

Chapter 2

Enterprise Data Management for Transaction and Analytical Processing

Enterprise data management has to support all business processes of a company from storing and providing data during daily operations, for example, sales, purchasing, and payroll accounting, to analyzing data for strategic decision making. Daily operations are supported by transaction processing systems. A transaction processing system contains applications that automate business activities [9, Chap. 1]. Sales order processing is a typical example for such an application. Analytical processing is provided by decision support systems. With their help, decision makers within companies gain insights into daily operations from which they defer information for making strategic decisions regarding the future of the company. Decision support systems have experienced an explosive growth in the 1990s [29].

According to Haderle [84], in the 1960s and 1970s the focus of commercial data management systems lay on batch and transaction processing environments. Batch processing means the definition of jobs that execute multiple operations within one request and that can run completely without user interaction. The response time requirements of batch jobs are flexible meaning that a response does not need to be produced immediately and they can be scheduled as resources become available, for example, during non-peak periods. This facilitates the management of the limited computing resources [9, Chap. 1]. Thus, batch processing was well suited to the goal of data management systems at that time, which was to manage concurrent read and write access to data, while minimizing resource utilization. Therefore, batch processing was the major application model and for some application areas, it is still today. In companies with a large customer base, dunning runs to determine the customers whose payments are late are an example for which batch jobs are scheduled in non-peak periods.

Transaction processing applications started to evolve at this time, too. Gray [72] defines a transaction as a mechanism that queries and transforms the state of the accessed data. A transaction is a fixed sequence of operations that have to be completely processed or not at all. It can contain several statements clustering these operations. The transaction concept emerged with four properties that incorporate essential features needed by business applications. These are atomicity, consistency, isolation, and durability (ACID [86]).

The isolation property means that events happening during the execution of a transaction are hidden from other transactions that are running concurrently. Gray et al. [76] originally introduced isolation in four *degrees of consistency*: (1) protecting other transactions from updates of a transaction, (2) additional protection from losing updates, (3) additional protection from reading incorrect data items, and (4) additional protection from reading incorrect relationships among data items. The other three properties were later defined in [73]. Atomicity ensures that all operations within a transaction are completed or none is. Consistency means that the data is left in a consistent state after the normal end of a transaction has been reached. Durability ensures that the results of a transaction upon successful completion are preserved by the system and that they survive any subsequent malfunction.

Early transaction processing systems already provided the functionality to interactively execute very short transactions, in a so-called on-line processing mode. This is where the term online transaction processing (OLTP) stems from. In online transaction processing, the changes and results are immediately visible as opposed to batch processing. In the beginning of the 1980s, the first commercial relational databases that enabled interactive processing appeared. Similar to OLTP, the term online analytical processing (OLAP) expresses that decision makers interactively work with the system.

This chapter presents the foundations that this thesis builds on. It describes the most relevant and widely used database design alternatives within the areas of transactional and analytical processing. The understanding of database design decisions taken in the past and the reasons behind facilitate building tomorrow's hybrid systems and the benchmarks to evaluate and compare them.

Section 2.1 starts with the description of the data models used within the databases for enterprise data management, explaining their characteristics. Figure 2.1 presents an overview of past and today's most commonly used data models and underlying database designs in OLTP and OLAP systems. Early data management was handled via flat files. To avoid implementing the access and data modification logic in each application, database management systems emerged as an abstraction layer that provides standard interfaces to manipulate data. A database management system (DBMS) is a collection of interrelated data and a set of programs to access that data. The term database refers to the collection of interrelated data, which contains information relevant to, e.g., an enterprise [191]. In OLAP systems dimensional modeling appeared along with the multidimensional and hybrid data models to better match the users needs. Concerning the relational data model, different database designs were introduced that are used in OLTP, OLAP, or in both kinds of systems.

The relational data model emerged as today's most prevalent data model used in business data processing in the areas of transaction processing as well as analytical processing. Therefore, the scope of the subsequent sections is narrowed down to the relational data model and the diversity of database designs within that area.

Section 2.2 covers database design alternatives and optimization opportunities in the area of relational databases. For later discussions, it includes relevant aspects

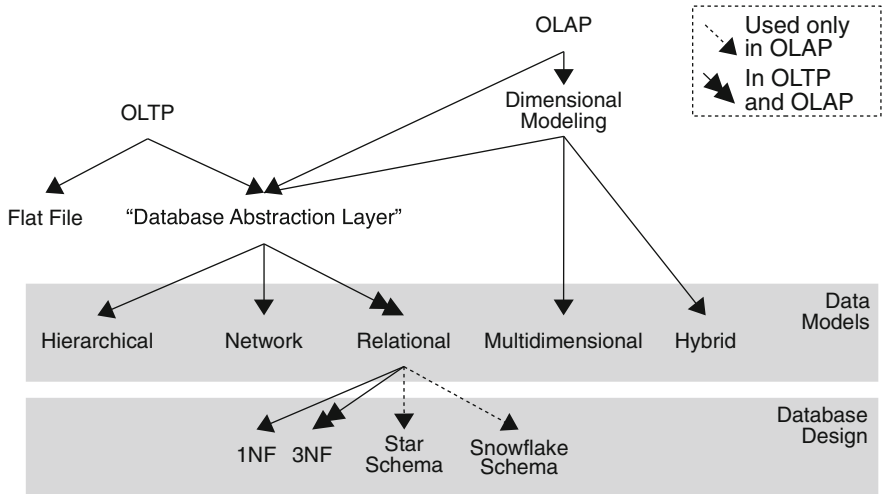


Fig. 2.1 Data management alternatives in OLTP and OLAP systems

from all database design levels: These are normalization on the logical level, optimization of access paths on the physical database design level, for example, indexes and views, and specific physical storage layouts used in database systems for transactional and analytical processing.

2.1 Data Models for Transaction and Analytical Processing

Codd [40] gives a definition of the term data model as the combination of

1. A collection of data structure types,
2. A set of operators and rules to retrieve or modify data from any part of the aforementioned data structure types, and
3. A number of general integrity rules that define consistent database states and changes of state.

Codd [40] emphasizes that all three parts of this definition of data models are equally important and argues that 2 and 3 are essential to understand how a structure based on a certain data model behaves. Without a defined set of operations any application using a structure has to be made aware of the structures internal workings regarding, for example, the conjunction of parts of the structure.

All examples in this section refer to a customer order scenario. Figure 2.2 presents the conceptual overview of the entities and relations for this scenario where customers order products and the respective products are being shipped to the customers. Each customer can place as many orders for as many products as needed

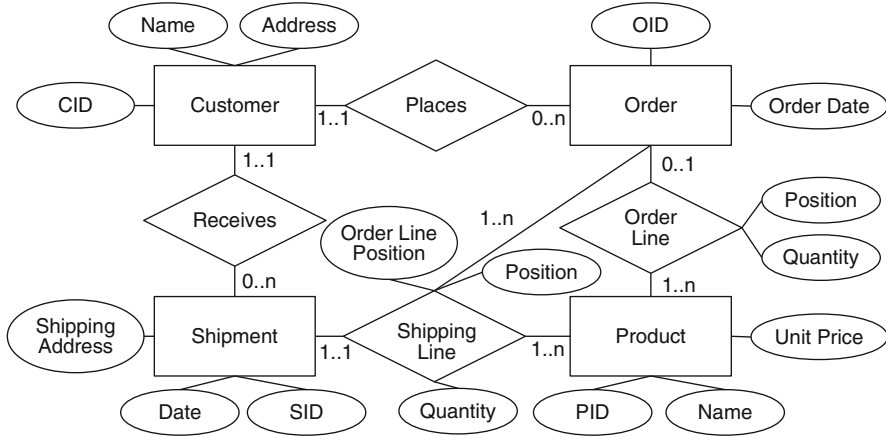


Fig. 2.2 Customer order entity relationship (E/R) diagram (Chen's Notation [33])

and can receive the ordered products in any number of shipments. According to, for example, availability, products within an order are assigned to shipments.

In the following, the data models for transaction and analytical processing systems that have been used at some point in the history or are still used are introduced. Their characteristics, which led to the development of new data models, are discussed, and finally they are compared according to their application areas.

2.1.1 Transaction Processing Systems

With the development of the first transaction processing systems in the 1960s, respective data models to store and manipulate data through applications in a standard way were introduced. Until the mid 1970s, most organizations used file systems to organize their data in so-called flat files. Only a small number used database systems [34]. The first database systems, developed in the 1960s were hierarchical and network systems. The hierarchical and network data models were afterwards defined as an abstraction from the real systems that already existed [40]. In contrast, Codd [38] introduced the relational data model before the implementation of relational database systems.

Flat File Model

The first step to manage data in a structured way was a simple organization of data in records (sets of fields) that were stored in files. The name flat file model or file management system stems from exactly this organization where the database consists of one or more files containing records.

To give a simple example, a company wants to store data about its customers and the orders of its customers, i.e., the company's sales orders. Using flat files, two alternatives for storing such data are immediately conceivable: (a) one file that includes records, with each record containing customer as well as sales order data, (b) two files with one including records of sales order data and the other including records of customer data. In alternative (a) the relationship between customers and their sales orders is explicit within each record, but all records for different sales orders of the same customer contain the same data concerning this customer. This makes updates to customer data a hazard as all records concerning the updated customer have to be changed. In alternative (b), this problem does not arise, but applications have to handle the relationships between customers and their orders themselves as flat files provide no concept for managing relationships.

Haderle [84] states that the flat file data organization already offered limited concurrency and recovery functionality. These are a basis for transaction processing. Moreover, since management of simple records did not include relationships between records or different kinds of records, the knowledge about data organization and determination of data semantics was left to the applications, i.e., no unified way of accessing data and standard data operations were defined as part of this data organization.

The drawback of this approach is that data access directly manipulates the physical data organization. If several applications access the same data, the logic for data access, interpretation, and operations has to be implemented in all of them. These redundant snippets of code increase maintenance overhead. In addition, advances in storage hardware technology introduce changes in the physical data organization to utilize new features and these entail changes to the code within each of the affected applications.

Hierarchical Model

An early work that separated the definition and layout of the data files from their access was a program called Generalized Data Update Access Method (GUAM) [163]. Access logic concerning data organization, maintenance, and data integrity was moved from the applications into a separate layer. Consequently, application code was simplified and the direct dependence of applications on physical data storage was removed [163].

GUAM built on the concept that larger components are created from the combination of smaller components resulting in a hierarchical structure [46, p. 24]. Thus, the hierarchical data model appeared in the 1960s out of the need to manage tremendous amounts of data created by complex manufacturing projects, such as the Apollo rocket project [47, p. 36]. In this context, IBM's Information Management System (IMS) was developed as a joint project with the North American Rockwell Space Division based on the GUAM program. The joint project ended in 1968, but IBM continued to develop IMS as a database and released it as a product called IMS/360 Version 1 in September 1969 [138].

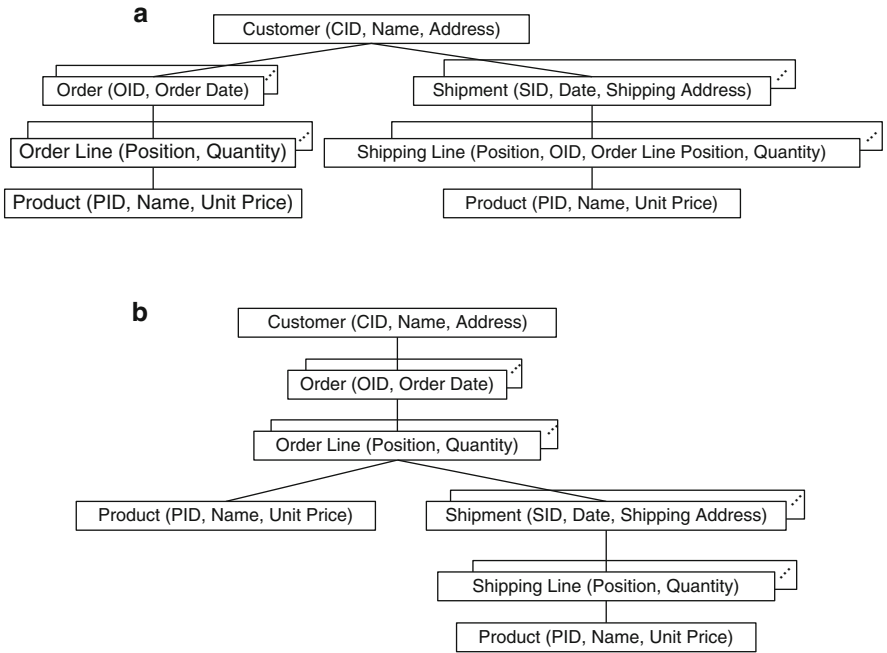


Fig. 2.3 Logical structures of the hierarchical customer order hierarchy. (a) No order/shipment mapping. (b) Direct mapping of shipments to order lines

The hierarchical model contains several levels starting from a root node, each level with nodes that act as the parents of the nodes on the next lower level if there is a relation between them. Consequently, the hierarchical model is a set of *one-to-many* relationships between parent and child nodes. Each node is stored as a record that maintains links to the node's parent and its children. Different types of nodes can be modeled in order to map diverse types of data entities, for example, a customer entity that contains the name of the customer and his contact data, or a product entity containing a description and pricing information. Applications retrieve data from hierarchical databases by finding the root node of a tree and then following the pointers stored in the records.

Figure 2.3 shows two alternative logical structures to model the customer order hierarchy. Depending on frequent access paths one or the other alternative is of advantage. Considering applications that frequently access shipments with no regard of the associated orders, the structure in Fig. 2.3a is beneficial as the depth of the tree that needs to be traversed is kept at a minimum and the number of branches correlates with the number of shipments. In contrast, shipments are scattered across orders in Fig. 2.3b, which benefits applications that rely on the relation between orders and shipments, such as, reporting orders that have not been shipped (completely).

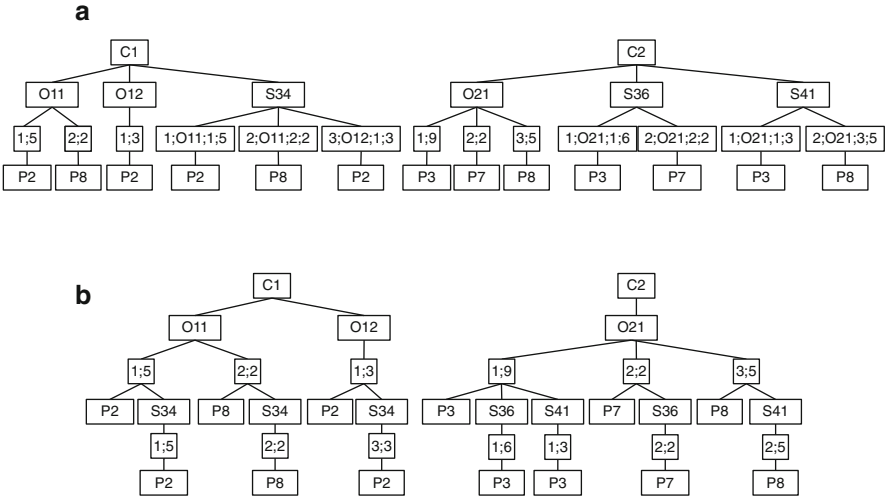


Fig. 2.4 Example hierarchy instances for the customer order hierarchy. (a) No order/shipment mapping. (b) Direct mapping of shipments to order lines

Figure 2.4 exemplifies the aforementioned alternatives for the hierarchical customer order hierarchy. Nodes in the hierarchical model are instances of entities, e.g., a customer named *C1*. Listing 2.1 presents the orders and shipments depicted in Fig. 2.4. Two customers *C1* and *C2* place orders for the products *P2*, *P3*, *P7*, and *P8*. The numbers provided in the order lines and shipment lines represent the ordered and shipped quantities.

The different modeling alternatives introduce differing levels of redundancy. While in the alternative in Fig. 2.4a only the product instances are repeated and an overview of complete orders and shipments per customer is simply achieved by scanning the respective subtrees, the alternative in Fig. 2.4b stores redundant shipment data for each order line besides the redundant product data, but it simplifies queries associating order lines with their shipments. For example, it is easy to determine that the order for product *P3* of customer *C2* is split across two shipments, that is, *S36* and *S41*, but to analyze the contents of an entire shipment, a scan of the complete subtree of this customer is needed.

Logical relationships were added in IMS/360 2.0 to efficiently handle many-to-many relationships [138]. They interrelate segments from different physical hierarchies building a logical hierarchy. Logically connected physical hierarchies may constitute a network data structure, although application data is still stored in one or more physical hierarchies [132]. Concerning the previous example, logical relationships can help to avoid the redundant storage of product records: An own hierarchy for products can be introduced and logical paths from the order line records of the customer order hierarchy to these product records can be created.

```

Customers = { (C1, "Werkstatt Hein", "Rosenthaler Grenzweg
              5, Berlin"),
              (C2, "Mc Tools", "361 Peachtree Street,
              Atlanta") }
Products = { (P2, "Integral Panel Clamp C4.a", 19.99),
              (P3, "Split Point Drill 35mm", 4.99),
              (P7, "Machinist's Chest XK044", 209.00),
              (P8, "Hot Ring Plier Set T68", 27.00) }
Orders(C1) = { (O11, 2011-01-05), (O12, 2011-01-07) }
Orders(C2) = { (O21, 2011-01-27) }
OrderLines(O11) = { (5, P2), (2, P8) }
OrderLines(O12) = { (3, P2) }
OrderLines(O21) = { (9, P3), (2, P7), (5, P8) }
Shipment(C1) = { (S34, 2011-01-14, "Rosenthaler Grenzweg
              5, Berlin") }
Shipment(C2) = { (S36, 2011-02-03, "1055 Ashbury Rd.,
              Fulton"),
              (S41, 2011-02-17, "1055 Ashbury Rd.,
              Fulton") }
ShippingLines(S34) = { (5, P2), (2, P8), (3, P2) }
ShippingLines(S36) = { (6, P3), (2, P7) }
ShippingLines(S41) = { (3, P3), (5, P8) }

```

Listing 2.1 Order and shipment example

IBM's IMS as the prevalent representative of the hierarchical data model is still in usage for enterprise data management today and widely spread in banking and insurance [121, 155].

Network Model

Similar to the hierarchical model, the network model has been defined after it was already used in database system implementations. Network models have been developed by the Conference on Data Systems Languages (CODASYL) Data Base Task Group (DBTG) [37]. On that account, they are also called CODASYL database models or DBTG database models.

Record types in the network model represent entities and set types represent relationships between entities. Each set type has exactly one owner record type and one or more member record types. The term record denotes a specific instance of an entity. A specific relation of this record to other records is called set. Each set contains exactly one owner record and zero or more member records of its defined member record types. In contrast to the hierarchical model, many-to-many associations can directly be represented by creating a new entity that represents this association [202].

Figure 2.5 presents the network model of the customer order example introduced in Fig. 2.2 according to the data structure diagram notation by Bachman [7]. Record

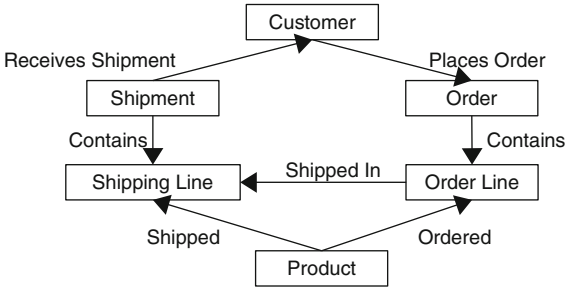


Fig. 2.5 Network model for customer orders and shipments

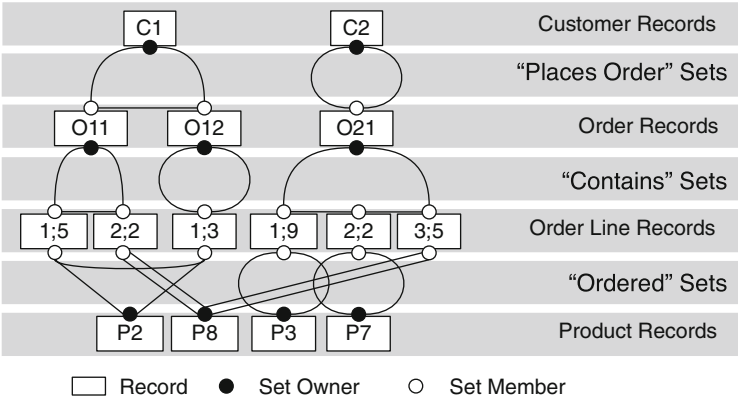


Fig. 2.6 Navigational routes through the customer order network model

types are depicted as rectangles and set types are depicted as arrows between record types. The owner record type of a set type is connected to the tail of the set types' arrow and its member record type is connected to the head.

In addition to defining the data structure, the network model sets up navigational routes through this data structure to access data. This is similar to the pointers in the hierarchical model. The navigational routes for a part containing customers, orders, order lines, and products of the network model depicted in Fig. 2.5 are shown in Fig. 2.6.

Through the network structure, entire contexts around entities can easily be retrieved along the modeled navigational paths. A query that exemplifies the advantage of the network structure over the hierarchical structure, assuming no logical relationships have been defined within the hierarchy, is: *Retrieve the top ten customers with respect to the sales amount of the best-selling product of customer C1.* In the hierarchical model above this requires traversing all nodes in the subtree of customer C1 to find the most-bought product and then traversing the entire tree to find the top ten other customers as product data is stored in the leaf nodes. In the network model the first step to find the best-selling product of customer C1 is the

same, but then the *Ordered* set associated to that product can be used to find and evaluate the orders of other customers. In this case, only customers having actually ordered that product are considered.

To achieve a similar behavior in the hierarchical model, IBM IMS provides bidirectional logical relationships [97] that are supported by a subset of its hierarchical database types. With these, logical paths from order line nodes of one physical hierarchy can be modeled that point to nodes of ordered products stored in another physical hierarchy and from the product nodes back to the first hierarchy to all order line nodes that contain this product. In contrast to the network model, this approach needs additional logical structures and pointers, increasing the overhead for the DBMS.

Relational Model

In the late 1970s, IBM developed “System R” as an experimental prototype for a DBMS that uses the relational database model (RDBMS – relational database management system). According to Astrahan et al. [6] System R was intended to demonstrate that a relational system can be used in real environments with a comparable performance to the existing systems of that time. The structured query language (SQL), initially called “SEQUEL” (structured English query language) [27], was developed as a query language to retrieve and manipulate stored data in System R. To test and evaluate System R, several installations at internal locations of IBM were set up. The project ended in 1979 with the result that the relational model is the basis for a viable database technology with commercial potential [93, p. 85]. Chamberlin et al. [28] came to the conclusion that databases on the basis of the relational model are able to support many concurrent users, who perform repetitive transactions and ad hoc queries. They argue that the high-level user interface enabled by the relational database model positively affects user productivity for developing new applications.

Along with System R, INGRES (Interactive Graphics and Retrieval System) was one of the first projects to provide a proof of concept for a practical DBMS based on the relational model. INGRES was started as a research project at the University of Berkeley, California. The motivation behind INGRES was the utilization of two basic characteristics of the relational model, namely, the high degree of data independence and the possibility for a high level query language [199]. INGRES included an own language called “QUERy Language” (QUEL). When SQL evolved as the standard database language [50], INGRES was converted to supporting SQL [93, p. 86], while continuing to support QUEL.

IBM’s System R and INGRES, however, were not the first to spawn a commercial relational database for the market. Greenwald et al. [77] claim that Oracle V.2, released in 1979, was the world’s first commercial relational database. According to Hernandez and Viescas [93], the first commercial version of INGRES entered the market in 1981. IBM announced its own RDBMS called SQL/Data System in 1981 [41] and shipped it starting in 1982. In 1983, IBM announced Database 2

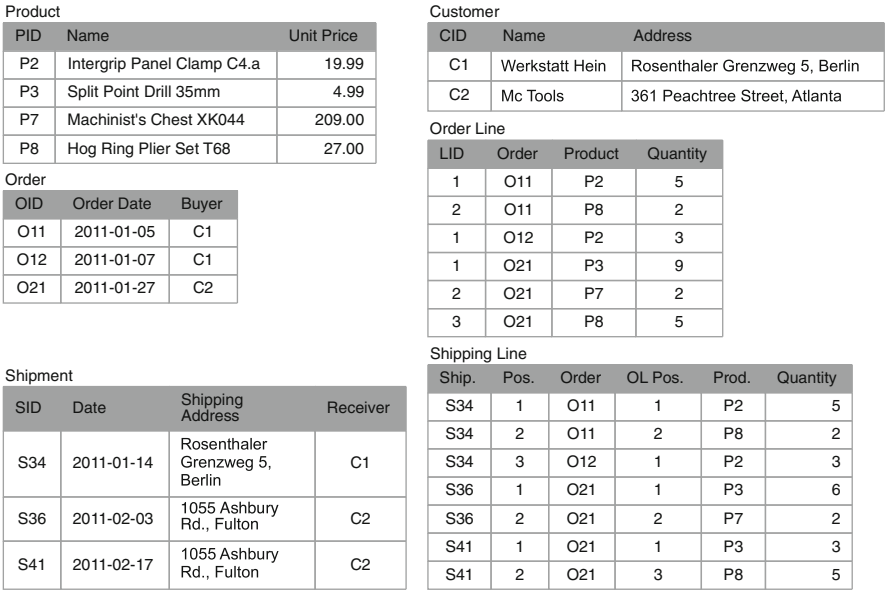


Fig. 2.7 Relational structure for the customer order example

(DB2), which was shipped starting in 1985. DB2 evolved from System R, IMS and technological advances made since then [85].

Compared to the network model, in the relational model an entity is represented as one or more tables. Single instances of an entity or parts of that entity are represented as a tuple within the tables that make up the entity. Relationships are also represented as tables, with their tuples associating the tuples of other tables (the instances of other entities) [143]. A tuple is a set of values, one for each characteristic of the entity, also called attribute. To uniquely identify each tuple of a table, a primary key can be defined as a subset of a table’s attributes such that no combination of their values is encountered more than once within the table. In contrast to the record (a tuple’s physical representation on the storage medium), a tuple does not declare an order on the attributes it contains. The attribute values of a tuple are accessed by their attribute names. Additionally, no order of tuples of a table exists on the logical abstraction layer.

Figure 2.7 shows the relational structure for the customer order example given in Fig. 2.2. In this structure, each entity is modeled as one table and the two relations *OrderLine* and *ShippingLine* are modeled as tables as well. They include references to those instances (tuples) of the entities taking part in the relation. For example, the *Product* and *Order* attributes of the *OrderLine* table connect the respective order and product instances. The relations *Places* and *Receives* are modeled as additional attributes in the tables of those entities referring to them. For the *Places* relation this means the attribute *Buyer* in the *Order* table and for *Receives* an attribute *Receiver* is added to the *Shipping* table. Such references to existing tuples within other tables

are called foreign keys if their elements are values of the primary key of another table [38].

Basic operations for data retrieval in the relational structure are set operations, such as, union, intersection, and difference and further typical operation are [69]:

- **Selection:** the restriction of the returned tuples to a subset defined by a condition (also called filter). A term often used in context with selection is selectivity. It is a measure of the size of the returned subset of tuples. High selectivity means that the condition applies to only a small number of tuples, which are returned. Low selectivity, in contrast, means that the condition is applicable to many tuples, which are then returned in the result set.
- **Projection:** returns a specified subset of attributes from a table or result set for the selected tuples.
- **Joins:** a conjunction of one table with itself or between different tables based on conditions defined on attributes.
- **Grouping:** collocation of tuples in the result set based on the values of specified attributes.
- **Aggregation:** summary of specified attributes according to a given function, such as, average or summation, and a given grouping.
- **Sorting:** ordering of tuples in the result set according to the values of specified attributes.

Entry point for data retrieval can be any table defined in the database. Navigation is possible through any compatible pair of attributes of one table or different tables rather than through predefined sets linking the tables.

Discussion of Data Models in Transaction Processing

Similar to file system models, the early hierarchical and network data models and applications relying on them were connected too closely. Data was manipulated via standard host programming languages. As a result, changes to the data storage in order to leverage latest advances in hardware technology induced changes in the applications to preserve functionality. This process endangered a companies investment in existing applications, that would have to be changed over and over again, causing additional costs [84].

Hierarchical models are very efficient for large amounts of data and provide high transaction throughput, but do not allow very flexible relations and require extensive application programming to use the database. Network models are more flexible regarding data access paths and many-to-many relationships are easier to implement than in the hierarchical model, but their structure can easily become very complicated [151, Chap. 20]. In both, network as well as hierarchical models, the database structure has to be designed around the access paths that applications use most often. Logical relationships within hierarchical databases provide the possibility to define additional access paths if needed, incurring some overhead through extra pointers, but reducing redundancy.

In contrast, a relational database is not restricted to specific application access paths, but rather provides the means to model a data set in a logical database design on which applications can work using a high-level query language. Through high level query languages, applications are completely decoupled from the physical representation of data on the storage medium [38]. Consequently, they are not affected any longer by changes in physical database design caused by advancements in hardware technology. The foundation for this independence from hardware changes is the complete specification of the relational model including its operators on top of which the high level interfaces are defined. The operators defined with the relational model provide access capabilities for single entity instances as well as for multiple instances at a time (called set processing). In the hierarchical and network models, set processing in the sense of mathematical sets had to be handled by the application programmers, who were forced to implement iterative loops to navigate through the data structures and collect all data [42]. An example query is the *selection of the combined value of all sales orders in January*. Some transaction processing applications, for example, dunning, and especially analytical use cases rely on set processing, though.

After the development of RDBMS, new database systems called object databases emerged. They targeted the encapsulation of structure and behavior along the lines of object-oriented programming. The object data model is not discussed in more detail as object databases are less relevant for business systems but rather established niche markets for application areas that rely on the management of complex objects. In response to the development of object databases, RDBMS vendors added extensions to their systems. Object-relational DBMS were the result, offering similar functionality as the object databases on top of a relational model, for example, access to the relational database through object-oriented routines using the given standard interfaces (e.g., SQL). Object databases have established a market for applications with special requirements directly benefiting from the functionality offered by object databases, such as, computer-aided design, telecommunication systems, or geographic information systems [53, Chap. 1]. Yet, they never set up a strong foothold in enterprise data management [12].

Some database vendors optimized their data models for specific applications in the 1980s to minimize computing resources [84]. An example was IBM's IMS/VS 1.1.4 fast path feature for critical banking applications that increased throughput by trading generality for specialized application program scheduling and data storage techniques [138]. Restrictions included, for example, no support for inserts in transaction mode or only providing one data access path [84]. Today, RDBMs are widely accepted and used in transaction processing scenarios [92, Chap. 1]. Database systems based on the other models are still in use, however, for special application areas that pose requirements that directly exploit their advantages. For example, IBM IMS with its hierarchical database model allows for a very high transaction throughput in settings that require complex data structures with many hierarchy levels. Thus, use cases for the IMS database are found in the finance, retail, and telecommunications industries, where management of highly complex records and high transaction throughputs are a essential [155].

Independently and in parallel with the development of hybrid databases based on the relational approach, another movement has originated that promotes a new kind of database systems that follow a non-relational approach. It is called NoSQL, standing for “not only SQL” or “not relational”. According to Cattell [24], these systems sacrifice some dimensions such as consistency, durability, availability, or query support to achieve others, e.g., higher availability and scalability. Increasingly variable document types and scalability issues in Web applications have created a push for NoSQL in the Web developer community [127]. Yet, their application in the enterprise environment is questionable. Stonebraker [198] states that NoSQL databases are most often considered for update- and lookup intensive OLTP workloads and not query intensive analytical workloads. This covers only a part of today’s enterprise workloads.

While early data management systems possessed transaction processing functionality as well as a wide range of decision support tools, developments in the late 1990s have resulted in an optimization of systems according to either the OLTP or the OLAP application area. Specialized data models that emerged to support analytical functionality efficiently will be discussed in the following.

2.1.2 Analytical Processing Systems

As transactional database systems became more sophisticated and stored more data, analysis needs also became more demanding. The extensive supply of data being stored in the transactional database was expected to be available for analysis. Database manufacturers in the mid 1990s attempted to address the transactional as well as the analytical side of business and even simple analytical queries took hours to run, creating the problem of “Too Much Data, Not Enough Information” [61]. Another challenge that systems providing analytical functionality face is the rate at which the needs and accordingly the requirements change. Since markets are becoming increasingly fast-paced, enterprises have to plan and react similarly fast, demanding the flexibility to serve new reporting needs, and include as much and as detailed data as possible, for both historical and up-to-date data analyses.

Dimensional Modeling in Analytical Processing

In contrast to transaction processing with insert, select and update behavior, analytical query processing can be classified as a read-only workload considering the actual user interaction. Bulk inserts to update data are executed in the background or during low system load times to avoid affecting user interaction performance. Queries are composed of complex data selection. The following query exemplifies where the complexity in the selection lies:

Display the number of purchases per month, product category, and city for the first quarter of 2011.

For this query, the entire set of sales data for the first quarter of the year 2011 is scanned to compute the number of purchases. In a typical database design for transaction processing based on the relational model, this would entail several joins, for example, between sales order header and item tables, product data tables, and tables containing region data. Especially the join of sales header and item tables is expensive because these are tables where transaction data is collected resulting in millions of entries according to business throughput. The dimensional modeling approach [119] typically used in analytical database design avoids joins between large tables and also reduces the number of joins compared to a normalized transactional database design.

In the dimensional modeling approach, data is separated into dimensions and facts. This separation stems from the classification of data into transaction and master data. Transaction data is accumulated during daily operations of a company and includes sales orders, payments, purchases, and production data to name just a few. Master data is changed only infrequently [221] compared to transaction data, that is changed with every business transaction. Examples for master data are customer, supplier, product, or location data.

Dimensions contain master data clustered into groups, for example all product specific data is stored in the product dimension, or time data in the time dimension [119, Chap. 1]. Dimensions are used for filtering and for creating the context by means of which facts are aggregated according to selections within the dimensions. Facts are numeric values from the transaction data [119, Chap. 1].

In the example query above, dimensions are *Time*, *Product*, and *Location*. The fact to be aggregated is *number of purchases*. The time dimension is used for filtering (*first quarter of 2011*) and the granularity level to aggregate the fact on is *month* of the time dimension, *category* of the product dimension, and *city* of the location dimension. Figure 2.8 shows an extract of the reporting result as a table for this query. The level of granularity defines how much detail is provided about the composition of the fact aggregate. Aggregating by product category and brand would increase the level of detail, whereas the aggregation by country would remove details from the report.

Typical operations based on the dimensional model include roll-up, drill-down, slice-and-dice, and pivot [30]. The names for these operations stem from imagining the dimensional data structure as a cube (in the case of three dimensions). Figure 2.9a shows the cube for the example report given in Fig. 2.8. The roll-up operation decreases the level of granularity, for example, showing the number of purchased products per quarter instead of per month (see Fig. 2.9b). Drill-down is an operation in the opposite direction, increasing the level of detail, for example, products per week, instead of per month. Slice-and-dice restricts the result set or relaxes restrictions on the result set according to further criteria on dimensions, for example, only showing the top ten products sold in January (slicing operation, see Fig. 2.9c), or only showing certain cities and categories (dicing operation, see Fig. 2.9d). Pivoting is the operation of rearranging the multidimensional view of the data. It is a table operation, for example, exchanging the columns and rows.

		Month			Total
City	Category	01/11	02/11	03/11	
Hanover	Pliers	2	9	4	15
	Clamps	13	15	12	40
	Drills	9	7	11	27
	Subtotal	24	31	27	82
Munich	Pliers	23	21	43	87
	Clamps	31	29	35	95
	Drills	54	97	67	218
	Subtotal	108	147	145	400
Berlin	Pliers	15	7	12	34
	Clamps	25	37	22	84
	Drills	23	19	15	57
	Subtotal	63	63	49	175
Total		195	241	221	657

Fig. 2.8 Reporting result for the number of purchases per city, category, and month

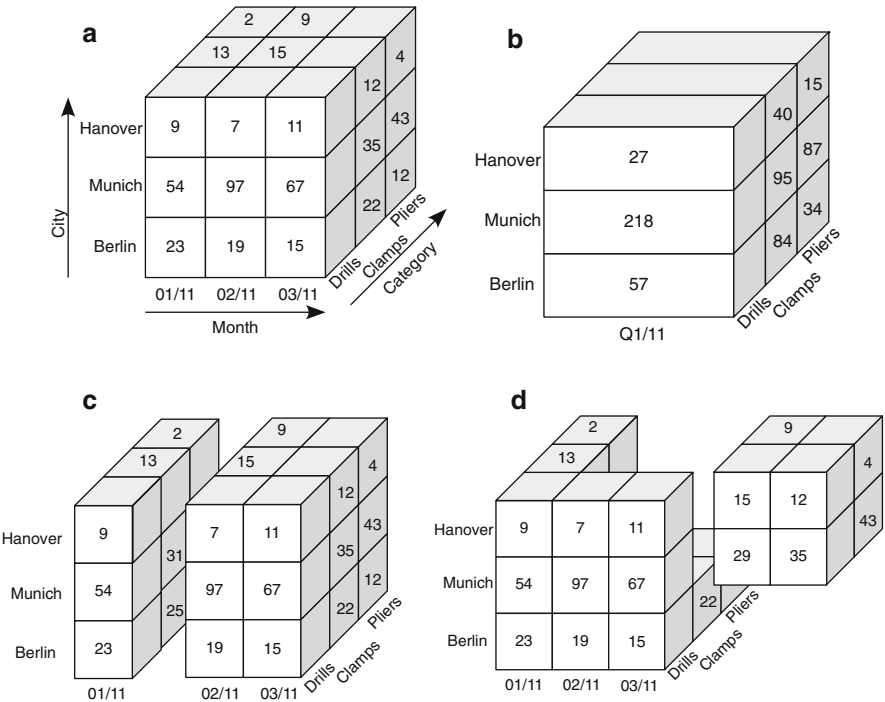


Fig. 2.9 Cube and operations for the number of purchases per city, category, month. (a) Cube structure. (b) Roll-up operation. (c) Slicing operation. (d) Dicing operation

From an application usage perspective, transactional and analytical workloads differ largely. Transactional applications automate clerical data processing tasks. These are structured and repetitive in their nature [30] and are concerned with single or a small number of instances of an entity, for example, the sales order of a specific customer and its associated line items. Transactions require up-to-date data. It is essential that changes once committed are preserved, and that they are not restricted by throughput or other resource scheduling constraints. Failures of customer transactions, for example, due to throughput limitations can have a direct impact on a company's revenue and, thus, have to be avoided.

Analytical systems support employees, e.g. executives, managers, and analysts to make better and faster decisions [30]. Much more data is accessed in one report compared to a transaction, especially including historical data, like sales data of a specific region for the entire past year. As analytical reports include many dimensions by means of which facts are aggregated, the underlying queries can become very complex, that is, contain many join, grouping, and sorting operations. Using optimized multidimensional data structures to store data already prepared for analytical needs reduces this complexity. Such data structures will be discussed in the following. The nature of analytics lies in retrieving data from the system, interpreting the results, and inquiring the reasons for unexpected results. Because any inquiry is conceivable, a considerable share of analytical queries is composed of ad-hoc queries. In many cases, they emerge from repetitive reports, for example, from questioning irregularities in the results of a report to create the annual financial statement.

Analytical Reporting Categories

Different categories of analytical reporting exist. Parameters for categorization are the period of source data that is taken into account, the business focus for which the analysis results are needed, or the primary users of the reports. Inmon [104] introduces two categories of reporting: operational and informational. Operational reporting analyzes the detailed transaction data of an enterprise with up-to-the-second accuracy. Inmon names daily production records, or flight-by-flight traveler logs as examples. The goal of informational reporting, in contrast, is strategic analyses and longer-term decisions. Therefore, informational reporting focuses mainly on summary data, such as, monthly sales trends or annual revenue by region, and less on details. The main difference between the two is the freshness of the data analyzed and the time window sizes that are analyzed. For the operational side this means maximum freshness and minimal time slices (hours, days). For the informational side it means less fresh data and longer periods to analyze, e.g. months, quarters, or years.

In contrast to Inmon, White [220] distinguishes three categories of reporting: strategic, tactical, and operational. Strategic reporting supports the execution of long-term business plans and measuring the progress toward organizational goals, such as growing market share or increasing revenues, based on high-level business

performance metrics. A simple, but typical example for such a metric in the context of sales order processing would be setting target net sales revenues per region per quarter and monitoring their fulfillment in comparison to actual sales numbers. The aim of tactical reporting is to monitor business initiatives, such as marketing campaigns that are set up to ensure reaching the long-term goals. Tactical reporting analyzes business operations within a time window of days, weeks, or months. Compared to Inmon, strategic and tactical reporting are subcategories of informational reporting, based on historical data, analyzing time frames of days to weeks to months (tactical reporting) and months to years (strategic reporting). Operational reporting, according to White and similar to Inmon is concerned with daily business and is based on intra-day analyses. Credit card fraud detection and inventory management are typical examples of operational reporting.

Data Storage for Analytical Processing

In the late 1990s, special OLAP systems evolved that solely focused on providing decision makers with information. These systems organize data according to the usage requirements of decision makers using a multidimensional paradigm suited to model the natural structure of decision support problems [54].

According to the implementation of their data storage, analytical processing engines can be distinguished into two basic types: These are relational OLAP (ROLAP) and multidimensional OLAP (MOLAP) engines [32]. MOLAP engines store multidimensional data directly in spatial data structures, for example arrays. Queries from the front end are directly mapped to these special data structures. In ROLAP engines, data is stored in a relational database with the engine implementing efficient algorithms for analytical operations that translate between the above logical multidimensional model and the relational storage model below. MOLAP engines provide exceptional performance in accessing data since the dimensional value combination of a fact is implicitly given by its address. However, as many value combinations may not exist, for example, some products are not sold in some regions resulting in empty values for facts in the associated array cells, MOLAP structures can become sparse leading to a poor utilization of storage space [30]. ROLAP does not suffer from this problem. If facts do not exist for a combination of dimension values, nothing is stored. Consequently, hybrid OLAP (HOLAP) approaches have been introduced to utilize the advantages of both storage techniques. Dense regions of a cube could be stored using MOLAP and sparse regions using ROLAP. Normalization of a data cube, that is choosing an appropriate ordering of attribute values (dimension values), aims at distributing data of a cube into dense and sparse regions, thereby increasing storage efficiency [115].

Relational databases are still the dominant choice as the underlying technology for analytical processing systems [139]. Thus, this work will further focus on ROLAP engines, instead of MOLAP and hybrid approaches.

Data Warehouse, Data Mart and Operational Data Store

A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data to support decision-making [108, Chap. 2]. Instead of clustering data according to the processes like in operational systems, data is structured according to the areas relevant for decision-making, leading to subject-orientation. For a retailer, subject areas of interest may be sales, products, or vendors. Integration means that data from several sources, for example, operational systems and external services, is collected and provided in a single source. The nonvolatile characteristic means that data already in the data warehouse is never updated. New data is added in snapshots, resulting in a historical record of the operational data. Time-variance in this context means that each data element is valid as of a certain point in time. Data records may be given a time stamp or validity interval to determine their validity period.

To create and update the data warehouse ETL tools are used. They transform the source data to fit the analytical business needs and support loading of the transformed data into the warehouse [214].

In many industries data warehouses are successfully employed. Their use cases include order shipment and customer support in manufacturing, user profiling and inventory management in the retail industry, claims, risk, and credit card analysis in the financial sector, call analysis and fraud detection in telecommunications to name but a few [30]. Although the idea is that a company has one data warehouse that is managed centrally as a source for all company-wide reporting needs, it is rarely achieved. Because of, for example, acquisitions and mergers the system landscape of companies is heterogeneous in the analytical as well as the operational domain.

In contrast to the data warehouse, the data mart contains customized data [101]. Inmon [102] defines a data mart as a data structure that is dedicated to serving the analytical needs of one department within a company. Data marts are differentiated into dependent and independent data marts. Dependent data marts are built from data coming from the data warehouse. Independent data marts are built from data coming from the operational systems. They are inexpensive to build, as organizational groups can collect their requirements and set up a data mart based on data in the operational systems without consulting other organizational groups. Inmon states that the drawback of this inexpensive setup of independent data marts can be their uncontrolled growth in the long term. For example, each department tries to service its reporting needs. Yet, none of the existing data marts of other departments fits their specific needs completely resulting in the creation of a new data mart. Thus, it can happen that the same detailed operational data is redundantly stored in many data marts that differ only slightly in their structure and context. Building dependent data marts requires a larger amount of foresight, as requirements of different groups have to be collected to build the basis for these data marts in the data warehouse. However, while independent data marts service instant reporting needs, dependent data marts provide a sound long-term foundation for information decisions.

The different approaches to designing a data warehouse should not be disregarded. Inmon's approach mentioned above is one option. Another option is Kimball's approach [109] to design a data warehouse as a collection of dimensionally modeled data marts. Yet, another approach is using independent data marts to design a data warehouse. This approach, however, is seen as inappropriate in the data warehouse community as ETL steps are repeated unnecessarily and they lack cross-department analysis and communication capabilities [114]. Kimball's approach achieves results quicker and simpler than Inmon's, but a common criticism is that it lacks enterprise-wide focus. Jukic [114] describes the different approaches and outcomes of Inmon's and Kimball's methodologies as a trade-off between extensiveness and power versus quickness and simplicity. A detailed comparison of the Inmon and Kimball approaches can be found in [21].

Another structure in the analytical landscape that is complementary to the data warehouse according to Inmon [108, Chap. 16] is the operational data store (ODS). It is a subject-oriented, integrated, volatile, current-valued, detailed-only collection of data to support a companies need of reporting on up-to-the-second, integrated, operational data [100]. Subject-orientation and integration are similarities between the data warehouse and the ODS. Differences lie in the freshness of data, the amount of data that is kept in the ODS, and that data in the ODS is updated by overwriting existing entries instead of adding another snapshot. Due to the updates, the ODS contains no historical data. In the ODS only detailed data is kept, whereas a data warehouse contains detailed and summary data. ODSs are categorized into different classes according to the length of their update intervals affecting the freshness of data they contain.

Figure 2.10 gives an example of how the different OLAP data stores can be associated and shows possible flows of data between them. Sources of data can be enterprise resource planning systems, files like spreadsheets, or external services to name just a few. The ETL process between the data sources and the data warehouse is not explicitly shown in this overview. A detailed comparison of data warehousing methodologies including all of the three mentioned analytical structured is given in [184].

Inmon [100] initially defined three classes of ODSs. Class I ODSs are kept synchronized with the operational systems they retrieve data from, so that data is available for reporting only seconds after it is inserted in the operational systems. Class II ODSs are updated periodically every hour or in similar time intervals. Updates to the operational systems are stored in an intermediate file, which is then loaded into the ODS. Class III ODSs are subject to the same process, but the time interval between data updates in the ODS can be 24 h or more. The business case for class II and III ODSs is the most common [100]. Operational costs for class I ODSs are much higher because of the immediate synchronization and business cases justifying this class of ODS are rare [100]. A fourth ODS type (class IV) was introduced later on. This class of ODS holds results of reports from the data warehouse [105]. The rationale for the creation of this ODS class was that the data warehouse did not provide responses in real-time. The Class IV ODS is able to do so

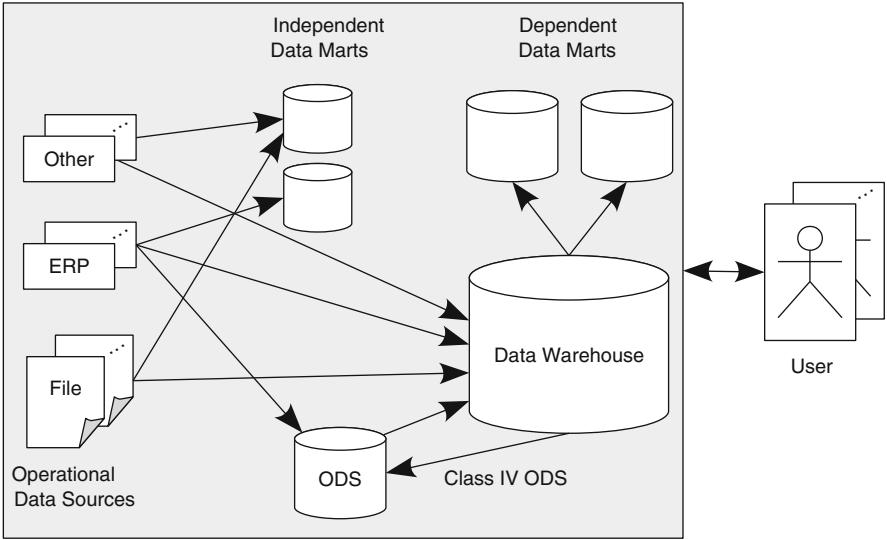


Fig. 2.10 OLAP data stores and data flow in between

for a limited number of prepared reports, thus trading off fast response times against freshness of data [106].

2.2 Relational Database Design

Database design incorporates the definition of the involved business entities, a database schema, and additional structures to accelerate access to data. It is divided into logical and physical database design.

[T]hink of the logical database design as the architectural blueprints and the physical database implementation as the completed home. – Hernandez [92, p. 29], 2003

Figure 2.11 presents the steps taken in the process of database design. Logical database design starts with the creation of the conceptual model. It specifies what data is included from the business perspective and which relationships exist between business entities [204]. The next step is the creation of the database-specific implementation model, that is, a database schema. While a data model is a set of mechanisms to structure data, a database schema is the definition of the structure of a specific data set with the help of a data model. The database schema as the logical design of the database has to be differentiated from the database instance, which is a snapshot of the data in the database at a given point in time [191, Chap. 2]. For relational databases, the database schema includes the definition of tables and the

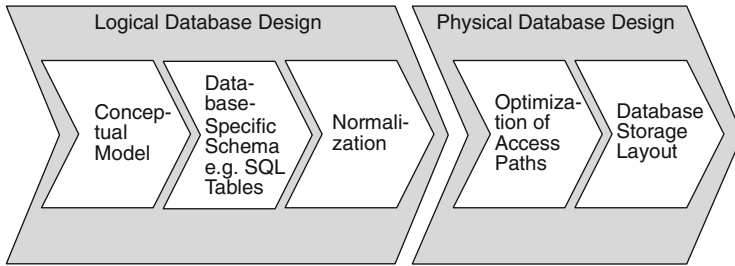


Fig. 2.11 Database design steps

normalization of those tables corresponding to the semantics of data relationships, which will be discussed in Sects. 2.2.1 and 2.2.2.

In 1980, Teorey and Fry [203] have classified logical database design into *Information Structure Design* (ISD) and *Information Structure Refinement* (ISR). Independent of the data model to be used, information structure design gathers all application requirements into a preliminary high-level schema. This phase results in entities, associated attributes that hold detailed information about the entities, and relationships between entities. In the ISR phase, the results of the ISD phase are further developed into a database-processable schema, for example, SQL table definitions. This basic distinction has remained valid in later work. Only a change of naming has taken place from ISD to conceptual modeling and ISR to comprise the creation of the database-specific implementation model, for example, the relational database schema.

Physical database design includes the definition of further access methods to optimize data access, that is, indexes and views, partitioning of tables, and data clustering. This amounts to structuring data with respect to specific machines and optimization of performance regarding a particular workload. Physical database design structures are presented in Sect. 2.2.3. Section 2.2.4 gives an overview of database-specific storage alternatives that have a strong impact on transactional and analytical processing.

2.2.1 Relational Database Schemas in Transaction and Analytical Processing

Transactional relational database schemas follow the principles of normalization. These were developed to achieve a database design with little or no redundancy to avoid false relationships and inconsistencies [84]. In [39], Codd introduces the objectives of normalization, that include obtaining a powerful retrieval capability by simplifying the set of relations. Thus, undesirable insertion, update, and deletion anomalies are removed. Additionally, the need for restructuring when adding new types of data is reduced, which increases the life span of applications.

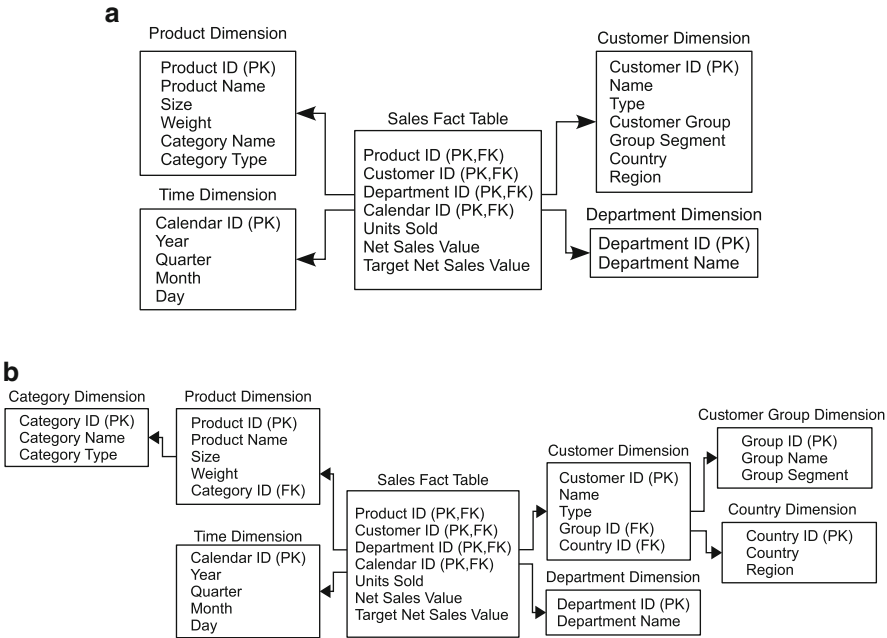


Fig. 2.12 Example ROLAP structures for the sales order example. (a) Star schema. (b) Snowflake schema

Analytical relational database schemas follow the dimensional modeling approach and the according relational database schemas are the star schema and the snowflake schema [54]. Figure 2.12 shows the star and snowflake schemas according to the sales order example. The star schema (see Fig. 2.12a) is composed of a central fact table and multiple dimension tables. The fact table contains data from business transactions or a snapshot summary of that data. It is connected through a many-to-one relationship to each dimension table.

To speed up joins between the dimensions and the fact table, which is relevant for every OLAP query, join and bitmap indexes are used [123]. The snowflake schema is an extension to the star schema (see Fig. 2.12b) to model hierarchies explicitly, thus normalizing the dimensions. Thereby, redundant data storage within the dimensions is reduced, but additional joins are needed if data from the outer dimensions is required.

Even normalized schemas, for example, in third normal form (3NF) [39] are employed in some analytical use cases, backing away from the dimensional model. The different analytical database schemas are used in different kinds of data stores that exist in the OLAP environment to cater for specific reporting needs. Martyn [137] states that it is reasonable to use the star schema in a data mart, the snowflake schema in a data warehouse and the 3NF schema in an ODS.

2.2.2 Normalization

While reducing redundancy and avoiding false relationships and inconsistencies, an increased level of normalization of a set of relations entails a penalty towards data retrieval. Data that could have been retrieved from one tuple in a denormalized design may have to be retrieved from several tuples in a normalized design. Kent [118] acknowledges, that the highest level of normalization does not need to be enforced, where performance requirements have to be taken into account. Mullins [146, Chap. 12] advocates normalization by pointing out that a normalized schema should never be denormalized unless a performance need arises, which cannot be solved in any other way.

Denormalization can be observed in the data schemas of productive systems for analytical processing as well as in transaction processing systems. Bock and Schrage [13] give an overview of denormalization techniques that are utilized in transaction processing schemas. Likewise, Mullins [146] discusses different types of denormalization and describes where they can be useful. Of these, the types *redundant data*, *repeating groups*, and *derivable data* can still be observed in today's transactional systems, mostly however because of legacy issues. Redundant data means including additional columns of another table in one table if they are always queried. This is only advisable if the included columns contain data that does not change frequently. Repeating groups comprises adding more columns for an attribute that can have multiple, but a limited and small number of values, instead of normalizing it into an own table with a foreign key relationship. A typical example is storing up to three telephone numbers for a customer. Thus, three *telephone number* columns are used instead of three rows in a second table. Derivable data comprises precomputing data and storing it in a column. An example is an extra column storing the *net sales value* of an order instead of aggregating it from the order line items each time it is requested.

In analytical systems, *pre-joined tables* and *report tables* are common. In pre-joined tables, as the name says, two or more tables are stored in their joined form, omitting redundant columns. Report tables are tables that already represent the report based on the data that is otherwise stored in several tables and can only be retrieved via complex SQL statements. Thus, the report results can be obtained using simple SQL queries. The star and snowflake schemas are members of this category. According to Martyn [137], the denormalization of OLAP schemas, that is, the star schema in the data warehouse, is acceptable because of the read-only character of the data warehouse and the opportunity of addressing potential update anomalies during the ETL process. Furthermore, for data warehouses where query performance is paramount, the snowflake schema is generally not recommended because of the reduction in query performance due to additional joins [171, Chap. 11]. Kimball and Ross also write that the “use of normalized modeling in the data warehouse presentation area defeats the whole purpose of data warehousing, namely, intuitive and high-performance retrieval of data.” [119, p. 11] Date [51, Chap. 7] views denormalization in a critical fashion and believes “that anything less

than a fully normalized design is strongly contraindicated” and that denormalization should only be used as a last resort if all other strategies to improve performance fail.

2.2.3 *Physical Database Design*

Physical database design is the technical specification of tables and the definition of additional structures to support query processing. The technical specification includes table names, column names, data types, primary keys, and foreign keys. Agrawal et al. [3] introduce physical design structures as any access path that is supported by a database server. These include indexes, views, (multidimensional) clustering, and partitioning of tables.

Indexes

Indexes provide alternative access paths to data items besides sequentially scanning a table and searching for a specific value. At the cost of write performance and storage space, indexes, thus, speed up data retrieval. Typical index structures are B-Tree, hash, and bitmap indexes with B-Tree being the most common [172, Chap. 8]. An index can be built on one or more columns and a table can have any number of indexes. The trade-off between index maintenance and fast data access in an ever-changing environment has to be considered. Some database engines support storing tables directly in an index structure, for example, Oracle8i index-organized tables [192]. The users or applications are typically not aware of indexes as the query optimizer decides if an existing index is used for a query or not.

Views

Views are defined as virtual relations [46] or single-relation images of queries [135]. Views are dynamically created through a stored query from the underlying data when required and may span one or more relations providing a subset of the entire data. A typical use case for views is security, where specific users should only have access to a subset of data, e.g. employees may view their own contract data, but not contract data of their colleagues whereas the manager has access to all contract data of his staff. In this case, users only have access to their dedicated view instead of to the tables the view is based on.

Special cases of views are materialized views. Gupta and Mumick [82] describe the materialized view as a view with its tuples being stored in the database. Thus, a materialized view is like a cache that stores a copy of data or derived data for quick access. A typical use case for materialized views in the OLTP as well as the OLAP environment is to store pre-computed data [189]. A typical use case in the OLAP environment is to efficiently implement the multidimensional structures for

analytical cubes [87] or the aforementioned report tables. Depending on their type, materialized views are updated upon changes to the base tables, which introduces overhead to the write operation. The overhead of the update process, also called view maintenance, should not outweigh the benefits achieved by creating and using the materialized view.

Clustering

Clustering of tables is another approach to improve data retrieval for a particular use case. Similar to storing a table in an index-like structure as described above, clustering influences the actual storage of a table. Records with the same value for a chosen attribute (or expression) are stored consecutively. Using more than one attribute as the basis for clustering is called multi-dimensional clustering [161]. In this case, the chosen attributes have to be orthogonal to each other to allow the creation of clusters. According to Lightstone and Bhattacharjee [128], multi-dimensional clustering is a powerful technique that offers significant performance benefits, especially for OLAP systems. OLAP queries directly profit from reduced I/O and CPU costs through their multidimensional nature.

Partitioning

Partitioning refers to cutting a table into pieces: either horizontally into subsets of complete tuples or vertically into subsets of complete columns. Horizontal partitioning has been introduced to improve the manageability for tables with many tuples. If partitions of a table are located on different machines, an additional benefit is increased availability as only part of a table becomes unavailable if a machine fails [57]. Vertical partitioning improves the performance of read access queries that need only a small set of the columns of a table. For databases that use a row-oriented physical storage layout partitioning according to query access reduces the overhead of reading columns that are not necessary for a query.

All of the above mentioned physical database design alternatives have been subject to research regarding the automation of design decisions. The question of how to determine an optimal storage scheme reaches back almost as far as the development of the relational model itself. Choenni et al. [36] define an optimal storage scheme as “the storage scheme which has the lowest cost in processing the workload defined on the database.” Because of the exponential complexity of determining an optimal storage scheme, research has shifted to determining a “good” storage scheme. A storage scheme is defined as “good” if an experienced human database designer would produce the same or a worse scheme with the same information content [36].

A variety of prototypes and tools exist that propose configurations for indexes and materialized views, for example, AutoAdmin for Microsoft SQL Server [31], or that automate partitioning, see Autopart for large scientific databases [162]. Zilio

et al. [227] introduced DB2 Design Advisor, which takes into account all of the above aspects of physical database design.

2.2.4 Database Storage

In addition to physical database design, aspects related to the data storage of relational databases influence query performance. Data storage is specified by where and how the data is stored. Aspects are the medium where the primary data set is located and the resulting data transport path to the processor as well as the layout of data in the primary location. The primary data location is the space where data is stored. During query processing, data is read from the primary location if it is not cached in a different possibly faster location. If changes occurred, data is written back to the primary location or is cached and written back later. Today's research and productive OLTP and OLAP database systems can be categorized into disk-resident and in-memory databases (IMDBs) [68] referring to the primary data location and two major storage layouts, the row-oriented and column-oriented storage layout, have evolved [2].

Disk-Based versus In-Memory Data Storage

IMDBs are less complex than disk-resident databases the main reason being that disk I/O as a mechanical process is avoided [71]. On disk, data is stored in blocks. If data is requested in a query, the block(s) containing the data need to be loaded from disk into main memory from where further processing takes place. Blocks that are read from disk are buffered temporarily in main memory. As a result, subsequent accesses to data already in memory avoid accessing the disk, thus speeding up operations. Other strategies are in place to avoid or reduce disk I/O. For example, if a block is accessed, other blocks that are likely to be accessed in the future can be pre-fetched from disk. Thus, subsequent queries for data from these blocks do not require any disk access.

Simply avoiding the disk access step by putting a disk-resident database into main memory can speed up operations. Yet, this strategy does not lead to the same speedup a pure IMDB achieves [68]. In the case of using a faster primary storage, such as RAM disks, the reason for the lesser speedup is that the algorithms tuned for disk access are still in place. For example, data is still accessed as though it was located on disk and is copied to a main memory area for further processing. This step is superfluous as the database already resides in main memory, but the entire process is faster nonetheless, as data is loaded from main memory and not from disk.

In the case of a completely cached disk-based database, data resides in main memory in the same structure as if it was just loaded from disk, that is, in blocks. However, data structures optimized for in-memory accesses instead of disk accesses achieve further improvement fully cached disk-based databases do not benefit from

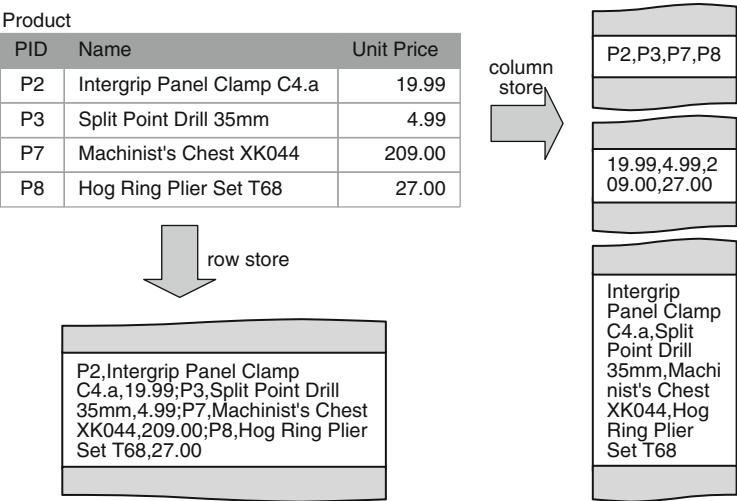


Fig. 2.13 Example for the layout of data in row and column stores

without modification. An example is the storage of data consecutively instead of in blocks. In IMDBs, logs are kept in non-volatile storage, such as, flash storage. Snapshots of the database are also written to this non-volatile storage in regular intervals to avoid loss of data.

Row versus Column-Orientation

The records of a table are stored consecutively in row-orientation. In column-orientation, the attribute values of a column are stored consecutively, meaning that each column of a table is stored separately. Figure 2.13 shows the *Product* table from the customer order example from Fig. 2.7 in row and column-oriented storage.

The benefit of row-orientation lies in operations such as adding new tuples to a table and look-ups of a set of attributes of a single tuple or a small number of tuples. Adding an entire tuple in a row store entails an address look-up for the location to store the record and a sequential write of the record data. In a column store an address look-up and a write is needed for each attribute value of the record [89]. The behavior is similar during a look-up of a set of attributes from a tuple or from a small number of tuples. The set of requested attributes for each tuple has to be reconstructed from the separately stored columns in a column store, whereas they can simply be read sequentially in a row store. Column stores benefit operations that access only a subset of columns from a wide table over a large set of tuples [1]. Examples of such behavior are summary operations such as *displaying the number of purchases per month, product category and city for the first quarter of 2011*, which are typical for analytical applications. In this operation, only the columns actually needed have to be transmitted to the CPU for processing in a column

store [2]. In the row store, data from columns that are not needed is read if the requested attributes are not stored consecutively in the record. Reading data in chunks the size of the CPU cache lines that are usually larger than single attribute values causes this.

Column stores are advantageous in scenarios where the schema has to adapt to changing requirements, that is, adding a new column to an existing table or changing a column, for example to fit larger values. Columns can be added to a table or existing columns can be changed, for example increasing the length of a text field, without having to touch data in the other columns. In contrast, increasing the size of an existing column or inserting a new column in a row store requires rearranging the storage of the entire table to adapt to the changed record.

Compression is another aspect where column stores prevail over row stores. The values in a column are of the same data type, which is beneficial for compression. Additionally, Krueger et al. [124] have shown in their analyses of enterprise systems that frequent occurrences of the same value are common for the majority of columns in a table. As a result, a better compression ratio in column stores is possible [1]. This facilitates (a) better utilization of bandwidth constraints when transporting data to the processor if results can be computed on compressed data or data is decompressed as late as possible and (b) higher utilization of storage space, which is especially relevant for IMDBs as memory space is still more expensive than disk space.

Until recently, the vast majority of commercial databases used for transaction processing as well as analytical processing followed the row-oriented storage model. By now, column stores and hybrid engines that offer column and row-oriented table layouts have taken hold in the OLAP domain (cf. Feinberg and Beyer [59]).

2.3 Summary

In this chapter, the basic concepts underlying transactional and analytical processing from the data model and database design perspectives were provided. Several data models to support transaction processing were highlighted. Experiences from different data models have led to the development of the relational model and relational databases with it, which are today's most prevalent model in enterprise data management for transactional as well as analytical purposes. According to Lindsay, "their [relational databases] adoption in business, government, and education has enabled the progress we've made in productivity and without these tools to manage the complex affairs of government, business, education, and science our progress would be really much slower." [223] As has been discussed in this chapter, relational databases are the dominant choice for OLTP as well as OLAP. Thus, the remainder of this work focuses on relational databases as the foundation for OLTP and OLAP. Relational database design variants and optimization strategies commonly applied in OLTP and OLAP have been presented and discussed.

**Benchmarking Transaction and Analytical Processing
Systems**

The Creation of a Mixed Workload Benchmark and its
Application

Bog, A.

2014, XIII, 164 p., Hardcover

ISBN: 978-3-642-38069-3