Chapter 19
Online Analytical Processing (OLAP)

The illustration depicted in Fig. 19.1 shows a typical journey of data from an operational database, data warehousing and OLAP to data analytics. All of the previous chapters focus on the left-hand side, the transformation process from the operational database to the data warehouse, where we discuss in length the step-by-step transformation process of various cases, using extensive case studies. In this chapter, we are going to focus on the middle part, that is, the *OLAP* part (or *Online Analytical Processing*) and *Business Intelligence* reporting.

An OLAP is an SQL query that retrieves data from the data warehouse. OLAP queries focus on data that would generally be used by management in the decision-making process. Hence, data that shows ranking, trends, sub-totals, grand totals, etc. would be of great interest to management. OLAP queries retrieve data from a data warehouse, but the retrieved results are not in an appropriate format for a presentation to management. These data are considered "raw". The retrieved data need to be properly formatted and presented to management using various reporting tools, which usually include graphs. These reports and graphs are called Business Intelligence. Therefore, the role of OLAP is to retrieve the necessary data from the data warehouse and to present this "raw" data to Business Intelligence for processing for presentation.

19.1 Sales Data Warehousing

A Sales data warehouse will be used throughout this chapter to illustrate how OLAP queries can retrieve data about sales. Figure 19.2 shows a layperson's view of a Sales data warehouse using a cube. It is clear that the cube has three dimensions, Time, Product and Location; each cell contains a Total Sales (Sales$) figure for a particular time, product and location.

The Sales star schema is shown in Fig. 19.3. Each dimension contains several attributes. The Time Dimension contains TimeID, Month and Year attributes,

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021

D. Taniar, W. Rahayu, *Data Warehousing and Analytics*, Data-Centric Systems and Applications, https://doi.org/10.1007/978-3-030-81979-8_19

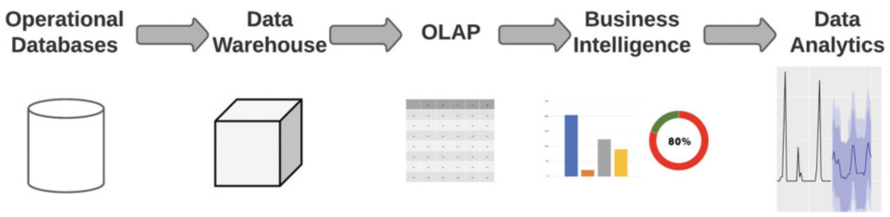

Fig. 19.1 A Data Journey: from operational to analytics

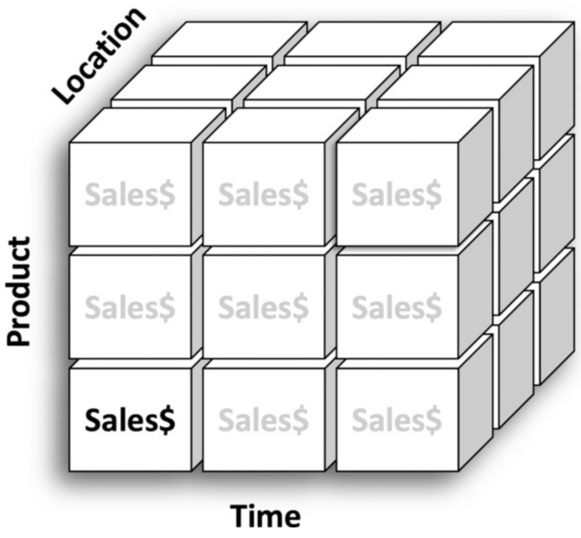

Fig. 19.2 A Sales cube

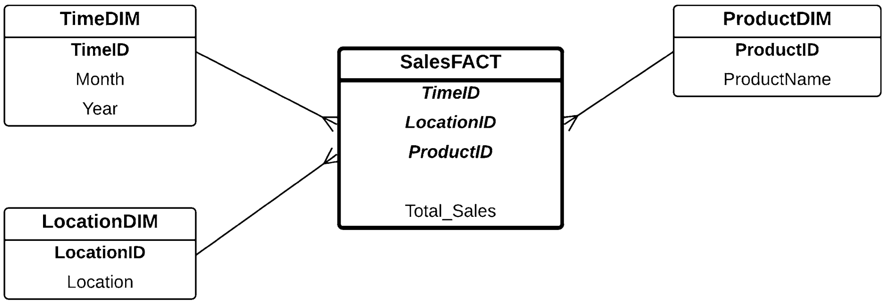

Fig. 19.3 A Sales star schema

where the TimeID attribute is the key identifier. The Product Dimension contains ProductID as the key identifier and the ProductName attribute. The Location Dimension contains LocationID and Location attributes. The Fact Table contains all three key identifiers from the dimensions as well as Total Sales as the fact measure.

Table 19.1 Time dimension

TimeID	Month	Year
201801	Jan	2018
201802	Feb	2018
201803	Mar	2018
201804	Apr	2018
201805	May	2018
201806	Jun	2018
...	...	...
201907	Jul	2019
201908	Aug	2019
201909	Sep	2019
201910	Oct	2019
201911	Nov	2019
201912	Dec	2019

Table 19.2 Location dimension

LocationID	Location
MEL	Melbourne
SYD	Sydney
ADL	Adelaide
PER	Perth

Table 19.3 Product dimension

ProductID	ProductName
C01	Clothing
S02	Shoes
A01	Accessories
C02	Cosmetics
K01	Kids and Baby

The following are the contents of the three dimension tables. For simplicity, the Sales Data Warehouse only contains data for a period of 2 years. Hence, the Time Dimension table, as shown in Table 19.1, contains 24 months. The Location Dimension table, as shown in Table 19.2, contains only four locations in Australia. The Product Dimension table, as shown in Table 19.3, contains five different product categories.

19.2 Basic Aggregate Functions

OLAP queries usually concentrate around aggregation and aggregate functions. The basic aggregate functions, which are common to SQL, are `count`, `sum`, `avg`, `max` and `min`, as well as the use of `group by` clause when using an aggregate function. These aggregate functions, as well as `group by`, have been used extensively in the previous chapters. However, for completeness of the discussions on OLAP, these basic aggregate functions are described again.

19.2.1 `count` Function

The `count` function counts the number of records in a table. Generally, there are three forms of `count`:

1. `count(*)`, which counts the number of records in a table
2. `count(attribute)`, which counts the number of records (excluding `null` values) in that specified attribute
3. `count(distinct attribute)`, which counts the number of unique records of the specified attribute

The following are the examples of the `count` function, and the respective query results are shown in Tables 19.4, 19.5, 19.6, and 19.7. Notice that there are 24 records in the Time Dimension table (see Table 19.4) but only 2 years in Table 19.5 as it uses `distinct`. Table 19.6 uses `distinct` and hence the results are 12 months, but Table 19.7 shows there are 24 months in the Time Dimension table. In this case, there is no difference between `count(*)` and `count(attribute)` as there are no `null` values in the Time Dimension table (e.g. both Tables 19.4 and 19.7 show 24 records).

```
select count(*) as Number_of_Records
from TimeDim;

select count(distinct Year) as Number_of_Years
from TimeDim;

select count(distinct Month) as Number_of_Months
from TimeDim;

select count(Month) as Number_of_Months
from TimeDim;
```

Table 19.4 `count(*)`

Number of records
24

Table 19.5 `count(distinct Year)`

Number of years
2

Table 19.6 `count(distinct Month)`

Number of months
12

Table 19.7 `count(Month)`

Number of months
24

Table 19.8 `sum(Total_Sales)`

Total Sales
2828318

19.2.2 `sum` Function

Given an attribute, the `sum` function calculates the total of the values in the specified attribute. Hence, the attribute required by the `sum` function must be a numerical attribute.

The following SQL command calculates the Total Sales in Melbourne in 2019. The result is shown in Table 19.8.

```
select sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and LocationID = 'MEL'
and Year = 2019;
```

19.2.3 `avg`, `max` and `min` Functions

The `avg`, `max` and `min` functions calculate the average, maximum and minimum value of the given attribute of a table. In an OLAP context, the attribute is a numerical attribute. The following retrieves the average, maximum and minimum Total Sales in Melbourne in 2019 from the Sales Fact Table.

```
select avg(Total_Sales) as Average_Sales
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and S.LocationID = 'MEL'
and T.Year = 2019;

select max(Total_Sales) as Maximum_Sales
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and S.LocationID = 'MEL'
and T.Year = 2019;

select min(Total_Sales) as Minimum_Sales
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and S.LocationID = 'MEL'
and T.Year = 2019;
```

The result of each of the above queries is one figure, which is average, maximum and minimum Total Sales, respectively. Tables 19.9, 19.10, and19.11 show the query results. These results show that Total Sales in Melbourne in 2019 range from $25,496 to $73,439, with an average of $47,138.63.

Table 19.9 Average sales

Average sales
47138.63

Table 19.10 Maximum sales

Maximum sales
73439

Table 19.11 Minimum sales

Minimum sales
25496

Table 19.12 Total sales by product

ProductName	Total sales
Clothing	715423
Cosmetics	633632
Shoes	645881
Accessories	355433
Kids and Baby	477949

19.2.4 `group by` Clause

Basic aggregate functions are often used in conjunction with `group by` clause in SQL. The `group by` clause is used to break down the aggregate value into several categories specified by the `group by` clause.

The following query retrieves the sum of Total Sales in Melbourne in 2019. The return value as shown previously in Table 19.8 is a single value which represents the sum of Total Sales.

```
select sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and LocationID = ’MEL’
and Year = 2019;
```

If we want to see the breakdown of this Total Sales by each Product, the `group by` clause can do it. The results are shown in Table 19.12.

```
select ProductName, sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T, ProductDim P
where S.TimeID = T.TimeID
and S.ProductID = P.ProductID
and LocationID = ’MEL’
and Year = 2019
group by ProductName
order by ProductName;
```

19.3 Cube and Rollup

Basic OLAP queries usually consist of new `group by` capabilities, namely, `group by cube` and `group by rollup`. The details are discussed in the following sections.

19.3.1 Cube

In OLAP, it is typical to produce a matrix visual or table which crosses two pieces of categorical information and contains sub-totals. Table 19.13 shows the Total Sales, together with sub-totals and the grand total of two Products (e.g. Clothing and Shoes) in two locations (e.g. Melbourne and Perth).

There are nine figures for Total Sales in Table 19.13. If we only use basic aggregate functions and `group by` clause, we need four separate SQL statements to retrieve all of these nine values. The first SQL retrieves each individual Total Sales of Clothing and Shoes in both locations, Melbourne and Perth, which is a total of four values. The second SQL retrieves the sub-totals of Clothing and Shoes in both locations (e.g. one sub-total for Clothing and another for Shoes). The third SQL retrieves the sub-totals of Melbourne and Perth for both products (e.g. one sub-total for Melbourne and another for Perth). Finally, the last SQL retrieves the grand total of Total Sales of both products in both locations. The `group by` clause is used in the first three SQL commands, but not the last one. The results are shown in Tables 19.14, 19.15, 19.16, and 19.17.

Table 19.13 Total sales by product

	Melbourne	Perth	
Clothing	$1,477,348	$861,268	$2,338,616
Shoes	$1,311,316	$897,153	$2,208,469
	$2,788,664	$1,758,421	$4,547,085

Table 19.14 Product-location total sales

ProductName	Location	Total sales
Clothing	Melbourne	1477348
Clothing	Perth	861268
Shoes	Melbourne	1311316
Shoes	Perth	897153

Table 19.15 Product sub-totals

ProductName	Total sales
Clothing	2338616
Shoes	2208469

Table 19.16 Location sub-totals

Location	Total sales
Melbourne	2788664
Perth	1758421

Table 19.17 Grand total

Total sales
4547085

```
select ProductName, Location,
  sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
group by ProductName, Location;

select ProductName, sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
group by ProductName;

select Location, sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
group by Location;

select sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes');
```

To avoid using four SQL commands, the `group by cube` clause can be used. The SQL is almost the same as the previous query expressed in `group by`, but we now use `group by cube`. The results are shown in Table 19.18, and all the nine Total Sales figures are there, although the formatting of the query result is not presented as professionally as required. Nonetheless, the results are all there.

```
select ProductName, Location, sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
```

```
group by cube (ProductName, Location)
order by ProductName, Location;
```

Rows 1–3 in Table 19.18 (e.g. Clothing (null)) indicate the sub-total for Clothing, whereas rows 4–6 (e.g. Shoes (null)) are the sub-total for Shoes. Hence, the sub-total for each Product is there. Rows 7 and 8 (e.g. (null) Melbourne and (null) Perth) show the sub-totals for Melbourne and Perth, respectively. Notice from these rows, the (null) in the query result indicates sub-total, not the (null) value of the column. Finally, the last row, which is (null) (null), indicates the grand total of Total Sales from both cities and products.

So, in short, the `group by cube` clause can be used to produce a matrix visual or table.

19.3.2 Rollup

`group by rollup` is quite similar to `group by cube`. If the above query is expressed using `group by rollup` as follows, the results are shown in Table 19.19. The sub-totals for Clothing and Shoes are still there, but not the sub-totals for the two cities. So the `rollup` results are a subset of those of `cube`. The `rollup` results indicate that if there is a (null) in the categorical column (e.g. the DimensionID attribute), there cannot be a not (null) value in the right-side column. Hence, (null) Melbourne and (null) Perth will be excluded from the `rollup` results.

Table 19.18 Product-location cube

	ProductName	Location	Total sales
1	Clothing	Melbourne	1477348
2	Clothing	Perth	861268
3	Clothing	(null)	2338616
4	Shoes	Melbourne	1311316
5	Shoes	Perth	897153
6	Shoes	(null)	2208469
7	(null)	Melbourne	2788664
8	(null)	Perth	1758421
9	(null)	(null)	4547085

Table 19.19 Product-Location rollup

	ProductName	Location	Total sales
1	Clothing	Melbourne	1477348
2	Clothing	Perth	861268
3	Clothing	(null)	2338616
4	Shoes	Melbourne	1311316
5	Shoes	Perth	897153
6	Shoes	(null)	2208469
7	(null)	(null)	4547085

Table 19.20 Location-Product rollup

	Location	ProductName	Total sales
1	Melbourne	Clothing	1477348
2	Melbourne	Shoes	1311316
3	Melbourne	(null)	2788664
4	Perth	Clothing	861268
5	Perth	Shoes	897153
6	Perth	(null)	1758421
7	(null)	(null)	4547085

On the other hand, Clothing (null) and Shoes (null), as well as (null) (null), are still in the `rollup` results (e.g. rows 3, 6 and 7). Therefore, the order of the attributes (or columns) in the `group by rollup` is important.

```
select ProductName, Location, sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
group by rollup (ProductName, Location)
order by ProductName, Location;
```

If we change the order of the attributes in the `group by rollup` to (Location, ProductName) as in the SQL below, the results will be different (see Table 19.20).

```
select Location, ProductName, sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
group by rollup (Location, ProductName)
order by Location, ProductName;
```

In contrast, the order of the attributes in `group by cube` does not matter because `group by cube` produces all combinations between the attributes.

19.3.3 Rollup vs. Cube

The difference between `rollup` and `cube` will become clearer if there are more than two attributes in the `group by`. The following SQL uses three attributes (e.g. ProductName, Location and TimeID) in the `cube`. The results are shown in Table 19.21. For simplicity, the query focuses on January data only (e.g. 201801 and 201901). The results show all possible sub-totals are included: individual Product, Location, Time and combinations of these.

Table 19.21 Product-Location-Time cube

	ProductName	Location	TimeID	Total sales
1	Clothing	Melbourne	201801	60325
2	Clothing	Melbourne	201901	64823
3	Clothing	Melbourne	(null)	125148
4	Clothing	Perth	201801	37279
5	Clothing	Perth	201901	41479
6	Clothing	Perth	(null)	78758
7	Clothing	(null)	201801	97604
8	Clothing	(null)	201901	106302
9	Clothing	(null)	(null)	203906
10	Shoes	Melbourne	201801	61119
11	Shoes	Melbourne	201901	63968
12	Shoes	Melbourne	(null)	125087
13	Shoes	Perth	201801	44556
14	Shoes	Perth	201901	27112
15	Shoes	Perth	(null)	71668
16	Shoes	(null)	201801	105675
17	Shoes	(null)	201901	91080
18	Shoes	(null)	(null)	196755
19	(null)	Melbourne	201801	121444
20	(null)	Melbourne	201901	128791
21	(null)	Melbourne	(null)	250235
22	(null)	Perth	201801	81835
23	(null)	Perth	201901	68591
24	(null)	Perth	(null)	150426
25	(null)	(null)	201801	203279
26	(null)	(null)	201901	197382
27	(null)	(null)	(null)	400661

```
select
  ProductName, Location, TimeID,
  sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
and TimeID in ('201801', '201901')
group by cube (ProductName, Location, TimeID)
order by ProductName, Location, TimeID;
```

If we use `group by rollup` instead, as in the following SQL query, the results are shown in Table 19.22. It is clear that (null) followed by non (null) in the categorical columns will be excluded in the rollup results. Hence, the rollup results are a subset of the cube results. Also be aware that the order of the attributes

Table 19.22
Product-Location-Time rollup

	ProductName	Location	TimeID	Total sales
1	Clothing	Melbourne	201801	60325
2	Clothing	Melbourne	201901	64823
3	Clothing	Melbourne	(null)	125148
4	Clothing	Perth	201801	37279
5	Clothing	Perth	201901	41479
6	Clothing	Perth	(null)	78758
7	Clothing	(null)	(null)	203906
8	Shoes	Melbourne	201801	61119
9	Shoes	Melbourne	201901	63968
10	Shoes	Melbourne	(null)	125087
11	Shoes	Perth	201801	44556
12	Shoes	Perth	201901	27112
13	Shoes	Perth	(null)	71668
14	Shoes	(null)	(null)	196755
15	(null)	(null)	(null)	400661

in the `group by rollup` is important as a different ordering of attributes will produce different results.

```
select
  ProductName, Location, TimeID,
  sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
and TimeID in ('201801', '201901')
group by rollup (ProductName, Location, TimeID)
order by ProductName, Location, TimeID;
```

19.3.4 Partial Cube and Partial Rollup

Partial cube or partial rollup means that one or more attributes are taken out from the cube and rollup. For example, if the full cube is `group by cube(A,B,C)` where *A*, *B* and *C* are attributes, an example of a partial cube is attribute *A*, which is taken out from the cube but is still within the group by `group by A cube(B,C)`. We need to understand what it means by a full cube `cube(A,B,C)` in the first place. `cube(A,B,C)` means that each of the attributes *A*, *B* and *C* (and their combinations) will have a sub-total in the query results; the attribute will have a (null) entry in the respective column. When an attribute (e.g. attribute *A*) is taken out from the cube, as in `group by A cube(B,C)`, it means that attribute *A* will not have a sub-total anymore, as it is not in the cube any longer.

Table 19.23 Product-Location-Time **partial cube**

	ProductName	Location	TimeID	Total sales
1	Clothing	Melbourne	201801	60325
2	Clothing	Melbourne	201901	64823
3	Clothing	Melbourne	(null)	125148
4	Clothing	Perth	201801	37279
5	Clothing	Perth	201901	41479
6	Clothing	Perth	(null)	78758
7	Clothing	(null)	201801	97604
8	Clothing	(null)	201901	106302
9	Clothing	(null)	(null)	203906
10	Shoes	Melbourne	201801	61119
11	Shoes	Melbourne	201901	63968
12	Shoes	Melbourne	(null)	125087
13	Shoes	Perth	201801	44556
14	Shoes	Perth	201901	27112
15	Shoes	Perth	(null)	71668
16	Shoes	(null)	201801	105675
17	Shoes	(null)	201901	91080
18	Shoes	(null)	(null)	196755

Let's compare `group by cube(A,B,C)` with `group by A cube(B,C)`. Using the Product-Location-Time cube in the previous section, the SQL for the full cube is as follows. The results are shown in Table 19.21 with 27 rows in the query results.

```
select
  ProductName, Location, TimeID,
  sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
and TimeID in ('201801', '201901')
group by cube (ProductName, Location, TimeID)
order by ProductName, Location, TimeID;
```

The next OLAP is partial cube, where ProductName is taken out from the cube, as in the following SQL command. Looking at the full cube results shown previously in Table 19.21, it is clear that rows 19–27 (which are the last nine rows from the full cube results) will be excluded from the partial cube results because of (null) in the ProductName column (see Table 19.23).

Table 19.24 Product-Location-Time **partial rollup**

	ProductName	Location	TimeID	Total sales
1	Clothing	Melbourne	201801	60325
2	Clothing	Melbourne	201901	64823
3	Clothing	Melbourne	(null)	125148
4	Clothing	Perth	201801	37279
5	Clothing	Perth	201901	41479
6	Clothing	Perth	(null)	78758
7	Clothing	(null)	(null)	203906
8	Shoes	Melbourne	201801	61119
9	Shoes	Melbourne	201901	63968
10	Shoes	Melbourne	(null)	125087
11	Shoes	Perth	201801	44556
12	Shoes	Perth	201901	27112
13	Shoes	Perth	(null)	71668
14	Shoes	(null)	(null)	196755

```
select
  ProductName, Location, TimeID,
  sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
and TimeID in ('201801', '201901')
group by ProductName, cube (Location, TimeID)
order by ProductName, Location, TimeID;
```

If we change the above partial cube to partial rollup, as in the following SQL command, rows 7–8 and 16–17 from the partial cube results in Table 19.23 will be excluded from the partial rollup results (see Table 19.24 which contains only 14 rows now).

```
select
  ProductName, Location, TimeID,
  sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
and TimeID in ('201801', '201901')
group by ProductName, rollup (Location, TimeID)
order by ProductName, Location, TimeID;
```

Table 19.25 Product-Location cube with `grouping`

	ProductName	Location	Total sales	Product	Location
1	Clothing	Melbourne	1477348	0	0
2	Clothing	Perth	861268	0	0
3	Clothing	(null)	2338616	0	1
4	Shoes	Melbourne	1311316	0	0
5	Shoes	Perth	897153	0	0
6	Shoes	(null)	2208469	0	1
7	(null)	Melbourne	2788664	1	0
8	(null)	Perth	1758421	1	0
9	(null)	(null)	4547085	1	1

19.3.5 `grouping` and `decode` Functions

The cube and rollup queries use the (null) entry to indicate that it is about the subtotal. While in OLAP we do not normally consider the presentation of the query results (because the presentation will be handled by Business Intelligence—see the last section in this chapter), sometimes we would like to change the wording of (null) to something more meaningful. To achieve this, the `grouping` and `decode` functions can be used.

The `grouping` function is a binary function that will produce 0 or 1. Let's look at the following cube example where the `grouping` function is used. The results are shown in Table 19.25. Note the last two columns show the `grouping` function results, where 1 indicates that the corresponding column has a (null) value.

```
select
  ProductName, D.Location, sum(Total_Sales) as Total_Sales,
  grouping(ProductName) as Product,
  grouping(D.Location) as Location
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and S.LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
group by cube (ProductName, D.Location)
order by ProductName, D.Location;
```

Now we can incorporate the `grouping` function into the `decode` function. The `decode` function is like an `if then else` statement in a programming language. It takes four parameters. If the first parameter matches with the second parameter, then the result is the third parameter; else the result is the fourth parameter.

The following is the SQL command to use both `decode` and `grouping` functions. The results are shown in Table 19.26. Note the (null) entries in ProductName and Location columns are now reworded more appropriately (e.g. All Products or All Locations).

```
select
  decode(grouping(ProductName),1,'All Products',ProductName)
    as Product_Name,
  decode(grouping(Location),1,'All Locations',Location)
    as Location,
  sum(Total_Sales) as Total_Sales
from SalesFact S, ProductDim P, LocationDim D
where S.ProductID = P.ProductID
and S.LocationID = D.LocationID
and LocationID in ('MEL', 'PER')
and ProductName  in ('Clothing', 'Shoes')
group by cube (ProductName, Location)
order by ProductName, Location;
```

19.4 Ranking

In addition to cube and rollup, ranking is one of the most common operations in Business Intelligence reporting. Naturally, management decision-making is often based on reports that show the ranking of certain aspects of the business.

19.4.1 Rank

OLAP ranking queries can be expressed using the `rank() over` function. The following example shows a ranking report based on the Total Sales of each Product. The data is limited to 2019 data of sales in Melbourne. The results are shown in Table 19.27.

Table 19.26 Product-Location cube with `decode`

	ProductName	Location	Total sales
1	Clothing	Melbourne	1477348
2	Clothing	Perth	861268
3	Clothing	All Locations	2338616
4	Shoes	Melbourne	1311316
5	Shoes	Perth	897153
6	Shoes	All Locations	2208469
7	All Products	Melbourne	2788664
8	All Products	Perth	1758421
9	All Products	All Locations	4547085

Table 19.27 Product sales ranking

	ProductName	Total sales	Sales rank	Custom rank
1	Clothing	715423	5	1
2	Shoes	645881	4	2
3	Cosmetics	633632	3	3
4	Kids and Baby	477949	2	4
5	Accessories	355433	1	5

```
select
  ProductName,
  sum(Total_Sales) as Total_Sales,
  rank() over(order by sum(Total_Sales)) as Sales_Rank,
  rank() over(order by sum(Total_Sales) desc) as Custom_Rank
from SalesFact S, ProductDim P, TimeDim T
where S.ProductID = P.ProductID
and S.TimeID = T.TimeID
and S.LocationID = 'MEL'
and Year = 2019
group by ProductName
order by Custom_Rank;
```

The `rank() over` function uses `order by Total Sales`, which sorts the records based on the Total Sales. The first `rank() over` sorts the Total Sales from the smallest to the largest value; hence, the ranking is from 5 to 1, where rank 5 has the largest Total Sales figure, and rank 1 is the lowest. However, in business reporting, the top figure should be ranked 1. In order to do this, the `order by` clause must use `desc` so that the sorting is from the largest to the smallest.

In case there is a tie between two Total Sales (e.g. just assume that Shoes and Cosmetics have exactly the same Total Sales), there are two ways to present the rankings: one is using the `rank() over` function, and the other is using the `dense_rank() over` function. To contrast these two ranking functions, let's look at these examples.

The following is the normal ranking method using the `rank() over` function. The results are shown in Table 19.28. In this example, we assume the Total Sales of Shoes and Cosmetics are the same; both are ranked 2. Since they are tied, there is no product ranked 3. The next ranking is rank 4.

```
select
  ProductName,
  sum(Total_Sales) as Total_Sales,
  rank() over(order by sum(Total_Sales) desc) as Rank
from SalesFact S, ProductDim P, TimeDim T
where S.ProductID = P.ProductID
and S.TimeID = T.TimeID
and S.LocationID = 'MEL'
and Year = 2019
group by ProductName
order by Rank;
```

Table 19.28 Product Sales ranking with (normal) rank

	ProductName	Total sales	Rank
1	Clothing	715423	1
2	Shoes	645881	2
3	Cosmetics	645881	2
4	Kids and Baby	477949	4
5	Accessories	355433	5

Table 19.29 Product Sales ranking with dense rank

	ProductName	Total sales	Dense rank
1	Clothing	715423	1
2	Shoes	645881	2
3	Cosmetics	645881	2
4	Kids and Baby	477949	3
5	Accessories	355433	4

Now it is the same query but uses `dense_rank() over` instead. The results in Table 19.29 show that no rank is skipped, even though there is a tie. So, in this case, after two rank 2, the next one is rank 3, so it is a dense rank.

```
select
  ProductName,
  sum(Total_Sales) as Total_Sales,
  dense_rank() over(order by sum(Total_Sales) desc)
    as Dense_Rank
from SalesFact S, ProductDim P, TimeDim T
where S.ProductID = P.ProductID
and S.TimeID = T.TimeID
and S.LocationID = 'MEL'
and Year = 2019
group by ProductName
order by Dense_Rank;
```

If there is no tie, there will be no difference between `rank() over` and `dense_rank() over` functions.

There is another function, `row_number() over`, which is not meant for ranking but shows a similar behaviour to ranking. The `row_number() over` function actually gives a row number based on the `order by` clause inside `row_number() over` function. The following is the same SQL command as above but uses the `row_number() over` function.

```
select
  ProductName,
  sum(Total_Sales) as Total_Sales,
  row_number() over(order by sum(Total_Sales) desc)
    as Row_Number
from SalesFact S, ProductDim P, TimeDim T
where S.ProductID = P.ProductID
and S.TimeID = T.TimeID
and S.LocationID = 'MEL'
and Year = 2019
group by ProductName
order by Row_Number;
```

Again assuming that Shoes and Cosmetics have exactly the same Total Sales, Table 19.30 shows the result of the `row_number() over` query. Note that in case of a tie, the row number will keep the increment. There is no particular ordering method for the records that are tied, which in this case are rows 2 and 3. Row

Table 19.30 Product Sales row number

	ProductName	Total Sales	Row Number
1	Clothing	715423	1
2	Shoes	645881	2
3	Cosmetics	645881	3
4	Kids and Baby	477949	4
5	Accessories	355433	5

Table 19.31 Top-2 Products with highest total sales

	ProductName	Total sales	Product rank
1	Clothing	715423	1
2	Shoes	645881	2

numbers are given to the query results based on the `order by` clause in the `row_number() over`.

19.4.2 Top-N and Top-Percent Ranking

When the ranking list is longer, the management often likes to focus on a few top ranks only. This is called *Top-N* ranking, where *N* is a number in the ranking list. The following SQL command retrieves the Top-2 Products based on their Total Sales. The results are shown in Table 19.31, showing Clothing and Shoes are the two top Products with the highest Total Sales. The data is limited to only Year 2019 in Melbourne.

Looking at the inner query, the inner query retrieves the ranking of all Products based on the Total Sales. The outer query filters these results by choosing only the Top-2 records.

```
select *
from
  (select
     ProductName,
     sum(Total_Sales) as Total_Sales,
     rank() over(order by sum(Total_Sales) desc)
       as Product_Rank
   from SalesFact S, TimeDim T, LocationDim L, ProductDim P
   where S.TimeID = T.TimeID
   and S.LocationID = L.LocationID
   and S.ProductID = P.ProductID
   and Year = 2019
   group by ProductName)
where Product_Rank <= 2;
```

Another way to choose the top few ranked items is by specifying the top rank percentage rather than specifying an integer number to indicate the top *N* records in the complete ranking list. The `percent_rank() over` function can be used.

Table 19.32 Percent rank of all products

	ProductName	Total sales	Percent Rank
1	Clothing	715423	1
2	Shoes	645881	0.75
3	Cosmetics	633632	0.5
4	Kids and Baby	477949	0.25
5	Accessories	355433	0

This function calculates the ranking in terms of the percentage. The following is the SQL query that shows the percentage ranking of all Products.

```
select ProductName, sum(Total_Sales) as Total_Sales,
  percent_rank() over (order by sum(Total_Sales))
    as Percent_Rank
from SalesFact S, TimeDim T, ProductDim P
where S.TimeID = T.TimeID
and S.ProductID = P.ProductID
and Year = 2019
and LocationID = 'MEL'
group by ProductName
order by Percent_Rank desc;
```

The results are shown in Table 19.32. The Percent Rank ranges from 0 (the bottom rank) to 1 (the top rank). Because our dataset only has five Products, the Percent Rank increments by 0.25 (or 25%) per Product. If we want to select the Top-2 Products, it could be tricky to use the `percent_rank` function. However, if we would like to retrieve the top 75% of the Product, then we can specify a condition in the outer query, such as the following:

```
select *
from (
  select ProductName, sum(Total_Sales) as Total_Sales,
    percent_rank() over (order by sum(Total_Sales))
      as Percent_Rank
  from SalesFact S, TimeDim T, ProductDim P
  where S.TimeID = T.TimeID
  and S.ProductID = P.ProductID
  and Year = 2019
  and LocationID = 'MEL'
  group by ProductName
  order by Percent_Rank)
where Percent_Rank >= 0.75;
```

19.4.3 Partition

In the previous sections, ranking is applied to one set of data, which is the Total Sales of each Product, and then the ranking method is to rank each Product based on its Total Sales. However, in some cases, we may not want to compare the Total Sales

Table 19.33 Product Sales ranking partition by Product

	ProductName	TimeID	Total sales	Rank by product
1	Clothing	201902	224253	1
2	Clothing	201901	207673	2
3	Clothing	201903	170238	3
4	Shoes	201902	182786	1
5	Shoes	201903	177432	2
6	Shoes	201901	167820	3

between Products, but we would like to rank each product internally, for example, based on the Month. So, each Product has its own internal ranking, that is, to rank the Month's Total Sales. In this case, we need a *partition*.

`partition by` is a clause within the `rank() over` function, which partitions the dataset into several partitions, and each partition will have its own internal ranking. The following shows an example of an internal ranking of Clothing and Shoes based on the monthly Total Sales for the period of January to March 2019. The results as shown in Table 19.33 show that Clothing has its own internal ranking based on monthly Total Sales and so does Shoes.

```
select
  ProductName, TimeID,
  sum(Total_Sales) as Total_Sales,
  rank() over(partition by ProductName
    order by sum(Total_Sales) desc) as Rank_by_Product
from SalesFact S, ProductDim P
where S.ProductID = P.ProductID
and TimeID in ('201901', '201902', '201903')
and P.ProductName in ('Clothing', 'Shoes')
group by ProductName, TimeID
order by ProductName, Rank_by_Product;
```

The results combine the two partitions (e.g. Clothing partition and Shoes partition) into one table in the query results because SQL queries produce the results in a single table format. If we do not carefully sort the results according to the partitioned attribute, we might mix the two partitions in the query results, which may confuse the readers. Nevertheless, the job of OLAP queries is to retrieve the required data. Business Intelligence Reporting tools will be used for presentation purposes. So, Table 19.33 might be presented as "two" cards or sheets, where the first sheet is for the Clothing and the second for the Shoes (refer to an illustration in Fig. 19.4).

A partition can be applied to multiple attributes. The following SQL command partitions the data based on Product as well as Time. It means Product has its own internal ranking based on Total Sales, and so does Time. The results are shown in Table 19.34. When multiple partitions are used in one query, the results can be confusing because both partitions are jumbled into one table. It will be easier to imagine the illustration in Fig. 19.5. The picture shows that the Product partition has "two" sheets, one for Clothing and the other for Shoes, each with its own ranking.

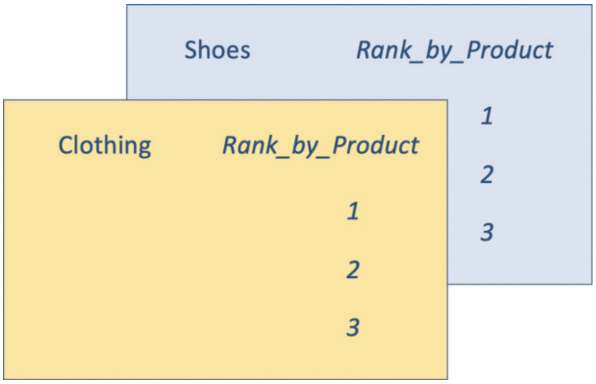

Fig. 19.4 Ranking partition by Product

Table 19.34 Product Sales ranking partition by Product and Time

	ProductName	TimeID	Total sales	Rank by product	Rank by time
1	Clothing	201902	224253	1	1
2	Clothing	201901	207673	2	1
3	Clothing	201903	170238	3	2
4	Shoes	201902	182786	1	2
5	Shoes	201903	177432	2	1
6	Shoes	201901	167820	3	2

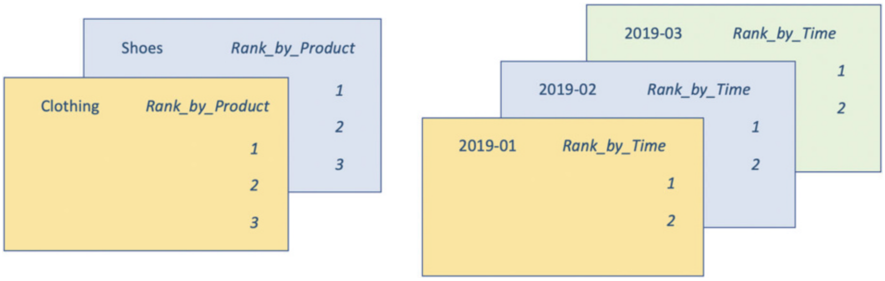

Fig. 19.5 Ranking partition by Product and Time

It also shows that the Time partition has “three” sheets because there are 3 months (e.g. January to March), again each with their own ranking based on the Total Sales.

```
select
  ProductName, TimeID,
  sum(Total_Sales) as Total_Sales,
  rank() over(partition by ProductName
    order by sum(Total_Sales) desc) as Rank_by_Product,
  rank() over(partition by TimeID
    order by sum(Total_Sales) desc) as Rank_by_TimeID
from SalesFact S, ProductDim P
```

```
where S.ProductID = P.ProductID
and TimeID in ('201901', '201902', '201903')
and P.ProductName in ('Clothing', 'Shoes')
group by ProductName, TimeID
order by ProductName;
```

Inspecting Table 19.33, the ranking based on Product is probably easy to see, as the Clothing rankings are in one group and the Shoes rankings are in the other. But checking the ranking for the Time can be confusing because of the absence of visual grouping. Row 1 for Clothing in 201902 is ranked 1. We need to find which other Product in 201902 is ranked 2. In this case, it is Shoes, which is on row 4. So, row 1 is on the same group with row 4, which is the ranking for 201902 in which Clothing has a higher Total Sales than Shoes. Based on this method, we can see that row 2 is in the same group as row 6, that is, 201901 ranking in which again Clothing has more Total Sales than Shoes. But for 201903 (see row 3 and row 5), Shoes is higher than Clothing.

19.5 Cumulative and Moving Aggregate

Another report that is common in business is cumulative and moving aggregates. Cumulative aggregate gets the cumulative sum, whereas moving aggregate is commonly used to get the moving average over a certain window size period. The former is usually used as an indicator of performance target, whereas the latter is often used to smooth out some outlier figures in a certain period of time.

19.5.1 Cumulative Aggregate

Cumulative aggregate uses the `rows unbounded preceding` clause in the `sum( sum() over)` function. The following is the SQL command to calculate the Cumulative Total Sales every month in 2019 in the Location of MEL.

Particularly, notice two things in this query: one is the use of `sum (sum (Total_Sales)) over`, and the other is the `rows unbounded preceding`. The first is to get the `sum` of `sum (Total_Sales)`. Note that `sum (Total_Sales)` calculates the sum of Total Sales per `group by`, whereas the first `sum` is to get the Cumulative Total Sales. The second important thing is `rows unbounded preceding`. The `unbounded` indicates that the cumulative total sales starts from the beginning or the first record of the query result. The query results are shown in Table 19.35. The cumulative column is highlighted.

Table 19.35 Cumulative sales

	LocationID	TimeID	Total Sales	**Cumulative**
1	MEL	201901	243,080	243,080
2	MEL	201902	259,353	502,433
3	MEL	201903	246,368	748,801
4	MEL	201904	237,193	986,714
5	MEL	201905	234,513	1,221,227
6	MEL	201906	235,630	1,456,857
7	MEL	201907	251,330	1,708,187
8	MEL	201908	239,575	1,947,762
9	MEL	201909	208,165	2,115,927
10	MEL	201910	228,710	2,384,637
11	MEL	201911	207,694	2,592,331
12	MEL	201912	235,987	2,828,318

```
select LocationID, S.TimeID,
  to_char(sum(Total_Sales), '999,999,999') as Total_Sales,
  to_char(sum(sum(Total_Sales)) over
    (order by LocationID, S.TimeID
    rows unbounded preceding), '999,999,999') as Cumulative
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and Year = 2019
and LocationID in ('MEL')
group by LocationID, S.TimeID;
```

Cumulative aggregate can incorporate partition, which is often used where the dataset is partitioned into several groups, and each group has its own cumulative aggregate. Table 19.35 contains data from one Location (e.g. MEL). Suppose we need one cumulative report of two Locations; in this case, we need to partition the data based on Location. The following SQL command calculates the Cumulative Total Sales of two Locations (e.g. MEL and PER), individually. Notice the `partition by LocationID` clause in the `sum(sum (Total_Sales))` function. The results are shown in Table 19.36. Notice that the Cumulative Total Sales resets when a new Location (e.g. PER) started.

```
select LocationID, TimeID as Time,
  to_char(sum(Total_Sales), '999,999,999') as Total_Sales,
  to_char(sum(sum(Total_Sales)) over
    (partition by LocationID
    order by LocationID, TimeID
    rows unbounded preceding), '999,999,999') as Cumulative
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and Year = 2019
and LocationID in ('MEL', 'PER')
group by Location, TimeID;
```

Table 19.36 Cumulative Sales partitioned by Location

	LocationID	TimeID	Total Sales	Cumulative
1	MEL	201901	243,080	243,080
2	MEL	201902	259,353	502,433
3	MEL	201903	246,368	748,801
4	MEL	201904	237,193	986,714
5	MEL	201905	234,513	1,221,227
6	MEL	201906	235,630	1,456,857
7	MEL	201907	251,330	1,708,187
8	MEL	201908	239,575	1,947,762
9	MEL	201909	208,165	2,115,927
10	MEL	201910	228,710	2,384,637
11	MEL	201911	207,694	2,592,331
12	MEL	201912	235,987	2,828,318
13	PER	201901	150,456	150,456
14	PER	201902	147,412	297,868
15	PER	201903	135,137	433,005
16	PER	201904	136,877	569,882
17	PER	201905	151,685	721,567
18	PER	201906	136,491	858,058
19	PER	201907	130,951	989,009
20	PER	201908	172,590	1,161,599
21	PER	201909	156,086	1,317,685
22	PER	201910	167,189	1,484,874
23	PER	201911	176,590	1,661,464
24	PER	201912	141,242	1,802,706

19.5.2 Moving Aggregate

Syntactically, the difference between cumulative aggregate and moving aggregate is very small. In cumulative aggregate, it uses `rows unbounded preceding`, whereby the `unbounded` states that the starting of the cumulative is from the first row of the query results. In moving aggregate, the starting will not be from the beginning but from how many records behind the current record in the query results. Hence, in moving aggregate, we use `rows` *n* `proceedings`, where *n* is an integer number which indicates how many rows it is behind the current row.

The following SQL command gives an example of Moving 3-Month Average Total Sales. In this case, $n = 2$ because the average calculation started from 2 months before the current month, and hence it becomes the 3-Month Average. The only exception is the average of the first 2 months, which are the first month Total Sales (in row 1) and the average of 2 months' Total Sales (in row 2). After these, the Average will be based on 3 months (see Table 19.37 for the complete results).

Also notice in the SQL command that `avg(sum (Total_Sales))` is used instead of `sum(sum (Total_Sales))`. This is because we wanted to

Table 19.37 Moving 3-Month Average Sales

	LocationID	TimeID	Total Sales	**Avg 3 Months**
1	MEL	201901	243,080	243,080
2	MEL	201902	259,353	251,217
3	MEL	201903	246,368	249,600
4	MEL	201904	237,913	247,878
5	MEL	201905	234,513	239,598
6	MEL	201906	235,630	236,019
7	MEL	201907	251,330	240,491
8	MEL	201908	239,575	242,178
9	MEL	201909	208,165	233,023
10	MEL	201910	228,710	225,483
11	MEL	201911	207,694	214,856
12	MEL	201912	235,987	224,130

calculate the average of 3 months Total Sales; hence, the `avg` is used. The `(sum (Total_Sales)` is still used because it calculates the `sum` of Total Sales for each `group by`.

```
select LocationID, S.TimeID,
  to_char(sum(Total_Sales), '999,999,999') as Total_Sales,
  to_char(avg(sum(Total_Sales)) over
    (order by LocationID, S.TimeID
    rows 2 preceding), '999,999,999') as Avg_3_Months
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and Year = 2019
and LocationID in ('MEL')
group by LocationID, S.TimeID;
```

19.6 Business Intelligence Reporting

The data retrieved by OLAP queries, as shown in the previous sections, are rather raw, in terms of their presentation. For example, the cube and rollup results which contain sub-group totals may not be presented in a clear and intuitive way for readers to see. OLAP query results containing multiple partitions can be confusing as the data from multiple partitions are combined into one query result report. In short, OLAP queries are not meant for presentation; rather their main job is only to retrieve the required data from the data warehouse. Once the requested data is retrieved, it will be passed on to the *Business Intelligence* (BI) reporting tool to ensure the data is presented professionally for management. This section demonstrates some of the reports or graphs that may be produced by any BI tool, by taking OLAP query results as the input.

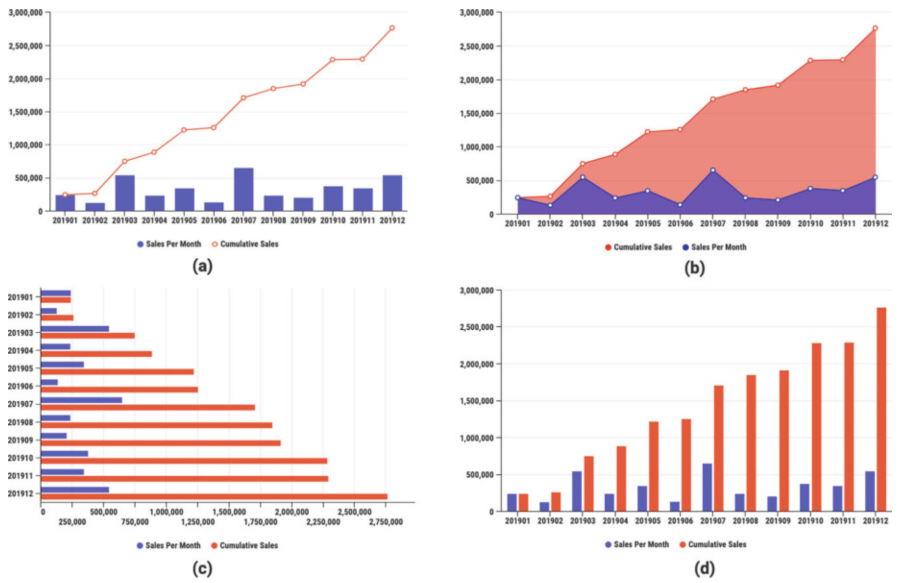

Fig. 19.6 Cumulative Sales in Melbourne

19.6.1 Cumulative and Moving Aggregate

An OLAP cumulative query, which is expressed in the following SQL, may produce Total Sales and Cumulative Sales presented in a graph for a better presentation (rather than in a table, as in Table 19.35). Figure 19.6 shows four different ways to present a cumulative report. Figure 19.6a shows the Monthly Total Sales in a bar graph and the Cumulative Total Sales in a line graph. Looking at this graph, we can see clearly the increase of the Cumulative Sales month by month, and hence the cumulative becomes intuitive. Figure 19.6b shows a different style for presenting a cumulative result, that is, using the line area graph for both Total Sales and its cumulative result. The other two graphs show both Total Sales and the cumulative result in bar graphs (e.g. vertical and horizontal styles). These are four of many possible ways to present a cumulative report. Nevertheless, the cumulative report presents one of these ways, which will be much clearer than in table format.

```
select TimeID, sum(Total_Sales) as Total_Sales,
  sum(sum(Total_Sales)) over (order by TimeID
    rows unbounded preceding) as Cumulative
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and Year = 2019
and LocationID in ('MEL')
group by TimeID;
```

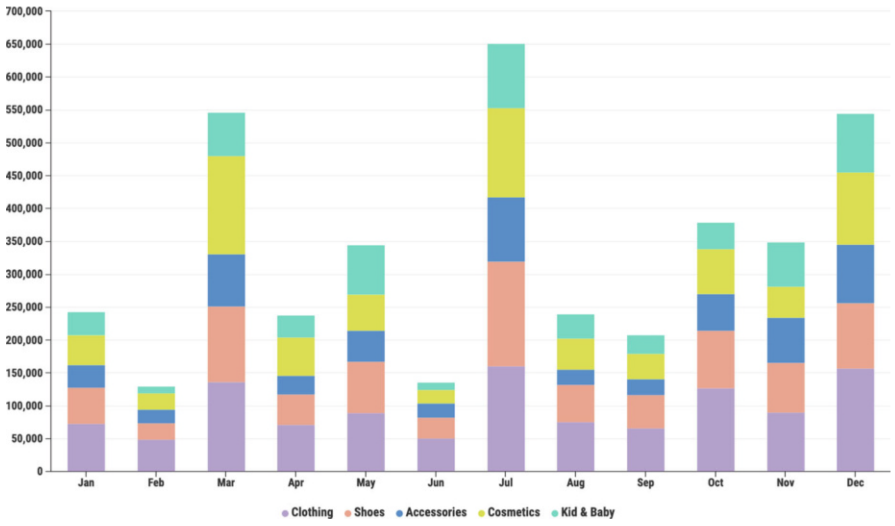

Fig. 19.7 Drilling down into each Product

If you need to drill down into each Product in each month, you can invoke the following OLAP query using simple `group by`, and the results are presented using a stacked bar graph in Fig. 19.7.

```
select S.TimeID, ProductName, sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T, ProductDim P
where S.TimeID = T.TimeID
and S.ProductID = P.ProductID
and Year = 2019
and LocationID in ('MEL')
group by S.TimeID, ProductName;
```

Sometimes, it is necessary to compare the cumulative results of two Locations, for example. Figure 19.8 shows the cumulative result of two Locations, Melbourne and Sydney. We can either invoke two cumulative queries, one for Melbourne and the other for Sydney, and then the graph will combine both results. Alternatively, combining both cumulative queries into one cumulative OLAP query will produce a combined table. Either way, the graph will simply take the results from the OLAP query and present them in a graph, as in Fig. 19.8. Having the cumulative results of two Locations is a good way to compare the performance of two Locations and to see how one Location's cumulative results catches up with the other toward the end of the year.

The last graph in this subsection is a moving aggregate. Figure 19.9 shows the Moving 3-Month Average. Notice the ups in March and July are balanced out through the rather low Total Sales in the respective previous months. The data for

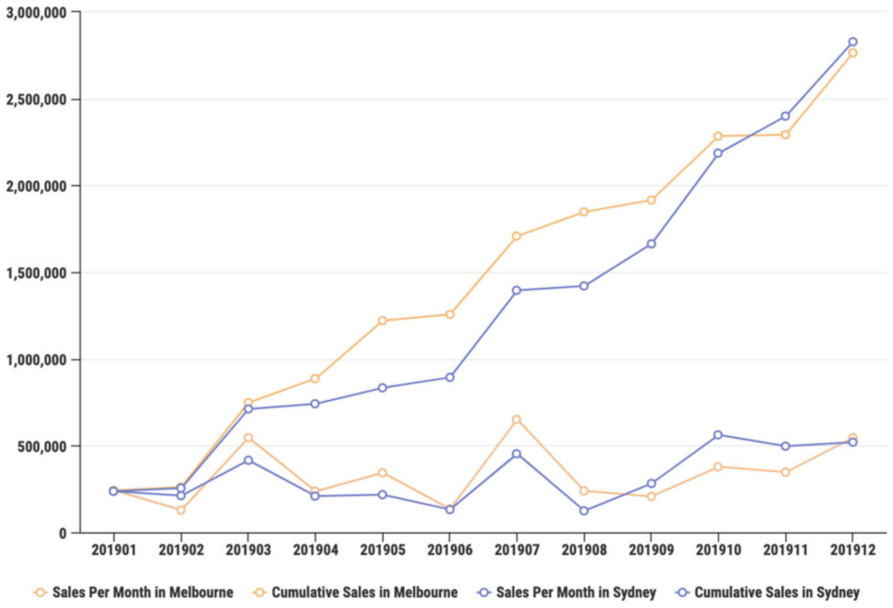

Fig. 19.8 Cumulative Sales of two Locations: Melbourne and Sydney

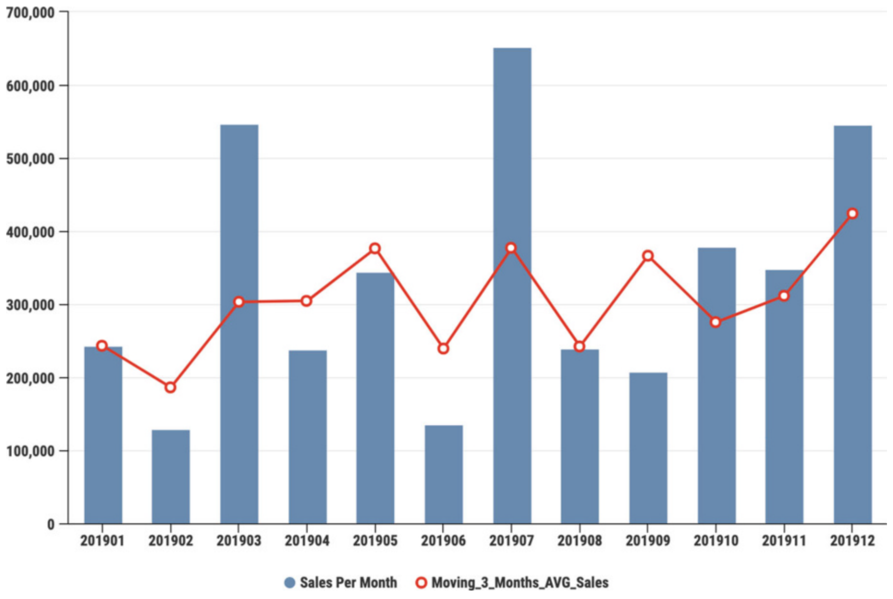

Fig. 19.9 Moving Average 3-Month Sales

this graph are taken from the following Moving 3-Month Average OLAP query:

```
select S.TimeID, sum(Total_Sales) as Total_Sales,
  avg(sum(Total_Sales)) over (order by S.TimeID rows 2 preceding)
    as Avg_3_Months
from SalesFact S, TimeDim T
where S.TimeID = T.TimeID
and Year = 2019
and LocationID in ('MEL')
group by S.TimeID;
```

19.6.2 Ratio

When assessing one particular Location, for example, it is very often that management would like to see how it compares with all the others, that is, finding the percentage contribution of this Location compared to the overall Total Sales of the company. This can easily be seen using a pie chart, as in Fig. 19.10. These two pie charts show the same data using different styles (e.g. 2D or 3D pie chart). This pie chart shows that the Total Sales of Melbourne and Sydney combined contribute to almost two-thirds of all Total Sales of the company.

The following query retrieves the Total Sales of each Location. The BI tool will be able to calculate the percentage of each Location and plot the pie charts.

```
select Location, sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T, LocationDim L
where S.TimeID = T.TimeID
and S.LocationID = L.LocationID
and Year = 2019
group by Location;
```

From each Location, we can drill down into the Product level, as shown in Fig. 19.11. The OLAP query simply adds Product into the `group by` clause.

```
select Location, ProductName, sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T, LocationDim L, ProductDim P
where S.TimeID = T.TimeID
and S.LocationID = L.LocationID
and S.ProductID = P.ProductID
and Year = 2019
group by Location, ProductName
order by Location, ProductName;
```

19.6.3 Ranking

Ranking is one of the most important tools in decision-making, that is, finding the top performing Products or Locations. In OLAP, the `rank() over` function can

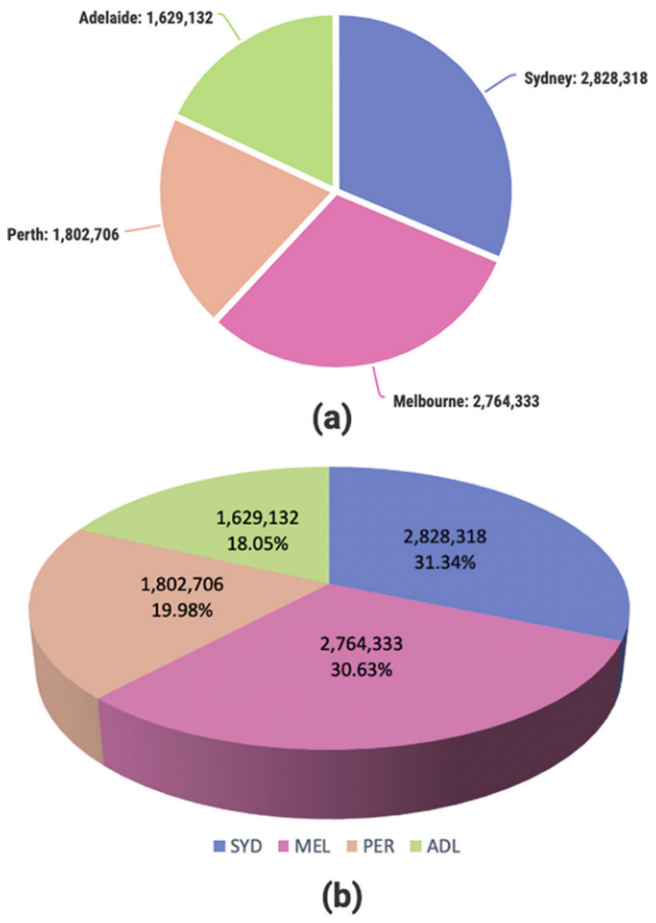

Fig. 19.10 Total Sales ratio in each Location

be used to calculate the ranking. Alternatively, we can simply use `order by` to sort the items in the report. Figure 19.12 shows four different ways of ranking the Products. We can use the normal vertical bar chart as in Fig. 19.12a or a horizontal bar chart as in Fig. 19.12b. Figure 19.12c is rather unusual but may still be used. The pie chart in Fig. 19.12d may not clearly show the ranking, especially when in this case, the first two Products performed approximately equally.

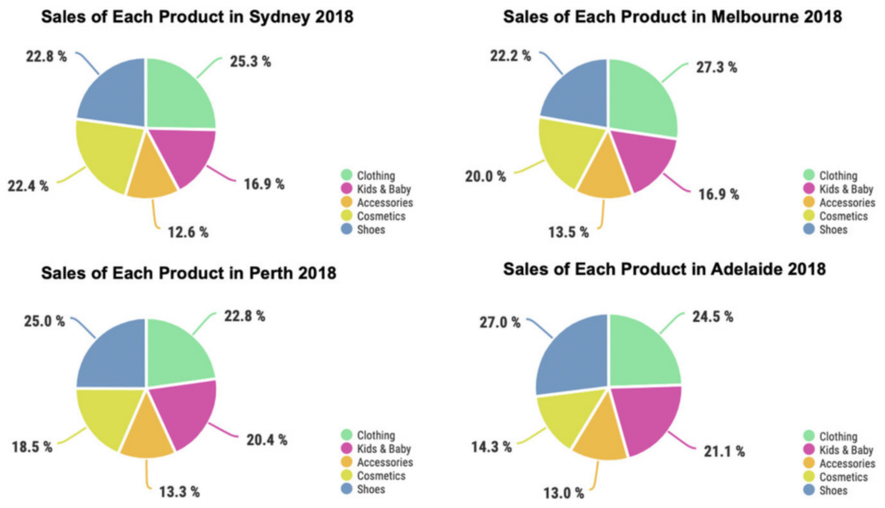

Fig. 19.11 Product Sales performance in various Locations

The OLAP query to retrieve the data for these charts is a simple `group by` query.

```
select ProductName, sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T, ProductDim P
where S.TimeID = T.TimeID
and S.ProductID = P.ProductID
and Year = 2019
group by ProductName
order by ProductName;
```

Another useful ranking is to rank the Locations to see which Locations perform well. This is shown in Fig. 19.13. This chart uses a different style, that is, a half-pie chart. It can be intuitive, depending on the data itself. In this case, the top Location is on the most left, going down to the most right. The query simply uses a similar `group by` clause but grouping and ordering based on Location rather than Product.

```
select Location, sum(Total_Sales) as Total_Sales
from SalesFact S, TimeDim T, LocationDim L
where S.TimeID = T.TimeID
and S.LocationID = L.LocationID
and Year = 2019
group by Location
order by Location;
```

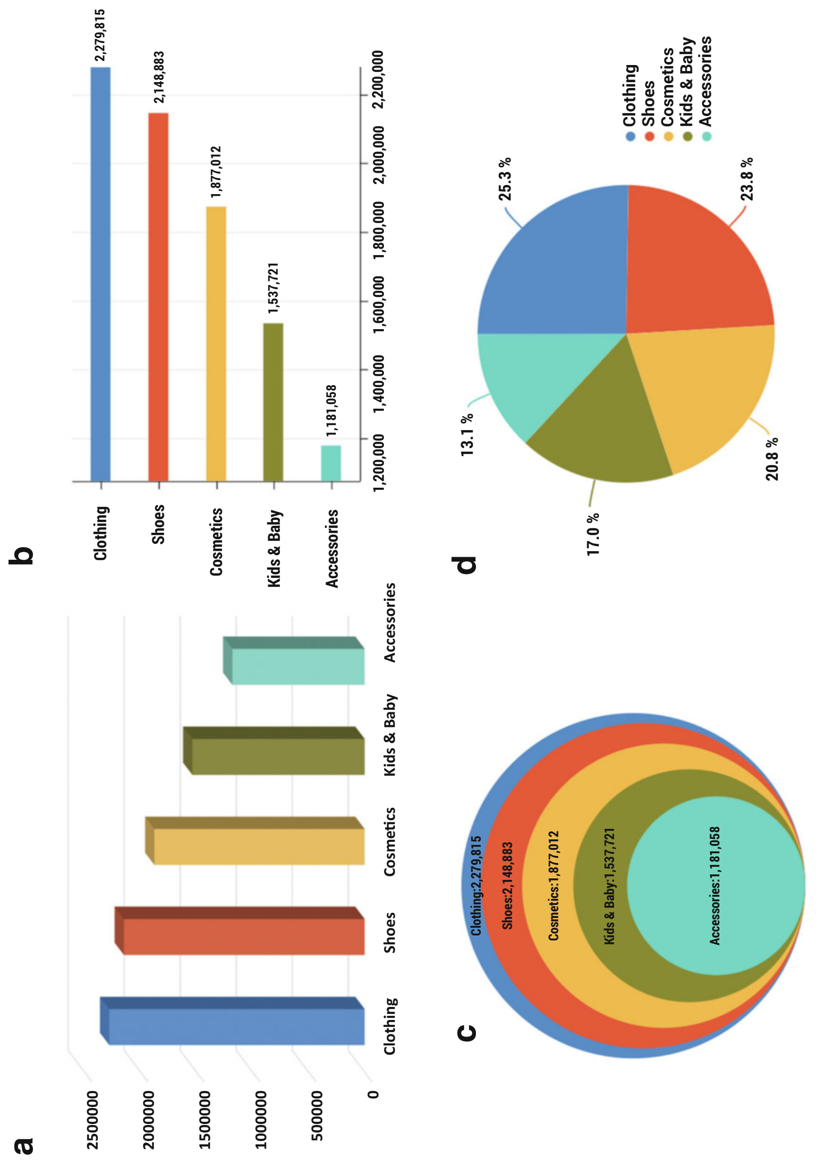

Fig. 19.12 Product ranking

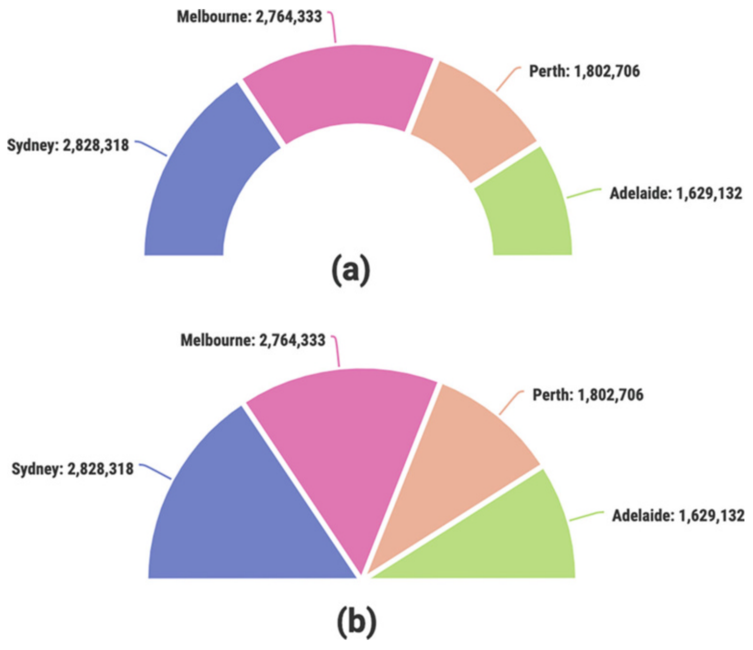

Fig. 19.13 Location ranking

19.6.4 A More Complete Report

A typical BI report combines many graphs and figures in various presentations to give management a more complete view of the business operation. Figure 19.14 shows the performance of various products (e.g. Clothing, Shoes, etc.), in the past year, as well as their respective cumulative figures. Some total figures are also given in the report. This report requires multiple OLAP queries to provide the report with adequate data.

The report (and all other reports shown in the previous subsections) does not have to have its parameters (e.g. which Location, or which Product, or which Year) statically embedded into the query. Rather, the parameters should be specified dynamically during runtime by the user, for example, the user may want to specify a certain Year of data for the report, etc. So, it is a dynamic BI report rather than a static one.

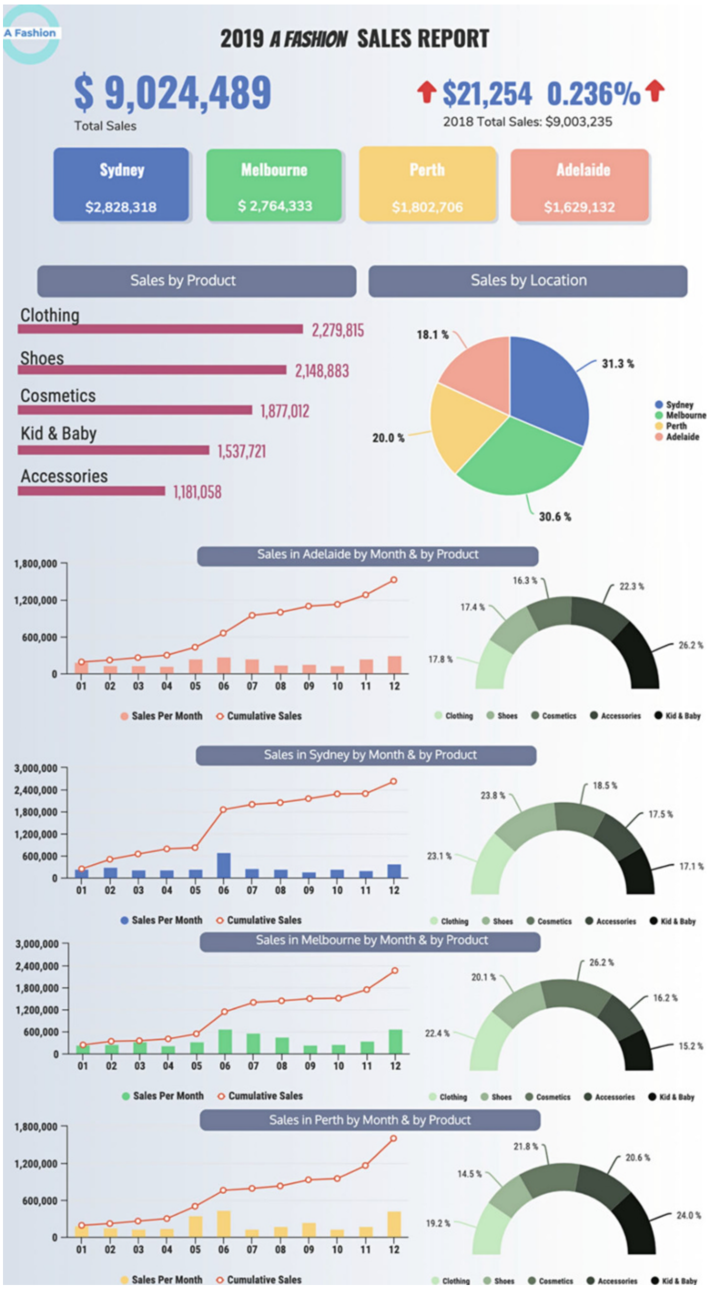

Fig. 19.14 A more complete BI report

19.7 Summary

This chapter mainly focuses on OLAP, which is the SQL query to retrieve data from the data warehouse. The retrieved data will then be formatted by the Business Intelligence Reporting tool. The OLAP queries discussed in this chapter are divided into several categories:

(a) Basic aggregate functions: `count`, `sum`, `avg`, `max` and `min`. The `group by` clause is often used in conjunction with these basic aggregate functions.
(b) Cube and Rollup: `group by cube` and `group by rollup`. The simple formatting of the query results can be enhanced through the `decode` and `grouping` functions.
(c) Ranking and Partition: `rank() over` and `dense_rank() over` functions. The `row_number() over` function has some similarities (as well as differences) to the ranking functions. The `partition` clause in the ranking function can be used to partition the dataset, each with its own ranking.
(d) Top-N and Top-Percentage Ranking: use of nested queries to retrieve Top-N and `percent_rank` function to retrieve Top-Percentage rankings.
(e) Cumulative and Moving Aggregate: `row unbounded preceding` or `row` *n* `proceeding` can be used to get the cumulative or moving aggregate values.

Finally, the last section in this chapter gives an illustration as to how the Business Intelligence reporting tool may take the data retrieved from the OLAP queries and present it in a visual way, which may assist decision-makers in understanding the data from the data warehouse.

19.8 Exercises

19.1 Using the Sales Data Warehouse case study presented in the chapter, write the SQL commands for the following OLAP queries:

(a) Retrieve the Product that has the lowest Total Sales in 2019.
(b) Retrieve the Percent Rank of the Product that has the lowest Total Sales in 2019.

19.2 Given the star schema in Fig. 19.15, write the SQL commands for the following OLAP queries:

(a) Display the Top-10 average property prices by suburb.
(b) Display the average price of properties by property-type description and suburb. It is not necessary to show the sub-totals or group totals or grand total.

19.3 Given the star schema in Fig. 19.16, write the SQL commands for the following OLAP queries. The Sales Method Dimension table lists all kinds of sales methods, such as In-Store, Online, Phone Order, etc.

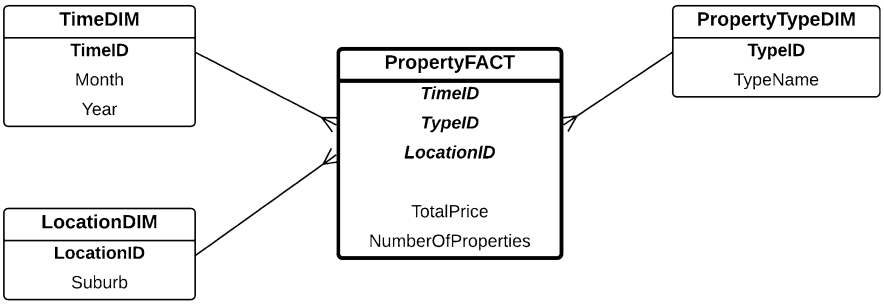

Fig. 19.15 Property Sales star schema

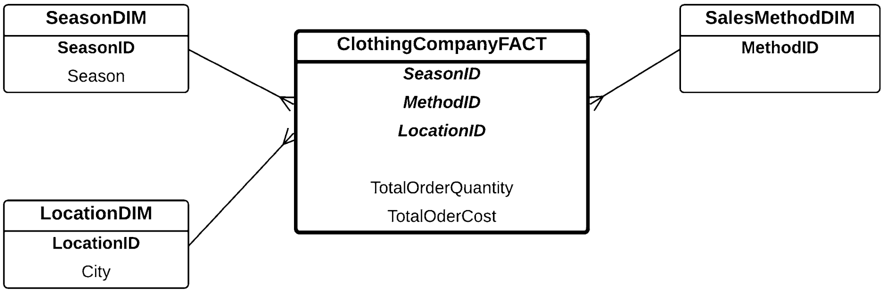

Fig. 19.16 Clothing Company star schema

(a) Perform a Cumulative Sum of Total Order Cost of all Online orders (Instruction: use all dimensions). Note that for each City, there should be a separate Cumulative Sum.
(b) Perform another Cumulative Sum of Total Order Cost but partitioned based on the SalesMethodID: one partition for Online orders, one partition for In-Store orders and one partition for Phone Order orders. Hints: It must also be partitioned based on Location (or Suburb). Hence, for example, Online orders for Melbourne will have one set of cumulative Total Order Cost, whereas Online orders for Sydney will have a separate set of cumulative Total Order Cost.
(c) Show the total order costs of each source order and rank them.
(d) Display the source order that generates the highest total order cost.

19.9 Further Readings

SQL is the basis for the OLAP implementation. OLAP queries usually use various OLAP functions, such as `cube`, `rollup`, etc., as well as ranking functions and aggregate functions (e.g. moving aggregates). Most database textbooks cover SQL in various depth, such as [1–10]. Specialized resources on SQL are as follows: [11–14].

Recent work on OLAP covers missing data [15], probabilistic data [16, 17], interval data [18], processing and optimization [19, 20], parallel processing [21–23] and spatial OLAP [24, 25].

In our book, the BI part is mainly used for presenting the OLAP results in a professional way, such as using graphs, charts and reports. The field of BI actually covers a wide range of topics, such as business requirements, decision-support systems and data integration and analytics. Further readings on BI can be found in the following books: [26–32].

References

1. C. Coronel, S. Morris, *Database Systems: Design, Implementation, and Management* (Cengage Learning, New York, 2018)
2. T. Connolly, C. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management* (Pearson Education, New York, 2015)
3. J.A. Hoffer, F.R. McFadden, M.B. Prescott, *Modern Database Management* (Prentice Hall, Englewood Cliffs, 2002)
4. A. Silberschatz, H.F. Korth, S. Sudarshan, *Database System Concepts, Seventh Edition* (McGraw-Hill, New York, 2020)
5. R. Ramakrishnan, J. Gehrke, *Database Management Systems*, 3rd edn. (McGraw-Hill, New York, 2003)
6. J.D. Ullman, J. Widom, *A First Course in Database Systems*, 2nd edn. (Prentice Hall, Englewood Cliffs, 2002)
7. H. Garcia-Molina, J.D. Ullman, J. Widom, *Database Systems: The Complete Book* (Pearson Education, New York, 2011)
8. P.E. O'Neil, E.J. O'Neil, *Database: Principles, Programming, and Performance*, 2nd edn. (Morgan Kaufmann, Los Altos, 2000)
9. R. Elmasri, S.B. Navathe, *Fundamentals of Database Systems*, 3rd edn. (Addison-Wesley-Longman, Reading, 2000)
10. C.J. Date, *An Introduction to Database Systems*, 7th edn. (Addison-Wesley-Longman, Reading, 2000)
11. J. Melton, *Understanding the New SQL: A Complete Guide, Second Edition*, vol I (Morgan Kaufmann, Los Altos, 2000)
12. C.J. Date, *SQL and Relational Theory—How to Write Accurate SQL Code*, 2nd edn. Theory in practice (O'Reilly, New York, 2012)
13. A. Beaulieu, *Learning SQL: Master SQL Fundamentals* (O'Reilly Media, New York, 2009)
14. M.J. Donahoo, G.D. Speegle, *SQL: Practical Guide for Developers*. The Practical Guides (Elsevier Science, Amsterdam, 2010)
15. M.B. Kraiem, K. Khrouf, J. Feki, F. Ravat, O. Teste, New OLAP operators for missing data, in *Actes des 13èmes journées francophones sur les Entrepôts de Données et l'Analyse en Ligne, Business Intelligence and Big Data, EDA 2017, Lyon, France, 3-5 mai 2017*, ed. by O. Boussaïd, F. Bentayeb, J. Darmont. RNTI, vol. B-13, pp. 53–66 (Éditions RNTI, 2017)
16. X. Xie, X. Hao, T.B. Pedersen, P. Jin, J. Chen, OLAP over probabilistic data cubes I: aggregating, materializing, and querying, in *Proceedings of the 32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016* (IEEE Computer Society, New York, 2016), pp. 799–810