The Mother of All Database Normalization Debates on Coding Horror

Wednesday, July 16, 2008 at 5:59AM

Todd Hoff in Database, Strategy

Jeff Atwood started a barn burner of a conversation in Maybe Normalizing Isn't Normal on how to create a fast scalable tagging system. Jeff eventually asks that terrible question: *which is better -- a normalized database, or a denormalized database?* And all hell breaks loose. I know, it's hard to imagine database debates becoming contentious, but it does happen :-) It's lucky developers don't have temporal power or rivers of blood would flow.

Here are a few of the pithier points (summarized):

- Normalization is not magical fairy dust you sprinkle over your database to cure all ills; it often creates as many problems as it solves. (Jeff)
- Normalize until it hurts, denormalize until it works. (Jeff)
- Use materialized views which are tables created and maintained by your RDBMS. So a materialized view will act exactly like a de-normalized table would - except you keep you original normalized structure and any change to original data will propagate to the view automatically. (Goran)
- According to Codd and Date table names should be singular, but what did they know. (Pablo, LOL)
- Denormalization is something that should only be attempted as an optimization when EVERYTHING else has failed. Denormalization brings with it it's own set of problems. You have to deal with the increased set of writes to the system (which increases your I/O costs), you have to make changes in multiple places when data changes (which means

either taking giant locks - ugh or accepting that there might be temporary or permanent data integrity issues) and so on. (Dare Obasanjo)

- What happens, is that people see "Normalisation = Slow", that makes them assume that normalisation isn't needed. "My data retrieval needs to be fast, therefore I am not going to normalise!" (Tubs)
- You can read fast and store slow or you can store fast and read slow. The biggest performance killer is so called physical read. Finding and accessing data on disk is the slowest operation. Unless child table is clustered indexed and you're using the cluster index in the join you will be making lots of small random access reads on the disk to find and access the child table data. This will be slow. (Goran)
- The biggest scalability problems I face are with human processes, not computer processes. (John)
- Don't forget that the fastest database query is the one that doesn't happen, i.e. caching is your friend. (Chris)
- Normalization is about design, denormalization is about optimization. (Peter Becker)
- You're just another knucklehead. (BuggyFunBunny)
- Lets unroll our loops next. RDBMS is about shared *transactional data*. If you really don't care about keeping the data right all the time, then how you store it doesn't matter.(Christog)
- Jeff, are you awake? (wiggle)
- Denormalization may be all well and good, when you need the performance and your system is STABLE enough to support it. Doing this in a business environment is a recipe for disaster, ask anyone who has spent weeks digging through thousands of lines of legacy code, making sure to support the new and absolutely required affiliation_4. Then do the whole thing over again 3 months later when some crazy customer has five affiliations. (Sean)
- Do you sex a cat, or do you gender it? (Simon)
- This is why this article is wrong, Jeff. This is why you're an idiot, in case the first statement wasn't clear enough. You just gave an excuse to be

lazy to someone who doesn't have a clue. (Marcel)

- This is precisely why you never optimize until *after* you profile (find objectively where the bottlenecks are). (TED)
- Another great way to speed things up is to do more processing outside of the database. Instead of doing 6 joins in the database, have a good caching plan and do some simple joining in your application. (superjason)
- Lastly - No one seems to have mentioned that a decently normalized db greatly reduces application refactoring time. As new requirements come along, you don't have have to keep pulling stuff apart and putting back in new configurations of the db, (Ed)
- Keep a de-normalized replica for expensive operations (e.g. reports), Cache Results for repeat queries (Memcache), Partition the database for scalability (vertical or horizontal) (Gareth)
- Speaking from long experience, if you don't normalize, you will have duplicates. If you don't have data constraints, you will have invalid data. If you don't have database relational integrity, you will have orphan "child" records, etc. Everybody says "we rely on the application to maintain that", and it never, never does. (A. Lloyd Flanagan)
- I don't think you can make any blanket statements on normal vs. non-normal form. Like everything else in programming, it all depends on requirements and intended goals. (Wayne)
- De-normalization is for reporting, not for OLTP. (Eric)
- Your six-way join is only needed because you used surrogate instead of natural keys. When you use natural keys, you will discover you need much fewer joins because often data you need is already present as foreign keys. (Leandro)
- What I think is funny is the number of people who think that because they use LINQ or Hibernate they aren't affected by these issues. (Sam)
- You miss the point of normalization entirely. Normalization is about optimizing large numbers of small CrUD operations, and retrieving small sets of data in order to support those crud operations. Think about people modifying their profiles, recording cash registers operations, recording

bank deposits and withdrawals. Denormalization is about optimizing retrieval of large sets of data. Choosing an efficient database design is about understanding which of those operations is more important. (RevMike)

- Multiple queries will hurt performance much less than the multi-join monstrosity above that will return indistinct and useless data. (Chris)
- Cache the generated view pages first. Then cache the data. You have to think about your content- very infrequently will anyone be updating it, it's all inserts. So you don't have to worry about normalization too much. (Matt)
- I wonder if one factor at play here is that it's very easy to write queries for de-normalized data, but it's relatively hard to write a query that scales well to a large data set. (Thomi)
- Denormalization is the last possible step to take, yet the first to be suggested by fools. (Jeremy)
- There is a simple alternative to denormalisation here -- to ensure that the values for a particular user_id are physically clustered. (David)
- The whole issue is pretty simple 99% of the time - normalized databases are write optimized by nature. Since writing is slower then reading and most database are for CRUD then normalizing makes sense. Unless you are doing *a lot* more reading then writing. Then if all else fails (indexes, etc.) then create de-normalized tables (in addition to the normalized ones). (Rob)
- Don't fear normalization. Embrace it. (Charles)
- I read on and discovered all the loons and morons who think they know a lot more then they do about databases. (Paul)
- Put all the indexable stuff into the users table, including zipcode, email_address, even mobile_phone -- hey, this is the future!; Put the rest of the info into a TEXT variable, like "extra_info", in JSON format. This could be educational history, or anything else that you would never search by; If you have specific applications (think facebook), create separate tables for them and join them onto the user table whenever you need.

(Greg)

- How is the data being used? Rapid inserts like Twitter? New user registration? Heavy reporting? How one stores data vs. how one uses data vs. how one collects data vs. how timely must new data be visible to the world vs. should be put OLTP data into a OLAP cube each night? etc. are all factors that matter. (Steve)
- It might be possible to overdo it, but trust me, I have had 20 times the problems with denormalized data than with normalized. (PRMAN)
- Your LOGICAL model should *always* be fully normalized. After all, it is the engine that you derive everything else from. Your PHYSICAL model may be denormalized or use system specific tools (materialized views, cache tables, etc) to improve performance, but such things should be done *after* the application level solutions are exhausted (page caching, data caches, etc.) (Jonn)
- For very large scale applications I have found that application partitioning can go a long way to giving levels of scalability that monolithic systems fail to provide: each partition is specialized for the function it provides and can be optimized heavily, and when you need the parts to co-operate you bind the partitions together in higher level code. (John)
- People don't care what you put into a database. They care what you can get out of it. They want freedom and flexibility to grow their business beyond "3 affiliations". (PRMan)
- People don't care what you put into a database. They care what you can get out of it. They want freedom and flexibility to grow their business beyond "3 affiliations". (Steve)
- I normalise, then have distinct (conceptually transient) denormalised cache tables which are hammered by the front-end. I let my model deal with the nitty-gritty of the fix-ups where appropriate (handy beginUpdate/ endUpdate-type methods mean the denormalised views don't get rebuilt more than necessary). (Mo)
- Stop playing with mySQL. (Jonathan)

- What a horrible, cowboy attitude to DB design. I hope you don't design any real databases. (Dave)
- IOW, scalability is not a problem, until it is. Strip away the scatalogical reference, and all you have is a boring truism. (Yawn)
- Is my Site OLTP? If the answer is yes then Normalize. Is my site OLAP? If the answer is yes then De-Normalize! (WeAreJimbo)
- This is a dangerous article, or perhaps you just haven't seen the number of horrific "denormalised" databases I have. People use these posts as excuses to build some truly horrific crimes against sanity. (AbGenFac)
- Be careful not to confuse a denormalised database with a non-normalised database. The former exists because a previously normailsed database needed to be 'optimised' in some way. The latter exists because it was 'designed' that way from scratch. The difference is subtle, but important. (Bob)

OK, more than a few quotes. There's certainly no lack of passion on the issue!

One thing I would add is to organize your application around an application level service layer rather than allowing applications to access the database directly at a low level. Amazon is a good example of this approach. Many of the denormalization comments have to do with the problems of data inconsistency, which is of course true because that's why normalization exists. Many of these problems can be reduced if there's a single service access point over which to get data. It's when data access is spread throughout an application that we see serious problems.

Related Articles

- Denormalization Patterns by Kenneth Downs
- When Databases Lie: Consistency vs. Availability in Distributed

Systems by Dare Obasanjo

- Stored procedure reporting & scalability by Jason Young
- When Not to Normalize your SQL Database by Dare Obasanjo
