

Executive Summary

Overall, the <dbname> database is in good shape, and there are no glaring problems with security, schema, or code. Performance is actually decent, mostly due to the fact that the database is small, and activity on the database is very heavily centered on only one table (dbo.dev_EventData) a few supporting stored procedures.

There were, of course, a number of issues discovered during analysis – only one of them that qualified as a critical issue (most clients have one or two critical findings), and it is very easy to remedy. Otherwise, the majority of the findings presented below are generally favorable in the sense that issues are easy to fix, and doing so will help squeeze extra performance out of your database – saving you needing to worry about buying or upgrading your hardware.

The findings below are broken down according to severity and impact. My main goal in presenting the findings is to provide customized education surrounding common (or dangerous and not-so-common) findings in terms of the causes, the impact, and solutions.

Findings

The Good

- Code is well formatted, and easy to read. (This is a major factor in helping to prevent logic errors and bugs, and ensuring proper maintenance.)
- The database is secure. Access provide to the application servers is sufficiently curtailed to the point that:
 - SQL Injection through your application will not be possible (unless the application is running any dynamic SQL directly against the database – though I would assume this isn't the case, and my trace against the production strengthens this hunch).
 - You application servers won't be able to elevate privileges or anything 'spooky' on the DB servers should your app servers become compromised. **Though if your app servers were compromised, hackers would have easy access to all of your data, and would be able to destroy your data very easily.** I'm providing instructions on how to mitigate this threat down below, in the [Other](#) section of the findings.

Critical

There was one critical finding – something that should be addressed immediately.

- **Size of Log File on Production Database.** The role of Log Files in SQL Server is, effectively, to facilitate atomicity (i.e. to ensure that data is always left in a dependable state) and to aid in enabling concurrency. Each time database data is updated, the value of the data before the operation, and after the operation is logged to the database's log file. Large operations, therefore can take up considerable log space. If sufficient log space is not available, operations will not be permitted to run and will be cancelled.

It is therefore critical to ensure that the log file for active databases is sufficiently large to handle day to day operations, as well as special operations (like the addition of a new column to a very large table). Happily, at each scheduled backup, changes found in the log file, that are flagged as having successfully completed, are permanently added to the database, and the space required to store the before and after images of the operation is recovered (meaning that one good way to keep your log file at a decent size is to back it up frequently).

The Problem: The Log file on <servername>.db1. <dbname>.com is currently 41 GB in size. A

good rule of thumb is to have a log file that is about 25% the size of a database's data file. The data file for the <dbname> database (<dbname>.mdf) is about 4GB in size (2GB of which is full of data). A suitable log file size would then be about 1GB in size. In other words, the current, production, log file is about 40 times what it should be.

The Impact: In addition to needlessly taking up disk space (which isn't that big of a concern to most organizations), the large size will add anywhere from 20 minutes to 2+ hours to restore the database from a backup (depending upon hardware – though most likely down on the lower end). Given that the <dbname> database spans 2 servers with replication for redundancy/failover, this may not SEEM like an issue, but a 'dumb' UPDATE or DELETE (something run without a WHERE clause by hand, etc.) will quickly replicate to both servers within seconds – and the only way to recover from this would be to do a point in time restore from the backup. **You won't want an extra 20+ minutes of down time while waiting for a backup to finish recovering because it is slowly churning through the process of needlessly allocating log space.**

The Solution: The log file needs to be large enough to:

- *Avoid fragmentation.* If the log file is constantly growing to meet processing needs, it becomes fragmented logically, and in the underlying File System – adding unnecessary performance overhead. It is therefore a good idea to keep the log file large enough to handle high volume periods, and to allow it to grow in larger 'chunks' to avoid unnecessary fragmentation each time the log file is allowed to grow.
- *Service Requests.* If the log file is restricted to a specified size, and a large operation (as described above) is executed, the operation won't be able to complete. Worse. If a number of very small operations 'fill up the log file' then normal, simple, INSERTs won't happen, and data will be lost. It is therefore critical to keep enough space available in the log file to service requests.

Therefore, I would suggest shrinking the log file back down to 2 GB. But, to account for the occasional large operation, or extra busy day, permit the log file to grow (in 250MB chunks to avoid fragmentation). Periodic backups will clear the Log File, preventing it from growing too large, and a simple batch file run as a SQL Server Agent task can be run monthly to shrink the file back down to the 2 GB desired size. The only downside to this operation is that it needs to take place right after a BACKUP operation or there's a chance that the log file will not release space as requested. In instances where the log file won't release space, it can be forced to do so, but at the cost of invalidating the log file – meaning that a subsequent BACKUP operation should be performed immediately after truncating the logs (if forcing them to truncate is required).

Creating scripts to enable all of these changes is an almost trivial matter for me, and I'm happy to provide them, but they will effectively require admin permissions to run them, in order to make sure that the database is properly backed up once the operation is complete. As such, this is something we'll want to discuss options for execution/implementation, etc.

Important

Important issues are problems that warrant attention, but don't need to be addressed immediately.

- **Data Type Coercion in Stored Procedures.** A large number of stored procedures specify parameters for the procedure, that don't match underlying queries. For example, in dbo.DoesSiteExistExt a site id (stored as an integer in dbo.dev_Site) is passed into the stored procedure as a varchar(50).

The Problem: This is actually a fairly common problem – and one that many developers overlook – but it can have two very important side-effects:

- **Data Truncation.** Not too much of a problem to be concerned, but remember, casting is an operation where humans (developers) tell the compiler how to translate one data

type into another – with no/minimal loss, coercion occurs when the computer has to make a guess, and under certain (once in a blue moon) circumstances it can guess wrongly with potentially dangerous consequences.

- **Non Optimal Search Conditions.** Only seasoned guru's know this (this is effectively undocumented to my knowledge), but it's QUITE common for a simple query like :

```
SELECT customerID, custAddress FROM massiveCustomerTable WHERE
@custID_storedInTableAsInt = @custID_representedAsVarchar
```

to experience major performance problems because of data type coercion. SQL Server relies upon complex statistics to be able to correctly make use of indexes – which can speed query response times in terms of orders of magnitude. When data types don't EXACTLY match (i.e. if you are trying to match a varchar(40) against a varchar(50) parameter you'll still encounter this issue), then SQL Server isn't able to accurately use statistics, and therefore can't use indexes properly, meaning that Index Seeks can be degraded to much more costly operations like clustered index scans and table scans (which can be orders of magnitude more expensive and lengthy).

The big problem, of course, is that both of these side-effects can frequently exist without any noticeable effects. The second issue can be a problem especially on peak days when a db with tables that have millions of rows are being pummeled.

The Solution: The solution can be complex, as changing just the stored procedure signatures will break calling/application code due to parameter mismatches. The only real way to fix this problem is to change the calling code and the stored procedure signatures – and test thoroughly in a dev environment (because subtle changes like this are highly prone to introducing bugs that will destroy data almost imperceptibly). However, unless stored procedures that access the dbo.dev_EventData table suffer from this problem, most can be safely ignored until a later date when changing the stored procedure signature in the client code makes more sense. In other words, this is something to DEFINITELY be aware of now (and for future applications), but something that probably won't be beneficial to investigate but for procedures that directly query dbo.dev_EventData

- **Failure to use OUTPUT and RETURN parameters in Stored Procedures.** SQL Server provides 3 parameter directions: INPUT, OUTPUT and INPUT/OUTPUT. It also provides a special RETURN parameter which is explicitly intended to return information about the success or failure of the stored procedure (many developers will attempt to overload this RETURN parameter as a 'short-cut,' but that is a major 'no-no' though SQL Server has no mechanism to enforce this). OUTPUT parameters and RETURN parameters are an excellent, and VERY light-weight, mechanism for returning status information from stored procedures, as well as row counts, new object ids, etc.

The Problem: The stored procedures in the <dbname> database do not make use, at all, of these standard interfaces for exchanging data. Instead a common strategy seems to be to SELECT a 'message' on the 'way out' of the stored procedure – effectively hijacking SQL Server's Tabular Data Stream – designed as a way to return BULKY result sets. (This 'message' ends up being a scalar result set, but the result set overhead is still invoked unnecessarily.) Worse, there are some bad factors frequently employed by these 'messages.' In some stored procedures, a 'message' will return an integer id or row count, or it may pull back a string such as 'data not found' – all variations possible in a single stored procedure. This effectively amounts to parameter overloading, which is a bad form of coupling that makes applications more brittle and harder to maintain. It also makes them much more susceptible to logic flaws.

The Solution: This one is tough. Given how entrenched this paradigm is in the <dbname> database, and given the fact that it currently WORKS – I'd suggest leaving these @message return values alone. Changing them will be expensive in terms of effort due to the potential

learning curve (your developers may already be painfully, and keenly aware of this issue – or they may not, in which case they'll need to learn about OUTPUT and RETURN parameters), the time involved to rework CORE logic around the interface, and testing. However, future changes to any existing stored procedures should seek to replace this data overloading, as should stored procedures added in the future. This, however, will require considerable change to the calling code (instead of doing a `Command.ExecuteScalar()`; you will need to change calls to `Command.ExecuteNonQuery()`; etc.), and a rework of the existing logic). If you would like further help in this area, let me know.

- **Non Normalized entities in dbo.dev_EventData table.** A quick, visual scan of `dbo.dev_EventData` reveals that the table is not correctly normalized, and holds at least two different business entities (most likely quite a few more). For example, run a simple query such as `SELECT TOP 500 * FROM dev_EventData`. Then scroll through the results – they're not homogenous – some rows are almost entirely NULL values – except for a description field, and other rows don't have Latitude and Longitude, but do have Mileage, where other rows are the complete inverse.

The problem: This clearly indicates non-normalized entities existing in the same table, meaning that queries against this table will require extra complexity, and deriving good reports from the table's data will be complicated. Furthermore, 'transparent' business rules will (and do) exist in stored procedures that INSERT or UPDATE against this table to account for the fact that different entities are being manipulated. This added complexity can have a major impact on the effectiveness of indexes, will require extra processing power, and will severely raise the potential of introducing logical flaws and bugs into existing code if it is touched or modified.

Mitigating factors: I've just painted a pretty bleak picture of your application. The architecture for a major component of your application is effectively flawed. While this IS accurate, it only BECOMES AN ISSUE when the performance required from your application becomes a matter of 'push coming to shove.' In other words, good hardware, and the fact that your application doesn't appear to be pummeling the database will go a long way toward mitigating the performance aspects of this flaw – but the possibility of logic flaws due to 'transparent' business rules is still very real (and has high 'bus factor' – i.e. it's not desirable because usually only one or two developers will completely understand the nuances well enough should you require changes or modifications).

The Solution: The solution for this one is tough – the only real way to fix this is to redesign the table and break it down into its constituent elements – i.e. a table for each entity. If each entity shares common attributes or data, it is possible (and quite effective) to sub-class the tables – i.e. build an `EventData` table that stores critical info about the asset involved, the time the event occurred, and other common data – along with an `EventID`. Then a `Mileage` table can be made, for example, that would contain NOTHING but an `EventID` (the one-to-one `EventID` of the event in the `EventData` table), and column (or columns) that store only data related to the mileage entity – i.e. extensions to the `DataEvent` of a 'mileage' nature.

That said, making these kind of changes will be quite a cumbersome task – as it effectively represents a re-architecture of a MAJOR component of your application. I'll include a sample script showing the basics of the architecture I've outlined above for completeness, but I'd actually recommend that this 'flaw' be put on a wish list somewhere – and ignored until a rewrite occurs anyhow, or until performance becomes a MAJOR concern (but even so, there are other things you can fix to help mitigate the performance impact – buying more RAM will be undoubtedly cheaper than fixing this flaw, and will buy you considerable 'mitigation'). But if you're planning a rewrite in the future, that would be the perfect place to implement this re-architecture.

Minor

- **Stored Procedures don't access objects with Fully Qualified Object Names.** It's always a good coding practice in SQL Server to reference the owner of an object when calling it. i.e. instead of `SELECT x,y,z FROM table`, specify `SELECT x,y,z FROM dbo.table`. The reason for this best practice is to ensure that objects are explicitly named (it is possible to have a `john.customers` table and a `sally.customers` table – selecting from `customers` is therefore ambiguous and SQL Server won't resolve this – but will instead check to see if the name of the currently connected user matches the database owner, if it doesn't it will look for `dbo.TableName`). Why should this matter? If login or user names are ever changed, names may not resolve correctly, and unpredictable results can occur. That said, your application is currently working fine – so ignore this unless you've got time to burn.
- **Non-Standard Constraint and Index Names.** A number of indexes and constraints have non-homogenous names. Most organizations use some sort of `prefix_name_explanation` type nomenclature, such as `IX_Customers_State` for an index name, or `PK_Customers` as a table's primary key constraint. A few of the indexes and constraints in the `<dbname>` database have `IX_#####` as the name, etc. (Not an issue really, just an FYI.)
- **SELECT * result sets in Stored Procedures.** A number of stored procedures return result sets with `SELECT * FROM sometable` type queries. Best practices actively discourage this type of coding, as the addition of a new column to the table can cause serious problems. First of all, if a column is merely appended to the 'end' of the table, `SELECT *` queries will still pull that data back, and send it to the client where it is ignored/discarded (after being transmitted over the wire and marshaled up and down stacks, etc.). Worse, if the new column is added 'inside' the table (which is technically a no-no, but it happens VERY frequently), and the calling application deals with columns by ordinal position, the calling application will either break or suffer logic problems.

Performance Considerations

As mentioned in the Executive Summary, the overall performance of the `<dbname>` database is currently quite good. A number of findings are presented below that can be considered for implementation depending upon the need to 'squeeze' extra performance out of your database. I'm a performance purist/freak who lives for tuning databases and squeezing extra performance out of every nook and cranny – but the truth is that unless you NEED to leverage more performance, your time, money, and effort are better spent elsewhere.

- **Multiple Data Files.** SQL Server allows for database data to be stored in multiple files. Each database requires a primary data file (`.mdf`) and a log file (`.ldf`), but for very active databases, the possibility of using multiple log files (`.ldf` files) exists, and can even allow for log data to 'straddle' multiple drives. The same goes for data files – and the notion of 'straddling' can provide considerable performance improvements as well as allow large databases to span data across multiple drives. A common strategy with larger databases is to break logical operations as well across multiple data files. For example, when a query is run against a table with millions of rows if an index is used, SQL Server can traverse the index on one file, and begin fetching results from another file – providing a considerable performance boost. Your database is currently right on the cusp of potentially seeing benefit from this approach, especially given that `dbo.dev_EventData` has > 10 million rows and a handful of indexes. My recommendation would be to leave your current data file setup as it exists (`<dbname>.mdf` and `<dbname>.ldf`), but add a secondary data file (`.ndf`) for indexes – and then move **NON CLUSTERED** (only) indexes on to this specialized data file – at least non clustered indexes from the `dbo.dev_DataEvent` table. This will be a fairly complex process, and can additionally cause a momentary lull in functionality on the site as the

indexes are moved/rebuilt. If you are interested in pursuing this option, let me know and I can cook up some scripts and instructions in an hour or two.

- **SET NOCOUNT ON.** SQL Server uses a very 'chatty' protocol to keep clients apprised of progress on the Server. For example, a stored procedure that checks if table has a row present, then does an update, followed by the assignment of a rowcount to a variable, etc. will necessitate communication BACK AND FORTH between SQL Server and the client application. This is by design, and lets your application know what is up at every stage of the operation. The problem is that your calling application simply doesn't care – and the result is that you add an extra layer of 'chat' (moving small amounts of data back and forth, up and down your entire TCP/IP and memory stacks, etc.) with no measurable benefit. To fix this, 'tell' SQL Server to 'stop the chatter, and just get the work done' by immediately specifying SET NOCOUNT ON at the top of your stored procedures, e.g.

```
CREATE PROC dbo.DoStuff
    @someVariable int
AS
    SET NOCOUNT ON
```

```
    SELECT something FROM dbo.SomeTable WHERE someColumn = @someVariable
```

```
    RETURN 0
```

```
GO
```

This will 'squell' the DONE_IN_PROC messages that SQL Server fires back to the client. If you have a sproc where you NEED this (which your sprocs almost assuredly do NOT need 'chattiness') you can simply SET NOCOUNT OFF where needed.

This change is very simple to make, and will make a small impact in overall performance.

- **CURSOR optimization:** A number of stored procedures in the <dbname> database use cursors. Procedural constructs by nature and mind-set, cursors are generally used by developers in lieu of set-based operations (which can usually accomplish the same functionality in a much more performant manner, but at a cost of increased complexity in the mind of most developers). It is very common for non-database-savvy developers to severely abuse cursors which imposes major performance problems. Happily none of the cursors currently present in the <dbname> database are abusive in nature; *however*, they could be quickly and easily optimized. SQL Server allows for a number of powerful options during cursor creation – most of which aren't needed in the majority of cases/uses, and add measurable overhead. Failure to be exact and explicit with these options results in default options that sadly come at the price of adding unnecessary and expensive overhead.

To correct this in your current stored procedures, simply make your CURSORS explicitly READ ONLY and FAST FORWARD. This will allow them to be created without the need for expensive pointers needed for rewinding or modification – making them considerably faster and more performant. To implement this change, simply change the extant

```
DECLARE <cursorName> CURSOR FOR SELECT ....
```

syntax to

```
DECLARE <cursorName> LOCAL READ_ONLY FAST_FORWARD FOR SELECT ...
```

This will explicitly set light-weight cursor options, and will also change the scope of the cursor to the current process, meaning that other threads won't need to have pointers created to this cursor as well.

This is a fix that I highly recommend, as it is easy to implement, and will provide immediate benefit. As such, here is a list of all stored procedures in your database that have cursors:

```

dbo.dev_Insert_Event_Data
dbo.CreateBuddyList
dbo.<sanitized>
dbo.UpdateTimeZone
dbo.dev_Get<sanitized>
dbo.dev_GetMapList
dbo.dev_GetSiteList
dbo.dev_<sanitized>Site<sanitized>
dbo.dev_<sanitized>SiteWrapper

```

- **Expensive Stored Procedures:** Analysis of the Profiler Trace that I executed against the <dbname> database while under load indicates that calls to the dbo.dev_InsertExternal_Event_Data stored procedure represent nearly 50% of the total calls to stored procedures in the database under load. This shouldn't be too surprising, given the nature of this stored procedure (to add rows to the dev_DataEvent table – which is the overwhelming bulk of the database – over 99% of database's allocated space is found in this table). This stored procedure does have a fairly high level of reads and writes (80% of the total reads and 97% of the total writes (the writes make sense, the reads don't). My hunch is that the excessive reads are due to index fragmentation (covered below). The good news is that if excessive IO becomes a bottleneck, throwing more RAM at the box should clean the problem up in a jiffy. It's also highly likely that a rewrite of the stored procedure – avoiding cursors will add other benefits – will add significant benefit should performance become a major consideration.
 - **Statistics:** A perusal of the statistics in the <dbname> database shows that they most likely aren't being updated regularly. SQL Server uses complex statistics, that it will create automatically if needed, to help ensure index precision and accuracy – better statistics means better query and index choices, which makes for faster queries with less CPU, and disk overhead. A regularly scheduled job can painlessly and effectively update statistics. I'll include information about this job in my Backup, Maintenance, and Replication review, but consider this a key priority to implement and rectify. (The review mentioned will provide you with a simple script to run to create a job that will correct this problem along with instructions for changing the frequency.)
 - **Rebuilding of indexes.** As data in a table is INSERTED, UPDATED, and DELETED indexes that track information about the data can frequently become fragmented. Clustered indexes, which define how the data is logically stored (i.e. if the phone book were a table it would have a clustered index along the following 'columns' in order: City, Last Name, First – but imagine that as people moved around, in and out, etc that the 'phone company' just scratched out the Hendersons in SmallTown and scribbled them in LittleTown – eventually things would be a mess). Indexes can be defragmented, which can have a DRAMATIC impact on performance. It is therefore recommended that key indexes be fragmented as frequently as needed – even daily if necessary on heavily used tables. For example, the index on dbo.dev_DataEvent is currently heavily fragmented (if it were a phone book it would have pages torn out of some sections and stapled here and there – extent fragmentation is at 98%, and page fragmentation is at 49%.) My Backup, Maintenance, and Replication analysis will include instructions for how to set this common maintenance 'operation' in to place.
- Stored Procedure Cache Hit Frequency:** By way of information, an analysis of how often stored procedures need to be recompiled (a semi-expensive operation indicating, generally, bad or tricky coding), showed that Cache Hit Frequency is VERY good: roughly 99%.

Coding Factors / Nit Picks

Coding Factors are a list of findings related to coding or logic that is less than optimal.

- **Superfluous use of BEGIN END statements in Stored Procedures.** A couple of stored procedures, such as `dbo.DeletePoi` wrap the procedure body in a BEGIN END block. This is permissible syntax, but is generally discouraged as BEGIN END can make the stored procedure look a bit like a User Defined Function which requires BEGIN END blocks (but also requires a RETURNS statement). A better way to terminate code logic is to simply append a GO at the end of the stored procedure to signify termination. Other examples: `dbo.Ensure_External_Asset`, `dbo.Ensure_External_Customer`, etc.
- **Non-Escaped Keywords.** A best coding practice is to escape out keywords in Stored Procedures, such as `[description]`, `[password]`, and `[name]`, etc (instead of just `description`, `password`, `name`, etc.). That said, there is no performance or functional penalty for not doing so (at least with the working/existing sprocs).

Other

Things you're probably aware of, but just in case:

- **Potential Logic Flaw.** The Stored Procedure `dbo.<sanitized>` uses a cursor, and 'returns' an error message when an underlying operation doesn't find an Asset to make inactive. However, because this is a cursor, only the LAST operation will be reported to the calling application. In other words, if 10 Assets are going to be 'inactivated,' the 4th could fail, and you'll never know about. So, either this isn't an issue – in which case there's no need to 'return' a response message. Or, if it is an issue, then it will fail silently. If you want to fix this business logic, and report failures your only option will be to break execution when a failure occurs with either a GOTO statement, or a RAISERROR operation (Let me know if you need further clarification.)
- **WARNING about the `dbo.A_owtest` Table.** This table looks like it is an older, non-used testing table. It looks like it could be easily deleted/dropped. However, it may be part of your Replication package. Ensure that it is not a published article before dropping it, or you will break Replication. (I'm assuming that you already know this, and that's why the table is still in production – but just in case (as this would be a perfectly honest mistake that would bring things crashing down...))
- **Security Concern.** Following up on the Security concern raised in the Good section, the best way to mitigate the risk of compromised web servers being able to be used in an attack to destroy or access all data is to change the permissions used by the application servers when connecting to the database. The best strategy would be to create a new SQL Server user account, something like `appservers` as a name (for example), and then grant that user ONLY the ability to EXECUTE stored procedures in the `<dbname>` database. (Make sure that the user is not a member of `db_owner`.) This way, the application servers can only execute stored procedures, and won't be able to delete tables, or quickly and easily select EVERYTHING out of them. (Let me know if you need help with the execution on this – I could create you scripts/instructions that will help in the transition in about an hour or two if desired.)

HEAP Tables. A large number of tables currently have no clustered index defined, meaning that rows inserted into these tables are not laid out logically in any order. Heap tables present a problem because controlling fragmentation on them is very difficult, and there are slight performance problems frequently associated with them. In general, heap tables should be avoided in all but the most specialized circumstances (there are FEW scenarios where heap tables make sense). However, none of the heap tables in the `<dbname>` database has enough rows to warrant any changes – I'm merely raising this point for your benefit and to be thorough.