11. Architecture of Database Systems

11.1 Introduction

Software systems generally have an architecture, ie. possessing of a structure (form) and organisation (function). The former describes identifiable components and how they relate to one another structurally; the latter describes how the functions of the various structural components interact to provide the overall functionality of the system as a whole. Since a database system is basically a software system (albeit complex), it too possesses an architecture. A typical architecture must define a particular configuration of and interaction between data, software modules, meta-data, interfaces and languages (see Figure 11-1).

The architecture of a database system determines its capability, reliability, effectiveness and efficiency in meeting user requirements. But besides the visible functions seen through some data manipulation language, a good database architecture should provide:

a) Independence of data and programs
b) Ease of system design
c) Ease of programming
d) Powerful query facilities
e) Protection of data

Figure 11-1 General Database System Architecture

The features listed above become especially important in large organisations where corporate data are held centrally. In such situations, no single user department has responsibility over, nor can they be expected to know about, all of the organisation’s data. This becomes the job of a Database Administrator (DBA) who has a daunting
range of responsibilities that include creating, expanding, protecting and maintaining the integrity of all data while addressing the interests of different present and future user communities. To create a database, a DBA has to analyse and assess the data requirements of all users and from these determine its logical structure (database schema). This, on the one hand, will need to be efficiently mapped onto a physical structure that optimises retrieval performance and the use of storage. On the other, it would also have to be mapped to multiple user views suited to the respective user applications. For large databases, DBA functions will in fact require the full time services of a team of many people. A good database architecture should have features that can significantly facilitate these activities.

11.2 Data Abstraction

To meet the requirements above, a more sophisticated architecture is in fact used, providing a number of levels of data abstraction or data definition. The database schema, also known as Conceptual Schema, mentioned above represents an information model at the logical level of data definition. At this level, we abstract out details like computer storage structures, their restrictions, or their operational efficiencies. The view of a database as a collection of relations or tables, each with fixed attributes and primary keys ranging over given domains, is an example of a logical level of data definition.

The details of efficiently organising and storing objects of the conceptual schema in computers with particular hardware configurations are dealt with at the internal (storage) level of data definition. This level is also referred to as the Internal Schema. It maps the contents of the conceptual schema onto structures representing tuples, associated key organisations and indexes, etc, taking into account application characteristics and restrictions of a given computer system. That is, the DBA describes at this level how objects of the conceptual schema are actually organised in a computer. Figure 11-2 illustrates these two levels of data definition.

Figure 11-2 The Logical and Internal Levels of Data Abstraction
At a higher level of abstraction, objects from the conceptual schema are mapped onto views seen by end-users of the database. Such views are also referred to as *External Schemas*. An external schema presents only those aspects of the conceptual schema that are relevant to the particular application at hand, abstracting out all other details. Thus, depending on the requirements of the application, the view may be organised differently from that in the conceptual schema, e.g. some tables may be merged, attributes may be suppressed, etc. There may thus be many views created—one for each type of application. In contrast, there is only one conceptual and one internal schema. All views are derived from the same conceptual schema. This is illustrated in Figure 11-3 which shows two different user views derived from the same conceptual schema.

Thus, modern database systems support three levels of data abstraction: External Schemas (User Views), Conceptual Schema, and Internal (Storage) Schema.

The DDL we discussed in earlier chapters is basically a tool only for conceptual schema definition. The DBA will therefore usually need special languages to handle the external and internal schema definitions. The internal schema definition, however, varies widely over different implementation platforms, i.e. there are few common principles for such definition. We will therefore say little more about them in this book.

![Figure 11-3 User Views (External Schema)](image)

As to external schema definitions, note that in the relational model, the Data Sub-Languages can be used to both describe and manipulate data. This is because the expressions of a Data Sub-Language themselves denote relations. Thus, a collection of new (derived) relations can be defined as an external schema.
For example, suppose the following relations are defined:

Customer( C#, Cname, Ccity, Cphone )
Product( P#, Pname, Price )
Transaction( C#, P#, Date, Qnt )

We can then define an external view with a construct like the following:

Define View My_Transaction_1 As
   Select Cname, Ccity, Date, Total_Sum=Price*Qnt
   From Customer, Transaction, Product
   Where Customer.C# = Transaction.C#
      & Transaction.P# = Product.P#

which defines the relation (view):

My_Transaction_1( Cname, Ccity, Date, Total_Sum )

This definition effectively maps the conceptual database structure into a form more convenient for a particular user or application. The extension of this derived table is itself derived from the extensions of the source relations. This is illustrated in Figure 11-4 below.

This is a very important property of the relational data model: a unified approach to data definition and data manipulation.

11.3 Data Administration

Functions of a DBA include:

• Creation of the database
To create a database, a DBA has to analyse and assess the requirements of the users and from these determine its logical structure. In other words, the DBA has to design a conceptual schema and a first variant of an internal schema. When the internal schema is ready, the DBA must load the database with actual data.

- Acting as intermediary between users and the database

A DBA is responsible for all user facilities determined by external schemas, i.e. the DBA is responsible for defining all external schemas or user views.

- Ensuring data privacy, integrity and security

In analysing user requirements, a DBA must determine who should have access to which data and subsequently arrange for appropriate privacy locks (passwords) for identified individuals and/or groups. The DBA must also determine integrity constraints and arrange for appropriate data validation to ensure that such constraints are never violated. Last, but not least, the DBA must make arrangements for data to be regularly backed up and stored in a safe place as a measure against unrecoverable data losses for one reason or another.

At first glance, it may seem that a database can be developed using the conventional “waterfall” technique. That is, the development process is a sequence of stages, with work progressing from one stage to the next only when the preceding stage has been completed. For relational database development, this sequence will include stages like eliciting user requirements, analysing data relationships, designing the conceptual schema, designing the internal schema, loading the database, defining user views and interfaces, etc, through to the deployment of user facilities and database operations.

In practice, however, when users start to work with the database, the initial requirements inevitably change for a number of reasons including experience gained, a growing amount of data to be processed, and, in this fast changing world, changes in the nature of the business it supports. Thus, a database need to evolve, learning from experience and allowing for changes in requirements. In particular, we may expect periodic changes to:

- improve database performance as data usage patterns changes or becomes clearer
- add new applications to meet new processing requirements
- modify the conceptual schema as understanding of the enterprise’s perception of data improves

Changing a database, once the conceptual and internal schemas have been defined and data actually loaded, can be a major undertaking even for seemingly small conceptual changes. This is because the data structures at the storage layer will need to be reorganised, perhaps involving complete regeneration of the database. A good DBMS should therefore provide facilities to modify a database with a minimum of inconvenience. The desired facilities can perhaps be broadly described to cover:

- performance monitoring
- database reorganisation
• database restructuring

By performance monitoring we mean the collection of usage statistics and their analysis. Statistics necessary for performance optimisation generally fall under two headings: static and dynamic statistics. The static statistics refer to the general state of the database and can be collected by special monitoring programs when the database is inactive. Examples of such data include the number of tuples per relation, the population of domains, the distribution of relations over available storage space, etc. The dynamic statistics refer to run-time characteristics and can be collected only when the database is running. Examples include frequency of access to and updating of each relation and domain, use of each type of data manipulation operator and associated response times, frequency of disk access for different usage types, etc.

It is the DBA’s responsibility to analyse such data, interpret them and where necessary take steps to reorganise the storage schema to optimise performance. Reorganising the storage schema also entails the subsequent physical reorganisation of the data themselves. This is what we mean by database reorganisation.

The restructuring of the conceptual schema implies changing its contents, such as:

• adding/removing data items (i.e. columns of a relation)
• adding/removing entire relations
• splitting/recombining relations
• changing a relation’s primary keys
• …etc

For example, assuming the relations as on page 3, suppose we now wish to record also for each purchase transaction the sales representative responsible for the sale. We will need therefore to add a column into the Transaction relation, say with column name R#:

\[
\text{Transaction( C#, P#, R#, Date, Qnt )}
\]

new attribute added

The intention, of course, is to record a unique value under this column to denote a particular sales representative. Details of such sales representatives will then be given in a new relation:

\[
\text{Representative( R#, Rname, Rcity, Rphone) }
\]

A restructured conceptual schema will normally be followed by a database reorganisation in the sense explained above.

11.4 Data Independence

Data independence refers to the independence of one user view (external schema) with respect to others. A high degree of independence is desirable as it will allow a DBA to change one view, to meet new requirements and/or to optimise performance, without
affecting other views. Relational databases with appropriate relational sub-languages have a high degree of data independence.

For example, suppose that the view

\[ \text{My\_Transaction\_1( Cname, Ccity, Date, Total\_Sum) } \]

as defined on page 4 no longer meet the user’s needs. Let’s say that Ccity and Date are no longer important, and that it is more important to know the product name and quantity purchased. This change is easily accommodated by changing the select-clause in the definition thus:

Define View My\_Transaction\_1 As
Select Cname, Pname, Qnt, Total\_Sum=Price*Qnt
From Customer, Transaction, Product
Where Customer.C# = Transaction.C#
& Transaction.P# = Product.P#

This replaces the original specification of Ccity and Date items

If each view is defined separately over the conceptual schema, then as long as the conceptual schema does not change, a view may be redefined without affecting other views. Thus the above change will have no effect on other views, unless they were built upon My\_Transaction\_1.

Data independence is also used to refer to the independence of user views relative to the conceptual schema. For example, the reader can verify that the change in the conceptual schema in the last section (adding the attribute R# to Transaction and adding the new relation Representative), does not affect My\_Transaction\_1 - neither the original nor the changed view! In general, if the relations and attributes referred to in a view definition are not removed in a restructuring, the view will not be affected. Thus we can accommodate new (additive) requirements without affecting existing applications.

Lastly, data independence may also refer to the extent to which we may change the storage schema without affecting the conceptual or external schemas. We will not elaborate on this as we have pointed out earlier that the storage level is too diverse for meaningful treatment here.

### 11.5 Data Protection

There are generally three types of data protection that any serious DBMS must provide. These were briefly described in Chapter 1 and we summarise them here:

1. Authorisational Security
   
   This refers to protection against unauthorised access and includes measures such as user identification and password control, privacy keys, etc.

2. Operational Security
   
   This refers to maintaining the integrity of data, i.e. protecting the database from the introduction of data that would violate identified integrity constraints.
3. Physical Security

This refers to procedures to protect the physical data against accidental loss or damage of storage equipment, theft, natural disaster, etc. It will typically involve making periodic backup copies of the database, transaction journalling, error recovery techniques, etc.

In the context of the relational data model, we can use relational calculus as a notation to define integrity constraints, i.e. we define them as formulae of relational calculus. In this case, however, all variables must be bound variables as we are specifying properties over their ranges rather than looking for particular instantiations satisfying some predicate. For example, suppose that for the Product relation, the Price attribute should only have a value greater than 100 and less than 99999. This can be expressed (in DSL Alpha style) as:

\[
\text{Range Product X ALL;}
\]

\[
(X.\text{Price} > 100 \land X.\text{Price} < 99999)
\]

This is interpreted as an assertion that must always be true. Any data manipulation that would make it false would be disallowed (typically generating messages informing the user of the violation). Thus, not only does the relational data model unify data definition and manipulation, but its control as well.

In the area of physical security, database backups should of course be done periodically. For this purpose, it is perhaps best to view a database as a large set of physical pages, where each page is a block of fixed size serving as the basic unit of interaction between the DBMS and storage devices. A database backup is thus essentially a copy of the entire set of pages onto another storage medium that is kept in a secure and safe place. Aside from the obvious need for backups against damage of storage devices, theft, natural disasters and the like, backups are necessary to recover a consistent database in the event of a database ‘crash’. Such crashes can occur in the course of a sequence of database transactions, particularly transactions that modify the database content.

Suppose, for example, that the last backup was done at time \( t_0 \), and subsequent to that, a number of update transactions were applied one after another. Suppose further that the first \( n \) transactions were successfully completed, but during the \((n+1)\)th transaction a system failure occurred (e.g. disk malfunction, operating system crash, power failure, etc) leaving some pages in a corrupted state. In general, it is not possible to just reapply the failed transaction—the failure could have corrupted the updates performed by previous transactions as well, or worse, it could have damaged the integrity of the storage model as to make some pages of the database unreadable! We have no recourse at this point but to go back to the last known consistent state of the database at time \( t_0 \), i.e. the entire contents of the last backup is reinstated as the current database. Of course, in doing so, all the transactions applied after \( t_0 \) are lost.

At this point it may seem reasonable that, to guard against losing too much work, backups should perhaps be done after each transaction—then at most only the work of
one transaction is lost in case of failure. However, many database applications today are transaction intensive typically involving many online users generating many transactions frequently (eg. online airline reservation system). Many databases, on the other hand, are very large and an entire backup could take hours to complete. While backup is being performed the database must be inactive. Thus, it should be clear that this proposition is impractical.

As it is clearly desirable that transactions since the last backup are also somehow saved in the event of crashes, an additional mechanism is needed. Essentially, such mechanisms are based on journalling successful transactions applied to a database. This simply means that a copy of each transaction (or affected pages) is recorded in a sequential file as they are applied to the database.

The simplest type of journalling is the Forward System Journal. In this, whenever a page is modified, a copy of the modified page is also simultaneously recorded into the forward journal.

To illustrate this mechanism, let the set of pages in a database be \( P = \{p_1, p_2, \ldots, p_n\} \). If the application of an update transaction \( T \) on the database changes \( P_T \), where \( P_T \subseteq P \), then \( T(P_T) \) will be recorded in the forward journal. We use the notation \( T(P_i) \) to denote the set of pages \( P_i \) after the transaction \( T \) has changed each page in \( P_i \). Likewise, we write \( T(p_i) \) to denote a page \( p_i \) after it has been changed by transaction \( T \). Furthermore, if \( T \) was applied successfully (ie. no crash during its processing), a separator mark, say ‘;’, would be written to the journal. Thus, after a number of successful transactions, the journal would look as follows:

\[
< T(P_{T_1}) ; T(P_{T_2}) ; \ldots ; T(P_{T_k}) ; >
\]

As a more concrete example, suppose transaction \( T_1 \) changed \( \{p_1, p_2, p_3\} \), \( T_2 \) changed \( \{p_2, p_3, p_4\} \), and \( T_3 \) changed \( \{p_3, p_4, p_5\} \), in that order and all successfully carried out. Then the journal would contain:

\[
< T_1( \{p_1, p_2, p_3\} ) ; T_2( \{T_1(p_2), T_1(p_3), p_4\} ) ; T_3( \{T_2(T_1(p_3)), T_2(p_3), p_5\} ) ; >
\]

Now suppose a crash occurred just after \( T_3 \) has been applied. The recovery procedure consists of two steps:

a) replace the database with the latest backup

b) read the system journal in the forward direction (hence the term ‘forward’ journal) and, for each set of journal pages that precedes the separator ‘;’, use it to replace the corresponding pages in the database. Effectively, this duplicates the effect of applying transactions in the order they were applied prior to the crash.

The technique is applicable even if the crash occurred during the last transaction. In this case, the journal for the last transaction would be incomplete and, in particular, the separator ‘;’ would not be written out. Say that transaction \( T_3 \) was interrupted after modifying pages \( p_3 \) and \( p_4 \) but before it could complete modifying \( p_5 \). Then the journal would look as follows:
In this case, recovery is exactly as described above except that the last incomplete block of changes will be ignored (no separator ';'). Of course, the work of the last transaction is lost, but this is unavoidable. It is possible, however, to augment the scheme further by saving the transaction itself until its effects are completely written to the journal. Then T3 above can be reapplied, as a third step in the recovery procedure.

While the forward journal can recover (almost) fully from a crash, its disadvantage is that it is still a relatively slow process—hundreds or even thousands of transactions may have been applied since the last full backup, and the corresponding journals of each of these transactions must be copied back in sequence to restore the state of the database. In some applications, very fast recovery is needed.

In these cases, the Backward System Journal will be the more appropriate journalling and recovery technique. With this technique, whenever a transaction changes a page, the page contents before the update is saved. As before, if the transaction successfully completes, a separator is written. Thus the backward journal for the same example as above would be:

\[<\{p_1, p_2, p_3\} ; \{T1(p_2), T1(p_3), p_4\} ; \{T2(T1(p_3)), T2(p_4), \ldots\}\rangle\]

Since each block of journal pages represents the state immediately before a transaction is applied, recovery consists of only one step: read the journal in the backward direction until the first separator and replace the pages in the database with the corresponding pages read from the journal. Thus, the backward journal is like an ‘undo’ file—the last block cancels the last transaction, the second last cancels the second last transaction, etc.

Features such as those discussed above can significantly facilitate the management of corporate data resources. Such features, together with the overall architecture and the Data Model examined in previous chapters, determine the quality of a DBMS and are thus often used as part of the principal criteria used in critical evaluation of competing DBMSs.