

# ***Guide to Developing Database Applications***

## **Revision 1.0 – Jay Walters – 7 November 2003**

© Jay Walters, 2003.

The purpose of this document is to provide some insight into developing the data access portion of applications. Typically developers working on the application side do not have much database expertise, but increasingly they are called upon to make sound decisions when it comes to data access.

The first section provides a brief overview of the key parts of an RDBMS. It introduces a large number of terms, which will be used through the paper. The second section of the document comprises some guidelines for well-developed applications. The third section covers some things to watch out for; each of the items in this section is a signpost that trouble may be brewing. The fourth and final section covers some advanced techniques, which are of use in certain limited situations. When it is relevant a brief discussion of platform specific issues follows the general discussion.

### ***Background***

I have developed this set of guidelines over the course of a 17 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 with alongside Sybase DBAs.

When building Object Oriented applications, much of the time we can rely on a COTS or open source O/R mapping tool to handle our interaction with the database. The bulk of this paper concerns the times when we cannot use this extra layer between our application and the database. Usually we find ourselves in this position when performance is critical, or there are some other unique constraints on the application.

### ***Introduction to SQL***

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**.

## Guidelines for Database Design

The distinction between logical and physical database design allows us to defer the identification and creation of indexes on our database tables until after we have identified most of the access paths to the data. 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. 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 as much as 40% of the space used by the actual table. As you can see, a table with 3 indexes will likely use more space for indexing the data than for storing the data.

Allow all joins to be on indexed columns. This constraint allows 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.

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.

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.

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 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 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 to them, and in the case of O/R frameworks because it doesn't interfere with their caching strategies.

## ***Filter and Sort in the Database***

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 one of the big 5 consulting firms. There were some significant performance problems with some reports written by some junior developers. When we 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 more common than you might think.

## ***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 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 the prepared statement. On the server side 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 performance improvements of up to 30%.

One drawback to this sharing of query plans is that one must ensure that all uses 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. Care should be taken when using this type of approach.

## ***Database Specific Behavior***

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.

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.

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. 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.

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.

### **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 an 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.

3GL 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.

## ***Reduce I/O (Disk and Network)***

When an application runs slowly, one of the first areas to investigate is how much I/O is it 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.

Within the database 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 by 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 is not. Furthermore, strategies to enhance performance typically work at spreading out writes to avoid contention within the database cache aggravating this situation. There is generally some window however small during a checkpoint when the database cannot process 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.

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.

## **Database Specifics**

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. A recent 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.

Some notable differences are the Oracle's procedural language, PL/SQL, supports the concept of a package that can encapsulate variables as well as stored procedures. It does not however support returning a result set. The work around is to return a cursor variable, which can operate, on a result set. Sybase stored procedures can return one or more result sets. Both vendors provide ways to execute dynamically created SQL statements within stored procedures.

## **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.

### ***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. Indexing the temporary tables or eliminating their use can generally resolve these issues.

## ***Recursive Data Structures***

I've included this section under potential problems not because there is implicitly something wrong with recursive data structures, 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, Oracle, have syntactical extensions to their SQL to support recursive queries to be expressed very simply. However, they don't provide significant performance improvements.

On at least one occasion it was easier for me to just extract all the tree data from the database and build the tree within a 3 GL. 3 GL's provide support for pointers to connect tree nodes directly and to navigate them at very low cost.

## ***Cache Tables***

If you find a cache table within the data model this is a good indicator that something might be wrong. This is a indicator of gross denormalization in that the same data is actually being stored in two places. 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 could simplify this entire design.

## ***Use of Functions in the Where Clause***

The purpose of functions provided by the server is to be used, but care must 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.

## **Advanced 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 small 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 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 a data warehouse. Every day a batch of data was prepared to be loaded into the data warehouse. Rather than the loader 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 acceptably efficient if all the data was ready to load, it was grossly inefficient if the amount of data to be loaded was significantly less than the full set.

### ***Increase Efficiency Under Load***

As the system becomes loaded we want the system to become more efficient. This will allow it to scale well. One example of this is 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. This is typically difficult to do in practice, but can yield very significant benefits.

### ***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. While the ideal is to strive for mapping transactions within the application into a single 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 update counts, 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.

### ***Glossary***

Driving Table – The table where the database starts the query. The first table that rows are selected from. In a well optimized query this table is the only table we could be using a table scan on, though indexed access is always preferable if retrieving less than 20% of the rows in the table.