The $C^2P$ algorithm consists of two main phases. The first phase (Phase I) organizes a set of objects into a number of sub-clusters, which are an effective representation of the final clusters. The Self Semi-Closest-Pair Query (Self-Semi-CPQ) is used to find pairs of objects $(p, p')$ that belong to a data set $S$ such as $\text{dist}(p, p') = \min_{x \in S} \{\text{dist}(p, x)\}$. The algorithm uses a graph representation that organizes the proximity information computed by the CPQ. Using the Depth-First Search the algorithm can efficiently find the $c$ connected components of the graph, which also comprises the sub-clusters of the data set. All objects that belong to the same connected component can be considered as a sub-cluster. When the number of defined sub-clusters, is equal to the required number of sub-clusters the Phase I terminates. Otherwise, the algorithm finds the center of each sub-cluster to represent it. Then the previously described procedure is iteratively applied to the set of $c$ cluster centers until the required number of sub-clusters is defined. The second phase is a specialization of the first phase using a different cluster representation so as to produce the finer final clustering. The second phase (Phase II) has as input the centers of sub-clusters defined in Phase I. At each iteration of Phase II, Self-CPQ finds the closest pair of clusters by finding the closest pair among their representatives. Then these two clusters are merged and the $r$ data objects, among all the objects of the merged clusters, that are closest to the cluster center are selected as representatives of the new cluster. Using multi-representatives instead of the center, $C^2P$ can effectively capture the shape and size of the clusters. The procedure terminates when the required number of clusters is reached. The above description shows that Phase II operates in a fashion analogous to a hierarchical agglomerative clustering algorithm.

The $C^2P$ algorithm is shown to scale well to large databases. Its time complexity for a dataset with $n$ objects is $O(n \log n)$.

**Key Applications**

*Hierarchical clustering* has applications in various fields of real world. In biology, it can be used to define taxonomies, categorize genes with similar functionality and gain insights into structures inherent in populations. Clustering may help to automate the process of analyzing and understanding spatial data. It is used to identify and extract interesting characteristics and patterns that may exist in large spatial databases. Also hierarchical clustering is used to discover significant groups of documents on the Web's huge collection of semi-structured documents. This classification of Web documents assists in information discovery.

**Cross-references**

- Balanced Trees
- Clustering Methods
- Graph Clustering
- Indexing
- k-Nearest Neighbors
- R-Tree

**Recommended Reading**

records. Though this paradigm is fairly common in data structures, the term hierarchical model most generally refers to the IMS model, a proprietary DBMS developed by IBM from the sixties that is still widely used.

The IMS model organizes data in tree structures of records augmented with additional links that compensate for the weaknesses of this base model. Data processing itself is hierarchical, starting from the root of a record tree then parsing the dependent records in depth-first, left-to-right, traversal order.

**Historical Background**

In 1966, IBM started the development of ICS (Information Control System), a data management software for, and with the collaboration of, American Rockwell, then in charge of building the Apollo spacecraft which was to send men on the moon. IBM commercialized ICS in 1969 under the name IMS (Information Management System) [1].

The first version offered storage and processing facilities for data records organized in trees and linearized according to the depth-first traversal order. The data could be stored on tapes and on disks and were processed sequentially. Thanks to the addressability of magnetic discs, IMS was later given direct access (through hashing and B-tree techniques) to root records and therefore was able to support transaction processing.

At that time, tree structures were strictly independent and not connected, a fact that led to much redundancy (a record cannot be stored in two different trees without duplication). IBM then introduced logical relationship types which explicitly linked records from different tree structures. A record could then be shared by several trees.

Later on, IMS was added secondary indexes that provided direct access to non-root records. IMS is now a complex and powerful data management and data communication environment mostly used by data-intensive batch and On-Line Transaction Processing (OLTP) applications.

Though System 2000 (SAS Institute), XML document structures (DTD and XML Schema) and standard file structures can legitimately claim to belong to the hierarchical family, the presentation will focus on the IMS model. Data description and manipulation languages, API and implementation techniques will not be addressed in this chapter. They can be found in references [2, 4–6].

**Foundations**

Due to historical reasons, in particular the incremental development of IMS, the description of its data structures is generally intricate. In this entry, they will be described by way of a simpler approach based on graph theory. Of course, some very specific details will be ignored.

**Graphs and Hierarchies**

This section relates database schemas with various kinds of graphs. Due to the limited scope of this chapter, only an intuitive view of the equivalence principles will be developed. Is-a hierarchies, in particular, will be ignored.

**Preliminary Definitions**

A database schema can be seen as a multigraph, whose nodes denote entity types and whose labeled edges represent binary relationship types (rel-types for short). This applies to any Entity-relationship schema, provided n-ary rel-types have been replaced by relationship entity types through some kind of reification transformation (such as T1 in Fig. 1). In this graph, edges are given a pictorial representation that specifies the functional dependencies that hold in the rel-type according to the usual arrow convention. A single arrow denotes a N:1 rel-type, a double arrow a 1:1 rel-type and a plain edge a N:N rel-type. N:1 and 1:1 rel-types are called functional as well as the edges that represent them.

A hierarchical graph is a binary graph such that (i) the edges are functional and (ii) the edges define a partial order on the nodes. In practical words, there is no chain of successive N:1 edges the starting and arrival nodes of which are the same node (in short, no circuit). Considering a directed edge drawn from B (the many side) to A (the one side), A is called a parent of B while B is a child of A. A is called a k-node if it is the child of k parents. A graph is a n-hierarchy if none of its nodes has more than n parents. A 0-node is a root of its hierarchy. A hierarchy comprises at least one root. A node that is not a parent is a leaf node. A 1-hierarchy is a set of trees, that is, a forest.

**Properties of Hierarchies**

1. Any multigraph can be losslessly transformed into a hierarchy. Three patterns can prevent an arbitrary schema from defining a hierarchy, namely complex rel-types (n-ary, with attributes), N:N rel-types
and rel-types that form a circuit. N-ary and N:N rel-types can be processed by transformations T1 and T2 respectively (Fig. 1). Any functional rel-type of a circuit transformed through T3 or T4 (Fig. 1) destroys this circuit. All these entity-generating transformations have been proved to preserve the semantics of the schema [3]. A large number of hierarchies can be derived from a definite multigraph.

2. *Any hierarchy can be losslessly transformed into a 2-hierarchy.* Transformation T3 shows that B, which is the child of A in the source schema loses its parent (which becomes its "spouse") in the target schema. A similar transformation exists for 1:1 rel-types (T4). Iteratively applying this transformation produces an equivalent 2-hierarchy from an arbitrary hierarchy (Fig. 2). This process is not unique, so that many 2-hierarchies can be derived from a given hierarchy.

3. *Any 2-hierarchy can be built by superimposing two forests on the same set of nodes.* For each child node of a 2-hierarchy $H$ with two parents, one of the parental edges is colored in black and the other one in gray. Each of the remaining edges is arbitrarily colored in black or in gray. The set of nodes together with black (resp. gray) edges form the black (resp. gray) subgraph(s). Each of them is a forest and their union is $H$. There are many ways to color the edges of a given 2-hierarchy (Fig. 3).

4. *Any arbitrary Entity-relationship schema is equivalent to the superimposition of two forests.* Properties 1 and 2 show that any Entity-relationship schema is equivalent to a 2-hierarchy, which in turn can be decomposed into two forests.

**IMS Data Structures**

IMS is by far the most popular DBMS based on the hierarchical data model. This section describes its main data structures.

**The IMS Global Schema**

The global schema of an IMS database can be modeled by a 2-hierarchy formed by two distinct forests. The nodes represent *segment types* (the IMS name for record types) and the edges N:1 *parent-child rel-types*. The black rel-types are called *physical rel-types* and the

---

**Hierarchical Data Model. Figure 1.** Transforming arbitrary relationship types into functional relationship types.

**Hierarchical Data Model. Figure 2.** Reduction of a $n$-hierarchy to a $n-1$-hierarchy by application of transformation T3.
gray ones *logical rel-types* (this naming is purely historical and bears little meaning at this level). According to the *color* of the rel-type that links them, two segment types will be called respectively physical/logical child and physical/logical parent. A rel-type bears no name. If two rel-types link the same pair of segment types, one of them is physical and the other one is logical.

IMS schemas use graphical conventions that are close to those of Figs. 1–3 but their edges have no arrow heads (Fig. 4).

IMS imposes additional constraints of this double tree structure:

1. a logical child must be a physical child; therefore a root segment type cannot be a logical child;
2. a logical child cannot be a logical parent;
3. a logical child cannot have a physical child that is also a logical child.

As a consequence, many logical children are leaf physical segment types (Fig. 3 – right).

A segment type has a name and a length. It is made up of fields. A field is an untyped byte string with which a name, a starting position and a length are associated. Some parts of the segment type may be left undefined, while some fields may overlap, which is a way of describing compound fields. In summary, a segment type comprises single-valued mandatory fields that can be atomic or compound.

Now only the physical (black) rel-types are considered. Each 0-node denotes a *physical root segment type*. Each physical root segment type, together with all the segment types to which it is directly or transitively connected through a physical rel-type, form a *physical database*. A root segment followed by all its direct and indirect child segments, in the depth-first order traversal (or preorder), is a *physical record* of this database. A physical database contains a sequence of physical records. One of the fields of the root segment type is the *sequence field* or *key*. No two records can have the same key value and the records are sorted on this key. Direct access is allowed to a record based on its key value. A field of a dependent segment type can also be declared a key. In this case, and unless otherwise stated, the children segments of a definite

---

**Hierarchical Data Model. Figure 3.** Any multigraph is equivalent to the superimposition of two forests.

**Hierarchical Data Model. Figure 4.** Two examples of logical database schemas.
parent have distinct values of their key. The key value of a segment prefixed by the concatenated key of its parent forms its concatenated key. The latter identifies this segment in the physical database. The implicit goal of a physical database is twofold:

- historically, a physical record collects all the data that are needed by an application program (process-oriented approach);
- according to the modern way of perceiving a database, a physical record collects the proper data of a major entity of the application domain, independent of the programs that will use it (domain-oriented approach).

An IMS database comprises one or several physical databases together with all their logical rel-types. Logical rel-types can be defined between two segment types pertaining to one or two physical databases.

The IMS Logical Database

A logical database is a virtual tree data structure derived from an IMS global database schema and tailored to the specific needs of an application program. Its role is similar to that of CODASYL subschemas and of relational views. The main derivation rules are the following:

1. the logical database schema \( L \) comprises a connected subset of the physical database schema \( PH \);
2. the root of \( L \) is the root of \( PH \) (secondary indexes allow more flexible structures);
3. the logical parent \( P \) of a logical child \( C \) of \( L \) can be concatenated with \( C \), giving fictitious segment type \( CP \);
4. all or some of the direct and transitive physical children of \( P \) can be attached to \( CP \);
5. all or some of the direct and transitive physical parents of \( P \) can be attached to \( CP \) as fictitious children.

Fig. 4 shows some examples of logical database schemas derived from the same IMS database.

Additional Constructs

Several other concepts are part of the IMS model and are devoted large sections in usual IMS presentations. They have not been included in the preceding sections inasmuch as they do not contribute to the understanding of the data model but rather appear, according to today’s conceptual standards, as minor idiosyncrasies. Two of them are briefly described, namely segment pairing and secondary indexes.

Segment pairing. Consider the hierarchical schema fragment of Fig. 5 (left), in which relationship segment type \( R \) materializes a N:N link between \( A \) and \( B \). Though this pattern is intuitive from the conceptual and physical points of view, it violates the spirit of tree structures that underlies the hierarchical model of IMS: for bidirectional access, both \( A \) and \( B \) must have a normal (physical) child of their own. So, each of them is given a physical child, namely \( RA \) and \( RB \), and it is linked to the other parent through a logical (gray) rel-type. However, since \( RA \) and \( RB \) are just clones of each other, they are declared paired (Fig. 5 – right). Physically, this pattern can be implemented according to two techniques: with a single child segment type (virtual pairing) or with two redundant child segment types whose contents are synchronized automatically (physical pairing).

Secondary indexes. Physical (or logical) records can be directly accessed through indexing techniques applied to the root sequence field. A secondary index allows additional access based on other fields of the root segment type or access to non root segment types. It is implemented as a special-purpose physical database, the index database, made up of one segment type (the pointer segment type), that collects all the values of the key fields together with the addresses of the segments that include these values. In fact, this technique is based on two segment types in the indexed database. The indexed values are extracted from the source segment type while the access is performed on the target segment type. These segment types are generally the same but the former can be the child of the latter. For instance, a secondary index can allow access to \( CUSTOMER \) segments that have at least one \( ORDER \) child segment with a given date of order. In addition, a logical database can be built on an index database in such a way
that its logical root segment type is any, possibly non-root, segment type of the indexed database.

**Entity-Relationship to Hierarchical Mapping**

As shown above, an Entity-relationship schema can always be translated into an equivalent 2-hierarchy, a structure that is close to the IMS data model. The translation is mostly based on a unique technique, that is, the transformation of a rel-type into a relationship entity type together with two or several functional rel-types. The procedure of deriving an IMS database schema from an Entity-relationship schema can be sketched as follows.

1. Each entity type is represented by a segment type, each attribute by a field and each functional rel-type by a parent-child rel-type (still *uncolored*).
2. A non functional rel-type is transformed into a segment type by techniques T1 and T2.
3. Is-a hierarchies are best transformed through the *one segment type per entity type* technique; subtypes are represented by physical children of their supertype segment type.
4. A circuit is opened by the transformation of one of its links by technique T3 and T4.
5. 1:1 rel-types are implemented as standard 1:N rel-types and controlled by the application programs.
6. The schema is then transformed into a 2-hierarchy by the technique illustrated in Fig. 3. When a segment type has two parents, two techniques can be used. The first one consists in marking one rel-type as *logical*. The second one applies transformations T3 or T4. The latter will be preferred when IMS structural rules concerning logical rel-types are violated or to make the schema more balanced (rel-type RA of the schema of Fig. 3 could be processed in the same way as RB).

Figure 6 illustrates some of these principles. The IMS global schema comprises three physical databases and three logical relationship types. For readability, segment pairing, according to which each logical child is duplicated as a dependent of its respective logical parent, is not shown.

**Key Applications**

IMS is a major legacy technology in which many large corporate databases are still implemented. It is mainly used for stable, slowly evolving, batch and OLTP applications, notably in banking companies. The complexity of the hierarchical model and its lack of flexibility in evolving domains make IMS technology less attractive for decisional applications, such as data warehouses.

**Cross-references**

- Database Management System
- Entity-Relationship Model
- Network Data Model
- Relational Model

**Recommended Reading**

2. Elmasri R. and Navathe S. Fundamentals of Database Systems (3rd edn.). Addison-Wesley, 2000. (The appendix on the hierarchical data model has been removed from later editions but is now available on the authors’ site.)
Hierarchical Data Organization

Hierarchical Data Summarization

Definition
Given a set of records data summaries on different attributes are frequently produced in data management systems. Commonly used examples are the number of records that fall into a set of ranges of an attribute or the minimum values in these ranges. To improve the efficiency in accessing summaries at different resolutions or due to a direct need for investigating a hierarchy that is inherent to the data type, such as dates, hierarchical versions of data summaries can be used. A data structure or algorithm is labelled as hierarchical if that structure or algorithm uses the concept of subcomponents to systematically obtain conceptually larger components. The method of obtaining a larger component is regularly induced by the user’s understanding of the domain, such as dates in a year, as well as the fact that hierarchies can also be created automatically by a set of rules embedded into the system. Thus, rules used in a data structure’s creation, e.g., B+-trees, are also considered as a means for hierarchical data summarization. In fact, different variants of popular data structures are used in hierarchical data summarization. Various algorithms for data reduction and aggregation have also adopted hierarchical processing techniques.

Historical Background
From a data structures point of view, foundations of hierarchical data summarization (HDS) techniques can be found in indexing literature for databases. Although many of the indexing techniques, e.g., B+-trees, are used for efficiently selecting records stored on a disk, they can also be considered as hierarchical summaries on large amounts of data. For multidimensional and spatial data, indices such as R-trees and quadtrees can be used for HDS.

Today, many versions of popular indexing techniques that directly target retrieval of summary information exist. Some indices are also used in query optimization due to their HDS capabilities, e.g., using a space decomposition one can guess the number of records in a certain region of data before a join operation can take place. More recently, spatial indexing techniques, for example quadtrees, were developed for distributed settings such as sensor networks for HDS.

Historically, histograms are the most basic structures that could be used for data summarization. They are frequently utilized in query optimization decisions. They are also used in data warehousing. Hierarchical versions of histograms were recently built and are of interest for HDS.

From an algorithmic point of view, techniques such as wavelet transformations, sketches, and data clustering with aggregation, when run in a hierarchical fashion, can be considered as HDS techniques. These techniques are extensively deployed in data management as well as in other fields of computer science over many years.

In recent years, for distributed data processing, variants of known algorithms have become popular in HDS. For example, researchers have introduced data aggregation techniques on sensor networks that can be considered as HDS techniques. These techniques are extensively deployed in data management as well as in other fields of computer science over many years.

Foundations
B+-trees are frequently used in databases. A B+-tree is given in Fig. 1 (only some parts of the tree are shown to