CHAPTER SIXTEEN

Reporting, Business Intelligence, and Dashboards

DATA ANALYSIS DOES NOT JUST CONSIST OF CRUNCHING NUMBERS. IT ALSO INCLUDES NAVIGATING THE CONTEXT and environment in which the need for data analysis arises. In this chapter and the next, we will look at two areas that often have a demand for data analysis and analytical modeling but that tend to be unfamiliar if you come from a technical background: in this chapter, we discuss business intelligence and corporate metrics; in the next chapter, financial calculations and business plans.

This material may seem a little out of place because it is largely not technical. But that is precisely why it is important to include this topic here: to a person with a technical background, this material is often totally new. Yet it is precisely in these areas that sound technical and analytical advice is often required: the primary consumers of these services are "business people," who may not have the necessary background and skills to make appropriate decisions without help. This places additional responsibility on the person working with the data to understand the problem domain thoroughly, in order to make suitable recommendations.

This is no joke. I have seen otherwise very smart people at high-quality companies completely botch business metrics programs simply because they lacked basic software engineering and math skills. As the person who (supposedly!) "understands data," I see it as part of my responsibility to understand what my clients actually want to *do* with the data—and advise them accordingly on the things they *should* be doing. But to do so effectively, it is not enough to understand the data—I also need to understand my clients.

That's the spirit in which these chapters are intended. The aim is to describe some of the ways that demand for data arises in a business environment, to highlight some of the traps for the unwary, and to give some advice on using data more successfully.

Business Intelligence

Businesses have been trying to make use of the data that they collect for years and, in the process, have accumulated a fair share of disappointments. I think we need to accept that the problem is hard: you need to find a way to represent, store, and make accessible a comprehensive view of all available data in such a way that is useful to anybody and for any purpose. That's just hard. In addition, to be comprehensive, such an initiative has to span the entire company (or at least a very large part of it), which brings with it a whole set of administrative and political problems.

This frustrating state of affairs has brought forth a number of attempts to solve what is essentially a conceptual and political problem using *technical* means. In particular, the large enterprise tool vendors saw (and see) this problem space as an opportunity!

The most recent iteration on this theme was data warehouses—that is, long-term, comprehensive data stores in which data is represented in a denormalized schema that is intended to be more general than the schema of the transactional databases and also easier to use for nontechnical users. Data is imported into the data warehouse from the transactional databases using so-called ETL (extraction, transformation, and load) processes.

Overall, there seems to be a feeling that data warehouses fell short of expectations for three reasons. First of all, since data warehouses are enterprise-wide, they respond slowly to changes in any one business unit. In particular, changes to the transactional data schema tend to propagate into the data warehouse at a glacial pace, if at all. The second reason is that accessing the data in the data warehouse never seems to be as convenient as it should be. The third and final reason is that doing something useful with the data (once obtained) turns out to be difficult—in part because the typical query interface is often clumsy and not designed for analytic work.

While data warehouses were the most recent iteration in the quest for making company data available and useful, the current trend goes by the name of *business intelligence*, or BI. The term is not new (Wikipedia tells me that it was first used in the 1950s), but only in the last one or two years have I seen the term used regularly.

The way I see it, business intelligence is an accessibility layer sitting on top of a data warehouse or similar data store, trying to make the underlying data more useful through better reporting, improved support for ad hoc data analysis, and even some attempts at canned predictive analytics.

Because it sits atop a database, all business intelligence stays squarely within the database camp; and what it aspires to do is constrained by what a database (or a database developer!) can do. The "analytics" capabilities consist mostly of various aggregate operations (sums, averages, and so on) that are typically supported by *OLAP* (Online

Analytical Processing) *cubes*. OLAP cubes are multi-dimensional contingency tables (*i.e.*, with more than two dimensions) that are precomputed and stored in the database and that allow for (relatively) quick summaries or projections along any of the axes. These "cubes" behave much like spreadsheets on steroids, which makes them familiar and accessible to the large number of people comfortable with spreadsheets and pivot tables.

In my experience, the database heritage (in contrast to a software engineering heritage) of BI has another consequence: the way people involved with business intelligence relate to it. While almost all software development has an element of *product* development to it, business intelligence often feels like *infrastructure* maintenance. And while the purpose of the former typically involves innovation and the development of new ways to please the customer, the latter tends to be more reactive and largely concerned with "keeping the trains on time." This is not necessarily a bad thing, as long as one pays attention to the difference in cultures.

What is the take away here? First of all, I think it is important to have realistic expectations: when it comes right down to it, business intelligence initiatives are mostly about better reporting. That is fine as far as it goes, but it does not require (or provide) much data analysis per se. The business users who are the typical customers of such projects usually don't need much help in defining the numbers they would like to see. There may be a need for help with visualization and overall user interface design, but the possibilities here tend to be mostly defined (and that means limited) by the set of tools being used.

More care needs to be taken when any of the "canned" analysis routines are being used that come bundled with many BI packages. Most (if not all) of these tools are freebies, thrown in by the vendor to pad the list of supported features, but they are likely to lack production strength and instead emphasize "ease of use." These tools will produce results, all right—but it will be our job to decide how *significant* and how *relevant* these results are.

We should first ask what these routines are actually doing "under the hood." For example, a clustering package may employ any one of a whole range of clustering algorithms (as we saw in Chapter 13) or even use a combination of algorithms together with various heuristics. Once we understand what the package does, we can then begin asking questions about the quality and, in particular, the significance of the results. Given that the routine is largely a black box to us, we will not have an intuitive sense regarding the extent of the region of validity of its results, for example. And because it is intended as an easy-to-use give away, it is not likely to have support for (or report at length about) nasty details such as confidence limits on the results. Finally, we should ask how relevant and useful these results are. Was there an original question that is being addressed—or was the answer mostly motivated by the ease with which it could be obtained?

One final observation: when there are no commercial tool vendors around, there is not much momentum for developing business intelligence implementations. Neither of the two major open source databases (MySQL and Postgres) has developed BI functionality or the kinds of ad hoc analytics interfaces that are typical of BI tools. (There are, however, a few open source projects that provide reporting and OLAP functionality.)

Reporting

The primary means by which data is used for "analysis" purposes in an enterprise environment is via reports. Whether we like it or not, much of "business intelligence" revolves around reporting, and "reporting" is usually a big part of what companies do with their data.

It is also one of the greatest sources of frustration. Given the ubiquity of reporting and the resources spent on it, one would think that the whole area would be pretty well figured out by now. But this is not so: in my experience, nobody seems to like what the reporting team is putting out—including the reporting team itself.

I have come to the conclusion that reporting, as currently understood and practiced, has it all wrong. Reporting is the one region of the software universe that has so far been barely touched by the notions of "agility" and "agile development." Reporting solutions are invariably big, bulky, and bureaucratic, slow to change, and awkward to use. Moreover, I think with regards to two specific issues they get it *exactly* wrong:

1. In an attempt to conserve resources, reporting solutions are often built generically: a single reporting system that supports all the needs of all the users. The reality, of course, is that the system does not serve the needs of *any* user (certainly not well), even as the overhead of the general-purpose architecture drives the cost through the roof.
2. Most reporting that I have seen confuses "up to date" with "real time." Data for reports is typically pulled in immediate response to a user's query, which ensures that the data is up to date but also (for many reports) that it will take a while before the report is available—often quite a while! I believe that this delay is the single greatest source of frustration with all reports, anywhere. For a user, it typically matters much more to get the data *right this minute* than to get it *up to this minute*!

Can we conceive of an alternative to the current style of reporting, one that actually delivers on its promise and is easy and fun to use? I think so (in fact, I have seen it in action), but first we need to slaughter a sacred cow: namely, that *one* reporting system should be able to handle all kinds of *different* requirements. In particular, I think it will be helpful to distinguish very clearly between *operational* and *representative* reports.

Representative reports are those intended for external users. Quarterly filings certainly fall into this category, as do reports the company may provide to its customers on various metrics. In short, anything that gets published.

Operational reports, in contrast, are those used by managers within the company to actually run the business. Such reports include information on the the number of orders shipped today, the size of the backlog, or the CPU loads of various servers.

These two report types have almost nothing in common! Operational reports need to be fast and convenient—little else matters. Representative reports need to be definitive and optically impressive. It is not realistic to expect a single reporting system to support both requirements simultaneously! I'd go further and say that the preparation of representative reports is always somewhat of a special operation and should be treated as such: "making it look good." If you have to do this a lot (*e.g.*, because you regularly send invoices to a large number of customers), then by all means automate the process—but don't kid yourself into thinking that this is still merely "reporting." (Billing is a *core* business activity for all service businesses!)

When it comes to operational reports, there are several ideas to consider:

Think "simple, fast, convenient." Reports should be simple to understand, quick (instantaneous) to run, and convenient to use. Convenience dictates that the users *must not* be required to fill in an input mask with various parameters. The most the user can be expected to do is to select one specific report from a fixed list of available ones.

Don't waste real estate. The whole point of having a report is the *data*. Don't waste space on other things, especially if they never change. I have seen reports in which fully one third of the screen was taken up by a header showing the company logo! In another case, a similar amount of space was taken up by an input mask. Column headers and explanations are another common culprit: once people have seen the report twice, they will know what the columns are. (You will still need headers, but they can be short.) Move explanatory material to a different location and provide a link to it. Remember: the reason people ran the report is to see the *data*.

Make reports easy to read. In particular, this means putting lots of data onto a single page that can be read by scrolling (instead of dividing the data across several pages that require reloading those pages). Use a large enough font and consider (gently!) highlighting every second line. Less is more.

Consider expert help for the visual design. Reports don't have to be ugly. It may be worth enlisting an expert to design and implement a report that *looks* pleasant and is easy to use. Good design will emphasize the content and avoid distracting embellishments. Developing good graphic designs is a specialized skill, and some people are simply better at this task than others. Remember: a report's ease of use is not an unnecessary detail but an essential quality!

Provide raw data, and let the user handle filtering and aggregation. This is a potentially radical idea: instead of providing a complicated input mask whereby the user has to specify a bunch of selection criteria and the columns to return, a report can simply return *everything* (within reason, of course) and leave it to the user to perform any desired filtering and aggregation. This idea is based on the realization that most people who use reports are going to be comfortable working with Excel (or an equivalent spreadsheet program). Hence, we can regard a report not as an end product but rather as a data feed for spreadsheets.

This approach has a number of advantages: it is simple, cheap, and flexible (because users are free to design their own reports). It also implies that the report needs to include additional columns, which are required for user-level filtering and aggregation.

Consider cached reports instead of real-time queries. Once the input mask has been removed, the content of a report is basically fixed. But once it is fixed, it can be run ahead of time and cached—which means that we can return the data to the user instantaneously. It also means that the database is hit only once no matter how often the report is viewed.

Find out what your users are doing with reports—and then try to provide it for them. I cannot tell how often I've witnessed the following scenario. The reporting team spends significant time and effort worrying about the details and layout of its reports. But a few doors down the hall, the first thing that the report's actual users do is cut-and-paste the results from the reporting system and import them into, yes, Excel. And then they often spend a lot of time manually editing and formatting the results so that they reflect the information that the users actually need. This occurs *every day* (or every week, or every hour—each time the report is accessed).

These edits are often painfully simple: the users need the report sorted on some numerical column, but this is impossible because the entry in that column is text: "Quantity 17." Or they need the difference between two columns rather than the raw values. In any case, it's usually something that could be implemented in half an hour, solving the problem once and for all. (These informal needs tend not to be recognized in formal "requirements" meetings, but they become immediately apparent if you spend a couple of hours tracking the the users' daily routines.)

Reports are for consumers, not producers. A common response to the previous item is that every user seems to have his own unique set of needs, and trying to meet all of them would lead to a proliferation of different reports.

There is of course some truth to that. But in my experience, certain reports are used by work groups in a fairly standard fashion. It is in these situations that the time spent on repetitive, routine editing tasks (such as those just described) is especially painful—and

avoidable. In such cases it might also be worthwhile to work with the group (or its management) to standardize their processes, so that in the end, a single report can meet everybody's needs.

But there is a bigger question here, too. Whose convenience is more important—the producers' or the users'? More broadly: for *whom* are the reports intended—for the reporting team or for the people looking at them?

Think about the proper metrics to show. For reports that show some form of summary statistics (as opposed to raw counts), think about which quantities to show. Will a mean (*e.g.*, "average time spent in queue") be sufficient, or is the distribution of values skewed, so that the median would be more appropriate? Do you need to include a measure for the width of the distribution (standard deviation or inter-quartile range)? (Answer: probably!) Also, don't neglect cumulative information (see Chapter 2).

Don't mix drill-down functionality with standard reporting. This may be a controversial item. In my opinion, reports are exactly that: standard overviews of the status of the system. Every time I run a report, I expect to find the same picture. (The numbers will change, of course, but not the overall view.) Drill-downs, on the other hand, are always different. After all, they are usually conducted in response to something out of the ordinary. Hence I don't think it makes sense devising a general-purpose framework for them; ad hoc work is best done using ad hoc tools.

Consider this: general-purpose frameworks are always clumsy and expensive yet they rarely deliver the functionality required. Would it be more cost-effective to forget about maintaining drill-down functionality in the reporting system itself and instead deploy the resources (*i.e.*, the developers) liberated thereby to address drill-down tasks on an ad hoc basis?

Don't let your toolset strangle you. Don't let your toolset limit the amount of value you can deliver. Many reporting solutions that I have seen can be awfully limiting in terms of the kind of information you can display and the formatting options that are available. As with any tool: if it gets in the way, evaluate again whether it is a net gain!

This is the list. I think the picture I'm trying to paint is pretty clear: fast, *simple*, and convenient reports that show lots of data but little else. Minimal overhead and a preference for cheap one-offs as opposed to expensive, general-purpose solutions. It's not all roses—in particular, the objection that a large number of cheap one-off reports might incur a significant total cost of ownership in the long run is well taken. On the other hand, every general-purpose reporting solution that I have seen incurred a similar cost of ownership—but did not deliver the same level of flexibility and convenience.

I think it is time to rethink reporting. The agile movement (whether right or wrong in all detail) has brought fresh life to software development processes. We should start applying its lessons to reporting.

Finally, a word about reporting tools. The promise of the reporting tools that I have seen is to consume data from "many sources" and to deliver reports to "many formats" (such as HTML, PDF, and Excel).

I have already suggested why I consider this largely an imaginary problem: I cannot conceive of a situation where you really need to deliver the same report in both HTML and PDF versions. If there is a requirement to support both formats, on close examination we will probably find that the HTML report is an operational report, whereas the PDF report is to be representational. There are probably additional differences between the two versions (besides the output format), in terms of layout, content, life cycle, and audience—just about everything.

Similar considerations apply regarding the need to pull data from many sources. Although this *does* occur, does it occur often enough that it should form the basis for the entire reporting architecture? Or does, in reality, most of the data come from relational databases and the odd case where some information comes from a different source (*e.g.*, an XML document, an LDAP server, or a proprietary data store) is best handled as a special case? (If you do in fact need to pull data from very different sources, then you should consider implementing a proper intermediate layer, one that extracts and *stores* data from all sources in a robust, common format. Reporting requires a solid and reliable data model. In other words, you want to isolate your reporting solution from the vagaries of the data sources—especially if these sources are "weird.")

The kinds of problems that reporting tools promise to solve strike me as classic examples of cases where a framework *seems* like a much better idea than it actually is. Sure, a lot of the tasks involved in reporting are lame and repetitive. However, designing a framework that truly has the flexibility required to function as a general-purpose tool is difficult, which leads to frameworks that are hard to use for everyone—and you still have to work around their limitations. The alternative is to write some boring but straightforward and most of all *simple* boilerplate code that solves *your* specific problems simply and well. I tend to think that some simple, problem-specific boilerplate code is in every way preferable to a big, complicated, all-purpose framework.

As for the actual delivery technology, I am all for simple tables and static, precomputed graphics—provided they are useful and well thought-out (which is not always as easy as it may seem). Specifically, I don't think that animated or interactive graphics—for example, using Adobe Flash, Microsoft Silverlight, or some other "Thick Client" technology—work well for reporting. Test yourself: how often do you want to wait for 5–10 seconds while some bar chart is slowly rendering itself (with all the animated bars growing individually from the base line)? Once you have seen this a few times, the "cute" effect has worn off, and the waiting becomes a drag. Remember that reports should be convenient, and that mostly means *quick*.

Thick clients do make sense as technologies for building "control consoles": complex user interfaces designed to operate a complex system that needs to be controlled in real time. But that's a very different job than reporting and should be (and usually is) treated as a core product with a dedicated software team.

Corporate Metrics and Dashboards

It is always surprising when a company doesn't have good, real-time, and consistent visibility into some of its own fundamental processes. It can be amazingly difficult to obtain insight into data such as: orders fulfilled today, orders still pending, revenue by item type, and so on.

But this lack of visibility should not come as surprise because up close, the problem is harder than it appears. Any business of sufficient size will have complex business rules, which furthermore may be inconsistent across divisions or include special exceptions for major customers. The IT infrastructure that provides the data will have undergone several iterations over the years and be a mixture of "legacy" and more current systems—none of which were primarily designed for our current purposes! The difficulties in presenting the desired data are nothing more than a reflection of the complexity of the business.

You may encounter two concepts that try to address the visibility problem just described: special *dashboards* and more general *metrics programs*. The goals of a metrics program are to *define* those quantities that are most relevant and should be tracked and to design and develop the infrastructure required to collect the appropriate data and make it accessible.

A dashboard might be the visible outcome of a metrics program. The purpose of a dashboard is to provide a high-level view of all relevant metrics in a single report (rather than a collection of individual, more detailed reports). Dashboards often include information on whether any given metric is within its desired range.

Dashboard implementations can be arbitrarily fancy, with various forms of graphical displays for individual quantities. An unfortunate misunderstanding results from taking the word "dashboard" too seriously and populating the report with graphical images of dials, as one might find in a car. Of course, this is beside the point and actually detracts from a legitimate, useful idea: to have a comprehensive, unified view of the whole set of relevant metrics.

I think it is important to keep dashboards simple. Stick to the original idea of all the relevant data on a single page—together with clear indications of whether each value is within the desired range or not.

As already explained when discussing reports, I do not believe that drill-down functionality should be part of the overall infrastructure. The purpose of the dashboard is

to highlight areas that need further attention, but the actual work on these areas is better done using individual, detailed research.

Recommendations for a Metrics Program

In case you find yourself on a project team to implement a metrics program, tasked to define the metrics to track and to design the required infrastructure, here are some concrete recommendations that you might want to consider.

Understand the cost of metrics programs. Metrics aren't free. They require development effort and deployment infrastructure of production-level strength, both of which have costs and overhead. Once in production, these systems will also require regular maintenance. None of this is free.

I think the single biggest mistake is to assume that a successful metrics program can be run as an add-on project without additional resources. It can't.

Have realistic expectations for the achievable benefit. The short-term effect of any sort of metrics program is likely to be small and possibly nondetectable. Metrics provide visibility and *only* visibility, but they don't improve performance. Only the decisions based on these metrics will (perhaps!) improve performance. But here the *marginal* gain can be quite small, since many of the same decisions might have been made anyway, based on routine and gut feeling.

The more important effect of a metrics program stems from the long-term effect it has on the organizational culture. A greater sense of accountability, or even the realization that there *are* different levels of performance, can change the way the business runs. But these effects take time to materialize.

Start with the actions that the metrics should drive. When setting out to define a set of metrics to collect, make sure to ask yourself: what decision would I make differently in response to the value of this metric? If none comes to mind, you don't need to collect it!

Don't define what you can't measure. This is a good one. I remember a metrics program where the set of metrics to track had been decided at the executive management level, based on what would be "useful" to see. Problem was, for a significant fraction of those quantities, no data was being collected and none could be collected because of limitations in the physical processes.

Build appropriate infrastructure. For a metrics program to be successful, it must be technically reliable, and the data must be credible. In other words, the systems that support it must be of *production-level quality* in regard to robustness, uptime, and reliability. For a company of any size, this requires databases, network infrastructure, monitoring—the whole nine yards. Plan on them! It will be difficult to be successful with only flat files and a CGI script (or with Excel sheets on a SharePoint, for that matter).

There is an important difference here between a more comprehensive *program* that purports to be normative and widely available, and an ad hoc report. Ad hoc reports can be extremely effective precisely because they do not require any infrastructure beyond a CGI script (or an Excel sheet), but they *do not scale*. They won't scale to more metrics, larger groups of users, more facilities, longer historical time frames, or whatever it is.

That being said, if all you need is an ad hoc report, by all means go for it.

Steer clear of manually collected metrics. First of all, manually collected metrics are neither reliable nor credible (people will forget to enter numbers and, if pressed, will make them up). Second, most people will resist having to enter numbers (especially in detail—think timesheets!), which will destroy the acceptance and credibility of the program. Avoid manually collected metrics at all cost.

Beware of aggregates. It can be very appealing to aggregate values as much as possible: "Just give me *one number* so that I see how my business is doing." The problem is that every aggregation step loses information that is impossible to regain: you can't unscramble an egg. And *actionable* information is typically *detailed* information. Knowing that my aggregated performance score has tanked is not actionable but knowing which *specific* system has failed is!

This leads us to questions about user interface design, roll-ups, and drill-downs. I think most of this is unnecessary. All that's required is a simple, high-level report. If details are required, one can always dig deeper in an ad hoc fashion.

Think about the math involved. The math required for corporate metrics is rarely advanced, but it still offers opportunities for mistakes. A common example occurs whenever we are forming a ratio—for example, to calculate the defect rate as the number of defects divided by the number of items produced. The problem is that the denominator can become zero (no items produced during the observation time frame), which makes it impossible to calculate a defect rate. There are different ways you can handle this (report as "not available," treat zero items produced as a special case, especially slick: add a small number to the denominator in your definition of the defect rate, so that it can never become zero), but you need to handle this possibility somehow (also see Appendix B).

There are other problems for which careful thinking about the best mathematical representation can be helpful. For example, to compare metrics they need to be normalized through rescaling by an appropriate scaling factor. For quantities that vary over many orders of magnitude, it might be more useful to track the logarithm instead of the raw quantity. Consider getting expert help: a specialist with sufficient analytical background can recognize trouble spots *and* make recommendations for how best to deal with them that may not be obvious.

Be careful with statistical methods that might not apply. Mean and standard deviation are good representations for the typical value and the typical spread only if the distribution of data points is roughly symmetrical. In many practical situations, this is *not* the case—waiting times, for instance, can never be negative and, although the "typical" waiting time may be quite short, there is likely to be a tail of events that take a very long time to complete. This tail will corrupt both mean and standard deviation. In such cases, median-based statistics are a better bet (see Chapter 2 and Chapter 9).

In general, it is necessary to study the nature of the data *before* settling on an appropriate way to summarize it. Again, consider expert help if you don't have the competency in-house.

Don't buy what you don't need. It is tempting to ask for a lot of detail that is not really required. Generally, it is not necessary to track sales numbers on a millisecond basis because we cannot respond to changes at that speed—and even if we could, the numbers would not be very meaningful because sales normally fluctuate over the course of a day.

Establish a meaningful time scale or the frequency with which to track changes. This time scale should be similar to the time scale in which we can make decisions and also similar to the time scale after which we see the results of those decisions. Note that this time scale might vary drastically: daily is probably good enough for sales, but for, say, the reactor temperature, a much shorter time scale is certainly appropriate!

Don't oversteer. This recommendation is the logical consequence of the previous one. Every "system" has a certain response time within which it reacts to changes. Applying changes more frequently than this response time is useless and possibly harmful (because it prevents the system from reaching a steady state).

Learn to distinguish trend and variation. Most metrics will be tracked over time, so what we have learned about time-series analysis (see Chapter 4) applies. The most important skill is to develop an understanding for the duration and magnitude of typical "noise" fluctuations and to distinguish them from significant changes (trends) in the data. Suppose sales dipped today by 20 percent: this is no cause for alarm if we know that sales fluctuate by ±25 percent from day to day. But if sales fall by 5 percent for five days in a row, that could possibly be a warning sign.

Don't forget the power of perverted incentives. When metrics are used to manage staff performance, this often means changing from a vague yet broad sense of "performance" to a much narrower focus on specifically those quantities that are being measured. This development can result in creating perverted incentives.

Take, for instance, the primary performance metric in a customer service call center: the number of calls a worker handles per hour, or "calls per hour." The best way for a call center worker who is evaluated solely in terms of calls per hour to improve her standing is by picking up the phone when it rings and hanging up immediately! By making calls per hour the dominant metric, we have implicitly deemphasized other important aspects, such as customer satisfaction (*i.e.*, quality).

Beware of availability bias. Some quantities are easier to measure than others and therefore tend to receive greater attention. In my experience, productivity is generally easier to measure than quality, with all the unfortunate consequences this entails.

Just because it can't be measured does not mean it does not exist. Some quantities cannot be measured. This includes "soft" factors such as culture, commitment, and fun; but also some very "hard" factors like customer satisfaction. You can't measure that—all you can measure directly are proxies (*e.g.*, the return rate). An alternative are surveys, but because participants decide themselves whether they reply, the results may be misleading. (This is known as *self-selection bias*.)

Above all, don't forget that a metrics program is intended to help the business by providing visibility—it should never become an end in itself. Also keep in mind that it is an effort to support others, not the other way around.

Data Quality Issues

All reporting and metrics efforts depend on the availability and quality of the underlying data. If the required data is improperly captured (or not captured at all), there is nothing to work with!

The truth of the matter is that if a company wants to have a successful business intelligence or metrics program, then its data model and storage solution *must be designed with reporting needs in mind*. By the time the demand for data analysis services rolls around, it is too late to worry about data modeling!

Two problems in particular occur frequently when one is trying to prepare reports or metrics: data may not be *available* or it may not be *consistent*.

Data Availability

Data may not be collected at all, often with the innocent argument that "nobody wanted to use it." That's silly: data that's directly related to a company's business is always relevant—whether or not anybody is looking at it right now.

If data is not available, this does not necessarily mean that it is not being collected. Data may be collected but not at the required level of granularity. Or it is collected but immediately aggregated in a way that loses the details required for later analysis. (For

instance, if server logs are aggregated daily into hits per page, then we lose the ability to associate a specific user to a page, and we also lose information about the order in which pages were visited.)

Obviously, there is a trade-off between the amount of data that can be stored and the level of detail that we can achieve in an analysis. My recommendation: try to keep as much detail as you can, even if you have to spool it out to tape (or whatever offline storage mechanism is available). Keep in mind that operational data, once lost, can *never* be restored. Furthermore, gathering new data takes *time* and cannot be accelerated. If you know that data will be needed for some planned analysis project, start collecting it *today*. Don't wait for the "proper" extraction and storage solution to be in place—that could easily take weeks or even months. If necessary, I do not hesitate to pull daily snapshots of relevant data to my local desktop, to preserve it temporarily, while a long-term storage solution is being worked out. Remember: every day that data is not collected is another day by which your results will be delayed.

Even when data is in principle collected at the appropriate level of detail, it may still not be available in a practical sense, if the storage schema was not designed with reporting needs in mind. (I assume here that the data in question comes from a corporate database—certainly the most likely case by far.) Three problems stand out to me in this context: lack of revision history, business logic commingled with data, and awkward encodings.

Some entities have a nontrivial life cycle: orders will go through several status updates, contracts have revisions, and so on. In such cases, it is usually important to preserve the full revision history—that is, all life-cycle events. The best way to do this is to model the time-varying state as a *separate entity*. For instance, you might have the `Order` entity (which contains, for example, the order ID and the customer ID) and the `OrderStatus`, which represents the actual status of the order (placed, accepted, shipped, paid, completed, ...), as well as a timestamp for the time that the status change took place. The current status is the one with the most recent status change. (A good way to handle this is with two timestamps: `ValidFrom` and `ValidTo`, where the latter is `NULL` for the current status.) Such a model preserves all the information necessary to study quantities like the typical time that orders remain in any one state. (In contrast, the presence of history tables with `OldValue` and `NewValue` columns suggests improper relational modeling.)

The important principle is that data is never *updated*—we only append to the revision history. Keep in mind that every time a database field is updated, the previous value is destroyed. Try to avoid this whenever you can! (I'd go so far as to say that CRUD—create, read, update, delete—is indeed a four-letter word. The only two operations that should ever be used are create and read. There may be valid operational reasons to move very old data to offline storage, but the data model should be designed in such a way that we never clobber existing data. In my experience, this point is far too little understood and even less heeded.)

The second common problem is business logic that is commingled with data in such a way that the data alone does not present an accurate picture of the business. A sure sign of this situation is a statement like the following: "Don't try to read from the database directly—you have to go through the access layer API to get all the business rules." What this is saying is that the DB schema was not designed so that the data can stand by itself: the business rules in the access layer are required to interpret the data correctly. (Another indicator is the presence of long, complicated stored procedures. This is worse, in fact, because it suggests that the situation developed inadvertently, whereas the presence of an access layer is proof of at least some degree of foreplanning.)

From a reporting point of view, the difficulty with a mandatory access layer like this is that a reporting system typically has to consume the data in bulk, whereas application-oriented access layers tend to access individual records or small collections of items. The problem is not the access layer as such—in fact, an abstraction layer between the database and the application (or applications) often makes sense. But it should be exactly that: an abstraction and access layer without embedded business logic, so that it can be bypassed if necessary.

Finally, the third problem that sometimes arises is the use of weird data representations, which (although complete) make bulk reporting excessively difficult. As an example, think of a database that stores only updates (to inventory levels, for example) but not the grand total. To get a view of the current state, it is now necessary to replay the entire transaction history since the beginning of time. (This is why your bank statement lists both a transaction history *and* an account balance!) In such situations it may actually make sense to invest in the required infrastructure to pull out the data and store it in a more manageable fashion. Chances are good that plenty of uses for the sanitized data will appear over time (build it, and they will come).

Data Consistency

Problems of data consistency (as opposed to data availability) occur in every company of sufficient size, and they are simply an expression of the complexity of the underlying business. Here are some typical examples that I have encountered.

- Different parts of the company use different definitions for the same metric. Operations, for example, may consider an order to be completed when it has left the warehouse, whereas the finance department does consider an order to be complete once the payment for it has been received.
- Reporting time frames may not be aligned with operational process flows. A seemingly simple question such as, "How many orders did we complete yesterday?" can quickly become complicated, depending on whose definition of "yesterday" we use. For example, in a warehouse, we may only be able to obtain a total for the number of orders completed per shift—but then how do we account for the shift that stretches from 10 at night to 6 the next morning? How do we deal with time zones? Simply

stating that "yesterday" refers to the local time at the corporate headquarters sounds simple but is probably not practical, since all the facilities will naturally do their bookkeeping and reporting according to their local time.

- Time flows backward. How does one account for an order that was later returned? If we want to recognize revenue in the quarter in which the order was completed but an item is later returned, then we have a problem. We can still report on the revenue accurately—but not in a timely manner. (In other words, final quarterly revenue reports cannot be produced until the time allowed to return an item has elapsed. Keep in mind that this may be a *long* time in the case of extended warranties or similar arrangements.)

Additional difficulties will arise if information has been lost—for instance, because the revision history of a contract has not been kept (recall our earlier discussion). You can probably think of still other scenarios in which problems of data or metric inconsistency occur.

The answer to this set of problems is not technical but administrative or political. Basically it comes down to agreeing on a common definition of all metrics. An even more drastic recommendation to deal with conflicting metrics is to declare one data source as the "normative" one; this does not make the data any more accurate, but it can help to stop fruitless efforts to reconcile different sources at any cost. At least that's the theory. Unfortunately, if the manager of an off-site facility can expect to have his feet held to the fire by the CEO over why the facility missed its daily goal of two million produced units by a handful of units last Friday, he will look for ways to pass the blame. And pointing to inconsistencies in the reports is an easy way out. (In my experience, one major drawback of all metrics programs is the amount of work generated to reconcile minute inconsistencies between different versions of the same data. The costs—in terms of frustration and wasted developer time—can be stunning.)

As practical advice I recommend striving as much as possible for clear definitions of all metrics, so that at least we know what we're talking about. Furthermore, wherever possible, try to make those metrics normative that are *practical* to gather, rather than those "correct" from a theoretical point of view (*e.g.*, report metrics in local instead of global time coordinates). Apply conversion factors behind the scenes, if necessary, but try to make sure that humans only need to deal with quantities that are meaningful and familiar to them.

Workshop: Berkeley DB and SQLite

For analysis purposes, the most suitable data format is usually the flat file. Most of the time, we will want all (or almost all) of the records in a data set for our analysis. It therefore makes more sense to read the whole file, possibly filter out the unneeded records, and process the rest, rather than to do an indexed lookup of only the records that we want.

Common as this scenario is, it does not always apply. Especially when it comes to reporting, it can be highly desirable to have access to a data storage solution that supports structured data, indexed lookup, and even the ability to merge and aggregate data. In other words, we want a database.

The problem is that most databases are *expensive*—and I don't (just) mean in terms of money. They require their own process (or processes), they require care and feeding, they require network access (so that people and processes can actually get to them). They must be designed, installed, and provisioned; very often, they require architectural approval before anything else. (The latter point can become such an ordeal that it makes anything requiring changes to the database environment virtually impossible; one simply has to invent solutions that do without them.) In short, most databases are expensive: both technically and politically.

Fortunately, other people have recognized this and developed database solutions that are cheap: so-called *embedded databases*. Their distinguishing feature is that they do not run in a separate process. Instead, embedded databases store their data in a regular file, which is accessed through a library linked into the application. This eliminates most of the overhead for provisioning and administration, and we can replicate the entire database simply by copying the data file! (This is occasionally very useful to "deploy" databases.)

Let's take a look at the two most outstanding examples of (open source) embedded databases: the Berkeley DB, which is a key/value hash map stored on disk, and SQLite, which is a complete relational database "in a box." Both have bindings to almost any programming language—here, we demonstrate them from Python. (Both are included in the Python Standard Library and therefore should already be available wherever Python is.)

Berkeley DB

The Berkeley DB is a key/value hash map (a "dictionary") persisted to disk. The notion of a persistent key/value database originated on Unix; the first implementation being the Unix `dbm` facility. Various reimplementations (`ndbm`, `gdbm`, and so on) exist. The original "Berkeley DB" was just one specific implementation that added some additional capabilities—mostly multiuser concurrency support. It was developed and distributed by a commercial company (Sleepycat) that was acquired by Oracle in 2008. However, the name "Berkeley DB" is often used generically for any key/value database.

Through the magic of operator overloading, a Berkeley DB also *looks* like a dictionary to the programmer* (with the requirement that keys and values must be *strings*):

```
import dbm

db = dbm.open( "data.db", 'c' )
```

*In Perl, you use a "tied hash" to the same effect.

```
db[ 'abc' ] = "123"
db[ 'xyz' ] = "Hello, World!"
db[ '42' ] = "42"

print db[ 'abc' ]

del db[ 'xyz' ]

for k in db.keys():
    print db[k]

db.close()
```

That's all there is to it. In particular, notice that the overhead ("boilerplate") required is precisely zero. You can't do much better than that.

I used to be a great fan of the Berkeley DB, but over time I have become more aware of its limitations. Berkeley DBs store single-key/single-value pairs—period. If that's what you want to do, then a Berkeley DB is great. But as soon as that's not *exactly* what you want to do, then the Berkeley DB simply is the wrong solution. Here are a few things you *cannot* do with a Berkeley DB:

- Range searches: `3 < x < 17`
- Regular expression searches: `x like 'Hello%'`
- Aggregation: `count(*)`
- Duplicate keys
- Result sets consisting of multiple records and iteration over result sets
- Structured data values
- Joins

In fairness, you can achieve some of these features, but you have to build them yourself (*e.g.*, provide your own serialization and deserialization to support structured data values) or be willing to lose almost all of the benefit provided by the Berkeley DB (you can have range or regular expression searches, as long as you are willing to suck in *all* the keys and process them sequentially in a loop).

Another area in which Berkeley DBs are weak is administrative tasks. There are no standard tools for browsing and (possibly) editing entries, with the consequence that you have to write your own tools to do so. (Not hard but annoying.) Furthermore, Berkeley DBs don't maintain administrative information about themselves (such as the number of records, most recent access times, and so on). The obvious solution—which I have seen implemented in just about every project using a Berkeley DB—is to maintain this

information explicitly and to store it in the DB under a special, synthetic key. All of this is easy enough, but it does bring back some of the "boilerplate" code that we hoped to avoid by using a Berkeley DB in the first place.

SQLite

In contrast to the Berkeley DB, SQLite (*http://www.sqlite.org/*) is a full-fledged relational database, including tables, keys, joins, and `WHERE` clauses. You talk to it in the familiar fashion through SQL. (In Python, you can use the DB-API 2.0 or one of the higher-level frameworks built on top of it.)

SQLite supports almost all features found in standard SQL with very few exceptions. The price you pay is that you have to design and define a schema. Hence SQLite has a bit more overhead than a Berkeley DB: it requires some up-front design as well as a certain amount of boilerplate code.

A simple example exercising many features of SQLite is shown in the following listing. It should pose few (if any) surprises, but it does demonstrate some interesting features of SQLite:

```
import sqlite3

# Connect and obtain a cursor
conn = sqlite3.connect( 'data.dbl' )
conn.isolation_level = None            # use autocommit!
c = conn.cursor()


# Create tables
c.execute( """CREATE TABLE orders
              ( id INTEGER PRIMARY KEY AUTOINCREMENT,
                customer )""" )
c.execute( """CREATE TABLE lineitems
              ( id INTEGER PRIMARY KEY AUTOINCREMENT,
                orderid, description, quantity )""" )

# Insert values
c.execute( "INSERT INTO orders ( customer ) VALUES ( 'Joe Blo' )" )
id = str( c.lastrowid )
c.execute( """INSERT INTO lineitems ( orderid, description, quantity )
              VALUES ( ?, 'Widget 1', '2' )""", ( id, ) )
c.execute( """INSERT INTO lineitems ( orderid, description, quantity )
              VALUES ( ?, 'Fidget 2', '1' )""", ( id, ) )
c.execute( """INSERT INTO lineitems ( orderid, description, quantity )
              VALUES ( ?, 'Part 17', '5' )""", ( id, ) )

c.execute( "INSERT INTO orders ( customer ) VALUES ( 'Jane Doe' )" )
id = str( c.lastrowid )
```

```
c.execute( """INSERT INTO lineitems ( orderid, description, quantity )
              VALUES ( ?, 'Fidget 2', '3' )""", ( id, ) )
c.execute( """INSERT INTO lineitems ( orderid, description, quantity )
              VALUES ( ?, 'Part 9', '2' )""", ( id, ) )


# Query
c.execute( """SELECT li.description FROM orders o, lineitems li
              WHERE o.id = li.orderid AND o.customer LIKE '%Blo'""" )
for r in c.fetchall():
    print r[0]

c.execute( """SELECT orderid, sum(quantity) FROM lineitems
              GROUP BY orderid ORDER BY orderid desc""" )
for r in c.fetchall():
    print "OrderID: ", r[0], "\t Items: ", r[1]


# Disconnect
conn.close()
```

Initially, we "connect" to the database—if it doesn't exist yet, it will be created. We specify autocommit mode so that each statement is executed immediately. (SQLite also supports concurrency control through explicit transaction.)

Next we create two tables. The first column is specified as a primary key (which implies that it will be indexed automatically) with an autoincrement feature. All other columns do not have a data type associated with them, because basically all values are stored in SQLite as strings. (It is also possible to declare certain type conversions that should be applied to the values, either in the database or in the Python interface.)

We then insert two orders and some associated line items. In doing so, we make use of a convenience feature provided by the `sqlite3` module: the last value of an autoincremented primary key is available through the `lastrowid` attribute (data member) of the current cursor object.

Finally, we run two queries. The first one demonstrates a join as well as the use of SQL wildcards; the second uses an aggregate function and also sorts the result set. As you can see, basically everything you know about relational databases carries over directly to SQLite!

SQLite supports some additional features that I have not mentioned. For example, there is an "in-memory" mode, whereby the entire database is kept entirely in memory: this can be very helpful if you want to use SQLite as a part of a performance-critical application. Also part of SQLite is the command-line utility `sqlite3`, which allows you to examine a database file and run ad hoc queries against it.

I have found SQLite to be extremely useful—basically everything you expect from a relational database but without most of the pain. I recommend it highly.

Further Reading

- *Information Dashboard Design: The Effective Visual Communication of Data.* Stephen Few. O'Reilly. 2006.
  This book addresses good graphical design of dashboards and reports. Many of the author's points are similar in spirit to the recommendations in this chapter. After reading his book, you might consider hiring a graphic or web designer to design your reports for you!