

Guide to Developing Database Applications

Revision 2.0 – Jay Walters – 11 May 2009

© Jay Walters, 2003-9.

The purpose of this document is to provide some insight into developing applications which interact extensively with an RDBMS. Typically developers working on the application side do not have much database expertise and rely on tools such as O/R mappers to get the job done. The purpose of this paper is to provide rules and ideas which are useful when an application hits a constraint, and the developer needs to leave O/R mapping tools behind. That being said, many of the ideas in this paper are useful to any developer working with a relational database.

This document provides a beginning framework for the design of applications which use RDBMS technology. The first section provides a small set of rules which are helpful when designing the database schema. These are all mom and apple pie, and are covered in any introduction to RDBMS design and programming. I expect to extend this section over time. The second section contains some rules for the application that is using and manipulating the data. The third section provides some things to watch out for and avoid. The fourth section provides some additional techniques which may be useful in specific circumstances. When it is relevant a brief discussion of platform specific issues follows the general discussion. Appendix A provides a brief overview of RDBMS terminology for readers who need a review. It introduces and defines a large number of terms, which are used through the paper.

Background

I have developed this set of ideas over the course of a 25+ year career working with applications built on top of RDBMS products. The bulk of my experience has been using Oracle, including both as a consultant for Oracle and at companies using Oracle. I have also worked as an Informix DBA and for many years as an application developer using Sybase. I regret that I have little direct experience with MySQL, though I expect the bulk of this information is platform independent.

When building Object Oriented applications, much of the time we can rely on an O/R mapping tool, i.e.. Hibernate, to handle our interaction with the database. O/R mapping tools smooth over the impedance mismatch between the RDBMS and an O-O programming model, however they do this at a cost in longer code paths and potentially undesirable side effects, e.g. early versions of Hibernate cached all objects which flowed through. O/R mappers are most useful for CRUD (create/read/update/delete) type applications. They are not particularly useful for applications which perform significant work in the database itself (breaks caching), do bulk reads or writes (can't make it much more efficient than raw SQL) or applications with difficult performance constraints. The bulk of this paper concerns the times when we cannot use this extra layer between our application and the database.

Guidelines for Database Design

The distinction between logical and physical database design allows us to defer the physical decisions about our database until after we have identified the access paths to the data. In plain English this means we can design the tables and indexes separately, and adding an index is mostly invisible to the application logic. It may have a big impact on space usage, locking or some other detail, but we consider those items part of physical database design..

Specifically we want to know most of the sql statements that will be used so that we can optimize the set of indexes on a table within the following constraints:

- Minimize the number of total indexed columns
- Allow all joins to be on indexed columns
- Where possible cover queries with an index
- Allow all appropriate filtering (non-join where clause) to use indexes

In the next sections I will provide further details towards each of these goals.

Minimize the total number of indexed columns

We want to minimize the number of indexed columns so that we can reduce the cost of DML operations, as well as to reduce the amount of space used by the indexes. Each time a row is inserted all the indexes need to be updated. Most modern RDBMS software uses a variety of key compression algorithms to minimize the space required for an index, but it can still easily require a significant fraction of of the space used by the actual table for a single index. As you can imagine, a table with multiple indexes could use more space for the indexes than for the data itself.

Allow all joins to be on indexed columns

This rule helps the query optimizer to use a looping query plan (often called nested loops) where for each row in the driving table it looks up via the index the corresponding row(s) in the joined table. If the columns are not indexed, then the optimizer will be choosing between a table scan for each row in the driving table or a sort-merge join which can be very inefficient for small result sets from large tables.

Note that for a nested loops query plan, the best case is that we make one index probe of the non-driving table for every row selected from the driving table. It is important to understand whether this is good or bad, given your specific database.

Where possible cover a query with an index

A covered query is one in which all the columns in the select clause are in a composite index which is used by the optimizer as the access path to the data. In this case the actual data row is never touched, which will save a logical I/O for each row in the result set. Sometimes we will add an extra column to a composite index in order to cover a frequent query with the index even though the column isn't part of the where clause of the query.

Note that this guideline is a tradeoff with the guideline about minimizing the number of columns indexed.

Index all filtering (non-join) where clause phrases

Here we want to be able to perform the filtering operation entirely in the index, though this is not always feasible. Because the index contains only the columns in the index and a reference to the location of the row in the table it is typically more compact than the table and thus more likely to remain in the database cache and quicker to load if not in the cache. For these reasons we want to perform as much of the query filtering and joining using indexes and only when we need to pull in columns for the result need to touch the data pages themselves.

Final Ideas

Finally, there are times when we don't need or want to build an index. If a query is going to return more than 20% of the rows from a table (this threshold may be different on different systems, but it is surprisingly low) the optimizer should be using a full table scan, so there is no need to use an index for the filtering of rows. If a suitable covering index exists, the RDBMS can use an index scan instead of a table scan to find the rows.

One more trick to beware of is how nulls are handled. Some RDBMS products don't index nulls in non-unique indexes. That means if a row contains a null in some column A which is indexed, that row will not appear in the index. If you query using a condition such as 'is null' then the RDBMS will have to perform a table scan to find the rows.

Guidelines for Application Development

This section of the document will discuss rules of good style and various approaches that can sometimes be used to improve performance of an application.

- Use Set Operations
- Filter and Sort in the Database
- Re-Use Query Plans
- Use Surrogate Keys
- Use Null Properly
- Reduce I/O (Disk and Network)

Each of the bulleted rules is discussed in the section below. Many of the rules are independent, however Use Set Operations is the most important, and the one most commonly broken.

Use Set Operations

This is a very important guideline for building applications on top of an RDBMS. The underpinning of SQL is set theory, and so relational databases are very efficient at processing set operations. It is almost always more efficient to execute a single set oriented operation inside the server than it is to iterate over a list either internal or external to the server.

Typically when you find widespread use of cursors, or iterative constructs in a 3 G/L this rule has been broken. The classic implementation of the anti-pattern is shown below:

```
Execute a Query which returns set A
For each row in A
  Update some row(s) in set B
  If update fails, log message
```

If we rewrite this operation as a set operation it would look like:

Update set B with data from A based on a join of sets A and B.

Another advantage of the second implementation is that it will do only the minimum work, whichever set is smaller will drive the join and thus ensure close to optimal behavior across a variety of input conditions. We can easily imagine a scenario using the first implementation where set B is much smaller than set A, but we still iterate over each item in set A.

This rule sits on the RDBMS side of the classic RDBMS/OO impedance mismatch. Object Oriented 3GL developers and O/R frameworks will generally pull data out of the database and into memory for iterative processing because it is more natural to them, and in the case of O/R frameworks because it doesn't interfere with their caching strategies. While these caching strategies are usually beneficial in the CRUD case, they quickly get in the way of more complicated applications.

Filter and Sort in the Database

This is an obvious statement, but filtering and sorting in the database almost always is faster than pulling the data out of the database and performing those activities in application space.

I will always remember one of my first consulting assignments when I worked at Oracle. I was providing DBA support as a sub-contractor to a well known consulting firm. There were some significant performance problems with some reports written by a junior developer. When I reviewed the source code for the reports I very quickly identified that the developer was not filtering the result set in the database, but was returning every row to the client application where he was performing the filtering. By adding a where clause which used an index to the query we saw dramatic performance improvements. Of course this is an easy win, but not using indexes properly and pulling across more data than is needed is probably more common than you might think.

Use Indexes where Possible

Here we want to make sure that query plans use indexes when they are available. There are two common ways where we fail on this idea.

First we use an argument in the where clause that does not match the type of the indexed attribute, and the RDBMS has to perform a type conversion on the indexed attribute. For some reason RDBMS products usually convert the data in the table to match the argument you specify (this is because a type conversion may change or truncate the data) and this means the index will not be used in this circumstance. We can also run into this problem when using functions in the where clause.

Second, when a table has a composite index, the where clause does not include the leading fields in the index, but only some trailing fields and so the index cannot be used. When the optimizer selects an index, it can pick a composite index when the leading columns of the index match the fields in the where clause because it can just ignore the trailing fields, but it cannot do the converse (match the trailing fields and ignore the leading fields).

A class problem in this genre is trying to use one side of the expression as a string, and the other side as a numeric or date type. Luckily it is usually easy to correct the SQL.

Make the Optimizers Life Easy and Your Code Simpler

Use the simplest queries that will get the job done. Over the last 20 years the SQL syntax has become increasingly rich allowing a variety of ways to express any given query. In the end most queries will be flattened out into joins by the parser and optimizer, so from a transparency point of view it is best to write the query that way from the start. This will help you think like the optimizer and understand what it is doing intuitively by reading the query.

You should always know the query plan you want, and ensure that the SQL helps the optimizer along that path. This probably conflicts with what the vendors will tell you, that their optimizer is very smart, but in the end they are not perfect.

Re-Use Query Plans

Query plans are the structures that are used within an RDBMS to identify the set of rows that should be operated upon by a DML or SQL statement. When the server receives the statement from the client program it parses the statement and then builds a query plan. Later this query plan will be used to process the statement. If the parsing and building of the query plan can be skipped then the server will be more efficient.

Sometimes the drivers for the database will also perform some optimizations on the client side if they know that a statement will be reused. The most common mechanism that can be used to notify the driver and/or the server that a statement will be reused is called a prepared statement. Some RDBMS products support the concept of the prepared statement in the server, and others do not. Inside the server creating a stored procedure is the most common mechanism for identifying code, which will be repeatedly executed. Using these two mechanisms will allow the database to efficiently process your client requests.

On the plus side, these optimizations can provide significant performance improvements. It is widely quoted that the conversion from dynamic SQL to prepared statements can improve the performance of an Oracle application by 30%.

Several times while I was a consultant with Oracle I was able to get quick improvements in application performance by converting from dynamic SQL to prepared statements. As a consultant use of prepared statements was one of the first things I would look for on any engagement.

One drawback to this sharing of query plans is that one must ensure that all users of the stored procedure of SQL statement should use the same query plan. Widely disparate parameters may cause the plan to be sub-optimal sometimes and optimal sometimes. Unfortunately, the RDBMS usually makes these decisions, but some thought on the part of the developer can ensure results are as expected. An example of this would be given an index on a column which has some values with very low cardinality and some values with very high cardinality, if the query plan is frozen we might do an index scan when we should do a table scan, or vice versa.

Database Specific Behavior

Oracle supports prepared statements within the kernel of the RDBMS. As a result using prepared statements with Oracle is very advantageous. Oracle caches SQL statements within the SGA (System Global Area). When a statement is submitted for parsing, first Oracle computes the hash code and checks for a match, for each matching statement the submitted statement is compared to the statement in the cache in a case sensitive manner. If a match is found, then the query plan from the cache is used. An unlimited number of sessions can use the same statement in the cache – including the parse tree and the query plan. In addition to these elements, statistics are also kept for each statement in the cache, which allows for identifying statements that are inefficient.

Off topic but important point. These statistics stored in the V\$ tables which include items such as logical I/O's, physical I/O's and uses of the statement allow you to rapidly identify which statements are the least efficient and most commonly used so without looking at the actual applications, you can quickly identify where your tuning energy should be spent.

Sybase does not support prepared statements within the kernel of the RDBMS/ Sybase caches stored procedures within the procedure cache. In this cache there will be one or more copies of the parse tree and query plan for each stored procedure. Sybase does not allow multiple sessions to share the same cache entry, so if multiple sessions are accessing the same stored procedure at the same time then there will be multiple copies of the stored procedure in the cache. There are options that can be enabled in the client side driver so that it will create a stored procedure in the server for each prepared statement in the client. This allows the server to cache the query plan for specific statements. This option is not enabled by default. It does not provide any benefit over using stored procedures and non-prepared SQL statements.

Use Surrogate Keys

It is quite common for domain keys to change during the life of an application. This can be both in the actual value for a simple key, or for the structure of a smart key. So for a smart key the business owns not just the contents, but the structure; two things that can be changed to cause pain in the database. Using smart keys is definitely not smart! Another common issue with domain keys is key reuse. This can cause no end of pain if historical data of any type is being stored. Propagating these changes to domain keys through the application and the database can be a very difficult and time-consuming operation.

Rather than using domain keys, use surrogate keys, which have no meaning besides being the key. A surrogate key is a key created within the application, typically a sequential number. Surrogate keys are fixed for the lifetime of the object and are typically never reused. Assigning surrogate key values can be performed within the database, or by the application.

Most RDBMS products have specific extensions to provide high performance sequential id generation. What they won't tell you is that inserting records with sequential ids into a database table with a primary key on that id isn't always a great idea. It will generate hot spots where there is contention for database pages in both the table and the index. It can require significant additional tuning in order to make this approach work in an environment where high throughput is required. You also need to carefully consider your approach if the database is distributed across multiple nodes. There is more discussion of this issue in the section on partitioning.

Once you have a surrogate primary key, you can use the business key as a secondary key for lookup but not for linking records within the database. This is a much more robust and flexible structure.

Once near the beginning of my career I worked on a project with a large chemical corporation's sales force. They used a hierarchical structure for their account number that was the domain key for their account data. The primary key was a smart key containing three distinct pieces of information, sales district, sales area (a subgroup of district) and a account number unique within the sales area. Once a year they would re-organize the sales force and in the process they would have to change the primary key value for a large fraction of the accounts in their database. This required application downtime and not a small amount of trouble to convert the database to the new accounts every year. Had they used surrogate keys, the account number would have just been another attribute and could have been easily changed as part of normal operations.

Another example of this type of mistake from the financial domain is using a ticker or security identifier as a permanent key for a security. Tickers, Cusips, Sedol's all change over time even if the identity of the security does not, and while re-use is very slow or non-existent for Cusips and Sedols, reuse of tickers can cause real headaches if you're using ticker for a primary key someplace.

Database Specific Behavior

In general the difficult part of using database specific extensions for the creation of surrogate keys is how they cache the numbers to enhance performance. Usually a block of ids is cached in memory and when that block is exhausted a new block is requested. If the database shuts down before the block is used, the unused portion of the block is wasted, leaving a hole in the sequence in your table. Generally you can tune this block size to improve performance or reduce the size of the holes.

Oracle provides the sequence object for generating a sequence of numbers. You can configure the sequence object to start with a specific value and to increment values by a certain amount. You can configure how many ids are in each block. Values are read from the sequence using a select statement.

Sybase provides the identity column type. There is a system variable @@identity that holds the last generated identity value for the current session. You can configure the number of ids in a block.

Use Null Properly

The Null value within a database refers to a lack of information, which is distinct from having a value, such as 0. This is analogous to a reference in Java where a null value indicates there is no referenced object. In the SQL world, null has very distinct behavior, such as null values don't appear in aggregate functions.

Naive programmers very often use "magic" numbers to indicate the concept of null, usually this would be a 0, -1, 9999; something that appears obvious at the time. These "magic" numbers tend to become maintenance problems over time as new programmers won't immediately know that "-1" is an unknown age, though an experienced one will often be able to guess this pretty quickly. When using aggregate functions within the database, rows with null columns used in aggregates will not be included, but rows with the magic values will need to be explicitly ignored within the query expression. Finally, these "magic" numbers usually need to be hidden from the user, so even the presentation logic may need to be aware of them.

Luckily the wide adoption of languages that consider numeric quantities as first rate objects, i.e. Java, C#, means that programmers are learning the concept of a null value.

Reduce I/O (Disk and Network)

When an application runs slowly, one of the first areas to investigate is how much I/O it is doing. Sometimes this can be obscured by the use of external components, but most database access can be identified through tools of one sort or another. A reduction in I/O will generally result in performance improvements, whether the I/O is disk or network related. This analysis will go forward under the assumption that we want persistent transactions.

Within a standard RDBMS there are three sources of disk I/O that we will need to consider. The first is writing to the transaction log. Whenever a transaction is committed

by the database, all the changes in that transaction must be written to the transaction log on disk prior to the commit call returning to the caller. The transaction log is written sequentially, and if the server is busy it will have some method of combining log writes from multiple transactions. If you measure the throughput of an RDBMS at some point it will be limited purely by the speed of writing to the transaction log.

The second source of disk I/O is the checkpoint. A checkpoint is when all the dirty blocks in the database cache are written to the disk. Checkpoints can be very disruptive for a few reasons. The first is that because they are writing dirty blocks from the cache, the larger the cache the longer the checkpoint. Second, updates to data blocks may be sequential when adding new data, but updates to index blocks other than the primary key are not. Furthermore, strategies to enhance performance typically work at spreading out writes to avoid contention within the database cache, but this can aggravate the situation as all the dirty blocks eventually need to be written to disk. There is generally some window however small during a checkpoint when the database cannot complete new transactions; this leads to increased transaction latency during a checkpoint.

The third source of disk I/O is reading blocks into memory in order to service query requests. Usually when a server is fetching blocks for an execution plan other than a table scan, each block fetched will go to the head of the LRU queue of blocks in the database cache. When a table scan is performed, the blocks typically don't go to the head, depending on the database they may or may not even go into the cache. If they were put into the cache, it would be like a cache flush on every large table scan and performance would suffer dramatically.

All of this long explanation of the various types of disk I/O, which occur on the server, provides a lot of information, but how can you act upon it. First spread the data over as many disk spindles as possible. This will increase the overall disk throughput and thus database performance. Separate log and data on the disks, this will keep log writing speed high by not requiring the log disks to respond to random read and write requests. Minimize the number of indices created on each table; this will reduce checkpoint requirements during heavy insert traffic. Avoid lots of small transactions, this will make better use of the log writing I/O subsystem.

Access to a modern RDBMS is almost always via a network connection, whether the server is local or remote. This means that minimizing network I/O will improve application performance. Some topics already discussed address this issue, for example filtering on the server and performing set operations in the database vs. iterating in the application will ensure that the database server only sends data to the application it actually needs. An additional strategy to minimize network I/O is to use stored procedures, which can reduce network traffic by batching up groups of commands and providing default values.

I worked on a project where we were very much limited by I/O speed. We built an extension to the NT performance monitor which allowed us to monitor the LVM on the database server as well as application metrics like latency and throughput from the

application servers. This allowed us to understand that checkpoints were our bottleneck, and not transaction log writes. Tools which allow the mixing of application metrics and various database and operating system metrics are very useful additions to the performance tuner's toolkit.

SQL Implementations and Vendor Extensions

One of the areas, which have only recently been addressed by the SQL standard, is the procedural language provided by the various vendors for writing stored procedures, triggers and anonymous procedural blocks. As a result each vendor has their own language, and the capabilities of the various languages can be quite different. Another development is the addition of a java virtual machine to many of the major databases such that these stored programs can be written in either Java or the server's native procedural language.

At one point these java implementations were much slower than those outside the database, but I am not up to date on how performance has changed since their introduction.

Potential Problems

This section will discuss a few structures that you may find when reviewing an application. They are identified in this section, because they are often an indicator of problems with the application design and implementation.

I once read a very good paper in the IEEE Software magazine whose essential premise was that one can get a proxy for application quality by database design quality. This is a metric which can be measured for COTS software during an evaluation prior to the customer getting enough experience with the software to estimate the quality using more traditional metrics. My intuition and experience tells me a poorly designed database will ensure a poor quality product.

Mismatched Datatypes

This problem manifests itself in ways such as a primary key in one table which is a varchar and in another table a numeric datatype. It might also include storing the key in both formats, for example using as both a character string and a numeric value. I have seen this used when the actual key value was generated as a numeric using the sybase identity mechanism, but for some reason someone wanted the key also as a character string.

Temporary Tables

There is nothing wrong with temporary tables in and of themselves, but if they are used instead of building up joins, or large amounts of data are inserted into them they can cause performance problems. Temporary tables can be indexed after creation and this may speed up queries which need to access a temp table. Elimination of the step which writes the temporary table can also be a big help, unfortunately sometimes eliminating an explicit temp table just pushes it into the query plan of a SQL statement. Usually this would be an improvement because this temp table does not need to exist across

transaction boundaries and is anonymous (not visible to any other user) so it can be managed more efficiently.

Note that explicit creation and population of temporary tables can be an indication that procedural techniques are being used when set techniques are more appropriate.

Sometimes the RDBMS just can't do as well as you can by creating a temporary table and using several smaller steps rather than a single large multi-table join. I have seen this with Sybase.

Recursive Data Structures

I've included this section under potential problems not because recursive data structures are always a problem, but instead because relational databases are notoriously inefficient at processing queries which operate on recursive data structures such as trees. This is one area where an object-oriented database may provide superior performance.

Experienced developers will try to avoid these type of structures up front, but sometimes it is the only way to build the application. If you must support recursive data structures then you should pay close attention to the queries that are written to access them.

Some databases, e.g. Oracle, have syntactical extensions to their SQL to support recursive queries to be expressed very simply. However, they don't provide significant performance improvements.

In my experience it has been easier and quicker to flatten the tree and store it in a database table that way, rather than to try and store the tree in the RDBMS recursively. This is generally an option when the tree is small and can be held entirely in memory, or can be partitioned so that a working set can be held in memory. It can also be easily done when the tree is relatively flat,. Application programming languages provide references and data structure libraries which can make the construction of a tree much easier outside the database.

Cache Tables

If you find a cache table within the data model this is a good indicator that something might be wrong. This is an indicator of gross denormalization in that the same data is actually being stored in two places, one because it is too slow to get to it in the other location. This can happen if data that should be in one table is spread across several tables and the developer couldn't identify from which table it should be fetched. Another scenario for this is when the developer couldn't see how to pull the data out of the base table efficiently. In both of these cases the solution is to resolve the problem via normalization and combining tables.

I once saw an example of this design where the developer spread data which should have been in one table across several, worse yet he didn't know which table the data might be in so finding data required a stored procedure to iterate over all the tables looking for the data. He needed to identify the most recent data of each type. In order to simplify

his program he then extracted the most recent data and put it into a cache table. Putting all the data into a single table and then writing a suitable correlated sub-query with proper indexing was used to simplify this entire design.

Use of Functions in the Where Clause

Database vendors always provide at least a minimal set of functions which operate on the data. These may be type conversions, date manipulation functions, string manipulation functions or some other type. Extra care should be taken when using functions within a where clause. Improper use of functions in an expression can negate the use of an index. An example of this would be `...a.x = upper(b.y)...`. In this case we will not be able to use an index on y as part of joining table b to table a. If table b is the driving table, then this will require us to perform some computations during the join, but will not keep us from using an index on column a.x.

Other Techniques

This section provides some additional ideas for building applications. These ideas don't apply to every situation, but perhaps they'll stimulate some out of the box thinking on your part.

Scale the application with the data

In the section on Set Operations I alluded to the advantage of set operations over iteration in that the server can identify the optimal query plan at runtime and thus let the smaller set drive the operation. This one basically comes down to keeping the algorithms within $O(n)$ or better whenever possible. An example of this would be, if we were trying to update a catalog with new prices, rather than use a cursor which requests the new price for each entry in the catalog – use a set operation to update the catalog with the set of updated prices. Changing from a row oriented approach to a set oriented approach will yield some improvements and driving the operation of the smaller set will provide additional improvements if the we are only updating some of the prices.

I recently ran across a good example of missing this guideline in an application for loading historical pricing data into a financial application. Every day the current price for a large and changing of securities was to be loaded. Rather than the application loading the data, which was actually prepared, it iterated through the list of all the data that could be sent asking for the data one piece at a time. While this algorithm was good for debugging, and not so obviously slow when loading all the data, it was grossly inefficient for the case of a partial load. In the end we were able to dramatically speed up this application by changing to a set operation where a single update statement loaded the data.

Increase Efficiency Under Load

As the system becomes loaded we want the system to become more efficient. This will allow it to scale well. It is much easier to describe than it is to implement successfully.

This generally comes down to a trade off between scalability and performance, or latency and throughput. If we can accept a slowdown on each transaction, then we can process many more of them per unit time.

One example of this is the piggyback log writes that RDBMS vendors use to reduce the number of log writes/commit to < 1 when the database is under a large load. Before a commit operation can complete, all changes made by the transaction must be written to the log on disk. Generally this constitutes a stream of small sequential writes into a single file. With lots of transactions going to the log, more than one transaction usually fits within the size of a single write to the log device, and thus a single write can log more than one transaction. When the RDBMS is loaded, it might have multiple transactions attempting to write to the log at one time, so batching them up has little impact on the latency of each transaction, but allows more transactions per unit of time to complete.

Designing an application to have this characteristic is often impossible, but when it is possible the payback in scalability will be very worthwhile.

Do you need ACID Application Transactions?

This is a trick that you can pull out of the hat when performance is critical and the application has certain favorable characteristics. One of the significant features of RDBMS products is the fact that they provide ACID transactions. ACID is atomic, consistent, isolated and durable and refers to properties of transactions.

While atomic and consistent are pretty much a given, you may be able to sacrifice durability and/or isolation. The largest cost involved in a durable transaction is the cost of the disk write. If you can accept non-durable state and just store it in memory, or else store it at the client system (in their memory) then you can speed up processing dramatically. Of course, this brings the issue of recovery to the forefront, but you may be able to make a tradeoff of this sort.

Many years ago I worked on a project where we built a distributed central system for processing lottery and pari-mutuel betting. We had a variety of types of application servers for handling different types of operations and we needed to support high transaction volume as well as an simple programming model for the servers. Our solution was to use a cookie in the RPC calls with the state data in it. We could always create the state from the transaction stream, but we didn't have to explicitly store it. We didn't need to encrypt our cookie as the state didn't need to be secret, we just used a cryptographic checksum to ensure that it wasn't tampered with by the client.

You may also be able to replace physical isolation with logical isolation. The normal case is for a one to one mapping of application transactions into database transaction. If the individual database transactions are by their nature isolated, and mutually exclusive then we can batch multiple application transactions into a single database transaction and if it succeeds we have achieved improved efficiency. As an added bonus, if the individual application transactions manage to perform operations which can somehow be aggregated, e.g. update counts or totals, then by batching application transactions, which

might update the same table, we can reduce the aggregate number of update statements that need to be executed. Of course if the transaction fails, then each application transaction may need to be individually processed and this will slow processing. In general this trick will trade-off increased latency for higher throughput. This is an example of improved efficiency under load as described above.

The waging central system I worked on was the first system of its type to use an RDBMS for persistence. When we began the serious benchmarking of the system we found we were limited to around 100/writes per second to the disk array where our database logs were located. As a result we could manage only 100 transactions per second in the database, and we needed closer to 1000. The application was already partitioned in a shared nothing architecture, but we could not tell the customer we needed 10x the number of computers. We modified the software to batch client transactions into groups as large as 8. In this way we were able to process close to 800 client transactions per second when the load was high, e.g. there were enough client transactions to batch. At low volumes we didn't have enough client transactions to batch, but we didn't need to batch them.

Some notes on scaling through partitioning

One way in which to scale up a database application is through sharding or partitioning of a database table. We can horizontally partition, which means putting different rows in different partitions, or we can vertically partition which means putting different columns in different partitions. For practical reasons, horizontal partitioning is the most widely used. It is an important part of a distributed shared nothing architecture, which is an optimal architecture for scalability.

When a database table is partitioned, the accompanying indexes are also usually partitioned, making management of the data easier since each partition is complete.

A DBA might choose to partition a database table for several reasons, including improving concurrency for insertions, simplifying deletion of data or improving query performance by making it easier to parallelize.

Most enterprise level database products support physical partitioning of tables within a database. This is most often done to optimize write I/O operations so insertions and updates to the table can be spread across more disk spindles. Since it can be done at the physical level, the application developer does not even need to be aware of it.

Database tables can also be partitioned across servers. This is generally not supported well by the RDBMS products and needs to be managed on an ad hoc basis by the developers and the DBA. This would typically be done in order to parallelize query processing, either applying more CPU resources or more memory than can be held in a single machine.

Data can be spread across the partitions using either a strategy which spreads the data evenly and randomly across the partitions, or a strategy can be used which locates

specific data on specific partitions. The strategy to be used depends on the purpose. If the designer is interested in parallelizing searches for data, then it should be spread evenly and randomly so a large number of partitions can be searched in parallel. This strategy can be used to reduce latency during queries against large datasets.

Partitioning also improves insertion performance. With multiple concurrent insertions going to different partitions, lock contention on both index and data blocks can be reduced.

Finally, you might want to adopt a partitioning strategy to cope with data which needs to be archived. For example one could store data in partitions by date and when a certain partition is no longer needed it could be just removed from the table in a single meta-data operation rather than requiring an expensive delete operation. Note that partitioning a table to cause locality of reference works against the concept of parallelizing queries, as it puts data which is related together on the disk, rather than spreading it across partitions so multiple threads of execution can find it more quickly.

Restructuring a table can be a costly because the database needs to move all the data around, this provides an opportunity to improve on the situation by executing the partitioning in your application. You might be able to modify you application to support on the fly partition changes, and hot restructuring of the table as well. Most databases do not support hot restructuring as the table needs to be locked while rows are moved. If you are using specific keys to identify the partitions, then you can add partitions for new key values without impacting existing partitions, but if you've used a random scheme then adding partitions requires moving many rows in order to ensure the data is evenly distributed. It would be very good operationally to design the application so the data can be migrated between partitions which the system is running.

Query Optimization in a Partitioned Environment

When a SQL or DML statement includes enough information that the database engine can identify specific partitions where matching data is stored, then the database engine will consider only the relevant partitions. On the other hand, if the query doesn't narrow down the partition where the data exists, then all partitions will need to be examined. Depending on how the data is partitioned and for what purpose, these might be good or bad characteristics.

As an example, if one partitioned a table on the primary key using a modulo or hash type function that randomly distributed the data amongst partitions, a request for a record by primary key could be directed to the exact partition which contains the data, but queries for records by secondary keys would run in parallel across all the partitions.

When I worked at Bidder's Edge, in the late 90's our search engine used a modulo scheme on the primary key to spread data out across first 2, then 4 and later 8 database servers. We stored index data in these "search" servers so that we could parallelize certain types of queries across all the auction items in our database. In a separate

central database we kept a 'master' record for each item. After an auction was complete, we removed the record from the "server" server, but kept it in our central database for certain queries and historical reporting.

This illustrates a few points, first we actually denormalized our database onto several shards for faster query performance. We kept a single copy of all the data for direct access by primary key, and a second copy of the data structured for query performance and stored as multiple shards. We were able to parallelize queries across all relevant database shards (sometimes we could rule out one or more shards) within the application). If we knew the primary key generally went to the central database directly. We could not run hot re-organizations and I had $\log_2(8)$ evenings when I spent the entire night migrating data between servers as we scaled up the website. In hind site, had we used a schema that abstained from shared elimination we could have performed hot migration of the data which would have provided better uptime for our website, and saved me a few long lonely nights.

Free lists and physical design

If you are trying to optimize for insertion performance you will want to examine options within your database for multiple insertion points. This may be called multiple free lists (oracle) or something else. When using a surrogate primary key designers will often use a table structured index, or clustered index and cause a hot spot for insertions on the tail index blocks. It is much better to use a more random key for the clustered key if possible to reduce contention and eliminate the hot spot.

Appendix A - Introduction to RDBMS Terminology

This section of the document will provide a brief overview of relational database technology in order to ensure a common language and understanding of the basic behaviors. Over the last several years in working with junior programmers I have found that starting with a basic understanding of how relational databases work to be a critical success factor.

Data within a relational database is stored within **tables**. A table is essentially a container for data. A table has a fixed set of **attributes**; these attributes can be mandatory or optional. Optional attributes allow the value **null**, mandatory attributes do not allow the value null. Another name for these attributes is **columns**. Each instance of these attributes is commonly called a **row**. It is good practice to identify a set of one or more attributes that uniquely identifies each row within a table. This set of attributes is known as the **primary key**. If the primary key contains more than one attribute it is known as a **composite key**. A single attribute primary key that includes sub-fields that may be separately used by an application is a **smart key**. We can use a **domain key** or a **surrogate key**, which is a new attribute we add during database design for the sole purpose of uniquely identifying each row in a table. When we create a relationship between two tables, e.g. A and B, the primary key from table A will be stored in table B as a **foreign key**.

Retrieving data from an RDBMS is done using the Structured Query Language (SQL) or just queries for short. Queries contain a variety of clauses, some of which are mandatory and some required. We must always identify the attributes to be returned (**select clause**) and logically where the data is to be found (**from clause**). We can optionally specify a **where clause** which provides filtering and joining, an **order by clause** for sorting, and a **group by clause** for aggregation. Some of these clauses can contain **sub-queries**. Sub-queries are embedded queries within a larger query that can be used for filtering the results. **Correlated sub-queries** use information from their parent query to filter their results. **Non-correlated sub-queries**, are independent of their parent.

When we match key attributes in two different tables as part of a query, we are **joining** the two tables. There are two types of joins, **inner** and **outer joins**. Inner joins require a matching row or rows in each table, so it is an intersection of the two tables. An outer join from table A to table B will include all matching rows from table A, and for each row the corresponding row or rows from table B. If there is no row to join to in table B, the database provides a single row with nulls for every attribute to the join. Most vendors provide a shorthand notation for outer joins. If the join is realized using **nested loops**, the table referenced in the outer loop is known as the **driving table** for the join.

Use of a surrogate key instead of a **domain key** decouples the **referential integrity** of the database from the application use of the domain key fields. Translated into English, this means that since we don't use domain keys as foreign keys, changing the domain key values won't require changing all the foreign keys within the database.

Indexes are additional objects created within a database that can be used to speed up access to the rows within a table. They also enforce uniqueness constraints, but otherwise their existence or non-existence is not visible to the application. The most common type of index uses a **B-tree** data structure. Most databases support storing the table data in the leaves of one B-tree, this is called a **clustered index** or an **Index organized table**. Another type of index commonly used in a data warehouse is a bitmap index. **Bitmap indexes** are very efficient at indexing fields with a small domain, however they are very expensive to update so they can't usually be used within an OLTP application.

A **view** is a stored query. It can be thought of as a virtual table. It can access one or more tables. Views can be used to enforce security, filter rows, or to augment the data in a table with computed attributes. If a view is on one table, the underlying table can usually be updated through the view. Multi-table views generally don't support update operations.

Stored procedures are programs written in a database specific language for execution within the database server itself. Stored procedures are often used for security purposes, to reduce network I/O or to collect operations together to simplify client side programming. **Triggers** are stored procedures that are attached to a table and defined to execute when a specific operation (add, update, delete) occurs on the table. Triggers can be used for referential integrity, to collect an audit trail, or to add computed columns to the table they are attached to, or to another table.

Referential integrity can be implemented via **declarative referential integrity, constraints**, or via stored procedures or triggers. Declarative referential integrity constraints can be used to handle cascading deletes and the restricting deletes if related rows exist.

We can generally divide the set of valid input statements into three sets, **queries**, data manipulation language (**DML**) and data definition language (**DDL**). Most products also support a procedural language, e.g. PL/SQL or Transact SQL. Typically we consider all these types of statements part of SQL, though SQL originally only referred to the query language itself. When the database processes a statement, it first parses the statement and then the **query optimizer** is invoked to generate an optimal access plan. Most databases will cache parse trees and query plans in some way as a performance optimization.

Database design is a two-step process, the first step is known as **Logical Database Design**. An **Entity-Relationship Diagram** (ERD) is used to model the logical database design.. It is the stage when the attributes are assigned to entities and relationships. Later entities will be mapped to tables, and relationships will be mapped to keys unless they have attributes in which case they'll be mapped into tables as well. **Physical Database Design** is the stage when indexes are added, and when the actual location within the database for the various objects is established. One of the strengths of the relational model is that application developers can work with the logical data model,

blissfully unaware of the physical data model. In an OODBMS changes to the physical design immediately percolate up to the application developer.

When a client program connects to the database server a **session** is created. Most database products provide some way to view all the currently active sessions on the server. There are many parameters that can be set for a session, such as **transaction isolation level** and **autocommit**. The transaction isolation level refers to items such as whether a given transaction honors locks placed by others when reading data or locks rows for itself when reading. Autocommit just identifies whether every statement is a separate transaction, or whether the application will manage transaction boundaries.

When the database is executing a query it may track two types of I/O operations, logical and physical. A **Logical I/O** is when the database accesses a specific page during statement execution. A **Physical I/O** means that the page was read in from disk. We want to minimize physical I/O operations as they are very slow. At the same time we also want to be aware of the number of logical I/O operations required as well. An old rule of thumb for Oracle was anything more than 15 logical I/Os per row returned from a query merited examination of the SQL to see if it needed to be tuned.