CHAPTER 1

The Information Architecture Principle

For any complex endeavor, there is value in beginning with a common principle to drive designs, procedures, and decisions. A credible principle is understandable, robust, complete, consistent, and stable. When an overarching principle is agreed upon, conflicting opinions can be objectively measured, and standards can be decided upon that support the principle.

The following principle encompasses the three main areas of information management: database design and development, enterprise data center management, and business intelligence analysis.

Information Architecture Principle: Information is an organizational asset, and, according to its value and scope, must be organized, inventoried, secured, and made readily available in a usable format for daily operations and analysis by individuals, groups, and processes, both today and in the future.

Unpacking this principle reveals several practical implications. First, there should be a known inventory of information, including its location, source, sensitivity, present and future value, and current owner. While most organizational information is stored in IT databases, uninventoried critical data is often found in desktop databases and spreadsheets scattered throughout the organization.

Just as the value of physical assets varies from asset to asset and over time, the value of information is also variable and so must be assessed. The value of the information may be high for an individual or department, but less valuable to the organization as a whole. Information that is critical today might be meaningless in a month, or information that may seem insignificant individually might become critical for organizational planning once aggregated.

If the data is to be made easily available in the future, current designs must be decoupled to avoid locking the data in a rigid, but brittle, database.

Based on the Information Architecture Principle, every data store can be designed or evaluated by seven interdependent data store objectives: simplicity, usability, data integrity, performance, availability, extensibility, and security.

This chapter sets a principle-based foundation for the book, which provides a solid reason for each of the following chapters. However, the principle addresses some advanced database concepts, so if you're new to databases I recommend that you dive right in to Chapter 3, "Exploring SQL Server Architecture," and come back to the database concepts chapters later.

Simplicity vs. Complexity

Underscoring the Information Architecture Principle and the other six data store objectives is a design methodology I jokingly call the *Mortgage-Driven Simplicity Method*. While corporate IT programmers might eventually be terminated if their projects repeatedly fail, the consequences for an independent consultant are much more immediate, and the mortgage must be paid.

As a database consultant, I tend to take on the riskiest database projects — those considered disasters by the client (who had fired the previous consulting firm, or firms, for failure to deliver). In every case, the client originally asks me to finish the last 10 percent, or to optimize a project that didn't quite meet the requirements. What I usually find is a hopeless design.

The primary lesson I've learned from successfully turning around dozens of failed database projects is that the number one reason why software projects fail is not incompetent programmers, but an overly complex design. They fail because the designers were unable, or unwilling, to imagine an elegant solution. I've learned a few lessons from having to pay my mortgage based solely on my ability to deliver databases that performed.

The Mortgage-Driven Simplicity Method states:

> Fear complexity. Continue to collaboratively iterate the design until the design team unanimously agrees that it's the simplest solution possible that meets the requirements.

Complexity

Complexity breeds complexity, and complexity causes several problems. The most common result of complexity is that the project will outright fail. While a complex design may appear to be clever, it seldom meets the requirements. Complexity also makes the solution more difficult for developers to understand and implement, resulting in late projects.

Assuming the project is completed, complexity negatively affects every one of the other six data store objectives (usability, data integrity, performance, availability, extensibility, security). A complex design makes it more difficult to retrieve and update the correct data, which affects both usability and data integrity. Additional components create extra work (additional reads, joins, extra updates) and adds interdependent variables, making tuning more complex, all of which reduce performance.

The complex design potentially creates additional points of failure; and when problems do occur, the complexity obscures the source of the problem, making it difficult or impossible to diagnose and fix, which drives up support costs and limits availability. Compared to a simpler design, complex designs are extremely difficult to modify and adapt to changing requirements. Finally, a complex set of components makes it less likely that data will be correctly secured.

There's a reason why the thesaurus entry for the word "difficult" includes the word "complex."

Simplicity

A simple solution is elegant and appears obvious, but saying that designing a simple solution is easy is like watching the Olympics — masters always make their skill look easy. To quote Dr. Einstein,

> *"Things should be made as simple as possible — but no simpler."*

The "as simple as possible" solution isn't a simpleton answer. The simplest solution that satisfies the requirements may have a level of complexity, but of all the possible designs, it's the simplest. Simplicity is measured as the number of components, technical variables, internal interfaces, and technologies involved in delivering the solution.

Designing a simple solution, in any discipline, is a difficult task that requires every one of the following ingredients:

- ✦ A complete understanding of the requirements
- ✦ A broad repertoire of patterns and solutions to draw from
- ✦ A complete understanding of the technical rules and nuances of the profession
- ✦ A creative mastery of the profession that knows when and how to push the envelope
- ✦ A solid understanding of the physical tools and materials used to implement the design and the environment within which the solution must function
- ✦ Enough trust between the designers that ideas can morph and grow based solely on their merits without personal egos or ownership
- ✦ The commitment to continue iterating the design for as long as it takes to design an elegant, slam-dunk, simple solution
- ✦ A healthy fear of complexity

When developing a design, if a few extra hours, or days, of collaboration results in an idea that eliminates a table or process while still meeting the requirements, that's a cause for celebration. Every reasonable reduction in complexity decreases the amount of implementation work and increases the chance of delivering a working product. Every extra dollar spent on design will bring a handsome return during implementation and even more in support costs over the life of the product

Some scoff at naming simplicity as the number one design objective of an inherently complex task like designing a database. From my experience, however, if the design team "gets" simplicity, then there's a good chance the project will be a success. If they don't value simplicity, I wouldn't want to be on that team.

The Usability Objective

The second data store objective is usability. The usability of a data store involves the completeness of meeting the organization's requirements, the suitability of the design for its intended purpose, the effectiveness of the format of data available to applications, and the ease of extracting information. The most common reason a database is less than usable is an overly complex or inappropriate design.

Suitability of Design

Several possible data store design models or types may be selected depending on the data store's scope and purpose.

The scope of the database involves the granularity of use within the organization (individual, department, business unit, company, enterprise) and the temporal nature of the data (for today only, for this fiscal year, forever). The rigor applied in the design and implementation should be proportional to the scope of the data.

Table 1-1 summarizes the data store design types and their suitability for various data store attributes (configuration and requirements).

Table 1-1: Suitability of Data Store Types by Level of Support

Attribute	*Relational DBMS*	*Object DB*	*O/R DBMS*	*Generic Pattern*	*Data Warehouse*
Suitable for master data store	●	◐	●	◐	○
Suitable for reference data store	◐	◐	◐	○	●
Data retrieval performance	◐	◐	◐	○	●
Supports schema flexibility	◐	●	●	●	○
Ease of SQL Query/traditional reporting tools	◐	○	◐	○	●
Well-established vendor support	●	○	◐	○	●
Requirement include several "is-a" relationships	○	●	●	○	○
Stores complex data types	◐	●	◐	○	◐
Complex multi-relational associations	◐	◐	●	○	◐
Ease of operations and tuning	◐	◐	●	◐	●
Persisting application objects	○	●	●	◐	○
Preventing data update anomaly	●	○	●	◐	○

○ represents poor support, ◐ represents limited support, and ● represents full support.

Data Store Configurations

An enterprise's data store configuration includes multiple types of data stores, as illustrated in Figure 1-1. There are multiple types of data stores: master data stores, caching data stores, reference data stores, data warehouses, and data marts.

A *master data store*, also referred to as the *operational database,* or *Online Transaction Processing (OLTP) database,* is used to collect first-generation transactional data essential to the day-to-day operation of the organization and unique to the organization. Customer, order, and shipping information are stored in a master data store. An organization might have a master data store to serve each unit or function within the organization.

For performance, master data stores are tuned for a balance of data retrieval and updates. Because these databases receive first-generation data, they are subject to data update anomalies, and benefit from normalization (detailed in the next section).

The master data store is part of Bill Gates' digital nervous system brain. It's the portion that collects all the information from every nerve and organizes the information so that it can be processed by the rest of the brain. The master data store is used for quick responses and instant recognition of the surroundings. For example, by quickly solving an order-handling problem, the master data store serves as the autonomic nervous system, or the reflexes, of an organization.

Caching data stores are optional read-only copies of the master data store, and are used to deliver data by partitioning some of the load off the master data store. A master data store might have multiple caching data stores to deliver data throughout the organization. Caching data stores might reside in a middle tier, or behind a web service. Caching data stores are tuned for high-performance data retrieval.

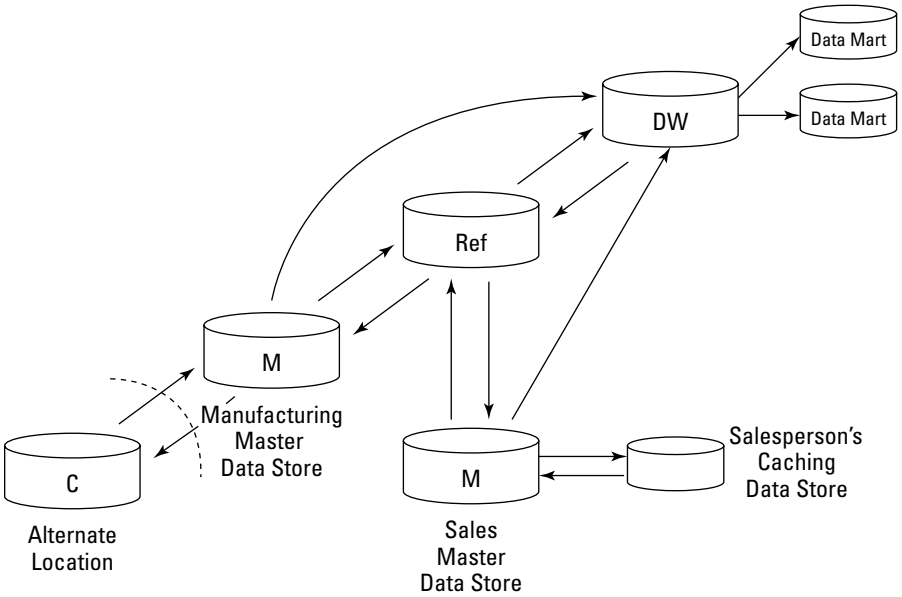

Figure 1-1: A typical organizational data store configuration includes several master data stores feeding a single data warehouse.

Reference data stores are primarily read-only and store generic data that is required by the organization but seldom changes — similar to the reference section of the library. Examples of reference data might be unit of measure conversion factors or ISO country codes. A reference data store is tuned for high-performance data retrieval.

A *data warehouse* collects large amounts of data from multiple master data stores across the entire enterprise using an *Extract-Transform-Load (ETL)* process to convert the data from the various formats and schema into a common format, designed for ease of data retrieval. Data warehouses also serve as the archival location, storing historical data and releasing some of the data load from the operational data stores. The data is also pre-aggregated, making research and reporting easier, thereby reducing errors.

A common data warehouse is essential for ensuring that the entire organization researches the same data set and achieves the same result for the same query — a critical aspect of Sarbanes-Oxley and other regulatory requirements.

Data marts are subsets of the data warehouse with pre-aggregated data organized specifically to serve the needs of one organizational group or one data domain.

The analysis process usually involves more than just SQL queries, and uses data cubes that consolidate gigabytes of data into dynamic pivot tables. Business intelligence (BI) is the combination of the ETL process, the data warehouse data store, and the acts of creating and browsing cubes.

Within Bill Gates' digital nervous system, the data warehouse serves as the memory of the organization. It stores history and is used for data mining such as trend analysis, such as finding out where (and why) an organization is doing well or is failing. The portion of the digital nervous system that is used by an organization for thoughtful musings — slowly turning over a problem and gaining wisdom — is the data warehouse and a BI cube.

Because the primary task of a data warehouse is data retrieval and analysis, the data-integrity concerns present with a master data store don't apply. Data warehouses are designed for fast retrieval and aren't normalized like master data stores. They are designed using a basic star schema or snowflake design. Locks generally don't apply, and the indexing is applied without adversely affecting inserts or updates.

Master Data Stores Design Styles

Database designers are not limited to the relational model. Several database design styles exist from which to choose depending on the requirements of the project.

Relational DBMSs

Relational databases are traditional databases that organize similar or related data into a single table. Relational databases are excellent with stable data schema requirements that include a certain minimum of *is-a* relationships (e.g., a customer is a contact).

Object-Oriented DBMSs

Object-oriented databases align the data structure with object-oriented application design (OOA/D) for the purpose of persisting application objects. OOA/D is based on the concept that an object is an instance of an object class. The class defines the properties and methods of the object, and contains all the code for the object. Each instance of the object can have its own internal variables and can interact with the software world outside the object on its own. Although the terms are different, the basic concept of organizing data is similar, as shown in Table 1-2.

Table 1-2: Comparing Database Terms

Development Style	*The Common List*	*An Item in the List*	*A Piece of Information in the List*
Spreadsheet	Spreadsheet/ worksheet/ named range	Row	Column/cell
Historic information	File	Record	Field
Relational algebra/ logical design	Entity	Tuple (*rhymes with couple*) or Relation	Attribute
SQL/physical design	Table	Row	Column
Object-oriented analysis and design	Class	Object instance	Property

Because the OO DBMSs must store objects, a key criterion for an OO DBMS is that it must be able to store complex objects, such as XML files, or .NET classes.

OO DBMSs are suitable for applications that expect significant schema change, include complex data types, involve several is-a relationships between classes, include complex multi-associations, and require ease of data connectivity with application software.

There are three primary types of object-oriented databases:

- An *object-persistence data store (OP DBMS)* is designed to be little more than a repository for object states. All integrity issues are handled by the object-oriented application code, and the database schema is designed to map perfectly with the application's class diagram.
- An *object-oriented data store (OO DBMS)* persists application objects and uses metadata (data that describes the way data is organized) to model the object-oriented class structure and enforce object-class integrity.
- An *object/relational data store (O/R DBMS)* persists the application objects, and models the class structure within a relational DBMS. O/R DBMSs provide the benefits of OO A/D with the traditional query and reporting ability of relational databases.

For more information about Nordic (New Object Relational Database Design), by the author, visit `www.SQLServerBible.com`.

Generic Pattern DBMS

The *generic pattern* database, sometimes called the *dynamic-diamond pattern,* illustrated in Figure 1-2, is sometimes used as an OO DBMS physical design within a RDBMS product. This design can be useful when applications require dynamic attributes. A manufacturing material-specifications system, for example, would require different attributes for nearly every material type. To further complicate matters, the attributes that are tracked frequently change depending on the Total Quality Management (TQM) or ISO 9000 process within the company. A purely relational database might use an entity for each material type, requiring constant schema changes to keep current with the material tracking requirements.

Enterprise Architecture

The term *software architecture* takes on varied meanings depending on who's using the term, and their view, or scope, of the world:

- ✦ When Microsoft refers to architecture they usually mean their .NET Framework or designing any system using multiple Microsoft products together to solve a problem.
- ✦ *Product architecture* means designing a single application that solves a problem using multiple components.
- ✦ *Infrastructure architecture* is the design of the network using routers, switches, firewalls, SANs, and servers.
- ✦ *Data architecture* is the organization and configuration of data stores.
- ✦ *Enterprise architecture* attempts to manage all of these architectures for an entire organization, and is commonly divided into three areas: infrastructure, application, and data.

How enterprise architecture is accomplished is a subject of much debate. To use the construction industry metaphor, the views on enterprise architecture range from a passive building code committee approving or rejecting plans based on the standards to actively planning the city. While every viewpoint begins with a set of architectural principles, they differ on how those principles are enforced and how much initiative the architecture team used when planning the principles.

The *building code* viewpoint backs away from recommending to a client (those within the organization with the authority to make purchasing decisions) which software or applications they might use. Instead, the architecture team approves or denies client proposals based on the proposal's compliance with the architectural principles and standards. If the proposal doesn't meet the standards, then the client is free to request a variance using a defined procedure (change request). Which applications are built, how they are designed, and who builds them is completely up to the client, not the software architect.

The *zoning board* viewpoint takes the building code view and adds the notion of planning the enterprise's application portfolio. For example, a zoning board style enterprise architect might determine that the organization needs a certain type of application and require the application to be run on the organization's infrastructure, but would leave the actual design or purchase of the application to the clients.

The *city planning* viewpoint is the most aggressive and takes on the role of technical leadership for the organization. As a city planner, the enterprise architect will proactively attempt to forecast the needs of the organization and work with clients to determine a future enterprise portfolio and overall architecture. The architect then works to move the organization toward the future plan by designing or acquiring software, refactoring existing software, and decommissioning software that doesn't conform to the plan. The success factors for this viewpoint are two-fold: The future plan is designed jointly by the clients and the architect; and the future plan is reviewed and updated as the organization and technologies evolve.

The issue of governance is a potential problem. If the architect who originated the principles and standards becomes the enforcer, there is a conflict of interest (lawmaker, law enforcer, judge, and jury). It also places the IT architects in the awkward position of software police, which can lead to division, conflict, and resentment between the IT organization and the enterprise it is trying to serve. The solution is to align the authority with the accountability by moving the architectural principles from the IT architecture office to the executive board. The IT architects recommend principles to the executive board. If the board understands the benefit to the organization and adopts the principles—and the procedures for exemptions—as their own, then IT and the entire organization must all adhere to the principles.

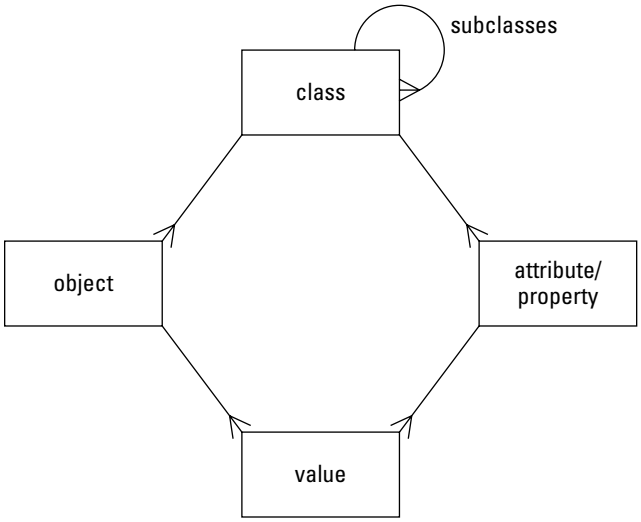

Figure 1-2: The generic pattern is an RDBMS physical schema that is sometimes employed to mimic an OO DBMS.

The `class` entity drives the database schema. The `class` entity includes a reflexive relationship to support object class inheritance. As with a hierarchical structure or organizational chart, this relationship permits each object class to have multiple subclasses, and each object class to have one base class. The `property` entity is a secondary entity to the `object` entity and enables each object class to contain multiple properties. An object is a specific instance of an object class. As such, it needs to have its own specific values for every property of its object class and all inherited object classes.

Although the result can be impressive, many complications are involved in this process. Many-to-many relationships, which exist in real life, are simulated within object-oriented databases by means of object collections. Properties must meet data-type and validation rules, which must be simulated by the data schema, rather than by SQL Server's built-in data-type and validation rules.

Data Integrity

The ability to ensure that persisted data can be retrieved without error is central to the Information Architecture Principle, and the first problem tackled by the database world. Without data integrity, a query's answer cannot be guaranteed to be correct, consequently, there's not much point in availability or performance.

As data is essentially entities and attributes, data integrity consists of entity integrity and domain integrity, which includes referential integrity and user-defined integrity. Transactional integrity, which deals with how data is written and retrieved, is defined by the ACID principles (atomicity, consistency, isolation, and durability), discussed in a later section, transactional faults, and isolation levels.

Entity Integrity

Entity integrity involves the structure (primary key and its attributes) of the entity. If the primary key is unique and all attributes are scalar and fully dependent on the primary key, then the integrity of the entity is good. In the physical schema, the table's primary key enforces entity integrity. Essentially, entity integrity is normalization.

Normalization is explained in detail in the next chapter, "Relational Database Modeling."

Domain Integrity

In relational theory terms, a domain is a set of possible values for an attribute, such as integers, bit values, or characters. *Domain integrity* ensures that only valid data is permitted in the attribute. Nullability (whether a null value is valid for an attribute) is also a part of domain integrity. In the physical schema, the data type and nullability of the row enforce domain integrity.

Referential Integrity

A subset of domain integrity, *referential integrity* refers to the domain integrity of foreign keys. Domain integrity says that if an attribute has a value, then that value must be in the domain. In the case of the foreign key, the domain is the list of values in the related primary key.

Referential integrity, therefore, is not an issue of the integrity of the primary key but of the foreign key.

The nullability of the column is a separate issue from referential integrity. It's perfectly acceptable for a foreign key column to allow nulls.

Several methods of enforcing referential integrity at the physical-schema level exist. Within a physical schema, a foreign key can be enforced by declarative referential integrity (DRI) or by a custom trigger attached to the table.

User-Defined Integrity

Besides the relational theory integrity concerns, the user-integrity requirements must also be enforced, as follows:

- ✦ Simple business rules, such as a restriction to a domain, limit the list of valid data entries. Check constraints are commonly used to enforce these rules in the physical schema.
- ✦ Complex business rules limit the list of valid data based on some condition. For example, certain tours may require a medical waiver. Implementing these rules in the physical schema generally requires stored procedures or triggers.

Some data-integrity concerns can't be checked by constraints or triggers. Invalid, incomplete, or questionable data may pass all the standard data-integrity checks. For example, an order without any order detail rows is not a valid order, but no automatic method traps such an order. SQL queries can locate incomplete orders and help in identifying other less measurable data-integrity issues, including the following:

- ✦ Wrong data
- ✦ Incomplete data
- ✦ Questionable data
- ✦ Inconsistent data

The quality of the data depends upon the people modifying the data. Data security — controlling who can view or modify the data — is also an aspect of data integrity.

Transactional Integrity

A transaction is a single logical unit of work, such as inserting 100 rows, updating 1,000 rows, or executing a logical set of updates. The quality of a database product is measured by its transactions' adherence to the ACID properties. ACID, you might recall, is an acronym for four interdependent properties: atomicity, consistency, isolation, and durability. Much of the architecture of SQL Server is founded on these properties. Understanding the ACID properties of a transaction is a prerequisite for understanding SQL Server:

- ✦ The transaction must be *atomic*, meaning all or nothing. At the end of the transaction either all of the transaction is successful or all of the transaction fails. If a partial transaction is written to disk, the atomic property is violated.
- ✦ The transaction must preserve database *consistency*, which means that the database must begin in a state of consistency and return to a state of consistency once the transaction is complete. For the purposes of ACID, consistency means that every row and value must agree with the reality being modeled, and every constraint must be enforced. If the order rows are written to disk but the order detail rows are not written, the consistency between the `Order` and the `OrderDetail` is violated.
- ✦ Each transaction must be *isolated,* or separated, from the effects of other transactions. Regardless of what any other transaction is doing, a transaction must be able to continue with the exact same data sets it started with. Isolation is the fence between two transactions. A proof of isolation is the ability to replay a serialized set of transactions on the same original set of data and always receive the same result.

  For example, assume Joe is updating 100 rows. While Joe's transaction is under way, Sue deletes one of the rows Joe is working on. If the delete takes place, Joe's transaction is not sufficiently isolated from Sue's transaction. This property is less critical in a single-user database than in a multi-user database.
- ✦ The *durability* of a transaction refers to its permanence regardless of system failure. Once a transaction is committed it stays committed. The database product must be constructed so that even if the data drive melts, the database can be restored up to the last transaction that was committed a split second before the hard drive died.

The nemesis of transactional integrity is concurrency — multiple users simultaneously attempting to retrieve and modify data. Isolation is less of an issue in small databases, but in a production database with thousands of users, concurrency competes with transactional integrity. The two must be carefully balanced; otherwise, either data integrity or performance will suffer.

SQL Server's architecture meets all the transactional-integrity ACID properties, providing that you, as the developer, understand them, develop the database to take advantage of SQL Server's capabilities, and the DBA implements a sound recovery plan. A synergy exists among

SQL Server, the hardware, the database design, the code, the database-recovery plan, and the database-maintenance plan. When the database developer and DBA cooperate to properly implement all these components, the database performs well and transactional integrity is high.

Transactional Faults

True isolation means that one transaction never affects another transaction. If the isolation is complete, then no data changes from outside the transaction should be seen by the transaction.

The isolation between transactions can be less than perfect in one of three ways: dirty reads, non-repeatable reads, and phantom rows. In addition, transactions can fail due to lost updates and deadlocks.

Dirty Reads

The most egregious fault is a transaction's work being visible to other transactions before the transaction even commits its changes. When a transaction can read another transaction's uncommitted updates, this is called a *dirty read*. The problem with dirty reads is that the data being read is not yet committed, so the transaction writing the data might be rolled back.

Non-Repeatable Reads

A *non-repeatable read* is similar to a dirty read, but a non-repeatable read occurs when a transaction can see the committed updates from another transaction. Reading a row inside a transaction should produce the same results every time. If reading a row twice results in different values, that's a non-repeatable read type of transaction fault.

Phantom Rows

The least severe transactional-integrity fault is a *phantom row*. Like a non-repeatable read, a phantom row occurs when updates from another transaction affect not only the result set's data values, but cause the `select` to return a different set of rows.

Of these transactional faults, dirty reads are the most dangerous, while non-repeatable reads are less so, and phantom rows are the least dangerous of all.

Lost Updates

A lost update occurs when two users edit the same row, complete their edits, and save the data, and the second user's update overwrites the first user's update.

Because lost updates occur only when two users edit the same row at the same time, the problem might not occur for months. Nonetheless, it's a flaw in the transactional integrity of the database that needs to be prevented.

Deadlocks

A deadlock is a special situation that occurs only when transactions with multiple tasks compete for the same data resource. For example, consider the following scenario:

- Transaction one has a lock on data A and needs to lock data B to complete its transaction.

  and

- Transaction two has a lock on data B and needs to lock data A to complete its transaction.

Each transaction is stuck waiting for the other to release its lock, and neither can complete until the other does. Unless an outside force intercedes, or one of the transactions gives up and quits, this situation could persist until the end of time.

Chapter 51, "Managing Transactions, Locking, and Blocking," includes walk-through examples of the transactional faults, isolation levels, and SQL Server locking.

Isolation Levels

At the physical level, any database engine that permits logical transactions must provide a way to isolate those transactions. The level of isolation, or the height of the fence between transactions, can be adjusted to control which transactional faults are permitted. The ANSI SQL-92 committee specifies four isolation levels: read uncommitted, read committed, repeatable read, and serializable.

In addition, Microsoft added snapshot isolation to SQL Server 2005. Essentially, snapshot isolation makes a virtual copy, or snapshot, of the first transaction's data, so other transactions do not affect it. This method can lead to lost updates.

The Value of Null

The relational database model represents missing data using the special value of *null*. The common definition of null is "unknown"; however, null can actually represent three subtly different scenarios of missing data:

- ✦ The column does not apply for this row — for example, if the person is not employed, then any value in the `EmploymentDate` column would be invalid.
- ✦ The data has not yet been entered, but likely will, such as in a contact row that has the name and phone number, and will hopefully have the address once a sale is made.
- ✦ The column for this row contains a value of "nothing" — for example, a comment column is valid for every row but may be empty for most rows.

Depending on the type of missing data, some designers use surrogate nulls (blanks, zeroes, or n/a) instead. However, multiple possible values for missing data can create consistency problems when querying data.

The nullability of a column, whether or not the column accepts nulls, may be defined when a column is created. Note that by default, SQL Server does not allow nulls, but the ANSI standard does.

Working with Nulls

Because null has no value, the result of any expression that includes null will also be unknown. If the contents of a bank account are unknown and its funds are included in a portfolio, the total value of the portfolio is also unknown. The same concept is true in SQL, as the following code demonstrates. Phil Senn, a wise old database developer, puts it this way: "Nulls zap the life out of any other value." As proof:

```
SELECT 1 + NULL
```

Result:

```
NULL
```

Another consequence of null's unknown value is that a null is not equal to another null. Therefore, to test for the presence of null, SQL uses the `IS NULL` syntax.

Both of these behaviors can be overridden. SQL Server will ignore nulls in expression when the connection setting `concat_null_yields_null` is set to off. The `ANSI nulls` setting controls whether null can be equal to another null.

Null Controversy

Most database developers design columns that allow nulls when they make sense.

Extreme database purists detest nulls and require that any relational database model not allow nulls. One method of avoiding nulls is to separate any null columns into a separate supertype/subtype table. This method merely replaces a nullable column with a nullable row, which requires a left-outer join to test for the presence of data or retrieve data. The resulting complexity affects not only performance but also data integrity. Instead of retrieving data with a null column, the application developer must be aware of the subtype table and be fluent with left-outer joins. This view focuses on a misguided understanding of a single data store objective, data integrity, at the expense of performance and usability.

Performance

Presenting readily usable information is a key aspect of the Information Architecture Principle. Although the database industry has achieved a high degree of performance, the ability to scale that performance to very large databases with more connections is still an area of competition between database engine vendors.

Because physical disk performance is the most significant bottleneck, the key to performance is reducing the number of physical page reads or writes required to perform a task. The five primary performance factors all seek to reduce the number of physical page reads.

Design

The database schema design can dramatically affect performance. The physical design must consider the query path. An overly complicated design, resulting in too many tables, requires additional joins when retrieving data, and additional writes when inserting or updating data.

Some database schemas discourage set-based processing by requiring that data move from one bucket to another as it's processed. If server-side application code includes several cursors, maybe it's not the fault of the application developer but the database designer.

Set-Based Processing

Relational algebra, the SQL language, and relational database engines are all optimized to work with sets of data. Poorly written code, whether row-by-row cursor-based code, or just poorly written SQL, is a common source of poor performance.

Writing excellent set-based queries requires a creative understanding of joins and subqueries, as discussed in Chapter 9, "Merging Data with Joins and Unions," and Chapter 10, "Including Subqueries and CTEs."

A well-written set-based query can perform the entire operation, reading each required page only once, whereas a cursor-based solution will process each row independently. In tests, a simple update operation using cursors takes 70 times as long as the same logic implemented using set-based code.

Indexing

Indexes are the bridge between the query and data. They improve performance by reducing the number of physical page reads required for a table read operation.

- *Clustered indexes* group rows together so they can be retrieved in one (or a few) physical page reads, rather than reading from numerous rows scattered throughout the table.
- Indexes enable the query optimizer to seek directly at the data rows, similar to how a book index can be used to find the correct page, instead of having to scan the entire table to locate the correct rows. Once the row is determined, the optimizer will perform a bookmark lookup to jump to the data page.

Indexing is a key part of the physical schema design and is considered more of an art than a science, but understanding the database engine's query optimizer and how the index structures work, combined with a knowledge of the database's specific schema and how the queries will be accessing the data, can make index design more exact.

Index tuning strategies are discussed in Chapter 50, "Query Analysis and Index Tuning."

Indexes have a downside as well. Although indexes help when reading from the table, they can adversely affect write performance. When a row is inserted or updated, the indexes must be also be kept in synch with the data. Therefore, when a table has multiple indexes, writes to the table will be slower. In other words, there's a tension between designing indexes for reading versus writing. Because an update or delete operation must locate the affected rows, write operations do benefit from frugal indexing. The different indexing requirements of reading versus writing is a major difference between transactional databases and databases designed for reporting or data warehousing.

Partitioning

Partitioning, or spreading the data across multiple disk spindles, is a method of improving the performance of very large databases (VLDs).

Chapter 53, "Scaling Very Large Databases," details SQL Server 2005's partitioning features.

Caching

Caching is the means of pre-fetching data from the physical hard drive so the data is in memory when required by a database operation. While caching is the job of the database engine, providing it with enough memory makes a difference.

Availability

The availability of information refers to the information's accessibility when required regarding uptime time, locations, and the availability of the data for future analysis. Recovery, redundancy, archiving, and network delivery all affect availability.

The system requirements drives availability, which is often described in terms of 9s. Five 9s means the system is available 99.999 percent of the required time, as shown in Table 1-3.

Table 1-3: The Chart of Nines

Percent	*Uptime*	*Downtime*	*Description*
99.99999	364d 23h 59m 56s	000d 00h 00m 03s	"7 nines"
99.9999	364d 23h 59m 29s	000d 00h 00m 31s	"6 nines"
99.999	364d 23h 54m 45s	000d 00h 05m 15s	"5 nines"
99.99	364d 23h 07m 27s	000d 00h 52m 33s	"4 nines"
99.95	364d 19h 37m 12s	000d 04h 22m 48s	-
99.9	364d 15h 14m 24s	000d 08h 45m 36s	"3 nines"
99.8	364d 15h 14m 24s	000d 17h 31m 12s	-
99.72603	364d 00h 00m 00s	001d 00h 00m 00s	exactly 1 day
99.5	363d 04h 12m 00s	001d 19h 48m 00s	-
99	361d 08h 24m 00s	003d 15h 36m 00s	"2 nines"
98	357d 16h 48m 00s	007d 07h 12m 00s	-
97	354d 01h 12m 00s	010d 22h 48m 00s	-

Chapter 36, "Recovery Planning," and Chapter 52, "Providing High Availability," both provide details on SQL Server 2005's availability features.

Redundancy

Redundancy is the identification of possible points of failure and avoiding or reducing the effects of the failure by providing a secondary solution. For some disciplines, redundancy suggests waste, but for data stores, redundancy is a good thing.

A *warm standby server* is a copy of the database on another server ready to go live at a moment's notice. Typically, this method uses log shipping to move the latest transaction to the warm standby server. A *clustered server* is an expensive hardware link between two servers such that when the primary server goes down, the backup server is immediately online.

At some point, the best hardware will fail. If the information is extremely critical to the organization, management may decide that it's too expensive (in lost time to the organization) to

restore the database and the transaction log to another server. A solution is to maintain a warm backup server on ready standby so that if the primary server fails, then the warm standby comes online with minimal availability impact.

Recovery

The availability method of last resort is to restore the database and recover the data from backup files. Recovery requires more than just a backup file. The write-ahead transaction log provides a way to recover all transactions committed since the last database backup.

Extensibility

The information architecture principle states that the information must be readily available today and in the future, which requires that the data store is *extensible*, able to be easily adapted to meet new requirements. As an industry, data integrity, performance, and availability are all mature and well understood, so the next major hurdle for the industry to conquer is extensibility.

Two design concepts lead to an extensible system: decoupling the database using an abstraction layer and generalizing entities when possible. Extensibility is also closely related to simplicity. Complexity breeds complexity, and inhibits adaptation.

Abstraction Layer

Many production databases were well designed when they were created, and served their initial purpose remarkably well, for a while. But as changes are required, the development effort becomes increasingly difficult and developers begin to complain about the database. Years later, the development team may still be stuck with a database that no longer serves the purpose but is too difficult to change. The system met the original requirements, but is now extremely expensive to maintain.

The source of the problem is not the database design, but the lack of encapsulation and coupling between the application and the database. An *abstraction layer* is the primary method of decoupling a data store from the application. Without an abstraction layer logically separating the database from the application, any database is destined to become brittle — the slightest change breaks an overwhelming amount of code.

A *data store abstraction layer* is a collection of software interfaces that serve as a gate through which all access to the database must pass. An abstraction layer is essentially a contract between the application code and the database that defines each database call and the parameters. An abstraction layer can be constructed using T-SQL stored procedures, or a software tier. With an abstraction layer in place and enforced, any direct access to the tables via SQL DML commands is blocked.

Designing and coding a T-SQL–based abstraction layer is explained in Chapter 25, "Creating Extensibility with a Data Abstraction Layer."

An abstraction layer supports extensibility by shielding the application code from schema changes, and vice versa. The implementation of either side of the contract (data store or application code) may be changed without affecting the abstraction layer contract or the other side of the contract.

Generalization

Generalization is the grouping of similar entities into a single entity, or table, so that a single table does more work. The database becomes more flexible because the generalized entities are more likely able to store a new similar entity.

Generalization depends on the idea that there is no single correct normalized schema. For any database problem, several correct solutions are possible. The difference between these solutions is often identifying the problem space's various entities. One view of the entities might identify several specific entities that are technically different but similar. A more generalized view would merge the similar entities, producing a simpler and more compact schema.

I can't overemphasize the importance of generalization. Even more than normalization, this is the technique I depend upon to design simple but flexible schemas. A recent database schema I analyzed used about 80 tables. Using generalization to combine five entities (and associated tables) into one entity, I reduced the schema to 17 tables, and the result was more flexible and powerful than the original. When a database developer brags that he has designed a database with a huge number of tables, I assume that he probably developed a wasteful system.

Security

The final primary database objective based on the Information Architecture Principle is security. For any organizational asset, the level of security must be secured depending on its value and sensitivity. For software, the security begins with the physical security of the data center and the operating system's security. Information security includes three additional components: restricting access to specific data using the database engine's security, identifying the owner of the information, and confirming the veracity of the data by identifying the source, including updates.

Restricted Access

Any production-grade database includes the capability to limit access to the data to individuals or group of individuals. The granularity of restriction may range from simply limiting access to specifying create, select, update, and delete privileges.

SQL Server 2005 security, which can be complex, is detailed in Chapter 40, "Securing Databases."

Information Ownership

The owner of the data is the person or organizational group who required the data. This person is typically not the person or process who entered or created the data. The owner is the person who is paying the cost of maintaining the data. For some systems, a single owner may be identified for the entire application. For other systems, each table or row may have an identified owner. For example, all the data within an inventory system may belong to the vice president of manufacturing, while the corporate client contact information may belong to the branch that is serving the contact. Row-level data ownership must be implemented in the schema by adding a `DataOwner` column.

Compliance with Sarbanes-Oxley not only requires that data is secure, but that any changes in data ownership and access restrictions are documented.

Audit Trails

Audit trails identify the source and time of creation and any updates. At the minimum level, the audit trail captures the created date/time, and the last updated date/time. The next level of auditing captures the username of the person or process who created or updated the data. A complete audit trail records every historical value. The ability to recreate the data at any given point in time is important for temporal, or time-based, data.

Chapter 24, "Exploring Advanced T-SQL Techniques," includes methods of creating audit trails.

Optimization Theory and SQL Server

Ask 20 DBAs for their favorite optimization technique or strategy, and you'll likely hear 40 different answers ranging from indexing to adding memory. Is it possible to order this heap of performance ideas? I believe so. Data modeling is essentially puzzling out the pattern of data. So is there is a pattern to the various strategies that can revealed by examining how one strategy affects another?

The first clue is that not all performance strategies perform well or uniformly because there's an inherent dependency between performance strategies that's easily overlooked. For instance, using indexing to improve the performance of a query reduces the duration of a transaction, which facilitates concurrency. So there's some kind of connection between indexing and concurrency. Maybe there's more. Certain performance strategies enable other strategies while some performance strategies have little effect if other strategies have not already been applied.

Optimization Theory addresses these dependencies and provides a framework for planning and developing an optimized data store. Optimization Theory identifies five key optimization strategies (see Figure 1-3). Each strategy has several specific techniques. Each strategy is enabled by its supporting strategy, and no strategy can overcome deficiencies in their supporting strategies.

Schema Design

Schema Design is my number-one performance strategy. Well-designed schemas enable you to develop set-based queries and make it easier to plan effective indexes.

To design an effective schema, you need to do the following:

- ✦ Avoid overcomplexity.
- ✦ Select a key carefully.
- ✦ Handle optional data.
- ✦ Enforce an abstraction layer.

I believe that the purist logical data modeler is the number-one performance problem in our industry because of the cascading problems caused by his burdensome designs.

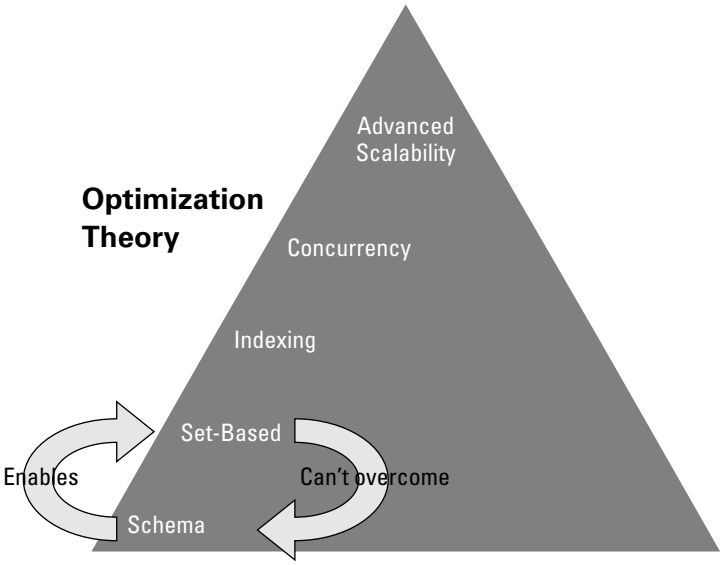

Figure 1-3: Optimization Theory explains that each optimization technique is dependent upon, and enabled by, other optimization techniques.

Queries

SQL is a set-based system, and iterative row-by-row operations actually function as zillions of small single-row sets. Whether the iterations take place as server-side SQL cursors or ADO loops through a record set, iterative code is costly. My number-two performance strategy is to use set-based solutions. But good set-based code can't overcome a clumsy or overly complex schema design.

When deciding where to use iterative code vs. set-based query, use Table 1-4 as a guiding rule of thumb.

Table 1-4: Coding Methods

Problem	*Best Solution*
Complex business logic	Queries, subqueries, CTEs
Dynamic DDL Generation	Cursors
Demoralizing a List	Multiple assignment variables or cursor
Crosstab	Query with pivot or case expression
Navigating a hierarchy	UDF or CTE
Cumulative totals, running sums	Cursor

Chapter 20, "Kill the Cursor!" explains how to create dramatic performance gains by refactoring complex logic cursors into set-based queries.

Indexing

Indexing is the performance bridge between queries and data and a key performance strategy. An indexing strategy that uses a clustered index to reduce bookmark lookups or group rows to a single data page, uses nonclustered indexes to cover queries and enable seeks, and avoids redundant indexes will speed set-based code. But well-designed indexes can't overcome nonscalable iterative code.

Designing clustered, nonclustered, and covering indexes are explained in detail in Chapter 50, "Query Analysis and Index Tuning."

Concurrency

Locking and blocking is more common a problem than most developers think, and too many DBA solve the problem by lowering the transaction isolation level using `nolock` — and that's dangerous.

Concurrency can be compared to a water fountain. If folks are taking long turns at the fountain or filling buckets, a line may form, and those waiting for the resource will become frustrated. Setting `nolock` is like saying, "Share the water." A better solution is to satisfy the needs with only a sip or to reduce the duration or the transaction. The best way to develop efficient transactions is to design efficient schemas, use set-based code, and index well.

When the schema, queries, and indexes are already reducing the transaction duration, be sure to place only the required logic within logical transactions and be careful with logic inside triggers, since they occur within the transaction. But reducing blocked resources won't overcome an unnecessary table scan.

Advanced Scalability

When the schema, queries, indexes, and transactions are all running smooth, you'll get the most out of SQL Server's high-end scalability features:

- ✦ Snapshot Isolation
- ✦ Partition Tables
- ✦ Index Views
- ✦ Service Broker

Will you see performance gains by using a performance technique without the enabling technologies in place? Maybe. But you'll see the greatest gains by enabling each layer with its enabling technologies.

Optimization Theory, the idea that there are dependencies between performance technologies, is an evolving concept. For my latest data on Optimization Theory, or my latest Performance Decision presentation, visit `www.SQLServerBible.com`.

Summary

The Information Architecture Principle is the foundation for database development. The principle unpacks to reveal seven interdependent data store objectives: simplicity, usability, data integrity, performance, availability, extensibility, security. Each objective is important in the design and development of any database.

The first part of this book continues exploring database concepts in the next chapter, which looks at relational database design.

✦ ✦ ✦