Databases for Digital Humanists

Harvey Quamen
Jon Bath
Welcome to DHSI 2019!

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 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.

To access the DHSI wifi network, simply go into your wireless settings and connect to the “DHSI” network and enter the password “dhsi2019”.

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....
DHSI Wi-Fi

Network name: DHSI
Passkey: dhsi2019
The 2019 schedule is just taking shape nicely! A very few things to confirm, add, etc, still but this is the place to be to find out what is happening when / where ...

### Sunday, 2 June 2019 [DHSI Registration + Suggested Outings]

If you're here a day or two before we begin, or staying a day or two afterwards, here are a few ideas of things you might consider doing ....

Suggested Outing 1, 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.

Suggested Outing 2, 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.

Suggested Outing 3, Salt Spring 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 Salt Spring 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 ....

Suggested Outing 4, 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 if 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.

And more!

Self-organised High Tea at the Empress Hotel, scooter rentals, visit to the Royal BC Museum, darts at Christie's 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!

### 9:00 to 4:00

**Early Class Meeting: 4. [Foundations] DH For Department Chairs and Deans (David Strong Building C124, Classroom)

Further details are available from instructors in mid May to those registered in the class. Registration materials will be available in the classroom.**

### 3:00 to 5:00

**DHSI Registration (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 OR even into the city for a nice meal.

### Monday, 3 June 2019

Your hosts for the week are Alyssa Arbuckle, Ray Siemens, and Jannaya Friggstad Jensen.

### 7:45 to 8:15

**Last-minute Registration (MacLaurin Building, Room A100)**
### 8:30 to 10:00

- **Welcome, Orientation, and Instructor Overview** *(MacLaurin A144)*
  - Welcome to the Territory
  - Welcome to DHSI: Ray Siemens, Alyssa Arbuckle
  - Welcome from UVic: Jonathan Bengtson (University Librarian), Alexandra D’Arcy (Associate Dean Research, Humanities)

#### Classes in Session (click for details and locations)

2. [Foundations] Introduction to Computation for Literary Criticism *(Clearihue A102, Lab)*
4. [Foundations] DH For Department Chairs and Deans *(David Strong Building C124, Classroom)*
5. [Foundations] Developing a Digital Project (With Omeka) *(Clearihue A031, Lab)*
9. Out-of-the-Box Text Analysis for the Digital Humanities *(Human and Social Development A160, Lab)*
10. Sound and Digital Humanities *(Cornett A120, Classroom)*
11. Critical Pedagogy and Digital Praxis in the Humanities *(Clearihue D132, Classroom)*
12. Digital Humanities for Japanese Culture: Resources and Methods *(McPherson Library A003, Classroom)*
13. Conceptualising and Creating a Digital Edition *(McPherson Library 210, Classroom)*
14. Retro Machines & Media *(McPherson Library 129, Classroom)*
15. Geographical Information Systems in the Digital Humanities *(Clearihue A105, Lab)*
16. Spatio-Temporal Data Analysis *(McPherson Library A003, Classroom)*
17. Introduction to IIIF: Sharing, Consuming, and Annotating the World’s Images *(Cornett A121, Classroom)*
18. Web APIs with Python *(Human and Social Development A170, Lab)*
19. Ethical Data Visualization: Taming Treacherous Data *(Cornett A128, Classroom)*
20. Linked Open Data and the Semantic Web *(Cornett A132, Classroom)*
22. The Frontend: Modern JavaScript & CSS Development *(Clearihue A030, Lab)*
24. Information Security for Digital Researchers *(David Strong Building C114, Classroom)*

### 10:15 to Noon

#### Lunch break / Unconference Coordination Session *(MacLaurin A144)*

Grab a sandwich and come on down!
Discussion topics, scheduling, and room assignments from among all DHSI rooms will be handled at this meeting.

### 1:30 to 4:00

- **Institute Lecture: Jacqueline Wernimont** *(Dartmouth C):* "Sex and Numbers: Pleasure, Reproduction, and Digital Biopower"
  - Chair: Anne Cong-Huyen *(U Michigan)* *(MacLaurin A144)*

Abstract: Drawing from Numbered Lives *(MIT 2018)*, this talk will consider a long history of sex-number entanglement in Anglo-American Cultures. Drawing on historical and contemporary objects and practices, Wernimont will ask "in what ways do theories of biopower, critical gender and critical race studies, and media studies" suggest that we can understand this set of entanglements and their impacts. NB: While relevant, this talk will not include discussions of sexual trauma or violence. It will include frank discussion of sex acts and various ways of translating sexual behavior into numbers.

### 5:00 to 6:00

**Opening Reception** *(University Club)*

### Tuesday, 4 June 2019

**9:00 to Noon**
- Classes in Session

**12:15 to 1:15**
- *Mystery* Lunches

**1:30 to 4:00**
- Classes in Session

**4:10 to 5:00**

DHSI Conference and Colloquium Lightning Talk Session 1 *(MacLaurin A144)*
Wednesday, 5 June 2019

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference

"Mystery" Lunches

Presentation: An Introduction to Scholarly Publishing with Manifold (MacLaurin A144)
Lunch included for those who [register here]

This presentation introduces Manifold Scholarship, a Mellon-funded digital publishing platform developed by the CUNY Graduate Center, The University of Minnesota Press, and Cast Iron Coding. Manifold allows you to create beautiful, dynamic open access projects that can include text, images, video, embedded resources, and social annotation. We will provide an overview of Manifold and demonstrate how faculty, students and staff in the digital humanities can use Manifold to publish open access scholarly works, conduct and participate in peer review, and create custom edited versions of public domain course texts and OER.

1:30 to 4:00
Classes in Session

4:15 to 5:15
DHSI Conference and Colloquium Lightning Talk Session 2 (MacLaurin A144)
Chair: Kim O'Donnell (Simon Fraser U)

- Catherine Ryu (Michigan State U), "Tone Perfect: Developing a Multimodal Audio Database for Mandarin Chinese as an Open Source"
- Kenzie Burchell (U Toronto Scarborough), "Making Responsible Reporting Practices Visible: Comparing newswire coverage of humanitarian crises in Syria"
- Jessica Linzel (Brock U), "The Shopkeeper Aristocracy: Mapping Trade Networks in Colonial Niagara"
- Kirsten Painter (U Washington), "From Bogatyr's to Bread: Digitization & Online Exhibition of Rare Russian Children's Books at the U Washington"
- John Barber (Washington State U), "A Mighty Span"

6:00 to 7:00
"Half Way There!" [An Informal, Self-Organized Birds of a Feather Get-Together] (Felicitas, Student Union Building)
Bring your DHSI nametag and enjoy your first tipple on us! [A great opportunity for an interest group meet-up ...]

Thursday, 6 June 2019

9:00 to Noon
Classes in Session

12:15 to 1:15
"Mystery" Lunches

[instructor lunch meeting]

1:30 to 4:00
Classes in Session

4:15 to 5:15
DHSI Conference and Colloquium Lightning Talk Session 3 (MacLaurin A144)
Chair: Kim O'Donnell (Simon Fraser U)

- Colleen Kolba (U South Florida), "What Comics can Teach our Students about Multimodal Literacy"
- Trish Baer (ETCL; U Victoria), "Preserving Digital Legacies: Archived Websites and Digital Discoverability"
- Suchismita Dutta (U Miami), "The Importance of Archival Transcription for Genre Building"
- Jeffrey Lawler (California State U, Long Beach), "Twining our way through the Past: Video Game Authoring as History Pedagogy"
Friday, 7 June 2019 [DHSI; ADHO Pedagogy SIG Conference Opening]

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch Reception / Course E-Exhibits (MacLaurin A100)

1:30 to 1:50
Remarks, A Week in Review (MacLaurin A144)

2:00 to 3:00
Joint Institute Lecture (DHSI and ADHO Pedagogy SIG Conference):
Matt Gold (CUNY Graduate Center and Association for Computers and the Humanities): “Thinking Through DH: Proposals for Digital Humanities Pedagogy”
Chair: Diane Jakacki (Bucknell U)

Abstract: How do we teach digital humanities, and how should DH be taught? What, indeed, should we teach when we teach DH? This talk will present a proposal for grounding digital humanities pedagogical practice in the research interests of our students and the epistemological foundations of our methods rather than through an approach grounded more central in data and methods.

3:30 to 5:00
Joint Reception: DHSI and ADHO Pedagogy SIG Conference (University Club)
E-Poetry Event (Chris Tanasescu)
Watch this space for details, including how to participate!

Saturday, 8 June 2019 [Conference, Colloquium, and Workshop Sessions]

8:00 to 9:00
Conference / Workshop Registration (MacLaurin A100)
The day's events are included with your DHSI registration. If you're not registered in DHSI, you're very welcome to join us by registering here as a Conference / Colloquium / Workshop participant. We'll have a nametag waiting for you!

Coffee, Tea, &c?
Looking for some morning coffee or tea, or a small nibble? Options and hours of operation for weekend campus catering are available here. Mystic Market usually opens around 10.00.

9:00 to 4:00
DHSI Conference and Colloquium Sessions
ADHO Pedagogy SIG Conference Sessions
Right2Left Workshop Sessions

9:00 to 4:00
All Day DHSI Workshop Session (click for workshop details and free registration for DHSI participants)
- 55. Introduction to Machine Learning in the Digital Humanities [8-9 June; All day, each day] (David Strong Building C124, Classroom)

9:00 to 9:10
Informal Greetings, Room Set-up (Lobby, outside Hickman 105)

Session 1
DHSI Colloquium and Conference (Hickman 105)
Digital Humanities & Literature, Chair: Kim O'Donnell (Simon Fraser U)
- Youngmin Kim (Dongguk U), “Transdiscursivity in the Convergence of Digital Humanities and World Literature”
- Caroline Winter (U Victoria), “Digitizing Adam Smith’s Literary Library”
- Kaitlyn Fralick (U Victoria); Kailey Fukushima (U Victoria); Sarah Karlson (U Victoria), “Victorian Poetry
<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:10 to 10:30</td>
<td><strong>ADHO Pedagogy SIG Conference</strong> (Hickman 110)</td>
</tr>
<tr>
<td></td>
<td>Chair: Katherine Faull (Bucknell U)</td>
</tr>
<tr>
<td></td>
<td>- Aaron Tucker and Nada Savicevic (Ryerson U), <strong>Write Here, Right Now: An Open Source eTextbook for the Flipped Classroom</strong></td>
</tr>
<tr>
<td></td>
<td>- Heather McAlpine (U Fraser Valley), <strong>Digital Meters: Using Text Encoding to Teach Literature in the Undergraduate Classroom</strong></td>
</tr>
<tr>
<td></td>
<td>- Tiina H. Airaksinen (U Helsinki), <strong>Digital Humanities in Cultural Studies: Creating a MOOC course for University Students and A-Level Students</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Right2Left Workshop</strong> (Hickman 116)</td>
</tr>
<tr>
<td></td>
<td>Keynote - Nathan P. Gibson (Ludwig Maximilians U, München): <strong>Thinking in -TR: Reorienting the Directional Assumptions of Global Digital Scholarship</strong></td>
</tr>
<tr>
<td>10:30 to 10:40</td>
<td><strong>Break</strong></td>
</tr>
<tr>
<td>10:40 to Noon</td>
<td><strong>Session 2</strong></td>
</tr>
<tr>
<td></td>
<td><strong>DHSI Colloquium and Conference</strong> (Hickman 105)</td>
</tr>
<tr>
<td></td>
<td>Digital Humanities &amp; Society, Chair: Eleanor Reed (Hastings C)</td>
</tr>
<tr>
<td></td>
<td>- Joel Zapata (Southern Methodist U), <strong>Uncovering the Southern Plains' Mexican American Civil Rights Movement</strong></td>
</tr>
<tr>
<td></td>
<td>- Ayo Oseinwo (U Ibadan), <strong>Online Newspaper Construction of Agitation for the Sovereign State of Biafra in Nigeria</strong></td>
</tr>
<tr>
<td></td>
<td>- Joseph Jones (U British Columbia), <strong>Testbed for an Approach to Distant Reading: Fictions That Represent Vietnam War Resisters in Canada</strong></td>
</tr>
<tr>
<td></td>
<td>- Brendan Mackie (U California, Berkeley), <strong>Visualizing Long-Term Cultural Change: An Example From The Birth of Civil Society</strong></td>
</tr>
<tr>
<td>Noon to 1:10</td>
<td><strong>ADHO Pedagogy SIG Conference</strong> (Hickman 110)</td>
</tr>
<tr>
<td></td>
<td>Chair: Laura Estill (St Francis Xavier U)</td>
</tr>
<tr>
<td></td>
<td>- Jane Jackson (Chinese U of Hong Kong), <strong>Interrogating digital spaces for intercultural meaning-making</strong></td>
</tr>
<tr>
<td></td>
<td>- Ryan Ikeda (UC Berkeley), <strong>Disrupting Digital Literacy: Situating Electronic Literature Among Public Education Initiatives</strong></td>
</tr>
<tr>
<td></td>
<td>- Christopher Church, Katherine Hepworth (U Nevada, Reno), <strong>We’re STEAMed! A call for balancing technical instruction and disciplinary content in the digital humanities</strong></td>
</tr>
<tr>
<td></td>
<td>- Chelsea Milbourne (Cal Poly, San Luis Obispo), <strong>Finding the Right Fit between Technology and Class Content: Reflections on Including Web Development in a Digital Storytelling Course</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Right2Left Workshop</strong> (Hickman 116)</td>
</tr>
<tr>
<td></td>
<td>- Edward “Eddie” Surman (Claremont Graduate U), <strong>Qualitative Digital Text Analysis and #Right2Left Languages: A Demonstration of Atlas.ti using the Hebrew Bible</strong></td>
</tr>
<tr>
<td>1:10 to 2:30</td>
<td><strong>Session 3</strong></td>
</tr>
<tr>
<td></td>
<td><strong>DHSI Colloquium and Conference</strong> (Hickman 105)</td>
</tr>
<tr>
<td></td>
<td>Digital Humanities &amp; Community, Chair: Claire Carlin (U Victoria)</td>
</tr>
<tr>
<td></td>
<td>- Pia Russel (U Victoria); Emily Stremel (U Victoria), <strong>Mentorship and disability: Supporting disabled employees in digital humanities</strong></td>
</tr>
<tr>
<td></td>
<td>- Amy Lueck (Santa Clara U), <strong>Virtually Emplacing Indigenous Memory</strong></td>
</tr>
<tr>
<td></td>
<td>- Md. Shehabul Alam (National U Bangladesh), <strong>Integrating Library Service with Union Information and Service Center: A Joint Initiative towards Digital Bangladesh</strong></td>
</tr>
<tr>
<td></td>
<td>- Veronica Gomez (Instituto de Humanidades y Ciencias Sociales (HuCoSo) - UNL-CONICET), <strong>Latin American E-literature and Location: The Nation Revisited in Electronic Literature Organization (ELO)</strong></td>
</tr>
<tr>
<td>2:30 to</td>
<td><strong>ADHO Pedagogy SIG Conference</strong> (Hickman 110)</td>
</tr>
<tr>
<td>3:30</td>
<td>Chair: Chris Tănăsescu (UC Louvain)</td>
</tr>
<tr>
<td></td>
<td>- Laura Estill (St Francis Xavier U), <strong>One Assignment, Three Ways: Assessing DH Projects in a Literature Course</strong></td>
</tr>
<tr>
<td></td>
<td>- Felix Bayode Oke, Stella N. Kpolugbo (Anchor U Lagos), <strong>The Multimodal Technique as a Pedagogical Tool in Pelu Awofeso’s White Lagos: A Definitive and Visual Guide to the Eyo Festival</strong></td>
</tr>
<tr>
<td></td>
<td>- Shu Wan (U Iowa), <strong>A digital “historical gaze” of Chinese students in Iowa, 1911-1930</strong></td>
</tr>
<tr>
<td></td>
<td>- Francesca Giannetti (Rutgers U, New Brunswick), <strong>“So near while apart: Correspondence Editions as Critical Library Pedagogy and Digital Humanities Methodology</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Right2Left Workshop</strong> (Hickman 116)</td>
</tr>
<tr>
<td></td>
<td>- Najla Jarkas (American U Beirut) and David Joseph Wrisley (NYU Abu Dhabi), <strong>RTL Software Localization and Digital Humanities: the Case Study of Translating Voyant Tools into Arabic</strong></td>
</tr>
</tbody>
</table>
Session 4

DHSI Colloquium and Conference (Hickman 105)
Digital Humanities & Media, Chair: Caroline Winter (U Victoria)
- Ashleigh Cassermete-Stanfield (U Chicago), “Sonifying Hamlet and Reading the Room”

ADHO Pedagogy SIG Conference (Hickman 110)
Chair: Aaron Tucker (Ryerson U)
Youngmin Kim (Dongguk U), “Teaching Digital Humanities and World Literature in Class”
Alice Fleerackers, Juan Pablo Alperin, Esteban Morales, Remi Kalir (Simon Fraser U, U Colorado Denver), “Online annotations in the classroom: How, why, and what do students learn from annotating course material?”
Andie Silva (York C and Graduate Center, CUNY), “Keeping it Local: Undergraduate DH as Feminist Practice”

Right2Left Workshop (Hickman 116)
- Joanna Byszuk (Institute of Polish Language, Polish Academy of Sciences, Warsaw/Computational Stylistics Group) and Alexey Khismatulin (Institute of Oriental Manuscripts, Russian Academy of Sciences, Saint Petersburg), “Attribution of Authorship for Medieval Persian Quasidas with Stylometry”
- Ilan Benatar (New York U), "#Right2Left Biblical Translations in Jewish Textual History: Case Studies in Judeo-Arabic and Judeo-Spanish"

Sunday, 9 June 2019 [Workshop Sessions]

DHSI Registration (MacLaurin Building, Room A100)

8:00 to 5:00
The day’s events are included with your DHSI registration. If you're not registered in DHSI, you're very welcome to join us by registering here as a Conference / Colloquium / Workshop participant. We'll have a nametag waiting for you!

Coffee, Tea, &c?
Looking for some morning coffee or tea, or a small nibble? Options and hours of operation for weekend campus catering are available here. Mystic Market usually opens around 10.00.

All Day Workshop Sessions (click for workshop details and free registration for DHSI participants)
- 55. Introduction to Machine Learning in the Digital Humanities [8-9 June; All day, each day] (David Strong Building C124, Classroom)
- 56. Pedagogy of the Digitally Oppressed: Anti-Colonial DH Methods and Praxis [9 June; All Day] (Hickman 115, Classroom)
- 57. Natural Language Processing and Network Coding Apps for Text & Textual Corpus Analysis in the Humanities [9 June; All Day] (David Strong Building C114, Classroom)

AM Workshop Sessions (click for workshop details and free registration for DHSI participants)
- 59. 3D Visualization for the Humanities [9 June; AM] (Cornett A229, Classroom)
- 60. It’s All Relational: AbTeC’s Indigenous Video Game Workshops as Storytelling Praxis [9 June; AM] (Cornett A121, Classroom)
- 61. Spatial DH: De-Colonizing Cultural Territories Online [9 June; AM] (Clearihue D130, Classroom)
- 63. Creating a CV for Digital Humanities Makers [9 June; AM] (David Strong Building C108, Classroom)

9:00 to Noon
- 64. 3D Visualization for the Humanities [9 June; AM] (Cornett A229, Classroom)
- 65. DHSI Knits: History of Textiles and Technology [9 June; PM] (Fine Arts 109, Classroom)
- 68. Linked Open Datafication for Humanities Scholars [9 June; PM] (McPherson Library A003, Classroom)
- 69. Stylo - WYSIWYM Text Editor for Humanities Scholars [9 June; PM] (McPherson Library A025, Classroom)

Noon to 1:00
Lunch (We recommend Mystic Market on weekends!)

PM Workshop Sessions (click for workshop details and free registration for DHSI participants)
- 65. Indigenous Futurities in the Classroom and Beyond [9 June; PM] (Cornett A121, Classroom)
- 66. DHSI Knits: History of Textiles and Technology [9 June; PM] (Fine Arts 109, Classroom)
- 67. Book History Pedagogy Using Scalar [9 June; PM] (Cornett A121, Classroom)
- 68. Linked Open Datafication for Humanities Scholars [9 June; PM] (McPherson Library A003, Classroom)
- 69. Stylo - WYSIWYM Text Editor for Humanities Scholars [9 June; PM] (McPherson Library A025, Classroom)

After the day, 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 OR even into the city for a bite to eat.

Monday, 10 June 2019
Your hosts for the week are Ray Siemens and Jannaya Friggstad Jensen.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:45 to 8:15</td>
<td>DHSI Last-minute Registration (<a href="#">MacLaurin A100</a>)</td>
</tr>
<tr>
<td>8:30 to 10:00</td>
<td>Welcome, Orientation, and Instructor Overview (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td>10:15 to Noon</td>
<td>Classes in Session (click for details and locations)</td>
</tr>
<tr>
<td></td>
<td>- 30. [Foundations] Databases for Digital Humanists (<a href="#">McPherson Library 210, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 33. Digital Storytelling (<a href="#">Cornett A120, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 34. Text Mapping as Modelling (<a href="#">Clearihue D131, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 35. Stylometry with R: Computer-Assisted Analysis of Literary Texts (<a href="#">Clearihue A102, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 36. Open Access and Open Social Scholarship (<a href="#">Clearihue D130, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 37. Digital Games as Tools for Scholarly Research, Communication and Pedagogy (<a href="#">Cornett A229, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 38. Queer Digital Humanities (<a href="#">David Strong Building C114, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 40. Introduction to Electronic Literature in DH: Research and Practice (<a href="#">Cornett A128, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 41. Surveillance and the Critical Digital Humanities (<a href="#">David Strong Building C108, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 42. Text Analysis with Python and the Natural Language Toolkit (<a href="#">Clearihue A103, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 43. Creating LAMP Infrastructure for Digital Humanities Projects (<a href="#">Human and Social Development A170, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 44. Processing Humanities Multimedia (<a href="#">Human and Social Development A150, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 46. Digital Humanities Pedagogy: Integration in the Curriculum (<a href="#">Cornett A121, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 47. Accessibility &amp; Digital Environments (<a href="#">Priestly Law Library 265, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 48. Agile Project Management (<a href="#">Cornett A132, Classroom/Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 49. XPath for Processing XML and Managing Projects (<a href="#">Clearihue A105, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 50. Endings: How to End (and Archive) your Digital Project (<a href="#">Priestly Law Library 192, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 51. Text Processing - Techniques &amp; Traditions (<a href="#">McPherson Library A025, Classroom</a>)</td>
</tr>
<tr>
<td></td>
<td>- 52. Introduction to Humanities Data Analysis &amp; Visualization in R (<a href="#">HDA, Human and Social Development A160, Lab</a>)</td>
</tr>
<tr>
<td></td>
<td>- 53. Introduction to Network Analysis in the Digital Humanities (<a href="#">Clearihue D132, Classroom</a>)</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference Coordination Session (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td></td>
<td>(Grab a sandwich and come on down!)</td>
</tr>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
<tr>
<td></td>
<td>Institute Lecture: Angel David Nieves (San Diego State U): &quot;3D Mapping and Forensic Traces of Testimony: Documenting Apartheid-Era Crimes Through the Digital Humanities&quot; Chair: Constante Crompton (U Ottawa) (<a href="#">MacLaurin A144</a>)</td>
</tr>
<tr>
<td>4:10 to 5:00</td>
<td>Abstract: In 1989 the killing of a queer, 14-year-old youth in Winnie Mandela's house named Stompie Seipei (an event that few in South Africa are willing to recall, let alone discuss, in any detail) -- is perhaps one of the most glaring examples where the queer and activist community was suppressed or erased from anti-apartheid/liberation histories. Digital humanities may actually help both reconstruct and recover a history that is still very early in the telling, despite what is commonly believed about the liberation struggle and the contributions of queer activists in the dismantling of apartheid. Perhaps it could explain why a youth such as Seipei was killed -- or at the very least, provide a more complex and messy narrative that permits one to know more how the history of queer anti-apartheid activists was suppressed. This talk outlines a methodology for &quot;messy thinking and writing&quot; in the digital humanities that -- through a queer and feminist intersectional framework -- permits a more complex layering of oral histories and 3D historical reconstructions.</td>
</tr>
<tr>
<td>5:00 to 6:00</td>
<td>Reception (<a href="#">University Club</a>)</td>
</tr>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>Lunch break / Unconference</td>
</tr>
<tr>
<td></td>
<td>&quot;Mystery&quot; Lunches</td>
</tr>
</tbody>
</table>

**Tuesday, 11 June 2019**
1:30 to 4:00
Classes in Session

4:15 to 5:15

DHSI Conference and Colloquium Lightning Talk Session 4 (MacLaurin A144)
Chair: Lindsey Seatter (U Victoria)

- Ashley Caranto Morford (U Toronto); Kush Patel (U Michigan); Arun Jacob (McMaster U), “‘Our DHs anti-colonial: Questions and challenges in dismantling colonial influences in digital humanities pedagogy”
- Luis Meneses (ETCL: U Victoria), "Identifying Changes in the Political Environment in Ecuador"
- Laura Horak (Carleton U), “Building the Transgender Media Portal”

6:00 to 8:00
DHSI Newcomer's Gathering (Grad House Restaurant, Graduate Student Centre)
Come down, buy meal and a beverage, and make some new friends!

Wednesday, 12 June 2019

9:00 to Noon
Classes in Session

Lunch break / Unconference

“Mystery” Lunches

12:15 to 1:15

Presentation: An Introduction Jupyter Notebooks for Researchers (MacLaurin A144)

This presentation introduces Jupyter Notebooks for researchers, via a partnership between Compute Canada and the Pacific Institute for the Mathematical Sciences (PIMS) including a large number of Canadian institutions. Read more here. Presenting is James Colliander, PIMS Director and team.

1:30 to 4:00
Classes in Session

4:15 to 5:15

DHSI Conference and Colloquium Lightning Talk Session 5 (MacLaurin A144)
Chair: Lindsey Seatter (U Victoria)

- Calin Murgu (New College of Florida), "Putting local metadata to strategic use: A Dashboard for visualizing 60 years of theses metadata”
- Jason Lajoie (U Waterloo), “Queer Critical Making and the Logic of Control”
- John Barber (Washington State U), "Zambezi River Bridge”

6:00 to 7:00

"Half Way There (yet again)!” [An Informal, Self-Organized Birds of a Feather Get-Together] (Felicitas, Student Union Building)

Bring your DHSI nametag and enjoy your first tipple on us! [A great opportunity for an interest group meet-up ....]

Thursday, 13 June 2019

9:00 to Noon
Classes in Session

Lunch break / Unconference

“Mystery” Lunches

12:15 to 1:15

[Instructor lunch meeting]

1:30 to 4:00
Classes in Session

Institute Lecture: Karina van Dalen-Oskam (Huygens Institute and U Amsterdam; Alliance of Digital Humanities Organizations): “The Riddle of Literary Quality: Some Answers”
Chair: Aaron Mauro (Penn State, Behrend C)
(MacLaurin A144)

4:10 to 5:00

Abstract: What is literature, and can you measure it? That is the key question of the project The Riddle of Literary Quality. “The Riddle” is a research project of the Huygens Institute for the History of the Netherlands (Amsterdam) in collaboration with the Fryeke Akademy (Leeuwarden) and the Institute for Logic, Language and Computation (University of Amsterdam). The Riddle combines computational analysis of writing style with the results of a large online survey of readers, completed by almost 14,000 participants. In my talk, I will go into
some of the main results of the project.

---

**Friday, 14 June 2019**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</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 2:00</td>
<td>Closing, DHSI in Review (MacLaurin A144)</td>
</tr>
</tbody>
</table>

**Contact info:**  
enstitut@uvic.ca  
P: 250-472-5401  
F: 250-472-5681
Databases for Digital Humanists

DHSI 2019
10–14 June 2019

Classroom: McPherson Library 210
Instructors: Harvey Quamen, Jon Bath
Website: [http://hucodev.artsrn.ualberta.ca/hquamen/DHSI/](http://hucodev.artsrn.ualberta.ca/hquamen/DHSI/)

Monday 10 June

8:30 – 10:00
- Go to the Welcome session in MacLaurin A144

10:15 – 12:00
- Instructor & Student Introductions
- Survey of course materials on the website
- Slideshow 1: Why Databases?
  (feel free to look at the 1a supplemental slideshow on your own)
- Begin Slideshow 2: Intro to Database Design

1:30 – 4:00
- Finish Slideshow 2: Intro to Database Design
- Slideshow 3: Relationships
- Worktime: Break-Out Groups to do design exercises
  -- Look at MySQL Tutorial Book 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 11 June

For the keeners: Read through the “Concepts & Strategies” section of the DHSI Tutorial PDF (in the “DHSI Text” folder), especially the two sections “Data Normalization Principles” and “Relationships.” There are other PDFs in the “Textbook Chapters” as well. They’re also on the same topics, but slightly different perspectives.

9:00 – 12:00
- Finish Design exercises and presentations (if necessary)
- Slideshow 4: Birdclub Design
• Installation of MySQL
  • Follow the instructions in the “Installing MySQL” PDF document for your platform.
  • 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
  -- * (pronounced star, not asterisk) means everything
  -- ORDER BY
  -- LIKE
  — the % wildcard character
• 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 12 June

For the Keeners: If you have time, move on to the other database examples. We’ll probably only have time in class to get to the movielens dataset (and then only on Thursday), but feel free to work through these exercises on your own.

9:00 – 12:00
• Slideshow 5: “You Say ‘Sequel,’ I Say ‘Ess-Que-El’: An Introduction to Structured Query Language”
• Codealongs:
  -- MySQL Functions: CONCAT(), COUNT(), DATE_FORMAT(), etc.
  -- Multiple-Table queries <- important, important, important!

1:30 – 4:00
• Slideshow 6: Data Types & CREATE TABLE
  -- DESCRIBE table
  -- ALTER TABLE
  ~ add active column to person
  ~ add primary key to sighting
• Codealong: build a space alien abduction database
• 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 13 June

For the keeners: Read through the “Left Join” and “Views” sections of the PDF tutorial. Continue designing and building your database.

9:00 – 12:00
- The LOAD DATA command: importing data from Excel and other sources
- Slideshow 7: Spatial Metaphors in SQL
- Finding missing information: the LEFT JOIN command
- Load the MovieLens database and do some preliminary queries

1:30 – 4:00
- Advanced SQL:
  — GROUP BY
  — HAVING
- Adding an INDEX to speed up slow queries
- Views (they look like tables, they act like tables, but they are not tables!)
- Worktime: build your database

Friday 14 June

9:00 – 12:00
- Worktime
- Brief intro to phpMyAdmin
- Final Wrap-up
# Digital Humanities Database Web Docs

## Introduction
The following exercises will guide you through installing MySQL on your computer, importing database files, and some basic MySQL syntax. You will need to interact with MySQL through a command line interface, so if you are not familiar with that you first may want to do a web search for “command line basics” plus your operating system name in order to learn more.

## Installing MySQL
MySQL is freely available for download and there is excellent support documentation.

Downloads are available at [http://dev.mysql.com/downloads/mysql/](http://dev.mysql.com/downloads/mysql/). If you are on Windows it is easiest to download “MySQL on Windows (Installer & Tools).” On OSX you will want to download the “MySQL Community Server” DMG archive.

Installation instructions are available for the current version (5.7) at [http://dev.mysql.com/doc/refman/5.7/en/installing.html](http://dev.mysql.com/doc/refman/5.7/en/installing.html). For Windows installation you will just need to run the installer program and it should take care of everything for you; be careful when entering passwords as if you type the root account password incorrectly, or later forget it, it can get very tricky.

If you are installing MySQL on OSX at the end of the installation process it will assign you a temporary password for the `root` account; be sure to copy it to a safe place. You will also need to perform an extra step to make it easy to launch MySQL from any folder. Open Terminal and enter the following if you are on OSX 10.9 or later (we recommend using cut-and-paste so that you get the quotation marks correct):

```bash
echo 'export PATH="/usr/local/mysql/bin:$PATH"' >> ~/.profile
```

If you are on OSX 10.8:

```bash
echo 'export PATH="/usr/local/mysql/bin:$PATH"' >> ~/.bash_profile
```

Once you have entered the line, press return. Nothing visible will happen, but you will have added the location of MySQL to your .profile file (and created the file if it did not exist before).

Restart your computer.

You can test your MySQL installation using the steps outlined at [http://dev.mysql.com/doc/refman/5.7/en/postinstallation.html](http://dev.mysql.com/doc/refman/5.7/en/postinstallation.html). You may also wish to follow the steps there to make your MySQL user accounts more secure, but they are not necessary if you are just using MySQL locally. Generally you would create a non-root account to use for normal operations, but for the purposes of this tutorial we’ll just be using root.
However if you just want to try things out as fast as possible, open up your command line application and enter “mysql -u root -p” and hit return. It should prompt you for the password for the ‘root’ account that you entered as part of the Windows installation process, or that was assigned during the OS X installation. You should get a MySQL welcome message.

The biggest problem people seem to have at this point is just getting MySQL running. If you get a “ERROR 2002 (HY000): Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)” it means that the MySQL server is not running. On OS X you can use the MySQL panel in System Preferences to start the server and/or to instruct it to start automatically after a restart. On Windows it should start automatically when you restart your computer, but if it doesn’t (or if you don’t want it to start automatically) you can use the “MySQL Notifier” application to control startup.

Once you are in enter “SHOW DATABASES;” and press return. All MySQL commands end with a semi-colon. If your command line prompt looks like “->” it means you have forgotten the semi-colon; cancel the current query by entering “\c” and try again. You should get something like this:

[insert show databases.png]

However, on OS X you might get “ERROR 1820 (HY000): You must reset your password using ALTER USER statement before executing this statement.” If this is the case, use the following to change the temporary password:

```
SET PASSWORD = PASSWORD('your_new_password');
```

Press return and you should get a confirmation message. Try entering “SHOW DATABASES;” again.

If at this point things still aren’t working for you, copy and paste the error message along with your operating system into a web search. MySQL has a wonderful user community and someone else has probably had your problem and posted a solution on the internet. For instance, if you forgot to copy down the temporary root password during install on OS X, or that password just doesn’t seem to work, you might need this advice (I know I did!):

http://mazharahmed.me/reset-the-mysql-root-password-in-osx/

**Querying the Jane Austen Movie Adaptation Database**

If you’d like to try out the queries in the chapter that have to do with the Jane Austen database you’ll first need to copy that database to your MySQL installation.

Download this file: austen.sql

Now open your command line application. At this point you should not be in MySQL; if your command line prompt is “mysql>” you’ll need to quit MySQL by entering “quit” or “\q”. Navigate to the folder where you downloaded the “austen.sql” file -- if you don’t know how to do this, now
would be a good time to do a web search for “command line basics” plus your operating system name in order to learn how.

Once you are in the correct folder you can start MySQL by entering “mysql -u root -p” and hitting return. You will be asked for your password and then MySQL will start (if it doesn’t you might want to work through the test outlined in the previous section).

To create a local instance of the Austen database enter “\ austen.sql”. This is one of the only MySQL commands that does not require the command to be terminated by a semi-colon. The system will return a log of the processes it goes through as the database is created.

The first step is to tell MySQL what database to use by entering “USE AUSTEN;”. Then you can see the tables by entering “SHOW TABLES;”. To see the structure of an individual table enter “DESCRIBE table_name;”; where “table_name” is the name of the table you want to know about. For example, “DESCRIBE novel;” returns:

[insert describe novel.png]

You can now work through the example queries described in the chapter. If things don’t seem to be working you’ll want to double check your typing; the most common problem is people forgetting the semi-colon. As mentioned previously, if your command line prompt looks like “->” it means you have not terminated the query with a semi-colon and the system is waiting for further instruction. It is often easier to enter complex queries over several lines, so MySQL will not actually run a query until it reads a semi-colon. However if you have done this by accident you can cancel the current query by entering “\c” and try again.

Once you have managed the queries outlined in the chapter you can give this one a try. We can use SQL to summarize data. For example, let us count the number of Austen adaptations by decade. We can tell MySQL to perform some math on the year column in order to determine an adaptation’s decade and then use the result as a “bin” or a “bucket” into which MySQL should put all the adaptation rows that have matching decades.

[insert count example.png]

FLOOR is a function that rounds numbers down, and COUNT(*) counts the number of items in each GROUP. Summaries like this can be useful because they raise new questions that help us to understand more about our data. Clearly, the number of Austen adaptations is rising. Is that due to our increasingly media-saturated culture? Or is Austen’s popularity growing? The database cannot answer this question, but it can help you discover questions like this that can form the basis for further research.

Querying the Factory Workers Database
Download this file: factory.sql

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](http://www.galbithink.org/names/manfacdd.htm).

Using the same process you used for the “austen.sql” file above, create a local instance of “factory.sql” on your database server. Tell MySQL what database to use by entering “USE FACTORY;”. Then you can see the tables by entering “SHOW TABLES;”.

You can then use DESCRIBE to see the structure for each table:

Try using some basic SELECT queries to explore the data. For example, to generate a list of all workers sorted from oldest to youngest you could use:

```
SELECT * FROM worker ORDER BY age DESC;
```

Think of some other questions that just require one table to answer, and give them a try. If you don’t know the correct MySQL syntax, just do a web search using “MySQL syntax …” and the answer is probably out there.

Once you’ve got a handle on that, try doing a multiple table query. To use the previous example, what if we want each worker listed by just name and age, but also want the city in which they worked:

```
SELECT last_name, given_name, age, location FROM worker, factory WHERE factory_num=factory.id ORDER BY age DESC;
```

The results table contains (SELECT) four columns: last_name, given_name, age, and location. In order to gather this information we need to use (FROM) both the worker and factory tables. The columns that link the two tables (WHERE) are factory_num, which is a foreign key in the worker table, and factory.id, which is the primary key in the factory table. Because both tables have an “id” column we need to tell MySQL which one we mean by prefixing it with the table name (factory.id). Finally we want the results sorted (ORDER BY) from the highest age to the lowest.

Here are a couple examples of more advanced queries that you can try out. Let’s say you wanted to know the average starting age of a worker at each factory:
The results table contains (SELECT) three columns: owner, location, and a calculated column called starting_age which is the average of the value of the current age of all workers in a given factory subtract their years of experience. In order to gather this information we need to use (FROM) both the worker and factory tables. The columns that link the two tables (WHERE) are factory_num, which is a foreign key in the worker table, and factory.id, which is the primary key in the factory table. Because both tables have an “id” column we need to know which one we mean by prefixing it with the table name. We want the average to be calculated by grouping (GROUP BY) all workers according to their factory. Finally we want the results sorted (ORDER BY) from the lowest starting age to the highest. Among others things that this tells us is that we have incomplete data (NULL) in our tables so we might want to look at the data from the first factory a little closer before drawing any conclusions.

Here’s one last example, and admittedly it is not the type of query you’ll likely be writing during your initial foray into MySQL. What if we wanted to see if we could determine family units working in a single factory? Although the database does not explicitly store this data, we could start by finding all people that work in a given factory that share the same last name. And if we also had their ages we could make some hypotheses about potential family relations:

This is an especially tricky query because we need to query the worker table twice in order to find potential matches. And because we are doing this for thousands of workers it may take 10-15 seconds to run on your computer. We are returning (SELECT) five columns of information: factory_num, last_name, name1, age1, name2 and age2. Because our result set actually has two rows for each pair (because there are two people), we eliminate duplicates by using the DISTINCT syntax. If two people share the last name we compare their given names. The name that comes first in the alphabet is assigned to name1; the syntax of line 2 reads if the given name of person 1 comes before the given name of person 2 according to alphabetical order, then assign the given name of person 1 to the column name1, but if the opposite is true then assign the given name of person 2 to the column name1. The next three lines of IF statements perform similar checks. In order to answer this query MySQL needs to use (FROM) the worker table, and this table must be joined (INNER JOIN) to itself. To disambiguate the two instances of the worker table we give one the temporary name “a”, and the other “b”. We can then use these temporary table names as prefixes to make it clear which instance we are referring to. When we use INNER JOIN to join two tables we do not use a WHERE clause, but instead use ON (but it functions in much the same way). We are joining the two instances of worker where they share the same last name and the same factory, but not the same first name. And we want the results sorted by (ORDER BY) last name and factory.

Looking at the results we can begin to make some guesses. John and Reuben Ackersley might be brothers. John Acton might be Elizabeth’s father. Or grandfather. Many Adams siblings seem
to work at factory 216, but it doesn’t seem like their parents do. The current dataset cannot answer these questions definitively, but it gives us a good idea of cases that it might be worth researching further. We might also begin to revise our query; by eliminating people with the same last name and given name did we unintentionally remove sons that were named after their fathers? Perhaps it would be worth running the query again with this criterion removed.

Further Reading

MySQL documentation:
http://dev.mysql.com/doc/

Learning MySQL:
http://shop.oreilly.com/product/9780596008642.do

MySQL Cookbook:
http://shop.oreilly.com/product/0636920032274.do
Database Concepts for Digital Humanists

Harvey Quamen
Associate Professor of English & Humanities Computing
University of Alberta
Edmonton, Alberta, Canada
hquamen@ualberta.ca

Jon Bath
Director, Humanities and Fine Arts Digital Research Centre
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
jon.bath@usask.ca
Documents & Spreadsheets are OK ....

- When “find” is good enough: “When was Virginia Woolf born?”
- When your data is simple: “I’m logging each student’s grades throughout the term.”
- When you don’t have a lot of repeated data (unlike this...!):
A Database is better when....:

• When your questions get more complicated: “Give me a list of UK writers born geographically north of Hull who wrote for periodical magazines, published a novel before age 30, moved to London, succumbed to drug or alcohol abuse, and died before they were 50 and give that to me on a map organized by British county.”

• When you’re looking for patterns rather than specific facts: “Show me family units of three or more people who work at the same factory.”

• When you have lots of repetition in your data:
But the best reason to use databases is because...

- ... databases have query languages — distinctive and different languages that allow you to ask sophisticated questions of datasets.
  
  - **Structured Query Language (SQL)** — queries the individual tables of a relational database. We’ll see a bit of this today. (Note: pronounced as either “ess-que-ell” or “sequel.”)
  
  - **Cypher** — Query language of a cool graph database called Neo4j (note the *Matrix* theme). Based on SQL.
  
  - **XQuery** — an XML-based query language that operates on texts marked up in an XML (eXtensible Markup Language) structure.
  
  - **Programming Languages** — for databases that have no inherent query language and that adhere to a “no SQL” philosophy.
  
  - **Faceted Browser** — A kind of “category browser” that lets users filter data based on various criteria. Not really a query language *per se*. 

Why MySQL?

- Relational databases (which use table-based data structures) are the oldest and most robust database structures around.

- Structured Query Language (SQL) is the original database query language. Subsequent languages either borrow from SQL or challenge its precepts.

- Eventually, you may be able to articulate very good reasons why your particular data set would be better represented by a different data structure or would be easier queried with a different language. Until you can articulate those reasons, though, it’s OK to start with a classic, road-tested and robust technology.
Why Databases?

Supplement

Digital Humanities Summer Institute
Figure 22: Protagonists of Parisian novels, and objects of their desire

★ objects of desire
Figure 12: Authorship of new novels, Britain 1800–1829: gender breakdown (percentage)

The 1810s show an even clearer pattern of female dominance, with women novelists out-producing their male counterparts in every year, and accounting for over 50 per cent of titles in six out of the eight years between 1810 and 1817... As these figures indicate, the publication of Jane Austen's novels was achieved not against the grain but during a period of female ascendancy. It is noticeable that Scott's earliest historical novels were launched when male authorship of fiction was at a lower than usual ebb.

Peter Garside, 'The English Novel in the Romantic Era'

“Traditional” Prosopography

“Prosopography is a collective biography, describing the external features of a population group that the researcher has determined has something in common (profession, social origins, geographic origins, etc.). Starting from a questionnaire biographical data are collected about a well-defined group of people. On the basis of these data answers may be found to historical questions.”

“New Style” Prosopography

“Instead, the evidence data will be recorded as a series of *factoids*—assertions made by the project team that a source ‘S’ at location ‘L’ states something (‘F’) about person ‘P’. . . . A **factoid is not a statement of fact** about a person; a collection of factoids does not record a ‘scholarly overview’ of a person that a scholar has derived from the sources s/he has read. Instead, each one records an assertion by a source at a particular spot about a person. Factoids may contradict each other—if one source says a person was an Armenian, and another that s/he was Bulgarian, then both factoids will be present in the database.”

One person can have many assertions about him/her. One source can produce many assertions.
Bradley & Short, “Texts into Databases: The Evolving Field of New-style Prosopography” (p. 10)
Architecture

HTML / CSS / JavaScript / D3 / Ajax

PHP / Python / Ruby / Perl

database (MySQL, e.g.)
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> USE watson;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM assertion LIMIT 20;

+----+----+---------+---------+----+----+---------+-------+-------+-------+-------+-------+
| id | id1 | id1_table | relationship | id2 | id2_table | source | page_begin | page_end | text  | notes |
|----|----|----------|--------------|----|----------|--------|------------|---------|-------|-------|-------|
| 1  | 2  | person   | NULL         | 1  | person   | 1      | 3          | 4       | NULL  | NULL  | NULL  |
| 2  | 2  | person   | NULL         | 1  | person   | 1      | 31         | 52      | NULL  | NULL  | NULL  |
| 3  | 2  | person   | NULL         | 1  | person   | 1      | 58         | 60      | NULL  | NULL  | NULL  |
| 4  | 2  | person   | NULL         | 1  | person   | 1      | 63         | 66      | NULL  | NULL  | NULL  |
| 5  | 2  | person   | NULL         | 1  | person   | 1      | 68         | 81      | NULL  | NULL  | NULL  |
| 6  | 2  | person   | NULL         | 1  | person   | 1      | 72         | 166     | NULL  | NULL  | NULL  |
| 7  | 2  | person   | NULL         | 1  | person   | 1      | 84         | 171     | NULL  | NULL  | NULL  |
| 8  | 2  | person   | NULL         | 1  | person   | 1      | 170        | 171     | NULL  | NULL  | NULL  |
| 9  | 2  | person   | NULL         | 1  | person   | 1      | 175        | NULL    | NULL  | NULL  | NULL  |
| 10 | 2  | person   | NULL         | 1  | person   | 1      | 178        | 182     | NULL  | NULL  | NULL  |
| 11 | 2  | person   | NULL         | 1  | person   | 1      | 186        | 187     | NULL  | NULL  | NULL  |
| 12 | 2  | person   | NULL         | 1  | person   | 1      | 191        | NULL    | NULL  | NULL  | NULL  |
| 13 | 2  | person   | NULL         | 1  | person   | 1      | 193        | NULL    | NULL  | NULL  | NULL  |
| 14 | 2  | person   | NULL         | 1  | person   | 1      | 196        | 197     | NULL  | NULL  | NULL  |
| 15 | 2  | person   | NULL         | 1  | person   | 1      | 205        | 207     | NULL  | NULL  | NULL  |
| 16 | 2  | person   | NULL         | 1  | person   | 1      | 211        | 212     | NULL  | NULL  | NULL  |
| 17 | 2  | person   | NULL         | 1  | person   | 1      | 217        | NULL    | NULL  | NULL  | NULL  |
| 18 | 2  | person   | NULL         | 1  | person   | 1      | 220        | NULL    | NULL  | NULL  | NULL  |
| 19 | 2  | person   | NULL         | 1  | person   | 1      | 230        | NULL    | NULL  | NULL  | NULL  |
| 20 | 2  | person   | NULL         | 1  | person   | 1      | NULL       | NULL    | NULL  | NULL  | NULL  |

20 rows in set (0.01 sec)

mysql>
Databases:
Intro to Relational Databases & Structured Query Language
A database is a collection of data that is organized and stored according to some purpose.

Defining my purpose precisely (audience, intentions, etc.) will help me make wise decisions later during the planning phases.

We need to store email addresses, and search by members’ names!

So no need for postal codes, but make sure we can alphabetize!
A relational database is organized into tables (which resemble HTML tables or spreadsheets). Each table represents a real-world object or entity. Entities are usually nouns: a person or a university or a contract or even an event.
Each table is organized into rows and columns. Each row in the table is a record, which corresponds to one instance of that real-world entity. If our table stores info about people, for example, then one record would store John Smith’s data.
Databases

An entity usually has several characteristics or attributes. Each column corresponds to one attribute. Attributes of a person might include first name, last name, email address, and/or phone number.
Databases

The intersection of a row and a column (sometimes called a field or a cell) corresponds to one piece of information. John Smith’s email can be located by finding John Smith’s record and reading the matching cell in the email column.
Spreadsheets vs. Databases

A database is like a group of spreadsheets that can talk to one another.

Databases use a special language, Structured Query Language (SQL), to do that.
A database table corresponds to one spreadsheet. A database consists of multiple tables.

The top row of the table contains headers that describe the contents of each column. These headers aren’t data, exactly, but rather metadata — they’re data about the data. They label the attributes, but do nothing more.

Each record holds the data for one of the entity’s instances. Here, a record seems to be a person who owns pets. Each cell holds one piece of data.
Querying the Database Table

I can ask questions of the table, and the database will give me an answer. If I ask, “Tell me the ID number of any person who has no pets,” the database will search the table’s columns and rows to find the answer.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Pets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Sue</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Lisa</td>
<td>0</td>
</tr>
</tbody>
</table>
I can ask questions of the table, and the database will give me an answer. If I ask, “Tell me the ID number of any person who has no pets,” the database will search the table’s columns and rows to find the answer.

Similarly, I could ask, “Tell me the name of any person who owns more than zero pets,” the database will search columns and rows to find that answer too. In this case, the right answer is Bob and Sue.
I can even ask questions that reach across multiple tables, and *that’s what makes a database far more powerful than a spreadsheet*. But I must organize my data very carefully to make use of that power. If my tables are disorganized, I cannot guarantee correct answers.
I can ask, "What’s Sue’s favorite pet’s name?" and query across tables to find the answer. My data has to be properly organized for a query like this to work. We’re about to spend some time learning those principles.
Database Terms

A database is a collection of data that is organized and stored according to some purpose.

The database is organized into tables that capture data about a real-world entity. Entities are nouns: people, places, things, events, etc. An entity might be somewhat abstract. You can think of it as an abstract template that describes concrete examples. (Car or animal might be entities.)

Each table is organized into rows and columns.

Each row in the table is a record, which corresponds to one instance (one example) of an entity. Honda Civic and giraffe are instances of car and animal.

Each column describes an attribute or a characteristic of the entity. Year and colour might be attributes of car. Species and common name might be attributes of animal.

Each cell or field contains one piece of information. In the animal table, the cell in the giraffe record under the species column might say Giraffa camelopardalis.
Database Relationships

- One-to-One
- One-to-Many
- Many-to-Many
Let’s say I’m building an address book. Almost every person I know has an email address. Both “person” and “email address” are good, old-fashioned nouns and they’re good candidates to be entities. Let’s temporarily model things that way.

This relationship is a **one-to-one relationship**, which I can signify by a single straight line between the tables:

Reading left to right: “Each person has one email address.”

Reading right to left: “Each email address belongs to one person.”
1-to-1 Relationships

However, from long experience I know that a one-to-one relationship across two tables is almost always wrong. It’ll simplify things in the long run to combine these into one table.

Therefore, I hereby authorize by divine fiat that email is simply an attribute of person. Let’s simplify our design by combining the tables.

Rule of Thumb: one-to-one relationships can almost always be modeled as attributes in another entity’s table.
1-to-1 Relationships

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>email1</th>
<th>email2</th>
<th>email3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shelly</td>
<td>Johnson</td>
<td><a href="mailto:shell@shaw.ca">shell@shaw.ca</a></td>
<td><a href="mailto:shell@gmail.com">shell@gmail.com</a></td>
<td><a href="mailto:sj@adco.net">sj@adco.net</a></td>
</tr>
</tbody>
</table>

And now what if I want to log multiple emails per person? One possible solution would be to add columns to my table.

That’s how we’d do it in a spreadsheet, after all.

Unless movies and TV have lied to me, that seems like the right thing to do.
1-to-1 Relationships

But most of my friends don’t have multiple email addresses and my good friend Bill, a paranoid sociopath recluse who lives off the grid, doesn’t have one at all.

In this case, I rethink my decision: email is indeed an entity. It needs its own table.

D’oh! Stupid TV.
1-to-1 Relationships

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>email1</th>
<th>email2</th>
<th>email3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shelly</td>
<td>Johnson</td>
<td><a href="mailto:shell@shaw.ca">shell@shaw.ca</a></td>
<td><a href="mailto:shell@gmail.com">shell@gmail.com</a></td>
<td><a href="mailto:sj@adco.net">sj@adco.net</a></td>
</tr>
<tr>
<td>Tom</td>
<td>Simmons</td>
<td><a href="mailto:ts@financial.net">ts@financial.net</a></td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Bill</td>
<td>Warwick</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

There's one big warning sign here suggesting that we could probably make a better decision: **data repetition**. Any kind of repeated data is a symptom of a bigger problem. Here, the repetition of NULL (remember, NULL means “no value”) is a cautionary sign that we have an entity struggling to be free.

**Hint**: Repeated values in a table -- especially lots of NULL values -- suggest that you need to restructure your tables.
**1-to-Many Relationships**

So let’s break out the email addresses into their own table. Now, adding a new email address is as simple as adding a new row in the `email` table. It doesn’t matter anymore if a person has zero or one or fifty email addresses. My table structure can accommodate all of those cases without modification. This is a stable design that will serve me well in the long run.

**Rule of thumb:** Try to structure your tables so that adding new data creates new rows, not additional columns.
This is now called a **one-to-many relationship** and I diagram it differently. The “many” side gets a so-called “crow’s foot.”

Reading left to right: “One person has many email addresses.”

Reading right to left: “An email address belongs to one person.”

**Notice:** No data repetition. Our NULL values have disappeared!
However, I need to build a “bridge” between the two tables. I model that relationship with keys. Every person gets a unique ID number, called a primary key.

I identify every email address by storing the owner’s ID from the person table. In the email table, it’s called a foreign key.

**Primary Key**: a signifier that uniquely identifies a table row.

**Foreign Key**: a key that’s a primary key in some other table.
Primary Key: Requirements

1. Primary keys must have unique values for each row in that table.

2. Primary keys must have a non-NULL value for the entire lifetime of that record.

3. A primary key’s value must never change during the lifetime of that record.

Some primary keys are “exposed” to the public: your social insurance number or your student number, for example. Most of the time, however, they’re never known. They’re just needed internally by the database to maintain data integrity.
1-to-Many Relationships

Databases model relationships with **keys**. When you store a primary key in a different table as a foreign key, you’ve modeled a relationship.

That’s the heart and soul of a relational database — modeling relationships between entities via primary/foreign keys.

**Rule of Thumb**: Store foreign keys on the “crow’s foot” side of the relationship.
Let's take a different example: actors and movies. This is a many-
to-many relationship.
Reading left to right: “Every actor appears in many movies.”
Reading right to left: “Every movie has many actors.”

This diagram — at least as it’s currently depicted — is impossible to model in the database. It will create a list of foreign keys in each table! (See the course packet)
Many-to-Many Relationships

The solution is to insert a junction table, which acts as a warehouse for foreign keys. Here, actor #2 has been in movies #1 and #4.

Notice that the many-to-many relationship has been broken down into 2 one-to-many relationships. As before, we store foreign keys on the crow’s foot side of the relationship.

A junction table’s main purpose is to store foreign keys. It may or may not correspond to a real-world entity.
Many-to-Many Relationships

Our junction table seems to be a real-world entity. We could call it role. An obvious attribute addition to the table is the character name, but we could add more data: salary, reviews, photos, quotations, interviews, talk show appearances, etc.

A junction table that is not a real-world entity often takes the names of the two tables it joins. Here, we’d probably call the junction table something like actor_movie.
A one-to-one relationship can almost always be combined into a single table. It suggests that something you decided was an entity is really just an attribute.

Too much data repetition (including NULL values) is a bad sign. It means that something you decided was a mere attribute is really a full-fledged entity.

Construct your tables so that adding new data creates new rows, not additional columns.

A many-to-many relationship is modeled with a junction table. Junction tables create 2 one-to-many relationships out of a single many-to-many relationship.

Store foreign keys on the “crow’s foot” side of the diagram.

Since relational databases model relationships between entities, repeating foreign keys is both OK and desirable. It’s the goal.
DATABASE DESIGN EXAMPLE

BIRDCLUB

the first rule of birdclub is “don’t talk about birdclub.”
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.
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.
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.

But wait! The current leader will change and I can always count this on-the-fly anytime I want!
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.

But wait! The current leader will change and I can always count this on-the-fly anytime I want!
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.

Design Principle: Don’t store things you can count or calculate!
**MEMBER TABLE**

<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>email</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Design Principle: Store one item per column.**

Here I’ve decided that **first name** is one thing and **last name** is another thing. Since MySQL can count and sort efficiently, I can make an alphabetized list very easily if **first** and **last** names are separate fields.

I give everyone an **id**, although this info might remain private. It’s a unique signifier for each person. Handy, since I have multiple Robert Smiths!
Which Sue? What’s a **Blackgird**? Does **10/4** mean October 4 or April 10? And which year? Were both the Blackgird and crow seen on that date? Or just one? What’s the difference here between a comma and a semi-colon? Does **Ju** mean June or July? And should we require the use of only ordinal numbers (1st, 2nd, 3rd) or cardinal numbers (1, 2, 3)? Or something else?

### Attempt #1: DISASTER!

<table>
<thead>
<tr>
<th>person</th>
<th>sightings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>magpie, Sept 12 Blackgird, 10/4 crow; raven, Ju 14th</td>
</tr>
</tbody>
</table>

Sightings Table

<table>
<thead>
<tr>
<th>person</th>
<th>sightings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>magpie, Sept 12 Blackgird, 10/4 crow; raven, Ju 14th</td>
</tr>
</tbody>
</table>
**Design Principle Violated: Store one item per column.** I’m storing 7 things right now in the sightings column. The field is impossible to search effectively because it contains too much data and that data is not standardized.

**Design Principle Violated: Store data in only one place.** I’m repeating Sue’s name from the member table and I will repeat bird names constantly (and probably misspell them!), which suggests I need a table for **birds**. It’s easy to update data stored in only one place.
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.

member table

sighting table

bird table

OK, let’s add a bird table. This might be a good opportunity for a dropdown menu on my website!
Here’s my bird table. I’ve decided to store both the scientific name (“Pica hudsonia”) as well as the common name (“Black-billed magpie”).

Some data designers might separate the scientific name into genus and species. That’s a fine decision. Here, however, I’ve decided that the whole scientific name simply counts as one thing.
The **sighting** table has two one-to-many relationships:

1. Each member can log many bird sightings.
2. Each bird can be seen many times.
The **sighting** table has two one-to-many relationships:

1. Each member can log many bird sightings.
2. Each bird can be seen many times.
The **sighting** table has two one-to-many relationships:

1. Each member can log many bird sightings.
2. Each bird can be seen many times.
DATES

Databases standardize dates in this format:

**YYYY-MM-DD**

For example,

- 2013-06-10
- 1963-11-22
- 2009-04-00
- 1999-00-00

You can zero-fill any parts of dates that are unknown. The XML community doesn't treat dates in this way, but it's common database practice.
MySQL permits you to store dates where the day or month and day are zero in a `DATE` or `DATETIME` column. This is extremely useful for applications that need to store birthdates for which you may not know the exact date. In this case, you simply store the date as '2009-00-00' or '2009-01-00'. If you store dates such as these, you should not expect to get correct results for functions such as `DATE_SUB()` or `DATE_ADD()` that require complete dates.

MySQL permits you to store a “zero” value of '0000-00-00' as a “dummy date.” This is in some cases more convenient than using NULL values, and uses less data and index space.
SPACES IN NAMES

Caveat: database column and table names must be only one word long.

Consequently, rather than using a column called first name, use first_name instead.

As MySQL breaks down the query language in order to interpret it, MySQL separates the command into separate words where the spaces occur. MySQL isn't smart enough to figure out that a really really long weird column name is really a_really_really_long_weird_column_name.

The same caveat applies to table names. Rather than creating a table called club members, build one called club_members instead.
DATA DESIGN PRINCIPLES

- Major nouns become entity tables. Not all nouns are worthy, however! And some nouns are invisible at first.

- An entity’s salient properties become its column attributes. Sometimes these are adjectives; sometimes these are nouns.

- Store only one thing per field. Don’t store lists of data in one field. Think of adding new data as new rows in a table.

- Store each piece of data in only one place. Data stored in only one place is less vulnerable to errors.

- Don’t store things you can calculate. For example, store a person’s birthdate, not age. MySQL can count and sort!
You Say ‘Sequel,’ I Say ‘Ess-Que-Ell’

A Brief Introduction to
Structured Query Language
Structured Query Language

SQL is an English-like language used to enter, modify, search for, or delete data from a database. (That fact betrays its 1970s roots.)

Its commands are imperative verbs. You are always commanding the database to do something.

SQL statements **always end with a semi-colon**.

SQL (or, more precisely, the data manipulation subset of SQL) really has only four verbs:

- **SELECT** — search for some data that satisfies certain criteria.
- **INSERT** — add a new row of data to a table.
- **UPDATE** — change a cell of data whose row already exists.
- **DELETE** — remove a row of data from a table.
I can ask questions of the table, and the database will give me an answer. If I ask, “Tell me the ID number of any person who has no pets,” the database will search the table’s columns and rows to find the answer.

```
SELECT id FROM number_of_pets WHERE pets = 0;
```

```
SELECT column FROM table WHERE criteria;
```
SQL keywords are traditionally capitalized. Some database software packages require that you always capitalize SQL keywords. MySQL doesn’t, but I’ll usually capitalize them anyway. Recognizing the SQL keywords is easier when you always see them capitalized.

Column names and table names are case sensitive. They always need to be capitalized exactly the same way as when you created them. I almost always use lowercase. **NB: they must be only one word long!**
SELECT pets FROM number_of_pets WHERE name = 'Bob';

SELECT column FROM table WHERE criteria;
Let’s add some data to our table using the INSERT command.
Notice that the order of the column names matches the order of the data exactly.
And notice again that non-numeric data must be enclosed in quotation marks.

```
INSERT INTO number_of_pets ( id, name, pets ) VALUES ( 4, 'Sarah', 1 );
```

```
INSERT INTO table ( list of columns )
VALUES ( list of data );
```
Let's add some data to our table using the **INSERT** command.

Notice that the order of the column names matches the order of the data exactly.

And notice again that non-numeric data must be enclosed in quotation marks.

```sql
INSERT INTO number_of_pets ( id, name, pets ) VALUES ( 4, 'Sarah', 1 );
```

```sql
INSERT INTO table ( list of columns )
VALUES ( list of data );
```
Sarah just bought a new pet. Let’s modify her record using the **UPDATE** command.

**UPDATE number_of_pets**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>pets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Sue</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Lisa</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Sarah</td>
<td>1</td>
</tr>
</tbody>
</table>

**UPDATE number_of_pets SET pets = 2 WHERE id = 4;**

**UPDATE table**

**SET column and data WHERE criteria;**

The criteria in the **WHERE** clause needs to be specific enough to change only the records you want to target!
We can modify multiple columns at the same time.

UPDATE number_of_pets
SET name = 'Susan', pets = 3
WHERE name = 'Sue';

If I have more than one Sue, this WHERE clause might not be specific enough. Using primary keys is safer!
We can modify multiple columns at the same time.

```
UPDATE number_of_pets
SET name = 'Susan', pets = 3
WHERE name = 'Sue';
```

If I have more than one Sue, this `WHERE` clause might not be specific enough. Using primary keys is safer!
Let’s say I want to find out, for each sighting in my database, the member’s name and the bird he or she saw.

Obviously, I can’t figure this out with just one table. I definitely need the sighting table, but it’s just full of foreign keys and doesn’t give me complete information.

So I’ll need to trace those keys to their original tables in order to find out member names (in the member table) and the bird names (in the bird table).

I need a systematic way to perform these multi-table queries.
1.) List all the tables you’ll need.
2.) List all the columns you’ll need (I use a dot!).
3.) Draw boxes around the columns you want to select.
4.) Draw lines between “shared key” columns.
5.) Write the query.
SELECT member.first, member.last, bird.common
FROM member, sighting, bird
WHERE member.id = sighting.person
AND sighting.bird = bird.id;

SELECT columns
FROM tables
WHERE criteria
AND/OR criteria
We can delete records too. Deleting requires extra precautions to make sure your `WHERE` clause is right because . . .

```sql
DELETE FROM number_of_pets
WHERE name = 'Bob';
```

**WARNING**

MySQL does not have an undo feature!
Once Bob's record is gone, it's gone forever.
Why?

DELETE FROM number_of_pets
WHERE name = 'Bob';

... because MySQL does not have an undo feature!
Welcome to a Nightmare

Not afraid?
You will be.
You will be.

Deletion: Welcome to a Nightmare
## Deleting Bob

Let’s say Bob moves to Tatooine and drops out of my club. I might consider just deleting him from my database. It seems logical, since he won’t be entering any more data here.

<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>email</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>Jones</td>
<td><a href="mailto:jonesy@blah.ca">jonesy@blah.ca</a></td>
<td>555-1234</td>
</tr>
<tr>
<td>2</td>
<td>Sue</td>
<td>Rogers</td>
<td><a href="mailto:sr@blah.ca">sr@blah.ca</a></td>
<td>555-9876</td>
</tr>
</tbody>
</table>

- Bob Jones
- Email: jonesy@blah.ca
- Phone: 555-1234
- Sue Rogers
- Email: sr@blah.ca
- Phone: 555-9876
### Bob is Dead to Me

<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>email</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Sue</td>
<td>Rogers</td>
<td><a href="mailto:sr@blah.ca">sr@blah.ca</a></td>
<td>555-9876</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Now my **member** table has no record of Bob at all. Primary key #1 is gone and *will never be used again by my table* (that’s OK — that’s how primary keys are supposed to work).

But I’ve got other problems just waiting for me.
Now the sighting table is a HUGE problem for me. In fact, I’ve got foreign keys pointing to Bob Jones from all over my database, and I have to chase them all down in order to delete them. If I miss even one, I’ll have a potentially broken query.

This is the nightmare of cascading deletes. Perhaps a better option is to modify my tables so that I don’t need to delete Bob.
### Handling Bob

<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>email</th>
<th>phone</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>Jones</td>
<td><a href="mailto:jonesy@blah.ca">jonesy@blah.ca</a></td>
<td>555-1234</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Sue</td>
<td>Rogers</td>
<td><a href="mailto:sr@blah.ca">sr@blah.ca</a></td>
<td>555-9876</td>
<td>yes</td>
</tr>
</tbody>
</table>

What if I add another column to my member table that tells me whether a member is active or not? I could define the column to hold yes/no values with a default of yes. When members drop out, I can simply change the field value to no.

This is a good solution, but I might need to alter some queries.
SELECT member.first, member.last, bird.common
FROM member, sighting, bird
WHERE member.id = sighting.person
AND sighting.bird = bird.id
AND member.active = 'yes';
Deleting Isn’t Always the Best Idea

Deleting primary key records will leave holes in my database that can possibly break some queries that used to work.

Deleting Bob, I’ll lose important data like past monthly winners, total number of birds seen by my club, and perhaps whether anyone ever saw very rare birds.

Adding a new field (active) will expand the number of queries I can ask. Now I can find all sightings by non-active members, if I should want to do that. I can see how many members I’ve had in the history of my club. I can still find out the last time anyone saw Bird X. If I add membership dates, I can chart my membership numbers across time. I can calculate average membership duration. Etc., etc., etc.
### When Deleting Bob is Necessary

<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>email</th>
<th>phone</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>Jones</td>
<td><a href="mailto:jonesy@blah.ca">jonesy@blah.ca</a></td>
<td>555-1234</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Sue</td>
<td>Rogers</td>
<td><a href="mailto:sr@blah.ca">sr@blah.ca</a></td>
<td>555-9876</td>
<td>yes</td>
</tr>
<tr>
<td>17</td>
<td>Bob</td>
<td>Jones</td>
<td><a href="mailto:jonesy@blah.ca">jonesy@blah.ca</a></td>
<td>555-1234</td>
<td>yes</td>
</tr>
</tbody>
</table>

If I’ve accidentally got Bob listed twice in my membership list, then I’ve got a headache that needs to be solved as soon as possible. Now I need to delete one of them, and I need to clean up every table that has foreign key 17.
How to Do a Cascaded Delete

I can modify the tables either by hand or by writing a program to do this:

- Delete the redundant Bob (#17) from member.
- Change all foreign keys #17 to foreign key #1 in every table where #17 might appear.

In SQL, it will look like this:

- `DELETE FROM member WHERE id = 17;`
- `UPDATE sighting SET person = 1 WHERE person = 17;`
- `UPDATE guide SET id = 1 WHERE id = 17;`
Multiple-table queries are the heart of a relational database. Make a chart to help you navigate through the tables.

Every DELETE in a relational database has the potential to become a cascading delete. Think carefully about how you want to handle obsolete data. Leaving it alone is sometimes the best option.

Instead of deleting data, you might want to add more columns. That might invite new queries that you hadn’t considered before.

Next time, we’ll see how you can define your tables to allow or disallow certain kinds or combinations of data. Knowing how MySQL stores data “under the hood” will allow you to leave some data validation and maintenance headaches to the database itself.
End of the Scary Nightmare

BAD THAT WASN’T SO, IT WAS?
Data Types

And other stuff you should know in order to amaze and dazzle your friends at parties after you finally give up that dream of being a magician and stop making ridiculous balloon animals and begin to do something productive instead.
MySQL: an application

Just like you use Word to produce documents ...

... you use MySQL to produce databases.
MySQL is a command-line tool. Only people having administrative privileges can create new databases. On your laptop, that’s you.

Your new database will be a completely empty shell — no tables, no data, nothing.

You’ll need to define everything from that point onwards.
Creating a Database (Step 1)

CREATE DATABASE database_name DEFAULT CHARSET utf8;

This command builds a new, empty database.

We’ve defined the default character set of the database as 8-bit Unicode, which is the new standard for cross-platform, internationalized text.

From here on, any new table will automatically inherit the Unicode character set and we won’t have to worry about it again.

Check out Unicode here: http://unicode.org/
Creating a Database (Step 2)

USE `database_name`;

This command tells MySQL that any new tables created should belong to the new database. This is the equivalent in a word processor of doing `File > New...` — a new document is created and your word processor begins using it. Any text you type will belong to that new document.
When I create my table, I need to tell MySQL which columns I want the table to have:

```
CREATE TABLE member (
    id,
    first,
    last,
    email,
    phone
);
```

But I need to give MySQL very specific information about what kinds of data go into these columns, so I’ll have to learn something about **data types**.
<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>email</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>Jones</td>
<td><a href="mailto:jonesy@blah.ca">jonesy@blah.ca</a></td>
<td>555-1234</td>
</tr>
</tbody>
</table>

This looks suspiciously like a number.

These look suspiciously like sequences of letters.

And I’m not quite sure what this would be.
If I perform the selection in MySQL, I get the following ... :

```
mysql> SELECT 555-1234;
+----------+
| 555-1234 |
+----------+
|     -679 |
+----------+
1 row in set (0.02 sec)
```

... which teaches me several things:

1) I can use MySQL as a calculator;

2) A phone number looks like a number, but it isn’t. It’s really an ordered sequence of characters that cannot be reduced or simplified, more commonly called a string.
# MySQL Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Numbers</strong></td>
<td>We need to tell MySQL whether it has a decimal point or not and how big the number might be.</td>
</tr>
<tr>
<td><strong>Text</strong></td>
<td>We need to tell MySQL either how many characters it might contain or just to treat it as a big chunk of text.</td>
</tr>
<tr>
<td><strong>Date / Time</strong></td>
<td>We need to tell MySQL whether it’s a date, a time, both, or just a part (like a year).</td>
</tr>
<tr>
<td><strong>Complex</strong></td>
<td>Typically lists of some kind; we need to tell MySQL whether one or multiple values of the list will be stored.</td>
</tr>
<tr>
<td><strong>NULL</strong></td>
<td>Not a value, but the absence of a value.</td>
</tr>
<tr>
<td>MySQL Data Types</td>
<td>Numeric</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td></td>
<td>integer – #’s without decimals:</td>
</tr>
<tr>
<td></td>
<td>(tinyint, smallint, mediumint,</td>
</tr>
<tr>
<td></td>
<td>int, bigint); float (float,</td>
</tr>
<tr>
<td></td>
<td>double, decimal)</td>
</tr>
<tr>
<td></td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>char, text and blob in various</td>
</tr>
<tr>
<td></td>
<td>sizes (tiny, medium, long);</td>
</tr>
<tr>
<td></td>
<td>varchar – variable character</td>
</tr>
<tr>
<td></td>
<td>(most common)</td>
</tr>
<tr>
<td></td>
<td>Date / Time</td>
</tr>
<tr>
<td></td>
<td>date, datetime, time, timestamp,</td>
</tr>
<tr>
<td></td>
<td>year</td>
</tr>
<tr>
<td></td>
<td>Complex</td>
</tr>
<tr>
<td></td>
<td>enum (field value is one of a</td>
</tr>
<tr>
<td></td>
<td>specific list); set (field values</td>
</tr>
<tr>
<td></td>
<td>can be multiple values from a</td>
</tr>
<tr>
<td></td>
<td>specific list)</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>Not a value, but the absence of a</td>
</tr>
<tr>
<td></td>
<td>value.</td>
</tr>
</tbody>
</table>
Signed

Unsigned
<table>
<thead>
<tr>
<th>MySQL Numeric Column Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>tinyint</td>
</tr>
<tr>
<td>signed: -128 to 127</td>
</tr>
<tr>
<td>unsigned: 0 to 255</td>
</tr>
<tr>
<td>smallint</td>
</tr>
<tr>
<td>signed: -32768 to 32767</td>
</tr>
<tr>
<td>unsigned: 0 to 65535</td>
</tr>
<tr>
<td>mediumint</td>
</tr>
<tr>
<td>signed: -8388608 to 8388607</td>
</tr>
<tr>
<td>unsigned: 0 to 16777215</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>signed: -2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned: 0 to 4294967295</td>
</tr>
<tr>
<td>bigint</td>
</tr>
<tr>
<td>signed: ± 9223372036854775808</td>
</tr>
<tr>
<td>unsigned: 0 to 18446744073709551615</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>can optionally specify bits of precision</td>
</tr>
<tr>
<td>(default: 4 bytes single; 8 bytes double)</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>8 bytes</td>
</tr>
<tr>
<td>decimal</td>
</tr>
<tr>
<td>number stored as a string; specify display width and number of decimals.</td>
</tr>
</tbody>
</table>
Character string columns are defined by specifying the maximum number of characters to be contained in the column. For example, if I know this is my longest string ...:

Edmonton, Alberta

...then I know that I can specify 17 characters maximum for the column (*note that I have to count spaces and punctuation too!*). A definition of `char(17)` would suffice. I might have shorter strings, though, and so I can tell MySQL to compress space by defining the column as `variable character`: `varchar(17)`. 
<table>
<thead>
<tr>
<th>id</th>
<th>first</th>
<th>last</th>
<th>email</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>Jones</td>
<td><a href="mailto:jonesy@blah.ca">jonesy@blah.ca</a></td>
<td>555-1234</td>
</tr>
</tbody>
</table>

- This is an integer that will never be larger than 250.
- These are varying-length strings. I estimate the longest will be 35 characters.
- This is a string whose longest entry will be 12 characters.
Primary Keys

A primary key is a special designation that describes how each row of the table will be uniquely identified. A primary key can contain one or more columns. Moreover:

- Primary keys must have unique values for each row.
- Primary keys must have a non-NULL value for the entire lifetime of that record.
- A primary key’s value must never change during the lifetime of that record.

I can have MySQL help me here. I can, for example:
- Define an integer column (unsigned may help)
- Force it to be NOT NULL
- Tell MySQL to make it auto_increment
Now I’m ready to define my `member` table. I need to tell MySQL the name of the table, plus the names and precise definitions of all the columns:

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

The `id` field is a TINYINT, and it’s unsigned, so it can store a number from 0 to 255.
Now I’m ready to define my *member* table. I need to tell MySQL the name of the table, plus the names and precise definitions of all the columns:

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

- The **id** field is a TINYINT, and it’s unsigned, so it can store a number from 0 to 255.
- It’s also the primary key, and I’ve established all the necessary criteria.
Now I’m ready to define my **member** table. I need to tell MySQL the name of the table, plus the names and precise definitions of all the columns:

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

- **The id field** is a TINYINT, and it’s unsigned, so it can store a number from 0 to 255.
- **It’s also the primary key**, and I’ve established all the necessary criteria.
- **My other fields** are varchars, ranging from 12 to 35 characters long.
Here’s my **sighting** table. Note that two of these fields are **foreign keys** — that is, **person** and **bird** contain values that are primary keys in their original tables, but here they are simply references to records in those tables. MySQL has support for foreign keys, but we’ll simplify this for now.

<table>
<thead>
<tr>
<th>person</th>
<th>bird</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td><strong>2011-09-28</strong></td>
</tr>
</tbody>
</table>

Foreign keys! I’ll define them with the same qualities they have in their original tables.

A date. I’ll just use the date data type.
CREATE TABLE sighting (
   person tinyint unsigned NOT NULL,
   bird tinyint unsigned NOT NULL,
   date date
);

Foreign keys; make them unsigned tinyint’s – same as the other table.
### More Table Definitions

<table>
<thead>
<tr>
<th>person</th>
<th>bird</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2011-09-28</td>
</tr>
</tbody>
</table>

CREATE TABLE `sighting` (
   `person` tinyint unsigned NOT NULL,
   `bird` tinyint unsigned NOT NULL,
   `date` date
);

Foreign keys; make them unsigned tinyint’s – same as the other table.

It would make no sense to have no person or no bird in a sighting record, so I’ll have MySQL help me here.
CREATE TABLE sighting (  
   person tinyint unsigned NOT NULL,  
   bird tinyint unsigned NOT NULL,  
   date date
);

It would make no sense to have no person or no bird in a sighting record, so I’ll have MySQL help me here.

Just a regular date column. I’ve decided not to store the time.

Foreign keys; make them unsigned tinyint’s – same as the other table.
INSERT revisited
As we’ve seen, `INSERT` creates a brand new row in the table and fills the columns with the data you provide. The more you know about your data types, the more prepared you’ll be to make sure `INSERT` works properly.

The general technique is first to list the columns into which the data goes, and then list the data:

```sql
INSERT INTO sighting (person, bird, date) VALUES (1, 2, '2011-09-30');
```

Notice that:

1. I need to list the **table** into which the data will be inserted.
2. I need to put quotation marks around any string value: ‘2011-09-30’
3. I do **not** need to put quotation marks around numbers.
4. Both the column list and the values list go inside parentheses: ( )
5. The data order matches the column order as listed here.
However, as long as I list the columns and as long as I list my data values correspondingly, I can insert values in any order I wish. I do not need to follow the column order of the table itself. For example:

```
mysql> INSERT INTO sighting (bird, date, person) VALUES (2, '2011-10-10', 4);
Query OK, 1 row affected (0.00 sec)
```

But if I make a mistake in either my column count or my data count, MySQL will generate an error and will not insert data:

```
mysql> INSERT INTO sighting (person, bird, date) VALUES (1);
ERROR 1136: Column count doesn't match value count at row 1
```

Using this technique, I can even skip certain columns if I want. In that case, those columns will either contain a default value (if I defined one in my CREATE TABLE statement) or else they will contain NULL.
Here’s another technique. If you’re feeling really bold, you can omit the column list. Notice, though, that I need to have a piece of data for each column and they need to be specified in the same order as in the table:

```
INSERT INTO sightings VALUES (1, 2, '2011-09-30');
```

If I get the number of data items wrong, I’ll get an error:

```
mysql> INSERT INTO sighting VALUES (1, 3);
ERROR 1136: Column count doesn't match value count at row 1
```

And if I get the columns wrong without specifying them, I have a total mess:

```
mysql> INSERT INTO sighting VALUES ('2011-09-28', 1, 3);
```

Because that query actually inserts these values:

```
255, 1, 2000-00-03
```
Why would this query ...

```
INSERT INTO sighting VALUES ('2011-09-28', 1, 3);
```

... inserted into this table ...

```
CREATE TABLE sighting (
   person tinyint unsigned NOT NULL,
   bird tinyint unsigned NOT NULL,
   date date
);
```

... produce these values?

```
255, 1, 2000-00-03
```

The string gets converted to a number but maxes out the tinyint at 255, 1 goes in untouched, and 3 gets interpreted as year 0, month 0, day 3.
Zero-Filled Dates

Missing dates are often the bane of a database person’s life. As we’ve seen, MySQL uses a standardized date of this format:

YYYY-MM-DD

However, if one or more of those parts are missing, you are allowed to fill them with zeroes. MySQL considers even the bizarre-looking date 0000-00-00 to be legal. So, for example:

0000-09-04  September 4th of an unknown year
2012-06-00  Sometime in June 2012
2009-00-23  The 23rd of an unknown month in 2009

NB: This is not a standard practice in the XML community, so if you’re moving documents into a database, you might need to make some decisions about how your project should handle missing dates.
Inserting Primary Keys

Remember how we defined our primary key in the member table?

id tinyint unsigned NOT NULL auto_increment PRIMARY KEY,

That says the column is an unsigned integer that gets automatically incremented every time a new row gets inserted. Technically, then, I don’t want to insert any value at all into the id field — I want to let MySQL handle it for me:

```
INSERT INTO member (id, first, last) VALUES (NULL, 'Deb', 'Jones');
```

Since, by definition, the column cannot be NULL, MySQL will calculate the next auto_increment number, and insert that value into the field. I don’t need to do any work.

(Caution: not all primary keys are defined as auto_increment integers, so inserting NULL into a primary key field isn’t an automatic answer! But when it works, it’s cool.)
INSERT INTO member (id, first, last) VALUES (NULL, ‘Deb’, ‘Jones’);

MySQL will find the latest ID inserted, add one, and use that number for the new record’s primary key, which you can discover with the `last_insert_id()` function:

```sql
mysql> select last_insert_id();
```

```
+------------------+
| last_insert_id() |
+------------------+
|         10       |
+------------------+
1 row in set (0.22 sec)
```
In a query, every string value needs to be surrounded by quotation marks. But that means a string that itself has a quotation mark or an apostrophe presents a problem. Inserting this string, for example ...

```
I don’t like peas.
```

... will generate this bizarre behaviour:

```
mysql> INSERT INTO delme VALUES ('I don't like peas.');
    '>
```

I need to type a closing quotation mark and then can cancel my query with \c:

```
mysql> INSERT INTO delme VALUES ('I don't like peas.');
    '>' \c
```
In a query, every string value needs to be surrounded by quotation marks. But that means a string that itself has a quotation mark or an apostrophe presents a problem. Inserting this string, for example ...

I don’t like peas.

... will generate this bizarre behaviour:

There are 3 quotation marks; MySQL is waiting for a fourth. It thinks there’s an unterminated string value.

```sql
mysql> INSERT INTO delme VALUES ('I don't like peas.');
> 'c
```

I need to type a closing quotation mark and then can cancel my query with \c:

```sql
mysql> INSERT INTO delme VALUES ('I don't like peas.'); 'c
```
I have two options available to me when inserting a string containing apostrophes or quotations:

1. Wrap the string in double quotation marks:

   mysql> INSERT INTO delme VALUES ("I don't like peas.");

2. Escape the single quotation mark by putting a slash before it. That tells MySQL that the quotation mark does not end the string:

   mysql> INSERT INTO delme VALUES ('I don't like peas.');

Either one is acceptable. You’ll see both quite often in your database career.
Let’s say I define a table this way:

```
CREATE TABLE delme (  
    num int default 7,  
    str varchar(10) default "yoip!"
);
```

<table>
<thead>
<tr>
<th>num</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>default: 7</td>
<td>default: yoip!</td>
</tr>
</tbody>
</table>

What happens when I start to insert values?
**Inserting Default Values**

First insert:

```sql
INSERT INTO delme VALUES (1, 'crikey!');
```

<table>
<thead>
<tr>
<th>num</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>crikey!</td>
</tr>
</tbody>
</table>
Second insert:

```sql
INSERT INTO delme (num) VALUES (2);
```

<table>
<thead>
<tr>
<th>num</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>crikey!</td>
</tr>
<tr>
<td>2</td>
<td>yoip!</td>
</tr>
</tbody>
</table>

Column value unspecified; gets default value yoip!
### Inserting Default Values

Third insert:

```sql
INSERT INTO delme (str) VALUES ('Hello!');
```

<table>
<thead>
<tr>
<th>num</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>crikey!</td>
</tr>
<tr>
<td>2</td>
<td>yoip!</td>
</tr>
<tr>
<td>7</td>
<td>Hello!</td>
</tr>
</tbody>
</table>

Column value unspecified; gets default value 7
CREATE TABLE member (  
id tinyint unsigned NOT NULL auto_increment PRIMARY KEY,  
first varchar(25),  
last varchar(25),  
email varchar(35),  
phone varchar(12),  
guide varchar(3) default 'no'
);
Define Your Tables

Using your database designs, begin defining the data types of the columns. When you’re ready, create a new database on your laptop (CREATE DATABASE) and then begin creating the tables (CREATE TABLE).
Spatial Metaphors in SQL
INNER JOIN

SELECT [...] 
FROM left, right 
WHERE left.id = right.id

Dialect difference: 
the comma is synonymous with INNER JOIN and WHERE is synonymous with ON
SELECT [...] 
left INNER JOIN right 
ON left.id = right.id

Dialect difference: 
the comma is synonymous with INNER JOIN and WHERE is synonymous with ON
```sql
SELECT member.first, member.last, bird.common
FROM member INNER JOIN sighting INNER JOIN bird
ON member.id = sighting.person
AND sighting.bird = bird.id;
```

<table>
<thead>
<tr>
<th>first</th>
<th>last</th>
<th>common</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Smith</td>
<td>Black-billed Magpie</td>
</tr>
<tr>
<td>Bob</td>
<td>Smith</td>
<td>Common Raven</td>
</tr>
<tr>
<td>Cody</td>
<td>Outland</td>
<td>Yellow-headed Blackbird</td>
</tr>
<tr>
<td>Bob</td>
<td>Smith</td>
<td>Red-winged Blackbird</td>
</tr>
<tr>
<td>Sue</td>
<td>McKinney</td>
<td>Bullock's Oriole</td>
</tr>
<tr>
<td>Fernando</td>
<td>Kinzer</td>
<td>Common Raven</td>
</tr>
<tr>
<td>Steve</td>
<td>Martin</td>
<td>Brown-headed Cowbird</td>
</tr>
<tr>
<td>Guy</td>
<td>Madril</td>
<td>Western Meadowlark</td>
</tr>
<tr>
<td>Sue</td>
<td>McKinney</td>
<td>Baltimore Oriole</td>
</tr>
<tr>
<td>Cody</td>
<td>Outland</td>
<td>Brewer's Blackbird</td>
</tr>
<tr>
<td>Kurt</td>
<td>Matis</td>
<td>Purple Finch</td>
</tr>
<tr>
<td>Bob</td>
<td>Smith</td>
<td>Rusty Blackbird</td>
</tr>
<tr>
<td>Robert</td>
<td>Smith</td>
<td>Common Grackle</td>
</tr>
<tr>
<td>Sue</td>
<td>McKinney</td>
<td>Red-winged Blackbird</td>
</tr>
<tr>
<td>Allan</td>
<td>Moffet</td>
<td>Field Sparrow</td>
</tr>
<tr>
<td>Max</td>
<td>Keown</td>
<td>Rusty Blackbird</td>
</tr>
</tbody>
</table>
MySQL does not have an OUTER JOIN syntax. However, it does have the UNION keyword. Oddly, you have to select the same number of columns from each table.
SELECT [...] 
FROM left LEFT JOIN right 
ON left.id = right.id
SELECT [...] 
FROM left LEFT JOIN right 
ON left.id = right.id 
WHERE right.id IS NULL
Right Join

guide

member
SELECT * FROM guide 
RIGHT JOIN member 
ON guide.id = member.id 
WHERE guide.id IS NULL;

But I find this query totally counterintuitive. I’m actually selecting nothing from the guide table!
Spatial Metaphors in SQL
The Future

or,

What Do I Need to Keep My Eye On?
SELECT last
FROM member
WHERE name = 'Bob';
SELECT AVG(rating) FROM rating ;

cell or row

needle in a haystack

calculation

patterns in data
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;
SELECT DISTINCT(a.factory_num) AS factory, b.last_name . . .
FROM worker AS a INNER JOIN worker AS b
ON a.last_name = b.last_name . . .

needle in a haystack

patterns in data

cell or row
calculation
distribution
SQL pattern
It’s likely that SQL has a limited capacity to represent complex data patterns internally (i.e., in terms of sets). Eventually we’ll reach a point beyond which SQL isn’t expressive enough for our needs.

If I were busy querying patterns in data (family units, social networks, etc.), I’d already start supplementing SQL with a scripting language (PHP, Python, Ruby, Perl, etc.).
SQL databases won’t disappear in the near future, but there are two “future database” options that look exciting and are worth exploring:

1. NoSQL Databases
2. Semantic Web

needle in a haystack
CouchDB stores data as a JSON record: "JavaScript Object Notation."

All column metadata & cell values are replaced by **key: value** pairs. There are no NULL values because then the key/value pair is just omitted.
There are no relational tables because related information is nested inside the record itself. Here, that’s batter.
These databases have abandoned SQL. Instead, they query the records with a traditional programming language.

Since CouchDB uses JSON objects to store data, it weaves seamlessly with the Javascript programming language.

Queries are accomplished in a two-step process. Both steps need to be programmed in Javascript:

1. **Map** — find records that are suitable targets from which to extract data to answer the query.

2. **Reduce** — extract the appropriate data from those records and process it (build a list, add numbers, calculate an average, etc.)

Consequently, these are sometimes called **map/reduce databases**. It’s the technique of choice for huge projects like Google.
Traditional web searches remain at the syntactic level of language and those searches remain binary. Is the word coffee present or absent on this page?

The semantic web invents a language so that the meaning of pages is searchable. The language can be expressed in a variety of ways, but the basic gist is a three-part scheme, sometimes called a triple:

Subject — Predicate — Object

Examples:

DHSI — is at — University of Victoria

University of Victoria — is in — Victoria

Victoria — is in — British Columbia

British Columbia — is in — Canada
Examples:

This blog — is written by — John Smith

John Smith — is — politically left-wing

This blog — is critical of — Stephen Harper

Stephen Harper — is — Prime Minister

Searching pages like this means that I can now search for tone, attitude, praise or support, argument, meaning, references, etc.
One way to write semantic web content is in eXtensible Markup Language (XML). The resulting language is called **Resource Description Framework** or RDF. It’s not pretty, but it’s designed to be machine-readable, not human-readable:

```xml
<gn:Feature rdf:about="http://sws.geonames.org/6174041/'">
  <gn:name>Victoria</gn:name>
  <gn:countryCode>CA</gn:countryCode>
  <gn:population>289625</gn:population>
  <wgs84_pos:lat>48.43294</wgs84_pos:lat>
  <wgs84_pos:long>-123.3693</wgs84_pos:long>
</gn:Feature>
```

GeoNames feature 6174041 — has name — Victoria
GeoNames feature 6174041 — has countryCode — CA
GeoNames feature 6174041 — has Population — 289625
GeoNames feature 6174041 — has latitude — 48.43294
GeoNames feature 6174041 — has longitude — -123.3693
I can also draw semantic web sentences as a graph. The subject and object are **nodes** on the graph and the predicate is an **edge** between them.
It’s possible to store semantic web triples in a regular SQL database. You can query the triples via SQL and you can output in a variety of formats, including RDF that’s written in XML.

However, there are more powerful, specialized databases that are colloquially called **triplestores**. They use an enhanced version of SQL called **SPARQL**, which is a comically recursive acronym that stands for “SPARQL Protocol and RDF Query Language”.
Wikipedia currently lists 35 different triplestore database packages written in a variety of computer languages. Some of the most popular:

- Sesame
- AllegroGraph
- Redland
- Mulgara

If you want to experiment with SPARQL and the semantic web, there's a semantic web version of Wikipedia called DBpedia:

http://dbpedia.org/
Summary: Plus

Relational Databases won’t go away anytime soon:

- They are a mature, robust technology, optimized for speed.
- They use an English-like query language that’s easier to use than equivalent XML technologies like XQuery, XPath or XSLT.
- They scale well and store data efficiently.
- They’re great for storing raw data that wasn’t in document format to begin with.
- They import and export easily to and from CSV, plain text, and XML.
- They work with data in its most atomic form. (XML = molecule)
- They are a column-centric data structure. XML is a row-centric data structure. (Quamen’s new revelation about financial docs.)
But:

• SQL has limited capacity to express complex semantic meaning. (Quamen’s argument: that limitation is not a property of the databases, but rather of the language(s) we use to query them.)

• Relational databases assume that data will occur in regular patterns.

• They do not work well for recursive data structures. (e.g., “a folder contains some documents and a folder which also contains some documents and a folder which also . . . etc.”)