Failures in the RDBMS world

By Willem Leenen

Reader PostgreSQL conference 2013
Dublin, Ireland

Content:

The Entity Attribute Value model

Source: http://www.simple-talk.com/opinion/opinion-pieces/bad-carma/

In my undergraduate career, I took a single course in philosophy, which was a requirement for my liberal arts degree in economics. Up to that point, my only exposure to philosophy was Monty Python, and the lyrics of overwrought 1960s folk songs and overblown 1970s rock songs. The course was entitled "Introduction to Ethics" and in the first session, the professor asked the class, "How many people are here just to fulfill a prerequisite?" It was apparent that the only hands that weren't raised belonged to those who didn't know the meaning of the word "prerequisite." The classroom was not quite a football stadium, but it probably should have been, as there were hundreds of people, and the majority of them appeared to be on athletic scholarships. Resignedly, the professor nodded and continued.

I immediately fell asleep but was soon roused when the professor asked, "What is *good*? In other words, what constitutes what is *good* and *right* and *decent*? How is *good* defined?" Despite a relatively spirited discussion, it became clear to all that nobody could really answer the question adequately. Finally, the professor, no doubt through long experience in teaching "Introduction to Ethics," posed the best answer of all the bad answers. He called it the "Schlitz Ethic." This was named after a well-known brand of fantastically bad American beer, whose advertising at the time always concluded with the jingle, "When it's right, you know it!"

When it's right, you know it. Implying that, although it is difficult to describe the meaning of *good*, you know *good* when you see it.

Warily, the entire class of 250 or so agreed that this worked as well as any of the other answers. The professor grinned sadly, correctly assuming that few of us appreciated the irony of beer advertising in relation to a discussion of ethics, and assigned Plato for homework.

When it's right, you know it. And stating the converse, *when it's not right, you know that too ...*

Obligatory disclaimer

I will fictionalize names to protect the identities of those involved, because almost all of the people who were involved are still active in the IT world, although the corporations involved have long ago bitten the dust, gone to meet their makers, joined the choir invisible, and are now pushing up daisies.

If you think you recognize any person or corporation in this account, then you are most certainly mistaken. Just have two stiff whisky drinks and that feeling of recognition should pass ...

Introducing Vision

Those of us who work in *Information Technology* (IT) have all been on a project where something important is just not right. We know it, most everyone knows it, but nobody is quite able to put his or her finger on the problem in a convincing way. Why I was not able to muster a convincing argument at the time I can only attribute to lack of experience. As an old adage states: *experience is what you get immediately after you need it*.

This story is about such an IT project, the most spectacular failure I have ever experienced. It resulted in the complete dismissal of a medium-sized IT department, and eventually led to the destruction of a growing company in a growing industry. The company, which we'll call "Upstart," was a successful and profitable subscription television business. Upstart at this time was a wholly owned subsidiary of a giant media conglomerate, which we'll call "BigMedia, Inc."

The project occurred in the early 1990s, and it was a custom-built order-entry and customer-service application, closely resembling what is now referred to as *Customer-Relationship Management* or *CRM*. The core functionality of the system included:

- Order entry and inventory
- Customer service, help desk
- General ledger, accounts receivable, billing, and accounts payable

The application was called "Vision" and the name was both its officially stated promise for Upstart as well as a self-aggrandizing nod to its architect. The application was innovative, in that it was built to be flexible enough to accommodate any future changes to the business. Not just any *foreseeable* future changes to the business, but *absolutely any* changes to the business, in any form. It was quite a remarkable claim, but Vision was intended to be the last application ever built. It achieved this utter flexibility by being completely data-driven, providing limitless abstraction, and using object-oriented programming techniques that were cutting-edge at the time.

Like many such projects that set out to create a mission-critical application, the development effort spanned two years, about a year longer than originally projected. But that was acceptable, because this was the application that would last forever, adapting to any future requirements, providing unlimited Return On Investment (ROI). When the application finally went "live," almost everybody in the company had invested so much in it that literally the fate of the company hinged on its success.

However, in the event of total project malfunction, mission-critical applications running the core business of multinational corporations are not permitted the luxury of the type of fast flameout demonstrated by thousands of "dot-com" companies in the era of the Internet bubble a few years. Within a month of Vision going "live," it was apparent to all but those most heavily vested in its construction that it was a failure. The timeline went as follows:

- *Autumn 1991*: Development begins—the "build versus buy" decision is settled in favor of "build" as Upstart decides to build a custom CRM application.
- *September 1993:* "Go Live" launch—hope and euphoria.
- *October 1993:* Frustration at poor performance and inability to produce reports.
- *November 1993:* Desperation.
- *December 1993:* Despair, IT director resigns.
- *January 1994:* Firings, IT department decimated.
- *February 1994:* New faces.
- *March 1994:* New replacement system chosen, the "build versus buy" decision is settled in favor of "buy (then customize)."
- *November 1994:* Replacement system launched, "Vision" retired.
- *March 1995:* BigMedia, Inc. sells Upstart to another company.

So, the entire debacle lasted three years, from inception to grave, generating an estimated ROI of minus $10 million. Such implosions remained rare until the spectacular supernovas when the Internet "bubble" burst in 2001.

Seeing and Vision

Vision was designed and implemented by a person called Randy. Randy was a vigorous proponent of object-oriented programming and modular programming languages such as Ada. From product development, he had moved into Oracle Consulting where he was regarded as brilliant but "controlling, sarcastic, and impatient", according to those who worked with him. Apparently, there were no openings in marketing at the time.

Oracle placed Randy on an engagement at Upstart in 1991 during the initial strategy phase of the Vision project. It wasn't long before Upstart realized that they had a systems architect with all of the answers. For each of their requirements, Randy had a solution:

Build an order-entry and customer-service application that integrates all of the elements of customer interaction

Upstart wanted a better end-user interface for their reservation agents. They also wanted that system to be capable of retrieving every bit of information about a customer while the reservation agent was interacting with the customer. Such systems now come shrink-wrapped, but in the early 1990s the "build-versus-buy" decision still came down firmly on the "build" side for this kind of functionality. Randy promised to build an application that could do absolutely anything.

Make it fast and reliable

Upstart spent a lot of money on its IT infrastructure. They bought the largest, most reliable servers available and the best database technology of the time (Oracle7). As far as Upstart was concerned, Vision would be fast, *ipso facto*, because it had the best server hardware and the best database software.

Build an application that is flexible enough to be adapted to meet any new requirements

This was the toughest requirement, and this was where Randy really impressed everyone. In computing, there is often a three-way trade-off between *speed*, *reliability*, and *flexibility*. This is truly the devil's triangle of IT, where the conventional wisdom is that you can pick any two out of the three.

Vendor after vendor promised Upstart that their product could meet all of their present requirements, speedily and reliably. But changes? Future requirements? This is where the hemming and hawing started. The vendors would admit, with varying degrees of enthusiasm, to the presence of an *application programming interface* or API, allowing modifications and customizations to the application. How easy is it to change? Well, that's what our professional services organizations are for!

It was not until Upstart talked to Randy that they heard a story that they liked, which was that all those existing products are built on the wrong technology.

Randy proposed building, from scratch, a completely *data-driven* application. Most conventional database applications are written in *code* using programming languages that access and manipulate *data*. Code is very difficult to change once it is written, but code controls data.

Randy's idea was that all of the forms, all of the reports, all of the functionality of the system would be stored as *metadata*, or *data about data*, and the only thing coded in a programming language would be an *engine* to process both metadata and data. For example, instead of defining a table just to store order-line information, why not also store the basic business logic about inserting, updating, and deleting order-lines in the table, as well as details about the structure of an order-line. Once an engine was built, then that would be the end of all coding, forever and ever, amen. After that, the engine would process data about data then the data itself, and in this fashion any form, report, or program could be built. When new functionality or modifications were needed to the system, you only had to change the metadata, not the code.

Theoretically, this allowed rapid implementation of changes and new applications, because while changes to programming code required recompilation and relinking, changes to metadata had no such requirements.

And Randy delivered on these promises—well, *most* of them anyway ...

The data-driven application

One of Randy's inspirations for this design was early *Computer-Aided Software Engineering* (CASE), in particular Oracle's own CASE*Tools product. In its early incarnations, this product included a table named SDD_ELEMENTS that contained both *data* and *metadata* indistinguishably—it was all just grist for the processing logic within the CASE product. The metadata was initialized as "seed data" by the product installation software, and then the "engine" used this as the basis for recursive queries back into itself when new data or metadata was being added. Randy commented that this recursive design was a "eureka" moment, an epiphany, for him and it came to fruition in the Vision system.

The idea was that nothing about the application logic was declared or coded. Not the business logic, and not the data structures for storage.

The pity is that Randy did not temper that epiphany with some real-world practicality. For example, what works very well for a CASE application supporting 1–5 developers who think a lot and type only a little may not work well at all for a mission-critical order-entry application supporting 300–400 reservation agents who think very little and type very rapidly. The database workload generated by a small number of users generating transactions intermittently has little in common with the database workload generated by an order-entry application. Just considering the growth in the volume of data between the two situations reveals striking differences.

Recalling the three-way design trade-off between *speed*, *reliability*, and *flexibility*, a CASE application needs *flexibility* the most in order to capture the most complex ideas and designs. It needs *reliability* and robustness as well in order for the application developer to trust it enough to use it. It does need to perform well, but with such a small user community, *speed* is probably the least important side of the triangle.

For an order-entry, inventory, customer-service, and financial application, *reliability* and high availability might be the most important characteristic of all, as the mission-critical application that is not available or which loses data is worth nothing. Next, such a system has to perform well, as the lack of *speed* for the dozens or hundreds of concurrent users can also render the application unfit to fulfil business requirements. Of the three qualities, *flexibility* may well be the least important attribute as far as business users are concerned.

The Vision systems best quality, *flexibility*, was least important to Upstart as a company, despite its attractiveness to Upstart's IT division. It was *reliable* enough, but it did not have the *speed* to fulfill its business requirements.

This is a classic example of what can happen when the priorities of the IT "department" do not match, and are not appropriately aligned with, the priorities of the "business." Often what appears desirable to the propeller heads and what is seen as a high priority (e.g., using cutting-edge technology, flexible, database independence, etc.) may implicitly conflict with the priorities and business requirements of the company. When IT decisions are made in a vacuum from the business rather than considering the wishes of the business foremost, these sad cases become an unfortunate reality.

When it's not right, you know it . . . or not?

In addition to mixing metadata with data, the Vision system had another peculiarity that made it truly memorable, at least to those with any experiences building databases.

The Vision system was comprised of a single table, named DATA, appropriately enough. When you consider the overriding goal of complete flexibility where all rules are interpreted at run time, it seems inevitable that every involving "structure" would also be made as generic and undistinguished as possible. Why have multiple tables when there is no differentiation of data structure? Instead, just pile all of the 150 or so different logical entities into the same table. Plunk the various items of data into generically defined columns—the application data itself contains the metadata identifying each item of data and how it should be stored and displayed.

The basic premise was that just about all of the features of the relational database were eschewed, and instead it was used like a filing system for great big plastic bags of data. Why bother with other containers for the data—just jam it into a generic black plastic garbage bag. If all of those bags full of different types of data all look the same and are heaped into the same pile, don't worry! We'll be able to differentiate the data after we pull it off the big pile and look inside.

Amazingly, Randy and his crew thought this was incredibly clever. Database engineer after database engineer were struck dumb by the realization of what Vision was doing, but the builders of the one-table database were blissfully aware that they were ushering in a new dawn in database design.

Here are some facts about the Vision system:

- The data model comprised a single table named DATA.
- The DATA table had 240+ columns.
- The primary key column was a numeric named SYSNO.
- Columns existed for attributes, such as TYPE, SUBTYPE, CATEGORY, SUBCATEGORY, STATUS, SUBSTATUS, GROUP, and OWNER, which were intended to fully describe what type, category, or grouping to which the row belonged. Each of these columns were themselves SYSNOs, joining back to other rows in DATA for more detail.
- The majority of columns in DATA provided sequencing, descriptive text, names, values, amounts, dates entered and modified, and so on. Some of these columns would be named and data-typed appropriately, while others were "generic spare" columns, several for each datatype.
- When the Vision system was finally decommissioned, a year after it went into production, the DATA table consumed 50GB of space.
- 40+ associated indexes consumed another 250GB of space.

Suppose you wanted to find a Customer record. To find that information, you first needed to retrieve metadata describing the structure of a customer entity from the DATA table. This might involve first a query on DATA to retrieve a row describing one of about 150 different logical "entities," then a row describing another specific entity of Customer. Then, it would be necessary to use the information retrieved so far to query DATA for rows related to "entity" to describe "columns," so that we know in what column on the DATA table the COMPANY_NAME and address information is stored within.

Once all of this metadata has been retrieved, we are ready to start querying the DATA table for a specific row for a specific customer. Using all of the metadata previously retrieved, we know how many of the 240 columns in data are populated with customer data, what type of data they should be, what the acceptable ranges of values are, how the numeric or date data should be displayed, and so forth.

A SQL query for a customer record in a conventional database application might look like this:

A SQL query for a customer record in a conventional database application might look like this:

```
Select name,
      mailing_street_addr1,
      mailing_street_addr2,
      mailing_city,
      mailing_state_province,
      mailing_postal_zip,
      ...
from company
where company_id = <company-ID>
```

In the Vision system, a similar SQL query might look like this:

```
Select cn.description,
      ma.string82,
      ma.string83,
      mc.string44,
      mc.string63,
      mz.numeric31,
      ...
from      data c,
      data cc,
      data cn,
      data ma,
      data mc,
      data mz,
      data ...
where      c.description = 'CUSTOMER'
and      cc.entity_id = c.sysno
and      cc.description = 'COLUMN'
and      ma.column_id = cc.sysno
and      ...
```

It was not unusual for such a simple query, intended only to retrieve information for one customer, to perform 6–10 recursive "self-joins" back to the DATA table, over and over again. More demanding queries, for instance those retrieving several customer records, or order details, required at least 12–15 recursive "self-joins." Every operation had first to retrieve the metadata about the structure of the data before retrieving or modifying the data itself.

---***Analogy---***
In real life, when a fire department receives an alarm about a fire in progress, the nearest fire station is notified and a crew is dispatched to handle the situation. The advantage of this situation is that both the dispatcher and crew handle the situation automatically and quickly. The disadvantage of this situation is that if there is a change in procedure, it may take some time to retrain the personnel to make them accustomed to the new procedure.

If this were instead handled the way the Vision system operated, each alarm would cause the fire dispatcher to check his or her training manuals on what to do when an alarm is received. Then, upon receipt of a dispatch, each fire crew member would have to check his or her training manuals to determine what to do when handling a fire call. Then, the crew would go handle the situation, carrying training manuals with them for any eventuality. The advantage of this approach is that each alarm would be handled the way the training manuals dictate, so that changes in procedures can be instantly implemented. The disadvantage, of course, is that you have dispatchers reading manuals when they should be dispatching, and fire crew members reading manuals when they should be fighting fires. Chaos in the streets, as trucks careen wildly while their drivers consult manuals instead of keeping their eyes on the road. Ah, I feel a Hollywood screenplay coming on ...
---***End Analogy***---

To illustrate its remarkable flexibility, the entire Vision application consisted of a single form. This single form utilized code extensions called "user exits" written in C which called the Vision "engine." This single form was therefore able to display and accept user input for any of the 150+ data entities that comprised all of the modules of the Vision system.

So, the Vision engine was able to generate these SQL statements very easily, because it was coded to piece together metadata to retrieve the data. However, each invocation of the form required that the relevant form be generated afresh. If 300 data-entry operators invoked a form, the logic to build the form was generated 300 times. If operators hopped back and forth between screens, then all of this form-generation logic occurred again and again and again. Please note that changes to order-entry forms tend to occur rather infrequently, so you have to wonder if this incredible flexibility was a solution in search of a problem.

For human beings, the situation was dire. What the Vision application's object-oriented "engine" did very rapidly and easily did not come quickly to the carbon-based meat-processors in the skulls of developers. For example, developers would be asked to compose a report about customer orders or summarize monthly sales. Sounds straightforward enough, but before retrieving the data, they had to retrieve the instructions on how to retrieve the data. Once they retrieved that logic, then they had to retrieve the structure in which data is stored, for each individual item of data required. Using simple, basic *ad-hoc* query tool or reporting tools (such as Oracle's SQL*ReportWriter or Crystal Reports)) became unbelievable drudgery, bordering on the impossible. Reporting tools that generated their own SQL, based upon expectations of a standard normalized data model, were completely out of the question. In order to shorten the number of levels of recursive logic, developers ended up hard-coding as much of the metadata as they could, in an effort to query only the data. This expediency drove Randy out of his mind, as he rightly perceived that all of Vision's flexibility was slowly and surely leaking out in each instance of hard-coded metadata.

Expanding Vision

The DATA table eventually grew to 50GB in size. Oracle7 did not have any partitioning, so as the table grew larger, every access became correspondingly slower. Using the 2,048 byte database blocks which were common at the time, most of the indexes on DATA grew to 5–6 branch levels, which made navigation of the branch levels expensive, even for UNIQUE scans into the index. Moreover, the clustering of data values in the table with regard to their value in the 40+ indexes was almost uniformly poor. The ordering in which rows were inserted into the table only matched the ordering of the data values on the SYSNO column, so that was the only index which had a good clustering factor of values in the index compared to the rows in the table. This meant that the index on the SYSNO column was the only index that was well suited to RANGE scans across the leaf blocks of the index. Unfortunately, since SYSNO was the primary key of the DATA table, a UNIQUE probe for individual rows from that index was the most common operation. So, the only index suited for RANGE scans was also the index that was hardly ever used for RANGE scans. All of the other columns against which RANGE scans might be performed were huge, with many branch levels from the "root" node to the "leaf" nodes, and they were poorly suited to scanning for multiple rows.

Moreover, because the DATA table contained over 150 logical data entities, and some of those entities were frequently updated and deleted from, all of the 40+ indexes had an enormous number of deleted entries. These "holes" in the index entries are due to the fact that Oracle does not remove an index entry, but rather enables its possible future reuse by simply nullifying the pointer to the table row and keeping the data value portion in its slot. Thus, as rows were deleted and updated in the DATA table, most of the indexes were becoming more and more sparsely populated over time.

Because the entire application was encapsulated inside that table, and because much of the data-driven logic contained within DATA was recursive, we noted that the growth rate of DATA was exponential. The phenomenon seemed to resemble the growth of the human population of the earth: as more people procreate, more people are added more rapidly, which causes more people to procreate, which causes more people to be added even more rapidly, which causes even more people to procreate, (and so on) ...

So, DATA itself was growing exponentially. Its indexes were mirroring that growth, but growing even faster as deleted entries compounded the problem. The Sequent server on which the database resided was among the largest in the United States at the time, but it was quickly being gobbled by this fast-expanding database.

It was at this point that I was called in.

Flawed Vision

My first recommendation, upon being consulted by Upstart to help improve performance, was to break all 150+ logical entities into separate tables, similar to any other conventional database application built on hierarchical or relational databases. This consultation occurred about a month prior to Vision going live, and the Upstart project managers smiled indulgently and commented, "Every database consultant who comes in here says that." The implication was, "Tell us something new, because that's not going to happen." They went on to explain that this application was something new, and that the way it stored data and metadata in the database would revolutionize computing. I listened carefully, because the customer is always right, but in the end, I had nothing else to offer, and the interview was ended. Randy was not present at that initial meeting, as his faithful were well able to deal with the infidel.

Four months later, after Vision had been in production for three months, I was called back in to help improve performance, which was crippling all areas of business within Upstart. Again, I looked at what the application was doing and how the DATA table was growing, and repeated the same conclusion: this application needs a more conventional data model, with each logical entity stored in its own relational table.

This time, the reception was a little different. Randy was present at this meeting and repeated what had been said by his minions on my previous visit: it was impossible to break up DATA. Vision's design was revolutionary, so be clever and think of something else. Change some database initialization parameters, or something. After all, the poor performance was Oracle's problem, as the Vision application "engine" was operating flawlessly, so the problem had to be Oracle.

I noticed that the IT director who had been present at the last meeting four months earlier was missing, and found that he had resigned only two weeks prior. The new IT director was present, and she was obviously not one of Randy's faithful. In fact, I couldn't help noticing that even the faithful were not presenting a solid front anymore, and that Randy seemed isolated. So, I pressed on with my litany of the problems with Vision.

Too many indexes

The DATA table had over 40 indexes, so each INSERT statement had to insert over 40 index entries for each table row inserted and each DELETE statement had to delete over 40 index entries for each table row deleted.

When modifications were made to existing data using Oracle's SQL*Forms v3.0, the program had a nasty habit of including all of the columns in the table in the resulting UPDATE statement. So, even when just one column was modified in the form, the resulting update would still attempt to change all of the columns maintained by the form. As a result, each UPDATE statement performed modifications on all 40 related index entries for each table row updated

Each UPDATE and DELETE made each of the 40 indexes more sparsely populated, making them less efficient for just about every type of index access method.

Too many branch levels

With the 2,048 byte database block size that was the norm at the time, each of these 40 indexes had four, five, and sometimes six levels of B*Tree branches. Thus, the simplest use of any of the indexes was several times more "expensive" than they needed to be, as all branch levels had to be traversed from the index's "root" node to one of the "leaf" nodes on each access.

Breaking the DATA table up into 150 smaller tables would allow the indexes on the smaller tables to have fewer branch levels, making each access more efficient.

Index access contention

One of the 40+ indexes, the unique index supporting the primary key column SYSNO, was populated with numeric values generated by a sequence generator. Likewise, a column named TIMESTAMP was populated with the current system DATETIME value. Both sequence values and datetimestamps are monotonically ascending data values which present special problems for indexes

Therefore, the index "leaf" block containing the highest values was always being fought over by all of the sessions inserting new rows, representing a knot of contention within the application that is difficult to resolve.

Could not exploit caching

Oracle7 had introduced a new CACHE command for caching individual tables into memory. With only one table in the application, it would be impossible to cache frequently used logical entities into memory.

Users could not write queries against DATA

It was too difficult and complex for humans to write queries on the DATA table, with the need to join back to DATA again and again in order to retrieve structural metadata before getting around to the actual purpose of the query itself.

All developers were now short-circuiting these recursive self-joins by memorizing specific SYSNO values and hard-coding specific SYSNO values into their queries. This had the effect of reducing the number of recursive self-joins in the query, allowing them to run and complete, but it also had the side effect of eliminating all of the flexibility designed into the Vision application.

As I described these shortcomings, Randy kept shaking his head, muttering about what a piece of junk the Oracle database was, what kind of fools complained about the complexity of a revolutionary data model when writing queries, and so on. Finally, he simply stood up and walked out of the meeting.

It was the last point, the least technical point, which resonated with everyone. Each of the previous points were technical mumbo-jumbo, but the point about **humans not being able to write queries** struck a nerve.

Death of a vision

Also present in the meeting was a sales representative from a market research company, whom we'll call "Joe," who had been trying to sell his market-research software to Upstart. Joe knew that he would have to extract data from Vision, as his application was an *Executive Information System* (EIS), which is more commonly referred to as a *data mart* today.

Joe mostly sat and listened throughout the meeting. After I finished my laundry list of problems with Vision, and after Randy stormed out of the room, he finally spoke. He asked one of the Vision project team leads, one of Randy's faithful, whom we'll call "Rick," whether Vision was indeed as complicated as I was making it out to be. Rick proudly answered, "Yes, it is." Joe then asked how long it took to train an application developer to implement changes in functionality, and Rick frowned, thought a moment, and then answered, "Six months." Joe looked thunderstruck and slowly repeated, "Six months?" Rick explained that they had just finished bringing a new application developer up to speed on Vision, and it took six months. "He was really bright, too!" Rick said pensively. Joe nodded and then turned to the new IT director.

He stated, "You are in a world of hurt. It takes six months to train a new maintenance person and a database consultant," waving at me, "says that the simplest of reports can only be created with the greatest difficulty using the most low-level reporting tools." He stood up, walked to the white board, picked up a pen, and wrote:

"***RUN LIKE HELL*!**"

Joe said, "You people," waving at the accounting director, "have been operating in the dark for over four months, unable to get real financial data from Vision. You haven't done a valid month-end closing since this new system started." The new IT director looked quickly at the accounting director, who looked a little sheepish as he explained, "We've never been able to get our reconciliation data." Rick quickly interjected, "It is pretty difficult to get that report out." Joe smiled and continued, "It would be a waste of your money and my time to have you buy my market-research application, because you would not be able to feed it any data either." At this, the marketing director straightened up, looking alarmed. I didn't realize it at the time, but this was the decisive blow that killed Vision.

That night, I took Joe out for drinks and we had a great time as he filled me in on all the undercurrents I had missed or failed to understand. Techies can be pretty pathetic at that stuff, but occasionally folks take pity on us, and I realized what I wanted to be when I grew up.

Private Vision

Another thing that the good folks at Upstart didn't fully realize was that Randy had his own vision. Although he had been introduced to Upstart as an employee of Oracle Consulting, he quickly cut them out of the picture by forming his own company and taking the entire engagement for himself. This is considered extremely bad form in any contracting business, where the basic ethic is "*you dance with whom what brung ya!*" Customer contacts are the life-blood of the IT contracting business, and contracting companies consider it a capital offense to steal a customer contact, and rightly so. As soon as it appeared that Upstart was leaning toward his ideas for the new application, Randy bailed from Oracle and formed his own business, which we'll call "MetaVision."

MetaVision and its founder signed a contract with Upstart to develop the Vision system for Upstart, but MetaVision managed to reserve for itself the source-code rights to resell Vision as it saw fit. This was an extraordinary concession made by Upstart, as works for hire are usually owned by the paying customer. How Randy managed this sweetheart deal is another mystery, but undoubtedly he felt that his design for a completely flexible design for Vision was innovative enough that he could not part with all rights to it. Additionally, since Vision was billed as the "last application ever to be built," Randy could have argued that he was putting himself out of business with this development effort, so he needed to retain rights.

Over the course of the two-year project to develop Vision, Randy charged Upstart $200/hour as the application architect, grossing an estimated $800,000 or so in billings. In addition, he hired two junior programmers and billed them to Upstart for $100/hour, grossing another $800,000 or so in billings over two years and then some. Since the two junior programmers were being paid around $60,000/year apiece (good money in those days!), Randy was doing quite well for himself, making over $500,000/year. For the few months that Vision was in production before the entire development team was fired, Randy continued to bill Upstart at the same pace, and even attempted to sue for breach of contract after he was fired.

All in all, Randy had found a lucrative gravy train, allowing him to bill well over $1.5 million over two years, while he got to experiment with his new ideas, and then keep the fruits of his labors to resell to other customers to boot. After the Vision project was cancelled, Randy dissolved MetaVision and went trekking in the Himalayas for several months. In the ten years that have elapsed since, I have not seen or heard anything about him, although I have been watching. I recently found the slides from a presentation from someone with his name, dated in 1998, on the practical aspects and technical challenges of operating a porn website. It seemed to fit, and the author was quite adamant about the advantages of copyrighting images even if they were stolen. He also recommended strongly that content management be as automated and flexible as possible, touting the software he developed for his own website. I'm sure that he's making as much money as ever.

Looking back at Vision

After the decision was made to replace Vision in January 1994, Upstart conducted a search for a replacement application. Analysis was initiated to rewrite Vision into a more conventional data model, but Upstart executives squashed "Vision II" before it ever got off the ground.

The software selection process was a repetition of the original selection process that had occurred two-and-a-half years earlier, which had resulted in the decision to develop Vision. Instead, this time the runner-up, a tried-but-true yet aged application based on Digital VAX-VMS and RMS (instead of Oracle) was selected. It was the complete antithesis of Vision and I was sure that Randy was aware of it.

While the "new" application was being deployed, I had imagined that I would disengage from Upstart and find a new engagement, but a fitting punishment was being devised for me also. Upstart and BigMedia, Inc. arranged to have me manage Vision as its new database administrator until it was retired, even though I had no prior experience as a DBA. Thus, my career swerved into database administration, away from application development, a detour I have yet to correct.

In November 1994, the new application finally went "live" and Vision was decommissioned. During those nine months, I learned hard lessons in 24x7 production support, managing "hot" backups and archived redo log files, managing space in the database as well as file systems and logical volumes, working with Oracle Support and applying patches, and database recovery at 4:00 a.m. I worked 20-hour days, 120-hour weeks, slept under desks to avoid direct lighting, ate the worst that vending machines could offer, and picked up all kinds of other bad habits that persist to this day.

The main lesson that I took away from this fiasco was that sometimes the cleverest people made the most fundamental errors, losing sight of the ultimate goal. The ultimate goal for the Vision system was to create an integrated order-entry and customer-service application to meet the needs of a fast-growing company, not to change the world of IT and usher in a brave new world of programming.

I sometimes wish that I too had heeded Joe's admonition to "RUN LIKE HELL" but at other times I realize that this job really does beat working for a living.

Author profile: Tim Gorman

Tim Gorman has worked in information technology (IT) with relational databases since 1984, as an Oracle database programmer since 1990, and as an Oracle database administrator since 1993. He is an independent consultant (http://www.EvDBT.com) specializing in performance tuning, database administration, high-availability solutions, troubleshooting and crisis management, software development, and data warehousing. He has co-authored three books, "Oracle8 Data Warehousing", "Essential Oracle8i Data Warehousing" (both from John Wiley & Sons) and "Oracle Insights: Tales of the Oak Table" (from Apress).

D of ACID: NoSQL solutions

A humorous look on fanboyz. Youtube link here

“If you need to build a globally distributed search engine that manages petabytes of data, fine, build your own database. But if you're like 99.9% of companies you can probably get by very well with something like MySQL and maybe memcache."

Transcript at: http://www.mongodb-is-web-scale.com/

A in ACID: Overcommiting application

Source: www.cmg.org/proceedings/2009/9135.pdf

Copyright Alexander Podelko

“There were the same symptoms for Oracle Database as for SQL Server: after a certain point, only response times increase as load increases, not throughput. The maximum was an average of 294 requests / sec.”

“And there were plenty of resources available: with 26.1% CPU utilization on the application server and 7.5% CPU utilization on the database server.”

“There were some indications that I/O was high: Average Disk Queue Length was about 2 for the physical disk with the Oracle data files.
However other important I/O counters didn't indicate that I/O was the bottleneck: %Idle Time was 15.4% (so calculating the real disk utilization as 100-%Idle Time we get 84.6%) and Avg. Disk Sec/Transfer was 6 ms (3ms for read and 10 ms for write).”

“Considering the complains of Oracle diagnostics tools about the high number of commits, one question was how commits were implemented. According, for example, to Coskan Gundogar's blog http://coskan.wordpress.com/2007/03/14/autocommit-with-jdbc-connections/ it should be set properly at JDBC level – it is autocommit by default. Having autocommit for each SQL request would kill any performance advantage of batch asynchronous processing.”

To avoid this situation you must set to off the autocommit option of your JDBC connection

connection conn= DriveManager.getConnection (“jdbc:oracle:oci:database”,”hr”,”hr”);

conn.setAutoCommit (false);

Goodbye, CouchDB

May 10th, 2012 by Steven Hazel

Here at Sauce Labs, we recently celebrated the completion of a significant project to improve our service uptime and reliability, as we transitioned the last of our CouchDB databases to MySQL. We'd outgrown CouchDB, to the point that a majority of our unplanned downtime was due to CouchDB issues, so wrapping up this migration was an important milestone for us.

CouchDB was a very positive experience at first, and its reliability isn't out of bounds for a database that is, after all, only on version 1.2. But our service is very sensitive to reliability issues, and ultimately we decided that this transition was the most efficient path forward for us.

Once we decided on MySQL (specifically, we're now using Percona, with its InnoDB-based XtraDB storage engine), we rearchitected our DB abstraction layer and one by one migrated all our databases, large and small. Our uptime was dramatically improved over the past couple months as we worked through the migration, and performance was slightly improved in the bargain.

This post describes our experience using CouchDB, and where we ran into trouble. I'll also talk about how this experience has affected our outlook on NoSQL overall, and how we designed our MySQL setup based on our familiarity with the positive tradeoffs that came with using a NoSQL database.

First, how did we get into trouble?

Everything Was Going to be Great

When we first started Sauce Labs back in 2008, we thought we were building something very different from the service we run today. We were excited to try a NoSQL db, having spent too many years using MySQL in ways that the designers of relational databases never imagined. CouchDB seemed well suited to our needs.

Our original product design featured a REST API for storing data on behalf of our customers, and the plan was to drop almost straight through to CouchDB's already RESTful API. This let us get a prototype up and running in a hurry. It didn't matter that CouchDB was new and not yet hardened by a lot of real-world usage, we thought, because our database I/O needs were meager, our app was naturally horizontally scalable, and our product was fault-tolerant. We could easily bridge any reliability gap just by keeping replicas and retrying things when they failed. What could go wrong?

What Could Go Wrong

As our little company grew, and we learned about the problems our customers faced, our product underwent several major changes. It made less sense over time to partition data so strictly by user. We came to rely more on database I/O performance.

In general, we found ourselves using CouchDB very differently from how we'd originally imagined we would, and much more the way most web apps use databases. That was still a reasonable way to use CouchDB, but the margin of safety we thought we had when choosing it slowly evaporated as our product evolved. And, as it turned out, we needed that margin of safety.

Sauce Labs' service is more sensitive to reliability issues than the average web app. If we fail a single request, that typically fails a customer's test, and in turn their entire build. Over time, reliability problems with CouchDB became a serious problem. We threw hardware at it. We changed the way we used CouchDB. We changed our software to rely much less on the database and do much less database I/O. Finally, we decided the best next step was to switch.

Again, none of this speaks particularly badly about CouchDB. It's a young database, and reliability and performance issues are to be expected. And in a way it's too bad, because we have no love for classical relational databases. We're convinced that NoSQL is the future. We're just not convinced it's the present.

Some Things We Really Liked about CouchDB

- No schemas. This was *wonderful*. What are schemas even for? They just make things hard to change for no reason. Sometimes you do need to enforce constraints on your data, but schemas go way too far. With CouchDB, adding new fields to documents was simple and unproblematic.
- Non-relational. Relational databases grew up solving problems where data integrity was paramount and availability was not a big concern. They have a lot of features that just don't make sense as part of the database layer in the context of modern web apps. Transactional queries with 6-way joins are tempting at first, but just get you into trouble when you need to scale. Preventing them from day one is usually easy.
- No SQL. It's 2012, and most queries are run from code rather than by a human sitting at a console. Why are we still querying our databases by constructing strings of code in a language most closely related to *freaking COBOL*, which after being constructed have to be parsed for every single query?

  SQL in its natural habitat

  Things like SQL injection attacks simply should not exist. They're a consequence of thinking of your database API as a programming language instead of a protocol, and it's just nuts that vulnerabilities still result from this poorly thought out 1970s design today.

- HTTP API. Being able to query the DB from anything that could speak HTTP (or run curl) was handy.

- Always-consistent, append-only file format. Doing DB backups just by copying files was simple and worry-free.
- Javascript as a view/query language was familiar and useful.
- Indexes on arbitrary calculated values seemed like a potentially great feature. We never ran into a really brilliant way to use them, though it was straightforward to index users by email domain name.
- Finally, it's worth pointing out that even under stress that challenged its ability to run queries and maintain indexes, CouchDB never lost any of our data.

The Problems We Encountered with CouchDB

Availability:

- In our initial setup, slow disk performance made CouchDB periodically fail all running queries. Moving to a much faster RAID setup helped, but as load increased, the problems came back. Percona is not breaking a sweat at this load level: our mysqld processes barely touch the CPU, we have hardly any slow queries, the cache is efficient enough that we're barely doing disk reads, and our write load is a very comfortably small percentage of the capacity of our RAID 10 arrays.
- Views sometimes lost their indexes and failed to reindex several times before finally working. Occasionally they'd get into a state in which they'd just reindex forever until we deleted the view file and restarted CouchDB. For our startup, this was agony. Surprise reindexing exercises were the last thing we needed as a small team already taking on a giant task list and fighting to impress potential big customers.
- Broken views sometimes prevented *all* views from working until the poison view file was removed, at which point view indexing restarted its time-consuming and somewhat unreliable work. I don't know how many times one of us was woken up by our monitoring systems at 4am to learn that our service was down because our database had suddenly become a simple key/value store without our permission.
- Compaction sometimes silently failed, and occasionally left files behind that had to be removed to make it work again. This led to some scary situations before we tightened up our disk usage alarms, because we discovered this when we had very little space left in which to do the compaction.
- In earlier versions, we ran into three or four different bugs relating to file handle usage. Bug reports led to quick fixes for these, and these problems were all gone by version 1.0.2.

Performance:

- There's really only one thing to say here, and that's that view query performance in CouchDB wasn't up to the level of performance we'd expect from roughly equivalent index-using queries in MySQL. This was not a huge surprise or a huge problem, but wow, a lot of things are quicker now, and our database machines are a lot less busy.

Maintenance headaches:

- When CouchDB fails, it tends to fail all running queries. That includes replication and compaction, so we needed scripts to check on those processes and restart them when necessary.
- View indexes are only updated when queried — insertion does not update the index. That means you have to write a script to periodically run all your views, unless you want them to be surprisingly slow when they haven't been queried in a while. In practice we always preferred view availability to any performance boost obtained by not updating indexes on insertion, but writing reliable scripts to keep view indexes up to date was tricky.
- The simple copying collector used for compaction can spend a lot of time looking at long-lived documents. That's particularly bad news when a database has both long-lived and short-lived documents: compaction takes a long time, but is badly needed to keep disk usage under control. Plus, you have to run compaction yourself, and monitoring to make sure it's working is non-trivial. Compaction should be automatic and generational.

Unfulfilled promise:

- CouchDB's design looks perfect for NoSQL staple features like automatic sharding, but this is not something it does.
- What is the point of mapreduce queries that can only run on a single machine? We originally assumed this feature was headed toward distributed queries.
- It was never clear to us what the CouchDB developers considered its core use cases. We saw development focus on being an all-in-one app server, and then on massive multi-direction replication for mobile apps. Both interesting ideas, but not relevant to our needs.

(We're told that a few of these issues have already been addressed in the recently-released CouchDB 1.2.)

We were able to work with CouchDB's performance, and over time we learned how to script our way around the maintenance headaches. And while we were worried that CouchDB seemed to be gravitating toward use cases very different from our own, it was the availability issues that eventually compelled us to switch. We talked about a number of possible choices and ultimately settled on a classic.

MySQL, the Original NoSQL Database

So why not switch to another document-oriented database like MongoDB, or another NoSQL database? We were tempted by MongoDB, but after doing some research and hearing a number of mixed reviews, we came to the conclusion that it's affected by a lot of the same maturity issues that made CouchDB tough for us to work with. Other NoSQL databases tended to be just as different from CouchDB as MySQL — and therefore just as difficult to migrate to — and a lot less well known to us. Given that we had experience with MySQL and knew it was adequate for our needs, it was hard to justify any other choice.

We're familiar with MySQL's downsides: among other things, it's terrible to configure (hint: the most important setting for performance is called `innodb_buffer_pool_size`), and its query engine, besides being SQL-oriented, guesses wrong about how to perform queries all the time. Experienced MySQL users expect to write a lot of `FORCE INDEX` clauses.

The InnoDB storage engine, on the other hand, is pretty great overall. It's been hardened by heavy use at some of the biggest internet companies over the past decade, dealing with workloads that resemble those faced by most modern developers. At the lowest level, almost any database is built on the same fundamentals of B-trees, hashes, and caching as InnoDB. And with respect to those fundamentals, any new database will have to work very hard to beat it on raw performance and reliability in real-world use cases. But maybe they won't all have to: Percona's forward-thinking key/value interface is a good example of how the solid InnoDB storage engine might make its way into true NoSQL architectures.

In switching to MySQL, we treated it as much like a raw storage engine as we reasonably could. So now we're back to using MySQL in the way that inspired so much NoSQL work in the first place:

- We ported our CouchDB model layer to MySQL in a way that had relatively minor impacts on our codebase. From most model-using code, using MySQL looks exactly the same as using CouchDB did. Except it's faster, and the DB basically never fails.
- We don't use foreign keys, or multi-statement transactions, or, so far, joins. When we need to horizontally scale, we're ready to do it. (But it'll be a while! Hardware has gotten more powerful since the days when sharding was invented, and these days you can go a long way with just a single write master.)
- We have a TEXT column on all our tables that holds JSON, which our model layer silently treats the same as real columns for most purposes. The idea is the same as Rails' ActiveRecord::Store. It's not super well integrated with MySQL's feature set — MySQL can't really operate on those JSON fields at all — but it's still a great idea that gets us close to the joy of schemaless DBs.

It's a nice combination of a proven, reliable database storage engine with an architecture on top of it that gives us a lot of the benefits of NoSQL databases. A couple months into working with this setup, we're finding it pretty hard to argue with this best-of-both-worlds approach.

COMMENTS

1. The first 3 bullet points about why they liked CouchDB via criticizing relational databases show just how little Sauce Labs really knows about relational databases, and their general lack of maturity and wisdom in the field of computer science.

   "No schemas. This was wonderful. What are schemas even for? They just make things hard to change for no reason." Right. I'm sure that all the work that Edgar F. Codd did to invent relational database schemas was just to make them hard to change, to piss off the Sauce Labs crew some 40 years later. You noobs obviously know nothing about relational calculus. http://en.wikipedia.org/wiki/Codd%27s_theorem

"Non-relational. Relational databases grew up solving problems where data integrity was paramount and availability was not a big concern. They have a lot of features that just don't make sense as part of the database layer in the context of modern web apps." More of the same ignorance. Yeah, modern web apps don't need stuff like ACID data persistence. Who cares if we lose the data, or if the data we get back is wrong but without any indication of error?

The fact that little MySQL makes your data way more "available" than CouchDB could seems to make that statement even more ludicrous.

"No SQL. It's 2012, and most queries are run from code rather than by a human sitting at a console. Why are we still querying our databases by constructing strings of code in a language most closely related to freaking COBOL, which after being constructed have to be parsed for every single query?" Guess you clueless beginners have not heard of prepared statements — parsed once. And you conflate SQL with relational database — since there are relational databases that do not use standard SQL. Although why you'd not use one just other than your irrational fear of "Teh SQL!" it's hard to imagine. Better start coding in binary machine language, because all that text you write as code is "strings of code in a language most closely related to freaking [some older language]".

N00bs. Punks. Come back and make this argument in 20 years.

2. You have to look at your problem space before deciding on a database technology. It's pretty obvious from Sauce Labs statements on how they use MySql that they don't need a relational db at all. They needed a reliable document store that scales without being I/O bound. They won't run into any of the issues that cause most people to look at NoSql over relational – i.e. range based, ad-hoc or aggregate queries over very large datasets. These are the some of the primary driving forces behind map/reduce and NoSql. They could probably get by with a really large SAN, fronted by Redis to hold indexes. Or if they can do hosted – go directly to DynamoDb.

   Seems to me the issue in Sauce Labs case was mischaracterization of the problem to be solved and selecting a solution without defining the requirements.

If all these new DBMS technologies are so scalable, why are Oracle and DB2 still on top of TPC-C? A roadmap to end their dominance.

Source: http://dbmsmusings.blogspot.co.uk/2012/05/if-all-these-new-dbms-technologies-are.html

(This post is coauthored by Alexander Thomson and Daniel Abadi)
In the last decade, database technology has arguably progressed furthest along the scalability dimension. There have been hundreds of research papers, dozens of open-source projects, and numerous startups attempting to improve the scalability of database technology. Many of these new technologies have been extremely influential---some papers have earned thousands of citations, and some new systems have been deployed by thousands of enterprises.

So let's ask a simple question: If all these new technologies are so scalable, why on earth are Oracle and DB2 still on top of the TPC-C standings? Go to the TPC-C Website with the top 10 results in raw transactions per second. As of today (May 16th, 2012), Oracle 11g is used for 3 of the results (including the top result), 10g is used for 2 of the results, and the rest of the top 10 is filled with various versions of DB2. How is technology designed decades ago still dominating TPC-C? What happened to all these new technologies with all these scalability claims?

The surprising truth is that these new DBMS technologies are not listed in the TPC-C top ten results **not** because that they do not care enough to enter, but rather because they would not win if they did.

To understand why this is the case, one must understand that scalability does not come for free. Something must be sacrificed to achieve high scalability. Today, there are three major categories of tradeoff that can be exploited to make a system scale. The new technologies basically fall into two of these categories; Oracle and DB2 fall into a third. And the later parts of this blog post describes research from our group at Yale that introduces a fourth category of tradeoff that provides a roadmap to end the dominance of Oracle and DB2.

These categories are:

(1) **Sacrifice ACID for scalability.** Our previous post on this topic discussed this in detail. Basically we argue that a major class of new scalable technologies fall under the category of "NoSQL" which achieves scalability by dropping ACID guarantees, thereby allowing them to eschew two phase locking, two phase commit, and other impediments to concurrency and processor independence that hurt scalability. All of these systems that relax ACID are immediately ineligible to enter the TPC-C competition since ACID guarantees are one of TPC-C's requirements. That's why you don't see NoSQL databases in the TPC-C top 10---they are immediately disqualified.

(2) **Reduce transaction flexibility for scalability.** There are many so-called "NewSQL" databases that claim to be both ACID-compliant and scalable. And these claims are true---to a degree. However, the fine print is that they are only linearly scalable when transactions can be completely isolated to a single "partition" or "shard" of data. While these NewSQL databases often hide the

complexity of sharding from the application developer, they still rely on the shards to be fairly independent. As soon as a transaction needs to span multiple shards (e.g., update two different user records on two different shards in the same atomic transaction), then these NewSQL systems all run into problems. Some simply reject such transactions. Others allow them, but need to perform two phase commit or other agreement protocols in order to ensure ACID compliance (since each shard may fail independently). Unfortunately, agreement protocols such as two phase commit come at a great scalability cost (see our 2010 paper that explains why). Therefore, NewSQL databases only scale well if multi-shard transactions (also called “distributed transactions” or “multi-partition transactions”) are very rare. Unfortunately for these databases, TPC-C models a fairly reasonable retail application where customers buy products and the inventory needs to be updated in the same atomic transaction. 10% of TPC-C New Order transactions involve customers buying products from a “remote” warehouse, which is generally stored in a separate shard. Therefore, even for basic applications like TPC-C, NewSQL databases lose their scalability advantages. That’s why the NewSQL databases do not enter TPC-C results --- even just 10% of multi-shard transactions causes their performance to degrade rapidly.

(3) **Trade cost for scalability**. If you use high end hardware, it is possible to get stunningly high transactional throughput using old database technologies that don’t have shared-nothing horizontally scalability. Oracle tops TPC-C with an incredibly high throughput of 500,000 transactions per second. There exists no application in the modern world that produces more than 500,000 transactions per second (as long as humans are initiating the transactions---machine-generated transactions are a different story). Therefore, Oracle basically has all the scalability that is needed for human scale applications. The only downside is cost---the Oracle system that is able to achieve 500,000 transactions per second costs a prohibitive $30,000,000!

Since the first two types of tradeoffs are immediate disqualifiers for TPC-C, the only remaining thing to give up is cost-for-scale, and that’s why the old database technologies are still dominating TPC-C. None of these new technologies can handle both ACID and 10% remote transactions.”
End of quote

---

(7) Scaling the Database: Data Types – by Michelle Ufford

It was 2 AM on February 4, 2008 and I was on the largest conference call in my life. There were dozens of people listening in, from developers to executives. The Super Bowl had concluded hours before and our Vice President of Marketing was asking *me* if the query was done yet so that everyone could call it a night. The problem was, the query had been running for hours and I had no idea when it would be done.

The query in question was running against a legacy SQL Server database that had grown to 2.5 terabytes and regularly reached 5,000 transactions per second. The database had been hastily developed the weekend before Super Bowl 2006 to facilitate reporting. Putting together a new, complex system just days before a major event is a pretty impressive feat. Unfortunately, best practices are rarely on the radar when developers are in "git 'er done" mode. Two years and much added functionality later, it was time for a major redesign.

Add to that, this database was on the receiving end of some pretty heavy Super Bowl activity. Go Daddy's 2008 Super Bowl commercial employed a marketing technique that was revolutionary at the time: drive visits to our website by airing a teaser and enticing the viewer to come to the website to see the commercial's too-hot-for-TV conclusion. And it worked! We experienced one of the largest advertising spikes in the history of Super Bowl ads. This was great news for the company, but being downhill of that kind of activity wreaked havoc on the database server. While customer-facing systems handled the spike admirably, it took 3.5 hours before all pending inserts were written to the analytical database.

Let's go back and examine what the query was trying to do. It was joining three large tables, the smallest of which had 200 million rows and the largest of which had 4.5 billion rows. The tables did not have partitioning or covering indexes, and all were clustered on a non-sequential GUID. This meant that we were essentially trying to scan and join every row in three very large tables to see if the results matched the aggregate condition. Later, I also discovered that the tables were 99.99% fragmented. No wonder everything was taking so long!

Fast forward to Super Bowl 2009. What a game! Santonio Holmes scored a touchdown in the last 35 seconds to win it for the Steelers. What's more, the game set a record for viewership. With a U.S. audience of 98.7 million viewers, it was the most watched Super Bowl in history (at the time). That played no small part in Go Daddy setting its own records, including new customers and orders. This time, however, the database performed a little better:

	2008	**2009**
Time to Catch Up on INSERTs	3.5 hours	20 minutes
Peak Transactions per Second	15,000	27,000
Time To Complete Complex Query	2.5 hours	< 1 minute
When I Was Able to Leave the Office	2:30 AM (next day)	9:00 PM (same day)

As you can see in the chart above, the database hit a sustained peak of 27,000 transactions *per second*. According to Microsoft SQLCAT, this was among the highest transactional rates *in the world* for a SQL Server instance at that time.

So what changed? Well, it certainly wasn't the hardware. The database was still running on the same commodity Dell PowerEdge 6850 server and NetApp FAS3040 array from the previous year. It also wasn't the application tier. They added a significant amount of new functionality but left the old code largely alone. The fact is, we were able to achieve an extraordinary boost in performance just by redesigning the database using best practices.

This post is the first in a series exploring database scalability and very large database (VLDB) performance tuning. In this series, we'll discuss performance tuning strategies that we have successfully implemented at Go Daddy and, when possible, discuss some of the results I've seen firsthand.

A word of warning: I'm going to assume this isn't your first database rodeo. Also, I really, *really* like code, so you can be sure that we'll be getting down and dirty in T-SQL whenever possible. 😉

Data Types

Although few realize it, one of the biggest decisions a database developer makes is the data types he or she chooses. A well-intentioned but ill-conceived decision to store a critical date or numeric column as a character column can cause years of pain for developers and waste money for a business. One Fortune 500 company I worked with experienced this firsthand. For reasons known only to the original developer(s), a critical column, ProductNumber, was defined as a CHAR(4). Finally, after 20+ years of custom integrations and creative workarounds, the company had no option left but to upgrade the data type. What followed was a very long and expensive modification to nearly every piece of software in the company. Had the developer(s) decided to use a standard integer-based data type, the company would have been able to save a lot of money in its software integrations and avoid a costly data type conversion process.

Back to Go Daddy in February of 2006. In the absence of clear requirements and not having the time to spend on researching it, the developers chose to use the most common data types: all integer columns were stored as a 4-byte INT, all date columns were an 8-byte DATETIME, and all string columns were stored as an NVARCHAR column. 16-byte UNIQUEIDENTIFIER columns were also liberally used to store GUIDs. Web developers generally like GUIDs, or *globally unique identifiers*. They're a great way to generate a unique value when a central issuing authority is not available. These are useful for all sorts of things, from uniquely identifying a person to maintaining a unique web session.

Let's look at one of the largest tables in the database. For security reasons, I'll replace actual column names with generic names.

```
CREATE TABLE dbo.BadTable
(
myGuid1   UNIQUEIDENTIFIER
, myGuid2   UNIQUEIDENTIFIER
, myDate1   DATETIME
, myDate2   DATETIME
, myID1     INT
, myID2     INT
, myID3     INT
, myID4     INT
, myID5     INT
, myID6     INT
, myText1   NVARCHAR(2)
, myText2   NVARCHAR(10)
, myText3   NVARCHAR(12)
, myText4   NVARCHAR(10)
, myText5   NVARCHAR(15)
, myText6   NVARCHAR(15)
, myText7   NVARCHAR(100)
, myText8   NVARCHAR(255)
, myText9   NVARCHAR(255)

CONSTRAINT PK_BadTable
PRIMARY KEY (myGuid1)
);
```

The developers chose to use NVARCHAR because they knew that Go Daddy does business internationally and would need to support international characters in some of its columns. Not having the time to identify which columns would need to support Unicode, the developers made the understandable decision to make everything Unicode. The problem with this is that character columns are usually some of the widest in a table, and Unicode consumes twice as much space as non-Unicode.

While we're on the topic of space, let's take a look at the storage requirements of the data types in the *dbo.BadTable* example above.

Data Type	**Values Supported**	**Storage Requirements**
varchar(*n*)	A variable-length non-Unicode alpha-numeric string	actual length + 2 bytes
nvarchar(*n*)	A variable-length Unicode alpha-numeric string	((actual length) * 2) + 2 bytes
char(*n*)	A fixed-length non-Unicode alpha-numeric string	*n* defined length
tinyint	0 to 255	1 byte
smallint	up to -/+ 32,767	2 bytes
int	up to -/+ 2,147,483,647	4 bytes
bigint	up to -/+ 9,223,372,036,854,775,807	8 bytes
smalldatetime	1900-01-01 to 2079-06-06, minute precision	4 bytes
datetime	1753-01-01 to 9999-12-13, sub-second precision	8 bytes
uniqueidentifier	36-character string in the form of xxxxxxxx- xxxx- xxxx- xxxx- xxxxxxxxxxxx, i.e. 6F9619FF-8B86-D011-B42D-00C04FC964FF	16 bytes

You might be saying, "Why should I care about 2 bytes versus 4 bytes? Storage is cheap! I can buy a 1 TB drive for $100." Well, all storage is not created equal, and the biggest bottleneck in most databases is IO. This leads me to the biggest performance tuning tip I can offer: **The more data you can fit on a data page, the better your database will perform.** It's really that simple. If you can maximize the storage of data on a page, you can get great database performance.

Using the table of data types above, let’s do some math to figure out how much space *dbo.BadTable* will consume. To do this, we’ll need to know the average length of each variable-length column:

Column	**Data Type**	**Average Length**	**Bytes**
myGuid1	UNIQUEIDENTIFIER		16
myGuid2	UNIQUEIDENTIFIER		16
myDate1	DATETIME		8
myDate2	DATETIME		8
myID1	INT		4
myID2	INT		4
myID3	INT		4
myID4	INT		4
myID5	INT		4
myID6	INT		4
myText1	NVARCHAR(2)	2	6
myText2	NVARCHAR(10)	8	18
myText3	NVARCHAR(12)	6	14
myText4	NVARCHAR(10)	8	18
myText5	NVARCHAR(15)	6	14
myText6	NVARCHAR(15)	9	20
myText7	NVARCHAR(100)	7	16
myText8	NVARCHAR(255)	32	66
myText9	NVARCHAR(255)	45	92
Total			336

Tip: *You can easily identify the average length of a column by selecting AVG(LEN(columnName)) on a table. Because it will scan an entire table, try to avoid running this type of query in peak hours.*

Based on the data types specified, *dbo.BadTable* consumes 336 bytes of storage per row. While I won’t go into the particulars, you also need to be aware that there are an additional 11 bytes of overhead for each row in this particular table. Thus, in actuality, each row inserted into this table consumes 347 bytes of space.

If you are using SQL Server 2005 or later, you can see this for yourself using the following code. This code will insert a row that simulates the typical data footprint of the table and will output the data page of *dbo.BadTable* using the undocumented DBCC PAGE command.

```
INSERT INTO dbo.BadTable
(
myGuid1
, myGuid2
, myDate1
, myDate2
, myID1
, myID2
, myID3
, myID4
, myID5
, myID6
, myText1
, myText2
, myText3
, myText4
, myText5
, myText6
, myText7
, myText8
, myText9
)
SELECT NEWID()
, NEWID()
, GETDATE()
, GETDATE()
, 2500
, 1150
, 185
, 11000
, 75
, 175
, REPLICATE('X', 2)
, REPLICATE('X', 8)
, REPLICATE('X', 6)
, REPLICATE('X', 8)
, REPLICATE('X', 6)
, REPLICATE('X', 9)
, REPLICATE('X', 7)
, REPLICATE('X', 32)
, REPLICATE('X', 45);

DECLARE @dbName SYSNAME = DB_NAME()
, @fileID INT
, @pageID INT;

SELECT @fileID = CONVERT (VARCHAR(6), CONVERT (INT,
SUBSTRING (au.first_page, 6, 1) +
SUBSTRING (au.first_page, 5, 1)))
, @pageID = CONVERT (VARCHAR(20), CONVERT (INT,
SUBSTRING (au.first_page, 4, 1) +
SUBSTRING (au.first_page, 3, 1) +
```

```sql
SUBSTRING (au.first_page, 2, 1) +
SUBSTRING (au.first_page, 1, 1)))
FROM sys.indexes AS i
JOIN sys.partitions AS p
ON i.[object_id] = p.[object_id]
AND i.index_id = p.index_id
JOIN sys.system_internals_allocation_units AS au
ON p.hobt_id = au.container_id
WHERE p.[object_id] = OBJECT_ID('dbo.BadTable')
AND au.first_page > 0x000000000000;

DBCC TraceOn (3604);
DBCC PAGE (@dbName, @fileID, @pageID, 3);
DBCC TraceOff (3604);
```

```
(1 row(s) affected)
DBCC execution completed. If DBCC printed error messages, contact your system
administrator.

PAGE: (1:283)

BUFFER:

BUF @0x0000000278751D40

bpage = 0x000000026C7FE000 bhash = 0×0000000000000000 bpageno = (1:283)
bdbid = 5 breferences = 1 bcputicks = 0
bsampleCount = 0 bUse1 = 39082 bstat = 0xb
blog = 0x15ab215a bnext = 0×0000000000000000

PAGE HEADER:

Page @0x000000026C7FE000

m_pageId = (1:283) m_headerVersion = 1 m_type = 1
m_typeFlagBits = 0×0 m_level = 0 m_flagBits = 0×8000
m_objId (AllocUnitId.idObj) = 88 m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594043695104
Metadata: PartitionId = 72057594039238656 Metadata: IndexId = 1
Metadata: ObjectId = 629577281 m_prevPage = (0:0) m_nextPage = (0:0)
pminlen = 76 m_slotCnt = 2 m_freeCnt = 7398
m_freeData = 790 m_reservedCnt = 0 m_lsn = (31:366:2)
m_xactReserved = 0 m_xdesId = (0:0) m_ghostRecCnt = 0
m_tornBits = -940717766 DB Frag ID = 1

Allocation Status
```

```
GAM (1:2) = ALLOCATED SGAM (1:3) = ALLOCATED
PFS (1:1) = 0×60 MIXED_EXT ALLOCATED 0_PCT_FULL DIFF (1:6) =
CHANGED
ML (1:7) = NOT MIN_LOGGED

Slot 0 Offset 0×60 Length 347

Record Type = PRIMARY_RECORD Record Attributes = NULL_BITMAP
VARIABLE_COLUMNS
Record Size = 347
Memory Dump @0x000000001165A060
```

The actual output is considerably longer; however, we only need this first section for the purposes of this example. Look at “Record Size.” This tells you how many bytes a row actually consumes. If you have variable-length columns, you will see the value change for each row. This value *does* include row overhead, so you will notice that the Record Size is 347, which is what I expected to see.

A data page can hold 8192 bytes. 96 bytes are reserved for the header, which leaves 8096 bytes available for data. That means a data page can only hold 23 rows in this table (8096 bytes / 347 bytes), assuming there’s no fragmentation. At the time of Super Bowl 2008, this table had 200 million rows. If you divide 200 million by 23 rows, it took 8,695,652 pages to hold our data. Multiply that by 8192 bytes, and you find that this table consumed 66 GB of storage space.

Now let’s look at this exact same table and select the appropriate data types based on our actual storage needs:

Column	**Values Actually Stored**
myGuid1	GUID
myGuid2	GUID
myDate1	Precision needed to the minute
myDate2	Precision needed to the second (not subsecond)
myID1	0 to 2,500
myID2	0 to 1,150
myID3	0 to 185
myID4	0 to 11,000
myID5	0 to 75
myID6	0 to 175
myText1	Always 2 non-Unicode characters
myText2	Average 8 non-Unicode characters
myText3	Average 6 non-Unicode characters
myText4	Average 8 non-Unicode characters
myText5	Average 6 non-Unicode characters
myText6	Average 9 non-Unicode characters
myText7	Average 7 Unicode characters
myText8	Average 32 Unicode characters
myText9	Average 45 non-Unicode characters

```
CREATE TABLE dbo.GoodTable
(
myNewID    INT IDENTITY(1,1)
, myGuid1    UNIQUEIDENTIFIER
, myGuid2    UNIQUEIDENTIFIER
, myDate1    SMALLDATETIME
, myDate2    SMALLDATETIME
, myID1      SMALLINT
, myID2      SMALLINT
, myID3      SMALLINT
, myID4      SMALLINT
, myID5      TINYINT
, myID6      TINYINT
, myText1    CHAR(2)
, myText2    VARCHAR(10)
, myText3    VARCHAR(12)
, myText4    VARCHAR(10)
, myText5    VARCHAR(15)
, myText6    VARCHAR(15)
, myText7    NVARCHAR(100)
, myText8    NVARCHAR(255)
, myText9    VARCHAR(255)

CONSTRAINT PK_GoodTable
PRIMARY KEY (myNewID)
);
```

Let’s take a minute to discuss some of the decisions made. First, notice that I added a new column, *myNewID*, to the table. This adds 4 bytes of space but serves a very valuable tuning purpose that I’ll discuss in a later blog post. UNIQUEIDENTIFIER is the correct data type to store a GUID, so I left those two columns as is. I then determined that only two of the string columns actually need to support Unicode, so I left those two columns alone as well. All of the other data types were changed. Why? The date columns were changed from DATETIME to SMALLDATETIME because I only needed minute-level precision. Also, 3 of my ID columns needed to store a value over 255 (the cutoff for a TINYINT) and under 2.1 billion (the cutoff for an INT), making SMALLINT an easy decision. But why did I choose to make *myID3*, which only needs to store values between 0 and 185, a SMALLINT instead of a TINYINT? When choosing the right data type, you need to choose the smallest data type that makes sense for the business. This ends up being equal parts art and science. My rationale is this: if I was already at 185 after only a year of data collection, there’s a good chance I would exceed the value limit of 255 sometime in the next year. Everyone has a different guideline, but mine is to allow for 3-5 years’ worth of reasonable growth in the business and application. Lastly, I was able to cut the storage requirements for all non-Unicode columns in half by switching from NVARCHAR to VARCHAR.

Column	**Data Type Needed**	**Average Length**	**Bytes**
myNewID	INT IDENTITY(1,1)		4
myGuid1	UNIQUEIDENTIFIER		16
myGuid2	UNIQUEIDENTIFIER		16
myDate1	SMALLDATETIME		4
myDate2	SMALLDATETIME		4
myID1	SMALLINT		2
myID2	SMALLINT		2
myID3	SMALLINT		2
myID4	SMALLINT		2
myID5	TINYINT		1
myID6	TINYINT		1
myText1	CHAR(2)	2	2
myText2	VARCHAR(10)	8	10
myText3	VARCHAR(12)	6	8
myText4	VARCHAR(10)	8	10
myText5	VARCHAR(15)	6	8
myText6	VARCHAR(15)	9	11
myText7	NVARCHAR(100)	7	16
myText8	NVARCHAR(100)	32	66
myText9	VARCHAR(255)	45	47
Total			232

The new table only requires 232 bytes to store the exact same data. Note that there is still an overhead of 11 bytes per row in the new table, too. You can see this for yourself, if you like, by re-running the DBCC PAGE code from earlier and replacing *dbo.BadTable* with *dbo.GoodTable*.

At the end of the day, here's how my changes impacted the performance of the table:

	Before	**After**	**% Change**
Defined Bytes Per Row	336	232	-31%
Row Overhead	11	11	0%
Total Bytes Per Row	347	243	-30%
Rows Per Page	23	33	45%
Pages to hold 200m rows	8,695,652	6,060,606	-30%
Table Storage Needs (GB)	66.34	46.24	-30%

By using the right data types, I was able to reduce the size of my table by 30%, which saves space both on my disks and in my backups. Even better, I was able to improve IO by 45%! Not too shabby.

One last thing I'd like to mention. Don't wait until you have a large table before paying attention to data types. Many of those small tables, such as reference or dimension tables, tend to become foreign keys in a much larger table. Therefore, selecting the right data types for those tables is equally important.

Database Lesson #1: Pay attention to your data types. They can have a huge impact on database performance.