

Database Design Considerations

Designing a database requires an understanding of both the business functions you want to model and the database concepts and features used to represent those business functions.

It is important to accurately design a database to model the business because it can be time consuming to change the design of a database significantly once implemented. A well-designed database also performs better.

When designing a database, consider:

- The purpose of the database and how it affects the design. Create a database plan to fit your purpose.
- Database normalization rules that prevent mistakes in the database design.
- Protection of your data integrity.
- Security requirements of the database and user permissions.
- Performance needs of the application. You must ensure that the database design takes advantage of Microsoft® SQL Server™ 2000 features that improve performance. Achieving a balance between the size of the database and the hardware configuration is also important for performance.
- Maintenance.
- Estimating the size of a database.

Creating a Database Plan

The first step in creating a database is creating a plan that serves both as a guide to be used when implementing the database and as a functional specification for the database after it has been implemented. The complexity and detail of a database design is dictated by the complexity and size of the database application as well as the user population.

The nature and complexity of a database application, as well as the process of planning it, can vary greatly. A database can be relatively simple and designed for use by a single person, or it can be large and complex and designed, for example, to handle all the banking transactions for hundreds of thousands of clients. In the first case, the database design may be little more than a few notes on some scratch paper. In the latter case, the design may be a formal document with hundreds of pages that contain every possible detail about the database.

In planning the database, regardless of its size and complexity, use these basic steps:

- Gather information.
- Identify the objects.
- Model the objects.
- Identify the types of information for each object.
- Identify the relationships between objects.

Gathering Information

Before creating a database, you must have a good understanding of the job the database is expected to perform. If the database is to replace a paper-based or manually performed information system, the existing system will give you most of the information you need. It is important to interview everyone involved in the system to find out what they do and what they need from the database. It is also important to identify what they want the new system to do, as well as to identify the problems, limitations, and bottlenecks of any existing system. Collect copies of customer statements, inventory lists, management reports, and any other documents that are part of the existing system, because these will be useful to you in designing the database and the interfaces.

Identifying the Objects

During the process of gathering information, you must identify the key objects or entities that will be managed by the database. The object can be a tangible thing, such as a person or a product, or it can be a more intangible item, such as a business transaction, a department in a company, or a payroll period. There are usually a few primary objects, and after these are identified, the related items become apparent. Each distinct item in your database should have a corresponding table.

The primary object in the **pubs** sample database included with Microsoft® SQL Server™ 2000 is a book. The objects related to books within this company's business are the authors who write the books, the publishers who manufacture the books, the stores which sell them, and the sales transactions performed with the stores. Each of these objects is a table in the database.

Modeling the Objects

As the objects in the system are identified, it is important to record them in a way that represents the system visually. You can use your database model as a reference during implementation of the database.

For this purpose, database developers use tools that range in technical complexity from pencils and scratch paper to word processing or spreadsheet programs, and even to software programs specifically dedicated to the job of data modeling for database designs. Whatever tool you decide to use, it is important that you keep it up-to-date.

SQL Server Enterprise Manager includes visual design tools such as the Database Designer that can be used to design and create objects in the database.

Identifying the Types of Information for Each Object

After the primary objects in the database have been identified as candidates for tables, the next step is to identify the types of information that must be stored for each object. These are the columns in the object's table. The columns in a database table contain a few common types of information:

- Raw data columns

These columns store tangible pieces of information, such as names, determined by a source external to the database.

- Categorical columns

These columns classify or group the data and store a limited selection of data such as true/false, married/single, VP/Director/Group Manager, and so on.

- Identifier columns

These columns provide a mechanism to identify each item stored in the table. These columns often have id or number in their names (for example, **employee_id**, **invoice_number**, and **publisher_id**). The identifier column is the primary component used by both users and internal

database processing for gaining access to a row of data in the table. Sometimes the object has a tangible form of ID used in the table (for example, a social security number), but in most situations you can define the table so that a reliable, artificial ID can be created for the row.

- Relational or referential columns

These columns establish a link between information in one table and related information in another table. For example, a table that tracks sales transactions will commonly have a link to the **customers** table so that the complete customer information can be associated with the sales transaction.

Identifying the Relationships Between Objects

One of the strengths of a relational database is the ability to relate or associate information about various items in the database. Isolated types of information can be stored separately, but the database engine can combine data when necessary. Identifying the relationships between objects in the design process requires looking at the tables, determining how they are logically related, and adding relational columns that establish a link from one table to another.

For example, the designer of the **pubs** database has created tables for titles and publishers in the database. The **titles** table contains information for each book: an identifier column named **title_id**; raw data columns for the title, the price of the book, and the publishing date; and some columns with sales information for the book. The table contains a categorical column named **type**, which allows the books to be grouped by the type of content in the book. Each book also has a publisher, but the publisher information is in another table; therefore, the **titles** table has a **pub_id** column to store just the ID of the publisher. When a row of data is added for a book, the publisher ID is stored with the rest of the book information.

Online Transaction Processing vs. Decision Support

Many applications fall into two main categories of database applications:

- Online transaction processing (OLTP)
- Decision support

The characteristics of these application types have a dramatic effect on the design considerations for a database.

Online Transaction Processing

Online Transaction processing database applications are optimal for managing changing data, and usually have a large number of users who will be simultaneously performing transactions that change real-time data. Although individual requests by users for data tend to reference few records, many of these requests are being made at the same time. Common examples of these types of databases are airline ticketing systems and banking transaction systems. The primary concerns in this type of application are concurrency and atomicity.

Concurrency controls in a database system ensure that two users cannot change the same data, or that one user cannot change a piece of data before another user is done with it. For example, if you are talking to an airline ticket agent to reserve the last available seat on a flight and the agent begins the process of reserving the seat in your name, another agent should not be able to tell another passenger that the seat is available.

Atomicity ensures that all of the steps involved in a transaction complete successfully as a group. If any step fails, no other steps should be completed. For example, a banking transaction may involve two steps: taking funds out of your checking account and placing them into your savings account. If the step that

removes the funds from your checking account succeeds, you want to make sure that the funds are placed into your savings account or put back into your checking account.

Online Transaction Processing Design Considerations

Transaction processing system databases should be designed to promote:

- Good data placement.

I/O bottlenecks are a big concern for OLTP systems due to the number of users modifying data all over the database. Determine the likely access patterns of the data and place frequently accessed data together. Use filegroups and RAID (redundant array of independent disks) systems to assist in this.

- Short transactions to minimize long-term locks and improve concurrency.

Avoid user interaction during transactions. Whenever possible, execute a single stored procedure to process the entire transaction. The order in which you reference tables within your transactions can affect concurrency. Place references to frequently accessed tables at the end of the transaction to minimize the duration that locks are held.

- Online backup.

OLTP systems are often characterized by continuous operations (24 hours a day, 7 days a week) for which downtime is kept to an absolute minimum. Although Microsoft® SQL Server™ 2000 can back up a database while it is being used, schedule the backup process to occur during times of low activity to minimize effects on users.

- High normalization of the database.

Reduce redundant information as much as possible to increase the speed of updates and hence improve concurrency. Reducing data also improves the speed of backups because less data needs to be backed up.

- Little or no historical or aggregated data.

Data that is rarely referenced can be archived into separate databases, or moved out of the heavily updated tables into tables containing only historical data. This keeps tables as small as possible, improving backup times and query performance.

- Careful use of indexes.

Indexes must be updated each time a row is added or modified. To avoid over-indexing heavily updated tables, keep indexes narrow. Use the Index Tuning Wizard to design your indexes.

- Optimum hardware configuration to handle the large numbers of concurrent users and quick response times required by an OLTP system.

Decision Support

Decision-support database applications are optimal for data queries that do not change data. For example, a company can periodically summarize its sales data by date, sales region, or product and store this information in a separate database to be used for analysis by senior management. To make business decisions, users need to be able to determine trends in sales quickly by querying the data based on various criteria. However, they do not need to change this data. The tables in a decision-support database are heavily indexed, and the raw data is often preprocessed and organized to support the various types of

queries to be used. Because the users are not changing data, concurrency and atomicity issues are not a concern; the data is changed only by periodic, bulk updates made during off-hour, low-traffic times in the database.

Decision Support Design Considerations

Decision-support system databases should be designed to promote:

- Heavy indexing.

Decision-support systems have low update requirements but large volumes of data. Use many indexes to improve query performance.

- Denormalization of the database.

Introduce preaggregated or summarized data to satisfy common query requirements and improve query response times.

- Use of a star or snowflake schema to organize the data within the database.

Normalization

The logical design of the database, including the tables and the relationships between them, is the core of an optimized relational database. A good logical database design can lay the foundation for optimal database and application performance. A poor logical database design can impair the performance of the entire system.

Normalizing a logical database design involves using formal methods to separate the data into multiple, related tables. A greater number of narrow tables (with fewer columns) is characteristic of a normalized database. A few wide tables (with more columns) is characteristic of a nonnormalized database.

Reasonable normalization often improves performance. When useful indexes are available, the Microsoft® SQL Server™ 2000 query optimizer is efficient at selecting rapid, efficient joins between tables.

Some of the benefits of normalization include:

- Faster sorting and index creation.
- A larger number of clustered indexes.
- Narrower and more compact indexes.
- Fewer indexes per table, which improves the performance of INSERT, UPDATE, and DELETE statements.
- Fewer null values and less opportunity for inconsistency, which increase database compactness.

As normalization increases, so do the number and complexity of joins required to retrieve data. Too many complex relational joins between too many tables can hinder performance. Reasonable normalization often includes few regularly executed queries that use joins involving more than four tables.

Sometimes the logical database design is already fixed and total redesign is not feasible. Even then, however, it might be possible to normalize a large table selectively into several smaller tables. If the database is accessed through stored procedures, this schema change could take place without affecting applications. If not, it might be possible to create a view that hides the schema change from the applications.

Achieving a Well-Designed Database

In relational-database design theory, normalization rules identify certain attributes that must be present or absent in a well-designed database. A complete discussion of normalization rules goes well beyond the scope of this topic. However, there are a few rules that can help you achieve a sound database design:

- A table should have an identifier.

The fundamental rule of database design theory is that each table should have a unique row identifier, a column or set of columns used to distinguish any single record from every other record in the table. Each table should have an ID column, and no two records can share the same ID value. The column or columns serving as the unique row identifier for a table is the primary key of the table.

- A table should store only data for a single type of entity.

Attempting to store too much information in a table can prevent the efficient and reliable management of the data in the table. In the **pubs** database in SQL Server 2000, the titles and publishers information is stored in two separate tables. Although it is possible to have columns that contain information for both the book and the publisher in the **titles** table, this design leads to several problems. The publisher information must be added and stored redundantly for each book published by a publisher. This uses extra storage space in the database. If the address for the publisher changes, the change must be made for each book. And if the last book for a publisher is removed from the title table, the information for that publisher is lost.

In the **pubs** database, with the information for books and publishers stored in the **titles** and **publishers** tables, the information about the publisher has to be entered only once and then linked to each book. Therefore, if the publisher information is changed, it must be changed in only one place, and the publisher information will be there even if the publisher has no books in the database.

- A table should avoid nullable columns.

Tables can have columns defined to allow null values. A null value indicates that there is no value. Although it can be useful to allow null values in isolated cases, it is best to use them sparingly because they require special handling that increases the complexity of data operations. If you have a table with several nullable columns and several of the rows have null values in the columns, you should consider placing these columns in another table linked to the primary table. Storing the data in two separate tables allows the primary table to be simple in design but able to accommodate the occasional need for storing this information.

- A table should not have repeating values or columns.

The table for an item in the database should not contain a list of values for a specific piece of information. For example, a book in the **pubs** database might be coauthored. If there is a column in the **titles** table for the name of the author, this presents a problem. One solution is to store the name of both authors in the column, but this makes it difficult to show a list of the individual authors. Another solution is to change the structure of the table to add another column for the name of the second author, but this accommodates only two authors. Yet another column must be added if a book has three authors.

If you find that you need to store a list of values in a single column, or if you have multiple columns for a single piece of data (**au_lname1**, **au_lname2**, and so on), you should consider placing the duplicated data in another table with a link back to the primary table. The **pubs** database has a table for book information and another table that stores only the ID values for the books and the IDs of the authors of the books. This design allows any number of authors for a

book without modifying the definition of the table and allocates no unused storage space for books with a single author.

Data Integrity

Enforcing data integrity ensures the quality of the data in the database. For example, if an employee is entered with an **employee_id** value of **123**, the database should not allow another employee to have an ID with the same value. If you have an **employee_rating** column intended to have values ranging from **1** to **5**, the database should not accept a value of **6**. If the table has a **dept_id** column that stores the department number for the employee, the database should allow only values that are valid for the department numbers in the company.

Two important steps in planning tables are to identify valid values for a column and to decide how to enforce the integrity of the data in the column. Data integrity falls into these categories:

- Entity integrity
- Domain integrity
- Referential integrity
- User-defined integrity

Entity Integrity

Entity integrity defines a row as a unique entity for a particular table. Entity integrity enforces the integrity of the identifier column(s) or the primary key of a table (through indexes, UNIQUE constraints, PRIMARY KEY constraints, or IDENTITY properties).

Domain Integrity

Domain integrity is the validity of entries for a given column. You can enforce domain integrity by restricting the type (through data types), the format (through CHECK constraints and rules), or the range of possible values (through FOREIGN KEY constraints, CHECK constraints, DEFAULT definitions, NOT NULL definitions, and rules).

Referential Integrity

Referential integrity preserves the defined relationships between tables when records are entered or deleted. In Microsoft® SQL Server™ 2000, referential integrity is based on relationships between foreign keys and primary keys or between foreign keys and unique keys (through FOREIGN KEY and CHECK constraints). Referential integrity ensures that key values are consistent across tables. Such consistency requires that there be no references to nonexistent values and that if a key value changes, all references to it change consistently throughout the database.

When you enforce referential integrity, SQL Server prevents users from:

- Adding records to a related table if there is no associated record in the primary table.
- Changing values in a primary table that result in orphaned records in a related table.
- Deleting records from a primary table if there are matching related records.

For example, with the **sales** and **titles** tables in the **pubs** database, referential integrity is based on the relationship between the foreign key (**title_id**) in the **sales** table and the primary key (**title_id**) in the **titles** table.

User-Defined Integrity

User-defined integrity allows you to define specific business rules that do not fall into one of the other integrity categories. All of the integrity categories support user-defined integrity (all column- and table-level constraints in CREATE TABLE, stored procedures, and triggers).

Database Performance

When you design a database, you must ensure that the database performs all the important functions correctly and quickly. Some performance issues can be resolved after the database is in production, but other performance issues may be the result of a poor database design and can be addressed only by changing the structure and design of the database.

When you design and implement a database, you should identify the large tables in the database and the more complex processes that the database will perform, and give special consideration to performance when designing these tables. Also consider the effect on performance of increasing the number of users who can access the database.

Examples of design changes that improve performance include:

- If a table containing hundreds of thousands of rows must be summarized for a daily report, you can add a column or columns to the table that contains preaggregated data to be used only for the report.
- Databases can be overnormalized, which means the database is defined with numerous, small, interrelated tables. When the database is processing the data in these tables, it has to perform a great deal of extra work to combine the related data. This extra processing can reduce the performance of the database. In these situations, denormalizing the database slightly to simplify complex processes can improve performance.

In conjunction with correct database design, correct use of indexes, RAID (redundant array of independent disks), and filegroups is important for achieving good performance.

Hardware Considerations

Generally, the larger the database, the greater the hardware requirements. But there are other determining factors: the number of concurrent users/sessions, transaction throughput, and the types of operations within the database. For example, a database containing infrequently updated data for a school library would generally have lower hardware requirements than a 1-terabyte (TB) data warehouse containing frequently analyzed sales, product, and customer information of a large corporation. Aside from the disk storage requirements, more memory and faster processors would be needed for the data warehouse to enable more of the data to be cached in memory and queries referencing large amounts of data to be processed quickly.

Maintenance

After a database has been created and all objects and data have been added and are in use, there will be times when maintenance must be performed. For example, it is important to back up the database regularly. You may also need to create some new indexes to improve performance. These issues should be taken into consideration when you design the database to minimize the effect on users, the time taken to perform the task, and the effort involved.

Maintenance design guidelines include:

- Designing the database to be as small as possible and to exclude redundant information.

Normalizing your database can help you achieve this. For example, reducing the size of the database can help reduce the time taken to back up or, more importantly, restore a database. This is especially important during a restore operation because the database is unavailable while it is being restored.

- Designing partitioned tables rather than a single table, if the table will contain a large number of rows.

For example, a table containing every credit card transaction received by a bank could be split into multiple tables, with each table holding data for a single month. This can ease index maintenance if new indexes would otherwise have to be added to improve query performance. It may be necessary to create the index only on data from the last three months because older data is no longer referenced. The larger the table, the longer it takes to create new indexes.

Microsoft® SQL Server™ 2000 provides the Database Maintenance Plan Wizard for automating many of these tasks, thereby reducing or removing the work involved in database maintenance.