Database development mistakes made by application developers [closed]

Asked 15 years, 9 months ago Modified 6 years, 2 months ago

Viewed 129k times

566

votes

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 13 years ago.

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

What are common database development mistakes made by application developers?

database database-design

Share

edited Sep 13, 2010 at 19:19

community wiki
12 revs, 6 users 40%
Charles Faiga

Near-duplicate of stackoverflow.com/questions/346659/…
– dkretz Mar 8, 2009 at 4:46

Comments disabled on deleted / locked posts / reviews

40 Answers

Sorted by: Highest score (default)

1 2 Next

1001
votes

✓

1. Not using appropriate indices

This is a relatively easy one but still it happens all the time. Foreign keys should have indexes on them. If you're using a field in a `WHERE` you should (probably) have an index on it. Such indexes should often cover multiple columns based on the queries you need to execute.

2. Not enforcing referential integrity

Your database may vary here but if your database supports referential integrity--meaning that all foreign keys are guaranteed to point to an entity that exists--you should be using it.

It's quite common to see this failure on MySQL databases. I don't believe MyISAM supports it. InnoDB does. You'll find people who are using MyISAM or those that are using InnoDB but aren't using it anyway.

More here:

- How important are constraints like NOT NULL and FOREIGN KEY if I’ll always control my database input with php?
- Are foreign keys really necessary in a database design?
- Are foreign keys really necessary in a database design?

3. Using natural rather than surrogate (technical) primary keys

Natural keys are keys based on externally meaningful data that is (ostensibly) unique. Common examples are product codes, two-letter state codes (US), social security numbers and so on. Surrogate or technical primary keys are those that have absolutely no meaning outside the system. They are invented purely for identifying the entity and are typically auto-incrementing fields (SQL Server, MySQL, others) or sequences (most notably Oracle).

In my opinion you should **always** use surrogate keys. This issue has come up in these questions:

- How do you like your primary keys?
- What's the best practice for primary keys in tables?
- Which format of primary key would you use in this situation.
- Surrogate vs. natural/business keys
- Should I have a dedicated primary key field?

This is a somewhat controversial topic on which you won't get universal agreement. While you may find some people, who think natural keys are in some situations OK, you won't find any criticism of surrogate keys other than being arguably unnecessary. That's quite a small downside if you ask me.

Remember, even countries can cease to exist (for example, Yugoslavia).

4. Writing queries that require `DISTINCT` to work

You often see this in ORM-generated queries. Look at the log output from Hibernate and you'll see all the queries begin with:

```
SELECT DISTINCT ...
```

This is a bit of a shortcut to ensuring you don't return duplicate rows and thus get duplicate objects. You'll sometimes see people doing this as well. If you see it too much it's a real red flag. Not that `DISTINCT` is bad or doesn't have valid applications. It does (on both counts) but it's not a surrogate or a stopgap for writing correct queries.

From Why I Hate DISTINCT:

> Where things start to go sour in my opinion is when a developer is building substantial query, joining tables together, and all of a sudden he realizes that it **looks** like he is getting duplicate (or even more) rows and his immediate response...his "solution" to this "problem" is to throw on the DISTINCT keyword and **POOF** all his troubles go away.

5. Favouring aggregation over joins

Another common mistake by database application developers is to not realize how much more expensive aggregation (ie the `GROUP BY` clause) can be compared to joins.

To give you an idea of how widespread this is, I've written on this topic several times here and been downvoted a lot for it. For example:

From SQL statement - “join” vs “group by and having”:

> First query:
>
> ```
> SELECT userid
> FROM userrole
> WHERE roleid IN (1, 2, 3)
> ```

> ```
> GROUP by userid
> HAVING COUNT(1) = 3
> ```
>
> Query time: 0.312 s
>
> Second query:
>
> ```
> SELECT t1.userid
> FROM userrole t1
> JOIN userrole t2 ON t1.userid = t2.userid
> AND t2.roleid = 2
> JOIN userrole t3 ON t2.userid = t3.userid
> AND t3.roleid = 3
> AND t1.roleid = 1
> ```
>
> Query time: 0.016 s
>
> That's right. The join version I proposed is **twenty times faster than the aggregate version.**

6. Not simplifying complex queries through views

Not all database vendors support views but for those that do, they can greatly simplify queries if used judiciously. For example, on one project I used a generic Party model for CRM. This is an extremely powerful and flexible modelling technique but can lead to many joins. In this model there were:

- **Party**: people and organisations;
- **Party Role**: things those parties did, for example Employee and Employer;

- **Party Role Relationship**: how those roles related to each other.

Example:

- Ted is a Person, being a subtype of Party;
- Ted has many roles, one of which is Employee;
- Intel is an organisation, being a subtype of a Party;
- Intel has many roles, one of which is Employer;
- Intel employs Ted, meaning there is a relationship between their respective roles.

So there are five tables joined to link Ted to his employer. You assume all employees are Persons (not organisations) and provide this helper view:

```
CREATE VIEW vw_employee AS
SELECT p.title, p.given_names, p.surname,
p.date_of_birth, p2.party_name employer_name
FROM person p
JOIN party py ON py.id = p.id
JOIN party_role child ON p.id = child.party_id
JOIN party_role_relationship prr ON child.id =
prr.child_id AND prr.type = 'EMPLOYMENT'
JOIN party_role parent ON parent.id =
prr.parent_id = parent.id
JOIN party p2 ON parent.party_id = p2.id
```

And suddenly you have a very simple view of the data you want but on a highly flexible data model.

7. Not sanitizing input

This is a huge one. Now I like PHP but if you don't know what you're doing it's really easy to create sites vulnerable to attack. Nothing sums it up better than the story of little Bobby Tables.

Data provided by the user by way of URLs, form data **and cookies** should always be treated as hostile and sanitized. Make sure you're getting what you expect.

8. Not using prepared statements

Prepared statements are when you compile a query minus the data used in inserts, updates and `WHERE` clauses and then supply that later. For example:

```
SELECT * FROM users WHERE username = 'bob'
```

vs

```
SELECT * FROM users WHERE username = ?
```

or

```
SELECT * FROM users WHERE username = :username
```

depending on your platform.

I've seen databases brought to their knees by doing this. Basically, each time any modern database encounters a new query it has to compile it. If it encounters a query it's seen before, you're giving the database the opportunity to cache the compiled query and the execution plan. By doing the query a lot you're giving the database the opportunity to figure that out and optimize accordingly (for example, by pinning the compiled query in memory).

Using prepared statements will also give you meaningful statistics about how often certain queries are used.

Prepared statements will also better protect you against SQL injection attacks.

9. Not normalizing enough

Database normalization is basically the process of optimizing database design or how you organize your data into tables.

Just this week I ran across some code where someone had imploded an array and inserted it into a single field in a database. Normalizing that would be to treat element of that array as a separate row in a child table (ie a one-to-many relationship).

This also came up in Best method for storing a list of user IDs:

> I've seen in other systems that the list is stored in a serialized PHP array.

But lack of normalization comes in many forms.

More:

- Normalization: How far is far enough?
- SQL by Design: Why You Need Database Normalization

10. Normalizing too much

This may seem like a contradiction to the previous point but normalization, like many things, is a tool. It is a means to an end and not an end in and of itself. I think many developers forget this and start treating a "means" as an "end". Unit testing is a prime example of this.

I once worked on a system that had a huge hierarchy for clients that went something like:

```
Licensee ->  Dealer Group -> Company -> Practice
-> ...
```

such that you had to join about 11 tables together before you could get any meaningful data. It was a good example of normalization taken too far.

More to the point, careful and considered denormalization can have huge performance benefits but you have to be really careful when doing this.

More:

- Why too much Database Normalization can be a Bad Thing
- How far to take normalization in database design?

- When Not to Normalize your SQL Database
- Maybe Normalizing Isn't Normal
- The Mother of All Database Normalization Debates on Coding Horror

11. Using exclusive arcs

An exclusive arc is a common mistake where a table is created with two or more foreign keys where one and only one of them can be non-null. **Big mistake.** For one thing it becomes that much harder to maintain data integrity. After all, even with referential integrity, nothing is preventing two or more of these foreign keys from being set (complex check constraints notwithstanding).

From A Practical Guide to Relational Database Design:

> We have strongly advised against exclusive arc construction wherever possible, for the good reason that they can be awkward to write code and pose more maintenance difficulties.

12. Not doing performance analysis on queries at all

Pragmatism reigns supreme, particularly in the database world. If you're sticking to principles to the point that they've become a dogma then you've quite probably made mistakes. Take the example of the aggregate queries from above. The aggregate version might look "nice" but its performance is woeful. A performance comparison should've ended the debate (but it didn't) but more to the point: spouting such ill-informed views in the first place is ignorant, even dangerous.

13. Over-reliance on UNION ALL and particularly UNION constructs

A UNION in SQL terms merely concatenates congruent data sets, meaning they have the same type and number of columns. The difference between them is that UNION ALL is a simple concatenation and should be preferred wherever possible whereas a UNION will implicitly do a DISTINCT to remove duplicate tuples.

UNIONs, like DISTINCT, have their place. There are valid applications. But if you find yourself doing a lot of them, particularly in subqueries, then you're probably doing something wrong. That might be a case of poor query construction or a poorly designed data model forcing you to do such things.

UNIONs, particularly when used in joins or dependent subqueries, can cripple a database. Try to avoid them whenever possible.

14. Using OR conditions in queries

This might seem harmless. After all, ANDs are OK. OR should be OK too right? Wrong. Basically an AND condition **restricts** the data set whereas an OR condition **grows** it but not in a way that lends itself to optimisation. Particularly when the different OR conditions might intersect thus forcing the optimizer to effectively to a DISTINCT operation on the result.

Bad:

```
... WHERE a = 2 OR a = 5 OR a = 11
```

Better:

```
... WHERE a IN (2, 5, 11)
```

Now your SQL optimizer may effectively turn the first query into the second. But it might not. Just don't do it.

15. Not designing their data model to lend itself to high-performing solutions

This is a hard point to quantify. It is typically observed by its effect. If you find yourself writing gnarly queries for relatively simple tasks or that queries for finding out relatively straightforward information are not efficient, then you probably have a poor data model.

In some ways this point summarizes all the earlier ones but it's more of a cautionary tale that doing things like query optimisation is often done first when it should be done second. First and foremost you should ensure you have a good data model before trying to optimize the performance. As Knuth said:

> Premature optimization is the root of all evil

16. Incorrect use of Database Transactions

All data changes for a specific process should be atomic. I.e. If the operation succeeds, it does so fully. If it fails, the data is left unchanged. - There should be no possibility of 'half-done' changes.

Ideally, the simplest way to achieve this is that the entire system design should strive to support all data changes through single INSERT/UPDATE/DELETE statements. In this case, no special transaction handling is needed, as your database engine should do so automatically.

However, if any processes do require multiple statements be performed as a unit to keep the data in a consistent state, then appropriate Transaction Control is necessary.

- Begin a Transaction before the first statement.
- Commit the Transaction after the last statement.
- On any error, Rollback the Transaction. And very NB! Don't forget to skip/abort all statements that follow after the error.

Also recommended to pay careful attention to the subtelties of how your database connectivity layer, and database engine interact in this regard.

17. Not understanding the 'set-based' paradigm

The SQL language follows a specific paradigm suited to specific kinds of problems. Various vendor-specific extensions notwithstanding, the language struggles to deal with problems that are trivial in langues like Java, C#, Delphi etc.

This lack of understanding manifests itself in a few ways.

- Inappropriately imposing too much procedural or imperative logic on the databse.
- Inappropriate or excessive use of cursors. Especially when a single query would suffice.
- Incorrectly assuming that triggers fire once per row affected in multi-row updates.

Determine clear division of responsibility, and strive to use the appropriate tool to solve each problem.

Share

edited Oct 21, 2018 at 1:50

community wiki
22 revs, 18 users 80%
cletus

9 On the MySQL statements about foreign keys, you're right that MyISAM doesn't support them, but you imply that merely using MyISAM is bad design. A reason I've used MyISAM is that InnoDB doesn't support FullText searches, and I don't think that's unreasonable. – Elle H Mar 8, 2009 at 4:39

1 I have to ask about #6. Using views like this is one of my favorite things to do, but I recently learned, to my horror, that with MySQL indexes on the underlying tables are only obeyed if the structure of the view allows use of the merge algorithm. Otherwise, a temp table is used and all your indexes are useless. It's even more alarming when you realize that a bunch of operations cause this behavior. It's a great way to turn a .01 sec query into to a 100 second one. Does anyone else here have experience with this? Check the links in my next comment. – Peter Bailey Jun 5, 2009 at 7:48

5 Completely disagree with #3. Yes, countries can cease to exist, but the country code will continue to represent the same thing. Same with currency codes or US States. It is dumb to use a surrogate key in these cases and creates more overhead in your queries as you must include an extra join. I would say that it is safer to say that you *probably* ought to use a surrogate for user-specific data (thus, not countries, currencies and US States). – Thomas Aug 2, 2010 at 17:32

1 RE: #11 The check constraint needed to enforce data integrity is trivial. There are other reasons for avoiding that design, but the need for "complex" check constraint isn't one of them. – Thomas Aug 2, 2010 at 17:34

2 With #3 you are not being honest. There are more downsides to the artificial key than "you may not need it." Specifically, using a natural key will give you the ability to control the order in which data in your table is written to the disk. If you know how your table will be queried, you can index it so concurrently-accessed rows will end-up in the same page. Furthermore, you can enforce data integrity using a unique composite index. If you need this you will have to add it in addition to your artificial key index. If said composite index is your pkey it's 2 birds killed with one stone. – Shane H Dec 13, 2010 at 15:49

110

votes

Key database design and programming mistakes made by developers

- **Selfish database design and usage.** Developers often treat the database as their personal persistent object store without considering the needs of other stakeholders in the data. This also applies to application architects. Poor database design and data integrity makes it hard for third parties working with the data and can substantially increase the system's life cycle costs. Reporting and MIS tends to be a poor cousin in application design and only done as an afterthought.
- **Abusing denormalised data.** Overdoing denormalised data and trying to maintain it within the application is a recipe for data integrity issues. Use denormalisation sparingly. Not wanting to add a join to a query is not an excuse for denormalising.
- **Scared of writing SQL.** SQL isn't rocket science and is actually quite good at doing its job. O/R mapping layers are quite good at doing the 95% of queries that are simple and fit well into that model. Sometimes SQL is the best way to do the job.
- **Dogmatic 'No Stored Procedures' policies.** Regardless of whether you believe stored procedures are evil, this sort of dogmatic attitude has no place on a software project.

- **Not understanding database design.** Normalisation is your friend and it's not rocket science. Joining and cardinality are fairly simple concepts - if you're involved in database application development there's really no excuse for not understanding them.

Share

edited May 23, 2017 at 11:55

community wiki
3 revs, 2 users 93%
ConcernedOfTunbridgeWells

2 One might argue that transactions should be done in transactional database and reporting and MIS should be done in a separate analysis database. Therefore you get the best of both worlds and everyone is happy (except for the poor mug who has to write the data transformation script to build the latter out of the former). – Chris Simpson Jul 13, 2009 at 22:58

Not just the poor mug writing the ETL - anyone using data from the system, the poor quality data in the MIS application that's boxed in because several key relationships aren't actually recorded at source, anyone involved in the endless reconciliation bun-fignts that ensue from the poor data quality. – ConcernedOfTunbridgeWells Jul 13, 2009 at 23:08

I could not possibly disagree more with point one. Databases are for persistence, they are not for inter-process communication. There are almost always better solutions to that problem. Unless there is an explicit requirement for it, you absolutely SHOULD treat the database as if nobody except for your application will ever use it. Even if there IS an explicit requirement, do some user story and root cause analysis on it and you will quite often discover a much better way of filling the requestor's intent. Then again, I do work at a company

where the phrase CQRS is somewhat common
– George Mauer Aug 5, 2010 at 19:48

3 Trivial example: I have an insurance policy administration system and need to load the state of 5 million claims into a ceded reinsurance system to calculate potential recoveries. The systems are older client-server COTS packages, designed to interface to even older mainframe systems. Both have to be reconciled for financial control purposes. This job is done once per month. By your logic I would write a series of user stories defining the requirements and ask the vendors to quote on adding a web service wrapper to their existing products. – ConcernedOfTunbridgeWells Aug 7, 2010 at 18:56

2 Then your DBA is either lazy or incompetent.
– ConcernedOfTunbridgeWells Jan 3, 2011 at 12:18

80

votes

1. Not using version control on the database schema
2. Working directly against a live database
3. Not reading up and understanding more advanced database concepts (indexes, clustered indexes, constraints, materialized views, etc)
4. Failing to test for scalability ... test data of only 3 or 4 rows will never give you the real picture of real live performance

Share

answered Mar 7, 2009 at 15:33

community wiki
Rad

1 I second, heavily, #1 and #2. Anytime I make a change to the DB I dump its schema and version it; I have three databases setup, a dev one, a staging one, and a live one - NOTHING ever gets "tested" on the live DB!! – Ixmatus Dec 13, 2010 at 17:52

46 votes

Over-use and/or dependence on stored procedures.

Some application developers see stored procedures as a direct extension of middle tier/front end code. This appears to be a common trait in Microsoft stack developers, (I'm one, but I've grown out of it) and produces many stored procedures that perform complex business logic and workflow processing. This is much better done elsewhere.

Stored procedures are useful where it has actuallly been proven that some real technical factor necessitates their use (for example, performance and security) For example, keeping aggregation/filtering of large data sets "close to the data".

I recently had to help maintain and enhance a large Delphi desktop application of which 70% of the business logic and rules were implemented in 1400 SQL Server stored procedures (the remainder in UI event handlers). This was a nightmare, primarily due to the difficuly of introducing effective unit testing to TSQL, lack of encapsulation and poor tools (Debuggers, editors).

Working with a Java team in the past I quickly found out that often the complete opposite holds in that environment.

A Java Architect once told me: "The database is for data, not code.".

These days I think it's a mistake to not consider stored procs at all, but they should be used sparingly (not by default) in situations where they provide useful benefits (see the other answers).

Share

edited Sep 23, 2010 at 17:24

community wiki
4 revs, 2 users 91%
Ashley Henderson

4 Stored procedures tend to become an island of hurt in any project where they are used, thus some developers make a rule "No stored procedures". So it looks like there is an open conflict beteween them. Your answer makes a good case for when to actually choose one way, or the other. – Warren P Jun 8, 2010 at 18:00

Benefits: security - you don't have to give applications the ability to "delete * from..."; tweaks - DBA's can tweak the queries without having to recompile/deploy the whole application; analysis - it's easy to recompile a bunch of procs after a data model change to ensure they are still valid; and, finally, considering SQL is executed by the database engine (not your application) then the concept of "database is for data, not code" is just retarded. – ChrisLively Apr 22, 2011 at 14:50

So, you'd enmesh your business logic in the UI, where it was divorced from the data being manipulated? This doesn't seem like such a good idea, particularly as data manipulation is most efficient when performed by the database server rather than by round-trips from the UI. That also means that it's more difficult to control the application because you can't rely on the database being in control of its data and potentially have different versions of a UI out there with different data manipulation going on. Not good. I don't let anything touch my data except through a stored procedure. – David T. Macknet Jun 13, 2011 at 14:39

If there is a need for separation of business logic from the UI, multi tier-architectures can be used. Or, a library with business objects and logic, used by different apps/UIs. Stored procedures lock your data/business logic to a specific database, changing a database in this case is very costly. And huge cost is bad. – Michał Turecki Aug 14, 2011 at 23:12

@too: Changing a database in most cases is very costly. Nevermind the idea of losing out on the performance and security features a particular DBMS provides. Further, additional tiers add complexity and decrease performance and additional layers are tied to your particular language. Finally, it's more likely the language being used will change than a database server. – ChrisLively Sep 7, 2011 at 20:30

41

votes

Number one problem? They only test on toy databases. So they have no idea that their SQL will crawl when the database gets big, and someone has to come along and fix it later (that sound you can hear is my teeth grinding).

Share

answered Mar 7, 2009 at 15:37

community wiki

Bob Moore

2 Size of the database is relevant, but a bigger issue is load-- even if you test on a real dataset you aren't testing performance of your queries when the database is under a production load, which can be a real eye-opener. – davidcl Jan 6, 2011 at 20:35

I would say that database size is bigger issue than load. I've seen many times, that there were missing crucial indexes - never been performance problem during tests, because whole database fit into memory – Cjxcz Odjcayrwl Jan 27, 2011 at 8:15

31 votes

Not using indexes.

Share

answered Mar 7, 2009 at 14:19

community wiki
Christophe Herreman

28 votes

Poor Performance Caused by Correlated Subqueries

Most of the time you want to avoid correlated subqueries. A subquery is correlated if, within the subquery, there is a reference to a column from the outer query. When this happens, the subquery is executed at least once for every row returned and could be executed more times if other conditions are applied after the condition containing the correlated subquery is applied.

Forgive the contrived example and the Oracle syntax, but let's say you wanted to find all the employees that have been hired in any of your stores since the last time the store did less than $10,000 of sales in a day.

```
select e.first_name, e.last_name
from employee e
where e.start_date >
        (select max(ds.transaction_date)
         from daily_sales ds
         where ds.store_id = e.store_id and
               ds.total < 10000)
```

The subquery in this example is correlated to the outer query by the store_id and would be executed for every employee in your system. One way that this query could be optimized is to move the subquery to an inline-view.

```
select e.first_name, e.last_name
from employee e,
     (select ds.store_id,
             max(s.transaction_date)
transaction_date
      from daily_sales ds
      where ds.total < 10000
      group by s.store_id) dsx
where e.store_id = dsx.store_id and
      e.start_date > dsx.transaction_date
```

In this example, the query in the from clause is now an inline-view (again some Oracle specific syntax) and is only executed once. Depending on your data model, this query will probably execute much faster. It would perform better than the first query as the number of employees grew. The first query could actually perform better if there were few employees and many stores (and perhaps many of stores had no employees) and the daily_sales table was indexed on store_id. This is not a likely scenario but shows how a correlated query could possibly perform better than an alternative.

I've seen junior developers correlate subqueries many times and it usually has had a severe impact on performance. However, when removing a correlated subquery be sure to look at the explain plan before and after to make sure you are not making the performance worse.

Share

edited Dec 13, 2010 at 15:48

community wiki
2 revs
adam

1 Great point, and to emphasize one of your related points -- test your changes. Learn to use explain plans (and see what the database is actually doing to execute your query, and what it costs), do your tests on a large dataset, and don't make your SQL overly-complex and unreadable/unmaintainable for an optimization that doesn't actually improve real performance. – Rob Whelan Dec 16, 2010 at 4:11

21 votes

In my experience:

Not communicating with experienced DBAs.

Share

answered Mar 7, 2009 at 14:42

community wiki
Kb.

17
votes

Using Access instead of a "real" database. There are plenty of great small and even free databases like SQL Express, MySQL, and SQLite that will work and scale much better. Apps often need to scale in unexpected ways.

Share

edited Mar 8, 2009 at 4:20

community wiki
3 revs, 2 users 89%
Nathan Voxland

16
votes

Forgetting to set up relationships between the tables. I remember having to clean this up when I first started working at my current employer.

Share

answered Mar 7, 2009 at 14:16

community wiki
TheTXI

Using Excel for storing (huge amounts of) data.

14
votes

I have seen companies holding thousands of rows and using multiple worksheets (due to the row limit of 65535 on previous versions of Excel).

Excel is well suited for reports, data presentation and other tasks, but should not be treated as a database.

Share

answered Aug 2, 2010 at 17:18

community wiki
ML--

14
votes

I'd like to add: Favoring "Elegant" code over highly performing code. The code that works best against databases is often ugly to the application developer's eye.

Believing that nonsense about premature optimization. Databases must consider performance in the original design and in any subsequent development. Performance is 50% of database design (40% is data integrity and the last 10% is security) in my opinion. Databases which are not built from the bottom up to perform will perform badly once real users and real traffic are placed against the database. Premature optimization doesn't mean no optimization! It doesn't mean you should write code that will almost always perform badly because you find it easier (cursors for example which should never be allowed in a production database unless all else has failed). It means you don't need to look at squeezing out that last little bit of performance until you need to. A lot is known about what will perform better on databases, to ignore this in design and development is short-sighted at best.

Share

edited Sep 23, 2010 at 17:28

community wiki
2 revs, 2 users 67%
HLGEM

2 +1 - Database programming involves optimising the behaviour of mechanical components. Note, however, that Knuth says premature optimisation is the root of all evil about 97% of the time (or words to that effect). Database design is one area where you really do have to think about this up front. – ConcernedOfTunbridgeWells Oct 21, 2009 at 8:00

2 Ahem... what you are talking about is optimization that is not premature. Some consideration of real usage is required from the beginning in database design (and application design as well, really). Knuth's rule is actually not trivial to follow, because you have to decide what's premature and what isn't -- it really comes down to "don't perform optimizations without data". The early performance-related decisions you're talking about *have* data -- certain designs will set unacceptable limits on future performance, and you can calculate them. – Rob Whelan Dec 16, 2010 at 4:20

13 votes

Not using parameterized queries. They're pretty handy in stopping SQL Injection.

This is a specific example of not sanitizing input data, mentioned in another answer.

Share

answered Mar 7, 2009 at 16:52

community wiki
Ash

3 Except sanitizing input is wrong. Sanitizing implies putting it somewhere where it can be dangerous. Parameterizing means keeping it out of the path of harm altogether. – Dustin Dec 13, 2010 at 9:38

12
votes

I hate it when developers use nested select statements or even functions the return the result of a select statement inside the "SELECT" portion of a query.

I'm actually surprised I don't see this anywhere else here, perhaps I overlooked it, although @adam has a similar issue indicated.

Example:

```
SELECT
    (SELECT TOP 1 SomeValue FROM SomeTable WHERE
SomeDate = c.Date ORDER BY SomeValue desc) As
FirstVal
    ,(SELECT OtherValue FROM SomeOtherTable WHERE
SomeOtherCriteria = c.Criteria) As SecondVal
FROM
    MyTable c
```

In this scenario, if MyTable returns 10000 rows the result is as if the query just ran 20001 queries, since it had to run the initial query plus query each of the other tables once for each line of result.

Developers can get away with this working in a development environment where they are only returning a few rows of data and the sub tables usually only have a small amount of data, but in a production environment, this kind of query can become exponentially costly as more data is added to the tables.

A better (not necessarily perfect) example would be something like:

```
SELECT
     s.SomeValue As FirstVal
    ,o.OtherValue As SecondVal
FROM
    MyTable c
    LEFT JOIN (
        SELECT SomeDate, MAX(SomeValue) as
SomeValue
        FROM SomeTable
        GROUP BY SomeDate
     ) s ON c.Date = s.SomeDate
    LEFT JOIN SomeOtherTable o ON c.Criteria =
o.SomeOtherCriteria
```

This allows database optimizers to shuffle the data together, rather than requery on each record from the main table and I usually find when I have to fix code where this problem has been created, I usually end up increasing the speed of queries by 100% or more while simultaneously reducing CPU and memory usage.

Share

answered Feb 23, 2011 at 16:25

community wiki
CStroliaDavis

12

votes

For SQL-based databases:

1. Not taking advantage of CLUSTERED INDEXES or choosing the wrong column(s) to CLUSTER.
2. Not using a SERIAL (autonumber) datatype as a PRIMARY KEY to join to a FOREIGN KEY (INT) in a parent/child table relationship.
3. Not UPDATING STATISTICS on a table when many records have been INSERTED or DELETED.
4. Not reorganizing (i.e. unloading, droping, re-creating, loading and re-indexing) tables when many rows have been inserted or deleted (some engines physically keep deleted rows in a table with a delete flag.)
5. Not taking advantage of FRAGMENT ON EXPRESSION (if supported) on large tables which have high transaction rates.
6. Choosing the wrong datatype for a column!
7. Not choosing a proper column name.

8. Not adding new columns at the end of the table.
9. Not creating proper indexes to support frequently used queries.
10. creating indexes on columns with few possible values and creating unnecessary indexes.
    ...more to be added.

Share

edited Jun 14, 2011 at 2:00

community wiki
2 revs
Frank Computer

1 A quibble: 2) is actually bad practice. I see what you're getting at - you want an unique index on that autonumber, and to use it as a surrogate key. But the primary key shouldn't be an autonumber, as that's not what a primary key IS: a primary key is "what the record is about," which (except for things like sales transactions) is NOT the autonumber, but some unique bit of information about the entity being modeled. – David T. Macknet Jun 13, 2011 at 14:43

the main reason for using autonumber for primary and foreign key is to guarantee that a parent-child join can be maintained irregardless of changes in any other columns. using a different primary key, like customer name or other data can be risky! – Joe R. Jun 14, 2011 at 1:54

@David: I stand corrected!.. its not necessary to use autonumber as the primary key, one can still have an indexed serial column in the parent, joining the surrogate in the child to guarantee the relation will not be severed, while having another column as a meaningful primary to locate the row! – Joe R. Sep 10, 2011 at 20:27

It's an issue of semantics, at the end of the day ... and Microsoft prefers primary keys to be meaningless, rather than meaningful. Debates around it rage on, but I fall into the "meaningful" camp. :) – David T. Macknet Sep 13, 2011 at 12:46

9

votes

- Not taking a backup before fixing some issue inside production database.
- Using DDL commands on stored objects(like tables, views) in stored procedures.
- Fear of using stored proc or fear of using ORM queries wherever the one is more efficient/appropriate to use.
- Ignoring the use of a database profiler, which can tell you exactly what your ORM query is being converted into finally and hence verify the logic or even for debugging when not using ORM.

Share

edited Jan 24, 2011 at 18:37

community wiki
3 revs
WhoIsNinja

8

votes

Not doing the correct level of normalization. You want to make sure that data is not duplicated, and that you are splitting data into different as needed. You also need to make sure you are not following normalization *too* far as that will hurt performance.

Share

answered Mar 7, 2009 at 14:22

community wiki
Nathan Voxland

How far is too far? If no data is duplicated how can you take it further? – finnw Mar 7, 2009 at 15:59

Normalization is a balance of removing redundant data and increasing flexibility vs decreased performance and increased complexity. Finding the correct balance takes experience and it changes over time. See en.wikipedia.org/wiki/Database_normalization for information on when to denormalize – Nathan Voxland Mar 7, 2009 at 16:32

8
votes

Treating the database as just a storage mechanism (i.e. glorified collections library) and hence subordinate to their application (ignoring other applications which share the data)

Share

answered Mar 7, 2009 at 16:02

community wiki
finnw

A corollary to to this is offloading too much query work to the application instead of keeping it in the db where it belongs. LINQ is particularly bad about this. – 3Dave Jun 18, 2010 at 17:08

8

votes

- Dismissing an ORM like Hibernate out of hand, for reasons like "it's too magical" or "not on **my** database".
- Relying too heavily on an ORM like Hibernate and trying to shoehorn it in where it isn't appropriate.

Share

answered Apr 2, 2009 at 18:33

community wiki
Adam Jaskiewicz

8

votes

1 - Unnecessarily using a function on a value in a where clause with the result of that index not being used.

Example:

```
where to_char(someDate,'YYYYMMDD') between
:fromDate and :toDate
```

instead of

```
where someDate >= to_date(:fromDate,'YYYYMMDD') and
someDate < to_date(:toDate,'YYYYMMDD')+1
```

And to a lesser extent: Not adding functional indexes to those values that need them...

2 - Not adding check constraints to ensure the validity of the data. Constraints can be used by the query optimizer, and they REALLY help to ensure that you can trust your invariants. There's just no reason not to use them.

3 - Adding unnormalized columns to tables out of pure laziness or time pressure. Things are usually not designed this way, but evolve into this. The end result, without fail, is a ton of work trying to clean up the mess when you're bitten by the lost data integrity in future evolutions.

Think of this, a table without data is very cheap to redesign. A table with a couple of millions records with no integrity... not so cheap to redesign. Thus, doing the correct design when creating the column or table is amortized in spades.

4 - not so much about the database per se but indeed annoying. Not caring about the code quality of SQL. The fact that your SQL is expressed in text does not make it OK to hide the logic in heaps of string manipulation algorithms. It is perfectly possible to write SQL in text in a manner that is actually readable by your fellow programmer.

Share

edited Oct 4, 2010 at 11:05

community wiki
4 revs, 3 users 80%
John Nilsson

7
votes

This has been said before, but: **indexes, indexes, indexes**. I've seen so many cases of poorly performing enterprise web apps that were fixed by simply doing a little profiling (to see which tables were being hit a lot), and then adding an index on those tables. This doesn't even require much in the way of SQL writing knowledge, and the payoff is huge.

Avoid data duplication like the plague. Some people advocate that a little duplication won't hurt, and will improve performance. Hey, I'm not saying that you have to torture your schema into Third Normal Form, until it's so abstract that not even the DBA's know what's going on. Just understand that whenever you duplicate a set of names, or zipcodes, or shipping codes, the copies WILL fall out of synch with each other eventually. It WILL happen. And then you'll be kicking yourself as you run the weekly maintenance script.

And lastly: use a clear, consistent, intuitive naming convention. In the same way that a well written piece of code should be readable, a good SQL schema or query should be readable and practically *tell* you what it's doing, even without comments. You'll thank yourself in six months, when you have to to maintenance on the tables. `"SELECT account_number, billing_date FROM national_accounts"` is infinitely easier to work with than "SELECT ACCNTNBR, BILLDAT FROM NTNLACCTS".

Share

answered Jul 28, 2009 at 20:23

community wiki
pbailey19

If you set them up correctly they won't but this involves the use of triggers which many people are allergic to. – HLGEM Oct 20, 2009 at 13:45

6
votes

Not executing a corresponding SELECT query before running the DELETE query (particularly on production databases)!

Share

answered Jan 6, 2010 at 12:11

community wiki
Jamol

5
votes

The most common mistake I've seen in twenty years: not planning ahead. Many developers will create a database, and tables, and then continually modify and expand the tables as they build out the applications. The end result is often a mess and inefficient and difficult to clean up or simplify later on.

Share

answered Jan 29, 2011 at 3:56

community wiki
Skatterbrainz

1 I can imagine the horrors that ensue in these situations... Schemaless databases are a much better fit for rapid prototyping and iterative development, but like everything else, such flexibility comes with various trade-offs. – Zsolt Török Jan 29, 2011 at 12:36

4 votes

a) Hardcoding query values in string
b) Putting the database query code in the "OnButtonPress" action in a Windows Forms application

I have seen both.

Share

edited Sep 23, 2010 at 17:29

community wiki
2 revs, 2 users 67%
Benoit

4 "Putting the DB query code in the "OnButtonPress" action in a Windows Form application" What's the database mistake here?

– recursive Mar 7, 2009 at 15:52

@recursive: it's a huge SQL injection vulnerability. Anyone can send arbitrary SQL to your server and it will be run verbatim. – Bill Karwin Mar 16, 2009 at 23:41

b) is an architecture mistake. Of course, coding queries directly in your app is a bad idea, anyway. – 3Dave Jun 18, 2010 at 17:11

4

votes

Not paying enough attention towards managing database connections in your application. Then you find out the application, the computer, the server, and the network is clogged.

Share

edited Sep 23, 2010 at 17:32

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Mar 29, 2009 at 13:38

chefsmart

6,981 ● 9 ● 44 ● 47

4

votes

1. Thinking that they are DBAs and data modelers/designers when they have no formal indoctrination of any kind in those areas.
2. Thinking that their project doesn't require a DBA because that stuff is all easy/trivial.
3. Failure to properly discern between work that should be done in the database, and work that should be done in the app.

4. Not validating backups, or not backing up.
5. Embedding raw SQL in their code.

Share

answered Jan 5, 2011 at 13:55

community wiki
jonesy

3
votes

Here is a link to video called ‘Classic Database Development Mistakes and five ways to overcome them’ by Scott Walz

Share

answered Mar 7, 2009 at 17:07

community wiki
Charles Faiga

3
votes

Not having an understanding of the databases concurrency model and how this affects development. It's easy to add indexes and tweak queries after the fact. However applications designed without proper consideration for hotspots, resource contention and correct operation (Assuming what you just read is still valid!) can require significant changes within the database and application tier to correct later.

Share

edited Sep 23, 2010 at 17:33

community wiki
2 revs, 2 users 50%
Einstein

3

votes

Not understanding how a DBMS works under the hood.

You cannot properly drive a stick without understanding how a clutch works. And you cannot understand how to use a Database without understanding that you are really just writing to a file on your hard disk.

Specifically:

1. Do you know what a Clustered Index is? Did you think about it when you designed your schema?
2. Do you know how to use indexes properly? How to reuse an index? Do you know what a Covering Index is?
3. So great, you have indexes. How big is 1 row in your index? How big will the index be when you have a lot of data? Will that fit easily into memory? If it won't it's useless as an index.
4. Have you ever used EXPLAIN in MySQL? Great. Now be honest with yourself: Did you understand even half of what you saw? No, you probably didn't. Fix that.

5. Do you understand the Query Cache? Do you know what makes a query un-cachable?
6. Are you using MyISAM? If you NEED full text search, MyISAM's is crap anyway. Use Sphinx. Then switch to Inno.

Share

answered Dec 13, 2010 at 15:42

community wiki
Shane H

2 A better analogy might be that one cannot properly *troubleshoot* a manual transmission without understanding a clutch. Plenty of people properly drive a stick-shift without knowing how a clutch works. – Michael Easter Dec 13, 2010 at 16:08

3
votes

1. Using an ORM to do bulk updates
2. Selecting more data than needed. Again, typically done when using an ORM
3. Firing sqls in a loop.
4. Not having good test data and noticing performance degradation only on live data.

Share

answered Dec 13, 2010 at 17:36

community wiki
Sriram

1 2 Next