 페이지를 가져와야 합니다. 우리는 이 인출 속도를 달성하기 위해 멀티 스레드 설계를 사용하여 전체 크롤러 시스템의 몇 가지 병목 현상을 해결하는 방법을 볼 것이다.  
이 자세한 설명을 진행하기 전에, 우리는 비전문가 크롤러가 만족해야 할 몇 가지 기본 특성을 가진 크롤러를 구축하려고 시도할 수 있는 독자를 위해 다시 한 번 강조합니다.  
1. 지정된 호스트에는 한 번에 하나의 연결만 열려 있어야 합니다.  
2. 호스트에 대한 연속적인 요청 사이에 몇 초의 대기 시간이 발생해야 합니다.  
3. 제20.2.1절에 규정된 공손성 제한을 준수해야 한다.

20.2.1 Crawler 아키텍처  
위에서 설명한 크롤링에 대한 간단한 계획은 그림 20.1과 같이 서로 맞는 몇 개의 모듈을 요구한다.  
1. 현재 크롤에서 가져올 URL이 아직 포함되어 있는 URL 프론티어입니다(연속 크롤링의 경우 URL이 이전에 가져온 것일 수 있지만 다시 가져오기 위해 다시 프런티어에 있습니다). 우리는 이것을 섹션 20.2.3에서 더 자세히 설명한다.  
2. URL에서 지정한 페이지를 가져올 웹 서버를 결정하는 DNS 확인 모듈. 20.2.2절에 자세히 설명되어 있습니다.  
3. http 프로토콜을 사용하여 URL에서 웹 페이지를 검색하는 가져오기 모듈.  
4. 가져온 웹 페이지에서 텍스트 및 링크 집합을 추출하는 구문 분석 모듈.  
5. 추출된 링크가 URL 프론티어에 이미 있는지 또는 최근에 가져온 것인지 여부를 결정하는 중복 제거 모듈입니다.



◮ Figure 20.1 The basic crawler architecture.

크롤링은 하나에서 잠재적으로 수백 개의 스레드에 의해 수행되며, 각 스레드는 그림 20.1의 논리 주기를 통해 반복됩니다. 이러한 스레드는 단일 프로세스에서 실행되거나 분산 시스템의 서로 다른 노드에서 실행되는 여러 프로세스 간에 분할될 수 있습니다. 우리는 URL 프론티어가 제자리에 있고 비어 있지 않다고 가정하는 것으로 시작하고 URL 프론티어의 구현에 대한 설명을 섹션 20.2.3으로 연기한다. 우리는 단일 URL이 가져오는 주기를 통해 진행되며, 다양한 검사와 필터를 통과한 다음, 마지막으로(연속 크롤링의 경우) URL 프런티어로 반환된다.  
크롤러 스레드는 일반적으로 http 프로토콜을 사용하여 프런티어로부터 URL을 가져와서 해당 URL에 웹 페이지를 가져오는 것으로 시작한다. 그런 다음 가져온 페이지는 임시 저장소에 기록되며, 여기서 여러 작업이 수행됩니다. 그런 다음 페이지가 구문 분석되고 텍스트와 텍스트의 링크가 추출됩니다. 텍스트(예: 굵은 글꼴의 용어)는 인덱서로 전달됩니다. 앵커 텍스트를 포함한 링크 정보도 인덱서로 전달되어 21장에서 설명하는 방식으로 순위를 매기는 데 사용됩니다. 또한 추출된 각 링크는 일련의 테스트를 거쳐 링크를 URL 프론티어에 추가해야 하는지 여부를 결정합니다.  
첫째, 스레드는 동일한 내용을 가진 웹 페이지가 다른 URL에서 이미 표시되는지 여부를 테스트합니다. 이를 위한 가장 간단한 구현은 체크섬과 같은 간단한 지문을 사용할 것이다(그림 20.1의 "Doc FP"s라는 레이블이 붙은 저장소에 배치). 보다 정교한 시험은 19장에서 설명한 것처럼 지문 대신 슁글을 사용할 수 있다.  
다음으로, URL 필터는 추출된 URL을 여러 테스트 중 하나를 기반으로 프론티어에서 제외해야 하는지 여부를 결정하는 데 사용됩니다. 예를 들어, 크롤은 특정 도메인(예: 모든 .com URL)을 제외하려고 할 수 있습니다. 이 경우 테스트는 단순히 .com 도메인에서 URL을 필터링합니다. 유사한 시험은 배타적이기보다는 포괄적일 수 있다. 웹 상의 많은 호스트들은 웹 사이트의 특정 부분을 로봇 제외 프로토콜로 알려진 로봇 제외 표준 하에 기어가기 금지된다. 이 작업은 로봇이라는 이름으로 프로토콜 파일을 배치하여 수행됩니다. 사이트의 URL 계층 루트에 있는 txt입니다. 여기 로봇 예가 있습니다."seargine"이라는 로봇을 제외하고 파일 계층 구조에서 위치가 /yoursite/temp/로 시작하는 URL을 로봇이 방문하지 않도록 지정하는 txt 파일입니다.

19장 참고

SHINGLING

슁글링은 중복 문서 처리 기법 중에 하나이다. 문서를 정해진 윈도우 사이즈의 슁글로 표현하고, 이것을 이용해 문서의 변경도를 측정한다. 슁글(shingle)은 연속된 k개의 단어 시퀀스를 의미한다. 이러한 슁글의 집합을 k-shingling이라고 하며, 이것을 이용하여 문서들의 중복률을 찾아낸다. 슁글의 크기가 크다면 연산비용은 줄어들지만, 정확도는 떨어진다.

ex) “a rose is a rose is a rose.“ The 4-shingles for this text (k = 4 is a typical value used in the detection of near-duplicate web pages) are “a rose is a”, “rose is a rose” and “is a rose is”.

User-agent: \*

Disallow: /yoursite/temp/

User-agent: searchengine

Disallow:

고려 중인 URL이 로봇 제한을 통과하는지 테스트하기 위해 웹 사이트에서 로봇 텍스트 파일을 가져와야 하므로 URL 프론티어에 추가할 수 있습니다. 프론티어에 추가할 각 URL에서 테스트를 위해 새로 가져오는 대신, 캐시는 호스트를 위해 최근에 가져온 파일의 복사본을 얻는 데 사용될 수 있습니다. 페이지로부터 추출된 많은 링크가 페이지를 가져온 호스트 내에 있으므로, 호스트의 로봇 텍스트 파일에 대해 테스트할 수 있기 때문에 특히 중요합니다. 따라서 링크 추출 프로세스 중에 필터링을 수행하여 테스트해야 하는 호스트 스트림에서 특히 높은 인접성을 가질 수 있습니다.  
캐시 적중률이 높아지므로 로봇 텍스트 파일의 경우 캐시 적중률이 높습니다. 불행하게도, 이것은 웹마스터들의 공손함에 대한 기대와 어긋난다. URL(특히 품질이 낮거나 거의 변경되지 않는 문서를 가리키는 URL)은 며칠 또는 몇 주 동안 프론티어에 있을 수 있습니다. 만약 우리가 그러한 URL을 프론티어에 추가하기 전에 로봇 필터링을 수행한다면, 그것의 로봇 텍스트 파일은 URL이 프론티어에서 큐로 대기하고 가져올 때쯤 바뀔 수 있었을 것이다. 따라서 웹 페이지를 가져오기 전에 즉시 로봇 필터링을 수행해야 합니다.  
밝혀진 바와 같이, 로봇 텍스트 파일의 캐시를 유지하는 것은 여전히 매우 효과적입니다.  
URL 프런티어로부터 대기중인 URL의 스트림에도 충분한 지역성이 있습니다.

URL NORMALIZATION

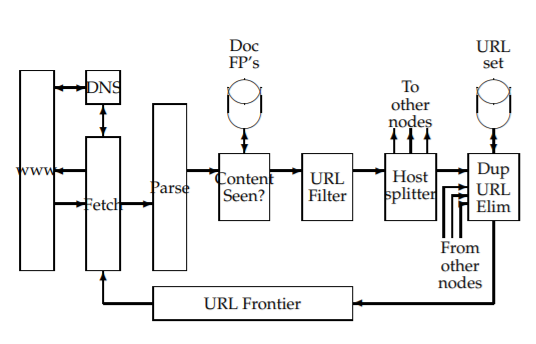
다음으로, URL은 다음과 같은 의미에서 정규화되어야 한다: 종종 웹 페이지 p에서 링크의 HTML 인코딩은 페이지 p에 상대적인 링크의 대상을 나타낸다. 따라서, 페이지의 HTML에 이렇게 인코딩된 상대 링크가 있다. en.wikipedia.org/wiki/Main\_Page:

<a href=”/wiki/Widipedia:General\_disclaimer” title=”Widipedia:General disclaimer”> Disclaimers </a>

points to the URL <http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer>.

마지막으로 URL이 중복 제거되었는지 확인합니다. URL이 이미 프론티어에 있거나(비연속 크롤인 경우) 이미 탐색된 경우에는 프론티어에 추가하지 않습니다. URL이 프론티어에 추가될 때, URL에 우선 순위가 할당되며, 이를 기준으로 URL이 결국 프런티어로부터 제거되어 페치됩니다. 이 우선 순위 큐의 세부 정보는 섹션 20.2.3에 나와 있습니다.  
특정 하우스키핑 작업은 일반적으로 전용 스레드에 의해 수행됩니다. 이 스레드는 크롤 진행률 통계(URL 크롤링, 프런티어 크기 등)를 기록하기 위해 몇 초마다 한 번씩 깨어난다는 점, 크롤링 종료 여부를 결정하거나 크롤링 체크포인트(크롤링 몇 시간마다 한 번씩)를 확인하는 점 등을 제외하고는 일반적으로 대기 상태입니다. 체크포인트에서 크롤러 상태(예: URL 프런티어)의 스냅샷이 디스크에 커밋됩니다. 치명적인 크롤러 오류가 발생하면 가장 최근의 체크포인트에서 크롤러가 다시 시작됩니다.

크롤러 분산  
우리는 크롤러의 스레드가 분산 크롤러 시스템의 서로 다른 노드에서 각각 다른 프로세스에서 실행될 수 있다고 언급했다. 이러한 분포는 확장에 필수적이며, 각 노드가 호스트를 "근접" 탐색하는 지리적으로 분산된 크롤러 시스템에서도 사용할 수 있다. 크롤러 노드 간에 탐색 중인 호스트의 파티셔닝은 해시 함수 또는 좀 더 구체적으로 조정된 정책에 의해 수행될 수 있다. 예를 들어, 패킷이 인터넷을 통해 이동하는 경로가 항상 지리적 근접성을 반영하지는 않으며, 어떤 경우에도 호스트의 도메인이 항상 물리적 위치를 반영하지 않지만, 유럽 도메인에 집중하기 위해 유럽에서 크롤러 노드를 찾을 수 있다.  
분산형 크롤러의 다양한 노드는 어떻게 URL을 통신하고 공유합니까? 이 아이디어는 각 노드에서 그림 20.1의 흐름을 복제하는 것인데, 한 가지 본질적인 차이점이 있다: URL 필터 다음에, 우리는 호스트 스플리터를 사용하여 URL을 담당하는 크롤러 노드에 각 생존 URL을 디스패치한다. 따라서 탐색 중인 호스트 집합은 노드 간에 분할된다. 이 수정된 흐름은 그림 20.2에 나와 있습니다. 호스트 스플리터의 출력은 분산 시스템에서 서로 다른 노드의 중복 URL 제거기 블록으로 들어갑니다.  
그러나 그림 20.2의 분산형 아키텍처에서 "콘텐츠 가시화?" 모듈은 다음과 같은 몇 가지 요소로 인해 복잡하다.  
1. URL 프런티어 및 중복 제거 모듈과 달리 문서 지문/슁글은 호스트 이름을 기준으로 분할할 수 없습니다. 동일한 콘텐츠(또는 매우 유사한 콘텐츠)가 서로 다른 웹 서버에 나타나지 않도록 하는 것은 없습니다. 따라서 지문/슁글 집합은 지문/슁글의 일부 속성을 기반으로 노드 간에 분할되어야 한다(예: 노드 수에 지문 모듈을 가져감). 이 지역성 불일치의 결과는 대부분의 "Content Seen?" 테스트가 원격 프로시저 호출을 초래한다는 것이다(그러나 배치 룩업 요청을 일괄 처리할 수 있다.)



◮ Figure 20.2 Distributing the basic crawl architecture.

2. 문서 지문/싱글의 스트림에 지역성이 거의 없습니다. 따라서, 인기 있는 지문을 캐싱하는 것은 도움이 되지 않습니다(인기 있는 지문이 없기 때문이다.  
3. 문서는 시간이 지남에 따라 변경되므로 계속 탐색하는 상황에서 오래된 지문/싱글을 컨텐츠에서 삭제할 수 있어야 합니다. 그러기 위해서는 URL 자체와 함께 문서의 지문/슁글을 URL 프론티어에 저장해야 합니다.

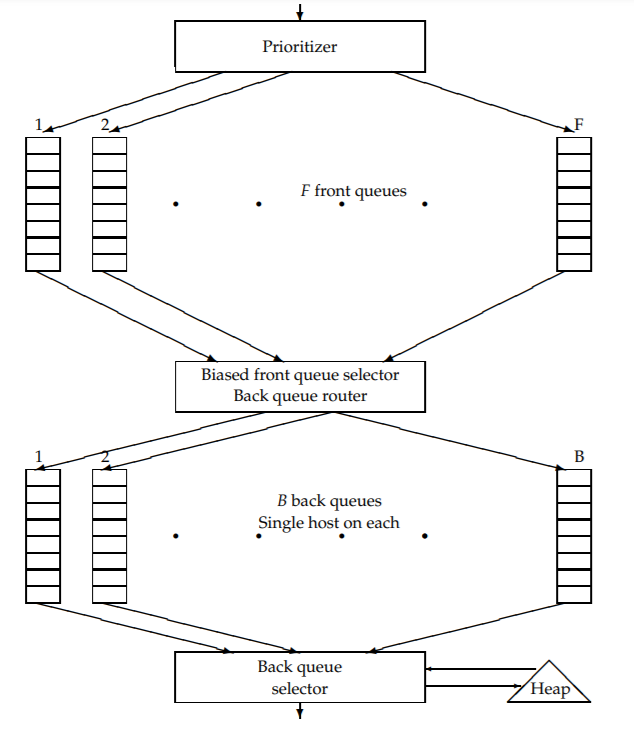
20.2.2 DNS Resolution (DNS주소를 IP주소로 변환하는 Recursive한 과정)  
IP 주소, DNS 확인, DNS 서버

각 웹 서버(그리고 실제로 인터넷에 연결된 호스트)는 고유한 IP 주소를 가지고 있다. 4바이트의 시퀀스는 일반적으로 점으로 분리된 4개의 정수로 표시된다. 예를 들어 207.142.131.248은 호스트 www.wikipedia.org 와 관련된 숫자 IP 주소이다. 텍스트 형식의 www.wikipedia.org 와 같은 URL이 있을 경우, 이를 IP 주소로 변환하는 프로세스(이 경우 207.142.131.248)는 DNS 확인 또는 DNS 조회라고 합니다. 여기서 DNS는 도메인 이름 서비스를 나타냅니다. DNS를 확인하는 동안 이 변환을 수행하려는 프로그램(우리의 경우 웹 크롤러의 구성 요소)은 변환된 IP 주소를 반환하는 DNS 서버에 접속합니다. (실제로 전체 변환은 단일 DNS 서버에서 수행되지 않을 수 있으며, 처음에 연결된 DNS 서버가 번역을 완료하기 위해 다른 DNS 서버를 재귀적으로 호출할 수 있습니다.). en.wikipedia.org/wiki/Domain\_Name\_System 와 같은 보다 복잡한 URL의 경우 DNS 확인을 담당하는 크롤러 구성 요소가 호스트 이름(이 경우 en.wikipedia.org)을 추출하고 호스트 en.wikipedia.org 의 IP 주소를 조회합니다.

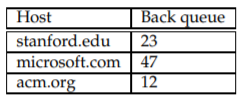
DNS 해결은 웹 탐색에서 잘 알려진 병목 현상입니다. 도메인 이름 서비스의 분산 특성으로 인해 DNS 확인에는 인터넷을 통한 여러 요청과 왕복이 수반될 수 있으며, 몇 초 또는 더 긴 시간이 필요할 수도 있습니다. 당장, 이것은 초당 수백 개의 문서를 가져오려는 우리의 목표를 위태롭게 합니다. 표준 해결 방법은 캐싱 방식을 도입하는 것입니다. 최근에 DNS 검색을 수행한 URL은 DNS 캐시에서 발견될 수 있으므로 인터넷의 DNS 서버로 이동할 필요가 없습니다. 그러나 공손도 제약(섹션 20.2.3 참조)을 준수하면 캐시 적중률이 제한된다.

DNS 해결에는 또 다른 중요한 어려움이 있다. 표준 라이브러리(크롤러를 개발하는 사람이 사용할 가능성이 있음)의 룩업 구현은 일반적으로 동기적이다. 즉, 도메인 이름 서비스에 대한 요청이 이루어지면 첫 번째 요청이 완료될 때까지 해당 노드의 다른 크롤러 스레드가 차단됩니다. 이를 피하기 위해 대부분의 웹 크롤러는 자체 DNS 확인기를 크롤러의 구성 요소로 구현합니다. 해결사 코드를 실행하는 스레드 i는 DNS 서버에 메시지를 보낸 다음 다른 스레드에 의해 신호가 수신되거나 설정된 시간 퀀텀이 만료될 때 다시 시작됩니다. 단일 별도의 DNS 스레드는 이름 서비스에서 들어오는 응답 패킷을 표준 DNS 포트(포트 53)에서 수신합니다. 응답을 수신하면 적절한 크롤러 스레드(이 경우 i)에 신호를 보내고 시간 퀀텀이 만료되어 아직 재개되지 않은 경우 응답 패킷을 전달합니다. 대기 시간 퀀텀이 일정한 횟수의 시도 동안 재시도 횟수가 만료되어 다시 시작되는 크롤러 스레드. 새 메시지를 DNS 서버로 전송하고 매번 시간별 대기 작업을 수행합니다. Mercator의 설계자는 5번의 시도 순서를 권장합니다. 대기 시간의 퀀텀은 이러한 시도마다 기하급수적으로 증가하는데, 메르카토르(Mercator)는 해결하는데 수십초가 걸리는 호스트 이름이 있다는 사실을 고려하여 1초에서 시작하여 약 90초로 종료되었다.

20.2.3 URL 프런티어  
노드의 URL 프런티어는 해당 크롤 프로세스(또는 다른 크롤 프로세스의 호스트 스플리터)에 의해 URL이 지정됩니다. 이것은 프론티어에 URL을 유지하며 크롤러 스레드가 URL을 찾을 때마다 순서대로 URL을 재등록합니다. 두 가지 중요한 고려사항은 변경사항이 URL을 반환하는 순서를 결정합니다. 첫째, 자주 바뀌는 고품질 페이지는 기어가 잦은 페이지보다 우선시되어야 한다. 따라서 페이지의 우선 순위는 변경률과 품질(일부 합리적인 품질 추정치 사용)의 함수여야 한다. 이 조합은 가져올 때마다 많은 수의 스팸 페이지가 완전히 변경되기 때문에 필요합니다.  
두 번째 고려사항은 예의입니다. 우리는 짧은 시간 내에 호스트에 대한 반복적인 가져오기 요청을 피해야 합니다. 많은 URL이 동일한 호스트의 다른 URL로 연결되는 참조 지역성 때문에 이러한 가능성이 악화됩니다. 따라서 단순 우선 순위 대기열로 구현된 URL 프론티어는 호스트에 대한 가져오기 요청을 버스트할 수 있습니다. 이 문제는 크롤러를 제한하여 언제든지 단일 호스트에서 하나의 스레드를 가져올 수 있도록 하는 경우에도 발생할 수 있습니다. 일반적인 휴리스틱은 연속적인 가져오기 요청 사이의 간격을 해당 호스트에서 가장 최근에 가져오는 데 걸린 시간보다 큰 순서로 삽입하는 것이다.  
그림 20.3은 URL 프론티어의 정중하고 우선 순위를 정하는 구현을 보여줍니다. 이것의 목표는 (i) 한 번에 하나의 호스트에만 연결되도록 하는 것이다. (ii) 호스트[[1]](#footnote-1)에 대한 연속적인 요청과 (iii) 높은 우선순위 페이지 사이에 몇 초의 대기 시간이 발생하는 것이다.  
두 개의 주요 하위 모듈은 그림의 위쪽에 있는 F 전면 대기열 세트와 아래쪽에 있는 B 후면 대기열 세트입니다. 이 모든 것이 FIFO 대기열입니다. 앞 큐는 우선 순위를 구현하는 반면, 뒷 큐는 공손함을 구현합니다. 프런티어에 추가된 URL의 흐름에서, 우선 순위는 먼저 URL의 가져오기 기록에 기초하여 1에서 F 사이의 정수 우선 순위 i를 URL에 할당한다(이 URL의 웹 페이지가 이전 크롤 간에 변경된 속도를 고려). 예를 들어, 빈번한 변경을 보이는 문서에는 더 높은 우선순위가 할당됩니다. 예를 들어 뉴스 서비스의 URL은 항상 가장 높은 우선순위로 할당될 수 있으므로 다른 휴리스틱은 애플리케이션에 의존적이며 명시적일 수 있다. 우선 순위가 i로 지정되었으므로 이제 URL이 전면 대기열의 i에 추가됩니다.  
각 B 백 큐는 (i) 크롤이 진행되는 동안에는 비어 있지 않으며 (ii) 단일 호스트의 URL만 포함합니다. 보조 테이블 T(그림 20.4)는 호스트에서 백 큐로의 매핑을 유지하는 데 사용됩니다. 백 큐가 비어 있고 전면 큐에서 다시 채워질 때마다 테이블 T를 그에 따라 업데이트해야 합니다.



◮ Figure 20.3 The URL frontier. 이미 탐색한 페이지에서 추출한 URL이 그림의 맨 위에 표시됩니다. URL을 요청하는 크롤 스레드는 그림 아래에서 해당 스레드를 추출합니다. 경로상 URL은 크롤링 우선 순위를 관리하는 여러 프론트 대기열 중 하나를 통해 흐르고 크롤러의 공손함을 관리하는 여러 백 대기열 중 하나가 표시됩니다.



◮ Figure 20.4 Example of an auxiliary hosts-to-back queues table.

또한 각 백 큐에 대해 하나의 항목이 있는 힙을 유지하며, 이 항목은 해당 큐에 해당하는 호스트에 다시 연결할 수 있는 가장 빠른 시간 te가 됩니다.  
프론티어로부터 URL을 요청하는 크롤러 스레드는 이 힙의 루트를 추출하고 (필요한 경우) 해당 시간 엔트리가 될 때까지 기다립니다. 그런 다음 추출된 힙 루트에 해당하는 백 큐 j의 머리 부분에 URL u를 가져옵니다. u를 가져온 후에 호출 스레드는 j가 비어 있는지 확인합니다. 만일 그렇다면, 그것은 전면 큐를 선택하고 그것의 머리에서 URL v를 추출한다. 전면 큐의 선택은 (일반적으로 임의의 프로세스에 의해) 더 높은 우선 순위의 큐로 편향되어, 높은 우선 순위의 URL이 백 큐로 더 빨리 흐르도록 한다. v를 검사하여 해당 호스트에서 URL을 포함하는 백큐가 이미 있는지 확인합니다. 그렇다면 v가 해당 대기열에 추가되고 이제 비어 있는 대기열 j에 삽입할 다른 후보 URL을 찾기 위해 전면 대기열에 다시 도달합니다. 이 프로세스는 j가 다시 비어 있지 않을 때까지 계속됩니다. 어떤 경우든 스레드는 마지막으로 가져온 URL in j의 속성(예: 호스트가 마지막으로 연결된 시간 및 마지막 가져오기에 걸린 시간)을 기반으로 j에 대한 힙 항목을 삽입한 후 처리를 계속한다. 예를 들어, 새 입력 항목은 현재 시간과 마지막 가져오기 시간의 10배가 될 수 있습니다.  
우선 순위를 할당하고 큐를 선택하는 정책과 함께 전면 대기열 수에 따라 시스템에 빌드할 우선 순위 속성이 결정됩니다. 백 큐의 수는 공손함을 존중하면서 모든 크롤 스레드를 계속 사용할 수 있는 범위를 결정합니다. Mercator의 설계자들은 크롤러 스레드보다 3배 많은 백 큐의 대략적인 규칙을 권고한다.  
웹 스케일 크롤에서 URL 프론티어는 사용 가능한 노드보다 더 많은 메모리를 요구할 정도로 커질 수 있습니다. 해결책은 대부분의 URL 프론티어를 디스크에 저장하도록 하는 것입니다. 각 대기열의 일부는 메모리에 보관되며, 디스크에서 더 많이 가져와 메모리에서 배출됩니다.

연습 20.1 분산 크롤링 시스템의 노드 간에 호스트(개별 URL보다)를 파티셔닝하는 것이 더 나은 이유는 무엇입니까?  
연습 20.2 호스트 스플리터가 중복 URL 제거기 앞에 와야 하는 이유는 무엇입니까?  
연습 20.3   
앞의 토론에서 우리는 두 가지 권장되는 "하드 상수"를 발견했다. 즉, 마지막 가져오기 시간의 10배인 증가분 1과 크롤 스레드 수의 3배인 백 큐의 수가 그것이다. 이 두 상수는 어떤 관계가 있습니까?

TERM PARTITIONING, DOCUMENT PARTITIONING

20.3 인덱스 배포  
섹션 4.4에서 분산 인덱싱을 설명했습니다. 이제 쿼리를 지원하는 대형 컴퓨터 클러스터[[2]](#footnote-2)에 대한 인덱스의 분포를 고려한다. 두 가지 분명한 대체 인덱스 구현은 용어별 분할, 글로벌 인덱스 구성이라고도 하는 문서와 로컬 인덱스 조직이라고도 하는 문서별 분할을 제안합니다. 전자에서 색인 용어 사전은 각 부분 집합이 노드에 상주하는 부분 집합으로 분할된다. 노드의 용어와 함께 해당 용어에 대한 게시물을 보관합니다. 쿼리는 쿼리 용어에 해당하는 노드에 라우팅됩니다.  
원칙적으로, 이것은 다른 쿼리 용어를 가진 쿼리 스트림이 다른 컴퓨터 세트에 영향을 미치기 때문에 더 큰 동시성을 허용한다. 실제로 어휘 용어별로 인덱스를 분할하는 것은 중요하지 않은 것으로 드러났습니다. 다중 단어 쿼리는 병합을 위해 노드 집합 간에 긴 게시 목록을 전송해야 하며, 그 비용은 더 큰 동시성을 초과할 수 있다. 파티션의 로드 밸런싱은 상대 용어 주파수의 사전 분석이 아니라 시간에 따라 표류하거나 갑작스러운 버스트가 나타날 수 있는 쿼리 용어의 분포와 그 동시 발생에 의해 관리된다. 좋은 파티션을 얻는 것은 쿼리 용어의 동시 발생 함수이며 정량화하기 쉽지 않은 목표를 최적화하기 위한 용어 클러스터링을 수반한다. 마지막으로, 이 전략은 동적 인덱싱을 구현하는 것을 더욱 어렵게 만든다.  
더 일반적인 구현은 문서별로 분할하는 것입니다. 각 노드는 모든 문서의 하위 집합에 대한 색인을 포함합니다. 각 쿼리는 모든 노드에 배포되며 사용자에게 프레젠테이션하기 전에 다양한 노드의 결과가 병합됩니다. 이 전략은 노드 간 통신을 줄이기 위해 더 많은 로컬 디스크 탐색을 거래합니다. 이 접근법의 한 가지 어려움은 단일 노드의 색인이 문서의 하위 집합만 포함하더라도, idf와 같이 점수 매김에 사용되는 전역 통계는 전체 문서 컬렉션에서 계산되어야 한다는 것이다. 이러한 프로세스는 분산 "백그라운드" 프로세스에 의해 계산되며, 이 프로세스는 새로운 전역 통계로 노드 인덱스를 주기적으로 새로 고칩니다.  
문서를 노드로 분할하는 방법은 무엇입니까? 섹션 20.2.1의 크롤러 아키텍처의 개발에 기초하여, 한 가지 간단한 접근 방식은 호스트에서 단일 노드에 모든 페이지를 할당하는 것이다. 이 파티셔닝은 호스트를 크롤러 노드로 파티셔닝할 수 있습니다. 이러한 분할의 위험은 많은 쿼리에서 결과의 우위는 소수의 호스트(그리고 소수의 인덱스 노드)에 있는 문서로부터 온다는 것이다.  
인덱스 노드 공간에 각 URL을 해시하면 노드 간에 쿼리 시간 계산이 보다 균일하게 분산된다. 쿼리 시 쿼리는 각 노드로 브로드캐스트되며 각 노드의 topk 결과가 쿼리에 대한 topk 문서를 찾기 위해 병합됩니다. 일반적인 구현 휴리스틱은 문서 컬렉션을 대부분의 쿼리(예: 21장의 기법을 사용하여)에서 높은 점수를 얻을 가능성이 있는 문서 인덱스와 나머지 문서를 사용하여 낮은 점수를 받는 인덱스로 분할하는 것이다. 섹션 7.2.1에 설명된 대로 높은 점수 인덱스에 일치하는 항목이 너무 적은 경우에만 낮은 점수 인덱스를 검색합니다.

20.4 연결 서버  
21장에서 명확해지기 위해, 웹 검색 엔진은 웹 그래프에서 빠른 연결 쿼리를 지원하는 Connectivity Server 연결 서버를 필요로 한다. 연결 쿼리일반 연결 쿼리는 지정된 URL에 연결되는 URL과 지정된 URL이 연결되는 URL 중 무엇입니까? 이를 위해 URL에서 아웃링크까지, 그리고 URL에서 인링크까지 매핑을 메모리에 저장하고자 합니다. 응용 분야에는 크롤 제어, 웹 그래프 분석, 정교한 크롤 최적화 및 링크 분석이 포함된다(21장에서 다룬다).  
웹이 40억 개의 페이지를 가지고 있고, 각각 다른 페이지로 연결되는 10개의 링크가 있다고 가정해 보자. 가장 간단한 형태로, 우리는 각 링크의 각 끝(소스 및 대상)을 지정하기 위해 32비트 또는 4바이트를 요구하며, 총계가 필요하다.

메모리 바이트 웹 그래프의 일부 기본 속성을 활용하여 이 메모리 요구 사항의 10% 미만으로 사용할 수 있습니다. 첫눈에 다양한 표준 솔루션에 적합한 데이터 압축 문제가 있는 것 같습니다. 그러나 우리의 목표는 단순히 웹 그래프를 메모리에 맞게 압축하는 것이 아니다. 연결 쿼리를 효율적으로 지원하는 방식으로 압축해야 한다. 이 과제는 인덱스 압축(5장)을 연상시킨다.  
우리는 각 웹 페이지가 고유한 정수로 표현된다고 가정한다. 이러한 정수를 할당하는 데 사용되는 특정 체계가 아래에 설명되어 있다. 우리는 반전된 인덱스와 유사한 인접 테이블을 구축한다. 이 테이블은 각 웹 페이지에 대한 행을 가지며 해당 정수로 정렬한다. 페이지 p의 행에는 정렬된 정수 목록이 있으며, 각각은 상단을 연결하는 웹 페이지에 해당합니다. 이 표를 통해 어떤 페이지를 맨 위에 링크하는 양식의 쿼리에 응답할 수 있습니까? 비슷한 방식으로, 우리는 byp에 연결된 페이지들로 된 표를 만든다.

1: [www.stanford.edu/alchemy](http://www.stanford.edu/alchemy)

2: [www.stanford.edu/biology](http://www.stanford.edu/biology)

3: [www.stanford.edu/biology/plant](http://www.stanford.edu/biology/plant)

4: [www.stanford.edu/biology/plant/copyright](http://www.stanford.edu/biology/plant/copyright)

5: [www.stanford.edu/biology/plant/people](http://www.stanford.edu/biology/plant/people)

6: [www.stanford.edu/chemistry](http://www.stanford.edu/chemistry)

◮ Figure 20.5 A lexicographically ordered set of URLs.

이 표 표현은 순진한 표현(우리가 각 링크를 두 개의 끝점, 각각 32비트 정수로 명시적으로 나타내는)으로 취한 공간을 50% 줄인다. 아래 설명은 각 페이지의 링크 표에 초점이 맞춰져 있습니다. 각 페이지 링크 테이블에도 기법이 적용된다는 것을 분명히 해야 합니다. 테이블의 스토리지를 더욱 줄이기 위해 다음과 같은 몇 가지 아이디어를 활용합니다.  
1. 목록 간의 유사성: 표의 많은 행에는 여러 개의 공통 항목이 있습니다. 따라서, 만약 우리가 몇 개의 유사한 행에 대한 프로토타입 행을 명시적으로 나타낸다면, 나머지는 프로토타입 행의 관점에서 간결하게 표현될 수 있다.  
2. 지역: 페이지의 많은 링크는 예를 들어, 동일한 호스트의 페이지인 "근처" 페이지로 이동합니다. 이는 링크의 대상을 인코딩할 때 종종 작은 정수를 사용할 수 있으므로 공간을 절약할 수 있음을 시사한다.  
3. 정렬된 목록에 간격 인코딩을 사용합니다. 각 링크의 대상을 저장하는 대신 행에 이전 항목의 오프셋을 저장합니다.  
  
우리는 이제 이러한 각각의 기술을 개발합니다.  
모든 URL의 사전순으로 각 URL을 영숫자 문자열로 처리하고 이러한 문자열을 정렬합니다. 그림 20.5는 이 정렬된 순서의 세그먼트를 보여줍니다. 진정한 사전 편찬적인 종류의 웹 페이지의 경우, URL의 도메인 이름 부분이 반전되어 www.stanford.edu이 edu.ju.ww가 되어야 하지만, 우리는 주로 단일 호스트에 대한 로컬 링크에 관심이 있기 때문에 여기서는 필요하지 않다.  
각 URL에 대해 이 순서의 위치를 고유 식별 정수로 할당합니다. 그림 20.6은 그러한 번호 매김의 예와 결과 표를 보여준다. 이 예제 순서에서 www.stanford.edu/biology은 정수 2를 할당받았는데, 이는 정수 2는 순서에서 두 번째이기 때문입니다.  
다음으로 우리는 유사성과 인접성을 얻기 위해 대부분의 웹 사이트가 구조화된 방식에서 비롯된 속성을 이용한다. 대부분의 웹 사이트에는 사이트의 각 페이지에서 사이트의 고정 페이지 세트(예: 저작권 통지, 사용 약관 등)로 연결되는 링크 세트가 있는 템플릿이 있습니다. 이 경우 웹 우리는 다음과 같은 전략을 채택한다. 우리는 각 테이블 행을 앞의 7개의 행으로 인코딩하면서 테이블을 따라 내려간다. 그림 20.6의 예에서, 우리는 네 번째 행을 "오프셋 2의 행과 동일"(표에서 앞의 두 행)으로 인코딩할 수 있었고, 9는 8로 대체되었다. 이를 위해서는 오프셋, 드롭된 정수(이 경우 9), 추가된 정수(이 경우 8)의 사양이 필요합니다. 7개의 선행 행만 사용하면 두 가지 이점이 있다. (i) 오프셋은 3비트만 사용하여 표현할 수 있다. 이 선택은 경험적으로 최적화된다. (7개의 선행 행과 8개의 선행 행에 대한 이유는 연습 20.4의 주제이다.) 그리고 (ii) 7개의 최대 오프셋을 인간 사이에서 값비싼 검색을 수행할 필요가 없는 것과 같은 작은 값으로 고정시킨다.y 후보 프로토타입은 현재 행을 표시할 대상입니다.  
앞의 7행 중 현재 행을 표현하기에 적합한 프로토타입이 없는 경우 어떻게 해야 합니까? 예를 들어, 이것은 우리가 테이블의 행들을 걸어갈 때 서로 다른 웹사이트들 사이의 각각의 경계에서 일어날 것이다. 이 경우 행은 빈 집합에서 시작하여 해당 행의 각 정수를 "추가"하는 것으로 표현됩니다. 간격 인코딩을 사용하여 각 행의 간격(실제 정수보다는)을 저장하고 값의 분포를 기반으로 이러한 간격을 단단히 인코딩함으로써 추가 공간 감소를 얻는다. 섹션 20.5에 언급된 실험에서, 여기에 요약된 일련의 기술은 링크당 평균 3비트를 사용하는 것으로 나타났다. 이는 순진한 표현에 필요한 64비트에서 크게 줄어든 것이다.  
이러한 아이디어는 메모리에 쉽게 맞는 상당한 규모의 웹 그래프를 나타내지만, 우리는 여전히 연결 쿼리를 지원해야 합니다. 이 표현에서 페이지에서 링크 집합을 검색하는 데 필요한 사항은 무엇입니까? 먼저 URL에서 테이블의 행 번호로 (해시) 인덱스를 조회해야 합니다. 다음으로, 우리는 이러한 항목을 재구성해야 하며, 이 항목은 다른 행의 항목으로 인코딩될 수 있다. 여기에는 다른 행들을 재구성하기 위한 오프셋을 따라야 합니다. 이 프로세스는 원칙적으로 많은 수준의 간접적인 과정을 거치게 됩니다. 그러나 실제로는 이런 일이 자주 일어나지 않는다. 이를 제어하기 위한 휴리스틱은 표 구성에 도입될 수 있다. 현재 행을 모델링할 후보 행으로 앞의 7 행을 검토할 때 현재 행과 후보 프로토타입 사이의 유사성 임계값을 요구한다. 이 임계값은 주의하여 선택해야 합니다. 임계값이 너무 높게 설정되면 프로토타입을 거의 사용하지 않고 많은 행을 새로 표현합니다. 임계값이 너무 낮으면 대부분의 행이 프로토타입으로 표현되므로 쿼리 시간에 행을 재구성하면 이전 프로토타입을 통해 여러 수준의 간접적인 방식으로 이어진다.  
  
연습 20.4  
우리는 7개의 선행 행 중 하나의 관점에서 행을 표현하면 프로토타입으로 사용 중인 선행 행 중 어느 것을 지정하기 위해 3비트 이하를 사용할 수 있다는 점에 주목했다. 왜 여덟 개의 선행 행이 아니라 일곱 개의 선행 행입니까? (힌트: 앞의 7개 행 중 어느 것도 좋은 프로토타입이 아닌 경우를 고려하십시오.)  
연습 20.5  
섹션 20.4의 체계에 대해, URL에서 발생한 링크 사건을 디코딩하면 많은 수준의 간접적인 결과를 초래할 수 있다는 점에 주목했다. 간접 수준 수가 URL 수에 따라 선형적으로 증가하는 예를 구성합니다.사이트의 페이지에 해당하는 행에는 공통적으로 많은 표 항목이 있습니다. 또한 URL 사전순으로 웹 사이트의 페이지가 표에서 연속 행으로 나타날 가능성이 매우 높습니다.

1: 1, 2, 4, 8, 16, 32, 64

2: 1, 4, 9, 16, 25, 36, 49, 64

3: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

4: 1, 4, 8, 16, 25, 36, 49, 64

◮ Figure 20.6 A four-row segment of the table of links.

우리는 다음과 같은 전략을 채택한다. 우리는 각 테이블 행을 앞의 7개의 행으로 인코딩하면서 테이블을 따라 내려간다. 그림 20.6의 예에서, 우리는 네 번째 행을 "오프셋 2의 행과 동일"(표에서 앞의 두 행)으로 인코딩할 수 있었고, 9는 8로 대체되었다. 이를 위해서는 오프셋, 드롭된 정수(이 경우 9), 추가된 정수(이 경우 8)의 사양이 필요합니다. 7개의 선행 행만 사용하면 두 가지 이점이 있다. (i) 오프셋은 3비트만 사용하여 표현할 수 있다. 이 선택은 경험적으로 최적화된다. (7개의 선행 행과 8개의 선행 행에 대한 이유는 연습 20.4의 주제이다.) 그리고 (ii) 7개의 최대 오프셋을 인간 사이에서 값비싼 검색을 수행할 필요가 없는 것과 같은 작은 값으로 고정시킨다.y 후보 프로토타입은 현재 행을 표시할 대상입니다.  
앞의 7행 중 현재 행을 표현하기에 적합한 프로토타입이 없는 경우 어떻게 해야 합니까? 예를 들어, 이것은 우리가 테이블의 행들을 걸어갈 때 서로 다른 웹사이트들 사이의 각각의 경계에서 일어날 것이다. 이 경우 행은 빈 집합에서 시작하여 해당 행의 각 정수를 "추가"하는 것으로 표현됩니다. 간격 인코딩을 사용하여 각 행의 간격(실제 정수보다는)을 저장하고 값의 분포를 기반으로 이러한 간격을 단단히 인코딩함으로써 추가 공간 감소를 얻는다. 섹션 20.5에 언급된 실험에서, 여기에 요약된 일련의 기술은 링크당 평균 3비트를 사용하는 것으로 나타났다. 이는 순진한 표현에 필요한 64비트에서 크게 줄어든 것이다.  
이러한 아이디어는 메모리에 쉽게 맞는 상당한 규모의 웹 그래프를 나타내지만, 우리는 여전히 연결 쿼리를 지원해야 합니다. 이 표현에서 페이지에서 링크 집합을 검색하는 데 필요한 사항은 무엇입니까? 먼저 URL에서 테이블의 행 번호로 (해시) 인덱스를 조회해야 합니다. 다음으로, 우리는 이러한 항목을 재구성해야 하며, 이 항목은 다른 행의 항목으로 인코딩될 수 있다. 여기에는 다른 행들을 재구성하기 위한 오프셋을 따라야 합니다. 이 프로세스는 원칙적으로 많은 수준의 간접적인 과정을 거치게 됩니다. 그러나 실제로는 이런 일이 자주 일어나지 않는다. 이를 제어하기 위한 휴리스틱은 표 구성에 도입될 수 있다. 현재 행을 모델링할 후보 행으로 앞의 7 행을 검토할 때 현재 행과 후보 프로토타입 사이의 유사성 임계값을 요구한다. 이 임계값은 주의하여 선택해야 합니다. 임계값이 너무 높게 설정되면 프로토타입을 거의 사용하지 않고 많은 행을 새로 표현합니다. 임계값이 너무 낮으면 대부분의 행이 프로토타입으로 표현되므로 쿼리 시간에 행을 재구성하면 이전 프로토타입을 통해 여러 수준의 간접적인 방식으로 이어진다.  
  
연습 20.4  
우리는 7개의 선행 행 중 하나의 관점에서 행을 표현하면 프로토타입으로 사용 중인 선행 행 중 어느 것을 지정하기 위해 3비트 이하를 사용할 수 있다는 점에 주목했다. 왜 여덟 개의 선행 행이 아니라 일곱 개의 선행 행입니까? (힌트: 앞의 7개 행 중 어느 것도 좋은 프로토타입이 아닌 경우를 고려하십시오.)  
연습 20.5  
섹션 20.4의 체계에 대해, URL에서 발생한 링크 사건을 디코딩하면 많은 수준의 간접적인 결과를 초래할 수 있다는 점에 주목했다. 간접 수준 수가 URL 수에 따라 선형적으로 증가하는 예를 구성합니다.

20.5 참조 및 추가 판독  
최초의 웹 크롤러는 1993년 봄에 쓰여진 Matthew Gray’s Wanderer 로 보인다. 메르카토르 크롤러는 Najork and Heydon (**Najork and Heydon 2001; 2002**); 때문이다. 이 장의 치료는 그들의 작업에 따른다. 웹 크롤링에 대한 다른 고전적인 초기 기술로는 **Burner (1997)**, **Brin and Page (1998), Cho et al. (1998)** 스탠퍼드 웹베이스 시스템(**Hirai et al. 2000**).이 있다. 2000). **Cho and Garcia-Molina (2002)**는 분산형 크롤러 노드 간의 다양한 통신 모드에 대한 분류법과 비교 연구를 제공한다. 로봇 배제 프로토콜 표준은 http://www.robotstxt.org/wc/exclusion.html에 설명되어 있다. **Boldi et al. (2002)** and **Shkapenyuk and Suel (2002)**은 대규모 분산형 웹 크롤러 구현에 대한 보다 최근의 세부 정보를 제공한다.  
DNS 해상도에 대한 우리의 논의 (섹션 20.2.2)는 IPv4 (Internet Protocol version 4의 경우)라고 알려진 인터넷 주소에 대한 현재의 규약을 사용한다. 각 IP 주소는 4 바이트의 순서이다. 미래에는 주소 규약(총칭 인터넷 주소 공간)이 IPv6(http://www.ipv6.org/)로 알려진 새로운 표준을 사용할 가능성이 높다.  
**Tomasic and Garcia-Molina (1993)** and **Jeong and Omiecinski (1995)**는 분산 인덱스에 대한 용어 분할 대 문서 분할을 평가하는 주요 초기 논문이다. 문서 분할은 적어도 항 분포가 일반적으로 그러하듯이 치우쳐 있는 경우 우수한 것으로 확인됩니다. 이 결과는 더 최근의 연구에서 일반적으로 확인되었다(**MacFarlane et al. 2000**). 그러나 결과는 분산형 시스템의 세부사항에 따라 다르다. 온라인 판 (c) 2009 캠브리지 UP 20.5 참조 및 추가 459 이상의 작업 스레드가 반대 결론에 도달했다(**RibeiroNeto and Barbosa 1998, Badue et al. 2001**). **Sornil (2001)**은 용어 분할과 문서 분할 간의 혼합인 분할 방식을 주장한다. **Barroso et al. (2003)**는 Google에서 사용되는 배포 방법을 설명합니다. 연결 서버의 첫 번째 구현은 **Bharat et al. (1998)**.에 의해 설명되었다. 이 장에서 논의된 체계는 현재 가장 잘 발표된 체계(인코딩을 위한 링크당 3비트를 달성함)로 여겨지며 **Boldi and Vigna (2004a;b)**.의 일련의 논문에서 설명된다.

1. The number of hosts is assumed to far exceed B. [↑](#footnote-ref-1)
2. Please note the different usage of “clusters” elsewhere in this book, in the sense of Chapters 16 and 17 [↑](#footnote-ref-2)