Beautiful Data

The Stories Behind Elegant Data Solutions

Edited by Toby Segaran
& Jeff Hammerbacher

O'REILLY®

Copyrighted material

Beautiful Data

The Stories Behind Elegant Data Solutions

Edited by Toby Segaran
& Jeff Hammerbacher

O'REILLY®

Copyrighted material

Beautiful Data
Edited by Toby Segaran and Jeff Hammerbacher

Copyright © 2009 O'Reilly Media, Inc. All rights reserved. Printed in Canada.

Published by O'Reilly Media, Inc. 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Julie Steele
Production Editor: Rachel Monaghan
Copyeditor: Genevieve d'Entremont
Indexer: Angela Howard
Proofreader: Rachel Monaghan
Cover Designer: Mark Paglietti
Interior Designer: Marcia Friedman
Illustrator: Robert Romano

Printing History:

July 2009: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Beautiful Data*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-15711-1
[F]

Copyrighted material

CONTENTS

PREFACE xi

1 SEEING YOUR LIFE IN DATA 1
by Nathan Yau

Personal Environmental Impact Report (PEIR) 2
your.flowingdata (YFD) 3
Personal Data Collection 3
Data Storage 5
Data Processing 6
Data Visualization 7
The Point 14
How to Participate 15

2 THE BEAUTIFUL PEOPLE: KEEPING USERS IN MIND WHEN DESIGNING DATA COLLECTION METHODS 17
by Jonathan Follett and Matthew Holm

Introduction: User Empathy Is the New Black 17
The Project: Surveying Customers About a New Luxury Product 19
Specific Challenges to Data Collection 19
Designing Our Solution 21
Results and Reflection 31

3 EMBEDDED IMAGE DATA PROCESSING ON MARS 35
by J. M. Hughes

Abstract 35
Introduction 35
Some Background 37
To Pack or Not to Pack 40
The Three Tasks 42
Slotting the Images 43
Passing the Image: Communication Among the Three Tasks 46
Getting the Picture: Image Download and Processing 48
Image Compression 50
Downlink, or, It's All Downhill from Here 52
Conclusion 52

Copyrighted material

4 CLOUD STORAGE DESIGN IN A PNUTSHELL 55
by Brian F. Cooper, Raghu Ramakrishnan, and Utkarsh Srivastava

Introduction 55
Updating Data 57
Complex Queries 64
Comparison with Other Systems 68
Conclusion 71

5 INFORMATION PLATFORMS AND THE RISE OF THE DATA SCIENTIST 73
by Jeff Hammerbacher

Libraries and Brains 73
Facebook Becomes Self-Aware 74
A Business Intelligence System 75
The Death and Rebirth of a Data Warehouse 77
Beyond the Data Warehouse 78
The Cheetah and the Elephant 79
The Unreasonable Effectiveness of Data 80
New Tools and Applied Research 81
MAD Skills and Cosmos 82
Information Platforms As Dataspaces 83
The Data Scientist 83
Conclusion 84

6 THE GEOGRAPHIC BEAUTY OF A PHOTOGRAPHIC ARCHIVE 85
by Jason Dykes and Jo Wood

Beauty in Data: Geograph 86
Visualization, Beauty, and Treemaps 89
A Geographic Perspective on Geograph Term Use 91
Beauty in Discovery 98
Reflection and Conclusion 101

7 DATA FINDS DATA 105
by Jeff Jonas and Lisa Sokol

Introduction 105
The Benefits of Just-in-Time Discovery 106
Corruption at the Roulette Wheel 107
Enterprise Discoverability 111
Federated Search Ain't All That 111
Directories: Priceless 113
Relevance: What Matters and to Whom? 115
Components and Special Considerations 115
Privacy Considerations 118
Conclusion 118

Copyrighted material

8	PORTABLE DATA IN REAL TIME *by Jud Valeski*	119
	Introduction	119
	The State of the Art	120
	Social Data Normalization	128
	Conclusion: Mediation via Gnip	131
9	SURFACING THE DEEP WEB *by Alon Halevy and Jayant Madhaven*	133
	What Is the Deep Web?	133
	Alternatives to Offering Deep-Web Access	135
	Conclusion and Future Work	147
10	BUILDING RADIOHEAD'S HOUSE OF CARDS *by Aaron Koblin with Valdean Klump*	149
	How It All Started	149
	The Data Capture Equipment	150
	The Advantages of Two Data Capture Systems	154
	The Data	154
	Capturing the Data, aka "The Shoot"	155
	Processing the Data	160
	Post-Processing the Data	160
	Launching the Video	161
	Conclusion	164
11	VISUALIZING URBAN DATA *by Michal Migurski*	167
	Introduction	167
	Background	168
	Cracking the Nut	169
	Making It Public	174
	Revisiting	178
	Conclusion	181
12	THE DESIGN OF SENSE.US *by Jeffrey Heer*	183
	Visualization and Social Data Analysis	184
	Data	186
	Visualization	188
	Collaboration	194
	Voyagers and Voyeurs	199
	Conclusion	203

Copyrighted material

13	WHAT DATA DOESN'T DO *by Coco Krumme*	205
	When Doesn't Data Drive?	208
	Conclusion	217
14	NATURAL LANGUAGE CORPUS DATA *by Peter Norvig*	219
	Word Segmentation	221
	Secret Codes	228
	Spelling Correction	234
	Other Tasks	239
	Discussion and Conclusion	240
15	LIFE IN DATA: THE STORY OF DNA *by Matt Wood and Ben Blackburne*	243
	DNA As a Data Store	243
	DNA As a Data Source	250
	Fighting the Data Deluge	253
	The Future of DNA	257
16	BEAUTIFYING DATA IN THE REAL WORLD *by Jean-Claude Bradley, Rajarshi Guha, Andrew Lang, Pierre Lindenbaum, Cameron Neylon, Antony Williams, and Egon Willighagen*	259
	The Problem with Real Data	259
	Providing the Raw Data Back to the Notebook	260
	Validating Crowdsourced Data	262
	Representing the Data Online	263
	Closing the Loop: Visualizations to Suggest New Experiments	271
	Building a Data Web from Open Data and Free Services	274
17	SUPERFICIAL DATA ANALYSIS: EXPLORING MILLIONS OF SOCIAL STEREOTYPES *by Brendan O'Connor and Lukas Biewald*	279
	Introduction	279
	Preprocessing the Data	280
	Exploring the Data	282
	Age, Attractiveness, and Gender	285
	Looking at Tags	290
	Which Words Are Gendered?	294
	Clustering	295
	Conclusion	300

Copyrighted material

18	BAY AREA BLUES: THE EFFECT OF THE HOUSING CRISIS *by Hadley Wickham, Deborah F. Swayne, and David Poole*	303
	Introduction	303
	How Did We Get the Data?	304
	Geocoding	305
	Data Checking	305
	Analysis	306
	The Influence of Inflation	307
	The Rich Get Richer and the Poor Get Poorer	308
	Geographic Differences	311
	Census Information	314
	Exploring San Francisco	318
	Conclusion	319
19	BEAUTIFUL POLITICAL DATA *by Andrew Gelman, Jonathan P. Kastellec, and Yair Ghitza*	323
	Example 1: Redistricting and Partisan Bias	324
	Example 2: Time Series of Estimates	326
	Example 3: Age and Voting	328
	Example 4: Public Opinion and Senate Voting on Supreme Court Nominees	328
	Example 5: Localized Partisanship in Pennsylvania	330
	Conclusion	332
20	CONNECTING DATA *by Toby Segaran*	335
	What Public Data Is There, Really?	336
	The Possibilities of Connected Data	337
	Within Companies	338
	Impediments to Connecting Data	339
	Possible Solutions	343
	Conclusion	348
	CONTRIBUTORS	349
	INDEX	357

Copyrighted material

Copyrighted material

Preface

When we were first approached with the idea of a follow-up to *Beautiful Code*, this time about data, we found the idea exciting and very ambitious. Collecting, visualizing, and processing data now touches every professional field and so many aspects of daily life that a great collection would have to be almost unreasonably broad in scope. So we contacted a highly diverse group of people whose work we admired, and were thrilled that so many agreed to contribute.

This book is the result, and we hope it captures just how wide-ranging (and beautiful) working with data can be. In it you'll learn about everything from fighting with governments to working with the Mars lander; you'll learn how to use statistics programs, make visualizations, and remix a Radiohead video; you'll see maps, DNA, and something we can only really call "data philosophy."

The royalties for this book are being donated to Creative Commons and the Sunlight Foundation, two organizations dedicated to making the world better by freeing data. We hope you'll consider how your own encounters with data shape the world.

Copyrighted material

How This Book Is Organized

The chapters in this book follow a loose arc from data collection through data storage, organization, retrieval, visualization, and finally, analysis.

Chapter 1, *Seeing Your Life in Data*, by Nathan Yau, looks at the motivations and challenges behind two projects in the emerging field of personal data collection.

Chapter 2, *The Beautiful People: Keeping Users in Mind When Designing Data Collection Methods*, by Jonathan Follett and Matthew Holm, discusses the importance of trust, persuasion, and testing when collecting data from humans over the Web.

Chapter 3, *Embedded Image Data Processing on Mars*, by J. M. Hughes, discusses the challenges of designing a data processing system that has to work within the constraints of space travel.

Chapter 4, *Cloud Storage Design in a PNUTShell*, by Brian F. Cooper, Raghu Ramakrishnan, and Utkarsh Srivastava, describes the software Yahoo! has designed to turn its globally distributed data centers into a universal storage platform for powering modern web applications.

Chapter 5, *Information Platforms and the Rise of the Data Scientist*, by Jeff Hammerbacher, traces the evolution of tools for information processing and the humans who power them, using specific examples from the history of Facebook's data team.

Chapter 6, *The Geographic Beauty of a Photographic Archive*, by Jason Dykes and Jo Wood, draws attention to the ubiquity and power of colorfully visualized spatial data collected by a volunteer community.

Chapter 7, *Data Finds Data*, by Jeff Jonas and Lisa Sokol, explains a new approach to thinking about data that many may need to adopt in order to manage it all.

Chapter 8, *Portable Data in Real Time*, by Jud Valeski, dives into the current limitations of distributing social and location data in real time across the Web, and discusses one potential solution to the problem.

Chapter 9, *Surfacing the Deep Web*, by Alon Halevy and Jayant Madhavan, describes the tools developed by Google to make searchable the data currently trapped behind forms on the Web.

Chapter 10, *Building Radiohead's House of Cards*, by Aaron Koblin with Valdean Klump, is an adventure story about lasers, programming, and riding on the back of a bus, and ending with an award-winning music video.

Chapter 11, *Visualizing Urban Data*, by Michal Migurski, details the process of freeing and beautifying some of the most important data about the world around us.

Chapter 12, *The Design of Sense.us*, by Jeffrey Heer, recasts data visualizations as social spaces and uses this new perspective to explore 150 years of U.S. census data.

Copyrighted material

Chapter 13, *What Data Doesn't Do*, by Coco Krumme, looks at experimental work that demonstrates the many ways people misunderstand and misuse data.

Chapter 14, *Natural Language Corpus Data*, by Peter Norvig, takes the reader through some evocative exercises with a trillion-word corpus of natural language data pulled down from across the Web.

Chapter 15, *Life in Data: The Story of DNA*, by Matt Wood and Ben Blackburne, describes the beauty of the data that is DNA and the massive infrastructure required to create, capture, and process that data.

Chapter 16, *Beautifying Data in the Real World*, by Jean-Claude Bradley, Rajarshi Guha, Andrew Lang, Pierre Lindenbaum, Cameron Neylon, Antony Williams, and Egon Willighagen, shows how crowdsourcing and extreme transparency have combined to advance the state of drug discovery research.

Chapter 17, *Superficial Data Analysis: Exploring Millions of Social Stereotypes*, by Brendan O'Connor and Lukas Biewald, shows the correlations and patterns that emerge when people are asked to anonymously rate one another's pictures.

Chapter 18, *Bay Area Blues: The Effect of the Housing Crisis*, by Hadley Wickham, Deborah F. Swayne, and David Poole, guides the reader through a detailed examination of the recent housing crisis in the Bay Area using open source software and publicly available data.

Chapter 19, *Beautiful Political Data*, by Andrew Gelman, Jonathan P. Kastellec, and Yair Ghitza, shows how the tools of statistics and data visualization can help us gain insight into the political process used to organize society.

Chapter 20, *Connecting Data*, by Toby Segaran, explores the difficulty and possibilities of joining together the vast number of data sets the Web has made available.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Copyrighted material

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Beautiful Data*, edited by Toby Segaran and Jeff Hammerbacher. Copyright 2009 O'Reilly Media, Inc., 978-0-596-15711-1."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9780596157111

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://oreilly.com

Copyrighted material

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

Copyrighted material

Copyrighted material

CHAPTER ONE

Seeing Your Life in Data

Nathan Yau

In the not-too-distant past, the Web was about sharing, broadcasting, and distribution. But the tide is turning: the Web is moving toward the individual. Applications spring up every month that let people track, monitor, and analyze their habits and behaviors in hopes of gaining a better understanding about themselves and their surroundings. People can track eating habits, exercise, time spent online, sexual activity, monthly cycles, sleep, mood, and finances online. If you are interested in a certain aspect of your life, chances are that an application exists to track it.

Personal data collection is of course nothing new. In the 1930s, Mass Observation, a social research group in Britain, collected data on various aspects of everyday life—such as beards and eyebrows, shouts and gestures of motorists, and behavior of people at war memorials—to gain a better understanding about the country. However, data collection methods have improved since 1930. It is no longer only a pencil and paper notepad or a manual counter. Data can be collected automatically with mobile phones and handheld computers such that constant flows of data and information upload to servers, databases, and so-called data warehouses at all times of the day.

With these advances in data collection technologies, the data streams have also developed into something much heftier than the tally counts reported by Mass Observation participants. Data can update in real-time, and as a result, people want up-to-date information.

Copyrighted material

It is not enough to simply supply people with gigabytes of data, though. Not everyone is a statistician or computer scientist, and not everyone wants to sift through large data sets. This is a challenge that we face frequently with personal data collection.

While the types of data collection and data returned might have changed over the years, individuals' needs have not. That is to say that individuals who collect data about themselves and their surroundings still do so to gain a better understanding of the information that lies within the flowing data. Most of the time we are not after the numbers themselves; we are interested in what the numbers mean. It is a subtle difference but an important one. This need calls for systems that can handle personal data streams, process them efficiently and accurately, and dispense information to nonprofessionals in a way that is understandable and useful. We want something that is more than a spreadsheet of numbers. We want the story in the data.

To construct such a system requires careful design considerations in both analysis and aesthetics. This was important when we implemented the Personal Environmental Impact Report (PEIR), a tool that allows people to see how they affect the environment and how the environment affects them on a micro-level; and your.flowingdata (YFD), an in-development project that enables users to collect data about themselves via Twitter, a microblogging service.

For PEIR, I am the frontend developer, and I mostly work on the user interface and data visualization. As for YFD, I am the only person who works on it, so my responsibilities are a bit different, but my focus is still on the visualization side of things. Although PEIR and YFD are fairly different in data type, collection, and processing, their goals are similar. PEIR and YFD are built to provide information to the individual. Neither is meant as an endpoint. Rather, they are meant to spur curiosity in how everyday decisions play a big role in how we live and to start conversations on personal data. After a brief background on PEIR and YFD, I discuss personal data collection, storage, and analysis with this idea in mind. I then go into depth on the design process behind PEIR and YFD data visualizations, which can be generalized to personal data visualization as a whole. Ultimately, we want to show individuals the beauty in their personal data.

Personal Environmental Impact Report (PEIR)

PEIR is developed by the Center for Embedded Networked Sensing at the University of California at Los Angeles, or more specifically, the Urban Sensing group. We focus on using everyday mobile technologies (e.g., cell phones) to collect data about our surroundings and ourselves so that people can gain a better understanding of how they interact with what is around them. For example, DietSense is an online service that allows people to self-monitor their food choices and further request comments from dietary specialists; Family Dynamics helps families and life coaches document key features of a family's daily interactions, such as colocation and family meals; and Walkability helps residents and pedestrian advocates make observations and voice their concerns about neighborhood

Copyrighted material

walkability and connections to public transit.[*] All of these projects let people get involved in their communities with just their mobile phones. We use a phone's built-in sensors, such as its camera, GPS, and accelerometer, to collect data, which we use to provide information.

PEIR applies similar principles. A person downloads a small piece of software called Campaignr onto his phone, and it runs in the background. As he goes about his daily activities—jogging around the track, driving to and from work, or making a trip to the grocery store, for example—the phone uploads GPS data to PEIR's central servers every two minutes. This includes latitude, longitude, altitude, velocity, and time. We use this data to estimate an individual's impact on and exposure to the environment. Environmental pollution sensors are not required. Instead, we use what is already available on many mobile phones—GPS—and then pass this data with context, such as weather, into established environmental models. Finally, we visualize the environmental impact and exposure data. The challenge at this stage is to communicate meaning in data that is unfamiliar to most. What does it mean to emit 1,000 kilograms of carbon in a week? Is that a lot or is that a little? We have to keep the user and purpose in mind, as they drive the system design from the visualization down to the data collection and storage.

your.flowingdata (YFD)

While PEIR uses a piece of custom software that runs in the background, YFD requires that users actively enter data via Twitter. Twitter is a microblogging service that asks a very simple question: *what are you doing right now?* People can post, or more appropriately, *tweet*, what they are doing via desktop applications, email, instant messaging, and most importantly (as far as YFD is concerned), SMS, which means people can tweet with their mobile phones.

YFD uses Twitter's ubiquity so that people can tweet personal data from anywhere they can send SMS messages. Users can currently track eating habits, weight, sleep, mood, and when they go to the bathroom by simply posting tweets in a specific format. Like PEIR, YFD shows users that it is the little things that can have a profound effect on our way of life. During the design process, again, we keep the user in mind. What will keep users motivated to manually enter data on a regular basis? How can we make data collection as painless as possible? What should we communicate to the user once the data has been logged? To this end, I start at the beginning with data collection.

Personal Data Collection

Personal data collection is somewhat different from scientific data gathering. Personal data collection is usually less formal and does not happen in a laboratory under controlled conditions. People collect data in the real world where there can be interruptions, bad network connectivity, or limited access to a computer. Users are not necessarily data experts, so when something goes wrong (as it inevitably will), they might not know how to adjust.

* CENS Urban Sensing, *http://urban.cens.ucla.edu/*

Copyrighted material

Therefore, we have to make data collection as simple as possible for the user. It should be unobtrusive, intuitive, and easy to access so that it is more likely that data collection becomes a part of the daily routine.

Working Data Collection into Routine

This is one of the main reasons I chose Twitter as YFD's data proxy from phone or computer to the database. Twitter allows users to post tweets via several outlets. The ability to post tweets via mobile phone lets users log data from anywhere their phones can send SMS messages, which means they can document something as it happens and do not have to wait until they have access to a computer. A person will most likely forget if she has to wait. Accessibility is key.

One could accomplish something similar with email instead of Twitter since most mobile phones let people send SMS to an email address, and this was in fact the original implementation of YFD. However, we go back to data collection as a natural part of daily routine. Millions of people already use Twitter regularly, so part of the challenge is already relieved. People do use email frequently as well, and it is possible they are more comfortable with it than Twitter, but the nature of the two is quite different. On Twitter, people update several times a day to post what they are doing. Twitter was created for this single purpose. Maybe a person is eating a sandwich, going out for a walk, or watching a movie. Hundreds of thousands tweet this type of information every day. Email, on the other hand, lends itself to messages that are more substantial. Most people would not email a friend to tell them they are watching a television program—especially not every day or every hour.

By using Twitter, we get this posting regularity that hopefully transfers to data collection. I tried to make data logging on YFD feel the same as using Twitter. For instance, if someone eats a salami sandwich, he sends a message: "ate salami sandwich." Data collection becomes conversational in this way. Users do not have to learn a new language like SQL. Instead, they only have to remember keywords followed by the value. In the previous example, the keyword is *ate* and the value is *salami sandwich*. To track sleep, a user simply sends a keyword: *goodnight* when going to sleep and *gmorning* when waking.

In some ways, posting regularity with PEIR was less challenging than with YFD. Because PEIR collects data automatically in the background, the user just has to start the software on his phone with a few presses of a button. Development of that software came with its own difficulties, but that story is really for a different article.

Asynchronous data collection

For both PEIR and YFD, we found that asynchronous data collection was actually necessary. People wanted to enter and upload data after the event(s) of interest had occurred. On YFD, people wanted to be able to add a timestamp to their tweets, and PEIR users wanted to upload GPS data manually.

Copyrighted material

As said before, the original concept of YFD was that people would enter data only when something occurred. That was the benefit and purpose of using Twitter. However, many people did not use Twitter via their mobile phone, so they would have to wait until a computer was available. Even those who did send SMS messages to Twitter often forgot to log data; some people just wanted to enter all of their data at the end of the day.

Needless to say, YFD now supports timestamps. It was still important that data entry syntax was as close to conversational as possible. To accommodate this, users can append the time to any of their tweets. For example, "ate roast chicken and potatoes at 6:00pm" or "goodnight at 23:00." The timestamp syntax is to simply append "at hh:mm" to the end of a tweet. I also found it useful to support both standard and military time formats. Finally, when a user enters a timestamp, YFD will record the most recent occurrence of the time, so in the previous "goodnight" example, YFD would enter the data point for the previous night.

PEIR was also originally designed only for "in the moment" data collection. As mentioned before, Campaignr runs on a user's mobile phone and uploads GPS data periodically (up to every 20 seconds) to our central server. This adds up to hundreds of thousands of data points for a single user who runs PEIR every day with very little effort from the user's side. Once the PEIR application is installed on a phone, a user simply starts the application with a couple of button presses. However, almost right from the beginning, we found we could not rely on having a network connection 100% of the time, since there are almost always areas where there is no signal from the service carrier. The simplest, albeit naive, approach would be to collect and upload data only when the phone has a connection, but we might lose large chunks of data. Instead, we use a cache to store data on a phone's local memory until connectivity resumes. We also provide a second option to collect data without any synchronous uploading at all.

The takeaway point is that it is unreasonable to expect people to collect data for events at the time they happen. People forget or it is inconvenient at the time. In any case, it is important that users are able to enter data later on, which in turn affects the design of the next steps in the data flow.

Data Storage

For both YFD and PEIR, it was important to keep in mind what we were going to do with the data once it was stored. Oftentimes, database mechanisms and schemas are decided on a whim, and the researchers regret it further down the road, either because their choice makes it hard to process the data or because the database is not extensible. The choice for YFD was not particularly difficult. We use MySQL for other projects, and YFD involves mostly uncomplicated `insert` and `select` statements, so it was easy to set up. Also, data is manually entered—not continuously uploaded like PEIR—so the size of database tables is not an issue in these early stages of development. The main concern was that I wanted to be able to extend the schema when I added new trackers, so I created the schema with that in mind.

Copyrighted material

PEIR, on the other hand, required more careful database development. We perform thousands of geography-based computations every few minutes, so we used PostGIS to add support for geographic objects to a PostgreSQL database. Although MySQL offers GIS and spatial extensions, we decided that PostGIS with PostgreSQL was more robust for PEIR's needs.

This is perhaps oversimplifying our database design process, however. I should back up a bit. We are a group of 10 or so graduate students with our own research interests, and as expected, work on individual components of PEIR. This affected how we work a great deal. PEIR data was very scattered to begin with. We did not use a unified database schema; we created multiple databases as we needed them, and did not follow any specific design patterns. If anyone joined PEIR during this mid-early stage, he would have been confused by where and what all the data was and who to contact to find out. I say this because I joined the PEIR project midway. To alleviate this scattered problem, we eventually froze all development, and one person who had his hand in all parts of PEIR skillfully pieced everyone's code and database tables together. It became quite clear that this consolidation of code and schemas was necessary once user experience development began. In retrospect, it would have been worth the extra effort to take a more calculated approach to data storage in the early goings, but such is the nature of graduate studies.

Coordination and code consolidation are not an issue with YFD, since there is only one developer. I can change the database schema, user interface, and data collection mechanism with little fuss. I also use Django, a Python web framework, which uses a model-view-control approach and allows for rapid and efficient development. I do, however, have to do everything myself. Because of the group's diversity in statistics, computer science, engineering, GIS, and environmental science, PEIR is able to accomplish more—most notably in the area of data processing, as discussed in the next section. So there are certainly advantages and disadvantages to developing with a large group.

Data Processing

Data processing is the important underpinning of the personal data collection system that users almost never see and usually are not interested in. They tend to be more interested in the results of the processing. This is the case for YFD. PEIR users, on the other hand, benefit from seeing how their data is processed, and it in turn affects the way they interpret impact and exposure.

The analytical component of PEIR consists of a series of server-side processing steps that start with GPS data to estimate impact and exposure. To be precise, we can divide the processing into four separate phases:*

* PEIR, *http://peir.cens.ucla.edu*

Copyrighted material

1. **Trace correction and annotation:** Where possible, the error-prone, undersampled location traces are corrected and annotated using estimation techniques such as map matching with road network and building parcel data. Because these corrections and annotations are estimates, they do carry along uncertainties.
2. **Activity and location classification:** The corrected and annotated data is automatically classified as *traveling* or *stationary* using web services to provide a first level of refinement to the model output for a given person on a given day. The data is also split into *trips* based on dwell time.
3. **Context estimation:** The corrected and classified location data is used as input to web-based information sources on weather, road conditions, and aggregated driver behaviors.
4. **Exposure and impact calculation:** Finally, the fine-grained, classified data and derived data is used as input to geospatial data sets and microenvironment models that are in turn used to provide an individual's personalized estimates.

While PEIR's focus is still on the results of this four-step process, we eventually found that users wanted to know more about how impact and exposure were estimated. So for each chunk of data we provide details of the process, such as what percentage of time was spent on a freeway and what the weather was like around where the user was traveling. We also include a detailed explanation for every provided metric. In this case, transparency in the estimation process allows users to see how their actions have an effect on impact and exposure rather than just knowing how much or how little they are polluting their neighborhood. There is, of course, such a thing as information overload, so we are careful in how much (and how little) we show. We address much of these issues in the next section.

Data Visualization

Once data is collected, uploaded, and processed, users need to be able to access, evaluate, and explore their data. The main design goal behind YFD and PEIR was to make personal data understandable to nonprofessionals. Data has to be presented in a way that is relatable; it has to be humanized. Oftentimes we get caught up in statistical charts and graphs, which are extremely useful, but at the same time we want to engage users so that they stay interested, continue collecting data, and keep coming back to the site to gauge their progress in whatever they are tracking. Users should understand that the data is about them and reflect the choices they make in their daily lives.

I like to think of data visualization as a story. The main character is the user, and we can go two ways. A story of charts and graphs might read a lot like a textbook; however, a story with context, relationships, interactions, patterns, and explanations reads like a novel. This is not to say that one or the other is better. There are plenty of interesting textbooks, and probably just as many—if not more—boring novels. We want something in between the textbook and novel when we visualize personal data. We want to present the facts, but we also want to provide context, like the who, what, when, where, and why of the numbers. We are after emotion. Data often can be sterile, but only if we present it that way.

Copyrighted material

PEIR

In the case of PEIR, we were met with the challenge of presenting scientific data—carbon impact, exposure to high levels of particulate matter, and impact to sensitive sites such as hospitals and schools. Impact and exposure are not a part of everyday conversation. Most people do not know whether 1,000 kilograms of carbon emissions in a day is a lot or a little. Is one hour of exposure to high levels of particulate matter normal? These types of questions factor into PEIR's visualization design. It is important to remember, however, that even though the resulting data is not immediately understandable, it is all derived from location data, which is extremely intuitive. There are perhaps few types of data that are so immediately understandable as one's place in physical space. Therefore, we use maps as the visualization anchor point and work from there.

Mapping location-based data

Location-based data drives the PEIR system, so an interactive map is the core of the user interface. We initially used the Google Maps API, but quickly nixed it in the interest of flexibility. Instead, we use Modest Maps. It is a display and interaction library for tile-based maps in Flash and implemented in ActionScript 3.0. Modest Maps provides a core set of features, such as panning and zooming, but allows designers and developers to easily customize displays. Modest Maps implementations can easily switch map tiles, whether the choice is to use Microsoft's map tiles, Google's, custom-built ones, or all of the above. We are free to adjust color, layout, and overall style, which lend themselves to good design practice and useful visualization, and the flexibility allows us to incorporate our own visualizations on the map or as a supplement. In the end, we do not want to limit ourselves to just maps, and Modest Maps provides the flexibility we need to do this.

Experimenting with visual cues

We experimented with a number of different ways to represent PEIR data before deciding on the final mapping scheme. During the design process, we considered several parameters:

- How can users interact with a lot of traces at once without cluttering the map?
- How can we represent both stationary (user is idle) and traveling (user is moving) data chunks at the same time?
- How do we display values from all four microenvironment models?
- What colors should we use to represent GPS trace, impact, and exposure?
- How do we shift focus toward the actual data and away from the underlying map tiles?

Mapping multivariate location traces

In the early stages of the design process, we mapped GPS traces the way that users typically see location tracks: simply a line that goes from point to point. This was before taking values from the microenvironment models into account, so the map was a basic implementation

Copyrighted material

using Modest Maps and tiles from OpenStreetMap. GPS traces were mono-colored and represented nothing but location; there was a circle at the end so that the user would know where the trip began and ended.

This worked to a certain extent, but we soon had to visualize more data, so we changed the format. We colored traces based on impact and exposure values. The color scheme used five shades of red. Higher levels of, say, carbon impact were darker shades of red. Similarly, trips that had lower carbon impact were lighter shades of red.

The running metaphor is that the more impact the user has on the environment, the more the trip should stand out on the map. The problem with this implementation was that the traces on the map did not stand out (Figure 1-1). We tried using brighter colors, but the brightly colored trips clashed with the existing colors on the map. Although we want traces to stand out, we do not want to strain the user's eyes. To solve this problem we tried a different mapping scheme that again made all trips on the map mono-color, but used circles to encode impact and exposure. All traces were colored white, and the model values were visually represented with circles that varied in size at the end of each trip. Greater values were displayed as circles larger in area while lesser values were smaller in area. This design scheme was short-lived.

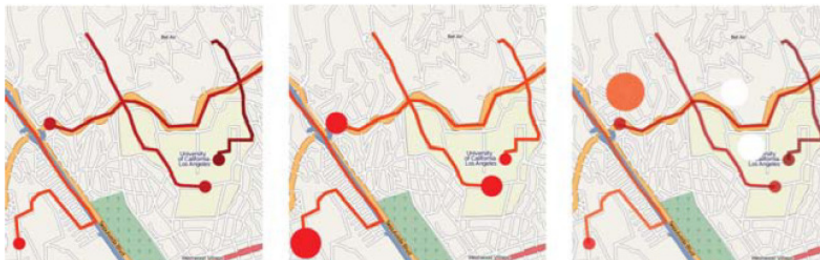


FIGURE 1-1. We experimented with different visual cues on a map to best display location data with impact and exposure values. The above shows three iterations during our preliminary design. The left map shows GPS traces color-coded by carbon impact; in the center map, we encoded impact with uni-color area circles; on the right, we incorporated GPS data showing when the user was idle and went back to using color-coding. (See Color Plate 1.)

One problem with representing values only at the end of a trace was that users thought the circles indicated that something happened at the very end of each trip. However, this is not the case. The map should show that something is happening during the entirety of a trip. Carbon is emitted everywhere you travel, not collected and then released at a destination.

We switched back to color-coding trips and removed the scaled area circles representing our models' values. At this point in the design process, we now had two types of GPS data: traveling and stationary. Traveling trips meant that the user was moving, whether on foot or in a vehicle; stationary chunks are times when the user is not moving. She might be sitting at a desk or stuck in traffic. To display stationary chunks, we did not completely abandon the idea of using area circles on the map. Larger circles mean longer duration, and smaller circles mean shorter duration. Similar to traveling trips, which are represented by

Copyrighted material

lines, area circles are color-coded appropriately. For example, if the user chooses to color-code by particulate matter exposure, a stationary chunk that was spent idle on the freeway is shown as a brightly colored circle.

However, we are again faced with same problem as before: trying to make traces stand out on the map without clashing with the map's existing colors. We already tried different color schemes for the traces, but had not yet tried changing the shades of the actual map. Inspired by Trulia Snapshot, which maps real estate properties, we grayscaled map tiles and inverted the color filters so that map items that were originally lightly colored turned dark and vice versa. To be more specific, the terrain was originally lightly colored, so now it is dark gray, and roads that were originally dark are now light gray. This darkened map lets lightly colored traces stand out, and because the map is grayscale, there is less clashing (Figure 1-2). Users do not have to try hard to distinguish their data from roads and terrain. Modest Maps provided this flexibility.

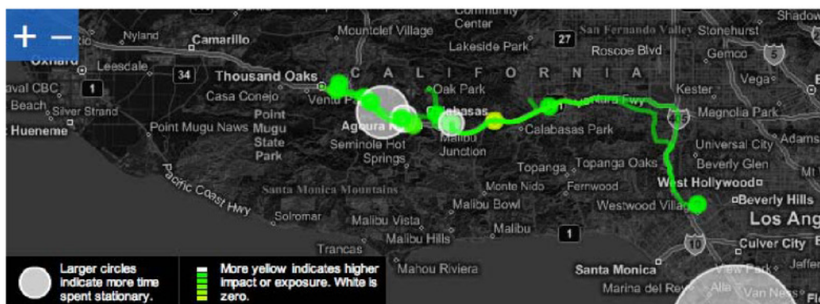


FIGURE 1-2. In the current mapping scheme, we use color filters to highlight the data. The map serves solely as context. Linked histograms show impact and exposure distributions of mapped data. When the user scrolls over a histogram bar, the corresponding GPS data is highlighted on the map. (See Color Plate 2.)

Choosing a color scheme

Once we established map tiles as the dark background and represented trips in the light foreground, we decided what colors to use. This is important because users recognize some colors as specific types of events. For example, red often means to stop or that there is danger ahead, whereas green means progress or growth, especially from an environmental standpoint.

It is also important to not use too many contrasting colors. Using dissimilar colors without any progression indicates categorical data. Model values, however, are on a continuous scale. Therefore, we use colors with a subtle gradient. In the earlier versions we tried a color scale that contained different shades of green. Users commented that because green usually means good or environmentally friendly, it was strange to see high levels of impact and exposure encoded with that color. Instead, we still use shades of green but also incorporate yellows. From low to high values, we incrementally shift from green to yellow, respectively. Trips that have impact or exposure values of zero are white.

Copyrighted material

Making trips interactive

Users can potentially map hundreds of trips at one time, providing an overview of traveling habits, impact, and exposure, but the user also needs to read individual trip details. Mapping a trip is not enough. Users have to be able to interact with trips so that they know the context of their travels.

When the user scrolls over a trip on the PEIR map, that trip is highlighted, while all other trips are made less prominent and blend in with the background without completely disappearing. To be more specific, transparency of the trip of interest is decreased while the other trips are blurred by a factor of five. Cabspotting, a visualization that maps cab activities in San Francisco, inspired this effect. When the user clicks on a trip on the map, the trip log automatically scrolls to the trip of interest. Again, the goal is to provide users with as much context as possible without confusing them or cluttering the screen.

These features, of course, handle multiple trips only to a certain extent. For example, if there are hundreds of long trips in a condensed area, they can be difficult to navigate due to clutter. This is an area we plan to improve as we incorporate user-contributed metadata such as tags and classification.

Displaying distributions

PEIR provides histograms on the right side of the map to show distributions of impact and exposure for selected trips. There are four histograms, one for each microenvironment model. The histograms automatically update whenever the user selects a trip from the trip log. If trips are mostly high in impact or exposure, the histograms are skewed to the right; similarly, if trips are mostly low in impact or exposure, the histograms are skewed to the left.

We originally thought the histograms would be useful since they are so widely used in statistics, but that proved not to be the case. The histograms actually confused more than they provided insight. Although a small portion of the test group thought they were useful, most expected the horizontal axis to be time and the vertical axis to be the amount of impact or exposure. People seemed more interested in patterns over time than overall distributions. Therefore, we switched to time-based bar charts (Figure 1-3). Users are able to see their impact and exposure over time and browse by week.

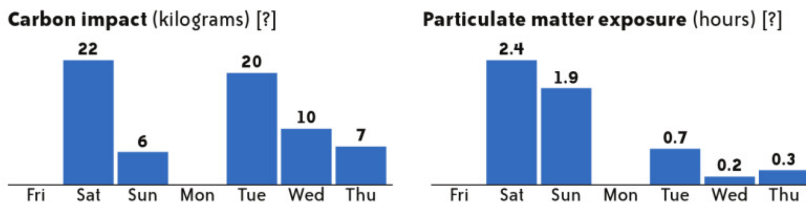


FIGURE 1-3. Time series bar charts proved to be more effective than value-based histograms.

Copyrighted material

Sharing personal data

PEIR lets users share their impact and exposure with Facebook friends as another way to compare values. It is through sharing that we get around the absolute scale interpretation of axes and shift emphasis onto relative numbers, which better helps users make inferences. Although 1,000 kilograms of carbon might seem like a lot, a comparison against other users could change that misconception. Our Facebook application shows aggregated values in users' Facebook profiles compared against other Facebook friends who have installed the PEIR Facebook application (Figure 1-4).

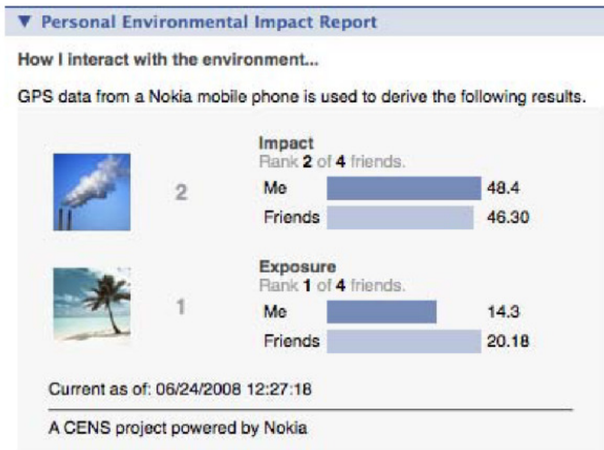


FIGURE 1-4. PEIR's Facebook application lets users share their impact and exposure findings as well as compare their values with friends. (See Color Plate 3.)

The PEIR Facebook application shows bar graphs for the user's impact and exposure and the average of impact and exposure for his or her friends. The application also shows overall rank. Those who have less impact or exposure are higher in rank. Icons also provide more context. If impact is high, an icon with a chimney spouting a lot of smoke appears. If impact is low, a beach with clear skies appears.

Shifting attention back to the PEIR interface, users also have a network page in addition to their personal profile. The network page again shows rankings for the last week of impact and exposure, but also shows how the user's friends rank. The goal is for users to try to climb in the rankings for least impact and exposure while at the same time encouraging their friends to try to improve. Although actual values in units of kilograms or hours for impact or exposure might be unclear at first, rankings are immediately useful. When users pursue higher ranking, values from PEIR microenvironment models mean more in the same way that a score starts to mean something while playing a video game.

The reader should take notice that no GPS data is shared. We take data privacy very seriously and make many efforts to keep certain data private, which is why only impact and exposure aggregates are shown in the network pages.

Copyrighted material

YFD

Whereas PEIR deals with data that is not immediately relatable, YFD is on the opposite side of the spectrum. YFD helps users track data that is a part of everyday conversation. Like PEIR, though, YFD aims to make the little things in our lives more visible. It is the aggregate of small choices that have a great effect. The visualization had to show this.

To begin, we go back to one of the challenges mentioned earlier. We want users to tweet frequently and work personal data collection into their daily Twitter routine. What are the motivations behind data collection? Why does a user track what he eats or his sleep habits? Maybe someone wants to lose weight so that he feels more confident around the opposite sex, or he wants to get more sleep so that he does not fall asleep at his desk. Another user, however, might want to gain weight, because she lost weight when she was sick, or maybe she sleeps too much and always feels groggy when she gets up. Others just might be curious. Whatever the motivation, it is clear that everyone has his or her own reasons for personal data collection. YFD highlights that motivation as a reminder to the user, because no matter what diet system someone is on or sleep program he is trying, people will not change unless they really want to. Notice the personal words of motivation in large print in the middle of the screen in Figure 1-5.

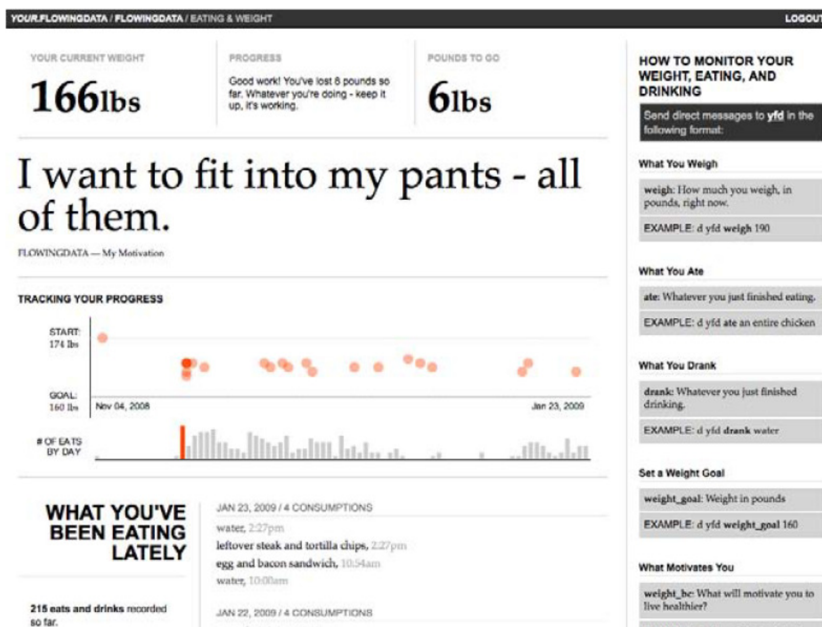


FIGURE 1-5. People track their weight and what they eat for different reasons. YFD places motivation front and center. (See Color Plate 4.)

It is also worth noting that each tracker's page shows what has happened most recently at the top. This serves a few purposes. First, it will update whenever the user tweets a data point, so that the user can see his status whenever he logs in to YFD. Second, we do not want to stray too far from the feel of Twitter, again to reinforce working YFD tweets into

Copyrighted material

the Twitter routine. Finally, the design choice largely came out of the experience with PEIR. Users seem to expect time-based visualization, so most YFD visualization is just that.

There is one exception, though—the feelings and emotions tracker (Figure 1-6). As anyone can tell you, emotions are incredibly complicated. How do you quantify happiness or sadness or nervousness? It did not seem right to break emotions down into graphs and numbers, so a sorted tag cloud is used instead. It somehow feels more organic. Emotions of higher frequency are larger than those that occur rarely. The YFD trackers are all modular at these early stages of development, but I do plan to eventually integrate all trackers as if YFD were a dashboard into a user's life. The feelings tracker will be in the center of it all. In the end, everything we do is driven by how we feel or how we want to feel.

FIGURE 1-6. Users can also keep track of how they feel. Unlike the other YFD trackers, the page of emotions does not have any charts or graphs. A word cloud was chosen to provide more organic-feeling visualization.

The Point

Data visualization is often all about analytics and technical results, but it does not have to be—especially with personal data collection. People who collect data about themselves are not necessarily after the actual data. They are mostly interested in the resulting information and how they can use their own data to improve themselves. For that to come through, people have to see more than just data in the visualization. They have to see themselves. Life is complex, data represents life, and users want to understand that complexity somehow. That does not mean we should dumb down the data or the information. Instead, we use the data visualization to teach and to draw interest. Once there is that interest, we can provide users with a way to dig deeper and explore their data, or more accurately, explore and understand their lives in that data. It is up to the statistician, computer scientist, and designer to tell the stories properly.

Copyrighted material

How to Participate

PEIR and YFD are currently by invitation only, but if you would like to participate, please feel free to visit our sites at *http://peir.cens.ucla.edu* or *http://your.flowingdata.com*, respectively. Also, if you are interested in collaborating with the PEIR research group to incorporate new models, strategies, or visualization, or if you have ideas on how to improve YFD, we would love to hear from you.

Copyrighted material

Copyrighted material

CHAPTER TWO

The Beautiful People: Keeping Users in Mind When Designing Data Collection Methods

Jonathan Follett and Matthew Holm

Introduction: User Empathy Is the New Black

ALWAYS KEEP THE WANTS AND NEEDS OF YOUR AUDIENCE IN MIND. THIS PRINCIPLE, WHICH GUIDES THE FIELD known as user experience (UX) design, seems painfully obvious—enough to elicit a roll of the eyes from any professional creating new, innovative digital technologies or improving upon already existing systems. "Yes! Of course there's a person using the product!"

But, while the benefits of following a user-centered design process can be great—like increased product usability and customer satisfaction, and reduced 800-number service calls—this deceptively simple advice is not always followed, especially when it comes to collecting data.

What Is UX?

UX is an emerging, multidisciplinary field focused on designing products and services that people can easily understand and use. Its primary concern is making systems adapt to and serve the user, rather than the other way around. (See Figure 2-1.) UX professionals can include practitioners and researchers in visual design, interaction design, information architecture, user interface design, and usability. And the field, which is strongly related to human factors and computer-human interaction, draws upon ethnography and psychology as well: UX professionals operate as user advocates. Generally, UX design techniques

Copyrighted material

are applied to desktop and web-distributed software, although proponents may use the term more broadly to describe the design of any complex experience—such as that of a museum exhibit or retail store visit.

The Benefits of Applying UX Best Practices to Data Collection

When it comes to data collection, user experience design is more important than ever. Data—that most valuable digital resource—comes from people and their actions, so designers and developers need to be constantly thinking about those people, and not just about the data they want to collect. The key method for collecting data from people online is, of course, through the use of the dreaded form. There is no artifact potentially more valuable to a business, or more boring and tedious to a participant.

As user experience practitioners, we regularly work with data collected from large audiences through the use of web forms. And we've seen, time and again, that the elegant visual design of forms can assist greatly in the collection of data from people. The challenge presented by any form design project is that, although it's easy enough to collect data from people, it can be exceptionally difficult to collect good data. Form design matters (see Figure 2-1), and can directly affect the quality of the data that you receive: better-designed forms gather more accurate and more relevant data.

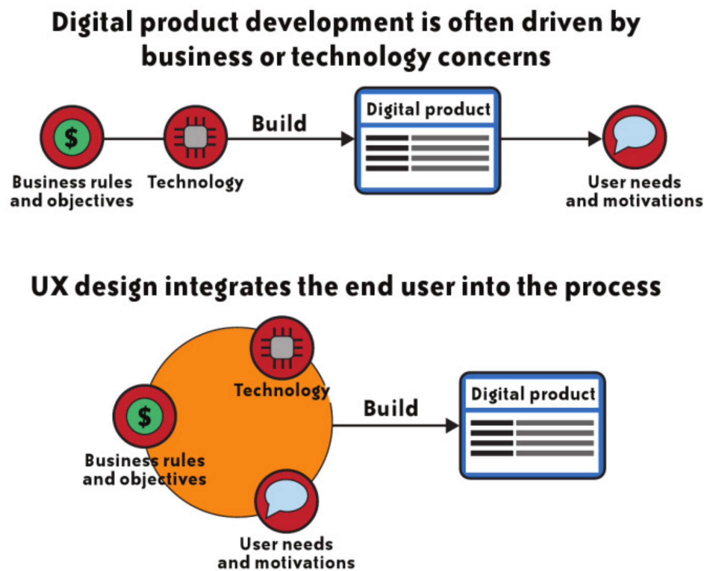


FIGURE 2-1. Rather than treating audience needs as an afterthought, the UX design process addresses audience needs, business requirements, and technical feasibility during the design stage.

So, what is it that drives people to fill in forms and create the data we need? And how can we, as designers and developers, encourage them to do it more efficiently, effectively, and accurately?

Copyrighted material

We'll take a look at a case study here, showing an example of simple form design using UX best practices and principles to increase the completion rate of unsolicited questionnaires.

The Project: Surveying Customers About a New Luxury Product

Our project was an online survey for a marketing consulting firm, Urban Wallace Associates, that was trying to gauge consumer interest in a new luxury product. (To maintain confidentiality, we've had to change some of the details throughout this chapter relating to the content of the survey questions.) The survey audience was the same demographic as the product's eventual retail audience: wealthy individuals between the ages of 55 and 75.

An email survey was not our client's first choice. Urban Wallace Associates had already attempted a telephone survey of the target group. "Normally, we get about 35% answering machines," says UWA President, Roger Urban. "In this group, we got more than 80% answering machines. When someone did pick up, it was usually the housekeeper!"

Unable to get a satisfactory sample of the target audience on the phone, our client turned to email. One of the reasons our client chose this communication method is because, for this affluent group, email is a near-universal utility. And while email faces its own set of gatekeepers—namely, automated junk mail filters—very few people, as of yet, hire others to read it for them. Even the wealthy still open their own emails.

Urban Wallace Associates secured an email marketing firm to help generate and prequalify the recipient list, and to deliver and track the outgoing messages. Our firm was brought in to design and build the survey landing page, which would open in the recipient's web browser when he clicked a link in the body of the email, and to collect the results into a database. Our primary focus in this task was maintaining an inviting atmosphere on the questionnaire web page, so that respondents would be more willing to complete the form. A secondary task was creating a simple interface for the client so that he could review live reporting results as the data came in.

Specific Challenges to Data Collection

Data collection poses specific challenges, including accessibility, trust, and user motivation. The following sections discuss how these issues affected our design.

Challenges of Accessibility

Advocates of web accessibility—designing so that pages and sites are still useful for people with special needs and disabilities—often say that designing a site that is accessible will also create a site that is more usable for everyone. This was not just a theoretical consideration in our case, since, with a target audience whose members were approaching or past retirement, age-related vision impairment was a real concern. Some 72% of Americans report vision impairment by the time they are 45 years of age.

Copyrighted material

The other side of the age issue—one rarely spoken of, for fears of appearing discriminatory—is that older people use computers and the Internet in fewer numbers and with less ease than younger people who grew up with computers in their lives. (Individuals with higher incomes generally use computers and the Internet more, however, so those age-related effects were mitigated in our sample group.) Respondents who are stymied by a confusingly designed survey are less likely to give accurate information—or, indeed, to complete the survey at all. In our case, as in all such projects, it pays to recall that essential adage: know your audience.

Challenges of Perception

While accessibility is a functional issue—a respondent cannot complete a survey if she can't read it—our project faced other challenges that were more emotional in nature, and depended on how the respondent perceived the questioner and the questions.

Building trust

Internet users are well aware that giving out information to people online can have serious consequences, ranging from increased spamming, phone solicitation, and junk mail all the way up to fraud and identity theft. Therefore, for those looking to do market research online, building trust is an important factor. Although the response to the product and our survey was ultimately quite positive overall (as we'll describe in more detail later on), there were several participants who, when asked why they were not interested in the product, responded with statements such as:

> "Don't trust your firm"
> "Unknown Offeror"
> "Don't believe what [the product] claims to deliver"
> "can't afford it...don't trust it...too good to be true so it probably isn't. PLEASE DO NOT CONTACT ME ABOUT THIS PRODUCT ANY MORE"

These responses illustrate the lengths to which we must go in order to build trust online. It was more important, in our case, because we were explicitly *not* selling anything—we were conducting research. "I don't want anything that sounds like a sales lead," our client, Roger Urban, told us at the outset. It would be necessary to provide clear links back to information about Urban Wallace Associates, so people could see what kind of firm was asking them questions, and to post clear verbiage that we were not collecting their personal data, and that we were not going to contact them again. The only wrinkle was that our client's research required knowing the U.S. state in which each respondent was living. So we would have to figure out a way to capture that information without violating the spirit of the trust we were trying to build.

Length of survey

Keeping the respondent from disengaging was one of our biggest concerns. The client and we agreed early on to keep the survey to a single screen. Multiple screens would not only require more patience from the respondent, but they might require additional action

Copyrighted material

(such as clicking a "go on to the next question" button). Any time a survey requires an action from the respondent, you're inviting him to decide that the extra effort is not worth it, and to give up. Further, we wanted to avoid intimidating the respondent at any point with the *perceived* length of the survey. Multiple screens, or the appearance of too many questions on a single screen, increase the likelihood that a respondent will bail out.

Accurate data collection

One particularly important problem we considered during the design stage of this survey was that the data we collected needed to be as accurate as possible—perhaps an obvious statement, but difficult nonetheless. Our form design had to elicit responses from the participants that were honest, and not influenced by, say, a subconscious desire to please the questioner (a common pitfall for research of this type). The difference between collecting opinion data and information that might be more administrative in nature, such as an address for shipping, is that shipping data can be easily validated, whereas opinion data, which is already subjective, has a way of being more slippery. And although the science of designing opinion polls and measuring the resulting data is not something we'll cover in depth in this chapter, we will discuss some of the language and other choices our team made to encourage accurate answers.

Motivation

Finally, although we've talked about concerns over how to make it *possible* for respondents to use the form, as well as the problems of getting them to trust us enough to keep participating, avoiding scaring them off with a lot of questions, and making sure we didn't subconsciously influence their answers, we haven't mentioned perhaps the most important part of any survey: why should the person want to participate at all? For this type of research survey, there is no profit motive to participate, unlike online forums such as Amazon's Mechanical Turk, in which users complete tasks in their spare time for a few dollars or cents per task. *But when there is no explicit profit to be made, how do you convince a person to take the time to answer your questions?*

Designing Our Solution

We've talked about some of the pitfalls inherent in a data-collecting project; in the next few sections, we discuss the nuts and bolts of our design, including typography, web browser compatibility, and dynamic form elements.

Design Philosophy

When we design to elicit a response, framing the problem from a user's perspective is critical. It's easy to get caught up in the technical constraints of a project and design for the computer, rather than the person using it. But form data is actively generated by a person (as opposed to being passively generated by a sensor or other input), and requires the participant to make decisions about how and whether to answer your questions. So, the way in which we collect a participant's data matters a great deal.

Copyrighted material

As we designed the web form for this project, we focused on balancing the motivations of survey participants with the business objectives of the client. The client's primary business goal—to gather data determining whether the target audience would be interested in purchasing a new luxury product—was in line with a user-centered design perspective. By placing the person in the central role of being both advisor and potential future customer, the business objectives provided strong justification for our user-centered design decisions.

Here are a couple of guidelines we used to frame our design decisions:

Respect the user
: Making our design people-centered throughout the process required thinking about our users' emotional responses. In order to convince them to participate, we had to first show them respect. They're not idiots; they're our potential customers. We all know this instinctively, but it's surprising how easily we can forget the principle. If we approach our users with respect, we'll naturally want the digital product we build for them to be accessible, usable, and easily understood. This perspective influences the choices we make for everything from language to layout to technology.

Make the person real
: In projects with rapid timelines or constrained budgets, we don't always have the resources to sculpt a complete user profile or persona based on target market research, or to observe users in their work environments. In these situations, a simple "guerilla" UX technique to create empathy for the user and guide design decisions is to think of a real person we know in the demographic, whom we'd legitimately like to help. We had several such stand-in personas to guide our thinking, including our aging parents and some former business mentors whom we know very well. Of course, imagining these people using our digital product is only a first step. Since we knew them well, we were also able to enlist some of them to help in preliminary testing of our design.

In the end, people will adapt their own behavior to work with just about any design, if they have to. The purpose of UX is to optimize those designs so people will want to use a product or service, and can use it more readily and easily, without having to adapt their behavior.

Designing the Form Layout

Generally, no matter how beautiful our form design, it's unlikely that it will ever rise to the level of delighting users. There is no designers' holy grail that can make people enthusiastic about filling out a form. However, form aesthetics do matter: clear information and visual design can mitigate users' boredom by clearly guiding their eyes and encouraging them to make it to the end, rather than abandoning the task halfway through. Good form design doesn't draw attention to itself and should be nearly invisible, always honoring its primary purpose, which is to collect accurate information from people. While form design needs to be both pleasing and professional in tone, in most cases, proper visual treatment will seem reserved and utilitarian in comparison to most other kinds of web pages. Form visual design can only be judged by how effectively it enables users to complete the task. For this project, the areas where we focused our design efforts were in the typography, page layout, and interaction design.

Copyrighted material

Web form typography and accessibility

In general, older readers have difficulty seeing small type. And survey participants are not so generous that they're willing to strain their eyes to read a form. Because the target audience for our survey project was older (55–75 years of age), we knew that overall legibility would be an issue.

We chose the sans serif typeface Arial (a close cousin via Microsoft of the modern workhorse Helvetica), which is standard-issue on nearly 100% of web browsers, and we set headers and body copy large at 20 pixels and 14 pixels, respectively. Although larger type caused the page to be slightly longer, the improvements in legibility were well worth it. Line spacing was not too tight, and was left-justified with a rag right. Line length was roughly 85 characters. And we set the majority of the text with the high contrast combination of black type on a white background, also for legibility considerations. While we did use color strategically to brighten the page and emphasize the main headers, we did not rely on it to provide any additional information to the user. We did this because, for the male audience, roughly 7–8% has some type of color blindness.

Giving them some space

A densely designed form with no breathing room is guaranteed to intimidate the user. So, leaving some open whitespace in a layout is key.

In our survey, the first section included a text description of the luxury product, which we asked participants to read and evaluate. Web readers are notorious for their short attention spans and tendency to skim text rather than read it all the way through. So, following web writing best practices, we separated the 250-word product description into subsections with headers, pulling out key bullet points and dividing it into easily digestible chunks (see Figure 2-2).

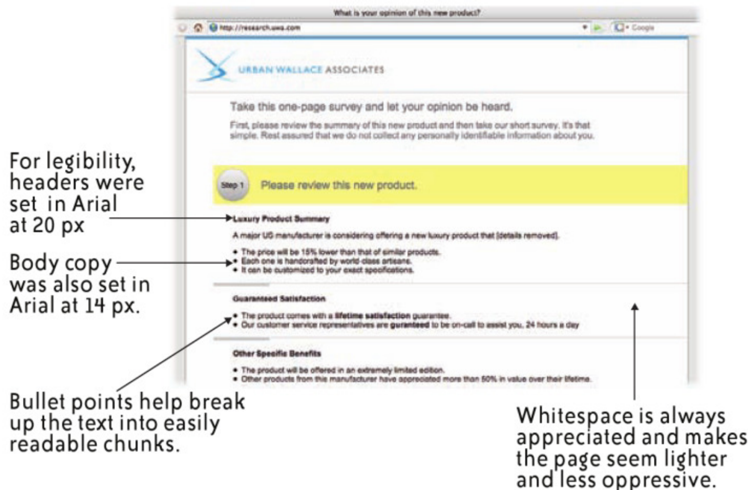


FIGURE 2-2. Designing for legibility. (See Color Plate 5.)

Copyrighted material

Accommodating different browsers and testing for compatibility

To make sure the form was usable by our audience, we designed the form page so it could be viewed easily in a variety of screen sizes, from an 800-pixel width on up. To accomplish this, we centered the form in the browser, using a neutral gray background on the right and left margins to fill the remaining space of widescreen monitors and ensure that the form wouldn't appear to be disembodied and floating. We also tested in all major web browsers, including the legacy IE6, to ensure that the dynamic form looked good and functioned well.

Interaction design considerations: Dynamic form length

Dynamic forms can "soften the blow" of having many questions to answer. Using JavaScript or other methods can create a soft reveal that allows the form to be subtly altered—or lengthened—based on user input (see Figures 2-3 and 2-4). These techniques allowed us to balance not scaring users off with a form that is too long on the one hand, and not infuriating them because they had been "deceived" about the form length on the other.

FIGURE 2-3. The survey starts with only three questions. (See Color Plate 6.)

Copyrighted material

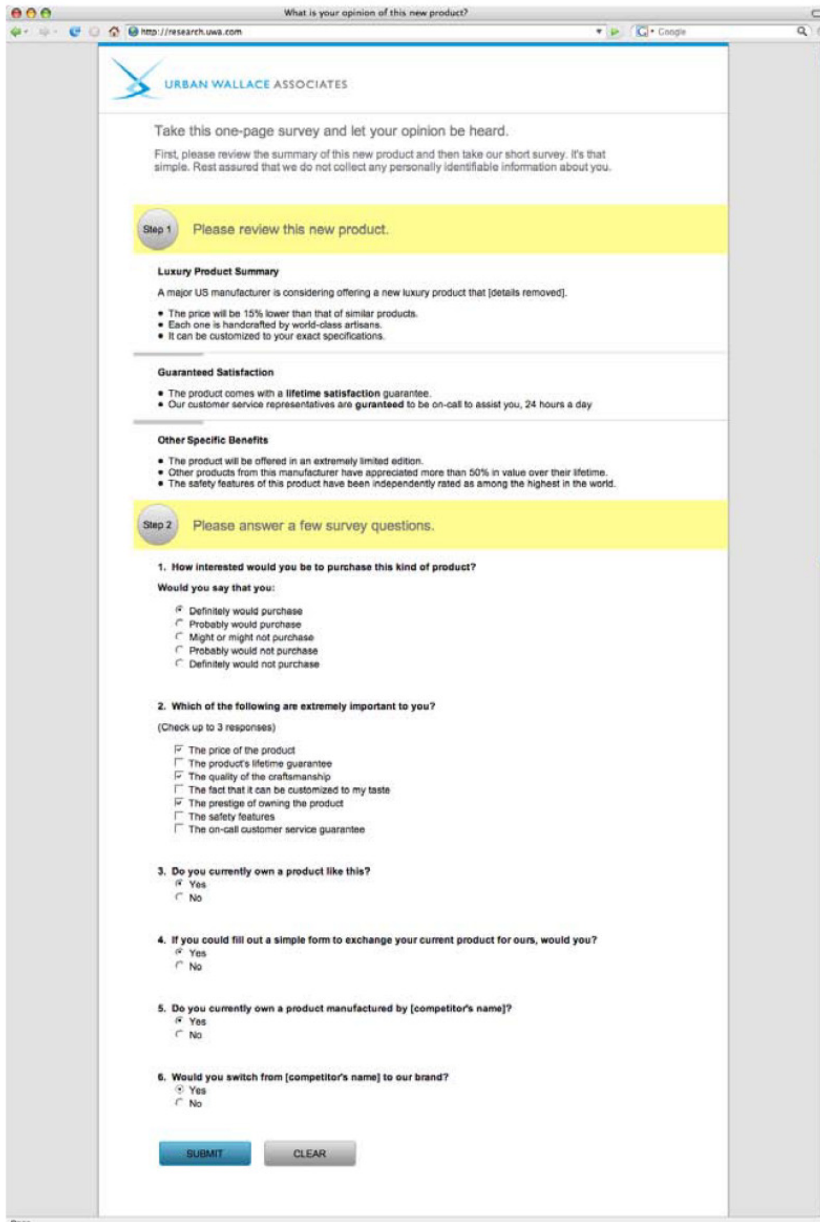


FIGURE 2-4. The survey may expand to up to six questions depending on user input. (See Color Plate 7.)

For our project, the readers, in effect, built the survey as they answered each question. We used a very simple piece of JavaScript code to make sure each new question was conditional upon an answer to previous questions. The idea for this solution came from another website we were working on at the time. In that project—a portfolio site for a designer—we used JavaScript to hide and reveal details about different projects, making it possible to take in all of the designer's work at a glance and then dive deeper into areas of interest, all

Copyrighted material

without leaving the home page. This idea—not overwhelming the user with too much information, yet making that information quickly accessible at the same time—was on our minds when we approached the survey design. Here is the code we used:

```
<script language="JavaScript">
//This finds the word "Yes" in an input value and displays the designated hiddenElement
(or hides it if "Yes" is not found)
function switchem(switchElement,hiddenElement) {
if (switchElement.value.search("Yes") > -1)
    document.getElementById(hiddenElement).style.display = '';
else
    document.getElementById(hiddenElement).style.display = 'none';
}
</script>

<script language="JavaScript">
//This finds the word "No" in an input value and displays the designated hiddenElement
(or hides it if "No" is not found)
function switchem2(switchElement,hiddenElement) {
if (switchElement.value.search("No") > -1)
    document.getElementById(hiddenElement).style.display = '';
else
    document.getElementById(hiddenElement).style.display = 'none';
}
</script>

...

<li id="survey1" class="surveynum">How interested would you be to purchase this kind of
product?
    <p><b>Would you say that you:</b></p>
    <ul class="nobullet">
        <li><input
onclick="switchem(this,'survey2');switchem2(this,'survey3');document.
getElementById('surveytextarea').value=''" type="radio" name="q1" value="Yes,
Definitely would purchase"> Definitely would purchase</li>
        <li><input
onclick="switchem(this,'survey2');switchem2(this,'survey3');document.
getElementById('surveytextarea').value=''" type="radio" name="q1" value="Yes, Probably
would purchase"> Probably would purchase</li>
        <li><input
onclick="switchem(this,'survey2');switchem2(this,'survey3');document.
getElementById('surveytextarea').value=''" type="radio" name="q1" value="Yes, Might or
might not purchase"> Might or might not purchase</li>
        <li><input
onclick="switchem(this,'survey2');switchem2(this,'survey3');document.
getElementById('q2a').checked=false;document.getElementById('q2b').
checked=false;document.getElementById('q2c').checked=false;document.
getElementById('q2d').checked=false;document.getElementById('q2e').
checked=false;;document.getElementById('q2f').checked=false;
document.getElementById('q2g').checked=false" type="radio" name="q1" value="No,
Probably would not purchase"> Probably would not purchase</li>
```


Copyrighted material

```
        <li><input
onclick="switchem(this,'survey2');switchem2(this,'survey3');document.
getElementById('q2a').checked=false;document.getElementById('q2b').
checked=false;document.getElementById('q2c').checked=false;document.
getElementById('q2d').checked=false;document.getElementById('q2e').
checked=false;;document.getElementById('q2f').checked=false;
document.getElementById('q2g').checked=false" type="radio" name="q1" value="No,
Definitely would not purchase"> Definitely would not purchase</li>
    </ul>
</li>
<li id="survey2" style="display:none" class="surveynum">Which of the following are
extremely important to you?
    <p>(Check up to 3 responses)</p>
    <ul class="nobullet">
       <li><input type="checkbox" name="q2" id="q2a" value="The price of the product">
The price of the product </li>
        <li><input type="checkbox" name="q2" id="q2b" value="The product's
lifetime guarantee"> The product's lifetime guarantee</li>
        <li><input type="checkbox" name="q2" id="q2c" value="The quality of
the craftsmanship"> The quality of the craftsmanship </li>
        <li><input type="checkbox" name="q2" id="q2d" value="The fact that it can be
customized to my taste"> The fact that it can be customized to my taste </li>
        <li><input type="checkbox" name="q2" id="q2e" value="The prestige of
owning the product"> The prestige of owning the product </li>
        <li><input type="checkbox" name="q2" id="q2f" value="The safety features"> The
safety features </li>
        <li><input type="checkbox" name="q2" id="q2g" value="The on-call customer
service guarantee"> The on-call customer service guarantee</li>
    </ul>
</li>
<li id="survey3" style="display:none" class="surveynum">Why are you not interested in
this product?
    <ul class="nobullet">
        <li><textarea id="surveytextarea" name="q3"></textarea></li>
    </ul>
</li>
```

The result is that selecting any of the three positive responses on the 5-point scale in Question 1 revealed a checklist that helped further identify what the respondent liked about the product (Figure 2-5). Selecting either of the two negative responses revealed a text area in which the respondent could explain, precisely, what he disliked about the product (Figure 2-6).

As programming goes, this is child's play and hardly worth mentioning. But the impact from the user's standpoint is subtle and powerful. It meant that we could "listen" and "respond" to the user's input in a very conversational manner. It also meant that the psychological impact of the form length is much lower, as users are facing only a three-question survey at the start. The survey potentially could expand to six questions, but all of this happens without the user ever leaving the survey landing page, and without forcing the user to actively click some sort of "Next page" button.

Copyrighted material

Step 2 Please answer a few survey questions.

1. How interested would you be to purchase this kind of product?

Would you say that you:

- ○ Definitely would purchase
- ◉ Probably would purchase
- ○ Might or might not purchase
- ○ Probably would not purchase
- ○ Definitely would not purchase

2. Which of the following are extremely important to you?

(Check up to 3 responses)

- ☐ The price of the product
- ☐ The product's lifetime guarantee
- ☐ The quality of the craftsmanship
- ☐ The fact that it can be customized to my taste
- ☐ The prestige of owning the product
- ☐ The safety features
- ☐ The on-call customer service guarantee

FIGURE 2-5. Detail of survey when the user answers "Yes" to Question 1. (See Color Plate 8.)

Step 2 Please answer a few survey questions.

1. How interested would you be to purchase this kind of product?

Would you say that you:

- ○ Definitely would purchase
- ○ Probably would purchase
- ○ Might or might not purchase
- ○ Probably would not purchase
- ◉ Definitely would not purchase

2. Why are you not interested in this product?

FIGURE 2-6. Detail of survey when the user answers "No" to Question 1. (See Color Plate 9.)

Designing trust

We did some concrete things to try to establish trust with the respondents and indicate that this was a legitimate survey, not a phishing expedition. First, we prominently displayed the client's company logo at the top of the web survey page. The logo itself linked back to the "About Us" area on Urban Wallace Associates' main website, so survey participants could see who they were communicating with. Additionally, we hosted the survey page on a subdomain of our client's main site, not on some third-party host.

As previously mentioned, our client's research needed the U.S. state of residence of each respondent. But, since we told respondents, "we do not collect any personally identifiable information about you," it would have been awkward to then start asking questions about where the person lived. Our solution was to record the visitor's IP address automatically, which would satisfy the U.S. state location requirement but not violate the respondent's privacy. After all, a user's IP data is logged anytime he or she visits *any* website, and, at

Copyrighted material

most, it can only be used to determine the city of that otherwise anonymous user's Internet Service Provider.

We then purchased an inexpensive data set of IP-to-State information. With it, we were able to match each IP address collected with the U.S. state in which it resided. Although we could have scripted our pages to access this database and match the numbers at the time of data collection, we chose to do the matching semi-automatically after the fact. For starters, the project budget and timeframe did not warrant purchasing the additional server power to handle the task. But more important, from a user perspective, was the delay this matching would have inevitably built into the survey completion process. Although it might have been more convenient for us to receive finalized data at once, it would have created an additional inconvenience for our user. When designing a data collection experience, it's important to think about what server tasks must take place during the survey in order for the user's needs to be met, and what tasks can be delayed until after data collection. Don't ask the user to do what you can do—or discover—on your own.

All of this leads us back to the central point of this chapter, which is also the final, and core, aspect of building trust: treat the respondent with respect. By demonstrating that you value the respondent and her time and intelligence, by interacting with her in a conversational manner (despite the fact that all survey questions are being delivered by a preprogrammed machine), and showing her that you've been "listening" to her answers (don't, for example, ask slight variations of the same question over and over again, which makes it seem as though you didn't pay attention to her original response), you'll increase trust, encourage real answers, and keep the respondent from disengaging.

Designing for accurate data collection

This sort of talk can seem a little touchy-feely at times, especially to people who only work with the hard numbers retrieved from data collection, and not the human beings who generated that data. But all of this user-centered focus is not just a matter of politeness—it's also crucial for the reliability of the data that we actually get. "For a survey like this," says Roger Urban, whose firm specializes in measuring market interest and customer satisfaction through face-to-face, mailed, telephone, and email surveys just like this, "you're dealing with extremely thin data sets, so the quality of that data is *really* important." In other words, when important decisions are being based on the answers given by only a few hundred people, those answers had better be *great*.

But great answers do not mean *positive* answers. After all, this is research, and just like scientists, we want to measure reality (Do customers care about price that much when it comes to this product? Is safety really their top concern, or not? *Are* they, in fact, happy with our service?), to see where our assumptions are wrong. "Techniques of persuasion are a disaster when it comes to research," says Roger Urban. People will, subconsciously, try to please researchers by answering in the way that they feel they are *supposed* to answer. Introducing persuasive techniques, whether implicit or explicit, will skew your research data. "If you want an artificially high positive," says Urban, "I can get it for you every time." But if you're making real business or policy decisions, what good is such data?

Copyrighted material

Motivation

You can't use persuasive techniques during the act of data collection, but you *do* need to persuade your respondents to participate in the first place. With no money involved, what is their motivation?

"There should always be some benefit," says Roger Urban, "even if that benefit is just 'voicing your opinion.'" Human beings are interesting creatures; where cold, hard cash may not be able to compel us, far more nebulous benefits may do the trick—for example, some well-placed flattery. We all like to be thought of as experts; validation that our opinion *is* important may be enough to convince us to spend time talking to a stranger. So, too, can the allure that we may be receiving "inside information" by participating, that we are glimpsing what the future holds. For example, what techie wouldn't be interested in participating in a survey that allowed us to glimpse the design of Apple's next i-gadget?

In our project's case, we knew that we were dealing with an older audience. The language in the initial email was important in terms of engaging the recipient, and our team went with an appeal to the respondent's expertise. In our first mailing, we tested two different headlines on equal-sized groups of recipients:

> "You can shape the face of [product information removed] for future generations."
> "We're seeking the voice of experience."

As it turned out, the first headline, though offering the respondent the power to steer the very direction of the future, apparently proved slightly more ephemeral and altruistic (after all, it implies that the benefit may be solely for future generations, not necessarily for the respondent) than the ego-stroking one that turns their age into a positive ("experience"). For the first headline, 12.90% of those who opened the email clicked through, and 16.22% of those completed the survey. For the second headline, 14.04% of those who opened the email clicked through, with 29.5% of those people completing the survey. When the second mailing was conducted two weeks later, with the "voice of experience" headline on all messages, it generated a click-through rate of 27.68% and a completion rate of 33.16%. This second email went to people on the list who did not open the first mailing. (One of the secrets of email surveys is that the second mailing to the same list generally receives just as many responses as the first.)

This is another aspect of UX philosophy that's worth remembering: test everything. In this case, test even your testing methods! When you have the time and resources, test different copy, test different layouts, and test different types of interaction design—all with actual users.

Reporting the live data results

In our project, one special consideration was that the recipient of the final data, the client, would also be a user of the system—with drastically different needs from those of the survey participants.

Copyrighted material

Because the project was time sensitive, the client needed to see the survey results quickly to determine whether the product was generally well received. For this use case scenario, our solution was an HTML page, accessible to the client, which displayed the data, crudely sorted with minimal formatting. The live, raw survey results were sorted first by mailing (two mailings of each headline were sent to two age segments—55 to 64 and 65 to 75) and then by people's Yes/No answers to the first question about their interest in the product.

Unlike the survey participants, who needed to be convinced to participate and encouraged to complete the form, the client was motivated by a desire to see the data as quickly as it was generated. For the client, speed and immediate access to the live results as they came in were more important than any other factors. Thus, his user experience reflected those priorities (see Figure 2-7).

#	ID	Mailing	State	Q1	Q2	Q3	Q4	Q5	Q6	Q7	IP Address
1	44	1	GA	Yes, Probably would purchase	[illegible]		Yes	Yes	No		24.110.208.16
2	54	1	VA	Yes, Probably would purchase	[illegible]		Yes	Yes	Yes	Yes	24.210.16.161
3	84	1	VA	Yes, Probably would purchase	[illegible]		No		No		64.12.116.13
4	104	1	FL	Yes, Probably would purchase	[illegible]		No		Yes	No	71.53.249.200
5	25	1	CA	Yes, Probably would purchase	[illegible]		Yes	Yes	No		69.3.100.166
6	103	1	CA	Yes, Probably would purchase	[illegible]		No		Yes	Yes	68.122.97.18
7	59	1	VA	Yes, Might or might not purchase	[illegible]		No		No		72.189.200.203
8	94	1	CO	Yes, Might or might not purchase	[illegible]		Yes	No	No		75.107.23.70
9	26	1	TX	Yes, Might or might not purchase	[illegible]		No		No		76.203.6.205
10	34	1	CA	Yes, Might or might not purchase	[illegible]		No		Yes	Yes	76.83.240.29

FIGURE 2-7. On this live data reporting screen, the client was able to see the survey results as they came in.

The raw data display was not, of course, the final deliverable. Upon the project conclusion we presented the client with fully sortable Excel spreadsheets of all the data we had collected (from eight total mailings, sent in two batches), including the U.S. state data that had not yet been generated at the time of the survey.

Results and Reflection

In the end, was all of this effort worth it? It's just a web form, right? People fill out millions of these things every day. Some might think that we don't need to put any more thought into how to design one—that the "problem" of creating a usable web form has already been solved, once and for all. But you should never underestimate the lack of effort that has been given to solving the most common design problems, particularly online. Most forms today are not much different than the ones that rolled out in the early 1990s.

Copyrighted material

Moreover, if there's one thing a good designer, especially one following UX principles, should know, it's that there is no such thing as a one-size-fits-all solution. Customization for your user group will almost always improve the experience—and, in this type of exercise, your data collection.

The results in our client's case appear to have been well worth the effort. We learned that, for this email marketing company's previous campaigns, normal rates of opened emails were in the 1–2% range; our mailing hit 4%. The normal click-through rate was 5–7% of opened emails; ours reached 21%. Most relevant, the normal rate of those who click through to the web page and then take action (i.e., complete the form) is usually 2–5%; for our design, that completion rate was 29%. (See Figure 2-8.)

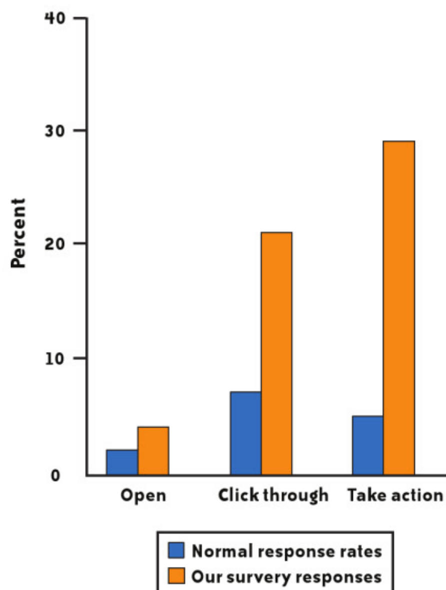


FIGURE 2-8. The response rates for our survey were significantly higher than the norm, which could be attributed to a better overall user experience.

There are, of course, other possible reasons why this survey performed so much better than this company's previous mailings. It's possible that the product was simply far more compelling than other products or topics on which the company had conducted surveys, and that the excitement generated by this product carried more people through to the end. It's also possible that the recipient pre-screening was far more accurate than usual, and this group was particularly well fitted to the product. There may even be an age bias at work—are older computer users more likely to open emails, read them, click through, and complete surveys than younger users, who may be more savvy and cautious about unsolicited emails? We're not aware of any studies on the subject, but it is a possibility. Indeed, although we can't rule out *any* of these explanations completely, the email company does

Copyrighted material

not appear to have been doing anything differently for our survey than it does for the hundreds of other surveys it regularly sends out. It's probably safe to conclude that our form design had something to do with the project's success.

Oh, and although it has no relevance to the survey design, we thought you might be interested to know that the reception of the product itself was extremely positive. While our client tells us that the product would have been viable to launch with a 10% positive response rate (answering "Yes" to the first survey question), it turned out that more than 16% of the respondents were interested in potentially buying it. What is the product? Unfortunately, confidentiality agreements preclude us from saying anything more about it.

If you want a glimpse, you'll just have to hope you're part of the next email survey. Don't be so quick to throw those emails in the trash; at the very least, you might learn something new about good—or bad—form design.

Copyrighted material

Copyrighted material

CHAPTER THREE

Embedded Image Data Processing on Mars

J. M. Hughes

Abstract

Spacecraft are unique engineering projects, with constraints and requirements not found in earth-bound artifacts. They must be able to endure harsh temperature extremes, the hard vacuum of space, and intense radiation, and still be lightweight enough for a rocket to loft them into space and send them to their destination. A spacecraft is an exercise in applied minimalism: just enough to do the job and no more. Everything that goes into the design is examined in terms of necessity, weight, and cost, and everything is tested, and then tested again, before launch day, including the embedded computer system that is the "brains" of the spacecraft and the software that runs on it. This chapter is an overview of how the image processing software on the *Phoenix* lander acquired and stored image data, processed the data, and finally sent the images back to Earth.

Introduction

When designing and programming an embedded system, one is faced with a variety of constraints. These include processor speed, execution deadlines, allowable interrupt latency, and memory constraints, among others. With a space mission, the constraints can be severe. Typically the computer onboard a space vehicle will have only enough expensive

Copyrighted material

radiation-hardened memory to fulfill the mission objectives. Its central processing unit (CPU) will typically be a custom-made device designed to withstand the damaging effects of high-energy cosmic rays. By commercial standards, the CPU isn't fast, which is typical of radiation-hardened electronics. The trade-off here is speed versus the ability to take a direct hit from an interstellar particle and keep on running. The dual-core CPU in a typical PC, for example, wouldn't last long in space (nor would much of the rest of the PC's electronics, for that matter).

Then there are the science objectives, which in turn drive the software requirements for functionality and performance. All must be reconciled within the confines of the spacecraft's computing environment, and after numerous trade-off decisions, the final product must be able to operate without fatal errors for the duration of the mission. In the case of a robotic spacecraft, any fault may be the end of the mission, so there are requirements for getting things right before the rockets light up and everything heads off into the wild blue yonder.

On May 25, 2008, the *Phoenix* Mars Lander touched down safely in the northern polar region of Mars. Figure 3-1 shows an artist's impression of what *Phoenix* might look like after landing. Unlike the rovers that moved about in the relatively warm regions near the Martian equator, *Phoenix* was a stationary lander sitting in a barren, frigid landscape where the atmospheric pressure is equivalent to being at an altitude of about 100,000 feet on Earth. The thin atmosphere on Mars is also mostly carbon dioxide. Not exactly an ideal vacation spot, but a good place to look for ancient frozen water.

FIGURE 3-1. Artist's impression of Phoenix on Mars (Image credit: NASA/JPL). (See Color Plate 10.)

Copyrighted material

The lander's mission was to look for direct evidence of water, presumably in the form of ice just below the surface (it found it, by the way), and possibly for indications that Mars could have once provided a habitat suitable for life. Because of the location of its landing site, the spacecraft had a limited lifespan; when the Martian winter set in, it would almost certainly be the end of *Phoenix*. At the high latitude of the landing site, the odds of the lander surviving a totally dark, frigid (–90° C or colder) winter under a blanket of carbon dioxide snow would be very, very slim, at best.

I was the principle software engineer for the imaging software on *Phoenix*, and in this chapter I will attempt to share with you some of the thinking that went into the various data-handling design decisions for the imaging flight software for the *Phoenix* Mars Lander. In JPL/NASA jargon it is called the "imaging flight software" because it was responsible for handling all the imaging chores on the surface of Mars, and it was qualified as "flight software" for the mission.

With the *Phoenix* Mars Lander, the challenge was to capture and process data from any of four different charge-coupled device (CCD) imagers (similar to what's in a common digital camera) simultaneously, and do it all in a very limited amount of pre-allocated memory in the spacecraft's main computer. Not only that, but the images might also need to be compressed prior to transmission back to Earth using one or more of several different compression methods. Just for good measure, some of the final data products (that is, the images) had to be chopped up into small segments, each with its own sequentially numbered header, to allow for efficient storage in the spacecraft's flash memory and reduce the amount of lost data should something happen to a packet during its journey from Mars to Earth. The resulting embedded code acquired and processed over 25,000 images during the operational lifetime of the *Phoenix* lander.

Some Background

But before we delve into the data handling, it would be a good idea to briefly introduce the main actors in the drama: the imagers (also referred to as the cameras) and the spacecraft's computer.

The primary computer on *Phoenix* was built around a RAD6000 CPU running at a maximum clock rate of 20 MHz, although it could also be operated at slower clock rates to conserve battery power. No cutting-edge technology here; this was basically a radiation-hardened, first-generation PowerPC with a mix of RAM and flash memory all crammed onto a set of VME circuit boards. After dealing with the landing chores, its primary functions involved handling communications with Earth (uplink and downlink in jargon-speak; see the sidebar "Uplink and Downlink" on page 38), monitoring the spacecraft's health, and coordinating the activities of the various science instruments via commands sent up from the ground. It used WindRiver's VxWorks real-time operating system (RTOS) with numerous extensions provided by the spacecraft contractor, Lockheed Martin. All of the flight software was written in C in accordance with a set of specific coding rules.

Copyrighted material

UPLINK AND DOWNLINK

In the jargon of space missions, the terms *uplink* and *downlink* refer to the transfer of data or commands to and from controllers on Earth to a spacecraft. Uplink refers to commands or data transferred to the spacecraft. Downlink occurs when the spacecraft sends data back to Earth.

Like many things in life, it's almost never a straightforward matter of pointing an antenna on the roof at the spacecraft and pressing the "Push To Talk" button. Commands or data to be uplinked must first pass through a review, and perhaps even some simulations, to make sure that everything is correct. Then, the commands and data are passed to mission controllers who will schedule when the uplink occurs (or is "radiated," in space-speak). And finally, it goes into NASA's Deep Space Network (DSN) communications system and gets radiated out into space. But that wasn't the final step, because in the case of *Phoenix* it had to be relayed by one of the orbiters now circling Mars, since *Phoenix* did not have the ability to talk to Earth directly. When the orbiter rose over the horizon on Mars, *Phoenix* would listen for any new uplink data.

Downlink was just as convoluted. Again, the orbiter would act as a relay, receiving the data from *Phoenix* and then passing it on to one of NASA's DSN antennas back on Earth. Then, it would make its way through various processing and relay steps until finally arriving at JPL. If it was image data, then the Mission Image Processing Laboratory (MIPL) at JPL would reassemble the images and make them available to the science teams eagerly awaiting the pictures at the science operations center at the University of Arizona.

Phoenix carried three primary cameras for surface science imaging: the Stereo Surface Imager (or SSI, with two CCDs), the Robotic Arm Camera (the RAC, with a single CCD), and the MECA Optical Microscope (OM) camera (again, a single CCD identical to the one used in the RAC). Figure 3-2 shows the flight model of the SSI, and Figure 3-3 shows the RAC attached to the robotic arm. The OM was tucked away inside the enclosure of the MECA instrument, which itself resembled a black box mounted on the upper deck surface of the lander.

The challenge was to devise a way to download the image data from each of the cameras, store the data in a pre-allocated memory location, process the data to remove known pixel defects, crop and/or scale the images, perform any commanded compression, and then slice-and-dice it all up into packets for hand-off to the main computer's downlink manager task for transmission back to Earth.

Each 1,024 × 1,024 pixel CCD in the SSI was capable of generating 2 megabytes of data, or 1 megapixel of 12-bit pixel values. Because it was a true stereo camera, the imagers in the SSI were often referred to as "eyes." Plus, it did look a bit like an old-fashioned robot's

Copyrighted material

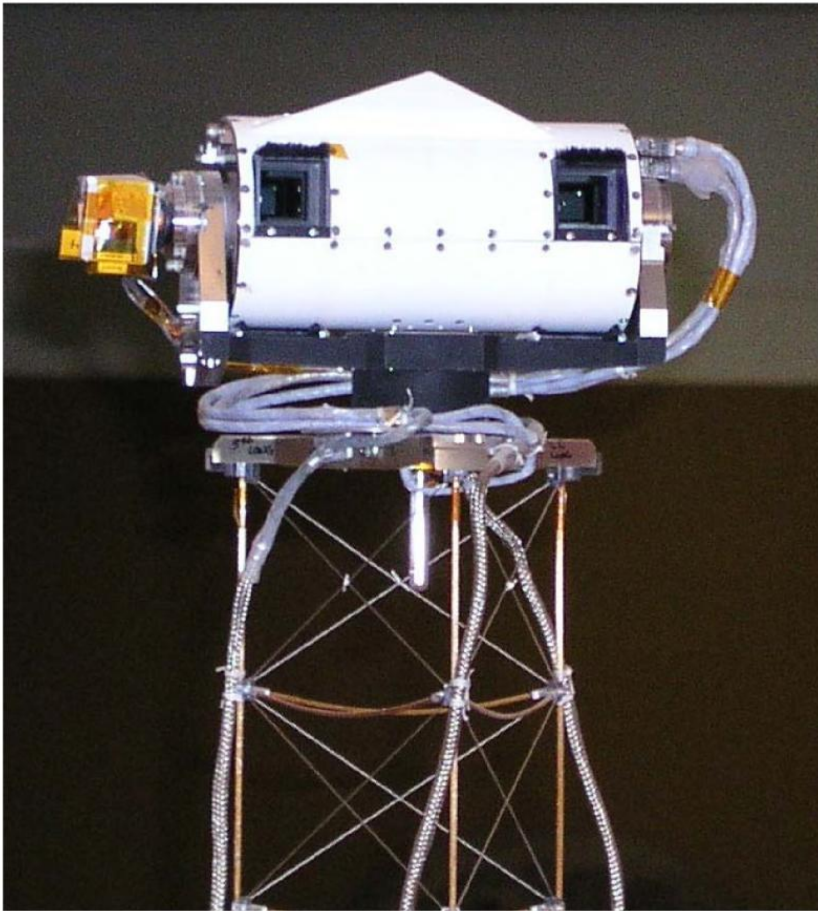


FIGURE 3-2. The Stereo Surface Imager (Image credit: University of Arizona/NASA/JPL). (See Color Plate 11.)

head. The RAC and OM cameras each contained a single 512 × 256 pixel CCD imager, and each generated 131,072 pixel values (or 262,144 bytes) of data (from now on I'll refer to both as the RAC/OM, because from the imaging software's point of view, they were identical imagers). Only 12 bits were actually used for each pixel worth of data from the CCD imagers, and what to do with the remaining 4 unused bits in a standard 16-bit "word" of memory generated some interesting discussions during the design phase, which I'll address in the next section. All of the images generated by SSI, RAC, and OM were monochrome, not color. Color was synthesized during processing back on Earth using separate images taken with either filters or special illumination.

I should note here that while the imaging software controlled the OM CCD to acquire images, it had nothing to do with the control of the MECA instrument and the electromechanical control of the optical microscope itself. That was handled by a separate real-time task written by the MECA team at JPL.

Copyrighted material

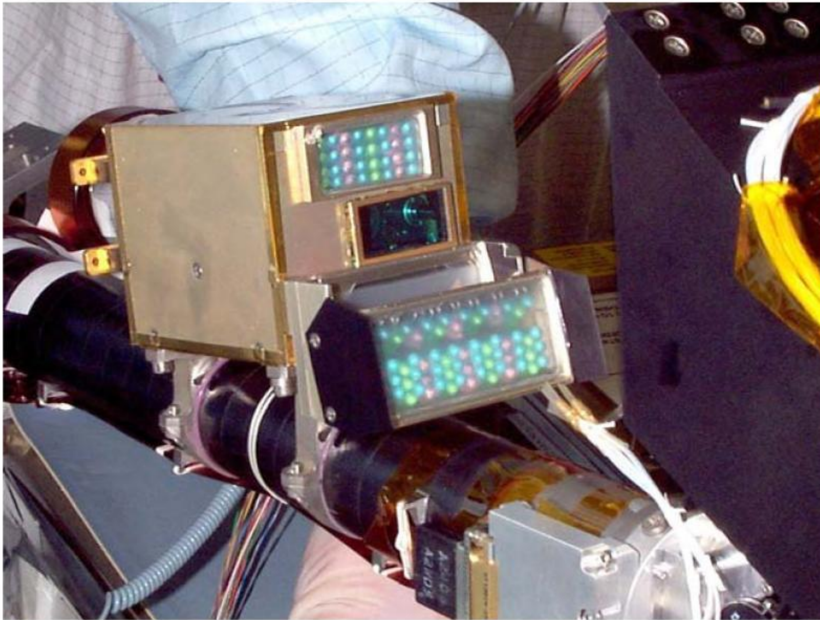


FIGURE 3-3. The Robotic Arm Camera (Image credit: University of Arizona/Max Planck/NASA/JPL).

Although 1 megapixel doesn't sound like much by the standards of today's consumer digital cameras, the CCD imagers used on *Phoenix* were custom-made for science imaging. Each CCD in the cameras cost tens of thousands of dollars, and only a limited number were ever made. They were reliable, robust, and precise, and each individual CCD was exhaustively tested and characterized pixel-by-pixel for sensitivity, noise, and defects, among other things. It is this level of characterization, and the reference data it generates, that sets a scientific CCD apart from the devices used in consumer cameras. Accurate characterization is what allows a researcher to have a high level of confidence that the image data accurately represents the scene that the camera captured. It is also a major contributor to the cost.

To Pack or Not to Pack

As with any highly constrained embedded system, the software needed to meet both its operational requirements and the constraints of its execution environment. As one might expect, these were not always complementary conditions, so trade-off decisions had to be made along the way. Both the SSI and the RAC/OM cameras utilized 12-bit conversion for the pixel data, which led to the first major trade-off decision: data packing. For a general high-level overview of binary data, see the sidebar "Binary Data" on page 41.

During the early design phase of the mission, the notion of packing the 12-bit pixel data came up and generated some interesting discussions. Given that only a limited amount of memory was available for image data storage, the concept of packing the 12-bit pixel data

Copyrighted material

into 16-bit memory space was appealing. By *packing*, I'm referring to storing the 12-bit pixel data contiguously, without any "wasted" bits in between—in effect ignoring 16-bit memory boundaries. But more efficient data storage came at the cost of increased processing time (unpacking, shifting, repacking). A digital image is an array (whether it's treated as a 1-D array or a 2-D array depends on what is being done to it), so any operation on an image involved handling the image data utilizing one or more algorithmic loops working through the array. The amount of data we were planning to push through the RAD6000 was significant, and even at the full-out clock rate of 20 MHz it was going to be painfully slow, so every CPU cycle counted.

In the end it was decided to "waste" a bit of memory and store each 12-bit pixel in a 16-bit memory location to keep things simple and avoid using any more CPU time than necessary. This decision was also driven by the desire, established early on, to avoid the use of multiple large processing buffers or result arrays by doing all image processing and compression in-place. Data packing would have made this rather challenging, and the resulting code would have been overly complex and could have shot down the whole in-place processing concept we wanted to implement.

BINARY DATA

Data is information. In computer systems it is represented numerically, since that it what the CPU in a computer deals with. Data can represent text, wherein each character has a unique numeric value, or it can represent images by encoding each pixel, or picture element, in an image with a numeric value representing its intensity, its color, or a combination of both characteristics. Given an appropriate numerical encoding scheme, a computer can process any type of data one might care to imagine, including audio, electrical potentials, text, images, or even the set of characteristics that define the differences between dogs and cats. But no matter what it represents, to the computer it's all just numbers. We supply the rules for how it will be encoded, processed, displayed, and interpreted.

Data also comes in a variety of sizes, depending on what it represents. For example, a window or door switch in a burglar alarm system needs only a single bit (or binary digit) to represent its two possible states: open or closed, 0 or 1. To represent a character in the English alphabet and punctuation, one needs about 100 numbers or so, each represented by 8 bits of data. Modern computers work in base 2, so 8 bits could represent any number from 0 to 255 ($11001000_2 = 200_{10}$, for example). In many computers there are preferred sizes for values expressed in base 2, typically in multiples of either 8 or 16 bits. In a 16- or 32-bit CPU, like the one used on *Phoenix*, memory can be efficiently accessed in "words" of 16 bits. Trying to access fewer bits than this (such as 8 bits) may actually be inefficient, so data that is greater than 8 bits in size but less than 16 bits is often stored in a 16-bit memory location along with some unused bits. This is how the image data on *Phoenix* was handled, because the electronics for the CCD imagers produced 12-bit-per-pixel data, but the spacecraft's memory was organized as either 16- or 32-bit storage locations.

Copyrighted material

The Three Tasks

In the VxWorks RTOS environment used for *Phoenix*, there really isn't anything that is synonymous to what a typical computer user might think of as individual programs. Nothing is loaded from a disk (there are no disks), and everything the computer will ever do is loaded into memory when the system first starts. It's actually all just one big program with a lot of smaller subprograms running more or less at the same time. These smaller subprogram activities are referred to as *tasks*, or *threads*, and they execute based on the availability of resources such as timed events or I/O (input/output) devices, and their assigned priority in the greater scheme of things (high-priority tasks get the chance to run more often than low-priority activities).

It was obvious from the outset that a minimum of two tasks would be needed for the surface image processing, one for each of the cameras. The SSI and RAC/OM were very different beasts, with different command sets and different operating characteristics. The SSI used all new controller hardware and incorporated CCD imagers identical to those used on the Mars Rovers. The RAC and OM imagers were originally built by the Max Planck Institute in Germany, and had been around for a while (one of the original designs was flown on the Huygens probe that landed on Titan). The RAC/OM controller hardware was actually a flight spare unit from the ill-fated Mars '98 mission, which apparently met a tragic end when its descent engines shut off prematurely a few hundred feet above the surface of Mars. But the data from each camera still needed to be processed, compressed, and then downlinked, and these operations weren't dependent on the physical data source. The image data was all 12 bits per pixel, and all that really varied was the geometry (height and width), and consequently how much image data would need to be handled.

Although there was a desire on the part of the spacecraft integration team (Lockheed Martin) to try to keep the number of science instrument tasks to a minimum, it became obvious early on that it didn't make much sense to duplicate the same image compression and downlink functions in both camera tasks. This would be wasteful in terms of limited program storage space, and it would effectively double the effort necessary to make changes in the code and then verify those changes. Consequently, a decision was made to use three tasks: one task would handle the SSI, one would deal with the RAC and OM cameras (one or the other, but not both at the same time, because the interface hardware wouldn't allow it), and a third would act as a shared resource to perform image compression and downlink processing using the data generated by the two camera control tasks, and it would run asynchronously. So while the SSI and RAC/OM tasks interfaced with the control electronics to acquire image data, control the internal temperature of the cameras, and perform motion control, the third task would do nothing but image data processing and downlink.

The image processing task was called the ICS, which stood for Image Compression SubSystem, although it ended up doing more than just compressing image data. A block diagram of the three tasks and the communications among them is shown in Figure 3-4.

Copyrighted material

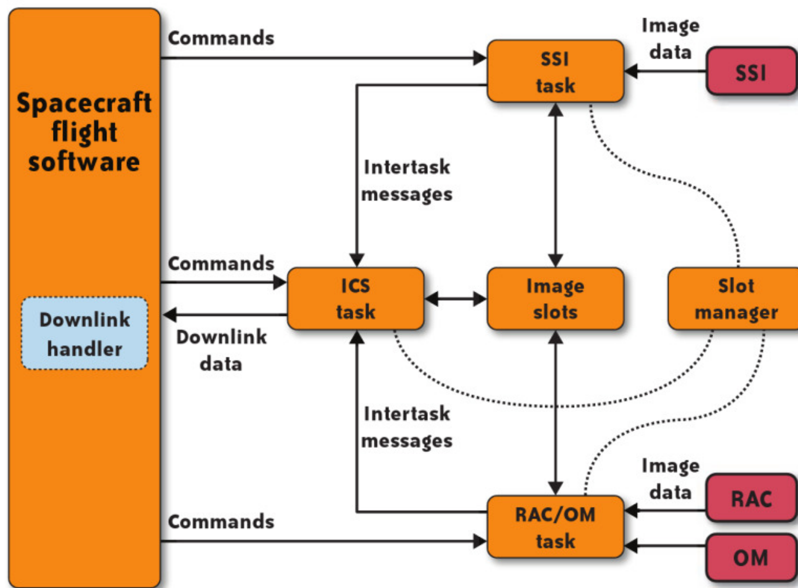


FIGURE 3-4. Imaging flight software tasks.

The decision to use three tasks did increase the level of complexity in the system, but it also reduced the amount of program storage space required. As an added bonus, having a third independent task made it much easier to make a change to a particular ICS processing function, and then test the functionality with data from either of the imaging tasks using real camera hardware or from a simulated image data source. This turned out to be a boon when doing extensive compression testing later on in the project, when over 15,000 test images were processed back-to-back through the ICS to verify its operation using an automated test setup.

The ICS also included two source modules, which contained shared functions for static memory management (known as the "slot manager") and image manipulation (decimation, subframing, pixel defect correction, and so on). These were not actually part of the ICS task, but rather served as thread-safe pseudolibraries to support the two camera control tasks and the ICS task. The reality was that there really wasn't any other convenient place to put this code, given the architecture constraints imposed on the instrument software, so it ended up with the ICS.

Slotting the Images

I mentioned earlier that the amount of memory available to each of the various instrument tasks was limited, but just how limited may be surprising to some, given that it is now commonplace to find 500 megabytes or even a gigabyte (or more) in a desktop PC. The initial memory allocation to both the SSI and RAC/OM tasks for image data storage was 230K short of a full 10 megabytes (10,255,360 bytes, to be exact). There was discussion of

Copyrighted material

increasing this after the spacecraft landed, which meant that any memory management scheme had to be flexible, but this was the design baseline. The default storage scheme needed to be able to handle at least four SSI images (or two pairs, consisting of one image for each "eye") and at least four RAC/OM images, all in the same memory space. The odd size meant that it wouldn't be possible to squeeze in more than four full-size SSI images, at least not initially.

In embedded systems, the use of dynamic memory allocation is usually considered to be a Really Bad Idea. To avoid issues with fragmented memory, memory leaks, null pointers, mystery crashes, and the possibility of losing the mission completely, the use of dynamic memory allocation (C's `malloc` function and its kin) was forbidden by the flight software coding rules. This meant that the imaging software had to manage the image data itself within whatever amount of memory was assigned to it, and it had to be robust and reliable.

The solution was the use of a set of functions that acted as a memory manager for the pre-allocated memory assigned to the ICS at boot-time. The memory manager was the key component of the image data processing. To prevent collisions, blocking semaphores were used to control shared access by each of the three imaging tasks (actually, any task in the spacecraft software could have used the shared memory, but only the cameras did so).

The static memory allocation was divided up into "slots," which could be either large enough to hold a full-size SSI image, or a smaller size for RAC/OM images. Figure 3-5 shows the default organization of the ICS image storage space.

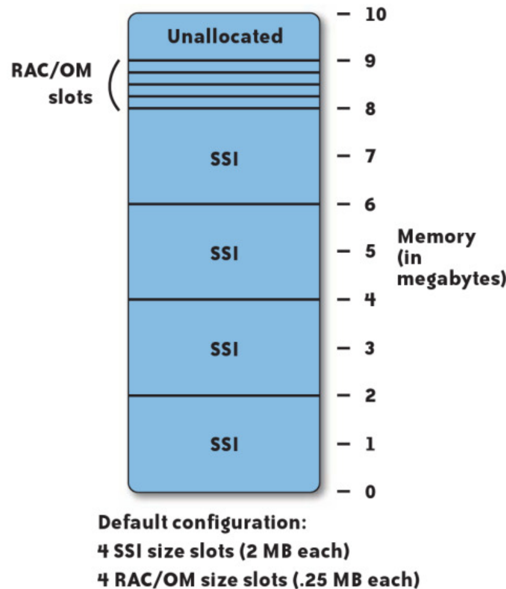


FIGURE 3-5. Default image slot assignments.

This is only one possible configuration, and the number of each type of slot could be changed on the fly via commands uplinked from Earth.

Copyrighted material

Each image also had an associated structure containing header data. The image header recorded things such as a code defining the camera that generated the image, the exposure time, the image processing options selected, the image dimensions, optical filters that may have been used, and how the image was compressed (if compression was used). Part of the header was filled in by the instrument task that generated the image, and the remainder was filled in by the ICS prior to sending the image data to the spacecraft downlink handler. Because the header data was not image data per se, it was stored in a separate set of slots until it was time to do the downlink operation. Each image slot and its associated header data had to be tracked and processed in tandem.

The memory manager was basically just a set of functions that operated on a set of arrays of structures, as shown in Figure 3-6. The current state of the memory slots was maintained by the arrays, which, in essence, constituted a dynamic model of the physical memory space and its contents.

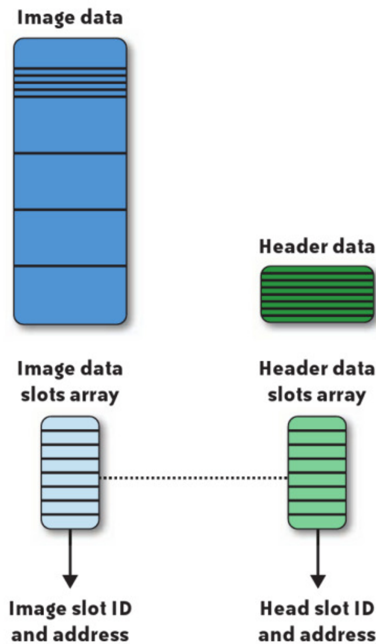


FIGURE 3-6. Memory slot manager arrays.

One of the arrays contained structures for image data, one per image slot. The C `typedef` for the structure is:

```
typedef struct {
    uint16_t    slot_status;          /**< Owned or unowned              */
    int16_t     slot_owner;           /**< -1 if slot is unowned         */
    int16_t     slot_size;            /**< either RAC/OM or SSI sized    */
    uint16_t    *slot_address;        /**< address of data space of slot */
} ics_img_slot_entry_t;
```


Copyrighted material

The second array contained structures pointing to header data entries, and its definition is:

```
typedef struct {
    uint16_t    slot_status;           /**< Owned or unowned                    */
    int16_t     slot_owner;            /**< -1 if unowned                       */
    int16_t     img_id;                /**< associated image data slot number   */
    uint16_t    hdr_data[ICS_HDR_SLOT_SZ];  /**< array for header data          */
} ics_hdr_slot_entry_t;
```

Notice that the header structure contains an entry for the image ID. This was essential, since slots could be allocated and released in any order, and there was no guarantee that the index of an image slot entry would be the same index for its associated header slot. Rather than rely on the index offset into the arrays always being in sync, the image ID was used to bind image and header data entries together.

The ability to dynamically reconfigure the image slot assignments allowed the memory manager to be tailored to specific mission activities. If the plan for a particular day on Mars (or a Sol, as it was called) involved imaging with the SSI, then one could configure the slots to minimize the number of RAC/MECA size allocations, which was the default configuration. If, on the other hand, the plan involved a lot of RAC or OM images, the memory could be configured to handle no SSI images and up to 39 of the smaller image sizes.

Passing the Image: Communication Among the Three Tasks

Image data fresh from one of the cameras was written into a slot by one of the camera tasks. After performing any required pixel correction or subframe operations, the camera task notified the ICS that a new image was available for processing. The ICS would then perform any commanded compression (either lossy or lossless) in place on the image within its slot, and then package and hand off the data for downlink. Only after the downlink was complete would the slot be released and become available for a new image. The sequence of events from exposure to image hand-off for the SSI camera is shown in Figure 3-7.

The entire sequence of events shown in Figure 3-6 was contingent on the availability of an image slot. If a slot was not available, the camera task would wait for a configurable period of time to allow the ICS to finish compressing and downlinking an image, which would result in a slot becoming available. If the ICS didn't release a slot within that period of time, the instrument task would generate an error message for the operators back on Earth and drop the image on the floor (there really wasn't any place else to drop it).

Once one of the camera instrument tasks obtained a slot, it "owned" that slot until it was handed off to the ICS, which then became the owner. Ownership verification was based on the slot ID (its number) assigned by the slot manager when the slot was initially allocated, an image ID code, and the camera instrument task ID. When a hand-off was made to the ICS, it verified that the ID codes presented matched those already recorded for that particular image slot.

Copyrighted material

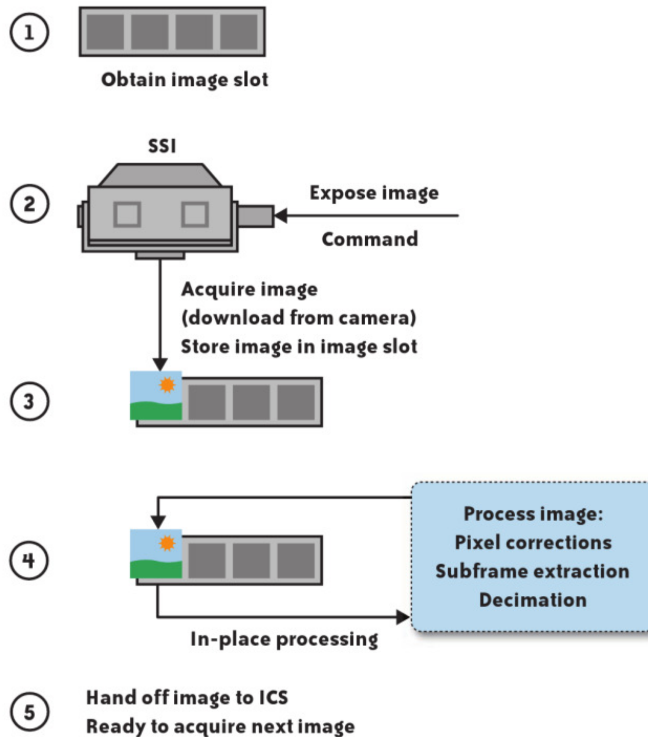


FIGURE 3-7. Image acquisition and hand-off sequence.

As mentioned earlier, the ICS ran asynchronously and was not tightly bound to either of the camera tasks. It was able to do this by leveraging the built-in message queue system in VxWorks, and through the use of the shared functions in the memory slot manager. Figure 3-8 shows a message sequence chart (MSC) type of representation of the steps involved along the way in getting data from a camera to finally sending it to downlink.

The use of the internal message queue (the S/C FSW process line, which stands for *spacecraft flight software*) allowed either camera task to issue a command to the ICS using the same mechanism as the commands uplinked from Earth. The command would sit in the queue until the ICS was done with its current activity. The cameras could continue to acquire images as long as there were slots available to store the image data, and the ICS would retrieve and process the data in turn until the queue was empty, without regard for what the cameras might be doing.

Note that Figure 3-8 doesn't show the error checking that went on during imaging activities. All in all, the number of lines of code dedicated to error checking and fault handling was roughly equal to the lines of code that actually processed or otherwise handled the data. Failure Mode, Effects, and Criticality Analysis (FMECA) techniques were employed early in the design life cycle and provided guidance during the implementation of the software and its fault-handling capabilities.

Copyrighted material

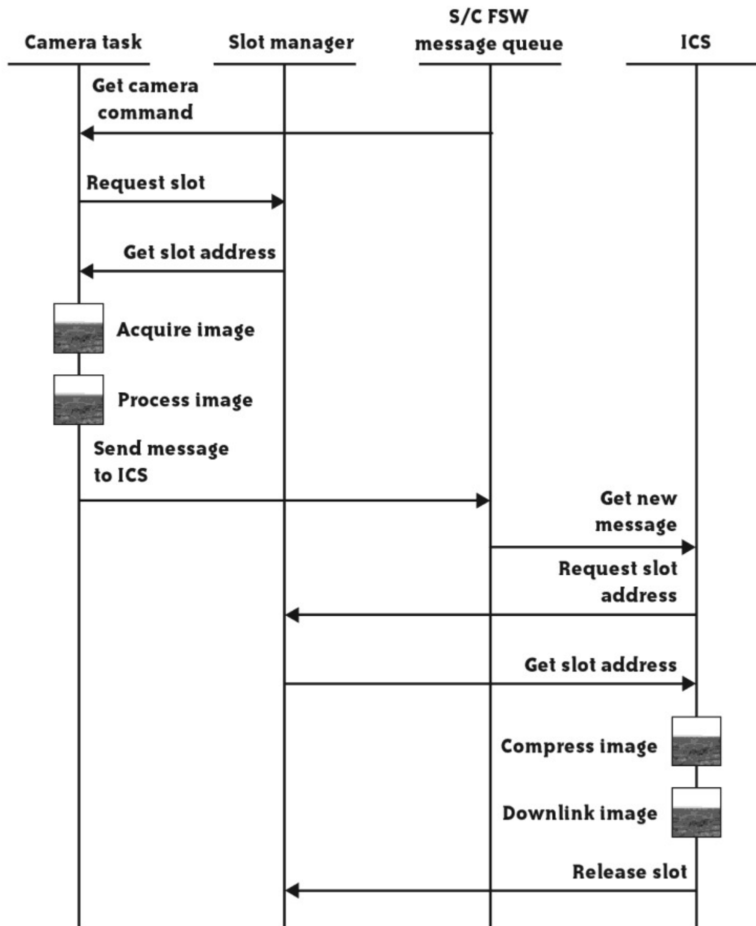


FIGURE 3-8. MSC representation of image acquisition, processing, and downlink activities.

The ICS serialized the data stream, but the use of the image slots and the command message queue allowed sets of images to be acquired in rapid (relatively speaking) succession. It also meant that there was some timing margin available for image acquisition that reduced the chances of operations being suspended while waiting for an image to be downlinked. Early test command sequences demonstrated that it was possible to do things like creating short "movies" (well, sort of, since it took about six seconds to download each image from one of the SSI cameras), or generate a large (30+ images) data set using the RAC or OM at different focal lengths.

Getting the Picture: Image Download and Processing

A lot went on between the time an image exposure occurred and the eventual hand-off to the ICS. Each camera in the system had its own control electronics to process commands, convert the analog signals from a CCD into 12-bit digital values, and then store the data in a hardware buffer until the flight software could download it into an image slot.

Copyrighted material

All this occurred under the control of logic embedded in radiation-hardened programmable gate array devices.

Once an image was acquired from a camera and written into an image slot, it was subjected to various forms of processing, all of which occurred in-place within the confines of an image slot. No additional large (image-sized) buffers were used for the processing or the results thereof, and only a few small buffers were necessary to hold intermediate results. The use of in-place processing was a key factor in the design of the imaging software, and allowed the three tasks to maintain a small memory footprint in the overall system. Figure 3-9 shows a comparison between a multiple-buffer approach and the single buffer (i.e., slot) in-place design used for the *Phoenix* imaging flight software.

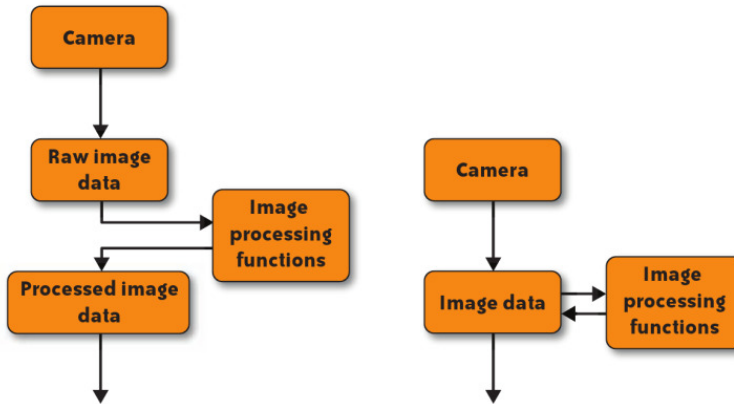


FIGURE 3-9. Multiple data buffers versus a single data buffer.

This was another design trade-off that was made early on in order to meet the image processing requirements and still stay within the amount of memory allocated to the cameras. Although it did meet the memory requirements, the downside was that there would be no “undo” operation. As shown in Figure 3-10, if an error occurred during image processing, either the entire image would be lost or a partially garbled image might be returned.

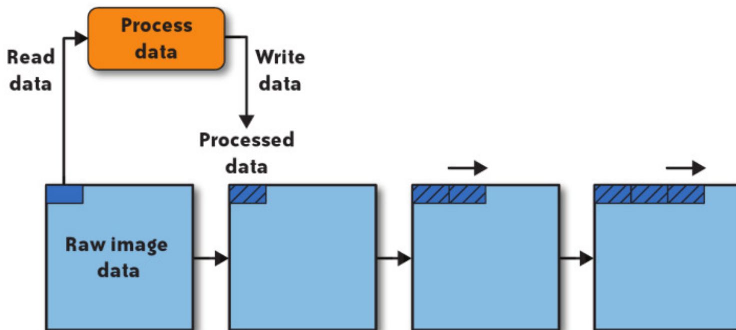


FIGURE 3-10. In-place data processing.

Copyrighted material

The processing algorithms walked through the data in an image slot, reading, processing, and then writing the data back. Some of the algorithms, such as pixel corrections, didn't change the geometry of the image but instead simply modified a single pixel value based on an uplinked table of known "bad" pixels (a pixel might be bad because it is not as sensitive as its neighbors, or it might be too sensitive). In the environment of space, it was expected that the odd cosmic ray could possibly blast through a pixel on a CCD and render it defective. Other operations, such as subframing, extracted a region from the original image, wrote it back into the slot, and adjusted the height and width parameters accordingly. Decimation employed a mathematical averaging technique to reduce image size by processing pixels in groups of 4, 9, or 16 to generate a single result pixel. The resulting images were reduced by 1/2, 1/3, or 1/4 in size, respectively, while minimizing the "stair-step" effect often seen with images that have been reduced using a subsampling technique wherein every 2nd, 3rd, or 4th pixel is retained and the rest discarded. This operation also wrote the modified data back into the image slot and adjusted the geometry parameters accordingly.

After an imaging task had completed the commanded processing, it would then send a message to the ICS (as described previously) and move on to the next command in the message queue. If an image slot was available, this could result in acquiring yet another image.

Image Compression

Just as the data produced by a robotic mission is precious, so is the communications bandwidth needed to return that data. For smaller images, such as those reduced by subframe or decimation operations, it could be acceptable to just downlink the image without compression. Larger images, such as the full-size SSI images, would consume a lot of downlink bandwidth, so compression was always considered as an option in such cases.

The ICS provided two forms of compression and two forms of size reduction using pixel mapping and scaling. Which type of compression or reduction would be used for a particular image depended largely on the level of image fidelity deemed necessary for the object of interest. In some cases, 8 bits per pixel would suffice; in other cases, the loss of fidelity inherent with JPEG compression was acceptable; and for the cases where the image had to retain as much fidelity as possible, there was a lossless compression method available.

In the ICS, a JPEG compressor, using all integer math and in-place operations, provided so-called "lossy" compression. JPEG is considered lossy because it discards some of the image data as a result of the compression process. It could compress image data to varying degrees by command. The final code was loosely based on the JPEG compressor flown on the Mars '98 mission, although only a part of that original code survived in the ICS for *Phoenix*. The original JPEG compressor used floating-point math, multiple full-size image arrays as buffers, and dynamic memory allocation. How that ever managed to make it into flight software is still a mystery to me, but it did. The use of floating-point values to represent

Copyrighted material

pixel data in the compression code also meant that it consumed four times as much memory per image as the native 16-bit integer representation of the original image.

A second form of compression, known as Rice Lossless or just Rice compression, used an algorithm developed by Robert Rice of the Jet Propulsion Laboratory. The Rice algorithm could compress image data by almost 2:1 with no data loss, whereas the JPEG algorithm discarded data during the compression. The Rice compression also operated on the image in-place in the image slot.

The two noncompression reduction methods used either a lookup table to map 12-bit pixel values to 8-bit values, or a bit reduction method that shifted the pixel data right by 4 bits to yield an 8-bit-per-pixel image. Both the JPEG and Rice compression functions would accept either 12- or 8-bit image data.

The decision to use the lossy JPEG compression or not typically came down to weighing various factors such as how accurate the data needed to be, how much bandwidth would be available, how much downlink storage was available in the spacecraft's main computer, and how much time was available to perform the compression (recall that the RAD6000 had a top speed of 20 MHz, so compressing a megapixel of image data could take over a minute).

When using the JPEG compression, the amount of compression to be applied was determined by a command parameter that specified the worst-case reduction ratio for the final data. In other words, instead of specifying a "quality" factor (which is typically how one tells a JPEG compressor how hard to work on an image), the ICS used a scaling factor and worked out the required compression level on its own. This was based on a quick-look analysis of the overall image entropy. The image entropy was an estimate of how "busy" the image was, and images with a higher level of entropy (lots of details and changes in brightness, such as a pebble-strewn patch of ground with sharp shadows) would require a higher compression setting to meet the final size goal. Images with low entropy, such as the Martian sky with a few clouds drifting by, wouldn't have a whole lot going on, and so would require a lesser amount of compression to meet the size target.

The scaling factor for JPEG compression was also used to divide the original image into segments. These segments were then fed into the JPEG compressor one at a time, and the output was written back into the image slot. The final result in the image slot prior to downlink was a set of small, self-contained JPEG images, the total size of which was equal to or less than the commanded size reduction ratio for the original image.

The Rice compressor included its own embedded method of segmentation, and it was downlinked by simply reading out the compressed data in the form of small packets sized to fit neatly into the flash memory in the spacecraft's main computer. The output of the lookup table and bit-reduction methods was also simply read out in flash-sized packets for downlink.

Copyrighted material

Downlink, or, It's All Downhill from Here

The last step in the process was the hand-off to the downlink manager in the spacecraft flight software. Some science instruments could simply pass their data to downlink and be done with it, but because of the large amount of data and the use of packetization, the ICS ended up doing a lot of downlink preprocessing on its own.

For the JPEG data, this meant handling each of the compressed segments individually. The first and last segments in a sequence always included a full-sized image data header. The intermediate segments got a smaller form of the header data, which included an image ID code and a sequence number. As each segment was read from the image slot, the header data was applied. The use of a sliding window form of readout allowed the segments to be packed end-to-end while assembling a flash-sized packet. This in turn allowed the downlink handler to maximize the use of the temporary flash storage space, because some of the compressed segments could be smaller than others if the part of the image corresponding to a segment had a low entropy. In fact, it was common to see compressed segment sizes vary widely, so packing them end-to-end avoided wasting any of the on-board flash memory.

Because the data consisted of uniquely identified segments, the loss of a downlink packet wouldn't consign the entire image to the garbage. The reconstruction and decompression software back on the ground at JPL could figure out what segments were missing and simply fill in the missing part of the image with black zero-value pixel data. If the missing data showed up later (which was possible, considering the rather torturous route the data took on its way down), then it could be placed into the image to fill in the missing pixels.

Once the data was passed to the downlink handler, the ICS was done, and it would release the image data slot. The entire process—from image exposure to completion of downlink hand-off—took between 3 to 10 minutes, depending on the CPU speed and what other additional imaging activities where slated to occur, such as auto-exposure and sun-finding (which are complex topics in their own right, so I haven't discussed them here).

Conclusion

The instrument software did much more than just take pictures and process image data. It also managed motion control with three degrees of freedom for the SSI, and the focus and viewport cover motors in the RAC. The RAC also supported multiple banks of red, blue, and green LEDs to illuminate whatever might be in the robotic arm scoop and create color images. Both the SSI and the RAC incorporated active thermal control, achieved either through the use of special heaters or by intentionally stalling a stepper motor to achieve self-heating. On top of all this, there was the error-checking and fault-recovery code. All in all, it was very busy software.

Copyrighted material

If I had it to do all over again, I suppose the main thing I would want to see changed would be that the cameras use their own embedded processors rather than rely on the spacecraft CPU. This would have made things much easier all around for everyone. Apart from that, I always felt that there was too much crammed into each of the instrument tasks. In other words, the thermal control should have been a separate task for each camera. This would have greatly reduced the complexity of each of the tasks, albeit at the expense of increasing the overall complexity of the intertask communications. At the outset, however, there wasn't enough evidence to build a compelling case for this, so the design was already firm (not really frozen, just very inflexible) by the time some new thermal requirements popped up that needed to be accommodated.

And, finally, I really had issues with the method chosen for performing a "heartbeat" check. I didn't know going in that the command message queue was going to be used for this purpose. What this did was impose a requirement on the instrument tasks to be able to drop whatever they were doing in order to check the command message queue on a regular basis for a "ping" message. I believe that a much better approach would have been for the instrument to register a callback function with the spacecraft flight software that could be used to check the value of a continuously updated counter variable on an asynchronous basis. If the value didn't change after some amount of time, the instrument task was probably hung. There was indeed a rather big squabble over this, but in the end the ping message was used simply because that's what had always been done and that's what the existing test systems were designed to handle. So even though the system wasn't designed to deal with tasks that could take minutes to process large amounts of image data, it wasn't going to be changed.

The *Phoenix* SSI and RAC/OM imaging software was a lot of work to design, implement, and test, and in the end it did what it was supposed to do for the entire life of the mission. Figure 3-11 is one of the first images (SS001EDN896308958_10D28R1M1) returned from the SSI on Sol 1, the spacecraft's first full day on Mars.

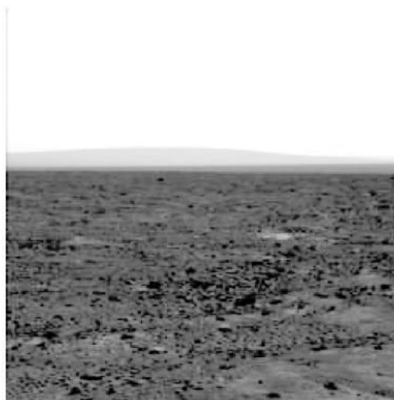


FIGURE 3-11. Image returned from the SSI on Sol 1 (Image credit: NASA/JPL/University of Arizona).

Copyrighted material

LEARNING MORE ABOUT PHOENIX

If you would like to know about the *Phoenix* mission, these are the primary places to start:

- *Phoenix* website at the University of Arizona: *http://phoenix.lpl.arizona.edu*
- *Phoenix* website at the Jet Propulsion Laboratory: *http://www.jpl.nasa.gov/news/phoenix/main.php*
- NASA's *Phoenix* website: *http://www.nasa.gov/mission_pages/phoenix/main/index.html*

At JPL, the MIPL folks do a lot of image processing for a variety of missions. You can learn more about what they do here:

- JPL's Mission Image Processing Laboratory: *http://www-mipl.jpl.nasa.gov/*

And if you would like to learn more about the RAD6000 CPU, image processing, or embedded systems, be sure to check out Wikipedia at *http://www.wikipedia.org*.

Copyrighted material

CHAPTER FOUR

Cloud Storage Design in a PNUTShell

Brian F. Cooper, Raghu Ramakrishnan, and Utkarsh Srivastava

Introduction

Yahoo! runs some of the world's most popular websites, and every month over half a billion people visit those sites. These websites are powered by database infrastructures that store user profiles, photos, restaurant reviews, blog posts, and a remarkable array of other kinds of data. Yahoo! has developed and deployed mature, stable database architectures to support its sites, and to provide low-latency access to data so that pages load quickly.

Unfortunately, these systems suffer from some important limitations. First, adding system capacity is often difficult, requiring months of planning and data reorganization, and impacting the quality of service experienced by applications during the transition. Some systems have a hard upper limit on the scale they can support, even if sufficient hardware were to be added. Second, many systems were designed a long time ago, with a single datacenter in mind. Since then, Yahoo! has grown to a global brand with a large user base spread all over the world. To provide these users with a good experience, we have to replicate data to be close to them so that their pages load quickly. Since the database systems did not provide global replication as a built-in feature, applications had to build it themselves, resulting in complex application logic and brittle infrastructure. Because of all the effort required to deploy a large-scale, geographically replicated database architecture, it was hard to quickly roll out new applications or new features of existing applications that depended on that architecture.

Copyrighted material

PNUTS is a system that aims to support Yahoo!'s websites and application platforms and address these limitations (Cooper et al. 2008). It is designed to be operated as a storage cloud that efficiently handles mixed read and write workloads from tenant applications and supports global data replication natively. Like many other distributed systems, PNUTS achieves high performance and scalability by horizontally partitioning the data across an array of storage servers. Complex analysis or decision-support workloads are not our focus. Our system makes two properties first-class features, baked in from the start:

Scale-out
: Data is partitioned across servers, and adding capacity is as easy as adding new servers. The system smoothly transfers load to the new servers.

Geo-replication
: Data is automatically replicated around the world. Once the developer tells the system at which colos* to replicate the data, the system takes care of the details of making it happen, including the details of handling failures (of machines, links, and even entire colos).

We also set several other goals for the system. In particular, we want application developers to be able to focus on the logic of their application, not on the nuts and bolts of operating the database. So we decided to make the database hosted, and to provide a simple, clean API to allow a developer to store and access data without having to tune a large number of parameters. Because the system is to be hosted, we wanted to make it as self-maintainable as possible.

While all of these goals are important to us, building a database system that could both scale-out and globally replicate data was the most compelling and immediate value proposition for the company. And as we began to design the system, it became clear that this required us to rethink many well-understood and long-used mechanisms in database systems (Ramakrishnan and Gehrke 2002).

The key idea we use to achieve both scale-out and geo-replication is to carry out only simple, cheap operations synchronously, and to do all the expensive heavy lifting asynchronously in the background. For example, when a user in California is trying to tag a photo with a keyword, she definitely does not want to wait for the system to commit that tag to the Singapore replica of the tag database (the network latency from California to Singapore can be as high as a second). However, she still wants her friend in Singapore to be able to see the tag, so the Singapore replica must be updated asynchronously in the background, quickly (in seconds or less) and reliably.

As another example of how we leverage asynchrony, consider queries such as aggregations and joins that typically require examining data on many different servers. As we scale out, the probability that some of these servers are slow or down increases, thereby adversely affecting request latency. To remedy this problem, we can maintain materialized

* *Colo*cation facility, or data center. Yahoo! operates a large number of these, spread across the world.

Copyrighted material

views that reorganize the base data so that (a predetermined set of) complex queries can be answered by accessing a single server. Similar to database replicas, updating each view synchronously would be prohibitively slow on writes. Hence, our approach is to update views asynchronously.

In the rest of this chapter, we explore the implications of focusing on scale-out and geo-replication as first-class features. We illustrate the main issues with an example, explain our basic approach, and discuss several issues and extensions. We then compare PNUTS with alternative approaches. Our discussion concentrates on the design philosophy, rather than the details of system architecture or implementation, and covers some features that are not in the current production version of the system in order to highlight the choices made in the overall approach.

Updating Data

As users interact with websites, their actions constantly result in database updates. The first challenge we examine is how to support this massive stream of updates while providing good performance and consistency for each update.

The Challenge

Imagine that we want to build a social networking site. Each user in our system will have a profile record, listing the user's name, hobbies, and so on. A user "Alice" might have friends all over the world who want to view her profile, and read requests must be served with stringent low-latency requirements. For this, we must ensure that Alice's profile record (and similarly, everyone else's) is globally replicated so those friends can access a local copy of the profile. Now say that one feature of our social network is that users can update their status by specifying free text. For example, Alice might change her status to "Busy on the phone," and then later change it to "Off the phone, anybody wanna chat?" When Alice changes her status, we write it into her profile record so that her friends can see it. The profile table might look like Table 4-1. Notice that to support evolving web applications, we must allow for a flexible schema and sparse data; not every record will have a value for every field, and adding new fields must be cheap.

TABLE 4-1. User profile table

Username	FullName	Location	Status	IM	BlogID	Photo	...
Alice	Alice Smith	Sunnyvale, CA	Off the phone, anybody wanna chat?	Alice345			...
Bob	Bob Jones	Singapore	Eating dinner		3411	*me.jpg*	...
Charles	Charles Adams	New York, New York	Sleeping		5539		...
...							

How should we update her profile record? A standard database answer is to make the update atomic by opening a *transaction*, writing all the replicas, and then closing the transaction by sending a commit message to all of the replicas. This approach, in line with the

Copyrighted material

standard ACID* model of database transactions, ensures that all replicas are properly updated to a new status. Even non-ACID databases, such as Google's BigTable (Chang et al. 2006), use a similar approach to synchronously update all copies of the data. Unfortunately, this approach works very poorly if we have geo-replication. Once Alice enters her status and clicks "OK," she may potentially wait a long time for her response page to load, as we wait for far-flung datacenters to commit the transaction. Moreover, to guarantee true atomicity, we would have to exclusive-lock Alice's status while the transaction is in progress, which means that other users will potentially be unable to see her status for a long time.

Because of the expense of atomic transactions in geographically separated replicas, many web databases take a *best-effort* approach: the update is written to one copy and then asynchronously propagated to the rest of the replicas. No locks are taken or validation performed to simulate a transaction. As the name "best-effort" implies, this approach is fraught with difficulty. Even if we can guarantee that the update is applied at all replicas, we cannot guarantee that the database ends in a consistent state. Consider a situation where Alice first updates her status to "Busy," which results in a write to a colo on the west coast of the U.S., as shown in Table 4-2.

TABLE 4-2. An update has been applied to the west coast replica

West coast	
Username	**Status**
Alice	Busy

East coast	
Username	**Status**
Alice	--

She then updates her status to "Off the phone," but due to a network disruption, her update is directed to an east coast replica, as shown in Table 4-3.

TABLE 4-3. A second update has been applied to the east coast replica

West coast	
Username	**Status**
Alice	Busy

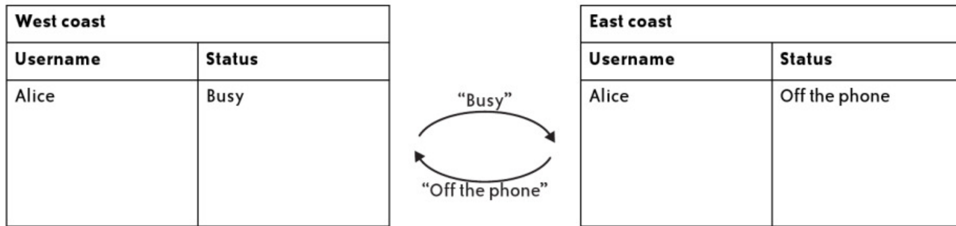
East coast	
Username	**Status**
Alice	Off the phone

Since update propagation is asynchronous, a possible sequence of events is as follows: "Off the phone" is written at the east coast before the "Busy" update reaches the east coast. Then, the propagated updates cross over the wire, as shown in Table 4-4.

* A transaction's changes are Atomic, Consistent, Isolated from the effects of other concurrent transactions, and Durable.

Copyrighted material

TABLE 4-4. The two updates cross during propagation



West coast	
Username	**Status**
Alice	Busy

East coast	
Username	**Status**
Alice	Off the phone

The "Busy" status overwrites the "Off the phone" status on the east coast, while the "Off the phone" status overwrites the "Busy" status on the west coast, resulting in the state shown in Table 4-5.

TABLE 4-5. Inconsistent replicas

West coast	
Username	**Status**
Alice	Off the phone

East coast	
Username	**Status**
Alice	Busy

Depending on which replica her friends look at, Alice's status will be different, and this anomaly will persist until Alice changes her status again.

To deal with this problem, some web-scale data stores implement *eventual consistency*: while anomalies like that described earlier may happen temporarily, eventually the database will resolve inconsistency and ensure that all replicas have the same value. This approach is at the heart of systems such as S3 in Amazon's Web Services. Eventual consistency is often achieved using techniques such as gossip and anti-entropy. Unfortunately, although the database will eventually converge, it is difficult to predict which value it will converge to. Since there is no global clock serializing all updates, the database cannot easily know if Alice's last status update was "Busy" or "Off the phone," and thus may end up converging the record to "Busy." Just when Alice is ready to chat with her friends, all of them think that she is busy, and this anomaly persists until Alice changes her status again.

Our Approach

We have struck a middle ground between strong consistency (such as ACID transactions) with its scalability limitations, and weaker forms of consistency (such as best effort or eventual consistency) with their anomalies. Our approach is timeline consistency: all replicas will go through the same timeline of updates, and the order of updates is equivalent to the order in which they were made to the database. This timeline is shown in Figure 4-1. Thus, the database will converge to the same value at all replicas, and that value will be the latest update made by the application.

FIGURE 4-1. Timeline of updates to Alice's status.

Copyrighted material

Timeline consistency is implemented by having a master copy where all the updates are made, with the changes later propagated to other copies asynchronously. This master copy serializes the updates and ensures that each update is assigned a sequence number. The order of sequence numbers is the order in which updates should be applied at all replicas, even if there are transient failures or misorderings in the asynchronous propagation of updates. We have chosen to have a master copy per record since many Yahoo! applications rely on a single table in which different records correspond to different users, each with distinct usage patterns. It is possible, of course, to choose other granularities for mastership, such as a master per partition (e.g., based on a key) of records.

Even in a single table, different records may have master copies located in different servers. In our example, Alice, who lives on the west coast, has a record that is mastered there, whereas her friend Bob, who lives in Singapore, has his record mastered in the Asian replica. The mastership of the record is stored as a metadata field in the record itself, as shown in Table 4-6.

TABLE 4-6. Profile table with mastership and version metadata

Username	_MASTER	_VERSION	FullName	...
Alice	West	32	Alice Smith	...
Bob	Asia	18	Bob Jones	...
Charles	East	15	Charles Adams	...
...				

Of course, a master copy seems at odds with our principle that only cheap operations should be done synchronously. If Alice travels to New York and updates her status from there, she must wait for her update operation to be forwarded to the west coast, since her profile record is mastered there; such high-latency cross-continental operations are what we are trying to minimize. Such cross-colo writes do occur occasionally, because of shifting usage patterns (e.g., Alice's travel), but they are rare. We analyzed updates to Yahoo!'s user database and found that 85% of the time, record updates were made to the colo containing the master copy. Of course, Alice may move to the east coast or to Europe, and then her writes will no longer be local, as the master copy for her record is still on the west coast. Our system tracks where the updates for a record are originating, and moves mastership to reflect such long-standing shifts in access patterns, in order to ensure that most writes continue to be local. (We discuss mastership in more detail in the next section.)

When an application reads a record, it typically reads the local replica. Unless that replica is marked as the master copy, it may be stale. The application knows that the record instance is some consistent version from the timeline, but there is no way for the application to know from the record itself whether it is the most recent version. If the application absolutely must have the most recent version, we allow it to request an *up-to-date read*; this request is forwarded to the master to get the latest copy of the record. An up-to-date read is expensive, but the common case of reading the local (possibly stale) replica is cheap, again in line with our design principles. Luckily, web applications are often tolerant of stale data. If Alice updates her status and her friend Bob does not see the new status right away, it is acceptable, as long as Bob sees the new status shortly thereafter.

Copyrighted material

Another kind of read that the application can perform is a *critical read*, to make sure that data only moves forward in time from the user's perspective. Consider a case where Alice changes her avatar (a picture representing the user). Bob may look at Alice's profile page (resulting in a read from the database) and see the new avatar. Then, Bob may refresh the page, and due to a network problem, be redirected to a replica that has not yet seen Alice's avatar update. The result is that Bob will see an older version of the data than the version he just saw. To avoid these anomalies for applications that want to do so, the database returns a version number along with the record for a read call. This version number can be stored in Bob's session state or in a cookie in his browser. If he refreshes Alice's profile page, the previously read version number can be sent along with his request, and the database will ensure that a record that is no older than that version is returned. This may require forwarding to the master copy. A read that specifies the version number is called a "critical read," and any replica with that version, or a newer version, is an acceptable result. This technique is especially helpful for users that update and then read the database. Consider Alice herself: after she updates her avatar, she will become confused if we show her any page with her old avatar. Therefore, when she takes an action that updates the database (like changing her avatar), the application can use the critical read mechanism to ensure that we never show her older data.

We also support a *test-and-set* operation that makes a write conditional upon the read version being the same as some previously seen version (whose version number is passed in as a parameter to the test-and-set request). In terms of conventional database systems, this provides a special case of ACID transactions, limited to a single record, using optimistic concurrency control.

More on mastership

We employ various techniques to ensure that read and write operations go on smoothly and with low latency, even in the presence of workload changes and failures.

For example, as we mentioned earlier, the system implements *record-level* mastership. If too many writes to the record are originating from a data center other than the current master, the mastership of the record is promptly transferred to that data center, and subsequent writes are done locally there. Moreover, transferring mastership is a cheap operation and happens automatically, thereby allowing the system to adapt quickly to workload changes.

We also implement a mechanism that allows reads and writes to continue without interruption, even during storage unit failures. When a storage unit fails, an *override* is issued (manually or automatically) for that storage unit, signifying that another data center can now accept writes on behalf of the failed storage unit (for records previously mastered at the failed storage unit). We take steps (details omitted here) to ensure that this override is properly sequenced with respect to the updates done at the failed storage unit. This is done to guarantee that timeline consistency is still preserved when the other data center starts accepting updates on behalf of the failed storage unit.

Copyrighted material

In PNUTS, all read and write requests go through a routing layer that directs them to the appropriate copy (possibly the master) of the record. This level of indirection is a key to how we provide uninterrupted system availability. Even when a storage unit has failed and its data is recovered on to another storage unit, or record masters are moved to reflect usage patterns, these changes are transparent to applications, which still continue to connect to routers and enjoy uninterrupted system availability, with requests seamlessly routed to the appropriate location.

Supporting ordered data

Our system is architected to support both hash-partitioned and range-partitioned data. We call the hash version of our database *YDHT*, for Yahoo! Distributed Hash Table, and the ordered version is called *YDOT*, for Yahoo! Distributed Ordered Table. Most of the system is agnostic to how the data is organized. However, there is one important issue that is sensitive to physical data organization. In particular, hash-organized data tends to spread load out among servers very evenly. If data is ordered, portions of the key space that are more popular will cause hotspots. For example, if status updates are ordered by time, the most recent updates will be of most interest to users, and the server with the data partition at the end of the time range will be the most loaded. We cannot allow hotspots to persist without compromising system scale-out.

Logically ordered data is actually stored in partitions of physically contiguous records, but with partitions arranged without regard to order, possibly across physical servers. We can address the hotspot issue by moving partitions dynamically in response to load. If a few hot partitions are on the same server, we can move them to servers that are less loaded. Moreover, we can also dynamically split partitions, so that the load on a particularly hot single partition can be divided amongst several servers.* This movement and splitting of partitions across storage units is distinct from the mechanism mentioned previously for changing the location of the master copy of a record: in this case, changing the record master affects the latency of updates that originate at a server, but does not in general reduce the cumulative read and write workload on a given partition of records. A particular special case that requires splitting and moving partitions is when we want to update or insert a large number of records. In that case, if we are not careful we can create a sever load imbalance by sending large batches of updates to the same few servers. Thus, it is necessary to understand something about how the updates are distributed in the key space, and if necessary, preemptively split and move partitions to prepare for the upcoming onslaught of updates (Silberstein et al. 2008).

We insulate applications from the details of the physical data organization. For single record reads and writes, the use of a routing layer shields applications from the effects of partition movement and splitting. For range scans, we need to provide a further abstraction: imagine

* The observant reader may have noticed that if all updates affect the partition containing the end of the time range, splitting this partition will not solve the problem, and some measure such as sorting by a composite key, e.g., user and time, is required.

Copyrighted material

that we want to scan all registered users whose age is between 21 and 30. Answering this query may mean scanning a partition with several thousand records on one server, then a second partition on another server, and so on. Each partition of several thousand records can be scanned quickly, since they are sequentially ordered on disk. We do not want the application to know that we might be moving or splitting partitions behind the scenes. A good way to do this is to extend the *iterator* concept: when an application is scanning, we return a group of records, and then allow the application to come back when it is ready to ask for the next group. Thus, when the application has completed one batch and has asked for more, we can switch them to a new storage server that has the partition with the next group of records.

Trading off consistency for availability

Timeline consistency handles the common case efficiently and with clean semantics, but it is not perfect. Occasionally, an entire datacenter will go down (e.g., if the power is cut) or become unreachable (e.g., if the network cable is cut), and then any records mastered in that datacenter will become unwriteable. This scenario exposes the known trade-off between consistency, availability, and partition tolerance: only two of those three properties can be guaranteed at all times. Since our database is global, partitions will happen and cannot cause an outage, and thus in reality we only have a choice between consistency and availability. If a datacenter goes offline, possibly with some new updates that have not yet been propagated to other replicas, we can either preserve consistency by disallowing updates until the datacenter comes back, or we can preserve availability by violating timeline consistency and allowing some updates to be applied to a nonmaster record.

Our system gives the application the ability to make this choice on a per-table basis. If the application has chosen availability over consistency for a particular table, and a datacenter goes offline, the system temporarily transfers mastership of any unreachable records in that table. This decision effectively forks the timeline to favor availability. An example is shown in Figure 4-2. After the lost colo is restored, the system automatically reconciles any records that have had conflicting updates, and notifies the application of these conflicts. The reconciliation ensures that the database converges to the same value everywhere, even if the timeline is not preserved. On the other hand, if the application has chosen consistency over availability, mastership is not transferred and the timeline is preserved, but some writes will fail.

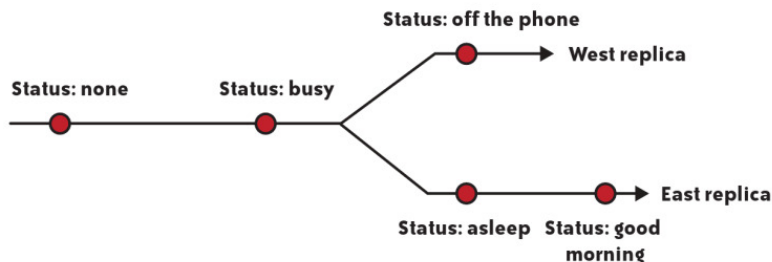


FIGURE 4-2. The west datacenter is offline, so the update timeline forks.

Copyrighted material

For certain operations, this trade-off between consistency and availability can be easier to manage. For example, imagine that an application wants to include polls, where users vote on various questions (like "What is your favorite color?") and the poll results are stored as counters in our database. Counter operations (like increment) are commutative, and can therefore be applied even to the nonmaster copy without breaking timeline consistency. Normally our replication mechanism transfers the new version of the record between replicas, but for commutative operations we would actually have to transmit the operation (e.g., increment). Then, whenever the master received the operation (either during normal operation or after a datacenter failure), it could apply it without worrying about whether it is out of order. The one restriction in this scheme is that we cannot mix commutative and noncommutative operations: setting the value of the counter at any time after the record inserted is forbidden, since we do not know how to properly order an increment and an overwrite of the value.

Another extension to our approach is to allow updates to multiple records. Many web workloads involve updates to a single record at a time, which is why we focused on timeline consistency at a per-record basis. However, it is occasionally desirable to update multiple records. For example, in our social networking application we might have binary friend links: if Alice and Bob are friends, then Alice appears in Bob's friend list and Bob appears in Alice's. When Alice and Bob become friends, we thus need to update two records. Because we do not provide ACID transactions, we cannot guarantee this update is atomic. However, we can provide *bundled writes*: with one call to the database, the application can request both writes, and the database will ensure that both writes eventually occur. To accomplish this, we log the requested writes, and the system retries the writes until they succeed. This approach preserves per-record timeline consistency, and since the retries can be asynchronous, preserves our performance goals.

In summary, timeline consistency provides a simple semantics for how record updates are propagated, and flexibility in how applications can trade-off read latency for currency. However, it does not support general ACID transactions—in particular, transactions that read and write multiple records.

Complex Queries

As web applications become more complex and interesting, they need to retrieve and combine information from the database in new and different ways. Next, we examine how to support those queries at a massive scale.

The Challenge

Our system is optimized for queries that touch one or just a few records. In particular, we can look up records by primary key; once we know Alice's username, it is straightforward to determine which partition contains her profile record and read it while loading her page. Also, our system can store data as hash-partitioned or range-partitioned tables. For range-partitioned tables, we can conduct range scans over ordered ranges of primary keys.

Copyrighted material

For example, we might store Alice's friends list by having one record per connection, where the primary key of each connection is the pair of user IDs for Alice and the friend (Table 4-7).

TABLE 4-7. Friends table

User1	User2	...
Alice	Bob	...
Alice	Charles	...
Alice	Dave	...
...		

In a range-partitioned table, all of the records prefixed with "Alice" will be clustered, and a short-range scan will be able to pick them up.

Now imagine that we want to add another feature to our social network site. Users can post photos and then comment on one another's photos. Alice might comment on Bob's photo, Charles's photo, and Dave's photo. When we display a photo, we want to show the set of comments associated with that photo. We also want to show Alice the set of comments she has made on other people's photos. We specify the primary key of the comments table as (`PhotoID,CommentID`) and store it as an ordered YDOT table (Table 4-8), so that all comments for the same photo are clustered and can be retrieved by a range query.

TABLE 4-8. Photo comments table

PhotoID	CommentID	Comment	Commenter
Photo123	18	Cool	Mary
Photo123	22	Pretty	Alice
Photo123	29	Interesting	Charles
...			

How can we collect the set of comments that Alice has made? We have to perform a join between Alice's profile record (which contains her username as a key) and the comment records (which have Alice's username as a foreign key). Because of our scale-out architecture, data is partitioned across many servers, so computing the join can require accessing many servers. This expensive operation drives up the latency of requests, both because multiple servers must be contacted and because a single query generates a great deal of server load (which slows down other requests).

Another type of query that can be expensive to compute in a scale-out system is group-by-aggregate queries. Imagine that users specify hobbies, and we want to count the number of users who have each hobby so that we can show Alice which hobbies are most popular. Such a query requires scanning all of the data and maintaining counts. The table scan will place prohibitive load on the system and certainly cannot be done synchronously, as Alice's page will take forever to load.

These examples show that while point lookups and range scans can be executed quickly, more expensive join and aggregation queries cannot be executed synchronously.

Copyrighted material

Our Approach

Our key principle for handling expensive operations is to do them asynchronously, but expensive queries cannot really be handled this way; we do not want to make Alice come back repeatedly to check whether the asynchronous query collecting all of her comments has completed.

Materialized views (Agarawal et al. 2009) can, however, be maintained asynchronously, and when Alice logs in she can quickly (and synchronously) query the view.* Although an asynchronously maintained view can be stale compared to the base data, the application already must be built to cope with stale replicas, so dealing with stale view data is usually acceptable. In fact, we treat a materialized view as a special kind of replica that both replicates and transforms data. By using the same mechanism that updates replicas to also update views, we ensure that views have similar reliability and consistency guarantees as replicated base data, without having to design and implement a second mechanism.

Even though view maintenance is done in the background, we still want to make it cheap. If view maintenance takes too many system resources, it will either disrupt synchronous read and write requests (adding latency to every query), or we will have to throttle it to run slowly, at which point the view will be so stale as to possibly be unusable. Thus, we have to find ways to make view maintenance efficient. Consider the earlier example where we want to show Alice all of the comments she has made on other people's photos. We will create a materialized view where comment data is reorganized to be clustered by the foreign key (username of the commenter) rather than the primary key. Then, all of the comments made by Alice will be clustered together. We can also place Alice's profile record in the view, keyed by her username, so that her profile and her comments are clustered. Computing the key/foreign key join is as easy as scanning the set of view records prefixed with "Alice", and then joining them. The result is shown in Table 4-9.

TABLE 4-9. Co-clustering joining profile and comment records

Alice	West	32	Alice Smith	...	← Profile record
Alice	Photo123	22	Pretty	...	← Comment records
Alice	Photo203	43	Nice	...	↓
Alice	Photo418	33	OK	...	
...					

Note that we do not prejoin the profile and comment records in the view. By merely co-locating records that would join, we make join maintenance cheap: whenever there is an update to a base record, we only have to update a single view record, even if that view record would join with multiple other records.

* Materialized views are not currently in the production version of the system.

Copyrighted material

How can we store profile and comment records in the same table? In a traditional database it would be difficult, since the two records have different schemas. However, a core feature of PNUTS is its ability to represent flexible schemas. Different records in the same table can have different sets of attributes. This feature is very useful in web applications since web data is often sparse; a database of items for sale will have different attributes (e.g., color, weight, RAM, flavor) depending on what kind of item it is. It turns out that flexible schemas are also key to implementing materialized join views so that we can colocate joining records from different tables.

The asynchronous view approach is useful for helping to answer other kinds of queries as well. A group-by-aggregation query can be effectively answered by a materialized view that has pregrouped, and maybe even preaggregated, the data. There are even "simple" queries, such as a selection over a nonprimary key attribute, that can be most effectively answered by a materialized view. Consider a query for users who live in Sunnyvale, California. Since our user table is keyed by username, this query normally requires an expensive table scan. However, we can use the materialized view mechanism to build a secondary index over the "location" field of the table, store the index in an ordered YDOT table, and then conduct a range scan over the "Sunnyvale, California" index records to answer our query (Table 4-10).

TABLE 4-10. Location index

Location	Username
Sunnyvale, CA	Alice
Sunnyvale, CA	Mary
Sunnyvale, CA	Steve
Sunnyvale, CA	Zach
...	

As with materialized views in other systems, we can create them effectively only if we know in advance what kinds of queries to expect. Luckily, in web-serving workloads, the queries are usually templates known in advance with specific parameters (such as the location or username) bound at runtime. As such, application developers know in advance which queries are complex enough to require materializing a view. To ask ad hoc queries over data stored in PNUTS, developers have to use our plug-ins to pull data out of our system into a compute grid running Hadoop, the open source implementation of MapReduce.

Once we have a few different mechanisms for handling complex queries, it will be useful to implement a query planner to help execute queries effectively. A planner helps remove some of the burden from the application developer, who can write declarative queries without worrying too much about how they will be executed. However, an effective query planner at our scale will require sophisticated statistics collection, load monitoring, network monitoring, and a variety of other mechanisms to make sure the planner has enough information about all the possible bottlenecks in the system to make the most effective query plan.

Copyrighted material

Comparison with Other Systems

When we began thinking about PNUTS, two other massive scale database systems from Google and Amazon had recently been announced, and a third from Microsoft would later be made public. As we developed our designs, we examined these other systems carefully to see whether some or all of their ideas could be useful to us. Some of the ideas from these systems influenced us, but we decided to build a new system with an architecture that was different in many ways. We now look at each of these systems and discuss why we decided to depart from their design principles.

Google's BigTable

BigTable (Chang et al. 2006) is a system designed to support many of Google's web applications. The system is based on horizontally partitioning a "big table" into many smaller tablets, and scattering those tablets across servers. This basic approach to scalability, as well as features such as flexible schema and ordered storage, are similar to the approach we took. However, there were several design decisions where we diverged from BigTable.

The first major difference was in our approach to replication. BigTable is built on top of the Google File System (GFS; Ghemawat et al. 2003), and GFS handles the replication of data by synchronously updating three copies of the data on three different servers. This approach works well in a single colo, where interserver latencies are low. However, synchronously updating servers in three different, widely dispersed colos is too expensive; Alice might wait a long time for her status to be updated, especially if her friends access a datacenter with a poor connection to the Internet backbone. To support cross-colo replication, we developed the timeline consistency model, and the associated mechanisms for mastership, load balancing, and failure handling.

We also decided not to enforce the separation between database server and filesystem that is enforced between BigTable and GFS. GFS was originally designed and optimized for scan-oriented workloads of large files (for example, for MapReduce). BigTable uses GFS by keeping a version history of each record, compacted into a file format called *SSTables* to save space. This means that on record reads and updates, the data must be decoded and encoded into this compressed format. Moreover, the scan-oriented nature of GFS makes BigTable useful for column-oriented scans (such as "retrieve all the locations of all the users"). In contrast, our primary workload is to read or update a single version of a single record or a small range of records. Thus, we store data on disk as complete records organized into a B-tree. This approach is optimized for quickly locating, and updating in-place, individual records identified by primary key.

PNUTS differs from BigTable in other ways as well. For example, we support multiple tables for an application, instead of one large table, and we support hash as well as ordered tables. A follow on to BigTable, called MegaStore (Furman et al. 2008), adds transactions, indexes, and a richer API, but still follows the basic architectural tenets of BigTable.

Copyrighted material

Amazon's Dynamo

Dynamo (DeCandia et al. 2007) is one of the systems Amazon has built recently for large-scale data workloads, and is the one most closely aligned with our goals of a highly available, massive scale structured record store. (Records in Dynamo are referred to as objects.) Dynamo provides write availability by allowing applications to write to any replica, and lazily propagating those updates to other replicas via a gossip protocol (explained next).

The decision to lazily propagate updates to deal with slow and failure-prone networks matches our own; however, our mechanism for replication is quite different. In a gossip protocol, an update is propagated to randomly chosen replicas, which in turn propagates it to other randomly chosen replicas. This randomness is essential to the probabilistic guarantees offered by the protocol, which ensures that most replicas are updated relatively quickly. In our setting, however, randomness is decidedly suboptimal. Consider an update Alice makes to her status in a colo on the west coast of the U.S. Under gossip, this update may be randomly propagated to a replica in Singapore, which then randomly propagates the update to a replica in Texas, which then randomly propagates the update to a replica in Tokyo. The update has crossed the Pacific Ocean three times, whereas a more deterministic approach could conserve scarce trans-Pacific backbone bandwidth and transfer it (and other updates) only once. Moreover, gossip requires the replica propagating the update to know which servers in which other colos have replicas, which makes it hard to move data between servers for load balancing or recovery.

Another key difference with Dynamo is the consistency protocol. Gossip lends itself to an eventual consistency model: all data replicas will eventually match, but in the interim, while updates are propagating, replicas can be inconsistent. In particular, replicas can have a state that is later deemed "invalid." Consider, for example, Alice, who updates her status from "Sleeping" to "Busy" and then updates her location from "Home" to "Work." Because of the order of updates, the only valid states of the record (from Alice's perspective, which is what matters) are `(Sleeping,Home)`, `(Busy,Home)`, and `(Busy,Work)`. Under eventual consistency, if the two updates are made at different replicas, some replicas might receive the update to "Work" first, meaning that those replicas show a state of `(Sleeping,Work)` temporarily. If Alice's boss sees this status, Alice might be in trouble! Applications that rely on the application of multiple updates to a record in the proper order need a stronger guarantee than eventual consistency. Although our timeline consistency model allows replicas to be stale, even stale replicas have a consistent version that reflects the proper update ordering.

There are various other differences with Dynamo: Dynamo provides only a hash table and not an ordered table, and we have opted for a more flexible mapping of data to servers in order to improve load balancing and recovery (especially for ordered tables, which might have unpredictable hot spots). Amazon also provides other storage systems besides Dynamo: S3 for storing blobs of data, and SimpleDB for executing queries over structured, indexed data. Although SimpleDB provides a richer API, it requires that the application come up with a partitioning of the data such that each partition is within a fixed size limit. Thus, data growth within a partition is restricted.

Copyrighted material

Microsoft Azure SDS

Microsoft has built a massive scale version of SQL Server (called SQL Data Services or SDS) as part of its Azure services offering (*http://hadoop.apache.org*). Again, the focus is on scalability through horizontal partitioning. A nice feature of SDS is the enhanced query capabilities made available by extensively indexing data and providing SQL Server as the query-processing engine. However, SDS achieves this query expressiveness by rigidly enforcing partitioning: applications create their own partitions and cannot easily repartition data. Thus, although you can ask expressive queries over a partition, if a partition grows or becomes hot, the system cannot easily or automatically relieve the hotspot by splitting the partition. Our decision to hide partitioning behind the abstraction of a table allows us to make and change partitioning decisions for load and recovery reasons. While this means that our query model is less expressive (since we do not support complex queries which cross partitions), we are continuing to look at ways to enhance our query functionality (for example, through views, as described earlier).

Another difference with SDS is that PNUTS has geographic replication built in as a first-class feature of the system. In at least the first release of SDS, the workload is expected to live within a single datacenter, and remote copies are only used in case of a total failure of the primary replica. We want Alice's friends in Singapore, Berlin, and Rio de Janeiro to have their own local, first-class copies of Alice's updates.

Other Related Systems

A variety of other systems have been built by companies who have scalability and flexibility needs similar to ours. Facebook has built Cassandra (Lakshman et al. 2008), a peer-to-peer data store with a BigTable-like data model but built on a Dynamo-like infrastructure. Consequently, Cassandra provides only eventual consistency.

Sharded databases (such as the MySQL sharding approach used by Flickr [Pattishall] and Facebook [Sobel 2008]) provide scalability by partitioning the data across many servers; however, sharding systems do not typically provide as much flexibility for scaling or globally replicating data as we desire. Data must be prepartitioned, just like in SimpleDB. Also, only one of the replicas can be the master and accept writes. In PNUTS, all replicas in different data centers can accept writes (although for different records).

Other Systems at Yahoo!

PNUTS is one of several cloud systems that are being built at Yahoo!. Two other components of the cloud are also targeted at data management, although they focus on a different set of problems than PNUTS. Hadoop (*http://hadoop.apache.org*), an open source implementation of the MapReduce framework (Dean and Ghemawat 2007), provides massively parallel analytical processing over large datafiles. Hadoop includes a filesystem,

Copyrighted material

HDFS, which is optimized for scans, since MapReduce jobs are primarily scan-oriented workloads. In contrast, PNUTS is focused on reads and writes of individual records. Another system is MObStor, which is designed to store and serve massive objects such as images or video. MObStor's goal is to provide low-latency retrieval and inexpensive storage for objects that do not change. Since many applications need a combination of record storage, data analysis, and object retrieval, we are working on ways to seamlessly integrate the three systems. A survey of our efforts to integrate these systems into a comprehensive cloud is at (Cooper et al. 2009).

Conclusion

When we embarked on the PNUTS project, we had in mind a system that could seamlessly scale to thousands of servers and multiple continents. Building such a system required more than clever engineering; it required us to reopen many settled debates in the database field. Although it was a relatively easy decision to jettison ACID, we soon realized we had to develop something to replace it, and thus developed the timeline consistency model. Although the model is relatively simple by design, handling complex corner cases, developing an efficient implementation mechanism, and mapping application use cases to the model required deep thinking and many iterations. Another point to note is that at first our customers and we were relatively blasé about restricting ourselves to a simple query language. However, as developers began trying to build real applications on top of PNUTS, we realized that the small fraction of the query workload that was more complex than we could handle would be a major stumbling block to the system's adoption. If we did not develop a mechanism to handle these queries, developers would have to resort to complicated workarounds, either implementing expensive operations (such as nested loop joins) in their application logic or frequently exporting data to external indexes to support their workload.

The field is in the early stages of cloud data management, and this is reflected in the many alternative system designs being built and deployed. We hope the ideas embodied in the PNUTS system can help us get closer to the goal of easily manageable, broadly applicable, multitenanted cloud database systems that provide applications with elastic, efficient, globally available, and extremely robust data backends.

Acknowledgments

PNUTS is a collaborative effort among many different people at Yahoo!. Leading the engineering effort are P.P.S. Narayan and Chuck Neerdaels. Other researchers on the project include Adam Silberstein and Rodrigo Fonseca. Brad McMillen and Pat Quaid help with the architecture of PNUTS and its place in Yahoo!'s cloud offerings. Other designers and developers of the system have included Phil Bohannon, Ramana Yerneni, Daniel Weaver, Michael Bigby, Nicholas Puz, Hans-Arno Jacobsen, Bryan Call, and Andrew Feng.

Copyrighted material

References

Azure Services Platform. *http://www.microsoft.com/azure/*.

Hadoop. *http://hadoop.apache.org*.

Agrawal, P., A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. "Asynchronous View Maintenance for VLSD Databases." In *SIGMOD*, 2009.

Chang, F. et al. "Bigtable: A distributed storage system for structured data." In *OSDI*, 2006.

Cooper, B. F., E. Baldeschwieler, R. Fonseca, J. J. Kistler, P.P.S. Narayan, Chuck Neerdaels, Toby Negrin, Raghu Ramakrishnan, Adam Silberstein, Utkarsh Srivastava, and Raymie Stata. "Building a cloud for Yahoo!" *IEEE Data Engineering Bulletin*, 32(1): 36–43, 2009.

Cooper, B. F., R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform." In *VLDB*, 2008.

Dean, J. and S. Ghemawat. "MapReduce: Simplified data processing on large clusters." In *OSDI*, 2004.

DeCandia, G. et al. "Dynamo: Amazon's highly available key-value store." In *SOSP*, 2007.

Furman, J. J., J. S. Karlsson, J.-M. Leon, A. Lloyd, S. Newman and P. Zeyliger. "Megastore: A Scalable Data System for User Facing Applications." In *SIGMOD*, 2008.

Ghemawat, S., H. Gobioff, and S.-T. Leung. "The Google File System." In *SOSP*, 2003.

Lakshman, A., P. Malik, and K. Ranganathan. "Cassandra: A Structured Storage System on a P2P Network." In *SIGMOD*, 2008.

Pattishall, D. V. "Federation at Flickr: Doing Billions of Queries Per Day." *http://www.scribd.com/doc/2592098/DVPmysqlucFederation-at-Flickr-Doing-Billions-of-Queries-Per-Day*.

Ramakrishnan, R. and J. Gehrke. Database Management Systems. McGraw-Hill, New York, NY, 2002.

Silberstein, A., B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. "Efficient bulk insertion into a distributed ordered table." In *SIGMOD*, 2008.

Sobel, J. "Scaling out." Facebook Engineering Blog, August 2008.

Copyrighted material

CHAPTER FIVE

Information Platforms and the Rise of the Data Scientist

Jeff Hammerbacher

Libraries and Brains

At the age of 17, I was fired from my job as a cashier at Scott's grocery store in Fort Wayne, Indiana. With only two months remaining before my freshman year of college, I saw in my unemployment an opportunity. Instead of telling my parents that I had been fired, I continued to leave the house every afternoon in my cashier's outfit: black pants, black shoes, white shirt, and smock. To my parents, I looked ready for some serious coupon scanning; in reality, I was pulling 10-hour shifts reading at the public library.

All reasonably curious people wonder how their brain works. At 17, I was unreasonably curious. I used my time at the library to learn about how brains work, how they break, and how they are rebuilt. In addition to keeping us balanced, regulating our body temperature, and making sure we blink our eyelids together every now and again, our brains ingest, process, and generate massive amounts of information. We construct unconscious responses to our immediate environment, short-term plans for locution and limb placement, and long-term plans for mate selection and education. What makes brains interesting is not just their ability to generate reactions to sensory data, but their role as repository of information for both plan generation and the creation of new information. I wanted to learn how that worked.

Copyrighted material

One thing about brains, though: they remain stubbornly housed within a single body. To collect information from many brains, we build libraries. The field of library science has evolved numerous techniques for herding the information stored in libraries to enable future consumption; a fun read on the topic is Alex Wright's *Glut* (Joseph Henry Press). In addition to housing information for future retrieval, libraries play a critical role in the creation of new information. As philosopher Daniel Dennett puts it, "a scholar is just a library's way of making another library."

Libraries and brains are two examples of Information Platforms. They are the locus of their organization's efforts to ingest, process, and generate information, and they serve to accelerate the process of learning from empirical data. When I joined Facebook in 2006, I naturally started to build an Information Platform. Because of the tremendous growth in the number of users on Facebook, the system our team built ended up managing several petabytes of data. In this chapter, I'll recount the challenges faced in building out Facebook's Information Platform and the lessons learned while constructing our solution from open source software. I'll also try to outline the critical role of the Data Scientist in using that information to build data-intensive products and services and helping the organization formulate and accomplish goals. Along the way, I'll recount how some other businesses have approached the problem of building Information Platforms over the decades.

Before we get started, I should point out that my clever plan to visit the library instead of the grocery store did not work out as intended. After a few blissful days of reading, I came out of the library one evening and couldn't locate my car. It was not uncommon for me to lose my car at the time, but the lot was empty, so I knew something was up. It turns out that my mom had figured out my scheme and gotten my car towed. During the long walk home, I internalized an important lesson: regard your own solutions with skepticism. Also, don't try to outsmart your mother.

Facebook Becomes Self-Aware

In September 2005, Facebook opened to non-college students for the first time and allowed high school students to register for accounts. Loyal users were outraged, but the Facebook team felt that it was the right direction for the site. How could it produce evidence to justify its position?

In addition, Facebook had saturated the student population at nearly all of the colleges where it was available, but there were still some colleges where the product had never taken off. What distinguished these laggard networks from their more successful peers, and what could be done to stimulate their success?

When I interviewed at Facebook in February 2006, they were actively looking to answer these questions. I studied mathematics in college and had been working for a nearly a year on Wall Street, building models to forecast interest rates, price complex derivatives, and hedge pools of mortgages; I had some experience coding and a dismal GPA. Despite my potentially suboptimal background, Facebook made me an offer to join as a Research Scientist.

Copyrighted material

Around the same time, Facebook hired a Director of Reporting and Analytics. The director had far more experience in the problem domain than me; together with a third engineer, we set about building an infrastructure for data collection and storage that would allow us to answer these questions about our product.

Our first attempt at an offline repository of information involved a Python script for farming queries out to Facebook's tier of MySQL servers and a daemon process, written in C++, for processing our event logs in real time. When the scripts worked as planned, we collected about 10 gigabytes a day. I later learned that this aspect of our system is commonly termed the "ETL" process, for "Extract, Transform, and Load."

Once our Python scripts and C++ daemon had siphoned the data from Facebook's source systems, we stuffed the data into a MySQL database for offline querying. We also had some scripts and queries that ran over the data once it landed in MySQL to aggregate it into more useful representations. It turns out that this offline database for decision support is better known as a "Data Warehouse."

Finally, we had a simple PHP script to pull data from the offline MySQL database and display summaries of the information we had collected to internal users. For the first time, we were able to answer some important questions about the impact of certain site features on user activity. Early analyses looked at maximizing growth through several channels: the layout of the default page for logged-out users, the source of invitations, and the design of the email contact importer. In addition to analyses, we started to build simple products using historical data, including an internal project to aggregate features of sponsored group members that proved popular with brand advertisers.

I didn't realize it at the time, but with our ETL framework, Data Warehouse, and internal dashboard, we had built a simple "Business Intelligence" system.

A Business Intelligence System

In a 1958 paper in the *IBM Systems Journal*, Hans Peter Luhn describes a system for "selective dissemination" of documents to "action points" based on the "interest profiles" of the individual action points. The author demonstrates shocking prescience. The title of the paper is "A Business Intelligence System," and it appears to be the first use of the term "Business Intelligence" in its modern context.

In addition to the dissemination of information in real time, the system was to allow for "information retrieval"—search—to be conducted over the entire document collection. Luhn's emphasis on action points focuses the role of information processing on goal completion. In other words, it's not enough to just collect and aggregate data; an organization must improve its capacity to complete critical tasks because of the insights gleaned from the data. He also proposes "reporters" to periodically sift the data and selectively move information to action points as needed.

Copyrighted material

The field of Business Intelligence has evolved over the five decades since Luhn's paper was published, and the term has come to be more closely associated with the management of structured data. Today, a typical business intelligence system consists of an ETL framework pulling data on a regular basis from an array of data sources into a Data Warehouse, on top of which sits a Business Intelligence tool used by business analysts to generate reports for internal consumption. How did we go from Luhn's vision to the current state of affairs?

E. F. Codd first proposed the relational model for data in 1970, and IBM had a working prototype of a relational database management system (RDBMS) by the mid-1970s. Building user-facing applications was greatly facilitated by the RDBMS, and by the early 1980s, their use was proliferating.

In 1983, Teradata sold the first relational database designed specifically for decision support to Wells Fargo. A few years later, in 1986, Ralph Kimball founded Red Brick Systems to build databases for the same market. Solutions were developed using Teradata and Red Brick's offerings, but it was not until 1991 that the first canonical text on data warehousing was published.

Bill Inmon's *Building the Data Warehouse* (Wiley) is a coherent treatise on data warehouse design and includes detailed recipes and best practices for building data warehouses. Inmon advocates constructing an enterprise data model after careful study of existing data sources and business goals.

In 1995, as Inmon's book grew in popularity and data warehouses proliferated inside enterprise data centers, The Data Warehouse Institute (TDWI) was formed. TDWI holds conferences and seminars and remains a critical force in articulating and spreading knowledge about data warehousing. That same year, data warehousing gained currency in academic circles when Stanford University launched its WHIPS research initiative.

A challenge to the Inmon orthodoxy came in 1996 when Ralph Kimball published *The Data Warehouse Toolkit* (Wiley). Kimball advocated a different route to data warehouse nirvana, beginning by throwing out the enterprise data model. Instead, Kimball argued that different business units should build their own data "marts," which could then be connected with a "bus." Further, instead of using a normalized data model, Kimball advocated the use of dimensional modeling, in which the relational data model was manhandled a bit to fit the particular workload seen by many data warehouse implementations.

As data warehouses grow over time, it is often the case that business analysts would like to manipulate a small subset of data quickly. Often this subset of data is parameterized by a few "dimensions." Building on these observations, the CUBE operator was introduced in 1997 by a group of Microsoft researchers, including Jim Gray. The new operator enabled fast querying of small, multidimensional data sets.

Both dimensional modeling and the CUBE operator were indications that, despite its success for building user-facing applications, the relational model might not be best for constructing an Information Platform. Further, the document and the action point, not the

Copyrighted material

table, were at the core of Luhn's proposal for a business intelligence system. On the other hand, an entire generation of engineers had significant expertise in building systems for relational data processing.

With a bit of history at our back, let's return to the challenges at Facebook.

The Death and Rebirth of a Data Warehouse

At Facebook, we were constantly loading more data into, and running more queries over, our MySQL data warehouse. Having only run queries over the databases that served the live site, we were all surprised at how long a query could run in our data warehouse. After some discussion with seasoned data warehousing veterans, I realized that it was normal to have queries running for hours and sometimes days, due to query complexity, massive data volumes, or both.

One day, as our database was nearing a terabyte in size, the `mysqld` daemon process came to a sudden halt. After some time spent on diagnostics, we tried to restart the database. Upon initiating the restart operation, we went home for the day.

When I returned to work the next morning, the database was still recovering. To get a consistent view of data that's being modified by many clients, a database server maintains a persistent list of all edits called the "redo log" or the "write-ahead log." If the database server is unceremoniously killed and restarted, it will reread the recent edits from the redo log to get back up to speed. Given the size of our data warehouse, the MySQL database had quite a bit of recovery to catch up on. It was three days before we had a working data warehouse again.

We made the decision at that point to move our data warehouse to Oracle, whose database software had better support for managing large data sets. We also purchased some expensive high-density storage and a powerful Sun server to run the new data warehouse.

During the transfer of our processes from MySQL to Oracle, I came to appreciate the differences between supposedly standard relational database implementations. The bulk import and export facilities of each database used completely different mechanisms. Further, the dialect of SQL supported by each was different enough to force us to rewrite many of our queries. Even worse, the Python client library for Oracle was unofficial and a bit buggy, so we had to contact the developer directly.

After a few weeks of elbow grease, we had the scripts rewritten to work on the new Oracle platform. Our nightly processes were running without problems, and we were excited to try out some of the tools from the Oracle ecosystem. In particular, Oracle had an ETL tool called Oracle Warehouse Builder (OWB) that we hoped could replace our handwritten Python scripts. Unfortunately, the software did not expect the sheer number of data sources we had to support: at the time, Facebook had tens of thousands of MySQL databases from which we collected data each night. Not even Oracle could help us tackle our scaling challenges on the ETL side, but we were happy to have a running data warehouse with a few terabytes of data.

Copyrighted material

And then we turned on clickstream logging: our first full day sent 400 gigabytes of unstructured data rushing over the bow of our Oracle database. Once again, we cast a skeptical eye on our data warehouse.

Beyond the Data Warehouse

According to IDC, the digital universe will expand to 1,800 exabytes by 2011. The vast majority of that data will not be managed by relational databases. There's an urgent need for data management systems that can extract information from unstructured data in concert with structured data, but there is little consensus on the way forward.

Natural language data in particular is abundant, rich with information, and poorly managed by a data warehouse. To manage natural language and other unstructured data, often captured in document repositories and voice recordings, organizations have looked beyond the offerings of data warehouse vendors to various new fields, including one known as enterprise search.

While most search companies built tools for navigating the collection of hyperlinked documents known as the World Wide Web, a few enterprise search companies chose to focus on managing internal document collections. Autonomy Corporation, founded in 1996 by Cambridge University researchers, leveraged Bayesian inference algorithms to facilitate the location of important documents. Fast Search and Transfer (FAST) was founded in 1997 in Norway with more straightforward keyword search and ranking at the heart of its technology. Two years later, Endeca was founded with a focus on navigating document collections using structured metadata, a technique known as "faceted search." Google, seeing an opportunity to leverage its expertise in the search domain, introduced an enterprise search appliance in 2000.

In a few short years, enterprise search has grown into a multibillion-dollar market segment that is almost totally separate from the data warehouse market. Endeca has some tools for more traditional business intelligence, and some database vendors have worked to introduce text mining capabilities into their systems, but a complete, integrated solution for structured and unstructured enterprise data management remains unrealized.

Both enterprise search and data warehousing are technical solutions to the larger problem of leveraging the information resources of an organization to improve performance. As far back as 1944, MIT professor Kurt Lewin proposed "action research" as a framework that uses "a spiral of steps, each of which is composed of a circle of planning, action, and fact-finding about the result of the action." A more modern approach to the same problem can be found in Peter Senge's "Learning Organization" concept, detailed in his book *The Fifth Discipline* (Broadway Business). Both management theories rely heavily upon an organization's ability to adapt its actions after reflecting upon information collected from previous actions. From this perspective, an Information Platform is the infrastructure required by a Learning Organization to ingest, process, and generate the information necessary for implementing the action research spiral.

Copyrighted material