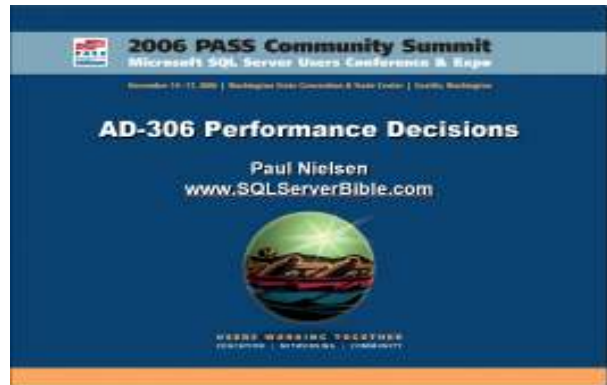


Optimization Theory

Paul Nielsen

Presented at PASS Global Summit 2006
Seattle, Washington

The database industry is intensely driven toward performance with numerous performance techniques and strategies. Prioritizing these techniques is confused by the different points of view from different database roles (DBA, data modeler, developer) and the common consultant's answer of "It depends."



Optimization Theory is a framework for performance based on the idea that there are inherent dependencies between various performance techniques. Basing an optimization strategy on these dependencies will maximize the database optimization.

The Database System

There are four technologies or components in the database system that affect database performance:



- **Server:** the physical hardware configuration (CPUs, memory, disk spindles, i/o buss), the operating system, and the SQL Server installation and configuration.
- **Database:** the physical schema, T-SQL code, and indexes within the database.
- **Maintenance:** the daily database maintenance jobs that keep the database running smooth such as index defragmentation, DBCC integrity checks, and maintaining index statistics.
- **Data Access:** the data tier or front-end application code that accesses the database.

All four technology components must function well to produce a well-performing database system; if one of the components is weak then the database system will fail or perform poorly. However, the database itself is the most difficult component to design and the one that drives the design of the other three components. For example, a well-designed database can significantly reduce the hardware requirements. Maintenance jobs

and data access code are both designed around the database and an overly complex database will complicate both the maintenance jobs and the data access code.



Because the database is the driving component of performance, the goal of optimization theory is to reduce the aggregate workload by minimizing the work of every individual transaction within the database:

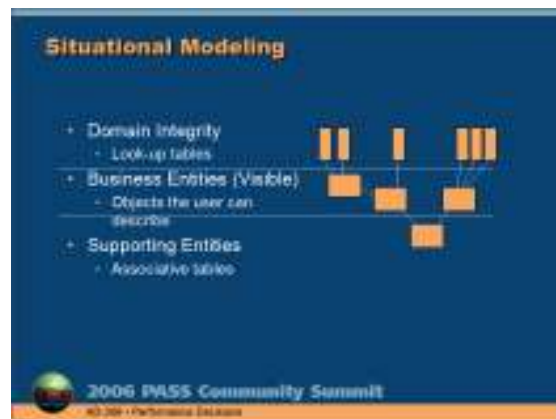
Optimization Theory: the physical schema enables set-based code, which enables indexing. The physical schema, set-based code and indexing together create short transactions which improve concurrency and scalability.

Physical Schema

The base layer of optimization theory is the database physical schema; it's the schema that enables every other area of optimization.

To design performance into the physical schema:

- Fear over-complexity. Work the data model through several iterations to drive down complexity.
- Always follow the Rules of One – one type to one table, one item to one PK (row), one attribute to one column. Realizing there are multiple possible correct ways to tell a story, consider normalization as the grammar not a waterfall process with a single correct result. Generalize entities to reduce complexity, but avoid both no generalization and hyper-generalization.
- Use *Situational Modeling* to design the physical primary and foreign keys according to the entity type. Use natural keys for domain integrity to reduce joins. Use single column surrogate keys for visible and supporting entities for performance.
- Design positive-logic structure rather than negative-logic, so queries can look for the presence of a row rather than use the “where not in (subquery)” code.
- Handle optional data (nulls) efficiently. Use nulls for consistency rather than surrogate nulls or missing one-to-one rows.
- De-normalize responsibly.



There's more to a database than performance. Future extensibility, or flexibility, is dependent on encapsulating the database using a data abstraction layer. Without a loose-coupling between the database and data access code the database risks becoming brittle.

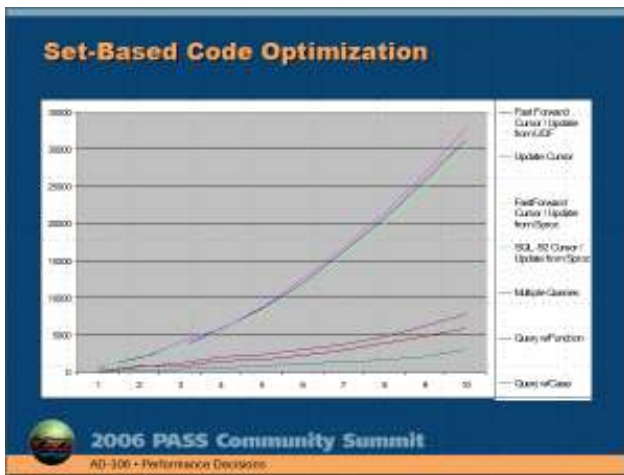
If the physical schema is well designed it will lend itself to easily writing set-based queries. However, a poorly designed database schema will encourage developers to write iterative code, or code that uses temporary buckets to manipulate data.

Set-Based Queries

Database engines are designed to handle data in sets. SQL is a declarative language meaning that the SQL query describes the problem and the query optimizer generates an execution plan to resolve the problem as a set.

Application programmers typically develop while-loops that handle data one row at a time. Iterative code is fine for application tasks such as populating a grid or combo box, but is inappropriate for a server-side code. Iterative T-SQL code, such as cursors, forces the database engine to perform multiple single row sets, instead of handling the problem in one larger, more efficient set. The performance cost is huge. Depending on the task, SQL cursors perform about half as well as set-based code and the performance difference expands with the size of the data. This is why set-based queries are so critical to database performance. Table n lists best practices for solving various types of code operations.

To Solve:	Best Practice:	Avoid Using:
Complex B-Logic	Multiple Queries, Subqueries & CTEs, User Defined Functions	Cursors
Denormalizing a list	Cursor, Multiple Assignment Variable	
Cross-tab Query	Case Expression, Pivot	Cursors, Correlated Queries
Navigating an adjacency list hierarchy	User Defined Function	Cursors, Recursive CTE
Cumulative Totals (running sum)	Cursor	Correlated Subquery
Iterative DDL Code	Cursor	



The worst offender by far is the server-side cursor. Cursors are evil. Many have heard this and might believe it, but tend to hedge the bet and admit that sometimes a problem is so complex a cursor is the only solution. I disagree. To kill the cursor written to solve complex business logic use:

- Use UDF to embed logic within Set-based Query
- Use Case Expressions for flexible logic
- Use data-driven database design
- Break complex task into multiple queries

Because set-based queries are implemented by query execution plans, query plan compilation and reuse affect performance.

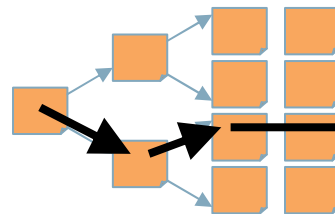
The combination of a clean schema, well-written set-based code, and an abstraction layer enables efficient indexing. It's much easier to create a good indexing plan when these are in place. But, indexes will not overcome the performance problems of iterative code and it's extremely difficult to plan indexes for an overly complex schema.

Indexing

A book's index is a shortcut between a question and the answer, which assumes both a known question and a page with the answer. Database indexes serve the same purpose.

Critical SQL Server indexing techniques include:

- Assigning the clustered index
- Designing multiple purpose indexes
- Eliminating redundant indexes
- Exploiting covering indexes
-



The data is the clustered index – the phone book style index of the data that serves two purposes. First, once the data is located using the b-tree index, all the columns are immediately available without any further work. Secondly, clustering physically gathers similar rows to a minimum of pages to reduce reads.

Non-clustered indexes are functionally the same as the index in the back of a book. They can help identify the data page (non-clustered index seek), but a bookmark lookup is needed to access data page columns.

If a non-clustered index can satisfy the query then SQL Server will use it to cover the needs of the query and not bother accessing the clustered index pages. Covering indexes are fantastic.

Following Optimization Theory, schema, queries, and indexes together reduce the transaction duration and set-up the database for excellent concurrency.

But as great as indexes are, they can't overcome the performance difficulties of iterative code. Ad-Hoc SQL code, and poorly written SQL code that returns unnecessary columns is much more difficult to index and will likely not take advantage of covering indexes. And, it's nearly impossible to properly index an overly complex schema.

Transactions

SQL Server, as an ACID compliant database engine, supports isolated transactions, whether the transaction is a single statement, or a logical transaction. At this point in Optimization Theory, the database should be running well. Three techniques that can be used to further tune transactions are:

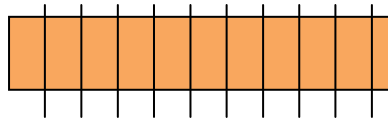
- Tighten logical transactions
- Use the right isolation level
- Snapshot Isolation

A clean schema, set-based queries, good indexes, and efficient transactions are the foundation of a scalable application. Conversely, if the schema, queries and indexes are poorly designed, no amount of transactional tuning will provide the performance level desired.

Advanced Scalability

Beyond typical optimization, SQL Server offers three key scalability features;

- Table Partitioning
- Indexed views
- Service broker



When applied to a database design according to Optimization Theory, table partitioning, indexing views, and service broker will perform, but expecting these scalability features to solve the performance issues caused by a poor design schema, a lack of set-based code, or inadequate indexes or long transactions will fail.

Optimization Theory was developed by Paul Nielsen and has been documented in MOC 2784 – Optimizing Queries using SQL Server 2005; SQL Server 2005 Performance TechNet Jan 2005; and SQL Server 2005 Bible, Wiley 2006

Paul Nielsen is a SQL Server MVP, author of the SQL Server Bible series (Wiley), Presenter for Total Training's SQL Server videos, Director of Global Community Development PASS. He developed the Nordic object/relational database schema for SQL Server. He regularly leads seminars on SQL Server Design and Optimization. His web site is www.SQLServerBible.com