## 3.1 Data Warehouse Implementation

Data warehouses contain huge volumes of data. OLAP servers demand that decision support queries be answered in the order of seconds. Therefore, it is crucial for data ware- house systems to support highly efficient cube computation techniques, access methods, and query processing techniques. In this section, we present an overview of methods for the efficient implementation of data warehouse systems.

### 3.1.1 Efficient Computation of Data Cubes

At the core of multidimensional data analysis is the efficient computation of aggregations across many sets of dimensions. In SQL terms, these aggregations are referred to as group-by's. Each group-by can be represented by a *cuboid*, where the set of group-by's forms a lattice of cuboids defining a data cube. In this section, we explore issues relating to the efficient computation of data cubes.

#### The compute cube Operator and the Curse of Dimensionality

One approach to cube computation extends SQL so as to include a compute cube oper- ator. The compute cube operator computes aggregates over all subsets of the dimensions specified in the operation. This can require excessive storage space, especially for large numbers of dimensions. We start with an intuitive look at what is involved in the efficient computation of data cubes.

**Example 3.11** **A data cube is a lattice of cuboids.** Suppose that you would like to create a data cube for *AllElectronics* sales that contains the following: *city, item, year*, and *sales in dollars*. You would like to be able to analyze the data, with queries such as the following:

- "*Compute the sum of sales, grouping by city and*
- *item.*" "*Compute the sum of sales, grouping by city.*"
- "*Compute the sum of sales, grouping by item.*"

What is the total number of cuboids, or group-by's, that can be computed for this data cube? Taking the three attributes, *city, item*, and *year*, as the dimensions for the data cube, and *sales_in_dollars* as the measure, the total number of cuboids, or group-by's, that can be computed for this data cube is $2^3 = 8$. The possible group-by's are the following: {(*city, item, year*), (*city, item*), (*city, year*), (*item, year*), (*city*), (*item*), (*year*), ()}, where () means that the group-by is empty (i.e., the dimensions are not grouped). These group-by's form a lattice of cuboids for the data cube, as shown in Figure 3.14. The **base cuboid** contains all three dimensions, *city, item*, and *year*. It can return the total sales for any combination of the three dimensions. The **apex cuboid**, or 0-D cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales. The base cuboid is the least generalized (most specific) of the cuboids. The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as all. If we start at the apex cuboid and explore downward in the lattice, this is equivalent to drilling down within the data cube. If we start at the base cuboid and explore upward, this is akin to rolling up.

An SQL query containing no group-by, such as "compute the sum of total sales," is a *zero-dimensional operation*. An SQL query containing one group-by, such as "compute the sum of sales, group by city," is a *one-dimensional operation*. A cube operator on $n$ dimensions is equivalent to a collection of group by statements, one for each subset

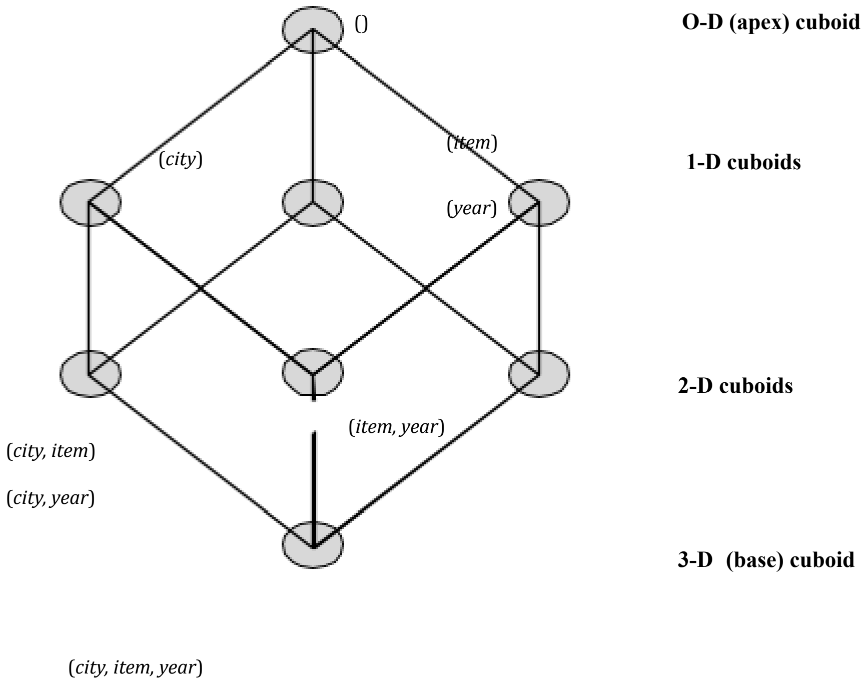


**Figure 3.14** Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three dimensions *city, item*, and *year*.

of the $n$ dimensions. Therefore, the cube operator is the $n$-dimensional generalization of the group by operator.

Based on the syntax of DMQL introduced in Section 3.2.3, the data cube in

Example 3.11 could be defined as

define cube sales cube [city, item, year]: sum(sales in dollars)

For a cube with $n$ dimensions, there are a total of $2^n$ cuboids, including the base cuboid. A statement such as

compute cube sales cube

would explicitly instruct the system to compute the sales aggregate cuboids for all of the eight subsets of the set *city, item, year* , including the empty subset. A cube computation operator was first proposed and studied by Gray et al. [GCB+97].

On-line analytical processing may need to access different cuboids for different queries. Therefore, it may seem like a good idea to compute all or at least some of the cuboids in a data cube in advance. Precomputation leads to fast response time and avoids some redundant computation. Most, if not all, OLAP products resort to some degree of pre- computation of multidimensional aggregates.

A major challenge related to this precomputation, however, is that the required storage space may explode if all of the cuboids in a data cube are precomputed, especially when the cube has many dimensions. The storage requirements are even more excessive when many of the dimensions have associated concept hierarchies, each with multiple levels. This problem is referred to as the **curse of dimensionality**. The extent of the curse of dimensionality is illustrated below.

"*How many cuboids are there in an* n-*dimensional data cube?*" If there were no hierarchies associated with each dimension, then the total number of cuboids for an $n$-dimensional data cube, as we have seen above, is $2^n$. However, in practice, many dimensions do have hierarchies. For example, the dimension *time* is usually not explored at only one conceptual level, such as *year*, but rather at multiple conceptual levels, such as in the hierarchy "*day* < *month* < *quarter* < *year*". For an $n$-dimensional data cube, the total number of cuboids that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$Total\ number\ of\ cuboids = \prod_{i=1}^{n}(L_i + 1), \qquad (3.1)$$

where $L_i$ is the number of levels associated with dimension $i$. One is added to $L_i$ in Equation (3.1) to include the *virtual* top level, all. (Note that generalizing to all is equiv- alent to the removal of the dimension.) This formula is based on the fact that, at most, one abstraction level in each dimension will appear in a cuboid. For example, the time dimension as specified above has 4 conceptual levels, or 5 if we include the virtual level all. If the cube has 10 dimensions and each dimension has 5 levels (including all), the total number of cuboids that can be generated is $5^{10} \quad 9.8 \quad 10^6$. The size of each cuboid also depends on the *cardinality* (i.e., number of distinct values) of each dimension. For example, if the *AllElectronics* branch in each city sold every item, there would be *city item* tuples in the *city-item* group-by alone. As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by's will grossly exceed the (fixed) size of the input relation.

By now, you probably realize that it is unrealistic to precompute and materialize all of the cuboids that can possibly be generated for a data cube (or from a base cuboid). If there are many cuboids, and these cuboids are large in size, a more reasonable option is *partial materialization*, that is, to materialize only *some* of the possible cuboids that can be generated.

## Partial Materialization: Selected Computation of Cuboids

There are three choices for data cube materialization given a base cuboid:

1. **No materialization**: Do not precompute any of the "nonbase" cuboids. This leads to computing expensive multidimensional aggregates on the fly, which can be extremely slow.
2. **Full materialization**: Precompute all of the cuboids. The resulting lattice of computed cuboids is referred to as the *full cube*. This choice typically requires huge amounts of memory space in order to store all of the precomputed cuboids.
3. **Partial materialization**: Selectively compute a proper subset of the whole set of possi- ble cuboids. Alternatively, we may compute a subset of the cube, which contains only those cells that satisfy some user-specified criterion, such as where the tuple count of each cell is above some threshold. We will use the term *subcube* to refer to the latter case, where only some of the cells may be precomputed for various cuboids. Partial materi- alization represents an interesting trade-off between storage space and response time.

The partial materialization of cuboids or subcubes should consider three factors: (1) identify the subset of cuboids or subcubes to materialize; (2) exploit the mate- rialized cuboids or subcubes during query processing; and (3) efficiently update the materialized cuboids or subcubes during load and refresh.

The selection of the subset of cuboids or subcubes to materialize should take into account the queries in the workload, their frequencies, and their accessing costs. In addi- tion, it should consider workload characteristics, the cost for incremental updates, and the total storage requirements. The selection must also consider the broad context of physical database design, such as the generation and selection of indices. Several OLAP products have adopted heuristic approaches for cuboid and subcube selection. A popular approach is to materialize the set of cuboids on which other frequently referenced cuboids are based. Alternatively, we can compute an *iceberg cube*, which is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold. Another common strategy is to materialize a *shell cube*. This involves precomputing the cuboids for only a small number of dimensions (such as 3 to 5) of a data cube. Queries on additional combinations of the dimensions can be computed on-the-fly. Because our aim in this chapter is to provide a solid introduction and overview of data warehousing for data mining, we defer our detailed discussion of cuboid selection and computation to Chapter 4, which studies data warehouse and OLAP implementation in greater depth. Once the selected cuboids have been materialized, it is important to take advantage of them during query processing. This involves several issues, such as how to determine the relevant cuboid(s) from among the candidate materialized cuboids, how to use available index structures on the materialized cuboids, and how to transform the OLAP opera- tions onto the selected cuboid(s). These issues are discussed in Section 3.4.3 as well as in Chapter 4.

Finally, during load and refresh, the materialized cuboids should be updated efficiently. Parallelism and incremental update techniques for this operation should be explored.

## 3.1.2 Indexing OLAP Data

To facilitate efficient data accessing, most data warehouse systems support index struc- tures and materialized views (using cuboids). General methods to select cuboids for materialization were discussed in the previous section. In this section, we examine how to index OLAP data by *bitmap indexing* and *join indexing*.

The **bitmap indexing** method is popular in OLAP products because it allows quick searching in data cubes. The bitmap index is an alternative representation of the *record ID (RID)* list. In the bitmap index for a given attribute, there is a distinct bit vector, $\boldsymbol{Bv}$, for each value $v$ in the domain of the attribute. If the domain of a given attribute consists of $n$ values, then $n$ bits are needed for each entry in the bitmap index (i.e., there are $n$ bit vectors). If the attribute has the value $v$ for a given row in the data table, then the bit representing that value is set to 1 in the corresponding row of the bitmap index. All other bits for that row are set to 0.

**Example 3.12** **Bitmap indexing.** In the *AllElectronics* data warehouse, suppose the dimension *item* at the top level has four values (representing item types): *"home entertainment," "computer," "phone,"* and *"security."* Each value (e.g., *"computer"*) is represented by a bit vector in the bitmap index table for *item*. Suppose that the cube is stored as a relation table with 100,000 rows. Because the domain of *item* consists of four values, the bitmap index table requires four bit vectors (or lists), each with 100,000 bits. Figure 3.15 shows a base (data) table containing the dimensions *item* and *city*, and its mapping to bitmap index tables for each of the dimensions. ■

Bitmap indexing is advantageous compared to hash and tree indices. It is especially useful for low-cardinality domains because comparison, join, and aggregation opera- tions are then reduced to bit arithmetic, which substantially reduces the processing time. Bitmap indexing leads to significant reductions in space and I/O since a string of charac- ters can be represented by a single bit. For higher-cardinality domains, the method can be adapted using compression techniques.

The **join indexing** method gained popularity from its use in relational database query processing. Traditional indexing maps the value in a given column to a list of rows having

| RID | *item* | *city* |
|---|---|---|
| R1 | H | V |
| R2 | C | V |
| R3 | P | V |
| R4 | S | V |
| R5 | H | T |
| R6 | C | T |
| R7 | P | T |
| R8 | S | T |

| RID | H | C | P | S |
|---|---|---|---|---|
| R1 | 1 | 0 | 0 | 0 |
| R2 | 0 | 1 | 0 | 0 |
| R3 | 0 | 0 | 1 | 0 |
| R4 | 0 | 0 | 0 | 1 |
| R5 | 1 | 0 | 0 | 0 |
| R6 | 0 | 1 | 0 | 0 |
| R7 | 0 | 0 | 1 | 0 |
| R8 | 0 | 0 | 0 | 1 |

| RID | V | T |
|---|---|---|
| R1 | 1 | 0 |
| R2 | 1 | 0 |
| R3 | 1 | 0 |
| R4 | 1 | 0 |
| R5 | 0 | 1 |
| R6 | 0 | 1 |
| R7 | 0 | 1 |
| R8 | 0 | 1 |

Base table Item bitmap index table City bitmap index table

Note: H for "home entertainment, " C for "computer, " P for "phone, " S for "security, " V for "Vancouver, " T for "Toronto."

**Figure 3.15** Indexing OLAP data using bitmap indices.

that value. In contrast, join indexing registers the joinable rows of two relations from a relational database. For example, if two relations $R(RID, A)$ and $S(B, SID)$ join on the attributes $A$ and $B$, then the join index record contains the pair $(RID, SID)$, where $RID$ and $SID$ are record identifiers from the $R$ and $S$ relations, respectively. Hence, the join index records can identify joinable tuples without performing costly join operations. Join indexing is especially useful for maintaining the relationship between a foreign key[3] and its matching primary keys, from the joinable relation.

The star schema model of data warehouses makes join indexing attractive for cross- table search, because the linkage between a fact table and its corresponding dimension tables comprises the foreign key of the fact table and the primary key of the dimen- sion table. Join indexing maintains relationships between attribute values of a dimension (e.g., within a dimension table) and the corresponding rows in the fact table. Join indices may span multiple dimensions to form **composite join indices**. We can use join indices to identify subcubes that are of interest.

**Example 3.13** **Join indexing.** In Example 3.4, we defined a star schema for *AllElectronics* of the form "*sales star* [*time, item, branch, location*]: *dollars sold* = sum *(sales in dollars)*". An exam- ple of a join index relationship between the *sales* fact table and the dimension tables for *location* and *item* is shown in Figure 3.16. For example, the "*Main Street*" value in the *location* dimension table joins with tuples T57, T238, and T884 of the *sales* fact table. Similarly, the "*Sony-TV*" value in the *item* dimension table joins with tuples T57 and T459 of the *sales* fact table. The corresponding join index tables are shown in Figure 3.17.

[3]A set of attributes in a relation schema that forms a primary key for another relation schema is called a **foreign key**.

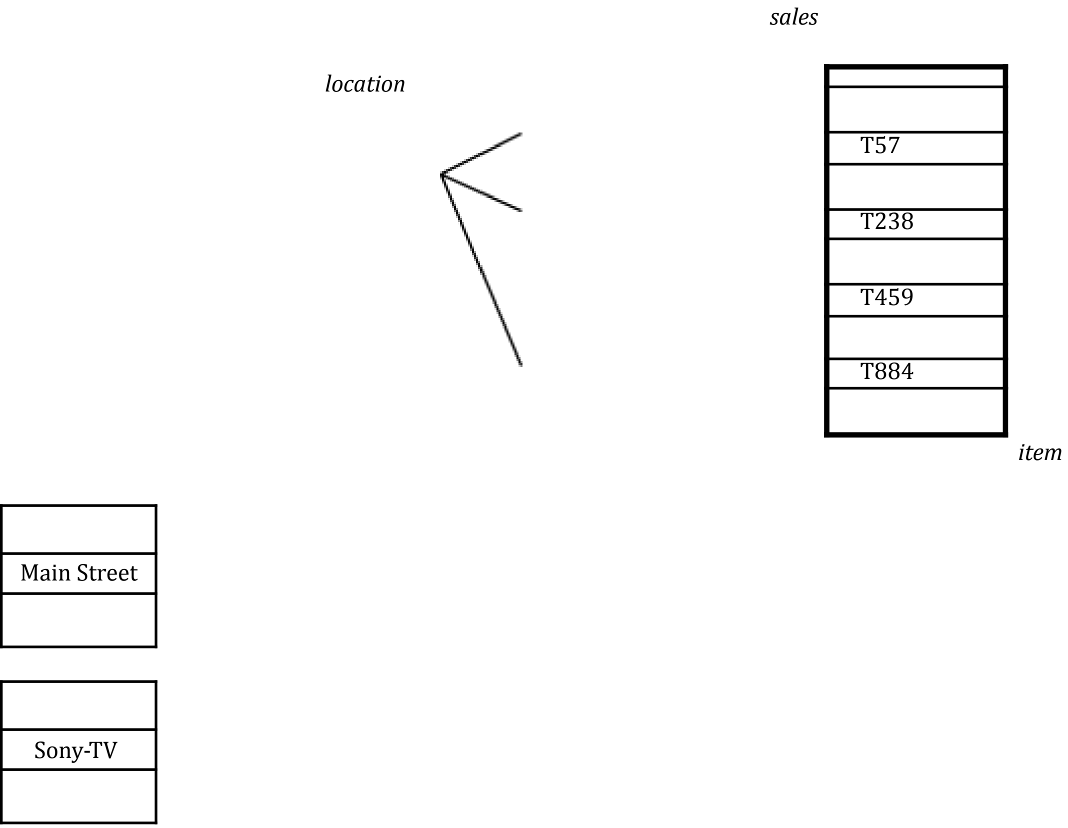


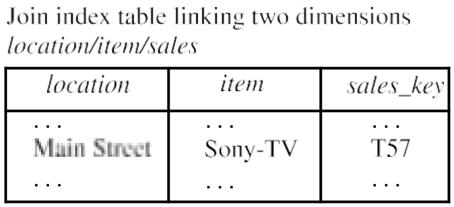
**Figure 3.16** Linkages between a *sales* fact table and dimension tables for *location* and *item*.

Join index table for *location/sales*

| *location* | *sales_key* |
|---|---|
| . . . | . . . |
| Main Street | T57 |
| Main Street | T238 |
| Main Street | T884 |
| . . . | . . . |

Join index table for *item/sales*

| *item* | *sales_key* |
|---|---|
| . . . | . . . |
| Sony-TV | T57 |
| Sony-TV | T459 |
| . . . | . . . |


Join index table linking two dimensions
*location/item/sales*

| *location* | *item* | *sales_key* |
|---|---|---|
| ... | ... | ... |
| Main Street | Sony-TV | T57 |
| ... | ... | ... |

**Figure 3.17** Join index tables based on the linkages between the *sales* fact table and dimension tables for

*location* and *item* shown in Figure 3.16.

Suppose that there are 360 time values, 100 items, 50 branches, 30 locations, and 10 million sales tuples in the *sales_star* data cube. If the *sales* fact table has recorded sales for only 30 items, the remaining 70 items will obviously not participate in joins. If join indices are not used, additional I/Os have to be performed to bring the joining portions of the fact table and dimension tables together. ■

To further speed up query processing, the join indexing and bitmap indexing methods can be integrated to form **bitmapped join indices**.

## 3.1.3 Efficient Processing of OLAP Queries

The purpose of materializing cuboids and constructing OLAP index structures is to speed up query processing in data cubes. Given materialized views, query processing should proceed as follows:

1. **Determine which operations should be performed on the available cuboids:** This involves transforming any selection, projection, roll-up (group-by), and drill-down operations specified in the query into corresponding SQL and/or OLAP operations. For example, slicing and dicing a data cube may correspond to selection and/or pro- jection operations on a materialized cuboid.
2. **Determinetowhichmaterializedcuboid(s) therelevantoperationsshouldbeapplied:** This involves identifying all of the materialized cuboids that may potentially be used to answer the query, pruning the above set using knowledge of "dominance" relation- ships among the cuboids, estimating the costs of using the remaining materialized cuboids, and selecting the cuboid with the least cost.

**Example 3.14 OLAP query processing.** Suppose that we define a data cube for *AllElectronics* of the form "*sales cube* [*time, item, location*]: sum*(sales in dollars)*". The dimension hierarchies used are "*day* < *month* < *quarter* < *year*" for *time*, "*item name* < *brand* < *type*" for *item*, and "*street* < *city* < *province or state* < *country*" for *location*.

Suppose that the query to be processed is on [*brand, province or state*], with the selection constant "*year = 2004*". Also, suppose that there are four materialized cuboids available, as follows:

- cuboid 1: {*year, item name, city*}
- cuboid 2: {*year, brand, country*}
- cuboid 3: {*year, brand, province or state*}
- cuboid 4: {*item name, province or state*} where *year = 2004*

*"Which of the above four cuboids should be selected to process the query?"* Finer-granularity data cannot be generated from coarser-granularity data. Therefore, cuboid 2 cannot be used because *country* is a more general concept than *province or state*. Cuboids 1, 3, and 4 can be used to process the query because (1) they have the same set or a superset of the dimensions in the query, (2) the selection clause in the query can imply the selection in the cuboid, and (3) the abstraction levels for the *item* and *loca- tion* dimensions in these cuboids are at a finer level than *brand* and *province or state*, respectively.

*"How would the costs of each cuboid compare if used to process the query?"* It is likely that using cuboid 1 would cost the most because both *item name* and *city* are

at a lower level than the *brand* and *province_or_state* concepts specified in the query. If there are not many *year* values associated with *items* in the cube, but there are several *item_names* for each *brand*, then cuboid 3 will be smaller than cuboid 4, and thus cuboid 3 should be chosen to process the query. However, if efficient indices are available for cuboid 4, then cuboid 4 may be a better choice. Therefore, some cost-based estimation is required in order to decide which set of cuboids should be selected for query processing. ■

Because the storage model of a MOLAP server is an $n$-dimensional array, the front-end multidimensional queries are mapped directly to server storage structures, which provide direct addressing capabilities. The straightforward array representation of the data cube has good indexing properties, but has poor storage utilization when the data are sparse. For efficient storage and processing, sparse matrix and data compression tech- niques should therefore be applied. The details of several such methods of cube compu- tation are presented in Chapter 4.

The storage structures used by dense and sparse arrays may differ, making it advan- tageous to adopt a two-level approach to MOLAP query processing: use array structures for dense arrays, and sparse matrix structures for sparse arrays. The two-dimensional dense arrays can be indexed by B-trees.

To process a query in MOLAP, the dense one- and two-dimensional arrays must first be identified. Indices are then built to these arrays using traditional indexing structures. The two-level approach increases storage utilization without sacrificing direct addressing capabilities.

*"Arethereanyotherstrategiesforansweringqueriesquickly?"* Somestrategiesforanswer- ing queries quickly concentrate on providing *intermediate feedback* to the users. For exam- ple, in **on-line aggregation**, a data miningsystemcan display "what itknows so far" instead of waiting until the query is fully processed. Such an approximate answer to the given data mining query is periodically refreshed and refined as the computation process continues. Confidence intervals are associated with each estimate, providing the user with additional feedback regarding the reliability of the answer so far. This promotes interactivity with the system—the user gains insight as to whether or not he or she is probing in the "right" direction without having to wait until the end of the query. While on-line aggregation does not improve the total time to answer a query, the overall data mining process should be quicker due to the increased interactivity with the system.

Another approach is to employ **top *N* queries**. Suppose that you are interested in find- ing only the best-selling items among the millions of items sold at *AllElectronics*. Rather than waiting to obtain a list of all store items, sorted in decreasing order of sales, you would like to see only the top $N$. Using statistics, query processing can be optimized to return the top $N$ items, rather than the whole sorted list. This results in faster response time while helping to promote user interactivity and reduce wasted resources.

The goal of this section was to provide an overview of data warehouse implementa- tion. Chapter 4 presents a more advanced treatment of this topic. It examines the efficient computation of data cubes and processing of OLAP queries in greater depth, providing detailed algorithms.

## 3.2 Data Warehouse Implementation

Data warehouses contain huge volumes of data. OLAP servers demand that decision support queries be answered in the order of seconds. Therefore, it is crucial for data ware- house systems to support highly efficient cube computation techniques, access methods, and query processing techniques. In this section, we present an overview of methods for the efficient implementation of data warehouse systems.

### 3.2.1 Efficient Computation of Data Cubes

At the core of multidimensional data analysis is the efficient computation of aggregations across many sets of dimensions. In SQL terms, these aggregations are referred to as group-by's. Each group-by can be represented by a *cuboid*, where the set of group-by's forms a lattice of cuboids defining a data cube. In this section, we explore issues relating to the efficient computation of data cubes.

#### The compute cube Operator and the Curse of Dimensionality

One approach to cube computation extends SQL so as to include a compute cube oper- ator. The compute cube operator computes aggregates over all subsets of the dimensions specified in the operation. This can require excessive storage space, especially for large numbers of dimensions. We start with an intuitive look at what is involved in the efficient computation of data cubes.

**Example 3.11** **A data cube is a lattice of cuboids.** Suppose that you would like to create a data cube for *AllElectronics* sales that contains the following: *city, item, year*, and *sales in dollars*. You would like to be able to analyze the data, with queries such as the following:

- "*Compute the sum of sales, grouping by city and*
- *item.*" "*Compute the sum of sales, grouping by city.*"
- "*Compute the sum of sales, grouping by item.*"

What is the total number of cuboids, or group-by's, that can be computed for this data cube? Taking the three attributes, *city, item*, and *year*, as the dimensions for the data cube, and *sales_in_dollars* as the measure, the total number of cuboids, or group-by's, that can be computed for this data cube is $2^3 = 8$. The possible group-by's are the following: {(*city, item, year*), (*city, item*), (*city, year*), (*item, year*), (*city*), (*item*), (*year*), ()}, where () means that the group-by is empty (i.e., the dimensions are not grouped). These group-by's form a lattice of cuboids for the data cube, as shown in Figure 3.14. The **base cuboid** contains all three dimensions, *city, item*, and *year*. It can return the total sales for any combination of the three dimensions. The **apex cuboid**, or 0-D cuboid, refers to the case where the group-by is empty. It contains the total sum of all sales. The base cuboid is the least generalized (most specific) of the cuboids. The apex cuboid is the most generalized (least specific) of the cuboids, and is often denoted as all. If we start at the apex cuboid and explore downward in the lattice, this is equivalent to drilling down within the data cube. If we start at the base cuboid and explore upward, this is akin to rolling up. ■

An SQL query containing no group-by, such as "compute the sum of total sales," is a *zero-dimensional operation*. An SQL query containing one group-by, such as "compute the sum of sales, group by city," is a *one-dimensional operation*. A cube operator on $n$ dimensions is equivalent to a collection of group by statements, one for each subset

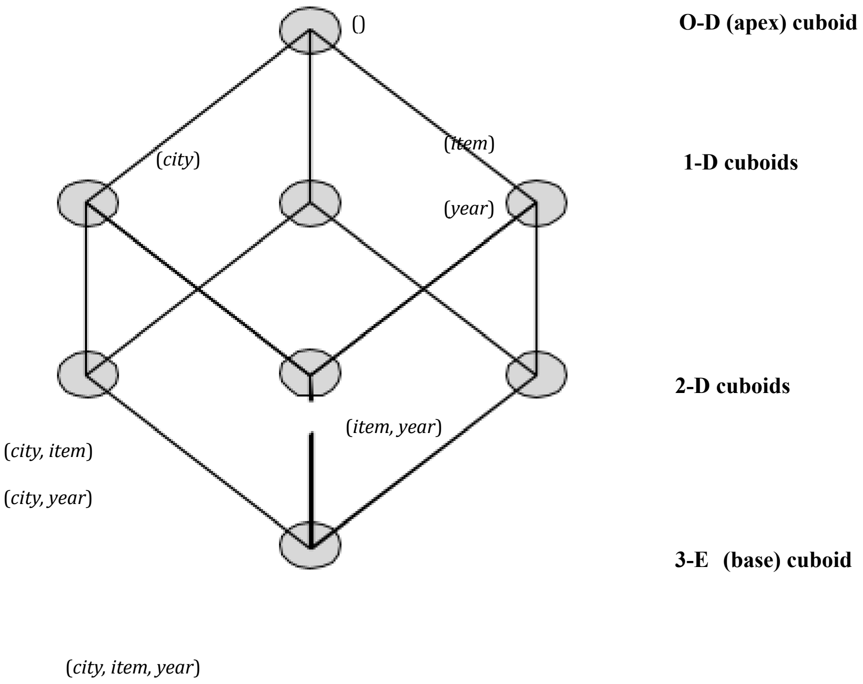


**Figure 3.14** Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three dimensions *city, item*, and *year*.

of the $n$ dimensions. Therefore, the cube operator is the $n$-dimensional generalization of the group by operator.

Based on the syntax of DMQL introduced in Section 3.2.3, the data cube in Example 3.11 could be defined as

define cube sales cube [city, item, year]: sum(sales in dollars)

For a cube with $n$ dimensions, there are a total of $2^n$ cuboids, including the base cuboid. A statement such as

compute cube sales cube

would explicitly instruct the system to compute the sales aggregate cuboids for all of the eight subsets of the set *city, item, year* , including the empty subset. A cube computation operator was first proposed and studied by Gray et al. [GCB+97].

On-line analytical processing may need to access different cuboids for different queries. Therefore, it may seem like a good idea to compute all or at least some of the cuboids in a data cube in advance. Precomputation leads to fast response time and avoids some redundant computation. Most, if not all, OLAP products resort to some degree of pre- computation of multidimensional aggregates.

A major challenge related to this precomputation, however, is that the required storage space may explode if all of the cuboids in a data cube are precomputed, especially when the cube has many dimensions. The storage requirements are even more excessive when many of the dimensions have associated concept hierarchies, each with multiple levels. This problem is referred to as the **curse of dimensionality**. The extent of the curse of dimensionality is illustrated below.

"*How many cuboids are there in an* n-*dimensional data cube?*" If there were no hierarchies associated with each dimension, then the total number of cuboids for an $n$-dimensional data cube, as we have seen above, is $2^n$. However, in practice, many dimensions do have hierarchies. For example, the dimension *time* is usually not explored at only one conceptual level, such as *year*, but rather at multiple conceptual levels, such as in the hierarchy "*day < month < quarter < year*". For an $n$-dimensional data cube, the total number of cuboids that can be generated (including the cuboids generated by climbing up the hierarchies along each dimension) is

$$\text{Total number of cuboids} = \prod_{i=1}^{n}(L_i + 1), \tag{3.1}$$

where $L_i$ is the number of levels associated with dimension $i$. One is added to $L_i$ in Equation (3.1) to include the *virtual* top level, all. (Note that generalizing to all is equiv- alent to the removal of the dimension.) This formula is based on the fact that, at most, one abstraction level in each dimension will appear in a cuboid. For example, the time dimension as specified above has 4 conceptual levels, or 5 if we include the virtual level all. If the cube has 10 dimensions and each dimension has 5 levels (including all), the total number of cuboids that can be generated is $5^{10}$ $9.8$ $10^6$.

The size of each cuboid also depends on the *cardinality* (i.e., number of distinct values) of each dimension. For example, if the *AllElectronics* branch in each city sold every item, there would be *city item* tuples in the *city-item* group-by alone. As the number of dimensions, number of conceptual hierarchies, or cardinality increases, the storage space required for many of the group-by's will grossly exceed the (fixed) size of the input relation.

By now, you probably realize that it is unrealistic to precompute and materialize all of the cuboids that can possibly be generated for a data cube (or from a base cuboid). If there are many cuboids, and these cuboids are large in size, a more reasonable option is *partial materialization*, that is, to materialize only *some* of the possible cuboids that can be generated.

## Partial Materialization: Selected Computation of Cuboids

There are three choices for data cube materialization given a base cuboid:

1. **No materialization**: Do not precompute any of the "nonbase" cuboids. This leads to computing expensive multidimensional aggregates on the fly, which can be extremely slow.
2. **Full materialization**: Precompute all of the cuboids. The resulting lattice of computed cuboids is referred to as the *full cube*. This choice typically requires huge amounts of memory space in order to store all of the precomputed cuboids.
3. **Partial materialization**: Selectively compute a proper subset of the whole set of possi- ble cuboids. Alternatively, we may compute a subset of the cube, which contains only those cells that satisfy some user-specified criterion, such as where the tuple count of each cell is above some threshold. We will use the term *subcube* to refer to the latter case, where only some of the cells may be precomputed for various cuboids. Partial materi- alization represents an interesting trade-off between storage space and response time.

The partial materialization of cuboids or subcubes should consider three factors:
(2) identify the subset of cuboids or subcubes to materialize; (2) exploit the mate- rialized cuboids or subcubes during query processing; and (3) efficiently update the materialized cuboids or subcubes during load and refresh.

The selection of the subset of cuboids or subcubes to materialize should take into account the queries in the workload, their frequencies, and their accessing costs. In addi- tion, it should consider workload characteristics, the cost for incremental updates, and the total storage requirements. The selection must also consider the broad context of physical database design, such as the generation and selection of indices. Several OLAP products have adopted heuristic approaches for cuboid and subcube selection. A popular approach is to materialize the set of cuboids on which other frequently referenced cuboids are based. Alternatively, we can compute an *iceberg cube*, which is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold. Another common strategy is to materialize a *shell cube*. This involves precomputing the cuboids for only a small number of dimensions (such as 3 to 5) of a data cube. Queries on additional combinations of the dimensions can be computed on-the-fly. Because our aim in this chapter is to provide a solid introduction and overview of data warehousing for data mining, we defer our detailed discussion of cuboid selection and computation to Chapter 4, which studies data warehouse and OLAP implementation in greater depth. Once the selected cuboids have been materialized, it is important to take advantage of them during query processing. This involves several issues, such as how to determine the relevant cuboid(s) from among the candidate materialized cuboids, how to use available index structures on the materialized cuboids, and how to transform the OLAP opera- tions onto the selected cuboid(s). These issues are discussed in Section 3.4.3 as well as in Chapter 4.

Finally, during load and refresh, the materialized cuboids should be updated efficiently. Parallelism and incremental update techniques for this operation should be explored.

## 3.2.2 Indexing OLAP Data

To facilitate efficient data accessing, most data warehouse systems support index struc- tures and materialized views (using cuboids). General methods to select cuboids for materialization were discussed in the previous section. In this section, we examine how to index OLAP data by *bitmap indexing* and *join indexing*.

The **bitmap indexing** method is popular in OLAP products because it allows quick searching in data cubes. The bitmap index is an alternative representation of the *record ID (RID)* list. In the bitmap index for a given attribute, there is a distinct bit vector, $B_v$, for each value $v$ in the domain of the attribute. If the domain of a given attribute consists of $n$ values, then $n$ bits are needed for each entry in the bitmap index (i.e., there are $n$ bit vectors). If the attribute has the value $v$ for a given row in the data table, then the bit representing that value is set to 1 in the corresponding row of the bitmap index. All other bits for that row are set to 0.

**Example 3.12** **Bitmap indexing.** In the *AllElectronics* data warehouse, suppose the dimension *item* at the top level has four values (representing item types): *"home entertainment," "computer," "phone,"* and *"security."* Each value (e.g., *"computer"*) is represented by a bit vector in the bitmap index table for *item*. Suppose that the cube is stored as a relation table with 100,000 rows. Because the domain of *item* consists of four values, the bitmap index table requires four bit vectors (or lists), each with 100,000 bits. Figure 3.15 shows a base (data) table containing the dimensions *item* and *city*, and its mapping to bitmap index tables for each of the dimensions. ■

Bitmap indexing is advantageous compared to hash and tree indices. It is especially useful for low-cardinality domains because comparison, join, and aggregation opera- tions are then reduced to bit arithmetic, which substantially reduces the processing time. Bitmap indexing leads to significant reductions in space and I/O since a string of charac- ters can be represented by a single bit. For higher-cardinality domains, the method can be adapted using compression techniques.

The **join indexing** method gained popularity from its use in relational database query processing. Traditional indexing maps the value in a given column to a list of rows having

| RID | *item* | *city* |
|---|---|---|
| R1 | H | V |
| R2 | C | V |
| R3 | P | V |
| R4 | S | V |
| R5 | H | T |
| R6 | C | T |
| R7 | P | T |
| R8 | S | T |

| RID | H | C | P | S |
|---|---|---|---|---|
| R1 | 1 | 0 | 0 | 0 |
| R2 | 0 | 1 | 0 | 0 |
| R3 | 0 | 0 | 1 | 0 |
| R4 | 0 | 0 | 0 | 1 |
| R5 | 1 | 0 | 0 | 0 |
| R6 | 0 | 1 | 0 | 0 |
| R7 | 0 | 0 | 1 | 0 |
| R8 | 0 | 0 | 0 | 1 |

| RID | V | T |
|---|---|---|
| R1 | 1 | 0 |
| R2 | 1 | 0 |
| R3 | 1 | 0 |
| R4 | 1 | 0 |
| R5 | 0 | 1 |
| R6 | 0 | 1 |
| R7 | 0 | 1 |
| R8 | 0 | 1 |

Base table Item bitmap index table City bitmap index table

Note: H for "home entertainment, " C for "computer, " P for "phone, " S for "security, " V for "Vancouver, " T for "Toronto."

**Figure 3.15** Indexing OLAP data using bitmap indices.

that value. In contrast, join indexing registers the joinable rows of two relations from a relational database. For example, if two relations $R(RID, A)$ and $S(B, SID)$ join on the attributes $A$ and $B$, then the join index record contains the pair $(RID, SID)$, where $RID$ and $SID$ are record identifiers from the $R$ and $S$ relations, respectively. Hence, the join index records can identify joinable tuples without performing costly join operations. Join indexing is especially useful for maintaining the relationship between a foreign key[3] and its matching primary keys, from the joinable relation.

The star schema model of data warehouses makes join indexing attractive for cross- table search, because the linkage between a fact table and its corresponding dimension tables comprises the foreign key of the fact table and the primary key of the dimen- sion table. Join indexing maintains relationships between attribute values of a dimension (e.g., within a dimension table) and the corresponding rows in the fact table. Join indices may span multiple dimensions to form **composite join indices**. We can use join indices to identify subcubes that are of interest.

**Example 3.13** **Join indexing.** In Example 3.4, we defined a star schema for *AllElectronics* of the form "*sales star* [*time, item, branch, location*]: *dollars sold* = sum *(sales in dollars)*". An exam- ple of a join index relationship between the *sales* fact table and the dimension tables for *location* and *item* is shown in Figure 3.16. For example, the "*Main Street*" value in the *location* dimension table joins with tuples T57, T238, and T884 of the *sales* fact table. Similarly, the "*Sony-TV*" value in the *item* dimension table joins with tuples T57 and T459 of the *sales* fact table. The corresponding join index tables are shown in Figure 3.17.

[3]A set of attributes in a relation schema that forms a primary key for another relation schema is called a **foreign key**.

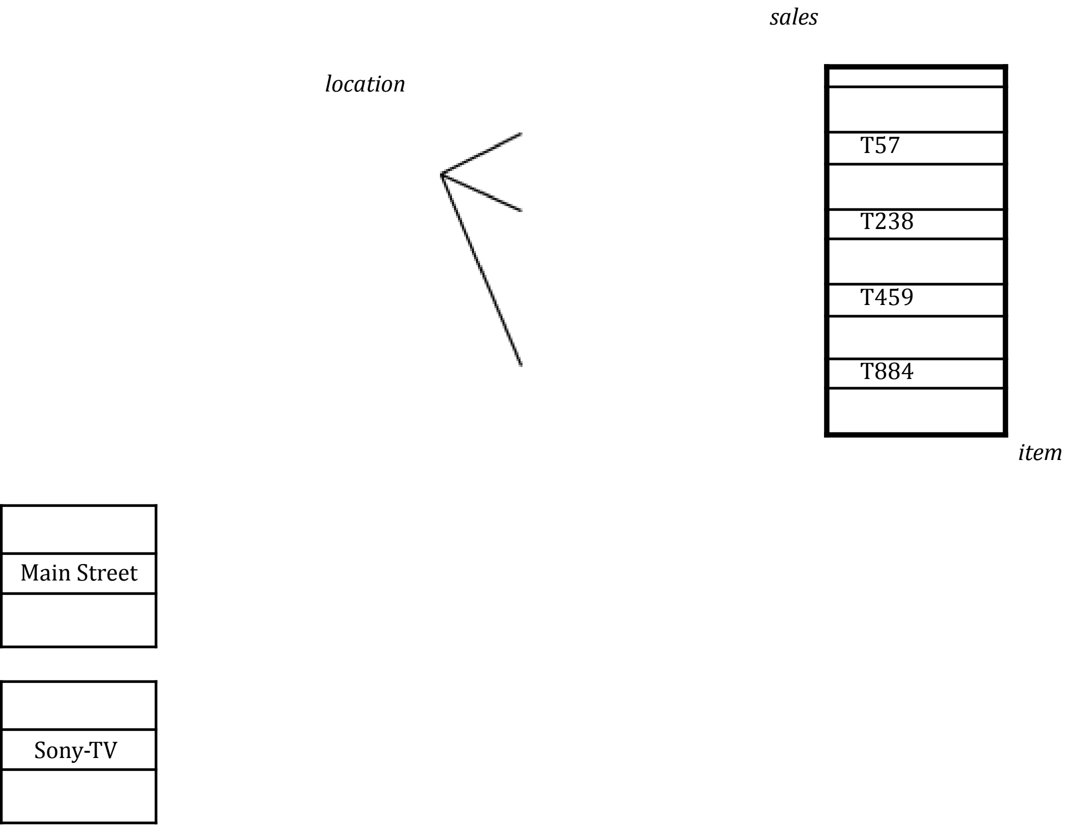


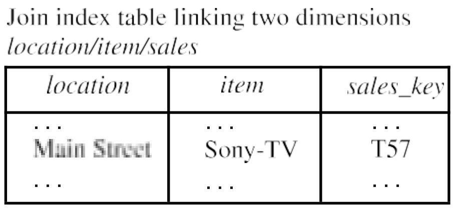
**Figure 3.16** Linkages between a *sales* fact table and dimension tables for *location* and *item*.

Join index table for *location/sales*

| *location* | *sales_key* |
|---|---|
| ... | ... |
| Main Street | T57 |
| Main Street | T238 |
| Main Street | T884 |
| ... | ... |

Join index table for *item/sales*

| *item* | *sales_key* |
|---|---|
| ... | ... |
| Sony-TV | T57 |
| Sony-TV | T459 |
| ... | ... |



Join index table linking two dimensions
*location/item/sales*

| *location* | *item* | *sales_key* |
|---|---|---|
| . . . | . . . | . . . |
| Main Street | Sony-TV | T57 |
| . . . | . . . | . . . |

**Figure 3.17** Join index tables based on the linkages between the *sales* fact table and dimension tables for

*location* and *item* shown in Figure 3.16.

Suppose that there are 360 time values, 100 items, 50 branches, 30 locations, and 10 million sales tuples in the *sales_star* data cube. If the *sales* fact table has recorded sales for only 30 items, the remaining 70 items will obviously not participate in joins. If join indices are not used, additional I/Os have to be performed to bring the joining portions of the fact table and dimension tables together. ■

To further speed up query processing, the join indexing and bitmap indexing methods can be integrated to form **bitmapped join indices**.

## 3.2.3 Efficient Processing of OLAP Queries

The purpose of materializing cuboids and constructing OLAP index structures is to speed up query processing in data cubes. Given materialized views, query processing should proceed as follows:

3. **Determine which operations should be performed on the available cuboids:** This involves transforming any selection, projection, roll-up (group-by), and drill-down operations specified in the query into corresponding SQL and/or OLAP operations. For example, slicing and dicing a data cube may correspond to selection and/or pro- jection operations on a materialized cuboid.
4. **Determinetowhichmaterializedcuboid(s) therelevantoperationsshouldbeapplied:** This involves identifying all of the materialized cuboids that may potentially be used to answer the query, pruning the above set using knowledge of "dominance" relation- ships among the cuboids, estimating the costs of using the remaining materialized cuboids, and selecting the cuboid with the least cost.

**Example 3.14** **OLAP query processing.** Suppose that we define a data cube for *AllElectronics* of the form "*sales cube* [*time, item, location*]: sum(*sales in dollars*)". The dimension hierarchies used are "*day* < *month* < *quarter* < *year*" for *time*, "*item name* < *brand* < *type*" for *item*, and "*street* < *city* < *province or state* < *country*" for *location*.

Suppose that the query to be processed is on {*brand, province_or_state* , with the selection constant "*year = 2004*". Also, suppose that there are four materialized cuboids available, as follows:

- cuboid 1: {*year, item_name, city*}
- cuboid 2: {*year, brand, country*}
- cuboid 3: {*year, brand, province or state*}
- cuboid 4: {*item_name, province_or_state*} where *year = 2004*

*"Which of the above four cuboids should be selected to process the query?"* Finer-granularity data cannot be generated from coarser-granularity data. Therefore, cuboid 2 cannot be used because *country* is a more general concept than *province or state*. Cuboids 1, 3, and 4 can be used to process the query because (1) they have the same set or a superset of the dimensions in the query, (2) the selection clause in the query can imply the selection in the cuboid, and (3) the abstraction levels for the *item* and *loca- tion* dimensions in these cuboids are at a finer level than *brand* and *province or state*, respectively.

*"How would the costs of each cuboid compare if used to process the query?"* It is likely that using cuboid 1 would cost the most because both *item name* and *city* are

at a lower level than the *brand* and *province_or_state* concepts specified in the query. If there are not many *year* values associated with *items* in the cube, but there are several *item_names* for each *brand*, then cuboid 3 will be smaller than cuboid 4, and thus cuboid 3 should be chosen to process the query. However, if efficient indices are available for cuboid 4, then cuboid 4 may be a better choice. Therefore, some cost-based estimation is required in order to decide which set of cuboids should be selected for query processing. ■

Because the storage model of a MOLAP server is an $n$-dimensional array, the front-end multidimensional queries are mapped directly to server storage structures, which provide direct addressing capabilities. The straightforward array representation of the data cube has good indexing properties, but has poor storage utilization when the data are sparse. For efficient storage and processing, sparse matrix and data compression tech- niques should therefore be applied. The details of several such methods of cube compu- tation are presented in Chapter 4.

The storage structures used by dense and sparse arrays may differ, making it advan- tageous to adopt a two-level approach to MOLAP query processing: use array structures for dense arrays, and sparse matrix structures for sparse arrays. The two-dimensional dense arrays can be indexed by B-trees.

To process a query in MOLAP, the dense one- and two-dimensional arrays must first be identified. Indices are then built to these arrays using traditional indexing structures. The two-level approach increases storage utilization without sacrificing direct addressing capabilities.

*"Arethereanyotherstrategiesforansweringqueriesquickly?"* Somestrategiesforanswer- ing queries quickly concentrate on providing *intermediate feedback* to the users. For exam- ple, in **on-line aggregation**, a data miningsystemcan display "what itknows so far" instead of waiting until the query is fully processed. Such an approximate answer to the given data mining query is periodically refreshed and refined as the computation process continues. Confidence intervals are associated with each estimate, providing the user with additional feedback regarding the reliability of the answer so far. This promotes interactivity with the system—the user gains insight as to whether or not he or she is probing in the "right" direction without having to wait until the end of the query. While on-line aggregation does not improve the total time to answer a query, the overall data mining process should be quicker due to the increased interactivity with the system.

Another approach is to employ **top *N* queries**. Suppose that you are interested in find- ing only the best-selling items among the millions of items sold at *AllElectronics*. Rather than waiting to obtain a list of all store items, sorted in decreasing order of sales, you would like to see only the top $N$. Using statistics, query processing can be optimized to return the top $N$ items, rather than the whole sorted list. This results in faster response time while helping to promote user interactivity and reduce wasted resources.

The goal of this section was to provide an overview of data warehouse implementa- tion. Chapter 4 presents a more advanced treatment of this topic. It examines the efficient computation of data cubes and processing of OLAP queries in greater depth, providing detailed algorithms.

## 3.3 From Data Warehousing to Data Mining

*"How do data warehousing and OLAP relate to data mining?"* In this section, we study the usage of data warehousing for information processing, analytical processing, and data mining. We also introduce on-line analytical mining (OLAM), a powerful paradigm that integrates OLAP with data mining technology.

### 3.3.1 Data Warehouse Usage

Data warehouses and data marts are used in a wide range of applications. Business executives use the data in data warehouses and data marts to perform data analysis and make strategic decisions. In many firms, data warehouses are used as an integral part of a *plan-execute-assess* "closed-loop" feedback system for enterprise management. Data warehouses are used extensively in banking and financial services, consumer goods and retail distribution sectors, and controlled manufacturing, such as demand- based production.

Typically, the longer a data warehouse has been in use, the more it will have evolved. This evolution takes place throughout a number of phases. Initially, the data warehouse is mainly used for generating reports and answering predefined queries. Progressively, it is used to analyze summarized and detailed data, where the results are presented in the form of reports and charts. Later, the data warehouse is used for strategic purposes, per- forming multidimensional analysis and sophisticated slice-and-dice operations. Finally, the data warehouse may be employed for knowledge discovery and strategic decision making using data mining tools. In this context, the tools for data warehousing can be categorized into *access and retrieval tools*, *database reporting tools*, *data analysis tools*, and *data mining tools*.

Business users need to have the means to know what exists in the data warehouse (through metadata), how to access the contents of the data warehouse, how to examine the contents using analysis tools, and how to present the results of such analysis.

There are three kinds of data warehouse applications: *information processing, analyt- ical processing*, and *data mining*:

- **Information processing** supports querying, basic statistical analysis, and reporting using crosstabs, tables, charts, or graphs. A current trend in data warehouse infor- mation processing is to construct low-cost Web-based accessing tools that are then integrated with Web browsers.
- **Analytical processing** supports basic OLAP operations, including slice-and-dice, drill-down, roll-up, and pivoting. It generally operates on historical data in both sum- marized and detailed forms. The major strength of on-line analytical processing over information processing is the multidimensional data analysis of

data warehouse data.

- **Data mining** supports knowledge discovery by finding hidden patterns and associa- tions, constructing analytical models, performing classification and prediction, and presenting the mining results using visualization tools.

"*How does data mining relate to information processing and on-line analytical processing?*" Information processing, based on queries, can find useful information. How- ever, answers to such queries reflect the information directly stored in databases or com- putable by aggregate functions. They do not reflect sophisticated patterns or regularities buried in the database. Therefore, information processing is not data mining.

On-line analytical processing comes a step closer to data mining because it can derive information summarized at multiple granularities from user-specified subsets of a data warehouse. Such descriptions are equivalent to the class/concept descrip- tions discussed in Chapter 1. Because data mining systems can also mine generalized class/concept descriptions, this raises some interesting questions: *"Do OLAP systems perform data mining? Are OLAP systems actually data mining systems?"*

The functionalities of OLAP and data mining can be viewed as disjoint: OLAP is a data summarization/aggregation *tool* that helps simplify data analysis, while data mining allows the *automated discovery* of implicit patterns and interesting knowledge hidden in large amounts of data. OLAP tools are targeted toward simplifying and supporting interactive data analysis, whereas the goal of data mining tools is to automate as much of the process as possible, while still allowing users to guide the process. In this sense, data mining goes one step beyond traditional on-line analytical processing.

An alternative and broader view of data mining may be adopted in which data mining covers both data description and data modeling. Because OLAP systems can present general descriptions of data from data warehouses, OLAP functions are essen- tially for user-directed data summary and comparison (by drilling, pivoting, slicing, dicing, and other operations). These are, though limited, data mining functionalities. Yet according to this view, data mining covers a much broader spectrum than simple OLAP operations because it performs not only data summary and comparison but also association, classification, prediction, clustering, time-series analysis, and other data analysis tasks.

Data mining is not confined to the analysis of data stored in data warehouses. It may analyze data existing at more detailed granularities than the summarized data provided in a data warehouse. It may also analyze transactional, spatial, textual, and multimedia data that are difficult to model with current multidimensional database technology. In this context, data mining covers a broader spectrum than OLAP with respect to data mining functionality and the complexity of the data handled.

Because data mining involves more automated and deeper analysis than OLAP, data mining is expected to have broader applications. Data mining can help busi- ness managers find and reach more suitable customers, as well as gain critical business insights that may help drive market share and raise profits. In addi- tion, data mining can help managers understand customer group characteristics and develop optimal pricing strategies accordingly, correct item bundling based not on intuition but on actual item groups derived from customer purchase pat- terns, reduce promotional spending, and at the same time increase the overall net effectiveness of promotions.

## 3.3.2 From On-Line Analytical Processing to On-Line Analytical Mining

In the field of data mining, substantial research has been performed for data mining on various platforms, including transaction databases, relational databases, spatial databases, text databases, time-series databases, flat files, data warehouses, and so on.

**On-line analytical mining (OLAM)** (also called **OLAP mining**) integrates on-line analytical processing (OLAP) with data mining and mining knowledge in multidi- mensional databases. Among the many different paradigms and architectures of data mining systems, OLAM is particularly important for the following reasons:

- **High quality of data in data warehouses:** Most data mining tools need to work on integrated, consistent, and cleaned data, which requires costly data clean- ing, data integration, and data transformation as preprocessing steps. A data warehouse constructed by such preprocessing serves as a valuable source of high-quality data for OLAP as well as for data mining. Notice that data mining may also serve as a valuable tool for data cleaning and data integration as well.
- **Available information processing infrastructure surrounding data warehouses:** Comprehensive information processing and data analysis infrastructures have been or will be systematically constructed surrounding data warehouses, which include accessing, integration, consolidation, and transformation of multiple heterogeneous databases, ODBC/OLE DB connections, Web-accessing and service facilities, and reporting and OLAP analysis tools. It is prudent to make the best use of the available infrastructures rather than constructing everything from scratch.
- **OLAP-based exploratory data analysis:** Effective data mining needs exploratory data analysis. A user will often want to traverse through a database, select por-tions of relevant data, analyze them at different granularities, and present knowl-edge/results in different forms. On-line analytical mining provides facilities for data mining on different subsets of data and at different levels of abstraction, by drilling, pivoting, filtering, dicing, and slicing on a data cube and on some intermediate data mining results. This, together with data/knowledge visualization tools, will greatly enhance the power and flexibility of exploratory data mining.
- **On-line selection of data mining functions:** Often a user may not know what kinds of knowledge she would like to mine. By integrating OLAP with multiple data mining functions, on-line analytical mining provides users with the flexibility to select desired data mining functions and swap data mining tasks dynamically.

## Architecture for On-Line Analytical Mining

An OLAM server performs analytical mining in data cubes in a similar manner as an

OLAP server performs on-line analytical processing. An integrated OLAM and OLAP architecture is shown in Figure 3.18, where the OLAM and OLAP servers both accept user on-line queries (or commands) via a graphical user interface API and work with the data cube in the data analysis via a cube API. A metadata directory is used to

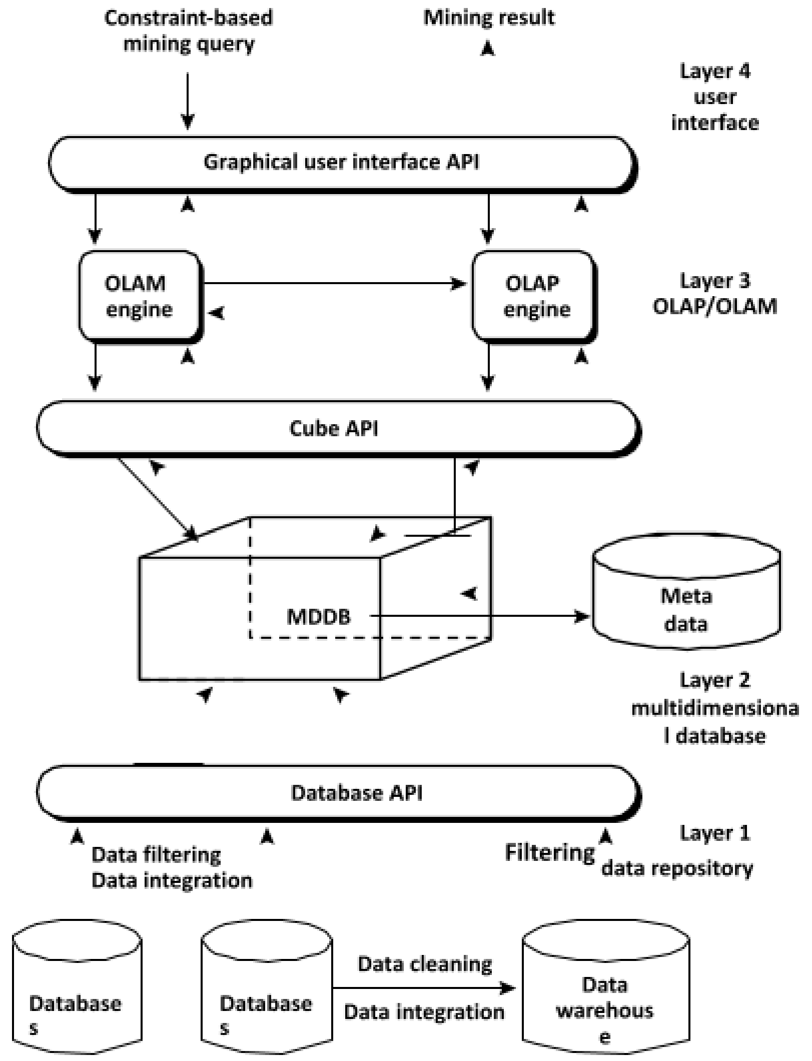


**Figure 3.18** An integrated OLAM and OLAP architecture.

guide the access of the data cube. The data cube can be constructed by accessing and/or integrating multiple databases via an MDDB API and/or by filtering a data warehouse via a database API that may support OLE DB or ODBC connections. Since an OLAM server may perform multiple data mining tasks, such as concept description, association, classification, prediction, clustering, time-series analysis, and so on, it usually consists of multiple integrated data mining modules and is more sophisticated than an OLAP server.

Chapter 4 describes data warehouses on a finer level by exploring implementation issues such as data cube computation, OLAP query answering strategies, and methods of generalization. The chapters following it are devoted to the study of data min- ing techniques. As we have seen, the introduction to data warehousing and OLAP technology presented in this chapter is essential to our study of data mining. This is because data warehousing provides users with large amounts of clean, organized, and summarized data, which greatly facilitates data mining. For example, rather than storing the details of each sales transaction, a data warehouse may store a summary of the transactions per item type for each branch or, summarized to a higher level, for each country. The capability of OLAP to provide multiple and dynamic views of summarized data in a data warehouse sets a solid foundation for successful data mining.

Moreover, we also believe that data mining should be a human-centered process. Rather than asking a data mining system to generate patterns and knowledge automat- ically, a user will often need to interact with the system to perform exploratory data analysis. OLAPsets agoodexample for interactive data analysis and provides the necessary preparations for exploratory data mining. Consider the discovery of association patterns, for example. Instead of mining associations at a primitive (i.e., low) data level among transactions, users should be allowed to specify roll-up operations along any dimension. For example, a user may like to roll up on the *item* dimension to go from viewing the data for particular TV sets that were purchased to viewing the brands of these TVs, such as SONY or Panasonic. Users may also navigate from the transaction level to the customer level or customer-type level in the search for interesting associations. Such an OLAP- style of data mining is characteristic of OLAP mining. In our study of the principles of data mining in this book, we place particular emphasis on OLAP mining, that is, on the *integration of data mining and OLAP technology*.

## 3.4 Summary

- A **data warehouse** is a *subject-oriented, integrated, time-variant*, and *nonvolatile* collection of data organized in support of management decision making. Several factors distinguish data warehouses from operational databases. Because the two systems provide quite different functionalities and require different kinds of data, it is necessary to maintain data warehouses separately from operational databases.

- A **multidimensional data model** is typically used for the design of corporate *data warehouses* and *departmental data marts*. Such a model can adopt a *star schema*, *snowflake schema*, or *fact constellation schema*. The core of the *multidimensional model* is the **data cube**, which consists of a large set of *facts* (or *measures*) and a number of *dimensions*. Dimensions are the entities or perspectives with respect to which an organization wants to keep records and are hierarchical in nature.

- A data cube consists of a **lattice of cuboids**, each corresponding to a different degree of summarization of the given multidimensional data.

- **Concept hierarchies** organize the values of attributes or dimensions into gradual levels of abstraction. They are useful in mining at multiple levels of abstraction.
- **On-line analytical processing (OLAP)** can be performed in data warehouses/marts using the multidimensional data model. Typical OLAP operations include *roll- up*, *drill-*(*down, across, through*), *slice-and-dice*, *pivot* (*rotate*), as well as statistical operations such as ranking and computing moving averages and growth rates. OLAP operations can be implemented efficiently using the data cube structure.
- Data warehouses often adopt a **three-tier architecture**. The bottom tier is a *warehouse database server*, which is typically a relational database system. The middle tier is an *OLAP server*, and the top tier is a *client*, containing query and reporting tools.
- A data warehouse contains **back-end tools and utilities** for populating and refresh- ing the warehouse. These cover data extraction, data cleaning, data transformation, loading, refreshing, and warehouse management.
- Data warehouse **metadata** are data defining the warehouse objects. A metadata repository provides details regarding the warehouse structure, data history, the algorithms used for summarization, mappings from the source data to warehouse form, system performance, and business terms and issues.
- OLAP servers may use **relational OLAP (ROLAP)**, or **multidimensional OLAP (MOLAP)**, or **hybrid OLAP (HOLAP)**. A ROLAP server uses an extended relational DBMS that maps OLAP operations on multidimensional data to standard relational operations. A MOLAP server maps multidimensional data views directly to array structures. A HOLAP server combines ROLAP and MOLAP. For example, it may use ROLAP for historical data while maintaining frequently accessed data in a separate MOLAP store.
- **Full materialization** refers to the computation of all of the cuboids in the lattice defin- ing a data cube. It typically requires an excessive amount of storage space, particularly as the number of dimensions and size of associated concept hierarchies grow. This problem is known as the **curse of dimensionality**. Alternatively, **partial materializa- tion** is the selective computation of a subset of the cuboids or subcubes in the lattice. For example, an **iceberg cube** is a data cube that stores only those cube cells whose aggregate value (e.g., count) is above some minimum support threshold.
- OLAP query processing can be made more efficient with the use of indexing techniques. In **bitmap indexing**, each attribute has its own bitmap index table. Bitmap indexing reduces join, aggregation, and comparison operations to bit arithmetic. **Join indexing** registers the joinable rows of two or more relations from a relational database, reducing the overall cost of OLAP join operations. **Bitmapped join indexing**, which combines the bitmap and join index methods, can be used to further speed up OLAP query processing.

- Data warehouses are used for *information processing* (querying and reporting), *ana- lytical processing* (which allows users to navigate through summarized and detailed data by OLAP operations), and *data mining* (which supports knowledge discovery). OLAP-based data mining is referred to as **OLAP mining**, or on-line analytical mining (**OLAM**), which emphasizes the interactive and exploratory nature of OLAP mining.