Welcome to DHSI 2016!

Thanks for joining the DHSI community!

In this booklet, you will find essential course materials prefaced by some useful information about getting settled initially at UVic, finding your way around, getting logged in to our network (after you’ve registered the day before our courses begin, and received your login information), and so on.

Given our community’s focus on things computational, it will be a surprise to no one that we might expect additional information online for some of the classes — your instructors will let you know — or that the most current version of all DHSI-related information may be found on our website at dhsi.org.

Do check in there first if you need anything that’s not in this coursepak.

And please don’t hesitate to be in touch with us at institut@uvic.ca or via Twitter at @AlyssaA_DHSI or @DHInstitute if we can be of any help ....
Sunday, 5 June 2016 [DHSI Registration, Meetings, Workshops]

9:00 to 4:00
- Early Class Meeting: 3. [Foundations] DH For Department Chairs and Deans (David Strong C114, Classroom)
  
  Further details are available from instructors. Registration materials will be available in the classroom.

12:30 to 5:00
- DHSI Registration (NEW LOCATION: MacLaurin Building, Room A100)
  
  After registration, many will wander to Cadboro Bay and the pub at Smuggler’s Cove OR the other direction to Shelbourne Plaza and Maude Hunter’s Pub.

1:00 to 4:00
- 3-hour Workshops
  - 45. 3D Visualization for the Humanities [5 June] (MacLaurin D107, Classroom)
  - 46. Beyond TEI: Metadata for Digital Humanities [5 June] (MacLaurin D109, Classroom)

4:00 to 5:00
- Workshop:
  - 44. Twitter Basics: An Introduction to Social Media [5 June] (MacLaurin D105, Classroom)

Monday, 6 June 2016

Your hosts for the week are Ray Siemens and Dan Sondheim.

7:45 to 8:15
- Last-minute Registration (MacLaurin Building, Room A100)

8:30 to 10:00
- Welcome, Orientation, and Instructor Overview (MacLaurin A144)

10:15 to Noon
- Classes in Session (click for details and locations)
  1. [Foundations] Text Encoding Fundamentals and their Application (Clearihue A102, Lab)
  2. [Foundations] Digitisation Fundamentals and their Application (Clearihue A051, Lab)
  3. [Foundations] DH For Department Chairs and Deans (David Strong C114, Classroom)
  4. [Foundations] Fundamentals of Programming/Coding for Human(s|ists) (MacLaurin D103, Lab)
  5. [Foundations] Understanding The Predigital Book: Technology and Texts (McPherson Library A003, Classroom)
  6. Out-of-the-Box Text Analysis for the Digital Humanities (Human and Social Development A160, Lab)
  7. Geographical Information Systems in the Digital Humanities (Human and Social Development A170, Lab)
  8. CloudPowering DH Research (David Strong C108, Classroom)
  9. Digital Storytelling (MacLaurin D111, Classroom)
 10. Digital Humanities Pedagogy: Integration in the Curriculum (Cornett A121, Classroom)
 11. Text Processing - Techniques & Traditions (Cornett A229, Classroom)
 12. Issues in Large Project Planning and Management (Hickman 120, Classroom)
 13. 3D Modelling for the Digital Humanities and Social Sciences (MacLaurin D010, Classroom)
 14. Digital Humanities Databases (Clearihue D132, Classroom)
 15. Creating LAMP Infrastructure for Digital Humanities Projects (Human and Social Development A270, Classroom)
 16. RDF and Linked Open Data (Human and Social Development A264, Classroom)
 17. Palpability and Wearable Computing (MacLaurin D016, Classroom)
 18. Drupal for Digital Humanities Projects (MacLaurin D109, Classroom)
 19. Introduction to Electronic Literature in DH: Research and Practice (MacLaurin D115, Classroom)

12:15 to 1:15
- Lunch break / Unconference Coordination Session (MacLaurin A144)

1:30 to 4:00
- Classes in Session

Institute Panel: Perspectives on DH (or, #myDHis ...)
- Jason Boyd (Ryerson U): “At Play in the Digital Humanities.”
  
  Abstract: This presentation will move from discussion of the Texting Wilde Project and designing playful interfaces for it to a more general reflection on the place of play in DH, informed by work in games studies,
Liz Grumbach (Texas A&M U): "Modding the Academy: eMOP, ARC, and Emerging (Digital) Humanities Paradigms."
Abstract: As "research staff" at the Initiative for Digital Humanities, Media, and Culture (IDHMC) at Texas A&M University, my digital humanities is about collaboration, project development, and finding effective methods for facilitating research in an academic field that still privileges traditional humanities research methodologies over emerging forms of scholarly communication and knowledge production. I will briefly describe two interdisciplinary projects run out of the IDHMC (the Early Modern OCR Project and the Advanced Research Consortium) to facilitate further discussion of how emerging digital methodologies can remodel our perceptions of how and where scholarly research is produced.

Claudia von Vacano (UC Berkeley): "Data Science in the Service of Humanity at Berkeley."
Abstract: The Division of Arts and Humanities at Berkeley is building an infrastructure to buttress digital humanities in conversation with data science. This includes critical analysis of a data science initiative. In this short talk, Digital Humanities (DH) at Berkeley Director Claudia von Vacano discusses radical organizational change within a strongly-established institution. Claudia argues that it is incumbent upon us to bring greater diversity to data science thinking in order to prevent data science from becoming a monolithic enterprise across universities. Now more than ever artists and humanists need to ensure that data science is at the service of humanity in its various expressions. Claudia discusses the specific organizational structures at Berkeley that are supporting this ambitious endeavor.

For a Digital Humanities graduate seminar, I developed an assignment that aimed to introduce students to the basic elements of constructing a digital edition based on a given text (Claude McKay’s 1922 collection of poetry, "Harlem Shadows"). The project was fully collaborative; fundamental decisions regarding the purpose of the site, the intended audience, and site design were left entirely to students, with research input from faculty. This turned out to be an effective way to get students new to DH oriented in at least one area of the field, while also keeping questions of social justice and representation of minority writers online a central focus.

Abstract: DH researchers increasingly find themselves in need of research supports that extend beyond the services offered in traditional humanities departments. These might include server space, programming expertise, database management, metadata support, and digital preservation services. Libraries need access to faculty and student subject expertise to enrich online collections and exhibits, and to improve discovery and research interfaces. There is an enormous amount of complementarity in our needs and interests, but new approaches are necessary to help us bridge dissimilar governance and funding models. This presentation will draw on our experience at UVic Libraries to propose models that can foster closer collaboration for the mutual benefit of the library and its DH partners.

Co-Chairs: Constance Crompton (UBC Okanagan) and Matt Huculak (U Victoria)

5:00 to 6:00
Light Reception
(Felicitas, Student Union Building)

Tuesday, 7 June 2016

4:10 to 5:00

4:15 to 5:30
DHSI Colloquium Session 1 (MacLaurin A144)
- Chair: TBA
- Working at the Intersection of Renaissance Studies and DH: An Update on Iter Initiatives. Randa El Khalib (U Victoria); Lindsey Seatter (U Victoria); William Bowen (U Toronto, Scarborough); Daniel Powell (King's College London); Ray Siemens (U Victoria)
- The NYU Libraries Web Hosting Pilot: An Update and Lessons Learned. Zach Coble (New York U)
- Exploring Place in the French of Italy. Heather Hill (Fordham U)
- Digital Representations of Petrarch's "Rerum Vulgarium Fragmenta". Isabella Magni (Indiana U)
- Voices of Southern Patagonia. Gustavo Navarro (U Nacional de la Patagonia Austral)

Wednesday, 8 June 2016

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
Thursday, 9 June 2016

1:30 to 4:00
Classes in Session

4:15 to 5:30
DHSI Colloquium Session 2 (MacLaurin A144)
- Chair: TBA
- Transforming Bad OCR into Useful Metadata? Exploring NLP Possibilities. Evan Williamson (U Idaho)
- Piloting Linked Open Data for Artists' Research at University of California, Irvine. Emilee Mathews (U California, Irvine)
- Four Words: The Role of Theory in Digital Humanities. Grant Glass (U North Carolina, Chapel Hill)
- Rethinking the Exhibition Catalogue: Documentation, Curation, and the Digital Humanities Project. Julia Polycyk-O'Neill (Brock U); Aleksandra Kaminska (Simon Fraser U)

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
[Instructor lunch meeting]

1:30 to 4:00
Classes in Session

4:15 to 5:00
DHSI Colloquium Session 3. Special Theme: Gender and the Digital (MacLaurin Building, Room A144)
- Chair: TBA
- A Textual Analysis of Female Renaissance Playwrights using R. Elizabeth Ramsay (Trent U)
- The Comparison of Human-Reading and Machine-Reading in Le Système de la Nature. Maryam Mozafari (Simon Fraser U)
- Does Gender Affect How Genre-Conformingly Writers Write? Sayan Bhattacharyya (U Illinois, Urbana-Champaign); Ted Underwood (U Illinois, Urbana-Champaign)

5:00 to 5:30
DHSI Colloquium (contd.), Electronic Literature Affiliated Event: Alan Sondheim, "Language, Accident, and the Virtual" (MacLaurin Building, Room A144)
Abstract: This talk engages concepts of blankness, geography, gamespace in virtual worlds, and what I term edgespace - the limits of the gamespace, where language occurs and seethes. I argue that the phenomenology of the real comes into play when living spaces are abandoned, where broken geographies are signs of a future already present. I present instances of digital language production in such spaces, working through virtual worlds such as Second Life and the Macgrid, as well as self-contained Open Sim software that can be run on most computers.

The edgespace is always uneasy, tottering, catastrophic; it is the space of the unalloyed digital, where things no longer operate within a classical or modernist tradition. Increasingly, this space characterizes our current place in the world, with its fractured media histories and environments of scorched earth, environmental depredation, and slaughter. We can work through and within such spaces, developing (as perhaps Occupy did) new forms of production, resistance, and digital culture.

5:30 to ~6:15
Performance: Vibrant Lives (Jessica Rajko, Eileen Standley, Jacque Wernimont, Stjepan Rajko) (Just outside MacLaurin Building, Room A100)
Critically commenting on this use of personal data, Vibrant Lives is an interactive installation that gives audiences a real-time sense of their own voluminous “data shed” (the data that we share as a part of everyday life). In this, Vibrant Lives troubles boundaries erected by ideas of disembodied, abstracted, “immaterial” metadata and people. By juxtaposing the different ways that we engage with technologies of communication and preservation, we ask our audiences to consider interplays of value, valuation, technology, information, and material/matter. Rather than exploring more traditional audio/visual methods for sharing data information, this work offers haptic (touch-based) feedback to elicit a more visceral understanding of what it means to “shed data.”

Friday, 10 June 2016 [DHSI; ELO and INKE Opening]

8:00 to 9:30
Electronic Literature Organization + INKE Registration (MacLaurin Building, Room A100)

9:00 to 9:30
ELO Welcome

9:00 to Noon
DHSI Classes in Session
9:30 to Noon  
Electronic Featured Session A

12:15 to 1:15  
Electronic Literature Organization + INKE Registration (MacLaurin Building, Room A100)
DHSI Lunch Reception / Course E-Exhibits (MacLaurin A100)  
ELO Gallery Exhibit Opens

1:30 to 3:00  
INKE Welcome

✓ Joint Institute Lecture, ELO, INKE, and DHSI
Panel Discussants: Stephanie Boluk (Pratt Institute), Diane Jakacki (Bucknell U), Elizabeth Loth (UC San Diego), and Anastasia Salter (U Central Florida)
Co-Chairs: Dene Grigar (Washington State U, Vancouver), Jon Bath (U Saskatchewan), Ray Siemens (U Victoria)
(MacLaurin A144)

Abstract: Can a feminist war game exist? War, traditionally the sole purview of men, is an essential site for asking critical questions about masculinist systems and mediated representations, especially since subjects, objects and agents are all instruments within the ideological narratives that frame the brutal history of armed conflict. Simply including female “warrior” characters in a war game, for example, continues to normalize the mechanism of war while extending its “inclusiveness” to groups that have traditionally been marginalized and victimized by it. However, prototyping complex intersections between mechanisms of war, digital game narratives, and inclusive feminist values suggests that feminist discourses can be used to denaturalize and reframe narratives of war in spaces of programmed play.

3:15 to 4:45  
ELO Concurrent Session 1: 1.1 Best Practices for Archiving E-Lit; 1.2 Medium & Meaning; 1.3 A Critical Look at E-Lit; 1.4 Literary Games; 1.5 eLit and the (Next) Future of Cinema; 1.6 Authors & Texts
ELO Action Session 1

5:00 to 6:00  
Joint Reception: ELO, INKE, and DHSI (University Club)

ELO Colloquium Poster Session
ELO New Scholars Poster Session

DHSI Colloquium Poster Session Presenters:
- Practicums in the Digital Humanities: Four KPU Case Studies. Greg Chan (Kwanten Polytechnic U) and ENGL 4300 Students (TBD)
- The HathiTrust Research Center: Supporting Large Scale Analysis of the HathiTrust Digital Library. Sayan Bhattacharyya (U Illinois); Nicholae Cline (Indiana U); Eleanor Dickson (U Illinois); Leanne Mobley (Indiana U)
- Digital Scholarship in the Institutional Repository. Jeri Wieringa (George Mason U)
- Canada’s Early Women Writers and All Their Relatives. Karyn Huenemann (Simon Fraser U)
- Possible Spanish Idiom In A Name At Nootka. Paula Johanson (U Victoria)

INKE Poster Session Presenter:
- Genre-based Approaches to Domain Modeling: A Case Study Involving Rulebooks. Kelly Colht (Information School, U Washington)

ELO Poster Session Presenters:
- Social Media for E-Lit Authors. Michael Rabby (Washington State U Vancouver)
- O True Apothecary! Kyle Booten (UC Berkeley, Center for New Media)
- Electronic Literature Production – A Case Study of Korporacja Ha!art. Aleksandra Malecka (Jagiellonian U); Piotr Marecki (Jagiellonian U)
- How to Upset the Contributors? The Responses of Collaborating Readers to the Publication of Crowdsourced Literature. Heiko Zimmermann (U Trier)
- Life Experience through Digital Simulation Narratives. David Nunez Ruiz (Neotipo)
- Building Stories. Kate Palermi (Washington State U Vancouver)
- Help Wanted and Skills Offered. by Deena Larsen (Independent Artist); Julianne Chatelain (U.S. Bureau of Reclamation)
- Beyond Original E-Lit: Deconstructing Austen Cybertexts. Meredith Dabek (Maynooth U)
- Arabic E-Lit (AEL) Project. Riham Hosny (Rochester Institute of Technology / Minia U)

7:30 to 9:00  
ELO Readings and Performances (Felicitas, Student Union Building)

Saturday, 11 June 2016 [ELO + INKE + Suggested Outings!]

8:00 to Noon  
Electronic Literature Organization + INKE Registration (MacLaurin Building, Room A100)

8:30 to 10:00  
ELO Lightning Round
ELO Gallery Exhibit Opens (10:00)
INKE Session 1

ELO Concurrent Session 2: 2.1 Literary Interventions; 2.2 E-Lit in Global Contexts; 2.3 Theoretical
<table>
<thead>
<tr>
<th>Time</th>
<th>Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:30 to Noon</td>
<td>Underpinnings; 2.4 E-Lit in Time and Space; 2.5 Understanding Bots</td>
</tr>
<tr>
<td></td>
<td>ELO Action Session 2</td>
</tr>
<tr>
<td></td>
<td>INKE Session 2</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch</td>
</tr>
<tr>
<td></td>
<td>ELO Artists’ Talks</td>
</tr>
<tr>
<td>1:30 to 3:00</td>
<td>ELO Concurrent Session 3: 3.1 E-Lit Pedagogy in Global Setting; 3.2 The Art of Computational Media; 3.3 Present Future Past; 3.4 Beyond Collaborative Horizons; 3.5 E-Loops, Reshuffling Reading And Writing In Electronic Literature Works; 3.6 Metaphorical Perspectives; 3.7 Embracing Bots</td>
</tr>
<tr>
<td></td>
<td>ELO Workshops (to 3.30)</td>
</tr>
<tr>
<td></td>
<td>INKE Session 3</td>
</tr>
<tr>
<td>3:30 to 5:00</td>
<td>ELO Featured Session B</td>
</tr>
<tr>
<td></td>
<td>INKE Session 4</td>
</tr>
<tr>
<td>6:00 to 9:00</td>
<td>ELO Informal Banquet (University Club) (INKE participants are invited to hang out with our ELO pals, too!)</td>
</tr>
<tr>
<td>9:00 to Midnight</td>
<td>ELO Screenings</td>
</tr>
</tbody>
</table>

**All day Suggested Outings**

<table>
<thead>
<tr>
<th>Outing</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suggested Outing 1</td>
<td>Botanical Beach (self-organised; car needed) A self-guided visit to the wet, wild west coast tidal shelf (and historically-significant former research site) at Botanical Beach: we recommend departing early (around 8.00 am) to catch low tide for a better view of the wonderful undersea life! Consider bringing a packed lunch to nibble-on while looking at the crashing waves when there, and then have an afternoon drink enjoying the view from the deck of the Port Renfrew Hotel.</td>
</tr>
<tr>
<td>Suggested Outing 2</td>
<td>Butchart Gardens (self-organised) A shorter journey to the resplendently beautiful Butchart Gardens and, if you like, followed by (ahem) a few minutes at the nearby Church and State Winery, in the Saanich Peninsula. About an hour there by public bus from UVic, or 30 minutes by car.</td>
</tr>
<tr>
<td>Suggested Outing 3</td>
<td>Salthspring Island (self-organised; a full day, car/bus + ferry combo) Why not take a day to explore and celebrate the funky, laid back, Canadian gulf island lifestyle on Salthspring Island, Ferry departs regularly from the Schwartz Bay ferry terminal, which is about one hour by bus / 30 minutes by car from UVic. You may decide to stay on forever ...</td>
</tr>
<tr>
<td>Suggested Outing 4</td>
<td>Paddling Victoria’s Inner Harbour (self-organised) A shorter time, seeing Victoria’s beautiful city centre from the waterways that initially inspired its foundation. A great choice is the day is sunny and warm. Canoes, kayaks, and paddle boards are readily rented from Ocean River Adventures and conveniently launched from right behind the store. Very chill.</td>
</tr>
<tr>
<td>And more!</td>
<td>Self-organised High Tea at the Empress Hotel, scooter rentals, visit to the Royal BC Museum, darts at Christies Carriage House, a hangry breakfast at a local diner, whale watching, kayaking, brew pub sampling (at Spinnaker’s, Swans, Moon Under Water, and beyond!), paddle-boarding, a tour of used bookstores, and more have also been suggested!</td>
</tr>
</tbody>
</table>

**Sunday, 12 June 2016 [ELO + DHSI Registration, Workshops]**

<table>
<thead>
<tr>
<th>Time</th>
<th>Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30 to 10:00</td>
<td>ELO Town Hall Meeting</td>
</tr>
<tr>
<td></td>
<td>ELO Gallery Exhibit Opens (10:00)</td>
</tr>
<tr>
<td>10:30 to Noon</td>
<td>ELO Concurrent Session 4: 4.1 Narratives and Narrativity; 4.2 Historical &amp; Critical Perspectives; 4.3 Emergent Media; 4.4 Narrative and Poetic Experiences; 4.5 The E-Literary Object; 4.6 Next Narrative ELO Action Session 3</td>
</tr>
<tr>
<td>12:15 to 1:30</td>
<td>Lunch</td>
</tr>
<tr>
<td></td>
<td>ELO Artists’ Talks</td>
</tr>
<tr>
<td>12.30 to 5:00</td>
<td>DHSI Registration (NEW LOCATION: MacLaurin Building, Room A100) After registration, many will wander to Cadboro Bay and the pub at Smuggler’s Cove OR the other direction to Shelbourne Plaza and Maude Hunter's Pub.</td>
</tr>
<tr>
<td>1:00 to 4:00</td>
<td>DHSI 3-hour Workshops</td>
</tr>
<tr>
<td></td>
<td>- 47. Crowdsourcing as a Tool for Research and Public Engagement [12 June] (David Strong C114, Classroom)</td>
</tr>
<tr>
<td></td>
<td>- 48. Text Analysis with the HathiTrust Research Center [12 June] (Cornett B129, Classroom)</td>
</tr>
<tr>
<td></td>
<td>- 49. Dynamics of Explanatory Annotation [12 June] (Clearihue D131, Classroom)</td>
</tr>
<tr>
<td></td>
<td>- 50. Dynamic Ontologies for the Humanities [12 June] (Cornett A128, Classroom)</td>
</tr>
<tr>
<td></td>
<td>- 52. Understanding Digital Video [12 June] (Human and Social Development A270, Classroom)</td>
</tr>
</tbody>
</table>
Monday, 13 June 2016

Your hosts for the week are Alyssa Arbuckle, Ray Siemens and Dan Sondheim.

8:15 to 8:45
DHSI Last-minute Registration (MacLaurin Building, Room A100)

9:00 to 10:00
Welcome, Orientation, and Instructor Overview (MacLaurin A144)

10:15 to Noon
Classes in Session (click for details and locations)

12:15 to 1:15
Lunch break / Unconference Coordination Session (MacLaurin A144)

1:30 to 4:00
Institute Lecture
Laura Estill (Texas A&M): "Digital Futures: Long-term Planning for your Project"
Chair: TBA (MacLaurin A144)

Abstract: The start of every digital project is as exhilarating as it is overwhelming. But when should you start looking ahead to the end? And does there have to be an end? As Robin Camille Davis reports, 45% of projects discussed at DH '05 are no longer available online. In this talk, I'd like to offer some best practices for thinking about your DH project in the long term, from workflow and technologies to people and resources. I will share my experience taking on a long-standing existing digital project (The World Shakespeare Bibliography Online). Although thinking about documentation and archiving might strike terror in your hearts, or make your eyelids droop with boredom, ultimately, having a well-thought-out plan leads to serenity and peace of mind (or at least leaves you free to worry about other important things, like how you're not flossing enough).

5:00 to 6:00
Light Reception (Felicitas, Student Union Building)
Tuesday, 14 June 2016

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference

1:30 to 4:00
Classes in Session

4:15 to 6:00
DHSI Colloquium Session 4. Special Theme: Building an Inclusive DH Community (MacLaurin A144)
- Chair: TBA
- Working in the Digital Humanities - An Exploration of Scholarly Practices for Early-career Academics. Steve Anderson (U California, Riverside); Matt Bouchard (U Toronto); Andy Keenan (U Toronto); Lee Zickel (Case Western Reserve U)
- DH Internships: Building Digital Humanities Capacity with Care at Emory. Alan G. Pike (Emory U)
- ABC's of Gamification: How we Gamified a Social Media Course. Rob Bajko (Ryerson U), Jaigris Hodson (Royal Roads U)
- Undergraduate Contributorship Models with TaDiRAH. Aaron Mauro (Penn State, Behrend)
- The Anti-MOOC: An Online Small Seminar Format for Distance Mentoring and Digital Public History Projects. Cathy Kroll (Sonoma State U)

Wednesday, 15 June 2016

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
ILiADS: Institute for Liberal Arts Digital Scholarship information session (location TBA)

1:30 to 4:00
Classes in Session

4:15 to 6:00
DHSI Colloquium Session 5 (MacLaurin A144)
- Chair: TBA
- Sounds and Digital Humanities. John F. Barber (Washington State U, Vancouver)
- Mapping a Digital History of Big Science. Elyse Graham (SUNY, Stony Brook)
- Mapping a Global Renaissance with 53,829 Texts. James Lee (Grinnell College)
- Quantifying the Language of Othering. Bryor Snejdella (McMaster U)
- Digitization and Dissemination of Movable Books Data. Emily Brooks (U Florida)

Thursday, 16 June 2016

9:00 to Noon
DHSI Classes in Session

12:15 to 1:15
Lunch break / Unconference
(Instructor lunch meeting)

1:30 to 4:00
Classes in Session

4:15 to 6:00
DHSI Colloquium Session 6 (MacLaurin A144)
- Chair: TBA
- Using DH to Increase Legal Literacy and Agency. Susan Tanner (Carnegie Mellon U)
- Detecting Text Reuse in Nineteenth-Century Legal Codes of Civil Procedure. Lincoln Mullen (George Mason U); Kellen Funk (Princeton U)
- Collection, Curation, and Collaboration: Representing Canadian Gay Liberationists. Nadine Boulay (Simon Fraser U); Anderson Tuguinay (Ryerson U); Stefanie Martin (Ryerson U); Seamus Riordan-Short (U British Columbia, Okanagan); Raymon Sandhu (U British Columbia, Okanagan)
- Privacy, Legality, and Feminism: How Do We Build a Feminist Politics into Open Access Data Structures? Emily Christina Murphy (Queen’s U)
- Recreating the Fiction Factory: The James Malcolm Rymer Collection. Rebecca Nesvet (U Wisconsin, Green Bay)
- Digital History and Archiving: Fostering the “Afterlife” and Accessibility of American Civil War Letters. Ashley Hughes (Texas Christian U)
<table>
<thead>
<tr>
<th>Time</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch Reception / Course E-Exhibits (MacLaurin A100)</td>
</tr>
<tr>
<td>1:30 to 2:45</td>
<td>Awards and Bursaries Recognition</td>
</tr>
<tr>
<td></td>
<td>Institute Lecture</td>
</tr>
<tr>
<td></td>
<td>James Cummings (Oxford U): “Grass Roots and Ivory Towers: Building communities and inspiring participation in the Digital Humanities”</td>
</tr>
<tr>
<td></td>
<td>Chair: TBA</td>
</tr>
<tr>
<td></td>
<td>(MacLaurin A144)</td>
</tr>
<tr>
<td>2:45 to 3:00</td>
<td>Closing, DHSI in Review (MacLaurin A144)</td>
</tr>
</tbody>
</table>

Abstract: The various fields and organisations in the big tent of Digital Humanities all approach community involvement in different ways. Outside traditional academic teaching, much of the work is done by communities which thrive partly as a result of in-kind commitments and volunteer labour, often donated in evenings and weekends by those striving to make their particular endeavour a success. By re-examining a number of the DH communities I have been involved with, I hope to trace some of their successes, failures, risks, and the necessary fragilities of their development and undertakings. This re-examination will encompass a range of community projects from entirely grass roots and mostly finance-free endeavours such as the open DH Awards (http://dhawards.org) and Digital Medievalist (http://digitalmedievalist.org) project through to self-funding break-even institutional activities such as DHOxSS (http://digital.humanities.ox.ac.uk/dhoxss/) and membership consortia such as the TEI Consortium (http://www.tei-c.org/). Each of these projects employs different strategies to ensure that their work relates to their communities and engages the public, with some methods proving more successful than others in eliciting help or volunteer labour from these communities. By investigating these methods, I hope to tease out the more successful strategies and to discuss how these might be employed by those setting up new DH initiatives.

Contact info:  
institut@uvic.ca P: 250-472-5401 F: 250-472-5681
Regional Map of Greater Victoria

Average Frequency
Regional Route
15–60 minute service with limited stops
Frequent Route
15 minute or better service 7am-7pm, Mon-Fri
Local Route
20–120 minute service

Legend
Direction of Travel
Route Name
Transit Exchange
Park & Ride Lot
(no overnight parking)
Major Stop
Connect to UVic: Windows 7 and Vista

NEW UVic wireless configuration utility
The UVic wireless configuration utility will automatically configure the "UVic" wireless network on your Windows XP SP3, Windows Vista, or Windows 7 computer.

Download now

Note: The UVic wireless configuration utility is still experimental; use this application at your own risk. UVic is not responsible for any damage caused by the use of the wireless configuration utility. Please report any problems to the Computer Help Desk.

If the above doesn't work, please follow the manual instructions listed below. After the initial configuration, you should automatically connect to UVic (the secure wireless network) when you are on campus.

1. Before you start this procedure, ensure the following:
   - Your wireless card and its drivers have been installed and you have rebooted your laptop since the installation.
   - Your laptop is powered on and booted up.
   - You are in an area with wireless coverage.
   - You have a NetLink ID and password.
   - You are using Windows to manage your wireless connections. If you are using a third-party application (sometimes network adaptors come with their own applications), you may experience problems during the configuration process.

2. Temporarily connect to the Internet using UVicStart, an Ethernet port, or your home network. Download the security certificate by right clicking thawte Primary Root CA and saving the thawte.cer file to your computer. Once the file is saved to your computer, locate the file, double click on it, select Install Certificate..., and follow the Certificate Import Wizard instructions.
3. Once you have successfully installed the certificate, open your **Start** menu and click on **Control Panel**.

4. Click on **Network and Internet** or **Network and Sharing Center**.
5. Click on **Network and Sharing Center**.

6. Click on **Manage wireless networks**, located on the left menu.
7. Click **Add**.

8. Click **Manually create a network profile**.
9. Enter the following information:
   - Network name: **UVic** (case sensitive).
   - Security type: select **WPA2-Enterprise**.
   - Encryption type: automatically sets to **AES**.
   - Security Key/Passphrase: (leave blank).

   Ensure both checkboxes are selected (by default, the second box is not). Click **Next**.

10. Click **Change connection settings**. For now, ignore the pop-up window in the bottom-right corner.
On the Connection tab, ensure the Connect to a more preferred network if available checkbox is not checked.

11. Click the Security tab. Ensure the authentication method is PEAP. Then click Settings.

12. Check the box beside thawte Primary Root CA in the list of Trusted Root Certification Authorities.
If you cannot find the correct certificate listed, please return to step 2 to download the certificate.

At the bottom of the dialogue, ensure that the Authentication Method is Secured password (EAP-MSCHAP v2). Click Configure.

13. Deselect the checkbox for Automatically use my Windows logon... and click OK.

14. Close the remaining windows. In the bottom-right corner of your screen, you should see a small window pop-up informing you that Additional information is required to connect to UVic. Click on it to provide additional information.

15. Enter your personal NetLink ID followed by @uvic.ca in the User name field, and your NetLink ID password in the Password field. Click OK.
You should now be connected to the **UVic** secure wireless network.

<table>
<thead>
<tr>
<th>Related support</th>
<th>Related services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>How-to's</strong></td>
<td></td>
</tr>
<tr>
<td>AirNet wireless coverage</td>
<td></td>
</tr>
<tr>
<td><strong>Setups</strong></td>
<td></td>
</tr>
<tr>
<td>Connect to eduroam: HTC mobile device</td>
<td></td>
</tr>
<tr>
<td>Connect to eduroam: Mac OS X 10.5 or newer</td>
<td></td>
</tr>
<tr>
<td>Connect to eduroam: Windows 7</td>
<td></td>
</tr>
<tr>
<td>Connect to eduroam: Windows Vista</td>
<td></td>
</tr>
<tr>
<td>Connect to eduroam: Windows XP</td>
<td></td>
</tr>
<tr>
<td>Connect to UVic: Android version 4 and newer</td>
<td></td>
</tr>
<tr>
<td>Connect to UVic: iPhone or iPod Touch</td>
<td></td>
</tr>
<tr>
<td>Connect to UVic: OS 10.5 and newer</td>
<td></td>
</tr>
<tr>
<td>Connect to UVic: Windows 7 and Vista</td>
<td></td>
</tr>
<tr>
<td>Connect to UVic: Windows XP</td>
<td></td>
</tr>
<tr>
<td>Connect to UVic wireless</td>
<td></td>
</tr>
<tr>
<td>Connect to UVic Wireless: eduroam</td>
<td></td>
</tr>
<tr>
<td>Wireless Internet</td>
<td></td>
</tr>
</tbody>
</table>
Connect to UVic: Mac OS X 10.5 and newer

After the initial configuration, you should automatically connect to UVic (the secure wireless network) when you are using UVic's wireless network.

1. Before you start this procedure, ensure the following:
   - Your wireless card and its drivers have been installed and you have rebooted your laptop since the installation.
   - Your laptop is powered on and booted up.
   - You are in an area with wireless coverage.
   - You have a NetLink ID and password.

2. At the top-right corner of your screen there should be the AirPort icon (a semi-circle). If you do not see this icon, your AirPort card or AirPort software may not have been installed properly.

3. Click on the AirPort icon (it may be partially darkened) to reveal a menu. Ensure your AirPort is On.

4. Scroll down the AirPort menu and select Join Other Network ...

5. In the window that opens, enter the following information:
   - Network Name: UVic (case sensitive)
   - Security: WPA2-Enterprise
   - User Name: your NetLink ID
   - Password: your NetLink ID password
   - 802.1X: Automatic

Click Join.
6. If you see a message about Mac OS X wanting to access your Keychain, click **Always Allow**.

7. A **Verify Certificate** window will open saying that the certificate is not trusted.
   - Click **Show Certificate**.
   - Check the box that says **Always trust “sac1cled050...”** (the exact name may vary) and click **Continue**.
   - If you are prompted for your computer password, enter it and click **OK**.
You should now be connected to the UVic secure wireless network. To disconnect from the wireless network, click on the AirPort icon and click Turn Airport Off. Next time you connect to UVic, you should not need to enter any additional credentials.

### Related support

**How-tos**
- AirNet wireless coverage
**Setups**
- Connect to eduroam: HTC mobile device
- Connect to eduroam: Mac OS X 10.5 or newer
- Connect to eduroam: Windows 7
- Connect to eduroam: Windows Vista
- Connect to eduroam: Windows XP
- Connect to UVic: Android version 4 and newer
- Connect to UVic: iPhone or iPod Touch
- Connect to UVic: OS 10.5 and newer
- Connect to UVic: Windows 7 and Vista
- Connect to UVic: Windows XP
- Connect to UVic wireless
- Connect to UVic Wireless: eduroam
- Wireless Internet

---

University of Victoria - 3800 Finnerty Road - Victoria BC V8P 5C2 - Canada - Phone: 1-250-721-7211 - Fax: 1-250-721-7212
Connect to UVic: iPhone or iPod Touch

After the initial configuration, you should automatically connect to UVic (the secure wireless network) when you are using UVic's wireless network.

1. Before you start this procedure, ensure the following:
   - Your device is using firmware version 4.0 or higher.
   - Your device is powered on and booted up.
   - You are in an area with wireless coverage.
   - You have a NetLink ID and password.
2. From the Home screen, press the Settings button.
3. Press the Wi-Fi option.
4. Under the Choose a Network... heading, select UVic.
5. Enter your personal **NetLink ID** followed by @uvic.ca in the **Username** field. Enter your **NetLink ID password** in the **Password** field. Press **Join**.

6. If prompted, press **Accept** to verify the **thawte Primary Root CA** certificate.
Your device should now be connected to the UVic secure wireless network.
University Systems

Internet and telephone

Wireless Internet

Connect to UVic: Android version 4 and newer

Please note: Android devices do not fully support Exchange ActiveSync encryption so they are not recommended standards.

After the initial configuration, you should automatically connect to UVic (the secure wireless network) when you are using UVic's wireless network. Please note that University Systems only supports devices running Android version 4 or newer; all other devices are best-effort support only.

1. Before you start this procedure, ensure the following:
   - Your device is running version 4.0 or higher
   - Your device is powered on and booted up.
   - You are in an area with wireless coverage.
   - You have a NetLink ID and password.

2. Go into Settings.

3. Press Wi-Fi.
4. Select **UVic**

![Image of Wi-Fi settings on an Android device]

Wi-Fi networks

**UVicOpen**
Connected

**eduroam**
Saved, Secured

**UVic**
Secured

5099251212
Not in range

**BELL_WIFI**
Not in range

Add network

Scan | Advanced

5. In the window that opens, enter your UVic **NetLink ID** followed by @uvic.ca (e.g. helpdesk@uvic.ca) in the Identity field. Enter the corresponding NetLink ID **password** and press **Connect**.
6. Your device should now be connected to the UVic wireless network.
An SQL query walks into a bar and sees two tables. It saunters over to them and asks, “May I join you?”
Monday

9:30 – 12:00
• Instructor & Student Introductions
• Survey of course materials on the website
• Lecture 1: Why Databases?
• Begin Lecture 2: Intro to Database Design

1:30 – 4:00
• Worktime: Break-Out Groups to do design exercises
  -- Look at MySQL Tutorial PDF, p. 57.
  -- hint: use the “Data Normalization Principles” section from the Tutorial document to help focus on entities and attributes.
• Presentation of Break Out Group design exercises (max 5 mins each)

Tuesday

Read through the “Concepts & Strategies” section of Database Concepts (in the “Tutorial” folder), especially the two sections “Data Normalization Principles” and “Relationships.”

9:30 – 12:00
• Finish Lecture 2: Intro to Database Design
• Installation of MySQL
• Follow the “MySQL Installation Instructions” for your platform (Macintosh: pp. 80–82; Windows: pp. 83–85).
• Read “MySQL From the Command Line” to learn how to launch and use MySQL
• Read “Top 11 MySQL Power User Tips” for good hints
• Read “From Novice to Guru” to see where your path may take you

1:30 – 4:00
• Finish Installation of MySQL
• Code-Along: Initializing the birdclub database; basic queries:
  -- SELECT
  -- ORDER BY
  -- LIKE
• Worktime:
• Work through the Birdclub Tutorial Exercises
  -- refer to the “MySQL Functions” section in the Tutorial document to learn more about how to wield your data with more power and flexibility.
• Work on your own database designs

---

**Wednesday**

Read through the “Birdclub” section of Database Concepts (in the “Tutorial” folder) and, if you have time, move on to the other database examples.

9:30 – 12:00
• Lecture 3: Relationships
• Lecture 4: “You Say ‘Sequel,’ I Say ‘Ess-Que-Ell’: An Introduction to Structured Query Language”

1:30 – 4:00
• Codealong Queries:
  -- USE DATABASE
  -- MySQL Functions: CONCAT(), COUNT(), DATE_FORMAT(), etc.
  -- Multiple-Table queries
  -- GROUP BY
• Lecture 5: Data Types & CREATE TABLE
  -- DESCRIBE table
  -- ALTER TABLE (add active column to member; add primary key to sighting)
• Worktime: Design and build your databases.
  -- Work through the data normalization principles on your own project. Bounce ideas off your instructors whenever you want. When you're ready, start creating your database.

---

**Thursday**
Read through the “Multiple-Table Queries” and “Left Join” sections of *Database Concepts*. Continue working through the sample databases in the “Database Exercises” section.

9:30 – 12:00

- The **LOAD DATA** command: importing data from Excel and other sources
- Lecture 6: Spatial Metaphors in SQL
- Finding missing information: the **LEFT JOIN** command
- Load the MovieLens database and do some preliminary queries
- Adding an **INDEX** to speed up slow queries
- Views
- Worktime:
  -- Additional tutorial exercises (remember to check the answers in the back of the *Database Concepts* PDF document)
  -- build your database.

1:30 – 4:00

- Worktime: build your database

---

**Friday**

9:30 – 12:00

- Show and Tell:
  -- phpMyAdmin
  -- other projects the instructors might be working on
- Final Wrap-up
# Table of Contents

*(version 1.2)*

## Concepts & Strategies

- Introduction to Database Concepts ................................................................. 2
- Data Normalization Principles ......................................................................... 4
- Relationships .................................................................................................. 7
- Multiple Table Queries ..................................................................................... 16
- Left Joins: Using MySQL to Find Things That Don’t Exist ............................ 19
- Top 11 MySQL Power User Tips ...................................................................... 21
- From Novice to Guru ....................................................................................... 25

## Database Exercises

- Birdclub Scenario & Table Definitions ............................................................. 32
- Birdclub ........................................................................................................... 34
- 2012 NFL Prospects ...................................................................................... 39
- MovieLens ...................................................................................................... 43
- 1818-1819 Factory Workers .......................................................................... 50
- More Database Design Exercises ................................................................. 57

## Reference Materials

- MySQL Functions ........................................................................................... 60

## Data Input & Output

- The `LOAD DATA` Command ......................................................................... 66
- Other Bulk Loading Techniques ..................................................................... 74
- `mysqldump` .................................................................................................. 77

## Installing and Using MySQL

- Macintosh Installation Instructions ............................................................... 80
Windows Installation Instructions ................................................................. 83
Securing a New MySQL Installation .................................................................. 86
MySQL From the Command Line ....................................................................... 88

**Answers**

Birdclub Queries .............................................................................................. 94
2012 NFL Prospects Queries ............................................................................. 99
MovieLens Queries ............................................................................................ 102
1818-1819 Factory Workers Queries .................................................................. 104
Concepts & Strategies
Introduction to Database Concepts

Definitions & Concepts:

A database is a collection of data that is organized and stored according to some purpose.
The database is organized into tables (which look like HTML tables or spreadsheets). Each table stores data about some real-world entity.

Each table is organized into rows and columns.

Each row in the table is a record, which usually corresponds to a one real-world object (like an animal or a car or an archival document).

A record can contain several pieces of information. A column (sometimes called an attribute or a field) corresponds to one of those pieces of information. They are often adjectives.

A relational database (as opposed to a “flat-file database”) is very good at relating (that is, matching up) information stored in one table to information stored in another table by looking for elements common to each of them.

A primary key is a special column in a table whose values (perhaps numbers, perhaps names) are unique within that column. A primary key has three qualities: 1) it is unique across all records in the table; 2) it has a non-NULL value for each record in the table for the entire lifetime of the record; 3) its value never changes during the lifetime of the record.

A foreign key is a value in a table that is a primary key from a different table. Foreign keys need not be unique. Foreign keys make relational databases possible.

General Design Principles:

1. Store only one piece of information in each field. What counts as “one piece of information” will depend on the purpose of your database. Generally speaking, though, no groups, lists, or comma-separated values!

2. Store each piece of data in only one place. If you’re finding the same name or value typed out again and again in different places, then you need to create a new table, store that data there, assign it a primary key, and use foreign keys elsewhere in the database.

3. Do not store anything that you can calculate. Do not store dynamically changing data, such as the number of days until you graduate or someone’s age; store a birthday instead.

4. Design tables so that adding new information creates new rows (records), not new columns.

5. Design tables to contain as few NULL values as possible. If you notice lots of NULL values (or even lots of repeated values) your table probably needs further normalization.
Data Modeling Strategies:

1. **Real-world entities**, which may range from the concrete (baseball cards, pets, contracts) to the abstract (friends, genres, rankings), become tables in the database. Entities are often nouns.

2. An entity’s attributes (or qualities or properties or identifiers) become columns in its table. Attributes are often adjectives. Choose an appropriate data type for each column.

3. **Specific instances** of an entity become rows or records in the table. Whereas person is an entity and therefore signifies a table, Bob and Sue and Jim are specific instances and so are represented by individual records in the table.

4. Unique attributes (identifiers) become **primary keys**. Interestingly, most entities do not naturally have unique identifiers (there are a lot of Jim Smiths in the world, for example), so most of the time we just assign arbitrary sequences of numbers as primary keys. Sometimes users are fully aware of their primary keys (student numbers, social insurance or social security numbers, etc.), but many times users never know their primary keys. Primary keys are not always public information.

5. When stored in another table, a primary key is known as a **foreign key**. Relationships in the database are modeled using foreign keys.
Data Normalization Principles

The classic definition of a database is that it's a collection of data that is organized and stored according to some purpose. That sounds pretty vague and you might think that a definition like that opens the door for a chaotic data-driven free-for-all. Not so, in practice.

There are indeed some best practices when it comes to organizing data in a database. And we have Raymond F. Boyce and Edgar F. Codd to thank for that. The system they developed is sometimes called Boyce-Codd Normal Form or BCNF for short. (Knowing that is a great ice-breaker at parties. Try it. You’ll see.)

Those principles are actually defined in mathematical terms, though. Knowing that fact might come in handy if you’re ever captured by space alien geniuses and your freedom hinges on your ability to provide a demonstrable proof that SQL works and that any well-structured query is indeed guaranteed to give you the right answer. If you’re just learning databases, though, the technical definitions of data normalization forms don’t help at all.

So here’s the shortcut. Quamen’s over-simplified and metaphoric approach to data normalization:

Step 1. Nouns
Step 2. Relationships
Step 3. Adjectives

Step 1. Create a separate table for each entity in your data.

Entities are usually real-world objects like birds, employees, cars, concerts, contracts, or events. Any important noun in a prose description of your dataset is probably an entity and should therefore get its own table.

Achieving so-called first normal form (1NF) really involves striving for these three goals:

1. There are no duplicated rows in the table. Each concrete instance of your entity occupies one and only one row. Each row should have some kind of unique identifier, which we call a primary key. Sometimes the dataset provides us with a good primary key but, if not, we can always invent one (usually an arbitrary but linear sequence of numbers).

2. Each cell (or field) contains only one value. There should no groups of data or comma-separated lists of data in any given cell.

3. Any given column contains the same kind of data. For example, avoid illicit mixing in one column of phone numbers and email addresses. Those should be two separate columns.
Step 2. Move to new tables any repetitive information in your existing entity tables.

After you’ve created an entity table, take it out for a test drive by filling it with some sample data. Look for information that gets repeated from row to row—data bits like addresses, course titles, names of bosses or instructors, dinner courses, authors of books, museum or archive names, event venues, movie genres, university names, information categories, building names, etc. These repeated bits of information should now be moved into their own tables. Often, they represent “sub-entities” that we simply didn't recognize as being important in Step 1. That's OK. That’s why we have a Step 2.

More examples: Shakespeare's plays are often classified as comedies, histories or tragedies. Many different movies were all directed by Martin Scorsese. An interesting group of countries are all located in Europe. Animals are classified into reptiles, amphibians, birds, mammals. Your personal library probably has many books by the same publisher.

Moving repetitive information into new tables should suggest to you that those new tables need to go through the data normalization process as well. Repeat the process on those tables: ensure row and column consistency, guarantee row uniqueness with a primary key, etc.

And in order to maintain the relationship between the old, original entity and this new one that you’ve just created, you should store the new table's primary key back in the original table as a foreign key. And, for me, that's the gist of second normal form (2NF): the creation of relationships between entities. Databases model relationships with a system of primary keys and foreign keys.

Data relationships are classically divided into three varieties: one-to-one, one-to-many, and many-to-many. The more you know about modeling data relationships, the easier 2NF becomes. There are enough subtleties about relationships, though, that they warrant their own discussion. Check out the section on “Relationships” in this course packet for more details.

Step 3. Make sure that all table columns are fully dependent upon the table's primary key.

This rule is really about the consistency and integrity of data. The relationship between any given column and its table should be analogous to the relationship between an adjective and its noun. Any columns in your tables that don't contribute like adjectives probably need to be moved elsewhere. Any columns whose values could “go stale” over time (that is to say, that simply become incorrect over time) should be reconceptualized.

Here are a few common litmus tests:

1. If we delete a row, will we lose any data that other records might need?

If (or when) we delete Paris Hilton's autobiography from the library database, will we also lose the fact that autobiographies are housed in Johnson Library? If we
delete dinner course #2 from the meal database, do we also lose track of which one is the salad fork? When Babe Ruth got traded to the Yankees, did we also accidentally lose the fact that the Red Sox play in Boston?

2. If we add a row, could we accidentally make any data entry mistakes that would compromise our database’s integrity?

When we inevitably add Paris Hilton’s autobiography back into the library database, could we accidentally record the mistake that autobiographies are housed in Campbell Library instead of Johnson Library? When we add a new course to the ambassador’s fancy dinner party, do we need to know which one is the salad fork? When we logged the Babe Ruth trade, did we mistakenly assign him to Chicago because misspelling the word *socks* is weird enough that surely only one team would do it?

3. Have we stored any values that we should calculate instead?

Don’t store someone’s age; store a birthdate. We can always calculate a person’s age on-the-fly when we need to know it. Don’t store the average price of bananas; store all the prices of bananas and then calculate today’s average price. Don’t store the highest and lowest test scores; calculate them. Don’t store the number of baseball trades so far this year; count them.

If the answer to any of those litmus test questions is “yes,” then you should either move the problematic information to a new table or else (in the case that your data can accidentally go stale) try to decide how to store it in such a way that it doesn’t implicitly have a “best before” date stamped on its forehead.

Reading the technical details about data normalization can turn you into a shark—you feel that you have to keep moving or else your eyes will roll back in your head and you’ll die from lack of oxygen. That’s because data normalization is, in its most fundamental form, about mathematical proof. And most of us don’t care about the mathematics or the implicit guarantees about truthiness that math can provide.

Rather, we’re interested in the heuristics—rules of thumb—and the “best practices” about how to create database tables effectively. The guidelines above simply translate mathematical concepts like functional and transitive dependencies into more metaphoric terms like relationships and adjectives.

Data normalization—especially the 2NF step—gets easier when you know more about how databases handle relationships between tables. Read on, MacDuff.
Relationships

synonyms: connected, associated, linked, coupled, allied, affiliated, corresponding, kindred, parallel, be relevant to, pertain to, concerned with, having a bearing on, appertain to, involved with, pertinent to, have a rapport with, identify with . . .

Database people talk about relationships between entities as if they’re obvious and self-evident. And maybe they are. But when I’m working on a new database design, I inevitably run through a list of synonyms for the term “relationship” as I try to wrangle my data into patterns.

The heart of a database relationship is that two different real-world entities are somehow linked or affiliated with one another. British Columbia is located in Canada, even though the province and the country are different entities and either can be discussed without reference to the other. John Lennon still has a connection to the Beatles even though neither entity, unfortunately, has survived. No matter how you say it, Uranus is still funny.

Despite the complexity of real-world relationships, though, databases model relationships as a connection between two tables using primary and foreign keys. That simplicity is perhaps why database people think relationships are self-evident. Nonetheless, database designers classify the relationship between entities into three types:

<table>
<thead>
<tr>
<th>relationship</th>
<th>description</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-to-one</td>
<td>one X has only one Y</td>
<td>• a person has one social insurance number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• a building has one address</td>
</tr>
<tr>
<td>one-to-many</td>
<td>one X has many Y’s</td>
<td>• a country has many cities</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• a student takes many classes</td>
</tr>
<tr>
<td>many-to-many</td>
<td>many X’s have many Y’s</td>
<td>• many different bands play at many different venues</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• many mythbusters bust many myths</td>
</tr>
</tbody>
</table>

One-to-One Relationships

A one-to-one relationship says that each entity of type X is connected to only one entity of type Y. For example, each person has one social insurance (or “social security”) number. Each student at a university has one identification number. Each class at the university has only one section identifier. Each automobile has one license plate number.
If we were to model that in a database, we'd draw a picture like this:

Here, each rectangle represents a table and the horizontal line represents a relationship between them. It's a single line—it doesn't branch or divide—so we will interpret that type of line as a one relationship. Database people would say that its cardinality is one. Here, the line is one on both sides, and so this diagram suggests that the relationship between class and section number is one-to-one.

In the database, we'd model that relationship by taking the primary key from one of the entities (say, section_number) and putting it into the class table as a foreign key. And anytime we want to discover the section number of any given class, we'll retrieve the foreign key, look across to the section_number table, and learn the section number.

Here's the dilemma. There's no reason why we couldn't simply store the section number directly in the class table. There's no need here for a relationship at all. The introduction of foreign keys unnecessarily complicates an otherwise simple and straightforward table design. So the best practice in this case is to remove (the SQL verb is to drop) the section_number table entirely and move its data into the class table.

```
mysql> DROP TABLE section_number;
```

The table below is not only a better solution to this particular problem, but it's the best way to handle almost all one-to-one relationships. I'm not prepared to say that distributing a one-to-one relationship across multiple tables is always wrong, but it's uncommon enough that if you're tempted to do it, you should sit down over a nice beverage and contemplate the maneuver.

Our revised table looks like this:
### One-to-Many Relationships

A one-to-many relationship says that each entity of type X has many entities of type Y. A country has many cities. A store sells many products. A person has many ancestors. When diagramming a one-to-many relationship, we use a line that branches on the end, sometimes called a “crow’s foot.” The crow’s foot signifies the **many** side of a relationship:

![Diagram of one-to-many relationship]

To read one of these diagrams, we read across from the table name to the shape of the line on the opposite end. It’s not obvious, but we ignore shape of the line at its beginning (here, where it connects to the `country` table). Going left to right, then, we read, “a country has many cities”:

![Diagram reading from left to right]

In the other direction, going from right to left, again we ignore the shape of the line at its beginning. We read the name of the table and read leftwards to the shape of the line at its finish: “a city has one country”:

![Diagram reading from right to left]

Using the diagram, we can now build tables to hold that data. One rule of thumb when building tables from these kinds of diagrams is that **the foreign key gets stored on the crow’s foot side of the relationship.**
**relationship.** Here, that means that we’ll take the primary key from the **country** table and stash it into the **city** table as a foreign key:

<table>
<thead>
<tr>
<th>code</th>
<th>country</th>
<th>id</th>
<th>city</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>Canada</td>
<td>1</td>
<td>Avignon</td>
<td>FR</td>
</tr>
<tr>
<td>US</td>
<td>United States</td>
<td>2</td>
<td>Paris</td>
<td>FR</td>
</tr>
<tr>
<td>FR</td>
<td>France</td>
<td>3</td>
<td>Marseilles</td>
<td>FR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>Victoria</td>
<td>CA</td>
</tr>
</tbody>
</table>

Although it’s common to use a sequence of integers as primary keys, in this case I’ve used the two-letter country codes from the International Organization for Standardization (http://www.iso.org/iso/country_codes/iso_3166_code_lists.htm). These two-letter codes are guaranteed to be unique for all countries in the world, so they work perfectly well as primary keys.

### Many-to-Many Relationships

In order to elaborate all the baroque subtleties of the many-to-many relationship, I’d like to work through three illuminating examples: 1) Many mythbusters bust many myths; 2) A pint of beer is a many-to-many relationship; and 3) a thesaurus.

**Example 1: Many mythbusters bust many myths.**

The popular Discovery Channel show, *Mythbusters*—hosted by Jamie Hyneman, Adam Savage, Tory Belling, Kari Byron and Grant Imahara—puts urban myths to the test on a weekly basis. The hosts generally split into two teams to do their busting and so no one mythbuster ever works solo. Sometimes one episode busts several myths and so we have here the classic structure of a many-to-many relationship: many mythbusters bust many myths and many myths are busted by many mythbusters.

Let’s build some tables and follow our one-to-many strategy of stashing foreign keys on the **many** side. Since we have here two tables with **many** relationships, we’ll just pick one at random:
Hmmm. Not good. Since every mythbuster has worked on so many myths, we end up with a column that's just a comma-separated list of myths busted. We have the same problem even if we flip the responsibility of holding the foreign keys to the other table:

Clearly, the rule of thumb that got us through the one-to-many examples is failing. Or is it? What if we could put another table in the middle? Wouldn't we break the many-to-many down into two one-to-many relationships? Yes, we would:

That middle table is called a junction table and its role is to help manage many-to-many relationships. In its simplest form, it's merely a 2-column stash of foreign keys (remember the rule: put the foreign key where the crow's foot is) but that's all we need in order to create two one-to-many relationships. In the process, it allows us to convert our comma-separated lists into individual table rows, which is one of the goals of data normalization.
In this case, our junction table does not represent any recognizable real-world entity, but it still needs a name. One common strategy is to combine the names of the two tables it's joining and so I've called this junction table buster_myth.

You can call it anything you want, but just calling it junction_table is less descriptive than you'd want, especially if your database ends up with a bunch of these. Trying to decipher the difference between junction_table_1, junction_table_2 and junction_table_3 isn't something you want to do late at night.

**Example 2: A pint of beer is a many-to-many relationship.**

The junction table in our last example did not correspond to a real-world entity, but that's not always the case. In fact, that's not even the case most of the time. We see entities all around us every day that function as real-world junction tables. A pint of beer is just such a thing.

Let's imagine that Arthur, John and Georg hit the pub. Arthur likes stouts, John prefers British bitters and Georg opts for German wheat beers. As the evening progresses, the gentlemen's minds expand, and they begin trying each other's brews. The result is a classic many-to-many relationship: many tipplers try many beers, and many beers were tried by many tipplers.

But because a pint is a real-world entity, we can store more information in our table than just foreign keys. Of course, the pint table still functions as a junction table, but its real-world existence means that we will probably think of it more as an entity than as a junction table.

Here's what our tables might look like:
We could store much more information on our **pint** table: the pub, the date and time, a beer review. The **pint** table transcends its lowly role as a junction table. It might even become the centrepiece of the whole database, containing more data and more columns than all the other tables combined.

Real-world entities act as junction tables all the time. A restaurant review manages a many-to-many relationship between restaurants and critics. A bird sighting table identifies a many-to-many relationship between people and birds. A movie enacts a many-to-many relationship between actors and directors.

**Example 3: A thesaurus articulates a many-to-many relationship between words and other words.**

The classic many-to-many relationship elaborates a relationship between three tables in the database. But in certain circumstances, a junction table can manage a many-to-many relationship with only one other table. A thesaurus is a good example. The thesaurus plucks a word from the dictionary (*relationship*, for instance) and points to other words elsewhere in the dictionary that mean roughly the same thing (*connected, associated, linked, coupled, allied, affiliated*, etc.)
One of the interesting decisions to make here is whether the relationship between words and their synonyms are one-way (“asymmetrical”) or two-way (“symmetrical”). It makes sense that if affiliated is a synonym for relationship, then relationship must automatically be a synonym for affiliated too. That’s a symmetrical relationship. In other words, the database designer needs to determine whether the one row (7, 1) is sufficient to cover both cases or whether the reverse, (1, 7), needs to be entered as well.

The best decision is probably to let the relationship work both ways. That makes the table smaller—half the size it would be otherwise, obviously. But that decision introduces new problems as well: any attempt to INSERT the pair (1, 7) into thesaurus should fail because (7, 1) is already there. And there’s not a wholly elegant way to solve that problem.

To begin, though, you could add a multi-column UNIQUE index to the thesaurus table, but that prevents only new entries that are in the same order as a row already existing in the table:

```
mysql> ALTER TABLE thesaurus ADD UNIQUE( word, synonym );
mysql> INSERT INTO thesaurus VALUES (7, 1);
ERROR 1062 (23000): Duplicate entry '7-1' for key 'word'
```

That works perfectly well. Unfortunately, you can still enter those values if you reverse the order:

```
mysql> INSERT INTO thesaurus VALUES (1, 7);
Query OK, 1 row affected (0.00 sec)
```

You could solve this dilemma by always sorting the two foreign keys so that the smallest number goes into the lefthand column and the larger number goes into the righthand column. (We can discount cases where the two numbers are the same because we probably don’t want to log the fact that a word is its own synonym.) Sorting the numbers is a bit of extra work (it’s a strategy used in the self join query in the Factory Workers dataset too), but in tandem with the UNIQUE index, sorting the values before INSERTing would successfully prevent multiple entries in the table.
Relationships Redux

The topic of determining relationships between database tables is, I think, under-represented in the database literature and in database tutorials. I suspect that's because relationships seem to be so intimately connected with the data itself that writers and teachers assume that nothing useful can be said. But I don't think that's true. Relationships occur in fairly regular, repeating patterns and—as we've seen here—most of them resolve into one-to-many relationships anyway.

Even those elusive many-to-many relationships are built up from one-to-many relationships. Identifying them hinges on being able to see patterns between three different tables, some of which may or may not actually correspond to real-world entities.

Regardless, relationships are the heart and soul of relational databases and learning to see relational patterns amid the chaos of data is one of the most important skills you can develop as a database designer.
Multiple Table Queries

The chart above lets me easily construct the following query:

```sql
mysql> SELECT member.first, member.last (fields to select)
    -> FROM member, sighting (from these tables)
    -> WHERE member.id = sighting.person; (following these keys)
```

That query yields the member names associated with every bird sighting in the table. And when I know that the query works properly, I can refine it to give me variations of data:

**Variation #1: Concatenation**

```sql
mysql> SELECT CONCAT(member.first, ' ', member.last) (fields to select)
    -> FROM member, sighting (from these tables)
    -> WHERE sighting.person = member.id; (following these keys)
```

**Variation #2: Distinct Records**

Or I could use the `DISTINCT()` function, which will eliminate the duplicates:

```sql
mysql> SELECT DISTINCT(concat(member.first, ' ', member.last)) (fields to select)
    -> FROM member, sighting (from these tables)
    -> WHERE sighting.person = member.id; (following these keys)
```

**Variation #3: Formatting**

I can even add column aliasing (`...AS name`), date formatting, date parsing like `YEAR()` and `MONTH()`, aggregate functions like `COUNT()`, and interesting groupings to give me different results:
mysql> SELECT CONCAT(member.first, ' ', member.last) AS name, 
       -> DATE_FORMAT(sighting.date, '%M %Y') AS month, 
       -> COUNT(sighting.person) AS total 
       -> FROM sighting, member 
       -> WHERE member.id = sighting.person 
       -> GROUP BY year(sighting.date), 
       -> MONTH(sighting.date), sighting.person;

Variation #4 (also select the bird’s common name):

mysql> SELECT CONCAT(member.first, ' ', member.last) AS name, 
       -> bird.common 
       -> FROM bird, member, sighting 
       -> WHERE sighting.person = member.id 
       -> AND sighting.bird = bird.id;

Write SQL queries to find the answers to the following:

1. Which guides do not have an email address? (Hint: not a left join!)
2. List the phone numbers of every member who has seen a Meadowlark.
3. Which guides have seen a purple finch?
4. How many total birds have been seen by members who have an email at “blah.ca”?

Go ahead, generate an error:

Try these incorrect INSERT queries just to see the error messages that MySQL generates:

mysql> INSERT INTO sighting VALUES (1, NULL, '2012-09-28');
mysql> INSERT INTO sighting (1, NULL, '2012-09-28');

Why do they generate different error messages?

Write INSERT queries to add the following sightings:

(Note: go ahead and pick your favourite year for these.)

George McFly, Rusty Blackbird, April 4 (pick a year)
Deb Jones, House Sparrow, 10 July (pick a year)
Bob Smith, Brewer’s Sparrow, 9 October (pick a year)

Write INSERT queries to add the following people to the member table:

Larry Smith, smithy@shaw.ca, no phone
Pete O’Malley, pom@shaw.ca, 780-555-9999
Sarah D. Ferguson, fergy@royalty.uk, 780-555-0011

A Random Note on SELECT Query Optimization:

People who optimize queries for a living recommend starting with the table having the fewest records. The first table listed in the WHERE clause of the query is often called the “driving table,” so the strategy to optimizing a query is to choose the best driving table, start with it, and then work your way towards the bigger tables.

Another strategy is to create indexes on the columns used most often in WHERE clauses. Primary keys and unique columns are automatically indexed, but you can create indexes on other columns as well. Don’t index everything, though — at some point, MySQL will spend more time updating the indexes than running your queries. Check out the MovieLens Dataset for more strategies about indexing tables.
I want to solve this problem:

I want to find data on every member of the Birdwatching Club who is not a guide. In other words, I want to find all data from people who do NOT have their ID listed in the guide table.

A Note on Left Joins:

“When faced with the problem of finding values in one table that have no match in (or that are missing from) another table, you should get in the habit of thinking, ‘aha, that’s a LEFT JOIN problem.’ A LEFT JOIN is similar to a regular join in that it attempts to match rows in the first (left) table with the rows in the second (right) table. But in addition, if a left table row has no match in the right table, a LEFT JOIN still produces a row—one in which all the columns from the right table are set to NULL. This means that you can find values that are missing from the right table by looking for NULL.”


First Try This (totally wrong):

My first strategy is simply to look for rows in which the ID from member and the ID from guide are not the same (notice that != means “is not equal to”):

```
mysql> SELECT * FROM member, guide
    -> WHERE member.id != guide.id;
```

It’s a logical strategy, but it doesn’t work. Notice that I get a huge result set! The query has produced a result set that includes *every possible combination of the data in which the two IDs are not equal*. It’s like those old combinatoric problems from high school math tests: “how many seating arrangements can you have with 7 dinner guests if Lisa can’t sit next to Charlie?” This isn’t what I wanted.

Next try this (partial solution):

```
mysql> SELECT member.*, guide.id AS 'gid' FROM member
    -> LEFT JOIN guide
    -> ON member.id = guide.id;
```
This is the **LEFT JOIN** that DuBois was talking about. Notice the two columns `id` and `gid` ("guide id") in the result set. The `id` column is full of all the members, as you'd expect. Note that the `gid` column has an id only for members who exist in the guide table. If the person doesn't exist in the guide table, the value here is **NULL**. Excellent! That's because `member` was the "left table" and `guide` was the "right table" in my `SELECT` query. As DuBois said, “if a left table row has no match in the right table, a LEFT JOIN still produces a row—one in which all the columns from the right table are set to NULL.” My right table has only one column, but that's OK. This is the clue I needed to get what I want.

**Finally try this (complete solution):**

Now that I know a **LEFT JOIN** will produce a result set that has strategically placed **NULL** values, I can select data on members who are not guides simply by selecting columns where `guide.id` is **NULL**. In this version of the query, I'll omit `guide.id` from my `SELECT` statement; that column will contain only **NULL** values anyway, so I don't really need to see it. The `guide.id` does have to be present, however, in my `WHERE` clause (otherwise I wouldn't get the right records, and that's the whole point!):

```
mysql> SELECT member.* FROM member
  -> left join guide
  -> ON member.id = guide.id
  -> WHERE guide.id IS NULL;
```

Now I've got what I want: all the data on club members who are **not** guides.

**On Your Own:**

Write a **LEFT JOIN** query to list all the birds that have not been seen by any club members.
Write a **LEFT JOIN** query to list all the bird club members who have not seen any birds.

**Answers:**

```
mysql> SELECT bird.* FROM bird
  -> LEFT JOIN sighting
  -> ON bird.id = sighting.bird
  -> WHERE sighting.bird IS NULL;

mysql> SELECT member.* FROM member
  -> LEFT JOIN sighting
  -> ON member.id = sighting.person
  -> WHERE sighting.person IS NULL;
```
Top 11 MySQL Power User Tips

1. Use \q or quit or exit to quit MySQL.

There are several ways to quit MySQL. The three easiest are to type either \q or quit or exit at the command line prompt, any of which will return you to your operating system’s command-line prompt.

```
mysql> \q
Bye
```

You can also just close the command-line window. That action automatically terminates anything you had running.

2. Use \h or \? or help to see a list of available commands.

MySQL has a built-in help system, but its content is limited to the series of backslash commands available to you (that is, it doesn’t help you with your SELECT or INSERT data manipulation commands—you’ll have to go online or to a reference book for that kind of help). You can see the list and a short description of each command by typing \h or \? or help at the command line.

3. Use the up and down arrow keys to scroll through your MySQL command history.

MySQL keeps track of every command you perform and you can scroll through that list using the up and down arrow keys on your keyboard. Did you just make one little mistake in a very long query? Hit the up arrow to go back to it and use the backward/forward arrows to scroll through the query to edit the typo. Just hit return from any point in the line and MySQL will re-execute the command.

This works even across sessions. Do you need to repeat that giant 6-table query you did yesterday? Don’t worry. It’s still in your history. Keep hitting the up arrow until you find it.

4. Use tab completion.

If you type only the first one or two letters from a table name or a column name and then hit the tab key, MySQL will try to complete the rest of the name for you automatically so that you don’t need to type it all. For example if I have an incredibly_long_ridiculous_column_name, I can type only part of the name and hit tab to have MySQL complete it for me:

```
mysql> SELECT inc[tab]
```

At which point, MySQL finishes the column name for me:

```
mysql> SELECT incredibly_long_ridiculous_column_name
```

And then I can finish typing the rest of the query:
5. If tab completion doesn’t work, rehash.

MySQL stores data for auto-completion in what’s called a hash table. Usually, the hash table is read from memory only when MySQL starts up, so if you’ve added new tables or columns during your current session, the hash table is likely out of date. If you find that tab auto-completion isn’t working properly, you can always force MySQL to reload the hash table with the \# command (the # character is often called the “hash” character by computer geeks):

```sql
mysql> \#
```

To ramp up your geekitude, then, you should read that as “slash hash.”

6. Terminate commands with \G instead of a semi-colon to see output in a vertical format.

Trivia: \g is synonymous with the semi-colon. Although the symbolism of \g and \G dates from prehistoric times, iconography on cave walls in France suggests that the g probably once stood for go. Sometimes, \G is referred to as “extended go” or, for short, “ego.”

Vertical format is useful for tables that have so many columns that they scroll around to the next line when you display them horizontally. It’s also useful for tables that contain longer passages of text.

7. Use \c to cancel a command.

Any MySQL command can be canceled by typing \c and hitting enter (or return). You can do this from anywhere in the command. For example:

```sql
mysql> SELECT * FROM memb\c
mysql>
```

MySQL dutifully cancels the SELECT command and just gives me a fresh prompt.

This trick needs a slight modification if you’re in the middle of a quoted string. In that case, MySQL needs you to terminate the string first and then it will let you cancel the command.

When you forget to terminate a string, MySQL’s default second-line prompt will change. For example, notice here that I forgot to add the closing quotation mark after Bob’s name:

```sql
mysql> SELECT * FROM member WHERE name = 'Bob;
>
```

MySQL’s second-line prompt now includes a quotation mark, which is MySQL’s way of reminding me that I forgot to terminate a string. If I want to cancel a command that has this particular mistake, I have to terminate the string first before I can use \c. Here’s the whole sequence:
8. Use zero and NULL appropriately.

Remember, whereas NULL means the absence of a value, zero is, on the other hand, a substantive value that simply says the quantity is none.

So if I have zero pets, my favourite pet's name is NULL.

If the grocery store is out of bananas, the quantity is zero.

If Bill doesn't have a middle name, that field is NULL.

The warehouse quantity of a part the company still stocks may be zero, but the warehouse quantity of a discontinued part should probably be NULL.

9. Zero fill the missing parts of dates.

This practice often throws people for a loop, but it's both legitimate and encouraged in database groups to put zeroes into the missing parts of dates. The SQL standardized date format is YYYY-MM-DD and—perhaps counter-intuitively—MySQL still considers a date legal if any or all of that is zero. So if I know something happened in July 1867, but I don't know on which day, I can represent the date like this:

1867-07-00

And if a letter is dated March 9, but I don't know which year, I represent the date this way:

0000-03-09

One terrible but common strategy is to assign unknown dates to Christmas or to New Year's Day under the presumption that nothing happens on those days anyway. That practice should be discouraged. Zero-filled dates were invented to solve exactly this problem. If someone was born in 1923 but we can't be any more specific than that, then the date should be recorded as

1923-00-00

You can store only parts of dates, too: year is a valid column type, for example.

Zero-filled dates are not common practice in XML tagging communities, by the way. Their approach is not to “add” any information that isn't already present, and so the Text Encoding Initiative, for example, won't validate a normalized date that has zero-filled parts. You can always extend or modify your XML schema to allow zero-filled dates, though.

10. Use mysqldump to back up or to move your database.
Every installation of MySQL comes with a utility program called **mysqldump** that you can run from the command line. It exports database data into a plain text file that's highly portable and easily reloadable. All our database initializations are accomplished merely by loading a **mysqldump** file.

The program has many different options, so I encourage you to look up all the possibilities in a MySQL reference book or online. You can export some or all databases, some or all tables, with or without data, with or without CREATE statements, etc.

If you’ve taken on the role of a database administrator, check into a program called **cron**. That program is a scheduler (it’s a misspelled spoof on **chronology**), and you can tell **cron** to run **mysqldump** every morning at, say, 3:17 a.m. and to copy the database backup file to a secure location.

**cron** is available on Unix, Mac and Linux. There's a version of **cron** for Windows, but Windows folks might simply want to look into the **at** command instead.

11. **It's easier to store a filename than it is to store images, videos, or audio files.**

Although it’s possible to store different kinds of files in a database (check out the BLOB column type), it’s often easier to store just the filename. Let’s say that you’re running a social networking site and your users upload image files to use as their avatars. Your programmer can write some code to put those image files in a directory, say, `/users/images/`, which makes your job easier because you can just store the filename:

```
+------+-------------------------+
| id   | filename                |
+------+-------------------------+
| 1    | /users/images/kevin.jpg |
+------+-------------------------+
```

Now you know where to fetch Kevin’s avatar from.

In archival applications, sometimes data files can’t be revealed to the public due to copyright issues or other restrictions. You can still store files, but you can append a field to your database to determine whether those files can be released to the public:

```
+-------------+--------------------------------+------------------------+
| id          | filename                        | publicly_available     |
+-------------+--------------------------------+------------------------+
| 1           | /archive/manuscripts/novel.xml  | no                     |
+-------------+--------------------------------+------------------------+
```
From Novice to Guru

There are a number of issues you’ll have to come to terms with on your path towards total database enlightenment. Listed below are some of the common issues—in no particular order—that new users should learn. This section is good for casual browsing. It’s full of things that make you say, “Ah, interesting.”

Strings vs. Numbers

In MySQL, a number is a kind of “stand-alone” entity like 467 or 3.14 or 13. MySQL will assume that a series of digits (with or without a decimal point) should be interpreted as a number.

However, a string is different. A string is an ordered sequence of characters that cannot be altered or reduced in any way. A name is a string of characters, for example.

Sometimes, however, things that look like numbers actually aren’t. For example, a phone number looks like a series of digits, but if I try to SELECT a phone number, MySQL interprets my actions as a mathematical operation:

```
mysql> SELECT 555-1873;
+----------+
| 555-1873 |
+----------+
| -1318    |
+----------+
1 row in set (0.04 sec)
```

To avoid confusion, all strings in MySQL should be delimited with quotation marks. It doesn’t matter if you use single quotation marks or doubles, but you always need to tell MySQL where your string begins and ends.

```
mysql> SELECT '555-1873';
+----------+
| 555-1873 |
+----------+
| 555-1873 |
+----------+
1 row in set (0.00 sec)
```

Mathematical Operators

As the last example showed, you can perform mathematical calculations in MySQL, and the database uses the mathematical operators common to computer programming languages. If you’ve never seen them before, here they are:
<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
</tbody>
</table>

You can use MySQL as a calculator:

```
mysql> SELECT 13 * 8 - 45;
+-------------+
| 13 * 8 - 45 |
+-------------+
|          59  |
+-------------+
1 row in set (0.05 sec)
```

Null vs. Zero

Reiterated throughout this course packet is the idea that NULL and zero do not mean the same thing.

- **NULL** represents the absence of a value. NULL should be used for missing data.
- Zero means that a definitive value exists and it just so happens to be none.

For example, in a dataset about farmers, a zero in a column called cows would signify that we know definitively that the farmer owned no cows. A NULL value in that column, on the other hand, would suggest that any information about this farmer’s cow ownership is missing and that we simply don’t know whether the farmer owned cows or not.

Other synonymous expressions for NULL might include “no value,” “unknown value,” “missing value,” “out of range,” “not applicable,” or “none of the above.”

See the 1818-1819 Factory Workers dataset for an example of the proper way to use NULL and zero.

**NB:** the MySQL keyword NULL is not the same as the string 'NULL'; the first represents the absence of a value whereas the latter represents the 4-character string N-U-L-L. Don’t accidentally turn NULL into a string by wrapping it in quotation marks.
Keywords: IS, IS NOT

**NULL** is a bit of a philosophical conundrum in that it’s impossible to tell whether two missing values are the same or not. Technically, then, in MySQL the expression `NULL = NULL` is meaningless and is neither true nor false. It cannot be evaluated.

Fortunately, MySQL provides us the **IS** and **IS NOT** keywords, which help us work with **NULL** values. For example:

```
mysql> SELECT * FROM farmer WHERE cows IS NULL;
```

```
mysql> SELECT * FROM farmer WHERE cows IS NOT NULL;
```

Both of which are different from this query:

```
mysql> SELECT * FROM farmer WHERE cows = 0;
```

Metacharacters: * and % and _

When working with MySQL tables, there are certain wildcard characters that can be quite useful. The most common is *, which colloquially means *everything*. The * metacharacter only stands in for columns and rows, however.

When working with string values, two useful metacharacters are % and _. The difference is that % means *any sequence of characters no matter how many*, but _ means *only one character no matter what it is*.

<table>
<thead>
<tr>
<th>metacharacter</th>
<th>functions on...</th>
<th>means...</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>columns, rows</td>
<td>everything</td>
</tr>
<tr>
<td>%</td>
<td>characters</td>
<td>any sequence of characters, no matter how many</td>
</tr>
<tr>
<td>_</td>
<td>characters</td>
<td>one character</td>
</tr>
</tbody>
</table>

Keywords: LIKE, NOT LIKE

Remember that string comparisons in MySQL are always case insensitive (that is to say, capital letters and lowercase letters are indistinguishable), and so therefore the character pattern `shake%` matches a variety of strings like *Shakespeare* and *shakES* and *sHAKe7* and *Shake_Rattle_and_Roll*. None of those strings are equal, however, and so when doing string
comparisons with wildcard characters the equals sign is the wrong tool to use. Instead, to find strings that match wildcard patterns we need to use the LIKE and NOT LIKE keyword phrases.

Remember, too, that string values always need to be delimited with quotation marks so that MySQL knows where the string begins and where it ends. The quotation marks are not part of the string itself and so they also function as metacharacters, although they do not affect the pattern for which we're searching.

```sql
mysql> SELECT nickname FROM user WHERE nickname LIKE 'shake%';

mysql> SELECT nickname FROM user WHERE nickname NOT LIKE 'shake%';
```

Given an imaginary table full of words, we might do this Scrabble-ic query, for example:

```sql
mysql> SELECT word
   -> FROM words
   -> WHERE word LIKE 'do_s';

+-------+
| word  |
+-------+
| does  |
| dogs  |
| doss  |
+-------+
3 rows in set (0.00 sec)
```

**Date Formats and Zero-Filled Dates**

The International Organization for Standardization (ISO) has made many recommendations about the best ways to store dates. Their formatting standard #8601 has been adopted by MySQL as the proper way to store dates. It uses (in part) this format:

`YYYY-MM-DD`

where

- **YYYY** is a four-digit Gregorian calendar year
- **MM** is the two-digit month of the year between 01 (January) and 12 (December)
- **DD** is the two-digit day of the month between 01 and 31

Interestingly, the standard also recognizes zero-filled parts as legitimate and the best practice here is to use zero-filled parts whenever date components are missing. MySQL regards even the bizarre-looking date `0000-00-00` as legitimate.

In practice, then, you should zero-fill any date parts that you don't know. According to Virginia Woolf, for example, Modernism began sometime on or around December 1910:
1910-12-00

Or, if we find an old receipt in an archive but the year is illegible:

0000-03-24

NB: although MySQL has many column types that store dates, dates and times must always be handled as strings. So when INSERTing them into the database, always wrap them in quotation marks.

**TIMESTAMP versus DATETIME Columns**

MySQL has many column types that effectively store dates and times, either as individual components or all concatenated together nicely. Two types that sometimes confuse new users are the `timestamp` and the `datetime` columns. They look identical, so the confusion is understandable:

```
+---------------------+---------------------+
| timestamp           | datetime            |
+---------------------+---------------------+
| 2012-04-28 16:03:45 | 2012-04-28 16:03:45 |
+---------------------+---------------------+
```

However, `timestamp` columns have a very interesting property. **The first (that is, leftmost) timestamp column in a database table will automatically update to the current date and time whenever data is INSERTed or UPDATED in that row.** In other words, timestamp columns are good for answering queries like, “When was this record last modified?”

A `datetime` column, once set, is locked. It won’t change unless you modify it on purpose. A `datetime` column is useful to record things like the creation date of a record or the exact date and time of an event. They are good for answering queries like, “At precisely what date and time did the universe begin?”

NB: only the first timestamp column in a row updates automatically. Subsequent timestamp columns in any row behave identically to `datetime` columns. Weird but true.

Remember, too, that you can always use the `NOW()` function to insert the current date and time into a column. You do not need to parse out and construct a string in the precise format that a date or time column needs.

**Renaming Columns**

MySQL allows users to change the names of columns when you perform a query. The column name doesn't change permanently; the new name is simply a label that MySQL uses on a one-time basis and then discards. Most of the time, renaming columns isn't really necessary. Occasionally,
however, the column name is either unwieldy or unintuitive or both. At those times, renaming is easily done.

For example, I often concatenate together names. But the concatenation function becomes the name of the column, which looks pretty nasty:

```
mysql> SELECT CONCAT(first, ' ', last) FROM member;
```

```
+--------------------------+
| CONCAT(first, ' ', last) |
| Bob Smith                |
| Robert Smith             |
| (... data omitted ...)   |
+--------------------------+
17 rows in set (0.00 sec)
```

Renaming the column gives the result a more intuitive label:

```
mysql> SELECT CONCAT(first, ' ', last) AS 'Birdclub Members' FROM member;
```

```
+------------------+
| Birdclub Members |
| Bob Smith        |
| Robert Smith     |
| (data omitted)   |
+------------------+
17 rows in set (0.00 sec)
```

By default, MySQL doesn't make you surround table names with quotation marks, but that's because it assumes that any table name will be only one word long. Labels can be longer than one word, however, so if you use a label that's more than one word long, just wrap it in quotation marks as I've done above.
Database Exercises
Scenario:
Let’s say I belong to a birdwatching club and I’m building a database for my club. I need to keep track of my club members and I need to keep track of the birds they’ve seen and the dates when they saw those birds. Every month, I’ll award a prize to the person who’s seen the most birds. Some club members are more experienced birdwatchers than others, and they are called “guides” because they can lead birdwatching hikes. Guides are not eligible for monthly prizes. Eventually, it would be nice to have a webpage listing all the monthly winners.

Table Definitions
Following are the table definitions and graphical representations of the tables that make up the birdclub database. A PK in a column signifies a primary key. An FK in a column signifies a foreign key. Note that, especially in older versions of MySQL, tables have no official way to define foreign keys. We won’t go into the specific database techniques for managing foreign keys, so I’ve simply defined those columns with exactly the same data type as the primary key column.

The Finished Tables:

CREATE TABLE member (  
id tinyint unsigned NOT NULL auto_increment PRIMARY KEY,  
first varchar(25),  
last varchar(25),  
email varchar(35),  
phone varchar(12)
);

| member |  |  |  |  |  |
|--------|---|---|---|---|
| id | first | last | email | phone |
| PK |  |  |  |  |

CREATE TABLE bird (  
id tinyint unsigned NOT NULL auto_increment PRIMARY KEY,  
common varchar(50),  
scientific varchar(50)
);
CREATE TABLE sighting (  
  person tinyint unsigned NOT NULL,  
  bird tinyint unsigned NOT NULL,  
  date date  
);

CREATE TABLE guide (  
  id tinyint unsigned NOT NULL,  
  unique (id);  
)

<table>
<thead>
<tr>
<th>bird</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>common</td>
<td>scientific</td>
</tr>
<tr>
<td>PK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sighting</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>person</td>
<td>bird</td>
<td>date</td>
</tr>
<tr>
<td>FK</td>
<td>FK</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>guide</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
</tr>
<tr>
<td>FK</td>
<td></td>
</tr>
</tbody>
</table>
Many-to-Many Relationships:
- Many members can see many birds.
- Many birds are seen by many people.

Relational databases can’t manage many-to-many relationships directly, so we invent an intermediary table called a junction table that transforms the many-to-many relationship into two one-to-many relationships. Sometimes, junction tables correspond to real-world entities. That’s the case here: our sighting table is both a junction table and a real-world entity.

Resolve via a Junction Table (sighting) that manages two One-to-Many Relationships:
- Each sighting has only one member, but each member has many sightings.
- Each sighting has only one bird, but each bird can have many sightings.

The database strategy is to store a foreign key from the other table on the “crow’s foot” side.

Caveat Emptor:

The guide table has a one-to-one relationship with the member table. In a production system, I would collapse the two. However, I use it in the LEFT JOIN tutorial, so it stays.
Birdclub Database Schema

Table Descriptions

Birdclub Exercise — 35

mysql> DESC member;
+-------+---------------------+------+-----+---------+----------------+
| Field | Type                | Null | Key | Default | Extra          |
|-------+---------------------+------+-----+---------+----------------+
| id    | tinyint(3) unsigned | NO   | PRI | NULL    | auto_increment |
| first | varchar(25)         | YES  |     | NULL    |                |
| last  | varchar(25)         | YES  |     | NULL    |                |
| email | varchar(35)         | YES  |     | NULL    |                |
| phone | varchar(12)         | YES  |     | NULL    |                |
| guide | enum('yes','no')    | YES  |     | no      |                |
+-------+---------------------+------+-----+---------+----------------+

mysql> DESC bird;
+------------+---------------------+------+-----+---------+----------------+
| Field      | Type                | Null | Key | Default | Extra          |
|------------+---------------------+------+-----+---------+----------------+
| id         | tinyint(3) unsigned | NO   | PRI | NULL    | auto_increment |
| common     | varchar(50)         | YES  |     | NULL    |                |
| scientific | varchar(50)         | YES  |     | NULL    |                |
+------------+---------------------+------+-----+---------+----------------+

mysql> DESC sighting;
+--------+---------------------+------+-----+---------+-------+
| Field  | Type                | Null | Key | Default | Extra |
|--------+---------------------+------+-----+---------+-------+
| person | tinyint(3) unsigned | NO   |     | 0       |       |
| bird   | tinyint(3) unsigned | NO   |     | 0       |       |
| date   | date                | YES  |     | NULL    |       |
+--------+---------------------+------+-----+---------+-------+

mysql> DESC guide;
+-------+---------------------+------+-----+---------+-------+
| Field | Type                | Null | Key | Default | Extra |
|-------+---------------------+------+-----+---------+-------+
| id    | tinyint(3) unsigned | NO   |     | 0       |       |
+-------+---------------------+------+-----+---------+-------+
Birdclub Database

[database strategies: learning SQL, working with multiple tables, primary/foreign key relationships, junction tables, data types (strings, numbers, dates), zero-filled dates, the value of NULL]

The birdclub database is a simple tutorial database that captures information for a fictional birdwatching club. It manages members’ information, contains a master list of birds, and allows members to log bird sightings.

It introduces many of the basic fundamental database concepts: querying tables, working with multiple tables, understanding how primary and foreign keys manage both one-to-many and many-to-many relationships, and understanding data types (including zero-filled dates and why NULL is not the same thing as zero).

Load the Database

Via the command line, navigate to the folder where birdclub.sql file is located. Launch MySQL and load the database with either the source command or the slash-dot syntax:

    mysql> . birdclub.sql

Explore the Database

Using the birdclub database diagram and using the SHOW TABLES and DESCRibe table_name commands, familiarize yourself with the structure and content of the database. Try these queries:

    mysql> SHOW TABLES;
    mysql> DESCRIBE member;                        <- DESC is a synonym
    mysql> DESCRIBE bird;

Use the SELECT ... WHERE command to retrieve data. Here are a list of queries for you to try:

    mysql> SELECT * FROM bird;
    mysql> SELECT * FROM bird WHERE common LIKE '%sparrow';
    mysql> SELECT common FROM bird WHERE scientific LIKE 'Icterus%';
    mysql> SELECT * FROM bird where id in (4, 7, 23, 98);
    mysql> SELECT * FROM member WHERE email like '%shaw.ca';
    mysql> SELECT email FROM member WHERE first='Bob';
    mysql> SELECT COUNT(*) FROM member;
mysql> SELECT COUNT(*) FROM bird WHERE scientific LIKE 'Corvus%';
mysql> SELECT LENGTH(scientific) FROM bird WHERE common LIKE '%Meadow%';

Note that you can hit “Enter” (or “Return”) and MySQL will continue its prompt on a second line:

mysql> SELECT common FROM bird WHERE scientific LIKE 'Icterus'
   -> ORDER BY common;

mysql> SELECT CONCAT(first, ' ', last) AS 'No Phone Number!' 
   -> FROM member 
   -> WHERE phone IS NULL;

Try a multi-table query:

mysql> SELECT CONCAT(member.first, ' ', member.last) AS name, bird.common 
   -> FROM member, sighting, bird 
   -> WHERE member.first = 'Bob' 
   -> AND member.id = sighting.person 
   -> AND sighting.bird = bird.id;

Translate the following SQL queries into English questions:

Which English-language questions do these queries answer? Execute the query and see if you can figure out the question.

mysql> SELECT common, LENGTH(common) 
   -> FROM bird 
   -> ORDER BY LENGTH(common) DESC 
   -> LIMIT 1;

mysql> SELECT common, LENGTH(common) 
   -> FROM bird 
   -> ORDER BY LENGTH(common) ASC 
   -> LIMIT 1;

mysql> SELECT bird.common, COUNT(sighting.bird) 
   -> FROM sighting, bird 
   -> WHERE sighting.bird = bird.id 
   -> GROUP BY sighting.bird;
Exercises

Given what you’ve learned about SQL and about various data issues, try your hand at writing queries for the following questions. Remember, the query answers and some additional discussion are located in the Answers section towards the end of this course packet.

Try to write queries that answer the questions as precisely as possible and that give as little extra information as possible (ideally, of course, we’d want to give no extra information at all because we want to make our users’ lives as easy as we can). When we use SQL to put a search function on our database-driven research project or our database-driven archive, users will be puzzled if the answer to a question they’ve asked forces them to wade through superfluous information to find the answer they were looking for.

Write SQL queries to answer the following questions:

Write valid SQL that gives you exactly the correct answer—no more, no less. See the Answers section at the end of the course packet if you get stuck.

1. What is the first name of the member whose last name is Martin?
2. How many birdclub members are there?
3. Which birdclub members have no email addresses?
4. How many birdclub members have listed a phone number?
5. List the birdclub member guides.
6. What is the scientific name of the black-billed magpie?
7. What is the common name of Spizella breweri?

Slightly Harder Queries:

1. Which two birdclub members have the longest first names?
2. List all the birdclub members in alphabetical order.
3. Which bird has the shortest scientific name?
4. Write a multiple-table query to display information for each bird sighting. List the member’s name, the bird that person saw, and the date. Try to list the sightings in chronological order.
One-to-Many Relationships:
• A Player has only one position, but each position has many players.
• A Player has only one school, but each school can have many players.

The database strategy is to store a key on the “crow’s foot” side as a foreign key.
### 2012 NFL Prospects Database Schema

#### Table Descriptions

```sql
mysql> DESC player;
+------------------+----------------------+------+-----+---------+----------------+
| Field            | Type                 | Null | Key | Default | Extra          |
+------------------+----------------------+------+-----+---------+----------------+
| id               | smallint(5) unsigned | NO   | PRI | NULL    | auto_increment |
| first_name       | varchar(60)          | YES  |     | NULL    |                |
| last_name        | varchar(60)          | YES  |     | NULL    |                |
| rank_at_position | tinyint(3) unsigned  | NO   |     | NULL    |                |
| rank_overall     | smallint(5) unsigned | NO   |     | NULL    |                |
| position         | tinyint(3) unsigned  | NO   |     | NULL    |                |
| school           | smallint(5) unsigned | NO   |     | NULL    |                |
+------------------+----------------------+------+-----+---------+----------------+

mysql> DESC position;
+-----------------+---------------------------+------+-----+---------+----------------+
| Field           | Type                      | Null | Key | Default | Extra          |
+-----------------+---------------------------+------+-----+---------+----------------+
| id              | tinyint(3) unsigned       | NO   | PRI | NULL    | auto_increment |
| position        | varchar(40)               | YES  |     | NULL    |                |
| offense_defense | enum('Offense','Defense') | NO   |     | Offense |                |
+-----------------+---------------------------+------+-----+---------+----------------+

mysql> DESC school;
+-------+----------------------+------+-----+---------+----------------+
| Field | Type                 | Null | Key | Default | Extra          |
+-------+----------------------+------+-----+---------+----------------+
| id    | smallint(5) unsigned  | NO   | PRI | NULL    | auto_increment |
| name  | varchar(75)          | YES  |     | NULL    |                |
| state | char(2)              | YES  |     | NULL    |                |
+-------+----------------------+------+-----+---------+----------------+
```
2012 NFL Prospects Database

In *Moneyball*, real-world baseball general manager Billy Beane (played by Brad Pitt in the movie) attempts to save the Oakland A's by using a new style of player evaluation—abandoning many of the traditional baseball statistics (like homeruns or runs batted in), Beane looks to other stats like percentage of times getting on base and advancing bases. This kind of analysis is classically called *sabermetrics* and was invented, or so people say, by people like Earnshaw Cook way back in the 1960s.

Although other sports aren't as amenable to statistics as baseball, that doesn't stop people from trying. For example, every spring, the National Football League evaluates hundreds of players as the pro teams prepare to draft college players in late April. The pro evaluations are kept top secret, but as the draft draws near amateur sabermetrics fans begin populating the internet with their evaluations like mushrooms sprouting after a spring rain.

The NFL Prospects Database is one such dataset. It has been compiled by Matt Miller for BleacherReport.com (available at <http://bleacherreport.com/articles/1146711-matt-millers-final-2012-nfl-draft-big-board>).

Miller's web pages break down the prospects by position and rank them in comparison to others players at the same position. Tantalizingly, he gives each of the 350 players an overall rank, but recompiling that list from Miller's website is a major headache. And so if we want to query the data differently, we need to recombine the data into a database. That's what I've done here.

**Load the Database**

Via the command line, navigate to the folder where the `nfl_prospects.sql` file is located. Launch MySQL and load the database with either the *source* command or the slash-dot syntax:

```
mysql> .nfl_prospects.sql
```

**Explore the Database**

Using the database schema diagram and the *SHOW TABLES* and *DESC table-name* commands, explore the database. It contains only three tables, one of which is an entity table (player) and two of which are lookup tables (tables that contain static, unchanging information and that can provide keys usable in other tables). The *position* table stores a list of canonical football positions and whether those positions are offense or defense. The *school* table contains a list of colleges and universities as well as the states (or provinces) in which they're located.
Query the Database

1. List all the players in alphabetical order (last name, first name).
2. List all the players by Overall Ranking.
3. List all the players by Overall Ranking in reverse order.
4. List all the players who went to school in Saskatchewan (SK).
5. Show all the players from Florida (FL). Sort them in alphabetical order.
6. Show the top 25 defensive players (according to their overall rank).
7. List all the players in alphabetical order who are ranked in the top 3 at their position.
8. Show all the players in the top 150 overall rankings who are NOT in the top 3 at their position and whose last names begin with "S" and who went to a school in the state of Florida.

Summary and Grouping

1. List the various positions and the number of prospects at each position in descending order.
2. Which state has the highest number of prospects?
**MovieLens Database Schema**

**Schema Diagram**

### Many-to-Many Relationships:
- Between **user** and **movie** (mediated by junction table `rating`, a real-world entity)
- Between **movie** and **genre** (mediated by junction table `movie_genre`, not a real-world entity)

### One-to-Many Relationships:
- Between **age** and **user** (store a foreign key in user)
- Between **occupation** and **user** (store a foreign key in user)
### MovieLens Database Schema

Table Descriptions

```sql
code: mysql
mysql> DESC user;
+-----------------+---------------------+------+-----+---------+-------+
| Field           | Type                | Null | Key | Default | Extra |
+-----------------+---------------------+------+-----+---------+-------+
| id              | mediumint(8) unsigned | NO   | PRI | NULL    |       |
| gender          | enum('F','M')       | YES  |     | NULL    |       |
| age             | tinyint(3) unsigned | YES  |     | NULL    |       |
| occupation      | tinyint(4)          | YES  |     | NULL    |       |
| zipcode         | varchar(10)         | YES  |     | NULL    |       |
+-----------------+---------------------+------+-----+---------+-------+

code: mysql
mysql> DESC movie;
+-----------+---------------------+------+-----+---------+-------+
| Field     | Type                | Null | Key | Default | Extra |
+-----------+---------------------+------+-----+---------+-------+
| id        | mediumint(8) unsigned | NO   | PRI | NULL    |       |
| title     | varchar(255)        | YES  |     | NULL    |       |
| year      | year(4)             | YES  |     | NULL    |       |
+-----------+---------------------+------+-----+---------+-------+

code: mysql
mysql> DESC rating;
+-----------+---------------------+------+-----+---------+-------+
| Field     | Type                | Null | Key | Default | Extra |
+-----------+---------------------+------+-----+---------+-------+
| user_id   | mediumint(8) unsigned | NO   |     | NULL    |       |
| movie_id  | mediumint(8) unsigned | NO   |     | NULL    |       |
| rating    | tinyint(4)          | NO   |     | NULL    |       |
| timestamp | datetime            | YES  |     | NULL    |       |
+-----------+---------------------+------+-----+---------+-------+

code: mysql
mysql> DESC movie_genre;
+----------+---------------------+------+-----+---------+-------+
| Field    | Type                | Null | Key | Default | Extra |
+----------+---------------------+------+-----+---------+-------+
| movie_id | mediumint(8) unsigned | NO   |     | NULL    |       |
| genre_id | tinyint(3) unsigned | NO   |     | NULL    |       |
+----------+---------------------+------+-----+---------+-------+

code: mysql
mysql> DESC genre;
+-------+---------------------+------+-----+---------+----------------+
| Field | Type                | Null | Key | Default | Extra          |
+-------+---------------------+------+-----+---------+----------------+
| id    | tinyint(3) unsigned | NO   | PRI | NULL    | auto_increment |
| genre | varchar(12)         | NO   |     | NULL    |                |
+-------+---------------------+------+-----+---------+----------------+
```
The MovieLens Database

This MovieLens database contains 1,000,000 movie ratings from 6,000 users on 4,000 movies. The data was collected by GroupLens Research at the University of Minnesota in 2000-2001 and is available for download from <http://www.grouplens.org/taxonomy/term/14>.

Load the Database

Via the command line, navigate to the folder where the movielens.sql file is located. Launch MySQL and load the database with either the source command or the slash-dot syntax:

```sql
mysql> \movielens.sql
```

Note: This is a big database. On my Mac laptop, this load takes several minutes.

Explore the Database

Use the SHOW TABLES and DESC table_name commands to examine the database. Notice that there are tables representing entities (movie, user, rating), some junction tables that manage many-to-many relations (rating, movie_genre), and some lookup tables (genre, occupation, age). Check out the PDF diagram for a visual representation of this database.

As we’ve seen in other databases, it’s not uncommon that a junction table might also represent a real-world entity. Here, a movie rating is a real-world thing (movies are rated from 1 to 5 stars), but the table itself also manages two relationships: 1) any given user might rate many movies; and 2) any given movie might have many reviewers. So the relationship between user and movie is many-to-many and therefore needs a junction table. Here, the rating junction table stores not only a foreign key for the user and a foreign key for the movie, but it also stores the rating that this user gave to this movie and the timestamp when the rating was entered into the database.

Query the Database

Spend a bit of time querying the database. Try to write queries for the following:

1. List the movies in alphabetical order.
2. List the movies by year and, within each year, in alphabetical order.
3. What is the average movie rating of all movies?
4. List the movies whose title contains the word vampire.
5. List only the oldest movie whose title contains the word vampire.
Grouping Queries

6. Display the number of movies released during each year.
7. Which year released the most movies? Display the year and the number of movies.

Finding the Top 10 Movies

It’s important to remember that the MovieLens database contains only raw scores. It’s just a million random ratings in no particular order, with no sort of aggregation or grouping.

In order to find out, for example, what the Top 10 Most Popular Movies are, we need to perform a query that calculates an average score for each movie and sorts the results in descending order. Since this is a multiple-table query, we need to list columns using the table-dot-column syntax. MySQL won’t try to decipher which columns belong to which tables on its own.

mysql> SELECT movie.title, movie.year, AVG(rating.rating) AS score
    -> FROM movie, rating
    -> WHERE movie.id = rating.movie_id
    -> GROUP BY movie.id
    -> ORDER BY score DESC
    -> LIMIT 10;

Try the query above and look at the result set. I don’t know about you, but I haven’t seen any of those movies and they’re not the classic set I’d expect. What’s going on? Let’s find out if the number of movie reviews for each movie has anything to do with it.

mysql> SELECT movie.title, movie.year, AVG(rating.rating) AS score,
    -> COUNT(rating.rating) AS reviews
    -> FROM movie, rating
    -> WHERE movie.id = rating.movie_id
    -> GROUP BY movie.id
    -> ORDER BY score DESC
    -> LIMIT 10;

Now we can see that our Top 10 movies have actually received very few reviews. In this result set, The Gate of Heavenly Peace is the most-reviewed movie, but it has only three ratings. We can probably get a better representation of the top movies if we restrict our results to movies that have had at least 30 or more ratings.

mysql> SELECT movie.title, movie.year, AVG(rating.rating) AS score,
    -> COUNT(rating.rating) AS reviews
    -> FROM movie, rating
    -> WHERE movie.id = rating.movie_id
    -> GROUP BY movie.id
    -> HAVING reviews > 30
That gives us a more recognizable list.¹

Create a View

No matter how you do it, it's an unwieldy query to find the Top 10 movies. But that's exactly the kind of query that we'd expect to do all the time with this particular data set. One solution is to create a view. (Note: views were introduced to MySQL in version 5.0. If you're using an older version of MySQL, you'll need to upgrade before you can work with views.)

A view is a stored query. It’s simply one customized way of examining—or looking at—our data. We define a name for our view, assign a query to it, and then we can examine our data from this point of view at any time simply by looking at the view:

```
mysql> CREATE VIEW top_10 AS
    -> SELECT movie.title, movie.year, AVG(rating.rating) AS score,
    -> COUNT(rating.rating) AS reviews
    -> FROM movie, rating
    -> WHERE movie.id = rating.movie_id
    -> GROUP BY movie.id
    -> HAVING reviews > 30
    -> ORDER BY score DESC
    -> LIMIT 10;
```

Hint: you can go backwards in your query history by hitting the up arrow on your keyboard. Then scroll backwards with the back arrow until you're at the beginning of the query and simply add the CREATE VIEW clause at the beginning, then hit return.

Now we can look at the Top 10 movies at any time with a simpler query.

```
mysql> SELECT * FROM top_10;
```

¹ Remember: WHERE operates only on already existing columns, not calculated columns. We need to use the HAVING clause on columns that have been calculated on-the-fly using functions like COUNT(). The following version of the query seems logical but will generate an error. If you ever get this error, it's a clue that you need to use the HAVING clause.

```
mysql> SELECT movie.title, movie.year, AVG(rating.rating) AS score,
    -> COUNT(rating.rating) AS reviews
    -> FROM movie, rating
    -> WHERE movie.id = rating.movie_id AND reviews > 30
    -> GROUP BY movie.id ORDER BY score DESC LIMIT 10;
```

ERROR 1054 (42S22): Unknown column 'reviews' in 'where clause'
Notice that the query still takes a few seconds to execute (we'll fix that in a moment). Notice, too, that if we do a `SHOW TABLES` at this point, that `top_10` is included in that list. It looks like a table but it isn't really a table.

**Characteristics of Views**

Views have several interesting properties:

1. A view looks like a table—and we can query it as if it were a table—but it's not a table.
2. A view’s query gets re-executed every time we invoke the view.
3. We can refine the view by treating it as if it were a normal table. We can sort, group, and limit its output as we would any other table.

Views, then, are ways of storing a question whose answer may change continuously. They’re especially useful for complex queries, but they can store simple queries just as easily.

Interestingly, too, we can query a view as if it were a table. For example, to list the five lowest ranked movies in the Top 10 (that is to say, movies ranked #6 to #10), we can query the `top_10` view:

```sql
mysql> SELECT * FROM top_10
-> ORDER BY score ASC
-> LIMIT 5;
```

**Adding an Index**

Clearly, the Top 10 view needs to handle quite a lot of data in order to derive its answer. On my four-year-old laptop, the query takes 6.2 seconds (your speed may vary, of course!):

```sql
mysql> SELECT * FROM top_10;
... <table omitted> ... 
10 rows in set (6.20 sec)
```

Part of the problem is that ratings for any given movie can be scattered anywhere among the 1,000,000 rows of the `rating` table, and MySQL has to work very hard to collect together all the ratings for one movie in order to average them. Wash, rinse, repeat. It has to do the same thing for the other 4,000 movies as well.

The solution is to create an index. An index for a database table works like an index in a book. Without an index, I need to re-read the entire book every time I want to find references to space aliens or to Lady Gaga. In effect, MySQL needs to read the entire book for every query too. Less metaphorically, MySQL needs to examine 1,000,000 rows for each of the 4,000 movies. That's the equivalent of querying a table containing 4 billion rows. Every time I want to calculate the average
rating for, say, movie #17, MySQL needs to read all one million rows to make sure it has collected
every possible rating for movie #17 before it can calculate the correct average.

But with a sorted index, MySQL can more easily decipher when it has collected all the appropriate
ratings for movie #17 and so it calculates the average much more quickly. As MySQL moves
through the sorted index, it soon arrives at data for movie #18 and MySQL knows it can now
calculate movie #17’s average because there is no more data lingering further down in the table.

Primary key columns automatically get indexes. MySQL does that even if you don’t tell it to. You
don’t have a choice about that one. But you can manually add an index to any column of a MySQL
database. You can even create multi-column indexes. Since both the `movie_id` and the `rating`
columns are crucial to this query, let’s add a multiple-column index:

```
mysql> ALTER TABLE rating ADD INDEX (movie_id, rating);
Query OK, 0 rows affected (11.54 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

On my laptop, it took nearly 12 seconds to build the index and to sort it, but now MySQL can use
that sorted list of these two columns to find ratings on any given movie much more efficiently.
Let’s see how much faster our query is now:

```
mysql> SELECT * FROM top_10;
... <table omitted> ...
10 rows in set (2.78 sec)
```

At 2.78 seconds, our query is now significantly faster than it was before.

**Extending This Database**

One idea for extending this database would be to add a lookup table that maps zip codes to states.
Then we could query the database in new ways:

- Do people from California rate comedies more highly than do people from New York?
- Do people from New York rank gritty crime dramas more highly than do people from
  other parts of the country?
- Are the Top 10 movies from the Midwest different from the Top 10 from the South?
- Are Westerns more popular in the west?
- Did any states not contribute any movie reviewers to this project? (A left join problem!)
- Which state would be the most likely candidate to host a Big Lebowski Fest?
### 1818–1819 Factory Workers Database Schema

#### Schema Diagram

The **worker** and **factory** tables are connected through the **factory_num** column in the **worker** table and the **id** column in the **factory** table. The **id** column in the **worker** table is a primary key, and the **id** column in the **factory** table is also a primary key. The **factory_num** column in the **worker** table is a foreign key, indicating a relationship between workers and factories.

#### worker

<table>
<thead>
<tr>
<th>column</th>
<th>key?</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>primary key</td>
<td>integer</td>
</tr>
<tr>
<td>factory_num</td>
<td>foreign key</td>
<td>integer</td>
</tr>
<tr>
<td>roster_position</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>sex</td>
<td></td>
<td>enumerated list</td>
</tr>
<tr>
<td>given_name</td>
<td></td>
<td>character</td>
</tr>
<tr>
<td>last_name</td>
<td></td>
<td>character</td>
</tr>
<tr>
<td>age</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>years_experience</td>
<td></td>
<td>integer</td>
</tr>
<tr>
<td>days_sick_last_year</td>
<td></td>
<td>integer</td>
</tr>
</tbody>
</table>

#### factory

<table>
<thead>
<tr>
<th>column</th>
<th>key?</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>primary key</td>
<td>integer</td>
</tr>
<tr>
<td>location</td>
<td></td>
<td>character</td>
</tr>
<tr>
<td>owner</td>
<td></td>
<td>character</td>
</tr>
<tr>
<td>number_of_workers</td>
<td></td>
<td>integer</td>
</tr>
</tbody>
</table>

Note: the **id** column is not a sequence of auto-generated integers. Rather, its contents are integers from the original data set. They accurately represent the factories, though, so there's no need to revise them.
## 1818–1819 Factory Workers
### Exercise — 51

#### Table Descriptions

**mysql> DESCR worker;**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>smallint(5) unsigned</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
</tr>
<tr>
<td>factory_num</td>
<td>smallint(5) unsigned</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>roster_position</td>
<td>smallint(5) unsigned</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>sex</td>
<td>enum('male','female','unknown')</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>given_name</td>
<td>varchar(100)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>last_name</td>
<td>varchar(200)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>age</td>
<td>tinyint(3) unsigned</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>years_experience</td>
<td>tinyint(4)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>days_sick_last_year</td>
<td>smallint(6)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**mysql> DESCR factory;**

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>smallint(5) unsigned</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>location</td>
<td>varchar(25)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>owner</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>number_of_workers</td>
<td>smallint(6)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>
1818-1819 Factory Workers Database

This dataset was originally compiled by economist Douglas Galbi and it features lists of workers from various factories in England in the years 1818 and 1819. You can download the raw data and learn more about it at <http://www.galbithink.org/names/manfacdd.htm>.

Data sets like this help to contribute to prosopographies—data sets about groups of people. Historical data sets like these are often merged together out of disparate historical records and so are often incomplete or inconsistent. Nonetheless, datasets like these begin to give us interesting pictures about the life of ordinary people during various periods of history.

Prosopographies have been around since the ancient Greeks, but the Victorians really put them on the scholarly map. The *Prosopographia Imperii Romani* is a “who’s who” of the Roman empire and (in two editions) took 120 years to complete. Relational databases have boosted the popularity of prosopographies because these new technologies make it easier to collect, manage and query these vast quantities of information.

Load the Database

Via the command line, navigate to the folder where the `factory.sql` file is located. Launch MySQL and load the database with either the `source` command or the slash-dot syntax:

```
mysql> \. factory.sql
```

Explore the Database

Using the PDF diagram and the `SHOW TABLES` and `DESC table_name` commands, examine the database. Do a few sample queries to see what the data looks like.

Data Issues: NULL vs. zero

Here’s a sample row from the original raw dataset. It’s a CSV file (that is, comma-separated values). Notice that the last two values of the row (years of experience and number of sick days) contain no values:

```
11,189,0,"Thomas","Sidbotham",23,`
```

Now compare the database data. (The record has very long horizontal rows, so I’m going to use the \G syntax instead of a semi-colon in order to display the record vertically):

```
mysql> SELECT * FROM worker WHERE last_name LIKE 'Sidbotham'\G
```
Notice that the years of experience and the number of sick days are not zero, but rather NULL. Those two values—NULL and zero—are not synonymous and here we need to interpret them quite differently. NULL represents the absence of a value and is often used to signify missing information. Zero implies that the data is valid and correct, and that the answer just happened to be none.

In this case, it would have been quite misleading to have INSERTed zeroes into those fields because we might subsequently misinterpret the record as saying that Thomas was a new factory worker (having zero previous years of experience) and that he was quite healthy (having zero sick days). But that's not what the original data says. Instead, the best practice here is to set those two values to NULL, which signifies that the data is missing. We simply don’t know how many years of experience Thomas had, nor do we know how many days he was sick.

**Data Issue: Aggregated Data Sets**

Browse through the database and you’ll eventually notice workers whose first names are simply initials. All those workers have an “unknown” sex. It’s hard to say, but my guess is that the original survey at that particular factory did not record a person’s sex and this field was added sometime afterwards. A researcher must have interpreted a person’s sex based on his or her name, but was unable to make the same deductions for people listed only by their initials.

Notice, too, that not all records in the database are listed that way, which suggests to me (and Galbi intimates as much too) that this dataset was cobbled together from smaller, separate individual surveys. It's not uncommon for historical data to be pieced together in that fashion and, if you have data like it, you may find it difficult to step through the data normalization steps with data that has been aggregated from multiple sources.

My caveat is to try to be aware of (and to try to privilege) data that's “original” while remaining more skeptical about data that might have been added subsequently due to a researcher’s interpretations.

**Query the Database**

Spend a bit of time querying the database. Try to write queries for the following:

1. Which worker under age 18 had the greatest number of sick days?
2. Does the number of rows in the **worker** table match the sum of the **number_of_workers** column from the **factory** table? What can we do about that?

3. From the **worker** table, how many people work at factory #1220? Query the factory table to learn more about factory #1220. How do you interpret the result?

**Advanced Query Example: Find Possible Families with a Self Join**

See if you can find potential family groups in the database. Likely candidates would be groups of people sharing the same last name and who all work at the same factory. This is a slightly more complicated type of query than a simple **GROUP BY** clause can handle. In this case, the answer is to do a **self join**. It’s a complex query and the resulting dataset needs some interpretation, but let’s try it.

Since we’re comparing data in subsequent rows (“does any given person have the same last name and same factory number as anyone else in the table?”), no combination of **GROUP BY** or **HAVING** will get us what we need. Instead, we need to join our **worker** table to itself so that we can write the query as if we’re using two different tables. In the process, we also want to eliminate “trivial” cases from our result set—that is, cases in which a row in the left table matches itself in the right table. To do that, we need to install several caveats into our query, which makes it look a bit daunting. Here it is:

```sql
mysql> SELECT DISTINCT(a.factory_num) AS factory, b.last_name,
    -> IF(a.given_name<b.given_name,a.given_name,b.given_name) AS name1,
    -> IF(a.given_name<b.given_name,a.age,b.age) AS age1,
    -> IF(a.given_name<b.given_name,b.given_name,a.given_name) AS name2,
    -> IF(a.given_name<b.given_name,b.age,a.age) AS age2
    -> FROM worker AS a INNER JOIN worker AS b
    -> ON a.last_name = b.last_name
    -> AND a.factory_num = b.factory_num
    -> AND a.given_name != b.given_name
    -> ORDER BY b.last_name, factory;
```

When we perform it, we get a large result set of over 7,000 rows, more rows than there are in the original worker table. Why? Well, because we’re actually finding combinations of data patterns here. It’s a little bit like the old logic puzzle: if you have four people to choose from and need to send two of them to the store, how many possible combinations are there?

Let’s look at just a few rows from the result set where the last name is **Burgess** to see what we’re getting:
What this says is that we have two potential families named Burgess, one at factory #11 and the other at the infamous factory #1220 (which we investigated in one of the practice queries above).

At factory #11, we have these three people: Catherine Burgess (age 35), John Burgess (age 36), and Maryann Burgess (age 12). Ah, that looks like a nice, nuclear family from the early 19th century. They probably have a nice bungalow with a soot-encrusted, white picket fence and a dog named Spot who has only three legs due to a wagon wheel incident.

At factory #1220, we have these three people: Peggy Burgess (age 30), Mary Burgess (age 20), and Kitty Burgess (age 17). Perhaps they’re sisters. Or perhaps Peggy is the aunt of Mary and Kitty. From this dataset alone, we can’t tell.

And despite the lack of many first names, we can still reconstruct potential family units. Look at the Tute family, for instance:

At factory #24, it’s likely that J. (age 31) and R. (age 28) are married and have a child M. (age 9).

What’s important about this query, though, is that we’re starting to investigate patterns in the data. Most of our queries thus far have provided specific answers to specific questions. This query begins the process of interpreting the data, however, and to do so we’ve encoded some research assumptions into our SQL (a family all has the same last name, a family probably works at the same factory) in order to see if any patterns emerge. The dataset isn’t complete enough for us to arrive at definitive conclusions, but we can use what we’ve learned here to investigate further.

Assuming that we can find out where factories #11 and #1220 existed, we could begin to search through other legal documents from that town (birth, marriage, death certificates, e.g.) in order
to learn more about these family groups. We could scour newspaper accounts and look for paper trails like criminal records or deeds or wills.

Over time, more and more of those datasets will come online but, until then, we might need to make a road trip.
**More Database Design Exercises**

Congratulations! You’ve just been hired as a database designer! Try your hand at designing some database tables for each of these exercises.

**Exercise 1 — Mapping Traffic Accidents**

The thriving metropolis of Possum Knob has a traffic problem. Eastside councilman Ebenezer Hobgoblin feels that the west side of town has the most dangerous intersections, but Green Party councilwoman Gertrude Higgenbotham thinks that minivans and SUVs are the real problem. Mayor Dullard, however, believes that there are simply more accidents during the wintertime. Design a database so that the city council can keep track of traffic accidents to see if any of these theories are correct.

**Exercise 2 — Pandemic**

The CDC is currently tracking a pandemic in Pathogen City, but they’re unsure of the cause. So far, CDC has a list of hospital patients and is busy interviewing them about locations they’ve recently visited so that the CDC scientists can track exposure. Interestingly, not all patients who visited the same location are displaying symptoms. Could the date of visit affect a patient’s risk? Design a database to help the CDC learn more.

**Exercise 3 — Political Coverup**

Ambitious young investigative bloggers Woodstein and Bernward are uncovering a scandalous political coverup about space aliens. They are sifting through thousands of email messages looking for clues. For each email, they know the sender, the recipient, and whether the email contains any secret codewords that signify the knowledge of aliens among us. Woodstein and Bernward seek to discover who knew and when so that they can reveal how the coverup progressed through the political ranks. Design a database so that our intrepid bloggers can find some answers. (You can assume that each email is addressed to only one recipient.)

**Exercise 4 — Gas Prices**

A consumer advocacy group is building a smartphone app that will crowdsource gas prices so that a user can locate the cheapest gas prices near his or her current location. Registered app users should be able to record current gas prices at various stations (the app should save station name, address, prices, and date). The app will also have to store a User ID and a password for each registered user so that users can see a record of all their entries.
Exercise 5 — Beerclub

The first rule of Beerclub is “Don’t talk about Beerclub.”

Every week in a random secret basement location, men and women gather to rate beers. Each Beerclub meeting features a selection of different beers that have been specially chosen by the meeting organizer. Beers are presented to members anonymously in glasses labeled only A, B, C, D, E, etc. Every attending member tastes each beer, then rates it on a 1 to 5 scale (5 is the best). At the end of the tasting, the beers are identified to the attendees and the beer rankings are calculated. Design a database to store the data.

Exercise 6 — DHSI Students

We’re building a database to keep track of DHSI students and the courses they’re taking. For each student, we’d like to log the following info: name, email address, whether they have a scholarship (yes/no), their home institution, and which course they’re taking. The DHSI admins would also like to be able to find out how many more students can enroll in each course or whether it’s currently full.

Exercise 7 — Land Use

A recent research project has been looking at historical land development in this area. The archive contains numerous Deeds of Sale which reveal the land plot numbers as well as the old owner, the new owner, and the amount paid. Occupations of all people are included as well as whether the land has buildings or not. Design a database to help the historians manage their data.

Exercise 8 — Mold & Spore Collection

Hip teenager Trite Bromide is the go-to dealer among his geek circle of friends for any mold or spore sample you’d want. On a slower-than-usual Friday night, he decides to catalogue his extensive mold and spore collection. He needs a database to record which species each specimen is, from whom and when he procured it, and to which of his many exciting friends he’s given samples. Since you’re not busy this Friday either, you’ve volunteered to help him out.
REFERENCE MATERIALS
MySQL Functions

*Functions* are built-in MySQL entities that perform certain tasks—for example, retrieving parts of a string, combining two or more strings together, splitting dates into constituent components, retrieving the last inserted auto increment key values, etc.

Typically, a function takes some *input* (sometimes referred to as “parameters” or “arguments”), which we must provide inside parentheses, and the function returns to us some *output*. Listed below are a few useful MySQL functions. Inside the parentheses are some descriptive keywords to help you remember which types of input the function takes. For more information, check out a reference book like the *MySQL Pocket Reference* (2nd edition, O’Reilly, 2007) or the online function list at <http://dev.mysql.com/doc/refman/5.6/en/functions.html>.

*Note: square brackets around an argument list mean that the arguments are optional. The brief explanation following the function will tell you the difference.*

```
Note: there can be no whitespace between a function name and the parenthesis following it. This helps the MySQL parser distinguish between function calls and references to tables or columns that happen to have the same name as a function. However, spaces around function arguments are permitted.
```

**String Functions**

**CONCAT( comma-separated list of fields and/or literal strings )**

This function appends a list of strings to each other, beginning with the leftmost string and working towards the rightmost string.

Example: `CONCAT( first, ' ', last )`  <!-- combines fields first, a space, and last

**CHAR_LENGTH( literal string of field )**

Returns the number of multi-byte characters in the string. This function is Unicode-safe. This function is synonymous with `CHARACTER_LENGTH()`.

Example: `CHAR_LENGTH('九面計')`  <-- returns 3
Example: `CHAR_LENGTH('cat')`  <-- returns 3

**LENGTH( literal string or field )**

Returns the number of characters in the string. *Note: this form of the function is not Unicode-safe!* If you’re working with Unicode characters sets, use `CHAR_LENGTH()` or `CHARACTER_LENGTH()` instead.

Example: `LENGTH('cat')`  <-- returns 3
LOWER( literal string or field )

Returns the string converted to all lowercase letters.

Example: LOWER( 'GoOFUs' ) ←-- returns 'goofus'

TRIM( literal string or field )

Removes blank spaces (i.e., "whitespace") from both the left and right sides of the string.
Note: any spaces in the interior of the string will remain intact!

Example: TRIM( ' a and b ' ) ←-- returns 'a and b'

UPPER( literal string or field )

Returns the string converted to all uppercase letters.

Example: UPPER( 'goofus' ) ←-- returns 'GOOFUS'

Date Functions

MySQL uses standardized dates that are strings in the format 'YYYY-MM-DD' which is a format recommended by the International Organization of Standards (ISO). Any fields or strings specified below are assumed to be in this format.

Users are allowed to “zero fill” missing parts of dates. Therefore, MySQL regards '2012-06-00' and '0000-11-18' and even '0000-00-00' as valid dates. Typically, the zero-filled parts of dates represent information that’s missing. '1867-07-00' suggests that an event happened in July 1867, but we don't know on which day it occurred.

Notice, too, that any details about a date (for example, day name, day of the year, week of the year, etc.) can be derived from any date already in YYYY-MM-DD format, so it’s not necessary to calculate those things when you’re inserting dates into your database. Don’t store things you can calculate!

CURDATE()

Returns the current date. Note: takes no argument.

CURTIME()

Returns the current time. Note: takes no argument.
DATE_ADD( field or string, INTERVAL amount )

Adds a specified amount of time to a date, datetime, or timestamp field. Note: the amount unit specifier is typically singular. Common units are SECOND, MINUTE, HOUR, DAY, MONTH, YEAR.

Example: DATE_ADD( '2012-04-09', INTERVAL 3 DAY ) <-- returns '2012-04-12'

DATE_FORMAT( field or string, format string )

Returns a date, datetime, or timestamp string formatted according to the codes in the format string. Using the DATE_FORMAT() function means that you never have to store extra information about any date because you can retrieve it by reformatting the date. Note: literal characters are allowed, so you can add commas and other useful formatting pieces to your string. Note: TIME_FORMAT() uses the following formatting codes, too, but works only with the time portion of the string. Common formatting strings include:

- %a: Abbreviated weekday name (Sun..Sat)
- %b: Abbreviated month name (Jan..Dec)
- %c: Month, numeric (0..12)
- %D: Day of the month with English suffix (0th, 1st, 2nd, 3rd, …)
- %d: Day of the month, numeric (00..31)
- %e: Day of the month, numeric (0..31)
- %H: Hour (00..23)
- %h: Hour (01..12)
- %i: Minutes, numeric (00..59)
- %j: Day of year (001..366)
- %k: Hour (0..23)
- %l: Hour (1..12)
- %M: Month name (January..December)
- %m: Month, numeric (00..12)
- %p: AM or PM
- %r: Time, 12-hour (hh:mm:ss followed by AM or PM)
- %S: Seconds (00..59)
- %s: Seconds (00..59)
- %T: Time, 24-hour (hh:mm:ss)
- %W: Weekday name (Sunday..Saturday)
- %w: Day of the week (0=Sunday..6=Saturday)
- %Y: Year, numeric, four digits
- %y: Year, numeric (two digits)

Example: DATE_FORMAT( '2012-12-25', '%a, %b %D, %Y' )
<-- returns 'Tue, Dec 25th, 2012'
DATE_SUB( field or string, INTERVAL amount )
Subtracts a specified amount of time to a date, datetime, or timestamp field. Note: the amount unit specifier is typically singular. Common units are SECOND, MINUTE, HOUR, DAY, MONTH, YEAR.
Example: DATE_SUB('2012-01-09', INTERVAL 2 MONTH ) <-- returns '2011-11-09'

DAY( field or string )
Returns the day of the month represented in the date field or date string. This function is synonymous with DAYOFMONTH().

DAYNAME( field or string )
Returns the name of the day represented in the date field or date string (i.e., “Thursday”).

MONTH( field or string )
Returns the number of the month represented in the date field or date string. To get the name of the month, you can use the MONTHNAME() or the DATE_FORMAT() function.

NOW() Returns the current date and time. Note: takes no argument. MySQL is smart enough to use only the parts of the date and time that it needs, so you can safely insert NOW() into a date field, for example, and MySQL will ignore the time component.

TIME_FORMAT( field or string, format string )
Returns a formatted version of the time portion of a field or string. See the formatting codes listed above under DATE_FORMAT() to learn more about how to format your time.

YEAR( field or string )
Returns the year as a four-digit number.

Aggregate Functions
You can perform some interesting aggregate operations on columns or fields, which means that you can do some work in your SQL query without having to write some code in PHP or another scripting language to perform these operations.

AVG( expression )
Returns the average of a column of numbers.
Example: SELECT AVG( test_score ) FROM grades;
COUNT( * )  
Returns the number of times that expression was not null. The * wildcard metacharacter will, in effect, count the number of rows in the result set.

Example: SELECT COUNT( * ) FROM birds;
Example: SELECT COUNT( * ) FROM birds WHERE name LIKE '%oriole';

MAX( expression )  
Returns the maximum value from the column or expression.

Example: SELECT MAX( depth ) FROM ocean;

MIN( expression )  
Returns the minimum value from the column or expression.

Example: SELECT MIN( circumference ) FROM planet;

SUM( expression )  
Returns the sum of the values from the column or expression.

Example: SELECT SUM( kilometers ) FROM travel;

Database Functions

MySQL has a few functions that will give you information about the database you're using.

DATABASE()  
Returns the name of the database currently being used.

Example: SELECT DATABASE();  <-- returns 'birdclub'

LAST_INSERT_ID()  
Returns the last value generated by an auto_increment field.

Note: PHP also has a mysql_insert_id() function, so you can retrieve this value from a web script without having to perform a MySQL query.

Example: SELECT LAST_INSERT_ID();  <-- could return, say, '4'
DATA INPUT & OUTPUT
The LOAD DATA Command

Q: When should I use LOAD DATA and when should I use the \ or source commands?

A: LOAD DATA imports files full of raw data (tab- or comma-separated values, for example), but \ and source import external files that contain SQL statements.

At some point in your long and storied database career, you’re going to need to import data from somewhere else (and I’m guessing that’s right now since you’re reading this). Data arrives in a dazzling array of costumes and disguises. Spreadsheets, for example, commonly export data as comma-separated values (so-called “CSV”) files. Word processing tables, on the other hand, often format their data as tab-delimited values—a row of data whose various entries are separated by tab characters. Still other formats separate individual data entries with colons or double colons or vertical pipe characters: |. Regardless, all of these are simply plain text files, so they’re good solutions to the more general problem of data portability. Fortunately, MySQL can import any and all of them with the LOAD DATA command.

The general format of the LOAD DATA command is as follows (I’ll enter it on multiple lines for clarity; remember that all SQL statements need to end with a semi-colon):

```
mysql> LOAD DATA LOCAL INFILE some_filename
    -> INTO TABLE some_table_name
    -> FIELDS TERMINATED BY some_character_sequence
    -> LINES TERMINATED BY some_other_character_sequence;
```

Three points to remember:

1. The FIELDS TERMINATED BY clause and the LINES TERMINATED BY clause must appear in the order specified above whenever both are used together. (They are in alphabetical order—fields before lines—if that helps.)

2. You can safely omit the FIELDS TERMINATED BY clause if your file is tab-delimited. A tab-delimited file is the default file format for the LOAD DATA command.

3. You can safely omit the LINES TERMINATED BY clause if your file uses the Unix/Linux/Macintosh line ending character (symbolized as \n). The \n line-ending is the default file format for the LOAD DATA command.
But what if your data isn't tab-delimited? And what if it was created on a Windows machine? Then you need to know a few more intimate details about what condition your data is in. In particular, you need to consider these two questions:

1. What's the field delimiter?

The field delimiter separates the individual data components within the line. As mentioned above, common field delimiters are commas, tabs, colons, and pipes. Here's what a sample record might look like in a variety of different formats:

**Tab-Separated (tab represented as →)**

Victoria → British Columbia → Canada

*Note: a more common representation of the tab character is the \t symbol.*

**Comma-Separated**

Victoria,British Columbia,Canada

*Note: CSV files will often wrap problematic entries with quotation marks, especially when an entry contains a comma.*

**Colon Delimited**

Victoria:British Columbia:Canada

*Note: many files, including things like Unix password files, are often colon-delimited. Some of the raw data entries for our tutorial databases were double-colon delimited:*

Victoria::British Columbia::Canada

Perhaps predictably, we have to tell LOAD DATA which character separates the fields in each record:

**FIELDS TERMINATED BY \t**

*← Yes, \t represents a tab character.*

or

**FIELDS TERMINATED BY \,\**

or

**FIELDS TERMINATED BY \:'\**

or even something as crazy as

**FIELDS TERMINATED BY \:|+|@4$’**

---

2 And yes, while you're doing this it does help to listen to “Just Dropped In (To See What Condition My Condition Is In)” from *The Big Lebowski* soundtrack.
As mentioned above, **LOAD DATA**’s default field delimiter is the tab character (\t). So if you have tab-delimited data, you can safely omit the **FIELDS TERMINATED BY** clause.

2. **What’s the record separator?**

A common text file strategy is to list one record per line. One line in the file represents one record in the database. Seems straightforward, yes? Sadly, it’s not. Oddly, we have to tell **LOAD DATA** how to interpret the end of a line. Weird, but true.

If you hit Enter on a Windows machine, you get a different invisible character than if you hit the same key on a Macintosh or on a Linux machine. Sure, the characters are invisible and they behave the same way on their respective computer platforms (each advances you to the next line), but that doesn’t mean they’re the same under the hood. There, they’re different things. Typically, hitting Enter or Return gives you some combination of a “linefeed” character (symbolized as \n) and a carriage return (symbolized as \r).

Consequently, this means you often need to know more about the internal guts of your text file than perhaps you really wanted to. But there are really only two line endings to know about: \r\n on Windows and \n on everything else.

<table>
<thead>
<tr>
<th>Platform</th>
<th>end of line character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>\r\n</td>
</tr>
<tr>
<td>Macintosh OS X</td>
<td>\n</td>
</tr>
<tr>
<td>Linux, Unix</td>
<td>\n</td>
</tr>
</tbody>
</table>

What does that mean for **LOAD DATA**? We have to use one of the following clauses in our **LOAD DATA** command to specify how our lines are terminated:

**LINES TERMINATED BY \r\n**

Or,

**LINES TERMINATED BY \n**

As mentioned above, the **LOAD DATA** command’s default character line ending is \n. So if you’re importing a file *that was created on a Macintosh or Linux or Unix machine*, you can safely omit the **LINES TERMINATED BY** clause. That’s true even if you’re importing data into MySQL on a Windows machine.

Just to clarify: *it doesn’t matter which operating system you’re currently using. It matters which operating system created the file you’re trying to import.*
How do you learn about the internal guts of your file? I like to use a simple programmer’s plain text editor because I can view and/or change all the line endings with one click of a mouse. Check out TextWrangler for Macintosh or Notepad++ for Windows.³

Practice

We’re going to gain some practice by loading a table from five different datasets. We’ll create a small new database with one empty table and then we’re going to use the LOAD DATA command in five different guises in order to import various datasets into it. So before you launch MySQL, change directories on the command line so that you’re in the Initializations folder.

1. Launch MySQL. Create a new database and USE it.

```
mysql> CREATE DATABASE geo DEFAULT CHARSET utf8;
mysql> USE geo;
```

The name of our database is `geo` and we’re telling MySQL that the data in it will be Unicode-compliant—that’s the `utf8` part. I always make my databases Unicode-compliant because it’s the best way to handle characters from different languages that have interesting diacritical marks. We’ll see how that affects our data momentarily.

2. Create a new table called `city`.

```
mysql> CREATE TABLE city (  
  -> city VARCHAR(30),  
  -> province VARCHAR(30),  
  -> country VARCHAR(30)  
  -> );
```

Every column will automatically be able to store Unicode data because that’s the default character set for the whole database.

3. First dataset: `cities.txt`

The first datafile, `cities.txt`, uses the default record separator `\n` and the default field separator `\t`. Therefore, we do not need to specify those in our LOAD FILE command.

```
mysql> LOAD DATA LOCAL INFILE 'cities.txt' INTO TABLE city;
Query OK, 11 rows affected (0.00 sec)
Records: 11 Deleted: 0 Skipped: 0 Warnings: 0
```

Let's look at the dataset we just loaded. Notice that the diacritical marks on Montréal, Québec should have loaded properly:

```
mysql> SELECT * FROM city;
```

Use a plain text editor to open the cities.txt data file. Notice that fields containing NULL are represented as \N in the data file. Don't confuse \N with \n. Capitalized \N represents NULL; the lowercase version \n represents a line-ending character. This trick works on the command line too.

4. Second dataset: cities_windows_csv.csv

The second dataset we'll add to our table is from a Windows CSV file called cities_windows_csv.csv. As you might have predicted, the record delimiter is the Windows \r\n character combination and the field separator is a comma.

Remember that the FIELDS clause must precede the LINES clause when both are used together.

```
mysql> LOAD DATA LOCAL INFILE 'cities_windows_csv.csv'
    -> INTO TABLE city
    -> FIELDS TERMINATED BY ','
    -> LINES TERMINATED BY '\r\n';
Query OK, 11 rows affected (0.00 sec)
Records: 11 Deleted: 0 Skipped: 0 Warnings: 0
```

Again, let's look at the loaded dataset just to make sure that we've added eleven new rows to the table.

```
mysql> SELECT * FROM city;
```

5. Third dataset: cities_pipe.txt

The third dataset is in the file called cities_pipe.txt. Since it uses the default \n line-ending character, we can safely omit the LINES TERMINATED BY clause. However, we still need to tell LOAD DATA that this file separates the individual fields with the pipe character: |.

```
mysql> LOAD DATA LOCAL INFILE 'cities_pipe.txt'
    -> INTO TABLE city
    -> FIELDS TERMINATED BY '|';
Query OK, 10 rows affected (0.00 sec)
Records: 10 Deleted: 0 Skipped: 0 Warnings: 0
```
6. Fourth dataset: cities_doublecolon_windows.txt

The fourth dataset uses a double-colon as the field terminator and the Windows character combination `\n\n` as the record terminator.

```
mysql> LOAD DATA LOCAL INFILE 'cities_doublecolon_windows.txt'
    -> INTO TABLE city
    -> FIELDS TERMINATED BY '::'
    -> LINES TERMINATED BY '\n\n';
Query OK, 9 rows affected (0.00 sec)
Records: 9  Deleted: 0  Skipped: 0  Warnings: 0
```

7. Fifth dataset: cities_wrong_order.txt

Our last dataset offers an interesting wrinkle. Feel free to take a look at it in a text editor. Although it uses the default field (\t) and line terminations (\n), the columns are out of order (they are listed as country, city, province) and there is a header row at the top that lists the labels for the columns. In this case, we want MySQL to skip the first row entirely and then insert data in a slightly different order than is presented. Here's how:

```
mysql> LOAD DATA LOCAL INFILE 'cities_wrong_order.txt'
    -> INTO TABLE city
    -> IGNORE 1 LINES
    -> (country, city, province);
```

You should now have a complete table. Do a `SELECT *` in MySQL to see how it looks.

Add a Primary Key Column

You'll notice that none of the datasets included a primary key column. It's often easier to combine all your datasets into one table first and then add a primary key column afterwards. It's simple to do with the `ALTER TABLE` command:

```
mysql> ALTER TABLE city
    -> ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT FIRST,
    -> ADD PRIMARY KEY (id);
```

Notice the keyword `FIRST`. MySQL's default position for adding new columns is at the end of a row, but it's customary that a primary key column is listed as the first column, so we've told MySQL to put our new key column there.

Also notice the comma after `FIRST`. That tells MySQL that there's more of this command to follow. In this case, we're also telling MySQL to use the newly generated column as a primary key. If you accidentally terminated the command too early, you can always use the `ALTER TABLE city ADD PRIMARY KEY (id)` command later.
Do a SELECT * to see the results.

Caveats to Remember with LOAD DATA

The LOAD DATA command is pretty powerful and, once you get used to wielding it, very flexible. Here are some other caveats and tips:

Get a good programming text editor for data cleanup tasks.

You might not have realized this yet, but all computer programs begin their lives as plain text files. Webpages are just plain text files too. So are data dumps. Plain text is everywhere—and that includes all the dirty data we're trying to clean up before we bring it into our databases.

I never use tools like Microsoft Word or Excel or Macintosh Pages or even Macintosh TextEdit for these types of duties. I need to know that I have plain text, the whole plain text and nothing but plain text. So I opt for one of the text editors favoured by computer programmers.

There are lots and lots of them. Feel free to browse around to find one that you like. Two simple-to-use and highly recommended editors are TextWrangler for Macintosh (http://www.barebones.com/products/textwrangler/) and Notepad++ for Windows (http://notepad-plus-plus.org). If you find one you like better, please use it. You won't regret downloading it. The two editors listed here are free, but there are some very nice ones out there that cost a little money.

I use a text editor to identify line-ending characters and, when needed, to change them all with just one click of my mouse. A programmer's text editor also has extremely sophisticated search-and-replace features too, which can come in handy.

And even if you don't need it yet, eventually you'll want to control the character set of your data. Try to put everything into Unicode (UTF-8). It was designed to become an industry standard, and we should do our best to put that into practice. Word processors often hide their character sets from you, but a programmer’s text editor gives you complete control over character sets.

LOAD DATA assumes that there are the same number of fields in the dataset as in the table and, moreover, that they are in the same order.

Importing data is easy when the data set has a one-to-one correspondence with the table structure. It's also pretty easy to rearrange the order of columns. It's slightly harder—but still possible—to skip columns and it's even possible to use LOAD DATA to do some pre-processing operations on your data before the rows are inserted. That's Jedi-level work, though, young Skywalker. If you feel the need to test your mettle, check out a good reference (like Paul Dubois's MySQL Cookbook from O'Reilly) to learn more.
NULL entries should be symbolized with \N

Don’t confuse the \n line-ending character with the \N character symbolizing NULL. If your
data has blanks for empty cells, you might need to do a little cleanup work with your newly
downloaded programming text editor before you’re ready to import.

Note: the \N character sequence for NULL works on the command line too. So you can use
this trick when you’re using \ or source to load data, and you can even use it when you’re
typing SQL by hand. Of course, you can always type out NULL too.

CSV files often enclose complex data inside quotation marks. Inform LOAD DATA
accordingly.

CSV is a surprisingly difficult file format to work with. Problematic entries (those
containing commas, for instance) are overridden by surrounding the data with quotation
marks. In other words, a CSV file is comma-delimited except when it’s quotation delimited.
Moreover, not all entries may be delimited by quotation marks—only the problematic ones.
You need to inform LOAD DATA of these idiosyncrasies so that it behaves properly.

In this case, tell it that the field delimiter is a comma but that there may be optional
quotation marks that should be stripped out before data is inserted into the table:

    FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY ""

That looks a bit odd, but the important character sequences here are quote + comma +
quote and single quote + double quote + single quote.

And just to make matters even worse, I suppose you could have this one (you’ll need to
check your data with your handy-dandy programming text editor to find out for sure):

    FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY """

To us mere mortals, that one ends with double quote + single quote + double quote.

Get diagnostics with the SHOW WARNINGS command.

When things go slightly sideways, MySQL will dutifully inform you that there are some
warnings awaiting. You can use the SHOW WARNINGS command to retrieve them. You can
then decide whether the problem warrants further attention or not.

One cautionary note about warnings—**you must ask for them immediately after they occur.**
Old warnings are cleared out and new warnings are loaded after every SQL command. So if
you had a problematic data entry two queries ago and want to revisit those old warnings, it’s
too late. Even if your most recent command(s) produced no warnings at all, those old
warnings from two queries ago are still gone.
Other Bulk Loading Techniques

As we've seen, MySQL can take input from the command line. But if you have to load a table with lots of records—even thousands—sitting and typing at the command line is no good. Fortunately, MySQL provides a variety of alternatives: you can bulk load records from a file that contains SQL statements. Here are the steps:

1. Using a text editor like Notepad or pico or TextWrangler, create a plain text file (not a Word doc!) that has well-formed SQL. The document will be a list of all the SQL statements that you would run from the command line if you had enough time.
2. Load the file using one of two techniques, depending whether you’re already in MySQL or whether you’re in Unix.

Step 1. Make a Plain Text File full of SQL

Let’s say that I have a table called country defined like this:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>char(2)</td>
</tr>
<tr>
<td>name</td>
<td>varchar(50)</td>
</tr>
</tbody>
</table>

I’m planning to load the table with ID values and official country names taken from the International Organization for Standardization list at <http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html>. I will generally insert data using a line like this:

```
INSERT INTO country (id, name) VALUES ('CA', 'Canada');
```

I can put lots of SQL statements into my plain text document, which I’ll call country_load.sql (a .txt ending would work, too, but .sql might help me remember better what’s in the document). Notice that I put a semi-colon at the end of each statement, just as I would if I were typing all this at the command line:

```
INSERT INTO country (id, name) VALUES ('AW', 'Aruba');
INSERT INTO country (id, name) VALUES ('AU', 'Australia');
INSERT INTO country (id, name) VALUES ('AT', 'Austria');
INSERT INTO country (id, name) VALUES ('CA', 'Canada');
INSERT INTO country (id, name) VALUES ('CN', 'China');
INSERT INTO country (id, name) VALUES ('CX', 'Christmas Island');
```
Step 2. Bulk Load the SQL

After creating my file, I have two choices when it comes to bulk loading. One option is to do it from within MySQL. The other is to load straight from the Unix command line.

**Option A: Loading from MySQL**

The command is `\. (“slash-dot”), followed by a space and the filename or its path (note: `mysql>` is your command-line prompt):

```sql
mysql>\. path/to/country_load.sql
```

File paths are simpler, of course, if you first `cd` to the directory where your bulk load document resides before you launch MySQL. If you’re already in the proper directory, then the command is simpler:

```sql
mysql>\. country_load.sql
```

Trivia: a synonym for the slash-dot syntax is the keyword `source`:

```sql
mysql> source country_load.sql
```

**Option B: Loading from Unix**

The command is a standard Unix-style redirect (note: `$` is your Unix command-line prompt):

```bash
$ mysql -u username -p database_name < path/to/country_load.sql
```

You’ll be asked to enter your password, and once you’ve successfully logged in, then MySQL will execute the file you’ve listed. If you first `cd` to the directory where your bulk load document resides, then the command is slightly simpler:

```bash
$ mysql -u username -p database_name < country_load.sql
```
Remember that you can put any valid SQL into your plain text document. For example, rather than telling MySQL via the command line which database I want it to use, I could add a line to the beginning of my .sql document. Let’s say I want to use my birdclub database:

```sql
USE birdclub;
INSERT INTO country (id, name) VALUES ('AW', 'Aruba');
INSERT INTO country (id, name) VALUES ('AU', 'Australia');
INSERT INTO country (id, name) VALUES ('AT', 'Austria');
(... remainder of document omitted ...)
```

In that case, I would execute my .sql document like this:

```bash
$ mysql -u username -p < country_load.sql
```

**Advice: Building a Bulk Load Document**

Copy and paste is a tried-and-true method. However, I usually prefer to write a small computer program or two that helps me generate my bulk-loading documents. Programming is a great skill for digital humanists and any language will do: PHP, Python, Ruby, and Perl are all popular languages. And so if you know some programming, you’ll want to brush up on some skills:

- **File Manipulations**: Learn how to open, read from, write to, and close a file from within your program. See PHP’s functions `file_get_contents()` and `file_put_contents()`, for example.
- **Regular Expressions**: Regular expressions are string pattern-matching templates, and are tremendously useful when creating bulk-loading SQL documents. Jeffrey Friedl’s *Mastering Regular Expressions* (O’Reilly) is the best regular expression book out there.
- **Scraping**: “To Scrape” means to extract data from existing web pages. You can retrieve an HTML page over the web with PHP’s `file_get_contents()` and you can scrape it using regular expressions. Your program can output legitimate SQL into the file that you’ll bulk load.

It’s possible to write your program so that it loads the data for you directly into the database. Even if I opt for that technique, though, I usually output at least one bulk-load SQL document. That way, in the event that my database ever crashes or needs resetting, I have an “initialization” document that I can load directly from the command line without having to re-run any of my specialized programs.
mysqldump

What if you need to move your database to another machine? Or upload it to a server? The answer is a program called mysqldump that you can run from the command line and that will export your database as a plain text file. The output can then be bulk loaded on the new machine. The general command structure is this (optional entities are in square brackets):

```
mysqldump [options] database_name [table_name] > filename
```

If no tables are listed, mysqldump dumps all the tables in the database. If filename does not exist, it will be created. Most (but not all!) mysqldump options are preceded by two hyphens and include:

--add-drop-table
Includes a statement to drop (that is, delete) each table if it already exists. This is especially useful if you want to re-sequence auto_increment columns or in order to compact tables so that they use minimal disk space. (Tables that endure a lot of DELETEs can lose their efficiency after a time.)

--no-create-db
Omits the CREATE DATABASE statement from the dump.

--no-create-info
Omits CREATE TABLE statements from the dump.

--no-data
Omit all data contained in the tables. This is useful if you want to clone an empty version of a database.

--opt
Optimize the output for fast reloading. Harder to read for humans, but easier to read for the machine.

--xml
Generate XML output

-u username
A valid username to access the database with (Note: ONE hyphen!)

-p
Tells mysqldump to prompt you for the password on the next line (Note: ONE hyphen!)

--password=pwd
Use pwd as password and don’t prompt for it (Note: TWO hyphens!)

Unless you specify otherwise, the output will just be sent to the screen—where it’s not very useful. So you should specify the filename into which the data should be routed using the familiar Unix redirection symbol: >. (Special Note: if your password contains a space, you’ll want to wrap it in quotation marks since a space is a traditional command line delimiter.)

Here’s an example of the mysqldump command (note: all on the same line; ignore the line wrap here!):
mysqldump --add-drop-table --no-data -u huco520 --password=huco520
   birdclub > birddump.txt

That will dump a special add-drop command for each table and the table definitions themselves, but no data. Remember that -- in MySQL signifies that what follows is a human-readable comment and that MySQL should ignore the rest of the line. A typical dump file looks like this (I’ve listed only the beginning and the first table):

   -- MySQL dump 9.10
   --
   -- Host: localhost    Database: birdclub
   -- ------------------------------------------------------
   -- Server version       4.0.18-standard
   --
   -- Table structure for table `bird`
   --

   DROP TABLE IF EXISTS bird;
   CREATE TABLE bird (  
id tinyint(3) unsigned NOT NULL auto_increment,  
common varchar(50) default NULL,  
scientific varchar(50) default NULL,  
PRIMARY KEY (id)
) TYPE=MyISAM;

If your tables contain good data and if you don’t especially care if you can read all the INSERT statements, you’ll probably just want to go with the optimized version:

   mysqldump --opt -u huco520 --password=huco520 birdclub > birddump.txt

Note: since Unix file extensions are all somewhat arbitrary anyway, your dump files can end in something like .txt or .sql extensions. Either is acceptable, although I often use .sql because it reminds me what’s inside the file.
Installing & Using MySQL
Macintosh Installation Instructions

Here are instructions to help you download and install MySQL on your Macintosh.

1. Discover which operating system version you’re running.

   Go under the Apple Menu to About This Mac.

   Write down your Operating System version: ______________________

2. Discover whether your processor is 32- or 64-bit.

   Still in the About This Mac window, click on More Info. In the resulting window, make sure Hardware is clicked in the left-hand pane and search for the Processor Name entry in the righthand pane. Circle your processor’s name in the table below.

<table>
<thead>
<tr>
<th>Processor Name</th>
<th>32- or 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core Solo</td>
<td>32 bit</td>
</tr>
<tr>
<td>Intel Core Duo</td>
<td>32 bit</td>
</tr>
<tr>
<td>Intel Core 2 Duo</td>
<td>64 bit</td>
</tr>
<tr>
<td>Intel Quad-Core Xeon</td>
<td>64 bit</td>
</tr>
<tr>
<td>Dual-Core Intel Xeon</td>
<td>64 bit</td>
</tr>
<tr>
<td>Quad-Core Intel Xeon</td>
<td>64 bit</td>
</tr>
<tr>
<td>Core i3</td>
<td>64 bit</td>
</tr>
<tr>
<td>Core i5</td>
<td>64 bit</td>
</tr>
<tr>
<td>Core i7</td>
<td>64 bit</td>
</tr>
</tbody>
</table>

3. Download MySQL.

   In your web browser, surf to

   http://dev.mysql.com/downloads/mysql/

   You’ll be offered a choice of packages to download depending upon your operating system version and whether you’re running a 32-bit system or a 64-bit system. Use the information above to download the proper version.
3. When MySQL downloads and unzips, install both packages (they look like open boxes) by double-clicking on them and following the instructions.

   The two packages will be called something similar to the following (note: your packages’ names may be different based on the version of MySQL and your operating system, but this should help you to identify whether you have downloaded packages that seem to be correct):

   1) mysql-5.5.23-osx10.6-x86.pkg
   2) MySQLStartupItem.pkg

   After the packages have run, MySQL is now installed. However, there are some additional steps you can take to help to insure that MySQL is as easy to use as possible.

4. Alter your Unix path to include MySQL’s directory.

   In Applications, launch the Terminal program. It’s the command-line interface for Macintosh. You’ll be using the Terminal window extensively as you use MySQL and so you may choose to keep Terminal in your Dock. (Click and hold on the dock icon, then select Options > Keep in Dock.)

   At the command line prompt, launch an old-fashioned, Unix text editor by typing

   `pico .bashrc`

   Then type the following (note: pay particular attention to spaces and punctuation!):

   ```
   PATH=$PATH:/usr/local/mysql/bin/
   export PATH
   ```

   Then type control x. (Control will look like a caret character):

   `^x`

   Hit y to answer yes to the “Save Modified buffer?” question, and then hit return to exit the text editor and to save the file.

5. Restart your computer so that the new changes take effect.

   You’re done. Go have a beverage. Then look at the document Securing a New MySQL Installation.

   If you’d rather not restart your computer right now, continue by following the rest of the instructions below.
6. **Push the new path into your environment.**
   
   In order for your computer to know where MySQL is, type the following at the command line prompt in the Terminal application:
   
   ```
   source .bashrc
   ```

7. **Start the MySQL daemon.**
   
   At the command-line prompt type
   
   ```
   sudo /Library/StartupItems/MySQLCOM/MySQLCOM start
   ```
   
   Then enter your administrative password (it will be invisible for security purposes) and hit return. If you’ve never used `sudo` before, you may get a warning message. That’s OK. Just answer yes.
   
   The MySQL daemon allows you to connect to the MySQL server on your computer via a socket. If the daemon isn’t running, you’ll get an error like this when you try to launch MySQL:
   
   ```
   ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)
   ```
   
   Note: next time you reboot your computer, the daemon will launch automatically and so you won’t need to start the daemon from the command line.
Windows Installation Instructions

Here are instructions to help you download and install MySQL on your Windows machine. I tested this process on a Windows 7 machine. If you’re running an older version of Windows, some of the configuration locations might be different, but the process is generally the same.

1. Determine whether your machine is 32-bit or 64-bit.

Open System by clicking the Start button, clicking Control Panel, clicking System and Maintenance (“System and Security” in Windows 7), and then clicking System.

In Vista or Windows 7, look for the System type information.

For other systems, check this chart:

<table>
<thead>
<tr>
<th>OS</th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Windows XP Professional Version [year]</td>
<td>32-bit</td>
</tr>
<tr>
<td>Microsoft Windows XP Professional x64 Edition Version [year]</td>
<td>64-bit</td>
</tr>
<tr>
<td>Windows XP Home</td>
<td>32-bit</td>
</tr>
<tr>
<td>Windows XP Media Center Edition</td>
<td>32-bit</td>
</tr>
<tr>
<td>Windows Vista Starter Edition</td>
<td>32-bit</td>
</tr>
<tr>
<td>Windows 7 Starter Edition</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

2. Download MySQL.

In your web browser, surf to

http://dev.mysql.com/downloads/mysql/

Download either the 32-bit or 64-bit MSI Installer package, depending on your machine’s configuration.

MySQL may ask you to sign up as a registered user, but that’s an optional step. Don’t worry. They’re not evil.

You can also choose the site from which to download MySQL.
3. Run the Installer and follow the instructions.

MySQL’s site also has a very clear set of instructions. Surf to


Note: Supposedly, a configuration wizard runs after the installation process to help you configure MySQL properly. On a recent test install, I didn’t have access to that Wizard. If your wizard runs, it should give you access to the MySQL Command Line Client, which will appear in the Start menu. If you don’t have a wizard, continue by following the steps below.

4. Discover the directory path to the MySQL application.

Open a new Explorer Window and click on the C: drive icon. Look in Program Files for the MySQL folder. If it’s not there, it might be in Program Files (x86). When you find it, double-click on the MySQL folder, then double-click on the MySQL Server 5.5 folder, then double-click on the bin folder.

The address bar should give you the path to MySQL. On my machine, that’s

C:\Program Files\MySQL\MySQL Server 5.5\bin

The next two steps will use that path to help configure MySQL.

5. Add MySQL’s path to your system variables.

The path is a semi-colon separated list of places on your computer where Windows will look when it’s asked to run a program. We’re going to add MySQL’s location to your path.

Click on Start > Control Panel. Then click on System and Security, then System, and then Advanced system settings. Click on the Environment Variables... button.

In the System variables panel, scroll down until you see the Path variable. Double-click on it to edit it.

Using the arrow keys, scroll all the way to the end of the Variable value box. Add the following: (Note the semi-colon at the front! The semi-colon separates this path from all the others in the list.)

;C:\Program Files\MySQL\MySQL Server 5.5\bin

Click OK and close out of all the Control Panel windows.
6. Add the MySQL daemon to the list of automatic startup programs.

   Open a command-line window by clicking Start and then typing cmd. Click on the cmd application to launch it.

   At the command line, type the following (note the quotation marks and double hyphen!):

   "C:\Program Files\MySQL\MySQL Server 5.5\bin\mysqld" --install

   The system should tell you that the service was successfully installed.

7. Restart your machine so that those changes can take effect.

   You will then be able to launch MySQL according to the instructions in the MySQL From the Command Line section of this course packet.
Securing a New MySQL Installation

A new MySQL installation is a bit like a brand-new house. Since there’s nothing of value in it, every possible door and window can be left wide, wide open. MySQL’s default installation, for example, includes users that have neither a name nor a password. That’s handy if you forgot your User ID and your password, but not otherwise.

Consequently, it’s not necessarily a good idea to leave your MySQL application in its initial state. You should secure a new MySQL installation by creating a new user ID for yourself and then either establishing passwords for important User IDs or else deleting them entirely.

In general, two user IDs will be important to keep and to use: 1) the new one that you’re about to invent for yourself; and 2) the so-called root user. Generally, root is the supreme administrative User ID that has all possible rights and privileges. For everyday use, though, it’s best to log in to MySQL under your own User ID, and to log in as root only when you need to do something that you yourself don’t even have privileges for.

1. Log in to MySQL as the root user.

   From the command line, log in to MySQL. (If you’re not sure how to get into the command line, take a quick glance at “Tips on Using MySQL from the Command Line” to get started.) Type:

   ```
   mysql -u root
   ```

2. Tell MySQL you’re going to use the mysql database.

   MySQL can manage several databases, of course, but the user-related data we want to change is in a database called mysql:

   ```
   mysql> USE mysql;
   ```

3. Establish passwords for the root user(s).

   Once you’re in MySQL, establish a password for the root user:

   ```
   mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD( 'choose_a_password' );
   Query OK, 0 rows affected (0.00 sec)
   ```

   If you’re running Windows, there is probably another root user to manage:

   ```
   mysql> SET PASSWORD FOR 'root'@'%' = PASSWORD( 'choose_a_password' );
   ```

   Alternatively, you can use the all-in-one, do-it-all-at-once commands:

   ```
   mysql> UPDATE user SET Password=PASSWORD( 'choose_a_password' ) -> WHERE User='root';
   ```
4. Remove users who do not have a User ID.
   Note: the User here is two single quotation marks back-to-back.
   
   mysql> DELETE FROM user WHERE User='';
   mysql> DELETE FROM db WHERE User='';

5. Create a new User ID for yourself.
   
   mysql> GRANT ALL ON *.* TO 'your_new_User_ID'
   -> IDENTIFIED BY 'your_new_password';
   mysql> FLUSH PRIVILEGES;
   mysql> \q
   <-- this is the standard way to exit MySQL

6. From the command line, log in to MySQL with your new user ID.
   
   mysql -u your_new_user_id -p
   Enter password: enter your new password; it won’t show up

   Welcome to the MySQL monitor. Commands end with ; or \g.
   Your MySQL connection id is 3
   Server version: 5.5.23 MySQL Community Server (GPL)

   Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

   Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

   Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

   mysql>
MySQL doesn’t have a Graphical User Interface (GUI) front-end because that’s the kind of thing you’re supposed to build in order to facilitate a user’s interaction with your data. Instead, MySQL is designed to be the behind-the-scenes, backend warehouse for your data. As a result, the easiest and simplest way to work with MySQL is via the barebones command line interface.

Sure, it’s a throwback to the 1970s and the days before mice allowed us to click and drag things. But it’s also the best way to learn how to work with your database because it’s the least mediated experience you can have with it. You’re seeing and manipulating data in its rawest form.

There are tools out there to help you when you need it (phpMyAdmin is a popular one), but I am a firm believer in understanding the rock-bottom, foundational concepts before using the tools.

A Brief Note on Command-Line Languages

Different systems use different command-line languages. Windows machines run a significantly updated version of MS-DOS, whereas Macintosh and Linux machines use some flavour of Unix. Gaining facility with your command-line language requires a little bit of practice but you don’t need to be a command-line expert in order to use MySQL on your system.

<table>
<thead>
<tr>
<th>action / question</th>
<th>MS-DOS command</th>
<th>Unix command</th>
</tr>
</thead>
<tbody>
<tr>
<td>which directory am I in?</td>
<td>cd</td>
<td>pwd (&quot;print working directory&quot;)</td>
</tr>
<tr>
<td>what’s the directory separator character?</td>
<td>\</td>
<td>/</td>
</tr>
<tr>
<td>how do I change directories?</td>
<td>cd directory name or chdir directory name</td>
<td>cd directory name</td>
</tr>
<tr>
<td>how do I go up one directory?</td>
<td>cd .. or chdir ..</td>
<td>cd ..</td>
</tr>
<tr>
<td>how do I change directories to an external device?</td>
<td>cd /d drive name</td>
<td>cd /Volumes/device name</td>
</tr>
<tr>
<td>how do I list the contents of a directory?</td>
<td>dir</td>
<td>ls (lowercase ell-ess)</td>
</tr>
<tr>
<td>what time is it?</td>
<td>time /t</td>
<td>date</td>
</tr>
<tr>
<td>what’s the date today?</td>
<td>date /t</td>
<td>date</td>
</tr>
</tbody>
</table>
1. Launch your system’s command-line interface.

On Windows, click the Start menu and type `cmd` in the search box (or in the Run box if you have an older version of Windows). Then click the `cmd` application in order to launch the command-line window.

On a Macintosh, launch Terminal, which is located in the Applications > Utilities folder. Just double-click it. You may want to keep Terminal in your Dock since you’ll be using it regularly.

2. If you’re planning to load data from an external file, make sure you’re in the right directory first.

MySQL knows which directory you were in when you launched it and MySQL will assume that any external file you try to load is located in that particular directory. If the file is not there, MySQL will have a difficult time finding it. The easiest thing to do is to get out of MySQL, change directories and then log in again. If you know you’re going to be accessing an external file, then, change directories first before you log in to MySQL.

Changing directories on Windows

Use the `cd` command to change directories to the place where your external file is located. If your document is on an external drive (like on a USB key in the E: drive, for example), you’ll need to use a special version of the `cd` command:

`cd /d E:`

That tells the `cd` command to use the special `/d` switch to change drives and to move to the E: drive. You can append a series of sub-directories to that command, of course. Just append the \ directory separator and the list of sub-directories.

In order to avoid repeatedly having to `cd` to an external drive when working with your MySQL databases, you may simply want to copy the tutorial installation files to your C: drive. You’ll still have to change directories before launching MySQL, but you might find the process easier.

Moreover, if your command-line interface is using a directory on your USB key, you won’t be able to eject it safely until you `cd` to something back on your C: drive.

Changing directories on Macintosh or *nix.

Use the `cd` command to change directories to the place where your external file is located. If your document is on an external drive (like a USB key, for example), you’ll need to use a special version of the `cd` command:

`cd /Volumes/name of USB key`
3. Log in to MySQL.

The mysql client (note the lack of capitalization there) is launched by typing its name on the command line. However, mysql requires a bit of extra information when it launches—in particular, it wants to know your username and it needs to know whether it should prompt you for your password (and yes it should).

We communicate our intentions to mysql via a few flags or switches (those terms are synonymous). Common switches are the -u switch (which signals that the following word is your username) and the -p switch (which tells mysql to prompt you on the next line for your password):

```
mysql  -u your_username_here -p
```

MySQL will then dutifully prompt you for your password, but the text you type will not be visible on screen, which is a minimal attempt to help secure your information from unscrupulous people who may be looking over your shoulder.

4. Select the database you want to use.

Just as word processors can control many different documents, MySQL can manipulate many different databases. And just as you need to open a particular word processing document before you can start editing it, you also need to tell MySQL which database you want to use:

```
USE database_name;
```

Remember the semi-colon at the end. That line is technically SQL and all SQL commands need to be terminated with a semi-colon (or, equivalently, the \g or \G character sequences).

If you start executing queries without having first selected a database to use, MySQL will complain a bit:

```
ERROR 1046 (3D000): No database selected
```

Note, too, that there's a login shortcut. You can actually specify the database you want to use as you launch MySQL from the command line:

```
mysql  -u your_username_here -p database_name
```

For example:

```
mysql -u root -p birdclub
```

That says, “log me in as root, prompt me for the password, and immediately begin using the birdclub database.”
5. Load external dump files with the source command or the slash-dot syntax.

Loading a so-called “dump file” is easy in MySQL and you’ll likely do it frequently during your future database voyages. MySQL expects these files to contain data that has been exported using the mysqldump utility.

You can use external dump files to initialize individual tables, to restore tables to a prior or a more pristine condition, to add data to your database and even (as we will do often) to create and load an entire database in one fell swoop. Dump files are also commonly used to transport entire databases from one web server to another.

MySQL has two synonymous syntaxes for loading external dump files: the source command and the slash-dot syntax:

```
mysql> \ external_filename
```

Or, equivalently:

```
mysql> source external_filename
```

Cautionary note: when you use the \ or source commands, MySQL will assume that those external files contain legitimate SQL commands. Any file that has been produced with the mysqldump utility (see #6 below) fits the bill of containing only legal SQL commands. You can create these files by hand, too (see “Other Bulk Loading Techniques”).

It’s possible to load files that contain just raw data (like those that have been exported from spreadsheet programs, for example) but that requires the LOAD DATA command. It has a slightly different syntax (see #7 below).

6. Export all or part of your database with the mysqldump utility.

The mysqldump utility is run from the operating system’s command line, not from within MySQL itself. Although it was once just a useful third-party utility, it is now bundled automatically with every installation of MySQL, so no one ever has to get along without it. And as a command-line utility, it can receive an impressive array of switches, all of which allow you to tailor and fine-tune your exported data to suit your current situation and needs.

Rather than detail all the mysqldump options here, there is another section of this course packet devoted solely to mysqldump and its various features. Check the Table of Contents to see where it’s located.

By the way, insatiably curious readers will want to know that there’s also a mysqlimport utility too. In practice, I rarely use it because I prefer to import directly within MySQL.
7. Import tab-delimited data or spreadsheet data (CSV format) with the **LOAD DATA** command.

MySQL can import tab-delimited or comma-separated value (CSV) files too, but there are a number of caveats you’ll need to keep an eye on. Rather than detail them here, this course packet contains a separate section on the **LOAD DATA** command. Check the Table of Contents to see where it is.

8. Quit MySQL with the `\q` or `quit` or `exit` commands.

There are really four ways to quit MySQL. Typing `\q` or `quit` or `exit` at the command line will terminate MySQL and will return you to your operating system’s command line prompt.

You can also just close the command line window. That will automatically terminate anything you had running at the command line. Obviously, if you intend to do some more work from the command line, closing the whole window might not be your best option.
Answers
1. What is the first name of the member whose last name is Martin?

```
mysql> SELECT first_name
    -> FROM member
    -> WHERE last = 'Martin';
```

2. How many birdclub members are there?

```
mysql> SELECT COUNT(*)
    -> FROM member;
```

**COUNT()** is called a *function*. You can think of functions as verbs because they generally perform actions like counting, adding, calculating or aggregating. And, like a verb, functions perform their actions on something—usually columns—and those column names are put inside parentheses that follow the function name. (See the list of functions elsewhere in this course packet for more information.)

In this case, we're using the * metacharacter, which is a wildcard character that colloquially means *everything*. In this context, we'd paraphrase our query as **select the count of all rows from the member table**. Knowing that every member occupies one row in the table, the number of rows equals the number of members.

You can rename columns if it makes the output more obvious. For example:

```
mysql> SELECT COUNT(*) AS 'Number of Members'
    -> FROM member;
```

**NOTE**: it's illegal in MySQL to have an extra space between the function name and the left parenthesis. Putting an illicit space there will generate an error:

```
mysql> SELECT COUNT( * ) FROM member;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '*) FROM member' at line 1
```

When you put a space between the function name and the left parenthesis, MySQL can no longer distinguish whether **COUNT()** is a function or whether **COUNT** is a column name. Assuming it's a column name, MySQL then treats the ( character as a grouping character, but gets totally flummoxed by the *) characters, at which point MySQL barfs.

3. Which birdclub members have no email addresses?
mysql> SELECT first, last
    -> FROM member
    -> WHERE email IS NULL;

Remember that **NULL** is a bit of a philosophical oddity. It isn’t a value, but rather it signifies the absence of a value. And since it’s impossible to tell if two nothingnesses are really the same nothingness, the equals sign (=) and the not equals sign (!=) do not work with **NULL**. Consequently, we have to use the **IS** and **IS NOT** keywords.

It’s possible to format this output more cleanly using functions and column renaming:

mysql> SELECT CONCAT(first, ' ', last) AS 'Members Having No Email'
    -> FROM member
    -> WHERE email IS NULL;

The **CONCAT()** function (it stands for **concatenation**, by the way; go ahead, look it up; you know you want to) appends one character string to the end of another. Notice that I put a blank space in the middle to separate the first name from the last name. But like any character string in MySQL, the blank space needs to be delimited with quotation marks otherwise MySQL wouldn’t know where the character string begins and where the character string ends.

4. How many birdclub members have listed a phone number?

mysql> SELECT COUNT(*) AS 'Phone Numbers'
    -> FROM member
    -> WHERE phone IS NOT NULL;

The question was, of course, “how many?” If you wanted to list the members who have listed their phones numbers, you could use a query like this:

mysql> SELECT CONCAT(first, ' ', last) AS 'Members Listing Phone Numbers'
    -> FROM member
    -> WHERE phone IS NOT NULL
    -> ORDER BY last, first;

5. List the birdclub member guides.

mysql> SELECT CONCAT(first, ' ', last) AS 'Birdclub Guides'
    -> FROM member
    -> WHERE guide = 'yes'
    -> ORDER BY last, first;

---

4 George Clooney’s character in *O Brother, Where Art Thou?* might assert that it’s two weeks from everywhere.
This query alphabetizes the guides, which wasn’t technically part of the question, but it’s useful sometimes to know how to do that.

6. What is the scientific name of the black-billed magpie?

Although you can use the LIKE keyword to answer this question, the most precise query uses = to search for an exact match:

```sql
mysql> SELECT scientific
-> FROM bird
-> WHERE common = 'black-billed magpie';
```

Remember that since string matches are case insensitive in MySQL (that is, in a WHERE clause MySQL does not distinguish between uppercase and lowercase letters), we need not capitalize the search string exactly as it appears in the database.

What if you wanted to do a case-sensitive search (that is to say, one in which the capitalization was identical)? You can add the conversation keyword BINARY before the search string:

```sql
mysql> SELECT scientific
-> FROM bird
-> WHERE common = BINARY 'black-billed magpie';
```

7. What is the common name of Spizella breweri?

```sql
mysql> SELECT common
-> FROM bird
-> WHERE scientific = 'Spizella breweri';
```

Again, remember that if you need to match the capitalization perfectly, add the BINARY keyword.

**Slightly Harder Queries:**

1. Which two birdclub members have the longest first names?

```sql
mysql> SELECT first, last
-> FROM member
-> ORDER BY LENGTH(first) DESC
-> LIMIT 2;
```

<-- Remember, DESC means DESCENDING
Obviously, his query doesn't tell us whether the names are the same length or how long they are.

2. List all the birdclub members in alphabetical order.

```sql
mysql> SELECT first, last -> FROM member -> ORDER BY last, first;
```

If you need prettier output, it's easy to modify the result:

```sql
mysql> SELECT CONCAT(first, ' ', last) AS Members -> FROM member -> ORDER BY last, first;
```

Notice that I've renamed the column using the `AS` keyword. Since my new column name contains only one word, though, I don't actually need to wrap it in quotation marks. MySQL's default behaviour is to assume that table and column names are only one word long. This one fits that criterion, so I don't need to wrap it in quotation marks. If the new column name were longer, though, then I'd need quotation marks.

3. Which bird has the shortest scientific name?

```sql
mysql> SELECT common -> FROM bird -> ORDER BY LENGTH(scientific) ASC -> LIMIT 1;
```

If you want more data, just add more columns to the `SELECT` clause:

```sql
mysql> SELECT common, scientific, LENGTH(scientific) -> FROM bird -> ORDER BY LENGTH(scientific) -> LIMIT 1;
```

4. Write a multiple-table query to display information for each bird sighting. List the member's name, the bird that person saw, and the date. Try to list the sightings in chronological order.

```sql
mysql> SELECT CONCAT(member.first, ' ', member.last) AS Person, -> bird.common, sighting.date -> FROM member, sighting, bird -> WHERE sighting.person = member.id
```
Notice that there are some zero-filled dates in the `sightings` table. Although it might look odd, it’s common database practice to put zeroes into dates when information is missing. Read the “Top 8 MySQL Tips” or “From Novice to Guru” in this course packet to learn more about zero-filled dates.
2012 NFL Prospects Exercise Query Answers

Query the Database

1. List all the players in alphabetical order (last name, first name).

   mysql> SELECT last_name, first_name
          -> FROM player
          -> ORDER BY last_name, first_name;

2. List all the players by Overall Ranking.

   mysql> SELECT rank_overall, first_name, last_name
          -> FROM player
          -> ORDER BY rank_overall;

3. List all the players by Overall Ranking in reverse order.

   mysql> SELECT rank_overall, first_name, last_name
          -> FROM player
          -> ORDER BY rank_overall DESC;

4. List all the players who went to school in Saskatchewan (SK).

   mysql> SELECT player.*
          -> FROM player, school
          -> WHERE school.state = 'SK'
          -> AND school.id = player.school;

   Yeah, I know Saskatchewan isn't a state. But when I built the database, I didn’t realize that there was anyone from a Canadian school in the mix.

   Notice here, too, that player.* selects every column from the player table.

5. Show all the players from Florida (FL). Sort them in alphabetical order.

   mysql> SELECT player.*
          -> FROM player, school
          -> WHERE school.state = 'FL'
          -> AND school.id = player.school
          -> ORDER BY player.last_name, player.first_name;

6. Show the top 25 defensive players (according to their overall rank).

   mysql> SELECT player.first_name, player.last_name, rank_overall
          -> FROM player, position
          -> ORDER BY rank_overall DESC;
-> WHERE player.position = position.id
-> AND position.offense_defense = 'Defense'
-> ORDER BY rank_overall
-> LIMIT 25;

7. List all the players in alphabetical order who are ranked in the top 3 at their position.

```
mysql> SELECT first_name, last_name
-> FROM player
-> WHERE rank_at_position <= 3
-> ORDER BY last_name, first_name;
```

8. Show all the players in the top 150 overall rankings who are NOT in the top 3 at their position and whose last names begin with "S" and who went to a school in the state of Florida.

```
mysql> SELECT *
-> FROM player, school
-> WHERE player.rank_at_position > 3
-> AND player.rank_overall <= 150
-> AND player.last_name LIKE 'S%'
-> AND player.school = school.id
-> AND school.state = 'FL';
```

This query has a very large `WHERE` clause, but that’s not a problem. As long as one of the sub-clauses contains a “bridge” between the two tables (`player.school = school.id`), we can choose appropriate query criteria from either table.

The order in which we list those criteria doesn’t affect the “correctness” of the query so, generally, the order is completely arbitrary. MySQL may rearrange those criteria behind the scenes in order to perform the query more efficiently, and we can always do that too. It’s called *query optimization*.

One optimization strategy is to list the criteria from the smallest tables first (that is, from the tables with the fewest rows), working our way toward the tables with the greatest number of rows. In our case, the `school` table contains the fewest rows, so we could optimize this query by listing the `school.state = 'FL'` criterion first and then working our way towards the criteria from the `player` table.

**Summary and Grouping**

1. List the various positions and the number of prospects at each position in descending order.

```
mysql> SELECT position.position,
-> COUNT(player.position) AS 'Number of Prospects'
-> FROM player, position
```
WHERE player.position = position.id
GROUP BY player.position
ORDER BY COUNT(player.position) DESC;

Notice that we are allowed to use a function in the ORDER BY clause.

2. Which state has the highest number of prospects?

This query is similar to the last one; we're just SELECTing based on different fields and LIMITing the number of rows in the result. But the strategy is the same.

mysql> SELECT school.state,
COUNT(school.state) AS 'Prospects From State'
FROM player, school
WHERE player.school = school.id
GROUP BY school.state
ORDER BY COUNT(school.state) DESC
LIMIT 1;
The MovieLens Exercise Query Answers

1. List the movies in alphabetical order.

```sql
mysql> SELECT *
     -> FROM movie
     -> ORDER BY title ASC;
```

2. List the movies by year and, within each year, in alphabetical order.

```sql
mysql> SELECT year, title
     -> FROM movie
     -> ORDER BY year, title;
```

3. What is the average movie rating of all movies?

```sql
mysql> SELECT AVG(rating)
     -> FROM rating;
```

4. List the movies whose title contains the word `vampire`.

```sql
mysql> SELECT *
     -> FROM movie
     -> WHERE title LIKE '%vampire%';
```

Remember two things: 1) keyword searches are case insensitive so it doesn't matter how we capitalize `vampire`; and 2) the % character is a wildcard character and it requires the LIKE keyword.

The % wildcard matches any number of characters. We put it on both sides of our search word so that we match `vampire` no matter where it might occur in the title.

Searching for `%vampire` would match the word only at the end of the title. (It allows for wildcard characters before `vampire` but not after.)

Searching for `vampire%` would match the word only at the beginning of the title. (It allows for wildcard characters after `vampire` but not before.)

5. List only the oldest movie whose title contains the word `vampire`.

```sql
mysql> SELECT title, year
     -> FROM movie
     -> WHERE title LIKE '%vampire%'
     -> ORDER BY year
     -> LIMIT 1;
```

---

Remember two things: 1) keyword searches are case insensitive so it doesn't matter how we capitalize `vampire`; and 2) the % character is a wildcard character and it requires the LIKE keyword.
Grouping Queries

6. Display the number of movies released during each year.

```sql
mysql> SELECT year, COUNT(year)
    -> FROM movie
    -> GROUP BY year;
```

7. Which year released the most movies? Display the year and the number of movies.

```sql
mysql> SELECT year, COUNT(year) AS Number
    -> FROM movie
    -> GROUP BY year
    -> ORDER BY Number DESC
    -> LIMIT 1;
```

This query is slightly tricky because we want to use the `ORDER BY` clause but that clause cannot operate on a function like `COUNT()`. Therefore, we use a subtle trick: we rename the results of the `COUNT()` column and `ORDER BY` the newly renamed column. MySQL is okay with this trick because we avoid calling the function a second time; we're merely sorting by the results it has already given us.

Note: Although we're renaming the `COUNT()` column, we're renaming it to a string that contains only one word. Therefore, we need not wrap the new name in quotation marks. If our column name contained more than one word (because the space character is a delimiter in MySQL), we'd need to use quotation marks to define the name of the column.
1818-1819 Factory Workers Exercise Query Answers

Query the Database

1. Which worker under age 18 had the greatest number of sick days?

```
mysql> SELECT * FROM worker
    -> WHERE age <=18
    -> ORDER BY days_sick_last_year DESC
    -> LIMIT 1;
```

Using a LIMIT 1 to find the single worker who had the most sick days is a logical strategy, but in case two or more workers happened to tie for the greatest number of sick days, I almost always run a query like this a second time and change the LIMIT to 4 or 6 or 8.

2. Does the number of rows in the worker table match the sum of the number_of_workers column from the factory table? What can we do about that?

No, they don’t match. Those two numbers are different. Querying each table produces slightly different answers:

```
mysql> SELECT COUNT(*) FROM worker;
```

```
mysql> SELECT SUM(number_of_workers) FROM factory;
```

Doing redundant queries like this might seem like a waste of time, but I think they’re occasionally useful. In this case, there might be a variety of explanations for the discrepancy. If the numbers from the factory table were compiled in 1818-1819, the error might be a simple mathematical mistake. On the other hand, if those numbers were compiled by the 20th-century economist, then perhaps the column number_of_workers should be dropped from the factory table. After all, one of the rules of databases is “don’t store what you can calculate.” Regardless, we should be skeptical about these numbers until we do some more investigating.

We could begin by double-checking the numbers. Try grouping the counts by factory number:

```
mysql> SELECT factory.id, factory.number_of_workers,
    -> COUNT(worker.factory_num)
    -> FROM factory, worker
    -> WHERE factory.id = worker.factory_num
    -> GROUP BY worker.factory_num;
```
It seems that certain factories have slightly inaccurate counts. Next, we could check for duplicate entries from those factories in the worker table. If we don't find duplicates (and given that many first names are simply initials, it might be impossible to determine whether two records are duplicates anyway), then the factory.number_of_workers column might simply be wrong and its values can safely be ignored.

The numbers here are pretty close and you might decide that this particular problem is inconsequential. But it's useful to have some strategies at your disposal for finding and correcting errors like these because eventually you might have an important dataset whose errors are too big to ignore.

3. From the worker table, how many people work at factory #1220? Query the factory table to learn more about factory #1220. How do you interpret the result?

```sql
mysql> SELECT COUNT(*)
    -> FROM worker
    -> WHERE factory_num = 1220;
```

As you can see, no factory #1220 is listed in the factory table. There is, however, a factory #122 and perhaps a stray zero was accidentally appended to the factory number in the worker table. However, given that there are factories #1231 and #1232, it's possible that #1220 is correct and the #122 in the factory table is wrong.
Managing & Using
MySQL

George Reese, Randy Jay Yarger & Tim King
with Hugh E. Williams
SECOND EDITION

Managing and Using MySQL

Part I. Introduction

1. MySQL: The World's Most Popular Database System

George Reese, Randy Jay Yarger, and Tim King
with Hugh E. Williams

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo
CHAPTER 3

SQL According to MySQL

The Structured Query Language (SQL) is used to read and write to MySQL databases. Using SQL, you can search for, enter, modify, or delete data. SQL is the most fundamental tool you will need for your interactions with MySQL. Even if you are using some application or graphical user interface to access the database, somewhere under the hood that application is generating SQL.

SQL is a sort of “natural” language. In other words, an SQL statement should read—at least on the surface—like a sentence of English text. This approach has both benefits and drawbacks, but the end result is a language unlike traditional programming languages such as C, Java, or Perl.

SQL Basics

SQL is “structured” in the sense that it follows a very specific set of rules. A computer program can parse a formulated SQL query easily. In fact, the O’Reilly book *lex & yacc* by John Levine, Tony Mason, and Doug Brown implements an SQL grammar to demonstrate the process of writing a program to interpret language! A *query* is a fully specified command sent to the database server, which then performs the requested action. Here’s an example of an SQL query:

```
SELECT name FROM people WHERE name LIKE 'Stac%'
```

As you can see, this statement reads almost like a form of broken English: “Select names from a list of people where the names are like Stac.” SQL uses few of the formatting and special characters that are typically associated with computer languages.

The SQL Story

IBM invented SQL in the 1970s shortly after Dr. E. F. Codd invented the concept of a relational database. From the beginning, SQL was an easy-to-learn, yet powerful language. It resembles a natural language such as English, so it is less daunting to a non-technical person. In the 1970s, even more than today, this advantage was important.
There were no casual hackers in the early 1970s. No one grew up learning BASIC or building web pages in HTML. The people programming computers were people who knew everything about how a computer worked. SQL was aimed at the army of non-technical accountants and business and administrative staff who would benefit from being able to access the power of a relational database.

SQL was so popular with its target audience, in fact, that in the 1980s, Oracle Corporation launched the world’s first publicly available commercial SQL system. Oracle SQL was a huge hit and spawned an entire industry built around SQL. Sybase, Informix, Microsoft, and several other companies have since come forward with their implementations of SQL-based relational database management systems (RDBMSs).

When Oracle and its first competitors hit the scene, SQL was still relatively new and there was no standard. It was not until 1989 that the ANSI standards body issued the first public SQL standard. These days, the standard is referred to as SQL89. That new standard, unfortunately, did not go far enough into defining the technical structure of the language. Thus, even though the various commercial SQL languages were drawing closer together, differences in syntax still made it nontrivial to switch among implementations. It was not until 1992 that the ANSI SQL standard came into its own.

The 1992 standard is called both SQL92 and SQL2. The SQL2 standard expanded the language to accommodate as many of the proprietary extensions added by the commercial implementations as possible. Most cross-DBMS tools have standardized on SQL2 as the way in which they talk to relational databases. Due to the extensive nature of the SQL2 standard, however, relational databases that implement the full standard are very complex and resource intensive.

SQL2 is not the last word on the SQL standard. With the growing popularity of object-oriented database management systems (OODBMS) and object-relational database management systems (ORDBMS), there has been increasing pressure to capture support for object-oriented database access in the SQL standard. The recent SQL3 standard is the answer to this problem.

When MySQL came along, it took a new approach to the business of database server development. Instead of manufacturing another giant RDBMS and risk having nothing more to offer than the big guys, Monty created a small, fast implementation of the most commonly used SQL functionality. Over the years, that basic functionality has grown to support just about anything you might want to do with most database applications.

**The Design of SQL**

As we mentioned earlier, SQL resembles a human language more than a computer language because it has a simple, defined imperative structure. Much like an English
sentence, individual SQL commands, called "queries," can be broken down into language parts. Consider the following examples:

```
CREATE TABLE people (name CHAR(10))
```

```
INSERT INTO people VALUES ('me')
```

```
SELECT name FROM people WHERE name LIKE '%e'
```

Most implementations of SQL, including MySQL, are case insensitive. Specifically, it does not matter how you type SQL keywords as long as the spelling is correct. The previous CREATE example could just as well be:

```
CREATE TABLE people (name CHAR(10))
```

The case insensitivity extends only to SQL keywords.* In MySQL, names of databases, tables, and columns are case-sensitive. This case sensitivity is not necessarily true for all database engines. Thus, if you are writing an application that should work against all databases, you should assume that names are case sensitive.

This first element of an SQL query is always a verb. The verb expresses the action you wish the database engine to take. While the rest of the statement varies from verb to verb, they all follow the same general format: you name the object upon which you are acting and then describe the data you are using for the action. For example, the query `CREATE TABLE people (name CHAR(10))` uses the verb `CREATE`, followed by the object `TABLE`. The rest of the query describes the table to be created.

An SQL query originates with a client (the application that provides the façade through which a user interacts with the database). The client constructs a query based on user actions and sends the query to the SQL server. The server must then process the query and perform the specified action. Once the server has done its job, it returns some value or set of values to the client.

Because the primary focus of SQL is to communicate actions to the database server, it does not have the flexibility of a general-purpose language. Most of the functionality of SQL concerns input to and output from the database: adding, changing, deleting, and reading data. SQL provides other functionality, but always with an eye towards how it can be used to manipulate the data within the database.

**Sending SQL to MySQL**

You can send SQL to MySQL using a variety of mechanisms. The most common way is through one of the programming APIs described in Part III. For the purposes of

---

* For the sake of readability, we capitalize all SQL keywords in this book. We recommend this convention as a solid "best practice" technique.
this chapter, however, we recommend you use the interactive command-line tool, `mysql`. When you run this program at the command line, it prompts you for SQL:

```
[09:04pm] carthage$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.22.29

Type 'help' for help.
```

```
mysql>
```

The previous `mysql` command says to connect to the MySQL server on the local machine as the user root (the `-u` option) with the client prompting you for a password (the `-p` option). Another option, `-h`, enables you to connect to MySQL servers on remote machines:

```
[09:04pm] carthage$ mysql -u root -h db.imaginary.com -p
```

There is absolutely no relationship between operating-system usernames and MySQL usernames. In other words, MySQL keeps its own list of users, and a MySQL administrator needs to add new users to MySQL independently of the host on which they reside. No one, therefore, has an account on a clean MySQL installation except root. This root is not the same root as your Unix root account. As a general rule, you should never connect to MySQL as root except when performing database administration tasks. If you have a clean installation of MySQL that you can afford to throw away, it is useful to connect as root for the purposes of this chapter so you can create and drop databases. Otherwise, you will have to connect to MySQL as whatever username has been assigned to you.

Once `mysql` is running, you can enter your SQL commands all on a single line or split them across multiple lines. MySQL waits for a semicolon before executing the SQL:

```
mysql> SELECT book_number
       -> FROM book
       -> ;
```

```
+---------------------+
<table>
<thead>
<tr>
<th>book_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
+---------------------+
3 rows in set (0.00 sec)
```

With the `mysql` command line, you generally get a command history depending on how your client tools were compiled. If a command history is compiled into your `mysql` client, you can use the up and down arrows on your keyboard to navigate through previously executed SQL commands.

* MySQL also accepts `\g` at the end of an SQL statement to indicate that the SQL should be executed.
Database Creation

To get started using MySQL, you need to create a database. First, let’s take a look at the databases that come with a clean MySQL installation using the `SHOW DATABASES` command. Upon installation of MySQL 3.23.40, the following tables already exist:

```
mysql> SHOW DATABASES;
+----------------+
| Database       |
+----------------+
| mysql          |
| test           |
+----------------+
2 rows in set (0.37 sec)
```

The first database, `mysql`, is MySQL’s system database, which you will learn more about in Chapter 5. The second database, `test`, is a play database you can use to learn MySQL and run tests against. You may find other databases on your server if you are not dealing with a clean installation. For now, however, we want to create a new database to illustrate the use of the MySQL `CREATE` statement:

```sql
CREATE DATABASE TEMPDB;
```

and then to work with the new database `TEMPDB`:

```sql
USE TEMPDB;
```

Finally, you can delete that database by issuing the `DROP DATABASE` command:

```sql
DROP DATABASE TEMPDB;
```

You can create new objects using the `CREATE` statement and destroy things using the `DROP` statement, just as we used them here.

Table Management

You should now feel comfortable connecting to a database on a MySQL server. For the rest of the chapter, you can use either the `test` database that comes with MySQL or your own play database. Using the `SHOW` command, you can display a list of tables in the current database the same way you used it to show databases. In a brand new installation, the test database has no tables. The following shows the output of the `SHOW TABLES` command when connected to the `mysql` system database:

```
mysql> USE mysql;
Database changed
mysql> SHOW TABLES;
+---------------+
| Tables_in_mysql |
+---------------+
| columns_priv   |
| db            |
| func          |
| host          |
| topics        |
| user          |
```

Table Management | 27
<table>
<thead>
<tr>
<th>tables_priv</th>
<th>user</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|             | +-----+
|             | 6 rows in set (0.00 sec)

These are the six system tables MySQL requires to do its work. To see what one of these tables looks like, you can use the DESCRIBE command:

```sql
mysql> DESCRIBE db;
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>char(60)</td>
<td>PRI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Db</td>
<td>char(64)</td>
<td>PRI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>char(16)</td>
<td>PRI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drop_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grant_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>References_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alter_priv</td>
<td>enum('N','Y')</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13 rows in set (0.36 sec)

This output describes each column in the table showing its data type, whether it can contain null values, what kind of key it is, any default values, and extra information. If all this means nothing to you, don’t worry. We will describe each of these elements as the chapter progresses.

You should now be ready to create your first table. First, connect back to the test database that comes with a clean MySQL install:

```sql
USE test;
```

Make sure you connect to the test database first, because you definitely do not want to add tables to the mysql database. The table, a structured container of data, is the most basic concept of a relational database. Before adding data to a table, you must define the table’s structure. Consider the following layout:

```
<table>
<thead>
<tr>
<th>people</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>address</td>
</tr>
<tr>
<td>id</td>
</tr>
</tbody>
</table>
```

Not only does the table contain the names of the columns, but it also contains the types of each field as well as any additional information the fields may have. A field’s data type specifies what kind of data the field can hold. SQL data types are similar to
data types in other programming languages. The full SQL standard allows for a large range of data types. MySQL implements most of them as well as a few MySQL-specific types.

The general syntax for table creation is:

```
CREATE TABLE table_name (  
    column_name1 type [modifiers],  
    [ , column_name2 type [modifiers]]  
)
```

What constitutes a valid identifier (a name for a table or column) varies between DBMSs. MySQL allows up to 64 characters in an identifier, supports the character $ in identifiers, and lets identifiers start with a valid number. More importantly, however, MySQL considers any valid letter for your local character set to be a valid letter for identifiers.

A column is the individual unit of data for a row within a table. A table may have any number of columns, but too many columns can make a table inefficient. This is where good database design, discussed in Chapter 7, becomes important. By creating properly normalized tables, you can join tables to perform searches across data housed in more than one table. We discuss the mechanics of a join later in the chapter.

Consider the following CREATE statement:

```
CREATE TABLE USER (  
    USER_ID BIGINT UNSIGNED NOT NULL PRIMARY KEY,  
    USER_NAME CHAR(10) NOT NULL,  
    LAST_NAME VARCHAR(30),  
    FIRST_NAME VARCHAR(30),  
    OFFICE CHAR(2) NOT NULL DEFAULT 'NY');
```

This statement creates a table called USER with five columns: USER_ID, USER_NAME, LAST_NAME, FIRST_NAME, and OFFICE. After each column name comes the data type for that column, followed by any modifiers.

The NOT NULL modifier indicates that the column may not contain any null values. If you try to assign a null value to that column, SQL will generate an error. Actually, there are a couple of exceptions to this rule. First, if the column is defined as AUTO_INCREMENT, a null value will cause a value to be generated automatically. (We cover auto-incrementing later in the chapter.) The second exception is when you specify a default value for a column, as we have for the OFFICE column in the previous example. In this case, the OFFICE column is assigned the default value of 'NY' when you assign a null value. (We will discuss data types and the PRIMARY KEY modifier later in this chapter.)

Like most things in life, destruction is much easier than creation. The command to drop a table from the database is:

```
DROP TABLE table_name
```
This command completely removes all traces of that table from the database. MySQL removes all data within the specified table from existence. If you have no backups of the table, you absolutely cannot recover from this action. The moral of this story is to always keep backups and be very careful about dropping tables. You will thank yourself for it someday.

With MySQL, you can specify more than one table to delete by separating the table names with commas. For example, DROP TABLE people, animals, plants would delete the three named tables. You can also use the IF EXISTS modifier to avoid an error, should the table not exist when you try to drop it. This modifier is useful for huge scripts designed to create a database and all its tables. Before creating the database, run a DROP TABLE IF EXISTS table_name command.

MySQL Data Types

In a table, each column has a type. As we mentioned earlier, an SQL data type is similar to a data type in traditional programming languages. While many languages define a bare-minimum set of types necessary for completeness, SQL goes out of its way to provide types such as DATE that will be useful to everyday users. You could store a DATE type in a more basic numeric type, but having a type specifically dedicated to the nuances of date processing adds to SQL’s ease of use—one of SQL’s primary goals.

Chapter 16 provides a full reference of SQL types supported by MySQL. Table 3-1 is an abbreviated listing of the most common types.

Table 3-1. Common MySQL data types (see Chapter 16 for a full list)

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>An integer value. MySQL allows an INT to be either signed or unsigned.</td>
</tr>
<tr>
<td>REAL</td>
<td>A floating-point value. This type offers a greater range and more precision than the INT type, but it does not have the exactness of an INT.</td>
</tr>
<tr>
<td>CHAR(length)</td>
<td>A fixed-length character value. No CHAR fields can hold strings greater in length than the specified value. Fields of lesser length are padded with spaces. This type is the most commonly used in any SQL implementation.</td>
</tr>
<tr>
<td>VARCHAR(length)</td>
<td>A variable-length character value.</td>
</tr>
<tr>
<td>TEXT(length)</td>
<td>A variable-length character value.</td>
</tr>
<tr>
<td>DATE</td>
<td>A standard date value. The DATE type stores arbitrary dates for the past, present, and future.</td>
</tr>
<tr>
<td>TIME</td>
<td>A standard time value. This type stores the time of day independent of a particular date. When used together with a date, a specific date and time can be stored. MySQL additionally supplies a DATETIME type that stores date and time together in one field.</td>
</tr>
</tbody>
</table>

MySQL supports the UNSIGNED attribute for all numeric types. This modifier forces the column to accept only positive (unsigned) numbers. Unsigned fields have an
upper limit that is double that of their signed counterparts. For instance, an unsigned TINYINT—MySQL’s single-byte numeric type—has a range of 0 to 255 instead of the 
−128 to 127 range of its signed counterpart.

MySQL provides more types than those mentioned in Table 3-1. In day-to-day pro-
gramming, however, you will use these types most often. The size of the data you 

wish to store plays a large role in the design of your MySQL tables.

**Numeric Types**

Before you create a table, you should know what kind of data you wish to store in 
the table. Beyond obvious decisions about whether your data is character-based or 
numeric, you should know the approximate size of the data to be stored. If it is a 
numeric field, what is its maximum possible value? What is its minimum possible 
value? Could that change in the future? If the minimum is always positive or zero, 
you should consider an unsigned type. You should always choose the smallest 
numeric type that can support your largest conceivable value. If, for example, you 
have a field that represents the population of a state, use an unsigned INT field. No 
state can have a negative population. Furthermore, an unsigned INT field is certainly 
large enough to represent a state’s population, unless that population grows to be 
roughly the current population of the entire Earth.

**Character Types**

Managing character types is a little more complicated. Not only do you have to 
worry about the minimum and maximum string lengths, but you also have to worry 
about the average size and the amount of variation. For our current purposes, an 
index is a field or combination of fields on which you plan to search—basically, the 
fields in your WHERE clause. Indexing is, however, much more complicated, so we will 
provide further details later in the chapter. What’s important to note here is that 
indexing on character fields works best when the field is a fixed length. If there is lit-
tle or, preferably, no variation in the length of your character-based fields, then a 
CHAR type is appropriate. An example of a good candidate for a CHAR field is a country 
code. The ISO provides a comprehensive list of standard two-character representa-
tions of country codes (US for the U.S., FR for France, etc.). Because these codes are 
always exactly two characters, a CHAR(2) is the best way to maintain the country code 
based on the ISO representation.

*States and provinces do not work the same way in internationalized applications. If you want to write an 
application that works in an international environment, make the columns for state and province codes 
CHAR(3), because Australia uses three-character state codes. Also note that there is a three-character ISO 
country-code standard.*
A value does not need to be constant length to use a CHAR field. It should, however, have very little variance. Phone numbers, for example, can be stored safely in a CHAR(13) field even though phone number lengths vary from nation to nation. The variance is little enough that there is no point in making a phone number field variable in length. Keep in mind that with a CHAR field, no matter how big the actual string being stored is, the field always takes up exactly the number of characters specified as the field’s size—no more, no less. Any difference between the length of the text being stored and the length of the field is made up by padding the value with spaces. While the few potential extra characters being wasted on a subset of the phone number data is not anything to worry about, you do not want to be wasting much more.

Variable-length text fields are appropriate for text fields with widely varying lengths. A good, common example of a field that demands a variable-length data type is a web URL. Most web addresses are relatively short (e.g., http://www.ora.com, http://www.imaginary.com, http://www.mysql.com) and consequently do not pose problems. Occasionally, however, you will run into web addresses such as:

http://www.winespectator.com/Wine/Spectator/
_notes|5527293926834323221480431354?Xv11=&Xr5=&Xv1=&type-region-
search-code=&Xa14=flora+springs&Xv4=

If you construct a CHAR field large enough to hold this URL, you will be wasting a significant amount of space for almost every other URL being stored. A variable-length field lets you define a field length that can store the odd, long-length value while not wasting all that space for the common, short-length values.

Variable-length text fields in MySQL use only as much space as necessary to store an individual value into the field. A VARCHAR(255) field that holds the string “hello world,” for example, takes up only 12 bytes (1 byte for each character plus an extra byte to store the length).

MySQL varies from the ANSI standard by not padding VARCHAR fields. Any extra spaces are removed from a value before it is stored.

You cannot store strings with lengths greater than the field length you have specified. With a VARCHAR(4) field, you can store at most a string with four characters. If you attempt to store the string “happy birthday,” MySQL will truncate the string to “happ.” The downside is that there is no way to store the odd string that exceeds your designated field size. Table 3-2 shows the storage space required by the different text data types to store the 144-character Wine Spectator URL shown earlier, the space required to store an average-sized 30-character URL, and the maximum string size for that data type.
Table 3-2. The storage space required by the different MySQL character types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Storage for 144-char string</th>
<th>Storage for 30-char string</th>
<th>Maximum string size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(150)</td>
<td>150</td>
<td>150</td>
<td>255</td>
</tr>
<tr>
<td>VARCHAR(150)</td>
<td>145</td>
<td>31</td>
<td>255</td>
</tr>
<tr>
<td>TINYTEXT(150)</td>
<td>145</td>
<td>31</td>
<td>255</td>
</tr>
<tr>
<td>TEXT(150)</td>
<td>146</td>
<td>32</td>
<td>65535</td>
</tr>
<tr>
<td>MEDIUMTEXT(150)</td>
<td>147</td>
<td>33</td>
<td>16777215</td>
</tr>
<tr>
<td>LONGTEXT(150)</td>
<td>148</td>
<td>34</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

In this table, note that storage requirements grow 1 byte at a time for the variable-length types of MEDIUMTEXT and LONGTEXT. This growth is due to the space required to store the size in variable-length fields. TEXT uses an extra byte to store the potentially greater length of the text it contains. Similarly, MEDIUMTEXT uses an extra 2 bytes over VARCHAR, and LONGTEXT uses an extra 3 bytes.

If after years of uptime with your database, you find that the world has changed and a field that once comfortably existed as a VARCHAR(25) must now hold strings as long as 30 characters, you are not out of luck. MySQL provides a command called ALTER TABLE that enables you to redefine a field type without losing any data:

```
ALTER TABLE mytable MODIFY mycolumn LONGTEXT
```

**Binary Data Types**

MySQL provides a set of binary data types that closely mirror their character counterparts. The MySQL binary types are CHAR BINARY, VARCHAR BINARY, TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB. The practical distinction between character types and their binary counterparts is the concept of encoding. Binary data is basically a chunk of data that MySQL makes no effort to interpret. Character data, on the other hand, is assumed to represent textual data from human alphabets. It is thus encoded and sorted based on rules appropriate to the character set in question. On an ASCII system, MySQL sorts binary data in a case-sensitive, ASCII order.

**Enumerations and Sets**

MySQL provides two other special kinds of types. The ENUM type allows you specify (enumerate) at table creation a list of possible values that can be inserted into that field. For example, if you have a column named fruit into which you want to allow only the values apple, orange, kiwi, or banana, you would assign this column the type ENUM:

```
CREATE TABLE meal(meal id INT NOT NULL PRIMARY KEY,
    fruit ENUM('apple', 'orange', 'kiwi',
               'banana'))
```
When you insert a value into that column, it must be one of the specified fruits. Because MySQL knows ahead of time which values are valid for the column, it can abstract them to some underlying numeric type. In other words, instead of storing apple in the column as a string, MySQL stores it internally as a single-byte number. However, you still refer to it as apple in a query or when you retrieve the value from MySQL. You also use apple when you call the table or view results from the table.

The MySQL SET type works in the same way, except it lets you store multiple values in a field at the same time and uses bits instead of bytes.

**Other Kinds of Data**

Every piece of data you will ever encounter can be stored using numeric or character types. Technically, you could even store numbers as character types. Just because you can do so, however, does not mean you should. Consider, for example, storing a date in the database. You could store that value as a Unix-style BIGINT or as a combination of several columns for the day, month, and year. How do you look for rows with a date value greater than two days after a specific date? Either you calculate the numeric representation of that date or employ a complex operation for a simple query mixing day, month, and year values.

Isn't all of that a major pain? Wouldn't it be nice if MySQL handled all of these issues for you? In fact, MySQL does. It provides several complex data types to help with abstract common concepts. It supports the concept of dates through the DATE data type. Other such data types include DATETIME and TIMESTAMP.

**Indexing**

While MySQL has better performance than any of the larger database servers, some problems still call for careful database design. For instance, if we had a table with millions of rows of data, a search for a specific row would take a long time. Most database engines allow indexes to aid in such searches.

Indexes help the database store data in a way that makes for quicker searches. Unfortunately, you sacrifice disk space and modification speed for the benefit of quicker searches. The most efficient use of indexes is to create an index for columns on which you tend to search the most. MySQL supports the following syntax for creating an index for a table:

```sql
CREATE INDEX index_name ON tablename (column1, column2, ..., columnN)
```
MySQL also lets you create an index at the same time you create a table using the following syntax:

```
CREATE TABLE material (id INT NOT NULL,
name CHAR(50) NOT NULL,
resistance INT,
melting_pt REAL,
INDEX index1 (id, name),
UNIQUE INDEX index2 (name))
```

The previous example creates two indexes for the table. The first index—named index1—consists of both the id and name fields. The second index includes only the name field and specifies that values for the name field must always be unique. If you try to insert a field with a name held by a row already in the database, the insert will fail. Generally, you should declare all fields in a unique index as NOT NULL.

Even though we created an index for name by itself, we did not create an index for just id. If we did want such an index, we would not need to create it—it is already there. When an index contains more than one column (for example: name, rank, and serial_number), MySQL reads the columns in order from left to right. Because of the structure of the index MySQL uses, any subset of the columns from left to right are automatically created as indexes within the “main” index. For example, name by itself and name and rank together are both indexes created “for free” when you create the index name, rank, serial_number. An index of rank by itself or name and serial_number together, however, is not created unless you explicitly create it yourself.

MySQL also supports the ANSI SQL semantics of a special index called a primary key. In MySQL, a primary key is a unique key with the name PRIMARY. By calling a column a primary key at creation, you are naming it as a unique index that will support table joins. The following example creates a cities table with a primary key of id:

```
CREATE TABLE cities (id INT NOT NULL PRIMARY KEY,
name VARCHAR(100),
pop MEDIUMINT,
founded DATE)
```

Before you create a table, you should determine which fields, if any, should be keys. As we mentioned above, any fields that will support joins are good candidates for primary keys. See Chapter 7 for a detailed discussion on how to design your tables with good primary keys.

ANSI SQL supports a special kind of key called a foreign key. Foreign keys help protect database integrity by enabling the database to manage things such as the deletion of rows with dependent relationships in other tables. Though MySQL supports the ANSI syntax for foreign keys, it does not actually use them to perform integrity checking in the database. This is a situation in which the introduction of a feature would cause a slowdown in performance with little real benefit. Applications themselves should generally worry about foreign key integrity.
Managing Data

The first thing you will probably want to do with a newly created table is add data to it. Once the data is in place, you need to maintain it—add to it, modify it, and perhaps even delete it.

Inserts

Adding a row to a table is one of the more straightforward concepts in SQL. You have already seen several examples of it in this book. MySQL supports the standard SQL INSERT syntax:

```
INSERT INTO table_name (column1, column2, ..., columnN)
VALUES (value1, value2, ..., valueN)
```

Under this syntax, you specify the columns followed by the values to populate those columns for the new row. When inserting data into numeric fields, you can insert the value as is; for all other fields, you must wrap them in single quotes. For example, to insert a row of data into a table of addresses, you might issue the following command:

```
INSERT INTO addresses (name, address, city, state, phone, age)
VALUES('Irving Forbush', '123 Mockingbird Lane', 'Corbin', 'KY',
      '(800) 555-1234', 26)
```

In addition, the escape character—\—by default—enables you to escape single quotes and other literal instances of the escape character:

```
# Insert info for the directory Stacie's Directory which
# is in c:\Personal\Stacie
INSERT INTO files (description, location)
VALUES ('Stacie's Directory', 'C:\Personal\Stacie')
```

MySQL allows you to leave out the column names as long as you specify a value for every column in the table in the order they were specified in the table's CREATE call. If you want to use the default values for a column, however, you must specify the names of the columns for which you intend to insert nondefault data. For example, if the earlier files table had contained a column called size, the default value would be used for Stacie's Directory. MySQL allows you to specify a custom default value in the table's CREATE call. If you do not have a default value set up for a column, and that column is NOT NULL, you must include that column in the INSERT statement with a non-NULL value.

Newer versions of MySQL support a nonstandard INSERT call for inserting multiple rows at once:

```
INSERT INTO foods VALUES (NULL, 'Oranges', 133, 0, 2, 39),
                        (NULL, 'Bananas', 122, 0, 4, 29),
                        (NULL, 'Liver', 232, 3, 15, 10)
```
While these nonstandard syntaxes supported by MySQL are useful for quick system administration tasks, you should not use them when writing database applications unless you really need the speed benefit they offer. As a general rule, you should stick as close to the ANSI SQL2 standard as MySQL will let you. By doing so, you make certain that your application can run against any other database in the future. Being flexible is especially critical for people with mid-range database needs because such users generally hope to become people with high-end database needs.

Another nonstandard syntax supported by MySQL enables you to specify the column name and value together:

```sql
INSERT INTO book SET title='The Vampire Lestat', author='Anne Rice';
```

Finally, you can insert data by using the data from some other table (or group of tables) to populate your new table. For example:

```sql
INSERT INTO foods (name, fat)
SELECT food_name, fat_grams FROM recipes
```

You should note that the number of columns in the INSERT call matches the number of columns in the SELECT call. In addition, the data types for the INSERT columns must match the data types for the corresponding SELECT columns. Finally, the SELECT clause in an INSERT statement cannot contain an ORDER BY modifier and cannot be selected from the same table where the INSERT occurs.

### Sequence Generation

The best kind of primary key is one that has absolutely no meaning in the database except to act as a primary key. Primary keys are the tools used to identify rows uniquely in a relational database. When you use information such as a username or an email address as a primary key, you are in effect saying that the username or email address is somehow an intrinsic part of who that person is. If that person ever changes his username or email address, you will have to go to great lengths to ensure the integrity of the data in the database. Consequently, it is a better design principle to use meaningless numbers as primary keys.

To achieve this, simply make a numeric primary key that increments every time you insert a new row. Looking at the cities table shown earlier, the first city you insert would have an id of 1, the second 2, the third 3, and so on. To successfully manage this primary key sequencing, you need some way to guarantee that a number can be read and incremented by only one client at a time. You accomplish this task by making the primary key field `AUTO_INCREMENT`. 
When you create a table in MySQL, you can specify at most one column as AUTO_INCREMENT. When you do this, you can have this column automatically insert the highest current value plus 1 for that column when you insert a row and specify NULL or 0 for that row's value. The AUTO_INCREMENT columns must be indexed. The following command creates the cities table with an AUTO_INCREMENT id field:

```sql
CREATE TABLE cities (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
                     name VARCHAR(100),
                     pop INT,
                     founded DATE)
```

The first time you insert a row, the id field for your first row will be 1 as long as you use NULL or 0 for that field in the INSERT statement. For example, this command takes advantage of the AUTO_INCREMENT feature:

```sql
INSERT INTO cities (id, name, pop)
VALUES (NULL, 'Houston', 3000000)
```

If no other values are in that table when you issue this command, MySQL will set this field to 1, not NULL (remember, it cannot be NULL). If other values are present in the table, the value inserted will be one greater than the largest current value for id.

You can also implement sequences by referring to the value returned by the LAST_INSERT_ID() function and doing your own incrementing:

```sql
UPDATE table_name SET id=LAST_INSERT_ID( id+1 );
```

The AUTO_INCREMENT attribute may be supplied for at most one column of an integer type in a table. In addition to being an integer type, the column must be either a primary key or the sole column in a unique index. When you attempt an insert into a table with such an integer field and fail to specify a value for that field (or specify a NULL value), a value of one greater than the column’s current maximum value will be automatically inserted.

Chapter 17 contains reference material on the LAST_INSERT_ID() function.

**Updates**

The insertion of new rows into a database is just the start of data management. Unless your database is read-only, you will probably also need to make periodic changes to the data. The standard SQL modification statement looks like this:

```sql
UPDATE table_name
SET column1=value1, column2=value2, ..., columnN=valueN
[WHERE clause]
```

* You can seed AUTO_INCREMENT to start at any arbitrary number by specifying the seed value at the end of the CREATE statement. To start incrementing at 1025, for example, you would add AUTO_INCREMENT = 1025 after the closing parentheses of the CREATE TABLE statement.
You specifically name the table you want to update and the values you want to assign in the SET clause, and then identify the rows to be affected in the WHERE clause. If you fail to specify a WHERE clause, MySQL will update every row in the table.

In addition to assigning literal values to a column, you can also calculate the values. You can even calculate the value based on a value in another column:

```
UPDATE years
SET end_year = begin_year + 5
```

This command sets the value in the end_year column equal to the value in the begin_year column plus 5 for each row in that table.

### The WHERE Clause

The previous section introduced one of the most important SQL concepts, the WHERE clause. In SQL, a WHERE clause enables you to pick out specific rows in a table by specifying a value that must be matched by the column in question. For example:

```
UPDATE bands
SET lead_singer = 'Ian Anderson'
WHERE band_name = 'Jethro Tull'
```

This UPDATE specifies that you should change only the lead_singer column for the row where band_name is identical to Jethro Tull. If the band_name column is not a unique index, that WHERE clause may match multiple rows. Many SQL commands employ WHERE clauses to help pick out the rows on which you wish to operate. Because the columns in the WHERE clause are columns on which you search, you should generally have indexes created around whatever combinations you commonly use. We discuss the kinds of comparisons you can perform in the WHERE clause later in the chapter.

### Deletes

Deleting data is a straightforward operation. You simply specify the table followed by a WHERE clause that identifies the rows you want to delete:

```
DELETE FROM table_name [WHERE clause]
```

As with other commands that accept a WHERE clause, the WHERE clause is optional. If you omit it, you will delete all of the records in the table! Of all the destructive commands in SQL, this is the easiest one to issue by mistake.

MySQL 4.0 has introduced a new, dangerous form of DELETE that supports the ability to delete from multiple tables with a single command:

```
DELETE table1, table2, ..., tablen
FROM table1, table2, ... tablen
[WHERE clause]
```
The `FROM` clause in this syntax does not mean the same thing as it does in the simpler form. In other words, it does not list the tables from which rows are deleted—it lists the tables referenced in the `WHERE` clause. If you are familiar with the `SELECT` statement, it works exactly the same as the `FROM` clause in `SELECT` statements. The tables you are deleting from are listed directly after the `DELETE` statement:

```sql
DELETE Author, Address
FROM Author, Book, Address
WHERE Author.author_id = Address.address_id
AND Author.author_id = Book.author_id
AND Book.publish_date < 1980;
```

This statement deletes all the authors and any address information you have for those authors in the `Address` table for every author with books published before 1980. The old books will remain in the `Book` table, because `Book` was not named after the `DELETE` keyword. We further cover the complexities of the `WHERE` clause later in the chapter.

### Queries

The last common SQL command, `SELECT`, enables you to view the data in the database. This action is by far the most common action performed in SQL. While data entry and modifications do happen on occasion, most databases spend the vast majority of their lives serving up data for reading. The general form of the `SELECT` statement is as follows:

```sql
SELECT column1, column2, ..., columnN
FROM table1, table2, ..., tableN
WHERE clause
```

This syntax is certainly the most common way to retrieve data from any SQL database. The `SELECT` statement enables you to identify the columns you want from one or more tables. The `WHERE` clause identifies the rows with the data you seek.

Of course, there are variations for performing complex and powerful queries. (We cover the full range of the `SELECT` syntax in Chapter 15.) The simplest form is:

```sql
SELECT 1;
```

This simple, though completely useless query returns a result set with a single row containing a single column with the value of 1. A more useful version of this query might be something like:

```sql
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| test       |
+------------+
1 row in set (0.01 sec)
```
The expression `DATABASE()` is a MySQL function that returns the name of the current database. (We will cover functions in more detail later in the chapter.) Nevertheless, you can see how simple SQL can provide a quick-and-dirty way of finding out important information.

Most of the time, however, you should use slightly more complex queries that help you pull data from a table in the database. The first part of a `SELECT` statement enumerates the columns you wish to retrieve. You may specify a `*` to say that you want to select all columns. The `FROM` clause specifies which tables those columns come from. The `WHERE` clause identifies the specific rows to be used and enables you to specify how to join two tables.

**Joins**

Joins put the “relational” in relational databases by enabling you to relate the data in one table with data in other tables. The basic form of a join is sometimes described as an *inner join*. Joining tables is a matter of specifying equality in columns from two tables:

```sql
SELECT book.title, author.name
FROM author, book
WHERE book.author = author.id
```

This query pulls columns from two different tables where a relationship exists between rows in the two tables. Specifically, this query looks for situations in which the value of the `author` column in the `book` table matches the `id` value in the `author` table. Consider a database in which the `book` table looks like Table 3-3 and the `author` table looks like Table 3-4.

**Table 3-3. A book table**

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Author</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Green Mile</td>
<td>4</td>
<td>894</td>
</tr>
<tr>
<td>2</td>
<td>Guards, Guards!</td>
<td>2</td>
<td>302</td>
</tr>
<tr>
<td>3</td>
<td>Imzadi</td>
<td>3</td>
<td>354</td>
</tr>
<tr>
<td>4</td>
<td>Gold</td>
<td>1</td>
<td>405</td>
</tr>
<tr>
<td>5</td>
<td>Howling Mad</td>
<td>3</td>
<td>294</td>
</tr>
</tbody>
</table>

**Table 3-4. An author table**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Citizen</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Isaac Asimov</td>
<td>US</td>
</tr>
<tr>
<td>2</td>
<td>Terry Pratchett</td>
<td>UK</td>
</tr>
<tr>
<td>3</td>
<td>Peter David</td>
<td>US</td>
</tr>
<tr>
<td>4</td>
<td>Stephen King</td>
<td>US</td>
</tr>
<tr>
<td>5</td>
<td>Neil Gaiman</td>
<td>UK</td>
</tr>
</tbody>
</table>
An inner join creates a virtual table by combining the fields of both tables for rows that satisfy the query in both tables. In our example, the query specifies that the author field of the book table must be identical to the id field of the author table. The query’s result would look like Table 3-5.

<table>
<thead>
<tr>
<th>Book title</th>
<th>Author name</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Green Mile</td>
<td>Stephen King</td>
</tr>
<tr>
<td>Guards, Guards!</td>
<td>Terry Pratchet</td>
</tr>
<tr>
<td>Imzadi</td>
<td>Peter David</td>
</tr>
<tr>
<td>Gold</td>
<td>Isaac Asimov</td>
</tr>
<tr>
<td>Howling Mad</td>
<td>Peter David</td>
</tr>
</tbody>
</table>

Neil Gaiman is nowhere to be found in these results. He is left out because there is no value for his author.id value found in the author column of the book table. In other words, he did not write any of the books in our database! An inner join contains only those rows that match the query exactly. We will discuss the concept of an outer join later in the chapter for situations in which we have an author in the database who does not have a book in the database.

**Aliasing**

When you use column names that are fully qualified with their table and column name, the names can grow to be quite unwieldy. In addition, when referencing SQL functions (which will be discussed later in the chapter), you will likely find it cumbersome to refer to the same function more than once within a statement. You can get around these issues by using aliases. An alias is usually a shorter and more descriptive way of referring to a cumbersome name. You can use it anywhere in the same SQL statement in place of the longer name. For example:

```sql
# A column alias
SELECT long_field_names_are_annoying AS myfield
FROM table_name
WHERE myfield = 'Joe'
# A table alias
SELECT people.names, tests.score
FROM tests, really_long_people_table_name AS people
```

**Ordering and Grouping**

The results from a SELECT are, by default, indeterminate in the order they will appear. Fortunately, SQL provides some tools for imposing discipline on this seemingly random list: ordering and grouping.
Basic ordering

You can tell a database to order any results you see by a certain column. For example, if you specify that a query should order the results by last_name, then the results will appear alphabetized according to the last_name value. Ordering is handled by the ORDER BY clause:

    SELECT last_name, first_name, age
    FROM people
    ORDER BY last_name, first_name

In this situation, we are ordering by two columns. You can order by any number of columns. You can also use the special ORDER BY RAND( ) clause to return results in a random order.

If you want to see things in reverse order, add the DESC (descending) keyword:

    ORDER BY last_name DESC

The DESC keyword applies only to the field that comes directly before it. If you are sorting on multiple fields, only the field directly before DESC is reversed; the others are sorted in ascending order.

Localized sorting

Sorting is actually a complex problem for applications that need to run on computers all over the world. The rules for sorting strings vary from alphabet to alphabet, even when two alphabets use mostly the same symbols. MySQL handles the problem of sorting by making it dependent on the character set used by the MySQL engine. Out of the box, the default character set is ISO-8859-1 (Latin-1). MySQL uses the sorting rules for Swedish and Finnish with ISO-8859-1.

To change the sorting rules, you change the character set. First, you need to make sure the correct character set was compiled into the server when you compiled and installed MySQL. With the proper character set compiled into the server, you can change the default character set by launching the server with the argument --default-character-set=CHARSET.

Because of the simplicity of the English alphabet, the use of a single set of sorting rules MySQL associates with ISO-8859-1 does not affect English sorting. This is not true, however, for languages such as Swedish and German, which both use the ISO-8859-1 character set. Swedish sorts å after z, while German sorts ä before a. The default rules therefore fail German users.

MySQL lets you address this problem by creating custom character sets. When you compile the driver, you can compile in support for whatever character sets you desire as long as you have a configuration file for that character set. This file contains the characters that make up the character set and the rules for sorting them. You can write your own or use the ones that come with MySQL.
The real problem here is that MySQL incorrectly associates sorting rules with character sets. A character set is nothing more than a grouping of characters with a related purpose. Nothing about the ISO-8859-1 character set implies sorting for Swedes, Italians, Germans, or anyone else. When working with MySQL, however, you need to remember that sorting rules are directly tied to the character set.

**Grouping**

Grouping lets you group rows with matching values for a specific column into a single row in order to operate on them together. You usually do this to perform aggregate functions on the results. We will go into functions a little later in the chapter.

Consider the following:

```sql
mysql> SELECT name, rank, salary FROM people;
+----------+-----+-------+
| name     | rank| salary|
|----------+-----+-------|
| Jack Smith| Private| 23000 |
| Jane Walker| General| 125000 |
| June Sanders| Private| 22000 |
| John Barker| Sergeant| 45000 |
| Jim Castle | Sergeant| 38000 |
+----------+-----+-------+
5 rows in set (0.01 sec)
```

If you want to get a list of different ranks, you can use the `GROUP BY` clause to get a full account of the ranks:

```sql
mysql> SELECT rank FROM people GROUP BY rank;
+-----+
| rank|
+-----+
| General|
| Private|
| Sergeant|
+-----+
3 rows in set (0.01 sec)
```

You should not, however, think of these results as simply a listing of the different ranks. The `GROUP BY` clause actually groups all of the rows matching the `WHERE` clause (in this case, every row) based on the `GROUP BY` clause. The two privates are thus grouped together into a single row with the rank `Private`. The two sergeants are similarly aggregated. With the individuals grouped according to rank, you can find out the average salary for each rank. Again, we will further discuss the functions you see in this example later in the chapter.
mysql> SELECT rank, AVG(salary) FROM people GROUP BY rank;

+------------+-----------+
| rank       | AVG(salary)|
|------------+-----------|
| General    | 125000.0000 |
| Private    | 22500.0000 |
| Sergeant   | 41500.0000 |
+------------+-----------+
3 rows in set (0.04 sec)

Here you see the true power of grouping. This query uses an aggregate function, AVG(), to operate on all of the rows grouped together for each row. In this case, the salaries of the two privates (23000 and 22000) are grouped together in the same row, and the AVG() function is applied to them.

The power of ordering and grouping combined with the utility of SQL functions enables you to do a great deal of data manipulation even before you retrieve the data from the server. However, you should take great care not to rely too heavily on this power. While it may seem more efficient to place as much processing load as possible onto the database server, this is not really the case. Your client application is dedicated to the needs of a particular client, while the server is shared by many clients. Because of the greater amount of work a server already has to do, it is almost always more efficient to place as little load as possible on the database server. MySQL may be the fastest database around, but you do not want to waste that speed on processing that can be handled by client applications.

**Limiting Results**

A WHERE clause is not the only way to constrain the results you see from a query. MySQL provides two other common mechanisms: HAVING and LIMIT.

You will most commonly use HAVING with the GROUP BY clause we just described. Like a WHERE clause, it defines your result set based on some set of calculations. Unlike a WHERE clause, it performs these calculations after your results have been retrieved from the tables in which they are stored. A WHERE clause, for example, scans the table in the database and pulls all records matching the WHERE clause. A HAVING clause, on the other hand, looks only at rows that have been pulled from a database after they have been extracted. The following query goes one step beyond our previous search for the average salary of different ranks in getting the average salaries only for ranks with an average salary greater than $100,000.

```sql
mysql> SELECT rank, AVG(salary) FROM people
    > GROUP BY rank HAVING AVG(salary) > 100000.00;

+------------+-----------+
| rank       | AVG(salary)|
|------------+-----------|
| General    | 125000.0000 |
+------------+-----------+
1 row in set (0.04 sec)
```
Restricting the result set in a WHERE clause would make no sense. If it were to be valid SQL, it would work on the entire table! Instead, we first want to perform the select and then find only those groups in the result set whose average salary is greater than $100,000. The HAVING clause enables us to perform that further restriction. More importantly, consider a case with both a WHERE clause and a HAVING clause:

```
mysql> SELECT rank, AVG(salary) FROM people
    > WHERE rank <> 'Private'
    > GROUP BY rank HAVING AVG(salary) > 100000.00;
```

```
+------------+------------------+
| rank       | AVG(salary)      |
+------------+------------------+
| General    | 125000.0000      |
+------------+------------------+
1 row in set (0.02 sec)
```

Because the HAVING clause executes on the results of the query, the average is calculated only for generals and sergeants—not the excluded privates.

Sometimes an application is looking for only the first few rows that match a query. Limiting queries can help prevent bogging down the network with unwanted results. MySQL enables an application to limit the number of results through a LIMIT clause in a query:

```
SELECT * FROM people ORDER BY name LIMIT 10;
```

To get the last 10 people from the table, you can use the DESC keyword. If you want people from the middle, however, you have to get a bit trickier; you need to specify the number of the first record you want to see (record 0 is the first record, 1 the second) and the number of rows you want to see:

```
SELECT * FROM people ORDER BY name LIMIT 19, 30;
```

This sample displays records 20 through 49. The 19 in the LIMIT clause tells MySQL to start with record 19, which is the twentieth record. The 30 then tells MySQL to return the next 30 records.

**SQL Operators**

So far, we have used the = operator for the obvious task of verifying that two values in a WHERE clause equal one another. Other fairly basic operations include <>, >, <, <=, and >=. Note that MySQL allows you to use either <> or != for “not equal.” Table 3-6 contains a full set of simple SQL operators.

*Table 3-6. The simple SQL operators supported by MySQL*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Arithmetic</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Arithmetic</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Arithmetic</td>
<td>Multiplication</td>
</tr>
</tbody>
</table>
Table 3-6. The simple SQL operators supported by MySQL (continued)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>Arithmetic</td>
<td>Division</td>
</tr>
<tr>
<td>=</td>
<td>Comparison</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;&gt; or !=</td>
<td>Comparison</td>
<td>Not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Comparison</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Comparison</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Comparison</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Comparison</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>AND</td>
<td>Logical</td>
<td>And</td>
</tr>
<tr>
<td>OR</td>
<td>Logical</td>
<td>Or</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical</td>
<td>Negation</td>
</tr>
</tbody>
</table>

MySQL operators have the following order of precedence:

1. BINARY
2. NOT !
3. - (unary minus)
4. * / %
5. + -
6. << >>
7. &
8. |
9. < <= > >= = <=> <> IN IS LIKE REGEXP RLIKE
10. BETWEEN
11. AND &&
12. OR ||

Logical Operators

SQL’s logical operators—AND, OR, and NOT—let you build more dynamic WHERE clauses. The AND and OR operators specifically let you add multiple criteria to a query:

```sql
SELECT USER_NAME
FROM USER
WHERE AGE > 18 AND STATUS = 'RESIDENT';
```

This sample query provides a list of all users who are residents and are old enough to vote. In other words, it finds every resident 18 years or older.
You can build increasingly complex queries and override MySQL's order of precedence with parentheses. The parentheses tell MySQL which comparisons to evaluate first:

```sql
SELECT USER_NAME
FROM USER
WHERE (AGE > 18 AND STATUS = 'RESIDENT')
OR (AGE > 18 AND STATUS = 'APPLICANT');
```

In this more complex query, we are looking for anyone currently eligible to vote as well as people who might be eligible in the near future. Finally, you can use the NOT operator to negate an entire expression:

```sql
SELECT USER_NAME
FROM USER
WHERE NOT (AGE > 18 AND STATUS = 'RESIDENT');
```

In this case, negation provides all the users who are not eligible to vote.

**Null's Idiosyncrasies**

Null is a tricky concept for most people new to databases to understand. As in other programming languages, null is not a value, but the absence of a value. This concept is useful, for example, if you have a customer profiling database that gradually gathers information about your customers as they offer it. When you first create a record, for example, you may not know how many pets the customer has. You want that column to hold NULL instead of 0 so you can tell the difference between customers with no pets and customers whose pet ownership is unknown.

The concept of null gets a little funny when you use it in SQL calculations. Many programming languages use null as simply another kind of value. In Java, the following syntax evaluates to true when the variable is null and false when it is not:

```java
str == null
```

The similar expression in SQL, `COL = NULL`, is neither true nor false—it is always NULL, no matter what the value of the COL column. The following query will therefore not act as you might expect:

```sql
SELECT title FROM book WHERE author = NULL;
```

Because the WHERE clause will never evaluate to true no matter what value is in the database for the author column, this query always provides an empty result set—even when you have author columns with NULL values. To test for "nullness," use the IS NULL and IS NOT NULL operators:

```sql
SELECT title FROM book WHERE author IS NULL;
```

MySQL also provides a special operator called the null-safe operator `<=>,` which you can use when you are not sure if you are dealing with null values. It returns true if both sides are null or false if both sides are not null:
mysql> SELECT 1 <=> NULL, NULL <=> NULL, 1 <=> 1;
+---------------+---------------+---------------+
| 1 <=> NULL    | NULL <=> NULL | 1 <=> 1       |
+---------------+---------------+---------------+
| 0             | 1             | 1             |
+---------------+---------------+---------------+
1 row in set (0.00 sec)

This simple query shows how the null-safe operator works with a variety of inputs.

**Membership Tests**

Sometimes applications need to check if a value is a member of a set of values or within a particular range. The IN operator helps with the former:

```sql
SELECT TITLE FROM BOOK WHERE AUTHOR IN ('Stephen King', 'Richard Bachman');
```

This query will return the titles of all books written by Stephen King.* Similarly, you can check for all books by authors other than Stephen King with the NOT IN operator.

To determine if a value is in a particular range, use the BETWEEN operator:

```sql
SELECT TITLE FROM BOOK WHERE BOOK_ID BETWEEN 1 AND 100;
```

Both of these simple examples could, of course, be replicated with the basic operators. The Stephen King check, for example, could have been done by using the = operator and an OR:

```sql
SELECT title
FROM book
WHERE author = 'Stephen King' OR author = 'Richard Bachman';
```

The check on book IDs could also have been done with an OR clause using the >= and <= or > and < operators. As your queries get more complex, however, membership tests can help you build both readable and better-performing queries than those you might create with the basic operators.

**Pattern Matching**

We provided a peek at ANSI SQL pattern matching earlier with the query:

```sql
SELECT name FROM people WHERE name LIKE 'Stack'
```

Using the LIKE operator, we compared a column value (name) to an incomplete literal ('Stack'). MySQL supports the ability to place special characters into string literals that match like wild cards. The % character, for example, matches any arbitrary number of characters, including no character at all. The above SELECT statement would therefore match Stacey, Stacie, Stacy, and even Stac. The character _ matches any single character. Stac_y would match only Stacey. Stac__ would match Stacie and Stacey, but not Stacy or Stac.

* Richard Bachman is a pseudonym used by Stephen King for some of his books.
Pattern-matching expressions should never be used with the basic comparison operators. Instead, they require the LIKE and NOT LIKE operators. It is also important to remember that these comparisons are case-insensitive except on binary columns.

MySQL supports a non-ANSI kind of pattern matching that is actually much more powerful using the same kind of expressions to which Perl programmers and grep users are accustomed. MySQL refers to these as extended regular expressions. Instead of LIKE and NOT LIKE, these operators must be used with the REGEXP and NOT REGEXP operators. MySQL provides synonyms for these: RLIKE and NOT RLIKE. Table 3-7 contains a list of the supported extended regular expression patterns.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any single character.</td>
<td>Stac.. matches any value containing the characters &quot;Stac&quot; followed by two characters of any value.</td>
</tr>
<tr>
<td>[]</td>
<td>Matches any character in the brackets. You can also match a range of characters.</td>
<td>[Ss]tacey matches values containing both &quot;Stacey&quot; and &quot;stacey.&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[a-zA-Z] matches values containing one instance of any character in the English (unaccented) portion of the Roman alphabet.</td>
</tr>
<tr>
<td>*</td>
<td>Matches zero or more instances of the character that precedes it.</td>
<td>Ap*e matches values containing &quot;Aple,&quot; &quot;Apple,&quot; &quot;Appple,&quot; etc.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Los.*es matches values containing the strings &quot;Los&quot; and &quot;es&quot; with anything in between.</td>
</tr>
<tr>
<td>^</td>
<td>What follows must come at the beginning of the value.</td>
<td>^Stacey matches values that start with &quot;Stacey.&quot;</td>
</tr>
<tr>
<td>$</td>
<td>What precedes it must end the value.</td>
<td>cheese$ matches any value ending in the string &quot;cheese.&quot;</td>
</tr>
</tbody>
</table>

You should note a couple of important facts about extended regular expressions. Unlike basic pattern matching, MySQL extended regular expressions are case sensitive. They also do not require a match for the entire string. The pattern simply needs to occur somewhere within the value. Consider the following example:

```sql
mysql> SELECT * FROM BOOK;

+----------+--------------------------+----------+
| BOOK_ID  | TITLE                    | AUTHOR   |
+----------+--------------------------+----------+
| 1        | Database Programming...  | George   |
| 2        | JavaServer Pages         | Hans     |
| 3        | Java Distributed...      | Jim      |
+----------+--------------------------+----------+
3 rows in set (0.01 sec)
```

In this table, we have three books from O'Reilly's Java series. The interesting thing about the Java series is that all books begin with or end with the word "Java." The first sample query checks for any titles LIKE 'Java':

```sql
SELECT * FROM BOOK WHERE TITLE LIKE 'Java';
```
mysql> SELECT TITLE FROM BOOK WHERE TITLE LIKE 'Java';
Empty set (0.01 sec)

Because LIKE looks for an exact match of the pattern specified, no rows match—none of the titles are exactly 'Java'. To find out which books start with the word Java using simple patterns, we need to add a %:

mysql> SELECT TITLE FROM BOOK WHERE TITLE LIKE 'Java%';
+------------------+
| TITLE            |
| JavaServer Pages |
| Java Distributed Computing |
+------------------+
2 rows in set (0.00 sec)

This query had two matches because only two of the books had titles that matched Java% exactly. The extended regular expression matches, however, are not exact matches. They simply look for the expression anywhere within the compared value:

mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP 'Java';
+------------------+
| TITLE            |
| Database Programming with JDBC and Java |
| JavaServer Pages |
| Java Distributed Computing |
+------------------+
3 rows in set (0.06 sec)

By simply changing the operator from LIKE to REGEXP, we changed how it matches things. Java appears somewhere in each of the titles, so the query returns all the titles. To find only the titles that start with the word Java using extended regular expressions, we need to specify that we are interested in the start:

mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP '^Java';
+------------------+
| TITLE            |
| JavaServer Pages |
| Java Distributed Computing |
+------------------+
2 rows in set (0.01 sec)

The same thing applies to finding titles with Java at the end:

mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP 'Java$';
+------------------+
| TITLE            |
| Database Programming with JDBC and Java |
+------------------+
1 row in set (0.00 sec)

The extended regular expression syntax is definitely much more complex than the simple pattern matching of ANSI SQL. In addition to the burden of extra complexity, you
should also consider the fact that MySQL extended regular expressions do not work in most other databases. When you need complex pattern matching, however, they provide you with power that is simply unsupportable by simple pattern matching.

**Advanced Features**

Using the SQL presented thus far in this chapter should handle 90% of your database programming needs. On occasion, however, you will need some extra power not available in the basic SQL functionality. We close out the chapter with a discussion of a few of these features.

**Full Text Searching**

MySQL introduced the ability to search on text elements within a text field in Version 3.23.23 through a special index called a FULLTEXT index. It specifically enables you to do something like:

```
INSERT INTO Document (url, page_text )
VALUES ('index.html', 'The page contents.');
SELECT url FROM Document WHERE MATCH ( page_text ) AGAINST ('page');
```

INSERT adds a row to a Document table containing the URL of a web page and its text content. SELECT then looks for the URLs of all documents with the word page embedded in their text.

**The Basics**

The magic behind full text searching lies in a FULLTEXT index. The CREATE statement for the Document table might look like this:

```
CREATE TABLE Document ( 
url VARCHAR(255) NOT NULL PRIMARY KEY,
page_text TEXT NOT NULL,
FULLTEXT ( page_text )
);
```

The FULLTEXT index enables you to search the index using words or phrases that will not match exactly and then weigh the relevance of any matches. As with other indexes, you can create multicolumn FULLTEXT indexes:

```
CREATE TABLE Document ( 
url VARCHAR(255) NOT NULL PRIMARY KEY,
title VARCHAR(100) NOT NULL,
page_text TEXT NOT NULL,
FULLTEXT ( title, page_text )
);
```

With this table structure, you can now search for documents that have the word MySQL anywhere in the title or body of the page. You must keep your searches structured against the index, not against the columns. In other words, you can match
against title and page_text together with this table, but you cannot look for words that exist only in the title unless you create a separate FULLTEXT index on it alone. Your combined search will look like the following:

```sql
SELECT url FROM Document
WHERE MATCH ( title, page_text ) AGAINST ('MySQL');
```

### Relevance values

The search that occurs here is a natural language search against the text in the specified columns. It is case insensitive. The result of the match is actually a relevance value that MySQL uses to rank the results. By default, MySQL shows the results with the most relevant results listed first and eliminates only those results with no relevance at all.

```sql
mysql> SELECT url FROM Document
    -> WHERE MATCH ( title, page_text ) AGAINST ('java');
```

<table>
<thead>
<tr>
<th>url</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.html</td>
</tr>
</tbody>
</table>

You can use the relevance to your advantage. To get a better picture of the relevance values, you can execute the following query:

```sql
mysql> SELECT url, WHERE MATCH ( title, page_text ) AGAINST ('java')
    -> FROM Document;
```

<table>
<thead>
<tr>
<th>url</th>
<th>MATCH ( title, page_text ) AGAINST ('java')</th>
</tr>
</thead>
<tbody>
<tr>
<td>index.html</td>
<td></td>
</tr>
<tr>
<td>java.html</td>
<td>1.8016148151891</td>
</tr>
<tr>
<td>perl.html</td>
<td>0</td>
</tr>
<tr>
<td>c.html</td>
<td>0</td>
</tr>
</tbody>
</table>

In this case, the `index.html` file is a web page about MySQL and `java.html` is about how MySQL and Java work together. As you might expect, the results show that `index.html` has no relevance to Java, while `java.html` has quite a bit of relevance.

In the above example, you can include MATCH in both the SELECT clause and the WHERE clause without incurring any extra overhead. MySQL is smart enough to notice that the two matches are identical and thus execute them a single time. Using MATCH in both places is useful when you want to make use of relevance.

You might expect that a match against MySQL would turn up high relevance for both documents. In reality, however, it turns up zero relevance. Because the phrase MySQL is present in more than half the rows, it is considered a *stopword* and thus discounted. A stopword is simply a word with no value for text matching. Common
stopwords are “the” and “but.” They are critical to achieving meaningful results in a full text search. Unfortunately, our sample database has four rows about MySQL and thus MySQL itself is considered a stopword.

**Boolean mode**

MySQL 4.0.1 introduced the ability to perform more complex searches with full text searching using much the same syntax you would use in Internet search engines. These complex searches are called Boolean mode searches. To execute a Boolean mode search, you use the same syntax except:

- Add `IN BOOLEAN MODE` to your `AGAINST` clause.
- Add modifiers in the text you are searching with.

For example:

```sql
SELECT url, title FROM Document
WHERE MATCH ( title, page_text ) AGAINST ( 'MySQL -Java' IN BOOLEAN MODE );
```

This query enables you to look for all documents that include MySQL but exclude Java. Common words that might otherwise be used as stopwords can be used in Boolean mode searches. Without any modifiers, a term in a Boolean mode search is considered optional. You can modify your Boolean mode searches with the following operators:

+ The word must be in the matched index.
- The word must not be in the matched index.
~ The word’s relevance is inverted. This operator is useful for removing words that MySQL is giving high relevance to, but you do not want in your hits. Unlike the - operator, returned rows can have the term in question.
< Decreases the word’s contribution to the overall relevance of a match.
> Increases a word’s contribution to the overall relevance of a match.
* This operator comes after the word it modifies. It works much like a wildcard, matching any words that start with the specified word.

() Groups words into subexpressions.

Now you can run complex queries such as:

```sql
SELECT url, title FROM Document
WHERE MATCH ( title, page_text ) AGAINST ( '+MySQL -optimiz* +Java <Perl>' IN BOOLEAN MODE );
```

This query asks for all of the rows about MySQL and either Java or Perl that do not have words beginning with `optimiz`. It will then rank Java documents higher than Perl documents.
Tips

MySQL determines relevance based on several criteria. It ignores any stopwords as well as words of three characters or less. A match against an index that finds a single hit in the database will have a higher relevance than a match against an index with many hits. In other words, rare words have greater value than common words, and overly common words become stopwords that have no value at all. It is therefore important to the utility of MySQL's full text searching that you have a large set of data to search against. Small data sets such as the one above with only four rows will produce odd results—like no hits on the word MySQL!

When you are adding a lot of data at once, such as when you are indexing the web pages on your web site, you should drop the FULLTEXT index, insert your updated web pages, then recreate the index. Inserts on tables with FULLTEXT indexes are quite expensive and work better if you do the indexing all at once.

The MySQL team is working hard on this fairly recent addition to MySQL. In the near future, you can expect the ability to look for phrases instead of just words as well as the ability to define your own words that must always be indexed. In fact, by the time you read this book, some new features have probably already been added to MySQL full text searching.

Transactions

MySQL recently introduced transactions along with SQL for executing statements in a transactional context. By default, MySQL is in a state called autocommit. Autocommit mode means that any SQL you send to MySQL is executed immediately. In some cases, however, you may want to execute two or more SQL statements together as a single unit of work.

A transfer between two bank accounts is the perfect example of such a transaction. The banking system needs to make sure that the debit from the first account and the credit to the second account occur as a single unit of work. If they are treated separately, the server could in theory crash between the debit and the credit. The result would be that you would lose that money!

By making sure the two statements occur as a single unit of work, transactions ensure that the first statement can be "rolled back" in the event that the second statement fails. To use transactions in MySQL, you first need to create a table using a transactional table type such as BDB or InnoDB. * If your MySQL install was not compiled with support for these table types, you cannot use transactions unless you reinstall. The SQL to create a transactional table specifies one of the transactional types:

```
CREATE TABLE ACCOUNT (  
    ACCOUNT_ID BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
```

* Not all platforms support these table types.
BALANCE DOUBLE

TYPE = BDB;

For a transaction against a transactional table to work, you need to turn off autocommit. You can do this with the command:

SET AUTOCOMMIT=0;

Now you are ready to begin using MySQL transactions. Transactions start with the BEGIN command:

BEGIN;

Your mysql client is now in a transactional context with respect to the server. Any change you make to a transactional table will not be made permanent until you commit it. Changes to nontransactional tables, however, will take place immediately. In the case of the account transfer, we issue the following statements:

UPDATE ACCOUNT SET BALANCE = 50.25 WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = 100.25 WHERE ACCOUNT_ID = 2;

Once you're done with any changes, complete the transaction by using the COMMIT command:

COMMIT;

The true advantage of transactions, of course, comes into play should an error occur in executing the second statement. To abort the entire transaction before a commit, issue the ROLLBACK command:

ROLLBACK;

In reality, the logic behind such complex transactional operations, including commits and rollbacks, requires solid design and well-structured error handling. We will cover these programmatic elements of transaction management in Chapter 8.

Of course, it would be useful if MySQL performed the actual math. It can do just that as long as you store the values you want with a SELECT call:

SELECT @FIRST := BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 1;
SELECT @SECOND := BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 2;
UPDATE ACCOUNT SET BALANCE = @FIRST - 25.00 WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = @SECOND + 25.00 WHERE ACCOUNT_ID = 2;

In addition to the COMMIT command, a handful of other commands will automatically end any current transaction as if a COMMIT had been issued. These commands are:

- ALTER TABLE
- BEGIN
- CREATE INDEX
- DROP DATABASE
- DROP TABLE
• LOCK TABLES
• RENAME TABLE
• TRUNCATE
• UNLOCK TABLES

Chapter 8 covers some of the more intricate details of using transactions in database applications.

Table Locking

Table locking is the poor man’s transaction. In short, MySQL lets you lock down a group of tables so that only a single client can use it. Unlike transactions, you are not limited by the type of table. You cannot, however, roll back any actions taken against a locked table.

Locking has two basic functions:

• Enables multiple statements to execute against a group of tables as one unit of work
• Enables multiple updates to occur faster under some circumstances

MySQL supports three kinds of locks: read, read local, and write. Both kinds of read locks lock the table for reading by a client and all other clients. As long as the lock is in place, no one can write to the locked tables. Read and read local locks differ in that read local allows a client to execute nonconflicting INSERT statements as long as no changes to the MySQL files from outside of MySQL occur while the lock is held. If changes might occur by agents outside of MySQL, a read lock is required.

A write lock locks the specified tables against all access—read or write—by any other client. To lock a table, use the following command:

    LOCK TABLES ACCOUNT WRITE;

Now that the ACCOUNT table is locked, you can read from it and modify the data behind it and be certain that no one else will change the data you read between your read and write operations:

    SELECT @BAL:=BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 1;
    UPDATE ACCOUNT SET BALANCE = @BAL * 0.03 WHERE ACCOUNT_ID = 1;

Finally, you need to release the locks:

    UNLOCK TABLES;

It is really important that you unlock the tables! Failing to do so can result in preventing further access to those tables. Finally, any table locking should be short lived. Long-lived locks seriously degrade database performance.
Functions

Functions in SQL are similar to functions in other programming languages such as C and Perl. The function takes zero or more arguments and returns some value. For example, the function `SQRT(16)` returns 4. Within a MySQL SELECT statement, functions may be used in one of two ways:

As a value to be retrieved

This form involves a function in the place of a column in the list of columns to be retrieved. The return value of the function, evaluated for each selected row, is part of the returned result set as if it were a column in the database. For example:

```
SELECT name, FROM_UNIXTIME(date)
FROM events
WHERE time > 90534323
```

This query selects the name of each event and the date of the event formatted in human-readable form for all events more recent than the given time. `FROM_UNIXTIME()` transforms a standard Unix time value into a human-readable form.

```
# The LENGTH() function returns the character length of
# a given string.
SELECT title, text, LENGTH(text)
FROM papers
WHERE author = 'Stacie Sheldon'
```

This query selects the title of a paper, the full text of the paper, and the length of the text in bytes for all of the papers authored by Stacie Sheldon. The `LENGTH()` function returns the character length of a given string.

As part of a WHERE clause

This form involves a function used in place of a constant when evaluating a WHERE clause. The value of the function is used for comparison for each row of the table. For example:

```
SELECT name
FROM entries
WHERE id = ROUND( (RAND() * 34) + 1 )
```

This query randomly selects the name of an entry from a pool of 35 entries. The `RAND()` function generates a random number between 0 and 1. This random value is then multiplied by 34 to turn the value into a number between 0 and 34. Incrementing the value by 1 provides a number between 1 and 35. The `ROUND()` function rounds the result to the nearest integer. The result is a whole number between 1 and 35 and will therefore match one of the ID numbers in the table.

```
SELECT name, FROM_UNIXTIME(date)
FROM events
WHERE time > (UNIX_TIMESTAMP() - (60 * 60 * 24 ))
```

* You can use aliasing, covered earlier in the chapter, to give the resulting columns "friendly" names.
† Remember that SQL is case insensitive. This particular function is simply written `FROM_UNIXTIME()` by convention. You can use `FROM_UNIXTIME()` or `From_UNIXtime()` if they feel more natural to you.
You may use functions in both the value list and the WHERE clause. This query selects the name and date of each event less than a day old. With no arguments, the UNIX_TIMESTAMP( ) function returns the current time in Unix format.

```
SELECT name
FROM people
WHERE password = ENCRYPT(name, LEFT(name, 2))
```

You may also use the value of a table field within a function. This example returns the names of people who used their names as passwords. The ENCRYPT( ) function returns a Unix password-style encryption of the specified string using the supplied two-character salt. The LEFT( ) function returns the left-most \( n \) characters of the specified string.

**Date functions**

The most common functions you use will likely be the MySQL functions that enable you to manipulate dates. You already saw some of these functions earlier for translating a Unix-style date into a human-readable form of the date. MySQL, of course, provides more powerful functions for doing things such as calculating the time between two dates:

```
SELECT TO_DAYS(NOW()) - TO_DAYS('2000-12-31');
```

This example provides the number of days that have passed in this millennium. The NOW( ) function returns the DATETIME representing the moment in time when the command was executed. Less obviously, the TO_DAYS( ) function returns the number of days since the year 1 B.C., represented by the specified DATE or DATETIME. 

Not everyone likes to see dates formatted the way MySQL provides them by default. Fortunately, MySQL lets you format dates to your own liking using the DATE_FORMAT function. It takes a DATE or DATETIME and a format string indicating how you want the date formatted:

```
mysql> SELECT DATE_FORMAT('1969-02-17', '%W, %M %D, %Y');
+--------------------------+
| DATE_FORMAT('1969-02-17', '%W, %M %D, %Y') |
| Monday, February 17th, 1969                     |
+--------------------------+
1 row in set (0.39 sec)
```

Chapter 15 contains a full list of valid tokens for the DATE_FORMAT( ) function.

---

* MySQL is actually incapable of representing this date. Valid date ranges in MySQL are from January 1, 1000, to December 31, 9999. There is also no support in MySQL for alternative calendar systems such as the Hebrew, Chinese, or Muslim calendars.
String functions

In addition to date functions, you are likely to use string functions. We saw one such function above: the `LENGTH()` function. This function provides the number of characters in the specified string. The most common string function you are likely to use, however, is the `TRIM()` function, which removes extra spaces from columns.

One interesting function is the `SOUNDEX()` function. It translates a word into its soundex representation. The soundex representation is a way of representing the sound of a string so that you can compare two strings to see if one is misspelled:

```
mysql> SELECT SOUNDEX('too');
+------------+
<table>
<thead>
<tr>
<th>SOUNDEX('too')</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOO</td>
</tr>
</tbody>
</table>
+------------+
1 row in set (0.42 sec)

mysql> SELECT SOUNDEX('two');
+------------+
<table>
<thead>
<tr>
<th>SOUNDEX('two')</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOO</td>
</tr>
</tbody>
</table>
+------------+
1 row in set (0.00 sec)
```

For these two homonyms, the `SOUNDEX()` function provided the same value. Consequently, an application can leverage this function to check spelling variations.

Outer Joins

MySQL supports a more powerful joining than the simple inner joins we saw earlier. Specifically, MySQL supports something called a left outer join (also known as simply an outer join), which you specify with the keywords `LEFT JOIN`. This type of join is similar to an inner join, except that it includes data in the first column named that does not match any in the second column. If you remember our author and book tables from earlier in the chapter, you will remember that our join would not list any authors who did not have books in our database. You may want to show entries from one table that have no corresponding data in the table to which you are joining. That is where an outer join comes into play:

```
SELECT book.title, author.name
FROM author
LEFT JOIN book ON book.author = author.id
```

This query is similar to the inner join that you already understand:

```
SELECT book.title, author.name
FROM author, book
WHERE book.author = author.id
```
Note that an outer join uses the keyword ON instead of WHERE. The key difference in results is that the new syntax of the outer join will include authors such as Neil Gaiman, for whom no book is in our database. The results of the outer join would therefore look like this:

<table>
<thead>
<tr>
<th>book.title</th>
<th>author.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Green Mile</td>
<td>Stephen King</td>
</tr>
<tr>
<td>Guards, Guards!</td>
<td>Terry Pratchett</td>
</tr>
<tr>
<td>lmzadi</td>
<td>Peter David</td>
</tr>
<tr>
<td>Gold</td>
<td>Isaac Asimov</td>
</tr>
<tr>
<td>Howling Mad</td>
<td>Peter David</td>
</tr>
<tr>
<td>NULL</td>
<td>Neil Gaiman</td>
</tr>
</tbody>
</table>

MySQL takes this concept one step further by using a natural outer join. A natural outer join will combine the rows from two tables that have identical column names with identical types and identical values:

```
SELECT my_prod.name
FROM my_prod
NATURAL LEFT JOIN their_prod
```

This natural join will list all product names with identical entries in the my_prod and their_prod tables.

**Unions**

One of the newest MySQL features as of MySQL 4.0 is the support for SQL unions. A union is simply a tool for combining the results from multiple selects into a single result set listing. A MySQL union looks like this:

```
SELECT first_name, last_name
FROM Author
UNION
SELECT fname, lname
FROM Editor;
```

This query will provide a list of all authors and editors in the database. The list will include in the first column the values of the first_name column for authors and the values of the fname column for editors. The second column will include the last_name values for authors and lname values for editors.

If one person is an author and an editor, he will appear a single time in the list. You can, however, get MySQL to show the person twice in the results by using the ALL keyword:

```
SELECT first_name, last_name
FROM Author
UNION ALL
SELECT fname, lname
FROM Editor;
```
Batch Processing

Batch loading is the act of loading a lot of data into or pulling a lot of data out of MySQL all at once. MySQL supports two types of batch loading.

Command-line loads

In the simplest kind of batch load, you stick all your SQL commands in a file and send the contents of that file to MySQL:

```
mysql -h somehost -u uid -p < filename
```

In other words, you are using the command line to pipe the SQL commands into the `mysql` command-line utility. The examples on this book's web site contain several SQL command files that you can load into MySQL in this manner before you run the examples.

The LOAD command

The `LOAD` command enables you to load data from a file containing only data (no SQL commands). For example, if you had a file containing the names of all the books in your collection with one book on each line and the title and author separated by a tab, you could use the following command to load that data into your book table:

```
LOAD DATA LOCAL INFILE 'books.dat' INTO TABLE BOOK;
```

This command assumes that the file `books.dat` has one line for each database record to be inserted. It further assumes that there is a value for every column in the table or `\N` for null values. So, if the `BOOK` table has three columns, each line of `books.dat` should have three tab-separated values.

The `LOCAL` keyword tells the `mysql` command line to look for the file on the same machine as the client.* Without it, MySQL looks for the file on the server. Of course, if you are trying to load something on the server, you need to have been granted the special `FILE` privilege. Finally, keep in mind that nonlocal loads refer to files relative to the installation directory of MySQL.

If you have a comma-separated value file such as an Excel file, you can change the delimiter of the `LOAD` command:

```
LOAD DATA LOCAL INFILE 'books.dat'
INTO TABLE BOOK
FIELDS TERMINATED BY ',';
```

If a file contains values that would cause duplicate records in the database, you can use the `REPLACE` and `IGNORE` keywords to dictate the correct behavior. `REPLACE` will cause the values from the file to replace the ones in the database, where the `IGNORE` keyword will cause the duplicate values to be ignored. The default behavior is to ignore duplicates.

* Reading from files local to the client is available only on MySQL 3.22.15 and later.
Pulling data from MySQL

Finally, MySQL provides a tool for copying the results of a SELECT from the database into a file:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS TERMINATED BY ','
FROM BOOK;
```

This query copies all rows in the BOOK table into the file books.dat. You could then use this file to load into an Excel spreadsheet or another database. Because this file is created on the server, it is created relative to the base directory for the database in use. On a Mac OS X basic installation, for example, this file is created as /usr/local/var/test/test.dat.

A more complex version of this command enables you to put quotes (or any other characters) around fields:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS ENCLOSED BY '"' TERMINATED BY ',',
FROM BOOK;
```

Of course, you probably want only the string fields (CHAR, VARCHAR, etc.) enclosed in quotes. You can accomplish this by adding the OPTIONALLY keyword:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS OPTIONALLY ENCLOSED BY '"' TERMINATED BY ',',
FROM BOOK;
```

Chapter 15 contains a full range of options for loading and extracting data from MySQL.
PHP 6/MySQL Programming

for the absolute beginner

NO EXPERIENCE REQUIRED

“This series shows that it’s possible to teach newcomers a programming language and good programming practices without being boring.”

—LOU GRINZO,
Reviewer for Dr. Dobb’s Journal

ANDY HARRIS
PHP 6/MySQL® Programming for the Absolute Beginner

Andy Harris

Course Technology PTR
A part of Cengage Learning
In Chapters 9 and 10, you learned how to create a basic database and connect it to a PHP program. PHP and MySQL are wonderful for working with basic databases. However, most real-world problems involve data that is too complex to fit in one table. Database designers have developed some standard techniques for handling complex data that reduce redundancy, improve efficiency, and provide flexibility. In this chapter, you learn how to use the relational model to build complex databases involving multiple entities. Specifically, you learn:

- How the relational model works
- How to build use-case models for predicting data usage
- How to construct entity-relationship diagrams to model your data
- How to build multi-table databases
- How joins are used to connect tables
- How to build a link table to model many-to-many relationships
- How to optimize your table design for later programming
INTRODUCING THE SPY DATABASE

In this chapter, you build a database to manage your international spy ring. (You do have an international spy ring, don’t you?) Saving the world is a complicated task, so you’ll need a database to keep track of all your agents. Secret agents are assigned to various operations around the globe, and certain agents have certain skills. The examples in this chapter will take you through the construction of such a database. You’ll see how to construct the database in MySQL. In Chapter 12, “Building a Three-Tiered Data Application,” you use this database to make a really powerful spymaster application in PHP.

The spy database reflects a few facts about my spy organization (called the Pantheon of Humanitarian Performance, or PHP).

- Each agent has a code name.
- Each agent can have any number of skills.
- More than one agent can have the same skill.
- Each agent is assigned to one operation at a time.
- More than one agent can be assigned to one operation.
- A spy’s location is determined by the operation.
- Each spy has an age (so I know when they should be claiming senior discounts).
- Each operation has only one location.

This list of rules helps explain some characteristics of the data. In database parlance, they are called business rules. I need to design the database so these rules are enforced.

IN THE REAL WORLD

I set up this particular set of rules in a somewhat arbitrary way because they help make my database as simple as possible while still illustrating most of the main problems encountered in data design. Usually you don’t get to make up business rules. Instead, you learn them by talking to those who use the data every day.

THE BADSPY DATABASE

As you learned in Chapter 9, “Using MySQL to Create Databases,” it isn’t difficult to build a data table, especially if you have a tool like phpMyAdmin. Table 11.1 illustrates the schema of my first pass at the spy database.
At first glance, the badSpy database design seems like it ought to work, but problems crop up as soon as you begin adding data to the table. Table 11.2 shows the results of the badSpy data after I started entering information about some of my field agents.

### Table 11.1 Bad Spy Schema

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>agentID</td>
<td>int(11)</td>
</tr>
<tr>
<td>name</td>
<td>varchar(30)</td>
</tr>
<tr>
<td>specialty</td>
<td>varchar(40)</td>
</tr>
<tr>
<td>assignment</td>
<td>varchar(40)</td>
</tr>
<tr>
<td>description</td>
<td>varchar(40)</td>
</tr>
<tr>
<td>location</td>
<td>varchar(20)</td>
</tr>
<tr>
<td>age</td>
<td>int(11)</td>
</tr>
</tbody>
</table>

### Inconsistent Data Problems

Gold Elbow’s record indicates that Operation Dancing Elephant is about infiltrating a suspicious zoo. Falcon’s record indicates that the same operation is about infiltrating a suspicious circus. For the purpose of this example, I’m expecting that an assignment has only one description, so one of these descriptions is wrong. There’s no way to know whether it’s a zoo or a circus by looking at the data in the table, so both records are suspect. Likewise, it’s hard
to tell if Operation Enduring Angst takes place in Lower Volta or Lower Votla, because the two records that describe this mission have different spellings.

The circus/zoo inconsistency and the Volta/Votla problem share a common cause. In both cases the data-entry person (probably a low-ranking civil servant, because international spymasters are far too busy to do their own data entry) had to type the same data into the database multiple times. This kind of inconsistency causes all kinds of problems. Different people choose different abbreviations. You may see multiple spellings of the same term. Some people simply do not enter data if it’s too difficult. When this happens, you cannot rely on the data. (Is it a zoo or a circus?) You also can’t search the data with confidence. (I’ll miss Blackford if I look for all operatives in Lower Volta, because he’s listed as being in Lower Votla.) If you look carefully, you notice that I misspelled “sabotage.” It will be very difficult to find everywhere this word is misspelled and fix them all.

**Problem with the Operation Information**
There’s another problem with this database. If for some reason Agent Rahab were dropped from the database (maybe she was a double agent all along), the information regarding Operation Raging Dandelion would be deleted along with her record, because the only place it is stored is as a part of her record. The operation’s data somehow needs to be stored separately from the agent data.

**Problems with Listed Fields**
The specialty field brings its own troubles to the database. This field can contain more than one entity, because spies should be able to do more than one thing. (My favorite combination is explosives and flower arranging.) Fields with lists in them can be problematic.

- It’s much harder to figure out what size to make a field that may contain several entities. If your most talented spy has 10 different skills, you need enough room to store all 10 skills in every spy’s record.
- Searching on fields that contain lists of data can be difficult.

You might be tempted to insert several different skill fields (maybe a skill1, skill2, and skill3 field, for example), but this doesn’t completely solve the problem. It is better to have a more flexible system that can accommodate any number of skills. The flat file system in this badSpy database is not capable of that kind of versatility.

**Age Issues**
The age field sounds like a good idea, but in real life it’s very difficult to use. People age every year, so how do I keep the ages up to date? I could update each spy’s age on his or her birthday,
but I’d need to have the birthday stored for each spy, and I’d need to run a script every day to check for any spy birthdays and increase the age. The other solution would be to simply age everyone once a year, but that doesn’t seem very satisfying.

**DESIGNING A BETTER DATA STRUCTURE**

The *spy* master database isn’t complicated, but the *badSpy* database shows a number of ways even a simple database can go wrong. This database is being used to save the free world, so it deserves a little more thought. Fortunately, data developers have come up with a number of ways to think about data structure.

It is usually best to back away from the computer and think carefully about how data is used before you write a single line of code.

**Defining Rules for a Good Data Design**

Data developers have come up with a list of rules for creating well-behaved databases:

- Break your data into multiple tables.
- Make no field with a list of entries.
- Do not duplicate data.
- Make each table describe only one entity.
- Don’t store information that should be calculated instead.
- Create a single primary key field for each table.

A database that follows all these rules will avoid most of the problems evident in the *badSpy* database. Fortunately, there are some well-known procedures for improving a database so it can follow all these rules.

**Normalizing Your Data**

Data programmers try to prevent the problems evident in the *badSpy* database through a process called *data normalization*. The basic concept of normalization is to break down a database into a series of tables. If each of these tables is designed correctly, the database is less likely to have the sorts of problems described so far. Entire books have been written about data normalization, but the process breaks down into three major steps, called normal forms.

**First Normal Form: Eliminate Listed Fields**

The normal forms are officially listed in terms that would put a lawyer or mathematician to sleep. One “official” description of the first normal form looks like this:
A table is in first normal form if and only if it represents a relation. It does not allow nulls or duplicate rows.

Yea, that’s catchy. It’s really a lot simpler than it sounds: Eliminate listed fields (like the specialty field in this example).

The goal of the first normal form (sometimes abbreviated 1NF) is to eliminate repetition in the database. The primary culprit in the badSpy database is the specialty field. Having two different tables, one for agents and another for specialties, is one solution.

Data designers seem to play a one-string banjo. The solution to almost every data design problem is to create another table. As you see, there is quite an art form to what should be in that new table.

The two tables would look somewhat like those shown in Tables 11.3 and 11.4.

### Table 11.3 Agent Table in 1NF

<table>
<thead>
<tr>
<th>Agent ID</th>
<th>Name</th>
<th>Assignment</th>
<th>Description</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Rahab</td>
<td>Raging Dandelion</td>
<td>Plant Crabgrass</td>
<td>Sudan</td>
</tr>
<tr>
<td>2</td>
<td>Gold Elbow</td>
<td>Dancing Elephant</td>
<td>Infiltrate suspicious zoo</td>
<td>London</td>
</tr>
<tr>
<td>3</td>
<td>Falcon</td>
<td>Dancing Elephant</td>
<td>Infiltrate suspicious circus</td>
<td>London</td>
</tr>
</tbody>
</table>

### Table 11.4 Specialty Table in 1NF

<table>
<thead>
<tr>
<th>Specialty ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>electronics</td>
</tr>
<tr>
<td>2</td>
<td>counterintelligence</td>
</tr>
<tr>
<td>3</td>
<td>sabotage</td>
</tr>
</tbody>
</table>

Note that I did not include all data in these example tables, but just enough to give you a sense of how these tables would be organized. Also, you learn later in this chapter a good way to reconnect these tables and assert the proper relationship between the agents and their specialties.
Second Normal Form: Eliminate Redundancies

The official form of the second normal form is just as inspiring as the first normal form:

A table is in second normal form (2NF) only if it is in 1NF and all nonkey fields are dependant entirely on the candidate key, not just part of it.

I bet the guy who wrote *that* is a lot of fun at parties...

Once all your tables are in the first normal form, the next step is to deal with all the potential redundancy issues. These mainly occur because data is entered more than one time. To fix this, you need to (you guessed it) build new tables. The agent table could be further improved by moving all data about operations to another table. Figure 11.1 shows a special diagram called an Entity Relationship diagram, which illustrates the relationships between these tables.

![Figure 11.1](image)

An Entity Relationship diagram (ER diagram) reveals the relationships between data elements. In this situation, I thought carefully about the data in the spy database. As I thought about the data, three distinct entities emerged. By separating the operation data from the agent data, I have removed redundancy: The user enters operational data only one time. This eliminates several of the problems in the original database. It also fixes the situation where an operation’s data was lost because a spy turned out to be a double agent. (I’m still bitter about that defection.)
I used a free program called DBDesigner 4 to build the ER diagrams for this chapter. A copy of this program is available on the CD-ROM that accompanies this book. Often, though, I just use a white board or paper.

The boxes in this diagram represent the *entities* (agents, operations, and specialties) and the lines between them represent the *relationships* between the entities. The reverse arrow (crow’s feet) symbols on the relationship lines describe the types of relationships between the various entities. Read on about the third normal form, and then I explain how the various relationship types work.

**Third Normal Form: Ensure Functional Dependency**
The third normal form concentrates on the elements associated with each entity.

The official description has the wit and charm you’ve come to expect:

*A table is in 3NF if it is in 2NF and has no transitive dependencies on the candidate key.*

For a table to be in the third normal form, that table must have a single primary key and every field in the table must relate only to that key. For example, the description field is a description of the operation, not the agent, so it belongs in the operation table.

In the third phase of normalization, you look through each piece of table data and ensure that it directly relates to the table in which it’s placed. If not, either move it to a more appropriate table or build a new table for it.

This diagram illustrates the three entities in the **spy** database (at least up to now) and the relationships between them. Each entity is enclosed in a rectangle, and the lines between each represent the relationships between the entities. Take a careful look at the relationship lines. They have crow’s feet on them to indicate some special relationship characteristics. There are essentially three kinds of relationships (at least in this overview of data modeling).

**Defining Relationship Types**
The easiest way to normalize your databases is with a stylized view of them such as the ER diagram. ER diagrams are commonly used as a data-design tool. Take another look at the ER diagram for the **spy** database in Figure 11.2.
Recognizing One-to-One Relationships

One-to-one relationships happen when each instance of entity A has exactly one instance of entity B. A one-to-one entity is described as a simple line between two entities with no special symbols on either end.

One-to-one relationships are rare, because if the two entities are that closely related, usually they can be combined into one table without any penalty. The spy ER diagram in Figure 11.2 has no one-to-one relationships.

Describing Many-to-One Relationships

One-to-many (and many-to-one) relationships happen when one entity can contain more than one instance of the other. For example, each operation can have many spies, but in this example, each agent can only be assigned to one mission at a time. Thus, the agent-to-operation relationship is considered a many-to-one relationship, because a spy can have only one operation, but one operation can relate to many agents. In this version of ER notation, I’m using crow’s feet to indicate the many sides of the relationship.

There are actually several different kinds of one-to-many relationships, each with a different use and symbol. For this overview, I treat them all the same and use the generic crow’s feet symbol. When you start writing more-involved databases, investigate data diagramming more closely by looking into books on data normalization and software engineering. Likewise, data normalization is a far more involved topic than the brief discussion in this introductory book.
Recognizing Many-to-Many Relationships

The final type of relationship shown in the spy ER diagram is a many-to-many relationship. This type of relationship occurs when each entity can have many instances of the other. Agents and skills have this type of relationship, because one agent can have any number of skills, and each skill can be used by any number of agents. A many-to-many relationship is usually shown by crow’s feet on each end of the connecting line.

It’s important to generate an ER diagram of your data including the relationship types, because different strategies for each type of relationship creation exist. These strategies emerge as I build the SQL for the improved spy database.

Building Your Data Tables

After designing the data according to the rules of normalization, you are ready to build sample data tables in SQL. It pays to build your tables carefully to avoid problems. I prefer to build all my tables in an SQL script so I can easily rebuild my database if (okay, when) my programs mess up the data structure. Besides, enemy agents are always lurking about preparing to sabotage my operations.

I also add plenty of sample data in the script. You don’t want to work with actual data early on, because you are guaranteed to mess up somewhere during the process. However, it is a good idea to work with sample data that is a copied subset of the actual data. Your sample data should anticipate some of the anomalies that might occur in actual data. (For example, what if a person doesn’t have a middle name?)

My entire script for the spy database is available on the book’s CD as buildSpy.sql. All SQL code fragments shown in the rest of this chapter come from that file and use the MySQL syntax.

Setting Up the System

I began my SQL script with some comments that describe the database and a few design decisions I made when building the database:

```
#########################################
# buildSpy.sql
# builds and populates all databases for spy examples
# uses mysql - should adapt easily to other rdbms
# by Andy Harris for PHP/MySQL for Abs. Beg
#########################################

#########################################
# conventions
# conventions
```
Notice that I specified a series of conventions. These self-imposed rules help make my database easier to manage. Some of the rules might not make sense yet (because I haven’t identified what a foreign key is, for instance), but the important thing is that I have clearly identified some rules that help later on.

The code then specifies the database and deletes all tables if they already existed. This behavior ensures that I start with a fresh version of the data. This is also ideal for testing, since you can begin each test with a database in a known state.

**Creating the agent Table**

The normalized agent table is quite simple. The actual table is shown in Table 11.5.

The only data remaining in the agent table is the agent’s name and a numerical field for the operation. The operationID field is used as the glue that holds together the agent and operation tables.

I’ve added a few things to improve the SQL code that creates the agent table.
These improvements enhance the behavior of the `agent` table, and simplify the table tremendously.

```
CREATE TABLE agent (  
    agentID int(11) NOT NULL AUTO_INCREMENT,  
    name varchar(50) default NULL,  
    operationID int(11) default NULL,  
    birthday date,  
    PRIMARY KEY  (agentID),  
    FOREIGN KEY (operationID) REFERENCES operation (operationID)  
);  
```

Recall that the first field in a table is usually called the primary key. Primary keys must be unique and each record must have one.

- I named each primary key according to a special convention. Primary key names always begin with the table name and end with `ID`. I added this convention because it makes things easier when I write programs to work with this data.
- The `NOT NULL` modifier requires you to put a value in the field. In practice, this ensures that all records of this table must have a primary key.
- The `AUTO_INCREMENT` identifier is a special tool that allows MySQL to pick a new value for this field if no value is specified. This will ensure that all entries are unique. In fact, when `AUTO_INCREMENT` is set, you cannot manually add a value to the field.
- I added an indicator at the end of the `CREATE TABLE` statement to indicate that `agentID` is the primary key of the `agent` table.

### Table 11.5

<table>
<thead>
<tr>
<th>agentID</th>
<th>name</th>
<th>operationID</th>
<th>birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bond</td>
<td>1</td>
<td>1961-08-30</td>
</tr>
<tr>
<td>2</td>
<td>Falcon</td>
<td>1</td>
<td>1975-05-23</td>
</tr>
<tr>
<td>3</td>
<td>Cardinal</td>
<td>2</td>
<td>1979-01-27</td>
</tr>
<tr>
<td>4</td>
<td>Blackford</td>
<td>2</td>
<td>1956-10-16</td>
</tr>
<tr>
<td>5</td>
<td>Rahab</td>
<td>3</td>
<td>1981-09-14</td>
</tr>
</tbody>
</table>
Not all databases use the `AUTO_INCREMENT` feature the same way as MySQL, but most offer an alternative. You might need to look up some other way to automatically generate key fields if you aren’t using MySQL. Check the Help system for whatever DBMS you’re using to learn any specific quirks.

- The `FOREIGN KEY` reference indicates that the `operationID` field acts as a reference to the operation table. Some databases use this information to reinforce relationships. Even if the database does not use this information, it can be useful documentation for the purpose of the field.

**Inserting a Value into the agent Table**
The `INSERT` statements for the `agent` table have one new trick made possible by the primary key’s `AUTO_INCREMENT` designation.

```sql
INSERT INTO agent VALUES(
    null, 'Bond', 1, '1961-08-30'
);
```

The primary key is initialized with the value `null`. This might be surprising because primary keys are explicitly designed to never contain a `null` value. Since the `agentID` field is set to `AUTO_INCREMENT`, the `null` value is automatically replaced with an unused integer. This trick ensures that each primary key value is unique.

**CONVERTING BIRTHDAY TO AGE**
One obvious change in the agent data is the inclusion of the `birthday` rather than the agent’s age. This reflects another important idea in data design: don’t store data that should be calculated. The age of an agent is dynamic, which leads to problems. However, you can store the agent’s birthdate (which doesn’t change) and calculate the agent’s age (and other potentially useful details) from this basic information.

**Introducing SQL Functions**
SQL has a number of functions built in, which allow you to manipulate the data in various ways. Table 11.6 illustrates a few commonly used SQL functions.

Many of these functions are used to work with dates and times, which (as you see in a moment) can be extremely useful.

The `birthday` value is stored in the `agent` table, but you need to determine the age of the agent, perhaps in years and months.
Finding the Current Date

Begin by using the `NOW()` function to retrieve the current date and time.

```sql
SELECT NOW()
NOW()
```

Determining Age with `DATEDIFF()`

These values on their own aren’t that useful, but you can compare the date returned by `NOW()` to the agent’s birthday to determine how old the agent is.

```sql
SELECT
  name,
  NOW(),
  birthday,
  DATEDIFF(NOW(),birthday)
FROM agent;
```

The `DATEDIFF()` function takes two date values and returns the difference between them as a number of days. (See Table 11.7.)

### Table 11.6 Common SQL Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT(A, B)</td>
<td>Concatenates two string values to create a single string output.</td>
</tr>
<tr>
<td>FORMAT(X, D)</td>
<td>Formats the number X to D significant digits</td>
</tr>
<tr>
<td>CURRDATE(), CURRTIME()</td>
<td>Returns the current date or time</td>
</tr>
<tr>
<td>NOW()</td>
<td>Returns the current date and time as one value</td>
</tr>
<tr>
<td>MONTH(), DAY(), YEAR(), WEEK(), WEEKDAY()</td>
<td>Extracts the given data from a date value</td>
</tr>
<tr>
<td>HOUR(), MINUTE(), SECOND()</td>
<td>Extracts the given data from a time value</td>
</tr>
<tr>
<td>DATEDIFF(A, B)</td>
<td>Determines the difference between two dates—commonly used to calculate ages</td>
</tr>
<tr>
<td>SUBTIMES(A, B)</td>
<td>Determines the difference between two times</td>
</tr>
<tr>
<td>FROMDAYS(INT)</td>
<td>Converts an integer number of days into a date value</td>
</tr>
</tbody>
</table>
Performing Math on Function Results
Of course, this is only mildly interesting. You can do some math on the results to get the age in years as shown in Table 11.8:

```sql
SELECT
    name,
    NOW(),
    birthday,
    DATEDIFF(NOW(),birthday) / 365 AS age
FROM agent;
```

### Table 11.8 Calculating Age in Years

<table>
<thead>
<tr>
<th>name</th>
<th>NOW()</th>
<th>birthday</th>
<th>age</th>
</tr>
</thead>
</table>

Be aware that sometimes leap years can confuse the `DATEDIFF` function, which may cause calculations to be off by a few days.
Converting Number of Days to a Date

Most of the standard math operations work in SQL, but there’s a better way. You can convert the number of days back to a date with the `FROM_DAYS()` function as in Table 11.9.

```sql
SELECT
    name,
    NOW(),
    birthday,
    DATEDIFF(NOW(), birthday) as daysOld,
    FROM_DAYS(DATEDIFF(NOW(), birthday))
FROM agent;
```

<table>
<thead>
<tr>
<th>name</th>
<th>NOW()</th>
<th>birthday</th>
<th>daysOld</th>
<th>FROM_DAYS(DATEDIFF(NOW(), birthday))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bond</td>
<td>2008-07-11 22:02:01</td>
<td>1961-08-30</td>
<td>17117</td>
<td>0046-11-12</td>
</tr>
<tr>
<td>Falcon</td>
<td>2008-07-11 22:02:01</td>
<td>1975-05-23</td>
<td>12103</td>
<td>0033-02-19</td>
</tr>
<tr>
<td>Blackford</td>
<td>2008-07-11 22:02:01</td>
<td>1956-10-16</td>
<td>18896</td>
<td>0051-09-26</td>
</tr>
<tr>
<td>Rahab</td>
<td>2008-07-11 22:02:01</td>
<td>1981-09-14</td>
<td>9797</td>
<td>0026-10-28</td>
</tr>
</tbody>
</table>

Extracting Years and Months from the Date

The `FROM_DAYS()` calculation will return the age as if it were a date in the ancient world, but now you can extract the year and days with appropriate functions as Table 11.10 illustrates:

```sql
SELECT
    name,
    NOW(),
    birthday,
    FROM_DAYS(DATEDIFF(NOW(), birthday)) as age,
    YEAR(FROM_DAYS(DATEDIFF(NOW(), birthday))) as years,
    MONTH(FROM_DAYS(DATEDIFF(NOW(), birthday))) as months
FROM agent;
```
Concatenating to Build the age Field

Finally, you can concatenate these values back to one field (See Table 11.11):

```
SELECT
    name,
    birthday,
    CONCAT(
        YEAR(FROM_DAYS(DATEDIFF(NOW(), birthday))), ' years, ',
        MONTH(FROM_DAYS(DATEDIFF(NOW(), birthday))), ' months') as age
FROM agent;
```

<table>
<thead>
<tr>
<th>name</th>
<th>birthday</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bond</td>
<td>1961-08-30</td>
<td>46 years, 11 months</td>
</tr>
<tr>
<td>Falcon</td>
<td>1975-05-23</td>
<td>33 years, 2 months</td>
</tr>
<tr>
<td>Cardinal</td>
<td>1979-01-27</td>
<td>29 years, 6 months</td>
</tr>
<tr>
<td>Blackford</td>
<td>1956-10-16</td>
<td>51 years, 9 months</td>
</tr>
<tr>
<td>Rahab</td>
<td>1981-09-14</td>
<td>26 years, 10 months</td>
</tr>
</tbody>
</table>

BUILDING A VIEW

While there’s nothing terribly difficult about all this function gymnastics, it’s way too much work to do all this every time you want to convert a birthday to a date. Well, that’s true. MySQL 5.0 and later includes a wonderful tool called the View, which allows you to take complex information like all these date calculations and store it in the database itself. Take a look at the following code:

```
TABLE 11.10  WORKING WITH YEAR() AND MONTH() FUNCTIONS

<table>
<thead>
<tr>
<th>name</th>
<th>NOW()</th>
<th>birthday</th>
<th>age</th>
<th>years</th>
<th>months</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackford</td>
<td>2008-07-11 22:03:09</td>
<td>1956-10-16</td>
<td>0051-09-26</td>
<td>51</td>
<td>9</td>
</tr>
</tbody>
</table>

TABLE 11.11  CREATING THE AGE FROM YEAR AND MONTH

<table>
<thead>
<tr>
<th>name</th>
<th>birthday</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bond</td>
<td>1961-08-30</td>
<td>46 years, 11 months</td>
</tr>
<tr>
<td>Falcon</td>
<td>1975-05-23</td>
<td>33 years, 2 months</td>
</tr>
<tr>
<td>Cardinal</td>
<td>1979-01-27</td>
<td>29 years, 6 months</td>
</tr>
<tr>
<td>Blackford</td>
<td>1956-10-16</td>
<td>51 years, 9 months</td>
</tr>
<tr>
<td>Rahab</td>
<td>1981-09-14</td>
<td>26 years, 10 months</td>
</tr>
</tbody>
</table>

Chapter 11 • Data Normalization

407
DROP VIEW IF EXISTS agentAgeView;
CREATE VIEW agentAgeView AS
SELECT
    name,
birthday,
operationID,
CONCAT(  
    YEAR(FROM_DAYS(DATEDIFF(NOW(), birthday))), ' years, ', 
    MONTH(FROM_DAYS(DATEDIFF(NOW(), birthday))), ' months' ) as age 
FROM agent;

If you look closely, it’s almost the same query used to generate the age from the birthday, but I added a new CREATE VIEW statement (and I included the operationID value, which might be useful later on). When you run this code, nothing overt happens, but the database creates a new structure called agentView. The cool part happens when you run the following query:

SELECT * FROM agentView;

This extremely simple query yields a marvelous result, shown in Table 11.12.

<table>
<thead>
<tr>
<th>name</th>
<th>birthday</th>
<th>operationID</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bond</td>
<td>1961-08-30</td>
<td>1</td>
<td>46 years, 11 months</td>
</tr>
<tr>
<td>Falcon</td>
<td>1975-05-23</td>
<td>1</td>
<td>33 years, 2 months</td>
</tr>
<tr>
<td>Cardinal</td>
<td>1979-01-27</td>
<td>2</td>
<td>29 years, 6 months</td>
</tr>
<tr>
<td>Blackford</td>
<td>1956-10-16</td>
<td>2</td>
<td>51 years, 9 months</td>
</tr>
<tr>
<td>Rahab</td>
<td>1981-09-14</td>
<td>3</td>
<td>26 years, 10 months</td>
</tr>
</tbody>
</table>

All the details of the age manipulation are buried. Now the agentView view can be treated just like a table (at least for SELECT queries) and it automatically creates an age from the birthday field. You can also do all the SELECT tricks on the view, and it still operates as expected:

SELECT
    name,
age
FROM agentView
WHERE
    age < 30;
A view isn’t exactly like a table. You can’t \texttt{UPDATE} or \texttt{INSERT} view data in a view. Views are meant to simplify \texttt{SELECT} queries. Also, the view data isn’t really stored in the database as such. The data is all stored in the tables, and the view is just a formatted way of looking at the data that’s actually stored in the tables. (See Table 11.13.)

**Creating a Reference to the operation Table**

Take a careful look at the \texttt{operationID} field of the \texttt{agent} table. This field contains an integer, which refers to a particular operation. I also added an indicator specifying \texttt{operationID} as a foreign key reference to the \texttt{operation} table. The \texttt{operationID} field in the \texttt{agent} table contains a reference to the primary key of the \texttt{operation} table. This type of field is referred to as a foreign key.

Some DBMS systems require you to specify primary and foreign keys. MySQL currently does not require this, but it’s a good idea to do so anyway for two reasons. First, it’s likely that future versions of MySQL will require these statements, because they improve a database’s reliability. Second, it’s good to specify in the code when you want a field to have a special purpose, even if the DBMS doesn’t do anything with that information.

**Building the operation Table**

The new \texttt{operation} table (Table 11.14) contains information referring to an operation.
Each operation gets its own record in the operation table. All the data corresponding to an operation is stored in the operation record. Each operation’s data is stored only one time.

This has a number of positive effects:

- It’s necessary to enter operation data only once per operation, saving time on data entry.
- Since there’s no repeated data, you won’t have data inconsistency problems (like the circus/zoo problem).
- The new database requires less space, because there’s no repeated data.
- The operation is not necessarily tied to an agent, so you won’t accidentally delete all references to an operation by deleting the only agent assigned to that mission. (Remember, this could happen with the original data design.)
- If you need to update operation data, you don’t need to go through every agent to figure out who was assigned to that operation. (Again, you would have had to do this with the old database design.)

The SQL used to create the operation table is much like that used for the agent table:

```
CREATE TABLE operation (  
  operationID int(11) NOT NULL AUTO_INCREMENT,  
  name varchar(50) default NULL,  
  description varchar(50) default NULL,  
  location varchar(50) default NULL,  
  PRIMARY KEY (`OperationID`)  
);
```

```
INSERT INTO operation VALUES(  
  null, 'Dancing Elephant',  
  'Infiltrate suspicious zoo', 'London'  
);
```

As you can see, the operation table conforms to the rules of normalization, and it also is much like the agent table. Notice that I’m being very careful about how I name things. SQL is (theoretically) case-insensitive, but I’ve found that this is not always true. (I have found this especially in MySQL, where the Windows versions appear unconcerned about case, but Unix versions treat operationID and OperationID as different field names.) I specified that all field
names will use camel case (just like you’ve been doing with your PHP variables). I also named the key field according to my own formula (table name followed by ID).

Using a Join to Connect Tables
The only downside to disconnecting the data tables is the necessity to rejoin the data when needed. The user doesn’t care that the operation and the agent are in different tables, but he will want the data to look as if they were on the same table. The secret to reattaching tables is a tool called the inner join. Take a look at the following SELECT statement in SQL:

```
SELECT
    agent.name AS 'agent',
    operation.name AS 'operation',
FROM
    agent, operation
WHERE
    agent.operationID = operation.operationID
ORDER BY agent.name;
```

At first glance, this looks like an ordinary query, but it is a little different. It joins data from two different tables. Table 11.15 illustrates the results of this query.

<table>
<thead>
<tr>
<th>agent</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackford</td>
<td>Enduring Angst</td>
</tr>
<tr>
<td>Bond</td>
<td>Dancing Elephant</td>
</tr>
<tr>
<td>Cardinal</td>
<td>Enduring Angst</td>
</tr>
<tr>
<td>Falcon</td>
<td>Dancing Elephant</td>
</tr>
<tr>
<td>Rahab</td>
<td>Furious Dandelion</td>
</tr>
</tbody>
</table>

Creating Useful Joins
An SQL query can pull data from more than one table. To do this, follow a few basic rules.

- Specify the field names more formally if necessary. Notice that the SELECT statement specifies `agent.name` rather than simply `name`. This is necessary because both tables contain a field called `name`. Using the `table.field` syntax is much like using a person’s first and last name. It’s not necessary if there’s no chance of confusion, but in a larger environment, the more complete naming scheme can avoid confusion.
• Use the AS clause to clarify your output. This provides an alias for the column and provides a nicer output. The ‘as’ component will show up as the column heading on the output table.

• Modify the FROM clause so it indicates both of the tables you’re pulling data from. The FROM clause up to now has only specified one table. In this example, it’s necessary to specify that data will be coming from two different tables.

• Indicate how the tables will be connected using a modification of the WHERE clause.

• The order of the table names in the WHERE clause does not matter, but getting the case incorrect can cause problems in some versions of MySQL.

Examining a Join without a WHERE Clause
The WHERE clause helps clarify the relationship between two tables. As an explanation, consider the following query:

```sql
SELECT
    agent.name AS 'agent',
    operation.name AS 'operation',
FROM
    agent, operation
ORDER BY agent.name;
```

This query is much like the earlier query, except it includes the operationID field from each table and it omits the WHERE clause. You might be surprised by the results, which are shown in Table 11.16.

The results of this query are called a Cartesian join, which shows all possible combinations of agent and operation. Of course, you don’t really want all the combinations—only those combinations where the two tables indicate the same operation ID.

Adding a WHERE Clause to Make a Proper Join
Without a WHERE clause, all possible combinations are returned. The only concern-worthy records are those where the operationID fields in the agent and operation tables have the same value. The WHERE clause returns only these values joined by a common operation ID.

The secret to making this work is the operationID fields in the two tables. You’ve already learned that each table should have a primary key. The primary key field is used to uniquely identify each database record. In the agents table, agentID is the primary key. In operations, operationID is the primary key. (You might note my unimaginative but very useful naming convention here.)
I was able to take all data that refers to the operation out of the agent table by replacing those fields with a field that points to the operations table’s primary key. A field that references the primary key of another table is called a foreign key. Primary and foreign keys cement the relationships between tables.

Adding a Condition to a Joined Query

Of course, you can still use the WHERE clause to limit which records are shown. Use the AND structure to build compound conditions. For example, this code returns the code name and operation name of every agent whose code name begins with B:

```sql
SELECT
    agent.name AS 'agent',
    operation.name AS 'operation',
FROM
    agent, operation
WHERE
    agent.operationID = operation.operationID
AND agent.name LIKE 'B%';
```
**The Truth About Inner Joins**

You should know that the syntax I provided here is a convenient shortcut supported by most DBMS systems. The inner join’s formal syntax looks like this:

```
SELECT agent.name, operation.name
FROM
    agent INNER JOIN operation
ON agent.OperationID = operation.OperationID ORDER BY agent.name;
```

Many data programmers prefer to think of the join as part of the `WHERE` clause and use the `WHERE` syntax. A few SQL databases (notably many offerings from Microsoft) do not allow the `WHERE` syntax for inner joins and require the `INNER JOIN` to be specified as part of the `FROM` clause. When you use this `INNER JOIN` syntax, the `ON` clause indicates how the tables will be joined.

### Creating a View to Store a Join

Very often, you’ll use a query to link up two (or more) tables that have been broken up by the normalization process. The `VIEW` statement that simplifies SQL functions can also be used to encode joins and make them easier to work with:

```
# build agent operation view
CREATE VIEW agentOpView AS
SELECT
    agent.name AS 'agent',
    operation.name AS 'operation',
    operation.description AS 'task',
    operation.location AS 'location'
FROM
    agent, operation
WHERE
    agent.operationID = operation.operationID;
```

This code is just an SQL `SELECT` statement linking together the `agent` and `operation` tables. I embedded the query in a `CREATE VIEW` structure, naming the view `agentOpView`. Notice that all
the data fields (but none of the keys) are available in the views, and I gave names to each field that hide the original table relationship. When you run this code, your database will show a ‘view’, which looks a lot like a table. You can run a query on it, as shown in Table 11.17.

```
SELECT * FROM agentOpView
```

<table>
<thead>
<tr>
<th>Table 11.17 Running the agentOpView</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent</td>
</tr>
<tr>
<td>Bond</td>
</tr>
<tr>
<td>Falcon</td>
</tr>
<tr>
<td>Cardinal</td>
</tr>
<tr>
<td>Blackford</td>
</tr>
<tr>
<td>Rahab</td>
</tr>
</tbody>
</table>

With agentOpView in place, I can run queries against agentOpView as if it were a real table. The view doesn’t really hold any data at all. It’s just a placeholder for the query that joins up the two tables. But it’s as easy to use as a real table, so you can do SELECT queries on the view as if it were a real table:

```
SELECT agent, location FROM agentOpView WHERE operation LIKE 'E%';
```

Views hide the join to make the data easier to use. Of course, you can’t do INSERT or UPDATE queries on a view, because it doesn’t really hold any data. Still, views make normalized data a lot easier to use than it used to be.

It’s interesting that the outcome of this view is looking very much like the original badSpy database (at least to the end user) but the data underneath is much safer and better organized than it was in the original data structure.

**Building a Link Table for Many-to-Many Relationships**

Once you’ve created an ER diagram, you can create new tables to handle all the one-to-many relationships. It’s a little less obvious what to do with many-to-many relationships, such as the link between agents and skills. Recall that each agent can have many skills, and several
agents can use each skill. The best way to handle this kind of situation is to build a special kind of table.

**Enhancing the ER Diagram**

Figure 11.3 shows a new version of the ER diagram that eliminates all many-to-many relationships.

The ER diagram in Figure 11.3 improves on the earlier version shown in Figure 11.2 in a number of ways.

- I added (PK) to the end of every primary key.
- I added (FK) to the end of every foreign key.
- The placements of the lines in the diagram are now much more important. I now draw a line only between a foreign key reference and the corresponding primary key in the other table. Every relationship should go between a foreign key reference in one table and a primary key in the other.
- The other main improvement is the addition of the agent_specialty table. This table is interesting because it contains nothing but primary and foreign keys. Each entry in this table represents one link between the agent and specialty tables. All the actual data
referring to the agent or specialty are encoded in other tables. This arrangement provides a great deal of flexibility.

Most tables in a relational database are about entities in the data set, but link tables are about relationships between entities.

Creating the specialty Table

The specialty table is simple, as shown in Table 11.18.

<table>
<thead>
<tr>
<th>specialtyID</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Electronics</td>
</tr>
<tr>
<td>2</td>
<td>Counterintelligence</td>
</tr>
<tr>
<td>3</td>
<td>Sabatoge</td>
</tr>
<tr>
<td>4</td>
<td>Doily Design</td>
</tr>
<tr>
<td>5</td>
<td>Explosives</td>
</tr>
<tr>
<td>6</td>
<td>Flower Arranging</td>
</tr>
</tbody>
</table>

As you can see, there is nothing in the specialty table that connects it directly with any particular agent. Likewise, you find no references to specialties in the agent table. The complex relationship between these two tables is handled by the new agent_specialty table.

CREATE TABLE agent_specialty (
    agent_specialtyID int(11) NOT NULL AUTO_INCREMENT,
    agentID int(11) default NULL,
    specialtyID int(11) default NULL,
    PRIMARY KEY (agent_specialtyID),
    FOREIGN KEY (agentID) REFERENCES agent (agentID),
    FOREIGN KEY (specialtyID) REFERENCES specialty (specialtyID)
);

This is called a link table because it manages relationships between other tables. Table 11.19 shows a sample set of data in the agent_specialty table.
Interpreting the agent_specialty Table with a Query

Of course, the agent_specialty table is not directly useful to the user, because it contains nothing but foreign key references. You can translate the data to something more meaningful with an SQL statement:

```
SELECT
    agent.name as 'Agent',
    specialty.name as 'Specialty'
FROM
    agent, specialty, agent_specialty
WHERE agent.agentID = agent_specialty.agentID
    AND specialty.specialtyID = agent_specialty.specialtyID;
```

It requires two comparisons to join the three tables. It is necessary to forge the relationship between agent and agent_specialty by common agentID values. It’s also necessary to secure the bond between specialty and agent_specialty by comparing the specialtyID fields. The results of such a query show that the correct relationships have indeed been joined, as you can see in Table 11.20.

The link table provides the linkage between tables that have many-to-many relationships. Each time you want a new relationship between an agent and a specialty, you add a new record to the agent_specialty table.

**Table 11.19 The agent specialty Table**

<table>
<thead>
<tr>
<th>agent specialty ID</th>
<th>agent ID</th>
<th>specialty ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Building a View for the Link Table
Many-to-many joins are complex enough that they deserve their own view as well.

```
# build agentSpecialty view
DROP VIEW IF EXISTS agentSpecialtyView;
CREATE VIEW agentSpecialtyView as
SELECT
    agent.name,
    specialty.name
FROM
    agent, specialty, agent_specialty
WHERE
    agent.agentID = agent_specialty.agentID
    AND specialty.specialtyID = agent_specialty.specialtyID;
```

You can then use the view to simplify queries. For example, you can determine which agents know flower arrangement with this query:

```
SELECT
    *
FROM
    agentSpecialtyView
WHERE
    specialty LIKE 'FLOWER%';
```
(You know, flower arrangement can be a deadly art in the hands of a skilled practitioner.)

**SUMMARY**

In this chapter you moved beyond programming to an understanding of data, the real fuel of modern applications. You learned how to take a poorly designed table and convert it into a series of well-organized tables that can avoid many data problems. You learned about three stages of normalization and how to build an Entity Relationship diagram. You can now recognize three kinds of relationships between entities and build normalized tables in SQL, including pointers for primary and foreign keys. You can connect normalized tables with `INNER JOIN` SQL statements. You know how to simulate a many-to-many relationship by building a link table. You learned how to build views to simplify working with functions and joins. The civilized world is safer for your efforts.

**CHALLENGES**

1. Locate ER diagrams for data you commonly work with. Examine these documents and see if you can make sense of them.
2. Examine a database you use regularly. Determine if it follows the requirements stated in this chapter for a well-designed data structure. If not, explain what might be wrong with the data structure and how it might be corrected.
3. Design an improved data structure for the database you examined in question 2. Create the required tables in SQL and populate them with sample data.
4. Design a database to describe data for a programming problem. (Be warned, most data problems are a lot more complex than they first appear.) Create a data diagram, then build the tables and populate them with sample data.